



Universidade Federal
do Rio de Janeiro

Escola Politécnica

APLICAÇÃO DE REDES NEURAIIS CONVOLUCIONAIS
DENSAMENTE CONECTADAS NO PROCESSAMENTO
DIGITAL DE IMAGENS PARA REMOÇÃO DE RUÍDO
GAUSSIANO

Leonardo Oliveira Mazza

Projeto de Graduação apresentado ao Curso
de Engenharia Controle e Automação da
Escola Politécnica, Universidade Federal do
Rio de Janeiro, como parte dos requisitos ne-
cessários à obtenção do título de Engenheiro.

Orientadores: Flávio Luis de Mello, D. Sc.

Heraldo Luís Silveira de Almeida, D. Sc.

Rio de Janeiro

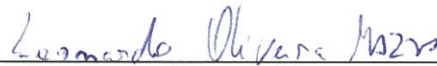
Fevereiro de 2017

APLICAÇÃO DE REDES NEURAIS CONVOLUCIONAIS
DENSAMENTE CONECTADAS NO PROCESSAMENTO
DIGITAL DE IMAGENS PARA REMOÇÃO DE RUÍDO
GAUSSIANO

Leonardo Oliveira Mazza

PROJETO DE GRADUAÇÃO APRESENTADO AO CURSO DE ENGENHARIA CONTROLE E AUTOMAÇÃO DA ESCOLA POLITÉCNICA, UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, COMO PARTE DOS REQUISITOS NECESSÁRIOS À OBTENÇÃO DO TÍTULO DE ENGENHEIRO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO.

Autor:



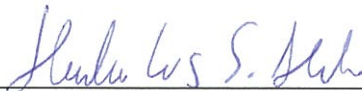
Leonardo Oliveira Mazza

Orientador:



Prof. Flávio Luis de Mello, D. Sc.

Orientador:



Prof. Heraldo Luís Silveira de Almeida, D. Sc.

Examinador:



José Gabriel Rodríguez Carneiro Gomes, Ph. D.

Examinador:



Pedro Henrique Pamplona Savarese, M. Sc.

Rio de Janeiro

Fevereiro de 2017

Declaração de Autoria e de Direitos

Eu, Leonardo Oliveira Mazza CPF 119.539.407-14, autor da monografia APLICAÇÃO DE REDES NEURAIS CONVOLUCIONAIS DENSAMENTE CONECTADAS NO PROCESSAMENTO DIGITAL DE IMAGENS PARA REMOÇÃO DE RUÍDO GAUSSIANO, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.



Nome do aluno

Mazza, Leonardo Oliveira

Aplicação De Redes Neurais Convolucionais Densamente Conectadas No Processamento Digital De Imagens Para Remoção De Ruído Gaussiano / Leonardo Oliveira Mazza. - Rio de Janeiro: UFRJ/Escola Politécnica, 2017.

XIV, 88 p.: il.; 29,7cm.

Orientadores: Orientadores: Flávio Luis de Mello,
Heraldo Luís Silveira de Almeida

Projeto de Graduação - UFRJ/Escola Politécnica/Curso de Engenharia de Controle e Automação, 2017.

Referências Bibliográficas: p. 78 - 82.

1. Redes Neurais Artificiais. 2. Redes Neurais Convolucionais Profundas. 3. Processamento de Imagens.
I. Mello, Flávio Luís de et al II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Controle e Automação. III. Título.

Aos meus avós Francisca, Vicente e Lidia.

AGRADECIMENTO

Agradeço à minha família pelo apoio e incentivo ao estudo.

Agradeço aos orientadores e professores pelas sugestões e correções.

Agradeço ao Pedro Savarese pela grande ajuda no trabalho.

Agradeço a Tatiane Tuha pelo apoio e pela ajuda na produção de ilustrações.

Agradeço às instituições de fomento a pesquisa (FAPERJ) pelo apoio ao trabalho.

Agradeço ao Laboratório de Instrumentação Biomédica por fornecer os equipamentos utilizados no trabalho.

RESUMO

Este trabalho propõe uma aplicação de uma rede neural convolucional profunda em remoção de ruído gaussiano de imagens. Redes profundas têm obtido êxito em diversas tarefas de classificação e regressão. Em particular, a Rede Neural Convolucional Densamente Conectada têm apresentado resultados competitivos em bases de dados padrão usadas em medidas de desempenho. Por isso, essa rede foi escolhida como modelo de regressão com o objetivo de remoção de ruído gaussiano. A metodologia para treinamento da rede foi iniciada pela criação de uma base de dados de imagens ruidosas. Essas imagens foram, então, colocadas na entrada da rede para filtragem e a saída foi comparada com a versão sem ruído a partir de uma função custo. Os parâmetros da rede foram, em seguida, alterados a partir de um método baseado em descida de gradiente a fim de minimizar o custo. Dessa forma, a rede treinada aprende a remover o ruído aplicado. O resultado final, comparado a outros métodos, possuiu melhores indicadores de desempenho em duas das três imagens de teste e obteve resultados próximos em uma terceira. Durante a filtragem com ruído de variância 400, as imagens Lena e Boats obtiveram SSIM de 0.84 e 0.83 respectivamente. Na imagem Lena, o resultado foi 0.05 superior ao método GSM e na imagem Boats 0.02 superior ao método SVR, ambos o segundo melhor método em cada imagem. Na imagem Barbara, o resultado foi o segundo melhor com SSIM 0.84, 0.02 inferior ao método GSM.

Palavras-Chave: Redes Neurais Artificiais, Redes Neurais Convolucionais Profundas, Processamento de Imagens.

ABSTRACT

This work evaluates an application of a deep convolutional neural network in image denoising, in particular gaussian noise removal. Deep convolutional neural networks have been successful in many regression and classification tasks. Notably, Densely Connected Convolutional Neural Network have shown competitive results in standard benchmark databases. Therefore, a regression model was chosen aiming at removing Gaussian noise. The training methodology began by the generation of a database of noisy images. Those images were, next, fed to the network and its output compared with the original version (without noise) by a cost function. The model parameter's were then updated according to a variation of gradient descent to minimize the cost. The trained network, thus, learnt to remove Gaussian noise. The final filter, compared to other methods, had the best result in two out of three test images and had similar results in the third one. Filtering in the case of noise variance of 400, images Lena and Boats had SSIM of 0.84 and 0.83 respectively. Lena image had 0.05 higher SSIM than GSM and Boats image had 0.02 higher than SVR, both second best methods for each case. Filtering Barbara image had second best result with SSIM 0.84, 0.02 lower than GSM.

Key-words: Artificial Neural Networks, Deep Convolutional Neural Networks, Image Processing.

SIGLAS

UFRJ - Universidade Federal do Rio de Janeiro

PEB - Programa de Engenharia Biomédica

ILSV - *ImageNet Large Scale Visual Recognition Competition*

COPPE - Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa em Engenharia

FAPERJ - Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro

CPU - *Central Processing Unit*

GPU - *Graphics Processing Unit*

BN - *Batch Normalization*

SSIM - *Structural Similarity Index*

RMSE - *Root Mean Square Error*

HT - *Hard Thresholding*

ST - *Soft Thresholding*

BL - *Bayesian Laplacian*

BG - *Bayesian Gaussian*

GSM - *Gaussian Scale Mixture*

SVR - *Support Vector Regression*

Sumário

1	Introdução	1
1.1	Tema	1
1.2	Delimitação	1
1.3	Justificativa	2
1.4	Objetivos	2
1.5	Metodologia	3
1.6	Descrição	4
2	Redes Neurais Convolucionais	5
2.1	Introdução	5
2.2	Interações Esparsas, Compartilhamento de Parâmetros e Campos Receptivos	10
2.3	<i>Pooling</i>	12
2.4	Treinamento	14
2.5	Regularização	17
2.6	Ativação	22
2.7	<i>Batch Normalization</i>	24
2.8	Classificação	26
2.9	Regressão	27
3	Redes Neurais Profundas	28
3.1	Arquiteturas e Desempenho	28
3.2	VGG	32
3.3	Residual Network	35
3.4	Rede Neural Convolucional Densamente Conectada	40

4	Implementação	45
4.1	Implementação de redes neurais em Keras	45
4.2	Rede Neural Convolutacional Densamente Conectada	54
4.3	Resultados Parciais	58
4.4	Remoção de Ruído Gaussiano	59
4.5	Resultados e Discussão	65
5	Conclusões	75
	Bibliografia	78
A	Código da implementação da Rede Neural Convolutacional Densa- mente Conectada	83

Lista de Figuras

2.1	Exemplo de filtro g 2×3 e <i>feature map</i> f 2×5 . Convolução é feita com filtro g' com ordem dos pesos invertida.	6
2.2	Convolução unidimensional de um <i>feature map</i> f 2×5 com um filtro g 2×3	7
2.3	Cálculo de um elemento A_1 de um <i>feature map</i> a partir de uma entrada com 3 <i>feature maps</i> f $3 \times 5 \times 5$ e um filtro g $3 \times 3 \times 3$. O cálculo de um elemento seguinte A_2 é feito deslizando-se o bloco em f para uma posição adjacente em cada um dos seus <i>feature maps</i> R, G e B.	9
2.4	Campos Receptivos em três camadas	11
2.5	Pooling (2,3) em um <i>feature map</i> 4×6	13
2.6	Exemplo de overfitting[1] no diagrama Erro x Época. A linha vermelha representa o erro de validação e a linha azul o erro de treino.	19
2.7	<i>Overfitting</i> [1]	19
2.8	Sem penalização L2	20
2.9	Com penalização L2	20
2.10	<i>Dropout</i> aplicado em uma Rede Neural(extraído de Srivastava et al[2]).	21
2.11	Resultados de treinamento com <i>Dropout</i> na base MNIST (extraído de Srivastava et al[2]).	22
2.12	ReLU[3]	23
2.13	PReLU[4]	23
2.14	Função Logística	24
2.15	(extraído de Krizhevsky et al[5]).	26
3.1	A composição de processos gera a observação de forma hierárquica (extraído de Lin et al[6]).	31
3.2	Ativações de uma rede neural convolucional (extraído de Zeiler et al[7]).	32

3.3	Um bloco residual com uma <i>shortcut connection</i> entre a entrada x e a saída (extraído de He et al [8]).	36
3.4	Bloco residual convolucional (extraído de He et al [8]).	36
3.5	Comparação entre duas redes convolucionais tradicionais (esquerda) e duas redes convolucionais residuais (direita). Curvas finas representam erro de treino e curvas em negrito a validação (extraído de He et al [8]).	38
3.6	Comparação entre redes convolucionais tradicionais (esquerda) e redes convolucionais residuais (meio) e uma rede convolucional residual extremamente profunda (direita) na CIFAR10. Curvas finas representam erro de treino e curvas em negrito a validação (extraído de He et al [8]).	38
3.7	Comparação entre duas redes de 1001 camadas na CIFAR10. Uma com o bloco residual original e outra com o novo bloco proposto. Curvas finas representam erro de treino e curvas em negrito a validação (extraído de He et al [9]).	40
3.8	Bloco Denso de 5 camadas e taxa de crescimento 4 (extraído de Huang et al [10]).	41
3.9	Estrutura de uma rede convolucional densamente conectada (extraído de Huang et al [10]).	43
3.10	Resultados de diferentes modelos nas bases de dados CIFAR10 e CIFAR100 com e sem <i>data augmentation</i> . ”+” indica o uso de <i>data augmentation</i> . Resultados em negrito indicam DenseNets com erros percentuais inferiores aos anteriores. ”*” indicam resultados gerados pelos autores da DenseNet para os quais não foram divulgados resultados originalmente (extraído de Huang et al [10]).	44
4.1	Função subjacente F que recebe um x e retorna um y	51
4.2	Função subjacente F que recebe um x e retorna um y	51
4.3	Dados para regressão. Círculos representam o dado ruidoso usado em regressão e a linha as amostras do modelo original	52
4.4	Resultado da otimização. Predição dos dados não usados em treino.	53
4.5	Curvas dos custos de treino e validação	54
4.6	Elemento básico de um bloco denso.	56
4.7	Cálculo do custo de uma entrada perturbada y para uma amostra original x	61

4.8	Quatro arquiteturas de rede testadas. A arquitetura <i>A</i> indica um cálculo de um bloco denso seguido de <i>batch normalization</i> (BN), ativação ReLU e uma convolução 1×1 que gera 1 <i>feature map em sua saída</i> . O diagrama seguinte <i>B</i> apresenta a mesma rede com a ativação ReLU removida. Em sucessão, a rede <i>C</i> com a adição de <i>batch normalization</i> , ReLU e convolução na primeira camada. Por fim, uma rede similar a acima com a adição de mais um <i>batch normalization</i> , ReLU e convolução na entrada.	62
4.9	Resultados de SSIM (acima) e RMSE (abaixo) para as imagens filtradas por cada modelo de rede para ruído de variância 200. Quanto mais próximo de 1, melhor o SSIM e quanto mais próximo de zero, melhor o RMSE.	66
4.10	Ilustração do erro absoluto(esquerda) e da imagem do SSIM(centro) da imagem <i>Lena</i> filtrada por <i>B K8 L8 v2</i> (acima) e por <i>D K8 L8</i> (abaixo). O erro absoluto da imagem filtrada pelo modelo <i>B K8 L8</i> apresenta manchas claramente presentes quando comparado ao erro da imagem filtrada por <i>D K8 L8</i> possivelmente responsáveis por seu maior valor RMSE. A imagem gerada durante o cálculo do SSIM não apresenta diferença significativa em cada caso, o que caracteriza o valor próximo observado. Verifica-se que o SSIM corretamente não possui uma alteração grande em seu valor dado que as imagens filtradas(direita) em cada caso não possuem diferenças muito significativas.	69
4.11	Resultados de SSIM (acima) e RMSE (abaixo) para as imagens filtradas por cada modelo de rede para ruído de variância 400. Quanto mais próximo de 1, melhor o SSIM e quanto mais próximo de zero, melhor o RMSE.	70
4.12	Resultado visual da filtragem em uma região da imagem <i>Lena</i> para o ruído gaussiano de variância 400. SSIM exibido entre parêntesis.	71
4.13	Resultados de SSIM (entre parênteses) na filtragem pelos métodos HT, ST, BG, BL, GSM, SVR para o caso do ruído gaussiano de variância 400 (extraído de laparra et al [11]). A imagem superior esquerda ilustra a imagem ruidosa.	72

4.14 Resultados de filtragem das imagens de teste *Barbara* (topo), *Boats* (meio),
Lena (abaixo). As imagens à esquerda indicam as versões ruidosas, as do
meio o resultado após o processamento e as à direita o valor absoluto da
diferença entre a imagem filtrada e a original sem ruído. 73

Lista de Tabelas

2.1	Tabela de Inferências da Rede 1	15
2.2	Tabela de Inferências da Rede 2	15
4.1	Erro percentual de teste nas bases de dados CIFAR100 (C100) e CIFAR10 (C10). Símbolo ”+” representa resultados com <i>data augmentation</i>	59
4.2	Treinamento de redes na arquitetura A. MSE é o menor erro médio quadrático obtido durante o treinamento no conjunto validação da CIFAR10. SSIM é o resultado da comparação da imagem Lena 256×256 cinza original com a imagem com ruído gaussiano de média zero e variância 400 filtrada pela rede avaliada pelo indicador <i>Structural Similarity Index</i> , quanto mais próximo de 1 melhor. O tempo t é o tempo por época do treinamento de cada rede e N seu número final de parâmetros.	63
4.3	Resultados do treinamento de modelos na arquitetura B	64
4.4	Resultados do treinamento de redes nas arquiteturas C e D, ambas com $K = 8$ e $L = 8$	65
4.5	Resultados[11] de SSIM e RMSE das imagens filtradas pelos métodos Hard Thresholding(HT)[12], Soft Thresholding(ST)[12], Bayesian Gaussian(BG)[13], Bayesian Laplacian(BL)[14], Gaussian Scale Mixture(GSM)[15], Support Vector Regression(SVR)[11] e uma borda superior de desempenho do SVR, o SVR^{opt} . O modelo proposto é denominado $D K8 L8$. Valores em negrito ressaltam os resultados SSIM mais próximos de 1 e menores RMSE em cada imagem.	67

4.6 Resultados[11] de SSIM e RMSE das imagens filtradas pelos métodos HT, ST, BG, BL, GSM, SVR e pelo modelo proposto *D K8 L8*. Valores em negrito ressaltam os resultados SSIM mais próximos de 1 e menores RMSE em cada imagem. 68

Capítulo 1

Introdução

1.1 Tema

O trabalho consiste na implementação da rede neural convolucional profunda do artigo Rede Neural Convolucional Densamente Conectada [10] e seu possível uso em processamento de imagens.

Sob esta óptica, faz-se uso de uma parte básica dessa rede, um bloco convolucional denso, como parte de um modelo usado numa regressão. Dessa maneira, usa-se um bloco fundamental do modelo da Rede Neural Convolucional Densamente Conectada para fins de se remover ruído gaussiano.

1.2 Delimitação

Além de remoção de ruído gaussiano, a reconstrução da imagem por redes neurais convolucionais pode, a princípio, ser usado para outras operações, como por exemplo, coloração de imagens preto e branco [16] [17]. Entretanto, como o tamanho das imagens usadas para treino é reduzido, é possível que sua aplicabilidade em imagens maiores possua menor efetividade. Este trabalho foi desenvolvido no âmbito do Laboratório de Inteligência de Máquina e Modelos de Computação (IM2C) da Escola Politécnica (Poli) da Universidade Federal do Rio de Janeiro (UFRJ) e do Laboratório de Instrumentação Biomédica (LIB) do Programa de Engenharia Biomédica (PEB) da COPPE.

1.3 Justificativa

Redes neurais convolucionais têm apresentado resultados relevantes para tarefas como classificação [10], inversão aproximada de funções [18] e predição de frames de vídeos [19]. Dessa forma, seu estudo tem impacto direto em assuntos como desenvolvimento de carros autônomos [20], melhoria de imagens médicas [21] e compressão de imagens [22]. Assim, sua relevância aumentou significativamente nos últimos anos.

Nesse contexto, a Rede Neural Convolucional Densamente Conectada tem se destacado por possuir o melhor resultado [10] em problemas de classificação de imagens em bases de dados padrão como CIFAR10 [23], CIFAR100 [23] e ImageNet [24]. Além disso, essa rede tem sido usada com sucesso em outras tarefas como em processamento de imagens para segmentação [25]. Como a parcela convolucional desse tipo de modelo costuma agir como extrator de características principais das imagens, é esperado que essa arquitetura seja um bom modelo para regressão a fim de remover ruído de imagens.

O modelo de ruído usado é o ruído gaussiano de média zero. Esse tipo de ruído pode surgir em imagens a partir de má iluminação, altas temperaturas e ruído em circuitos eletrônicos [26]. Outra justificativa para a escolha do modelo é o fato de que pelo teorema central do limite espera-se que o valor médio de outros tipos de ruído também se aproxime do modelo gaussiano.

1.4 Objetivos

O objetivo do trabalho é avaliar o emprego de Rede Neural Convolucional Densamente Conectada no processamento de imagens com ruído gaussiano. Assim, os objetivos específicos são: (1) reproduzir o erro de predição obtido por [10] na classificação das imagens da base de dados CIFAR10 com o uso da rede convolucional profunda proposta Rede Neural Convolucional Densamente Conectada. (2) comparar os resultados da reprodução e os resultados publicados segundo a diferença entre o erro de predição obtido e o publicado. (3) comparar o número de parâmetros

total do modelo produzido com uma outra reprodução apontada pelos autores. (4) utilizar uma das partes desse modelo como base para restauração de imagens com ruído gaussiano, empregando como métricas de qualidade o erro médio quadrático e Structural Similarity Index [27]. (5) avaliar a diferença entre a restauração a partir de outros métodos e a partir da rede neural.

1.5 Metodologia

A reprodução da Rede Neural Convolutacional Densamente Conectada foi feita com o uso da biblioteca Keras [28] em Python [29]. Keras é uma biblioteca construída a partir de dois backends: Theano [30] e Tensorflow [31]. Ambas as bibliotecas são capazes de gerar código em CUDA [32] o que permite processamento dos algoritmos de otimização em placa de vídeo.

Para fins de remover ruído gaussiano de imagens, foi usado um dos sub-blocos da Rede Neural Convolutacional Densamente Conectada como modelo matemático para regressão. Em seguida, foi feita uma busca de parâmetros desse modelo que minimizasse a diferença quadrática entre entrada ruidosa e sua versão original. Dessa forma, o modelo final aprende a aplicar a operação de remover ruído. A base de dados com ruído gaussiano foi gerada a partir da base CIFAR10 [23], composta por 60000 imagens RGB 32x32 pixels. Dessas, 45000 foram usadas para treino, 5000 usadas para validação e 10000 para teste. O ruído gaussiano adicionado às imagens, com média zero e desvio padrão 20, foi gerado com o uso da semente aleatória zero a partir da biblioteca numpy [33]. Após a estimação de parâmetros, faz-se uma avaliação do modelo de remoção de ruído a partir de outras imagens. Para tal, adiciona-se ruído gaussiano numa nova imagem usada como entrada para a rede. Em seguida, são comparados o indicadores de similaridade com a imagem original da imagem com ruído e da imagem processada. Por fim, comparam-se os resultados com os obtidos por outros métodos de filtragem [11].

1.6 Descrição

No capítulo 2 serão mostrados os princípios de funcionamento de uma rede neural convolucional. Além disso, também serão apresentados métodos que previnem *overfitting* e métodos que tornam possível o treinamento de modelos profundos.

O capítulo 3 apresenta a estrutura da Rede Neural Convolucional Densamente Conectada em detalhes e a comparação de seus resultados com os da reprodução.

O capítulo 4 mostra a estrutura usada a partir da rede para gerar o modelo usado na restauração de imagens com ruído gaussiano e seus resultados.

Por fim, apresenta-se a conclusão sobre a qualidade da reprodução e da restauração de imagens.

Capítulo 2

Redes Neurais Convolucionais

2.1 Introdução

Redes neurais convolucionais são redes neurais artificiais em que se aplica a operação de convolução em pelo menos uma de suas camadas. Esse tipo de rede foi desenvolvido para um conjunto particular de problemas em que cada amostra de uma base de dados segue uma topologia específica. Nessa topologia, considera-se a existência de uma relação entre valores de índice próximo em uma representação da amostra. Por exemplo, considerando-se uma imagem como uma matriz, espera-se que valores de índices próximos sejam altamente relacionados. Da mesma forma, considerando-se sinais de áudio como vetor espera-se que o valor de coordenadas com índices próximos sejam relacionados. Esse tipo de topologia é explorado em redes convolucionais a partir da sua operação mais fundamental, a convolução.

Define-se a convolução unidimensional discreta entre dois vetores f e g como:

$$f * g[x] = \sum_{m=-\infty}^{\infty} f[m]g[x - m]$$

No caso de processamento digital de áudio em um canal, esta equação é responsável por parte do cálculo da saída de uma primeira camada convolucional. Nesse exemplo, f é a entrada, g é o filtro que contém parte dos ganhos da camada e o resultado da convolução entre um dos filtros g e a entrada é chamado de *feature map*. Uma camada convolucional usual contém um conjunto de N filtros g . Como

f				
7	8	9	10	11
12	13	14	15	16

g		
1	2	3
4	5	6

g'		
3	2	1
6	5	4

Figura 2.1: Exemplo de filtro g 2×3 e *feature map* f 2×5 . Convolução é feita com filtro g' com ordem dos pesos invertida.

cada filtro g gera, na sua saída, um *feature map*, a saída de uma camada convolucional possui um conjunto de *feature maps* $\in R^{N \times M}$, em que N é o número de filtros e M a dimensão do vetor na saída de cada convolução. Assim, no caso particular de uma entrada f que inicialmente era um vetor (sinal de áudio em um canal), a saída de uma camada convolucional passa a ser uma matriz com um número de linhas N igual ao número de filtros g da camada anterior e número de colunas M igual ao tamanho de cada vetor na saída da convolução de cada filtro.

Num caso mais geral em que a entrada f de uma camada já seria uma matriz, a operação que define a convolução unidimensional é modificada. Nessa situação, a convolução entre um filtro g e um conjunto de *feature maps* f (Figura 2.1) será a convolução no sentido das linhas da matriz g e do conjunto de *feature maps*, como ilustrado na Figura 2.2.

No caso de imagens, o procedimento para o cálculo de uma saída convolucional é similar, diferenciando-se somente pelo uso de convolução bidimensional em vez de unidimensional. Define-se a convolução bidimensional discreta como:

$$f * g[x, y] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f[n, m]g[x - n, y - m]$$

Assim, da mesma forma que no caso do sinal de áudio, considerando-se como entrada de uma camada convolucional uma imagem cinza, para cada filtro g gera-se um *feature map* na saída da camada. A saída dessa primeira camada convolucional também possuirá uma dimensão adicional, um tensor $\in R^{F \times N \times M}$, em que F é o número de *feature maps*, N o número de linhas do tensor e M o número de colunas. Num caso mais geral, o filtro g de uma camada cuja entrada possui mais de um

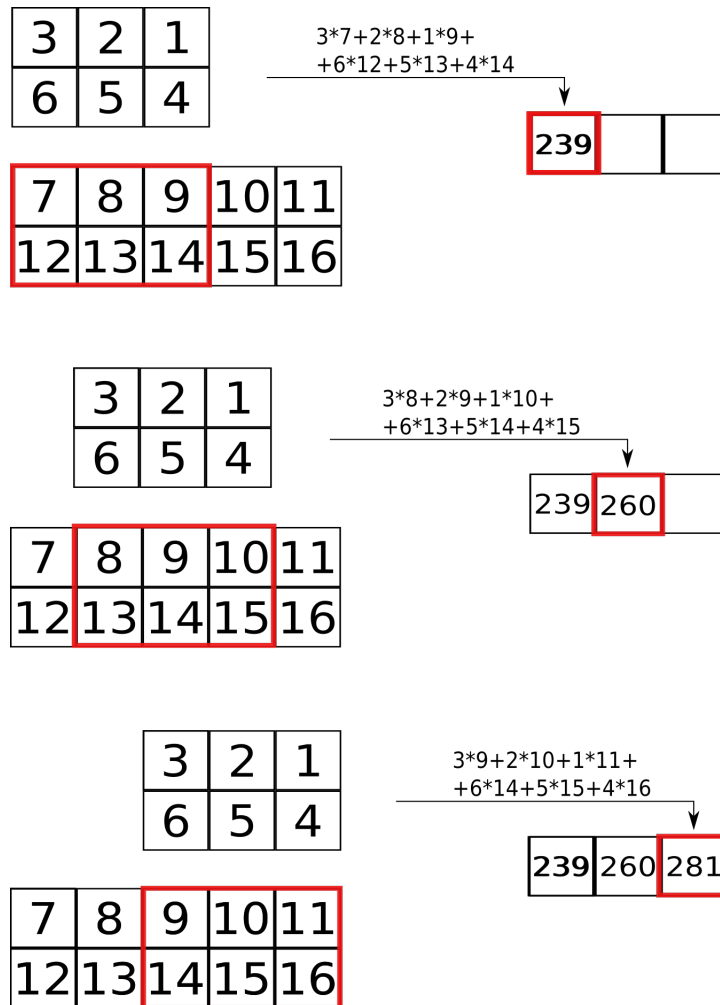


Figura 2.2: Convolução unidimensional de um *feature map* f 2×5 com um filtro g 2×3 . A coluna da esquerda ilustra o deslizamento da uma janela g sobre a entrada f , o que gera cada elemento do *feature map* à direita.

$feature\ map \in R^{F_{n-1} \times N_f \times M_f}$, em que F_{n-1} é o número de *feature maps* da camada anterior, N_f o número de linhas do filtro e M_f número de colunas do filtro. Com isso, o tensor que representa um filtro g pode ser visualizado como um cubo em que cada um dos elementos indexados por sua primeira dimensão (matrizes) será responsável por uma convolução com o respectivo *feature map* na sua entrada. Por fim, o resultado da convolução de cada um desses elementos indexados de um filtro g com cada um dos *feature maps* correspondentes da saída da camada anterior é somado, o que define o *feature map* na saída deste filtro. Com isso, a união de todos os *feature maps* gerados, um para cada filtro g da camada convolucional, gera a saída dessa camada. O procedimento para o cálculo de um dos *feature maps* na saída de uma camada convolucional é ilustrado na figura 2.3. A fim de se processar imagens com mais de um canal, pode-se considerar cada canal da imagem como um *feature map*. Isso fornece uma generalização natural para entradas multicanal o que permite o processamento de imagens em outros formatos, como RGB, RGBA e YCbCr. Nesse caso, o processamento se daria da mesma forma que o de uma camada convolucional com mais de um *feature map* na entrada.

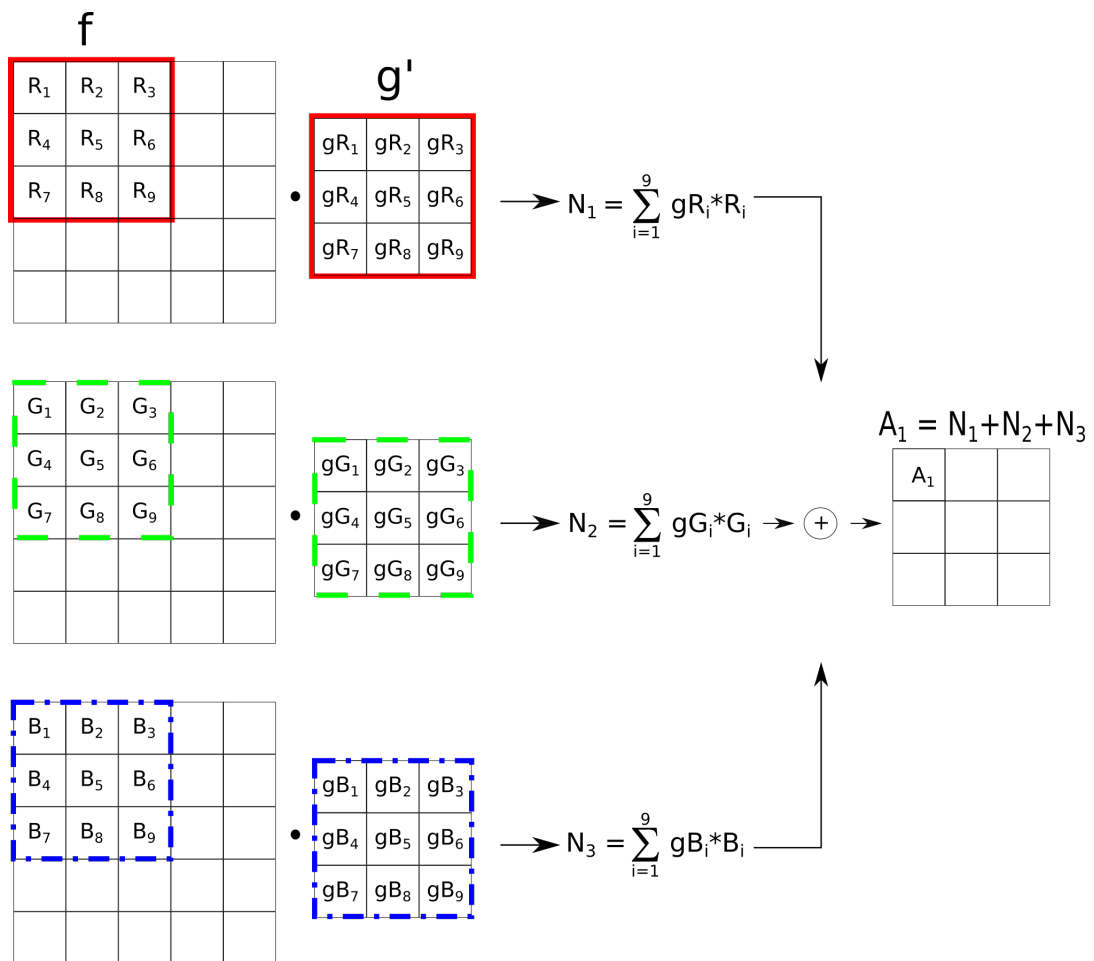


Figura 2.3: Cálculo de um elemento A_1 de um *feature map* a partir de uma entrada com 3 *feature maps* f $3 \times 5 \times 5$ e um filtro g $3 \times 3 \times 3$. O cálculo de um elemento seguinte A_2 é feito deslizando-se o bloco g em f para uma posição adjacente em cada um dos seus *feature maps* R , G e B . Ilustra-se uma convolução de uma entrada f com um filtro g somente nos valores válidos de f . Isso causa uma diminuição no tamanho final do *feature map* se comparado à entrada f . Um método comum para evitar essa alteração é adicionar zeros no contorno de f de tal forma que a saída resultante possua o mesmo tamanho da entrada.

2.2 Interações Esparsas, Compartilhamento de Parâmetros e Campos Receptivos

Conceitos importantes na análise de redes neurais convolucionais são Interações Esparsas, Compartilhamento de Parâmetros e Campos Receptivos [34]. Redes neurais tradicionais aplicam uma multiplicação de matrizes entre um vetor linha de entrada e a matriz de pesos da camada, o que gera o vetor linha de saída. Nessa matriz, cada coluna representa um neurônio e o produto interno entre o neurônio e o vetor de entrada gera um componente do vetor de saída. Esse tipo de operação faz com que cada elemento no vetor de entrada afete o vetor de saída. Diferentemente, a operação de convolução age somente sobre uma região definida pelas dimensões do filtro g da máscara de convolução. No caso de imagens, por exemplo, um filtro g com número de linhas e colunas igual a 3 causa uma dependência do valor central somente nos pixels adjacentes do próprio *feature map* e em todas as regiões 3×3 equivalentes dos outros *feature maps*. Se for considerada a saída de uma camada como um conjunto de 3 *feature maps*, cada um com tamanho 32×32 , o número de parâmetros do filtro g (e portanto o número de pixels usados) no cálculo do pixel central será $3 \times 3 \times 3 = 27$ pixels. Comparativamente, uma rede neural tradicional faria uso de todos os valores disponíveis para o cálculo dos pixels na saída, ou $3 \times 32 \times 32 = 3072$ pixels. Define-se a característica de que só ocorrem interações entre valores em regiões próximas de Campos Receptivos. Esse aspecto torna a convolução ideal para tratamento de sinais que podem apresentar grande dimensionalidade como sinais de áudio e imagens.

Observa-se ainda a partir do exemplo que a camada convolucional faz uso de um mesmo filtro g que irá iterar sobre toda a imagem para gerar o *feature map* na saída. Uma camada de uma rede neural tradicional, em comparação, possuiria uma coluna na sua matriz de pesos para cada pixel de saída. Assim, o conjunto de parâmetros que gera cada elemento do vetor de saída é usado somente uma vez para cada elemento da saída. Ao ato de se utilizar dos mesmos parâmetros para geração de mais de um elemento na saída dá-se o nome Compartilhamento de Parâmetros. Como se usa novamente o mesmo conjunto de parâmetros mais de uma vez, um dos seus

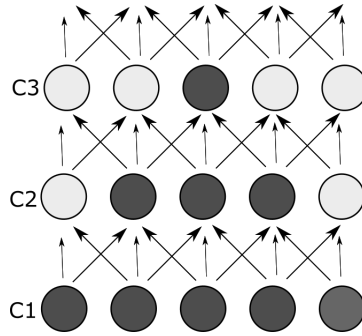


Figura 2.4: Campos Receptivos em três camadas

efeitos é um menor uso de memória da camada convolucional. Outra propriedade devido ao Compartilhamento de Parâmetros nas camadas convolucionais é o da equivariância à translação [34]. Uma função f ser equivariante a outra função g equivale a afirmar que, caso a entrada varie por um determinado valor, a saída terá seu valor alterado pelo mesmo número. Se f for equivariante a g , portanto $f(g(x)) = g(f(x))$. Assim, caso a imagem de entrada seja transladada por um determinado valor, a saída da camada convolucional também o será.

Nota-se também que, apesar das interações de uma camada convolucional serem esparsas, não se limita o acesso de filtros subsequentes a regiões maiores do sinal original. A informação originária em uma camada $C1$ chega numa subsequente $C3$ a partir de todos os valores nas regiões acessadas pelos filtros anteriores. Dessa maneira, se for considerado um sinal unidimensional f de tamanho 5 e um filtro g de tamanho 3, um elemento da camada $C3$ recebe informação de todos os elementos da primeira camada $C1$, como pode ser visualizado na Figura 2.4 em que cada seta representa um peso do filtro g .

Na imagem, um conjunto de três setas recebido por um mesmo elemento representa o filtro g . Chama-se campo receptivo a região de onde um elemento recebe informação. Na Figura 2.4 observa-se que o tamanho do campo receptivo de um elemento em $C3$, considerando-se a camada $C2$, será o tamanho do filtro g . Entretanto, se for considerado a camada $C1$ como a entrada, um elemento de $C3$ possui informação de um total de 5 elementos anteriores. Com isso, pode-se concluir que

elementos em camadas mais profundas de uma rede convolucional são capazes de receber informações de áreas maiores da imagem apesar das interações entre camadas serem esparsas.

Em suma, uma rede convolucional é um tipo especializado de modelo focado em conjuntos de dados com características específicas. A formulação de uma camada convolucional foi desenvolvida a fim de fazer uso de aspectos particulares o que permite um processamento mais eficiente desse tipo de dado. Como consequência, suas aplicações iniciais foram capazes de obter resultados significativamente superiores em competições de classificação [5] e seu desenvolvimento a mantém como melhor opção nesse tipo de tarefa [35]. A convolução como parte central do modelo matemático permite o processamento de sinais de grande dimensionalidade a partir de Compartilhamento de Parâmetros e Campos Receptivos o que diminui consideravelmente o número de parâmetros do modelo. Isso ocorre, entretanto, sem que haja perda de generalidade já que os Campos Receptivos de elementos em camadas mais profundas aumentam a cada camada. Por fim, o uso da convolução também dá ao modelo a propriedade da equivariância a translação, ou seja, caso a entrada seja transladada de um determinado valor, a saída também o será da mesma forma.

2.3 *Pooling*

Apesar de diminuir o problema da grande dimensionalidade de dados como imagens e sinais de áudio, as redes convolucionais usualmente ainda exigem reduções adicionais. Tarefas de classificação de imagens, por exemplo, podem envolver amostras RGB de tamanho 256×256 . Nesse caso, o número de atributos de uma entrada será $256 \times 256 \times 3 = 196608$. Como uma camada convolucional comum pode possuir por volta de 256 *feature maps*, calcula-se que a saída de uma camada intermediária possua $256 \times 256 \times 256 = 16777216$ atributos. Percebe-se com isso que a diminuição na dimensionalidade é essencial para a viabilidade modelo. As camadas de *pooling* são responsáveis por essa redução. Ao passar um *feature map* por um *pooling*, definem-se regiões a partir das quais somente um valor será enviado à camada seguinte. Diferentemente de uma camada convolucional, um *pooling* usualmente não altera o número de *feature maps*, em vez disso, reduz-se o número de linhas e/ou colunas

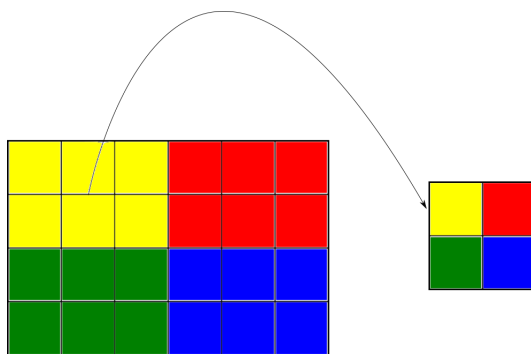


Figura 2.5: Pooling (2,3) em um *feature map* 4x6

da entrada. Por exemplo, um *pooling* de dimensões (2,3) em um *feature map* de dimensões 4x6 resulta num *feature map* com dimensões 2x2. Nesse exemplo, houve uma redução no número de atributos por um fator de 6. O procedimento é ilustrado na Figura 2.5 em que regiões de mesma cor do *feature map* 4x6 à esquerda gera valores para o *feature map* 2x2 à direita.

O cálculo da saída de um *pooling* pode ser definido de diversos modos, em que se aplica uma função a todos os elementos de uma mesma região e calcula-se a saída correspondente. As funções normalmente usadas são *Average Pooling* e *Max Pooling*. No primeiro, calcula-se a média de todos os valores da região para o cálculo da saída. No segundo, o valor da saída será o maior dentre os valores de uma região.

Além de reduzir o tamanho de cada *feature map*, o *pooling* também introduz invariância na rede [34]. Por exemplo, se for usado um *Max Pooling* e um *feature map* de tamanho 4x6 sofrer um deslocamento de 1 pixel para a esquerda, exige-se somente que o valor máximo de cada área não troque de região para que a saída não se altere. Da mesma forma, outros tipos de *pooling* como o *average pooling* também minimizam os impactos de alterações nas entradas na saída. A invariância do modelo é desejável, por exemplo, em problemas de classificação. Nesse caso, a presença de algo na imagem, como um focinho num cachorro [36], pode ser mais definitivo para uma classificação do que sua posição na imagem.

Em resumo, a redução dimensional é indispensável em muitas arquiteturas de redes convolucionais. As camadas *pooling* estão usualmente presentes a fim de se reduzir o número de parâmetros de cada *feature map*. Os métodos de *Pooling* mais comuns são *Max Pooling* e *Average Pooling* e seu uso ainda introduz uma maior invariância à entrada.

Existe ainda, um tipo especial de *pooling* que só é usado previamente à última camada convolucional, o *Global Average Pooling*[37]. Como será discutido na seção 2.8, a saída de *feature maps* de uma rede convolucional usada em classificação é passada para uma rede neural tradicional. Entretanto, essas últimas camadas tradicionais podem possuir um grande número de parâmetros e ter propensão à má generalização [37]. Por isso, introduz-se o *global average pooling*. Nele, reduz-se a dimensão dos *feature maps* da última camada convolucional para 1×1 ao se tirar a média de cada um individualmente. Com isso, ocorre uma redução da dimensionalidade da última camada convolucional, o que diminui a propensão à má generalização [37].

2.4 Treinamento

O treinamento de uma rede neural convolucional ocorre exatamente da mesma forma que o treinamento de uma rede neural tradicional. O procedimento de estimação de parâmetros dá-se a partir de um processo de otimização não linear em que se minimiza uma função custo relacionada à acurácia do modelo. Considerando-se um problema de classificação, cada amostra em uma base de dados possui um rótulo associado a si. Nesse caso, o objetivo final de um modelo seria minimizar o erro de classificação. Entretanto, a função custo usualmente não é construída a partir do erro de classificação, mas a partir de medidas da distância entre a inferência e o rótulo real. Isso ocorre porque dois modelos distintos podem possuir a mesma acurácia em um conjunto de dados, mas qualidade real diferente. Por exemplo, numa tarefa de classificação de imagens em que se quer decidir se uma imagem representa uma fruta ou um animal o resultado de uma inferência pode ser o representado na Tabela 2.1.

Tabela 2.1: Tabela de Inferências da Rede 1

Rede 1	Inferência		Real	
	Fruta	Animal	Fruta	Animal
Amostra 1	0,6	0,4	1	0
Amostra 2	0,3	0,7	0	1

Tabela 2.2: Tabela de Inferências da Rede 2

Rede 2	Inferência		Real	
	Fruta	Animal	Fruta	Animal
Amostra 1	0,8	0,2	1	0
Amostra 2	0,1	0,9	0	1

Observa-se a partir das tabelas 2.1 e 2.2 que ambas as redes foram capazes de acertar a classificação real das duas amostras. Apesar disso, também se percebe que a rede 2 foi capaz de inferir com um maior grau de certeza as amostras corretas. Por esse motivo, a medida usada nas funções custo de redes neurais não incorre exatamente na acurácia do modelo, mas em medidas da distância entre a inferência e o rótulo real. Exemplos comuns de funções custo são o erro médio quadrático e entropia cruzada. O custo então é definido a partir da amostra observada e da inferência do modelo. Considerando-se o erro médio quadrático como função custo, seu cálculo é:

$$E(W, X, y) = \frac{1}{N} \left(\sum_{i=0}^{N-1} (F(W, X_i) - y_i)^t (F(W, X_i) - y_i) \right)$$

Em que X é o conjunto de entrada do modelo, y o conjunto de cada rótulo real correspondente, W o conjunto de parâmetros, F a inferência (função da entrada e dos parâmetros) e N o número de amostras em X usadas. No exemplo, o custo da primeira rede com o erro médio quadrático:

$$Custo_1 = \frac{1}{2} ((0 - 0.3)^2 + (1 - 0.7)^2 + (0 - 0.4)^2 + (1 - 0.6)^2) = 0.5$$

No caso da segunda rede o erro é

$$Custo_2 = \frac{1}{2} ((0 - 0.1)^2 + (1 - 0.9)^2 + (0 - 0.2)^2 + (1 - 0.8)^2) = 0.05$$

Determinando-se o custo a partir da inferência corretamente expressa a qualidade da segunda rede capaz de se aproximar melhor do rótulo real, diferentemente da acurácia em que ambas teriam o mesmo valor.

Quando se utiliza toda a base de dados a fim de se gerar a função custo (isto é, N são todas as amostras do treino em X), denomina-se otimização em batch ou batelada. Usualmente, entretanto, escolhe-se N como uma pequena parcela de toda a base de treino, a esse método se denomina otimização em minibatch. Nesse segundo caso, um passo de otimização ocorrerá para cada subconjunto dos dados de cada minibatch. Enfim, os parâmetros são estimados a partir de algum método de otimização, em que se faz a regressão a partir da minimização do custo.

Métodos de otimização em redes neurais usualmente são baseados em descida de gradiente. Como até modelos simples podem alcançar ordens de 10^6 ou mais parâmetros, opta-se por métodos que exigem uma quantidade de memória linearmente relacionada com o número de parâmetros. O método de otimização mais básico é a descida de gradiente estocástica. A atualização em uma de suas iterações ocorre decrescendo-se o vetor de parâmetros a partir do vetor gradiente do custo no ponto atual (do espaço de parâmetros) escalado por um parâmetro α :

$$W_{n+1} = W_n - \alpha \nabla E(W_n)$$

Na equação, W_n é o vetor de todos os parâmetros do modelo no passo atual, α é a constante que multiplica o gradiente do custo, denominada taxa de aprendizado, e ∇E o gradiente da função custo. No caso de otimização em batch, toda a base de dados é usada em uma única atualização do gradiente. Em minibatches, em contraste, são feitas diversas atualizações para uma única utilização de toda a base de dados. O número de atualizações depende da relação entre o tamanho do minibatch e o tamanho total da base. Por exemplo, se um conjunto de dados possui 64000 amostras e o tamanho do minibatch é de 64, serão feitas 10000 atualizações nos parâmetros em comparação a 1 atualização no caso de otimização em batch. Essa grande diferença faz modelos treinados com minibatch normalmente convergirem mais rapidamente. Minibatches menores, contudo, são uma representação pior da base original, o que diminui a qualidade de um passo da descida de gradiente. Quando todas as amostras de treino são usadas para atualização dos parâmetros, denomina-se uma época. Um treinamento de uma rede neural segue de um conjunto de épocas a fim de minimizar uma função custo relacionada a acurácia do modelo.

2.5 Regularização

Algoritmos em aprendizado de máquina têm por objetivo obter resultados acurados tanto em dados previamente observados como também em novas amostras. Em um conjunto particular de algoritmos, os supervisionados, parte de amostras são associadas a rótulos com o objetivo de se inferir um rótulo a partir de atributos da amostra. Nesse tipo de aplicação deseja-se ser capaz de corretamente prever

o rótulo tanto de amostras anteriormente presentes como também de novas amostras. Entretanto, modelos em redes neurais costumam possuir um grande número de parâmetros de modo a possuir grande poder de representabilidade. Por isso, esse tipo de modelo é sujeito a *overfitting*. *Overfitting* ocorre quando há uma grande diferença entre a acurácia do modelo para amostras já observadas e novas amostras. Isso pode ocorrer porque o modelo decora as amostras observadas anteriormente ou também porque se ajusta sobre ruído em vez do verdadeiro modelo subjacente. Por esse motivo, para se medir a real capacidade de generalização, faz-se uso de validação cruzada. Essa técnica divide a base de dados em três conjuntos: treino, validação e teste. As amostras do conjunto de treino são as usadas no modelo para se efetuar a estimação de parâmetros enquanto o conjunto de validação é usado para se estimar a validade do modelo em um conjunto de dados não anteriormente observados. Durante todo o treinamento, a validação pode ser computada como aproximação inicial da real acurácia do modelo. Dessa forma, essa parte da base de dados é usada para se escolher parâmetros externos ao modelo (hiperparâmetros), como arquitetura ou parâmetros presentes em algoritmos de otimização. Contudo, a alteração de hiperparâmetros a fim de se minimizar um erro de validação pode causar um viés que torna o erro de validação uma versão subestimada do erro real [38]. Com isso, define-se o conjunto de teste, usado somente se estimar a acurácia do modelo, sem a alteração de parâmetros ou hiperparâmetros. Objetiva-se, então, que um modelo seja capaz de obter uma baixa taxa de erros no conjunto de treino sem que haja uma taxa de erros muito superior nos conjuntos de teste e validação. Nessa óptica, o *overfitting* pode ser observado no momento do treinamento quando o erro de treino diminui e o erro de validação aumenta como pode ser visto na Figura 2.6.

No momento demarcado pela exclamação, o erro de treino decresce e o erro de validação aumenta, o que caracteriza o *overfitting*. Esse tipo de evento ocorre quando, por exemplo, o modelo se ajusta sobre o ruído em vez do verdadeiro modelo subjacente. Quando isso ocorre, o modelo passa a classificar amostras do conjunto de teste erroneamente baseado no treino sobre o ruído, visível na Figura 2.7.

O modelo subjacente é representado pela linha preta que divide as amostras azuis e vermelhas em dois grupos. Um modelo afetado por *overfitting* é representado

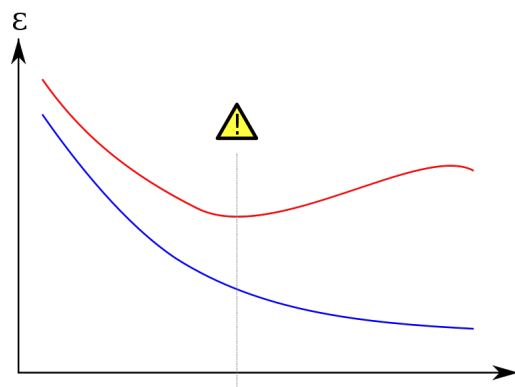


Figura 2.6: Exemplo de overfitting[1] no diagrama Erro x Época. A linha vermelha representa o erro de validação e a linha azul o erro de treino.

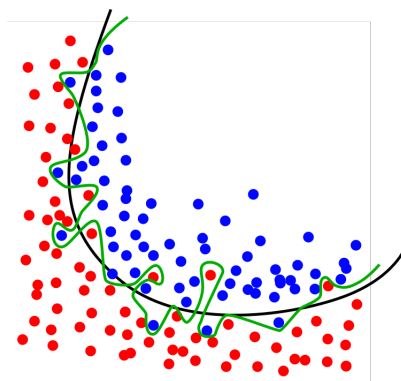


Figura 2.7: *Overfitting*[39]

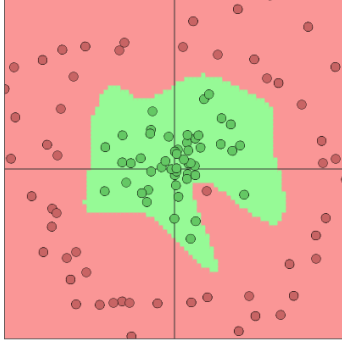


Figura 2.8: Sem penalização L2

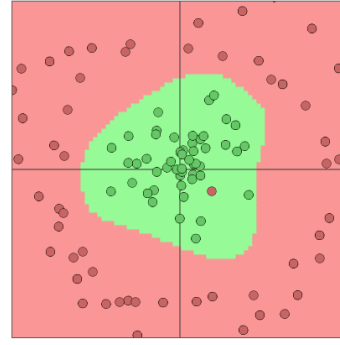


Figura 2.9: Com penalização L2

pela linha verde que contorna cada amostra possível a fim de minimizar o erro durante um treino. Num momento de teste, as amostras com ruído possivelmente estariam situadas em posições diferentes. Com isso, caso uma amostra azul durante o teste esteja situado na região onde um *outlier* vermelho esteve durante o treino, essa amostra seria classificada erroneamente como vermelha.

Métodos para redução de *overfitting* são chamados regularização. Esse tipo de método tenta reduzir a diferença entre o erro de estimação na base de treino e o erro na validação. Alguns exemplos comuns são a penalização por normas L1 e L2 [40] e *Dropout* [2]. As penalizações L1 e L2 adicionam um fator à função custo com uma norma dos parâmetros do modelo. A norma L1 consiste na adição do valor absoluto dos pesos à função custo caracterizado por:

$$L1Norm = \alpha \sum_{i=1}^N |W_i|$$

Enquanto a norma L2:

$$L2Norm = \alpha \sum_{i=1}^N W_i^2$$

W consiste em todos os parâmetros que se deseja penalizar no modelo. Exemplifica-se o uso da penalização com a ferramenta ConvnetJS [41]. As Figuras 2.8 e 2.9 mostram uma base de dados de pontos distribuídos de forma circular em que foi adicionado um ponto vermelho dentro da região de pontos verdes como um ruído. O exemplo mostra duas situações, na primeira, o resultado do treino sem a adição de norma l2 e a segunda com norma l2=0.0001:

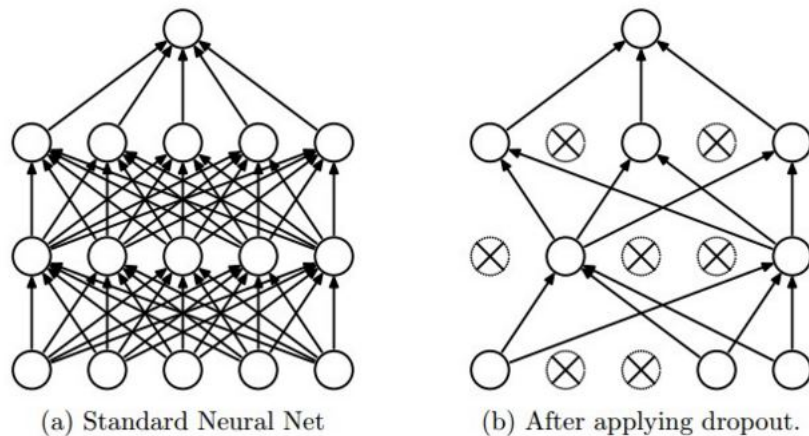


Figura 2.10: Dropout aplicado em uma Rede Neural (extraído de Srivastava et al[2]).

Observa-se claramente que o modelo gerado sem norma L2 realizou um ajuste que inclui em si o ponto vermelho que representa o ruído. Com uso de norma L2, entretanto, o modelo na Figura 6 foi capaz de generalizar melhor ao ignorar a presença de um ponto vermelho na região central dos pontos verdes. Uma limitação presente nesse método decorre do fato que os valores adequados dos parâmetros L1 e L2 adicionados no modelo variam caso a caso. Assim, a escolha de um valor ideal exige uma busca de parâmetros.

Outro método utilizado para redução de overfitting é o *Dropout*. Esse método consiste no descarte de aleatório de unidades entre camadas da rede o que dificulta a co-adaptação de parâmetros. Argumenta-se [2] que um dos motivos que permite que redes neurais decorem entradas durante o treino e se ajustem ao ruído de amostras advém da adaptação conjunta de parâmetros da rede. Então, o método proposto define um modo para se dificultar a coordenação entre parâmetros durante o treinamento. Dessa forma, diminui-se, entretanto, a capacidade de inferência do modelo no treino, mas se melhora a capacidade de generalização final. Visualiza-se o método na Figura 2.10.

Como as conexões entre camadas são interrompidas de maneira aleatória durante o treino, a co-adaptação de parâmetros se torna mais difícil, o que tende

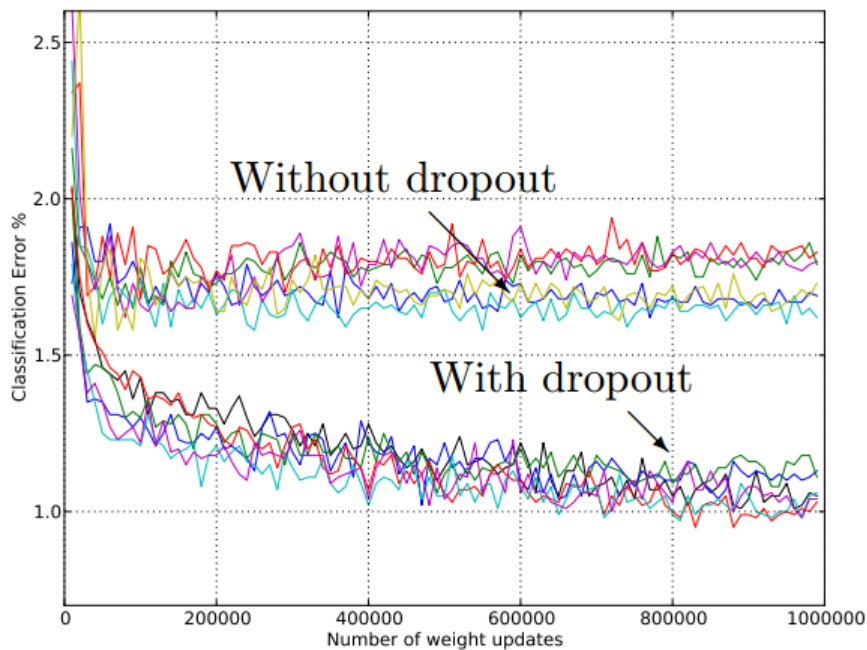


Figura 2.11: Resultados de treinamento com *Dropout* na base MNIST (extraído de Srivastava et al[2]).

a aumentar a capacidade de generalização. Resultados apresentados indicam uma melhora significativa no erro de validação após a adição de *Dropout*, como visto na Figura 2.11.

Nos exemplos mostrados, o uso de *dropout* foi consistentemente superior à sua ausência. Na Figura 2.11 observa-se a redução do erro de classificação de por volta de 1,6% para 0,8%. A aplicação desse método causou a queda do número de amostras classificadas erroneamente pela metade. Esse tipo de resultado demonstra a importância de regularização em modelos de aprendizado de máquina.

2.6 Ativação

O poder de representação de redes neurais artificiais reside na capacidade de se ajustar satisfatoriamente em modelos não lineares. Redes neurais artificiais tradicionais aplicam uma multiplicação de matrizes entre camadas seguida de uma função não linear. Denomina-se a função não linear função de ativação. Caso não se aplicasse funções não lineares entre camadas, a saída do modelo seria a composição

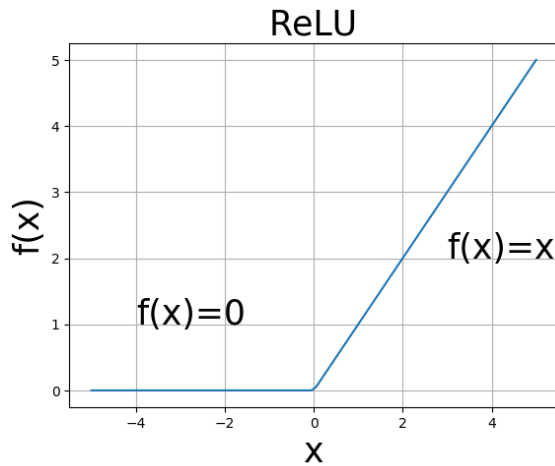


Figura 2.12: ReLU[3]

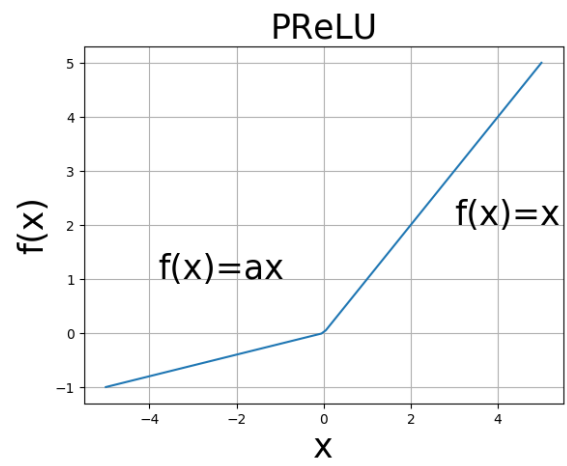


Figura 2.13: PReLU[4]

de multiplicações de matrizes, que se resume à multiplicação de uma única matriz. Nesse exemplo, o modelo não seria capaz de representar funções não lineares. Por isso, adicionam-se funções de ativação entre cada camada do modelo. Isso também se estende para o caso de redes neurais convolucionais: a convolução é uma operação linear. Assim, a adição de alguns tipos de funções ativação não lineares aumenta a capacidade de representação do modelo. Esse efeito é provado no teorema da aproximação universal [42].

Alguns exemplos de função de ativação são ReLU [3], PReLU[4] e Logística (Figura 2.14). A ReLU zera entradas inferiores a zero e retorna o mesmo valor caso receba uma entrada positiva (Figura 2.12). A PReLU escala uma entrada negativa por um fator α e retorna a mesma entrada caso ela seja positiva (Figura 2.13).

Outra função de ativação comum é a logística com sua equação definida por:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Cada uma das funções possui um conjunto de propriedades. A sigmoide deixou de ser usada devido ao rápido decréscimo de seu gradiente com o número de camadas do modelo. A ReLU e PReLU, fazem parte de modelos padrão usados atualmente em um grande número de redes [10] [17] [35]. Propriedades positivas dessas funções são a derivada constante unitária para valores de entrada positivos. Com a

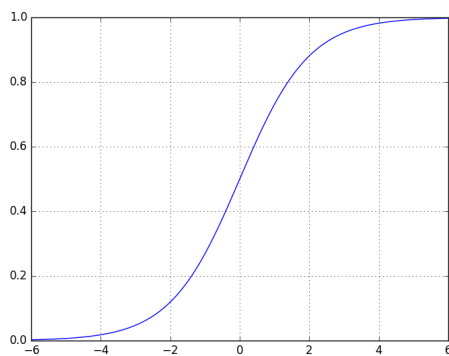


Figura 2.14: Função Logística

adição de camadas, o cálculo do gradiente do custo depende multiplicativamente das derivadas das funções de ativação. Assim, a presença de uma derivada constante é algo desejável. No conjunto de entradas negativas, a PReLU diferencia-se da ReLU por possuir um parâmetro estimável α . Argumenta-se que a adição desse parâmetro foi responsável por um dos primeiros modelos capazes de superar o desempenho humano em certos problemas de classificação [4].

2.7 *Batch Normalization*

Um método usado para se diminuir a interdependência entre camadas de um modelo é o *Batch Normalization*[43]. Como discutido na seção 2.4, o treinamento de redes neurais usualmente é feito por otimização em *minibatches*. Um dos problemas reportados no treinamento de redes é o fato de que ocorre alteração de propriedades estatísticas de uma camada para outra do modelo. O uso de uma função de ativação sem média zero pode, por exemplo, alterar o valor médio de um *minibatch* de uma camada para outra. Esse tipo de efeito torna o treino mais errático e dificulta convergência. O método do *batch normalization* propõe a adição de uma camada de normalização que diminui esse problema.

A operação realizada por uma camada de *batch normalization* atua de duas formas diferentes. Considerando-se X o conjunto de amostras de um *minibatch*, primeiro

subtrai-se o valor médio μ_i de cada atributo x_{ki} das N amostras X_k com n atributos cada de um *minibatch* seguido de uma divisão pelo seu desvio padrão σ_i .

$$X_k = \{x_{k1}, x_{k2}, x_{k3}, \dots, x_{ki}, \dots, x_{kn}\}$$

$$\mu_i = \frac{1}{N} \times \sum_{m=1}^N x_{mi}$$

$$\sigma_i = \sqrt{\frac{1}{N-1} \times \sum_{m=1}^N (x_{mi} - \mu_i)^2}$$

$$x'_{ki} = \frac{x_{ki} - \mu_i}{\sigma_i}$$

Em seguida, multiplica-se o atributo x'_{ki} da amostra por um parâmetro γ seguido pela adição de um novo parâmetro β .

$$x''_{ki} = \gamma x'_{ki} + \beta$$

Por fim, x''_{ki} é o valor de saída do atributo i de uma amostra k de um *minibatch*.

A subtração pela média seguida pela divisão pelo desvio causa uma diminuição na dependência entre camadas visto que camadas anteriores não mais seriam capazes de alterar o primeiro e segundo momentos estatísticos centrais do *minibatch* em camadas seguintes. Entretanto, caso seja positivo para um modelo que se altere os valores médio e variância entre camadas, o *batch normalization* causaria um detrimento na otimização. Por isso, existe um segundo passo em que dois parâmetros por atributo da amostra são adicionados na camada, o que dá a capacidade de se adicionar um valor médio e se multiplicar por um escalar. Assim, se torna possível que o *batch normalization* implemente uma identidade. Resultados de uma maior independência entre camadas incluem convergência mais acelerada, aumento na acurácia e efeitos de regularização [43]. Essas melhorias tornaram o *batch normalization* parte essencial de diversos modelos competitivos [10] [35].

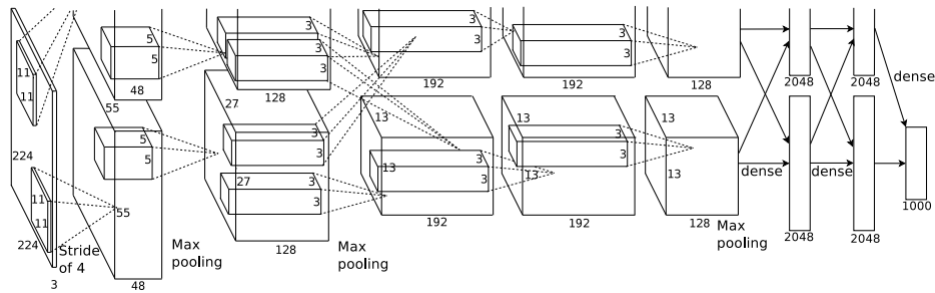


Figura 2.15: Rede Convolutacional AlexNet (extraído de Krizhevsky et al[5]).

2.8 Classificação

Uma das primeiras redes neurais convolucionais bem sucedidas participou da competição de classificação de imagens em alta resolução ImageNet. Sua arquitetura ilustra um formato de uma rede convolutacional (Figura 2.15).

A imagem de entrada tem um tamanho $224 \times 224 \times 3$, totalizando 150528 atributos na entrada. Uma primeira camada convolutacional é aplicada com convoluções de tamanho $11 \times 11 \times 96$, o que resulta nos 96 *feature maps* da camada seguinte. Num próximo passo, ocorrem convoluções de tamanho $5 \times 5 \times 256$ seguido de um *max pooling* que reduz o tamanho de cada *feature map* para 27×27 . Na camada convolutacional seguinte, é aplicada uma convolução $3 \times 3 \times 384$ e novamente aplica-se um *max pooling* que reduz o *feature map* para 13×13 . Em seguida, o número de *feature maps* e de camadas convolucionais é mantido com a aplicação de uma convolução $3 \times 3 \times 384$. Uma última convolução $3 \times 3 \times 256$ é aplicada seguida de um último *max pooling*. O resultado dessas operações é finalmente transformado em um vetor que é passado a uma rede neural artificial tradicional com múltiplas camadas. Nessa rede, a matriz de pesos da primeira e segunda camada da rede tradicional possui 4096 colunas, totalizando 4096 neurônios cada. Por fim, a saída da última camada é enviada aos 1000 neurônios da última camada em que se aplica a função *softmax*. Essa última camada é responsável por normalizar as saída de modo que o somatório de todas as saídas da rede possua valor 1. A saída da *softmax* possui a mesma dimensão da entrada, o cálculo de um de seus elementos j dá-se por:

$$\text{Softmax}(x)_j = \frac{e^{x_j}}{\sum_{i=1}^N e^{x_i}}$$

Essa rede ilustra a arquitetura fundamental de uma rede convolucional: camadas convolucionais alternadas com *pooling* que terminam numa rede neural tradicional para classificação. Modelos mais recentes empregam ainda, o uso de *batch normalization* e *Dropout*, numa sequência comum de Convolução, *batch normalization*, Ativação, *Pooling*, *Dropout*.

2.9 Regressão

Redes neurais convolucionais aplicadas em problemas de regressão possuem arquitetura semelhante ao apresentado na Figura 2.15. Um modelo simples de regressão para uma saída de mesmas dimensões da entrada seria idêntico ao mostrado, com a exceção das camadas de *pooling* e sem a saída numa rede neural tradicional. Dessa forma, alguns modelos de regressão apresentam sequências de camadas como Convolução, *batch normalization*, Ativação, *Dropout*. Na última camada, entretanto, não se usa *Dropout* e pode-se não aplicar uma ativação, optando-se por uma camada linear. Usualmente, a medida de erro nesse caso é algo como o erro médio quadrático ou erro médio absoluto.

Nesse tipo de rede, o rótulo usado no aprendizado supervisionado usualmente possui o mesmo tamanho da entrada. Assim, caso a tarefa de regressão seja a remoção de ruído de um áudio, a entrada será o áudio com ruído e o rótulo será sua versão sem ruído. Dada uma função custo do tipo erro médio quadrático, a tarefa da rede será aproximar ao máximo a saída inferida do rótulo, ou seja, indiretamente a rede aprende a remover ruído do áudio. Esse mesmo tipo de formatação pode ser usado se efetuar outros tipos de tarefas, como a solução de problemas inversos. Nesse caso, o rótulo é a amostra original e a entrada da rede é a amostra processada por alguma transformação. Por exemplo, caso se deseje remover *blur*, a entrada da rede será a imagem com *blur* e o rótulo a imagem original. Da mesma forma, pode-se treinar uma rede para colorir imagens usando-se uma imagem original como rótulo e sua versão em preto e branco como entrada.

Capítulo 3

Redes Neurais Profundas

3.1 Arquiteturas e Desempenho

Redes neurais profundas são aquelas que possuem múltiplas camadas ocultas [44]. Uma camada oculta é uma camada que não é uma entrada nem uma saída. Por exemplo, uma rede neural tradicional profunda f para classificação pode ser composta de três matrizes de peso h_1, h_2 e h_3 seguidas de uma função *softmax*:

$$h_1 = ReLU(xW_1)$$

$$h_2 = ReLU(h_1W_2)$$

$$h_3 = ReLU(h_2W_3)$$

$$f(x) = Softmax(h_3)$$

Nesse exemplo, cada camada oculta é representada a partir de um h_i . Pode-se observar ainda que uma camada é associada a uma entrada e a uma matriz de peso W_i . Cada uma recebe como entrada a saída da camada anterior, o que caracteriza a profundidade. Caso uma amostra de entrada seja um vetor com 4 atributos, a matriz de pesos W_1 possuirá 4 linhas. Seu número de colunas é livre, mas define o número de linhas da matriz de peso seguinte W_2 . Assim, caso W_1 possua 10 colunas, W_2 será uma matriz com 10 linhas. A última matriz de pesos W_3 tem seu número de linhas definido pelo número de colunas da matriz anterior e seu número de colunas definido a partir do número de classificações possíveis. Dessa forma, caso W_2 possua 30 colunas e a tarefa de classificação exija 8 classificações possíveis, W_3 será uma

matriz 30×8 . Ou seja, uma camada $W_i \in R^{N \times K}$ possui N linhas tal que N é o número de colunas da camada anterior e K o número de linhas da camada W_{i+1} . Isso mostra que para se definir a estrutura básica de uma rede neural tradicional só é necessária a definição do número de colunas de todas as matrizes de peso.

Entretanto, com o aumento da profundidade surgem problemas durante a otimização do modelo. A divergência do gradiente ou sua convergência para zero em pontos de sela é algo comum em espaços de grande dimensionalidade[44]. Uma das técnicas apresentadas capazes de facilitar a convergência do treinamento é o *batch normalization*. Sua presença dificulta divergência e adiciona um fator de regularização [43]. A posição comum de um *batch normalization* é após a transformação linear e anterior à ativação, como:

$$\begin{aligned} h_1 &= ReLU (BN (xW_1)) \\ h_2 &= ReLU (BN (h_1W_2)) \\ h_3 &= ReLU (BN (h_2W_3)) \\ f(x) &= Softmax (h_3) \end{aligned}$$

Apesar de aumentar a complexidade da otimização, o aumento do número de camadas permite o aprendizado de aspectos de mais alto nível nos dados. A presença de mais de uma camada oculta faz com que o modelo seja capaz de aprender características de maneira hierárquica [44]. Isto é, camadas mais próximas da entrada aprendem alguns tipos de características diferentes das camadas seguintes que se basearão nelas. Com isso, espera-se que camadas iniciais sejam responsáveis pela detecção de traços básicos e camadas mais profundas pela detecção da sua composição. Esse caráter hierárquico pode ser a razão para o sucesso das redes neurais profundas.

Argumenta-se [6] que a ubiquidade dos processos de composição na natureza cause esse efeito. Anteriormente à aquisição de uma amostra, o dado de interesse passa por uma série de transformações até ser colhido. A composição dessas transformações dá origem à amostra obtida. Na Figura 3.1 esse processo é ilustrado. Na coluna di-

reita vê-se de cima para baixo um processo de geração de uma imagem por software. Parte-se de conceitos básicos como a definição original do que será desenhado, seguido de parâmetros do programa, métodos de síntese de imagens e transformações até a imagem final. Em cada passo, ocorre uma composição de mudanças nos dados originais que gera o dado final. Na coluna esquerda o mesmo ocorre para observações astronômicas. Parâmetros cosmológicos fundamentais sofrem um conjunto de mudanças até que sejam adquiridos como dados em um telescópio. O fato que modelos que explicam fenômenos de uma maneira hierárquica são bem sucedidos indica que arquiteturas nesse mesmo formato também podem o ser.

Assim, uma abordagem hierárquica torna-se uma opção natural no procedimento inverso de se obter uma informação a partir da amostra. Particularmente no contexto de imagens, procedimentos de visualização[7] são capazes de demonstrar empiricamente que camadas iniciais de uma rede convolucional são ativadas por características básicas e camadas mais profundas por aspectos mais complexos da imagem. Na figura 3.2 observa-se o um exemplo de uma rede treinada para classificação. Nas imagens, as regiões de mais alta ativação são as cores diferentes de cinza. Na primeira camada (*Layer 1*), as ativações ocorrem em posições relacionadas a aspectos como contornos e texturas. Na quarta camada (*Layer 4*) as ativações acontecem em estruturas mais complicadas, já filtradas para o que se quer identificar.

A medida de desempenho em classificação usualmente ocorre a partir dos erros top-1 e top-5. O erro top- n é a fração das imagens em que as n predições mais prováveis estão incorretas. Isto é, o erro top-1 é a fração das imagens em que a predição mais provável do modelo está errada. Da mesma forma, o erro top-5 é a fração das imagens em que todas das cinco predições mais prováveis são as incorretas. Bases de dados usualmente usadas em medidas de desempenho são CIFAR10[23], CIFAR100[23] e ImageNet [24]. Nas bases CIFAR10 e CIFAR100 reporta-se o erro top-1 e na ImageNet o erro top-5. CIFAR10 é um problema de classificação em imagens de coloridas 32x32 com 10 classes. CIFAR100 é um problema de classificação de imagens 32x32 coloridas em 100 classes. Ambas as bases possuem 50000 imagens para treino e 10000 imagens para teste. Nessas duas bases existem duas categorias: com e sem *data augmentation*. No caso sem *data augmentation* as únicas imagens

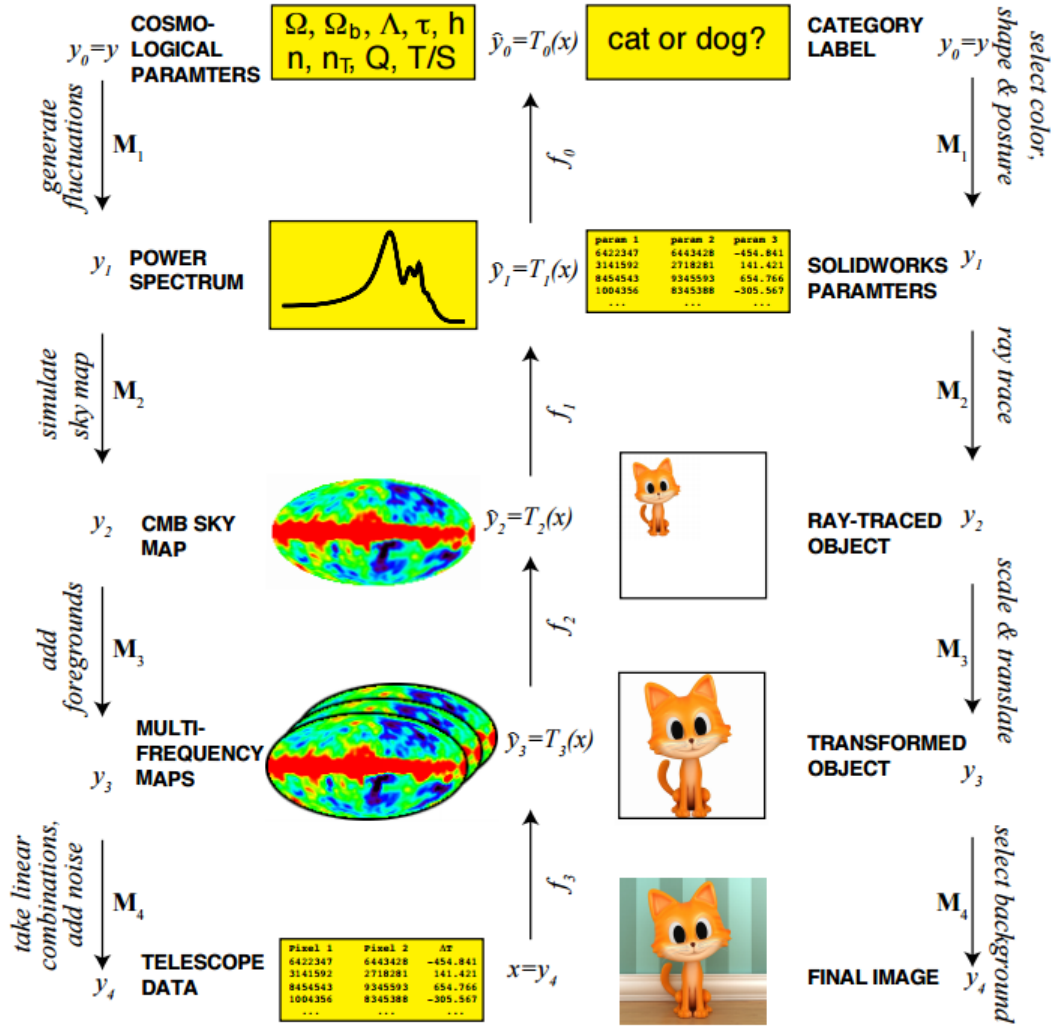


Figura 3.1: A composição de processos gera a observação de forma hierárquica (extraído de Lin et al[6]).

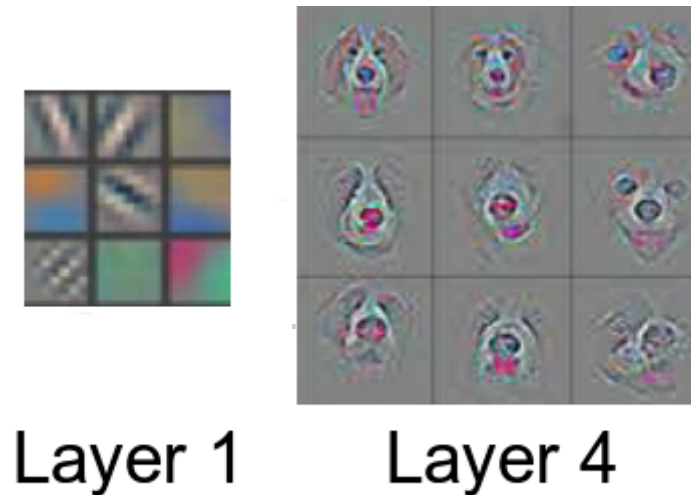


Figura 3.2: Ativações de uma rede neural convolucional - Camadas profundas são associadas à estruturas mais complexas (extraído de Zeiler et al[7]).

usadas para treino são as já existentes na base. Com *data augmentation*, são aplicadas técnicas para se alterar algumas imagens de um *minibatch* mantendo-se o rótulo com o fim de se simular uma base de dados maior. Exemplos de técnicas são a deslocamentos horizontais e verticais da imagem, e seu giro horizontal. A medida de erro em ambos os casos é o top-1. ImageNet é uma base de dados com 14 milhões de imagens de alta definição. Dessas, por volta de 1 milhão possui um rótulo dentre um total de 1000 possíveis classes. Devido ao grande número de classes, espera-se que ocorra ambiguidade em rótulos devido à maior possibilidade de sobreposição [45]. Imagens podem, por exemplo, possuir elementos de duas classes simultaneamente. Por isso, a medida de erro top-5 pode ser mais correta ao definir o desempenho nesse caso. Na competição *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC), reporta-se duas medidas de erro, o erro top-1 e o erro top-5.

3.2 VGG

Após o sucesso da AlexNet, novas redes convolucionais surgiram a fim de superar seu resultado na ILSVRC, uma delas foi a VGG[46]. A VGG foi uma rede desen-

volvida pelo *Visual Geometry Group* da universidade de Oxford. Seu modelo foi desenvolvido em diversos formatos diferentes um deles na sua versão com 16 camadas. Sua arquitetura inicialmente converte a imagem de entrada para um tamanho pré-definido 224x224. Na versão VGG16, seguem-se 5 blocos convolucionais, em que $Conv(a, b \times b, c)$ representa a convolução que gera a *feature maps*, com filtros tamanho $b \times b \times n$ (n o número de *feature maps* da camada anterior) e c o conjunto de *feature maps* por onde serão passados os filtros. ReLU representa a função *Rectified Linear Unit* (figura 2.12) e $MaxPool(a \times b, c)$ representa um *MaxPool* (figure 2.5) de tamanho $a \times b$:

Primeiro bloco:

$$h_1 = ReLU(Conv(64, 3 \times 3, x))$$

$$h_2 = ReLU(Conv(64, 3 \times 3, h_1))$$

$$\hat{h}_2 = MaxPool(2 \times 2, h_2)$$

Segundo bloco:

$$h_3 = ReLU(Conv(128, 3 \times 3, \hat{h}_2))$$

$$h_4 = ReLU(Conv(128, 3 \times 3, h_3))$$

$$\hat{h}_4 = MaxPool(2 \times 2, h_4)$$

Terceiro bloco:

$$h_5 = ReLU(Conv(256, 3 \times 3, \hat{h}_4))$$

$$h_6 = ReLU(Conv(256, 3 \times 3, h_5))$$

$$h_7 = ReLU(Conv(256, 3 \times 3, h_6))$$

$$\hat{h}_7 = MaxPool(2 \times 2, h_7)$$

Quarto bloco:

$$\begin{aligned} h_8 &= ReLU \left(Conv \left(512, 3 \times 3, \hat{h}_7 \right) \right) \\ h_9 &= ReLU \left(Conv \left(512, 3 \times 3, h_8 \right) \right) \\ h_{10} &= ReLU \left(Conv \left(512, 3 \times 3, h_9 \right) \right) \\ \hat{h}_{10} &= MaxPool \left(2 \times 2, \hat{h}_{10} \right) \end{aligned}$$

Quinto bloco:

$$\begin{aligned} h_{11} &= ReLU \left(Conv \left(512, 3 \times 3, \hat{h}_{10} \right) \right) \\ h_{12} &= ReLU \left(Conv \left(512, 3 \times 3, h_{11} \right) \right) \\ h_{13} &= ReLU \left(Conv \left(512, 3 \times 3, h_{12} \right) \right) \\ \hat{h}_{13} &= MaxPool \left(2 \times 2, \hat{h}_{12} \right) \end{aligned}$$

Cada *Pooling*($a \times b$) reduz o número de pixels de cada *feature maps* pelo produto ab . Como existem cinco blocos $MaxPool(2 \times 2)$, os *feature maps* da última camada são reduzidos por um fator 32 em cada dimensão resultando num tamanho 7×7 . Como a última convolução h_{13} gera 512 *feature maps*, o número de termos na última camada é $512 \times 7 \times 7 = 25088$. Por fim, os últimos *feature maps* são redimensionados num vetor (função *Flatten*) e são lançados em camadas classificadoras. Nelas o vetor será multiplicado por matrizes seguidos de ativação até a última camada de normalização *softmax*:

$$\begin{aligned} \hat{h}_{13} &= Flatten \left(\hat{h}_{13} \right) \\ h_{14} &= ReLU \left(\hat{h}_{13} W_1 \right) \\ h_{15} &= ReLU \left(\hat{h}_{14} W_2 \right) \\ h_{16} &= ReLU \left(\hat{h}_{15} W_3 \right) \\ \hat{h}_{16} &= Softmax(h_{16}) \end{aligned}$$

Na VGG16 as matrizes de peso W_1 e W_2 possuem 4096 colunas cada. Por fim, a última camada anterior à *softmax* possui uma matriz W_3 com 1000 colunas,

como determinado pelo número de classes. Esse modelo representa uma extensão direta da AlexNet, com a alteração que o número de convoluções entre cada camada de *pooling* foi aumentado. A VGG, assim como a AlexNet, é um protótipo de uma rede convolucional comum, em que um conjunto de convoluções é aplicado seguido das não linearidades e de uma camada de *pooling*. Na saída da última camada de *pooling* os *feature maps* são redimensionados na forma de um vetor e usados para classificação com as últimas camadas tradicionais seguidas da normalização *softmax*. Vale ressaltar que apesar de representarem apenas um total de três camadas, as matrizes de peso representam grande parte do número total de parâmetros. A VGG16 possui cerca de 138 milhões de parâmetros; desses, por volta de 123 milhões residem nas três últimas camadas, ou 89% de todos os parâmetros do modelo. Essa rede fez parte da ILSVRC de 2014 e obteve o primeiro lugar na tarefa de localização e segundo na de classificação, sendo derrotada pela rede da Google GoogLeNet. Seu resultado apresentou um erro de classificação top-1 de 24.7% e top-5 de 7.5%.

3.3 Residual Network

Uma alteração significativa ocorreu com introdução das redes residuais. Antes os modelos eram compostos por um sequenciamento de camadas convolucionais e *pooling*. Com a aplicação do conceito de *shortcut connections* foram criados os blocos residuais que permitiram o treinamento de redes mais profundas[8]. Uma *shortcut connection* é uma conexão no modelo entre a entrada de um bloco que realiza uma computação e a sua saída. Nas redes residuais essa conexão realiza uma soma entre a entrada e a saída (Figura 3.3). Essa alteração causou duas mudanças fundamentais: facilitou-se que uma camada aprendesse a identidade e simplificou-se o processamento de cada camada. A função aplicada $H(x)$ por uma camada é:

$$H(x) := F(x) + x$$

Nessa configuração, para que uma rede implemente a identidade basta que todos os parâmetros na função $F(x)$ convirjam para zero, considerado mais simples do que convergência para identidade diretamente. Assim, a informação de uma camada anterior poderia fluir para camadas subsequentes mais facilmente. Um

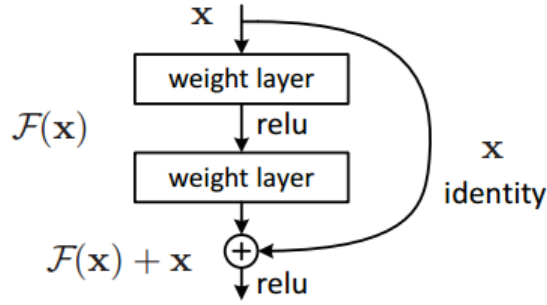


Figura 3.3: Um bloco residual com uma *shortcut connection* entre a entrada x e a saída (extraído de He et al [8]).

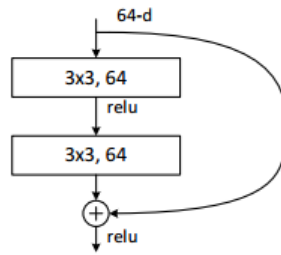


Figura 3.4: Bloco residual convolucional (extraído de He et al [8]).

segundo fato apresentado[8] a favor de um possível melhor funcionamento da rede estaria numa maior simplicidade na informação que deve ser aprendida pela rede. Como os parâmetros do modelo são definidos da função F , toma-se a hipótese de que seria mais simples que a camada se ajustasse à diferença entre a entrada e a saída (ou resíduo) do que diretamente à função H . Rearranjando-se a equação que define H , a função F deve aprender o mapa:

$$F(x) = H(x) - x$$

No contexto de redes convolucionais, uma primeira versão do bloco residual básico foi construído da mesma forma, substituindo-se as matrizes de peso (denotadas por *weight layer* na figura 3.3) por convoluções, como visto na figura 3.4. A adição desse tipo de bloco permitiu pela primeira vez a otimização de redes mais

profundas, entre 10 e 1000 camadas [8]. Uma comparação apresentada demonstrou que anterior ao uso de blocos residuais a adição de camadas poderia ser prejudicial à otimização do modelo. No exemplo, redes com duas profundidades foram treinadas em duas configurações diferentes: com e sem *shortcut connections*, ou seja, duas redes convolucionais tradicionais foram comparadas com duas redes convolucionais residuais. O resultado na Figura 3.5 à esquerda representa as duas redes convolucionais tradicionais e o resultado à direita duas redes residuais. Compara-se as redes com o mesmo número de parâmetros: *plain-18* com *ResNet-18* e *plain-34* com *ResNet-34*. Observa-se no gráfico da esquerda que a rede com 34 camadas apresentou um erro superior à de 18 camadas. Demonstra-se, então, que a simples adição de camadas pode ser negativo para a otimização. Isso pode ser percebido já que uma rede com mais camadas tem a capacidade de representar a mesma função de uma rede menos camadas: basta que as camadas extras se aproximem da identidade. Tal situação faria com que o modelo mais profundo aplicasse exatamente a mesma função que o modelo mais raso e por isso ambos teriam o mesmo desempenho. Entretanto, no exemplo, o modelo de 34 camadas apresenta resultado inferior, o que mostra que o processo de otimização pode ser mais crucial para o desempenho do que a capacidade geral de representação da rede. Nas redes residuais à direita, no entanto, a rede mais profunda foi capaz de obter um erro de teste menor, como seria o esperado. Conclui-se, então, que a adição de *shortcut connections* foi de fato capaz de melhorar a otimização do modelo mais profundo. Um detalhe a se notar nos gráficos da figura 3.5 é o fato do erro de treinamento das redes *plain-18* e *plain-34* serem superiores e mais ruidosos que o erro de teste, enquanto o oposto é o esperado, fato não elucidado no artigo.

Contudo, apesar de permitir a otimização de modelos mais profundos, a adição de camadas extras ainda pode ser prejudicial à otimização. Um experimento com redes de mais de 1000 camadas residuais foi realizado comparando-se com redes mais rasas residuais e com redes mais rasas convolucionais tradicionais. Nele, observou-se que redes residuais extremamente profundas ainda sofrem de problemas de otimização. São testadas redes convolucionais tradicionais com profundidades variando entre 20 e 56, redes residuais entre 20 e 110 e uma rede residual de 1202 camadas.

O resultado do treinamento é ilustrado na figura 3.6. A rede convolucional

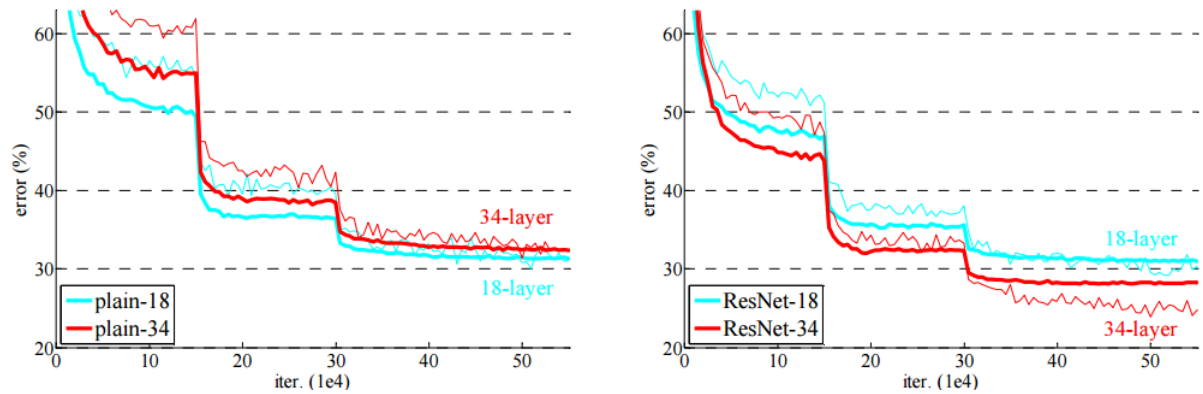


Figura 3.5: Comparação entre duas redes convolucionais tradicionais (esquerda) e duas redes convolucionais residuais (direita). Curvas finas representam erro de treino e curvas em negrito a validação (extraído de He et al [8]).

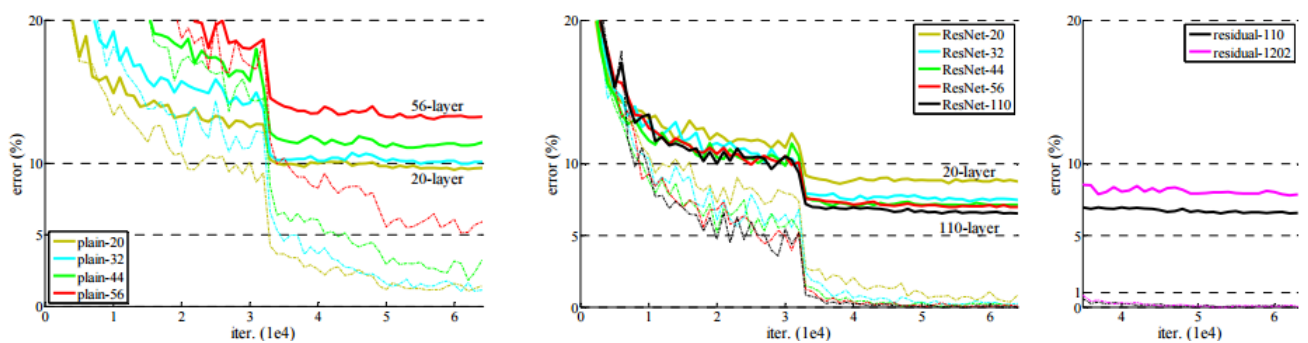


Figura 3.6: Comparação entre redes convolucionais tradicionais (esquerda) e redes convolucionais residuais (meio) e uma rede convolucional residual extremamente profunda (direita) na CIFAR10. Curvas finas representam erro de treino e curvas em negrito a validação (extraído de He et al [8]).

tradicional que apresentou melhor resultado foi a de 20 camadas. A adição de camadas foi prejudicial à otimização como pode ser observado, já que redes com maior número de camadas produziram resultados piores. No caso das redes residuais entre 20 e 110 camadas, a rede com maior número de camadas possuiu o melhor resultado, próximo a 6% de erro. A rede de 1202 camadas possuiu um erro de 7.61%, superior à de 110, o que comprova que o problema da otimização de redes com esse tipo de profundidade não foi resolvido. Contudo, essa rede ainda foi capaz de assumir um erro que foi por volta da metade da melhor rede convolucional tradicional. Isso é um indicativo de que, apesar de não solucionar completamente os problemas de otimização de redes mais profundas, *shortcut connections* certamente são benéficas ao treinamento.

A percepção de que *shortcut connections* permitem melhor otimização gerou um novo modelo de bloco residual. Nele, a entrada do bloco é adicionada a uma informação processada sem que haja uma ativação anterior à saída[9]. No modelo original (Figura 3.4), a saída do bloco é processada por uma função de ativação o que resulta na saída. Nessa nova proposição, a entrada do bloco é adicionada à saída sem qualquer alteração.

A figura 3.7 mostra o treinamento de duas redes de mais de 1000 camadas na CIFAR10. O novo modelo proposto apresentou um melhor resultado que o original sendo capaz de possuir um erro inferior à melhor rede com bloco residual original, que possuía 110 camadas. Essa foi outra indicação de que a presença conexões que realizam operações de identidade a podem ser essenciais à otimização de redes profundas.

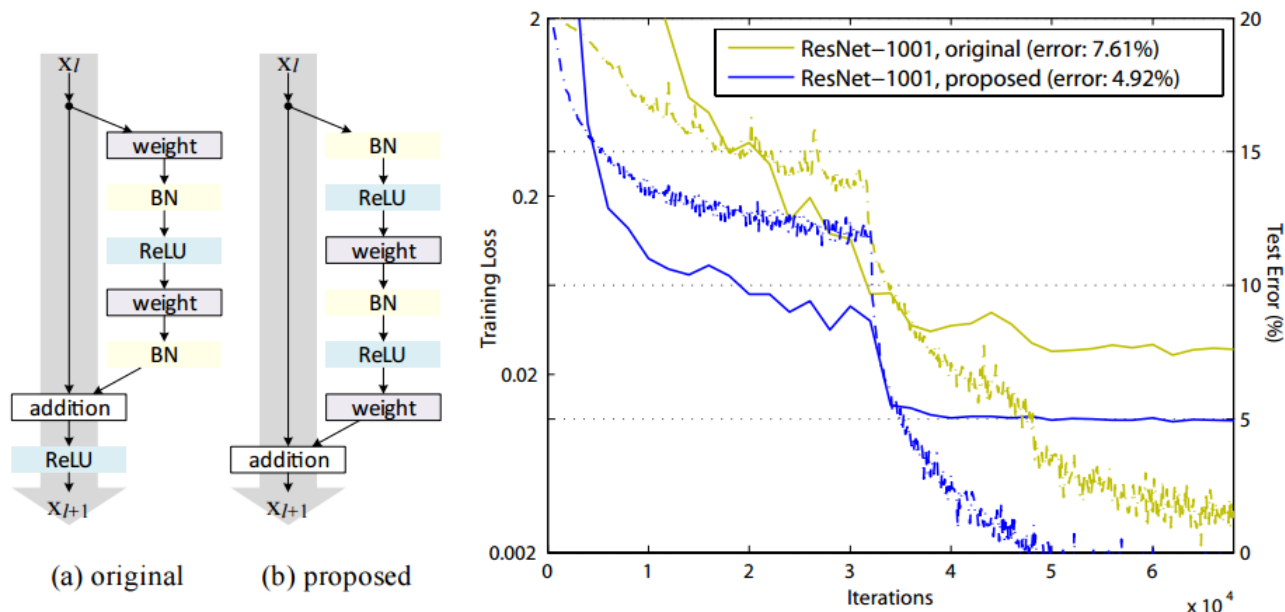


Figura 3.7: Comparação entre duas redes de 1001 camadas na CIFAR10. Uma com o bloco residual original e outra com o novo bloco proposto. Curvas finas representam erro de treino e curvas em negrito a validação (extraído de He et al [9]).

3.4 Rede Neural Convolutacional Densamente Conectada

Redes neurais convolucionais densamente conectadas se baseiam em *shortcut connections*. Diferentemente de redes residuais, as redes convolucionais densamente conectadas não somam a entrada diretamente com a saída. Em vez disso, aplica-se uma concatenação. Duas partes principais as definem: o bloco denso e o bloco de transição. Em um bloco denso, cada camada recebe a saída de todas as camadas convolucionais anteriores. Assim, se L camadas anteriores gerassem um número de *feature maps* K_i cada uma, uma camada seguinte $L + 1$ receberia uma concatenação de $\sum_{i=1}^L K_i$ *feature maps*. Em sua definição original, o número de *feature maps* gerado de camadas convolucionais dentro de um mesmo bloco denso é fixo K . Assim, o número de *feature maps* recebidos em uma camada é o número de *feature maps* da entrada, somados à todos os *feature maps* gerados em anteriormente:

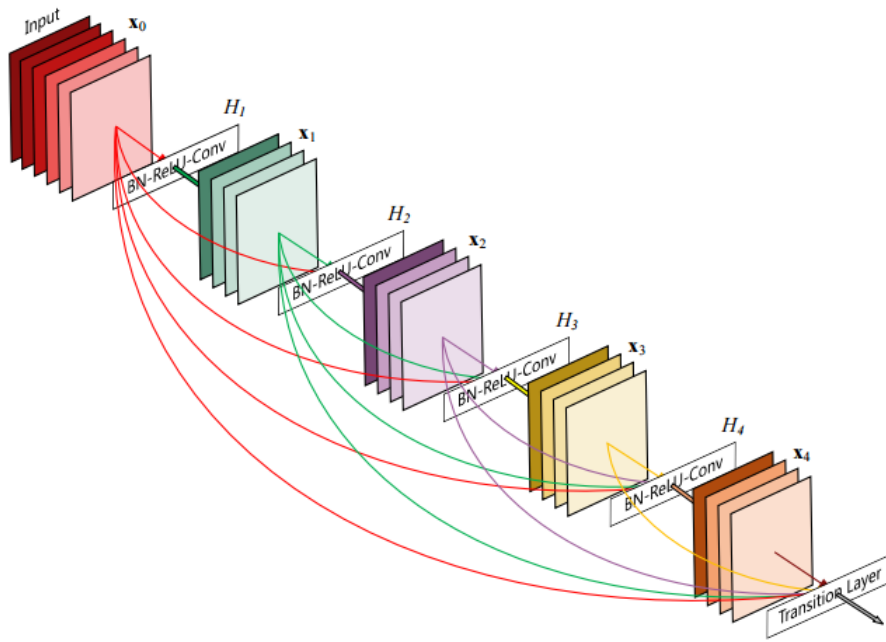


Figura 3.8: Bloco Denso de 5 camadas e taxa de crescimento 4 (extraído de Huang et al [10]).

$$K_1 + \sum_{i=2}^L K = K_1 + K \times (L - 1)$$

Em que K_1 é o número de *feature maps* na entrada. O valor K define a taxa com que o número de *feature maps* concatenados aumenta a cada camada convolucional, por isso ele é chamado de taxa de crescimento do bloco denso. A figura 3.8 ilustra um bloco denso com 5 camadas e taxa de crescimento 4. Observa-se que apesar do número de *feature maps* recebido por cada camada aumentar linearmente, o número gerado por cada uma delas é fixo, definido pela taxa de crescimento 4.

Apesar disso, o número total de *feature maps* recebidos por todas as camadas de um bloco denso aumenta quadraticamente com o número de camadas do bloco, dado que:

$$N_{featMaps} = \sum_{l=1}^L K_1 + K(l - 1) = LK_1 + \frac{L(L - 1)K}{2}$$

Com L igual ao número total de camadas do bloco. Esse valor é importante porque implica que consumo de memória de um programa que implementa essa rede também pode aumentar nesse mesma taxa. Por isso, introduz-se a camada de transição. A partir dela, todas as conexões anteriores ao bloco não se conectam mais a nenhuma camada posterior. Isso limita o aumento quadrático de conexões para o número de camadas dentro de um mesmo bloco denso. A configuração de uma rede convolucional densamente conectada é a intercalação de blocos densamente conectados e transições. Nota-se a partir da figura 3.8, entretanto, que apesar da saída de cada bloco convolucional apresentar um número de *feature maps* independente do número de camadas do bloco, a saída na transição é uma concatenação de todas as saídas anteriores. Isto é, seu número de *feature maps* aumenta quadraticamente com o número de camadas do bloco denso anterior.

A estrutura geral da rede neural densamente conectada aplicada à base de dados CIFAR10 incluiu uma convolução inicial que expande o número de *feature maps* da entrada. Em seguida, a saída dessa convolução se torna a entrada de um bloco denso. A saída do bloco denso é a concatenação de todas as saídas de suas convoluções internas e de sua entrada. Num próximo passo, é feita uma convolução e um *pooling*. Nessa convolução optou-se pelo design em que fosse mantido o número de *feature maps* na sua entrada para saída. Como consequência, o número de *feature maps* na entrada de um bloco denso seguinte aumenta quadraticamente com o número de camadas de um bloco denso anterior. Esse efeito torna custosa a adição de blocos densos, mas foi diminuído com uma alteração posterior no modelo em que se reduz o número de *feature maps* na transição[10]. Esse processo se repete até o último bloco denso. Na sua saída, entretanto, não é feita convolução, ocorre somente uma ativação ReLU seguida de *pooling* e uma camada de pesos tradicional que termina em uma *softmax*.

Em detalhes, cada convolução dentro de um bloco denso ocorre com uma sequência de *Batch Normalization*, ativação, convolução e *Dropout*:

$$C_i = Dropout(Conv(ReLU(BN(x))))$$

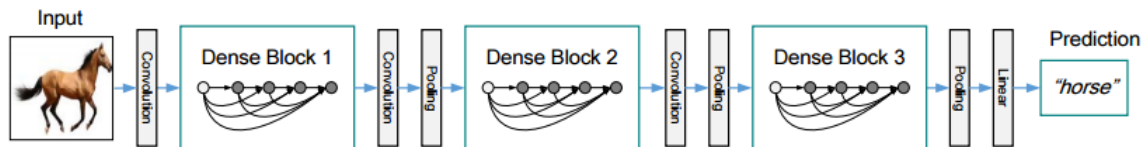


Figura 3.9: Estrutura de uma rede convolucional densamente conectada (extraído de Huang et al [10]).

Em cada transição, as convoluções ocorrem da mesma forma, mas terminadas por uma camada de *pooling*:

$$C_{T_i} = \text{AveragePooling}(\text{Dropout}(\text{Conv}(\text{ReLU}(\text{BN}(x))))))$$

Na figura 3.9, visualiza-se a configuração de uma rede convolucional densamente conectada como publicada no artigo. Diferentemente do divulgado, a implementação no Git dos autores não termina em uma camada de *pooling* seguida de uma camada linear (matriz de pesos de redes tradicionais). No código divulgado, entretanto, aplica-se um *Batch Normalization* seguido da ativação ReLU anteriormente ao último *pooling*. Por fim, o último *pooling* é um *Global Average Pooling*, que, sim, segue para a última camada do modelo.

Os resultados da rede neural densamente conectada superaram os anteriores em problemas padrão de classificação como CIFAR10, CIFAR100. Na CIFAR10, os resultados sem *data augmentation* foram de um erro de 5.83% em uma rede de 3 blocos densos e um total de 100 camadas com taxa de crescimento 24. Com *data augmentation*, o erro foi de 3.74%. Na CIFAR100, a mesma rede obteve erros de 23.42% sem *data augmentation* e 19.25% com. Esses resultados podem ser comparados com outras redes na Figura 3.10. Visualiza-se que as DenseNets são capazes de obter resultados melhores em cada uma das medidas de desempenho. Além disso, esses resultados são obtidos com um uso significativamente menor de parâmetros. Observa-se também que nos casos sem *data augmentation* a rede é capaz de obter resultados melhores que os anteriores por uma grande margem. Espera-se, portanto, que a rede neural densamente conectada seja um bom modelo em outras tarefas

Method	Depth	Params	C10	C10+	C100	C100+
Network in Network [22]	-	-	10.41	8.81	35.68	-
All-CNN [31]	-	-	9.08	7.25	-	33.71
Deeply Supervised Net [20]	-	-	9.69	7.97	-	34.57
Highway Network [33]	-	-	-	7.72	-	32.39
FractalNet [17]	21	38.6M	10.18	5.22	35.34	23.30
with Dropout/Drop-path	21	38.6M	7.33	4.60	28.20	23.73
ResNet [11]	110	1.7M	-	6.61	-	-
ResNet (reported by [13])	110	1.7M	13.63	6.41	44.74	27.22
ResNet with Stochastic Depth [13]	110	1.7M	11.66	5.23	37.80	24.58
	1202	10.2M	-	4.91	-	-
Wide ResNet [41]	16	11.0M	-	4.81	-	22.07
	28	36.5M	-	4.17	-	20.50
with Dropout	16	2.7M	-	-	-	-
ResNet (pre-activation) [12]	164	1.7M	11.26*	5.46	35.58*	24.33
	1001	10.2M	10.56*	4.62	33.47*	22.71
DenseNet ($k = 12$)	40	1.0M	7.00	5.24	27.55	24.42
DenseNet ($k = 12$)	100	7.0M	5.77	4.10	23.79	20.20
DenseNet ($k = 24$)	100	27.2M	5.83	3.74	23.42	19.25

Figura 3.10: Resultados[10] de diferentes modelos nas bases de dados CIFAR10 e CIFAR100 com e sem *data augmentation*. ”+” indica o uso de *data augmentation*. Resultados em negrito indicam DenseNets com erros percentuais inferiores aos anteriores. ”*” indicam resultados gerados pelos autores da DenseNet para os quais não foram divulgados resultados originalmente (extraído de Huang et al [10]).

envolvendo o processamento de imagens.

Capítulo 4

Implementação

4.1 Implementação de redes neurais em Keras

Os experimentos foram realizados em um computador com processador i5-6500 @3.2GHz, 16GB de memória RAM e placa de vídeo GTX-1070 fornecido pelo Laboratório de Instrumentação Biomédica.

A implementação de redes convolucionais ocorre em duas partes. Primeiro, define-se um grafo computacional que representa as operações a serem realizadas pelo modelo. Segundo, esse grafo gera um código que é compilado. Caso o processamento seja feito em CPU o código é gerado em C, caso em GPU o código é gerado em CUDA. Em ambos os casos, retorna-se um objeto que realiza as operações definidas. O procedimento de geração de código e construção do grafo é feito pelo *backend* Theano. A biblioteca Keras fornece funcionalidades que permitem um acesso de mais alto nível e que são mais focadas em aprendizado de máquina. Por exemplo, no caso da implementação de redes neurais tradicionais em Theano é necessário definir o número de linhas de cada matriz de pesos; isso ocorre apesar do fato de que o número de linhas da matriz de uma camada seguinte já ser definida pelo número de colunas da matriz anterior. O mesmo ocorre na definição das dimensões de filtros em camadas convolucionais. No caso de processamento de imagens, um filtro de uma camada convolucional possui 3 dimensões, duas que definem o tamanho do filtro e a profundidade que é definida pelo número de *feature maps* na saída da camada anterior. Na implementação em Keras, as dimensões redundantes são suprimidas, não são explicitamente definidas. Além disso, a biblioteca Keras já possui bases de

dados padrão, funções custo predefinidas, diversos otimizadores e métodos de *data augmentation*.

Um exemplo de uma rede com uma única camada convolucional possui uma camada de entrada representada por $Input(shape = (nFeatMaps, nLinhas, nColunas))$. Caso a entrada seja uma imagem em tons de cinza, $nFeatMaps = 1$, caso seja uma imagem RGB $nFeatMaps = 3$. Como o exemplo só possui camadas convolucionais, não existe real limitação no tamanho da imagem de entrada, diferentemente de redes tradicionais. Nesse caso, não é necessário definir o número de linhas e o número de colunas das imagens de entrada. Para que sejam aceitas imagens de qualquer tamanho, define-se $nLinhas = nColunas = None$. Camadas convolucionais são implementadas pela função *Convolution2D* que recebe como argumento o número de filtros da camada e seus tamanhos em número de linhas e colunas. Também são definidos o modo como a camada convolucional irá tratar da convolução no caso das bordas. O argumento $border_mode = 'same'$ define que serão adicionados pixels extras no contorno do *feature map* de modo que a saída da convolução possua o mesmo número de linhas e colunas da entrada. No caso de uma convolução 3×3 , por exemplo, é adicionado um pixel no contorno da imagem. Existem ainda outros argumentos, como a presença ou não de viés (parâmetro escalar a ser somado em todo *feature map*) e formato da entrada, que não será usado. A composição de camadas é feita a partir de uma API funcional. Nela, cada camada seguinte recebe como entrada a saída da camada anterior no formato:

```
entradaCamada = Input(shape=(nFeatMaps,nLinhas,nColunas))
camada_1 = AplicaOperação(Argumentos) (entradaCamada)
camada_2 = AplicaOperação(Argumentos) (camada_1)
camada_3 = AplicaOperação(Argumentos) (camada_2)
```

No exemplo, a camada de entrada recebe como argumento um formato definido como uma tupla em Python. As camadas seguintes são aplicam uma operação, como *Dropout*, convolução (*Convolution2D*, *Convolution1D*) e *pooling* (*AveragePooling2D*, *AveragePooling1D*). Cada uma dessas operações recebe um conjunto de argumentos, como a probabilidade de *Dropout* ou o tamanho de um filtro convolu-

cional e a sua camada de entrada.

Assim que o grafo da rede é construído, chama-se a função `Model` que constrói o objeto. Seus argumentos são a entrada do modelo *input* e a saída *output*. No exemplo apresentado, a função `Model` para uma rede com entrada *entradaCamada* e saída *camada_3* é:

```
modelo = Model(input=entradaCamada,output=camada_3)
```

Da mesma forma, pode-se escolher arbitrariamente outras camadas de saída ou mesmo duas simultaneamente:

```
modelo = Model(input=entradaCamada,output=[camada_2,camada_3])
```

Esse modelo possui como entrada a mesma entrada do modelo anterior, mas possui duas saídas: a segunda e terceira camadas ocultas. Num exemplo real, um modelo com uma convolução 3×3 com um filtro pode ser implementado como:

```
inp = Input(shape=(1,None,None))
out =
    Convolution2D(1,3,3,border_mode='same',init='normal',bias=False)(inp)
modelo_convolutacional = Model(input=inp, output=out)
```

A camada convolutacional aplica uma convolução 3×3 , seus pesos são inicializados por uma distribuição normal, *border_mode* = "same" garante que a entrada e a saída possuirão o mesmo número de linhas e colunas e *bias* = *False* faz com que não se some um parâmetro escalar a todos os termos da saída.

Exemplifica-se o cálculo de uma saída desse modelo para a entrada:

$$input_mat = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

A matriz de convolução *w* será o impulso discreto escalado por um fator 2, portanto:

$$w = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Como a convolução de uma entrada com o impulso é a própria entrada e a convolução é um operador linear, o resultado esperado da convolução de w com $input_mat$ é $input_mat$ escalado por um fator 2:

$$output = \begin{pmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{pmatrix}$$

Assim, a implementação desta ilustração em Keras é dada pelo código a seguir:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from __future__ import print_function
import numpy as np
np.random.seed(1234)
from keras.layers import Input
from keras.layers.convolutional import Convolution2D
from keras.models import Model
print("Building Model...")
inp = Input(shape=(1, None, None))
output = Convolution2D(1, 3, 3, border_mode='same',
    init='normal', bias=False)(inp)
model_network = Model(input=inp, output=output)
print("Weights before change:")
print (model_network.layers[1].get_weights())
w = np.asarray([
    [[
    [0,0,0],
    [0,2,0],
    [0,0,0]
    ]]
```

```

    ])
input_mat = np.asarray([
    [
    [1.,2.,3.],
    [4.,5.,6.],
    [7.,8.,9.]
    ]
])

model_network.layers[1].set_weights(w)
print("Weights after change:")
print(model_network.layers[1].get_weights())n
print("Input:")
print(input_mat)
print("Output:")
print(model_network.predict(input_mat))

```

Nesse código, o atributo *layers* do objeto retornado por *Model* (*model_network.layers*) armazena as camadas do modelo, *model_network.layers[1]* representa a segunda camada, *Convolution2D* (seguinte à camada *Input*). O método *set_weights* altera os parâmetros da camada para seu argumento de entrada. Portanto, *model_network.layers.set_weights(w)* altera os pesos para o valor *w*. O método *get_weights* retorna os parâmetros de uma dada camada do modelo. Por fim, o cálculo da saída ocorre a partir do método *predict* que recebe como argumento a entrada *input_mat*. No primeiro momento em que esse método é chamado, o grafo computacional é compilado a saída é calculada. Em modelos para regressão, entretanto, o ato de compilar o grafo é feito de forma explícita, já que nesse passo é necessário definir o otimizador, parâmetros externos ao modelo (como a taxa de aprendizado) e a função custo.

A saída do programa inclui os pesos da inicialização por uma distribuição gaussiana, os pesos alterados para o impulso escalado por um fator 2, a entrada do programa e a saída:

```
Building Model...
```

```
Weights before change:
```

```
[array([[[[ 0.02860754,  0.08360302,  0.05814168],
          [ 0.03888381,  0.0131907 , -0.08987349],
          [-0.03580984,  0.01422897, -0.02480469]]]], dtype=float32)]
```

Weights after change:

```
[array([[[[ 0.,  0.,  0.],
          [ 0.,  2.,  0.],
          [ 0.,  0.,  0.]]]], dtype=float32)]
```

Input:

```
[[[[ 1.  2.  3.]
     [ 4.  5.  6.]
     [ 7.  8.  9.]]]]
```

Output:

```
[[[[ 2.  4.  6.]
     [ 8. 10. 12.]
     [14. 16. 18.]]]]
```

Como esperado, a saída possui o mesmo número de linhas e colunas da entrada e corresponde à própria entrada escalada de um fator 2.

Em regressão, adicionam-se dois métodos após a definição do grafo computacional: *compile* e *fit*. O método *compile* recebe três argumentos principais: o método de otimização, a função custo e a métrica durante a validação. Após sua chamada, o modelo do grafo é compilado o que permite que se faça a otimização. Isso é feito pelo método *fit*. Seus argumentos principais são os dados de treino e validação, o tamanho de um *minibatch* e o número de épocas.

Num exemplo de regressão pode-se assumir a existência de uma função subjacente f que recebe uma entrada x e retorna uma saída y (Figura 4.1). Uma observação real, entretanto, está sujeita à ruído. Por isso, uma rede deve ser capaz de aproximar um modelo subjacente a partir de observações \hat{y} : os dados y afetados por um processo Z (Figura 4.2).

No exemplo, a função f é uma sigmoide com um único parâmetro θ :

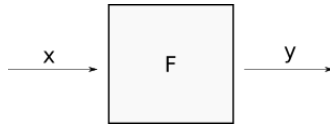


Figura 4.1: Função subjacente F que recebe um x e retorna um y

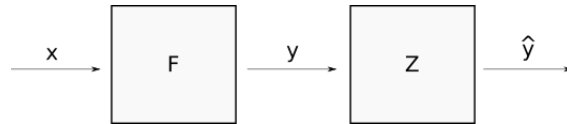


Figura 4.2: Função subjacente F que recebe um x e retorna um y

$$f(x, \theta) = \frac{1}{1 + e^{\theta x}}$$

Para $\theta = -3$, gera-se um conjunto de dados de 300 amostras de entrada x variando uniformemente de -3 a 3 aplicada na função f para gerar a saída y . Em seguida, adiciona-se a y um ruído de distribuição normal de média zero e desvio 0.1 que representa o processo Z e gera a saída \hat{y} . Os dados obtidos são representados na Figura 4.3.

O modelo nesse caso, é somente um parâmetro escalar seguido de uma sigmóide. Sua implementação em keras:

```
inp = Input(shape=(1,))
l = Dense(1)(inp)
out = Activation("sigmoid")(l)
model = Model(input=inp,output=out)
```

Em seguida, o modelo é compilado com o método *compile* e otimizado com o *fit*:

```
model.compile(SGD(lr=0.1, momentum=0.9),
              loss='mean_squared_error',
              metrics=['mean_squared_error'])
```

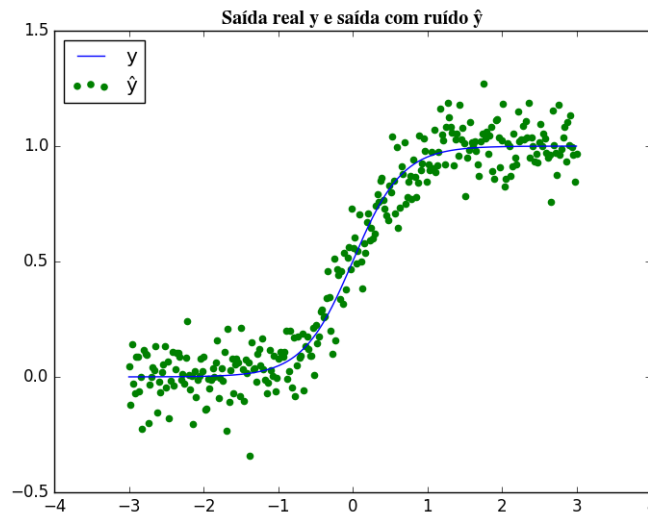


Figura 4.3: Dados para regressão. Círculos representam o dado ruidoso usado em regressão e a linha amostras do modelo original

```

model.fit(X_train, y_train,
          batch_size=batch_size,
          nb_epoch=nb_epoch,
          validation_data=(X_test, y_test),
          shuffle=True)

```

O otimizador usado foi a descida de gradiente estocástica (SGD) com a taxa de aprendizado α igual a 0.1 . Outro parâmetro utilizado foi o *momentum* = 0.9 . Esse parâmetro representa uma variação da descida de gradiente estocástica clássica. Originalmente, os parâmetros do modelo são atualizados a partir da subtração de uma escala gradiente, em que a escala é a taxa de aprendizado. Nessa variação, a atualização do passo atual usa a atualização do passo anterior e o gradiente corrente para alterar os parâmetros do modelo.

$$W_{n+1} = W_n - \alpha \nabla E(W_n) - \beta \alpha \nabla E(W_{n-1})$$

Em que β é a constante de momento igual a 0.9 no exemplo. O treinamento foi feito dividindo-se 90% dos dados para treino e 10% para teste. Foram realizadas um total

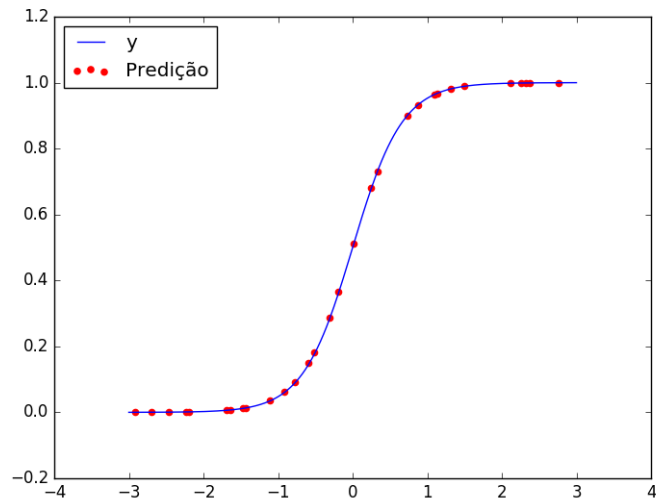


Figura 4.4: Resultado da otimização. Predição dos dados não usados em treino.

de 10 épocas, o erro médio quadrático final na validação foi de 0.0101. O modelo final apresentou um θ de -2.97. O resultado pode ser observado na Figura 4.4. Os dados ilustrados representam os amostras usadas para validação, não utilizadas para treino. Apesar da presença de ruído, o modelo foi capaz de se aproximar razoavelmente do modelo original.

Por se tratar de um problema simples, o modelo rapidamente converge. As curvas do custo aos longo das épocas dos conjuntos de validação e treino são ilustrados na 4.5. Observa-se que em apenas uma época o custo de treino cai abruptamente e satura.

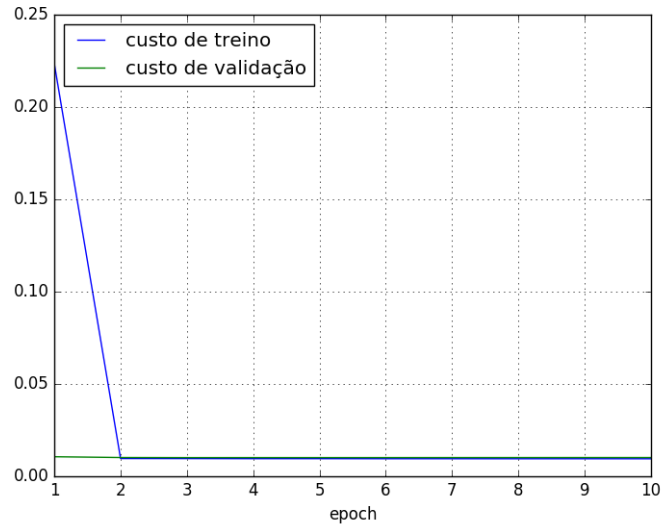


Figura 4.5: Curvas dos custos de treino e validação

4.2 Rede Neural Convolutacional Densamente Conectada

Inicialmente é feito um pré-processamento em toda base de imagens. Calcula-se a média e o desvio padrão de todos os pixels de cada canal do conjunto de treino. O resultado é um total de 3 médias R_μ, G_μ e B_μ e 3 desvios R_σ, G_σ e B_σ , uma média e um desvio para cada canal RGB. Nas imagens, tanto de treino como validação, subtrai-se de cada canal o valor correspondente de média e em seguida divide-se pelo desvio desse mesmo canal. Um pixel i de cada imagem é pré-processado, portanto, como:

$$R_{preproc_i} = \frac{(R_i - R_\mu)}{R_\sigma}$$

$$G_{preproc_i} = \frac{(G_i - G_\mu)}{G_\sigma}$$

$$B_{preproc_i} = \frac{(B_i - B_\mu)}{B_\sigma}$$

Os dados são, em seguida, enviados à rede para classificação. O modelo foi desenvolvido em 4 partes: a entrada, o bloco densamente conectada, a transição e a

saída. Na entrada, aplica-se uma primeira convolução 3×3 com 16 filtros. A saída de 16 *feature maps* é então enviada ao bloco denso. Seus argumentos de entrada são a última saída da rede *net*, uma lista com o número de filtros por convolução *feature_map_n_list*, o somatório dos filtros aplicados *n_filter*, a probabilidade de *Dropout* *droprate* e a norma l2 *weight_decay*.

```
denseBlock_layout(net,feature_map_n_list,n_filter,droprate=0.,weight_decay=1e-4)
```

Como o modelo sempre utiliza o mesmo número de filtros por convolução dentro de um bloco denso, *feature_map_n_list* será uma lista de inteiros iguais. O argumento *n_filter* é acrescido pelo número de filtros de cada convolução. Como cada camada recebe uma concatenação de todos os *feature maps* anteriores (cada um gerado por um filtro), esse argumento contém o número de *feature maps* na entrada de uma camada. Tal argumento é necessário para que se possa escolher o número filtros na camada convolucional após do bloco denso como função do número de *feature maps* na sua saída. A saída de cada camada convolucional é armazenada em uma lista *layer_list* e concatenada com a camada *merge* do *keras* em modo *concat*.

```
net = merge(layer_list , mode='concat',concat_axis=1)
```

Dado que cada saída de uma camada consiste em 4 eixos (amostra,*feature map*,linha,coluna), o eixo em que ocorre a concatenação é o segundo, indexado pelo valor 1, portanto *concat_axis* é igual a 1. Após a concatenação ocorrem as operações de *batch normalization*, *ReLU*, convolução e *Dropout*. O elemento básico de um bloco denso é ilustrado na Figura 4.6. A entrada *merge* recebe as saídas de todas as convoluções anteriores e gera uma saída com o número de *feature maps* fixo na taxa de crescimento da rede (no caso, 12).

Na transição são aplicadas 5 operações: *Batch Normalization*, ativação *ReLU*, camada convolucional, *Dropout* e *average pooling*. *Batch Normalization* e *ReLU* são aplicados diretamente. A camada convolucional consiste numa convolução 1×1 que mantém o número de *feature maps* da sua entrada, portanto *n_filter feature maps*. O *Dropout* descrito possui uma probabilidade de 0.2 e é seguido de um *average pooling*

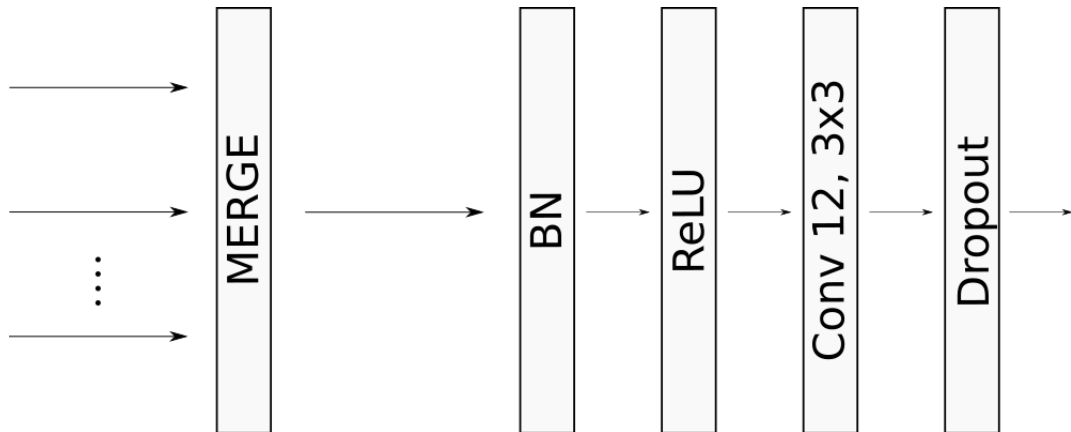


Figura 4.6: Elemento básico de um bloco denso.

(2, 2) que diminui o tamanho da imagem por um fator de 2 no número de linhas e colunas.

Na saída ocorrem 5 operações: *Batch Normalization*, ativação *ReLU*, *Global Average Pooling*, uma camada linear tradicional e finalmente a *softmax*.

O treinamento foi feito num total de 300 épocas, assim como o proposto. A taxa de aprendizado α foi inicializada em 0.1 e alterada para 0.01 após a época 150 e para 0.001 após a época 225. Para tal, usou-se o argumento *callbacks* do método *fit* com um objeto da classe *LearningRateScheduler*. Esse objeto recebe uma função *lr_schedule* que recebe a época atual e retorna a taxa de aprendizado desejada.

```
def lr_schedule(epoch):
    if epoch < 150: rate = 0.1
    elif epoch < 225: rate = 0.01
    elif epoch < 300: rate = 0.001
    else: rate = 0.0001
    print (rate)
    return rate

lrate = LearningRateScheduler(lr_schedule)
```

A fim de se armazenar o modelo com menor erro percentual ao longo do treino, também foi usado um objeto da classe *ModelCheckpoint*. Nela, define-se argumentos como o nome do arquivo em que o modelo será salvo e o critério para que ele seja salvo. No caso, o critério foi o menor erro percentual de classificação.

```
model_checkpoint = ModelCheckpoint(filepath, monitor='val_acc',  
    verbose=1, save_best_only=True, save_weights_only=False, mode='auto')
```

Os principais argumentos da classe *ModelCheckpoint* são as entradas *monitor* e *save_best_only*. O primeiro argumento define o critério que definirá o melhor modelo e o segundo que os melhores modelos não serão sobrescritos. Os objetos *lr_scheduler* e *model_checkpoint* foram agrupados em uma lista e passados como o argumento *callbacks* do método *fit*. Dessa forma, foi possível seguir o método de treinamento proposto para a rede e obter o melhor modelo durante o treinamento considerando-se o erro de validação.

Dois grandes desafios na implementação foram a falta de informações detalhadas e os extensos tempos de cada teste. Nos testes iniciais, não foi possível obter resultados comparáveis aos publicados em [10]. Isso se deveu ao fato que parte da estrutura da rede foi omitida no artigo. Particularmente a presença de *batch normalization* nas camadas de transição foi fundamental para o funcionamento. Apesar disso, a descrição original não menciona seu uso. Somente após busca no código fonte dos autores foi possível se aproximar dos resultados publicados. Outro grande desafio se deveu à limitação de recursos computacionais. Sem o uso de GPU, o treino completo duraria aproximadamente 1460 horas. O processamento em placa de vídeo, por sua vez, reduziu o tempo total por um fator de 77, o que a tornou indispensável para a viabilidade do projeto. Mesmo assim, o treinamento completo de cada versão do modelo durou cerca de 19 horas, o que penaliza amplamente erros de implementação. Erros de difícil percepção como posição incorreta de alguma camada causam mal funcionamento e exigem o início de novos testes. Apesar disso, foi possível obter resultados próximos dos divulgados.

Os testes para validação do modelo ocorreram em duas bases de imagens: CIFAR10 e CIFAR100. Ambas as bases possuem um total de 60000 imagens disponíveis, 50000 para treino e 10000 para teste. Dividiu-se a base de treino em 90% para treino (45000 imagens) e 10% para validação (5000 imagens). O teste foi feito com as 10000 imagens restantes. Os resultados são exibidos com e sem *data augmentation*. Sem *data augmentation* as imagens da base de treino são usadas para treino diretamente. No resultado com *data augmentation* substitui-se algumas imagens de treino por suas versões perturbadas. As perturbações aplicadas foram: translação horizontal de 10% do tamanho da imagem, translação vertical de 10% do tamanho da imagem e inversão horizontal de 180°. As translações ocorrem de forma aleatória por uma distribuição uniforme. Sua faixa é do negativo do percentual do tamanho da imagem seu valor positivo. Por exemplo, uma imagem 30×30 com 10% de translação horizontal é transladada aleatoriamente por um valor variando de -3 até 3. A inversão horizontal ocorre com probabilidade de 50%. A imagem final será uma composição de todas as transformações, iniciando-se pelas translações e terminando na inversão horizontal.

4.3 Resultados Parciais

Apresenta-se o número de parâmetros e o erro percentual da reimplementação. O modelo final possuiu o mesmo número de parâmetros de uma das implementações apontadas por Huang et al [47], um total de 1019722. O erro percentual de teste nas bases de dados CIFAR100 e CIFAR10 são visualizados na tabela 4.1. Embora possua o mesmo número de parâmetros, observa-se erros percentuais superiores na reimplementação. Isso pode ter ocorrido devido a diferenças em parâmetros externos ao modelo, como constantes da otimização, diferentes constantes padrão das bibliotecas (a implementação original é feita em Torch [48]) ou diferença na inicialização. Apesar disso, rede ainda possui uma taxa de acerto aceitável, o que mantém a indicação de que sua aplicação em tarefas de processamento de imagens seria positiva.

Erro percentual (%) da DenseNet (L=40,k=12)				
Rede	C10	C10+	C100	C100+
Original	7.00	5.24	27.55	24.42
Reimplementação	7.6	6.45	28.52	25.94

Tabela 4.1: Erro percentual de teste nas bases de dados CIFAR100 (C100) e CIFAR10 (C10). Símbolo ”+” representa resultados com *data augmentation*.

4.4 Remoção de Ruído Gaussiano

Os experimentos foram realizados em quatro etapas. Primeiro, gerou-se uma base de dados em escala cinza a partir da CIFAR10. Segundo, adicionou-se a ela ruído gaussiano para gerar uma base de imagens ruidosas. Terceiro, treinou-se uma rede na CIFAR10 cinza para remoção do ruído. Quarto, aplicou-se ruído gaussiano em outras imagens, seguido de filtragem e comparou-se o resultado com o de outro procedimento de remoção de ruído.

A base de imagens cinza foi gerada a partir de imagens RGB da CIFAR10. Considerando-se cada componente R, G e B de cada imagem uma matriz, a imagem cinza resultante Y foi calculada a partir da luminância NTSC (o mesmo método do MATLAB da função `rgb2gray` usada em imagens dos artigos em futura comparação):

$$Y = 0.2989 \times R + 0.5870 \times G + 0.1140 \times B$$

A geração da base com ruído foi feita a partir da adição de ruído aleatório gaussiano de média zero e desvio padrão 20. A cada imagem Y da base CIFAR10 cinza foi adicionada uma imagem de ruído R_u de mesmas dimensões para gerar sua versão ruidosa Y_r na forma:

$$Y_r = Y + R_u$$

A imagem de ruído gaussiano foi gerada a partir da função `numpy.random.normal` da biblioteca Numpy com semente aleatória zero (`numpy.random.seed(0)`).

O método para a remoção de ruído foi usar a base de imagens cinza ruidosas como entrada da rede e a base de imagens sem ruído como rótulo. Assim, a regressão que minimiza o erro médio quadrático entre a entrada e o rótulo busca remover o ruído que diferencia as duas imagens. Dessa forma, o treinamento ensina a rede como remover o ruído aplicado nas imagens de entrada. Ressalta-se, com isso, que essa estrutura com a imagem original como rótulo e a entrada perturbada poderia ser empregada em diversas outras aplicações. Por exemplo, se a imagem original fosse colorida e a perturbação fosse sua versão em escala cinza, a rede aprenderia a colorir. Caso a entrada fosse a uma versão desfocada da imagem original, a rede aprenderia melhorar o foco. Para uma dada imagem x , uma imagem y perturbada por uma função f pode ser representada como:

$$y = f(x)$$

O objetivo, então, de um modelo ideal g^* seria receber uma imagem y e gerar x novamente:

$$g^*(y) = g^*(f(x)) = x$$

Ou seja, um modelo ideal g^* é igual à função inversa de f :

$$g^*(f(x)) = x \implies g^* = f^{-1}$$

Como, todavia, f pode ser não inversível, busca-se um modelo que torne g mais próximo de f^{-1} para um conjunto relevante de entradas. Nesse escopo, espera-se que para uma entrada perturbada $f(x)$, $g(f(x))$ seja igual a um \hat{x} , próximo de x . Nesse sentido, o processo de regressão deverá alterar os parâmetros do modelo g a fim de minimizar uma medida de distância entre \hat{x} e x . Caso a otimização seja bem sucedida e as entradas durante o treinamento sejam variadas o suficiente, espera-se que o modelo gerado g também seja capaz de se aproximar de f^{-1} para outras imagens não observadas durante o treino. Um modelo g receberá, então, imagens perturbadas por uma função f que se deseja inverter e a saída \hat{x} será comparada com x original. A função E que realiza essa comparação define o custo (Figura 4.7), cujo gradiente será computado como parte da atualização dos parâmetros de g .

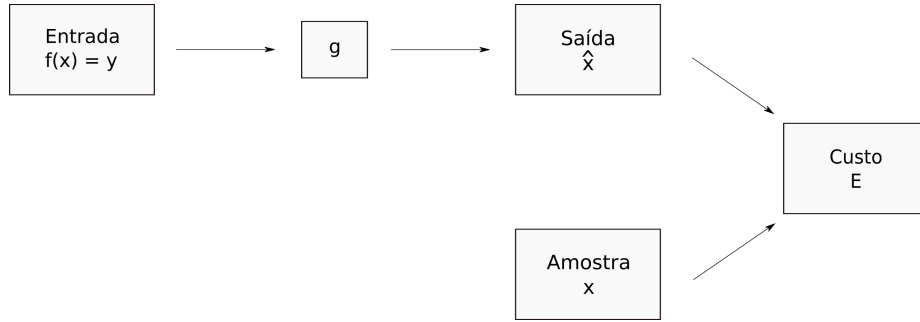


Figura 4.7: Cálculo do custo de uma entrada perturbada y para uma amostra original x .

Nesse sentido, a função f a ser invertida é a adição de ruído gaussiano ($f(Y) = Y + R = Y_r$) e o modelo g uma rede neural convolucional baseada na Rede Neural Convolucional Densamente Conectada. Uma limitação desse método decorre do fato que a adição de ruído gaussiano eventualmente ultrapassa os valores máximo e mínimo da imagem original, com valores superiores a 255 ou inferiores a zero. Isso somente ocorre em imagens em que ruído foi adicionado de forma artificial, o que causa uma diferença entre as imagens usadas em treino e possível uso em imagens externas. No caso do valor de um elemento da imagem na saída da rede não pertencer a essa faixa, ele será saturado superiormente em 255 e inferiormente em zero.

Cada arquitetura foi gerada a partir de uma alteração da arquitetura anterior. Em cada caso, foram testados modelos em diferentes configurações (valores de K e L do bloco denso), quando alterações nos valores de K e L geraram resultados melhores, essas mudanças foram mantidas nos modelos seguintes. Dessa forma, foram testadas 4 arquiteturas (figura 4.8). Os modelos foram treinados em 20 épocas. As normas $L2$ foram de 0.0001 aplicadas em todos os parâmetros. O método de otimização foi uma variação da descida de gradiente estocástico NAdam[49]. Um *Dropout* de probabilidade 0.3% foi aplicado em todas as camadas convolucionais após o bloco denso exceto a última. A função custo usada foi o erro médio quadrático. Apresenta-se o tempo de treinamento t , o número de parâmetros N do modelo e o menor valor de erro médio quadrático MSE. Testou-se também a similaridade SSIM [27] entre a

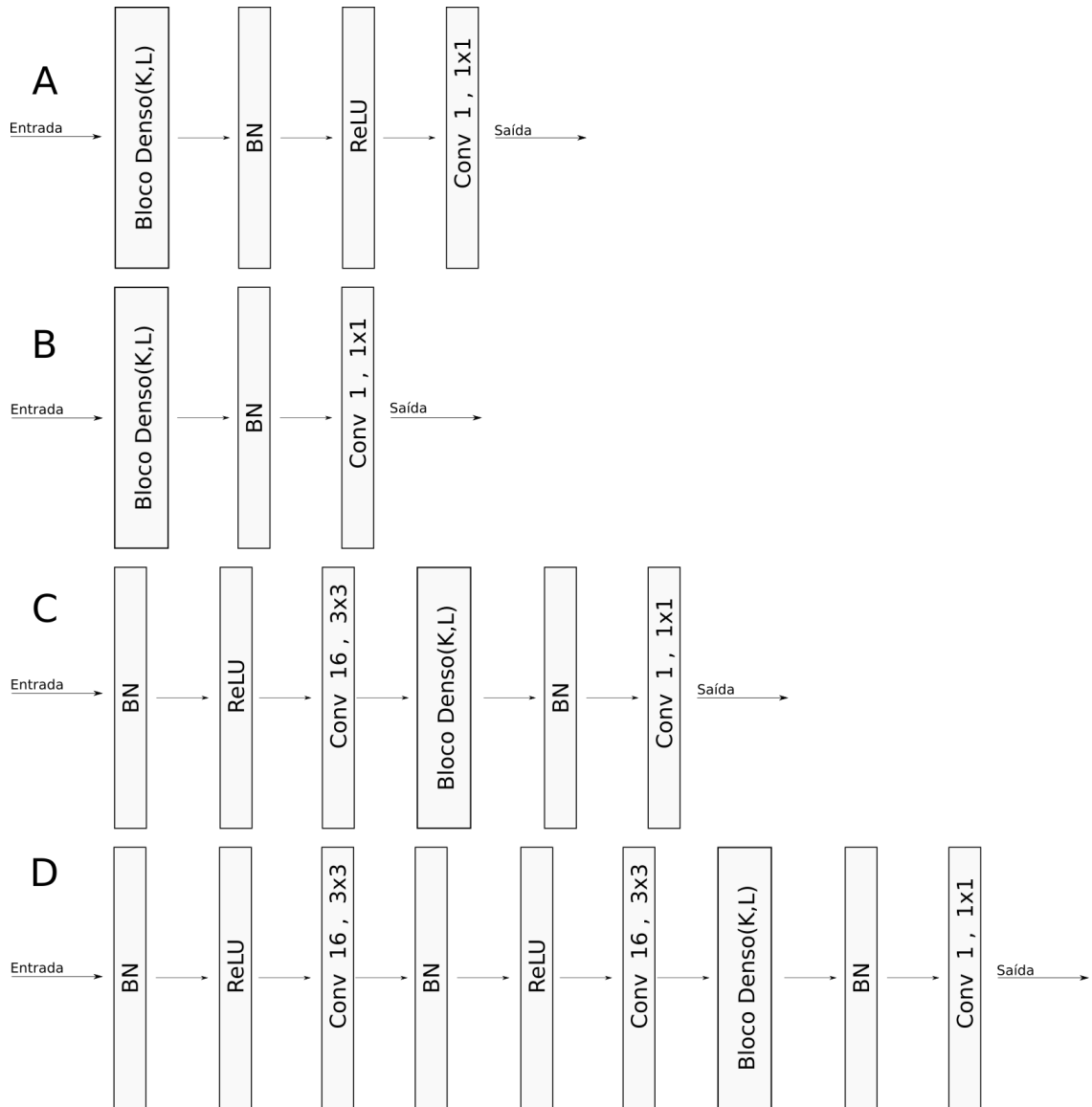


Figura 4.8: Quatro arquiteturas de rede testadas. A arquitetura *A* indica um cálculo de um bloco denso seguido de *batch normalization* (BN), ativação ReLU e uma convolução 1×1 que gera 1 *feature map* em sua saída. O diagrama seguinte *B* apresenta a mesma rede com a ativação ReLU removida. Em sucessão, a rede *C* baseada na *B* com a adição de *batch normalization*, ReLU e convolução na primeira camada. Por fim, a rede *D* similar à *C* com a adição de *batch normalization*, ReLU e convolução na entrada.

Redes na arquitetura A				
Rede	MSE	SSIM	t(s)	N
K=10 L=10	0.03243	0.83	65	42623
K=12 L=10	0.03136	0.82	74	60863
K=12 L=12	0.03063	0.80	113	88875

Tabela 4.2: Treinamento de redes na arquitetura A. MSE é o menor erro médio quadrático obtido durante o treinamento no conjunto validação da CIFAR10. SSIM é o resultado da comparação da imagem Lena 256×256 cinza original com a imagem com ruído gaussiano de média zero e variância 400 filtrada pela rede avaliada pelo indicador *Structural Similarity Index*, quanto mais próximo de 1 melhor. O tempo t é o tempo por época do treinamento de cada rede e N seu número final de parâmetros.

imagem Lena 256×256 em escala cinza ruidosa após a filtragem e a imagem original. A arquitetura A foi a primeira testada em três versões diferentes. Em cada uma foram variados os parâmetros K e L do bloco denso. Observa-se na tabela 4.2 que o aumento dos parâmetros K e L causa um acréscimo no número total de parâmetros e do tempo de execução de cada época. Visualiza-se também, contudo, que apesar de ocorrer uma diminuição no erro médio quadrático observado nos modelos maiores, a similaridade SSIM entre a imagem original e a imagem filtrada não aumenta. Isso ilustra o fato do índice SSIM nem sempre possuir a mesma tendência do erro MSE.

Como método para geração de novos modelos, escolhe-se o modelo que possuiu o melhor SSIM (modelo A com $K=10$ e $L=10$) e faz-se uma alteração: remove-se a última camada ReLU, o que gera o modelo B . Apresenta-se o resultado do treinamento na tabela 4.3. Observa-se que após a remoção da ativação ReLU, uma redução no MSE do modelo com $K = 10$ e $L = 10$, com uma queda no SSIM de 0.01. Curiosamente, o modelo com $K = 8$ e $L = 8$ obteve seu melhor índice SSIM numa das épocas iniciais (época 6), quando seu erro médio quadrático era superior ao das outras redes. Comparativamente, ilustra-se a rede $K = 8$ e $L = 8$ treinada até a última época. Há uma redução de MSE de 0.03310 para 0.02860, mas o SSIM cai

Redes na arquitetura B				
Rede	MSE	SSIM	t(s)	N
K=10 L=10	0.02764	0.82	67	42623
K=8 L=8 Época 6	0.03310	0.84	35	17363
K=8 L=8 Época 20	0.02860	0.79	35	17363
K=4 L=4	0.03180	0.76	8	1115
K=10 L=4	0.03120	0.81	13	6011

Tabela 4.3: Resultados do treinamento de modelos na arquitetura B

de 0.84 para 0.79. Esse resultado pode estar associado ao efeito de espalhamento do erro médio quadrático [50], que diminui o erro médio, mas pode ser responsável pela geração de imagens embaçadas. O modelo com $K = 4$ e $L = 4$ possuiu resultados inferiores, com baixo SSIM e alto MSE. O modelo $K = 10$ e $L = 4$ foi capaz de obter um SSIM de 0.81 com aproximadamente um sétimo do número de parâmetros do primeiro ($K = 10$ e $L = 10$).

A partir dos bons resultados do modelo com $K = 8$ e $L = 8$ na arquitetura B, duas novas arquiteturas foram criadas a partir dela, C e D (figura 4.8). Na C adicionam-se três operações na entrada: *batch normalization*, ReLU e convolução com 16 filtros 3×3 . Na arquitetura D, essas mesmas operações são adicionadas duas vezes em sequência. O resultado dos treinamentos são exibidos na tabela 4.4. A rede na estrutura D com possuiu, com isso, o menor erro médio quadrático e melhor SSIM na filtragem da imagem Lena 256×256 com ruído gaussiano de média zero e variância 400.

Redes com K=8 e L=8				
Estruturas	MSE	SSIM	t(s)	N
C	0.02734	0.82	48	26434
D	0.02718	0.84	51	28770

Tabela 4.4: Resultados do treinamento de redes nas arquiteturas C e D, ambas com $K = 8$ e $L = 8$.

4.5 Resultados e Discussão

Os resultados são apresentados nas três imagens usadas para medida de desempenho como no artigo de Laparra et al [11]: *Barbara*, *Boats* e *Lena*. As imagens foram transformadas para escala cinza pelo mesmo método da função do MATLAB `rgb2gray` e foram redimensionadas das versões originais 512×512 para 256×256 por amostragem impulsiva. As imagens 256×256 foram adicionadas de ruído em dois experimentos: no primeiro, foi adicionado ruído gaussiano de média zero e variância 200, no segundo, foi adicionado ruído gaussiano de média zero e variância 400. Por fim, as imagens são filtradas por diferentes métodos e o resultado da filtragem é avaliado pela raiz quadrada do erro médio quadrático (RMSE) e por SSIM entre as imagens filtradas e as sem ruído.

Os resultados para as imagens com ruído gaussiano de variância 200 para cada modelo de rede são apresentados na figura 4.9. Cada modelo é caracterizado pela letra que define sua arquitetura e o os valores K e L do bloco denso. Por exemplo, *A K10 L10* representa o modelo na arquitetura A com $K=10$ e $L=10$. No caso particular do modelo *B K8 L8*, foram treinados dois modelos de mesma arquitetura com mesmo tamanho do bloco denso. A diferença do treinamento foi o número de épocas treinadas. Para diferenciação, o modelo treinado em 20 épocas foi denominado *B K8 L8 v1* e o treinado em 6 épocas *B K8 L8 v2*. Os modelos que apresentaram melhor SSIM nesse experimento foram os *C K8 L8* e *D K8 L8*, com valores superiores a 0.89 na imagem *Lena*, razoavelmente acima de 0.88 na imagem *Barbara* e pouco acima de 0.87 na imagem *Boats*. O *D K8 L8* possuiu

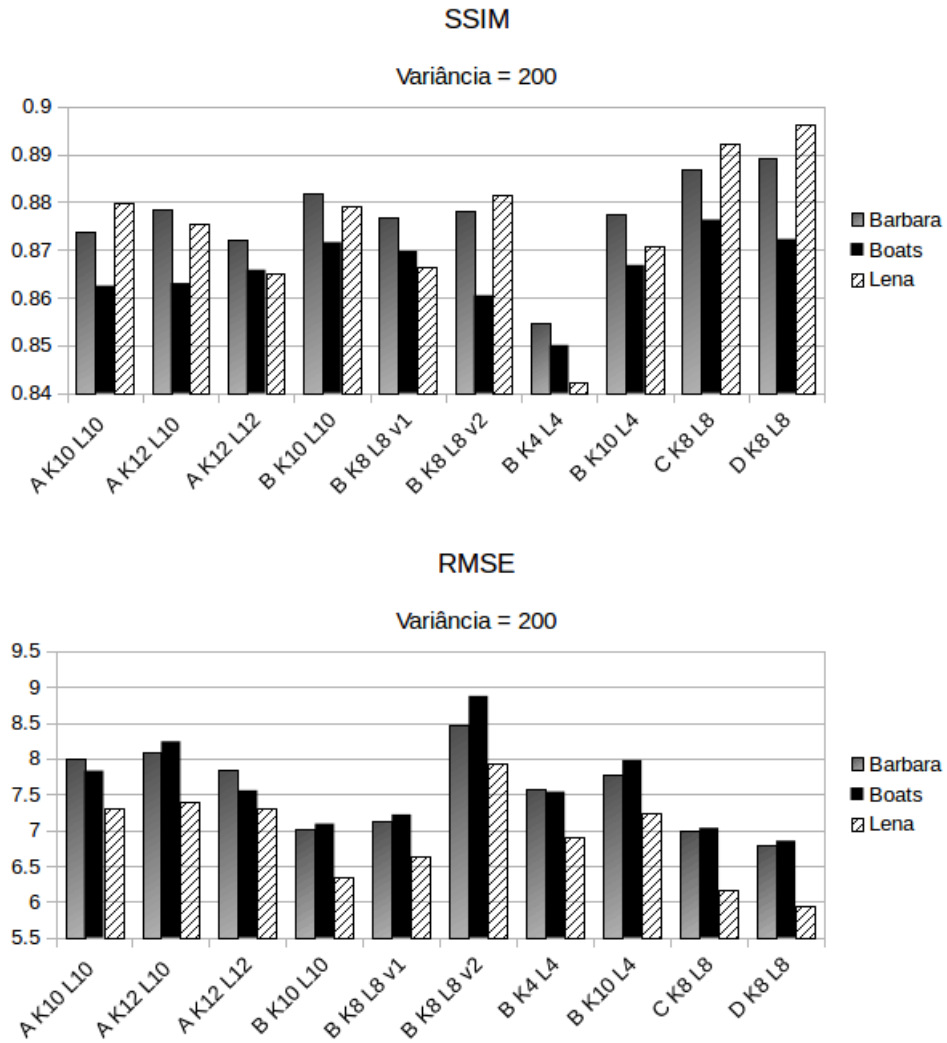


Figura 4.9: Resultados de SSIM (acima) e RMSE (abaixo) para as imagens filtradas por cada modelo de rede para ruído de variância 200. Quanto mais próximo de 1, melhor o SSIM e quanto mais próximo de zero, melhor o RMSE.

Testes com variância 200						
Método	Barbara		Boats		Lena	
	SSIM	RMSE	SSIM	RMSE	SSIM	RMSE
HT	0.77	16.48	0.76	13.62	0.73	18.97
ST	0.78	14.37	0.79	10.26	0.74	12.59
BG	0.80	14.14	0.79	11.70	0.76	12.75
BL	0.81	12.95	0.83	8.30	0.78	11.66
GSM	0.90	8.94	0.87	8.94	0.83	13.61
SVR	0.87	10.11	0.84	10.16	0.81	12.54
SVR OPT	0.87	10.11	0.85	10.36	0.82	12.30
D K8 L8	0.89	6.80	0.87	6.85	0.90	5.95

Tabela 4.5: Resultados[11] de SSIM e RMSE das imagens filtradas pelos métodos Hard Thresholding(HT)[12], Soft Thresholding(ST)[12], Bayesian Gaussian(BG)[13], Bayesian Laplacian(BL)[14], Gaussian Scale Mixture(GSM)[15], Support Vector Regression(SVR)[11] e uma borda superior de desempenho do SVR, o SVR^{opt} . O modelo proposto é denominado $D K8 L8$. Valores em negrito ressaltam os resultados SSIM mais próximos de 1 e menores RMSE em cada imagem.

SSIM ligeiramente superior na imagem *Barbara*, ligeiramente inferior na imagem *Boats* e pouco superior na imagem *Lena* em relação o segundo melhor modelo $C K8 L8$. Na medida RMSE, quatro modelos foram próximos: $B K10 L10$, $B K8 L8 v2$, $C K8 L8$ e $D K8 L8$, mas os dois últimos possuíram os menores erros. Por isso, o modelo $D K8 L8$ foi selecionado para como método para remoção de ruído gaussiano e usado em comparação com outros métodos.

Os valores de SSIM nas tabelas foram arredondados para o valor mais próximo com duas casas decimais pela função `numpy.around` da biblioteca Numpy. Compara-se, então, o resultado do $D K8 L8$ com os métodos descritos em Laparra et al [11] na Tabela 4.5 para ruído com variância 200. Nessa tabela, a rede $D K8 L8$ possuiu o segundo melhor SSIM na imagem *Barbara*, inferior por 0.01 se comparado ao método GSM, o mesmo SSIM na imagem *Boats* e SSIM superior por 0.07 na imagem *Lena*. Em todos os casos a filtragem pela rede possuiu melhor RMSE que os outros métodos.

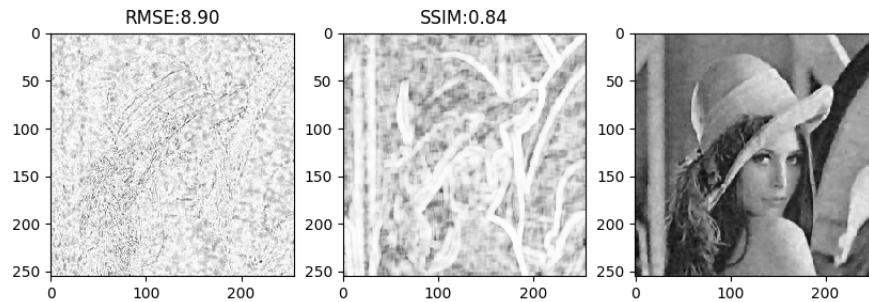
Testes com variância 400						
Método	Barbara		Boats		Lena	
	SSIM	RMSE	SSIM	RMSE	SSIM	RMSE
HT	0.67	24.52	0.68	20.15	0.67	20.22
ST	0.69	19.04	0.71	16.16	0.66	19.72
BG	0.70	20.40	0.70	19.17	0.67	19.26
BL	0.73	16.52	0.77	10.26	0.67	18.45
GSM	0.86	11.02	0.80	17.40	0.79	15.95
SVR	0.83	13.13	0.81	10.73	0.78	14.50
SVR OPT	0.83	13.13	0.81	10.73	0.78	14.06
D K8 L8	0.84	8.43	0.83	8.31	0.84	7.71

Tabela 4.6: Resultados[11] de SSIM e RMSE das imagens filtradas pelos métodos HT, ST, BG, BL, GSM, SVR e pelo modelo proposto *D K8 L8*. Valores em negrito ressaltam os resultados SSIM mais próximos de 1 e menores RMSE em cada imagem.

No experimento em que as imagens de validação foram corrompidas com ruído gaussiano de variância 400, os modelos convolucionais foram novamente testados. Seus resultados são ilustrados na figura 4.11. No índice SSIM dois modelos se destacaram: *B K8 L8* e *D K8 L8*. O primeiro possuiu SSIM acima do segundo nas imagens *Barbara* e *Lena* por uma pequena margem. Na imagem *Boat*, o modelo *D K8 L8* teve um melhor SSIM por um valor inferior a 0.002. Curiosamente, apesar de possuir altos valores SSIM, no critério RMSE o modelo *B K8 L8* possuiu um dos piores resultados dentre todos. Esse fato pode ser observado na figura 4.10 em que a figura do erro absoluto é ilustrada ao lado da figura gerada durante o cálculo SSIM. Comparativamente, o modelo *D K8 L8* obteve os menores valores RMSE. Assim, ele também foi escolhido para comparação com os métodos HT, ST, BG, BL, GSM e SVR.

A comparação dos métodos publicados com a rede convolucional na arquitetura *D K8 L8* é exibida na tabela 4.6. Na imagem *Barbara*, a rede possuiu um SSIM 0.02 inferior ao método GSM. Na imagem *Boat*, obteve-se um resultado 0.02 superior ao segundo melhor método SVR. Na última imagem *Lena*, o resultado de SSIM foi superior ao segundo melhor método GSM por 0.05, que representa um

BK8L8 v2



DK8L8

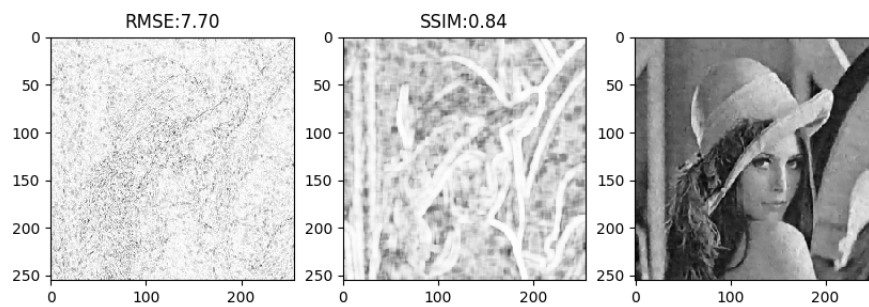


Figura 4.10: Ilustração do erro absoluto(esquerda) e da imagem do SSIM(centro) da imagem *Lena* filtrada por *B K8 L8 v2*(acima) e por *D K8 L8*(abaixo). O erro absoluto da imagem filtrada pelo modelo *B K8 L8* apresenta manchas claramente presentes quando comparado ao erro da imagem filtrada por *D K8 L8* possivelmente responsáveis por seu maior valor RMSE. A imagem gerada durante o cálculo do SSIM não apresenta diferença significativa em cada caso, o que caracteriza o valor próximo observado. Verifica-se que o SSIM corretamente não possui uma alteração grande em seu valor dado que as imagens filtradas(direita) em cada caso não possuem diferenças muito significativas.

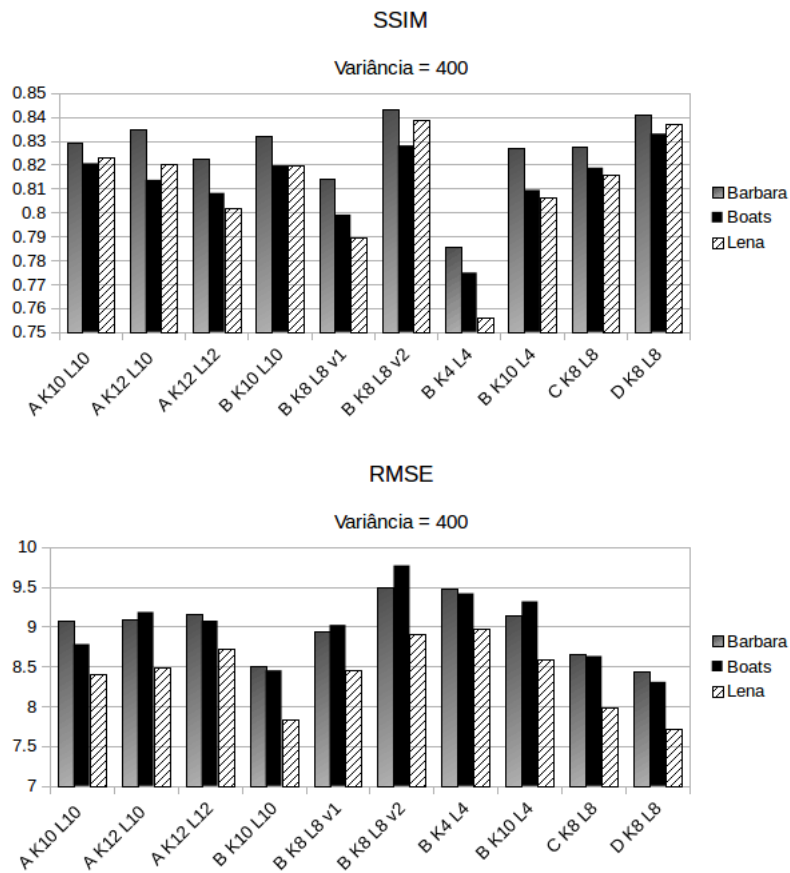


Figura 4.11: Resultados de SSIM (acima) e RMSE (abaixo) para as imagens filtradas por cada modelo de rede para ruído de variância 400. Quanto mais próximo de 1, melhor o SSIM e quanto mais próximo de zero, melhor o RMSE.

D K8 L8 (0.84)



Figura 4.12: Resultado visual da filtragem em uma região da imagem *Lena* para o ruído gaussiano de variância 400. SSIM exibido entre parêntesis. A imagem foi gerada com interpolação de ordem zero.

aumento significativo. Novamente, o valor RMSE das imagens filtradas pela rede *D K8 L8* foi substancialmente inferior aos demais.

Visualmente, pode-se comparar o resultado do modelo proposto na figura 4.12 e dos métodos HT, ST, BG, BL, GSM, SVR na figura 4.13. As imagens do artigo possuem tamanho maior que original, mas não se sabe o método de interpolação aplicado para a geração das imagens maiores, o que pode dificultar uma comparação direta com o resultado da figura 4.12.

Por fim, avalia-se visualmente os resultados de filtragem do modelo *D K8 L8* nas imagens teste para o caso de variância 400 na figura 4.14. Na terceira coluna, o erro absoluto contém as diferenças entre a imagem original e filtrada, informações não recuperadas pelo processamento e artefatos adicionados pela filtragem. Na imagem superior (*Barbara*), observam-se regiões que concentram erro, como nas listras do pano que encobre sua cabeça e no quadriculado do pano sobre mesa. Na imagem do meio (*Boats*), da mesma forma, também possui regiões com maior erro em posições que possuem maiores níveis de detalhe como no mastro e na proximidade da interseção dos barcos. Na imagem inferior (*Lena*), pode-se visualizar que houve a maior presença de erro no contorno e detalhes do chapéu e na região dos olhos. Em todos os casos, as regiões menos recuperadas foram aquelas de maior detalhe, como contornos e arestas.

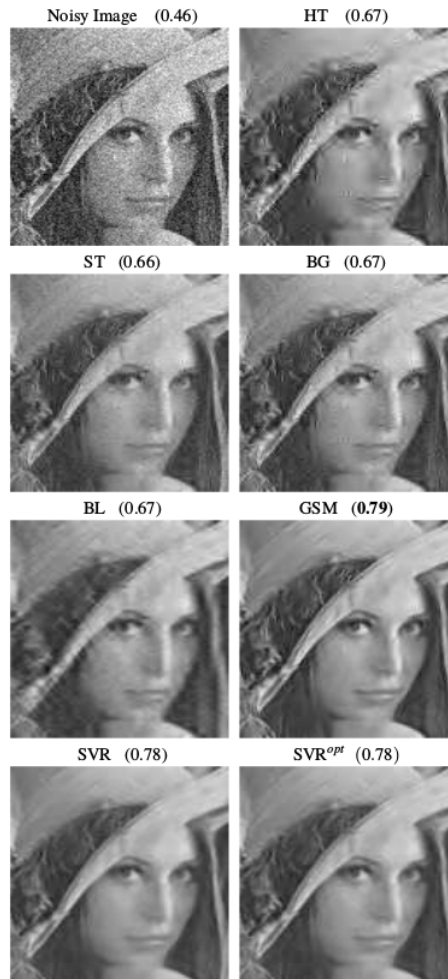


Figura 4.13: Resultados de SSIM (entre parêntesis) na filtragem pelos métodos HT, ST, BG, BL, GSM, SVR para o caso do ruído gaussiano de variância 400 (extraído de laparra et al [11]). A imagem superior esquerda ilustra a imagem ruidosa.

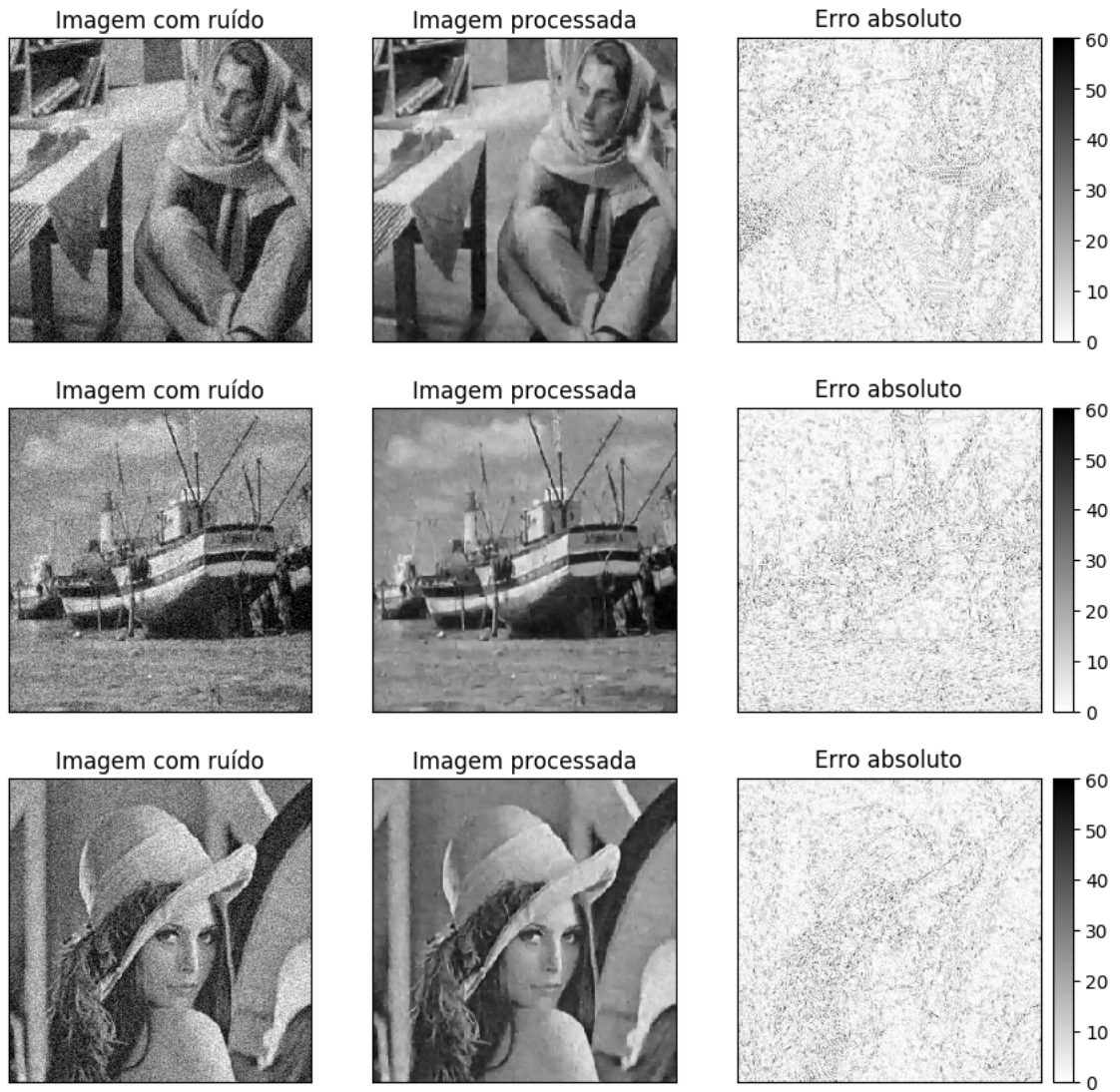


Figura 4.14: Resultados de filtragem das imagens de teste *Barbara* (topo), *Boats* (meio), *Lena* (abaixo). As imagens à esquerda indicam as versões ruidosas, as do meio o resultado após o processamento e as à direita o valor absoluto da diferença entre a imagem filtrada e a original sem ruído.

Em suma, o modelo *D K8 L8* foi capaz de obter resultados satisfatórios na remoção de ruído gaussiano. No caso de ruído com variância 200, foi obtido índice de similaridade igual ao GSM na imagem *Boats* e superior na imagem *Lena* por 0.07. Na imagem (*Barbara*) o SSIM foi inferior por apenas 0.01. Isso mostra a possibilidade de aplicação de uma rede treinada em uma situação, com ruído de variância 400, num outro caso próximo, com nível de ruído diferente (variância 200). Na filtragem de imagens com ruído de variância 400, a rede possuiu um resultado inferior por 0.02 na imagem *Barbara* em relação ao GSM e superior pelo mesmo valor na imagem *Boats* se comparado ao SVR. Na imagem *Lena*, o resultado de SSIM foi superior ao segundo melhor método (GSM) por 0.05. Outro ponto importante pode ser observado nas imagens de erro da figura 4.14. Nelas, observa-se um efeito borrado comumente associado à função custo erro médio quadrático [50]. Isso sugere que outras funções custo ou outros métodos que evitam funções custo explícitas [51] [52] podem ser ainda melhor sucedidos na remoção de ruído.

Capítulo 5

Conclusões

A partir de modelos baseados em redes neurais tradicionais, foram apresentados modelos de redes neurais convolucionais. Observou-se que esse tipo de modelo especializado é capaz de obter bons resultados em tarefas que envolvem processamento de sinais como os de áudio e imagem. No caso de imagens, redes convolucionais têm obtido êxito particularmente em tarefas como classificação e regressão. Por exemplo, na base de imagens CIFAR10, os modelos com maior taxa de acerto em classificação são baseados nessas redes. Em regressão, aplicações como segmentação, usual em visão computacional, e solução de problemas inversos, presentes na área do processamento de imagens, também têm sido assuntos com emprego bem sucedido. Essa série de resultados positivos instiga mais aplicações, como predição de conteúdo afetivo, geração automática de imagens e clusterização. Foram apresentados também, modelos de redes neurais convolucionais profundas, que, pela profundidade, possuem tendência a uma mais difícil otimização, juntamente com um maior poder de representação. Apresentaram-se modelos profundos como a *Residual Network* que faz uso de *skip connections* a fim de facilitar o processo de otimização. Baseada nessas conexões, foi apresentada a Rede Neural Convolucional Densamente Conectada, capaz de possuir o melhor desempenho em classificação em diversas bases de dados e, simultaneamente, um número significativamente menor de parâmetros em seu modelo se comparado a outras redes de desempenho similar.

O trabalho, então, utilizou-se de uma parte fundamental dessa rede para aplicação em regressão, o bloco convolucional densamente conectado. Primeiramente, sua im-

plementação foi validada a partir de uma reimplementação da Rede Neural Convocional Densamente Conectada e os resultados em classificação comparados com os originais publicados. A comparação foi feita em 4 situações, nas duas bases de dados CIFAR10 e CIFAR100 e em cada uma delas com *data augmentation* e sem *data augmentation*. A reimplementação possuiu um resultado pouco pior que o original (tabela 4.1), mas foi próxima o suficiente para sua aplicação.

Para a remoção de ruído de imagens cinza, foi gerada uma base de imagens cinza a partir da CIFAR10 e sua versão ruidosa. A base ruidosa foi gerada pela adição de ruído gaussiano de variância 400 e média zero nas imagens cinza da CIFAR10. Durante o treinamento, as imagens ruidosas formaram a entrada da rede. Sua saída filtrada foi comparada com a versão original por uma função custo. Em seguida, os parâmetros do modelo foram ajustados a fim de minimizar a diferença entre a imagem filtrada e a imagem original. Dessa forma, o modelo aprendeu a remover ruído gaussiano.

Diferentes modelos tiveram seus resultados comparados a fim de se selecionar o mais apropriado para remoção de ruído gaussiano. Por fim, o modelo baseado na arquitetura D (figura 4.8) com bloco densamente conectado com $K=8$ e $L=8$, nomeado *D K8 L8*, foi selecionado. Seu desempenho foi comparado com o de outros métodos de filtragem em 3 imagens padrão (*Barbara*, *Boats*, *Lena*) corrompidas por ruído gaussiano em dois experimentos. No primeiro, as imagens foram adicionadas de ruído gaussiano com variância 200. No segundo, por ruído de variância 400. Em cada situação, foi comparado um índice de similaridade SSIM e o RMSE entre a imagem filtrada e a original. No resultado do primeiro experimento, dentre os outros métodos a serem comparados, o GSM possuiu o melhor resultado. A rede *D K8 L8* possuiu SSIM 0.01 inferior a ele na imagem *Barbara*, mesmo SSIM na imagem *Boats* e SSIM 0.07 superior na imagem *Lena*. No segundo experimento, o método GSM possuiu o melhor SSIM na imagem *Bárbara*, superior à *D K8 L8* por 0.02. Na imagem *Boats*, por sua vez, a *D K8 L8* obteve SSIM superior ao GSM por 0.03 e superior ao segundo melhor método SVR por 0.02. Na última imagem, *Lena*, a rede *D K8 L8* apresenta SSIM superior ao segundo melhor método, GSM, por 0.05. Em ambos os experimentos, a rede *D K8 L8* possuiu RMSE substancialmente

inferior aos outros métodos em todas as imagens.

Conclui-se, assim, que o modelo foi bem sucedido na remoção de ruído gaussiano. O modelo *D K8 L8*, apesar de treinado em imagens com ruído de variância 400, obteve resultados competitivos tanto nas imagens corrompidas por ruído de variância 400 como também de variância 200. Além disso, também se ressalta que essa mesma metodologia pode ser empregada em outras aplicações. Sugere-se como trabalho futuro o tratamento de possíveis limitações do modelo, como o fato do valor de elementos da imagem saírem da faixa original de zero a 255 durante a adição do ruído. Em casos que se deseja aproximar a inversa de uma perturbação, basta que se aplique a perturbação em uma base de imagens, e que sejam alterados os parâmetros do modelo a fim de minimizar uma medida de diferença entre a imagem filtrada e a imagem original. Em conclusão, redes neurais convolucionais profundas, apesar de especializadas, possuem uma ampla gama de aplicações por configurarem poderosos modelos matemáticos.

Referências Bibliográficas

- [1] “Wikipedia Overfitting”, https://upload.wikimedia.org/wikipedia/commons/thumb/1/1f/Overfitting_svg.svg/2000px-Overfitting_svg.svg.png, 2010, Accessed: 2016-10-5.
- [2] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., *et al.*, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, *J. Mach. Learn. Res.*, v. 15, n. 1, pp. 1929–1958, Jan. 2014.
- [3] NAIR, V., HINTON, G. E., “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *International Conference on Machine Learning*, pp. 11–33, Haifa, 2010.
- [4] HE, K., ZHANG, X., REN, S., *et al.*, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, <https://arxiv.org/pdf/1502.01852.pdf>, 2015.
- [5] KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E., “Imagenet classification with deep convolutional neural networks”. In: *Advances in Neural Information Processing Systems*, pp. 1106–1114, 2012.
- [6] LIN, H. W., TEGMARK, M., “Why does deep and cheap learning work so well?”, <https://arxiv.org/pdf/1608.08225.pdf>, 2016.
- [7] ZEILER, M. D., FERGUS, R., *Visualizing and Understanding Convolutional Networks*, Cham, Springer International Publishing, pp. 818–833, 2014.
- [8] HE, K., ZHANG, X., REN, S., *et al.*, “Deep Residual Learning for Image Recognition”, <https://arxiv.org/pdf/1512.03385.pdf>, 2015.

- [9] HE, K., ZHANG, X., REN, S., *et al.*, “Identity Mappings in Deep Residual Networks”, <https://arxiv.org/pdf/1603.05027.pdf>, 2016.
- [10] HUANG, G., LIU, Z., WEINBERGER, K. Q., *et al.*, “Densely Connected Convolutional Networks”, <https://arxiv.org/pdf/1608.06993.pdf>, 2016.
- [11] LAPARRA, V., GUTIERREZ, J., CAMPS-VALLS, G., *et al.*, “Image Denoising with Kernels Based on Natural Image Relations”, *Journal of Machine Learning Research*, pp. 873–903, 2010.
- [12] DONOHO, D. L., “De-noising by Soft-thresholding”, *IEEE Trans. Inf. Theor.*, v. 41, n. 3, pp. 613–627, May 1995.
- [13] BUCCIGROSSI, R., SIMONCELLI, E., “Image compression via joint statistical characterization in the wavelet domain”, *IEEE Transactions on Image Processing*, v. 8, n. 12, pp. 1688–1701, 1999.
- [14] FIGUEIREDO, M., NOWAK, R., “Wavelet-based image estimation: an empirical Bayes approach using Jeffreys noninformative prior”, *IEEE Transactions on Image Processing*, v. 10, n. 9, pp. 1322–1331, 2001.
- [15] PORTILLA, J., STRELA, V., WAINWRIGHT, M., *et al.*, “Image denoising using scale mixtures of gaussians in the wavelet domain”, *IEEE Transactions on Image Processing*, v. 12, n. 11, pp. 1338–1351, nov 2003.
- [16] ZHANG, R., ISOLA, P., EFROS, A. A., “Colorful Image Colorization”, <https://arxiv.org/pdf/1603.08511.pdf>, 2016.
- [17] XU, L., REN, J. S., LIU, C., *et al.*, “Deep Convolutional Neural Network for Image Deconvolution”. In: Ghahramani, Z., Welling, M., Cortes, C., *et al.* (eds.), *Advances in Neural Information Processing Systems 27*, Curran Associates, Inc., pp. 1790–1798, 2014.
- [18] ISOLA, P., ZHU, J.-Y., ZHOU, T., *et al.*, “Image-to-Image Translation with Conditional Adversarial Networks”, <https://arxiv.org/pdf/1611.07004.pdf>, 2016.

- [19] KALCHBRENNER, N., OORD, A. V. D., SIMONYAN, K., *et al.*, “Video Pixel Networks”, <https://arxiv.org/pdf/1610.00527.pdf>, 2016.
- [20] CHEN, C., “Deep Learning for Self-driving Car”, <http://www.princeton.edu/~alaink/Orf467F14/Deep%20Driving.pdf>, Accessed: 2016-12-10.
- [21] MILLETARI, F., NAVAB, N., AHMADI, S.-A., “V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation”, <https://arxiv.org/pdf/1606.04797.pdf>, 2016.
- [22] SVOBODA, P., HRADIS, M., BARINA, D., *et al.*, “Compression Artifacts Removal Using Convolutional Neural Networks”, <https://arxiv.org/pdf/1605.00366.pdf>, 2016.
- [23] KRIZHEVSKY, A., *Learning Multiple Layers of Features from Tiny Images*. M.Sc. dissertation, University of Toronto, Abril 2009.
- [24] RUSSAKOVSKY, O., DENG, J., SU, H., *et al.*, “ImageNet Large Scale Visual Recognition Challenge”, *International Journal of Computer Vision (IJCV)*, v. 115, n. 3, pp. 211–252, 2015.
- [25] JÉGOU, S., DROZDZAL, M., VAZQUEZ, D., *et al.*, “The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation”, <https://arxiv.org/pdf/1611.09326.pdf>, 2016.
- [26] CATTIN, P., “Image Restoration: Introduction to Signal and Image Processing”, [https://miac.unibas.ch/SIP/06-Restoration.html#\(14\)](https://miac.unibas.ch/SIP/06-Restoration.html#(14)), Accessed: 2016-12-1.
- [27] WANG, Z., BOVIK, A., SHEIKH, H., *et al.*, “Image Quality Assessment: From Error Visibility to Structural Similarity”, *IEEE Transactions on Image Processing*, v. 13, n. 4, pp. 600–612, apr 2004.
- [28] CHOLLET, F., “Keras”, <https://github.com/fchollet/keras>, 2015, Accessed: 2016-5-5.
- [29] ROSSUM, G. V., DRAKE, F., “Python Programming Language”, www.python.org, 2001, Accessed: 2010-4-2.

- [30] AL, J. B. E., “Theano: A CPU and GPU Math Compiler in Python”, <http://deeplearning.net/software/theano/>, 2010, Accessed: 2016-5-5.
- [31] AL., M. A. E., “TensorFlow: Large-scale machine learning on heterogeneous systems”, <https://www.tensorflow.org/>, 2010, Accessed: 2016-5-5.
- [32] NICKOLLS, J., BUCK, I., GARLAND, M., *et al.*, “Scalable Parallel Programming with CUDA”, *Queue*, v. 6, n. 2, pp. 40–53, Mar. 2008.
- [33] WALT, S. V. D., COLBERT, S. C., VAROQUAUX, G., “The NumPy Array: A Structure for Efficient Numerical Computation”, *Computing in Science & Engineering*, v. 13, n. 2, pp. 22–30, mar 2011.
- [34] GOODFELLOW, I., BENGIO, Y., COURVILLE, A., “Deep Learning”, 2016. Book in preparation for MIT Press.
- [35] XIE, S., GIRSHICK, R., DOLLÁR, P., *et al.*, “Aggregated Residual Transformations for Deep Neural Networks”, <https://arxiv.org/pdf/1611.05431.pdf>, 2016.
- [36] ZEILER, M. D., FERGUS, R., “Visualizing and Understanding Convolutional Networks”, <https://arxiv.org/pdf/1311.2901.pdf>, 2013.
- [37] LIN, M., CHEN, Q., YAN, S., “Network In Network”, <https://arxiv.org/pdf/1312.4400.pdf>, 2013.
- [38] CAWLEY, G. C., TALBOT, N. L., “On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation”, *J. Mach. Learn. Res.*, v. 11, pp. 2079–2107, Aug. 2010.
- [39] CHABACANO, “Wikipedia Sobreajuste”, <https://pt.wikipedia.org/wiki/Sobreajuste>, 2010, Accessed: 2016-10-5.
- [40] KROGH, A., HERTZ, J. A., “A Simple Weight Decay Can Improve Generalization”. In: *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4*, pp. 950–957, 1992.

- [41] KARPATY, A., “ConvNetJS: Deep Learning in your browser”, <http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>, Accessed: 2016-12-15.
- [42] HORNIK, K., “Approximation capabilities of multilayer feedforward networks”, *Neural Networks*, v. 4, n. 2, pp. 251–257, jan 1991.
- [43] IOFFE, S., SZEGEDY, C., “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, <https://arxiv.org/pdf/1502.03167.pdf>, 2015.
- [44] BENGIO, Y., “Learning Deep Architectures for AI”, *Found. Trends Mach. Learn.*, v. 2, n. 1, pp. 1–127, Jan. 2009.
- [45] LAPIN, M., HEIN, M., SCHIELE, B., “Loss Functions for Top-k Error: Analysis and Insights”, <https://arxiv.org/pdf/1512.00486.pdf>, 2015.
- [46] SIMONYAN, K., ZISSERMAN, A., “Very Deep Convolutional Networks for Large-Scale Image Recognition”, <https://arxiv.org/pdf/1409.1556.pdf>, 2014.
- [47] “DenseNet Git”, <https://github.com/liuzhuang13/DenseNet>, Accessed: 2016-12-01.
- [48] COLLOBERT, R., BENGIO, S., MARIÉTHOZ, J., *Torch: a modular machine learning software library*, Idiap-RR Idiap-RR-46-2002, IDIAP, 0 2002.
- [49] DOZAT, T., “Incorporating Nesterov Momentum into Adam”, http://cs229.stanford.edu/proj2015/054_report.pdf, 2015, Accessed: 2016-8-6.
- [50] MATHIEU, M., COUPRIE, C., LECUN, Y., “Deep multi-scale video prediction beyond mean square error”, <https://arxiv.org/pdf/1511.05440.pdf>, 2015.
- [51] GOODFELLOW, I. J., POUGET-ABADIE, J., MIRZA, M., *et al.*, “Generative Adversarial Networks”, <https://arxiv.org/pdf/1511.06434.pdf>, 2014.
- [52] RADFORD, A., METZ, L., CHINTALA, S., “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”, <https://arxiv.org/pdf/1511.06434.pdf>, 2015.

Apêndice A

Código da implementação da Rede Neural Convolutacional Densamente Conectada

```
from __future__ import print_function
import numpy as np
np.random.seed(1234)

from keras.datasets import cifar10
from keras.layers import merge, Input, Dropout
from keras.layers.convolutional import Convolution2D, ZeroPadding2D,
    AveragePooling2D
from keras.layers.core import Dense, Activation, Flatten
from keras.layers.normalization import BatchNormalization
from keras.layers.pooling import MaxPooling2D, GlobalAveragePooling2D

from keras.models import Model
from keras.preprocessing.image import ImageDataGenerator
from keras.utils import np_utils
from keras.regularizers import l2, activity_l2, l1
from keras.optimizers import SGD, Adam, RMSprop, Nadam
from keras.callbacks import LearningRateScheduler, ModelCheckpoint,
    EarlyStopping
```

```

from keras.callbacks import History
import keras.backend as K
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import sys
import csv
from itertools import izip
from keras.layers.advanced_activations import LeakyReLU

sys.setrecursionlimit(10000)

def bnReluConvDrop(depth, kernelx,
    kernely, droprate=0., stride=(1,1), weight_decay=1e-4, bnL2norm=0.0001):
    def f(net):
        net =
            BatchNormalization(mode=0, axis=1, gamma_regularizer=l2(bnL2norm), beta_regularizer=
net = Activation('relu')(net)
net = Convolution2D(depth, kernelx, kernely, border_mode='same',
    init='he_normal', subsample=stride,
    bias=False, W_regularizer=l2(weight_decay), input_shape=(3,
    None, None))(net)
    if droprate>0.:
        net=Dropout(droprate)(net)
    return net
    return f

def bnConv(depth, kernelx,
    kernely, stride=(1,1), weight_decay=1e-4, bnL2norm=0.0001):
    def f(net):
        net =
            BatchNormalization(mode=0, axis=1, gamma_regularizer=l2(bnL2norm), beta_regularizer=
net = Convolution2D(depth, kernelx, kernely, border_mode='same',
    init='he_normal', subsample=stride,
    bias=False, W_regularizer=l2(weight_decay), input_shape=(3,

```

```

        None, None))(net)
    return net
return f

def bnReluConvDrop_2(depth, kernelx,
kernely, droprate=0., stride=(1,1), weight_decay=1e-4, bnL2norm=0.0001):
def f(net):
    net =
        BatchNormalization(mode=2, axis=1, gamma_regularizer=l2(bnL2norm), beta_regularizer=
net = Activation('relu')(net)
net = Convolution2D(depth, kernelx, kernely, border_mode='same',
    init='he_normal', subsample=stride,
    bias=False, W_regularizer=l2(weight_decay), input_shape=(None,
    None, None))(net)
    if droprate>0.:
        net=Dropout(droprate)(net)
    return net
return f

def bnLeakyReLUConvDrop_2(depth, kernelx,
kernely, droprate=0., stride=(1,1), weight_decay=1e-4, bnL2norm=0.0001):
def f(net):
    net =
        BatchNormalization(mode=2, axis=1, gamma_regularizer=l2(bnL2norm), beta_regularizer=
net = LeakyReLU(alpha=0.2)(net)
net = Convolution2D(depth, kernelx, kernely, border_mode='same',
    init='he_normal', subsample=stride,
    bias=False, W_regularizer=l2(weight_decay), input_shape=(None,
    None, None))(net)
    if droprate>0.:
        net=Dropout(droprate)(net)
    return net
return f

```

```

def
denseBlock_layout_2(net,feature_map_n_list,n_filter,droprate=0.,weight_decay=1e-4):
layer_list=[net]
n_filter = n_filter
for i in xrange(len(feature_map_n_list)):
    net = bnReluConvDrop_2(feature_map_n_list[i], 3,
        3,droprate=droprate,stride=(1,1),weight_decay=weight_decay)(net)
    layer_list.append(net)
    net = merge(layer_list , mode='concat',concat_axis=1)
    n_filter+=feature_map_n_list[i]
return net,n_filter

def
denseBlock_layout_2_Leaky(net,feature_map_n_list,n_filter,droprate=0.,weight_decay=1e-
layer_list=[net]
n_filter = n_filter
for i in xrange(len(feature_map_n_list)):
    net = bnLeakyReLUConvDrop_2(feature_map_n_list[i], 3,
        3,droprate=droprate,stride=(1,1),weight_decay=weight_decay)(net)
    layer_list.append(net)
    net = merge(layer_list , mode='concat',concat_axis=1)
    n_filter+=feature_map_n_list[i]
return net,n_filter

def DropAp(pool_size=(2,2),strides=(2,2), droprate = 0.0):
def f(net):
    if droprate != 0.:
        net = Dropout(droprate)(net)
    net = AveragePooling2D(pool_size, strides=strides,
        border_mode='valid', dim_ordering='default')(net)
    return net
return f

```

```

def
    denseBlock_layout(net,feature_map_n_list,n_filter,droprate=0.,weight_decay=1e-4):
layer_list=[net]
n_filter = n_filter
for i in xrange(len(feature_map_n_list)):
    net = bnReluConvDrop(feature_map_n_list[i], 3,
        3,droprate=droprate,stride=(1,1),weight_decay=weight_decay)(net)
    layer_list.append(net)
    net = merge(layer_list , mode='concat',concat_axis=1)
    n_filter+=feature_map_n_list[i]
return net,n_filter

def
dense_net(n_dense_blocks,input_shape,feature_map_n_list,n_filter_initial=16,droprate=0.5)
n_filter = n_filter_initial
img_input = Input(shape=input_shape)
x = Convolution2D(n_filter_initial, 3, 3, border_mode='same',
    init='he_normal', W_regularizer =
    l2(bnL2norm),bias=False,input_shape=(3, None, None))(img_input)
for i in xrange(n_dense_blocks-1):
    x,n_filter =
        denseBlock_layout(x,feature_map_n_list,n_filter,droprate=droprate)
    x =
        BatchNormalization(mode=0,axis=1,gamma_regularizer=l2(bnL2norm),beta_regularizer=l2(bnL2norm))(x)
    x = Activation('relu')(x)
    x = Convolution2D(n_filter, 1, 1, border_mode='same',
        init='he_normal', W_regularizer =
        l2(bnL2norm),activation='linear',bias=False,input_shape=(3,
        None, None))(x)
    x = DropAp(droprate=droprate)(x)

x,n_filter =
    denseBlock_layout(x,feature_map_n_list,n_filter,droprate=droprate)

```



```
x =  
    BatchNormalization(mode=0,axis=1,gamma_regularizer=l2(bnL2norm),beta_regularizer=l2(bnL2norm))(x)  
x = Activation('relu')(x)  
x = GlobalAveragePooling2D()(x)  
preds = Dense(nb_classes, activation='softmax',  
              W_regularizer=l2(l2_dense_penalization))(x)  
  
model = Model(input=img_input, output=preds)  
return model
```
