# Extracted Text from PDF

## Page 1

UNIT IT INTRODUCTION TO DEEP LEARNING  
  
   
  
Deep Feed Forward Neural Networks — Gradient Descent — Back Propagation and other Different{ation  
Algorithms- Vanishing Gradient Problem- Mitigation — Rectified Linear Unit(ReLU)- Heuristics for Avgiding  
Bad Local Minima — Heuristics for Faster Training — Nestors Accelerated Gradient Descent — Regulariqation  
for Deep Learning — Dropout — Adversial Training — Optimization for Training Deep models.  
  
   
  
History of Deep Learning [DL]:  
  
Q) The chain rule that underlies the back-propagation algorithm was invented in the  
seventeenth century (Leibniz, 1676; L’H6pital, 1696)  
  
Q) Beginning in the 1940s, the function approximation techniques were used to motivate  
machine learning models such as the perceptron  
  
The earliest models were based on linear models. Critics including Marvin Minsky  
pointed out several of the flaws of the linear model family, such as its inability to learn  
the XOR function, which led to a backlash against the entire neural network approach  
Efficient applications of the chain rule based on dynamic programming began to appear  
in the 1960s and 1970s  
  
Werbos (1981) proposed applying chain rule techniques for training artificial neural  
networks. The idea was finally developed in practice after being independently  
rediscovered in different ways (LeCun, 1985; Parker, 1985; Rumelhart et al., 1986a)  
Following the success of back-propagation, neural network research gained popularity  
and reached a peak in the early 1990s. Afterwards, other machine learning techniques  
became more popular until the modern deep learning renaissance that began in 2006  
The core ideas behind modern feedforward networks have not changed substantially  
since the 1980s. The same back-propagation algorithm and the same approaches to  
gradient descent are still in use.  
  
Most of the improvement in neural network performance from 1986 to 2015 can be  
attributed to two factors. First, larger datasets have reduced the degree to which statistical  
generalization is a challenge for neural networks. Second, neural networks have become much  
larger, because of more powerful computers and better software infrastructure. A small  
number of algorithmic changes have also improved the performance of neural networks  
noticeably. One of these algorithmic changes was the replacement of mean squared error with  
the cross-entropy family of loss functions. Mean squared error was popular in the 1980s and  
1990s but was gradually replaced by cross-entropy losses and the principle of maximum  
likelihood as ideas spread between the statistics community and the machine learning  
community.  
  
   
  
The other major algorithmic change that has greatly improved the performance of  
feedforward networks was the replacement of sigmoid hidden units with piecewise linear  
hidden units, such as rectified linear units. Rectification using the max{0, z} function was  
introduced in early neural network models and dates back at least as far as the Cognitron and  
Neo-Cognitron (Fukushima, 1975, 1980).  
  
For small datasets, Jarrett et al. (2009) observed that using rectifying nonlinearities is  
even more important than learning the weights of the hidden layers. Random weights are  
  
sufficient to propagate useful information through a rectified linear network, enabling the  
classifier layer at the top to learn how to map different feature vectors to class identities. When  
more data is available, learning begins to extract enough useful knowledge to exceed the  
  
1

## Page 2

performance of randomly chosen parameters. Glorot et al. (2011a) showed that learning is far  
easier in deep rectified linear networks than in deep networks that have curvature or two-sided  
saturation in their activation functions.  
  
When the modern resurgence of deep learning began in 2006, feedforward networks  
continued to have a bad reputation. From about 2006 to 2012, it was widely believed that  
feedforward networks would not perform well unless they were assisted by other models, such  
as probabilistic models. Today, it is now known that with the right resources and engineering  
practices, feedforward networks perform very well. Today, gradient-based learning in  
feedforward networks is used as a tool to develop probabilistic models. Feedforward networks  
continue to have unfulfilled potential. In the future, we expect they will be applied to many  
more tasks, and that advances in optimization algorithms and model design will improve their  
performance even further.  
  
Deep Neural Network  
  
retraining)  
Multi-layered Ea { 2  
  
XOR Perceptron  
ADALINE (Backpropagation)  
  
Perceptron  
Golden Age Dark Age (“Al Winter”)  
Electronic Brain  
  
1960 1970 1980 1990 rut) rut)  
  
A DGS ALP a 2P  
  
S.McCulloch-W. Pitts F.Rosenblatt —B., Widrow ~ M. Hoff M, Minsky - S, Papert D, Rumelhart-G,Hinton-R. Wiliams \_\_V. Vapnik ~C. Cortes G Hinton = 5 Ruslan  
  
   
  
eKo @o] | >  
  
pe  
  
e,@ 0}  
  
   
  
   
  
   
  
   
  
— Bectnad Err  
  
   
  
   
  
   
  
\* Adjustable Weights + Leamable Weights and Threshold + XOR Problem + Solution ononlinearty separable problems + Limitations of arin rir knowiedge + Hirarhica fate Learing  
+ Weights are not Leamed + Big computation, loca optima ard overting + Kemelfuncbon: Human Intervention  
  
2.1 A Probabilistic Theory of Deep Learning  
  
Probability is the science of quantifying uncertain things. Most of machine learning and deep  
learning systems utilize a lot of data to learn about patterns in the data. Whenever data is utilized  
in a system rather than sole logic, uncertainty grows up and whenever uncertainty grows up,  
probability becomes relevant.  
  
By introducing probability to a deep learning system, we introduce common sense to the  
system. In deep learning, several models like Bayesian models, probabilistic graphical models,  
Hidden Markov models are \_ used. They depend entirely on probability concepts.  
  
Real world data is chaotic. Since deep learning systems utilize real world data, they require a  
tool to handle the chaotic Ness.

## Page 3

Deep Feedforward Networks  
  
\* Deep feedforward networks, also often called feedforward neural networks, or  
multilayer perceptrons (MLPs).  
  
\* The goal of a feedforward network 1s to approximate some function / \*.  
  
¢ A feedforward network defines a mapping yv = f (x; 8) and learns the value of the  
parameters @ that result in the best function approximation.  
  
\* These models are called feedforward because information flows through the  
function being evaluated from x, through the intermediate computations used to  
define f, and finally to the output y.  
  
\* There are no feedback connections in which outputs of the model are fed back intq  
itself. When feedforward neural networks are extended to include feedback  
connections, they are called recurrent neural Networks.  
  
¢ Feedforward neural networks are called networks because they are  
typically represented by composing together many different functions.  
  
¢ The model is associated with a directed acyclic graph describing how the  
functions are composed together.  
  
¢ For example, we might have three functions f(1) ,f(2), and f(3) connected  
in a chain, to form f(x) = f(3)(f (2) C)(x))).  
  
¢ These chain structures are the most commonly used structures of neural  
networks.  
  
¢ In this case, f(1) is called the first layer of the network, f(2) is called the  
second layer, and so on.  
  
¢ The overall length of the chain gives the depth of the model.  
  
¢ The final layer of a feedforward network is called the output layer. During  
neural network training, we drive f(x) to match f(x).  
  
¢ The training data provides us with noisy, approximate examples of f \*(x)  
evaluated at different training points.

## Page 4

\* Each example x is accompanied by a label y = f (x).  
  
\* The training examples specify directly what the output layer must do at each point x; it must  
produce a value that is close to y.  
  
\* The behavior of the other layers is not directly specified by the training data.  
  
\* The learning algorithm must decide how to use those layers to produce the desired output, but  
the training data does not say what each individual layer should do.  
  
\* Instead, the learning algorithm must decide how to use these layers to best implement an  
approximation of. tS  
  
\* Because the training data does not show the desired output for each of these layers, these layer}  
are called hidden layers.  
  
\* Finally, these networks are called nevral because they are loosely inspired by neuroscience.  
\* Each hidden layer of the network is typically vector-valued.  
\* The dimensionality of these hidden layers determines the width of the model.  
  
\* Each unit resembles a neuron in the sense that it receives input from many other units and  
computes its own activation value.  
  
\* The idea of using many layers of vector-valued representation is drawn from neuroscience.  
  
\* The choice of the functions f(i)(x) used to compute these representations is also loosely guided  
by neuroscientific observations about the functions that biological neurons compute.  
  
\* One way to understand feedforward networks 1s to begin with linear models and consider how to  
overcome their limitations.  
  
\* Linear models, such as logistic regression and linear regression, are appealing because they may be fit  
efficiently and reliably, either in closed form or with convex optimization.  
  
\* Linear models also have the obvious defect that the model capacity is limited to linear functions, so the  
model cannot understand the interaction between any two input variables.  
  
\* To extend linear models to represent nonlinear functions of x, we can apply the linear model not to x itself  
but to a transformed input g(x), where g is a nonlinear transformation.  
  
\* The question 1s then how to choose the mapping 9.  
  
1. One option is to use a very generic g, such as the infinite-dimensional g that 1s implicitly used by kemel  
machines based on the RBF kernel.  
  
2. Another option is to manually engineer @.  
  
3. The strategy of deep learning is to learn g. In this approach, we have a model y = f(x; 1”) = g(x; 6)" w.  
We now have parameters @ that we use to learn g from a broad class of functions, and parameters w that  
map from g(x) to the desired output. This 1s an example of a deep feedforward network, with g defining a  
hidden layer. This approach is the only one of the three that gives up on the convexity of the traming  
problem, but the benefits outweigh the harms. In this approach, we parametrize the representation as 9(x; 4)  
and use the optimization algorithm to find the @ that corresponds to a good representation.  
  
4

## Page 5

Example: Learning XOR  
  
¢ The XOR function (“exclusive or”) is an operation on two binary values, x1 and x2.  
¢ When exactly one of these binary values is equal to 1, the XOR function returns 1.  
  
\* Otherwise, it returns 0. The XOR function provides the target function y = f(x) that we  
want to learn.  
  
Evaluated on our whole training set, the MSE loss function is  
  
J(@) - S= (s\* (@) — f(x: 0) . (6.1)  
  
rex  
  
Now we must choose the form of our model, f(x;@). Suppose that we choose  
a linear model, with @ consisting of w and b. Our model is defined tp be  
  
f(a: w,b) = a2! w+ b. (6.2)  
  
We can minimize J(@) in closed form with respect to w and 6 using the normal  
equations.  
  
¢ After solving the normal equations, we obtain w = 0 and b = .  
  
¢ . The linear model simply outputs 0.5 everywhere.

## Page 6

Original a2 space Learned h space  
  
1  
Ay  
  
Figure 6.1: Solving the XOR problem by learning a representation. The bold numbers  
printed on the plot indicate the value that the learned function must output at each point.  
(Left)A linear model applied directly to the original input cannot implement the XOR  
function. When «2, = 0, the model’s output must increase as 22 increases. When x7, = 1,  
the model’s output must decrease as r2 increases. A linear model must apply a fixed  
coefficient wy to rg. The linear model therefore cannot use the value of +; to change  
the coefficient on x2 and cannot solve this problem. (Rightj/In the transformed space  
represented by the features extracted by a neural network, a linear model can now solve  
the problem. In our example solution, the two points that must have output 1 have been  
collapsed into a single point in feature space. In other words, the nonlinear features have  
mapped both w = [1,0] 7 and 2 = [0,1]' to a single point in feature space, h = [1,0] °.  
The linear model can now describe the function as increasing in /, and decreasing in hy.  
In this example, the motivation for learning the feature space is only to make the model  
capacity greater so that it can fit the training set. In more realistic applications, learned  
representations can also help the model to generalize.  
  
Bi  
  
Figure 6.2: An example of a feedforward network, drawn in two different styles. Specifically,  
this is the feedforward network we use to solve the XOR example. It has a single hidden  
layer containing two units. (Left)In this style, we draw every unit as a node in the graph,  
This style is very explicit and unambiguous but for networks larger than this example  
it can consume too much space. (Right)In this style, we draw a node in the graph for  
each entire vector representing a layer’s activations. This style is much more compact,  
Sometimes we annotate the edges in this graph with the name of the parameters that  
describe the relationship between two layers. Here, we indicate that a matrix W describes  
the mapping from a2 to h, and a vector w describes the mapping from h to y. We  
typically omit the intercept parameters associated with each layer when labeling this kind  
of drawing.

## Page 7

2}  
  
max{0,  
  
)=  
  
9(2)  
  
Figure 6.3: The rectified linear activation function, This activation function is the default  
activation function recommended for use with most feedforward neural networks. Applying  
this function to the output of a linear transformation yields a nonlinear transformation.  
However, the function remains very close to linear, in the sense that is a piecewise linear  
function with two linear pieces. Because rectified linear units are nearly linear, they  
preserve many of the properties that make linear models easy to optimize with gradient-  
based methods. They also preserve many of the properties that make linear models  
generalize well. A common principle throughout computer science is that we can build  
complicated systems from minimal components. Much as a Turing machine’s memory  
needs only to be able to store 0 or 1 states, we can build a universal function approximator  
from rectified linear functions.  
  
Gradient Descent  
  
1. Definition:  
  
¢ Anoptimization algorithm used to minimize a cost (loss) function by iteratively updating  
  
model parameters.  
2. Steps:  
¢ Start with initial parameters (wo, bo).  
  
¢ Update parameters using the gradient of the cost function:  
  
OJ OJ  
Te,’ best = by - "9b,"  
  
where 1 is the learning rate, and J is the cost function.  
  
Wii = We —

## Page 8

3. Types:  
  
e Batch Gradient Descent:  
  
e Uses the entire dataset for each update.  
  
e Slow for large datasets.  
  
e Stochastic Gradient Descent (SGD):  
  
e Updates parameters after each training sample.  
e Faster but introduces noise.  
  
e Mini-Batch Gradient Descent:  
  
e Acompromise between batch and stochastic approaches.

## Page 9

2.2 Back Propagation Networks (BPN)  
2.2.1. Need for Multilayer Networks  
  
e Single Layer networks cannot used to solve Linear Inseparable problems &  
can only be used to solve linear separable problems  
Single layer networks cannot solve complex problems  
Single layer networks cannot be used when large input-output data set is  
available  
Single layer networks cannot capture the complex information’s available in  
the training pairs  
  
Hence to overcome the above said Limitations we use Multi-Layer Networks.  
  
2.2.2. Multi-Layer Networks  
  
e Any neural network which has at least one layer in between input and output  
layers is called Multi-Layer Networks  
  
e Layers present in between the input and out layers are called Hidden Layers  
  
eInput layer neural unit just collects the inputs and forwards them to the next  
higher layer  
  
e Hidden layer and output layer neural units process the information’s feed to  
them and produce an appropriate output  
  
e Multi -layer networks provide optimal solution for arbitrary classification  
problems  
  
e Multi -layer networks use linear discriminants, where the inputs are non  
linear  
  
2.2.3. Back Propagation Networks (BPN)  
  
Introduced by Rumelhart, Hinton, & Williams in 1986. BPN is a Multi-  
layer Feedforward Network but error is back propagated, Hence the name Back  
Propagation Network (BPN). It uses Supervised Training process; it has a  
systematic procedure for training the network and is used in Error Detection and  
Correction. Generalized Delta Law /Continuous Perceptron Law/ Gradient Descent  
Law is used in this network. Generalized Delta rule minimizes the mean squared  
error of the output calculated from the output. Delta law has faster convergence rate  
when compared with Perceptron Law. It is the extended version of Perceptron  
Training Law. Limitations of this law is the Local minima problem. Due to this the  
convergence speed reduces, but it is better than perceptron’s. Figure 1 represents a  
BPN network architecture. Even though Multi level perceptron’s can be used they  
are flexible and efficient that BPN. In figure 1 the weights between input and the  
hidden portion is considered as Wij and the weight between first hidden to the next  
layer is considered as Vjx. This network is valid only for Differential Output  
functions. The Training process used in backpropagation involves three stages,  
which are listed as below  
  
1. Feedforward of input training pair

## Page 10

2. Calculation and backpropagation of associated error  
  
3. Adjustments of weights  
  
   
  
Wi Vik  
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
x C) vii C) Output layer  
¥  
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
a a  
OSGO  
a =  
«a  
OHO  
  
Input layer Hidden layer 1 Hidden layer 2  
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
   
  
Figure 1: Back Propagation Network  
2.2.4. BPN Algorithm  
The algorithm for BPN is as classified int four major steps as follows:  
Initialization of Bias, Weights  
  
Feedforward process  
  
1  
2  
3. Back Propagation of Errors  
4  
  
Updating of weights & biases  
Algorithm:  
  
I. Initialization of weights:  
Step 1: Initialize the weights to small random values near zero  
Step 2: While stop condition is false , Do steps 3 to 10  
Step 3: For each training pair do steps 4 to 9  
II. Feed forward of inputs  
Step 4: Each input xi is received and forwarded to higher layers (next  
hidden)  
Step 5: Hidden unit sums its weighted inputs as follows  
Zinj = Woj + XUxiwij  
Applying Activation function  
Zj = f(Zinj)  
This value is passed to the output layer  
Step 6: Output unit sums it’s weighted inputs  
Yink= Voj + X ZjVjx  
Applying Activation function  
  
10

## Page 11

Yk = f(yink)  
Ill. Backpropagation of Errors  
  
Step 7: dk = (tk —  
Y)f(yink ) Step 8:  
Sinj = X OjVix  
IV. Updating of Weights & Biases  
Step 8: Weight correction is  
A  
Wij = 06xZ; bias Correction  
is  
A  
Woj = 06k  
V. Updating of Weights & Biases  
Step 9: continued:  
New Weight is  
Wij(new) = Wijold) +  
Awij Vik(new) =  
Viola + AVix  
New bias is  
Woj(new) = Woj(old)  
+ AWoj Vok(new) =  
Vok(old) + AVok  
  
Step 10: Test for Stop Condition  
  
2.2.5 Merits  
\* Has smooth effect on weight correction  
\* Computing time is less if weight’s are small  
\* 100 times faster than perceptron model  
\* Has a systematic weight updating procedure  
  
2.2.6. Demerits  
Learning phase requires intensive calculations  
Selection of number of Hidden layer neurons is an issue  
Selection of number of Hidden layers is also an issue  
Network gets trapped in Local Minima  
Temporal Instability  
Network Paralysis  
Training time is more for Complex problems

## Page 12

Assume that the neurons have a sigmoid activation function,  
  
perform a forward pass and a backward pass on the network.  
  
Assume that the actual output of y is 0.5 and learning rate is 1.  
  
Perform another forward pass.  
  
W,,=0. 6  
Forward Pass: Compute output for y3, y4 and y5.  
  
a= Oms\*x) yi = Fai) =  
  
J  
al= (wi3 \*x1) + (w23 \*x2)  
  
= (0.1 \* 0.35) + (0.8 \* 0.9) = 0.755  
  
— — 1 =  
y3=Hla-ay ppp 7 0.688  
  
a2= (wi4 \*x1) + (w24\* x2)  
  
= (0.4 \* 0.35) + (0.6 \* 0.9) = 0.68  
  
= f(ar 2)= aa = 0.6637  
  
a3 = (w3s \*y3) + (was \* ya)

## Page 13

= (0.3 \* 0.68) + (0.9 \* 0.6637) = 0.801  
  
1  
  
y5 = fas) - (1+e-0.801)  
  
= 0.69 Network Output  
  
Error=y farget— ys = 0.5-0.69= 0.19  
  
Aw, =70,0,  
  
6,=0,(l-o,)(¢,-0,) if isan output unit  
6, =0,(1-o, >> OW, if j isa hidden unit  
k  
  
where n is a constant called the learning rate  
  
ty is the correct teacher output for unit j  
  
Pe  
  
Qj is the error measure for unit j  
  
Backward Pass: Compute 63, 54 and 45.  
  
For output unit:  
65 = y(1-y) (Ytarget — Y).  
= 0.69\*(1-0.69)\*(0.5-0.69)= -0.0406  
  
For hidden unit:  
  
53=Y3(1-Y3) W35 \* 55  
= 0.68\*(1 -0.68)\*(0.3\* -0.0406) = -0.00265  
  
54=Y4(1-Y4)W45\* 55  
=0.6637 \* (1-0.6637)\* (0.9 \* -0.0406) =-0.0082

## Page 14

Compute new weights  
  
Aw]j = 1 901  
  
AW45= 1 85 Y4= 1\* -0.0406 \* 0.6637 = -0.0269  
  
W45 (new) = AW45 + W45 (old) = -0.0269 + (0.9) = 0.8731  
  
Aw14=184x1 = 1\* -0.0082 \* 0.35 = -0.00287  
W14 (new) = AW14 + W14 (old) = -0.00287+ 0.4 = 0.3971  
  
¢ Similarly, update all other weights  
  
Updated w,,  
  
Ul  
0.1 -0.00265 0.35 1  
0. -0.00265 0.9 1  
0.4 -0.0082 0.35 1  
1  
1  
1  
  
0.6 -0.0082 0.9  
  
0.3 -0.0406 0.68  
  
0.9 -0.0406 0.6637  
y; = 0.68  
  
n= os @)~w,-0 221”  
: x . >  
W,, =0.3971 40  
W,; =0.7976“ a  
ya — H, )W,;=0.8731 Ontpaty  
W,,=0.5926 \\_\*/  
y, = 0.6637  
  
Again need to apply the feed forward

## Page 15

hain rule of differentiation in backpropagation  
  
C  
Neural Network  
  
Hidden layer:  
  
H11=0.68, O(h11) =0.66 O1(target output)=0 ,y1=0.1636, O(y1) =0.54  
H12=0.76, O(h12)=0.68 O2(target output) =1, y2=0.324, O(y2)=0.58  
H21=1.306, O(h21) =0.786  
  
H22=1.504, O(h22) =0.818  
Whew = Wold- learning rate \* slope  
Whew = Wold- \* 0L(w)/ ow  
  
- The unknown is derivative ie, slope  
  
Loss=0.234, Learning rate =0.1  
  
- OL/ 6wi=0L/ 6 O(y2). 6 O(y2)/ dy2. dy2/éwr  
  
=-0.42 \* 0.58\*0.42\*0.818 = -0.0836  
  
OL / 6 O(y2) = A/ 6 O(y2) ./21( O(y1)-01)\*+( O(y2)-02)"I  
-- Value of L  
  
= 2.1/2 ( O (y2)-O2) -- differentiate w.r.t y2  
  
= O(y2)-Oo  
= 0.58-1 =-0.42

## Page 16

6 O(y2)/ dy2= 6 O(x)/ Ox =d/ Ox(1/1+e\*)  
= [ formula f’(x) = f(x)(1-f(x))]  
= (1/1+e\*) (1-1/1+e\*)  
= (x). (1- O@))  
Therefore, 0 O(y2)/ dy2 = O(y2). (1- O(y2))  
= 0.58 (1-0.58)  
=0.58\*0.42  
Oy2/Owi2= 6/Owi2(w12\* O(h22)+w10 O(h21)+b2)  
  
= O(h22) =0.818 (it is partial derivative so other are nullify)  
  
Win =Wl2- aL / éwi2  
= 0.3 -0.1\*-0.0836 => 0.30836  
  
The vanishing gradient problem  
  
It is a challenge faced in training artificial neural networks, particularly  
deep feedforward and recurrent neural networks. This issue arises  
during the backpropagation process, which is used to update the  
weights of the neural network through gradient descent. The gradients  
are calculated using the chain rule and propagated back through the  
network, starting from the output layer and moving towards the input  
layer. However, when the gradients are very small, they can diminish  
as they are propagated back through the network, leading to minimal  
or no updates to the weights in the initial layers. This phenomenon is  
known as the vanishing gradient problem.

## Page 17

ACG OSA) oe  
  
PX  
Oy)  
  
ae  
  
WR)  
  
SASS  
aw XO RS “sh  
fae EN  
wr on Sa Sed OS oi a  
PM DN  
CRI PORK  
4 Es UAE YOR  
ne K BO HS VY LX ry  
a es oN BE  
  
TSX] a  
Ee  
  
iy,  
mid  
  
Certain activation functions, like the sigmoid function, squishes a large input space into a  
small input space between 0 and 1. Therefore, a large change in the input of the sigmoid  
  
function will cause a small change in the output. Hence, the derivative becomes small.  
  
As an example, Image 1 is the sigmoid function and its derivative. Note how  
when the inputs of the sigmoid function becomes larger or smaller (when |x|  
  
becomes bigger), the derivative becomes close to zero.

## Page 18

3. However, when n hidden layers use an activation like the sigmoid  
  
function n small derivatives are multiplied together. Thus, the gradient  
decreases exponentially as we propagate down to the initial layers.  
  
. Asmall gradient means that the weights and biases of the initial layers will  
not be updates effectively with each training session, since the initial layers  
are often crucial to recognizing the core elements of the input data, it can  
lead to overall inaccuracy of the whole network.  
  
Why it’s Significant:  
  
1. For shallow network with only a few layers that use these activations, this  
isn’t a big problem. However, when more layers are used, it can cause the  
gradient to be too small for training to work effectively.  
  
. Gradients of neural network are found using backpropagation, simply put  
backpropagation finds the derivatives of the network by moving layer from  
the final layer to the initial one. By the chain rule the derivatives of each  
layer are multiplied down the network (from the final layer to the initial)  
  
to compute the derivatives of the initial layers.

## Page 19

To mitigate the vanishing gradient problem, several strategies  
have been developed:  
  
Long Short-Term Memory(LSTM)  
  
@ Soas of now, we have seen there are two major factors that affect the gradient size - weights  
and their derivatives of the activation function. A simple LSTM helps the gradient size to  
remain constant, The activation function we use in the LSTM often works as an identity  
function which is a derivative of 1. So in gradient backpropagation, the size of the gradient  
does not vanish,  
  
Residual Neural Network  
  
@ The skip or bypass connection inr residual network is useful in any network to bypass the  
data from a few layers, Basically, it allows information to skip the layers. Using these  
connections, information can be transferred from layer n to layer n+t. Here to perform this  
thing we need to connect the activation function of layer n to the activation function of  
n+t, This causes the gradient to pass between the layers without any modification in size,  
  
ReLu Activation Function  
  
e Ifthe ReLU function is used for activation in a neural network in place of a sigmoid  
function, the value of the partial derivative of the loss function will be having values  
of 0 or 1 which prevents the gradient from vanishing. The use of ReLU function thus  
prevents the gradient from vanishing  
  
Activation Functions: Using activation functions such as Rectified Linear  
Unit (ReLU) and its variants (Leaky ReLU, Parametric ReLU, etc.) can help prevent  
the vanishing gradient problem. ReLU and its variants have a constant gradient for  
positive input values, which ensures that the gradients do not diminish too quickly  
during backpropagation.  
  
Weight Initialization: Properly initializing the weights can help prevent gradients  
from vanishing.

## Page 20

Activation Function  
  
An activation function in a neural network defines how the weighted sum of the input is  
transformed into an output from a node or nodes in a layer of the network.  
  
Sometimes the activation function is called a “transfer function.” If the output range of the  
activation function is limited, then it may be called a “squashing function.” Many activation  
functions are nonlinear and may be referred to\*as the “nonlinearity” in the layer or the network  
design.  
  
The choice of activation function has a large impact on the capability and performance of the  
neural network, and different activation functions may be used in different parts of the model.  
Technically, the activation function is used within or after the internal processing of each node in  
the network, although networks are designed to use the same activation function for all nodes in  
a layer.  
  
Anetwork may have three types of layers: input layers that take raw input from the domain,  
hidden layers that take input from another layer and pass output to another layer, and output  
layers that make a prediction.  
  
All hidden layers typically use the same activation function. The output layer will typically use a  
different activation function from the hidden layers and is dependent upon the type of prediction  
required by the model.  
  
Activation functions are also typically differentiable, meaning the first-order derivative can be  
calculated for a given input value. This is required given that neural networks are typically trained  
using the backpropagation of error algorithm that requires the derivative of prediction error in  
order to update the weights of the model.  
  
Activation Function Choices for Hidden Layers  
  
=~

## Page 21

BINARY STEP FUNCTION  
  
Pex = 34 2  
1  
  
=p &>o  
  
thrrotol > O  
  
. oO X<O 06  
Fem = a 9620.6  
  
0.0 2.5  
  
Binary Step Activation Function  
  
e Binary step function depends on a threshold value that decides whether a neuron should  
be activated or not.  
  
e The input fed to the activation function is compared to a certain threshold; if the input is  
greater than it, then the neuron is activated, else it is deactivated, meaning that its output  
  
is not passed on to the next hidden layer.  
Here are some of the limitations of binary step function:  
  
e It cannot provide multi-value outputs—for example, it cannot be used for multi-class  
classification problems.  
The gradient of the step function is zero, which causes a hindrance in the backpropagation  
process.

## Page 22

LINEAR ACTIVATION FUNCTION  
  
   
  
4 2 o 2  
x  
  
Linear Activation Function  
  
e The linear activation function, also known as "no activation," or "identity function"  
(multiplied x1.0), is where the activation is proportional to the input.  
e The function doesn't do anything to the weighted sum of the input, it simply spits out  
  
the value it was given.  
However, a linear activation function has two major problems :  
  
e It's not possible to use backpropagation as the derivative of the function is a constant  
and has no relation to the input x.  
  
e All layers of the neural network will collapse into one if a linear activation function is  
used. No matter the number of layers in the neural network, the last layer will still be  
a linear function of the first layer. So, essentially, a linear activation function turns the  
neural network into just one layer.

## Page 23

NON-LINEAR ACTIVATION FUNCTION  
  
Because of its limited power, linear does not allow the model to create complex mappings  
  
between the network's inputs and outputs.  
Non-linear activation functions solve the following limitations of linear activation functions:  
  
e They allow backpropagation because now the derivative function would be related to the  
input, and it's possible to go back and understand which weights in the input neurons can  
provide a better prediction.  
  
They allow the stacking of multiple layers of neurons as the output would now be a  
non-linear combination of input passed through multiple layers. Any output can be  
represented as a functional computation in a neural network.  
  
Sigmoid Activation Function  
  
   
  
— sigmoid  
— gradient  
  
1  
  
I=  
  
Following chain rule to find derivative  
  
   
  
   
  
   
  
fx) = “ a fa)=—| fase ofa) =e  
  
x -100 -75 -50 -25 00 25 50 75  
x  
  
   
  
   
  
   
  
   
  
e\*  
  
= ——\_\_ = —\_\*  
(+e tey “  
  
“x \_\_ | = -  
f(a) = (-e") Tae) =fO)\*0-F00)  
  
-| x  
(+e)

## Page 24

It is commonly used for models where we have to predict the probability as an  
output. Since probability of anything exists only between the range of 0 and 1,  
sigmoid is the right choice because of its range.  
  
The function is differentiable and provides a smooth gradient, i.e., preventing jumps  
in output values. This is represented by an S-shape of the sigmoid activation  
function.  
  
Drawbacks: \*  
  
1,  
  
As the gradient value approaches zero, the network ceases to learn and suffers  
  
from the Vanishing gradient problem.  
  
The output of the logistic function is not symmetric around zero. So the output of all  
the neurons will be of the same sign. This makes the training of the neural network  
more difficult and unstable.  
  
2. tan-h  
  
tanh function  
  
Tanh function is very similar to the sigmoid/logistic activation function, and even has  
the same S-shape with the difference in output range of -1 to 1.  
  
Advantages of using this activation function are:  
  
e The output of the tanh activation function is Zero centered; hence we can easily  
map the output values as strongly negative, neutral, or strongly positive.  
Usually used in hidden layers of a neural network as its values lie between -1 to  
1; therefore, the mean for the hidden layer comes out to be 0 or very close to it.  
It helps in centering the data and makes learning for the next layer much easier.  
  
Drawback:  
  
it also faces the problem of vanishing gradients similar to the sigmoid activation  
function.

## Page 25

ReLU — Rectified Linear Unit  
  
ReLU, short for rectified linear unit, is a non-linear activation function used  
for deep neural networks in machine learning. It is also known as the rectifier  
activation function. It helps in mitigate the vanishing gradient problem during  
machine learning model training and enabling neural networks to learn more  
complex relationships in data.  
  
If a model input is positive, the ReLU function outputs the same value. If a  
model input is negative, the ReLU function outputs zero.  
  
e ReLu helps models to learn faster and it's performance is better  
  
Nonlinearity isa statistical term that describes a relationship between  
variables that is not linear, or cannot be expressed with a straight line  
  
Rectified Linear Unit (ReLU)  
Activation Function  
  
   
  
— relu  
gradient  
  
f(x) = max {0, x}  
  
   
  
   
  
   
  
f'(x)=1, x>=0  
=0, x<0O  
  
e ReLU stands for Rectified Linear Unit. »  
  
¢ Although it gives an impression of a linear function, ReLU has a derivative function and allows for  
backpropagation while simultaneously making it computationally efficient.  
The main catch here is that the ReLU function does not activate all the neurons at the same time.  
The neurons will only be deactivated if the output of the linear transformation is less than 0.  
The advantages of using ReLU as an activation function are as follows:  
  
b  
Since only a certain number of neurons are activated, the ReLU function is far more  
  
computationally efficient when compared to the sigmoid and tanh functions.  
ReLU accelerates the convergence of gradient descent towards the global minimum of the loss  
function due to its linear, non-saturating property.

## Page 26

Disadvantages  
1, Itis not zero-centred, making training slightly unstable and requiring more iterations to train on for better  
performance.  
  
. It completely removes the negative values from the calculation making those neurons inactive. Suppose a cas4  
where the weight multiplication and bias addition will always output a negative value irrespective of the input  
  
values. Here, the entire network architecture will fall into a dead state, known as dead relu.  
  
. ReLU is unbounded, which helps solve the vanishing gradient problem, but it becomes prone to exploding  
  
gradient issues if weight or bias values have a higher magnitude.  
Heuristics for Avoiding Bad Local Minima  
  
Avoiding bad local minima in deep learning is especially important because  
deep neural networks often involve highly non-convex loss landscapes. Here are  
key heuristics tailored for deep learning:  
  
1. Careful Weight Initialization  
e Proper initialization can help networks converge to better solutions.  
  
o Xavier Initialization: Scales weights based on the number of  
input/output neurons.  
  
He Initialization: Suitable for networks with ReLU activations.  
  
Orthogonal Initialization: Ensures diversity in the initial weights.

## Page 27

2. Optimizers with Momentum  
  
e Use advanced optimization algorithms that incorporate momentum or  
adaptive learning rates:  
  
o SGD with Momentum: Helps accelerate convergence and escape  
flat minima.  
  
Adam: Combines momentum with adaptive learning rates to  
improve optimization stability.  
  
RMSProp: Adjusts the learning rate for each parameter based on  
recent gradients.  
  
3. Learning Rate Scheduling  
  
e Adjust the learning rate dynamically during training:  
o Exponential Decay: Gradually reduces the learning rate over time.  
  
Cosine Annealing: Cyclically reduces and resets the learning rate  
to avoid shallow minima.  
  
Warm Restarts: Temporarily increase the learning rate to escape  
potential traps.  
  
4. Stochastic Gradient Descent (SGD)  
  
¢ The stochastic nature of mini-batch SGD introduces noise into gradient  
updates, which can help escape bad local minima.  
  
5. Batch Normalization  
e Normalize intermediate activations to stabilize the training process.  
  
e It smooths the loss landscape and allows higher learning rates, reducing  
the likelihood of bad minima.  
  
6. Over-Parameterization  
  
e Use larger networks with more parameters. Over-parameterized models  
often lead to smoother loss landscapes with fewer bad local minima.  
  
7. Regularization  
e Encourage generalizable solutions:  
  
o L2 Regularization (Weight Decay): Penalizes large weights,  
smoothing the loss landscape.  
  
27

## Page 28

o Dropout: Randomly drops units during training, preventing  
reliance on specific paths.  
  
8. Skip Connections  
  
¢ Modern architectures like ResNets and DenseNets incorporate skip  
connections, which help smooth loss landscapes and make optimization  
easier.  
  
9. Data Augmentation  
  
- Diversify the training dataset with augmented samples. This can prevent  
the model from overfitting to bad minima.  
  
10. Loss Function Design  
¢ Use loss functions that are less likely to cause optimization traps:  
  
o Label Smoothing: Adds noise to the target labels, preventing  
overconfidence.  
  
Auxiliary Losses: Add additional loss terms (e.g., from  
intermediate layers) to guide optimization.

## Page 29

11. Gradual Model Scaling  
  
¢ Train on simpler tasks or smaller models and progressively increase  
complexity (e.g., curriculum learning or progressive growing of  
networks).  
  
12. Perturbations  
- Inject noise into the training process:  
  
co Add noise to gradients or weights to encourage exploration of the  
loss landscape.  
  
Gradient Noise Injection: Stochastically perturbs gradients during  
updates.  
  
13. Transfer Learning  
  
- Fine-tune pre-trained models instead of training from scratch. Pre-trained  
weights often start closer to good minima.  
  
14. Ensemble Methods  
  
¢ Train multiple models and ensemble their outputs. This mitigates the  
impact of any single model being trapped in a bad minimum.  
  
15. Visualization and Analysis  
- Use tools like:  
co t-SNE or PCA: Visualize embeddings and optimization progress.  
  
co Loss Surface Visualization: Assess how smooth or rugged the loss  
landscape is.  
  
By combining these heuristics, deep learning practitioners can improve the  
robustness of training and avoid getting stuck in bad local minima,  
ultimately achieving better generalization and performance.  
  
Heuristics for Faster Training  
  
Faster training in deep learning can significantly reduce computational costs and|  
speed up model development. Below are heuristics and techniques to achieve  
faster training:  
  
1. Efficient Data Handling  
1. Data Preprocessing:  
o Normalize or standardize input data to improve convergence.  
  
o Use data augmentation during preprocessing to enhance  
generalization without additional training data.  
  
2. Data Pipeline Optimization:  
  
o Use frameworks like TensorFlow Data API or PyTorch  
DataLoader for efficient data loading.  
  
Prefetch, parallelize, and batch data loading to prevent I/O  
bottlenecks.  
  
3. Mixed Precision Training:  
  
o Use lower-precision (e.g., FP 16) computations instead of FP32  
where supported by hardware, like NVIDIA GPUs with Tensor  
Cores.  
  
29

## Page 30

2. Optimizers  
1. Momentum-Based Optimizers:  
  
o Use optimizers like Adam, RMSProp, or SGD with Momentum  
for faster convergence.  
  
2. Learning Rate Scheduling:  
  
o Use learning rate decay methods like:  
  
= Step Decay: Reduce learning rate at specific epochs.  
  
Cosine Annealing: Smoothly adjust learning rates in cycles.  
  
Warm Restarts: Temporarily increase learning rate to  
escape plateaus.  
  
3. Model Architecture Design  
1. Efficient Architectures:  
  
o Use modern, optimized architectures like MobileNet,  
EfficientNet, or ResNet.  
  
o Consider pruning or using lightweight layers (e.g., depthwise  
separable convolutions).  
  
2. Skip Connections:  
  
co Incorporate residual or skip connections (e.g., ResNet) to reduce  
vanishing gradient issues and accelerate training.  
  
4. Training Techniques  
1. Batch Size Optimization:  
  
o Use the largest batch size that fits in memory. Larger batches  
typically lead to faster convergence.  
  
2. Gradient Accumulation:  
  
o If memory is limited, accumulate gradients over smaller batches to  
simulate a larger effective batch size.  
  
3. Gradient Clipping:  
  
o Clip gradients to prevent exploding gradients, enabling faster  
convergence without destabilizing updates.  
  
4. Knowledge Distillation:  
  
o Usea pre-trained "teacher" model to guide the "student" model for  
faster convergence.

## Page 31

5. Regularization and Early Stopping  
1. Dropout and Weight Decay:  
  
o Regularize training to prevent overfitting, reducing the need for  
long training times.  
  
2. Early Stopping:  
  
o Monitor validation loss and stop training once it stops improving,  
avoiding unnecessary epochs.  
  
6. Transfer Learning  
  
e Fine-tune a pre-trained model instead of training from scratch,  
significantly reducing the number of epochs required.  
  
7. Parallelism and Hardware Utilization  
1. GPU/TPU Utilization:  
  
o Use hardware accelerators like GPUs or TPUs for parallel  
computation.  
  
2. Multi-GPU Training:  
  
o Distribute training across multiple GPUs using frameworks like  
PyTorch Distributed Data Parallel or TensorFlow Mirrored  
Strategy.  
  
3. Distributed Training:  
  
o Leverage cluster-based training for large-scale models or datasets.  
  
8. Hyperparameter Tuning  
  
1. Automated Search:  
  
o Use tools like Optuna, Ray Tune, or Hyperopt to find optimal  
hyperparameters quickly.  
  
2. Learning Rate Finder:  
  
o Use methods like the learning rate range test to determine the best  
learning rate.

## Page 32

9. Checkpoints and Resuming  
  
¢ Save and resume training checkpoints to avoid restarting from scratch i  
case of interruptions.  
  
10. Profiling and Debugging  
1. Profiling Tools:  
  
o Use tools like NVIDIA Nsight, TensorFlow Profiler, or PyTorch  
Profiler to identify bottlenecks.  
  
2. Debugging and Visualization:  
  
Use frameworks like TensorBoard to monitor loss, accuracy, and other metrics  
to diagnose inefficiencies  
  
11. Pruning and Compression  
1. Weight Pruning:  
o Remove unimportant weights to reduce model complexity and training time.  
  
2. Quantization:  
  
o Use quantized operations during training or inference for faster computations.  
  
12. Avoid Overtraining  
  
¢ Monitor training metrics and validation performance closely. Avoid training too long  
to minimize unnecessary compute time.  
  
By implementing these heuristics, which can effectively reduce the time required to train  
deep learning models while maintaining or improving their performance.

## Page 33

Nesterov's Accelerated Gradient Descent (NAG):  
  
Gradient descent  
  
It is essential to understand Gradient descent before we look at Nesterov Accelerated  
Algorithm. Gradient descent is an optimization algorithm that is used to train our model.  
The accuracy of a machine learning model is determined by the cost function. The lower the  
cost, the better our machine learning model is performing. Optimization algorithms are used  
to reach the minimum point of our cost function. Gradient descent is the most common  
optimization algorithm. It takes parameters at the start and then changes them iteratively to  
reach the minimum point of our cost function.  
  
Initial  
  
Weight / Gradient  
we /  
a  
  
Incremental  
  
Step NY  
  
a  
<\_< Minimum Cost  
  
Derivative of Cost  
  
we take some initial weight, and according to that, we are positioned at some  
point on our cost function. Now, gradient descent tweaks the weight in each  
iteration, and we move towards the minimum of our cost function accordingly.  
  
   
  
model is very important as it can cause problems while training.  
  
A low learning rate assures us to reach the minimum point, but it takes a lot o  
iterations to train, while a very high learning rate can cause us to cross the  
minimum point, a problem commonly known as overshooting.  
  
The size of our steps depends on the learning rate of our model. The higher the  
learning rate, the higher the step size. Choosing the correct learning rate for or

## Page 34

Too low Just right Too high  
  
Asmall learning rate The optimal learning  
requires many updates rate swiftly reaches the  
before reaching the minimum point  
minimum point  
  
Too large of a learning rate  
causes drastic updates  
which lead to divergent  
  
\* behaviors  
  
Drawbacks of gradient descent  
  
The main drawback of gradient descent is that it depends on the learning rate and the  
gradient of that particular step only. The gradient at the plateau, also known as saddle  
points of our function, will be close to zero. The step size becomes very small or even zero.  
Thus, the update of our parameters is very slow at a gentle slope.  
  
Let us look at an example. The starting point of our model is ‘A’. The loss function will  
decrease rapidly on the path AB because of the higher gradient. But as the gradient  
decreases from B to C, the learning is negligible. The gradient at point ‘C’ is zero, and it is the  
saddle point of our function. Even after many iterations, we will be stuck at ‘C’ and will not  
reach the desired minimum ‘D’.  
  
A  
  
D  
  
This problem is solved by using momentum in our gradient descent.  
  
34

## Page 35

Gradient descent with momentum  
  
The issue discussed above can be solved by including the previous gradients in our  
calculation. The intuition behind this is if we are repeatedly asked to go in a particular  
direction, we can take bigger steps towards that direction.  
  
The weighted average of all the previous gradients is added to our equation, and it acts a4  
momentum to our step.  
  
cost  
Movement =  
  
Negative of Gradient + Momentum  
=> Negative of Gradient  
  
seee> Momentum  
  
=———p> Real Movement  
  
Gradient =0

## Page 36

As we start to descend, the momentum increases, and even at gentle slopes  
where the gradient is minimal, the actual movement is large due to the added  
momentum.  
  
But this added momentum causes a different type of problem. We actually cross  
the minimum point and have to take a U-turn to get to the minimum point.  
Momentum-based gradient descent oscillates around the minimum point, and  
we have to take a lot of U-turns to reach the desired point. Despite these  
oscillations, momentum-based gradient descent is faster than conventional  
gradient descent.  
  
To reduce these oscillations, we can use Nesterov Accelerated Gradient.  
  
NAG resolves this problem by adding a look ahead term in our equation. The intuition behi  
NAG can be summarized as ‘look before you leap’. Let us try to understand this through an}  
example.  
  
Wiook\_ahead  
Wo  
  
(a) Momentum-Based Gradient Descent (b) Nesterov Accelerated Gradient Descent  
  
OL \_ Negative(—) OL \_ Negative(  
  
=)  
= = 5 = 2 = 2 TY  
0 Owo —\_ Positive(+) 0 Owo Negative(—)

## Page 37

As can see, in the momentum-based gradient, the steps become larger and larger due to the  
accumulated momentum, and then we overshoot at the 4th step. We then have to take  
steps in the opposite direction to reach the minimum point.  
  
However, the update in NAG happens in two steps. First, a partial step to reach the look-  
ahead point, and then the final update. We calculate the gradient at the look-ahead point  
and then use it to calculate the final update. If the gradient at the look-ahead point is  
negative, our final update will be smaller than that of a regular momentum-based gradient.  
Like in the above example, the updates of NAG are similar to that of the momentum-based  
gradient for the first three steps because the gradient at that point and the look-ahead point  
are positive. But at step 4, the gradient of the look-ahead point is negative.  
  
In NAG, the first partial update 4a will be used to go to the look-ahead point and then the  
gradient will be calculated at that point without updating the parameters. Since the gradient  
at step 4b is negative, the overall update will be smaller than the momentum-based  
gradient descent.  
  
We can see in the above example that the momentum-based gradient descent takes six  
steps to reach the minimum point, while NAG takes only five steps.  
  
This looking ahead helps NAG to converge to the minimum points in fewer steps and reduce  
the chances of overshooting.  
  
How NAG Works  
  
We saw how NAG solves the problem of overshooting by ‘looking ahead’. Let us see how  
this is calculated and the actual math behind it.  
  
Update rule for gradient descent:  
  
Wee = We - Vt  
  
In this equation, the weight (W) is updated in each iteration. n is the learning rate, and Vwt  
is the gradient.  
  
Update rule for momentum-based gradient descent:  
  
In this, momentum is added to the conventional gradient descent equation. The update  
equation is  
  
Wei1 = Wy - update;  
  
update, is calculated by:  
update; = y - update;\_; + nVw;y

## Page 38

update =  
  
update - updatey + nVw, = nVui  
  
updates = 7 - update, + nVwe = 7: nVw, + 7Vwe  
  
updates - updates + nVw3 = ¥(y-7Vwi + nVwe) + 7Vw3  
- updateg + nVw3 = 77 -nVui + y-nVwe + nVw3  
  
update, - update; + nVwa “3. nVwi + a. nVwe+7-nVw3+nVwa  
  
update, - updates; + nVur = -}. nVwi + ym, nVui +... + 1Vwre  
  
This is how the gradient of all the previous updates is added to the current update.  
  
Update rule for NAG:  
  
Wi+1 = Wy — update;  
  
While calculating the update;, We will include the look ahead gradient (VWiggk ahead):  
update; = y - updatey\_1 + NVWigok\_ahead  
  
VWiook\_ahead is calculated by:  
  
Wlook\_ahead = Wt - Y\* updatey\_1  
  
This look-ahead gradient will be used in our update and will prevent overshooting.  
  
Pseudocode  
  
Initialize 6 (parameters), y (momentum), n (learning rate), and v (velocity vector)  
Repeat until convergence:  
1. Lookahead step: compute temporary parameters  
@\_lookahead = 6 - y \* v  
. Compute gradient at lookahead position  
g = Vf(6\_lookahead)  
. Update velocity using gradient  
vey\*vtn\*g  
. Update parameters using velocity  
  
@=@-v

## Page 39

Key Features  
1. Lookahead Gradient:  
  
e NAG anticipates the future position of the parameters based on momentum before  
  
computing the gradient.  
e This leads to more accurate updates, especially near minima.  
2. Faster Convergence:  
e The correction term helps prevent overshooting and improves convergence speed.  
3. Smoothness:  
  
e NAG provides smoother parameter updates compared to standard momentum methods.  
  
Advantages  
1. Improved Accuracy:  
e By accounting for the momentum’s effect, NAG often finds better minima.  
2. Better Near Minima:  
e Itis particularly effective at fine-tuning when the optimization is near a minimu  
3. General Applicability:  
  
¢ Can be used with non-convex loss functions common in deep learning.  
  
Comparison with Momentum  
  
Aspect Momentum NAG  
  
Gradient Location Current position Lookahead position  
  
Update Behavior Can overshoot minima Anticipates future updates, reducing overshooting  
Convergence Speed Slower in some cases Faster convergence  
  
Accuracy Near Minima Less precise More precise  
  
In frameworks like TensorFlow and PyTorch, NAG can be implemented easily  
  
39

## Page 40

import torch  
  
import torch.optim as optim  
  
Model =...#define your model  
  
Optimizer =optim.SGD(model.parameters(),lr=0.01, momentum=0.9,nesterov=True)  
  
Applications  
1. Training deep neural networks with high-dimensional parameter spaces.  
2. Accelerating convergence for convex and non-convex optimization problems.  
  
3. Reducing oscillations in optimization paths.  
  
   
  
Nesterov's Accelerated Gradient Descent is a robust and efficient tool, making it a popular choice fqr  
  
deep learning practitioners.  
  
2.3 Regularization  
A fundamental problem in machine learning is how to make an  
algorithm that will perform well not just on the training data, but also onnew  
inputs. Many strategies used in machine learning are explicitly designed to  
reduce the test error, possibly at the expense of increased training error.  
These strategies are known collectively as regularization.  
  
Definition: - “any modification we make to a learning algorithm that is  
intended to reduce its generalization error but not its training error.”  
  
o  
  
“+ In the context of deep learning, most regularization strategies are  
based on regularizing estimators.  
  
o  
  
\*\* Regularization of an estimator works by trading increased bias for  
reduced variance.  
  
An effective regularizer is one that makes a profitable trade, reducing  
variance significantly while not overly increasing the bias.  
  
Many regularization approaches are based on limiting the capacity of models,  
such as neural networks, linear regression, or logistic regression, by adding a  
parameter norm penalty Q(8) to the objective function J. We denote the  
regularized objective function by J~  
  
JO; X, y) = (0; X, y) + dQ)  
  
where a € [0, 00) is a hyperparameter that weights the relative contribution of  
the norm penalty term, Q, relative to the standard objective function J. Setting a to  
0 results in no regularization. Larger values of o correspond to more regularization.  
  
40

## Page 41

“+ The parameter norm penalty Q that penalizes only the weights of the affine  
transformation at each layer and leaves the biases unregularized.  
  
2.3.1 L2 Regularization  
One of the simplest and most common kind of parameter norm penalty is L2  
  
parameter & it’s also called commonly as weight decay. This regularization strategy  
drives the weights closer to the origin by adding a regularization term  
  
m2(e8)= %|[|w ll.  
  
Degree 4 Degree 15  
  
— Mode!  
—— True function  
  
—— Model  
—— True function  
  
e Samples © Samples  
  
Good Fit High Variance  
  
L2 regularization is also known as ridge regression or Tikhonov regularization. To  
simplify, we assume no bias parameter, so 0 is just w. Such a model has the  
following total objective function.  
  
J(w:X,y) = Sw Yaw + J(w;X,y),  
with the corresponding parameter gradient  
Vw (w; X,y) = aw + Vad (w; X,y). (  
To take a single gradient step to update the weights, we perform this update  
we—w-—e(awt+ VwJ(w;X,y)). (  
Written another way, the update is  
  
w< (1l—ca)w —eVwJ(w: X.y).  
  
“\* We can see that the addition of the weight decay term has modified the learning  
rule to multiplicatively shrink the weight vector by a constant factor on each step,  
  
just before performing the usual gradient update. This describes what happens in a  
single step.  
  
“+ The approximation AJ  
by  
  
Given

## Page 42

J(@) J (w\*) \* ( aw\*) | FA(w — w\*).  
  
Where H is the Hessian matrix of J with respect to w evaluated at w+.

## Page 43

The minimum of \*J occurs where its gradient VwJ(w) = H(w — w\*) is equal to ‘0’  
  
To study the eff ect of weight decay,  
  
ow + Fl (w — w\*) =o  
(FF + aF)w = Fw\*  
ap — (AF + at) ) Aw  
  
As a approaches 0, the regularized solution ~w approaches w\*. But what happens as a grows?  
Because H is real and symmetric, we can decompose it into a diagonal matrix A and an  
orthonormal basis of eigenvectors, Q, such that H = QAQ’. Applying Decomposition to the  
above equation, We Obtain  
  
a0 (QA! + at) 1QAQ! ww  
[QA + ang! ] QAQ!' w\*  
QCA + aT) 1AQ! w\*.  
  
Figure 2: Weight updation effect  
  
The solid ellipses represent contours of equal value of the unregularized objective. The dotted  
circles represent contours of equal value of the L 2 regularizer. At the point w~, these competing  
objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small.  
The objective function does not increase much when moving horizontally away from w\* . Because  
the objective function does not express a strong preference along this direction, the regularizer has a  
strong effect on this axis. The regularizer pulls w1 close to zero. In the second dimension, the  
objective function is very sensitive to movements away from w\* . The corresponding eigenvalue is  
large, indicating high curvature. As a result, weight decay affects the position of w2 relatively little.

## Page 44

L2 Regularization  
  
A linear regression that uses the L2 regularization technique is called ridge regression. In other words, in ridge  
regression, a regularization term is added to the cost function of the linear regression, which keeps the  
magnitude of the model's weights (coefficients) as small as possible. The L2 regularization technique tries to  
keep the model's weights close to zero, but not zero, which means each feature should have a low impact on the  
output while the model's accuracy should be as high as possible.  
  
l m  
2  
3 AY wi 4  
  
rl  
  
Ridge Regression Cost Function = Loss Function +  
  
Where \ controls the strength of regularization, and w, are the model's weights (coefficients).  
  
By increasing A, the model becomes flattered and underfit. On the other hand, by decreasing A, the model  
becomes more overfit, and with A = 0, the regularization term will be eliminated.  
  
   
  
2.3.2 L1 Regularization  
  
While L2 weight decay is the most common form of weight decay, there are other ways to  
penalize the size of the model parameters. Another option is to use L1 regularization.  
  
L1 Regularization  
  
Least Absolute Shrinkage and Selection Operator (/asso) regression is an alternative to ridge for regularizing  
linear regression. Lasso regression also adds a penalty term to the cost function, but slightly different, called L1  
regularization. L1 regularization makes some coefficients zero, meaning the model will ignore those features.  
Ignoring the least important features helps emphasize the model's essential features.  
  
m  
Lasso Regression Cost Function = Loss Function + 1y> wy  
jo  
  
Where \ controls the strength of regularization, and w, are the model's weights (coefficients).  
  
Lasso regression automatically performs feature selection by eliminating the least important features.

## Page 45

> L1 regularization on the model parameter w is defined as the sum of absolute values of the  
individual parameters.  
  
Q(@) = |\ew| la = SF Jewel.  
L1 weight decay controls the strength of the regularization by scaling the penalty Q using a  
  
positive hyperparameter a. Thus, the regularized objective function J°(w; X, y) is given by  
  
J(w;X,y) =a||w||i + J(w; X,y),  
with the corresponding gradient as  
  
Vw) (w; X,y) = asign(w) + Vwd(X,y;w), ——- fal  
  
By inspecting equation 1, we can see immediately that the effect of L 1 regularization is quite  
different from that of L 2 regularization. Specifically, we can see that the regularization  
contribution to the gradient no longer scales linearly with each wi ; instead it is a constant factor  
with a sign equal to sign(wi).  
  
Quadratic approximation of the L 1 regularized objective function decomposes into a sum over the parameters  
  
J(w: X,y) = J(w\*; X,y) + > jHis(w —wry)+ aluil :  
  
The problem of minimizing this approximate cost function has an analytical solution with the following form:  
  
s a  
w; = sign(w; ) max ¢ |w;| — 7 50>.  
a2  
  
Consider the situation where w \* i > 0 for all i. There are two possible outcomes:  
  
1. The case where w} < Ha: Here the optimal value of w; under the regularized  
objective is simply w; = 0. This occurs because the contribution of J (w; X,y)  
to the regularized objective J (w;X,y) is overwhelmed—in direction i—by  
the L! regularization, which pushes the value of w; to zero.  
  
. The case where w} > 7-. In this case, the regularization does not move the  
  
optimal value of w; to zero but instead just shifts it in that direction by a  
distance equal to 7-.

## Page 46

Elastic Net Regularization  
  
The third type of regularization,(you may have guessed by now) uses both  
most optimized output  
In addition to setting and choosing a lambda value elastic net also allows us to tune the alpha parameter where a  
  
orresponds to ridge and a = 1 to lasso, Simply pul, if you plug in 0 for alpha, the penalty function reduces to  
  
the L1 (ridge) term and if we set alpha to 1 we get the L2 (lasso) term  
  
Cost function of Elastic Net Regularization  
  
J( 8), Be, «+++» Bm) Vly 3 iP +Ma >> 3,|4 sey 3?)  
  
Therefore we can choose an alpha value between 0 and 1 to optimize the elastic net(here we can adjust the  
  
nlage of each regularization,thus giving the name elastic). Effectively this will shrink some coefficients and set  
  
© 0 for sparse selection  
  
2.3.3 Difference between L1 & L2 Parameter Regularization  
  
> L1 regularization attempts to estimate the median of data, L2 regularization makes estimation  
for the mean of the data in order to evade overfitting.

## Page 47

L1 regularization can add the penalty term in cost function. But L2 regularization appends the  
squared value of weights in the cost function.  
  
L1 regularization can be helpful in features selection by eradicating the unimportant features,  
whereas, L2 regularization is not recommended for feature selection  
  
L1 doesn’t have a closed form solution since it includes an absolute value and it is a non-  
differentiable function, while L2 has a solution in closed form as it’s a square of a weight  
  
   
  
S.No  
  
L1 Regularization  
  
L2 Regularization  
  
   
  
Panelizes the sum of absolute  
value of weights.  
  
penalizes the sum of square weights.  
  
   
  
It has a sparse solution.  
  
It has a non-sparse solution.  
  
   
  
It gives multiple solutions.  
  
It has only one solution.  
  
   
  
Constructed in feature selection.  
  
No feature selection.  
  
   
  
Robust to outliers.  
  
Not robust to outliers.  
  
   
  
It generates simple and  
interpretable models.  
  
It gives more accurate predictions when the output  
variable is the function of whole input variables.  
  
   
  
Unable to learn complex data  
patterns.  
  
Able to learn complex data patterns.  
  
   
  
   
  
   
  
Computationally inefficient over  
non-sparse conditions.  
  
   
  
Computationally efficient because of having  
analytical solutions.  
  
   
  
Early Stopping  
  
Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the  
  
validation set. When we see that the performance on the validation set is getting worse, we immediately  
  
stop the training on the model. This is known as early stopping.  
  
In the above image, we will stop training at the dotted line since after that our model will start overfitting  
  
on the training data.

## Page 48

2.4 Batch Normalization:  
  
It is a method of adaptive reparameterization, motivated by the difficulty of training  
very deep models.In Deep networks, the weights are updated for each layer. So the output  
will no longer be on the same scale as the input (even though input is  
normalized).Normalization - is a data pre-processing tool used to bring the numerical data to  
a common scale without distorting its shape.when we input the data to a machine or deep  
learning algorithm we tend to change the values to a balanced scale because, we ensure that  
  
our model can generalize appropriately.(Normalization is used to bring the input into a  
balanced scale/ Range).  
  
Let's understand this through an example, we have a deep neural network as shown in the following image.

## Page 49

Initially, our inputs X1, X2, X3, X4 are in normalized form as they are coming from the pre-processing stage. When  
the input passes through the first layer, it transforms, as a sigmoid function applied over the dot product of input  
X and the weight matrix W.  
  
xX, On j@  
  
Xx  
  
-  
W, ‘eo -  
  
h, = o(W,X)  
  
x h, = o(W,h,) = o(W,0(W,X))  
Normalize the inputs . ° °  
  
O = a(W,h, ,)  
  
wea  
  
Image Source: https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/  
  
Even though the input X was normalized but the output is no longer on the same scale. The  
data passes through multiple layers of network with multiple times(sigmoidal) activation functions  
are applied, which leads to an internal co-variate shift in the data.  
  
This motivates us to move towards Batch Normalization  
  
Normalization is the process of altering the input data to have mean as zero and standard deviation  
value as one.  
  
2.4.1 Procedure to do Batch Normalization:  
  
(1) Consider the batch input from layer h, for this layer we need to calculate the mean of this hidden  
activation.  
(2) After calculating the mean the next step is to calculate the standard deviation of the hidden  
activations.  
(3) Now we normalize the hidden activations using these Mean & Standard Deviation values. To do  
this, we subtract the mean from each input and divide the whole value with the sum of standard  
deviation and the smoothing term (¢).  
(4) As the final stage, the re-scaling and offsetting of the input is performed. Here two components  
of the BN algorithm is used, y(gamma) and f (beta). These parameters are used for re-scaling (y) and  
shifting(B) the vector contains values from the previous operations.  
  
These two parameters are learnable parameters, Hence during the training of neural network,  
the optimal values of y and £ are obtained and used. Hence we get the accurate normalization of each  
batch.

## Page 50

2.5. Shallow Networks  
Shallow neural networks give us basic idea about deep neural network which consist  
of only 1 or 2 hidden layers. Understanding a shallow neural network gives us an  
understanding into what exactly is going on inside a deep neural network A neural network is  
built using various hidden layers. Now that we know the computations that occur in a  
particular layer, let us understand how the whole neural network computes the output for a  
given input X. These can also be called the forward-propagation equations.  
  
ZN = Wurx + pl  
Allag (Z!4)  
Z2\ — weir All + pl  
  
= AM = (22)  
  
1 The first equation calculates the intermediate output Z/fof the first hidden layer.  
  
2, The second equation calculates the final outout A/jof the first hidden layer.  
  
3, The third equation calculates the intermediate output Z[2Jof the output layer.  
  
4, The fourth equation calculates the final output A/2Jof the output layer which is also the final  
output of the whole neural network.  
  
Shallow-Deep Networks: A Generic Modification to Deep Neural Networks  
  
Internal Layers Final Classifier  
  
-—-7TTT\_+1\_11\_\_  
  
conv1 conv2 conv3 conv4 Final ;  
Prediction  
  
r-L-; r-t-s,  
  
l 1  
‘FR; \ FR, [conv: Convolutional layer  
  
1!  
Internal  
Classifier i  
(IC) 1 full , .  
ot : FR: Feature Reduction layer  
7 :  
Internal Internal  
Prediction Prediction  
  
-- ¥ ull: Fully connected layer  
  
Figure 2:Shallow Networks — Generic Model  
  
50

## Page 51

2.5.1 Difference Between a Shallow Net & Deep Learning Net:  
  
Shallow Net’s  
  
One Hidden layer(or very less no. of  
Hidden Layers)  
  
Takes input only as VECTORS  
  
Shallow net’s needs more parameters  
to have better fit  
  
Shallow networks with one Hidden  
layer (same no of neurons as DL)  
cannot place complex functions over  
the input space  
  
The number of units in a shallow  
network grows exponentially with  
task complexity.  
  
Shallow network is more difficult to  
train with our current algorithms (e.g.  
it has issues of local minima etc)  
  
Deep Learning Net’s  
  
Deep Net’s has many layers of Hidden  
layers with more no. of neurons in  
each layers  
  
DL can have raw data like image, text  
as inputs  
  
DL can fit functions better with less  
parameters than a shallow network  
  
DL can compactly express highly  
complex functions over input space  
  
DL don’t need to increase it  
size(neurons) for complex problems  
  
Training in DL is easy and no issue of  
local minima in DL  
  
   
  
Dropout  
  
a,  
  
x oH t ae a  
  
\  
  
% MK “ ae ey  
  
(a) Standard Neural Network  
  
at  
) ntl  
ad  
  
(b) Network after Dropout

## Page 52

Drawbacks of Dropout:  
  
. Increased Training Time: Dropout increases the training time of the neural network, as the  
network needs to be trained multiple times with different subsets of neurons dropped  
out. However, this can be mitigated by parallelizing the training process.  
  
. Reduced learning rate: The use of dropout can reduce the effective learning rate of the  
network, which can slow down the learning process.  
  
. Can cause instability: In some cases, dropout can cause instability during training,  
particularly if the dropout rate is too high. This can be addressed by tuning the dropout  
rate and adjusting other hyperparameters.  
  
. Cannot be used with all types of networks: Dropout is not suitable for all types of neural  
networks, particularly those with a small number of neurons or those with a small number  
of layers.  
  
Adversial Training:  
  
Adversarial training is a technique used to improve the robustness  
and performance of machine learning models, particularly neural  
networks, by training them on adversarial examples.  
  
Adversarial examples are inputs modified by small, carefully  
calculated perturbations that lead to incorrect outputs from the  
model. For example:  
  
In image classification, a small noise addition might cause the  
model to misclassify an image of a cat as a dog.  
  
These perturbations are usually computed to maximize the  
model's loss for a given input.

## Page 53

Optimization  
\* Deep learning relies on optimization methods.  
\* The training efficiency of the model is directly influenced by the  
optimization algorithm's performance.  
  
\* Understanding the fundamentals of different optimization algorithms  
and the function of their hyperparameters, on the other hand, will  
allow us to modify hyperparameters in a targeted manner to improve  
deep learning model performance.  
  
\* In simple words, Optimization algorithms are responsible for reducing  
losses and provide most accurate results possible.  
  
¢ The weight is initialized using some initialization strategies and is  
updated with each epoch according to the equation.  
  
\* The best results are achieved using some optimization strategies or  
algorithms called Optimizer.  
  
The goal of Optimization in Deep learning  
  
¢ Although optimization may help deep learning by lowering the loss  
function, the aims of optimization and deep learning are fundamentally  
different.  
  
¢ The former is more focused on minimizing an objective, whereas the  
latter is more concerned with finding a good model given a finite  
  
quantity of data.  
  
¢ Training error and generalization error, for example, vary in that the  
optimization algorithm's objective function is usually a loss function  
based on the training dataset, and the purpose of optimization is to  
minimize training error.  
  
\* Deep learning (or, to put it another way, statistical inference) aims to  
decrease generalization error. In order to achieve the latter, we must be  
aware of overfitting as well as use the optimization procedure to lower  
the training error.

## Page 54

Some of the Optimization techniques:  
¢ Gradient Descent Deep Learning Optimizer  
¢ Stochastic Gradient Descent Deep Learning Optimizer  
¢ Mini-batch Stochastic Gradient Descent  
¢ Adagrad(Adaptive Gradient Descent) Optimizer  
¢ RMSprop (Root Mean Square) Optimizer  
¢ Adam Deep Learning Optimizer  
¢ AdaDelta Deep Learning Optimizer  
  
1. Gradient Descent Deep Learning Optimizer  
  
\* Gradient Descent is the most common optimizer in the class. Calculus is  
used in this optimization process to make consistent changes to the  
parameters and reach the local minimum.  
  
Gradient descent works with a set of coefficients, calculates their cost, and  
looks for a cost value that is lower than the current one. It shifts to a lesser  
weight and updates the values of the coefficients. The procedure continues  
until the local minimum is found. A local minimum is a point beyond which  
it is impossible to go any farther.  
  
This algorithm is apt for cases where optimal points cannot be found by  
equating the slope of the function to 0. For the function to reach minimum  
value, the weights should be altered. With the help of back propagation, loss  
is transferred from one layer to another and “weights” parameter are also  
modified depending on loss so that loss can be minimized.  
  
Cost function: 6=0—a-VJ(8)  
  
¢ As for Gradient Descent algorithm, the entire data set is loaded at a  
time. This makes it computationally intensive. Another drawback is  
there are chances the iteration values may get stuck at local minima or  
saddle point and never converge to minima. To obtain the best  
solution, the must reach global minima.  
  
   
  
Jw, b) global maximum  
  
local maximum  
  
local minimum  
  
global minimum  
  
   
  
   
  
1 Ll L  
0.2 0.4 0.6 0.8

## Page 55

\* Concept: Updates model parameters in the direction of the steepest descent of the loss  
  
function.  
  
¢ Formula:  
§=0-7n- VJ(8)  
  
where 77 is the learning rate and VJ(8) is the gradient of the loss.  
  
¢ Limitation: Requires the entire dataset for each update, making it slow for large datasets.  
  
2. Stochastic Gradient Descent Deep Learning Optimizer  
  
¢ On large datasets, gradient descent may not be the best solution. We use  
stochastic gradient descent to solve the problem. The word stochastic refers  
to the algorithm's underlying unpredictability. Instead of using the entire  
dataset for each iteration, we use a random selection of data batches in  
stochastic gradient descent. As a result, we only sample a small portion of  
the dataset. The first step in this technique is to choose the starting  
parameters and learning rate. Then, in each iteration, mix the data at random  
to get an estimated minimum. When compared to the gradient descent  
approach, the path taken by the algorithm is full of noise since we are not  
using the entire dataset but only chunks of it for each iteration.  
  
As a result, SGD requires more iterations to attain the local minimum. The  
overall computing time increases as the number of iterations increases.  
  
However, even when the number of iterations is increased, the computation  
cost remains lower than that of the gradient descent optimizer. As a result, if  
the data is large and the processing time is a consideration, stochastic  
gradient descent should be favored over batch gradient descent.  
  
Concept: Updates parameters using one data sample at a time, making it faster than regular  
  
gradient descent.  
  
Formula:  
  
6=8-n- VI(6;2;,yi)  
  
where (2;, yi) is a single data point.  
  
¢ Limitation: Noisy updates can cause fluctuations, making convergence less stable.

## Page 56

3. Mini-batch Stochastic Gradient Descent  
  
\* MB-SGD is an extension of SGD algorithm. It overcomes the time-  
consuming complexity of SGD by taking a batch of points / subset of  
points from dataset to compute derivative.  
  
¢ Because the method employs batching, all of the training data does  
not need to be placed into memory, making the process more  
efficient.  
  
\* In addition, the cost function in mini-batch gradient descent is noisier  
than that in batch gradient descent but smoother than that in  
stochastic gradient descent.  
  
© Concept: Combines the advantages of batch gradient descent and SGD by updating parameters  
  
using small batches of data.  
  
© Formula:  
  
1 n  
G=0-9-— VIO; 2i,%)  
  
i=1  
where n is the batch size.  
  
¢ Advantage: Balances speed and stability, making it widely used in practice.  
  
4. Adagrad(Adaptive Gradient Descent) Optimizer  
  
¢ Adaptive Gradient as the name suggests adopts the learning rateof  
parameters by updating it at each iteration depending on the position it is  
present, 1.e- by adapting slower learning rates when features are occurring  
frequently and adapting higher learning rate when features are infrequent.  
  
¢ Technically it acts on learning rate parameter by dividing the learning rate  
by the square root of gamma, which is the summation of all gradients  
squared.  
  
¢ In the update rule, AdaGrad modifies the general learning rate N at each  
step for all the parameters based on past computations. One of the biggest  
disadvantages is the accumulation of squared gradients in the denominator.  
Since every added term is positive, the accumulated sum keeps growing  
during the training. This makes the learning rate to shrink and eventually  
become small. This method is not very sensitive to master step size and also  
converges faster.

## Page 57

Concept: Adapts the learning rate for each parameter based on the history of gradients, giving  
  
larger updates to less frequently updated parameters.  
  
Update Rule:  
  
6=0-  
  
where G is the sum of past squared gradients.  
Advantage: Handles sparse data well.  
  
Limitation: Learning rate may decay too much over time.  
  
5. AdaDelta  
  
¢ It is simply an extension of AdaGrad that seeks to reduce its  
monotonically decreasing learning rate. Instead of summing all the  
past gradients, AdaDelta restricts the no. of summation values to a  
limit (w). In AdaDelta, the sum of past gradients (w) is defined as  
“Decaying Average of all past squared gradients”. The current average  
at the iteration then depends only on the previous average and current  
gradient.  
  
Concept: A refinement of Adagrad that restricts the accumulation of past squared gradients,  
  
allowing learning rates to vary adaptively over time.  
Update Rule:  
  
n°9  
VElg’| +e  
  
with updates scaled by a running average of parameter updates.  
  
e Advantage: No manual tuning of learning rates and solves Adagrad's decay issue.  
6. RMSprop (Root Mean Square) Optimizer  
  
\* Root Mean Squared Prop is another adaptive learning rate method that  
tries to improve AdaGrad. Instead of taking cumulative sum of  
squared gradients like in AdaGrad, we take the exponential moving  
average. The first step in both AdaGrad and RMSProp 1s identical.  
RMSProp simply divides learning rate by an exponentially decaying  
average.

## Page 58

¢ Concept: A variant of Adagrad that resolves the issue of decaying learning rates by using an  
  
exponentially weighted moving average of past gradients.  
Update Rule:  
u]  
  
/ Elg?| +€ 9  
  
where E[g”] is the moving average of squared gradients.  
  
Advantage: Works well for non-stationary objectives (e.g., deep learning).  
  
7. Adaptive Moment Estimation (Adam) Deep Learning Optimizer  
  
\* It is a combination of RMSProp and Momentum. This method  
computes adaptive learning rate for each parameter. In addition to  
storing the previous decaying average of squared gradients, it also  
holds the average of past gradient similar to Momentum. Thus, Adam  
behaves like a heavy ball with friction which prefers flat minima in  
error surface.  
  
W.1=(1—-A) w,—9V f,(w,)  
  
Momentum update Nesterov momentum update  
  
lookahead” gradient  
  
step (bit different thar  
momentum ‘onginal)  
step  
actual step  
  
Concept: Combines the benefits of RMSprop and momentum by using adaptive learning rates  
  
gradient  
step  
  
and maintaining exponentially moving averages of gradients and their squares.  
Update Rule:  
  
m  
" Vite  
  
where 7 and @ are bias-corrected estimates of the mean and variance of gradients.  
  
6=6-  
  
e Advantage: Fast convergence and widely used in deep learning.