Deep Learning

Samuel Navarro

June 9, 2019

Contents

1 Tensorflow

A is a 0-dimensional int32 tensor.

```
A = tf.constant(1234)
```

B is a 1-dimensional int32 tensor.

```
B = tf.constant([123, 456, 789])
```

C is a 2-dimensional int32 tensor

```
C = tf.context([ [123, 456, 789], [222, 333, 444] ] )
```

A *Tensorflow Session* as shown above, is an environment for running graph. The session is in charge of allocating the operations to GPUs and/or CPUs, including remote machines.

What if you want to use a non-constant?

This is where `tf.placeholder()` and `feed_dict` come into place.

You can't just set `x` to your dataset and put it in Tensorflow, because over time you'll want your Tensorflow model to take in different datasets with different parameters. You need `tf.placeholder()`.

```
x = tf.placeholder(tf.string)
y = tf.placeholder(tf.int32)
z = tf.placeholder(tf.float32)


with tf.Session() as sess:
        output = sess.run(x, feed_dict={x: 'Hello World',
        y: 123, z: 45.67})
```

Listing 1: Feed Dict

`tf.placeholder()` returns a tensor that gets its value from data passed to the `tf.session.run()` function, allowing you tu set the input right before the sessions runs.

Session's feed_dict

Tensorflow Linear Function

The way you turn your scores into probabilities is using the **softmax** function. Scores in the context of logistic regression are also called logits.

Softmax function The softmax function should take in `x`, a one or two dimensional array of logits. In the one dimensional case, the array is just a single set of logits. In the two dimensional case, *each column in the array is a set of logits.* Thats why we use `axis=0`

What happens to the softmax probabilities when you multiply the logits by 10?

Probabilities get close to 0.0 or 1.0

What happens to the softmax probabilities when you divide the logits by 10? The probabilities get close to the uniform distribution.

Since all the scores decrease in magnitude, the resulting softmax probabilities will be closer to each other.

1.1 Gradient Descent

One simple principle is that we always want our variables to have zero mean and equal variance whenever possible. This is also good when we do optimization.

In the case of images, you can do:

$$\frac{Channel - 128}{128}$$

.

And, we also want to have a good weight initialization. There is a lot of methods to make this but for now we can just use the weights from a Gaussian distribution with mean zero and sd sigma. The sigma values

represent the order of magnitude of your output at the initial point of your optimization.

Because of the use of softmax in top of it, the order of magnitude also determines the peakiness of you initial probability distribution. Large sigma means your distribution will have large peaks, very opinitionated.

A small sigma means that your distributions is very uncertain about things. It's better to begin with uncertain distribution and let the optimization more confident as the training progress.

So, we should use the small sigma to begin with.

We need to measure the error on the test data.

One rule of thumb is that if computing your loss takes n floating points of calculation, computing the gradients takes 3 times that compute.

And because the loss is huge because it depends on all the data, the gradients could be a very expensive operation.

In order to same computation power, we can cheat, we can use the average loss for a very small fraction of the training data. (something like from 1 to 1000 training samples).

The way we pick our samples must be random, in other case it would not work.

So, the process is like this: **Stochastic Gradient Descent**

1. Pick a random set of examples.
2. Compute the loss for that sample.
3. Compute the derivative of that sample.
4. Pretend that that derivate is the right direction to use. (Is not right at all the right direction)

Each calculation is faster because is trained on less data but the price we pay is that we need to take many more steps.

SGD scales well with model data and model size. But because is fundamentally a pretty bad optimizer, it comes with a lot of issues in practice.

Tricks to help **SGD**

- Inputs of zero mean and equal variance (small).
- Initial weigths should be random with mean = 0 and equal variance (small)
- Momentum: We can run an average of the gradient $M < -0.9M + \delta L$ Beause the gradient give us the direction, this result in more faster convergence.
- Learning Rate Decay: Is benefical to take smaller and smaller steps as you train. Some like to apply an exponential decay to the learning

rate, some try to make it smaller every time the learning rate reaches its plateau. There's a lot of ways to go about it but the key thing is to lower every time.

About the learning rate tunning, you should never trust how fast you train, it's better to thing on terms of how good you train.

So, it should be the case that a lower learning rate give us lower loss but in more steps.

This is why **SGD** is often refered as black magic. Because you often have many hyperparameters:

- initial learning rate
- momentum
- batch size
- weight initialization

One of the rules of thumb is that when things don't work, always try to lower your learning rate first.

Adagrad is a modification of **SGD** which implicitly does momentum and learning decay for you.

1.2 Mini-batching

Is a technique for training on subsets of the dataset instead of all the data at one time. This provides the ability to train a model, even if a computer lacks the memory to store the entire dataset.

Mini-batching is computationally inefficient, since you can't calculate the loss simultaneously across all samples. However, this is a small price to pay in order to be able to run the model at all.

It's also quite useful combined with **SGD**. The idea is:

1. Randomly shuffle the data at the start of each epoch.
2. Create the mini-batches.
3. For each mini-batch, you train the network weights with gradient descent.

Since these batches are random, you're performing **SGD** with each batch.

Example of the sizes:

In case where the size of the batches would vary, we can take advantage of TensorFlow's `tf.placeholder()` function to receive the varying batch sizes.

Table 1: Float 32 Size

train_features	Shape: (55000, 784)	Type: float32	172480000 bytes
train_labels	Shape: (55000, 10)	Type: float32	2200000 bytes
weights	Shape: (784,10)	Type: float32	31360 bytes
biases	Shape: (10,)	Type: float32	40 bytes

```
features = tf.placeholder(tf.float32, [None, n_input])
labels = tf.placeholder(tf.float32, [None, n_classes])
```

The **None** dimension is a placeholder for the batch size. At runtime, Tensorflow will accept any batch size greater than 0.

2 Deep Neural Networks

2.1 Regularization

- Early Termination: We stop training as soon as we stop improving.
- Regularization: Applying artificial constraints on my network that implicitly reduce the number of free parameters.
  - L2 Regularization: $\mathcal{L}' = \mathcal{L} + \beta \frac{1}{2} ||\omega||_2^2$
- Dropout:
  - Image you have one layer that goes to another layer, the values that goes from the next are often called activations, now, take those activations and randomly for every example you train your network, set $p \epsilon \{0, 1\}$ to zero. By doing this, your network cannot rely on any activation to be present.
  - **If dropout doesn't work, you should probably be using a bigger network.**
  - When you want to evaluate your model, you don't want randomness, you want your model to be deterministic. You want to take the consensus over this redundant models.
  - You get the consensus by taking the average of the redundant models.
  - $y_e \sim E(y_t)$. To make sure this expectation holds:
    - During training: Not only zero-out the activation with p probability, but you also scale the remaining activations by a factor of 2 (In this case 2 because we used p=0.5).
    - In order to compensate for dropped units, `tf.nn.dropout()` multiplies all units that are kept by `1 / keep_drop`

* During validation or testing, you should keep all of the units to maximize accuracy.

Remember that L2 norm stands for:

$$\frac{1}{2}(\omega_1^2 + \omega_2^2 + ... + \cdots + \omega_n^2).$$

This is easy because we can even take the derivate by hand (That is: ω)

3 Convolutional Neural Networks

If you know the structure of your data, it will help a lot for the learning process.

Let's say that you want to classify letters, you already know that the color space doesn't matter in the letters, so, it would be better to use `R + G + B / 3` instead of the 3 channels.

Another useful things is **Translation Invariance**: the idea is that you explicitly tell your network that objects in images are largely the same wherever they appear on the image. (Different positions, same object).

Another example of **Translation Invariance** is that when you train a Neural Network in text, most of the time, it doesn't matter whether the Kitten world appears in the text, so you don't need to learn the text Kitten every time it appears on the sentence.

The way you achieve this in Neural Networks is using what is called **Weight Sharing**: When you know that two inputs can contain the same kind of information, you share the weights and train the weights jointly for those inputs.

Statistical Invariance: Things that don't change on average across time and space.

For images, the idea of **Weight Sharing** will get us to study *Convolutional Neural Networks*, for text and sequences in general, in will lead us to embeddings and *Recurrent Neural Networks*

ConvNets: are Neural Networks that share their parameters across space.[1]

The amount by which the filter slides is referred to as the 'stride'. The stride is a hyperparameter. Increasing the stride reduces the size of your model by reducing the number of total patches each layer observes.

However, this usually comes with a reduction in accuracy.

What's important with the kernels, is that we are grouping together adjacent pixels and treating them as a collective. In a normal, non-convolutional neural network, we would have ignored this adjacency. In a normal network, we would have connected every pixel in the input image to a neuron in the

[1] There's a really clear explanation in the video from the resources

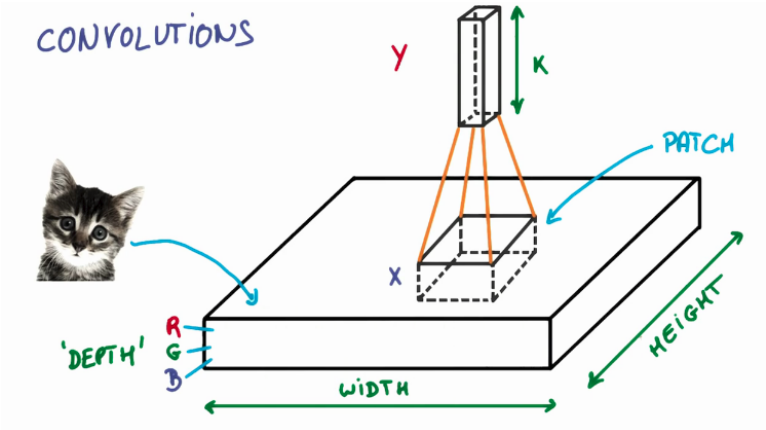

Figure 1: convolutions

next layer. In doing so, we would not have taken advantage of the fact that pixels in an image are close together for a reason and have special meaning.

Filter Depth

It's common to have more than one filter. Different filters pick up different qualities of a patch. For example, one filter might look for a particular color, while another might look for a kind of object of a specific shape. The amount of filters in a convolutional layer is called the filter depth.

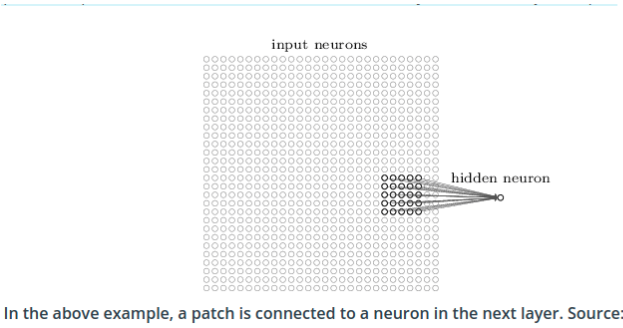

Figure 2: Patch_connection

How many neurons does each patch connect to?

That's dependent on our filter depth. If we have a depth of `k`, we connect each patch of pixels to `k` neurons in the next layer. This gives us the height of `k` in the next layer, as shown below. In practice, `k` is a hyperparameter we tune, and most CNNs tend to pick the same starting values.

But why connect a single patch to multiple neurons in the next layer? Isn't one neuron good enough?

Multiple neurons can be useful because a patch can have multiple interesting characteristics that we want to capture.

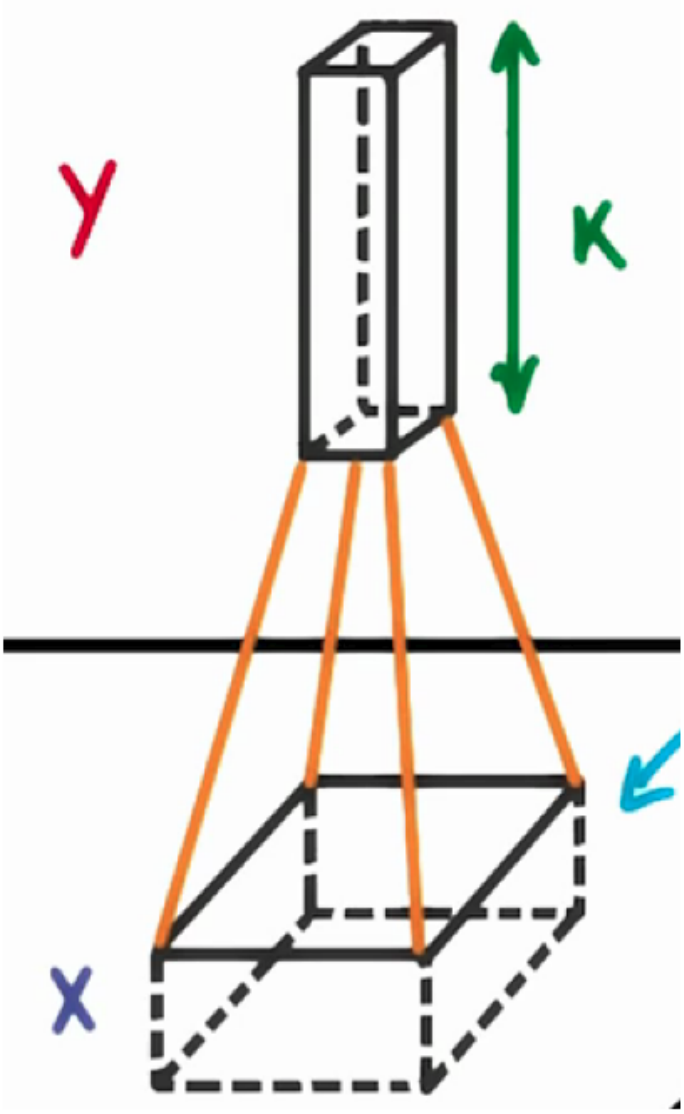

Figure 3: Patch_Connection

For example, one patch might include some white teeth, some blonde whiskers, and part of a red tongue. In that case, we might want a filter depth of at least three - one for each of teeth, whiskers, and tongue.

Having multiple neurons for a given patch ensures that our CNN can learn to capture whatever characteristics the CNN learns are important.

3.1 Parameter Sharing

As we saw earlier, the classification of a given patch in an image is determined by the weights and biases corresponding to that patch.

If we want a cat that's in the top left patch to be classified in the same way as a cat in the bottom right patch, we need the weights and biases corresponding to those patches to be the same, so that they are classified the same way.

This is exactly what we do in CNNs. The weights and biases we learn for a given output layer are shared across all patches in a given input layer. Note that as we increase the depth of our filter, the number of weights and biases we have to learn still increases, as the weights aren't shared across the output channels.

Another thing is that **sharing parameters** not only helps us with **translation invariance**, but also give us a smaller, more scalable model.

3.2 Padding

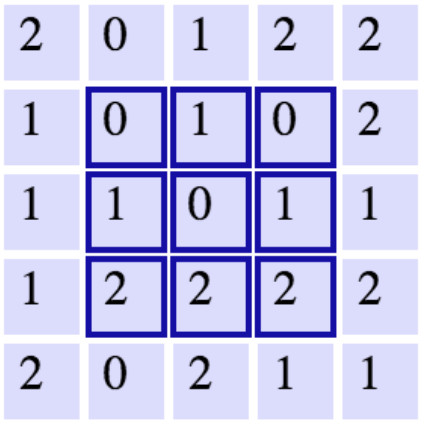

Figure 4: padding

We see that we can fit at most three patches in each direction, giving us a dimension of 3x3 in our next layer.

As we can see, the width and height of each subsequent layer decreases in the above scheme. But we want to maintain the same width and height across layers so that we can continue to add layers without worrying about the dimensionality shrinking and so that we have consistency.

One way is to simply add a border of 0s to our original `5x5` image.[2]

[2] **Valid Padding:** you don't go pass the edge. **Same Padding:** You go off the edge

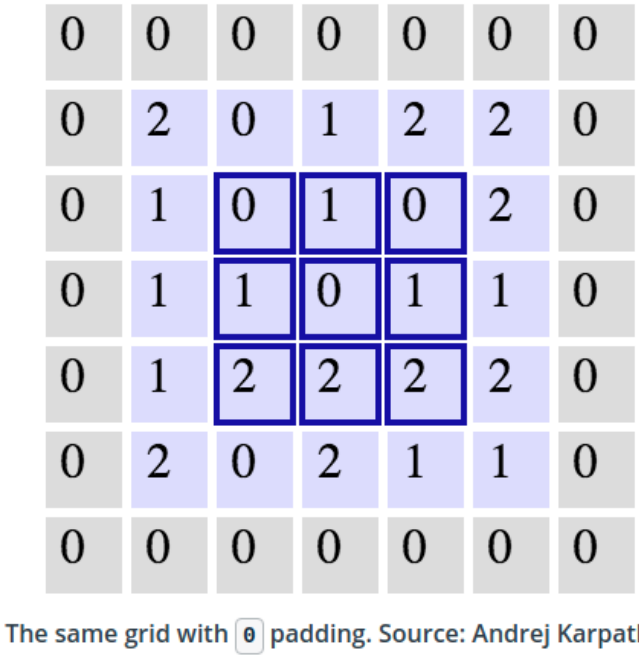

Figure 5: 0 Padding

The Figure 5 ensures that we would expand our original image to a `7x7`. With this, we now see how our next layer's size is again `5x5`

3.2.1 Dimensionality

Given:

- our input layer has a width of `W` and a height of
- our convolutional layer has a filter size `F`
- We have a stride `S`
- Padding `P`
- Number of filters `K`

To obtain the Width or Height of the next layer:

$$W/H_{out} = \frac{W/H - F + 2P}{S} + 1.$$

and put zeros.

```
input = tf.placeholder(tf.float32, (None, 32, 32, 3))
# (height, width, input_depth, output_depth)
filter_weights = tf.Variable(tf.truncated_normal((8, 8, 3, 20)))
filter_bias = tf.Variable(tf.zeros(20))
strides = [1, 2, 2, 1] # (batch, height, width, depth)
padding = 'SAME'
conv = tf.nn.conv2d(input, filter_weights, strides, padding)
+ filter_bias
```

Listing 2: Convolution Output Shape

The output depth would be equal to the number of filters $D_{out} = K$.
The output volume would be $W_{out} * H_{out} * D_{out}$
Example:

- Input shape 32x32x3
- 20 filters 8x8x3
- Stride 2 for both height and width
- Padding = 1

The output is 14x14x20, but one important thing to note is that we won't get the same result with Tensorflow because it uses a different algorithm (it uses the `ceil` of the output)

Continuing with the example, without parameter sharing, How many parameters does the convolutional layer have?

The **hint** is that without parameter sharing, each neuron in the output layer must connect to each neuron in the filter. In addition, each neuron in the output layer must also connect to a single bias neuron.

The output layer was of shape: $14x14x20$ so, the answer is:

$$parameters = (14 * 14 * 20) * (8 * 8 * 3 + 1) = 756560.$$

Parameter Sharing

This is the number of parameters actually used in a convolution layer `tf.nn.conv2d()`.

Again, the output layer shape is $14x14x20$.

The **hint** is that with parameter sharing, each neuron in an output channel shares its weights with every other neuron in that channel. So the number of parameters is equal to the number of neuron in the filter, plus a bias neuron, all multiplied by the number fo channels in the output layer. (or, the number of filters?)

$$parameters = (8 * 8 * 3 + 1) * 20 = 3860.$$

With weight sharing we use the same filter for an entire depth slice. Because of this we can get rid of $14 * 14$ and be left with only 20.

3.2.2 Tensorflow Convolution Layer

The example in Code 3 shows Convolutional layers in Tensorflow.

In TensorFlow, `strides` is an array of 4 elements; the first element in this array indicates the stride for batch and last element indicates stride for features. It's good practice to remove the batches or features you want to skip from the data set rather than use a stride to skip them.

You can always set the first and last element to **1** in `strides` in order to use all batches and features.

The middle two elements are the strides for height and width respectively.

To make life easier, the code is using `tf.nn.bias_add()` to add the bias. Using `tf.add()` doesn't work when the tensors aren't the same shape.

3.2.3 Improvements

Pooling

By now, the way we reduce the feature map size. But this is a very aggressive way to do it because we lose a lot of information. We can run with a very small stride (of 1) but then took all the convolutions in a neighborhood and combine them.

The most common way to go about **pooling** is max pooling:
$y = max(X_i)$
Characteristics of using max pooling:

- Parameter free
- Often more accurate
- more expensive (¿?) But why? If we just said that it was parameter free.
- more Hyper parameters.
  - Pooling Size
  - Pooling Stride

Conceptually, the benefit of the max pooling operation is to reduce the size of the input, and allow the neural network to focus on only the most important elements. **Max Pooling** does this by only retaining the maximum value for each filtered area, and removing the remaining values.

```
# Output depth
k_output = 64

# Image Properties
image_width = 10
image_height = 10
color_channels = 3

# Convolution filter
filter_size_width = 5
filter_size_height = 5

# Input/Image
input = tf.placeholder(
    tf.float32,
    shape=[None, image_height, image_width, color_channels])

# Weight and bias
weight = tf.Variable(tf.truncated_normal(
    [filter_size_height, filter_size_width, color_channels, k_output]))
bias = tf.Variable(tf.zeros(k_output))

# Apply Convolution
conv_layer = tf.nn.conv2d(input, weight, strides=[1, 2, 2, 1],
padding='SAME')
# Add bias
conv_layer = tf.nn.bias_add(conv_layer, bias)
# Apply activation function
conv_layer = tf.nn.relu(conv_layer)
```

Listing 3: Convolution Layer Code

Figure 6: ConvNet Architecture

Another form of pooling is average pooling: $y = mean(X_i)$

It's a little more like providing a blur low resolution view of the feature map.

The next Code 4 we can see an implementation of `max pooling`. 2x2 filters with a stride of 2x2 are common in practice for max pooling.

The `ksize` and `strides` parameters are structured as 4-element lists. `[batch, height, width, channels]`.

For both `ksize` and `strides`, the batch and channel dimensions are typically set to `1`.

Pooling Layers are generally used to: decrease the size of the output

```
conv_layer = tf.nn.conv2d(input, weight, strides=[1, 2, 2, 1],
padding='SAME')
conv_layer = tf.nn.bias_add(conv_layer, bias)
conv_layer = tf.nn.relu(conv_layer)
# Apply Max Pooling
conv_layer = tf.nn.max_pool(
    conv_layer,
    ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1],
    padding='SAME')
```

Listing 4: Max Pooling

and prevent overfitting.

Reducing overfitting is a consequence of the reducing the output size, which in turn, reduces the number of parameters in future layers.

Recently, **pooling layers** have fallen out of favor. Some reasons are:

- Recent datasets are so big and complex we're more concerned about underfitting.
- Dropout is a much better regularizer.
- Pooling results in a loss of information. Think about the max pooling operation as an example. We only keep the largest of n numbers, thereby disregarding n-1 numbers completely.

To calculate the numbers given the max pooling layers, say we have this:

- We have an input of shape 4x4x5
- Filter of shape 2x2
- A stride of 2 for both the height and width

Note: For a pooling layer, the output depth is the same as the input depth. Additionally, the pooling operation is applied individually for each depth slice.

The shape of the output is. 2x2x5.

1x1 Convolutions

The classic convolution it's basically a small classifier for a patch of the image but it's only a **linear classifier** But if you add a 1x1 convolution in the middle, you now have a mini neural network running over the patch instead of a linear classifier.

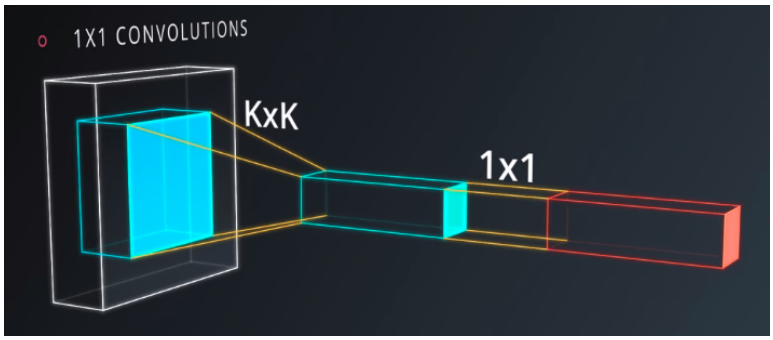

Figure 7: 1x1 Convolution

Interspersing your convolutions with 1x1 convolutions is a very inexpensive way to make your models deeper and have more parameters without

completely changing the structure. They're also very cheap (they are not convolutions, they are just matrix multiplies and they have relatively few parameters.)

Inception Module

The idea is that, at each layer of your ConvNet, usually you need to pick between different options: pooling, 1x1 convolution, 5x5 convolution, etc.

Instead of that, let's just make a composition of all these choices. This is represented in Figure 8.

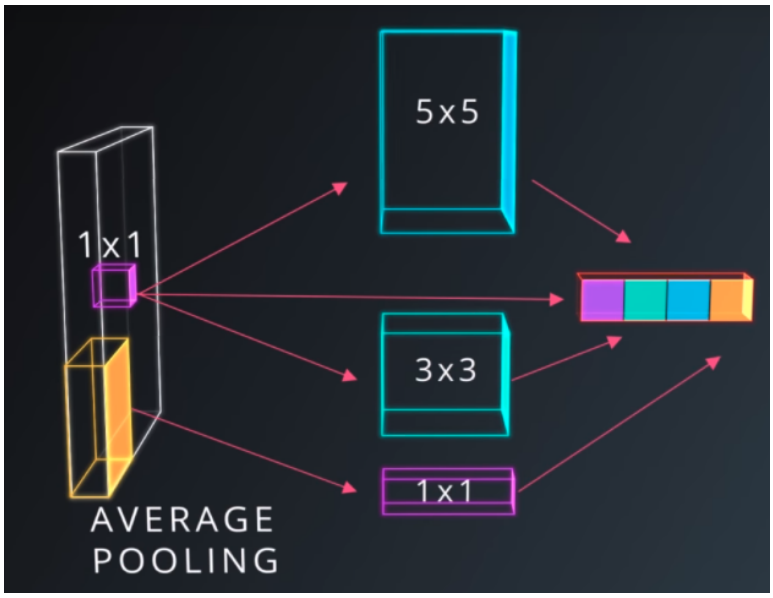

Figure 8: Inception

At the top, you simply concatenate the output of each operator.

You can choose this parameters in such a way that the total number of parameters in your model is very small, yet the model performs better than if you had a simple convolution.