NVIDIA. DEEP LEARNING INSTITUTE

LEARNING DEEP LEARNING

Theory and Practice of Neural Networks, Computer Vision, Natural Language Processing, and Transformers Using TensorFlow

MAGNUS EKMAN

Contents

Chapter 5

Toward DL: Frameworks and Network Tweaks

An obvious next step would be to see if adding more layers to our neural networks results in even better accuracy. However, it turns out getting deeper networks to learn well is a major obstacle. A number of innovations were needed to overcome these obstacles and enable deep learning (DL). We introduce the most important ones later in this chapter, but before doing so, we explain how to use a DL framework. The benefit of using a DL framework is that we do not need to implement all these new techniques from scratch in our neural network. The downside is that you will not deal with the details in as much depth as in previous chapters. You now have a solid enough foundation to build on. Now we switch gears a little and focus on the big picture of solving real-world problems using a DL framework. The emergence of DL frameworks played a significant role in making DL practical to adopt in the industry as well as in boosting productivity of academic research.

Programming Example: Moving to a DL Framework

In this programming example, we show how to implement the handwritten digit classification from Chapter 4, "Fully Connected Networks Applied to Multiclass Classification," using a DL framework. In this book, we have chosen to use the two frameworks TensorFlow and PyTorch. Both of these frameworks are popular and flexible. The TensorFlow versions of the code examples are interspersed throughout the book, and the PyTorch versions are available online on the book Web site.

TensorFlow provides a number of different constructs and enables you to work at different abstraction levels using different application programming interfaces (APIs). In general, to keep things simple, you want to do your work at the highest abstraction level possible because that means that you do not need to implement the low-level details. For the examples we will study, the Keras API is a suitable abstraction level. Keras started as a stand-alone library. It was not tied to TensorFlow and could be used with multiple DL frameworks. However, at this point, Keras is fully supported inside of TensorFlow itself. See Appendix I for information about how to install TensorFlow and what version to use.

Appendix I also contains information about how to install PyTorch if that is your framework of choice. Almost all programming constructs in this book exist both in TensorFlow and in PyTorch. The section "Key Differences between PyTorch and TensorFlow" in Appendix I describes some key differences between the two frameworks. You will find it helpful if you do not want to pick a single framework but want to master both of them.

The frameworks are implemented as Python libraries. That is, we still write our program as a Python program and we just import the framework of choice as a library. We can then use DL functions from the famework in our program. The initialization code for our TensorFlow example is shown in Code Snippet 5-1.

Code Snippet 5-1 Import Statements for Our TensorFlow/Keras Example

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical
import numpy as np
import logging
```

```
tf.get_logger().setLevel(logging.ERROR)
tf.random.set_seed(7)

EPOCHS = 20
BATCH_SIZE = 1
```

As you can see in the code, TensorFlow has its own random seed that needs to be set if we want reproducible results. However, this still does not guarantee that repeated runs produce identical results for all types of networks, so for the remainder of this book, we will not worry about setting the random seeds. The preceding code snippet also sets the logging level to only print out errors while suppressing warnings.

We then load and prepare our MNIST dataset. Because MNIST is a common dataset, it is included in Keras. We can access it by a call to `keras.datasets.mnist` and `load_data`. The variables `train_images` and `test_images` will contain the input values, and the variables `train_labels` and `test_labels` will contain the ground truth (Code Snippet 5-2).

Code Snippet 5-2 Load and Prepare the Training and Test Datasets

```
# Load training and test datasets.
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images,
                               test_labels) = mnist.load_data()

# Standardize the data.
mean = np.mean(train_images)
stddev = np.std(train_images)
train_images = (train_images - mean) / stddev
test_images = (test_images - mean) / stddev

# One-hot encode labels.
train_labels = to_categorical(train_labels, num_classes=10)
test_labels = to_categorical(test_labels, num_classes=10)
```

Just as before, we need to standardize the input data and one-hot encode the labels. We use the function `to_categorical` to one-hot encode our labels

instead of doing it manually, as we did in our previous example. This serves as an example of how the framework provides functionality to simplify our implementation of common tasks.

If you are not so familiar with Python, it is worth pointing out that functions can be defined with optional arguments, and to avoid having to pass the arguments in a specific order, optional arguments can be passed by first naming which argument we are trying to set. An example is the **num_classes** argument in the **to_categorical** function.

We are now ready to create our network. There is no need to define variables for individual neurons because the framework provides functionality to instantiate entire layers of neurons at once. We do need to decide how to initialize the weights, which we do by creating an initializer object, as shown in Code Snippet 5-3. This might seem somewhat convoluted but will come in handy when we want to experiment with different initialization values.

Code Snippet 5-3 Create the Network

```
# Object used to initialize weights.
initializer = keras.initializers.RandomUniform(
    minval=-0.1, maxval=0.1)

# Create a Sequential model.
# 784 inputs.
# Two Dense (fully connected) layers with 25 and 10 neurons.
# tanh as activation function for hidden layer.
# Logistic (sigmoid) as activation function for output layer.
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(25, activation='tanh',
                       kernel_initializer=initializer,
                       bias_initializer='zeros'),
    keras.layers.Dense(10, activation='sigmoid',
                       kernel_initializer=initializer,
                       bias_initializer='zeros')])
```

The network is created by instantiating a `keras.Sequential` object, which implies that we are using the Keras Sequential API. (This is the simplest API, and we use it for the next few chapters until we start creating networks that require a more advanced API.) We pass a list of layers as an argument to the `Sequential` class. The first layer is a `Flatten` layer, which does not do computations but only changes the organization of the input. In our case, the inputs are changed from a 28×28 array into an array of 784 elements. If the data had already been organized into a 1D-array, we could have skipped the `Flatten` layer and simply declared the two `Dense` layers. If we had done it that way, then we would have needed to pass an `input_shape` parameter to the first `Dense` layer because we always have to declare the size of the inputs to the first layer in the network.

The second and third layers are both `Dense` layers, which means they are fully connected. The first argument tells how many neurons each layer should have, and the `activation` argument tells the type of activation function; we choose `tanh` and `sigmoid`, where `sigmoid` means the *logistic sigmoid function*. We pass our `initializer` object to initialize the regular weights using the `kernel_initializer` argument. The bias weights are initialized to 0 using the `bias_initializer` argument.

One thing that might seem odd is that we are not saying anything about the number of inputs and outputs for the second and third layers. If you think about it, the number of inputs is fully defined by saying that both layers are fully connected and the fact that we have specified the number of neurons in each layer along with the number of inputs to the first layer of the network. This discussion highlights that using the DL framework enables us to work at a higher abstraction level. In particular, we use layers instead of individual neurons as building blocks, and we need not worry about the details of how individual neurons are connected to each other. This is often reflected in our figures as well, where we work with individual neurons only when we need to explain alternative network topologies. On that note, Figure 5-1 illustrates our digit recognition network at this higher abstraction level. We use rectangular boxes with rounded corners to depict a layer of neurons, as opposed to circles that represent individual neurons.

We are now ready to train the network, which is done by Code Snippet 5-4. We first create a `keras.optimizer.SGD` object. This means that we want to use stochastic gradient descent (SGD) when training the network. Just as with the initializer, this might seem somewhat convoluted, but it provides flexibility to adjust parameters for the learning process, which we explore soon. For now, we just set the learning rate to 0.01 to match what we did in our plain Python example. We then prepare the model for training by calling the model's `compile`

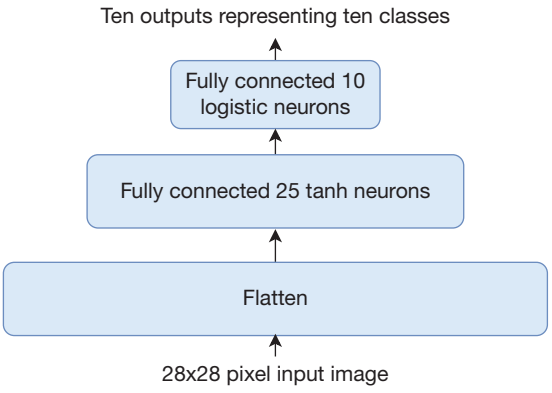

Figure 5-1 Digit classification network using layers as building blocks

function. We provide parameters to specify which `loss` function to use (where we use `mean_squared_error` as before), the optimizer that we just created and that we are interested in looking at the `accuracy` metric during training.

Code Snippet 5-4 Train the Network

```
# Use stochastic gradient descent (SGD) with
# learning rate of 0.01 and no other bells and whistles.
# MSE as loss function and report accuracy during training.
opt = keras.optimizers.SGD(learning_rate=0.01)

model.compile(loss='mean_squared_error', optimizer = opt,
              metrics =['accuracy'])

# Train the model for 20 epochs.
# Shuffle (randomize) order.
# Update weights after each example (batch_size=1).
history = model.fit(train_images, train_labels,
                    validation_data=(test_images, test_labels),
                    epochs=EPOCHS, batch_size=BATCH_SIZE,
                    verbose=2, shuffle=True)
```

We finally call the `fit` function for the model, which starts the training process. As the function name indicates, it fits the model to the data. The first two arguments specify the training dataset. The parameter `validation_data` is

the test dataset. Our variables EPOCHS and BATCH_SIZE from the initialization code determine how many epochs to train for and what batch size we use. We had set BATCH_SIZE to 1, which means that we update the weight after a single training example, as we did in our plain Python example. We set `verbose=2` to get a reasonable amount of information printed during the training process and set `shuffle` to `True` to indicate that we want the order of the training data to be randomized during the training process. All in all, these parameters match what we did in our plain Python example.

Depending on what TensorFlow version you run, you might get a fair number of printouts about opening libraries, detecting the graphics processing unit (GPU), and other issues as the program starts. If you want it less verbose, you can set the environment variable TF_CPP_MIN_LOG_LEVEL to 2. If you are using bash, you can do that with the following command line:

```
export TF_CPP_MIN_LOG_LEVEL=2
```

Another option is to add the following code snippet at the top of your program.

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
```

The printouts for the first few training epochs are shown here. We stripped out some timestamps to make it more readable.

```
Epoch 1/20

loss: 0.0535 - acc: 0.6624 - val_loss: 0.0276 - val_acc: 0.8893

Epoch 2/20

loss: 0.0216 - acc: 0.8997 - val_loss: 0.0172 - val_acc: 0.9132

Epoch 3/20

loss: 0.0162 - acc: 0.9155 - val_loss: 0.0145 - val_acc: 0.9249

Epoch 4/20

loss: 0.0142 - acc: 0.9227 - val_loss: 0.0131 - val_acc: 0.9307
```

```
Epoch 5/20

loss: 0.0131 - acc: 0.9274 - val_loss: 0.0125 - val_acc: 0.9309

Epoch 6/20

loss: 0.0123 - acc: 0.9313 - val_loss: 0.0121 - val_acc: 0.9329
```

In the printouts, `loss` represents the mean squared error (MSE) of the training data, `acc` represents the prediction accuracy on the training data, `val_loss` represents the MSE of the test data, and `val_acc` represents the prediction accuracy of the test data. It is worth noting that we do not get exactly the same learning behavior as was observed in our plain Python model. It is hard to know why without diving into the details of how TensorFlow is implemented. Most likely, it could be subtle issues related to how initial parameters are randomized and the random order in which training examples are picked. Another thing worth noting is how simple it was to implement our digit classification application using TensorFlow. Using the TensorFlow framework enables us to study more advanced techniques while still keeping the code size at a manageable level.

We now move on to describing some techniques needed to enable learning in deeper networks. After that, we can finally do our first DL experiment in the next chapter.

The Problem of Saturated Neurons and Vanishing Gradients

In our experiments, we made some seemingly arbitrary changes to the learning rate parameter as well as to the range with which we initialized the weights. For our perceptron learning example and the XOR network, we used a learning rate of 0.1, and for the digit classification, we used 0.01. Similarly, for the weights, we used the range –1.0 to +1.0 for the XOR example, whereas we used –0.1 to +0.1 for the digit example. A reasonable question is whether there is some method to the madness. Our dirty little secret is that we changed the values simply because our networks did not learn well without these changes. In this section, we discuss the reasons for this and explore some guidelines that can be used when selecting these seemingly random parameters.

To understand why it is sometimes challenging to get networks to learn, we need to look in more detail at our activation function. Figure 5-2 shows our two S-shaped functions. It is the same chart that we showed in Figure 3-4 in Chapter 3, "Sigmoid Neurons and Backpropagation."

One thing to note is that both functions are uninteresting outside of the shown *z*-interval (which is why we showed only this *z*-interval in the first place). Both functions are more or less straight horizontal lines outside of this range.

Now consider how our learning process works. We compute the derivative of the error function and use that to determine which weights to adjust and in what direction. Intuitively, what we do is tweak the input to the activation function (*z* in the chart in Fig. 5-2) slightly and see if it affects the output. If the *z*-value is within the small range shown in the chart, then this will change the output (the *y*-value in the chart). Now consider the case when the *z*-value is a large positive or negative number. Changing the input by a small amount (or even a large amount) will not affect the output because the output is a horizontal line in those regions. We say that the neuron is *saturated*.

Saturated neurons can cause learning to stop completely. As you remember, when we compute the gradient with the backpropagation algorithm, we propagate the error backward through the network, and part of that process is to multiply the derivative of the loss function by the derivative of the activation function. Consider

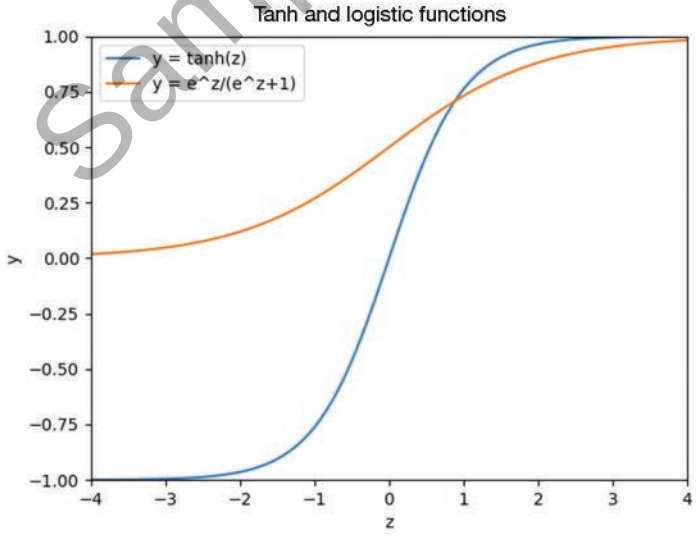

Figure 5-2 The two S-shaped functions tanh and logistic sigmoid

what the derivatives of the two activation functions above are for *z*-values of significant magnitude (positive or negative). The derivative is 0! In other words, no error will propagate backward, and no adjustments will be done to the weights. Similarly, even if the neuron is not fully saturated, the derivative is less than 0. Doing a series of multiplications (one per layer) where each number is less than 0 results in the gradient approaching 0. This problem is known as the *vanishing gradient problem*. Saturated neurons are not the only reason for vanishing gradients, as we will see later in the book.

Saturated neurons are insensitive to input changes because their derivative is 0 in the saturated region. This is one cause of the **vanishing gradient** problem where the backpropagated error is 0 and the weights are not adjusted.

Initialization and Normalization Techniques to Avoid Saturated Neurons

We now explore how we can prevent or address the problem of saturated neurons. Three techniques that are commonly used—and often combined—are weight initialization, input standardization, and batch normalization.

WEIGHT INITIALIZATION

The first step in avoiding saturated neurons is to ensure that our neurons are not saturated to begin with, and this is where weight initialization is important. It is worth noting that, although we use the same type of neurons in our different examples, the actual parameters for the neurons that we have shown are much different. In the XOR example, the neurons in the hidden layer had three inputs including the bias, whereas for the digit classification example, the neurons in the hidden layer had 785 inputs. With that many inputs, it is not hard to imagine that the weighted sum can swing far in either the negative or positive direction if there is just a little imbalance in the number of negative versus positive inputs if the weights are large. From that perspective, it kind of makes sense that if a neuron has a large number of inputs, then we want to initialize the weights to a smaller value to have a reasonable probability of still keeping the input to the activation function close to 0 to avoid saturation. Two popular weight initialization strategies are Glorot initialization (Glorot and Bengio, 2010) and He initialization (He et al., 2015b). Glorot initialization is recommended for tanh- and

sigmoid-based neurons, and He initialization is recommended for ReLU-based neurons (described later). Both of these take the number of inputs into account, and Glorot initialization also takes the number of outputs into account. Both Glorot and He initialization exist in two flavors, one that is based on a uniform random distribution and one that is based on a normal random distribution.

> We do not go into the formulas for **Glorot** and **He initialization**, but they are good topics well worth considering for further reading (Glorot and Bengio, 2010; He et al., 2015b).

We have previously seen how we can initialize the weights from a uniform random distribution in TensorFlow by using an initializer, as was done in Code Snippet 5-4. We can choose a different initializer by declaring any one of the supported initializers in Keras. In particular, we can declare a Glorot and a He initializer in the following way:

```
initializer = keras.initializers.glorot_uniform()
initializer = keras.initializers.he_normal()
```

Parameters to control these initializers can be passed to the initializer constructor. In addition, both the Glorot and He initializers come in the two flavors `uniform` and `normal`. We picked uniform for Glorot and normal for He because that is what was described in the publications where they were introduced.

If you do not feel the need to tweak any of the parameters, then there is no need to declare an initializer object at all, but you can just pass the name of the initializer as a string to the function where you create the layer. This is shown in Code Snippet 5-5, where the `kernel_initializer` argument is set to `'glorot_uniform'`.

Code Snippet 5-5 Setting an Initializer by Passing Its Name as a String

```
model = keras.Sequential([
        keras.layers.Flatten(input_shape=(28, 28)),
        keras.layers.Dense(25, activation='tanh',
                           kernel_initializer='glorot_uniform',
                           bias_initializer='zeros'),
        keras.layers.Dense(10, activation='sigmoid',
                           kernel_initializer='glorot_uniform',
                           bias_initializer='zeros')])
```

We can separately set `bias_initializer` to any suitable initializer, but as previously stated, a good starting recommendation is to just initialize the bias weights to 0, which is what the `'zeros'` initializer does.

INPUT STANDARDIZATION

In addition to initializing the weights properly, it is important to preprocess the input data. In particular, standardizing the input data to be centered around 0 and with most values close to 0 will reduce the risk of saturating neurons from the start. We have already used this in our implementation; let us discuss it in a little bit more detail. As stated earlier, each pixel in the MNIST dataset is represented by an integer between 0 and 255, where 0 represents the blank paper and a higher value represents pixels where the digit was written.[1] Most of the pixels will be either 0 or a value close to 255, where only the edges of the digits are somewhere in between. Further, a majority of the pixels will be 0 because a digit is sparse and does not cover the entire 28×28 image. If we compute the average pixel value for the entire dataset, then it turns out that it is about 33. Clearly, if we used the raw pixel values as inputs to our neurons, then there would be a big risk that the neurons would be far into the saturation region. By subtracting the mean and dividing by the standard deviation, we ensure that the neurons get presented with input data that is in the region that does not lead to saturation.

BATCH NORMALIZATION

Normalizing the inputs does not necessarily prevent saturation of neurons for hidden layers, and to address that problem Ioffe and Szegedy (2015) introduced batch normalization. The idea is to normalize values inside of the network as well and thereby prevent hidden neurons from becoming saturated. This may sound somewhat counterintuitive. If we normalize the output of a neuron, does that not result in undoing the work of that neuron? That would be the case if it truly was just normalizing the values, but the batch normalization function also contains parameters to counteract this effect. These parameters are adjusted during the learning process. Noteworthy is that after the initial idea was published, subsequent work indicated that the reason batch normalization works is different than the initial explanation (Santurkar et al., 2018).

> Batch normalization (Ioffe and Szegedy, 2015) is a good topic for further reading.

1. This might seem odd because a value of 0 typically represents black and a value of 255 typically represents white for a grayscale image. However, that is not the case for this dataset.

There are two main ways to apply batch normalization. In the original paper, the suggestion was to apply the normalization on the input to the activation function (after the weighted sum). This is shown to the left in Figure 5-3.

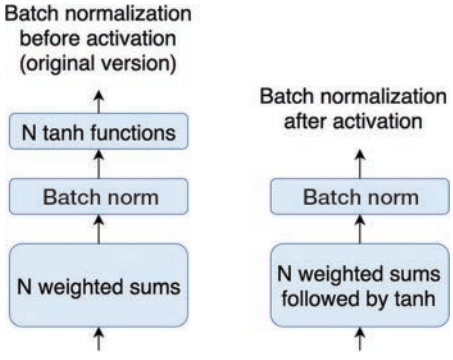

Figure 5-3 Left: Batch normalization as presented by Ioffe and Szegedy (2015). The layer of neurons is broken up into two parts. The first part is the weighted sums for all neurons. Batch normalization is applied to these weighted sums. The activation function (tanh) is applied to the output of the batch normalization operation. Right: Batch normalization is applied to the output of the activation functions.

This can be implemented in Keras by instantiating a layer without an activation function, followed by a `BatchNormalization` layer, and then apply an activation function without any new neurons, using the `Activation` layer. This is shown in Code Snippet 5-6.

Code Snippet 5-6 Batch Normalization before Activation Function

```
keras.layers.Dense(64),
keras.layers.BatchNormalization(),
keras.layers.Activation('tanh'),
```

However, it turns out that batch normalization also works well if done after the activation function, as shown to the right in Figure 5-3. This alternative implementation is shown in Code Snippet 5-7.

Code Snippet 5-7 Batch Normalization after Activation Function

```
keras.layers.Dense(64, activation='tanh'),
keras.layers.BatchNormalization(),
```

Cross-Entropy Loss Function to Mitigate Effect of Saturated Output Neurons

One reason for saturation is that we are trying to make the output neuron get to a value of 0 or 1, which itself drives it to saturation. A simple trick introduced by LeCun, Bottou, Orr, and Müller (1998) is to instead set the desired output to 0.1 or 0.9, which restricts the neuron from being pushed far into the saturation region. We mention this technique for historical reasons, but a more mathematically sound technique is recommended today.

We start by looking at the first couple of factors in the backpropagation algorithm; see Chapter 3, Equation 3-1(1) for more context. The formulas for the MSE loss function, the logistic sigmoid function, and their derivatives for a single training example are restated here:[2]

$$\textit{MSE loss}: \; e(\hat{y}) = \frac{(y-\hat{y})^2}{2}, \qquad e'(\hat{y}) = -(y-\hat{y})$$

$$\textit{Logistic}: \; S(z_f) = \frac{1}{1-e^{-z_f}}, \qquad S'(z_f) = S(z_f)\cdot\left(1-S(z_f)\right)$$

We then start backpropagation by using the chain rule to compute the derivative of the loss function and multiply by the derivative of the logistic sigmoid function to arrive at the following as the error term for the output neuron:

$$\textit{Output neuron error term}: \; \frac{\partial e}{\partial z_f} = \frac{\partial e}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_f} = -(y-\hat{y})\cdot S'(z_f)$$

We chose to not expand $S'(z_f)$ in the expression because it makes the formula unnecessarily cluttered. The formula reiterates what we stated in one of the previous sections: that if $S'(z_f)$ is close to 0, then no error will backpropagate through the network. We show this visually in Figure 5-4. We simply plot the derivative of the loss function and the derivative of the logistic sigmoid function as well as the product of the two. The chart shows these entities as functions of the output value *y* (horizontal axis) of the output neuron. The chart assumes that the desired output value (ground truth) is 0. That is, at the very left in the chart, the output value matches the ground truth, and no weight adjustment is needed.

2. In the equations in Chapter 3, we referred to the output of the last neuron as *f* to avoid confusing it with the output of the other neuron, *g*. In this chapter, we use a more standard notation and refer to predicted value (the output of the network) as $\hat{y}$.

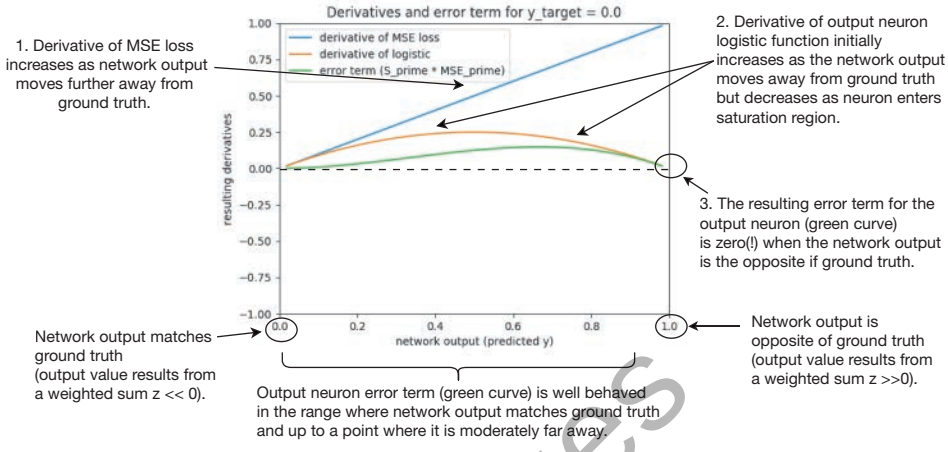

Figure 5-4 Derivatives and error term as function of neuron output when ground truth y (denoted y_target in the figure) is 0

As we move to the right in the chart, the output is further away from the ground truth, and the weights need to be adjusted. Looking at the figure, we see that the derivative of the loss function (blue) is 0 if the output value is 0, and as the output value increases, the derivative increases. This makes sense in that the further away from the true value the output is, the larger the derivative will be, which will cause a larger error to backpropagate through the network. Now look at the derivative of the logistic sigmoid function. It also starts at 0 and increases as the output starts deviating from 0. However, as the output gets closer to 1, the derivative is decreasing again and starts approaching 0 as the neuron enters its saturation region. The green curve shows the resulting product of the two derivatives (the error term for the output neuron), and it also approaches 0 as the output approaches 1 (i.e., the error term becomes 0 when the neuron saturates).

Looking at the charts, we see that the problem arises from the combination of the derivative of the activation function approaching 0, whereas the derivative of the loss function never increases beyond 1, and multiplying the two will therefore approach 0. One potential solution to this problem is to use a different loss function whose derivative can take on much higher values than 1. Without further rationale at this point, we introduce the function in Equation 5-1 that is known as the *cross-entropy loss function:*

$$\textit{Cross entropy loss}:\ e(\hat{y}) = -\left(y \cdot \ln(\hat{y}) + (1-y) \cdot \ln(1-\hat{y})\right)$$

Equation 5-1 Cross-entropy loss function

Substituting the cross-entropy loss function into our expression for the error term of the output neuron yields Equation 5-2:

$$\frac{\partial e}{\partial z_f} = \frac{\partial e}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_f} = -\left(\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right) \cdot S'(z_f) = \hat{y} - y$$

Equation 5-2 Derivative of cross-entropy loss function and derivative of logistic output unit combined into a single expression

We spare you from the algebra needed to arrive at this result, but if you squint your eyes a little bit and remember that the logistic sigmoid function has some e^x terms, and we know that $ln(e^x) = x$ and the derivative of $ln(x) = x^{-1}$, then it does not seem farfetched that our seemingly complicated formulas might end up as something as simple as that. Figure 5-5 shows the equivalent plot for these functions. The *y*-range is increased compared to Figure 5-4 to capture more of the range of the new loss function. Just as discussed, the derivative of the cross-entropy loss function does increase significantly at the right end of the chart, and the resulting product (the green line) now approaches 1 in the case where the neuron is saturated. That is, the backpropagated error is no longer 0, and the weight adjustments will no longer be suppressed.

Although the chart seems promising, you might feel a bit uncomfortable to just start using Equation 5-2 without further explanation. We used the MSE loss function in the first place, you may recall, on the assumption that your likely familiarity with linear regression would make the concept clearer. We even stated that using MSE together with the logistic sigmoid function is not a good choice.

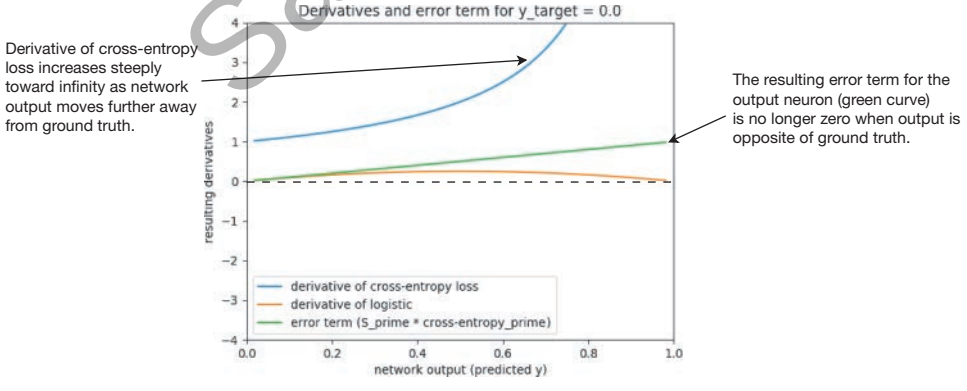

Figure 5-5 Derivatives and error term when using cross-entropy loss function. Ground truth y (denoted y_target in the figure) is 0, as in Figure 5-4.

We have now seen in Figure 5-4 why this is the case. Still, let us at least give you some insight into why using the cross-entropy loss function instead of the MSE loss function is acceptable. Figure 5-6 shows how the value of the MSE and cross-entropy loss function varies as the output of the neuron changes from 0 to 1 in the case of a ground truth of 0. As you can see, as y moves further away from the true value, both MSE and the cross-entropy function increase in value, which is the behavior that we want from a loss function.

Intuitively, by looking at the chart in Figure 5-6, it is hard to argue that one function is better than the other, and because we have already shown in Figure 5-4 that MSE is not a good function, you can see the benefit of using the cross-entropy loss function instead. One thing to note is that, from a mathematical perspective, it does not make sense to use the cross-entropy loss function together with a tanh neuron because the logarithm for negative numbers is not defined.

As further reading, we recommend learning about information theory and maximum-likelihood estimation, which provides a rationale for the use of the cross-entropy loss function.

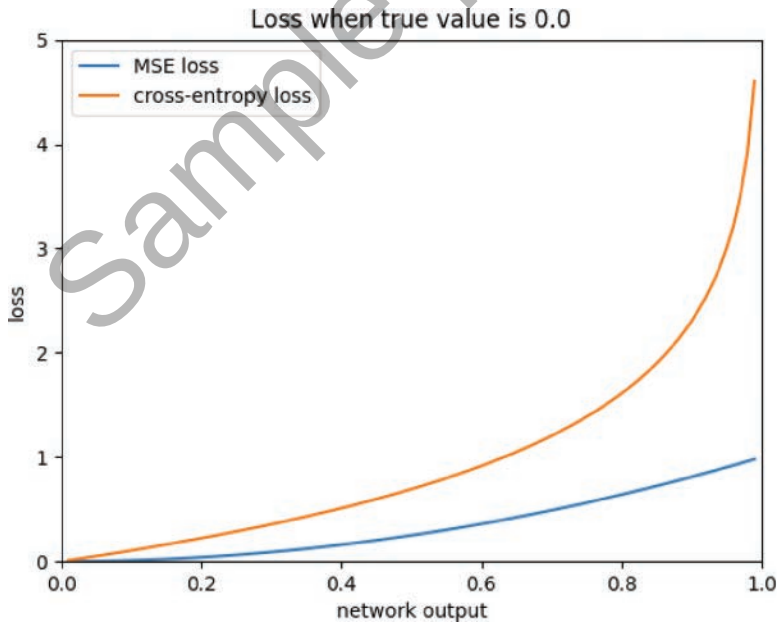

Figure 5-6 Value of the mean squared error (blue) and cross-entropy loss (orange) functions as the network output $\hat{y}$ changes (horizontal axis). The assumed ground truth is 0.

In the preceding examples, we assumed a ground truth of 0. For completeness, Figure 5-7 shows how the derivatives behave in the case of a ground truth of 1.

The resulting charts are flipped in both directions, and the MSE function shows exactly the same problem as for the case when ground truth was 0. Similarly, the cross-entropy loss function solves the problem in this case as well.

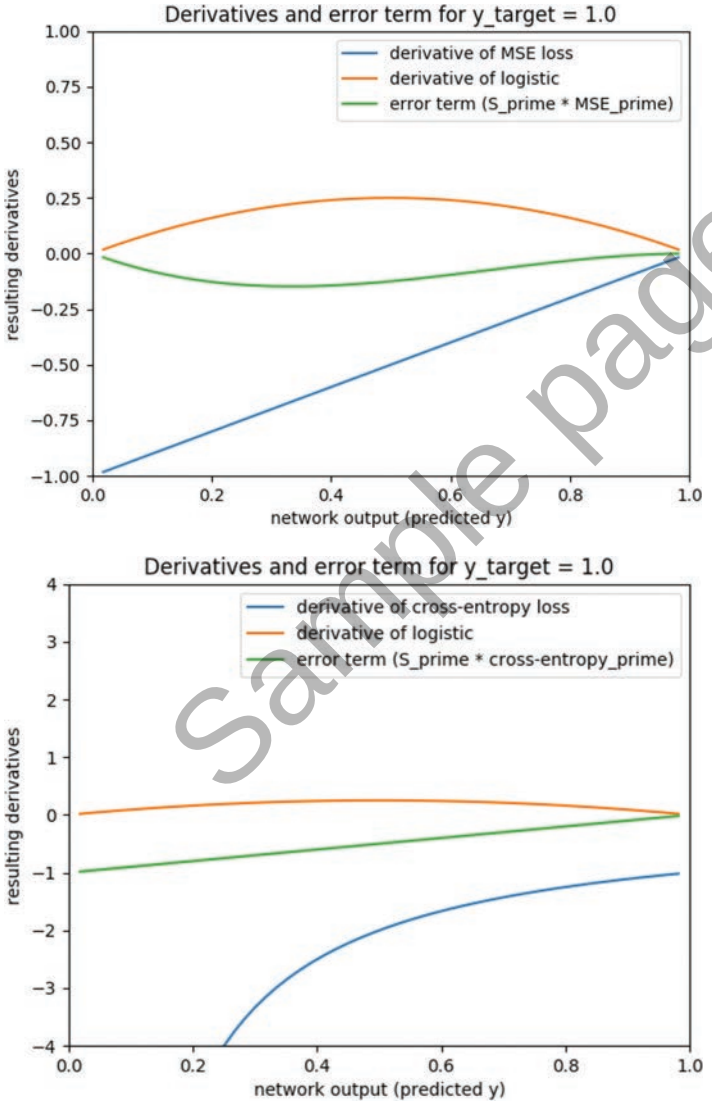

Figure 5-7 Behavior of the different derivatives when assuming a ground truth of 1. Top: Mean squared error loss function. Bottom: Cross-entropy loss function.

COMPUTER IMPLEMENTATION OF THE CROSS-ENTROPY LOSS FUNCTION

If you find an existing implementation of a code snippet that calculates the cross-entropy loss function, then you might be confused at first because it does not resemble what is stated in Equation 5-1. A typical implementation can look like that in Code Snippet 5-8. The trick is that, because we know that y in Equation 5-1 is either 1.0 or 0.0, the factors y and (1-*y*) will serve as an `if` statement and select one of the *ln* statements.

Code Snippet 5-8 Python Implementation of the Cross-Entropy Loss Function

```
def cross_entropy(y_truth, y_predict):
        if y_truth == 1.0:
            return -np.log(y_predict)
        else:
            return -np.log(1.0-y_predict)
```

Apart from what we just described, there is another thing to consider when implementing backpropagation using the cross-entropy loss function in a computer program. It can be troublesome if you first compute the derivative of the cross-entropy loss (as in Equation 5-2) and then multiply by the derivative of the activation function for the output unit. As shown in Figure 5-5, in certain points, one of the functions approaches 0 and one approaches infinity, and although this mathematically can be simplified to the product approaching 1, due to rounding errors, a numerical computation might not end up doing the right thing. The solution is to analytically simplify the product to arrive at the combined expression in Equation 5-2, which does not suffer from this problem.

In reality, we do not need to worry about these low-level details because we are using a DL framework. Code Snippet 5-9 shows how we can tell Keras to use the cross-entropy loss function for a binary classification problem. We simply state `loss='binary_crossentropy'` as an argument to the `compile` function.

Code Snippet 5-9 Use Cross-Entropy Loss for a Binary Classification Problem in TensorFlow

```
model.compile(loss='binary_crossentropy',
              optimizer = optimizer_type,
              metrics =['accuracy'])
```

In Chapter 6, "Fully Connected Networks Applied to Regression," we detail the formula for the categorical cross-entropy loss function, which is used for multiclass classification problems. In TensorFlow, it is as simple as stating `loss='categorical_crossentropy'`.

Different Activation Functions to Avoid Vanishing Gradient in Hidden Layers

The previous section showed how we can solve the problem of saturated neurons in the output layer by choosing a different loss function. However, this does not help for the hidden layers. The hidden neurons can still be saturated, resulting in derivatives close to 0 and vanishing gradients. At this point, you may wonder if we are solving the problem or just fighting symptoms. We have modified (standardized) the input data, used elaborate techniques to initialize the weights based on the number of inputs and outputs, and changed our loss function to accommodate the behavior of our activation function. Could it be that the activation function itself is the cause of the problem?

How did we end up with the tanh and logistic sigmoid functions as activation functions anyway? We started with early neuron models from McCulloch and Pitts (1943) and Rosenblatt (1958) that were both binary in nature. Then Rumelhart, Hinton, and Williams (1986) added the constraint that the activation function needs to be differentiable, and we switched to the tanh and logistic sigmoid functions. These functions kind of look like the sign function yet are still differentiable, but what good is a differentiable function in our algorithm if its derivative is 0 anyway?

Based on this discussion, it makes sense to explore alternative activation functions. One such attempt is shown in Figure 5-8, where we have complicated the activation function further by adding a linear term $0.2*x$ to the output to prevent the derivative from approaching 0.

Although this function might well do the trick, it turns out that there is no good reason to overcomplicate things, so we do not need to use this function. We remember from the charts in the previous section that a derivative of 0 was a problem only in one direction because, in the other direction, the output value already matched the ground truth anyway. In other words, it is fine with a derivative of 0 on one side of the chart. Based on this reasoning, we can consider

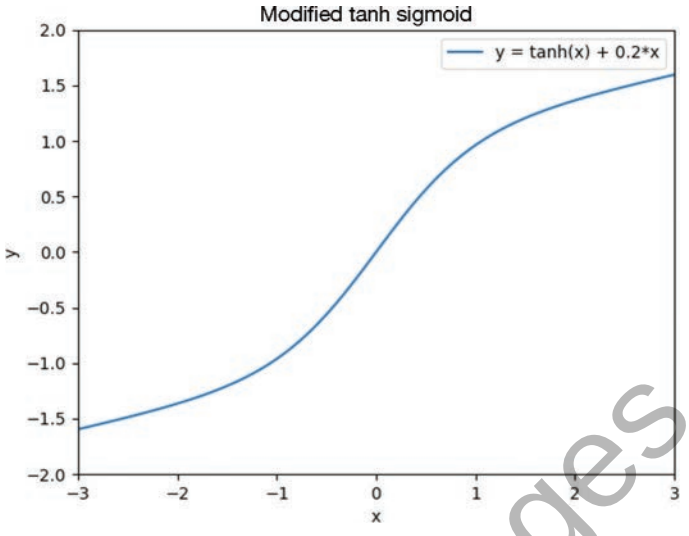

Figure 5-8 Modified tanh function with an added linear term

the rectified linear unit (ReLU) activation function in Figure 5-9, which has been shown to work for neural networks (Glorot, Bordes, and Bengio, 2011).

Now, a fair question is how this function can possibly be used after our entire obsession with differentiable functions. The function in Figure 5-9 is not

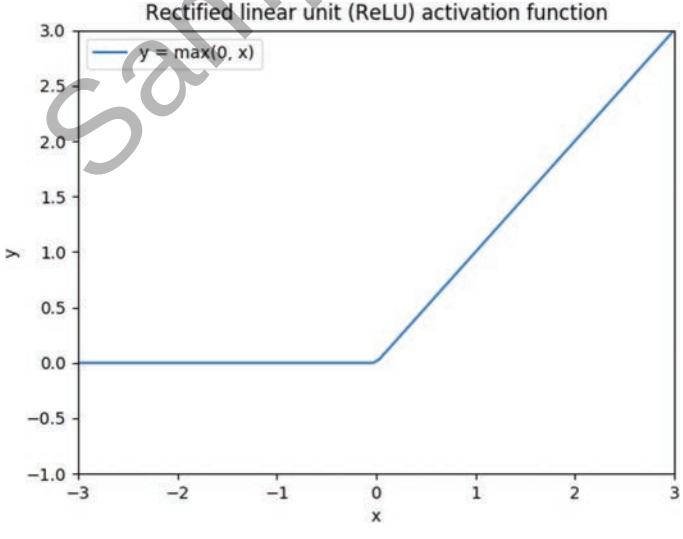

Figure 5-9 Rectified linear unit (ReLU) activation function

differentiable at $x = 0$. However, this does not present a big problem. It is true that from a mathematical point of view, the function is not differentiable in that one point, but nothing prevents us from just defining the derivative as 1 in that point and then trivially using it in our backpropagation algorithm implementation. The key issue to avoid is a function with a discontinuity, like the sign function. Can we simply remove the kink in the line altogether and use $y = x$ as an activation function? The answer is that this does not work. If you do the calculations, you will discover that this will let you collapse the entire network into a linear function and, as we saw in Chapter 1, "The Rosenblatt Perceptron," a linear function (like the perceptron) has severe limitations. It is even common to refer to the activation function as a *nonlinearity,* which stresses how important it is to not pick a linear function as an activation function.

The **activation function** should be **nonlinear** and is even often referred to as a **nonlinearity** instead of activation function.

An obvious benefit with the ReLU function is that it is cheap to compute. The implementation involves testing only whether the input value is less than 0, and if so, it is set to 0. A potential problem with the ReLU function is when a neuron starts off as being saturated in one direction due to a combination of how the weights and inputs happen to interact. Then that neuron will not participate in the network at all because its derivative is 0. In this situation, the neuron is said to be dead. One way to look at this is that using ReLUs gives the network the ability to remove certain connections altogether, and it thereby builds its own network topology, but it could also be that it accidentally kills neurons that could be useful if they had not happened to die. Figure 5-10 shows a variation of the ReLU function known as *leaky ReLU,* which is defined so its derivative is never 0.

Given that humans engage in all sorts of activities that arguably kill their brain cells, it is reasonable to ask whether we should prevent our network from killing its neurons, but that is a deeper discussion.