Backpropagation

An algorithm that trains a neural network.

weights / bias : optimum value is found for it

Epoch: An epoch simply means the model has seen every sample in your dataset once

One complete pass through all training data

Batch

epochs = 5

If there are 4 rows

1 epoch : It sees all 4 rows at once (parallel)

5 epoch : 20 times , Batch = 4

Here batch size = 1

for i in range (epochs): (5)

for j in range (X.shape[0]): (4)

select 1 row (random)

Predict (using forward prop)

Calc^c loss (mse)

Update weights $= w_0 - \eta \frac{\partial L}{\partial w}$

Calculate avg loss for every epoch

Learning = minimizing mistakes

If changing a weight reduces error, change it in that direction

⇒ Loss function is a function of all trainable parameters

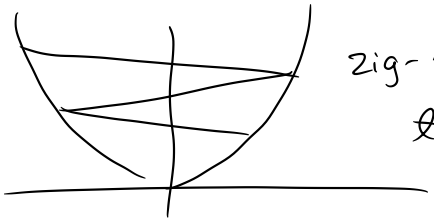
→ Concept of gradient:

↳ fancy word for derivative

Y = f(x) → dependent

$\frac{dy}{dx} = 2$ So, upon doing (1) change in x would bring 2 change in y

$w_n = w_n - \eta \frac{\partial L}{\partial w}$



zig-zag motion

that's why keep learning rate

We don't do $\frac{\partial L}{\partial w} = 0$ because neural networks contain millions of parameters

They contain nonlinear activations (ReLU, sigmoid, tanh)

No analytic solution exists for them

$w \leftarrow w - \eta \nabla L(w)$

What is convergence?

We keep on updating our weights till

$$w_n' = w_n - \eta \frac{\partial L}{\partial w_n} \approx 0$$

$\frac{\partial L}{\partial w_n} \approx 0$ means $w_n' \approx w_n$

we have hit a flat surface.

For Regression problems, output: linear activation

Why do we need weights & bias ??

Neural networks learn a function that maps input & output

We don't know the fn explicitly, so the network learns it from data

Weighted sum:

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 - \cdots$$

Weights represent importance of inputs

large weight → input matters a lot

Add bias:

Bias shift the decision.

→ Controls when the neuron activates

→ Like an adjustable threshold.

Activation:

Without it, the neural networks become just a linear model

$$y = w_2 (w_1 x) = (w_2 w_1) x$$

add 1) Non-linearity

2) Ability to learn complex functions.

Multiple layers

Each layer learns features of increasing complexity

Is this info important enough to react

pass nothing?

pass little?

pass +vely or -vely

pass strongly ?

Memoization: optimization technique from Dynamic programming (DP) that speeds up computations by caching techniques from Dynamic Programming (DP) that speeds

let's say Fibonacci Recursion

$$f(n) = f(n-1) + f(n-2)$$

Stores outputs of f^ns with specific inputs (e.g., f(x)) in a table

There are 3 variants of gradient descent, which differ in how much data we use to compute the gradient of the objective f^n.

Trade off b/n time & accuracy

Batch GD (Vanilla GD)

entire dataset update

50 pts → predict

5 epochs

50 predictions (compare it with y)

```
epoch = 10
  for in range(10):
      y_hat = np.dot(X,w) + b.
```
↓ 50 values.

y = 50 values

w,b update

10 times update

no. of epochs = no. of updates

Stochastic GD (SGD)

```
epoch = 10
for in range(10):
    shuffle
    for i in range(x.shape(0)
```
↳ 1 random pt.

↳ y_hat → forward

10 × 50 = 500

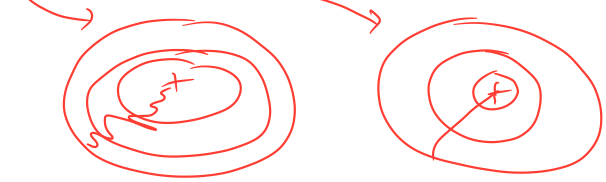
frequency of weight update is higher

Which is faster (given same no. of epochs): Batch

Which is faster to converge (given same no. of epochs: SGD

Stochastic : We keep pt in random and do

Batch : We take the whole dataset into account.



The spiky behaviour has good as well bad effects

Batch might fall into local minima

sgd might jump out of it

Vectorization in Batch

np.dot(X, W) + b

(For small datasets its very fast)

(x) → dataset of 10 crore

problem when the dataset is very large.

BGD ↔ SGD:

320 rows

Batch size = 32

Mini-batch Gradient Descent

No. of batches = 10 updates

no of batches = no. of updates = $\frac{\text{no. of rows}}{\text{Batch size}}$

fast : bgd > mbgd > sgd

conv : bgd < mbgd < sgd

Why is batch_size is provided in multipe of (2)?

Optimization technique 4, 8, 32, 128

RAM takes binary values

faster

What if batch_size doesn't divide #no. rows properly?

Rows = 400 Batch size = 150 So, 3 updates per epoch.

B1 : 150, B2 : 150 B3 : 100

Vanishing Gradient Problem

$\frac{d(\text{sigmoid})}{dx}$ = 0-0.5

The neural network might stop training.

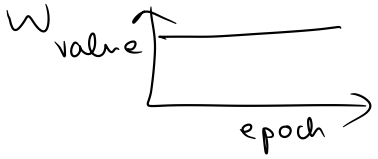
Major reasons why NNs failed in 80

$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial o_{11}} \cdot \frac{\partial o_{11}}{\partial w_{11}}$

How to recognize:

1) focus on loss: epoch → no changes → VGP

2) weights → graph values



How to handle Vanishing Gradient Problem?

1) Reduce model complexity. Not applicable because we want the model to identify complex patterns.
2) Using RELU → derivative → 0 or 1
   dying relu: relu → 0
   ↳ then makes it zero.
3) Proper Weight Initialization (GLORAT, XAVIER)
4) Batch Normalization
5) Residual Network

Sigmoid, tanh leads to Vanishing gradient problem.

Solved by ReLU, leaky ReLu

Exploding Gradient Problem

RNNs $10, 10, 10, 10 = 10^4$ linear or exponential

$\frac{\partial L}{\partial w} = d_1 \cdot d_1 \cdot d_2 \cdot d_3$ tanh

$w_h = w_o - \eta \frac{\partial L}{\partial w}$

How to improve a neural network??

1) Fine tuning NN hyper parameters:
   - No. of hidden layers
   - no. of neurons per layer
   - Learning rate
   - Optimizer
   - Batch Size.
   - Activation function
   - no. of epochs.
2) By solving problems:
   - Vanishing gradients / Exploding
   - Not enough data
   - Slow training
   - Overfitting

Fine tuning Hyperparameters:

1) No. of hidden layer

Multiple hidden layers with fewer neurons > 1 hidden layer with large no. of neurons.

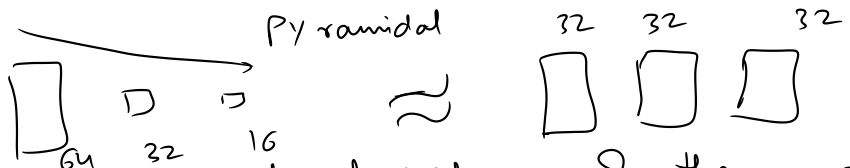
Representation learning. } Hierarchy learning

Initial layers capture primitive features.

Transfer learning

Stop adding layers till overfitting starts.

2) Neuron / Layer



primitive features are captured early on & they are more in number.

Start with more & then reduce it based on overfitting

3) Batch Size

Batch Stochastic Min-Batch

Smaller

better results

large

training faster

learning rate scheduler (Start with small value & then raise it)

(warming the learning rate)

4) Epochs : → 100 → 500 → 1000

Early Stopping

Keras → Stabilize

Keras

↓

call back.

Problems with Neural Networks

- Vanishing & exploding gradients
  - ↳ Weight init
  - ↳ Activation fn
  - ↳ Batch Norm
  - ↳ Gradient clipping
- Not Enough data
  - ↳ Transfer learning
  - ↳ Unsupervised pretrained
- Slow training
  - Optimizers
  - learning rate scheduler
- Overfitting
  - l_1 & l_2 reg
  - Dropouts

Early Stopping

monitor = "val_loss"

min_delta = 0.0001

:

Tells where performance on validation set reduces

To what epochs we should continue

Normalizing Inputs

(Age) x_1 , x_2 (salary)

w_1 w_2 Magn is high

$$w_2 = w_2 - \eta \frac{\partial L}{\partial w_2}$$