UNIVERSIDADE FEDERAL DE SANTA CATARINA

AVALIAÇÃO DE REDES NEURAIS PARA A GERAÇÃO DE IMAGENS

Rafael Rabello Moser

Florianópolis - SC
2018 / 2

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

AVALIAÇÃO DE REDES NEURAIS PARA A GERAÇÃO DE IMAGENS

Rafael Rabello Moser

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação

Florianópolis - SC
2018 / 2

Rafael Rabello Moser

AVALIAÇÃO DE REDES NEURAIS PARA A GERAÇÃO DE IMAGENS

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação

---

Prof. Elder Rizzon Santos, Dr.
Universidade Federal de Santa Catarina

Banca Examinadora

Prof. Mauro Roisenberg, Dr.
Universidade Federal de Santa Catarina

Thiago Ângelo Gelaim, Msc.
Universidade Federal de Santa Catarina

RESUMO

Estúdios e equipes de desenvolvimento independentes tem ficado cada vez mais comuns nos últimos anos, com muitos destes se deparando com bastante sucesso. Em função do aumento da acessibilidade a computadores, dispositivos e à Internet, a prospecção de ser um desenvolvedor independente tem se mostrado uma idéia cada vez mais atrativa. Muitas vezes, essas equipes ou estúdios são compostos por apenas uma pessoa - esta sendo, geralmente, um(a) programador(a). Com restrições de recursos, nem sempre se pode pagar por profissionais de outras áreas, tais como arte e música, para auxiliar na criação de jogos. Soluções comuns para isso são motores de jogos (do inglês, *game engines*), que facilitam a criação de componentes de jogos. Mesmo assim, ainda sobram muitas atividades que o desenvolvedor deve fazer manualmente, especialmente no que diz respeito à arte do jogo.

Programas capazes de automatizar tais processos manuais e demorados certamente se fazem ser extremamente úteis, uma vez que reduz consideravelmente o tempo e dinheiro necessários para a execução das mesmas. O objetivo deste trabalho foi de realizar uma análise comparativa de algumas abordagens com redes neurais para a geração de imagens para que, futuramente, possa ser criada uma ferramenta baseada na abordagem mais apropriada encontrada que resolva, pelo menos em parte, o problema de reduzir o custo de criação de alguns dos componentes de arte de jogos, mais especificamente de objetos individuais (i.e.: árvores, animais, pessoas/rostos, etc).

Para tal, foi realizada uma comparação entre redes recorrentes, convolucionais e redes generativas adversariais, na forma de auto-codificadores, em cima do conjunto MNIST e, no caso da convolucional, em cima do conjunto Cifar10, para testar a viabilidade de se utilizar estas redes para resolver este problema. Através da aplicação de técnicas de aprendizagem de máquina e redes neurais, ao final dos experimentos realizados com algumas arquiteturas diferentes de redes neurais, foi demonstrado que estas conseguem ser ferramentas poderosas na tarefa de geração automatizada de imagens.

Palavras-chave: inteligência artificial, redes neurais, aprendizagem de máquina, geração procedural, desenvolvimento de jogos, criatividade computacional

ABSTRACT

Independent studios and development teams have become increasingly common in recent years, with many of these finding themselves quite successful. Due to the increased accessibility to computers, devices and the Internet, the prospect of being an independent developer has been an increasingly attractive idea. Often, these teams or studios consist of only one person - this is usually a programmer. With resource constraints, these developers can not always afford professionals from other areas, such as art and music, to aid in the creation of games. Common solutions to this are game engines, which make it easy to create game components. Even so, there are still many activities that the developer must do manually, especially with regard to the art of the game.

Programs that automate such time-consuming and manual processes are certainly extremely useful as they greatly reduce the time and money required to execute them. The goal of this work was to perform a comparative analysis of some approaches with neural networks for the generation of images so that, in the future, a tool based on the most appropriate approach found can be created that solves, at least in part, the problem of reducing the cost to create some of the game art components, more specifically individual, physical objects (ie: trees, animals, people / faces, etc).

For this purpose, a comparison was made between recurrent, convolutional and generative adversarial networks, in the form of autocoder, on top of the MNIST and, in the convolutional case, on the Cifar10 datasets, to test the feasibility of using these networks to solve this problem. Through the application of machine learning techniques and neural networks, at the end of the experiments performed with the different chosen neural network architectures, it has been demonstrated that these can be powerful tools in the task of automated image generation.

Keywords: artificial intelligence, neural networks, machine learning, procedural generation, game development, computational creativity

LISTA DE ABREVIAÇÕES E SIGLAS

IA	Inteligência Artificial
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
GAN	Generative Adversarial Network
ML	Machine Learning
GPC	Geração Procedural de Conteúdo
CC	Criatividade Computacional
DBN	Deep Belief Network
RBM	Restricted Boltzmann Machine
AE	Autoencoder
VAE	Variational Autoencoder
DRAW	Deep Recurrent Attentive Writer
DCGAN	Deep Convolutional Generative Adversarial Network
MNIST	Modified National Institute of Standards and Technology
CIFAR10	Canadian Institute For Advanced Research

SUMÁRIO

INTRODUÇÃO 11

OBJETIVOS 12

METODOLOGIA DE PESQUISA 12

FUNDAMENTAÇÃO 13

CRIATIVIDADE 13

CRIATIVIDADE COMPUTACIONAL 14

GERAÇÃO PROCEDURAL DE CONTEÚDO 14

REDES NEURAIS 16

ESTRATÉGIAS DE TREINAMENTO 17

CARACTERÍSTICAS ADICIONAIS A SEREM CONSIDERADAS 18

REDES NEURAIS CONVOLUCIONAIS 21

MÁQUINAS RESTRITAS DE BOLTZMANN 24

AUTO-CODIFICADORES 25

TRABALHOS RELACIONADOS 26

DRAW: UMA REDE NEURAL RECORRENTE PARA A GERAÇÃO DE IMAGENS 27

APRENDIZAGEM DE REPRESENTAÇÃO NÃO-SUPERVISIONADA COM REDES GENERATIVAS ADVERSARIAIS CONVOLUCIONAIS PROFUNDAS 31

GERAÇÃO CONDICIONAL DE IMAGENS COM DECODIFICADORES PixelCNN 35

OTIMIZANDO REDES NEURAIS PARA A GERAÇÃO DE IMAGENS 38

AVALIAÇÃO DE REDES NEURAIS PARA O DESENVOLVIMENTO DA FERRAMENTA 42

PRÉ-PROCESSAMENTO E ORGANIZAÇÃO DOS DADOS 46

AS REDES 48

GERAÇÃO DE AMOSTRAS 49

DESCRIÇÃO DO EXPERIMENTO 49

ANÁLISE DOS RESULTADOS 54

CONCLUSÃO DOS RESULTADOS 71

CONCLUSÃO E TRABALHOS FUTUROS 72

REFERÊNCIAS 74

LISTA DE FIGURAS

Figura 1	**21**
Figura 2	**21**
Figura 3	**22**
Figura 4	**22**
Figura 5	**23**
Figura 6	**25**
Figura 7	**28**
Figura 8	**28**
Figura 9	**29**
Figura 10	**30**
Figura 11	**30**
Figura 12	**33**
Figura 13	**33**
Figura 14	**33**
Figura 15	**34**
Figura 16	**36**
Figura 17	**36**
Figura 18	**36**
Figura 19	**38**
Figura 20	**39**
Figura 21	**44**
Figura 22	**46**
Figura 23	**48**
Figura 24	**55**
Figura 25	**59**
Figura 26	**61**
Figura 27	**62**
Figura 28	**63**
Figura 29	**64**
Figura 30	**65**
Figura 31	**68**
Figura 32	**70**

LISTA DE TABELAS

Tabela 1	**27**
Tabela 2	**52**
Tabela 3	**53**
Tabela 4	**53**
Tabela 5	**55**
Tabela 6	**56**
Tabela 7	**56**
Tabela 8	**59**
Tabela 9	**61**
Tabela 10	**61**
Tabela 11	**66**
Tabela 12	**66**
Tabela 13	**69**
Tabela 14	**70**
Tabela 15	**70**

1. INTRODUÇÃO

Inteligência Artificial (IA) é, como um campo da computação, definida como sendo a ciência e a engenharia envolvidas em criar máquinas, especialmente programas, que exibem inteligência (MCCARTHY *et al.*, 1956). Geralmente, o termo é usado quando uma máquina imita funções cognitivas que humanos tipicamente associam a outros humanos, tais como resolução de problemas e aprendizagem. Campo fundado e termo cunhado em 1956, em uma conferência na Faculdade de Dartmouth, inúmeros programas rapidamente surgiram para demonstrar as capacidades que essa área prometia, ganhando, inclusive, bastante investimento do Departamento de Defesa dos Estados Unidos, na década de 60. Entre as façanhas mais comuns de computadores considerados inteligentes na época, estavam a de vencer no jogo de damas, resolvendo problema algébricos e demonstrando alguns teoremas. Apesar do entusiasmo, os programas da época não conseguiam resolver alguns problemas mais difíceis, resultando, em meados da década de 70, na perda de investimentos, causando uma queda na pesquisa na área.

Foi no começo da década de 80 que o interesse pela área reacendeu, com a popularização da abordagem conexionista. Uma das estratégias que se popularizou consideravelmente, dentro do contexto dessa abordagem, é a das Redes Neurais Artificiais ou, mais simplesmente, Redes Neurais (RN). Estas consistem em uma abordagem fortemente inspirada na organização estrutural e funcional dos sistemas nervosos de animais - com neurônios e suas conexões (RUMELHART; MCCLELLAND, 1986).

O principal objetivo das redes neurais é de resolver problemas de uma forma semelhante à do cérebro humano, apesar de, mesmo as redes neurais mais modernas, trabalharem com quantidades de unidades ordens de magnitude abaixo daquelas de um cérebro. Ainda assim, já mostra-se extremamente eficiente e eficaz na área de aprendizagem de máquina - sistemas capazes de aprender com dados - sendo aplicadas na resolução de múltiplos problemas, tais como visão computacional e reconhecimento de padrões, coisas tipicamente difíceis de serem resolvidas com abordagens mais tradicionais de programação (BISHOP, 1995), baseadas em regras.

A inteligência artificial tem sido aplicada em jogos desde o seu nascimento, como uma área de pesquisa. Aparecia bastante, por exemplo, na forma de adversários, como no jogo de Nim, em 1951 (GRANT, 1952), e xadrez, também em 1951 (COPELAND, 2000). Na década de 80, começou a ser aplicada à geração procedural (ou automatizado) de conteúdo (GPC) e, recentemente, ganhou espaço na área acadêmica (TOGELIUS, *et al.*, 2011). GPC consiste em um espaço de soluções de design ser explorado para elaborar novas variações. A prática mais comum é de um usuário fornecer conteúdo pronto, o qual é analisado pelo gerador, e este cria conteúdo novo em um estilo semelhante (TOGELIUS, *et al.*, 2016).

Já existe um rico ecossistema de ferramentas que permitem a criação de muitos dos componentes de um jogo - tais como arte visual, música, entre outros - proceduralmente. Exemplos destas incluem ferramentas como a *Unity* (disponível em https://unity3d.com), *Unreal Engine* (disponível em https://www.unrealengine.com) e *Amazon Lumberyard* (disponível em https://aws.amazon.com/lumberyard). Tais ferramentas já reduzem consideravelmente os custos de se criar um jogo, mas, geralmente, exigem que o usuário possua considerável conhecimento dos aspectos técnicos das mesmas (POWLEY, *et al.*, 2016). Com o intuito de reduzir ainda mais os custos (tempo, dinheiro e conhecimento técnico) envolvidos na criação de componentes de jogos, o trabalho visa elaborar uma solução que facilite a criação

destes componentes - mais especificamente, os componentes gráficos. RNs, junto com aprendizagem de máquina, já se mostraram muito úteis para o reconhecimento e criação de componentes visuais/gráficos (OORD, *et al*., 2016a; OORD, *et al*., 2016b; TIELEMAN, 2014; THEIS & BETHGE, 2015), portanto, foi a abordagem escolhida como base para a resolução do problema proposto.

1.1. OBJETIVOS

1.1.1. Objetivo Geral

O principal objetivo deste trabalho é desenvolver uma solução computacional, utilizando técnicas de aprendizagem de máquina, para a geração de componentes gráficos para jogos.

1.1.2. Objetivos Específicos

Para alcançar o objetivo principal deste trabalho, foram traçados os seguintes objetivos específicos:

1. Identificar e analisar as atuais técnicas de redes neurais e aprendizagem de máquina aplicadas à geração de imagens;
2. O componente de geração (rede neural) deverá ser capaz de gerar uma imagem de um objeto, em 2D, como sugestão para o usuário;
3. Especificar os testes necessários para avaliar a eficácia e eficiência das abordagens analisadas em 1;
4. Avaliar os resultados obtidos em 3 e identificar melhorias e/ou abordagens alternativas para apoiar o desenvolvimento da ferramenta desejada;

1.2. METODOLOGIA DE PESQUISA

Inicialmente, será realizado um levantamento de técnicas de RN e AM relevantes ao problema abordado pelo trabalho. Este levantamento se dará através do estudo e leitura de obras relevantes ao trabalho, a serem definidas pelo autor, juntamente com a orientação e sugestões do orientador. Ao final desta etapa, as abordagens e estudos julgados mais promissores serão escolhidos para servirem como base para a etapa de desenvolvimento.

Em seguida, um levantamento de ferramentas para a aplicação de aprendizagem de máquina já existentes, será realizado. No caso de nenhuma ferramenta apropriada ser encontrada, será feito um estudo dos artefatos necessários a serem desenvolvidos, seguido de desenvolvimento dos mesmos. A ferramenta, para a conclusão deste trabalho, deverá depender apenas de bibliotecas e frameworks com código aberto e livre. Ao final desta etapa a ferramenta deverá poder, a partir de uma imagem fornecida(e após o devido treinamento), detectar a presença de e identificar o objeto desejado. Adicionalmente, serão realizados testes para determinar como e onde realizar correções e/ou otimização da mesma.

Uma vez obtidos resultados satisfatórios com a ferramenta de aprendizagem, o trabalho prosseguirá com o levantamento de ferramentas existentes de geração de componentes gráficos - e as técnicas que estas usam - para avaliar possíveis abordagens para o desenvolvimento da ferramenta de geração. Em seguida, realizado o desenvolvimento da ferramenta de geração de componentes gráficos. Esta deverá

fazer uso da aprendizagem realizada pela ferramenta de aprendizagem e gerar sugestões de componentes gráficos dos objetos desejados.

Como objetivo inicial do trabalho (no contexto da ferramenta de geração), a saída desejada da ferramenta será na forma de uma imagem, contendo uma instância do objeto desejado em 2D. Será, então, avaliada a precisão com a qual a ferramenta consegue gerar o objeto desejado, determinada pela frequência com a qual o usuário aceita ou não a imagem gerada. Também será levado em consideração a velocidade com a qual a ferramenta é capaz de elaborar uma sugestão. Adicionalmente, o estudo das ferramentas escolhidas no início desta etapa também servirá para comparar com os resultados obtidos a partir dos testes da solução proposta. Semelhante à ferramenta de aprendizagem, os testes serão utilizados para determinar como e onde realizar as devidas correções e/ou otimizações.

Os resultados dos testes de ambas as ferramentas servirão tanto para avaliar pontos de melhoria das mesmas, quanto como referência para comparar com ferramentas semelhantes. Assim, poderá ser avaliado a qualidade do que foi desenvolvido e se os critérios de aceite foram atendidos.

2. FUNDAMENTAÇÃO

2.1. CRIATIVIDADE

Criatividade se manifesta em inúmeras áreas, tais como arte, invenção de objetos e idéias ou conceitos novos. De um modo geral, é difícil definir precisamente o que, de fato, é criatividade.

“*There are as many research-based definitions of creativity as there are approaches to studying it*.” (BINK & MARSH, 2001)

Apesar disso, alguns autores convergem para definições semelhantes. Criatividade é definida por STERNBERG (1998) como “a capacidade de criar uma solução que é tanto original quanto apropriada”. Boden (1991) define criatividade, de forma semelhante, como a “habilidade de elaborar idéias ou artefatos que são novos, surpreendentes e valiosas”. Entende-se por “idéias” como sendo conceitos, poemas, composições musicais, teorias, etc., e “artefatos” como sendo esculturas, motores, máquinas, etc. Adicionalmente, Boden afirma que a criatividade é “uma questão de usar seus recursos computacionais para explorar e, às vezes ir além de, espaços conceituais familiares”. Boden ainda separa criatividade de em duas dimensões:

- Criatividade histórica (**H-creativity**) vs. criatividade pessoal ou psicológica (**P-creativity**), para quando a idéia é novidade para a humanidade como um todo ou se é novidade apenas para o agente que gerou a idéia, respectivamente;
- Criatividade exploratória (**E-creativity**) vs. criatividade transformacional (**T-creativity**), que são baseadas na exploração de espaços conceituais apropriados para a tarefa e quando é possível alterar as regras utilizadas para definir esse espaços conceituais, respectivamente.

No contexto deste trabalho, em função das ferramentas e da natureza das abordagens utilizadas, será dado mais ênfase à criatividade exploratória.

2.2. CRIATIVIDADE COMPUTACIONAL

A criatividade computacional é um campo de pesquisa dentro da Inteligência Artificial que visa construir e trabalhar com sistemas computacionais que criam artefatos e idéias (COLTON & WIGGINS, 2012). Existe uma importante distinção a ser feita entre as abordagens de pesquisa de IA e as especificamente da área de criatividade computacional: grande parte das pesquisas e projetos realizados com IA tradicional operam em um paradigma baseado em *resolver um problema*. Dependendo do processo necessário para a resolução do problema, pode-se modelar o problema de tal modo que possamos resolvê-lo com algo como prova de teorema automatizada, ou até aprendizagem de máquina, por exemplo.

Por outro lado, não parece ser apropriado tratar uma composição musical ou a produção de um desenho como um problema a ser resolvido (COLTON & WIGGINS, 2012). Colton & Wiggins (2012) afirmam que o preferível, na área de criatividade computacional, é operar em um paradigma de *geração de artefatos*, onde a automação de uma tarefa inteligente é vista como uma oportunidade de produzir algo de valor cultural.

A criatividade geralmente envolve a descoberta de novos conhecimentos, portanto, trabalhos em abordagens que alcançam esse objetivo automaticamente são relevantes (DUCH, Włodzisław, 2006). Um exemplo disso é um modelo heurístico de uma inteligência artificial baseada em busca para modelar processos criativos que levaram a descobertas científicas históricas (LANGLEY et al., 1987). Apesar de apresentar modelos que diferem do que foi gerado por especialistas nos campos testados (por exemplo: astronomia, química, etc.) (LANGLEY et al., 1987; LANGLEY & JONES, 1988; KOCABAS & LANGLEY, 2001), a IA demonstrou ser capaz de gerar alguns modelos bastante interessantes.

Outros modelos computacionais interessantes que abordam a criatividade são Copycat, Metacat e Magnificat (MITCHELL, 1993; HOFSTADTER, 1995; REHLING, 2001). Esses modelos funcionam de um modo sensível ao contexto, ainda que flexível, para que possam realizar desafios como, por exemplo, com exemplos de letras do alfabeto em um determinado estilo artístico, gerar as letras restantes do alfabeto em um estilo semelhante.

2.3. GERAÇÃO PROCEDURAL DE CONTEÚDO

No contexto de jogos, geração procedural de conteúdo (GPC), pode ser definida como a “criação automática e algorítmica de conteúdo de jogos com interação limitada ou indireta do usuário.” (TOGELIUS et al., 2011a). Conteúdo, nesse caso, refere-se a artefatos do jogo, tais como: fases, mapas, texturas, itens, música, etc. Para alguns, GPC pode ser considerada um subconjunto de uma abordagem mais ampla, que é a de “métodos generativos” (COMPTON et al., 2013). Uma das diferenças principais entre GPC e outros métodos generativos é que GPC precisa levar em consideração o contexto e as restrições do jogo que está sendo gerado, enquanto que outros métodos generativos não possuem, necessariamente, requisitos (TOGELIUS et al., 2016). Togelius *et al.* (2016) citam alguns exemplos do quê que se enquadra em PCG:

- um sistema capaz de criar armas novas, em um jogo de tiro, em resposta às ações coletivas do jogadores, de tal forma que o jogador é apresentado com

versões melhoradas de armas que os jogadores demonstraram ter gostado de usar;
- um programa capaz de criar, por conta própria, jogos de tabuleiro completos, jogáveis e balanceados, tomando jogos de tabuleiros existentes como ponto de partida;
- uma ferramenta que rapidamente popula um mundo com vegetação.

Por outro lado, também são citados exemplos do quê *não* se encaixa em GPC:

- um jogador artificial para um jogo de tabuleiro;
- um editor de mapas que permite ao usuário inserir e remover objetos, sem realizar nenhuma geração por conta própria.

Hendrix *et al.* (2013) propõe um sistema de classificação do *conteúdo* gerado por técnicas de GPC:

- **Game Bits** - unidades elementares do jogo que não afetam a jogabilidade quando consideradas independentemente, tais como: estruturas, vegetação e sons;
- **Game Space** - representa os ambientes jogáveis. Consiste de várias unidades de Game Bits. Entram nessa categoria: mapas internos (prédios, cavernas, etc), mapas externos (campos, florestas), entre outros;
- **Game System** - planejamento urbano, redes de estradas, regras de interação entre elementos de um ambiente, etc;
- **Game Scenarios** - nessa categoria entram conceitos como eventos, que, junto com mapas, fazem surgir o conceito de fases;
- **Game Design** - engloba regras de jogabilidade e objetivos.

Para este trabalho a categoria de interesse é a de Game Bits.

Abordagens de PCG tipicamente se enquadram em uma de três categorias: construtiva, gerar-e-testar (do inglês *generate-and-test*) e baseado em busca (TOGELIUS et al., 2011b), esta última sendo a mais popular, atualmente (CHEN & ROBERTS, 2013).

Na abordagem construtiva, o algoritmo gera conteúdo em apenas uma passada de modo previsível e, tipicamente, em pouco tempo. Muitos algoritmos PCG são do tipo construtivo, tais como Perlin noise e L-sistema (SHAKER, M. et al., 2015). Essa abordagem é comumente utilizada para a geração de componentes “decorativos” como, por exemplo, plantas. Para este tipo de tarefa, abordagens construtivas funcionam bem o suficiente, mas encontram dificuldades quando o conteúdo gerado está sujeito a regras e restrições de jogabilidade (SHAKER, M. et al., 2015).

É possível superar as limitações das abordagens construtivas incorporando um mecanismo de teste para avaliar se o resultado do algoritmo está em conforme dos requisitos. Abordagens do tipo gerar-e-testar incorporam tanto um mecanismo de gerar conteúdo quanto um mecanismo de testar o mesmo (TOGELIUS, J. et al., 2011b). Tipicamente, nas abordagens gerar-e-testar, se o conteúdo gerado é rejeitado na avaliação, este é descartado e o processo de geração é executado novamente.

PCG baseado em busca, apesar de tratada separadamente, pode ser considerada um caso especial da abordagem gerar-e-testar e difere-se da seguinte forma, do típico gerar-e-testar:

- a função de avaliação não descarta nem aceita o conteúdo gerado, mas, ao invés disso, atribui uma nota ao conteúdo. Tal função pode ser chamada de função de avaliação ou utilidade (do inglês *fitness function* ou *utility function*);
- a geração de um novo "conteúdo candidato" é fortemente influenciado pelo valor atribuído ao conteúdo anterior, permitindo, assim, direcionar as iterações de conteúdo para obterem valores cada vez mais altos.

Várias técnicas diferentes de PCG já foram usadas com sucesso, como modelos de Markov para a geração de mapas 2D *platformers* (i.e. Super Mario Bros.) (SNODGRASS, 2016), algoritmos genéticos para a geração de terrenos, algoritmo de menor caminho anisotrópico para a geração de redes de ruas (DE CARLI et al., 2011), algoritmos evolucionários para a geração de mapas para jogos de estratégia (TOGELIUS et al., 2011b) e redes neurais para a geração de imagens (OORD et al., 2016a; OORD et al., 2016b; TIELEMAN, 2014).

Neste trabalho, serão exploradas tr^s abordagens que fazem uso de redes neurais, mais especificamente, redes neurais convolucionais (CNN, do inglês *convolutional neural network*), redes recorrentes (RNN, do inglês *recurrent neural network*), na forma de LSTMs (Long Short-Term Memory) e redes generativas adversariais convolucionais. Detalhes sobre estas abordagens serão explorados mais adiante.

2.4. REDES NEURAIS

Redes neurais profundas tem recebido cada vez mais atenção da comunidade de aprendizagem de máquina nos últimos anos. Estas são caracterizadas por várias unidades, agrupadas em camadas, com as camadas mais baixas estando conectadas aos dados de entrada e as mais altas e intermediárias apenas indiretamente conectadas às entradas, sendo reservadas para processamento. De acordo com Tieleman (2014), tais redes têm demonstrado melhor desempenho do que redes "rasas", que não possuem tais divisões de funções entre as unidades e que, apesar de inúmeros estudos e testes terem sido realizados, cada um levando a inúmeras conclusões, ainda não se tem certeza do porquê que estas redes desempenham de tal maneira, visto a quantidade enorme de variáveis e fenômenos interagindo com e influenciando uns aos outros.

Mesmo assim, DNNs já foram utilizadas com graus variáveis de sucessos, como reconhecimento e geração de dígitos manuscritos (HINTON et al., 2006), reconhecimento de brinquedos em fotos (LECUN et al., 2004) e objetos gerais (FEI-FEI et al., 2004; TORRALBA et al., 2007), inclusive em dados com a dimensão de tempo, como áudio (MOHAMMED et al., 2009; MOHAMMED et al., 2012) e vídeo (SUTSKEVER & HINTON, 2007). Em geral, de acordo com Tieleman (2014), DNNs demonstraram ser mais úteis em dados com alta dimensionalidade.

De acordo com Tieleman (2014), inúmeros tipos de dados possuem uma estrutura hierárquica, o que inspira pesquisadores a montarem redes hierárquicas para lidar com esses dados. Considerando imagens como exemplo, no nível mais baixo são pixels que, juntos, formam bordas, cores (e gradientes); em um nível mais alto, grupos dessas estruturas intermediárias constituem formas e objetos que, por sua vez formam uma cena.

Em uma DNN, a primeira camada recebe como entrada os pixels de uma imagem, com cada camada sucessiva processando e interpretando os dados em função dos resultados da camada anterior. Conforme os dados vão passando por camadas mais "altas", vai se formando uma representação de mais alto-nível desses

dados. A DNN funciona partindo do pressuposto que existe uma estrutura inerente aos dados fornecidos, e que espera-se que a rede consiga aproveitar a presença de tal estrutura (TIELEMAN, 2014).

Sutskever & Hinton (2008) afirmam que qualquer função suave pode ser bem aproximada por uma rede rasa de largura (unidades por camada) exponencial ou por uma rede estreita de profundidade (número de camadas) exponencial. Sabe-se que um baixo nível de profundidade restringe o conjunto de funções que uma rede é capaz de aproximar eficientemente (MINKSY & PAPERT, 1969; ALLENDER, 1996; BENGIO, 2009; BENGIO & LECUN, 2007), o que significa que, redes neurais profundas tendem a ser mais complicadas e difíceis de serem treinadas, mas possuem um potencial bem maior que redes rasas.

Tieleman (2014) afirma que, do mesmo jeito que existem tarefas para as quais DNNs são extremamente apropriadas, existem as tarefas em que DNNs simplesmente não ajudarão: dados com pouca ou nenhuma estrutura. Nesses casos, tentar extrair significado e relações dos dados não irá adiantar, sendo outras abordagens, como máquinas de vetor de suporte (SVM - Support Vector Machines) para classificação (CORTES & VAPNIK, 1995), mais apropriadas.

2.5. ESTRATÉGIAS DE TREINAMENTO

Uma vez estabelecida a arquitetura da rede, é necessário determinar como treiná-la. Pode-se usar uma função objetivo junto com um otimizador numérico ou, alternativamente, treinar uma camada de cada vez. Após esse treinamento, é possível treinar com o modelo completo e calibrar melhor seus parâmetros. Por causa disso, esse tipo de treinamento é comumente chamado de pré-treinamento, e pode ser usado tanto para extração de características quanto para modelagem de distribuições (TIELEMAN, 2014).

Uma das estratégias adotadas neste trabalho é a de treinamento por camada. A maior motivação para o treino por camada é o fato de que treinar um modelo completo do zero é muito difícil, como se tenta fazer com abordagens mais populares, como o método do gradiente. O método do gradiente tende a se deparar com dois problemas: o primeiro ocorre quando os parâmetros do modelo são inicializados com valores pequenos, resultando em gradientes pequenos em relação às camadas inferiores (BENGIO et al., 2007). O segundo problema, é que muitos acreditam que o método do gradiente é incapaz de chegar em uma solução suficientemente boa se os valores não forem inicializados já pertos de uma solução decente. O pré-treinamento é uma tentativa de resolver esses problemas, oferecendo uma heurística para encontrar parâmetros iniciais aceitáveis (TIELEMAN, 2014).

Um método comumente utilizado para treinamento por camada é a Máquina Restrita de Boltzmann. Em RBMs binárias, o tipo de RBM utilizada na segunda abordagem, tem-se o gradiente *positivo*, que corresponde à redução da energia dos casos de treinamento e é facilmente computável com precisão, e o gradiente *negativo*, que corresponde a aumentar a energia de configurações às quais o modelo atribui altas probabilidades. Este, entretanto, é muito custoso para computar com precisão. Isso o torna um tópico de estudo interessante, pois torna necessário elaborar métodos de baixo custo para estimá-lo (TIELEMAN, 2014). Um método comumente usado para realizar tal aproximação eficientemente é o algoritmo de Divergência Contrastiva (CD, do inglês Contrastive Divergence) com 1 passo (CD-1) (HINTON, 2002; BENGIO & DELALLEAU, 2007; SUTSKEVER & TIELEMAN, 2010), que tenta, pelo menos, estimar com certa precisão a direção do gradiente negativo. Para conseguir uma amostra aproximada de um determinado modelo, o algoritmo inicia uma cadeia de

Markov com um dos dados de treinamento utilizados para estimar o gradiente positivo, realiza um número (passado como parâmetro para o algoritmo) determinado de atualizações de Gibbs e trata o resultado como uma amostra do modelo. Uma variação interessante do CD, o Mean Field CD (MF-CD) (HINTON, 2002), é totalmente determinístico, o que permite o uso de taxas de aprendizagem maiores com segurança.

Segundo Tieleman, existem algumas desvantagens com a abordagem de treinamento por camada. Essa abordagem geralmente parte do pressuposto que cada camada é treinada como se não tivesse nenhuma outra camada que precisasse ser treinada após a mesma. Ainda de acordo com Tieleman, a maior dessas desvantagens é que essa forma de treinamento, se for usado como ponto de partida para algum procedimento de otimização dos parâmetros da rede inteira, resulta no surgimento de mais meta-parâmetros. Consequentemente, se os meta-parâmetros da fase de pré-treino não forem bem escolhidos, o pré-treino pode prejudicar mais do que ajudar. Por isso, argumenta Tieleman, é importante treinar as camadas (apesar de ser uma de cada vez) de forma a levar em consideração que a saída desta será utilizada por uma camada subsequente. Adicionalmente, as camadas inferiores devem tentar preservar o máximo possível a informação contida nos dados após a transformação aplicada aos mesmos, focando mais na codificação e extração de características, enquanto que as camadas superiores têm um foco maior em modelar e/ou transformar os dados.

O tipo de treinamento(supervisionado ou não-supervisionado) é tipicamente do mesmo tipo da tarefa em questão, afirma Tieleman, mas que é possível obter bons resultados mesmo com, por exemplo, treinamento supervisionado associado a uma tarefa não-supervisionada. Nesse caso, trata-se de um treinamento não-supervisionado para a realização de uma tarefa supervisionada.

O treinamento, então, é composto por uma fase de inicialização não-supervisionada(pré-treino) seguida de uma fase de treinamento supervisionada. De acordo com Tieleman, a primeira cai em uma de duas categorias: pré-treinamento generativo e pré-treino de extração de características, que é como os auto-codificadores funcionam. Ainda de acordo com Tieleman, um pré-treinamento não-supervisionado deve ser considerado uma heurística, visto que a função objetivo desta fase tende a não ser muito relacionada à função objetiva da fase seguinte, com o treinamento supervisionado.

Segundo Tieleman, a maior motivação para se realizar um treinamento não-supervisionado surge do fato de que uma entrada qualquer possui bastante informação potencialmente útil que pode ser ignorada por métodos mais simples que tentam modelar $P(Y|X)$ diretamente, onde o treinamento não-supervisionado permite detectar estruturas que possam *potencialmente* facilitar a subsequente modelagem de $P(Y|X)$. Adicionalmente, essa abordagem permite o uso de muito mais parâmetros antes que o modelo sofra um sobre-ajuste, mas que acarreta na função objetiva do pré-treino ser diferente da função objetiva final.

As mesmas desvantagens do pré-treino “guloso” se aplicam aqui, diz Tieleman: um número elevado de meta-parâmetros e o risco das duas fases de treinamento não combinarem direito devido às funções objetivas serem muito diferentes uma da outra. Um exemplo do segundo problema seria um pré-treinamento não-supervisionado extraindo características dos dados que acabam não sendo úteis para o treinamento supervisionado.

2.6. CARACTERÍSTICAS ADICIONAIS A SEREM CONSIDERADAS

Segundo Tieleman (2014), não é incomum os dados passarem por alguma etapa de pré-processamento antes de serem alimentados à rede. Às vezes é necessário para colocar os dados em um formato mais facilmente manipulados pela rede ou, pelo menos, otimizá-los segundo as fraquezas/forças da rede em questão. Também pode acontecer de ter um receio por parte dos pesquisadores de que, ao não ajustar os dados, a rede simplesmente o fará nas primeiras camadas, portanto, o pré-processamento ajuda a reduzir o tempo de processamento e o tamanho da rede, mais especificamente, o número de camadas.

2.6.1. Otimização Numérica

Tieleman (2014) afirma que uma das partes de computação que toma mais tempo em uma rede neural é a da otimização numérica, mas que tende a não receber tanta atenção quanto deveria. A maioria das implementações adota a abordagem padrão de usar o gradiente estocástico e assumem que pouco pode ser feito para otimizar ainda mais, mas foi demonstrado que um esforço maior em otimização pode permitir que novas classes de modelos, especialmente os bem profundos ou recorrentes), aprendam muito bem, na prática (HINTON & SALAKHUTDINOV, 2006; MARTENS, 2010).

Com o método do gradiente nunca se chega a um local optimum da função objetivo. Em algum ponto, o valor da função objetiva estabiliza e assume-se que atingiu-se o valor ótimo local optimum, mas nao tem como garantir que, com mais otimizações, não se obteria resultados melhores e que, na verdade, o que se observa é a “falha do otimizador”. Essa “falha” pode ser útil como um sinal para interromper o processo mais cedo, mas o ideal é ter um pouco mais de controle sobre quando isso acontece (TIELEMAN, 2014).

Uma abordagem que tem se tornado bastante popular, como uma alternativa para otimização de software, é o uso de hardware melhor. Foi demonstrado que é possível acelerar consideravelmente vários algoritmos de aprendizagem de máquina usando dispositivos originalmente destinados a processamento gráfico e que foram especializados para computação científica (RAINA et al., 2009; MNIH, 2009; TIELEMAN, 2010).

2.6.2. Propriedades de Neurônios Desejáveis

Trabalhos recentes (RANZATO et al., 2006; LEE et al., 2009; LEE et al, 2010) mostram que uma propriedade interessante dos neurônios é a de esparsidade. Nesse contexto, esparsidade é definido como o fenômeno de ter um grupo de unidades que raramente são ativadas. De acordo com Tielemen (2014), o interesse se dá pela interpretabilidade que essa propriedade oferece: uma unidade que é ativada um tanto quanto raramente muito provavelmente corresponde a algo muito específico e, espera-se, fácil de identificar na sequência de entradas, ao contrário de uma unidade que é ativada com frequência oferece menos informação(em uma ativação individual) quanto ao quê está acionando-a. Vale ressaltar que, em unidades mais ativas, apesar de cada ativação carregar menos informação, devido à frequência, a unidade carrega, *em média*, mais informação, ou seja, possui um nível mais alto de entropia.

Outra propriedade de interesse é a de invariância. O motivo do interesse se dá pelo fato de que grande parte das invariâncias correspondem a transformações, às vezes significativas, dos dados de entrada no que diz respeito à distância euclidiana, porém sem muitas alterações na semântica dos dados(i.e. com dados de áudio, uma pessoa falando mais alto ou mais baixo, os dados no nível mais baixo serão diferentes,

mas a semântica permanece a mesma). Portanto, um bom sistema deve ser capaz de focar nas características de alto nível, sem ser excessivamente afetado por variâncias que não mudam a semântica dos dados. Para redes de várias camadas, é importante distinguir entre invariância da saída e das representações intermediárias. Geralmente deseja-se invariância de saída, o que não necessariamente implica em uma representação interna invariante (TIELEMAN, 2014).

Quando surge uma relação entre as dimensões dos dados, torna-se interessante que determinadas unidades estejam conectadas apenas às unidades que representam as dimensões relacionadas. Isso se chama conectividade local. Um exemplo disso seria ter uma imagem como entrada e cada unidade(representada por um pixel) é relevante, ou seja, conectada, apenas às unidades que correspondem aos pixels na vizinhança. O que torna essa característica interessante, afirma Tieleman (2014) é como a conectividade nas camadas inferiores de uma rede combina com a idéia de termos dados estruturados e com hierarquias. Outro ponto positivo é que já modelando a rede levando em conectividade local em consideração, o modelo pode ser trabalhado com menos parâmetros do que se tivesse uma conectividade de nível mais global.

Robustez e independência das unidades também são consideradas características importantes para uma rede, visto que, se uma unidade depende de todas as outras fazerem tudo corretamente para esta contribuir com informação útil, o modelo tende a não ser muito confiável. Uma técnica para mitigar esse problema é o método de regularizaçao *dropout* (HINTON et al., 2012). Esta técnica consiste em, cada vez que um gradiente é calculado em um caso de treinamento, uma fração(50% tende a ser uma boa escolha na maioria dos casos) das unidades(pode ser tanto as ocultas como as de entrada) são temporariamente desativadas à força, de modo que não possam fazer sua parte. O resultado disso é que as unidades restantes também aprendem a operar sem contar com a presença das unidades vizinhas, tornando-as mais independentes.

2.6.3. Meta-parâmetros em DNNs

Existem muitos aspectos a serem considerados quando projetando uma rede neural, tais como arquitetura, estratégia de otimização e função objetiva. De acordo com Tieleman (2014), em alguns casos, é possível se ter uma certa intuição de qual estratégia funciona melhor, mas, devido ao enorme número de meta-parâmetros envolvidos, o ideal é poder testar as diferentes combinações de forma automatizada.

A maioria dos meta-parâmetros são dedicados à parte da arquitetura do modelo. Para começar, temos o número de camadas. Ainda não se tem um consenso sobre qual é número ideal de camadas, ainda mais que pode variar conforme a tarefa a ser realizada. Felizmente, avanços em otimizações permitiram a viabilidade de números mais elevados de camadas (MARTENS, 2010; SUTSKEVER et al., 2013). Em seguida, é necessário decidir a largura das camadas. Todas as camadas são da mesma largura, a não ser que se tenha razões muito específicas para o contrário (SALAKHUTDINOV & HINTON, 2009). Camadas largas tendem a ajudar, mas não fica claro quais camadas estão se beneficiando mais do recurso computacional adicional. Finalmente, é necessário decidir quais tipos de unidades serão utilizadas, sendo a mais comum, atualmente, a unidade logística. Além da logística, também tem as ReLUs(do inglês, Rectifier Linear Unit), que têm adquirido mais popularidade e não sofrem tanto quanto outros tipos com o problema do *vanishing gradient*.

Existe, também, uma abordagem mais manual de se otimizar os parâmetros de uma rede. Um problema com isso é que, dependendo do conjunto de treinamento(e de

seu tamanho), os valores escolhidos podem acabar servindo apenas para o mesmo. Adicionalmente, do ponto de leitores, o processo que levou à escolha dos valores pode não ficar claro(ou até ser omitido) em publicações, disfarçando o quão sensível é o modelo para uma determinada variação nos mesmos. Por último, argumenta Tieleman (2014), o pesquisador não consegue garantir que as escolhas de valores de meta-parâmetros estão sequer próximas de serem ótimas, a não ser que cada um seja validado - o que pode levar tempo considerável da parte do pesquisador.

Uma abordagem mais automatizada seria de busca em grade (do inglês, *grid*) sobre alguns possíveis valores de meta-parâmetros (ERHAN et al., 2009; PINTO et al., 2009). Em termos de programação, essa técnica se resume a alguns loops aninhados. A vantagem dessa abordagem é que facilita a identificação de sensibilidade de variação de valores nos meta-parâmetros, enquanto que a desvantagem é que, inerentemente, são necessários muitas iterações e o número de meta-parâmetros que podem ser calibrados é apenas logarítmico em relação ao número de execuções dos testes, afirma Tieleman.

DNNs são difíceis de serem estudadas sistematicamente. Entre os obstáculos estão a quantidade elevada de meta-parâmetros, tempos longos de execução e a natureza caixa-preta das redes neurais. Análises empíricas dificilmente ajudam pois necessariamente envolvem execuções bem mais curtas e restritas em ambientes de teste do que se vê em aplicações concretas. Adicionalmente, argumenta Tieleman, a configuração de cada experimento de cada equipe é diferente em pelo menos alguns aspectos, portanto, difíceis de serem comparadas (TIELEMAN, 2014).

2.7. REDES NEURAIS CONVOLUCIONAIS

Redes neurais convolucionais (CNN, do inglês *Convolutional Neural Network*) são um tipo especial de rede neural, mais especificamente, de DNN, que já foram aplicadas em vários problemas de reconhecimento de padrões, como visão computacional, reconhecimento de fala, etc. A CNN foi inicialmente inspirada por Hubel & Wiesel (1962) e continuamente implementada por vários pesquisadores. Alguns exemplos de experiências bem-sucedidas de CNNs são NeoCognitron (FUKUSHIMA, 1980), para reconhecimento de padrões com tolerância à variância da posição dos mesmos, LeNet5 (LECUN et al., 1998), para reconhecimento de documentos de texto, HMAX (SERRE et al., 2007), para reconhecimento de cenas visuais complexas, AlexNet (KRIZHEVSKY et al., 2012) e GoogleNet (SZEGEDY et al., 2015), para reconhecimento e classificação de imagens da base ImageNet, e ResNet (HE et al., 2015), também para reconhecimento de imagens.

A idéia principal de CNNs é de conseguir tolerar, até certo grau, invariância nos dados de entrada, através de três conceitos arquiteturais: *campos receptivos locais*, *pesos compartilhados* e *subamostragem* (do inglês, *subsampling*) (LECUN et al., 1998). A origem dessa preocupação surgiu em função do problema que redes FF tradicionais (do inglês, *feed-forward*), especialmente as de múltiplas camadas, possuem suas camadas completamente conectadas entre si, resultando em um número elevado de neurônios necessários e, consequentemente, mais parâmetros para aprender, para executar a mesma tarefa (especialmente no contexto de processamento de imagens, onde cada pixel é um ponto de dado). Uma consequência do alto grau de conectividade entre as camadas é a perda de informação espacial embutida nas entradas. CNNs, por outro lado, por fazer uso do compartilhamento de pesos, uma CNN pode ser mais profunda com menos parâmetros (AGHBAM & HERAVI, 2017).

Diferente de redes neurais tradicionais, CNNs possuem uma arquitetura especial. A arquitetura de uma CNN é tipicamente composta por uma camada convolucional e uma camada de subamostragem, como mostrado na figura abaixo. A camada convolucional implementa uma operação de convolução, enquanto que uma operação de subamostragem, também chamado de *pooling*, é realizada na camada de subamostragem. Como dito anteriormente, uma CNN é construída baseada em três conceitos principais: compartilhamento de pesos, campos receptivos locais e subamostragem.

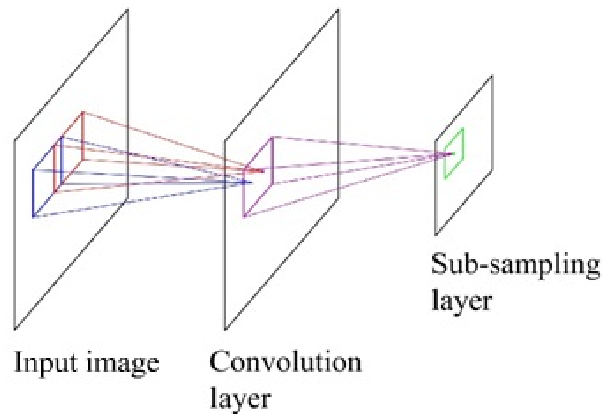


Figura 1. *A arquitetura típica de uma CNN tradicional, mostrando uma camada convolucional, seguida de uma camada de subamostragem. Pode-se usar várias combinações destas camadas.*

Na rede neural FF, a entrada está totalmente conectada com a camada oculta seguinte, para cada neurônio. Na CNN, por outro lado, apenas uma região pequena da entrada está conectada a um neurônio da camada oculta seguinte. Colocado de outra forma: cada neurônio da camada oculta estará conectado a uma pequena região da camada anterior, que se chama de *campo receptivo local*. Por exemplo, se os campos tem uma área de 3x3 pixels, um neurônio da primeira camada convolucional corresponde a 9 pixels da camada de entrada, como ilustrado na figura 2, abaixo.

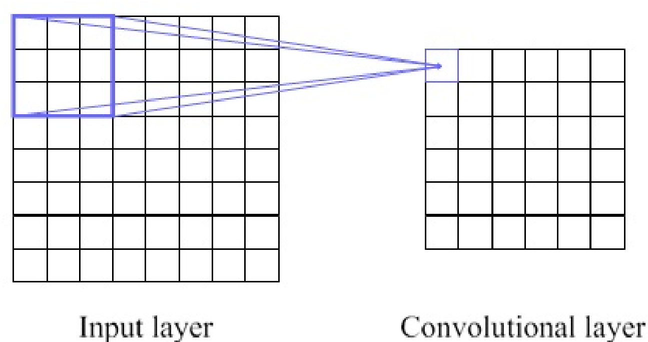


Figura 2. *Um exemplo de campos receptivos locais em uma camada convolucional.*

Na camada convolucional, os neurônios estão organizados em múltiplas camadas ocultas paralelas, chamadas de mapas de características (do inglês *feature map*). Cada neurônio em um desses mapas está conectado a um campo receptivo local e, para cada mapa, todos os neurônios compartilham do parâmetro de peso, conhecido com *kernel* ou *filtro*. A figura abaixo mostra como uma entrada de 32x32

pixels é treinada por uma camada convolucional com filtro 3x3 e 5 mapas de características.

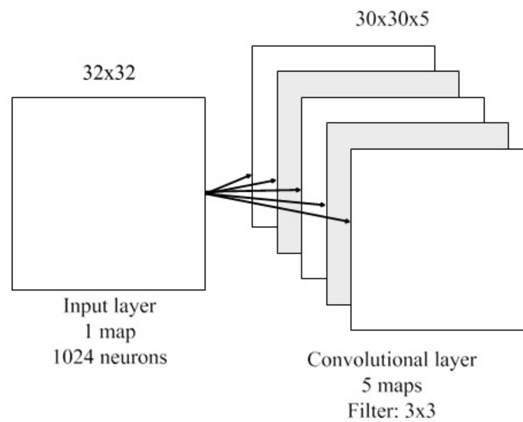


Figura 3. *Um exemplo de uma camada convolucional. Tamanho do filtro é 3x3 com 5 mapas de características.*

Como dito anteriormente, uma CNN também pode conter camadas de subamostragem, ou pooling. Quando se tem uma camada de subamostragem, este é usado imediatamente após uma camada convolucional, ou seja, as saídas da camada convolucional são as entradas para a camada de sub-amostragem da rede. A idéia da camada de amostragem é de gerar características invariantes à translação através da computação de estatísticas das ativações convolucionais de um campo receptivo que corresponde ao mapa de características. Estas estatísticas podem ser obtidas utilizando o valor máximo, mínimo ou médio, por exemplo, dos valores que vierem da camada convolucional anterior. O tamanho do campo, nesse caso, depende do tamanho da subamostragem. Um exemplo de como funciona para cada mapa é ilustrado na figura, abaixo, seguido de uma ilustração de como o processo funcionaria na presença de múltiplos mapas.

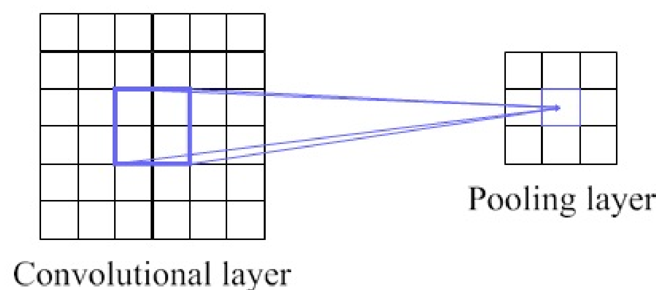


Figura 4. *Exemplo de uma camada de subamostragem em um mapa de características, com tamanho de subamostragem 2x2.*

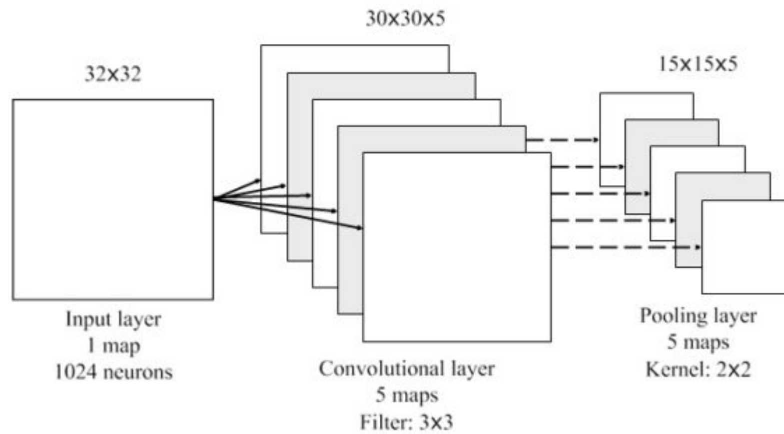


Figura 5. *Exemplo das camadas convolucionais e de pooling para a extração de mapas de características.*

2.8. MÁQUINAS RESTRITAS DE BOLTZMANN

Uma Máquina de Boltzmann é uma rede de unidades semelhantes a neurônios conectadas simetricamente que tomam decisões estocásticas sobre se devem estar ligadas ou desligadas. As Máquinas de Boltzmann possuem um algoritmo de aprendizagem simples (HINTON & SEJNOWSKI, 1983) que lhes permite descobrir características interessantes que representam regularidades complexas nos dados de treinamento.

As Máquinas de Boltzmann são usadas para resolver dois problemas computacionais bastante diferentes. Para um problema de *busca*, os pesos nas conexões são fixos e são usados para representar uma função de custo. A dinâmica estocástica de uma máquina Boltzmann permite que ela faça amostragens de vetores de estados binários que possuem valores baixos para a função de custo. Para um problema de *aprendizagem*, a Máquina de Boltzmann é mostrada um conjunto de vetores de dados binários e deve aprender a gerar esses vetores com alta probabilidade. Para fazer isso, deve encontrar pesos nas conexões para que, em relação a outros vetores binários possíveis, os vetores de dados tenham valores baixos para a função de custo. Para resolver um problema de aprendizagem, as máquinas Boltzmann fazem muitas pequenas atualizações para seus pesos e cada atualização exige que eles solucionem muitos problemas de busca diferentes (HINTON, 2007).

O algoritmo de aprendizagem é muito lento em redes com muitas camadas de detecção de características, mas é rápido em um tipo específico de Máquina de Boltzmann: as Máquinas Restritas de Boltzmann (RBM, do inglês *Restricted Boltzmann Machine*), que possuem uma única camada de detectores de recursos. Uma RBM é um modelo baseado em energia para aprendizagem não-supervisionada (SMOLENSKY, 1986) e consiste em duas camadas de unidades: uma visível, para representar os dados, e uma camada oculta. O tipo mais simples de RBM, que é o que forma a base para a segunda abordagem deste trabalho, é onde as unidades têm uma saída binária, apesar de poder modelar com outras distribuições, como a gaussiana e a de Poisson. A RBM binária tem três conjuntos de parâmetros: um viés para cada

unidade visível, um viés para cada unidade oculta e um conjunto de conexões com pesos entre cada par de unidade visível e oculta.

Após a aprendizagem de uma camada oculta, os vetores de atividade das unidades escondidas, quando são conduzidos pelos dados reais, podem ser tratados como "dados" para treinar outra RBM. Isso pode ser repetido para aprender quantas camadas ocultas forem necessárias. Depois de aprender várias camadas ocultas dessa maneira, toda a rede pode ser vista como um modelo generativo único e multicamada e cada camada escondida adicional melhora um limite inferior na probabilidade de o modelo multicamada gerar os dados de treinamento (HINTON et al., 2006).

Aprender uma camada oculta de cada vez é uma maneira muito eficaz de aprender redes neurais profundas com muitas camadas ocultas e milhões de pesos. Embora a aprendizagem não seja supervisionada, os recursos de nível mais alto são geralmente muito mais úteis para a classificação do que os vetores de dados brutos. Essas redes profundas podem ser aperfeiçoadas para melhorar a classificação ou redução de dimensionalidade usando o algoritmo de backpropagation (HINTON & SALAKHUTDINOV, 2006). Alternativamente, elas podem ser ajustadas para serem melhores modelos generativos usando uma versão do algoritmo "wake-sleep" (HINTON et al., 2006).

2.9. AUTO-CODIFICADORES

Redes neurais estão entre os métodos mais bem-sucedidos na área de visão computacional e são aptas para aproveitarem bem a crescente quantidade de dados e pode computacional à nossa disposição (KRISZHEVKSY, A. et al, 2012). Entre as diferentes abordagens para a criação de redes neurais capazes de reconhecer imagens, está a explorada por Tieleman (2014), que consiste em aprender a gerar imagens antes de tentar reconhecê-las.

Existem uma série de maneiras através das quais redes neurais conseguem gerar imagens, com o método mais utilizado, sendo o de modelos geradores probabilísticos (TIELEMAN, 2014). Existem vários tipos destes modelos geradores, e a proposta de Tieleman (2014) trabalha em cima de modelos do tipo não-direcionados.

Um dos tipos de redes neurais explorados por Tieleman (2014) para a geração de imagens é o auto-codificador - com ênfase no componente de decodificação. Auto-codificadores, também chamados de redes neurais autocodifidoras, são uma espécie de algoritmo não-supervisionado para a geração de codificações eficientes (LIOU et al., 2006; LIOU et al., 2013). Auto-codificadores visam aprender uma representação, ou codificação, para um conjunto de dados, geralmente com o intuito de reduzir a dimensionalidade dos dados e, recentemente, tem sido cada vez mais utilizado para a aprendizagem de modelos generativos (KINGMA & WELLING, 2013; BOESEN et al., 2015).

Apesar de auto-codificadores típicos serem determinísticos, o decodificador é capaz de gerar imagens diferentes graças ao fato que o mesmo pode, após a rede ser treinada, receber dados de entrada diferentes. Tipicamente, permite-se que a rede chegue na sua própria representação dos dados, o que torna o processo bastante impreciso e incerto (TIELEMAN, 2014). A figura 6, abaixo, mostra uma visão geral da arquitetura típica de um auto-codificador:

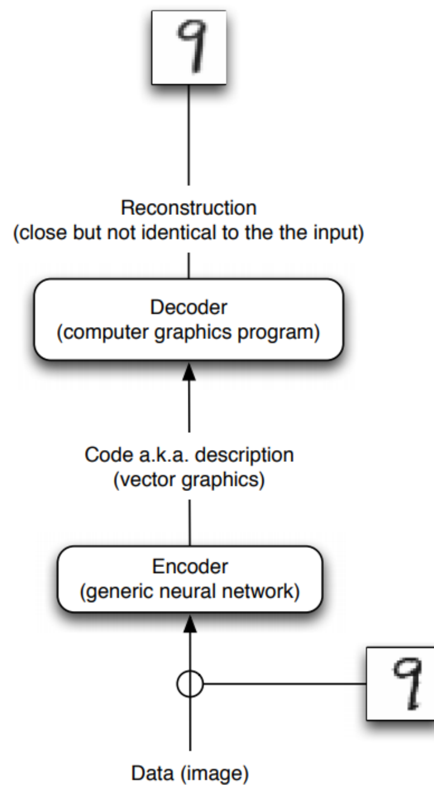


Figura 6. *Visão geral da arquitetura de um auto-codificador baseado em gráficos (*TIELEMAN, 2014).

3. TRABALHOS RELACIONADOS

A ferramenta principal, utilizada em todas as abordagens, será o TensorFlow. Desenvolvida pelo Google, TensorFlow é uma interface para expressar algoritmos de aprendizagem de máquina, e uma implementação para executar esses algoritmos. Uma computação expressa usando o TensorFlow pode ser executada com pouca ou nenhuma alteração em uma ampla variedade de sistemas heterogêneos, desde dispositivos móveis, como telefones móveis e tablets, até sistemas distribuídos em grande escala de centenas de máquinas e milhares de dispositivos computacionais, como GPUs.

O sistema é flexível e pode ser usado para expressar uma grande variedade de algoritmos, incluindo algoritmos de treinamento e inferência para modelos de redes neurais profundas. Este tem sido utilizado para realizar pesquisas e para implantar sistemas de aprendizagem de máquinas em produção em inúmeras áreas de ciência da computação e outros campos, incluindo reconhecimento de fala, visão computacional, robótica, recuperação de informações, processamento de linguagem natural e extração de informações geográficas. A API do TensorFlow e uma implementação de referência foram lançadas como um pacote open-source sob a licença Apache 2.0 em Novembro de 2015 e estão disponíveis em www.tensorflow.org.

Neste trabalho, três abordagens para a geração de imagens são exploradas. Em duas das abordagens, são utilizadas redes convolucionais (uma delas dentro do contexto de redes generativas adversariais), enquanto que, na terceira, é utilizada redes recorrentes. Na seção a seguir, são detalhadas as abordagens, conforme elaboradas nos trabalhos relacionados.

3.1. DRAW: UMA REDE NEURAL RECORRENTE PARA A GERAÇÃO DE IMAGENS

A maioria das abordagens de geração automática de imagens visa gerar a imagem inteira, com todos os componentes que a compõe, de uma só vez (DAYAN et al., 1995; HINTON & SALAKHUTDINOV, 2006; LAROCHELLE & MURRAY, 2011). Isso impõe a condição de que todos os pixels da imagem estejam condicionados em função de uma distribuição latente. Essa estratégia não escala facilmente para imagens cada vez maiores (GREGOR, K. et al., 2015). A arquitetura proposta por *DRAW (Deep Recurrent Attentive Writer)* (GREGOR, K. et al., 2015) visa incentivar uma mudança para uma forma mais natural de construção de imagens, onde partes individuais de uma cena são incrementalmente melhoradas independentemente uma das outras.

O cerne da arquitetura DRAW é composto por um par de redes neurais recorrentes, onde uma atua como codificador, que comprime a imagem durante a fase de treinamento, e a segunda funciona como um decodificador, que tenta reconstruir as imagens em função dos dados fornecidos. Estas redes são então treinadas de ponta-a-ponta com o SGD, onde a função de perda é um limite superior variacional na *função de probabilidade logaritmica* dos dados. Em função disso, esta arquitetura pertence à classe de auto-codificadores variacionais (*VAE*, do inglês *variational auto-encoders*). A diferença principal entre a DRAW e outros VAEs é a característica supracitada de, ao invés de tentar gerar a imagem de uma só vez, construir a cena iterativamente, através de mudanças aplicadas pelo decodificador.

3.1.1. A rede DRAW

A estrutura básica de uma rede DRAW é semelhante à de outros VAEs: uma rede codificadora determina a distribuição de códigos latentes que contém informações salientes sobre os dados de entrada; uma rede decodificadora que recebe amostragens destas distribuições e os utiliza para condicionar sua própria distribuição em cima das imagens. Três diferenças importantes diferenciam a DRAW de outros VAEs: a primeira é que nesta arquitetura ambas as redes são recorrentes, portanto, estas trocam entre si *sequências* de amostras de código; adicionalmente, o codificador é ciente das saídas do decodificador. Em segundo lugar, as saídas do decodificador são sucessivamente adicionadas à distribuição que irá, eventualmente, gerar os dados, ao invés de gerar os dados de uma só vez. E, terceiro, é utilizado um mecanismo dinâmico de atenção que determina tanto a região da entrada que é analisada pelo codificador quanto a região de saída do decodificador. Isso permite que a rede saiba *onde ler*, *onde escrever* e o *quê escrever*.

3.1.2. Operações de Leitura e de Escrita

Na instanciação mais simples de DRAW, toda a imagem de entrada é passada para o codificador a cada passo de tempo, e o decodificador modifica toda a matriz do canvas a cada passo de tempo. No entanto, essa abordagem não permite que o codificador se concentre em apenas parte da entrada ao criar a distribuição latente e nem permite que o decodificador modifique apenas uma parte do vetor do canvas. Em outras palavras, não fornece à rede um mecanismo explícito de atenção seletiva, que acredita-se ser crucial para a geração de imagens em larga escala. Esse modelo básico, sem mecanismo de atenção é chamada pelos autores de “DRAW sem atenção”.

Para incorporar o mecanismo de atenção seletiva à rede sem sacrificar os benefícios do treinamento com gradiente descendente, os autores se inspiraram nos mecanismos diferenciais de atenção usados na síntese de caligrafia (GRAVES, 2013) e Máquinas de Turing Neurais (GRAVES et al., 2014). Ao contrário dos trabalhos supracitados, é utilizada uma abordagem com uma forma de atenção explicitamente bidimensional, onde um vetor de filtros Gaussianos bidimensionais é aplicado na imagem, produzindo um "patch" de imagem de localização e zoom levemente variados. Os autores ressaltam que essa configuração, que se referem simplesmente como “DRAW”, lembra um pouco as transformações usadas nos auto-codificadores baseados em computação gráfica (TIELEMAN, 2014).

Uma grade $N \times N$ de filtros Gaussianos é posicionada na imagem especificando as coordenadas do centro da grade e a distância dos passos entre os filtros adjacentes. O passo controla o "zoom" do patch; ou seja, quanto maior o passo, maior será a área da imagem original visível no patch de atenção, mas menor será a resolução efetiva do patch.

3.1.3. Resultados Experimentais

Os experimentos para determinar a habilidade da arquitetura DRAW de gerar imagens realistas foram realizados em cima de três conjuntos de dados: o MNIST, o Street View House Numbers (SVHN) (NETZER et al., 2011) e o CIFAR-10. De acordo com os autores, as imagens geradas pela rede são sempre novas (não apenas cópias de exemplos do conjunto de treinamento), e são praticamente indistinguíveis dos dados reais para MNIST e SVHN; as imagens CIFAR geradas são um pouco borradas, mas ainda contêm estruturas reconhecíveis de cenas naturais. Os resultados MNIST binarizados melhoram substancialmente o estado da arte. Os parâmetros de rede para todos os experimentos são apresentados na tabela abaixo. O algoritmo de otimização Adam (KINGMA & BA, 2014) foi utilizado em todo o processo.

Task	#glimpses	LSTM $\#h$	$\#z$	Read Size	Write Size
100×100 MNIST Classification	8	256	-	12×12	-
MNIST Model	64	256	100	2×2	5×5
SVHN Model	32	800	100	12×12	12×12
CIFAR Model	64	400	200	5×5	5×5

Tabela 1. *Os parâmetros utilizados nas redes em cada tarefa: o número de “vislumbradas” realizadas pela rede codificadora (glimpses), o número de unidades ocultas (#h), número de amostragens (#z), e o tamanho da área de leitura e escrita.*

De acordo com os autores (GREGOR et al., 2015), para testar a eficácia da classificação do mecanismo de atenção do DRAW (ao invés de sua capacidade de auxiliar na geração de imagens), foi avaliado o seu desempenho na tarefa do MNIST 100×100 traduzido e desordenado (MNIH et al., 2014). Cada imagem no MNIST desordenado contém vários fragmentos de aparência próximos a de dígitos que a rede deve distinguir do dígito verdadeiro a ser classificado. Para classificar o dígito, a rede (uma rede LSTM recorrente) utiliza uma camada *softmax*. Como ilustrado na figura 7, abaixo, ter um modelo de atenção iterativo permite que a rede amplie progressivamente a região relevante da imagem e ignore a desordem fora dela.

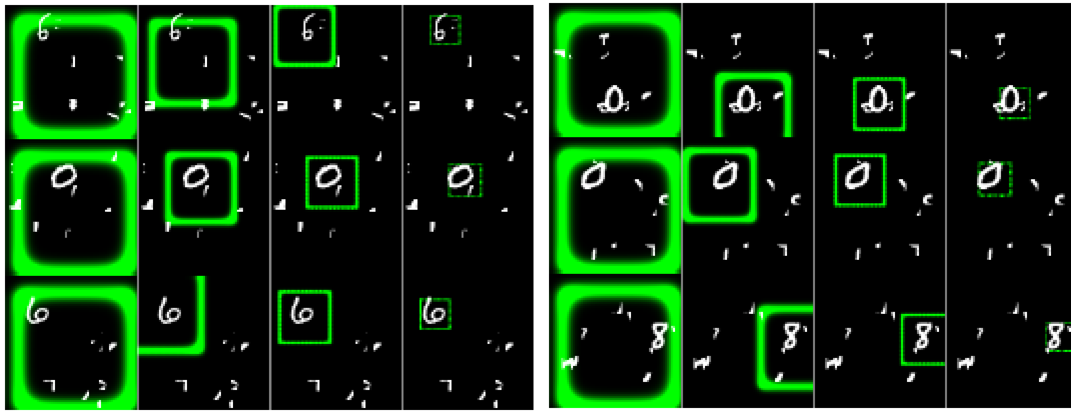


Figura 7. Classificação desordenada de MNIST com atenção: *Cada sequência mostra uma sucessão de quatro "vislumbradas" tomadas pela rede enquanto classifica o MNIST desordenado traduzido. O retângulo verde indica o tamanho e a localização do patch de atenção, enquanto a largura da linha representa a variação dos filtros.*

Para o experimento de geração de imagens MNIST, os autores treinaram uma rede DRAW completa como um modelo generativo. Segundo os autores, o DRAW *sem atenção* obteve desempenho semelhante às outras arquiteturas avaliadas por eles, enquanto que o DRAW *com atenção* conseguiu melhorar consideravelmente o estado da arte.

Uma vez treinada, a rede pôde gerar as imagens MNIST. Pode-se notar, na figura 8, a diferença entre a escrita *sem atenção* (à direita), onde cada imagem é iterativamente melhorada de uma forma mais global, e a escrita *com atenção*, onde a rede melhora iterativamente, porém localmente, cada imagem:

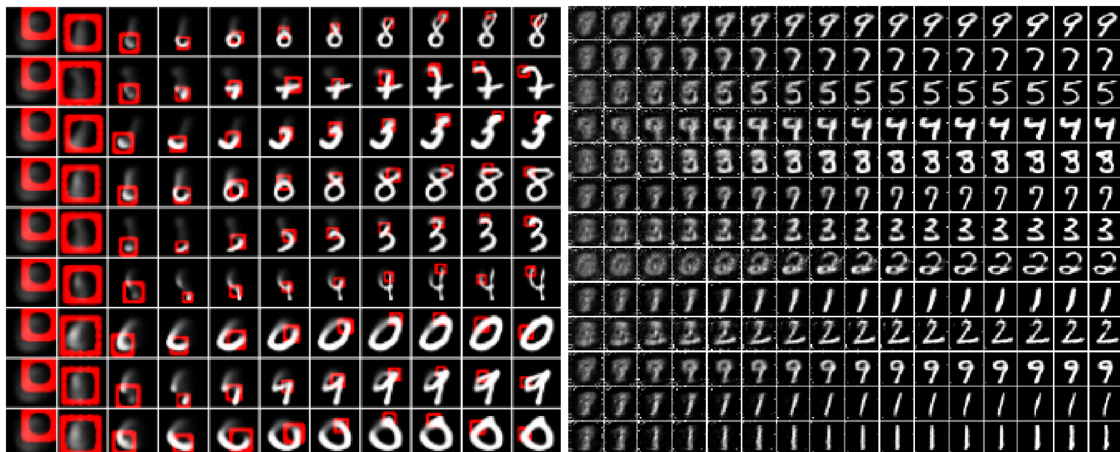


Figura 8. *Exemplo de dígitos MNIST gerados com atenção (à esquerda) e sem atenção (à direita).*

Segundo (GREGOR et al., 2015), principal motivação para usar um modelo generativo com mecanismo de atenção é que imagens grandes podem ser construídas iterativamente, desenvolvendo uma pequena parte da imagem de cada vez. Para testar essa capacidade de maneira controlada, os autores treinaram o DRAW para gerar imagens com duas imagens MNIST 28x28 escolhidas aleatoriamente e colocadas em locais aleatórios em fundo preto 60x60. Nos casos em que os dois dígitos se sobrepõem, as intensidades dos pixels foram somadas em cada ponto e

truncadas para não serem maiores que 1. Exemplos de dados gerados são mostrados, abaixo, na figura 9. A rede geralmente gera um dígito e depois o outro, sugerindo uma capacidade de recriar cenas compostas de peças simples.

Figura 9. *Exemplo de imagens MNIST geradas com dois dígitos.*

Segundo (GREGOR et al., 2015), os dígitos MNIST são muito simplistas em termos de estrutura visual, portanto, seria interessante testar o quão bem DRAW se apresentava em imagens naturais. O primeiro experimento de geração de imagem natural usou o conjunto de dados de vários dígitos do Street View House Numbers (NETZER et al., 2011). Foi utilizado o mesmo pré-processamento que (GOODFELLOW et al., 2013), produzindo uma imagem 64x64 do número da casa para cada exemplo de treinamento. A rede foi então treinada usando patches 54x54 extraídos em locais aleatórios das imagens pré-processadas. O conjunto de treinamento SVHN contém 231.053 imagens e o conjunto de validação contém 4,701 imagens. De acordo com os autores (GREGOR et al., 2015), apesar do tempo treinamento, o DRAW não sofreu *overfitting* dos dados de treinamento, conforme mostrado na figura 10:

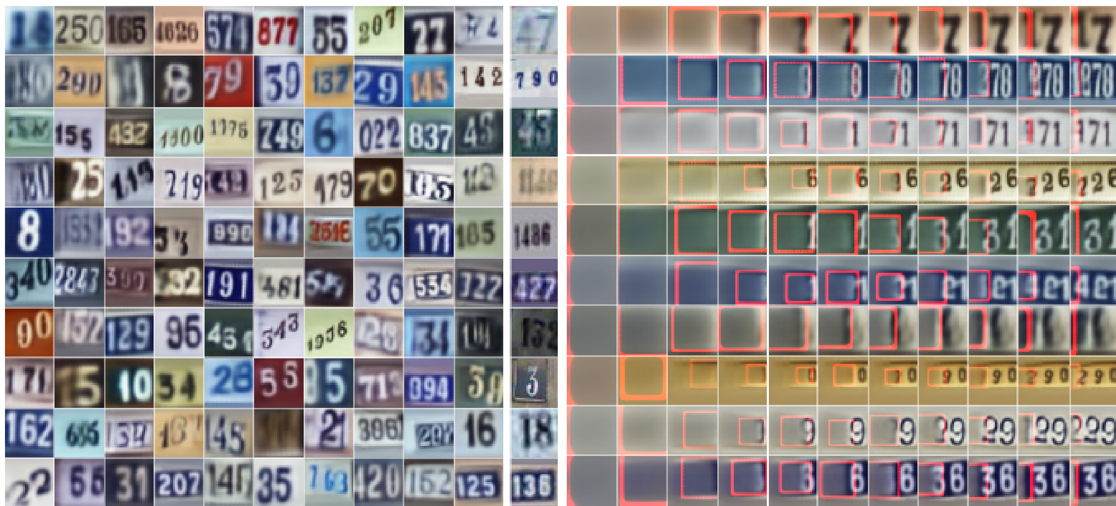


Figura 10. Imagens SVHN geradas: *A coluna mais à direita (da imagem à esquerda) mostra as imagens de treinamento mais próximas (na distância L^2) das imagens geradas ao lado delas.*

De acordo com os autores (GREGOR et al., 2015), o conjunto de dados mais desafiador ao o DRAW foi sujeitado foi a coleção de imagens naturais CIFAR-10 (KRIZHEVSKY, 2009). O CIFAR-10 é muito diversificado e, com apenas 50.000 exemplos de treinamento, é muito difícil gerar objetos de aparência realista sem *overfitting* (em outras palavras, sem copiar do conjunto de treinamento). No entanto, as imagens na figura 11 demonstram que o DRAW é capaz de capturar grande parte da forma, cor e composição das fotografias reais:

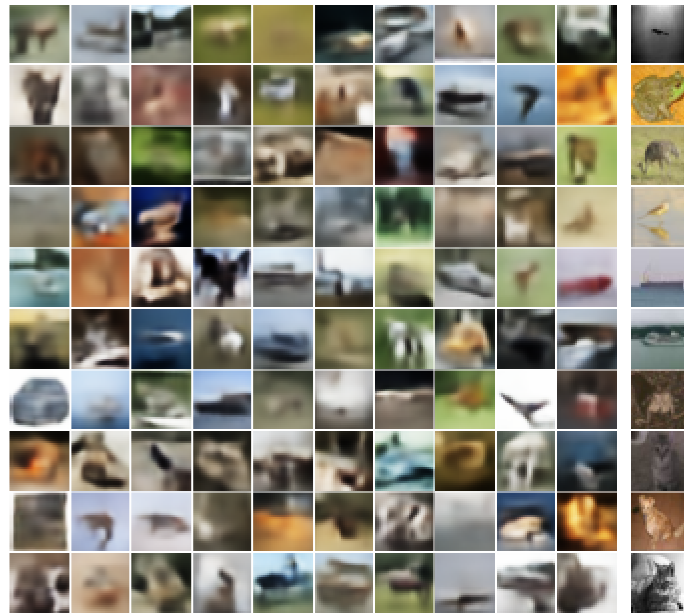


Figura 11. Imagens CIFAR geradas: *Coluna mais à direita mostra os exemplos de treinamento mais próximos à coluna ao lado.*

3.2. APRENDIZAGEM DE REPRESENTAÇÃO NÃO-SUPERVISIONADA COM REDES GENERATIVAS ADVERSARIAIS CONVOLUCIONAIS PROFUNDAS

Aprender representações reutilizáveis de características a partir de grandes conjuntos de dados não-rotulados tem sido uma área de pesquisa ativa. No contexto da visão computacional, pode-se aproveitar a quantidade praticamente ilimitada de imagens e vídeos não-rotulados para aprender boas representações intermediárias, que podem então ser usadas em uma variedade de tarefas de aprendizagem supervisionadas, como a classificação de imagens (RADFORD, A. & CHINTALA, S., 2016). Os autores (RADFORD, A. & CHINTALA, S., 2016) propõe que uma maneira de construir boas representações de imagem é treinando GANs (do inglês *Generative Adversarial Networks*) (GOODFELLOW et al., 2014), e depois reutilizando partes das redes geradoras e discriminadoras como extratores de características para tarefas supervisionadas. De acordo com os autores (RADFORD, A. & CHINTALA, S., 2016), sabe-se que as GANs são instáveis no treinamento, geralmente resultando em redes geradoras que produzem saídas sem sentido. Tem havido pesquisas publicadas muito limitadas na tentativa de entender e visualizar o que as GANs aprendem, e as representações intermediárias de GANs de várias camadas.

Neste trabalho, (RADFORD, A. & CHINTALA, S., 2016) fazem as seguintes contribuições:

- Propõem e avaliam um conjunto de restrições na topologia arquitetural das GANs Convolucionais que as tornam estáveis para treinamento na maioria dos ambientes, dando o nome para esta classe de arquiteturas de GANs Convolucionais Profundas (DCGAN)
- Utilizam as redes discriminadoras treinadas para tarefas de classificação de imagens, mostrando desempenho competitivo com outros algoritmos não-supervisionados
- São visualizados os filtros aprendidos pelas GANs e demonstram empiricamente que filtros específicos aprenderam a desenhar objetos específicos
- Mostram que os geradores têm interessantes propriedades aritméticas vetoriais, permitindo fácil manipulação de muitas qualidades semânticas das amostras geradas.

3.2.1. Abordagem e Arquitetura do Modelo

As tentativas históricas de aumentar as GANs usando CNNs para modelar imagens não tiveram êxito. Segundo os autores, foram encontradas dificuldades para escalar as GANs usando arquiteturas CNN usadas comumente na literatura supervisionada. No entanto, foi identificada uma família de arquiteturas que resultaram em treinamento estável em vários conjuntos de dados e permitiram o treinamento de modelos geradores com maior resolução e maior profundidade.

Essencial para a abordagem foi adotar e modificar três mudanças recentemente demonstradas nas arquiteturas da CNN.

A primeira é a rede toda convolucional (SPRINGENBERG et al., 2014), que substitui funções de agrupamento espacial determinístico (como *maxpooling*) por convoluções distribuídas, permitindo que a rede aprenda sua própria subamostragem espacial.

A segunda é a tendência de eliminar camadas totalmente conectadas sobre características convolucionais. O exemplo mais forte disso é o agrupamento médio global que foi utilizado em modelos de classificação de imagens de última geração (Mordvintsev et al.). Descobrimos que o agrupamento médio global aumentou a estabilidade do modelo, mas prejudicou a velocidade de convergência. Um meio termo de conectar diretamente as características convolucionais mais altas à entrada e à

saída, respectivamente, do gerador e do discriminador, funcionou bem. Para o discriminador, a última camada de convolução é achatada e depois alimentada em uma única saída sigmóide.

A terceira é a Normalização em Lote (IOFFE & SZEGEDY, 2015), que estabiliza o aprendizado normalizando a entrada para cada unidade para ter média e variação unitária iguais a zero. Isso ajuda a lidar com problemas de treinamento que surgem devido à inicializações de má qualidade e ajuda o fluxo do gradiente em modelos mais profundos. Aplicação direta de *batchnorm* em todas as camadas no entanto, resultou em oscilação da amostra e instabilidade do modelo. Isso foi evitado ao não aplicar *batchnorm* para a camada de saída da rede geradora e a camada de entrada da discriminadora.

A ativação de ReLU (NAIR & HINTON, 2010) é usada na rede geradora, com exceção da camada de saída que usa a função *Tanh*. Os autores observaram que o uso de uma ativação limitada permitiu que o modelo aprendesse mais rapidamente a saturar e cobrir o espaço de cores da distribuição de treinamento. Dentro da rede discriminadora, constataram que a ativação *LeakyReLU* (MAAS et al., 2013) (XU et al., 2015) funciona bem, especialmente para modelagem de maior resolução. Isto está em contraste com o papel GAN original, que usou a ativação *Maxout* (GOODFELLOW et al., 2013).

No geral, os autores recomendam a seguinte arquitetura:

Diretrizes de arquitetura para GANs Convolucionais Profundas estáveis:

- Substitua quaisquer camadas de pooling com convoluções em passos (discriminadora) e convoluções de passos fracionários (geradora).
- Use *batchnorm* tanto na rede geradora quanto na rede discriminadora.
- Remova camadas ocultas completamente conectadas para poder usar arquiteturas mais profundas.
- Use a ativação ReLU na rede geradora para todas as camadas, com exceção da saída, onde deve ser utilizado o Tanh.
- Use a ativação LeakyReLU na rede discriminadora em todas as camadas.

3.2.2. Detalhes de Treinamento Adversarial

Os autores (RADFORD, A. & CHINTALA, S., 2016) treinaram e testaram as DCGANs em três conjuntos de dados: Large-scale Scene Understanding (LSUN) (YU et al., 2015), Imagenet-1k e um conjunto de dados recém-montado, Faces.

Nenhum pré-processamento foi aplicado nas imagens de treinamento, além de escalar para a faixa de ativação da função *tanh* [-1, 1]. Todos os modelos foram treinados com SGD em mini-lotes, com os mini-lotes de tamanho 128. Todos os pesos foram inicializados a partir de uma distribuição Normal centrada em zero com desvio padrão de 0,02. No LeakyReLU, a inclinação do “vazamento” foi definida para 0,2 em todos os modelos. De acordo com os autores, embora o trabalho anterior do GAN tenha usado o momento para acelerar o treinamento, foi utilizado o otimizador Adam (KINGMA & BA, 2014) com hiperparâmetros sintonizados. Como taxa de aprendizagem, foi utilizada 0,0002, pois taxas mais altas (conforme sugestões em outros trabalhos) demonstraram ser muito altas. Além disso, foi constatado que deixar o momentum β_1 com o valor sugerido de 0,9 resultou na oscilação e instabilidade do treinamento, enquanto que reduzindo-o para 0,5 ajudou a estabilizar o treinamento (RADFORD, A. & CHINTALA, S., 2016).

À medida que a qualidade visual das amostras dos modelos geradores de imagem melhorou, aumentaram as preocupações de *overfitting* e memorização das

amostras de treinamento. Para demonstrar como o modelo escala com mais dados e maior resolução, (RADFORD, A. & CHINTALA, S., 2016) treinaram um modelo no conjunto de dados de quartos LSUN, contendo pouco mais de 3 milhões de exemplos de treinamento. Análises recentes mostraram que existe uma ligação direta entre a rapidez com que os modelos aprendem e seu desempenho de generalização (HARDT et al., 2015).

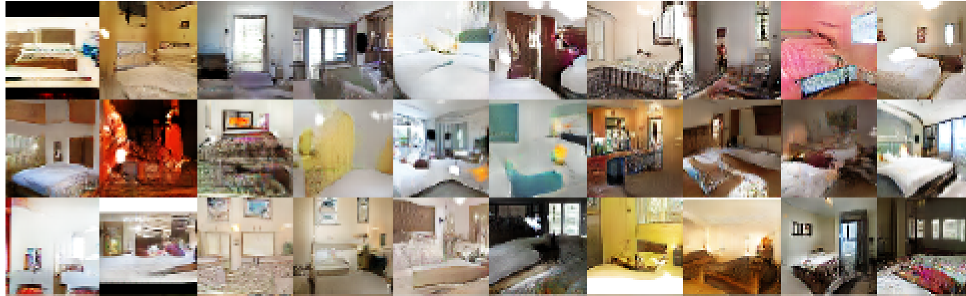


Figura 12. *Quartos gerados após cinco épocas de treinamento. Parece haver evidência de underfitting visual através de texturas de ruído repetidas em várias amostras, como as placas da base de algumas das camas.*

Para o conjunto Faces, os autores recortaram imagens contendo rostos humanos de consultas aleatórias de imagens da web de nomes de pessoas. Os nomes das pessoas foram adquiridos da DBpedia, com o critério de que nasceram na era moderna. Este conjunto de dados tem 3 milhões de imagens 10 mil pessoas. Foi aplicado um detector facial OpenCV em cima dessas imagens, mantendo as detecções com resolução suficientemente alta, o que forneceu aos autores aproximadamente 350.000 quadros de rostos. Os quadros de rosto, mostrados abaixo, foram utilizados o para treinamento.

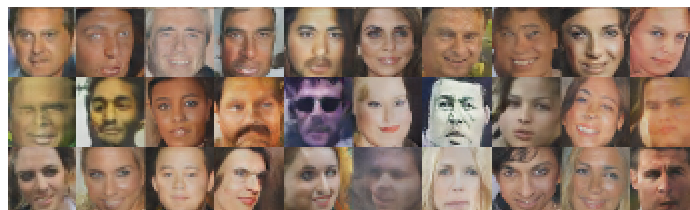


Figura 13. *Imagens geradas de um DCGAN treinado no conjunto de dados Faces.*

Os autores usaram o Imagenet-1k (DENG et al., 2009) como uma fonte de imagens naturais para treinamento não-supervisionado. Os treinamentos foram realizados em cima de imagens recortadas, centralizadas e redimensionadas para 32x32 (RADFORD, A. & CHINTALA, S., 2016).

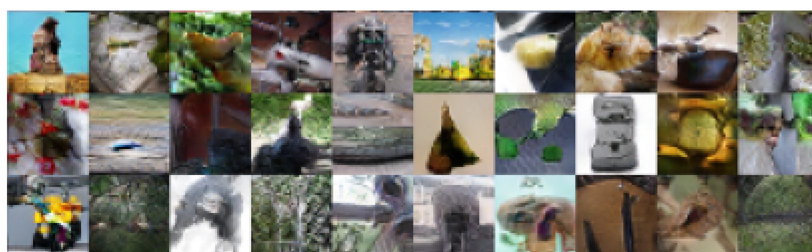


__Figura 14.__ Imagens geradas de um DCGAN treinado no conjunto de dados Imagenet-1k.

3.3. GERAÇÃO CONDICIONAL DE IMAGENS COM DECODIFICADORES PixelCNN

Avanços recentes em modelagem de imagens com redes neurais, como a PixelRNN (do inglês, Pixel Recurrent Neural Network) (OORD et al., 2016a) e LSTMs (do inglês, Long Short-Term Memory) (THEIS & BETHGE, 2015), tornaram viáveis a geração de imagens naturais diversificadas que capturam com sucesso as estruturas implícitas dos dados de treinamento. Estes modelos, contudo, não levam em consideração informações mais abstratas contidas nos dados, porém relevantes, para condicionar as imagens geradas(i.e.: presença de certos objetos na imagem ou alguma restrição de estado).

Além da arquitetura PixelRNN (OORD et al., 2016a), que modela a distribuição dos pixels através de LSTMs bi-dimensionais, a abordagem a seguir apresenta uma versão da arquitetura usando redes neurais convolucionais, a PixelCNN. De acordo com os autores originais (OORD et al., 2016a; OORD et al., 20016b), as PixelRNNs demonstraram, no geral, melhor desempenho. Por outro lado, as PixelCNNs eram significativamente mais rápidas para serem treinadas, dado que convoluções são mais facilmente paralelizáveis. Com a quantidade de pixels em conjuntos de imagens grandes, isso se mostrou ser uma imensa vantagem.

O abordagem apresentada explora o potencial da modelagem condicional de imagens ao construir em cima da versão convolucional da arquitetura utilizada na PixelRNN, resultando em dois modelos novos, a Gated PixelCNN e, ainda, uma variante desta, a Conditional PixelCNN (OORD et al., 2016b). De uma maneira geral, a idéia destas arquiteturas se baseia em utilizar conexões auto-regressivas para modelar as imagens pixel por pixel, decompondo a distribuição conjunta da imagem como um produto de condicionais.

3.3.1. Gated PixelCNN

PixelCNNs e PixelRNNs modelam a distribuição conjunta dos pixels de uma imagem ***x*** como o seguinte produto de distribuições condicionais, onde x_i é um único pixel:

$$p(x) = \prod_{i=1}^{n^2} p(x_i|x_1, \ ..., \ x_{i-1}) \, ,$$

Equação 1

A ordenação das dependências entre os pixels são determinadas por varredura, da esquerda para a direita, de cima para baixo. Ao final, cada pixel depende exclusivamente dos pixels acima e à esquerda. Na PixelCNN, toda distribuição condicional é modelada por uma rede neural convolucional e, para garantir que a rede não consiga usar informação dos pixels à frente e abaixo do pixel atual, é aplicada uma máscara aos filtros das convoluções, como mostra a figura 15:

1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

Figura 15. *Um exemplo de matriz utilizada para mascarar os filtros 5x5 para garantir que o modelo não consiga ler os pixels abaixo (ou imediatamente à direita) do pixel atual para realizar suas previsões.*

Adicionalmente, para cada pixel, os canais de cores são modelados sucessivamente, com (B - Blue) sendo condicionado por (R, G - Red, Green) e (G) condicionado por (R). Em seguida, os 256 possíveis valores de cada canal de cor são modelados através do uso da função *softmax*.

Tipicamente, a PixelCNN consiste de uma pilha de camadas convolucionais mascaradas que aceitam, como entrada, uma imagem *N x N x 3* e produz uma entre *N* x *N* x 3 x 256 previsão de valor de pixel. O uso de convoluções permite realizar a previsão para todos os pixels, através da equação acima, em paralelo durante a etapa de treinamento. Por outro lado, durante a fase de amostragem, as previsões devem ser realizadas sequencialmente para que o resultado da previsão seja realimentado para a rede para que a mesma possa calcular o próximo pixel (OORD et al., 2016b).

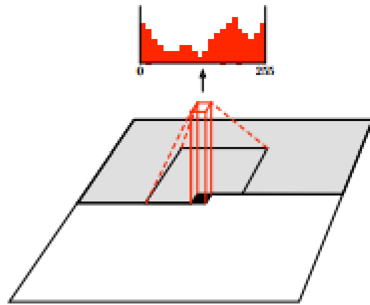
As PixelRNNs mostraram ter maior desempenho do que as PixelCNNs como modelo generativo (OORD et al., 2016b). Dois fatores surgiram como possíveis causas: as conexões recursivas das camadas de LSTM espaciais usadas nas PixelRNNs permitiam cada camada da rede de acessar toda a vizinhança dos pixels anteriores, enquanto que a vizinhança disponível para a PixelCNN crescia linearmente com a quantidade de camadas convolucionais; nas PixelRNNs, as LSTMs atuavam como unidades multiplicativas, o que possivelmente permite a modelagem de interações mais complexas. Para o primeiro ponto, a diferença de desempenho pôde ser, em grande parte, reduzida ao incluir um número suficiente de camadas convolucionais. A segunda alteração consiste em substituir as unidades ReLu entre as convoluções mascaradas da PixelCNN original pela unidade de ativação abaixo:

$$y = tanh(W_{k,j} * x) \odot \sigma(W_{k,j} * x) ,$$

Equação 2

onde σ representa função não-linear sigmóide, *k* é o número da camada, $\odot$ é o produto entre os elementos e $*$ é o operador de convolução. O modelo resultante é a Gated PixelCNN.

Um problema da PixelCNN original é que esta possui um ponto cego que fica indisponível para realizar previsões Oord et al. (2016b). A solução encontrada para resolver esse problema foi de fazer o uso combinado de duas pilhas de redes convolucionais: uma que condiciona na fileira atual, até o pixel que está sendo avaliado(pilha horizontal), e uma que condiciona com todas as fileiras acima(pilha vertical. A pilha vertical não possui uma máscara, o que permite que o campo receptivo desta cresça de forma retangular, sem campo cego algum. Após cada camada, a saída de ambas as pilhas são combinadas, de tal forma que cada camada da pilha horizontal recebe como entrada a saída da camada horizontal anterior e da pilha vertical. De acordo com (OORD et al., 2016b), se, ao invés disso, a saída da pilha horizontal fosse alimenta à pilha vertical, a rede poderia utilizar de informações sobre os pixels à direita e abaixo, o que não é desejado.



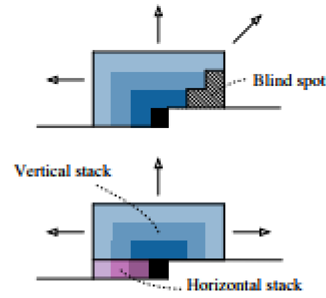


Figura 16. *Visualização da PixelCNN que mapeia a vizinhança de pixels para a previsão do próximo pixel.*

Figura 17. *Ponto cego no campo receptivo da PixelCNN tradicional, que a impede utilizar alguns pixels para a previsão, solucionado com o uso de duas pilhas convolucionais(azul e roxo).*

3.3.2. Conditional PixelCNN

Dada uma descrição de alto nível como um vetor **h**, foi modelada a distribuição condicional *p*(x|**h**) das imagens que correspondem a essa descrição, modeladas pela função abaixo:

$$p(\mathbf{x}|\mathbf{h}) = \prod_{i=1}^{n^2} p(x_i|x_1, ..., x_{i-1}, \mathbf{h})$$

Equação 3

A distribuição condicional é modelada adicionando termos que dependem de **h** às ativações antes das não-linearidades na equação 2:

$$\mathbf{y} = \tanh(W_{k,f} * \mathbf{x} + V_{k,f}^T \mathbf{h}) \odot \sigma(W_{k,g} * \mathbf{x} + V_{k,g}^T \mathbf{h}),$$

Equação 4

onde *k* é o número da camada. Oord et al. (2016b) ressaltam que o condicionamento não depende da localização do pixel na imagem, o que é apropriado contanto que **h** contenha apenas informações do *quê* deveria estar na imagem e não *onde*.

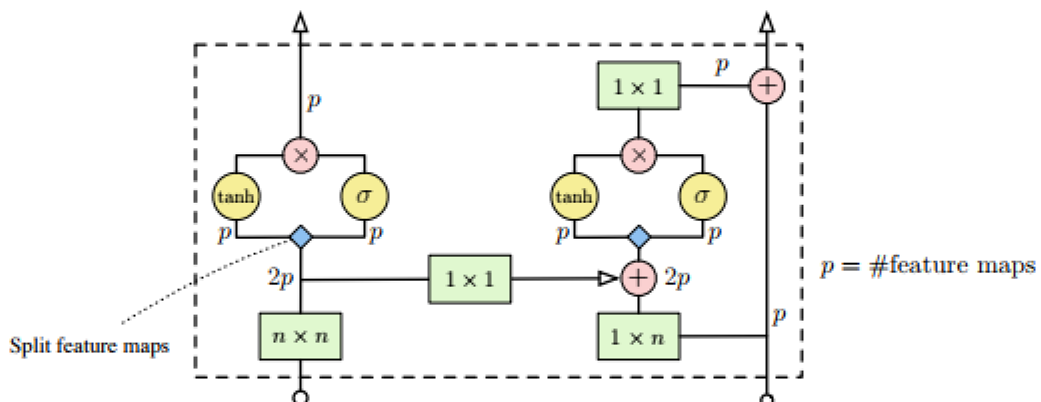


Figura 18. *Uma camada na arquitetura Gated PixelCNN.Operações de convolução mostradas em verde, multiplicação entre elementos em vermelho. As convoluções com*

W_f e W_g da equação 2 são combinadas em uma só operação, mostrada em azul, que separa os 2p mapas de características em 2 grupos de p.

De acordo com Oord et al. (2016b), devido à capacidade das PixelCNN de modelar diversas distribuições $p(x|\mathbf{h})$, é possível usá-las como decodificadores em arquiteturas neurais já existentes, como os auto-codificadores.

3.4. OTIMIZANDO REDES NEURAIS PARA A GERAÇÃO DE IMAGENS

A segunda abordagem segue o trabalho de Tieleman (2014). A idéia proposta é de integrar conhecimento específico de domínio com um sistema de aprendizagem. No contexto de processamento de imagens, a abordagem mais comum que se vê é de usar redes convolucionais com *pooling* (TIELEMAN, 2014). Em reconhecimento de áudio, usa-se a Transformação de Fourier, transformação Cepstral e, às vezes, convolução ao longo do tempo (LANG et al., 1990), para citar algumas técnicas. Isso tudo é exemplo de incorporar o conhecimento de domínio já no sistema de reconhecimento. Tieleman estuda uma alternativa: incorporar esse conhecimento no sistema de geração que, por sua vez, pode guiar o sistema de reconhecimento. Essa abordagem chama-se *síntese por análise* (HALLE & STEVENS, 1962; HALLE & STEVENS, 1959).

Trabalhos recentes envolvendo transformações de auto-codificadores começaram a introduzir mais idéias de arquitetura de redes em sistemas semelhantes aos auto-codificadores (HINTON et al., 2011). O propósito de Tieleman (2014) não era encontrar uma solução, propriamente dito, mas, sim, realizar uma prova de conceito, que é a de usar um sistema de geração para treinar um sistema de reconhecimento.

3.4.1. O Codificador

A arquitetura geral do codificador será a de uma DNN *feedforward* determinística, com a entrada sendo a imagem original e a saída uma descrição do desenho vetorial da imagem. A rede possui quatro camadas, uma de entrada, três ocultas e uma de saída, com conexões diretas das duas primeiras camadas ocultas à camada de saída, com o intuito de evitar o problema do desaparecimento do gradiente (TIELEMAN, 2014). A figura 19 mostra a arquitetura geral do codificador:

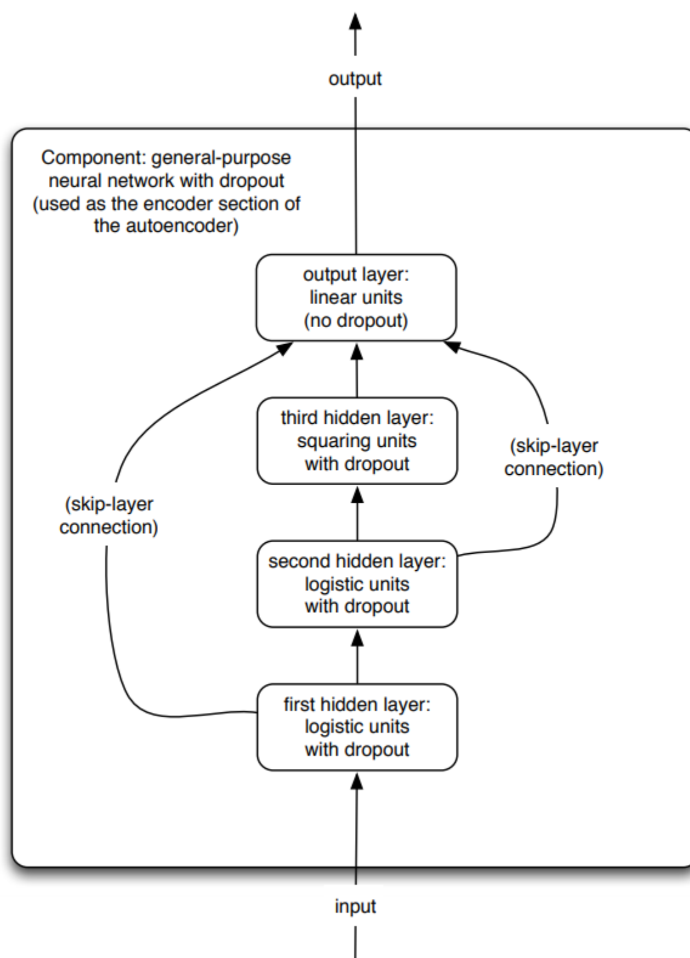


Figura 19. *Arquitetura geral da rede neural utilizada como codificador (TIELEMAN, 2014).*

Cada camada oculta era composta por 666 unidades, com as duas primeiras camadas usando a não-linearidade logística e a terceira usando a não-linearidade quadrática.

Tieleman ressalta que não foi investido muito esforço na escolha desta configuração, pois o foco do trabalho era na parte do decodificador, e o codificador funcionou bem o suficiente para os experimentos. Tieleman ainda fala que experimentou com um codificador com uma camada oculta (mas com o mesmo número total de unidades, todas logísticas) em cima dos dados MNIST e funcionou tão bem quanto, sugerindo que, de fato, o decodificador era a parte crucial de toda a estrutura.

Para o projeto da linguagem do código e do decodificador, Tieleman trabalha com o conceito de "cápsulas", inicialmente introduzidas em (HINTON et al., 2011). No contexto do decodificador, cápsulas servem o mesmo propósito que as unidades na camada de código de um auto-codificador, apesar de ser mais sofisticado. Enquanto que uma unidade trabalha com apenas um valor, cápsulas trabalham com conjuntos de valores, e o projetista impõe uma interpretação em cima desses valores.

Cápsulas representam componentes de uma imagem (como um objeto) e possuem 3 valores (que Tieleman chama de "pose"), *x*, *y* e intensidade (também interpretável como brilho ou claridade). Tieleman ainda chama as configurações que especificamente incluem um valor para claridade de "iPose". Além dessa configuração, cada cápsula possui um "template" de imagem (do componente), associado a si

mesmo. O template final de cada cápsula é aprendido, junto com o modelo. Uma vez estabelecido, cada template é constante. No decodificador, cada cápsula copia seu template na saída do modelo (imagem final), na posição e intensidade determinadas pela *pose*. Tieleman ressalta que o template de uma cápsula não vem do codificador. A cada imagem de entrada, o codificador traduz seus componentes em *poses* e o decodificador aplica as transformações (posição e intensidade) de cada template apropriado, de acordo com as *poses* recebidas, para ser montada na imagem final, como mostra a imagem abaixo:

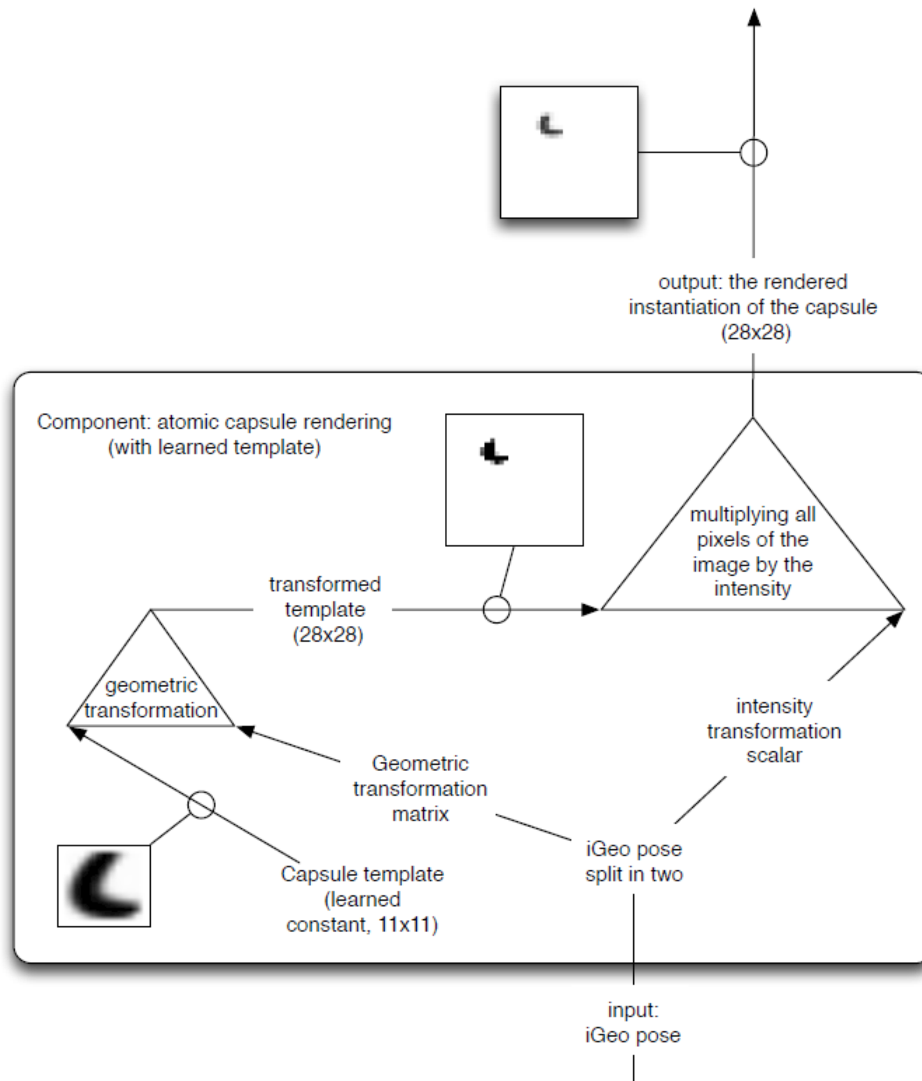


Figura 20. *Arquitetura geral da rede neural utilizada como decodificador (TIELEMAN, 2014).*

Em seguida, Tieleman incrementa as cápsulas com a capacidade de realizarem transformações/funções afins, ou seja, permite aplicar seus templates com transformações escalares ao longo de dois eixos, rotação e cisalhamento.

Um conceito muito importante nessa próxima etapa, diz Tieleman, é o de quadros de coordenadas(do inglês, *coordinate frames*). No escopo desta abordagem, o sistema de coordenadas utilizado é o de duas dimensões e não três, como é típico com computação gráfica. Em uma imagem, existem vários quadros de coordenadas: uma para cada objeto(atômico ou composto) e uma para a imagem como um todo

(neste caso, chamado de *world frame*, por ser o quadro “global”). Cada quadro possui uma tupla (chamada, aqui, de descrição de localização), composta pelos eixos do mundo - aqui, *x* e *y*(posição). Podemos traduzir uma descrição para para outra, como a de um objeto para a do mundo. Para isso, basta pegar o vetor de coordenadas do objeto e multiplicar por uma matriz de transformação *{objeto → mundo}*. O contrário pode ser obtido através de *{objeto → mundo}* = *{mundo → objeto}* $^{-1}$. Adicionalmente, se quisermos saber a posição de um segundo objeto(um observador, por exemplo) relativa ao primeiro, basta multiplicar o vetor de coordenadas do mundo, *{mundo → observador}* e, assim, multiplicar *{objeto → mundo}* por *{mundo → observador}* que, temos *{objeto → observador}*.

Segundo Tieleman, podemos usar esse mesmo raciocínio no auto-codificador. Cada template representa um objeto e são, em relação a seu próprio quadro, de tamanho “1” - um pixel do template que está bem no meio está na posição (0.5, 0.5) das coordenadas do template.

A parte geométrica da *pose* (tamanho, posição, etc) descreve a transformação entre o quadro do objeto e o quadro do mundo(imagem). Vale ressaltar que, para o mundo, seu quadro também é de uma unidade - um pixel da saída que está bem no meio da saída está na posição (0.5, 0.5) das coordenadas do mundo.

Em seguida foi necessário determinar como representar a transformação. De acordo com Tieleman, a maneira que melhor funcionou na hora da modelagem foi de descrever a transformação final na forma de uma sequência, com cada passo composto de qual transformação aplicar e por quanto(algo como: rotacionar 30º sentido horário, cisalhar 1% pela direita e escalar +120% horizontalmente). Segundo Tieleman, experimentos posteriores mostraram, inclusive, que esse tipo de representação se mostrou mais fácil de ser manipulada pela rede neural do que outras representações propostas.

Essas transformações são representadas na forma de multiplicações sucessivas de matrizes e, dada a natureza de algumas dessas transformações, não são comutativas, portanto, a ordem das mesmas é importante. Com alguns testes, Tieleman argumenta que, para realizar a transformação *{objeto → mundo}* a ordem que mais pareceu ser conveniente para a rede foi {*rotacionar → cisalhar → escalar → transladar}*. Outro ponto que Tieleman levanta é que inversão de matrizes(divisões) são inconvenientes para redes, pois podem acarretar em comportamentos indesejados nos gradientes. A solução encontrada é de simplesmente multiplicar as matrizes de transformação em ordem inversa, podendo realizar transformações do tipo *{mundo → objeto}* sem utilizar divisões.

A próxima etapa é determinar como combinar a contribuição de cada cápsula. Entre as abordagens testadas por Tieleman, a que funcionou melhor consistia em avaliar o quanto que cada cápsula contribuía para determinado pixel de saída e, em função disso, selecionar a cápsula com a maior contribuição.

Com tudo isso feito, pode-se começar a construir cápsulas compostas, que representam objeto compostos. Até então, tinha-se o que Tieleman chamou de Cápsulas Atômicas(do inglês, *Atomic Capsules*), ou ACs. Começa-se, agora, a lidar com CCs(do inglês, *Composite Capsules*).

Com mais esse nível de modelagem, as ACs não possuem mais um referencial em relação ao mundo e, sim, ao objeto a qual pertencem, visto que um conjunto de componentes de um objeto mantém uma *pose* constante dentro do contexto do objeto(assumindo um objeto rígido) ou aproximadamente igual(para objetos menos rígidos). Se quisermos saber a *pose* da AC em relação ao mundo, basta realizar os passos de multiplicação de matrizes no início desta seção.

No auto-codificador, a CC terá uma *pose* “CC-in-world”, que descreve sua orientação relativa ao mundo, enquanto que a AC terá uma *pose* “AC-in-CC”, que descreve sua orientação relativa à CC.

ACs são rígidos por natureza, visto que seus templates são constantes. Segundo Tieleman, se quisermos representar objetos rígidos, com o que foi descrito acima já basta, mas, com objetos sujeitos a certa deformação, é necessário incrementar as capacidades do CC para representar *distorções*, através de algumas variáveis adicionais. Estas variáveis podem descrever como que as *poses* AC-in-CC dos componentes de uma CC variam do que é normal para aquela CC. Uma CC é, portanto, definida por uma função aprendida que mapeia esse componente de distorção de sua *pose* para as *poses* das AC-in-CC de seus componentes.

A próxima preocupação, segundo Tieleman, é a de permitir que CCs compartilhem ACs. Inicialmente, parte-se do pressuposto que ACs fazem parte de um conjunto bem-definido e isolado. Se existem múltiplos CCs, cada um tem seu próprio conjunto de ACs, de modo hierárquico, com o ponto negativo de resultar em um número muito grande de ACs e, consequentemente, maior tempo de computação. De acordo com Tieleman, foi possível contornar esse problema, sem acarretar em um aumento muito significativo de computação: realiza-se uma espécie de “mistura” de CCs, gerenciada por um gerente de mistura. Cada CC continua tendo suas próprias ACs. A condição para que isso funcione é de que o gerenciador de mistura selecione apenas uma CC, de tal forma que apenas suas ACs tenham algum efeito.

O gerenciador deve tomar decisões diferenciáveis, portanto, durante a aprendizagem, o gerenciador não ativa apenas uma das misturas, mas, sim, todas as misturas e utiliza pesos para realizar aprendizagem em cima de dos componentes mais adequados. Segundo Tieleman, isso acaba criando problemas para otimizações. A solução encontrada foi de incentivar o gerenciador a dar fortes preferências para as misturas com maior confiança na maior parte do tempo.

Uma das vantagens encontradas na modelagem das misturas é a de conseguir resolver ambiguidades, diz Tieleman. Se tivermos, por exemplo, uma CC especializada em gerar quadrados e outra CC especializada em gerar losangos, ambas são capazes de gerar a mesma imagem, diferenciando-se apenas em como cada uma aplica as transformações geométricas.

4. AVALIAÇÃO DE REDES NEURAIS PARA O DESENVOLVIMENTO DA FERRAMENTA

A criação de componentes gráficos de jogos é um processo custoso, especialmente no quesito empenho. Exige domínio de ferramentas de modelagem ou habilidade artística (que leva muito tempo para desenvolver se ainda não a tiver) por parte do usuário - no contexto desse problema, o desenvolvedor de jogos independente. Muitas ferramentas modernas já oferecem módulos de geração de componentes, capazes de gerar inúmeras instâncias de um determinado objeto (i.e., uma árvore) em pouquíssimo tempo. No entanto, o desenvolvedor está sujeito às opções que a ferramenta oferece. Ocasionalmente, essas ferramentas permitem que a própria comunidade desenvolva extensões para propósitos específicos, mas a ferramenta está sujeita a ficar cheia de módulos instalados que somente são utilizados ocasionalmente, deixando a ferramenta muito pesada.

Existem mercados virtuais, também, de componentes gráficos, oferecendo tanto componentes pagos quanto componentes livres de custo, porém, limitados pelo

que o criador original fez. Em ambos os casos, o desenvolvedor pode se deparar com uma parede bem rapidamente. Ter uma ferramenta que eliminasse a limitação de ter que usar apenas o que fosse explicitamente criado por outros e permitisse que o próprio desenvolvedor criasse, facilmente e rapidamente, as próprias soluções e pudesse gerar inúmeros componentes gráficos, cada um diferente do anterior, cortaria um tremendo custo do processo artístico do desenvolvedor independente (e de qualquer um que usufruísse da ferramenta).

Considerando os objetivos específicos propostos para este trabalho, podemos começar a elaborar melhor a estrutura e os processos para realizar os experimentos para que possamos, logo em seguida, desenvolver uma ferramenta de geração de imagens. De um modo geral, após os devidos estudos de soluções já existentes, os objetivos se resumem a comparar abordagens com redes neurais de diferentes arquiteturas para o reconhecimento e geração de imagens. Uma vez realizada as avaliações dos resultados, a arquitetura julgada a mais apropriada será escolhida para fazer parte da ferramenta a ser desenvolvida.

Em função desses objetivos, podemos separar o experimento em 6 etapas:

1. Estabelecimento de um mecanismo para aquisição de conjuntos de imagens para treino;

2. Tratamento das imagens adquiridas da primeira parte, conforme for necessário;

3. O treinamento da rede neural reconhecedora de objetos, realizado em cima das imagens tratadas na etapa anterior;

4. O treinamento da rede neural geradora, condicionada pela classe do objeto que se deseja gerar;

5. Geração de imagens pela rede neural geradora;

6. Avaliação dos resultados - comparar a qualidade das imagens geradas, tempo de treinamento, consumo de memória e tempo de geração das imagens de cada rede;

Já que diferentes abordagens serão estudadas, esse processo (ou apenas a etapa relevante) se repetirá para cada uma das abordagens a serem comparadas. A seguir, serão detalhadas melhor cada uma das etapas citadas.

A primeira etapa consiste em organizar os conjuntos de dados para realizar um pré-processamento. Pode ser montado manualmente ou, se possível, encontrar conjuntos já prontos (inclusive com o pré-processamento já realizado). Podemos, então, seguir para a segunda etapa, que é a de pré-processamento, de fato. No caso deste experimento, estamos lidando com imagens, então considera-se que estas devem estar todas redimensionadas ou recortadas para o mesmo tamanho e com o objeto de interesse o mais próximo do centro da imagem possível. Uma vez que o conjunto de dados estiver organizado de forma satisfatória, é possível passar para a

terceira etapa, a da rede neural reconhecedora. As imagens serão alimentadas para esta rede para que esta tente aprender uma representação dos objetos presentes nas imagens fornecidas. Cada imagem está vinculada a uma classe, que informa qual o tipo de objeto está presente na mesma. Com isso, pode-se vincular uma representação aprendida a uma determinada classe. Uma vez treinada a rede reconhecedora, prosseguimos para a quarta etapa, onde será realizado o treinamento da segunda rede neural, a geradora, para fazer o caminho inverso da primeira: em função da representação e classe de objeto fornecidos, a rede geradora deverá poder reconstruir (aproximadamente), uma imagem da classe apropriada que corresponda à representação fornecida. Com isso, os resultados da rede geradora podem ser alimentados a um pequeno módulo para, de fato, salvar a imagem gerada, na quinta e última etapa.

Um visão mais geral das etapas do experimento é ilustrada na figura 21, abaixo, e será descrita em mais detalhes nas próximas subseções.

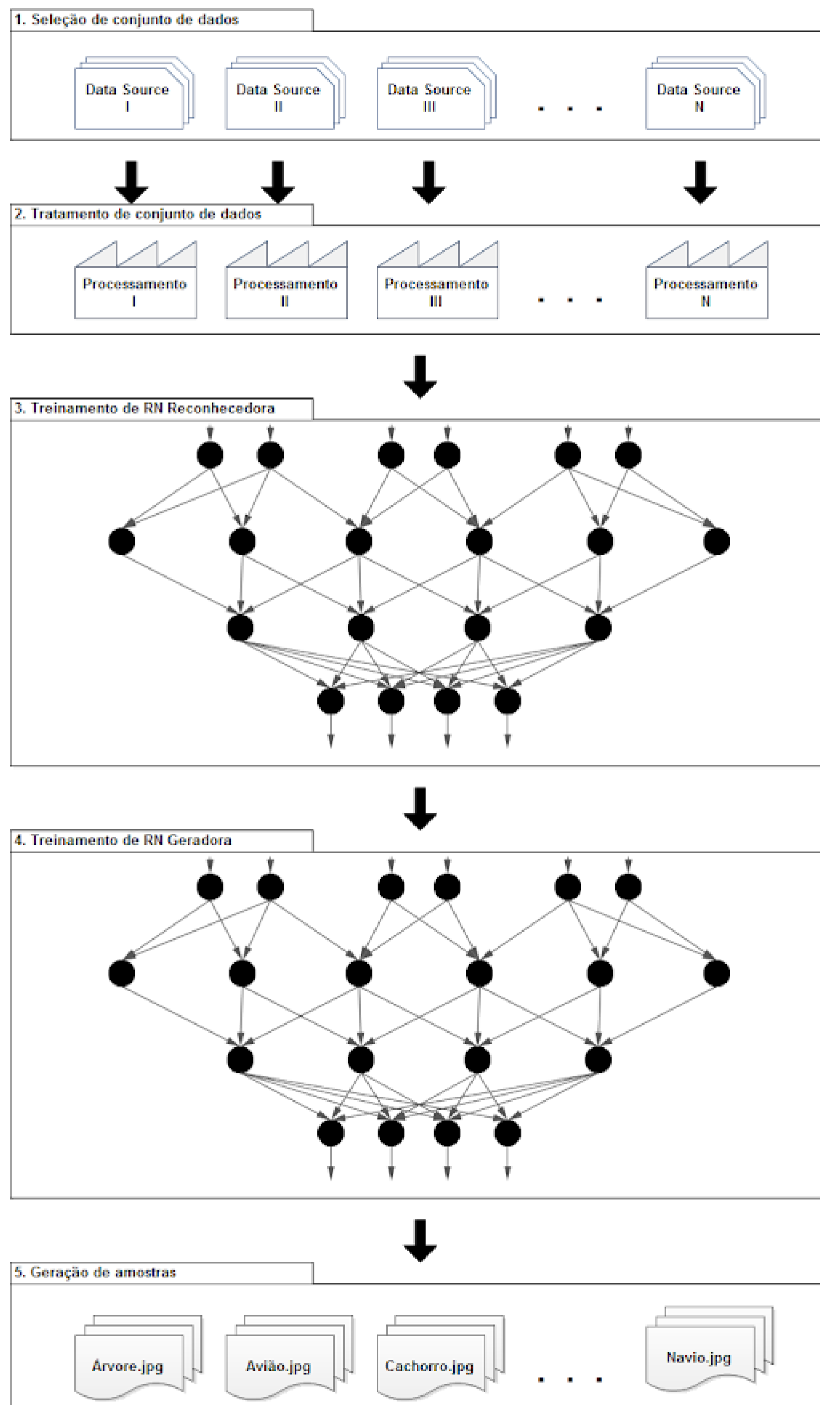


Figura 21. *Ilustração dos componentes do experimento realizado.*

4.1. PRÉ-PROCESSAMENTO E ORGANIZAÇÃO DOS DADOS

Nesta etapa, como o objetivo é utilizar fontes arbitrárias de imagens para o treinamento da rede neural reconhecedora, é de se esperar que exista bastante variabilidade entre as imagens obtidas em quesitos como a resolução da imagem e a posição do objeto de interesse na mesma. Para não aumentar demais a complexidade dos experimentos, assume-se que a estrutura da rede (número de neurônios e conexões) é estática, o que significa que os dados de entrada para a rede devem passar por uma etapa de pré-processamento para regularizá-los, o que consiste em deixar todas as imagens com a mesma resolução e, na medida do possível, deixar o objeto de interesse o mais centralizado possível. Adicionalmente, será necessário, onde aplicável, separar entre imagens coloridas e imagens em preto e branco.

Para poder obter resultados comparáveis entre as abordagens e com o atual estado da arte, foi decidido realizar os testes em cima de conjuntos de dados já bem estabelecidos e de fácil aquisição. Em todos os conjuntos, as imagens já vem normalizadas em relação a tamanho e posição do objeto de interesse, além de metadados indicando qual a classe do objeto presente na imagem. Os conjuntos de dados escolhidos foram o **MNIST**, o **Cifar10** e o **ImageNet**.

MNIST

O conjunto de dados MNIST (Modified National Institute of Standards and Technology) é uma compilação de milhares de imagens de dígitos escritos à mão, aproveitadas do conjunto original do NIST. Ficou bastante popular como o conjunto de dados padrão para se utilizar em treinamento de sistemas de processamento de imagens e de aprendizagem de máquina.

Este conjunto contém 60000 imagens de treinamento e 10000 imagens de validação, todas 28x28 pixels, onde metade do conjunto de treinamento e do conjunto de validação é composta por imagens do conjunto de treinamento original, e a outra metade do conjunto de treinamento e de validação são do conjunto de validação original.

Cifar10

O conjunto de dados CIFAR-10 (Canadian Institute for Advanced Research) é um subconjunto de um conjunto (bem) maior de 80 milhões de imagens - o “tiny images dataset”, que também se popularizou para tarefas de aprendizagem de máquina e visão computacional.

Este conjunto contém 60000 imagens 32x32 pixels, coloridas, associadas a 10 classes diferentes. Estas classes representam aviões, carros, pássaros, gatos, veados/cervos, cachorros, sapos, cavalos, navios e caminhões, distribuídas igualmente (6000 imagens de cada classe).

ImageNet

Outro conjunto que se popularizou para ser utilizado como referência na qualidade de sistemas de computação visual e aprendizagem de máquina, com foco em reconhecimento de objetos.

Este conjunto não contém imagens diretamente e, sim, por 14 milhões de URLs de imagens, manualmente anotados com o objeto principal presente nas mesmas. Devida à variedade de imagens neste conjunto (tamanho e posição do objeto na imagem), será utilizado um conjunto com as mesmas imagens que o original, porém redimensionadas para 32x32 pixels, e cortadas para centralizar o objeto em questão. Vale ressaltar que também existe uma grande variação na distribuição de quantidade de imagens por classe.

Para os experimentos realizado foi utilizado um framework de aprendizagem de máquina, o TensorFlow. Apesar de poderem ser baixados de outras fontes manualmente (ou através de algum script), já vem incluído no pacote principal do TensorFlow, para facilitar a aquisição e rápida manipulação dos dados, uma API para baixar e carregar para as redes neurais construídas os conjuntos MNIST e Cifar10. Cada conjunto já vem com as imagens normalizadas (tamanho da imagem e posição do objeto de interesse) e devidamente separadas em conjunto de treino e de teste, assim como os rótulos de classes associados a cada imagem. Para estes, foi desnecessário passar por uma etapa de pré-processamento.

O ImageNet é um conjunto tratado de maneira diferente. Para o ImageNet, é necessário baixar os recursos manualmente (ou através de algum script) do site http://image-net.org/. Através do site é possível baixar as imagens (ou URLs das mesmas), assim como os metadados que informam a posição do objeto de interesse na imagem. No caso do ImageNet, as imagens têm tamanhos diferentes e, portanto, precisam passar por uma etapa de pré-processamento. Fazendo uso dos metadados dos **Object Bounding Boxes**, como são chamados, que informam a posição dos objetos na imagem, bastaria utilizar uma biblioteca de processamento de imagem para realizar o recorte e redimensionamento adequados de cada imagem. Por exemplo:

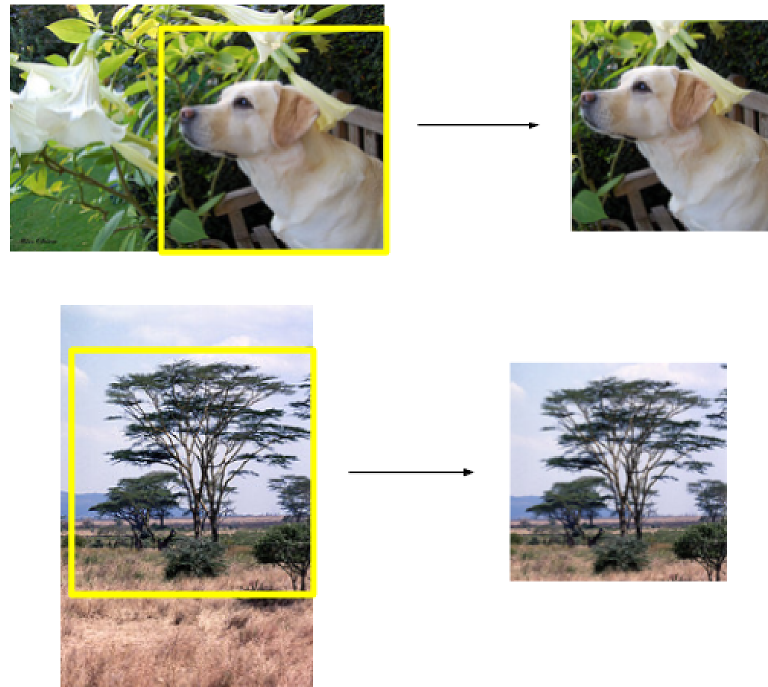


Figura 22. *Exemplo de pré-processamento a ser realizado nas imagens do conjunto ImageNet: recorte e centralização e, em seguida, redimensionamento.*

Adicionalmente, é necessário fazer o próprio mapeamento das imagens baixadas e as classes associadas. Entretanto, em função de restrições de tempo, não

foi realizado o pré-processamento, assim como os experimentos previstos, para o conjunto do ImageNet.

4.2. AS REDES

4.2.1. Rede Reconhecedora (Codificadora)

A primeira rede neural na ferramenta será utilizada para reconhecer um objeto específico em uma posição arbitrária dentro de uma imagem qualquer. Pelo fato de os conjuntos de dados utilizados já estarem com as imagens normalizadas e organizadas por classe do objeto, o papel desta rede se reduz ao de um codificador. O que isso significa é que o objetivo da rede é de aprender representações ou codificações para cada classe de objeto presente nos dados apresentados. Esta é apenas uma parte do conjunto que é formado com a segunda rede.

Como explicado acima, esta rede se responsabiliza por mapear uma entrada de dados para um vetor de representação. No caso dos experimentos realizados, o codificador vai receber, como entrada (desconsiderando parâmetros de configuração da rede em si), um vetor $1 \times largura \times altura$ que representa a imagem planificada (ou *n* vetores, 1 para cada imagem, para aproveitar paralelismo) e um vetor em codificação *one-hot* que representa a classe da imagem de entrada. O resultado final, após o treino, é o vetor de representação para a classe treinada - e é este vetor a saída da rede codificadora que, por sua vez, é fornecido como entrada para a rede decodificadora, junto com o vetor de codificação da classe da imagem que se deseja gerar.

4.2.2. Rede Geradora (Decodificadora)

Enquanto a primeira rede da ferramenta é a codificadora, a segunda ganha o papel de decodificador, ou seja, esta aprende, a partir de uma determinada representação de dados, a gerar uma saída que esteja de acordo com as características dos dados utilizados para treinar a rede codificadora.

Não tem restrições quanto à arquitetura da rede para ser utilizada para o codificador e o decodificador. A arquitetura DRAW, por exemplo, faz uso de RNNs (rede neurais recorrentes), tanto para o codificador quanto para o decodificador. Adicionalmente, é possível combinar arquiteturas diferentes entre o codificador e o decodificador, dando origem aos auto-codificadores variacionais.

Utilizando as duas redes em conjunto, temos o que é chamado de um auto-codificador. Colocando de outra maneira, tendo as transições φ e ψ tal que:

$$\varphi : \chi \rightarrow \beta$$
$$\psi : \beta \rightarrow \chi$$

Onde φ é a transição que representa o mapeamento dos dados para uma representação aprendida pela rede codificadora e ψ é a transição que representa o mapeamento de uma representação aprendida pela rede decodificadora para gerar elementos dos dados utilizados para o codificador.

Tendo como entrada o vetor de representação obtido com a rede codificadora e o vetor de codificação da classe da imagem que se deseja gerar, o objetivo da rede decodificadora é tentar reconstruir o objeto que corresponde à representação fornecida. A arquitetura geral simplificada do auto-codificador é mostrado na figura 23, abaixo:

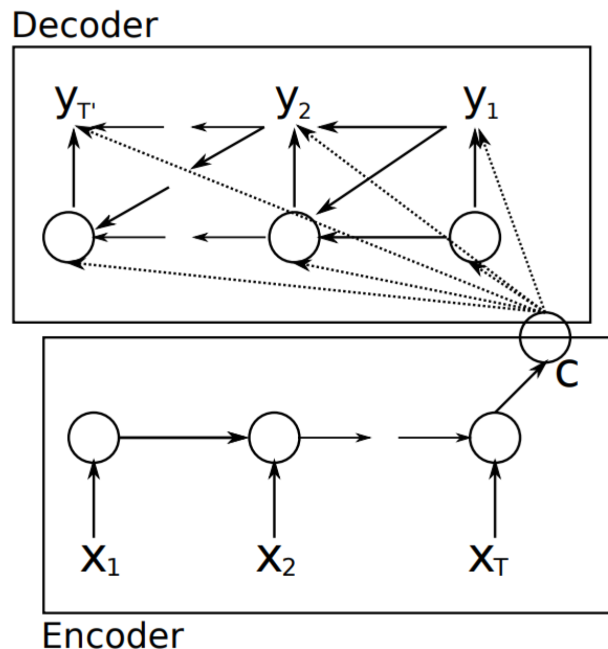


Figura 23. *Imagem do artigo "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", por CHO, Kyunghyun et al.*

4.3. GERAÇÃO DE AMOSTRAS

Esta parte do experimento se resume a utilizar qualquer biblioteca manipulação de as imagen. Para gerar os dados da imagem, basta a alimentar à rede decodificadora o vetor de representação da classe da imagem que se deseja gerar, que a saída da rede, para cada pixel, será a previsão da mesma sobre qual deverá ser o valor desse pixel. Em seguida, basta utilizar a biblioteca escolhida para efetivamente escrever em disco a imagem. No caso deste trabalho, foi utilizada a biblioteca de manipulação de imagens *Pillow*, para Python.

4.4. DESCRIÇÃO DO EXPERIMENTO

Para a realização dos experimentos, foram selecionadas 3 arquiteturas de redes neurais para serem comparadas. Estas são a DRAW, a Conditional PixelCNN e a DCGAN. Todas as três implementações encontradas foram construídas em cima do framework de aprendizagem de máquina, TensorFlow.

Adicionalmente, para cada arquitetura, serão realizados testes com diferentes conjuntos de parâmetros, a fim de determinar qual arquitetura consegue atingir os melhores resultados com o menor uso de recursos (tanto de tempo quanto de

máquina). Como o espaço de busca das melhores combinações para cada arquitetura é muito grande (junto com os tempos de treinamento envolvidos), alguns parâmetros serão mantidos fixos para todos os experimentos de todas as arquiteturas.

Devido a restrições de tempo, não foi possível realizar experimentos com o conjunto do ImageNet e apenas uma das abordagens foi testada em cima do Cifar10.

4.4.1. Ambiente de Teste

A fim de se ter uma noção do hardware que possibilitou os resultados obtidos, os detalhes da máquina utilizada são:

- MacBook Air (Mid-2012)
- 8GB de memória RAM 1600 MHz DDR3
- Processador Intel i7 2GHz

O TensorFlow fornece uma API que possibilita delegar o processamento e o uso de memória para a(s) placa(s) gráfica(s) da máquina, o que permite aproveitar um maior paralelismo na hora de treinar a rede. No entanto, no momento o framework apenas dá suporte para placas gráficas da NVIDIA. Como a máquina utilizada não possuía placa gráfica compatível, todo o processamento e uso de memória ficou com o CPU e memória principal da máquina.

4.4.2. Método de Avaliação

Para cada conjunto de dados, cada arquitetura será avaliada em função da qualidade das imagens geradas, junto com o tempo necessário para treinar e gerar as mesmas e memória e o consumo de memória. A qualidade em si será determinada conforme as imagens possuírem certas características esperadas da classe associada e não possuírem certas aspectos indesejados. Será detalhado, abaixo, os critérios desejados e indesejados de cada classe, em cada conjunto de dados. Em seguida, será exposta a função para atribuir a nota final para cada arquitetura.

MNIST

Para o conjunto de dados MNIST, serão avaliados os seguintes critérios, em cima das 10 classes presentes, considerados positivos:

- **Traço contínuo:**
  Aqui é avaliado se um número é gerado sem interrupções em seus traços. Por exemplo, o número “**0**” pode, ocasionalmente, faltar fechar o círculo perfeitamente. Neste caso, essa instância não só deixa de ganhar pontos como também perde pontos;
- **Traço forte:**
  Avaliado quanto à espessura e força (ou solidez) dos traços - resultado de pixels pretos entre os pixels brancos que, quando muito misturados, pode dar uma impressão de translucidez ou que foi simplesmente gerado um borrão. Por exemplo, o número “**1**” pode ser gerado muito fino ou o até com uma espessura decente, mas parecendo apenas um borrão vertical;

- **Facilmente identificável:**
  Avaliado quanto à facilidade de identificar o número gerado. Pode gerar dúvida se era para ser gerado um "**8**" ou um "**3**", ou um "**4**" ou "**9**", se possuir traços incompletos nos lugares "certos";
- **Forma correta:**
  Referente à forma geral esperada, em si, da classe. Um círculo ou elipse para o "**0**", dois círculos, um em cima do outro, para o "**8**";
- **Variedade:**
  Alguns números podem ser escritos de formas levemente diferentes, como o "**2**", que pode ser escrito dessa forma, entras as aspas, e com um estilo mais cursivo, "**ᒿ**", ou o "**7**", que pode ou não ter um traço paralelo à base passando pelo equador do número. Como isso pode ser fortemente influenciado pelos dados de treinamento, a ausência de variedade significativa, como o estilo em si da classe, não será taxado como ponto negativo. Será avaliado em cima do conjunto de elementos da mesma classe, ao invés de elementos individuais, como os outros critérios;

A cada item será atribuído um valor de 2 pontos. Serão avaliados, também, alguns aspectos considerados negativos:

- **Traço inacabado:**
  A consequência de não possuir as características apontadas pelo primeiro item positivo;
- **Presença de elementos inesperados:**
  Avaliado quanto à presença de anomalias significativas, como, por exemplo, borrões, riscos inesperados (i.e: um traço cruzando um número "**8**");
- **Forma incorreta ou imprecisa:**
  Referente à forma geral esperada. Por exemplo, um "**6**" pode fazer uma volta muito fechada, parecendo a letra "**C**", por exemplo. Ainda usando este exemplo, aqui entram outros pontos, pois pode surgir a dúvida de se foi um "**6**" mal-feito ou um "**0**" com traço incompleto;

A estes itens será atribuído o valor de -2 pontos. A nota atribuída a cada exemplo gerado de cada classe será na forma do somatório dos itens positivos e negativos. Ocasionalmente, dependendo do tamanho do modelo e do tempo de treinamento, podem ser gerados instâncias completamente irreconhecíveis. Nesses casos, a nota atribuída àquela instância é automaticamente zerada. Note que não existe, necessariamente, um ponto negativo para cada ponto positivo, ou melhor, a ausência de um determinado ponto positivo não acarreta em um ponto negativo. Neste caso, a instância sendo avaliada simplesmente deixa de ganhar pontos. A equação para uma determinada instância fica, então, assim:

$$N_i^c = \sum_{n=0}^{k} 2 \cdot P_n^+ - \sum_{m=0}^{l} 2 \cdot P_m^-$$

Equação de avaliação de cada instância de classe, onde P_n^+ é o *n-ésimo* ponto positivo sendo avaliado, P_m^- é o m-*ésimo* ponto negativo sendo avaliado e N_i^c é a nota de uma determinada instância de uma classe *c*.

Visto que, ao final do treinamento, são gerados múltiplos exemplos de cada classe, será realizada uma simples média aritmética entre as notas de cada instância da mesma classe, ou seja:

$$N_c = avg([N_1^c, N_2^c, ..., N_n^c])$$

Para contemplar os experimentos em que a rede gera um número desigual de instâncias de cada classe, a etapa seguinte para calcular a nota da arquitetura será dada pela média aritmética ponderada das notas de cada classe:

$$N_A = w_avg([N_1, N_2, ..., N_c])$$

Por fim, serão incluídas as pontuações referentes ao tempo necessário para treinar as redes, o consumo de memória e o tempo de geração de imagens. É interessante estudar como atribuir um peso ao tempo de treino e uso de memória, pois é um tanto quanto circunstancial: o ambiente em que é utilizada a ferramenta e o custo monetário dos recursos computacionais no momento. No caso do ambiente onde foi realizado este experimento, foi utilizado CPU e memória principal da máquina. Por um lado, tem-se memória para modelos maiores (potencial de resultado melhores) e tamanho de lotes (batch size) maiores (maior potencial de paralelização), além de ser, hoje em dia, relativamente barata. Enquanto isso, o CPU, além de ter poucos núcleos, é de uso genérico. Redes neurais são excelentes estruturas para se aproveitar de paralelismo, podendo resultar em tempos de treinamento menores, portanto, tendo apenas acesso a um CPU acaba desperdiçando esse potencial de paralelismo, resultando, infelizmente, em tempos de treinamento maiores.

Por outro lado, pode-se ter acesso a placas gráficas (assume-se placas compatíveis com o TensorFlow, para o contexto deste experimento). Placas gráficas possuem processadores especializados em operações matemáticas - praticamente todo o peso computacional de redes neurais - e, geralmente, com muitos núcleos. Consequência imediata disso é um maior aproveitamento de paralelismo - o TensorFlow abstrai para o desenvolvedor a necessidade de gerenciar threads e distribuir carga entre as placas gráficas e seus respectivos processadores/núcleos. No entanto, os preços de placas gráficas com memória comparáveis à memória principal (no que diz respeito a GB) são mais agressivos. No brasil, um placa gráfica da NVIDIA (nova) com 2GB pode variar desde R$400,00 reais para mais de R$1000,00 (preço varia entre modelos com divergência em outras especificações e entre fabricantes diferentes), ou seja, maior aproveitamento de paralelismo, porém em cima de modelos menores (o que pode resultar em resultados insatisfatórios). Placas gráficas da NVIDIA de 8GB (para ficar comparável à quantidade de memória principal) já

começam a aparecer a partir de R$1600,00 (e com mais núcleos), permitindo, assim, modelos maiores, sem adicionar um custo de tempo necessariamente linear em relação ao tamanho do modelo (até atingir o limite de paralelismo aproveitável pela arquitetura da rede).

Considerando que o ambiente onde foi realizado os experimentos fez uso de CPU e memória principal, o tempo de treino terá um peso maior e o consumo de memória terá um peso menor, pouco significativo. Outro ponto a ser considerado é quanto ao tempo de geração das imagens: como será visto adiante, os tempos de geração não são muito significativos, levando apenas alguns segundos para todas as arquiteturas, portanto, o peso atribuído a este critério também será menor.

Adicionalmente, visto que esses critérios - tempo de treino, memória utilizada e tempo de geração - não possuem um limite superior e podem ter diferenças muito grandes, os valores serão normalizados para o intervalo **[0, 1]**, pegando o maior valor observado quando este varia durante o treino (no caso, a memória utilizada, pois esta oscilava ao longo do tempo), já que isso pode vir a ser um fator limitador. Para evitar valores iguais a **0** e **1**, após normalizá-los, o cálculo de normalização será realizado da seguinte forma:

$$f(value, max, min) = (value - (min - 1)) \div ((max + 1) - min)$$

Onde *max* é o maior valor da lista sendo avaliada (i.e., entre todos os valores de tempo) e *min* é o menor valor da lista sendo avaliada.

Já que valores menores, nesse caso, são melhores, o valor encontrado será, então, subtraído de **1**. A nota final da arquitetura sendo avaliada será, então, calculada da seguinte maneira:

$$N_F = N_A \cdot (P_T \cdot (1 - T_{N)}) \cdot (P_M \cdot (1 - M_N)) \cdot (P_G \cdot (1 - G_N))$$

onde a nota final N_F é o produto entre a nota da amostragem N_A e os produtos dos pesos de tempo de treinamento P_T, consumo de memória P_M e tempo de geração P_G e os respectivos valores normalizados, com:

- P_T tendo peso 4,0
- P_M tendo peso 1,5 e
- P_G tendo peso 1,1

4.4.3. Parâmetros dos Experimentos

Conforme apresentado anteriormente, são muitas combinações de parâmetros para serem testadas, portanto, alguns parâmetros serão escolhidos para serem constantes ao longo dos experimentos. Os parâmetros escolhidos para serem mantidos fixos são o **otimizador**, o **passo de aprendizagem**, e o **batch size**. Para as funções de ativação, são utilizadas conforme detalhados nos respectivos trabalhos originais.

Épocas Treinadas	Batch Size	Otimizador	Passo de Aprendizagem
25	64	Adam Optimizer	2×10^{-4}

Tabela 2. *Configurações aplicadas em todas as arquiteturas (onde aplicável).*

A escolha inicial de número de épocas a serem treinadas é arbitrária e serve mais como um ponto de partida para cada arquitetura. Algumas arquiteturas, para determinado tamanho de modelo, podem se beneficiar de mais épocas de treinamento para gerarem imagens melhores. Nesses casos, são realizados experimentos com mais épocas e/ou tamanhos de modelos diferentes. Em outros casos, é possível que a rede estabilize antes do planejado, fazendo com que épocas de treinamento adicionais sejam desnecessárias.

O Adam (do inglês, *adaptive moment estimation)* Optimizer (KINGMA & BA, 2015) é um método para otimização estocástica que tem ganhado bastante popularidade no contexto de redes neurais. Isto se dá por ser, de acordo com os criadores, eficiente e por requerer apenas gradientes de primeira ordem com pouca necessidade de memória. O método calcula as taxas individuais de aprendizagem adaptativa para diferentes parâmetros a partir de estimativas do primeiro e segundo momentos dos gradientes.

4.5. ANÁLISE DOS RESULTADOS

4.5.1. DRAW

Arquitetura		Recurrent Neural Network (LSTM)	
Função de ativação		Tahn	
Codificador		**Decodificador**	
Camadas Ocultas	1	**Camadas Ocultas**	1
Neurônios por camada	256	**Neurônios por camada**	256

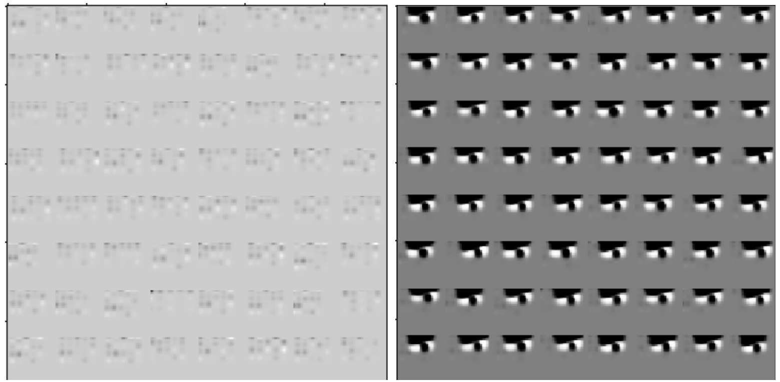
Tabela 3. *Detalhes da arquitetura da rede DRAW.*

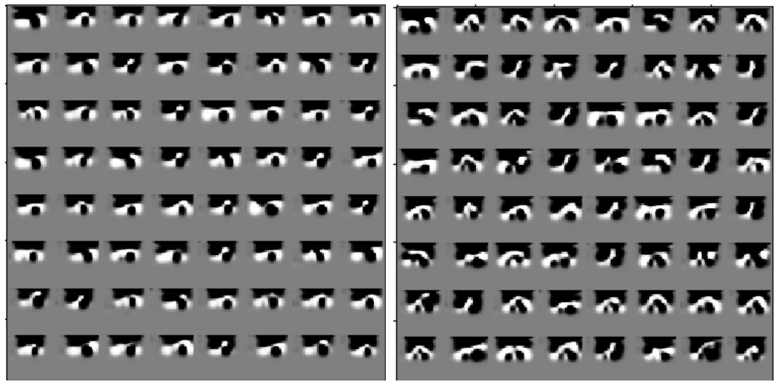
Para esta arquitetura não é utilizado, explicitamente, o conceito de épocas. Ao invés disso, usa iterações para gerar imagens iniciais e, iterativamente, refiná-las.

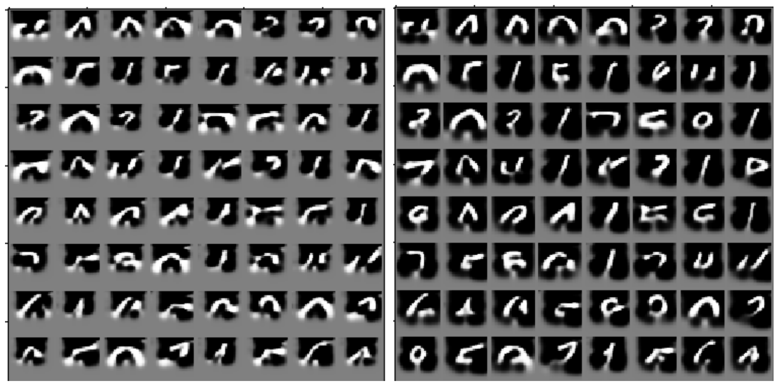
MNIST			
Épocas (para estabilizar)	**Duração - Treino (Minutos)**	**Memória Utilizada (MB)**	**Duração - Geração (Segundos)**
10000	312m	400-500MB	2s

Tabela 4. *Exemplo de resultados obtidos com a arquitetura DRAW.*

Segue, abaixo, a progressão de imagens geradas por esta arquitetura, ao longo das 10000 iterações, da esquerda para a direita, de cima para baixo:







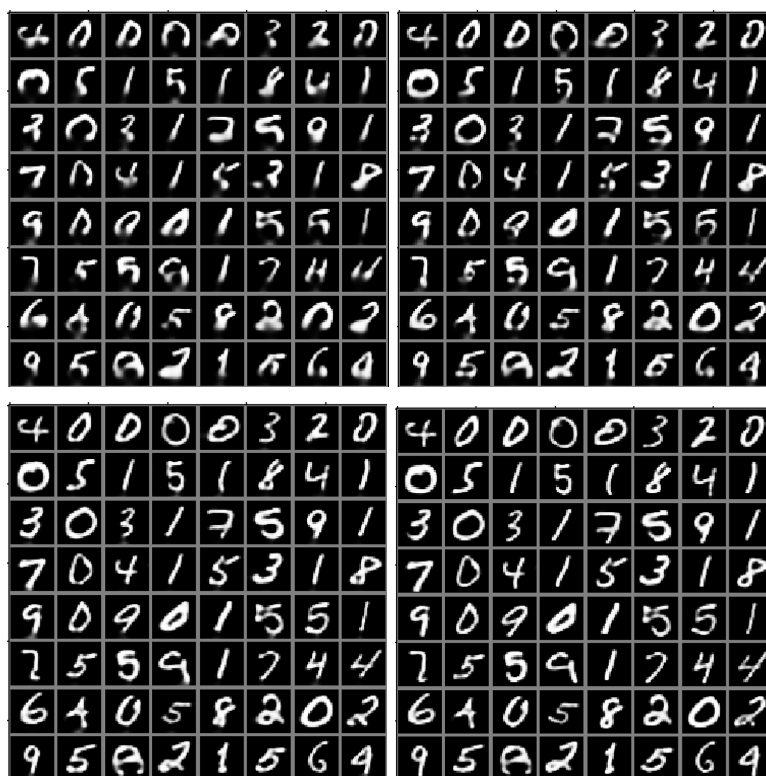


Figura 24. *Exemplos de imagens geradas pela arquitetura DRAW em 10 iterações de melhorias sucessivas.*

Também foram realizados experimentos com até 100000 iterações, mas não se notou melhorias significativas que justificassem o tempo adicional de treinamento.

Analisando a última (décima) imagem gerada, segue a avaliação do que foi gerado:

Classe	Observações	Instâncias	Nota	V. Pond
0	- Traço contínuo x 10 - Traço forte x 10 - Forma correta x 5 - Facilmente identificado x 12 - Variedade - Traço inacabado x 2 - Forma incorreta ou imprecisa x 5	12	**5,16**	**0,9675**
1	- Traço contínuo x 12 - Traço forte x 11 - Forma correta x 12 - Facilmente identificado x 12 - Variedade	11	**8,72**	**1,49875**
2	- Traço contínuo x 4 - Traço forte x 3	4	**7,5**	**0,46875**

	- Forma correta x 4 - Facilmente identificado x 4 - Variedade - Presença de anomalia x 1			
3	- Traço contínuo x 4 - Traço forte x 2 - Forma correta x 4 - Facilmente identificado x 4 - Variedade	4	**7,5**	**0,46875**
4	- Traço contínuo x 7 - Traço forte x 6 - Forma correta x 7 - Facilmente identificado x 7 - Variedade	7	**8,0**	**0,875**
5	- Traço contínuo x 11 - Traço forte x 10 - Forma correta x 11 - Facilmente identificado x 11 - Variedade	11	**8,0**	**1,375**
6	- Traço contínuo x 1 - Traço forte x 1 - Forma correta x 2 - Facilmente identificado x 2 - Traço inacabado x 1	2	**5,0**	**0,15625**
7	- Traço contínuo x 4 - Traço forte x 3 - Forma correta x 3 - Facilmente identificado x 4 - Variedade - Forma incorreta ou imprecisa x 1	4	**7,0**	**0,4375**
8	- Traço contínuo x 1 - Traço forte x 4 - Forma correta x 2 - Facilmente identificado x 4 - Variedade - Traço inacabado x 3	4	**4,5**	**0,28125**
9	- Traço contínuo x 3 - Traço forte x 5 - Forma correta x 5 - Facilmente identificado x 5 - Variedade - Traço inacabado x 2	5	**6,8**	**0,53125**

Tabela 5. *Análise dos resultados com a arquitetura DRAW.*

Média Ponderada (*w_avg*)	**7,06**

4.5.2. DCGAN

Detalhes da Arquitetura

Arquitetura	Generative Adversarial Network + Convolutional		
Discriminadora		**Geradora**	
Camadas Ocultas	**5**	**Camadas Ocultas**	**5**
Camada	**Função de Ativação**	**Camada**	**Função de Ativação**
1	LeakyReLU	1	ReLU
2	LeakyReLU	2	ReLU
3	LeakyReLU	3	ReLU
4	LeakyReLU	4	ReLU
5	Linear	5	ReLU
Saída	Sigmoid	**Saída**	Tahn

Tabela 6. *Detalhes da arquitetura da rede DCGAN.*

MNIST			
Épocas (para estabilizar)	**Duração - Treino (Minutos)**	**Memória Utilizada (MB)**	**Duração - Geração (Segundos)**
25	808m	600-700MB	1,4s

Tabela 7. *Exemplo de resultados obtidos com a arquitetura DCGAN.*

Segue, abaixo, as amostras geradas por esta arquitetura após as 25 épocas:

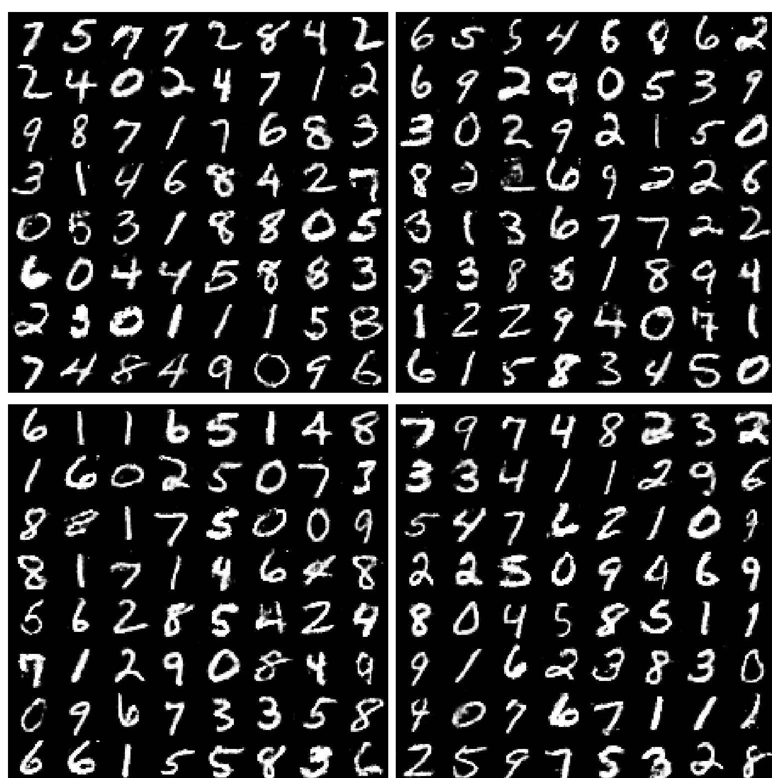


Figura 25. *Exemplos de imagens geradas pela arquitetura DCGAN ao final de 25 épocas.*

Já que todas as imagens geradas nesse exemplo são da mesma sessão de treinamento, a escolha de qual amostragem avaliar foi arbitrária - a última dessa lista:

Classe	Observações	Instâncias	Nota	V. Pond
0	- Traço contínuo x 5 - Traço forte x 5 - Forma correta x 4 - Facilmente identificado x 5 - Variedade - Forma incorreta ou imprecisa x 1 - Presença de anomalia x 1	5	**7,2**	**0,5625**
1	- Traço contínuo x 9 - Traço forte x 9 - Forma correta x 8 - Facilmente identificado x 9 - Variedade - Forma incorreta ou imprecisa x 1	9	**7,77**	**1,09265**
2	- Traço contínuo x 9 - Traço forte x 7 - Forma correta x 9 - Facilmente identificado x 9 - Variedade	9	**7,55**	**1,06171**

	- Presença de anomalia x 1			
3	- Traço contínuo x 6 - Traço forte x 4 - Forma correta x 6 - Facilmente identificado x 6 - Variedade - Presença de anomalia x 1	6	**7,33**	**0,68718**
4	- Traço contínuo x 5 - Traço forte x 3 - Forma correta x 6 - Facilmente identificado x 6 - Variedade - Traço inacabado x 1	6	**6,66**	**0,62437**
5	- Traço contínuo x 6 - Traço forte x 4 - Forma correta x 6 - Facilmente identificado x 6 - Variedade - Presença de anomalia x 1	6	**7,33**	**0,68718**
6	- Traço contínuo x 5 - Traço forte x 4 - Forma correta x 4 - Facilmente identificado x 4 - Variedade - Forma incorreta ou imprecisa x 1	5	**6,8**	**0,53125**
7	- Traço contínuo x 7 - Traço forte x 6 - Forma correta x 6 - Facilmente identificado x 6 - Variedade - Forma incorreta ou imprecisa x 1	7	**7,14**	**0,78093**
8	- Traço contínuo x 4 - Traço forte x 4 - Forma correta x 5 - Facilmente identificado x 5 - Variedade - Traço inacabado x 1	5	**7,2**	**0,5625**
9	- Traço contínuo x 4 - Traço forte x 5 - Forma correta x 6 - Facilmente identificado x 6 - Variedade - Traço inacabado x 2	6	**6,66**	**0,62437**

Tabela 8. *Análise dos resultados com a arquitetura DCGAN.*

Média Ponderada (*w_avg*)	**7,21**

Comparando com o resultado dos autores do trabalho original:

Figura 26. *Exemplos de dígitos gerados pela arquitetura DCGAN, obtidos pelos autores (RADFORD, A. & CHINTALA, S., 2016).*

4.5.3. PixelCNN

Detalhes da Arquitetura

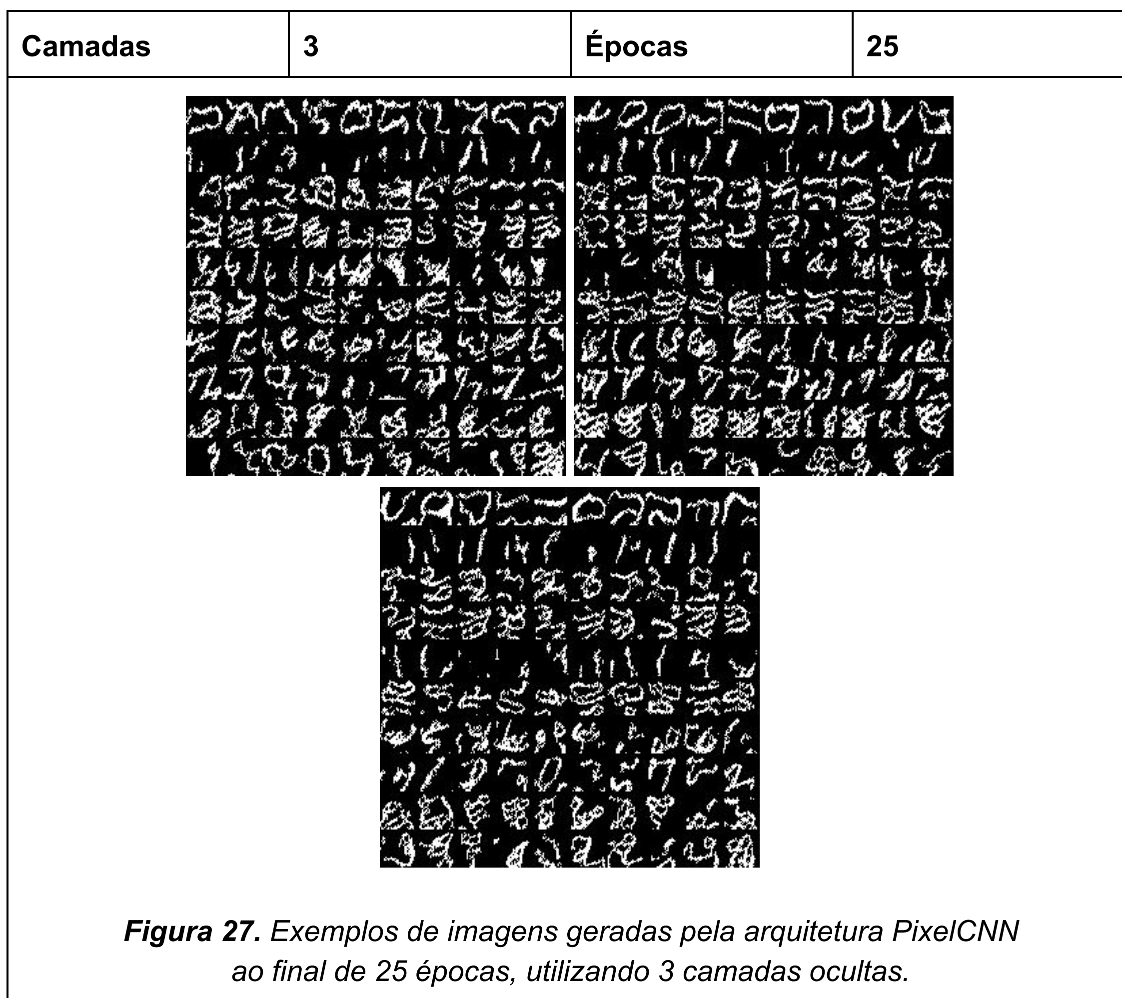
Arquitetura	Gated Convolutional Neural Network		
Função de Ativação	ReLU	**Camadas de Convolução**	7
Feature Maps p/ Cada Camada Conv.	16	**Feature Maps p/ Cada Camada Conv. de Saída**	32

Tabela 9. *Detalhes da arquitetura da rede PixelCNN.*

MNIST			
Épocas (para estabilizar)	**Duração - Treino (Minutos)**	**Memória Utilizada (MB)**	**Duração - Geração (Segundos)**
25	~410m	~1100MB	8s
35	~580m	~1100MB	8s
50	~820m	~1100MB	8s
50	~2150m	~2000MB	10s

Tabela 10. *Exemplo de resultados obtidos com a arquitetura PixelCNN, em cima do conjunto MNIST.*

Aqui, a rede gera uma amostragem ao final da primeira época e, depois, a cada 10 épocas:

<table>
<tr><td>Camadas</td><td>3</td><td>Épocas</td><td>25</td></tr>
<tr><td colspan="4">
Figura 27. Exemplos de imagens geradas pela arquitetura PixelCNN ao final de 25 épocas, utilizando 3 camadas ocultas.</td></tr>
</table>

<table>
<tr><td>Camadas</td><td>3</td><td>Épocas</td><td>35</td></tr>
<tr><td colspan="4"></td></tr>
</table>

Figura 28. *Exemplos de imagens geradas pela arquitetura PixelCNN ao final de 35 épocas, utilizando 3 camadas ocultas.*

Camadas	**3**	**Épocas**	**50**

Figura 29. *Exemplos de imagens geradas pela arquitetura PixelCNN ao final de 50 épocas, utilizando 3 camadas ocultas.*

Camadas	**6**	**Épocas**	**50**

Figura 30. *Exemplos de imagens geradas pela arquitetura PixelCNN ao final de 50 épocas, utilizando 6 camadas ocultas.*

Com as configurações de apenas 3 camadas (ocultas), percebe-se que um aumento no número de épocas de treinamento não se traduz em um aumento de qualidade nas imagens geradas. Isso se dá, muito provavelmente, pelo fato de que o modelo já atingiu o limite da dimensionalidade da representação do objeto da imagem. Por outro lado, pode-se observar como a adição de apenas 3 camadas a mais melhorou significativamente a qualidade das imagens geradas, porém, a um custo um pouco mais de 2,5 vezes o tempo da versão com apenas 3 camadas, treinada pelo mesmo número (50) de épocas. Dada a clara diferença entre a qualidade das imagens geradas com a rede de 6 camadas e as de 3 camadas, será avaliada apenas os resultados da rede de 6 camadas, observando a imagem da última época (sexta imagem):

Classe	Observações	Instâncias	Nota	V. Pond
0	- Traço contínuo x 8 - Traço forte x 0 - Forma correta x 7 - Facilmente identificado x 8 - Variedade - Forma incorreta ou imprecisa x 2 - Presença de anomalia x 2	10	**2,0**	**0,2**
1	- Traço contínuo x 1 - Forma correta x 1 - Facilmente identificado x 1 - Irreconhecível x 9	10	**0,6**	**1,06**
2	- Traço contínuo x 2 - Traço forte x 2 - Forma correta x 2 - Facilmente identificado x 2 - Irreconhecível x 6	10	**1,6**	**0,16**
3	- Traço contínuo x 3 - Traço forte x 2 - Forma correta x 3 - Facilmente identificado x 6 - Variedade - Traço inacabado x 1 - Presença de anomalia x 4 - Irreconhecível x 4	10	**2,0**	**0,2**
4	- Traço contínuo x 3 - Traço forte x 3 - Forma correta x 3 - Facilmente identificado x 3 - Variedade - Presença de anomalia x 1 - Irreconhecível x 7	10	**1,0**	**0,1**
5	- Traço contínuo x 2 - Traço forte x 2 - Forma correta x 2 - Facilmente identificado x 2 - Variedade - Irreconhecível x 8	10	**2,6**	**0,26**
6	- Irreconhecível x 10	10	**0**	**0,0**
7	- Traço contínuo x 3 - Traço forte x 2 - Forma correta x 6	10	**2,6**	**0,26**

	- Facilmente identificado x 4 - Variedade - Traço inacabado x 3 - Irreconhecível x 4			
8	- Traço contínuo x 1 - Traço forte x 0 - Forma correta x 1 - Facilmente identificado x 1 - Presença de anomalia x 1 - Irreconhecível x 9	10	**0,4**	**0,04**
9	- Traço contínuo x 3 - Traço forte x 3 - Forma correta x 3 - Facilmente identificado x 3 - Irreconhecível x 7	10	**2,4**	**0,24**

Tabela 11. *Análise dos resultados com a arquitetura PixelCNN, em cima do conjunto MNIST.*

Média Ponderada (*w_avg*)	**1,52**

* **Os autores do trabalho original (OORD et al., 2016b) não realizaram testes em cima do MNIST**

Cifar10			
Épocas (para estabilizar)	**Duração - Treino (Minutos)**	**Memória Utilizada (MB)**	**Duração - Geração (Segundos)**
50	2650	~2000MB	8s

Tabela 12. *Exemplo de resultados obtidos com a arquitetura PixelCNN, em cima do conjunto Cifar10.*

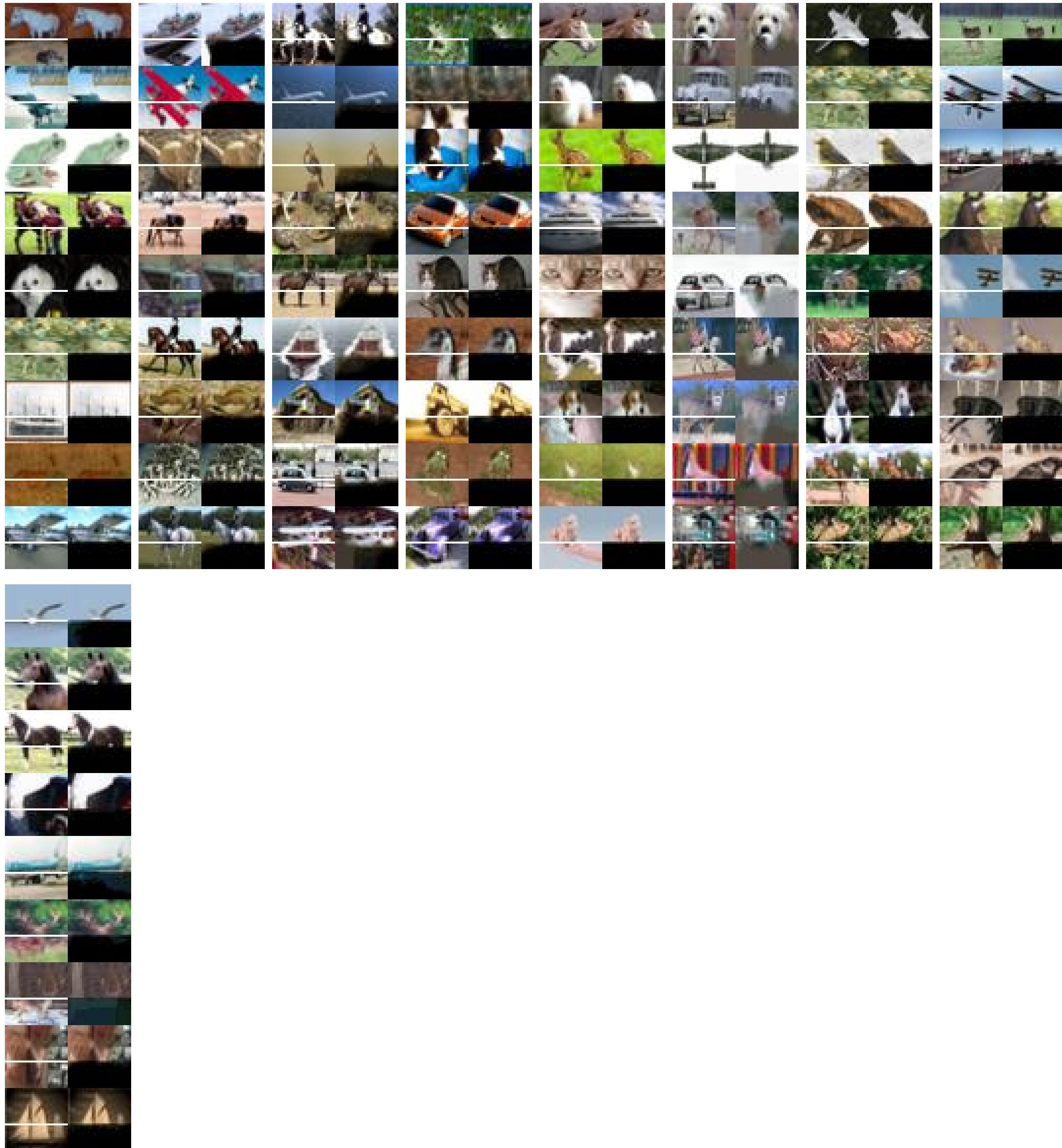


Figura 31. *Exemplos de imagens Cifar10 geradas pela arquitetura PixelCNN ao final de 50 épocas, utilizando 6 camadas ocultas.*

O teste que foi realizado acima não foi de geração completa de imagem. Foi, na verdade, em cima da tarefa de, em função de uma imagem incompleta, tentar preencher o resto do espaço com detalhes coerentes à estrutura já presente. Por não ter resultados de outras estruturas para comparar tempo, consumo de memória e tempo de geração, a nota será calculada como a média da porcentagem de objetos reconhecíveis e irreconhecíveis (o que pode ser propenso a acontecer em função do tamanho das imagens geradas) nas amostras geradas (conforme mostradas acima) em cada coluna.

Coluna	Reconhecíveis	Irreconhecíveis
1	7	2

2	9	0
3	9	0
4	8	1
5	7	2
6	7	2
7	6	3
8	6	3
9	6	3

Tabela 13. *Análise dos resultados obtidos com a arquitetura PixelCNN, em cima do conjunto Cifar10.*

Nota Final	**80,24**

Para o conjunto Cifar10 podemos ver que, com uma rede razoavelmente pequena, apesar do tempo extenso de treinamento, a rede foi capaz de completar as imagens fornecidas com estruturas coerentes com o que já estava presente na imagem, sendo possível gerar imagens facilmente reconhecíveis, com poucas exceções, e com alta acurácia das estruturas esperadas de cada classe.

Comparando com o resultado do trabalho original (OORD et al., 2016b):

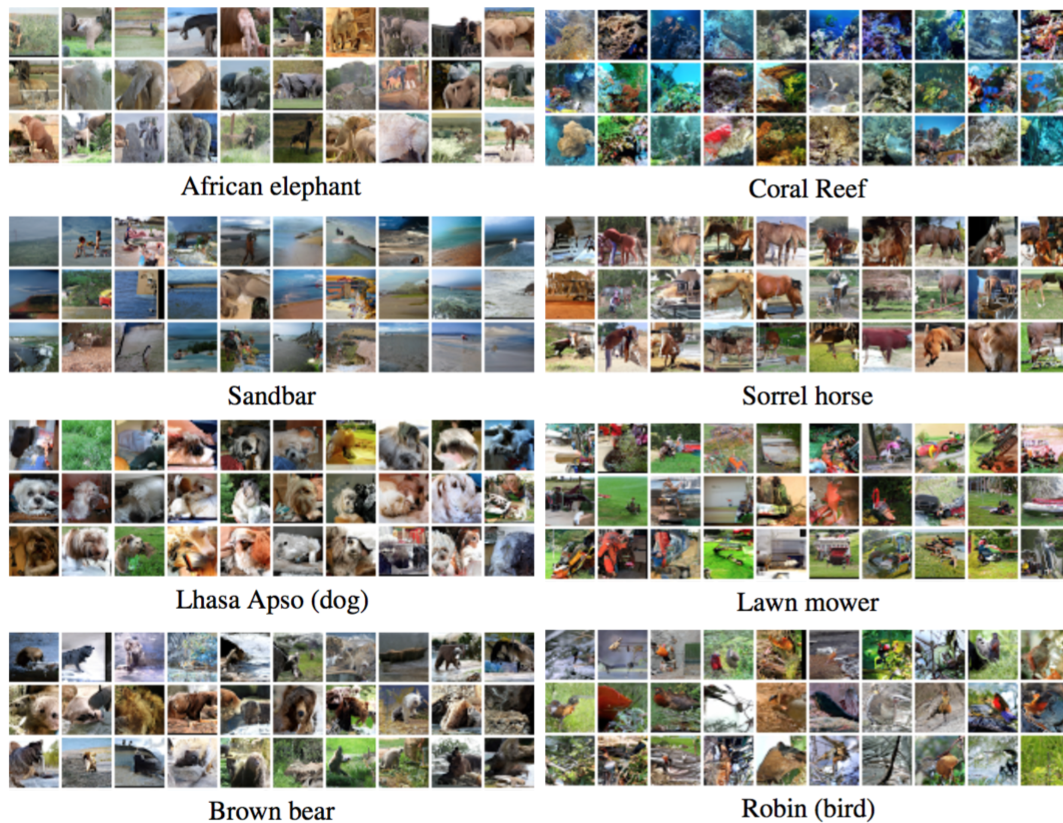


Figura 32. *Exemplos de imagens Cifar10 geradas pela arquitetura PixelCNN, obtidas pelos autores (OORD et al., 2016b).*

4.6. CONCLUSÃO DOS RESULTADOS

Notas MNIST

Normalizado	DRAW	DCGAN	PixelCNN
Tempo de Treino	**0,00054**	**0,27010**	**0,99945**
Memória	**0,00066**	**0,13382**	**0,99933**
Tempo de Geração	**0,15094**	**0,09433**	**0,90566**

Tabela 14. *Notas normalizadas de cada arquitetura para cada critério em cima do conjunto MNIST.*

Arquitetura	Equação da Nota	NF
DRAW	$(1 - 0,00054) \cdot 4 \cdot (1 - 0,00066) \cdot 1,5 \cdot (1 - 0,15094) \cdot 1,1 \cdot 7,06$	**39.51**
DCGAN	$(1 - 0,27010) \cdot 4 \cdot (1 - 0,13382) \cdot 1,5 \cdot (1 - 0,09433) \cdot 1,1 \cdot 7,21$	**27.25**

PixelCNN	$(1 - 0,99945) \cdot 4 \cdot (1 - 0,99933) \cdot 1,5 \cdot (1 - 0,90566) \cdot 1,1 \cdot 1,52$	**3.48e^{-7}**

Tabela 15. *Notas finais de cada arquitetura em cima do conjunto MNIST.*

Em uma análise inicial já se percebe claramente a disparidade entre a PixelCNN e as outras arquiteturas, para o MNIST. A PixelCNN precisaria de mais épocas de treinamento (mais tempo) ou mais camadas ocultas (mais tempo e mais memória) para obter resultados melhores (ou até mais épocas *e* mais camadas ocultas). Por outro lado, teve resultados satisfatórios com o Cifar10, apesar de ainda manter os tempos altos de treinamento.

Considerando os resultados que os autores que propuseram esta arquitetura obtiveram, sabe-se que é possível gerar imagens bem variadas e bem realistas, mas talvez fique inviável para o usuário pelo fator tempo se não for utilizado um ambiente com mais recursos, como placas gráficas (e, aqui, o limitador financeiro) e clusters para realizar os treinos.

Já para o DCGAN e o DRAW é um resultado interessante, pois foram obtidos resultados que mostram o DCGAN gerando mais resultados melhores, dando uma nota intermediária (a média ponderada das classes), ao custo um pouco mais de tempo e de memória, resultado em uma nota final um pouco mais baixa.

Considerando o MNIST, o DRAW parece ser o mais apropriado, visto que conseguiu gerar imagens satisfatórias em menos tempo e consumindo menos memória. Seria interessante realizar testes entre as duas arquiteturas em cima dos outros conjuntos para ver como que cada uma se comporta. No trabalho original do DRAW, os autores conseguiram gerar imagens realistas com uma boa fidelidade às estruturas, formas e cores das fotos originais.

Pelo trabalho original da DCGAN, os autores mostram como esta arquitetura consegue resultados muito bons nos conjuntos de dados *Imagenet-1k*, *Large-scale Scene Understanding* e o *Faces Dataset*, o que significa um altíssimo potencial de variedade de imagens geradas de diferentes classes.

5. CONCLUSÃO E TRABALHOS FUTUROS

Resolver o problema de síntese de imagens, seja com redes neurais ou outra abordagem, não é um problema fácil, e pode ser abordado de diversas maneiras. Este trabalho estudou três arquiteturas diferentes de redes neurais para avaliar e demonstrar a capacidade destas de gerar imagens realistas e diversificadas - uma habilidade extremamente útil para a geração de componentes gráficos para jogos. É fato que é recomendado (às vezes imprescindível) ter bastante recursos computacionais ou então ficar sujeito a tempos extensos de treinamento. Ainda mais que, ao lidar com uma abordagem como redes neurais, o espaço de busca é muito massivo, dada a quantidade de parâmetros envolvidos. Independentemente disso, os resultados mostram-se promissores, com imagens sendo geradas com estruturas reconhecíveis e em tempo hábil (na maioria dos casos), especialmente ao considerar os resultados obtidos pelos respectivos trabalhos originais.

Algumas arquiteturas, como as que envolvem redes generativas adversariais, estão sendo exploradas para das mais diversas maneiras no contexto de geração de imagens. Já foram demonstradas com sucesso em serem capazes de façanhas tais como realizar transferência de estilo de uma imagem para outra (i.e., o estilo de um pintor para uma foto moderna ou aplicar efeito de noite a uma imagem de dia), gerar imagens de uma pessoa em várias poses (tendo apenas uma foto da pessoa de referência e aprender poses de fotos de outras pessoas), geração (de imagem) de peças de roupas com o estilo de uma foto de referência, gerar fotos de pessoas "inventadas" baseadas em rostos existentes, entre muitas outras.

Outro trabalho futuro que poderia ser realizado é de adaptar as arquiteturas escolhidas neste trabalho com as estratégias detalhadas por (TIELEMAN, 2014). Adicionalmente, entre as outras arquiteturas a serem exploradas, poderiam ser realizados experimentos RBMs, visto que isso não foi possível dentro do tempo para este trabalho, mas que, de acordo com (TIELEMAN, 2014), RBMs tem grande potencial para serem utilizadas para aprendizagem de características.

A cada ano que passa, o mercado vê a entrada de novas placas gráficas, com arquiteturas cada vez mais especializadas para o uso de redes neurais, e as existentes vão ficando mais baratas, permitindo que cada vez mais pessoas possam ter acesso a equipamento apropriado para realizar experimentos como este. Com isso, aumenta-se o potencial espaço de busca sendo explorado quanto às tarefas que redes neurais conseguem resolver.

Conforme discussão no capítulo 4, nem todos os experimentos planejados, por restrição de tempo, puderam ser executados, assim como a funcionalidade de utilizar fontes de dados dinâmicas e a funcionalidade de retro-alimentar a rede geradora com o próprio resultado para treinar a mesma ainda mais. Com isso, uma das primeiras sugestões de trabalho futuro fica como realizar os devidos experimentos com as arquiteturas utilizadas neste trabalho nos conjuntos Cifar10 e ImageNet. E, nos últimos anos, surgiram vários conjuntos de dados que poderiam ser interessantes de se testar, tais como o Fashion-MNIST e o LSUN, pois lidam com objetos que se vê com frequência em jogos e que seria vantajoso ter a capacidade de gerar inúmeros objetos novos das classes presentes nesses conjuntos.

Uma limitação que é observado ao utilizar conjuntos como o MNIST e o Cifar10, é o tamanho das imagens originais. Por se tratar de imagens pequenas (28x28 e 32x32 pixels para o MNIST e Cifar10, respectivamente), as arquiteturas são feitas para gerar imagens do mesmo tamanho. Em muitos casos, é desejável poder ter imagens maiores, pois, com apenas 28x28 pixels, aumentar o tamanho da imagem (sem melhorar a resolução) rapidamente degrada a qualidade da mesma. Então, seria interessante ver em um trabalho futuro serem exploradas abordagens para conseguir gerar imagens grandes a partir de representações treinadas de imagens pequenas.

Outra sugestão de trabalho futuro seria a implementação, de fato, da ferramenta completamente integrada, para ser usada de forma mais fluida - possivelmente com interface gráfica, e a possibilidade de escolher entre várias opções de arquiteturas para ser utilizada. Adicionalmente poderia ter um repositório de modelos já treinados que qualquer usuário poderia baixar para não ter que treinar um modelo do zero (porém não estaria impedido de o fazer) ou até mesmo treinar um modelo "pronto" mais ainda. Dessa forma, com um ecossistema de usuários

contribuindo com modelos capazes de gerar imagens das mais diversas coisas, o esforço individual para colocar as mãos em um rico conjunto de componentes gráficos fica substancialmente reduzido, que, no final das contas, é o problema principal que este trabalho tenta atacar para resolver.

Quando se trata do que experimentar fazer com redes neurais, possibilidades é o que não falta. Como última idéia de trabalho futuro, um estudo sobre o uso de redes neurais gerarem modelos 3D, como também já foi feito com GANs, sejam elas treinadas em cima de outros modelos 3D ou até mesmo gerar o modelo 3D a partir de uma imagem (original ou gerada por trabalhos como este).

Não é só para a síntese de imagens que servem redes neurais. Estas já estão sendo utilizadas com sucesso em síntese de vídeos e de sons (incluindo músicas e vozes). Quem sabe, em um futuro bem próximo, com uma coleção bem selecionada de redes neurais (tudo integrado a uma ferramenta única) e, claro, bons conjuntos de dados para treinamento, o cidadão comum poderá ser, com baixo custo, tanto o programador quanto o roteirista, o artista e o compositor dos componentes de seus próprios jogos, trazendo para a indústria um verdadeiro “boom” criativo.

6. REFERÊNCIAS

AGHBAM, H.H., HERAVI, E.J. Guide to Convolutional Neural Networks: A Practical Application to Traffic-sign detection and classification, 2017.

ALLENDER, E. Circuit complexity before the dawn of the new millennium. Foundations of Software Technology and Theoretical Computer Science, pp. 1–18, 1996.

BENGIO, Y. Learning deep architectures for AI. Now Publishers Inc, 2009.

BENGIO, Y., & LECUN, Y. Scaling learning algorithms towards AI. Large-Scale Kernel Machines, 2007.

BENGIO, Y., LAMBLIN, P., POPOVICI, D., & LAROCHELLE, H. Greedy layer-wise training of deep networks. Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference (p. 153), 2007.

BINK, M.L. & MARSH, R.L. Cognitive regularities in creative activity. Review of General Psychology, vol. 4, pp. 59-78, 2001.

BISHOP, C.M. Neural Networks for Pattern Recognition(livro), 1995.

BODEN, M.A. The Creative Mind: Myths and Mechanisms, 1991.

BOESEN, A., LARSEN, L., SONDERBY, S.K. Generation Faces with Torch. Disponível em http://torch.ch/blog/2015/11/13/gan.html, 2015.

CHEN, Ken & ROBERTS, Jonathan. Learning-Based Procedural Content Generation. Disponível em https://arxiv.org/abs/1308.6415v2.pdf, 2013.

COLTON, Simon & WIGGINS, G.A. Computational Creativity: The Final Frontier? Proceedings of the 20th European Conference on Artificial Intelligence, pp. 21-26, 2012.

COMPTON, K., OSBORN, J.C., MATEAS, M. Generative methods. In: Proceedings of the 4th Work- shop on Procedural Content Generation in Games, 2013.

COPELAND, Jack. A Brief History of Computing at AlanTuring.net, 2000

CORTES, C., & VAPNIK, V. Support-vector networks. Machine learning, 20, 273–297, 1995.

DAYAN, P., HINTON, G.E., Neal, R. M., & ZEMEL, R. S. The helmholtz machine. Neural computation, 7(5):889–904, 1995.

DE CARLI, D.M., D ORNELLAS, M.C., POZZER, C.T. A Survey of Procedural Content Generation Techniques Suitable to Game Development. Proceedings of SBGames, 2011.

DUCH, Włodzisław. Computational Creativity. 2006 International Joint Conference on Neural Networks, 2006.

ERHAN, D., MANZAGOL, P.A., BENGIO, Y., BENGIO, S., & VINCENT, P. The difficulty of training deep architectures and the effect of unsupervised pre-training. Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS 09), pp. 153–160, 2009.

FEI-FEI, L., FERGUS, R., & PERONA, P. Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. IEEE. CVPR 2004. Workshop on Generative-Model Based Vision, 2004.

FUKUSHIMA, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, Vol. 36, No. 4, pp. 193–202, 1980.

GOODFELLOW, I., et al. Generative adversarial nets. NIPS, 2014.

GOODFELLOW, I. et al. Deep Learning(livro). MIT Press, 2016.

GRANT, Eugene F.; Lardner. The Talk of the Town – It, The New Yorker, Rex (2 de Agosto, 1952).

GRAVES, A. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850, 2013.

GRAVES, A., WAYNE, G., & DANIHELKA, I.. Neural turing machines. arXiv preprint arXiv:1410.5401, 2014.

GREGOR, K., DANIHELKA, I., GRAVES, A., REZENDE, D.J., and WIERSTRA, D. DRAW: A recurrent neural network for image generation. Proceedings of the 32nd International Conference on Machine Learning, 2015.

HALLE, M., & STEVENS, K.N. Speech recognition: A model and a program for research. Information Theory, IRE Transactions on, 8, 155–159, 1962.

HALLE, M., & STEVENS, K.N. Analysis by synthesis. Proc. Seminar on Speech Compression and Processing, 1959.

HE, K., ZHANG, X., REN, S., SUN, J. Deep Residual Learning for Image Recognition. Disponível em https://arxiv.org/pdf/1512.03385.pdf, 2015.

HE, Kaiming et al. Deep Residual Learning for Image Recognition. Disponível em https://arxiv.org/pdf/1512.03385v1.pdf, 2015.

HENDRIKX, M., MEIJER, S., VAN DER VELDEN, J., IOSUP, A. (2013). Procedural content generation for games: A survey. ACM Trans. Multimedia Comput. Commun. Appl. 9 1:1–1:22, 2013.

HINTON, G.E. Boltzmann machine.
Disponível em http://www.scholarpedia.org/article/Boltzmann_machine, 2007.

HINTON, G.E., KRIZHEVSKY, A., & WANG, S. Transforming auto-encoders. Artificial Neural Networks and Machine Learning–ICANN 2011, 44–51, 2011.

HINTON, G.E., OSINDERO, S., and TEH, Y. W. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527-1554, 2006.

HINTON, G.E., & SALAKHUTDINOV, R. Reducing the Dimensionality of Data with Neural Networks. Science, 313, 504–507, 2006.

HINTON, G.E. & SEJNOWSKI, T.J. Optimal Perceptual Inference. Proceedings of the IEEE conference on Computer Vision and Pattern Recognition, Washington DC, pp. 448-453, 1983.

HINTON, G.E., SRIVASTAVA, N., KRIZHEVSKY, A., SUTSKEVER, I., & SALAKHUTDINOV, R.R. Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580, 2012.

HOFSTADTER, D.R. Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought. NY: Basic Books, 1995.

HUBEL, D.H. & WIESEL, T.N. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. The Journal of Physiology, Vol. 160, No. 1, pp. 106–154.2, 1962.

KINGMA, D.P. & WELLING, M. Auto-encoding Variational Bayes. Disponível em https://arxiv.org/pdf/1312.6114.pdf, 2013.

KINGMA, D.P. & BA, J.L. Adam: A Method for Stochastic Optimization. Disponível em https://arxiv.org/pdf/1412.6980.pdf, 2015.

KOCABAS, Sakir & LANGLEY, Pat. An integrated framework for extended discovery in particle physics. Proc. 4th Int. Conf. on Discovery Science, Washington, D.C. Springer, pp. 182–195, 2001.

KRIESEL, David. A Brief Introduction to Neural Networks(livro), 2005.

KRIZHEVSKY, A., SUTSKEVER, I., & HINTON, G. E. ImageNet classification with deep convolutional neural networks. NIPS (p. 4), 2012.

LANG, K.J., WAIBEL, A.H., & HINTON, G.E. A time-delay neural network architecture for isolated word recognition. Neural networks, 3, 23–43, 1990.

LANGLEY, Pat & JONES, Randolph. A computational model of scientific insight. In R. Sternberg (Ed.), The nature of creativity. Cambridge Uni. Press, 1988.

LANGLEY, Pat, SIMON, H.A., BRADSHAW, G.A., and ZYTKOW, J.M. Scientific Discovery: Computational Exploration of the Creative Process. Cambridge, MA: MIT Press, 1987.

LAROCHELLE, H. & MURRAY, I. The neural autoregressive distribution estimator. Journal of Machine Learning Research, 15:29–37, 2011.

LECUN, Y., BOTTOU L., BENGIO, Y., HAFFNER, P. Gradient-based learning applied to document recognition. Proceedings of the IEEE, Vol. 86, No. 11, pp. 2278–2324, 1998.

LECUN, Y., HUANG, F.J., & BOTTOU, L. Learning methods for generic object recognition with invariance to pose and lighting. Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 97–104, 2004.

LEE, H., GROSSE, R., RANGANATH, R., & NG, A.Y. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. Proceedings of the 26th Annual International Conference on Machine Learning, pp. 609–616, 2009.

LIOU, C., HUANG, J., YANG, W. Modeling word perception using the Elman network. Neurocomputing, vol. 71, pp. 3150-3157, 2008.

LIOU, C., CHENG, W., LIOU, J., LIOU, D. Autoencoder for words. Neurocomputing, vol. 139, pp. 85-96, 2014.

MARTENS, J. Deep learning via Hessian-free optimization. Proceedings of the 27th international conference on Machine learning, 2010.

MCCARTHY, John et al. A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, 1955.

MCCARTHY, John. What Is Artificial Intelligence at http://www-formal.stanford.edu/jmc/whatisai/whatisai.html, 2007.

MINSKY, M.L., & PAPERT, S. Perceptrons: An introduction to computational geometry. MIT Press, 1969.

MITCHELL, Melanie. Analogy-Making as Perception: A Computer Model. Cambridge, MA: MIT Press, 1993.

MNIH, V. CUDAMat: A CUDA-based matrix class for Python (Technical Report UTML TR 2009-004). University of Toronto, Department of Computer Science, 2009.

MNIH, Volodymyr et al. Recurrent models of visual attention. In Advances in Neural Information Processing Systems, pp. 2204–2212, 2014.

MOHAMED, A., DAHL, G., & HINTON, G. Deep Belief Networks for phone recognition, 2009.

MOHAMED, A., DAHL, G.E., & HINTON, G.E. Acoustic modeling using deep belief networks. Audio, Speech, and Language Processing, IEEE Transactions on, 20, pp. 14 –22, 2012.

NAIR, V., & HINTON, G. 3-d object recognition with deep belief nets. Advances in Neural Information Processing Systems, 2010.

NITSCHE, Michael et al. Designing Procedural Game Spaces: A Case Study. Proceedings of FuturePlay 2006, 2006.

OORD, A., KALCHBRENNER, N., KAVUKCUOGLU, K. Pixel Recurrent Neural Networks. Disponível em https://arxiv.org/pdf/1601.06759v3.pdf, 2016a.

OORD, A., KALCHBRENNER, N., KAVUKCUOGLU, K., VINYALS, O., ESPEHOLT, L., GRAVES, A. Conditional Image Generation With PixelCNN Decoders. Disponível em https://arxiv.org/pdf/1606.05328v2.pdf, 2016b.

PINTO, N., DOUKHAN, D., DICARLO, J.J., & COX, D.D. A high-throughput screening approach to discovering good forms of biologically inspired visual representation, 2009.

POWLEY, Edward J. et al. Automated Tweaking of Levels for Casual Creation of Mobile Games. Proceedings of the 2nd Computational Creativity and Games Workshop, 2016.

RADFORD, A. & CHINTALA, S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. Disponível em https://arxiv.org/pdf/1511.06434.pdf, 2016.

RAINA, R., MADHAVAN, A., & Ng, A.Y. Large-scale deep unsupervised learning using graphics processors. Proceedings of the 26th Annual International Conference on Machine Learning, pp. 873–880, 2009.

RANZATO, M.A., POULTNEY, C., CHOPRA, S., & LECUN, Y. Efficient learning of sparse representations with an energy-based model. Advances in neural information processing systems, 19, 2006.

REHLING, J.A. Letter Spirit (Part Two): Modeling Creativity in a Visual Domain. PhD Thesis, Indiana University, 2001.

RUMELHART, David, MCCLELLAND, James. Parallel Distributed Processing: Explorations in the Microstructure of Cognition. MIT Press, 1986.

SALAKHUTDINOV, R., & HINTON, G. Semantic hashing. International Journal of Approximate Reasoning, 50, 969–978, 2009.

SERRE, T., WOLF, L., BILESCHI, S., RIESENHUBER, M., POGGIO, T. Robust Object Recognition with Cortex-Like Mechanisms. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 29, No. 3, pp. 411–426, 2007.

SHAKER, M., SHAKER, N., TOGELIUS, J., ABOU-ZLEIKHA, M. A Progressive Approach to Content Generation. European Conference on the Applications of Evolutionary Computation, pp. 381-393, 2015.

SNODGRASS, Sam. General Statistical Approaches to Procedural Map Generation. Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, July, 2016.

STERNBERG, R.J. Handbook of Human Creativity. Cambridge Uni. Press, 1998.

SUTSKEVER, I., & HINTON, G.E. Deep, narrow sigmoid belief networks are universal approximators. Neural computation, 20, 2629–2636, 2008.

SUTSKEVER, I., & HINTON, G.E. Learning multilevel distributed representations for high-dimensional sequences. Proceeding of the Eleventh International Conference on Artificial Intelligence and Statistics, pp. 544–551, 2007.

SUTSKEVER, I., MARTENS, J., DAHL, G., & HINTON, G. On the importance of initialization and momentum in deep learning. Proceedings of the 30th International Conference on Machine Learning (ICML-13), pp. 1139–1147, 2013.

SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., RABINOVICH, A.. Going deeper with convolutions. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1–9, 2015.

THEIS, Lucas & BETHGE, Matthias. Generative Image Modeling Using Spatial LSTMs. Disponível em https://arxiv.org/pdf/1506.03478v2.pdf, 2015

TIELEMAN, T. Gnumpy: an easy way to use GPU boards in Python (Technical Report UTML TR 2010-002). University of Toronto, Department of Computer Science, 2010.

TIELEMAN, T. Optimizing Neural Networks that Generate Images. Graduate Department of Computer Science, University of Toronto, 2014.

TOGELIUS, J., KASTBJERG, E., SCHEDL, D., YANNAKAKIS, G.N. What is procedural content generation?: Mario on the borderline. Proceedings of the 2nd International Workshop on Procedural Content Generation in Games, 2011a.

TOGELIUS, J., SHAKER, N., NELSON, M.J. Procedural Content Generation in Games: A Textbook and an Overview of Current Research, pp. 1-15, 2016.

TOGELIUS, J., YANNAKAKIS, G.N., STANLEY, K.O., BROWNE, Cameron. Search-Based Procedural Content Generation: A Taxonomy and Survey. IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, 2011b.

TOGELIUS, Jain et al. Autoencoders for Level Generation, Repair, and Recognition. ICCC Workshop on Computational Creativity and Games, 2016.

TOGELIUS, Jain et al. Search-based Procedural Content Generation: A Taxonomy and Survey. Computational Intelligence and AI in Games, IEEE Transaction(pág 172-186), 2011.

TOGELIUS, Jain et al. The 2010 Mario AI Championship: Level Generation Track. Computational Intelligence and AI in Games, IEEE Transaction(pág 332-347), 2011.

TORRALBA, A., FERGUS, R., & FREEMAN, W.T. Object and scene recognition in tiny images. J. Vis., 7, 193–193, 2007.

UZDIAL, Matthew et al. Toward Game Level Generation from Gameplay Videos. Proceedings of the 2015 Workshop on Procedural Content Generation, Asilomar, California, 2015.

YANNAKAKIS, G.N. & TOGELIUS J. Experience-driven Procedural Content Generation. IEEE Transactions on Affective Computing, Volume: 2, Issue: 3, July-Sept., 2011.

APÊNDICE A - ARTIGO

AVALIAÇÃO DE REDES NEURAIS PARA A GERAÇÃO DE IMAGENS

Rafael Rabello Moser

Departamento de Informática e Estatística - Universidade Federal de Santa Catarina

Florianópolis, SC - Brasil

rrmoser@gmail.com

Abstract. *The goal of this work was to perform a comparative analysis of some approaches with neural networks for the generation of images so that, in the future, a tool based on the most appropriate approach found can be created that solves, at least in part, the problem of reducing the cost to create some of the game art components. For this purpose, a comparison was made between recurrent, convolutional and generative adversarial networks, in the form of autocoder, on top of the MNIST and, in the convolutional case, on the Cifar10 datasets, to test the feasibility of using these networks to solve this problem. Through the application of machine learning techniques and neural networks, at the end of the experiments performed with the different chosen neural network architectures, it has been demonstrated that these can be powerful tools in the task of automated image generation.*

Resumo. *O objetivo deste trabalho foi realizar uma análise comparativa de algumas abordagens com redes neurais para a geração de imagens para que, futuramente, possa ser criada uma ferramenta baseada na abordagem mais apropriada encontrada que resolva, pelo menos em parte, o problema de reduzir o custo de criação de alguns dos componentes de arte de jogos. Para tal, foi realizada uma comparação entre redes recorrentes, convolucionais e redes generativas adversariais, na forma de auto-codificadores, em cima do conjunto MNIST e, no caso da convolucional, em cima do conjunto Cifar10, para testar a viabilidade de se utilizar estas redes para resolver este problema. Através da aplicação de técnicas de aprendizagem de máquina e redes neurais, ao final dos experimentos realizados com algumas arquiteturas diferentes de redes neurais, foi demonstrado que estas conseguem ser ferramentas poderosas na tarefa de geração automatizada de imagens.*

1. Introdução

O principal objetivo das redes neurais é de resolver problemas de uma forma semelhante à do cérebro humano, apesar de, mesmo as redes neurais mais modernas, trabalharem com quantidades de unidades ordens de magnitude abaixo daquelas de um cérebro. Ainda assim, já mostra-se extremamente eficiente e eficaz na área de aprendizagem de máquina - sistemas capazes de aprender com dados - sendo aplicadas na resolução de múltiplos problemas, tais como visão computacional e reconhecimento de padrões, coisas tipicamente difíceis de serem resolvidas com abordagens mais tradicionais de programação (BISHOP, 1995), baseadas em regras.

A inteligência artificial tem sido aplicada em jogos desde o seu nascimento, como uma área de pesquisa. Aparecia bastante, por exemplo, na forma de adversários, como no jogo de Nim, em 1951 (GRANT, 1952), e xadrez, também em 1951 (COPELAND, 2000). Na década de 80, começou a ser aplicada à geração procedural (ou automatizado) de conteúdo (GPC) e, recentemente, ganhou espaço na área acadêmica (TOGELIUS, *et al.*, 2011). GPC consiste em um espaço de soluções de design ser explorado para elaborar novas variações. A prática mais comum é de um usuário fornecer conteúdo pronto, o qual é analisado pelo gerador, e este cria conteúdo novo em um estilo semelhante (TOGELIUS, *et al.*, 2016).

Já existe um rico ecossistema de ferramentas que permitem a criação de muitos dos componentes de um jogo - tais como arte visual, música, entre outros - proceduralmente. Exemplos destas incluem ferramentas como a *Unity* (disponível em https://unity3d.com), *Unreal Engine* (disponível em https://www.unrealengine.com) e *Amazon Lumberyard* (disponível em https://aws.amazon.com/lumberyard). Tais ferramentas já reduzem consideravelmente os custos de se criar um jogo, mas, geralmente, exigem que o usuário possua considerável conhecimento dos aspectos técnicos das mesmas (POWLEY, *et al.*, 2016). Com o intuito de reduzir ainda mais os custos (tempo, dinheiro e conhecimento técnico) envolvidos na criação de componentes de jogos, o trabalho visa elaborar uma solução que facilite a criação destes componentes - mais especificamente, os componentes gráficos. RNs, junto com aprendizagem de máquina, já se mostraram muito úteis para o reconhecimento e criação de componentes visuais/gráficos (OORD, *et al.*, 2016a; OORD, *et al.*, 2016b; TIELEMAN, 2014; THEIS & BETHGE, 2015), portanto, foi a abordagem escolhida como base para a resolução do problema proposto.

Na seção 2, a seguir, será dado uma breve explicação dos trabalhos relacionados que dão apoio ao trabalho realizado. Na seção 3 será detalhado o desenvolvimento e a realização dos experimentos e, por fim, na seção 4, será exposto as conclusões e possíveis rumos para trabalhos futuros.

2. Trabalhos Relacionados

2.1. DRAW: Uma Rede Neural Recorrente para a Geração de Imagens

O cerne da arquitetura DRAW é composto por um par de redes neurais recorrentes, onde uma atua como codificador, que comprime a imagem durante a fase de treinamento, e a segunda funciona como um decodificador, que tenta reconstruir as imagens em função dos dados fornecidos. Estas redes são então treinadas de ponta-a-ponta com o SGD, onde a função de perda é um limite superior variacional na *função de probabilidade logaritmica* dos dados. Em função disso, esta arquitetura pertence à classe de auto-codificadores variacionais (*VAE*, do inglês *variational auto-encoders*). A diferença principal entre a DRAW e outros VAEs é a característica supracitada de, ao invés de tentar gerar a imagem de uma só vez, construir a cena iterativamente, através de mudanças aplicadas pelo decodificador.

A estrutura básica de uma rede DRAW é semelhante à de outros VAEs: uma rede codificadora determina a distribuição de códigos latentes que contém informações salientes sobre os dados de entrada; uma rede decodificadora que recebe amostragens destas distribuições e os utiliza para condicionar sua própria distribuição em cima das imagens. Três diferenças importantes diferenciam a DRAW de outros VAEs: a primeira é que nesta arquitetura ambas as redes são recorrentes, portanto, estas trocam entre si *sequências* de amostras de código; adicionalmente, o codificador é ciente das saídas do decodificador. Em segundo lugar, as saídas do decodificador são sucessivamente adicionadas à distribuição que irá, eventualmente, gerar os

dados, ao invés de gerar os dados de uma só vez. E, terceiro, é utilizado um mecanismo dinâmico de atenção que determina tanto a região da entrada que é analisada pelo codificador quanto a região de saída do decodificador. Isso permite que a rede saiba *onde ler*, *onde escrever* e o *quê escrever*.

Na instanciação mais simples de DRAW, toda a imagem de entrada é passada para o codificador a cada passo de tempo, e o decodificador modifica toda a matriz do canvas a cada passo de tempo. No entanto, essa abordagem não permite que o codificador se concentre em apenas parte da entrada ao criar a distribuição latente e nem permite que o decodificador modifique apenas uma parte do vetor do canvas. Em outras palavras, não fornece à rede um mecanismo explícito de atenção seletiva, que acredita-se ser crucial para a geração de imagens em larga escala. Esse modelo básico, sem mecanismo de atenção é chamada pelos autores de “DRAW sem atenção”.

Para incorporar o mecanismo de atenção seletiva à rede sem sacrificar os benefícios do treinamento com gradiente descendente, os autores se inspiraram nos mecanismos diferenciais de atenção usados na síntese de caligrafia (GRAVES, 2013) e Máquinas de Turing Neurais (GRAVES et al., 2014). Ao contrário dos trabalhos supracitados, é utilizada uma abordagem com uma forma de atenção explicitamente bidimensional, onde um vetor de filtros Gaussianos bidimensionais é aplicado na imagem, produzindo um "patch" de imagem de localização e zoom levemente variados. Os autores ressaltam que essa configuração, que se referem simplesmente como “DRAW”, lembra um pouco as transformações usadas nos auto-codificadores baseados em computação gráfica (TIELEMAN, 2014).

2.2. Aprendizagem de Representação Não-Supervisionada com Redes Generativas Adversariais Convolucionais Profundas

Os autores (RADFORD, A. & CHINTALA, S., 2016) propõe que uma maneira de construir boas representações de imagem é treinando GANs (do inglês *Generative Adversarial Networks*) (GOODFELLOW et al., 2014), e depois reutilizando partes das redes geradoras e discriminadoras como extratores de características para tarefas supervisionadas. De acordo com os autores (RADFORD, A. & CHINTALA, S., 2016), sabe-se que as GANs são instáveis no treinamento, geralmente resultando em redes geradoras que produzem saídas sem sentido. Tem havido pesquisas publicadas muito limitadas na tentativa de entender e visualizar o que as GANs aprendem, e as representações intermediárias de GANs de várias camadas.

Neste trabalho, (RADFORD, A. & CHINTALA, S., 2016) fazem as seguintes contribuições:

- Propõem e avaliam um conjunto de restrições na topologia arquitetural das GANs Convolucionais que as tornam estáveis para treinamento na maioria dos ambientes, dando o nome para esta classe de arquiteturas de GANs Convolucionais Profundas (DCGAN)
- Utilizam as redes discriminadoras treinadas para tarefas de classificação de imagens, mostrando desempenho competitivo com outros algoritmos não-supervisionados
- São visualizados os filtros aprendidos pelas GANs e demonstram empiricamente que filtros específicos aprenderam a desenhar objetos específicos
- Mostram que os geradores têm interessantes propriedades aritméticas vetoriais, permitindo fácil manipulação de muitas qualidades semânticas das amostras geradas.

No geral, os autores recomendam a seguinte arquitetura:

- Diretrizes de arquitetura para GANs Convolucionais Profundas estáveis:
- Substitua quaisquer camadas de pooling com convoluções em passos (discriminadora) e convoluções de passos fracionários (geradora).
- Use *batchnorm* tanto na rede geradora quanto na rede discriminadora.
- Remova camadas ocultas completamente conectadas para poder usar arquiteturas mais profundas.
- Use a ativação ReLU na rede geradora para todas as camadas, com exceção da saída, onde deve ser utilizado o Tanh.
- Use a ativação LeakyReLU na rede discriminadora em todas as camadas.

2.3. Geração Condicional de Imagens com Decodificadores PixelCNN

Avanços recentes em modelagem de imagens com redes neurais, como a PixelRNN (do inglês, Pixel Recurrent Neural Network) (OORD et al., 2016a) e LSTMs (do inglês, Long Short-Term Memory) (THEIS & BETHGE, 2015), tornaram viáveis a geração de imagens naturais diversificadas que capturam com sucesso as estruturas implícitas dos dados de treinamento. Estes modelos, contudo, não levam em consideração informações mais abstratas contidas nos dados, porém relevantes, para condicionar as imagens geradas(i.e.: presença de certos objetos na imagem ou alguma restrição de estado).

Além da arquitetura PixelRNN (OORD et al., 2016a), que modela a distribuição dos pixels através de LSTMs bi-dimensionais, a abordagem a seguir apresenta uma versão da arquitetura usando redes neurais convolucionais, a PixelCNN. De acordo com os autores originais (OORD et al., 2016a; OORD et al., 20016b), as PixelRNNs demonstraram, no geral, melhor desempenho. Por outro lado, as PixelCNNs eram significativamente mais rápidas para serem treinadas, dado que convoluções são mais facilmente paralelizáveis. Com a quantidade de pixels em conjuntos de imagens grandes, isso se mostrou ser uma imensa vantagem.

O abordagem apresentada explora o potencial da modelagem condicional de imagens ao construir em cima da versão convolucional da arquitetura utilizada na PixelRNN, resultando em dois modelos novos, a Gated PixelCNN e, ainda, uma variante desta, a Conditional PixelCNN (OORD et al., 2016b). De uma maneira geral, a idéia destas arquiteturas se baseia em utilizar conexões auto-regressivas para modelar as imagens pixel por pixel, decompondo a distribuição conjunta da imagem como um produto de condicionais.

3. Desenvolvimento e Experimentos

A criação de componentes gráficos de jogos é um processo custoso, especialmente no quesito empenho. Exige domínio de ferramentas de modelagem ou habilidade artística (que leva muito tempo para desenvolver se ainda não a tiver) por parte do usuário - no contexto desse problema, o desenvolvedor de jogos independente. Muitas ferramentas modernas já oferecem módulos de geração de componentes, capazes de gerar inúmeras instâncias de um determinado objeto (i.e., uma árvore) em pouquíssimo tempo. No entanto, o desenvolvedor está sujeito às opções que a ferramenta oferece. Ocasionalmente, essas ferramentas permitem que a

própria comunidade desenvolva extensões para propósitos específicos, mas a ferramenta está sujeita a ficar cheia de módulos instalados que somente são utilizados ocasionalmente, deixando a ferramenta muito pesada.

Existem mercados virtuais, também, de componentes gráficos, oferecendo tanto componentes pagos quanto componentes livres de custo, porém, limitados pelo que o criador original fez. Em ambos os casos, o desenvolvedor pode se deparar com uma parede bem rapidamente. Ter uma ferramenta que eliminasse a limitação de ter que usar apenas o que fosse explicitamente criado por outros e permitisse que o próprio desenvolvedor criasse, facilmente e rapidamente, as próprias soluções e pudesse gerar inúmeros componentes gráficos, cada um diferente do anterior, cortaria um tremendo custo do processo artístico do desenvolvedor independente (e de qualquer um que usufruísse da ferramenta).

Considerando os objetivos específicos propostos para este trabalho, podemos começar a elaborar melhor a estrutura e os processos para realizar os experimentos para que possamos, logo em seguida, desenvolver uma ferramenta de geração de imagens. De um modo geral, após os devidos estudos de soluções já existentes, os objetivos se resumem a comparar abordagens com redes neurais de diferentes arquiteturas para o reconhecimento e geração de imagens. Uma vez realizada as avaliações dos resultados, a arquitetura julgada a mais apropriada será escolhida para fazer parte da ferramenta a ser desenvolvida.

Pré-processamento e organização dos dados

Nesta etapa, como o objetivo é utilizar fontes arbitrárias de imagens para o treinamento da rede neural reconhecedora, é de se esperar que exista bastante variabilidade entre as imagens obtidas em quesitos como a resolução da imagem e a posição do objeto de interesse na mesma. Para não aumentar demais a complexidade dos experimentos, assume-se que a estrutura da rede (número de neurônios e conexões) é estática, o que significa que os dados de entrada para a rede devem passar por uma etapa de pré-processamento para regularizá-los, o que consiste em deixar todas as imagens com a mesma resolução e, na medida do possível, deixar o objeto de interesse o mais centralizado possível. Adicionalmente, será necessário, onde aplicável, separar entre imagens coloridas e imagens em preto e branco.

Para poder obter resultados comparáveis entre as abordagens e com o atual estado da arte, foi decidido realizar os testes em cima de conjuntos de dados já bem estabelecidos e de fácil aquisição. Em todos os conjuntos, as imagens já vem normalizadas em relação a tamanho e posição do objeto de interesse, além de metadados indicando qual a classe do objeto presente na imagem. Os conjuntos de dados escolhidos foram o **MNIST**, o **Cifar10**

As Redes

A primeira rede neural na ferramenta será utilizada para reconhecer um objeto específico em uma posição arbitrária dentro de uma imagem qualquer. Pelo fato de os conjuntos de dados utilizados já estarem com as imagens normalizadas e organizadas

por classe do objeto, o papel desta rede se reduz ao de um codificador. O que isso significa é que o objetivo da rede é de aprender representações ou codificações para cada classe de objeto presente nos dados apresentados. Esta é apenas uma parte do conjunto que é formado com a segunda rede.

Enquanto a primeira rede da ferramenta é a codificadora, a segunda ganha o papel de decodificador, ou seja, esta aprende, a partir de uma determinada representação de dados, a gerar uma saída que esteja de acordo com as características dos dados utilizados para treinar a rede codificadora.

Não tem restrições quanto à arquitetura da rede para ser utilizada para o codificador e o decodificador. A arquitetura DRAW, por exemplo, faz uso de RNNs (rede neurais recorrentes), tanto para o codificador quanto para o decodificador. Adicionalmente, é possível combinar arquiteturas diferentes entre o codificador e o decodificador, dando origem aos auto-codificadores variacionais.

Utilizando as duas redes em conjunto, temos o que é chamado de um auto-codificador. Colocando de outra maneira, tendo as transições φ e ψ tal que:

$$\varphi : \chi \rightarrow \beta$$
$$\psi : \beta \rightarrow \chi$$

Onde φ é a transição que representa o mapeamento dos dados para uma representação aprendida pela rede codificadora e ψ é a transição que representa o mapeamento de uma representação aprendida pela rede decodificadora para gerar elementos dos dados utilizados para o codificador.

Geração de Amostras

Esta parte do experimento se resume a utilizar qualquer biblioteca manipulação de as imagen. Para gerar os dados da imagem, basta a alimentar à rede decodificadora o vetor de representação da classe da imagem que se deseja gerar, que a saída da rede, para cada pixel, será a previsão da mesma sobre qual deverá ser o valor desse pixel. Em seguida, basta utilizar a biblioteca escolhida para efetivamente escrever em disco a imagem. No caso deste trabalho, foi utilizada a biblioteca de manipulação de imagens *Pillow*, para Python.

Método de Avaliação

Para cada conjunto de dados, cada arquitetura será avaliada em função da qualidade das imagens geradas, junto com o tempo necessário para treinar e gerar as mesmas e memória e o consumo de memória. A qualidade em si será determinada conforme as imagens possuírem certas características esperadas da classe associada e não possuírem certas aspectos indesejados. Será detalhado, abaixo, os critérios desejados e indesejados de cada classe, em cada conjunto de dados. Em seguida, será exposta a função para atribuir a nota final para cada arquitetura.

MNIST

Para o conjunto de dados MNIST, serão avaliados os seguintes critérios, em cima das 10 classes presentes, considerados positivos:

- **Traço contínuo**
- **Traço forte**
- **Facilmente identificável**
- **Forma correta**
- **Variedade**

A cada item será atribuído um valor de 2 pontos. Serão avaliados, também, alguns aspectos considerados negativos:

- **Traço inacabado**
- **Presença de elementos inesperados**
- **Forma incorreta ou imprecisa**

A estes itens será atribuído o valor de -2 pontos. A nota atribuída a cada exemplo gerado de cada classe será na forma do somatório dos itens positivos e negativos. Ocasionalmente, dependendo do tamanho do modelo e do tempo de treinamento, podem ser gerados instâncias completamente irreconhecíveis. Nesses casos, a nota atribuída àquela instância é automaticamente zerada. Note que não existe, necessariamente, um ponto negativo para cada ponto positivo, ou melhor, a ausência de um determinado ponto positivo não acarreta em um ponto negativo. Neste caso, a instância sendo avaliada simplesmente deixa de ganhar pontos. A equação para uma determinada instância fica, então, assim:

$$N_i^c = \sum_{n=0}^{k} 2 \cdot P_n^+ - \sum_{m=0}^{l} 2 \cdot P_m^-$$

Equação de avaliação de cada instância de classe, onde P_n^+ é o *n-ésimo* ponto positivo sendo avaliado, P_m^- é o m-*ésimo* ponto negativo sendo avaliado e N_i^c é a nota de uma determinada instância de uma classe *c*.

Visto que, ao final do treinamento, são gerados múltiplos exemplos de cada classe, será realizada uma simples média aritmética entre as notas de cada instância da mesma classe, ou seja:

$$N_c = avg([N_1^c, N_2^c, ..., N_n^c])$$

Para contemplar os experimentos em que a rede gera um número desigual de instâncias de cada classe, a etapa seguinte para calcular a nota da arquitetura será dada pela média aritmética ponderada das notas de cada classe:

$$N_A = w_avg([N_1, N_2, ..., N_c])$$

Por fim, serão incluídas as pontuações referentes ao tempo necessário para treinar as redes, o consumo de memória e o tempo de geração de imagens.

Considerando que o ambiente onde foi realizado os experimentos fez uso de CPU e memória principal, o tempo de treino terá um peso maior e o consumo de memória terá um peso menor, pouco significativo. Outro ponto a ser considerado é quanto ao tempo de geração das imagens: como será visto adiante, os tempos de geração não são muito significativos, levando apenas alguns segundos para todas as arquiteturas, portanto, o peso atribuído a este critério também será menor.

Adicionalmente, visto que esses critérios - tempo de treino, memória utilizada e tempo de geração - não possuem um limite superior e podem ter diferenças muito grandes, os valores serão normalizados para o intervalo **[0, 1]**, pegando o maior valor observado quando este varia durante o treino (no caso, a memória utilizada, pois esta oscilava ao longo do tempo), já que isso pode vir a ser um fator limitador. Para evitar valores iguais a **0** e **1**, após normalizá-los, o cálculo de normalização será realizado da seguinte forma:

$$f(value, max, min) = (value - (min - 1)) \div ((max + 1) - min)$$

Onde *max* é o maior valor da lista sendo avaliada (i.e., entre todos os valores de tempo) e *min* é o menor valor da lista sendo avaliada.

Já que valores menores, nesse caso, são melhores, o valor encontrado será, então, subtraído de **1**. A nota final da arquitetura sendo avaliada será, então, calculada da seguinte maneira:

$$N_F = N_A \cdot (P_T \cdot (1 - T_{N)}) \cdot (P_M \cdot (1 - M_N)) \cdot (P_G \cdot (1 - G_N))$$

onde a nota final N_F é o produto entre a nota da amostragem N_A e os produtos dos pesos de tempo de treinamento P_T, consumo de memória P_M e tempo de geração P_G e os respectivos valores normalizados, com:

1. P_T tendo peso 4,0
2. P_M tendo peso 1,5 e
3. P_G tendo peso 1,1

Parâmetros dos Experimentos

Os parâmetros escolhidos para serem mantidos fixos são o **otimizador**, o **passo de aprendizagem**, e o **batch size**. Para as funções de ativação, são utilizadas conforme detalhados nos respectivos trabalhos originais.

Épocas Treinadas	Batch Size	Otimizador	Passo de Aprendizagem
25	64	Adam Optimizer	2×10^{-4}

A escolha inicial de número de épocas a serem treinadas é arbitrária e serve mais como um ponto de partida para cada arquitetura. Algumas arquiteturas, para determinado tamanho de modelo, podem se beneficiar de mais épocas de treinamento para gerarem imagens melhores. Nesses casos, são realizados experimentos com mais épocas e/ou tamanhos de modelos diferentes. Em outros casos, é possível que a

rede estabilize antes do planejado, fazendo com que épocas de treinamento adicionais sejam desnecessárias.

Resultados

MNIST - DRAW			
Épocas (para estabilizar)	**Duração - Treino (Minutos)**	**Memória Utilizada (MB)**	**Duração - Geração (Segundos)**
10000	312m	400-500MB	2s

MNIST - DCGAN			
Épocas (para estabilizar)	**Duração - Treino (Minutos)**	**Memória Utilizada (MB)**	**Duração - Geração (Segundos)**
25	808m	600-700MB	1,4s

MNIST - PixelCNN			
Épocas (para estabilizar)	**Duração - Treino (Minutos)**	**Memória Utilizada (MB)**	**Duração - Geração (Segundos)**
25	~410m	~1100MB	8s
35	~580m	~1100MB	8s
50	~820m	~1100MB	8s
50	~2150m	~2000MB	10s

Cifar10 - PixelCNN			
Épocas (para estabilizar)	**Duração - Treino (Minutos)**	**Memória Utilizada (MB)**	**Duração - Geração (Segundos)**
50	2650	~2000MB	8s

Notas MNIST

Normalizado	DRAW	DCGAN	PixelCNN

Tempo de Treino	0,00054	0,27010	0,99945
Memória	0,00066	0,13382	0,99933
Tempo de Geração	0,15094	0,09433	0,90566

Arquitetura	Equação da Nota	NF
DRAW	$(1 - 0,00054) \cdot 4 \cdot (1 - 0,00066) \cdot 1,5 \cdot (1 - 0,15094) \cdot 1,1 \cdot 7,06$	39.51
DCGAN	$(1 - 0,27010) \cdot 4 \cdot (1 - 0,13382) \cdot 1,5 \cdot (1 - 0,09433) \cdot 1,1 \cdot 7,21$	27.25
PixelCNN	$(1 - 0,99945) \cdot 4 \cdot (1 - 0,99933) \cdot 1,5 \cdot (1 - 0,90566) \cdot 1,1 \cdot 1,52$	$3.48e^{-7}$

Em uma análise inicial já se percebe claramente a disparidade entre a PixelCNN e as outras arquiteturas, para o MNIST. A PixelCNN precisaria de mais épocas de treinamento (mais tempo) ou mais camadas ocultas (mais tempo e mais memória) para obter resultados melhores (ou até mais épocas *e* mais camadas ocultas). Por outro lado, teve resultados satisfatórios com o Cifar10, apesar de ainda manter os tempos altos de treinamento.

Considerando os resultados que os autores que propuseram esta arquitetura obtiveram, sabe-se que é possível gerar imagens bem variadas e bem realistas, mas talvez fique inviável para o usuário pelo fator tempo se não for utilizado um ambiente com mais recursos, como placas gráficas (e, aqui, o limitador financeiro) e clusters para realizar os treinos.

Já para o DCGAN e o DRAW é um resultado interessante, pois foram obtidos resultados que mostram o DCGAN gerando mais resultados melhores, dando uma nota intermediária (a média ponderada das classes), ao custo um pouco mais de tempo e de memória, resultado em uma nota final um pouco mais baixa.

Considerando o MNIST, o DRAW parece ser o mais apropriado, visto que conseguiu gerar imagens satisfatórias em menos tempo e consumindo menos memória. Seria interessante realizar testes entre as duas arquiteturas em cima dos outros conjuntos para ver como que cada uma se comporta. No trabalho original do DRAW, os autores conseguiram gerar imagens realistas com uma boa fidelidade às estruturas, formas e cores das fotos originais.

Pelo trabalho original da DCGAN, os autores mostram como esta arquitetura consegue resultados muito bons nos conjuntos de dados *Imagenet-1k*, *Large-scale Scene Understanding* e o *Faces Dataset*, o que significa um altíssimo potencial de variedade de imagens geradas de diferentes classes.

4. Conclusão e Trabalhos Futuros

Resolver o problema de síntese de imagens, seja com redes neurais ou outra abordagem, não é um problema fácil, e pode ser abordado de diversas maneiras. Este trabalho estudou três arquiteturas diferentes de redes neurais para avaliar e demonstrar a capacidade destas de gerar imagens realistas e diversificadas - uma

habilidade extremamente útil para a geração de componentes gráficos para jogos. É fato que é recomendado (às vezes imprescindível) ter bastante recursos computacionais ou então ficar sujeito a tempos extensos de treinamento. Ainda mais que, ao lidar com uma abordagem como redes neurais, o espaço de busca é muito massivo, dada a quantidade de parâmetros envolvidos. Independentemente disso, os resultados mostram-se promissores, com imagens sendo geradas com estruturas reconhecíveis e em tempo hábil (na maioria dos casos), especialmente ao considerar os resultados obtidos pelos respectivos trabalhos originais.

A cada ano que passa, o mercado vê a entrada de novas placas gráficas, com arquiteturas cada vez mais especializadas para o uso de redes neurais, e as existentes vão ficando mais baratas, permitindo que cada vez mais pessoas possam ter acesso a equipamento apropriado para realizar experimentos como este. Com isso, aumenta-se o potencial espaço de busca sendo explorado quanto às tarefas que redes neurais conseguem resolver.

Uma limitação que é observado ao utilizar conjuntos como o MNIST e o Cifar10, é o tamanho das imagens originais. Por se tratar de imagens pequenas (28x28 e 32x32 pixels para o MNIST e Cifar10, respectivamente), as arquiteturas são feitas para gerar imagens do mesmo tamanho. Em muitos casos, é desejável poder ter imagens maiores, pois, com apenas 28x28 pixels, aumentar o tamanho da imagem (sem melhorar a resolução) rapidamente degrada a qualidade da mesma. Então, seria interessante ver em um trabalho futuro serem exploradas abordagens para conseguir gerar imagens grandes a partir de representações treinadas de imagens pequenas.

Dessa forma, com um ecossistema de usuários contribuindo com modelos capazes de gerar imagens das mais diversas coisas, o esforço individual para colocar as mãos em um rico conjunto de componentes gráficos fica substancialmente reduzido, que, no final das contas, é o problema principal que este trabalho tenta atacar para resolver.

5. Referências

BISHOP, C.M. Neural Networks for Pattern Recognition(livro), 1995.

COPELAND, Jack. A Brief History of Computing at AlanTuring.net, 2000.

GOODFELLOW, I., et al. Generative adversarial nets. NIPS, 2014.

GRANT, Eugene F.; Lardner. The Talk of the Town – It, The New Yorker, Rex (2 de Agosto, 1952).

GRAVES, A. Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850, 2013.

GRAVES, A., WAYNE, G., & DANIHELKA, I.. Neural turing machines. arXiv preprint arXiv:1410.5401, 2014.

GREGOR, K., DANIHELKA, I., GRAVES, A., REZENDE, D.J., and WIERSTRA, D. DRAW: A recurrent neural network for image generation. Proceedings of the 32nd International Conference on Machine Learning, 2015.

OORD, A., KALCHBRENNER, N., KAVUKCUOGLU, K. Pixel Recurrent Neural Networks. Disponível em https://arxiv.org/pdf/1601.06759v3.pdf, 2016a.

OORD, A., KALCHBRENNER, N., KAVUKCUOGLU, K., VINYALS, O., ESPEHOLT, L., GRAVES, A. Conditional Image Generation With PixelCNN Decoders. Disponível em https://arxiv.org/pdf/1606.05328v2.pdf, 2016b.

RADFORD, A. & CHINTALA, S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. Disponível em https://arxiv.org/pdf/1511.06434.pdf, 2016.

THEIS, Lucas & BETHGE, Matthias. Generative Image Modeling Using Spatial LSTMs. Disponível em https://arxiv.org/pdf/1506.03478v2.pdf, 2015

TIELEMAN, T. Optimizing Neural Networks that Generate Images. Graduate Department of Computer Science, University of Toronto, 2014.

TOGELIUS, Jain et al. Search-based Procedural Content Generation: A Taxonomy and Survey. Computational Intelligence and AI in Games, IEEE Transaction(pág 172-186), 2011.

TOGELIUS, Jain et al. Autoencoders for Level Generation, Repair, and Recognition. ICCC Workshop on Computational Creativity and Games, 2016.

POWLEY, Edward J. et al. Automated Tweaking of Levels for Casual Creation of Mobile Games. Proceedings of the 2nd Computational Creativity and Games Workshop, 2016.

APÊNDICE B - Código

DRAW (disponível em *https://github.com/rafesc/draw*, baseado no código de *https://github.com/ericjang*)

draw.py

```
#!/usr/bin/env python

import tensorflow as tf
from tensorflow.examples.tutorials import mnist
import numpy as np
import os

tf.flags.DEFINE_string("data_dir", "", "")
tf.flags.DEFINE_boolean("read_attn", True, "enable attention for reader")
tf.flags.DEFINE_boolean("write_attn", True, "enable attention for writer")
FLAGS = tf.flags.FLAGS

## MODEL PARAMETERS ##

A, B = 28, 28  # image width,height
img_size = B * A  # the canvas size
enc_size = 256  # number of hidden units / output size in LSTM
dec_size = 256
read_n = 5  # read glimpse grid width/height
write_n = 5  # write glimpse grid width/height
read_size = 2 * read_n * read_n if FLAGS.read_attn else 2 * img_size
write_size = write_n * write_n if FLAGS.write_attn else img_size
z_size = 10  # QSampler output size
T = 10  # MNIST generation sequence length
batch_size = 100  # training minibatch size
train_iters = 10000
learning_rate = 1e-3  # learning rate for optimizer
eps = 1e-8  # epsilon for numerical stability
```

```
## BUILD MODEL ##

DO_SHARE = None  # workaround for variable_scope(reuse=True)

x = tf.placeholder(tf.float32, shape=(batch_size, img_size))  # input (batch_size * img_size)
e = tf.random_normal((batch_size, z_size), mean=0, stddev=1)  # Qsampler noise
lstm_enc = tf.contrib.rnn.LSTMCell(enc_size, state_is_tuple=True)  # encoder Op
lstm_dec = tf.contrib.rnn.LSTMCell(dec_size, state_is_tuple=True)  # decoder Op


def linear(x, output_dim):
    """
    affine transformation Wx+b
    assumes x.shape = (batch_size, num_features)
    """
    w = tf.get_variable("w", [x.get_shape()[1], output_dim])
    b = tf.get_variable("b", [output_dim], initializer=tf.constant_initializer(0.0))
    return tf.matmul(x, w) + b


def filterbank(gx, gy, sigma2, delta, N):
    grid_i = tf.reshape(tf.cast(tf.range(N), tf.float32), [1, -1])
    mu_x = gx + (grid_i - N / 2 - 0.5) * delta  # eq 19
    mu_y = gy + (grid_i - N / 2 - 0.5) * delta  # eq 20
    a = tf.reshape(tf.cast(tf.range(A), tf.float32), [1, 1, -1])
    b = tf.reshape(tf.cast(tf.range(B), tf.float32), [1, 1, -1])
    mu_x = tf.reshape(mu_x, [-1, N, 1])
    mu_y = tf.reshape(mu_y, [-1, N, 1])
    sigma2 = tf.reshape(sigma2, [-1, 1, 1])
    Fx = tf.exp(-tf.square(a - mu_x) / (2 * sigma2))
    Fy = tf.exp(-tf.square(b - mu_y) / (2 * sigma2))  # batch x N x B
    # normalize, sum over A and B dims
    Fx = Fx / tf.maximum(tf.reduce_sum(Fx, 2, keep_dims=True), eps)
    Fy = Fy / tf.maximum(tf.reduce_sum(Fy, 2, keep_dims=True), eps)
    return Fx, Fy


def attn_window(scope, h_dec, N):
```

```
    with tf.variable_scope(scope, reuse=DO_SHARE):
        params = linear(h_dec, 5)
    # gx_,gy_,log_sigma2,log_delta,log_gamma=tf.split(1,5,params)
    gx_, gy_, log_sigma2, log_delta, log_gamma = tf.split(params, 5, 1)
    gx = (A + 1) / 2 * (gx_ + 1)
    gy = (B + 1) / 2 * (gy_ + 1)
    sigma2 = tf.exp(log_sigma2)
    delta = (max(A, B) - 1) / (N - 1) * tf.exp(log_delta)  # batch x N
    return filterbank(gx, gy, sigma2, delta, N) + (tf.exp(log_gamma),)


## READ ##
def read_no_attn(x, x_hat, h_dec_prev):
    return tf.concat([x, x_hat], 1)


def read_attn(x, x_hat, h_dec_prev):
    Fx, Fy, gamma = attn_window("read", h_dec_prev, read_n)

    def filter_img(img, Fx, Fy, gamma, N):
        Fxt = tf.transpose(Fx, perm=[0, 2, 1])
        img = tf.reshape(img, [-1, B, A])
        glimpse = tf.matmul(Fy, tf.matmul(img, Fxt))
        glimpse = tf.reshape(glimpse, [-1, N * N])
        return glimpse * tf.reshape(gamma, [-1, 1])

    x = filter_img(x, Fx, Fy, gamma, read_n)  # batch x (read_n*read_n)
    x_hat = filter_img(x_hat, Fx, Fy, gamma, read_n)
    return tf.concat([x, x_hat], 1)  # concat along feature axis


read = read_attn if FLAGS.read_attn else read_no_attn


## ENCODE ##
def encode(state, input):
    """
    run LSTM
```

```
    state = previous encoder state
    input = cat(read,h_dec_prev)
    returns: (output, new_state)
    """
    with tf.variable_scope("encoder", reuse=DO_SHARE):
        return lstm_enc(input, state)


## Q-SAMPLER (VARIATIONAL AUTOENCODER) ##

def sampleQ(h_enc):
    """
    Samples Zt ~ normrnd(mu,sigma) via reparameterization trick for normal dist
    mu is (batch,z_size)
    """
    with tf.variable_scope("mu", reuse=DO_SHARE):
        mu = linear(h_enc, z_size)
    with tf.variable_scope("sigma", reuse=DO_SHARE):
        logsigma = linear(h_enc, z_size)
        sigma = tf.exp(logsigma)
    return (mu + sigma * e, mu, logsigma, sigma)


## DECODER ##
def decode(state, input):
    with tf.variable_scope("decoder", reuse=DO_SHARE):
        return lstm_dec(input, state)


## WRITER ##
def write_no_attn(h_dec):
    with tf.variable_scope("write", reuse=DO_SHARE):
        return linear(h_dec, img_size)


def write_attn(h_dec):
    with tf.variable_scope("writeW", reuse=DO_SHARE):
        w = linear(h_dec, write_size)  # batch x (write_n*write_n)
```

```
    N = write_n
    w = tf.reshape(w, [batch_size, N, N])
    Fx, Fy, gamma = attn_window("write", h_dec, write_n)
    Fyt = tf.transpose(Fy, perm=[0, 2, 1])
    wr = tf.matmul(Fyt, tf.matmul(w, Fx))
    wr = tf.reshape(wr, [batch_size, B * A])
    # gamma=tf.tile(gamma,[1,B*A])
    return wr * tf.reshape(1.0 / gamma, [-1, 1])



write = write_attn if FLAGS.write_attn else write_no_attn

## STATE VARIABLES ##

cs = [0] * T  # sequence of canvases
mus, logsigmas, sigmas = [0] * T, [0] * T, [
    0] * T  # gaussian params generated by SampleQ. We will need these for computing loss.
# initial states
h_dec_prev = tf.zeros((batch_size, dec_size))
enc_state = lstm_enc.zero_state(batch_size, tf.float32)
dec_state = lstm_dec.zero_state(batch_size, tf.float32)

## DRAW MODEL ##

# construct the unrolled computational graph
for t in range(T):
    c_prev = tf.zeros((batch_size, img_size)) if t == 0 else cs[t - 1]
    x_hat = x - tf.sigmoid(c_prev)  # error image
    r = read(x, x_hat, h_dec_prev)
    h_enc, enc_state = encode(enc_state, tf.concat([r, h_dec_prev], 1))
    z, mus[t], logsigmas[t], sigmas[t] = sampleQ(h_enc)
    h_dec, dec_state = decode(dec_state, z)
    cs[t] = c_prev + write(h_dec)  # store results
    h_dec_prev = h_dec
    DO_SHARE = True  # from now on, share variables



## LOSS FUNCTION ##
```

```
def binary_crossentropy(t, o):
    return -(t * tf.log(o + eps) + (1.0 - t) * tf.log(1.0 - o + eps))


# reconstruction term appears to have been collapsed down to a single scalar value (rather
than one per item in minibatch)
x_recons = tf.nn.sigmoid(cs[-1])

# after computing binary cross entropy, sum across features then take the mean of those
sums across minibatches
Lx = tf.reduce_sum(binary_crossentropy(x, x_recons), 1)  # reconstruction term
Lx = tf.reduce_mean(Lx)

kl_terms = [0] * T
for t in range(T):
    mu2 = tf.square(mus[t])
    sigma2 = tf.square(sigmas[t])
    logsigma = logsigmas[t]
    kl_terms[t] = 0.5 * tf.reduce_sum(mu2 + sigma2 - 2 * logsigma, 1) - .5  # each kl term is
(1xminibatch)
KL = tf.add_n(kl_terms)  # this is 1xminibatch, corresponding to summing kl_terms from 1:T
Lz = tf.reduce_mean(KL)  # average over minibatches

cost = Lx + Lz

## OPTIMIZER ##

optimizer = tf.train.AdamOptimizer(learning_rate, beta1=0.5)
grads = optimizer.compute_gradients(cost)
for i, (g, v) in enumerate(grads):
    if g is not None:
        grads[i] = (tf.clip_by_norm(g, 5), v)  # clip gradients
train_op = optimizer.apply_gradients(grads)

## RUN TRAINING ##

data_directory = os.path.join(FLAGS.data_dir, "mnist")
```

```
if not os.path.exists(data_directory):
    os.makedirs(data_directory)
train_data = mnist.input_data.read_data_sets(data_directory, one_hot=True).train  # binarized
(0-1) mnist data

fetches = []
fetches.extend([Lx, Lz, train_op])
Lxs = [0] * train_iters
Lzs = [0] * train_iters

sess = tf.InteractiveSession()

saver = tf.train.Saver()  # saves variables learned during training
tf.global_variables_initializer().run()
# saver.restore(sess, "/tmp/draw/drawmodel.ckpt") # to restore from model, uncomment this
line

for i in range(train_iters):
    xtrain, _ = train_data.next_batch(batch_size)  # xtrain is (batch_size x img_size)
    feed_dict = {x: xtrain}
    results = sess.run(fetches, feed_dict)
    Lxs[i], Lzs[i], _ = results
    if i % 100 == 0:
        print("iter=%d : Lx: %f Lz: %f" % (i, Lxs[i], Lzs[i]))

## TRAINING FINISHED ##

canvases = sess.run(cs, feed_dict)  # generate some examples
canvases = np.array(canvases)  # T x batch x img_size

out_file = os.path.join(FLAGS.data_dir, "draw_data.npy")
np.save(out_file, [canvases, Lxs, Lzs])
print("Outputs saved in file: %s" % out_file)

ckpt_file = os.path.join(FLAGS.data_dir, "drawmodel.ckpt")
print("Model saved in file: %s" % saver.save(sess, ckpt_file))

sess.close()
```

plot_data.py

```
# takes data saved by DRAW model and generates animations
# example usage: python plot_data.py noattn /tmp/draw/draw_data.npy

import matplotlib
import sys
import numpy as np

interactive = False  # set to False if you want to write images to file

if not interactive:
    matplotlib.use('Agg')  # Force matplotlib to not use any Xwindows backend.
import matplotlib.pyplot as plt


def xrecons_grid(X, B, A):
    """
    plots canvas for single time step
    X is x_recons, (batch_size x img_size)
    assumes features = BxA images
    batch is assumed to be a square number
    """
    padsize = 1
    padval = .5
    ph = B + 2 * padsize
    pw = A + 2 * padsize
    batch_size = X.shape[0]
    N = int(np.sqrt(batch_size))
    X = X.reshape((N, N, B, A))
    img = np.ones((N * ph, N * pw)) * padval
    for i in range(N):
        for j in range(N):
            startr = i * ph + padsize
            endr = startr + B
            startc = j * pw + padsize
            endc = startc + A
            img[startr:endr, startc:endc] = X[i, j, :, :]
    return img


if __name__ == '__main__':
    prefix = sys.argv[1]
    out_file = sys.argv[2]
    [C, Lxs, Lzs] = np.load(out_file)
    T, batch_size, img_size = C.shape
    X = 1.0 / (1.0 + np.exp(-C))  # x_recons=sigmoid(canvas)
    B = A = int(np.sqrt(img_size))
    if interactive:
        f, arr = plt.subplots(1, T)
    for t in range(T):
        img = xrecons_grid(X[t, :, :], B, A)
        if interactive:
```

```
            arr[t].matshow(img, cmap=plt.cm.gray)
            arr[t].set_xticks([])
            arr[t].set_yticks([])
        else:
            plt.matshow(img, cmap=plt.cm.gray)
            imgname = '%s_%d.png' % (
                prefix, t)  # you can merge using imagemagick, i.e. convert -delay 10 -loop 0 *.png
mnist.gif
            plt.savefig(imgname)
            print(imgname)
    f = plt.figure()
    plt.plot(Lxs, label='Reconstruction Loss Lx')
    plt.plot(Lzs, label='Latent Loss Lz')
    plt.xlabel('iterations')
    plt.legend()
    if interactive:
        plt.show()
    else:
        plt.savefig('%s_loss.png' % (prefix))
```

DCGAN (disponível em [*https://github.com/rafesc/DCGAN-tensorflow*](https://github.com/rafesc/DCGAN-tensorflow), baseado no código de *https://github.com/carpedm20*)

download.py

```
from __future__ import print_function
import os
import sys
import json
import zipfile
import argparse
import requests
import subprocess
from tqdm import tqdm
from six.moves import urllib

parser = argparse.ArgumentParser(description='Download dataset for DCGAN.')
parser.add_argument('datasets', metavar='N', type=str, nargs='+', choices=['celebA', 'lsun',
'mnist'],
                    help='name of dataset to download [celebA, lsun, mnist]')


def download(url, dirpath):
    filename = url.split('/')[-1]
    filepath = os.path.join(dirpath, filename)
    u = urllib.request.urlopen(url)
    f = open(filepath, 'wb')
    filesize = int(u.headers["Content-Length"])
    print("Downloading: %s Bytes: %s" % (filename, filesize))
```

```
    downloaded = 0
    block_sz = 8192
    status_width = 70
    while True:
        buf = u.read(block_sz)
        if not buf:
            print('')
            break
        else:
            print('', end='\r')
        downloaded += len(buf)
        f.write(buf)
        status = (("[%-" + str(status_width + 1) + "s] %3.2f%%") %
                  ('=' * int(float(downloaded) / filesize * status_width) + '>', downloaded * 100. /
filesize))
        print(status, end='')
        sys.stdout.flush()
    f.close()
    return filepath


def download_file_from_google_drive(id, destination):
    URL = "https://docs.google.com/uc?export=download"
    session = requests.Session()

    response = session.get(URL, params={'id': id}, stream=True)
    token = get_confirm_token(response)

    if token:
        params = {'id': id, 'confirm': token}
        response = session.get(URL, params=params, stream=True)

    save_response_content(response, destination)


def get_confirm_token(response):
    for key, value in response.cookies.items():
        if key.startswith('download_warning'):
            return value
    return None


def save_response_content(response, destination, chunk_size=32 * 1024):
    total_size = int(response.headers.get('content-length', 0))
    with open(destination, "wb") as f:
        for chunk in tqdm(response.iter_content(chunk_size), total=total_size,
                          unit='B', unit_scale=True, desc=destination):
            if chunk:  # filter out keep-alive new chunks
                f.write(chunk)


def unzip(filepath):
    print("Extracting: " + filepath)
    dirpath = os.path.dirname(filepath)
    with zipfile.ZipFile(filepath) as zf:
        zf.extractall(dirpath)
```

```
    os.remove(filepath)


def download_celeb_a(dirpath):
    data_dir = 'celebA'
    if os.path.exists(os.path.join(dirpath, data_dir)):
        print('Found Celeb-A - skip')
        return

    filename, drive_id = "img_align_celeba.zip", "0B7EVK8r0v71pZjFTYXZWM3FlRnM"
    save_path = os.path.join(dirpath, filename)

    if os.path.exists(save_path):
        print('[*] {} already exists'.format(save_path))
    else:
        download_file_from_google_drive(drive_id, save_path)

    zip_dir = ''
    with zipfile.ZipFile(save_path) as zf:
        zip_dir = zf.namelist()[0]
        zf.extractall(dirpath)
    os.remove(save_path)
    os.rename(os.path.join(dirpath, zip_dir), os.path.join(dirpath, data_dir))


def _list_categories(tag):
    url = 'http://lsun.cs.princeton.edu/htbin/list.cgi?tag=' + tag
    f = urllib.request.urlopen(url)
    return json.loads(f.read())


def _download_lsun(out_dir, category, set_name, tag):
    url = 'http://lsun.cs.princeton.edu/htbin/download.cgi?tag={tag}' \
          '&category={category}&set={set_name}'.format(**locals())
    print(url)
    if set_name == 'test':
        out_name = 'test_lmdb.zip'
    else:
        out_name = '{category}_{set_name}_lmdb.zip'.format(**locals())
    out_path = os.path.join(out_dir, out_name)
    cmd = ['curl', url, '-o', out_path]
    print('Downloading', category, set_name, 'set')
    subprocess.call(cmd)


def download_lsun(dirpath):
    data_dir = os.path.join(dirpath, 'lsun')
    if os.path.exists(data_dir):
        print('Found LSUN - skip')
        return
    else:
        os.mkdir(data_dir)

    tag = 'latest'
    # categories = _list_categories(tag)
    categories = ['bedroom']
```

```
    for category in categories:
        _download_lsun(data_dir, category, 'train', tag)
        _download_lsun(data_dir, category, 'val', tag)
    _download_lsun(data_dir, '', 'test', tag)


def download_mnist(dirpath):
    data_dir = os.path.join(dirpath, 'mnist')
    if os.path.exists(data_dir):
        print('Found MNIST - skip')
        return
    else:
        os.mkdir(data_dir)
    url_base = 'http://yann.lecun.com/exdb/mnist/'
    file_names = ['train-images-idx3-ubyte.gz',
                  'train-labels-idx1-ubyte.gz',
                  't10k-images-idx3-ubyte.gz',
                  't10k-labels-idx1-ubyte.gz']
    for file_name in file_names:
        url = (url_base + file_name).format(**locals())
        print(url)
        out_path = os.path.join(data_dir, file_name)
        cmd = ['curl', url, '-o', out_path]
        print('Downloading ', file_name)
        subprocess.call(cmd)
        cmd = ['gzip', '-d', out_path]
        print('Decompressing ', file_name)
        subprocess.call(cmd)


def prepare_data_dir(path='./data'):
    if not os.path.exists(path):
        os.mkdir(path)


if __name__ == '__main__':
    args = parser.parse_args()
    prepare_data_dir()

    if any(name in args.datasets for name in ['CelebA', 'celebA', 'celebA']):
        download_celeb_a('./data')
    if 'lsun' in args.datasets:
        download_lsun('./data')
    if 'mnist' in args.datasets:
        download_mnist('./data')
```

main.py

```
import os
import numpy as np

from model import DCGAN
from utils import pp, visualize, show_all_variables
```

```
import tensorflow as tf

flags = tf.app.flags
flags.DEFINE_integer("epoch", 25, "Epoch to train [25]")
flags.DEFINE_float("learning_rate", 0.0002, "Learning rate of for adam [0.0002]")
flags.DEFINE_float("beta1", 0.5, "Momentum term of adam [0.5]")
flags.DEFINE_float("train_size", np.inf, "The size of train images [np.inf]")
flags.DEFINE_integer("batch_size", 64, "The size of batch images [64]")
flags.DEFINE_integer("input_height", 108, "The size of image to use (will be center cropped).
[108]")
flags.DEFINE_integer("input_width", None,
                "The size of image to use (will be center cropped). If None, same value as
input_height [None]")
flags.DEFINE_integer("output_height", 64, "The size of the output images to produce [64]")
flags.DEFINE_integer("output_width", None,
                "The size of the output images to produce. If None, same value as output_height
[None]")
flags.DEFINE_string("dataset", "celebA", "The name of dataset [celebA, mnist, lsun]")
flags.DEFINE_string("input_fname_pattern", "*.jpg", "Glob pattern of filename of input images
[*]")
flags.DEFINE_string("checkpoint_dir", "checkpoint", "Directory name to save the checkpoints
[checkpoint]")
flags.DEFINE_string("data_dir", "./data", "Root directory of dataset [data]")
flags.DEFINE_string("sample_dir", "samples", "Directory name to save the image samples
[samples]")
flags.DEFINE_boolean("train", False, "True for training, False for testing [False]")
flags.DEFINE_boolean("crop", False, "True for training, False for testing [False]")
flags.DEFINE_boolean("visualize", False, "True for visualizing, False for nothing [False]")
flags.DEFINE_integer("generate_test_images", 100, "Number of images to generate during
test. [100]")
FLAGS = flags.FLAGS


def main(_):
   pp.pprint(flags.FLAGS.__flags)

   if FLAGS.input_width is None:
      FLAGS.input_width = FLAGS.input_height
   if FLAGS.output_width is None:
      FLAGS.output_width = FLAGS.output_height

   if not os.path.exists(FLAGS.checkpoint_dir):
      os.makedirs(FLAGS.checkpoint_dir)
   if not os.path.exists(FLAGS.sample_dir):
      os.makedirs(FLAGS.sample_dir)

   # gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.333)
   run_config = tf.ConfigProto()
   run_config.gpu_options.allow_growth = True

   with tf.Session(config=run_config) as sess:
      if FLAGS.dataset == 'mnist':
         dcgan = DCGAN(
            sess,
            input_width=FLAGS.input_width,
            input_height=FLAGS.input_height,
```

```
            output_width=FLAGS.output_width,
            output_height=FLAGS.output_height,
            batch_size=FLAGS.batch_size,
            sample_num=FLAGS.batch_size,
            y_dim=10,
            z_dim=FLAGS.generate_test_images,
            dataset_name=FLAGS.dataset,
            input_fname_pattern=FLAGS.input_fname_pattern,
            crop=FLAGS.crop,
            checkpoint_dir=FLAGS.checkpoint_dir,
            sample_dir=FLAGS.sample_dir,
            data_dir=FLAGS.data_dir)
    else:
        dcgan = DCGAN(
            sess,
            input_width=FLAGS.input_width,
            input_height=FLAGS.input_height,
            output_width=FLAGS.output_width,
            output_height=FLAGS.output_height,
            batch_size=FLAGS.batch_size,
            sample_num=FLAGS.batch_size,
            z_dim=FLAGS.generate_test_images,
            dataset_name=FLAGS.dataset,
            input_fname_pattern=FLAGS.input_fname_pattern,
            crop=FLAGS.crop,
            checkpoint_dir=FLAGS.checkpoint_dir,
            sample_dir=FLAGS.sample_dir,
            data_dir=FLAGS.data_dir)

    show_all_variables()

    if FLAGS.train:
        dcgan.train(FLAGS)
    else:
        if not dcgan.load(FLAGS.checkpoint_dir)[0]:
            raise Exception("[!] Train a model first, then run test mode")

    # to_json("./web/js/layers.js", [dcgan.h0_w, dcgan.h0_b, dcgan.g_bn0],
    #                 [dcgan.h1_w, dcgan.h1_b, dcgan.g_bn1],
    #                 [dcgan.h2_w, dcgan.h2_b, dcgan.g_bn2],
    #                 [dcgan.h3_w, dcgan.h3_b, dcgan.g_bn3],
    #                 [dcgan.h4_w, dcgan.h4_b, None])

    # Below is codes for visualization
    OPTION = 1
    visualize(sess, dcgan, FLAGS, OPTION)


if __name__ == '__main__':
    tf.app.run()
```

model.py

```
from __future__ import division
import os
import time
from glob import glob

from ops import *
from utils import *


def conv_out_size_same(size, stride):
   return int(math.ceil(float(size) / float(stride)))


class DCGAN(object):
   def __init__(self, sess, input_height=108, input_width=108, crop=True,
              batch_size=64, sample_num=64, output_height=64, output_width=64,
              y_dim=None, z_dim=100, gf_dim=64, df_dim=64,
              gfc_dim=1024, dfc_dim=1024, c_dim=3, dataset_name='default',
              input_fname_pattern='*.jpg', checkpoint_dir=None, sample_dir=None,
data_dir='./data'):
       """

       Args:
         sess: TensorFlow session
         batch_size: The size of batch. Should be specified before training.
         y_dim: (optional) Dimension of dim for y. [None]
         z_dim: (optional) Dimension of dim for Z. [100]
         gf_dim: (optional) Dimension of gen filters in first conv layer. [64]
         df_dim: (optional) Dimension of discrim filters in first conv layer. [64]
         gfc_dim: (optional) Dimension of gen units for for fully connected layer. [1024]
         dfc_dim: (optional) Dimension of discrim units for fully connected layer. [1024]
         c_dim: (optional) Dimension of image color. For grayscale input, set to 1. [3]
       """
       self.sess = sess
       self.crop = crop

       self.batch_size = batch_size
       self.sample_num = sample_num

       self.input_height = input_height
       self.input_width = input_width
       self.output_height = output_height
       self.output_width = output_width

       self.y_dim = y_dim
       self.z_dim = z_dim

       self.gf_dim = gf_dim
       self.df_dim = df_dim

       self.gfc_dim = gfc_dim
       self.dfc_dim = dfc_dim

       # batch normalization : deals with poor initialization helps gradient flow
       self.d_bn1 = batch_norm(name='d_bn1')
       self.d_bn2 = batch_norm(name='d_bn2')
```

```
        if not self.y_dim:
            self.d_bn3 = batch_norm(name='d_bn3')

        self.g_bn0 = batch_norm(name='g_bn0')
        self.g_bn1 = batch_norm(name='g_bn1')
        self.g_bn2 = batch_norm(name='g_bn2')

        if not self.y_dim:
            self.g_bn3 = batch_norm(name='g_bn3')

        self.dataset_name = dataset_name
        self.input_fname_pattern = input_fname_pattern
        self.checkpoint_dir = checkpoint_dir
        self.data_dir = data_dir

        if self.dataset_name == 'mnist':
            self.data_X, self.data_y = self.load_mnist()
            self.c_dim = self.data_X[0].shape[-1]
        else:
            data_path = os.path.join(self.data_dir, self.dataset_name, self.input_fname_pattern)
            self.data = glob(data_path)
            if len(self.data) == 0:
                raise Exception("[!] No data found in '" + data_path + "'")
            np.random.shuffle(self.data)
            imreadImg = imread(self.data[0])
            if len(imreadImg.shape) >= 3:  # check if image is a non-grayscale image by checking
channel number
                self.c_dim = imread(self.data[0]).shape[-1]
            else:
                self.c_dim = 1

            if len(self.data) < self.batch_size:
                raise Exception("[!] Entire dataset size is less than the configured batch_size")

        self.grayscale = (self.c_dim == 1)

        self.build_model()

    def build_model(self):
        if self.y_dim:
            self.y = tf.placeholder(tf.float32, [self.batch_size, self.y_dim], name='y')
        else:
            self.y = None

        if self.crop:
            image_dims = [self.output_height, self.output_width, self.c_dim]
        else:
            image_dims = [self.input_height, self.input_width, self.c_dim]

        self.inputs = tf.placeholder(
            tf.float32, [self.batch_size] + image_dims, name='real_images')

        inputs = self.inputs

        self.z = tf.placeholder(
            tf.float32, [None, self.z_dim], name='z')
        self.z_sum = histogram_summary("z", self.z)
```

```
    self.G = self.generator(self.z, self.y)
    self.D, self.D_logits = self.discriminator(inputs, self.y, reuse=False)
    self.sampler = self.sampler(self.z, self.y)
    self.D_, self.D_logits_ = self.discriminator(self.G, self.y, reuse=True)

    self.d_sum = histogram_summary("d", self.D)
    self.d__sum = histogram_summary("d_", self.D_)
    self.G_sum = image_summary("G", self.G)

    def sigmoid_cross_entropy_with_logits(x, y):
      try:
        return tf.nn.sigmoid_cross_entropy_with_logits(logits=x, labels=y)
      except:
        return tf.nn.sigmoid_cross_entropy_with_logits(logits=x, targets=y)

    self.d_loss_real = tf.reduce_mean(
      sigmoid_cross_entropy_with_logits(self.D_logits, tf.ones_like(self.D)))
    self.d_loss_fake = tf.reduce_mean(
      sigmoid_cross_entropy_with_logits(self.D_logits_, tf.zeros_like(self.D_)))
    self.g_loss = tf.reduce_mean(
      sigmoid_cross_entropy_with_logits(self.D_logits_, tf.ones_like(self.D_)))

    self.d_loss_real_sum = scalar_summary("d_loss_real", self.d_loss_real)
    self.d_loss_fake_sum = scalar_summary("d_loss_fake", self.d_loss_fake)

    self.d_loss = self.d_loss_real + self.d_loss_fake

    self.g_loss_sum = scalar_summary("g_loss", self.g_loss)
    self.d_loss_sum = scalar_summary("d_loss", self.d_loss)

    t_vars = tf.trainable_variables()

    self.d_vars = [var for var in t_vars if 'd_' in var.name]
    self.g_vars = [var for var in t_vars if 'g_' in var.name]

    self.saver = tf.train.Saver()

  def train(self, config):
    d_optim = tf.train.AdamOptimizer(config.learning_rate, beta1=config.beta1) \
      .minimize(self.d_loss, var_list=self.d_vars)
    g_optim = tf.train.AdamOptimizer(config.learning_rate, beta1=config.beta1) \
      .minimize(self.g_loss, var_list=self.g_vars)
    try:
      tf.global_variables_initializer().run()
    except:
      tf.initialize_all_variables().run()

    self.g_sum = merge_summary([self.z_sum, self.d__sum,
                                self.G_sum, self.d_loss_fake_sum, self.g_loss_sum])
    self.d_sum = merge_summary(
      [self.z_sum, self.d_sum, self.d_loss_real_sum, self.d_loss_sum])
    self.writer = SummaryWriter("./logs", self.sess.graph)

    sample_z = np.random.uniform(-1, 1, size=(self.sample_num, self.z_dim))

    if config.dataset == 'mnist':
```

```
    sample_inputs = self.data_X[0:self.sample_num]
    sample_labels = self.data_y[0:self.sample_num]
else:
    sample_files = self.data[0:self.sample_num]
    sample = [
        get_image(sample_file,
                  input_height=self.input_height,
                  input_width=self.input_width,
                  resize_height=self.output_height,
                  resize_width=self.output_width,
                  crop=self.crop,
                  grayscale=self.grayscale) for sample_file in sample_files]
    if (self.grayscale):
        sample_inputs = np.array(sample).astype(np.float32)[:, :, :, None]
    else:
        sample_inputs = np.array(sample).astype(np.float32)

counter = 1
start_time = time.time()
could_load, checkpoint_counter = self.load(self.checkpoint_dir)
if could_load:
    counter = checkpoint_counter
    print(" [*] Load SUCCESS")
else:
    print(" [!] Load failed...")

for epoch in xrange(config.epoch):
    if config.dataset == 'mnist':
        batch_idxs = min(len(self.data_X), config.train_size) // config.batch_size
    else:
        self.data = glob(os.path.join(
            config.data_dir, config.dataset, self.input_fname_pattern))
        np.random.shuffle(self.data)
        batch_idxs = min(len(self.data), config.train_size) // config.batch_size

    for idx in xrange(0, int(batch_idxs)):
        if config.dataset == 'mnist':
            batch_images = self.data_X[idx * config.batch_size:(idx + 1) * config.batch_size]
            batch_labels = self.data_y[idx * config.batch_size:(idx + 1) * config.batch_size]
        else:
            batch_files = self.data[idx * config.batch_size:(idx + 1) * config.batch_size]
            batch = [
                get_image(batch_file,
                          input_height=self.input_height,
                          input_width=self.input_width,
                          resize_height=self.output_height,
                          resize_width=self.output_width,
                          crop=self.crop,
                          grayscale=self.grayscale) for batch_file in batch_files]
            if self.grayscale:
                batch_images = np.array(batch).astype(np.float32)[:, :, :, None]
            else:
                batch_images = np.array(batch).astype(np.float32)

        batch_z = np.random.uniform(-1, 1, [config.batch_size, self.z_dim]) \
            .astype(np.float32)
```

```
            if config.dataset == 'mnist':
              # Update D network
              _, summary_str = self.sess.run([d_optim, self.d_sum],
                                              feed_dict={
                                                self.inputs: batch_images,
                                                self.z: batch_z,
                                                self.y: batch_labels,
                                              })
              self.writer.add_summary(summary_str, counter)

              # Update G network
              _, summary_str = self.sess.run([g_optim, self.g_sum],
                                              feed_dict={
                                                self.z: batch_z,
                                                self.y: batch_labels,
                                              })
              self.writer.add_summary(summary_str, counter)

              # Run g_optim twice to make sure that d_loss does not go to zero (different from
paper)
              _, summary_str = self.sess.run([g_optim, self.g_sum],
                                              feed_dict={self.z: batch_z, self.y: batch_labels})
              self.writer.add_summary(summary_str, counter)

              errD_fake = self.d_loss_fake.eval({
                self.z: batch_z,
                self.y: batch_labels
              })
              errD_real = self.d_loss_real.eval({
                self.inputs: batch_images,
                self.y: batch_labels
              })
              errG = self.g_loss.eval({
                self.z: batch_z,
                self.y: batch_labels
              })
            else:
              # Update D network
              _, summary_str = self.sess.run([d_optim, self.d_sum],
                                              feed_dict={self.inputs: batch_images, self.z: batch_z})
              self.writer.add_summary(summary_str, counter)

              # Update G network
              _, summary_str = self.sess.run([g_optim, self.g_sum],
                                              feed_dict={self.z: batch_z})
              self.writer.add_summary(summary_str, counter)

              # Run g_optim twice to make sure that d_loss does not go to zero (different from
paper)
              _, summary_str = self.sess.run([g_optim, self.g_sum],
                                              feed_dict={self.z: batch_z})
              self.writer.add_summary(summary_str, counter)

              errD_fake = self.d_loss_fake.eval({self.z: batch_z})
              errD_real = self.d_loss_real.eval({self.inputs: batch_images})
              errG = self.g_loss.eval({self.z: batch_z})
```

```
            counter += 1
            print("Epoch: [%2d/%2d] [%4d/%4d] time: %4.4f, d_loss: %.8f, g_loss: %.8f" \
                % (epoch, config.epoch, idx, batch_idxs,
                  time.time() - start_time, errD_fake + errD_real, errG))

            if np.mod(counter, 100) == 1:
                if config.dataset == 'mnist':
                    samples, d_loss, g_loss = self.sess.run(
                        [self.sampler, self.d_loss, self.g_loss],
                        feed_dict={
                            self.z: sample_z,
                            self.inputs: sample_inputs,
                            self.y: sample_labels,
                        }
                    )
                    save_images(samples, image_manifold_size(samples.shape[0]),
                                './{}/train_{:02d}_{:04d}.png'.format(config.sample_dir, epoch, idx))
                    print("[Sample] d_loss: %.8f, g_loss: %.8f" % (d_loss, g_loss))
                else:
                    try:
                        samples, d_loss, g_loss = self.sess.run(
                            [self.sampler, self.d_loss, self.g_loss],
                            feed_dict={
                                self.z: sample_z,
                                self.inputs: sample_inputs,
                            },
                        )
                        save_images(samples, image_manifold_size(samples.shape[0]),
                                    './{}/train_{:02d}_{:04d}.png'.format(config.sample_dir, epoch, idx))
                        print("[Sample] d_loss: %.8f, g_loss: %.8f" % (d_loss, g_loss))
                    except:
                        print("one pic error!...")

            if np.mod(counter, 500) == 2:
                self.save(config.checkpoint_dir, counter)

  def discriminator(self, image, y=None, reuse=False):
    with tf.variable_scope("discriminator") as scope:
      if reuse:
        scope.reuse_variables()

      if not self.y_dim:
        h0 = lrelu(conv2d(image, self.df_dim, name='d_h0_conv'))
        h1 = lrelu(self.d_bn1(conv2d(h0, self.df_dim * 2, name='d_h1_conv')))
        h2 = lrelu(self.d_bn2(conv2d(h1, self.df_dim * 4, name='d_h2_conv')))
        h3 = lrelu(self.d_bn3(conv2d(h2, self.df_dim * 8, name='d_h3_conv')))
        h4 = linear(tf.reshape(h3, [self.batch_size, -1]), 1, 'd_h4_lin')

        return tf.nn.sigmoid(h4), h4
      else:
        yb = tf.reshape(y, [self.batch_size, 1, 1, self.y_dim])
        x = conv_cond_concat(image, yb)

        h0 = lrelu(conv2d(x, self.c_dim + self.y_dim, name='d_h0_conv'))
        h0 = conv_cond_concat(h0, yb)

        h1 = lrelu(self.d_bn1(conv2d(h0, self.df_dim + self.y_dim, name='d_h1_conv')))
```

```
            h1 = tf.reshape(h1, [self.batch_size, -1])
            h1 = concat([h1, y], 1)

            h2 = lrelu(self.d_bn2(linear(h1, self.dfc_dim, 'd_h2_lin')))
            h2 = concat([h2, y], 1)

            h3 = linear(h2, 1, 'd_h3_lin')

            return tf.nn.sigmoid(h3), h3

    def generator(self, z, y=None):
        with tf.variable_scope("generator") as scope:
            if not self.y_dim:
                s_h, s_w = self.output_height, self.output_width
                s_h2, s_w2 = conv_out_size_same(s_h, 2), conv_out_size_same(s_w, 2)
                s_h4, s_w4 = conv_out_size_same(s_h2, 2), conv_out_size_same(s_w2, 2)
                s_h8, s_w8 = conv_out_size_same(s_h4, 2), conv_out_size_same(s_w4, 2)
                s_h16, s_w16 = conv_out_size_same(s_h8, 2), conv_out_size_same(s_w8, 2)

                # project `z` and reshape
                self.z_, self.h0_w, self.h0_b = linear(
                    z, self.gf_dim * 8 * s_h16 * s_w16, 'g_h0_lin', with_w=True)

                self.h0 = tf.reshape(
                    self.z_, [-1, s_h16, s_w16, self.gf_dim * 8])
                h0 = tf.nn.relu(self.g_bn0(self.h0))

                self.h1, self.h1_w, self.h1_b = deconv2d(
                    h0, [self.batch_size, s_h8, s_w8, self.gf_dim * 4], name='g_h1', with_w=True)
                h1 = tf.nn.relu(self.g_bn1(self.h1))

                h2, self.h2_w, self.h2_b = deconv2d(
                    h1, [self.batch_size, s_h4, s_w4, self.gf_dim * 2], name='g_h2', with_w=True)
                h2 = tf.nn.relu(self.g_bn2(h2))

                h3, self.h3_w, self.h3_b = deconv2d(
                    h2, [self.batch_size, s_h2, s_w2, self.gf_dim * 1], name='g_h3', with_w=True)
                h3 = tf.nn.relu(self.g_bn3(h3))

                h4, self.h4_w, self.h4_b = deconv2d(
                    h3, [self.batch_size, s_h, s_w, self.c_dim], name='g_h4', with_w=True)

                return tf.nn.tanh(h4)
            else:
                s_h, s_w = self.output_height, self.output_width
                s_h2, s_h4 = int(s_h / 2), int(s_h / 4)
                s_w2, s_w4 = int(s_w / 2), int(s_w / 4)

                # yb = tf.expand_dims(tf.expand_dims(y, 1),2)
                yb = tf.reshape(y, [self.batch_size, 1, 1, self.y_dim])
                z = concat([z, y], 1)

                h0 = tf.nn.relu(
                    self.g_bn0(linear(z, self.gfc_dim, 'g_h0_lin')))
                h0 = concat([h0, y], 1)

                h1 = tf.nn.relu(self.g_bn1(
```

```
            linear(h0, self.gf_dim * 2 * s_h4 * s_w4, 'g_h1_lin')))
        h1 = tf.reshape(h1, [self.batch_size, s_h4, s_w4, self.gf_dim * 2])

        h1 = conv_cond_concat(h1, yb)

        h2 = tf.nn.relu(self.g_bn2(deconv2d(h1,
                                [self.batch_size, s_h2, s_w2, self.gf_dim * 2], name='g_h2')))
        h2 = conv_cond_concat(h2, yb)

        return tf.nn.sigmoid(
            deconv2d(h2, [self.batch_size, s_h, s_w, self.c_dim], name='g_h3'))

def sampler(self, z, y=None):
    with tf.variable_scope("generator") as scope:
        scope.reuse_variables()

        if not self.y_dim:
            s_h, s_w = self.output_height, self.output_width
            s_h2, s_w2 = conv_out_size_same(s_h, 2), conv_out_size_same(s_w, 2)
            s_h4, s_w4 = conv_out_size_same(s_h2, 2), conv_out_size_same(s_w2, 2)
            s_h8, s_w8 = conv_out_size_same(s_h4, 2), conv_out_size_same(s_w4, 2)
            s_h16, s_w16 = conv_out_size_same(s_h8, 2), conv_out_size_same(s_w8, 2)

            # project `z` and reshape
            h0 = tf.reshape(
                linear(z, self.gf_dim * 8 * s_h16 * s_w16, 'g_h0_lin'),
                [-1, s_h16, s_w16, self.gf_dim * 8])
            h0 = tf.nn.relu(self.g_bn0(h0, train=False))

            h1 = deconv2d(h0, [self.batch_size, s_h8, s_w8, self.gf_dim * 4], name='g_h1')
            h1 = tf.nn.relu(self.g_bn1(h1, train=False))

            h2 = deconv2d(h1, [self.batch_size, s_h4, s_w4, self.gf_dim * 2], name='g_h2')
            h2 = tf.nn.relu(self.g_bn2(h2, train=False))

            h3 = deconv2d(h2, [self.batch_size, s_h2, s_w2, self.gf_dim * 1], name='g_h3')
            h3 = tf.nn.relu(self.g_bn3(h3, train=False))

            h4 = deconv2d(h3, [self.batch_size, s_h, s_w, self.c_dim], name='g_h4')

            return tf.nn.tanh(h4)
        else:
            s_h, s_w = self.output_height, self.output_width
            s_h2, s_h4 = int(s_h / 2), int(s_h / 4)
            s_w2, s_w4 = int(s_w / 2), int(s_w / 4)

            # yb = tf.reshape(y, [-1, 1, 1, self.y_dim])
            yb = tf.reshape(y, [self.batch_size, 1, 1, self.y_dim])
            z = concat([z, y], 1)

            h0 = tf.nn.relu(self.g_bn0(linear(z, self.gfc_dim, 'g_h0_lin'), train=False))
            h0 = concat([h0, y], 1)

            h1 = tf.nn.relu(self.g_bn1(
                linear(h0, self.gf_dim * 2 * s_h4 * s_w4, 'g_h1_lin'), train=False))
            h1 = tf.reshape(h1, [self.batch_size, s_h4, s_w4, self.gf_dim * 2])
            h1 = conv_cond_concat(h1, yb)
```

```
            h2 = tf.nn.relu(self.g_bn2(
                deconv2d(h1, [self.batch_size, s_h2, s_w2, self.gf_dim * 2], name='g_h2'),
train=False))
            h2 = conv_cond_concat(h2, yb)

            return tf.nn.sigmoid(deconv2d(h2, [self.batch_size, s_h, s_w, self.c_dim],
name='g_h3'))

    def load_mnist(self):
        data_dir = os.path.join(self.data_dir, self.dataset_name)

        fd = open(os.path.join(data_dir, 'train-images-idx3-ubyte'))
        loaded = np.fromfile(file=fd, dtype=np.uint8)
        trX = loaded[16:].reshape((60000, 28, 28, 1)).astype(np.float)

        fd = open(os.path.join(data_dir, 'train-labels-idx1-ubyte'))
        loaded = np.fromfile(file=fd, dtype=np.uint8)
        trY = loaded[8:].reshape((60000)).astype(np.float)

        fd = open(os.path.join(data_dir, 't10k-images-idx3-ubyte'))
        loaded = np.fromfile(file=fd, dtype=np.uint8)
        teX = loaded[16:].reshape((10000, 28, 28, 1)).astype(np.float)

        fd = open(os.path.join(data_dir, 't10k-labels-idx1-ubyte'))
        loaded = np.fromfile(file=fd, dtype=np.uint8)
        teY = loaded[8:].reshape((10000)).astype(np.float)

        trY = np.asarray(trY)
        teY = np.asarray(teY)

        X = np.concatenate((trX, teX), axis=0)
        y = np.concatenate((trY, teY), axis=0).astype(np.int)

        seed = 547
        np.random.seed(seed)
        np.random.shuffle(X)
        np.random.seed(seed)
        np.random.shuffle(y)

        y_vec = np.zeros((len(y), self.y_dim), dtype=np.float)
        for i, label in enumerate(y):
            y_vec[i, y[i]] = 1.0

        return X / 255., y_vec

    @property
    def model_dir(self):
        return "{}_{}_{}_{}".format(
            self.dataset_name, self.batch_size,
            self.output_height, self.output_width)

    def save(self, checkpoint_dir, step):
        model_name = "DCGAN.model"
        checkpoint_dir = os.path.join(checkpoint_dir, self.model_dir)

        if not os.path.exists(checkpoint_dir):
```

```
            os.makedirs(checkpoint_dir)

        self.saver.save(self.sess,
                        os.path.join(checkpoint_dir, model_name),
                        global_step=step)

    def load(self, checkpoint_dir):
        import re
        print(" [*] Reading checkpoints...")
        checkpoint_dir = os.path.join(checkpoint_dir, self.model_dir)

        ckpt = tf.train.get_checkpoint_state(checkpoint_dir)
        if ckpt and ckpt.model_checkpoint_path:
            ckpt_name = os.path.basename(ckpt.model_checkpoint_path)
            self.saver.restore(self.sess, os.path.join(checkpoint_dir, ckpt_name))
            counter = int(next(re.finditer("(\d+)(?!.*\d)", ckpt_name)).group(0))
            print(" [*] Success to read {}".format(ckpt_name))
            return True, counter
        else:
            print(" [*] Failed to find a checkpoint")
            return False, 0
```

ops.py

```
import math
import numpy as np
import tensorflow as tf

from tensorflow.python.framework import ops

from utils import *

try:
    image_summary = tf.image_summary
    scalar_summary = tf.scalar_summary
    histogram_summary = tf.histogram_summary
    merge_summary = tf.merge_summary
    SummaryWriter = tf.train.SummaryWriter
except:
    image_summary = tf.summary.image
    scalar_summary = tf.summary.scalar
    histogram_summary = tf.summary.histogram
    merge_summary = tf.summary.merge
    SummaryWriter = tf.summary.FileWriter

if "concat_v2" in dir(tf):
    def concat(tensors, axis, *args, **kwargs):
        return tf.concat_v2(tensors, axis, *args, **kwargs)
else:
    def concat(tensors, axis, *args, **kwargs):
        return tf.concat(tensors, axis, *args, **kwargs)
```

```
class batch_norm(object):
    def __init__(self, epsilon=1e-5, momentum=0.9, name="batch_norm"):
        with tf.variable_scope(name):
            self.epsilon = epsilon
            self.momentum = momentum
            self.name = name

    def __call__(self, x, train=True):
        return tf.contrib.layers.batch_norm(x,
                                            decay=self.momentum,
                                            updates_collections=None,
                                            epsilon=self.epsilon,
                                            scale=True,
                                            is_training=train,
                                            scope=self.name)


def conv_cond_concat(x, y):
    """Concatenate conditioning vector on feature map axis."""
    x_shapes = x.get_shape()
    y_shapes = y.get_shape()
    return concat([
        x, y * tf.ones([x_shapes[0], x_shapes[1], x_shapes[2], y_shapes[3]])], 3)


def conv2d(input_, output_dim,
           k_h=5, k_w=5, d_h=2, d_w=2, stddev=0.02,
           name="conv2d"):
    with tf.variable_scope(name):
        w = tf.get_variable('w', [k_h, k_w, input_.get_shape()[-1], output_dim],
                            initializer=tf.truncated_normal_initializer(stddev=stddev))
        conv = tf.nn.conv2d(input_, w, strides=[1, d_h, d_w, 1], padding='SAME')

        biases = tf.get_variable('biases', [output_dim], initializer=tf.constant_initializer(0.0))
        conv = tf.reshape(tf.nn.bias_add(conv, biases), conv.get_shape())

        return conv


def deconv2d(input_, output_shape,
             k_h=5, k_w=5, d_h=2, d_w=2, stddev=0.02,
             name="deconv2d", with_w=False):
    with tf.variable_scope(name):
        # filter : [height, width, output_channels, in_channels]
        w = tf.get_variable('w', [k_h, k_w, output_shape[-1], input_.get_shape()[-1]],
                            initializer=tf.random_normal_initializer(stddev=stddev))

        try:
            deconv = tf.nn.conv2d_transpose(input_, w, output_shape=output_shape,
                                            strides=[1, d_h, d_w, 1])

        # Support for verisons of TensorFlow before 0.7.0
        except AttributeError:
            deconv = tf.nn.deconv2d(input_, w, output_shape=output_shape,
                                    strides=[1, d_h, d_w, 1])

        biases = tf.get_variable('biases', [output_shape[-1]], initializer=tf.constant_initializer(0.0))
```

```
        deconv = tf.reshape(tf.nn.bias_add(deconv, biases), deconv.get_shape())

        if with_w:
            return deconv, w, biases
        else:
            return deconv


def lrelu(x, leak=0.2, name="lrelu"):
    return tf.maximum(x, leak * x)


def linear(input_, output_size, scope=None, stddev=0.02, bias_start=0.0, with_w=False):
    shape = input_.get_shape().as_list()

    with tf.variable_scope(scope or "Linear"):
        try:
            matrix = tf.get_variable("Matrix", [shape[1], output_size], tf.float32,
                                     tf.random_normal_initializer(stddev=stddev))
        except ValueError as err:
            msg = "NOTE: Usually, this is due to an issue with the image dimensions.  Did you
correctly set '--crop' or '--input_height' or '--output_height'?"
            err.args = err.args + (msg,)
            raise
        bias = tf.get_variable("bias", [output_size],
                               initializer=tf.constant_initializer(bias_start))
        if with_w:
            return tf.matmul(input_, matrix) + bias, matrix, bias
        else:
            return tf.matmul(input_, matrix) + bias
```

utils.py

```
from __future__ import division
import math
import random
import pprint
import scipy.misc
import numpy as np
from time import gmtime, strftime
from six.moves import xrange

import tensorflow as tf
import tensorflow.contrib.slim as slim

pp = pprint.PrettyPrinter()

get_stddev = lambda x, k_h, k_w: 1 / math.sqrt(k_w * k_h * x.get_shape()[-1])


def show_all_variables():
    model_vars = tf.trainable_variables()
    slim.model_analyzer.analyze_vars(model_vars, print_info=True)
```

```
def get_image(image_path, input_height, input_width,
              resize_height=64, resize_width=64,
              crop=True, grayscale=False):
    image = imread(image_path, grayscale)
    return transform(image, input_height, input_width,
                     resize_height, resize_width, crop)


def save_images(images, size, image_path):
    return imsave(inverse_transform(images), size, image_path)


def imread(path, grayscale=False):
    if (grayscale):
        return scipy.misc.imread(path, flatten=True).astype(np.float)
    else:
        return scipy.misc.imread(path).astype(np.float)


def merge_images(images, size):
    return inverse_transform(images)


def merge(images, size):
    h, w = images.shape[1], images.shape[2]
    if (images.shape[3] in (3, 4)):
        c = images.shape[3]
        img = np.zeros((h * size[0], w * size[1], c))
        for idx, image in enumerate(images):
            i = idx % size[1]
            j = idx // size[1]
            img[j * h:j * h + h, i * w:i * w + w, :] = image
        return img
    elif images.shape[3] == 1:
        img = np.zeros((h * size[0], w * size[1]))
        for idx, image in enumerate(images):
            i = idx % size[1]
            j = idx // size[1]
            img[j * h:j * h + h, i * w:i * w + w] = image[:, :, 0]
        return img
    else:
        raise ValueError('in merge(images,size) images parameter '
                         'must have dimensions: HxW or HxWx3 or HxWx4')


def imsave(images, size, path):
    image = np.squeeze(merge(images, size))
    return scipy.misc.imsave(path, image)


def center_crop(x, crop_h, crop_w,
                resize_h=64, resize_w=64):
    if crop_w is None:
        crop_w = crop_h
    h, w = x.shape[:2]
```

```
    j = int(round((h - crop_h) / 2.))
    i = int(round((w - crop_w) / 2.))
    return scipy.misc.imresize(
        x[j:j + crop_h, i:i + crop_w], [resize_h, resize_w])


def transform(image, input_height, input_width,
              resize_height=64, resize_width=64, crop=True):
    if crop:
        cropped_image = center_crop(
            image, input_height, input_width,
            resize_height, resize_width)
    else:
        cropped_image = scipy.misc.imresize(image, [resize_height, resize_width])
    return np.array(cropped_image) / 127.5 - 1.


def inverse_transform(images):
    return (images + 1.) / 2.


def to_json(output_path, *layers):
    with open(output_path, "w") as layer_f:
        lines = ""
        for w, b, bn in layers:
            layer_idx = w.name.split('/')[0].split('h')[1]

            B = b.eval()

            if "lin/" in w.name:
                W = w.eval()
                depth = W.shape[1]
            else:
                W = np.rollaxis(w.eval(), 2, 0)
                depth = W.shape[0]

            biases = {"sy": 1, "sx": 1, "depth": depth, "w": ['%.2f' % elem for elem in list(B)]}
            if bn != None:
                gamma = bn.gamma.eval()
                beta = bn.beta.eval()

                gamma = {"sy": 1, "sx": 1, "depth": depth, "w": ['%.2f' % elem for elem in
list(gamma)]}
                beta = {"sy": 1, "sx": 1, "depth": depth, "w": ['%.2f' % elem for elem in list(beta)]}
            else:
                gamma = {"sy": 1, "sx": 1, "depth": 0, "w": []}
                beta = {"sy": 1, "sx": 1, "depth": 0, "w": []}

            if "lin/" in w.name:
                fs = []
                for w in W.T:
                    fs.append({"sy": 1, "sx": 1, "depth": W.shape[0], "w": ['%.2f' % elem for elem in
list(w)]})

                lines += """
          var layer_%s = {
            "layer_type": "fc",
```

```
          "sy": 1, "sx": 1,
          "out_sx": 1, "out_sy": 1,
          "stride": 1, "pad": 0,
          "out_depth": %s, "in_depth": %s,
          "biases": %s,
          "gamma": %s,
          "beta": %s,
          "filters": %s
        };""" % (layer_idx.split('_')[0], W.shape[1], W.shape[0], biases, gamma, beta, fs)
          else:
            fs = []
            for w_ in W:
              fs.append(
                {"sy": 5, "sx": 5, "depth": W.shape[3], "w": ['%.2f' % elem for elem in
list(w_.flatten())]})

            lines += """
        var layer_%s = {
          "layer_type": "deconv",
          "sy": 5, "sx": 5,
          "out_sx": %s, "out_sy": %s,
          "stride": 2, "pad": 1,
          "out_depth": %s, "in_depth": %s,
          "biases": %s,
          "gamma": %s,
          "beta": %s,
          "filters": %s
        };""" % (layer_idx, 2 ** (int(layer_idx) + 2), 2 ** (int(layer_idx) + 2),
                 W.shape[0], W.shape[3], biases, gamma, beta, fs)
      layer_f.write(" ".join(lines.replace("'", "").split()))


def make_gif(images, fname, duration=2, true_image=False):
  import moviepy.editor as mpy

  def make_frame(t):
    try:
      x = images[int(len(images) / duration * t)]
    except:
      x = images[-1]

    if true_image:
      return x.astype(np.uint8)
    else:
      return ((x + 1) / 2 * 255).astype(np.uint8)

  clip = mpy.VideoClip(make_frame, duration=duration)
  clip.write_gif(fname, fps=len(images) / duration)


def visualize(sess, dcgan, config, option):
  image_frame_dim = int(math.ceil(config.batch_size ** .5))
  if option == 0:
    z_sample = np.random.uniform(-0.5, 0.5, size=(config.batch_size, dcgan.z_dim))
    samples = sess.run(dcgan.sampler, feed_dict={dcgan.z: z_sample})
    save_images(samples, [image_frame_dim, image_frame_dim],
               './samples/test_%s.png' % strftime("%Y-%m-%d-%H-%M-%S", gmtime()))
```

```
  elif option == 1:
    values = np.arange(0, 1, 1. / config.batch_size)
    for idx in xrange(dcgan.z_dim):
      print(" [*] %d" % idx)
      z_sample = np.random.uniform(-1, 1, size=(config.batch_size, dcgan.z_dim))
      for kdx, z in enumerate(z_sample):
        z[idx] = values[kdx]

      if config.dataset == "mnist":
        y = np.random.choice(10, config.batch_size)
        y_one_hot = np.zeros((config.batch_size, 10))
        y_one_hot[np.arange(config.batch_size), y] = 1

        samples = sess.run(dcgan.sampler, feed_dict={dcgan.z: z_sample, dcgan.y:
y_one_hot})
      else:
        samples = sess.run(dcgan.sampler, feed_dict={dcgan.z: z_sample})

      save_images(samples, [image_frame_dim, image_frame_dim],
'./samples/test_arange_%s.png' % (idx))
  elif option == 2:
    values = np.arange(0, 1, 1. / config.batch_size)
    for idx in [random.randint(0, dcgan.z_dim - 1) for _ in xrange(dcgan.z_dim)]:
      print(" [*] %d" % idx)
      z = np.random.uniform(-0.2, 0.2, size=(dcgan.z_dim))
      z_sample = np.tile(z, (config.batch_size, 1))
      # z_sample = np.zeros([config.batch_size, dcgan.z_dim])
      for kdx, z in enumerate(z_sample):
        z[idx] = values[kdx]

      if config.dataset == "mnist":
        y = np.random.choice(10, config.batch_size)
        y_one_hot = np.zeros((config.batch_size, 10))
        y_one_hot[np.arange(config.batch_size), y] = 1

        samples = sess.run(dcgan.sampler, feed_dict={dcgan.z: z_sample, dcgan.y:
y_one_hot})
      else:
        samples = sess.run(dcgan.sampler, feed_dict={dcgan.z: z_sample})

      try:
        make_gif(samples, './samples/test_gif_%s.gif' % (idx))
      except:
        save_images(samples, [image_frame_dim, image_frame_dim],
              './samples/test_%s.png' % strftime("%Y-%m-%d-%H-%M-%S", gmtime()))
  elif option == 3:
    values = np.arange(0, 1, 1. / config.batch_size)
    for idx in xrange(dcgan.z_dim):
      print(" [*] %d" % idx)
      z_sample = np.zeros([config.batch_size, dcgan.z_dim])
      for kdx, z in enumerate(z_sample):
        z[idx] = values[kdx]

      samples = sess.run(dcgan.sampler, feed_dict={dcgan.z: z_sample})
      make_gif(samples, './samples/test_gif_%s.gif' % (idx))
  elif option == 4:
    image_set = []
```

```
    values = np.arange(0, 1, 1. / config.batch_size)

    for idx in xrange(dcgan.z_dim):
      print(" [*] %d" % idx)
      z_sample = np.zeros([config.batch_size, dcgan.z_dim])
      for kdx, z in enumerate(z_sample): z[idx] = values[kdx]

      image_set.append(sess.run(dcgan.sampler, feed_dict={dcgan.z: z_sample}))
      make_gif(image_set[-1], './samples/test_gif_%s.gif' % (idx))

    new_image_set = [merge(np.array([images[idx] for images in image_set]), [10, 10]) \
        for idx in range(64) + range(63, -1, -1)]
    make_gif(new_image_set, './samples/test_gif_merged.gif', duration=8)


def image_manifold_size(num_images):
  manifold_h = int(np.floor(np.sqrt(num_images)))
  manifold_w = int(np.ceil(np.sqrt(num_images)))
  assert manifold_h * manifold_w == num_images
  return manifold_h, manifold_w
```

PixelCNN (disponível em *https://github.com/rafesc/gated-pixel-cnn*, baseado no código de *https://github.com/jakebelew*)

conv_ops_test.py

```
import numpy as np
import tensorflow as tf

from ops import conv

def run(op):
 with tf.Session() as sess:
  tf.initialize_all_variables().run()
  outputs = sess.run(op)
  print("outputs shape: {}:".format(outputs.shape))
  print(np.transpose(outputs, axes=(0,3,1,2))[0,:,:,:])
  return outputs

def create_inputs(image_shape, num_inputs):
 num_batch = 1
 image_h, image_w = image_shape
 inputs_shape = [num_batch, image_h, image_w, num_inputs]
 return np.arange(1, np.prod(inputs_shape) + 1, dtype=np.float32).reshape(inputs_shape)

def create_ones_inputs(image_shape, num_inputs):
 num_batch = 1
 image_h, image_w = image_shape
 inputs_shape = [num_batch, image_h, image_w, num_inputs]
 return tf.ones(inputs_shape, dtype=tf.float32)
```

```
def expected_from_list(e_list):
  e_list = np.array(e_list)
  e_list = np.transpose(e_list, axes=(1,2,0))
  e_list = np.expand_dims(e_list, axis=0)
  return e_list

def assert_equals(array, expected):
  if not np.array_equal(array, expected):
    raise Exception("Array is not what is expected")

def matrix_to_string(array):
  '''Creates a string that is in the proper format to create a numpy array from.'''
  np.set_printoptions(precision=0)
  string = '['
  h, w = array.shape
  for i in range(h):
    if not (i == 0):
      string += ' '
    string += '['
    for j in range(w):
      string += ' {0:.0f}'.format(array[i,j]) + '.'
      if (j < w-1):
        string += ', '
    string += ']'
    if (i < h-1):
      string += ',\n'
  return string + ']'

def test_first_layer():
  print("==========================================")
  print("First Layer: 7 x 7 conv mask A (1 layer)")
  print("Grayscale [1 in, 16 out]")
  input_shape = [9,9]
  kernel_shape = [7,7]
  mask_type = "A"
  num_inputs, num_outputs, data_num_channels = 1, 2, 1

  expected_0 = np.array(
  [[ 0.,  1.,  2.,  3.,  3.,  3.,  3.,  3.,  3.],
   [ 4.,  6.,  8.,  10.,  10.,  10.,  9.,  8.,  7.],
   [ 8.,  11.,  14.,  17.,  17.,  17.,  15.,  13.,  11.],
   [ 12.,  16.,  20.,  24.,  24.,  24.,  21.,  18.,  15.],
   [ 12.,  16.,  20.,  24.,  24.,  24.,  21.,  18.,  15.],
   [ 12.,  16.,  20.,  24.,  24.,  24.,  21.,  18.,  15.],
   [ 12.,  16.,  20.,  24.,  24.,  24.,  21.,  18.,  15.],
   [ 12.,  16.,  20.,  24.,  24.,  24.,  21.,  18.,  15.],
   [ 12.,  16.,  20.,  24.,  24.,  24.,  21.,  18.,  15.]])
  expected = expected_from_list([expected_0, expected_0])

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, mask_type, data_num_channels,
weights_initializer=tf.ones_initializer, scope="7x7_conv_mask_A_Grayscale"))
  #print(matrix_to_string(outputs[0,:,:,0]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)
```

```
  print("-------------------------------------------")
  print("Color [3 in, 48 out]")
  num_inputs, num_outputs, data_num_channels = 3, 6, 3

  expected_0 = np.array(
  [[ 0.,  3.,  6.,  9.,  9.,  9.,  9.,  9.,  9.],
   [ 12.,  18.,  24.,  30.,  30.,  30.,  27.,  24.,  21.],
   [ 24.,  33.,  42.,  51.,  51.,  51.,  45.,  39.,  33.],
   [ 36.,  48.,  60.,  72.,  72.,  72.,  63.,  54.,  45.],
   [ 36.,  48.,  60.,  72.,  72.,  72.,  63.,  54.,  45.],
   [ 36.,  48.,  60.,  72.,  72.,  72.,  63.,  54.,  45.],
   [ 36.,  48.,  60.,  72.,  72.,  72.,  63.,  54.,  45.],
   [ 36.,  48.,  60.,  72.,  72.,  72.,  63.,  54.,  45.],
   [ 36.,  48.,  60.,  72.,  72.,  72.,  63.,  54.,  45.]])

  expected_1 = np.array(
  [[ 1.,  4.,  7.,  10.,  10.,  10.,  10.,  10.,  10.],
   [ 13.,  19.,  25.,  31.,  31.,  31.,  28.,  25.,  22.],
   [ 25.,  34.,  43.,  52.,  52.,  52.,  46.,  40.,  34.],
   [ 37.,  49.,  61.,  73.,  73.,  73.,  64.,  55.,  46.],
   [ 37.,  49.,  61.,  73.,  73.,  73.,  64.,  55.,  46.],
   [ 37.,  49.,  61.,  73.,  73.,  73.,  64.,  55.,  46.],
   [ 37.,  49.,  61.,  73.,  73.,  73.,  64.,  55.,  46.],
   [ 37.,  49.,  61.,  73.,  73.,  73.,  64.,  55.,  46.],
   [ 37.,  49.,  61.,  73.,  73.,  73.,  64.,  55.,  46.]])

  expected_2 = np.array(
  [[ 2.,  5.,  8.,  11.,  11.,  11.,  11.,  11.,  11.],
   [ 14.,  20.,  26.,  32.,  32.,  32.,  29.,  26.,  23.],
   [ 26.,  35.,  44.,  53.,  53.,  53.,  47.,  41.,  35.],
   [ 38.,  50.,  62.,  74.,  74.,  74.,  65.,  56.,  47.],
   [ 38.,  50.,  62.,  74.,  74.,  74.,  65.,  56.,  47.],
   [ 38.,  50.,  62.,  74.,  74.,  74.,  65.,  56.,  47.],
   [ 38.,  50.,  62.,  74.,  74.,  74.,  65.,  56.,  47.],
   [ 38.,  50.,  62.,  74.,  74.,  74.,  65.,  56.,  47.],
   [ 38.,  50.,  62.,  74.,  74.,  74.,  65.,  56.,  47.]])

  expected = expected_from_list([expected_0, expected_1, expected_2] * 2)

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, mask_type, data_num_channels,
weights_initializer=tf.ones_initializer, scope="7x7_conv_mask_A_Color"))
  #print(matrix_to_string(outputs[0,:,:,2]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)

def test_gated_conv_3x3_layer():
  print("=========================================")
  print("Gated Conv Layers: 3 x 3 gated conv mask B (Multiple layers)")
  print("Grayscale [16 in, 16 out]")
  input_shape = [5,5]
  kernel_shape = [3,3]
  mask_type = "B"
  num_inputs, num_outputs, data_num_channels = 2, 2, 1
```

```
  expected_0 = np.array(
  [[ 2.,  4.,  4.,  4.,  4.],
   [ 6.,  10.,  10.,  10.,  8.],
   [ 6.,  10.,  10.,  10.,  8.],
   [ 6.,  10.,  10.,  10.,  8.],
   [ 6.,  10.,  10.,  10.,  8.]])

  expected = expected_from_list([expected_0, expected_0])

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, mask_type, data_num_channels,
weights_initializer=tf.ones_initializer, scope="3x3_gated_conv_mask_B_Grayscale"))
  #print(matrix_to_string(outputs[0,:,:,0]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)
  print("------------------------------------------")
  print("Color [3 in, 48 out]")
  num_inputs, num_outputs, data_num_channels = 3, 6, 3

  expected_0 = np.array(
  [[ 1.,  4.,  4.,  4.,  4.],
   [ 7.,  13.,  13.,  13.,  10.],
   [ 7.,  13.,  13.,  13.,  10.],
   [ 7.,  13.,  13.,  13.,  10.],
   [ 7.,  13.,  13.,  13.,  10.]])

  expected_1 = np.array(
  [[ 2.,  5.,  5.,  5.,  5.],
   [ 8.,  14.,  14.,  14.,  11.],
   [ 8.,  14.,  14.,  14.,  11.],
   [ 8.,  14.,  14.,  14.,  11.],
   [ 8.,  14.,  14.,  14.,  11.]])

  expected_2 = np.array(
  [[ 3.,  6.,  6.,  6.,  6.],
   [ 9.,  15.,  15.,  15.,  12.],
   [ 9.,  15.,  15.,  15.,  12.],
   [ 9.,  15.,  15.,  15.,  12.],
   [ 9.,  15.,  15.,  15.,  12.]])

  expected = expected_from_list([expected_0, expected_1, expected_2] * 2)

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, mask_type, data_num_channels,
weights_initializer=tf.ones_initializer, scope="3x3_gated_conv_mask_B_Color"))
  #print(matrix_to_string(outputs[0,:,:,0]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)

def test_gated_horiz_mask_layer():
  print("==========================================")
  print("Gated Conv Layers: 1 x 3 gated horiz conv mask B (Multiple layers)")
  print("Grayscale [16 in, 16 out]")
```

```
  input_shape = [5,5]
  kernel_shape = [1,3]
  mask_type = "B"
  num_inputs, num_outputs, data_num_channels = 2, 2, 1

  expected_0 = np.array(
  [[ 2.,  4.,  4.,  4.,  4.],
   [ 2.,  4.,  4.,  4.,  4.],
   [ 2.,  4.,  4.,  4.,  4.],
   [ 2.,  4.,  4.,  4.,  4.],
   [ 2.,  4.,  4.,  4.,  4.]])

  expected = expected_from_list([expected_0, expected_0])

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, mask_type, data_num_channels,
weights_initializer=tf.ones_initializer, scope="1x3_gated_horiz_conv_mask_B_Grayscale"))
  #print(matrix_to_string(outputs[0,:,:,0]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)
  print("------------------------------------------")
  print("Color [3 in, 48 out]")
  num_inputs, num_outputs, data_num_channels = 3, 6, 3

  expected_0 = np.array(
  [[ 1.,  4.,  4.,  4.,  4.],
   [ 1.,  4.,  4.,  4.,  4.],
   [ 1.,  4.,  4.,  4.,  4.],
   [ 1.,  4.,  4.,  4.,  4.],
   [ 1.,  4.,  4.,  4.,  4.]])

  expected_1 = np.array(
  [[ 2.,  5.,  5.,  5.,  5.],
   [ 2.,  5.,  5.,  5.,  5.],
   [ 2.,  5.,  5.,  5.,  5.],
   [ 2.,  5.,  5.,  5.,  5.],
   [ 2.,  5.,  5.,  5.,  5.]])

  expected_2 = np.array(
  [[ 3.,  6.,  6.,  6.,  6.],
   [ 3.,  6.,  6.,  6.,  6.],
   [ 3.,  6.,  6.,  6.,  6.],
   [ 3.,  6.,  6.,  6.,  6.],
   [ 3.,  6.,  6.,  6.,  6.]])

  expected = expected_from_list([expected_0, expected_1, expected_2] * 2)

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, mask_type, data_num_channels,
weights_initializer=tf.ones_initializer, scope="1x3_gated_horiz_conv_mask_B_Color"))

  #print(matrix_to_string(outputs[0,:,:,2]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop
```

```
  assert_equals(outputs, expected)

def test_last_layers():
  print("===========================================")
  print("Last Layers: 1 x 1 conv mask B (2 layers)")
  print("Grayscale - (No masking required) [32 in, 32 out]")
  input_shape = [5,5]
  kernel_shape = [1,1]
  mask_type = "B"
  num_inputs, num_outputs, data_num_channels = 2, 2, 1

  expected_0 = np.array(
  [[ 2.,  2.,  2.,  2.,  2.],
   [ 2.,  2.,  2.,  2.,  2.],
   [ 2.,  2.,  2.,  2.,  2.],
   [ 2.,  2.,  2.,  2.,  2.],
   [ 2.,  2.,  2.,  2.,  2.]])

  expected = expected_from_list([expected_0, expected_0])

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, mask_type, data_num_channels,
weights_initializer=tf.ones_initializer, scope="1x1_conv_mask_B_Grayscale"))
  #print(matrix_to_string(outputs[0,:,:,0]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)
  print("-----------------------------------------")
  print("Color [3 in, 48 out]")
  num_inputs, num_outputs, data_num_channels = 3, 6, 3

  expected_0 = np.array(
  [[ 1.,  1.,  1.,  1.,  1.],
   [ 1.,  1.,  1.,  1.,  1.],
   [ 1.,  1.,  1.,  1.,  1.],
   [ 1.,  1.,  1.,  1.,  1.],
   [ 1.,  1.,  1.,  1.,  1.]])

  expected_1 = np.array(
  [[ 2.,  2.,  2.,  2.,  2.],
   [ 2.,  2.,  2.,  2.,  2.],
   [ 2.,  2.,  2.,  2.,  2.],
   [ 2.,  2.,  2.,  2.,  2.],
   [ 2.,  2.,  2.,  2.,  2.]])

  expected_2 = np.array(
  [[ 3.,  3.,  3.,  3.,  3.],
   [ 3.,  3.,  3.,  3.,  3.],
   [ 3.,  3.,  3.,  3.,  3.],
   [ 3.,  3.,  3.,  3.,  3.],
   [ 3.,  3.,  3.,  3.,  3.]])

  expected = expected_from_list([expected_0, expected_1, expected_2] * 2)

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
```

```
    num_outputs, kernel_shape, mask_type, data_num_channels,
weights_initializer=tf.ones_initializer, scope="1x1_conv_mask_B_Color"))
  #print(matrix_to_string(outputs[0,:,:,2]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)

def repeat_matrix(pattern, num_rows, num_columns):
  pattern = np.repeat(pattern[:, :, np.newaxis], num_rows, axis=2)
  return np.repeat(pattern[:, :, :, np.newaxis], num_columns, axis=3)

def test_masked_vert_mask_layer():
  print("===========================================")
  print("Gated Conv Layers: 1 x 3 vertical conv mask (Multiple layers)")
  print("Grayscale [16 in, 16 out]")
  input_shape = [5,5]
  kernel_shape = [3,3]
  num_inputs, num_outputs, data_num_channels = 2, 2, 1

  expected = np.array(
  [[ 0.,  0.,  0.,  0.,  0.],
   [ 4.,  6.,  6.,  6.,  4.],
   [ 4.,  6.,  6.,  6.,  4.],
   [ 4.,  6.,  6.,  6.,  4.],
   [ 4.,  6.,  6.,  6.,  4.]])

  expected = expected_from_list([expected] * num_outputs)

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, 'V', data_num_channels, weights_initializer=tf.ones_initializer,
scope="1x3_vert_conv_mask_Grayscale"))
  #print(matrix_to_string(outputs[0,:,:,0]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)
  print("-------------------------------------------")
  print("Color [3 in, 48 out]")
  num_inputs, num_outputs, data_num_channels = 3, 6, 3

  expected = np.array(
  [[ 0.,  0.,  0.,  0.,  0.],
   [ 6.,  9.,  9.,  9.,  6.],
   [ 6.,  9.,  9.,  9.,  6.],
   [ 6.,  9.,  9.,  9.,  6.],
   [ 6.,  9.,  9.,  9.,  6.]])

  expected = expected_from_list([expected] * num_outputs)

  outputs = run(conv(
    create_ones_inputs(input_shape, num_inputs),
    num_outputs, kernel_shape, 'V', data_num_channels, weights_initializer=tf.ones_initializer,
scope="1x3_vert_conv_mask_Color"))
  #print(matrix_to_string(outputs[0,:,:,2]))
  # invalid to crop_mask for A or B, only horiz or vertical mask can crop

  assert_equals(outputs, expected)
```

```
np.set_printoptions(suppress=True)

test_first_layer()
test_gated_conv_3x3_layer()
test_gated_horiz_mask_layer()
test_last_layers()
test_masked_vert_mask_layer()

print("\n***** All tests passed *****")
```

main.py

```
import logging
logging.basicConfig(format="[%(asctime)s] %(message)s", datefmt="%m-%d %H:%M:%S")
import os
import time

import numpy as np
import tensorflow as tf

import core.data.cifar_data as cifar
import core.data.mnist_data as mnist
from network import Network
from statistic import Statistic
import utils as util

flags = tf.app.flags

# network
flags.DEFINE_integer("batch_size", 100, "size of a batch")
flags.DEFINE_integer("gated_conv_num_layers", 7, "the number of gated conv layers")
flags.DEFINE_integer("gated_conv_num_feature_maps", 16, "the number of input / output
feature maps in gated conv layers")
flags.DEFINE_integer("output_conv_num_feature_maps", 32, "the number of output feature
maps in output conv layers")
flags.DEFINE_integer("q_levels", 4, "the number of quantization levels in the output")

# training
flags.DEFINE_float("max_epoch", 100000, "maximum # of epochs")
flags.DEFINE_float("learning_rate", 1e-3, "learning rate")
flags.DEFINE_float("grad_clip", 1, "value of gradient to be used for clipping")

# data
flags.DEFINE_string("data", "mnist", "name of dataset [mnist, color-mnist, cifar]")
flags.DEFINE_string("runtime_base_dir", "./", "path of base directory for checkpoints,
data_dir, logs and sample_dir")
flags.DEFINE_string("data_dir", "data", "name of data directory")
flags.DEFINE_string("sample_dir", "samples", "name of sample directory")

# generation
flags.DEFINE_string("occlude_start_row", 18, "image row to start occlusion")
```

```
flags.DEFINE_string("num_generated_images", 9, "number of images to generate")

# Debug
flags.DEFINE_boolean("is_train", True, "training or testing")
flags.DEFINE_string("log_level", "INFO", "log level [DEBUG, INFO, WARNING, ERROR,
CRITICAL]")
flags.DEFINE_integer("random_seed", 123, "random seed for python")

conf = flags.FLAGS

# logging
logger = logging.getLogger()
logger.setLevel(conf.log_level)

# random seed
tf.set_random_seed(conf.random_seed)
np.random.seed(conf.random_seed)


def validate_parameters(conf):
  if conf.data not in ["mnist", "color-mnist",  "cifar"]:
    raise ValueError("Configuration parameter 'data' is '{}'. Must be one of [mnist, color-mnist,
cifar]"
                     .format(conf.data))


def preprocess(q_levels):

  def preprocess_fcn(images, labels):
    # Create the target pixels from the image. Quantize the scalar pixel values into q_level
indices.
    target_pixels = np.clip(((images * q_levels).astype('int64')), 0, q_levels - 1) # [N,H,W,C]
    return (images, target_pixels)

  return preprocess_fcn


def get_dataset(data_dir, q_levels):
  if conf.data == "mnist":
    dataset = mnist.get_dataset(data_dir, preprocess(q_levels), reshape=False)
  elif conf.data == "color-mnist":
    dataset = mnist.get_colorized_dataset(data_dir, preprocess(q_levels), reshape=False)
  elif conf.data == "cifar":
    dataset = cifar.get_dataset(data_dir, preprocess(q_levels))

  return dataset


def generate_from_occluded(network, images):
  occlude_start_row = conf.occlude_start_row
  num_generated_images = conf.num_generated_images

  samples = network.generate_from_occluded(images, num_generated_images,
occlude_start_row)

  occluded = np.copy(images[0:num_generated_images,:,:,:])
  # render white line in occlusion start row
```

```
  occluded[:,occlude_start_row,:,:] = 255
  return samples, occluded


def train(dataset, network, stat, sample_dir):
  initial_step = stat.get_t()
  logger.info("Training starts on epoch {}".format(initial_step))

  train_step_per_epoch = dataset.train.num_examples / conf.batch_size
  test_step_per_epoch = dataset.test.num_examples / conf.batch_size

  for epoch in range(initial_step, conf.max_epoch):
    start_time = time.time()

    # 1. train
    total_train_costs = []
    for _ in xrange(train_step_per_epoch):
      images = dataset.train.next_batch(conf.batch_size)
      cost = network.test(images, with_update=True)
      total_train_costs.append(cost)

    # 2. test
    total_test_costs = []
    for _ in xrange(test_step_per_epoch):
      images = dataset.test.next_batch(conf.batch_size)
      cost = network.test(images, with_update=False)
      total_test_costs.append(cost)

    avg_train_cost, avg_test_cost = np.mean(total_train_costs), np.mean(total_test_costs)
    stat.on_step(avg_train_cost, avg_test_cost)

    # 3. generate samples
    images, _ = dataset.test.next_batch(conf.batch_size)
    samples, occluded = generate_from_occluded(network, images)
    util.save_images(np.concatenate((occluded, samples), axis=2),
            dataset.height, dataset.width * 2, conf.num_generated_images, 1,
            directory=sample_dir, prefix="epoch_%s" % epoch)

    logger.info("Epoch {}: {:.2f} seconds, avg train cost: {:.3f}, avg test cost: {:.3f}"
            .format(epoch,(time.time() - start_time), avg_train_cost, avg_test_cost))


def generate(network, height, width, sample_dir):
      logger.info("Image generation starts")
      samples = network.generate()
      util.save_images(samples, height, width, 10, 10, directory=sample_dir)


def main(_):
  model_dir = util.get_model_dir(conf,
      ['data_dir', 'sample_dir', 'max_epoch', 'test_step', 'save_step',
       'is_train', 'random_seed', 'log_level', 'display', 'runtime_base_dir',
       'occlude_start_row', 'num_generated_images'])
  util.preprocess_conf(conf)
  validate_parameters(conf)

  data = 'mnist' if conf.data == 'color-mnist' else conf.data
```

```
  DATA_DIR = os.path.join(conf.runtime_base_dir, conf.data_dir, data)
  SAMPLE_DIR = os.path.join(conf.runtime_base_dir, conf.sample_dir, conf.data, model_dir)

  util.check_and_create_dir(DATA_DIR)
  util.check_and_create_dir(SAMPLE_DIR)

  dataset = get_dataset(DATA_DIR, conf.q_levels)

  with tf.Session() as sess:
    network = Network(sess, conf, dataset.height, dataset.width, dataset.channels)

    stat = Statistic(sess, conf.data, conf.runtime_base_dir, model_dir, tf.trainable_variables())
    stat.load_model()

    if conf.is_train:
      train(dataset, network, stat, SAMPLE_DIR)
    else:
      generate(network, dataset.height, dataset.width, SAMPLE_DIR)


if __name__ == "__main__":
  tf.app.run()
```

network.py

```
from logging import getLogger

import tensorflow as tf

from ops import *
from utils import *

logger = getLogger(__name__)

class Network:

  def __init__(self, sess, conf, height, width, num_channels):
    logger.info("Building gated_pixel_cnn starts")

    self.sess = sess
    self.data = conf.data
    self.height, self.width, self.channel = height, width, num_channels
    self.pixel_depth = 256
    self.q_levels = q_levels = conf.q_levels

    self.inputs = tf.placeholder(tf.float32, [None, height, width, num_channels]) # [N,H,W,C]
    self.target_pixels = tf.placeholder(tf.int64, [None, height, width, num_channels]) #
[N,H,W,C] (the index of a one-hot representation of D)

    # input conv layer
    logger.info("Building CONV_IN")
    net = conv(self.inputs, conf.gated_conv_num_feature_maps, [7, 7], "A", num_channels,
scope="CONV_IN")
```

```
    # main gated layers
    for idx in xrange(conf.gated_conv_num_layers):
      scope = 'GATED_CONV%d' % idx
      net = gated_conv(net, [3, 3], num_channels, scope=scope)
      logger.info("Building %s" % scope)

    # output conv layers
    net = tf.nn.relu(conv(net, conf.output_conv_num_feature_maps, [1, 1], "B", num_channels,
scope='CONV_OUT0'))
    logger.info("Building CONV_OUT0")
    self.logits = tf.nn.relu(conv(net, q_levels * num_channels, [1, 1], "B", num_channels,
scope='CONV_OUT1')) # shape [N,H,W,DC]
    logger.info("Building CONV_OUT1")

    if (num_channels > 1):
      self.logits = tf.reshape(self.logits, [-1, height, width, q_levels, num_channels]) # shape
[N,H,W,DC] -> [N,H,W,D,C]
      self.logits = tf.transpose(self.logits, perm=[0, 1, 2, 4, 3]) # shape [N,H,W,D,C] ->
[N,H,W,C,D]

    flattened_logits = tf.reshape(self.logits, [-1, q_levels]) # [N,H,W,C,D] -> [NHWC,D]
    target_pixels_loss = tf.reshape(self.target_pixels, [-1]) # [N,H,W,C] -> [NHWC]

    logger.info("Building loss and optims")
    self.loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
      flattened_logits, target_pixels_loss))

    flattened_output = tf.nn.softmax(flattened_logits) #shape [NHWC,D], values [probability
distribution]
    self.output = tf.reshape(flattened_output, [-1, height, width, num_channels, q_levels])
#shape [N,H,W,C,D], values [probability distribution]

    optimizer = tf.train.RMSPropOptimizer(conf.learning_rate)
    grads_and_vars = optimizer.compute_gradients(self.loss)

    new_grads_and_vars = \
       [(tf.clip_by_value(gv[0], -conf.grad_clip, conf.grad_clip), gv[1]) for gv in grads_and_vars]
    self.optim = optimizer.apply_gradients(new_grads_and_vars)

    show_all_variables()

    logger.info("Building gated_pixel_cnn finished")

  def predict(self, images):
    '''
    images # shape [N,H,W,C]
    returns predicted image # shape [N,H,W,C]
    '''
    # self.output shape [NHWC,D]
    pixel_value_probabilities = self.sess.run(self.output, {self.inputs: images}) # shape
[N,H,W,C,D], values [probability distribution]

    # argmax or random draw # [NHWC,1]  quantized index - convert back to pixel value
    pixel_value_indices = np.argmax(pixel_value_probabilities, 4) # shape [N,H,W,C], values
[index of most likely pixel value]
    pixel_values = np.multiply(pixel_value_indices, ((self.pixel_depth - 1) / (self.q_levels - 1)))
```

```
#shape [N,H,W,C]

    return pixel_values

  def test(self, images, with_update=False):
    if with_update:
      _, cost = self.sess.run([self.optim, self.loss],
                              { self.inputs: images[0], self.target_pixels: images[1] })
    else:
      cost = self.sess.run(self.loss, { self.inputs: images[0], self.target_pixels: images[1] })
    return cost

  def generate_from_occluded(self, images, num_generated_images, occlude_start_row):
    samples = np.copy(images[0:num_generated_images,:,:,:])
    samples[:,occlude_start_row:,:,:] = 0.

    for i in xrange(occlude_start_row,self.height):
      for j in xrange(self.width):
        for k in xrange(self.channel):
          next_sample = self.predict(samples) / (self.pixel_depth - 1.) # argmax or random draw
here
          samples[:, i, j, k] = next_sample[:, i, j, k]

    return samples

  def generate(self, images):
    samples = images[0:9,:,:,:]
    occlude_start_row = 18
    samples[:,occlude_start_row:,:,:] = 0.

    for i in xrange(occlude_start_row,self.height):
      for j in xrange(self.width):
        for k in xrange(self.channel):
          next_sample = self.predict(samples) / (self.pixel_depth - 1.) # argmax or random draw
here
          samples[:, i, j, k] = next_sample[:, i, j, k]

    return samples
```

ops.py

```
import logging
logging.basicConfig(format="[%(asctime)s] %(message)s", datefmt="%m-%d %H:%M:%S")

import numpy as np
import tensorflow as tf

WEIGHT_INITIALIZER = tf.contrib.layers.xavier_initializer()

logger = logging.getLogger(__name__)

def get_shape(layer):
  return layer.get_shape().as_list()
```

```
def conv(
    inputs,
    num_outputs,
    kernel_shape, # [kernel_height, kernel_width]
    mask_type, # None, 'A', 'B' or 'V'
    data_num_channels,
    strides=[1, 1], # [column_wise_stride, row_wise_stride]
    padding="SAME",
    activation_fn=None,
    weights_initializer=WEIGHT_INITIALIZER,
    weights_regularizer=None,
    biases_initializer=tf.zeros_initializer,
    biases_regularizer=None,
    scope="conv2d"):
  with tf.variable_scope(scope):
    mask_type = mask_type.lower()
    if mask_type == 'v' and kernel_shape == [1,1]:
      # No mask required for Vertical 1x1 convolution
      mask_type = None
    num_inputs = get_shape(inputs)[-1]

    kernel_h, kernel_w = kernel_shape
    stride_h, stride_w = strides

    assert kernel_h % 2 == 1 and kernel_w % 2 == 1, \
      "kernel height and width should be an odd number"

    weights_shape = [kernel_h, kernel_w, num_inputs, num_outputs]
    weights = tf.get_variable("weights", weights_shape,
      tf.float32, weights_initializer, weights_regularizer)

    if mask_type is not None:
      mask = _create_mask(num_inputs, num_outputs, kernel_shape, data_num_channels,
mask_type)
      weights *= tf.constant(mask, dtype=tf.float32)
      tf.add_to_collection('conv2d_weights_%s' % mask_type, weights)

    outputs = tf.nn.conv2d(inputs,
        weights, [1, stride_h, stride_w, 1], padding=padding, name='outputs')
    tf.add_to_collection('conv2d_outputs', outputs)

    if biases_initializer != None:
      biases = tf.get_variable("biases", [num_outputs,],
          tf.float32, biases_initializer, biases_regularizer)
      outputs = tf.nn.bias_add(outputs, biases, name='outputs_plus_b')

    if activation_fn:
      outputs = activation_fn(outputs, name='outputs_with_fn')

    logger.debug('[conv2d_%s] %s : %s %s -> %s %s' \
        % (mask_type, scope, inputs.name, inputs.get_shape(), outputs.name,
outputs.get_shape()))

    return outputs

# for this type layer: num_outputs = num_inputs = number of channels in input layer
```

```
def gated_conv(inputs, kernel_shape, data_num_channels, scope="gated_conv"):
  with tf.variable_scope(scope):
    # Horiz inputs/outputs on left in case num_inputs not multiple of 6, because Horiz is RGB
gated and Vert is not.
    # inputs shape [N,H,W,C]
    horiz_inputs, vert_inputs  = tf.split(3, 2, inputs)
    p = get_shape(horiz_inputs)[-1]
    p2 = 2 * p

    # vertical n x n conv
    # p in channels, 2p out channels, vertical mask, same padding, stride 1
    vert_nxn = conv(vert_inputs, p2, kernel_shape, 'V', data_num_channels,
scope="vertical_nxn")

    # vertical blue diamond
    # 2p in channels, p out channels, vertical mask
    vert_gated_out = _gated_activation_unit(vert_nxn, kernel_shape, 'V', data_num_channels,
scope="vertical_gated_activation_unit")

    # vertical 1 x 1 conv
    # 2p in channels, 2p out channels, no mask?, same padding, stride 1
    vert_1x1 = conv(vert_nxn, p2, [1, 1], 'V', data_num_channels, scope="vertical_1x1")

    # horizontal 1 x n conv
    # p in channels, 2p out channels, horizontal mask B, same padding, stride 1
    horiz_1xn = conv(horiz_inputs, p2, kernel_shape, 'B', data_num_channels,
scope="horizontal_1xn")
    horiz_gated_in = vert_1x1 + horiz_1xn

    # horizontal blue diamond
    # 2p in channels, p out channels, horizontal mask B
    horiz_gated_out = _gated_activation_unit(horiz_gated_in, kernel_shape, 'B',
data_num_channels, scope="horizontal_gated_activation_unit")

    # horizontal 1 x 1 conv
    # p in channels, p out channels, mask B, same padding, stride 1
    horiz_1x1 = conv(horiz_gated_out, p, kernel_shape, 'B', data_num_channels,
scope="horizontal_1x1")

    horiz_outputs = horiz_1x1 + horiz_inputs

    return tf.concat(3, [horiz_outputs, vert_gated_out])

def _create_mask(
    num_inputs,
    num_outputs,
    kernel_shape,
    data_num_channels,
    mask_type, # 'A', 'B' or 'V'
    ):
    '''
    Produces a causal mask of the given type and shape
    '''
    mask_type = mask_type.lower()
    kernel_h, kernel_w = kernel_shape

    center_h = kernel_h // 2
```

```
    center_w = kernel_w // 2

    mask = np.ones(
      (kernel_h, kernel_w, num_inputs, num_outputs), dtype=np.float32) # shape [KERNEL_H,
KERNEL_W, NUM_INPUTS, NUM_OUTPUTS]

    if mask_type == 'v':
      mask[center_h:, :, :, :] = 0.
    else:
      mask[center_h, center_w+1:, :, :] = 0.
      mask[center_h+1:, :, :, :] = 0.

      if mask_type == 'b':
        mask_pixel = lambda i,j: i > j
      else:
        mask_pixel = lambda i,j: i >= j

      for i in range(num_inputs):
        for j in range(num_outputs):
          if mask_pixel(i % data_num_channels, j % data_num_channels):
            mask[center_h, center_w, i, j] = 0.

    return mask

# implements equation (2) of the paper
# returns 1/2 number of channels as input
def _gated_activation_unit(inputs, kernel_shape, mask_type, data_num_channels,
scope="gated_activation_unit"):
  with tf.variable_scope(scope):
    p2 =  get_shape(inputs)[-1]

    # blue diamond
    # 2p in channels, 2p out channels, mask, same padding, stride 1
    # split 2p out channels into p going to tanh and p going to sigmoid
    bd_out = conv(inputs, p2, kernel_shape, mask_type, data_num_channels,
scope="blue_diamond") #[N,H,W,C[,D]]
    bd_out_0, bd_out_1 = tf.split(3, 2, bd_out)
    tanh_out = tf.tanh(bd_out_0)
    sigmoid_out = tf.sigmoid(bd_out_1)

  return tanh_out * sigmoid_out
```

statistic.py

```
import os
import tensorflow as tf
from logging import getLogger

logger = getLogger(__name__)

class Statistic(object):
  def __init__(self, sess, data, runtime_base_dir, model_dir, variables, max_to_keep=20):
    self.sess = sess
```

```
    self.reset()

    with tf.variable_scope('t'):
      self.t_op = tf.Variable(0, trainable=False, name='t')
      self.t_add_op = self.t_op.assign_add(1)

    self.model_dir = os.path.join(runtime_base_dir, model_dir)
    self.saver = tf.train.Saver(variables + [self.t_op], max_to_keep=max_to_keep)
    self.writer = tf.train.SummaryWriter('%s/logs/%s' % (runtime_base_dir, model_dir),
self.sess.graph)

    with tf.variable_scope('summary'):
      scalar_summary_tags = ['train_l', 'test_l']

      self.summary_placeholders = {}
      self.summary_ops = {}

      for tag in scalar_summary_tags:
        self.summary_placeholders[tag] = tf.placeholder('float32', None, name=tag.replace(' ',
'_'))
        self.summary_ops[tag]  = tf.scalar_summary('%s/%s' % (data, tag),
self.summary_placeholders[tag])

  def reset(self):
    pass

  def on_step(self, train_l, test_l):
    self.inject_summary({'train_l': train_l, 'test_l': test_l}, self.t)
    self.save_model(self.t)
    self.t = self.t_add_op.eval(session=self.sess)
    self.reset()

  def get_t(self):
    return self.t

  def inject_summary(self, tag_dict, t):
    summary_str_lists = self.sess.run([self.summary_ops[tag] for tag in tag_dict.keys()], {
      self.summary_placeholders[tag]: value for tag, value in tag_dict.items()
    })
    for summary_str in summary_str_lists:
      self.writer.add_summary(summary_str, t)

  def save_model(self, t):
    logger.info("Saving checkpoints...")
    model_name = type(self).__name__

    if not os.path.exists(self.model_dir):
      os.makedirs(self.model_dir)
    self.saver.save(self.sess, self.model_dir, global_step=t)

  def load_model(self):
    logger.info("Initializing all variables")
    tf.initialize_all_variables().run()

    logger.info("Loading checkpoints...")
    ckpt = tf.train.get_checkpoint_state(self.model_dir)
    if ckpt and ckpt.model_checkpoint_path:
```

```
      ckpt_name = os.path.basename(ckpt.model_checkpoint_path)
      fname = os.path.join(self.model_dir, ckpt_name)
      self.saver.restore(self.sess, fname)
      logger.info("Load SUCCESS: %s" % fname)
    else:
      logger.info("Load FAILED: %s" % self.model_dir)

    self.t = self.t_op.eval(session=self.sess)
```

utils.py

```
import logging
logging.basicConfig(format="[%(asctime)s] %(message)s", datefmt="%m-%d %H:%M:%S")

import os
import sys
import urllib
import pprint
import tarfile
import tensorflow as tf

import datetime
import dateutil.tz
import numpy as np

import scipy.misc

pp = pprint.PrettyPrinter().pprint
logger = logging.getLogger(__name__)

def mprint(matrix, pivot=0.5):
  for array in matrix:
    print "".join("#" if i > pivot else " " for i in array)

def show_all_variables():
  total_count = 0
  for idx, op in enumerate(tf.trainable_variables()):
    shape = op.get_shape()
    count = np.prod(shape)
    total_count += int(count)
  logger.info("Total number of variables: %s" % "{:,}".format(total_count))

def get_timestamp():
  now = datetime.datetime.now(dateutil.tz.tzlocal())
  return now.strftime('%Y_%m_%d_%H_%M_%S')

def binarize(images):
  return (np.random.uniform(size=images.shape) < images).astype('float32')

def save_images(images, height, width, n_row, n_col,
      cmin=0.0, cmax=1.0, directory="./", prefix="sample"):
  channels = images.shape[3]
```

```
  if channels == 1:
    images = images.reshape((n_row, n_col, height, width))
    images = images.transpose(0, 2, 1, 3)
    images = images.reshape((height * n_row, width * n_col))
  else:
    images = images.reshape((n_row, n_col, height, width, channels))
    images = images.transpose(0, 2, 1, 3, 4)
    images = images.reshape((height * n_row, width * n_col, channels))

  filename = '%s_%s.jpg' % (prefix, get_timestamp())
  scipy.misc.toimage(images, cmin=cmin, cmax=cmax) \
      .save(os.path.join(directory, filename))

def get_model_dir(config, exceptions=None):
  attrs = config.__dict__['__flags']
  pp(attrs)

  keys = attrs.keys()
  keys.sort()
  keys.remove('data')
  keys = ['data'] + keys

  names =[]
  for key in keys:
    # Only use useful flags
    if key not in exceptions:
      names.append("%s=%s" % (key, ",".join([str(i) for i in attrs[key]])
          if type(attrs[key]) == list else attrs[key]))
  return os.path.join('checkpoints', *names) + '/'

def preprocess_conf(conf):
  options = conf.__flags

  for option, value in options.items():
    option = option.lower()

def check_and_create_dir(directory):
  if not os.path.exists(directory):
    logger.info('Creating directory: %s' % directory)
    os.makedirs(directory)
  else:
    logger.info('Skip creating directory: %s' % directory)
```