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).

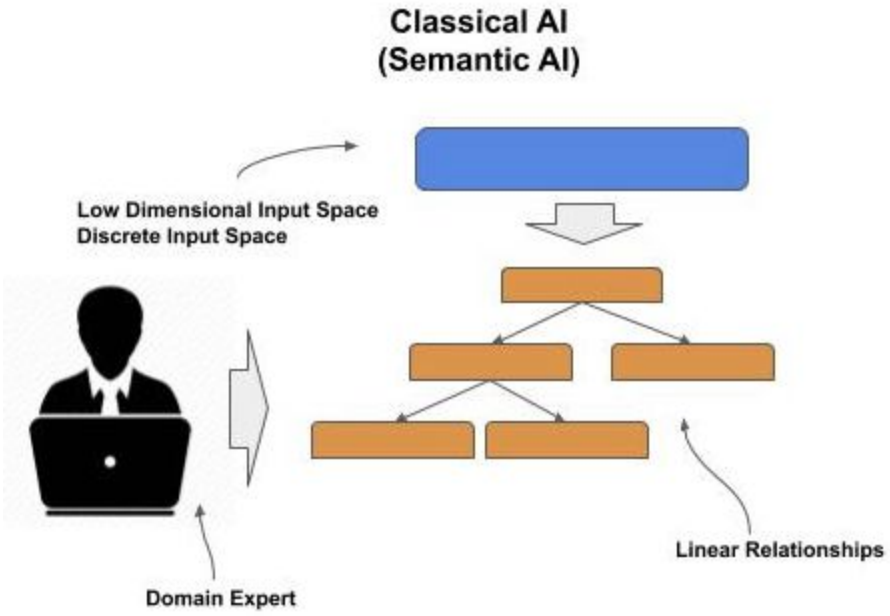
And the above list does not cover other new things we see in production, such as QA, A/B testing, auditing and bias adjustments.

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).

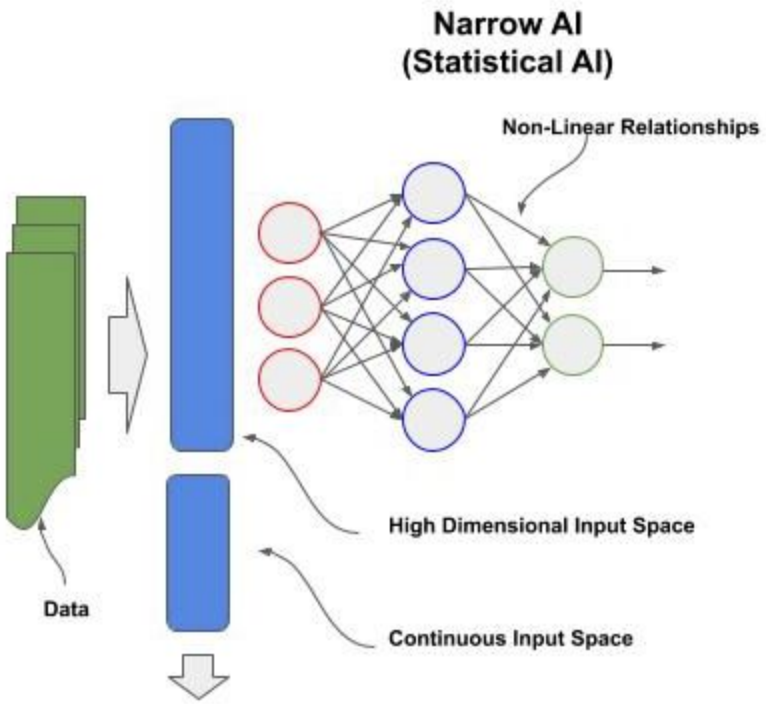


Classical AI works well in input spaces that are low-dimensionality (i.e., low number of distinct inputs), the input space can be broken into discrete segments (e.g., categorical, bins), and there is a strong linear relationship between the discrete space and the output. Systems like this were well suited 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>
```

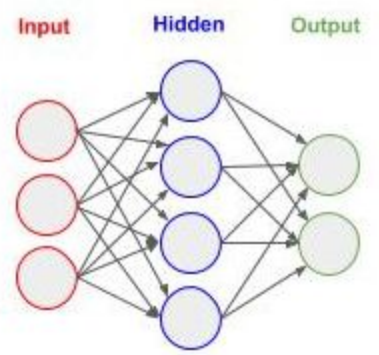
This is showing you what `Input(shape=(13,))` evaluates to. It produces a tensor object by the name 'input_1:0'. This name will be useful later in assisting you in debugging your models. The '?' in shape shows that the input object takes an unbounded number of entries (your examples or rows) of 13 elements 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).

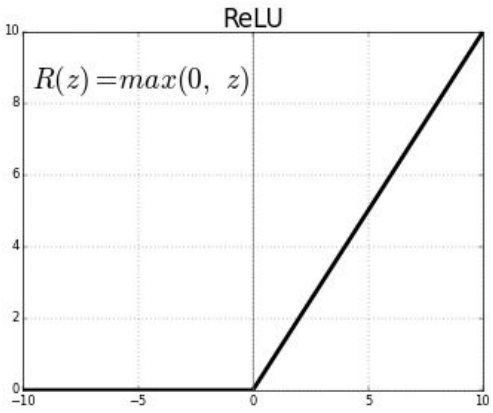


image source: https://towardsdatascience.com

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).

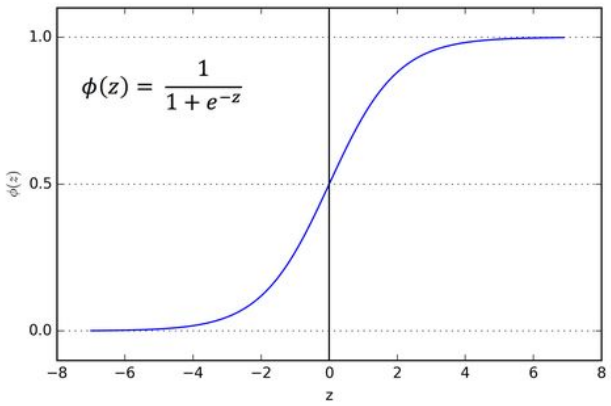


image source: https://towardsdatascience.com

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.

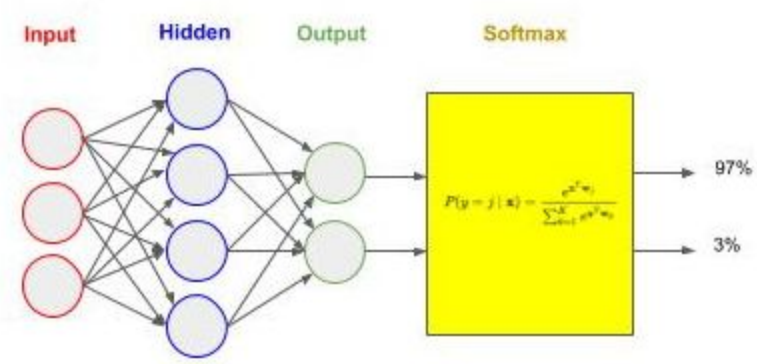
We can already see we will have some problems. For example, men on average as adults are taller than women. But during the preteen years, girls tend to be taller than boys. We know on average that men get heavier early in their adult years in comparison to their teenage years, but women on average are 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.

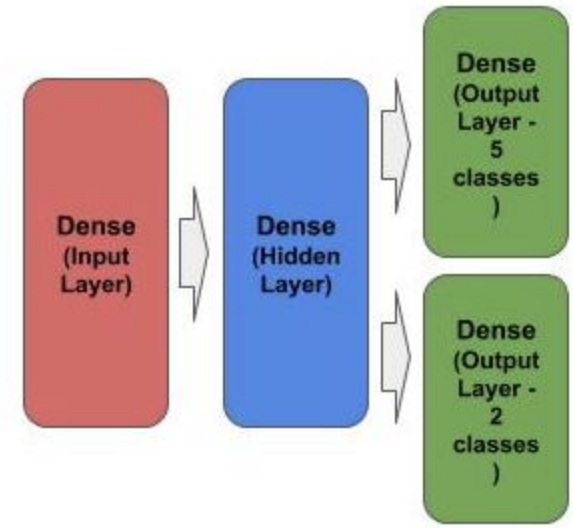
[height, weight, nose surface area] -> neural network -> [preteen, female]

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.

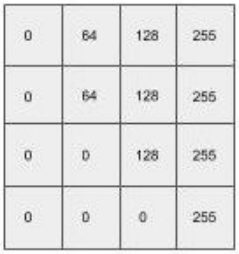
```
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import Dense

# 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'])
```

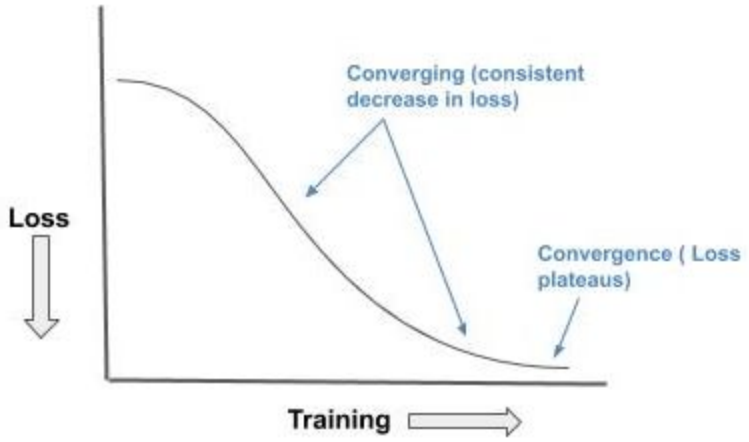
Let's now look at the layers using the summary() method. As you can see, the first layer in the summary is the flattened layer and shows that the output from the layer is 784 nodes. That's what we want. Also notice how many parameters the network will need to *"learn"* during training ~ nearly 700,000.

```
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%.

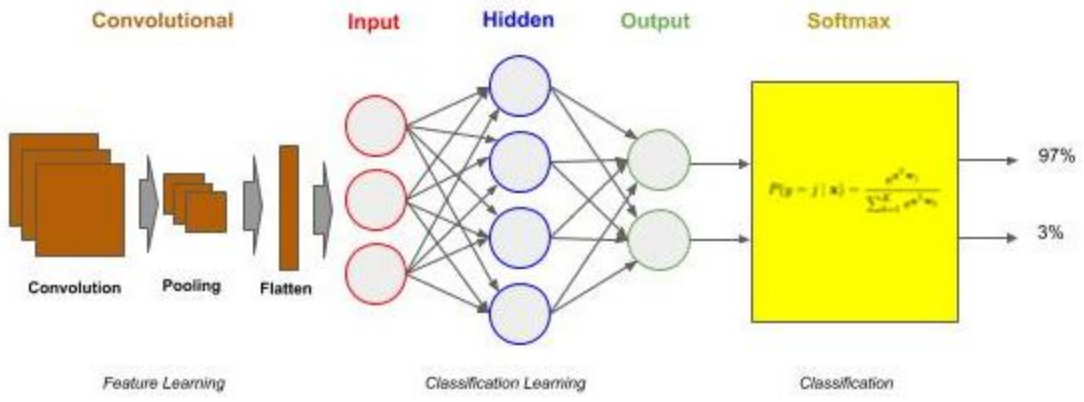
In the example code below, we've added a 50% dropout to the input and hidden layer. Notice that we placed it before the activation (ReLU) function. Since dropout will cause the signal from the node, when dropped out, to be zero, it does not matter whether you add the Dropout layer before or after the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

Chapter 2 - Convolutional and ResNet Neural Networks

Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) are a type of neural network that can be viewed as consisting of two parts, a frontend and a backend. The backend is a deep neural network (DNN), which we have already covered. The name convolutional neural network comes from the frontend, referred to as a convolutional layer(s). The frontend acts as a preprocessor. The DNN backend does the *"classification learning"*. The CNN frontend preprocesses the image data into a form which is computationally practical for the DNN to learn from. The CNN frontend does the *"feature learning"*.

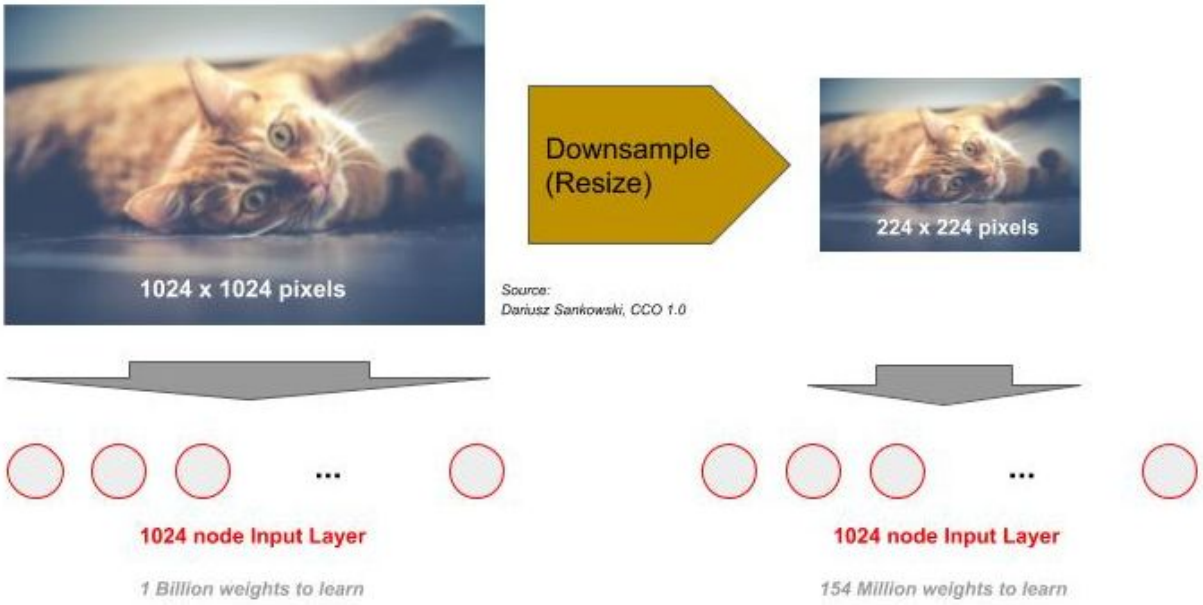


CNN Classifier

Once we get to larger image sizes, the number of pixels for a DNN becomes computationally too expensive to be feasible. Presume you have a 1MB image, where each pixel is represented by a single byte (0..255 value). At 1MB you have one million pixels. That would require an input vector of 1,000,000 elements. And let's assume that the input layer has 1024 nodes. The number of weights to *"update and learn"* would be over a billion (1 million x 1024) at just the input layer! Yeaks. Back to a supercomputer and a lifetime of computing power. Let's contrast this to our earlier MNIST example where we had 784 pixels times 512 nodes on our input layer. That's 400,000 weights to learn, which is considerably smaller than 1 billion. You can do the former on your laptop, but don't dare try the latter.

Downsampling (Resize)

To solve the problem of having too many parameters, one approach is to reduce the resolution of the image (downsampling). If we reduce the image resolution too far, at some point we may lose the ability to distinguish clearly what's in the image -- it becomes fuzzy and/or has artifacts. So, the first step is to reduce the resolution down to the level that we still have enough details. The common convention for everyday computer vision is around 224 x 224. We do this by resizing (discussed in a later tutorial). Even at this lower resolution and three channels for color images, and an input layer of 1024 nodes, we still have 154 million weights to *"update and learn"* (224 x 224 x 3 x 1024).



Pet Cat (Pixabay) - License

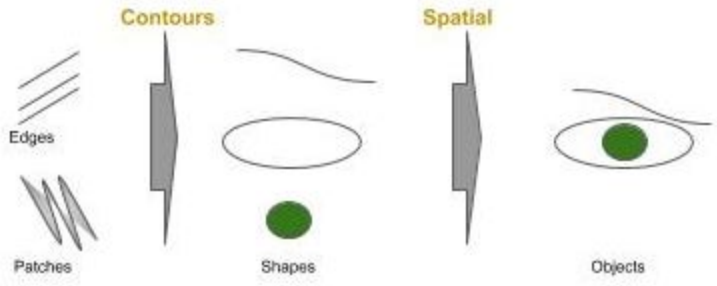
So training on real-world images was out of reach with neural networks until the introduction of using convolutional layers. To begin with, a convolutional layer is a frontend to a neural network, which transforms the images from a high dimensional pixel based image to a substantially lower dimensionality feature based image. The substantially lower dimensionality features can then be the input vector to a DNN. Thus, a convolutional frontend is a frontend between the image data and the DNN.

But let's say we have enough computational power to use just a DNN and learn 154 million weights at the input layer, as in our above example. Well, the pixels are very position dependent on the input layer. So we learn to recognize a "cat" on the left-side of the picture. But then we shift the cat to the middle of the picture. Now we have to learn to recognize a "cat" from a new set of pixel positions - Wah! Now move it to the right, add the cat lying down, jumping in the air, etc.

Learning to recognize an image from various perspectives, is referred to as translational invariance. For basic 2D renderings like digits and letters, this works (brute-force), but for everything else, it's not going to work.

Feature Detection

For these higher resolution and more complex images, we do recognition by detecting and classifying features instead of classifying pixel positions. Visualize an image, and ask yourself what makes you recognize what's there? Go beyond the high level of asking is that a person, a cat, a building, but ask why can you seperate in a picture a person standing in front of a building, or a person holding a cat. Your eyes are recognizing low-level features, such as edges, blurs, contrast, etc. These low-level features are built up into contours and then spatial relationships. Suddenly, the eye/brain has the ability to recognize nose, ears, eyes - that's a cat face, that's a human face.



A convolutional layer performs the task of feature detection within an image. Each convolution consists of a set of filters. These filters are NxM matrices of values that are used to detect the likely presence (detection) of a feature. Think of them as little windows. They are slid across the image, and at each location a comparison is made between the filter and the pixel values at that location. That comparison is done with a matrix dot product, but we will skip the statistics here. What's important, is the result of this operation will generate a value that indicates how strongly the feature was detected at that location in the image. For example, a value of 4 would indicate a stronger presence of the feature than the value of 1.

Prior to neural networks, imaging scientists hand designed these filters. Today, the filters along with the weights in the neural network are *"learned"*. In a convolutional layer, one specifies the size of the filter and the number of filters. Typical filter sizes are 3x3 and 5x5, with 3x3 the most common. The number of filters varies more, but they are typically multiples of 16, such as 16, 32 or 64 are the most common in shallow convolutional neural networks, and 256, 512 and 1024 in deep convolutional neural networks. Additionally, one specifies a stride. The stride is the rate that the filter is slid across the image. For example, if the stride is one, the filter advances one pixel at a time, thus the filter would partially overlap with the previous step in a 3x3 filter (and consequently so would a stride of 2). In a stride of 3, there would be no overlap. Most common practice is to use strides of 1 and 2. Each filter that is *"learned"* produces a feature map, which is a mapping (where) on how strongly the feature is detected in the image.

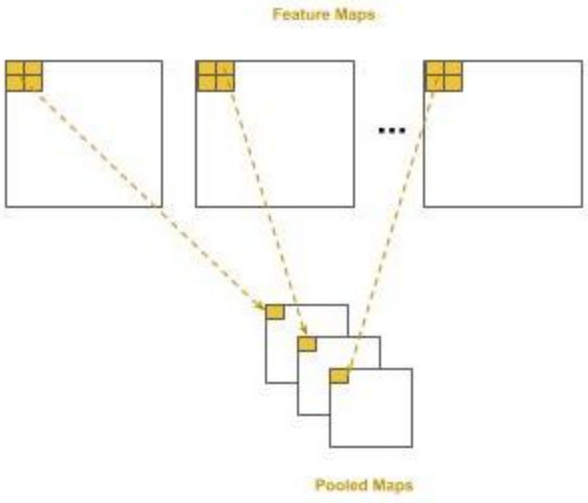
When there are multiple convolutional layers, the common practice is to keep the same or increase the number of filters on deeper layers, and to use stride of 1 on the first layer and 2 on deeper layers. The increase in filters provides the means to go from coarse detection of features to more detailed detection within coarse features, while the increase in stride offsets the increase in size of retained data, also referred to as feature pooling, which is also referred to as feature map downsampling. In convolutional neural networks, there are two types of downsampling, pooling (discussed next) and feature pooling (discussed later). In the former, a fixed algorithm is used to downsample the size of the image data. In feature pooling, the best downsampling algorithm for the specific dataset is *"learned"*.

More Filters => More Data
Bigger Strides => Less Data

Pooling

Even though each feature map generated typically is equal or less in size of the image, because we generate multiple feature maps (e.g., 16), the total data size has gone up. Yeaks! The next step is to reduce the total amount of data, while retaining the features detected and corresponding spatial relationship between the detected features.

This step is referred to as pooling. Pooling is the same as downsampling (or sub-sampling); whereby the feature maps are resized to a smaller dimension using either max (downsampling) or mean pixel average (sub-sampling) within the feature map. In pooling, we set the size of the area to pool as a NxM matrix as well as a stride. The common practice is a 2x2 pool size with a stride of 2. This will result in a 75% reduction in pixel data, while still preserving enough resolution that the detected features are not lost through pooling.

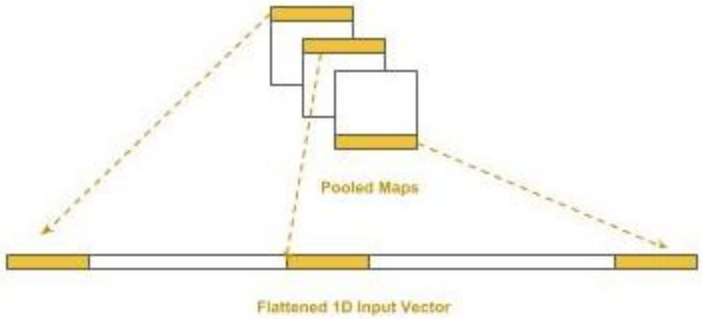


Another way to look at pooling is in the context of information gain. By reducing unwanted or less informative pixels (e.g., background) we are reducing entropy and making the remaining pixels more informative.

Flattening

Recall that deep neural networks take vectors as input, that's one dimensional arrays of numbers. In the case of the pooled maps, we have a list (plurality) of 2D matrices, so we need to transform these into a single 1D vector which then becomes the input vector to the DNN. This process is called flattening; that is, we flatten the list of 2D matrices into a single 1D vector. It's pretty straight forward. We start with the first row of the first pooled map as the beginning of the 1D vector. We then take the 2nd row and append it to the end, and then the 3rd row, and so forth. We then proceed to the second pooled map and do the same process, continuously appending each row, until we've completed the last pooled map. As long as we follow the same sequencing through pooled maps, the spatial relationship between detected features will be maintained across images for training and inference (prediction).

For example, if we have 16 pooled maps of size 20x20 and three channels per pooled map (e.g., RGB channels in color image), our 1D vector size will be 16 x 20 x 20 x 3 = 19,200 elements.



Basic CNN

Let's get started now with **TF.Keras**. Let's assume a hypothetical situation, but resembles the real world today. Your company's application supports human interfaces and currently can be accessed through voice activation. You've been tasked with developing a proof of concept to demonstrate expanding the human interface for Section 503 compliance accessibility to include a sign language interface.

What you should not do is assume to train the model using arbitrary labeled sign language images and image augmentation. The data, its preparation and the design of the model must match the actual "in the wild" deployment. Otherwise, beyond disappointing accuracy, the model might learn noise exposing it to false positives of unexpected consequences, and being vulnerable to hacking. We will discuss this in more detail in later parts.

For our proof of concept, we are only going to show recognizing hand signs for the letters of the english alphabet (A .. Z). Additionally, we assume that the individual will be signing directly in front of the camera from a dead-on perspective. Things we don't want to learn as an example, is the ethnicity of the hand signer. So for this, and other reasons, color is not important. To make our model not learn color ("the noise") we will train it in grayscale mode. That is, we will design the model to learn and predict (inference) in grayscale. What we do want to learn are contours of the hand.

The code sample below is written in the Sequential API method and in long form, where activation functions are specified using the corresponding method (vs. specifying them as a parameter when adding the corresponding layer).

We will design the model in two parts, the convolutional frontend and the DNN backend. We start by adding a convolutional layer of 16 filters as the first layer using the Conv2D class object. Recall that the number of filters equals the number of feature maps that will be generated, in this case 16. The size of each filter will be a 3x3, which is specified by the parameter kernel_size and a stride of 2 by the parameter strides. Note that for strides a tuple of (2, 2) is specified instead of a single value 2. The first digit is the horizontal stride (across) and the second digit is the vertical stride (down). It's a common convention for stride that the horizontal and vertical are the same; therefore one commonly says a "stride of 2" instead of "a 2x2 stride".

You may ask about what is with the 2D part in the name Conv2D. The 2D means that input to the convolutional layer will be a matrix (2-dimensional array). For the purpose of this chapter, we will stick with 2D convolutionals, which are the common practice for computer vision.

Let's calculate what the output size will be from this layer. As you recall, at stride of one, each output feature map will be the same size of the image. With 16 filters, that would be 16X the input. But since we used stride of two (feature pooling), each feature map will be reduced by 75%, so the total output size will be 4X the input.

The output from the convolution layer is then passed through a rectified linear unit activation function, which is then passed to the max pooling layer, using the MaxPool2D class object. The size of the pooling region will be 2x2, specified by the parameter pool_size, with a stride of 2 by the parameter strides. The pooling layer will reduce the feature maps by 75% into pooled feature maps.

Let's calculate the output size after the pooling layer. We know that the size coming in is 4X the input. With an additional 75% reduction, the output size is the same as the input. So what have we gained here? First, we have trained a set filters to learn a first set of coarse features (information gain), eliminated non-essential pixel information (reduce entropy), and learned the best method to downsample the feature maps. Hum, seems we gained a lot.

The pooled feature maps are then flattened, using the Flatten class object, into a 1D vector for input into the DNN. We will glance over the parameter padding. It is sufficient for our purposes to say that in almost all cases, you will use the value 'same'; it's just that the default is 'valid' and therefore you need to explicitly add it.

Finally, we pick an input size for our images. We like to reduce the size to as small as possible without losing detection of the features which are needed for recognizing the contours of the hand. In this case, we choose 128 x 128. The Conv2D class has a quirk in that it always requires specifying the number of channels, instead of defaulting to one for grayscale; thus we specified it as (128, 128, 1) instead of (128, 128).

```
# Keras's Neural Network components
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, ReLU, Activation
# Kera's Convolutional Neural Network components
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten

model = Sequential()
# Create a convolutional layer with 16 3x3 filters and stride of two as the input
# layer.
model.add(Conv2D(16, kernel_size=(3, 3), strides=(2, 2), padding="same",
                 input_shape=(128,128,1)))
# Pass the output (feature maps) from the input layer (convolution) through a
# rectified linear unit activation function.
model.add(ReLU())
# Add a pooling layer to max pool (downsample) the feature maps into smaller pooled
# feature maps.
```

```
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# Add a flattening layer to flatten the pooled feature maps to a 1D input vector
# for the DNN classifier
model.add(Flatten())

# Add the input layer for the DNN, which is connected to the flattening layer of
# the convolutional frontend.
model.add(Dense(512))
model.add(ReLU())
# Add the output layer for classifying the 26 hand signed letters
model.add(Dense(26))
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'])
```

Let's look at the details of the layers in our model using the summary() method.

```
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 64, 64, 16)        160
_________________________________________________________________
re_lu_1 (ReLU)               (None, 64, 64, 16)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 32, 16)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 16384)             0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               8389120
_________________________________________________________________
re_lu_2 (ReLU)               (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 26)                13338
_________________________________________________________________
activation_1 (Activation)    (None, 26)                0
=================================================================
Total params: 8,402,618
Trainable params: 8,402,618
Non-trainable params: 0
```

Here's how to read the Output Shape column. For the Conv2D input layer, the output shape shows (None, 64, 64, 16). The first value in the tuple is the number of examples (i.e., batch size) that will be passed through on a single forward feed. Since this is determined at training time, it is set to None to indicate it will be bound when the model is being fed data. The last number is the number of filters, which we set to 16. The two numbers in the middle 64, 64 are the output size of the feature maps, in this case 64 x 64 pixels each (for a total of 16). The output size is determined by the filter size (3 x 3), the stride (2 x 2) and the padding (same). The combination that we specified will result in the height and width being halved, for a total reduction of 75% in size.

For the MaxPooling2D layer, the output size of the pooled feature maps will be 32 x 32. By specifying a pooling region of 2 x 2 and stride of 2, the height and width of the pooled feature maps will be halved, for a total reduction of 75% in size.

The flattened output from the pooled feature maps is a 1D vector of size 16,384, calculated as 16 x (32 x 32). Let's see if this adds up to what we calculated earlier that the output size of the feature maps should be the same as the input size. Our input is 128 x 128, which is 16,384 which matches the output size from the Flatten layer.

Each element (pixel) in the flattened pooled feature maps is then inputted to each node in the input layer of the DNN, which has 512 nodes. The number of connections between the flattened layer and the input layer is therefore 16,384 x 512 = ~8.4 million. That's the number of weights to *"learn"* at that layer and where most of the computation will (overwhelmingly) occur.

Let's now show the same code example in a variation of the Sequential method style where the activation methods are specified using the parameter activation in each instantiation of a layer (e.g., Conv2D(), Dense()).

```
# Keras's Neural Network components
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Kera's Convolutional Neural Network components
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten

model = Sequential()

# Create a convolutional layer with 16 3x3 filters and stride of two as the input
# layer.
model.add(Conv2D(16, kernel_size=(3, 3), strides=(2, 2), padding="same",
                 activation='relu', input_shape=(128,128, 1)))
# Add a pooling layer to max pool (downsample) the feature maps into smaller pooled
# feature maps.
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
# Add a flattening layer to flatten the pooled feature maps to a 1D input vector
```

```
# for the DNN.
model.add(Flatten())

# Create the input layer for the DNN, which is connected to the flattening layer of
# the convolutional front-end.
model.add(Dense(512, activation='relu'))
model.add(Dense(26, 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 show the same code example in a third way using the Functional API method. In this approach we separately define each layer, starting with the input vector and proceed to the output layer. At each layer we use polymorphism to invoke the instantiated class (layer) object as a callable and pass in the object of the previous layer to connect it to.

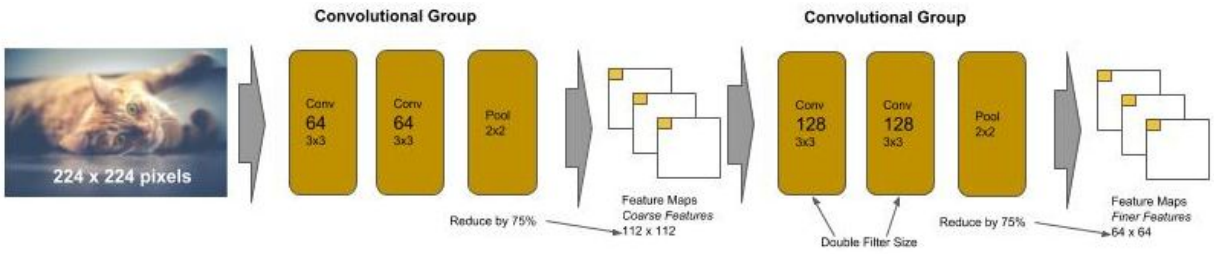
For example, for the first Dense layer, when invoked as a callable, we pass as the parameter the layer object for the `Flatten` layer. As a callable, this will cause the `Flatten` layer and the first Dense layer to be fully connected (i.e., each node in the Flatten layer will be connected to every node in the Dense layer).

```
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten

# Create the input vector (128 x 128).
inputs  = Input(shape=(128, 128, 1))
layer   = Conv2D(16, kernel_size=(3, 3), strides=(2, 2), padding="same",
                 activation='relu')(inputs)
layer   = MaxPooling2D(pool_size=(2, 2), strides=(2, 2))(layer)
layer   = Flatten()(layer)
layer   = Dense(512, activation='relu')(layer)
outputs = Dense(26, activation='softmax')(layer)

# Now let's create the neural network, specifying the input layer and output layer.
model = Model(inputs, outputs)
```

VGG Networks

The **VGG** type of CNN was designed by the *Visual Geometry Group* at *Oxford*. It was designed to compete in the international *ImageNet* competition for image recognition for 1000 classes of images. The **VGGNet** in the 2014 contest took first place on image location task and second place on the image classification task.

It is designed using a handful of principles that are easy to learn. The convolutional frontend consists of a sequence of pairs (and later triples) of convolutions of the same size, followed by a max pooling. The max pooling layer downsamples the generated feature maps by 75% and the next pair (or triple) of convolutional layers then doubles the number of learned filters. The principle behind the convolution design was that the early layers learn coarse features and subsequent layers, by increasing the filters, learn finer and finer features, and the max pooling is used between the layers to minimize growth in size (and subsequently parameters to learn) of the feature maps. Finally, the DNN backend consists of two identical sized dense hidden layers of 4096 nodes each, and a final dense output layer of 1000 nodes for classification.



The best known versions are the VGG16 and VGG19. The VGG16 and VGG19 that were used in the competition, along with their trained weights from the competition were made publicly available. They have been frequently used in transfer learning, where others have kept the convolutional frontend, and corresponding weights, and attached a new DNN backend and retrained for new classes of images.

So, we will go ahead and code a VGG16 in two coding styles. The first in a sequential flow, and the second procedurally using “reuse” functions for duplicating the common blocks of layers, and parameters for their specific settings. We will also change specifying kernel_size and pool_size as keyword parameters and instead specify them as positional parameters.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

model = Sequential()

# First convolutional block
```

```
model.add(Conv2D(64, (3, 3), strides=(1, 1), padding="same",
          activation="relu", input_shape=(224, 224, 3)))
model.add(Conv2D(64, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(MaxPooling2D((2, 2), strides=(2, 2))) # reduce feature maps by 75%

# Second convolutional block - double the number of filters
model.add(Conv2D(128, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(Conv2D(128, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(MaxPooling2D((2, 2), strides=(2, 2))) # reduce feature maps by 75%

# Third convolutional block - double the number of filters
model.add(Conv2D(256, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(Conv2D(256, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(Conv2D(256, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(MaxPooling2D((2, 2), strides=(2, 2))) # reduce feature maps by 75%

# Fourth convolutional block - double the number of filters
model.add(Conv2D(512, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(Conv2D(512, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(Conv2D(512, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(MaxPooling2D((2, 2), strides=(2, 2))) # reduce feature maps by 75%

# Fifth (Final) convolutional block
model.add(Conv2D(512, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(Conv2D(512, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(Conv2D(512, (3, 3), strides=(1, 1), padding="same", activation="relu"))
model.add(MaxPooling2D((2, 2), strides=(2, 2))) # reduce feature maps by 75%

# DNN Backend
model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dense(4096, activation='relu'))

# Output layer for classification (1000 classes)
model.add(Dense(1000, activation='softmax'))

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

You just coded a VGG16 - nice. Let's now code the same using a procedural "reuse" style. In this example we created a procedure (function) conv_block() which builds the convolutional blocks, and takes as parameters the number of layers in the block (2 or 3), and number of filters (64, 128, 256 or 512). Note that we kept the first convolutional layer outside of the conv_block.

The first layer needs the input_shape parameter. We could have coded this as a flag to conv_block, but since it would only occur one time, then it's not reuse. So we inline it instead.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

def conv_block(n_layers, n_filters):
    """
        n_layers : number of convolutional layers
        n_filters: number of filters
    """
    for n in range(n_layers):
        model.add(Conv2D(n_filters, (3, 3), strides=(1, 1), padding="same",
                         activation="relu"))
    model.add(MaxPooling2D(2, strides=2))

# Convolutional Frontend
model = Sequential()
model.add(Conv2D(64, (3, 3), strides=(1, 1), padding="same", activation="relu",
                 input_shape=(224, 224, 3)))
conv_block(1, 64)
conv_block(2, 128)
conv_block(3, 256)
conv_block(3, 512)
conv_block(3, 512)

# DNN Backend
model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dense(4096, activation='relu'))

# Output layer for classification (1000 classes)
model.add(Dense(1000, activation='softmax'))

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

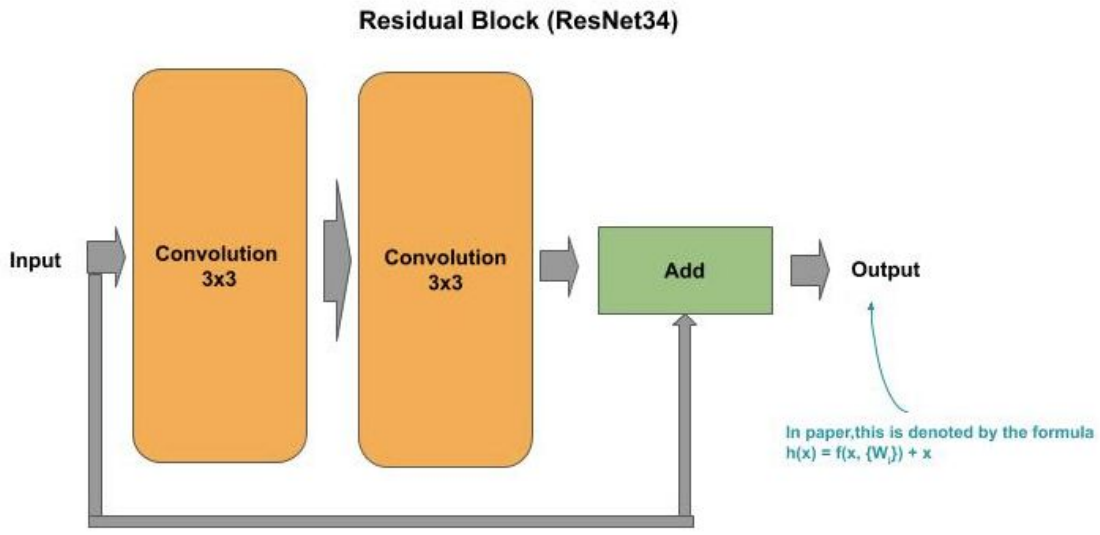
Try running model.summary() on both examples and you will see that the output is identical.

ResNet Networks

The **ResNet** type of CNN was designed by Microsoft Research. It was designed to compete in the international *ImageNet* competition. The **ResNet** in the 2015 contest took first place in all categories for *ImageNet* and *COCO* competition.

ResNet, and other architectures within this class, use different layer to layer connection patterns. The pattern we've discussed so far (ConvNet and VGG) use the fully connected layer to layer pattern.

ResNet 34 introduced a new block layer and layer connection pattern, residual blocks and identity connection, respectively. The residual block in ResNet 34 consists of blocks of two identical convolutional layers without a pooling layer. Each block has an identity connection which creates a parallel path between the input of the residual block and its output. Like VGG, each successive block doubles the number of filters. Pooling is done at the end of the sequence of blocks.



One of the problems with neural networks is that as we add deeper layers (under the presumption of increasing accuracy) their performance can degrade. That is, it can get worse not better. There are several reasons for this. As we go deeper, we are adding more parameters (weights). The more parameters, the more places that each input in the training data will fit to the excess parameters. That is, instead of generalizing the neural network will simply learn each training example (rote memorization). The other issue is covariate shift, where the distribution of the weights will widen (spread further apart) as we go deeper, resulting in making it more difficult for the neural network to converge. In the former case, we will see a degradation in performance on the test (holdout) data and the later on the training data, as well as vanishing or exploding gradient.

Residual blocks allow neural networks to be built with deeper layers without a degradation in performance on the test data. A ResNet block could be viewed as a VGG block with the addition of the identity link. While the VGG-style of the block performs feature detection, the identity link retains the input for the next subsequent block; whereby the input to the next block consists of both the previous features detection and input.
By retaining information from the past (previous input), this block design allows neural networks to go deeper than the VGG counterpart, with increase in accuracy. Mathematically, we could represent the VGG and ResNet as below. For both cases, we want to learn a formula for h(x) which is the distribution (e.g., labels) of the test data. For VGG, we are learning a function f(x, {W}), where {W} are the weights. For ResNet, we modify the equation by adding the term "+ x", which is the identity.

VGG: h(x) = f(x, {W})
ResNet: h(x) = f(x, {W}) + x

Below is a code snippet showing how a residual block can be coded in **TF.Keras** using the Sequential API method approach. The variable x represents the output of a layer, which is the input to the next layer. At the beginning of the block, we retain a copy of the previous block/layer output as the variable shortcut. We then pass the previous block/layer output (x) through two convolutional layers, each time taking the output from the previous layer as input into the next layer. Finally, the last output from the block (retained in the variable x) is added (matrix addition) with the original value of x (shortcut). This is the identity link; which is commonly referred to as a shortcut.

```
shortcut = x
x = layers.Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding='same')(x)
x = layers.ReLU()(x)
x = layers.Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding='same')(x)
x = layers.ReLU()(x)
x = layers.add([shortcut, x])
```

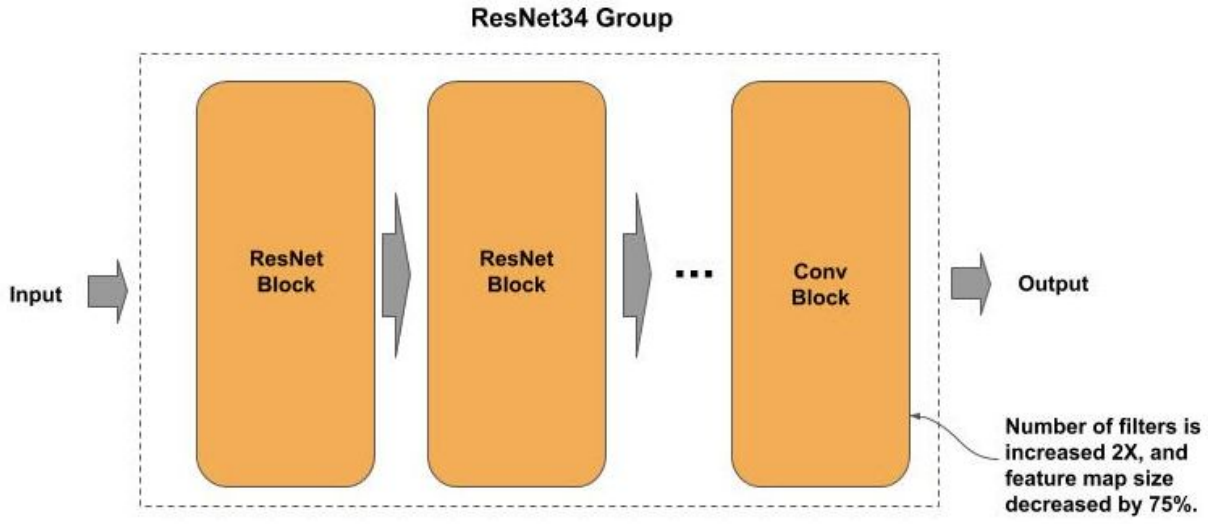
Let's now put the whole network together, using a procedural style. Additionally, we will need to add the entry convolutional layer of ResNet, and then the DNN classifier.

Like we did for the VGG example, we define a procedure (function) for generating the residual block pattern, following the pattern we used in the above code snippet. For our procedure residual_block(), we pass in the number of filters for the block and the input layer (i.e., output from previous layer).

The ResNet architectures take as input a (224, 224, 3) vector. That is, an RGB image (3 channels), of 224 (height) x 224 (width) pixels. The first layer is a basic convolutional layer, consisting of a convolution using a fairly large filter size of 7 x 7. The output (feature maps) are then reduced in size by a max pooling layer.

After the initial convolutional layer, there is a succession of groups of residual blocks, where each successive group doubles the number of filters (similar to VGG). Unlike VGG though, there is no pooling layer between the groups that would reduce the size of the feature maps. Now, if we connected these blocks directly with each other, we have a problem. That is, the input to the next block has the shape based on the previous block's filter size (let's call it X). The next block by doubling the filters will cause the output of that residual block to be double in size (let's call it 2X). The identity link would attempt to add the input matrix (X) and the output matrix (2X). Yeaks, we get an error, indicating we can't broadcast (for add operation) matrices of different sizes.

For ResNet, this is solved by adding a convolutional block between each "doubling" group of residual blocks. The convolutional block doubles the filters to reshape the size and doubles the stride to reduce the feature map size by 75% (i.e., feature pooling).



The output of the last residual block group is passed to a pooling and flattening layer (GlobalAveragePooling2D), which is then passed to a single Dense layer of 1000 nodes (i.e., number of classes).

```
from tensorflow.keras import Model
import tensorflow.keras.layers as layers

def residual_block(n_filters, x):
```

```
    """ Create a Residual Block of Convolutions
        n_filters: number of filters
        x        : input into the block
    """
    shortcut = x
    x = layers.Conv2D(n_filters, (3, 3), strides=(1, 1), padding="same",
                      activation="relu")(x)
    x = layers.Conv2D(n_filters, (3, 3), strides=(1, 1), padding="same",
                      activation="relu")(x)
    x = layers.add([shortcut, x])
    return x

def conv_block(n_filters, x):
    """ Create Block of Convolutions without Pooling
        n_filters: number of filters
        x        : input into the block
    """
    x = layers.Conv2D(n_filters, (3, 3), strides=(2, 2), padding="same",
                      activation="relu")(x)
    x = layers.Conv2D(n_filters, (3, 3), strides=(2, 2), padding="same",
                      activation="relu")(x)
    return x

# The input tensor
inputs = layers.Input(shape=(224, 224, 3))

# First Convolutional layer, where pooled feature maps will be reduced by 75%
x = layers.Conv2D(64, kernel_size=(7, 7), strides=(2, 2), padding='same',
                  activation='relu')(inputs)
x = layers.MaxPool2D(pool_size=(3, 3), strides=(2, 2), padding='same')(x)

# First Residual Block Group of 64 filters
for _ in range(2):
    x = residual_block(64, x)

# Double the size of filters and reduce feature maps by 75% (strides=2, 2) to fit
the next Residual Group
x = conv_block(128, x)

# Second Residual Block Group of 128 filters
for _ in range(3):
    x = residual_block(128, x)

# Double the size of filters and reduce feature maps by 75% (strides=2, 2) to fit
the next Residual Group
x = conv_block(256, x)
```

```
# Third Residual Block Group of 256 filters
for _ in range(5):
    x = residual_block(256, x)

# Double the size of filters and reduce feature maps by 75% (strides=2, 2) to fit
the next Residual Group
x = conv_block(512, x)

# Fourth Residual Block Group of 512 filters
for _ in range(2):
    x = residual_block(512, x)

# Now Pool at the end of all the convolutional residual blocks
x = layers.GlobalAveragePooling2D()(x)

# Final Dense Outputting Layer for 1000 outputs
outputs = layers.Dense(1000, activation='softmax')(x)

model = Model(inputs, outputs)
```

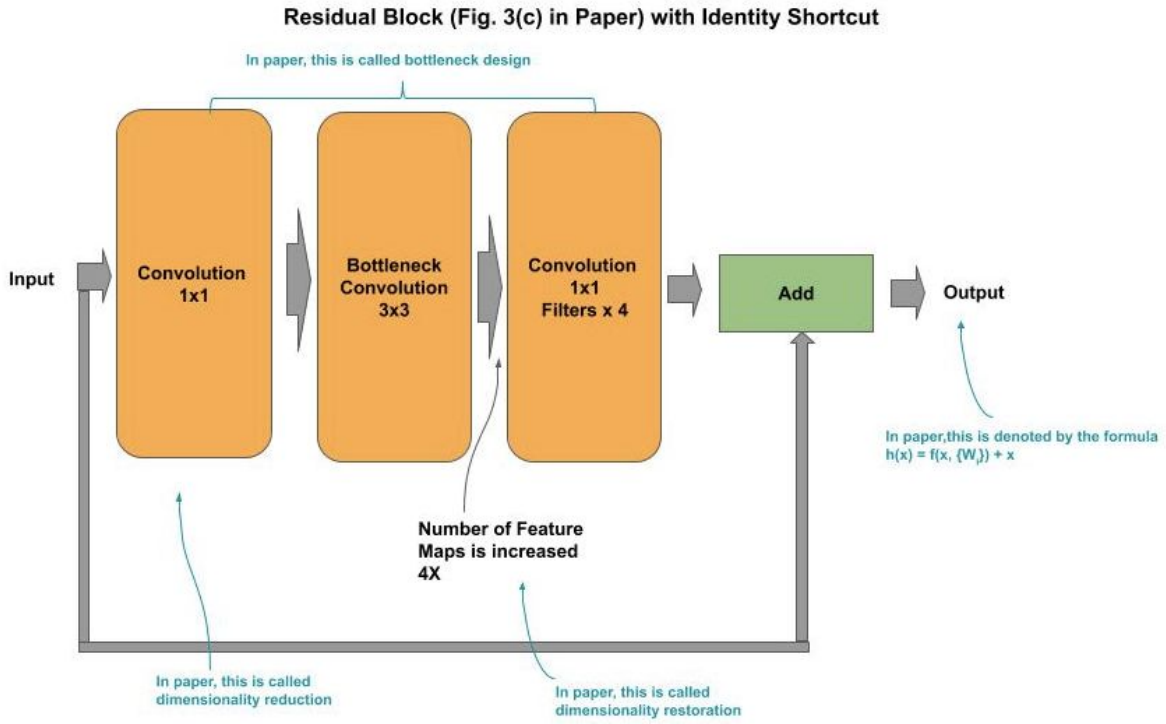
Let's now run model.summary(). We see that the total number of parameters to learn is 21 million. This is in contrast to the VGG16 which has 138 million parameters. So the ResNet architecture is 6 times computationally faster. This reduction is mostly achieved by the construction of the residual blocks. Notice how the DNN backend is just a single output Dense layer. In effect, there is no backend. The early residual block groups act as the CNN frontend doing the feature detection, while the latter residual blocks perform the classification. In doing so, unlike VGG, there was no need for several fully connected dense layers, which would have substantially increased the number of parameters.

Unlike the previous discussion of pooling; where the size of each feature map is reduced accordingly to the size of the stride, GlobalAveragePooling2D is like a supercharged version of pooling where each feature map is replaced by a single value; which in this case is the average of all values in the corresponding feature map. For example, if the input was 256 feature maps, the output would be a 1D vector of size 256. After ResNet, this became the general practice for deep convolutional neural networks to use GlobalAveragePooling2D at the last pooling stage; which benefited from a substantial number of parameters coming into the classifier, without significant loss in representational power.

Another advantage is the identity link, which provided the ability to add deeper layers, without degradation, for higher accuracy.

ResNet50 introduced a variation of the residual block referred to as the bottleneck residual block. In this version, the group of two 3x3 convolution layers are replaced by a group of 1x1, then 3x3, and then 1x1 convolution layer. The first 1x1 convolution performs a dimension reduction reducing the computational complexity, and the last convolutional restores the dimensionality increasing the number of filters by a factor of 4. The middle 3x3 convolution is referred to as the bottleneck convolution, like the neck of a bottle.

The bottleneck residual block allows for deeper neural networks, without degradation, and further reduction in computational complexity.



Below is a code snippet for writing a bottleneck residual block as a reusable function:

```
def bottleneck_block(n_filters, x):
    """ Create a Bottleneck Residual Block of Convolutions
        n_filters: number of filters
        x        : input into the block
    """
    shortcut = x
    x = layers.Conv2D(n_filters, (1, 1), strides=(1, 1), padding="same",
                      activation="relu")(x)
    x = layers.Conv2D(n_filters, (3, 3), strides=(1, 1), padding="same",
                      activation="relu")(x)
    x = layers.Conv2D(n_filters * 4, (1, 1), strides=(1, 1), padding="same",
                      activation="relu")(x)
```

```
    x = layers.add([shortcut, x])
    return x
```

Residual blocks introduced the concept of representational power and representational equivalence. Representational power is how powerful is a block as a feature extractor. The concept of representational equivalence is in the idea of can the block be factored into a lower computational complexity, while maintaining representational power. The design of the residual bottleneck block was demonstrated to maintain representational power of the ResNet34 block, with a lower computational complexity.

Batch Normalization

Another problem with adding deeper layers in a neural network is the *vanishing gradient* problem. This is actually about computer hardware. During training (process of backward propagation and gradient descent), at each layer the weights are being multiplied by very small numbers, specifically numbers less than 1. As you know, two numbers less than one multiplied together make an even smaller number. When these tiny values are propagated through deeper layers they continuously get smaller. At some point, the computer hardware can't represent the value anymore - and hence, the *vanishing gradient*.

The problem is further exacerbated if we try to use half precision floats (16 bit float) for the matrix operations versus single precision (32 bit float). The advantage of the former is that the weights (and data) are stored in half the amount of space and using a general rule of thumb by reducing the computational size in half, we can execute 4 times as many instructions per compute cycle. The problem of course is that with even smaller precision, we will encounter the *vanishing gradient* even sooner.

Batch normalization is a technique applied to the output of a layer (before or after the activation function). Without going into the statistics aspect, it normalizes the shift in the weights as they are being trained. This has several advantages, it smoothes out (across a batch) the amount of change, thus slowing down the possibility of getting a number so small that it can't be represented by the hardware. Additionally, by narrowing the amount of shift between the weights, convergence can happen sooner using a higher learning rate and reducing the overall amount of training time. Batch normalization is added to a layer in **TF.Keras** with the BatchNormalization() class.

In earlier implementations, batch normalization was implemented as post-activation. That is, the batch normalization would occur after the convolution and dense layers. At the time, it was debated whether the batch normalization, before or after the activation function.

Below is a code example of using post-activation batch normalization in both before and after an activation function, in both a convolution and dense layer.

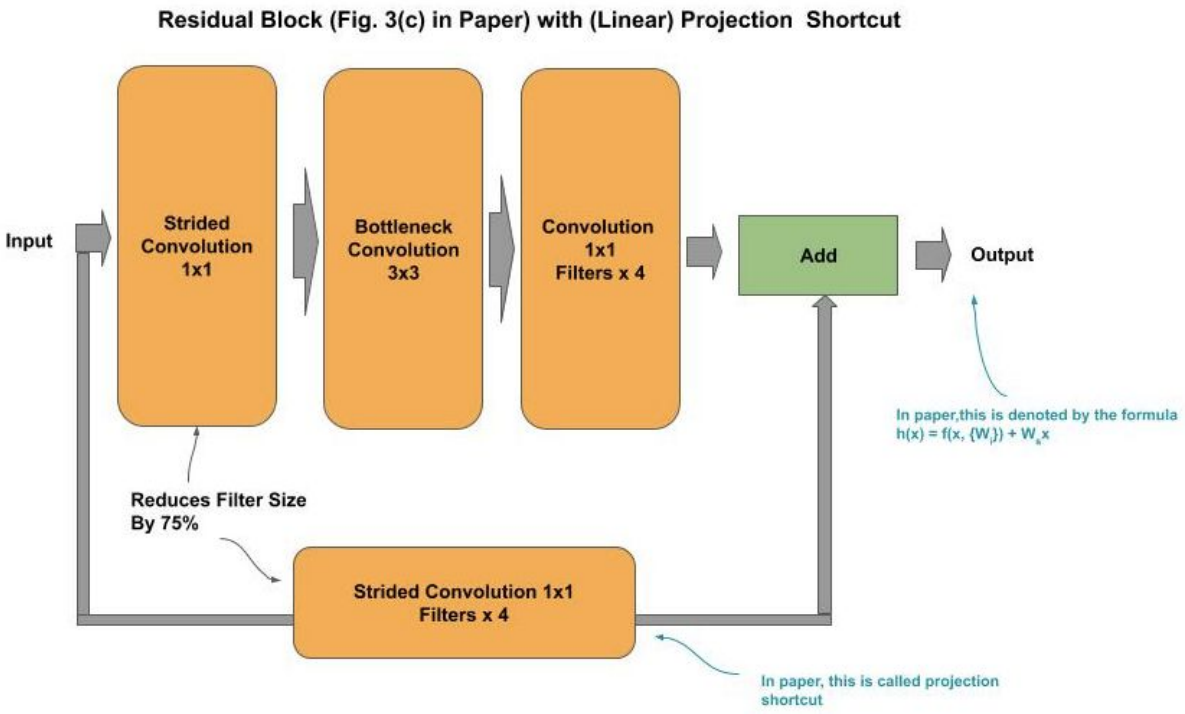
```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, ReLU, BatchNormalization, Flatten
from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Conv2D(64, (3, 3), strides=(1, 1), padding='same',
                 input_shape=(128, 128, 3)))
# Add a batch normalization (when training) to the output before the activation
# function.
model.add(BatchNormalization())
model.add(ReLU())

model.add(Flatten())

model.add(Dense(4096))
model.add(ReLU())

# Add a batch normalization (when training) to the output after the activation
# function.
model.add(BatchNormalization())
```

ResNet50

ResNet50 is a well-known model, which is commonly reused as a stock model, transfer learning, shared layers in objection detection, and performance benchmarking. There are three versions of the model, referred to **v1**, **v2** and **v3**.

ResNet50 v1 formalized the concept of a convolutional group. A convolutional group is a set of convolutional blocks which share a common configuration, such as the number of filters. In v1, the neural network is decomposed into groups, where each group doubles the number of filters from the previous group.

Additionally, the concept of a separate convolution block to double the number of filters was removed and replaced by a residual block that uses linear projection. Each group starts with a residual block using linear projection on the identity link to double the number of filters, while the remaining residual blocks pass the input directly to the output for the matrix add operation. Additionally, the first 1x1 convolution in the residual block with linear projection uses a stride of two (feature pooling), which is also known as a strided convolution, reducing the feature map sizes by 75%.



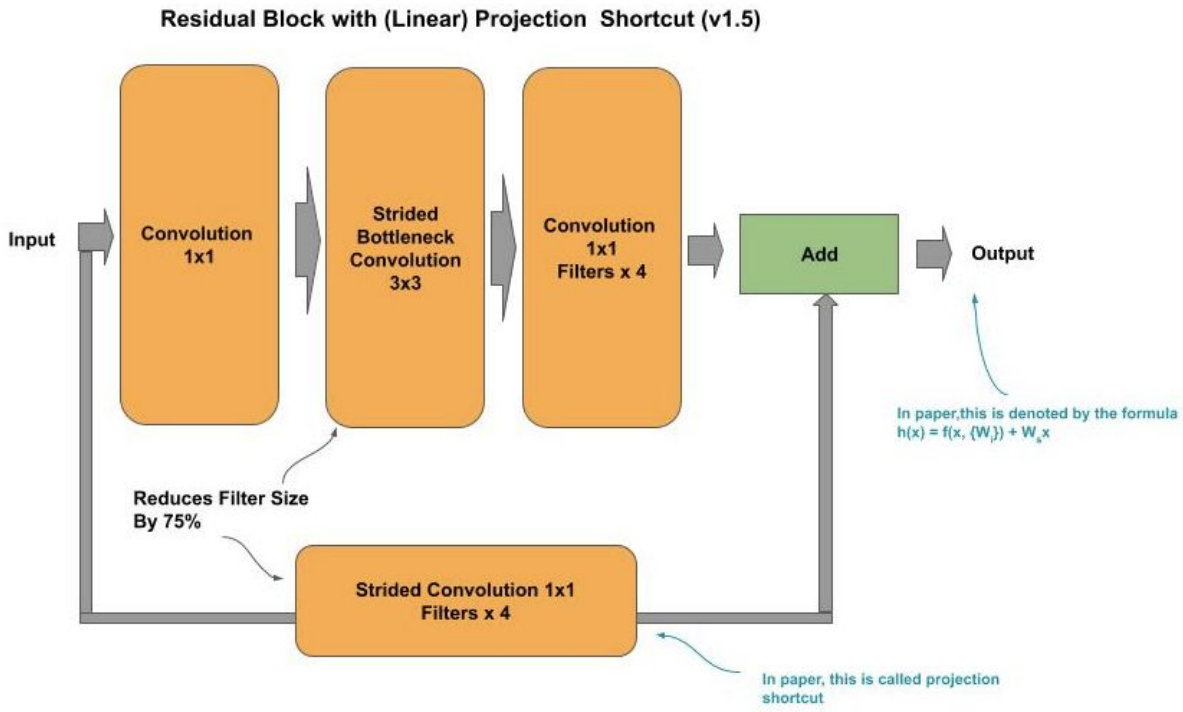
Below is an implementation of **ResNet50 v1** using the bottleneck block combined with batch normalization.

```
from tensorflow.keras import Model
import tensorflow.keras.layers as layers

def identity_block(x, n_filters):
    """ Create a Bottleneck Residual Block of Convolutions
        n_filters: number of filters
        x        : input into the block
    """
    shortcut = x

    # dimensionality reduction
    x = layers.Conv2D(n_filters, (1, 1), strides=(1, 1))(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # bottleneck layer
    x = layers.Conv2D(n_filters, (3, 3), strides=(1, 1), padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # dimensionality restoration
    x = layers.Conv2D(n_filters * 4, (1, 1), strides=(1, 1))(x)
```

```
    x = layers.BatchNormalization()(x)

    x = layers.add([shortcut, x])
    x = layers.ReLU()(x)

    return x

def projection_block(x, n_filters, strides=(2,2)):
    """ Create Block of Convolutions with feature pooling
        Increase the number of filters by 4X
        x        : input into the block
        n_filters: number of filters
    """
    # construct the identity link
    # increase filters by 4X to match shape when added to output of block
    shortcut = layers.Conv2D(4 * n_filters, (1, 1), strides=strides)(x)
    shortcut = layers.BatchNormalization()(shortcut)

    # construct the 1x1, 3x3, 1x1 convolution block

    # feature pooling when strides=(2, 2)
    x = layers.Conv2D(n_filters, (1, 1), strides=strides)(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    x = layers.Conv2D(n_filters, (3, 3), strides=(1, 1), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # increase the number of filters by 4X
    x = layers.Conv2D(4 * n_filters, (1, 1), strides=(1, 1))(x)
    x = layers.BatchNormalization()(x)

    # add the  identity link to the output of the convolution block
    x = layers.add([x, shortcut])
    x = layers.ReLU()(x)

    return x

# The input tensor
inputs = layers.Input(shape=(224, 224, 3))

# First Convolutional layer, where pooled feature maps will be reduced by 75%
x = layers.ZeroPadding2D(padding=(3, 3))(inputs)
x = layers.Conv2D(64, kernel_size=(7, 7), strides=(2, 2), padding='valid')(x)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
```

```
x = layers.ZeroPadding2D(padding=(1, 1))(x)
x = layers.MaxPool2D(pool_size=(3, 3), strides=(2, 2))(x)

x = projection_block(64, x, strides=(1,1))

# First Residual Block Group of 64 filters
for _ in range(2):
    x = identity_block(64, x)

# Double the size of filters and reduce feature maps by 75% (strides=2, 2) to fit
the next Residual Group
x = projection_block(128, x)

# Second Residual Block Group of 128 filters
for _ in range(3):
    x = identity_block(128, x)

# Double the size of filters and reduce feature maps by 75% (strides=2, 2) to fit
the next Residual Group
x = projection_block(256, x)

# Third Residual Block Group of 256 filters
for _ in range(5):
    x = identity_block(256, x)

# Double the size of filters and reduce feature maps by 75% (strides=2, 2) to fit
the next Residual Group
x = projection_block(512, x)

# Fourth Residual Block Group of 512 filters
for _ in range(2):
    x = identity_block(512, x)

# Now Pool at the end of all the convolutional residual blocks
x = layers.GlobalAveragePooling2D()(x)

# Final Dense Outputting Layer for 1000 outputs
outputs = layers.Dense(1000, activation='softmax')(x)

model = Model(inputs, outputs)
```

v1.5 introduced a refactoring of the bottleneck design and further reducing computational complexity, while maintaining representational power. The feature pooling (strides=2) in the residual block with linear projection, is moved from the first 1x1 convolution to the 3x3 convolution, reducing computational complexity and increasing results on ImageNet by 0.5 percent.




```
def projection_block(x, n_filters, strides=(2,2)):
    """ Create Block of Convolutions with feature pooling
        Increase the number of filters by 4X
        x        : input into the block
        n_filters: number of filters
    """
    # construct the identity link
    # increase filters by 4X to match shape when added to output of block
    shortcut = layers.Conv2D(4 * n_filters, (1, 1), strides=strides)(x)
    shortcut = layers.BatchNormalization()(shortcut)

    # construct the 1x1, 3x3, 1x1 convolution block

    # dimensionality reduction
    x = layers.Conv2D(n_filters, (1, 1), strides=(1, 1))(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # bottleneck layer
    # feature pooling when strides=(2, 2)
    x = layers.Conv2D(n_filters, (3, 3), strides=(2, 2), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # dimensionality restoration
```

```
    # increase the number of filters by 4X
    x = layers.Conv2D(4 * n_filters, (1, 1), strides=(1, 1))(x)
    x = layers.BatchNormalization()(x)

    # add the  identity link to the output of the convolution block
    x = layers.add([x, shortcut])
    x = layers.ReLU()(x)
    return x
```

v2 introduced the concept of pre-activation batch normalization, where the batch normalization and activation function are placed prior (instead of after) the corresponding convolution or dense layer. This has now become the common practice, as depicted below for implementation of the residual block with identity link in v2:

```
def identity_block(x, n_filters):
    """ Create a Bottleneck Residual Block of Convolutions
        n_filters: number of filters
        x        : input into the block
    """
    shortcut = x

    # dimensionality reduction
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    x = layers.Conv2D(n_filters, (1, 1), strides=(1, 1))(x)
    # bottleneck layer
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    x = layers.Conv2D(n_filters, (3, 3), strides=(1, 1), padding="same")(x)

    # dimensionality restoration
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    x = layers.Conv2D(n_filters * 4, (1, 1), strides=(1, 1))(x)

    x = layers.add([shortcut, x])
    return x
```