

Neural Networks and Deep Learning

Summer Of Science 2020

IIT Bombay

Richeek Das

Mentor : Priya Singh

Contents

1	Mathematical Preliminaries	4
1.1	Linear Algebra Basics	4
1.1.1	Scalars, Vectors, Matrices and Tensors	4
1.1.2	Vectors and Vector Spaces	4
1.1.3	Norms	5
1.2	Eigendecomposition	5
1.3	Singular Value Decomposition	6
1.4	The Moore-Penrose Pseudoinverse	6
1.5	Principal Component Analysis	7
2	Working of Basic Machine Learning Algorithms	9
2.1	Linear Regression	9
2.1.1	The Normal Equation	10
2.1.2	Computational Complexity	12
2.2	Gradient Descent	13
2.2.1	Batch Gradient Descent	13
2.2.2	Stochastic Gradient Descent	15
2.2.3	Mini-batch Gradient Descent	17
2.3	Polynomial Regression	17
2.4	Learning Curves	19
2.5	Regularized Linear Models	21
2.5.1	Ridge Regression	21
2.5.2	Lasso Regression	22
2.5.3	Elastic Net	23
2.5.4	Early Stopping	23
2.6	Logistic Regression	24
2.6.1	Training and Cost Function	24
2.6.2	Softmax Regression	25
2.7	So a overview of things covered so far :	26
2.8	Decision Tree : An Overview	26
2.8.1	Decision rules	26
2.9	Random Forest	26
2.9.1	Comparison to Logistic Regression :	27
2.10	Conclusion	28
2.11	See Also	28
3	A Review On Visualizing and Understanding Convolutional Networks	29
3.1	Introduction	29
3.2	Basic Model Overview	29
3.2.1	Layer Description	29

3.2.2	Basic Training Details	30
3.3	Visualization	30
3.3.1	Visualization with a DeConvolutional Network(deconvnet)	30
3.3.2	Convnet Visualization	31
3.4	Conclusion	31
3.5	References	33

Chapter 1

Mathematical Preliminaries

1.1 Linear Algebra Basics

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 \cdot 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 $\psi_1, \psi_2 \dots \psi_n$ 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 \mathbf{W} if any vector in \mathbf{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 :

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

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

On an intuitive level, the norm of a vector \mathbf{x} measures the distance from the origin to the point \mathbf{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 :

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

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**, $(v(1), \dots, v(n))$, with corresponding eigenvalues $(\lambda_1, \dots, \lambda_n)$.

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

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

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:

$$A = UDV^T$$

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 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

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

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

$$A^+ = VD^+U^T$$

, 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 $[x(1), \dots, x(m)]$ 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,

$$\begin{aligned} & (x - g(c))^T (x - g(c)) \\ &= x^T x - 2x^T g(c) + g(c)^T g(c) \end{aligned}$$

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

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

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

$$\begin{aligned} c^* &= \arg \min_c -2x^T Dc + c^T I_l c \\ &= \arg \min_c -2x^T Dc + c^T c \end{aligned}$$

We can solve this optimization with vector calculus :

$$\begin{aligned} \nabla_c (-2x^T Dc + c^T c) &= 0 \\ -2D^T x + 2c &= 0 \\ c &= D^T x \end{aligned}$$

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**.

$$f(x) = D^T x$$

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

$$r(x) = g(f(x)) = DD^T x$$

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:

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

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

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

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 $\mathbf{X}_{i,:} = x^{(i)\top}$. Rewriting :

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

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

$$\begin{aligned} & \arg \min_d \|\mathbf{X} - \mathbf{X} d d^T\|_F^2 \\ &= \arg \min_d \text{Tr} \left((\mathbf{X} - \mathbf{X} d d^T)^\top (\mathbf{X} - \mathbf{X} d d^T) \right) \\ &= \arg \min_d \text{Tr} \left(\mathbf{X}^\top \mathbf{X} - \mathbf{X}^\top \mathbf{X} d d^T - d d^T \mathbf{X}^\top \mathbf{X} + d d^T \mathbf{X}^\top \mathbf{X} d d^T \right) \\ &= \arg \min_d -\text{Tr}(\mathbf{X}^\top \mathbf{X} d d^T) - \text{Tr}(d d^T \mathbf{X}^\top \mathbf{X}) + \text{Tr}(d d^T \mathbf{X}^\top \mathbf{X} d d^T) \end{aligned}$$

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

$$\begin{aligned} &= \arg \min_d -2\text{Tr}(\mathbf{X}^\top \mathbf{X} d d^T) + \text{Tr}(\mathbf{X}^\top \mathbf{X} d d^T d d^T) \\ & \quad \text{subject to } d^T d = 1 \end{aligned}$$

.

$$= \arg \min_d -2\text{Tr}(\mathbf{X}^\top \mathbf{X} d d^T) + \text{Tr}(\mathbf{X}^\top \mathbf{X} d d^T)$$

Therefore,

$$\begin{aligned} &= \arg \min_d -\text{Tr}(\mathbf{X}^\top \mathbf{X} d d^T), \text{ subject to } d^T d = 1 \\ &= \arg \min_d \text{Tr}(d^T \mathbf{X}^\top \mathbf{X} d) \end{aligned}$$

This optimization problem may be solved using **eigendecomposition**. Specifically, the optimal d is given by the **eigenvector** of \mathbf{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 :

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

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 :

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

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

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

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

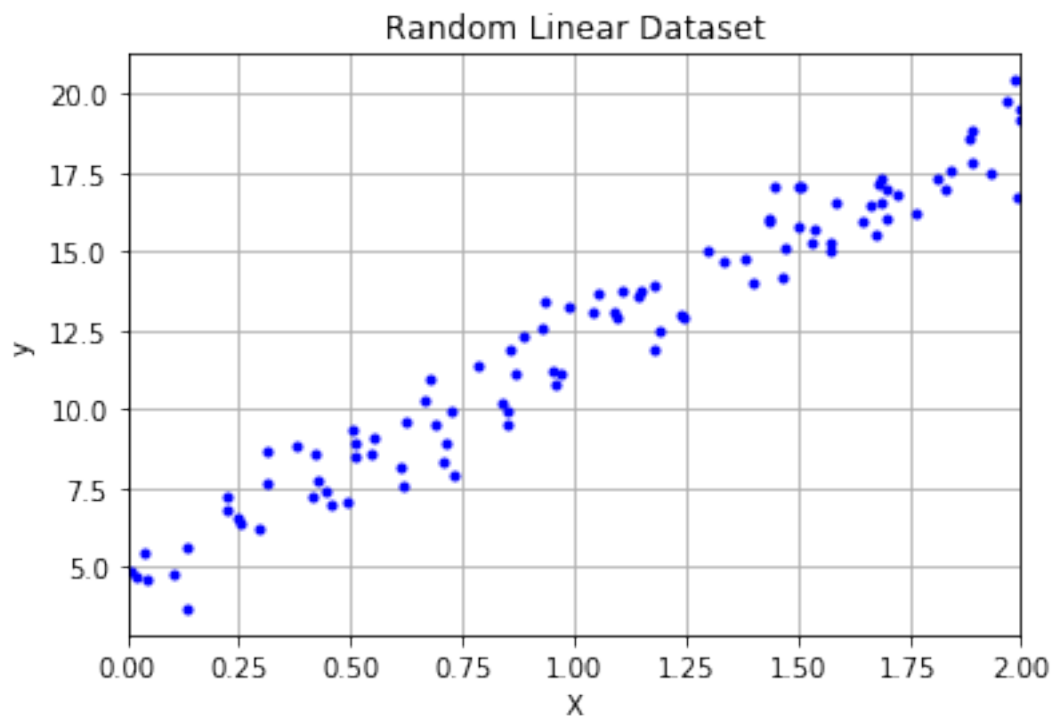
X = 2 * np.random.rand(100,1)
y = 5 + 7 * X + np.random.randn(100,1)
## Generates a random linear dataset with some Gaussian Noise
```

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
```

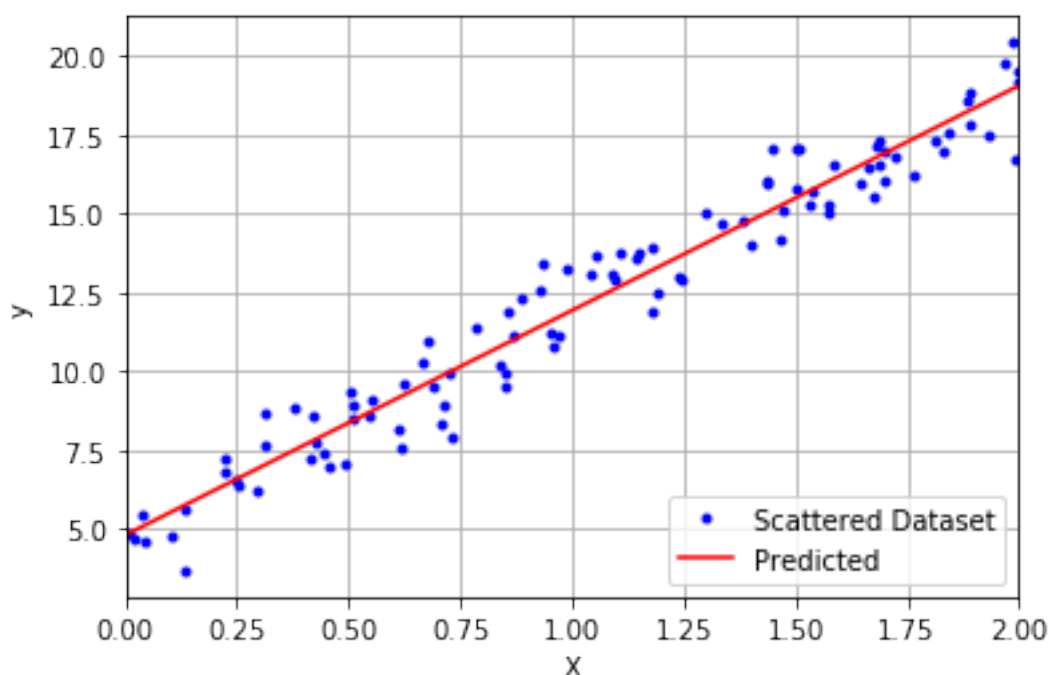
```
[[4.80833324]]
[[7.12658528]]
```

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 :

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

where, X^+ is the **pseudoinverse** of 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 $X^T X$ is not invertible, such as if its not a square matrix or if some features are redundant, but the pseudoinverse is always defined.

REMEMBER TO PUT THE LINK HERE

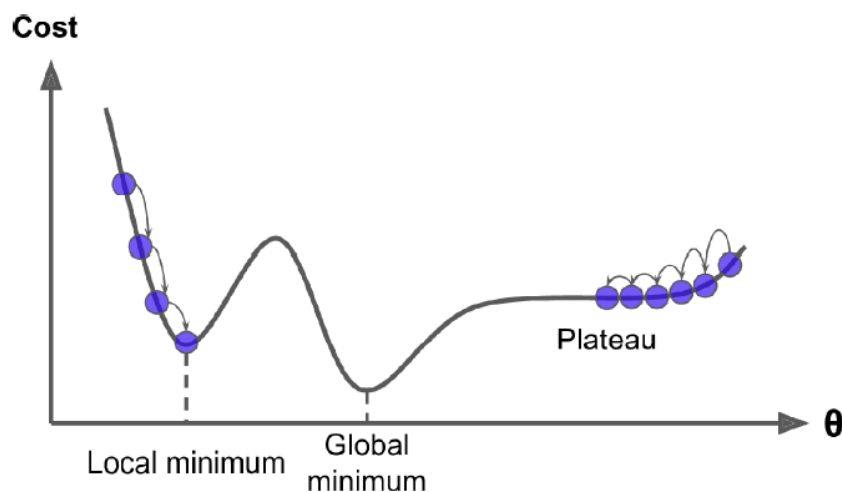
2.1.2 Computational Complexity

The **Normal Equation** computes the inverse of $X^T X$, 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 need to 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 :

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

Gradient vector of the cost function :

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

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 :

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

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

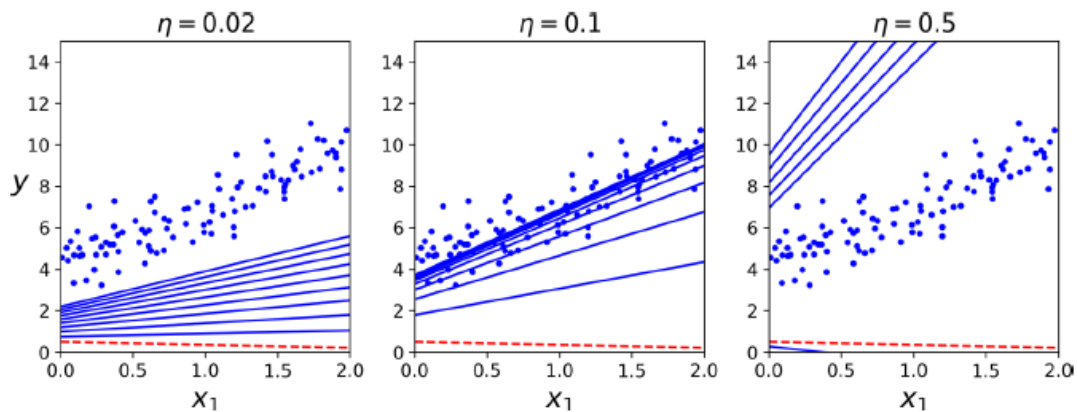
      for i in range(n_iterations):
          gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
          theta = theta - learning_rate*gradients
```

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*.

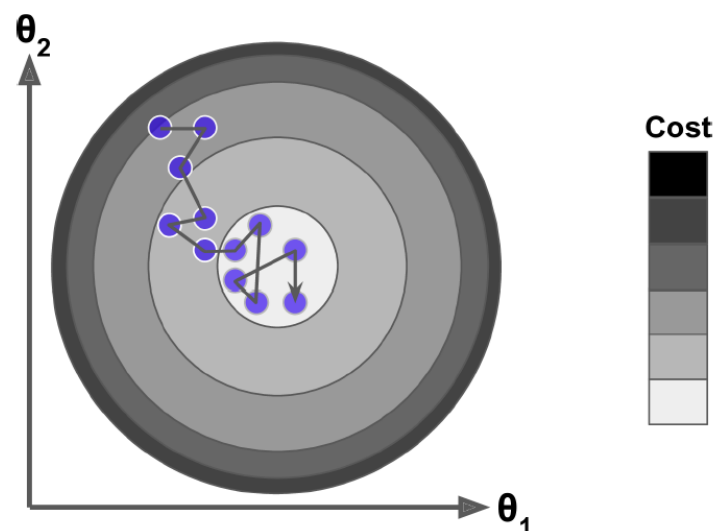


If its too low, then it'll take a lot of time to reach the optimal solution, whereas if its too high, the algorithm diverges and we won't even reach the solution. We can use Scikit GridSearchCV for finding the optimal hyperparameter.

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

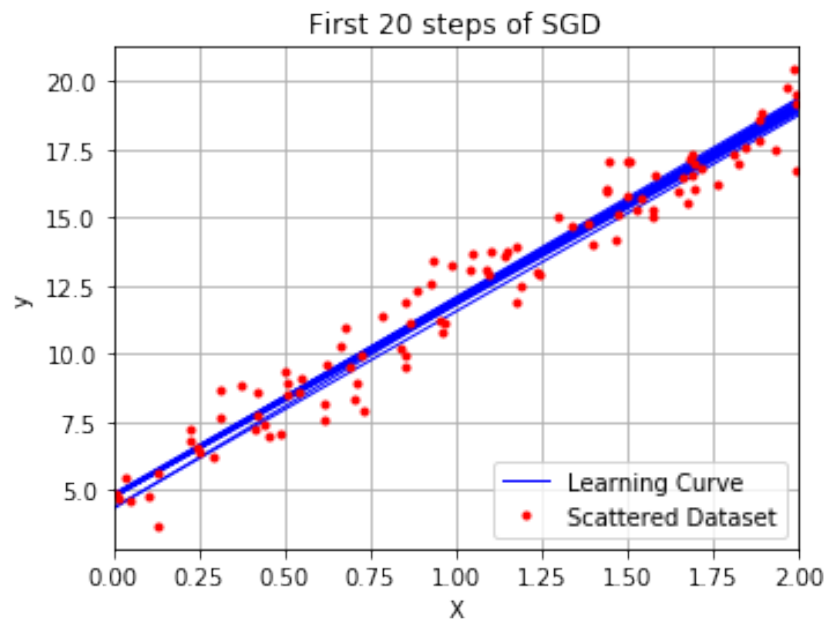
fig = plt.figure()
ax = fig.add_subplot(111)

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 :

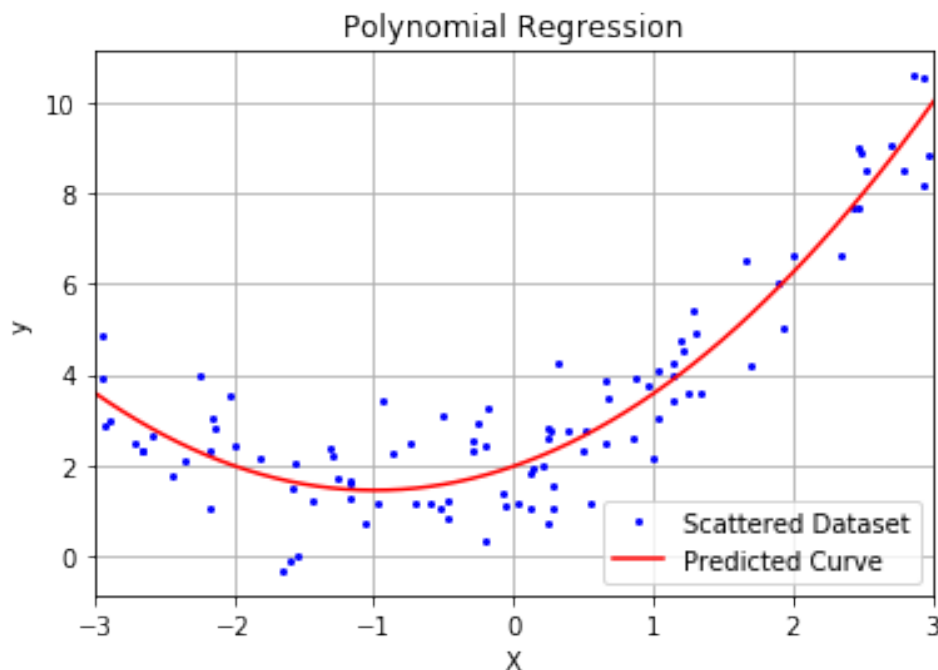
```
[13]: from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

```
[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()

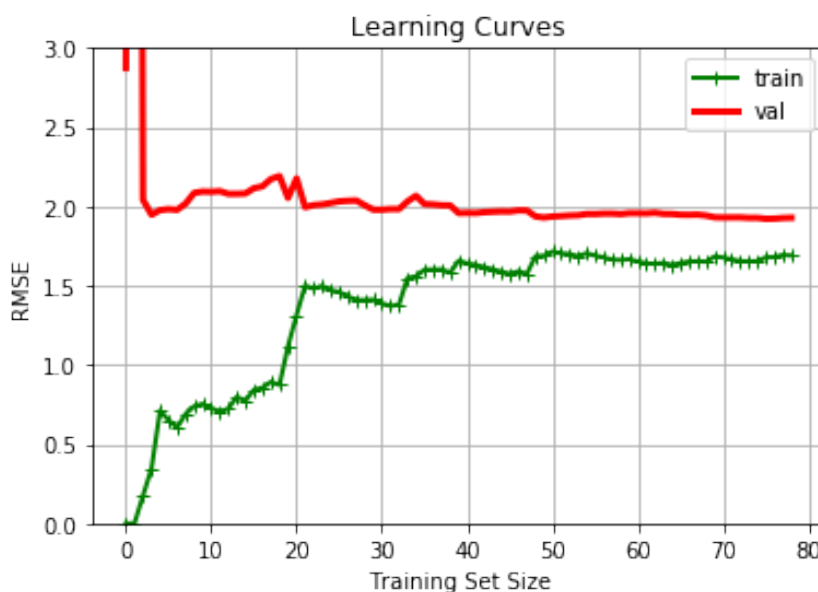
```

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 :

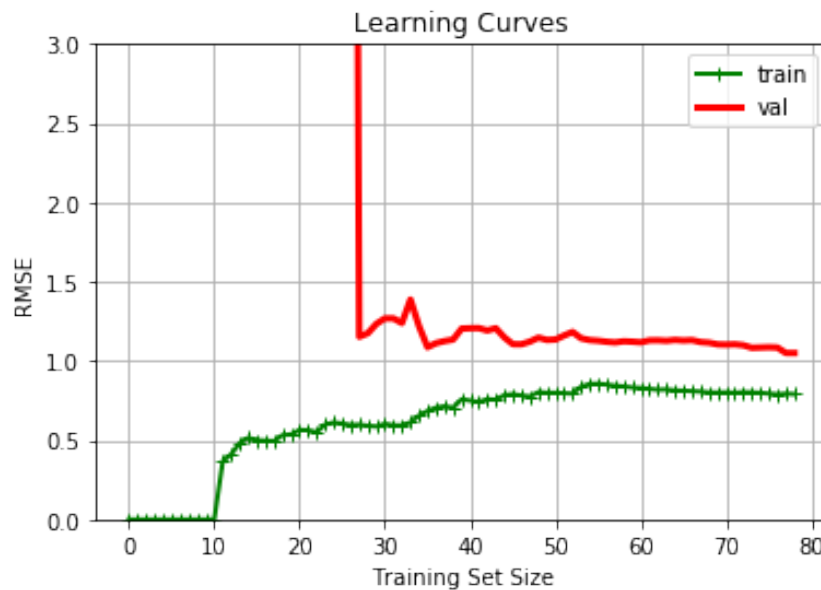
```

[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 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 :

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

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 :

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

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

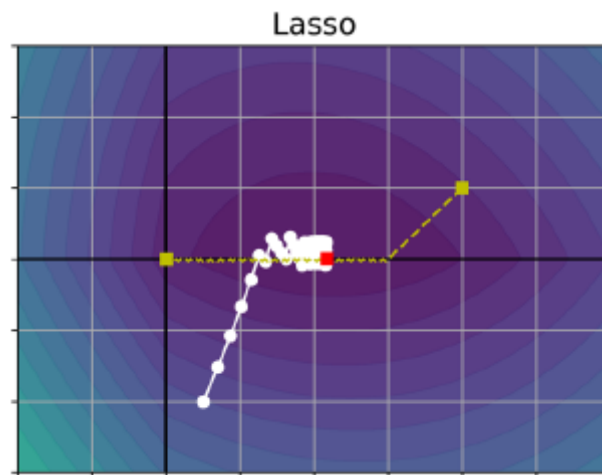
Least Absolute Shrinkage and Selection Operator Regression (usually simply called Lasso Regression) is another regularized version of Linear Regression. But it uses the norm of the weight vector instead of half the square of the norm as the regularization term.

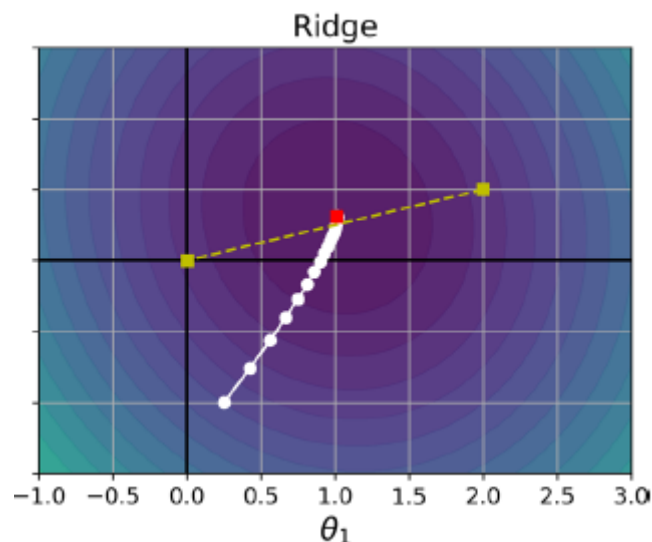
Lasso Regression cost function :

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

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 :

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

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** :

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

σ is the *sigmoid function*. It takes the $\mathbf{x}^T \theta$ 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 $\mathbf{x}^T \theta$ 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 :

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

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

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

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

Well, here we don'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 :

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

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 :

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

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**.

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

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.

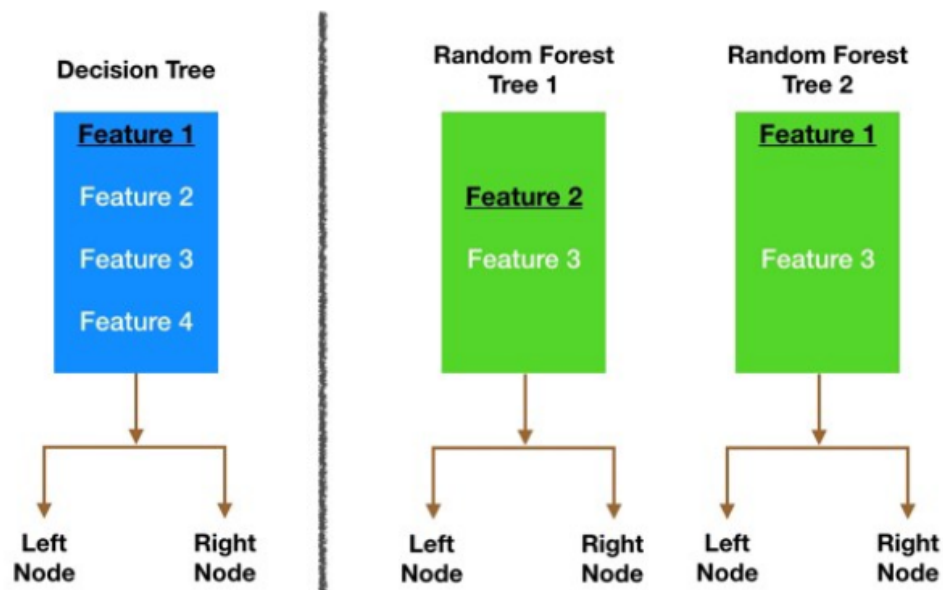
So how to ensure that the models are diversified ?

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

A Review On Visualizing and Understanding Convolutional Networks

3.1 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 visualization technique that gives insight into the functioning of intermediate feature layers and the operation of the classifier.

3.2 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.

3.2.1 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 responses across feature maps.(optional)

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

3.2.2 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**.

3.3 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.

3.3.1 Visualization with a DeConvolutional Network(deconvnet)

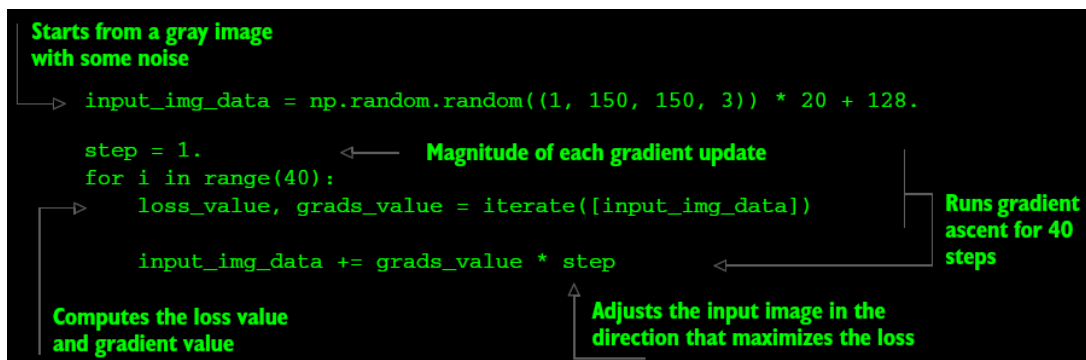
A deconvnet can be thought of as a convnet model that uses the same components but in reverse. To visualize a convnet, a *deconvnet* is attached to each of its layers, providing a perfect path back to its **input pixel space**.

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.



```

Starts from a gray image
with some noise
input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.

step = 1.
for i in range(40):
    loss_value, grads_value = iterate([input_img_data])
    input_img_data += grads_value * step

```

Annotations in the image:

- Magnitude of each gradient update**: points to `step = 1.`
- Runs gradient ascent for 40 steps**: points to the `for i in range(40):` loop.
- Computes the loss value and gradient value**: points to `loss_value, grads_value = iterate([input_img_data])`.
- Adjusts the input image in the direction that maximizes the loss**: points to `input_img_data += grads_value * step`.

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.

3.3.2 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.

3.4 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.

3.5 References

- [The Original Paper](#)
- Deep Learning with Python by **François Chollet**