Implementação e Validação de Redes Neurais Capsulares

Antonio Pedro de Sousa Vieira
Faculdade de Engenharia Elétrica e de Computação
Universidade de Campinas
apsvieira95@gmail.com

Resumo

Redes neurais convolucionais são hoje dominantes na tarefa de classificação de objetos em imagens e vídeos. Essas redes fazem uso de camadas de *max-pooling*, replicação de mapas de características e técnicas de aumento de dados para introduzir tolerância a transformações afins, de modo que as redes sejam capazes de identificar objetos de uma determinada classe independentemente de sua orientação, tamanho ou posição em uma imagem. Redes neurais formadas por cápsulas - *CapsNets* - foram propostas como uma arquitetura alternativa para introduzir tolerância sem descartar informação ou realizar procedimentos de aumento de dados. Redes simples com cápsulas alcançaram bons desempenhos em tarefas de reconhecimento e separação de imagens de datasets como MNIST [1]. A proposta deste trabalho é realizar uma implementação de redes neurais com cápsulas e reproduzir os principais experimentos realizados em [1]. Por fim, propõe-se a realização de experimentos em outros conjuntos de dados com configurações diferentes de imagens, a fim de comparar a acurácia de classificação de arquiteturas *CapsNets* com arquiteturas convolucionais tradicionais em outras tarefas.

1 Introdução

Redes neurais artificiais são formadas pela aglomeração de unidades computacionais básicas, chamadas de neurônios por terem sido originalmente fundamentadas em um modelo matemático para um neurônio [2]. Este modelo, por vezes chamado de *perceptron*, propõe que a saída de cada unidade corresponde à aplicação de alguma "função de ativação" à sua entrada. Redes *feed-forward*, nas quais não há laços de realimentação - recorrência - entre as camadas da rede, são formadas pela adição de neurônios a uma camada ou pela adição de novas camadas à rede. Este trabalho é voltado para redes sem recorrência.

As camadas da rede que não recebem entrada externa e não produzem saída externa são denominadas de escondidas. Essas camadas se comunicam apenas com outras camadas da rede e não com o restante do programa ou com o usuário. Neste contexto, chama-se de Rede Neural Profunda (*Deep Neural Network*, DNN) uma rede que possui múltiplas camadas escondidas. Não há um limiar definido ou um consenso claro pra dizer quando se trata de uma rede profunda. O processo de treinamento supervisionado dessas redes, comumente chamado de *Deep Learning*, é uma área de grande atividade. Boa parte dessa atividade foi motivada por trabalhos envolvendo aplicação de DNNs às áreas de visão computacional [3] e reconhecimento de fala [4].

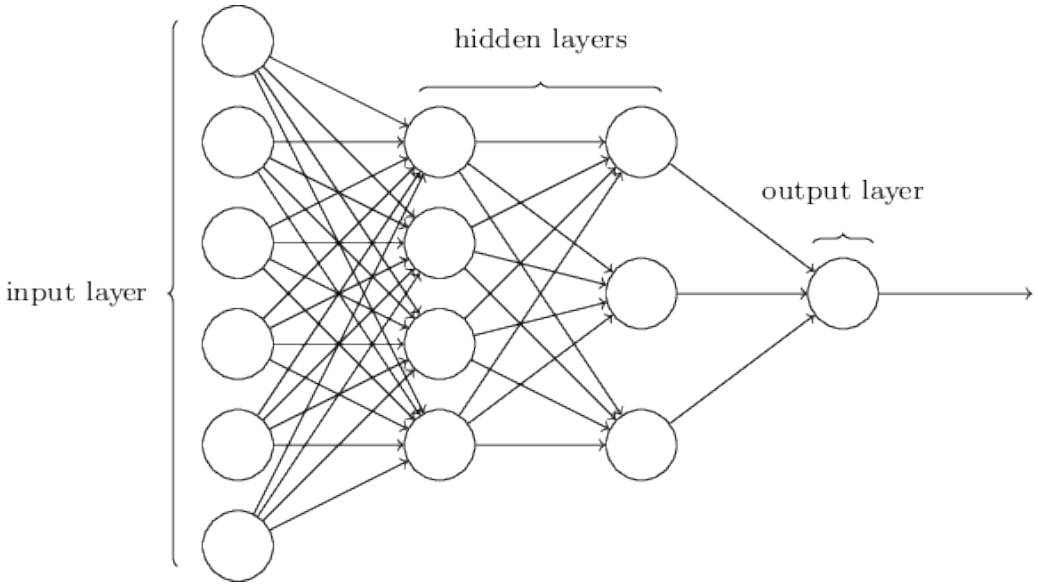


Figura 1: Representação de uma rede neural *feed-forward*, com camadas totalmente conectadas. Essa rede é composta por uma camada de entrada, duas camadas escondidas e uma unidade de saída. (Reproduzido de [5, Cap. 1])

Redes neurais convolucionais são aquelas nas quais algumas ou todas as camadas realizam operações de convolução. Essas redes tornaram-se um tópico de intenso estudo após o desempenho de redes convolucionais profundas em tarefas de classificação de imagens [3] e de subsequentes avanços com outras redes convolucionais em tarefas de classificação de imagens [6, 7, 8, 9] e de segmentação de imagens [10, 11]. Argumenta-se [1] que, agora que redes convolucionais se tornaram a solução dominante em reconhecimento de objetos, deve-se procurar entender possíveis limitações desses modelos.

Uma dessas limitações é a baixa capacidade de lidar com transformações afins - como rotação ou mudança de escala. Redes convolucionais possuem melhor tolerância à translação por conta em parte da natureza da operação de convolução. Nessa operação, um mesmo mapa de parâmetros é aplicado a diversas pequenas janelas de uma imagem. Dessa forma, deslocamentos de características das imagens se traduzem em deslocamentos das ativações produzidas pela convolução. Além disso, a operação de *pooling*, que introduz equivariância a pequenas translações, também contribui para a tolerância a translação, já que pequenas variações podem ser interpretadas como um deslocamento do pixel que será escolhido como ativado dentro de uma mesma região.

A alternativa apresentada [1] para obter melhor tolerância a transformações afins e utilizar um modelo de rede neural que se aproxime mais de um modelo de visão humana é fazer uso de cápsulas [12], sendo essas, agora, as menores unidades de decisão da rede. Cada cápsula produz em sua saída um vetor de características, cujo módulo representa a intensidade de ativação. Estes vetores são usados para realizar um procedimento de roteamento entre as cápsulas, de forma que cada uma delas se relacione diferentemente com as unidades de outras camadas. Essa relação é modificada durante o treinamento. As cápsulas apresentariam como vantagem uma melhor tolerância a transformações afins variadas sem necessidade de realização de *data augmentation* [1].

O modelo proposto é utilizado em problemas de classificação de imagens e comparado a resultados

obtidos por modelos mais complexos. É criado também um modelo de rede convolucional comum, chamado de *baseline* e usado como referência, projetado para ter número de parâmetros similar ao modelo com cápsulas e apresentar bom desempenho nas tarefas especificadas. Essa arquitetura capsular é base para uma arquitetura diferente para redes capsulares [13], que foi aplicada aos conjuntos de dados MNIST, smallNORB [14] e CIFAR-10 [15]. A variação proposta consiste, majoritariamente, em deixar de usar a representação vetorial para as cápsulas e passar a usar um escalar para representar a intensidade de ativação e uma matriz normalizada para codificar todo o restante, como pose, coloração e intensidade luminosa. A mudança na arquitetura vem acompanhada do uso de uma nova abordagem para o roteamento entre cápsulas, fazendo uso do algoritmo de maximização de expectativas.

A arquitetura explorada nesse trabalho tem três principais motivações. A primeira é uma possível redução do número de parâmetros em relação a redes convolucionais, sendo possível manter um desempenho competitivo com uma rede menor. A segunda é a possibilidade de maior tolerância a transformações afins. Por fim, o bom desempenho em identificação de elementos sobrepostos em imagens, como exemplificado por um experimento realizado com o conjunto de dados *MultiMNIST* [1].

2 Blocos funcionais da rede capsular

Ainda que seu funcionamento difira em alguns pontos, as redes capsulares são compostas majoritariamente pelos mesmos blocos funcionais que compoem redes convolucionais. Os principais desses blocos são as próprias camadas convolucionais, além de camadas lineares - também chamadas de totalmente conectadas ou densas - que realizam as operações mais custosas e armazenam os parâmetros que são modificados durante o treinamento. As maiores diferenças da arquitetura capsular se encontram nas camadas capsulares com *routing*. Essa seção tem o objetivo de esclarecer o funcionamento dos principais componentes das arquiteturas desenvolvidas.

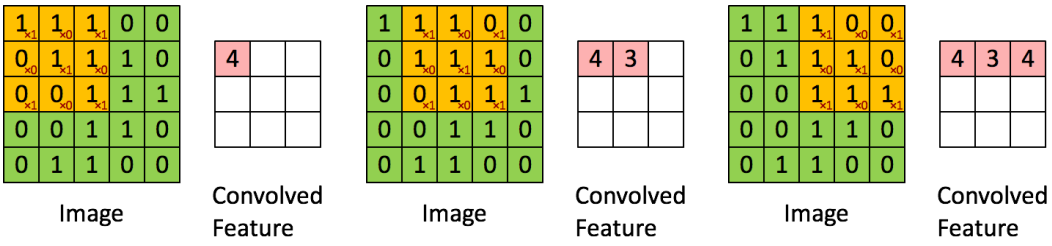
2.1 Camada Linear

Camadas que aplicam transformações lineares sobre os dados de entrada. A saída dessas camadas, dada uma entrada vetorial x, assume a forma $Ax + b$, onde A e b são, respectivamente, a matriz de coeficientes angulares e o vetor de coeficientes lineares da transformação. Os parâmetros A e b são otimizados durante o treinamento.

2.2 Camada Convolucional

Camadas que aplicam operações de convolução.[1] A operação realizada varia em formulação matemática de acordo com a implementação e frequentemente difere da definição clássica de convolução. Essas operações são aplicações locais de operadores lineares: a operação de convolução pode ser interpretada como a multiplicação de uma matriz de entrada por uma matriz de parâmetros. Essa matriz de parâmetros é formada pela replicação de parâmetros de uma matriz menor, chamada *kernel*, de modo a formar uma matriz duplamente circulante. Esse *kernel* também pode ser entendido como

[1] Frequentemente, essas camadas realizam operações de correlação cruzada, como é o caso do framework PyTorch, que foi utilizado para desenvolver este trabalho. Uma operação de convolução pode ser interpretada como uma operação de correlação cruzada com *kernel* invertido. `https://pytorch.org/docs/stable/nn.html#conv2d`



(a) Passo 1 **(b)** Passo 2 **(c)** Passo 3

Figura 2: Ilustração de uma convolução bidimensional. O *kernel* (em amarelo) é deslocado sobre a entrada a cada passo. Os parâmetros da matriz de entrada e do *kernel* são multiplicados elemento a elemento. Cada posição do *kernel* gera uma posição da saída. Nesse exemplo, o passo é unitário e não há preenchimento das bordas.(Retirado de [17])

uma janela de parâmetros que é deslocada sobre os dados de entrada. Cada aplicação dessa janela de parâmetros a uma porção da entrada gera um elemento da saída. A Figura 2 representa alguns passos de uma convolução bidimensional com *kernel*.

As equações (1) e (2) descrevem a altura e a largura da matriz de saída em uma convolução bidimensional no *framework* Pytorch [16]. O parâmetro de *padding* determina a quantidade de pixels utilizados para o preenchimento das bordas da entrada. O parâmetro de dilatação (*dilation*) determina o espaçamento entre as posições em que o *kernel* é aplicado (ou o espaçamento entre posições do *kernel*). Por fim, o parâmetro *stride* corresponde ao passo de deslocamento do *kernel* sobre a entrada em cada direção.

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * padding[0] - dilation[0] * (kernel\_size[0] - 1) - 1}{stride[0]} + 1 \right\rfloor \tag{1}$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * padding[1] - dilation[1] * (kernel\_size[1] - 1) - 1}{stride[1]} + 1 \right\rfloor \tag{2}$$

2.3 ReLU

Função de ativação que dá resposta nula para entradas negativas e replica a entrada para entradas positivas. É formalmente definida como $max(0, x)$. Essa função é uma das funções de ativação mais comuns em redes convolucionais. A Figura 3 mostra uma comparação entre a função ReLU e a função sigmóide para entradas em um pequeno segmento da reta real. A Figura 4 mostra as respostas de algumas generalizações da função ReLU para o mesmo intervalo de entrada. É interessante notar que algumas dessas funções têm o mesmo comportamento da função ReLU para valores positivos.

Em [18] são feitas diversas considerações sobre os primeiros usos de funções lineares por partes em redes neurais, sendo citados também exemplos de generalizações dessa função. A utilização da ativação ReLU em redes neurais profundas é comumente creditada a [19], onde são explorados alguns efeitos da substituição do uso de sigmoides ou tangentes hiperbólicas como funções de ativação. Entre os efeitos observados estão a maior esparsidade da atividade da rede e o melhor desempenho em tarefas de classificação de imagens.

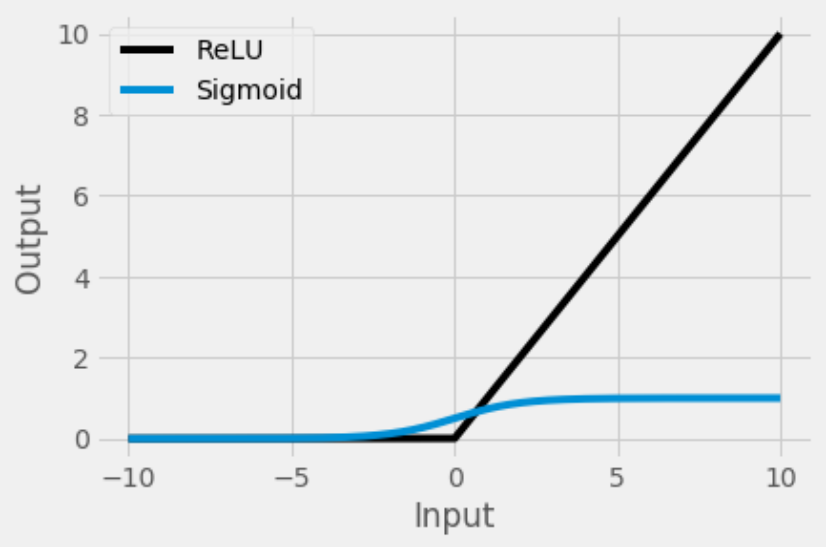


Figura 3: Funções ReLU e sigmoide. O fato de a função ReLU produzir zeros para todos os valores negativos leva a uma maior esparsidade das ativações.

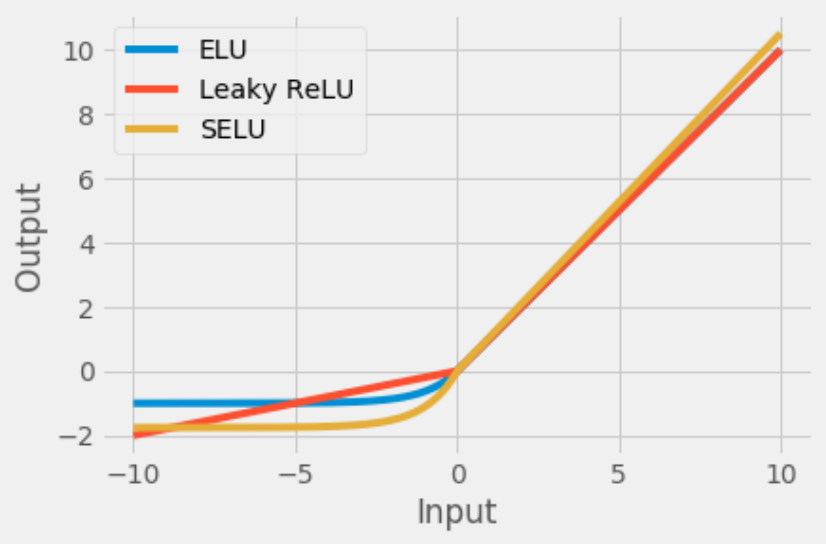


Figura 4: Generalizações da função ReLU. Para as funções ELU e Leaky ReLU, a resposta no semieixo positivo é igual à ReLU. As principais diferenças dessas generalizações são as respostas no semieixo negativo, para as quais as três apresentam formulações diferentes.

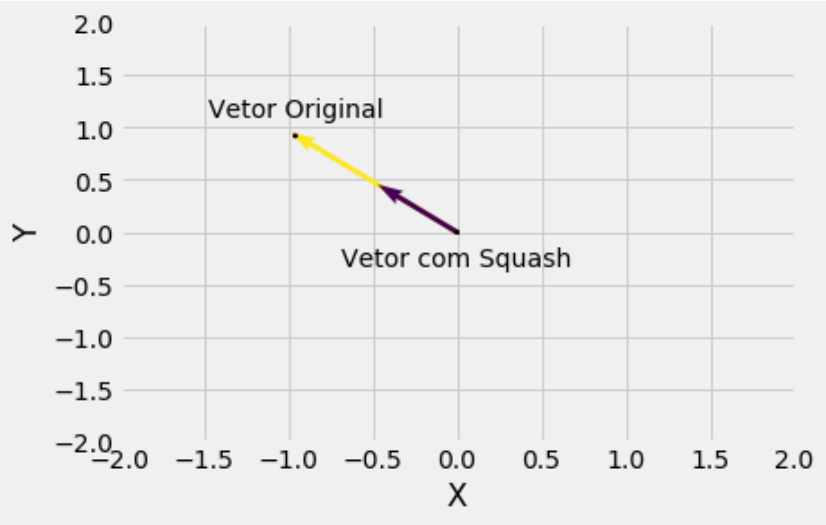


Figura 5: Exemplo de aplicação da função *squash* a um vetor bidimensional. O vetor, após a aplicação da função, mantém a mesma direção e sentido, mas tem seu módulo alterado.

2.4 Squash

Não-linearidade proposta para manter os tensores de saída das camadas capsulares com valores próximos do intervalo $[0, 1]$ [1]. Usada nas camadas capsulares da rede e no procedimento de roteamento. Pode ser interpretada como uma normalização do tensor de entrada. A Figura 5 mostra o resultado da aplicação dessa função a um vetor bidimensional.

$$\mathbf{y} = \frac{\|\mathbf{x}\|^2}{1 + \|\mathbf{x}\|^2} \frac{\mathbf{x}}{\|\mathbf{x}\|} \tag{3}$$

Há três casos em que é interessante analisar o valor dessa função:

1. Vetores com módulo muito pequeno: $\lim_{\|\mathbf{x}\|\to 0} \frac{\|\mathbf{x}\|^2}{1+\|\mathbf{x}\|^2} \to 0 \Rightarrow \mathbf{y} \to 0$
2. Vetores com módulo muito grande: $\lim_{\|\mathbf{x}\|\to \infty} \frac{\|\mathbf{x}\|^2}{1+\|\mathbf{x}\|^2} \to 1 \Rightarrow \mathbf{y} \to \frac{\mathbf{x}}{\|\mathbf{x}\|} = \hat{\mathbf{x}}$, onde $\hat{\mathbf{x}}$ é o vetor unitário na direção de $\hat{\mathbf{x}}$.
3. Vetores com módulo aproximadamente unitário: $\lim_{\|\mathbf{x}\|\to 1} \frac{\|\mathbf{x}\|^2}{1+\|\mathbf{x}\|^2} \to \frac{1}{2} \Rightarrow \mathbf{y} \to \frac{1}{2}\hat{\mathbf{x}}$

2.5 Routing

Procedimento utilizado para calcular a concordância entre duas camadas consecutivas de cápsulas [1]. Saídas que são alinhadas são reforçadas, pois a operação feita para medir a concordância é um produto escalar. Esse procedimento é usado somente se uma camada e a camada anterior a ela forem capsulares. O procedimento pode ser interpretado como a determinação de log-probabilidades usando como entrada a saída de uma camada e uma *prior* para as log-probabilidades. Para o conjunto de dados MNIST, a *prior* é uma distribuição uniforme sobre as classes.

Figura 6: Imagens retiradas do conjunto de treino do conjunto de dados MNIST.

3 Experimentos com MNIST

3.1 O Conjunto de Dados

O conjunto de dados MNIST [20] é composto de 70000 imagens, sendo 60000 destas para treino e 10000 separadas para teste. Todas as imagens são de tamanho 28x28 pixels, em escala de cinza. A Figura 6 contém algumas imagens retiradas do conjunto de treinamento. As imagens do conjunto de teste têm o mesmo formato.

3.2 Arquitetura da Rede Capsular

A arquitetura descrita [1] para esse conjunto de dados é composta de um codificador e um decodificador. O codificador, composto por uma camada convolucional e duas camadas capsulares, é a parte central da rede e pode ser utilizado independentemente. O decodificador, composto de três camadas totalmente conectadas, tem o papel de reconstruir as imagens de entrada a partir da saída do codificador.

3.2.1 Arquitetura do Codificador

A Figura 7 mostra a arquitetura do codificador. Nessa seção são delineadas as principais propriedades dessa sub-rede. Os valores de altura e largura das matrizes de saída podem ser determinados a partir das equações (1) e (2). A dimensão de tamanho de *batch* é omitida, já que é constante em todas as camadas e o funcionamento da rede independe desse valor. A saída da camada *PrimaryCaps* é redimensionada antes da entrada da camada *DigitCaps*, sendo esse o motivo de os formatos das duas serem diferentes. A saída da camada *DigitCaps* é achatada até ter formato 10

- **ReLU Conv1**
  1. Camada de entrada convolucional, com ativação ReLU e uso de *Batch Normalization*.
  2. Kernel de tamanho 9x9, com passo unitário.
  3. Entrada com formato (1 canal, 28 pixels de altura, 28 pixels de largura).
  4. Saída com formato (256 canais, 20 pixels de altura, 20 pixels de largura).
- **PrimaryCaps**
  1. Camada capsular com aplicação da função *squash* como não-linearidade, sem roteamento na entrada.

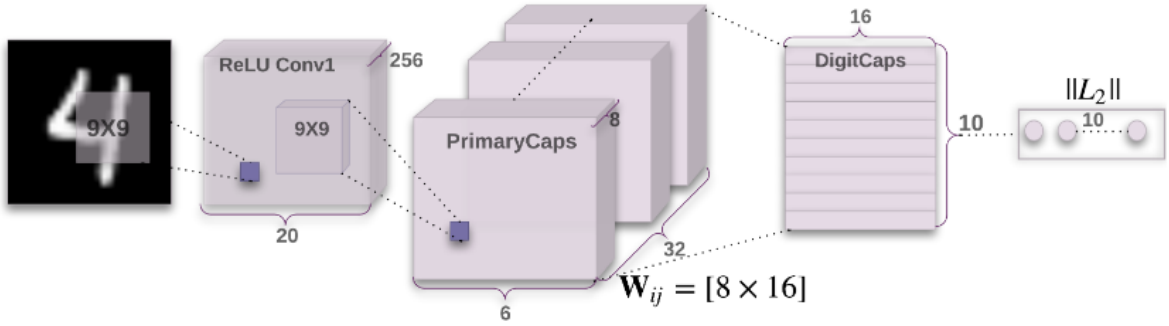


Figura 7: Arquitetura do codificador da rede capsular. A matriz W_{ij} representa a relação entre uma cápsula i da camada *PrimaryCaps* e uma cápsula j da camada *DigitCaps*. (Reproduzido de [1, Fig. 1])

2. 32 cápsulas com 8 dimensões cada, com kernel de tamanho 9x9 e passo 2.
3. Saída com tamanho (32 cápsulas, 8 dimensões, 6 pixels de altura, 6 pixels de largura).

- **DigitCaps**

1. Camada com 10 cápsulas, cada uma com dimensão 16. Usa *squash* e *routing*.
2. Entrada com tamanho (8 dimensões, 32 x 6 x 6).
3. Saída com tamanho (10 classes, 32 x 6 x 6, 8, 16), onde os tamanhos 8 e 16 representam o número de dimensões das cápsulas das camadas *PrimaryCaps* e *DigitCaps*.

3.2.2 Arquitetura do Decodificador

1. Camada densa com saída de tamanho 512, com ativação *ReLU*.
2. Camada densa com saída de tamanho 1024, com ativação *ReLU*.
3. Camada densa com saída de tamanho 784, com ativação sigmóide.

3.3 Arquitetura da Rede de Referência

A rede *baseline* é composta por três camadas convolucionais seguidas de três camadas densas [1]. Todas as camadas convolucionais têm 256 canais de saída, kernel de tamanho 5x5, passo unitário e ativação *ReLU*. As duas últimas camadas densas estão conectadas com *dropout*. A saída da última camada densa tem tamanho 10, igual ao número de classes do conjunto de dados.

3.4 Metodologia

Para cada uma das arquiteturas de rede, foram realizados testes com *batches* de tamanhos 16 e 32. Em todos os casos, o treinamento é feito com 50 épocas, usando *early stopping* com parâmetro de paciência 10. Além disso, 20% dos dados do conjunto de treinamento são separados aleatoriamente

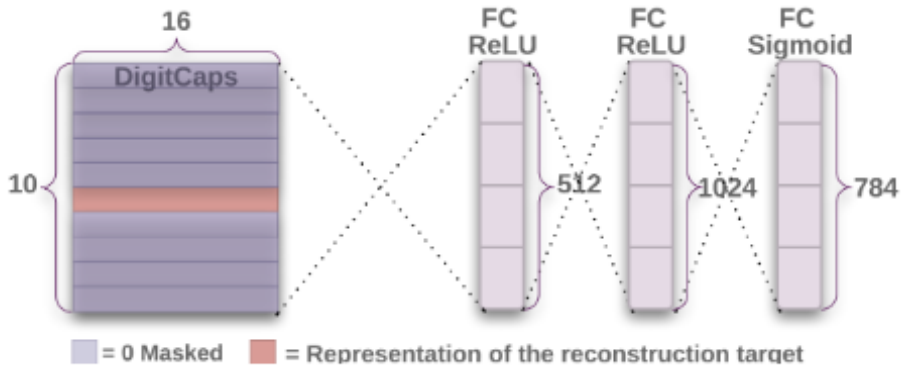


Figura 8: Arquitetura do decodificador da rede capsular. Aqui, *FC* é abreviação para *fully connected*, representando uma camada totalmente conectada. (Reproduzido de [1, Fig. 2])

para formar um conjunto de validação. A subseção 3.4.1 é dedicada explicar parte da terminologia utilizada nesse contexto.

Durante o treinamento, as imagens são transformadas através de translações de até 2 pixels em cada direção, aplicadas aleatoriamente. As mesmas transformações são aplicadas aos dados para todos os modelos treinados. Para testes, não são aplicadas transformações sobre as imagens.

3.4.1 Terminologia relevante

O Conjunto de Validação A verdadeira intenção de criar um modelo de aprendizagem de máquina é que esse modelo tenha bom desempenho em amostras que não foram utilizadas no processo de treinamento. Comumente, refere-se a esse processo de manter um bom desempenho em novas amostras como generalização. O treinamento é feito de modo a reduzir o erro sobre o conjunto de treino. Em teoria, não há como saber qual será o erro sobre o conjunto de testes sem expor o modelo ao conjunto de testes. Por outro lado, se o conjunto de testes for usado para escolher o melhor modelo, teremos uma métrica irreal da verdadeira capacidade de generalização, já que o conjunto de testes não será um conjunto de novas amostras. Para estimar a capacidade de generalização de um modelo, podemos separar parte do conjunto de treino e medir a generalização para esse conjunto, escolhendo o modelo que tenha menor erro nesse novo conjunto de validação.

Early Stopping e Paciência Para tentar encontrar uma configuração da rede que tenha bom desempenho em dados de teste, é feito o uso do conjunto de validação como referência para o desempenho que a rede terá em casos de teste. O procedimento para a escolha desse modelo é, em linhas gerais, o descrito no Algoritmo 1.

Batching e Época É prática comum separar os conjuntos de treino, validação e testes em pequenos grupos determinados de maneira aleatória. Isso é motivado por restrições de memória, já que grandes conjuntos de imagem não cabem completamente em memória, e pela convergência estável e rápida de métodos de gradiente descendente que usam *minibatches* [21, 22]. Uma *época* geralmente se refere a quando o número total de imagens do conjunto de treino é usado. A amostragem para a

Algorithm 1 Early Stopping

Require: $patience, epochs$

$bestAccuracy \leftarrow 0$
$counter \leftarrow 0$
$i \leftarrow 0$
while $counter < patience$ and $i < epochs$ **do**
  if $accuracy_i > bestAccuracy$ **then**
    $bestAccuracy \leftarrow accuracy_i$
    $bestEpoch \leftarrow i$
    $counter \leftarrow 0$
  else
    $counter \leftarrow counter + 1$
  end if
end while
return $bestEpoch$

[1]

criação de *minibatches* frequentemente é feita sem reposição, de forma que a cada época cada uma das imagens do conjunto de treino é mostrada uma única vez para a rede neural. Para os conjuntos de validação e testes, a amostragem para a criação de *minibatches* não é realizada de maneira aleatória, já que a ordem de análise dessas imagens não influencia nos resultados.

3.4.2 Laço de Treinamento

O laço principal de otimização é muito similar para todas as redes utilizadas. A principal alteração para diferentes modelos é a inclusão de um segundo tensor de saída para os experimentos onde é usado o decodificador nas redes capsulares. O Algoritmo 2 representa esse laço de maneira simplificada. Nesse algoritmo estão incluídos o processo de *early stopping* e o armazenamento dos resultados intermediários.

O algoritmo de *back-propagation* e o passo de atualização dos pesos, feito pelo otimizador, são implementados no *framework* Pytorch. O otimizador Adam [23], com os parâmetros padrão, foi utilizado em todos os processos de treinamento. Esses parâmetros são os mesmos utilizados no *framework* Tensorflow [24]. A função de avaliação dos modelos *evaluateModel* é delineada no Algoritmo 3. Essa função mede o número de acertos de classificação de um modelo em um certo conjunto de dados. Esses números são convertidos em um acerto percentual. Essa métrica é utilizada para escolher o melhor modelo.

3.5 Resultados

Os acertos percentuais de cada modelo testado são disponibilizados na Tabela 1. Cada um dos modelos foi treinado cinco vezes segundo o procedimento descrito na subseção 3.4. A partir desses resultados, a média e o desvio padrão de erros percentuais para cada um dos modelos foram calculados. Esses resultados são mostrados na Tabela 2.

Algorithm 2 Training

Require: $model, numEpochs, lossFunction, optimizer, trainData, trainTargets, validationData, validationTargets, patience, batchSize$

$lossHistory \leftarrow zeros[1..epochs]$
$accuracyHistory \leftarrow zeros[1..epochs]$
$bestValidationAccuracy \leftarrow 0$
$patienceCounter \leftarrow 0$
$numTrainingBatches \leftarrow count(trainData)/batchSize$
$epoch \leftarrow 0$
for $numEpochs$ iterations **do**
  $totalLoss \leftarrow 0$
  $epoch \leftarrow epoch + 1$
  $batch \leftarrow 0$
  for $numTrainingBatches$ iterations **do**
    $batch \leftarrow batch + 1$
    $model.gradients \leftarrow 0$
    $modelResults \leftarrow model(trainData[batch * batchSize : (batch + 1) * batchSize])$
    $loss \leftarrow lossFunction(modelResults, trainTargets[batch*batchSize : (batch+1)*batchSize])$
    $model.gradients \leftarrow backpropagate(model, loss)$
    $model \leftarrow optimizer.UpdateWeights(model.gradients)$
    $totalLoss \leftarrow totalLoss + loss$
  end for
  $lossHistory[epoch] \leftarrow totalLoss/numTrainingBatches$
  if patience **then**
    $accuracyHistory[epoch] \leftarrow evaluateModel(validationData, validationTargets, count(validationData))$
    if $accuracyHistory[epoch] > bestValidationAccuracy$ **then**
      $bestValidationAccuracy \leftarrow accuracyHistory[epoch]$
      $patienceCounter \leftarrow 0$
      $saveModelState(model, filepath)$
    else
      $patienceCounter \leftarrow patienceCounter + 1$
      if $patienceCounter = patience$ **then**
        return $model, lossHistory, accuracyHistory$
      end if
    end if
  end if
end for
return $model, lossHistory, accuracyHistory$

[1]

Algorithm 3 Model Evaluation

Require: $validationData, validationTargets, numValidationImages$

$correctPredictions \leftarrow 0$
$i \leftarrow 0$
for $numValidationImages$ iterations **do**
  $modelResults \leftarrow model(validationData[i])$
  $logProbabilities \leftarrow softmax(logProbabilities)$
  $prediction \leftarrow idxmax(logProbabilities)$
  if prediction == validationTargets[i] **then**
    $correctPredictions \leftarrow correctPredictions + 1$
  end if
end for
return $correctPredictions$

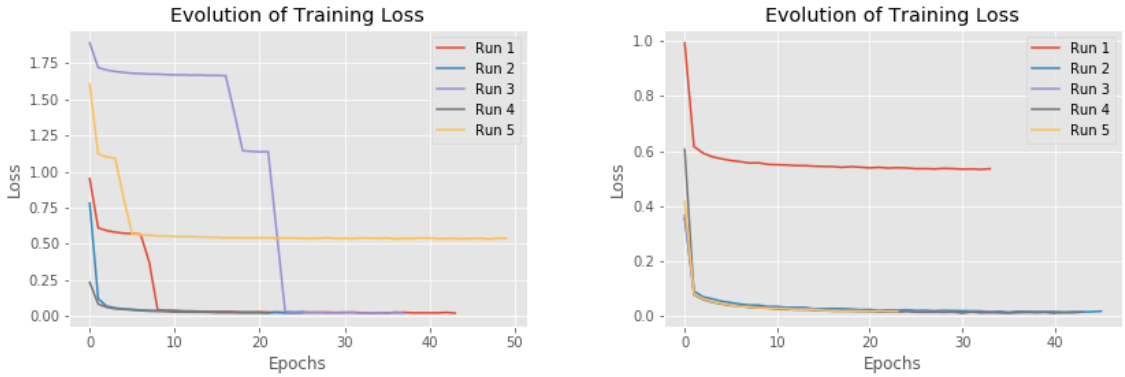
[1]

Execution	Baseline (16)	Baseline (32)	Caps (16, 1)	Caps (32, 1)	Caps (16, 3)	Caps (32, 3)
Run 1	99.07%	89.11%	99.27%	99.18%	99.31%	99.25%
Run 2	99.26%	99.15%	99.27%	99.43%	99.33%	99.29%
Run 3	99.31%	99.42%	99.19%	99.21%	99.29%	99.22%
Run 4	99.27%	99.37%	99.31%	99.27%	99.20%	99.26%
Run 5	89.57%	99.30%	99.21%	99.19%	99.19%	99.23%

Tabela 1: Resultados de 5 execuções diferentes para cada modelo. Os números dentro dos parênteses representam, respectivamente, o tamanho dos *minibatches* e o número de iterações de *routing* utilizados.

Model	Routing	Batch Size	MNIST(Error %)
Baseline	-	16	2.70 ± 4.32
Baseline	-	32	2.73 ± 4.56
CapsNet	1	16	0.75 ± 0.05
CapsNet	1	32	0.74 ± 0.10
CapsNet	3	16	0.74 ± 0.06
CapsNet	3	32	0.75 ± 0.03

Tabela 2: Erros percentuais para o conjunto de dados MNIST em diferentes configurações experimentais.



(a) Batch Size 16 **(b)** Batch Size 32

Figura 9: Evolução da métrica de perda durante o treinamento da rede de referência. Destaque para a Execução 5 na figura (a) e para a Execução 1 na figura (b), correspondentes às execuções com menor desempenho de classificação.

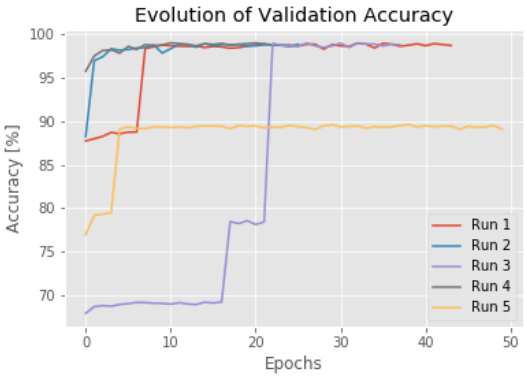
4 Discussão

Os resultados obtidos nesse trabalho são significativamente inferiores aos reportados originalmente [1], tanto para a rede *baseline* como para a rede capsular. Esses resultados indicam que as redes com arquitetura capsular apresentam convergência mais estável, representada pelo menor desvio padrão obtido para todas as configurações. Para as duas configurações da rede de referência analisadas, houve um resultado consideravelmente inferior, com erros na faixa de 10%. Isso indica a ocorrência de problemas de convergência que podem ter sido causados por uma inicialização ruim dos pesos da rede ou por aleatoriedades presentes no processo de treinamento, por exemplo. Se forem analisados apenas os exemplos em que a convergência para a rede de referência é boa, tem-se resultados em uma faixa similar aos obtidos para as redes capsulares.

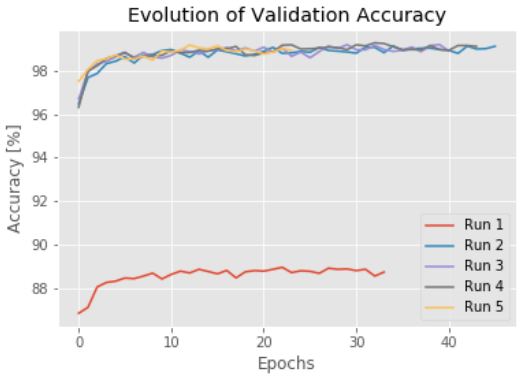
A Figura 9 ilustra a evolução das métricas de perda ao longo do treinamento da rede de referência. As curvas de perda das execuções que obtêm menor desempenho de classificação indicam que esses processos de otimização estabilizaram em mínimos locais. Isso pode ser notado pela estabilização em valores maiores de perda ao longo de grande número de épocas. A Figura 10 mostra a evolução da métrica de acurácia, que indica o desempenho de classificação dos modelos, sendo possível traçar certo paralelo entre os degraus de aumento dos acertos com os degraus de redução de erro.

As curvas de convergência para a rede capsular apresentam comportamento mais estável, com as métricas de perda evoluindo de forma similar e para um mesmo valor em cada configuração. As métricas de perda e acurácia para esses processos de treinamento estão representadas nas Figuras 11, 12, 13 e 14. As curvas de treinamento indicam que a convergência dos modelos é consideravelmente mais rápida para modelos com *minibatch* de tamanho 16, com valores de acurácia de 95% na primeira época, enquanto os modelos com tamanho de *minibatch* 32 apresentam acurácia por volta de 92% na primeira época. Apesar disso, ambas as configurações convergem para valores similares de acurácia e perda ao longo do treinamento, não havendo nesses experimentos uma execução que fique estagnada em um mínimo local de desempenho consideravelmente pior.

Os experimentos envolvendo a reconstrução das imagens com as redes capsulares não puderam

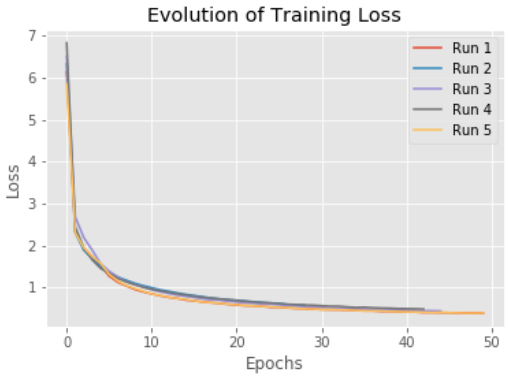


(a) Batch Size 16

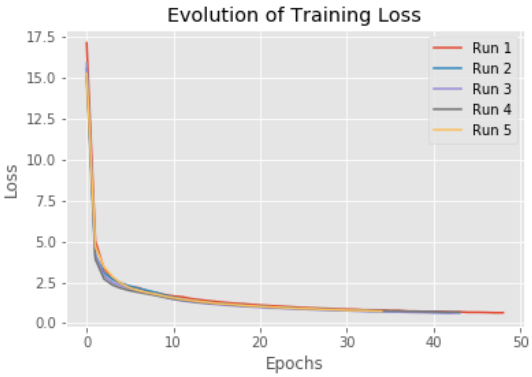


(b) Batch Size 32

Figura 10: Evolução da métrica de acurácia durante o treinamento da rede de referência. Destaque para a Execução 5 na figura (a) e para a Execução 1 na figura (b).

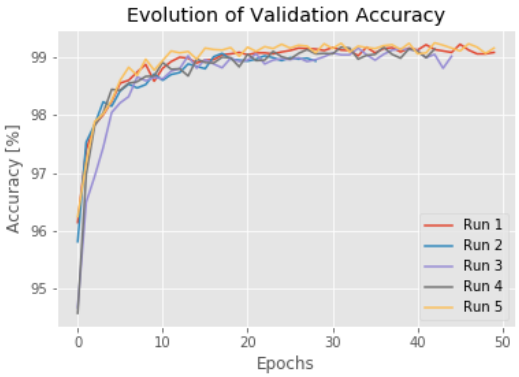


(a) Batch Size 16

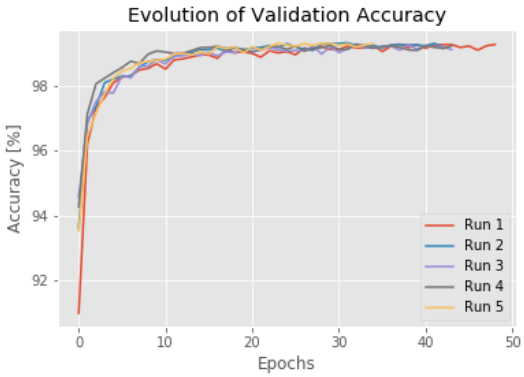


(b) Batch Size 32

Figura 11: Evolução da métrica de perda durante o treinamento da rede capsular com uma iteração de roteamento.

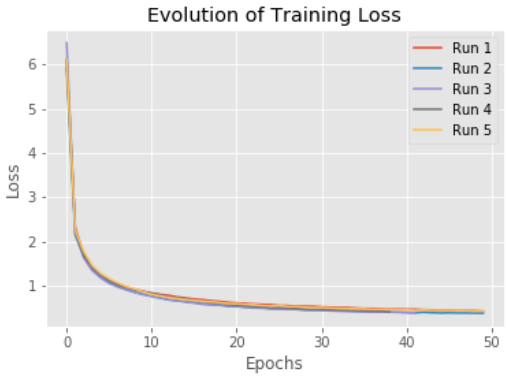


(a) Batch Size 16

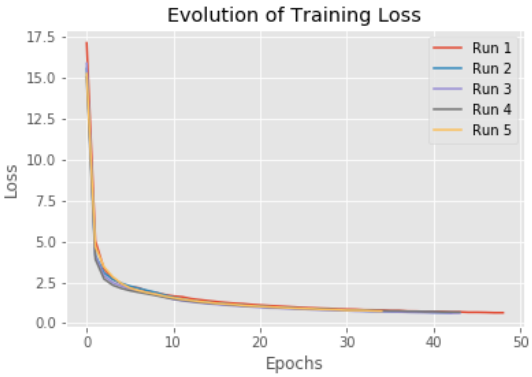


(b) Batch Size 32

Figura 12: Evolução da métrica de acurácia durante o treinamento da rede capsular com uma iteração de roteamento.

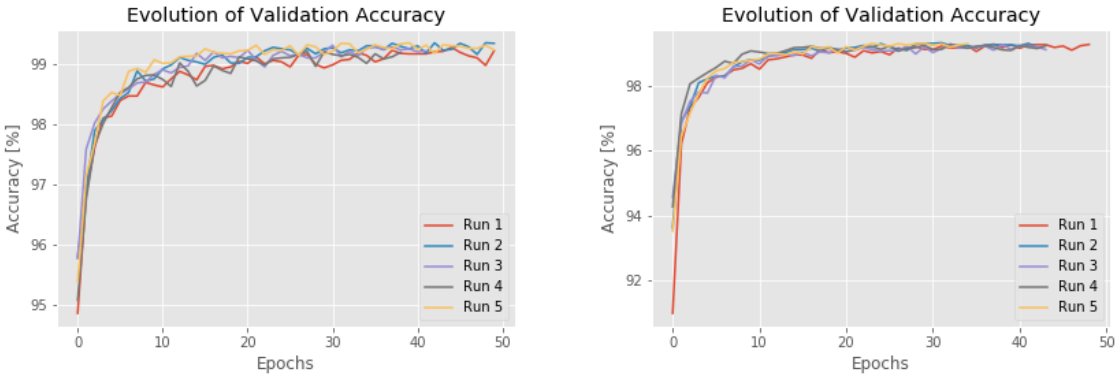


(a) Batch Size 16



(b) Batch Size 32

Figura 13: Evolução da métrica de perda durante o treinamento da rede capsular com três iterações de roteamento.



(a) Batch Size 16 **(b)** Batch Size 32

Figura 14: Evolução da métrica de acurácia durante o treinamento da rede capsular com três iterações de roteamento.

ser concluídos em tempo hábil. A principal causa disso foi o elevado custo computacional de treinar as redes com arquitetura capsular implementadas. Enquanto é possível que a implementação seja a causa desse custo elevado, não foram detectadas formas óbvias de acelerar o processo ao longo do desenvolvimento deste trabalho. Os recursos computacionais disponíveis eram suficientes para o treinamento de apenas um modelo por vez. A duração aproximada do treinamento de um modelo capsular foi de 20h no hardware disponível. Assim, cada experimento demandou por volta de quatro dias de execução em GPU. Como a inclusão do decodificador aumenta o custo computacional da rede, os experimentos que não fazem uso dessa sub-rede foram priorizados.

5 Conclusão

Os resultados obtidos para a implementação da arquitetura *CapsNet* desenvolvida neste trabalho indicam a possível existência de erros de implementação ou de erros no procedimento de treinamento. A ocorrência de resultados consideravelmente inferiores também para a rede de referência indica que é possível que as diferenças não estejam na implementação da rede capsular em si. Ainda assim, os resultados mostram bons desempenhos para a arquitetura proposta, sendo consistentemente superior aos desempenhos alcançados pela rede de referência desenvolvida.

A realização dos experimentos com o decodificador, para a reconstrução das imagens de entrada, é uma importante etapa que não foi possível concluir. A presença do decodificador permite a realização de experimentos de exploração dos resultados e do significado das saídas das cápsulas e permanece algo a acrescentar na implementação desenvolvida. Além disso, os experimentos com os conjuntos *affNIST* e *MultiMNIST* são interessantes para que se possa explorar a tolerância a transformações afins e a capacidade de discernir objetos sobrepostos. Ambos os experimentos deixaram de ser executados por restrições de tempo e *hardware* e são importantes sequências dos experimentos realizados neste trabalho.

Agradecimentos

Gostaria de agradecer ao colega Ivan Marin pelos debates, comentários e correções ao longo da concepção e da execução deste trabalho. Agradeço também a Gabriella Erbolato pela ajuda na revisão do texto da proposta de projeto e deste relatório, além do constante incentivo e companhia ao longo de todo o processo. Por fim, aos companheiros da República SantoMé e aos amigos e amigas que me ajudaram a seguir são e feliz, além de terem muitas vezes oferecido importantes correções, dedico o último esforço deste curso de graduação e meu mais sincero muito obrigado.

Referências

[1] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," *CoRR*, vol. abs/1710.09829, 2017.

[2] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, Dec 1943.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[4] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, pp. 82–97, Nov 2012.

[5] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[8] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015.

[9] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[10] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *CoRR*, vol. abs/1411.4038, 2014.

[11] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015.

[12] G. E. Hinton, A. Krizhevsky, and S. D. Wang, "Transforming auto-encoders," in *Artificial Neural Networks and Machine Learning – ICANN 2011* (T. Honkela, W. Duch, M. Girolami, and S. Kaski, eds.), (Berlin, Heidelberg), pp. 44–51, Springer Berlin Heidelberg, 2011.

[13] G. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with em routing," 2018.

[14] Y. LeCun, F. J. Huang, and L. Bottou, "Learning methods for generic object recognition with invariance to pose and lighting," in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 2, pp. II–97–104 Vol.2, June 2004.

[15] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[16] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[17] "Feature extraction using convolution."

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[19] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15 of *Proceedings of Machine Learning Research*, (Fort Lauderdale, FL, USA), pp. 315–323, PMLR, 11–13 Apr 2011.

[20] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[21] D. R. Wilson and T. R. Martinez, "The general inefficiency of batch training for gradient descent learning," *Neural Netw.*, vol. 16, pp. 1429–1451, Dec. 2003.

[22] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *CoRR*, vol. abs/1804.07612, 2018.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.