**Learning Rate Schedule in Practice: an example with Keras and TensorFlow 2.0**

A tutorial to add and customize **learning rate schedule**

One of the painful things about training a neural network is the sheer number of hyperparameters we have to deal with. For example

* Learning rate
* Momentum or the hyperparameters for Adam optimization algorithm
* Number of layers
* Number of hidden units
* Mini-batch size
* Activation function
* etc

Among them, the most important parameter is the learning rate. If your learning rate is set to low, training will progress very slowly as you are making very tiny updates to the weights in your network. However, if your learning rate is set too high, it can cause undesirable divergent behavior in your loss function.

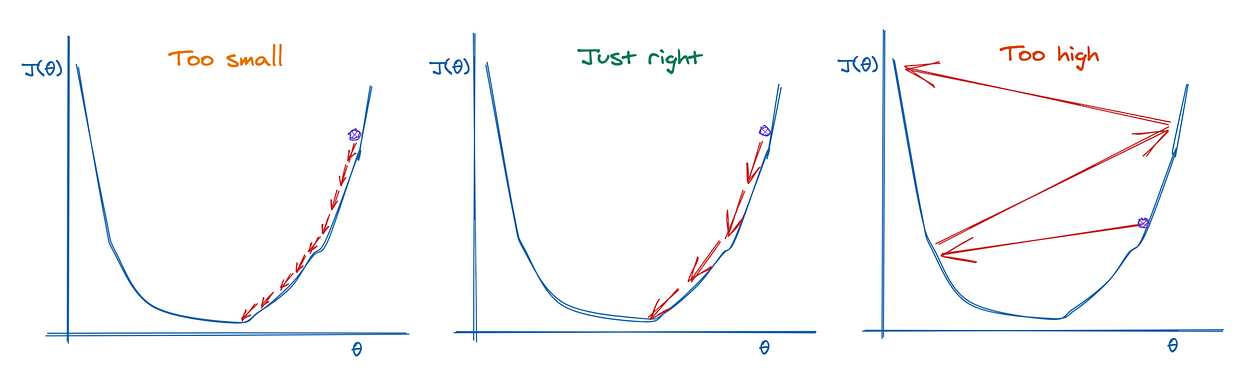


Image created by author using <https://excalidraw.com/>

When training a neural network, it is often useful to reduce the learning rate as the training progresses. This can be done by using **learning rate schedules** or **adaptive learning rate**. In this article, we will focus on adding and customizing **learning rate schedule** in our machine learning model and look at examples of how we do them in practice with Keras and TensorFlow 2.0

**Learning Rate Schedules**

**Learning Rate Schedules** seek to adjust the learning rate during training by reducing the learning rate according to a pre-defined schedule. The popular learning rate schedules include

1. Constant learning rate
2. Time-based decay
3. Step decay
4. Exponential decay

For the demonstration purpose, we will build an image classifier to tackle Fashion MNIST, which is a dataset that has 70,000 grayscale images of 28-by-28 pixels with 10 classes.

Please check out [my Github repo](https://github.com/BindiChen/machine-learning/blob/master/tensorflow2/006-learning-rate-schedules/learning-rate-schedules.ipynb) for source code.

**Using Keras to load the dataset**

Keras provides some utility functions to fetch and load common datasets, including Fashion MNIST. Let’s load Fashion MNIST

fashion\_mnist = keras.datasets.fashion\_mnist  
**(X\_train\_full, y\_train\_full), (X\_test, y\_test) = fashion\_mnist.load\_data()**

The dataset is already split into a training set and a test set. Here is the shape and data type of the training set:

>>> X\_train\_full.shape  
**(60000, 28, 28)**  
>>> X\_train\_full.dtype  
**dtype('uint8')**

We are going to train the neural network using Gradient Descent, we must scale the input feature down to the 0–1 range. And for faster training on a local machine, let’s just use the first 10,000 images.

X\_train, y\_train = **X\_train\_full[:10000]/255.0**, **y\_train\_full[:10000]**

**Creating a Model**

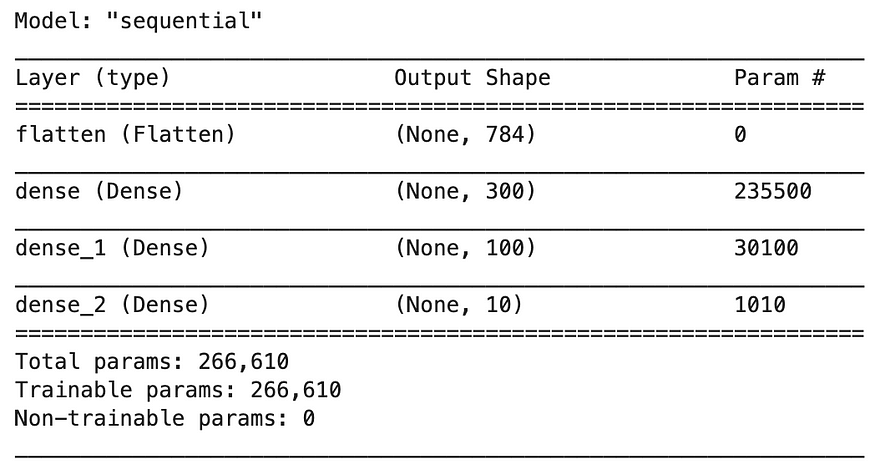
Now let’s build the neural network! There are [3 ways to create a machine learning model with Keras and TensorFlow 2.0](https://towardsdatascience.com/3-ways-to-create-a-machine-learning-model-with-keras-and-tensorflow-2-0-de09323af4d3). Since we are building a simple fully connected neural network and for simplicity, let’s use the easiest way: Sequential Model with Sequential().

from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Flattendef create\_model():   
 model = Sequential([  
 Flatten(**input\_shape=(28, 28)**),  
 Dense(300, activation='relu'),  
 Dense(100, activation='relu'),  
 Dense(**10, activation='softmax'**),  
 ])  
 return model

Our model has the following specifications:

* The first layer (also known as the input layer) has the input\_shape to set the input size (28, 28) which matches the training data. The input layer is a Flatten layer whose role is simply to convert each input image into a 1D array.
* And then it is followed by 2 Dense layers, one with 300 units, and the other with 100 units. Both of them use the relu activation function.
* The output Dense layer has 10 units and the softmax activation function.

model = create\_model()  
model.summary()



**1. Constant learning rate**

The constant learning rate is the default schedule in all Keras Optimizers. For example, in the [SGD optimizer](https://keras.io/api/optimizers/sgd/), the learning rate defaults to 0.01.

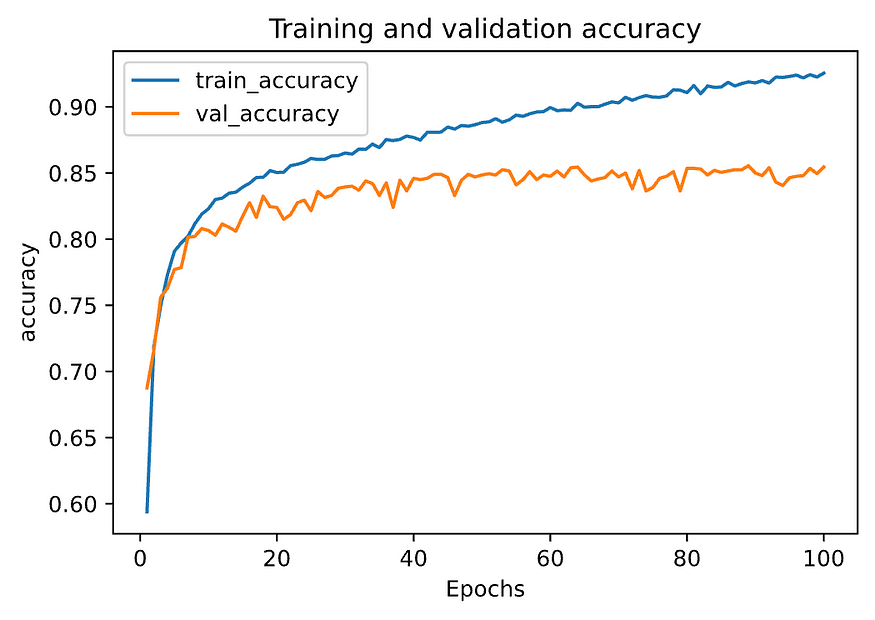
To use a custom learning rate, simply instantiate an SGD optimizer and pass the argument learning\_rate=0.01 .

sgd = tf.keras.optimizers.SGD(**learning\_rate=0.01**)model.compile(  
 **optimizer=sgd,**   
 loss='sparse\_categorical\_crossentropy',   
 metrics=['accuracy']  
)

And to fit the model to training data:

history\_constant = model.fit(  
 X\_train,   
 y\_train,   
 epochs=100,   
 validation\_split=0.2,  
 batch\_size=64  
)

Let’s plot the model accuracy and this can serve as a baseline for us to experiment with other learning rate schedules.



Constant Learning Rate — accuracy plot

**2. Time-based decay**

Time-based decay is one of the most popular learning rate schedules. Formally, the time-based decay is defined as:

learning\_rate = lr \* 1 / (1 + decay \* epoch)

where lr is the previous learning rate, decay is a hyperparameter and epoch is the iteration number. When the decay is zero, this has no effect on changing the learning rate. When the decay is specified, it will decrease the learning rate from the previous epoch by the given fixed amount. The value of decay is normally implemented as

decay = initial\_learning\_rate / num\_of\_epoches

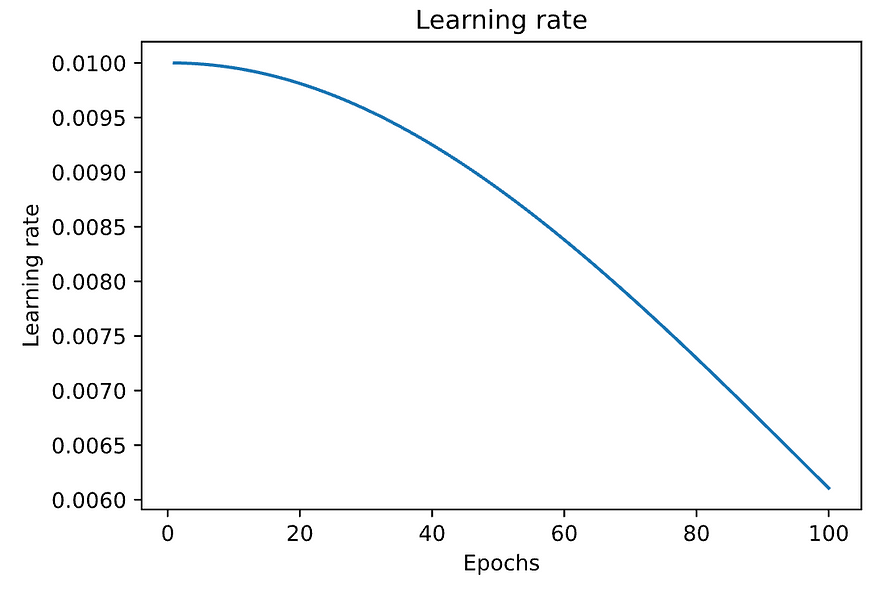
In Keras, one way to implement the time-based decay is by defining a time-based decay function **lr\_time\_based\_decay()**and pass it to LearningRateScheduler callback.

initial\_learning\_rate = 0.01  
epochs = 100  
decay = initial\_learning\_rate / epochs**def lr\_time\_based\_decay(epoch, lr):  
 return lr \* 1 / (1 + decay \* epoch)**history\_time\_based\_decay = model.fit(  
 X\_train,   
 y\_train,   
 epochs=100,   
 validation\_split=0.2,  
 batch\_size=64,  
 **callbacks=[LearningRateScheduler(lr\_time\_based\_decay, verbose=1)],**  
)

And below are the plots of accuracy and learning rate.



Time-based decay — accuracy plot



Time-based decay — learning rate plot

**3. Step decay**

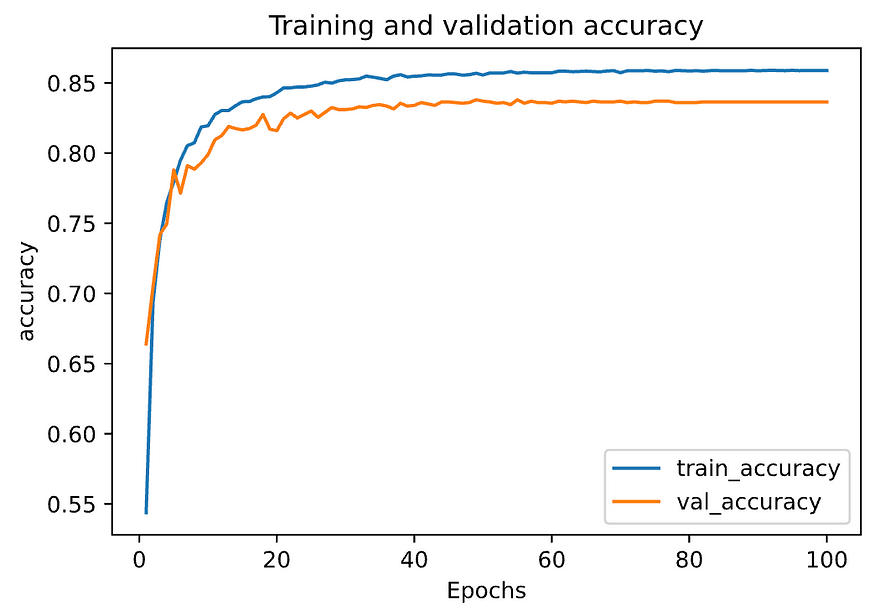
Another popular learning rate schedule is to systematically drop the learning rate at specific times during training. Formally, it is defined as:

learning\_rate = initial\_lr \* drop\_rate^floor(epoch / epochs\_drop)

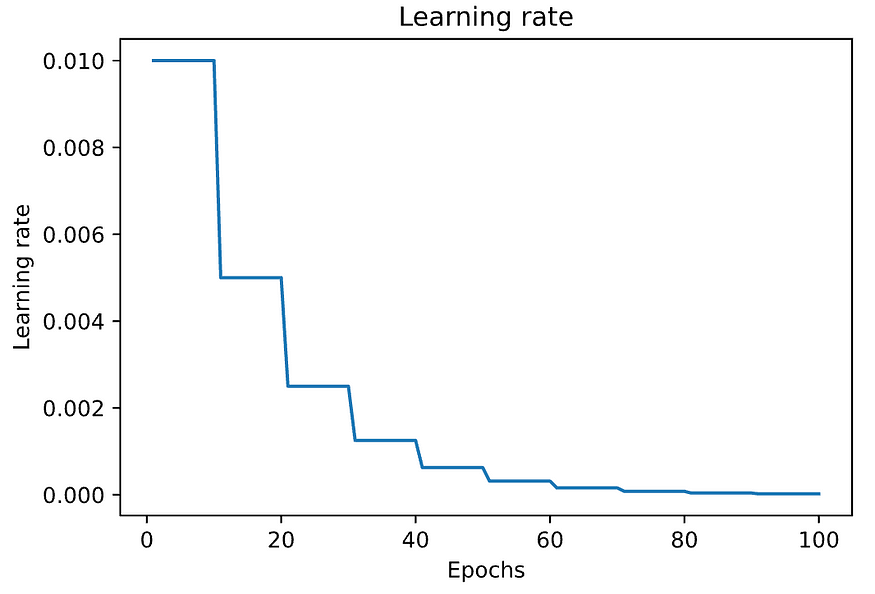
Where initial\_lr is the initial learning rate such as 0.01, the drop\_rate is the amount that the learning rate is modified each time if it is changed, epoch is the current epoch number, and epochs\_drop is how often to change the learning rate such as 10 epochs. Similarly, we can implement this by defining a step decay function **lr\_step\_decay()** and pass it to LearningRateScheduler callback.

initial\_learning\_rate = 0.01**def lr\_step\_decay(epoch, lr):  
 drop\_rate = 0.5  
 epochs\_drop = 10.0  
 return initial\_learning\_rate \* math.pow(drop\_rate, math.floor(epoch/epochs\_drop))**# Fit the model to the training data  
history\_step\_decay = model.fit(  
 X\_train,   
 y\_train,   
 epochs=100,   
 validation\_split=0.2,  
 batch\_size=64,  
 **callbacks=[LearningRateScheduler(lr\_step\_decay, verbose=1)],**  
)

And below are the plots of the accuracy and learning rate.



Step-based decay — accuracy plot



Step-based decay — learning rate

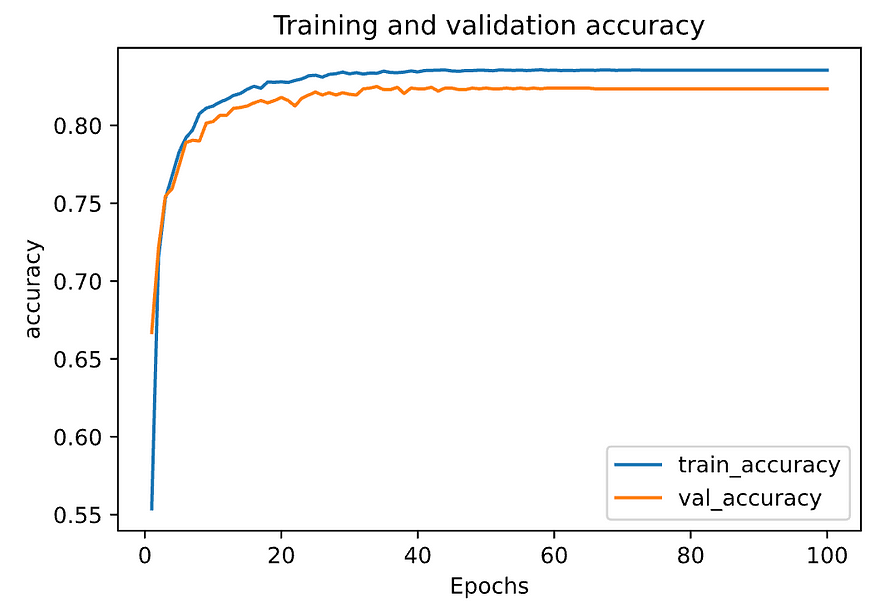
**4. Exponential decay**

Another popular learning rate schedule is to drop the learning rate at an exponential rate. Formally, it is defined as:

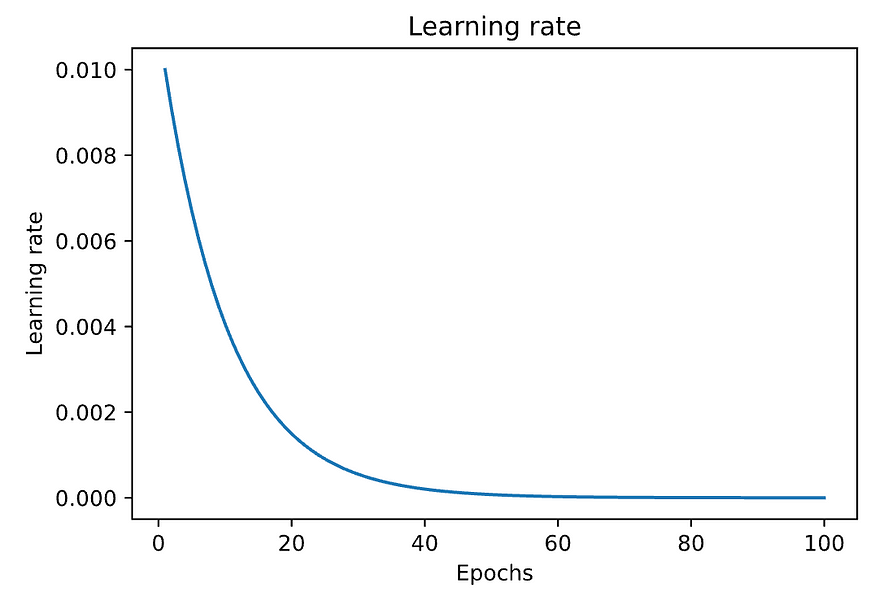
learning\_rate = initial\_lr \* e^(−k \* epoch)

Where initial\_lr is the initial learning rate such as 0.01, k is a hyperparameter, and epoch is the current epoch number. Similarly, we can implement this by defining an exponential decay function **lr\_exp\_decay()** and pass it to LearningRateScheduler callback.

initial\_learning\_rate = 0.01**def lr\_exp\_decay(epoch, lr):  
 k = 0.1  
 return initial\_learning\_rate \* math.exp(-k\*epoch)**# Fit the model to the training data  
history\_exp\_decay = model.fit(  
 X\_train,   
 y\_train,   
 epochs=100,   
 validation\_split=0.2,  
 batch\_size=64,  
 **callbacks=[LearningRateScheduler(lr\_exp\_decay, verbose=1)],**  
)



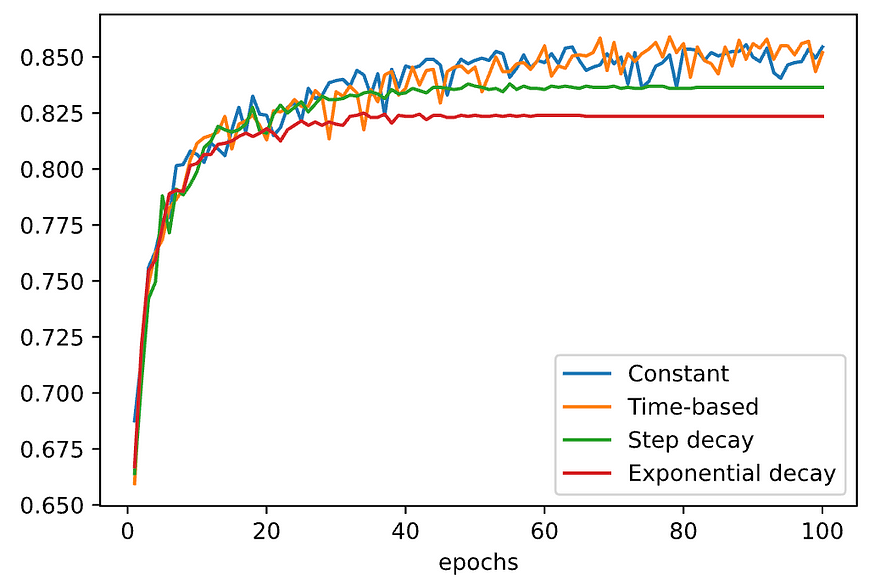
Exponential decay — accuracy plot



Exponential decay — learning rate plot

**Compare model accuracy**

Finally, let us compare the model accuracy using different learning rate schedules.



Looks like **Constant** and **Time-based** learning rates have better performance than **Step decay** and **Exponential decay** for this particular tutorial. Bear in mind that, this tutorial only uses the first 10,000 images with some arbitrary value forinitial\_learning\_rate=0.01, validation\_split=0.2 and batch\_size=64 .

In a real-world application, they are a lot more to consider for tunning the learning rate. Please check out the paper “[Practical Recommendations for Gradient-based Training of Deep Architectures](https://arxiv.org/pdf/1206.5533v2.pdf)” for some best practices.

**That’s it**

Thanks for reading. This article has covered the most popular **Learning Rate Schedules**. Next time, we will take a look at the **Adaptive Learning Rate**.