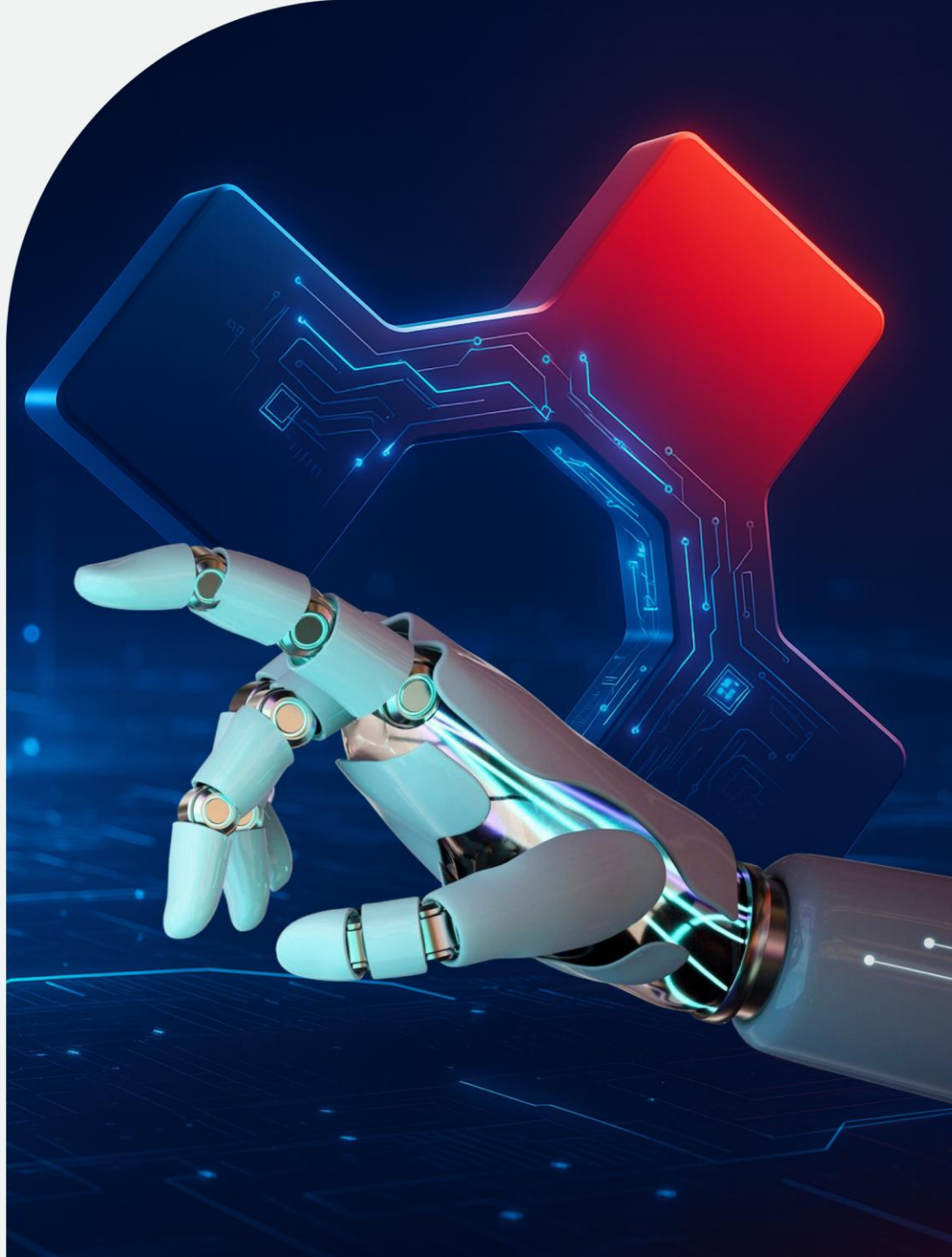
NCIA

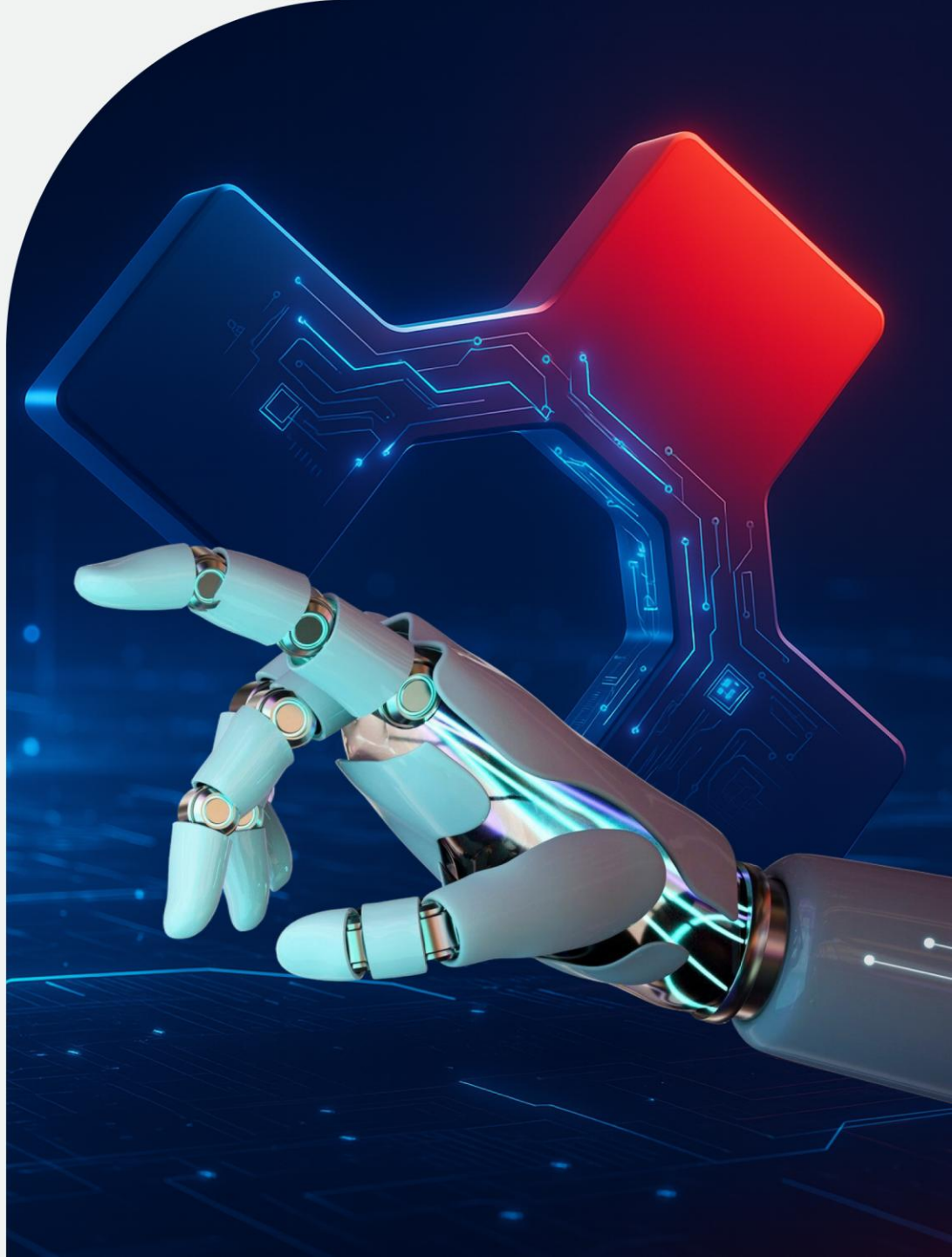
Núcleo de Capacitação em Inteligência Artificial



NCIA | FOXCONN FPF tech

Introduction to Artificial Neural Networks

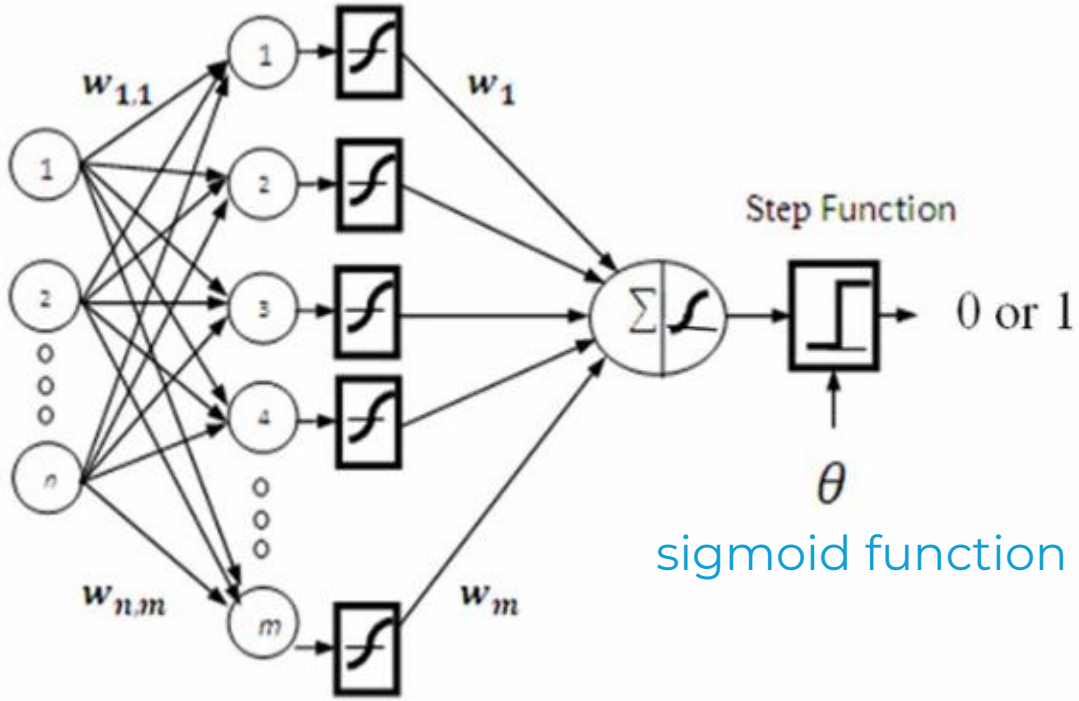
Classification MLPs; Implementing MLPs with Keras; Building an Image Classifier Using the Sequential API; Building a Regression MLP Using the Sequential API; Building Complex Models Using the Functional API.



NCIA | FOXCONN FPF tech

MLPs em Tarefas de Classificação

Os Perceptrons Multicamadas (MLPs) também são usados para tarefas de classificação. Em um problema binário, utilizamos apenas um neurônio de saída com função sigmoide, que gera um valor entre **0** e **1**. Esse valor representa a probabilidade da classe positiva; a probabilidade da classe negativa é **1** menos esse valor.

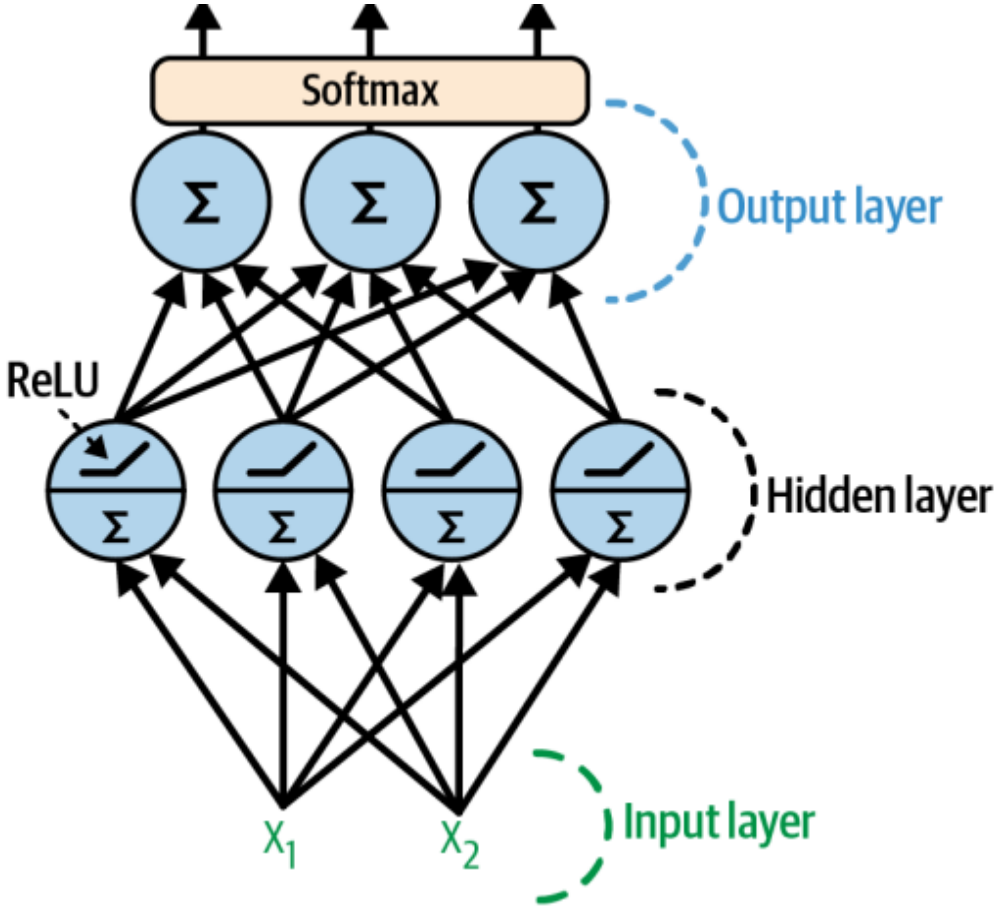


Classificação Multirrótulo

MLPs podem lidar facilmente com classificação multirrótulo, em que cada exemplo pode pertencer a várias classes simultaneamente. Por exemplo, um sistema pode prever se um e-mail é spam ou não, e se é urgente ou não. Para isso, usamos um neurônio de saída para cada rótulo, todos com função sigmoide. As probabilidades não precisam somar **1**, permitindo combinações como “spam urgente” ou “não urgente e não spam”.

Quando cada exemplo pertence a apenas uma entre várias classes possíveis, usamos um neurônio por classe e a função softmax na saída.

O softmax converte as saídas em probabilidades que somam 1, representando a probabilidade de cada classe ser a correta.
Esse tipo de tarefa é chamado de classificação multiclasse.



Função de perda

Como o MLP prevê distribuições de probabilidade, a função de perda mais adequada é a entropia cruzada (cross-entropy).
O Scikit-Learn oferece a classe MLPClassifier, semelhante ao MLPRegressor, mas que minimiza cross-entropy em vez do erro quadrático médio (MSE).
Um bom teste inicial é o conjunto de dados Iris, com uma única camada oculta de 5 a 10 neurônios, após o escalonamento das variáveis.

A Tabela ao lado resume a arquitetura típica de um MLP de classificação, incluindo camadas, funções de ativação e perda.
Recomenda-se praticar com o TensorFlow Playground, ajustando hiperparâmetros como número de camadas, neurônios e funções de ativação, para compreender melhor o comportamento das redes MLP.

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

1. Implementing MLPs with Keras

Keras é a API de alto nível do TensorFlow para construção, treinamento e execução de redes neurais profundas.
Criado por François Chollet em 2015 como um projeto open source, o Keras tornou-se popular por sua facilidade de uso, flexibilidade e design elegante.
A partir de agora, utilizaremos o Keras para implementar um Perceptron Multicamadas (MLP) aplicado à classificação de imagens.

2. Building an Image Classifier Using the Sequential API

Usaremos a API Sequencial do Keras para construir nosso classificador de imagens.
O conjunto Fashion MNIST será utilizado, contendo 70.000 imagens em tons de cinza de 28 × 28 pixels, divididas em 10 classes de roupas e acessórios.
Esse conjunto é mais desafiador que o MNIST: enquanto um modelo linear atinge cerca de 92% de acurácia no MNIST, ele alcança apenas 83% no Fashion MNIST.
A API Sequencial permite criar o modelo camada por camada, facilitando a definição de camadas densas e funções de ativação.

2.1 Using Keras to load the dataset

O Keras fornece funções utilitárias para baixar e carregar datasets comuns, incluindo Fashion MNIST. O Fashion MNIST já vem embaralhado e dividido em 60.000 imagens de treino e 10.000 imagens de teste.
Para validação durante o treinamento, podemos reservar as últimas 5.000 imagens do conjunto de treino. Essas funções permitem focar na construção do modelo, sem se preocupar com o pré-processamento manual dos dados:

```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

Ao carregar MNIST ou Fashion MNIST com Keras, cada imagem é representada como um array 2D de 28 × 28 pixels, diferente do array 1D de tamanho 784 usado pelo Scikit-Learn. Além disso, os pixels são inteiros de 0 a 255, enquanto em algumas implementações com Scikit-Learn podem ser floats.
É importante verificar o formato (shape) e o tipo de dado (dtype) do conjunto de treinamento antes de alimentar o modelo.

```
>>> X_train.shape
(55000, 28, 28)
>>> X_train.dtype
dtype('uint8')
```

Para simplificar, reduziremos as intensidades dos pixels para o intervalo de 0 a 1, dividindo-as por 255,0 (isso também as converte em flutuantes):

```
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

Com MNIST, quando o rótulo é igual a 5, significa que a imagem representa o dígito 5 escrito à mão. Fácil. Para MNIST de Moda, no entanto, precisamos da lista de nomes de classes para saber com o que estamos lidando:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

Por exemplo, a primeira imagem no conjunto de treinamento representa uma bota de cano curto:

```
>>> class_names[y_train[0]]
'Ankle boot'
```

A Figura a seguir mostra alguns exemplos do conjunto de dados Fashion MNIST :

2.2 Creating the model using the sequential API

Agora vamos construir a rede neural! Aqui está uma classificação MLP com duas camadas ocultas:

```
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Input(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Primeiro, definimos a seed do TensorFlow para que os pesos iniciais sejam os mesmos a cada execução, garantindo reprodutibilidade.
Em seguida, criamos um modelo Sequencial, que empilha camadas de forma linear.
A primeira camada é a Input layer, seguida de uma Flatten layer que transforma cada imagem 28 × 28 em um vetor 1D de 784 elementos.
Depois adicionamos a primeira Dense layer oculta com 300 neurônios e ReLU, seguida de uma segunda Dense layer com 100 neurônios e ReLU.
Finalmente, adicionamos a camada de saída Dense com 10 neurônios e softmax, uma para cada classe exclusiva.

O método summary() do Keras exibe todas as camadas do modelo, incluindo o nome de cada camada, a forma da saída (output shape) e o número de parâmetros.
O None no output shape indica que o tamanho do batch pode variar.
No final, o resumo mostra o total de parâmetros, dividindo-os em treináveis e não-treináveis.
Neste modelo, temos apenas parâmetros treináveis, mas alguns modelos podem conter parâmetros não-treináveis:

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 300)               235500

 dense_1 (Dense)             (None, 100)               30100

 dense_2 (Dense)             (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

As camadas Dense podem ter muitos parâmetros. A primeira camada oculta com 784 entradas e 300 neurônios possui 235.500 parâmetros no total.
Cada camada deve ter um nome único. Keras gera nomes automaticamente em snake_case e garante unicidade global, facilitando a combinação de modelos sem conflitos.
É possível listar todas as camadas de um modelo usando o atributo layers ou acessar uma camada específica pelo nome com get_layer(name):

```
>>> model.layers
[<keras.layers.core.flatten.Flatten at 0x7fa1dea02250>,
 <keras.layers.core.dense.Dense at 0x7fa1c8f42520>,
 <keras.layers.core.dense.Dense at 0x7fa188be7ac0>,
 <keras.layers.core.dense.Dense at 0x7fa188be7fa0>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
```

Em uma Dense layer, os pesos de conexão são inicializados aleatoriamente para quebrar a simetria, e os biases são inicializados com zeros.
É possível personalizar a inicialização usando kernel_initializer para os pesos e bias_initializer para os biases.
Keras oferece diversos inicializadores, listados em https://keras.io/api/layers/initializers, e a escolha do inicializador pode impactar a convergência e performance do modelo.

2.3 Compiling the model using the sequencial API

Para classificação multiclasse com rótulos esparsos (um índice por instância), usamos softmax na saída e sparse_categorical_crossentropy como função de perda.
Se os rótulos forem vetores one-hot, usamos categorical_crossentropy.
Para classificação binária ou multilabel, usamos sigmoid na saída e binary_crossentropy como perda.
Escolher a ativação e a perda corretamente garante que o modelo aprenda probabilidades coerentes.

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

O otimizador "sgd" indica que o modelo será treinado usando Stochastic Gradient Descent.
Keras aplica backpropagation para calcular gradientes e atualizar os pesos.
Otimizadores mais avançados serão discutidos nas aulas posteriores e melhoram a atualização dos pesos, não o cálculo de gradientes.
Para medir o desempenho do classificador, usamos metrics=["accuracy"], que indica a proporção de previsões corretas durante treinamento e avaliação.

2.4 Training and evaluating the model

Agora o modelo está pronto para ser treinado. Para isso, basta chamar seu método fit():

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Epoch 1/30
1719/1719 [==============================] - 2s 989us/step
  - loss: 0.7220 - sparse_categorical_accuracy: 0.7649
  - val_loss: 0.4959 - val_sparse_categorical_accuracy: 0.8332
Epoch 2/30
1719/1719 [==============================] - 2s 964us/step
  - loss: 0.4825 - sparse_categorical_accuracy: 0.8332
  - val_loss: 0.4567 - val_sparse_categorical_accuracy: 0.8384
[...]
Epoch 30/30
1719/1719 [==============================] - 2s 963us/step
  - loss: 0.2235 - sparse_categorical_accuracy: 0.9200
  - val_loss: 0.3056 - val_sparse_categorical_accuracy: 0.8894
```

Para treinar uma rede no Keras, usamos fit(X_train, y_train, epochs=...), escolhendo explicitamente o número de épocas (o padrão é 1 e quase nunca basta). É recomendável informar também um conjunto de validação: ao fim de cada época, o Keras calcula a loss e as métricas extras nesse conjunto, permitindo acompanhar o desempenho "real". Se o modelo vai muito melhor no treino do que na validação, é um sinal de overfitting (ou até de algum bug, como descompasso entre dados de treino e validação).

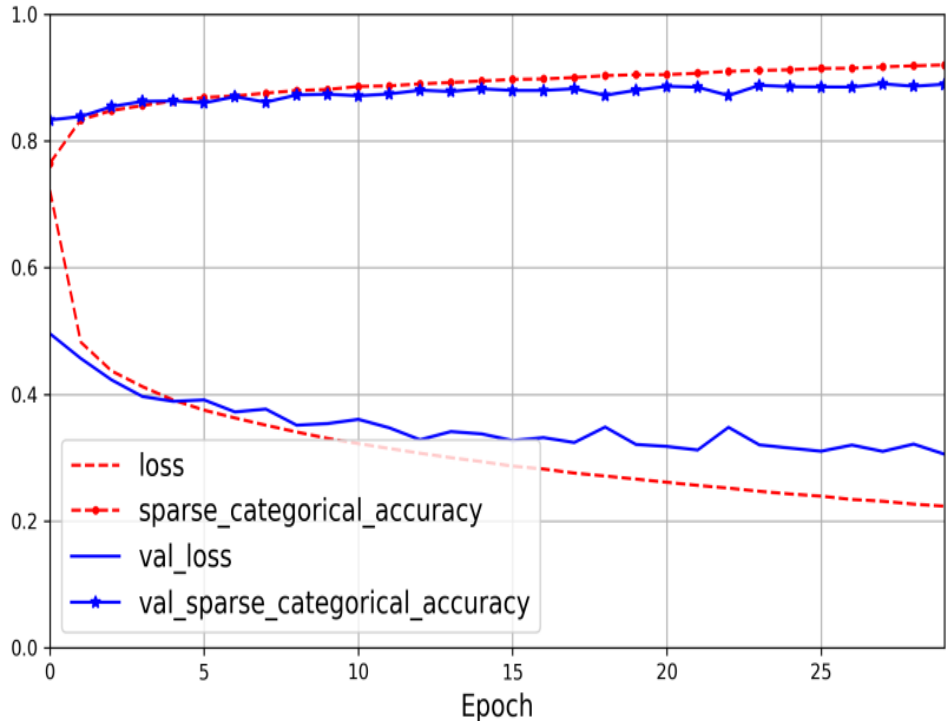
Durante o processo, o Keras exibe uma barra de progresso baseada em mini-lotes; o batch_size padrão é 32. Em um exemplo com 55.000 amostras, isso resulta em 1.719 batches por época (1.718 de 32 e 1 de 24). Na figura a seguir, você vê o tempo médio por amostra, além de loss e acurácia tanto em treino quanto em validação. É desejável que a loss de treino caia ao longo das épocas; se a acurácia de validação ficar apenas um pouco abaixo da de treino (e.g., ~88,94% após 30 épocas), há algum overfitting, mas possivelmente controlado.

Quando há desbalanceamento de classes, podemos corrigir o viés no cálculo da loss com class_weight (mais peso para classes raras, menos para frequentes). Se precisarmos de pesos por instância específica, usamos sample_weight. Fornecendo ambos, o Keras multiplica os pesos. Para a validação, só sample weights podem ser passados (como terceiro item em validation_data); class weights não se aplicam ao conjunto de validação.

Por fim, fit() retorna um objeto History, com history.params (parâmetros de treino), history.epoch (as épocas percorridas) e, principalmente, history.history (um dicionário com as séries de loss e métricas por época, em treino e validação). Convertendo history.history para um DataFrame do Pandas e chamando .plot(), obtemos curvas de aprendizado claras para diagnosticar evolução, overfitting e estabilização do treinamento.

```
import matplotlib.pyplot as plt
import pandas as pd

pd.DataFrame(history.history).plot(
    figsize=(8, 5), xlim=[0, 29], ylim=[0, 1], grid=True, xlabel="Epoch",
    style=["r--", "r--.", "b-", "b-*"])
plt.show()
```



Além dos hiperparâmetros principais, ajuste também o batch size. Quando satisfeito com a validação, use model.evaluate() no conjunto de teste para medir o erro de generalização. O método retorna loss e métricas calculadas no teste e aceita argumentos como batch_size e sample_weight.

```
>>> model.evaluate(X_test, y_test)
313/313 [==============================] - 0s 626us/step
  - loss: 0.3243 - sparse_categorical_accuracy: 0.8864
[0.32431697845458984, 0.8863999843597412]
```

2.4 Using the model to make predictions

Agora, vamos usar o método predict() do modelo para fazer previsões sobre novas instâncias. Como não temos novas instâncias reais, usaremos apenas as três primeiras instâncias do conjunto de teste:

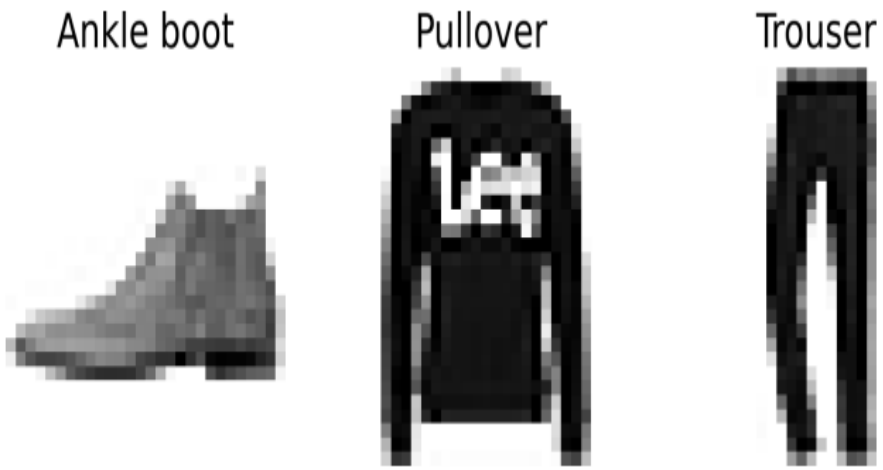
```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.02, 0.  , 0.97],
       [0.  , 0.  , 0.99, 0.  , 0.01, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

O modelo estima uma probabilidade para cada classe (0 a 9), assim como o método predict_proba() do Scikit-Learn, e a soma dessas probabilidades é 1 devido ao uso da função *softmax*. Por exemplo, para uma imagem, pode prever 96% para a classe 9 (bota de tornozelo), 2% para a classe 7 (tênis) e 1% para a classe 5 (sandália), com as demais sendo desprezíveis. Isso indica alta confiança de que a imagem representa um calçado, provavelmente uma bota. Se quisermos apenas a classe mais provável, usamos o método argmax() para obter o índice correspondente.

```
>>> import numpy as np
>>> y_pred = y_proba.argmax(axis=-1)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Aqui, o classificador realmente classificou todas as três imagens corretamente (essas imagens são mostradas na Figura ao lado):

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1], dtype=uint8)
```



Agora você sabe como usar a API sequencial para construir, treinar e avaliar uma classificação MLP. Mas e quanto à regressão?

3. Building a Regression MLP Using the Sequential API

Vamos resolver o problema de preços de casas com uma MLP em Keras, usando a API Sequential. A ideia é prever um único valor contínuo (o preço) a partir de atributos tabulares. Mantemos a arquitetura já usada antes: três camadas ocultas com 50 neurônios cada. O foco aqui é mostrar como montar, treinar, avaliar e usar essa rede para regressão, destacando o que muda em relação à classificação.

O modelo tem entrada tabular, passa por três camadas densas com 50 neurônios cada (ativação padrão ReLU é comum, ainda que o texto não a detalhe), e termina em uma camada de saída com um único neurônio sem função de ativação, pois estamos prevendo um número real. Essa saída linear permite que o modelo represente qualquer valor contínuo, em vez de probabilidades.

Na regressão, a camada de saída tem apenas um neurônio e não usa ativação; a função de perda é o erro quadrático médio (MSE) e a métrica de acompanhamento é a raiz do erro quadrático médio (RMSE), que tem a mesma unidade do alvo e é mais interpretável. O otimizador usado é o Adam, como no MLPRegressor do Scikit-Learn, o que facilita a comparação entre as abordagens.

Não usamos Flatten porque os dados já são vetores; em vez disso, colocamos uma camada Normalization como primeira camada, que desempenha o papel do StandardScaler do Scikit-Learn. Essa camada precisa “aprender” as estatísticas do treino com adapt(X_train) antes de chamar model.fit(...), garantindo que cada recurso seja normalizado consistentemente dentro do próprio grafo do Keras.

Fluxo em Keras (Sequential): build → compile → fit → evaluate → predict

Com a API Sequential, definimos as camadas em sequência, depois compilamos o modelo informando otimizador, perda e métricas, treinamos com fit(...), avaliamos com evaluate(...) para obter o RMSE e, por fim, usamos predict(...) para gerar estimativas de preço. O processo é análogo ao de classificação, mas com as trocas específicas de regressão (saída linear, MSE e RMSE).

```
tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

A API Sequential é limpa e direta, mas fica limitada quando precisamos de arquiteturas não lineares (com ramificações), camadas compartilhadas, ou modelos com múltiplas entradas e/ou múltiplas saídas. Em cenários reais — como combinar diferentes fontes de dados tabulares e textuais, prever vários alvos ao mesmo tempo ou construir conexões “em paralelo” — essa flexibilidade é essencial.

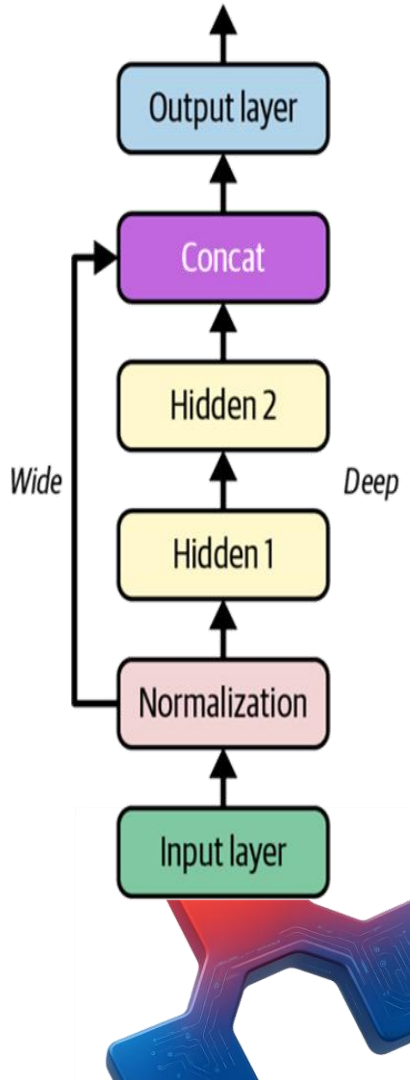
A API Funcional permite desenhar redes como grafos dirigidos acíclicos: você declara tensores de entrada, aplica camadas como funções (cada chamada transforma tensores), e conecta caminhos diferentes antes da saída. Assim, é possível criar topologias complexas mantendo clareza e reprodutibilidade, com suporte nativo a múltiplas entradas/saídas e compartilhamento de pesos entre ramos. Em resumo: quando o Sequential não basta, a API Funcional oferece a expressividade necessária sem perder a simplicidade do Keras.

3. Building Complex Models Using the Functional API

A principal característica da Wide & Deep Neural Network é que ela conecta todas ou parte das entradas diretamente à camada de saída.
Isso permite que a rede aprenda padrões complexos por meio do caminho profundo (deep path) e regras simples pelo caminho curto (short path).

Uma rede Wide & Deep combina dois caminhos ao mesmo tempo: um “caminho profundo” (deep), com várias camadas para aprender padrões complexos, e um “caminho curto/largo” (wide), que liga parte (ou todos) os atributos diretamente à camada de saída para capturar regras simples e relações lineares de forma imediata. Essa arquitetura, proposta em 2016 por Heng-Tze Cheng e colaboradores, permite que o modelo “memorize” padrões diretos enquanto “generaliza” via representações profundas, superando limitações de MLPs puramente sequenciais.

A figura apresenta duas rotas partindo das entradas: no ramo “deep”, as features passam por uma pilha de camadas densas até chegar à saída; no ramo “wide”, as próprias entradas (ou combinações simples delas) conectam-se diretamente à camada de saída. Visualmente, vemos as setas das entradas se bifurcando: uma sequência longa atravessa múltiplas camadas (deep), enquanto outra rota curta vai quase reta até o nó de saída (wide). A mensagem visual é que o modelo aprende simultaneamente representações profundas e regras de baixa complexidade, fundindo ambos os sinais na predição final.



Vamos construir uma rede neural para resolver o problema habitacional da Califórnia:

```
normalization_layer = tf.keras.layers.Normalization()
hidden_layer1 = tf.keras.layers.Dense(30, activation="relu")
hidden_layer2 = tf.keras.layers.Dense(30, activation="relu")
concat_layer = tf.keras.layers.Concatenate()
output_layer = tf.keras.layers.Dense(1)

input_ = tf.keras.layers.Input(shape=X_train.shape[1:])
normalized = normalization_layer(input_)
hidden1 = hidden_layer1(normalized)
hidden2 = hidden_layer2(hidden1)
concat = concat_layer([normalized, hidden2])
output = output_layer(concat)

model = tf.keras.Model(inputs=[input_], outputs=[output])
```

No código apresentado, as primeiras cinco linhas criam todas as camadas necessárias para construir o modelo. Essas camadas incluem operações de normalização, camadas densas para processamento, uma camada de concatenação e a camada de saída.
As seis linhas seguintes usam essas camadas como funções, conectando a entrada à saída de forma simbólica, ou seja, sem processar dados reais ainda.
A última linha cria um objeto Keras Model, que indica ao Keras quais são as entradas e saídas do modelo, permitindo que ele saiba como propagar informações e treinar a rede.

Primeiro, criamos cinco camadas importantes:

- Uma camada Normalization, que padroniza os valores das entradas, garantindo que todas as features tenham média zero e desvio padrão igual a um. Isso facilita o aprendizado e acelera a convergência.
- Duas camadas Dense com 30 neurônios cada, usando função de ativação ReLU, que permite modelar relações não lineares complexas. Uma camada Concatenate, que combina a saída de diferentes camadas ou caminhos, permitindo que a rede integre informações de forma flexível.
- Uma camada Dense final com um único neurônio, sem função de ativação, que gera a saída final do modelo, adequada para regressão ou problemas onde não queremos limitar a saída a um intervalo específico.

Em seguida, criamos um objeto Input chamado input_. Este objeto define a forma e o tipo de dados que o modelo vai receber, funcionando como uma especificação simbólica da entrada. O nome input_ é usado para não conflitar com a função padrão do Python input(). Um modelo pode ter múltiplas entradas, o que é especialmente útil em arquiteturas não sequenciais, como Wide & Deep Neural Networks, permitindo processar subconjuntos diferentes de features por caminhos distintos.

A camada Normalization é aplicada como se fosse uma função, passando o objeto Input. Isso é característico da Functional API do Keras, que permite construir modelos de forma simbólica e flexível. O resultado, normalized, também é simbólico: nenhum dado real é processado, apenas a arquitetura é construída. Essa abordagem permite combinar camadas de formas complexas, como caminhos paralelos, múltiplas entradas ou saídas, ou concatenações, sem precisar de loops ou lógica imperativa para definir o fluxo de dados.

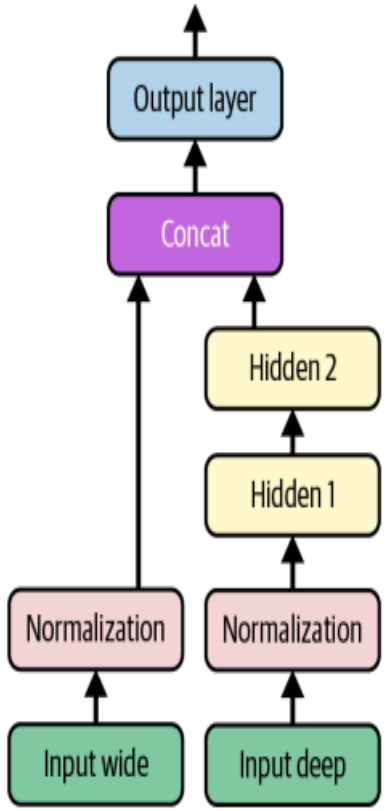
Em seguida, conectamos normalized à primeira camada oculta hidden_layer, gerando hidden, e depois passamos hidden para hidden_layer , gerando hidden. Até aqui, estamos criando um fluxo sequencial simbólico, semelhante a uma MLP tradicional, mas ainda podemos adicionar caminhos paralelos ou combinações mais complexas. Esse passo é fundamental para construir a parte “deep” da rede, que vai capturar padrões complexos e não lineares nos dados.

Após construir o modelo, o fluxo de trabalho é o mesmo de sempre:

- Compilação: definimos o otimizador, função de perda e métricas.
- Adaptação da Normalization layer: calcula estatísticas das entradas reais para normalizar os dados corretamente.
- Treinamento (fit): o modelo aprende a ajustar os pesos para minimizar a função de perda.
- Avaliação e previsões: podemos medir o desempenho do modelo e gerar previsões para novos dados. Tudo isso funciona da mesma forma que em uma MLP sequencial, mas agora com maior flexibilidade para arquiteturas complexas.

Se quisermos enviar subconjuntos diferentes de features por caminhos distintos, como na Figura ao lado, podemos usar múltiplas entradas.
Exemplo:

- Features 0 a 4 passam pelo caminho wide, conectando diretamente à saída para capturar padrões simples.
- Features 2 a 7 passam pelo caminho deep, passando por múltiplas camadas densas para capturar padrões complexos.
  Figura ao lado: mostra visualmente dois caminhos paralelos:
- O caminho profundo (deep) processa algumas features por múltiplas camadas densas.
- O caminho curto (wide) recebe outras features e conecta diretamente à saída.
  Essa abordagem combina aprendizado de padrões simples e complexos na mesma rede, aumentando a capacidade de generalização.



Alguns pontos importantes neste exemplo, em comparação com o anterior: Cada camada Dense é criada e chamada na mesma linha. Isso é uma prática comum em Keras, pois torna o código mais conciso sem perder clareza. No entanto, isso não funciona com a camada normalization, porque precisamos manter uma referência à camada para chamar o método adapt() antes de treinar o modelo.

Para combinar os caminhos wide e deep, usamos tf.keras.layers.concatenate(), que:

- Cria automaticamente uma camada Concatenate.
- Aplica essa camada aos inputs fornecidos, unindo suas saídas em um único tensor simbólico. Essa abordagem simplifica o código e permite que múltiplos caminhos paralelos sejam combinados de forma elegante.

Ao criar o modelo, especificamos inputs=[input_wide, input_deep], pois temos duas entradas diferentes. O Keras entende que cada entrada deve ser processada separadamente antes da concatenação, permitindo que cada caminho aprenda padrões distintos das features que recebe.

Ao treinar o modelo com fit(), não passamos mais uma única matriz X_train. Em vez disso, passamos um par de matrizes: (X_train_wide, X_train_deep), correspondentes a cada entrada da rede. O mesmo se aplica aos conjuntos de validação (X_valid), teste (X_test) e novas entradas (X_new) quando usamos evaluate() ou predict(). Isso garante que cada caminho da rede receba as features corretas e aprenda tanto padrões simples quanto complexos de forma simultânea.

Agora podemos compilar o modelo normalmente, mas quando chamamos o método fit(), em vez de passar uma única matriz de entrada X_train, precisamos passar um par de matrizes (X_train_wide, X_train_deep), uma por entrada. O mesmo vale para X_valid e também para X_test e X_new quando você chama evaluate() ou predict().

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])

X_train_wide, X_train_deep = X_train[:, :5], X_train[:, 2:]
X_valid_wide, X_valid_deep = X_valid[:, :5], X_valid[:, 2:]
X_test_wide, X_test_deep = X_test[:, :5], X_test[:, 2:]
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]

norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit((X_train_wide, X_train_deep), y_train, epochs=20,
                    validation_data=((X_valid_wide, X_valid_deep), y_valid))
mse_test = model.evaluate((X_test_wide, X_test_deep), y_test)
y_pred = model.predict((X_new_wide, X_new_deep))
```

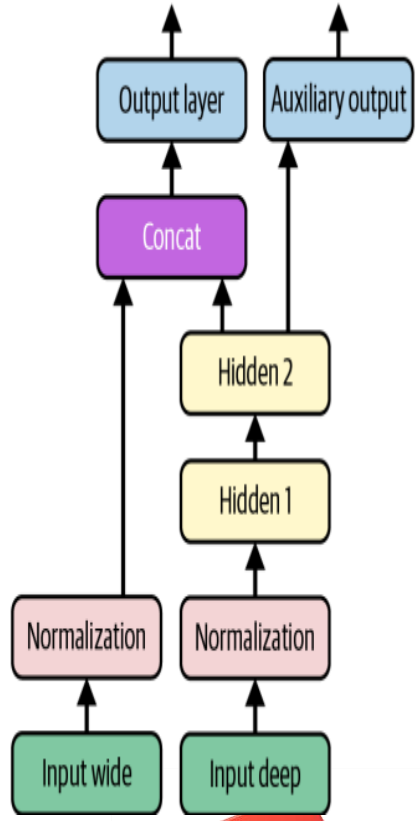
Existem diversos casos em que você pode querer que uma rede neural tenha múltiplas saídas. Um motivo é que a tarefa pode exigir múltiplos tipos de previsão simultaneamente. Por exemplo, ao analisar uma imagem, você pode querer localizar o objeto principal (uma tarefa de regressão) e classificá-lo (uma tarefa de classificação). Neste caso, uma única rede com múltiplas saídas resolve as duas tarefas ao mesmo tempo.

Outro caso é quando você tem múltiplas tarefas independentes baseadas nos mesmos dados.Você poderia treinar uma rede neural separada para cada tarefa, mas muitas vezes obtém melhores resultados ao treinar uma única rede com uma saída por tarefa. Isso acontece porque a rede aprende features compartilhadas que são úteis para todas as tarefas, aumentando a eficiência e a generalização.

Exemplo de multitask learning: Imagine classificar expressões faciais e acessórios em fotos de rostos.

- Uma saída classifica a expressão (sorrindo, surpreso, etc.).
- Outra saída identifica se a pessoa está usando óculos ou não.
  A rede aprende representações internas que servem para ambas as tarefas, aproveitando melhor os dados disponíveis.

Outro uso de múltiplas saídas é como técnica de regularização. Regularização é uma restrição de treinamento cujo objetivo é reduzir o overfitting e melhorar a capacidade de generalização do modelo. Por exemplo, podemos adicionar uma saída auxiliar (auxiliary output) à rede para garantir que partes internas do modelo aprendam algo útil por conta própria, sem depender das camadas posteriores. A figura ao lado mostra um modelo com saída principal e uma saída auxiliar. A saída auxiliar serve como regularização, ajudando a rede a aprender representações mais robustas e generalizáveis.



Adicionar uma saída extra é bem fácil: basta conectá-la à camada apropriada e adicioná-la à lista de saídas do modelo. Por exemplo, o código a seguir constrói a rede representada na Figura anterior:

```
[...]  # Same as above, up to the main output layer
output = tf.keras.layers.Dense(1)(concat)
aux_output = tf.keras.layers.Dense(1)(hidden2)
model = tf.keras.Model(inputs=[input_wide, input_deep],
                       outputs=[output, aux_output])
```

Quando uma rede neural possui múltiplas saídas, cada saída precisa de uma função de perda própria. Ao compilar o modelo, devemos passar uma lista de funções de perda, uma para cada saída. Se passarmos apenas uma função de perda, o Keras assumirá que a mesma perda deve ser usada para todas as saídas, o que nem sempre é adequado.

Por padrão, o Keras calcula todas as perdas separadamente e simplesmente as soma para obter a perda final usada no treinamento. Isso funciona bem, mas em muitos casos nem todas as saídas têm a mesma importância.

Exemplo: em uma rede com saída principal e saída auxiliar (usada como regularização), nos preocupamos muito mais com a saída principal.
A saída auxiliar serve apenas para melhorar o aprendizado interno do modelo, portanto sua perda deve ter peso menor.

Felizmente, ao compilar o modelo, é possível definir os pesos das perdas para cada saída. Dessa forma, podemos atribuir maior peso à saída principal e peso menor à saída auxiliar, controlando a influência de cada perda no treinamento do modelo. Isso ajuda a rede a focar no que é mais importante, mantendo a regularizaçãoproporcionada pela saída auxiliar.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss=("mse", "mse"), loss_weights=(0.9, 0.1), optimizer=optimizer,
              metrics=["RootMeanSquaredError"])
```

Agora, ao treinar o modelo, precisamos fornecer rótulos para cada saída. Neste exemplo, a saída principal e a saída auxiliar devem tentar prever a mesma coisa, portanto, devem usar os mesmos rótulos. Portanto, em vez de passar y_train, precisamos passar (y_train, y_train) ou um dicionário {"output": y_train, "aux_output": y_train} se as saídas forem denominadas "output" e "aux_output". O mesmo vale para y_valid e y_test:

```
norm_layer_wide.adapt(X_train_wide)
norm_layer_deep.adapt(X_train_deep)
history = model.fit(
    (X_train_wide, X_train_deep), (y_train, y_train), epochs=20,
    validation_data=((X_valid_wide, X_valid_deep), (y_valid, y_valid))
)
```

Quando avaliamos o modelo, Keras retorna a soma ponderada das perdas, bem como todas as perdas e métricas individuais:

```
eval_results = model.evaluate((X_test_wide, X_test_deep), (y_test, y_test))
weighted_sum_of_losses, main_loss, aux_loss, main_rmse, aux_rmse = eval_results
```

Da mesma forma, o método predict() retornará previsões para cada saída:

```
y_pred_main, y_pred_aux = model.predict((X_new_wide, X_new_deep))
```

O método predict() retorna uma tupla e não possui um argumento return_dict para obter um dicionário. No entanto, você pode criar um usando model.output_names:

```
y_pred_tuple = model.predict((X_new_wide, X_new_deep))
y_pred = dict(zip(model.output_names, y_pred_tuple))
```

Como você pode ver, é possível construir todos os tipos de arquiteturas com a API funcional. A seguir, veremos uma última maneira de construir modelos Keras.

FPFtech