Mathematical Foundations of Deep Learning: A Brief Overview with Application to Image Classification Problem

Erdi KARA
Texas Tech University, Mathematics and Statistics

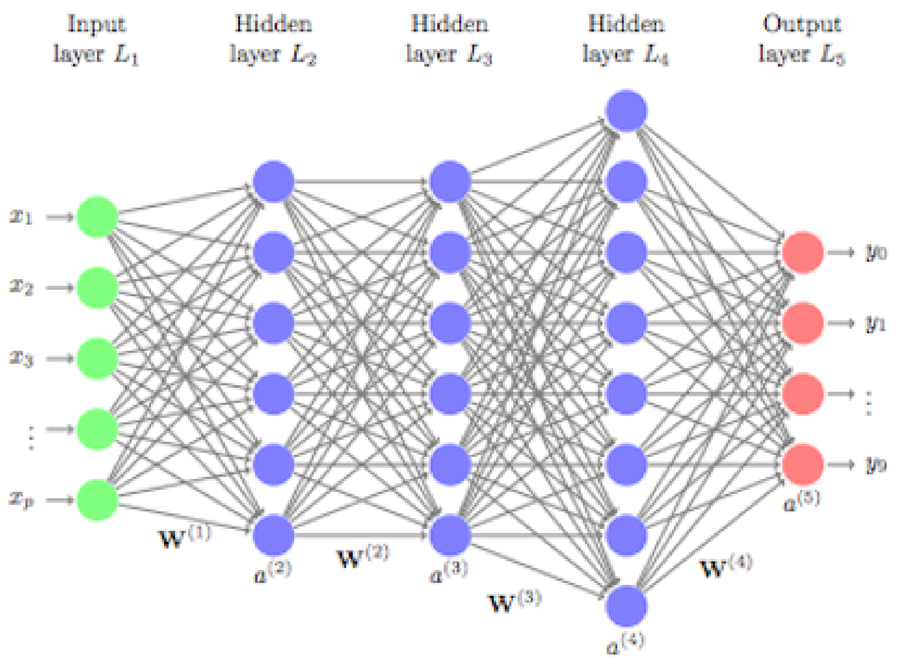
Introduction

In this small report, we will explain the underlying ideas of neural networks. We first briefly introduce the mathematics behind it and then give a quick overview on regularization techniques. Then, we will show how these tools can be used to classify images using Pytorch[1].

1 Feedforward Neural Networks

1.1 Preliminary Concepts

Deep learning can be considered as a subclass of machine learning that utilize multiple-layer architecture to extract information from a given data. Each layer consist of units, called *neurons*, which is a numerical representation of the processed information. Each neuron in the architecture has two typical parameters (possibly other parameters as well), *a weight* and *a bias*. A neural network architecture is composed of an *input layer, hidden layer(s)* an *output layer*. Each hidden layer has a certain level of connection with the previous and the next layer. We define the *depth* of a network as the number of layers except the first one while the *width* is the the maximum number of neurons in a layer. A network with depth 4 and width 8 can be seen in Figure 1.1 where weights and biases are displayed in matrix form which we will see later.



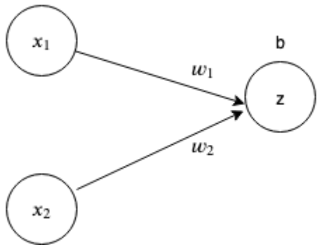
Input layer is used to hold the row input data. We then pass the data to the hidden layers which process this data using weights and biases of the neurons belonging to the layer. Passing the information from one layer to another is called a *forward pass*. In this way, information flows through the hidden layers until the output layer. After the forward pass is completed, we compute the loss of the algorithm against a known true value using a *cost function*. In other words, we measure the quality of the prediction. We then use this information to update the weights and biases of the layers starting from the last one to the first. This backward-updating process is called *back propagation*. We process the entire data by forward-passing and back-propagating. One such complete cycle is called an *epoch*. We repeat the same process for several epochs to improve the quality of the prediction. The main goal is to update the weights and biases (possibly other parameters as well) to minimize the cost function in each epoch so that we can obtain more accurate predictions.

Deep neural network (DNN) can be regarded as a collective name of all similar methods. The network above is called the feed forward (sometimes multilayer perceptrons) because the output is not fed back to the system. Neural networks having this property are called recurrent neural networks(RNN). Another type of neural network is the convolution neural networks (CNN) which we will discuss in Section 2. In this report, we primarily focus on feed forward and convolution networks. However, fundamental principles we will discuss in the next section is applicable to a broad range of neural network models.

1.2 Mathematical Foundations

1.2.1 Motivation

In mathematical terms, a feed forward neural network defines an approximation scheme $\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x}; \boldsymbol{\theta})$ to approximate a function $y = f(x)$. The network optimizes or *learns* the parameter family $\boldsymbol{\theta}$ to find the best approximation to the function $f(x)$. We can start with an illustrative example. Consider the following network with 2 input neurons and a single output. As a convention, weights and bias belong to the next layer.



Lets define the output in a linear fashion

$$z = x_1w_1 + x_2w_2 + b$$

and choose a target discrete function $y = f(x, y)$ such that $(2, 7) \to 11$. So prescribes inputs are $x_1 = 2, x_2 = 7$ and output $y = 11$. Thus the approximation becomes

$$z = 2w_1 + 7w_2 + b$$

Here we call w_1, w_2 and b as *learnable or tunable* parameters. Next, we can define a cost function of the form

$$C(z) = \frac{1}{2}(y - z)^2$$

for a prescribes value y. In this example,we explicitly have

$$C(z) = \frac{1}{2}(11 - z)^2$$

The core idea in a neural network is to find the optimal values of learnable parameters to minimize the cost function. In this problem, we will try to find the optimal parameters so that $z \approx 11$. Dynamics of these type problems are well studied in convex optimization theory. We will not go through the details of this theory but just describe the methods to handle this problem. One of the most common way to obtain a good approximation is to use the *gradient decent* algorithm.

It is very well known that the maximum rate of change of a differentiable function occurs in the direction of the negative gradient. The cost we define is an *implicit function* of learnable parameters. In this method, we assign some initial values to these parameters, compute the gradient, take a step in the direction of the negative gradient and update the value of z. The step size taken in the direction of the negative gradient is called the *learning rate*. This quantity measures how fast we update the learnable parameters. We then repeat this process several times and try to obtain an approximation to $y = 11$. We first compute the partial derivatives of the cost function

$$\frac{\partial C}{\partial w_1} = \frac{\partial z}{\partial w_1}\frac{\partial C}{\partial z} = 2(z - 11)$$

$$\frac{\partial C}{\partial w_2} = \frac{\partial z}{\partial w_2}\frac{\partial C}{\partial z} = 7(z - 11)$$

$$\frac{\partial C}{\partial b} = \frac{\partial z}{\partial b}\frac{\partial C}{\partial z} = 1(z-11)$$

We then assign some initial values for the corresponding parameters, let $w_1^0 = 0, w_2^0 = 0$ and $b^0 = 1$ and learning rate $\eta = 0.01$. Now, we can set up the update loop as follows for this particular problem.

Algorithm 1 Gradient Decent

```
$w_1^{(0)} = 0, w_2^{(0)} = 0, b^{(0)} = 1, \eta = 0.01$
$z^{(0)} = 2w_1^{(0)} + 7w_2^{(0)} + b^{(0)}$
  for $n = 0..14$ do
      $w_1^{(n+1)} = w_1^{(n)} - \eta(2(z^{(n)} - 11))$
      $w_2^{(n+1)} = w_2^{(n)} - \eta(7(z^{(n)} - 11))$
      $b^{(n+1)} = b^{(n)} - \eta(z^{(n)} - 11)$
      $z^{(n+1)} = 2w_1^{(n+1)} + 7w_2^{(n+1)} + b^{(n+1)}$
  end for
```

Note that the number of iterations 15 above corresponds to the notion of *epoch* define in Section 1.1. Also, the first three lines in the for loop is the back propagation stage and the last line is forward pass. After 15 epochs, we get the following approximations.

$$w_1 = 0.215681, w_2 = 1.50976, b = 0.21568$$

If we recompute the output using the last updated results, we have

$$z = x_1 w_1 + x_2 w_2 + b = 10.99975$$

which has the error of the order 10^{-5}. At this point, we should note the following points;

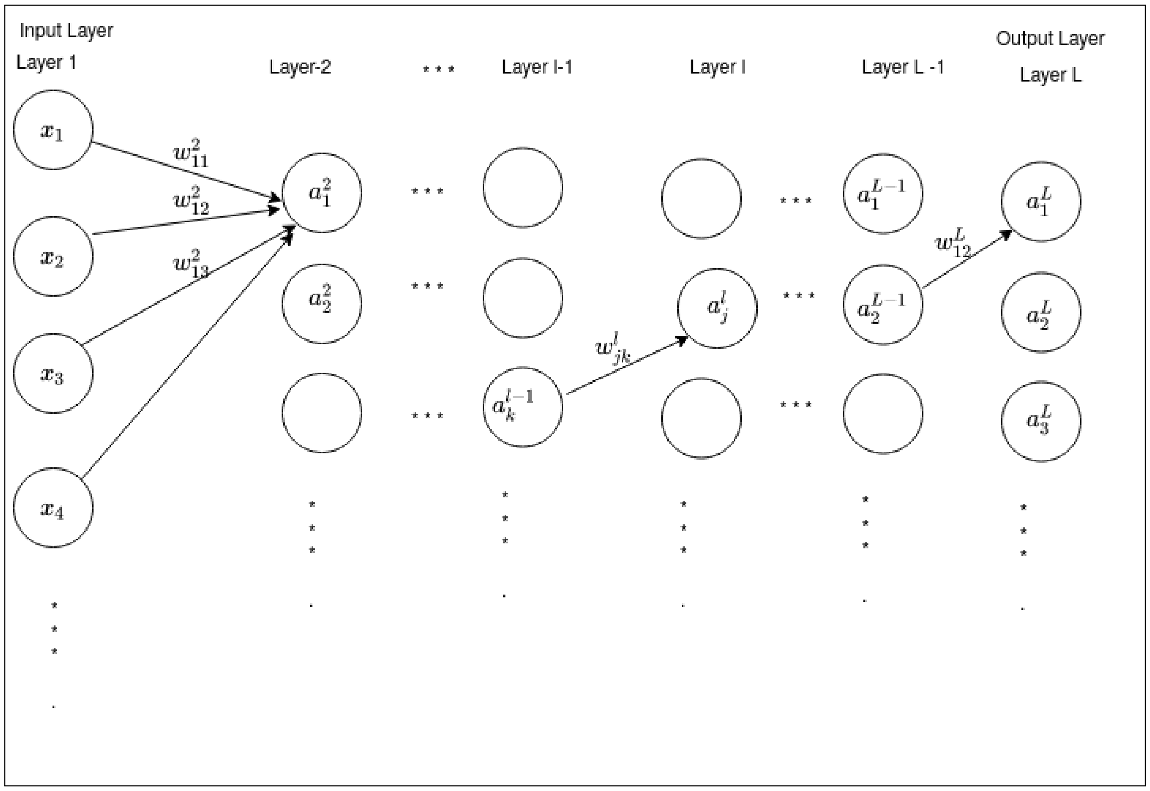
- Learning rate defines the behavior of the gradient decent. If we choose $\eta = 0.1$, the iteration does not converge to the optimal values. Thus choosing an optimal learning rate is an essential pre-processing part of any deep learning model.
- Number of epochs is also another factor which effects the approximation. For example, if we iterate only 5 epochs, the erros is of the order 10^{-2}.
- This network is essentially a linear model. We will modify it in a nonlinear fashion using an *activation function* to enable it to learn more complex tasks. In fact, introducing an activation function has huge implications on learning task that we will discuss in the next sections.
- Gradient of a real valued function is a vector valued quantity. The algorithm above is componentwise representation of the gradient decent method.

1.2.2 Gradient Based Learning

In this section, we will generalize the ideas above for any feed forward network We will closely follow the conventions in [2]. Lets consider the following notations:

w^l_{jk} → weight from the k^{th} neuron in layer $l-1$ to the j^{th} neuron in layer l.
b^l_j → bias of the j^{th} neuron in layer l
a^l_j → activation of the j^{th} neuron in layer l

A typical fully connected network with L-layer can be seen in the Figure 1.2.2. For simplicity, we did not draw all connections.



Next we define an activation function. The most commons ones are given as follows:

$$\psi(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \quad \rightarrow \text{Rectified linear unit} \tag{1}$$

$$\psi(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \quad \rightarrow \text{Binary step} \tag{2}$$

$$\psi(x) = \sigma(x) = \frac{1}{1+e^{-x}} \quad \rightarrow \text{sigmoid} \tag{3}$$

$$\psi(x) = \tanh x = \frac{1}{1+e^{-2x}} \quad \rightarrow \text{tan hyperbolic} \tag{4}$$

$$\psi(x) = \arctan x \quad \rightarrow \text{inverse tangent} \tag{5}$$

A model determined by a generic activation function $\psi(x)$ can be defined by the relationship given in (6).

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{6a}$$

$$a_j^l = \psi(z_j^l) \tag{6b}$$

where the sum is over the neurons in the $(l-1)^{th}$ layer. We will call z_j^l as an output and a_j^l as an activation. Relationship between two consecutive layers can be expressed as a matrix multiplication as follows:

$$\begin{bmatrix} z_1^l \\ z_2^l \\ .. \\ z_j^l \\ .. \end{bmatrix} = \begin{bmatrix} w_{11}^l & w_{12}^l & w_{13}^l & ... \\ w_{21}^l & w_{22}^l & w_{23}^l & ... \\ .. & .. & .. & .. \\ .. & .. & w_{jk}^l & .. \\ .. & .. & .. & .. \end{bmatrix} \begin{bmatrix} a_1^{l-1} \\ a_2^{l-1} \\ .. \\ a_k^{l-1} \\ .. \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ .. \\ b_j^l \\ .. \end{bmatrix} \tag{7a}$$

$$\begin{bmatrix} a_1^l \\ a_2^l \\ .. \\ a_j^l \\ .. \end{bmatrix} = \psi \begin{bmatrix} z_1^l \\ z_2^l \\ .. \\ z_j^l \\ .. \end{bmatrix} \tag{7b}$$

In matrix-vector notation, we can write

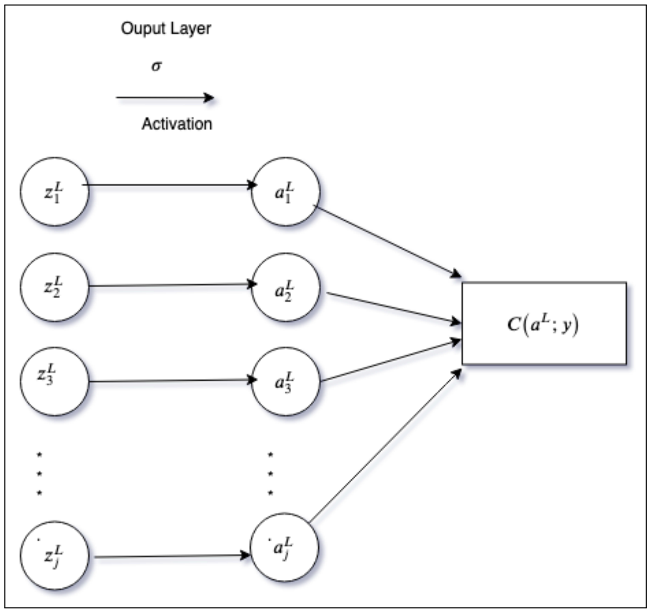
$$\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l \tag{8}$$

Note that if there are $\boldsymbol{M}$ neurons in l^{th} layer and $\boldsymbol{N}$ neurons in $(l-1)^{th}$ layer, W becomes **MxN** matrix.

Let m and n be the dimensions of the input and output layers. By following a similar path, we will derive the fundamental equations which enable us to approximate a given function $\vec{y}_n = f(\vec{x}_m)$. We should note that this function has a discrete form in practical applications. In other words, we try to optimize the parameters of the architecture to find the best approximate for the data points $(\vec{x}, \vec{y})$. Let

$$C = C(a^L; y) \tag{9}$$

be a generic cost function which we will specify later on. Although C is defined over the last output layer, it is an implicit function of all weights and biases in the architecture. We can relook at the structure of the last layer in Figure 1.2.2.



We want to find an expression of the rate of change of the cost function with respect to the weights and biases of the architecture. Lets define a generic error expression:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \tag{10}$$

Using the chain rule with 6b, we can write

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\psi'(z_j^L) \tag{11}$$

In vectorial form

$$\boxed{\delta^L = \nabla_a C \odot \psi'(z^L)} \tag{12}$$

where $\odot$ represents component-wise (Hadamard) product with the gradient $\nabla_a C = \left[\frac{\partial C}{\partial a_1^L}, \frac{\partial C}{\partial a_2^L}, \dots, \frac{\partial C}{\partial a_n^L}\right]^T$. Once z^L is computed through forward pass, (12) can be evaluated with a prescribed cost function. In other words, this information is sufficient to update the weights and biases in the last layer. Now, we will propagate this error all the way back to the second layer. Lets look at the rate of change of the cost function with respect to the output z_j^l. Carefully note that all neurons z^{l+1} in the $(l+1)^{th}$ layer is dependent on z_j^l. Thus the following relation holds:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\frac{\partial C}{\partial z_k^{l+1}} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1}$$

If we rewrite the relation (6a), $z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1}\psi(z_j^l) + b_k^{l+1}$, we obtain $\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\psi'(z_j^l)$. This implies

$$\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\psi'(z_j^l) \tag{13}$$

In vectorial form, we have

$$\boxed{\delta^l = (W^{l+1})^T\delta^{l+1} \odot \psi'(z^l)} \tag{14}$$

Our main goal is to propagate the error on the previous layers to update the weights and biases. From (6a), we can write

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial z_j^l}{\partial b_j^l}\frac{\partial C}{\partial z_j^l} = 1 \cdot \delta_j^l \tag{15}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial z_j^l}{\partial w_{jk}^l}\frac{\partial C}{\partial z_j^l} = a_k^{l-1}\delta_j^l \tag{16}$$

Using these last two equations in the light of (12) and (14), we can describe one cycle forward pass and back prorogation as follows:

Algorithm 2 Gradient Decent for a Feed Forward Network

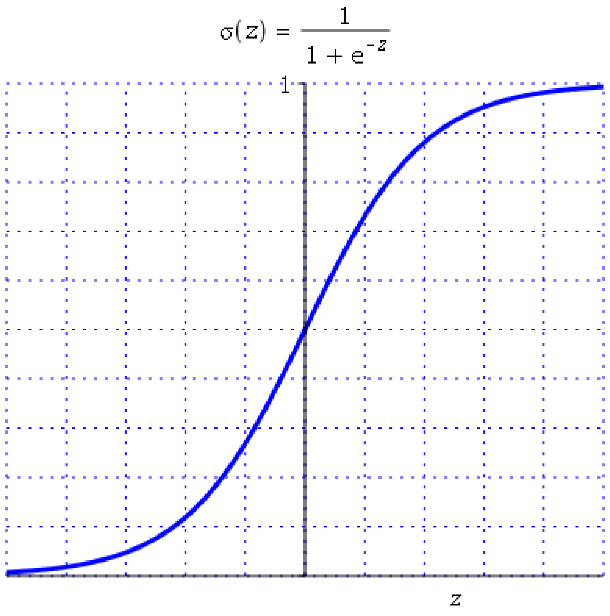
```
Set input data $\vec{x}_m$ and learning rate $\eta$
Assign initial values to all weights and biases.
**Feed Forward**
**for** $l = 2, 3 \ldots L$ **do**
    $a^l = \psi(W^l a^{l-1} + b^l)$
**end for**
Compute the output loss and update the last layer parameters
$\delta^L = \nabla_a C \odot \psi'(z^L)$
$W^L \leftarrow W^L - \eta\delta^L(a^{L-1})^T$
$b^L \leftarrow b^L - \eta\delta^L$
**Back Propagation**
**for** l = L-1, L-2, ...2 **do**
    $\delta^l = (W^{l+1})^T\delta^{l+1} \odot \psi'(z^l)$
    $W^l \leftarrow W^l - \eta\delta^l(a^{l-1})^T$
    $b^l \leftarrow b^l - \eta\delta^l$
**end for**
```

In the back propagation step, $\delta^l(a^{l-1})^T$ is the matrix form of the component-wise expression (16). Above algorithm is the description for a single epoch and a single data point.

Note that we use a generic learning rate η for all layers but in practice one can use a parameter-based learning rate and adjust the learning rate layer-wise or parameter-wise. For example, one of the most popular optimization methods in deep learning ADAM(Adaptive Moment Estimation)[3] computes adaptive learning rates for each parameter but trade off in this scheme is that it requires more time to train comparing to other methods.

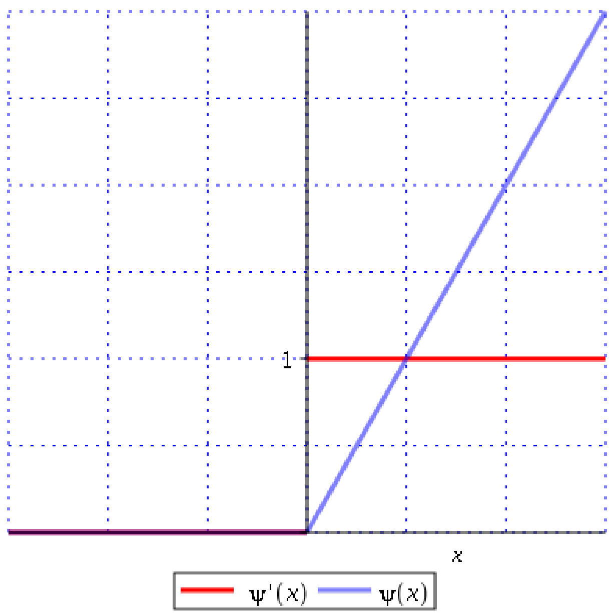
Some comments on the activation function

1. Let's consider the equation (12) with the sigmoid function. As we can see from the figure (1), the range of σ is $[0,1]$. Thus, if $\sigma(z_j^L) \approx 0$ or 1, its derivative at this point is near flat, i.e, $\sigma'(z_j^L) \approx 0$. Thus the weight associated with this node most likely changes very slowly in gradient decent or we say learns slowly. This type of neurons are called saturated or near saturation. Similar discussions hold also for the equation (14) which pertains to the early layers. This problem is also known as learning slowdown.



2. In the light of the same discussion, let's take the equation (14) which we we utilize to update the weights and biases at the corresponding layers. Notice that the saturation effect in the last layers may propagate back to the early layers and cause neurons to learn very slowly or even completely stop the parameter updates. The effect can be observed more clearly especially in the first layers of very deep architectures. This problem is known as the *vanishing gradient problem.* That being said, we should note that the others terms in (12) can balance the saturation problem and the model may not suffer from vanishing gradient. Thus, this phenomena requires careful mathematical analysis.

3. One way to avoid saturation problem would be to define an activation function whose derivative does not approach to zero in its domain.One of the most common activation function used to address this point is the *rectified linear unit (Relu)* defined as
$$\psi(x) = max(x,0)$$
which is always positive with a positive derivative. In practice, Relu is almost the best choice. See figure (3).



We have not discussed a specific form of the cost function so far. Assume that we have a data set containing N_p data points of the forms

$$\mathcal{D} = \{(\vec{x^i}, \vec{y^i})\}_{i=1}^{N_p} \tag{17}$$

These are predefined input and output values. We prefer to use vector notation because we usually deal with multidimensional input and output. We will use this data set to update the learnable parameters of the architecture. In other words, we train the model using this data set. Thus, we will call $\mathcal{D}$ as a *training set*. A proper cost function for this data set must be of the following form

$$C = \sum_{i=1}^{N_p} C^{(i)}(\vec{y^i}; a^{(i)L}) \tag{18}$$

where $a^{(i)L}$ is the output activation or predicted value for $\vec{x^i}$ and the form of $C^{(i)}$ is to be specified. One common cost function is the mean-square error function defined as

$$C = \frac{1}{2N_p} \sum_{i=1}^{N_p} \|\vec{y^i} - a^{(i)L}\|^2 \tag{19}$$

where the vector norm $\|\cdot\|$ can be defined as the Euclidean distance, i.e,

$$\|\vec{y^i} - a^{(i)L}\|^2 = \sum_{k=1}^{n} (y_k^i - a_k^{(i)L})^2$$

In gradient decent algorithm, we must compute the gradient of the cost function $\nabla_a C$ to evaluate the term $\delta^L = \nabla_a C \odot \psi'(z^L)$. Using (18), we have

$$\nabla_a C = \sum_{i=1}^{N_p} \nabla_a C^{(i)} \tag{20}$$

Once we have this value, we can then enter the loop and update the parameters. However if N_p is very large, i.e. training set consists of large number of samples, finding the exact value of the gradient is not feasible from computational cost viewpoint. The most common approach is to update the parameters after some specified value M such that $M < N_p$. This approach is called *mini-batch gradient decent.* In mathematical terms;

$$\nabla_a C = \sum_{i=1}^{N_p} \nabla_a C^{(i)} \approx \sum_{i=1}^{M} \nabla_a C^{(i)} \tag{21}$$

M is called the *batch* size and along with the learning rate, this is another parameter which should be taken into account in training stage.

Cross Entropy Cost Function

Let's consider the first example in (1.2.1) with an activation $\psi(x)$. If we rewrite (11) for a generic data point $(\vec{x}, \vec{y})$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\psi'(z_j^L) = (a_j^L - y_j)\psi'(z_j^L) \tag{22}$$

If we employ this result in (16) for the last layer we obtain

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}\delta_j^L = a_k^{L-1}(a_j^L - y_j)\psi'(z_j^L) \tag{23}$$

In the discussion above, we see that the term $\psi'(z_j^L)$ can lead to vanishing gradient problem. One idea would be to use the rectified linear unit as an activation function. Another approach is to define a cost function which can eliminate the derivative term in (23). Let's define so called *cross entropy* cost function along with sigmoid activation. Again for a generic point $(\vec{x}, \vec{y})$

$$C = -\sum_j \left[y_j \ln(a_j^L) + (1 - y_j)\ln(1 - a_j^L) \right] \tag{24}$$

where $a_j^L = \sigma(z_j^L)$. First note that sigmoid function satisfies $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. From here,

$$\frac{\partial C}{\partial a_j^L} = -\frac{y_j}{a_j^L} + \frac{1 - y_j}{1 - a_j^L} = -\frac{y_j}{\sigma(z_j^L)} + \frac{1 - y_j}{1 - \sigma(z_j^L)} = \frac{\sigma(z_j^L) - y_j}{\sigma(z_j^L)(1 - \sigma(z_j^L))} = \frac{\sigma(z_j^L) - y_j}{\sigma'(z_j^L)}$$

If we plug this result into (23), we obtain

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}\delta_j^L = a_k^{L-1}(a_j^L - y_j) \tag{25}$$

Notice that this last expression is free from the derivative term. Thus, it theoretically avoids learning slow down.

Softmax Layer
Assume that instead of using an activation function, we define another type of output layers of the form

$$a_j^L = \gamma(z_j^L) = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{26}$$

γ is called a *softmax* function. We can see that γ is monotonically increasing, positive and less than 1. Moreover $\sum_j \gamma(z_j^L) = 1$. Thus, we can consider the range of γ as a probability distribution. In other words, the output of softmax layer can be interpreted as a probability vector. For example, in image classification problems, it is common to label different categories with number tags such as 'A':1, 'B':2, 'C':3 and set the correct output as $A : [1, 0, 0]$, $B : [0, 1, 0]$, etc. So if the model endowed with softmax outputs a vector such as $[0.90, 0.08, 0.02]$, we interpret that the input is most likely A and less likely C. Without softmax, the entries of this vector are real numbers and the maximum entry is taken as the predicted value. Notice that softmax provides a nice way to interpret the predictions from probability point of view.
It is very common to employ logarithmic softmax with negative log-likelihood cost function

$$C = -\ln(\gamma(z_j^L)) \tag{27}$$

and partial derivatives related to weight and biases gives a similar result to (25). One can also use softmax along with the cross entropy cost.

2 Convolution Neural Networks(CNN)

In this section, we will introduce the basic principles of another neural network model, convolutional neural networks(CNN). CNN models are essential tools for for image analysis problems. In the previous section, we discussed DNN models where each neuron are essentially connected to each other in the architecture. This actually means that we don't assume any meaningful connection between the neurons. A CNN models aims to explore possible hidden connections in the input data and thus to learn with a more *regularized* way.

Introduction

Let's start with the formal definition of the convolution operation. Convolution is a special integral transform. Let f and g be two functions. " f convolution g " is a function defined as

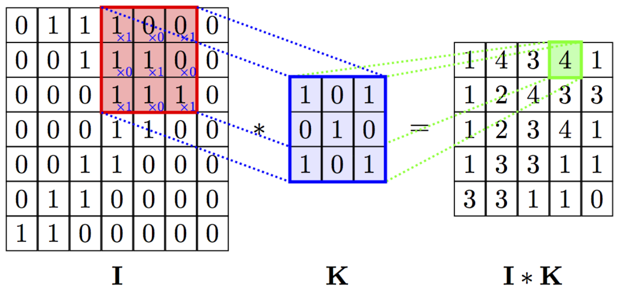
$$(f * g)(x) = \int_{-\infty}^{\infty} f(\tau)g(x-\tau)d\tau \tag{28}$$

Geometrically, we first reflect the function $g(\tau)$ to $g(-\tau)$ and add a time offset so that we obtain $g(t-\tau)$. We then multiply $g(t-\tau)$ with $f(\tau)$, vary t from $-\infty$ to ∞ and compute the integral over all t's where both function are non-zero. In practical application, we can use define a discrete a convolution in 2D as follows:

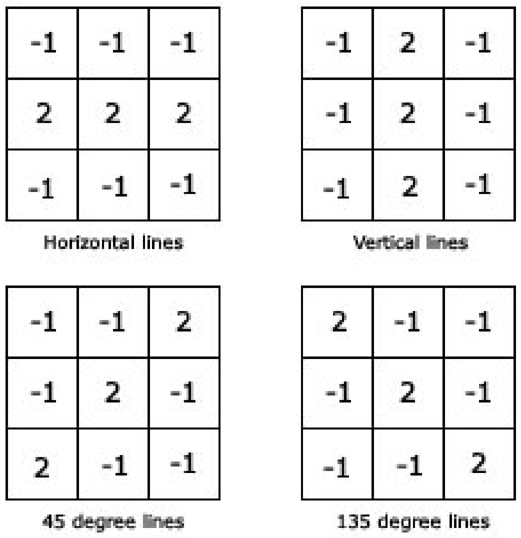
$$(f * g)(x, y) = \sum_{i,j} w_{ij} f(\tau_i, \tau_j) g(x - \tau_i, y - \tau_j) \tag{29}$$

where w_{ij} are the integral weights.

Let's consider a concrete example to illustrate how (29) can be used in image processing. Let the function f represents an $m \times n$ pixel image,i.e., $f(x_i, y_j)$ is the pixel value at (i, j) location. We will call $g(x, y)$ as a *kernal* function. Assume that as a discrete function $g(x, y)$ has 3×3 pixel support size. In this case, for a specific pixel location, the sum in (29) is performed over this 3×3 region. We can start with the upper left most pixel, apply 3×3 discrete convolution and slide g to the next pixel and perform the same operation. When we scan the entire image, the result is essentially another image. Here, the sliding amount is called the *stride* size while the 3×3 is called the *filter size*. An illustrative figure can be seen below.



The convolution or filtering operation plays a central role in image processing. Discrete kernels are used for noise removal, edge detection, image enhancement and so many other image processing operations [4]. For example, following kernels can be used as edge detectors.



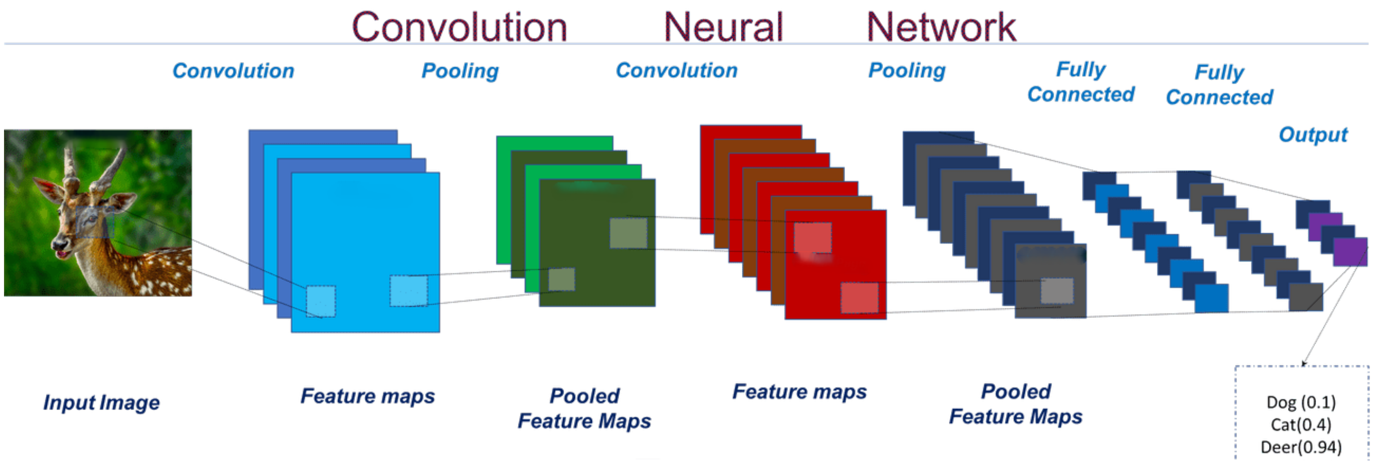
There are some details in the implementation of image filters such as zero padding, wrapping, reflection etc but we will not go through the details. We should however mention the following formula concerning the output dimension after the convolution. For a square kernel of size $k \times k$, and image of size $W \times W$ with padding p, stride S, the output image is of the size

$$W_{out} = \frac{W - k + 2p}{S} + 1$$

The kernels in the figure above are specifically designed for edge detection. However we can claim that any type of filter applied to an image reveals *some pattern* even if it is not qualified as a meaningful pattern.

CNN models use the kernels to explore the features in an image in a more systematic way but the major difference is that the kernels are determined by the learnable parameters what we call *shared weights* which don't have prescribed values but evolve over the course the training. Let's go through the details.

CNNs have three major components; local receptive fields, shared weights and a pooling layer. Shared weights are different type of kernels and the local receptive field is the portion of the image where we apply the kernel. Carefully note that even though we tend to think the image as mxn pixel size, we in fact have $m \cdot n$ input neurons to be passed to the architecture. In CNN models, convolution operation is usually followed by a *pooling* operation where we further condense the convoluted information. A visial structure of CNN can be seen in the figure below(Source https://vinodsblog.com).



Let's formally define how to apply convolution operation in a CNN model. In convolution layers, we have convolution operation instead of matrix multiplication, which we can define as follows.

$$z_{x,y}^{l+1} = w^{l+1} * \psi(z_{x,y}^{l}) + b_{x,y}^{l+1} = \sum_{a=1}^{k} \sum_{b=1}^{k} w_{a,b}^{l+1} \psi(z_{x-a,y-b}^{l}) + b_{x,y}^{l+1} \tag{30}$$

where w is the $k \times k$ kernel at the l^{th} layer, i.e, $k \times k$ matrix with entries $w_{a,b}^{l}$. The output image of the convolution operation is called a *feature map*. Note that the kernel w is a parameter which is same for the corresponding feature map but different for the others. As for the bias, we can use one bias per convolution filter (*tied bias*) or one bias for eaach kernel location(*untied bias*). We used untied bias here. Now assume that we have a gray scale image of size 50×50. If we prescribe 7 feature maps with 3×3 kernels then we have $7 \times (3 \times 3 + 1) = 70$ parameters with tied bias. Note that for a color image with 3 color channels, we have $7(3 \times 3 \times 3 + 1) = 196$ parameters. The output size of each feature map with zero padding stride 1 becomes $\frac{50-3+0}{1} + 1 = 48$. Convolution is followed by a pooling operation which reduce the size of convolution output with an operation called pooling. The most common ones are max pooling and L2 pooling which we define as

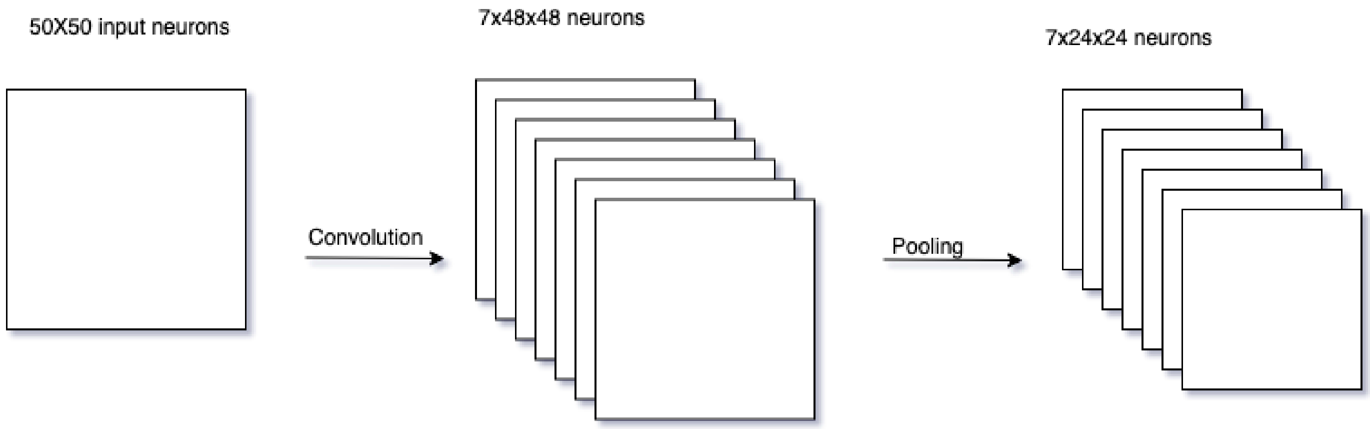
$$z_{i,j} = max\{z_{i+n-1,j+n-1} : 1 \leq n \leq p\} \qquad \text{max pooling} \tag{31a}$$

$$z_{i,j} = \sqrt{\sum_{n=1}^{p}\sum_{n=1}^{p} z^2_{i+n-1,j+n-1}} \qquad \text{L2 pooling} \tag{31b}$$

where p is the pooling size. Max pooling outputs the maximum element in $p \times p$ pooling region while L2 pooling computes the L2 average. Output dimension of a pooling operation can be found as follows:

$$W_{out} = \frac{W - p}{S} + 1$$

where W is the input size, p is the kernel size and S is the stride. Complete picture can be seen in the figure below for $S = 2, p = 2$.



2.1 Back Propagation in CNN Models

In this section, we will derive the fundamental equation of backpropagation in CNN models. Let's define

$$\delta^l_{x,y} = \frac{\partial C}{\partial z^l_{x,y}} \tag{32}$$

for a generic cost function. Using the chain rule and equation (30), we can write

$$\delta^l_{x,y} = \frac{\partial C}{\partial z^{l+1}_{x',y'}} \frac{\partial z^{l+1}_{x',y'}}{\partial z^l_{x,y}} = \sum_{x'}\sum_{y'} \delta^{l+1}_{x',y'} \frac{\partial}{\partial z^l_{x,y}} \left(\sum_a \sum_b w^{l+1}_{a,b} \psi(z^l_{x'-a,y'-b}) + b^{l+1}_{x',y'} \right) \tag{33}$$

Note that the derivative term above is 0 except the indexes satisfying $x'-a = x$ and $y'-b = y$, i.e., $a = x' - x, b = y' - y$. Using the definition of the convolution (30), we have

$$\delta^l_{x,y} = \sum_{x'}\sum_{y'} \delta^{l+1}_{x',y'} w^{l+1}_{x'-x,y'-y}\psi'(z^l_{x,y}) = \delta^{l+1} * w^{l+1}_{-x,-y}\psi'(z^l_{x,y}) \tag{34}$$

Sometimes it is written $w_{-x,-y} = ROT180(w_{x,y})$ Here ROT180 represent the 180 degree rotation of the kernel. If start indexing w from the middle element entry, $w_{-x,-y}$ can be found by rotating the same kernel 180 degree about $(0,0)$ index.

We then compute the partial derivatives using (34).

$$\frac{\partial C}{\partial w^l_{a,b}} = \sum_x\sum_y \frac{\partial C}{\partial z^l_{x,y}}\frac{\partial z^l_{x,y}}{\partial w^l_{a,b}} = \sum_x\sum_y \delta^l_{x,y}\frac{\partial}{\partial w^l_{a,b}}\Big(\sum_{a'}\sum_{b'} w^l_{a',b'}\psi(z^{l-1}_{x-a',y-b'}) + b^l_{x,y}\Big) =$$

$$\sum_x\sum_y \delta^l_{x,y}\psi(z^{l-1}_{x-a,y-b}) = \delta^l_{a,b} * \psi(z^{l-1}_{-a,-b})$$

So we have

$$\frac{\partial C}{\partial w^l_{a,b}} = \delta^l_{a,b} * \psi(z^{l-1}_{-a,-b}) \tag{35}$$

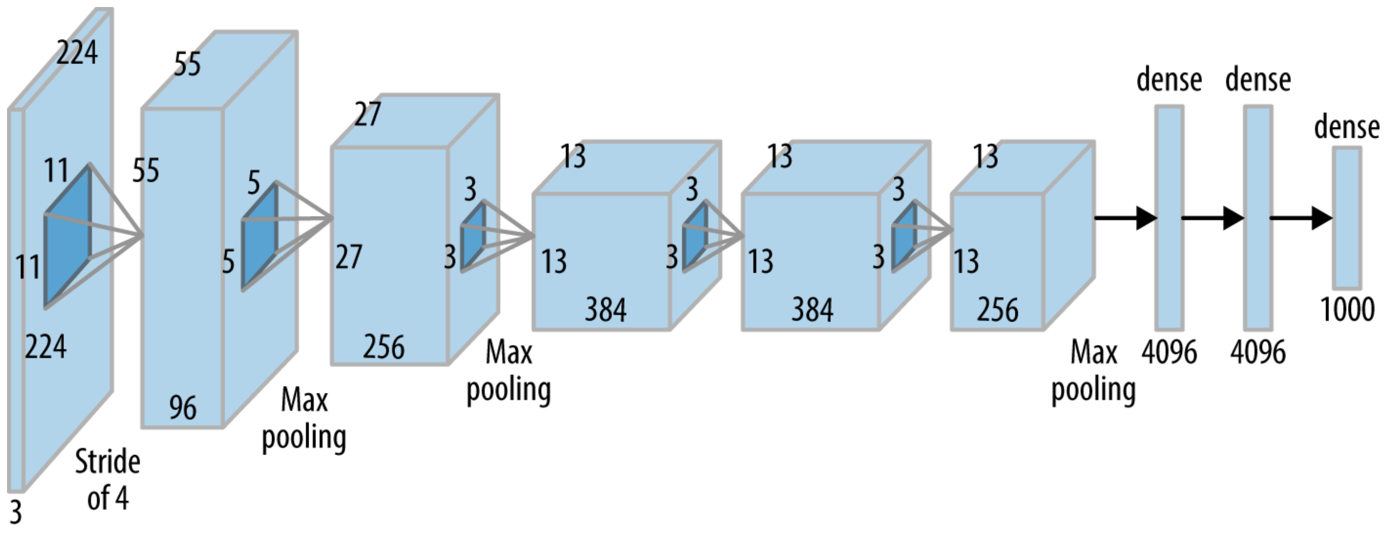
and similarly

$$\frac{\partial C}{\partial b^l_{a,b}} = \delta^l_{a,b} \tag{36}$$

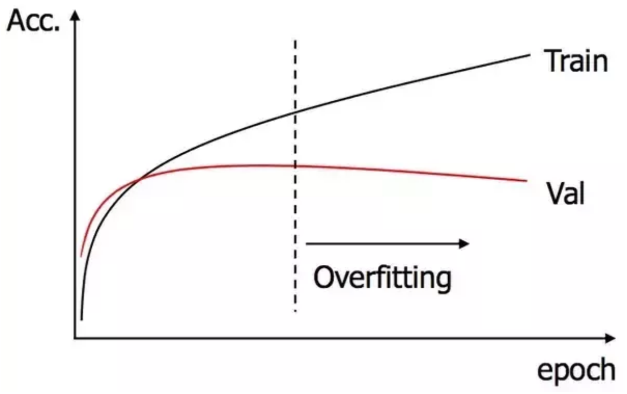
We can close this section with an interesting note. As we see from the construction, we do not pre-assign any specific kernel type (Prewitt, Sobel, Gabor filters etc) to reveal some patters in the input images. However it is observed that as we train the architecture, shared weights are learning to reveal meaningful *patterns* . In particular early layers learn basic shapes such as vertical, horizontal lines while the deeper layers tend to learn more compact patters such as cars, ships, mountains etc.

3 Overfitting and Underfitting

Typical deep neural network with dense and convolutional layers contain massive number of learnable parameters. For example, let's consider one of the first large scale CNN model Alexnet [5] which can be seen in the figure below. It has eight layers; five convolutional layers, some of them followed by max-pooling layers, and three fully connected layers. Total number of weights in Alexnet is $\mathbf{60,954,656}$!. It was the winner of the Large Scale Visual Recognition Challenge in 2012 [6] where the task is to classify 150,000 images in 1000 different categories. The training dataset is composed of 1.2 million images.



In mathematical terms, we are using millions of free parameters to solve an approximation problem. Due to the massive number of parameters, the model can quickly learn or memorize all sorts of details of the training data but may not be able to generalize to or perform well on a different data which was not introduced to the model. This problem is known as *overfitting*. In any deep neural architecture, this problem must be taken with a great care. In the same sense, we can also define *underfitting* which is the case where the model can not capture the underlying structure of the data. There are various ways to detect whether a model is overfitting. One of the most effective way is to monitor the accuracy on training and testing set. *If the validation accuracy is considerably lower than the training accuracy*, this means that the model is failing to generalize beyond the training data set. See the figure below.



In fact one obvious way to tackle with overfitting problem, which we implicitly discussed throughout this report, is to split the data set into training and validation subsets. There is no agreed-opon ratio but 80% vs 20% splitting scheme is very common. We then train the data on training data set and monitor the accuracy on the validation set. In addition, we can consider a third subset where we can tune the other parameters such as the number of epochs, learning rate. We will discuss several methods how to avoid overfitting problem along with how to improve the ability of learning of a model.

4 Improving Neural Networks

In the previous sections, we already discussed several methods which can enhance the architecture's capacity to learn on a given data set. In this section, we further introduce various ways to improve the quality of a model. We should note that our ultimate goal is to efficiently build a model which can make accurate predictions in the most general setting. We can collectively call these techniques as regularization methods. These techniques enable us to train faster, eliminate overfitting and better generalize to the new situations. Note that regularization is an huge active research area . Thus, this section should be taken as a brief and incomplete overview of the existing techniques. For an extensive overview, refer to [7].

4.1 Momentum Based Gradient Decent

In the previous section, we have seen that the ultimate goal of the gradient decent is to gradually converge to a point which minimizes the cost function and we tried to achieve this taking small steps in the direction of the negative gradient. From theoretical perspective, this is a very challenging problem. Because the cost function essentially operates in a very high dimensional space. We mentioned that the Alexnet has around 60M parameters so the cost function represents a $60M$-dimensional surface. The primary challenge here is to distinguish between the local minimum and the global minimum since we are dealing with a *nonconvex* surface. Stochastic gradient decent (SGD) may help us to tackle with this problem for relatively shallow local minimum points but one should introduce different methods to achieve further results for *deep local minimum* points [8].

Momemtum-based gradient decent which is motivated by Hessian approximation to the cost function addresses this problem. We will not further pursue this topic here but just describe the procedure. Idea is to modify the gradient decent with a *momemtum variable*. For a generic quantity of interest w,

$$\begin{aligned} v^{(n)} &= \mu v^{(n-1)} + \eta \nabla C \\ w^{(n)} &= w^{(n-1)} + v^{(n)} \end{aligned} \tag{37}$$

where μ is momentum variable and η is the learning rate. Roughly speaking momentum parameter smooths the step size taken in the direction of the gradient. It also dramatically decreases the training time by quickly minimizing the cost function. Modern optimization schemes such as ADAGrad, RMSProp, ADAM either directly use momentum parameter or take advantage of similar approaches.

4.2 Regularization

In regularization technique, we add a penalty term to the usual cost function C_0 as follows

$$C = C_0 + \frac{\lambda}{2n} \sum_w \|w\|_2^2 \tag{38a}$$

$$C = C_0 + \frac{\lambda}{2} \sum_w \|w\|_1 \tag{38b}$$

(38a) called *L2* (or Ridge regression) regularization and (38b) is called *L1* regularization. Here λ is the regularization parameter and n as the number of training data points. If the gradient decent is applied to (38a) for a generic parameter w, we can see

$$w^{(k+1)} = (1 - \frac{\eta\lambda}{n})w^{(k)} - \eta\frac{\partial C}{\partial w} \tag{39}$$

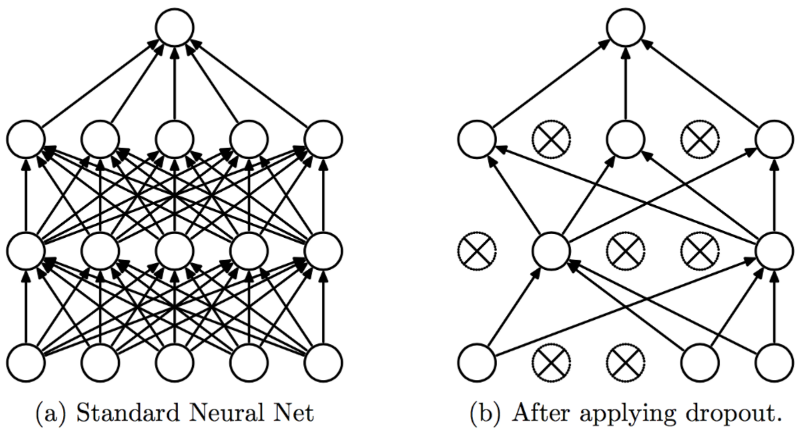
The term $1 - \dfrac{\eta\lambda}{n}$ is called the *weight decay*. From (39), it is clear that the weight decay makes the weight smaller. We have seen that the forward pass is govern by the equation

$$z^l = \psi(W^l a^{l-1} + b^l)$$

By decreasing the weights, we essentially reduce the effect of the activation function, i.e, we don't allow the model to fully utilize its free variables. So, we interpolate the data with a less complex function. This, in turn, effectively reduces overfitting.

4.3 Dropout

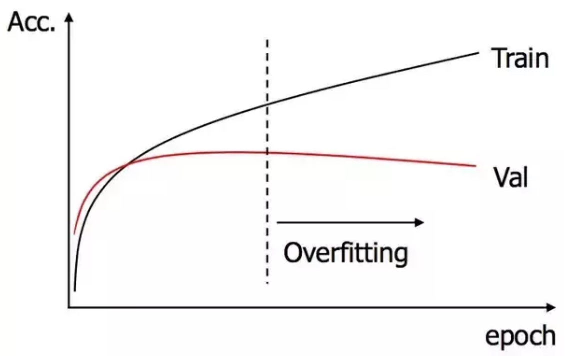
One of the most effective regularization technique is to randomly drop some of the nodes during the training, for example in each epoch. The idea is to prevent the model to learn over certain nodes and spread the learning procedure across the architecture. This technique is called *dropout*. However we do not change the main skeleton of the model by dropping the neurons but zero out the parameters of the corresponding neurons. Dropout is generally implemented in a random fashion. At each epoch, individual neurons are either dropped out of the model with probability $1 - p$ or or kept with probability p so that we obtain a simplified model. Note that incoming and outgoing edges to a dropped-out neurons are also removed. See figure below.


(a) Standard Neural Net (b) After applying dropout.

Carefully note that all neurons are present when we start testing the accuracy of the model in the validation stage. Thus, dropout is the property of training not validation. Because of that, Pytorch provides two modes of the model, training mode where we apply dropout, batch normalization and evaluation mode where we calculate the plain predictions.

4.4 Early Stopping

Let's reexamine the figure (3) again.



It is obvious that the model is overfitting beyond the dashed line. Having an observation like this gives us a very cheap way to set an optimal number of epochs. Or we can set a threshold on the acceptable difference between the training and validation accuracy or loss and stop whenever the model reaches that threshold. This is called *early stopping*. In fact, the graph above is one of the essential dynamics of a network that we should monitor in training stage.

4.5 Preprocessing

Since any kind of data contains noise and random variations, it is very common to perform preprocessing on the data set before the training. For example, resizing to a fixed size and normalizing the data using a fixed mean and standard deviation is an essential practice in computer vision tasks. One common practice is to use ImageNet statistics for normalization;

$$\begin{aligned} mean &= [0.485, 0.456, 0.406] \\ std &= [0.229, 0.224, 0.225] \end{aligned}$$

for each color channel accordingly and the pixel values are scaled to $[0, 1]$. It has been shown that the normalizing the input data helps the model to learn faster and efficiently [9].

4.6 Shuffling

We discussed the stochastic gradient decent in section (1.2.2) and introduced the notion of batch in (21). We pass a batch of images to the architecture, compute the loss and update the parameters and repeat the process until we process all the training data. We called this one epoch loop. One of the very effective technique to avoid overfitting and better train the model is to shuffle the data in *each epoch* and so pass different batch of images to the model. Thus, the parameters is updated in each epoch with different batches of the same size.

4.7 Batch Normalization

In the first section, we have seen that the information flows through the network with affine maps followed by activation function and then parameters are updated backward. For a particular state of an hidden layer, the later layers may perform well to pass the required data. However if the the distribution of the values in one layer changes dramatically, the later layers may not be able to adjust themselves to this quick changes. This problem is known as *internal covariant shift* [10]. So the idea behind batch normalization is to reduce the amount of shift in the distribution of the values in the hidden layers. This is achieved by normalizing the values either before at the activation (z^l) level or after the activation (a^l). Let μ^l and σ^l be the mean and the standard deviation of the z values at the l^{th} layer. Then we can define the output of the batch normalization at the activation level

$$\tilde{z}^l = \gamma\Big[\frac{z^l - \mu^l}{\sigma^l}\Big] + \beta \tag{40}$$

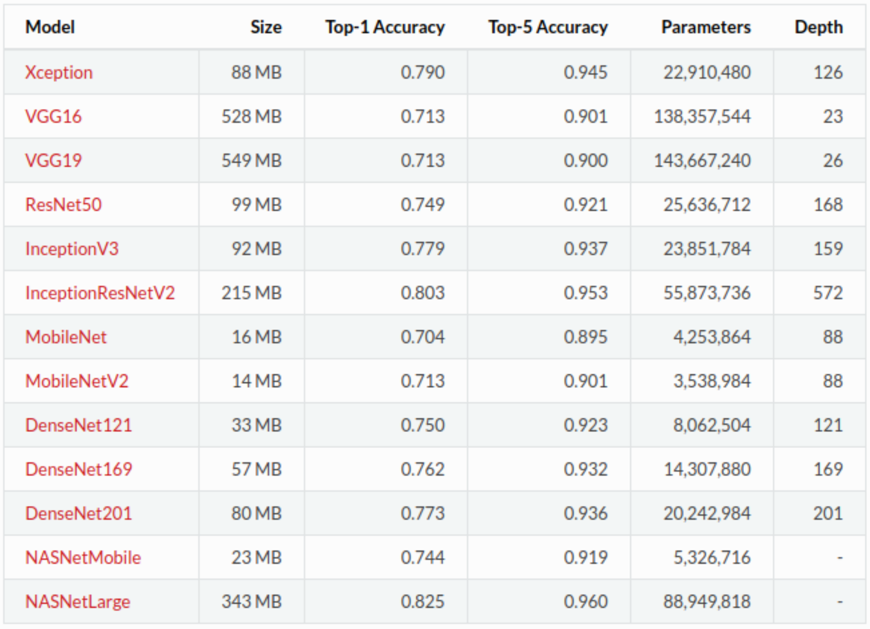
where γ and β are learnable parameters. Note that without these parameters, normalization is performed with $\mu = 0$ and $\sigma = 1$ but adding learnable paramaters gives more flexibility to the model.

4.8 Data Augmentation

This is another common practice in deep learning to avoid overfitting and increase the performance of the model so that it make accurate predictions in a more general setting. In vision problems, we can apply random transformation to the image data in training stage. Instead of feeding the network with the same images in each epoch, we can perform operations such as flipping, cropping, rotation in *each epoch*. Carefully note that even though it can be done, we do not extend the existing data set but introduce a slightly different version in each epoch. This method is almost essential especially when very few data is available. In this way, the architecture does exactly what we want, it learns not only one type of image but also its flipped, blurred, rotated and all sorts of different versions of it.

4.9 Transfer Learning

Training time of an architecture is essentially dependent on the computing platform. All of the well-known neural network models are trained on highly advanced and powerful platforms with massive data sets. For example, AlexNet mentioned above was trained for 6 days in parallel setting on two Nvidia Geforce GTX 580 GPUs. Although one can wrok on a GPU environment with decent price nowadays , training the architecture for days, assuming we optimized the hyperparamters, is not feasible in many cases. So using an existing pretrained model by modifying based on the problem is a very common practice in deep learning. This is called *transfer learning*. Modern deep learning libraries such as Pytorch, Tensorflow provides well known models such ResNET, Alexnet, VGG etc. One can directly import them into the working environment and train with a little effort to obtain a model which performs well on a particular data set. Pretrained models can be examined with Top-1 and Top-5 errors they achieved on a particular database, for example ImageNet. We mentioned that with a softmax layer, models predictions constitute a probability distribution. In Top-1 score, we check if the top class, i.e having the highest probability, is the same as the target label. In Top-5 score, we check if the target label is one of the top 5 predictions of the model. Note that ImageNet has 1000 classes. For a detailed comparison of some of the pretrained models, see the figure below.



Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	99 MB	0.749	0.921	25,636,712	168
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

4.10 Hyper Parameter Optimization

We introduced different parameters so far. We can classify them into two categories; learnable parameters and hyper-parameters. In the first category, we have weights and biases which we update in gradient decent. Hyper-parameters are the ones we set prior to training. Number of layers or neurons, learning rate, momentum, number of epochs, weight decay, batch size and other problem-based parameters are considered as hyperparamters. Finding the optimal values for those parameters can be a very challenging task and there is a huge research efforts in this area. Actually one can define an *hyper-parameter space* and search for the optimal values which can provide an efficient and accurate way of building an architecture. However, when we consider the number of parameters and the size of the training set in practical applications, it is obvious that this effort can require a substantial computing resources, such as paralelization on multiple GPUs and large clusters.

5 An Image Classification Problem

We will apply the techniques we introduced in the report above for an image classification problem. We will work on the Intel Image Classification problem [11]. Data contains around 25k images of size 150×150 distributed under 6 categories;

'building' : 0, 'forest': 1, 'glacier' : 2, 'mountain' : 3, 'sea' : 4, 'street' : 5

There are around 14k images in training set, 3k in validation set and 7k for prediction. We will use one of the most well-known Python machine learning library *Pytorch*. In this report, we design two different architectures.In practical applications, it is very common to start with a pretrained model. So In the first one, we implement a transfer learning with VGG16 [12].We achieved %92.5 test accuracy and %94.1 training accuracy and reported several other results. We also design a custom convolutional neural network and try to fine tune using the techniques introduced above. We obtained %86 testing accuracy here at the time we finish the report but we will spend more time on it in the future.

In order to create a neural network in Pytorch, we use *torch.nn* package which contains necessary building blocks to build a network. Within nn package, we have *nn.Modules* class which is the base class for all of the other neural net modules. So the core idea in neural network programming in pytorch is to build the architecture on the top of *nn.Modules*,i.e, we define the model as a subclass of *nn.Modules*. Once we we create the architecture, we should define a forward pass method. To do so we must overwrite *nn.Modules*'s *forward()* method. We use some functionalities (cost functions, activation functions, dropout, pooling, calculating the gradients etc) provided by *nn.Functional* package to achive this. Once we have the gradients, we can update the weights and biases or other learnable parameters using *nn.Optim* package.

Once we have the architecture with the necessary components, we can develop a strategy for hyper parameter estimation. There is no single best way to do it but our strategy is to create a cartesian product of some of the learnable parameters in outermost loop and train the model over a single element of this product space. We can call this product as hyperspace and define as

$$H_1 x H_2 x..H_n = \{(h_1, h_2, ..h_n) : h_1 \in \{\text{learning rates}\}, h_2 \in \{\text{batch sizes}\}, ..\} \tag{41}$$

We will then enter the training loop, monitor the outcomes such as loss, accuracy and pass them to the *Tensorboard* environment [13] which is a great tool to visualize the outcomes for post processing. In the same loop, we will also write the outcomes in a csv file using python data analysis library *pandas* [14] for further processing. Note that even few elements in each set in (41) results a large product space. For example three hyper parameter sets each having 4 elements yield a product space with 64 elements. Thus we will use a small epoch value in the training loop.

If we can get a reasonable accuracy for one of the element of the hyperspace, we will stick to these parameters and train the model with a larger epoch value. Moreover we will design the algorithm in a way that every experiment returns a model which has the highest test accuracy score over the entire search. Jupyter notebooks pertaining to this project can be found in my in this repository [15].

References

[1] https://pytorch.org/.

[2] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2015.

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] Chris Solomon and Toby Breckon. *Fundamentals of Digital Image Processing: A Practical Approach with Examples in Matlab*. Wiley Publishing, 1st edition, 2011.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[6] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[7] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[8] Nikhil Buduma and Nicholas Locascio. *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*. O'Reilly Media, Inc., 1st edition, 2017.

[9] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[11] https://www.kaggle.com/puneet6060/intel-image-classification.

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[13] https://www.tensorflow.org/guide/summaries_and_tensorboard.

[14] https://pandas.pydata.org/.

[15] https://github.com/erkara/Intel-Image-Classification-with-Pytorch.