## Multi-Layer Perceptron and Backpropagation

An MLP is composed of one (passthrough) input layer, one or more layers of LTUs, called *hidden layers*, and one final layer of LTUs called the *output layer* (see Figure 10-7). Every layer except the output layer includes a bias neuron and is fully connected to the next layer. When an ANN has two or more hidden layers, it is called a *deep neural network* (DNN).

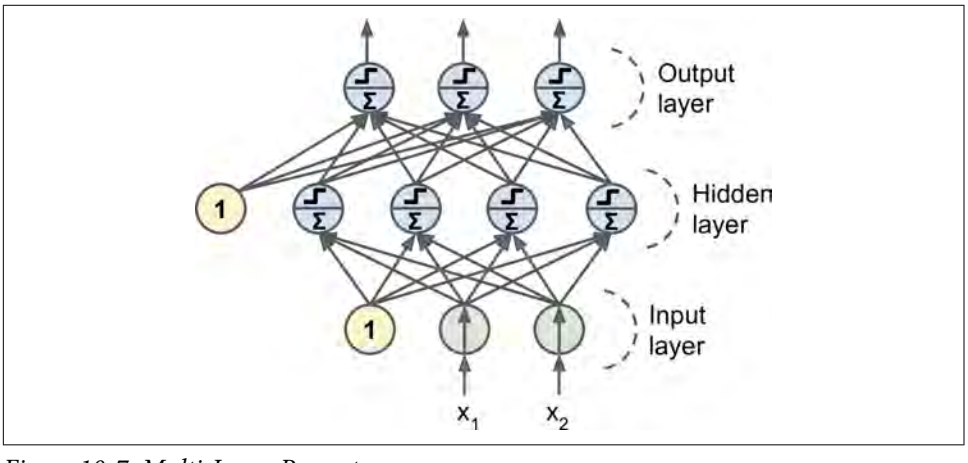


*Figure 10-7. Multi-Layer Perceptron*

For many years researchers struggled to find a way to train MLPs, without success. But in 1986, D. E. Rumelhart et al. published a groundbreaking article[8] introducing the *backpropagation* training algorithm.[9] Today we would describe it as Gradient Descent using reverse-mode autodiff (Gradient Descent was introduced in Chapter 4, and autodiff was discussed in Chapter 9).

For each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer (this is the forward pass, just like when making predictions). Then it measures the network's output error (i.e., the difference between the desired output and the actual output of the network), and it computes how much each neuron in the last hidden layer contributed to each output neuron's error. It then proceeds to measure how much of these error contributions came from each neuron in the previous hidden layer—and so on 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 in the network (hence the name of the algorithm). If you check out the

8 "Learning Internal Representations by Error Propagation," D. Rumelhart, G. Hinton, R. Williams (1986).

9 This algorithm was actually invented several times by various researchers in different fields, starting with P. Werbos in 1974.

reverse-mode autodiff algorithm in Appendix D, you will find that the forward and reverse passes of backpropagation simply perform reverse-mode autodiff. The last step of the backpropagation algorithm is a Gradient Descent step on all the connection weights in the network, using the error gradients measured earlier.

Let's make this even shorter: for each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally slightly tweaks the connection weights to reduce the error (Gradient Descent step).

In order for this algorithm to work properly, the authors made a key change to the MLP's architecture: they replaced the step function with the logistic 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. The backpropagation algorithm may be used with other *activation functions*, instead of the logistic function. Two other popular activation functions are:

*The hyperbolic tangent function $\tanh(z) = 2\sigma(2z) - 1$*
: Just like the logistic function it 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), which tends to make each layer's output more or less normalized (i.e., centered around 0) at the beginning of training. This often helps speed up convergence.

*The ReLU function (introduced in Chapter 9)*
: $\text{ReLU}(z) = \max(0, z)$. It is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around). However, in practice it works very well and has the advantage of being fast to compute. Most importantly, the fact that it does not have a maximum output value also helps reduce some issues during Gradient Descent (we will come back to this in Chapter 11).

These popular activation functions and their derivatives are represented in Figure 10-8.

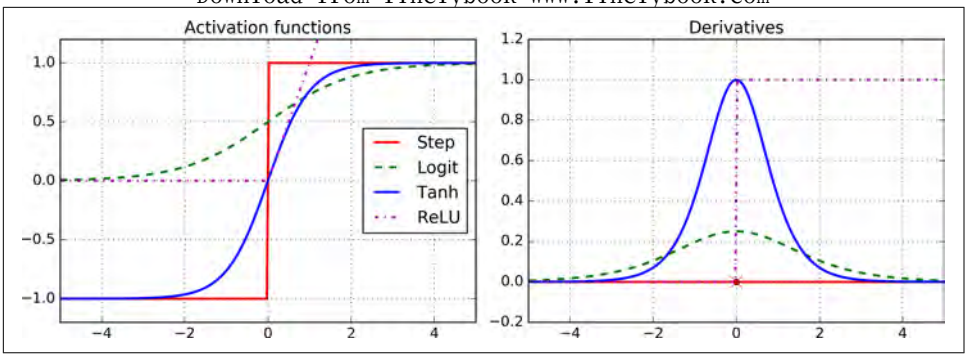


*Figure 10-8. Activation functions and their derivatives*

An MLP is often used for classification, with each output corresponding to a different binary class (e.g., spam/ham, urgent/not-urgent, and so on). When the classes are exclusive (e.g., classes 0 through 9 for digit image classification), the output layer is typically modified by replacing the individual activation functions by a shared *softmax* function (see Figure 10-9). The softmax function was introduced in Chapter 3. The output of each neuron corresponds to the estimated probability of the corresponding class. Note that the signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

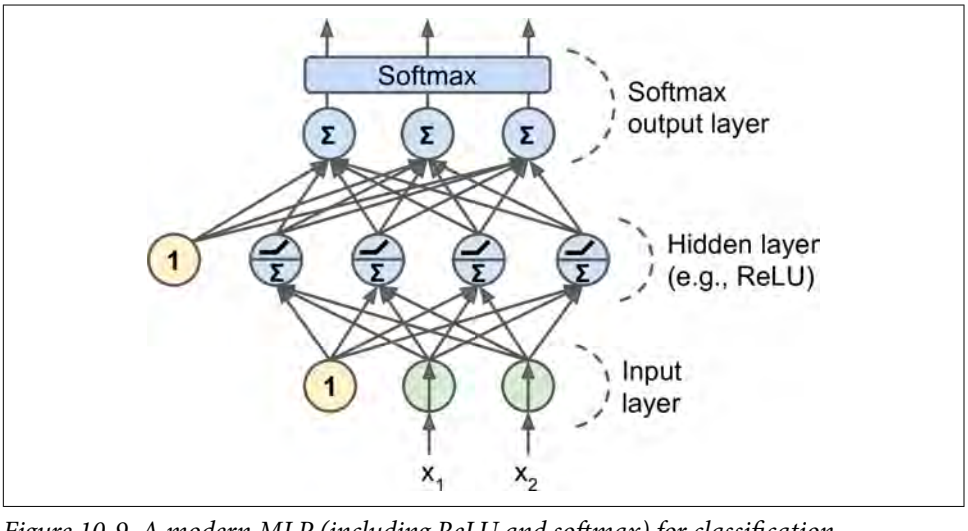


*Figure 10-9. A modern MLP (including ReLU and softmax) for classification*

Biological neurons seem to implement a roughly sigmoid (S-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that the ReLU activation function generally works better in ANNs. This is one of the cases where the biological analogy was misleading.

## Training an MLP with TensorFlow's High-Level API

The simplest way to train an MLP with TensorFlow is to use the high-level API TF.Learn, which is quite similar to Scikit-Learn's API. The `DNNClassifier` class makes it trivial to train a deep neural network with any number of hidden layers, and a softmax output layer to output estimated class probabilities. For example, the following code trains a DNN for classification with two hidden layers (one with 300 neurons, and the other with 100 neurons) and a softmax output layer with 10 neurons:

```
import tensorflow as tf

feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300, 100], n_classes=10,
                                         feature_columns=feature_columns)
dnn_clf.fit(x=X_train, y=y_train, batch_size=50, steps=40000)
```

If you run this code on the MNIST dataset (after scaling it, e.g., by using Scikit-Learn's `StandardScaler`), you may actually get a model that achieves over 98.1% accuracy on the test set! That's better than the best model we trained in Chapter 3:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = list(dnn_clf.predict(X_test))
>>> accuracy_score(y_test, y_pred)
0.98180000000000001
```

The TF.Learn library also provides some convenience functions to evaluate models:

```
>>> dnn_clf.evaluate(X_test, y_test)
{'accuracy': 0.98180002, 'global_step': 40000, 'loss': 0.073678359}
```

Under the hood, the `DNNClassifier` class creates all the neuron layers, based on the ReLU activation function (we can change this by setting the `activation_fn` hyperparameter). The output layer relies on the softmax function, and the cost function is cross entropy (introduced in Chapter 4).

The TF.Learn API is still quite new, so some of the names and functions used in these examples may evolve a bit by the time you read this book. However, the general ideas should not change.