Date: 26 / april / 2020

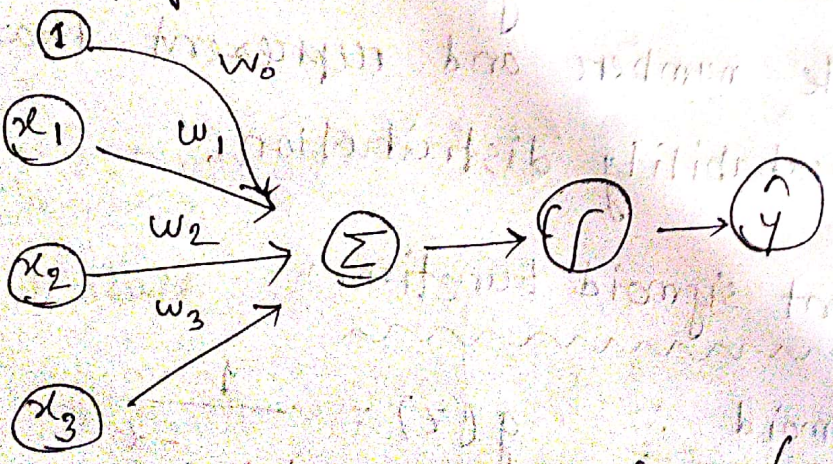
* Always focus on documentation
* Linear algebra (matrix multiplication)

Author: Kazi Amit Hasan
Email: kaziamithasan89@gmail.com

Introduction to deep learning : (MIT 6.S191)

The Perceptron
- The structure building block of deep learning

Forward Propagation



$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

w_0 → bias

$\hat{y}$ → output

g → non-linear activation function

$\sum_{i=1}^{m} x_i w_i$ → Linear combination of inputs

bias: used to shift the activation function left/right based on inputs

Protivas

Date : ____/____/____

activation function

$$\hat{y} = g(w_0 + x^T w)$$

where, $X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ $W = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

Ex: Sigmoid function.

$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$

* sigmoids are really useful when we want a single number and represent that number as probability distribution.

Different sigmoid functions:

① Sigmoid : $g(z) = \frac{1}{1+e^{-z}}$

$$g'(z) = g(z)(1 - g(z))$$

Protivas

Date : ...../...../.....

(ii) Hyperbolic tangent:

$$g(z) = \frac{e^{z} - e^{-z}}{1 - g(z)^2}$$

$$g'(z) = 1 - g(z)^2$$

(iii) Rectified Linear unit (ReLU)

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1 & ; \quad z > 0 \\ 0 & ; \quad \text{otherwise} \end{cases}$$

Importance of activation function:

- to introduce non-linearities into the network.

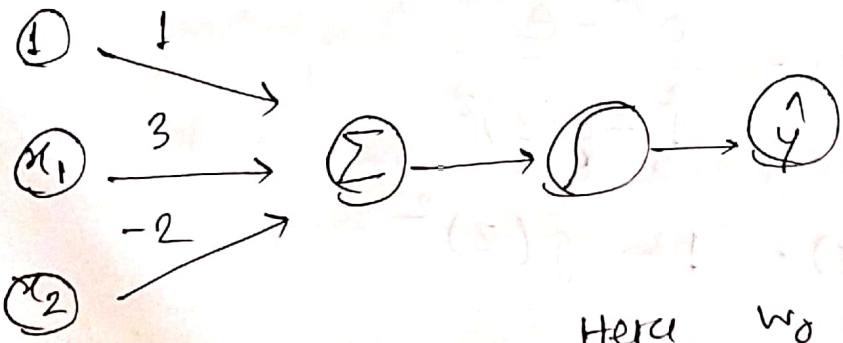
* Linear activation function produce linear decisions no matter what the network size is.

* Non-linearities allow us to approx arbitary complex functions.

Protivas

Date :...../...../.....

Examples



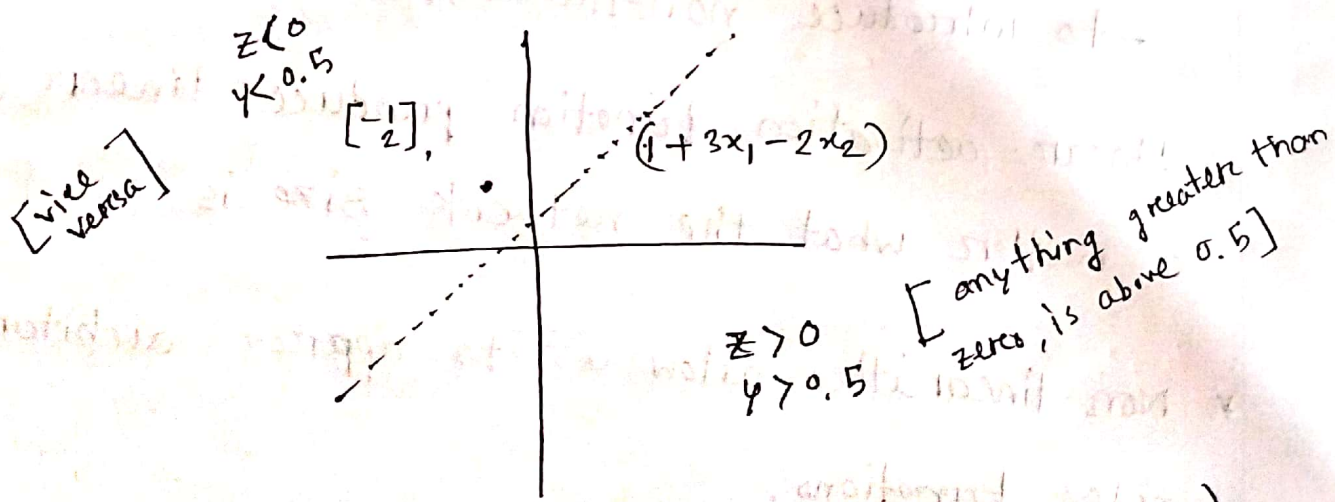
Here, $w_0 = 1$

$$w = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$

$$\therefore \hat{y} = g(w_0 + x^T w)$$

$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

$$\therefore \hat{y} = g(1 + 3x_1 - 2x_2) \quad [\text{2D line}]$$



Assume $x = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$ $\quad \therefore \hat{y} = g(1 + 3(-1) - (2 \cdot 2))$

$$= g(-6) \approx 0.002$$

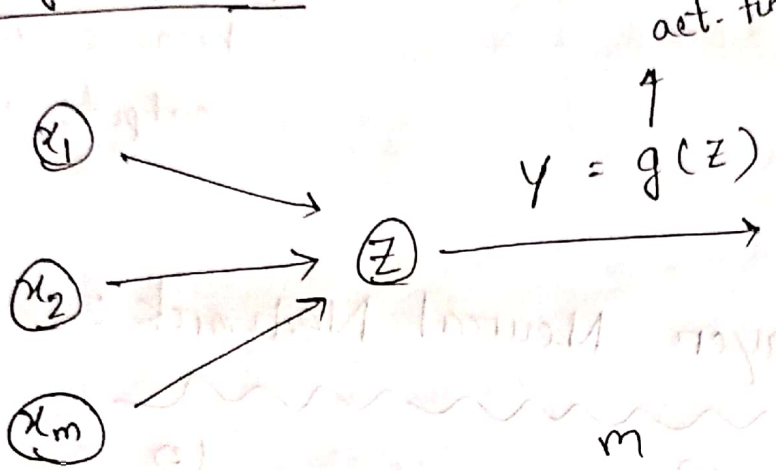
[sigmoid]

Protivas

Date : ..../..../....

Building Neural Networks with Perceptron:

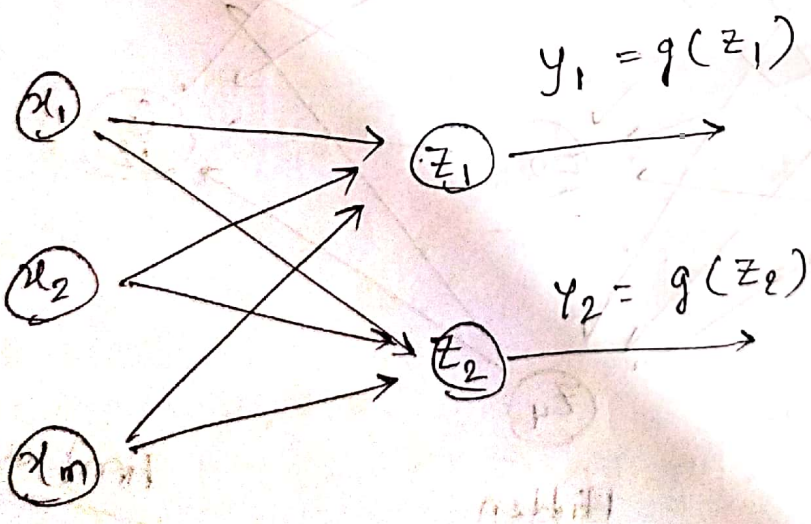
Perceptron (simplified):



$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$

Multi-output Perceptron:

$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$



Dense layer: all inputs are densly connected to all outputs.

Protivas

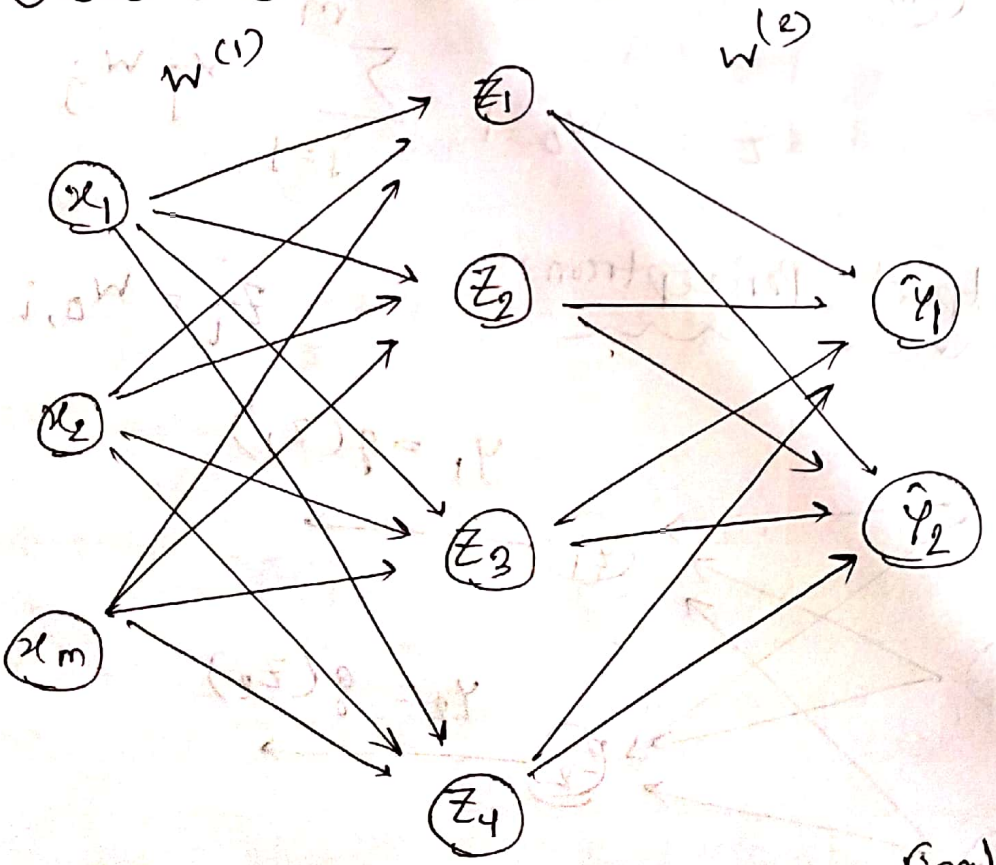
Date :...../...../.....

```
import tensorflow as tf
layer = tf.keras.layers.Dense(units = 2)
```

↓

because we have two outputs in the layer

Single Layer Neural Network :



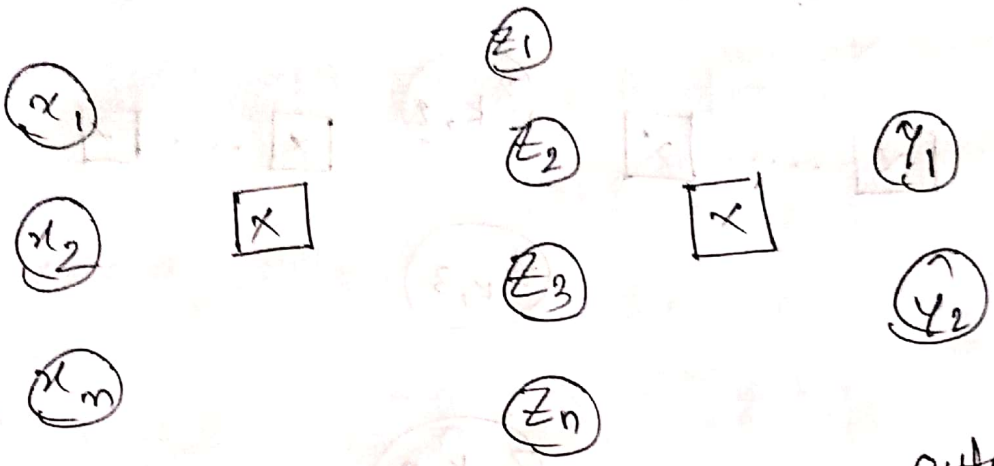
Inputs — Hidden — final output

$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)}$$

$$\hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

Date : ____/____/____

Multi output perceptron :

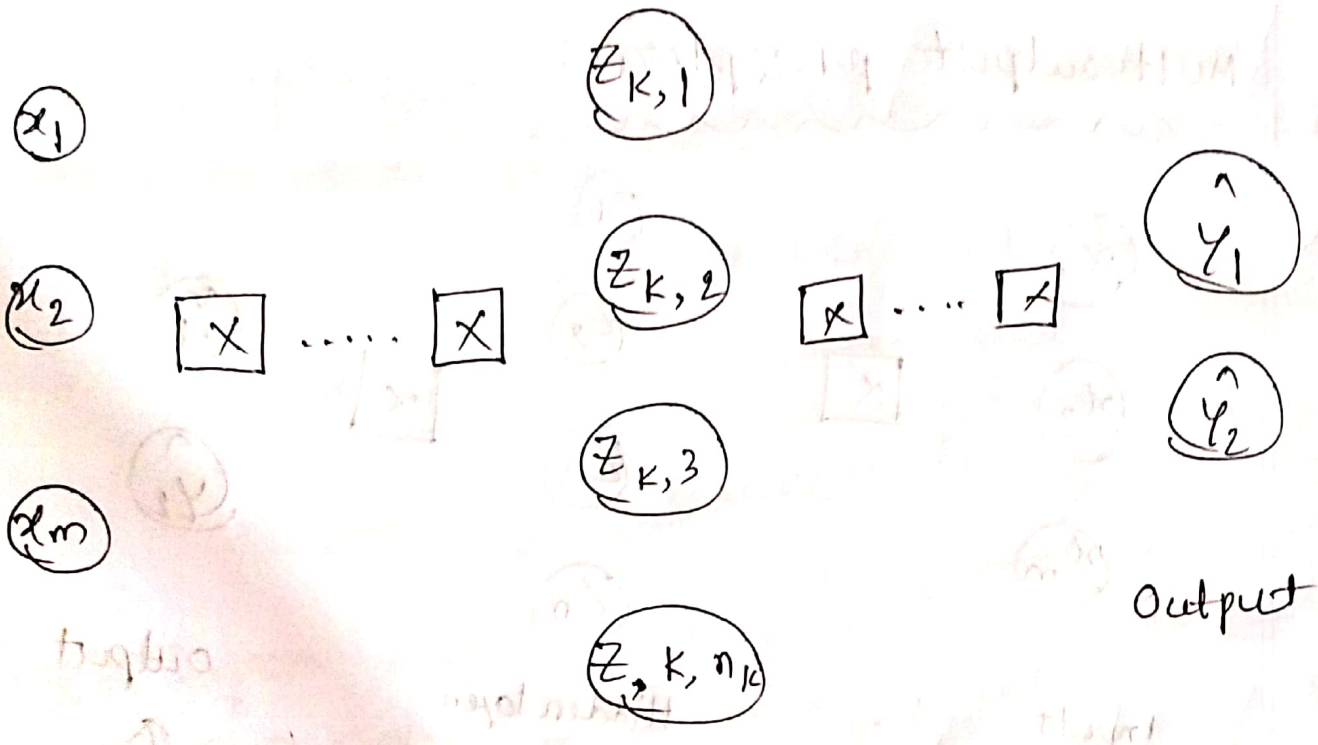



```
model = tf.keras.Sequential([
        tf.keras.layers.Dense(n),
        tf.keras.layers.Dense(2)])
```

Deep Neural Network :

P.T.O.

Date : ...../...../.....



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

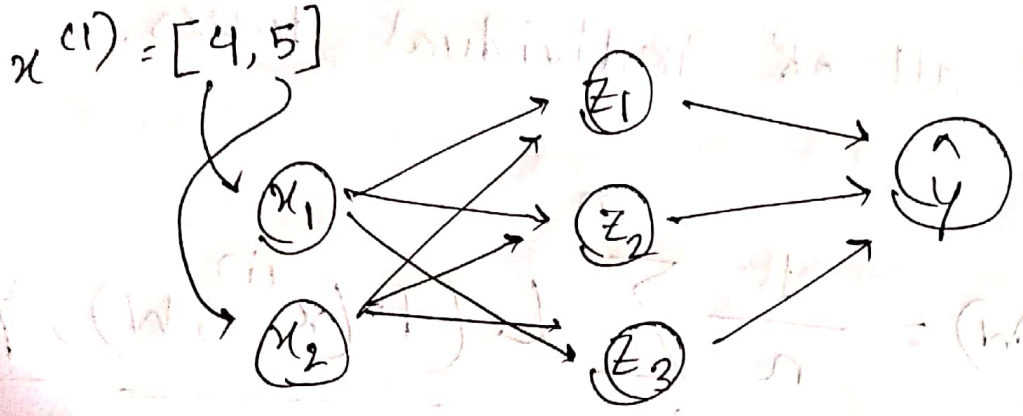
```
tf.keras.layers.Dense(n1),
tf.keras.layers.Dense(n2),
    ⋮          ⋮
tf.keras.layers.Dense(2)
```

Applying Neural Networks :

Ex: Will I pass?

x_1 = No. of lectures attended

x_2 = Hours spent on project



Pred = 0.1

Which is wrong

Actual = 1 (Yes, passed)

train করানো হয়নি

Quantifying loss:

- the loss of network that measures that cost incurred from incorrect predictions.

$$\mathcal{L}\left(\underbrace{f(x^{(i)}; W)}_{\text{pred}}, \underbrace{y^{(i)}}_{\text{actual}}\right)$$

Empirical Loss: / obj function / Cost function / Empirical Risk

The empirical loss measures the total loss over entire dataset.

- Avg of all of individual losses.

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} L\underbrace{(f(x^{(i)}; W)}_{\text{pred}}, \underbrace{y^{(i)}}_{\text{actual}})$$

$$\begin{bmatrix} f(x) \\ 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \begin{matrix} \\ \times \\ \times \\ \checkmark \\ \\ \end{matrix} \begin{bmatrix} y \\ 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

Binary Cross Entropy loss:

```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, predicted))
```

Date :...../...../.....

- Cross entropy loss can be used with models that's output is between 0 & 1.

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{actual}} \log\left(\underbrace{f(x^{(i)}; W)}_{\text{pred}}\right) + \underbrace{(1-y^{(i)})}_{\text{act}} \log \left(1 - \underbrace{f(x^{i}; W)}_{\text{pred}}\right)$$

- What if we have a regression problem instead to classification problem? [grade কমবে]

Mean Squared Error loss:

- used to regression model that output continuous real numbers.

- Prediction of network, actual true output, then substracting them, take the squared error and that's the loss, this is the network should optimize.

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \left(\underbrace{y^{(i)}}_{\text{actual}} - \underbrace{f(x^{(i)}; W)}_{\text{pred}}\right)^2$$

Date : ____/____/____

Training Neural Networks:

Loss Optimization:

- we need to find the network that achieve the lowest loss.

$$W^* = \arg\min \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f(x^{(i)}; W), y^{(i)}\right)$$

$$W^* = \arg\min_W J(W)$$

set of weights that tries to minimize the loss

* Loss function is a function of the network weights

Date :...../...../.....

compute gradient = $\frac{\partial j(w)}{\partial(w)}$

↳ we are taking this to understand the direction of maximum ascent,. basically tells us the conditions of the path (up/down)]

→ up direction খুঁজে পাওয়ার পর, reverse করব, lowest point-এ যাওয়ার, until it converge to a local minima.

This is gradient descent algorithm

Algo: ① Initialize weights randomly

② Loop until convergence

③ compute gradient = $\frac{\partial j(w)}{\partial(w)}$

④ Update weights, $W \leftarrow W - \eta \frac{\partial J(w)}{\partial(w)}$

↓ learning rate

⑤ Return weights

while True:

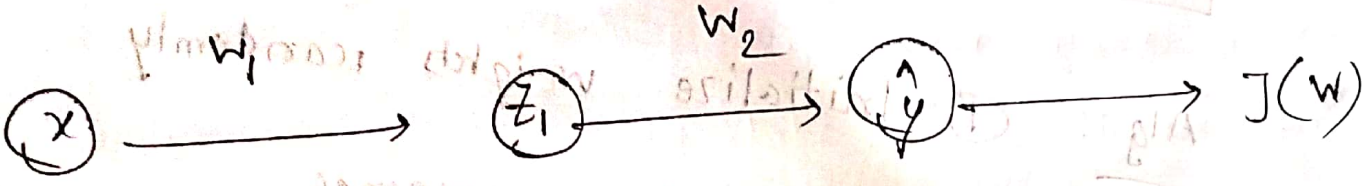
with tf.GradientTape() as g:

loss = compute_loss(weights)

gradient = g.gradient(loss, weights)

weights = weights - lr * gradient

back-propagation : (application of chain rule)



How a small change in one weight (w_2/w_1) can affect the final loss!

Date : ____/____/____

$$\frac{\partial J(W)}{\partial W_2} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial W_2} \quad \text{[chain rule]}$$

Again,

$$\frac{\partial J(W)}{\partial W_1} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial W_1}$$

$$= \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial \hat{z}_1} \times \frac{dz_1}{\partial W_1}$$

[chain rule]

→ Repeat this for every weight in the layer using gradients from ~~latter~~ later layers.

Date : ____/____/____

Neural Networks in Practice!

Optimization:

Loss functions can be difficult to optimize.

optimizing through gradient descent:

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$

η → learning rate (determines how much steps we should take in the direction of the gradient)

little small ↑ are

- small learning rate converges slowly and get stuck in false local minima

- we may stuck in local minima because we are not as aggressive we should be

Protivas

Date: ____/____/____

- Large learning rate overshoot, become unstable and diverge

- Smooth learning rate converges smoothly and avoid local minima.

How to deal with it?

① Try a lot of different learning rates.

② Design adaptive learning rate that adapts to the landscape.

Adaptive learning rates:

- means learning rates are no longer fixed.
- it can increase/decrease through training.

depending on: ① how large the gradient is
② how fast the learning happening
③ size of particular weights

Date : ____/____/____

Algorithms:

(I) SGD

(II) Adam

(III) Adadelta

(IV) Adagrad

(V) RMSProp

```
optimizer = tf.keras.optimizer.SGD()
```

Mini-batch:

- Backpropagation is very expensive to compute

Now we taking a single point instead of all and compute the gradient, then in next iteration, we take a different point.

Date :...../...../.....

Stochastic Gradient Descent:

Algorithm:

1. Initialize weights randomly
2. Loop until converge
3. Pick a single point i
4. Comput gradient, $\frac{\partial J_i(w)}{\partial w} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(w)}{\partial w}$ (B: batch)
5. Update weights, $w \leftarrow w - \eta \frac{\partial j(w)}{\partial w}$
6. Return weights.

Downsides:

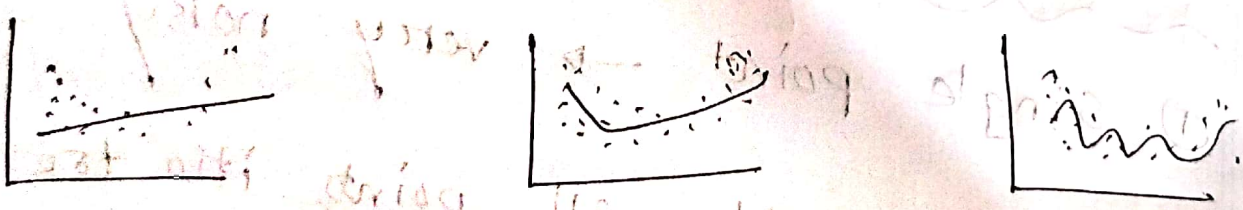
(I) single point → very noisy.

(II) If we consider all points, it's too expensive.

Solution: Take a Mini-batch

- more accurate estimations of gradient.
- smoother convergence
- allow for ~~teal~~ larger learning rates
- Mini-batches leads to fast training
- can parallelize computations + achieve sigficant speed increases on GPU's

Overfitting: (generalization in ML)



Underbitting

- model does not have capacity to fully learn data

Ideal fit

Overfit

Too complex.

Date : ____/____/____

Regularization:

— it is a technique that constrain our optimization problem to discourage complex models..

— it improves generalization of our model on unseen data.

→ Dropout :

— During training, some of the activations are set to zero. (randomly).

tf. keras. layers. Dropout (p = 0.5)

— forced network to not rely on any 1 or node.

Date :____/____/____

Early stopping:

— stop training before we have a chance to overfit.

Thanks