ECE C147/C247, Winter 2024 **Discussion 5**
Neural Networks & Deep Learning Prof. J. Kao
UCLA; Department of ECE TAs: T. Monsoor, Y. Liu, S. Rajesh, L. Julakanti, K. Pang

In this discussion, we will discuss the following concepts related to the optimization of neural networks:

- Challenges with optimization
- Momentum
- Adaptive learning rates

1 Challenges with optimization

When talking about optimization in the context of neural networks, we are discussing non-convex optimization.

Convex optimization involves a function in which there is only one optimum, corresponding to the global optimum (maximum or minimum). There is no concept of local optima for convex optimization problems, making them relatively easy to solve — these are common introductory topics in undergraduate and graduate optimization classes.

Non-convex optimization involves a function which has multiple optima, only one of which is the global optima. Depending on the loss surface, it can be very difficult to locate the global optima

For a neural network, the curve or surface that we are talking about is the loss surface. Since we are trying to minimize the prediction error of the network, we are interested in finding the global minimum on this loss surface — this is the aim of neural network training.

There are multiple problems associated with it:

- **Learning rate**: Too small a learning rate takes too long to converge, and too large a learning rate will mean that the network will not converge.
- **Local optima**: One local optimum may be surrounded by a particularly steep loss function, and it may be difficult to 'escape' this local optimum.
- **Loss surface**: Even if we can find the global minimum, there is no guarantee that it will remain the global minimum indefinitely. A good example of this is when training on a dataset that is not representative of the actual data distribution — when applied to new data, the loss surface will different. This is one reason why trying to make the training and test datasets representative of the total data distribution is of such high importance. Another good example is data which habitually changes in distribution due to its dynamic nature — an example of this would be user preferences for popular music or movies, which changes day-to-day and month-to-month.

Fortunately, there are methods available that provide ways to tackle all of these challenges, thus mitigating their potentially negative ramifications.

1.1 Local minima

Previously, local minima were viewed as a major problem in neural network training. Nowadays, researchers have found that when using sufficiently large neural networks, most local minima incur a low cost, and thus it is not particularly important to find the true global minimum — a local minimum with reasonably low error is acceptable.

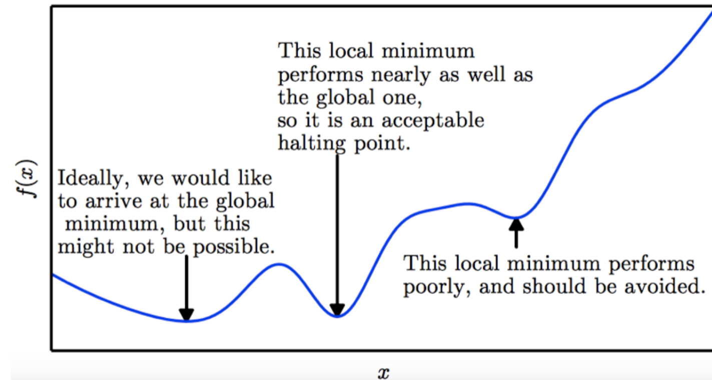

Figure 1: Local minima

1.2 Saddle Points

Recent studies indicate that in high dimensions, saddle points are more likely than local minima. Saddle points are also more problematic than local minima because close to a saddle point the gradient can be very small. Thus, gradient descent will result in negligible updates to the network and hence network training will cease.

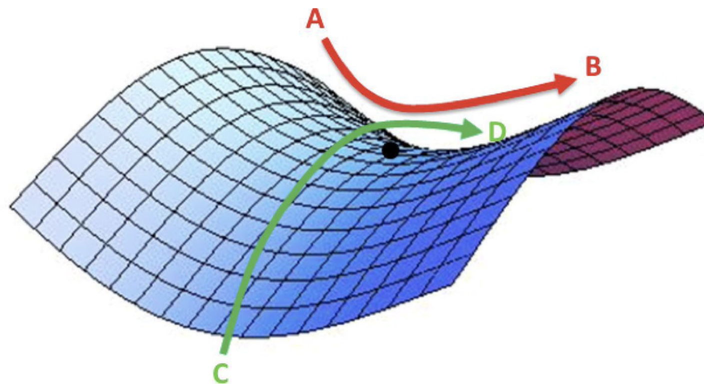

Figure 2: Saddle point — simultaneously a local minimum and a local maximum.

Explain intuitively that why local minimas become exponentially rare as the dimensionality of the parameter space increases.

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.

Figure 3: (Left) Vanilla SGD, (right) SGD with momentum.

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.

The name momentum comes from the fact that this is analogous to the notion of linear momentum from physics. An object that has motion (in this case it is the general direction that the optimization algorithm is moving) has some inertia which causes them to tend to move in the direction of motion. Thus, if the optimization algorithm is moving in a general direction, the momentum causes it to 'resist' changes in direction, which is what results in the dampening of oscillations for high curvature surfaces.

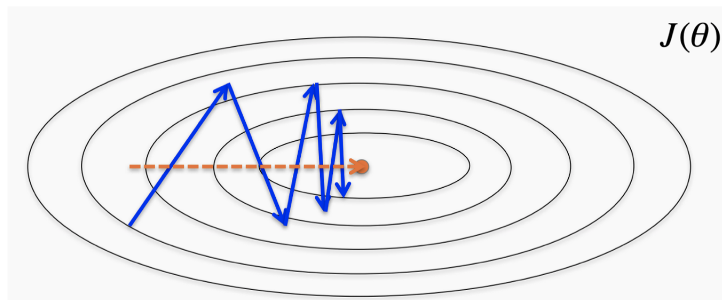

Figure 4: Dampening of oscillations

Momentum uses past gradients for updating values, as shown in the formula below. The value v associated with momentum is often called the 'velocity'. More weight is applied to more recent gradients, creating an exponentially decaying average of gradients.

$$v \leftarrow \alpha v - \epsilon g$$
$$\theta \leftarrow \theta + v$$

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.

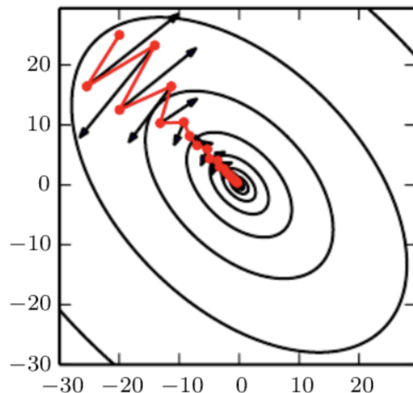

Figure 5: SGD without momentum (black) compared with SGD with momentum (red).

2.1 Nesterov momentum

The main difference is in classical momentum you first correct your velocity and then make a big step according to that velocity (and then repeat), but in Nesterov momentum, you first make a step into velocity direction and then make a correction to a velocity vector based on a new location (then repeat). The difference is subtle but in practice, it can make a huge difference. This concept can be difficult to comprehend, so below is a visual representation of the difference between the traditional momentum update and Nesterov momentum.

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \mathcal{L}(\theta + \alpha v)$$
$$\theta \leftarrow \theta + v$$

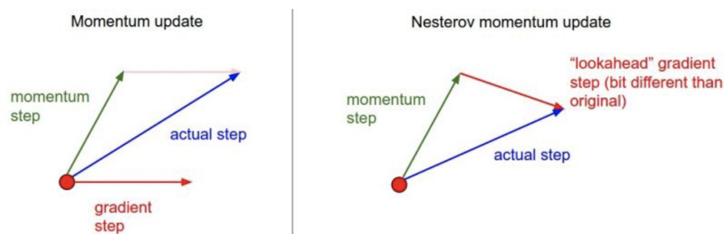

Figure 6: Nesterov momentum

3 Adaptive learning rates

As we have discussed previously, another major challenge for training deep networks is appropriately selecting the learning rate. Choosing the correct learning rate has long been one of the most troublesome aspects of training deep networks because it has a major impact on a network's performance. A learning rate that is too small doesn't learn quickly enough, but a learning rate that is too large may have difficulty converging as we approach a local minimum or region that is ill-conditioned.

One of the major breakthroughs in modern deep network optimization was the advent of learning rate adaption. The basic concept behind learning rate adaptation is that the optimal learning rate is appropriately modified over the span of learning to achieve good convergence properties. Over the next several sections, we'll discuss AdaGrad, RMSProp, and Adam, three of the most popular adaptive learning rate algorithms.

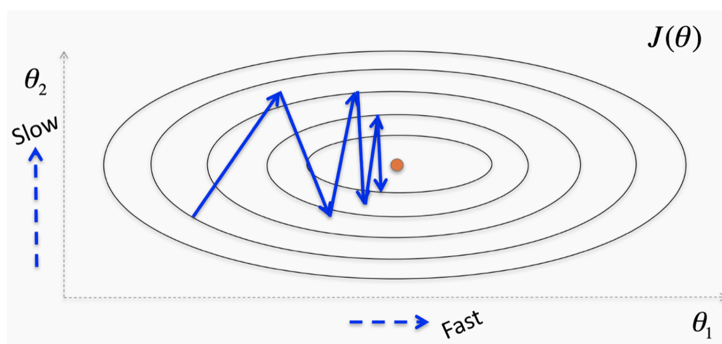

Figure 7: Different learning rates for parameters

3.1 Adagrad

The first algorithm we'll discuss is AdaGrad, which attempts to adapt the global learning rate over time using an accumulation of the historical gradients, first proposed by Duchi et al. in 2011. Specifically, we keep track of a learning rate for each parameter. This learning rate is inversely scaled with respect to the square root of the sum of the squares (root mean square) of all the parameter's historical gradients.

We can express this mathematically. We initialize a gradient accumulation vector $\mathbf{a} = 0$. At every step, we accumulate the square of all the gradient parameters as follows

$$a \leftarrow a + g \odot g$$

Then we compute the update as usual, except our global learning rate ϵ is divided by the square root of the gradient accumulation vector:

$$\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{\mathbf{a}} + \nu} \odot g$$

where we added a tiny number ν to the denominator to prevent division by zero.

On a functional level, this update mechanism means that the parameters with the largest gradients experience a rapid decrease in their learning rates, while parameters with smaller gradients only observe a small decrease in their learning rates. The ultimate effect is that AdaGrad forces more progress in the more gently sloped directions on the error surface, which can help overcome ill-conditioned surfaces. AdaGrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This, in turn, causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

3.2 RMSProp - Exponentially Weighted Moving Average of Gradients

While AdaGrad works well for simple convex functions, it isn't designed to navigate the complex loss surfaces of deep networks. Flat regions may force AdaGrad to decrease the learning rate before it reaches a minimum. The conclusion is that simply using a naive accumulation of gradients isn't sufficient.

Our solution is to bring back a concept we introduced earlier while discussing momentum to dampen fluctuations in the gradient. Compared to naive accumulation, exponentially weighted moving averages also enable us to "toss out" measurements that we made a long time ago. More specifically, our update to the gradient accumulation vector is now as follows:

$$a \leftarrow \beta a + (1 - \beta) g \odot g$$

The decay factor β determines how long we keep old gradients. The smaller the decay factor, the shorter the effective window. Plugging this modification into AdaGrad gives rise to the RMSProp learning algorithm, first proposed by Geoffrey Hinton. Overall, RMSProp has been shown to be a highly effective optimizer for deep neural networks, and is a default choice for many seasoned practitioners.

3.3 Adam

Before concluding our discussion of modern optimizers, we discuss one final algorithm—Adam. Spiritually, we can think about Adam as a variant combination of RMSProp and momentum.

The basic idea is as follows. We want to keep track of an exponentially weighted moving average of the gradient (essentially the concept of velocity in classical momentum), which we can express as follows:

$$v \leftarrow \beta_1 v + (1 - \beta_1) g$$

This is our approximation of what we call the first moment of the gradient, $E[g]$. And similarly to RMSProp, we can maintain an exponentially weighted moving average of the historical gradients. This is our estimation of what we call the second moment of the gradient, $E[g \odot g]$

$$a \leftarrow \beta_2 a + (1 - \beta_2) g \odot g$$

However, it turns out these estimations are biased relative to the real moments because we start off by initializing both vectors to the zero vector. In order to remedy this bias, we derive a correction factor for both estimations.

$$\tilde{v} = \frac{1}{1-\beta_1^t} v$$
$$\tilde{a} = \frac{1}{1-\beta_2^t} a$$

We can then use these corrected moments to update the parameter vector, resulting in the final Adam update:

$$\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{\tilde{a}} + \nu} \odot \tilde{v}$$

4 Practice problems

1. **Vanishing and exploding gradients**
   Let's consider the following neural network with n layers. As features at the first layer are

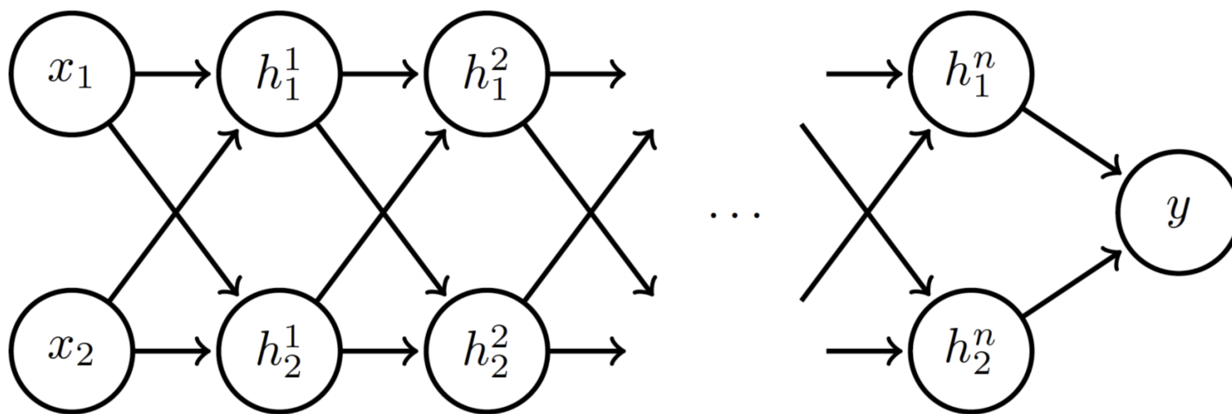

Figure 8: DNN with n layers

propagated through the network, they undergo affine transformations described as follows:

$$h^i = Wx$$
$$h^i = Wh^{i-1}, \;\; i = 2, 3, \cdots, n$$
$$y = h_1^n + h_2^n$$

   (a) Assuming $W = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$, write down the expression for h_1^n and h_2^n.

   (b) Assuming $x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $a = 1$, and $b = 2$ write down the values of y and ∇y. What happens to ∇y as n gets large.

(c) Assuming $x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $a = 0.5$, and $b = 0.9$ write down the values of y and ∇y. What happens to ∇y as n gets large.

2. **Bias correction in Adam**

Recall, that in Adam optimizer, at the tth iteration we perform the following steps:

$$\begin{aligned}
\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g_t} \\
\mathbf{a}_t &\leftarrow \beta_2 \mathbf{a}_{t-1} + (1-\beta_2)\mathbf{g_t} \odot \mathbf{g_t} \\
\tilde{\mathbf{v}}_t &= \frac{1}{1-\beta_1^t}\mathbf{v}_t \qquad \text{(bias correction for first moment)} \\
\tilde{\mathbf{a}}_t &= \frac{1}{1-\beta_2^t}\mathbf{a}_t \qquad \text{(bias correction for second moment)} \\
\theta_t &\leftarrow \theta_{t-1} - \frac{\varepsilon}{\sqrt{\tilde{\mathbf{a}}_t} + \nu} \odot \tilde{\mathbf{v}}_t,
\end{aligned}$$

where ν is a small value to prevent zero-division, β_1 and β_2 are the running average parameter for the first and second moment estimation and ε is the step size hyperparameter. From the above set of equations, you can observe that Adam algorithm designs the "bias correction" steps for the first and second moment estimation of the gradients. In this question, we are going to derive the correction factors.

We will treat the gradients along the optimization trajectory as random variables, and assume that $\mathbf{g_1}, \mathbf{g_2}, \ldots$, are i.i.d. with some distribution that has the first and second moment. That is, we assume

$$\begin{aligned}
\mathbb{E}[\mathbf{g_t}] &= \mu, \quad t = 1, 2, \ldots \\
\mathbb{E}[\mathbf{g_t}^2] &= \mathbf{s}, \quad t = 1, 2, \ldots
\end{aligned}$$

where for simplicity, we denote $\mathbf{g_t} \odot \mathbf{g_t}$ as $\mathbf{g_t}^2$. We first expand the recursive relation and express $\mathbf{v}_t$ in terms of $\mathbf{g_1}, \mathbf{g_2}, \ldots, \mathbf{g_t}$.

This gives

$$\begin{aligned}
\mathbf{v}_t &= (1-\beta_1)\mathbf{g_t} + \beta_1(1-\beta_1)\mathbf{g_{t-1}} + \beta_1^2(1-\beta_1)\mathbf{g_{t-2}} + \ldots + \beta_1^{t-1}(1-\beta_1)\mathbf{g_1} \\
&= (1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i}\mathbf{g_i} \qquad (1)
\end{aligned}$$

and similarly,

$$\mathbf{a}_t = (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i}\mathbf{g_i}^2 \qquad (2)$$

Then consider the expectation of $\mathbf{v}_t$, $\mathbb{E}[\mathbf{v}_t]$, and compare with μ.

Show that the correction factor $\gamma_1 = \dfrac{1}{1-\beta_1^t}$ satisfies

$$\gamma_1 \mathbb{E}[\mathbf{v}_t] = \mu = \mathbb{E}[\mathbf{g_t}].$$

You will see that $\gamma_2 = \dfrac{1}{1-\beta_2^t}$ corrects $\mathbb{E}[\mathbf{a}_t]$ to $\mathbf{s}$ (i.e. $\mathbb{E}[\mathbf{g_t}^2]$) in a similar way.

3. **Gradient of L_∞ norm**

   Let,

$$L(\mathbf{W}) = \|\mathbf{W}\|_\infty,$$

   where $\mathbf{W} \in \mathbb{R}^{n \times n}$. Compute $\nabla_{\mathbf{W}} L(\mathbf{W})$.