Deep Learning
by DeepLearning.AI

Lucas Paiolla Forastiere

February 3, 2022

Contents

Chapter 1

Introduction

The term deep learning refers to training *neural networks*, sometimes very big neural networks. But what are neural networks?

So let's suppose we want to predict housing prices based on the size of the house. And let's say we'll use Logistic Regression to do that. But as we know, house prices can't be negative, so we simply say the value of the house is 0 if the Logistic Regression would predict something negative.

That's indeed the simplest neural network we can have, we have a single input `size` and a single output `price` and in the middle we have a single neuron: the logistic regression.

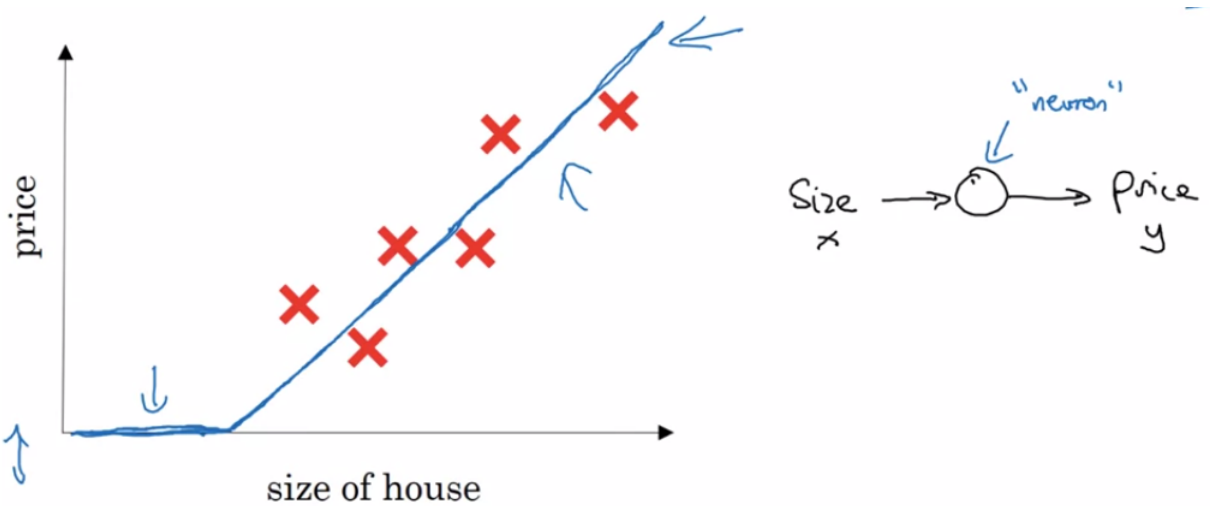


Figure 1.1: Here we see the graph of the problem we described.

That function which is zero and than linear is called *ReLU* and it's used a lot in neural networks. It stands for *Rectified Linear Unit.*

So to get a bigger neural network, we stack these neurons. Instead of predicting using only the size of house, we could use the number of bedrooms, zip code and wealth. We could use the size and number of bedrooms to predict the family size; use the zip code to predict the walkability; and use the zip code and wealth to predict the school quality. And then, we could use the family size, walkability and school quality to predict the price. See in the picture:

However, in general what we have is something a little more complex than that. We would have something like figure 1.3. Here we see that the internal nodes (which are called **hidden nodes** or **hidden neurons** or **hidden units**) receive the output of all the

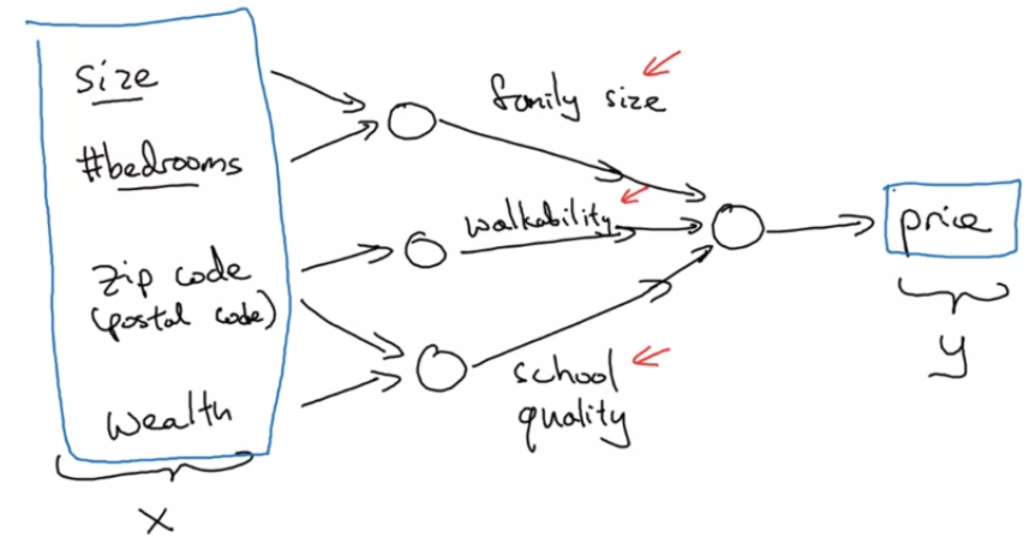


Figure 1.2: Now we have a more complex neural network, which is the stack of many ReLUs.

previous nodes to make it predictions. These hidden nodes don't really have a meaning like the example we gave. We don't try to predict family size or walkability or whatever, we simply let the neural network decide what that neuron will output in order to predict the final output `price` in the better way it can.

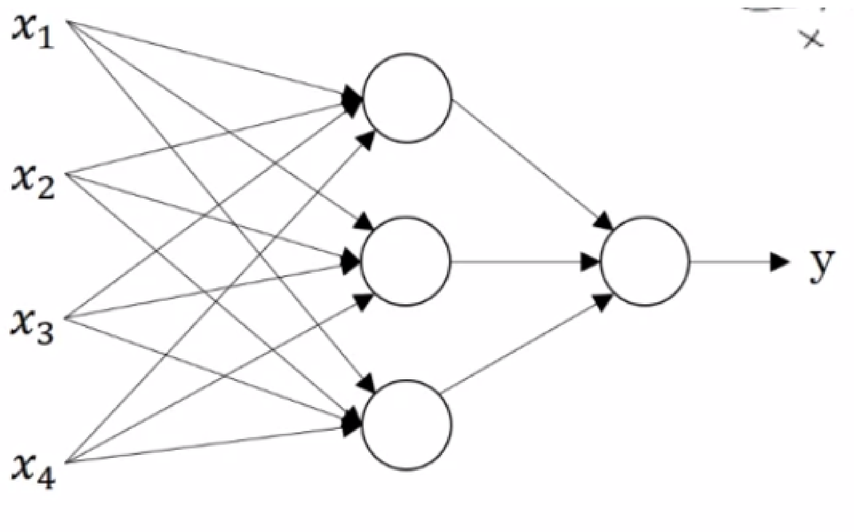


Figure 1.3: The generic form of a neural network.

We can use neural networks in many applications, here we're going to foucos in **supervised learning**, which are problems that you have a set of variables called input (represented by x) and an output (y) related to that input. In order to solve these kind of problems, there are many kinds of neural networks. The one we saw is the most common one, but there are others, like convolutional nn or recurrent nn.

Another thing that's important to decide what kind of nn we'll use is knowing if the data we're leading with is *structured* or *unstructured*.

Structured data is data in the form of a table. We have a very clear set of input variable X and a set of output variables y. Each line of our table represents one instance

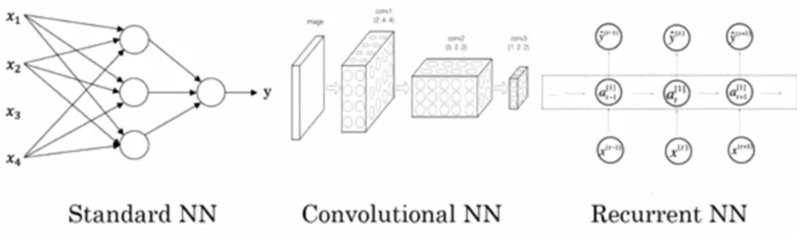


Figure 1.4: Examples of neural networks.

of data with many inputs and one or more outputs related to those inputs.

Unstructured data is all the other kinds of data: audio, video, texts, images, etc.

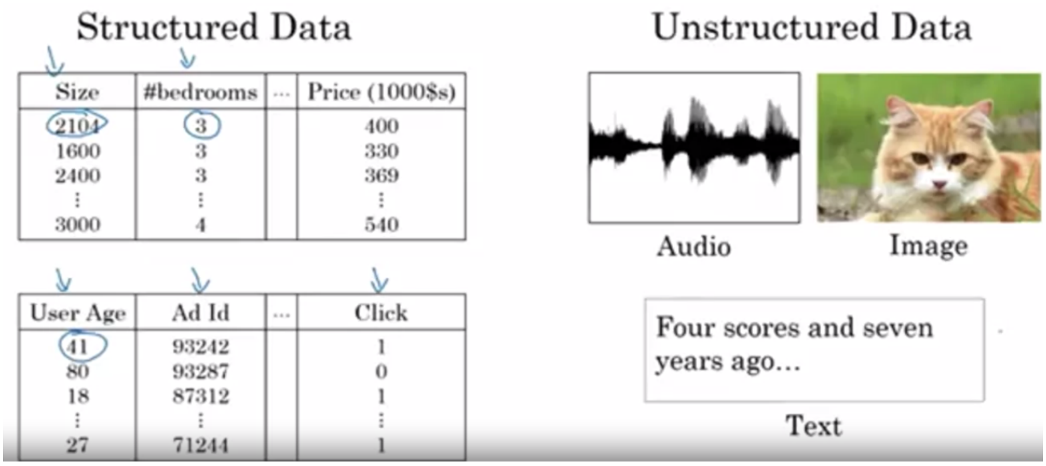


Figure 1.5: The two kinds of data.

It turns out that machine learning algorithms performed better on structured data over the years and more recently neural networks are performing better also on unstructured data.

Why is Deep Learning taking off? This is one of the questions we must ask ourselves when begining to learn deep learning. Let's see the graph of the performance of the machine learning algorithms versus the amount of data that we provide to then. We see that traditional learning algorithms have a plato where they can't improve anymore, which neural networks can lead with that data as we make than bigger and bigger.

We also see in the graph that when we don't have a large amount of data, NNs all algorithms perform pretty much the same.

So in order to answer our question, we have to understand the evolution of three things: *data*, *computation* and *algorithms*.

Through the years, the amount of data available was inscreased a lot, so NNs can take advantage from that. Also the computation power was inscreased with the use of GPUs to make a large amount of computations. And finally new algorithms have been developed to make NNs faster. That's the main reason why deep learning is taking off.

1.1 Notation

Before continuing, we need to define the notation we're going to use.

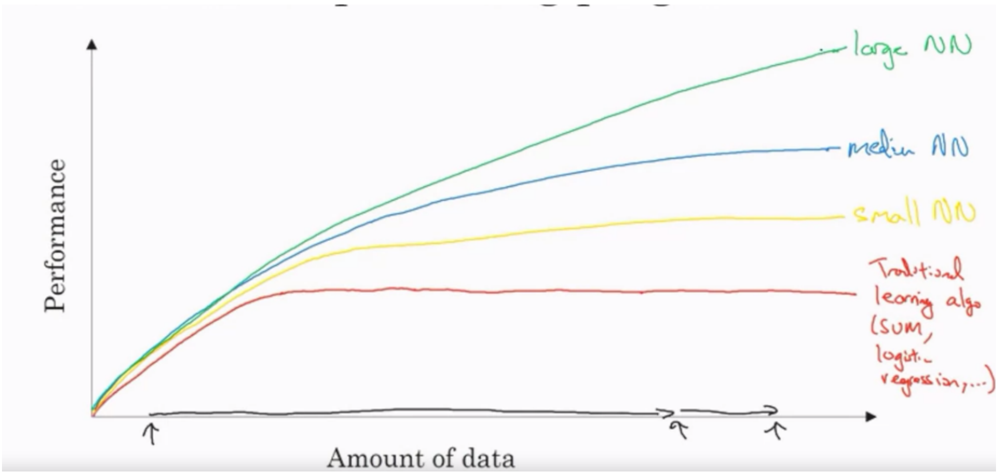


Figure 1.6: The performance of machine learning algorithms in respect to the data we provide to them.

- (x, y) will denote a single training input;
- m or m_{train} denotes the number of training examples;
- $x^{(i)}$ denotes the i-th training input;
- $y^{(i)}$ denotes the i-th traning output;
- n_x or n denotes the number of dimensions x has (or the number of features);
- m_{test} denotes the number of testing examples;
- X is the matrix of all traning examples. It's defined as:

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

  X is an $m \times n$ matrix;

- Y is the matrix of all outputs. It's defined as:

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

  Y is a $1 \times m$ matrix.

Observation. In other courses we might see X defined as the transpouse of the matrix we've just defined. But it turns out that when using this definiting, it's much easier to implement algorithms, so remember te definition we're going to use through out the course.

The same thing for Y. We see that here Y is the transpouse of that it's tends to be in other courses.

1.2 Logistic Regression as a Neural Network

To end this introduction, we'll see the basics of neural network programming using the simplest NN we can: a logistic regression.

So let's recall what's logistic regression and why it's useful. Logistic Regression is used in binary classification, the kind of problem where we have an input and want to predict between 0 or 1. An example could be an image and we want to say it what's a cat (1) or not (0).

Basicly we want an algorhthm to estimate the probability of $y = 1$ given x. In math we write:

$$\hat{y} = P(y = 1 \,|\, x), \qquad x \in \mathbb{R}^n$$

Logistic Regression estimates this quantity using the formula:

$$\hat{y} = \sigma(w^T x + b),$$

where w and b are parameters to be discovered and σ is the **sigmoid function**:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

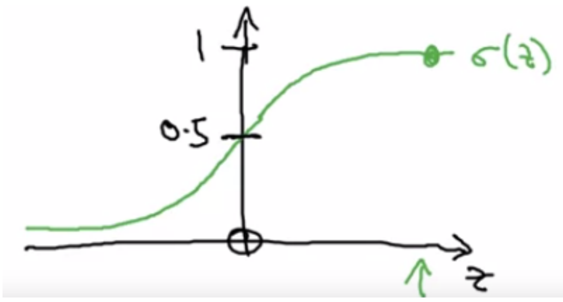


Figure 1.7: A sigmoid graph.

It's also common to create a new input $x_0 = 1$ and use the x vector as $x \in \mathbb{R}^{n+1}$ and use the formula $\hat{y} = \sigma(\theta^T x)$, where

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \qquad \theta_0 = b \qquad w = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

To find the parameters b and w, we need to define a **cost function**, which is a function that says how badly our algorithm is performing. This is a function that we want to minimize and when we minimize, we find the best values of b and w.

The **cost** function is a function of all training examples, while a **loss function** or **error function** is a function of a single traning example that measures how well our algorithm is performing.

For logistic regression, we use the loss function:

$$\mathcal{L}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Notice that this is the same as:

$$\mathcal{L}(\hat{y}, y) = \begin{cases} -\log(\hat{y} - 1), & \text{if } y = 0 \\ -\log \hat{y}, & \text{if } y = 1 \end{cases}$$

That give us the cost function:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

1.2.1 Gradient Descent

We know have:

- A way of predicting the classes 0 or 1 using the sigmoid function;
- A way of measuring the error of our predictions.

What we need now is a way of chaning our parameters b and w in order to minimize the error. That's what the **gradient descent** algorithm does.

Let's first see a general graph of the cost function. In general, it looks like figure 1.8

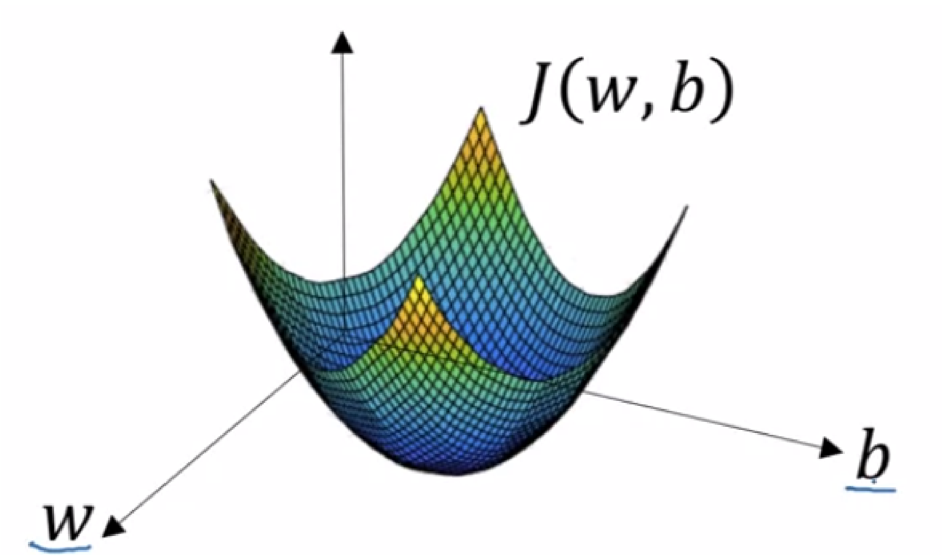


Figure 1.8: A generic graph of the cost function.

We see that J is what we call a **convex function**, which means that it is a function that was only one **local minimum** (or local maxima). This property is very important if we want to apply the gradient descent algorithm.

In Gradient Descent, we initialize b and w randomly and take steps into the direction that leads us to the lowest possible value of J. In order to do that, we calculate the **gradient** (the derivatives in each direction) of the function J and take a step in the opposite direction of the gradient.

Proposition 1.1

The gradient gives us the direction of the maximum increase of the a function.

Algorithm 1.1: Gradient Descent

Repeat {

$$w := w - \alpha \frac{\partial w}{\partial J(w,b)}$$

$$b := b - \alpha \frac{\partial b}{\partial J(w,b)}$$

}

In the algorithm, α is what we call the **learning rate**. It's how large we should step in the direction of the maximum decrease. If we take big steps, we can go much faster to the global minumum, but we might not be so accurated. On the other hand, we be take small steps, we can find the global minum accuratedly, but achieve it much slower.

Gradient Descent is a general optimization algorithm and can be applied to any convex function. So now we need to understand how to use it with logistic regression.

After calculating the derivatives, we'll have:

Algorithm 1.2: Gradient Descent for Logistic Regression

Repeat {

$J = 0;\ dw = 0;\ db = 0$

For $i = 1 \cdots m$:

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J+ = -\left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})\right]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw+ = x^{(i)} dz^{(i)}$

$db+ = dz^{(i)}$

$J/ = m$

$dw/ = m;\ db/ = m$

$w := w - \alpha dw$

$b := b - \alpha db$

}

This version of the algorithm uses a for loop to compute J, dw and db. But when implementing the code into Python or other language, we always try to **vectorize** the code to make it faster.

Algorithm 1.3: Gradient Descent for Logistic Regression Vectorized

Repeat {

$Z = w^T X + b$

$A = \sigma(Z)$

$dZ = A - Y$

$$dw = \frac{1}{m} X dZ^T$$

$$db = \frac{1}{m} \sum dZ$$

$w := w - \alpha dw$

$b := b - \alpha db$

```
}
```

Chapter 2

Basical Concepts

So let's start introducing some notation.

Whenever we use:

$$z^{[i]}$$

we're talking about the z values of the i-th *layer* of the neural network.

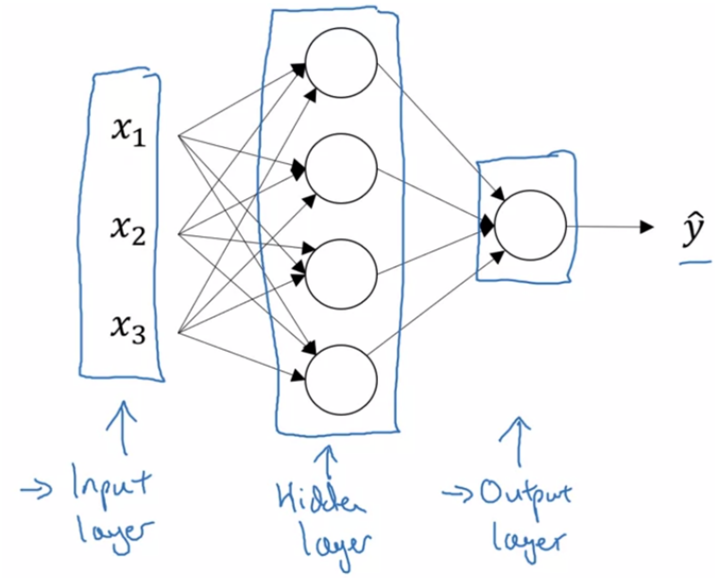
Thus, we have the following important equation:

$$\boxed{z^{[i]} = W^{[i]}a^{[i-1]} + b^{[i]}} \tag{2.1}$$

$$\boxed{a^{[i]} = \sigma(z^{[i]})} \tag{2.2}$$

2.1 Representation

Let's ilustrate a simple neural net:



This nn is divided into three parts:

- The **input layer** is the layer were we enter the data, the features. It's represented by a single vector x, which is a column of our inputs matrix X. Another way to denote the input is by $a^{[0]}$;

- The **output layer** is the layer were we get our answer. It can be one of more nodes and it's represented by a single vector y. It can be a binary vector, continuous vector, it could be even a codification of an image, sound, word or any codification just like the input;

- Finally, the **hidden layer** are all the nodes which are between the input and output, they are the intermediately computations that our neural net does. We don't need to stick to just one hidden layer, there could be many of them.

One more thing to keep in mind is that the neural network drawn in the picture is called a *two layer* neural network, because we don't count the input layer as a "real" layer (it's the layer zero).

Each layer (real layer) will have **parameters** associated with them, which we'll denote $W^{[i]}$ and $b^{[i]}$ for the i-th layer.

To understand what are node computes given an input, let's focus in the first node of the hidden layer in the figure. We can see that the inputs x_1, x_2, x_3 are all passed to that node, which uses tham to output something to the node in the output layer (or it could be outputed to the next hidden layer).

So basicaly a node is a logistic regression, it computes:

$$a = \sigma(Wx + b)$$

But as we have many computations like this, we have to use indeces to denote the first hidden layer and the first node of that layer. So the computations would be described as following:

$$z_1^{[1]} = W_1^{[1]T}x + b_1^{[1]}$$
$$a_1^{[1]} = \sigma(z_1^{[1]})$$

The superscript means we're talking about the first (real) layer and the subscript means we're talking about the first node of that layer. Therefore $a_i^{[l]}$ is the **activation value** (output value) of the i-th node in the l-th layer.

Of course, when coding a neural network, we don't compute each one of these equations using a for loop, we vectorize in order to compute the whole vector $z^{[l]}$ at once using equation 2.1.

Vectorizing the input Now, one more thing that we want is to be able to predict *mulpible* inputs at once. Indeed we can do that with some vectorization. Let's see how.

So let's recall that

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

and let's define a matrix $Z^{[l]}$ in which the j-th column is the vector $z^{[l](j)}$ (i.e., the value before the activation function of the layer l when applied to the input j).

$$Z^{[l]} = \begin{bmatrix} | & | & & | \\ z^{[l](1)} & z^{[l](2)} & \cdots & z^{[l](m)} \\ | & | & & | \end{bmatrix}$$

and also define the matrix $A^{[l]}$, which is $\sigma(Z^{[l]})$:

$$A^{[l]} = \begin{bmatrix} | & | & & | \\ a^{[l](1)} & a^{[l](2)} & \cdots & a^{[l](m)} \\ | & | & & | \end{bmatrix}$$

We can see that each row of matrix A tell us the activation value for a particular node of that layer for each input. For instance, the value $A^{[l]}_{i,j}$ is the value of the i-th neuron in the l-th layer of the neural network when we input the j-th input.

Given these matrices, we can train over all inputs using the vectorized formula:

$$\boxed{Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}} \tag{2.3}$$

$$\boxed{A^{[l]} = \sigma(Z^{[l]})} \tag{2.4}$$

and recall that $X = A^{[0]}$.

2.2 Activation Functions

The **activation function** is the function g applied to z. We've been currently using the *sigmoid function* σ, but there are some other functions that can be used and can significatly change our results and performances.

Another function one can use is the tanh function, given by the formula:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

which is a shifted version of the sigmoid function. We have $\tanh(0) = 0$ instead of $\sigma(0) = 0.5$.

It turns out that the tanh function frequently works better than the sigmoid, because the values are between -1 and 1 instead of 0 and 1 and thus the average values tend to be closer to zero.

The exception is for the output layer, where it's good to have a number between 0 and 1 in many cases, so we could keep using the sigmoid function there and use the tanh on the hidden layers.

One of the down sides of these two functions is that when z is very large, the gradient is very small, because they're very flat for large (positive or negative) values of z.

That's why many times we see the **ReLU** (rectified linear unit) function being used:

$$ReLU(z) = \max(0, z)$$

Indeed the ReLU is the most used activation in practice and should be the first choise. One of the only down sides of it is that the derivative for values lower than zero is zero, but it works fine in practice.

There's modified version of the ReLU called **Leaky ReLU**, which is given by:

$$LReLU(z) = \max(0.01z, z)$$

It's just like the ReLU function, but it's not zero for any negative value of z and, thus, the derivative is not null on those points. Actually, indeed there no reason for the 0.01, it could be any small value and we can turn that into a hyperparameter of our algorithm.

Why activation functions Now that we've seen so many activation functions we might want to understand why do we even need them. And most importantly, why do we need they to be non-linear.

We could try using $a = g(z) = z$ (i.e., doing nothing). It turns out that if we have just the identity function, we can't express *complex* (non-linear) decision boundaries.

We'll not prove it here, but if we just use a linear function in all hidden nodes and a sigmoid, it is equivalent to a standard logistic regression in terns of what it can express.

$$a^{[1]} = w^{[1]}x + b^{[1]}$$

$$\begin{aligned}
a^{[2]} &= w^{[2]}a^{[1]} + b^{[2]} \\
&= w^{[2]}\left(a^{[1]} = w^{[1]}x + b^{[1]}\right) + b^{[2]} \\
&= \left(w^{[2]}w^{[1]}\right)x + \left(w^{[2]}b^{[1]} + b^{[2]}\right) = \\
&= w'x + b'
\end{aligned}$$

Above we can see that from one layer to the next we still have a linear equation (a composition of two linear functions is a linear function).

2.3 Backpropagation

Now we're going to understand how to find the optimal values for W and b using back-propagation.

The first step in this direction is calculating the derivative of the activation functions. So let's to it.

Sigmoid

$$g(z) = \frac{1}{1+e^{-z}}$$

$$\begin{aligned}
\frac{dg(z)}{dz} &= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) \\
&= g(z)(1-g(z))
\end{aligned}$$

Tanh

$$g(z) = \tanh(z)$$

$$\begin{aligned}
\frac{dg(z)}{dz} &= 1 - (\tanh(z))^2 \\
&= 1 - g(z)^2
\end{aligned}$$

ReLU

$$g(z) = \max(0, z)$$

$$\frac{dg(z)}{dz} = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z > 0 \\ \text{undef}, & \text{if } z = 0 \end{cases}$$

Leaky ReLU

$$g(z) = \max(0.01z, z)$$

$$\frac{dg(z)}{dz} = \begin{cases} 0.01, & \text{if } z < 0 \\ 1, & \text{if } z > 0 \\ \text{undef}, & \text{if } z = 0 \end{cases}$$

Now let's understand how to do **gradient descent**.

Let's denote L the number of layers of our network. Them, the last value derivative (the first we're going to compute) is given by the formula:

$$dz^{[L]} = a^{[L]} - y \tag{2.5}$$

The values of dW and db, for any layer are given by the formulas:

$$dW^{[l]} = dz^{[l]} a^{[l-1]T} \tag{2.6}$$

$$db^{[l]} = dz^{[l]} \tag{2.7}$$

The other values of dz for any layer other than the output layer are given by the formulas:

$$dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}\left(z^{[l]}\right) \tag{2.8}$$

where $*$ denotes que *element-wise* product.

There's also a vectorized way of computing this for all the training examples at once.

$$dZ^{[L]} = A^{[L]} - Y \tag{2.9}$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{2.10}$$

$$db^{[l]} = \frac{1}{m} \text{sum}\left(dZ^{[l]}, \text{axis=1}\right) \tag{2.11}$$

$$dZ^{[l]} = W^{[l+1]T} dZ^{[l+1]} * g^{[l]\prime}\left(Z^{[l]}\right) \tag{2.12}$$

2.4 Random Initialization

Finanlly, we need to initialize the parameters of the neural network. It turns out that initializing everything as zero or everything as the same number, then all nodes of a layer will be exactly the same even with many iterations of gradient descent.

For the values of b, we can initialize with zero, but for W, we prefer to initialize with *small* random numbers. We like the numbers to be small though because if we're using tanh or σ, big numbers will have a very plat derivative and, thus, the algorithm will learn slower.

2.5 Notation for deep neural networks

We'll use L to describe the number of layers the neural network have (remember, the input layer is not a real layer).

$n^{[l]}$ denotes the number of nodes on layer l.
$a^{[l]}$ denotes the activation values on layer l.
$g^{[l]}$ denotes the activation function used on layer l.
Also, the dimensions of $W^{[l]}$ and $b^{[l]}$ are $\left(n^{[l]},\ n^{[l-1]}\right)$ and $\left(n^{[l]},\ 1\right)$.

Chapter 3

Practical Aspects of Deep Learning

In this chapter we're going to focus on the practical aspects of deep learning, like how to choose the hyperparameters, how to make the code faster, how to apply regularization and others.

The first thing to understand is that, in practical, we divide our data set into three:

- *Training set*, in which we're going to train the model;
- *Hold-out cross validation set (or dev set)*, in which we're going to evaluate our model and choose hyperparameters;
- *Test set*, in which we're going to evaluate our final model, with the hyperparameters setted.

In the previous era of Machine Learning, we would use 70% for the training/dev sets and 30% for testing, or 60/20/20%. But now in the big data era, we can use much small fractions for the testing and hold-out sets (like 1% or even less, since this can be equivalent to about 10000 examples).

Another important thing to keep in mind is that the training set and test set must have the *same distribution* of that. For instance, if we train on images of cats coming from the web, than we should not test on images of cats coming from mobile cameras.

3.1 Regularization

We've seen that the cost function of our neural network is a function J such that:

$$J\left(W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y^{(i)}}, y^{(i)})$$

Regularizing a neural net model means adding a penality when the model makes $W^{[i]}$ bigger. This helps the model to prevent overfitting.

Thus, we have:

$$J\left(W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y^{(i)}}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|W^{[l]}\right\|^2$$

Where

$$\left\|W^{[l]}\right\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} \left(W_{i,j}^{[l]}\right)^2$$

is called the *Frobenius norm* (sometimes we write $\left\|W^{[l]}\right\|_F^2$).

Here λ is called the **regularization factor** and the bigger it is, the more we penalise the model for having bigger $W^{[i]}$.

Another kind of regularization is called **dropout regularization**. In this kind of regularization, we set a small probability of removing a node at all and have a new smaller network.

One way to implement it is called *inverted dropout*.

```
l = 3
keep-prob = 0.8
d3 = np.random.rand(a3.shape[0]. a3.shape[1]) < keep-prob
a3 = np.multiply(a3, d3)
a3 /= keep-prob
```

The last line is used to keep the espected value of a the same, so we keep the same scale of z when doing $z = wa + b$.

When using dropout, though, we *don't apply it when using the neural net to predict values*.

To explain intuitively why dropout works, we need to think about the nodes connected to a particular node. If the NN just give importance to one of the those nodes, then it might be dropped and the model will perform poorly. Therefore, to work well, the model needs to spread the weights into all the neurons, shirking them.

There are other ways to perform regularization.

Data augmentation is when we change the data to get more data. For instance, if we have a image dataset, we can flip the images, reduce saturation, blur the image, and do many other things to get more images to train the make our mode better.

If we have a sound dataset, we can make the sound loud, or add noise, supression and so on.

Early stopping is when we stop iterating our model using the error in the dev set. While the error in the train error one decreases, the error in the dev set tends to decrease for a while and than starts incrasing. What early stopping does is stop iterating the model when the error in the dev set starts increasing.

3.2 Optimizing the Problem

3.2.1 Normalizing input

Normalizing the input is very important because it garantees to us that the input data will have zero mean and variance one. To perform normalization, we calculate:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

and

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$$

where $x^{(i)2}$ mean elementwise power.

Please note that these values are calculated using the *training set*. And them we'll apply the below formula to *every* input that we'll feed into the network (it can come from the training, dev or test set).

$$x := \frac{x-\mu}{\sigma}$$

Normalizing the input is important because it garantees that Gradient Descent will converge much faster, since J will have a more circular bow shape, since of a elliptical one.

3.2.2 Vanishing / Exploding Gradients

In this section we're going to cover a problem that we can face when traning neural networks. Sometimes, the gradients become too small or too big and we can't training anything at all.

To understand that, suppose we have a very deep neural network. It's not so hard to imagine that as computations are performed upon computations, we can get very large or very small numbers for values like the activation function.

For instance, suppose all $W^{[l]}$ have values of 1.5 then if we don't use an activation function, our final output will be something proportional to $W^{[1]}W^{[2]}\cdots W^{[L]}$, which will me proportional to 1.5^L. If L is very big, this can be huge.

If we changed 1.5 by 0.5, then we would have amoust zero.

To solve (parcially) this problem, we need to have a careful initialization of our weights.

Let's just consider a single neuron. If we ignore b, then we have

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

The later n is, the smaller we want w_i to be, because we want z to be approximately in the range $[-1,1]$. One thing that we would wish is to have

$$\mathrm{Var}\,(w_i) = \frac{1}{n}$$

So we can initialize it like:

$$W^{[i]} = \text{np.random.randn(shape)} \cdot \text{np.sqrt}\left(\frac{1}{n^{[l-1]}}\right)$$

It turns out that if we are using ReLU, it's better to have a variance of $\frac{2}{n}$, so we can replace the one by 2 in the sqrt.

The variatian with 1 is called the *Xavier initialization*, but we can use another variatian in which we multiply by:

$$\sqrt{\frac{2}{n^{[l-1]}+n^{[l]}}}$$

These factors could all be used as a base model, but we can really tune this as a hyperparameter if we want, although it's not a very important hyperparameter.

3.2.3 Gradient Checking

A very common technique when using Gradient Descent is to use **Gradient Checking** to make sure our gradient computations are been performed right. It will not go to the final model, but we use it to make sure we've coded everything right.

To do it, basicaly we approximate the gradients numericaly and compare with the gradient we're computing using the formulas. Of course we don't do that when we want to predict any value or use our model in the real world because we're computing the gradient two times (and computing it numericaly is not efficient). We just use this technique when coding the backpropagation to make sure we're doing it right.

To approximate the gradient, it's very since, we just use:

$$g(\theta) \approx \frac{f(\theta + \varepsilon) - f(\theta - \varepsilon)}{2\varepsilon}$$

which is called *two-sided approximation*.

So the first thing in order to use gradient checking is to reshape our parameters $W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}$ into just one big vector θ. And it's derivative will be $d\theta$

Now we're going to compute $d\theta_{\approx}$ which can be computed doing:

$$d\theta_{\approx i} = \frac{J(\theta_1, \theta_2, \cdots, \theta_i + \varepsilon, \cdots, \theta_k) - J(\theta_1, \theta_2, \cdots, \theta_i - \varepsilon, \cdots, \theta_k)}{2\varepsilon}$$

For each value of i.

And what we want is

$$d\theta_{\approx} \approx d\theta$$

To now that, we can calculate:

$$\frac{\|d\theta_{\approx} - d\theta\|}{\|d\theta_{\approx}\| + \|d\theta\|} < \delta$$

for a small value of δ. It could be 10^{-7} to make sure it's really correct or 10^{-5}. If the difference is greater than something like 10^{-3} than probability we have a bug somewhere.

3.3 Optimization Algorithms

In this section, we're going to view many algorithms that allow us to train much faster.

3.3.1 Mini-batch Gradient Descent

What we've been doing in terms of gradient descent so far is to process the whole training set to take a step in the opposite direction of the gradient.

If m, our number of training exemples, is very large (which is normal in Big Data applications), then a single gradient step is very expensive.

One way to optimize that is to use what's called **mini-batches**. Basically we split our training set into multiple training sets (if say 1000 examples) and we run each step of gradient descent using one of these sets.

That's what we call an **epoch** of gradient descent (a pass of forward-prop, cost computation, back-prop and parameters update).

Of course, by doing this, we're not optimizing the original J function (which is computed for every training example). However, we'll get a low cost model at the end.

If we plot the cost of J at each iteration (epoch), we'll not have that classical curve in which J decreases at each iteration. The curve will be a little noisy, increasing and decreasing. But in the long run, it'll converge to some low value. And not just that, it does it much faster than the original gradient descent (which is also called **batch gradient descent**).

One important concept to mention is the size of the mini-batches. If the size is one, than we call the algorithm **stochastic gradient descent**. This will make our epochs really fast, but we'll lead to an algorithm with a much higher cost than we could get using greater batches. Stochastic gradient descent will be *around* the minimum optima of J. The greater our mini-batches are, the more closer the that mininum optima we'll lead. That's the trade-off between speed and precision.

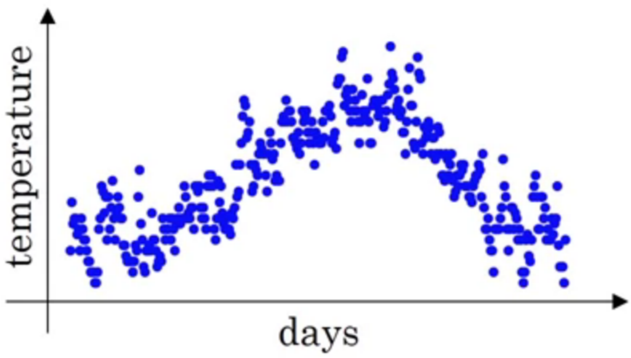
We can also train with mini-batch until the cost is very close to the mininum optima and then change to batch gradient descent and run some more epochs to make sure that our algorithm is really precise.

In practical, because how the computer memory and vectorization works, people tend to use powers of two for the size of the mini-batches. So that's a thing to keep in mind. Common values are 2^6 up to 2^9.

3.3.2 Exponentially weighted averages

Now we're going to cover other optimization algorithms that are better than gradient descent. But to understand them, we need to talk about a concept of statistics which is called **exponentially weighted (moving) averages**.

To understand that, let's use an example. We'll plot the temperatures over the days.



Here we sse that the values are very noisy, going up and down on each day. However, if we want to visualize the *trend* of the values, we can apply the moving average technique.

Basicaly, instead of plotting these points (let's call them θ_i), we'll plot V_i.

First, we initialize V_0 as zero.

$$V_0 = 0$$

then, we get the next value by averaging the previous value of V_i and the current value of θ_i.

$$V_1 = 0.9V_0 + 0.1\theta_1$$

a general formula would be:

$$\boxed{V_i = 0.9V_{i-1} + 0.1\theta_i}$$

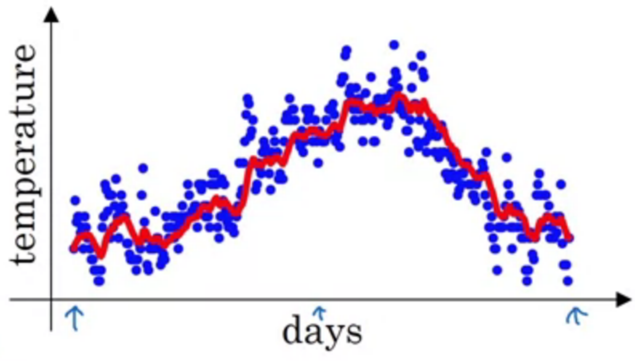


Figure 3.1: The moving average of the temperatures.

If we compute and plot that, we'll get figure 3.1

We see that we got a much more smooth curve because we're always considering the previous value to create the next one. We could make the formula even more general:

$$\boxed{V_i = \beta V_{i-1} + (1-\beta)\theta_i}$$

where β is the percentage of how much we would like to consider the previous value.

Indeed the value of β will dictate approximately how many previous iterations we're considering on each point. To have an idea of how many previous points we're considering, we use the formula

$$\frac{1}{1-\beta}$$

Therefore, for $\beta = 0.9$, we're considering approximately the 10 previous temperatures in order to get the current value. That's why it's called a *moving* average or a *local* average.

The lower β is, the less temperatures we'll be considering on each average and, therefore, the faster those averages will change, leading to a noiser curve, but that adapts faster. And if β is big, we'll be considering more days and, therefore, the average will change more slowly, leading to a smoother curve that has some latency to adapt to new trends.

This value will be a hyperparameter in our models.

To understand a little better this algorithm, let's try to get a non-iterative formula. We'll get something like:

$$V_i = 0.1\theta_i + 0.1 \times 0.9\theta_{i-1} + 0.1 \times 0.9^2\theta_{i-2} + \cdots + 0.1 \times 0.9^k\theta_{i-k} + \cdots$$

So, indeed, we're calculating V_i as an average of *all* previous values. But the farest the values are from the current we're calculating, the less impact it have on the current one. Indeed, we see that the influence decays *exponentially* (that's why this term apears in the name).

If we sum all coeficients, we would except to get exactly 1, since this is an average. But unfortunatelly, that's not what happens. In fact, we say that this is approximately an average, because we're missing an **bias term**.

However, using this kind of average is very good computacionally, because we don't have to keep track of (say) the last n terms to compute a local average. We can do that in $O(1)$ for each element.

If we would implement this algorithm, it would be like:

Algorithm 3.1: Exponentially weighted averages

$V_\theta = 0$
Repeat {
 $V_\theta := \beta V_\theta + (1-\beta)\theta_t$
}

As one can see, this algorithm is very memory and time efficient.
Finally, we can make a correction to the bias term by dividing V_i by $1-\beta^i$.
So we can use the formula:

$$V_i = \frac{\beta V_{i-1} + (1-\beta)\theta_i}{1-\beta^i}$$

This will help a lot in the first iterations where we still don't have many values in the window. Think of it as $1-\beta^i$ being a good approximation for the sum of the coefficient of the average.

3.3.3 Gradient Descent with Momentum

The first algorithm we'll see is **gradient descent with momentum**. Basicaly, instead of using the derivatives in each iteration, we're going to average the derivatives using exponentially weighted averages to get a more smooth descent to the minimum. This algorithm works pretty much always better than normal gradient descent.

Algorithm 3.2: Gradient Descent with Momentum

On iteration t:
 Compute dW, db on current batch (or mini-batch)
 $V_{dW} = \beta V_{dW} + (1-\beta)dW$
 $V_{db} = \beta V_{db} + (1-\beta)db$
 $W = W - \alpha V_{dW}$
 $b = b - \alpha V_{db}$

This averaging over the last k derivatives makes our algorithm take this *momentum*, which is going to create speed into the minimum direction, avoiding oscilations in other directions like standart gradient descent does.

The downside of this algorithm is that now we have two hyperparameters instead of just one: α and β. In practice, values close to $\beta = 0.9$ work good.

One more note is that in practice people tend to not use *bias correction* because after just 10 or so iterations the algorithm will have a very small bias value.

3.3.4 RMSprop

RMSprop which means for Root-Mean-Square prop. It also speed gradient descent by "normalizing" the different directions. To do that, it takes the element-wise square of the gradient, average it over the last k squares and than divide the gradient by the squareroot of that.

Algorithm 3.3: RMSprop

On iteration t:

Compute dW, db on current batch (or mini-batch)

$S_{dW} = \beta V_{dW} + (1-\beta)dW^2$

$S_{db} = \beta V_{db} + (1-\beta)db^2$

$W = W - \alpha\dfrac{dW}{\sqrt{S_{dW}+\varepsilon}}$

$b = b - \alpha\dfrac{db}{\sqrt{S_{db}+\varepsilon}}$

The ε value here is just to garantee we're not dividing by zero.

3.3.5 Adam

Adaptative moment estimation optimization algorithm will basicaly take the two previous ideas (momentum and rmsprop) and put them together. It's one of the best algorithms known so far along RMSprop.

Algorithm 3.4: Adam

$V_{dW} := S_{dW} := V_{db} := S_{db} := 0$

On iteration t:

Compute dW, db on current batch (or mini-batch)

$V_{dW} = \beta_1 V_{dW} + (1-\beta_1)dW$

$V_{db} = \beta_1 V_{db} + (1-\beta_1)db$

$S_{dW} = \beta_2 V_{dW} + (1-\beta_2)dW^2$

$S_{db} = \beta_2 V_{db} + (1-\beta_2)db^2$

$V_{dW} = \dfrac{V_{dW}}{1-\beta_1^t}$

$V_{db} = \dfrac{V_{db}}{1-\beta_1^t}$

$S_{dW} = \dfrac{S_{dW}}{1-\beta_2^t}$

$S_{db} = \dfrac{S_{db}}{1-\beta_2^t}$

$W = W - \alpha\dfrac{V_{dW}}{\sqrt{S_{dW}+\varepsilon}}$

$b = b - \alpha\dfrac{V_{db}}{\sqrt{S_{db}+\varepsilon}}$

So as one can see, we're basicaly doing what we're doing in RMSprop, but now instead of using dW, db, we're using the average over the last k values, just like in momentum.

One more thing here is that we're using bias correction, which is very common to do in this algorithm.

Now we have three hyperparameters: α, β_1, β_2. In practice, α still needs to be tuned, and common values for the other two are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. There's also the value of ε, but this is very rare to chance. We usually use something like $\varepsilon = 10^{-8}$.

3.3.6 Learning Rate Decay

One more technique that can be used through all the optimization algorithms is **learning rate decay**. Basicaly, we slowly reduce the α parameter in our algorithm by the time. In the begining, we're far alway from the minimum, so we can set our learning rate higher and as we go towards the minimum, we reduce this learning rate to make it smaller and more precise. One way to do that, is to use the formula:

$$\alpha = \frac{1}{\text{decay\_rate} \cdot \text{num\_epoch}} \cdot \alpha_0$$

Other formulas use exponentially decay:

$$\alpha = 0.95^{\text{num\_epoch}} \cdot \alpha_0$$

$$\alpha = \frac{k}{\sqrt{\text{num\_epoch}}} \cdot \alpha_0$$

3.4 Hyperparameter tuning

We've seem so far that the algorithms have lots of hyperparameters to turn, like the number of layers, number of nodes in each layer, mini-bath sizes, α, β_1, β_2. Also the decision of using one optimization algorithm instead of other could be considered as a hyperparameter.

We need to understand how to select the values of the hyperparameters we're going to try. In the early machine learning times, we used a grid to try the hyperparameters. For instance, if we had h_1 and h_2, than we would say in each values of them we wanted to try. Let's say we want to try $h_1 = \{ v_1, v_2, v_3 \}$ and $h_2 = \{ u_1, u_2, u_3 \}$. Them we would try all the combinations of these values:

$$h_1 \times h_2 = \{ (v_1, u_1), (v_1, u_2), \cdots, (v_3, u_3) \}$$

This is good when the number of parameters is small. But in deep learning, the number of parameters tend to be large and, thus, we use random points (i.e., for each hyperparameter, we pick a random value in some region).

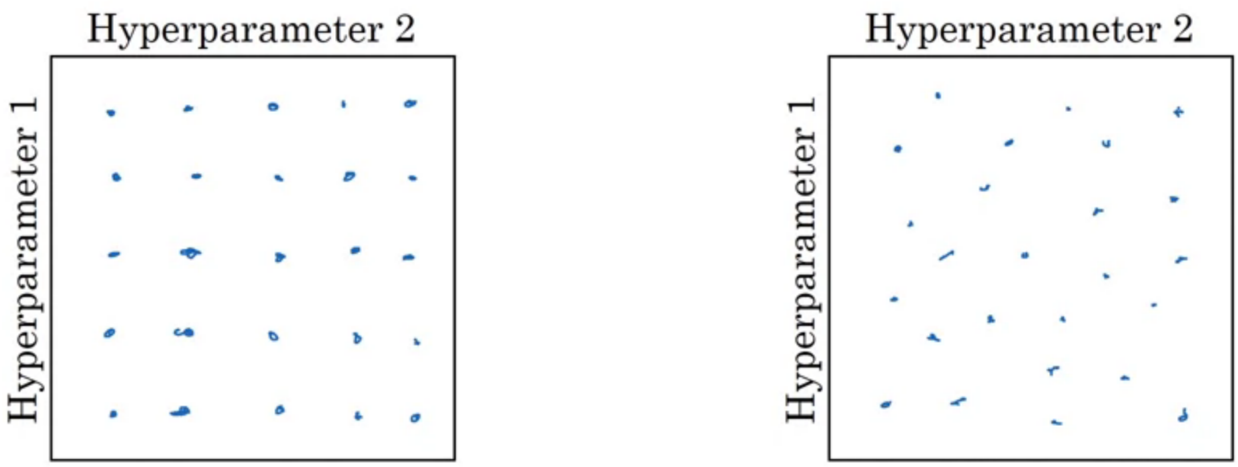


Figure 3.2: In the left we see the Grid Search and in the right, we see the random search

Another good thing to do is to perform many hyperparameter searchs. In the first one, we can use larger scales for each hyperparameter and discover that the best value for, let's say, α is between 0.9 and 1.1. Than we can perform a new search sampling values of α just in that region and discover that the best value is approximately 0.96.

3.4.1 Selection the appropriate scale for hyperparameters

In many cases, we don't want to *uniformily* sample values from a particular range. Suppose we use the range $[0.0001, 1]$ for the learning rate α. Then it's clear that most (90%) of the time we'll select values between 0.1 and 1. That's not what we want. We want, to select values between $(0.0001, 0.001)$, $(0.001, 0.01)$, $(0.01, 0.1)$ and $(0.1, 1)$ in the same proportion.

Therefore, what we need to do is to sample using a *logarithmic* random scale.

To do that in python, one way is the following:

```
r = -4 * np.random.rand()
alpha = 10 ** r
```

Scale for parameter β Another important topic is how to pick a scale for the parameter β. We've seem that $\beta = 0.9$ means we're averaging over the 10 previous gradients and if $\beta = 0.999$, we're averaging over the last 1000 gradients. So we want to have a linear sample *in the number of gradients*. To do that, we can think in $1 - \beta$ and get the interval $(0.1, 0.001)$. And just like in the example from above, we can use a logarithmic scale, but now reversed. Therefore, we use:

$$\beta = 1 - 10^r$$

Instead of the code above.

3.5 Batch normalization

As we've seem before, input normalization can help a lot to optimize our algorithm and train faster. Another technique is called **batch normalization** in which normalize the values in between the hidden layers.

So basicaly for each intermediate value $z^{[l]}$, we normalize it using:

$$\mu = \frac{1}{m} \sum_i z^{[l](i)}$$

$$\sigma^2 = \frac{1}{m} \sum_i \left(z^{[l](i)} - \mu\right)^2$$

$$z^{[l](i)}_{\text{norm}} = \frac{z^{[l](i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

But indeed, we don't want all the intermediate values to be normalized, so we do:

$$\tilde{z}^{[l](i)} = \gamma z^{[l](i)}_{\text{norm}} + \beta$$

where γ and β are learnable parameters of the model (using their gradient as with W and b).

This allows us to change the mean and variance of each layer to our benefit.

Basicaly now, instead of doing this process:

$$X \xrightarrow{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow{g^{[1]}} a^{[1]} \xrightarrow{W^{[2]},b^{[2]}} z^{[2]} \xrightarrow{g^{[2]}} a^{[2]} \to \cdots$$

We're doing:

$$X \xrightarrow{W^{[1]},b^{[1]}} z^{[1]} \xrightarrow{\gamma^{[1]},\beta^{[1]}} \tilde{z}^{[1]} \xrightarrow{g^{[1]}} a^{[1]} \xrightarrow{W^{[2]},b^{[2]}} z^{[2]} \xrightarrow{\gamma^{[2]},\beta^{[2]}} \tilde{z}^{[2]} \xrightarrow{g^{[2]}} a^{[2]} \to \cdots$$

And we can compute the gradient of all parameters to update each of them using Gradient Descent.

Observation. We also need to say that indeed, when using batch normalization, we don't need to use the b parameter at all, because β will always change it before computing a. Therefore, the only parameters we need are W, γ, β.

3.6 Multiclass problems

So far, we've seem only how to do binary classification with neural nets. Now, let's see a logistic regression variatiante that allow us to have multiple classes: **softmax regression**.

Notation 3.1

- C will denote the number of classes;
- The classes will be numbers from 0 to $C-1$.

The first modification we'll do is in the output layer. Now $\hat{y}$ will be a vector instead of a single number and $\hat{y}_i = p(i \mid x)$, i.e., the probability of class i given the input x.

We'll use an **softmax** layer in the output layer to achive such goal. To do so, we'll compute $z^{[L]}$ as usual and use the **softmax activation function**, which is:

$$t = e^{(z^{[L]})}$$
$$\hat{y} = a^{[L]} = \frac{e^{(z^{[L]})}}{\sum_i t_i}$$

As you can see, we basicaly compute this vector t and divide by the sum of its components (it's like we were normalizing t).

Indeed, if we sum the coordinates of $\hat{y}$, they'll add up to 1.

3.7 Deep Learning Frameworks

We've learned so far how to program deep learning algorithms from scratch but there are several python frameworks that are used in practice when implementing these deep learnings algorithms.

Some of the most used DL frameworks are:

- Keras
- TensorFlow
- Torch

Chapter 4

Machine Learning Projects

In this chapter, we're going to focus on how to structure our machine learning projects so that we don't waste a lot of time (say) colleting data when the bottleneck say the number of layers or not regularizing.

4.1 Setting up your goal

One of the first things we need to set up is a **single real value** as a metric goal. By donig this, one can evaluate the model and have a clear goal (improve $x\%$ of that metric).

Also, sometimes we want to restric our model to certain conditions like *running time*, we want it to be low. So we would just really care about our model it is runs in less than (say) 100 ms. This kind of metric is called an **satisficing** metric, while the real value we said before is called an **optimizing** metric.

We want to maximize (or minimize) the optimizing metric under the condition given by the satisficing metric.

All these metrics must be calculated under a training set, or evaluation (dev) set or testing set. So let's talk a little about how to set these sets.

Basicaly, the gold rule for when spliting the sets is to keep the same distribution in each one of them. If we have data of pictures of cats tha come from cellphone, regular cameras and profisional cameras, we want to have the same percentage of each of these types through all sets.

When to change the evaluation metric? Let's suppose that we have two algorithms A and B with 3% and 5% classification errors respectively, but suppose that algorithm A is classifing wrongly photos of pornograph as photos of cats and would show it to the customers, which is a massive error to our company. Therefore, we prefer algorithm B even if it missclassifies more.

When things like this happen, we need to either change our evaluation metric or change our dev and training sets to adapt to the kind of error we're having. One example is weighting the examples to make porn pictures have a very large error while non-porn pictures have a normal error.

4.2 Comparing to humam-performance

One of the things we should aim when using a machine learning system to accomplish a certain goal is to check how well humans perform that goal. By doing this we can have a kind of standard line of what is a good or bad model. A good model is one that does the task better than us humans.

When we look at the graph of accuracy of models over time, we see that it's very easy to surpass the human level of performance and than it starts to slow down the increase in performance until is barly reaches the maximum amount of performance, which is called the **Bayes optimal error**, which is a theorical best possible error.

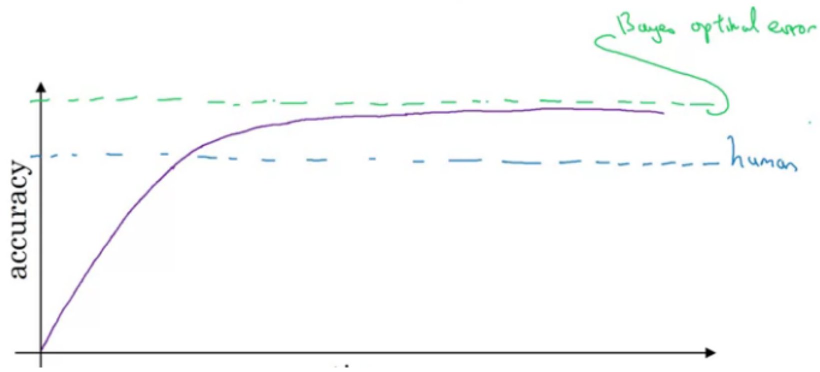


Figure 4.1:

One nice trick that we can use for tasks that humans are good at is to use specific knowladge to see where the model are is getting things wrong and try to get more labeled examples for that situation in order to help the model. We also can look at the missclassified examples to check if the algorithm have a high bias or high variance.

Avoidable bias Understanding human level of performance is also important because we can estimate the bayes optimal error. If we know that humans have 1% of error and our algorithm has 8% of error in the training set and 10% in the dev set, then we should try to reduce bias to achieve something closer to 1%. Meanwhile, if our algorithm had 1.5% in the training set and 3% in the dev set, then we should concentrate in reducing variance to make the 3% closer to 1.5%.

How to define human-level error/performance Let's say we have medical pictures to give a certain diagnostics. Then, to define human-level of error, we should get the best possible team of experienced doctors to discuss the pictures and give their opinions on that picture. The error that those people have will be lower than a typical doctor or a single experienced doctor. Therefore, we use this error as the goal.

4.3 Error Analysis

When we're trying to create an algorithm that does some kind of human task, manually checking the errors our algorithm is making can give us some insights of how to improve. This is called **error analysis**.

One of the most simples things we can do in error analysis is check what are the classes that a classifier is mostly making mistakes. Than, we could focus on getting more examples of that class.

Another example could be the case where many images with a white background are being missclassified. Thus, it could be a good ideia to collet more images with white background and train on those images.

If we have some high percentage of error for many different kinds of imagens, we could also evaluate all those ideas im parallel, with many teams working on those different kinds of errors and trying to improve them.

Incorrectly Labeled Data In many cases, our algorithm can perform some errors because the training data itself has some misslabed examples. What should we do when we encounter that?

The first thing to say is that DL algorithms are quite robust to random errors in the trainig set. If the erros are quite random, then it's oks to leave them as they are because this is probably not going to affect much the outputs. However, if the error has a bias (i.e., it's not random), then we definitely should fix those incorrectly labeled examples.

For instance, if all white dogs are labeled as cats, them our algorithm will learn to classify white dogs as if they were cats.

When doing error analysis, we create a table where the lines are the missclassified examples and the columns are the possible errors associated with that particular example. Then, we sum up all the "checks" in each column and see which errors are the most crucial to our application. If incorrectly labeled data is one of those things, we might add a column to check when that instance was misslabeled and, if at the end, there were a lot of them, we might separete a team to work on fixing those labels.

Iterating One important thing to do then building a machine learning project is to first build it quickly and them use error analysis to avaluate your model and improve it. After knowning where to improve, to that and repeat everything again.

4.4 Mismatched Training and Dev/Test Sets

In many cases, teams building Deep Learning systems just throw a bunch of instances into the training set to make it better. And in many of these situations, the training and dev/test sets will end up with different distributions. So how to deal with those cases?

Suppose we have pictures of cats and $200,000$ of them come from the web, have a high resolution, where taken by professionals while just $10,000$ of them come from mobile app, by amartures that will use our app to predict if the image is a cat or not.

Our final user will input a regular image, taken by a phone, while our main source of imagens aren't like that. So what do to?

Option 1 is to just mix these two kinds of imagens and shuffle them around the training, dev and test sets. Now we do have the same distribution of pictures, but we'll end up with a very small proportion of mobile photos in our dev and test sets, which is not what we want at all.

Option 2 is to just put all the imagens from the web into our training set together with some of the mobile images and leave the rest of the mobile images into the dev and test sets (so they'll end up with no web images). Suppose we use all the $200,000$ images from the web into the training set and add up more $5,000$ images from the mobile app, while the dev and testing set will have $2,500$ images from the mobile app each.

Bias and Variance One thing to notice is that bias and variance analysis will be different if we have mismatched training and dev/test sets. Before, we knew that if the training error is 1% and the dev error is 10%, them we have a large variance and probably the algorithm is overfitting. But if the training and dev data come from different distributions, we can no longer say that. Maybe the dev set just has poor quality images and it's harder to predict them, we don't know.

In order to accomplish this error, we can create a new set of data that we'll can it the *training-dev set*, which has the same distribution of the training set, but it's not used for training. Now we use the training-dev to check problems like bias and variance and if it's ok, we know that the 10% is just from the data mismatch.

Addressing Data Mismatch We've seem that creating a training-dev set can show us the amount of error we have due to data mismatch from the training and dev/test sets. So how to address that in order to reduce this error? Indeed there are not automatic ways to do that, but we can look at a few things.

One possible thing is to carry out manual error analysis to try to udnerstand difference. Than, we would try to make the training data more similar, or collect more data similar to dev and test sets. For instance, we could try to make the images of the cats more blury, or low resolution.

4.5 Transfer Learning and Multi-Task Learning

In many cases, we can use a NN trained in some task to improve another NN in other task, that's called **transfer learning**. Let's see how to do it.

Suppose we have a NN to make image recognition and want to do radiology diagnosis. What we could do it just delete the last layer of the NN (the output layer) and randomly initialize a new output layer for the new task.

By doing this, we hope that when the NN learned about how to recognize imagens, it already may know how to recognize parts of images to help us in the diagnosis.

Another example could be a NN for speech recognition and we want a system for a trigger word. We can add several new layers and just train them, or we can train the whole network, but starting with the pre-trained weights.

Transfer learning is useful when we have a pre-trained model that was trained with lots of data, but we're trying to solve a problem that has a small dataset.

What we could also do is try to learn from multiple tasks. That's called **multi-task learning**. Usually we use this kind of learning when we have a NN that is trying to do many things at the same time. For instance, if we have an autonomus car that detects pedestrians, cars, stop signs, traffic lights, etc. Then we could train a single NN that have a large output that predicts all thoses things at the same time. By doing this, we hope that learning one of those things will help to learn the others and lead to a better performance (that's what usually happen in practice).

Multi-task learning is useful when we have similar tasks that can benefit from each other and when we have pretty much the same amount of data for each task.

4.6 End-to-End Deep Learning

In many applications, we have a pipeline with many different stages until we reach a final result. For instance, in speech recognition, we may have to transform the audio into some set of features that will be transformed into phonemes that will be transformed into words that will be transformed into the trascript.

In some cases, when we have a lot of data, we can bypass all those intermediate steps and translate audio into text with a single NN. That's what's called **end-to-end deep learning**.

Chapter 5

Convolutional Neural Networks and Applications

Convolutional Neural Networks are one special kind of NNs that are specially suited for *Computer Vision* problems. Some of these problems include:

- Image classification;
- Object detection;
- Style transfer;
- ...

One of the most challeging facts about Computer Vision is that images can be very large. If we have a 1000×1000 image, than we have an input 3 million variables (because there are 3 color channels). That unquestionably huge for a "standard" neural net (or what we'll call *dense* NN or *multilayer perceptron* network - MLP). CNNs are suited for those tasks because they allow us to reduce the number of parameters we would have with a MLP.

5.1 Convolution Operation

First, we need to understand the convolution operation itself (which is independent of the concept of a neural net). To ilustrate that, we'll use the example of *edge detection*.

So suppose we want to detect vertical lines in a certain picture. One way to do that is by creating a **filter** (also called **kernel**) which is a 3×3 matrix what we'll **convolve** that matrix with the image matrix.

The output of a convolution if given as the following (look at image 5.1):

If we have a 6×6 image, we'll put our filter uppon the image, one the upper left corner. So the filter will be aligned with a 3×3 submatrix of the image. Then, we multiply the terms that are aligned and sum all of them together to get the answer. Figure 5.2 ilustrates this process.

Then, we shift the filter one the left and compute it again until we've moved the filter all the way through the image.

Formally, we could define it as following:

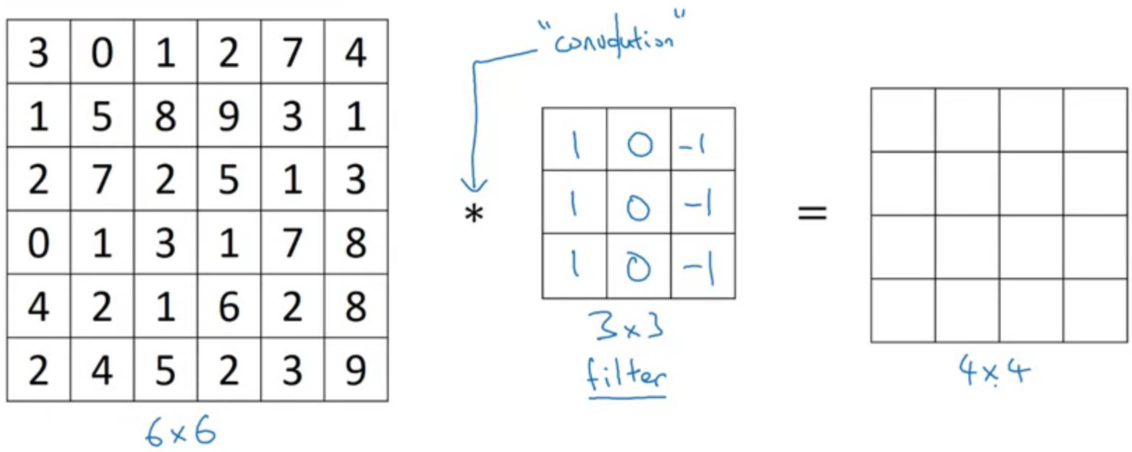


Figure 5.1: Here we have a 6×6 matrix representing an image and a 3×3 filter that we'll convolve with the image resulting in a 4×4, which will be a new image.

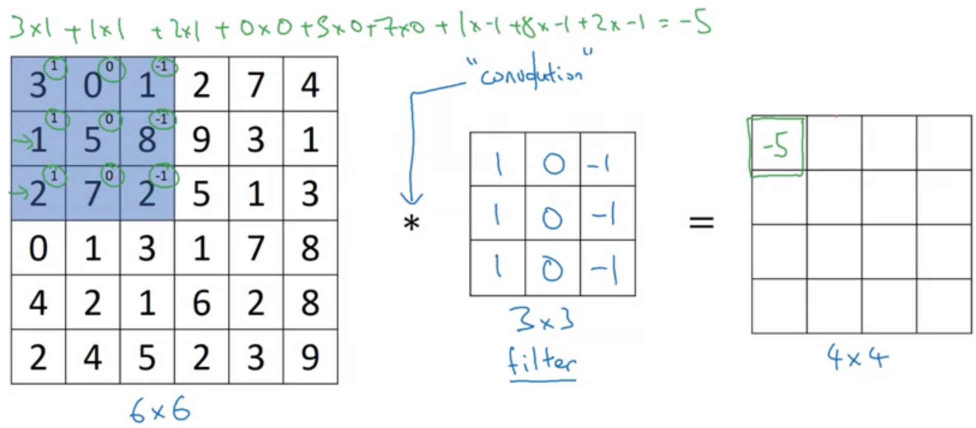


Figure 5.2: Here we see an example of how to compute the elements of the resulting matrix.

$$C = \left(\sum_{l=0}^{n-1} \sum_{k=0}^{n-1} A_{i+k,j+l} \cdot F_{k,l} \right)_{i,j},$$

where A is a matrix and F is a $n \times n$ filter.

To understand why that 3×3 matrix tells us where are the vertical edges, we could think that it cares when we have bright pixels on the left and dark pixels on the right, don't carring much about what's in the middle.

If we wanted to detect horizontal edges, we could use:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Also, there's no particular reson why we would like these particular numbers, we could use:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

which is called the *sobel filter*, or:

$$\begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

which is called the *scharr filter*.

We can also try to learn the filter to have a good edge detector. Indeed, it can learn to detect 45 edges or 70 edges or whatever is better to the problem you're trying to solve. Later in this chapter, we'll see how to make a neural network learn these filters as parameters (but you can already see that it reduces drastically the number of parameters we'll learn, because now instead of having a weight for each pixel, we just have to learn these filters, which are smaller than the original picture).

5.1.1 Padding

Before we procede, we need to talk about *padding*. We've seen that after convolving those matrices, the output is *smaller* than the original input. In many applications, we don't want that, we want the output to have the same dimentions of the original input.

Another down side of that we've been doing is the fact that the border pixels are used way less often than the middle pixels (because there are more filter pisitions that overlad the middle pixels).

Padding is a solution to both of those problems. Before we apply the convolution, we pad the image, extending it with aditional zeros on the border.

Originally, if f is the filter size and n is the input dimension, then the output would have $n - f + 1$ dimensions. But if add an extra border (or pad) of p pixels around the image, than the output dimension will be $n + 2p - f + 1$.

Therefore, to preserve the image dimensions, we must have $p = \dfrac{f-1}{2}$.

Indeed, the two most common type of convolutions are the ones with no padding or the ones with a padding that leads to an output with same size as the input (also, by convention, f is usually odd). We usually called these "valid" and "same" convolutions respectively.

5.1.2 Strided convolutions

Another modification we can do to the convolutions is adding strides, which means that we're not going to step just one to the right or one down, but more than that. This step parameter is called the *stride*.

If we say we'll have a stride of 2, than we'll jump two squares to the left (or down) instead of just one. That leads us to a smaller output, because we'll be computing less times that element-wise product.

Therefore now we have three parameters: f, p and s, the stride. If our input has size n, them our output will have size equal to: $\left\lfloor \dfrac{n + 2p - f}{s} \right\rfloor + 1$.

5.1.3 Convolutions over tensors

The final step before we apply convolutions to neural networks is learning how they work with not just 2d matrices, but with nd matrices (also called *tensors*).

So let's suppose we want to convolve on RBG images. So now we have a $6 \times 6 \times 3$ image. To do that, we need a $3 \times 3 \times 3$ filter (or one filter for each color channel). And notice that the last 3 **must** match the last dimention of the image.

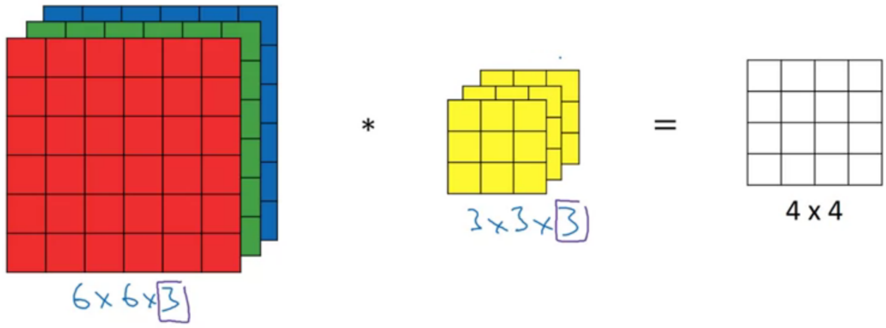


Figure 5.3: Here we have a visualization of a tensor convolution.

Then the operation is very intuitive, you multiply each number of the matrix by the coresponding number in the filter and sum all those numbers.

5.2 One layers of a Convolutional Network

Basicaly, to create a layer of a convolutional network, we need to do many convolutions of the same previous layer (all of them with the same size). Then, we add biases b to those matrices and, finally, apply an activation function (like ReLU or Sigmoid). The last thing we must do is stack those matrices to create a tensor output that will be feeded into the next layer.

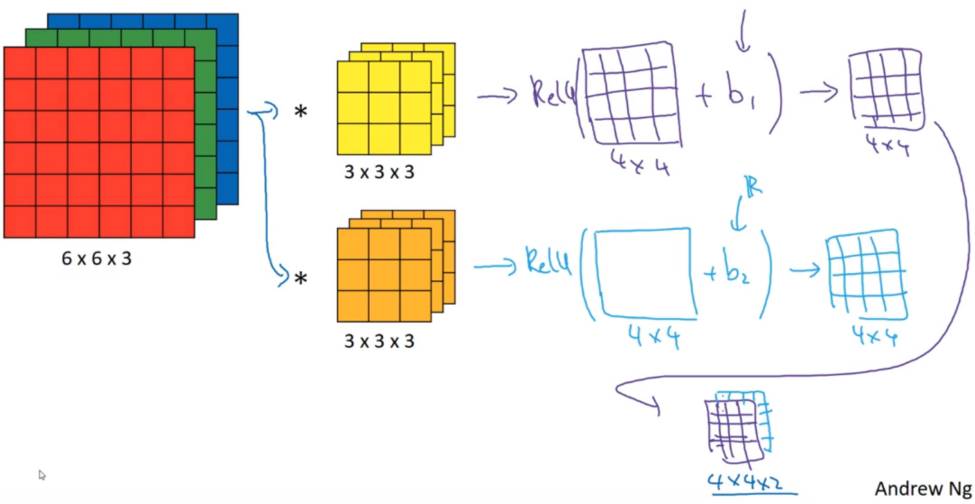


Figure 5.4: Here we see an ilustration of a convolutional layer.

5.2.1 Notation

Let's set some notation we'll be using:

- $f^{[l]}$ is the filter size of the l-th layer;
- $p^{[l]}$ is the padding of the l-th layer;
- $s^{[l]}$ is the stride of the l-th layer;
- $c^{[l]}$ is the number of filters of the l-th layer;
- $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ is the input shape for layer l;
- $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$ is the filter shape for layer l;
- $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ is the weights shape for layer l;
- $n_c^{[l]}$ is the bias shape for layer l.

5.3 Pooling layers

Usually, CNNs don't have just convolutional layers (or CONV layers for short), but can also have **Pooling layers** (or POOL layers for short) and **fully connected layers** (or FC layers for short).

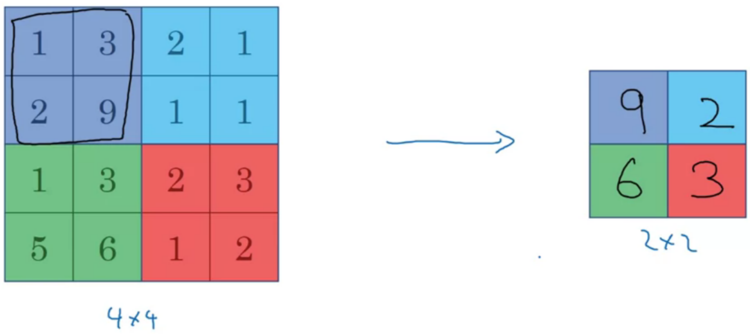


Figure 5.5: An example of max pooling. This example has filter size and stride equal to 2.

Pooling layers are a way to speed up computations and make inputs lower. Figure 5.5 gives an example of pooling layer. Basicaly, we have that moving window just like in convolutions, but instead of multipling and summing the numbers, we just apply an **aggregation function** (like max, min, prod, sum, mean, std) over those numbers.

One of the most common aggregation function is *max pooling* (because it's like we were preserving the filters were some kind of feature was detected). But, in theory, any aggregation could be used.

Also, the concepts of stride, padding and filter size are the sames here, since we have a moving filter.

5.4 Case Studies and Classic Networks

In this section, we're going to study some classic convolution networks to see why they're good and how to join the building blocks we've seem so far.

5.4.1 Classical Networks

LeNet-5 This network was made to recognize $32 \times 32 \times 1$ handwritten digits.

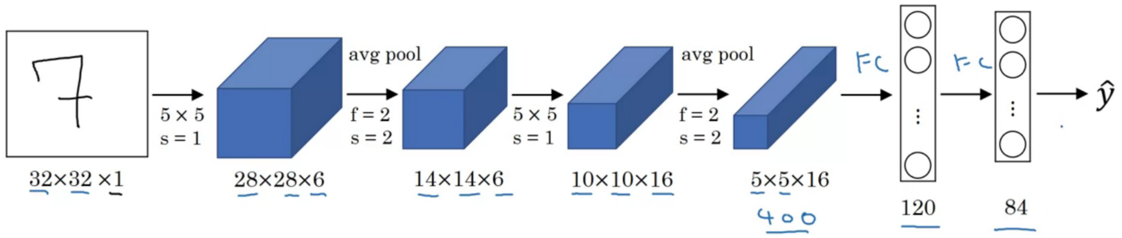


Figure 5.6: Here we see the architecture of LeNet-5. At that time, we used more avg pooling than max pooling.

It was a very small network, with 60k parameters.

We see that it decreases the first and second dimensions and increases the number of channels. Also it starts by interchanging pooling and conv layers and ends with some fc layers.

AlexNet This networks was made to work with $227 \times 227 \times 3$ images.

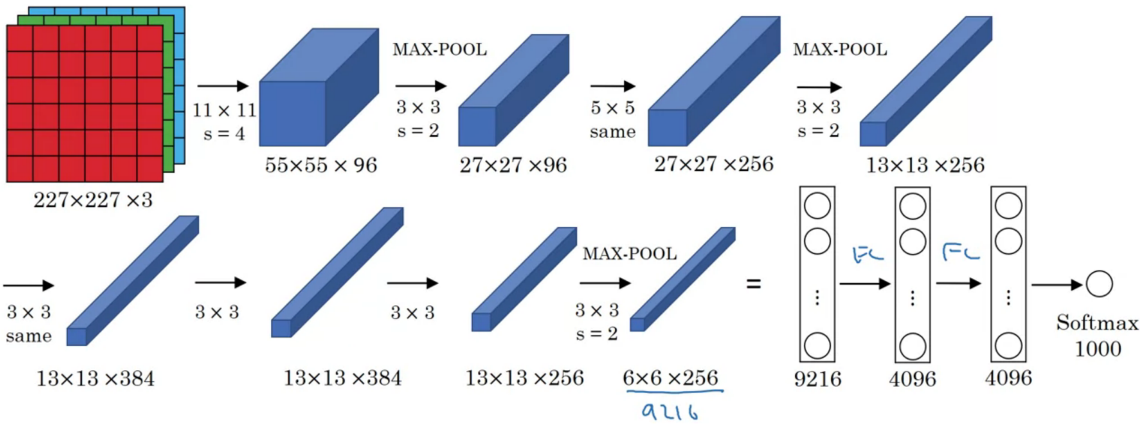


Figure 5.7: Here we have a architecture of AlexNet. This is a much bigger network.

AlexNet is similar to LeNet, but much bigger, with about 60M parameters. They use ReLU here and Max Pooling instead of Avg Pooling.

VGG-16 Instead of using a lot of different layers with different parameters, VGG-16 uses just CONV layers with 3×3 filters, stride of 1 and same padding; and MAX POOL layers of 2×2 and $s = 2$. This network is very deep, so we're not going to draw it here.

This network is huge, with 138M parameters.

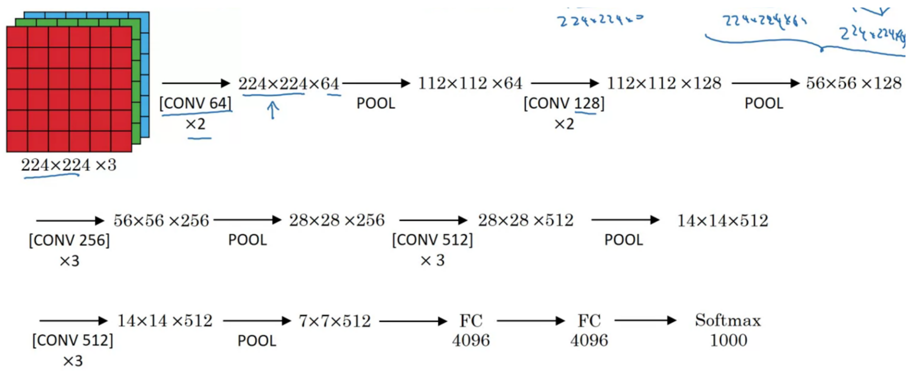


Figure 5.8: Here we see how big is vgg16.

5.4.2 Residual Networks (ResNets)

One of the most famous networks today is the Residual Network. This kind of network use something called **skip layers**, which allow us to feed a deeper layer with a shallow layer and skip some layers in the network. This allow us to avoid some problems we've already mentioned such as Vanishing / Exploding Gradients and make it possible to have networks of over 100 layers.

First, we need to understand what is a **residual block** (there's where the name comes from).

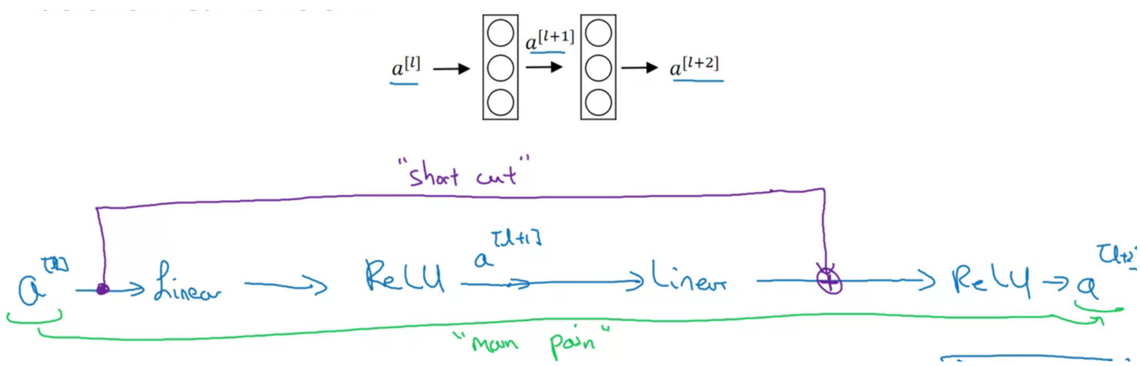


Figure 5.9: Here we see an example of residual layer. Instead of following the main path we already know, res layers allow a previous activation to *short cut* to the next layer and be added to before the ReLU of that layer.

Sometimes, we can also call these short cuts as *skip connections*.

A ResNet is just a Neural Net with lots of these skip connections. In practice we noticed that this kind of connections allowed to train for much longer, keeping decreasing the error.

5.4.3 Inception Network

1×1 convolutions First let's take a look at 1×1 convolutions. You might think this is just multiplying by a number, but it's not quite that. Suppose you have a $32 \times 32 \times 192$ layer and you want to shirking it to a $32 \times 32 \times 32$ layer. How to do that?

One way is using 1×1 convolutions. Basicaly we use 32 $1 \times 1 \times 192$ filters. What these filters do is multiply all the numbers in the "z" coordinate together and output them into some activation function. So it's like we were passing those numbers into a FC layer and outputing the next layer.

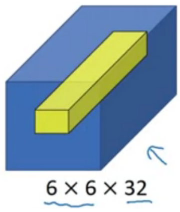


Figure 5.10: A 1×1 convolution filter ilustrated.

Inception networks use *different* filters in the same layers and stack the results in the output.

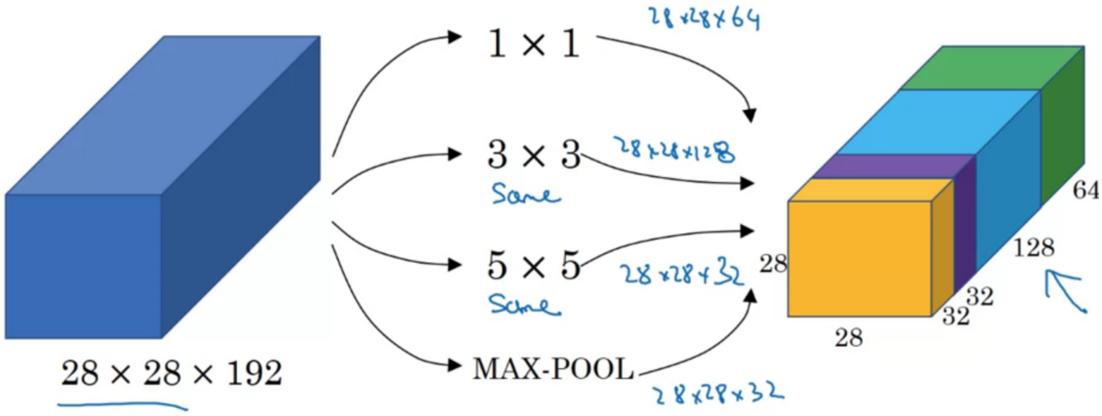


Figure 5.11: Here we see an example of inseption layer, where we apply different filters in the same layer.

5.5 Data augmentation methods

Data augmentation is the technique when we create new instances based on the instances we already have. This technique allows us to have "more" data to train, although we're not really grabbing more data.

The positive points are that you can have more instances with very little cost, but the down side is that you might be overfitting that particular kind of data, so it must be done with caution.

Let's explore some methods os data augmentation when applied to computer vision (i.e. images).

- The most common method is *mirroring* the image;
- We can also take *random crops* of the image;
- *Rotation* is another technique;
- *Shearing*;

- *Local warping*;
- Another kind is *color shifting*, in which we change the colors a little to make it color resilient (one can use PCA in order to find out which colors change more or less);

5.6 Object Detection

Now we're going to learn how to detect object on the image. In object detection, we want to localize potentially several objects on an image, which is different from classifing whether an object is or not on that image.

Before moving on detection problem where we might have several objects, we'll try a simpler problem called *classification with localization*, where we classify if there is or not an object on the image and, if there is, we localize the postion of that object using a *bounding box*.

5.6.1 Object Localization

To solve this problem, instead of having a single softmax neuron at the end of our network, we'll add four more output nodes:

$$b_x, b_y, b_h, b_w$$

These numbers represent the position and dimensions of the bounding box that we'll put around the object if it exists. b_x, b_y is the center of the image.

Now, let's extend the problem a little and say that we might have different objects and we also want to say if there an object, then what object it is.

Suppose we have n objects. Then we'll encode the objects using an one-hot encoding and output that encoding as well. So our output will look like this:

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}$$

p_c is the probability that the image contains an object and, if it contains an object, than b_x, b_y, b_h, b_w will be the bounding box and $c_1, c_2, \ldots, c_n$ will be the one-hot enconding of the object.

If there's no object, then p_c must be zero and the other values will be all *don't cares* (meaning they can be anything).

To define the loss function, basicaly we have two cases:

- When y_1 (p_c) is 1, we take the sum of squared differences of all entries of the vector;
- When y_1 is 0, we take the squared difference of y_1 only.

Although, in practice we can use difference suberrors for each entry (use one kind of error for the bounding box, one for the one-hot enconding and one for the probability of containing an object).

5.6.2 Landmark detection

Instead of outputing a bouding box, we may want the neural network to output **lendmarks**, which are important points on the image.

Suppose we want to perform face recognition. Than we might get the lendmarks of an image and check if there are eyes, month, nouse, etc.

Each lendmark is a pair of numbers $(l_x,\ l_y)$.

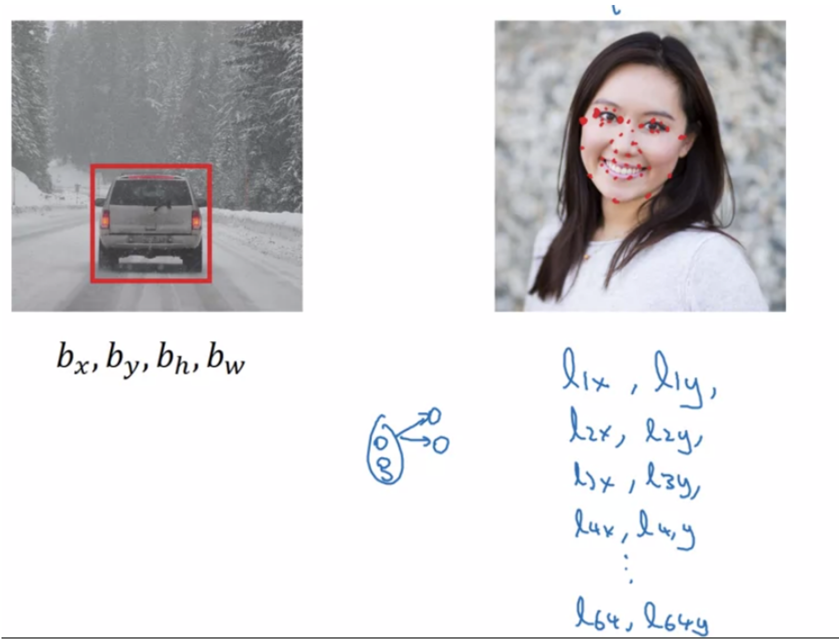


Figure 5.12: The difference between the bounding box and the landmarks.

Now our network outputs if there's an image and a set of nodes for the x and y position of each landmark.

When training, it's important to have consistent landmarks. That means we must always have the same amount of lendmarks and they must be in the same location. So if lendmark 1 is the left corner of the left eye, that must be true for all imagens.

5.6.3 Slinding windows detection algorithm

Let's now see a basic algorithm for solving object detection algorithms.

So if we have images of cars and we have an algorithm that solves the problem of classifying the image as a car or not car, then we can now take a picture with many cars and slide a window across the image, checking if each window is or not a car. If it is a car, then we have a bounding box (the window we are currently in).

We might change the window size the slide for many sizes.

This concept is very similar to that concept of passing a convolutional filter over a matrix.

Of course, the computational cost of this algorithm is very high, because each time we slide the window, we have to ask the network is that new image is or not a car (or any object we're detecting).

Fortunately, there's a much singler and cheaper way to implement this algorithm using convolutions. To understand that, though, we have to understand how to turn fully connected layers into convolutional layers. Let's see that first.

Suppose we have a $5 \times 5 \times 16$ volume and pass it to a 400 node FC. Then we could instead of that use 400 filters of $5 \times 5 \times 16$. Then the resulting dimention will be $1 \times 1 \times 400$. So instead of viewing that FC layer as a plain stuff, we say it has a depth of 400.

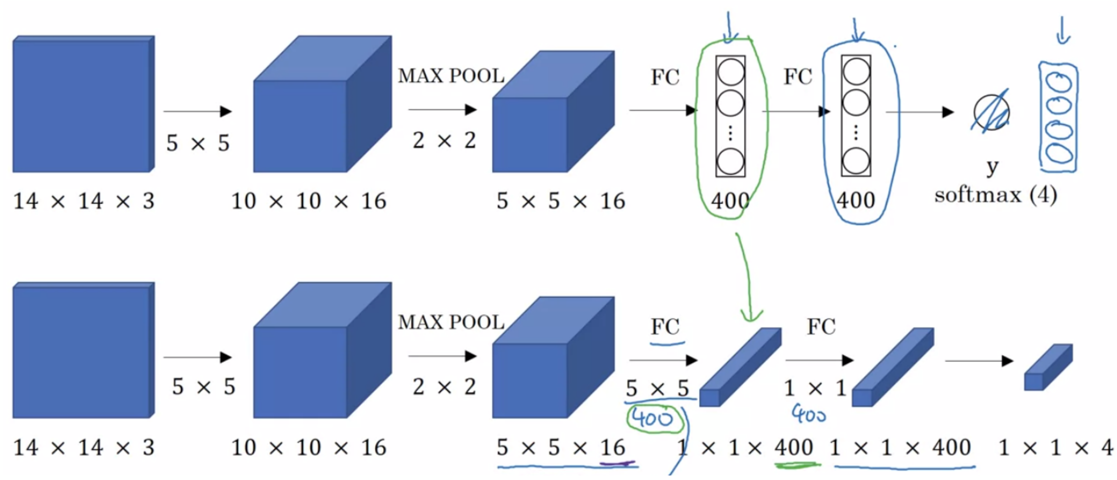


Figure 5.13: The difference between the FC layers and Conv layers.

Now let's see how to speed up the sliding window algorithm.

So suppose we have a $16 \times 16 \times 3$ data set and the ConvNet of figure above. In the original sliding window algorithm, we would apply the ConvNet four times (stride of 2) in the images. But instead of that, what we do is just feed the $16 \times 16 \times 3$ images into the ConvNet, resulting in a $2 \times 2 \times 4$ output. Each "square" on that output is equal to one window in the original algorithm.

5.6.4 YOLO algorithm

Even using the convolutional aproach, the sliding windows algorithm will not output precise bounding boxes, because it depends on the window size the stride used. The YOLO algorithm, however can do that also with only one pass of a convolutional network (what's why it's called *You Only Look One*).

In this algorithm, you divide the original image into a grid (3×3 in the example, but much bigger grid in real life). Now each square of the grid is considered a single image and we apply the localization algorithm we've learned later. Therefore, for each square we'll have a probability of containing an object, a bounding box and a one-hot encoding. So let's suppose the one-hot enconding contains 3 numbers. Then we'll have a "global" output of $3 \times 3 \times 8$ $(1 + 4 + 3 = 8)$.

What we do then is use a convolutional network with the size of the input that maps to that $3 \times 3 \times 8$ output. The network will predict the whole image at once, but will output at maximum one bounding box per square in the grid, which is awesome.

We just need to note some things about this method.

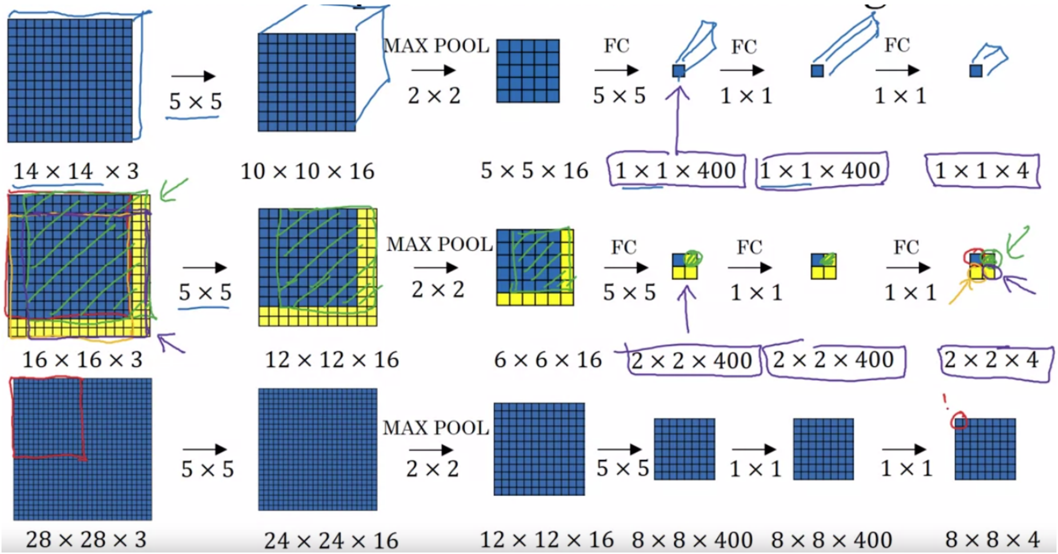


Figure 5.14: Each square on the final output represent one window of the original one, but everything is done with one pass.

Figure 5.15: Here we see an example of the grid in the YOLO algorithm.

1. *An object could be between two or more squares.* What we do then? In that case, we asign the object using its central point. The cell with the central point is considered the cell where that object is at;

2. *How to train the network? How to create the training examples?* To do that, we use the convention that the upper-left corner of the image is $(0,\ 0)$ and the lower-right is $(1,\ 1)$. Then b_x and b_y will be values between 0 and 1 and b_h and b_w represent the percentage of coverage of the image. In particular, they can be greater than 1 when the bounding box is greater than a single cell.

Intersection over Union

To if an algorithm is predicting correctly a bounding box, we use the concept of *Intersection over Union* (or IoU). To compute that, we take the predicted bounding box and the real bouding box and compute the intersection and union of both, than we divide the union by the intersection. The closer to 1, the better. In particular, the value is 0 is the intersection is null (the predicted bounding box is completely out the real bouding box) and is 1 if the bouding boxes are exactly the same.

In general we use 0.5 as a good value to achieve, but you might change that acording to your needs.

Non-max Suppresion

In YOLO, we divide the image into many cells and an object might fall into many of them and many of them might detect that same object. We need therefore a way to keep just one bounding box for each object. How to do that with an algorithm?

What's where the *non-max suppresion* comes into play.

In this algorithm, we select the bounding box with the highest p_c and look at all the remaining rectangles to check if any of them has a high IoU. If so, them we remove those bounding boxes, because they belong to the same object (of course, we look only at those that were predicted for the same object we're looking right now). Them we do that again for the second highest p_c and so on, until we only have one bouding box for each object.

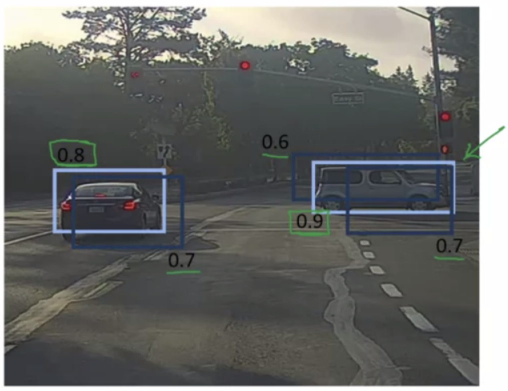


Figure 5.16: Here we see an example of non-max suppresion. The bouding boxes with p_c equals to 0.9 and 0.8 are kept while the others will be excluded. Also notice that we don't exclude the bb with $p_c = 0.8$ when looking at the one with $p_c = 0.9$ because they have an IoU of 0 and, thus, aren't the same object.

Anchor Boxes

What if one cell of our grid wants to detect multiple objects?

Let's suppose we have a car and a pedestrian in front of it and their center points are so close together that fall into the same cell. That's one of the reasons why to use *anchor boxes.*

In this concept, we allow a cell to predict many bounding boxes according to a predefined shape, called anchor box. In this case, we could predefine two anchor boxes, one which is in a portrait shape and another in a landspace shape. the portrait one will detect the car and the landspace one will detect the pedestrian.

So now instead of having that 8 nodes output for each cell of the grid (in our previous example), we would have 16 nodes for each cell, because we would have two probabilities, two bounding boxes and two one-hot encodings.

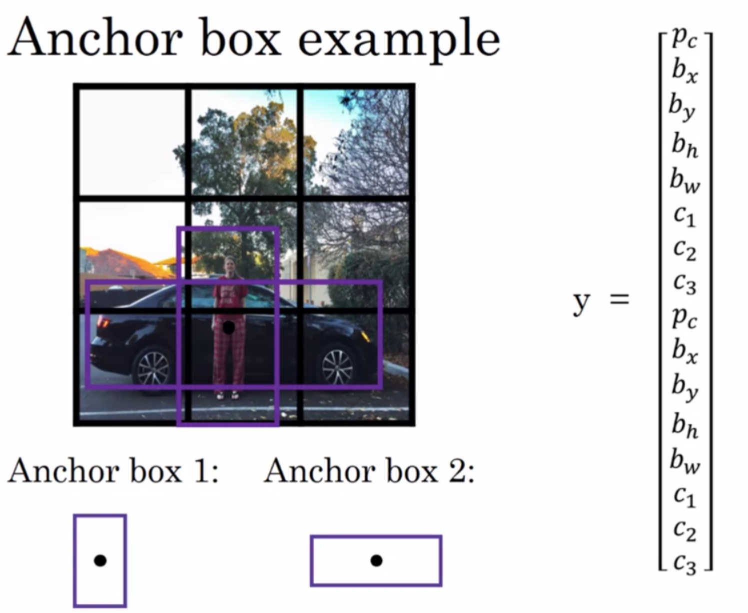


Figure 5.17: Here we see the example showing the anchor boxes predefined and how the y label will look like for each cell of the grid.

One doubt might be how we should where we'll predict a class if we just have one class. Well, in that scenario, we'll output it to the anchor box that is more closer to the bounding box we're predicting. We use the IoU of the bounding box with the anchor box and output to the anchor box that is closer to what we're predicting.

If in the example the upper output (p, b and c) refer to anchor box 1 and the lower output refer to anchor box 2, then we're likely to predict pedestrians using the lower part of the output and cars using the upper part.

One might think that this concept is very useless because that cenario where a single cell wants to predict more than one bb is very unlikely to occour and the downside of doubling the output is greater than that benefit. But actually if we match the anchor boxes to the usual shape of the classes we want to predict, than it can contribute to our algorithm because each output cell will be specialized in predicting a certain class.

In our example, some cells will always predict the pedestrians and others will always predict cars (because pedestrians have that portrait shape while cars have a landspace one).

5.7 Semantic Segmentation

In this section we're going to explore another vision problem: *semantic segmentation.*

In this kind of problem, instead of identifying objects using bounding boxes, we input an imagem and want the network to color every single pixel with a color that corespond to a particular kind of object. All the sky will be colored in black, all the cars will be colored in brown, all the buildings will be colored in blue and so on.

So how to do that? The CNNs we've seen so far are only able to make the input smaller and smaller (and deeper), but we need a away of applying convolutions and after a while, making the shape of the computations go back to the original size. To make that, we'll need to implement a *transpose convolution.*

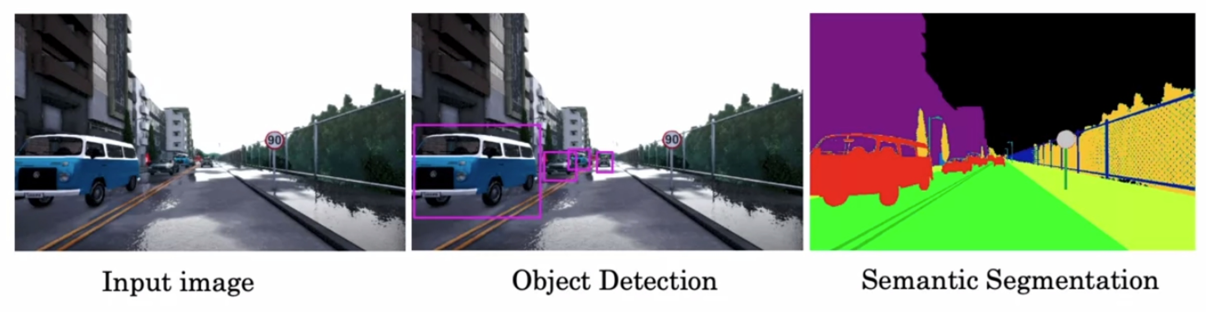


Figure 5.18: In the picture we see the differences between object detection and semantic segmentation problems.

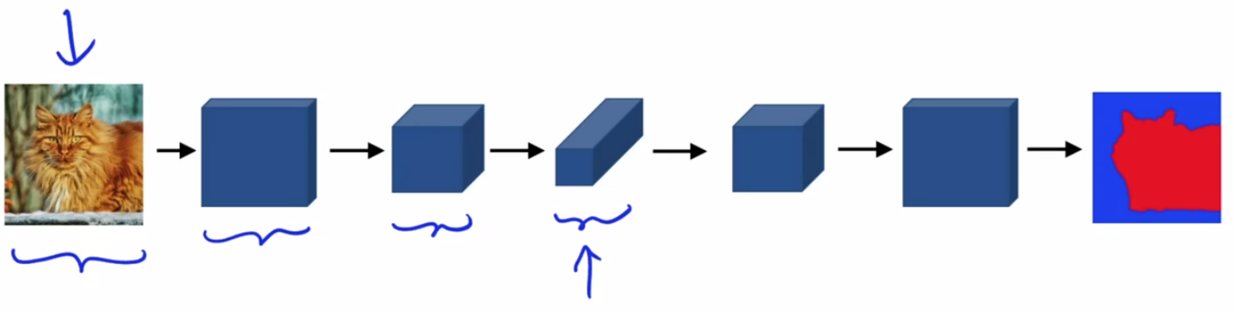


Figure 5.19: Here we see an example of neural network for semantic segmental. We decrease the height and weight and increase the number of channels until some point and them increase the height and weight and decrease the number of channels to the original size.

5.7.1 Transpose Convolutions

Understanding transpose convolutions is very important to understand how to get a 2×2 input and blow it up to a 4×4 output (or any input smaller than output).

To learn that, let's use an example. Suppose we have a 2×2 input and want a 4×4 output. So we're going to use a 3×3 filter, padding of 1 and stride of 2.

Our input and filter look like:

$$I = \begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix} \qquad F = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 1 \\ 0 & 2 & 1 \end{bmatrix}$$

First, we initialize the output matrix with zeros and pad it by 1 (yes, we pad the output, not the input). Then, for each entry in the input, we're going to perform the following operations:

1. Multiply the filter matrix by that number;

2. Now put the filter matrix in the right position in the output matrix (I'll show what I mean by right position in the figure);

3. Then sum each element in the filter with the current element on the position that match with the filter entry.

Simple like that!

Now what I mean by "right position"? As we've padded the output matrix, now it's 6×6. So for the entry $0, 0$ of the input matrix, I'll match the $0, 0$ entry of the filter with the $0, 0$ entry of the output matrix.

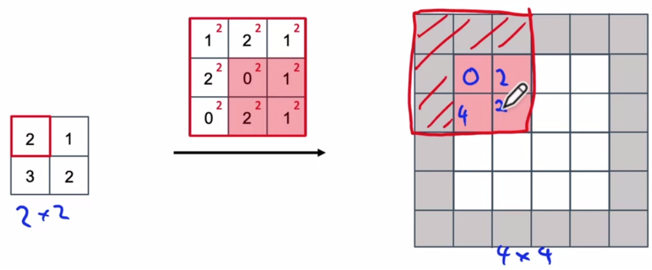


Figure 5.20: Here we see what happens in the first step of the transpose convolution. We multiply the filter by 2, put it in the $0, 0$ position of the output matrix and put the numbers on the right spots (ignoring the padding because it will not enter in the final output).

Then, I move on to the next entry in the input matrix, which is $0, 1$. Since we have a stride of 2, will multiply those coordinates by 2 resulting in $0, 2$. Now I put the filter in the $0, 2$ entry of the output matrix and sum the filter numbers as I said before.

Then, I move to $1, 0$ and, therefore, match the filter if the position $2, 0$ of the output matrix.

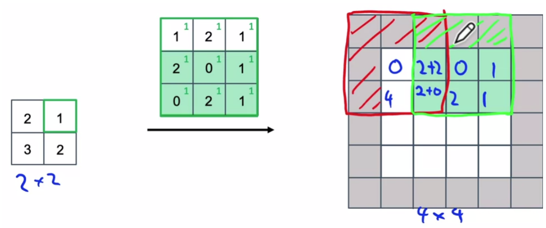


Figure 5.21: Here we see the next step of the covolution.

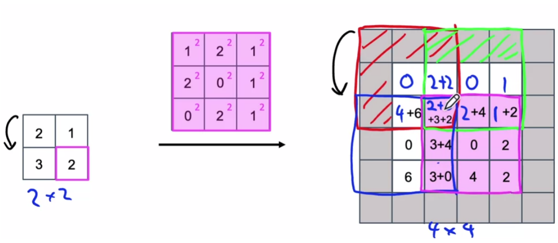


Figure 5.22: Here is the output matrix at the end of the transpose convolution.

At the end, our output matrix will be:

$$O = \begin{bmatrix} 0 & 4 & 0 & 1 \\ 10 & 7 & 6 & 3 \\ 0 & 7 & 0 & 2 \\ 6 & 3 & 4 & 2 \end{bmatrix}$$

5.7.2 U-Net

Now we'll use the transpose convolutions to build a U-Net, which is the main architecture for solving semantic segmentation problems.

So as we've seem before, we use normal convolutions to reduce the image into a smaller shape and then we use transpose convolutions to make it back to the original shape. That small shape region will help us to reduce the amount of information about the image, forcing the network to really learn the fundamental of that we're trying to map.

But also, there's one more stuff we need to add to that architecture to call it a U-Net, and that's skip connections from the normal convolution part to the transpose convolution part. Those skip connections will make the model use not just the fundamental information of the image, but also join that with the high resolution information contained in the first half of the network.

Let's take a look at the U-Net architecture.

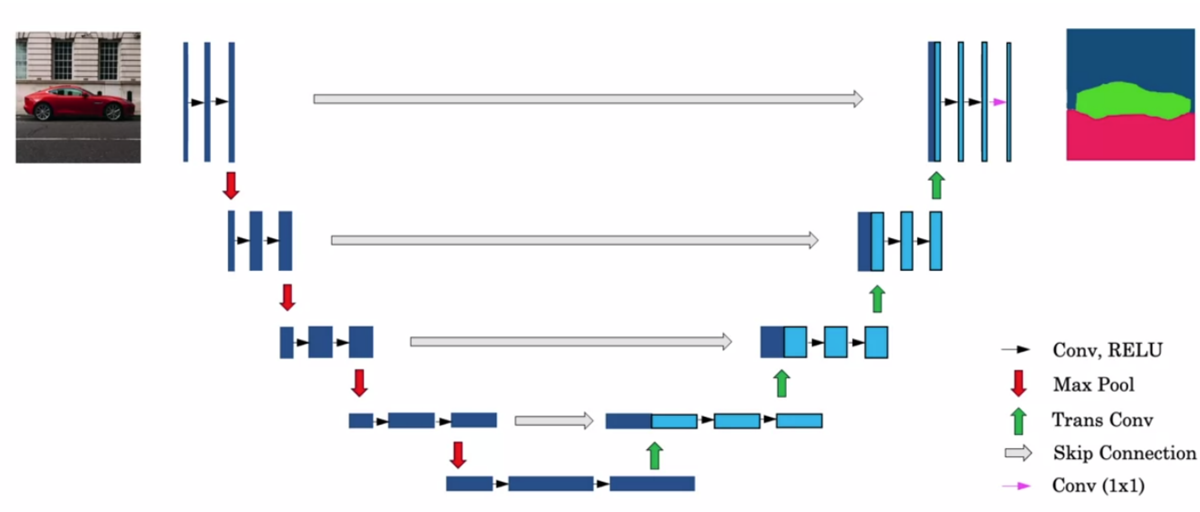


Figure 5.23: That's the original U-Net architecture.

We can see that in this architecture, we use convolutions actually with the "same" padding, to keep the input size and than use a max pooling layer to reduce the input size. Each time we use two conv layers and one max pooling layer.

Than, after a while, we start using transpose convolution layers instead of max pooling, to return to the original size. So we do two conv layers and one trans conv layer (but in the first conv layer we have a skip connection from the part of the network where we had the same size, but in the first half).

At the end, we have a 1×1 convolution to get our output.

One can see that the architecture is called U-Net becuase it looks like an U. We have a *downscaling part*, a *bridge part* and a *upscaling part*.

5.8 Face Recognition

Before moving to the face recognition problem, we have to talk about the **face verification** problem, in which we're given an input image and a name/ID and need to output whether the input image is that of the claimed person. That's called a one to one problem.

The recognition problem is much harder, because it envolves more than one person. In the face recognition problem, we have a datebase with K persons and, given an input image, we need to ouput the ID of the person if the image is any of the K persons in the database or output "not recognized". That a one to K problem.

If we have a face verificiation system with a 99% accuracy and apply it to the recognition problem, we'll have K times more chances of having an error, because now we have K persons.

5.8.1 One-shot learning

One-shot learning is a kind of machine learning algorithm that just need one input to be trained. In the case of a face recognition problem, that's quite often because we might have just one photo of each person in our database and want to, given a new photo, predict if it's from anyone in the database or is not present in the database.

We could apply the image into a CNN and predict a vector of $k+1$ dimention with a softmax layer where k is the number of persons. That vector is the probability of each person or none of them (that why the $+1$). That however is very unlikely to give good results because just one photo isn't enough to train a CNN. The what if we want to add new people to the datebase, we would need to retrain the CNN, losing everything.

In this kind of problem, we want to learn a "similarity" function, which is a function $d(img1, img2)$ measuring the degree of difference between images.

Is that degree is low for two images (below a threshold τ), we say the two images belong to the same person.

5.8.2 Siamese network

But how to actually learn that d function? One good way is using a *simamese network*. In this kind of neural network, we have a simple CNN just like we had throughout this chapter, with some Conv Layers and some FC Layers at the end. But just in the head (the last layer) of that network, instead of having a softmax unit to output some small vector, we actually cut that and have a big FC layer at the end. Let's suppose we have a FC layer with say 128 units.

We define this output as an *encoding* of the input. So if we apply a picture $x^{(1)}$ through the network, it will output the encoding of that image $f(x^{(1)})$.

Therefore, each image $x^{(i)}$ has an encoding $f(x^{(i)})$. What we want now is to compare those encodings in a way that the same person will have similar encodings.

So we can use $d(x^{(i)}, x^{(j)})$ as $\left\|x^{(i)} - x^{(j)}\right\|^2$. And we want d to be a large number if i and j are different people and d to be a small number if they're the same. We need, then, to make our network learn parameters such that these conditions are satisfied using backpropagation.

5.8.3 Triplet Loss Function

To train a siamese network, we need to define a loss function. The main loss function used is the *triplet loss function*, which needs two photos of the same person and one photo of another person.

For this loss function, we set an *anchor*, a *positive* instance and a *negative* instance. That's our triplet $\langle A, P, N\rangle$.

What we want now is to set the difference between the anchor and the positive case to be lower than the difference between the anchor and the negativa case. That leads to the equation:

$$\|f(A) - f(P)\|^2 \leq \|f(A) - f(N)\|^2$$

However one might see that it can be solved quite trivialy by just predicting zeros or actually by predicting the same value for any image. If $f(x) = c$ for a constant vector c, then we would have $0 \leq 0$ always, which would be interpreted as a good thing.

That's why we also add a *margin term* α, which is a hyperparameter. We say that the difference between those differences much be at least α. In the equation we have:

$$\|f(A) - f(P)\|^2 + \alpha \leq \|f(A) - f(N)\|^2$$

or:

$$\boxed{\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha \leq 0}$$

And now we define the error function as a sum over all the triplets:

$$\mathcal{L}(A, P, N) = \max\left(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha,\ 0\right)$$

$$\boxed{J = \sum_{i=1}^{n} \mathcal{L}\left(A^{(i)}, P^{(i)}, N^{(i)}\right)}$$

Also, one might see that we need more than one picture per person to train the network. Our training set could have 10k picture of 1k people, with an average of 10 pictures per person.

When training the network, we do need more than one photo, but when applying it just to encode a new person, we'll not need more than one photo because we are assuming our network learned how to encode a photo in a way that's good to, when exposed to a photo of the same person, have a close encode.

Now the question we still need to answer is how to choose our triplet $\langle A, P, N\rangle$. If we choose very different people, our constrain will be easily satisfied, so we need to, somehow, this to make is hard for the network (show hard examples).

And we actually know what's hard for the network because, when training, some triplets will have $d(A, P)$ and $d(A, N)$ much closer than others. These are the ones we like, because they will push the network futher into increasing $d(A, N)$ or decreasing $d(A, P)$ until the point the difference is smaller than α.

Therefore, we chose the triplets there aren't still satisfied.

5.8.4 Face Verification as a Binary Classification Problem

Instead of using the Triplet Loss Function that we've seem before, we can also concatenate the outputs of the simamese network into a final logistic regression output that predictes 1 is the images are the same person or 0 if it's not.

That's a nice way of learning a similarity function instead of creating one.

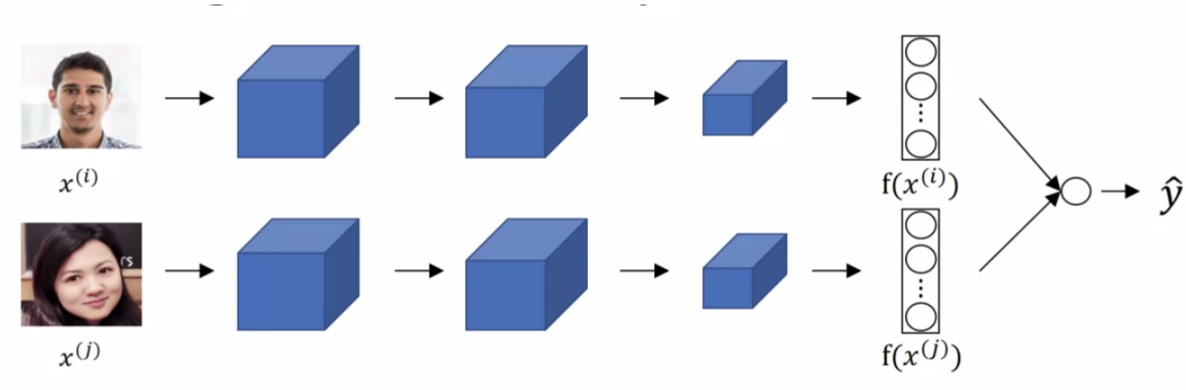


Figure 5.24: We pass both images through the same CNN, which encodes them into a feature vector and use those vectors as inputs for a final layer which predicts 1 or 0. If we're not in the training fase, but in the deploy face of our project, we can precompute the encodings of the saved people the use the input to compare with all of them and check if the input is any of the saved people.

5.9 Neural Style Transfer

Neural Style Transfer is a technique that allow us get any image and apply a style of another image over it.

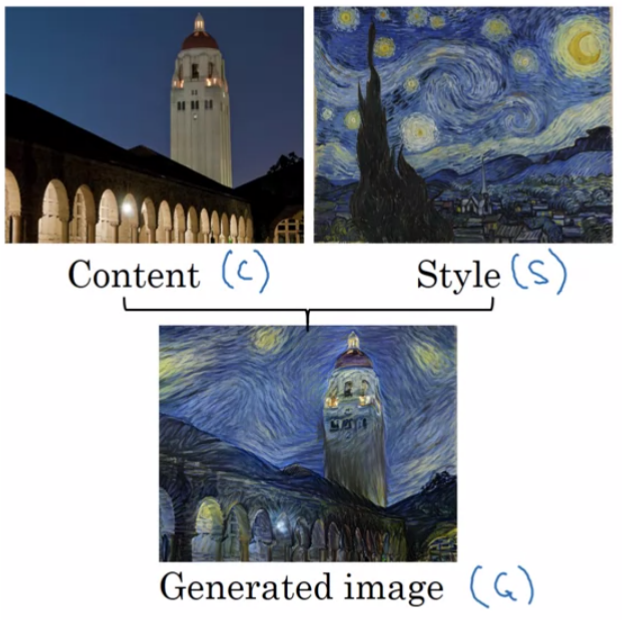


Figure 5.25: Here we have an example of neural style transfer.

In order to transfer a style, we need to get all the features learned in many different layers of a ConvNet. So let's understand a little better what exactly is a CNN learning.

To get intuition on that, let's make an exercise of picking a single neuron unit in a layer of the network. Then let's plot all the images that maximize the activation of that unit. But since a neuron in the first layer don't actually sees much of the image, we're going to plot only the pixels that the neuron really sees in that image of maximum activation.

Plotting that that, we'll get a figure like 5.26. Let's now repeat that for other layers.

Figure 5.26: This figure shows the 9 images for 9 random nodes in the first layer of a CNN. These images are the ones that maximize the respective neurons. We can clearly see that each 3×3 square in the matrix above detects a different pattern of image. Most of them detect lines with some particular colors. That's because in we got neurons from the first layer.

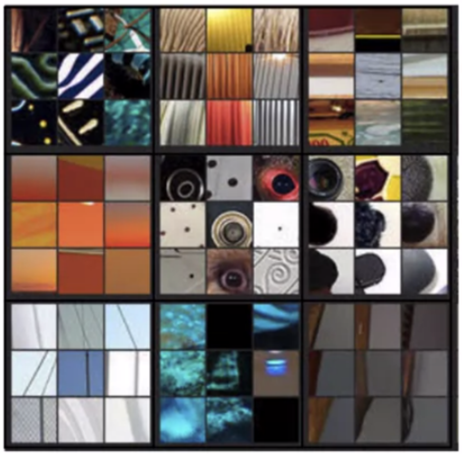


Figure 5.27: This figure shows the same as the previous, but for neurons from the second layer. We see that we're building complexibility.

5.9.1 Cost Function

The first step to apply neural style transfer is creating a cost function. So let's remember that we have a Concent C, a Style S, and want a Generated image G. We want to define $J(G)$ to be the cost function. Then, we can use Gradient Descent to minimize that function.

We'll break that cost function into two parts, the concent cost and the style cost:

$$J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G)$$

The style cost measures how close is the style of S to the style of G, and the content cost measures how close is the content of C to the content of G. We'll average those costs using two hyperparameters α and β.

We could really just use one hyperparameter, but the original authors of the technique use two, so we'll just do the same.

Figure 5.28: Now we see the same as the two previous figures, but for layer 3. We can see that a neuron is detecting people, while other is detecting tires and so on. We observe that each neuron is learning a different complex pattern.

After defining the cost function, we initialize G randonly and run gradient descent updating the pixels of G until it looks good.

Content Cost Function For the content cost function, we recall on what we've seen before on what each layer of the ConvNet learns. To define this cost, we get a pre-trained CNN and select a layer l (nether too shallow or too deep). If we run two images, that layer will have different outputs. Those outputs can be similar or not, but in general they will be similar if the *content* of both images is similar. So we'll define that difference as the content cost function.

Formally, if we have two images C and G, let $a^{[l](C)}$ and $a^{[l](G)}$ be the activations of layer l on those images. Then, we define:

$$J_{\text{content}}(C, G) = \frac{1}{2} \left\| a^{[l](C)} - a^{[l](G)} \right\|^2$$

Observation. We're using a as a flattened vector of the layer.

Style Cost Function To measure the style of an image, we need to define what is style.

Formally, we'll define it as the *correlation between activations across different channels of a convolutional layer.* Let's understand what that mean.

Basicaly we've seen before that each activation (neuron) detects different kinds of shapes and patterns in an image. We can define the style as the correlation of those patterns (i.e., if a pattern is present in an image, than another pattern is also present). This gives us a formal way to express style.

So to calculate that, we need to define a *style matrix G*.

First, let's define $a^{[l]}_{i,j,k}$ as the activation at (i, j, k) of our convolutional layer. The matrix $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$ (a squared matrix of dimensions equal to the number of channels of a layer). An element $G^{[l]}_{k,k'}$ of the matrix computes the correlation for layer l between channel k and k'.

To compute the matrix, we compute each element as:

$$G^{[l]}_{k,k'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l]}_{ijk} a^{[l]}_{ijk'}$$

Observation. The right formula actually indicate that those values are calculated over the Style S, but I'll ommit that since the formula would be too populated.

We now define the style cost function for a particular layer l as:

$$J^{[l]}_{\text{style}}(S, G) = \frac{1}{\left(2n_H^{[l]} n_W^{[l]} n_C^{[l]}\right)^2} \left\| G^{[l](S)} - G^{[l](G)} \right\|^2$$

or

$$\boxed{J^{[l]}_{\text{style}}(S, G) = \frac{1}{\left(2n_H^{[l]} n_W^{[l]} n_C^{[l]}\right)^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'} \right)^2}$$

And the authors found out that the result is better if we sun those correlations between many different layers, so the final style cost function is:

$$\boxed{J_{\text{style}} = \sum_l \lambda_l J^{[l]}_{\text{style}}(S, G)}$$

and we use an additional set of hyperparameters λ_l for that.

Finally, the final cost function will be:

$$\boxed{J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G)}$$