## Optimization in Neural Networks

A neural network can be defined as a framework that combines inputs and tries to guess the output. If we are lucky enough to have some results, called "the ground truth", to compare the outputs produced by the network, we can calculate the **error**. So the network guesses, calculates some error function, guesses again, trying to minimize this error, guesses again, until the error does not go down any more. This is optimization.

In neural networks, the most commonly used optimization algorithms are flavors of **GD (gradient descent)**. The *objective function* used in gradient descent is the *loss function* which we want to minimize.

This tutorial will lean heavily on Keras now, so I will give a brief Keras refresher.

### A K`eras` Refresher

`Keras` is a Python library for deep learning that can run on top of both Theano or TensorFlow, two powerful Python libraries for fast numerical computing created and released by Facebook and Google, respectively.

Keras was developed to make developing deep learning models as fast and easy as possible for research and practical applications. It runs on Python 2.7 or 3.5 and can seamlessly execute on GPUs and CPUs.

Keras is built on the idea of a model. At its core we have a sequence of layers called the `Sequential` model which is a linear stack of layers. Keras also provides the functional API, a way to define complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

We can summarize the construction of deep learning models in Keras using the Sequential model as follows:

1. **Define your model**: create a `Sequential` model and add layers.
2. **Compile your model**: specify loss function and optimizers and call the `.compile()` function.

3. **Fit your model**: train the model on data by calling the `.fit()` function.
4. **Make predictions**: use the model to generate predictions on new data by calling functions such as `.evaluate()` or `.predict()`.

You may be asking yourself — how can you examine the performance of the model as it is running? This is a good question, and the answer to that is using *callbacks*.

#### Callbacks: taking a peek into our model while it's training

You can look at what is happening in various stages of your model by using `callbacks`. A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training. You can pass a list of callbacks (as the keyword argument callbacks) to the `.fit()` method of the Sequential or Model classes. The relevant methods of the callbacks will then be called at each stage of the training.

- A callback function you are already familiar with is `keras.callbacks.History()`. This is automatically included in `.fit()`.
- Another very useful one is `keras.callbacks.ModelCheckpoint` which saves the model with its weights at a certain point in the training. This can prove useful if your model is running for a long time and a system failure happens. Not all is lost then. It's a good practice to save the model weights only when an improvement is observed as measured by the `acc`, for example.
- `keras.callbacks.EarlyStopping` stops the training when a monitored quantity has stopped improving.
- `keras.callbacks.LearningRateScheduler` will change the learning rate during training.

We will apply some callbacks later. For full documentation on `callbacks` see https://keras.io/callbacks/.

The first thing we must do is import a lot of different functions in order to make our lives easier.

Another additional step you can do if you want your network to work using random numbers but for the result to be repeatable is to use a *random seed*. This basically produces the same sequence of numbers each time, although they are still pseudorandom (these are a great way for comparing models and also testing for reproducibility).

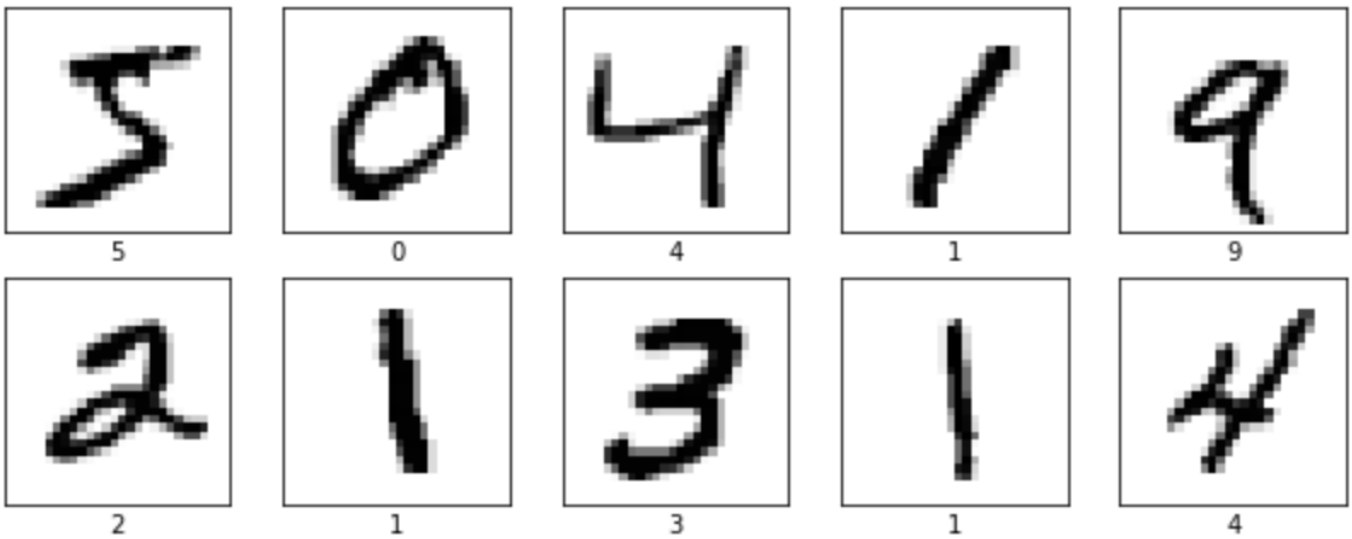
#### Step 1 — Deciding on the network topology (not really considered optimization but is obviously very important)

We will use the MNIST dataset which consists of grayscale images of handwritten digits (0–9) whose dimension is 28x28 pixels. Each pixel is 8 bits so its value ranges from 0 to 255.

Obtaining the dataset is very easy since there is a function for it built-in to `Keras` .

Our output for our *X* and *Y* data is (60000, 28, 28) and (60000,1) respectively. It is always a good suggestion to print some of the data to check the values (and the data type if necessary).

We can check the training data by looking at one image of each of the digits to make sure that none of them are missing from our data.




The last check is for the dimensions of the training and test sets, which can be done relatively easily:

We find that we have 60,000 training images and 10,000 test images. The next thing to do is preprocess the data.

#### Preprocessing the data

To run our NN we need to preprocess the data (these steps can be performed interchangeably):

- First, we need to make the 2D image arrays into 1D (flatten them). We can either perform this by using array reshaping with `numpy.reshape()` or the `keras`' method for this: a layer called `keras.layers.Flatten` which transforms the format of the images from a 2d-array (of 28 by 28 pixels), to a 1D-array of 28 * 28 = 784 pixels.
- Then we need to normalize the pixel values (give them values between 0 and 1) using the following transformation:

$$x := \frac{x - x_{min}}{x_{max} - x_{min}}$$

In our case, the minimum is zero and the maximum is 255, so the formula becomes simply $x:=x/255$.

We now want to one-hot encode our data.

Now we are finally ready to build our model!

### Step 2 — Adjusting the `learning rate`

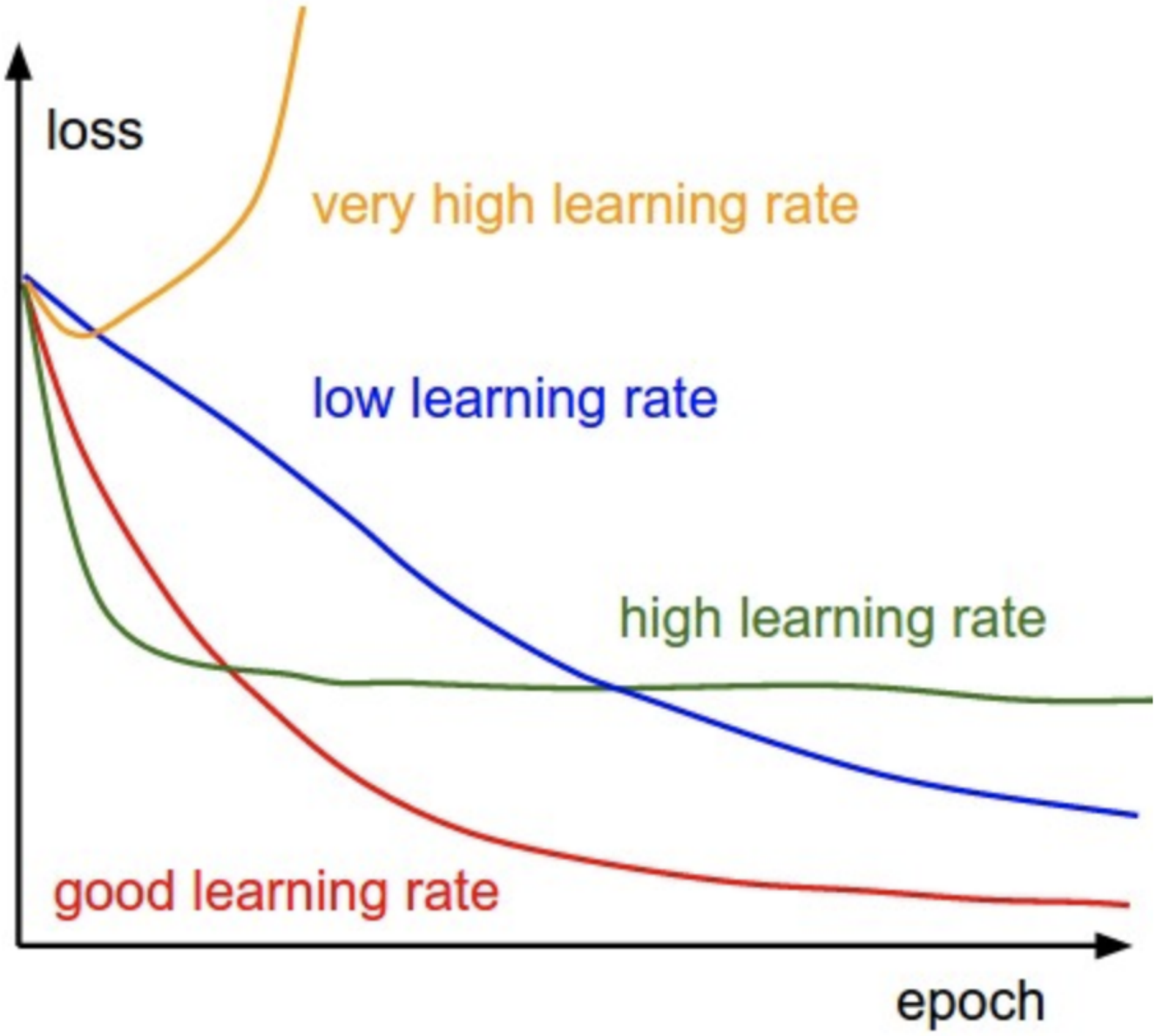
One of the most common optimization algorithms is Stochastic Gradient Descent (SGD). The hyperparameters that can be optimized in SGD are `learning rate`, `momentum`, `decay` and `nesterov`.

`Learning rate` controls the weight at the end of each batch, and `momentum` controls how much to let the previous update influence the current weight update. `Decay` indicates the learning rate decay over each update, and `nesterov` takes the value "True" or "False" depending on if we want to apply Nesterov momentum.

Typical values for those hyperparameters are lr=0.01, decay=1e-6, momentum=0.9, and nesterov=True.

The learning rate hyperparameter goes into the `optimizer` function which we will see below. Keras has a default learning rate scheduler in the `SGD` optimizer that decreases the learning rate during the stochastic gradient descent optimization algorithm. The learning rate is decreased according to this formula:

$$lr = lr \times 1/(1 + decay * epoch)$$




Source: http://cs231n.github.io/neural-networks-3

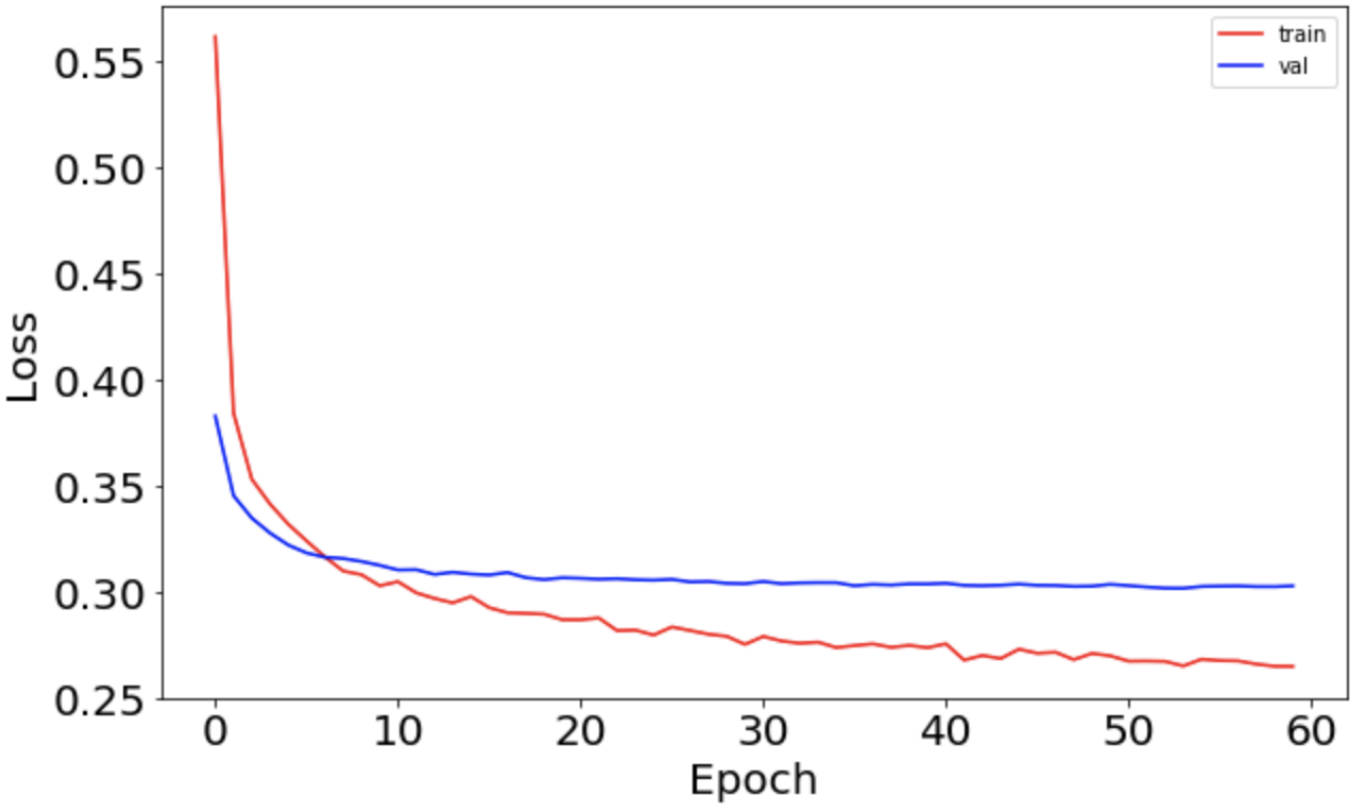
Let's implement a learning rate adaptation schedule in `Keras`. We'll start with SGD and a learning rate value of 0.1. We will then train the model for 60 epochs and set the decay argument to 0.0016 (0.1/60). We also include a momentum value of 0.8 since that seems to work well when using an adaptive learning rate.

Next, we build the architecture of the neural network:

We can now run the model and see how well it performs. This took around 20 minutes on my machine and may be faster or slower on yours depending on your machine.

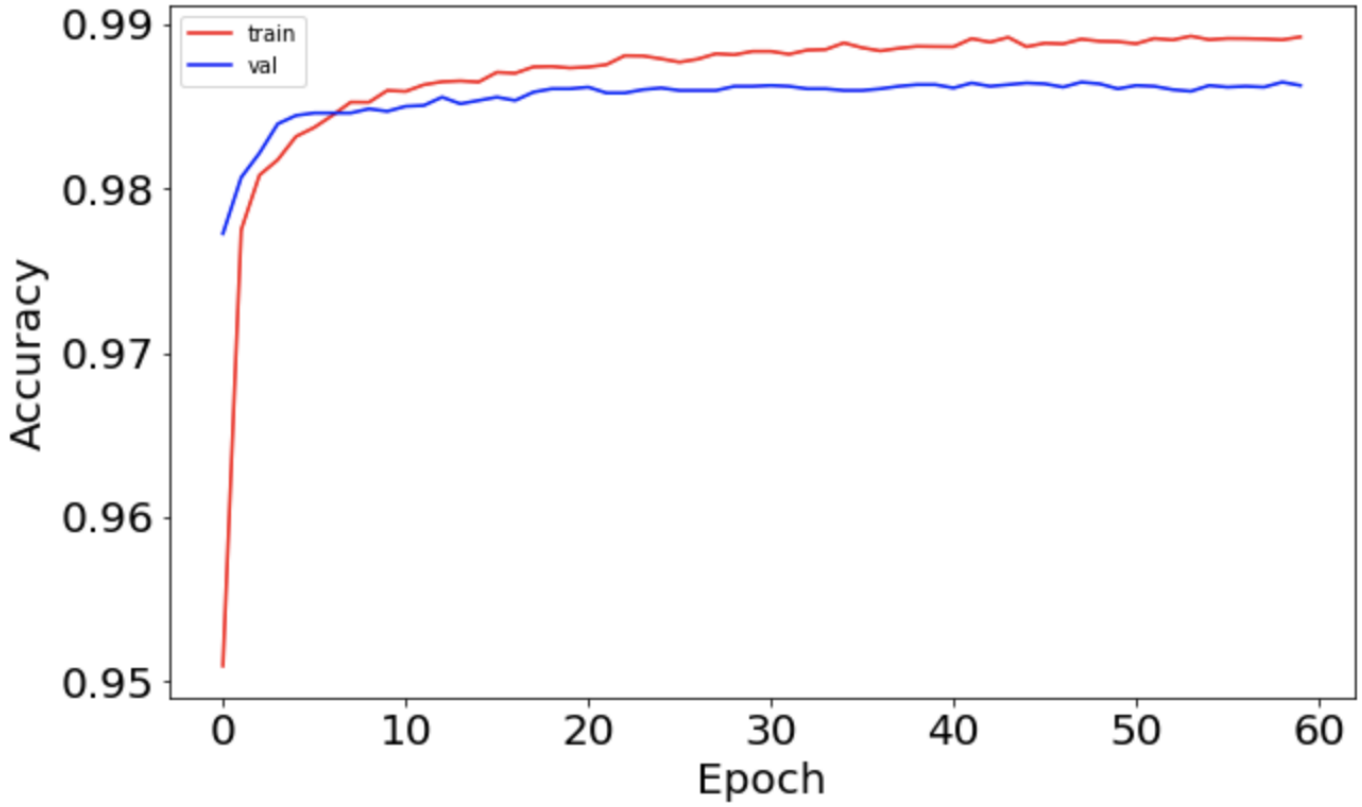
After it has finished running, we can plot the accuracy and loss function as a function of epochs for the training and test sets to see how the network performed.

The loss function plot looks as follows:




Loss as a function of epochs.

And this is the accuracy:




We will now look at applying a customized learning rate.

#### Apply a custom learning rate change using `LearningRateScheduler`

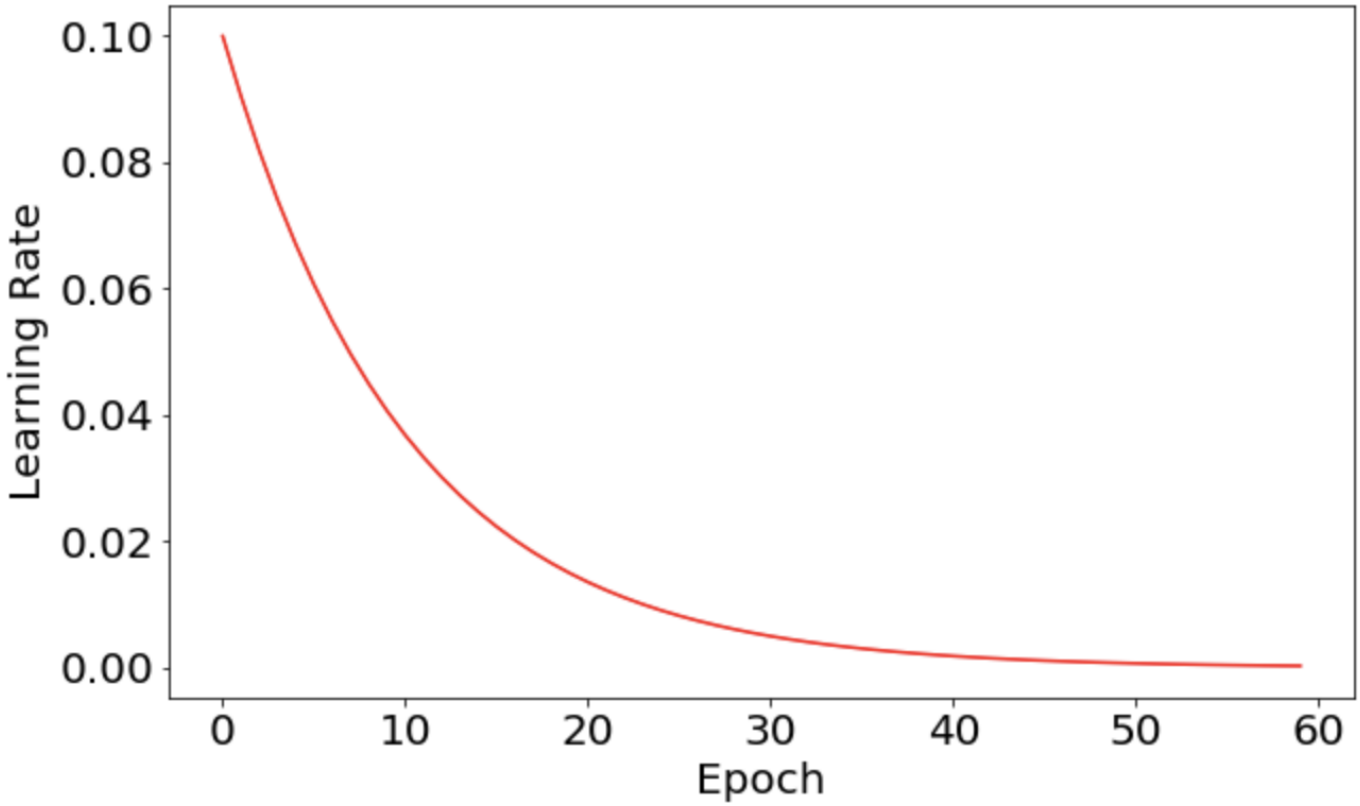
Write a function that performs the exponential learning rate decay as indicated by the following formula:

$lr=lr0 \times e\text{^}(-kt)$

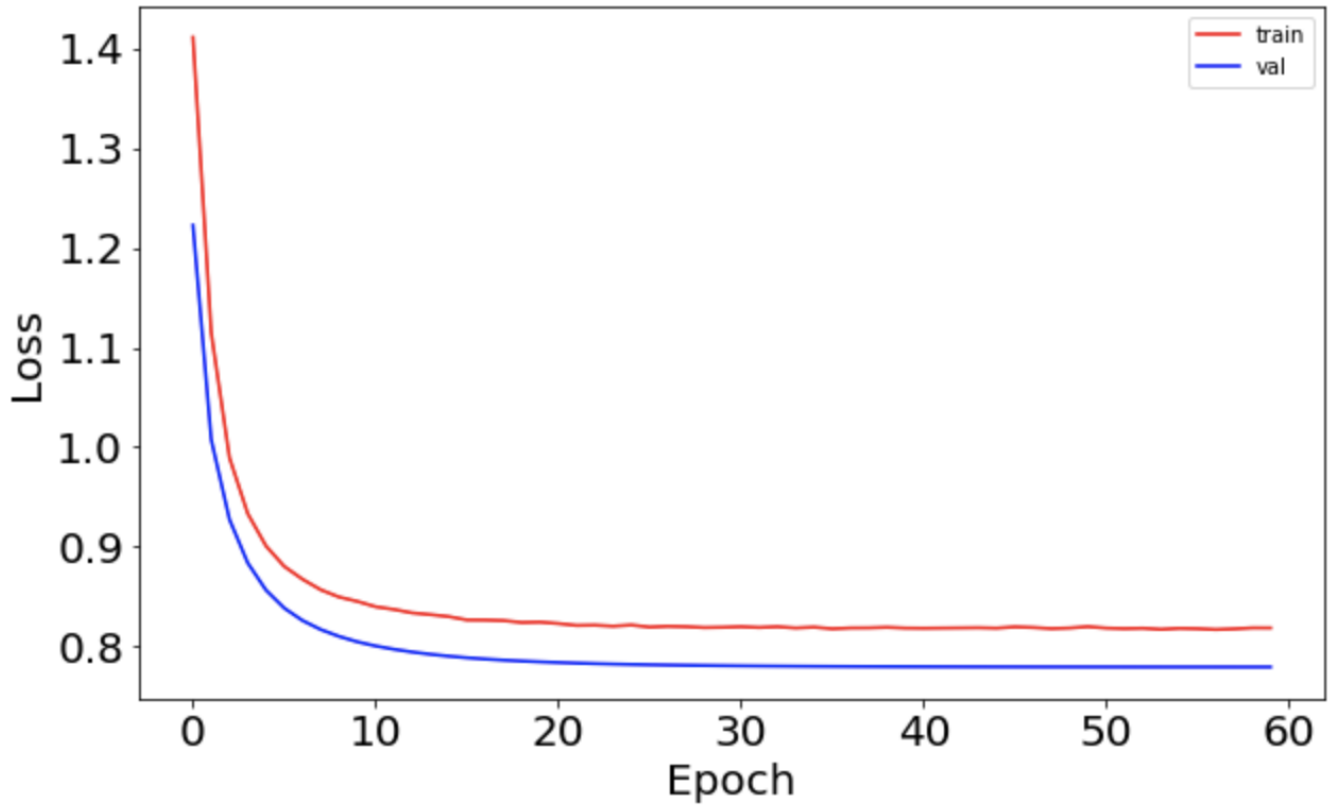
This is very similar to before so I will do this in one code block and describe the differences.

We see here that the only thing that has changed here is the presence of the `exp_decay` function that we defined, and its use in the `LearningRateScheduler` function. Notice we also chose to add a few callbacks to our model this time.

We can now plot the learning rate and loss functions as functions of the number of epochs. The learning rate plot is incredibly smooth as it follows our predefined exponentially decaying function.




The loss function also looks smoother now as compared to before.



This shows you that developing a learning rate scheduler can be a helpful way to improve neural network performance.

#### Step 3 — Choosing an `optimizer` and a `loss function`

When constructing a model and using it to make our predictions, for example, to assign label scores to images (“cat”, “plane”, etc), we want to measure our success or failure by defining a “loss” function (or objective function). The goal of optimization is to efficiently calculate the parameters/weights that minimize this loss function. `keras` provides various types of loss functions.

Sometimes the “loss” function measures the “distance”. We can define this “distance” between two data points in various ways suitable to the problem or dataset. The distance used depends on the data type and the specific problem that is being tackled. For example, in natural language processing (which analyses textual data), the Hamming distance is much more common to use.

**Distance**

- Euclidean
- Manhattan
- others such as Hamming which measures distances between strings, for example. The Hamming distance of “carolin” and “cathrin” is 3.

**Loss functions**

- MSE (for regression)
- categorical cross-entropy (for classification)
- binary cross entropy (for classification)

### Step 4 — Deciding on the `batch size` and `number of epochs`

The **batch size** defines the number of samples that will be propagated through the network.

For instance, let's say you have 1000 training samples and you want to set up a `batch_size` equal to 100. The algorithm takes the first 100 samples (from 1st to 100th) from the training dataset and trains the network. Next, it takes the second 100 samples (from 101st to 200th) and trains the network again. We can keep doing this procedure until we have propagated all samples through the network.

Advantages of using a batch size < number of all samples:

- It requires less memory. Since you train the network using fewer samples, the overall training procedure requires less memory. That's especially important if you are not able to fit the whole dataset in your machine's memory.
- Typically networks train faster with mini-batches. That's because we update the weights after each propagation.

Disadvantages of using a batch size < number of all samples:

- The smaller the batch the less accurate the estimate of the gradient will be.

The **number of epochs** is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset.

One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches.

There are no hard and fast rules for selecting batch sizes or the number of epochs, and there is no guarantee that increasing the number of epochs provides a better result than a lesser number.

#### Step 5 — Random restarts

This method does not seem to have an implementation in `keras`. It can be done easily by altering `keras.callbacks.LearningRateScheduler`. I will leave this as an exercise for the reader, but it essentially involves resetting the learning rate after a specified number of epochs for a finite number of times.

## Tuning Hyperparameters using Cross-Validation

Now instead of trying different values by hand, we will use GridSearchCV from Scikit-Learn to try out several values for our hyperparameters and compare the results.

To do cross-validation with `keras` we will use the wrappers for the Scikit-Learn API. They provide a way to use Sequential Keras models (single-input only) as part of your Scikit-Learn workflow.

There are two wrappers available:

`keras.wrappers.scikit_learn.KerasClassifier(build_fn=None, **sk_params)`, which implements the Scikit-Learn classifier interface,

`keras.wrappers.scikit_learn.KerasRegressor(build_fn=None, **sk_params)`, which implements the Scikit-Learn regressor interface.

### Trying Different Weight Initializations

The first hyperparameter we will try to optimize via cross-validation is different weight initializations.

The results from our GridSearch are:

```
Best Accuracy for 0.9689333333333333 using {'init_mode': 'lecun_uniform'}
 mean=0.9647, std=0.001438 using {'init_mode': 'uniform'}
 mean=0.9689, std=0.001044 using {'init_mode': 'lecun_uniform'}
 mean=0.9651, std=0.001515 using {'init_mode': 'normal'}
 mean=0.1124, std=0.002416 using {'init_mode': 'zero'}
 mean=0.9657, std=0.0005104 using {'init_mode': 'glorot_normal'}
 mean=0.9687, std=0.0008436 using {'init_mode': 'glorot_uniform'}
 mean=0.9681, std=0.002145 using {'init_mode': 'he_normal'}
 mean=0.9685, std=0.001952 using {'init_mode': 'he_uniform'}
```

We see that the best results are obtained either from the model using lecun_uniform initialization or glorot_uniform initialization and that we can achieve close to 97% accuracy with our network.

#### Save Your Neural Network Model to JSON

The Hierarchical Data Format (HDF5) is a data storage format for storing large arrays of data including values for the weights in a neural network.

You can install HDF5 Python module: `pip install h5py`

Keras gives you the ability to describe and save any model using the JSON format.

#### Cross-validation with more than one hyperparameters

Usually, we are not interested in looking at how just one parameter changes, but how multiple parameter changes can affect our results. We can do cross-validation with more than one parameters simultaneously, effectively trying out combinations of them.

**Note: Cross-validation in neural networks is computationally expensive**. Think before you experiment! Multiply the number of features you are validating on to see how many combinations there are. Each combination is evaluated using the *k*-fold cross-validation (*k* is a parameter we choose).

For example, we can choose to search for different values of:

- batch size
- number of epochs
- initialization mode

The choices are specified into a dictionary and passed to GridSearchCV.

We will now perform a GridSearch for `batch size`, `number of epochs` and `initializer` combined.

One last question before we end: what do we do if the number of parameters and the number of values we have to cycle through in our GridSearchCV is particularly large?

This can be a particularly troublesome problem — imagine a situation where there are 5 parameters being selected for and 10 potential values that we have selected for each parameter. The number of unique combinations of this is $10^5$, which means we would have to train a ridiculously large number of networks. Clearly, it would be insanity to actually do it this way, so it is common to use RandomizedCV as an alternative.

RandomizedCV allows us to specify all of our potential parameters, and then for each fold in the cross-validation, it selects a random subset of parameters to use for the current model. In the end, the user can select the optimal set of parameters and use these as an approximate solution.