Chapter-3

Chapter–3

Introduction to Convolution Neural Network

It is assumed that reader knows the concept of Neural Network.

When it comes to Machine Learning, Artificial Neural Networks perform really well. Artificial Neural Networks are used in various classification task like image, audio, words. Different types of Neural Networks are used for different purposes, for example for predicting the sequence of words we use Recurrent Neural Networks more precisely an LSTM, similarly for image classification we use Convolution Neural Network. In this blog, we are going to build basic building block for CNN.
Before diving into the Convolution Neural Network, let us first revisit some concepts of Neural Network. In a regular Neural Network there are three types of layers:

1. **Input Layers:** It's the layer in which we give input to our model. The number of neurons in this layer is equal to total number of features in our data (number of pixels incase of an image).
2. **Hidden Layer:** The input from Input layer is then feed into the hidden layer. There can be many hidden layers depending upon our model and data size. Each hidden layers can have different numbers of neurons which are generally greater than the number of features. The output from each layer is computed by matrix multiplication of output of the previous layer with learnable weights of that layer and then by addition of learnable biases followed by activation function which makes the network nonlinear.
3. **Output Layer:** The output from the hidden layer is then fed into a logistic function like sigmoid or softmax which converts the output of each class into probability score of each class.

The data is then fed into the model and output from each layer is obtained this step is called feedforward, we then calculate the error using an error function, some common error functions are cross entropy, square loss error etc. After that, we backpropagate into the model by calculating the derivatives. This step is called Backpropagation which basically is used to minimize the loss.

Here's the basic python code for a neural network with random inputs and two hidden layers.

filter_none

brightness_4

```
activation = lambda x: 1.0/(1.0 + np.exp(-x)) # sigmoid function
```

```
input = np.random.randn(3, 1)

hidden_1 = activation(np.dot(W1, input) + b1)

hidden_2 = activation(np.dot(W2, hidden_1) + b2)

output = np.dot(W3, hidden_2) + b3
```

W1,W2,W3,b1,b2,b3 are learnable parameter of the model.

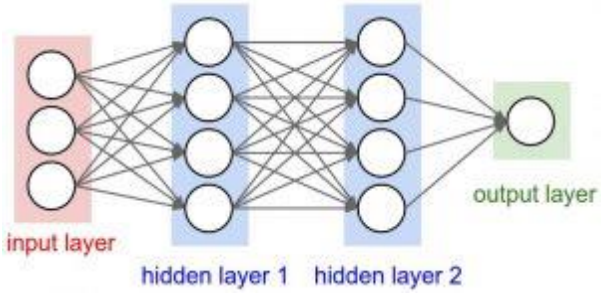
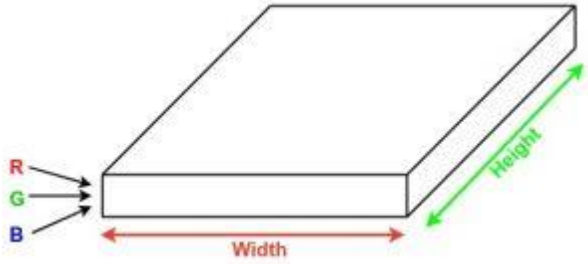


Image source: cs231n.stanford.edu

Convolution Neural Network

Convolution Neural Networks or covnets are neural networks that share their parameters. Imagine you have an image. It can be represented as a cuboid having its length, width (dimension of the image) and height (as image generally have red, green, and blue channels).



Now imagine taking a small patch of this image and running a small neural network on it, with say, k outputs and represent them vertically. Now slide that neural network across the whole image, as a result, we will get another image with different width, height, and depth. Instead of just R, G and B channels now we have more channels but lesser width and height. his operation is called Convolution. If patch size is same as that of the image it will be a regular neural network. Because of this small patch, we have fewer weights.

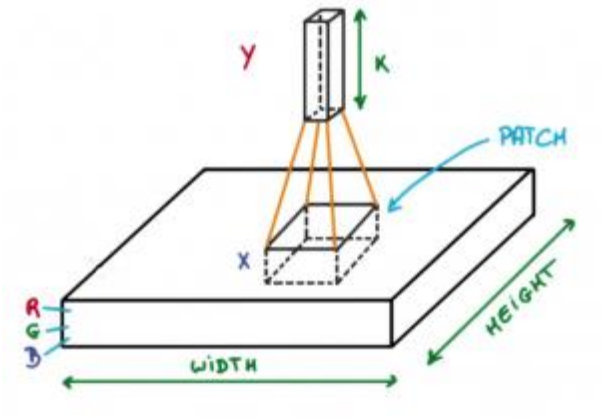


Image source: Deep Learning Udacity

Now let's talk about a bit of mathematics which is involved in the whole convolution process.

- Convolution layers consist of a set of learnable filters (patch in the above image). Every filter has small width and height and the same depth as that of input volume (3 if the input layer is image input).
- For example, if we have to run convolution on an image with dimension 34x34x3. Possible size of filters can be axax3, where 'a' can be 3, 5, 7, etc but small as compared to image dimension.
- During forward pass, we slide each filter across the whole input volume step by step where each step is called stride (which can have value 2 or 3 or even 4 for high dimensional images) and compute the dot product between the weights of filters and patch from input volume.
- As we slide our filters we'll get a 2-D output for each filter and we'll stack them together and as a result, we'll get output volume having a depth equal to the number of filters. The network will learn all the filters.

Layers used to build ConvNets

A covnets is a sequence of layers, and every layer transforms one volume to another through differentiable function.

Types of layers:

Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

1. **Input Layer:** This layer holds the raw input of image with width 32, height 32 and depth 3.
2. **Convolution Layer:** This layer computes the output volume by computing dot product between all filters and image patch. Suppose we use total 12 filters for this layer we'll get output volume of dimension 32 x 32 x 12.
3. **Activation Function Layer:** This layer will apply element wise activation function to the output of convolution layer. Some common activation functions are RELU: max(0, x), Sigmoid: 1/(1+e^-x), Tanh, Leaky RELU, etc. The volume remains unchanged hence output volume will have dimension 32 x 32 x 12.
4. **Pool Layer:** This layer is periodically inserted in the covnets and its main function is to reduce the size of volume which makes the computation fast reduces memory and also prevents from overfitting. Two common types of pooling layers are **max**

pooling and **average pooling**. If we use a max pool with 2 x 2 filters and stride 2, the resultant volume will be of dimension 16x16x12.

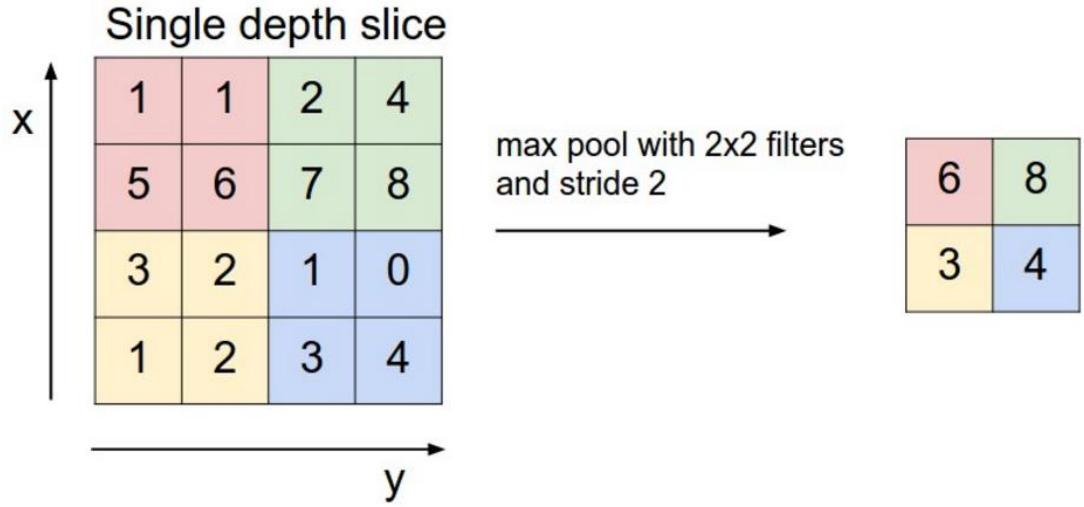
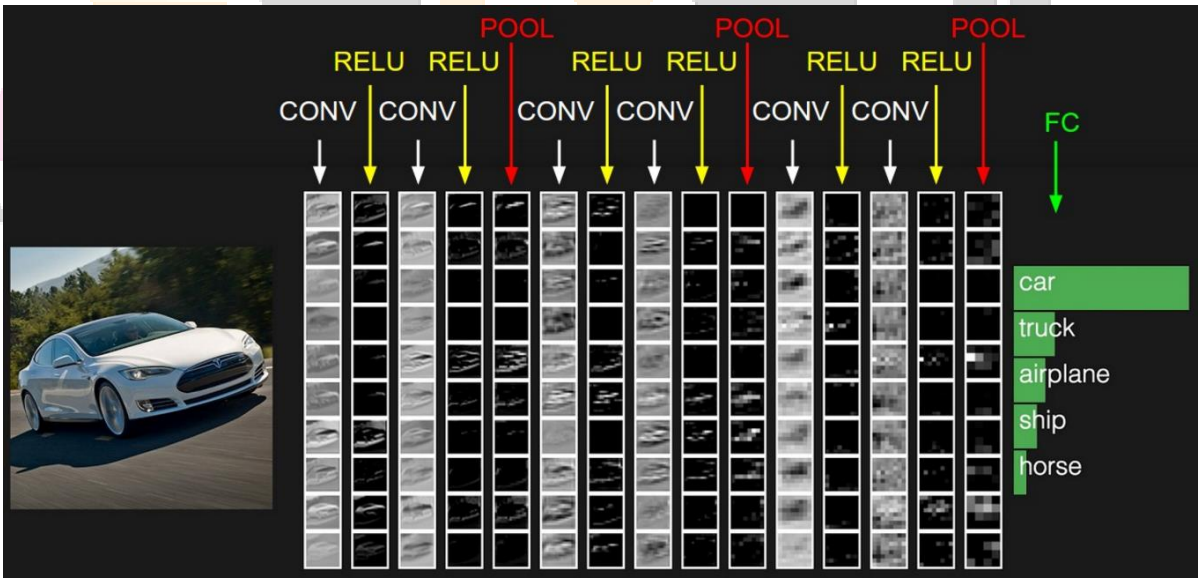


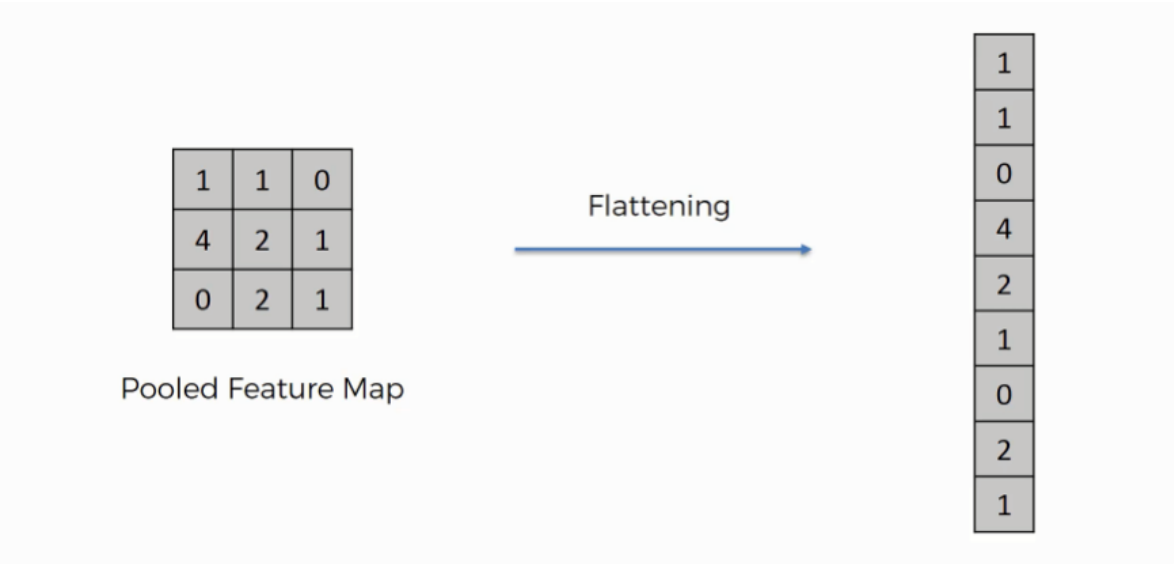
Image source: cs231n.stanford.edu

5. **Fully-Connected Layer:** This layer is regular neural network layer which takes input from the previous layer and computes the class scores and outputs the 1-D array of size equal to the number of classes.

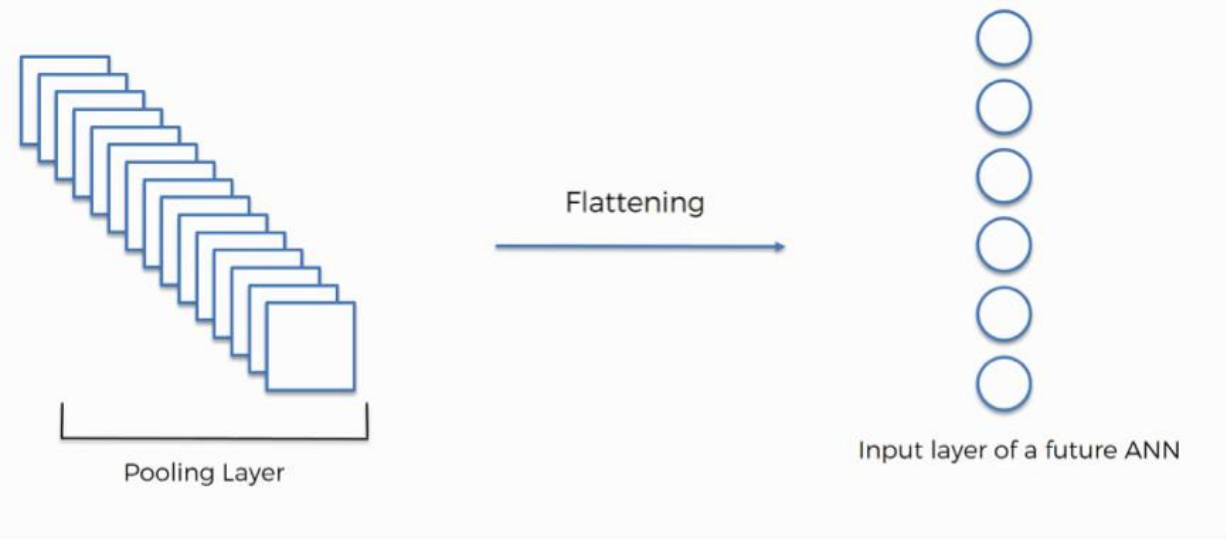


Flattening

As the name of this step implies, we are literally going to flatten our pooled feature map into a column like in the image below.



The reason we do this is that we're going to need to insert this data into an artificial neural network later on.



As you see in the image above, we have multiple pooled feature maps from the previous step.

What happens after the flattening step is that you end up with a long vector of input data that you then pass through the artificial neural network to have it processed further.

To sum up, here is what we have after we're done with each of the steps that we have covered up until now:

- Input image (starting point)
- Convolutional layer (convolution operation)
- Pooling layer (pooling)
- Input layer for the artificial neural network (flattening)

Padding and Stride

Open the notebook in Colab

In the previous example, our input had a height and width of 33 and a convolution kernel with a height and width of 22, yielding an output with a height and a width of 22. In general, assuming the input shape is nh×nwnh×nw and the convolution kernel window shape is kh×kwkh×kw, then the output shape will be
(6.3.1)

$$(nh−kh+1)×(nw−kw+1).(nh−kh+1)×(nw−kw+1).$$

Therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel window.

In several cases we might want to incorporate particular techniques—padding and strides, regarding the size of the output:

- In general, since kernels generally have width and height greater than 11, that means that after applying many successive convolutions, we will wind up with an output that is much smaller than our input. If we start with a 240×240240×240 pixel image, 1010 layers of 5×55×5 convolutions reduce the image to 200×200200×200 pixels, slicing off 30%30% of the image and with it obliterating any interesting information on the boundaries of the original image. *Padding* handles this issue.
- In some cases, we want to reduce the resolution drastically if say we find our original input resolution to be unwieldy. *Strides* can help in these instances.

Padding

As described above, one tricky issue when applying convolutional layers is that of losing pixels on the perimeter of our image. Since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the

boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to 00. In Fig.1, we pad a 3×53×5 input, increasing its size to 5×75×7. The corresponding output then increases to a 4×64×6 matrix.

Fig.1 Two-dimensional cross-correlation with padding. The shaded portions are the input and kernel array elements used by the first output element: 0×0+0×1+0×2+0×3=00×0+0×1+0×2+0×3=0.

In general, if we add a total of phph rows of padding (roughly half on top and half on bottom) and a total of pwpw columns of padding (roughly half on the left and half on the right), the output shape will be
(6.3.2)

$$(nh-kh+ph+1)\times(nw-kw+pw+1).(nh-kh+ph+1)\times(nw-kw+pw+1).$$

This means that the height and width of the output will increase by phph and pwpw respectively. In many cases, we will want to set ph=kh−1ph=kh−1 and pw=kw−1pw=kw−1 to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that khkh is even here, we will pad ph/2ph/2 rows on both sides of the height. If khkh is odd, one possibility is to pad ⌈ph/2⌉⌈ph/2⌉ rows on the top of the input and ⌊ph/2⌋⌊ph/2⌋ rows on the bottom. We will pad both sides of the width in the same way.

Convolutional neural networks commonly use convolutional kernels with odd height and width values, such as 11, 33, 55, or 77. Choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.

Moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clerical benefit. For any two-dimensional array X, when the kernels size is odd and the number of padding rows and columns on all sides are the same, producing an output with the same height and width as the input, we know that the output Y[i, j] is calculated by cross-correlation of the input and convolution kernel with the window centered on X[i, j].

In the following example, we create a two-dimensional convolutional layer with a height and width of 33 and apply 11 pixel of padding on all sides. Given an input with a height and width of 88, we find that the height and width of the output is also 88.

```
from mxnet import np, npx
from mxnet.gluon import nn
```

```
npx.set_np()

# For convenience, we define a function to calculate the convolutional layer.
# This function initializes the convolutional layer weights and performs
# corresponding dimensionality elevations and reductions on the input and
# output
def comp_conv2d(conv2d, X):
    conv2d.initialize()
    # (1, 1) indicates that the batch size and the number of channels
    # (described in later chapters) are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: batch and
    # channel
    return Y.reshape(Y.shape[2:])

# Note that here 1 row or column is padded on either side, so a total of 2
# rows or columns are added
conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
X = np.random.uniform(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

When the height and width of the convolution kernel are different, we can make the output and input have the same height and width by setting different padding numbers for height and width.

```
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on both sides of the height and width are 2 and 1,
# respectively
conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

Stride

When computing the cross-correlation, we start with the convolution window at the top-left corner of the input array, and then slide it over all locations both down and to the right. In previous examples, we default to sliding one pixel at a time. However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one pixel at a time, skipping the intermediate locations.

We refer to the number of rows and columns traversed per slide as the *stride*. So far, we have used strides of 11, both for height and width. Sometimes, we may want to use a larger stride. Fig.2 shows a two-dimensional cross-correlation operation with a stride of 33 vertically and 22 horizontally. We can see that when the second element of the first column is output, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is output. When the convolution window slides three columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

Fig.2 Cross-correlation with strides of 3 and 2 for height and width respectively. The shaded portions are the output element and the input and core array elements used in its computation: 0×0+0×1+1×2+2×3=80×0+0×1+1×2+2×3=8, 0×0+6×1+0×2+0×3=60×0+6×1+0×2+0×3=6.

In general, when the stride for the height is shsh and the stride for the width is swsw, the output shape is
(6.3.3)

$$\lfloor (nh-kh+ph+sh)/sh \rfloor \times \lfloor (nw-kw+pw+sw)/sw \rfloor . \lfloor (nh-kh+ph+sh)/sh \rfloor \times \lfloor (nw-kw+pw+sw)/sw \rfloor .$$

If we set ph=kh−1ph=kh−1 and pw=kw−1pw=kw−1, then the output shape will be simplified to ⌊(nh+sh−1)/sh⌋×⌊(nw+sw−1)/sw⌋⌊(nh+sh−1)/sh⌋×⌊(nw+sw−1)/sw⌋. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be (nh/sh)×(nw/sw)(nh/sh)×(nw/sw).
Below, we set the strides on both the height and width to 22, thus halving the input height and width.

Padding and Stride Open the notebook in Col

In the previous example, our input had a height and width of 33 and a convolution kernel with a height and width of 22, yielding an output with a height and a width of 22. In general, assuming the input shape is nh×nwnh×nw and the convolution kernel window shape is kh×kwkh×kw, then the output shape will be
(6.3.1)

$$(nh-kh+1)\times(nw-kw+1).(nh-kh+1)\times(nw-kw+1).$$

Therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel window.

In several cases we might want to incorporate particular techniques—padding and strides, regarding the size of the output:

- In general, since kernels generally have width and height greater than 11, that means that after applying many successive convolutions, we will wind up with an output that is much smaller than our input. If we start with a 240×240240×240 pixel image, 1010 layers of 5×55×5 convolutions reduce the image to 200×200200×200 pixels, slicing off 30%30% of the image and with it obliterating any interesting information on the boundaries of the original image. *Padding* handles this issue.
- In some cases, we want to reduce the resolution drastically if say we find our original input resolution to be unwieldy. *Strides* can help in these instances.

Padding

As described above, one tricky issue when applying convolutional layers is that of losing pixels on the perimeter of our image. Since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to 00. In Fig. 3.1, we pad a 3×53×5 input, increasing its size to 5×75×7. The corresponding output then increases to a 4×64×6 matrix.

Fig. 6.3.1 Two-dimensional cross-correlation with padding. The shaded portions are the input and kernel array elements used by the first output element: 0×0+0×1+0×2+0×3=00×0+0×1+0×2+0×3=0.

In general, if we add a total of phph rows of padding (roughly half on top and half on bottom) and a total of pwpw columns of padding (roughly half on the left and half on the right), the output shape will be
(6.3.2)

$$(nh-kh+ph+1)\times(nw-kw+pw+1).(nh-kh+ph+1)\times(nw-kw+pw+1).$$

This means that the height and width of the output will increase by phph and pwpw respectively. In many cases, we will want to set ph=kh−1ph=kh−1 and pw=kw−1pw=kw−1 to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that khkh is even here, we will pad ph/2ph/2 rows on both sides of the height. If khkh is odd, one possibility is to pad ⌈ph/2⌉⌈ph/2⌉ rows on the top of the input and ⌊ph/2⌋⌊ph/2⌋ rows on the bottom. We will pad both sides of the width in the same way.

Convolutional neural networks commonly use convolutional kernels with odd height and width values, such as 11, 33, 55, or 77. Choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.

Moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clerical benefit. For any two-dimensional array X, when the kernels size is odd and the number of padding rows and columns on all sides are the same, producing an output with the same height and width as the input, we know that the output Y[i, j] is calculated by cross-correlation of the input and convolution kernel with the window centered on X[i, j].

In the following example, we create a two-dimensional convolutional layer with a height and width of 33 and apply 11 pixel of padding on all sides. Given an input with a height and width of 88, we find that the height and width of the output is also 88.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# For convenience, we define a function to calculate the convolutional layer.
# This function initializes the convolutional layer weights and performs
# corresponding dimensionality elevations and reductions on the input and
# output
def comp_conv2d(conv2d, X):
    conv2d.initialize()
    # (1, 1) indicates that the batch size and the number of channels
    # (described in later chapters) are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: batch and
    # channel
    return Y.reshape(Y.shape[2:])

# Note that here 1 row or column is padded on either side, so a total of 2
# rows or columns are added
conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
X = np.random.uniform(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

When the height and width of the convolution kernel are different, we can make the output and input have the same height and width by setting different padding numbers for height and width.

```
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on both sides of the height and width are 2 and 1,
# respectively
conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

6.3.2. Stride

When computing the cross-correlation, we start with the convolution window at the top-left corner of the input array, and then slide it over all locations both down and to the right. In previous examples, we default to sliding one pixel at a time. However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one pixel at a time, skipping the intermediate locations.

We refer to the number of rows and columns traversed per slide as the *stride*. So far, we have used strides of 11, both for height and width. Sometimes, we may want to use a larger stride. Fig. 6.3.2 shows a two-dimensional cross-correlation operation with a stride of 33 vertically and 22 horizontally. We can see that when the second element of the first column is output, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is output. When the convolution window slides three columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

Fig. 6.3.2 Cross-correlation with strides of 3 and 2 for height and width respectively. The shaded portions are the output element and the input and core array elements used in its computation: $0\times0+0\times1+1\times2+2\times3=8$$0\times0+0\times1+1\times2+2\times3=8$, $0\times0+6\times1+0\times2+0\times3=6$$0\times0+6\times1+0\times2+0\times3=6$.

In general, when the stride for the height is shsh and the stride for the width is swsw, the output shape is
(6.3.3)

$$\lfloor (nh-kh+ph+sh)/sh \rfloor \times \lfloor (nw-kw+pw+sw)/sw \rfloor . \lfloor (nh-kh+ph+sh)/sh \rfloor \times \lfloor (nw-kw+pw+sw)/sw \rfloor .$$

If we set ph=kh−1ph=kh−1 and pw=kw−1pw=kw−1, then the output shape will be simplified to ⌊(nh+sh−1)/sh⌋×⌊(nw+sw−1)/sw⌋⌊(nh+sh−1)/sh⌋×⌊(nw+sw−1)/sw⌋. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be (nh/sh)×(nw/sw)(nh/sh)×(nw/sw).
Below, we set the strides on both the height and width to 22, thus halving the input height and width.

Convolution Layer

Convolution is the first layer to extract features from an input image. Convolution preserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs such as image matrix and a filter or kernel.

- An image matrix (volume) of dimension **(h x w x d)**
- A filter **(f_h x f_w x d)**
- Outputs a volume dimension **(h - f_h + 1) x (w - f_w + 1) x 1**

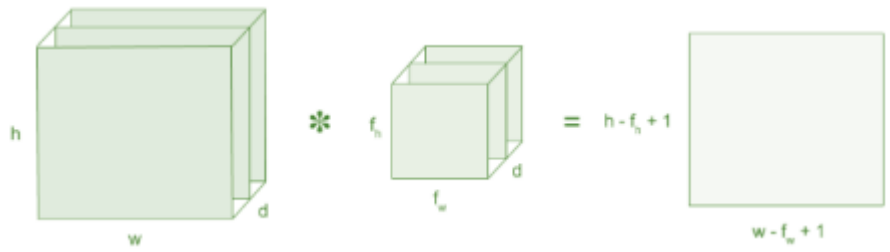


Figure 3: Image matrix multiplies kernel or filter matrix

Consider a 5 x 5 whose image pixel values are 0, 1 and filter matrix 3 x 3 as shown in below

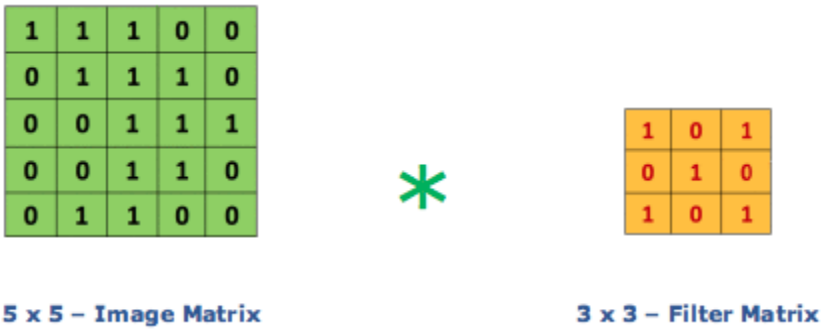


Figure 4: Image matrix multiplies kernel or filter matrix

Then the convolution of 5 x 5 image matrix multiplies with 3 x 3 filter matrix which is called **"Feature Map"** as output shown in below

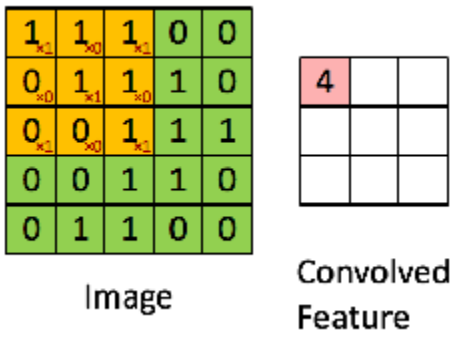


Figure 5: 3 x 3 Output matrix

Convolution of an image with different filters can perform operations such as edge detection, blur and sharpen by applying filters. The below example shows various convolution image after applying different types of filters (Kernels).

Figure 7 : Some common filters

Strides

Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on. The below figure shows convolution would work with a stride of 2.

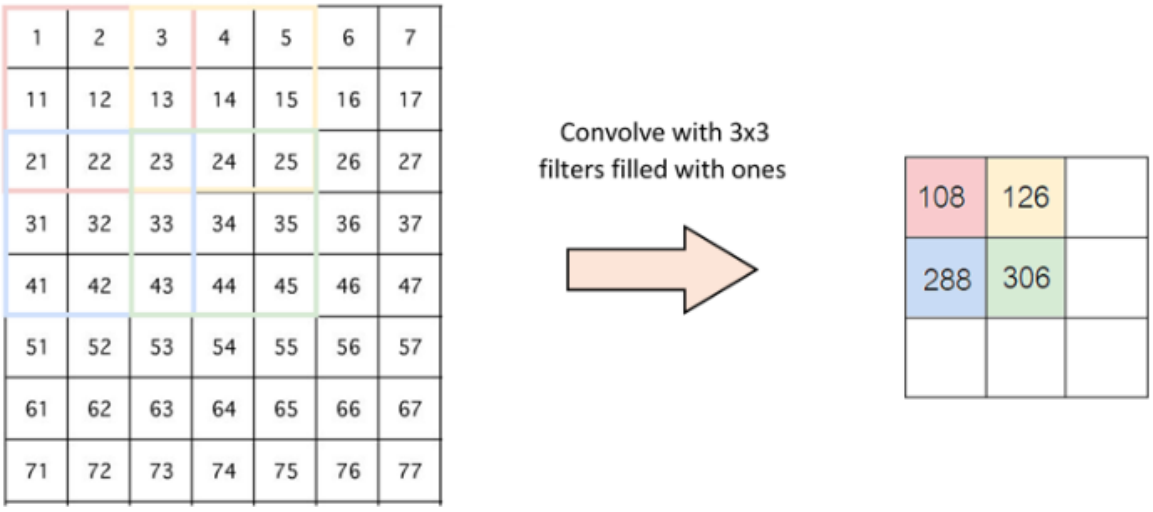


Figure 6 : Stride of 2 pixels

Padding

Sometimes filter does not fit perfectly fit the input image. We have two options:

- Pad the picture with zeros (zero-padding) so that it fits
- Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.

Non Linearity (ReLU)

ReLU stands for Rectified Linear Unit for a non-linear operation. The output is $f(x) = max(0,x)$.

Why ReLU is important : ReLU's purpose is to introduce non-linearity in our ConvNet. Since, the real world data would want our ConvNet to learn would be non-negative linear values.

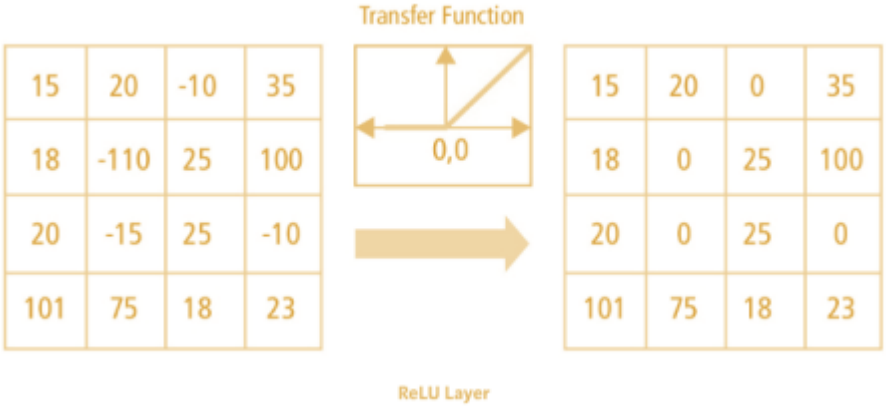


Figure 7 : ReLU operation

There are other non linear functions such as tanh or sigmoid that can also be used instead of ReLU. Most of the data scientists use ReLU since performance wise ReLU is better than the other two.

Pooling Layer

Pooling layers section would reduce the number of parameters when the images are too large. Spatial pooling also called subsampling or downsampling which reduces the dimensionality of each map but retains important information. Spatial pooling can be of different types:

- Max Pooling
- Average Pooling
- Sum Pooling

Max pooling takes the largest element from the rectified feature map. Taking the largest element could also take the average pooling. Sum of all elements in the feature map call as sum pooling.

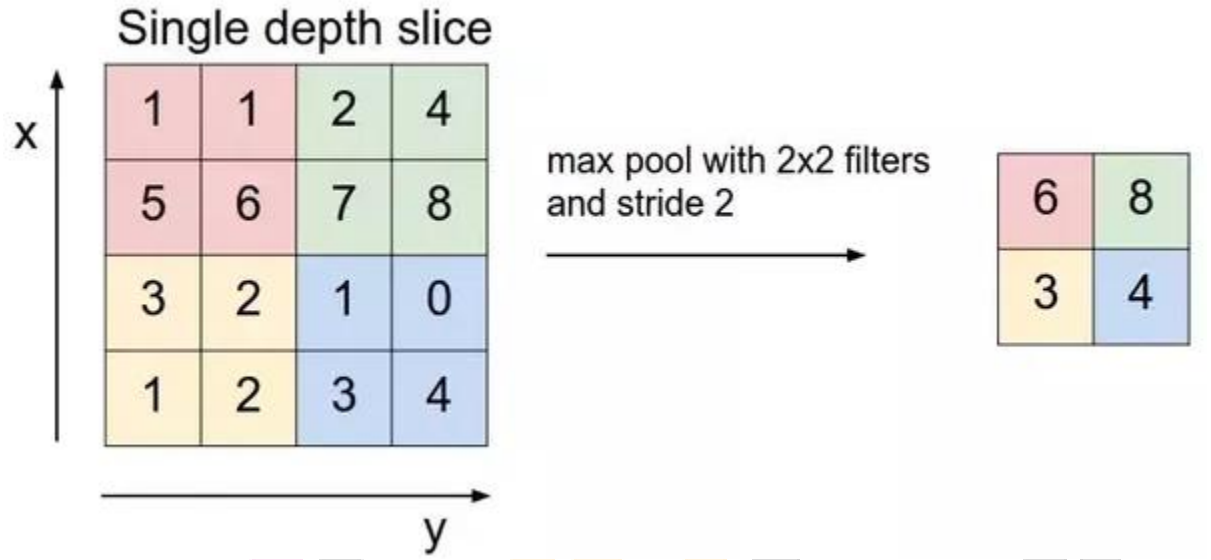


Figure 8 : Max Pooling

Fully Connected Layer

The layer we call as FC layer, we flattened our matrix into vector and feed it into a fully connected layer like a neural network.

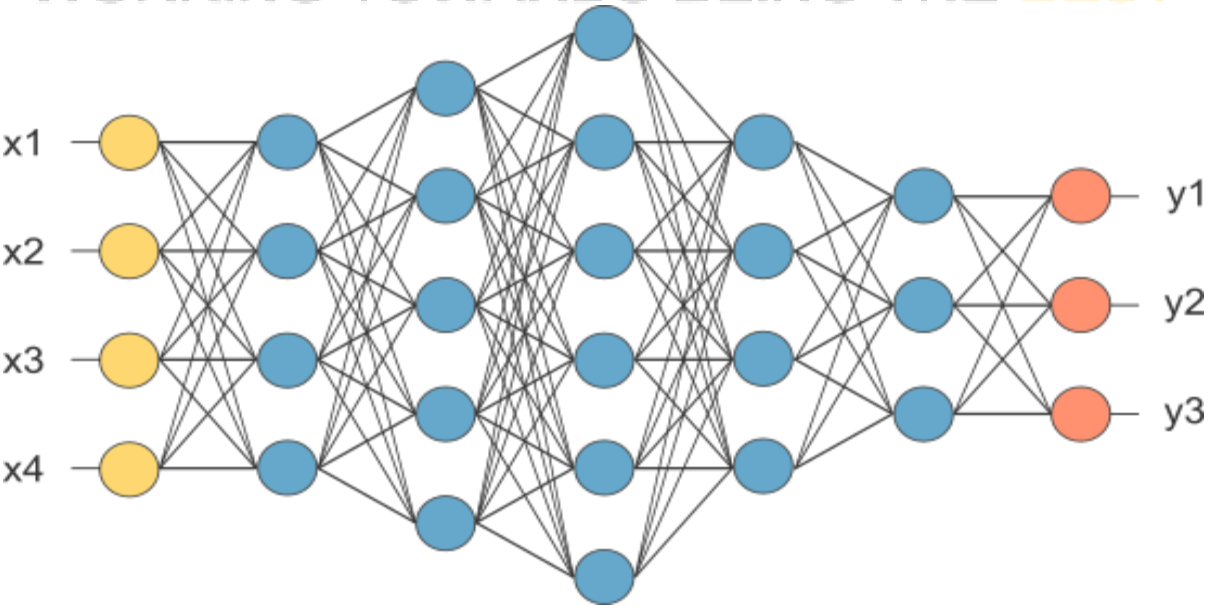


Figure 9 : After pooling layer, flattened as FC layer

In the above diagram, the feature map matrix will be converted as vector (x1, x2, x3, …). With the fully connected layers, we combined these features together to create a model. Finally, we

have an activation function such as softmax or sigmoid to classify the outputs as cat, dog, car, truck etc.,

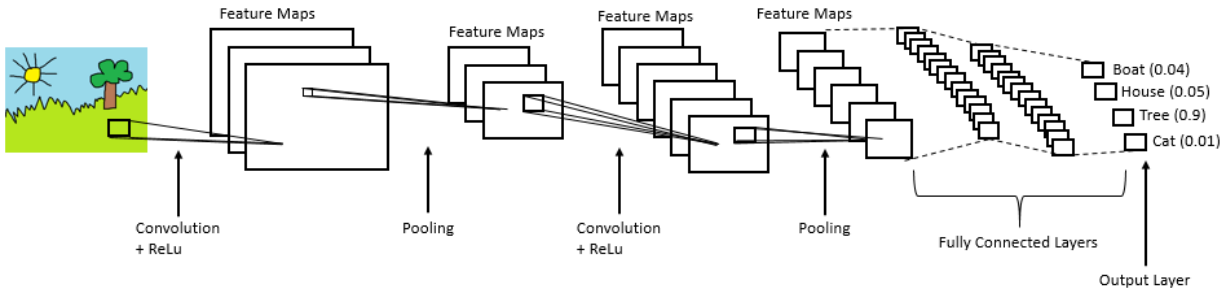


Figure 10 : Complete CNN architecture

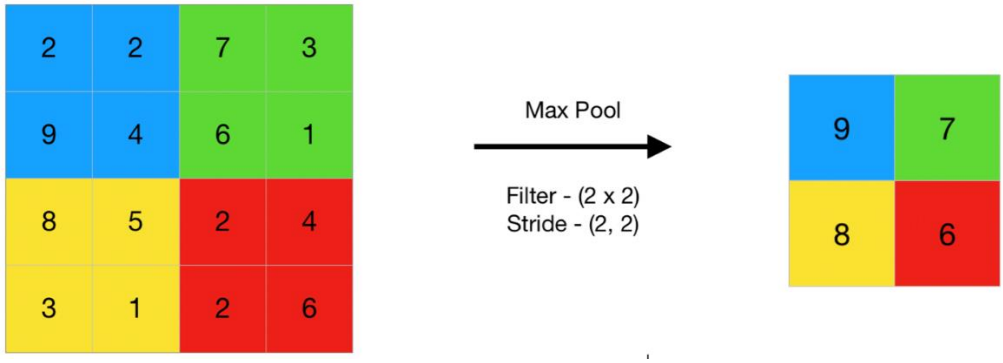
Pooling Layers?

- Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.
- The pooling layer summarises the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarised features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image.

Types of Pooling Layers

1. Max Pooling

Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.

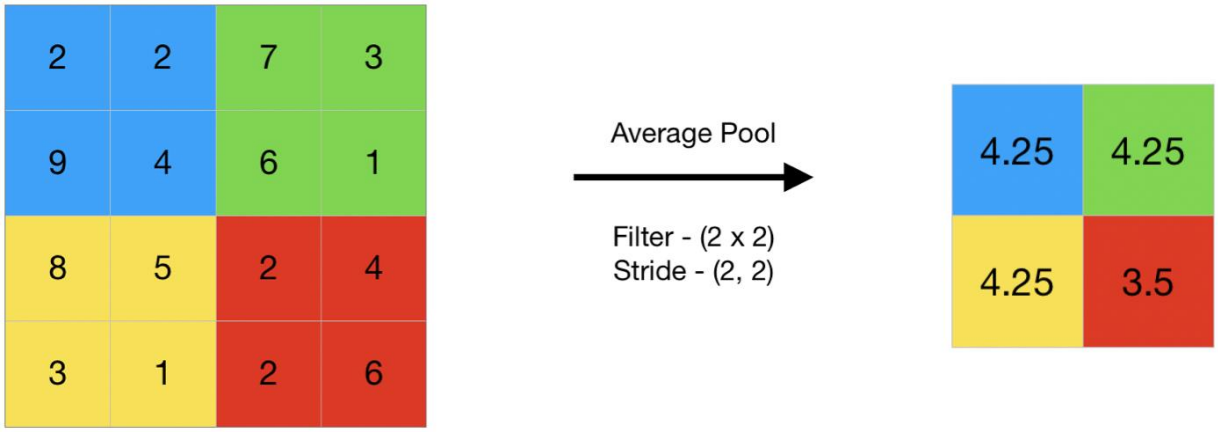


This can be achieved using MaxPooling2D layer in keras as follows:

2. Average Pooling

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives

the average of features present in a patch.



One shot learning

One-shot learning is an object categorization problem, found mostly in computer vision. Whereas most machine learning based object categorization algorithms require training on hundreds or thousands of samples/images and very large datasets, one-shot learning aims to learn information about object categories from one, or only a few, training samples/images.

The ability to learn object categories from few examples, and at a rapid pace, has been demonstrated in humans and it is estimated that a child has learned almost all of the 10 ~ 30 thousand object categories in the world by the age of six. This is due not only to the human mind's computational power, but also to its ability to synthesize and learn new object classes from existing information about different, previously learned classes. Given two examples from two different object classes: one, an unknown object composed of familiar shapes, the second, an unknown, amorphous shape; it is much easier for humans to recognize the former than the latter, suggesting that humans make use of existing knowledge of previously learned classes when learning new ones. The key motivation for the one-shot learning technique is that systems, like humans, can use prior knowledge about object categories to classify new objects

As with most classification schemes, one-shot learning involves three main challenges:

- *Representation*: How should we model objects and categories?
- *Learning*: How may we acquire such models?
- *Recognition*: Given a new image, how do we detect the presence of a known object/category amongst clutter, and despite occlusion, viewpoint, and lighting changes?

One-shot learning differs from single object recognition and standard category recognition algorithms in its emphasis on *knowledge transfer*, which makes use of prior knowledge of learnt categories and allows for learning on minimal training examples.

- **Knowledge transfer by model parameters**: One set of algorithms for one-shot learning achieves knowledge transfer through the reuse of model parameters, based on the similarity between previously and newly learned classes. Classes of objects are first learned on numerous training examples, then new object classes are learned using transformations of model parameters from the previously learnt classes or selecting relevant parameters for a classifier as in M. Fink.
- **Knowledge transfer by sharing features**: Another class of algorithms achieves knowledge transfer by sharing parts or features of objects across classes. In a paper presented at CVPR 2005 by Bart and Ullman, an algorithm extracts "diagnostic information" in patches from already learnt classes by maximizing the patches' mutual information, and then applies these features to the learning of a new class. A *dog* class, for example, may be learned in one shot from previous knowledge of *horse* and *cow* classes, because *dog* objects may contain similar distinguishing patches.
- **Knowledge transfer by contextual information**: Whereas the previous two groups of knowledge transfer work in one-shot learning relied on the similarity between new object classes and the previously learned classes on which they were based, transfer by contextual information instead appeals to global knowledge of the scene in which the object is placed. A paper presented at NIPS 2004 by K. Murphy et al. uses such global information as frequency distributions in a conditional random field framework to recognize objects Another algorithm by D. Hoiem et al. makes use of contextual information in the form of camera height and scene geometry to prune object detection. Algorithms of this type have two advantages. First, they should be able to learn object classes which are relatively dissimilar in visual appearance; and second, they should perform well precisely in situations where an image has not been hand-cropped and carefully aligned, but rather which naturally occur.

One-shot learning is a classification task where one, or a few, examples are used to classify many new examples in the future.

This characterizes tasks seen in the field of face recognition, such as face identification and face verification, where people must be classified correctly with different facial expressions, lighting conditions, accessories, and hairstyles given one or a few template photos.

Modern face recognition systems approach the problem of one-shot learning via face recognition by learning a rich low-dimensional feature representation, called a face embedding, that can be calculated for faces easily and compared for verification and identification tasks.

One-Shot Learning and Face Recognition

Typically, classification involves fitting a model given many examples of each class, then using the fit model to make predictions on many examples of each class.

For example, we may have thousands of measurements of plants from three different species. A model can be fit on these examples, generalizing from the commonalities among the measurements for a given species and contrasting differences in the measurements across species. The result, hopefully, is a robust model that, given a new set of measurements in the future, can accurately predict the plant species.

One-shot learning is a classification task where one example (or a very small number of examples) is given for each class, that is used to prepare a model, that in turn must make predictions about many unknown examples in the future.

In the case of one-shot learning, a single exemplar of an object class is presented to the algorithm.

— Knowledge transfer in learning to recognize visual objects classes, 2006.

This is a relatively easy problem for humans. For example, a person may see a Ferrari sports car one time, and in the future, be able to recognize Ferraris in new situations, on the road, in movies, in books, and with different lighting and colors.

Humans learn new concepts with very little supervision – e.g. a child can generalize the concept of "giraffe" from a single picture in a book – yet our best deep learning systems need hundreds or thousands of examples.

— Matching Networks for One Shot Learning, 2017.

One-shot learning is related to but different from zero-shot learning.

This should be distinguished from zero-shot learning, in which the model cannot look at any examples from the target classes.

— Siamese Neural Networks for One-shot Image Recognition, 2015.

Face recognition tasks provide examples of one-shot learning.

Specifically, in the case of face identification, a model or system may only have one or a few examples of a given person's face and must correctly identify the person from new photographs with changes to expression, hairstyle, lighting, accessories, and more.

In the case of face verification, a model or system may only have one example of a persons face on record and must correctly verify new photos of that person, perhaps each day.

As such, face recognition is a common example of one-shot learning.

Introduction to Dimensionality Reduction

Machine Learning: The machine learning is nothing but a field of study which allows computers to "learn" like humans without any need of explicit programming.

What is Predictive Modeling: Predictive modeling is a probabilistic process that allows us to forecast outcomes, on the basis of some predictors. These predictors are basically features that come into play when deciding the final result, i.e. the outcome of the model.

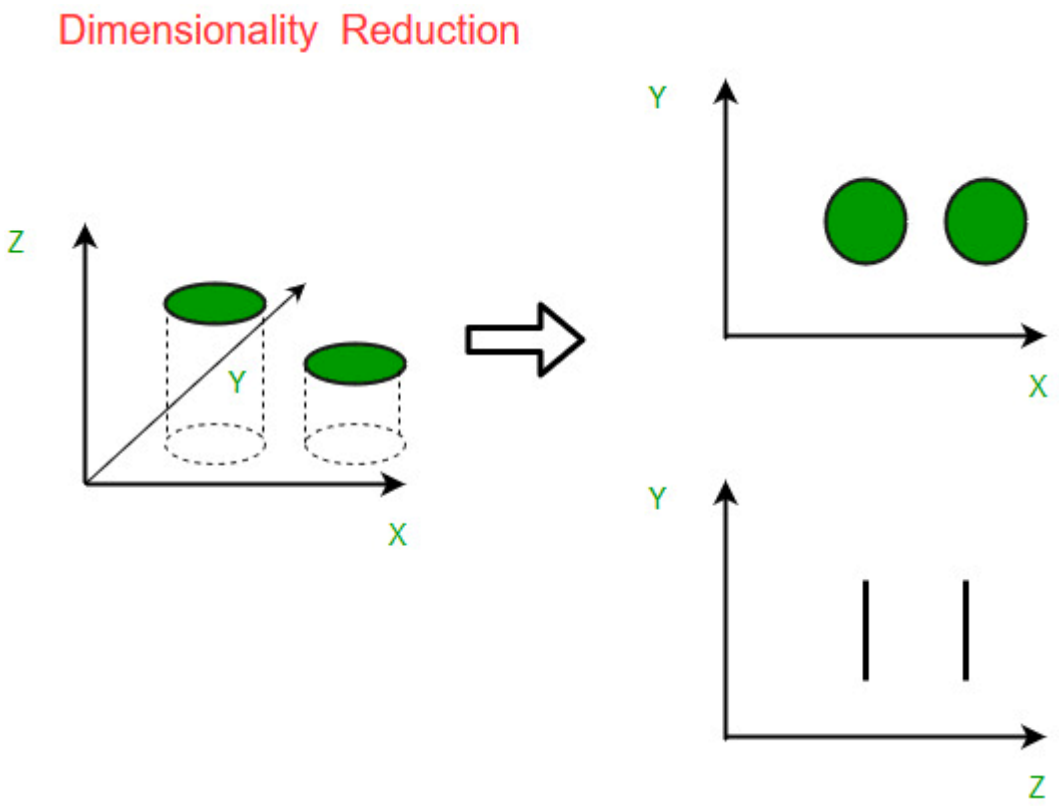
What is Dimensionality Reduction?

In machine learning classification problems, there are often too many factors on the basis of which the final classification is done. These factors are basically variables called features. The higher the number of features, the harder it gets to visualize the training set and then work on it. Sometimes, most of these features are correlated, and hence redundant. This is where dimensionality reduction algorithms come into play. Dimensionality reduction is the process of reducing the number of random variables under consideration, by obtaining a set of principal variables. It can be divided into feature selection and feature extraction.

Why is Dimensionality Reduction important in Machine Learning and Predictive Modeling?

An intuitive example of dimensionality reduction can be discussed through a simple e-mail classification problem, where we need to classify whether the e-mail is spam or not. This can involve a large number of features, such as whether or not the e-mail has a generic title, the content of the e-mail, whether the e-mail uses a template, etc. However, some of these features may overlap. In another condition, a classification problem that relies on both humidity and rainfall can be collapsed into just one underlying feature, since both of the aforementioned are correlated to a high degree. Hence, we can reduce the number of features in such problems. A 3-D classification problem can be hard to visualize, whereas a 2-D one can be mapped to a simple 2 dimensional space, and a 1-D problem to a simple line. The below figure illustrates this concept, where a 3-D

feature space is split into two 1-D feature spaces, and later, if found to be correlated, the number of features can be reduced even further.



Components of Dimensionality Reduction

There are two components of dimensionality reduction:

- **Feature selection:** In this, we try to find a subset of the original set of variables, or features, to get a smaller subset which can be used to model the problem. It usually involves three ways:
  1. Filter
  2. Wrapper
  3. Embedded
- **Feature extraction:** This reduces the data in a high dimensional space to a lower dimension space, i.e. a space with lesser no. of dimensions.

Methods of Dimensionality Reduction

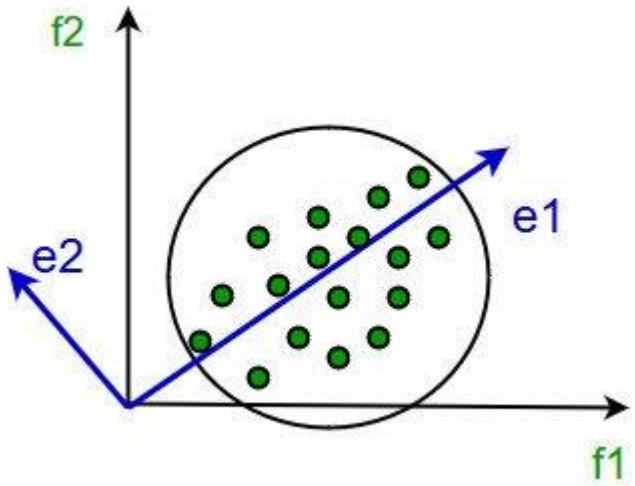
The various methods used for dimensionality reduction include:

- Principal Component Analysis (PCA)
- Linear Discriminant Analysis (LDA)
- Generalized Discriminant Analysis (GDA)

Dimensionality reduction may be both linear or non-linear, depending upon the method used. The prime linear method, called Principal Component Analysis, or PCA, is discussed below.

Principal Component Analysis

This method was introduced by Karl Pearson. It works on a condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.



It involves the following steps:

- Construct the covariance matrix of the data.
- Compute the eigenvectors of this matrix.
- Eigenvectors corresponding to the largest eigenvalues are used to reconstruct a large fraction of variance of the original data.

Hence, we are left with a lesser number of eigenvectors, and there might have been some data loss in the process. But, the most important variances should be retained by the remaining eigenvectors.

Advantages of Dimensionality Reduction

- It helps in data compression, and hence reduced storage space.
- It reduces computation time.
- It also helps remove redundant features, if any.

Disadvantages of Dimensionality Reduction

- It may lead to some amount of data loss.
- PCA tends to find linear correlations between variables, which is sometimes undesirable.
- PCA fails in cases where mean and covariance are not enough to define datasets.

- We may not know how many principal components to keep- in practice, some thumb rules are applied.

Inception Network V1

Inception net achieved a milestone in CNN classifiers when previous models were just going deeper to improve the performance and accuracy but compromising the computational cost. The Inception network, on the other hand, is heavily engineered. It uses a lot of tricks to push performance, both in terms of speed and accuracy. It is the winner of the ImageNet Large Scale Visual Recognition Competition in 2014, an image classification competition, which has a significant improvement over ZFNet (The winner in 2013), AlexNet (The winner in 2012) and has relatively lower error rate compared with the VGGNet (1st runner-up in 2014).

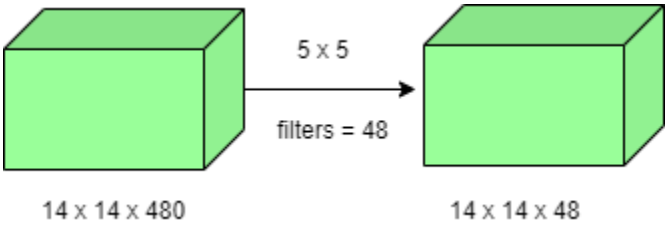
The major issues faced by deeper CNN models such as VGGNet were:

- Although, previous networks such as VGG achieved a remarkable accuracy on the ImageNet dataset, deploying these kinds of models is highly computationally expensive because of the deep architecture.
- Very deep networks are susceptible to overfitting. It is also hard to pass gradient updates through the entire network.

Before digging into Inception Net model, it's essential to know an important concept that is used in Inception network:

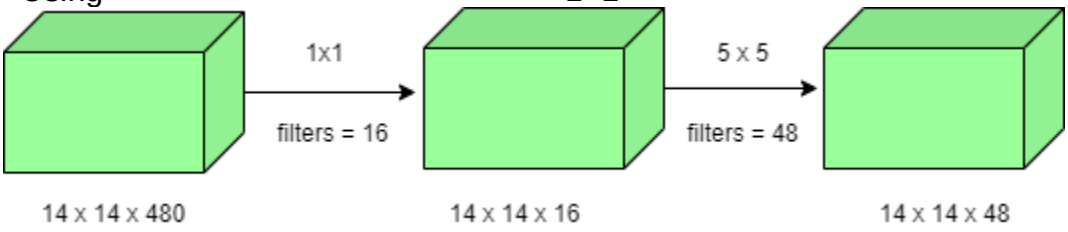
1 X 1 convolution: A *1×1* convolution simply maps an input pixel with all its respective channels to an output pixel. *1×1* convolution is used as a dimensionality reduction module to reduce computation to an extent.

- For instance, we need to perform *5×5* convolution without using *1×1* convolution as below:



Number of operations involved here is (14×14×48) × (5×5×480) = 112.9M

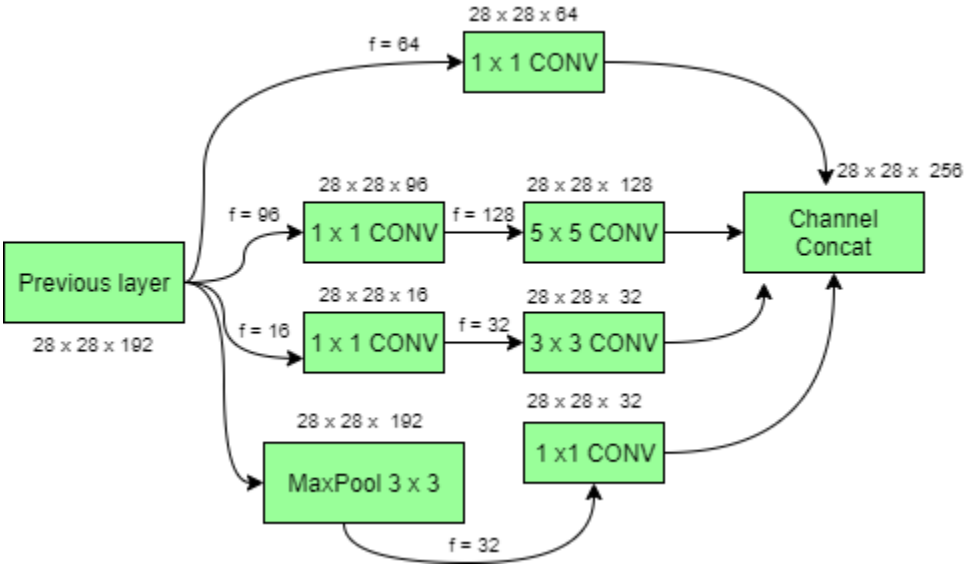
Using 1×1 convolution:



Number of operations for 1×1 convolution = (14×14×16) × (1×1×480) = 1.5M
Number of operations for 5×5 convolution = (14×14×48) × (5×5×16) = 3.8M
After addition we get, 1.5M + 3.8M = 5.3M

Which is immensely smaller than 112.9M! Thus, 1×1 convolution can help to reduce model size which can also somehow help to reduce the overfitting problem.

Inception model with dimension reductions:



Deep Convolutional Networks are computationally expensive. However, computational costs can be reduced drastically by introducing a 1 x 1 convolution. Here, the number of input channels is limited by adding an extra 1×1 convolution before the 3×3 and 5×5 convolutions. Though adding an extra operation may seem counter-intuitive but 1×1 convolutions are far cheaper than 5×5 convolutions. Do note that the 1×1 convolution is introduced after the max-pooling layer, rather than before. At last, all the channels in the network are concatenated together i.e. (28 x 28 x (64 + 128 + 32 + 32)) = 28 x 28 x 256.

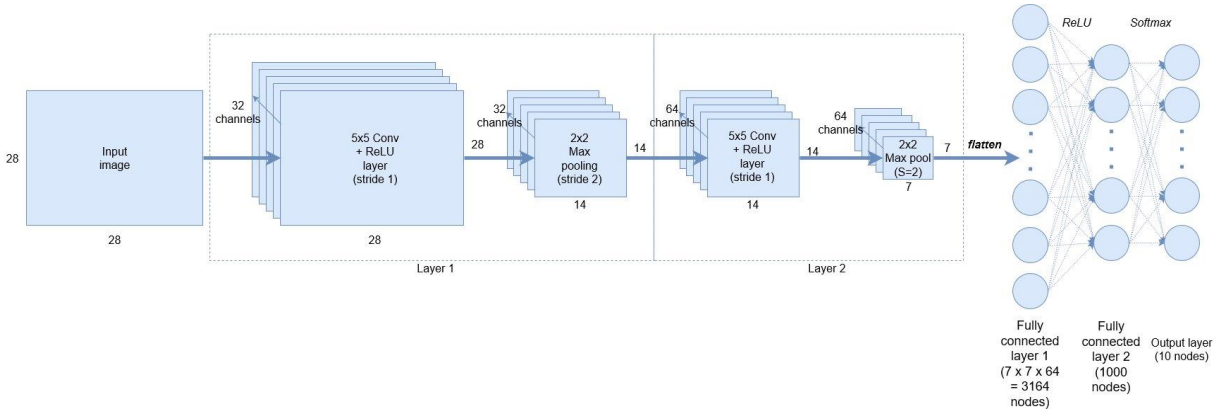
GoogLeNet Architecture of Inception Network:

This architecture has 22 layers in total! Using the dimension-reduced inception module, a neural network architecture is constructed. This is popularly known as GoogLeNet (Inception v1). GoogLeNet has 9 such inception modules fitted linearly. It is 22 layers deep (27, including the pooling layers). At the end of the architecture, fully connected layers were replaced by a global average pooling which calculates the average of every feature map. This indeed dramatically declines the total number of parameters.

Thus, Inception Net is a victory over the previous versions of CNN models. It achieves an accuracy of top-5 on ImageNet, it reduces the computational cost to a great extent without compromising the speed and accuracy.

A TensorFlow based convolutional neural network

TensorFlow makes it easy to create convolutional neural networks once you understand some of the nuances of the framework's handling of them. In this tutorial, we are going to create a convolutional neural network with the structure detailed in the image below. The network we are going to build will perform MNIST digit classification, as we have performed in previous tutorials (here and here). As usual, the full code for this tutorial can be found here.



Example convolutional neural network

As can be observed, we start with the MNIST 28×28 greyscale images of digits. We then create 32, 5×5 convolutional filters / channels plus ReLU (rectified linear unit) node activations. After this, we still have a height and width of 28 nodes. We then perform down-sampling by applying a 2×2 max pooling operation with a stride of 2. Layer 2 consists of the same structure, but now with 64 filters / channels and another stride-2 max pooling down-sample. We then flatten the output to get a fully connected layer with 3164 nodes, followed by another hidden layer of 1000 nodes. These layers will use ReLU node activations. Finally, we use a softmax classification layer to output the 10 digit probabilities.

Let's step through the code.

Input data and placeholders

The code below sets up the input data and placeholders for the classifier.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

# Python optimisation variables
learning_rate = 0.0001
epochs = 10
batch_size = 50

# declare the training data placeholders
# input x - for 28 x 28 pixels = 784 - this is the flattened image
data that is drawn from
# mnist.train.nextbatch()
x = tf.placeholder(tf.float32, [None, 784])
# dynamically reshape the input
x_shaped = tf.reshape(x, [-1, 28, 28, 1])
# now declare the output data placeholder - 10 digits
y = tf.placeholder(tf.float32, [None, 10])
```

TensorFlow has a handy loader for the MNIST data which is sorted out in the first couple of lines. After that we have some variable declarations which determine the optimisation behaviour (learning rate, batch size etc.). Next, we declare a placeholder (see this tutorial for explanations of placeholders) for the image input data, *x*. The image input data will be extracted using the mnist.train.nextbatch() function, which supplies a flattened 28×28=784 node, single channel greyscale representation of the image. However, before we can use this data in the TensorFlow convolution and pooling functions, such as conv2d() and max_pool() we need to reshape the data as these functions take 4D data only.

The format of the data to be supplied is [i, j, k, l] where *i* is the number of training samples, *j* is the height of the image, *k* is the weight and *l* is the channel number. Because we have a greyscale image, *l* will always be equal to 1 (if we had an RGB image, it would be equal to 3). The MNIST images are 28 x 28, so both *j* and *k* are equal to 28. When we reshape the input data *x* into *x_shaped*, theoretically we don't know the size of the first dimension of *x*, so we don't know what *i* is. However, tf.reshape() allows us to put -1 in place of *i* and it will dynamically reshape based on the number of training samples as the training is performed. So we use [-1, 28, 28, 1] for the second argument in tf.reshape().

Finally, we need a placeholder for our output training data, which is a [?, 10] sized tensor – where the 10 stands for the 10 possible digits to be classified. We will use the

mnist.train.next_batch() to extract the digits labels as a one-hot vector – in other words, a digit of "3" will be represented as [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

Defining the convolution layers

Because we have to create a couple of convolutional layers, it's best to create a function to reduce repetition:

```
def create_new_conv_layer(input_data, num_input_channels, num_filters,
filter_shape, pool_shape, name):
    # setup the filter input shape for tf.nn.conv_2d
    conv_filt_shape = [filter_shape[0], filter_shape[1], num_input_channels,
                       num_filters]

    # initialise weights and bias for the filter
    weights = tf.Variable(tf.truncated_normal(conv_filt_shape, stddev=0.03),
                                      name=name+'_W')
    bias = tf.Variable(tf.truncated_normal([num_filters]), name=name+'_b')

    # setup the convolutional layer operation
    out_layer = tf.nn.conv2d(input_data, weights, [1, 1, 1, 1], padding='SAME')

    # add the bias
    out_layer += bias

    # apply a ReLU non-linear activation
    out_layer = tf.nn.relu(out_layer)

    # now perform max pooling
    ksize = [1, pool_shape[0], pool_shape[1], 1]
    strides = [1, 2, 2, 1]
    out_layer = tf.nn.max_pool(out_layer, ksize=ksize, strides=strides,
                               padding='SAME')

    return out_layer
```

I'll step through each line/block of this function below:

```
conv_filt_shape = [filter_shape[0], filter_shape[1], num_input_channels,
                       num_filters]
```

This line sets up a variable to hold the shape of the weights that determine the behaviour of the 5×5 convolutional filter. The format that the conv2d() function receives for the filter is: [filter_height, filter_width, in_channels, out_channels]. The height and width of the filter are provided in the filter_shape variables (in this case [5, 5]). The number of input channels, for the first convolutional layer is simply 1, which corresponds to the single channel greyscale MNIST image. However, for the second convolutional layer it takes the output of the first convolutional layer, which has a 32 channel output. Therefore, for the second convolutional layer, the input channels is 32. As defined in the block diagram above, the number of output channels of the first layer is 32, and for the second layer it is 64.

```
# initialise weights and bias for the filter
weights = tf.Variable(tf.truncated_normal(conv_filt_shape, stddev=0.03),
                                  name=name+'_W')
bias = tf.Variable(tf.truncated_normal([num_filters]), name=name+'_b')
```

In these lines we create the weights and bias for the convolutional filter and randomly initialise the tensors. If you need to brush up on these concepts, check out this tutorial.

```
# setup the convolutional layer operation
out_layer = tf.nn.conv2d(input_data, weights, [1, 1, 1, 1], padding='SAME')
```

This line is where we setup the convolutional filter operation. The variable *input_data* is self-explanatory, as are the weights. The size of the weights tensor show TensorFlow what size the convolutional filter should be. The next argument [1, 1, 1, 1] is the *strides* parameter that is required in conv2d(). In this case, we want the filter to move in steps of 1 in both the *x* and *y* directions (or height and width directions). This information is conveyed in the strides[1] and strides[2] values – both equal to 1 in this case. The first and last values of *strides* are always equal to 1, if they were not, we would be moving the filter between training samples or between channels, which we don't want to do. The final parameter is the padding. Padding determines the output size of each channel and when it is set to "SAME" it produces dimensions of:

out_height = ceil(float(in_height) / float(strides[1]))

out_width = ceil(float(in_width) / float(strides[2]))

For the first convolutional layer, *in_height = in_width = 28*, and *strides[1] = strides[2] = 1*. Therefore the padding of the input with 0.0 nodes will be arranged so that the *out_height =*

out_width = 28 – there will be no change in size of the output. This padding is to avoid the fact that, when traversing a (*x,y*) sized image or input with a convolutional filter of size (*n,m*), with a stride of 1 the output would be (*x-n+1,y-m+1*). So in this case, without padding, the output size would be (24,24). We want to keep the sizes of the outputs easy to track, so we chose the "SAME" option as the padding so we keep the same size.

```
# add the bias
out_layer += bias
# apply a ReLU non-linear activation
out_layer = tf.nn.relu(out_layer)
```

In the two lines above, we simply add a bias to the output of the convolutional filter, then apply a ReLU non-linear activation function.

```
# now perform max pooling
ksize = [1, pool_shape[0], pool_shape[1], 1]
strides = [1, 2, 2, 1]
out_layer = tf.nn.max_pool(out_layer, ksize=ksize, strides=strides,
                           padding='SAME')

return out_layer
```

The max_pool() function takes a tensor as its first input over which to perform the pooling. The next two arguments *ksize* and *strides* define the operation of the pooling. Ignoring the first and last values of these vectors (which will always be set to 1), the middle values of *ksize* (*pool_shape[0]* and *pool_shape[1]*) define the shape of the max pooling window in the *x* and *y* directions. In this convolutional neural networks example, we are using a 2×2 max pooling window size. The same applies with the *strides* vector – because we want to down-sample, in this example we are choosing strides of size 2 in both the *x* and *y* directions (*strides[1]* and strides[2]). This will halve the input size of the (*x,y*) dimensions.

Finally, we have another example of a padding argument. The same rules apply for the 'SAME' option as for the convolutional function conv2d(). Namely:

out_height = ceil(float(in_height) / float(strides[1]))
out_width = ceil(float(in_width) / float(strides[2]))

Punching in values of 2 for *strides[1]* and *strides[2]* for the first convolutional layer we get an output size of (14, 14). This is a halving of the input size (28, 28), which is what we are looking for. Again, TensorFlow will organise the padding so that this output shape is what is achieved, which makes things nice and clean for us.

Finally we return the out_layer object, which is actually a sub-graph of its own, containing all the operations and weight variables within it. We create the two convolutional layers in the main program by calling the following commands:

```
# create some convolutional layers
layer1 = create_new_conv_layer(x_shaped, 1, 32, [5, 5], [2, 2], name='layer1')
layer2 = create_new_conv_layer(layer1, 32, 64, [5, 5], [2, 2], name='layer2')
```

As you can see, the input to *layer1* is the shaped input *x_shaped* and the input to *layer2* is the output of the first layer. Now we can move on to creating the fully connected layers.

The fully connected layers

As previously discussed, first we have to flatten out the output from the final convolutional layer. It is now a 7×7 grid of nodes with 64 channels, which equates to 3136 nodes per training sample. We can use tf.reshape() to do what we need:

```
flattened = tf.reshape(layer2, [-1, 7 * 7 * 64])
```

Again, we have a dynamically calculated first dimension (the -1 above), corresponding to the number of input samples in the training batch. Next we setup the first fully connected layer:

```
# setup some weights and bias values for this layer, then activate
with ReLU
wd1 = tf.Variable(tf.truncated_normal([7 * 7 * 64, 1000], stddev=0.03),
name='wd1')
bd1 = tf.Variable(tf.truncated_normal([1000], stddev=0.01), name='bd1')
dense_layer1 = tf.matmul(flattened, wd1) + bd1
dense_layer1 = tf.nn.relu(dense_layer1)
```

If the above operations are unfamiliar to you, please check out my previous TensorFlow tutorial. Basically we are initialising the weights of the fully connected layer, multiplying them

with the flattened convolutional output, then adding a bias. Finally, a ReLU activation is applied. The next layer is defined by:

```
# another layer with softmax activations
wd2 = tf.Variable(tf.truncated_normal([1000, 10], stddev=0.03), name='wd2')
bd2 = tf.Variable(tf.truncated_normal([10], stddev=0.01), name='bd2')
dense_layer2 = tf.matmul(dense_layer1, wd2) + bd2
y_ = tf.nn.softmax(dense_layer2)
```

This layer connects to the output, and therefore we use a soft-max activation to produce the predicted output values *y_*. We have now defined the basic structure of our convolutional neural network. Let's now define the cost function.

The cross-entropy cost function

We could develop our own cross-entropy cost expression, as we did in the previous TensorFlow tutorial, based on the value *y_*. However, then we have to be careful about handling NaN values. Thankfully, TensorFlow provides a handy function which applies soft-max followed by cross-entropy loss:

```
cross_entropy =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=dense_layer2,
labels=y))
```

The function *softmax_cross_entropy_with_logits()* takes two arguments – the first (*logits*) is the output of the matrix multiplication of the final layer (plus bias) and the second is the training target vector. The function first takes the soft-max of the matrix multiplication, then compares it to the training target using cross-entropy. The result is the cross-entropy calculation per training sample, so we need to reduce this tensor into a scalar (a single value). To do this we use *tf.reduce_mean()* which takes a mean of the tensor.

The training of the convolutional neural network

The following code is the remainder of what is required to train the network. It is a replication of what is explained in my previous TensorFlow tutorial, so please refer to that tutorial if anything is unclear. We'll be using mini-batches to train our network. The essential structure is:

- Create an optimiser
- Create correct prediction and accuracy evaluation operations

- Initialise the operations
- Determine the number of batch runs within an training epoch
- For each epoch:
- For each batch:
- Extract the batch data
- Run the optimiser and cross-entropy operations
- Add to the average cost
- Calculate the current test accuracy
- Print out some results
- Calculate the final test accuracy and print

The code to execute this is:

```
# add an optimiser
optimiser =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cross_entropy)

# define an accuracy assessment operation
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# setup the initialisation operator
init_op = tf.global_variables_initializer()

with tf.Session() as sess:
    # initialise the variables
    sess.run(init_op)
    total_batch = int(len(mnist.train.labels) / batch_size)
    for epoch in range(epochs):
        avg_cost = 0
```

```
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size=batch_size)
            _, c = sess.run([optimiser, cross_entropy],
                            feed_dict={x: batch_x, y: batch_y})
            avg_cost += c / total_batch
        test_acc = sess.run(accuracy,
                            feed_dict={x: mnist.test.images, y: mnist.test.labels})
        print("Epoch:", (epoch + 1), "cost =", "{:.3f}".format(avg_cost), "
                test accuracy: {:.3f}".format(test_acc))

    print("\nTraining complete!")
    print(sess.run(accuracy, feed_dict={x: mnist.test.images, y:
mnist.test.labels}))
```

The final code can be found on this site's GitHub repository. Note the final code on that repository contains some TensorBoard visualisation operations, which have not been covered in this tutorial and will have a dedicated future article to explain.

Caution: This is a relatively large network and on a standard home computer is likely to take at least 10-20 minutes to run.

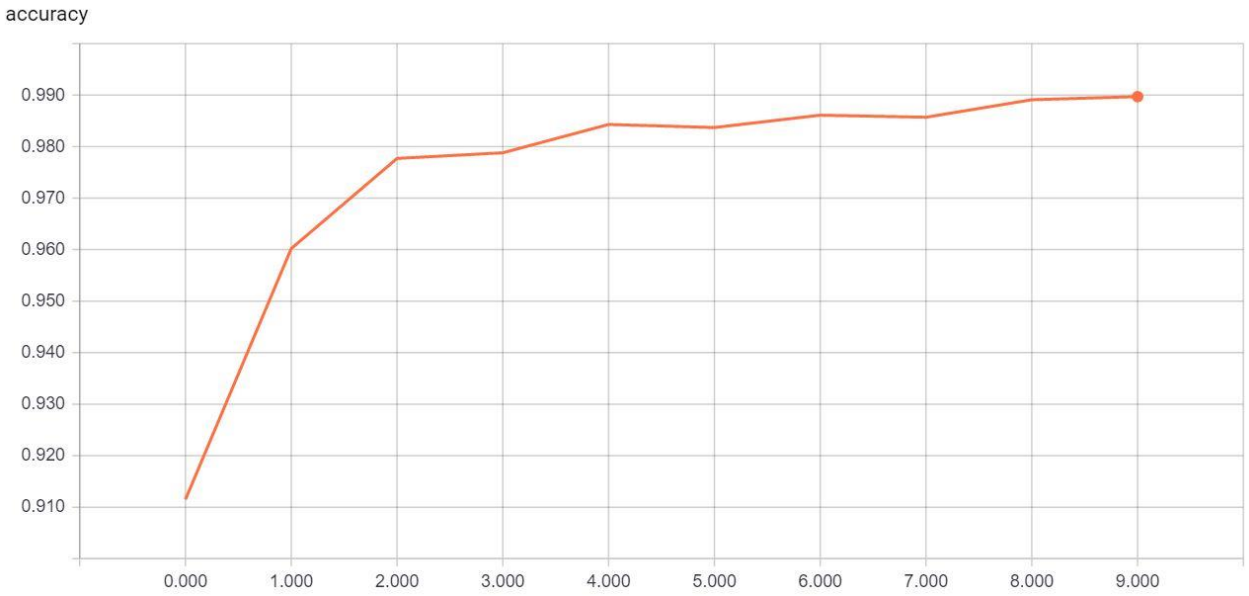
The results

Running the above code will give the following output:

```
Epoch: 1 cost = 0.739  test accuracy: 0.911
Epoch: 2 cost = 0.169  test accuracy: 0.960
Epoch: 3 cost = 0.100  test accuracy: 0.978
Epoch: 4 cost = 0.074  test accuracy: 0.979
Epoch: 5 cost = 0.057  test accuracy: 0.984
Epoch: 6 cost = 0.047  test accuracy: 0.984
Epoch: 7 cost = 0.040  test accuracy: 0.986
Epoch: 8 cost = 0.034  test accuracy: 0.986
Epoch: 9 cost = 0.029  test accuracy: 0.989
Epoch: 10 cost = 0.025  test accuracy: 0.990

Training complete!
0.9897
```

We can also plot the test accuracy versus the number of epoch's using TensorBoard (TensorFlow's visualisation suite):



Convolutional neural network MNIST accuracy

As can be observed, after 10 epochs we have reached an impressive prediction accuracy of 99%. This result has been achieved without extensive optimisation of the convolutional neural network's parameters, and also without any form of regularisation. This is compared to the best accuracy we could achieve in our standard neural network ~98% – as can be observed in the previous tutorial.

The accuracy difference will be even more prominent when comparing standard neural networks with convolutional neural networks on more complicated data-sets, like the CIFAR data. However, that is a topic for another day. Have fun using TensorFlow and convolutional neural networks! By the way, if you want to see how to build a neural network in Keras, a more stream-lined framework, check out my Keras tutorial. Also, if you'd like to explore more deep learning architectures in TensorFlow, check out my recurrent neural networks and LSTM tutorial.