Deep Learning by Design

From Zero to Automatic Learning with Tensorflow 2.0

Andrew Ferlitsch, Google Cloud AI

Version January 2020

Topics

Preface

As a Googler, one of my duties is to educate software engineers on how to use machine learning. I already had experience creating online tutorials, meetups, conference presentations, training workshops, and coursework for private coding schools and university graduate studies, but I am always looking for new ways to effectively teach.

Welcome to my latest approach, the idiomatic programmer. My audience are software engineers, machine learning engineers and junior to mid-level data scientists. While for the later, one may assume that the initial chapters would seem redundant - but with my unique approach you will likely find additional insight as a welcomed refresher.

You should know at least the basics of Python. It's okay if you still struggle with what is a compression, what is a generator; you still have some confusion with the weird multi-dimensional array slicing, and this thing about which objects are mutable and non-mutable on the heap. For this tutorial it’s okay.

For those software engineers wanting to become a machine learning engineer -- What does that mean? A machine learning engineer (MLE) is an applied engineer. You don't need to know statistics (really you don't!), you don't need to know computational theory. If you fell asleep in your college calculus class on what a derivative is, that's okay, and if somebody asks you to do a matrix multiplication, feel free to ask, “why?”

Your job is to learn the knobs and levers of a framework, and apply your skills and experience to produce solutions for real world problems. That's what I am going to help you with and that’s what the composable design pattern using TF.Keras is about.

Tensorflow (TF) is a low-level graph based (symbolic programming) framework, while Keras was an abstraction on top of Tensorflow as a high-level abstraction (imperative programming), which is now fused into Tensorflow 2.0. Composable is an abstraction on top of an abstraction (TF.Keras).

Composable moves beyond building a model to building amalgamations of models that are entire applications for real world production.

The Machine Learning Steps

You've likely seen this before. A successful ML engineer will need to decompose a machine learning solution into the following steps:

1. Identify the Type of Model for the Problem
2. Design the Model
3. Prepare the Data for the Model
4. Train the Model
5. Deploy the Model

That is very 2017. It's now 2020 and a lot of things have progressed. There are now vasts numbers of model types, but with abstractions -- like composable -- we are seeing model types converging and likely by 2022 we will see just a handful of abstractions covering all types of production.

Outside of research, ML practitioners don't design models, they guide the design of the models. Data preparation is becoming more automated, some of which is moving into the models and other cases handled upstream by other models that have learned to prepare data.

We don't train one model anymore, we train a plurality of models, and for each model we train instances of the model in stages from warmup, pre-training and finally full-training.

Models are deployed in a wide variety of manners from the cloud, to mobile devices, to IoT (edge) devices. The deployment may modify the model (e.g., quantization, compression), perform continuous evaluation and be updated and versioned with continuous integration/continuous development (CI/CD).

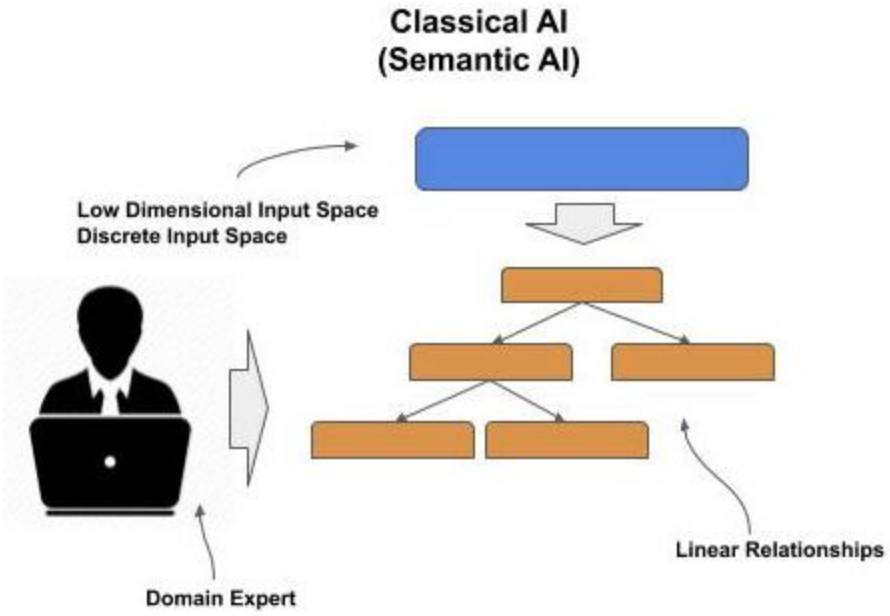
And the above list does not cover other new things we see in production, such as QA, A/B testing, auditing and bias adjustments.

Everyone is specializing, and likely you will too, pick a speciality that fits your skills and passion.

Welcome to AI in 2020.

Classical vs Narrow AI

Let's briefly cover the difference between classical AI (also known as semantic AI) and today's modern narrow AI (also known as statistical AI). In classical AI, models were designed as rule-based systems. These systems were used to solve problems that could not be solved by a mathematical equation. Instead the system is designed to mimic a subject matter expert (also known as domain expert).

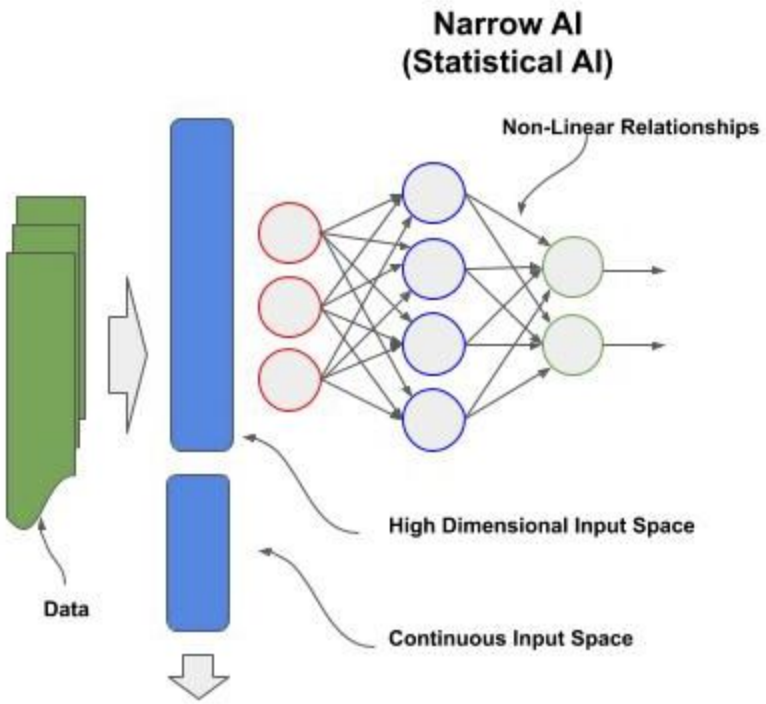


Classical AI works well in input spaces that are low-dimensionality (i.e., low number of distinct inputs), the input space can be broken into discrete segments (e.g., categorical, bins), and there is a strong linear relationship between the discrete space and the output. Systems like this were well suited for problems like predicting the quality of a wine. But they failed to scale to larger problems, where accuracy would drop dramatically and need for continuous refinement of the rules; as well as inconsistencies between domain experts in designing the rules.

In narrow AI, one trains a model on a large amount of data, alleviating the need for a domain expert. A model is constructed using principles of statistics for sampling distributions to learn patterns in the distribution that can then be applied with high accuracy to samples not seen by training (i.e.., population distribution). When trained with large amounts of data which would be representative (sampling distribution) of the population distribution, they can model problems that have substantially higher dimensionality in the input space (i.e., large number of distinct inputs), and inputs which can be a mix of discrete and continuous.

These models work well with input space that has a high level of non-linearity to the outputs (i.e., predictions), by learning the boundaries to segment up the input space, where within the segments there is a high level of linear relationship to the output. Finally, these types of models based on statistics and large amounts of data are referred to as narrow AI in that they are good at solving narrow problems, but is still challenging to generalize them to problems of a wide scope.

In addition to covering deep learning for narrow AI, in later chapters we will cover today's technique of generalizing models as we move into an era of pre-AGI (artificial general intelligence).


Narrow AI
(Statistical AI)
Non-Linear Relationships
High Dimensional Input Space
Data
Continuous Input Space

Chapter 1 - Deep Neural Networks

We will start with some basics.

The Input Layer

The input layer to a neural network takes numbers! All the input data is converted to numbers. Everything is a number. The text becomes numbers, speech becomes numbers, pictures become numbers, and things that are already numbers are just numbers.

Neural networks take numbers either as vectors, matrices or tensors. They are names for the number of dimensions in an array. A **vector** is a one dimensional array, like a list of numbers. A **matrix** is a two dimensional array, like the pixels in a black and white image, and a **tensor** is any array three or more dimensions. That's it.

Speaking of numbers, you might have heard terms like normalization or standardization. Hum, in standardization the numbers are converted to be centered around a mean of zero and one standard deviation on each side of the mean; and you say, 'I don't do statistics!' I know how you feel. Don't sweat. Packages like **scikit-learn** and **numpy** have library calls that do it for you, like its a button to push and it doesn't even need a lever (no parameters to set!).

Speaking of packages, you're going to be using a lot of **numpy**. What is this? Why is it so popular? In the interpretive nature of Python, the language poorly handles large arrays. Like really big, super big arrays of numbers - thousands, tens of thousands, millions of numbers. Think of Carl Sagan's infamous quote on the size of the Universe - billions and billions of stars. That's a tensor!

One day a C programmer got the idea to write in low level C a high performance implementation for handling super big arrays and then added an external Python wrapper. Numpy was born. Today **numpy** is a class with lots of useful methods and properties, like the property `shape` which tells you the shape (dimensions) of the array, or the `where()` method which allows you to do SQL like queries on your super big array.

All Python machine learning frameworks (TensorFlow, PyTorch, ...) will take as input on the input layer a **numpy** multidimensional array. And speaking of C, or Java, or C+, ..., the input layer in a neural network is just like the parameters passed to a function in a programming language. That's it.

Let's get started. I assume you have Python installed (version 3.X). Whether you directly installed it, or it got installed as part of a larger package, like Anaconda, you got with it a nifty command like tool called `pip`. This tool is used to install any Python package you will ever need again from a single command invocation. You go `pip install` and then the name of the package. It goes to the global repository PyPi of Python packages and downloads and installs the package for you. It's so easy.

We want to start off by downloading and installing the **Tensorflow** framework, and the **numpy** package. Guess what their names are in the registry, tensorflow and numpy - so obvious! Let's do it together. Go to the command line and issue the following:

```
cmd> pip install tensorflow
cmd> pip install numpy
```

With Tensorflow 2.0, Keras is builtin and the recommended model API, referred to now as **TF.Keras**.

TF.Keras is based on object oriented programming with a collection of classes and associated methods and properties. Let's start simple. Say we have a dataset of housing data. Each row has fourteen columns of data. One column has the sale price of a home. We are going to call that the *"label"*. The other thirteen columns have information about the house, like the sqft and property tax, etc. It's all numbers. We are going to call those the *"features"*. What we want to do is *"learn"* to predict (or estimate) the *"label"* from the *"features"*. Now before we had all this compute power and these awesome machine learning frameworks, people did this stuff by hand (we call them data analysts) or using formulas in an Excel spreadsheet with some amount of data and lots and lots of linear algebra.

We will start by first importing the **Keras** module from **TensorFlow**, and then instantiate an `Input` class object. For this class object, we define the shape (i.e., dimensions) of the input. In our example, the input is a one dimensional array (i.e., vector) of 13 elements, one for each feature.

```
from tensorflow.keras import Input

Input(shape=(13,))
```

When you run the above two lines in a notebook, you will see the output:

```
<tf.Tensor 'input_1:0' shape=(?, 13) dtype=float32>
```

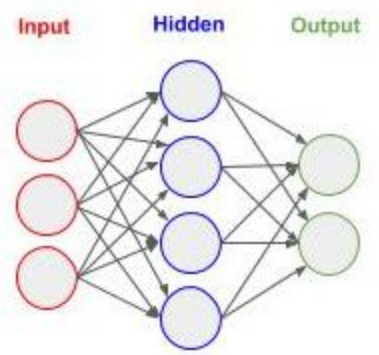
This is showing you what `Input(shape=(13,))` evaluates to. It produces a tensor object by the name 'input_1:0'. This name will be useful later in assisting you in debugging your models. The '?' in shape shows that the input object takes an unbounded number of entries (your examples or rows) of 13 elements each. That is, at run-time it will bind the number of one dimensional vectors of 13 elements to the actual number of examples (rows) you pass in, referred to as the (mini) batch size. The 'dtype' shows the default data type of the elements, which in this case is a 32-bit float (single precision).

Deep Neural Networks (DNN)

DeepMind, Deep Learning, Deep, Deep, Deep. Oh my, what's this? It just means that the neural network has one or more layers between the input layer and the output layer. Visualize a directed graph in layers of depth. The root nodes are the input layer and the terminal nodes are the output layer. The layers in between are known as the hidden (deep) layers. That's it. A four-layer DNN architecture would look like this:

input layer
hidden layer
hidden layer
output layer

For our purposes, we will start with every node in every layer, except the output layer, is the same type of node. And that every node on each layer is connected to every other node on the next layer. This is known as a fully connected neural network (FCNN). For example, if the input layer has three nodes and the next (hidden) layer has four nodes, then each node on the first layer is connected to all four nodes on the next layer for a total of 12 (3x4) connections.



Feed Forward

The DNN (and CNN) are known as feed forward neural networks. This means that data moves through the network sequentially in one direction (from input to output layer). That's like a function in procedural programming. The inputs are passed as parameters (i.e., input layer), the function performs a sequenced set of actions based on the inputs (i.e., hidden layers) and outputs a result (i.e., output layer).

There are two distinctive styles, which you will see in blogs and other tutorials, when coding a forward feed network in **TF.Keras**. I will briefly touch on both so when you see a code snippet in one style you can translate it to the other.

The Sequential API Method

The Sequential API method is easier to read and follow for beginners, but the trade off is its less flexible. Essentially, you create an empty forward feed neural network with the Sequential class object, and then "add" one layer at a time, until the output layer. In the examples below, the ellipses represent pseudo code.

```
from tensorflow.keras import Sequential

model = Sequential()
model.add( ...the first layer... )
model.add( ...the next layer... )
model.add( ...the output layer... )
```

Alternatively, the layers can be specified in sequential order as a list passed as a parameter when instantiating the Sequential class object.

```
model = Sequential([ ...the first layer...,
                     ...the next layer...,
                     ...the output layer...
                   ])
```

The Functional API Method

The Functional API method is more advanced, allowing you to construct models that are non-sequential in flow --such as branches, skip links, and multiple inputs and outputs. You build the layers separately and then "tie" them together. This latter step gives you the freedom to connect layers in creative ways. Essentially, for a forward feed neural network, you create the layers, bind them to another layer(s), and then pull all the layers together in a final instantiation of a Model class object.

```
input = layers.(...the first layer...)
hidden = layers.(...the next layer...)( ...the layer to bind to... )
output = layers.(...the output layer...)( /the layer to bind to... )
model = Model(input, output)
```

Input Shape vs Input Layer

The input shape and input layer can be confusing at first. They are not the same thing. More specifically, the number of nodes in the input layer does not need to match the shape of the input vector. That's because every element in the input vector will be passed to every node in the input layer. If our input layer is ten nodes, and we use our above example of a thirteen element input vector, we will have 130 connections (10 x 13) between the input vector and the input layer.

Each one of these connections between an element in the input vector and a node in the input layer will have a *weight* and the connection between the two has a *bias*. This is what the neural network will *"learn"* during training. These are also referred to as parameters. That is, these values stay with the model after it is trained. This operation will otherwise be invisible to you.

The Dense() Layer

In **TF.Keras**, layers in a fully connected neural network (FCNN) are called Dense layers, as depicted in the picture above. A Dense layer is defined as having an "n" number of nodes, and is fully connected to the previous layer. Let's continue and define in **TF.Keras** a three layer neural network, using the Sequential API method, for our example. Our input layer will be ten nodes, and take as input a thirteen element vector (i.e., the thirteen features), which will be connected to a second (hidden) layer of ten nodes, which will then be connected to a third (output) layer of one node. Our output layer only needs to be one node, since it will be outputting a single real value (e.g. - the predicted price of the house). This is an example where we are going to use a neural network as a *regressor*. That means, the neural network will output a single real number.

```
input layer  = 10 nodes
hidden layer = 10 nodes
output layer  = 1 node
```

For input and hidden layers, we can pick any number of nodes. The more nodes we have, the better the neural network can learn, but more nodes means more complexity and more time in training and predicting.

In the example below, we have three add() calls to the class object Dense(). The add() method "adds" the layers in the same sequential order we specified them in. The first (positional) parameter is the number of nodes, ten in the first and second layer and one in the third layer. Notice how in the first Dense() layer we added the (keyword) parameter input_shape. This is where we will define the input vector and connect it to the first (input) layer in a single instantiation of Dense().

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
# Add the first (input) layer (10 nodes) with input shape 13 element vector (1D).
model.add(Dense(10, input_shape=(13,)))
# Add the second (hidden) layer of 10 nodes.
model.add(Dense(10))
# Add the third (output) layer of 1 node.
model.add(Dense(1))
```

Alternatively, we can define the sequential sequence of the layers as a list parameter when instantiating the Sequential class object.

```
from keras import Sequential
from keras.layers import Dense

model = Sequential([
                    # Add the first (input) layer (10 nodes)
                    Dense(10, input_shape=(13,)),
                    # Add the second (hidden) layer of 10 nodes.
                    Dense(10),
                    # Add the third (output) layer of 1 node.
                    Dense(1)
                    ])
```

Let's now do the same but use the Functional API method. We start by creating an input vector by instantiating an Input class object. The (positional) parameter to the Input() object is the shape of the input, which can be a vector, matrix or tensor. In our example, we have a vector that is thirteen elements long. So our shape is (13,). I am sure you noticed the trailing comma! That's to overcome a quirk in Python. Without the comma, a (13) is evaluated as an expression. That is, the integer value 13 is surrounded by a parenthesis. Adding a comma will tell the interpreter this is a tuple (an ordered set of values).

Next, we create the input layer by instantiating a Dense class object. The positional parameter to the Dense() object is the number of nodes; which in our example is ten. Note the peculiar syntax that follows with a (inputs). The Dense() object is a callable. That is, the object returned by instantiating the Dense() object can be callable as a function. So we call it as a function, and in this case, the function takes as a (positional) parameter the input vector (or layer output) to connect it to; hence we pass it inputs so the input vector is bound to the ten node input layer.

Next, we create the hidden layer by instantiating another Dense() object with ten nodes, and using it as a callable, we (fully) connect it to the input layer.

Then we create the output layer by instantiating another Dense() object with one node, and using it as a callable, we (fully) connect it to the hidden layer.

Finally, we put it altogether by instantiating a Model class object, passing it the (positional) parameters for the input vector and output layer. Remember, all the other layers in-between we already connected so we don't need to specify them when instantiating the Model() object.

```
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import Dense

# Create the input vector (13 elements).
inputs = Input((13,))
# Create the first (input) layer (10 nodes) and connect it to the input vector.
input = Dense(10)(inputs)
# Create the next (hidden) layer (10 nodes) and connect it to the input layer.
hidden = Dense(10)(input)
# Create the output layer (1 node) and connect it to the previous (hidden) layer.
output = Dense(1)(hidden)
# Now let's create the neural network, specifying the input layer and output layer.
model = Model(inputs, output)
```

Activation Functions

When training or predicting (inference), each node in a layer will output a value to the nodes in the next layer. We don't always want to pass the value 'as-is', but instead sometimes we want to change the value by some manner. This process is called an activation function. Think of a function that returns some result, like return result. In the case of an activation function, instead of returning result, we would return the result of passing the result value to another (activation) function, like return A(result), where A() is the activation function. Conceptually, you can think of this as:

```
def layer(params):
    """ inside are the nodes """
    result = some_calculations
    return A(result)

def A(result):
    """ modifies the result """
    return some_modified_value_of_result
```

Activation functions assist neural networks in learning faster and better. By default, when no activation function is specified, the values from one layer are passed as-is (unchanged) to the next layer. The most basic activation function is a step function. If the value is greater than 0, then a 1 is outputted; otherwise a zero. It hasn't been used in a long, long time.

Let's pause for a moment and discuss what's the purpose of an activation function. You likely have heard the phrase *non-linearity.* What is this? To me, more importantly is what it is not.

In traditional statistics, we worked in low dimensional space where there was a strong linear correlation between the input space and output space, that could be computed as a polynomial transformation of the input that when transformed had a linear correlation to the output.

In deep learning, we work in high dimensional space where there is substantial non-linearity between the input space and output space. What is non-linearity? It means that an input is not (near) uniformly related to an output based on a polynomial transformation of the input. For example, let's say one's property tax is a fixed percentage rate (r) of the house value. In this case, the property tax can be represented by a function that multiplies the rate by the house value -- thus having a linear (i.e., straight line) relationship between value (input) and property tax (output).

$$tax = F(value) = r * value$$

Let's look at the logarithmic scale for measuring earthquakes, where an increase of one, means the power released is ten times greater. For example, an earthquake of 4 is 10 times stronger than a 3. By applying a logarithmic transform to the input power we have a linear relationship between power and scale.

$$scale = F(power) = log(power)$$

In a non-linear relationship, sequences within the input have different linear relationships to the output, and in deep learning we want to learn both the separation points as well as the linear functions for each input sequence. For example, consider age vs. income to demonstrate a non-linear relationship. In general, toddlers have no income, grade-school children have an allowance, early-teens earn an allowance + money for chores, later teens earn money from jobs, and then when they go to college their income drops to zero! After college, their income gradually increases until retirement, when it becomes fixed. We could model this non-linearity as sequences across age and learn a linear function for each sequence, such as depicted below.

income = F1(age) = 0	for age [0..5]
income = F2(age) = c1	for age[6..9]
income = F3(age) = c1 + (w1 * age)	for age[10..15]
income = F4(age) = (w2 * age)	for age[16..18]
income = F5(age) = 0	for age[19..22]
income = F6(age) = (w3 * age)	for age[23..64]
income = F7(age) = c2	for age [65+]

Activation functions assist in finding the non-linear separations and corresponding clustering of nodes within input sequences which then learn the (near) linear relationship to the output.

There are three activation functions you will use most of the time; they are the rectified linear unit (ReLU), sigmoid and softmax. The rectified linear unit passes values greater than zero as-is (unchanged); otherwise zero (no signal).

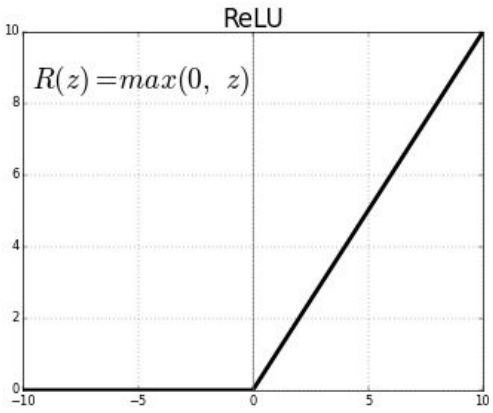


image source: https://towardsdatascience.com

The rectified linear unit is generally used between layers. In our example, we will add a rectified linear unit between each layer.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, ReLU

model = Sequential()
# Add the first (input) layer (10 nodes) with input shape 13 element vector (1D).
model.add(Dense(10, input_shape=(13,)))
# Pass the output from the input layer through a rectified linear unit activation
# function.
model.add(ReLU())
# Add the second (hidden) layer (10 nodes).
model.add(Dense(10))
# Pass the output from the input layer through a rectified linear unit activation
# function.
model.add(ReLU())
# Add the third (output) layer of 1 node.
model.add(Dense(1))
```

Let's take a look inside our model object and see if we constructed what we think we did. You can do this using the summary() method. It will show in sequential order a summary of each layer.

```
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_56 (Dense)             (None, 10)                140
_________________________________________________________________
re_lu_18 (ReLU)              (None, 10)                0
_________________________________________________________________
dense_57 (Dense)             (None, 10)                110
_________________________________________________________________
re_lu_19 (ReLU)              (None, 10)                0
_________________________________________________________________
dense_58 (Dense)             (None, 1)                 11
=================================================================
Total params: 261
Trainable params: 261
Non-trainable params: 0
_________________________________________________________________
```

For the above, you see the summary starts with a Dense layer of ten nodes (input layer), followed by a ReLU activation function, followed by a second Dense layer (hidden) of ten nodes, followed by a ReLU activation function, and finally followed by a Dense layer (output) of one node. Yup, we got what we expected.

Next, let's look at the parameter field in the summary. See how for the input layer it shows 140 parameters. You wonder how that's calculated? We have 13 inputs and 10 nodes, so 13 x 10 is 130. Where does 140 come from? You're close, each connection between the inputs and each node has a weight, which adds up to 130. But each node has an additional bias. That's ten nodes, so 130 + 10 = 140. It's the weights and biases the neural network will *"learn"* during training. A bias is a learned offset, conceptually equivalent to the y-intercept (b) in the slope of a line, which is where the line intercepts the y-axis.

$$y = b + mx$$

At the next (hidden) layer you see 110 params. That's ten outputs from the input layer connected to each of the ten nodes from the hidden layer (10x10) plus the ten biases for the nodes in the hidden layers, for a total of 110 parameters to *"learn"*.

Shorthand Syntax

TF.Keras provides a shorthand syntax when specifying layers. You don't actually need to separately specify activation functions between layers, as we did above. Instead, you can specify the activation function as a (keyword) parameter when instantiating a Dense() layer.

The code example below does exactly the same as the code above.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
# Add the first (input) layer (10 nodes) with input shape 13 element vector (1D).
model.add(Dense(10, input_shape=(13,), activation='relu'))
# Add the second (hidden) layer (10 nodes).
model.add(Dense(10, activation='relu'))
# Add the third (output) layer of 1 node.
model.add(Dense(1))
```

Let's call the summary() method on this model.

```
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_59 (Dense)             (None, 10)                140
_________________________________________________________________
dense_60 (Dense)             (None, 10)                110
_________________________________________________________________
dense_61 (Dense)             (None, 1)                 11
=================================================================
Total params: 261
Trainable params: 261
Non-trainable params: 0
_________________________________________________________________
```

Hum, you don't see the activations between the layers as you did in the earlier example. Why not? It's a quirk in how the summary() method displays output. They are still there.

Optimizer (Compile)

Once you've completed building the forward feed portion of your neural network, as we have for our simple example, we now need to add a few things for training the model. This is done with the compile() method. This step adds the *backward propagation* during training. That's a big phrase! Each time we send data (or a batch of data) forward through the neural network, the neural network calculates the errors in the predicted results (*loss*) from the actual values (*labels*) and uses that information to incrementally adjust the weights and biases of the nodes - what we are *"learning"*.

The calculation of the error is called a *loss*. It can be calculated in many different ways. Since we designed our neural network to be a *regresser* (output is a real value ~ house price), we want to use a loss function that is best suited for a *regresser*. Generally, for this type of neural network, the *Mean Square Error* method of calculating a loss is used.

The compile() method takes a (keyword) parameter loss where we can specify how we want to calculate it. We are going to pass it the value 'mse' for *Mean Square Error*.

The next step in the process is the optimizer that occurs during *backward propagation*. The optimizer is based on *gradient descent*; where different variations of the *gradient descent* algorithm can be selected. This term can be hard to understand at first. Essentially, each time we pass data through the neural network we use the calculated loss to decide how much to change the weights and biases in the layers by. The goal is to gradually get closer and closer to the correct values for the weights and biases to accurately predict (estimate) the *"label"* for each example. This process of progressively getting closer and closer is called *convergence*. As the *loss* gradually decreases we are converging and once the *loss* plateaus out, we have *convergence*, and the result is the accuracy of the neural network. Before using *gradient descent*, the methods used by early AI researchers could take years on a supercomputer to find *convergence* on a non-trivial problem. After the discovery of using the *gradient descent* algorithm, this time reduced to days, hours and even just minutes on ordinary compute power. Let's skip the math and just say that *gradient descent* is the data scientist's pixie dust that makes *convergence* possible.

For our *regresser* neural network we will use the rmsprop method (root mean square property).

```
model.compile(loss='mse', optimizer='rmsprop')
```

Now we have completed building your first 'trainable' neural network. Before we embark on preparing data and training the model, we will cover several more neural network designs first.

DNN Binary Classifier

Another form of a DNN, is a *binary classifier*, also known as a *logistic classifier*. In this case, we want the neural network to predict whether the input is or is not something. That is, the output can have two states (or classes): yes/no, true/false, 0/1, etc.

For example, let's say we have a dataset of credit card transactions and each transaction is labeled as whether it was fraudulent or not (i.e., the label - what we want to predict).

Overall, the design approach so far doesn't change, except the activation function of the 'single node' output layer and the loss/optimizer method.

Instead of using a linear activation function on the output node, we will use a sigmoid activation function. The sigmoid squashes all values to be between 0 and 1, and as values move away from the center they quickly move to the extremes of 0 and 1 (i.e., asymptotes).

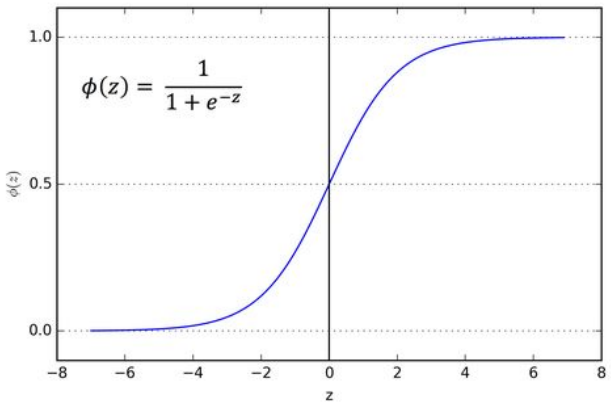


image source: https://towardsdatascience.com

We will now code this in several different styles. Let's start by taking our previous code example, where we specify the activation function as a (keyword) parameter. In this example, we add to the output Dense() layer the parameter activation='sigmoid' to pass the output result from the final node through a sigmoid function.

Next, we are going to change our loss parameter to 'binary_crossentropy'. This is the loss function that is generally used in a binary classifier (*logistic classifier*).

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
# Add the first (input) layer (10 nodes) with input shape 13 element vector (1D).
model.add(Dense(10, input_shape=(13,), activation='relu'))
# Add the second (hidden) layer (10 nodes).
model.add(Dense(10, activation='relu'))
# Add the third (output) layer of 1 node, and set the activation function to a
# Sigmoid.
model.add(Dense(1, activation='sigmoid'))

# Use the Binary Cross Entropy loss function for a Binary Classifier.
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

Not all the activation functions have their own class method, like the ReLU(). This is another quirk in the **TF.Keras** framework. Instead, there is a class called Activation() for creating any of the supported activations. The parameter is the predefined name of the activation function. In our example, 'relu' is for the rectified linear unit and 'sigmoid' for the sigmoid. The code below does the same as the code above.

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

model = Sequential()
# Add the first (input) layer (10 nodes) with input shape 13 element vector (1D).
model.add(Dense(10, input_shape=(13,)))
# Pass the output from the input layer through a rectified linear unit activation
# function.
model.add(Activation('relu'))
# Add the second (hidden) layer (10 nodes)
model.add(Dense(10))
# Pass the output from the hidden layer through a rectified linear unit activation
# function.
model.add(Activation('relu'))
# Add the third (output) layer of 1 node.
model.add(Dense(1))
# Pass the output from the output layer through a sigmoid activation function.
model.add(Activation('sigmoid'))

# Use the Binary Cross Entropy loss function for a Binary Classifier.
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

Now we will rewrite the same code using the Functional API approach. Notice how we repeatedly used the variable `x`. This is a common practice. We want to avoid creating lots of one-time use variables. Since we know in this type of neural network, the output of every layer is the input to the next layer (or activation), except for the input and output, we continuously use `x` as the connecting variable.

By now, you should start becoming familiar with the different styles and approaches. This will be helpful when reading blogs, online tutorials and stackoverflow questions which will aid in translating those snippets into the style/approach you choose.

```
from tensorflow.keras import Model, Input
from tensorflow.keras.layers import Dense, ReLU, Activation

# Create the input vector (13 elements)
inputs = Input((13,))
# Create the first (input) layer (10 nodes) and connect it to the input vector.
x = Dense(10)(inputs)
# Pass the output from the input layer through a rectified linear unit activation
# function.
x = Activation('relu')(x)
# Create the next (hidden) layer (10 nodes) and connect it to the input layer.
```

```
x = Dense(10)(x)
# Pass the output from the hidden layer through a rectified linear unit activation
# function.
x = Activation('relu')(x)
# Create the output layer (1 node) and connect it to the previous (hidden) layer.
x = Dense(1)(x)
# Pass the output from the output layer through a sigmoid activation function.
output = Activation('sigmoid')(x)
# Now let's create the neural network, specifying the input layer and output layer.
model = Model(inputs, output)

# Use the Binary Cross Entropy loss function for a Binary Classifier.
model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])
```

DNN Multi-Class Classifier

Another form of a DNN is a *multi-class classifier*, which means that we are going to classify (predict) from more than one class (label). For example, let's say from a set of body measurements (e.g., height and weight) and gender we want to predict if someone is a baby, toddler, preteen, teenager or adult, for a total of five classes.

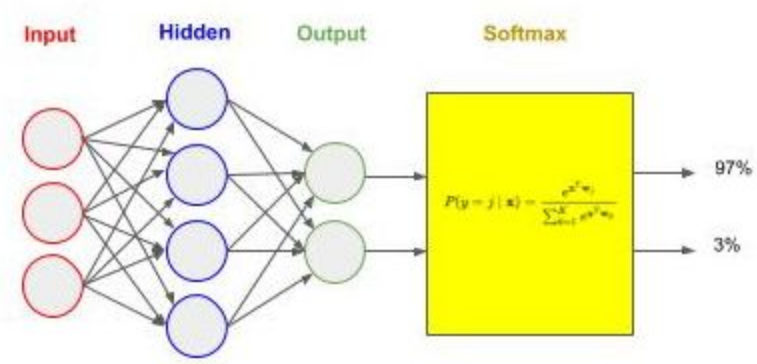
We can already see we will have some problems. For example, men on average as adults are taller than women. But during the preteen years, girls tend to be taller than boys. We know on average that men get heavier early in their adult years in comparison to their teenage years, but women on average are less likely. So we should anticipate lots of problems in predicting around the preteen years for girls, teenage years for boys, and adult years for women.

These are examples of non-linearity, where there is not a linear relationship between a feature and a prediction, but is instead broken into segments of disjoint linearity. This is the type of problem neural networks are good at.

Let's add a fourth measurement, the nose surface area. Studies have shown that for girls and boys, the surface area of the nose continues to grow between ages 6 and 18 and essentially stops at 18

So now we have four "*features"* and a *"label"* that consists of five classes. We will change our input vector in the next example to four, to match the number of features, and change our output layer to five nodes, to match the number of classes. In this case, each output node corresponds to one unique class (i.e., baby, toddler, etc). We want to train the neural network so each output node outputs a value between 0 and 1 as a prediction. For example, 0.75 would mean that the node is 75% confident that the prediction is the corresponding class (e.g., toddler).

Each output node will independently learn and predict its confidence on whether the input is the corresponding class. This leads to a problem in that because the values are independent, they won't add up to 1 (i.e, 100%). The function *softmax* is a mathematical function that will take a set of values (i.e., the outputs from the output layer) and squash them into a range between 0 and 1 and where all the values add up to 1. Perfect. This way, we can take the output node with the highest value and say both what is predicted and the confidence level. So if the highest value is 0.97, we can say we estimated the confidence at 97% in our prediction.



Next, we will change the activation function in our example to 'softmax'. Then we will set our loss function to 'categorical_crossentropy'. This is generally the most common used for multi-class classification. Finally, we will use a very popular and widely used variant of gradient descent called the *Adam Optimizer* ('adam'). *Adam* incorporates several aspects of other methods, such as rmsprop (root mean square) and adagrad (adaptive gradient), along with an adaptive learning rate. It's generally considered best-in-class for a wide variety of neural networks

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
# Add the first (input) layer (10 nodes) with input shape 4 element vector (1D).
model.add(Dense(10, input_shape=(4,), activation='relu'))
# Add the second (hidden) layer (10 nodes).
model.add(Dense(10, activation='relu'))
# Add the third (output) layer of 5 nodes, and set the activation function to a
# Softmax.
model.add(Dense(5, activation='softmax'))

# Use the Categorical Cross Entropy loss function for a Multi-Class Classifier.
model.compile(loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy'])
```

DNN Multi-Label Multi-Class Classifier

Another form of a DNN is a *multi-label multi-class classifier*, which means we will predict two or more classes (labels) per input. Let's use our previous example of predicting whether someone is a baby, toddler, preteen, teenager or adult. In this example, we will remove gender from one of the *"features"* and make it one of the *"labels"* to predict. That is, our input will be the height, weight and nose surface area, and our outputs will be two (multiple) classes (labels): age category (baby, toddler, etc) and gender (male or female). An example prediction might look like below.

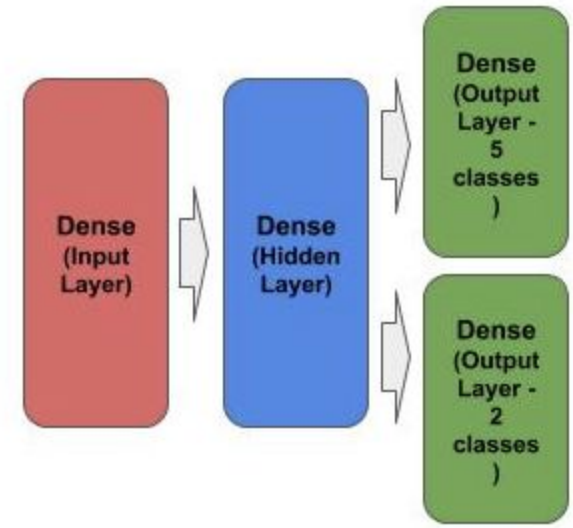
[height, weight, nose surface area] -> neural network -> [preteen, female]

For a *multi-label multi-class classifier,* we need to make a few changes from our previous *multi-class classifier.* On our output layer, our number of output classes is the sum for all the output categories. In this case, we previously had five and now we add two more for gender for a total of 7. We also want to treat each output class as a binary classifier (i.e., yes/no), so we change the activation function to a 'sigmoid'. For our compile statement, we set the loss function to 'binary_crossentropy', and the optimizer to 'rmsprop'.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
# Add the first (input) layer (10 nodes) with input shape 3 element vector (1D).
model.add(Dense(10, input_shape=(3,), activation='relu'))
# Add the second (hidden) layer (10 nodes).
model.add(Dense(10, activation='relu'))
# Add the third (output) layer of 7 nodes, and set the activation function to a
# Sigmoid.
model.add(Dense(7, activation='sigmoid'))

# Use the Binary Cross Entropy loss function for a Multi-Label Multi-Class
# Classifier.
model.compile(loss='binary_crossentropy',
        optimizer='rmsprop',
        metrics=['accuracy'])
```

Do you see a potential problem with this design? Let's assume we output the two classes (labels) with the highest values (between 0 and 1). What if on a prediction, the neural network predicts both preteen and teenager with high confidence and male/female with lower confidence? Well, we could fix that with some post-logic by selecting the highest confidence from the first five output classes and select the highest confidence from the last two classes (gender).

The Functional API gives us the ability to fix this as part of the neural network without adding any post-logic. In this case, we want to replace the output layer which combines the two sets of classes with two parallel output layers, one for the first set of classes (baby, toddler, etc) and one for the second set of classes (gender), which is depicted below:



This design can also be referred to as a neural network with multiple outputs. In the neural network example below, only the final output layer differs from our the previous one above. Instead of the output from the hidden layer going to a single output layer, it is passed in parallel to two output layers. One output layer will predict whether the input is a baby, toddler, etc and the other will predict the gender.

Then when we put it all together with the Model class, instead of passing in a single output layer, we pass in a list of output layers: [output1, output2]. Finally, since each of the output layers make independent predictions, we can return to treating them as a *multi-class classifier*; whereby, we return to using 'categorical_crossentropy' as the loss function and 'adam' as the optimizer.

Since we will be training the model to do multiple independent predictions, this is also known as a *multi-task* model.

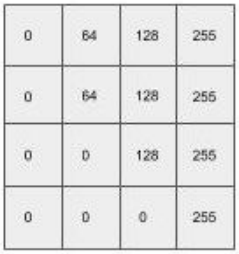
```
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import Dense

# Create the input vector (3 elements)
inputs = Input((3,))
# Create the first (input) layer (10 nodes) and connect it to the input vector.
x = Dense(10, activation='relu')(inputs)
# Create the next (hidden) layer (10 nodes) and connect it to the input layer.
x = Dense(10, activation='relu')(x)
# Create the two output layers and connect both to the previous (hidden) layer.
output1 = Dense(5, activation='softmax')(x)
output2 = Dense(2, activation='softmax')(x)
# Now let's create the neural network, specifying the input layer and the multiple
# output layers.
model = Model(inputs, [output1, output2])

# Use the Category Cross Entropy loss function for this Multi-Label Multi-Class
# Classifier.
model.compile(loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy'])
```

So which design is the correct (or better) for a *multi-label multi-class classifier*? It depends on the application. If all the classes (labels) are from a single category, then one would use the first pattern (single-task); otherwise, from different categories, one would use the second pattern (multi-task). The example we gave would use the multi-task pattern. For an example of the former, consider a neural network that classifies the scene background of an image, such as mountains, lake, ocean view, etc. What if one image had mountains and a lake? In this case, one would want to predict both classes mountains and lake.

Simple Image Classifier

Using neural networks for image classification is now used throughout computer vision. Let's start with the basics. For small size "gray scale" images, we can use a DNN similar to what we have already described. This type of DNN has been widely published in use of the MNIST dataset; which is a dataset for recognizing handwritten digits. The dataset consists of grayscale images of size 28 x 28 pixels. Each pixel is represented by an integer value between 0 and 255, which is a proportional scale on how white the pixel is (0 is black, 255 is white, and values between are shades of gray).

We will need to make one change though. A grayscale image is a matrix (2D array). Think of them as a grid, sized height x width, where the width are the columns and the height are the rows. A DNN though takes as input a vector (1D array). Yeaks!



Flattening

We are going to do classification by treating each pixel as a *"feature"*. Using the example of the MNIST dataset, the 28 x 28 images will have 784 pixels, and thus 784 *"features"*. We convert the matrix (2D) into a vector (1D) by flattening it. Flattening is the process where we place each row in sequential order into a vector. So the vector starts with the first row of pixels, followed by the second row of pixels, and continues by ending with the last row of pixels.

In our next example below, we add a layer at the beginning of our neural network to flatten the input, using the class Flatten. The remaining layers and activations are typical for the MNIST dataset. Note that the input shape to the Flatten() object is the 2D shape (28, 28). The output from this object will be a 1D shape of (784,).

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Flatten, ReLU, Activation

model = Sequential()
# Take input as a 28x28 matrix and flatten into a 784 vector.
```

```
model.add(Flatten(input_shape=(28,28)))
# Add the first (input) layer (512 nodes) with input shape 784 element vector (1D).
model.add(Dense(512, activation='relu'))
# Add the second (hidden) layer (512 nodes).
model.add(Dense(512, activation='relu'))
# Add the third (output) layer (10 nodes) with sigmoid activation function.
model.add(Dense(10, activation='softmax'))

# Use the Categorical Cross Entropy loss function for a Multi-Class Classifier.
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

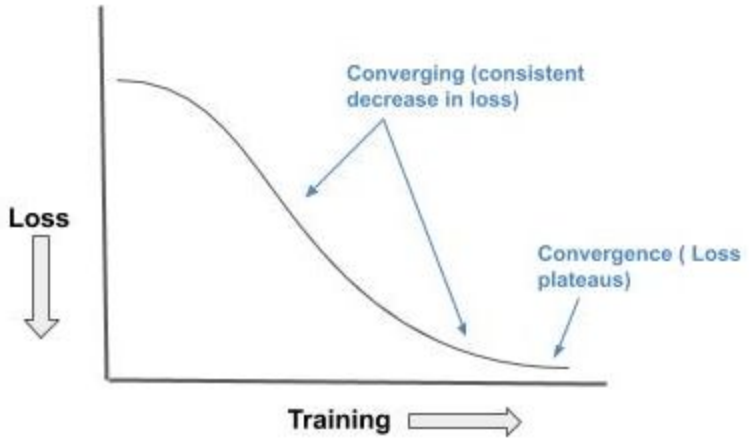
Let's now look at the layers using the summary() method. As you can see, the first layer in the summary is the flattened layer and shows that the output from the layer is 784 nodes. That's what we want. Also notice how many parameters the network will need to *"learn"* during training ~ nearly 700,000.

```
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_69 (Dense)             (None, 512)               401920
_________________________________________________________________
re_lu_20 (ReLU)              (None, 512)               0
_________________________________________________________________
dense_70 (Dense)             (None, 512)               262656
_________________________________________________________________
re_lu_21 (ReLU)              (None, 512)               0
_________________________________________________________________
dense_71 (Dense)             (None, 10)                5130
_________________________________________________________________
activation_10 (Activation)   (None, 10)                0
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
```

Overfitting and Dropout

During training (discussed later), a dataset is split into training data and test data (also known as holdout data). Only the training data is used during the training of the neural network. Once the neural network has reached *convergence*, training stops.



Afterwards, the training data is forward fed again without *backward propagation* enabled (i.e., no learning) to obtain an accuracy. This is also known as running the trained neural network in inference mode (prediction). In a train/test split (train/eval/test discussed later), the test data, which has been set aside and not used as part of training, is forward feed again without *backward propagation* enabled to obtain an accuracy.

Ideally, the accuracy on the training data and the test data will be nearly identical. In reality, the test data will always be a little less. There is a reason for this.

Once you reach *convergence*, continually passing the training data through the neural network will cause the neurons to more and more fit the data samples versus generalizing. This is known as overfitting. When the neural network is *overfitted* to the training data, you will get high training accuracy, but substantially lower accuracy on the test/evaluation data.

Even without training past the *convergence*, you will have some *overfitting*. The dataset/problem is likely to have non-linearity (hence why you're using a neural network). As such, the individual neurons will converge at a non-equal rate. When measuring *convergence*, you're looking at the overall system. Prior to that, some neurons have already converged and the continued training will cause them to overfit. Hence, why the test/evaluation accuracy will always be at least a bit less than the training.

Regularization is a method to address *overfitting* when training neural networks. The most basic type of regularization is called *dropout*. Dropout is like forgetting. When we teach young children we use root memorization, like the 12x12 times table (1 thru 12). We have them iterate, iterate, iterate, until they recite in any order the correct answer 100% of the time. But if we ask them 13 times 13, they would likely give you a blank look. At this point, the times table is overfitted in their memory. We then

switch to abstraction. During this second teaching phase, some neurons related to the root memorization will die (outside the scope of this article). The combination of the death of those neurons (forgetting) and abstraction allows the child's brain to generalize and now solve arbitrary multiplication problems, though at times they will make a mistake, even at times in the 12 x 12 times table, with some probabilistic distribution.

The *dropout* technique in neural networks mimics this process. Between any layer you can add a dropout layer where you specify a percentage (between 0 and 1) to forget. The nodes themselves won't be dropped, but instead a random selection on each forward feed during training will not pass a signal forward (forget). So for example, if you specify a dropout of 50% (0.5), on each forward feed of data a random selection of 1/2 of the nodes will not send a signal.

The advantage here is that we minimize the effect of localized *overfitting* while continuously training the neural network for overall *convergence*. A common practice for dropout is setting values between 20% and 50%.

In the example code below, we've added a 50% dropout to the input and hidden layer. Notice that we placed it before the activation (ReLU) function. Since dropout will cause the signal from the node, when dropped out, to be zero, it does not matter whether you add the Dropout layer before or after the activation function.

```
from keras import Sequential
from keras.layers import Dense, Flatten, ReLU, Activation, Dropout

model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(512))
# Add dropout of 50% at the input layer.
model.add(Dropout(0.5))
model.add(ReLU())

model.add(Dense(512))
# Add dropout of 50% at the hidden layer.
model.add(Dropout(0.5))
model.add(ReLU())

model.add(Dense(10))
model.add(Activation('softmax'))

# Use the Categorical Cross Entropy loss function for a Multi-Class Classifier.
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```