Deep learning
Tópicos complementares

Otimização dos modelos de DL; topologias alternativas

Exemplos/ figuras adaptados de F. Chollet, "Deep Learning with Python" e Andrew Ng (deeplearning.ai)

Avaliação de modelos de DL

Treino

Usado para treino dos modelos

Validação
(*dev set*)

Usado para otimizar modelos/ hiper-parâmetros

Teste

Usado para estimar erro do modelo "final" e comparação de modelos

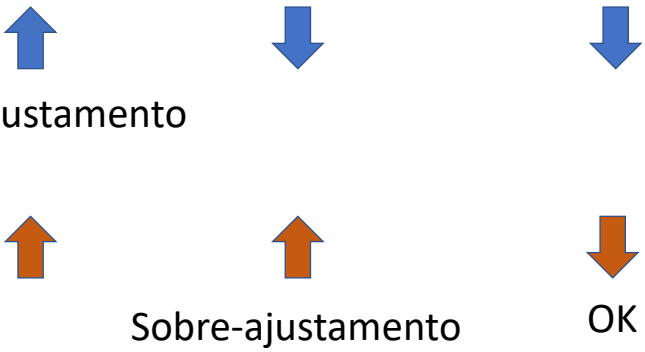
Sobre e sub-ajustamento

Erro no conjunto de **treino**

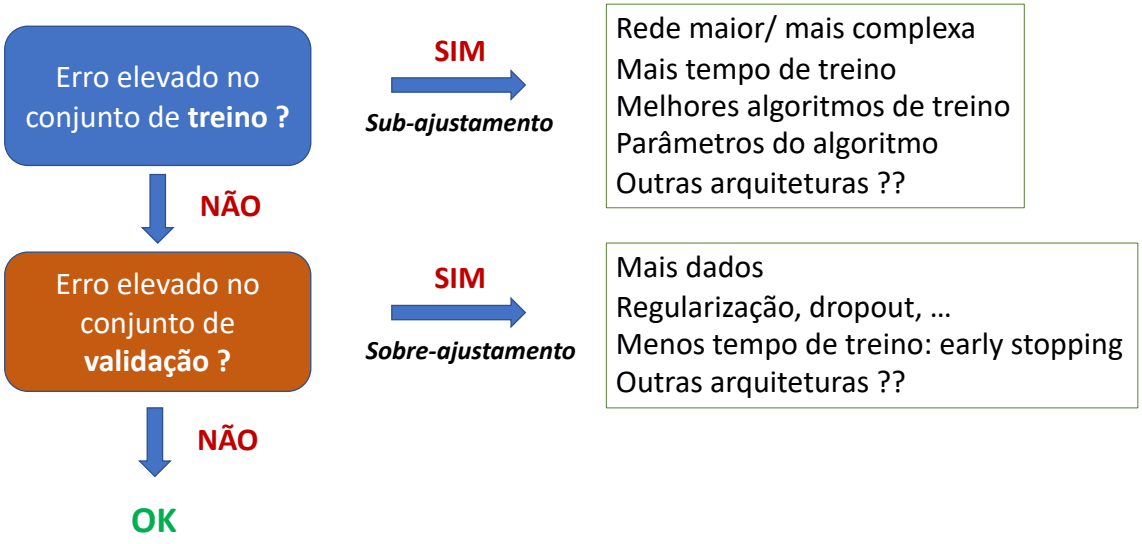
Bias

Erro no conjunto de **validação**

Variance



Workflow genérico



Otimização das redes neuronais: algoritmos de treino

Existem diversos algoritmos de treino de redes neuronais, todos baseados em métodos de gradiente descendente e nas ideias do backpropagation original (chain rule)

Ao longo dos últimos anos foram introduzidas diversas melhorias que resultaram em novos algoritmos mais eficientes que includem o RMSProp, o Adam, e muitos outros

Os algoritmos de treino incluem um conjunto de parâmetros (dependentes do algoritmo) que podem ser otimizados tal como outros hiper-parâmetros e dos quais se destaca a **taxa de aprendizagem (α)**

Melhorias ao algoritmos de treino: batches

Consideração de treino em **mini-batches** (lotes contendo parte dos exemplos) em vez de considerar todos os exemplos disponíveis para calcular erros e suas derivadas e atualizar os pesos

Tamanho dos batches a considerar varia tipicamente entre os 64 e os 512, sendo aconselhado usar bases de 2; *batch* completo deve poder ser guardado na memória do CPU/GPU

Este valor pode ser otimizado como um híper-parâmetro

Se o conjunto de dados for pequeno, pode usar-se o tamanho do batch igual ao nº de exemplos de treino (algoritmo gradiente descendente orginal)

Se batch = 1 -> *stochastic gradient descent*

Melhorias ao algoritmos de treino: momentum

Algoritmos mais recentes usam **momentum** – cálculo de média móvel das derivadas do erro nas últimas atualizações

Forma de remover algum ruído das atualizações, permitindo usar taxas de aprendizagem mais altas e acelerar a convergência do algoritmo de treino

Acrescenta um parâmetro (β) que define o peso a dar ao valor anterior da média (novo batch contribui com 1 – β) – valor típico de 0.9; pode ser otimizado como outros híper-parâmetros

Melhorias ao algoritmos de treino: RMSProp e Adam

Algoritmo **RMSProp** usa também médias móveis, mas neste caso do quadrado dos erros (usa também um parâmetro β – valor típico 0.999)

Atualização dos pesos divide a derivada calculada pela raíz quadrada desta média móvel (somando um pequeno valor ε para evitar valores de zero; valor típico 10^{-8})

Algoritmo **Adam** combina o momentum com a estratégia do RMSProp, tendo assim dois parâmetros para atualização β1 e β2 e o ε)

Melhorias ao algoritmos de treino: *learning rate decay*

A taxa de aprendizagem (α) é um parâmetro muito importante qualquer que seja o algoritmo usado e normalmente o seu valor necessita ser otimizado

Os algoritmos de treino tipicamente suportam uma alternativa que passa pela diminuição do valor da taxa de aprendizagem multiplicando o seu valor por 1 menos um parâmetro (*decay*) em cada vez que os pesos são atualizados

Melhorias ao algoritmos de treino: *batch normalization*

Em muitos casos, tem-se verificado que há vantagens em normalizar as saídas das camadas internas de uma rede neuronal (em particular de camadas Dense e Convolucionais)

Esta normalização é feita para cada batch usado no treino

A normalização ajuda à convergência do algoritmo de treino e tem um pequeno efeito contra o sobre-ajustamento

```
conv_model.add(layers.Conv2D(32, 3, activation='relu'))
conv_model.add(layers.BatchNormalization())
```

```
dense_model.add(layers.Dense(32, activation='relu'))
dense_model.add(layers.BatchNormalization())
```

Exemplo: conjunto de dados MNIST

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) =
        mnist.load_data()
print(train_images.shape, test_images.shape)
print(len(train_labels), len(test_labels))
```

Carregar os dados

Verificar dimensões

```
from keras.utils import to_categorical
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Ajustar as dimensões para tornar as entradas num vetor 1D por cada exemplo
Standardizar valores dividindo por 255
Converter outputs em variáveis categóricas

Exemplo: DNNs para o MNIST

Definir a arquitetura da rede; feed forward

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Exemplo: definindo algoritmos de treino

Definir algoritmo de treino e seus parâmetros - ver keras.io/optimizers)

```
from keras import optimizers

algo = optimizers.SGD(lr = 0.01, momentum = 0.9, decay = 0.0)
algo = optimizers.RMSProp(lr = 0.001, rho = 0.99, decay = 0.0)
algo = optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

lr – learning rate
momentum – beta
decay – decay LR
rho – beta do RMSProp
beta_1 e beta_2 – betas do Adam

```
network.compile(optimizer=algo,
                loss='categorical_crossentropy',
                metrics=['accuracy'])
network.fit(train_images, train_labels, epochs=5, batch_size=128)
test_loss, test_acc = network.evaluate(test_images, test_labels)
```

Controlando o sobre-ajustamento: hiper-parâmetros

Já estudamos algumas formas de contrariar o problema do sobre-ajustamento, incluindo regularização, dropout e paragem antecipada do treino.

Estes métodos implicam um conjunto de híper-parâmetros que também podem (e devem) ser otimizados:

- Parâmetros lambda das regularizações L1 e L2
- Taxa de *dropout* para cada camada
- Parâmetro *patience* do EarlyStopping

Otimização de hiper-parâmetros

A obtenção de um modelo de alta qualidade para qualquer problema implica tipicamente a necessidade de otimizar o valor de vários híper-parâmetros, relacionados quer com a capacidade de aprendizagem quer com a generalização (evitando overfitting)

O facto de haver muitos parâmetros que podem ser otimizados, e os custos em termos de poder computacional levam a que muitas vezes se tenha que escolher quais os parâmetros mais relevantes a escolher

Na prática, costumam optimizar-se essencialmente:

- Arquitetura da rede, incluindo nº e tipo de camadas, nº de neurónios, funções de ativação
- Algoritmo de treino e seus parâmetros principais (taxa de aprendizagem)
- Parâmetros de regularização(dropout, L1/L2)

Otimização de hiper-parâmetros

A otimização de híper-parâmetro passa por considerar diversas configurações alternativas dos modelos, treiná-los e estimar uma métrica de erro sobre o conjunto de exemplos de validação (ou fazer validação cruzada no conjunto de dados de treino)

Uma hipótese é a **procura em grelha**, testando todas as combinações possíveis de um conjunto valores possíveis para um conjunto de híper-parâmetros

Dadas as inúmeras hipóteses, pode ser preferível uma **procura aleatória** que permita testar mais parâmetros e valores distintos

Exemplo: otimização de hiperparâmetros

```
def setup_model(topo, dropout_rate, input_size, output_size):

   model = Sequential()
   model.add(Dense(topo[0], activation="relu", input_dim = input_size))
   if dropout_rate > 0: model.add(Dropout(dropout_rate))
   for i in range(1,len(topo)):
      model.add(Dense(topo[i], activation="relu"))
      if dropout_rate > 0: model.add(Dropout(dropout_rate))
   model.add(Dense(output_size))
   model.add(Activation('softmax'))

   return model
```

Configura modelos para classificação com multiclasses, recebendo a topologia e a taxa de dropout
Exercícios: adaptar para classificação binária; receber outros parâmetros (e.g. função ativação)

Exemplo: otimização de hiperparâmetros

```
def train_dnn(model, alg, lr, Xtrain, Ytrain, epochs = 5, batch_size = 64):
   if alg == "adam":
      optimizer = optimizers.Adam(lr = lr)
   elif alg == "rmsprop":
      optimizer = optimizers.RMSprop(lr = lr)
   elif alg == "sgd_momentum":
      optimizer = optimizers.SGD(lr = lr, momentum = 0.9)
   else: optimizer = optimizers.SGD(lr = lr)

   model.compile(optimizer = optimizer, loss = "categorical_crossentropy", metrics = ["accuracy"])
   model.fit(Xtrain, Ytrain, epochs = epochs, batch_size = batch_size, verbose = 0)

   return model
```

Treina modelo (DNN) para problema multiclasse; recebe algoritmo, taxa aprendizagem, epochs, batch_size fixos
Exercício: adaptar para outras métricas e para problemas de classificação binária

Exemplo: otimização de hiperparâmetros

```
def dnn_optimization(opt_params, Xtrain, Ytrain, Xval, Yval, iterations = 10, verbose = True):
   from random import choice
   if verbose: print("Topology\tDropout\tAlgorithm\tLRate\tValLoss\tValAcc\n")
   best_acc = None
   input_size = Xtrain.shape[1]
   output_size = Ytrain.shape[1]

   if "topology" in opt_params: topologies = opt_params["topology"]
   else: topologies = [[100]]
   if "algorithm" in opt_params: algs = opt_params["algorithm"]
   else: algs = ["adam"]
   if "lr" in opt_params: lrs = opt_params["lr"]
   else: lrs = [0.001]
   if "dropout" in opt_params: dropouts = opt_params["dropout"]
   else: dropouts= [0.0]
…
```

Exemplo: otimização de hiperparâmetros

```
  for it in range(iterations):
     topo = choice(topologies)
     dropout_rate = choice(dropouts)
     dnn = setup_model (topo, dropout_rate, input_size, output_size)
     alg = choice(algs)
     lr = choice(lrs)
     dnn = train_dnn(dnn, alg, lr, Xtrain, Ytrain, )
     val_loss, val_acc = dnn.evaluate(Xval, Yval, verbose = 0)

     if verbose:
        print(topo, "\t", dropout_rate, "\t", alg, "\t", lr, "\t", val_loss, "\t", val_acc)

     if best_acc is None or val_acc > best_acc:
        best_acc = val_acc
        best_config = (topo, dropout_rate, alg, lr)

  return best_config, best_acc
```

Exercício:
- Adaptar para outros hiper-parãmetros
- Adaptar para problemas de classificação binária e regressão
- Adaptar para outras arquitecturas

Exemplo: otimização de hiperparâmetros

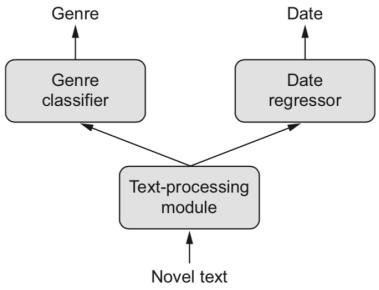
```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
…

opt_pars = {"topology":[[100], [100,50], [250], [250,100]],
          "algorithm": [ "adam", "rmsprop", "sgd_momentum"],
          "lr": [0.01, 0.001],
          "dropout": [0, 0.2, 0.5]}

best_config, best_acc = dnn_optimization(opt_pars, train_images,
                              train_labels, test_images, test_labels)
print(best_config)
print(best_acc)
```

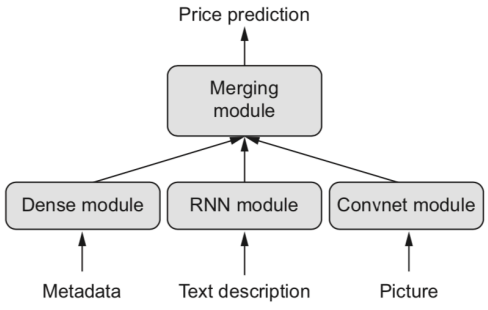
Problemas com múltiplas entradas e saídas

Os exemplos que vimos até ao momento tinham sempre um único tipo de entrada e uma variável de saída

Em alguns casos, podemos ter problemas em que faça sentido ter entradas de vários tipos (textuais, imagens, etc) e em que possamos ter várias variáveis de saída

Nesses casos, não faz sentido treinar modelos individuais pois as entradas/ saídas não são independentes





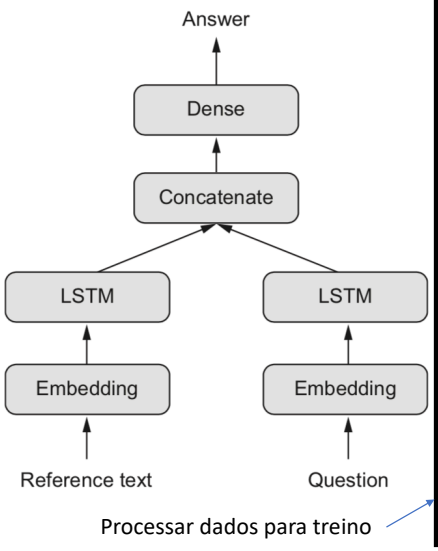
API functional do Keras

Com a API funcional podemos manipular diretamente tensors e montar modelos de forma mais flexível do que usando a classe **Sequential**

```
from keras import layers, input
from keras.models import Model

input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
model.summary()
...
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
model.fit(x_train, y_train, epochs=10, batch_size=128)
```

Exemplo de um modelo sequencial usando a API funcional

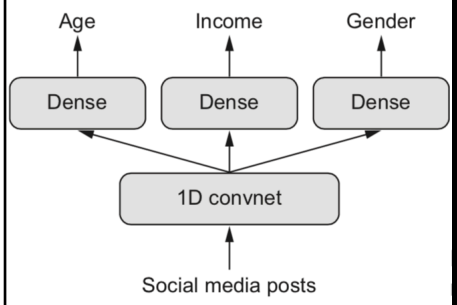
Modelos com múltiplas entradas



Processar dados para treino

```
from keras import layers, input
from keras.models import Model

text_input= Input(shape=(None,), dtype='int32', name='text')

embedded_text = layers.Embedding(64, 10000)(text_input)
encoded_text = layers.LSTM(32)(embedded_text)
quest_input= Input(shape=(None,), dtype='int32', name='quest')
embedded_quest = layers.Embedding(64, 10000)(quest_input)
encoded_quest= layers.LSTM(32)(embedded_quest)
concaten = layers.concatenate([encoded_text, encoded_quest],
                              axis = -1)
answer = layers.Dense(500,activation="softmax")(concaten)
model = Model([text_input, question_input], answer)
model.compile (...)
...
model.fit([text_data, quest_data], answers_data, ...)
```

Modelos com múltiplas saídas

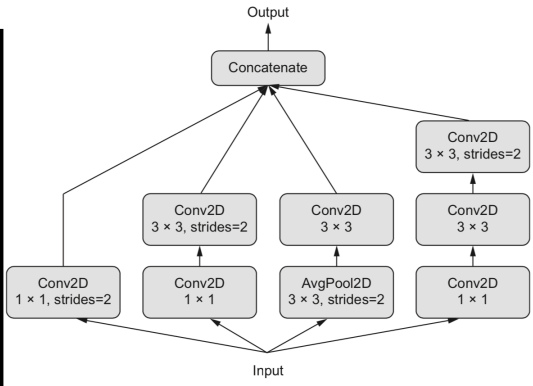



```
from keras import layers, input
from keras.models import Model

posts_input= Input(shape=(None,), dtype='int32', name=posts')

embedded_posts = layers.Embedding(256, 50000)(posts_input)
x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)
x = layers.MaxPooling1D(5)(x)
…
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128,activation="relu")(x)
age_pred = layers.Dense(1, name='age')(x)
income_pred = …
gender_pred = layers.Dense(1, activation='sigmoid', name='gender')(x)

model = Model(posts_input, [age_pred, income_pred, gender_pred])
model.compile (…, loss= ['mse', 'categorical_crossentropy',
                         'binary_crossentropy'])
…
model.fit posts, [age_targets, income_targets, gender_targets] )
```

Processar dados para treino

Modelos com topologias definidas por grafos acíclicos

```
from keras import layers

branch_a = layers.Conv2D(128, 1, …)(x)
branch_b = layers.Conv2D(128, 1,…)(x)
branch_b = layers.Conv2D(128, 3,…)(branch_b)
branch_c = layers.AveragePooling2D(3,…)(x)
branch_c = layers.Conv2D(128, 3,…)(branch_c)
branch_d = layers.Conv2D(128, 1, …)(x)
branch_d = layers.Conv2D(128, 3,…)(branch_d)
branch_d = layers.Conv2D(128, 3,…)(branch_d)

output = layers.concatenate(
    [branch_a, branch_b, branch_c, branch_d],
        axis=-1)
```



Exemplo: Inception module

Outras possibilidades

API functional do Keras permite:

- Definir modelos completes como “camadas”, montando com outros modelos
- Partilhar pesos entre camadas distintas da rede
- Implementar redes com conexões “residuais”: somar outputs de camadas distintas

Uso de Callbacks permite criar “logs” do processo de treino; usando o TensorBoard podem-se visualizar estes logs para fazer debug ao processo de treino

Exemplos avançados

- Geração de textos com LSTMs

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.1-text-generation-with-lstm.ipynb

- Neural style transfer

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.3-neural-style-transfer.ipynb

- Geração de imagens com VAEs e GANs

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.4-generating-images-with-vaes.ipynb

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.5-introduction-to-gans.ipynb