PES UNIVERSITY ONLINE

Machine Intelligence

Nishtha Varshney
Department of Computer Science & Engineering

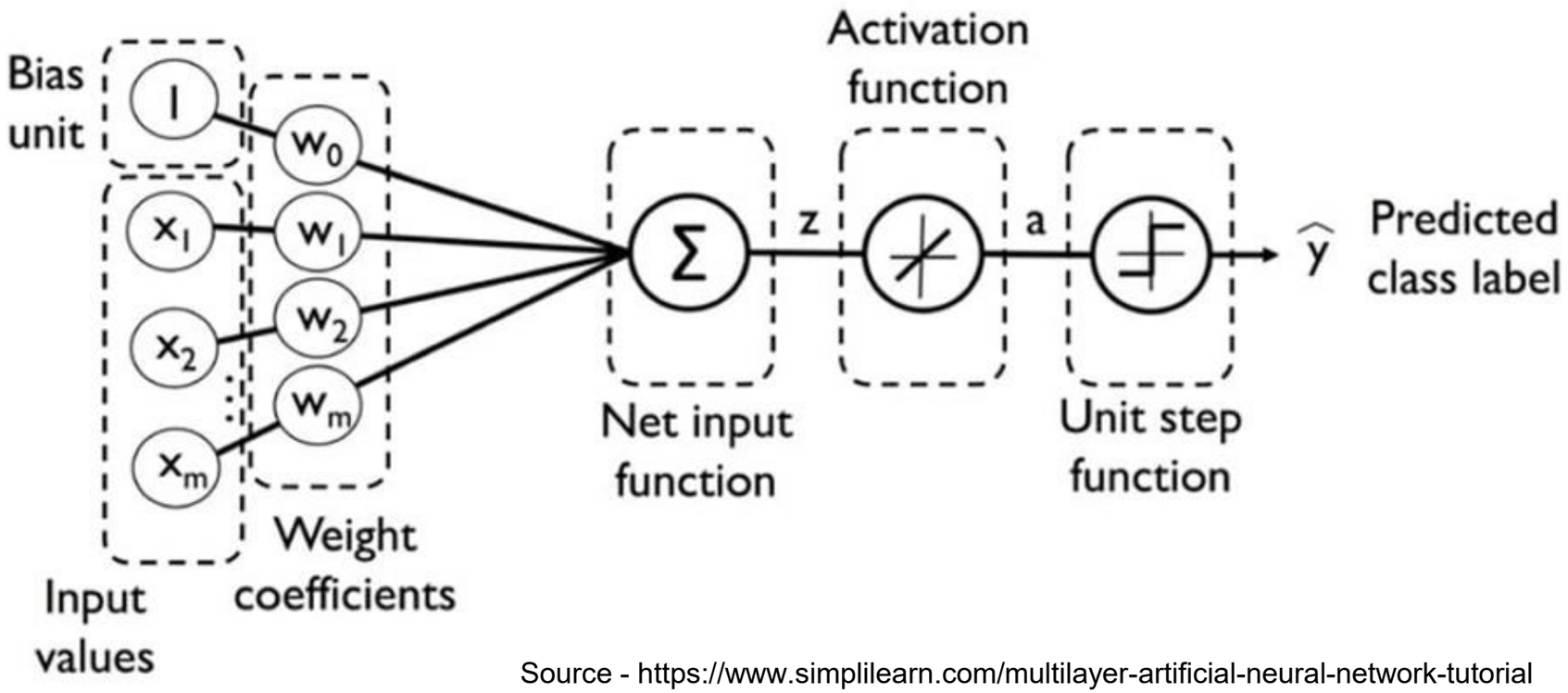
UE19CS303-Machine Intelligence

Disclaimer

These slides are prepared from various resources from Internet and universities from India and Abroad and with the help of TAs.

Machine Intelligence

Single-layer Neural Network - A RECAP



Source - https://www.simplilearn.com/multilayer-artificial-neural-network-tutorial

- A single-layer neural network represents the most simple form of neural network, in which there is only one layer of input nodes that send weighted inputs to a subsequent layer of receiving nodes
- Weights are updated based on a unit function in perceptron rule or on a linear function in Adaline Rule.

Machine Intelligence

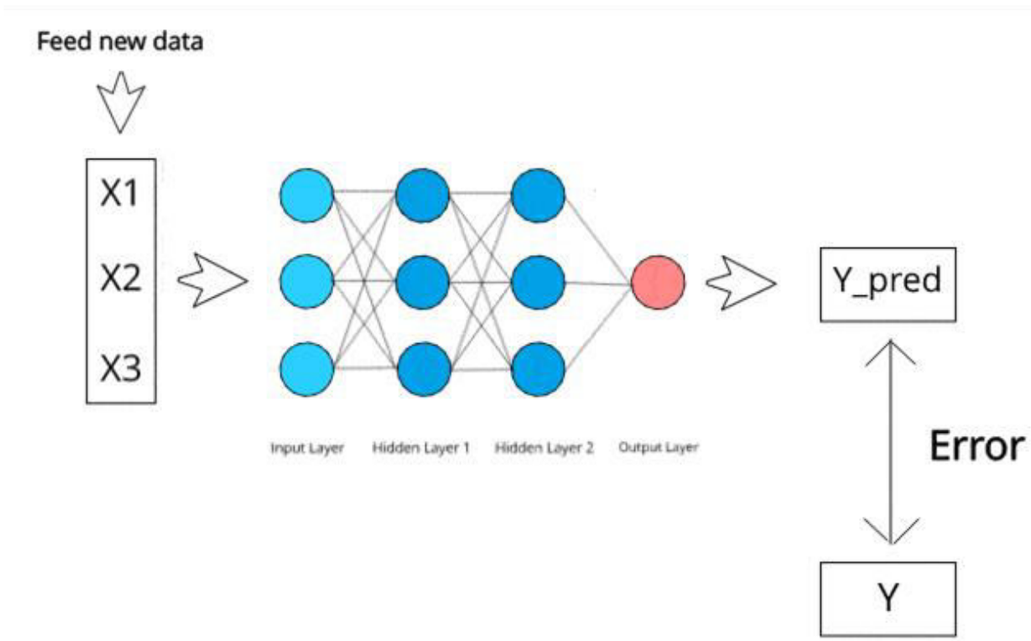
Multi-layer Neural Network - A RECAP

A fully connected multi-layer neural network is called a Multilayer Perceptron(MLP)

The MLP learning procedure is as follows :

- Starting with the input layer, propagate data forward to the hidden layers and output layer. This step is the forward propagation.
- Based on the output, calculate the error (the difference between the predicted and known outcome). The error needs to be minimized.
- Backpropagate the error. Find its derivative with respect to each weight in the network, and update the model.

Repeat the three steps given above over multiple epochs to learn ideal weights.
Finally, the output is taken via a threshold function to obtain the predicted class labels



Source - https://www.kdnuggets.com/2019/10/introduction-artificial-neural-networks.html

Machine Intelligence

Optimizers in Neural Networks

- Optimizers are algorithms or methods used to update the parameters of the neural network such as weights and learning rate to reduce the losses.
- Solve optimization problems by minimizing or maximising the function.

How do Optimizers work?

- Responsible for reducing the losses and providing the most accurate results possible.
- The weight is initialized using some initialization strategies and is updated with each epoch according to the update equation

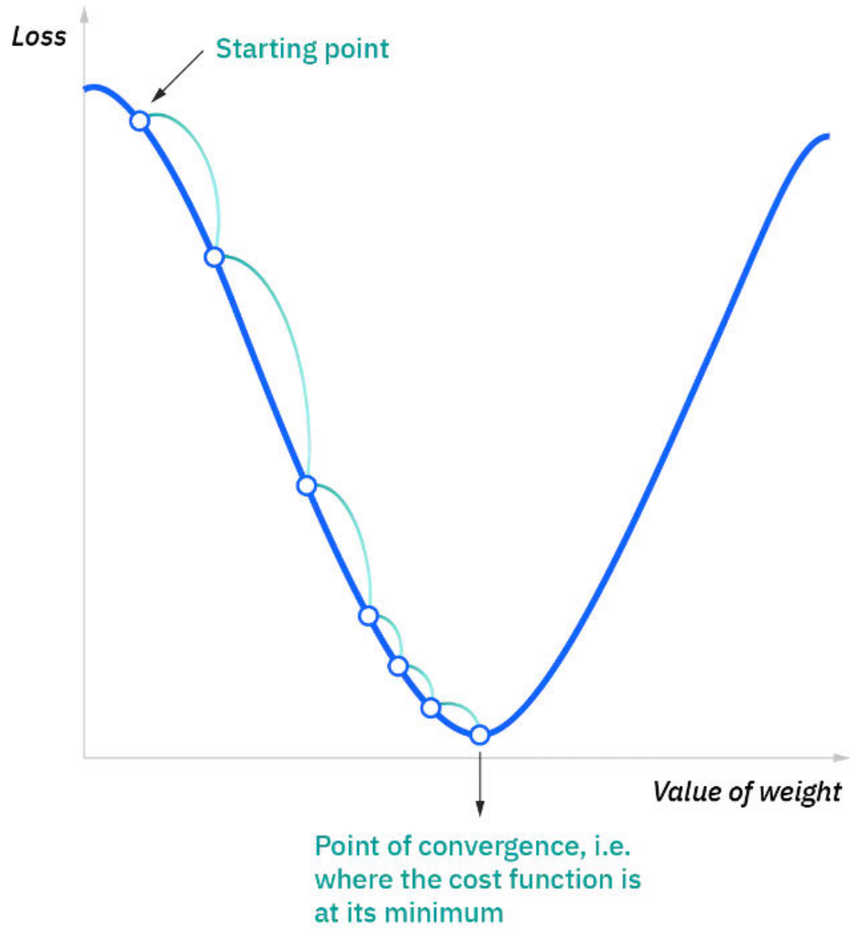
$$W_{new} = W_{old} - lr * (\nabla_W L)_{W_{old}}$$

Machine Intelligence

How optimizers work ??

- Let’s think of a hiker trying to get down a mountain with a blindfold on. It’s impossible to know which direction to go in, but there’s one thing she can know: if she’s going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she’ll reach the base.

- Similarly, it’s impossible to know what your model’s weights should be right from the start. But with some trial and error based on the loss function you can end up getting there eventually.
- How you should change your weights or learning rates of your neural network to reduce the losses is defined by the **optimizers** you use.



Source - https://www.ibm.com/cloud/learn/gradient-descent

Optimizers

We'll learn about different types of optimizers and how they exactly work to minimize or maximize the loss function.

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini Batch Stochastic Gradient Descent (MB-SGD)
4. SGD with momentum
5. Adaptive Gradient (AdaGrad)
6. RMSProp
7. Adam

Machine Intelligence

Gradient Descent (GD) - RECAP

- Gradient descent is the most basic and first-order optimization algorithm which is dependent on the first-order derivative of a loss function.
- We start with a random point on the function and move in the negative direction of the gradient of the function to reach the local/global minima.

What is Gradient?

"A gradient measures how much the output of a function changes if you change the inputs a little bit."

Algorithm 1: gradient_descent()

$t \leftarrow 0;$
$max\_iterations \leftarrow 1000;$
while $t < max\_iterations$ **do**
  $w_{t+1} \leftarrow w_t - \eta \nabla w_t;$
  $b_{t+1} \leftarrow b_t - \eta \nabla b_t;$
end

Source - http://www.cse.iitm.ac.in/~miteshk/CS7015/Slides/Teaching/pdf/Lecture5.pdf

Machine Intelligence

Gradient Descent (GD) - RECAP

Learning rate

Gradient of the loss wrt weight parameter

$$w_{new} = w_{old} - \eta \frac{\partial \mathcal{L}}{\partial w}$$

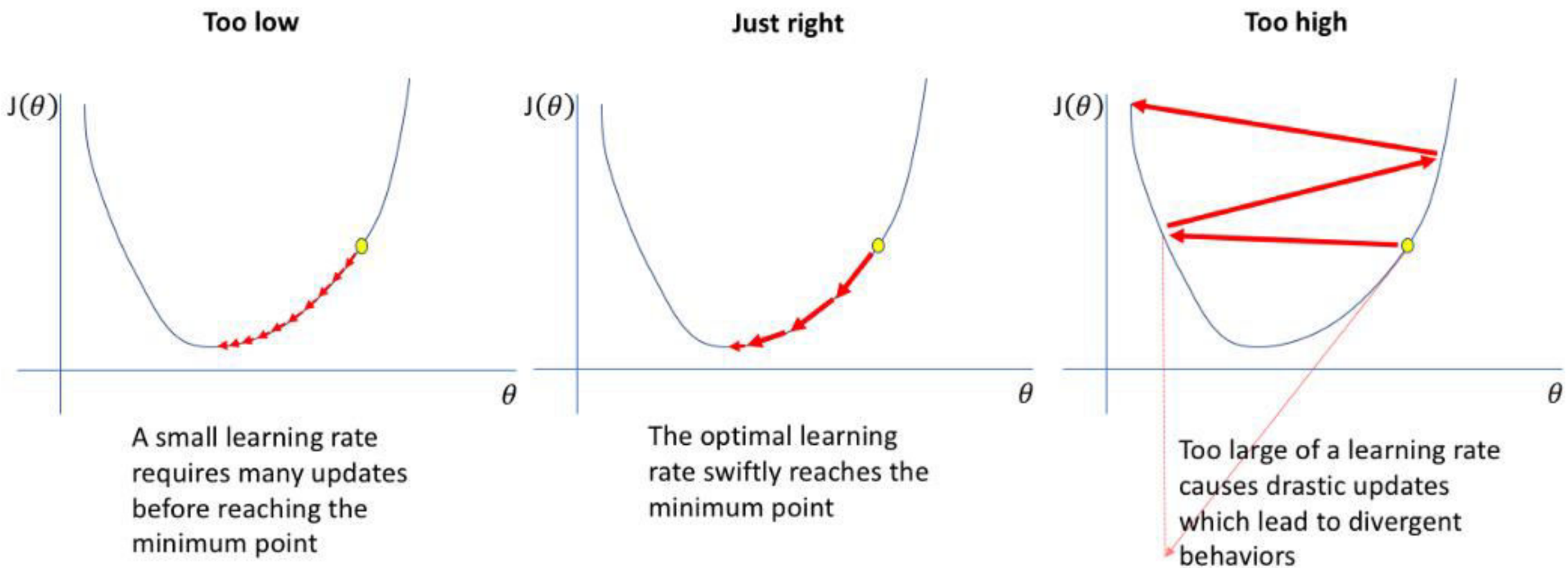
We update parameters in the negative gradient direction to minimize the loss.

Note: Gradient of Loss Function wrt model parameters (w)

Machine Intelligence

Gradient Descent (GD) - Importance of Learning Rate

- The steps gradient descent takes into the direction of the local minimum, are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.



Note: We do not know the optimal learning rate!

Source - https://medium.com/mlearning-ai/optimizers-in-deep-learning-7bf81fed78a0

Gradient Descent (GD)

Advantages:

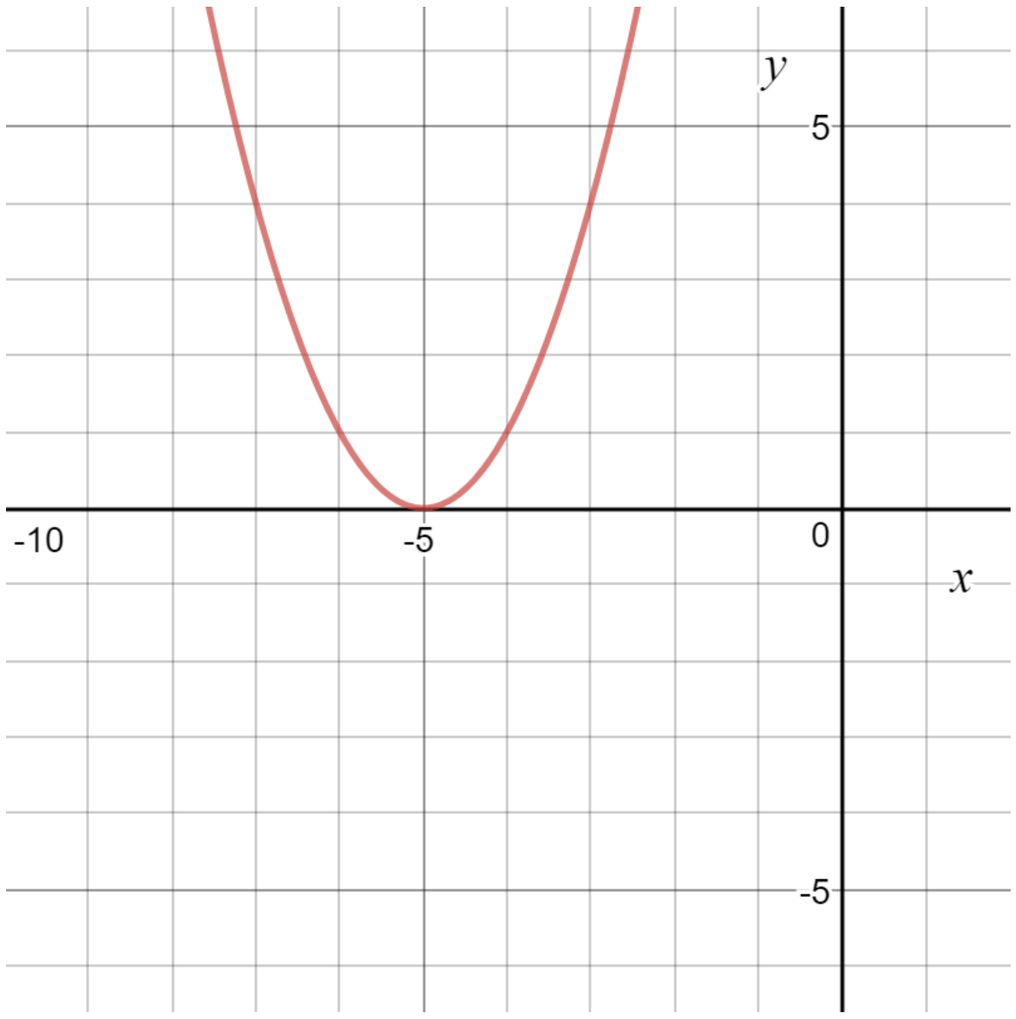
- Easy computation.
- Easy to implement.
- Easy to understand.

Disadvantages:

- May get trapped at local minima.
- Weights are changed after calculating the gradient on the whole dataset. So, *if the dataset is too large then this may take years to converge to the minima*.
- Requires large memory to calculate the gradient on the whole dataset.

Gradient Descent (GD) - Example by hand

Question : Find the local minima of the function $y=(x+5)^2$ starting from the point x=3



- By looking at the graph $y = (x+5)^2$ we can say it reaches its minimum value when x = -5
- Hence x=-5 is the local and global minima of the function.

Now, let's see how to obtain the same numerically using **gradient descent and python**

Machine Intelligence

Gradient Descent (GD) - Example by hand

Question : Find the local minima of the function $y=(x+5)^2$ starting from the point x=3

$$X_0 = 3$$

$$Learning\ rate = 0.01$$

$$\frac{dy}{dx} = \frac{d}{dx}(x+5)^2 = 2 * (x+5)$$

Note: Gradients based on first-order derivatives may be approximate! All the improvements to GD / SGD that we discuss later, make use of, in some sense, second-order derivatives (curvatures)!

```
# The algorithm starts at x=3
cur_x = 3
# Learning rate
rate = 0.01
#This tells us when to stop the algorithm
precision = 0.000001
previous_step_size = 1
# maximum number of iterations
max_iters = 10000
#iteration counter
iters = 0
#Gradient of our function
df = lambda x: 2*(x+5)
```

Machine Intelligence

Gradient Descent (GD) - Example by hand

Question : Find the local minima of the function $y=(x+5)^2$ starting from the point x=3

$$X_1 = X_0 - (learning\ rate) * (\frac{dy}{dx})$$

$$X_1 = 3 - (0.01) * \big(2 * (3 + 5)\big) = 2.84$$

Stop loop when difference between x values from 2 consecutive iterations is less than 0.000001 or when number of iterations exceeds 10,000

Step 2 : Loop to perform gradient descent

```
while previous_step_size > precision and iters < max_iters:
    #Store current x value in prev_x
    prev_x = cur_x
    #Grad descent
    cur_x = cur_x - rate * df(prev_x)
    #Change in x
    previous_step_size = abs(cur_x - prev_x)
    #iteration count
    iters = iters+1
    print("Iteration",iters,"\nX value is",cur_x)

print("The local minimum occurs at", cur_x)
```

Stochastic Gradient Descent (SGD)

- It's a changed version of the GD method, where the model parameters are updated on every iteration.
- That is - after every training sample, the loss function is tested and the model is updated.
- These frequent updates result in converging to the minima in less time, but it comes at the cost of **increased variance** - that can make the model overshoot the required position.

Randomly shuffle (reorder) training examples

Repeat {
  for $i := 1, \ldots, m$ {
    $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
    (for every $j = 0, \ldots, n$)
  }
}

Source - https://www.hackerearth.com/blog/developers/3-types-gradient-descent-algorithms-small-large-data-sets/

Machine Intelligence

Stochastic Gradient Descent (SGD) vs Gradient Descent (GD)

- Batch gradient descent refers to calculating the derivative from all training data before calculating an update.
- Stochastic gradient descent refers to calculating the derivative from each training data instance and calculating the update immediately.

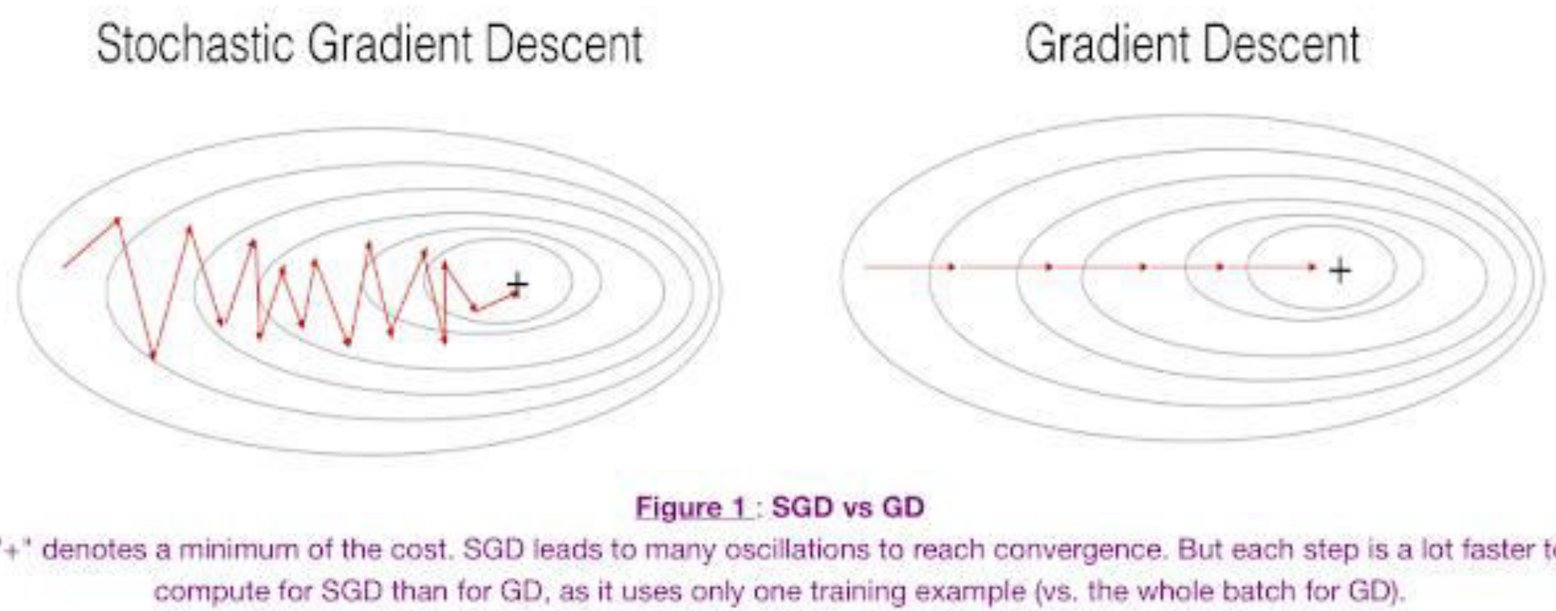


Figure 1 : SGD vs GD

"+" denotes a minimum of the cost. SGD leads to many oscillations to reach convergence. But each step is a lot faster to compute for SGD than for GD, as it uses only one training example (vs. the whole batch for GD).

1. On the left, we have Stochastic Gradient Descent (where m=1 per step) we take a Gradient Descent step for each example and on the right is Batch Gradient Descent (1 step per entire training set).
2. SGD seems to be quite noisy, at the same time it is much faster but may not converge to a minimum.

Source - http://ankit-ai.blogspot.com/2018/11/optimization-algorithms-for-machine.html

Stochastic Gradient Descent (SGD)

Advantage:

Memory requirement is less compared to the GD algorithm as the derivative is computed taking only 1 point at once.

Disadvantages:

1. The time required to complete 1 epoch is large compared to the GD algorithm.
2. Takes a long time to converge.
3. May get stuck at local minima.

Machine Intelligence

Mini Batch Stochastic Gradient Descent (MB-SGD)

- MB-SGD algorithm is an extension of the SGD algorithm and it overcomes the problem of large time complexity
- In every iteration we use a set of 'm' training examples called batch to compute the gradient of the cost function.
- In this way, algorithm
  - reduces the variance of the parameter updates,
  - makes computing of gradient very efficient.

Machine Intelligence

Mini Batch Stochastic Gradient Descent (MB-SGD)

- Assume that the number of samples in the training set is 1000, then each mini-batch is only a subset of it. Assume that each mini-batch contains 10 samples, so that the entire training dataset can be divided into 100 mini-batches
- The pseudo code is:

Say $b = 10, m = 1000.$

Repeat {

for $i = 1, 11, 21, 31, \ldots, 991$ {

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

(for every $j = 0, \ldots, n$) } }

where ‘ b ‘ is number of batches and ‘ m ‘ is number training examples.

Source - https://www.hackerearth.com/blog/developers/3-types-gradient-descent-algorithms-small-large-data-sets/

Mini Batch Stochastic Gradient Descent (MB-SGD)

Advantages:

Less time complexity to converge compared to standard SGD algorithm.

Disadvantages:

- The update of MB-SGD is much noisy compared to the update of the GD algorithm.
- Take a longer time to converge than the GD algorithm.
- May get stuck at local minima.

Machine Intelligence

SGD with Momentum

Stochastic gradient descent with momentum remembers the update Δw at each iteration, and determines the next update as a linear combination of the gradient and the previous update:

- SGD is a popular optimization strategy. But it can be slow.
- Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations
- It does this by adding a fraction γ (momentum) of the update vector of the past time step to the current update vector -

$$v_t = \gamma v_{t-1} + \eta \nabla J(\theta; x, y)$$

$$\theta = \theta - v_t$$

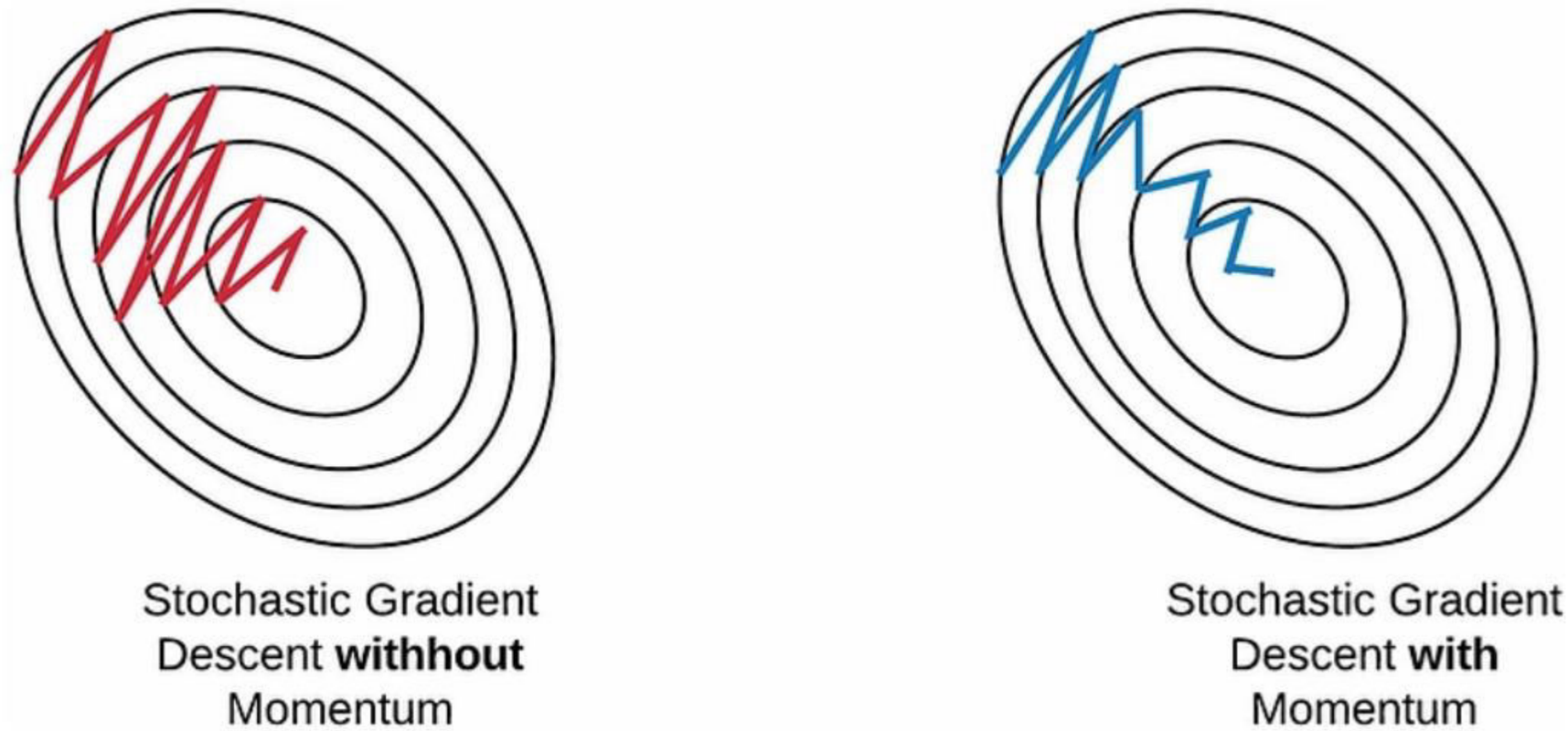
Momentum Gradient descent takes gradient of previous time steps into consideration

Momentum at time 't' is computed using all previous updates giving more weightage to recent updates compared to the previous update.

This speeds up the convergence.

Machine Intelligence

SGD with Momentum



SGD with momentum denoises the gradients and converges faster as compared to SGD.

Source - https://cedar.buffalo.edu/~srihari/CSE676/8.3%20BasicOptimizn.pdf

Advantages:

1. Has all advantages of the SGD algorithm.
2. Converges faster than the GD algorithm.

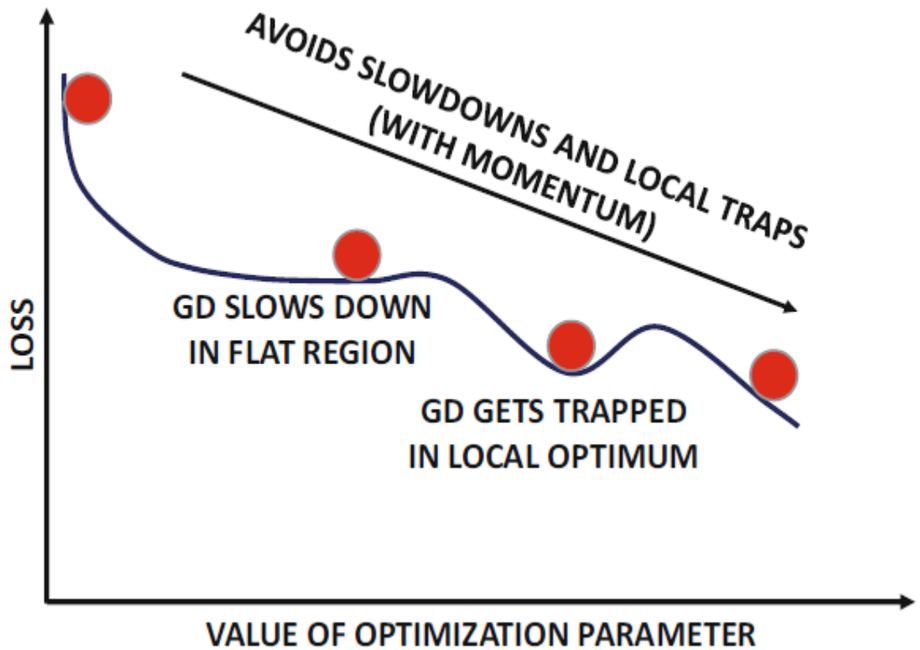
Disadvantages:

We need to compute one more variable for each update.

Machine Intelligence

Discussion: GD with Momentum

• Momentum generally helps the GD to negotiate well flat regions and local optima on the loss surface.
•An easy - to - visualize analogy is that of a marble rolling down a slope that is able to overcome small potholes and other local distortions because of its momentum. (see the following figure)



(Source: "Linear Algebra and Optimization for Machine Learning" by Charu C. Aggarwal)

• Medium term to long term trends of consistent directions from the past need to be used, weighing the recent values more. In practice, we use just the previous value. (This is fine as it is computed using the same principle and thus it does represent the history)
• The momentum often results in "overshooting" the optimal point!!
• Still, by tweaking the momentum parameter, we often get performance that is far superior to the simple GD method.

Machine Intelligence

AdaGrad

- For all the previously discussed algorithms the **learning rate remains constant.**
- So the key idea of AdaGrad is to have an **adaptive learning rate** for each of the weights.
- It performs **smaller updates** for parameters associated with frequently occurring features, and **larger updates** for parameters associated with infrequently occurring features.
- In its update rule, **Adagrad modifies the general learning rate η** at each time step t for every parameter θt based on the past gradients for θt:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \cdot g_t$$

G_t is sum of the squares of the past gradients w.r.t. to all parameters θ

All parameters?? NO.

AdaGrad

- The benefit of AdaGrad is that it eliminates manually tuning of the learning rate.

- Its main weakness is the accumulation of the squared gradients(G_t) in the denominator.

- Since every added term is positive, the accumulated sum keeps growing during training, causing the learning rate to shrink and becoming infinitesimally small and further resulting in a vanishing gradient problem.

Discussion - AdaGrad

- Adaptive Learning Rate for each parameter individually over the period of training.
- Let J be the Loss Function.
- Let w_i be one specific parameter and A_i be the aggregated squared value of the partial derivative of J wrt this particular parameter w_i
- This aggregate value is updated in each iteration as follows:

$$A_i \leftarrow A_i + \left(\partial J / \partial w_i \right)^2$$

- Now, the parameter w_i is updated as follows:

$$w_i \leftarrow w_i - \left(\alpha / \sqrt{A_i} \right) \left(\partial J / \partial w_i \right)$$

(A small positive constant ϵ can be added to $\sqrt{A_i}$ for numerical stability)

Discussion - AdaGrad

•The learning parameter is scaled based on the aggregate value A_i

• If the parameter is varying slowly (the Loss Surface wrt this parameter is relatively flat / has gentle slope),

$\sqrt{A_i}$ will be relatively small and thus larger update in the parameter occurs. This would be fine.

• On the other hand, if the parameter is varying fast (the Loss Surface wrt this parameter is relatively bumpy

/ has steep slope), $\sqrt{A_i}$ will be relatively high and thus smaller update in the parameter occurs. Again, this is appropriate to ensure that the optimal point is not missed!

The above update process is carried out independently for each parameter!!

•One additional problem is that, because of aggregation, very old and quite "stale" data continues to influence the present!

RMSProp

- RMSProp is Root Mean Square Propagation
- Resolves Adagrad's radically diminishing learning rates by using a moving average of the squared gradient.
- Utilizes the magnitude of the recent gradient descents to normalize the gradient.
- Learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.
- Divides the learning rate by the average of the exponential decay of squared gradients (taking care of the undesirable influence of "stale" data)

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{(1-\gamma)g^2{}_{t-1} + \gamma g_t + \varepsilon}} \cdot g_t$$

Implementation does not require storing the past gradients

γ is the decay term that takes value from 0 to 1. gt is moving average of squared gradients

Discussion : Adam - Adaptive Moment Estimation

- Adam (ADaptive Moment Estimation) can be considered as a combination of AdaGrad and RMSProp.

- It maintains an exponential average (using a decay factor) of the squared gradients as in RMSProp.

- Additionally, it maintains exponential average (using another decay factor) of the gradients also incorporating momentum).

- It uses the former for scaling the learning rate and the later in the place of regular gradient!
-

Discussion : Adam - Adaptive Moment Estimation

- Further, it addresses the issue of bias towards 0 in the initial iterations. (As gradients are initialized to 0 in the beginning, we get bias towards 0 in early iterations.) Adam uses bias-correction factors to address this issue.

• Because of these advantages Adam is perhaps the most popular variant of the gradient-based algorithms.

Adam - Adaptive Moment Estimation

- Adam can be viewed as a combination of Adagrad, which works well on sparse gradients and RMSprop which works well in online and nonstationary settings.
- Reduces the radically diminishing learning rates of Adagrad
- Adam implements the exponential moving average of the gradients to scale the learning rate
- It keeps an exponentially decaying average of past gradients

Algorithm: Generalized Adam

S0. Initialize $m_0 = 0$ and x_1

For $t = 1, \dots, T$, do

S1. $m_t = \beta_{1,t} m_{t-1} + (1 - \beta_{1,t}) g_t$

S2. $\hat{v}_t = h_t(g_1, g_2, \dots, g_t)$

S3. $x_{t+1} = x_t - \frac{\alpha_t m_t}{\sqrt{\hat{v}_t}}$

End

Adam - Adaptive Moment Estimation

- Adam algorithm first updates the exponential moving averages of the gradient(m_t) and the squared gradient(v_t) which is the estimates of the first and second moment.
- Hyper-parameters β1, β2 ∈ [0, 1) control the exponential decay rates of these moving averages as shown below

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

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of first and second moment respectively

Machine Intelligence

Adam - Adaptive Moment Estimation

- Moving averages are initialized as 0 leading to moment estimates that are biased around 0 especially during the initial timesteps. This initialization bias can be easily counteracted resulting in bias-corrected estimates

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

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

$\hat{m}_t$ and $\hat{v}_t$ are bias corrected estimates of first and second moment respectively

- The values for β_1 and β_2 suggested based on empirical evidence are 0.9 and 0.999. These are the values used in almost all applications.
- These bias adjustment factors influence the estimates only in early iterations. In later iterations, with larger values of t, both the adjustment factors become almost 0. (This is fine as we have no more requirement for bias correction!)

- Finally, we update the parameter -

$$\theta_{t+1} = \theta_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$

Which optimizer is better?

To get a more accurate insight into the performance of the optimizers , Let's assess each of the following metrics, including the graphs, on an average of 10 runs.

- Each optimizer is configured with the default hyperparameters of TensorFlow. SGD has a learning rate of 0.01, and doesn't use momentum.

- AdaGrad has an learning rate of 0.001, an initial accumulator value of 0.1, and an epsilon value of 1e-7.

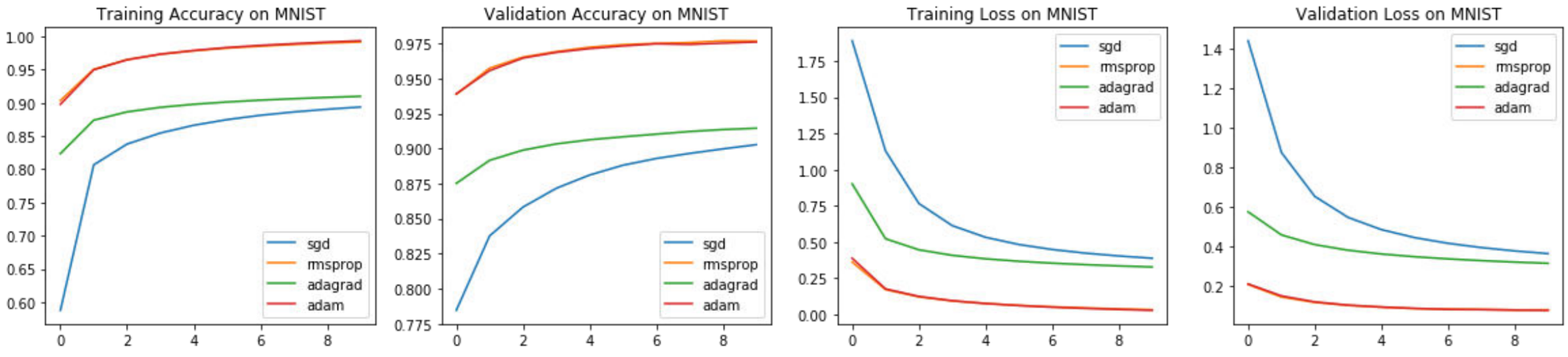
- RMSProp uses a learning rate of 0.001, rho is 0.9, no momentum and epsilon is 1e-7.

- Adam use a learning rate 0.001 as well. Adam's beta parameters were configured to 0.9 and 0.999 respectively. Finally, epsilon=1e-7

Dataset used - MNIST -
Full code - https://drive.google.com/file/d/1bJEZ0p_BbMJGejgB9I1DV3BYy5OEuF-d/view?usp=sharing

Machine Intelligence

Which algorithm is better?



Training performance on MNIST with different optimizers

Optimizer	Training accuracy	Training loss	Validation accuracy	Validation loss
SGD	89.54%	0.3798	90.27%	0.3657
AdaGrad	91.06%	0.3228	91.45%	0.3167
RMSProp	99.37%	0.0248	97.67%	0.0813
Adam	99.56%	0.0210	97.60%	0.0790

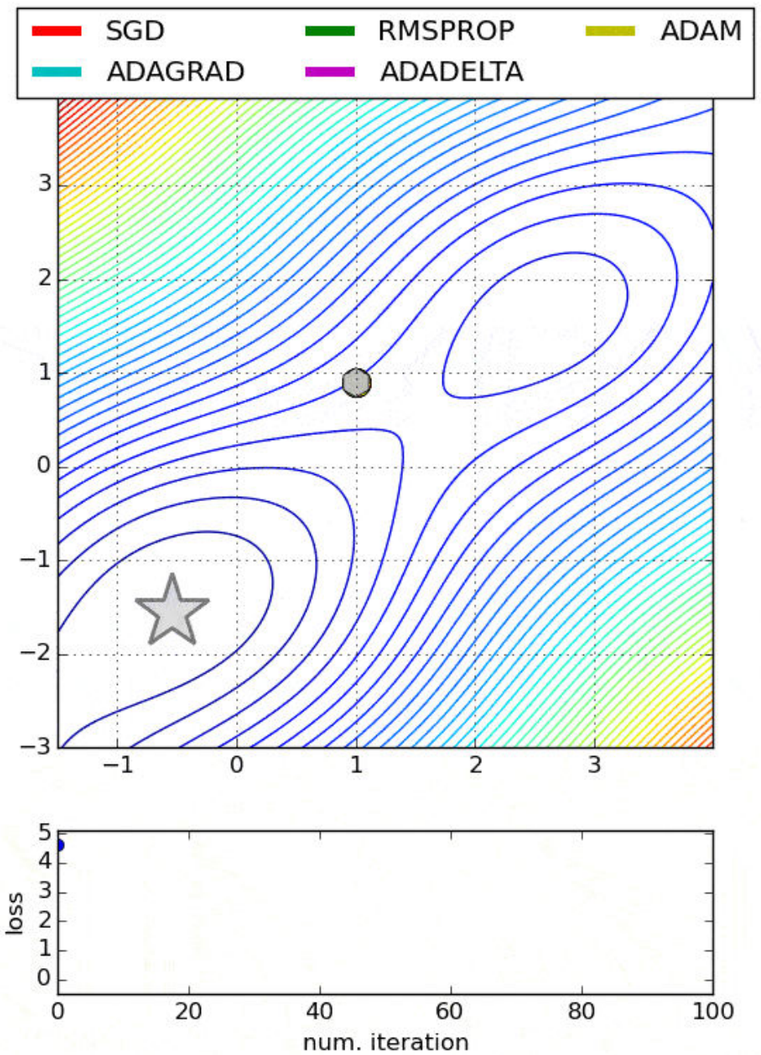
- Even though MNIST is a small dataset, and considered to be easily trained on, there were some noticeable differences in performance of the optimizers.
- **SGD was by far the worst optimizer**
- **RMSProp and Adam performed extremely similar.**
- This can be expected because their key difference is the use of gradients as momentum by Adam, and MNIST is a very small dataset and therefore it also has small gradients, too.

Source - https://heartbeat.fritz.ai/an-empirical-comparison-of-optimizers-for-machine-learning-models-b86f29957050

Machine Intelligence

Which algorithm is better?

Visualising adaptive learnin



THANK YOU

Nishtha Varshney

Department of Computer Science & Engineering

nishuvr13@gmail.com