16. Learning Rate Schedulers

In our last chapter, we trained the MiniVGGNet architecture on the CIFAR-10 dataset. To help alleviate the effects of overfitting, I introduced the concept of adding a *decay* to our learning rate when applying SGD to train the network.

In this chapter we'll discuss the concept of *learning rate schedules*, sometimes called learning rate annealing or adaptive learning rates. By adjusting our learning rate on an epoch-to-epoch basis, we can reduce loss, increase accuracy, and even in certain situations reduce the total amount of time it takes to train a network.

16.1 Dropping Our Learning Rate

The most simple and heavily used learning rate schedulers are ones that progressively reduce learning rate over time. To consider why learning rate schedules are an interesting method to apply to help increase model accuracy, consider our standard weight update formula from Section 9.1.6:

```
W += -args["alpha"] * gradient
```

Recall that the learning rate α controls the "step" we make along the gradient. Larger values of α imply that we are taking bigger steps while smaller values of α will make tiny steps – if α is zero, the network cannot make any steps at all (since the gradient multiplied by zero is zero).

In our previous examples throughout this book, our learning rates were constant – we typically set $\alpha = \{0.1, 0.01\}$ and then trained the network for a fixed number of epochs without changing the learning rate. This method may work well in some situations, but it's often beneficial to *decrease* our learning rate over time.

When training our network, we are trying to find some location along our loss landscape where the network obtains reasonable accuracy. It doesn't have to be a global minima or even a local minima, but in practice, simply finding an area of the loss landscape with reasonably low loss is "good enough".

If we constantly keep a learning rate high, we could overshoot these areas of low loss as we'll be taking *too large* of steps to descend into these areas. Instead, what we can do is decrease our learning rate, thereby allowing our network to take smaller steps – this decreased rate enables our network to descend into areas of the loss landscape that are "more optimal" and would have otherwise been missed entirely by our larger learning rate.

We can, therefore, view the process of learning rate scheduling as:

1. Finding a set of reasonably "good" weights early in the training process with a higher learning rate.
2. Tuning these weights later in the process to find more optimal weights using a smaller learning rate.

There are two primary types of learning rate schedulers that you'll likely encounter:

1. Learning rate schedulers that decrease gradually based on the epoch number (like a linear, polynomial, or exponential function).
2. Learning rate schedulers that drop based on *specific* epoch (such as a piecewise function).

We'll review both types of learning rate schedulers in this chapter.

16.1.1 The Standard Decay Schedule in Keras

The Keras library ships with a time-based learning rate scheduler – it is controlled via the `decay` parameter of the optimizer classes (such as `SGD`).

Going back to our previous chapter, let's take a look at the code block where we initialize SGD and MiniVGGNet:

```
37 # initialize the optimizer and model
38 print("[INFO] compiling model...")
39 opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
40 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
41 model.compile(loss="categorical_crossentropy", optimizer=opt,
42     metrics=["accuracy"])
```

Here we initialize our `SGD` optimizer with a learning rate of $\alpha = 0.01$, a momentum $\gamma = 0.9$, and indicate that we are using Nesterov accelerated gradient. We then set our decay to be the learning rate divided by the total number of epochs we are training the network for (a common rule of thumb), resulting in $0.01/40 = 0.00025$.

Internally, Keras applies the following learning rate schedule to adjust the learning rate after *every batch update* — it is a misconception that Keras updates after every *epoch*. Keep this in mind when using the default learning rate scheduler supplied with Keras.

The update formula follows:

```
lr = init_lr * (1.0 / (1.0 + decay * iterations))
```

Using the CIFAR-10 dataset as an example, we have a total of 50,000 training images. If use a batch size of 64, that implies there are a total of `ceil(50000 / 64) = 782` steps per epoch. Therefore, a total of 782 weight updates need to be applied before an epoch completes.

To see an example of the learning rate schedule calculation, let's assume the parameters mentioned above that our initial learning rate is $\alpha = 0.01$ and `decay = 0.01 / 40.0 = 0.00025` (with an assumption that we are training for forty epochs).

The learning rate at step zero, before any learning rate schedule has been applied, is:

```
lr = 0.01 * (1.0 / (1.0 + 0.00025 * (0 * 782))) = 0.01
```

Epoch	Learning Rate (α)
0	0.01000
1	0.00836
2	0.00719
...	...
37	0.00121
38	0.00119
39	0.00116

Table 16.1: A table demonstrating how our learning rate decreases over time using 40 epochs, an initial learning rate of $\alpha = 0.01$ and a decay term of $0.01/40$.

At the beginning of epoch one we can see the following learning rate:

```
lr = 0.01 * (1.0 / (1.0 + 0.00025 * (1 * 782))) = 0.00836
```

Table 16.1 continues this learning rate schedule calculation.

Using this time-based learning rate decay, our MiniVGGNet model obtained 83% classification accuracy, as shown in Chapter 15. I would encourage you to set `decay=0` in the SGD optimizer and then rerun the experiment. You'll notice that the network also obtains $\approx 83\%$ classification accuracy; however, by investigating the learning plots of the two models in Figure 16.1, you'll notice that overfitting is starting to occur as validation loss rises past epoch 25 (*left*).

This result is in contrast to when we set `decay=0.01 / 40` (*right*) and obtain a much nicer learning plot (and not to mention, higher accuracy). By using learning rate decay we can often not only improve our classification accuracy but also lessen the effects of overfitting, thereby increasing the ability of our model to generalize.

Figure 16.1: **Left:** Training MiniVGGNet on CIFAR-10 *without* learning rate decay. Notice how loss starts to increase past epoch 25, indicating that overfitting is happening. **Right:** Applying a decay factor of $0.01/40$. This reduces the learning rate over time, helping alleivate the affects of overfitting.

16.1.2 Step-based Decay

Another popular learning rate scheduler is step-based decay where we systematically drop the learning rate after *specific* epochs during training. The step decay learning rate schedulers can be thought of as a piecewise function, such as in Figure 16.2. Here the learning rate is constant for a number of epochs, then drops, and is constant once more, then drops again, etc.

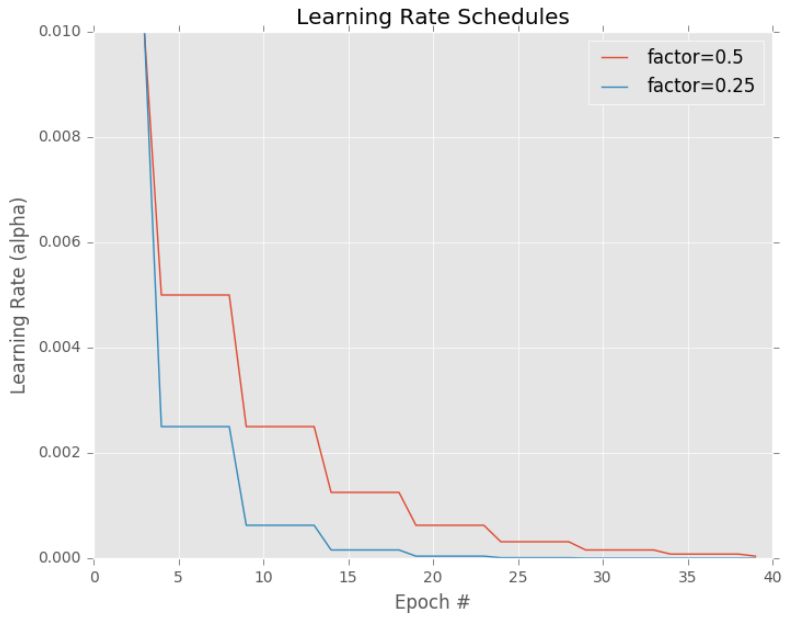

Figure 16.2: An example of two learning rate schedules that drop the learning rate in a piecewise fashion. Lowering the *factor* value increases the speed of the drop. In each case, the learning rate approaches zero at the final epoch.

When applying step decay to our learning rate, we have two options:

1. Define an equation that models the piecewise drop in learning rate we wish to achieve.
2. Use what I call the `ctrl + c` method to training a deep learning network where we train for some number of epochs at a given learning rate, eventually notice validation performance has stalled, then `ctrl + c` to stop the script, adjust our learning rate, and continue training.

We'll primarily be focusing on the equation-based piecewise drop to learning rate scheduling in this chapter. The `ctrl + c` method is more advanced and is normally applied to larger datasets using deeper neural networks where the exact number of epochs required to obtain reasonable accuracy is unknown. I cover `ctrl + c` training *heavily* inside the *Practitioner Bundle* and *ImageNet Bundle* of this book.

When applying step decay, we often drop our learning rate by either (1) half or (2) an order of magnitude after every fixed number of epochs. For example, let's suppose our initial learning rate is $\alpha = 0.1$. After 10 epochs we drop the learning rate to $\alpha = 0.05$. After another 10 epochs of training (i.e., the 20th total epoch), α is dropped by 0.5 again, such that $\alpha = 0.025$, etc. In fact, this is the exact same learning rate schedule plotted in Figure 16.2 above (red line). The blue line displays a much more aggressive drop with a factor of 0.25.

16.1.3 Implementing Custom Learning Rate Schedules in Keras

Conveniently, the Keras library provides us with a `LearningRateScheduler` class that allows us to define a *custom learning rate function* and then have it *automatically applied* during the training process. This function should take the epoch number as an argument and then compute our desired learning rate based on a function that we define.

In this example, we'll be defining a piecewise function that will drop the learning rate by a certain factor F after every D epochs. Our equation will thus look like this:

$$\alpha_{E+1} = \alpha_I \times F^{(1+E)/D} \tag{16.1}$$

Where α_I is our initial learning rate, F is the factor value controlling the rate in which the learning rate drops, D is the "drop every" epochs value, and E is the current epoch. The *larger* our factor F is, the *slower* the learning rate will decay. Conversely, the *smaller* the factor F is the *faster* the learning rate will decrease.

Written in Python code, this equation might be expressed as:

```
alpha = initAlpha * (factor ** np.floor((1 + epoch) / dropEvery))
```

Let's go ahead and implement this custom learning rate schedule and then apply it to MiniVGGNet on CIFAR-10. Open up a new file, name it `cifar10_lr_decay.py`, and let's start coding:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from sklearn.preprocessing import LabelBinarizer
7  from sklearn.metrics import classification_report
8  from pyimagesearch.nn.conv import MiniVGGNet
9  from tensorflow.keras.callbacks import LearningRateScheduler
10 from tensorflow.keras.optimizers import SGD
11 from tensorflow.keras.datasets import cifar10
12 import matplotlib.pyplot as plt
13 import numpy as np
14 import argparse
```

Lines 2-14 import our required Python packages as in the original `minivggnet_cifar10.py` script from Chapter 15. However, take notice of **Line 9** where we import our `LearningRateScheduler` from the Keras library – this class will enable us to define our own custom learning rate scheduler.

Speaking of a custom learning rate scheduler, let's define that now:

```
16 def step_decay(epoch):
17     # initialize the base initial learning rate, drop factor, and
18     # epochs to drop every
19     initAlpha = 0.01
20     factor = 0.25
21     dropEvery = 5
22
23     # compute learning rate for the current epoch
24     alpha = initAlpha * (factor ** np.floor((1 + epoch) / dropEvery))
```

```
25
26      # return the learning rate
27      return float(alpha)
```

Line 16 defines the `step_decay` function which accepts a single required parameter – the current `epoch`. We then define the initial learning rate (`0.01`), the drop factor (`0.25`), set `dropEvery = 5`, implying that we'll drop our learning rate by a factor of 0.25 every five epochs (**Lines 19-21**).

We compute the new learning rate for the current epoch on **Line 24** using the Equation 16.1 above. This new learning rate is returned to the calling function on **Line 27**, allowing Keras to internally update the optimizer's learning rate.

From here we can continue on with our script:

```
29  # construct the argument parse and parse the arguments
30  ap = argparse.ArgumentParser()
31  ap.add_argument("-o", "--output", required=True,
32      help="path to the output loss/accuracy plot")
33  args = vars(ap.parse_args())
34
35  # load the training and testing data, then scale it into the
36  # range [0, 1]
37  print("[INFO] loading CIFAR-10 data...")
38  ((trainX, trainY), (testX, testY)) = cifar10.load_data()
39  trainX = trainX.astype("float") / 255.0
40  testX = testX.astype("float") / 255.0
41
42  # convert the labels from integers to vectors
43  lb = LabelBinarizer()
44  trainY = lb.fit_transform(trainY)
45  testY = lb.transform(testY)
46
47  # initialize the label names for the CIFAR-10 dataset
48  labelNames = ["airplane", "automobile", "bird", "cat", "deer",
49      "dog", "frog", "horse", "ship", "truck"]
```

Lines 30-33 parse our command line arguments. We only need a single argument, `--output`, the path to our output loss/accuracy plot. We then load the CIFAR-10 dataset from disk and scale the pixel intensities to the range $[0, 1]$ on **Lines 37-40**. **Lines 43-45** handle one-hot encoding the class labels.

Next, let's train our network:

```
51  # define the set of callbacks to be passed to the model during
52  # training
53  callbacks = [LearningRateScheduler(step_decay)]
54
55  # initialize the optimizer and model
56  opt = SGD(lr=0.01, momentum=0.9, nesterov=True)
57  model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
58  model.compile(loss="categorical_crossentropy", optimizer=opt,
59      metrics=["accuracy"])
60
61  # train the network
62  H = model.fit(trainX, trainY, validation_data=(testX, testY),
63      batch_size=64, epochs=40, callbacks=callbacks, verbose=1)
```

Line 53 is important as it initializes our list of `callbacks`. Depending on how the callback is defined, Keras will call this function at the start or end of every epoch, mini-batch update, etc. The `LearningRateScheduler` will call `step_decay` at the end of every epoch, allowing us to update the learning prior to the *next* epoch starting.

Line 56 initializes the `SGD` optimizer with a momentum of `0.9` and Nestrov accelerated gradient. The `lr` parameter will be ignored here since we'll be using the `LearningRateScheduler` callback so we could *technically* leave this parameter out entirely; however, I like to include it and have it match `initAlpha` as a matter of clarity.

Line 57 initializes MiniVGGNet which we then train for 40 epochs on **Lines 62 and 63**.

Once the network is trained we can evaluate it:

```
65 # evaluate the network
66 print("[INFO] evaluating network...")
67 predictions = model.predict(testX, batch_size=64)
68 print(classification_report(testY.argmax(axis=1),
69     predictions.argmax(axis=1), target_names=labelNames))
```

As well as plot the loss and accuracy:

```
71 # plot the training loss and accuracy
72 plt.style.use("ggplot")
73 plt.figure()
74 plt.plot(np.arange(0, 40), H.history["loss"], label="train_loss")
75 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
76 plt.plot(np.arange(0, 40), H.history["accuracy"], label="train_acc")
77 plt.plot(np.arange(0, 40), H.history["val_accuracy"], label="val_acc")
78 plt.title("Training Loss and Accuracy on CIFAR-10")
79 plt.xlabel("Epoch #")
80 plt.ylabel("Loss/Accuracy")
81 plt.legend()
82 plt.savefig(args["output"])
```

To evaluate the effect the drop factor has on learning rate scheduling and overall network classification accuracy, we'll be evaluating two drop factors: 0.25 and 0.5, respectively. The drop factor of 0.25 will decrease significantly faster than the 0.5 rate, as we know from Figure 16.2 above.

Again, take notice how much faster the 0.25 drop factor lowers our learning rate. We'll go ahead and evaluate the faster learning rate drop of 0.25 (**Line 20**) – to execute our script, just issue the following command:

```
$ python cifar10_lr_decay.py --output output/lr_decay_f0.25_plot.png
[INFO] loading CIFAR-10 data...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
34s - loss: 1.6380 - acc: 0.4550 - val_loss: 1.1413 - val_acc: 0.5993
Epoch 2/40
34s - loss: 1.1847 - acc: 0.5925 - val_loss: 1.0986 - val_acc: 0.6057
...
Epoch 40/40
34s - loss: 0.5423 - acc: 0.8081 - val_loss: 0.5899 - val_acc: 0.7885
[INFO] evaluating network...
```

```
              precision    recall  f1-score   support

    airplane       0.81      0.81      0.81      1000
  automobile       0.91      0.89      0.90      1000
        bird       0.71      0.65      0.68      1000
         cat       0.63      0.60      0.62      1000
        deer       0.72      0.79      0.75      1000
         dog       0.70      0.67      0.68      1000
        frog       0.80      0.88      0.84      1000
       horse       0.86      0.83      0.84      1000
        ship       0.87      0.90      0.88      1000
       truck       0.87      0.87      0.87      1000

   avg / total       0.79      0.79      0.79     10000
```

Here we see that our network obtains only 79% classification accuracy. The learning rate is dropping quite aggressively – after epoch fifteen α is only 0.00125, meaning that our network is taking very small steps along the loss landscape. This behavior can be seen in the Figure 16.3 (*left*) where validation loss and accuracy have essentially stagnated after epoch fifteen.

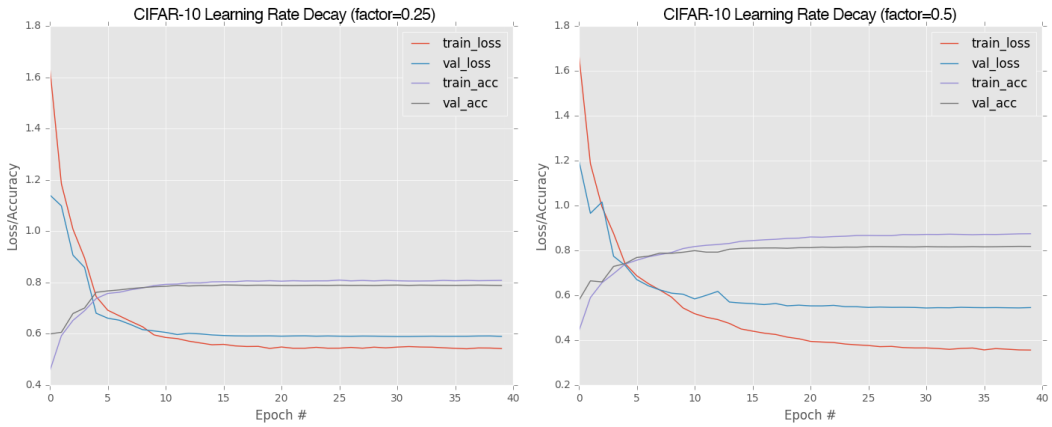

Figure 16.3: **Left:** Plotting the accuracy/loss of our network using a faster learning rate drop with a factor of 0.25. Notice how loss/accuracy stagnate past epoch 15 as the learning rate is too small. **Right:** Accuracy/loss of our network with a slower learning rate drop ($factor = 0.5$). This time our network is able to continue to learn past epoch 30 until stagnation occurs.

If we instead change the drop factor to 0.5 by setting `factor = 0.5` inside of `step_decay`:

```
16 def step_decay(epoch):
17     # initialize the base initial learning rate, drop factor, and
18     # epochs to drop every
19     initAlpha = 0.01
20     factor = 0.5
21     dropEvery = 5
```

And then re-run the experiment, we'll obtain higher classification accuracy:

```
$ python cifar10_lr_decay.py --output output/lr_decay_f0.5_plot.png
[INFO] loading CIFAR-10 data...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
35s - loss: 1.6733 - acc: 0.4402 - val_loss: 1.2024 - val_acc: 0.5771
Epoch 2/40
34s - loss: 1.1868 - acc: 0.5898 - val_loss: 0.9651 - val_acc: 0.6643
...
Epoch 40/40
33s - loss: 0.3562 - acc: 0.8742 - val_loss: 0.5452 - val_acc: 0.8177
[INFO] evaluating network...
             precision    recall  f1-score   support

   airplane       0.85      0.82      0.84      1000
 automobile       0.91      0.91      0.91      1000
       bird       0.75      0.70      0.73      1000
        cat       0.68      0.65      0.66      1000
       deer       0.75      0.82      0.78      1000
        dog       0.74      0.74      0.74      1000
       frog       0.83      0.89      0.86      1000
      horse       0.88      0.86      0.87      1000
       ship       0.89      0.91      0.90      1000
      truck       0.89      0.88      0.88      1000

avg / total       0.82      0.82      0.82     10000
```

This time, with the slower drop in learning rate we obtain 82% accuracy. Looking at the plot in Figure 16.3 (*right*) we can see that our network continues to learn past epoch 25-30 until loss stagnates on the validation data past epoch 30 the learning rate is very small at `2.44e-06` and is unable to make any significant changes to the weights to influence the loss/accuracy on the validation data.

16.2 Summary

The purpose of this chapter was to review the concept of *learning rate schedulers* and how they can be used to increase classification accuracy. We discussed the two primary types of learning rate schedulers:

1. Time-based schedulers that gradually decrease based on epoch number.
2. Drop-based schedulers that drop based on a *specific* epoch, similar to the behavior of a piecewise function.

Exactly *which* learning rate scheduler you should use (if you should use a scheduler at all) is part of the experimentation process. Typically your first experiment *would not* use any type of decay or learning rate scheduling so you can obtain a *baseline* accuracy and loss/accuracy curve.

From there you might introduce the standard time-based schedule provided by Keras (with the rule of thumb of `decay = alpha_init / epochs`) and run a second experiment to evaluate the results. The next few experiments might involve swapping out a time-bases schedule for a drop-based one using various drop factors.

Depending on how challenging your classification dataset is along with the depth of your network, you might opt for the `ctrl + c` method to training as detailed in the *Practitioner Bundle* and *ImageNet Bundle* which is the approach taken by most deep learning practitioners when training networks on the ImageNet dataset.

Overall, be prepared to spend a *significant amount of time* training your networks and evaluating different sets of parameters and learning routines. Even simple datasets and projects can take 10's to 100's of experiments to obtain a high accuracy model.

At this point in your study of deep learning you should understand that training deep neural networks is part science, part art. My goal in this book is to provide you with the science behind training a network along with the common rules of thumb that I use so you can learn the "art" behind it – but keep in mind that *nothing* beats *actually running the experiments yourself*.

The more practice you have at training neural networks, logging the results of what did work and what didn't, the better you'll become at it. There is *no shortcut* when it comes to mastering this art – you need to put in the hours and become comfortable with the SGD optimizer (and others) along with their parameters.