Introduction to Artificial Neural Networks with Keras

Deep Learning

Stijn Lievens & Sabine Devreese

2023-2024

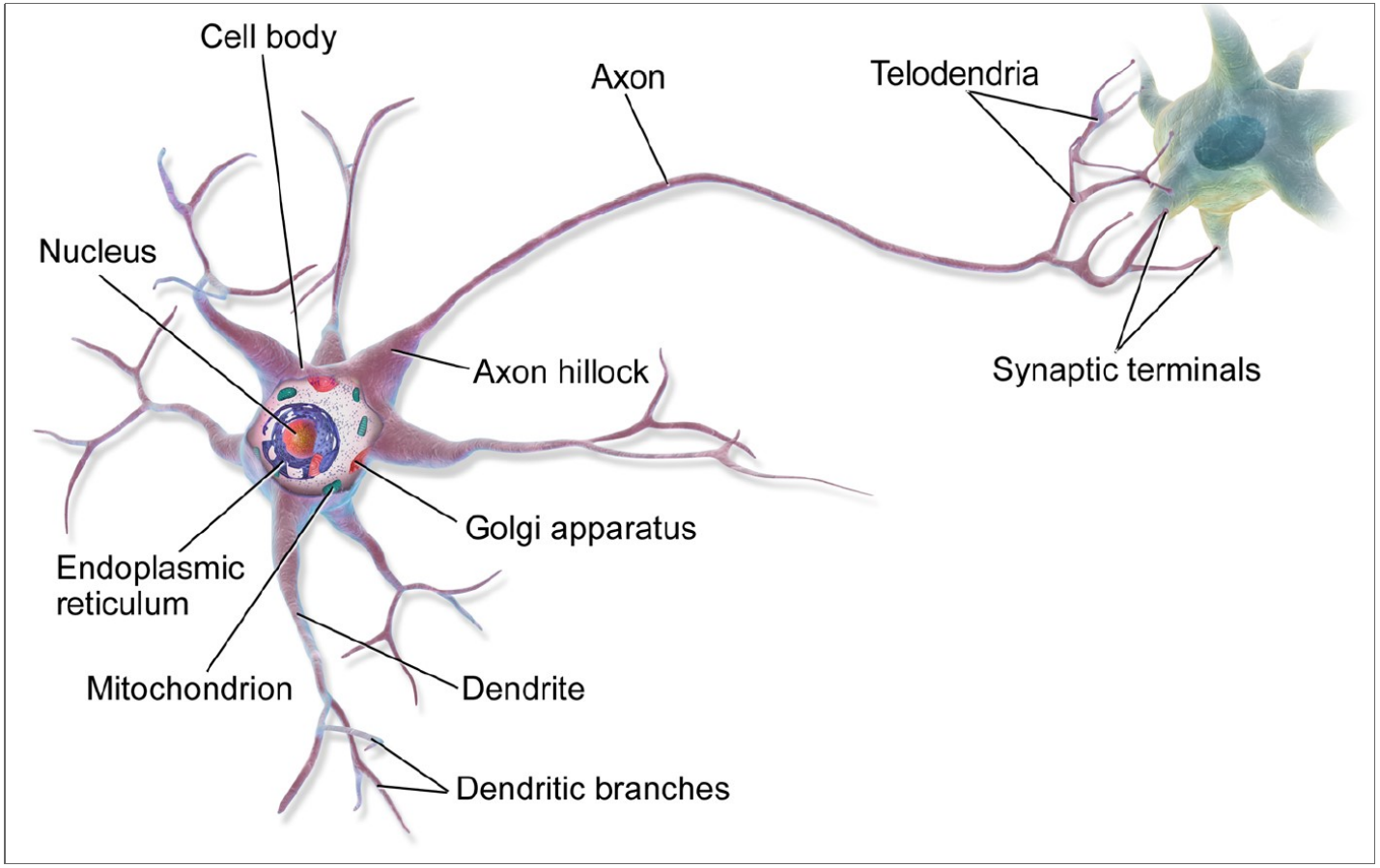
10.1 From Biological to Artificial Neurons

10.1.1 Biological Neurons

Artificial Neural Networks

- An **Artificial Neural Network** (**ANN**) is a Machine Learning model inspired by the networks of biological neurons found in our brains.
- They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks such as
  - classifying billions of images (e.g., Google Images)
  - powering speech recognition services (e.g., Apple's Siri)
  - recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube)
  - learning to beat the world champion at the game of Go (DeepMind's AlphaGo)

From Biological to Artificial Neurons



Biological Neuron

From Biological to Artificial Neurons

- Biological neurons produce short electrical impulses called action potentials (APs, or just signals).
- These signals travel along the axons and make the synapses release chemical signals called neurotransmitters.
- When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses.
- Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions, with each neuron typically connected to thousands of other neurons.

10.1.2 Logical Computations with Neurons

10.1.3 The Perceptron

Perceptrons

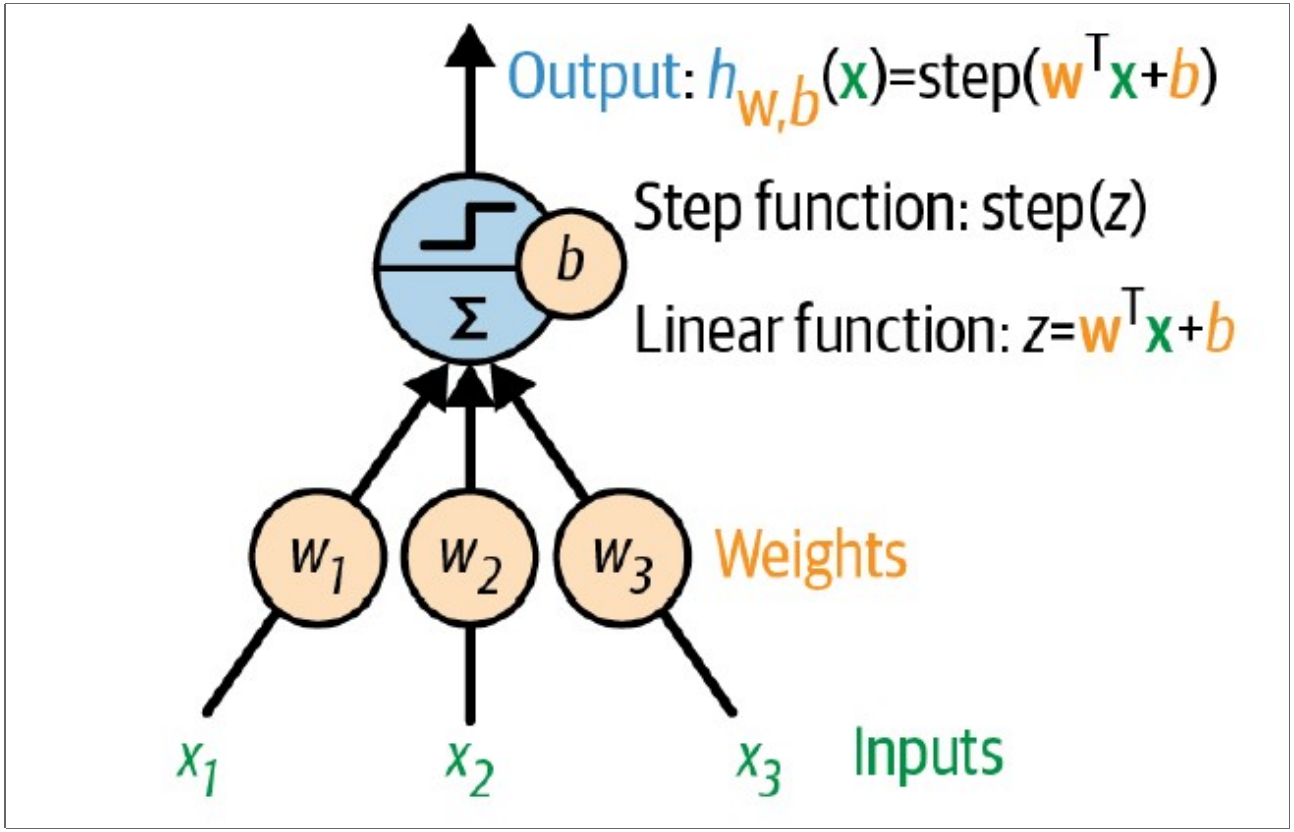
- The **perceptron** is one of the simplest ANN architectures, invented in 1957.
- It is based on an artificial neuron called a *threshold logic unit* (TLU), or sometimes a linear threshold unit (LTU).
- The inputs and output are numbers and each input connection is associated with a weight.
- The TLU computes a weighted sum of its inputs:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{w}^T \mathbf{x} + b$$

- Finally, it applies a step function:

$$\text{output} = \text{step}(z)$$

Threshold Logic Unit



A threshold logic unit

Step Functions

- The step function used in perceptrons is typically the **Heaviside step function**:

$$H(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$$

Single TLU

- A single TLU can be used for simple linear binary classification.
- Just like logistic regression, it computes a linear combination of the inputs
  - if the result exceeds a threshold, it outputs the positive class;
  - otherwise, it outputs the negative class.

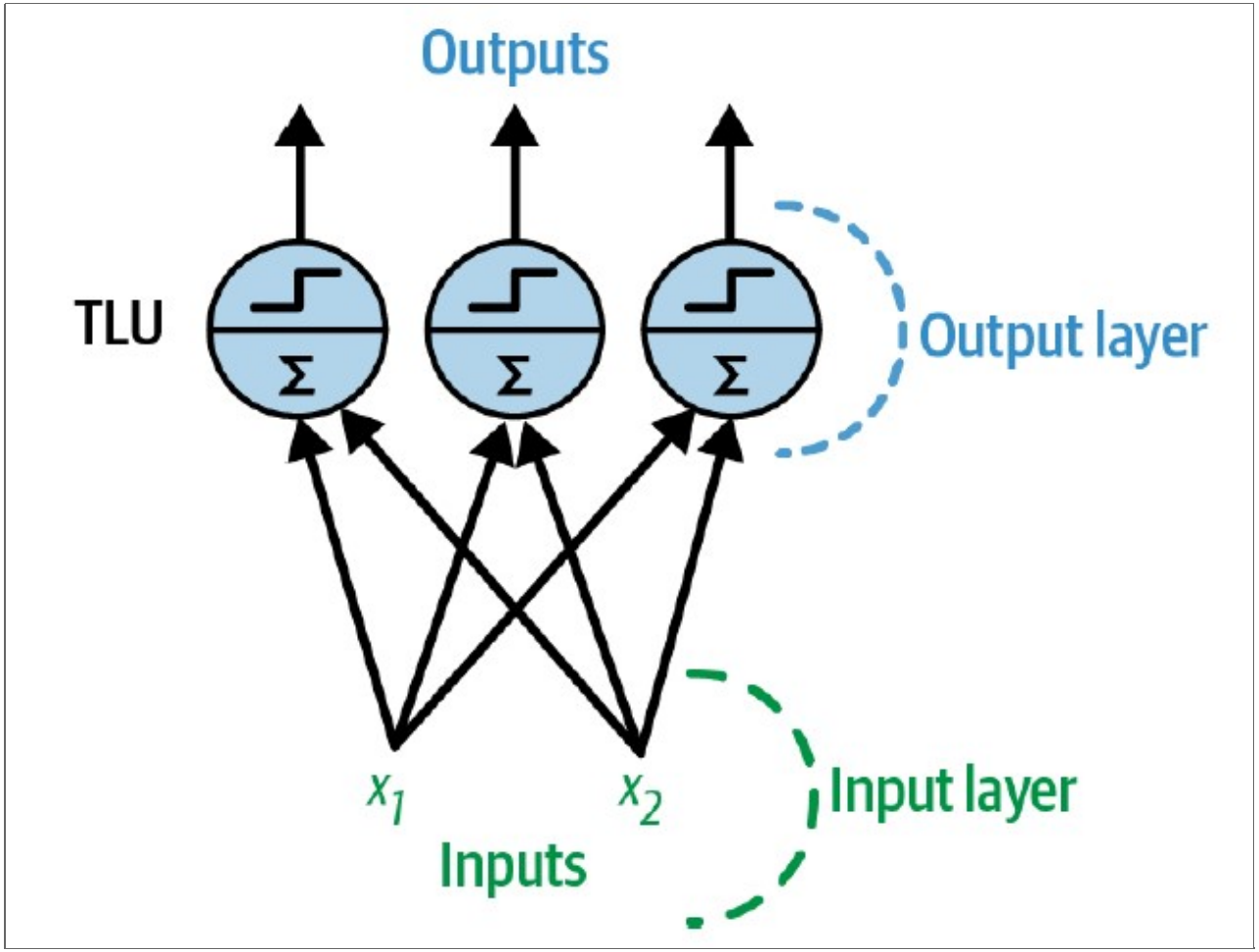
Single TLU

- You could, for example, use a single TLU to classify iris flowers (into two classes) based on petal length and width (also adding an extra bias feature $x_0 = 1$, just like was done in previous chapters).
- Training a TLU in this case means finding the right values for w_0 , w_1 and w_2.
  - w_0 is the bias, w_1 and w_2 are the weights for petal length and petal width respectively.

Perceptron

- A **perceptron** is a single layer of TLUs, with each TLU connected to all the inputs.
- When *all the neurons in a layer are connected to every neuron in the previous layer* (i.e., its input neurons), the layer is called a **fully connected layer**, or a **dense layer**.
- The inputs of the perceptron are fed to special passthrough neurons called input neurons: they output whatever input they are fed.
- All the input neurons form the **input layer**.

Perceptron



Perceptron with two inputs and three outputs

Perceptron Computation

- A single TLU essentially performs a dot product followed by a step function.
- A matrix vector multiplication performs the dot product of each row of the matrix with the vector.
- A matrix multiplication performs the dot product of each row of the first with each column of the second matrix.

Perceptron Computation

The output of a perceptron for many instances can be computed as:

$$h_{\mathbf{W},b}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

- $\mathbf{X}$ = matrix of input features: shape `(num instances, num features)`
- $\mathbf{W}$ = weight matrix containing all the connection weights except for the ones from the bias neuron. Shape `(num features, num outputs)`
- $\mathbf{b}$ = bias vector. One bias term per neuron. Shape `(num outputs,)`

Perceptron Computation

On the previous slide,

- ϕ = activation function. When the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

Broadcasting

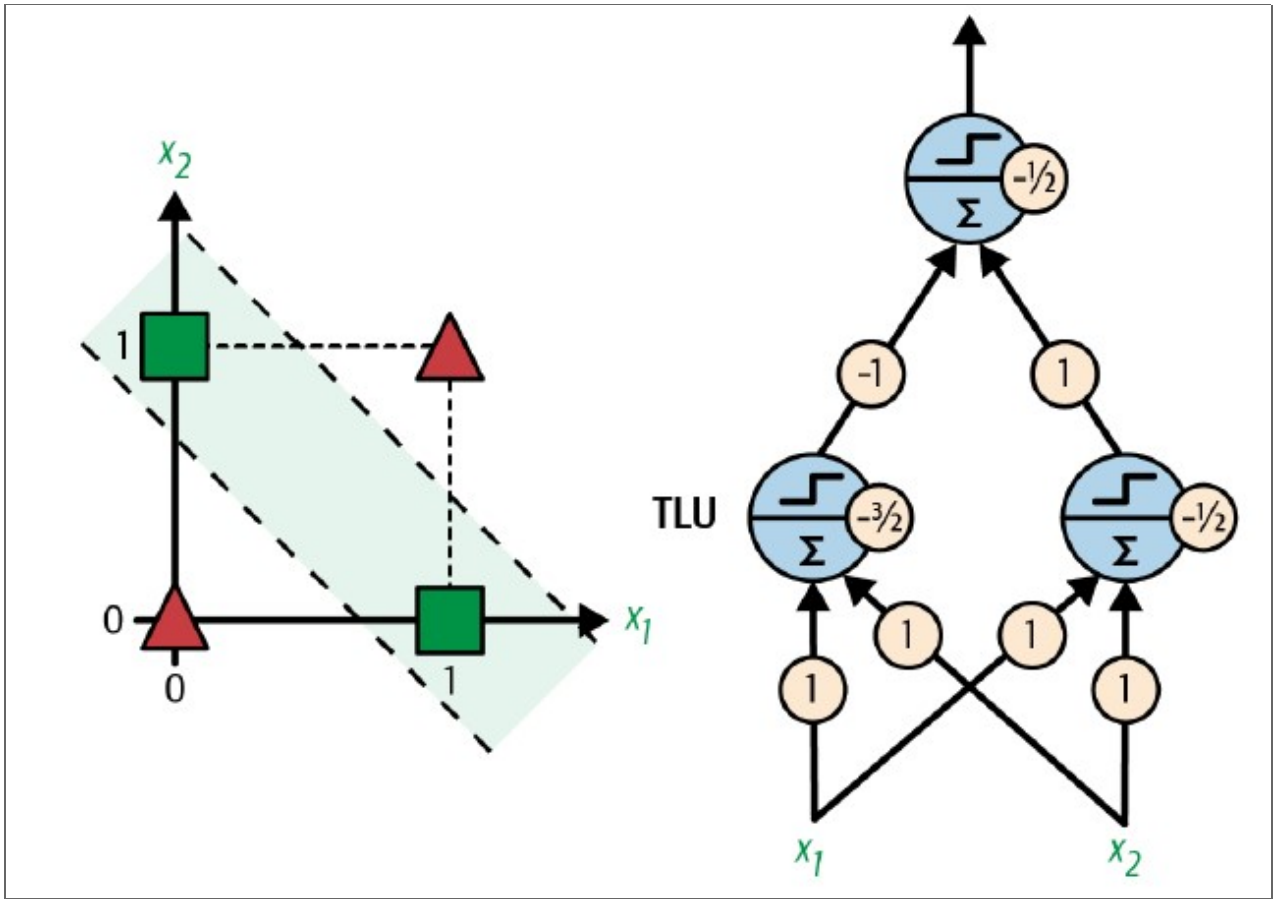
- Note that $\mathbf{XW} + \mathbf{b}$ makes no sense mathematically speaking.
- $\mathbf{XW}$ has shape `(num instances, num outputs)`.
- $\mathbf{b}$ has shape `(num outputs,)`
- **Broadcasting** is applied and this adds $\mathbf{b}$ to every row of $\mathbf{XW}$.
- Also, the activation is applied *to every element of the result*.

Learning the Weights

- There exists a learning algorithm that will converge to a solution if the training instances are linearly separable. This is the **perceptron convergence theorem**.
  - We don’t go into the details of this algorithm.

The XOR Problem

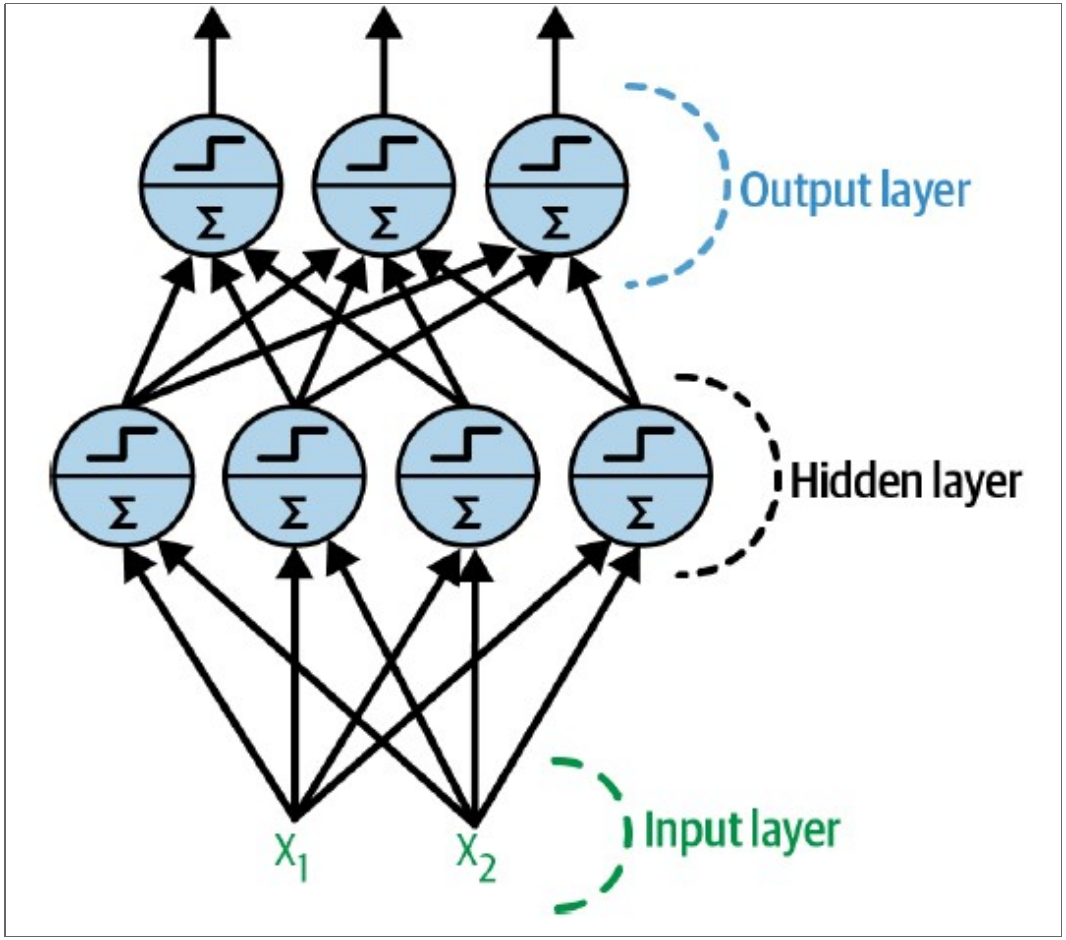
- A perceptron can not solve the XOR problem, but a **multilayer perceptron** can:



MLP for the XOR problem

10.1.4 The Multilayer Perceptron and Backpropagation

The Multilayer Perceptron and Backpropagation



MLP with 2 input neurons, 4 neurons in hidden layer and 3 neurons in output layer

Multilayer Perceptron

- A **multilayer perceptron** simply consists of stacked perceptrons.
- The output of one layer becomes the input to the next layer.
- A **neural network** is called a **deep neural network** when it contains many hidden layers.
  - The definition of “many” changed considerably over the years!
- MPLs are **feedforward** neural networks because the signal only flows in one direction.

Training MLPs with Backpropagation

- The algorithm that is used for training MLP's was published in 1986 and is called **backpropagation**.
- The algorithm goes like this

```
initialize weights randomly
for epoch in num_epochs
  for mini-batch in epoch
    compute results on mini-batch (forward pass)
    compute gradient of loss w.r.t. weights (backward pass)
    update weights in direction opposite gradient
```

Forward Pass

- Each mini-batch is passed to the network's input layer, which sends it to the first hidden layer.
- The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch).
- The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer.
- The forward pass is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.

Backward Pass

- The algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- Then it computes how much each output connection contributed to the error.

Backward Pass

- The algorithm then measures how much of these error contributions came from each connection in the layer below, working backward until the algorithm reaches the input layer.
- This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network.

Backward Pass

*The backward pass implements an algorithm called **reverse-mode autodiff**. You have already seen this in the course Mathematics for Machine Learning*

Backpropagation Summary

Backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases (i.e. the parameters) to reduce the error (gradient descent step).

Activation Functions

- The gradient of the Heaviside and sign function is zero everywhere (except at zero itself where the gradient is infinite)
  - Thus, this gives no useful information on how to update the weights
- The step function was replaced by the **sigmoid** function (or logistic function):

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

- The sigmoid function is differentiable, and has a non-zero derivative everywhere.

Other Activation Functions

- The **hyperbolic tangent** is another activation function

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$$

- Just like the logistic function, this activation function is S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function).
- That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

Other Activation Functions

- **Rectified Linear Unit** or **ReLU** for short:

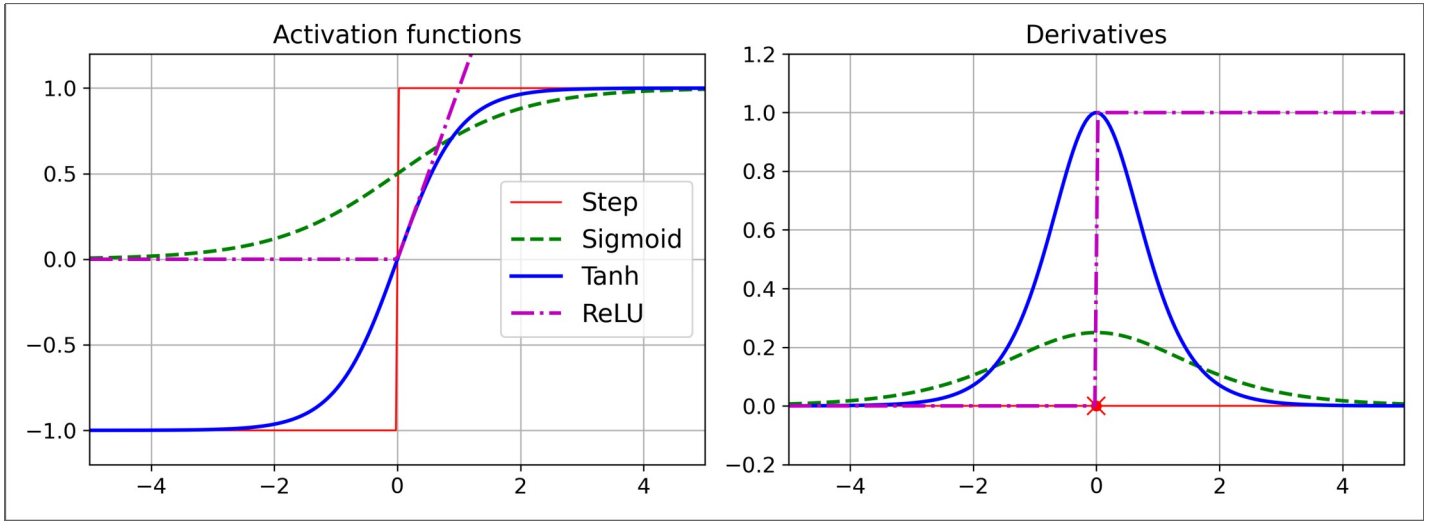
$$\mathrm{ReLU}(z) = \max(0, z)$$

- is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for $z < 0$.

ReLU

- In practice, however, ReLU works very well and has the advantage of being fast to compute, *so it has become the default*.
- Most importantly, the fact that it does not have a maximum output value helps reduce some issues during Gradient Descent.

Activation functions



Activation functions and their derivatives

Why are Activation Functions Needed?

- Composing several linear transformations gives a linear transformation.
- For example, if $f(x) = 2x + 3$ and $g(x) = 5x–1$, then composing these two linear functions gives you another linear function:

$$f(g(x)) = 2(5x–1) + 3 = 10x + 1.$$

Why are Activation Functions Needed?

- So if you don’t have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can’t solve very complex problems with that.
- Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

10.1.5 Regression MLPs

Regression MLPs

- If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value.
- For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension.
  - For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons.

Regression MLPs

- In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values.

Regression MLPs

- If you want to guarantee that the output will always be positive, then you can use the ReLU activation function in the output layer.
- Alternatively, you can use the **softplus activation function**, which is a smooth variant of ReLU:

$$\text{softplus}(z) = \log(1 + \exp(z)).$$

- It is close to 0 when z is negative, and close to z when z is positive.

Regression MLPs

- Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and then scale the labels.

Regression MLPs

- The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead.
- The **Huber loss** is a combination of MSE and MAE.
  - It is quadratic for small errors, but linear for larger errors.

Regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	mostly ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or Huber (see chapter 11)

10.1.6 Classification MLPs

Classification MLPs

- For a binary classification problem, you just need a single output neuron using the logistic activation function
  - The output will be a number between 0 and 1, which is interpreted as the estimated probability of the positive class.
  - This is the same interpretation as for logistic regression.
- The estimated probability of the negative class is equal to one minus that number.

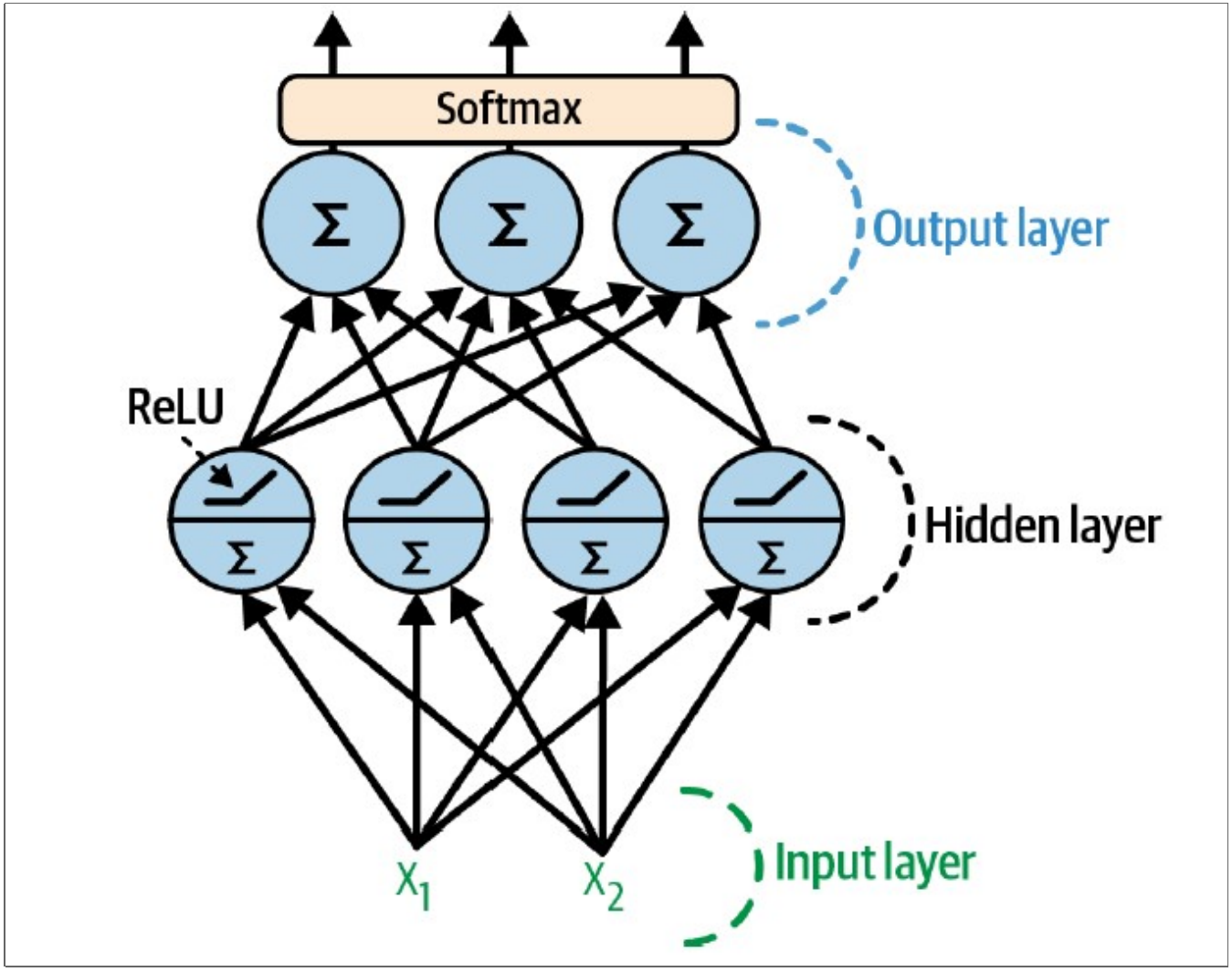
Classification MLPs

- MLPs can also easily handle **multilabel binary classification** tasks.
  - For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent.

Classification MLPs

- If each instance belongs to a *single class*, out of three or more possible classes (e.g., classes 0 through 9 for digit classification), then you need one output neuron per class, and you should use the softmax activation function for the output layer.

Classification MLPs



MLP for classification into three disjunct classes

Softmax Activation

- Softmax is typically used for **multiclass classification**
- The softmax is calculated as follows:
  - $$\mathrm{softmax}(z_1, z_2, \ldots, z_k) =$$
    - First, compute $\exp z = e^z$ for each z. You get $(e^{z_1}, e^{z_2}, \ldots, e^{z_k})$.
    - All the numbers are now positive.
    - Divide each number by the sum of all numbers $\implies$ the values will sum to one. Let $Z = e^{z_1} + e^{z_2} + \cdots + e^{z_k}$, then

$$\mathrm{softmax}(z_1, z_2, \ldots, z_k) = \left(\frac{e^{z_1}}{Z}, \frac{e^{z_2}}{Z}, \ldots, \frac{e^{z_k}}{Z}\right)$$

MLP Classification Architecture

Hyperparameter	Binary Classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output activation	Logistic	Logistic	Softmax
Loss function	(Binary) Cross entropy	(Binary) Cross entropy	(Categorical) Cross entropy

Loss Functions

On the next slides we give the formulas for the different loss functions, so that it is easy to reference them later.
Note: Most of these formulas are mentioned in Chapter 4 of the book.

MSE Loss Function

We recap the loss function for regression problems. The MSE should be familiar to you from previous courses.

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2.$$

Here, m is the number of examples, $\hat{y}^{(i)}$ is the predicted value for example i and $y^{(i)}$ is the true value for example i.

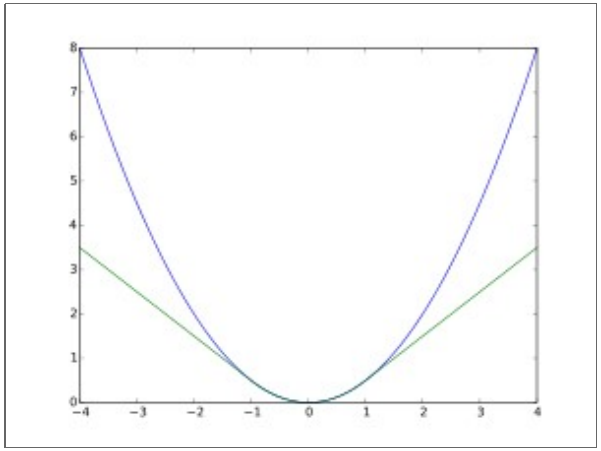
Huber Loss Function

The Huber loss for a single example:

$$H(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq 1 \\ |y - \hat{y}| - \frac{1}{2} & \text{otherwise} \end{cases}$$

In this formula, y is the true label, while $\hat{y}$ is the predicted value.
Note: in general the Huber loss also contains a parameter δ that determines the threshold at which the loss changes from quadratic to linear. In the above formula, $\delta = 1$.

Huber Loss Function



Huber loss (green) vs. quadratic loss (blue).
Source: Wikipedia

Binary Cross Entropy

The binary cross entropy loss for a single example:

$$\mathrm{BCE}(y, \hat{y}) = -\big(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\big)$$

Here y is the true label, which is either 0 or 1, while $\hat{y}$ is the predicted probability that the label is 1.
Note: the binary cross entropy is the loss function used in logistic regression, so you should already be familiar with it.

Categorical Cross Entropy

The categorical cross entropy loss generalizes the binary cross entropy to the case where there are more than K classes (with $K > 2$). For a single example, it is defined as:

$$\text{CCE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{k=1}^{K} \mathbf{y}_k \log(\hat{\mathbf{y}}_k)$$

Here, $\mathbf{y}$ is a one-hot encoded vector of length K and $\hat{\mathbf{y}}$ is a vector of length K containing the predicted probabilities for each class.

10.2 Implementing MLPs with Keras

10.2.1 Building an Image Classifier Using the Sequential API

Keras

- Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate, and execute all sorts of neural networks.
- Its documentation (or specification) is available at **keras.io**.
- It is based on a Tensorflow backend.
- The most popular Deep Learning library, after Keras and TensorFlow, is Facebook's PyTorch library. The good news is that its API is quite similar to Keras's (in part because both APIs were inspired by Scikit-Learn and Chainer), so once you know Keras, it is not difficult to switch to PyTorch.

Building an Image Classifier

- First, we need to load a dataset.
- In this chapter we will tackle *Fashion MNIST*, which is a drop-in replacement of MNIST (introduced in Chapter 3). It has the exact same format as MNIST (70,000 grayscale images of 28 × 28 pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem turns out to be significantly more challenging than MNIST.
- For example, a simple linear model reaches about 92% accuracy on MNIST, but only about 83% on Fashion MNIST.

Load the Data

```
import tensorflow as tf
import matplotlib.pyplot as plt

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:]

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

- Tensorflow was imported in the idiomatic way.
- A training and validation set was created.
- We inspected the shape and data type.

Rescale and Inspect Data

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

y_train # array([9, 0, 0, ..., 9, 0, 2], dtype=uint8)

# Map numbers to meaningful class names
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
  for col in range(n_cols):
    index = n_cols * row + col
    plt.subplot(n_rows, n_cols, index + 1)
    plt.imshow(X_train[index], cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_train[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```

Fashion MNIST Samples

Fashion MNIST samples

The Sequential API

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(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"))
```

- The **Sequential** model is a single stack of layers, connected sequentially.
- Specify the `Input` layer. The `input_shape` does *not* include the batch dimension.
- `Flatten` turns each input image into a 1D array. This layer has no parameters.

The Sequential API

```
# Repeated from previous slide
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(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"))
```

- Next, a `Dense` (or fully connected layer) with 300 neurons is added with the ReLU activation function.
- A second fully connected layer is added with 100 neurons.
- The output layer has 10 neurons, one for each class. The softmax activation function is used because the classes are exclusive.

The Sequential API

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

- This code is equivalent to the previous code, but passes a *list of layers* to the constructor of `Sequential`.
- Also, the explicit `Input` layer is not used but instead the `input_shape` parameter is used on the first layer.

Model Summary

```
model.summary()
```

could produce something like:

```
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
_________________________________________________________________
```

Number of Parameters

- The first fully connected layer has

$$784 \times 300 + 300 = 785 \times 300 = 235\,500$$

  parameters! This is a lot.
- Hence, the model will be very flexible but may risk overfitting when there is not a lot of training data.

Model Layers

```
model.layers
```

may give something like

```
[<keras.layers.core.flatten.Flatten at 0x137b8f81190>,
 <keras.layers.core.dense.Dense at 0x137b8f813a0>,
 <keras.layers.core.dense.Dense at 0x137b8dbac10>,
 <keras.layers.core.dense.Dense at 0x137b8dc13a0>]
```

You can access the parameters of a layer by using `get_weights`. In this case connection weights and bias terms are returned:

```
weights, biases = model.layers[1].get_weights()
weights.shape # (784, 300)
biases.shape # (300,)
```

Parameter Initialization

- The connection weights are initialized randomly (to break symmetry).
- The bias terms are all initialized to zero.

Compiling the Model

- After a model is created, you must call its `compile()` method to specify
  - the loss function
  - and the optimizer to use.
- Optionally, you can specify a list of extra metrics to compute during training and evaluation.

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

Compiling the Model

```
# Repeated from previous slide
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

- We use `sparse_categorical_crossentropy` because we have *sparse labels* (i.e., for each instance, there is just a target class index, from 0 to 9 in this case), and the classes are exclusive.
- When a target probability is given for each class for each instance, use `categorical_crossentropy`. In this case `[0.,0., 0., 1., 0., 0., 0., 0., 0., 0.]` would represent the class `3`.

Compiling the Model

```
# Repeated from previous slide
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

- Using `sgd` means that we will train the model using **s**tochastic **g**radient **d**escent.
  - More efficient optimizers are available.
- `metrics=['accuracy']` means that we want to measure the model's accuracy during training and evaluation.

Compiling the Model

The previous code is equivalent to

```
model.compile(loss=tf.keras.losses.sparse_categorical_crossentropy,
              optimizer=tf.keras.optimizers.SGD(),
              metrics=[tf.keras.metrics.sparse_categorical_accuracy])
```

- With the second version, you can change the learning rate to use:

```
optimizer=tf.keras.optimizers.SGD(learning_rate=0.05)
```

would use a learning rate of 0.05 instead of the default 0.01.

Converting to One-Hot Labels

```
tf.keras.utils.to_categorical([0, 5, 1, 0], num_classes=10)
```

results in

```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

- Note: this is an alternative to using `pd.get_dummies()` in pandas.

Converting from One-Hot Labels

```
np.argmax(
    [[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
     [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
     [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
     [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.]],
    axis=1
)
```

yields

```
array([0, 5, 1, 0])
```

- This may be useful to convert predictions to class indices.

Training and Evaluating the Model

```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))
```

- Calling `fit` trains the model. Pass
  - `X_train` and `y_train` as the training data and labels;
  - the number of epochs to train for as `epochs`;
  - an optional validation set as `validation_data`. This will be used to measure the loss and any metrics at the end of each epoch. This can be used to detect overfitting.

Training and Evaluating the Model

The previous command may produce something like this (truncated):

```
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
```

Note: the model is overfitting slightly.

Training and Evaluating the Model

The `fit()` method returns a `History` object containing

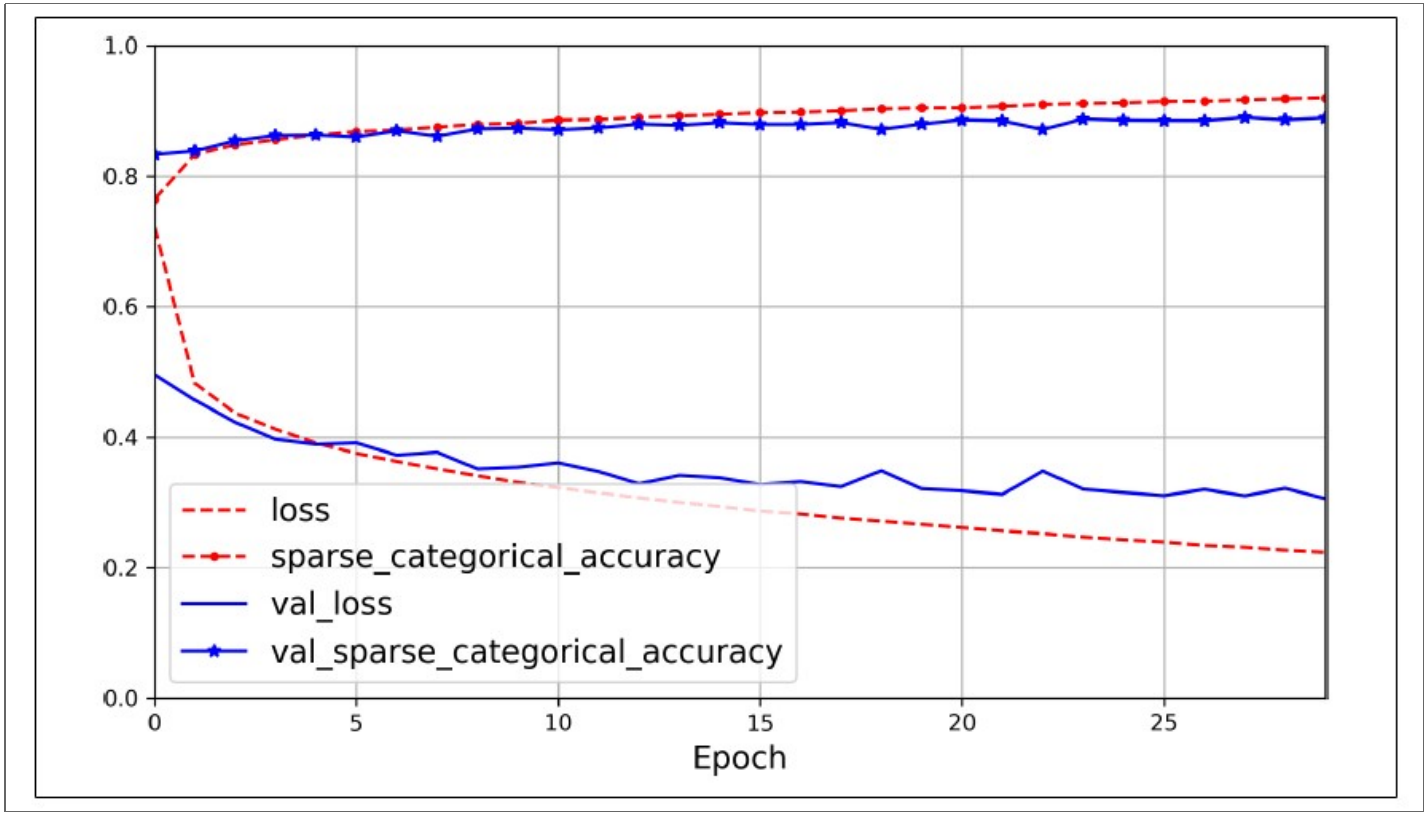
- the training parameters (`history.params`),
- the list of epochs it went through (`history.epoch`)
- a dictionary (`history.history`) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any).
    - Use this dictionary to create a pandas DataFrame and call its `plot` method. You will get the **learning curves**.

Learning Curves

The following code will plot the learning curves for the model we just trained. We will see training and validation loss and accuracy as a function of the epochs.

```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid))

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()
```

Learning Curves



Learning curves for fashion MNIST

Evaluation on the Test Set

The `evaluate()` method will measure the loss and any metrics on the test set.

```
model.evaluate(X_test, y_test)
```

could give something like

```
[0.32431697845458984, 0.8863999843597412]
```

- The first value is the loss (cross-entropy) on the test set.
- The second value is the accuracy on the test set.

Evaluation on the Test Set

You should only evaluate on the test set after you have selected your final model (using the validation set).

Making Predictions

The `predict()` method will return class probabilities for each instance given to it.

```
X_new = X_test[:3]  # pretend these are new instances
y_proba = model.predict(X_new)
y_proba.round(2)
```

gives

```
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)
```

Making Predictions

Extracting the class with the highest probability is easy using `np.argmax()`:

```
y_pred = y_proba.argmax(axis=-1)
y_pred
```

In this case we get:

```
array([9, 2, 1])
```

The actual class names are:

```
np.array(class_names)[y_pred]
```

which gives

```
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

10.2.2 Building a Regression MLP Using the Sequential API

Building a Regression MLP

- We build a regression MLP to predict housing prices using the California housing dataset.
  - This dataset only has numerical features.
  - This dataset has no missing values.
  - We can use the `fetch_california_housing()` function from `sklearn` to load the dataset.

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)
```

Building a Regression MLP

- We will use an MLP containing 3 hidden layers composed of 50 neurons each, using the ReLU activation function.
- Since we are predicting a single value for each instance, we only need a single output neuron.
- The output layer will not utilize any activation function.
- We use the `mean_squared_error` loss function, which is well suited for regression tasks.
- We will use the `Adam` optimizer, which is a variant of Stochastic Gradient Descent (see Chapter 11).

Building a Regression MLP

- We don't need a `Flatten` layer, but will use a `Normalization` layer to ensure the features are scaled properly.
  - The `Normalization` layer does the same as Scikit-Learn's `StandardScaler` but it must be adapted to the training data *before* `fit` is called.

```
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
norm_layer.adapt(X_train)
```

The `adapt()` method computes the mean and variance of each feature and stores them as the layer's variables.

Building a Regression MLP

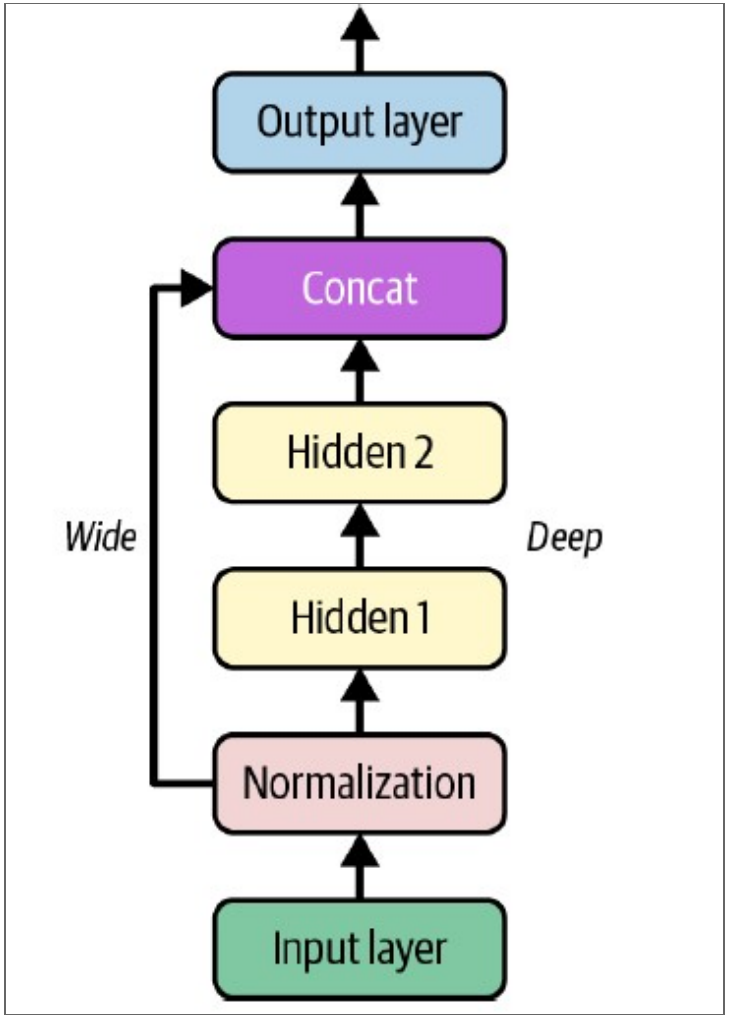
```
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
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)
```

10.2.3 Building Complex Models Using the Functional API

Wide and Deep Neural Network

- The `Functional` API can be used to build much more complex architectures, that are **non-sequential**.
- An example of such an architecture is the *Wide & Deep* neural network introduced in a 2016 paper by Heng-Tze Cheng et al.

Wide and Deep Neural Network



Architecture of Wide & Deep neural network

Wide and Deep Neural Network

```
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)
```

- In this code we create 5 layers.
  - a `Normalization` layer
  - two `Dense` layers with 30 neurons each and both using the ReLU activation function
  - a `Concatenate` layer
  - a `Dense` output layer with a single neuron

Wide and Deep Neural Network

```
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)
```

- Next, we specify the connections between the layers by calling them like functions.
  - Note: no actual data is being processed yet.
  - Everything is symbolic in order to contruct the model.
- The `Input` layer specifies the shape and optionally the data type of the inputs (with 32-bit floats as the default)
- `concat_layer` is passed a list of inputs to concatenate.

Wide and Deep Neural Network

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

- Finally, we create a `Model` and specify which inputs and outputs to use.
- Now, we can compile the model, adapt the normalization layer and fit the model as usual.
- We can also ask for a model summary.

Wide and Deep Neural Network

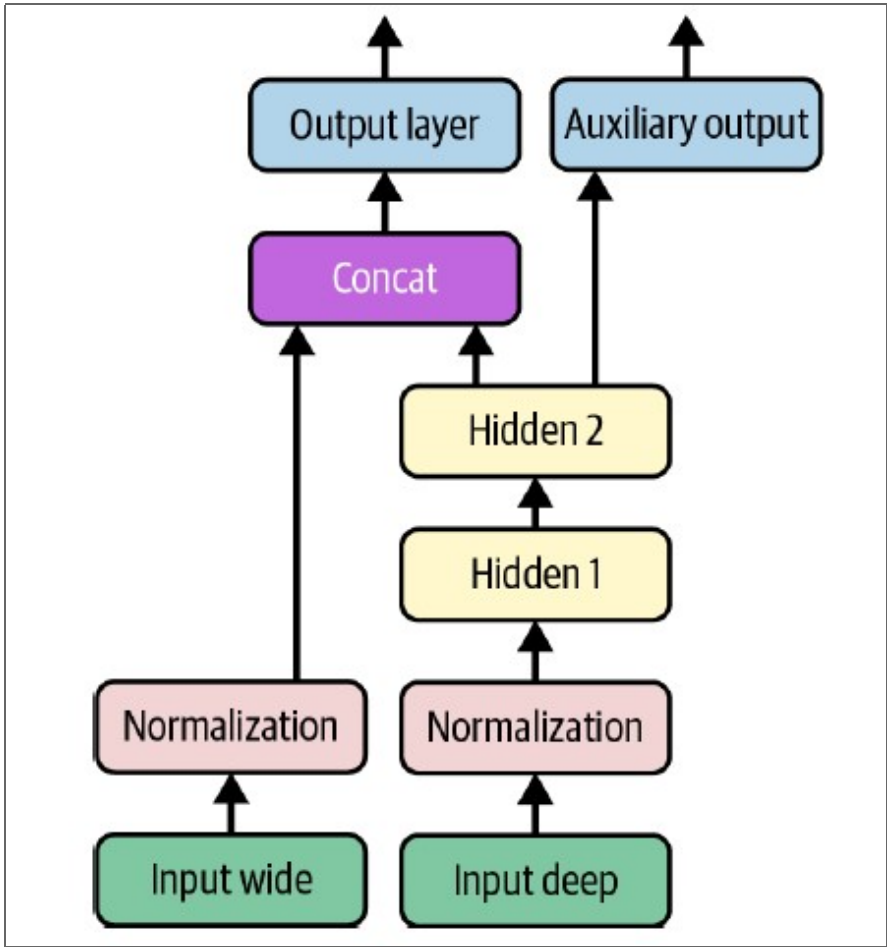
```
model.summary()
```

which gives

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 8)]          0           []

 normalization (Normalization)  (None, 8)            17          ['input_1[0][0]']

 dense (Dense)                  (None, 30)           270         ['normalization[0][0]']

 dense_1 (Dense)                (None, 30)           930         ['dense[0][0]']

 concatenate (Concatenate)      (None, 38)           0           ['input_1[0][0]',
                                                                  'dense_1[0][0]']

 dense_2 (Dense)                (None, 1)            39          ['concatenate[0][0]']

==================================================================================================
Total params: 1,256
Trainable params: 1,239
Non-trainable params: 17
__________________________________________________________________________________________________
```

Multiple Inputs and Outputs

Suppose we want to create this architecture:



Multiple inputs and outputs

Multiple Inputs and Outputs

- The ‘wide’ input will take the first 5 features of each instance (0 to 4).
- The ‘deep’ input will take the last 6 features (2 to 7).
- We will add an auxiliary output on top of the hidden layer for regularization purposes.
- In practice you may also want multiple outputs because
  - the tasks demands it, e.g. classification and localization in an image
  - you have multiple independent tasks on the same data

Multiple Inputs and Outputs

```
input_wide = tf.keras.layers.Input(shape=[5])  # features 0 to 4
input_deep = tf.keras.layers.Input(shape=[6])  # features 2 to 7
norm_layer_wide = tf.keras.layers.Normalization()
norm_layer_deep = tf.keras.layers.Normalization()
norm_wide = norm_layer_wide(input_wide)
norm_deep = norm_layer_deep(input_deep)
hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)
concat = tf.keras.layers.concatenate([norm_wide, hidden2])
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])
```

Multiple Inputs and Outputs

```
# Part of previous code
hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm_deep)
...
concat = tf.keras.layers.concatenate([norm_wide, hidden2])
```

- Note how the `Dense` layer is created and called in one go.
  - This makes the code more concise.
  - For the `Normalization` layer we did not do this because we need a reference to the layer to be able to call `adapt()` later on.
- `tf.keras.layers.concatenate` creates a `Concatenate` layer and calls it with the given inputs.

Multiple Inputs and Outputs

```
# Part of previous code
model = tf.keras.Model(inputs=[input_wide, input_deep],
                       outputs=[output, aux_output])
```

- We pass the inputs and the outputs of the model as lists.
- The code below splits the training set into two parts:
  - one for the wide path
  - one for the deep path

```
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:]
# pretend new instances
X_new_wide, X_new_deep = X_test_wide[:3], X_test_deep[:3]
```

Multiple Inputs and Outputs

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

- We compile the model with two losses, one for each output.
  - In this example the two losses are the same, but in general they could be different.
- `loss_weights` specifies the relative importance of each loss.
  - In this example the main output is more important than the auxiliary output.

Multiple Inputs and Outputs

Instead of passing a tuple `loss=("mse", "mse")`, you can pass a dictionary `loss={"output": "mse", "aux_output": "mse"}`, assuming you created the output layers with `name="output"` and `name="aux_output"`. Just like for the inputs, this clarifies the code and avoids errors when there are several outputs. You can also pass a dictionary for loss_weights.

Multiple Inputs and Outputs

The following code trains the model for 20 epochs. Note how we pass the two training sets, one for each input:

```
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))
)
```

Multiple Inputs and Outputs

```
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
```

- `evaluate` returns the weighted sum of the losses, as well as all the individual losses and metrics.
- With `return_dict=True` as an argument, `evaluate` returns a dictionary

Multiple Inputs and Outputs

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

- `predict` returns a tuple of predictions, one per output.
- You can turn this tuple into a dictionary with the following code:

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

Functional API Summary

The functional API can be used to build all sorts of architectures.

10.2.4 Using the Subclassing API to Build Dynamic Models

Subclassing API

- With the sequential and functional APIs, the model's architecture is static.
  - This has many advantages: easy to save; structure can be displayed; shapes can be inferred, ...
- But, models may involve loops, varying shapes, conditional branching, and other dynamic behaviors.
- This is possible with the **Subclassing API**.
  - Subclass the `tf.keras.Model` class.
  - The `__init__()` method creates the layers.
  - The model's architecture is defined in the `call()` method.

Wide & Deep Model with Subclassing API

```
class WideAndDeepModel(tf.keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs)  # needed to support naming the model
        self.norm_layer_wide = tf.keras.layers.Normalization()
        self.norm_layer_deep = tf.keras.layers.Normalization()
        self.hidden1 = tf.keras.layers.Dense(units,activation=activation)
        self.hidden2 = tf.keras.layers.Dense(units,activation=activation)
        self.main_output = tf.keras.layers.Dense(1)
        self.aux_output = tf.keras.layers.Dense(1)

    def call(self, inputs):
        input_wide, input_deep = inputs
        norm_wide = self.norm_layer_wide(input_wide)
        norm_deep = self.norm_layer_deep(input_deep)
        hidden1 = self.hidden1(norm_deep)
        hidden2 = self.hidden2(hidden1)
        concat = tf.keras.layers.concatenate([norm_wide, hidden2])
        output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return output, aux_output
```

Flexibility but at a Cost

- Keras cannot inspect the model's architecture.
- The model cannot be cloned using `tf.keras.models.clone_model()`.
- The `summary()` method only displays a list of layers, without any information on how they are connected to each other.
- Keras cannot check shapes and types ahead of time.

> *With the subclassing API it is easy to make mistakes and hard to debug them. Only use it when you really need extra flexibility.*

10.2.5 Saving and Restoring a Model

Saving a Model

Saving a trained Keras model is very easy:

```
model.save("my_keras_model", save_format="tf")
```

- This saves the model in *SavedModel* format. This is a directory containing multiple files and subdirectories.
  - `saved_model.pb` contains the model's architecture
  - The `variables/` subdirectory contains the model's parameter values (including weights, biases, normalization statistics, the optimizer's parameters, etc.)

Saving a Model

Let's look at the contents of the directory:

```
for path in sorted(Path("my_keras_model").glob("**/*")):
    print(path)
```

gives

```
my_keras_model/assets
my_keras_model/keras_metadata.pb
my_keras_model/saved_model.pb
my_keras_model/variables
my_keras_model/variables/variables.data-00000-of-00001
my_keras_model/variables/variables.index
```

Restoring a Model

Restoring a model is just as easy and could be done in a new Python session, where you don’t have access to the model’s source code:

```
model = tf.keras.models.load_model("my_keras_model")
```

10.2.6 Using Callbacks

Using Callbacks

- The `fit()` method accepts a `callbacks` argument that lets you specify a list of objects that Keras will call
  - at the start and end of training,
  - at the start and end of each epoch,
  - and even before and after processing each batch.
- `ModelCheckpoint` saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

```
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_checkpoints",
    save_best_only=True)
history = model.fit([...], callbacks=[checkpoint_cb])
```

Early Stopping

`EarlyStopping` interrupts training when it measures no progress on the validation set for a number of epochs (defined by the `patience` argument):

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
    restore_best_weights=True)
history = model.fit([...], callbacks=[checkpoint_cb, early_stopping_cb])
```

- When using `EarlyStopping`, the number of epochs can be set to a large value.
  - training will stop automatically when there is no more progress.
- `EarlyStopping` stores the weights of the best model in RAM and restores them at the end of training.

10.2.7 Using TensorBoard for Visualization

What is TensorBoard?

- TensorBoard is an interactive visualisation tool to
  - view learning curves during training,
  - compare curves and metrics between multiple runs,
  - visualize the computation graph,
  - analyze training statistics,
  - ...

Using TensorBoard

- To use TensorBoard, you must change your program so that it outputs the data you want to visualize to special binary log files called *event files*.
  - Each binary data record is called a *summary*.
- The TensorBoard server monitors the log directory to pick up any changes.

> *In general you point the TensorBoard server to a root directory and configure your program so that it writes to a different subdirectory every time it runs.*

The Log Directory Name

```
from pathlib import Path
from time import strftime

def get_run_logdir(root_logdir="my_logs"):
    return Path(root_logdir) / strftime("run_%Y_%m_%d_%H_%M_%S")

run_logdir = get_run_logdir()
```

- Every time you call `get_run_logdir()`, you get the name of a different subdirectory, based on the current date and time.
  - Note: this doesn't actually create the directory.

Using the TensorBoard Callback

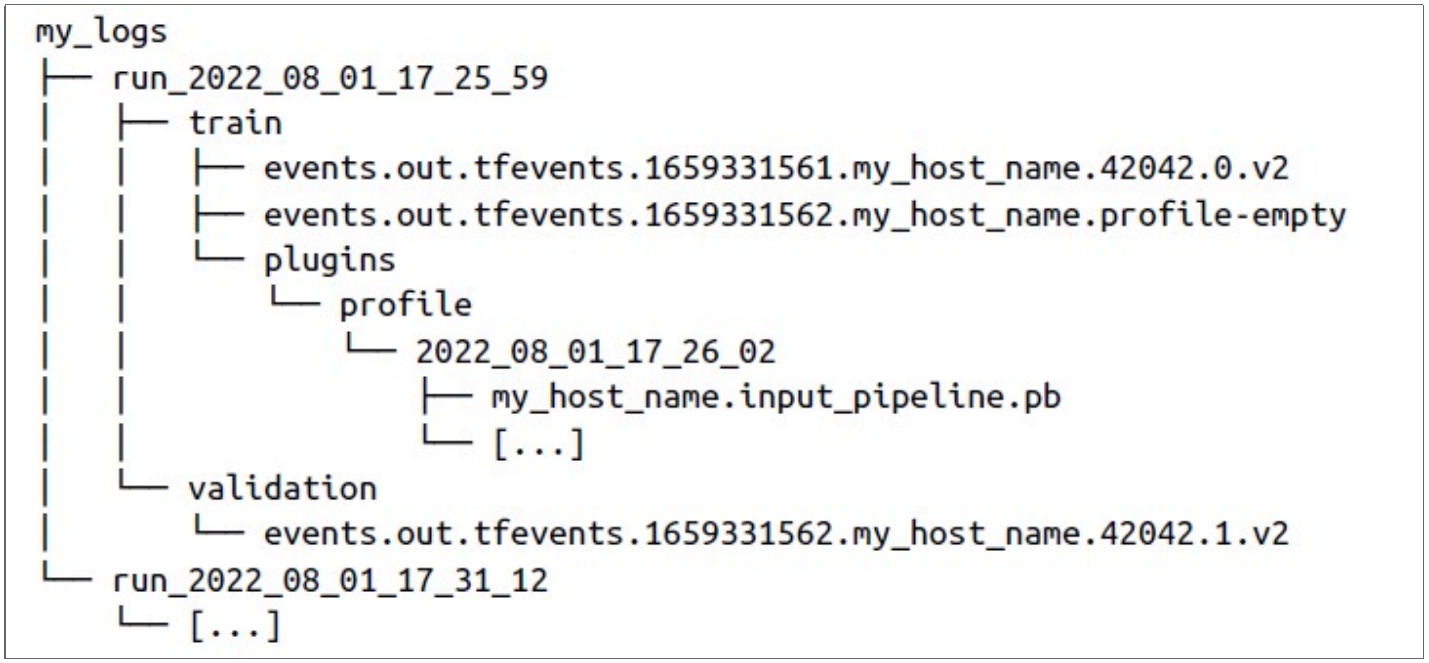
- Next, create a `tf.keras.callbacks.TensorBoard` callback, pointing it to the log directory.
- Pass this callback to the `fit()` method in the `callbacks` argument.

```
run_logdir = get_run_logdir()
tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir,
                                                profile_batch=(100, 200))
history = model.fit([...], callbacks=[tensorboard_cb])
```

- Note: we will profile the network between batches 100 and 200 during the first epoch.
  - It takes a few epochs for the neural network to “warm up”.
  - Profiling uses resources, so it’s best not to do it for every batch.

Directory Structure

- If you perform two training runs, you will get two subdirectories in the root log directory.



TensorBoard log directory structure

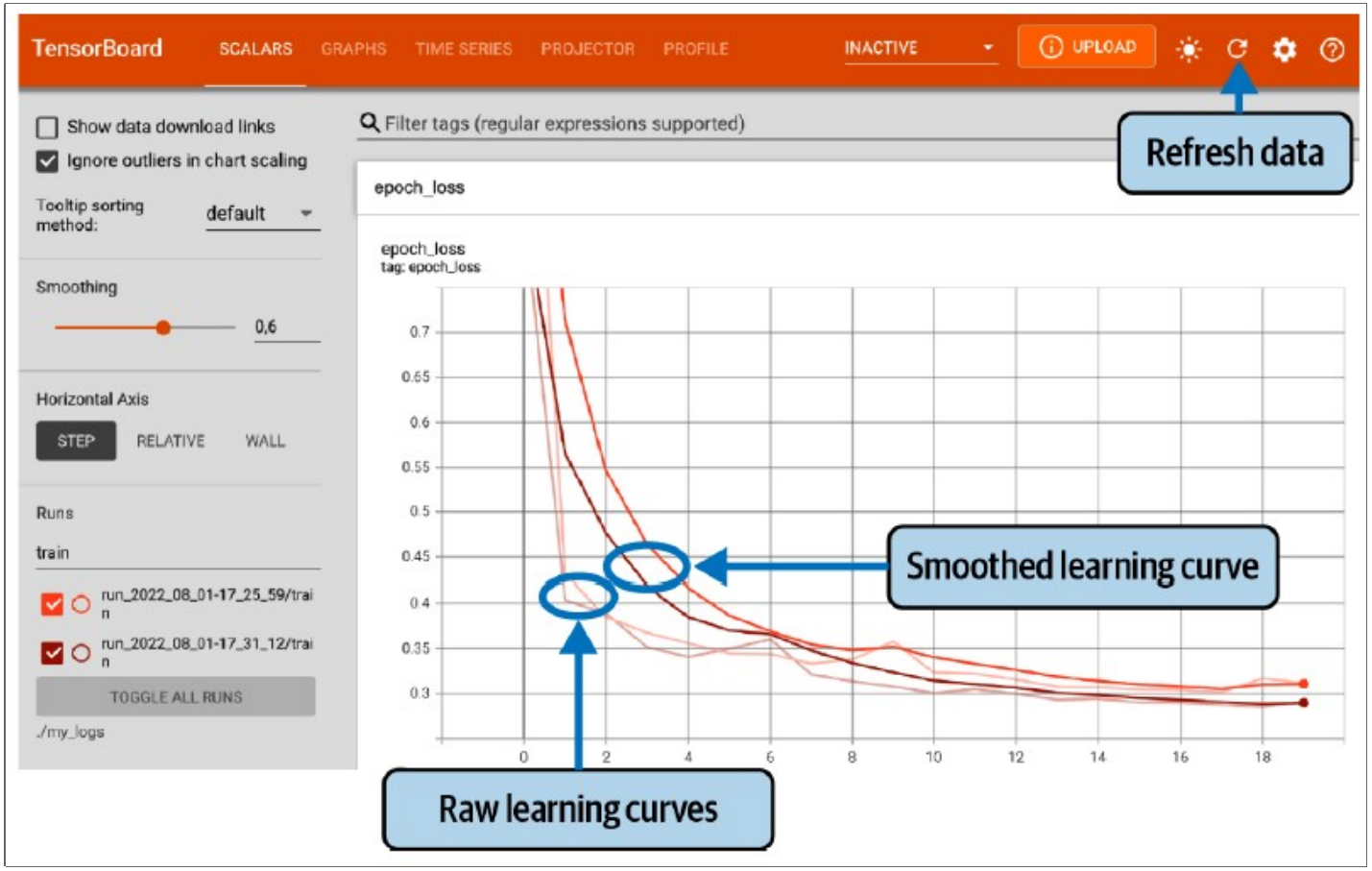
Starting the TensorBoard Server

- The TensorBoard server can be started as follows:

```
%load_ext tensorboard
%tensorboard --logdir=./my_logs
```

- The TensorBoard server will be available at http://localhost:6006

The TensorBoard Interface



The TensorBoard interface

10.3 Fine-Tuning Neural Network Hyperparameters

Hyperparameter Tuning

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak.

- network architecture
- number of layers
- number of neurons per layer
- type of activation function to use in each layer
- the weight initialization logic
- ...

Keras Tuner

10.3.1 Number of Hidden Layers

Number of Hidden Layers

- Single hidden layer can model complex functions with enough neurons.
- Deep networks are more *parameter efficient* for complex problems.
  - Deep networks use exponentially fewer neurons than shallow networks.
  - Deep networks achieve better performance with the same amount of training data.

Hierarchical Architecture

- Real-world data often has a **hierarchical structure**.
- Deep neural networks automatically take advantage of this structure
  - Lower hidden layers model low-level structures
  - Intermediate hidden layers combine low-level structures to model intermediate-level structures
  - Highest hidden layers and output layer combine intermediate structures to model high-level structures
- Very often you can use **transfer learning**, i.e. reuse the lower layers of a pretrained network.

Advice

- One or two hidden layers work well for many problems.
  - E.g. on the MNIST dataset you can achieve high accuracy (>97%) with one or two hidden layers.
- More complex problems require more hidden layers.
  - Ramp up the number of hidden layers until you start overfitting.
- Large image classification or speech recognition may need dozens of layers.
  - They also need a huge amount of training data.
- Pretrained networks can be reused for similar tasks, improving training speed and data requirement.

10.3.2 Number of Neurons per Hidden Layer

Number of Neurons per Layer

- The number of neurons in the input and output layers is determined by the type of input and output your task requires.
- In general it’s fine to use the same number of neurons in each hidden layer.
  - However, it can sometimes help to make the first hidden layer bigger than the others.
- Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting.

Number of Neurons per Layer

It's often simpler and more efficient to pick a model with more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting: the ***stretch pants*** *approach.*

10.3.3 Learning Rate, Batch Size, and Other Hyperparameters

Learning Rate

- most important hyperparameter
- train the model for a few hundred iterations, starting with a very low learning rate (e.g., 10^{-5}) and gradually increase it up to a very large value (e.g., 10).
- this is done by multiplying the learning rate by a constant factor at each iteration.
- plot the loss as a function of the learning rate (using a log scale for the learning rate), and find the learning rate value where the loss starts shooting back up.
  - The optimal learning rate will be a bit lower than the point at which the loss starts to climb.

Optimizer

- Choosing a better optimizer than plain old Mini-batch Gradient Descent (and tuning its hyperparameters) is also quite important.
- We will see several advanced optimizers in Chapter 11.

Batch Size

- The batch size can also have a significant impact on the model's performance and the training time.
- The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently, so the training algorithm will see more instances per second.
  - in practice, large batch sizes often lead to training instabilities, especially at the beginning of training.