Idiomatic Programmer

Learning Keras

Handbook 1: Computer Vision Models

Andrew Ferlitsch, Google AI

The Idiomatic Programmer - Learning Keras

Handbook 1: Computer Vision Models

Part 1 - Deep Neural Networks

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, and coursework for coding school, but I am always looking for new ways to effectively teach.

Welcome to my latest approach, the idiomatic programmer. My audience are software engineers who are proficient in a non-AI framework, like Angular, React, Django, etc. You know at least the basics of Python. It's okay if you still struggle with what is a comprehension, 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 its okay.

You want 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.

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

Identify the Type of Model for the Problem

The **Keras** framework is about building neural network models. Think of it as a bunch of nodes with lines connected between them. Hum, that sounds like a graph. Data flows from one node to another node in a specific direction. Hum, that sounds like a directed graph. The graph has an entry point (the input layer) and an exit point (the output layer). Between the entry and exit point are levels of nodes, which are called the layers. At depth 0 is the input layer. At depth n is the output layer. Everything in between is an internal layer, which can have names like hidden, pooling, convolution, dropout, activation, normalization, etc. Let's not get bogged down by this, they are just types of nodes. In **Keras**, these nodes are represented by class objects. Each of these classes takes parameters which adjust how the node works. Alas, the class objects and corresponding methods are the buttons to push and the parameters are the levers to adjust. That's it.

Neural Network layouts fall into four primary categories:

1. DNN (Deep Neural Networks) - These are good for numerical solutions.
2. CNN (Convolutional Neural Networks) - These are good for computer vision solutions and (audio) signal processing.
3. RNN (Recurrent Neural Networks) - These are good for text and speech recognition, and anything else that has a time sequence nature to it.
4. GAN (Generative Adversarial Networks) - These are good for synthesizing creative works, and reconstruction.

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 (Keras, 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 (preferably version 3). 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 **Keras** framework, the **numpy** package, and **TensorFlow** as the backend to **Keras**. Guess what their names are in the registry, keras, tensorflow and numpy - so obvious! Let's do it together. Go to the command line and issue the following:

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

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 **Keras** framework (which you installed earlier with pip), which by default uses **TensorFlow** as the backend, 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 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>
```

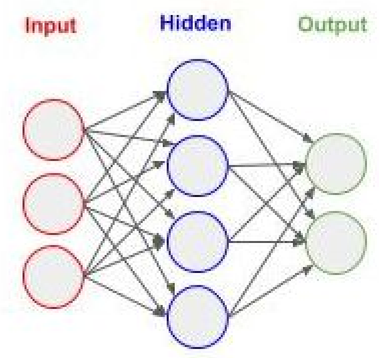
This is showing you what `Input(shape=(13,))` evaluates to. It produces a tensor object by the name 'input_1:0'. This name will be useful later in assisting you in debugging your models. The '?' in shape shows that the input object takes an unbounded number of entries (your samples or rows) of 13 elements each. That is, at run-time it will bind the number of one dimensional vectors of 13 elements to the actual number of samples (rows) you pass in. 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, when coding a forward feed network in **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 Method Approach

The Sequential 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.

```
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 Method (layers) Approach

The layers approach (referred to in Keras as the Functional method) is more advanced. 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 *bias*. This is what the neural network will *"learn"* during training. This operation will otherwise be invisible to you.

The Dense() Layer

In **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 "n" number of nodes, and is fully connected to the previous layer. Let's continue and define in **Keras** a three layer neural network, using the Sequential 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 ("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.

Let's first do this using the Sequential method. 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().

```python
from keras import Sequential
from 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.

```python
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 layers (Functional API) method. We start by creating an input vector by instantiating an Input() class object. The (positional) parameter to the Input() class 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 surrounded by a parenthesis. By 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() class 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() class 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() class 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() class 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 keras import Input, Model
from 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.

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).

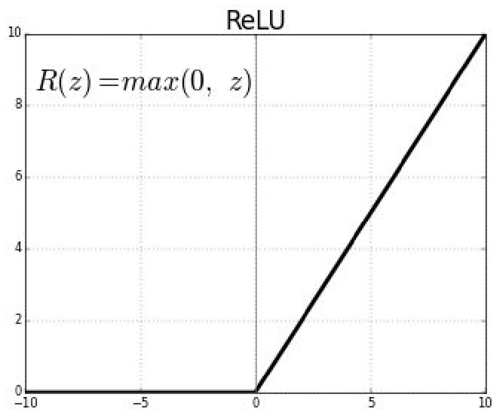


image source: https://towardsdatascience.com

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

```
from keras import Sequential
from 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 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 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's that 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.

At the next (hidden) layer you see 110 params. That's 10 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

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.

```python
from keras import Sequential
from 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 in 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 sample. 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 networks. 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 (*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.

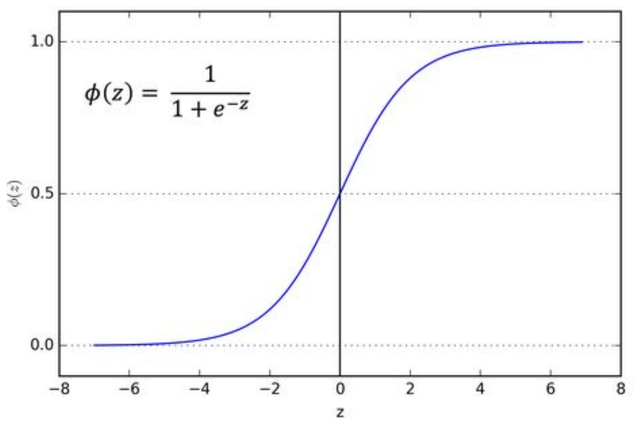


image source: https://towardsdatascience.com

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 keras import Sequential
from 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 **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 keras import Sequential
from 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 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 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 layers approach. Notice how we repetitively 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 just 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 keras import Sequential, Model, Input
from 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 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 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) more than one class. 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.

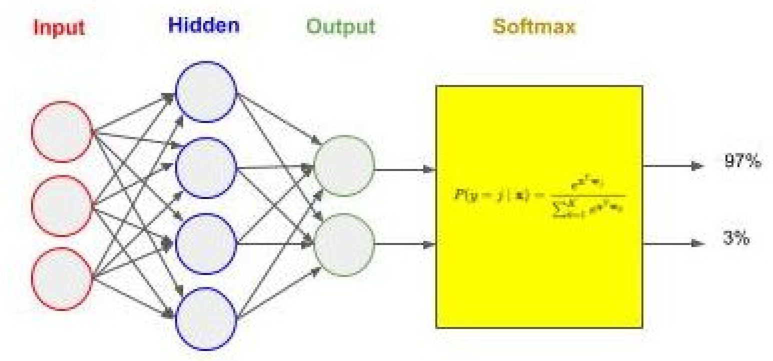
We can already see we will have some problems. For example, men on average as adults are taller than women. But during the preteen years, girls tend to be taller than boys. We know on average that men get heavier early in their adult years in comparison to their teenage years, but women on average are 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, 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 (https://www.ncbi.nlm.nih.gov/pubmed/3579170).

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%). 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 had 97% confidence 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 keras import Sequential
from 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'])
```

A Note on K.backend

If you've started looking at other code snippets on the web, you will find the use of K.backend. The **Keras** framework is an abstraction layer on top of other computational graph based machine learning networks. These are known as the backend, such as **TensorFlow©**, **Theano©** and **CNTK©**.

The backend module gives you direct access to the implementation in the backend. By default, **TensorFlow** is the backend, and **Theano** is being phased out. I would recommend not using this syntax because it risks using constructs that may (or have) become deprecated.

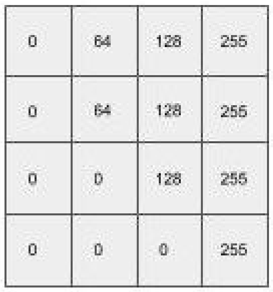
Below is an example of a code snippet where you might see someone directly referring to an implementation in the backend, in this case using the backend's implementation of the hyperbolic tangent (tanh) activation function.

```
from keras import Sequential
from keras.layers import Dense, Activation
# import the backend module and refer to it with the alias K
from keras import backend as K

model = Sequential()
# Add the first (input) layer (10 nodes) and use the backend's implementation
# of tanh for the activation function
model.add(Dense(10, activation=K.tanh, input_shape=(13,)))
```

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.

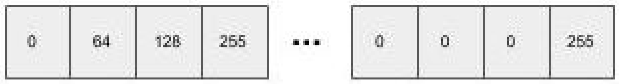
We will need to make one change though. A grayscale image is a matrix (2D array). Think of them as a grid, sized height x width, where the width are the columns and the height are the rows. A DNN though takes as input a vector (1D array). Yeaks!



0	64	128	255
0	64	128	255
0	0	128	255
0	0	0	255

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.



0	64	128	255	...	0	0	0	255

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.

```
from keras import Sequential
from 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))
model.add(ReLU())
# Add the second (hidden) layer (512 nodes).
model.add(Dense(512))
model.add(ReLU())
# Add the third (output) layer (10 nodes) with Sigmoid activation function.
model.add(Dense(10))
model.add(Activation('sigmoid'))

# 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, to be discussed in subsequent tutorial, a dataset is split into training data and test 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 also known as running the trained neural network in inference mode (prediction). In a train/test split (train/eval/test discussed in subsequent tutorial), 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 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('sigmoid'))

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

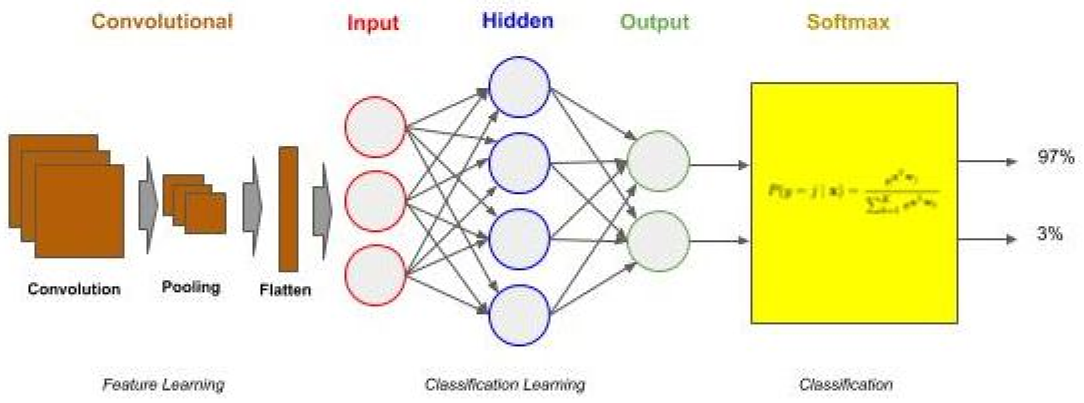
Next

In the second part, we will cover the principal behind convolutional neural networks (CNN) and basic design patterns for constructing.

Part 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".

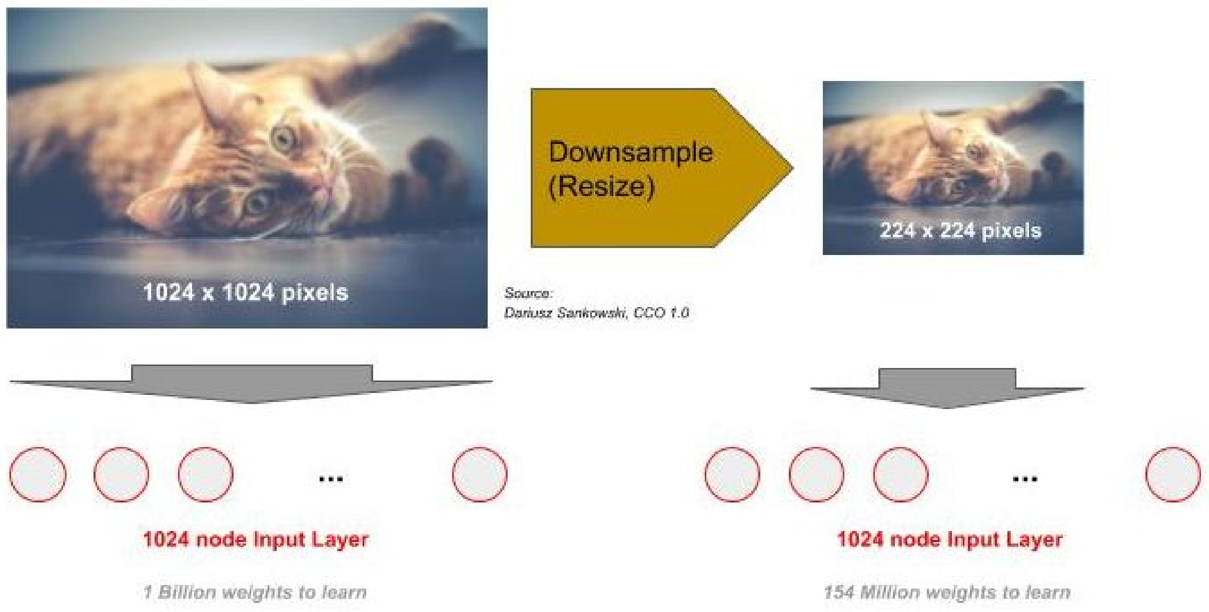


CNN Classifier

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 input layer has 1024 nodes. The number of weights to *"update and learn"* 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).

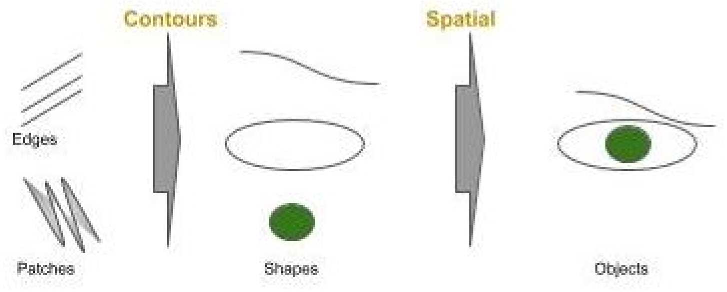


So training on real-world images was out of reach with neural networks until the discovery of using convolutional layers. To begin with, a convolutional layer is a frontend to a neural network, which transforms the images from a high dimensional pixel based image to a substantially lower dimensionality 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. For basic 2D renderings like digits and letters, this works (brute-force), but for everything else, it's not going to work.

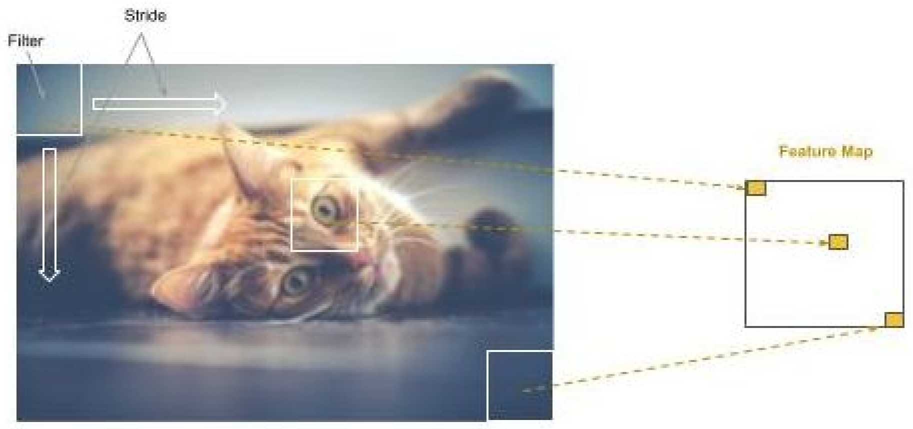
Feature Detection

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 have 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. 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.



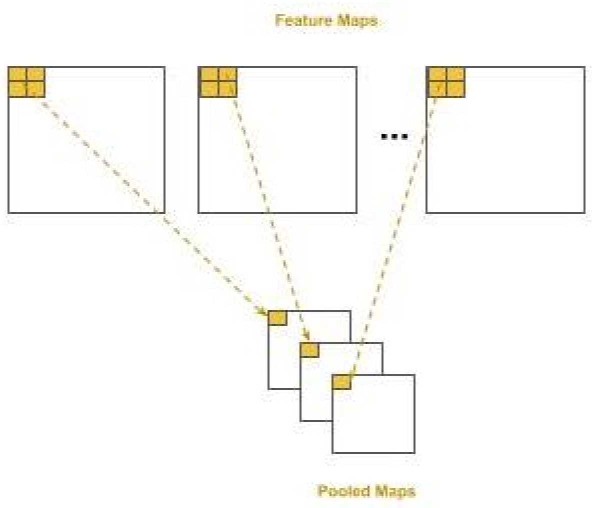
When there are multiple convolutional layers, the common practice is to keep the same or increase the number of filters on deeper layers, and to use stride of 1 on the first layer and 2 on deeper layers. The increase in filters provides the means to go from coarse detection of features to more detailed detection within coarse features, while the increase in stride offsets the increase in size of retained data.

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

Pooling

Even though each feature map generated is less than the size of the image (e.g., ~75% to 90% reduction), 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.

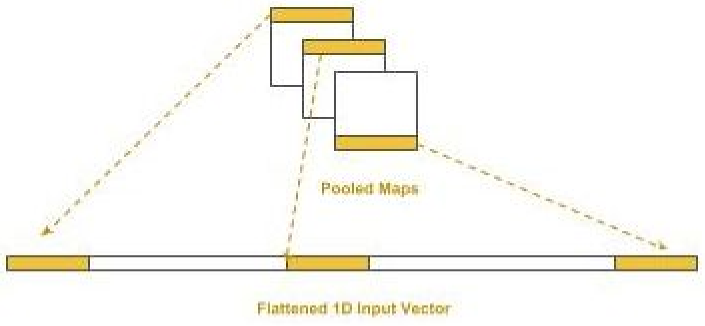
This step is referred to as pooling. Pooling is the same as downsampling (or sub-sampling); whereby the feature maps are resized to a smaller dimension using either max (downsampling) or mean (sub-sampling) pixel average within the feature map. In pooling, we set the size of the area to pool as a NxM 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.



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 **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` method style 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 tutorial, we will stick with 2D convolutionals, which are the common practice for computer vision.

The output from the convolution layer is then passed through a rectified linear unit activation, 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. 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 suffice for our purposes to say that in almost all cases, you will use the value `same`; it's just that the default is `not same` 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 keras.models import Sequential
from keras.layers import Dense, ReLU, Activation
# Kera's Convolutional Neural Network components
from 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 samples (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). 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 keras.models import Sequential
from keras.layers import Dense

# Kera's Convolutional Neural Network components
from 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 layers approach. 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.

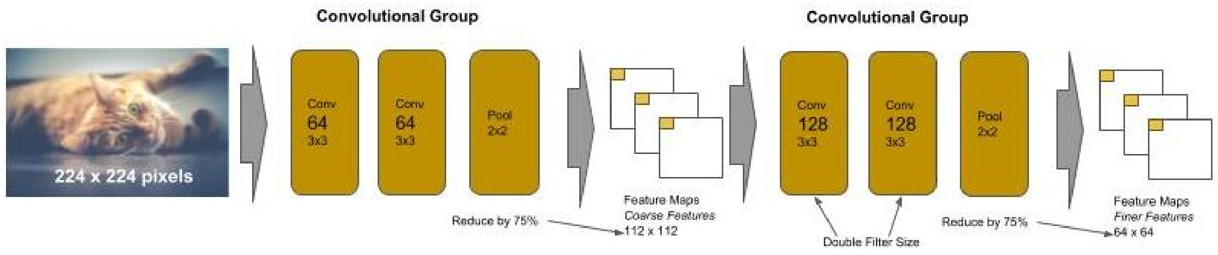
For example, for the first `Dense` layer, when invoked as a callable, we pass as the parameter the layer object for the `Flatten` layer. As a callable, this will cause the `Flatten` layer and the first `Dense` layer to be fully connected (i.e., each node in the `Flatten` layer will be connected to every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

```
from keras import Input, Model
from keras.layers import Dense
from 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)
output = Dense(26, activation='softmax')(layer)

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

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 keras import Sequential
from 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) call 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 keras import Sequential
from 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'])
```

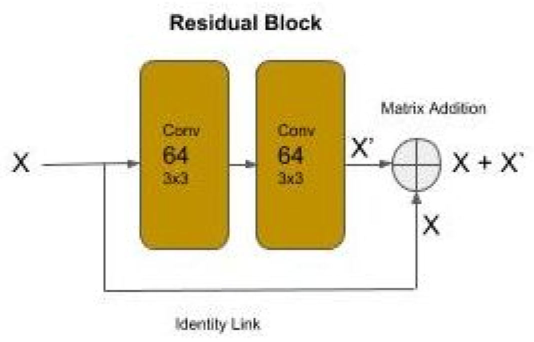
Try running `model.summary()` on both examples and you will see that the output is identical.

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. We won't go into why, but just discuss how this phenomenon can be addressed. Residual blocks allow neural networks to be built with deeper layers without a degradation in performance.

Below is a code snippet showing how a residual block can be coded in **Keras** using the `Sequential` 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.

```
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])
```

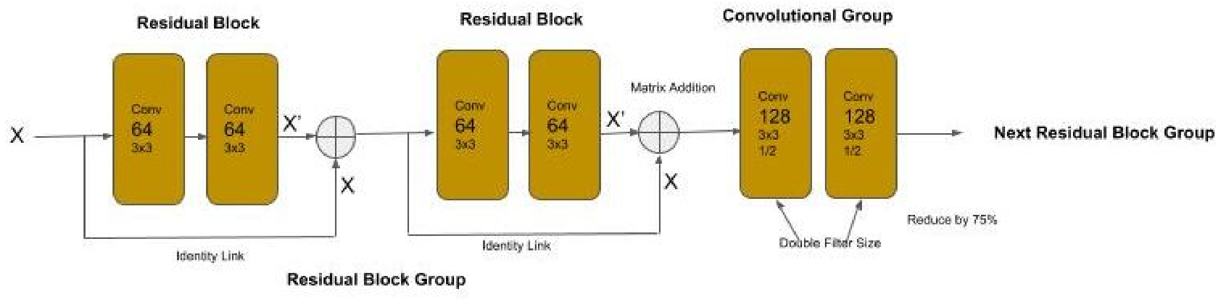
Let's now put the whole network together, using a procedural style. Additionally, we will need to add the entry convolutional layer of ResNet, which is not a residual block, and then the DNN backend.

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 (similar to VGG) each successive group doubles the number of filters. 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 size by 75%.



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 (number of classes).

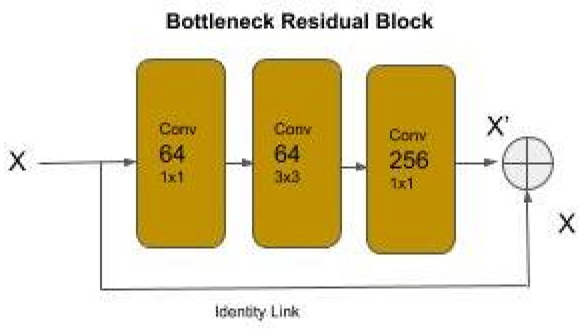
```
from keras import Model
import keras.layers as layers

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(3):
    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)(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 of 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 top residual block groups act as the CNN frontend doing the feature detection, while the bottom 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.

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 1x1 convolutions perform 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 bottleneck residual group allows for deeper neural networks, without degradation, and further reduction in computational complexity.



Below is a code snippet for writing a bottleneck 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
```

Batch Normalization

Another problem with adding deeper layers in a neural network is the *vanishing gradient* problem. This is actually about the 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 **Keras** with the `BatchNormalization()` class. It is though still of debate, whether the best practice is to add it before or after the activation function.

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

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

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

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

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))(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)

    x = layers.Conv2D(n_filters * 4, (1, 1), strides=(1, 1))(x)
    x = layers.BatchNormalization()(x)

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

def conv_block(n_filters, x, strides=(2,2)):
    """ Create Block of Convolutions with feature pooling
        Increase the number of filters by 4X
        n_filters: number of filters
        x        : input into the block
    """
    # 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 = conv_block(64, x, strides=(1,1))

# First Residual Block Group of 64 filters
for _ in range(3):
    x = bottleneck_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 = bottleneck_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 = bottleneck_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 = bottleneck_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)(x)

model = Model(inputs, outputs)
```

Next

In the next part, we will discuss the principles and design patterns of the next evolution of wide layers in convolutional neural networks.

Part 3 - Wide Convolutional Networks - ResNeXt, Inception

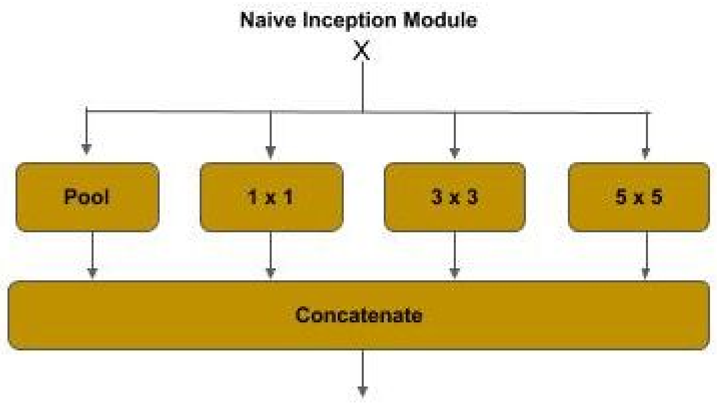
Inception v1 (GoogLeNet) - Wide Convolutional Neural Network

In this section, we will cover the design patterns for several wide convolutional neural networks. Up to now, we've focused on networks with deeper layers, block layers and shortcuts. Starting in 2014 with Inception v1 (GoogleNet) and 2015 with ResNeXt (Microsoft Research) and Inception v2, neural network designs moved into wide layers (vs. deeper layers). Essentially, a wide layer is having multiple convolutions in parallel and then concatenating their outputs.

The **GoogLeNet** (Inception v1) won the 2014 ImageNet contest and introduced the 'inception module". The inception module is a convolutional layer with parallel convolutions of different sizes, with the outputs concatenated together. The principle behind it was that instead of trying to pick the best filter size for a layer, each layer has multiple filter sizes in parallel, and the layer learns which size is the best.

Naive Inception Module

The figure below shows the naive inception module, which demonstrates the principle behind the inception module. In a conventional convolution block, the output from the convolution layer is passed through a pooling layer for dimensionality reduction. The output from a previous convolution layer is branched. One branch is passed through a pooling layer for dimensionality reduction, as in a conventional convolutional layer, and the remaining branches are passed through convolution layers of different sizes (i.e., 1x1, 3x3, 5x5). The outputs from the pooling and the other convolution layers are then concatenated together.



The theory behind it, is that the different filter sizes capture different levels of detail. The 1x1 convolution captures fine details of features, while the 5x5 captures more abstract features. The code below demonstrates a naive inception module. The input from a previous layer x is branched and passed through a max pooling layer, 1x1, 3x3 and 5x5 convolution, which are then concatenated together.

```
# The inception branches (where x is the previous layer)
x1 = layers.MaxPooling2D((3, 3), strides=(1,1), padding='same')(x)
x2 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x3 = layers.Conv2D(96, (3, 3), strides=(1, 1), padding='same')(x)
x4 = layers.Conv2D(48, (5, 5), strides=(1, 1), padding='same')(x)

# concatenate the filters
x = layers.concatenate([x1, x2, x3, x4])
```

By setting `padding='same'`, the height and width dimensions are preserved. This allows concatenating the filters together. For example, if the above the input to this layer was 28x28x256, the dimensions at the branch layers would be:

x1 (pool)	: (?, 28, 28, 256)
x2 (1x1)	: (?, 28, 28, 64)
x2 (3x3)	: (?, 28, 28, 96)
x3 (5x5)	: (?, 28, 28, 48)

After the concatenation, the output would be:

x (concat)	: (?, 28, 28, 464)

A `summary()` for these layers shows 544K parameters to train.

```
max_pooling2d_161 (MaxPooling2D (None, 28, 28, 256)  0           input_152[0][0]
__________________________________________________________________________________
_______________
conv2d_13130 (Conv2D)           (None, 28, 28, 64)   16448       input_152[0][0]
__________________________________________________________________________________
_______________
conv2d_13131 (Conv2D)           (None, 28, 28, 96)   221280      input_152[0][0]
__________________________________________________________________________________
_______________
conv2d_13132 (Conv2D)           (None, 28, 28, 48)   307248      input_152[0][0]
__________________________________________________________________________________
_______________
concatenate_429 (Concatenate)   (None, 28, 28, 464)  0
```

```
max_pooling2d_161[0][0]
                                                                     conv2d_13130[0][0]
                                                                     conv2d_13131[0][0]
                                                                     conv2d_13132[0][0]
=======================================================================================
================
Total params: 544,976
Trainable params: 544,976
```

If the `padding='same'` argument is left out (defaults to padding='valid'), the shapes would be instead:

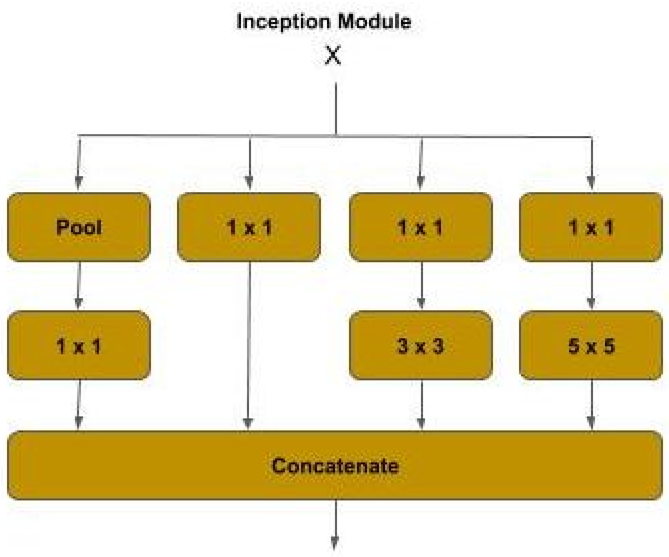
x1 (pool)	: (?, 26, 26, 256)
x2 (1x1)	: (?, 28, 28, 64)
x2 (3x3)	: (?, 26, 26, 96)
x3 (5x5)	: (?, 24, 24, 48)

Since the width and height dimensions do not match, if one tried to concatenate these layers, one would get an error:

ValueError: A `Concatenate` layer requires inputs with matching shapes except for the concat axis. Got inputs shapes: [(None, 26, 26, 256), (None, 28, 28, 64), (None, 26, 26, 96), (None, 24, 24, 48)]

Inception v1 Module

In the **GoogLeNet**, a further dimensionality reduction was introduced by adding a 1x1 convolution (bottleneck) to the pooling, 3x3 and 5x5 branches. This dimension reduction reduced the overall computational complexity by nearly ⅔.



Below is an example of an Inception v1 module:

```
# The inception branches (where x is the previous layer)
x1 = layers.MaxPooling2D((3, 3), strides=(1,1), padding='same')(x)
x1 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x1)
x2 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x3 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x3 = layers.Conv2D(96, (3, 3), strides=(1, 1), padding='same')(x3)
x4 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x4 = layers.Conv2D(48, (5, 5), strides=(1, 1), padding='same')(x4)

# concatenate the filters
x = layers.concatenate([x1, x2, x3, x4])
```

A `summary()` for these layers shows 198K parameters to train.

```
max_pooling2d_162 (MaxPooling2D (None, 28, 28, 256)  0          input_153[0][0]
__________________________________________________________________________________
_______________
conv2d_13135 (Conv2D)          (None, 28, 28, 64)   16448      input_153[0][0]
__________________________________________________________________________________
_______________
conv2d_13137 (Conv2D)          (None, 28, 28, 64)   16448      input_153[0][0]
__________________________________________________________________________________
_______________
conv2d_13133 (Conv2D)          (None, 28, 28, 64)   16448
max_pooling2d_162[0][0]
__________________________________________________________________________________
```

```
______________
conv2d_13134 (Conv2D)          (None, 28, 28, 64)   16448       input_153[0][0]
__________________________________________________________________________________
______________
conv2d_13136 (Conv2D)          (None, 28, 28, 96)   55392       conv2d_13135[0][0]
__________________________________________________________________________________
______________
conv2d_13138 (Conv2D)          (None, 28, 28, 48)   76848       conv2d_13137[0][0]
__________________________________________________________________________________
______________
concatenate_430 (Concatenate)  (None, 28, 28, 272)  0           conv2d_13133[0][0]
                                                                conv2d_13134[0][0]
                                                                conv2d_13136[0][0]
                                                                conv2d_13138[0][0]
==================================================================================
===============
Total params: 198,032
Trainable params: 198,032
```

The diagram below shows a block overview of Inception v1 (GoogLeNet). We have presented the architecture differently than the conventional diagrams, for the purpose of drawing one's attention to the block and group structure of the neural network. GoogLeNet consists of four groups:

Stem Convolutional Group :

This is the entry point into the neural network. The inputs (images) are processed by a sequential (deep) set of convolutions and max pooling, much like a conventional ConvNet.

Wide Convolutional Group :

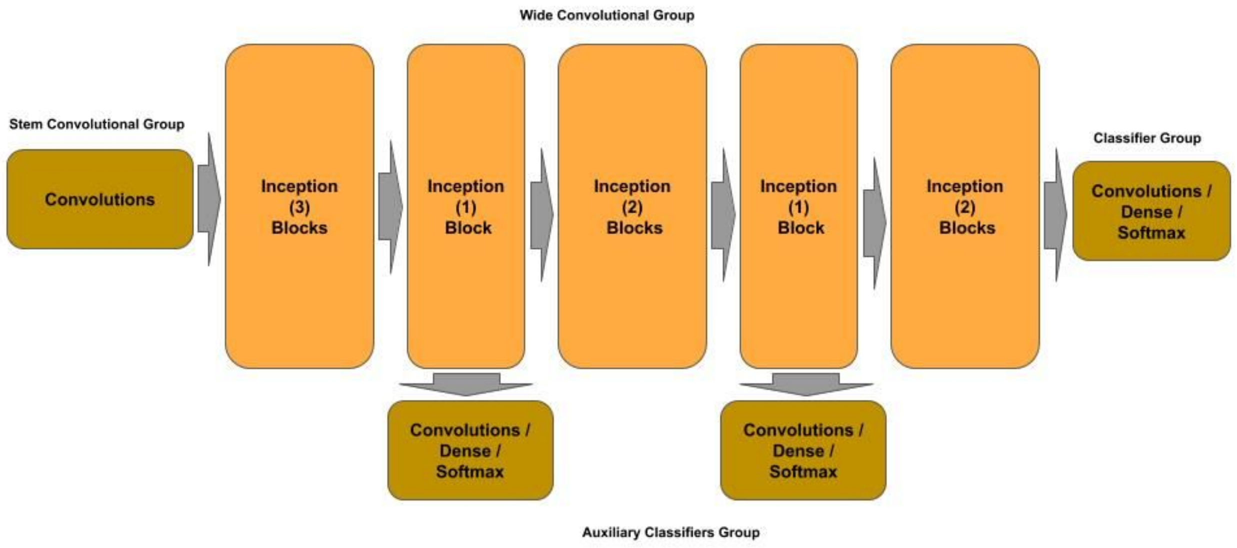
This is a set of seven inception blocks in five groups. The wider blocks in the diagram represent a group of two or three inception blocks, and the thinner ones are a single inception block, for a total of nine inception blocks. The fourth and seventh block (single blocks) are separated out to highlight they have an additional component, referred to as the auxiliary classifier.

Auxiliary Classifiers Group :

This is a set of two classifiers, acting as auxiliary (aids) in training the neural network, each consisting of a convolutional layer, a dense layer and a final softmax activation function.

Classifier Group

This is a single and final classifier in both training the neural network and in prediction (inference).



The GoogleLeNet architecture introduced the concept of an auxiliary classifier. The principle here is that as a neural network gets deeper in layers (i.e., the front layers get further away from the final classifier) the front layers are more exposed to a vanishing gradient and increased time (e.g., number of epochs) to train the weights in the front most layers.

The theory is that at the semi-deep layers, there is some information to predict (classify) what the input is, albeit with less accuracy than the final classifier. These earlier classifiers are closer to the front layers and thus less prone to a vanishing gradient. During training, the cost function becomes a combination of the losses of the auxiliary classifiers and the final classifier.

ResNeXt - Wide Residual Neural Network

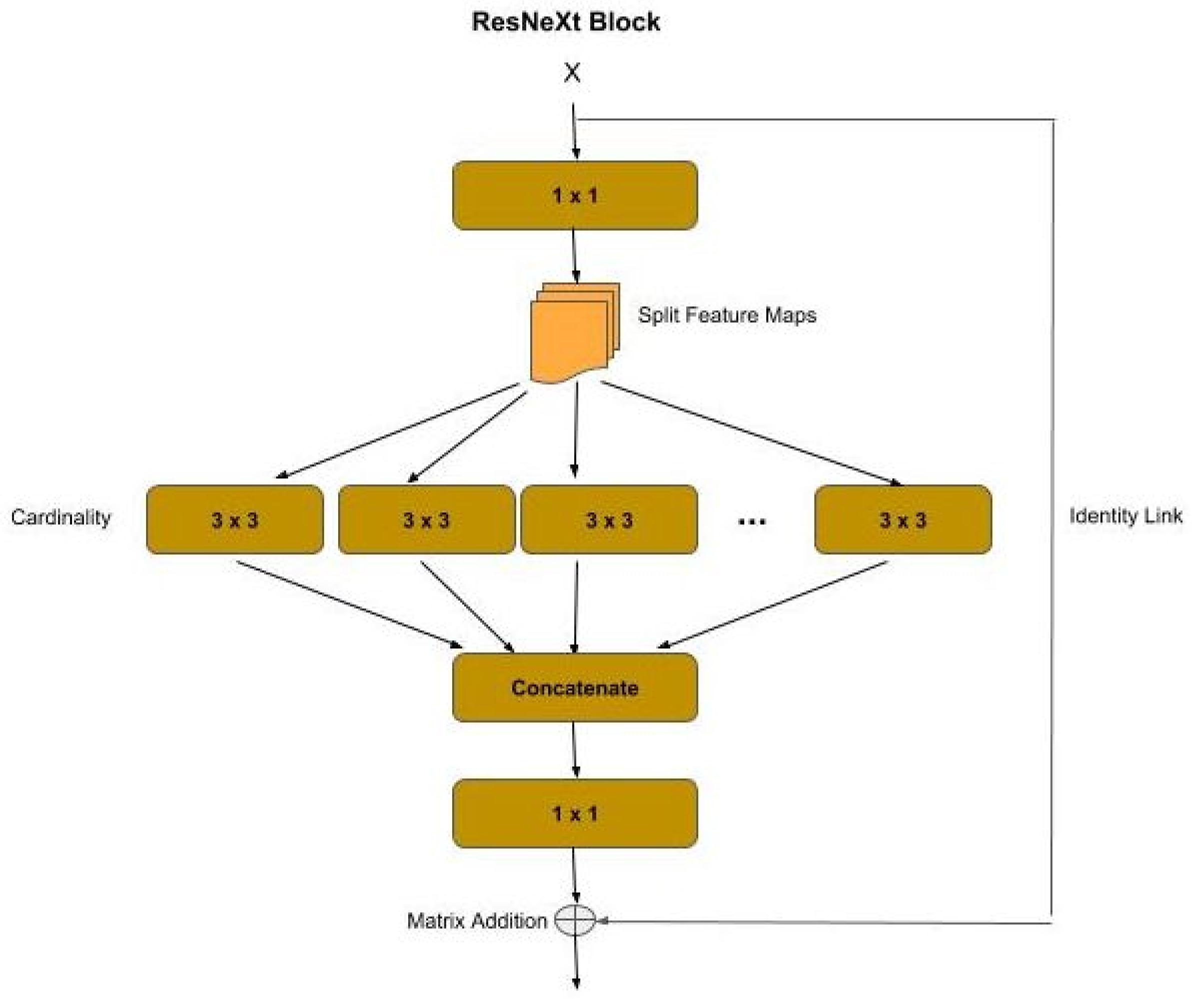
ResNeXt (Microsoft Research) was the winner of the 2015 ImageNet competition and introduced the concept of a wide residual (block) neural network and cardinality (i.e., width). This architecture uses a split, branch and merge method of parallel convolutional blocks. The number of parallel convolutional blocks (i.e., width) is referred to as the cardinality. For example, in the 2015 competition, the ResNeXt architecture used a cardinality of 32, meaning each ResneXt layer consisted of 32 parallel convolutional blocks.

At each ResNeXt layer, the input from the previous layer is split across the parallel convolutional blocks, and the output (feature maps) from each convolutional block is concatenated, and finally the input to the layer is matrix added to the concatenated output (identity link) to form the residual block.

On blogs and tutorials throughout the Internet, one can find several variations of the wide residual block group (resnet layer) used in ResNeXt. We will describe here the prevailing convention (used in **ResNeXt-50**).

The wide residual block group consists of:

1. A first bottleneck convolution (1 x 1 kernel)
2. A split-branch-concatenate convolution of a cardinality N
3. A final bottleneck convolution (1 x 1 kernel)
4. An identity link between the input and the final convolution output

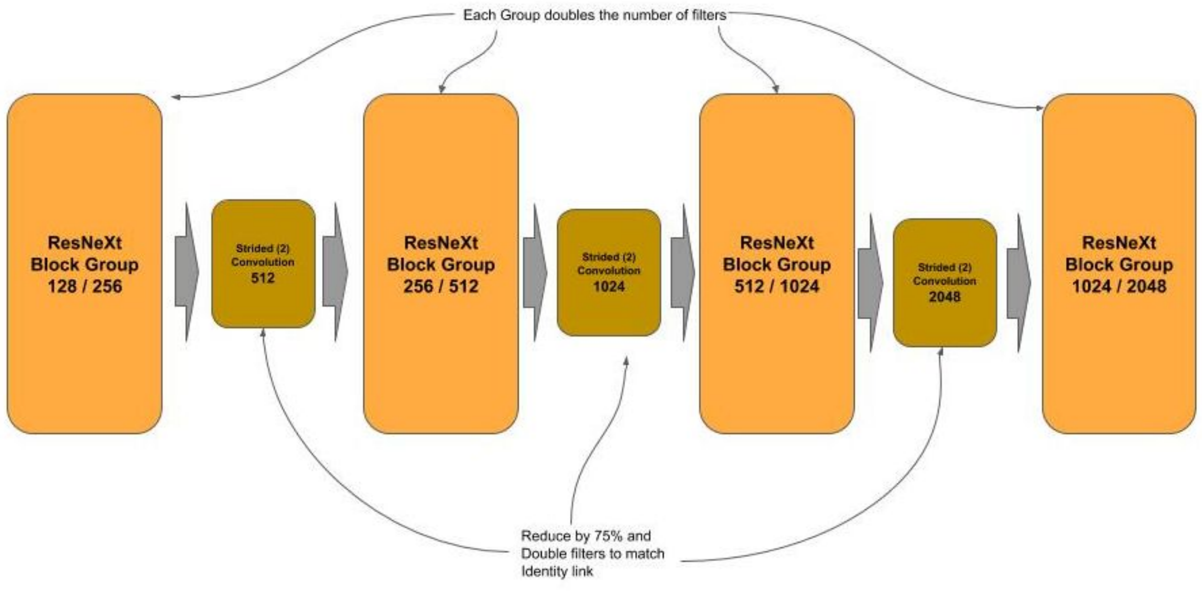


The first bottleneck convolution performs a dimensionality reduction (feature pooling), similar to its use in a residual block or inception module. After the bottleneck convolution, the feature maps are split among the parallel convolutions according to the cardinality. For example, if the number of feature maps (filters) is 128 and the cardinality is 32, then each parallel convolution will get 4 feature maps (128 / 32). The outputs from the parallel convolutions are then concatenated back into a full set of feature maps, which are then passed through a final bottleneck convolution for another dimensionality reduction (feature pooling). As in the residual block, there is an identity link between the input to and output from the resnext block, which is then matrix added.

The code below implements a **ResNeXt-50** architecture, with batch normalization and rectified linear activation unit (ReLU) added after each convolution. The architecture starts with a stem convolution block for the input, which consists of a 7 x 7 convolution which is then passed through a max pooling layer for reducing the data.

Following the stem are four groups of resnext blocks. Each group progressively doubles the number of filters outputted vs. the input. Between each block is a strided convolution, which serves two purposes:

1. It reduces the data by 75% (feature pooling)
2. It doubles the filters from the output of the previous layer, so when the identity link is made between the input of this layer and its output, the number of filters match for the matrix addition operation.


Each Group doubles the number of filters
ResNeXt Block Group 128 / 256
Strided (2) Convolution 512
ResNeXt Block Group 256 / 512
Strided (2) Convolution 1024
ResNeXt Block Group 512 / 1024
Strided (2) Convolution 2048
ResNeXt Block Group 1024 / 2048
Reduce by 75% and Double filters to match Identity link

After the final resnext group, the output is passed through a max pooling layer (and flattened) and then passed to a single dense layer for classification.

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

def _resnext_block(shortcut, filters_in, filters_out, cardinality=32):
    """ Construct a ResNeXT block
        shortcut   : previous layer and shortcut for identity link
        filters_in : number of filters  (channels) at the input convolution
        filters_out: number of filters (channels) at the output convolution
        cardinality: width of cardinality layer
    """

    # Bottleneck layer
    x = layers.Conv2D(filters_in, kernel_size=(1, 1), strides=(1, 1),
                      padding='same')(shortcut)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Cardinality (Wide) Layer
    filters_card = filters_in // cardinality
    groups = []
    for i in range(cardinality):
        group = layers.Lambda(lambda z: z[:, :, :, i * filters_card:i *
                              filters_card + filters_card])(x)
        groups.append(layers.Conv2D(filters_card, kernel_size=(3, 3),
                                    strides=(1, 1), padding='same')(group))

    # Concatenate the outputs of the cardinality layer together
    x = layers.concatenate(groups)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Bottleneck layer
    x = layers.Conv2D(filters_out, kernel_size=(1, 1), strides=(1, 1),
                      padding='same')(x)
    x = layers.BatchNormalization()(x)

    # special case for first resnext block
    if shortcut.shape[-1] != filters_out:
        # use convolutional layer to double the input size to the block so it
        # matches the output size (so we can add them)
        shortcut = layers.Conv2D(filters_out, kernel_size=(1, 1), strides=(1, 1),
                                 padding='same')(shortcut)
        shortcut = layers.BatchNormalization()(shortcut)

    # Identity Link: Add the shortcut (input) to the output of the block
    x = layers.add([shortcut, x])
    x = layers.ReLU()(x)
    return x

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

# Stem Convolutional layer
x = layers.Conv2D(64, kernel_size=(7, 7), strides=(2, 2), padding='same')(inputs)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.MaxPool2D(pool_size=(3, 3), strides=(2, 2), padding='same')(x)

# First ResNeXt Group
for _ in range(3):
    x = _resnext_block(x, 128, 256)

# strided convolution to match the number of output filters on next block and
# reduce by 2
x = layers.Conv2D(512, kernel_size=(1, 1), strides=(2, 2), padding='same')(x)

# Second ResNeXt Group
for _ in range(4):
    x = _resnext_block(x, 256, 512)

# strided convolution to match the number of output filters on next block and
# reduce by 2
x = layers.Conv2D(1024, kernel_size=(1, 1), strides=(2, 2), padding='same')(x)

# Third ResNeXt Group
for _ in range(6):
    x = _resnext_block(x, 512, 1024)

# strided convolution to match the number of output filters on next block and
# reduce by 2
x = layers.Conv2D(2048, kernel_size=(1, 1), strides=(2, 2), padding='same')(x)

# Fourth ResNeXt Group
for _ in range(3):
    x = _resnext_block(x, 1024, 2048)

# Final Dense Outputting Layer for 1000 outputs
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(1000)(x)

model = Model(inputs, outputs)
```

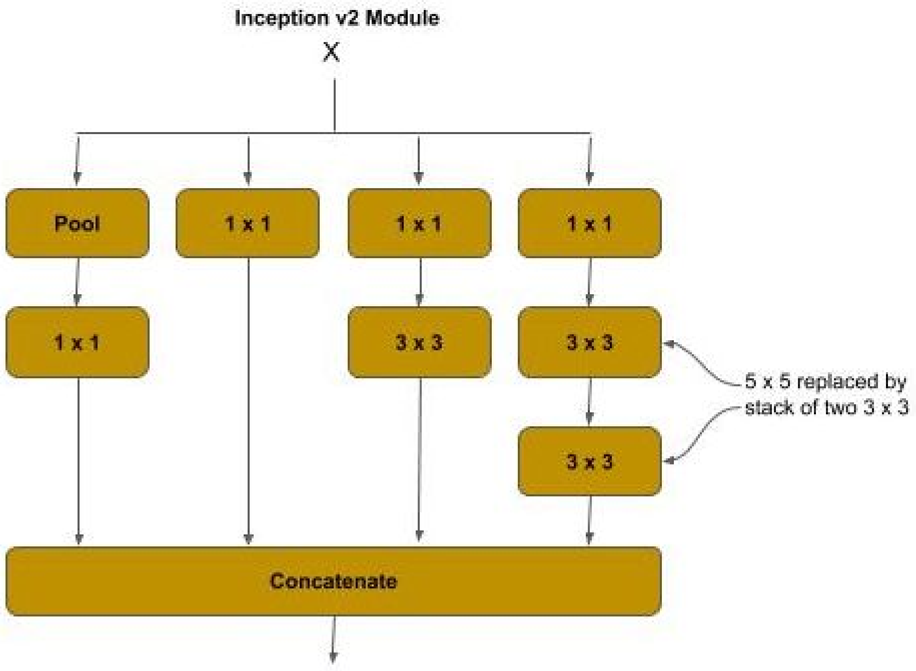
In the above code, the `Lambda()` method performs the splitting of the feature maps. The sequence `z[:, :, :, i * filters_card:i * filters_card + filters_card]` is a sliding window that is splitting the input feature maps along the fourth dimension.

Inception v2 - Factoring Convolutions

Inception v2 introduced the concept of factorization for the more expensive convolutions in an inception module to reduce computational complexity, and reduce information loss from representational bottlenecks.

The larger a filter (kernel) size is in a convolution, the more computationally expensive it is. The paper which presented the Inception v2 architecture calculated that the 5 x 5 convolution in the inception module was 2.78x more computationally expensive than a 3 x 3. In the Inception v2 module, the 5 x 5 filter is replaced by a stack of two 3 x 3 filters, which results in a reduction of computational complexity of the replaced 5 x 5 filter by 33%.

Additionally, representational bottleneck loss occurs when there is large differences in filter sizes. By replacing the 5 x 5 with two 3 x 3, all the non-bottleneck filters are now of the same size, and the overall accuracy of the Inception v2 architecture increased over Inception v1.



Below is an example of an inception v2 module:

```
# The inception branches (where x is the previous layer)
x1 = layers.MaxPooling2D((3, 3), strides=(1,1), padding='same')(x)
x1 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x1)
x2 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x3 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x3 = layers.Conv2D(96, (3, 3), strides=(1, 1), padding='same')(x3)
x4 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
x4 = layers.Conv2D(48, (3, 3), strides=(1, 1), padding='same')(x4)
x4 = layers.Conv2D(48, (3, 3), strides=(1, 1), padding='same')(x4)

# concatenate the filters
x = layers.concatenate([x1, x2, x3, x4])
```

A `summary()` for these layers shows 169K parameters to train, when compared to 198K for the inception v1 module.

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
===============
input_163 (InputLayer)          (None, 28, 28, 256)  0
__________________________________________________________________________________
_______________
conv2d_13148 (Conv2D)           (None, 28, 28, 64)   16448       input_163[0][0]
__________________________________________________________________________________
_______________
max_pooling2d_163 (MaxPooling2D (None, 28, 28, 256)  0           input_163[0][0]
__________________________________________________________________________________
_______________
conv2d_13146 (Conv2D)           (None, 28, 28, 64)   16448       input_163[0][0]
__________________________________________________________________________________
_______________
conv2d_13149 (Conv2D)           (None, 28, 28, 48)   27696       conv2d_13148[0][0]
__________________________________________________________________________________
_______________
conv2d_13144 (Conv2D)           (None, 28, 28, 64)   16448
max_pooling2d_163[0][0]
__________________________________________________________________________________
_______________
conv2d_13145 (Conv2D)           (None, 28, 28, 64)   16448       input_163[0][0]
__________________________________________________________________________________
_______________
conv2d_13147 (Conv2D)           (None, 28, 28, 96)   55392       conv2d_13146[0][0]
__________________________________________________________________________________
_______________
conv2d_13150 (Conv2D)           (None, 28, 28, 48)   20784       conv2d_13149[0][0]
```

```
_________________________________________________________________________________
_______________
concatenate_431 (Concatenate)   (None, 28, 28, 272)  0           conv2d_13144[0][0]
                                                                 conv2d_13145[0][0]
                                                                 conv2d_13147[0][0]
                                                                 conv2d_13150[0][0]
=================================================================================
===============
Total params: 169,664
Trainable params: 169,664
```

Inception v3 - Stem and Auxiliary Classifier Improvements

The **Inception v3** architecture primarily made updates to improve the auxiliary classifiers, which is outside the scope of this section; and additionally, the 7 x 7 convolution in the stem convolution group was factorized and replaced by a stack of three 3 x 3 convolutions, where:

1. The first 3 x 3 is a strided convolution (strides=2, 2) which performs a feature map reduction.
2. The second 3 x 3 is a regular convolution.
3. The third 3 x 3 doubles the number of filters.

Below is example code for implementing the Inception v3 stem group:

```
# Inception v3 stem, 7 x 7 is replaced by a stack of 3 x 3 convolutions.
x = layers.Conv2D(32, (3, 3), strides=(2, 2), padding='same')(input)
x = layers.Conv2D(32, (3, 3), strides=(1, 1), padding='same')(x)
x = layers.Conv2D(64, (3, 3), strides=(1, 1), padding='same')(x)

# max pooling layer
x = layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

# bottleneck convolution
x = layers.Conv2D(80, (1, 1), strides=(1, 1), padding='same')(x)

# next convolution layer
x = layers.Conv2D(192, (3, 3), strides=(1, 1), padding='same')(x)

# strided convolution - feature map reduction
x = layers.Conv2D(256, (3, 3), strides=(2, 2), padding='same')(x)
```

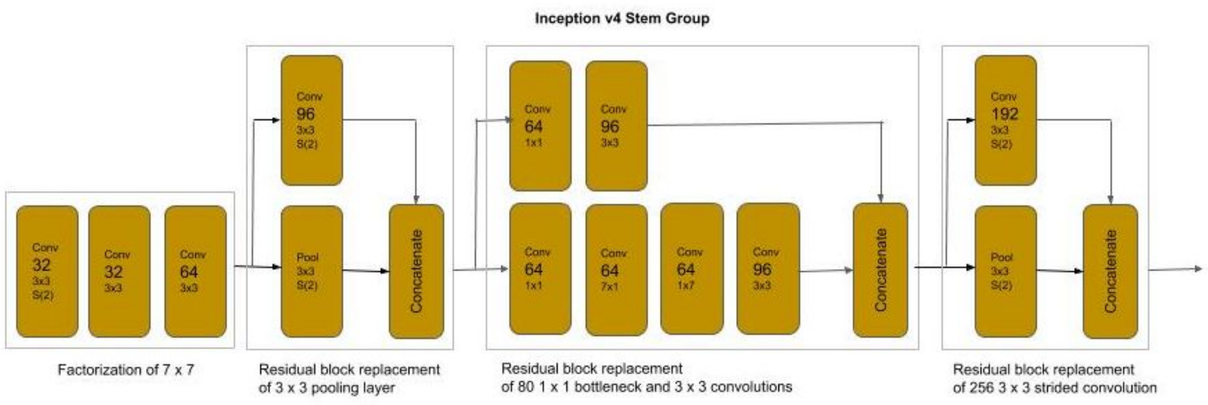
A `summary()` for the stem group shows 614K parameters to train with input (229, 229, 3):

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_2 (InputLayer)         (None, 229, 229, 3)       0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 115, 115, 32)      896       
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 115, 115, 32)      9248      
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 115, 115, 64)      18496     
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 58, 58, 64)        0         
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 58, 58, 80)        5200      
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 58, 58, 192)       138432    
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 29, 29, 256)       442624    
=================================================================
Total params: 614,896
Trainable params: 614,896
```

Inception v4 - Stem Improvements

The **Inception v4** architecture made additional improvements to the stem convolution group to reduce computational complexity by introducing reduction blocks (which appear to be a variation of a residual block). In a reduction block, the input is branched and then merged with a concatenation operation.

1. The max pooling layer is replaced with a reduction block.
   a. A max pooling layer
   b. A 3 x 3 strided convolution
2. The bottleneck and second convolution layer are replaced with a variation of a reduction block.
   a. The bottleneck convolution is reduced from size (from 80 to 64) and stacked with a 3 x 3 convolution.
   b. The second convolution layer is replaced with a stack of 1 x 1 bottleneck, a nx1 and 1xn pair and a 3 x 3 convolution.
3. The final strided convolution is replaced with a reduction block.
   a. A max pooling layer
   b. A 3 x 3 strided convolution



Below is example code for implementing the Inception v4 stem group:

```
# Inception v4 stem, 7 x 7 is replaced by a stack of 3 x 3 convolutions.
x = layers.Conv2D(32, (3, 3), strides=(2, 2), padding='same')(inputs)
x = layers.Conv2D(32, (3, 3), strides=(1, 1), padding='same')(x)
x = layers.Conv2D(64, (3, 3), strides=(1, 1), padding='same')(x)

# max pooling is replaced by reduction block of max pooling and strided convolution
bpool = layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)
b3x3  = layers.Conv2D(96, (3, 3), strides=(2, 2), padding='same')(x)
x = layers.concatenate([bpool, b3x3])

# bottleneck and 3x3 convolution replaced by reduction block
bnxn = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
bnxn = layers.Conv2D(64, (7, 1), strides=(1, 1), padding='same')(bnxn)
bnxn = layers.Conv2D(64, (1, 7), strides=(1, 1), padding='same')(bnxn)
bnxn = layers.Conv2D(96, (3, 3), strides=(1, 1), padding='same')(bnxn)
b3x3 = layers.Conv2D(64, (1, 1), strides=(1, 1), padding='same')(x)
bnxn = layers.Conv2D(96, (3, 3), strides=(1, 1), padding='same')(b3x3)
x = layers.concatenate([bnxn, b3x3])

# 3x3 strided convolution replaced by reduction block
bpool = layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)
b3x3  = layers.Conv2D(192, (3, 3), strides=(2, 2), padding='same')(x)
x = layers.concatenate([bpool, b3x3])
```

A `summary()` for the stem group shows 426K parameters to train with input (229, 229, 3) vs. the 614K for the Inception v3 stem group:

```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
===============
input_8 (InputLayer)           (None, 229, 229, 3)  0

__________________________________________________________________________________
_______________
conv2d_45 (Conv2D)             (None, 115, 115, 32) 896         input_8[0][0]

__________________________________________________________________________________
_______________
conv2d_46 (Conv2D)             (None, 115, 115, 32) 9248        conv2d_45[0][0]

__________________________________________________________________________________
_______________
conv2d_47 (Conv2D)             (None, 115, 115, 64) 18496       conv2d_46[0][0]

__________________________________________________________________________________
_______________
max_pooling2d_8 (MaxPooling2D) (None, 58, 58, 64)   0           conv2d_47[0][0]

__________________________________________________________________________________
_______________
conv2d_48 (Conv2D)             (None, 58, 58, 96)   55392       conv2d_47[0][0]

__________________________________________________________________________________
_______________
concatenate_8 (Concatenate)    (None, 58, 58, 160)  0
max_pooling2d_8[0][0]
                                                                conv2d_48[0][0]

__________________________________________________________________________________
_______________
conv2d_53 (Conv2D)             (None, 58, 58, 64)   10304
concatenate_8[0][0]

__________________________________________________________________________________
_______________
conv2d_54 (Conv2D)             (None, 58, 58, 96)   55392       conv2d_53[0][0]

__________________________________________________________________________________
_______________
concatenate_9 (Concatenate)    (None, 58, 58, 160)  0           conv2d_54[0][0]
                                                                conv2d_53[0][0]

__________________________________________________________________________________
_______________
max_pooling2d_9 (MaxPooling2D) (None, 29, 29, 160)  0
concatenate_9[0][0]

__________________________________________________________________________________
_______________
conv2d_55 (Conv2D)             (None, 29, 29, 192)  276672
concatenate_9[0][0]

__________________________________________________________________________________
_______________
concatenate_10 (Concatenate)   (None, 29, 29, 352)  0
max_pooling2d_9[0][0]
                                                                conv2d_55[0][0]
==================================================================================
===============
Total params: 426,400
Trainable params: 426,400
```

Next

In the next part, we will cover the principal and design patterns of today’s most state-of-the-art convolutional neural networks.

Part 4 - Advanced Computer Vision Models - DenseNet, Xception, MobileNet

This part covers a variety of more advanced topics to modeling in computer vision.

Pre-Stems

As we have shown, the conventional model designs for computer vision start with a stem group. The stem is the input and first layers of the model. Following the stem are the blocks, such as residual, reduction, wide convolution, inception modules, etc, and finally the classifier.

These designs have been optimized for specific feature and pooled map sizes. This optimization corresponds then to a specific input shape that will retain the feature and pooled map sizes that were optimized for the stem. For example, Inception uses an input shape of (299, 299, 3), while Resnet used (224, 224, 3) in their corresponding competition entries.

Generally, your input data is preprocessed into a specific shape for the target network architecture. But what if you want to reuse the image data preprocessed for one neural network on another with a different input shape, without reprocessing it.

The simplest approach would be to resize() the preprocessed image data, which could introduce the following issues:

- The amount of computational time expended in re-processing the images.
- Introduction of artifacts as a result of resizing, particularly if upsampled.

The conventional practice when reusing preprocessed image data for a stem that takes a different input shape is:

- If the input shape of the new stem is smaller, then downsample (resize()) the preprocessed images (e.g., (299, 299, 3) => (224, 224, 3)).
- If the input shape of the new stem is greater, then zero pad the image to enlarge the dimensions to the new shape (e.g., (224, 224, 3) => (230, 230 3). That is your adding an outer pad of zeros to match the shape.

Below is an example code snippet of a stem that will output feature maps with the dimensions of 112 x 112, when given an input size of (230, 230, 3):

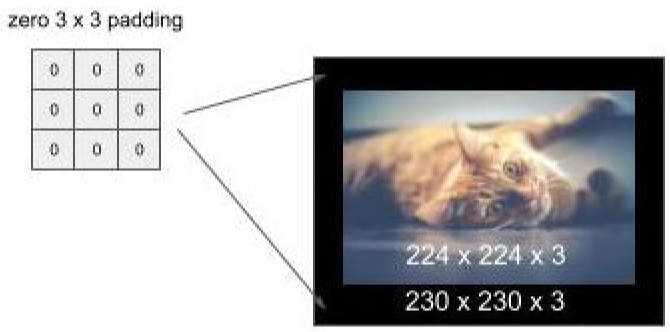
```
from keras import layers, Input

inputs = Input(shape=(230, 230, 3))

# this stem's expected shape is (230, 230, 3)
x = layers.Conv2D(64, (7, 7), strides=(2,2))(inputs)
X = layers.BatchNormalization()(x)
x = layers.ReLU()(x)

# this will output: (?, 112, 112, 64)
print(x.shape)
```

Let’s now assume that we instead use the input shape (224, 224, 3) on the same stem, as in the code example below. In this case, the stem would output a 109 x 109 feature maps instead of 112 x 112 which the architecture was not optimized for.

```
from keras import layers, Input

inputs = Input(shape=(224, 224, 3))

# this stem's expected shape is (230, 230, 3)
x = layers.Conv2D(64, (7, 7), strides=(2,2))(inputs)
X = layers.BatchNormalization()(x)
x = layers.ReLU()(x)

# this will output: (?, 109, 109, 64)
print(x.shape)
```

This will be more problematic if the model has been pretrained (weights). The weights trained for the feature maps won’t match now.

We fix this problem by adding padding around the input shape to match the input shape that the neural network architecture was optimized and/or trained for. In our example, we need to extend the 224 x 224 to 230 x 230. We can do this by adding a 3 x 3 pad (of zero values) around the image, as shown below:



The code below demonstrates using the `ZeroPadding2D()` method for adding a pad, where the parameter (3, 3) specifies the height and width of the pad to place around the image.

```
from keras import layers, Input

# not the input shape expected by the stem (which is (230, 230, 3)
inputs = Input(shape=(224, 224, 3))

# pre-stem
inputs = layers.ZeroPadding2D((3, 3))(inputs)
# this will output: (230, 230, 3)
print(inputs.shape)

# this stem's expected shape is (230, 230, 3)
x = layers.Conv2D(64, (7, 7), strides=(2,2))(inputs)
X = layers.BatchNormalization()(x)
x = layers.ReLU()(x)

# this will output: (?, 112, 112, 64)
print(x.shape)
```

Representational Equivalence and Factorization

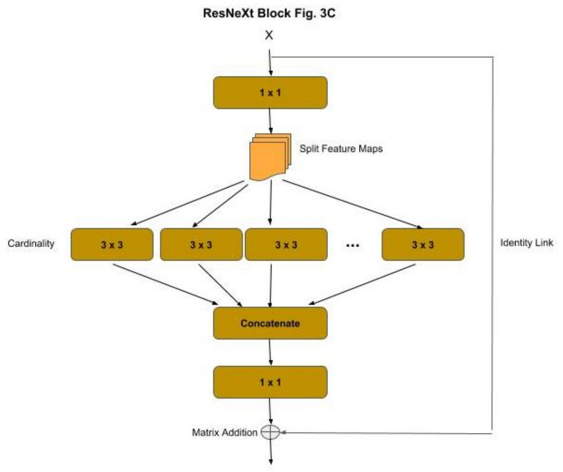
In the previous part on ResNeXt and Inception we discussed factorization. Factorization is decomposing a first block structure into smaller components which are representationally equivalent. The term representational equivalence means that the replaced design will maintain the same accuracy during training.

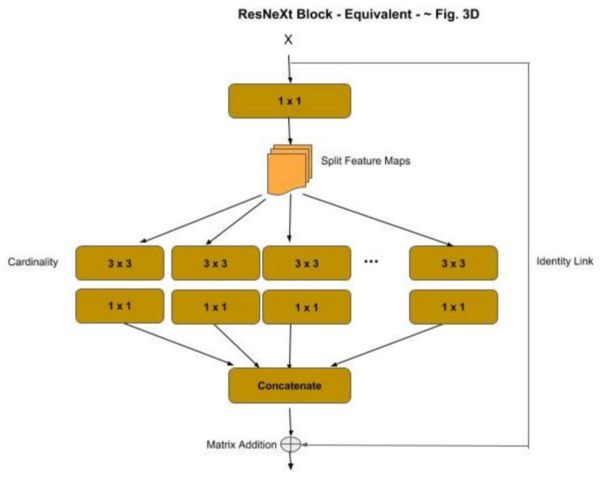
The goal of the factorization is to replace one design with another that is representationally equivalent but its computational complexity is less. The result is a smaller model to train, as measured by the number of weights, but retains the same accuracy.

The architectures for ResNeXt and for later versions of Inception relied on factorization. Typically, these factorizations are inferred through computational theory and then proved empirically.

For example, the wide convolution block replacement of the residual block in the ResNeXt was inferred, and then empirically proven, factorization. The widely published Figure 3 from the corresponding paper shows that the design in Fig. 3C was representationally equivalent to that of Fig. 3A and Fig. 3B but had less computational complexity.

While never presented in the paper, or published evidence of empirical data, using the argument in their paper it could have been hypothesized that the 1x1 bottleneck convolution after the concatenation of the 3 x 3 convolution splits, could have been factored by splitting it along with the 3 x 3 convolution. The hypothetical Fig. 3D depicts this factorization. The ResNeXt block of Fig. 3C when used in **ResNeXt-50** had a computational complexity of 25M trainable parameters. Replacing the block with the factorization of Fig. 3D in the same **ResNeXt-50** architecture has 16.7M trainable parameters, with a reduction of over 33% in computational complexity.





Below is the implementation in **Keras**:

```
# ResNext variant, bottleneck in group
import keras.layers as layers
from keras import Model


def _resnext_block(x, filters_in, filters_out, cardinality=32):
    """ Construct a ResNeXT block
        shortcut   : previous layer
        filters_in : number of filters  (channels) at the input convolution
        filters_out: number of filters (channels) at the output convolution
        cardinality: width of cardinality of the layer
    """
    shortcut = x # shortcut for the identity link

    # Bottleneck layer
    x = layers.Conv2D(filters_in, kernel_size=(1, 1), strides=(1, 1),
                      padding='same')(shortcut)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Cardinality (Wide) Layer
    filters_card = filters_in // cardinality
    groups = []
    for i in range(cardinality):
        group = layers.Lambda(lambda z: z[:, :, :, i * filters_card:i *
                              filters_card + filters_card])(x)
        group = layers.Conv2D(filters_card, kernel_size=(3, 3), strides=(1, 1),
                              padding='same')(group)
        group = layers.Conv2D(filters_card * 2, kernel_size=(1, 1), strides=(1, 1),
                               padding='same')(group)
        group = layers.BatchNormalization()(group)
        groups.append(group)

    # Concatenate the outputs of the cardinality layer together
    x = layers.concatenate(groups)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # special case for first resnext block
    if shortcut.shape[-1] != filters_out:
        # use convolutional layer to double the input size to the block so it
        # matches the output size (so we can add them)
        shortcut = layers.Conv2D(filters_out, kernel_size=(1, 1), strides=(1, 1),
                                 padding='same')(shortcut)
        shortcut = layers.BatchNormalization()(shortcut)

    # Identity Link: Add the shortcut (input) to the output of the block
    x = layers.add([shortcut, x])
    x = layers.ReLU()(x)
    return x

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

# First Convolutional layer
x = layers.Conv2D(64, kernel_size=(7, 7), strides=(2, 2), padding='same')(inputs)
```

```
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)
x = layers.MaxPool2D(pool_size=(3, 3), strides=(2, 2), padding='same')(x)

for _ in range(3):
    x = _resnext_block(x, 128, 256)

# strided convolution to match the number of output filters on next block and
# reduce by 2
x = layers.Conv2D(512, kernel_size=(1, 1), strides=(2, 2), padding='same')(x)

for _ in range(4):
    x = _resnext_block(x, 256, 512)

# strided convolution to match the number of output filters on next block and
# reduce by 2
x = layers.Conv2D(1024, kernel_size=(1, 1), strides=(2, 2), padding='same')(x)

for _ in range(6):
    x = _resnext_block(x, 512, 1024)

# strided convolution to match the number of output filters on next block and
# reduce by 2
x = layers.Conv2D(2048, kernel_size=(1, 1), strides=(2, 2), padding='same')(x)

for _ in range(3):
    x = _resnext_block(x, 1024, 2048)

# Final Dense Outputting Layer for 1000 outputs
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(1000)(x)

model = Model(inputs, outputs)
model.summary()
```

Below is the ending of the `summary()` output:

```
batch_normalization_1162 (Batch (None, 7, 7, 2048)   8192
concatenate_300[0][0]
__________________________________________________________________________________
_______________
re_lu_391 (ReLU)                (None, 7, 7, 2048)   0
batch_normalization_1162[0][0]
__________________________________________________________________________________
_______________
add_70 (Add)                    (None, 7, 7, 2048)   0           re_lu_389[0][0]
```

```
                                                                   re_lu_391[0][0]
__________________________________________________________________________________
____________
re_lu_392 (ReLU)                 (None, 7, 7, 2048)   0            add_70[0][0]
__________________________________________________________________________________
____________
global_average_pooling2d_4 (Glo (None, 2048)         0            re_lu_392[0][0]
__________________________________________________________________________________
____________
dense_4 (Dense)                  (None, 1000)         2049000
global_average_pooling2d_4[0][0]
==================================================================================
============
Total params: 16,777,960
Trainable params: 16,701,800
```

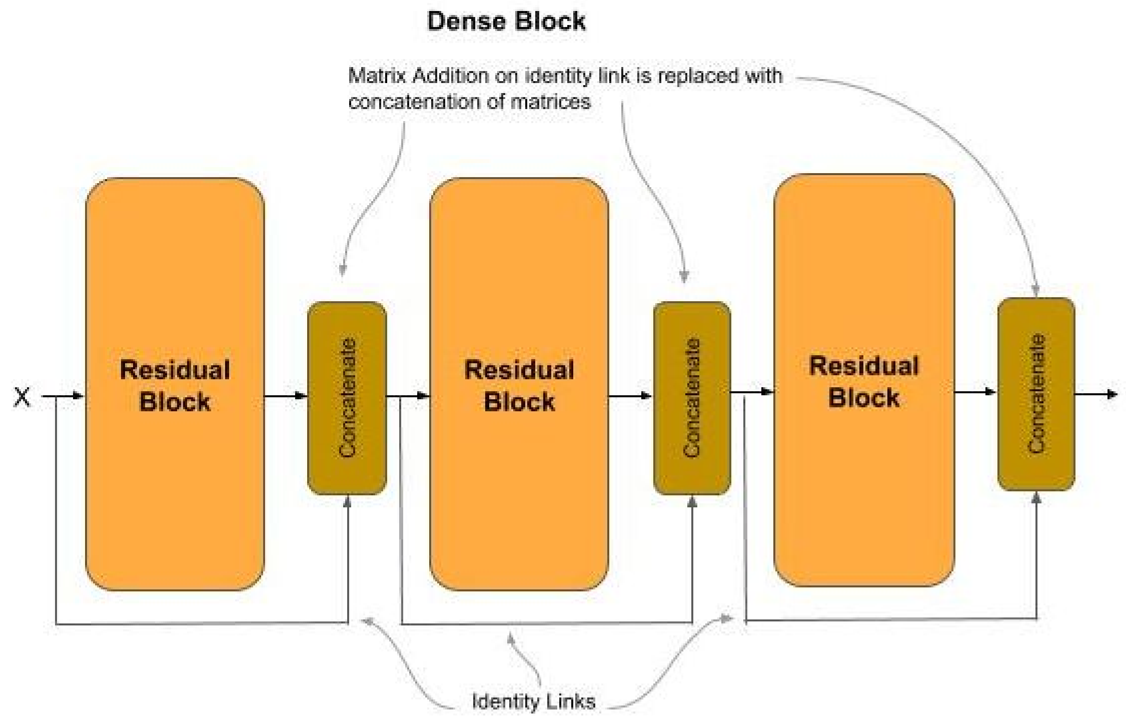
DenseNet

DenseNet introduced the concept of a Densely Connected Convolutional Network. The corresponding paper won CVPR 2017 (Best Paper Award). It is based on the principal, that the output of each residual block layer is connected to the input of every subsequent residual block layer. This extends the concept of identity links.

Prior to DenseNet, an identity link between the input and output of a residual block was combined by matrix addition. In a dense block, the input to the residual block is concatenated to the output of the residual block. This change introduced the concept of feature (map) reuse.

As an example for comparison, let's assume the output of a layer are feature maps of size 28 x 28 x 10. After a matrix addition, the outputs continue to be 28 x 28 x 10 feature maps. The values within them are the addition of the residual block's input and output, and thus do not retain the original values (i.e., they have been merged). In the dense block, the input feature maps are concatenated (not merged) to the residual block output, and the original value of the identity link is preserved. In our example, assuming the input and output where 28 x 28 x 10, after the concatenation the output will be 28 x 28 x 20. Continuing to the next block, the output will be 28 x 28 x 30. In this way, the output of each layer is concatenated into the input of each subsequent layer (hence the reference to densely connected).

The diagram below depicts the general construction and identity linking between residual blocks in a dense block.



The replacing of the matrix addition with a concatenation has the advantages of:

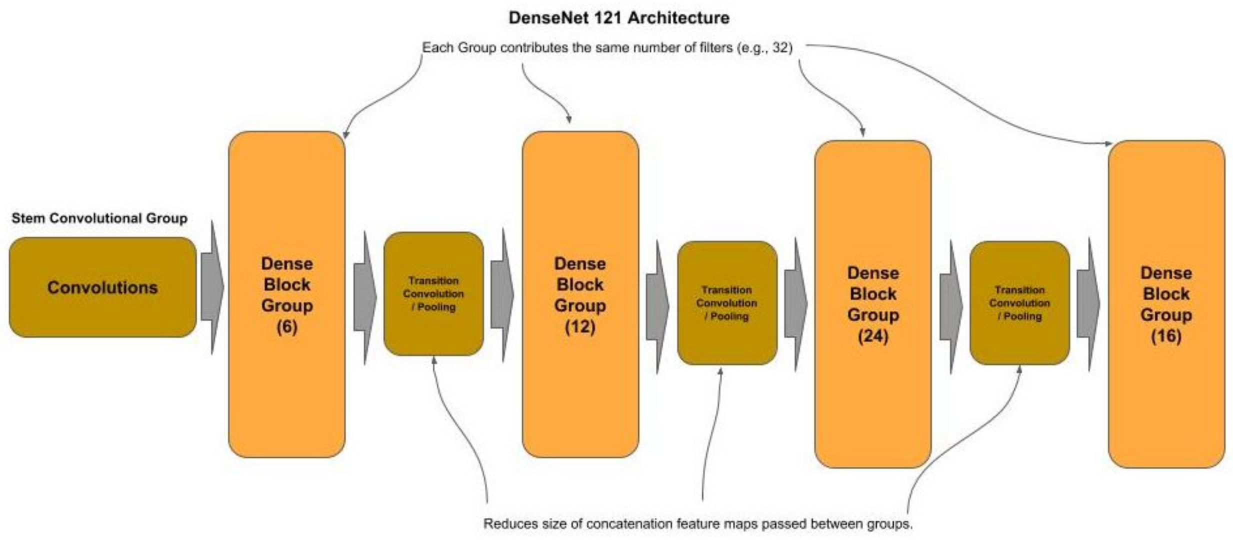
- Further alleviating the vanishing gradient problem over deeper layers.
- Further reducing the computational complexity (parameters) with narrower feature maps.

With concatenation, the distance between the output (classifier) and the feature maps is shorter which reduces the vanishing gradient problem, allowing for deeper networks for higher accuracy.

The reuse of feature maps has representational equivalence with the former operation of a matrix addition, but with more narrow layers. With the narrower layers, the overall number of parameters to train is reduced.

To further reduce computational complexity, a transition block is inserted between each dense block. The transition block is a strided convolution and feature pooling used to reduce the overall size of the concatenated feature maps (feature reuse) as they move from one dense group to the next.

Below is a diagram of the DenseNet 121 architecture.


DenseNet 121 Architecture
Each Group contributes the same number of filters (e.g., 32)
Stem Convolutional Group
Convolutions
Dense Block Group (6)
Transition Convolution / Pooling
Dense Block Group (12)
Transition Convolution / Pooling
Dense Block Group (24)
Transition Convolution / Pooling
Dense Block Group (16)
Reduces size of concatenation feature maps passed between groups.

Below is a code example implementing the **DenseNet 121** architecture. The architecture consists of:

- A Stem Convolution Block consisting of a 7 x 7 kernel, which has become a conventional practice, for extracting coarse features.
- Four Dense Groups, of 6, 12, 24 and 16 dense blocks respectively.
- A transition block between each Dense Group for reducing the size of the concatenated feature maps.
- A Classifier Block for classification of the output.

```
# Dense Net 121
from keras import layers, Input, Model

def stem(inputs):
    """ The Stem Convolution Group
        inputs : input tensor
    """
    # First large convolution for abstract features for input 230 x 230 and output
    # 112 x 112
    x = layers.Conv2D(64, (7, 7), strides=2)(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    # Add padding so when downsampling we fit shape 56 x 56
    x = layers.ZeroPadding2D(padding=((1, 1), (1, 1)))(x)
    x = layers.MaxPooling2D((3, 3), strides=2)(x)
    return x

def dense_block(x, nblocks, nb_filters):
    """ Construct a Dense Block
        x          : input layer
        nblocks    : number of residual blocks in dense block
        nb_filters: number of filters in convolution layer in residual block
    """
    # Construct a group of residual blocks
    for _ in range(nblocks):
        x = residual_block(x, nb_filters)
    return x

def residual_block(x, nb_filters):
    """ Construct Residual Block
        x          : input layer
        nb_filters: number of filters in convolution layer in residual block
    """
    shortcut = x # remember input tensor into residual block

    # Bottleneck convolution, expand filters by 4 (DenseNet-B)
    x = layers.Conv2D(4 * nb_filters, (1, 1), strides=(1, 1))(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # 3 x 3 convolution with padding=same to preserve same shape of feature maps
    x = layers.Conv2D(nb_filters, (3, 3), strides=(1, 1), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Concatenate the input (identity) with the output of the residual block
    # Concatenation (vs. merging) provides Feature Reuse between layers
    x = layers.concatenate([shortcut, x])
    return x

def trans_block(x, reduce_by):
    """ Construct a Transition Block
        x         : input layer
        reduce_by: percentage of reduction of feature maps
    """

    # Reduce (compression) the number of feature maps (DenseNet-C)
    # shape[n] returns a class object. We use int() to cast it into the dimension
    # size
    nb_filters = int( int(x.shape[3]) * reduce_by )

    # Bottleneck convolution
    x = layers.Conv2D(nb_filters, (1, 1), strides=(1, 1))(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    # Use mean value (average) instead of max value sampling when pooling
    # reduce by 75%
    x = layers.AveragePooling2D((2, 2), strides=(2, 2))(x)
    return x

inputs = Input(shape=(230, 230, 3))

# Create the Stem Convolution Group
x = stem(inputs)

# number of residual blocks in each dense block
blocks = [6, 12, 24, 16]

# pop off the list the last dense block
last   = blocks.pop()

# amount to reduce feature maps by (compression) during transition blocks
reduce_by = 0.5

# number of filters in a convolution block within a residual block
nb_filters = 32

# Create the dense blocks and interceding transition blocks
for nblocks in blocks:
    x = dense_block(x, nblocks, nb_filters)
    x = trans_block(x, reduce_by)

# Add the last dense block w/o a following transition block
x = dense_block(x, last, nb_filters)

# Classifier
# Global Average Pooling will flatten the 7x7 feature maps into 1D feature maps
x = layers.GlobalAveragePooling2D()(x)
# Fully connected output layer (classification)
x = layers.Dense(1000)(x)

model = Model(inputs, x)
model.summary()
```

Below is the ending of the `summary()` output. Note how the number of trainable parameters in this 121 layer DenseNet architecture is half of that of a 50 layer ResNeXt architecture.

```
conv2d_120 (Conv2D)           (None, 7, 7, 32)     36896       re_lu_119[0][0]
__________________________________________________________________________________
```

```
_______________
batch_normalization_120 (BatchN (None, 7, 7, 32)     128         conv2d_120[0][0]
__________________________________________________________________________________
_______________
re_lu_120 (ReLU)                (None, 7, 7, 32)     0
batch_normalization_120[0][0]
__________________________________________________________________________________
_______________
concatenate_58 (Concatenate)    (None, 7, 7, 1024)   0
concatenate_57[0][0]
                                                                 re_lu_120[0][0]
__________________________________________________________________________________
_______________
global_average_pooling2d_1 (Glo (None, 1024)         0
concatenate_58[0][0]
__________________________________________________________________________________
_______________
dense_1 (Dense)                 (None, 1000)         1025000
global_average_pooling2d_1[0][0]
==================================================================================
===============
Total params: 7,946,408
Trainable params: 7,925,928
Non-trainable params: 20,480
```

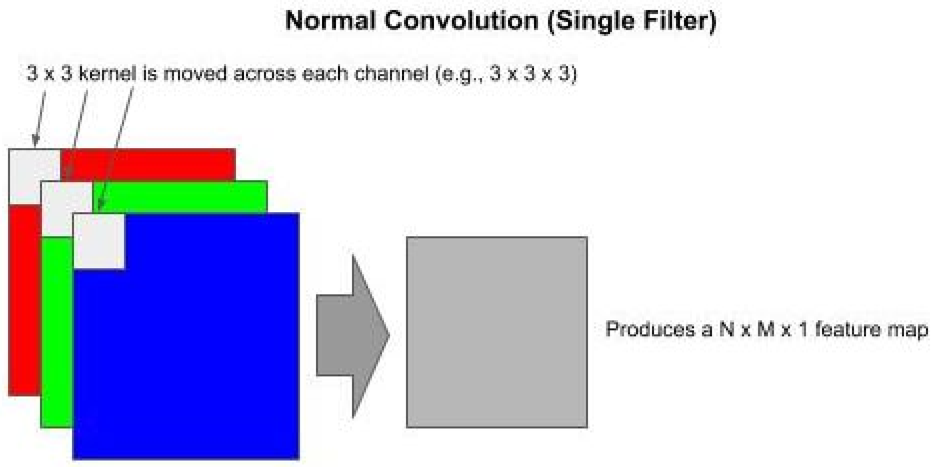
Xception

The Xception (Extreme Inception) architecture was introduced by Google in 2017 as a further improvement over the Inception v3 architecture. In this architecture the factorization of a convolution into a spatial separable convolution in an inception module is replaced with a depthwise separable convolution.

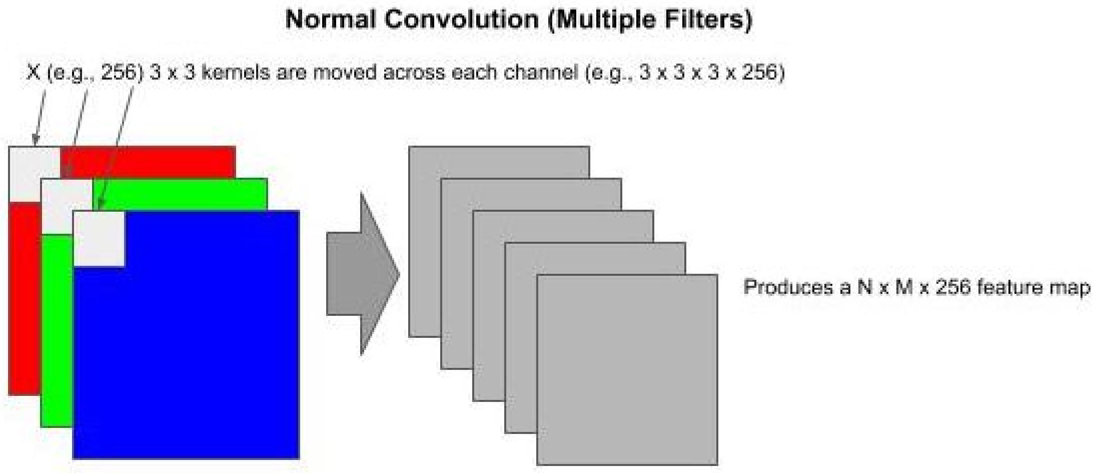
Normal Convolution

In a normal convolution, the kernel (e.g., 3 x 3) is applied across the height (H), width (W) and depth (D, channels). Each time the kernel is moved, the number of multiply operations equals the number of pixels as H x W x D.

On an RGB image (3 channels) with a 3 x 3 kernel done across all 3 channels, one has 3 x 3 x 3 = 27 multiply operations, producing a N x M x 1 (e.g., 8 x 8 x 1) feature map (per kernel), where N and M are the resulting height and width of the feature map.


Normal Convolution (Single Filter)
3 x 3 kernel is moved across each channel (e.g., 3 x 3 x 3)
Produces a N x M x 1 feature map

If we specify 256 filters for the output of the convolution, we have 256 kernels to train. In the RGB example using 256 3x3 kernels, we have 6,912 multiply operations each time the kernels move.


Normal Convolution (Multiple Filters)
X (e.g., 256) 3 x 3 kernels are moved across each channel (e.g., 3 x 3 x 3 x 256)
Produces a N x M x 256 feature map

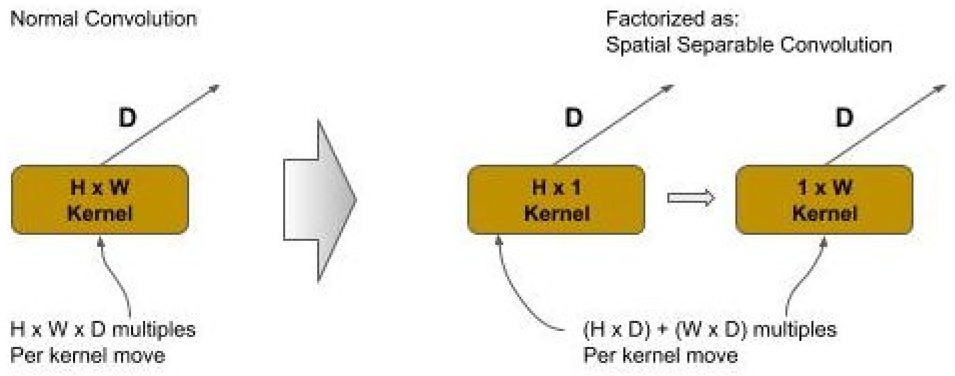
Spatial Separable Convolution

A spatial separable convolution factors a 2D kernel (e.g., 3x3) into two smaller 1D kernels. Representing the 2D kernel as H x W, then the factored two smaller 1D kernels would be H x 1 and 1 x W.

This factorization does not always maintain representational equivalence. When it does, it lowers the total number of computations by 1/3. This factorization is used in the Inception v2 module.

In the RGB example with a 3 x 3 kernel, a normal convolution would be 3 x 3 x 3 (channels) = 27 multiply operations each time the kernel is moved.

In the same RGB example with a factored 3 x 3 kernel, a spatial separable convolution would be (3 x 1 x 3) + (1 x 3 x 3) = 18 multiply operations each time the kernel is moved.


Normal Convolution
D
H x W Kernel
H x W x D multiples Per kernel move
Factorized as: Spatial Separable Convolution
D
H x 1 Kernel
D
1 x W Kernel
(H x D) + (W x D) multiples Per kernel move

In the RGB example using 256 3x3 kernels, we have 4,608 multiples each time the kernels move.

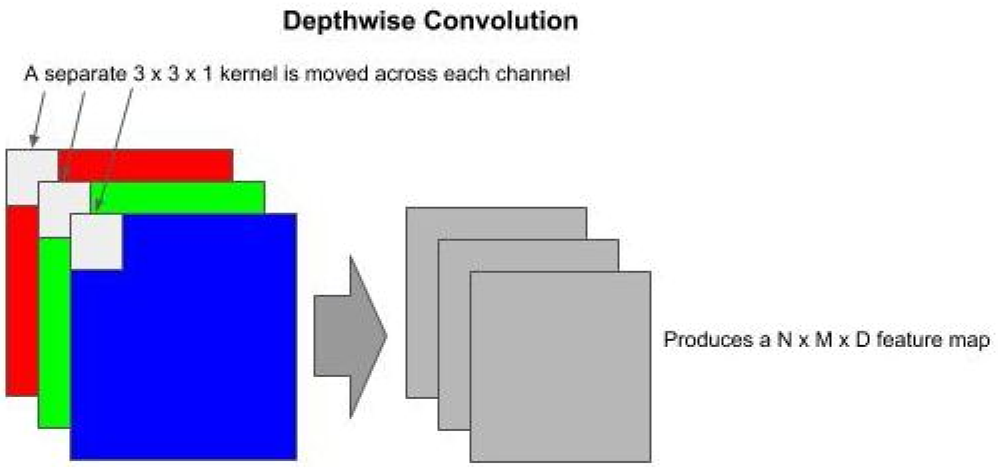
Depthwise Separable Convolution

A depthwise separable convolution is used where a convolution cannot be factored into a spatial separable convolution without representational loss. A depthwise spatial convolution factors a 2D kernel into two 2D kernels, where the first is a depthwise convolution and the second is a pointwise convolution.

Depthwise Convolution

In a depthwise convolution the kernel is split into single H x W x 1 kernels, one per channel, and where each kernel operates on a single channel instead across all channels.

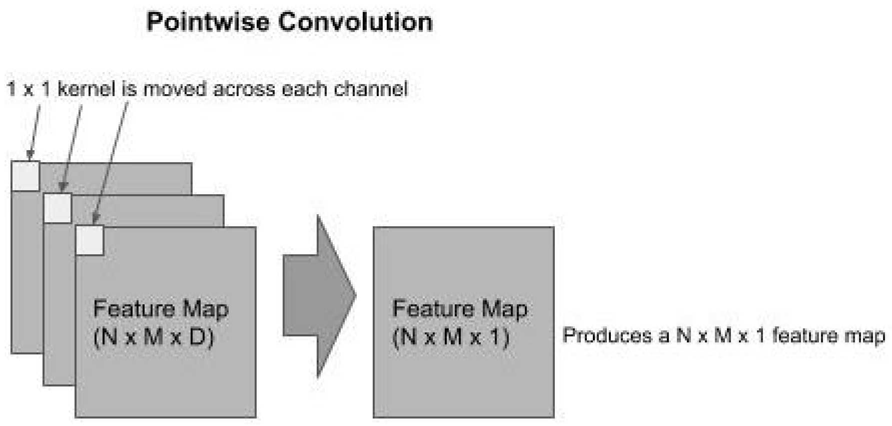
In the RGB example using a 3x3 kernel, a depthwise convolution would be three 3x3x1 kernels. While the number of multiply operations as the kernel is moved is the same as the normal convolution (e.g., 27 for 3x3 on three channels), the output is a D depth feature map instead of a 2D (depth=1) feature map.


Depthwise Convolution
A separate 3 x 3 x 1 kernel is moved across each channel
Produces a N x M x D feature map

Pointwise Convolution

The output from a depthwise convolution is then passed as the input to a pointwise convolution, which forms a depthwise separable convolution. The pointwise convolution is used to combine the outputs of the depthwise convolution and expand the number of feature maps to match the specified number of filters (feature maps).

A pointwise convolution has a 1x1xD (number of channels). It will iterate through each pixel producing a N x M x 1 feature map, which replaces the N x M x D feature map.

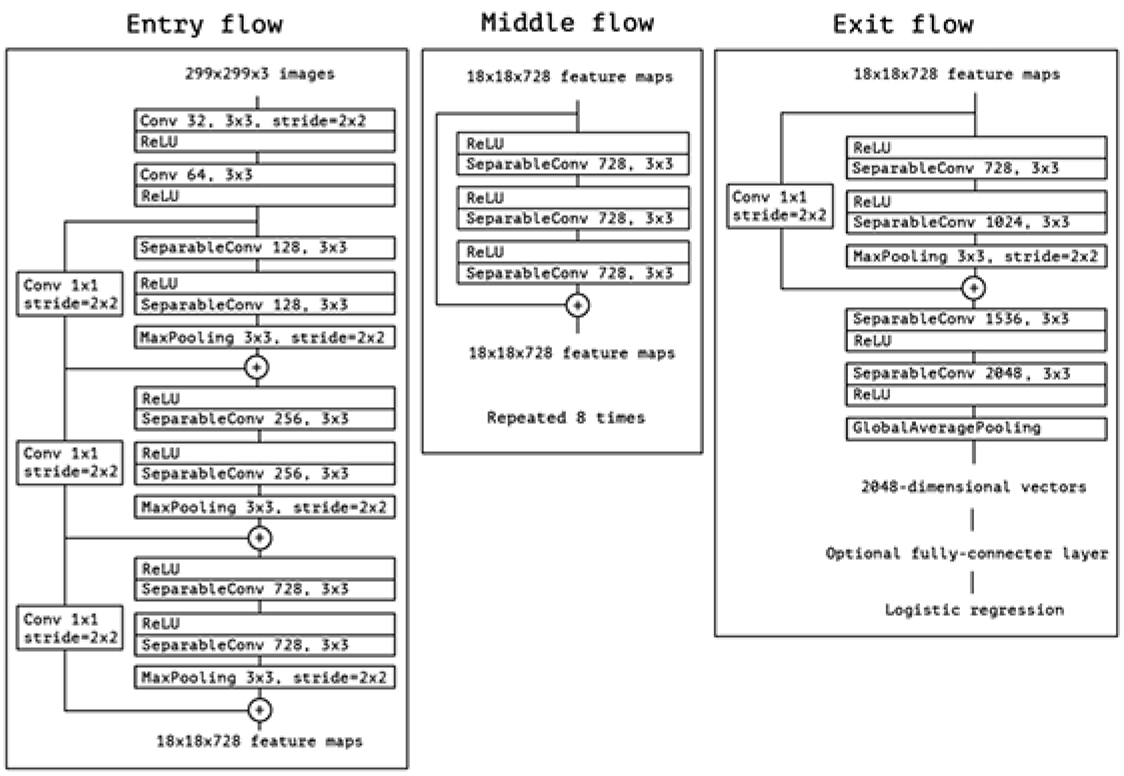

Pointwise Convolution
1 x 1 kernel is moved across each channel
Feature Map (N x M x D)
Feature Map (N x M x 1)
Produces a N x M x 1 feature map

In the pointwise convolution we use 1x1xD kernels, one for each output. As in the early example, of our output is 256 filters (feature maps), we will use 256 1x1xD kernels.

In the RGB example using a 3x3x3 kernel for the depthwise convolution, we have 27 multiply operations each time the kernel moves. This would be followed by a 1x1x3x256 (where 256 is the number of outputted filters) which is 768. The total number of multiply operations would be 795, vs. 6912 for a normal convolution and 4608 for a spatial separable convolution.

In the Xception architecture, the spatial separable convolutions in the inception module are replaced with a depthwise separable convolution, reducing computational complexity (number of multiply operations) by 83%.

Below is the architecture of Xception, as depicted by its author Francois Chollet:



The code below is an implementation of Xception. The code is partitioned into an entry flow, middle flow and exit flow section. The entry flow is further sub-partitioned into a stem and body, and the exit flow is further sub-partitioned into a body and classifier.

The depthwise separable convolution is implemented using the layers method `SeparableConv2D()`, where the parameters are otherwise the same as the method `Conv2D()`.

```
# Xception
from keras import layers, Input, Model

def entryFlow(inputs):
    """ Create the entry flow section
        inputs : input tensor to neural network
    """
    def stem(inputs):
        """ Create the stem entry into the neural network
            inputs : input tensor to neural network
        """
        # First convolution
        x = layers.Conv2D(32, (3, 3), strides=(2, 2))(inputs)
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)

        # Second convolution, double the number of filters
        x = layers.Conv2D(64, (3, 3), strides=(1, 1))(x)
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)
        return x

    # Create the stem to the neural network
    x = stem(inputs)

    # Create three residual blocks
    for nb_filters in [128, 256, 728]:
        x = residual_block_entry(x, nb_filters)

    return x

def middleFlow(x):
    """ Create the middle flow section
        x : input tensor into section
    """

    # Create 8 residual blocks
    for _ in range(8):
        x = residual_block_middle(x, 728)
    return x

def exitFlow(x):
    """ Create the exit flow section
        x : input tensor into section
    """
    def classifier(x):
        """ The output classifier
            x : input tensor
        """
        # Global Average Pooling will flatten the 10x10 feature maps into 1D
        # feature maps
        x = layers.GlobalAveragePooling2D()(x)
        # Fully connected output layer (classification)
        x = layers.Dense(1000)(x)
        return x

    shortcut = x

    # First Depthwise Separable Convolution
    x = layers.SeparableConv2D(728, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)

    # Second Depthwise Separable Convolution
    x = layers.SeparableConv2D(1024, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Create pooled feature maps, reduce size by 75%
    x = layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

    # Add strided convolution to identity link to double number of filters to
    # match output of residual block for the add operation
    shortcut = layers.Conv2D(1024, (1, 1), strides=(2, 2),
                             padding='same')(shortcut)
    shortcut = layers.BatchNormalization()(shortcut)

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

    # Third Depthwise Separable Convolution
    x = layers.SeparableConv2D(1556, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Fourth Depthwise Separable Convolution
    x = layers.SeparableConv2D(2048, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Create classifier section
    x = classifier(x)

    return x

def residual_block_entry(x, nb_filters):
    """ Create a residual block using Depthwise Separable Convolutions
        x         : input into residual block
        nb_filters: number of filters
    """
    shortcut = x

    # First Depthwise Separable Convolution
    x = layers.SeparableConv2D(nb_filters, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Second depthwise Separable Convolution
    x = layers.SeparableConv2D(nb_filters, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Create pooled feature maps, reduce size by 75%
    x = layers.MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

    # Add strided convolution to identity link to double number of filters to
    # match output of residual block for the add operation
    shortcut = layers.Conv2D(nb_filters, (1, 1), strides=(2, 2),
                             padding='same')(shortcut)
    shortcut = layers.BatchNormalization()(shortcut)

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

    return x

def residual_block_middle(x, nb_filters):
    """ Create a residual block using Depthwise Separable Convolutions
        x         : input into residual block
        nb_filters: number of filters
    """
    shortcut = x

    # First Depthwise Separable Convolution
    x = layers.SeparableConv2D(nb_filters, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Second depthwise Separable Convolution
    x = layers.SeparableConv2D(nb_filters, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Third depthwise Separable Convolution
    x = layers.SeparableConv2D(nb_filters, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    return x

inputs = Input(shape=(299, 299, 3))
```

```
# Create entry section
x = entryFlow(inputs)
# Create the middle section
x = middleFlow(x)
# Create the exit section
x = exitFlow(x)

model = Model(inputs, x)
model.summary()
```

Below is the ending output of the `summary()` method.

```
separable_conv2d_34 (SeparableC (None, 10, 10, 2048) 3202740     re_lu_34[0][0]
__________________________________________________________________________________
_______________
batch_normalization_40 (BatchNo (None, 10, 10, 2048) 8192
separable_conv2d_34[0][0]
__________________________________________________________________________________
_______________
re_lu_35 (ReLU)                 (None, 10, 10, 2048) 0
batch_normalization_40[0][0]
__________________________________________________________________________________
_______________
global_average_pooling2d_1 (Glo (None, 2048)         0           re_lu_35[0][0]
__________________________________________________________________________________
_______________
dense_1 (Dense)                 (None, 1000)         2049000
global_average_pooling2d_1[0][0]
==================================================================================
===============
Total params: 22,999,464
Trainable params: 22,944,896
Non-trainable params: 54,568
```

MobileNet

MobileNet is an architecture introduced by Google in 2017 for producing smaller networks which can fit on mobile and embedded devices, while maintaining accuracy close to their larger network counterparts. The MobileNet architecture replaces normal convolutions with depthwise separable convolutions to further reduce computational complexity.

Two additional hyperparameters were introduced for *thinning* the size of the network: *width multiplier* and *resolution multiplier*.

Width Multiplier

The first hyperparameter introduced was the width multiplier α (alpha), which thins a network uniformly at each layer.

The value of α (alpha) is between 0 and 1, and will reduce the computational complexity of a mobile net by α (alpha)**2. Typically values are 0.25 (6%), 0.50 (25%), and 0.75 (56%).

In tests results reported in the paper, a non-thinned mobilenet-224 had a 70.6% accuracy on ImageNet with 4.2 million parameters and 569 million matrix multi-add operations, while a 0.25 (width multiplier) mobilenet-224 had 50.6% accuracy with 0.5 million parameters and 41 million matrix multi-add operations.

Resolution Multiplier

The second hyperparameter introduced was the resolution multiplier ρ (rho), which thins the input shape and consequently the feature map sizes at each layer.

The value of ρ (rho) is between 0 and 1, and will reduced computational complexity of a mobile net by ρ (rho)**2.

In tests results reported in the paper, a 0.25 (resolution multiplier) mobilenet-224 had 64.4% accuracy with 4.2 million parameters and 186 million matrix multi-add operations.

Below is an implementation of a mobilenet-224:

```
# MobileNet
from keras import layers, Input, Model

def stem(inputs, alpha):
    """ Create the stem group
        inputs : input tensor
        alpha  : width multiplier
    """
    # Apply the width filter to the number of feature maps
    filters = int(32 * alpha)

    # Normal Convolutional block
    x = layers.ZeroPadding2D(padding=((0, 1), (0, 1)))(inputs)
    x = layers.Conv2D(filters, (3, 3), strides=(2, 2), padding='valid')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    # Depthwise Separable Convolution Block
    x = depthwise_block(x, 64, alpha, (1, 1))
    return x
```

```python
def classifier(x, alpha, dropout, nb_classes):
    """ Create the classifier group
        inputs    : input tensor
        alpha     : width multiplier
        dropout   : dropout percentage
        nb_classes: number of output classes
    """

    # Flatten the feature maps into 1D feature maps (?, N)
    x = layers.GlobalAveragePooling2D()(x)

    # Reshape the feature maps to (?, 1, 1, 1024)
    shape = (1, 1, int(1024 * alpha))
    x = layers.Reshape(shape)(x)
    # Perform dropout for preventing overfitting
    x = layers.Dropout(dropout)(x)

    # Use convolution for classifying (emulates a fully connected layer)
    x = layers.Conv2D(nb_classes, (1, 1), padding='same')(x)
    x = layers.Activation('softmax')(x)
    # Reshape the resulting output to 1D vector of number of classes
    x = layers.Reshape((nb_classes, ))(x)

    return x

def depthwise_block(x, nb_filters, alpha, strides):
    """ Create a Depthwise Separable Convolution block
        inputs    : input tensor
        nb_filters: number of filters
        alpha     : width multiplier
        strides   : strides
    """
    # Apply the width filter to the number of feature maps
    filters = int(nb_filters * alpha)

    # Strided convolution to match number of filters
    if strides == (2, 2):
        x = layers.ZeroPadding2D(padding=((0, 1), (0, 1)))(x)
        padding = 'valid'
    else:
        padding = 'same'

    # Depthwise Convolution
    x = layers.DepthwiseConv2D((3, 3), strides, padding=padding)(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    # Pointwise Convolution
    x = layers.Conv2D(filters, (1, 1), strides=(1, 1), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    return x


alpha      = 1    # width multiplier
dropout    = 0.5  # dropout percentage
nb_classes = 1000 # number of classes

inputs = Input(shape=(224, 224, 3))

# Create the stem group
x = stem(inputs, alpha)

# First Depthwise Separable Convolution Group
# Strided convolution - feature map size reduction
x = depthwise_block(x, 128, alpha, strides=(2, 2))
x = depthwise_block(x, 128, alpha, strides=(1, 1))

# Second Depthwise Separable Convolution Group
# Strided convolution - feature map size reduction
x = depthwise_block(x, 256, alpha, strides=(2, 2))
x = depthwise_block(x, 256, alpha, strides=(1, 1))

# Third Depthwise Separable Convolution Group
# Strided convolution - feature map size reduction
x = depthwise_block(x, 512, alpha, strides=(2, 2))
for _ in range(5):
    x = depthwise_block(x, 512, alpha, strides=(1, 1))

# Fourth Depthwise Separable Convolution Group
# Strided convolution - feature map size reduction
x = depthwise_block(x, 1024, alpha, strides=(2, 2))
x = depthwise_block(x, 1024, alpha, strides=(1, 1))

x = classifier(x, alpha, dropout, nb_classes)

print(x)

model = Model(inputs, x)
model.summary()
```

Below is the output ending of the method `summary()`:

```
global_average_pooling2d_2 (  (None, 1024)              0
_________________________________________________________________
reshape_1 (Reshape)          (None, 1, 1, 1024)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1, 1, 1024)        0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 1, 1, 1000)        1025000
_________________________________________________________________
activation_1 (Activation)    (None, 1, 1, 1000)        0
_________________________________________________________________
reshape_2 (Reshape)          (None, 1000)              0
=================================================================
Total params: 4,264,808
Trainable params: 4,242,920
Non-trainable params: 21,888
```