CS 337, Fall 2023
Backpropagation, Regularization and Optimizers

Scribes: Josyula Venkata Aditya, Yuvraj Singh, Keshava Kotagiri
Shubham Hazra, Komma Sharanya, Namrata Jha
Edited by: Mayank Jain

September 25, 2023

Disclaimer. Please note this document has not received the usual scrutiny that formal publications enjoy. This may be distributed outside this class only with the permission of the instructor.

1 Backpropagation: Recap

Backpropagation is a method used to find the gradients in a computational graph. The computational graph is constructed in such a way that it is a directed acyclic graph. Let us say that we have nodes in the computational graph $u, v_1, v_2, \cdots v_n$ as shown in the diagram, in such a way that there are edges from u to v_i $\forall i \in \{1, 2, \cdots n\}$ and these are the only edges emanating from u. Hence, the node u is "dependent" on nodes $v_1, v_2, \cdots v_n$ for backpropagation (whereas in the forward propagation, the nodes $v_1, v_2, \cdots v_n$ are dependent on u). Let us say that the gradients of the loss function $\mathcal{L}$ with respect to v_i's are

$$\frac{\partial \mathcal{L}}{\partial v_i}$$

Then, the gradient of the loss function with respect to u can be written as

$$\frac{\partial \mathcal{L}}{\partial u} = \sum_{i=1}^{n} \frac{\partial \mathcal{L}}{\partial v_i} \cdot \frac{\partial v_i}{\partial u}$$

This is basically the chain rule for partial derivatives. To utilize this equation for backpropagation, we would assume that in the above equation, $\frac{\partial \mathcal{L}}{\partial v_i}$ $\forall i$ are known before computing the gradient with respect to u, and since there is an edge from u to v_i, there is a function f_i such that $v_i = f_i(u)$(keeping other variables and parameters fixed). Hence, the second terms in the summation, $\frac{\partial v_i}{\partial u}$, are also easily computable.

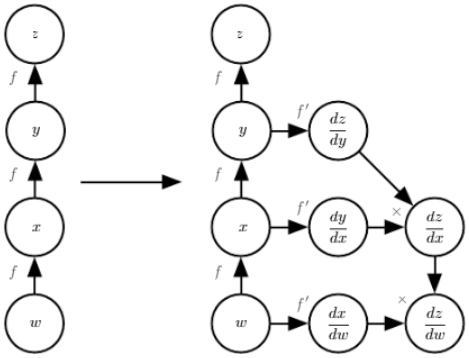


Figure 1: This diagram is taken from *Deep Learning by Goodfellow, Bengio and Courville*. It depicts backpropagation for a linear computational graph.

The above equation can be applied recursively over the layers in a neural network, and since the structure of the computational graph is a directed acyclic graph, this recursion will eventually terminate on reaching the input layer, or the layer immediately next to it. The base case would be initializing the gradient of the node in the computational graph corresponding to the loss function $\mathcal{L}$ to be 1, since the gradient of $\mathcal{L}$ with respect to itself is 1.

2 Regularization

2.1 L_2 Regularization

This regularizer has been our friend since the time we started learning regularization in Machine Learning. Consider the case of multi-class classification in a neural network. The final layer in the network would typically be a softmax layer which converts real-valued outputs to probabilites - numbers in the range $(0, 1)$. Let $\mathcal{D} = \{(x_i, y_i)\}_{i \in [N]}$ be the training data where x_i's are the features and y_i's are the corresponding class labels. Let $\{\hat{y}_{ik}\}$ be the softmax output of input x_i corresponding to the k^{th} class as computed by the neural network. Considering the weights of the neural network concatenated in a single flat vector $\mathbf{w}$, we can define the regularized cross entropy loss as

$$\mathcal{L}_{\mathbf{w}}(\mathcal{D}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \mathcal{I}(y_i = k) \log \hat{y}_{ik} + \beta \mathbf{w}^T \mathbf{w}$$

where $\mathcal{I}$ is the indicator function and β is the regularization strength. Note that there may be other forms of L_2 regularization as well which assign different regularization strengths to weights corresponding to different layers of the network. Biases aren't included in $\mathbf{w}$ because it can lead to underfitting, just as was the case in regularized linear regression.

2.2 Dropout Regularization

This kind of regularization is native to neural networks. It is simple to implement and has proven to be quite effective in practice. To motivate this regularizer, consider *bagging* in classical ML. In essence, an ensemble of machine learning models are trained over different training sets and then each test point is evaluated using multiple trained models. This is not easy to achieve in deep learning because the training time for deep networks is high and training itself is memory intensive. *Dropout* provides a simple surrogate to this problem. The general intuition is that we want to create(or pretend to create) an ensemble of neural networks by removing one or more neurons from the base architecture. Consider a number $p \in (0, 1)$. The training algorithm can be written as follows

1. For every batch $\mathcal{B}$ of the training data, do the following
2. Consider the base neural network, now for each neuron that is **not** in the input layer or the output layer, choose to deactivate that neuron i.e. make its output zero with a probability p.
3. Train this subset of the neural network on $\mathcal{B}$.

Note that unlike bagging, there is *parameter sharing* in dropout regularization. That is, the parameters from a previous training step are carried over to the next training step, which uses a different subset of the neural network whereas in bagging, each model is trained independently of the others. This difference between dropout and bagging allows dropout to be an an inexpensive surrogate for bagging. There have been studies where p changes across layers of the network, which did not lead to great improvements. During forward propagation, dropout requires that the neurons which have been deactivated in that training step must have output zero. This is fairly easy to implement - just use a mask vector for each layer and multiply the layer output with mask. During backpropagation with dropouts, the values of the nodes post-dropout must be used in the computational graph, and not pre-dropout. This essentially just replaces the dropped nodes with zero without having to modify the algorithm for backpropagation.

2.2.1 During Testing

There are two methods to evaluate the neural ensemble on a test datum:

- **Method 1**: Create multiple sub-networks using the same dropout probability p and average the predictions from all these sub-networks. This is similar to evaluation method in bagging.
- **Method 2**: There is another way to evaluate the test datum which uses a single forward propagation of the entire base network(without any dropout), and perform well empirically. Intuitively, each neuron will be active with a probability $1-p$ in the ensemble. So, the idea is to multiply the weights attached to each neuron by $1-p$ and making one single forward propagation to get the result corresponding to the test datum.

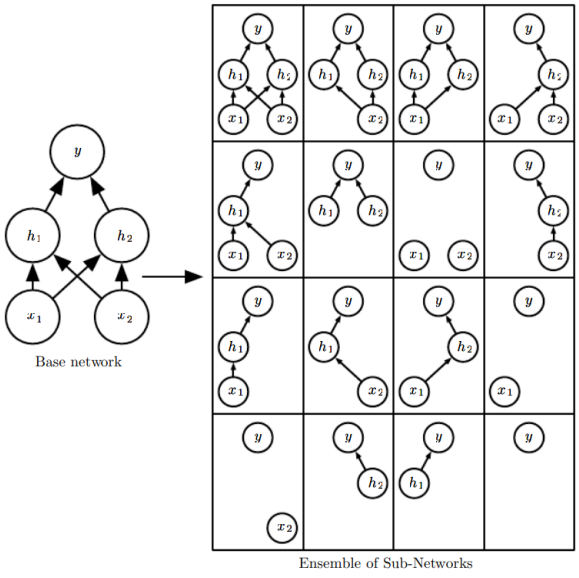


Figure 2: This diagram is taken from *Deep Learning by Goodfellow, Bengio and Courville*. It shows the various possible sub-networks that can be formed using dropouts.

2.3 Early Stopping

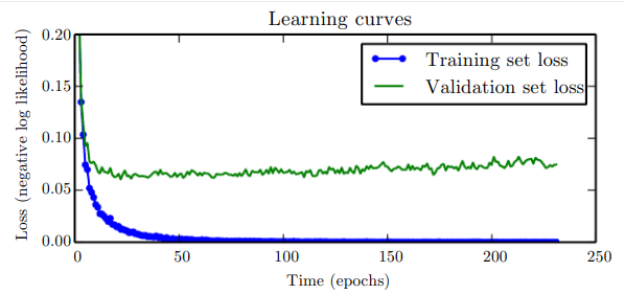
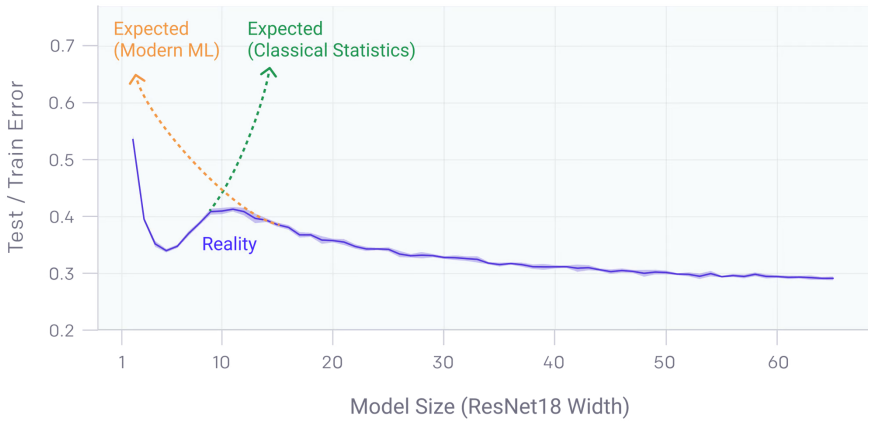


Figure 3: This diagram is taken from *Deep Learning by Goodfellow, Bengio and Courville.*

Let us recall the above graph of training set and validation set losses vs number of epochs. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.
Ideally, one would want training to proceed until the point he/she is satisfied with the validation and training accuracies/loss values achieved. That is, a point where the training loss is not too low implying a higher validation loss neither too high which again implies higher validation loss. One way to implement this is to keep a track of the number of consecutive epochs where the validation loss has failed to decrease, keep storing the models with the best intermediate validation error and stop training after the validation error has failed to decrease over a certain number of epochs(this is a hyperparameter and needs to be tuned) i.e. *early stopping.*
Early stopping can also be used along with other regularizers, since it is easy to implement without changing the learning step. For neural networks especially, it turns out that *early stopping* works well in that it is believed that neural networks tend to remember training examples as the number of epochs becomes high.

2.3.1 Additional Reading - Double descent

Double descent is a phenomenon where we observe strong test performance from very overfit, complex models. As we can see in Figure 4, the test error first decreases as the model complexity increases, then increases because of overfitting, and then decreases again as the model complexity increases further (i.e. the model is even more overfit).
It seems to suggest that memorization eventually leads to better generalization.



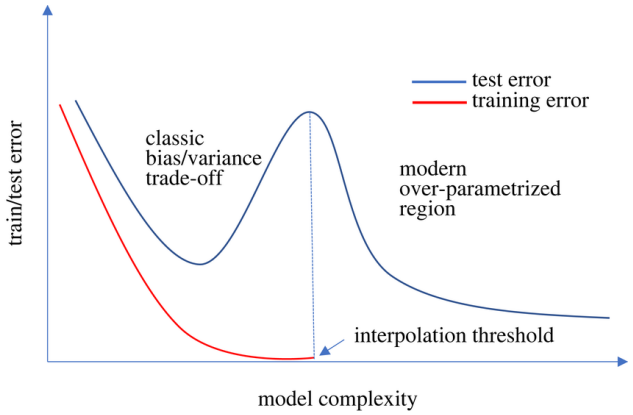


Figure 4: Double Descent

3 Learning Rate Scheduling

The fundamental gradient descent algorithm i.e.

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}_{\mathbf{w}}(\mathcal{D})$$

η being the learning rate, assumes a constant learning rate throughout the training process. In practice it has been observed that the convergence is better on decaying the learning rate, either in a linear way or an exponential way. This can be understood in the sense that we do not want to take long steps as we move closer to the minima. The following graph shows training accuracy vs learing rate.

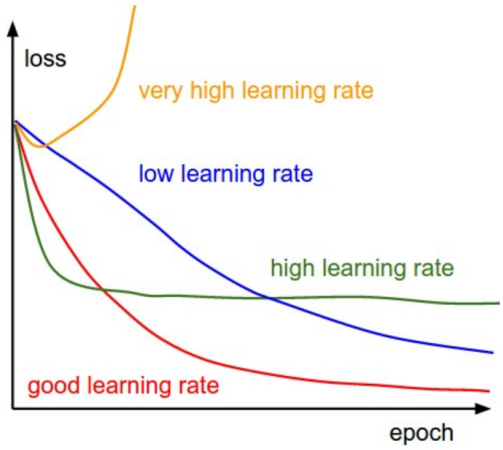


Figure 5: This diagram is taken from CS231n of Stanford University

As can be seen, high learning rates initially move faster but then do not get to the minima whereas low learning rates are too slow to converge. Very high learning rates can even lead to divergence. In between all these is an optimal learning rate schedule which moves at the right speed to the right minima. Adaptive optimizers such as **Adagrad** and **Adam** perform online learning rate scheduling by examining the gradients, with Adam being one of the best optimizers.
As a sidenote, **Simulated Annealing** on learning-rate can be used alongside gradient descent to act as a learning-rate scheduler.
In the next section, we will look at some popular optimizers and understand the concept of **momentum**.

4 Optimizers

Notation: Henceforth, $\mathbf{g}_t$ will denote the expression $\nabla_{\mathbf{w}}\mathcal{L}_{\mathbf{w}_{t-1}}(\mathcal{D})$

Let us first look at the drawbacks of vanilla gradient descent algorithm.

- In complex scenarios, there can be situations with high gradients which induces a high variance in the weight parameters and the loss function. This is detrimental to achieving the goal of the descent algorithm.
- If there are *plateau* regions, i.e. flat regions then the gradient in those regions will be close to zero and hence making no/very slow progress on descent. This can potentially occur when there are multiple undesirable local minima.

We will now make a "first" improvement to the vanilla SGD algorithm by using *momentum*.

4.1 SGD + Momentum Optimizer

Consider a number $\beta \in [0, 1]$. Define velocity terms $\mathbf{v}_t$ for each training step t as follows

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \eta \mathbf{g}_t \tag{1}$$

Redefine the weight update step as

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \mathbf{v}_t \tag{2}$$

To get an understanding of what velocity is actually contributing to, let us expand the RHS of the velocity update

$$\begin{aligned}
\mathbf{v}_t &= \eta \mathbf{g}_t + \beta \mathbf{v}_{t-1} \\
\mathbf{v}_t &= \eta \mathbf{g}_t + \beta(\eta \mathbf{g}_{t-1} + \beta \mathbf{v}_{t-2}) \\
&\vdots \\
\mathbf{v}_t &= \eta \mathbf{g}_t + \beta \eta \mathbf{g}_{t-1} + \beta^2 \eta \mathbf{g}_{t-2} + \cdots + \beta^t \mathbf{v}_0
\end{aligned}$$

If we set $\mathbf{v}_0$ to 0, then the velocity is just an exponentially weighted sum of all the gradients calculated till that point. Hence, it *smoothens* gradients and will aid in making movement even when there are flat regions. However, there is a chance of overshooting local minimas because velocity includes a weighted sum of gradients, which may not be zero at a local minima of the loss function.

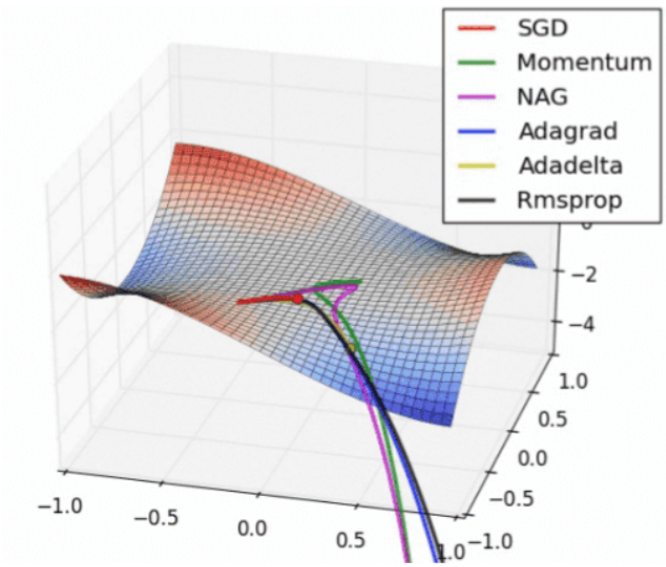


Figure 6: This diagram is by Alec Radford. It shows the paths taken by different optimizers during gradient descent.

4.2 Adagrad Optimizer

Adagrad is an online learning rate scheduler. It is also one of those few examples in machine learning where theoretical studies and empirical studies reinforce each other. Consider two weight parameters $\mathbf{w}_1$ and $\mathbf{w}_2$ such that

$$\nabla_{\mathbf{w}_1} \mathcal{L}_{\mathbf{w}}(\mathcal{D}) >> \nabla_{\mathbf{w}_2} \mathcal{L}_{\mathbf{w}}(\mathcal{D})$$

In such a case, using the same small learing rate for both $\mathbf{w}_1$ and $\mathbf{w}_2$ will cause a significant change in $\mathbf{w}_1$ and little or no change to $\mathbf{w}_2$. This is undesirable as large weight updates might be detrimental to the descent's progress. Hence, there is a need to have learning rates that adapt to parameters. On this note, let us define $\mathbf{s}_t$ for each training step such that

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t \tag{3}$$

where $\odot$ denotes element-wise product. This way, if the gradients $\mathbf{g}_t$ are large, then $\mathbf{s}_t$ will be large. The weight update step will now be

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t \tag{4}$$

ϵ is to prevent numerical overflows during the initial epochs if s_0 is set to 0.
Observe that the effective learning rate will be

$$\eta' = \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}}$$

That is, the parameters with high values of gradient will have low effective learning rate and the parameters with lower values of gradient will have higher effective learning rate. This is exactly what we desired. However, $\mathbf{s}_t$ monotonically increases with t, so the effective learning rate will decrease with t and it might be very slow to converge after a point. RMSProp Optimizer deals with this problem. Also, it has been observed that removing the square-root from the effective learning rate degrades the performance of the optimizer. Can you think of why is that so?

4.3 RMSProp Optimizer

This makes just one change to Adagrad, which is the following

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t \odot \mathbf{g}_t \tag{5}$$

where $\gamma \in [0, 1]$
Now, no longer does $\mathbf{s}_t$ monotonically increase with t. In fact, it relies more on the recent gradients, just like velocity. If the recent gradients are small, then $\mathbf{s}_t$ will be small and the effective learning rate will be higher, overcoming the problem in Adagrad.

Next, we look at the Adam Optimizer which combines the ideas of adaptive learning-rate scheduling along with momentum.

4.4 Adam Optimizer

Adam combines ideas from RMSProp and momentum. The definitions of s_t and v_t are

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t \odot \mathbf{g}_t$$
$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\mathbf{g}_t$$

where $\beta, \gamma \in [0, 1]$
Redefine the weight update as

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{v}_t \tag{6}$$

As you can see, the $\mathbf{v}_t$ part in the update rule is inspired from momentum and the effective learning-rate part is inspired from RMSProp. In the research paper which introduces Adam, there is one more layer of detail. Define $\hat{\mathbf{s}}_t$ and $\hat{\mathbf{v}}_t$ as

$$\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\gamma^t}$$
$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta^t}$$

The weight update will now be

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}} \odot \hat{\mathbf{v}}_t \tag{7}$$

The reason as to why bias corrections are applied is discussed in the next lecture.