[PSI3471 aula 12. Início]
Alunos/as de PSI3471: Com prof. Magno, vocês estudaram CNN como uma evolução de MLP. Nesta aula, vamos estudar como CNN sob um outro ponto de vista: como uma forma de projetar automaticamente um conjunto de filtros lineares adequados para realizar uma determinada tarefa de visão computacional. O algoritmo de Viola e Jones usou uma série de filtros lineares para detectar rostos humanos, só que o projeto do sistema teve que ser feito de forma semi-manual. CNN irá automatizar o projeto de todo o sistema.

Rede neural convolucional (CNN) em Tensorflow/Keras

1 Introdução

Relembrando, na apostila *classif-ead*, classificamos o conjunto MNIST de dígitos manuscritos usando algoritmos de aprendizado clássicos, obtendo ≈2,5% de erro com "vizinho mais próximo" e 1,9% de erro com SVM. Depois, na apostila densakeras-ead, obtivemos ≈1,61% de erro usando "rede neural densa" em Keras. Nunca conseguimos obter taxa de erro substancialmente menor que 1,6%. Parece que, usando algoritmo de aprendizado clássico, não há como obter uma taxa de erro substancialmente menor que 1,6%.

O que tem de errado com todos esses algoritmos clássicos? Todos ignoram a relação espacial entre os pixels. Todos eles convertem a imagem 2D num vetor 1D ("flatten") antes processá-la. Por exemplo, convertem uma imagem com 28×28 pixels em um vetor de 784 elementos; ou uma imagem 14×14 em um vetor com 196 elementos (figura F1). Assim, esses algoritmos não conseguem distinguir um par de pixels vizinhos de um par de pixels distantes.

CNN consegue levar em consideração as relações de vizinhança entre os pixels. Nesta apostila, usando CNN, vamos atingir erro de 0,37%.

Nota: O aluno Guilherme Nicolino (2024) usou deskew para diminuiu o erro de 1.98% para 1.46% (usando SVM de SkLearn. Adicionando HOG aos atributos, diminuiu o erro de 1.46% para 0.87% (SVM de SkLearn). O cropping por bounding box diminuiu o erro de 0.87% para 0.81%. ~/psi/SistInt241/solucao_aluno. HOG leva em consideração as relações de vizinhança entre os pixels.

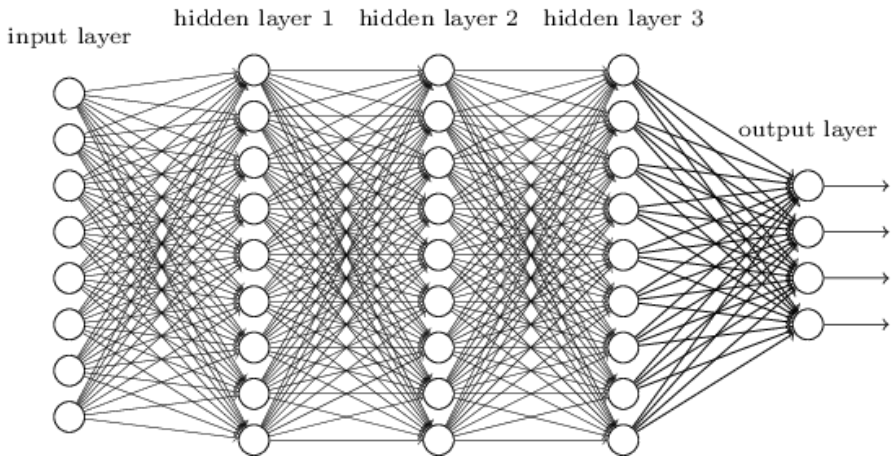


Figura F1: Uma rede neural convencional não leva em consideração as relações espaciais dos pixels, pois imagem 2-D é convertida em vetor 1-D antes de entrar na rede [figura de Nielsen].

Nota: Vision Transformer (ViT) converte uma imagem numa sequência de patches com informação de posição. ViT é melhor que CNN para datasets muito grandes. Para datasets pequenos (caso mais comum), CNN é melhor que ViT. Além disso, CNN é computacionalmente mais rápido que ViT.

Relembre como funciona o algoritmo Viola-Jones que detecta faces humanas (apostila “classifead”). Um algoritmo que classifica uma janela em face/não-face não dá bom resultado se alimentá-lo diretamente com os valores dos pixels da janela, pois a quantidade de atributos é muito grande. Assim, os autores utilizaram um conjunto de filtros lineares “tipo Haar” para extrair os atributos úteis e alimentaram o algoritmo de aprendizado com as saídas desses filtros. Figura 1a mostra duas convoluções (entre muitas) usadas para distinguir faces humanas das não-faces. Executando mais de 6000 filtros diferentes e concatenando as saídas, foi possível classificar se uma janela contém (ou não) um rosto humano.

O primeiro filtro utiliza o fato de que a região dos olhos é mais escura do que a região das bochechas. O segundo filtro utiliza o fato de que a região dos olhos é mais escura do que a região entre os olhos. O computador aprende que uma face humana precisa ter dois olhos, um nariz e uma boca.

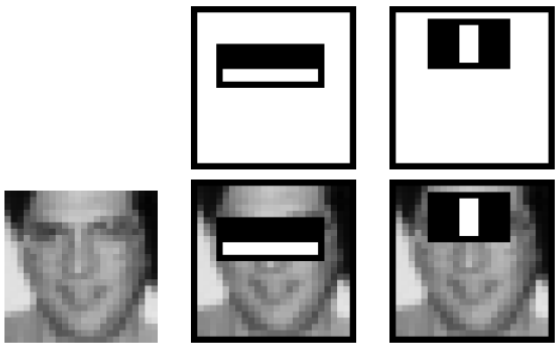


Figura 1: Duas (das 6061) convoluções necessárias para detectar face (figura de [Viola2001]).

Seria possível usar uma ideia semelhante para extrair bons atributos de MNIST e melhorar a classificação? Para identificar um dígito “3”, talvez possa usar “template matchings” que detectem as pontas de retas voltadas para esquerda, as linhas verticais e horizontais (pontos vermelhos na figura 2).

Depois, poderia detectar formas “ɔ” (contém) fazendo uma segunda camada de convolução que agrupa as detecções da primeira camada (pontos azuis na figura 2).

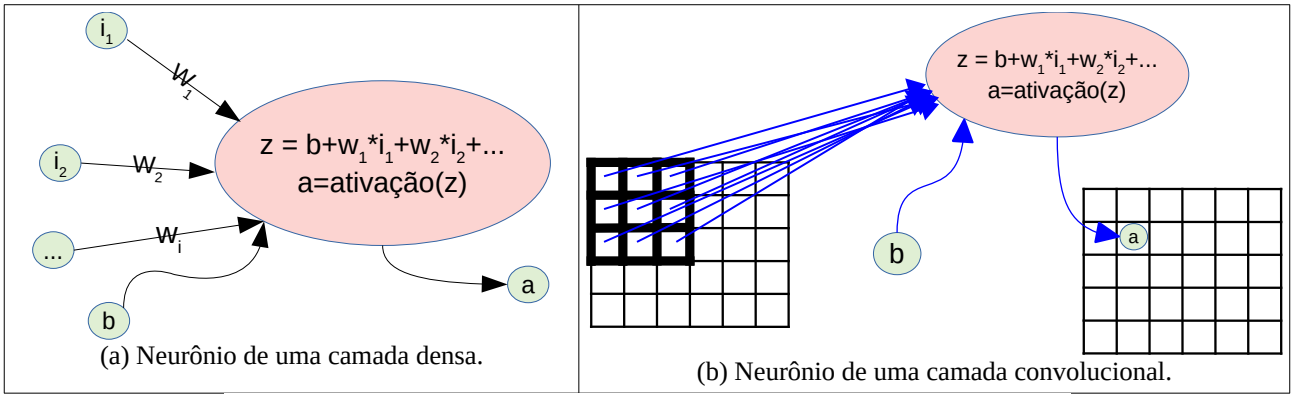
Por fim, poderia dizer que são necessárias duas formas “ɔ”, colocadas uma em cima da outra, para caracterizar o dígito “3”. CNN utiliza esta ideia.

imagem a classificar	Pontas voltadas para esquerda	Linhas verticais	Linhas horizontais	Duas formas “ɔ” caracterizam “3”

Figura 2: Seria possível usar convoluções para melhorar a classificação de MNIST?

É muito difícil projetar manualmente os filtros adequados para extrair bons atributos e concatená-los adequadamente. Usando CNN, o algoritmo de aprendizado escolhe para nós, automaticamente, os filtros adequados.

Para que uma rede neural projete automaticamente uma convolução, vamos substituir o neurônio que recebe as entradas $\{i_1, i_2, ...\}$ e gera uma saída a (figura 3a) pelo neurônio que efetua uma convolução numa imagem de entrada produzindo um atributo de saída (figura 3b).



(a) Neurônio de uma camada densa.

(b) Neurônio de uma camada convolucional.

Figura 3: Neurônios de camada densa e de camada convolucional.

Figura 4 mostra a aplicação de um neurônio convolucional 5×5 numa imagem 28×28. Durante o treino, CNN procura 25 pesos e 1 viés (26 parâmetros) que ajudam a classificar a imagem de entrada em dígitos 0, ..., 9. Depois de treinado, os mesmos 26 parâmetros serão aplicados para todos os pixels, varrendo a imagem. O neurônio convolucional gerará uma imagem de saída (conhecido como mapa de atributos).

As saídas dos filtros passam por uma função de ativação não-linear, por exemplo, “relu” ou “sigmoide”. Assim como a função *matchTemplate* do OpenCV, a saída (24×24) é menor do que a entrada (28×28) se trabalhar no modo “valid”. Se trabalhasse no modo “same” a saída teria o mesmo tamanho que a entrada.

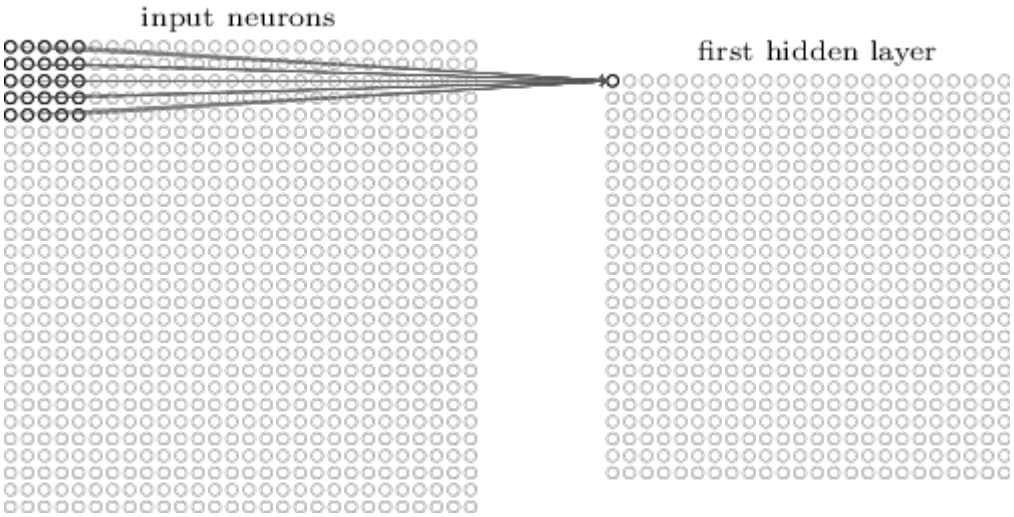


Figura 4: CNN possui filtro linear nas primeiras camadas [de Nielsen].

Na prática, em vez de um, vários filtros diferentes serão projetados pelo algoritmo de retro-propagação. A figura 5 mostra o projeto e a aplicação de 3 filtros diferentes que produzem 3 mapas de atributos.

Como CNN consegue projetar os filtros adequados para realizar uma certa tarefa de visão computacional? Ela primeiro inicializa aleatoriamente todos os parâmetros dos filtros (veja Anexo A.1 da apostila densakeras-ead). Depois, efetua repetidamente retro-propagação, onde cada parâmetro é modificado um pouco de cada vez para que a função custo (diferença entre a saída desejada e a obtida) diminua.

Nota: Quem escolhe o número de camadas, a quantidade e os tamanhos dos filtros é (normalmente) o ser humano.

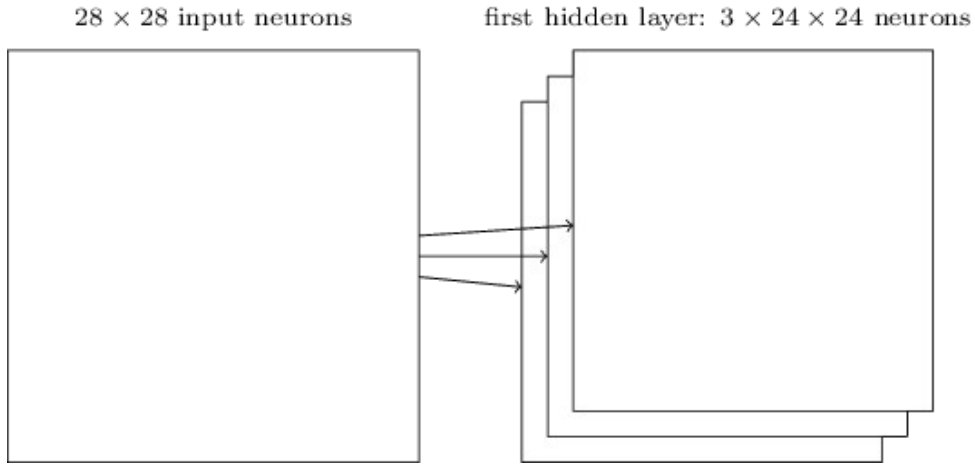


Figura 5: CNN projeta e aplica vários filtros lineares diferentes na imagem de entrada.

As saídas dos filtros (depois de passar pela função de ativação) normalmente alimentam camadas de agrupamento (“pooling” - figura 6). Estas camadas reduzem a dimensionalidade da imagem. Para isso, CNN divide a imagem em blocos e calcula o máximo (max-pooling) ou a média (average-pooling) dentro de cada bloco.

A intuição ao fazer max-pooling para diminuir a resolução é que não interessa exatamente onde fica uma ponta de reta do dígito “3”, desde que tenha uma ponta de reta numa certa região da imagem.

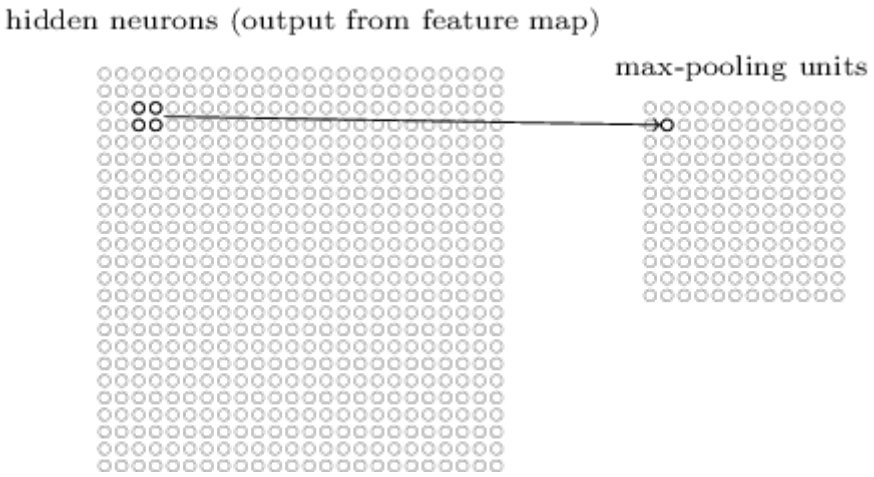


Figura 6: Max-pooling divide imagem em blocos (por exemplo 2×2) e calcula o maior elemento em cada bloco. Extraído de [Nielsen].

Muitas bibliotecas (como Keras e PyTorch) permitem especificar o passo (stride) de pooling. Se stride for igual ao tamanho do bloco (por exemplo, stride de 2 para bloco 2×2), será como se dividisse imagem em blocos e calculasse máximo/média em cada bloco. Se stride for igual a 1, pooling será aplicado pixel a pixel, como num filtro janela móvel.

As saídas das camadas de agrupamento (pooling) entram em outras camadas convolucionais. Esta segunda camada convolucional combina as ativações de vários filtros da primeira camada convolucional, procurando descobrir padrões mais complexos. Na figura 2, a segunda camada busca o símbolo “contém” (“כ”), procurando o conjunto formado por duas pontas de reta, duas linhas horizontais e uma linha vertical.

O que escrevo é somente uma explicação intuitiva de como CNN funciona. Na realidade, ela procura padrões visuais que não possuem significados muito intuitivos.

As camadas convolucionais e “max-pooling” (parte vermelha da figura 7) extraem as características a serem usadas pelo classificador final, tipicamente uma rede neural com camadas densas (parte azul da figura 7).

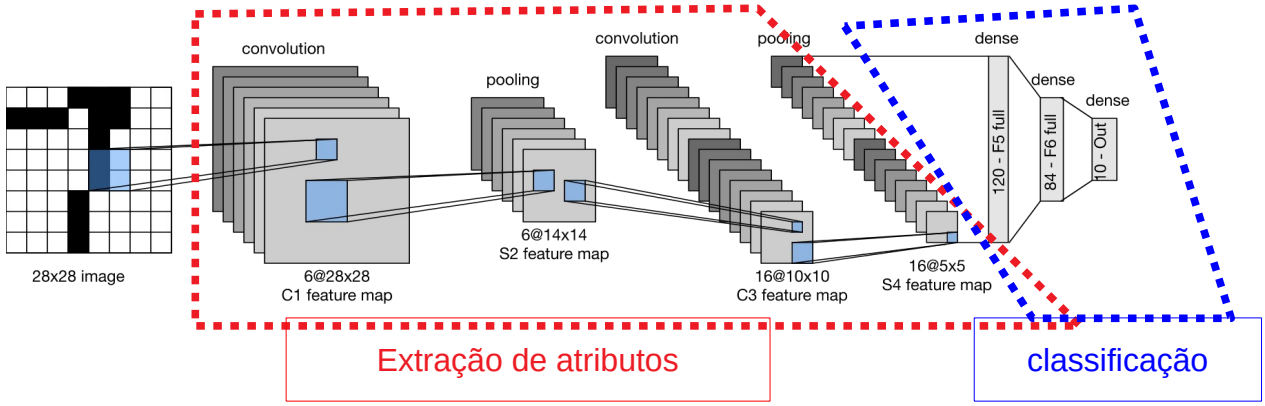


Figura 7: Modelo “inspirado em LeNet” para reconhecer dígitos MNIST é composta de duas partes: extração de atributos (vermelho) e classificação (azul). Filtros convolucionais são 5×5. Sub-amostragem (max_pooling) é feita usando janela 2×2 com passo (stride) 2.

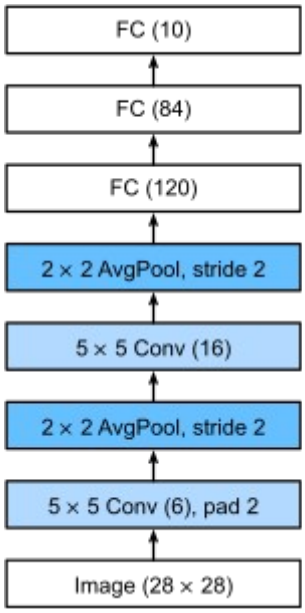


Figura F: LeNet-5 original com taxa de acerto de 98.4% no MNIST (Wikipedia).

2 Classificação de MNIST por CNN

O programa 1 (cnn1.py) classifica os dígitos de MNIST usando uma rede neural convolucional em Keras. O modelo de rede, inspirado na rede LeNet [LeCun1989a, LeCun1989b, Figs. 7 e F], está na figura 8.

Linhas 11-20: O programa faz a leitura do MNIST e converte os valores dos pixels para o intervalo de -0.5 a +0.5.
Nota: Poderia testar se o resultado melhora se normalizar a entrada.

Linhas 15 e 16: Como antes, os rótulos de saída (AY e QY com números entre 0 e 9) são convertidos em vetores de categorias (one-hot-encoding, AY2 e QY2) para alimentar a rede neural com 10 saídas.
Por exemplo, 3 é convertido em vetor [0001000000]. Não precisaria fazer esta conversão se usar função de perda *sparse_categorical_crossentropy*.

Linhas 21-22: O programa converte os tensores AX e QX de dimensões (60000, 28, 28) e (10000, 28, 28) em tensores de dimensões (60000, 28, 28, 1) e (10000, 28, 28, 1). Isto é necessário pois a camada Conv2D espera receber tensores no formato 28×28×1, com a última dimensão indicando número de bandas de cor.
No nosso caso, o número de bandas é 1, pois as imagens são em níveis de cinza. Se a imagem de entrada fosse colorida, o número de bandas seria 3.

Linhas 24-32: A arquitetura da rede é especificada aqui.

Linhas 25-26: A primeira camada convolucional recebe uma image 28×28 e aplica 20 convoluções 5×5, gerando 20 mapas de atributos 24×24.

Linha 27: A camada max-pooling com bloco 2×2 reduz a resolução espacial dos 20 mapas de 24×24 para 12×12.

Linha 28: A segunda camada convolucional recebe 20 mapas 12×12 e aplica 40 convoluções 5×5, gerando 40 mapas 8×8.
Nota: Uma camada convolucional 5×5 aplica, na verdade, convolução 5×5×*n*, onde *n* é o número de bandas da imagem (ou mapa de atributo) de entrada.

Linha 29: A resolução do mapa de atributo é reduzida de 8×8 para 4×4 por max-pooling 2×2. Neste ponto, CNN extraiu 4×4×40 atributos que julgou serem úteis para classificar os dígitos.

Linha 30: Esses atributos passam pela camada “flatten” que converte tensor 4×4×40 em vetor com 640 elementos.

Linhas 31 e 32: Esses 640 atributos irão alimentar uma rede neural densa responsável pela classificação, com duas camadas com 200 e 10 neurônios.

Linhas 34-36: O programa imprime a arquitetura da rede.

Linhas 38-39: O programa “compila” a rede especificada em Keras para ser executada em Tensorflow.

Linha 42: O programa faz o treinamento, onde *batch_size*=100 indica que será feita retro-propagação usando lotes com 100 amostras e o treino será repetido por 30 épocas.

Linhas 45-48: O programa aplica o modelo treinado nos dados de teste, calculando a taxa de erro de teste.

Linha 55: O modelo treinado é gravado no arquivo cnn1.keras.

```
 1 #cnn1.py - pos2021
 2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
 3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
 4 import tensorflow.keras as keras
 5 from tensorflow.keras.datasets import mnist
 6 from tensorflow.keras.models import Sequential
 7 from tensorflow.keras.layers import Dropout, Conv2D, MaxPooling2D, Dense, Flatten
 8 from tensorflow.keras import optimizers
 9 import numpy as np; import sys; import os; from time import time
10
11 (AX, AY), (QX, QY) = mnist.load_data() # AX [60000,28,28] AY [60000,]
12 AX=255-AX; QX=255-QX
13
14 nclasses = 10
15 AY2 = keras.utils.to_categorical(AY, nclasses) # 3 -> 0001000000
16 QY2 = keras.utils.to_categorical(QY, nclasses)
17
18 nl, nc = AX.shape[1], AX.shape[2] #28, 28
19 AX = (AX.astype('float32') / 255.0)-0.5 # -0.5 a +0.5
20 QX = (QX.astype('float32') / 255.0)-0.5 # -0.5 a +0.5
21 AX = np.expand_dims(AX,axis=3) # AX [60000,28,28,1]
22 QX = np.expand_dims(QX,axis=3)
23
24 model = Sequential() # 28x28
25 model.add(Conv2D(20, kernel_size=(5,5), activation='relu',
26   input_shape=(nl, nc, 1) )) #24x24x20
27 model.add(MaxPooling2D(pool_size=(2,2))) #12x12x20
28 model.add(Conv2D(40, kernel_size=(5,5), activation='relu')) #8x8x40
29 model.add(MaxPooling2D(pool_size=(2,2))) #4x4x40
30 model.add(Flatten()) #640
31 model.add(Dense(200, activation='relu')) #200
32 model.add(Dense(nclasses, activation='softmax')) #10
33
34 from tensorflow.keras.utils import plot_model
35 plot_model(model, to_file='cnn1.png', show_shapes=True);
36 model.summary()
37
38 opt=optimizers.Adam()
39 model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
40
41 t0=time()
42 model.fit(AX, AY2, batch_size=100, epochs=30, verbose=2)
43 t1=time(); print("Tempo de treino: %.2f s"%(t1-t0))
44
45 score = model.evaluate(QX, QY2, verbose=False)
46 print('Test loss: %.4f'%(score[0]))
47 print('Test accuracy: %.2f %%'%(100*score[1]))
48 print('Test error: %.2f %%'%(100*(1-score[1])))
49
50 t2=time()
51 QP2=model.predict(QX); QP=np.argmax(QP2,1)
52 t3=time(); print("Tempo de predicao: %.2f s"%(t3-t2))
53 nerro=np.count_nonzero(QP-QY); print("nerro=%d"%(nerro))
54
55 model.save('cnn1.keras')
```

Programa 1: Classificação de MNIST usando rede neural convolucional.
https://colab.research.google.com/drive/1_ZvdBAuj2fEuaRF2Y7xb_ZKw-INvmxQ3?usp=sharing
~/haepi/deep/keras/conv/cnn3/cnn1.py

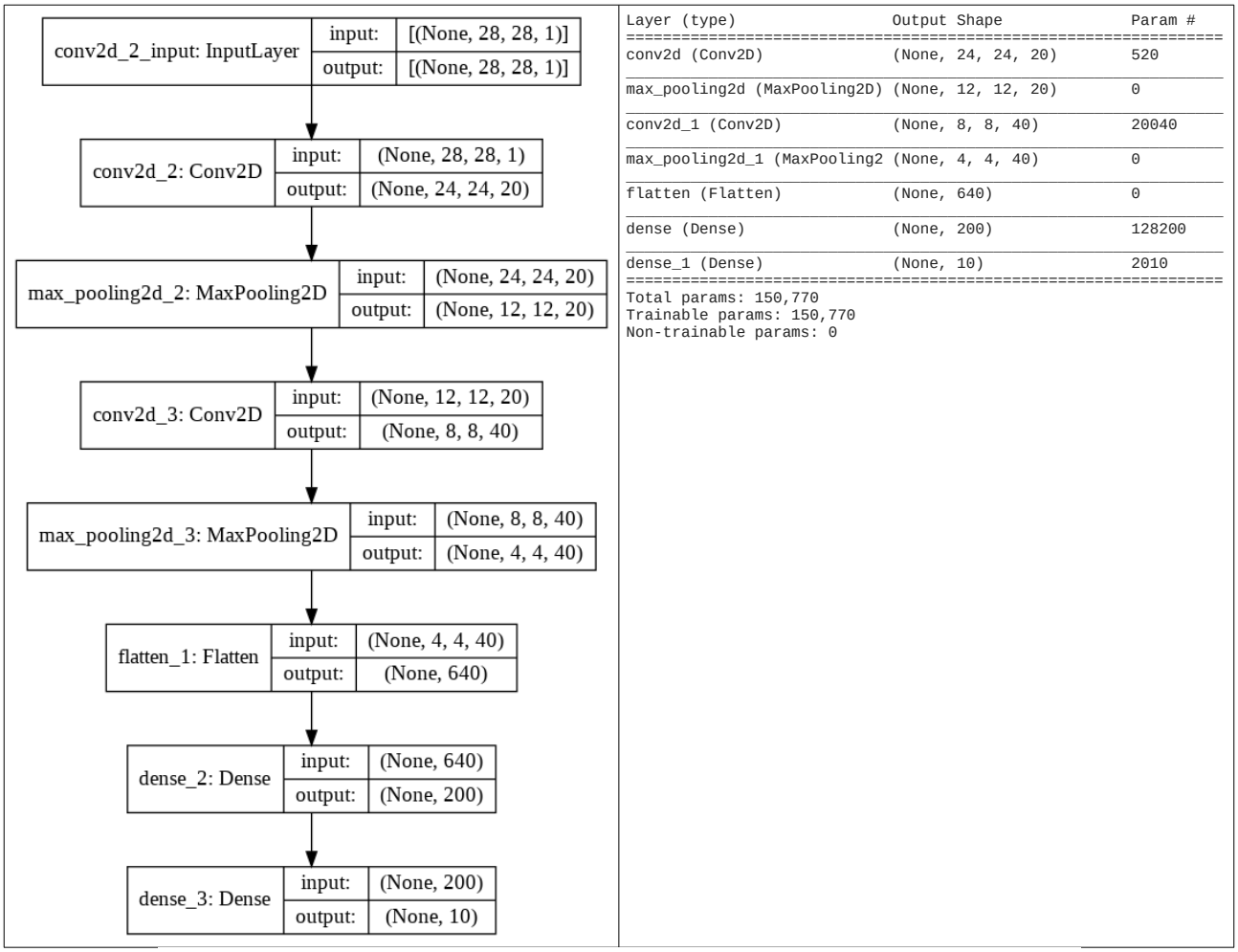


Figura 8: Modelo de rede “tipo LeNet” do programa cnn1.py (programa 1).

A saída obtida, rodando este programa, é:

```
Epoch  1/30 - 2s - loss: 0.1573 - accuracy: 0.9546
Epoch  5/30 - 1s - loss: 0.0161 - accuracy: 0.9946
Epoch 10/30 - 1s - loss: 0.0079 - accuracy: 0.9972
Epoch 15/30 - 1s - loss: 0.0043 - accuracy: 0.9986
Epoch 20/30 - 1s - loss: 0.0019 - accuracy: 0.9994
Epoch 25/30 - 1s - loss: 0.0026 - accuracy: 0.9993
Epoch 30/30 - 1s - loss: 0.0036 - accuracy: 0.9990
Tempo de treino: 36.94 s
Test loss: 0.0437
Test accuracy: 99.34 %
Test error: 0.66 %
Tempo de predicao: 0.55 s
nerro=66
```

Obtivemos taxa de erro de teste 0,66%, de longe a menor taxa que conseguimos até agora. O erro obtido está muito abaixo de 1,6%, a menor taxa de erro que tínhamos obtido usando algoritmos clássicos. Por outro lado, podemos ver que o custo e erro de treino (0,0036 e 0,1%) são muito menores que o custo e erro de teste (0,044 e 0,66%), indicando que há “overfitting”. O tempo de processamento foi 54s no computador local com GPU e 37s no Colab.

Nota: O ajuste excessivo ou overfitting é um comportamento indesejável de aprendizado de máquina que ocorre quando o modelo de aprendizado de máquina fornece previsões precisas para dados de treinamento, mas não para novos dados. [https://aws.amazon.com/pt/what-is/overfitting/]

Nota: cada vez que faz o treino, será gerado um modelo com uma taxa de erro um pouco diferente, pois os parâmetros da rede são inicializados aleatoriamente.

Vamos visualizar as 10 primeiras linhas da saída QP2 executando os seguintes comandos:

```
model=keras.models.load_model("cnn1.keras")
QP2=model.predict(QX)
np.set_printoptions(precision=2)
print(QP2[0:10])
```

```
[[3.50e-23 1.44e-16 5.68e-21 5.67e-17 1.75e-14 3.39e-20 5.36e-30 1.00e+00 3.00e-22 2.87e-15]
 [1.52e-18 1.31e-23 1.00e+00 1.08e-28 1.17e-25 6.44e-35 3.19e-20 4.00e-29 3.46e-27 1.27e-30]
 [1.52e-10 1.00e+00 2.29e-13 1.01e-19 7.70e-14 1.03e-09 2.39e-17 7.15e-10 2.62e-12 7.38e-18]
 [1.00e+00 1.11e-24 5.64e-17 1.20e-19 2.13e-26 3.92e-19 1.72e-11 2.37e-23 1.77e-15 1.50e-18]
 [2.31e-26 1.72e-15 2.76e-25 2.21e-23 1.00e+00 1.32e-17 1.05e-18 4.82e-22 1.06e-17 1.42e-15]
 [3.17e-15 1.00e+00 6.40e-17 1.34e-26 5.49e-17 6.40e-16 8.88e-24 1.47e-11 2.14e-17 3.77e-22]
 [1.04e-30 5.38e-17 2.97e-25 1.59e-27 1.00e+00 1.52e-19 1.35e-27 2.07e-22 1.19e-14 1.43e-15]
 [2.80e-18 2.32e-16 1.43e-12 6.50e-15 7.74e-11 3.73e-15 4.20e-23 3.19e-15 6.47e-14 1.00e+00]
 [1.39e-17 3.83e-16 3.03e-24 1.38e-24 7.47e-21 1.00e+00 3.75e-09 4.64e-26 2.72e-09 1.85e-18]
 [7.67e-26 4.19e-24 1.16e-23 2.54e-16 5.12e-11 1.96e-19 3.87e-30 5.81e-15 1.51e-18 1.00e+00]]
```

A saída imita “probabilidades”, pois estamos usando ativação “softmax” na última camada da rede e função custo “categorical_crossentropy”. Todas as saídas estão entre 0 e 1. Além disso, se somar as 10 saídas de cada linha, sempre dá 1.0.

Exercício: Execute programa 1 (cnn1.py) para obter a rede cnn1.keras. Anote a acuracidade obtida.

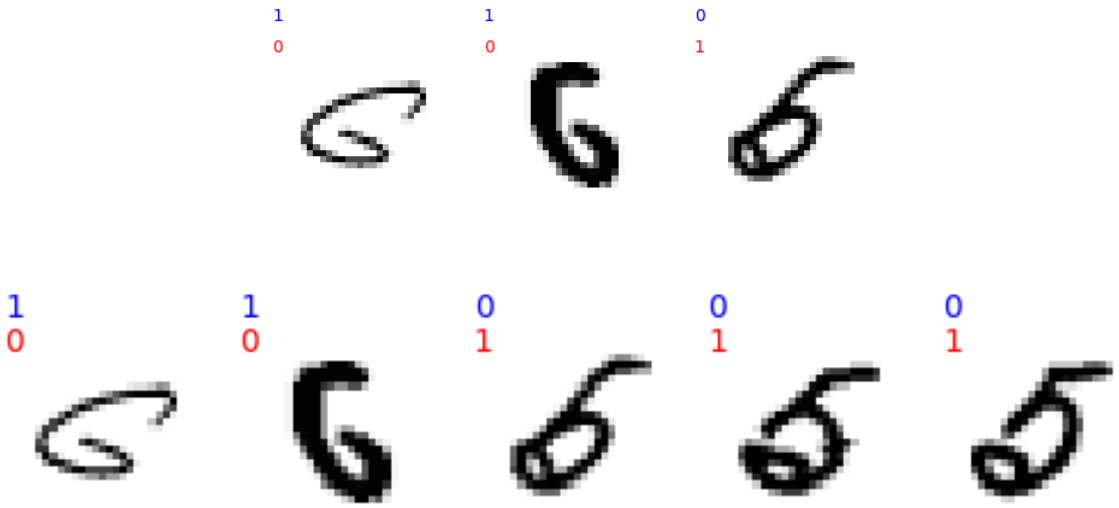
Exercício: Troque o pré-processamento para usar *Normalization* layer, como fizemos nos programas mlp1.py e mlp2.py da apostila densakeras-ead.

Exercício: Faça diferentes alterações no programa 1 para atingir taxa de erro menor que 0,66%, sem fazer “data augmentation”. Isto não é difícil, pois já consegui taxas de erros menores usando parâmetros um pouco diferentes. Na verdade, teria que repetir o treino algumas vezes e calcular a média, pois cada treino resulta numa taxa de erro diferente. Algumas alterações possíveis são: (a) Mudar a estrutura da rede. (b) Mudar número e tamanho dos filtros. (c) Mudar função de ativação. (d) Mudar o otimizador e/ou os seus parâmetros. (e) Mudar o tamanho do batch. (f) Mudar o número de épocas. Descreva (como comentários dentro do seu programa .cpp ou .py) a taxa de erro que obteve, o tempo de processamento, as alterações feitas e os testes que fez para chegar ao seu programa com baixa taxa de erro.

Exercício: Modifique programa 1 para que a rede extraia as características das imagens (os valores entre a parte vermelha e azul da figura 7). Utilize as características das imagens de treinamento para treinar algum algoritmo de aprendizagem de máquina clássica (como Boost, decision-tree, etc). Verifique a taxa de acerto classificando as imagens de teste.

Exercício: O programa 1 utiliza 60000 imagens de treinamento. Modifique-o para que utilize apenas 6000 primeiras imagens de treino, isto é 10% dos dados de treino iniciais. Não tem problema se alguns dígitos possuem um pouco mais exemplos de treinamento do que outros. Ajuste o programa para obter a menor taxa de erro possível quando o programa classifica as 10000 imagens de teste. Qual foi a taxa de erro obtida? O seu vídeo deve mostrar claramente que está utilizando apenas 6000 imagens de treino, imprimindo AX.shape[0] e AY.shape[0]. Também deve deixar claro no vídeo a taxa de erro obtida.
Depois, treine com 600 imagens de treino.
Nota: As taxas de erro típicas são 1,9% para 6000 imagens e 8% para 600 imagens.
Nota: Acrescentar camadas dropout diminui um pouco os erros.

Exercício: Modifique o programa 1 para obter um modelo que classifica somente os dígitos “5” e “6”. Isto é, você deve treinar o modelo fornecendo somente os exemplos de treino de “5” e “6” do MNIST. Represente dígito “5” com rótulo 0 e dígito “6” com rótulo 1. Depois, faça o teste com os 1850 dígitos “5” e “6” de teste do MNIST. MNIST não é balanceado, de forma que a quantidade exemplos de teste de cada classe não dá exatamente 1000. Imprima a taxa de erro de teste (tipicamente 0,16%-0,38%, com 3-7 erros cometidos ao classificar 1850 dígitos). Imprima todos os dígitos classificados incorretamente, obtendo uma figura como exemplos abaixo (se a quantidade de erros for 3 ou 5).



onde o rótulo azul é a classificação correta e o rótulo vermelho é a classificação dada pelo algoritmo (0=“5” e 1=“6”). O trecho do programa abaixo imprime imagem como na figura acima (QEX, QEY e QEP são os exemplos de teste classificados incorretamente respectivamente imagens, rótulos verdadeiros e rótulos fornecidos pelo classificador).

```
f = plt.figure()
for i in range(QEX.shape[0]):
  f.add_subplot(1,QEX.shape[0],i+1)
  plt.imshow( QEX[i], cmap="gray")
  plt.axis("off");
  plt.text(0,-3,QEY[i],color="b")
  plt.text(0, 2,QEP[i],color="r")
plt.savefig("QE.png")
plt.show()
```

onde QEX é o tensor “(3, 28, 28, 1) float32” com as imagens classificadas incorretamente (supondo que há 3 erros), QEY é o vetor “(3,) uint8” com os rótulos verdadeiros e QEP é o vetor “(3,) uint8” com as classificações dadas pelo modelo.

[Nota: A solução privada está em ~/deep/keras/conv/17/56.py]

Exercício: Fashion_MNIST é um BD muito semelhante à MNIST. Consiste de 60000 imagens de treino e 10000 imagens de teste 28×28, em níveis de cinza, com as categorias:

```
categories=["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
categorias=["Camiseta", "Calça", "Pulôver", "Vestido", "Casaco", "Sandália", "Camisa", "Tênis", "Bolsa", "Botins"]
```

Esse BD pode ser carregado com os comandos:

```
from keras.datasets import fashion_mnist
(AX, AY), (QX, QY) = fashion_mnist.load_data()
```

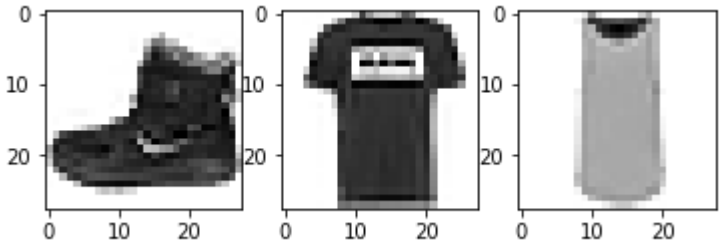
Modifique o programa 1 (cnn1.py) para que classifique fashion_mnist, **usando a função de perda *sparse_categorical_crossentropy*** em vez de *categorical_crossentropy* (não vai precisar mais calcular *one-hot-encoding* com função *to_categorical*, veja anexo A.6.b2 da apostila densakeras-ead). Para que o treino não demore excessivamente, vamos manter epochs=30 e batch_size=100. Ajuste o programa para obter a menor taxa de erro possível (sem alterar epochs=30 e batch_size=100). Qual foi a taxa de erro obtida? Deixe claro no vídeo a taxa de erro obtida. Mostre as 20 primeiras imagens de teste do BD na tela e no vídeo, com as classificações verdadeiras e geradas pelo algoritmo, semelhante à figura abaixo:

Para mostrar as 20 imagens:

```
from matplotlib import pyplot as plt
categorias=["Camiseta", "Calça", "Pulôver", "Vestido", "Casaco", "Sandália", "Camisa", "Tênis", "Bolsa", "Botins"]
f = plt.figure()
for i in range(20):
  f.add_subplot(4,5,i+1)
  plt.imshow( QX[i,:,:,0], cmap="gray", vmin=-0.5, vmax=+0.5)
  plt.axis("off");
  plt.text(0,-3,categorias[QY[i]],color="b")
  plt.text(0, 2,categorias[QP[i]],color="r")
plt.savefig("nomefigura.png")
plt.show()
```

Para mostrar 3 imagens em níveis de cinza lado a lado:

```
from matplotlib import pyplot as plt
f = plt.figure()
f.add_subplot(1,3,1)
plt.imshow(img1,cmap="gray")
f.add_subplot(1,3,2)
plt.imshow(img2,cmap="gray")
f.add_subplot(1,3,3)
plt.imshow(img3,cmap="gray")
plt.show(block=True)
```



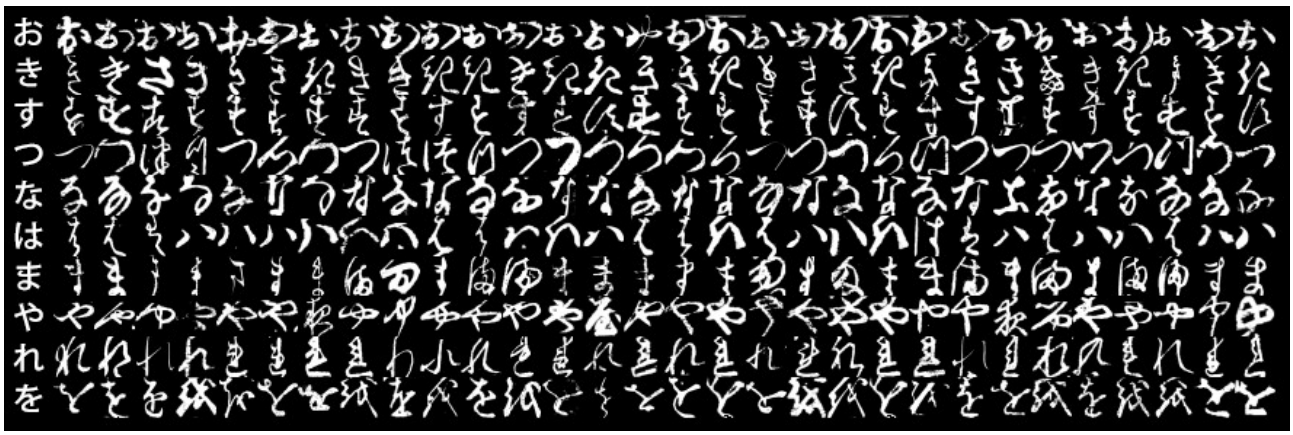
Solução privada: Rodando o programa fashion_mnist1.py, obtemos: Tempo de treino: 45.50 s Test error: 9.74 %
https://colab.research.google.com/drive/1lThVXPEcn3j7pXUSXfaLPyhxLgs21K3B

Solução privada: Rodando o programa fashion_mnist3.py com data augmentation, obtemos: Tempo de treino: 743.64 s Test error: 7.12 %
https://colab.research.google.com/drive/1CGdw6UnyPsJVY0NZxgs9tQ6tOss1-Kt9#scrollTo=a4aVP3czWi1v

Solução privada: Rodando o programa fashion_mnist_sparse1.py, obtemos: Tempo de treino: 72.76 s Test error: 9.53 %
https://colab.research.google.com/drive/1ghQJbFoR27lgHp0u7we5P64ItlWHV94P

Nota: Resolvendo este problema usando rede densa, obtém Test error: 11.22 %
https://colab.research.google.com/drive/18fKGaOLzIPTrylL8pw8xNagw1efee8NE?usp=sharing
Pelas características de BD, não há grande diferença de taxa de erro entre modelo denso e convolucional.

[PSI3471 aulas 11/12. Lição de casa #1/1. Vale 10.] O site https://github.com/rois-codh/kmnist traz Kuzushiji-MNIST, um conjunto de imagens de caracteres japoneses manuscritos distorcidos, que pode substituir diretamente MNIST, pois consiste de 60000+10000 imagens 28×28.



Para baixar esse conjunto, basta copiar e executar a célula Python fornecida em:
https://github.com/rois-codh/kmnist/blob/master/download_data.py
Quando executar a célula, faça as seleções como abaixo para baixar os arquivos no seu diretório default:

```
Please select a download option:
1) Kuzushiji-MNIST (10 classes, 28x28, 70k examples)
2) Kuzushiji-49 (49 classes, 28x28, 270k examples)
3) Kuzushiji-Kanji (3832 classes, 64x64, 140k examples)
> 1
Please select a download option:
1) MNIST data format (ubyte.gz)
2) NumPy data format (.npz)
> 2
```

Depois, para carregar o BD, basta executar:

```
AX = np.load("kmnist-train-imgs.npz")['arr_0']
AY = np.load("kmnist-train-labels.npz")['arr_0']
QX = np.load("kmnist-test-imgs.npz")['arr_0']
QY = np.load("kmnist-test-labels.npz")['arr_0']
```

Classifique este banco de dados usando redes nerais densa e convolucional.

- Use a função de perda *sparse_categorical_crossentropy* em vez de *categorical_crossentropy*. Fazendo isso, não será necessário mais calcular *one-hot-encoding* com função *to_categorical* (veja anexo A.6.b2 da apostila densakeras-ead).
- Não calcule *softmax*, pois *softmax* introduz instabilidade numérica. Para não usar nenhuma função de ativação, especifique *activation="linear"* ou *activation=None*. Depois, use a função de perda *sparseCategoricalCrossentropy* com parâmetro *from_logits=True*. Isto calculará a função de perda sem calcular *softmax*.

Pode usar como modelos os programas:

- Densa: Programa 4 da apostila densakeras-ead (mlp1.py)
- Convolucional: Programa 1 desta apostila (cnn1.py)

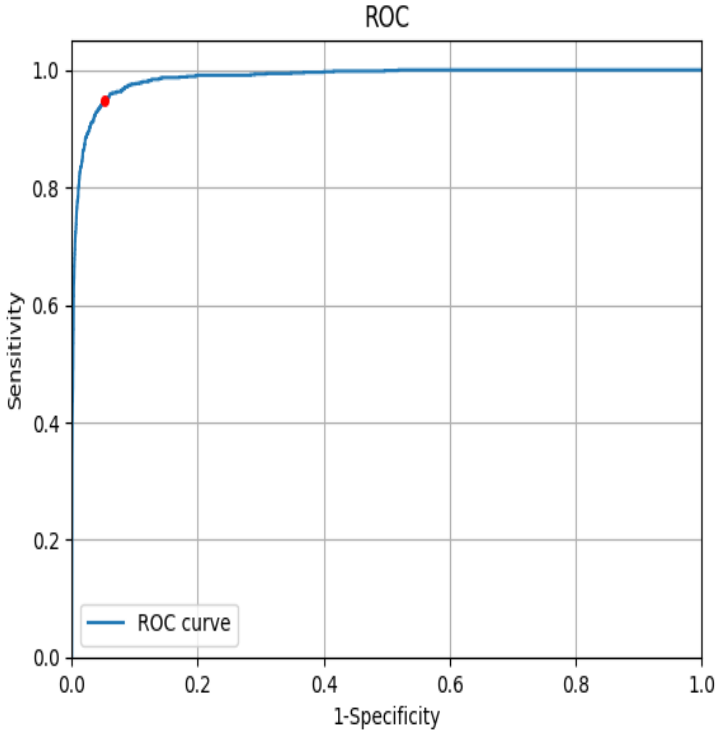
Para não gastar muito tempo treinando, execute durante apenas 30 épocas usando *batch_size*=100. Ajuste os programas para obter as menores taxas de erro possíveis. Compare as taxas de erros das redes densa e convolucional. Quem obtiver taxa erro baixa recebe nota mais alta.

Solução privada:
Densa: https://colab.research.google.com/drive/1MEBijyqaCw8eYUvi3QA_ii3N0xh722D-#scrollTo=a4aVP3czWi1v
Convolucional https://colab.research.google.com/drive/13wQwGNgh3Y-DXLplVPdA2CckwnITfGen#scrollTo=a4aVP3czWi1v
Densa erro de teste: 9.02 %. Convolucional erro de teste: 4.35 %

Exercício: Vamos supor que as imagens de pulôver do fashion-mnist representam exames de imagem de pacientes “doentes” e as demais representam exames de pacientes “sadios”. Escreva um programa que usa fasion-mnist e identifique se um paciente está “doente” ou “sadio”, classificando o seu exame em “pulôver” ou “não-pulôver”. Treine o seu modelo com as imagens de treino do fashion-mnist e faça a predição usando as imagens de teste. Adote limiar=0.5 e calcule sensibilidade e especificidade. Plote curva ROC e calcule AUC. Calcule acuracidade no ponto de EER (equal error rate).

https://en.wikipedia.org/wiki/Sensitivity_and_specificity
https://en.wikipedia.org/wiki/Receiver_operating_characteristic
https://www.w3schools.com/python/python_ml_auc_roc.asp
https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

A curva ROC obtida deve ser semelhante a:



Nota: Minha solução dá sensibilidade 90,20% e especididade=97,14% usando o limiar 0,5.
EER=5,23%
Solução privada:
https://colab.research.google.com/drive/1wtjaLvzKBfGy_jK5YBs-EOLrF_tnMo_i#scrollTo=YXec8kUmpTl-

3 Visualização dos filtros

Para entender o que acontece dentro de uma CNN, é necessário visualizar os filtros da rede e as ativações das camadas intermediárias.

A figura 9 mostra os aspectos dos 20 filtros 5×5 da primeira camada da rede (estou usando o modelo *cnn1.h5* com taxa de erro 0,68% que deixei disponível no site http://www.lps.usp.br/hae/apostila/cnn1.h5).

Nota: cnn1.h5 foi treinada com os pixels das imagens indo de 0 a 1 (e não de -0.5 a +0.5). Além disso, a camada densa escondida tem 1000 neurônios (em vez de 200 como agora).

De um modo geral, CNN projetou filtros com características visuais pouco intuitivas. Porém, algumas delas possuem interpretações intuitivas. Podemos pensar nesses filtros como se executassem *template matching*.

Por exemplo, o filtro azul detecta (aproximadamente) as bordas esquerdas das retas verticais. As imagens filtradas por esse filtro na figura 11b mostram que, de fato, ele detecta as bordas esquerdas das retas verticais.

Outro exemplo, o filtro vermelho detecta (aproximadamente) as bordas inferiores das retas horizontais, o que pode ser confirmado olhando as saídas respectivas na figura 11b.

Normalmente, CNN não projeta filtros com interpretações visuais intuitivas. Tive que repetir o treino algumas vezes, até que aparecessem os filtros com características visuais interessantes.

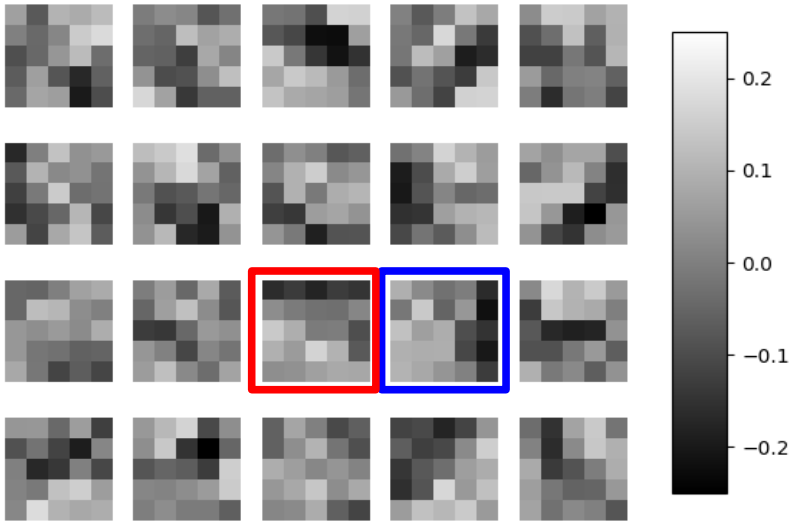


Figura 9: Os pesos das 20 convoluções 5×5 da primeira camada. Os filtros marcados em vermelho e azul detectam respectivamente retas horizontais e verticais.

O programa completo que visualiza os 20 filtros da primeira camada convolucional está abaixo.

```
 1 #plotafiltro2.py
 2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
 3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
 4 import tensorflow.keras as keras
 5 import numpy as np; import sys; import cv2
 6 from matplotlib import pyplot as plt
 7
 8 url='http://www.lps.usp.br/hae/apostila/cnn1.h5'
 9 import os; nomeArq=os.path.split(url)[1]
10 if not os.path.exists(nomeArq):
11   print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
12   os.system("wget -U 'Firefox/50.0' "+url)
13 else:
14   print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
15
16 model=keras.models.load_model("cnn1.h5")
17
18 (filters, biases) = model.get_layer(index=0).get_weights()
19 filters=np.squeeze( filters )
20 print(filters.shape)
21 filters2=np.empty( (filters.shape[2], filters.shape[0], filters.shape[1]) )
22 print(filters2.shape)
23 for i in range(filters.shape[2]):
24   filters2[i,:,:]=filters[:,:,i]
25
26 f = plt.figure()
27 for i in range(20):
28   f.add_subplot(4,5,i+1)
29   plt.imshow(filters2[i],vmin=-0.25, vmax=0.25, cmap="gray")
30   plt.axis('off')
31 plt.subplots_adjust(bottom=0.1, right=0.8, top=0.9)
32 cax = plt.axes([0.85, 0.15, 0.06, 0.7])
33 plt.colorbar(cax=cax)
34 plt.savefig("filtros0.png")
35 plt.show()
```

Programa: Mostra os filtros da primeira camada convolucional.
https://colab.research.google.com/drive/1Gab5WeuRYRtDjaFPEJYFPRxjQO53g2Kh?usp=sharing [~/deep/keras/conv/plotafiltro/plotafiltro2.py]

O trecho de código abaixo faz download do modelo *cnn1.h5* do meu site, se esse arquivo já não estiver no seu diretório default:

```
 8 url='http://www.lps.usp.br/hae/apostila/cnn1.h5'
 9 import os; nomeArq=os.path.split(url)[1]
10 if not os.path.exists(nomeArq):
11   print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
12   os.system("wget -U 'Firefox/50.0' "+url)
13 else:
14   print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
```

Para carregar o modelo *cnn1.h5* pré-treinado, use:

```
16 model=keras.models.load_model("cnn1.h5")
```

Nota: Nas versões antigas, Keras salvava o modelo com extensão ".h5". Na versão atual, Keras salva o seu modelo com extensão ".keras".

Para acessar os pesos e vieses dos filtros da primeira camada pelo índice da camada:

```
18 (filters, biases) = model.get_layer(index=0).get_weights()
```

Nota: É possível chamar o filtro pelo nome como abaixo. Porém, o nome do filtro fica mudando a cada execução, de forma que é melhor chamar pelo índice.

```
(filters, biases) = model.get_layer("conv2d_1").get_weights()
#Nome da primeira camada convolucional e' conv2d_1 ou conv2d ou ...
```

O tensor “filters” obtido como acima deve ter shape 5×5×1×20. Para poder visualizar os 20 filtros como imagens em níveis de cinza, devemos converter esse tensor para shape 20×5×5. O seguinte trecho de programa faz essa conversão.

```
19 filters=np.squeeze(filters) #Elimina dimensao de comprimento unitario
20
21 filters2=np.empty( (filters.shape[2], filters.shape[0], filters.shape[1]) )
22
23 for i in range(20):
24    filters2[i,:,:]=filters[:,:,i]
```

Depois, para visualizar os 20 filtros:

```
26 f = plt.figure()
27 for i in range(20):
28    f.add_subplot(4,5,i+1)
29    plt.imshow(filters2[i],vmin=-0.25, vmax=0.25, cmap="gray")
30    plt.axis('off')
31 plt.subplots_adjust(bottom=0.1, right=0.8, top=0.9)
32 cax = plt.axes([0.85, 0.15, 0.06, 0.7])
33 plt.colorbar(cax=cax)
34 plt.savefig("filtros0.png")
35 plt.show()
```

Exercício: Acrescente no programa 1 (*cnn1.py*) o trecho de programa que mostra os filtros projetados. Execute o programa modificado. Você consegue identificar alguns filtros com características visuais intuitivas?

Exercício: Escreva um programa que permite visualizar os filtros no final de cada época de treino, para poder acompanhar a evolução dos filtros durante o treino.

Exercício: Escreva um programa que classifica se o dígito é “3” ou não. Pode ser que apareçam filtros que detectam ponta de reta voltada para esquerda, reta horizontal e reta vertical.

Figura 10 mostra os 40 filtros 5×5 da segunda camada. Note que cada um desses 40 filtros será aplicado aos 20 mapas de atributos 12×12 da primeira camada, resultando em 40 mapas de atributos 8×8. Assim, a segunda camada convolucional consiste de 40 pesos 5×5×20 e não 40 pesos 5×5.

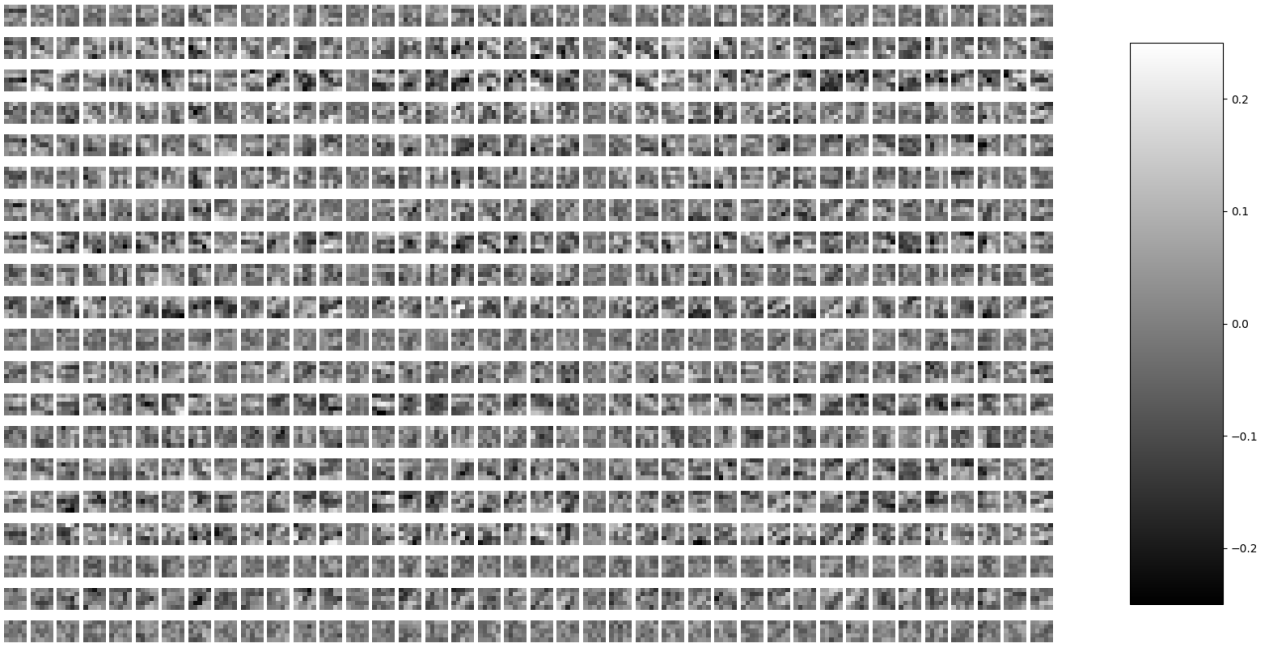


Figura 10: 40 filtros 5×5 da segunda camada. Cada coluna representa os pesos de um filtro. A quantidade de pesos de cada filtro é 5×5×20, pois cada uma matriz 5×5 diferente será aplicado a cada uma das 20 bandas do mapa de atributo de entrada. [Solução privada: ~/deep/keras/conv/plotafiltro/plotafiltro3.py]

Vamos pensar por que são 40 pesos 5×5×20 e não 40 pesos 5×5. No exemplo de detecção de dígito "3", é necessário juntar as correlações altas para "pontas de reta", "retas horizontais" e "retas verticais" sendo que cada uma destas correlações está num mapa de atributo diferente. Assim, usando filtros 5×5, é impossível achar local onde aparecem "2 pontas de reta", "2 retas horizontais" e "1 reta vertical" para formar o símbolo "ɔ". É necessário usar um filtro que possa olhar diversos mapas de ativação ao mesmo tempo. Usando filtros 5×5×20, é possível achar locais onde aparecem o símbolo "ɔ".

Exercício: Escreva um programa que permite visualizar os 40 filtros da segunda camada convolucional da rede cnn1.h5 http://www.lps.usp.br/hae/apostila/cnn1.h5 , como na figura 10. Se você usar *cnn1.h5* que deixei no site, deve visualizar exatamente os mesmos filtros da figura 10.
[solução privada em: ~/deep/keras/conv/plotafiltro/plotafiltro3.py]

4 Visualizar ativação

É possível visualizar a ativação dos neurônios de uma camada intermediária (por exemplo, após a primeira camada convolucional) durante a inferência, criando em Keras uma rede "lógica", "virtual", ou "artificial" como abaixo:

```
model=models.load_model("cnn1.h5")
intermediate_layer_model = Model(inputs=model.input,
                                 outputs=model.get_layer(index=0).output)
y = intermediate_layer_model.predict(x)
```

onde *x* é a imagem de um dígito no formato "(1, 28, 28, 1) float32" e *y* é a matriz de ativação.

Nota: Este truque de criar uma rede "virtual" para ter acesso às camadas intermediárias é usado com frequência em Keras. Esta operação é meio lenta e aparentemente não é uma simples atribuição, de forma que deve evitar chamá-la repetidamente dentro de um loop.

Nota: cnn1.h5 foi treinado com os pixels das imagens indo de 0 a 1 (e não de -0.5 a +0.5).

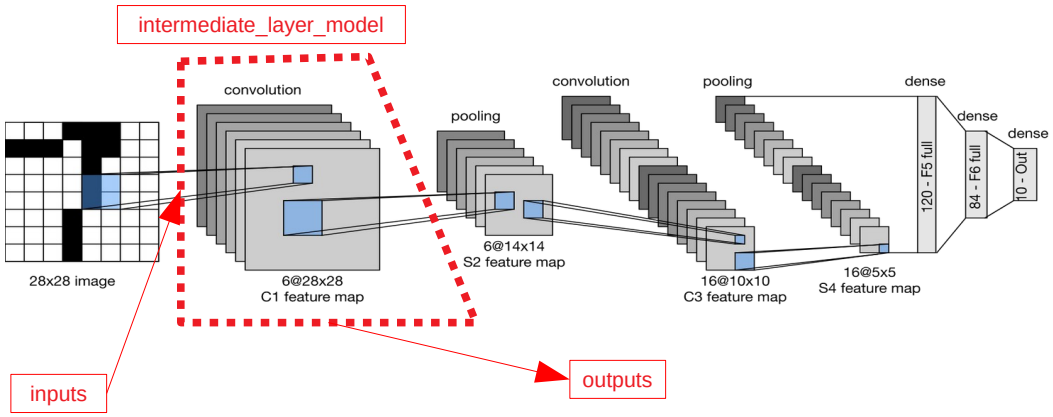


Figura: Rede "virtual" *intermediate_layer_model* foi criada para ter acesso aos mapas de atributos no interior da rede.

A figura 11 mostra nas 3 colunas:

(a) As imagens de entrada de alguns dígitos "1" e "3".

(b) As ativações da primeira camada convolucional, isto é, as saídas das 20 primeiras convoluções. Olhando juntamente com os filtros da figura 9, é possível concluir que as saídas observadas correspondem às saídas dos filtros.

(c) As ativações da segunda camada max_pooling, ou seja, as 40×4×4=640 atributos extraídos automaticamente pela CNN e que serão usadas pelas camadas densas para classificar os dígitos. Olhando essas características, é muito simples separar os dígitos "1" de "3". Basta olhar, por exemplo, para os dois primeiros blocos. Eles são completamente pretos para dígitos "1" mas está com bastante ativação para dígitos "3".

Exercício: Escreva um programa que lê as imagens dos dígitos da figura 11 e gera as 40 imagens 4×4 das ativações após o segundo max-pooling (como na figura 11c). As 4 imagens (2 dígitos "1" e 2 dígitos "3") estão no site [http://www.lps.usp.br/hae/apostila/convkeras.zip].
[Solução privada em ~/deep/keras/conv/ativacao/ativacao1.py]

Exercício: Explique por que as ativações e atributos das figuras 11b e 11c não podem ser negativos.

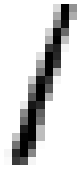
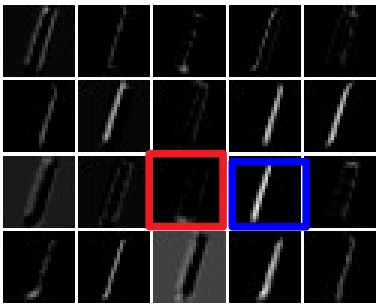
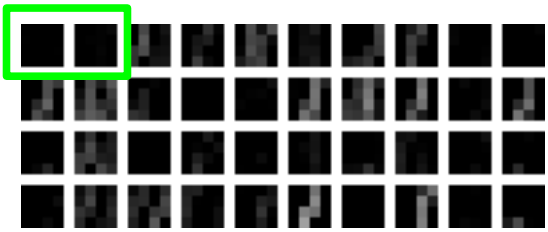
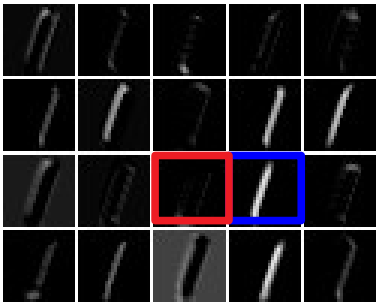
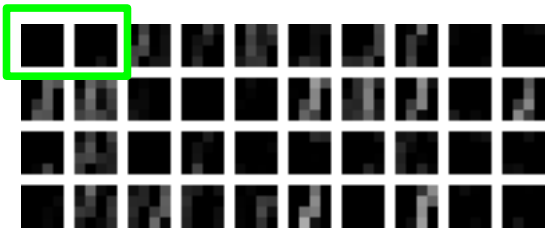
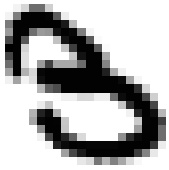
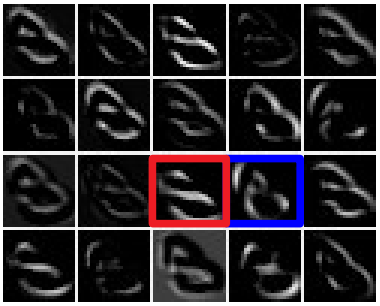
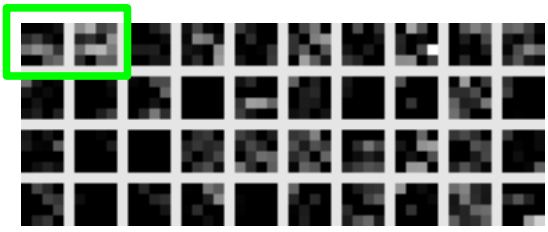
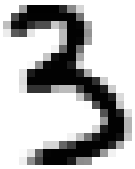
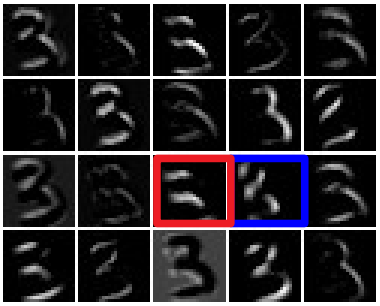
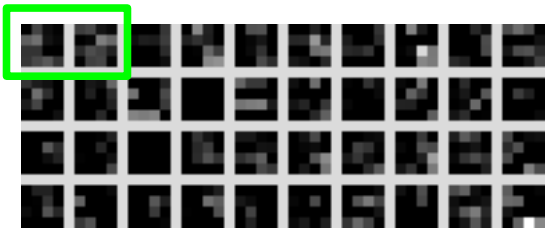
(a) imagem	(b) 20 saídas dos filtro da primeira camada	(c) características escolhidas
 at_1_002_dig.png		
at_1_005_dig.png		
 at_3_018_dig.png		
 at_3_032_dig.png		

Figura 11: (a) Dígito a classificar. (b) Ativações da primeira camada (preto=0, branco=1). (c) As características extraídas automaticamente pela rede convolucional (preto=0, branco=2).
https://colab.research.google.com/drive/1Lq1-ud2_va0PB57mmn-IJC8pS-5VzoNV?usp=sharing

```python
#ativacao1.py
url='http://www.lps.usp.br/hae/apostila/cnn1.h5'
import os; nomeArq=os.path.split(url)[1]
if not os.path.exists(nomeArq):
  print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
  os.system("wget -U 'Firefox/50.0' "+url)
else:
  print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())

import tensorflow.keras as keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras import models
from tensorflow.keras.models import Model
import numpy as np; import cv2
from matplotlib import pyplot as plt;
import matplotlib.patches as patches
import sys

(_,_), (qx, QY) = mnist.load_data()
qx=255-qx

nl, nc = qx.shape[1], qx.shape[2] #28, 28
QX = qx.astype('float32') / 255.0 # 0 a 1

model=models.load_model("cnn1.h5")

lista=[]; digito=[]
ndig=2; #quantidade procurado
j=0 #indice de QY
for dig in (1,3):
  for i in range(ndig):
    while QY[j]!=dig and j<QY.shape[0]:
      j+=1
    if j>=QY.shape[0]:
      sys.exit("Erro inesperado")
    lista.append(j); digito.append(dig)
    j+=1

intermediate_layer_model1 = Model(inputs=model.input,
                                  outputs=model.get_layer(index=0).output)
for j,dig in zip(lista,digito):
  print("Imagem digito=%d, indice=%d"%(dig,j))

  st="di_%1d_%03d_dig.png"%(dig,j); cv2.imwrite(st,qx[j])
  plt.imshow(qx[j],cmap="gray"); plt.axis("off"); plt.show()

  x=QX[j].copy(); x=np.expand_dims(x,axis=0); x=np.expand_dims(x,axis=-1);
  y = intermediate_layer_model1.predict(x)
  y = np.squeeze(y,0)
  y2=np.empty( (y.shape[2], y.shape[0], y.shape[1]) )
  for i in range(y2.shape[0]):
    y2[i,:,:]=y[:,:,i]

  fig, axes = plt.subplots(nrows=4, ncols=5)
  i=0
  for ax in axes.flat:
    im=ax.imshow(y2[i], vmin=0, vmax=1, cmap="gray"); i+=1
    ax.axis('off')
    rect = patches.Rectangle((50,100),40,30,linewidth=10,edgecolor='r',facecolor='none')
    ax.add_patch(rect)
  fig.subplots_adjust(right=0.8)

  cbar_ax=fig.add_axes([0.85, 0.15, 0.05, 0.7])
  fig.colorbar(im,cax=cbar_ax)

  st="a0_%1d_%03d_dig.png"%(dig,j)
  plt.savefig(st)
  plt.show()
```

Programa: Visualiza ativações. https://colab.research.google.com/drive/1Lq1-ud2_va0PB57mmn-IJC8pS-5VzoNV?usp=sharing

Exercício: Gere mapa de 640 atributos para dígitos de 0 a 9 e verifique se realmente é fácil classificá-los.

5 Classificação de MNIST por CNN com “data augmentation”

Para diminuir ainda mais a taxa de erro na classificação MNIST, é possível usar alguns “truques”.

O primeiro deles é aumentar artificialmente os dados de treinamento, chamado “data augmentation”. A ideia é pegar cada imagem de treino (AX) e deslocar um pixel em cada direção (norte, sul, leste, oeste), obtendo 5 imagens de treino (a original mais as quatro deslocadas, linhas 28-38 marcadas em amarelo). Com isso, obtemos dado de treino 5 vezes maior do que o original. Keras possui rotinas sofisticadas de “data augmentation” prontas. Porém, aqui vamos fazer manualmente “data augmentation” para que a ideia fique mais clara.

Nota: Em Python, sempre que possível, deve-se evitar usar “loops”, pois “loops” são muito lentos em Python. Assim, as rotinas para deslocar a imagem NSLO foram escritas sem “loops”. Veja, por exemplo, https://numpy.org/doc/stable/reference/arrays.indexing.html para mais detalhes.

O segundo são as camadas “Dropout” (linhas 57, 60 e 62, marcadas em amarelo). Estas camadas desativam aleatoriamente uma certa porcentagem dos neurônios de uma camada da rede durante o treino. Isto parece diminuir “overfitting”, pois obriga que a CNN funcione mesmo com vários neurônios desativados.

O terceiro é diminuir learning rate toda vez que a rede para de melhorar, através de uma função “callback” (linhas 69, 70 e 71).

O programa obtido com essas alterações está em programa 2. Rodando este programa, obtemos:

```
Test loss: 0.0180
Test accuracy: 99.63 %
Test error: 0.37 %
```

A taxa de erro obtida é 0,37%. O resultado é comparável ao estado da arte:
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

Os 37 dígitos classificados incorretamente estão mostrados na figura 12. Repare que mesmo nós, seres humanos, temos dificuldade para classificar corretamente muitos deles. Esta taxa de erro (0,37%) é bem menor do que a taxa de erro típico de um ser humano (2%).

Exercício: Modifique o programa 2 para obter taxa de erro menor que 0,37%. O treino deve ser repetido algumas vezes e calcular a média. Sugestões: (a) Use regularização. (b) Faça mais data augmentation. (c) Use ensemble e TTA. (d) Use redes convolucionais avançadas.

Exercício: Escreva um programa que imprime os dígitos classificados incorretamente, como na figura 12.

Exercício: Modifique o programa cnn2.py para usar somente convoluções 3×3 ou de tamanho menor (2×2 ou 1×1). O número de camadas convolucionais pode ser maior do que no programa cnn2.py. (Nota: A rede convolucional VGG utiliza justamente esta ideia.) Esta rede deve atingir taxa de erro de teste menor que 0,5% (a minha solução chegou ao erro 0,39%). Para que a execução não demore demasiadamente, mantenha batch_size=1000, epochs=100.

Solução privada em https://colab.research.google.com/drive/1JxlFd8Fafi49FNVRanzL_6iQPCMLTZ50?usp=sharing

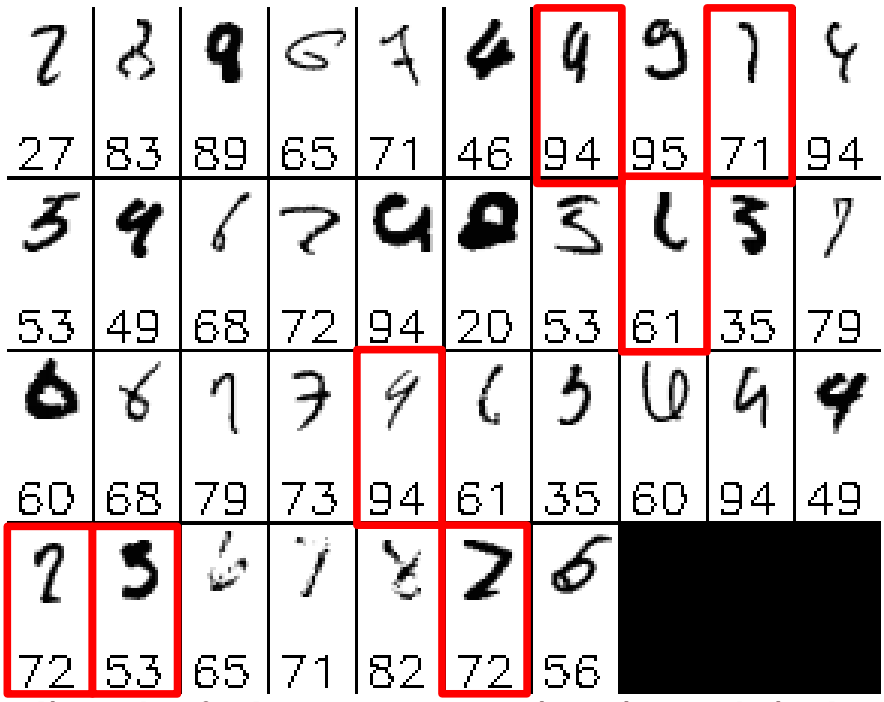


Figura 12: Os 37 dígitos classificados incorretamente. Número à esquerda é a classificação correta. Número à direita é a classificação dada pelo algoritmo. Marquei em vermelho os dígitos onde a classificação dada pela CNN parece mais correta do que a classificação verdadeira.

```python
#cnn2.py - grad2020 - Testado em TF2 em Colab
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import tensorflow.keras as keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dropout, Conv2D, MaxPooling2D, Dense, Flatten
from tensorflow.keras import optimizers
from tensorflow.keras.callbacks import ReduceLROnPlateau
import numpy as np; import sys

def deslocaEsquerda(a):
  d=a.copy(); d[:,0:-1]=a[:,1:]; return d

def deslocaDireita(a):
  d=a.copy(); d[:,1:]=a[:,0:-1]; return d

def deslocaCima(a):
  d=a.copy(); d[0:-1,:]=a[1:,:]; return d

def deslocaBaixo(a):
  d=a.copy(); d[1:,:]=a[0:-1,:]; return d

print("Lendo MNIST")
(AX, AY), (QX, QY) = mnist.load_data()
AX=255-AX; QX=255-QX

print("Fazendo manualmente data augmentation")
AX.resize((5*60000,28,28))
AY.resize((5*60000,1))
for s in range(60000):
  AX[s+60000]=deslocaEsquerda(AX[s])
  AX[s+2*60000]=deslocaDireita(AX[s])
  AX[s+3*60000]=deslocaCima(AX[s])
  AX[s+4*60000]=deslocaBaixo(AX[s])
  AY[s+60000]=AY[s]
  AY[s+2*60000]=AY[s]
  AY[s+3*60000]=AY[s]
  AY[s+4*60000]=AY[s]

print("Convertendo para categorico e float")
nclasses = 10
AY2 = keras.utils.to_categorical(AY, nclasses)
QY2 = keras.utils.to_categorical(QY, nclasses)
nl, nc = AX.shape[1], AX.shape[2] #28, 28
AX = AX.astype('float32') / 255.0 - 0.5 # -0.5 a +0.5
QX = QX.astype('float32') / 255.0 - 0.5 # -0.5 a +0.5
AX = AX.reshape(AX.shape[0], nl, nc, 1)
QX = QX.reshape(QX.shape[0], nl, nc, 1)

print("Construindo modelo")
model = Sequential()
model.add(Conv2D(20, kernel_size=(5,5), activation='relu', input_shape=(nl,nc,1)))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(40, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(200, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(50, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(nclasses, activation='softmax'))

print("Treinando modelo")
opt=optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=opt, loss="categorical_crossentropy", metrics=["accuracy"])

reduce_lr = ReduceLROnPlateau(monitor='accuracy',
  factor=0.9, patience=2, min_lr=0.0001, verbose=True)
model.fit(AX, AY2, batch_size=1000, epochs=100, verbose=2,
          validation_data=(QX, QY2), callbacks=[reduce_lr])

score = model.evaluate(QX, QY2, verbose=False)
print('Test loss: %.4f'%(score[0]))
print('Test accuracy: %.2f %%'%(100*score[1]))
print('Test error: %.2f %%'%(100*(1-score[1])))
model.save("cnn2.h5")
```

Programa 2: Classificação de MNIST usando CNN com “data augmentation” manual.

https://colab.research.google.com/drive/16PRv2xq_ehj4eJpNRC2ph3Iyjz-PPtVL?usp=sharing

~/haepi/deep/keras/conv/cnn3/cnn2.py

6 Treino, validação e teste

Em aprendizado de máquina, os dados de teste não podem ser usados durante o treino *de jeito nenhum* para ajudar a melhorar a rede. Isto constitui um "vazamento de informação". Por exemplo, é errado gravar a rede quando a taxa de erro, medido nos dados de teste, é a menor possível. Também é errado diminuir learning rate quando o desempenho do modelo medido nos dados de teste para de melhorar. Fazendo isto, a rede obtida estará adaptada para classificar bem os dados de teste mas quase certamente funcionará pior quando processar novos dados não vistos.

Isto pode parecer óbvio nos exemplos simples, mas deixa de ser óbvio no meio de um projeto real complexo.

No programa 2, estamos usando os dados de teste (QX, QY) para validação – isto é perigoso e não é recomendado. Porém, neste caso, não usamos os dados de teste para nada exceto para acompanhar a taxa de acerto de teste durante o treino.

Se precisar testar o desempenho do modelo durante o treino, o certo é separar um subconjunto dos dados de treino para essa finalidade, denominado de dados de validação (VX, VY). Durante o treino, o conjunto de treino (AX, AY) deve ser usado para ajustar os parâmetros da rede e o conjunto de validação (diferente do conjunto de teste) pode ser usado para estimar a taxa de erro da rede, para diminuir learning rate quando o modelo para de melhorar ou para gravar o modelo obtido quando se chega num modelo bom (medido nos dados de validação). Somente quando a rede estiver completamente treinada, o conjunto de teste deve ser usado.

Nota: Alguns autores utilizam o termo "conjunto de validação" como sinônimo para "conjunto de teste", em vez de usar no sentido descrito acima.

Referências:

[Nielsen] http://neuralnetworksanddeeplearning.com/

[Lecun1989a] LeCun, Y. ""Generalization and network design strategies" Technical Report CRG-TR-89-4. Department of Computer Science, University of Toronto. [http://yann.lecun.com/exdb/publis/pdf/lecun-89.pdf]

[LeCun1989b] LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W.; Jackel, L. D. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition" Neural Computation. MIT Press - Journals. 1 (4): 541–551. [https://doi.org/10.1162%2Fneco.1989.1.4.541]

[Goodfellow2016] Ian Goodfellow and Yoshua Bengio and Aaron Courville, "Deep Learning", https://www.deeplearningbook.org/

[PSI3471 aula 12. Fim]