ML - HW3 Theory

Amirhossein Izadi - 99243018

Q1.

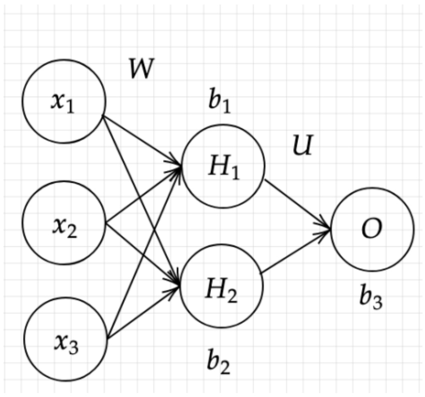
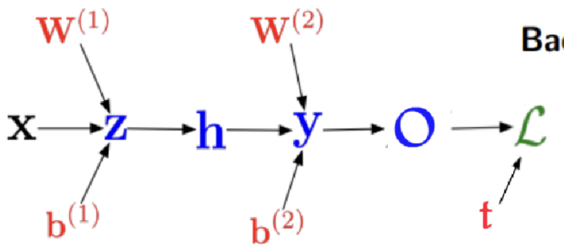


Figure 1: a neural network W represent hidden wieghts and U represents output weights



$$\begin{tikzcd}
 & W^{(1)} \arrow[d] & & W^{(2)} \arrow[d] & & \\
x \arrow[r] & z \arrow[r] & h \arrow[r] & y \arrow[r] & O \arrow[r] & \mathcal{L} \\
 & b^{(1)} \arrow[u] & & b^{(2)} \arrow[u] & & t \arrow[u]
\end{tikzcd}$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$
$$\overline{o} = \overline{\mathcal{L}}\,(\mathbf{y} - \mathbf{t})$$
$$\overline{\mathbf{y}} = \overline{o} \circ \sigma'(y)$$
$$\overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^{\top}$$
$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$
$$\overline{\mathbf{h}} = \mathbf{W}^{(2)\top}\overline{\mathbf{y}}$$
$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$
$$\overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^{\top}$$
$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

Forward pass:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{h} = \sigma(\mathbf{z})$$
$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$
$$o = \sigma(\mathrm{y})$$
$$\mathcal{L} = \frac{1}{2}\|\mathbf{t} - o\|^2$$

Figure 2: We can model our neural network via this graph and the rest function declarations

$$\beta = 0.9\ (momentum)\ ,\ \alpha = 0.9\ (learning\ rate)\ ,\ \sigma(z) = \frac{1}{1 + e^{-z}}\ (activation\ function)$$

$$X = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}\ ,\ y = [1]$$

$$W_{hidden} = \begin{bmatrix} 0.2 & 0.4 & -0.5 \\ -0.3 & 0.1 & 0.2 \end{bmatrix}\ ,\quad B_{hidden} = \begin{bmatrix} -0.4 \\ 0.2 \end{bmatrix}$$

$$z_{\ hidden} = W_{hidden}\ .\ X + B_{hidden}$$
$$a_{\ hidden} = \sigma(\ z_{\ hidden}\ )$$

$$W_{output} = \begin{bmatrix} -0.3 & -0.2 \end{bmatrix}\ ,\quad B_{output} = \begin{bmatrix} 0.1 \end{bmatrix}$$

$$z_{\ output} = W_{output}\ .\ a_{\ hidden} + B_{output}$$
$$a_{\ output} = \sigma(\ z_{output}\ )$$

Iteration 1:

(I) forward pass:

$$z_{\ hidden} = W_{hidden}\ .\ X + B_{hidden}$$

$$z_{\ hidden} = \begin{bmatrix} 0.2 & 0.4 & -0.5 \\ -0.3 & 0.1 & 0.2 \end{bmatrix} . \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -0.4 \\ 0.2 \end{bmatrix} = \begin{bmatrix} -0.7 \\ 0.1 \end{bmatrix}$$

$$a_{\ hidden} = \sigma(\ z_{\ hidden}\ ) = \sigma\left(\begin{bmatrix} -0.7 \\ 0.1 \end{bmatrix}\right) = \begin{bmatrix} \sigma(-0.7) \\ \sigma(0.1) \end{bmatrix} = \begin{bmatrix} \dfrac{1}{1 + e^{0.7}} \\ \dfrac{1}{1 + e^{-0.1}} \end{bmatrix} = \begin{bmatrix} 0.3318 \\ 0.5245 \end{bmatrix}$$

$$z_{\ output} = W_{output} \ . \ a_{\ hidden} + B_{output}$$

$$z_{\ output} = \begin{bmatrix} -0.3 & -0.2 \end{bmatrix} . \begin{bmatrix} 0.3318 \\ 0.5245 \end{bmatrix} + \begin{bmatrix} 0.1 \end{bmatrix} = \begin{bmatrix} -0.1044 \end{bmatrix}$$

$$a_{\ output} = \sigma(z_{output}) = \sigma\Big(\big[-0.10444 \big]\Big) = \big[\ \sigma(-0.10444) \ \big] = \left[\frac{1}{1 + e^{0.10444}} \right] = [\ 0.4739 \ ]$$

(II) compute costs:

According to the question although Cross Entropy is a better choice for Cost function but we must use Least square error as cost function. (Here i use both)

$$J_{LSE}(y, \hat{y}\ ) = \frac{1}{2} \sum_{i=1}^{m} (\ \hat{y}_i - y_i)^2 = \frac{1}{2} (\ \hat{y} - y)^2$$

$$J_{LSE} = \frac{1}{2} (\ 0.4739 - 1 \ )^2 = 0.1384$$

here we have only one record (m = 1)

$$J_{CE}(y, \hat{y}\ ) = -y \cdot log(\ \hat{y}\ ) - (1 - y) \cdot log(1 - \hat{y}\ )$$

$$J_{CE} = -1 \ . \ log(0.4739) - (1 - 1) \ . \ log \ (1 - 0.4739) = 0.7467$$

In stastical machine learning term log refers to natural log (ln)

(III) backward pass:

compute gradients :

$$Assume:\ \delta_{output} = \frac{\partial J_{LSE}}{\partial a_{output}} \times \frac{\partial a_{output}}{\partial z_{output}} = (a_{output} - y) \,.\, [a_{output} \circ (1 - a_{output})]$$

$$\xrightarrow{*} \frac{\partial J_{LSE}}{\partial W_{output}} = \delta_{output} \times \frac{\partial z_{output}}{\partial W_{output}} = \delta_{output} \,.\, a_{hidden}^{T}$$

$$\xrightarrow{**} \frac{\partial J_{LSE}}{\partial B_{output}} = \delta_{output}$$

$$Assume:\ \delta_{hidden} = \delta_{output} \,.\, W_{output}^{T} \circ [a_{hidden} \circ (1 - a_{hidden})]$$

$$\frac{\partial z_{output}}{\partial a_{hidden}} = W_{output}^{T} \quad , \quad \frac{\partial a_{hidden}}{\partial z_{hidden}} = [a_{hidden} \circ (1 - a_{hidden})]$$

$$\xrightarrow{*} \frac{\partial J_{LSE}}{\partial W_{hidden}} = \delta_{hidden} \times \frac{\partial z_{hidden}}{\partial W_{hidden}} = \delta_{hidden} \,.\, X^{T}$$

$$\xrightarrow{**} \frac{\partial J_{LSE}}{\partial B_{hidden}} = \delta_{hidden}$$

(IV) GD with momentum:

Initialize momentum terms $V_{dW_{output}}$, $V_{dB_{output}}$, $V_{dW_{hidden}}$, $V_{dB_{hidden}}$ to zero.
we apply update rule by consideration of momentum term and learning rate :

$$\delta_{output} = -0.1311$$ **(OUTPUT LAYER)**

$$V_{dW_{output}} \leftarrow \beta\, V_{dW_{output}} - \alpha \frac{\partial J_{LSE}}{\partial W_{output}}$$
$$W_{output} = W_{output} + V_{dW_{output}}$$

$$V_{dW_{output}} \leftarrow -(0.9)\,\delta_{output} \begin{bmatrix} 0.3318 & 0.5245 \end{bmatrix} = \begin{bmatrix} 0.0391 & 0.0619 \end{bmatrix}$$

$$W_{output} = \begin{bmatrix} -0.3 & -0.2 \end{bmatrix} + \begin{bmatrix} 0.0391 & 0.0619 \end{bmatrix} = \begin{bmatrix} \mathbf{-0.2609} & \mathbf{-0.1381} \end{bmatrix}$$

$$V_{dB_{output}} \leftarrow \beta V_{dB_{output}} - \alpha \frac{\partial J_{LSE}}{\partial B_{output}}$$
$$B_{output} = B_{output} + V_{dB_{output}}$$

$$V_{dB_{output}} \leftarrow -(0.9)\,\delta_{output} = [0.1180]$$
$$B_{output} = 0.1 + 0.1180 = \mathbf{0.2180}$$

(HIDDEN LAYER)

$$\delta_{hidden} = \delta_{output} \begin{bmatrix} -0.3 \\ -0.2 \end{bmatrix} \circ \begin{bmatrix} 0.2217 \\ 0.2494 \end{bmatrix} = \begin{bmatrix} -0.0087 \\ -0.0065 \end{bmatrix}$$

$$V_{dW_{hidden}} \leftarrow \beta V_{dW_{hidden}} - \alpha \frac{\partial J_{LSE}}{\partial W_{hidden}}$$
$$W_{hidden} = W_{hidden} + V_{dW_{hidden}}$$

$$V_{dW_{hidden}} \leftarrow -(0.9) \begin{bmatrix} -0.0087 \\ -0.0065 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.0078 \\ -0.0059 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.0078 & 0 & -0.0078 \\ -0.0059 & 0 & -0.0059 \end{bmatrix}$$

$$W_{hidden} = \begin{bmatrix} 0.2 & 0.4 & -0.5 \\ -0.3 & 0.1 & 0.2 \end{bmatrix} + \begin{bmatrix} -0.0078 & 0 & -0.0078 \\ -0.0059 & 0 & -0.0059 \end{bmatrix} = \begin{bmatrix} 0.1922 & 0.4 & -0.5078 \\ -0.3059 & 0.1 & 0.1941 \end{bmatrix}$$

$$V_{dB_{hidden}} \leftarrow \beta V_{dB_{hidden}} - \alpha \frac{\partial J_{LSE}}{\partial B_{hidden}}$$
$$B_{hidden} = B_{hidden} + V_{dB_{hidden}}$$

$$V_{dB_{hidden}} \leftarrow -(0.9) \begin{bmatrix} -0.0087 \\ -0.0065 \end{bmatrix} = \begin{bmatrix} -0.0078 \\ -0.0059 \end{bmatrix}$$

$$B_{hidden} = \begin{bmatrix} -0.4 \\ 0.2 \end{bmatrix} + \begin{bmatrix} -0.0078 \\ -0.0059 \end{bmatrix} = \begin{bmatrix} -0.4078 \\ 0.1941 \end{bmatrix}$$

Iteration 2:

(I) forward pass:

$$z_{\,hidden} = W_{hidden} \,.\, X + B_{hidden}$$

$$z_{\,hidden} = \begin{bmatrix} 0.1922 & 0.4 & -0.5078 \\ -0.3059 & 0.1 & 0.1941 \end{bmatrix} . \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} -0.4078 \\ 0.1941 \end{bmatrix} = \begin{bmatrix} -0.7234 \\ 0.0823 \end{bmatrix}$$

$$a_{\,hidden} = \sigma(\,z_{\,hidden}\,) = \sigma\left(\begin{bmatrix} -0.7234 \\ 0.0823 \end{bmatrix}\right) = \begin{bmatrix} \sigma(-0.7234) \\ \sigma(0.0823) \end{bmatrix} = \begin{bmatrix} 0.3266 \\ 0.5205 \end{bmatrix}$$

$$z_{\,output} = W_{output} \,.\, a_{\,hidden} + B_{output}$$

$$z_{\,output} = \begin{bmatrix} -0.2609 & -0.1381 \end{bmatrix} . \begin{bmatrix} 0.3266 \\ 0.5205 \end{bmatrix} + \begin{bmatrix} 0.2180 \end{bmatrix} = \begin{bmatrix} 0.0609 \end{bmatrix}$$

$$a_{\,output} = \sigma(\,z_{output}\,) = \sigma\left(\begin{bmatrix} 0.0609 \end{bmatrix}\right) = \begin{bmatrix} \sigma(0.0609) \end{bmatrix} = \begin{bmatrix} 0.5152 \end{bmatrix}$$

(II) compute costs:

$$J_{LSE}(y, \widehat{y}\,) = \frac{1}{2} \sum_{i=1}^{m} (\,\widehat{y}_i - y_i)^2 = \frac{1}{2}(\,\widehat{y} - y)^2$$

$$J_{LSE} = \frac{1}{2}(\,0.5152 - 1\,)^2 = 0.1175$$

here we have only one record (m = 1)

$$J_{CE}(y,\widehat{y}) = -y \cdot log(\widehat{y}) - (1-y) \cdot log(1-\widehat{y})$$

$$J_{CE} = -1 \,.\, log(0.5152) - (1-1) \,.\, log\,(1-0.5152) = 0.6632$$

(III) backward pass:

as same as the iteration 1.

(IV) GD with moumentum:

ike the iteration 1. we follow the mentioned steps

Q2.

we at first we review each of adaptive learning algorithms. (source1 - source2 - source3)

1. SGD:

Mini batch SGD sits right in the middle of the two previous ideas combining the best of both worlds. It randomly selects **n** training examples, the so-called mini-batch, rom the whole dataset and computes the gradients only from them. It essentially tries to approximate Batch Gradient Descent by sampling only a subset of the data.

Mathematically:

$$w = w - \texttt{learning\_rate} \cdot \nabla_w L(x_{(i:i+n)}, y_{(i:i+n)}, W)$$

GD comes with some limitations and problems that might negatively affect the training:

If the loss function changes quickly in one direction and slowly in another, it may result in a high oscillation of gradients making the training progress very slow.

- If the loss function has a local minimum or a saddle point, it is very possible that SGD will be stuck there without being able to "jump out" and proceed in finding a better minimum. This happens because the gradient becomes zero so there is no update in the weight whatsoever.

- The gradients are still noisy because we estimate them based only on a small sample of our dataset. The noisy updates might not correlate well with the true direction of the loss function.

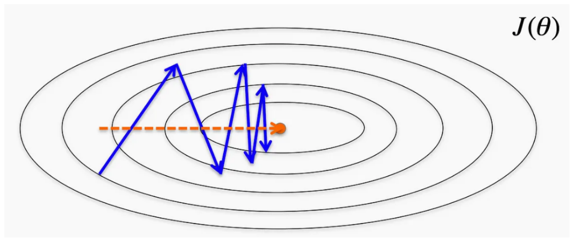
- Choosing a good loss function is tricky and requires time-consuming experimentation with different hyperparameters.

- The same learning rate is applied to all of our parameters, which can become problematic for features with different frequencies or significance.

2. Momentum:

One problem with stochastic gradient descent (SGD) is the presence of oscillations which result from updates not exploiting curvature information. This results in SGD being slow when there is high curvature.By taking the average gradient, we can obtain a faster path to optimization. This helps to dampen oscillations because gradients in opposite directions get canceled out.

Momentum can also help us reduce the oscillation of the gradients because the velocity vectors can smooth out these highly changing landscapes. Finally, it reduces the noise of the gradients (stochasticity) and follows a more direct walk down the landscape.



Momentum Based Gradient Descent Update Rule

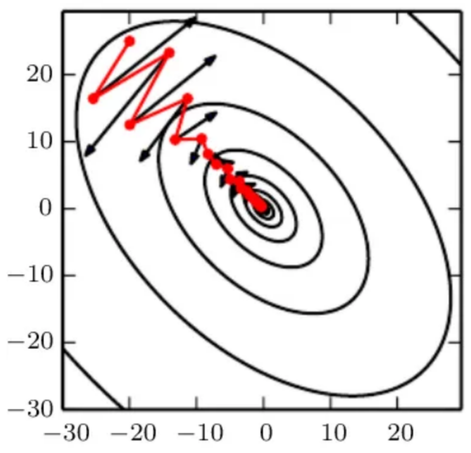
$$\vartheta_t = \gamma * \vartheta_{t-1} + \eta \bigtriangledown \omega_t$$

$$\omega_{t+1} = \omega_t - \vartheta_t$$

$\gamma \rightarrow$Hyperparameter to control the history

$\vartheta \rightarrow$Momentum

We can see the effects of adding momentum on an optimization algorithm. The first few updates show no real advantage over vanilla SGD — since we have no previous gradients to utilize for our update. As the number of updates increases our momentum kickstarts and allows faster convergence. if we use bias form of moumentum the start will be smooth too and it takes less time to convergence.



SGD without momentum (black) compared with SGD with momentum (red).

3. RMSprop:

as we know RMSprop is a utilized edition of Adagrad. A big drawback of Adagrad is that as time goes by, the learning rate becomes smaller and smaller due to the monotonic increment of the running squared sum. A solution to this problem is a modification of the algorithm, called RMSProp, which can be thought of as a "Leaky Adagrad". In essence, we add once again the notion of friction by decaying the sum of the previous squared gradients.

As we did in Momentum based methods, we multiply our term (here the running squared sum) with a constant value (the decay rate). That way we hope that the algorithm will not slow down over the course of training as Adagrad does.

RMSProp

$$\vartheta_t = \beta * \vartheta_{t-1} + (1 - \beta)(\nabla \omega_t)^2$$
$$\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{(\vartheta_t)} + \epsilon} \nabla \omega_t$$

4. Adam:

think this is the most efficient algorithm out there. It's a perfect blend of Momentum Based GD and RMSProp. It takes two histories, one is for momentum and one for learning rate. That's how the Adam algorithm combining the advantages of two algorithms. One term ensures that the learning rate does not get killed for weights and the other ensures that the gradient update does not only depends on current derivatives also the prev derivatives.

Adam

$$m_t = \beta_1 * \vartheta_{t-1} + (1-\beta)(\bigtriangledown \omega_t)$$
$$\vartheta_t = \beta_2 * \vartheta_{t-1} + (1-\beta)(\bigtriangledown \omega_t)^2$$
$$\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{(\vartheta_t)}+\epsilon} \bigtriangledown m_t$$

Comparison and Results:

-Speed of Convergence:

Adam generally converges faster due to its combination of momentum and adaptive learning rates. RMSprop is also faster than basic SGD, addressing its limitations.

-Global Minimum Convergence:

Adam and RMSprop have better chances of converging to global minima due to their adaptive learning rates and momentum-like terms.

-Update Rule:

SGD: It updates the weights in the opposite direction of the gradient of the loss function with respect to the weights.
Momentum: It adds a fraction (momentum term) of the previous update to the current update.
RMSprop: It adapts the learning rates of each parameter separately based on the average of recent squared gradients.
Adam: Combines the ideas of momentum and RMSprop. It maintains two moving averages for each parameter - one for the gradients and one for the squared gradients.

-Learning Rate Adaptation:

SGD: Uses a fixed learning rate.
Momentum: Can adapt the effective learning rate by considering the momentum term.
RMSprop: Adapts learning rates for each parameter based on the magnitude of recent gradients.
Adam: Dynamically adjusts the learning rates for each parameter using both first-order (momentum) and second-order (RMSprop) information.

-**Momentum:**

SGD: Does not have an explicit momentum term.

Momentum: Has a momentum term that helps accelerate SGD in the relevant direction and dampens oscillations.

RMSprop: Does not have an explicit momentum term.

Adam: Incorporates momentum by maintaining a moving average of the gradients.

Adaptivity

SGD: Non-adaptive - uses a fixed learning rate.

Momentum: Partially adaptive due to the momentum term.

RMSprop: Adaptive learning rates for each parameter.

Adam: Adaptive learning rates for each parameter, combining momentum and RMSprop.

-**Bias Correction:**

SGD, Momentum, RMSprop: Typically do not include bias correction.

Adam: Incorporates bias correction to the moving averages of the gradients and squared gradients.

-**Use Cases:**

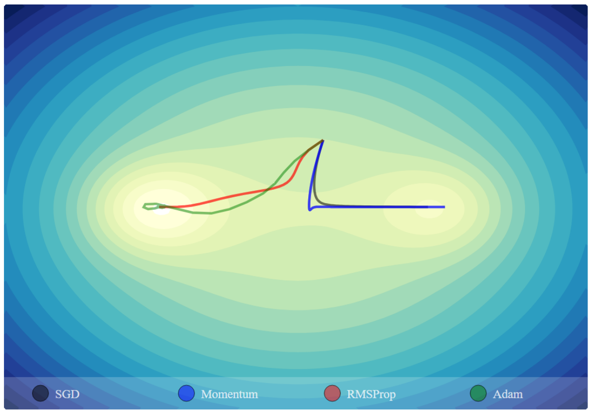
SGD: Simple and often used as a baseline. Suitable for convex and non-convex optimization problems.

Momentum: Useful for accelerating convergence, especially in the presence of noisy gradients.

RMSprop: Effective for non-stationary problems with sparse data.

Adam: Generally performs well across a wide range of optimization tasks and is widely used in practice.

Now that the general differences of each algorithm have been outlined, let's choose some specific points in a simulation and examine the practical behavior of each algorithm:

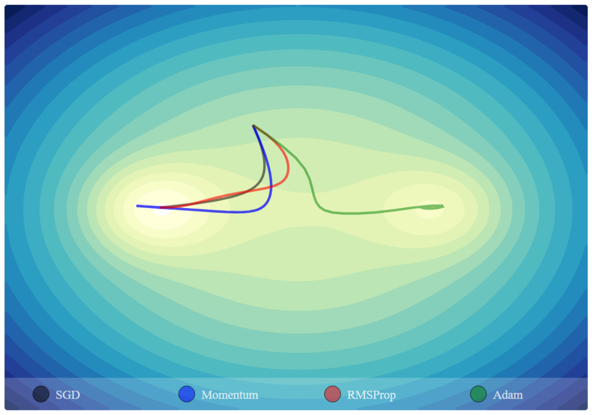


The SGD algorithm, as explained earlier, randomly selects a data point initially and, due to its high learning rate, quickly moves towards the center of the convex space in a nearly straight path. However, as it progresses, its learning rate decreases, leading to slower direction changes to reach the optimal point. Also, due to the reduced learning rate, it converges to a local optimum and cannot escape from this starting point to find a global optimum.

The Momentum algorithm has a movement pattern similar to SGD but, upon reaching the center, it continues slightly along its path due to the influence of the momentum term. The speed of this algorithm also increases initially due to the accumulation of gradients, but when reaching the center, it becomes slow because the effects of gradients need to be altered, resulting in a slow continuation of the path. However, after the complete change in direction, its speed increases again, surpassing SGD in the final point, thanks to the higher speed it had. Similar to SGD, when it reaches the center, it cannot find the global optimum due to the slow speed.

The RMSProp algorithm, as explained earlier, starts quickly moving towards the center due to the adaptive learning rates for each weight. However, under the influence of changing learning rates, its direction shifts towards the global optimal point, creating a curve as it moves. The previous learning rates, which were directed towards the center of the plane, decrease, altering its path towards the global optimal point with a smoother motion.

The Adam algorithm, as mentioned, is a combination of RMSProp and Momentum. It moves faster than other algorithms due to the influence of both components and reaches its destination. Its path is influenced by the speed of weight changes. In the early stages, similar to RMSProp, it moves towards the global optimum, and later, with an increase in the speed of weight changes, it deviates from the smooth motion of RMSProp, creating a curve under the center. The slight oscillation observed in its movement is due to the increasing speed in the later stages, contributing to a slight deviation in its direction.



The Momentum quickly accelerates towards the global optimum due to its ability to accumulate velocity over iterations. Its initial momentum allows it to overcome local minima efficiently, reaching the global optimum before the other algorithms.

The Adam, being a combination of Momentum and RMSProp, follows a trajectory that benefits from both components. Initially, it moves smoothly towards the global optimum, leveraging the adaptive learning rates and momentum. However, as it approaches the optimal point, the momentum slows down due to the changing direction of gradients, causing Adam to settle into a local optimum before reaching the global optimum.

The SGD, with its straightforward and high learning rate, quickly moves towards the global optimum. However, due to its lack of momentum and adaptive learning rates, it might overshoot the optimal point, oscillate around it, and eventually converge to the global optimum in a more erratic fashion compared to Momentum.

The RMSProp, benefiting from adaptive learning rates, rapidly converges towards the global optimum. The algorithm's ability to adjust learning rates based on the magnitude of gradients allows it to navigate efficiently through the optimization landscape. In this scenario, RMSProp converges to the global optimum after Momentum but before Adam.

Q3.

It sounds like we are encountering an issue known as the **vanishing gradient problem** while training Neural Networks (usaully deep NN) with gradient-based methods (example, Back Propagation)(figure 3.1). as the question described the problem when in DNN we start on the output layer and propagate the error all the way to the input layer and updating the weights with respect to the loss function as we go.As we mentioned earlier the weights are updated using their partial derivative with respect to the loss function. The problem is that these gradients get smaller and smaller as we approach the lower layers of the network.therefore it leads to the lower layers' weights barely changing when training the network.

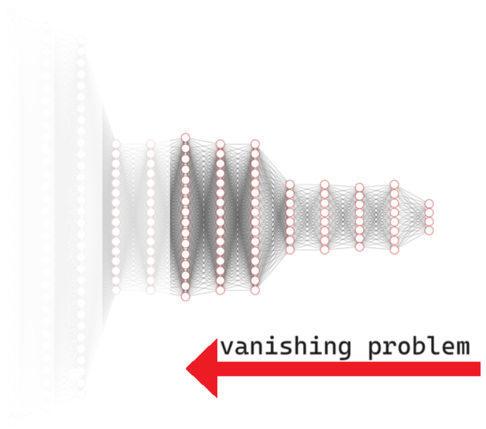


Figure 3.1

This can be caused due the following:

1.Activation Functions: Certain activation functions, like the sigmoid or tanh function, squishes a large input space into a small input space. Therefore, a large change in the input of the sigmoid function will cause a small change in the output and it also satureted in $\pm\infty$. (figure 3.2 and 3.3)

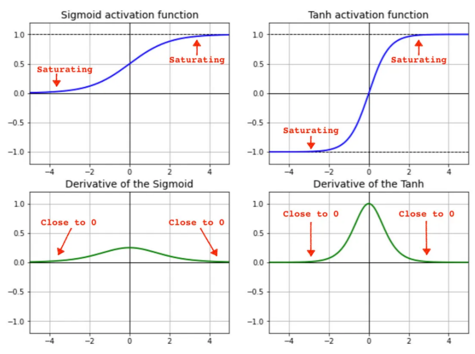


Figure 3.2: image from towardsdatascience.com

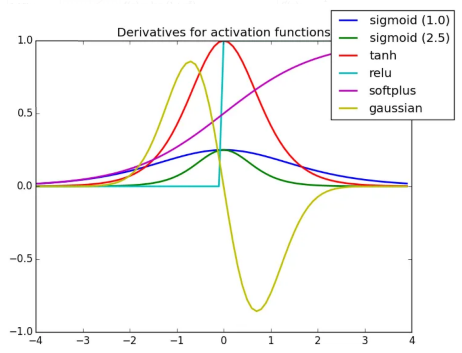
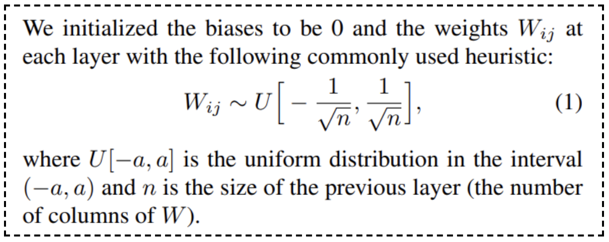


Figure 3.3: image from towardsdatascience.com

The simplest solution is to use other activation functions, such as ReLU or Leaky/Random ReLU, which don't cause a small derivative becuase they do not saturate for positive values. This activations at most saturates on one direction and thus are more resilient to the vanishing of gradients. (source)

2.Weights initialization: initial weights can give the network a good starting point to learn. And if we can select the initial weights of the network more carefully, we can have the network converge faster and learn better. If the weights are too large or too small, we saw that the derivatives of some non linear activation functions (like sigmoid) becomes very small, almost equal to 0.

if we initialized all the weights with 0, all weights have the same value in subsequent iterations. This makes hidden layers symmetric and this process continues for all the n iterations.Thus initialized weights with zero make your network no better than a linear model. also its good to metion its not cuase vanishing grandient problem just it lost its efficiency but we want to solve vanishing gradient and save the efficiency. We need to make sure that the weights are in a reasonable range and all not equal to zero before we start training the network. Xavier initialization and some other enhancement (like He Normal Initialization)
on it are some attempts to do exactly that. (figure 3.4)


We initialized the biases to be 0 and the weights W_{ij} at each layer with the following commonly used heuristic:

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right], \quad (1)$$

where $U[-a, a]$ is the uniform distribution in the interval $(-a, a)$ and n is the size of the previous layer (the number of columns of W).

Figure 3.4: from "understanding the difficulty of training deep neural networks" paper by Xavier Glorot & Yoshua Bengio

3.Dept of the neural network:

By the chain rule, the derivatives of each layer are multiplied down the network (from the final layer to the initial) to compute the derivatives of the initial layers with regard to backpropagation. when n hidden layers use an activation like the sigmoid function, n small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers. A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. if we don't want to change our activation function we must change the architucture(due to this we can us Residual networks or LSTM but we don't discuss about regarding to course syllabus) or use the adaptive learning rate (moumentum,adagrad,..) and Dropout Layers(Figure 3.5) or etc.

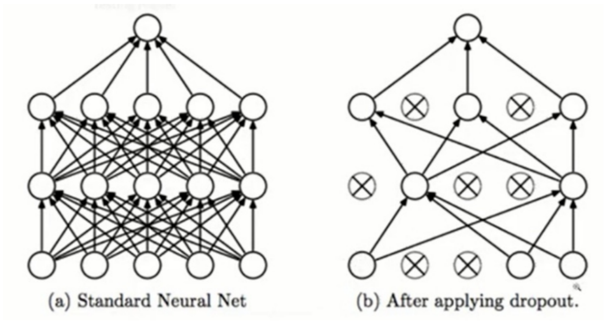


Figure 3.5

Q4.

$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}_{3\times1} \rightarrow \; w^T = \begin{bmatrix} w_0 & w_1 & w_2 \end{bmatrix} \quad , \quad X(x) = \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}_{3\times1}$$

$$u(x) = w^T . X(x) \qquad , \qquad y(x) = \sigma\,(u(X)) = \frac{1}{1+e^{-\,u(x)}}$$

$$L_{CE}(x,c) = -\,c\,ln\,(\,y(x)\,) - (1-\,c\,)\,ln\,(1-\,y(x))$$

$$J\;(D,w) = \frac{1}{N}\sum_{k=1}^{N}[\;L_{CE}\,(x_k,c_k)\,]$$

$$\delta = \frac{\partial\,L_{CE}}{\partial\,y} \times \frac{\partial\,y}{\partial\,u} \times \frac{\partial\,u}{\partial\,w} = [\,y(x)-c\,]\,.\;X(x)$$

$$\frac{\partial L_{CE}}{\partial y} = \frac{-c}{y(x)} + \frac{1-c}{1-y(x)}$$

$$\frac{\partial y}{\partial u} = y(x)\,(1-y(x))$$

$$\frac{\partial u}{\partial w} = X(x)$$

$$\frac{\partial J}{\partial w} = \frac{1}{N}\sum_{k=1}^{N}[\,y(x_k)-c_k\,]\,.\;X(x_k)$$