Chapter - 3 :- Neural Network Optimizers.

3.1 Optimizers & Momentum.

we have considered approaches to gradient descent that vary the number of data points involved in a step.

However, they all have used the standard update formula:

$$\omega := \omega - \alpha . \nabla J$$

There are several variants to updating the weights which give better performance in practice.

These successive "tweaks" each attempt to improve the previous idea.

The resulting methods are reffered to as "optimizers".

Momentum

Only change direction by a little bit each time

Keeps a 'running average' of step directions, smoothing but out the variations of individual points.

$$V_t := \eta \cdot V_{t-1} - \alpha \cdot \nabla J$$

$$W := W - V_t$$

Here η is reffered as momentum

More the momentum more the smoothening.

It is generally given "<1"

Nesterov Momentum:-

Idea: control overshooting by looking head.

Apply gradient only to the non-momentum component

$$v_t = \eta \cdot v_{t-1} - \alpha \nabla (J - \eta \cdot v_{t-1})$$

$$W := W - v_t$$

3.2.1 AdaGrad

Idea: scale the update for each weight seperately

1. Update frequently- updated weights less
2. Keep running sum of previous updates
3. Divide new update by factor of previous sum

↓

Scalling down the new update as it should less than previous one with starting $G_i(0) = 0$

$$G_i(t) = G_i(t-1) + \left(\frac{\partial L}{\partial W_i}(t)\right)^2$$ G will continue to increase

$$w := w - \frac{\eta}{\sqrt{G_t} + \epsilon} \cdot \nabla J$$ This leads to smaller update each iteration.

3.2.2 RMSProp

Quite similar to AdaGrad.

- Rather than using the sum of previous gradients, decay older gradients more than more recent ones
- More adaptive to recent updates

3.2.3 ADAM

Idea: use both first-order & second order change info & decay both over time

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J$$

$$V_t = \beta_2 V_{t-1} + (1-\beta_2)\nabla J$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

(bias)

$$\hat{V}_t = \frac{V_t}{1-\beta_2^t}$$

$$w := w - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t.$$

(Adam)

3.3 Details of Training Neural Networks

Given an example

we know how to compute the derivative for each weight

1. How exactly do we update the weights?

2. How often ? (after each traing data point?
   after all training data points?)

Gradient Descent:-

classical approach:- get derivative of entire data set & then take a step in that direction

- $W_{new} = W_{old} - \alpha * \text{derivative}$

Pros - Each step is informed by all data

Cons - very slow.

Stochastic Gradient Descent:-

Get derivative for just one point, & take a step in that direction

- Steps are less informed, but you take more
- should balance out the mistakes
- probably want a smaller step size
- Also helps regularise

Compromise approach: Mini-batch

Get derivative as a for a small set of points & then take a step in that direction

- Typical mini batch sizes are 16, 32
- strikes a balance between 2 extremes

Comparison of Batching Approaches:-

SGD — Mini Batch — Full batch GD

Fast, less accurate step

slower, more accurate step

Full Batch Terminology:-

Full-Batch:- use entire data set to compute Gradient before updating

Mini-Batch:- use a smaller portion of data, to compute gradient before updating

SGD:- use a single example to compute gradient before updating

Epochs:- refers to a single pass through all of training data

- In full batch gradient, there would be one step taken per epoch
- In SGD learning, there would be n steps per epochs (n = training set size)

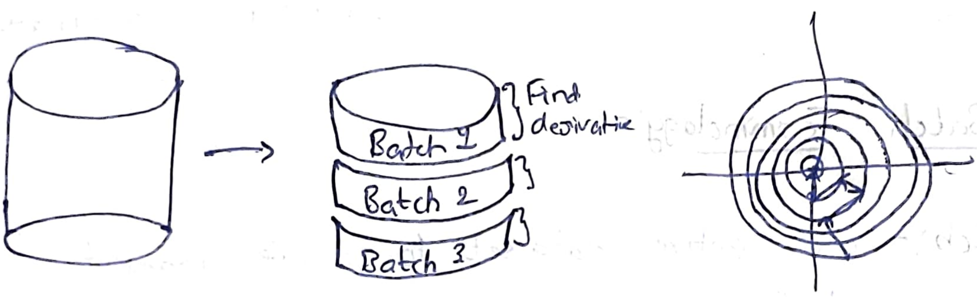
- In Minibatch, there would be n/batch size steps taken per epoch
- when training, we often refer to the number of epochs needed for the model to be trained

3.4 Data Shuffling

To avoid any cyclical movement & aid covergence it is recommended to shuffle the data after each epoch

This way, the data is not seen in the same order every time, & the batches are not the exact same ones.

Training in Action:



First epoch complete

3.5 Transforms

Scaling Inputs

In our old discussions of backpropagation we briefly touched on the formula for the gradient used to update the values of our weights w:

$$\frac{\partial J}{\partial w^{(i)}} = (\hat{y} - y) \cdot a^{(i)}$$

And at each iteration of gradient Descent

$$w_{new} = w_{old} - \alpha \cdot \text{derivative}$$

when $i = 0$, we are using the input values X as part of derivative to update w_{new}

This means if we donot normalize the input values. those with higher values will update much more quickly than those with lower values.

This imbalance can greatly slow down the speed at which our model converges

Ways to Scale Inputs:-

① Linear scaling to [0,1]: Min-Max scaler = $x_i = \dfrac{x_i - x_{min}}{x_{max} - x_{min}}$

② Linear scaling to [-1,1]: Standard scaler : $x_i = 2\left(\dfrac{x_i - x_{min}}{x_{max_i} - x_{min}}\right) - 1$