Rishabh Indoria
21F3001823

Deep Learning

October 15, 2024

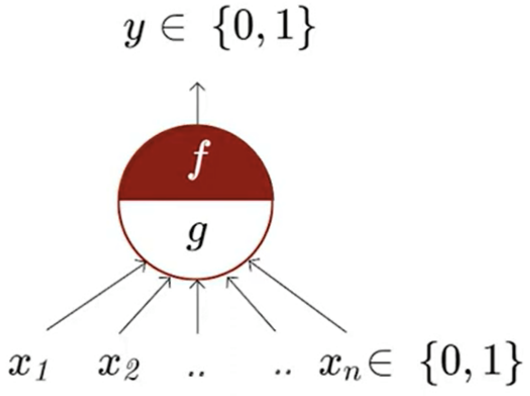
1 History of Deep Learning

1.1 Biological Neurons

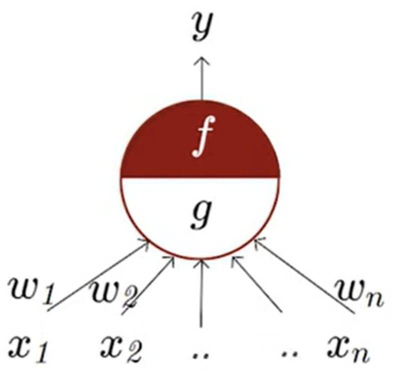
- **Reticular Theory**: Nervous system is a single continuous network as opposed to a network of many discrete cells.
- **Staining Technique**: Chemical reaction that allows to examine nervous tissue in much greater detail.
- **Neuron Doctrine**: Nervous system is actually made up of discrete individual cells forming a network.

1.2 From Spring to Winter of AI

- **McCulloch Pitts Neuron**: A highly simplified model of the neuron.
- **Perceptron**: "The perceptron may eventually be able to learn, make decisions, and translate languages."
- "The embryo of an electric computer that the Navy expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."
- Book "**Perceptrons**" by Minsky and Papert outlined the limits of what perceptrons could do.
- **Backpropagation**: First used in the context of artificial neural networks.
- **Universal Approximation Theorem**: A multilayered network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision.



(a) Simple Model of the Neuron



(b) Model of the Perceptron

Figure 1: First models of neurons

1.3 The Deep Revival

- **Unsupervised Pre-Training**: Described an effective way of initializing the weights that allows deep autoencoders networks to learn a low-dimensional representation of data.

1.4 From Cats to Convolutional Neural Networks

- **Hubel and Wiesel Experiment**: Experimentally showed that each neuron has a fixed receptive field -i.e. a neuron will fire only in response to visual stimuli in a specific region in the visual space(Motivation for CNNs).

1.5 Faster, higher, stronger

- **Better Optimization Methods**: Faster convergence, better accuracies. Examples: AdaGrad, RMSProp, Adam, Nadam, etc.
- **Better Activation Functions**: Many new functions have been proposed, leading to better convergence and performance. Example: tanh, ReLu, Leaky Relu, etc.

1.6 The Curios Case of Sequences

- **Sequences**: Each unit in the sequence interacts with other units. Need models to capture this interaction. Example: Speech, Videos, etc.
- **Hopfield Network**: Content-addressable memory systems for storing and retrieving patterns.
- **Jordan Network**: The output state of each time step is fed to the next time step, thereby allowing interactions between time steps in the sequence.
- **Elman Network**: The hidden state of each time step is fed to the next time step, thereby allowing interactions between time steps in the sequence.
- Very hard to train RNNs.
- **Long Short Term Memory**: Solve complex long time lag tasks that could never be solved before(solved the problem of vanishing gradient).
- **Sequence to Sequence Models**: Introduction to Attention!!, However, they were unable to capture the contextual information of a sentence.
- **Transformers**: Introduced a paradigm shift in the field of NLP. GPT and BERT are the most commonly used transformer based architectures.

1.7 Beating humans at their own game

- Human-level control through deep reinforcement learning for playing Atari Game
- **OpenAI Gym**: Toolkit for developing and comprising reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.
- **OpenAI Gym Retro**: A platform for reinforcement learning research on games, which contains 1,000 games across a variety of backing emulators.
- **MuZero**: Masters Go, chess, shogi and Atari without needing to be told the rules, thanks to its ability to plan winning strategies in unknown environments.
- **Player of Games(PoG)**: A general purpose algorithm that unifies all previous approached. Learn to play under both perfect and imperfect information games.

1.8 The rise of Transformers

- **Rule Based Systems**: Initial Machine Translation Systems used handcrafted rules and dictionaries to translate sentences between few politically important language pairs.
- **Statistical MT**: The IBM Models for Machine Translation gave a boost to the idea of data driven statistical NLP, probability based models.
- **Neural MT**: The introduction of seq2seq models and attention. Bigger, hungrier, better models.
- **From Language to Vision**: A vision model based as closely as possible on the Transformer architecture, originally designed for text-based tasks.
- **From Discrimination to Generation**: Sample(Generate) data from the learned probability distribution.Variable Auto Encoders(VAE), Generational Adversarial Networks(GAN), Flow-based models to achieve it.
- GANs don't scale, are unstable and capture less diversity. Diffusion models are one of the alternatives to GAN. Inspired by an idea from non-equilibrium thermodynamics.

1.9 Call for Sanity

- **Paradox of Deep Learning**: Deep learning works so well despite high capacity(susceptible to overfitting), numerical instability(vanishing/exploding gradients), sharp minima(leads to overfitting), non-robustness.
- **Interpretable Machine Learning: A Guide for Making Black Box Models Explainable** by Christoph Molnar.
- **AI Audit challenge**: AI systems must be evaluated for legal compliance, in particular laws protecting people from illegal discrimination. This challenge seeks to broaden the tools available to people who want to analyze and regulate them.

- **Analog AI**: Programmable resistors are the key building blocks in analog deep learning, just like transistors are the core element of digital processors(faster computation).

1.10 The AI revolution in basic Science Research

- **Protein-Folding Problem**: Model proposed to predict protein structure, which would lead to better drug development.
- **Astronomy: Galaxy Evolution**: Predict how galaxies would look like as it gets older.

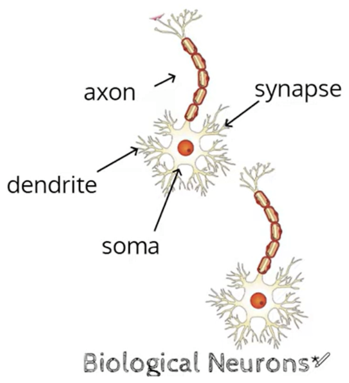
1.11 Efficient Deep Learning

- Build models on small devices, aka phones. Deploying models in resource-constrained devices.

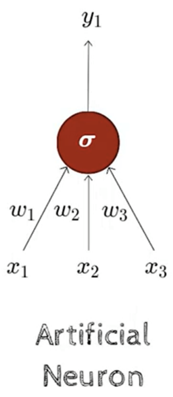
2 Multi Layered Network of Perceptrons

2.1 Biological Neurons

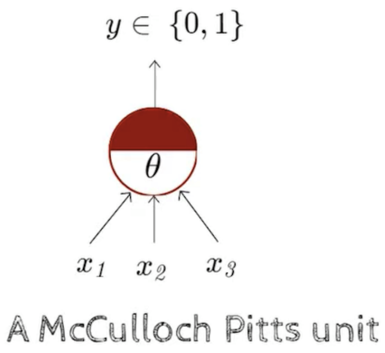
- The most fundamental unit of a deep neural network is called an artificial neuron.
- The inspiration comes from biology(more specifically, from the brain).
- **biological neurons = neural cells = neural processing units**
- **Dendrite**: Receives signals from other neurons
  Synapse: Point of connection to other neurons
  Soma: Processes the information
  Axon: Transmits the output of this neuron.
- This massively parallel network also ensures that there is division of work. Each neuron may perform a certain role or respond to a certain stimulus. The neurons in the brain arranged in a hierarchy.



(a) Biological Neuron



(b) Artificial Neuron

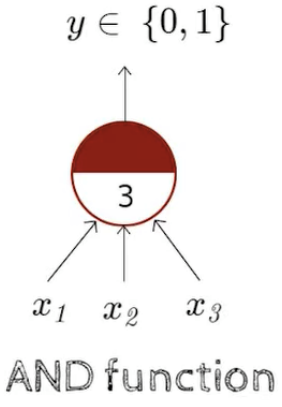


(c) A McCulloch Pitts Unit

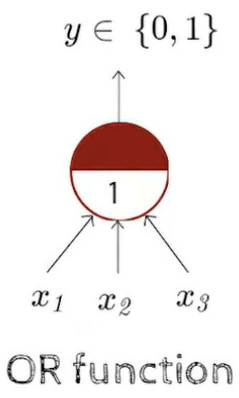
Figure 2: Basic Neuron Structure

- **McCulloch Pitts Neuron**: Proposed a highly simplified computational model of the neuron. g aggregates the inputs, and the function f takes a decision based on this aggregation. The inputs can be excitatory or inhibitory.
  Example: $y = 0$ if any x_i is inhibitory, else $g(x_1, ..., x_n) = g(x) = \sum_{i=1}^{n} x_i$
  $y = f(g(x)) = 1$ if $g(x) \geq \theta$ else 0
  θ is called the thresholding parameter.
- circle at the end indicates inhibitory input if any inhibitory input is 1 then the output will be 0.
- Linear separability(for boolean functions): There exists a line(plane) such that all inputs which produce a 1 lie on one side of the line(plane) and all inputs which produce a 0 lie on the other side of the line(plane).
- A single McCulloch Pitts Neuron can be used to represent boolean functions which are linearly separable.

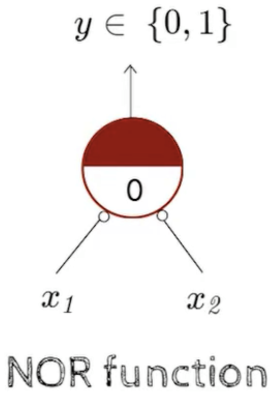
- It can be trivially seen that $\theta = 0$ for the Tautology(always ON) function.



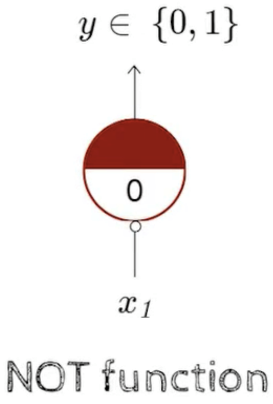
(a) AND function



(b) OR function



(c) NOR function

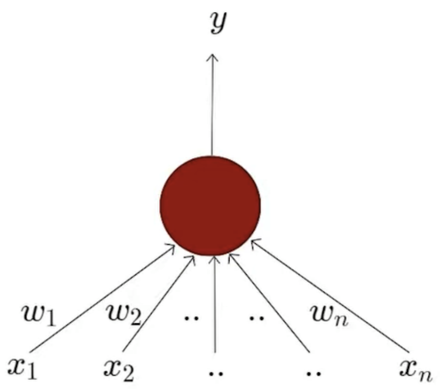


(d) NOT function

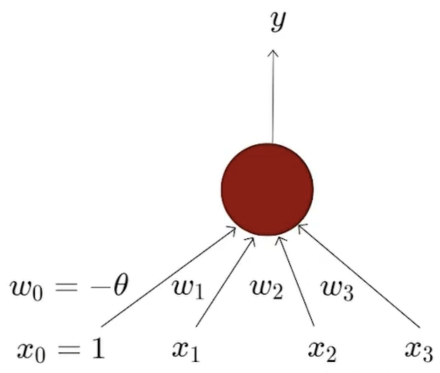
Figure 3: Basic Boolean functions

2.2 Perceptrons

- **Classical Perceptron**: Frank Rosenblatt proposed this model, which was refined and carefully analyzed by Minsky and Papert.
- Main difference is the introduction of numerical weights for inputs and a mechanism for learning these weights.
- $y = 1$ *if* $\sum_{i=1}^{n} w_i \times x_i \geq \theta$ else 0.
  If we take θ on one side and represent $w_0 = -\theta$ and $x_0 = 1$, then we have
  $y = 1$ *if* $\sum_{i=0}^{n} w_i \times x_i \geq 0$ else 0.
- w_0 is called the bias as it represents the **prior**(prejudice).
- From the equation, it can be clearly seen that a perceptron separates the input space into two halves. A single perceptron can only be used to implement linearly separable functions.
- The difference between this and the MP neuron is that now we have weights that can be learned, and the inputs can be real values.



(a) Perceptron



(b) Perceptron with w_0

Figure 4: Classical Perceptrons

2.3 Perceptron Learning Algorithm

- **Perceptron learning algorithm**: Consider two vectors $w = [w_0, ..., w_n]$ and $x = [1, x_1, ..., x_n]$, then $w \times x = w^T x$ or the dot product. We can rewrite the perceptron rule as $y = 1$ *if* $w^T x \geq 0$ else 0.
- We are interested in finding the line $w^T x = 0$ which divides the input space into two halves. The angle α between w and any point x which lies on the line must be 90° because $\cos\alpha = \frac{w^T x}{||w||||x||} = 0$. So, the vector w is perpendicular to every point on the line, then it is perpendicular to the line itself.

- For any point such that $w^T x > 0$ we will have $\alpha < 90°$ and for any point such that $w^T x < 0$ we will have $\alpha > 90°$.

Algorithm 1 Perceptron Learning Algorithm

```
 1: $P \leftarrow$ inputs with label 1;
 2: $N \leftarrow$ inputs with label 0;
 3: Initialize $\mathbf{w}$ randomly
 4: while !convergence do
 5:     Pick random $x \in P \cup N$;
 6:     if $x \in P$ and $\sum_{i=0}^{n} w_i \times x_i < 0$ then
 7:         $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
 8:     end if
 9:     if $x \in N$ and $\sum_{i=0}^{n} w_i \times x_i \geq 0$ then
10:         $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
11:     end if
12: end while
13: ▷ the algorithm converges when all the inputs are classified correctly
```

- For $x \in P$ if $w^T x < 0$ then it means that the angle α between this x and the current w is greater than $90°$, but we want it to be $< 90°$.
  When we do $w_{new} = w + x$, alpha changes as follows
  $$\cos\alpha_{new} \propto (w_{new}^T x) = (w+x)^T x = W^T x + x^T x = \cos\alpha + x^T x$$
  $$\cos\alpha_{new} > \cos\alpha \implies \alpha_{new} < \alpha$$
  α_{new} becomes less than, α which is exactly what we wanted, we may not get $< 90°$ in one shot so we keep doing it. Similarly, it can be shown for $x \in N$.

- **Definition**: Two sets P and N of points in an $n-$dimensional space are called absolutely linearly separable if $n+1$ real numbers $w_0, ..., w_n$ exist such that every point $(x_1, ..., x_n) \in P$ satisfies $\sum_{i=1}^{n} w_i \times x_i \geq w_0$ and every point $(x_1, ..., x_n) \in N$ satisfies $\sum_{i=1}^{n} w_i \times x_i < w_0$.

- **Proposition**: If the sets P and N are finite and linearly separable, the perceptron learning algorithm updates the weight vector w a finite number of times. In other words: if the vectors in P and N are tested cyclically one after the other, a weight vector w is found after a finite number of steps t which can separate the two sets.

- **Proof of Convergence**: If $x \in N$, then $-x \in P$ ($\because w^T x < 0 \implies w^T(-x) \geq 0$).
  We can now consider only a single set $P' = P \cup N^-$ and for every element $p \in P'$ ensure that $w^T p \geq 0$.

Algorithm 2 Modified Perceptron Learning Algorithm

```
 1: $P \leftarrow$ inputs with label 1;
 2: $N \leftarrow$ inputs with label 0;
 3: $N^- \leftarrow$ negations of all points in $N$;
 4: $P' \leftarrow P \cup N^-$
 5: Initialize $\mathbf{w}$ randomly
 6: while !convergence do
 7:     Pick random $p \in P'$;
 8:     if $\sum_{i=0}^{n} w_i \times p_i < 0$ then
 9:         $\mathbf{w} = \mathbf{w} + \mathbf{p}$;
10:     end if
11: end while
12: ▷ the algorithm converges when all the inputs are classified correctly
```

Further we will normalize all the p's so that $||p|| = 1$, notice this does not change anything in our step.
Let w^* be the normalized solution vector(we know one exists whose value we don't know).
Now suppose at some time step t we inspected the point p_i and found that $w^T p_i < 0$, then we make correction $w_{t+1} = w_t + p_i$.
Let β be the angle between w^* abd w_{t+1}, then we have $\cos\beta = \frac{w^* w_{t+1}}{||w_{t+1}||}$

$$Numerator\ = w^* \cdot w_{t+1} = x^* \cdot (w_t + p_i) = w^* \cdot w_t + w^* \cdot p_i \geq w^* \cdot w_t + \delta \ (\delta = \min(w^* \cdot p_i | \forall i))$$
$$\geq w^* \cdot (w_{t-1} + p_j) + \delta \geq w^* \cdot w_{t-1} + w^* \cdot p_j + \delta \geq w^* \cdot w_{t-1} + 2\delta \geq w^* w_0 + (k)\delta \ (By\ Induction)$$

$$\begin{aligned} Denominator^2 &= ||w_{t+1}||^2 = (w_t + p_i) \cdot (w_t + p_i) = ||w_t||^2 + 2w_t \cdot p_i + ||p_i||^2 \le ||w_t||^2 + ||p_i||^2 \\ &\le ||w_t||^2 + 1 \le (||w_{t-1}||^2 + 1) + 1 \le ||w_0||^2 + (k) \end{aligned}$$

So, we have $Numerator \ge w^* \cdot w_0 + (k)\delta$ and $Denominator^2 \le ||w_0||^2 + (k)$

$$\cos\beta \ge \frac{w^* \cdot w_0 + k\delta}{\sqrt{||w_0||^2 + k}}$$

$\cos\beta$ grows proportional to $\sqrt{k}$
As k(number of corrections) increase $\cos\beta$ can become arbitrarily large, but since $\cos\beta \le 1$, k must be bounded by a maximum order.
Thus, there can only be a finite number of corrections (k) to w and the algorithm will converge!

2.4 Linearly Separable Boolean Function

- One simple example that is not linearly separable is XOR

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \ge 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \ge 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

Table 1: XOR Truth Table

- Most real world data is not linearly separable and will always contain some **outliers**.
- How many boolean functions can we design from 2 inputs.

x_1	x_2	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 2: Functions of 2 inputs

Out of these, only XOR and !XOR are not linearly separable.

- In general, we can have 2^{2^n} boolean functions in n inputs.

2.5 Representation Power of a Network of Perceptrons

- We will assume True $= +1$ and False $= -1$, we consider 2 inputs and 4 perceptrons with specific weights. The bias of each perceptron is -2. Each of these perceptrons is connected to an output perceptron by weights (which need to be learned). The output of this perceptron (y) is the output of this network.

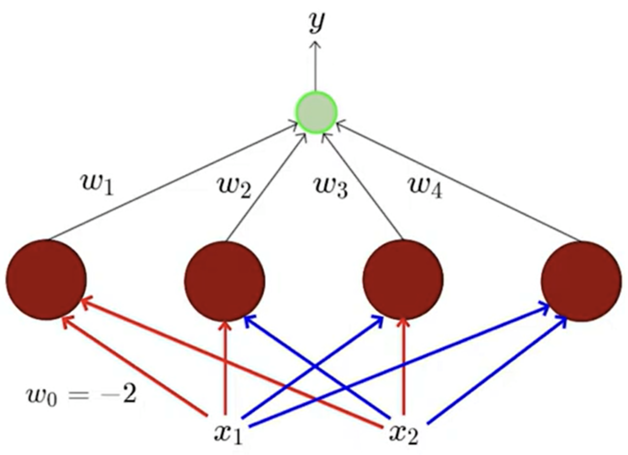


Figure 5: Network of Perceptrons

red edge indicates $w = -1$, and blue edge indicates $w = +1$. x_1 has weights $-1, -1, +1, +1$ and x_2 has weights $-1, +1, -1, +1$ from left to right respectively.

- The above network contains 3 layers.
  The layer containing the inputs (x_1, x_2) is called the **input layer**.
  The middle layer containing the 4 perceptrons is called the **hidden layer**.
  The final layer containing one output neuron is called the **output layer**.
  The output of the 4 perceptrons in the hidden layer are denoted by h_1, h_2, h_3, h_4.
  The red and blue edges are called layer 1 weights
  w_1, w_2, w_3, w_4 are called layer 2 weights.

- This network can be used to implement **any** boolean function linearly separable or not. Each perceptron in the middle layer fires only for a specific input and no perceptrons fire for the same input.

x_1	x_2	XOR	h_1	h_2	h_3	h_4	$\sum_{i=1}^{4} w_i h_i$
0	0	0	1	0	0	0	w_1
1	0	1	0	1	0	0	w_2
0	1	1	0	0	1	0	w_3
1	1	0	0	0	0	1	w_4

Table 3: Truth Table for the Network

  This results in four independent conditions.

- In case of 3 inputs we would now have 8 perceptrons in the hidden layer.
- **Theorem**: Any boolean function of n inputs can be represented exactly by a network of perceptrons containing one hidden layer with 2^n perceptrons and one output layer containing 1 perceptron.
- **Note**: A network of $2^n + 1$ perceptrons is not necessary but sufficient.
- **Catch**: As n increases the number of perceptrons in the hidden layers increases exponentially.
- We care about boolean functions, because we can model real world examples into boolean functions or classification problems.
- The network we saw is formally known as Multilayer Perceptron(MLP) or more appropriately "Multilayered Network of Perceptrons".

3 Sigmoid Neurons

3.1 What are they?

- The thresholding logic used by a perceptron is very harsh! It is a characteristic of the perceptron function itself which behaves like a **step function**.
- For most real world applications we would expect a smoother decision function which gradually changes from 0 to 1.
- Instead we would use sigmoid function

$$y = \frac{1}{1 + \exp(-(\omega_0 + \sum_{i=1}^{n} \omega_i x_i))}$$

- We no longer see a sharp transition around the threshold ω_0. Also, y now takes any real value in $[0, 1]$.
- This value can also be interpreted as probability.

3.2 Typical Supervised Machine Learning Setup

- **Data**: $\{x_i, y_i\}_{i=1}^{n}$, we have n data point where x_i is a vector of $\mathbb{R}^m$. Assume $y = \hat{f}(x; \theta)$.
- **Model**: Our approximation of the relation between x and y.

$$\text{For example, } \hat{y} = \frac{1}{1 + e^{-w^T x}} \text{ or } \hat{y} = w^T x \text{ or } \hat{y} = x^T W x$$

- **Learning algorithm**: An algorithm for learning the parameters w of the model.
- **Objective/Loss/Error function**: To guide the learning algorithm. One possibility is $\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$.
- The learning algorithm should aim to minimize the loss function.

3.3 Learning Parameters

- For ease of explanation we will take $f(x) = \frac{1}{1+e^{-(wx+b)}}$, only a single parameter.
- Assume input for training data is $\{x_i, y_i\}_{i=1}^{N} \to N$ pairs of (x, y).
- Training Objective: Find w and b such that $\mathcal{L}(w,b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i))^2$ is minimized.
- **Guess Work(Infeasible)**: Intuitively guess what should be the values of w and b. May never end and is not feasible for writing algorithms.
- **Update rule**: $\theta_{new} = \theta + \eta\Delta\theta$, how to find $\Delta\theta$?
- **Taylor Series**: A way of approximating any continuously differentiable function $\mathcal{L}(w)$ using polynomials of degree n. The higher the degree the better the approximation!

$$\mathcal{L}(w) = \mathcal{L}(w_0) + \frac{\mathcal{L}'(w_0)}{1!}(w - w_0) + \frac{\mathcal{L}''(w_0)}{2!}(w - w_0)^2 + ...$$

- Linear approximation is the first order approximation of the Taylor series. Quadratic approximation is the second order approximation of the Taylor series.
- **Key point**: You can only do this for a very small $\Delta = w - w_0$.
- Let's assume $\Delta\theta = u$, then from Taylor series we have

$$\mathcal{L}(\theta + \eta u) = \mathcal{L}(\theta) + \eta u^T \nabla_\theta \mathcal{L}(\theta) + \frac{\eta^2}{2!} u^T \nabla_\theta^2 \mathcal{L}(\theta) u + ... = \mathcal{L}(\theta) + \eta u^T \nabla_\theta \mathcal{L}(\theta)$$

- η is typically small, so $\eta^2, \eta^3, ... \to 0$
- Now, we want new loss less than the current loss

$$\mathcal{L}(\theta + \eta u) - \mathcal{L}(\theta) < 0 \implies u^T \nabla_\theta \mathcal{L}(\theta) < 0$$

- Let β be the angle between u^T and $\nabla_\theta \mathcal{L}(\theta)$ and $k = \|u^T\|\|\nabla_\theta \mathcal{L}(\theta)\|$, then we can write

$$-1 \leq cos(\beta) = \frac{u^T \nabla_\theta \mathcal{L}(\theta)}{\|u^T\|\|\nabla_\theta \mathcal{L}(\theta)\|} \leq 1 \implies -k \leq u^T \nabla_\theta \mathcal{L}(\theta) \leq k$$

- $u^T \nabla_\theta \mathcal{L}(\theta)$ is most negative when $cos(\beta) = -1$ or $\beta = 180°$
- **Parameter Update Rule**: From the above we can now say

$$w_{t+1} = w_t - \eta \nabla w_t$$
$$b_{t+1} = b_t - \eta \nabla b_t$$

- We can now write the gradient descent algorithm for our problem as follows

Algorithm 3 Gradient Descent

```
GradientDescent
1: $t \leftarrow 0$
2: $max\_iterations \leftarrow 1000$
3: $w, b \leftarrow$ initialize randomly
4: while $t < max\_iterations$ do
5:     $w_{t+1} \leftarrow w_t - \eta \nabla w_t$
6:     $b_{t+1} \leftarrow b_t - \eta \nabla b_t$
7:     $t \leftarrow t + 1$
8: end while
```

- where ∇w and ∇b are defined for sigmoid neuron as

$$\nabla w = \frac{\partial \mathcal{L}(x)}{\partial w} = (f(x) - y)\frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right) = (f(x) - y) \times f(x) \times (1 - f(x)) \times x$$

$$\nabla b = \frac{\partial \mathcal{L}(x)}{\partial b} = (f(x) - y) \times f(x) \times (1 - f(x))$$

3.4 Representation Power of a multiplayer network of sigmoid neurons

- A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision.

4 Feed Forward Neural Networks

4.1 Structure of Feed Forward Neural Network

- The input to the network is an $n-$dimensional vector.
- The network contains $L-1$ hidden layers having n neurons each. Value of n could be different in each layer.
- Finally, there is one output layer containing k neurons, corresponding to $kclasses$.
- Each neuron in the hidden layer and output layer can be split into two parts: pre-activation and activation(a_i and h_i are vectors).
- The input layer can be called the $0-$th layer and the output layer can be called the $L-$th layer.
- $W_i \in \mathbb{R}^{n\times n}$ and $b_i \in \mathbb{R}^n$ are the weight and bias between layers $i-1$ and i $(0 < i < L)$.
- $W_L \in \mathbb{R}^{k\times n}$ and $b_L \in \mathbb{R}^k$ are the weight and bias between the last hidden layer and the output layer.
- The pre-activation at layer i is given by
$$a_i(x) = b_i + W_i h_{i-1}(x)$$
- The activation at layer i is given by
$$h_i(x) = g(a_i(x))$$
- g is an element wise function and is called the **activation function**.
- The activation at the output layer is given by
$$f(x) = h_L(x) = O(a_L(x))$$
where O is the output activation function
- For ease of notation $a_i(x) = a_i$ and $h_i(x) = h_i$.
- We also have $h_0 = x$ and $h_L = \hat{y} = \hat{f}(x)$
- We can write $\hat{y}$ as
$$\hat{y}_i = \hat{f}(x_i) = O(W_3 g(W_2 g(W_1 x_i + b_1) + b_2) + b_3)$$
This becomes our model as specified in 3.2.
- Parameters become $\theta = W_1, ..., W_L, b_1, ..., b_L$
- We can now write our gradient descent algorithm more concisely as

Algorithm 4 Gradient Descent Modified

```
GRADIENT-DESCENT( )
1: $t \leftarrow 0$;
2: $max_iterations \leftarrow 1000$;
3: Initialize $\theta_0 = [W_1^0, ..., W_L^0, b_1^0, ...., b_L^0]$
4: while $t++ < max_iterations$ do
5:     $\theta_{t+1} \leftarrow \theta_t - \eta\nabla\theta_t$
6: end while
```

where we have

$$\theta = [W_1, ..., W_L, b_1, ..., b_L] \text{ and } \nabla\theta_t = [\frac{\partial\mathcal{L}(\theta)}{\partial W_{1,t}}, ..., \frac{\partial\mathcal{L}(\theta)}{\partial W_{L,t}}, \frac{\partial\mathcal{L}(\theta)}{\partial b_{1t}}, ..., \frac{\partial\mathcal{L}(\theta)}{\partial b_{1t}}]^T$$

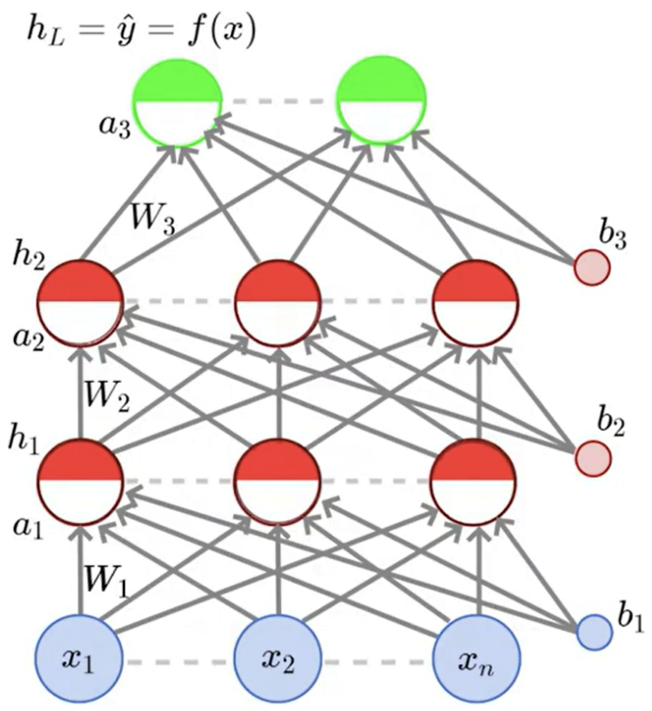


Figure 6: Feed Forward Network

4.2 Output functions and Loss functions

- Assume we are trying to predict values in the range of all real values, an appropriate loss function would be mean squared error function.

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k}(\hat{y}_{ij} - y_{ij})^2$$

- If we want to predict values in the range of real values, regression, then we can take the output function as a linear function

$$f(x) = h_L = O(a_L) = W_O a_L + b_O$$

- Given that we are performing classification, then we can use cross entropy loss function. This is known as negative log likelihood function.

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k}(y_{ij}log(\hat{y}_{ij}) + (1 - y_{ij})log(1 - \hat{y}_{ij}))$$

- Ensure that $\hat{y}$ is a probability distribution, one appropriate output function can be the softmax function

$$\hat{y}_j = O(a_L)_j = \frac{e^{a_{L,j}}}{\sum_{i=1}^{k} w^{a_{L,i}}}$$

	Real Values	Probabilities
Output Activation	Linear	Softmax
Loss Function	Squared Error	Cross Entropy

Table 4: Choice of Output and Loss functions based on type of Output

4.3 Backpropagation

- Intuitively it can be written as

$$\underbrace{\frac{\partial \mathscr{L}(\theta)}{\partial W_{111}}}_{\text{Talk to the weight directly}} = \underbrace{\frac{\partial \mathscr{L}(\theta)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_3}}_{\text{Talk to the output layer}} \underbrace{\frac{\partial a_3}{\partial h_2} \frac{\partial h_2}{\partial a_2}}_{\text{Talk to the previous hidden layer}} \underbrace{\frac{\partial a_2}{\partial h_1} \frac{\partial h_1}{\partial a_1}}_{\text{Talk to the previous hidden layer}} \underbrace{\frac{\partial a_1}{\partial W_{111}}}_{\text{and now talk to the weights}}$$

Figure 7: Backpropagation Intuition

- For further discussion, the output function is considered to be softmax and the loss function is considered to be cross entropy.
- **Gradient w.r.t output layer**: We start with the part "Talk to the output layer".

$$\mathcal{L}(\theta) = -\log \hat{y}_l \ (l = \text{ true class label})$$

$$\frac{\partial}{\partial \hat{y}_i} \mathcal{L}(\theta) = \begin{cases} -\frac{1}{\hat{y}_l}, & \text{if } i = l \\ 0, & \text{otherwise} \end{cases} = [0, ..., 0, -\frac{1}{\hat{y}_l}, 0, ..., 0]^T = -\frac{1}{\hat{y}_l} e_l$$

$e_l$is a $k-$dimensional vector whose l^{th} element is 1 and rest are 0.

$$\frac{\partial \mathcal{L}(\theta)}{\partial a_{L,i}} = -\frac{1}{\hat{y}_l} \frac{\partial \hat{y}_l}{\partial a_{L,i}}$$

Since, $\hat{y}_l$ is calculated using cross entropy it is dependent on all a_L, i

$$\hat{y}_j = \frac{e^{a_{Lj}}}{\sum_i e^{a_{Li}}}$$

$$\frac{\partial \hat{y}_l}{\partial a_{L,i}} = \begin{cases} \hat{y}_l(1 - \hat{y}_l), & \text{if } i = l \\ -\hat{y}_l \hat{y}_i, & \text{if } i \neq l \end{cases}$$

We can now write the entire derivative in one shot as

$$\frac{\partial \mathcal{L}(\theta)}{\partial a_{L,i}} = -\frac{1}{\hat{y}_l}(\hat{y}_l)(1_{l=i} - \hat{y}_i) = -(e_l - \hat{y})$$

- **Chain rule among multiple paths**: If a function $p(z)$ can be written as a function of intermediate results $q_i(z)$ then we have

$$\frac{\partial p(z)}{\partial z} = \sum_m \frac{\partial p(z)}{\partial q_m} \frac{\partial q_m}{\partial z}$$

- **Gradient w.r.t hidden units**: Now we "Talk to the hidden layers", in this case $p(z)$ is the loss function, $z = h_{ij}$ and $q_m(z) = a_{Lm}$. We have

$$\frac{\partial \mathcal{L}(\theta)}{\partial h_{ij}} = \sum_{m=1}^{k} \frac{\mathcal{L}(\theta)}{\partial a_{i+1,m}} W_{i+1,m,j}$$

$$\frac{\partial \mathcal{L}(\theta)}{\partial a_{ij}} = \frac{\partial \mathcal{L}(\theta)}{\partial h_{ij}} \frac{\partial h_{ij}}{\partial a_{ij}} = \frac{\partial \mathcal{L}(\theta)}{\partial h_{ij}} g'(a_{ij})$$

- **Gradient w.r.t Parameters**: Finally, we "talk to the weights"

$$\frac{\partial \mathcal{L}(\theta)}{\partial W_{kij}} = \frac{\partial \mathcal{L}(\theta)}{\partial a_{ki}} \frac{\partial a_{ki}}{\partial W_{kij}} = \frac{\partial \mathcal{L}(\theta)}{\partial a_{ki}} h_{k-1,j}^T$$

$$\frac{\partial \mathcal{L}(\theta)}{\partial b_{ki}} = \frac{\partial \mathcal{L}(\theta)}{\partial a_{ki}} \frac{\partial a_{ki}}{\partial b_{ki}} = \frac{\partial \mathcal{L}(\theta)}{\partial a_{ki}}$$

- We can now write the full pseudocode as

Algorithm 5 Forward Propagation

```
1: for $k = 1$ to $L - 1$ do
2:     $a_k = b_k + W_k h_{k-1}$
3:     $h_k = g(a_k)$
4: end for
5: $a_L = b_L + W_L h_{L-1}$
6: $\hat{y} = O(a_L)$
```

Algorithm 6 Backward Propagation

```
1: $\nabla_{a_L} \mathcal{L}(\theta) = -(e(y) - \hat{y})$
2: for $k = L - 1$ to $1$ do
3:     $\nabla_{W_k} \mathcal{L}(\theta) = \nabla_{a_k} \mathcal{L}(\theta) h_{k-1}^T$
4:     $\nabla_{b_k} \mathcal{L}(\theta) = \nabla_{a_k} \mathcal{L}(\theta)$
5:     $\nabla_{h_{k-1}} \mathcal{L}(\theta) = W_k^T \nabla_{a_k} \mathcal{L}(\theta)$
6:     $\nabla_{a_{k-1}} \mathcal{L}(\theta) = \nabla_{h_{k-1}} \mathcal{L}(\theta) \odot [..., g'(a_{k-1,j}), ...]$
7: end for
```

- g' for logistic function $\sigma(z)$ is $g(z)(1 - g(z))$ and for tanh function is $1 - (g(z))^2$.
- On flat surfaces, gradient descent moves very slow. How do we solve this?

5 Gradient Descent Types

5.1 Momentum based Gradient Descent

- **Intuition**: If I am repeatedly being asked to move in the same direction then I should probably gain some confidence and start taking bigger steps in that direction. Just as a ball gains momentum while rolling down a slope.
- Update rule for momentum based gradient descent is

$$\begin{aligned} u_t &= \beta u_{t-1} + \nabla w_t, \qquad u_0 = \nabla w_0,\ u_{-1} = 0 \\ w_{t+1} &= w_t - \eta u_t \end{aligned}$$

- We are not only considering the current gradient, but we are also giving some importance to past history. β is typically less than 1, so we give decreasing importance to previous histories. We have

$$u_t = \beta u_{t-1} + \nabla w_t = \beta^2 u_{t-2} + \beta \nabla w_{t-1} + \nabla w_t = ... = \sum_{\tau=0}^{t} \beta^{t-\tau} \nabla w_\tau$$

- In addition to current update, also look at the history of updates.
- Even in the regions having gentle slopes, momentum based gradient descent is able to take large steps because the momentum carries it along.
- However, there is a possibility of overshooting our goal. This overshoot could lead to oscillations as well.
- Despite these oscillations, it will converge faster than gradient descent.

5.2 Nesterov Accelerated Gradient Descent

- Can we do something to reduce the oscillations observed in momentum based gradient descent?
- **Intuition**: Look before you leap, we ask the weights to move by two parts βu_{t-1} and ∇w_t. The idea is to first move by βu_{t-1} and then compute ∇w_t. So instead of relying only on current gradient we are essentially "looking ahead" and computing new gradient.
- Update rule for NAG is

$$\begin{aligned} u_t &= \beta u_{t-1} + \nabla(w_t - \beta u_{t-1}) \\ w_{t+1} &= w_t - \eta u_t \\ &\text{with } u_{-1} = 0,\ \text{and } 0 \le \beta \le 1 \end{aligned}$$

5.3 Stochastic vs Batch Gradient Descent

- Regular gradient descent goes over the entire data once before updating the parameters. Because this is the true gradient of the loss as derived earlier. Hence, theoretical guarantees hold.
- Imagine we have a million points in the training data, then it will take very long.
- In **stochastic** version, we update the parameters for every point in the data. Now if we have a million data points then we will make a million updates in each epoch.
- However, this is an approximate gradient. Hence, we have no guarantee that each step will decrease the loss.
- So, going over a large data once is bad, going over a single point and updating is bad, but going over some data points and updating is ok.
- This is the idea for **mini-batch** gradient descent.
- 1 epoch is one pass over the entire data, 1 step is one update of the parameters, N is the number of data points, and B is the mini batch size.
- So, for regular gradient descent the number of epochs is the same as number of steps, for stochastic gradient descent the number of steps is the same as N, and for mini-batch gradient descent the number of steps is the same as $\frac{N}{B}$.
- It is intuitive that a larger batch size is better, but we sacrifice on time.

Algorithm	**# of steps in 1 epoch**
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini Batch Gradient Descent	$\frac{N}{B}$

Table 5: Stochastic vs Mini Batch Gradient Descent

5.4 Scheduling Learning Rate

- Instead of using momentum and NAG we could have simply increased the learning rate, but on regions which have a steep slope, the already large gradient would blow up farther.
- What we need is for the learning rate to be small when gradient is high and vice versa.
- Tune learning rate, try different values on a log scale: $0.0001, 0.001, 0.001, 0.1, 1.0$.
- Run a few epochs with each of these and figure out a learning rate which works best. Now do a finer search around this value.
- These are just heuristics, no clear winning strategy.
- **Annealing learning rate**: Decrease the learning rate as we get close to the minima.
- **Step Decay**: Halve the learning rate after every 5 epochs or halve the learning rate after an epoch if the validation error is more than what it was at the end of the previous epoch.
- **Exponential Decay**: $\eta = \eta_0^{-kt}$, where η_0 and k are hyperparameters and t is a step number. However, choosing k becomes complex.
- **1/t Decay**: $\eta = \frac{\eta_0}{1+kt}$, again choosing k is a bit tricky, ideally we won't use these learning rates.
- For momentum the following schedule was suggested by Sutskever et al., 2013

  $$\beta_t = min(1 - 2^{-1-\log_2(\lfloor \frac{t}{250} \rfloor + 1)}, \beta_{max})$$

  β_{max} is chosen from $\{0.999, 0.995, 0.99, 0.9, 0\}$
- In practice, often a line search is done to find a relatively better value of η. Update w using different values of η then pick the best w based on loss function.

6 Adaptive Learning Rates

- **Intuition**: Decay the learning rate for parameters in proportion to their update history(more updates means more decay).
- Update rule for **AdaGrad**

$$v_t = v_{t-1} + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \times \nabla w_t$$

  and similar set of equations for b_t.

- **Intuition**: AdaGrad decays the learning rate very aggressively(as the denominator grows). As a result, after a while, the frequent parameters will start receiving very small updates. To avoid this we can decay the denominator.
- Update rule for **RMSprop**

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \times \nabla w_t$$

  Give lower and lower weightage to previous gradients

- RMSprop converges more quickly than AdaGrad by being less aggressive on decay.
- In RMSprop, the deltas start oscillating after a while, learning rate can increase, decrease or remain constant. Whereas in AdaGrad learning rate can only decrease as the denominator constantly grows.
- In case of oscillations, setting η_0 properly could solve the problem.
- Both are **sensitive** to initial learning rate, initial conditions of parameters and corresponding gradients.
- **AdaDelta** avoids setting initial learning rate η_0

$$v_t = \beta v_{t-1} + (1-\beta)(\nabla w_t)^2$$
$$\Delta w_t = -\frac{\sqrt{u_{t-1} + \epsilon}}{\sqrt{v_t + \epsilon}} \nabla w_t$$
$$w_{t+1} = w_t + \Delta w_t$$
$$u_t = \beta u_{t-1} + (1-\beta)(\Delta w_t)^2$$

  Now the numerator, in the effective learning rate, is a function of past gradients. Observe that the u_t that we compute at t will be used only in the next iteration. Essentially one history runs behind the other.

- After some i iterations, v_t will start decreasing and the ratio of the numerator to the denominator starts increasing. If the gradient remains low for a subsequent time steps, then the learning rate grows accordingly.
- Therefore, AdaDelta allows the numerator to increase or decrease based on the current and past gradients.
- **Intuition**: Do everything that RMSprop does to solve the decay problem of AdaGrad, plus used a cumulative history of the gradients.
- Update rule for **Adam(Adaptive Moments)**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla w,\ \hat{m}_t = \frac{m_t}{1-\beta_1^t} \leftarrow \text{Incorporating classical momentum}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2,\ \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

  Note that we are taking a running average of the gradients as m_t.

- One way of looking at this is that we are interested in the expected value of the gradient and not on a single point estimate computed at time t.
- $\hat{m}_t$ and $\hat{v}_t$ are bias correction terms, we do this to avoid very high learning rate update. It can also be derived by taking expectation of m_t.

- Bias correction is the problem of exponential averaging.
- General form of L^p norm is $(|x_1|^p + |x_2|^p + ... + |x_n|^p)^{\frac{1}{p}}$, as $p \to \infty$, L^p boils down to taking the max value.
- In Adam, we called $\sqrt{v_t}$ as L^2 norm, why not replace it with L^∞ norm. Therefore, we have

$$\begin{aligned}
v_t &= \max(\beta_2^{t-1}|\nabla w_1|, \beta_2^{t-2}|\nabla w_2|, ..., |\nabla w_t|) \\
v_t &= \max(\beta_2 v_{t-1}, |\nabla w_t|) \\
w_{t+1} &= w_t - \frac{\eta_0}{v_t}\nabla w_t
\end{aligned}$$

  Observe that we didn't use bias corrected v_t as max norm is not susceptible to initial zero bias.
- Suppose that we initialize w_0 such that the gradient at the w_0 is high. Suppose further that the gradients for the next subsequent iterations are also zero because x is sparse. Ideally, we don't want the learning rate to change its value when $\nabla w_t = 0$.
- Update Rule for **MaxProp** is similar to RMSprop

$$\begin{aligned}
v_t &= \max(\beta v_{t-1}, |\nabla w_t|) \\
w_{t+1} &= w_t - \frac{\eta}{v_t + \epsilon}\nabla w_t
\end{aligned}$$

- We can extend the same idea to Adam and call it **AdaMax**

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla e_t, \ \hat{m}_6 = \frac{m_t}{1-\beta_1^t} \\
v_t &= \max(\beta_2 v_{t-1}, |\nabla w_t|) \\
w_{t+1} &= w_t - \frac{\eta}{v_t + \epsilon}\hat{m}_t
\end{aligned}$$

- **Intuition**: We know NAG is better than momentum based GD, why not just incorporate it with Adam.
- We can rewrite NAG such that there is no $t-1$ term, as

$$\begin{aligned}
g_{t+1} &= \nabla w_t \\
m_{t+1} &= \beta m_t + \eta g_{t+1} \\
w_{t+1} &= w_t - \eta(\beta m_{t+1} + g_{t+1})
\end{aligned}$$

- Update rule for **NAdam(Nesterov Adam)**

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla w, \ \hat{m}_t = \frac{m_t}{1-\beta_1^t} \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla w_t)^2, \ \hat{v}_t = \frac{v_t}{1-\beta_2^t} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon}(\hat{m}_{t+1} + \frac{(1-\beta_1)\nabla w_t}{1-\beta_1^{t+1}})
\end{aligned}$$

- It is common for new learners to start out using off the shelf optimizers, which are later replaced by custom designed ones.
- **Cyclical Learning Rate**: Suppose the loss surface has a saddle point. Suppose further that the parameters are initialized, and the learning rate is decreased exponentially. After some iterations, the parameters will reach near the saddle point. Since, the learning rate has decreased exponentially, the algorithm has no way of coming out of the saddle point.
  What if we allow the learning rate to increase after some iterations.
- **Rationale**: Often, difficulty in minimizing the loss arises from saddle points rather than poor local minima.
- A simple way to solve this is to vary the learning rate cyclically. One example is triangular learning rate.
- **Cosine Annealing(Warm Re-Start)**: also based on cyclical learning rate

$$\eta_t = \eta_{min} + \frac{\eta_{max} - \eta_{min}}{2}(1 + \cos(\pi\frac{t\%(T+1)}{T})), \ T \text{ is restart interval}$$

- **Warm-start**: Using a low initial learning rate helps the model to warm and converge better.