## Course: Artificial Intelligence and Machine Learning Code: 20CS51I

# WEEK - 10: KERAS

- **Keras**
  - **Callbacks**
  - **Commonly used callbacks**
- **Monitor neural network performance with TensorBoard**
  - **TensorBoard Basics**
  - **TensorBoard Setup**

- **Understand Model Behaviour During Training**
- **Reduce overfitting with Dropout Layer**
- **How to save trained model Local deployment with TensorFlow Model Server**

## Session No. 7

# Keras:

## Callbacks

A callback is an object that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc).

You can use callbacks to:

- Write TensorBoard logs after every batch of training to monitor your metrics
- Periodically save your model to disk
- Do early stopping
- Get a view on internal states and statistics of a model during training.

### Usage of callbacks via the built-in `fit()` loop

You can pass a list of callbacks (as the keyword argument `callbacks`) to the `.fit()` method of a model:

```
my_callbacks = [
```

```
    tf.keras.callbacks.EarlyStopping(patience=2),
    tf.keras.callbacks.ModelCheckpoint(filepath='model.{epoch:02d}-{val_loss:.2f}.h5'),
    tf.keras.callbacks.TensorBoard(log_dir='./logs'),
]
model.fit(dataset, epochs=10, callbacks=my_callbacks)
```

The relevant methods of the callbacks will then be called at each stage of the training.

Commonly used Keras Callbacks APIs with the help of some examples.

1. `EarlyStopping`: a callback designed for early stopping.
2. `CSVLogger`: a callback streams epoch results to a CSV file.
3. `ModelCheckpoint` : a callback to save the Keras model or model weight during training
4. `ReduceLROnPlateau` : a callback to reduce the learning rate when a metric has stopped improving.
5. `LearningRateScheduler`: a callback for **learning rate schedules**.
6. `LambdaCallback`: a callback for creating custom callbacks on-the-fly.

## 1. EarlyStopping

`EarlyStopping` is a built-in callback designed for early stopping. First, let's import it and create an early stopping object:

```
from tensorflow.keras.callbacks import EarlyStoppingearly_stopping = EarlyStopping()
```

`EarlyStopping()` has a few options and by default:

- `monitor='val_loss'`: to use validation loss as performance measure to terminate the training.

- `patience=0`: is the number of epochs with no improvement. The value `0` means the training is terminated as soon as the performance measure gets worse from one epoch to the next.

Next, we just need to pass the callback object to `model.fit()` method.

```
history = model.fit(
    X_train,
    y_train,
    epochs=50,
    validation_split=0.20,
    batch_size=64,
    verbose=2,
    callbacks=[early_stopping]
)
```

You can see that `early_stopping` get passed in a list to the `callbacks` argument. It is a list because in practice we might be passing a number of callbacks for performing different tasks, for example, debugging and learning rate schedules.

By executing the statement, you should get an output like below:

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/50
8000/8000 - 6s - loss: 1.5632 - accuracy: 0.5504 - val_loss: 1.1315 -
val_accuracy: 0.6605
......
......
Epoch 10/50
8000/8000 - 2s - loss: 0.5283 - accuracy: 0.8213 - val_loss: 0.5539 -
val_accuracy: 0.8170
Epoch 11/50
8000/8000 - 2s - loss: 0.5141 - accuracy: 0.8281 - val_loss: 0.5644 -
val_accuracy: 0.7990
```

The training gets terminated at Epoch 11 due to the increase of `val_loss` value and that is exactly the conditions `monitor='val_loss'` and `patience=0`.

It's often more convenient to look at a plot, let's run `plot_metric(history, 'loss')` to get a clear picture. In the below graph, validation loss is shown in orange and it's clear that validation error increases at Epoch 11.

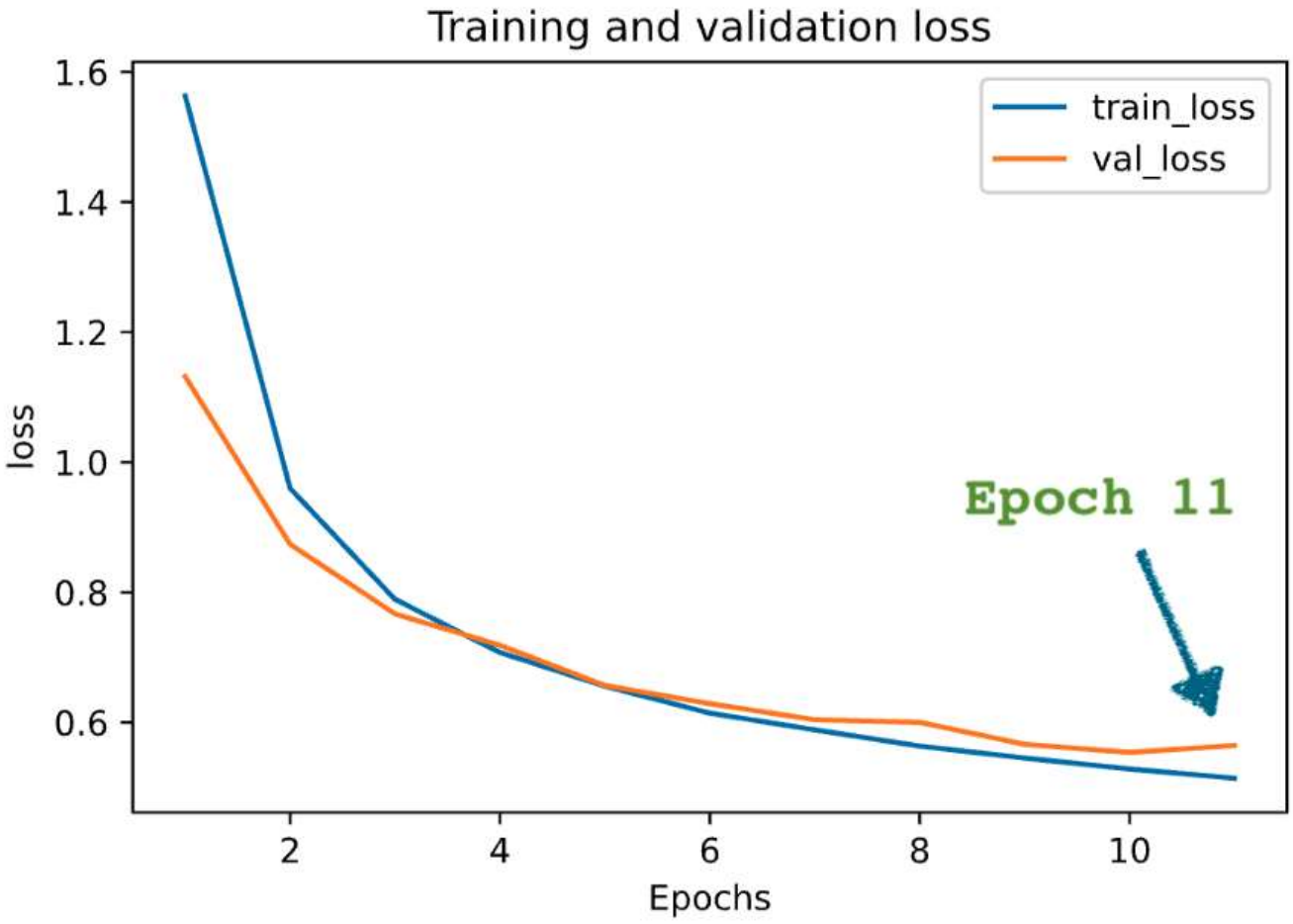



Image made by author (Please check out notebook)

### Arguments

Apart from the options `monitor` and `patience` we mentioned early, the other 2 options `min_delta` and `mode` are likely to be used quite often.

```
EarlyStopping(
    monitor='val_loss',
    patience=0,
    min_delta=0,
    mode='auto'
)
```

- `monitor='val_loss'`: to use validation loss as performance measure to terminate the training.

- `patience=0`: is the number of epochs with no improvement. The value `0` means the training is terminated as soon as the performance measure gets worse from one epoch to the next.

- **`min_delta`**: Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than `min_delta`, will count as no improvement.

- **`mode='auto'`**: Should be one of `auto`, `min` or `max`. In `'min'` mode, training will stop when the quantity monitored has stopped decreasing; in `'max'` mode it will stop when the quantity monitored has stopped increasing; in `'auto'` mode, the direction is automatically inferred from the name of the monitored quantity.

And here is an example of a customized early stopping:

```
custom_early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=3,
    min_delta=0.001,
    mode='max'
)
```

`monitor='val_accuracy'` to use **validation accuracy** as performance measure to terminate the training. `patience=3` means the training is terminated as soon as 3 epochs with no improvement. `min_delta=0.001` means the validation accuracy has to improve by at least 0.001 for it to count as an improvement. `mode='max'` means it will stop when the quantity monitored has stopped increasing.

Let's go ahead, run it with the customized early stopping, and plot the accuracy.

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/50
......
Epoch 12/50
8000/8000 - 2s - loss: 0.5043 - accuracy: 0.8290 - val_loss: 0.5311 -
val_accuracy: 0.8250
Epoch 13/50
8000/8000 - 3s - loss: 0.4936 - accuracy: 0.8332 - val_loss: 0.5310 -
val_accuracy: 0.8155
Epoch 14/50
8000/8000 - 2s - loss: 0.4835 - accuracy: 0.8353 - val_loss: 0.5157 -
```

```
val_accuracy: 0.8245
Epoch 15/50
8000/8000 - 2s - loss: 0.4757 - accuracy: 0.8397 - val_loss: 0.5299 -
val_accuracy: 0.8060
```

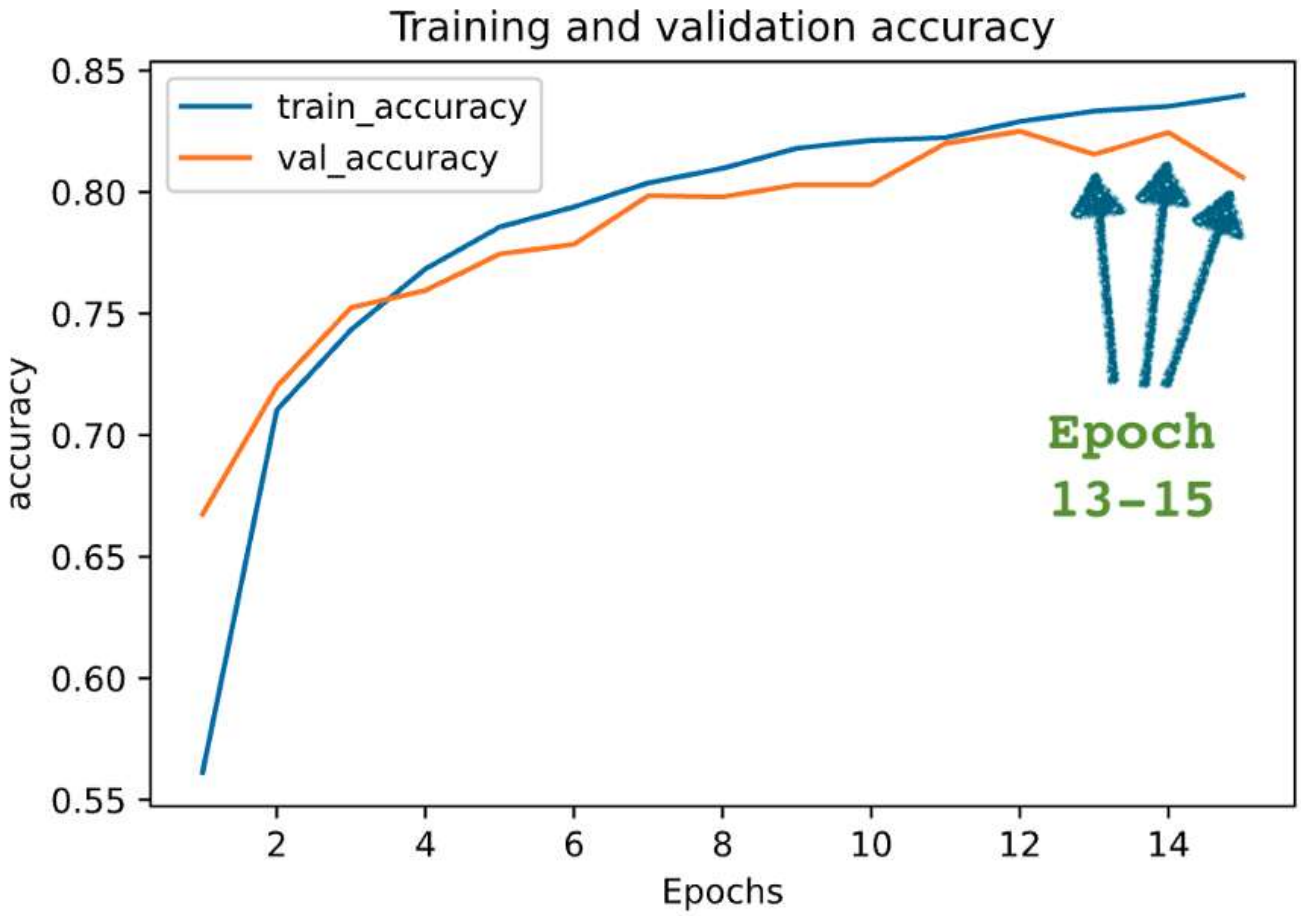



Image made by author (Please check out notebook)

This time, the training gets terminated at Epoch 15 as there are 3 epochs with no improvement on validation accuracy (It has to be ≥ 0.001 to count as an improvement).

For more about early stopping, please check out this article:

## 2. CSVLogger

`CSVLogger` is a callback that streams epoch results to a CSV file. First, let's import it and create a `CSVLogger` object:

```
from tensorflow.keras.callbacks import CSVLoggercsv_log =
CSVLogger("results.csv")
```

Next, we just need to pass the **csv_log** object to `model.fit()` method.

```
history_csv_logger = model.fit(
    X_train,
    y_train,
    epochs=10,
    validation_split=0.20,
    batch_size=64,
    verbose=2,
    callbacks=[csv_log]
)
```

Once the training is completed, we can view the information in the CSV file

```
import pandas as pdpd.read_csv("results.csv", index_col='epoch')
```

| epoch | accuracy | loss | val_accuracy | val_loss |
|---|---|---|---|---|
| 0 | 0.540375 | 1.632193 | 0.6285 | 1.190760 |
| 1 | 0.688250 | 0.999455 | 0.6965 | 0.903381 |
| 2 | 0.728125 | 0.818999 | 0.7280 | 0.791272 |
| 3 | 0.755125 | 0.733289 | 0.7595 | 0.725264 |
| 4 | 0.772500 | 0.678768 | 0.7810 | 0.681306 |
| 5 | 0.789125 | 0.638452 | 0.7860 | 0.648267 |
| 6 | 0.796750 | 0.606171 | 0.7900 | 0.620067 |
| 7 | 0.806375 | 0.579147 | 0.7780 | 0.625771 |
| 8 | 0.810000 | 0.559040 | 0.7945 | 0.593174 |
| 9 | 0.816375 | 0.541813 | 0.8065 | 0.569976 |

Arguments

Apart from the compulsory argument `filename`, the other 2 options `separator` and `append` are likely to be used quite often.

```
CSVLogger(filename, separator=',', append=False)
```

- `filename` : is the filename of the CSV file.
- **separator**: string used to separate elements in the CSV file.
- **append**: is boolean and default to `False`, `True` append if file exists (useful for continuing training). `False`: overwrite existing file.

## 3. ModelCheckpoint

`ModelCheckpoint` is a callback to save the Keras model or model weight during training, so the model or weights can be loaded later to continue the training from the state saved.

First, let's import it and create a `ModelCheckpoint` object:

```
from tensorflow.keras.callbacks import ModelCheckpointcheckpoint_path = 'model_checkpoints/'checkpoint = ModelCheckpoint(
    filepath=checkpoint_path,
    save_freq='epoch',
    save_weights_only=True,
    verbose=1
)
```

Next, let's pass the `checkpoint` object to `model.fit()` method for training.

```
history_checkpoint = model.fit(
    X_train,
    y_train,
    epochs=10,
    validation_split=0.20,
    batch_size=64,
    verbose=2,
    callbacks=[checkpoint]
)
```

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/10

Epoch 00001: saving model to model_checkpoints/
8000/8000 - 3s - loss: 1.5483 - accuracy: 0.5533 - val_loss: 1.1100 - val_accuracy: 0.6785
Epoch 2/10

Epoch 00002: saving model to model_checkpoints/
8000/8000 - 1s - loss: 0.9443 - accuracy: 0.7212 - val_loss: 0.8616 - val_accuracy: 0.7140
Epoch 3/10

Epoch 00003: saving model to model_checkpoints/
8000/8000 - 1s - loss: 0.7846 - accuracy: 0.7509 - val_loss: 0.7594 - val_accuracy: 0.7530
Epoch 4/10

Epoch 00004: saving model to model_checkpoints/
8000/8000 - 1s - loss: 0.7058 - accuracy: 0.7742 - val_loss: 0.7243 - val_accuracy: 0.7405
Epoch 5/10

Epoch 00005: saving model to model_checkpoints/
8000/8000 - 1s - loss: 0.6549 - accuracy: 0.7880 - val_loss: 0.6744 - val_accuracy: 0.7640
```

We should be able to see the above printout during the training.

Once the training is completed, we can get the test accuracy by running:

```
>>> get_test_accuracy(model, X_test, y_test)
accuracy: 0.779
```

### Loading weights

Let's create a new model `new_model` to demonstrate how loading weights work. And by running `get_test_accuracy(new_model, X_test, y_test)`, we get the test accuracy **0.086** for a model without loading any trained weights.

```
# Create a new model
>>> new_model = create_model()# Without loading weight
>>> get_test_accuracy(new_model, X_test, y_test)
accuracy: 0.086
```

Next, let's load weights with `load_weights('model_checkpoints/')` and get its test accuracy again. This time, we should be able to see the same accuracy as we trained the model `model` .

```
# Load weights
>>> new_model.load_weights('model_checkpoints/')
>>> get_test_accuracy(new_model, X_test, y_test)
accuracy: 0.779
```

Arguments

Below are the commonly used arguments you should know when using `ModelCheckpoint` callback

- `filepath`: string or `PathLike`, the path to save the model file. `filepath` can contain named formatting options, for example, if `filepath` is `weights.{epoch:02d}-{val_loss:.2f}`, then the model checkpoints will be saved with the epoch number and the validation loss in the filename.
- `save_freq`: `'epoch'` or integer. When using `'epoch'`, the callback saves the model after each epoch. When using integer, the callback saves the model at end of this many batches.
- `save_weights_only`: if `True`, then only the model's weights will be saved (`model.save_weights(filepath)`), else the full model is saved (`model.save(filepath)`).

## 4. ReduceLROnPlateau

`ReduceLROnPlateau` is a callback to reduce the learning rate when a metric has stopped improving. This callback monitors a quantity and if no improvement is seen for a `patience` number of epochs, the learning rate is reduced by `factor` value (`new_lr = lr * factor`). Let's see how this works with the help of an example.

First, let's import it and create a `ReduceLROnPlateau` object:

```
from tensorflow.keras.callbacks import ReduceLROnPlateaureduce_lr =
ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.2,
    patience=2,
    min_lr=0.001,
    verbose=2
)
```

`monitor='val_loss'` to use **validation loss** as performance measure to reduce the learning rate. `patience=2` means the learning rate is reduced as soon as 2 epochs with no improvement. `min_delta=0.001` means the validation loss has to improve by at least 0.001 for it to count as an improvement. `factor=0.2` means the new learning rate will be reduced as `new_lr = lr * factor`.

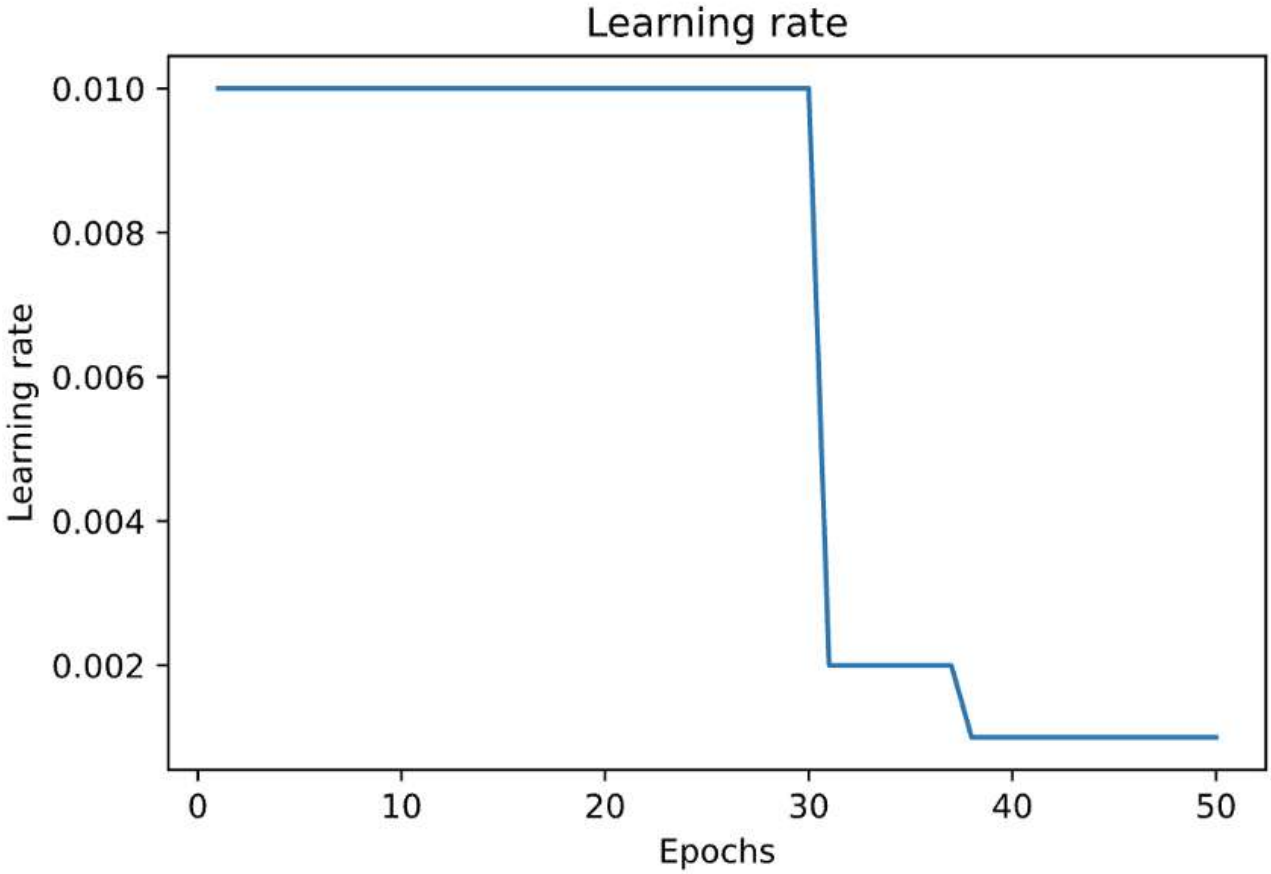
Let's train the model with the `reduce_lr` callback

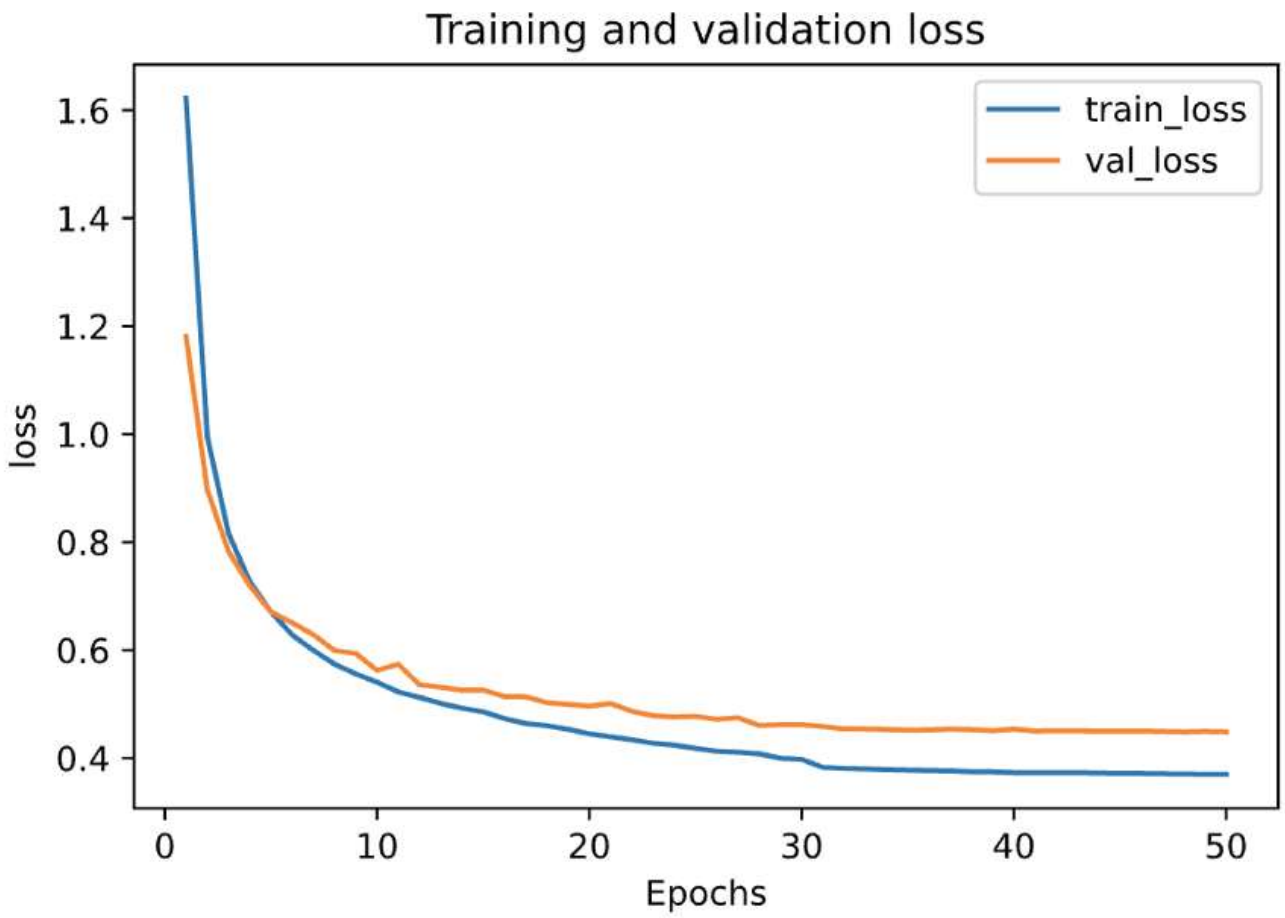
```
history_reduce_lr = model.fit(
    X_train,
    y_train,
    epochs=50,
    validation_split=0.20,
    batch_size=64,
    verbose=2,
    callbacks=[reduce_lr]
)
```

```
Epoch 27/50
8000/8000 - 1s - loss: 0.4108 - accuracy: 0.8624 - val_loss: 0.4747 - val_accuracy: 0.8325
Epoch 28/50
8000/8000 - 1s - loss: 0.4079 - accuracy: 0.8618 - val_loss: 0.4603 - val_accuracy: 0.8430
Epoch 29/50
8000/8000 - 1s - loss: 0.3994 - accuracy: 0.8656 - val_loss: 0.4617 - val_accuracy: 0.8360
Epoch 30/50

Epoch 00030: ReduceLROnPlateau reducing learning rate to 0.0019999999552965165.
8000/8000 - 1s - loss: 0.3975 - accuracy: 0.8641 - val_loss: 0.4618 - val_accuracy: 0.8405
Epoch 31/50
8000/8000 - 1s - loss: 0.3826 - accuracy: 0.8712 - val_loss: 0.4584 - val_accuracy: 0.8400
Epoch 32/50
8000/8000 - 1s - loss: 0.3807 - accuracy: 0.8721 - val_loss: 0.4539 - val_accuracy: 0.8420
Epoch 33/50
8000/8000 - 1s - loss: 0.3799 - accuracy: 0.8714 - val_loss: 0.4538 - val_accuracy: 0.8455
Epoch 34/50
8000/8000 - 1s - loss: 0.3786 - accuracy: 0.8714 - val_loss: 0.4529 - val_accuracy: 0.8435
Epoch 35/50
8000/8000 - 1s - loss: 0.3778 - accuracy: 0.8726 - val_loss: 0.4516 - val_accuracy: 0.8440
Epoch 36/50
8000/8000 - 1s - loss: 0.3769 - accuracy: 0.8725 - val_loss: 0.4521 - val_accuracy: 0.8430
Epoch 37/50

Epoch 00037: ReduceLROnPlateau reducing learning rate to 0.001.
8000/8000 - 1s - loss: 0.3763 - accuracy: 0.8730 - val_loss: 0.4536 - val_accuracy: 0.8430
Epoch 38/50
8000/8000 - 1s - loss: 0.3746 - accuracy: 0.8733 - val_loss: 0.4526 - val_accuracy: 0.8445
Epoch 39/50
```

You should get an output like above. In the above output,
the `ReduceLROnPlateau` callback has been triggered at Epoch 30 and 37.

Let's plot the learning rate and loss to get a clear picture.






Arguments

Below are the commonly used arguments you should know when using `ReduceLROnPlateau` callback

- `monitor='val_loss'`: to use validation loss as performance measure to reduce the learning rate.
- `factor`: the factor by which the learning rate will be reduced. `new_lr = lr * factor`.
- `patience`: is the number of epochs with no improvement.
- `min_delta`: Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than `min_delta`, will count as no improvement.

- `mode='auto'`: Should be one of `auto`, `min` or `max`. In `'min'` mode, the learning rate will be reduced when the quantity monitored has stopped decreasing; in `'max'` mode learning rate will be reduced when the quantity monitored has stopped increasing; in `'auto'` mode, the direction is automatically inferred from the name of the monitored quantity.

- `min_lr`: lower bound on the learning rate.

## 5. LearningRateScheduler

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**. `LearningRateScheduler` is a built-in callback for **learning rate schedules**.

Let's see how `LearningRateScheduler` works with the help of an example.

First, let's import it and create a schedule function:

```
from tensorflow.keras.callbacks import LearningRateSchedulerdef
lr_decay(epoch, lr):
    if epoch != 0 and epoch % 5 == 0:
        return lr * 0.2
    return lr
```

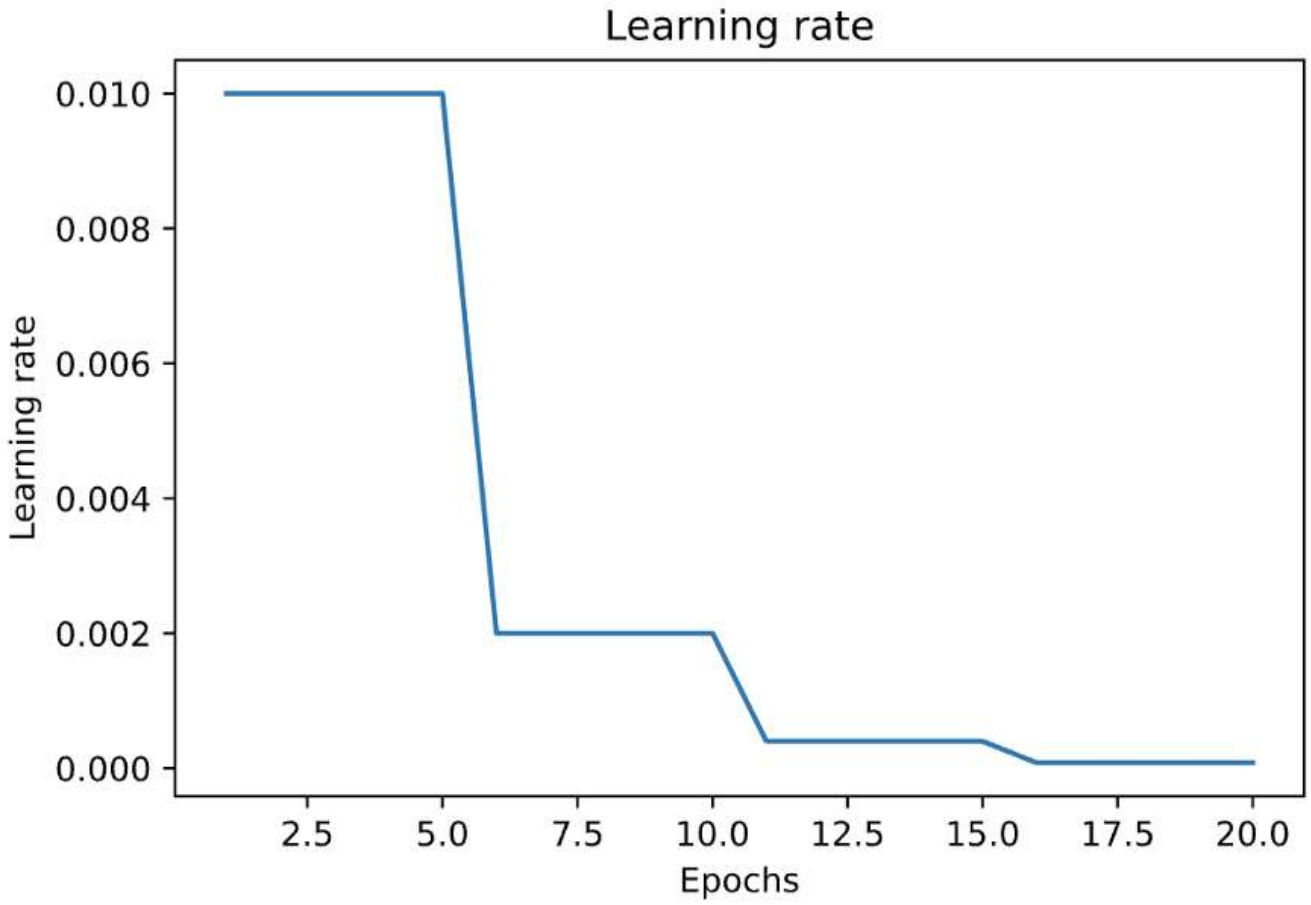
`lr_decay()` takes 2 arguments `epoch` (current epoch) &`lr`( current learning rate), and returns a new learning rate. Our`lr_decay()` function will reduce the learning rate by a factor of 0.2 in every 5 epochs.

Let's train the model with the `reduce_lr` callback

```
history_lr_schedule = model.fit(
    X_train,
    y_train,
    epochs=20,
    validation_split=0.20,
    batch_size=64,
    verbose=2,
    callbacks=[LearningRateScheduler(lr_decay, verbose=1)]
)
```

```
Train on 8000 samples, validate on 2000 samples

Epoch 00001: LearningRateScheduler reducing learning rate to 0.009999999776482582.
Epoch 1/20
8000/8000 - 3s - loss: 1.5567 - accuracy: 0.5614 - val_loss: 1.1245 - val_accuracy: 0.6720

Epoch 00002: LearningRateScheduler reducing learning rate to 0.009999999776482582.
Epoch 2/20
8000/8000 - 1s - loss: 0.9460 - accuracy: 0.7106 - val_loss: 0.8574 - val_accuracy: 0.7265

Epoch 00003: LearningRateScheduler reducing learning rate to 0.009999999776482582.
Epoch 3/20
8000/8000 - 1s - loss: 0.7757 - accuracy: 0.7487 - val_loss: 0.7673 - val_accuracy: 0.7380

Epoch 00004: LearningRateScheduler reducing learning rate to 0.009999999776482582.
Epoch 4/20
8000/8000 - 1s - loss: 0.6925 - accuracy: 0.7740 - val_loss: 0.7008 - val_accuracy: 0.7545

Epoch 00005: LearningRateScheduler reducing learning rate to 0.009999999776482582.
Epoch 5/20
8000/8000 - 1s - loss: 0.6433 - accuracy: 0.7851 - val_loss: 0.6497 - val_accuracy: 0.7720

Epoch 00006: LearningRateScheduler reducing learning rate to 0.0019999999552965165.
Epoch 6/20
8000/8000 - 1s - loss: 0.6098 - accuracy: 0.7996 - val_loss: 0.6400 - val_accuracy: 0.7825

Epoch 00007: LearningRateScheduler reducing learning rate to 0.0019999999862164259.
```

You should get an output like above. And below is the plot of the learning rate.



## 6. LambdaCallback

Another useful callback is `LambdaCallback`. It is similar to `Callback` and allows us to build custom callbacks on-the-fly.

`LambdaCallback` is constructed with the following anonymous functions that will be called at the appropriate time.

- `on_epoch_begin`: called at the beginning of every epoch.
- `on_epoch_end`: called at the end of every epoch.
- `on_batch_begin`: called at the beginning of every batch.
- `on_batch_end`: called at the end of every batch.
- `on_train_begin`: called at the beginning of model training.
- `on_train_end`: called at the end of model training.

Note that the callbacks expect positional arguments, as:

- `on_epoch_begin` and `on_epoch_end` expect two positional arguments: `epoch`, `logs`
- `on_batch_begin` and `on_batch_end` expect two positional arguments: `batch`, `logs`
- `on_train_begin` and `on_train_end` expect one positional argument: `logs`

Let's see how `LambdaCallback` works with the help of an example.

First, let's import it and create 3 different `LambdaCallback`:

```
from tensorflow.keras.callbacks import LambdaCallbackepoch_callback =
LambdaCallback(
    on_epoch_begin=lambda epoch,logs: print('Starting Epoch
```

```
{}!'.format(epoch+1))
)batch_loss_callback = LambdaCallback(
    on_batch_end=lambda batch,logs: print('\n After batch {}, the loss
is {:7.2f}.'.format(batch, logs['loss']))
)train_finish_callback = LambdaCallback(
    on_train_end=lambda logs: print('Training finished!')
)
```

Let's train the model with above callbacks

```
history_lambda_callback = model.fit(
    X_train,
    y_train,
    epochs=2,                   # change epoch to 2 for demo purpose
    validation_split=0.20,
    batch_size=2000,            # change to 2000 for demo purpose
    verbose=False,
    callbacks=[epoch_callback, batch_loss_callback,
train_finish_callback]
)
```

```
Starting Epoch 1!

 After batch 0, the loss is    0.41.

 After batch 1, the loss is    0.41.

 After batch 2, the loss is    0.42.

 After batch 3, the loss is    0.39.
Starting Epoch 2!

 After batch 0, the loss is    0.40.

 After batch 1, the loss is    0.42.

 After batch 2, the loss is    0.41.

 After batch 3, the loss is    0.41.
Training finished!
```

# Monitor neural network performance with TensorBoard

Neural networks by their very nature are hard to reason about. You can't really find out how or why something happened in a neural network, because they are too complex for that. Also, there's a real art to selecting the right number of layers, the right number of neurons per layers
and which optimizer you should use.

There is however a great tool, called Tensorboard that makes things a little easier and it works with Keras, a higher level neural network library that I happen to use.

### What is tensorboard?

Tensorflow, the deep learning framework from Google comes with a great tool to debug algorithms that you created using the framework, called Tensorboard.

It hosts a website on your local machine in which you can monitor things like accuracy, cost functions
and visualize the computational graph that Tensorflow is running based on what you defined in Keras.

### Installing Tensorboard

Tensorboard is a separate tool you need to install on your computer.
You can install Tensorboard using `pip` the python package manager:

```
pip install Tensorboard
```

### Collecting run data from your Keras program

With Tensorboard installed you can start collecting data from your Keras program.
For example, if you have the following network defined:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()

model.add(Dense(10, input_shape=(784,)))
model.add(Activation('softmax'))

model.compile(optimizer='sgd', loss='categorical_cross_entropy')

model.fit(x_train, y_train, verbose=1)
```

This neural network is compiled with a standard Gradient Descent optimizer and a Categorical Cross Entropy loss function.
Finally the network is trained using a labelled dataset.

When you run this code you will find that nothing appears on screen and there's no way to know how well things are going.
This can be problematic if you're spending multiple hours training a network and discover that it leads to nothing.

Tensorboard can help solve this problem. For this you need to modify your code a little bit:

```
from time import time

from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.callbacks import TensorBoard

model = Sequential()

model.add(Dense(10, input_shape=(784,)))
model.add(Activation('softmax'))

model.compile(optimizer='sgd', loss='categorical_crossentropy')

tensorboard = TensorBoard(log_dir="logs/{}".format(time()))

model.fit(x_train, y_train, verbose=1, callbacks=[tensorboard])
```

You need to create a new `TensorBoard` instance and point it to a log directory where data should be collected.
Next you need to modify the `fit` call so that it includes the tensorboard callback.

### Monitoring progress

Now that you have a tensorboard instance hooked up you can start to monitor the program by executing
the following command in a separate terminal:

```
tensorboard --logdir=logs/
```

Notice that the logdir setting is pointing to the root of your log directory.
I told the tensorboard callback to write to a subfolder based on a timestamp.
Writing to a separate folder for each run is necessary, so you can compare different runs.
When you point Tensorboard to the root of the log folder, it will automaticall pick up all the runs you perform.

## TensorFlow ModelServer:

## Reduce overfitting with Dropout Layer:

- Keras supports dropout regularization.
- The simplest form of dropout in Keras is provided by a Dropout core layer.
- When created, the dropout rate can be specified to the layer as the probability of setting each input to the layer to zero. This is different from the definition of dropout rate from the papers, in which the rate refers to the probability of retaining an input.
- Therefore, when a dropout rate of 0.8 is suggested in a paper (retain 80%), this will, in fact, will be a dropout rate of 0.2 (set 20% of inputs to zero).
- Below is an example of creating a dropout layer with a 50% chance of setting inputs to zero.

```
layer = Dropout(0.5)
```