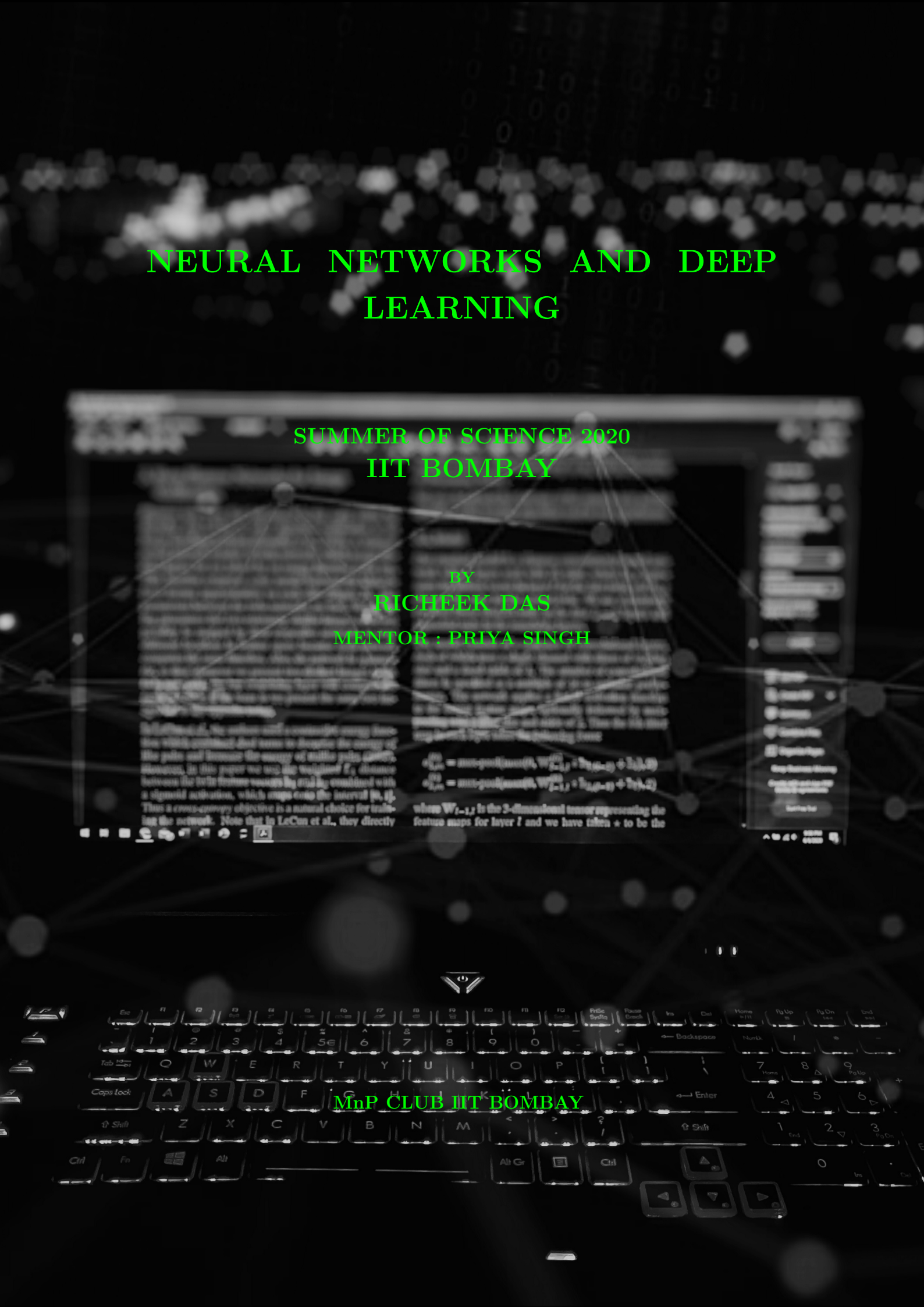
NEURAL NETWORKS AND DEEP LEARNING



SUMMER OF SCIENCE 2020

IIT BOMBAY

BY

RICHEEK DAS

MENTOR : PRIYA SINGH

MnP CLUB IIT BOMBAY

Contents

Chapter 1

Mathematical Preliminaries

1.1 Linear Algebra Basics

This report is aimed towards anyone who has undergrad level knowledge of *Linear Algebra* and *Calculus*, and wants to understand the basics of Neural Networks from ground up.

Linear Algebra is one of the most widely used branch of mathematics in almost all scientific and engineering disciplines. A good understanding of linear algebra is essential for understanding and working with many machine learning algorithms, especially deep learning algorithms as well this report.

1.1.1 Scalars, Vectors, Matrices and Tensors

- **Scalars** : A scalar is just a single number, in contrast to most of the other objects studied in linear algebra, which are usually arrays of multiple numbers.
- **Vectors** : A vector is an array of numbers. The numbers are arranged in such a way that we can identify each individual number by its index in that ordering. If each element is in $\mathbb{R}$, and the vector has n elements, then the vector lies in the set formed by taking the Cartesian product of $\mathbb{R}$ n times, denoted as $\mathbb{R}^n$.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

- **Matrix** : A matrix is a 2-D array of numbers, so each element is identified by two indices instead of just one. If a real-valued matrix $\mathbf{A}$ has a height of m and a width of n, then we say that $\mathbf{A} \in \mathbb{R}^{m \times n}$.
- **Tensors** : It is an array with more than two axes. In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor.

1.1.2 Vectors and Vector Spaces

The most important structure in linear algebra is a vector which we will represent as ψ. This vector resides in a vector space that satisfies some axioms that give this structure it's core defining properties. This is famously known as the *bra-ket* notation and the . is used to represent a vector. The same vector in the vector space $\mathbb{C}^n$ is conveniently represented as a column vector.

We will be generally working in the vector space $\mathbb{C}^n$ over the field $\mathbb{C}$. A set of vectors $\psi_1, \psi_2 \cdots \psi_n$ is said to be *linearly-independent* if for any set of coefficients a_i,

$$\sum_{i=1}^{n} a_i \psi_i = 0 \implies a_i = 0 \forall 1 \leq i \leq n$$

A set of vectors ψ_i is said to be a spanning set of a vector space **W** if any vector in **W** can be represented as a linear combination of that set. Note that such a set may be non-finite however we will be dealing with finite dimensional vector spaces only. If this set is linearly independent then this set is called a *basis* of that vector space. We can also show that the cardinality of all bases are same and this value is termed as the *dimension* of that vector space.

1.1.3 Norms

Sometimes we need to measure the size of a vector. In machine learning, we usually measure the size of vectors using a function called a **norm**. Formally, the L^p norm is given by :

$$||x||_p = \Big(\sum_i |x_i|^p \Big)^{\frac{1}{p}}$$

for $p \in \mathbb{R}, p \geq 1$

On an intuitive level, the norm of a vector **x** measures the distance from the origin to the point **x**. But on, a more rigorous note, its a function which satisfies the following few properties :

- $f(x) = 0 \implies x = 0$
- $f(x + y) \leq f(x) + f(y)$ (the triangle's inequality)
- $\forall \alpha \in \mathbb{R}, f(\alpha x) = |\alpha| f(x)$

The L^2 norm, with $p = 2$, is known as the **Euclidean norm**.

One other norm that commonly arises in machine learning is the L^∞ norm, also known as the **max norm**. This norm simplifies to the absolute value of the element with the largest magnitude in the vector,

$$||x||_\infty = \max_i |x_i|.$$

Sometimes we may also wish to measure the size of a matrix. In the context of deep learning, the most common way to do this is with the otherwise obscure **Frobenius norm**:

$$||A||_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

, which is analogous to the L^2 norm of a vector.

The **trace operator** provides an alternative way of writing the **Frobenius norm** of a matrix :

$$||A||_F = \sqrt{Tr(AA^\intercal)}$$

1.2 Eigendecomposition

Many mathematical objects can be understood better by breaking them into constituent parts, or finding some properties of them that are universal, not caused by the way we choose to represent them.

One of the most widely used kinds of matrix decomposition is called **eigen-decomposition**, in which we decompose a matrix into a set of eigenvectors and eigenvalues.

An **eigenvector** of a square matrix A is a non-zero vector v such that multiplication by A alters only the scale of v:

$$Av = \lambda v$$

If v is an **eigenvector** of A, then so is any rescaled vector sv for $s \in \mathbb{R}, s \neq 0$. Moreover, sv still has the same eigenvalue. For this reason, we usually only look for unit **eigenvectors**. Suppose that a matrix A has n linearly independent **eigenvectors**, $(v(1), \cdots, v(n))$, with corresponding eigenvalues $(\lambda_1, \cdots, \lambda_n)$.

We may concatenate all of the eigenvectors to form a matrix V with one **eigenvector** per column: $V = [v^{(1)}, \cdots, v^{(n)}]$. Likewise, we can concatenate the eigenvalues to form a vector $\lambda = [\lambda_1, \cdots, \lambda_n]^\intercal$. The **eigendecomposition** of A is then given by

$$A = Vdiag(\lambda)V^{-1}$$

1.3 Singular Value Decomposition

The **singular value decomposition** (SVD) provides another way to factorize a matrix, into singular vectors and singular values.SVD is more generally applicable. Every real matrix has a singular value decomposition.

The singular value decomposition is similar, except this time we will write A as a product of three matrices:

$$A = UDV^\intercal$$

Suppose that A is an $m \times n$ matrix. Then U is defined to be an $m \times m$ matrix, D to be an $m \times n$ matrix, and V to be an $n \times n$ matrix. Each of these matrices is defined to have a special structure. The matrices U and V are both defined to be orthogonal matrices. The matrix D is defined to be a diagonal matrix. Note that D is not necessarily square.

The elements along the diagonal of D are known as the **singular values** of the matrix A. The columns of U are known as the **left-singular vectors**. The columns of V are known as as the **right-singular vectors**.

1.4 The Moore-Penrose Pseudoinverse

Matrix inversion is not defined for matrices that are not square. Suppose we want to make a left-inverse B of a matrix A, so that we can solve a linear equation

$$Ax = y$$

by left-multiplying each side to obtain

$$x = By$$

Depending on the structure of the problem, it may not be possible to design a unique mapping from A to B.

The **Moore-Penrose pseudoinverse** allows us to make some headway in these cases. The **pseudoinverse** of A is defined as a matrix

$$A^+ = \lim_{\alpha \to 0} (A^\intercal A + \alpha I)^{-1} A^\intercal$$

Practical algorithms for computing the **pseudoinverse** are not based on this definition, but rather the formula

$$A^+ = VD^+U^\intercal$$

, where U, D and V are the **singular value decomposition** of A, and the **pseudoinverse** D^+ of a diagonal matrix D is obtained by taking the reciprocal of its non-zero elements then taking the transpose of the resulting matrix.

1.5 Principal Component Analysis

One simple machine learning algorithm, **principal components analysis** or **PCA** can be derived using only knowledge of basic linear algebra. So, let's do that !

Suppose we have a collection of m points $[x(1), ..., x(m)]$ in $\mathbb{R}^n$. Suppose we would like to apply lossy compression to these points. Lossy compression means storing the points in a way that requires less memory but may lose some precision.

One way we can encode these points is to represent a **lower-dimensional** version of them. For each point $x(i) \in \mathbb{R}^n$ we will find a corresponding **code vector** $c(i) \in \mathbb{R}^l$. We will want to find some encoding function that produces the code for an input, $f(x) = c$, and a decoding function that produces the reconstructed input given its code, $x \approx g(f(x))$.

PCA is defined by our choice of the decoding function. Specifically, to make the decoder very simple, we choose to use matrix multiplication to map the code back into $\mathbb{R}^n$. Let $g(c) = Dc$, where $D \in \mathbb{R}^{n \times l}$ is the matrix defining the decoding.

Computing the optimal code for this decoder could be a difficult problem. To keep the encoding problem easy, PCA constrains the columns of D to be orthogonal to each other. To give the problem a unique solution, we constrain all of the columns of D to have unit norm.

One way to generate optimal code point c^* for each input point x, is to minimize distance between the input point x and its reconstruction $g(c^*)$. In PCA we use the L^2 norm.

$$c^* = \arg\min_c ||x - g(c)||_2$$

Since, both are minimized by the same value of c, this directly implies $\implies$

$$c^* = \arg\min_c ||x - g(c)||_2^2$$

The function being minimized simplifies to,

$$(x - g(c))^\intercal (x - g(c))$$
$$= x^\intercal x - 2x^\intercal g(c) + g(c)^\intercal g(c)$$

We can now omit the first term from the function to be minimized because, its independent of c.

$$c^* = \arg\min_c -2x^\intercal g(c) + g(c)^\intercal g(c)$$

Since, $g(c) = Dc$, we can substitute it here and by the orthogonality and unit norm constraints on D:

$$c^* = \arg\min_c -2x^\intercal Dc + c^\intercal I_l c$$
$$= \arg\min_c -2x^\intercal Dc + c^\intercal c$$

We can solve this optimization with vector calculus :

$$\nabla_c(-2x^\intercal Dc + c^\intercal c) = 0$$
$$-2D^\intercal x + 2c = 0$$
$$c = D^\intercal x$$

This makes the algorithm efficient: we can optimally encode x just using a matrix-vector operation. To encode a vector, we apply the **encoder function**.

$$f(x) = D^\intercal x$$

Using a further matrix multiplication, we can also define the PCA reconstruction operation:

$$r(x) = g(f(x)) = DD^\intercal x$$

Next, we need to choose the encoding matrix D. To do so, we use the idea of minimizing the L^2 distance between inputs and reconstructions. We minimize the **Frobenius norm** of the matrix of errors computed over all dimensions and all points:

$$D^* = \arg\min_D \sqrt{\sum_{i,j} \left(x_j^{(i)} - r(x^{(i)})_j\right)^2}, \text{ subject to } D^\intercal D = I_l$$

Let's consider the case where, $l = 1$. In this case, D is just a single vector d. Then the problem reduces to,

$$d^* = \arg\min_d \sum_i ||x^{(i)} - d^\intercal x^{(i)} d||_2^2, \text{ subject to } ||d||_2 = 1$$

Let's write this in a more compact notation. Let $\mathbf{X} \in \mathbb{R}^{m \times n}$ be the matrix defined by stacking all of the vectors describing the points, such that $\mathbf{X}_{i,:} = x^{(i)\intercal}$. Rewriting :

$$d^* = \arg\min_d ||X - Xdd^\intercal||_F^2, \text{ subject to } d^\intercal d = 1$$

Disregarding the constraint for the moment, we can simplify the **Frobenius norm** portion as follows:

$$\arg\min_d ||X - Xdd^\intercal||_F^2$$

$$= \arg\min_d Tr\Big(\big(X - Xdd^\intercal\big)^\intercal \big(X - Xdd^\intercal\big)\Big)$$

$$= \arg\min_d Tr\big(X^\intercal X - X^\intercal Xdd^\intercal - dd^\intercal X^\intercal X + dd^\intercal X^\intercal Xdd^\intercal\big)$$

$$= \arg\min_d -Tr\big(X^\intercal Xdd^\intercal\big) - Tr\big(dd^\intercal X^\intercal X\big) + Tr\big(dd^\intercal X^\intercal Xdd^\intercal\big)$$

We can cycle the order of the matrices inside a trace :

$$= \arg\min_d -2Tr\big(X^\intercal Xdd^\intercal\big) + Tr\big(X^\intercal Xdd^\intercal dd^\intercal\big)$$

$$\textbf{subject to } d^\intercal d = 1$$

.

$$= \arg\min_d -2Tr\big(X^\intercal Xdd^\intercal\big) + Tr\big(X^\intercal Xdd^\intercal\big)$$

Therefore,

$$= \arg\min_d -Tr\big(X^\intercal Xdd^\intercal\big), \text{ subject to } d^\intercal d = 1$$

$$= \arg\min_d Tr\big(d^\intercal X^\intercal Xd\big)$$

This optimization problem may be solved using **eigendecomposition**. Specifically, the optimal d is given by the **eigenvector** of X corresponding to the **largest eigenvalue**.

This derivation is specific to the case of $l = 1$ and recovers only the **first principal component**. More generally, when we wish to recover a basis of principal components, the matrix D is given by the l **eigenvectors** corresponding to the **largest eigenvalues**. This may be shown using proof by induction. This can be done as a good exercise !

Chapter 2

Working of Basic Machine Learning Algorithms

Things we will look at here :

- **Linear Regression**
- **Gradient Descent**
- **Polynomial Regression**
- **Learning Curves**
- **Regularized Linear Models**
- **Logistic Regression**
- **Softmax Regression**
- **Decision Tree**
- **Random Forest**

2.1 Linear Regression

A linear model makes a prediction by simply computing a weighted **sum** of the **input features**, plus a constant called the bias term (also called the **intercept term**)

Linear Regression model prediction :

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Here :

- $\hat{y}$ is the predicted value
- n is the number of features
- x_i is the i^{th} feature value
- θ_j is the j^{th} model parameter, i.e the feature weights.

It can be further elegantly written as :

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^\intercal \cdot \mathbf{x}$$

Here :

- θ is the model's *parameter column vector* containing the bias term θ_0 and the feature weights.
- $\mathbf{x}$ is the instance's *feature vector* containing x_0 to x_n, with x_0 always equal to 1
- Needless to say the $\theta \cdot \mathbf{x}$ is the usual dot product
- h_θ is the hypothesis function, using the model parameters θ

So, let's come to the algorithms handling the training part of Linear Regression

The most common performance measure of a regression model is the **Root Mean Square Error (RMSE)**. Therefore, to train a **Linear Regression** model, we need to find the value of θ that minimizes the **RMSE**. It is simpler to minimize the **mean squared error (MSE)** than the **RMSE**, and it leads to the same result.

The **MSE** of a **LR** hypothesis h_θ on a training set $\mathbf{X}$ is calculated using :

$$\mathbf{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^\intercal \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

This function mentioned above is called the **cost function** for a **Linear Regression model**

To find the value of θ that minimizes the cost function, there is a mathematical tool that gives the result directly. Its called the *Normal Equation*

2.1.1 The Normal Equation

$$\hat{\theta} = \left(\mathbf{X}^\intercal \mathbf{X}\right)^{-1}\mathbf{X}^\intercal \mathbf{y}$$

Here :

- $\hat{\theta}$ is the value of θ that minimizes the cost function.
- $\mathbf{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$.

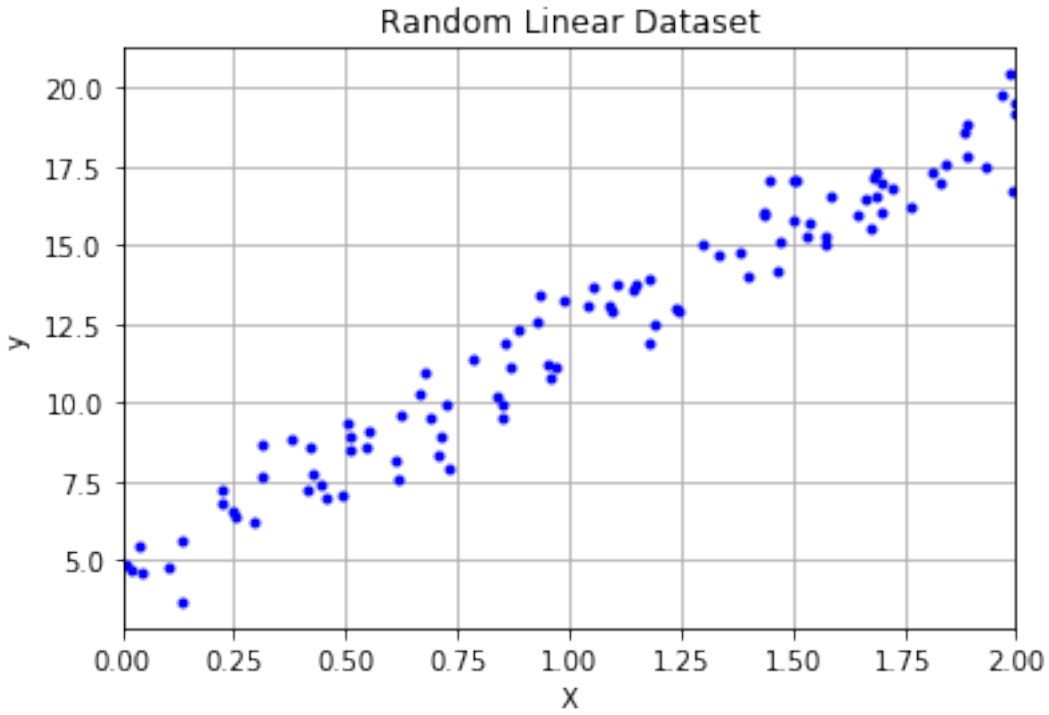
Let's test this normal equation with some code !

[1]:
```
import numpy as np

X = 2 * np.random.rand(100,1)
y = 5 + 7 * X + np.random.randn(100,1)
## Generates a random linear dataset with some Gaussian Noise
```

Let's plot it :

[2]:
```
import matplotlib.pyplot as plt

plt.plot(X, y, " bo", markersize = 3)
plt.xlim(0.00, 2.00)
```

```
plt.grid(True)
plt.xlabel("X")
plt.ylabel("y")
plt.title("Random Linear Dataset")
plt.show()
```



We will use the `inv()` function from *NumPy*'s linear algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication :

[3]:
```
X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Function intended for : $y = 5 + 7x$

Expression found :

[4]:
```
print(theta_best[[0]]) ## Intercept term
print(theta_best[[1]]) ## First weight term
```

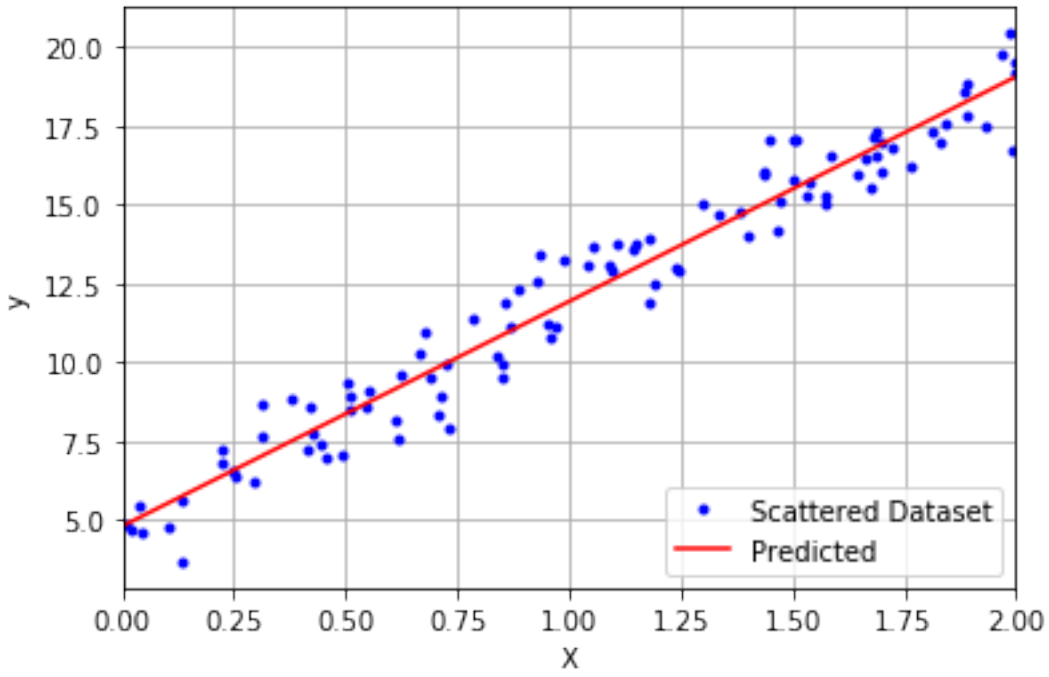
```
[[4.80833324]]
[[7.12658528]]
```

Now, let's get the predicted linear function using the $\hat{\theta}$ found using normal equations :

[5]:
```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)
```

Let's plot the prediction :

[6]:
```
plt.plot(X, y, " bo", markersize = 3, label = "Scattered Dataset")
plt.xlim(0.00, 2.00)
plt.grid(True)
plt.xlabel("X")
plt.ylabel("y")
plt.plot(X_new, y_predict, "r-", label = "Predicted")
plt.legend(loc = "lower right")
plt.show()
```



In *NumPy* we usually calculate $\hat{\theta}$ using :

$$\hat{\theta} = \mathbf{X}^{+} y$$

where, $\mathbf{X}^{+}$ is the **pseudoinverse** of $\mathbf{X}$, aka the **Moore-Penrose inverse**. The pseudoinverse is calculated using a standard matrix factorization technique called *Singular Value Decomposition(SVD)* > Can be found in the mathematical preliminary part

This approach is **more efficient** than computing the *Normal Equation*, plus it also handles the edge cases. The *Normal Equation* may not work if the matrix $\mathbf{X}^{\intercal}\mathbf{X}$ is not invertible, such as if its not a square matrix or if some features are redundant, but the pseudoinverse is always defined.

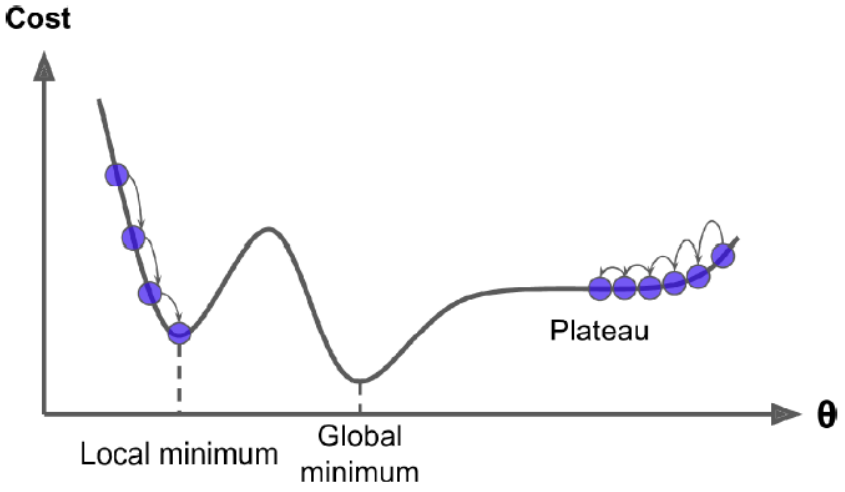
2.1.2 Computational Complexity

The **Normal Equation** computes the inverse of $\mathbf{X}^{\intercal}\mathbf{X}$, which is an $(n+1) \times (n+1)$ matrix (where n is the number of features). The computational complexity of inverting such a matrix is typically about $O(n^3)$, depending on the implementation.

The **SVD** approach has $O(n^2)$. Both these approaches are linear with regard to the number of instances in the training set (they are $O(m)$), so they handle large training sets efficiently.

2.2 Gradient Descent

Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems.



Here, we start by filling θ with random values (*random initialization*). Then we improve it gradually, taking one $d\theta$ step at a time, each step making an attempt to reduce the cost function till it converges to a minimum.

Learning Rate hyperparameter : It speaks about the size of the steps taken. if learning rate is too small then the algorithm will make a huge amount of iterations to converge(in-efficient). On the other hand, if its too high, then it might even make the algorithm diverge, and fail to reach a good solution.

Well, one problem that arises at once, is the fact that there can be multiple local minimas, thus making it difficult to find the global minima with *random initialization* plan, we previously had.

Fortunately, the **MSE cost function** for a **Linear Regression** model happens to be a convex function (we can easily verify it) ! This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly. These two facts have a great consequence: **Gradient Descent** is guaranteed to approach arbitrarily close the global minimum(well, but excluding other factors).

When using Gradient Descent, we should ensure that all features have a similar scale (e.g in **Python**, using `Scikit-Learn StandardScaler` class)(there are other scalers as well, which suit to different occasions), or else it will take much longer to converge.

2.2.1 Batch Gradient Descent

To implement *GD* we nee dto compute the gradient of the *cost function* with regard to each model parameter θ_j. Therefore, we need to calculate the *partial derivative* with respect to θ_j. The equation looks something like this :

$$\frac{\partial}{\partial \theta_j}\mathbf{MSE}\Big(\theta\Big) = \frac{2}{m}\sum_{i=1}^{m}\left(\theta^{\intercal}\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

Gradient vector of the cost function :

$$\nabla_\theta \mathbf{MSE}\Big(\theta\Big) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathbf{MSE}\Big(\theta\Big) \\ \frac{\partial}{\partial \theta_1}\mathbf{MSE}\Big(\theta\Big) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathbf{MSE}\Big(\theta\Big) \end{pmatrix} = \frac{2}{m}\mathbf{X}^\intercal\Big(\mathbf{X}\theta - y\Big)$$

The only problem(or benefit, as you see it) of this algorithm is that it involves calculations over the full training set $\mathbf{X}$ at each step of *Gradient Descent*. That is why it is called the *Batch/Full Gradient Descent*. This makes this algorithm terribly slow but kind of accurate.

However, Gradient Descent scales well with the number of features. Training a *Linear Regression* model when there are a lot of features is much faster using *Gradient Descent* than using the *Normal Equation* or *SVD decomposition* :)

Finding the gradient, gives us the uphill direction, so quote trivially, the *Gradient Descent Step* looks like this :

$$\theta^{(next)} = \theta - \eta\nabla_\theta \mathbf{MSE}\Big(\theta\Big)$$

Here, η is the learning rate of the model.

Now let's try the same example we used with *Normal Equation* :

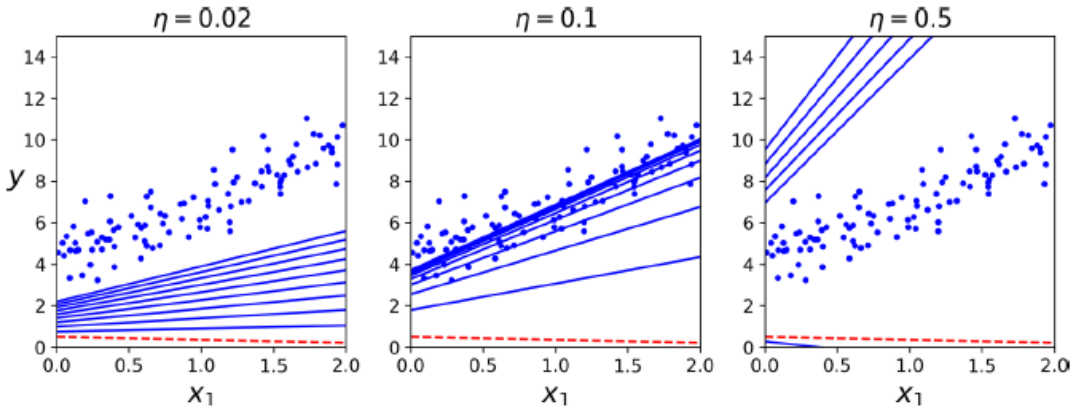
[7]:
```
learning_rate = 0.1
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) #Random initialization to start with

for i in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - learning_rate*gradients
```

The final θ we got:

[8]:
```
theta
```

[8]:
```
array([[4.80833324],
       [7.12658528]])
```

That's the same as what the Normal Equation found ! But the case would be different if we had used a different *learning rate*.

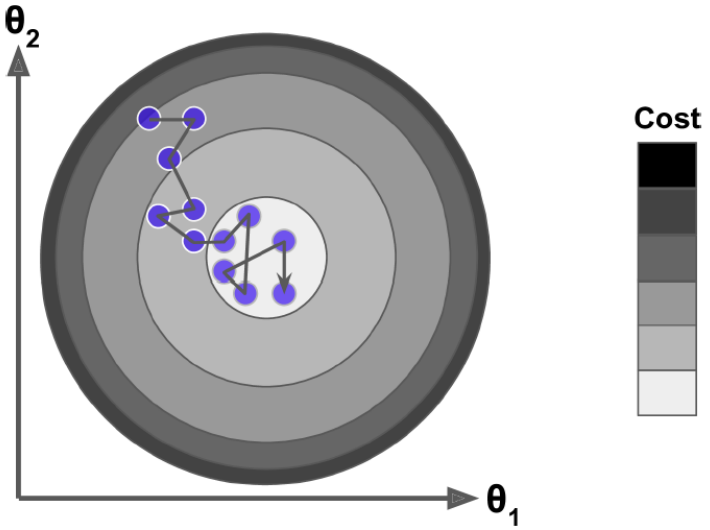


If its too low, then it'll take a lot of time to reach the optimal solution, whereas if its too high, the algorithm diverges and we won't even reach the solution. We can use `Scikit GridSearchCV` for finding the optimal hyperparameter.

For the number of iterations : Best practice is to set a very large number of iterations, and to interrupt the algorithm, when the **gradient vector** becomes smaller than a certain *tolerance*(ϵ).

2.2.2 Stochastic Gradient Descent

"Stochastic" means "random". *Stochastic Gradient Descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration.

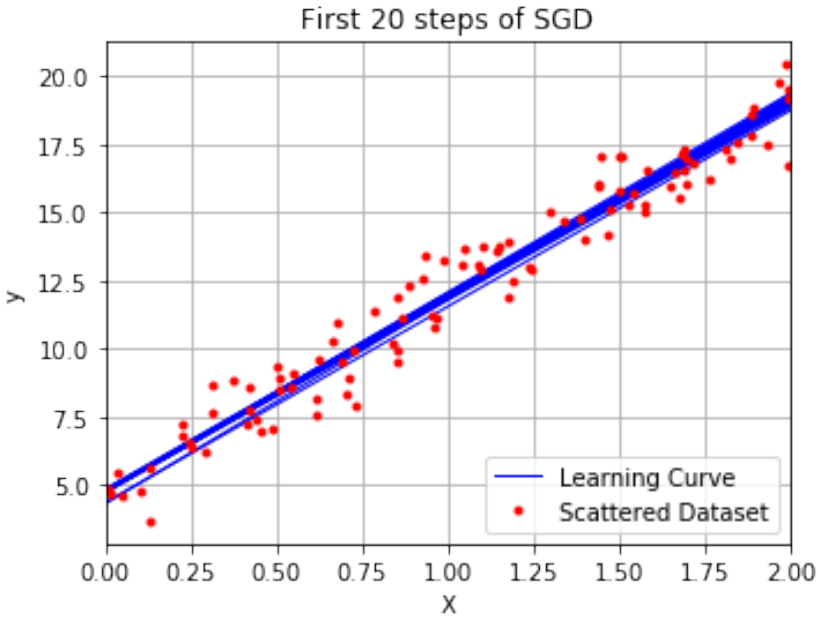


Due to its randomness, **SGD** has a better chance of finding the global minima, than *Batch Gradient Descent*. So, the randomness is good for escaping the local optima, but bad because it can never settle at the minima. One solution would be to gradually decrease the learning rate. The steps start out large, then it dies down and settles at the global minima. This process is also known as ***simulated annealing***. the function which determines the learning rate at each

iteration is called the ***learning schedule***.

Code implementation of *SGD* :

[9]:
```
n_epochs = 50
t0, t1 = 5, 50 # Learning Schedule Hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # random initialization

fig = plt.figure()
ax = fig.add_subplot(111)

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients

    if epoch < 20 :
        X_new = np.array([[0], [2]])
        X_new_b = np.c_[np.ones((2, 1)), X_new]
        y_predict = X_new_b.dot(theta)
        if epoch == 1:
            ax.plot(X_new, y_predict, "b-", linewidth = 1, label = "Learning␣
 ↪Curve")
        else :
            ax.plot(X_new, y_predict, "b-", linewidth = 1)

ratio = 0.6
xleft, xright = ax.get_xlim()
ybottom, ytop = ax.get_ylim()
ax.set_aspect(abs((xright-xleft)/(ybottom-ytop))*ratio)

plt.xlabel("X")
plt.ylabel("y")
plt.xlim(0.00,2.00)
plt.plot(X, y, "ro", markersize = 3, label = "Scattered Dataset")
plt.grid(True)
plt.legend(loc="lower right")
plt.title("First 20 steps of SGD")
plt.show()
```



By convention we iterate by rounds of m iterations; each round is called an *epoch*. While the *Batch Gradient Descent code* iterated **1000 times** through the whole training set, this code goes through the training set only **50 times** and reaches an acceptable solution :

[10]:
```
theta
```

[10]:
```
array([[4.81314772],
       [7.16962854]])
```

2.2.3 Mini-batch Gradient Descent

In this algorithm, at each step, instead of computing the gradients based on the full training set (as in *Batch GD*) or based on just one instance (as in *Stochastic GD*), *Mini-batch GD* computes the gradients on small random sets of instances called **mini-batches**.

The algorithm's progress in parameter space is less erratic than with *Stochastic GD*, especially with fairly large mini-batches but both *Stochastic GD and Mini-batch GD* will reach the minimum if you use a good learning schedule.

2.3 Polynomial Regression

We can actually use a linear model to fit non-linear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called ***Polynomial Regression***.

Let's generate some noise in some non-linear quadratic data :

[11]:
```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

Now, we'll use `Scikit-Learn PolynomialFeatures` class to transform our training data :

[12]:
```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X_poly[0]
```

[12]:
```
array([2.86334939, 8.19876974])
```

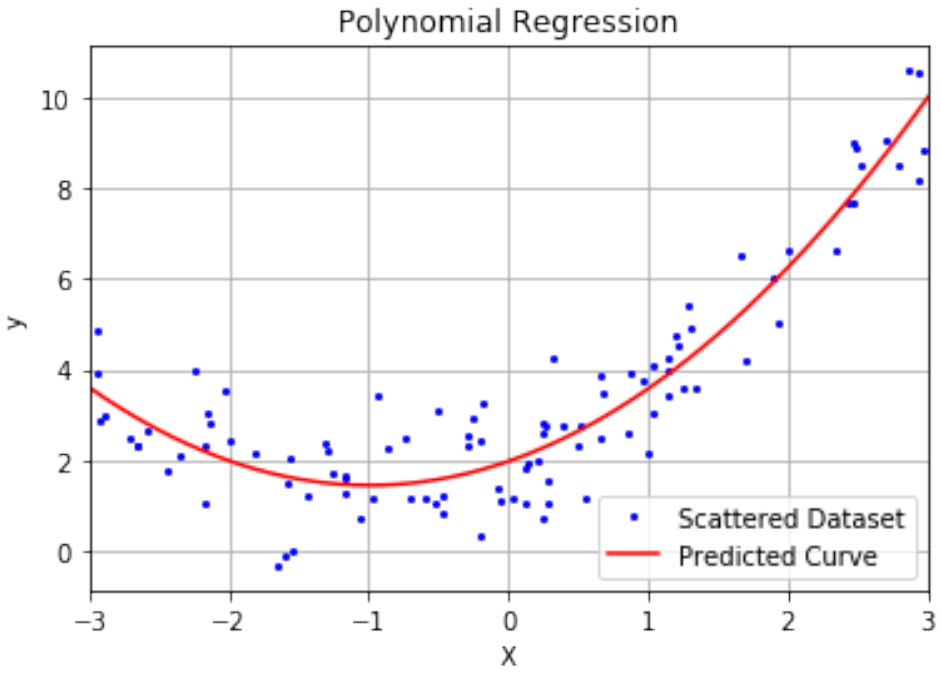
Now let's fit a ***Linear Regression*** model to this extended training data(we are using `Scikit-Learn LinearRegression` here :

[13]:
```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

[13]:
```
(array([1.98031152]), array([[1.06214383, 0.53454417]]))
```

Let's plot and see what we get !

[14]:
```
X_new = np.arange(-3,3.2,0.1)
X_new_b = np.c_[X_new, X_new**2]
y_predict =   np.c_[np.ones((62,1)).dot(lin_reg.intercept_) + X_new*lin_reg.
 ↪coef_[[0][0]][0] + (X_new**2)*lin_reg.coef_[[0][0]][1]]

plt.plot(X, y, "bo", markersize = 2, label = "Scattered Dataset")
plt.plot(X_new, y_predict, "r-", label = "Predicted Curve")
plt.title("Polynomial Regression")
plt.grid("True")
plt.xlim(-3,3)
plt.xlabel("X")
plt.ylabel("y")
plt.legend(loc="lower right")
plt.show()
```



2.4 Learning Curves

If you perform high-degree Polynomial Regression, you will likely fit the training data much better than with plain Linear Regression.

But a very-high degree polynomial has chances of severely *overfitting* the training data, while a linear model might have chances of *underfitting* the same data. So we can use **cross-validation** to get an estimate of the model's performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then the model is overfitting. If it performs poorly on both, then it is underfitting.

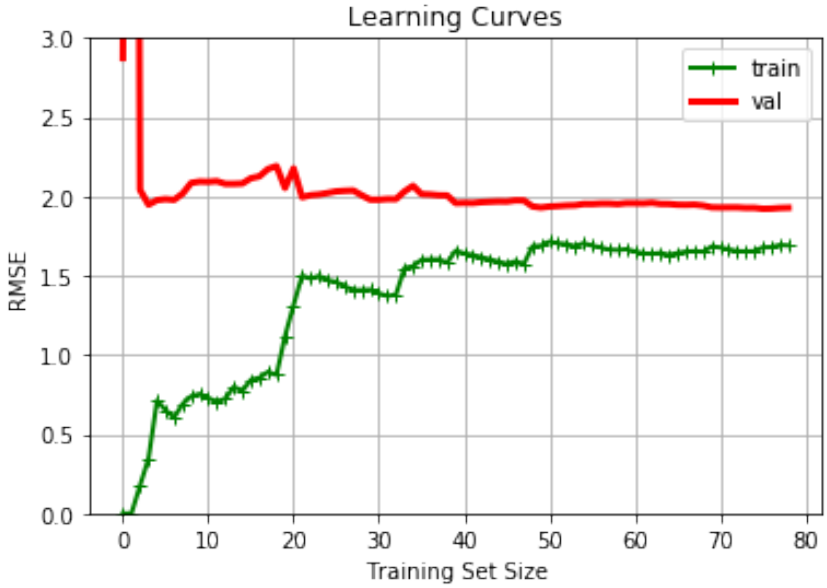
Another way to tell it, is to look at the *learning curves* : These are plots of the model's performance on the training set and the validation set as a function of the training set size. Let's generate the *Learning Curves* here :

[15]:
```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

##A function that, given some training data, plots the learning curves of a␣
 ↪model:
def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m],
        y_train_predict))
```

```
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "g-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "r-", linewidth=3, label="val")
    plt.grid(True)
    plt.legend(loc = "upper right")
    plt.xlabel("Training Set Size")
    plt.ylabel("RMSE")
    plt.title("Learning Curves")
    plt.ylim(0.00,3.00)
    plt.show()
```

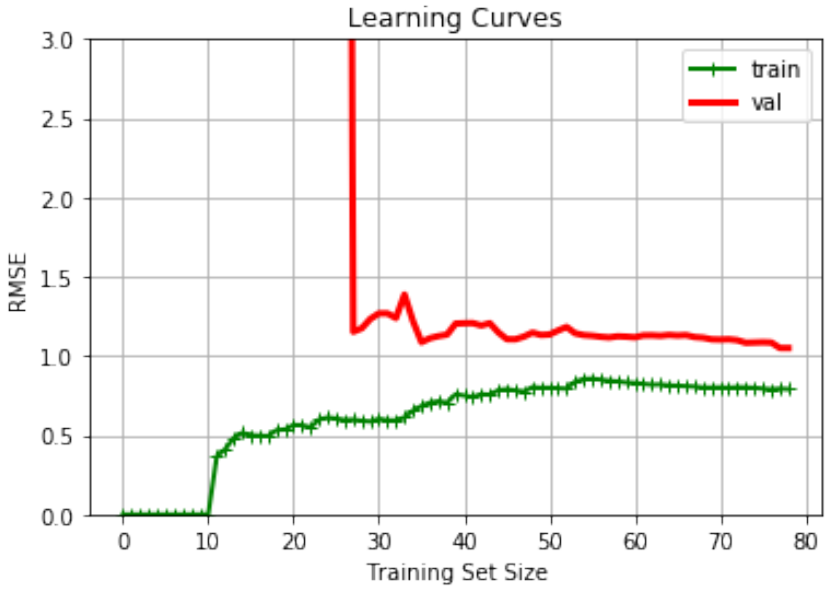
For a plain *Linear Regression* model :

[16]:
```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```



This is a typical example of underfitting. Both training and validation curves have reached a plateau, they are close and high. Let's plot the *learning curve* for a 10th-degree polynomial :

[17]:
```
##Create a pipeline for Scikit first :)

from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline([
                        ("poly_features", PolynomialFeatures(degree=10,
                        include_bias=False)),
                        ("lin_reg", LinearRegression()),
                        ])

plot_learning_curves(polynomial_regression, X, y)
```



Differences :

- The error on the training data is much lower than with the Linear Regression model.
- There's a gap between the curves, which indicates that this model is the heavily overfitting :(

Note : Its good to notice that, as the train set size increases the model performs much better, just like any other overfitting scenario.

2.5 Regularized Linear Models

The fewer degrees of freedom a model has, the harder it will be for it to overfit the data. A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, *regularization* is typically achieved by constraining the weights of the model. We will now look at **Ridge Regression, Lasso Regression, and Elastic Net**, which implement three different ways to constrain the weights.

2.5.1 Ridge Regression

Ridge Regression cost function :

$$J(\theta) = \mathbf{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

Here :

- $\alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$ is the regularization term. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. We apply the regularization term only during the model training. Once the model is trained, we use the unregularized performance.

- The hyperparameter α controls how much we want to regularize the model. If $\alpha = 0$, then *Ridge Regression* is just *Linear Regression*. If α is very large, then all weights end up very close to zero and the result is a flat line going through the mean.

Ridge Regression closed-form solution :

$$\hat{\theta} = \left(\mathbf{X}^\intercal \mathbf{X} + \alpha \mathbf{A}\right)^{-1} \mathbf{X}^\intercal \mathbf{y}$$

We can perform *Ridge Regression* either by computing a closed-form equation or by performing *Gradient Descent*. The pros and cons are the same.

2.5.2 Lasso Regression

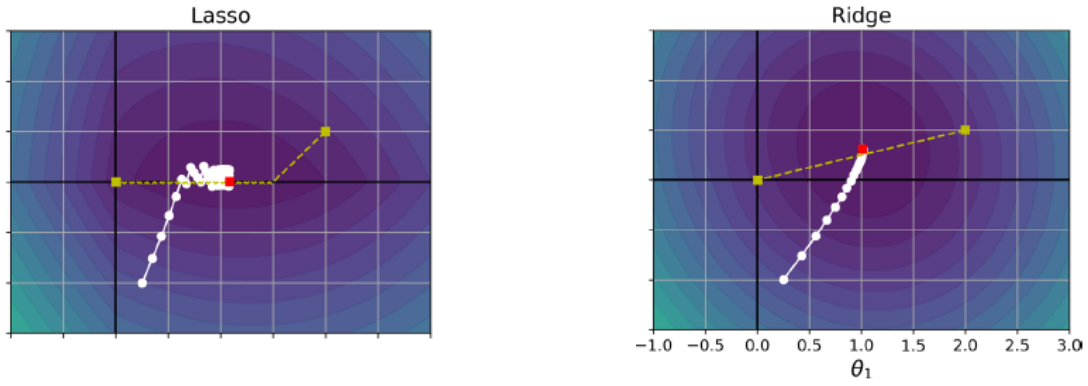
Least Absolute Shrinkage and Selection Operator Regression (usually simply called Lasso Regression) is another regularized version of Linear Regression. But it uses the norm of the weight vector instead of half the square of the norm as the regularization term.

Lasso Regression cost function :

$$J(\theta) = \mathbf{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

An important characteristic of *Lasso Regression* is that it tends to eliminate the weights of the least important features. *Lasso Regression* automatically performs feature selection and outputs a *sparse model*.

Comparison of **Lasso and Ridge models** :



The axes represent two model parameters, and the background contours represent different loss functions. The small white circles show the path that Gradient Descent takes to optimize some model parameters that were initialized.

2.5.3 Elastic Net

Elastic Net is a middle ground between *Ridge Regression* and *Lasso Regression*. The regularization term is a simple mix of both *Ridge* and *Lasso's regularization terms*, and you can control the mix ratio r. When $r = 0$, *Elastic Net* is equivalent to *Ridge Regression*, and when $r = 1$, it is equivalent to *Lasso Regression*.

Elastic Net cost function :

$$J(\theta) = \mathbf{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^{n} \theta_i^2$$

2.5.4 Early Stopping

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called **early stopping**.

As the algorithm learns, the prediction error(RMSE) on the training set goes down, along with its prediction error on the validation set. After a number of *epochs* the **validation error stops decreasing and starts to go back up**. This is an indication of premature overfitting. We need to stop right here.

It functions as a simple and efficient regularization technique. We just need to find the minima of the *validation error* :)

Basic implementation of early stopping :

[ ]:
```
from sklearn.base import clone
from sklearn.preprocessing import StandardScaler
# prepare the data
poly_scaler = Pipeline([
                        ("poly_features", PolynomialFeatures(degree=90,
                                                             include_bias=False)),
                        ("std_scaler", StandardScaler())
                        ])

X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None

for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)

    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

With `warm_start=True`, when the `fit()` method is called it continues training where it left off, instead of restarting from scratch.

2.6 Logistic Regression

Logistic Regression is commonly used to estimate the probability that an instance belongs to a particular class. Therefore, it is a binary classifier.

Just like a Linear Regression model, a *Logistic Regression* model, computes a weighted sum of the input features and it outputs the *logistic* of this result.

Logistic Regression model **estimated probability** :

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma\big(\mathbf{x}^\intercal\theta\big)$$

σ is the *sigmoid function*. It takes the $\mathbf{x}^\intercal\theta$ and squeezes its value to a number between 0 and 1.

Notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a *Logistic Regression* model predicts 1 if $\mathbf{x}^\intercal\theta$ is positive and 0 if it is negative.

2.6.1 Training and Cost Function

The objective of training is to set the parameter vector θ so that the model estimates high probabilities for positive instances (y = 1) and low probabilities for negative instances (y = 0).

Such a cost function which implements this concept for a single training instance :

$$c(\theta) = \begin{cases} -\mathbf{log}(\hat{p}) & \text{if y = 1} \\ -\mathbf{log}(1-\hat{p}) & \text{if y = 0} \end{cases}$$

The cost function over the whole training set is the average cost over all training instances :

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\Big[y^{(i)}log(\hat{p}^{(i)}) + (1-y^{(i)})log(1-\hat{p}^{(i)})\Big]$$

This is known as, *Logistic Regression cost function(log loss)*.

Well, here we doon't have a generalised Normal Equation, but we can easily check that the cost function(log loss) is actually convex, so we can apply our second option, the **Gradient Descent**.

The partial derivatives of the cost function with respect to the j^{th} model parameter is :

$$\frac{\partial}{\partial\theta_j}\mathbf{J}(\theta) = \frac{1}{m}\sum_{i=1}^{m}\Big(\sigma(\theta^\intercal\mathbf{x^{(i)}}) - y^{(i)}\Big)x_j^{(i)}$$

That's it, we can now apply *Batch, Stochastic or Mini-Batch Gradient Descent* to use this *Logistic Regression model* :)

2.6.2 Softmax Regression

The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers. This is called **Softmax Regression, or Multinomial Logistic Regression.**

The idea is simple: when given an instance x, the Softmax Regression model first computes a score $s_k(x)$ for each class k, then estimates the probability of each class by applying the **softmax function** (also called the **normalized exponential**) to the scores.

Softmax score for k^{th} class :

$$s_k(x) = \mathbf{x}^\intercal\theta^{(k)}$$

Note that each class has its own dedicated parameter vector $\theta^{(k)}$. All these vectors are typically stored as rows in a **parameter matrix** Θ.

We can estimate the probability that the instance belongs to class k by running the scores through the **softmax function**.

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x})) = \frac{\mathbf{exp}(s_k(\mathbf{x}))}{\sum_{j=1}^{K} \mathbf{exp}(s_j(\mathbf{x}))}$$

Just like the *Logistic Regression* classifier, the **Softmax Regression** classifier predicts the class with the highest estimated probability.

2.7 So a overview of things covered so far :

We saw the working of quite a few, Machine Learning **Regressors** and **Classifiers**. We will now move on to more interesting stuff :) . But before that let's summarize a bit. Firstly we started off with *Linear Regression*, and we saw that using *Normal Equation* was a pretty slow idea, so we moved towards *SVD decomposition*, and it was faster but it wasn't the thing that we wanted ! Then we looked at a much better optimization algorithm called the *Gradient Descent*.

After we had solved the *speed* issue concerning *Linear Regression* we moved on to, the predictions and fitting issues it has. On the way of doing so we looked at an obvious choice of *Polynomial Regression*. Then we explored a new judging meter called *Learning Curves* and found that a high degree polynomial might overfit the training data, while a linear model might heavily underfit the same data. So an obvious choice to determine the correct polynomial degree was to limit its degrees of freedom and we did that using a class of models called the *Regularized Linear Models*.

Then, we moved on to a commonly used binary classifier *Logistic Regression*, and its generalized version *Softmax Regression*. Now let's take a look at *Decision Trees and Random Forests* and how those compare with the logistic regression model we just studied.

2.8 Decision Tree : An Overview

Decision tree is the most powerful and popular tool for classification and prediction. A Decision tree is a flowchart like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.

2.8.1 Decision rules

The decision tree can be linearized into decision rules, where the outcome is the contents of the leaf node, and the conditions along the path form a conjunction in the if clause. In general, the rules have the form:

```
if condition1 and condition2 and condition3 then outcome.
```

Decision rules can be generated by constructing association rules with the target variable on the right. They can also denote temporal or causal relations.

2.9 Random Forest

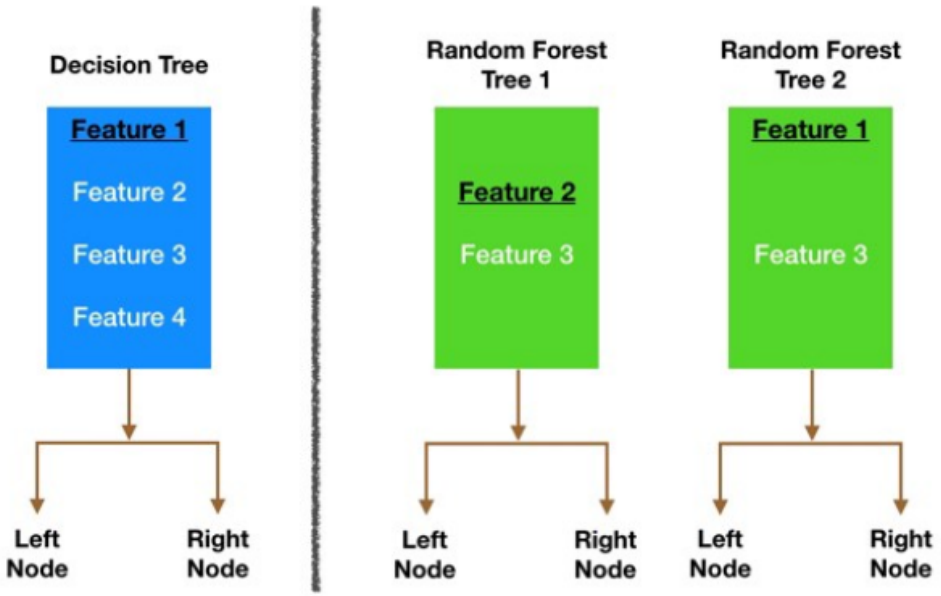
The *Random forest* is a classification algorithm consisting of many uncorrelated decisions trees. It uses **bagging** and **feature randomness** when building each individual tree to try to create an uncorrelated forest of trees whose prediction by committee is more accurate than that of any individual tree.

So how to ensure that the models are diversified ?

We can use the following two methods :

- **Bagging (Bootstrap Aggregation)** - Decision trees are very sensitive to the data they are trained on and small changes to the training set can result in significantly different tree structures. Random forest takes advantage of this by allowing each individual tree to randomly sample from the dataset with replacement, resulting in different trees. This process is known as bagging. With bagging we are not subsetting the training data into smaller chunks and training each tree on a different chunk. Rather, if we have a sample of size N, we are still feeding each tree a training set of size N (unless specified otherwise). But instead of the original training data, we take a random sample of size N with replacement.

- **Feature Randomness** - In a normal decision tree, when it is time to split a node, we consider every possible *feature* and pick the one that produces the most separation between the observations in the *left node* vs those in the *right node*. In contrast, each tree in a random forest can pick only from a *random subset of features*. This forces even more variation amongst the trees in the model and ultimately results in *lower correlation* across trees and more *diversification*.



Node splitting in a random forest model is based on a random subset of features for each tree.

The underlined feature is eventually selected, and the rest of the diagram is self-explanatory.

2.9.1 Comparison to Logistic Regression :

None of them are "better" than the other, but following best practices we can jot down a few points :

- If the problem/data is linearly separable, then first we should try logistic regression. If we don't know, then still we start with logistic regression because that will be our baseline, followed by non-linear classifier such as random forest.
- If our data is categorical, then random forest should be our first choice; however, logistic regression can be dealt with categorical data.
- If we want easy to understand results, logistic regression is a better choice because it leads to simple interpretation of the explanatory variables.
- If speed is our criteria, then logistic regression should be our choice.
- If the data is unbalanced, then random forest may be a better choice.
- If number of data objects are less than the number of features, logistic regression should not be used.

2.10 Conclusion

Now, that we have a rough idea about the working of quite a few **regressors** and **classifiers**, we will move on into the deep world of learning :)

2.11 See Also

- Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow by Aurélien Géron.
- Python Machine Learning by Sebastian Raschka and Vahid Mirjalili
- Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville
- Towards Data Science

Chapter 3

Neural Networks

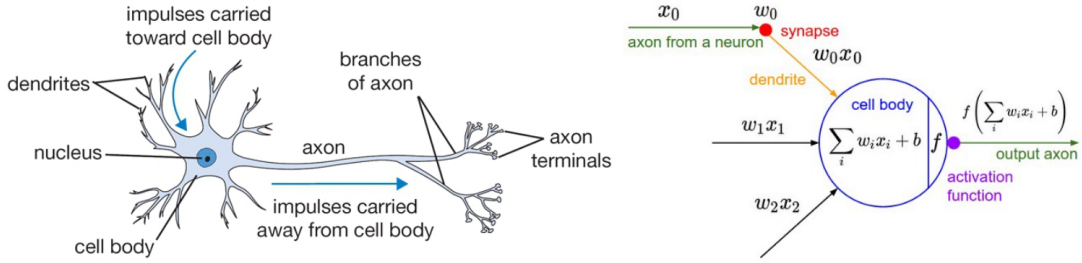
3.1 Introduction

The definition of a neural network, more properly referred to as an 'Artificial Neural Network'(ANN), is provided by the inventor of the world's one of the first neurocomputers, Dr. Robert Hecht-Nielsen as :

> `...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.`

3.1.1 Biological Motivation

The basic computational and signalling unit of the brain is the **neuron**. The diagram below shows a drawing of a biological neuron (left) and a common mathematical model (right).



A basic unit of a neural network is **neuron**. It receives input from some other **nodes** or from an external source and computes an output. Each input is associated with a **weight(w)**. the node applies a function to the weighted sum of its input.

3.2 Neural Network Architecture

We can breakdown our Neural Network Model into :

1. **Input Nodes/ Input Layer** : This layer is just used to pass the information to the next layer (hidden layer most of the time).
2. **Hidden Nodes/ Hidden Layer** : Here, intermediate processing is done. they perform computations and then transfer the weights from input layer to the next layer.

3. **Output Nodes/ Output Layer** : Here the activation function is applied, that maps to the desired output format.

4. **Connections and Weights** : The network consists of connections, each connection transferring the output of a neuron i to the input of a neuron j. Each connection is assigned a weight w_{ij}

5. **Activation Function** : The activation function of a node defines the output of that node given a set of inputs. It is the *non-linear activation function* that allows such networks to compute nontrivial problems using a small number of nodes.

6. **Learning Rule** : The *learning rule* is a rule or an algorithm which modifies the parameters of a NN.

3.3 Types Of Neural Network

There are many classes of neural networks and these classes also have sub-classes. here, we are going to discuss a few of them :)

1. **Feed Forward Neural Networks**
   (a) **Single Layer Perceptron**
   (b) **Multi Layer Perceptron(MLP)**
   (c) **Convolutional Neural Networks(CNN)**
2. **Recurrent Neural Networks(RNN)**
3. **Bayesian Neural Network(BNN)**

Here, we are going to take a look at MLPs and CNN.

3.4 Feed Forward Neural Network

A Feed-Forward neural network is an ANN where connections between the units do not form a cycle. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes.

the linear models for regression and classification discussed in the previous chapters, are based on linear combinations of fixed nonlinear basis functions $\phi_j(x)$ and take the form :

$$y(x, w) = f\Big(\sum_{j=1}^{M} w_j \phi_j(x)\Big)$$

where f is the nonlinear activation function, in the case of classification and is the identity in the case of regression.

Neural Networks use basis functions that follow the same form as above and each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are adaptive parameters. So now, we have a basic neural network model which can be described as a series of functional transformations. First we construct M linear combinations of the input variables $x_1, \cdots, x_D$ in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

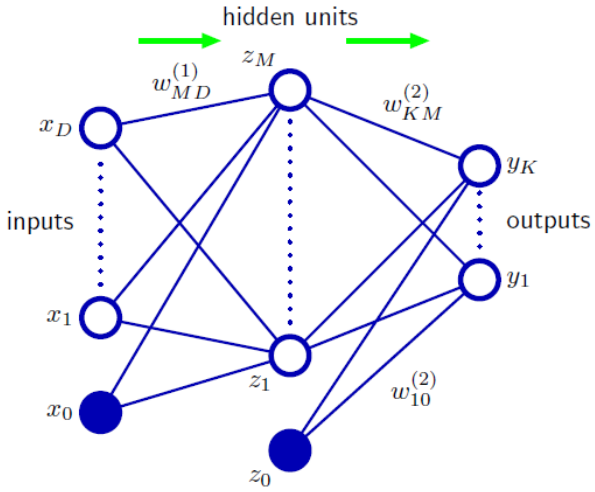
where $j = 1, \cdots, M$ and the superscript (1) indicates that the parameters belong to the first layer of the network. Parameters $w_{ji}^{(1)}$ are the **weights** and $w_{j0}^{(1)}$ are the **biases**. The quantities a_j are known as **activations**. Each of them is then transformed, using a *differentiable, nonlinear activation function* h.

$$z_j = h(a_j)$$

These quantities correspond to the outputs of the basis functions and are called the *hidden units*. The nonlinear functions h are generally chosen to be sigmoidal functions such as *logistic sigmoid* or the *hyperbolic tangent*. These values are again linearly combined to give *output unit activations*.

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

where $k = 1, \cdots, K$ and K is the total number of outputs.



Thus for standard regression problems, the activation function is the identity so $y_k = a_k$. here, each output unit activation is transformed using a logistic sigmoid function.

$$y_k = \sigma(a_k)$$

where,

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

We can combine these various stages to give the overall network function that, for **sigmoidal** output unit activation functions, takes the form

$$y_k(x, w) = \sigma\Big(\sum_{j=1}^{M} w_{kj}^{(2)} h\Big(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\Big) + w_{k0}^{(2)}\Big)$$

We can obviously absorb the bias functions into the weights since this forms a basis of the vector space. So the overall network becomes :

$$y_k(x, w) = \sigma\Big(\sum_{j=1}^{M} w_{kj}^{(2)} h\Big(\sum_{i=1}^{D} w_{ji}^{(1)} x_i\Big)\Big)$$

Since, these have more than two layers, Neural Network is also called the MLP(Multi Layer Perceptron).A key difference compared to the perceptron, however, is that the neural network uses continuous sigmoidal nonlinearities in the hidden units, whereas the perceptron uses step-function nonlinearities. This means that the neural network function is differentiable with respect to the network parameters.

3.4.1 Weight Space Symmetries

One property of feed-forward networks, which will play a role when we consider **Bayesian** model comparison, is that multiple distinct choices for the weight vector w can all give rise to the same mapping function from inputs to outputs.

3.5 Network Training

Given a training set comprising a set of input vectors x_n, where $n = 1, \cdots, N$, together with a corresponding set of target vectors t_n, we minimize the error function.

$$E(w) = \frac{1}{2}\sum_{n=1}^{N} ||y(x_n, w) - t_n||^2$$

we can provide a much more general view of network training by first giving a probabilistic interpretation to the network outputs.

for the moment we consider a single target variable t that can take any real value. We assume that t has a Gaussian distribution with an x-dependent **mean**, which is given by the output of the neural network, so that

$$p(t|x, w) = \mathcal{N}(t|y(x, w), \beta^{-1}$$

where, β is the precision of the Gaussian Noise. Given a data set of N independent, identically distributed observations $X = x_1, ..., x_N$, along with corresponding target values $t = t_1, \cdots, t_N$, we can construct the corresponding likelihood function.

$$p(t|X, w, \beta) = \prod_{n=1}^{N} p(t_n|x_n, w, \beta)$$

Taking the negative logarithm, we get the error function:

$$\frac{\beta}{2}\sum_{n=1}^{N}[y(x_n, w) - t_n]^2 - \frac{N}{2}ln\beta + \frac{N}{2}ln(2\pi)$$

Later, we can discuss the Bayesian Neural Networks, but here we can discuss about the **maximum likelihood approach**. Sum of squares error function :

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}[y(x_n, w) - t_n]^2$$

Let w_{ML} correspond to the maximum likelihood solution. Value of β can be found by minimizing the negative log likelihood to give

$$\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n=1}^{N}[y(x_n, w_{ML}) - t_n]^2$$

This can be evaluated once the iterative optimization required to find w_{ML} is completed. Also :

$$\frac{\partial E}{\partial a_k} = y_k - t_k$$

Using the cross-entropy error function instead of the sum-of-squares for a classification problem leads to faster training as well as improved generalization. Cross entropy error function :

$$E(w) = -\sum_{n=1}^{N}[t_n ln y_n + (1 - t_n)ln(1 - y_n)]$$

Finally, we consider the standard **multiclass** classification problem in which each input is assigned to one of K mutually exclusive classes. The binary target variables $t_k \in [0, 1]$ have

a $1-of-K$ coding scheme indicating the class and the network outputs are interpreted as $y_k(x,w) = p(t_k = 1|x)$ leading to the following function :

$$E(w) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} ln y_k(x_n, w)$$

and the output unit activation function used for it is the softmax function :

$$y_k(x,w) = \frac{\exp(a_k(x,w))}{\sum_j \exp(a_j(x,w))}$$

3.5.1 Parameter Optimization

We turn next to the task of finding a weight vector w which minimizes the chosen function $E(w)$. Note that if we make a small step in weight space from w to $w + \delta w$ then the change in the error function is $\delta E\ \delta w^{\intercal}\nabla E(w)$, where the vector $\nabla E(w)$ points in the direction of greatest rate of increase of the error function. We resort to iterative numerical procedures for finding an analytical solution. Most techniques involve choosing some initial value $w^{(0)}$ for the weight vector and then moving through weight space in a succession of steps of the form

$$w^{(\tau+1)} = w^{(\tau)} + \Delta w^{(\tau)}$$

Many algorithms make use of gradient information and so they require that after each update the value $\nabla E(w)$ is evaluated at the new vector weight $w^{(\tau+1)}$. In order to understand the importance of gradient information, let's consider this :

3.5.2 Local Quadratic Approximation

Consider the Taylor expansion of $E(w)$ around some point $\hat{w}$ in weight space

$$E(w) \simeq E(\hat{w}) + (w-\hat{w})^T b + \frac{1}{2}(w-\hat{w})^T H (w-\hat{w})$$

Here **b** is defined to be the gradient of E evaluated at $\hat{w}$.

$$b \equiv \nabla E|_{w=\hat{w}}$$

and the **Hessian Matrix** $H = \nabla\nabla E$ has elements

$$(H)_{ij} \equiv \frac{\partial E}{\partial w_i \partial w_j}|_{w=\hat{w}}$$

The corresponding local approximation to the gradient is given by

$$\nabla E \simeq b + H(w-\hat{w})$$

For points w that are sufficiently close to $\hat{w}$, these expressions will give reasonable approximations for the error and its gradient.

Consider the particular case of a local quadratic approximation around a point w^* that is a minimum of the error function. In this case there is no linear term, because $\nabla E = 0$ at w^*

$$E(w) = E(w^*) + \frac{1}{2}(w-w^*)H(w-w^*)$$

where the Hessian $\mathbb{H}$ is evaluated at w^*. In order to interpret this geometrically, consider the eigenvalue equation for the Hessian matrix

$$Hu_i = \lambda_i u_i$$

where the eigenvectors u_i form a complete **orthonormal**(important) set. Now we can write :

$$w - w^* = \sum_i \alpha_i u_i$$

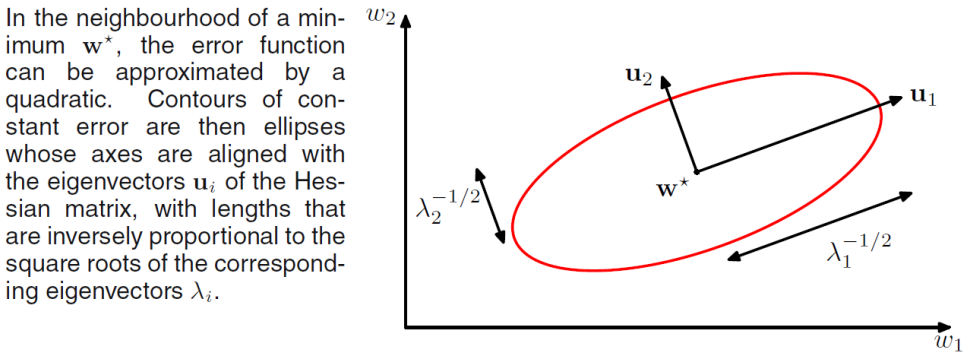
This can be in simple terms regarded as the transformation of coordinate system, in which the origin is translated to the point w^*, and the axes are rotated to align with the eigenvectors. the

error function :

$$E(w) = E(w^*) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2$$

A matrix $\mathbb{H}$ is said to be positive definite iff,

$$v^T H v > 0, \forall v$$



Because the eigenvectors $[u_i]$ form a complete set, an arbitrary vector v can be written in the form.

$$v = \sum_i c_i u_i$$

Then we have :

$$v^T H v = \sum_i c_i^2 \lambda_i$$

so $\mathbb{H}$ will be positive definite iff, all of its eigenvalues are positive. For a one-dimensional weight space, a stationary point w^* will be a minimum if

$$\frac{\partial^2 E}{\partial w^2}|_{w^*} > 0$$

The corresponding result in D-dimensions is that the **Hessian** matrix, evaluated at w^*, should be positive definite.

3.5.3 Use Of Gradient Information For Reducing Complexity

It is possible to evaluate the gradient of an error function efficiently by means of the **backpropagation** procedure. The use of this **gradient information** can lead to significant improvements in the speed with which the minima of the error function can be located.

In the quadratic approximation to the error function, the error is specified by the quantities b and $\mathbb{H}$, which contains a total of $W(W+3)/2$ independent elements because $\mathbb{H}$ is symmetric, where W is the dimensionality of w. The location of the minimum of this quadratic approximation depends on the $O(W^2)$ parameters and each of those would take $O(W)$ steps for evaluation. Thus time complexity is $O(W^3)$.

Now if we use the algorithm that makes use of gradient information,, each evaluation of ∇E brings W items of information. So we need just $O(W)$ gradient evaluations, each of which take $O(W)$ steps. So time complexity now becomes $O(W^2)$.

So we use Gradient Descent Optimization for reaching the error minima.

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)})$$

where $\eta > 0$ is known as the *learning rate*.

3.6 Error Backpropagation

Our goal in this section is to find an efficient technique for evaluating the gradient of an error function $E(w)$ for a feed-forward neural network.

3.6.1 Evaluation Of Error-Function Derivatives

We now derive the backpropagation algorithm for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions, and a broad class of error function.

$$E(w) = \sum_{n=1}^{N} E_n(w)$$

We will consider the problem of evaluating $\nabla E_n(w)$ for one such term in the error function. Consider the first simple linear model :

$$y_k = \sum_i w_{ki} x_i$$

together with an error function which takes the form :

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2$$

where $y_{nk} = y_k(x_n, w)$. The gradient of the error function becomes :

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni}$$

We shall now see how this simple result extends to the more complex setting of multilayer feed-forward networks. In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i$$

where z_i is the activation of a unit. This sum is transformed by a non-linear activation function h to give the activation z_j of unit j in the form

$$z_j = h(a_j)$$

E_n depends on the weight w_{ji} only via the summed input a_j to unit j. Therefore we can apply partial chain rule

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

A new notation $\implies$

$$\delta_j(error) = \frac{\partial E_n}{\partial a_j}$$

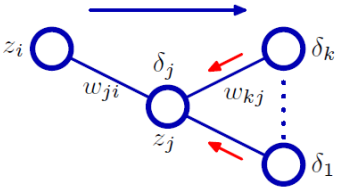
Using the last few equations we get :

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

$$\implies \frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$$

this is our required derivative and its obtained simple by multiplying the value of δ**error** for the unit at the output end of the weight by the value of z for the unit at the input end of the weight.

Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.



Thus, in order to evaluate the derivatives, we need only to calculate the value of δ_j for each hidden and output unit in the network and then apply the above equation. We will make use of chain rule fir partial derivatives :

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j}$$

where the sum runs over all units k to which unit j sends connections. If we now substitute the definition of δ, and make use of the above equations we get the *backpropagation formula*,

$$\delta_j = h^{'}(a_j)\sum_k w_{kj}\delta_k$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network, as illustrated in the figure above.

So, key points of error backpropagation :

1. Apply an input vector x_n to the network and forward propagate through the network using :
$$a_j = \sum_i w_{ji}z_i \ and \ z_j = h(a_j)$$
2. Evaluate the δ_k for all the output unit using
$$\delta_k = y_k - t_k$$
3. Backpropagate the δ's using $\delta_j = h^{'}(a_j)\sum_k w_{kj}\delta_k$ to obtain δ_j for each hidden unit in the network.
4. Use $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$ to evaluate the required derivatives.

3.6.2 A Simple Example

The above derivation of the backpropagation procedure allowed for general forms for the error function, the activation functions, and the network topology. So for illustrating this let's consider a two-layer network together with a sum-of-squares error, in which the output units have linear activation functions, so that $y_k = a_k$, while the hidden units have logistic sigmoid activation functions given by

$$h(a) \equiv tanh(a)$$

Its derivative :

$$h^{'}(a) = 1 - h(a)^2$$

Standard sum-of-error function -

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2$$

where, y_k is the activation of output unit k and t_k is the corresponding target for a particular input pattern x_n. For each pattern in training set we first perform a forward propagation using

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)}x_i$$

$$z_j = tanh(a_j)$$

$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j$$

Now, we compute the δ's for each output using

$$\delta_k = y_k - t_k$$

Then we backpropagate these to obtain the δ's for the hidden unit using

$$\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k$$

Finally, we get the derivatives wrt the first layer and second layer weights

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i \ and \ \frac{\partial E_n}{\partial w_{ji}^{(2)}} = \delta_k z_j$$

3.7 Convolutional Networks

Let's talk about this approach of creating models, that are invariant to certain transformation of inputs. For this approach of convolutional neural networks, we need to build the invariance properties into the structure of a neural network.

These notions are incorporated into convolutional neural networks through three mechanisms: (i) local receptive fields, (ii) weight sharing, and (iii) subsampling. In the convolutional layer the units are organized into planes each of which is called a *feature map*. Units in a feature map each take inputs only from a small subregion of the image, and all of the units in a feature map are constrained to share the same weight values. For instance, a feature map might consist of 100 units arranged in a 10×10 grid, with each unit taking inputs from a 5×5 pixel patch of the image. The whole feature map therefore has 25 adjustable weight parameters plus one adjustable bias parameter.

Input values from a patch are linearly combined using the weights and the bias, and the result transformed by a sigmoidal nonlinearity. If we think of the units as feature detectors, then all of the units in a feature map detect the same pattern but at different locations in the input image. If the input image is shifted, the activations of the feature map will be shifted by the same amount but will otherwise be unchanged.

The outputs of the convolutional units form the inputs to the **subsampling** layer of the network.

In a practical architecture, there may be several pairs of convolutional and subsampling layers. At each stage there is a larger degree of invariance to input transformations compared to the previous layer. The whole network can be trained by error minimization using backpropagation to evaluate the gradient of the error function. This involves a slight modification of the usual backpropagation algorithm to ensure that the shared-weight constraints are satisfied. Due to the use of local receptive fields, the number of weights in the network is smaller than if the network were fully connected. Furthermore, the number of independent parameters to be learned from the data is much smaller still, due to the substantial numbers of constraints on the weights.

In the last chapter we will discuss about the **Summer of Code** project of mine and the hyperparameter tuning I did in the **Gesture Detection** models.

Chapter 4

Building A Basic Deep Neural Net Library From Scratch

4.1 Introduction

In this chapter we are going go through the key mathematical points of ML and make a Python library from scratch to build deep neural networks with a variety of layers (Fully Connected Dense, Activation, etc.).

4.2 In Broader Terms :

The broad terms to be kept in mind :

1. We feed the **input** data to the neural network.
2. The **output** data propagates through the layers.
3. Using the output, we calculate the **error** which is a scalar.
4. Finally we adjust the given **weight param** by subtracting the derivative of the **error** with respect to the parameter itself.
5. We iterate.

Importance of Step 4 : If we add or modify one layer from the network,the output changes, which indeed changes the error, which in turn might change the derivative of the error w.r.t the parameters. We need to be able to compute the derivatives regardless of the **network architecture, regardless of the activation functions** and **regardless of the loss we use.**

In order to achieve that, we must implement **modular use of each layer.**

We will be using our prior knowledge of *Neural Networks* gained from this report and introduce the key points required to implement a working Neural Network Framework. This will make this clearer and press upon their real significance !

4.2.1 Layer Definition

Each layer(Dense, MaxPooling, Convolutional, Dropout, etc.) have **input** and **output** in common.

$$X \rightarrow \lessdot \supset \curvearrowright \searrow \rightarrow Y$$

Forward Propagation

Output of one layer is the input for the next layer. This is called **forward propagation**. We give the input data to the first layer, then the output of every layer becomes the input of the next layer until we reach the end of the network.

After comparing the result of the network output and the desired output we can calculate an error. The goal is to minimize this error. This is called **backward propagation**.

Gradient Descent

We have discussed this in detail earlier. So here's a quick reminder.

Basically, we want to change some parameter in the network (call it w) so that the total error E decreases. We do it by gradient descent :

$$w \leftarrow w - \alpha \frac{\partial E}{\partial w}$$

Where, α is a parameter in the range [0,1] and its called the **learning rate**. We need to be able to find the value of $\frac{\partial E}{\partial w}$ for all network architectures.

Backward Propagation

If, we give a layer the derivative of the error with respective to its output ($\frac{\partial E}{\partial Y}$) then it must be able to provide the derivative of the error with respect to its input ($\frac{\partial E}{\partial X}$).

$$\frac{\partial E}{\partial X} \leftarrow \lessdot \supset \curvearrowright \searrow \leftarrow \frac{\partial E}{\partial Y}$$

The trick here, is that if we have access to $\frac{\partial E}{\partial Y}$ we can very easily calculate $\frac{\partial E}{\partial W}$ (if the layer has any trainable parameters) without knowing anything about the *network architecture !* We simply use the chain rule :

$$\frac{\partial E}{\partial w} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w}$$

The unknown is $\frac{\partial y_j}{\partial w}$ which we know, because we have access to every layers output ! So we can update our parameters since :

$$\frac{\partial E}{\partial Y} = [\frac{\partial E}{\partial y_1}, \frac{\partial E}{\partial y_2}, \cdots, \frac{\partial E}{\partial y_j}]$$

$\frac{\partial E}{\partial X}$ for one layer is the $\frac{\partial E}{\partial Y}$ for the previous layer. Therefore, we can propagate the error !

$$\frac{\partial E}{\partial x_i} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

All you need to understand is the backpropagation ! After that we'll be able to code a Deep Convolutional Network in no time !

4.2.2 Abstract Base Class : Layer

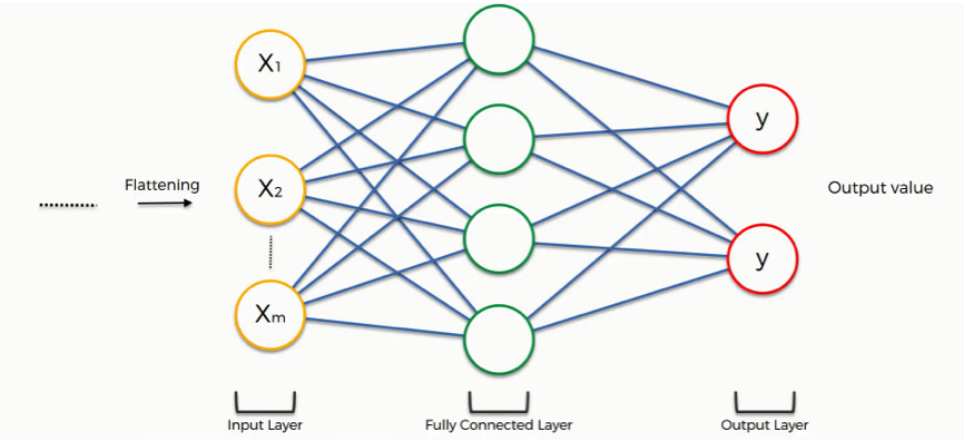
The abstract class *Layer*, which all other layers will inherit from, handles simple properties which are an **input**, an **output**, and both **forward** and **backward methods**.

[1]:
```
class Layer:
    def __init__(self):
        self.input = None
        self.output = None

    def forward_propagation(self, input):
        raise NotImplementedError

    def backward_propagation(self, output_error, learning_rate):
        raise NotImplementedError
```

The `learning_rate` parameter should be something like an update policy like an `optimizerin Keras`. But for simplicity, we are just going to be passing a learning rate and update out parameters using **gradient descent**.

4.2.3 Fully Connected Layer

A **fully connected** layer or **FC Layer** is a **Dense layer** and these are the most basic layers, as every input neurons are connected to every output neurons.



Forward Propagation

Value for each output neuron can be calculated as :

$$y_j = b_j + \sum_i x_i w_{ij}$$

This can be written in a simple dot product :

$$X = [x_1, \cdots, x_i], W = \begin{bmatrix} w_{11} & \cdots & w_{1j} \\ \vdots & \ddots & \vdots \\ w_{i1} & \cdots & w_{ij} \end{bmatrix}, B = [b_1, \cdots, b_j]$$

$$Y = XW + B$$

Backward Propagation

Suppose we have a matrix containing the derivative of the error with respect to that layer's output. We need :

1. The derivative of the error with respect to the parameters ($\frac{\partial E}{\partial W}$, $\frac{\partial E}{\partial B}$)
2. The derivative of the error wrt the input ($\frac{\partial E}{\partial X}$)

Let's calculate $\frac{\partial E}{\partial W}$:

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \cdots & \frac{\partial E}{\partial w_{1j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{i1}} & \cdots & \frac{\partial E}{\partial w_{ij}} \end{bmatrix}$$

Using the chain rule we can :

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_{ij}} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}x_i$$

Therefore,

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1}x_1 & \cdots & \frac{\partial E}{\partial y_j}x_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_1}x_i & \cdots & \frac{\partial E}{\partial y_j}x_i \end{bmatrix}$$

$$= \begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$= X^t \frac{\partial E}{\partial Y}$$

Lets calculate $\frac{\partial E}{\partial B}$:

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial b_1} & \frac{\partial E}{\partial b_2} & \cdots & \frac{\partial E}{\partial b_j} \end{bmatrix}$$

Again $\frac{\partial E}{\partial B}$ needs to be of the same size as B itself, one gradient per bias. We can use the chain rule again :

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial b_j} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial b_j}$$

$$= \frac{\partial E}{\partial y_j}$$

So we conclude that :

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$= \frac{\partial E}{\partial Y}$$

We can now get our $\frac{\partial E}{\partial X}$:

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial x_2} & \cdots & \frac{\partial E}{\partial x_i} \end{bmatrix}$$

Again using chain rule,

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \cdots + \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

$$= \frac{\partial E}{\partial y_1} w_{i1} + \cdots + \frac{\partial E}{\partial y_j} w_{ij}$$

The whole matrix :

$$\frac{\partial E}{\partial X} = \begin{bmatrix} (\frac{\partial E}{\partial y_1} w_{11} + \cdots + \frac{\partial E}{\partial y_j} w_{1j}) & \cdots & (\frac{\partial E}{\partial y_1} w_{i1} + \cdots + \frac{\partial E}{\partial y_j} w_{ij}) \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix} \begin{bmatrix} w_{11} & \cdots & w_{i1} \\ \vdots & \ddots & \vdots \\ w_{1j} & \cdots & w_{ij} \end{bmatrix}$$

$$= \frac{\partial E}{\partial Y} W^t$$

We have the three formulas for the FC layer !

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} W^t$$

$$\frac{\partial E}{\partial W} = X^t \frac{\partial E}{\partial Y}$$

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}$$

Where $\frac{\partial E}{\partial Y}$ = `output_error`

We can now code the fully connected layer :

FC Layer Code :

[2]:
```
import numpy as np

class FCLayer(Layer):
    # input_size = number of input neurons
    # output_size = number of output neurons
    def __init__(self, input_size, output_size):
        #Random Initialization
        self.weights = np.random.rand(input_size, output_size) - 0.5
        self.bias = np.random.rand(1, output_size) - 0.5

    # returns output for a given input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = np.dot(self.input, self.weights) + self.bias
        return self.output

    # computes dE/dW, dE/dB for a given output_error=dE/dY. Returns␣
↪input_error=dE/dX.
    def backward_propagation(self, output_error, learning_rate):
        input_error = np.dot(output_error, self.weights.T)
        weights_error = np.dot(self.input.T, output_error)
        # dBias = output_error

        # update parameters
        self.weights -= learning_rate * weights_error
        self.bias -= learning_rate * output_error
        return input_error
```

4.2.4 Activation Layer

All the above calculations were obviously completely linear. But, we need to add **non-linearity** to the model by applying non-linear functions to the output od some layers.

We need to re-calculate $\frac{\partial E}{\partial X}$. Let `f` be the activation function and `f'` be the derivative of it.

Forward Propagation

For a given input X , the output is simply the activation function applied to every element of X . Which means input and output have the same dimensions :

$$Y = \begin{bmatrix} f(x_1) & \cdots & f(x_i) \end{bmatrix}$$

$$= f(X)$$

Backward Propagation

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \cdots & \frac{\partial E}{\partial x_i} \end{bmatrix}$$

$$= \left[\frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \cdots + \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial x_i}\right]$$

$$= \left[\frac{\partial E}{\partial y_1}f'(x_1) + \cdots + \frac{\partial E}{\partial y_i}f'(x_i)\right]$$

$$= \begin{bmatrix}\frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_i}\end{bmatrix} \odot \begin{bmatrix}f'(x_1) & \cdots & f'(x_i)\end{bmatrix}$$

$$= \frac{\partial E}{\partial Y} \odot f'(X)$$

The $\odot$ is a **element-wise multiplication symbol**

Activation Layer Code

[3]:
```
# inherit from base class Layer
class ActivationLayer(Layer):
    def __init__(self, activation, activation_prime):
        self.activation = activation
        self.activation_prime = activation_prime

    # returns the activated input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    # Returns input_error=dE/dX for a given output_error=dE/dY.
    # learning_rate is not used because there is no "learnable" parameters.
    def backward_propagation(self, output_error, learning_rate):
        return self.activation_prime(self.input) * output_error
```

We can also write some of our activation functions and their derivatives in a separate file.

[4]:
```
import numpy as np

# Tan
def tanh(x):
    return np.tanh(x)

def tanh_prime(x):
    return 1-np.tanh(x)**2

# Sigmoid
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def sigmoid_prime(x):
    y = sigmoid(x)
    return y*(1 - y)
```

```
#ReLU
def relu(x):
    return np.maximum(x, 0)

def relu_prime(x):
    return 1. * (x > 0)
```

4.2.5 Loss Function

The error of the network, which measures how good or bad the network did for a given input data is defined by us. One of the most common is called **MSE - Mean Squared Error**.

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i^* - y_i)^2$$

You can think of the loss as a last layer which takes all the output neurons and squashes them into one single neuron. What we need now, as for every other layer, is to define $\frac{\partial E}{\partial Y}$

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_i} \end{bmatrix}$$

$$= \frac{2}{n}\begin{bmatrix} y_1 - y_1^* & \cdots & y_i - y_i^* \end{bmatrix}$$

$$= \frac{2}{n}(Y - Y^*)$$

Loss Function Code

[5]:
```
import numpy as np

# loss function and its derivative
def mse(y_true, y_pred):
    return np.mean(np.power(y_true-y_pred, 2))

def mse_prime(y_true, y_pred):
    return 2*(y_pred-y_true)/y_true.size
```

4.2.6 Network Class

We'll finally create a new `Network` class to create and train our neural networks.

[6]:
```
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.loss_prime = None

    # add layer to network
    def add(self, layer):
        self.layers.append(layer)
```

```
    # set loss to use
    def use(self, loss, loss_prime):
        self.loss = loss
        self.loss_prime = loss_prime

    # predict output for given input
    def predict(self, input_data):
        # sample dimension first
        samples = len(input_data)
        result = []

        # run network over all samples
        for i in range(samples):
            # forward propagation
            output = input_data[i]
            for layer in self.layers:
                output = layer.forward_propagation(output)
            result.append(output)

        return result

    # train the network
    def fit(self, x_train, y_train, epochs, learning_rate):
        # sample dimension first
        samples = len(x_train)

        # training loop
        for i in range(epochs):
            err = 0
            for j in range(samples):
                # forward propagation
                output = x_train[j]
                for layer in self.layers:
                    output = layer.forward_propagation(output)

                # compute loss (for display purpose only)
                err += self.loss(y_train[j], output)

                # backward propagation
                error = self.loss_prime(y_train[j], output)
                for layer in reversed(self.layers):
                    error = layer.backward_propagation(error, learning_rate)

            # calculate average error on all samples
            err /= samples
            print('epoch %d/%d   error=%f' % (i+1, epochs, err))
```

4.3 Building Neural Networks

Now, we can use our class to create a Neural Network ! We are going to build two neural networks : **a simple XOR** and **a MNIST solver**.

4.3.1 XOR Solver

It's a common way to start testing your neural network using a simple XOR !

[7]:
```
import numpy as np

# training data
x_train = np.array([[[0,0]], [[0,1]], [[1,0]], [[1,1]]])
y_train = np.array([[[0]], [[1]], [[1]], [[0]]])

# network
net = Network()
net.add(FCLayer(2, 3))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(3, 1))
net.add(ActivationLayer(tanh, tanh_prime))

# train
net.use(mse, mse_prime)
net.fit(x_train, y_train, epochs=1000, learning_rate=0.1)

# test
out = net.predict(x_train)
print(out)
```

```
epoch 1/1000   error=0.544856
epoch 2/1000   error=0.357912
epoch 3/1000   error=0.322824
epoch 4/1000   error=0.312566
epoch 5/1000   error=0.308259
.
.
.
epoch 996/1000   error=0.000381
epoch 997/1000   error=0.000380
epoch 998/1000   error=0.000380
epoch 999/1000   error=0.000379
epoch 1000/1000   error=0.000378
[array([[0.00134211]]), array([[0.97179254]]), array([[0.9733828]]),
array([[-0.0023152]])]
```

That's near perfect ! Let's do the same with MNIST :

4.3.2 Solve MNIST

MNIST Dataset consists of images of digits from **0** to **9**, of shape **28x28x1**. The goal is to predict what digit is drawn on a picture.

[8]:
```
import numpy as np
from keras.datasets import mnist
from keras.utils import np_utils

# load MNIST from server
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# training data : 60000 samples
# reshape and normalize input data
x_train = x_train.reshape(x_train.shape[0], 1, 28*28)
x_train = x_train.astype('float32')
x_train /= 255
# encode output which is a number in range [0,9] into a vector of size 10
# e.g. number 3 will become [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
y_train = np_utils.to_categorical(y_train)

# same for test data : 10000 samples
x_test = x_test.reshape(x_test.shape[0], 1, 28*28)
x_test = x_test.astype('float32')
x_test /= 255
y_test = np_utils.to_categorical(y_test)

# Network
net = Network()
net.add(FCLayer(28*28, 100))                # input_shape=(1, 28*28)    ;
 ↪output_shape=(1, 100)
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(100, 50))                   # input_shape=(1, 100)      ;
 ↪output_shape=(1, 50)
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(50, 10))                    # input_shape=(1, 50)       ;
 ↪output_shape=(1, 10)
net.add(ActivationLayer(tanh, tanh_prime))

# train on 1000 samples
# as we didn't implemented mini-batch GD, training will be pretty slow if we
 ↪update at each iteration on 60000 samples...
net.use(mse, mse_prime)
net.fit(x_train[0:1000], y_train[0:1000], epochs=35, learning_rate=0.1)

# test on 3 samples
out = net.predict(x_test[0:3])
print("\n")
print("predicted values : ")
print(out, end="\n")
print("true values : ")
print(y_test[0:3])
```

```
Using TensorFlow backend.

epoch 1/35   error=0.232654
```

```
epoch 2/35   error=0.096434
epoch 3/35   error=0.077761
epoch 4/35   error=0.066916
epoch 5/35   error=0.0590363
.
.
.
epoch 31/35   error=0.011761
epoch 32/35   error=0.011311
epoch 33/35   error=0.010829
epoch 34/35   error=0.010454
epoch 35/35   error=0.010081


predicted values :
[array([[ 0.00148303,  0.01767697, -0.04613821,  0.03588162, -0.02785326,
        -0.03008815,  0.04855139,  0.9703428 ,  0.15975008, -0.17255709]]),
array([[ 0.45671587, -0.0235585 ,  0.82551488,  0.00215151, -0.05089819,
        -0.09097523,  0.0714639 , -0.03358172, -0.04276478,  0.05011182]]),
array([[ 0.00655831,  0.97988483, -0.02336553,  0.114868  , -0.02212076,
        -0.1761728 ,  0.00937956,  0.17123095, -0.0439778 ,  0.0195196 ]])]
true values :
[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

PERFECT this works too ! This is how we create basic Deep Learning Neural Networks in no time ! This surely gives us the strength and courage to pry deeper into the hidden layers of Convolutional Neural Networks and discover the patterns they encode in there. So, in the next chapter we are going to visualize what patterns are encoded in the hidden layers, using something called **De-Convolutional Networks**, .

Chapter 5

A Review On Visualizing and Understanding Convolutional Networks

5.1 Introduction

Here, I am going to summarize a paper by **Matthew D. Zeiler and Rob Fergus from the Dept. of Computer Science, New York University, USA**. Its a quite recent paper of 2014 titled "Visualizing and Understanding Convolutional Networks" and it aims to understand the behind-the-scenes of Convolutional Neural Networks and help us extract specific features and exploit the concept of **CNN** even further.

As the abstract of the paper says, Large Convolutional Network models have recently demonstrated impressive classification performance. However, there is no clear understanding of why they perform so well, or how they might be improved.

In this summary of the mentioned paper, we will be talking mainly about a novel vizualization technique that gives insight into the functioning of intermediate feature layers and the operation of the classifier.

5.2 Basic Model Overview

Standard convnet models were used in the entire paper. These models map a 2D input image, via a series of layers, to a probability vector, over a number of different classes.

5.2.1 Layer Description

Each layer consists of :

1. Convolution of the previous layer output with a set of learned filters.
2. Passing the responses through a rectified linear function ($relu(x) = max(x, 0)$)
3. Max Pooling over local neighbourhoods(optional)
4. A local contrast operation that normalizes the reponses across feature maps.(optional)

Additionally, the top few layers are conventional fully-connected networks and the final layers is a *softmax classifier*.

5.2.2 Basic Training Details

The models were trained using a large set of labeled images, where *label* is a discrete variable indicating the **true class**. **Cross-Entropy** was used as a *loss function*, and the parameters of the network (filters in the convolutional layers, weight matrices in the fully connected layers and biases) are trained by *back-propagating* the derivative of the *loss with respect to the parameters throughout the network*, and updating the parameters via **Stochastic Gradient Descent**.

5.3 Visualization

Understanding the operation of a convnet requires interpreting the feature activity in intermediate layers. Here we will talk about a good way to map these activities back to the **input pixel space**, showing what input pattern originally caused a given activation in the feature maps.

5.3.1 Visualization with a DeConvolutional Network(deconvnet)

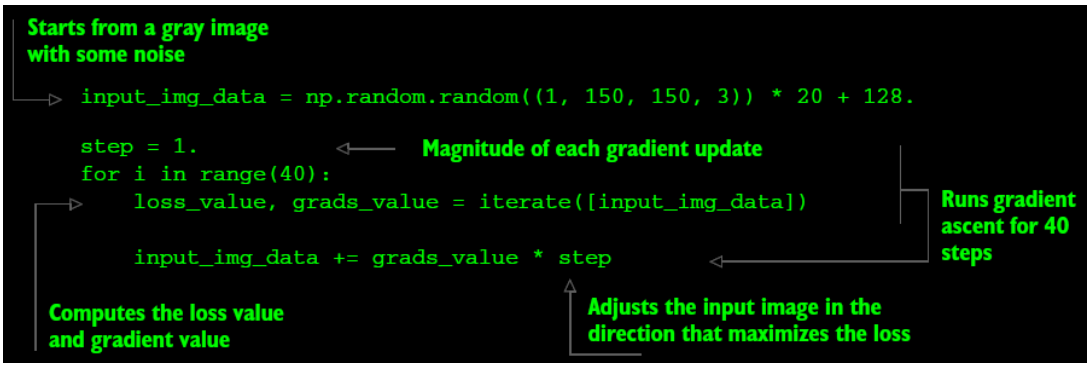
A deconvnet can be thought of as a convnet model that uses the same components but in reverse. To visualize a convnet, a *deconvnet* is attached to each of its layers, providing a perfect path back to its **input pixel space**.

To examine a given convnet activation, we successively **(i)** unpool, **(ii)** rectify, and **(iii)** filter to reconstruct the layer beneath that gave rise to the activation which the neural net chose. This process is repeated until *input pixel space* is reached.

Let's explore each step further :

Unpooling

In the convnet, the max pooling operation is non-invertible. But, we can obtain an approximate inverse by recording the locations of the maxima within each pooling region. We can do so, by using the (opposite of Stochastic Gradient Descent), Stochastic Gradient Ascent for a few steps until we reach satisfactorily close to our unpooled convnet.



This is just an example code snippet from Python performing **Gradient Ascent** and **iterate** is a **Keras** backend function to compute the value of loss tensor and the gradient tensor, given a NumPy Tensor.

Rectification

The convnet uses **relu non-linearities**, which rectify the feature maps thus ensuring the feature maps are always positive. To obtain valid feature reconstructions at each layer, we pass the reconstructed signal through a **relu non-linearity**.

Filtering

The convnet uses learned filters to convolute the feature maps from the previous layer. To approximately invert this, the deconvnet uses transposed versions of the same filters, which means flipping each filter vertically and horizontally.

5.3.2 Convnet Visualization

We can show the practical use of deconvnet to visualize the feature activations, on the ImageNet validation set.

Feature Visualization :

In the figure 2 below, the top 9 activations, each projected separately down to its input pixel space, have been shown, also showing its invariance to input deformations.

Quoting the **original** excerpt from the paper :

> The projections from each layer show the hierarchical nature of the features in the network. Layer 2 responds to corners and other edge/color conjunctions. Layer 3 has more complex invariances, capturing similar textures (e.g. mesh patterns (Row 1, Col 1); text (R2,C4)). Layer 4 shows significant variation, and is more class-specific: dog faces (R1,C1); bird's legs (R4,C2). Layer 5 shows entire objects with significant pose variation, e.g. keyboards (R1,C11) and dogs (R4).

A natural question that comes up is if the model truly identifies the location of the object in the image or if it just uses the surrounding context to get the picture. To check this, the author systematically occludes different portions of the input image using a grey square. The outputs of the classifier model, clearly showed that the model was localizing the objects within the scene and the probability of the prediction drops significantly on occlusion.

Another surprising observation was that removing two of the middle convolutional layers made a relatively small difference to the error rate. However, removing both the middle convolution layers and the fully connected layer yielded a model with only 4 layers whose performance was dramatically worse. This would suggest that the overall depth of the model is important for obtaining good performance, but changing the size of the fully connected layers made little difference to performance, however, increasing the size of the middle convolution layers gave a useful gain in performance. But increasing these, while also enlarging the fully connected layers results in over-fitting.

5.4 Conclusion

This paper explored **large convolutional neural network models**, trained for image classification.

Firstly, the paper presented a novel way to visualize the activity within the model. This revealed the features to be far from random, uninterpretable patterns. Rather, they showed many intuitively desirable properties.

Secondly, it also showed how these visualizations can be used to identify problems within the model and so obtain better results.

Thirdly, it further explained that having a minimum depth to the network, rather than any individual section, is vital to the model's performance.

FIGURE 2 : Layer Activations in Deep Neural Network

But above all, the key takeaway from this paper is the concept of De-Convolutional Network, and its implementation.

5.5 References

- The Original Paper
- Deep Learning with Python by **François Chollet**

Chapter 6

Model Documentations

6.1 Introduction

Under my **SoC** project "Gestures For 3D Space" I had created a **Keras** based Deep CNN model for identifying Hand Gestures using direct image feed. For that project, I had to play around with **Transfer Learning**, using pretrained convnets like *VGG16, ResNet50, GoogleNET, DenseNet121* while exploring and fine-tuning the hyperparameters to deliver dependable results in real-life applications.

So here is a detailed documentation of the models with a **VGG16** convolutional base that I had used, along with the *Accuracy* and *Loss Curves*, *Confusion Matrices*, *F1 scores* and *Error Rates*.

6.2 Dataset Description

For this task to be successful and accurate, I had to create a relatively small dataset of 2750 different images for each class spread over 5 classes :

- **Class Indices** : {'cool': 0, 'fist': 1, 'ok': 2, 'stop': 3, 'yo': 4}
- **Train Set** : 375 images per class
- **Validation Set** : 125 images per class
- **Test Set** : 50 images per class (It's pretty less though)

6.2.1 Data Augmentation used :

```
rescale=1./255
rotation_range=45
width_shift_range=0.2
featurewise_center = True
featurewise_normalisation = True
height_shift_range=0.2
shear_range=0.2
zoom_range=0.2
horizontal_flip=True
fill_mode='nearest'
```

A sample of the dataset(Just to show what it looks like) :

6.3 Models

Convolution Base has been taken to be the **VGG16** model with an input size of $(224, 224, 3)$ with pretrained weights from **"imagenet"** training set.

6.3.1 Model 1

This was just a basic model to get a working baseline with our dataset.

Specifications

Model Classifier Top :

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(128,
          activation='relu'))
model.add(layers.Dense(128,
          activation='relu'))
model.add(layers.Dense(128,
          activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(5,
          activation='softmax'))
```

Model Compilation :

```
optimizer = Adam in default
loss = "categorical_crossentropy"
```

Model Fitting :

```
epochs = 50
steps_per_epoch = 100
batch_size = 15
validation_steps = 50
```

Results

Validation Accuracy : 90%

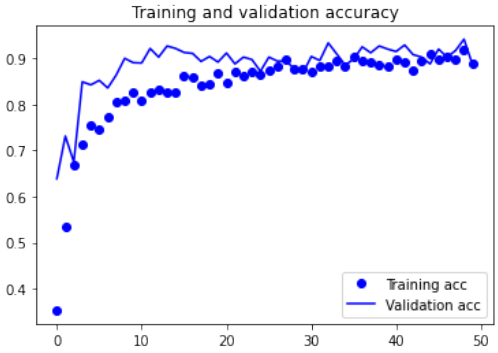
F1 Score : 0.8864

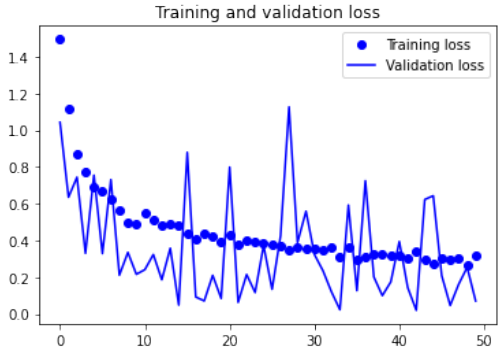
Confusion Matrix :

```
array([[26,  0,  4,  3, 17],
       [ 0, 48,  0,  2,  0],
       [ 0,  0, 50,  0,  0],
       [ 0,  0,  0, 50,  0],
       [ 0,  0,  0,  0, 50]],
      dtype=int64)
```

Seems like it is pretty heavily confusing, "cool" sign with "yo" which is kind of apparently obvious, since both of the gestures have 2 fingers.

Accuracy and Loss Curves





6.3.2 Model 2

Decreased the *learning rate* of the optimizer, increased the *train batch size* from *15* to *32*, set the *validation steps* and *steps per epoch* to something that seemed optimal and increased the *no of epochs* to analyse it further.

Specifications

Model Classifier Top :

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(128,
        activation='relu'))
model.add(layers.Dense(128,
        activation='relu'))
model.add(layers.Dense(128,
        activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(5,
        activation='softmax'))
```

Model Compilation :

```
optimizer = Adam(lr=2e-5)
loss = "categorical_crossentropy"
```

Model Fitting :

```
epochs = 100
steps_per_epoch = 120
batch_size = 32
validation_steps = None
```

Results

Validation Accuracy : 93%

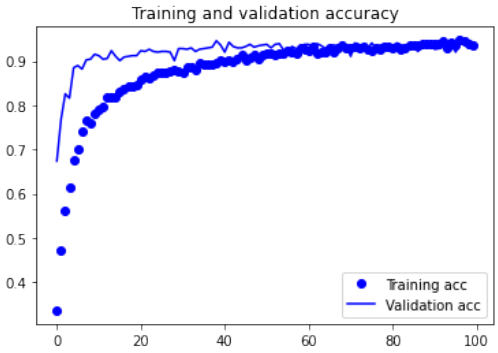
F1 Score : 0.9207

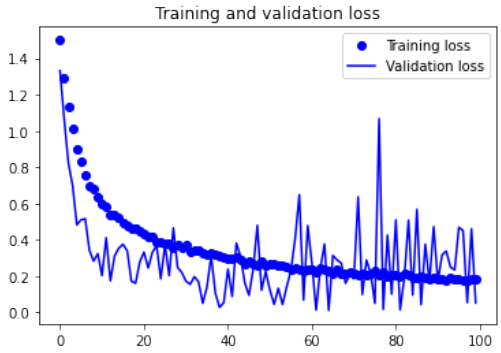
Confusion Matrix :

```
array([[35,  0,  4,  4,  7],
       [ 0, 48,  0,  2,  0],
       [ 0,  0, 50,  0,  0],
       [ 0,  0,  0, 50,  0],
       [ 0,  0,  0,  2, 48]],
      dtype=int64)
```

Seems like it is still heavily confusing, "cool" sign with "yo".

Accuracy and Loss Curves





6.3.3 Model 3

Increased the *learning rate* of the optimizer, did major changes to the model classifier(as listed below) and increased the *no of epochs* to analyse it further.

Specifications

Model Classifier Top :

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256,
        activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(5,
        activation='softmax'))
```

Model Compilation :

```
optimizer = Adam(lr=1e-4)
loss = "categorical_crossentropy"
```

Model Fitting :

```
epochs = 200
steps_per_epoch = 120
batch_size = 32
validation_steps = None
```

Results

Model is **UNDERFITTING** and the *loss* is still pretty high :(

Validation Accuracy : 95%

F1 Score : 0.9429

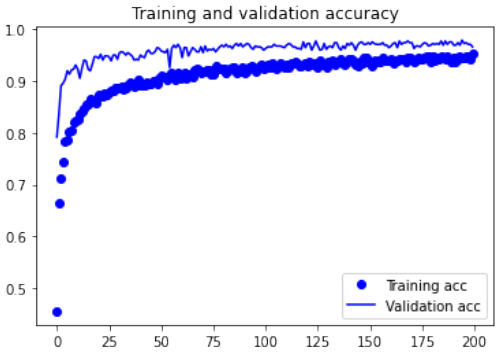
Confusion Matrix :

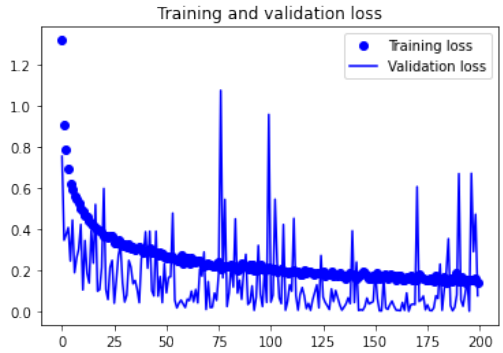
```
array([[40,  0,  3,  2,  5],
       [ 0, 49,  0,  1,  0],
       [ 0,  0, 50,  0,  0],
       [ 0,  0,  0, 50,  0],
       [ 0,  0,  0,  3, 47]],
      dtype=int64)
```

Seems like it is still confusing, “cool” sign with “yo” :(

But first, we need to fight the Underfit problem !

Accuracy and Loss Curves





6.3.4 Model 4

Removed the *Dropout layer* from the classifier top and further increased the number of *epochs* to analyse it further.

Specifications

Model Classifier Top :

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256,
        activation='relu'))
model.add(layers.Dense(5,
        activation='softmax'))
```

Model Compilation :

```
optimizer = Adam(lr=1e-4)
loss = "categorical_crossentropy"
```

Model Fitting :

```
epochs = 200
steps_per_epoch = 120
batch_size = 32
validation_steps = None
```

Results

Validation Accuracy : 96.8%

F1 Score : 0.9344

Confusion Matrix :

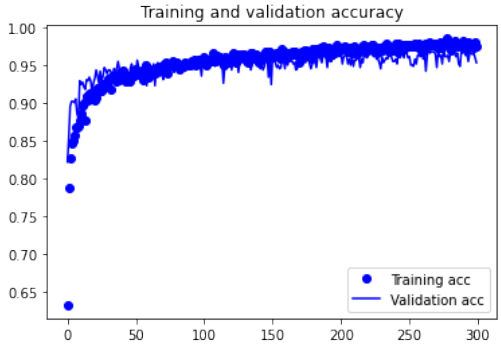
```
array([[38,  0,  2,  4,  6],
       [ 0, 47,  0,  3,  0],
       [ 0,  0, 50,  0,  0],
       [ 0,  0,  0, 50,  0],
       [ 0,  0,  0,  1, 49]],
       dtype=int64)
```

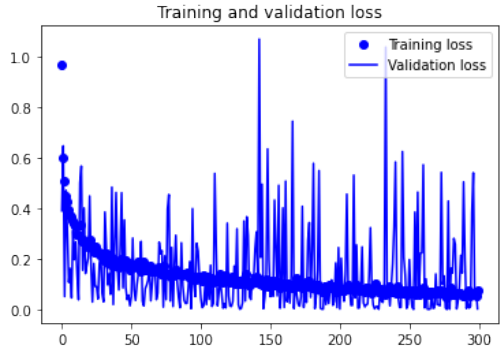
Seems like it is still confusing, “cool” sign with “yo” -_-

But, as we see the Underfit problem has been solved !

Now let’s fine tune the final layers of the freezed **VGG16** base to gain a performance boost !

Accuracy and Loss Curves





6.3.5 Model 4 Fine Tuned

Unfreezed `block5_conv1` layer of `vgg16` and retrained it with a reduced learning rate :

```
epochs = 100
optimizer = optimizers.Adam(lr=1e-4)
loss = "categorical_crossentropy"
```

Results

Validation Accuracy : 98.08%

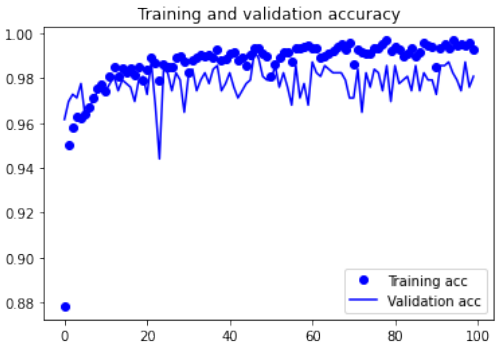
F1 Score : 0.9959

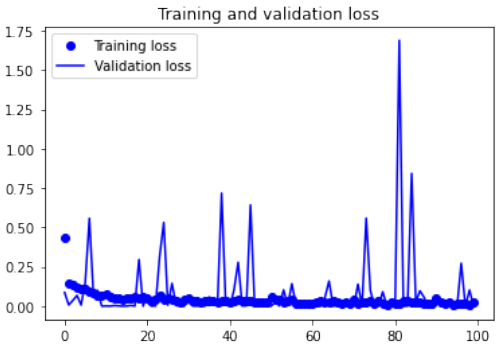
Seems like that fixed its confusion :)

Confusion Matrix :

```
array([[50,  0,  0,  0,  0],
       [ 0, 50,  0,  0,  0],
       [ 0,  0, 50,  0,  0],
       [ 0,  1,  0, 49,  0],
       [ 0,  0,  0,  0, 50]],
       dtype=int64)
```

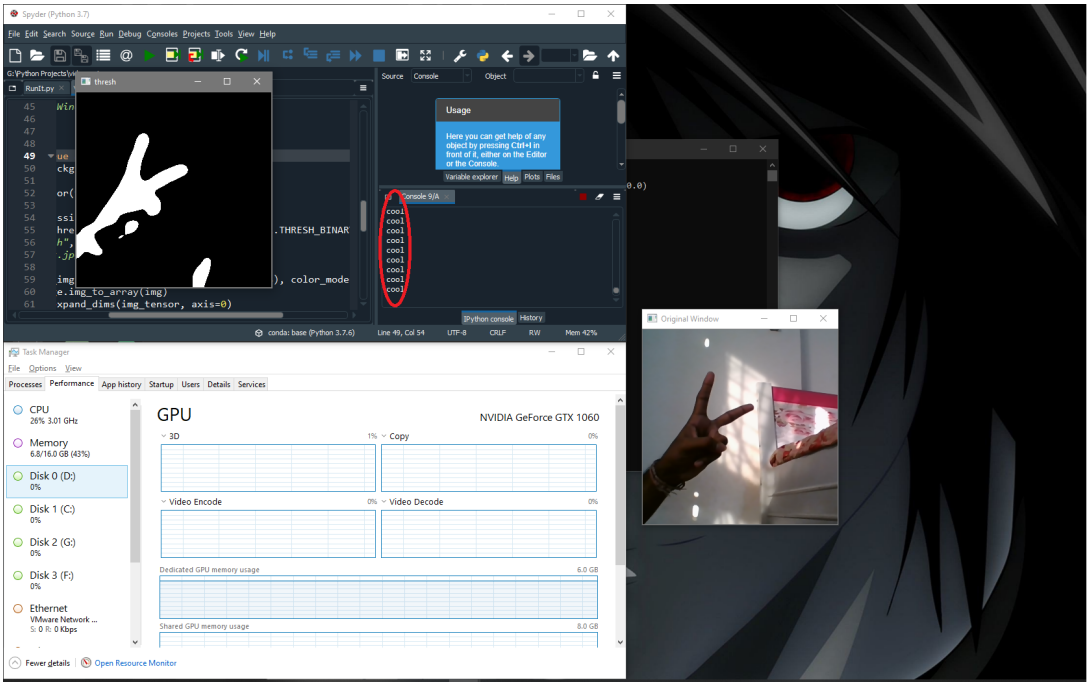
Accuracy and Loss Curves





6.4 Conclusion

Here's a random picture(Yes that's $\mathcal{L}$ in the background) of a very preliminary display of my working model :



With these **model documentations** from a practical and working example of a real-life application, I end my **Summer Of Science Neural Networks and Deep Learning Report**.

I hope this has accomplished to induce a spark in your **neurons** to venture further into the world of those mimicking black-boxes(as we call them now) and uncover some of the mysteries in which these are still shrouded !