Relatório_15_Redes Neurais Convolucionais 2 (Deep Learning) (II)

Davi Bezerra Barros

Redes Neurais Convolucionais

O que é convolução?

Convolução é uma operação matemáticas que combinam duas imagens para gerar uma terceira imagem. A imagem de saída é o resultado da aplicação do filtro na imagem de entrada, fazendo multiplicações e somando os resultados em cada posição do filtro.

A convolução então, é considerada como uma modificação de imagem; a imagem resultante é obtida convertendo a imagem original usando filtros que atuam como detector de características.

Exemplos de Convolução

- Fazer desfocagem em uma imagem com Filtro Gaussiano, mostrando um efeito embaçado.
- Dê destaque aos limites das imagens enfatizando os contornos dos objetos.

Implementando convolução

A convolução ocorre quando os pixels do filtro são multiplicados ponto por ponto pelos pixels da imagem e seus resultados são somados. A fórmula para a convolução é dada por:

$$(A * w)_{ij} = \sum_{i'=0}^{K-1} \sum_{j'=0}^{K-1} A(i+i', j+j') w(i', j')$$

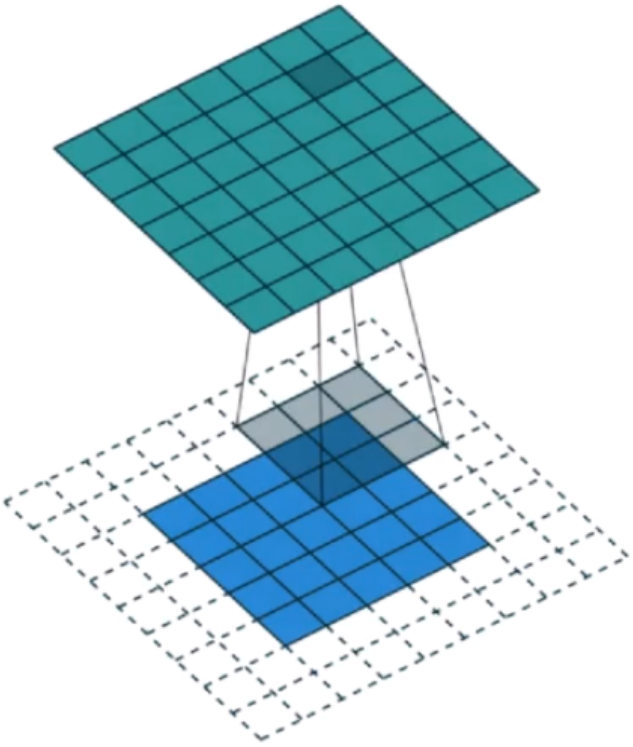
Modos de Convolução

Os modos de convolução definem como os filtros serão aplicados na imagem:

- **mode ='valid':** O filtro é aplicado dentro dos limites da imagem, e gera uma imagem menor.
- **mode = 'same'** : O filtro é aplicado até seu centro se sobrepor à borda da imagem. Os pixels que extrapolam a imagem são prenchidos com zeros **CNN code preparation**

Etapas utilizadas para implementar o código de machine learning, utilizando tensorflow

- **mode = 'full' :** O filtro é aplicado até suas bordas se sobreporem às bordas da imagem. Os pixels que extrapolam a imagem são prenchidos com zeros



Interpretação Geométrica:

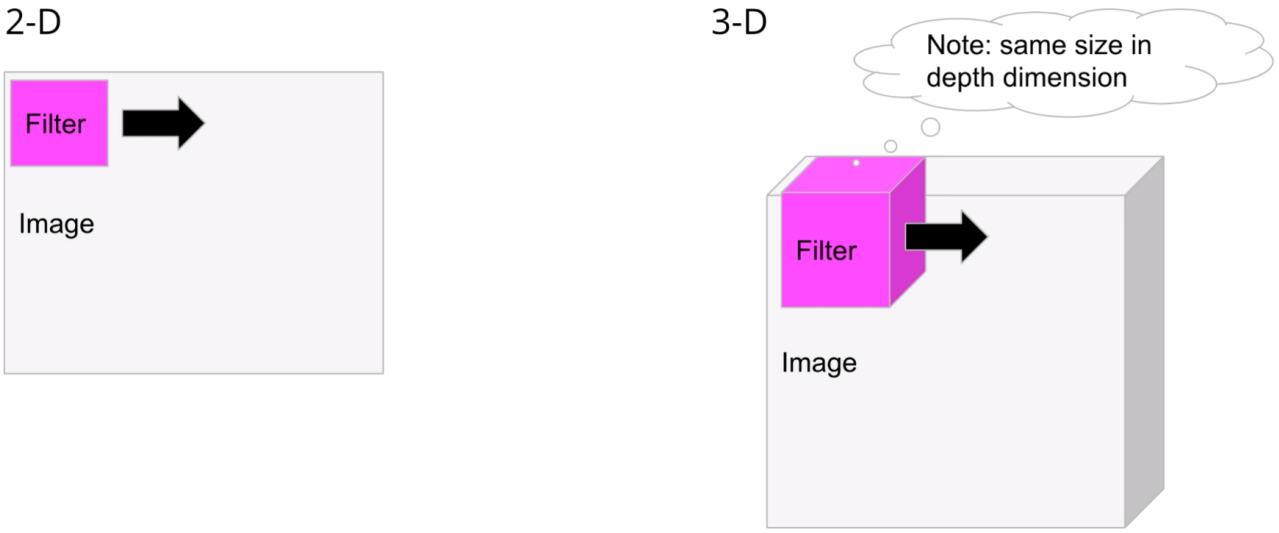
A magnitude do produto escalar entre dois vetores está relacionada ao cosseno do ângulo entre eles. Na convolução, isso significa que seu valor em um ponto da imagem significa semelhança entre um bloco da imagem e um filtro. Ângulos muito próximos de zero resultam em valores altos, enquanto ângulos próximos de 180 graus produzem valores baixos.

Assim, o filtro desliza pela imagem, procurando por regiões que apresentam similaridade com o padrão do filtro. Onde o padrão é encontrado, a convolução gera um valor alto, e onde não há padrão, o valor da convolução é baixo.

Convolução em imagens coloridas

Em geral, uma rede neural aceita imagens coloridas como entrada. Diferentemente das imagens monocromáticas, que são representadas como matrizes bidimensionais, as imagens coloridas são representadas como matrizes tridimensionais; Uma dimensão para cada uma das cores RGB (vermelho verde e azul).

O processo de convolução utiliza um filtro que também é tridimensional, e a aplicação do filtro sobre a imagem é realizada da mesma forma, deslizando o filtro sobre a imagem e fazendo uma soma ponderada dos pixels:



Dessa forma, o filtro que era monocromático agora busca padrões de cor, e atributos buscados pelo filtro em um dos canais de cor não terão correspondência nos outros canais, mesmo que tenham exatamente a mesma geometria.O cálculo da convolução com três canais resulta em uma matriz bidimensional:

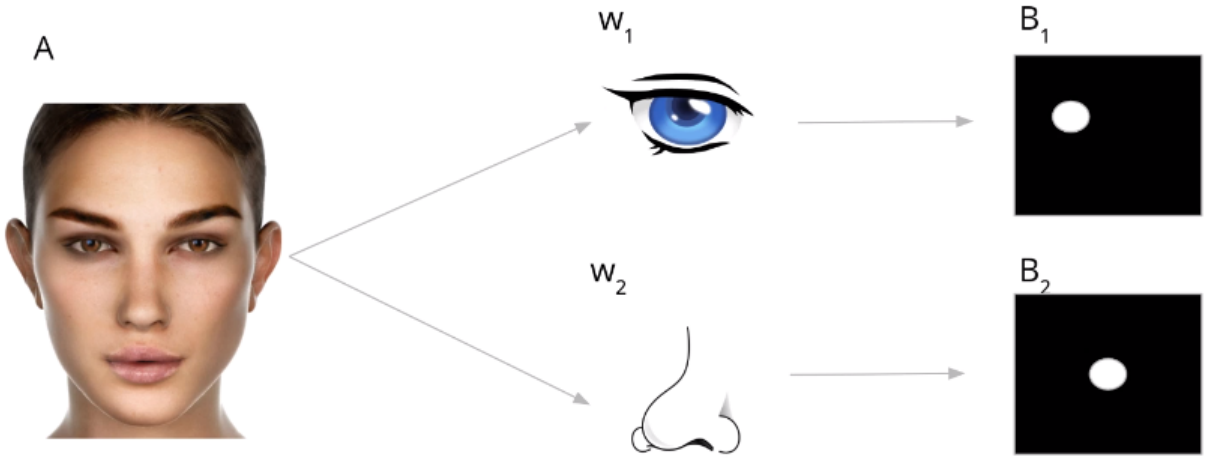
$$(A * w)_{ij} = \sum_{c=1}^{3} \sum_{i'=0}^{K-1} \sum_{j'=0}^{K-1} A(i+i', j+j', c) w(i', j', c)$$

2 indices

3 indices

Múltiplos atributos

No caso de uma rede neural que busca por mais de uma característica na imagem, como um detector facial, múltiplos filtros são utilizados, um para cada característica buscada.



No exemplo acima, as saídas produzidas pela convolução da imagem **A** com s filtros **W1** e **W2** são ambas unidimensionais. Porém, ao combinar as saídas **B1** e **B2**, o resultado se torna tridimensional, assim como a imagem de entrada. Assim, múltiplos filtros podem ser utilizados para detectar múltiplas características, e a dimensionalidade do resultado é proporcional à quantidade de filtros utilizados.

VetorizaçãoME-W34

Na transição entre as redes convolucional e densa, não é necessário vetorizar cada canal de cor separadamente para usar como input. A convolução pode ser feita com uma única operação, desta vez resultando em uma saída tridimensional, com o canal de cor sendo a terceira dimensão. Esta operação é representada pela seguinte expressão:

$$B(i,j,c) = \sum_{c'=1}^{3} \sum_{i'=0}^{K-1} \sum_{j'=0}^{K-1} A(i+i', j+j', c') w(c', i', j', c)$$

Feature maps

Chamar a terceira dimensão de cor apenas faz sentido para a imagem de entrada, com formato HxWx**3**. Após as convoluções, os resultados são do formato HxWx **#** (# = valor arbitrário),e a terceira dimensão não necessariamente representa as cores.

Feature map é a terminologia utilizada para definir a saída; um conjunto imagens 2D que mostram onde o atributo foi encontrado na imagem de entrada. Assim, a última dimensão representa a quantidade de feature maps, e não as cores.

Camada de convolução

Termo bias e função de ativação

A adição de um termo de bias e uma função de ativação, assim como na rede neural densa, permite encontrar características não lineares ou funções não lineares de entrada. Em uma camada de convolução, diferentemente de uma camada densa, o vetor bias não tem o mesmo tamanho da matriz convolucionada:

$$Conv\ layer: \ \sigma(W * x + b)$$

$$Dense\ layer: \ \sigma(W^T x + b)$$

Em teoria, esta operação não seria permitida pelas regras de aritméticas de matrizes, mas é permitida pela biblioteca NumPy utilizando a técnica de *broadcasting*. O termo bias é unidimensional, e aplicado a cada pixel para cada *feature map*.

Parameter sharing

A convolução economiza parâmetros em comparação com uma rede densa, pois reutiliza os mesmos parâmetros para as operações.

Por exemplo, supondo uma convolução no nodo *valid*:

- Imagem de entrada: 32 x 32 x 3
- Filtro: 3 x 5 x 5 x 64
- Imagem de saída: 28 x 28 x 64
- Total de parâmetros: 3 x 5 x 5 x 64 = 4800

Na rede densa:

- Imagem de entrada: 32 x 32 x 3 = 3072
- Vetor de saída: 28 x 28 x 64 = 50176
- Imagem de saída: 28 x 28 x 64
- Matriz de pesos: 3072 x 50176 = 154,140,672

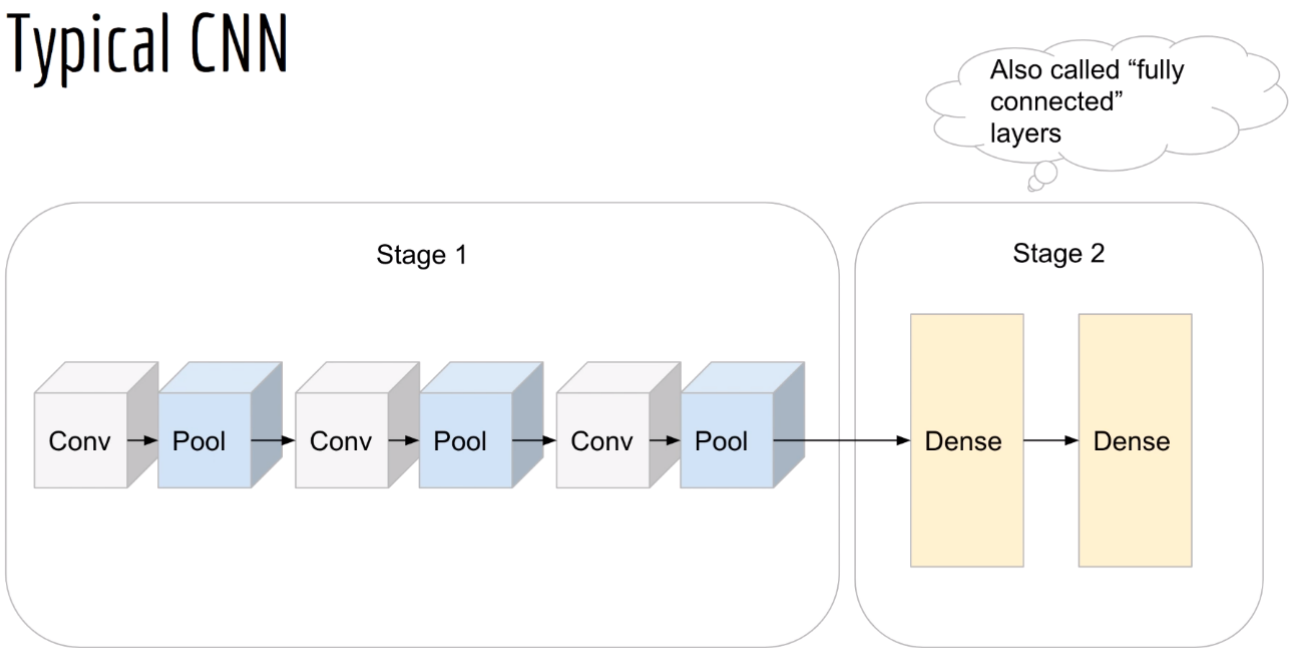
São 32 mil vezes mais parâmetros que a rede convolucional, o que teria um custo computacional excessivamente alto.

Aprendendo Filtros Convolucionais:

Os filtros são aprendidos durante o processo de treinamento da rede neural. Através do ajuste do modelo (backpropagation), a rede ajusta os pesos e vieses da camada de convolução para otimizar o desempenho, Esse processo permite que a rede descubra automaticamente os padrões mais importantes na entrada para a tarefa em questão.

Arquitetura da CNN

A arquitetura de uma CNN comum é estruturada em dois estágios:



- **Estágio 1:** Camadas convolucionais seguidas por camadas de pooling.
- **Estágio2:** Uma série de camadas densas totalmente conectadas.

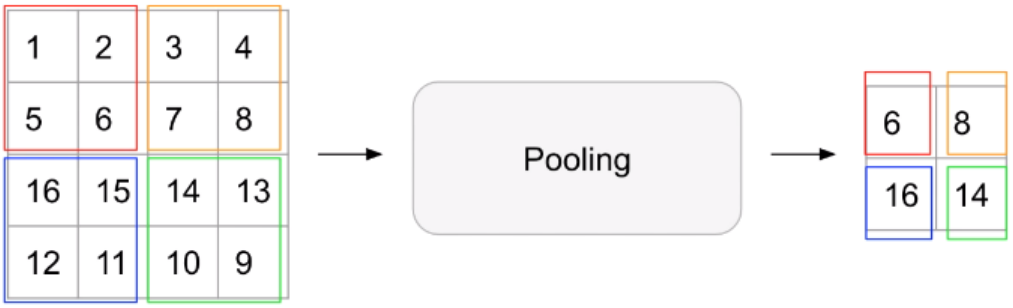
Cada camada convolucional apenas processa os atributos encontrados nas camadas anteriores, e as camadas densas fazem a classificação ou regressão linear.

Pooling:

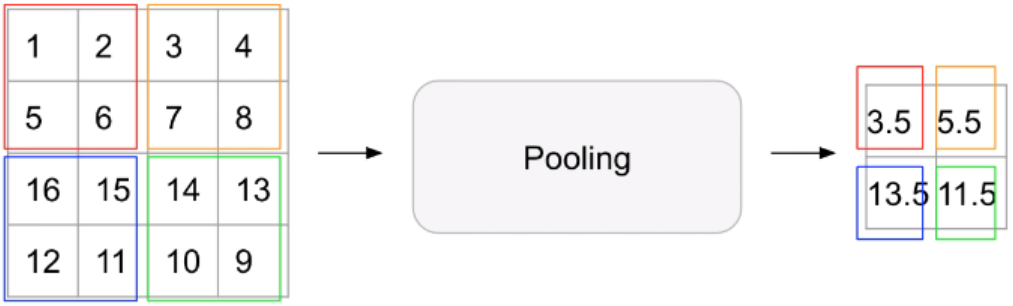
As camadas de pooling reduzem a imagem original, para que se tenha menos dados para processar. Geralmente se usa um pooling de tamanho 2, que reduz a imagem a 50% de seu tamanho original. Uma imagem com 100x100 pixels ficaria com 50x50 pixels após ser reduzida pelo pooling.

Há dois tipos de pooling:

- **Max pooling:** Retorna o maior valor a cada intervalo de 2x2 pixels da imagem.



- **Average pooling:** Retorna a média dos valores a cada intervalo de 2x2 pixels da imagem:



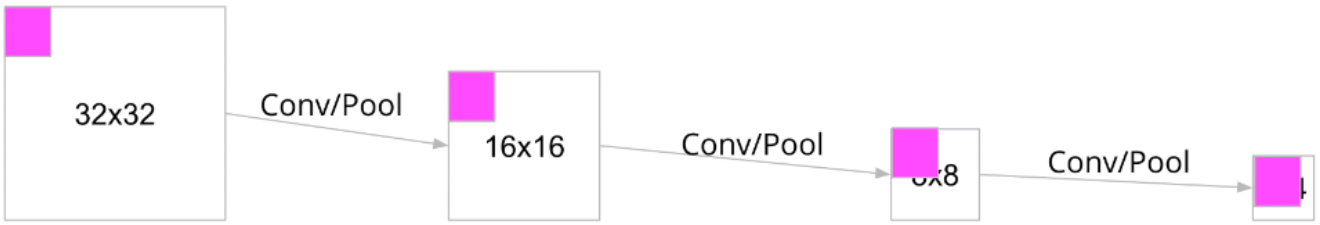
Além de reduzir a quantidade de dados ao diminuir o tamanho da imagem, o pooling permite a *invariância translacional*; A informação importante é se o atributo buscado foi encontrado na imagem, não o local onde ele foi encontrado. Isso permite com que o modelo reconheça os atributos sem precisar aprender cada variação possível que eles possam assumir.

Por que alternar convolução e pooling?

As CNN aprendem atributos de forma hierárquica, aumentando a precisão e especificidade das características buscadas a cada camada, dos casos mais gerais aos mais específicos:

```
![[Pasted image 20240602211648.png]]
```

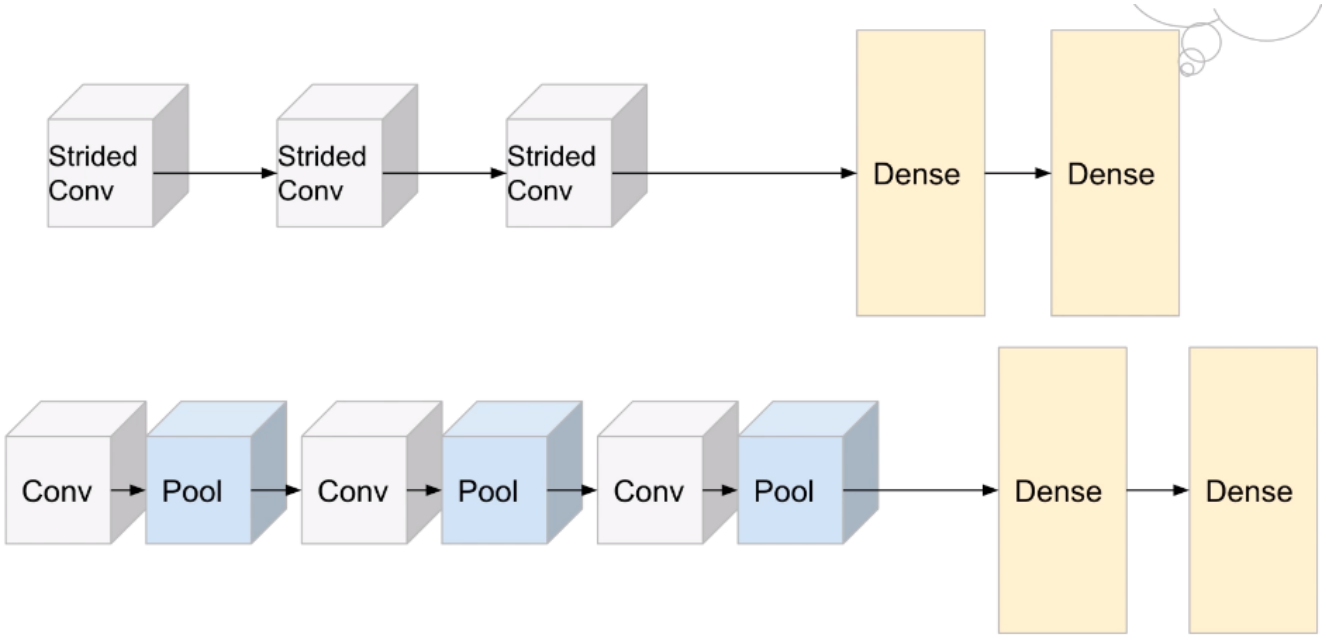
A cada pooling, o tamanho da imagem diminui mas o tamanho dos filtros se mantém o mesmo, e a cada camada, o filtro busca por padrões que ocupam cada vez mais espaço na imagem; Rostos, feições etc.



A cada camada convolucional, se perde a informação de *onde* o atributo buscado foi encontrado na imagem anterior. Em contrapartida, ganha-se informações a respeito de quais atributos foram encontrados na imagem, com o aumento da quantidade de *feature maps*

Stride

Em alternativa ao pooling, Stride é um parâmetro que dita o movimento do filtro sobre a imagem. Ao realizar uma operação de convolução, o *stride* determina quantos pixels o filtro se move a cada passo. Por exemplo, um stride de 1 move o filtro um pixel de cada vez, enquanto uma passada de 2 move dois pixels. Uma passada maior produzirá uma dimensão de saída menor, reduzindo o tamanho da imagem da mesma forma que o pooling:



Isso funciona pois leva-se em consideração que os pixels vizinhos a uma janela de pixels provavelmente tem a mesma natureza, já que são componentes de um único 'algo' maior; uma parte homogênea da imagem.

Global Max Pooling

Uma camada de *Global Max Pooling* é utilizada para lidar com imagens de diferentes tamanhos, visto que no mundo real as imagens raramente têm as mesmas dimensões. O pro

Passo1: carregar os dados

Fashion MNIST : Conjunto de imagens criado par substituir o MNIST em algoritmos de aprendizado de máquina, pois se tornou muito fácil de ser resolvido.

- Imagens em escala de cinza, com 28x28 pixels;
- Rótulos com nomes de roupas: t-shirt, pants, shoes etc.

CIFAR-10: Imagens genéricas coloridas com dimensões de 32x32x3

- Rótulos: automóveis, sapo, cavalo, casam etc

Passo 2: construir o modelo
Criar uma rede neural convolucional utilizando camadas convolucionais utilizando a estrutura apresentada anteriormente e uma nova maneira de utilizar a a API do Keras: Functional API. Esta forma é mais conveniente para construir modelos mais complexos.

Passo 3: Treinar o modelo
Passo 4: Avaliar o modelo
Passo 5: Fazer predições

Essas etapas são agnósticas ao modelo utilizado.

Implementando as etapas:

1. Carregando o Fashion MNST

Ao executar os comandos:

```python
 # para Fashion Mnist
tf.keras.datasets.fashion_mnist.loaD_data()

#Para CIFAR-10:
tf.keras.datasets.cifar10.load_data()

(x_train, y_train), (x_test, y_test) = load_data()
```

Ambos retornam às tuplas, onde a primeira tupla é o dataset de treinamento, e a segunda tupla é o dataset de testes.

As imagens tem exatamente o mesmo formato das imagens em MNIST: N x 28 x 28 com valores de pixel variando de 0 a 255. Como este não é o formato ideal para uma CNN, que espera uma dimensão de cor, esta é então adicionada manualmente. As imagens são então são alteradas para o formato N x 28 x 28 x 1. A dimensão extra não adiciona nenhuma informação à imagem original.

No dataset CIFAR-10, as imagens tem o formato N x 32 x 32 x 3 e a função flatten() é utilizada para redimensionar os rótulos, que são do formato N x 1.

2. Construindo o modelo

A construção do modelo é feita utilizando o Keras Functional API, que permite a construção de modelos flexiveis ao utilizar funções para representar as camadas de convolução.

Ao construir um modelo convencional, cria-se uma lista com as camadas de entrada e densa, e essa lista é passada para o construtor sequencial. O construtor retorna um modelo no qual podemos chamar os ajuste, predição e outras operações.

```python
model = Sequential([Input(shape=(D,)), Dense(128, activation ='relu'), Dense(K, activation ='softmax'),])

...
model.fit(...)
model.predict(...)
```

Na forma funcional, cria-se as camadas e para cada camada são passados os inputs, para receber o output como se fossem funções:

```python
i = Input(shape = (D,))
x = Dense(128, activation = 'relu')(i)
x = Dense(K, activation - 'softmax')(x)

model = Model(i,x)

...
model.fit(...)
model.predict(...)
```

As camadas são objetos mas se comportam como funções. As saídas e entradas são então passadas para o construtor do modelo, que retorna o modelo como objeto.

2.1. Criando uma CNN usando API Funcional

```python
i = Input(shape = x_train[0].shape)
x = Conv2D(32, (3, 3), strides=2, activation = 'relu')(i)
x = Conv2D(64, (3, 3), strides=2, activation = 'relu')(x)
x = Conv2D(128, (3, 3), strides=2, activation = 'relu')(x)
x = Flatten()(x)
x = Dense(512, activation = 'relu')(x)
x = Dense(K, activation='softmax')(x)

model = Model(i, x)
```

A variável X é atualizada com a saída da layer
Argumentos do Conv2D:

- **'32'** = Número de feature maps na saída
- **'(3, 3)'**= Dimensões do filtro
- **'strides'** = Passos do filtro sobre a imagem
- '**relu**' = Função de ativação dos neurônios

2.2. Implementando CNN com Fashion MNIST

Importando o dataset:

```python
fashion_mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train), (x_test,y_test) = fashion_mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
print("x_train.shape:", x_train.shape)
```

x_train.shape: (60000, 28, 28)

Adicionando uma dimensão extra para a cor e criando classes:

```python
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print(x_train.shape)

#Numero de classses
K = len(set(y_train))
print("numero de classes: ",K)
```

60000, 28, 28, 1)
numero de classes: 10

Construção das camadas utilizando API funcional:

```python
 #construindo o codigo usando API funcinoal
i = Input(shape=x_train[0].shape)
x = Conv2D(32, (3, 3), strides=2, activation = 'relu')(i)
x = Conv2D(64, (3, 3), strides=2, activation = 'relu')(x)
x = Conv2D(128, (3, 3), strides=2, activation = 'relu')(x)
x = Flatten()(x)
x = Dropout(0.2)(x)
x = Dense(512, activation = 'relu')(x)
x = Dropout(0.2)(x)
x = Dense(K, activation = 'softmax')(x)

model = Model(i,x)
```

Treinando o modelo:

```python
#Compilar e fit

model.compile(optimizer ='adam',
loss = 'sparse_categorical_crossentropy',
metrics = ['accuracy'])
r = model.fit(x_train, y_train, validation_data = (x_test, y_test), epochs=15)
```

```
Epoch 1/15
1875/1875 [==============================] - 38s 19ms/step - loss: 0.5227 - accuracy: 0.8067 - val_loss: 0.4107 - val_accuracy: 0.8450
Epoch 2/15
1875/1875 [==============================] - 33s 17ms/step - loss: 0.3657 - accuracy: 0.8630 - val_loss: 0.3479 - val_accuracy: 0.8711
Epoch 3/15
1875/1875 [==============================] - 32s 17ms/step - loss: 0.3157 - accuracy: 0.8804 - val_loss: 0.3331 - val_accuracy: 0.8741
Epoch 4/15
1875/1875 [==============================] - 39s 21ms/step - loss: 0.2817 - accuracy: 0.8935 - val_loss: 0.3153 - val_accuracy: 0.8824
Epoch 5/15
1875/1875 [==============================] - 56s 30ms/step - loss: 0.2557 - accuracy: 0.9032 - val_loss: 0.3023 - val_accuracy: 0.8935
Epoch 6/15
1875/1875 [==============================] - 45s 24ms/step - loss: 0.2353 - accuracy: 0.9118 - val_loss: 0.2943 - val_accuracy: 0.8935
Epoch 7/15
1875/1875 [==============================] - 33s 18ms/step - loss: 0.2134 - accuracy: 0.9179 - val_loss: 0.2876 - val_accuracy: 0.8988
Epoch 8/15
1875/1875 [==============================] - 32s 17ms/step - loss: 0.1979 - accuracy: 0.9250 - val_loss: 0.3036 - val_accuracy: 0.8956
Epoch 9/15
1875/1875 [==============================] - 31s 17ms/step - loss: 0.1831 - accuracy: 0.9313 - val_loss: 0.3160 - val_accuracy: 0.8996
Epoch 10/15
1875/1875 [==============================] - 33s 17ms/step - loss: 0.1697 - accuracy: 0.9347 - val_loss: 0.3142 - val_accuracy: 0.9010
Epoch 11/15
1875/1875 [==============================] - 34s 18ms/step - loss: 0.1573 - accuracy: 0.9408 - val_loss: 0.3151 - val_accuracy: 0.8993
Epoch 12/15
1875/1875 [==============================] - 32s 17ms/step - loss: 0.1500 - accuracy: 0.9423 - val_loss: 0.3088 - val_accuracy: 0.9037
Epoch 13/15
1875/1875 [==============================] - 33s 17ms/step - loss: 0.1413 - accuracy: 0.9456 - val_loss: 0.3428 - val_accuracy: 0.9038
Epoch 14/15
1875/1875 [==============================] - 32s 17ms/step - loss: 0.1354 - accuracy: 0.9487 - val_loss: 0.3369 - val_accuracy: 0.9011
Epoch 15/15
1875/1875 [==============================] - 31s 17ms/step - loss: 0.1272 - accuracy: 0.9506 - val_loss: 0.3503 - val_accuracy: 0.9034
```

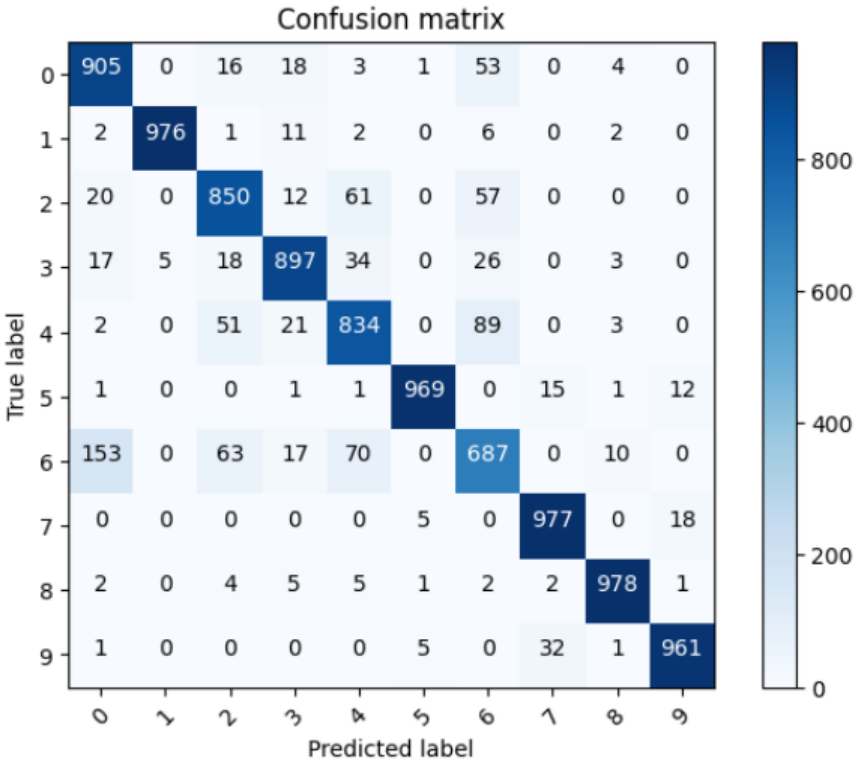
Aqui a assertividade do modelo está baixa por que o Fashion MNIST é mais 'Difícil' que o mnist original, pois as imagens são mais complexas e algumas de suas classes são muito semelhantes, como as classes **camisa e vestido**, por exemplo. Otimizações serão necessárias para melhorar a taxa de acerto do modelo.

Gerando matriz de confusão para avaliar o modelo:

Python

```python
from sklearn.metrics import confusion_matrix
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

  if normalize:
    cm = cm.astype('float') / cm.sum(axis = 1)[:,np.newaxis]
    print("matriz de confusão normalizada")
  else:
    print('Matriz de confusão, sem normalização')

  print(cm)

  plt.imshow(cm, interpolation = 'nearest', cmap = cmap)
  plt.title(title)
  plt.colorbar()
  tick_marks = np.arange(len(classes))
  plt.xticks(tick_marks, classes, rotation = 45)
  plt.yticks(tick_marks, classes)

  fmt = '.2f' if normalize else 'd'
  thresh = cm.max() /2.
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i,j], fmt),
             horizontalalignment = "center",
```

```
            color = "white" if cm[i,j] > thresh else "black")
  plt.tight_layout()
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  plt.show()

p_test = model.predict(x_test).argmax(axis = 1)
cm = confusion_matrix(y_test, p_test)
plot_confusion_matrix(cm, list(range(10)))
```

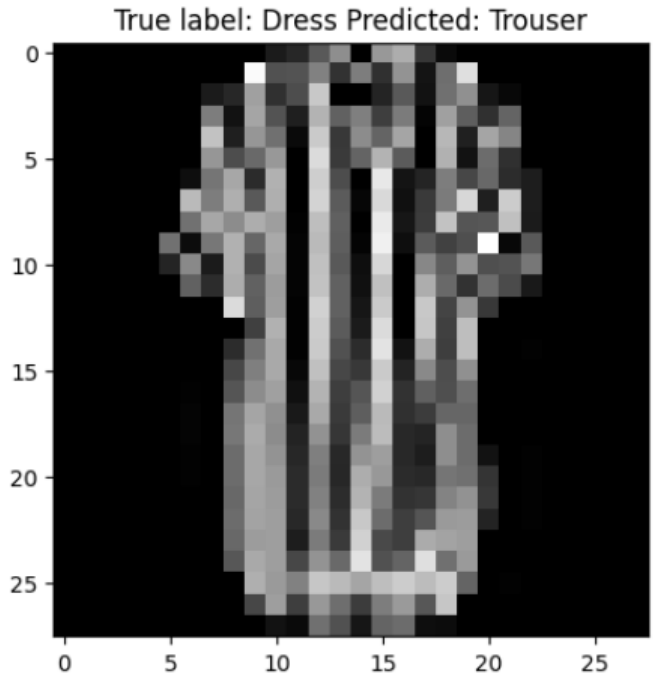


Mapeando os rótulos e mostrando exemplos mal classificados:

Python

```
#Mapeamento de rótulos
labels = '''T-shirt/top
Trouser
Pullover
Dress
Coat
Sandal
Shirt
Sneaker
Bag
Ankle boot'''.split()

#Mostrando exemplos mal classificados
misclassified_idx = np.where(p_test != y_test)[0]
i = np.random.choice(misclassified_idx)
plt.imshow(x_test[i].reshape(28,28), cmap='gray')
plt.title("True label: %s Predicted: %s" % (labels[y_test[i]], labels[p_test[i]]))
```

```
Text(0.5, 1.0, 'True label: Dress Predicted: Trouser')
```

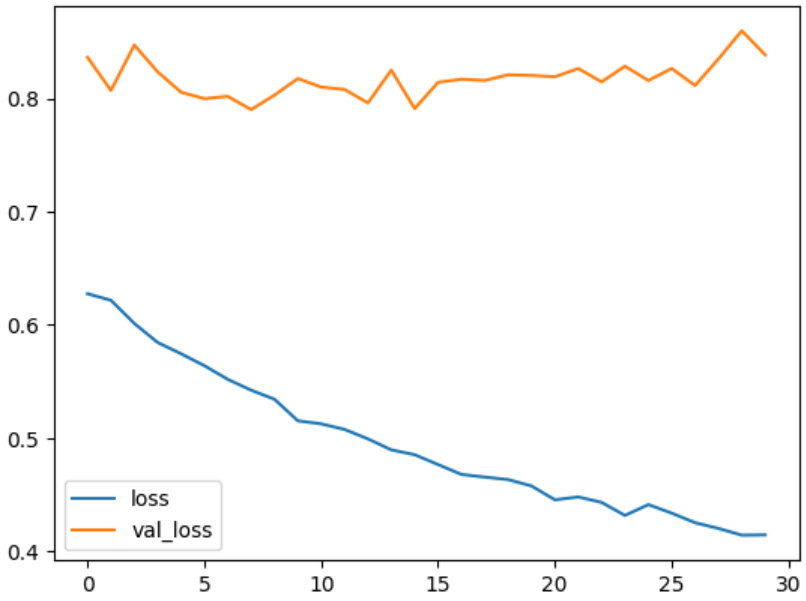


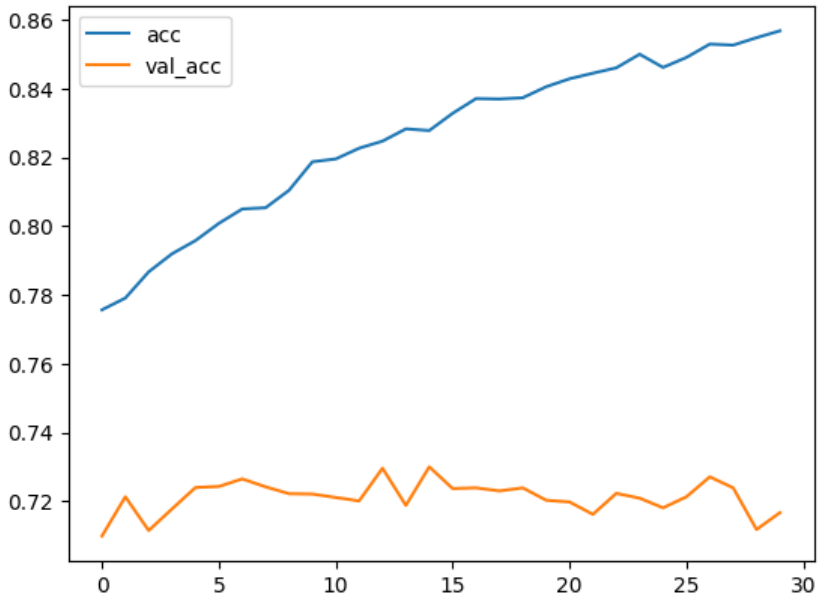
2.3. Implementando CNN com CIFAR-10

A arquitetura das redes neurais é basicamente a mesma. A diferença é que as imagens de CIFAR-10 tem 3 canais de cor, e o primeiro filtro tem o formato 3 x 3 x 3 x 32 .

3-4. Treinando e avaliando o modelo

O modelo apresentado na aula estava sobre ajustado aos dados de treinamento, com 15 épocas de treinamento. Para investigar melhor seus resultados, refiz o treinamento com 30 épocas para ver a evolução das métricas de erro e precisão. Os resultados obtidos foram os seguintes:



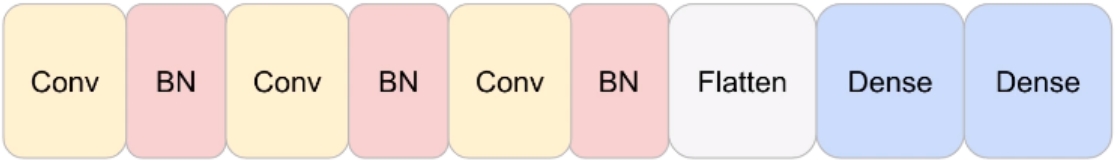


De acordo com o gráfico de erro(loss), o modelo está sobre ajustado aos dados de treinamento. O aumento da precisão nos dados de treinamento junto com a baixa precisão nos dados de teste reforçam que o modelo está sobre ajustado. Alterações podem ser feitas para que o modelo generalize melhor, como regularization, ou ***data augmentation*** para aumentar o volume de dados para treinamento.

Data augmentation

Melhorando o modelo utilizando data augmentation. Este processo cria mais imagens a partir das imagens existentes no dataset, modificando as imagens com rotações, espelhamento, shifting, etc, mas mantendo o seu significado. Este processo é feito utilizando a API do Keras no Tensorflow.

Batch Normalization

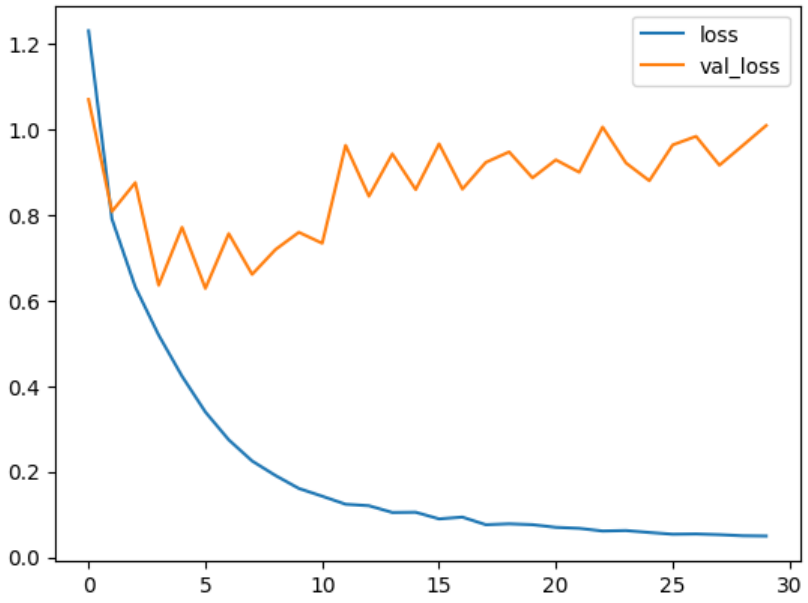


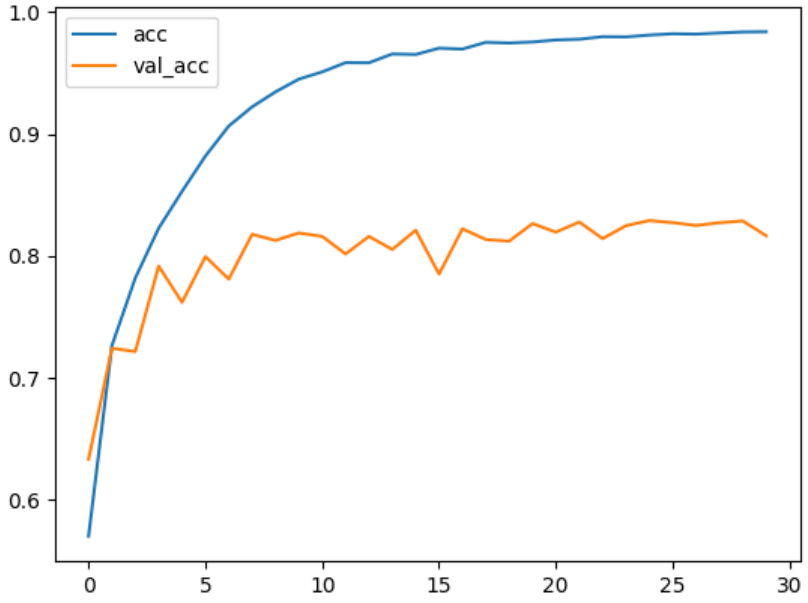
Durante o treinamento, para cada mini-batch, uma camada de Batch Normalization normaliza os inputs de uma camada para que tenham média zero e variância um. Isso é feito subtraindo a média e dividindo pelo desvio padrão dos inputs.

$$z = (x - \mu)/\sigma$$

Melhorando o Cifar-10

Para melhorar o modelo, a arquitetura da rede foi feita seguindo o modelo da rede VGG, que utiliza múltiplos grupos de convolução. A nova rede para o cifar-10 é composta por três grupos de convolução intercalados com camadas de pooling, camadas de normalização entre as convoluções e data augmentation para diversificar os dados de treinamento. Os resultados mostram que o modelo atinge valores mais altos de precisão tanto nos dados de treinamento quanto nos dados de teste, o que indica uma melhor capacidade de generalização, mas ainda tende a errar mais nos dados de teste:





Processamento de linguagem natural

Embeddings

O problema de classificar palavras:

Palavras são objetos categóricos, e não podem ser multiplicados por números. Isso é um problema pois as palavras não podem ser multiplicadas pelas matrizes de peso para treinar o modelo de linguagem.

One Hot Encoding:

Uma solução seria criar usar ***One-hot encoding*** para criar vetores de características para identificar as palavras dentro de um conjunto de dados de um vocabulário / linguagem. O problema desta solução é que os vetores precisariam ser do tamanho da quantidade de palavras na linguagem, e as distâncias euclidianas entre os vetores seriam as mesmas, logo, todas as palavras são equidistantes e esta solução não tem uma estrutura geométrica útil para uma rede neural.

Embedding Layers no TensorFlow

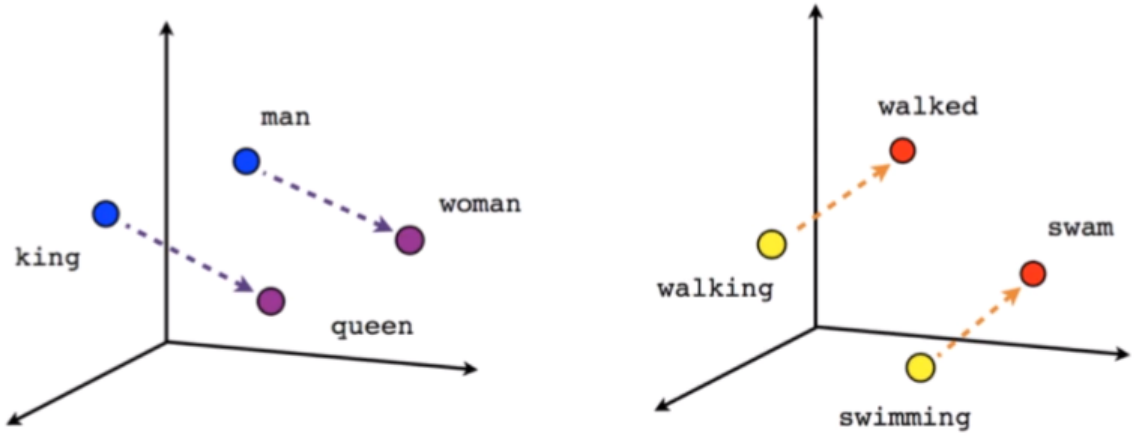
Para resolver o problema geométrico, é feito o mapeamento da sequência de palavras em um vetor com sequências de índices únicos e arbitrários destas palavras. Por exemplo:

```
[ "i", "like", ""cats"] -> [50,25,3]
```

A embedding layer então mapeia cada índice a vetores correspondentes àquelas palavras:

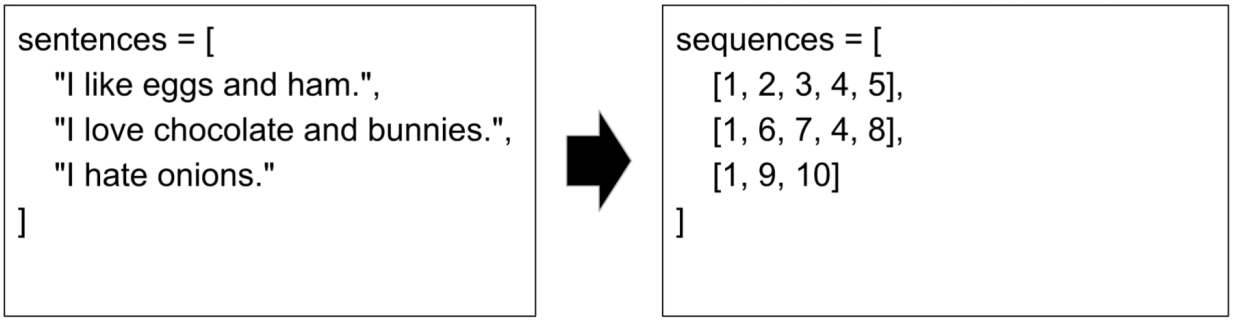
`[50,25,3] -> [[0.3, -0.5], [1.2,-0.7], [-2.1, 0.9]]`

Assim obtemos uma matriz T x D, sendo T o tamanho da expressão e D as dimensões do vetor de palavras Isso permite que as palavras e expressões tenham significado geométrico e possam ser associadas a pesos na rede neural:



Convertendo palavras para inteiros:

1. Primeiramente é necessário realizar um processo chamado de 'tokenização': O texto não é originalmente formatado como uma sequência de palavras, mas como uma string. Primeiro devemos dividir a string de texto nas palavras que a compõe, criando uma lista de strings, onde cada palavra é um token. O Tensorflow e o Keras possuem uma função que faz isso automaticamente, chamada de **Tokenizer**.
2. O Tokenizer então converte a lista de strings em uma lista de inteiros onde Cada inteiro corresponde a uma palavra:



3. Todas as strings precisam ter o mesmo tamanho, pois as RNN's não aceitam sequências de tamanhos diferentes. Para resolver isso, o tensorflow usa *padding* para preencher a menor sequência com zeros e a deixar do tamanho da maior sequência. A função utilizada para isso é a **pad_sequences**. A parametrização do padding é importante pois a adição de zeros no início ou no final da frase influencia a eficiência da rede neural em diferentes cenários.
4. A saída é uma matriz 2D **N x T** onde N é o numero de amostras e T é o tamanho da maior sequência. **X[n,t]** representa o índice da palavra na sentença n e posição de tempo t.

O que fazemos com essa matriz N x T?

A matriz é passada por uma embedding layer para se obter um tensor N x T x D, basicamente convertendo cada índice de palavra em um vetor palavra 3D compatível com uma rede RNN.Essa saída então é treinada normalmente com gradient descent.

Info

Os conteúdos apresentados nas seções 7 e 8 já foram discutidos anteriormente neste documento.

Diretrizes gerais para desenvolver uma CNN

Todas as CNNs seguem uma estrutura composta por duas principais estruturas:

- Camadas de convolução: camadas que processam uma imagem de entrada e a convoluciona com um filtro(kernel), responsável por extrair as características desejadas.
- Camadas densas: Utilizam as características extraídas nas camadas de convolução para realizar classificação.

Com exceção de redes generativas, as arquiteturas de CNNs geralmente utilizam camadas de convolução seguidas por camadas densas, e seguir este padrão torna mais fácil de desenvolve-las. Ao serem passadas pelas camadas convolucionais seguidas por camadas de Pooling, as imagens geralmente diminuem em tamanho e aumentam em profundidade(canais de características) conforme progride o nível de abstração das características extraídas. O hiperparâmetros destas camadas também geralmente seguem um senso comum, como por exemplo, o aumento da dimensionalidade dos filtros ao longo das convoluções.

As arquiteturas de CNN's mais famosas são:

VGG: Formada por 16 camadas divididas em blocos de convoluções seguidos por pooling:

- 2 convoluções > max pooling
- 2 convoluções > max pooling
- 3 convoluções > max pooling
- 3 convoluções > max pooling
- 3 convoluções > max pooling
- 3 camadas totalmente conectadas

AlexNet: Formada por 5 camadas de convolução seguidas por três camadas totalmente conectadas

GoogLeNet: Utiliza um modulo "inception" que convoluciona a imagem com kernels de diferentes tamanhos e concatena seus resultados.

Loss Functions

Mean Squared Error: É uma medida utilizada para calcular a diferença entre os valores previstos por um modelo e os valores reais, calculado como a média da diferença dos quadrados destes valores. Os valores reais e previstos são elevados ao quadrado para garantir que os erros sempre sejam positivos, evitando cancelamento entre eles. O MSE é muito utilizado em regressão linear e outras aplicações de machine learning

Binary Cross-Entropy: É a função de perda usada para problemas de classificação binária, como detecção de spam e fraude, e supõe que os resultados seguem uma distribuição de Bernoulli. Esta função mede a incerteza associada às previsões do modelo, minimizando a diferença entre as previsões e os valores reais.

Categorical Cross-Entropy: É a função de perda utilizada em problemas com muitas classes, onde cada amostra pode pertencer a múltiplas classes. Ela mede a diferença entre a distribuição real e a distribuição prevista pelo modelo, penalizando fortemente as previsões incorretas. É muito utilizada em reconhecimento de imagens e processamento de linguagem natural.

Gradient Descent

É um algoritmo utilizado para treinar os modelos de redes neurais, minimizando a função de perda ao ajustar iterativamente os parâmetros do modelo para reduzir o erro. Ele calcula o gradiente da função de perda em relação aos parâmetros os ajusta na direção oposta ao gradiente. É um algoritmo muito eficaz pois permite encontrar soluções aproximadas através de métodos numéricos.

Stochastic gradient descent: É uma variação da descida gradiente utilizada em grandes conjuntos de dados, tornando o processo mais eficiente e com um menor

custo computacional. O SGD calcula o gradiente usando apenas um subconjunto aleatório das amostras em cada iteração. Embora isso torne o processo um pouco menos preciso, o modelo ainda converge de forma muito eficiente.

Momentum: Um mecanismo utilizado para aumentar a eficiência do processo de otimização ao definir uma "velocidade" acumulada das atualizações dos parâmetros anteriores, ajudando a acelerar o treinamento em regiões com gradientes desiguais, e mínimos locais.

Taxas de aprendizado variáveis

Enquanto o momentum acelera a descida do gradiente, as taxas de aprendizado variáveis ajustam a taxa de aprendizado ao longo do tempo para evitar grandes saltos e melhorar a precisão perto do mínimo, otimizando o treinamento. Os tipos de aprendizado variável são:

- **Step decay:** A taxa de aprendizado é reduzida em um intervalo regular.
- **Decaimento exponencial:**diminui a taxa de aprendizado exponencialmente ao longo do tempo.
- **Decaimento Linear:** Diminui a taxa de aprendizado linearmente.
- **Decaimento Adaptativo:** Ajusta a taxa de aprendizado com base em gradientes passados.

Adam: É um otimizador de aprendizado que combina momentum e taxas de aprendizado adaptativas(RMSprop) para melhorar a eficiência da atualização dos pesos da rede neural. Isso é feito medindo a média exponencial do gradiente e uma média exponencial dos quadrados dos gradiente para ajustar a direção do movimento, ajudando o modelo a convergir mais rapidamente.