Chapter 1

Feed-Forward neural networks

To understand the technique used in this report, it is necessary to understand basic neural networks functionning. Given a scenario with a training set of labeled data $(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ denotes the training example composed of multiple features, say $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$, and $\mathbf{y}$ the corresponding label. Let's introduce the idea of 'perceptron'.
Perceptrons are the building blocks of neural networks, and the best way to get stated is with an example. Assume at the university's admission office the students are evaluated with two pieces of information, the results of a test and their grades in school. Let's take a look at some sample students, see fig. 1.1.

Figure 1.1: test vs grades

The data on the figure can be nicely seperated with a line, where most students above the line get accepted and most students under the line get rejected, see fig. 1.2. Therefore this line is going to be our model.
The model makes a couple of mistakes since there are a few blue points that are under the line and few over the line, but they are considered as noise and add no new information to our model. Now, the natural question that arises: how do we find the line ?

We start by labeling the axies $\mathbf{x} = \{x_1, x_2\}$. The boundering line seperating the students has a linear equation specifically: $2x_1 + x_2 - 18 = 0$. Ploting the grades in the equation

Figure 1.2: seperating line

gives rise to a score, if the score is poitive –the student gets ploted in above the line–, the student gets accepted with otherwise not. This is called a prediction.

In a more general case, our boundary will be an equation of the following form:

$$w_1 x_1 + w_2 x_2 + b = 0.$$

Abbreviating this equation into vector notation:

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \tag{1.1}$$

Where $\mathbf{w} = \{w_1, w_2\}$. We refer to $\mathbf{x}$ as the input, $\mathbf{w}$ as the weights and b as the bias. Here $\mathbf{y} = \{0, 1\}$ is the label, where 0 indicates the student being rejected whereas 1 indicates the student being accepted. Finally, our prediction is going to be called $\hat{y}$ and it will be what the algorithm predicts that the label will be, namely:

$$\hat{y} = \begin{cases} 1, & w \cdot x + b \geq 0 \\ 0, & w \cdot x + b < 0 \end{cases} \tag{1.2}$$

and the goal of the algorithm is to have $\hat{y}$ resembling y as closely as possible. Reorgnizing the equations in a graph and generalizing, gives rise to fig. 1.3. Here the bias is consider as a dummy input with value 1 to the Perceptron with weight b.

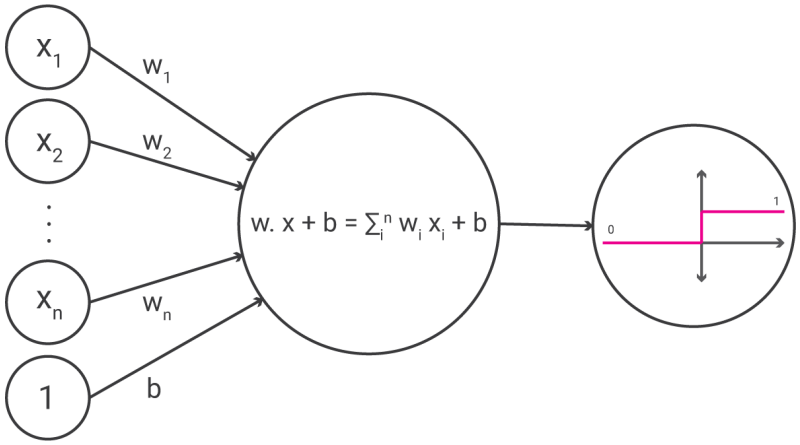


Figure 1.3: perceptron's graph representation

1.1 Cost function

In order to estimate the accuracy of the algorithm, or otherwise stated determine how well a certain prediction given by the algorithm is, we may establish a cost function, which measures the error the algorithm makes on some prediction (cost function is ofter refered to as error function). There are more that one choise for such a function. Equation (1.3) can be used, it is called "The Mean Squared Error".

$$L(w,b) = \frac{1}{2}\sum_{i=1}^{n} ||y-\hat{y}||^2. \tag{1.3}$$

This function, becomes large when our network approximates y badly, and small when the approximation is accurate. Additionally notice that if we set $L_x = \frac{1}{2}(y-\hat{y})^2$ we have that:

$$L(w,b) = \sum_{i=1}^{n} L_{x_i}. \tag{1.4}$$

This property will be important in the algorithm described in Section 2.2.3.

Another way of defining a cost function is using the "The Maximum Likelihood Estimation" technique, since the sigmoid function deals with probabilities. We take the joint probability of the entire training set, assuming the training examples being independent events:

$$L(w,b) = p(y^{(1)},y^{(2)},\ldots,y^{(n)}|x^{(1)},\ldots,x^{(n)}) = \prod_{i=1}^{n} p(y^{(i)}|x^{(i)}) \tag{1.5}$$

where $x^{(i)}$, $y^{(i)}$ represent the i^{th} training example and label respectively. Thus by maximizing the joint probability, or respectively minimizing the $-\log$ of the likelihood, we can get an estimate of the parameters *w and b*. Now, in our worked example, the neural network can be treated as a random variable having a Birnoulli distribution, therefore

eq. (1.5) can be rewritten as follows:

$$L(w,b) = -\sum_{i=1}^{n} y_i \log(\hat{y}) + (1-y)\log(1-\hat{y}). \tag{1.6}$$

Equation (1.6) is usually the cost function used for Bernoulli distributed labelled data. It is ofter refered to as **binary cross-entropy** or BCE for short.

1.2 Gradient descent

In order to minimize the cost function we rely on optimization algorithms from numerical methods as it is unpractical to solve manually. The technique used in deep learning is the gradient descent.
The gradient of a differentiable function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ at a point $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is a vector in $\mathbb{R}^n$ of the form

$$\nabla f(x) = (\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x)) \tag{1.7}$$

It is a well known result that, given a point $x \in \mathbb{R}^n$, the gradient at that point the direction of steepest ascent. Given that f is differentiable at x, the vector $-\nabla f(x)$ indicates the direction of steepest descent of the function f at the point x. In order to obtain the minimum value of the function, the gradient descent strategy tells us to start at a given $x_0 \in \mathbb{R}^n$, calculatethe value of $\nabla f(x_0)$, and then proceed to calculate a new point $x_1 = x_0 - \alpha \nabla f(x_0)$, where $\alpha > 0$ is called the **learning rate**. We then repeat this process, creating a sequence $\{x_i\}$ defined by our initial choice of x_0, the learning rate α, and the rule: $x_{i+1} = x_i - \alpha \nabla f(x_0)$. This sequence continues until we approach a region close to our desired minimum. The method of gradient descent when taken continuously over infinitesimally small increments (that is, taking the limit $\alpha \to 0$) usually converges to a local minimum. However, depending on the location of the initial x_0, the local minimum achieved may not be the global minimum of the function. Furthermore, since when carrying out calculations on an unknown function we must take discrete steps (which vary in length depending on the learning rate), we are not even guaranteed a local minimum but rather may oscillate close to one, or even 'jump' past it altogether if the learning rate is too big. Still, even with these possible complications, gradient descent is a surprisingly successful method for many real life applications and is the most standard method of training for feedforward neural networks and many other machine learning algorithms. Given that our cost function indicates how poorly our neural network approximates a given function, by calculating the gradient of the cost function with respect to the weights and biases of the network and adjusting these parameters in the direction opposite to the gradient, we will decrease our error and therefore lead us closer to an adequate network

(in most cases) see .

1.2.1 Gradient calculation

Before applying the gradient descent technique, we can clearly see that the an output of 0 or 1 is problematic since the derivatives would be 0. Therefore the gradient descent technique will not work. To remedy this, following the MLE, a Birnoulli distribution has been defined on y, therefore the neural net needs to predict $\hat{y} = p(y = 1|x) = \sigma(x)$. For this number to be a valid probability, it must lie in the interval $[0, 1]$.
A good approach would ensure the existence of a strong gradient whenever the model has the wrong answer. And for consistency with the perceptron's decision rule (eq. (1.2)), a very positive linear combination of the input x has to have a probability close to 1 and vise versa (see fig. 1.4), otherwise

$$\lim_{x\to+\infty} \sigma(x) = 1, \qquad \lim_{x\to-\infty} \sigma(x) = 0. \tag{1.8}$$

This approach is based on the sigmoid function:

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

This function is suitable for the problem at hand, namely binary classification. However, depending on the output $\hat{y}$ other functions might be used. For example if a neural network is used to predict continuous non-bounded function that takes values in the interval $]-\infty, +\infty[$, then a more clever choise is the linear activation function, which is defined as: $f(x) = x$. More on that in section.

Figure 1.4: sigmoid vs step function. The two plots clearly showcast the continuiuty of the sigmoid.

Now, let us apply the gradient descent technique to our network. Our goal is to calculate the gradient of L at a point $x = (x_1, \ldots, x_n)$ given by the partial derivatives, see eq. (1.7). In addition, the property eq. (1.4) now become important. In fact, we are only going to

calculate the value of ∇L_x for a given labeled data point and then add the values of the gradient together, see below.

$$\nabla L(w,b) = \nabla(\sum_{i=1}^{n} L_x) = \sum_{i=1}^{n} \nabla L_{x_i}. \tag{1.9}$$

The error produced by each point is simply: $L_x = -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$. In order to calcualte the derivative of this error with respect to the weights, we'll first calculate $\frac{\partial}{\partial w_j}\hat{y}$, where $\hat{y} = \sigma(w \cdot x + b)$.

$$\begin{aligned}
\frac{\partial}{\partial w_j}\hat{y} &= \frac{\partial}{\partial w_j}\sigma(w \cdot x + b) \\
&= \sigma(w \cdot x + b)(1 - \sigma(w \cdot x + b)) \cdot \frac{\partial}{\partial w_j}(w \cdot x + b) \\
&= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(w \cdot x + b) \\
&= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(w_1 x_1 + \ldots + w_j x_j + \ldots w_n x_n + b) \\
&= \hat{y}(1-\hat{y}) \cdot x_j.
\end{aligned}$$

Now we can go ahead and calculate the derivative of the error L at a point x, with respect to the weight w_j.

$$\begin{aligned}
\frac{\partial}{\partial w_j} L_x &= \frac{\partial}{\partial w_j}[-y\log(\hat{y}) - (1-y)\log(1-\hat{y})] \\
&= -y\frac{\partial}{\partial w_j}\log(\hat{y}) - (1-y)\frac{\partial}{\partial w_j}(1-\hat{y}) \\
&= -y\frac{1}{\hat{y}} \cdot \frac{\partial}{\partial w_j}\hat{y} - (1-y)\frac{1}{1-\hat{y}} \cdot \frac{\partial}{\partial w_j}(1-\hat{y}) \\
&= -y(1-\hat{y}) \cdot x_j + (1-y)\hat{y} \cdot x_j \\
&= -(y-\hat{y})x_j
\end{aligned}$$

A similar calculation will show that

$$\frac{\partial}{\partial b} L_x = -(y-\hat{y})$$

Therefore, since the gradient descent step simply consists in subtracting a multiple of the gradient of the error function at every point, then this updates the weights in the following way:

$$w_i' = w_i + \alpha(y - \hat{y})x_i \tag{1.10}$$

$$b' = b + \alpha(y - \hat{y}) \tag{1.11}$$

1.3 Neural network architecture

In our work example, the target function was a simple linear function. However, in real world situations the input data is much more complex and often cannot be seperated with a line. That is where neural nets shine. Neural networks also refered to as **feedforward** neural nets or **multilayer perceptron** (MLPs) are as the name indicates are stacks of perceptrons, where each **unit** receives the input x, calcualtes the inner product with a set of weights and apply a non-linearty to the result, then these results are fed to a next layer of units that does the same calculations and so on. The overall length of the chain gives the **depth** of the model. The final layer of such a network is called the **the output layer**, whereas the intermediate layer are refered to as **hidden layers**. The goal of the feedforward network is to approximate some function f^*. The training example specify directly what the output layer must do at each point x; it must produce a value that is close to y. Therefore, the function computed after the linear combination is important. This function and the functions used in the hidden layers are refered to as **activation functions**. For example for a classifier, the fucntion maps an input x to a category y, a natural choise of activation is the sigmoid; whereas in a regression problem, where the output is continuous non-bounded that takes values in the interval $]-\infty,+\infty[$, a more clever choise is the linear activation funciton, which is defined as: $f(x) = x$. The figure below depicts the architecture described above.

For notation, set $w^n_{a,b} \in \mathbb{R}$ as the weight between the a^th unit in the $(n-1)^th$ layer with k units to the b^th unit in the n^th layer with j units.

$$w_n = \begin{pmatrix} w^n_{1,1} & \cdots & w^n_{1,k} \\ \vdots & \ddots & \vdots \\ w^n_{j,1} & \cdots & w^n_{j,k} \end{pmatrix} \tag{1.12}$$

The bias can be added as a dummy unit with input $x_{n+1} = 1$, which is a constant $b \in \mathbb{R}^j$. In order to calculate the output a_n of the n^th layer, we use the formula

$$a_n = \sigma(w_n \cdot a_{n-1} + b_n).$$

In the above equation, the activation function σ is applied element-wise to each element of the resulting vector. As the computations are carried out along the network's layers, the final function f calculated by a network of depth N is

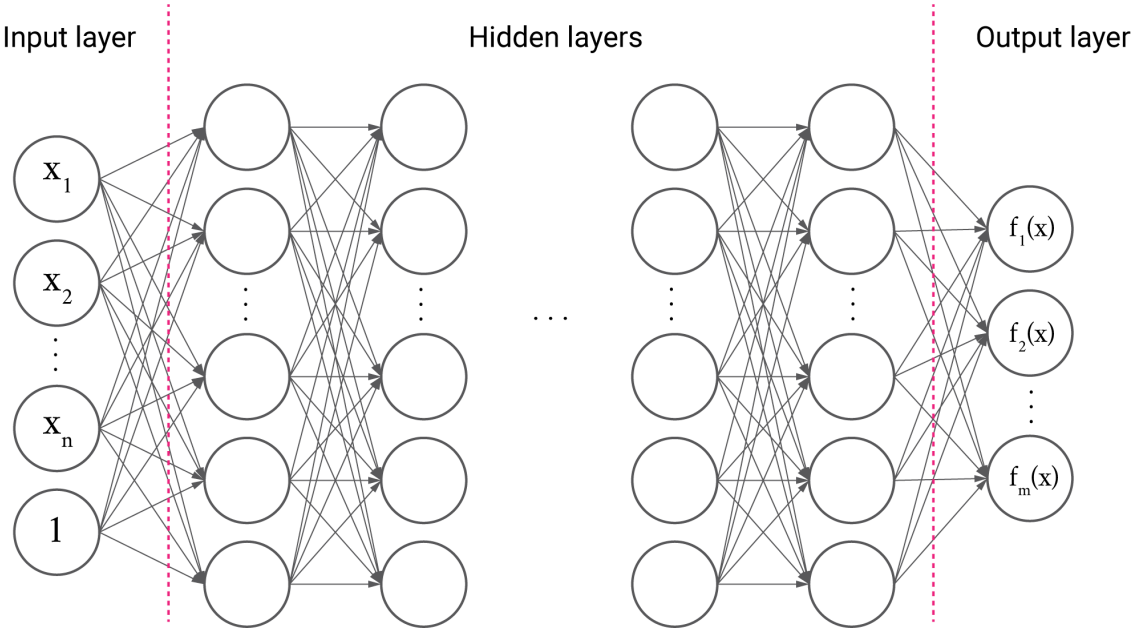


Figure 1.5: A visual representation of a feedforward network which approximates some function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ by computing the function $f^*(x) = (f_1^*(x), \ldots, f_m^*(x))$. In this approach, the network is shown as a directed weighted graph. Here $x = (x_1, \ldots, x_n)$

$$f(x) = \sigma(w_N \sigma(\ldots \sigma(w_2 \sigma(w_1 \cdot x + b_1) + b_2)) + b_N)$$

1.4 Backpropagation

In order to train a neural network, the same techniques are used as in section 1.2.1. First we difine a cost function (which is the same as in the perceptron algorithm eq. (1.6),but with a much more complex $\hat{y}$), we calculate the feedforward pass (we calculate the output $\hat{y}$), and then calculate the the gradient of the cost function L with respect to every single weight and bias in the network, we get the following gradient vector $\nabla L = (\ldots, \frac{\partial}{\partial w_{i,j}^l} L, \ldots)$. Then applying the gradient step look likes

$$w_{i,j}^{\prime l} = w_{i,j}^l - \alpha \frac{\partial}{\partial w_{i,j}^l} L$$
$$b_j^{\prime l} = b_j - \alpha \frac{\partial}{\partial b_j^l} L$$

The big challenge of applying gradient descent to neural networks is calculating these partial derivatives. This is where backpropagation comes in. This algorithm first tells us how to calculate these values for the last layer of connections, and with these results then inductively goes "backwards" through the network, calculating the partial

derivatives of each layer until it reaches the first layer of the network. Hence the name "backpropagation".

For the purpose of this section it is useful to consider the values of each layer before the activation function step. Consider

$$z_j^l = \sum_k w_{j,k}^l a_k^{l-1} + b_j^l \qquad \text{so that} \qquad a_j^l = \sigma(z_j^l).$$

Additionally, we denote the following:

$$\delta_j^l = \frac{\partial}{\partial z_j^l} L \tag{1.13}$$

This value will be useful for propagating the algorithm backwards through the network and directly related to $\frac{\partial}{\partial w_{i,j}^l} L$ and $\frac{\partial}{\partial b_j^l} L$ by the chain rule. since

$$\frac{\partial L}{\partial w_{i,j}^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{i,j}^l} = \delta_j^l a_i^{l-1} \tag{1.14}$$

$$\frac{\partial L}{\partial b_j^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l. \tag{1.15}$$

The value a_j^{l-1} has already been calculated through the forward pass. The only remaining term to calculate is δ_j^l and we obtain our gradient. Our first step is calculating this value for the last layer of the network, that is, δ_j^N for a network with N layers. Since $a_j^N = \sigma(z_j^N)$, again using the chain rule

$$\delta_j^N = \frac{\partial L}{\partial a_j^N} \frac{\partial a_j^N}{\partial z_j^N} = \frac{\partial L}{\partial a_j^N} \sigma'(z_j^N) \tag{1.16}$$

which can be easily calculated by a computer if we know how to calculate σ' (which should be true for any practical activation function).

Now we will only need to "propagate" this backwards in the network in order to obtain δ_j^{N-1}. In order to do so, apply the chain rule once again

$$\begin{aligned} \delta_j^{N-1} &= \frac{\partial L}{\partial z_j^{N-1}} \\ &= \sum_i^k \frac{\partial L}{\partial z_i^N} \frac{\partial z_i^N}{\partial z_j^{N-1}} \\ &= \sum_i^k \delta_i^N \frac{\partial z_i^N}{\partial z_j^{N-1}}. \end{aligned}$$

If we focus on the term $\frac{\partial z_i^N}{\partial z_j^{N-1}}$, we find that

$$\begin{aligned}\frac{\partial z_i^N}{\partial z_j^{N-1}} &= \frac{\partial(\sum_k w_{i,k}^N a_k^{N-1} + b_i^N)}{\partial z_j^{N-1}} \\ &= \frac{\partial(w_{i,j}^N \sigma(z_j^{N-1}))}{\partial z_j^{N-1}} \\ &= w_{i,j}^N \sigma'(z_j^{N-1})\end{aligned}$$

which, again, can be easily calculated by a computer given the network. Therefore

$$\delta_j^{N-1} = \sum_i^k \delta_i^L w_{i,j}^N \sigma'(z_j^{N-1}). \tag{1.17}$$

This formula tells us how to calculate any δ_j^l in the network, assuming we know δ^{l+1}. We finally developed a way to calculate all the δ_j^l 's, given that we know what the values of δ_j^{l+1} are. Thus, by propagating this method backwards through the layers of the network we are able to find all our desired partial derivatives, and can therefore calculate the value of ∇L as a function of the weights and biases of the network and execute the method of gradient descent.

1.5 Problems related to neural nets

Neural networks are extremely powerful function approximators, but if care is not taken during the design of architecture since there are many parameter one can tune (depth, number of units in each layer, ...). Therefore, a complex design namely high number of units in each layer and a deep network can lead to **overfitting**. Overfitting is the case where the overall cost is really small (The network is doing very well on the training set) but the generalization of the model to unseen data is poor and unreliable. There are many solutions proposed to break this effect such as dropout which consists of randomly zeroing the output of some units in each layer to force the algorithm to take different routes through the network and better generalize.

Another famous problem neural nets suffer from is **local minimum** problem, where the gradient descent optimizer might converge to a local minimum rather than the global one. But as deep learning is evolving, we now understand that at higher dimensions the chance of getting in such situation is very unlikely due to tremendous number of dimensions deep networks deals with.

Another issue in deep neural nets is the **vanishing gradients**. As we learned from back-propagation, each of the neural network's weights receive an update proportional to the partial derivative of the error function with respect to the current weight in each iteration

of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. As one example of the problem cause, traditional activation functions such as the sigmoid function have gradients in the range (0, 1), and backpropagation computes gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the "front" layers in an N-layer network, meaning that the gradient decreases exponentially with N while the front layers train very slowly. To remedy this problem other activation functions might be used in the hidden layers. The behavior of the hidden layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f. Therefore the choise of the activation function in those layers is irrelevant, which makes the use of other activations possible. Many functions have been proposed to escape the trap of vanishing gradients, namely the ReLU function is of popularity in deep learning. The ReLU stands for rectified linear unit defined as $ReLU(x) = \max(0, x)$.

1.6 batch and stochastic gradient descent

Batch gradient descent is just anoter name for the gradient descent discussed so far. It involves calculations over the full training set to take a single step as a result of which it is very slow on very large training data due to the size of the weight matrices that take up large memory portions. Thus it become very computationally expensive to do batch GD. One can take advantage of the property mentioned on section 1.1, eq. (1.4). Therefore, instead going through the entire dataset at each iteration we select a few elements from the training set, commonly selected by randomly sampling from all the available labeled data, calculate the gradient, update the network's weights and repeat the process until the network arrives at satisfactory results. The gradients computations are faster as there is much fewer data to manipulate in a single time. This technique is refered to as **stochastic** gradient descent. One downside though of SGD is, once it reaches close to the minimum value then it doesn't settle down, instead bounces around which gives us a good value for model parameters but not optimal which can ve solved by reducing the learning rate at each step which can reduce the bouncing and SGD might settle down at global minimum after some time.

Chapter 2

Neural network Variant: Convolutional Neural Networks

Convolutional networks, also known as convolutional neural networks, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [5].

2.1 The convolution operation

The convolution operation is well known in the engineering terminology, which, in its most general form, is an operation on two functions of a real-valued argument. defined as:

$$s[n] = y[n] * x[n] = \sum_{k=-\infty}^{k=\infty} y[k]x[n-k] \tag{2.1}$$

We are interested in the discrete convolution operation, since data on a computer is presented as discrete values rather than continuous signals. The eq. (2.1) presented above is for discrete time signals.

In convolution neural network terminology, the first argument to the convolution is often referred to as **the input**, and the second argument as **the kernel**. The output is sometimes referred as the **feature map**. The input is usually a multidimensional array of data (RGB images), and the kernel is usually a multidimensional array of parameters

that are adapted by the learning algorithm. These multidimensional arrays are referred as tensors. Finally, we often use convolution over more than one axis at a time. For example if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$S[m,n] = I[m,n] * K[m,n] = \sum_i \sum_j I[i,j]K[m-i,n-j]. \tag{2.2}$$

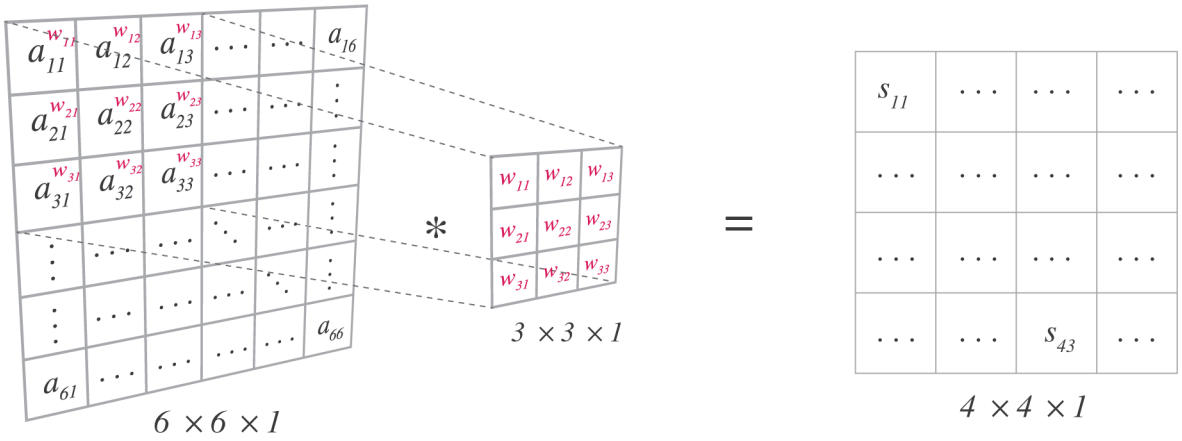
Convolution is commutative, meaning we can equivalently write:

$$S[m,n] = K[m,n] * I[m,n] = \sum_i \sum_j I[m-i,n-j]K[i,j]. \tag{2.3}$$

While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

$$S[m,n] = I[m,n] * K[m,n] = \sum_i \sum_j I[m+i,n+j]K[i,j]. \tag{2.4}$$

Many machine learning libraries implement cross-correlation but call it convolution. See fig. 2.1 for an example of convolution applied to a 2d tensor (gray-scale image).



$$s_{11} = a_{11}\,w_{11} + a_{12}\,w_{12} + a_{13}\,w_{13} + a_{21}\,w_{21} + a_{22}\,w_{22} + a_{23}\,w_{23} + a_{31}\,w_{31} + a_{32}\,w_{32} + a_{33}\,w_{33}$$

$$s_{43} = a_{43}\,w_{11} + a_{44}\,w_{12} + a_{45}\,w_{13} + a_{53}\,w_{21} + a_{54}\,w_{22} + a_{54}\,w_{23} + a_{63}\,w_{31} + a_{64}\,w_{32} + a_{65}\,w_{33}$$

Figure 2.1: An example of 2-D convolution

2.2 Convolution networks architecture

Convolution leverages three important ideas that can help improve a machine learning system: sparse connectivity, parameter sharing and equivariant representations.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit, see fig. 2.2. Convolutional networks, however, typically have sparse interactions (also referred to as **sparse connectivity** or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters, and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m [5]. For graphical demonstrations of sparse connectivity, see fig. 2.3 and fig. 2.4.

Rearranging each vector as a matrix, the relationship between the nodes in each layer are more obvious, see fig. 2.4.

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere fig. 2.2. That is the reason, traditional nets are referred as to Fully connected networks or Dense networks.

In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. In fig. 2.4, each of the color coded image quarters are connected to a single color coded node in the next layer. All of these connections have exactly the same shared weights, see fig. 2.1, the weights w_{11} through w_{33} do not change as the filter slides through the

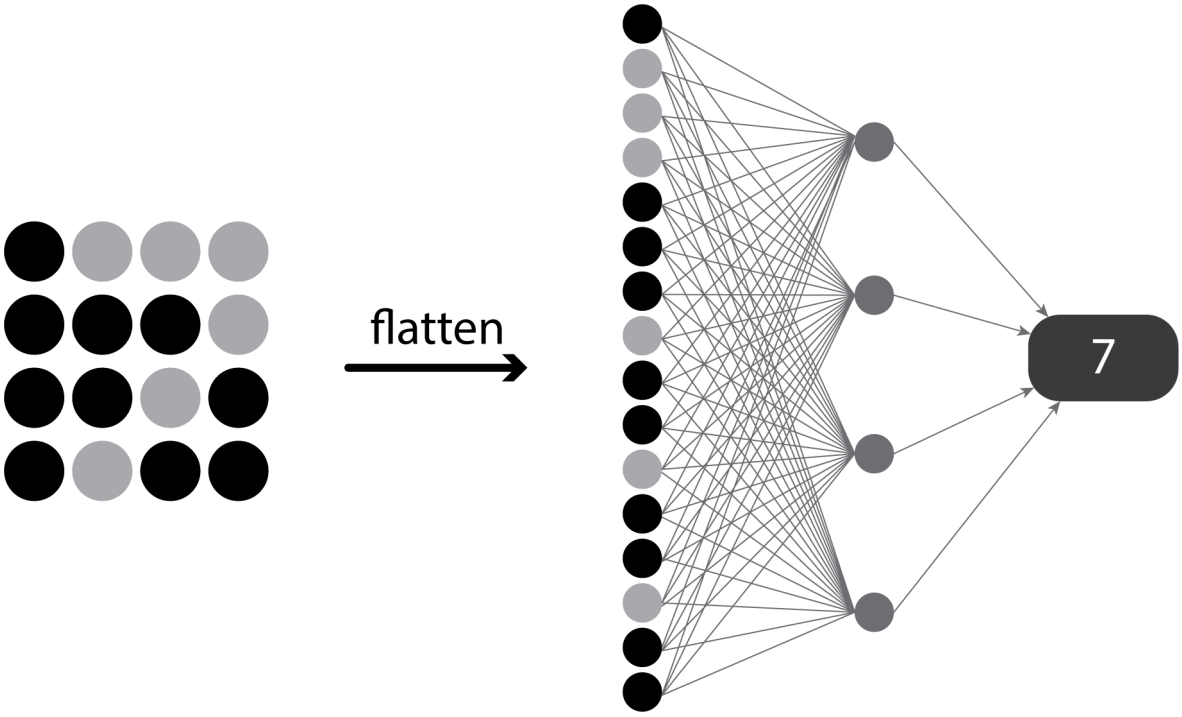


Figure 2.2: Traditional neural network connections. The last layer has been replace by a black box for simplicity

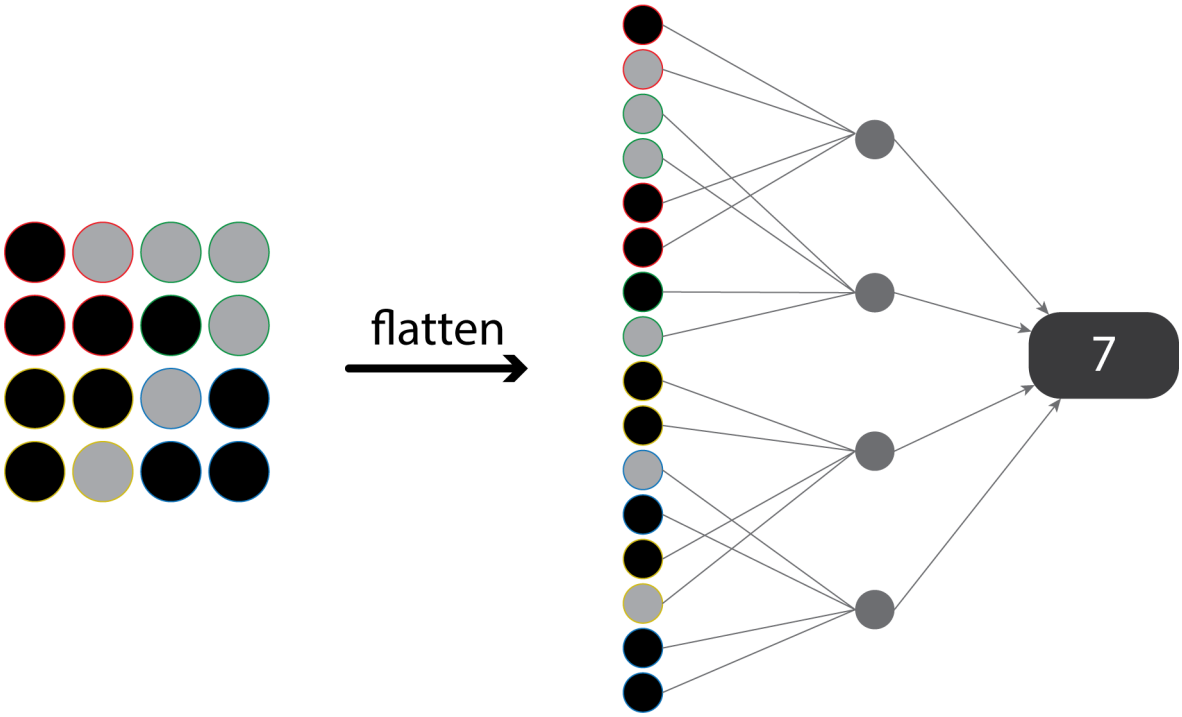


Figure 2.3: Sparce connectivity

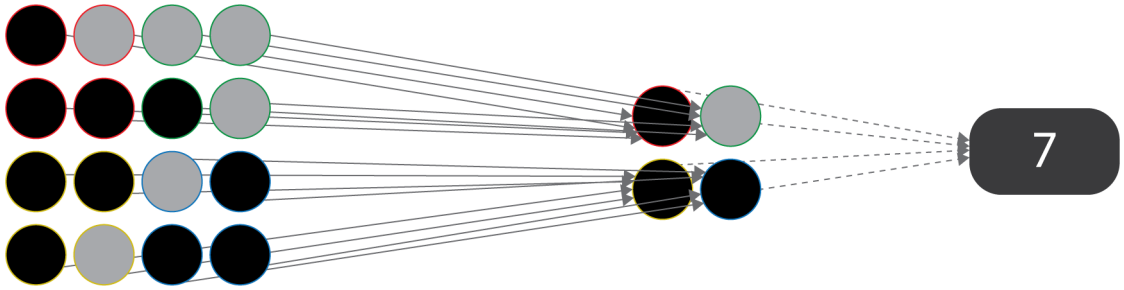


Figure 2.4: Sparce connectivity after rearrangement

image. This does not affect the runtime of forward propagation—it is still $O(k \times n)$—but it does further reduce the storage requirements of the model to k parameters. The particular form of parameter sharing causes the layer to have a property called **equivariance to translation**.

To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, that is, shifts it, then the convolution function is equivariant to g. For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that $I' = g(I)$ is the image function with $I'(x,y) = I(x-1,y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I, then apply convolution, the result will be the same as if we applied convolution to I, then applied the transformation g to the output. With images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image.

Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations. To illustrate these principles in action, we shall use a hand picked filter that used to detect edges in a image, see below.

$$K = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

These filters are called high pass filters. They enhance high frequency components in

an image. Frequency in images just like in signals is the rate of change of the intensity, which areas in neighboring pixels that rapidly changes for example from very dark to very light (in grayscale images). See fig. 2.5 to see the effect of applying the above filter to a grayscale image.

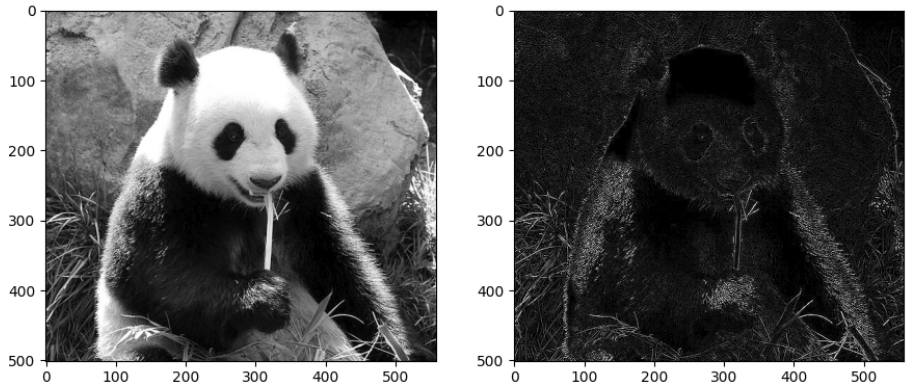


Figure 2.5: 2D convolution. Where there is no change or little change of intensity in the original picture, the high pass filter block those areas out and turn the pixels black. But in the areas where a pixel is way brighter than its immediate neighbors, the high pass filter enhance the change and create a line. This has the effect of emphasizing edges. Edges are just areas in an image where the intensity changes very quickly. This images has been obtain by convolving the filter K with the image in the left, as we can see the three principles discussed above apply to this filter. The values of K didn't change while convolving (shared parameters). Space connectivity where the filter looks only to a small portion of the image at a time. And the equivariant translation, where we clearly see that no matter the position of the edge in the image the filter successful highlight it.

2.3 Convolutional layer

The convolutional layer is produced by applying a series of many different image filters, also known as convolutional kernels, to an input image.

In the example shown, 4 different filters produce 4 differently filtered output images. When we stack these images, we form a complete convolutional layer with a depth of 4. See fig. 2.7.

In case of colored images, computer interprets them as 3D-tensor $(Height \times width \times channels)$. Here channels are the RGB channels. When performing convolution, the kernel K is itself chosen to be three dimensional as well. A typical kernel K would be $3 \times 3 \times 3$. The resulting output feature map would be $(Height \times Width)$. In order to depict multiple patterns in the image, instead of having a single kernel, multiple kernel are defined. Now each resulting output feature map can be considered as an image channel and stack them to get a 3 dimensional array. The latter 3D array can be used as input to another convolutional layer to discover patterns within the patterns that we discovered in the first convolutional layer. This operation can be repeated multiple times to discover various patterns within

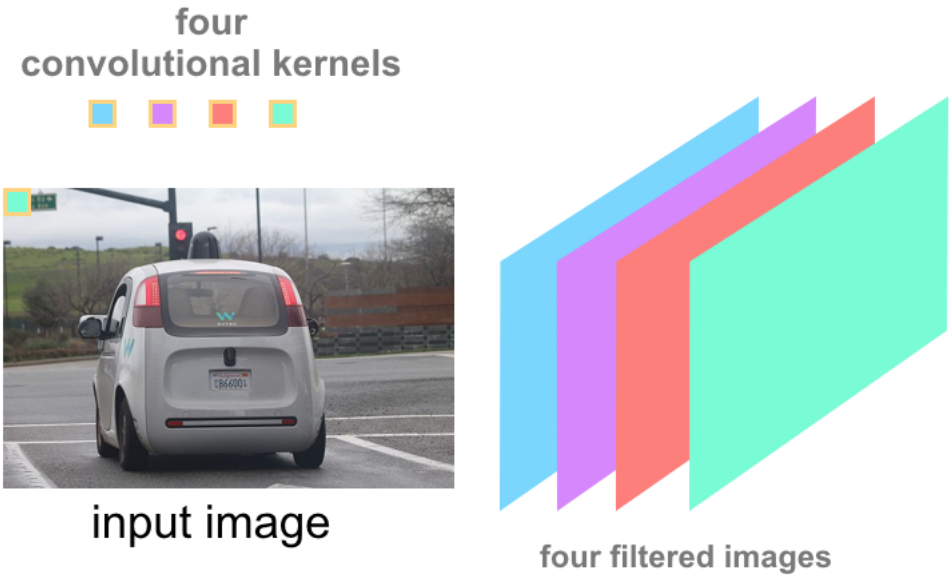


Figure 2.6: Multiple filters for mutiple pattern detection

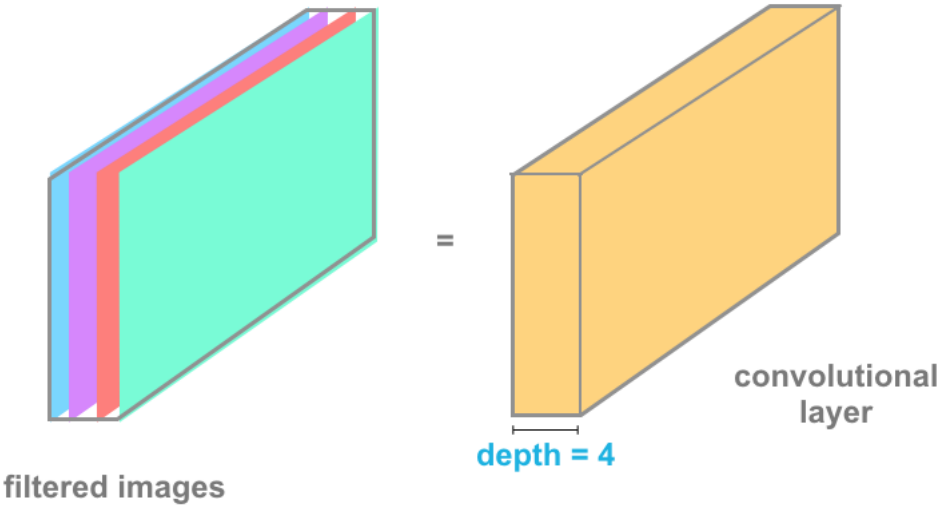


Figure 2.7: A complete convolutional layer with 4 filters

the input image.

In CNNs, inference works the same way as old plain neural network. Both convolutional and Dense layers have weights and biases and initial randomly generated. Therefore, in the case of CNNs where the weights take the form of convolutional kernel or filters, those kernels are randomly generated and so are the patterns that they're initially designed to detect. As with Fully connected networks, when we construct a CNN, we will always specify a loss function. In the case of multiclass classification, this will be categorical cross-entropy loss(equ from chapter 1). Then as we train the model through back propagation, the filters are updated at each iteration to take on values that minimizes the loss function. In other words, the CNN determines what kind of patterns it needs to detect base on the loss function.

2.4 Stride and padding

The behavior of a convolutional neural network can be controlled by specifying the number of filters and the size of each filter, these are referred to as **hyper-parameters**. For instance, to increase the number of nodes in a convolutional layer, you could increase the number of filters. To increase the size of the detected patterns, you could increase the size of the filters. But there are more hyper-parameters than we can tune. One of these hyper-parameters is referred to as the stride of the convolution. The stride is just the amount by which the filter slides over the image. In the previous example fig. 2.1, the stride was one. We move the convolution window horizontally and vertically across the image one pixel at a time [1]. The width and height of the output of the convolution is given by eq. (2.5), if the input image is $n \times n$, with a filter $f \times f$:

$$n - f + 1 \times n - f + 1 \tag{2.5}$$

If we introduce the stride parameter s, eq. (2.5) can be rewritten as follow:

$$\lfloor \frac{n-f}{s} \rfloor + 1 \times \lfloor \frac{n-f}{s} \rfloor + 1 \tag{2.6}$$

One downside of the convolution operation is the shrinking input dimensions. Indeed, according to eq. (2.5), the input dimension shrinks each time by few pixels which can be an undesirable effect in very deep networks, where the image can shrink to very small dimensions. Another downside of the convolution is, the top left pixel (or corners of an image in general) is only involved in one pass of the filter, whereas if we take a pixel in the middle, then many 2×2 regions will overlap that pixel. It as if the pixels at the corners are used much less in the output, so information is thrown away near the edge of the image. Therefore to solve both of this problems, before applying the convolution we can pad the image with additional boarders, for instance 1 pixel, see fig. 2.8. Therefore the width and

height of the output feature map is calculated as:

$$\lfloor \frac{n-f+2p}{s} \rfloor + 1 \times \lfloor \frac{n-f+2p}{s} \rfloor + 1 \tag{2.7}$$

Now with this additional boarder of zeros, the output feature maps' dimensions can be made equal to the input's dimension by setting the appropriate padding value. And the corner pixels contribute more in the output feature map.

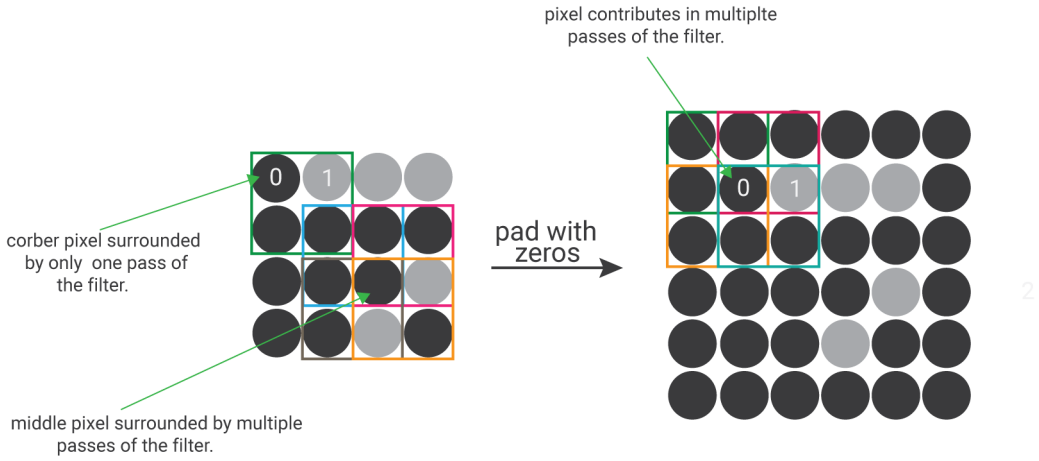


Figure 2.8: Padding example.

2.5 Pooling

Pooling function is the next type of layer in convolutional neural networks. It replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include average pooling of a rectangular neighborhood [5]. see figure on how to perform max pooling.

In all cases, pooling helps to make the representation approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is.For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. Another improvement that pooling brings is the computational efficiency of the network. The reason being is that pooling reports summary statistics for regions spaced with stride s (typically 2 is used), therefore the next layer has roughly s times fewer inputs to process and reduces the memory requirements for storing parameters [5].

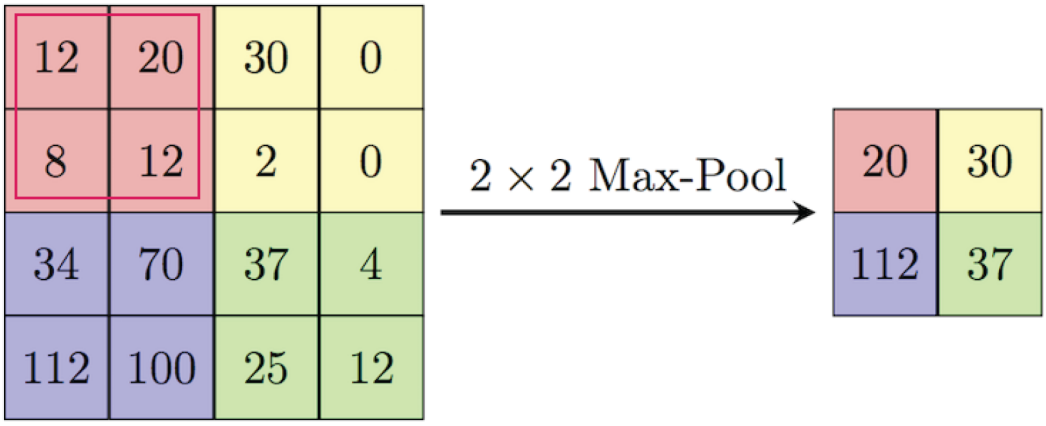


Figure 2.9: Maxpooling example. As in the convolution operation, we slide a window across the image typically a 2×2 window. The value of the corresponding node in the max pooling layer is calculated by just taking the maximum of the pixels contained in the window. The pooling function is applied independently on every feature map in the input stack. The output is a stack with same number of feature maps with width and height reduced by a factor of two.

Therefore, most CNNs are composed of only those two layers: Pooling and convolution. We begin with convolution layers which detects regional patterns in an image using a series of filters. Typically, just like fully connected networks, an activation function is applied to the output feature maps. RelU activation function is used as it has proven to be extremely efficient in object classification tasks. Then pooling layers follow the convolutional layers to reduce the dimensionality of their input tensors. CNNs are designed with the goal of taking an input image and gradually making it much deeper than it is tall or wide. As the network gets deeper, it is actually extracting more and more complex patters and features that help identify the content and objects in an image. CNNs are usually referred to as **feature extractors**. Another issue that rises when training CNNs, is the input image dimensions. Since training requires large datasets of thousands of images, it no surprise that these images are of different sizes and shapes. Therefore CNNs requires a fixed sized input due to batch training. Indeed, instead of passing one image at a time through the network, we usually pass batches of images which are just stacks of images. But in order to do that, all the images have to have the same width and height. So, we have to pick an image size and resize all of our images to that same size before doing anything else.

2.6 Transfer learning

Usually training very deep networks from scratch is a very tedious task; huge datasets are required for the task to better generalize to real life situations. Modern CNNs usually take 2-3 weeks to train across multiple GPUs. However, it has been revealed that deep networks trained on natural images exhibits a curious phenomenon in common: on the first layer

they learn general features similar to color blobs and edges. Such first layer features appear not to be *specific* to a particular dataset or task, but *general* in that they are applicable to many datasets and tasks [6]. This means it may be useful to transfer this knowledge to other similar tasks. This technique is refered to as **transfer learning**. Deep CNNs are good condidates for this task because they are usually trained on general tasks (like image classification of daily life objects) and have many adjustable layers. As [6] states the transferability of features decreases as the distance between the base task and target task increases, but that transferring features even from distant tasks can be better than using random features. A final surprising result is that initializing a network with transferred features from almost any number of layers can produce a boost to generalization that lingers even after "fine-tuning" to the target dataset. One of the strategies used when using transfer learning is refered to as **fine-tuning**. This simply means retraining the whole or parts of the pretrained CNN. This is done by retraining with the new dataset without changing the architecture or reinitialize the weights. The existing weight are said to be *fine-tuned* to the new task at hand [7].

Chapter 3

CNN application: Object detection

The concept of convolution and convolutional neural networks has been applied to many real life problems: including object classification object detection, speech recognition, disease depiction in medical images, self driving cars, and many more. In this is chapter, we will focus on present the state-of-the-art detection systems: YOLO object detection which stands for you only look once and R-CNN which stands for Region-CNN. Object detection is the task of detecting, meaning classifying and localizing instances of semantic objects of a certain class (in our case algerian car plates along with their digits). An object detection algorithm should not only be able to classify an object but as well as localizing it in an image by drawing a bounding box around it. See fig. 3.1.

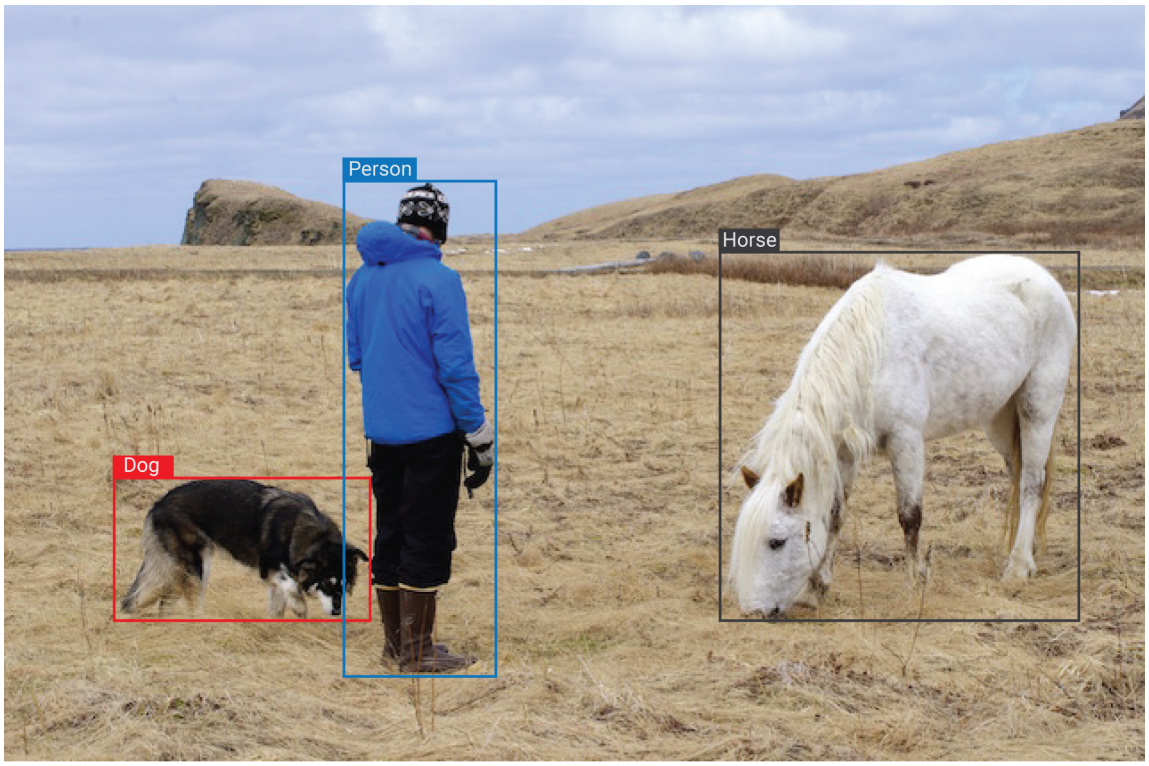


Figure 3.1: Example of what an object detection system should accomplish

3.1 YOLO: you only look once

Over the past few years, the YOLO algorithm have evolved quite a lot going from YOLOv1 all through version four. The different improvements that this algorithm went through are just the fruits of many research developements in the deep learning field incorporated into it to make it more robust and less prone to errors. In this section we shall present the version three of YOLO. Version four has only been developed in april 2020 during the middle of the pandemic. Many techniques have been included in this last paper which makes a bit difficult since we have to go through all the new details. Therefore we shall only present version three which we already have a solid background of.

3.1.1 Bounding boxes

The YOLO algorithm divides the input image into an $S \times S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object [10]. Each grid cell predicts B bounding boxes, using anchor boxes. Anchor boxes are predefined boxes of certain width and height. Those boxes are defined to capture the scale and aspect ratio of specific object classes you want to detect. They are typically chosen based on object sizes in the training data [2], see . Anchor boxes have been introduced to solve two issues (second issue discussed in section 3.1.2). Objects in the YOLO algorithm are associated with grid cells that their centers fall into. If two objects' centers fall into the same grid cell we wont be able to predict both objects. Therefore, we can associate each grid cell with multiple anchor boxes each responsible to detect only one object in that cell. A typical number of boxes used is three. See fig. 3.2. Each bounding box is associated with a confidence score, which reflects how confident the network is that the bounding box contains an object (also called objectness) [10]. This should be ideally 1 if there is an object otherwise 0 [9].Then $b_x,\ b_y,\ b_h,\ b_w$ that defines the bounding box, where b_x and b_y represents the box's center coordinates and $b_h,\ b_w$, the height and width respectively [3]. And the class confidence scores. For instance if we are building a self driving car object detection system, we may want to detects cars, pedestrians and motorcycles. Therefore, each grid cell will be associated with an $((5 + number\ of\ classes\ to\ detect) \times number\ of\ anchor\ boxes)$ dimensional vector. See fig. 3.2.

3.1.2 Network design

The network is a series of convolutional and pooling layers chosen so that the network eventually maps the input image $W \times H \times 3$ to an output volume $S \times S \times ((5 + number\ of\ classes\ to\ detect) \times number\ of\ anchor\ boxes)$. YOLO's convolutional layers down-sample the image by a factor of 32 Now, to train the convolutional neural network, we pick an image size of 416×416. This number has been chosen because we want an odd

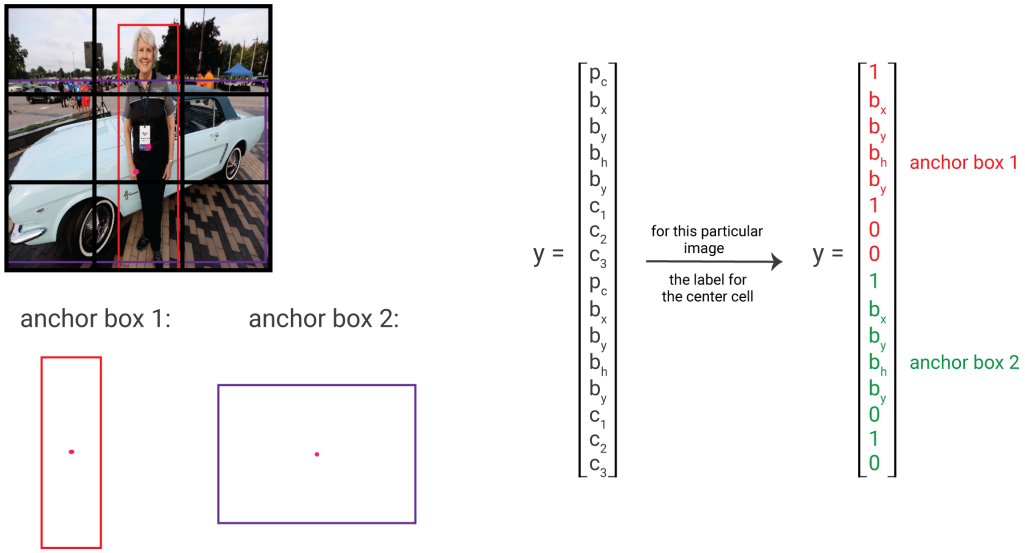


Figure 3.2: Example of anchor boxes. As we can see on the figure, the anchor boxes capture the scale and aspect ratio of cars and pedestrians. Indeed, most cars and humans will have approximately the same scale and aspect ratio. The vector y is composed of the objectness score as well as the bounding boxes and the class probabilities repeated for each anchor box. Here two anchor boxes have been used. YOLOv3 uses 3 anchor boxes. The image has been divided into a 3×3 grid just for illustration. The vector y represents the manual labeling for the central cell. Anchor box 1 is associated with the pedestrian while the second one is associated with the car.

number of locations in our feature map so there is a single center cell. Objects, especially large objects, tend to occupy the center of the image so it's good to have a single location right at the center to predict these objects instead of four locations that are all nearby [8]. so by using an input image of 416 we get an output feature map of 13×13. The second issue anchor boxes address is the training instability [8]. In fact, during the early epochs of training if b_x *and* b_y are randomly initialized, the network struggles to converge to the right ground truth box's center. To overcome this problem, YOLO predicts location coordinates b_x *and* b_y relative to the grid cell. This bounds the ground truth to fall between 0 and 1. We use sigmoid activation to constrain the network's prediction to fall in this range. The network predicts B bounding boxes at each cell in the output feature map. The network predicts 5 coordinates for each bounding box t_x, t_y, t_h, t_w *and* t_0, see fig. 3.3. If the cell is offset from the top left corner of the image by $(c_x; c_y)$ and the anchor box has width and height pw, p_h, then the predictions correspond to [8]:

$$b_x = \sigma(t_x) + c_x \tag{3.1}$$

$$b_y = \sigma(t_y) + c_y \tag{3.2}$$

$$b_w = p_w e^{t_w} \tag{3.3}$$

$$b_h = p_h e^{t_h} \tag{3.4}$$

Since we constrain the location prediction the parametrization is easier to learn, making the network more stable [8], see fig. 3.4. The question that naturally rises is: How, at the beginning, do we get $p_w,\ p_h$? Otherwise, how to assign an anchor box to a ground truth object ? The answer to this question is given is section 3.1.3 as we need to define an important function (IoU) to proceed.

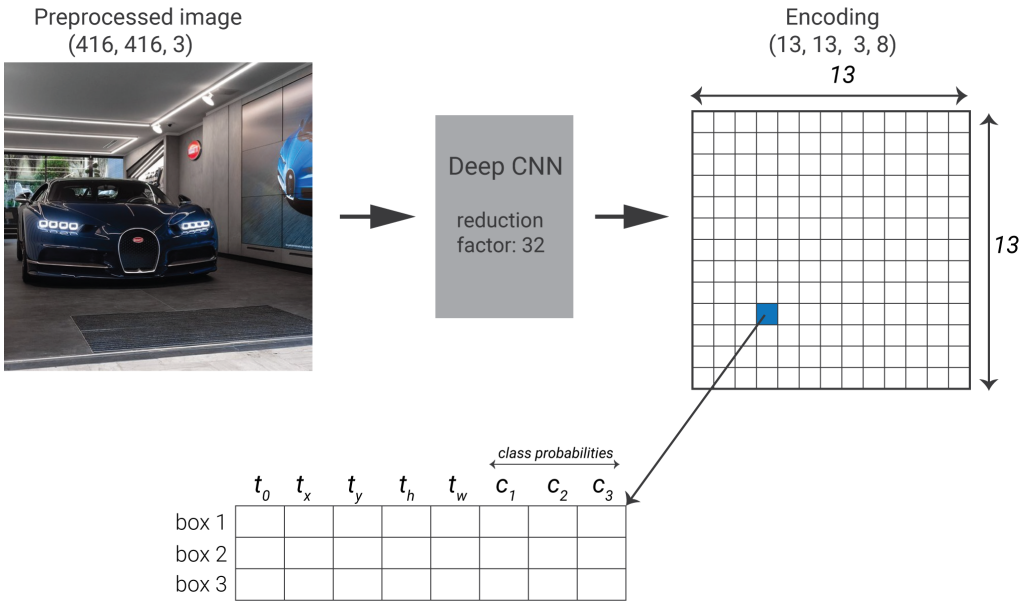


Figure 3.3: The true output of YOLOv3 after introducing the training instability issue. The net works output a $13 \times 13 \times (8 \times 3)$ in this case, or simply put $13 \times 13 \times 24$ output volume. Each grid cell outputs three bounding boxes.

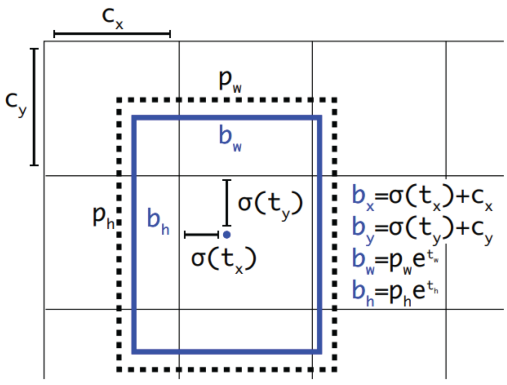


Figure 3.4: Bounding box calculation. We predict the width and height of the box as offsets from manually chosen anchor boxes. We predict the center coordinates of the box using a sigmoid function.

During training we optimize the following, multi-part loss function. As we can see the first sum is over scales, meaning different regions of the network. Indeed the network used in YOLOv3 does have only one output but three. The architecture of the network is discussed further in appendix.

$$\sum_{scales} \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{obj} \left[(t_x - \hat{t}_x)^2 + (t_y - \hat{t}_y)^2 + (t_w - \hat{t}_w)^2 + (t_h - \hat{t}_h)^2 \right]$$
$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{obj} \left[-\log(\sigma(t_o)) + \sum_{k=1}^{C} BCE(\hat{y}_k, y_k \right]$$
$$+ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{i,j}^{noobj} \left[-log(1 - \sigma(t_o)) \right] \quad (3.5)$$

where $1_{i,j}^{obi}$ denotes if object appears in cell i and that the j^{th} anchor box in cell i is "responsible" for that prediction. If an anchor box is not assigned to a ground truth object it incurs no loss for coordinate or class predictions, only objectness. In cells that contain an object, the bounding box coordinates are calculated using the sum-squared loss function. Each box predicts the classes the bounding box may contain using multilabel classification. In other words, binary cross-entropy loss is used (BCE). The same binary cross-entropy loss is used to for objecness prediction as eq. (3.5) states it.

3.1.3 Processing the algorithm's output

After training, the network at inference time will find multiple detections. In fact, for each cell in the $S \times S$ grid, using B anchor boxes, the algorithm will infer B bounding boxes for each cell, which makes a total of $B \times S^2$. Therefore an object can be detected multiple times. **Non-max suppression** is an algorithm that cleans up those detections and makes sure each object gets detected only once. Before discussing it though, let us introduce an important function called **Intersection over Union** (IoU for short) that calculates how much a box or a rectangle overlaps another. So, IoU calculates the area defined by the intersection of the two boxes and divide it by the area defined by their union. See fig. 3.5.

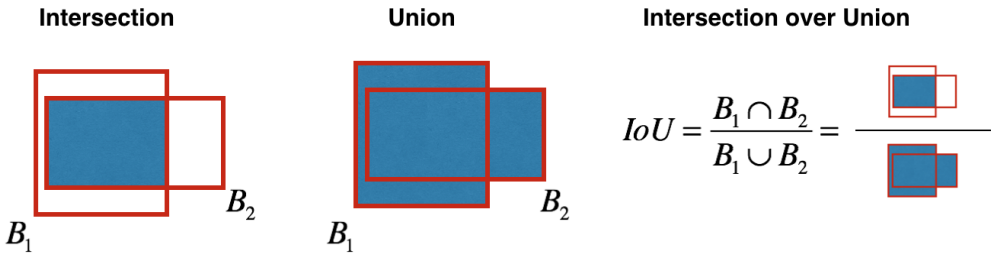


Figure 3.5: Intersection over union.

IoU is an evaluation metric used to measure the accuracy of an object detection system on a particular data set. Indeed, most object detection algorithm will judge a detection to be correct if the IoU between the ground truth box and the detected box is more than 0.5,

see fig. 3.6. We often see this evaluation metric used in object detection challenges such as the popular PASCAL VOC challenge [4].

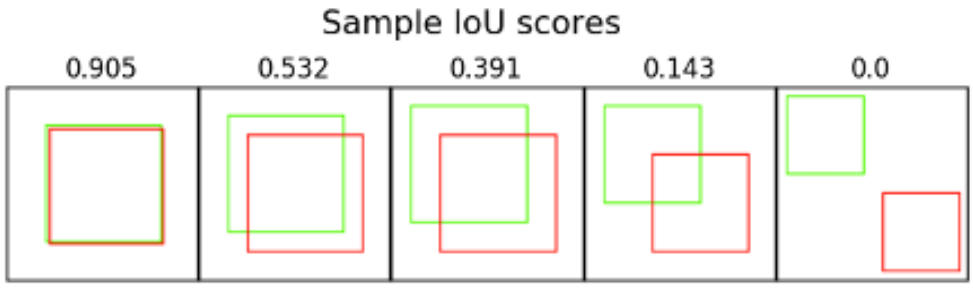


Figure 3.6: Sample IoU scores.

Back to our original question, how non-max suppression works. First, all the boxes having an $objectness \times the\ class\ probability$ less or equal than some threshold are discarded (typical value used is .6). While there any remaining boxes, we pick the box with the largest $objectness \times the\ class\ probability$ and output it as a prediction. Then we discard an remaining box with $IoU \geq 0.5$ with the box outputed in the previous step. This algorithm ensures that each object is detected only once.

In section 3.1.2 we discussed how the bounding boxes are being computed, and we finished it with a question: How does the anchor boxes being assigned to ground truth objects at the beginning? YOLOv3 assigns the anchor with the highest Intersection-over-Union (IoU) overlap with a ground truth box.

3.2 Faster R-CNN

Several object detection techniques and models have been developed over the years. Each with its benifits and drawbacks. In this section we shall explore the faster region-CNNs technique to tackle this task.

Faster R-CNN model is composed of two networks: region proposal network (RPN) for generating region proposals and a network using these proposals to detect objects. The main different here with its' predecessor Fast R-CNN is that the later uses an algorithm called "selective search" to generate region proposals. The time cost of generating region proposals is much smaller in RPN than selective search, since the RPN network does a significant part of computation which is overlaping with the computation needed for the object detection network. in short, RPN ranks region boxes (called anchors) from most likely to less likely to contain an object and proposes the ones most likely containing

objects. The architecture is as shown in fig. 3.7.

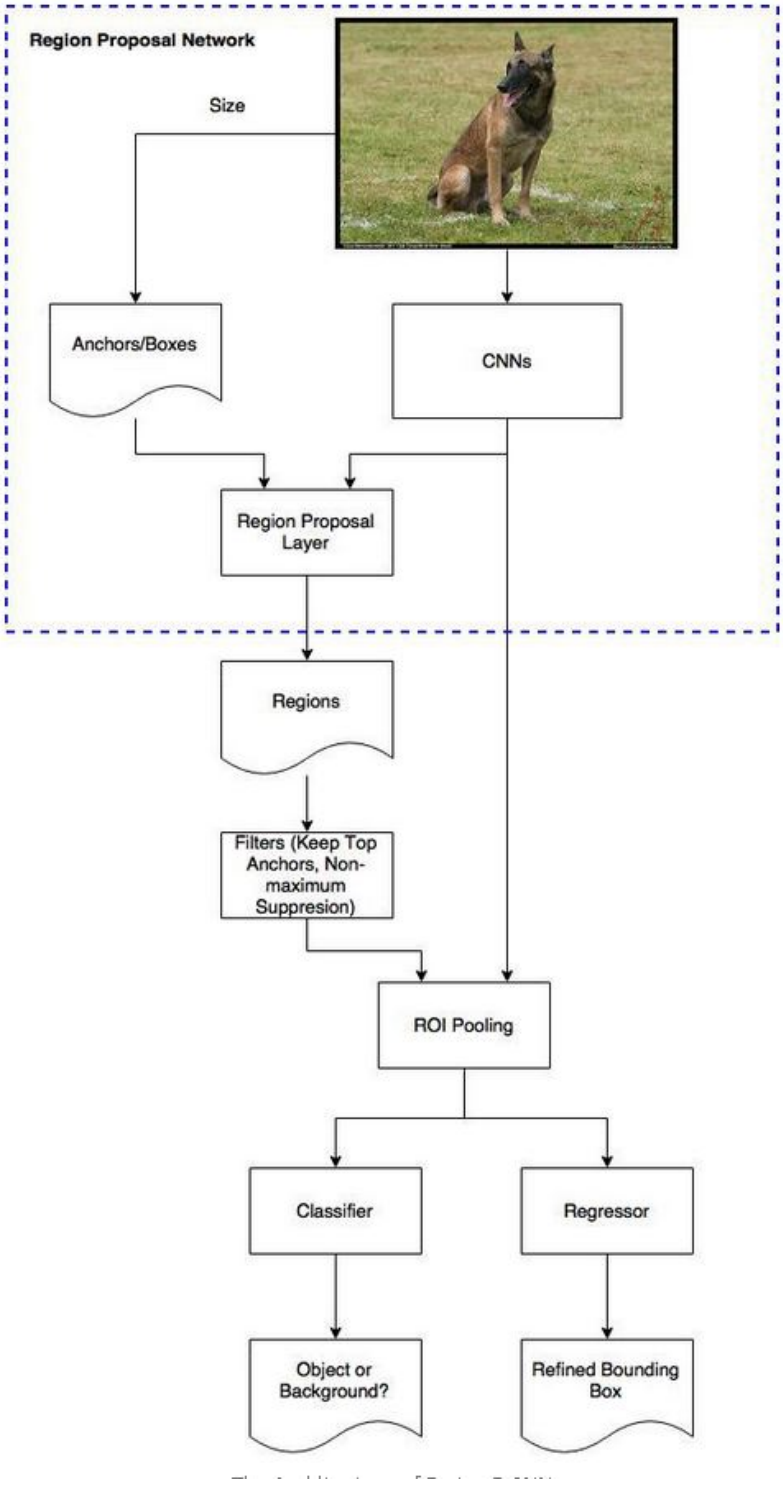


Figure 3.7: rcnn1

3.2.1 Anchors

In the default configuration of Faster R-CNN, it considers 9 anchors at each position of an image. fig. 3.8 shows 9 anchors at the position $(320, 320)$ of an image with size $(600, 800)$.

The colors represent three scales or sizes: $128 \times 128,\ 256 \times 256,\ 512 \times 512$. And in each color we have three boxes that have height width ratios $1:1,\ 1:2$ and $2:1$ respectively. These two parameters are called "scales" and "aspect ratios", and they have a significant effect on the performance of our model. The RPN selects a position in a given image at every stride of 16 where it generates those 9 anchors. In an image of the same size as fig. 3.8 there will be 1989 (39×51) positions. This leads to 17901 (1989×9) boxes to consider. This number of anchors is hardly smaller than the technique of of sliding window and pyramid. The advantage here is that we can use region proposal netowrk, to significantly reduce number of boxes that will be considered by the classifier network.

These anchors work well for Pascal VOC dataset as well as the COCO dataset. However you have the freedom to design different kinds of anchors/boxes. For example, you are designing a network to detect passengers/pedestrians, you may not need to consider the very short, very big, or square boxes. A uniform set of anchors may increase the speed as well as the accuracy.

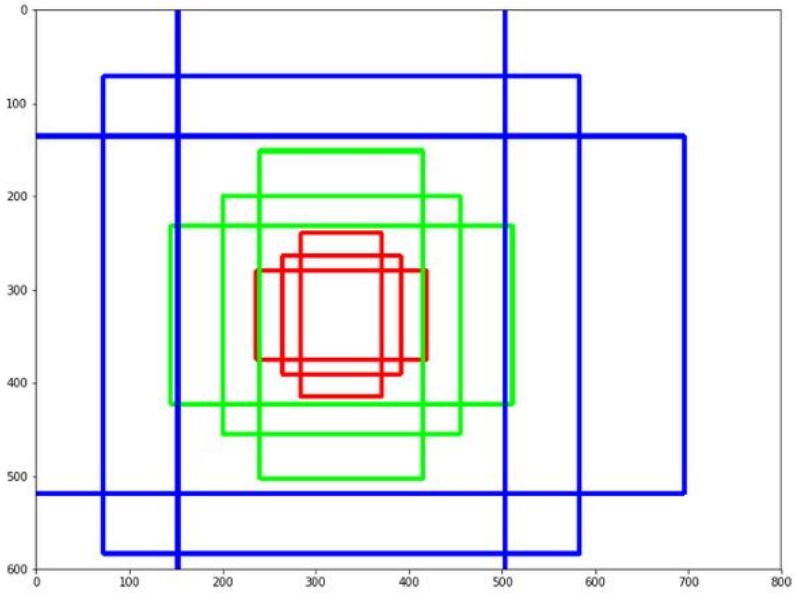


Figure 3.8: rcnn2

3.2.2 Region Proposal Network

The input to the RPN module is the feature map of an image, the RPN then generates centers on the original image for each "pixel" in a feature map obtained from a forward pass through a pretrained CNN. It then generates 9 anchors around each center according to the specified scales and aspect ratios. The output of a RPN is a set of probabilities for each anchor that determine the probability of a certain anchor being an object or not. It also outputs a set of error estimations for the anchors which overlap with a ground truth box. these outputs will be examined by a classifier and regressor to eventually check

the occurrence of objects. To be more precise, RPN predicts the possibility of an anchor being background or foreground, and refines the dimensions of an anchor. Since the RPN performs a classicfication task, it will go through a training process for which we must have a clear definition of the dataset and the labels. In this case our dataset is the anchors defined for each image. As for the labels; the basic idea is that we want to label the anchors having the higher overlaps with ground-truth boxes (the bounding box surrounding the object we wish to detect)as foreground, and the ones with lower overlaps as background. For this we use the IOU (Intersection Over Union) function. If the value of the IOU is higher than a certain threshold then it would be labeled as forground otherwise it is labeled as background.

RPN also performs a regression task on the same anchors in order to correct the dimensions and location of these same anchors. For each anchor it computes an estimation of error on the dimensions called t_w for the width and t_h for the height as well as on the location of the anchor center t_x and t_y such that

$$t_w = log(\frac{w}{w_a}) \tag{3.6}$$

$$t_h = log(\frac{h}{h_a}) \tag{3.7}$$

$$t_x = \frac{x - x_a}{w_a} \tag{3.8}$$

$$t_y = \frac{y - y_a}{h_a} \tag{3.9}$$

x, y, w, h are the groud truth box center coordinates, width and height. $x_a, y_a, h_a and w_a$ and anchor boxes center cooridinates, width and height.

The final and most important component of the training process is the loss function

$$L(p_i, t_i) = (\frac{1}{N_{cls}}) \sum_i L_{cls}(p_i, p_i^*) + \lambda (\frac{1}{N_{reg}}) \sum_i p_i^* L_{reg}(t_i, t_i^*) \tag{3.10}$$

where p_i is the predicted probability of objectness and p_i^* is the actual score. t_i and t_i^* are the predicted cooridinates and actual coordinates respectively. The ground-truth label p_i^* is 1 if the the anchor is positive and 0 if the anchor is negative.

3.2.3 ROI Pooling

Region of interest pooling (also known as RoI pooling) purpose is to perform max pooling on inputs of non-uniform sizes to obtain fixed-size feature maps (e.g. 7×7). This layer takes two inputs :

- A fixed-size feature map obtained from a deep convolutional network with several convolutions and max-pooling layers
- An $N \times 5$ matrix of representing a list of regions of interest, where N is the number of RoIs. The first column represents the image index and the remaining four are the co-ordinates of the top left and bottom right corners of the region.

For every region of interest from the input list, it takes a section of the input feature map that corresponds to it and scales it to some pre- defined size (e.g., 7×7). The scaling is done by:

- Dividing the region proposal into equal-sized sections (the number of which is the same as the dimension of the output)
- Finding the largest value in each section
- Copying these max values to the output buffer.

The result is that from a list of rectangles with different sizes we can quickly get a list of corresponding feature maps with a fixed size. Note that the dimension of the RoI pooling output doesn't actually depend on the size of the input feature map nor on the size of the region proposals. It's determined solely by the number of sections we divide the proposal into. What's the benefit of RoI pooling? One of them is processing speed. If there are multiple object proposals on the frame (and usually there'll be a lot of them), we can still use the same input feature map for all of them. Since computing the convolutions at early stages of processing is very expensive, this approach can save us a lot of time. The fig. 3.9 below shows the working of ROI pooling.

This will be the input to a classifier network, which is a copy of the pretrained backbone network we used to obtain the feature map, and which will be refered to as "Fast RCNN classifier network". This network will further branch out to a classification head and regression head. The loss function for the Fast-RCNN network is defined in the same way as the RPN loss, Except for the significance of the variables. p_i is the predicted class scores for every class of objects we want to detect and p_i^* is the actual score. t_i and t_i^* are the predicted cooridinates and actual coordinates, respectively. The ground-truth label p_i^* is 1 for a certain class of objects if the reigon outputed by the ROI layer contains that object and 0 if it does not.

3.2.4 Faster RCNN training

For training the entire model there must be a well defined loss function which encapsulates all of the losses mentioned before. It can be considered as an estimation for error in the

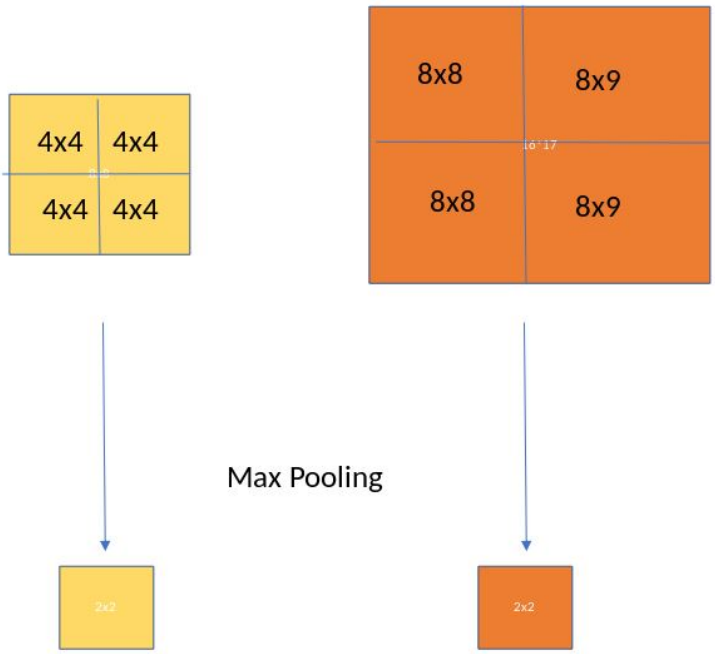


Figure 3.9: rcnn3

model, regardless of where the error occures. The total loss is defined as nothing more than the sum of both losses (RPN loss and Fast RCNN classifier network loss).

$$\text{Total loss} = \text{RPN loss} + \text{Fast RCNN classifier loss} \tag{3.11}$$

The training process would proceed using the same optimization and regularization techniques discussed earlier.

Bibliography

[1] https://classroom.udacity.com/courses/ud188/lessons.

[2] https://www.mathworks.com/help/vision/ug/anchor-boxes-for-object-detection.html.

[3] https://www.coursera.org/learn/convolutional-neural-networks?

[4] http://host.robots.ox.ac.uk/pascal/VOC/.

[5] Aaron Courville Ian Goodfellow, Yoshua Bengio. *Deep learning*. MIT press, 2017.

[6] Yoshua Bengio Jason Yosinski, Jeff Clune and Hod Lipson. How transferable are features in deep neural networks? *arXiv:1411.1792v1 [cs.LG] 6 Nov 2014*, November 2014.

[7] Hogne Jorgensen. *Automatic License Plate Recognition using Deep Learning Techniques*. PhD thesis, Norwegian University of Science and Technology, Department of Computer Science, July 2017.

[8] Ali Farhadi Joseph Redmon. Yolo9000: Better, faster, stronger. *arXiv arXiv:1612.08242v1*, December 2016.

[9] Ali Farhadi Joseph Redmon. Yolov3: An incremental improvement. *arXiv arXiv:1804.02767v1*, April 2018.

[10] Ross Girshick Ali Farhadi Joseph Redmon, Santosh Divvala. You only look once: Unified, real-time object detection. *arXiv arXiv:1506.02640v5*, May 2016.

Symbols

CNN Convolutional neural networks

ANN Artifitial neural networks