AI in Biomedical Data

Dr. M.B. Khodabakhshi
Amir Hossein Fouladi
Alireza Javadi

github.com/mbkhodabakhshi/AI_in_BiomedicalData

ستاد توسعه فناوری های
هوش مصنوعی و رباتیک

یادگیری ماشین در زیست پزشکی

Chapter 10. Introduction to Artificial Neural Networks with Keras

دکتر محمدباقر خدابخشی
mb.khodabakhshi@gmail.com

Presenter: Dr.Khodabakhshi

- The first part of this chapter introduces artificial neural networks, starting with a quick tour of the very first ANN architectures and leading up to *Multilayer Perceptrons* (MLPs), which are heavily used today (other architectures will be explored in the next chapters).
- In the second part, we will look at how to implement neural networks using the popular Keras API. This is a beautifully designed and simple high level API for building, training, evaluating, and running neural networks

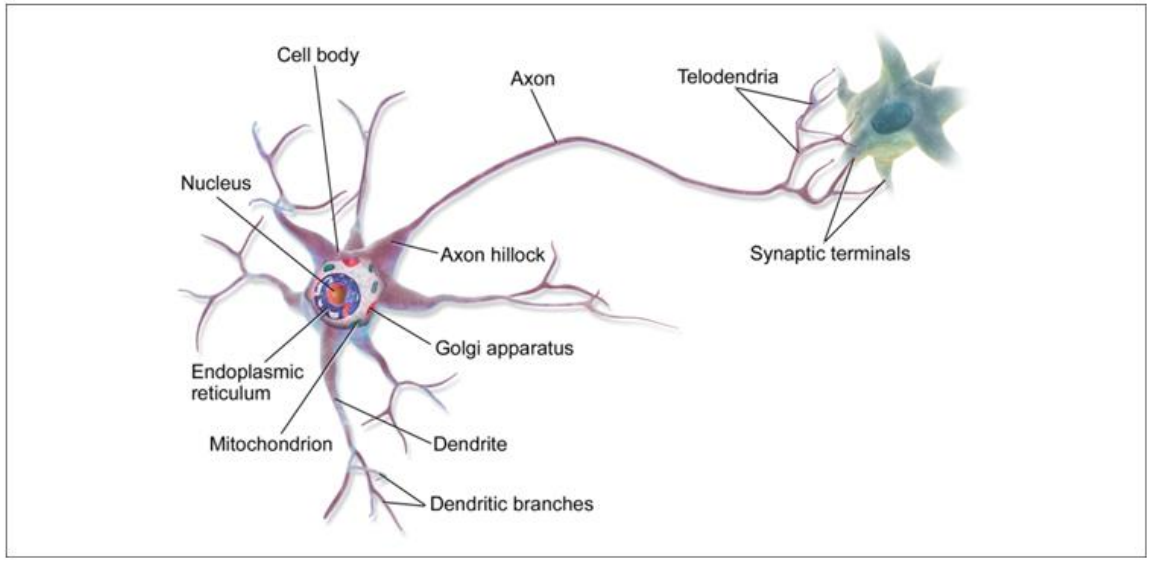


Figure 10-1. Biological neuron[4]

The Perceptron

- The *Perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron (see Figure 10-4) 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 ($z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{x}^\intercal \mathbf{w}$),
- then applies a *step function* to that sum and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$, where $z = \mathbf{x}^\intercal \mathbf{w}$.

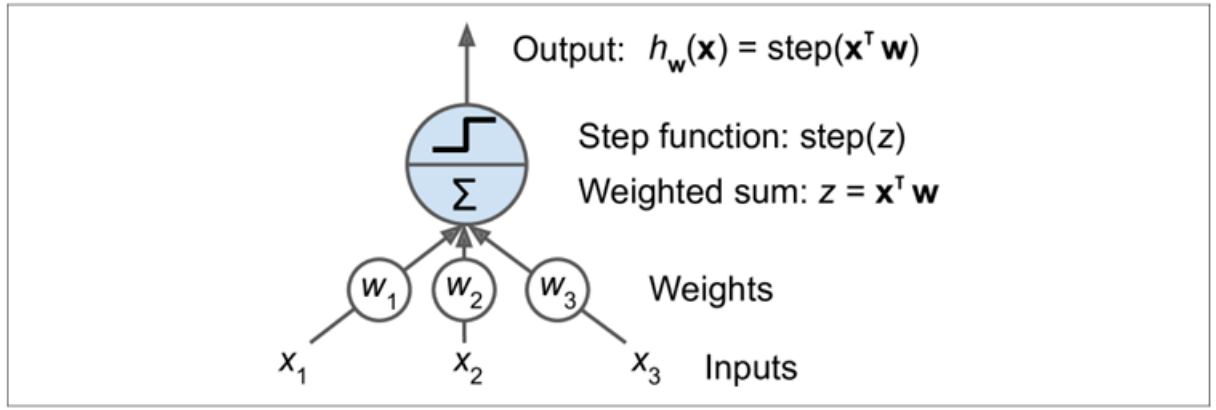
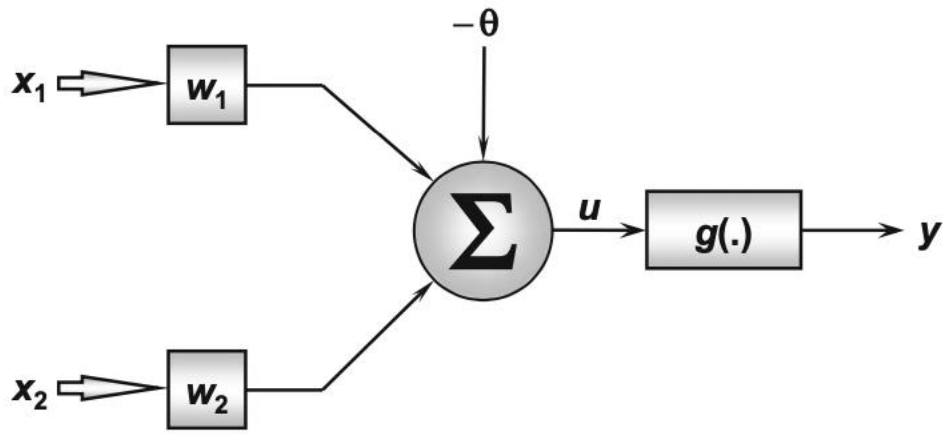
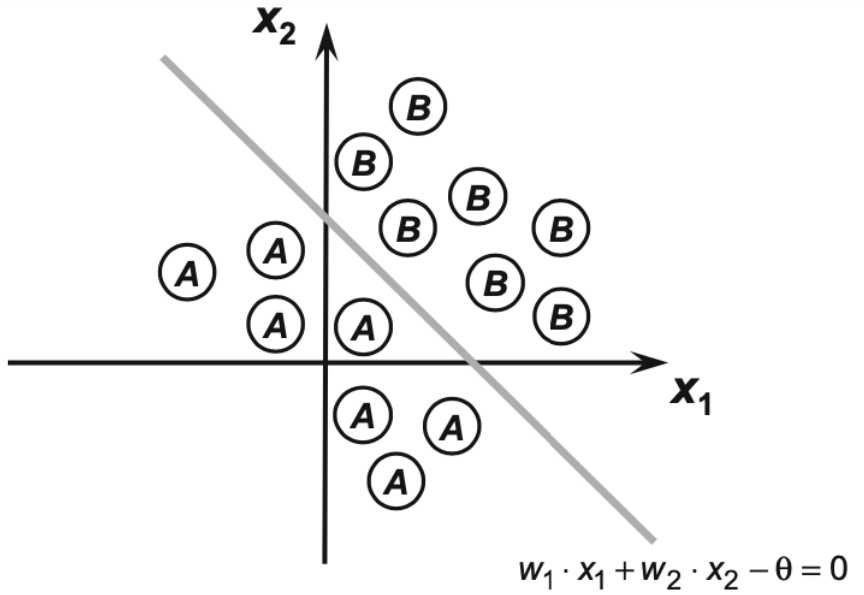


Figure 10-4. Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a step function

$$y = \begin{cases} 1, & \text{if } \sum w_i \cdot x_i - \theta \geq 0 \Leftrightarrow w_1 \cdot x_1 + w_2 \cdot x_2 - \theta \geq 0 \\ -1, & \text{if } \sum w_i \cdot x_i - \theta < 0 \Leftrightarrow w_1 \cdot x_1 + w_2 \cdot x_2 - \theta < 0 \end{cases}$$





$$w_1 \cdot x_1 + w_2 \cdot x_2 - \theta = 0$$

The most common step function used in Perceptrons is the *Heaviside step function* (see Equation 10-1). Sometimes the sign function is used instead.

Equation 10-1. Common step functions used in Perceptrons (assuming threshold = 0)

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

- It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class.
- Otherwise, it outputs the negative class (just like a Logistic Regression or linear SVM classifier).
- Training a TLU in this case means finding the right values for w_0, w_1, and w_2

- 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*.
- Moreover, an extra bias feature is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a *bias neuron*, which outputs 1 all the time.
- A Perceptron with two inputs and three outputs is represented in Figure 10-5. This Perceptron can classify instances simultaneously into three different binary classes, which makes it a multioutput classifier.

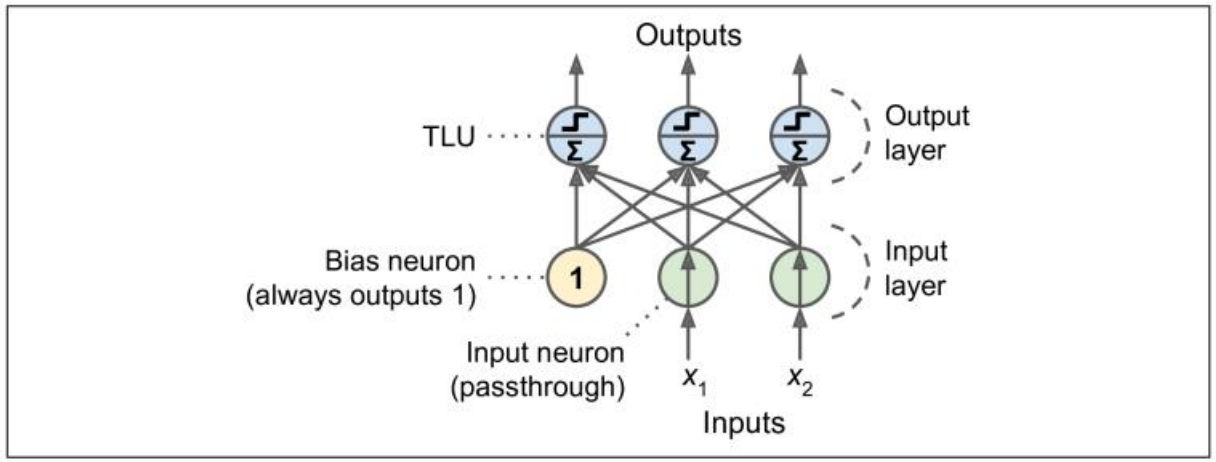


Figure 10-5. Architecture of a Perceptron with two input neurons, one bias neuron, and three output neurons

Equation 10-2. Computing the outputs of a fully connected layer

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

- As always, **X** represents the matrix of input features. It has one row per instance and one column per feature.
- The weight matrix **W** contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector **b** contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function ϕ is called the *activation function*: when the artificial neurons are TLUs, it is a step function.

So, how is a Perceptron trained? The Perceptron training algorithm proposed by Rosenblatt was largely inspired by *Hebb's rule*.

Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction; the Perceptron learning rule reinforces connections that help reduce the error.

More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

Equation 10-3. Perceptron learning rule (weight update)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

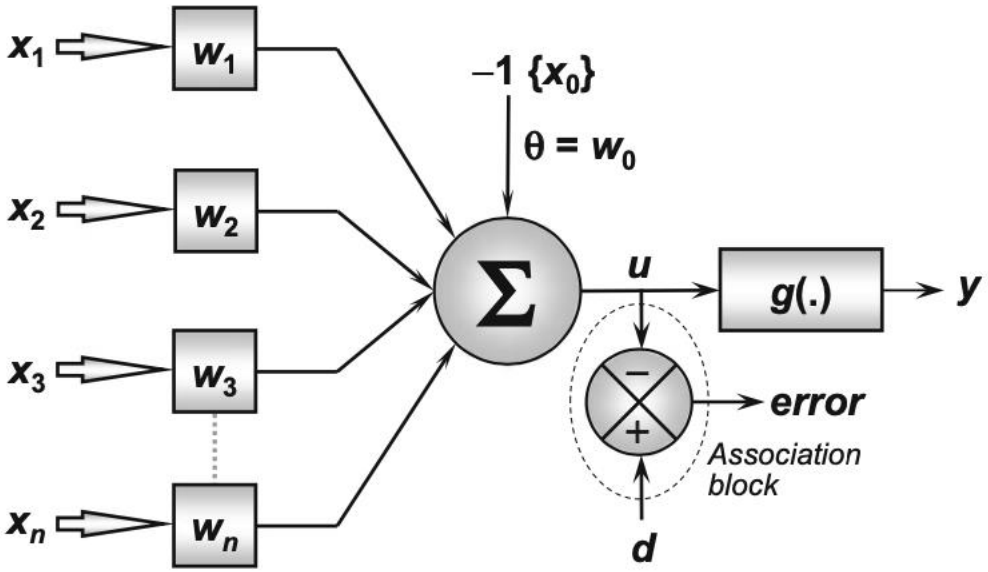
- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

The ADALINE (Adaptive Linear Element)

- Similar to Perceptron, the **ADALINE also consists of a single neural layer**, and **a single artificial neuron** composes it.

- The arrangement of several ADALINEs into a single network is called **MADALINE** (Multiple ADALINE) (Widrow and Winter 1988).

Just like the Perceptron, due to its structural simplicity, the **ADALINE network is majorly used on pattern classification problems involving only two distinct classes**.



$$u = \sum_{i=1}^{n} w_i \cdot x_i - \theta \Leftrightarrow u = \sum_{i=0}^{n} w_i \cdot x_i$$

$$y = g(u),$$

❖ The major contribution of the ADALINE was the **introduction of a learning algorithm named Delta rule**

Delta rule:

❖ Or Widrow-Hoff learning rule, also known as LMS (Least Mean Square) algorithm or Gradient Descent method.

❖ The idea of the Delta rule, when *p* training samples are available, is to **minimize the difference between the desired output {d} and the response {u}** of the linear combiner, considering all *p* samples, in order to adjust the weights and threshold of the neuron.

More specifically, the rule performs the minimization of the squared error between u and d to adjust the weight vector $\boldsymbol{w} = [\theta \quad w_1 \quad w_2 \cdots w_n]^T$ of the network. In summary, the objective is to obtain an optimal $\boldsymbol{w}^*$ so that the squared error $\{E(\boldsymbol{w}^*)\}$ of the whole sample set is as low as possible. In mathematical notation, considering an optimal weight configuration, it can be stated that:

$$E(\boldsymbol{w}^*) \leq E(\boldsymbol{w}), \text{ for } \forall \boldsymbol{w} \in \Re^{n+1} \tag{4.4}$$

$$E(\boldsymbol{w}^*) \leq E(\boldsymbol{w}), \text{ for } \forall \boldsymbol{w} \in \Re^{n+1} \tag{4.4}$$

The function of the squared error related to the p training samples is defined by:

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{p}(d^{(k)} - u)^2 \tag{4.5}$$

$$u = \sum_{i=1}^{n} w_i \cdot x_i - \theta \Leftrightarrow u = \sum_{i=0}^{n} w_i \cdot x_i \tag{4.1}$$

Substituting the result of (4.1) into (4.5) outcomes:

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{p}\left(d^{(k)} - \left(\sum_{i=1}^{n} w_i \cdot x_i^{(k)} - \theta\right)\right)^2 \tag{4.6}$$

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{p}\left(d^{(k)} - (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(k)} - \theta)\right)^2 \tag{4.7}$$

The next step consists of the application of the gradient operator on the mean squared error with respect to vector $\boldsymbol{w}$, in order to search for an optimal value for the squared error function given by (4.7), that is:

$$\nabla E(\boldsymbol{w}) = \frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} \tag{4.8}$$

Using the result of (4.7) in (4.8) results:

$$\nabla E(\boldsymbol{w}) = \sum_{k=1}^{p} \left(d^{(k)} - (\boldsymbol{w}^T \cdot \boldsymbol{x}^{(k)} - \theta) \right) \cdot (-\boldsymbol{x}^{(k)}) \tag{4.9}$$

Returning the value of (4.1) into (4.10) results:

$$\nabla E(\boldsymbol{w}) = -\sum_{k=1}^{p} (d^{(k)} - u) \cdot \boldsymbol{x}^{(k)} \tag{4.10}$$

Finally, the steps for adapting the weight vector must be executed in the opposite direction of the gradient, because the optimization goal is to minimize the squared error. In this condition, the variation $\Delta \boldsymbol{w}$ to update the ADALINE weight vector is given by:

$$\Delta \mathbf{w} = -\eta \cdot \nabla E(\mathbf{w}) \tag{4.11}$$

Inserting the result of (4.10) in (4.11), we obtain:

$$\Delta \boldsymbol{w} = \eta \sum_{k=1}^{p} (d^{(k)} - u) \cdot \boldsymbol{x}^{(k)} \tag{4.12}$$

In a complementary way, we can express (4.12) by:

$$\boldsymbol{w}^{\text{current}} = \boldsymbol{w}^{\text{previous}} + \eta \sum_{k=1}^{p} (d^{(k)} - u) \cdot \boldsymbol{x}^{(k)} \tag{4.13}$$

Alternatively, to simplify the expression, the update of $\boldsymbol{w}$ can be performed discretely after presenting each kth training sample, that is:

$$\boldsymbol{w}^{\text{current}} = \boldsymbol{w}^{\text{previous}} + \eta \cdot (d^{(k)} - u) \cdot \boldsymbol{x}^{(k)}, \quad \text{with } k = 1 \ldots p \tag{4.14}$$

In algorithmic notation, the previous expression is equivalent to

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \cdot (d^{(k)} - u) \cdot \boldsymbol{x}^{(k)}, \quad \text{with } k = 1 \ldots p, \tag{4.15}$$

where:

$\boldsymbol{w} = [\,\theta \quad w_1 \quad w_2 \cdots w_n\,]^T$ is the vector containing the threshold and weights;
$\boldsymbol{x}^{(k)} = [\,-1 \quad x_1^{(k)} \quad x_2^{(k)} \cdots x_n^{(k)}\,]^T$ is the kth training sample;
$d^{(k)}$ is the kth training sample;
u is the output value of the linear additive element; and
η is a constant that defines the learning rate.

Similarly to the Perceptron, the learning rate η defines how fast the network training process advances in the direction of the minimum point of the squared error function given by (4.5). Usually, it assumes values within the range of $0 < \eta < 1$.

❖ **Weight adjustment for Perceptron**

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \cdot (d^{(k)} - y) \cdot \boldsymbol{x}^{(k)},$$

❖ **Weight adjustment for ADALINE**

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta \cdot (d^{(k)} - u) \cdot \boldsymbol{x}^{(k)}, \quad \text{with } k = 1 \ldots p,$$

❖ The stopping criterion is stipulated **by employing the mean squared error** with respect to all training samples is defined by:

$$\bar{E}(\boldsymbol{w}) = \frac{1}{p} \sum_{k=1}^{p} \left(d^{(k)} - u \right)^2 \qquad (4.16)$$

❖ The algorithm converges when the difference of the mean squared error between two successive epochs is small enough. That is:

$$\left| \bar{E}(\boldsymbol{w}^{\text{current}}) - \bar{E}(\boldsymbol{w}^{\text{previous}}) \right| \leq \varepsilon, \qquad (4.17)$$

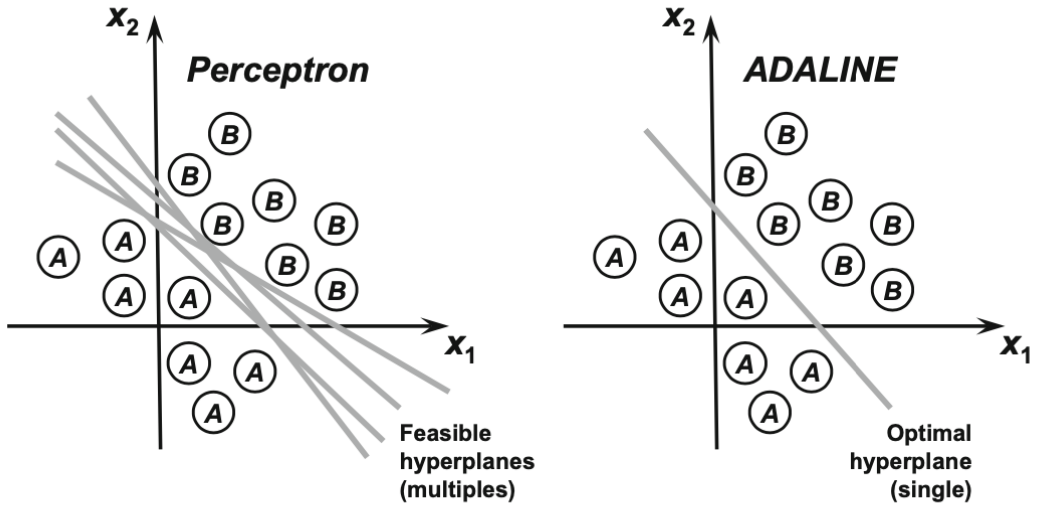


Fig. 4.5 Comparison between the separation of classes performed by the Perceptron and the ADALINE

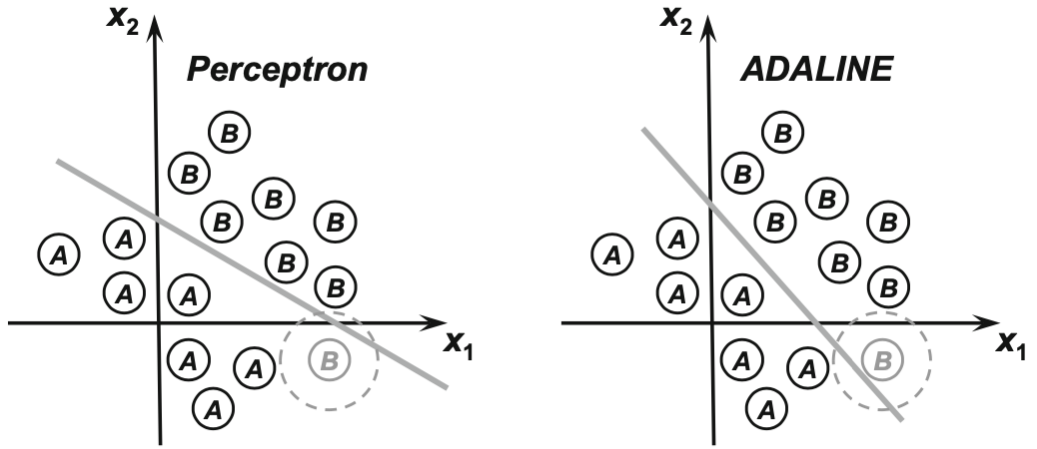


Fig. 4.6 Application of the Perceptron and the ADALINE when considering a noisy sample

Scikit-Learn provides a `Perceptron` class that implements a single-TLU network. It can be used pretty much as you would expect—for example, on the iris dataset (introduced in Chapter 4):

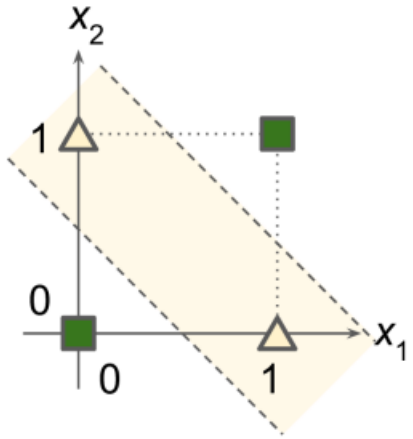
```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

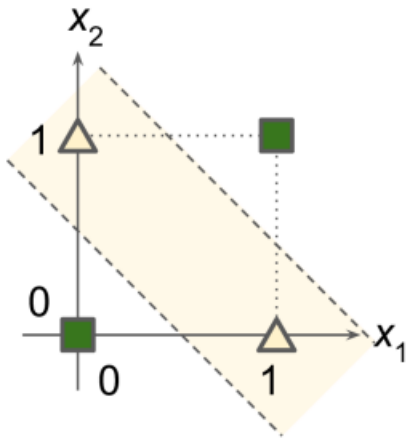
You may have noticed that the Perceptron learning algorithm strongly resembles Stochastic Gradient Descent.

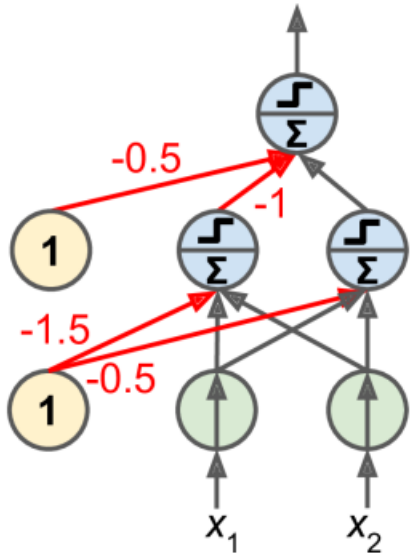
In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters: `loss="perceptron"`, `learning_rate="constant"`, `eta=0.1` (the learning rate), and `penalty=None` (no regularization).

- In their 1969 monograph *Perceptrons*, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons—in particular, the fact that they are incapable of solving some trivial problems (e.g., the *Exclusive OR* (XOR) classification problem).

- It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons.
- The resulting ANN is called a *Multilayer Perceptron* (MLP). An MLP can solve the XOR problem, as you can verify by computing the output of the MLP : with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. All connections have a weight equal to 1, except the four connections where the weight is shown.







The Multilayer Perceptron and Backpropagation

- ✓ An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer* (see Figure 10-7).
- ✓ This architecture is an example of a feedforward neural network (FNN).

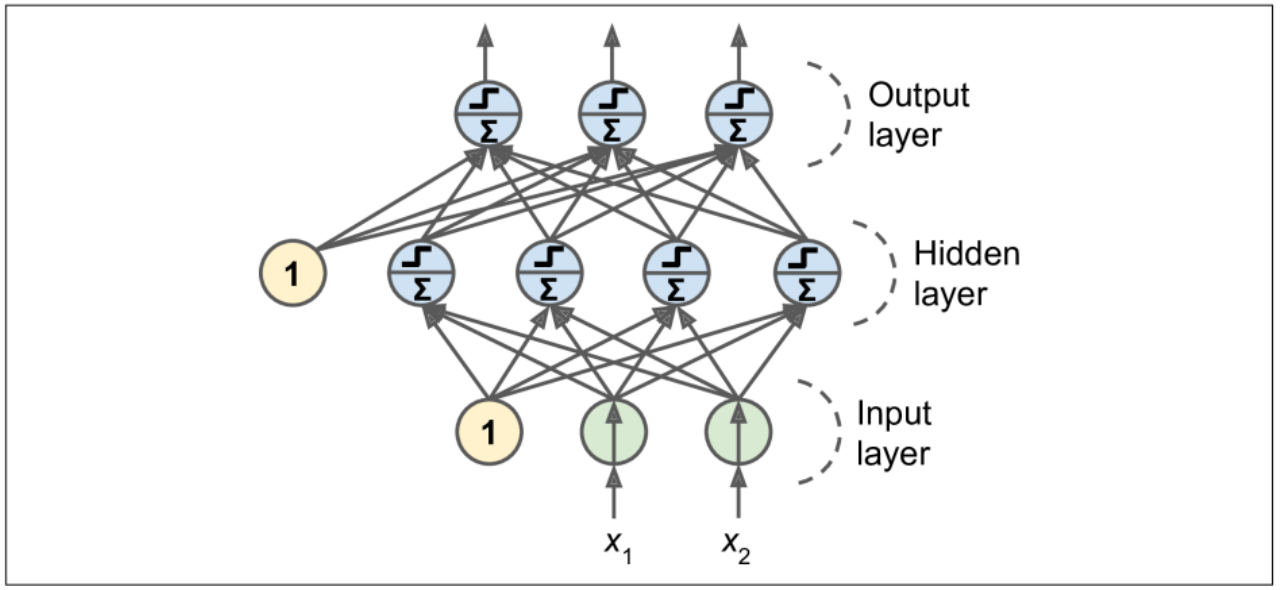
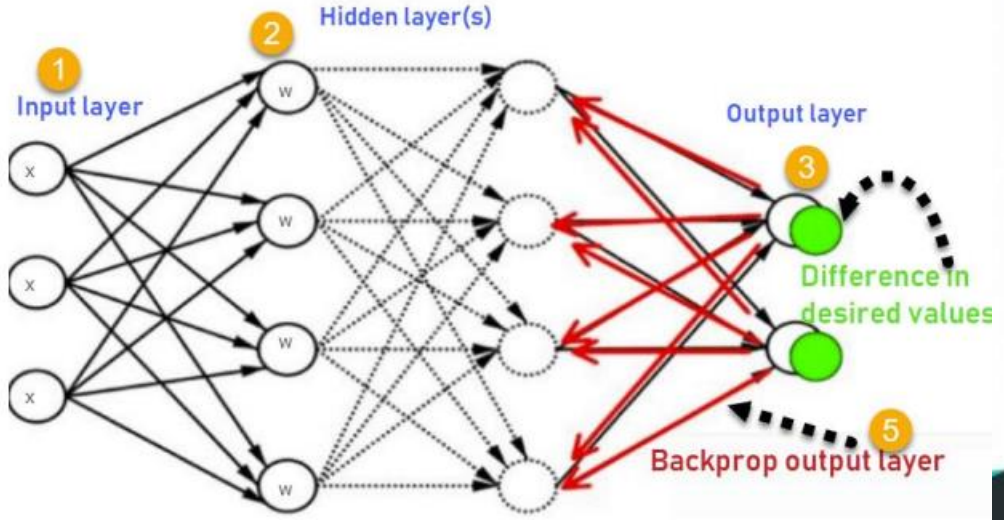


Figure 10-7. Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

- For many years researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper that introduced the *backpropagation* training algorithm, which is still used today.
- In short, it is Gradient Descent (introduced in Chapter 4) using an efficient technique for computing the gradients automatically: in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter.
- This algorithm is so important that it's worth summarizing it: for each training instance, the backpropagation algorithm first makes a prediction (forward pass) and measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally tweaks the connection weights to reduce the error (Gradient Descent step).
-



- In order for this algorithm to work properly, its authors made a key change to the MLP's architecture: they replaced the step function with the logistic (sigmoid) function, $\sigma(z) = 1 / (1 + \exp(-z))$. This was essential because the step function contains only flat segments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface), while the logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step.
- In fact, the backpropagation algorithm works well with many other activation functions, not just the logistic function. Here are two other popular choices:

✓ ***The hyperbolic tangent function:*** $\tanh(z) = 2\sigma(2z) - 1$

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.

✓ ***The Rectified Linear Unit function: ReLU(z) = max(0, z)***

The ReLU function 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 $z < 0$. In practice, however, it 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 (we will come back to this in Chapter 11).

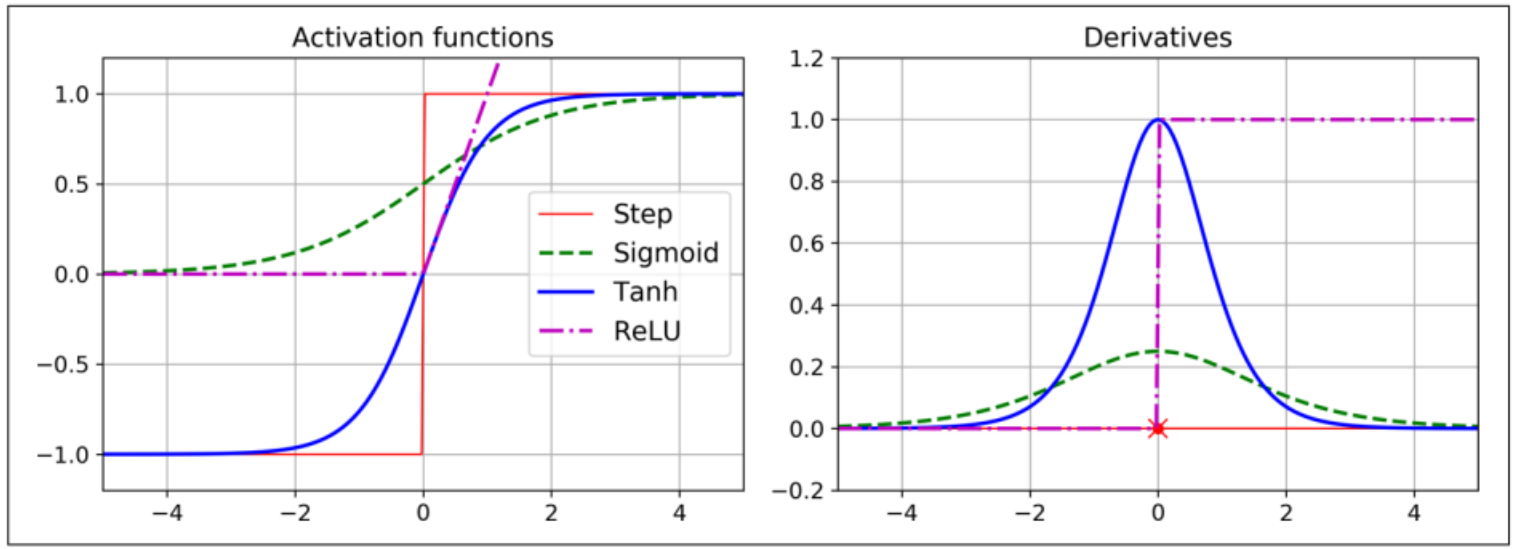
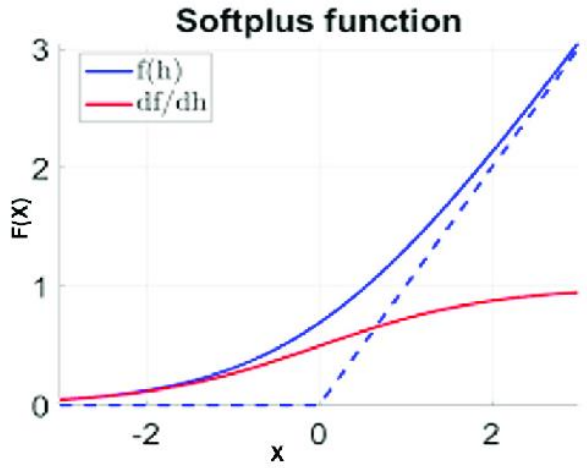


Figure 10-8. Activation functions and their derivatives

MLP for Regression

- 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.
- 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. 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 to the appropriate range: 0 to 1 for the logistic function and –1 to 1 for the hyperbolic tangent.



- ✓ 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. Alternatively, you can use the Huber loss, which is a combination of both.

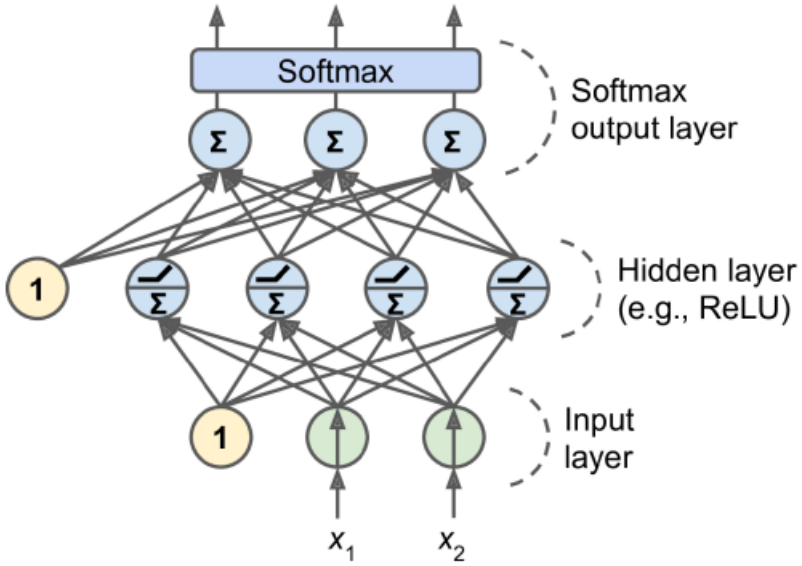
The Huber loss is quadratic when the error is smaller than a threshold δ (typically 1) but linear when the error is larger than δ. The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{if } |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{if } |a| > \delta \end{cases}$$

where $a = y - f(x)$ is the residual, and δ is a threshold parameter.

MLP for Classification

- MLPs can also be used for classification tasks. 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.
- MLPs can also easily handle multilabel binary classification tasks. In this case, you would need two output neurons, both using the logistic activation function. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels.
- If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see Figure 10-9). The softmax function (introduced in Chapter 4) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive). This is called multiclass classification.



In Softmax, each class has its own dedicated parameter vector θ (k) .
All these vectors are typically stored as rows in a parameter matrix Θ.

The idea is simple: when given an instance **x**, the Softmax Regression model first computes a score $s_k(\mathbf{x})$ for each class k, then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores. The equation to compute $s_k(\mathbf{x})$ should look familiar, as it is just like the equation for Linear Regression prediction (see Equation 4-19).

Equation 4-19. Softmax score for class k

$$s_k(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\theta}^{(k)}$$

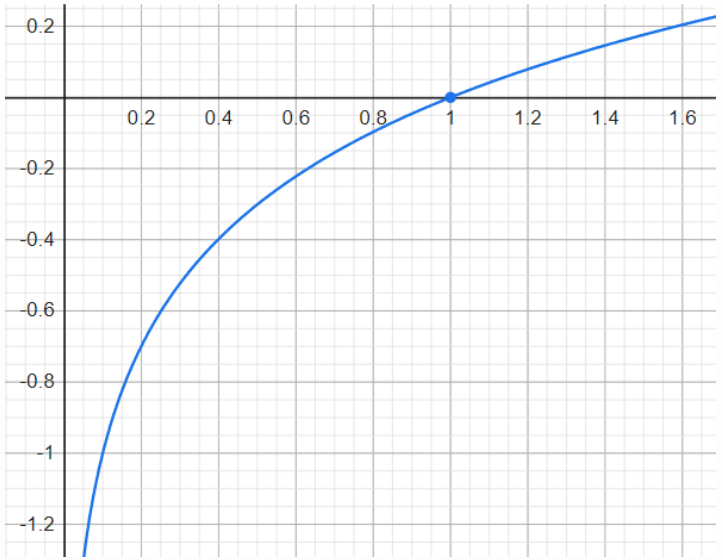
Once you have computed the score of every class for the instance **x**, you can estimate the probability $\hat{p}_k$ that the instance belongs to class k by running the scores through the softmax function (Equation 4-20). The function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials). The scores are generally called logits or log-odds (although they are actually unnormalized log-odds).

Equation 4-20. Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}$$

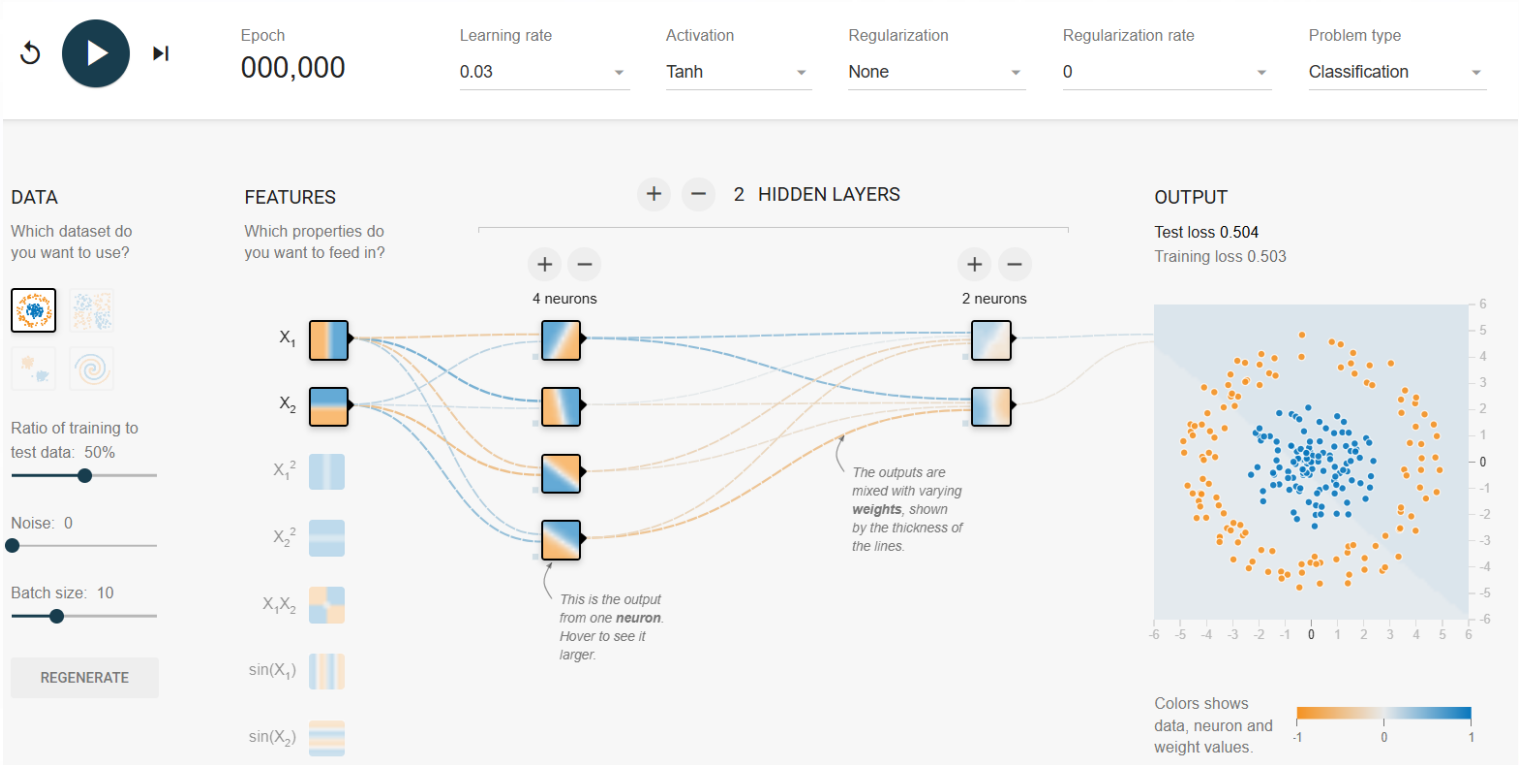
- ✓ Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (also called the log loss, see Chapter 4) is generally a good choice. Cross entropy is frequently used to measure how well a set of estimated class probabilities matches the target classes.

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$



https://playground.tensorflow.org/

The TensorFlow Playground is a handy neural network simulator built by the TensorFlow team. In this exercise, you will train several binary classifiers in just a few clicks, and tweak the model's architecture and its hyperparameters to gain some intuition on how neural networks work and what their hyperparameters do. Take some time to explore the instructions followed by Exercise 1 in Chapter 10.



Implementing MLPs with 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 *https://keras.io/*.
- TensorFlow itself now comes bundled with its own Keras implementation, tf.keras. It only supports TensorFlow as the backend, but it has the advantage of offering some very useful extra features: for example, it supports TensorFlow's Data API, which makes it easy to load and preprocess data efficiently. For this reason, we will use tf.keras in this book.

- 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), so once you know Keras, it is not difficult to switch to PyTorch, if you ever want to. PyTorch's popularity grew exponentially in 2018, largely thanks to its simplicity and excellent documentation, which were not TensorFlow 1.x's main strengths. However, TensorFlow 2 is arguably just as simple as PyTorch, as it has adopted Keras as its official high-level API and its developers have greatly simplified and cleaned up the rest of the API.

Install tensorflow 2.0

```
$ python3 -m pip install -U tensorflow
```

For GPU support, at the time of this writing you need to install `tensorflow-gpu` instead of `tensorflow`, but the TensorFlow team is working on having a single library that will support both CPU-only and GPU-equipped systems. You will still need to install extra libraries for GPU support (see *https://tensorflow.org/install* for more details). We will look at GPUs in more depth in Chapter 19.

To test your installation, open a Python shell or a Jupyter notebook, then import TensorFlow and tf.keras and print their versions:

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

The second version is the version of the Keras API implemented by tf.keras. Note that it ends with `-tf`, highlighting the fact that tf.keras implements the Keras API, plus some extra TensorFlow-specific features.

Now let's use tf.keras! We'll start by building a simple image classifier.

Building an Image Classifier Using the Sequential API

Figure 10-11 shows some samples from the Fashion MNIST dataset.

Figure 10-11. Samples from Fashion MNIST

https://colab.research.google.com/github/ageron/handson-ml2/blob/master/10_neural_nets_with_keras.ipynb#scrollTo=4jk6SluZiYWX

Creating the model using the Sequential API

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

- The first line creates a `Sequential` model. This is the simplest kind of Keras model for neural networks that are just composed of a single stack of layers connected sequentially. This is called the Sequential API.
- Next, we build the first layer and add it to the model. It is a `Flatten` layer whose role is to convert each input image into a 1D array: if it receives input data `X`, it computes `X.reshape(-1, 1)`. This layer does not have any parameters; it is just there to do some simple preprocessing. Since it is the first layer in the model, you should specify the `input_shape`, which doesn't include the batch size, only the shape of the instances. Alternatively, you could add a `keras.layers.InputLayer` as the first layer, setting `input_shape=[28,28]`.
- Next we add a `Dense` hidden layer with 300 neurons. It will use the ReLU activation function. Each `Dense` layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron). When it receives some input data, it computes Equation 10-2.
- Then we add a second `Dense` hidden layer with 100 neurons, also using the ReLU activation function.
- Finally, we add a `Dense` output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).

Instead of adding the layers one by one as we just did, you can pass a list of layers when creating the `Sequential` model:

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

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235,500
dense_1 (Dense)	(None, 100)	30,100
dense_2 (Dense)	(None, 10)	1,010

```
Total params: 266,610 (1.02 MB)
Trainable params: 266,610 (1.02 MB)
Non-trainable params: 0 (0.00 B)
```

This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data. We will come back to this later.

You can easily get a model's list of layers, to fetch a layer by its index, or you can fetch it by name:

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods. For a `Dense` layer, this includes both the connection weights and the bias terms:

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       ...,
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
         0.00272203, -0.06793761]], dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., ...,  0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

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"])
```

- ✓ ″sparse_categorical_crossentropy″ loss 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.
- ✓ ″categorical_crossentropy “ if instead we had one target probability per class for each instance (such as one-hot vectors, e.g. `[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]` to represent class 3), then we would need to use it.
- ✓ ″binary_crossentropy″ if we were doing binary classification (with one or more binary labels), then we would use the ″sigmoid″ (i.e., logistic) activation function in the output layer instead of the ″softmax″ activation function, and we would use the ″binary_crossentropy″ loss.

If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, use the `keras.utils.to_categorical()` function. To go the other way round, use the `np.argmax()` function with `axis=1`.

- Regarding the optimizer, `"sgd"` means that we will train the model using simple Stochastic Gradient Descent. In other words, Keras will perform the backpropagation algorithm.
- We will discuss more efficient optimizers in Chapter 11 (they improve the Gradient Descent part).

When using the `SGD` optimizer, it is important to tune the learning rate. So, you will generally want to use `optimizer=keras.optimizers.SGD(lr=???)` to set the learning rate, rather than `optimizer="sgd"`, which defaults to `lr=0.01`.

Training and evaluating the model

Now the model is ready to be trained. For this we simply need to call its `fit()` method:

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [======] - 3s 49us/sample - loss: 0.7218     - accuracy: 0.7660
                                     - val_loss: 0.4973 - val_accuracy: 0.8366
Epoch 2/30
55000/55000 [======] - 2s 45us/sample - loss: 0.4840     - accuracy: 0.8327
                                     - val_loss: 0.4456 - val_accuracy: 0.8480
[...]
Epoch 30/30
55000/55000 [======] - 3s 53us/sample - loss: 0.2252     - accuracy: 0.9192
                                     - val_loss: 0.2999 - val_accuracy: 0.8926
```

The `fit()` method returns a `History` object containing the training parameters (`history.params`), the list of epochs it went through (`history.epoch`), and most importantly 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). If you use this dictionary to create a pandas DataFrame and call its `plot()` method, you get the learning curves shown in Figure 10-12:

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

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range to [0-1]
plt.show()
```

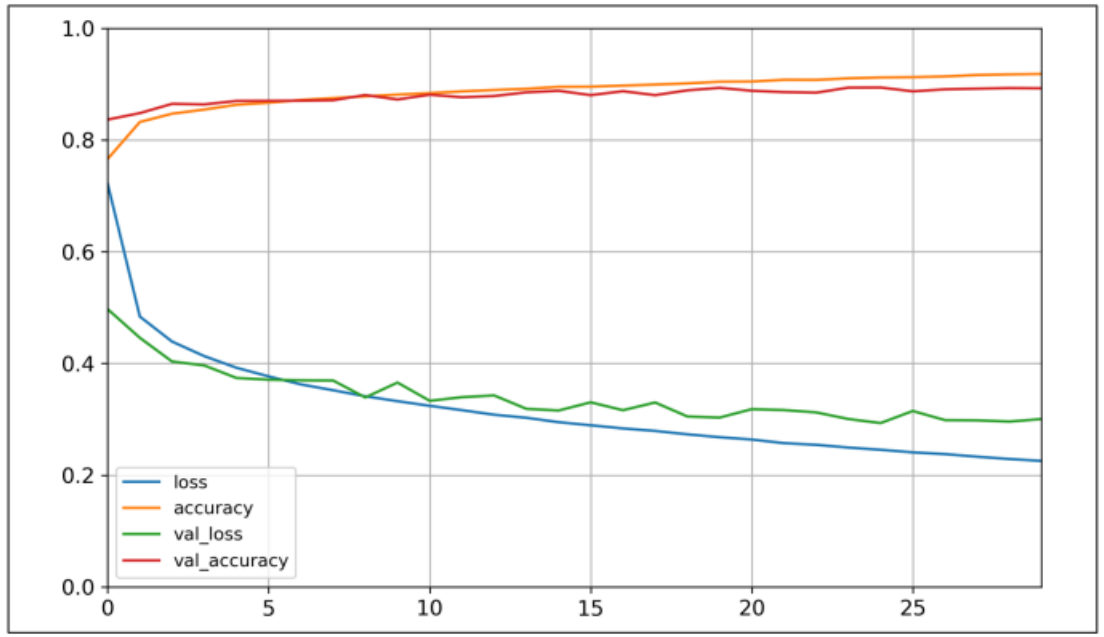


Figure 10-12. Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

The training curve should be shifted by half an epoch to the left

```
>>> model.evaluate(X_test, y_test)
10000/10000 [==========] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

Using the model to make predictions

Next, we can use the model's `predict()` method to make predictions on new instances. Since we don't have actual new instances, we will just use the first three instances of the test set:

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.03, 0.  , 0.01, 0.  , 0.96],
       [0.  , 0.  , 0.98, 0.  , 0.02, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the pre dict_classes() method instead:

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

Here, the classifier actually classified all three images correctly (these images are shown in Figure 10-13):

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

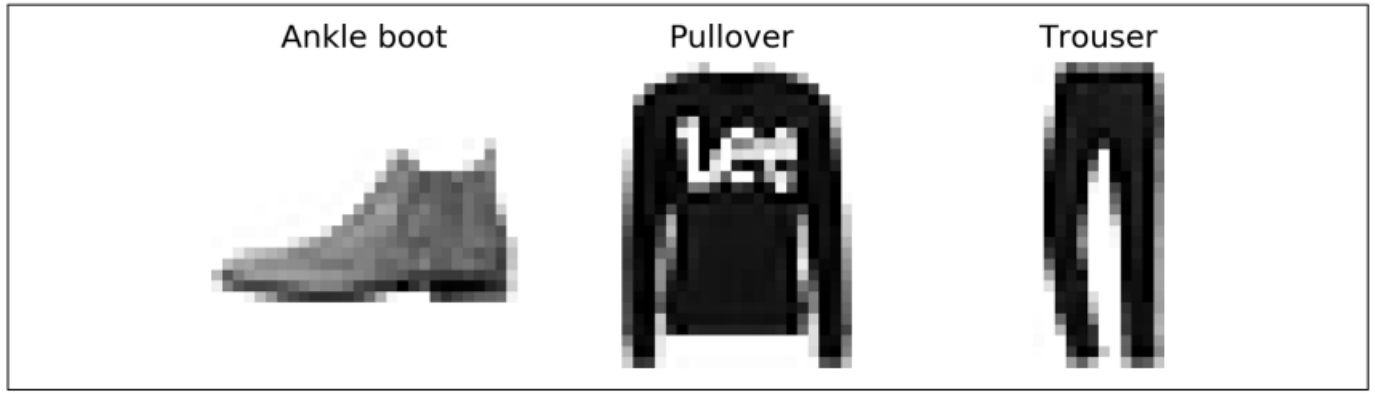


Figure 10-13. Correctly classified Fashion MNIST images

Building a Regression MLP Using the Sequential API

Let's switch to the California housing problem and tackle it using a regression neural network. For simplicity, we will use Scikit-Learn's `fetch_california_housing()` function to load the data. This dataset is simpler than the one we used in Chapter 2, since it contains only numerical features (there is no `ocean_proximity` feature), and there is no missing value. After loading the data, we split it into a training set, a validation set, and a test set, and we scale all the features:

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

housing = fetch_california_housing()

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

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

Using the Sequential API to build, train, evaluate, and use a regression MLP to make predictions is quite similar to what we did for classification. The main differences are the fact that the output layer has a single neuron (since we only want to predict a single value) and uses no activation function, and the loss function is the mean squared error. Since the dataset is quite noisy, we just use a single hidden layer with fewer neurons than before, to avoid overfitting:

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # pretend these are new instances
y_pred = model.predict(X_new)
```

Building Complex Models Using the Functional API

One example of a nonsequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al. It connects all or part of the inputs directly to the output layer, as shown in Figure 10-14. This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path). In contrast, a regular MLP forces all the data to flow through the full stack of layers; thus, simple patterns in the data may end up being distorted by this sequence of transformations.

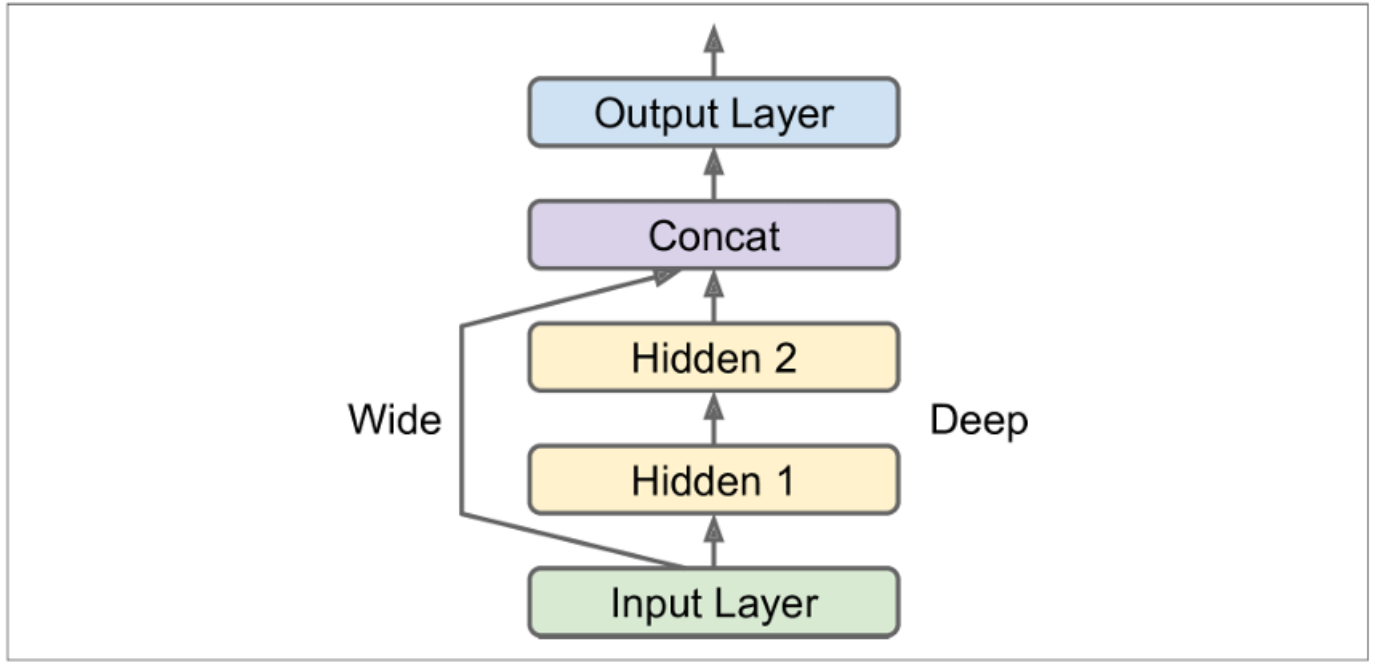


Figure 10-14. Wide & Deep neural network

Let's build such a neural network to tackle the California housing problem:

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.Concatenate()([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

- First, we need to create an `Input` object.[18] This is a specification of the kind of input the model will get, including its `shape` and `dtype`. A model may actually have multiple inputs, as we will see shortly.
- Next, we create a `Dense` layer with 30 neurons, using the ReLU activation function. As soon as it is created, notice that we call it like a function, passing it the input. This is why this is called the Functional API. Note that we are just telling Keras how it should connect the layers together; no actual data is being processed yet.
- We then create a second hidden layer, and again we use it as a function. Note that we pass it the output of the first hidden layer.
- Next, we create a `Concatenate` layer, and once again we immediately use it like a function, to concatenate the input and the output of the second hidden layer. You may prefer the `keras.layers.concatenate()` function, which creates a `Concatenate` layer and immediately calls it with the given inputs.
- Then we create the output layer, with a single neuron and no activation function, and we call it like a function, passing it the result of the concatenation.
- Lastly, we create a Keras `Model`, specifying which inputs and outputs to use.

But what if you want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path (see Figure 10-15)? In this case, one solution is to use multiple inputs. For example, suppose we want to send five features through the wide path (features 0 to 4), and six features through the deep path (features 2 to 7):

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

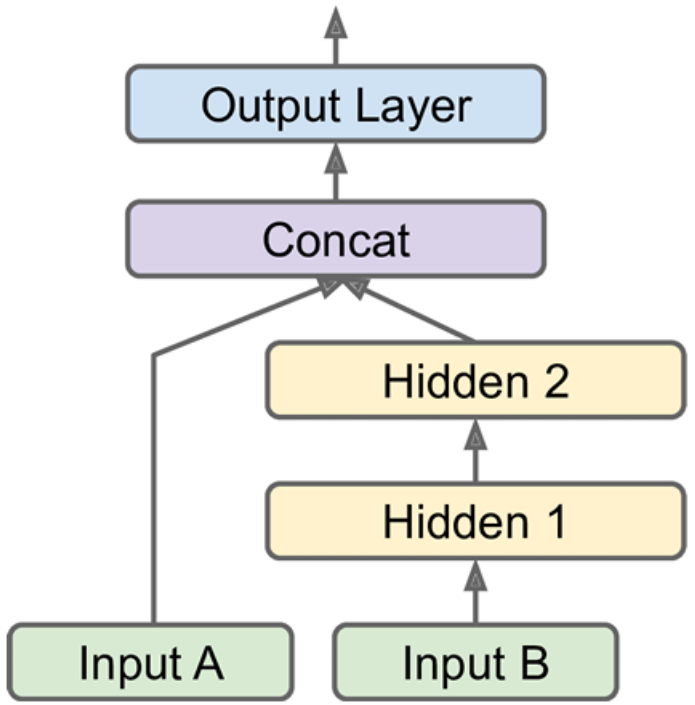


Figure 10-15. Handling multiple inputs

The code is self-explanatory. You should name at least the most important layers, especially when the model gets a bit complex like this. Note that we specified `inputs=[input_A, input_B]` when creating the model. Now we can compile the model as usual, but when we call the `fit()` method, instead of passing a single input matrix `X_train`, we must pass a pair of matrices `(X_train_A, X_train_B)`: one per input.[19] The same is true for `X_valid`, and also for `X_test` and `X_new` when you call `evaluate()` or `predict()`:

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                    validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```

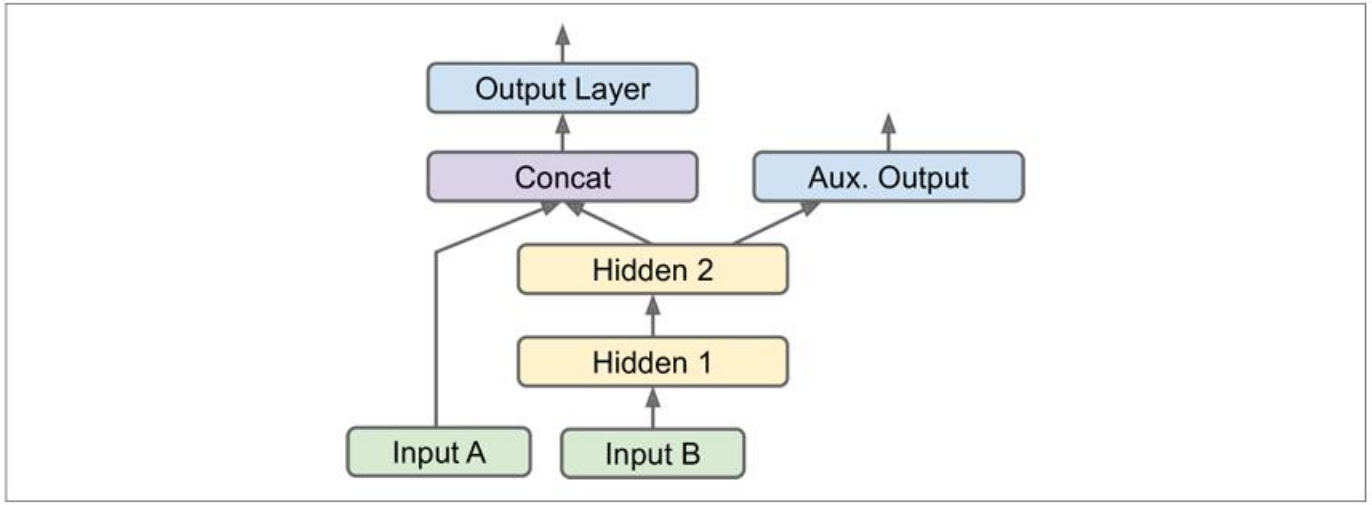


Figure 10-16. Handling multiple outputs, in this example to add an auxiliary output for regularization

- Another use case is as a regularization technique (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize). For example, you may want to add some auxiliary outputs in a neural network architecture (see Figure 10-16) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.

```
[...] # Same as above, up to the main output layer
output = keras.layers.Dense(1, name="main_output")(concat)
aux_output = keras.layers.Dense(1, name="aux_output")(hidden2)
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

Each output will need its own loss function. Therefore, when we compile the model, we should pass a list of losses[20] (if we pass a single loss, Keras will assume that the same loss must be used for all outputs). By default, Keras will compute all these losses and simply add them up to get the final loss used for training. We care much more about the main output than about the auxiliary output (as it is just used for regularization), so we want to give the main output's loss a much greater weight. Fortunately, it is possible to set all the loss weights when compiling the model:

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

Now when we train the model, we need to provide labels for each output. In this example, the main output and the auxiliary output should try to predict the same thing, so they should use the same labels. So instead of passing `y_train`, we need to pass `(y_train, y_train)` (and the same goes for `y_valid` and `y_test`):

```
history = model.fit(
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,
    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

When we evaluate the model, Keras will return the total loss, as well as all the individual losses:

```
total_loss, main_loss, aux_loss = model.evaluate(
    [X_test_A, X_test_B], [y_test, y_test])
```

Similarly, the `predict()` method will return predictions for each output:

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

As you can see, you can build any sort of architecture you want quite easily with the Functional API. Let's look at one last way you can build Keras models.

Saving and Restoring a Model

When using the Sequential API or the Functional API, saving a trained Keras model is as simple as it gets:

```
model = keras.models.Sequential([...]) # or keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")
```

Keras will use the HDF5 format to save both the model's architecture (including every layer's hyperparameters) and the values of all the model parameters for every layer (e.g., connection weights and biases). It also saves the optimizer (including its hyperparameters and any state it may have). In Chapter 19, we will see how to save a tf.keras model using TensorFlow's SavedModel format instead.

You will typically have a script that trains a model and saves it, and one or more scripts (or web services) that load the model and use it to make predictions. Loading the model is just as easy:

```
model = keras.models.load_model("my_keras_model.h5")
```

But what if training lasts several hours? This is quite common, especially when training on large datasets. In this case, you should not only save your model at the end of training, but also save checkpoints at regular intervals during training, to avoid losing everything if your computer crashes. But how can you tell the `fit()` method to save checkpoints? **Use 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. For example, the `ModelCheckpoint` callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

```
[...] # build and compile the model
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

Moreover, if you use a validation set during training, you can set `save_best_only=True` when creating the `ModelCheckpoint`. In this case, it will only save your model when its performance on the validation set is the best so far. This way, you do not need to worry about training for too long and overfitting the training set: simply restore the last model saved after training, and this will be the best model on the validation set. The following code is a simple way to implement early stopping (introduced in Chapter 4):

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # roll back to best model
```

Another way to implement early stopping is to simply use the `EarlyStopping` call back. It will interrupt training when it measures no progress on the validation set for a number of epochs (defined by the `patience` argument), and it will roll back to the best model. You can combine both callbacks to save checkpoints of your model (in case your computer crashes) and interrupt training early when there is no more progress (to avoid wasting time and resources):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                  restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb])
```

The number of epochs can be set to a large value since training will stop automatically when there is no more progress. In this case, there is no need to restore the best model saved because the `EarlyStopping` callback will keep track of the best weights and restore them for you at the end of training.

Using TensorBoard for Visualization

TensorBoard is a great interactive visualization tool that you can use to view the learning curves during training, compare learning curves between multiple runs, visualize the computation graph, analyze training statistics, view images generated by your model, visualize complex multidimensional data projected down to 3D and automatically clustered for you, and more! This tool is installed automatically when you install TensorFlow, so you already have it.

To use it, you must modify 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 will monitor the log directory, and it will automatically pick up the changes and update the visualizations: this allows you to visualize live data (with a short delay), such as the learning curves during training. In general, you want to point the TensorBoard server to a root log directory and configure your program so that it writes to a different subdirectory every time it runs. This way, the same TensorBoard server instance will allow you to visualize and compare data from multiple runs of your program, without getting everything mixed up.

Let's start by defining the root log directory we will use for our TensorBoard logs, plus a small function that will generate a subdirectory path based on the current date and time so that it's different at every run. You may want to include extra information in the log directory name, such as hyperparameter values that you are testing, to make it easier to know what you are looking at in TensorBoard:

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # e.g., './my_logs/run_2019_06_07-15_15_22'
```

The good news is that Keras provides a nice `TensorBoard()` callback:

```
[...] # Build and compile your model
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[tensorboard_cb])
```

And that's all there is to it! It could hardly be easier to use. If you run this code, the `TensorBoard()` callback will take care of creating the log directory for you (along with its parent directories if needed), and during training it will create event files and write summaries to them. After running the program a second time (perhaps changing some hyperparameter value), you will end up with a directory structure similar to this one:

```
my_logs/
├── run_2019_06_07-15_15_22
│   ├── train
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
│   │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
│   │   └── plugins/profile/2019-06-07_15-15-32
│   │       └── local.trace
│   └── validation
│       └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
└── run_2019_06_07-15_15_49
    └── [...]
```

There's one directory per run, each containing one subdirectory for training logs and one for validation logs. Both contain event files, but the training logs also include profiling traces: this allows TensorBoard to show you exactly how much time the model spent on each part of your model, across all your devices, which is great for locating performance bottlenecks.

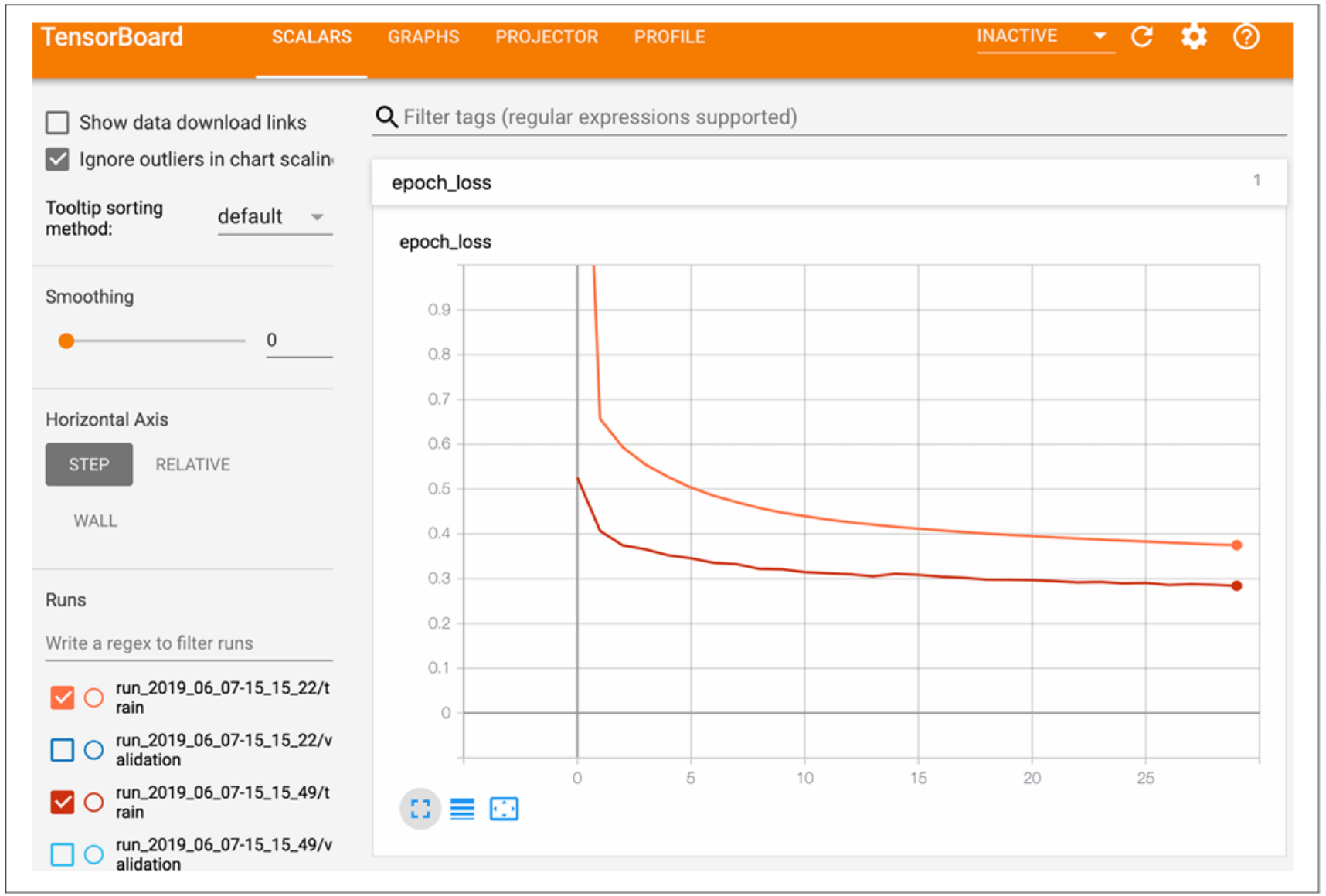


Figure 10-17. Visualizing learning curves with TensorBoard