SAIF

Shanghai Advanced
Institute of Finance
上海高级金融学院

量化俱乐部-深度学习-Introduction2ANN

2019-11-03

Menu

1. From Biological to Artificial Neurons
2. Training a DNN using Plain Tensorflow
3. Fine-Tuning Neural Network Hyperparameters

自我介绍

张骁喆，高金FMBA 2017。10年计算机工作经验，熟悉开发，测试，运维，项目管理，产品设计各环节。
曾就职eBay，唯品会，2016加入SAP，现担任SAP Cloud部门开发团队主管。
2017高金开始接触量化投资和python，目前毕业论文研究方向使用机器学习在A股进行量化策略研究。
量化投资策略咨询/人工智能咨询培训/金融科技项目管理产品管理。

Intro

Birds inspired us to fly, and nature has inspired many other inventions. It seems only logical, then, to look at the brain’s architecture for inspiration on how to build an intelligent machine. This is the key idea that inspired artificial neural networks (ANNs). However, although planes were inspired by birds, they don’t have to flap their wings. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying “units” rather than “neurons”), lest we restrict our creativity to biologically plausible systems.

ANNs are at the very core of Deep Learning. 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)

In this chapter, we will introduce artificial neural networks, starting with a quick tour of the very first ANN architectures. Then we will present Multi-Layer Perceptrons (MLPs) and implement one using TensorFlow to tackle the MNIST digit classification problem

From Biological to Artificial Neurons

Surprisingly, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. McCulloch and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic.

In 1906s, When it became clear that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere and ANNs entered a long dark era. But by the 1990s, powerful alternative Machine Learning techniques such as Support Vector Machines (see Chapter 5) were favored by most researchers. as they seemed to offer better results and stronger theoretical foundations.

From Biological to Artificial Neurons

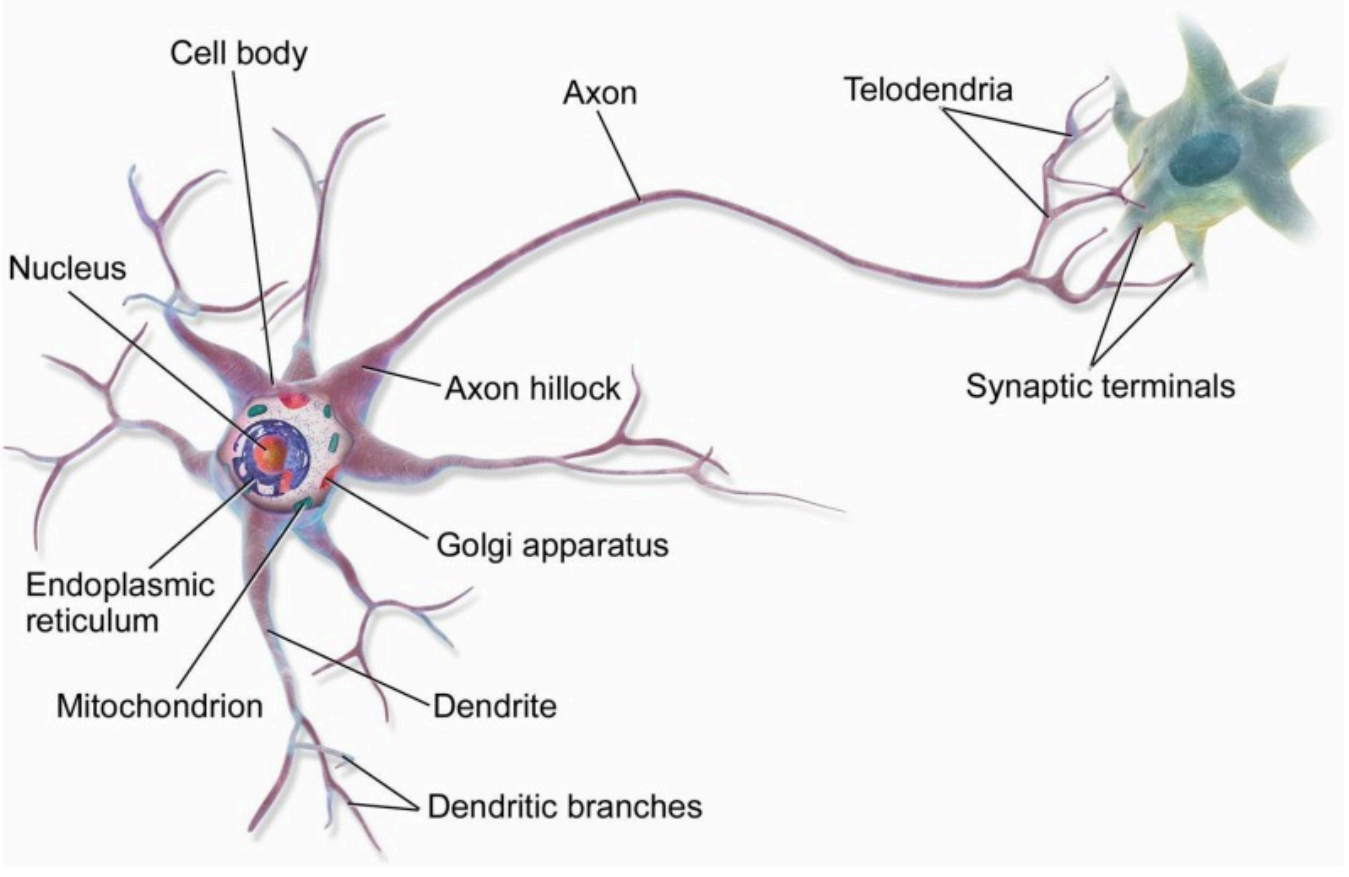
Finally, we are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? There are a few good reasons to believe that this one is different and will have a much more profound impact on our lives :

1. There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
2. The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore’ s Law, but also thanks to the gaming industry, which has produced powerful GPU cards by the millions.
3. The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have a huge positive impact.
4. Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice (or when it is the case, they are usually fairly close to the global optimum)

Biological Neurons

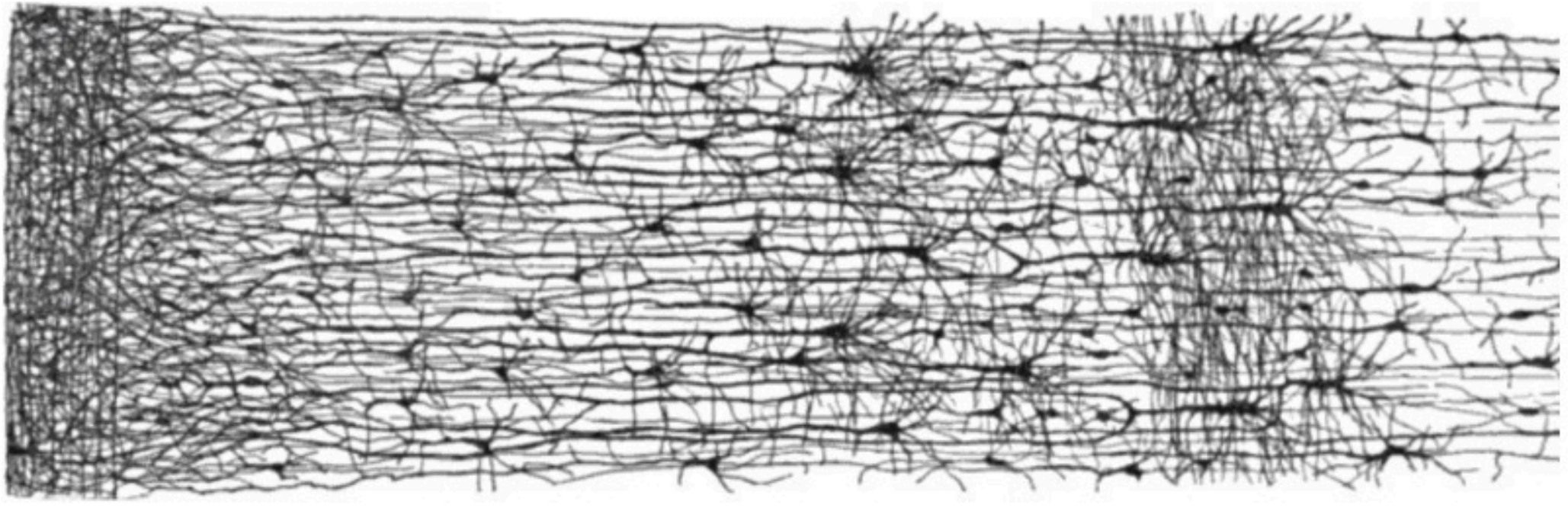
Before we discuss artificial neurons, let's take a quick look at a biological neuron. It is an unusual-looking cell mostly found in animal cerebral cortexes (e.g., your brain).

Biological neurons receive short electrical impulses called signals from other neurons via these synapses. When a neuron receives a sufficient number of signals from other neurons within a few milliseconds, it fires its own signals.



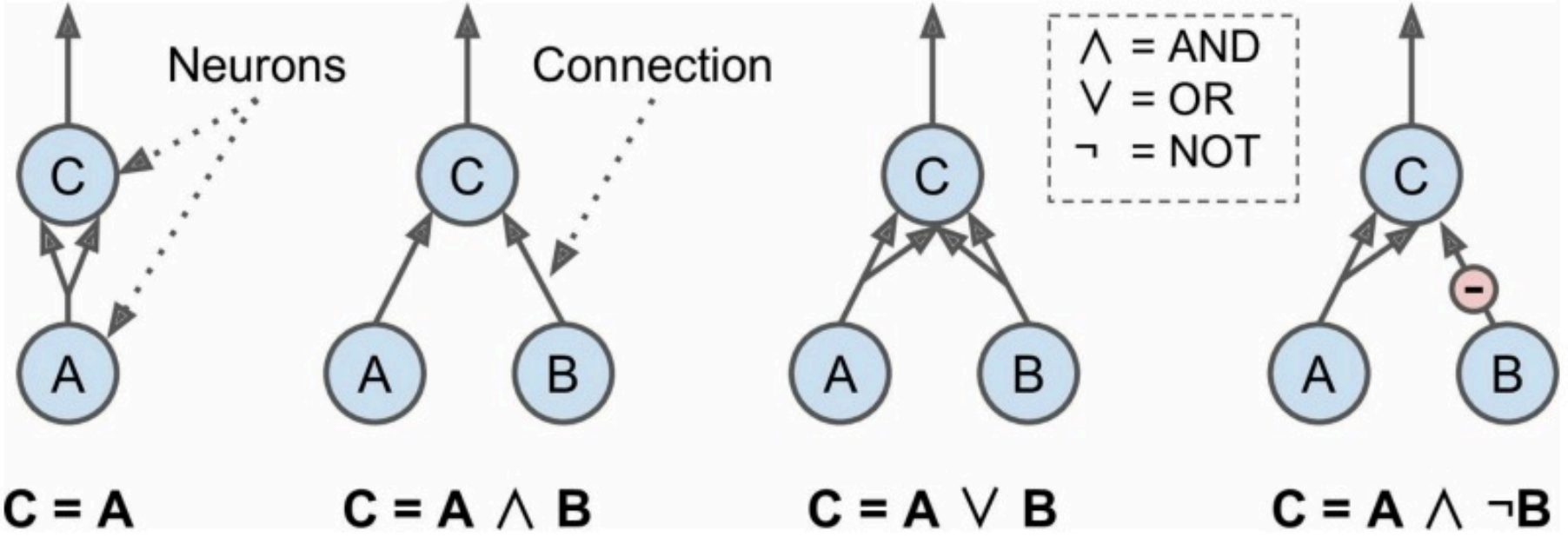
Biological Neurons

Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions of neurons, each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a vast network of fairly simple neurons.



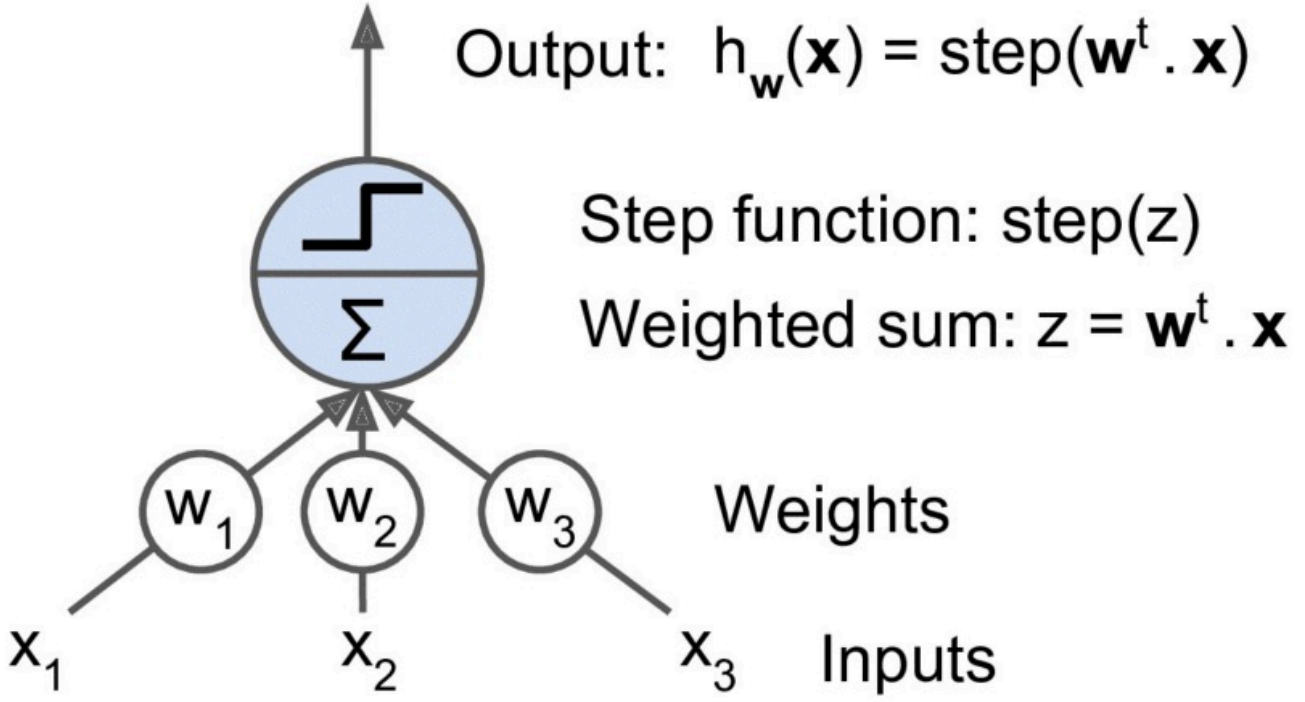
Logical Computations with Neurons

An artificial neuron: it has one or more binary (on/off) inputs and one binary output. The artificial neuron simply activates its output when more than a certain number of its inputs are active. McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want.



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 called a linear threshold unit (LTU): the inputs and output are now numbers (instead of binary on/off values) and each input connection is associated with a weight. The LTU computes a weighted sum of its inputs， then applies a step function to that sum and outputs the result.



The Perceptron

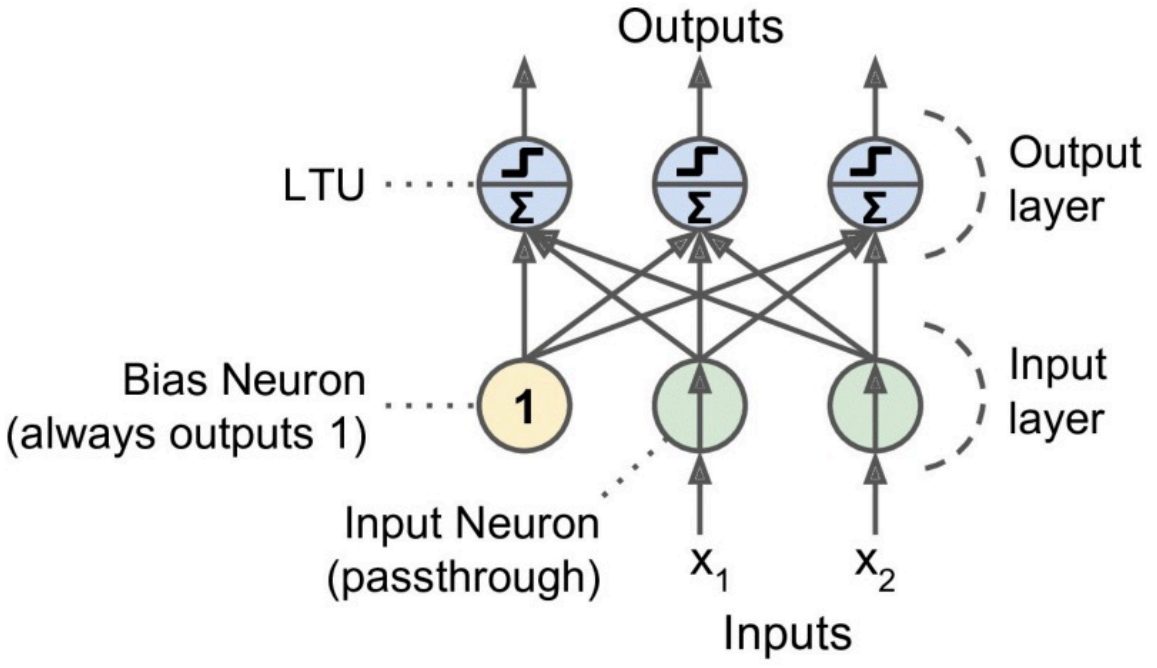
The most common step function used in Perceptrons is the Heaviside step function. Sometimes the sign function is used instead.

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

A single LTU can be used for simple linear binary classification. It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class. For example, you could use a single LTU to classify iris flowers based on the petal length and width (also adding an extra bias feature x0 = 1, just like we did in previous chapters). Training an LTU means finding the right values for w0, w1, and w2

The Perceptron

A Perceptron is simply composed of a single layer of LTUs, with each neuron connected to all the inputs. These connections are often represented using special passthrough neurons called input neurons: they just output whatever input they are fed. Moreover, an extra bias feature is generally added (x0 = 1). This bias feature is typically represented using a special type of neuron called a bias neuron, which just outputs 1 all the time.



The Perceptron

So how is a Perceptron trained? The Perceptron training algorithm proposed by Frank Rosenblatt was largely inspired by Hebb’ s rule. “Cells that fire together, wire together”

Perceptrons are trained using a variant of this rule that takes into account the error made by the network; it does not reinforce connections that lead to the wrong output. 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.

The Perceptron

Scikit-Learn provides a Perceptron class that implements a single LTU network. It can be used pretty much as you would expect — for example, on the iris dataset.

In [2]:
```
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)

per_clf = Perceptron(max_iter=100, tol=-np.infty, random_state=42)
per_clf.fit(X, y)

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

In [3]:
```
y_pred
```

Out[3]:
```
array([1])
```

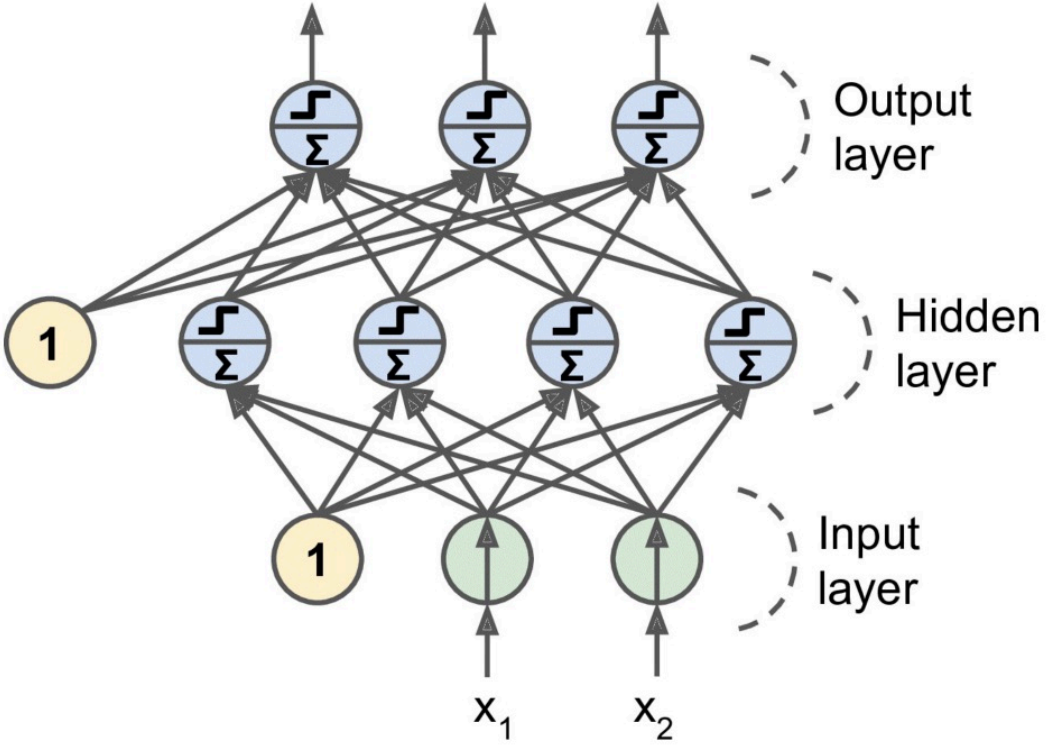
The Perceptron

You may have recognized 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", eta0=1 (the learning rate), and penalty=None (no regularization)

Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they just make predictions based on a hard threshold. This is one of the good reasons to prefer Logistic Regression over Perceptrons.

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. 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)



Multi-Layer Perceptron and Backpropagation

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 introducing the backpropagation training algorithm. 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 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)

Multi-Layer Perceptron – Activation Function

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, $\sigma(z) = 1 / (1 + \exp(-z))$. Logistic function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step. The backpropagation algorithm may be usedwith other activation functions, instead of the logistic function. Two other popular activation functions are:

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

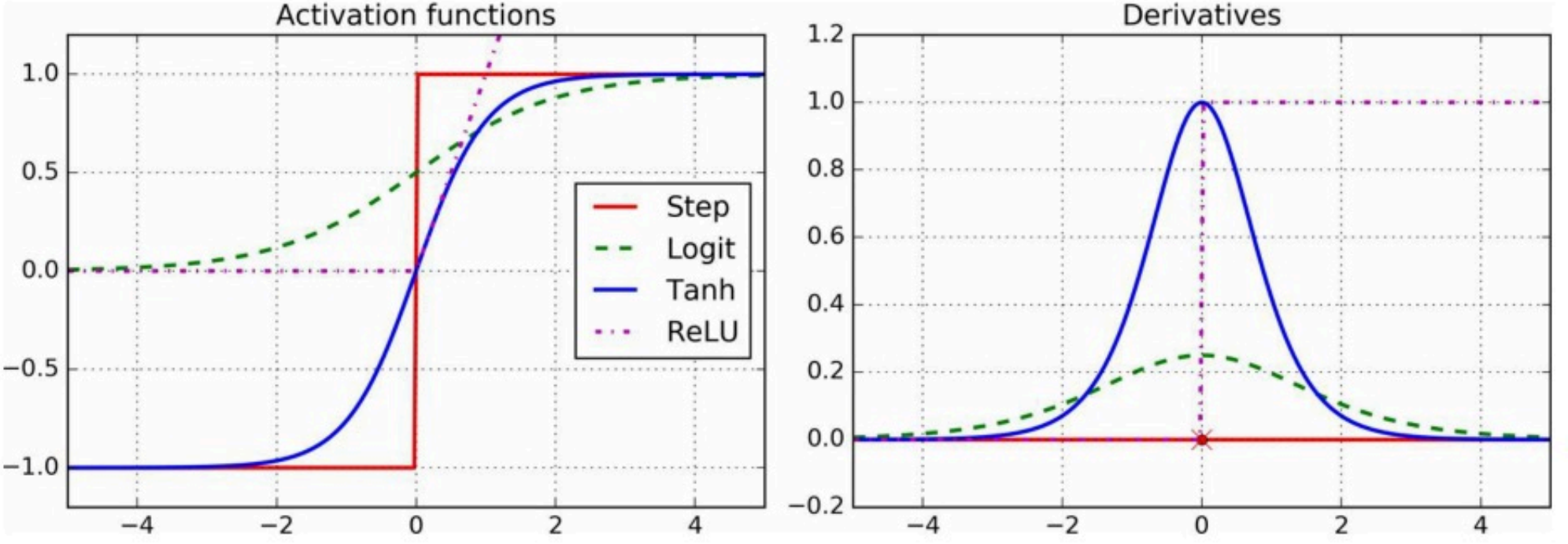
Just like the logistic function it is S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1) in the case of the logistic function), which tends to make each layer’ s output more or less normalized (i.e., centered around 0) at the beginning of training.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

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

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.

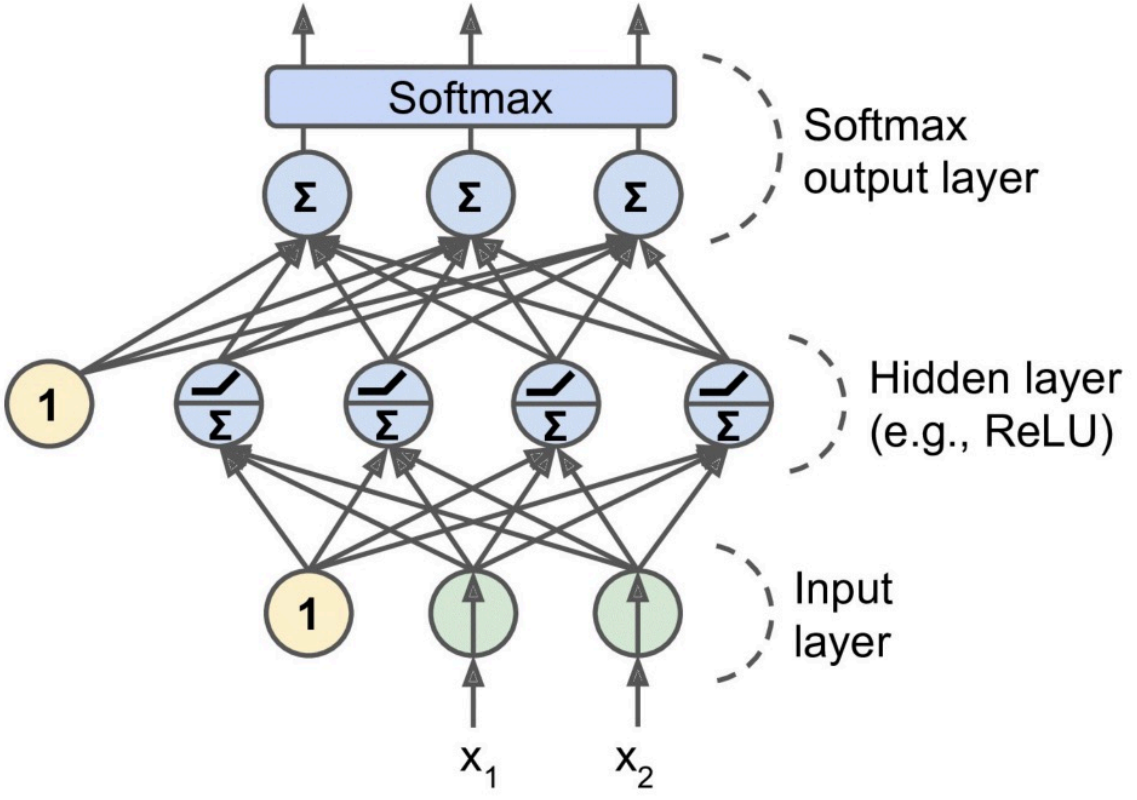
Multi-Layer Perceptron – Activation Function

These popular activation functions and their derivatives are represented:



Multi-Layer Perceptron – Activation Function

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**. The output of each neuron corresponds to the estimated probability of the corresponding class.



Training an MLP with TensorFlow's High-Level

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:

In [9]:
```
import tensorflow as tf
```

In [12]:
```
feature_cols = [tf.feature_column.numeric_column("X", shape=[28 * 28])]
dnn_clf = tf.estimator.DNNClassifier(hidden_units=[300,100], n_classes=10,
                                     feature_columns=feature_cols)

input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"X": X_train}, y=y_train, num_epochs=40, batch_size=50, shuffle=True)
dnn_clf.train(input_fn=input_fn)
```

Training an MLP with TensorFlow's High-Level

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 97.% accuracy on the test set! That's better than the best model we trained earlier. The TF.Learn library also provides some convenience functions to evaluate models:

```
In [13]: test_input_fn = tf.estimator.inputs.numpy_input_fn(
             x={"X": X_test}, y=y_test, shuffle=False)
         eval_results = dnn_clf.evaluate(input_fn=test_input_fn)
```

```
In [14]: eval_results

Out[14]: {'accuracy': 0.9795,
          'average_loss': 0.10352725,
          'loss': 13.104714,
          'global_step': 44000}
```

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).

Training a DNN Using Plain TensorFlow

If you want more control over the architecture of the network, you may prefer to use TensorFlow’ s lower-level Python API (introduced in Chapter 9). In this section we will build the same model as before using this API, and we will implement Mini-batch Gradient Descent to train it on the MNIST dataset. The first step is the construction phase, building the TensorFlow graph. The second step is the execution phase, where you actually run the graph to train the model.

Plain TensorFlow – Construction Phase

First we need to import the tensorflow library. Then we must specify the number of inputs and outputs, and set the number of hidden neurons in each layer:

```
In [15]: import tensorflow as tf

         n_inputs = 28*28  # MNIST
         n_hidden1 = 300
         n_hidden2 = 100
         n_outputs = 10
```

Plain TensorFlow – Construction Phase

Next, just like we did in Chapter 9, you can use placeholder nodes to represent the training data and targets. The shape of X is only partially defined. We know that it will be a 2D tensor (i.e., a matrix), with instances along the first dimension and features along the second dimension, and we know that the number of features is going to be 28 x 28 (one feature per pixel), but we don't know yet how many instances each training batch will contain. So the shape of X is (None, n_inputs). Similarly, we know that y will be a 1D tensor with one entry per instance, but again we don't know the size of the training batch at this point, so the shape is (None)

```
In [*]: reset_graph()

        X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
        y = tf.placeholder(tf.int32, shape=(None), name="y")
```

Plain TensorFlow – Construction Phase

Now let's create the actual neural network. The placeholder X will act as the input layer; during the execution phase, it will be replaced with one training batch at a time (note that all the instances in a training batch will be processed simultaneously by the neural network). Now you need to create the two hidden layers and the output layer. The two hidden layers are almost identical: they differ only by the inputs they are connected to and by the number of neurons they contain. The output layer is also very similar, but it uses a softmax activation function instead of a ReLU activation function. So let's create a neuron_layer() function that we will use to create one layer at a time. It will need parameters to specify the inputs, the number of neurons, the activation function, and the name of the layer:

In [17]:
```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

Plain TensorFlow – Construction Phase

Okay, so now you have a nice function to create a neuron layer. Let's use it to create the deep neural network! The first hidden layer takes X as its input. The second takes the output of the first hidden layer as its input. And finally, the output layer takes the output of the second hidden layer as its input.

In [18]:
```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1",
                           activation=tf.nn.relu)
    hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2",
                           activation=tf.nn.relu)
    logits = neuron_layer(hidden2, n_outputs, name="outputs")
```

Notice that once again we used a name scope for clarity. Also note that logits is the output of the neural network before going through the softmax activation function: for optimization reasons, we will handle the softmax computation later.

Plain TensorFlow – Construction Phase

As you might expect, TensorFlow comes with many handy functions to create standard neural network layers, so there's often no need to define your own neuron_layer() function like we just did. For example, TensorFlow's fully_connected() function creates a fully connected layer, where all the inputs are connected to all the neurons in the layer. It takes care of creating the weights and biases variables, with the proper initialization strategy, and it uses the ReLU activation function by default (we can change this using the activation_fn argument). As we will see in Chapter 11, it also supports regularization and normalization parameters. Let's tweak the preceding code to use the fully_connected() function instead of our neuron_layer() function. Simply import the function and replace the dnn construction section with the following code:

```
In [ ]: with tf.name_scope("dnn"):
            hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                                      activation=tf.nn.relu)
            hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                                      activation=tf.nn.relu)
            logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
            y_proba = tf.nn.softmax(logits)
```

Plain TensorFlow – Construction Phase

Now that we have the neural network model ready to go, we need to define the cost function that we will use to train it. Just as we did for Softmax Regression in Chapter 4, we will use cross entropy. As we discussed earlier, cross entropy will penalize models that estimate a low probability for the target class. TensorFlow provides several functions to compute cross entropy. We will use sparse_softmax_cross_entropy_with_logits(): it computes the cross entropy based on the "logits" (i.e., the output of the network before going through the softmax activation function), and it expects labels in the form of integers ranging from 0 to the number of classes minus 1 (in our case, from 0 to 9). This will give us a 1D tensor containing the cross entropy for each instance. We can then use TensorFlow's reduce_mean() function to compute the mean cross entropy over all instances.

In [31]:
```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

Plain TensorFlow – Construction Phase

We have the neural network model, we have the cost function, and now we need to define a GradientDescentOptimizer that will tweak the mode parameters to minimize the cost function. Nothing new; it’ s just like we did in Chapter 9:

In [32]:
```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

Plain TensorFlow – Construction Phase

The last important step in the construction phase is to specify how to evaluate the model. We will simply use accuracy as our performance measure. First, for each instance, determine if the neural network's prediction is correct by checking whether or not the highest logit corresponds to the target class. For this you can use the in_top_k() function. This returns a 1D tensor full of boolean values, so we need to cast these booleans to floats and then compute the average. This will give us the network's overall accuracy:

```
In [33]: with tf.name_scope("eval"):
             correct = tf.nn.in_top_k(logits, y, 1)
             accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Plain TensorFlow – Construction Phase

And, as usual, we need to create a node to initialize all variables, and we will also create a Saver to save our trained model parameters to disk:

In [34]:
```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

***This concludes the construction phase. This was fewer than 40 lines of code, but it was pretty intense: we created placeholders for the inputs and the targets, we created a function to build a neuron layer, we used it to create the DNN, we defined the cost function, we created an optimizer, and finally we defined the performance measure. Now on to the execution phase.*

Plain TensorFlow – Execution Phase

This code opens a TensorFlow session, and it runs the init node that initializes all the variables. Then it runs the main training loop: at each epoch, the code iterates through a number of mini-batches that corresponds to the training set size. Each mini-batch is fetched via the next_batch() method, and then the code simply runs the training operation, feeding it the current mini-batch input data and targets. Next, at the end of each epoch, the code evaluates the model on the last mini-batch and on the full training set, and it prints out the result. Finally, the model parameters are saved to disk.

In [25]:
```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for X_batch, y_batch in shuffle_batch(X_train, y_train, batch_size):
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_batch = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_val = accuracy.eval(feed_dict={X: X_valid, y: y_valid})
        print(epoch, "Batch accuracy:", acc_batch, "Val accuracy:", acc_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Using the Neural Network

Now that the neural network is trained, you can use it to make predictions.

First the code loads the model parameters from disk. Then it loads some new images that you want to classify. Remember to apply the same feature scaling as for the training data (in this case, scale it from 0 to 1). Then the code evaluates the logits node. If you wanted to know all the estimated class probabilities, you would need to apply the softmax() function to the logits, but if you just want to predict a class, you can simply pick the class that has the highest logit value (using the argmax() function does the trick)

In [26]:
```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt") # or better, use save_path
    X_new_scaled = X_test[:20]
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

```
INFO:tensorflow:Restoring parameters from ./my_model_final.ckpt
```

In [27]:
```
print("Predicted classes:", y_pred)
print("Actual classes:   ", y_test[:20])
```

```
Predicted classes: [7 2 1 0 4 1 4 9 6 9 0 6 9 0 1 5 9 7 3 4]
Actual classes:    [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4]
```

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network topology (how neurons are interconnected), but even in a simple MLP you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initialization logic, and much more. How do you know what combination of hyperparameters is the best for your task?

It helps to have an idea of what values are reasonable for each hyperparameter, so you can restrict the search space. Let’ s start with the number of hidden layers.

Number of Hidden Layers

For many problems, you can just begin with a single hidden layer and you will get reasonable results. It has actually been shown that an MLP with just one hidden layer can model even the most complex functions provided it has enough neurons. For a long time, these facts convinced researchers that there was no need to investigate any deeper neural networks. But they overlooked the fact that deep networks have a much higher parameter efficiency than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, making them much faster to train.

Not only does this hierarchical architecture help DNNs converge faster to a good solution, it also improves their ability to generalize to new datasets.

In summary, for many problems you can start with just one or two hidden layers and it will work just fine. For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers (or even hundreds, but not fully connected ones, as we will see in Chapter 13), and they need a huge amount of training data. However, you will rarely have to train such networks from scratch. it is much more common to reuse parts of a pretrained one.

Number of Neurons per Hidden Layer

Obviously the number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, the MNIST task requires 28 x 28 = 784 input neurons and 10 output neurons. As for the hidden layers, a common practice is to size them to form a funnel, with fewer and fewer neurons at each layer.

For example, a typical neural network for MNIST may have two hidden layers, the first with 300 neurons and the second with 100.

In general you will get more bang for the buck by increasing the number of layers than the number of neurons per layer.

A simpler approach is to pick a model with more layers and neurons than you actually need, then use early stopping to prevent it from overfitting (and other regularization techniques, especially dropout, as we will see in Chapter 11).

Activation Functions

In most cases you can use the ReLU activation function in the hidden layers (or one of its variants, as we will see in Chapter 11). It is a bit faster to compute than other activation functions, and Gradient Descent does not get stuck as much on plateaus, thanks to the fact that it does not saturate for large input values (as opposed to the logistic function or the hyperbolic tangent function, which saturate at 1).

For the output layer, the softmax activation function is generally a good choice for classification tasks (when the classes are mutually exclusive). For regression tasks, you can simply use no activation function at all.

SAIF

Shanghai Advanced
Institute of Finance
上海高级金融学院