### TENSOR FLOW

### Popular Libraries for Deep Learning

There are specific libraries available to develop [deep learning applications](https://www.simplilearn.com/applications-of-data-science-deep-learning-ai-artificial-intelligence-article) such as:

* Keras
* TensorFlow
* DLJ4
* Theano
* Torch

### Why Use TensorFlow?

* One of TensorFlow’s best qualities is that it makes code development easy. The readily available APIs save users from rewriting some of the code that would otherwise have been time-consuming. TensorFlow speeds up the process of training a model. Additionally, the chances of errors in the program are also reduced, typically by 55 to 85 percent.
* The other important aspect is TensorFlow is highly scalable. You can write your code and then make it run either on CPU, GPU, or across a cluster of these systems for the training purpose.
* Generally, training the model is where a large part of the computation goes. Also, the process of training is repeated multiple times to solve any issues that may arise. This process leads to the consumption of more power, and therefore, you need a distributed computing. If you need to process large amounts of data, TensorFlow makes it easy by running the code in a distributed manner.
* GPUs, or graphical processing units, have become very popular. Nvidia is one of the leaders in this space. It is good at performing mathematical computations, such as matrix multiplication, and plays a significant role in deep learning. TensorFlow also has integration with C++ and Python API, making development much faster.

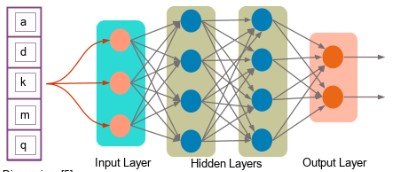
## What is TensorFlow?

TensorFlow is an open-source library that the Google Brain team developed in 2012. Python is by far the most common language that TensorFlow uses. You can import the TensorFlow library into your Python environment and perform in-depth learning development.

There is a sure way in which the program gets executed. You first create nodes, which process- the data in the form of a graph. The data gets stored in the form of tensors, and the tensor data flows to various nodes.

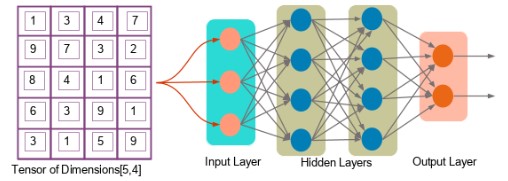
## What is a Tensor?

A tensor is a mathematical object represented as arrays of higher dimensions. These arrays of data with different sizes and ranks get fed as input to the neural network. These are the tensors.

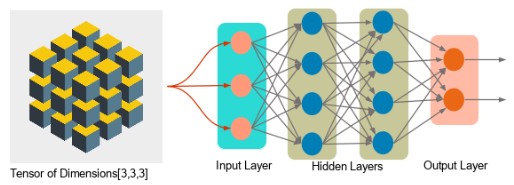


You can have arrays or vectors, which are one-dimensional, or matrices, which are two-dimensional. But tensors can be more than three, four or five-dimensional. Therefore, it helps in keeping the data very tight in one place and then performing all the analysis around that.

Let us look at an example of a tensor of [5,4] dimensions (two-dimensional).



Next, you can see a tensor of dimension [3,3,3] (three-dimensional).

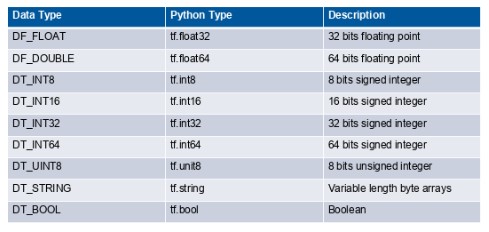


## Tensor Rank

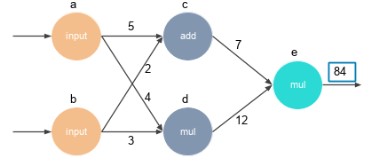
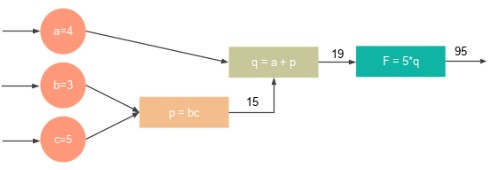
* Tensor rank is nothing but the dimension of the tensor. It starts with zero. Zero is a scalar that doesn't have multiple entries in it. It's a single value.
* For example, s = 10 is a tensor of rank 0 or a scalar.
* V = [10, 11, 12] is a tensor of rank 1 or a vector.
* M = [[1, 2, 3],[4, 5, 6]] is a tensor of rank 2 or a matrix.
* T = [[[1],[2],[3]],[[4],[5],[6]],[[7],[8],[9]]] is a tensor of rank 3 or a tensor.

## Tensor Data Type

In addition to rank and shape, tensors also have a data type. The following is a list of the data type:



## Building a Computation Graph

* Everything in TensorFlow is based on designing a computational graph. The graph has a network of nodes, with each node operating addition, multiplication, or evaluating some multivariate equation.
* The code is written to build the graph, create a session, and execute that graph. A graph has nodes that represent mathematical operations, and an edge represents tensors. In TensorFlow, a computation is explained using a data flow graph.
* Everything is an operation. Not only adding two variables but creating a variable is also an operation. Every time you assign a variable, it becomes a node. You can perform mathematical operations, such as addition and multiplication on that node.
* You start by building up these nodes and executing them in a graphical format. That is how the TensorFlow program is structured.
* 
* Here's an example that depicts how a computation graph gets created. Let's say you want to perform the following calculation: F(a,b,c) = 5(a+bc)
* The three variables a, b, and c translate into three nodes within a graph, as shown.
* 
* In TensorFlow, assigning these variables is also an operation.
* Step 1 is to build the graph by assigning the variables.

Here, the values are:

a = 4

b = 3

c = 5

* Step 2 of building the graph is to multiply b and c.

p = b\*c

* Step 3 is to add ‘a’ to ‘bc.’

q = a + p

Then, we need multiple q, and 5.

F = 5\*q

Finally, you get the result.

Here, we have six nodes. First, you define each node and then create a session to execute the node. This step, in turn, will go back and execute each of the six nodes to get those values.

**Programming Elements in TensorFlow**

Unlike other programming languages, TensorFlow allows you to assign data to three different data elements:

1. Constants
2. Variables
3. Placeholders

### Constants

Constants are parameters with values that do not change. We use the tf.constant() command to define a constant.

Example:

a = tf.constant(2.0, tf.float32)

b = tf.constant(3.0)

Print(a, b)

You cannot change the values of constants during computation. Specifying the data, the constant is optional.

### Variables

Variables allow us to add new trainable parameters to the graph. To define a variable, we use the tf.Variable() command and initialize them before running the graph in a session.

Example:

W = tf.Variable([.3],dtype=tf.float32)

b = tf.Variable([-.3],dtype=tf.float32)

x = tf.placeholder(tf.float32)

linear\_model = W\*x+b

### Placeholders

Placeholders allow us to feed data to a TensorFlow model from outside a model. It permits value to be assigned later. To define a placeholder, we use the tf.placeholder() command.

Example:

a = tf.placeholder(tf.float32)

b = a\*2

With tf.Session() assess:

result = sess.run(b,feed\_dict={a:3.0})

### Print Result

* The Print Result is somewhat similar to a variable but primarily used for feeding data from outside. Typically, when you perform a deep learning exercise, you cannot get all the data in one shot and store it in memory. That will become unmanageable. You will generally get data in batches.
* Let’s assume you want to train your model, and you have a million images to perform this training. One of the ways to accomplish this would be to create a variable, load all the images, and analyze the results. However, this might not be the best way, as the memory might slow down or there may be performance issues.
* The issue is not just storing the images. You need to perform training as well, which is an iterative process. It may need to load the images several times to train the model. It's not just the storing of million images, but also the processing that takes up memory.
* Another way of accomplishing this is by using a placeholder, where you read the data in batches. And maybe out of the million images, you get a thousand images at a time, process them and then get the next thousand and so on.
* That is the idea behind the concept of a placeholder; it is primarily used to feed your model. You read data from outside and feed it to a graph using a variable name (in the example, the variable name is feed\_dict).
* When you're running the session, you specify how you want to feed the data to your model.

### Session

Once you create a graph, you need to execute it by calling a session or using a method called run. A session is run to evaluate the nodes, which is called the TensorFlow runtime.

You can create a session by giving the command as shown:

sess = tf.Session()

Consider an example shown below:

a = tf.constant(5.0)

b = tf.constant(3.0)

c = a\*b

# Launch Session

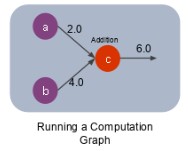
sess = tf.Session()

# Evaluate the Tensor c

print(sess.run(c))

There are three nodes: a, b, and c. We create a session and run the node ‘c’ as this is where the mathematical operation is carried out, and the result is obtained.

On running the node c, first nodes a and b will get created, and then the addition will be done at node c. We will get the result ‘6’ as shown below:



The next section of this TensorFlow tutorial focuses on how you can perform linear regression using TensorFlow.

Extra

* Tensor Flow is the second machine learning framework that Google created and used to design, build, and train deep learning models. You can use the Tensor Flow library do to numerical computations, which in itself doesn’t seem all too special, but these computations are done with data flow graphs.
* In these graphs, nodes represent mathematical operations, while the edges represent the data, which usually are multidimensional data arrays or tensors, that are communicated between these edges.
* Tensor Flow is the premier open-source deep learning framework developed and maintained by Google. Although using Tensor Flow directly can be challenging, the modern tf.keras API beings the simplicity and ease of use of Keras to the Tensor Flow project.
* Using tf.keras allows you to design, fit, evaluate, and use deep learning models to make predictions in just a few lines of code. It makes common deep learning tasks, such as classification and regression predictive modeling, accessible to average developers looking to get things done.

### What Are Keras and tf.keras?

* [Keras](https://keras.io/) is an open-source deep learning library written in Python.
* During the period of 2015-2019, developing deep learning models using mathematical libraries like TensorFlow, Theano, and PyTorch was cumbersome, requiring tens or even hundreds of lines of code to achieve the simplest tasks. The focus of these libraries was on research, flexibility, and speed, not ease of use.
* Keras was popular because the API was clean and simple, allowing standard deep learning models to be defined, fit, and evaluated in just a few lines of code.
* A secondary reason Keras took-off was because it allowed you to use any one among the range of popular deep learning mathematical libraries as the backend (e.g. used to perform the computation), such as [TensorFlow](https://github.com/tensorflow/tensorflow), [Theano](https://github.com/Theano/Theano), and later, [CNTK](https://github.com/microsoft/CNTK). This allowed the power of these libraries to be harnessed (e.g. GPUs) with a very clean and simple interface.
* In 2019, Google released a new version of their TensorFlow deep learning library (TensorFlow 2) that integrated the Keras API directly and promoted this interface as the default or standard interface for deep learning development on the platform.
* This integration is commonly referred to as the tf.keras interface or API (“tf” is short for “TensorFlow“). This is to distinguish it from the so-called standalone Keras open source project.

Standalone Keras. The standalone open source project that supports TensorFlow, Theano and CNTK backends.

tf.keras. The Keras API integrated into TensorFlow 2.

* The Keras API implementation in Keras is referred to as “tf.keras” because this is the Python idiom used when referencing the API.
* First, the TensorFlow module is imported and named “tf“; then, Keras API elements are accessed via calls to tf.keras
* Given that TensorFlow was the de facto standard backend for the Keras open source project, the integration means that a single library can now be used instead of two separate libraries. Further, the standalone Keras project now recommends all future Keras development use the *tf.keras* API.
* Keras users who use multi-backend Keras with the TensorFlow backend switch to tf.keras in TensorFlow 2.0. tf.keras is better maintained and has better integration with TensorFlow features (eager execution, distribution support and other).

### Deep Learning Model Life-Cycle

The five steps in the life-cycle are as follows:

* Define the model.
* Compile the model.
* Fit the model.
* Evaluate the model.
* Make predictions.

Define the model

* Defining the model requires that you first select the type of model that you need and then choose the architecture or network topology.
* From an API perspective, this involves defining the layers of the model, configuring each layer with a number of nodes and activation function, and connecting the layers together into a cohesive model.

Models can be defined either with the Sequential API or the Functional API, and we will take a look at this in the next section.

# define the model

model = ..

Compile the model

* Compiling the model requires that you first select a loss function that you want to optimize, such as mean squared error or cross-entropy.
* It also requires that you select an algorithm to perform the optimization procedure, typically stochastic gradient descent, or a modern variation, such as Adam. It may also require that you select any performance metrics to keep track of during the model training process.
* From an API perspective, this involves calling a function to compile the model with the chosen configuration, which will prepare the appropriate data structures required for the efficient use of the model you have defined.
* The optimizer can be specified as a string for a known optimizer class, e.g. ‘*sgd*‘ for stochastic gradient descent, or you can configure an instance of an optimizer class and use that.

#1.For a list of supported optimizers(tf.keras Optimizers)

opt = SGD(learning\_rate=0.01, momentum=0.9)

model.compile(optimizer=opt, loss='binary\_crossentropy')

#binary\_crossentropy--- for binary classification.

#sparse\_categorical\_crossentropy ----for multi-class classification.

#mse (mean squared error) ----for regression.

#2.For a list of supported loss functions(tf.keras Loss Functions)

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

#3.For a list of supported metrics(tf.keras Metrics)

#Metrics are defined as a list of strings for known metric functions or a list of functions to call to evaluate predictions.

# compile the model

model.compile(optimizer='sgd', loss='binary\_crossentropy', metrics=['accuracy'])

Fit the Model

* Fitting the model requires that you first select the training configuration, such as the number of epochs (loops through the training dataset) and the batch size (number of samples in an epoch used to estimate model error).
* Training applies the chosen optimization algorithm to minimize the chosen loss function and updates the model using the back propagation of error algorithm.
* Fitting the model is the slow part of the whole process and can take seconds to hours to days, depending on the complexity of the model, the hardware you’re using, and the size of the training dataset.
* From an API perspective, this involves calling a function to perform the training process. This function will block (not return) until the training process has finished
* While fitting the model, a progress bar will summarize the status of each epoch and the overall training process. This can be simplified to a simple report of model performance each epoch by setting the “*verbose*” argument to 2. All output can be turned off during training by setting “*verbose*” to 0.

Evaluate the Model

* Evaluating the model requires that you first choose a holdout dataset used to evaluate the model. This should be data not used in the training process so that we can get an unbiased estimate of the performance of the model when making predictions on new data.
* The speed of model evaluation is proportional to the amount of data you want to use for the evaluation, although it is much faster than training as the model is not changed.
* From an API perspective, this involves calling a function with the holdout dataset and getting a loss and perhaps other metrics that can be reported.

Make a Prediction

* Making a prediction is the final step in the life-cycle.
* It requires you have new data for which a prediction is required, e.g. where you do not have the target values.
* From an API perspective, you simply call a function to make a prediction of a class label, probability, or numerical value: whatever you designed your model to predict.
* You may want to save the model and later load it to make predictions. You may also choose to fit a model on all of the available data before you start using it.

### Sequential Model API (Simple)

* It is referred to as “*sequential*” because it involves defining a [Sequential class](https://www.tensorflow.org/api_docs/python/tf/keras/Sequential) and adding layers to the model one by one in a linear manner, from input to output.
* The sequential API is easy to use because you keep calling *model.add()* until you have added all of your layers.

### Functional Model API (Advanced)

* The functional API is more complex but is also more flexible.
* It involves explicitly connecting the output of one layer to the input of another layer. Each connection is specified.
* First, an input layer must be defined via the *Input* class, and the shape of an input sample is specified. We must retain a reference to the input layer when defining the model.
* As such, it allows for more complicated model designs, such as models that may have multiple input paths (separate vectors) and models that have multiple output paths

### How to Develop Deep Learning Models

1. Develop Multilayer Perceptron Models

* A Multilayer Perceptron model, or MLP for short, is a standard fully connected neural network model.
* It is comprised of layers of nodes where each node is connected to all outputs from the previous layer and the output of each node is connected to all inputs for nodes in the next layer.
* An MLP is created by with one or more *Dense* layers. This model is appropriate for tabular data that is data as it looks in a table or spreadsheet with one column for each variable and one row for each variable. There are three predictive modeling problems you may want to explore with an MLP; they are binary classification, multiclass classification, and regression.

1. Develop Convolutional Neural Network Models

* Convolutional Neural Networks, or CNNs for short, are a type of network designed for image input.
* They are comprised of models with [convolutional layers](https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/) that extract features (called feature maps) and [pooling layers](https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/) that distill features down to the most salient elements.
* CNNs are most well-suited to image classification tasks, although they can be used on a wide array of tasks that take images as input.

1. Develop Recurrent Neural Network Models

* Recurrent Neural Networks, or RNNs for short, are designed to operate upon sequences of data.
* They have proven to be very effective for natural language processing problems where sequences of text are provided as input to the model. RNNs have also seen some modest success for time series forecasting and speech recognition.
* The most popular type of RNN is the Long Short-Term Memory network, or LSTM for short. LSTMs can be used in a model to accept a sequence of input data and make a prediction, such as assign a class label or predict a numerical value like the next value or values in the sequence.

### How to Visualize a Deep Learning Model

* The architecture of deep learning models can quickly become large and complex.
* As such, it is important to have a clear idea of the connections and data flow in your model. This is especially important if you are using the functional API to ensure you have indeed connected the layers of the model in the way you intended.
* There are two tools you can use to visualize your model: a text description and a plot.

Model Text Description

A text description of your model can be displayed by calling the [summary() function](https://www.tensorflow.org/api_docs/python/tf/keras/Model#summary) on your model.

Model Architecture Plot

You can create a plot of your model by calling the [plot\_model() function](https://www.tensorflow.org/api_docs/python/tf/keras/utils/plot_model).

This will create an image file that contains a box and line diagram of the layers in your model.

### How to Plot Model Learning Curves

* Learning curves are a plot of neural network model performance over time, such as calculated at the end of each training epoch.
* Plots of learning curves provide insight into the learning dynamics of the model, such as whether the model is learning well, whether it is underfitting the training dataset, or whether it is overfitting the training dataset.
* First, you must update your call to the fit function to include reference to a [validation dataset](https://machinelearningmastery.com/difference-test-validation-datasets/). This is a portion of the training set not used to fit the model, and is instead used to evaluate the performance of the model during training.
* You can split the data manually and specify the *validation\_data* argument, or you can use the *validation\_split* argument and specify a percentage split of the training dataset and let the API perform the split for you.
* The fit function will return a *history* object that contains a trace of performance metrics recorded at the end of each training epoch. This includes the chosen loss function and each configured metric, such as accuracy, and each loss and metric is calculated for the training and validation datasets
* A learning curve is a plot of the loss on the training dataset and the validation dataset. We can create this plot from the *history* object using the [Matplotlib](https://matplotlib.org/) library.

### How to Save and Load Your Model

* Training and evaluating models is great, but we may want to use a model later without retraining it each time.
* This can be achieved by saving the model to file and later loading it and using it to make predictions.
* This can be achieved using the *save()* function on the model to save the model. It can be loaded later using the [load\_model() function](https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model).
* The model is saved in H5 format, an efficient array storage format. As such, you must ensure that the [h5py library](https://www.h5py.org/) is installed on your workstation.
* This can be achieved using *pip*; for example: pip install h5py

### How to Get Better Model Performance

* You will discover some of the techniques that you can use to improve the performance of your deep learning models.
* A big part of improving deep learning performance involves avoiding overfitting by slowing down the learning process or stopping the learning process at the right time.

### How to Reduce Overfitting With Dropout

* Dropout is a clever regularization method that reduces overfitting of the training dataset and makes the model more robust.
* This is achieved during training, where some number of layer outputs are randomly ignored or “*dropped out*.” This has the effect of making the layer look like – and be treated like – a layer with a different number of nodes and connectivity to the prior layer.
* Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.
* You can add dropout to your models as a new layer prior to the layer that you want to have input connections dropped-out.
* This involves adding a layer called [Dropout()](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout) that takes an argument that specifies the probability that each output from the previous to drop. E.g. 0.4 means 40 percent of inputs will be dropped each update to the model.
* You can add Dropout layers in MLP, CNN, and RNN models, although there are also specialized versions of dropout for use with CNN and RNN models that you might also want to explore.

### How to Accelerate Training With Batch Normalization

* The scale and distribution of inputs to a layer can greatly impact how easy or quickly that layer can be trained.
* This is generally why it is a good idea to scale input data prior to modeling it with a neural network model.
* Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.
* You can use batch normalization in your network by adding a batch normalization layer prior to the layer that you wish to have standardized inputs. You can use batch normalization with MLP, CNN, and RNN models.

### How to Halt Training at the Right Time With Early Stopping

* Neural networks are challenging to train.
* Too little training and the model is underfit; too much training and the model overfits the training dataset. Both cases result in a model that is less effective than it could be.
* One approach to solving this problem is to use early stopping. This involves monitoring the loss on the training dataset and a validation dataset (a subset of the training set not used to fit the model). As soon as loss for the validation set starts to show signs of overfitting, the training process can be stopped.
* Early stopping can be used with your model by first ensuring that you have a [validation dataset](https://machinelearningmastery.com/difference-test-validation-datasets/). You can define the validation dataset manually via the *validation\_data* argument to the *fit()* function, or you can use the *validation\_split* and specify the amount of the training dataset to hold back for validation.
* You can then define an EarlyStopping and instruct it on which performance measure to monitor, such as ‘*val\_loss*‘ for loss on the validation dataset, and the number of epochs to observed overfitting before taking action, e.g. 5.
* This configured [EarlyStopping](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping) callback can then be provided to the *fit()* function via the “*callbacks*” argument that takes a list of callbacks.
* This allows you to set the number of epochs to a large number and be confident that training will end as soon as the model starts overfitting. You might also like to create a learning curve to discover more insights into the learning dynamics of the run and when training was halted.

### A Gentle Introduction to Early Stopping to Avoid Overtraining Neural Networks

* A major challenge in training neural networks is how long to train them.
* Too little training will mean that the model will underfit the train and the test sets. Too much training will mean that the model will overfit the training dataset and have poor performance on the test set
* A compromise is to train on the training dataset but to stop training at the point when performance on a validation dataset starts to degrade. This simple, effective, and widely used approach to training neural networks is called early stopping.
* you will discover that stopping the training of a neural network early before it has overfit the training dataset can [reduce overfitting](https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/) and improve the generalization of deep neural networks.

This tutorial is divided into five parts; they are:

1. The Problem of Training Just Enough
2. Stop Training When Generalization Error Increases
3. How to Stop Training Early
4. Examples of Early Stopping
5. Tips for Early Stopping

## The Problem of Training Just Enough

* Training neural networks is challenging.
* When training a large network, there will be a point during training when the model will stop generalizing and start learning the statistical noise in the training dataset.
* This overfitting of the training dataset will result in an increase in generalization error, making the model less useful at making predictions on new data.
* The challenge is to train the network long enough that it is capable of learning the mapping from inputs to outputs, but not training the model so long that it overfits the training data.
* One approach to solving this problem is to treat the [number of training epochs](https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/) as a hyperparameter and train the model multiple times with different values, then select the number of epochs that result in the best performance on the train or a holdout test dataset.
* The downside of this approach is that it requires multiple models to be trained and discarded. This can be computationally inefficient and time-consuming, especially for large models trained on large datasets over days or weeks.

## Stop Training When Generalization Error Increases

* An alternative approach is to train the model once for a large number of training epochs.
* During training, the model is evaluated on a holdout validation dataset after each epoch. If the performance of the model on the validation dataset starts to degrade (e.g. loss begins to increase or accuracy begins to decrease), then the training process is stopped.
* The error measured with respect to independent data, generally called a validation set, often shows a decrease at first, followed by an increase as the network starts to over-fit. Training can therefore be stopped at the point of smallest error with respect to the validation data set
* The model at the time that training is stopped is then used and is known to have good generalization performance.
* This procedure is called “*early stopping*” and is perhaps one of the oldest and most widely used forms of neural network regularization.
* This strategy is known as early stopping. It is probably the most commonly used form of regularization in deep learning. Its popularity is due both to its effectiveness and its simplicity.
* If regularization methods like weight decay that update the loss function to encourage less complex models are considered “*explicit*” regularization, then early stopping may be thought of as a type of “*implicit*” regularization, much like using a smaller network that has less capacity.

## How to Stop Training Early

Early stopping requires that you configure your network to be under constrained, meaning that it has more capacity than is required for the problem.

When training the network, a larger number of training epochs is used than may normally be required, to give the network plenty of opportunity to fit, then begin to overfit the training dataset.

There are three elements to using early stopping; they are:

* Monitoring model performance.
* Trigger to stop training.
* The choice of model to use.

### Monitoring Performance

* The performance of the model must be monitored during training.
* This requires the choice of a dataset that is used to evaluate the model and a metric used to evaluate the model.
* It is common to split the training dataset and use a subset, such as 30%, as a validation dataset used to monitor performance of the model during training. This validation set is not used to train the model. It is also common to use the loss on a validation dataset as the metric to monitor, although you may also use prediction error in the case of regression, or accuracy in the case of classification.
* The loss of the model on the training dataset will also be available as part of the training procedure, and additional metrics may also be calculated and monitored on the training dataset.
* Performance of the model is evaluated on the validation set at the end of each epoch, which adds an additional computational cost during training. This can be reduced by evaluating the model less frequently, such as every 2, 5, or 10 training epochs.

### 2. Early Stopping Trigger

* Once a scheme for evaluating the model is selected, a trigger for stopping the training process must be chosen.
* The trigger will use a monitored performance metric to decide when to stop training. This is often the performance of the model on the holdout dataset, such as the loss.
* In the simplest case, training is stopped as soon as the performance on the validation dataset decreases as compared to the performance on the validation dataset at the prior training epoch (e.g. an increase in loss).
* More elaborate triggers may be required in practice. This is because the training of a neural network is stochastic and can be noisy. Plotted on a graph, the performance of a model on a validation dataset may go up and down many times. This means that the first sign of overfitting may not be a good place to stop training.

### 3. Model Choice

* At the time that training is halted, the model is known to have slightly worse generalization error than a model at a prior epoch.
* As such, some consideration may need to be given as to exactly which model is saved. Specifically, the training epoch from which weights in the model that are saved to file.
* This will depend on the trigger chosen to stop the training process. For example, if the trigger is a simple decrease in performance from one epoch to the next, then the weights for the model at the prior epoch will be preferred.
* If the trigger is required to observe a decrease in performance over a fixed number of epochs, then the model at the beginning of the trigger period will be preferred.
* Perhaps a simple approach is to always save the model weights if the performance of the model on a holdout dataset is better than at the previous epoch. That way, you will always have the model with the best performance on the holdout set.
* Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters

## Plot Learning Curves to Select a Trigger

Before using early stopping, it may be interesting to fit an under constrained model and monitor the performance of the model on a train and validation dataset.

Plotting the performance of the model in real-time or at the end of a long run will show how noisy the training process is with your specific model and dataset.

This may help in the choice of a trigger for early stopping.

## Monitor an Important Metric

Loss is an easy metric to monitor during training and to trigger early stopping.

The problem is that loss does not always capture what is most important about the model to you and your project.

It may be better to choose a performance metric to monitor that best defines the performance of the model in terms of the way you intend to use it. This may be the metric that you intend to use to report the performance of the model.

## Suggested Training Epochs

A problem with early stopping is that the model does not make use of all available training data.

It may be desirable to avoid overfitting and to train on all possible data, especially on problems where the amount of training data is very limited.

A recommended approach would be to treat the number of training epochs as a hyperparameter and to grid search a range of different values, perhaps using k-fold cross-validation. This will allow you to fix the number of training epochs and fit a final model on all available data.

Early stopping could be used instead. The early stopping procedure could be repeated a number of times. The epoch number at which training was stopped could be recorded. Then, the average of the epoch number across all repeats of early stopping could be used when fitting a final model on all available training data.

This process could be performed using a different split of the training set into train and validation steps each time early stopping is run.

An alternative might be to use early stopping with a validation dataset, then update the final model with further training on the held out validation set.

## Early Stopping With Cross-Validation

The k-fold cross-validation procedure is designed to estimate the generalization error of a model by repeatedly refitting and evaluating it on different subsets of a dataset.

Early stopping is designed to monitor the generalization error of one model and stop training when generalization error begins to degrade.

They are at odds because cross-validation assumes you don’t know the generalization error and early stopping is trying to give you the best model based on knowledge of generalization error.

It may be desirable to use cross-validation to estimate the performance of models with different hyperparameter values, such as learning rate or network structure, whilst also using early stopping.

In this case, if you have the resources to repeatedly evaluate the performance of the model, then perhaps the number of training epochs may also be treated as a hyperparameter to be optimized, instead of using early stopping.

Instead of using cross-validation with early stopping, early stopping may be used directly without repeated evaluation when evaluating different hyperparameter values for the model (e.g. different learning rates).

One possible point of confusion is that early stopping is sometimes referred to as “*cross-validated training*.” Further, research into early stopping that compares triggers may use cross-validation to compare the impact of different triggers.

## Overfit Validation

Repeating the early stopping procedure many times may result in the model overfitting the validation dataset.

This can happen just as easily as overfitting the training dataset.

One approach is to only use early stopping once all other hyperparameters of the model have been chosen.

Another strategy may be to use a different split of the training dataset into train and validation sets each time early stopping is used.

### A Gentle Introduction to Batch Normalization for Deep Neural Networks

* Training deep neural networks with tens of layers is challenging as they can be sensitive to the initial random weights and configuration of the learning algorithm.
* One possible reason for this difficulty is the distribution of the inputs to layers deep in the network may change after each mini-batch when the weights are updated. This can cause the learning algorithm to forever chase a moving target. This change in the distribution of inputs to layers in the network is referred to the technical name “internal covariate shift.”
* Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

## Overview

This tutorial is divided into five parts; they are:

1. Problem of Training Deep Networks
2. Standardize Layer Inputs
3. How to Standardize Layer Inputs
4. Examples of Using Batch Normalization
5. Tips for Using Batch Normalization

## Problem of Training Deep Networks

* Training deep neural networks, e.g. networks with tens of hidden layers, is challenging.
* One aspect of this challenge is that the model is updated layer-by-layer backward from the output to the input using an estimate of error that assumes the weights in the layers prior to the current layer are fixed.
* Very deep models involve the composition of several functions or layers. The gradient tells how to update each parameter, under the assumption that the other layers do not change.
* Because all layers are changed during an update, the update procedure is forever chasing a moving target.
* For example, the weights of a layer are updated given an expectation that the prior layer outputs values with a given distribution. This distribution is likely changed after the weights of the prior layer are updated.
* Training Deep Neural Networks is complicated by the fact that the distribution of each layer’s inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities.

## Standardize Layer Inputs

* Batch normalization, or batchnorm for short, is proposed as a technique to help coordinate the update of multiple layers in the model.
* Batch normalization provides an elegant way of reparametrizing almost any deep network. The reparametrization significantly reduces the problem of coordinating updates across many layers.
* It does this scaling the output of the layer, specifically by standardizing the activations of each input variable per mini-batch, such as the activations of a node from the previous layer. Recall that standardization refers to rescaling data to have a mean of zero and a standard deviation of one, e.g. a standard Gaussian.
* Batch normalization reparametrizes the model to make some units always be standardized by definition
* Standardizing the activations of the prior layer means that assumptions the subsequent layer makes about the spread and distribution of inputs during the weight update will not change, at least not dramatically. This has the effect of stabilizing and speeding-up the training process of deep neural networks.
* Batch normalization acts to standardize only the mean and variance of each unit in order to stabilize learning, but allows the relationships between units and the nonlinear statistics of a single unit to change.
* Normalizing the inputs to the layer has an effect on the training of the model, dramatically reducing the number of epochs required. It can also have a regularizing effect, reducing generalization error much like the use of activation regularization.
* Batch normalization can have a dramatic effect on optimization performance, especially for convolutional networks and networks with sigmoidal nonlinearities.
* BatchNorm impacts network training in a fundamental way: it makes the landscape of the corresponding optimization problem be significantly more smooth. This ensures, in particular, that the gradients are more predictive and thus allow for use of larger range of learning rates and faster network convergence.

## How to Standardize Layer Inputs

* Batch normalization can be implemented during training by calculating the mean and standard deviation of each input variable to a layer per mini-batch and using these statistics to perform the standardization.
* Alternately, a running average of mean and standard deviation can be maintained across mini-batches, but may result in unstable training.
* After training, the mean and standard deviation of inputs for the layer can be set as mean values observed over the training dataset.
* For small [mini-batch sizes](https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/) or mini-batches that do not contain a representative distribution of examples from the training dataset, the differences in the standardized inputs between training and inference (using the model after training) can result in noticeable differences in performance. This can be addressed with a modification of the method called Batch Renormalization (or BatchRenorm for short) that makes the estimates of the variable mean and standard deviation more stable across mini-batches.
* Batch Renormalization extends batchnorm with a per-dimension correction to ensure that the activations match between the training and inference networks.
* This standardization of inputs may be applied to input variables for the first hidden layer or to the activations from a hidden layer for deeper layers.
* In practice, it is common to allow the layer to learn two new parameters, namely a new mean and standard deviation, Beta and Gamma respectively that allow the automatic scaling and shifting of the standardized layer inputs. These parameters are learned by the model as part of the training process.
* Importantly the backpropagation algorithm is updated to operate upon the transformed inputs, and error is also used to update the new scale and shifting parameters learned by the model.
* The standardization is applied to the inputs to the layer, namely the input variables or the output of the activation function from the prior layer. Given the choice of activation function, the distribution of the inputs to the layer may be quite non-Gaussian. In this case, there may be benefit in standardizing the summed activation before the activation function in the previous layer.

## Tips for Using Batch Normalization

This section provides tips and suggestions for using batch normalization with your own neural networks.

### Use With Different Network Types

Batch normalization is a general technique that can be used to normalize the inputs to a layer.

It can be used with most network types, such as Multilayer Perceptrons, Convolutional Neural Networks and Recurrent Neural Networks

### Probably Use Before the Activation

* Batch normalization may be used on the inputs to the layer before or after the activation function in the previous layer.
* It may be more appropriate **after** the activation function if for s-shaped functions like the hyperbolic tangent and logistic function.
* It may be appropriate **before** the activation function for activations that may result in non-Gaussian distributions like the rectified linear activation function, the modern default for most network types.
* The goal of Batch Normalization is to achieve a stable distribution of activation values throughout training, and in our experiments we apply it before the nonlinearity since that is where matching the first and second moments is more likely to result in a stable distribution

### Use Large Learning Rates

* Using batch normalization makes the network more stable during training.
* This may require the use of much larger than normal learning rates, that in turn may further speed up the learning process.
* In a batch-normalized model, we have been able to achieve a training speedup from higher learning rates, with no ill side effect

### Less Sensitive to Weight Initialization

Deep neural networks can be quite sensitive to the technique used to initialize the weights prior to training.

The stability to training brought by batch normalization can make training deep networks less sensitive to the choice of weight initialization method.

### Alternate to Data Preparation

* Batch normalization could be used to standardize raw input variables that have differing scales.
* If the mean and standard deviations calculated for each input feature are calculated over the mini-batch instead of over the entire training dataset, then the batch size must be sufficiently representative of the range of each variable.
* It may not be appropriate for variables that have a data distribution that is highly non-Gaussian, in which case it might be better to perform data scaling as a pre-processing step.

### A Gentle Introduction to Dropout for Regularizing Deep Neural Networks

* Deep learning neural networks are likely to quickly overfit a training dataset with few examples.
* Ensembles of neural networks with different model configurations are known to reduce overfitting, but require the additional computational expense of training and maintaining multiple models.
* A single model can be used to simulate having a large number of different network architectures by randomly dropping out nodes during training. This is called dropout and offers a very computationally cheap and remarkably effective regularization method to [reduce overfitting and improve generalization error](https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/) in deep neural networks of all kinds.

## Overview

This tutorial is divided into five parts; they are:

1. Problem With Overfitting
2. Randomly Drop Nodes
3. How to Dropout
4. Examples of Using Dropout
5. Tips for Using Dropout Regularization

### Problem With Overfitting

* Large neural nets trained on relatively small datasets can overfit the training data.
* This has the effect of the model learning the statistical noise in the training data, which results in poor performance when the model is evaluated on new data, e.g. a test dataset. Generalization error increases due to overfitting.
* One approach to reduce overfitting is to fit all possible different neural networks on the same dataset and to average the predictions from each model. This is not feasible in practice, and can be approximated using a small collection of different models, called an ensemble.
* With unlimited computation, the best way to “regularize” a fixed-sized model is to average the predictions of all possible settings of the parameters, weighting each setting by its posterior probability given the training data.
* A problem even with the ensemble approximation is that it requires multiple models to be fit and stored, which can be a challenge if the models are large, requiring days or weeks to train and tune.

### Randomly Drop Nodes

* Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel.
* During training, some number of layer outputs are randomly ignored or “*dropped out*.” This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different “*view*” of the configured layer.
* By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections
* Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.
* This conceptualization suggests that perhaps dropout breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust.
* Dropout simulates a sparse activation from a given layer, which interestingly, in turn, encourages the network to actually learn a sparse representation as a side-effect. As such, it may be used as an alternative to activity regularization for encouraging sparse representations in autoencoder models.
* Because the outputs of a layer under dropout are randomly subsampled, it has the effect of reducing the capacity or thinning the network during training. As such, a wider network, e.g. more nodes, may be required when using dropout.

### How to Dropout

* Dropout is implemented per-layer in a neural network.
* It can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers such as the long short-term memory network layer.
* Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer. It is not used on the output layer.
* The term “dropout” refers to dropping out units (hidden and visible) in a neural network.
* A new hyperparameter is introduced that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained. The interpretation is an implementation detail that can differ from paper to code library.
* A common value is a probability of 0.5 for retaining the output of each node in a hidden layer and a value close to 1.0, such as 0.8, for retaining inputs from the visible layer.
* In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.
* Dropout is not used after training when making a prediction with the fit network.
* The weights of the network will be larger than normal because of dropout. Therefore, before finalizing the network, the weights are first scaled by the chosen dropout rate. The network can then be used as per normal to make predictions.
* If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time
* The rescaling of the weights can be performed at training time instead, after each weight update at the end of the mini-batch. This is sometimes called “*inverse dropout*” and does not require any modification of weights during training. Both the Keras and PyTorch deep learning libraries implement dropout in this way.

## Tips for Using Dropout Regularization

This section provides some tips for using dropout regularization with your neural network.

### Use With All Network Types

Dropout regularization is a generic approach.

It can be used with most, perhaps all, types of neural network models, not least the most common network types of Multilayer Perceptrons, Convolutional Neural Networks, and Long Short-Term Memory Recurrent Neural Networks.

In the case of LSTMs, it may be desirable to use different dropout rates for the input and recurrent connections.

### Dropout Rate

The default interpretation of the dropout hyperparameter is the probability of training a given node in a layer, where 1.0 means no dropout, and 0.0 means no outputs from the layer.

A good value for dropout in a hidden layer is between 0.5 and 0.8. Input layers use a larger dropout rate, such as of 0.8.

### Use a Larger Network

* It is common for larger networks (more layers or more nodes) to more easily overfit the training data.
* When using dropout regularization, it is possible to use larger networks with less risk of overfitting. In fact, a large network (more nodes per layer) may be required as dropout will probabilistically reduce the capacity of the network.
* A good rule of thumb is to divide the number of nodes in the layer before dropout by the proposed dropout rate and use that as the number of nodes in the new network that uses dropout. For example, a network with 100 nodes and a proposed dropout rate of 0.5 will require 200 nodes (100 / 0.5) when using dropout.

### Grid Search Parameters

Rather than guess at a suitable dropout rate for your network, test different rates systematically.

For example, test values between 1.0 and 0.1 in increments of 0.1.

This will both help you discover what works best for your specific model and dataset, as well as how sensitive the model is to the dropout rate. A more sensitive model may be unstable and could benefit from an increase in size.

### Use a Weight Constraint

Network weights will increase in size in response to the probabilistic removal of layer activations.

Large weight size can be a sign of an unstable network.

To counter this effect a weight constraint can be imposed to force the norm (magnitude) of all weights in a layer to be below a specified value. For example, the maximum norm constraint is recommended with a value between 3-4

### Use With Smaller Datasets

Like other regularization methods, dropout is more effective on those problems where there is a limited amount of training data and the model is likely to overfit the training data.

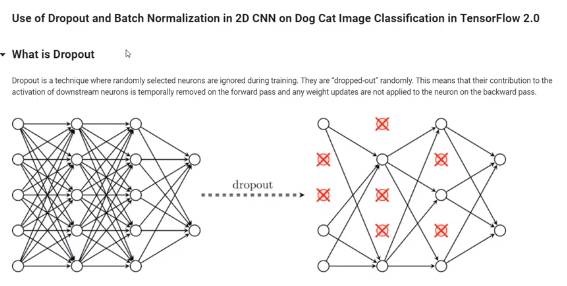
Problems where there is a large amount of training data may see less benefit from using dropout.

For very large datasets, regularization confers little reduction in generalization error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularization.

### Summary

* Large weights in a neural network are a sign of a more complex network that has overfit the training data.
* Probabilistically dropping out nodes in the network is a simple and effective regularization method.
* A large network with more training and the use of a weight constraint are suggested when using dropout.

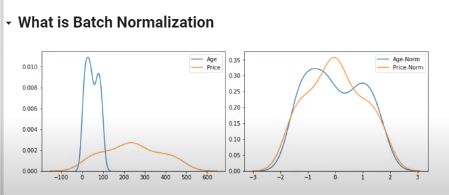
**Using Dropout and Batch Normalization in 2D CNN Dog Cat Image Classification**



* Dropout drops some neurons randomly, so that the model learned in a generalised way. As NN learns neurons were settled in to the context within the networks and weights of the networks are tuned for a specific feature to provide specialization.
* Eg: For a fully connected network weights of the neurons are so trained that there bias is low & variance is very high. For a unseen data they try to provide low accuracy and higher loss. Its like a regularization technique which prevents over fitting

**Batch Normalization**

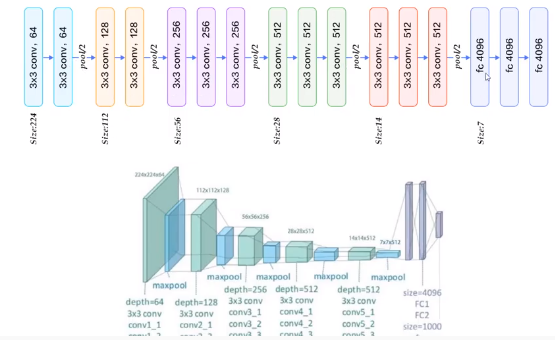
* Here we bring the data in the same shape.



* Due to the normalization layer between these fully connected layers the range of i/p distribution of each layer stays the same and no matter changes in the previous layer. It brings all the i/ps centred to 0

**VGG16 & 19**

**VGG16**

* ****

**VGG19**

