Deep Learning - COSC2779

Neural Network Optimization

Dr. Ruwan Tennakoon

RMIT UNIVERSITY

Semester 2, 2022

Reference: *Chapter 7,8: Ian Goodfellow et. al., "Deep Learning", MIT Press, 2016.*

Outline

Machine Learning

The **Task** can be expressed an unknown target function:

$$\mathbf{y} = f\left(\mathbf{x}\right)$$

ML finds a Hypothesis (model), $h\left(\cdot\right)$, from hypothesis space $\mathcal{H}$, which approximates the unknown target function.

$$\hat{\mathbf{y}} = h^{*}\left(\mathbf{x}\right) \approx f\left(\mathbf{x}\right)$$

The **Experience** is typically a data set, $\mathcal{D}$, of values

$$\mathcal{D} = \left\{\left(\mathbf{x}^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right)\right\}_{i=1}^{N}$$

$*$Assume supervised learning for now

The **Performance** is typically numerical measure that determines how well the hypothesis matches the experience.

Last week we discussed about *a representation* of Hypothesis (model), $h\left(\cdot\right)$.

$$h\left(\mathbf{x}\right) = h^{(3)}\left(h^{(2)}\left(h^{(1)}\left(\mathbf{x}\right)\right)\right)$$

This week: How can we find the optimal hypothesis $h^{*}\left(\mathbf{x}\right)$?

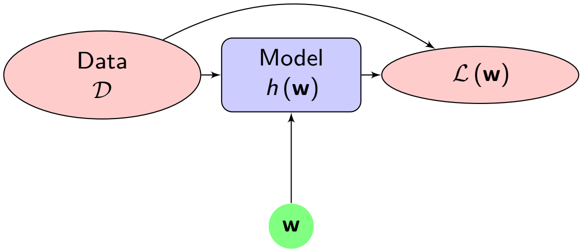
Objectives for this Lecture

- Explore techniques that can be used to find the optimal hypothesis in a NN.
- Understand the optimization techniques so that we can come up with the "best approach for a problem" in a way that is better than random or exhaustive search of the applicable techniques in the deep learning toolbox.
- Start understanding how to do evidence based debugging when the model is not learning.
- Applicable to all NN types, not just feed-forward.

Outline

Loss Function

;


$$\begin{tikzcd} \text{Data } \mathcal{D} \arrow[r] \arrow[rr, bend left] & \text{Model } h(\mathbf{w}) \arrow[r] & \mathcal{L}(\mathbf{w}) \\ & \mathbf{w} \arrow[u] & \end{tikzcd}$$

We would like to finding the parameters, **w**, of a neural network that reduce a cost function $\mathcal{L}(\mathbf{w})$.

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{L}(\mathbf{w})$$

Loss Function

We would like to finding the parameters, **w**, of a neural network that reduce a cost function $\mathcal{L}(\mathbf{w})$.

$$\mathbf{w}^* = \operatorname*{argmin}_{\mathbf{w}} \mathcal{L}(\mathbf{w})$$

The cost function usually consists of two parts. The loss and regularization terms.

$$\mathcal{L}(\mathbf{w}) = \underbrace{\mathbb{E}_{(\mathbf{x},y)\sim p_{data}} \mathrm{L}(y, h(\mathbf{x};\mathbf{w}))}_{\text{Risk}} + \lambda \underbrace{\mathrm{R}(\mathbf{w})}_{\text{Regularization}}$$

Here p_{data} is the data-generating distribution. $\mathrm{L}()$ is some function that quantify the deviation from expected result.

This is an optimization problem. But, in ML, We do not know p_{data}, Therefore we cannot minimize **risk**.

$*$Out main attention in these slides is given to supervised learning.

Loss Function

In ML, We do not know p_{data}. Therefore we minimize the empirical risk. Minimize the expected loss on the training set.

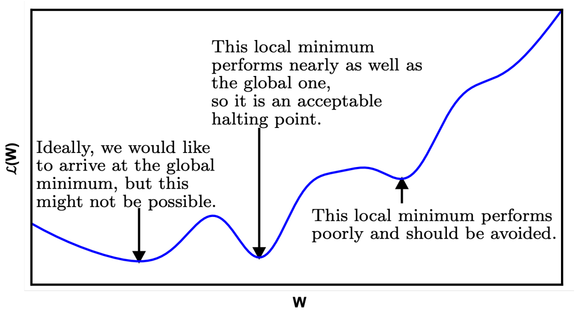
$$\mathcal{L}(\mathbf{w}) = \underbrace{\mathbb{E}_{(\mathbf{x},y)\sim\hat{p}_{data}} \mathrm{L}\left(y, h\left(\mathbf{x};\mathbf{w}\right)\right)}_{\text{Empirical Risk}} + \lambda \mathrm{R}\left(\mathbf{w}\right)$$

Here $\hat{p}_{data}$ is the training data distribution.

$$\mathbb{E}_{(\mathbf{x},y)\sim\hat{p}_{data}} \mathrm{L}\left(y, h\left(\mathbf{x};\mathbf{w}\right)\right) = \frac{1}{N}\sum_{i=1}^{N} \mathrm{L}\left(y^{(i)}, h\left(\mathbf{x}^{(i)};\mathbf{w}\right)\right)$$

The training process based on minimizing this average training error is known as **empirical risk minimization**.

Loss Function

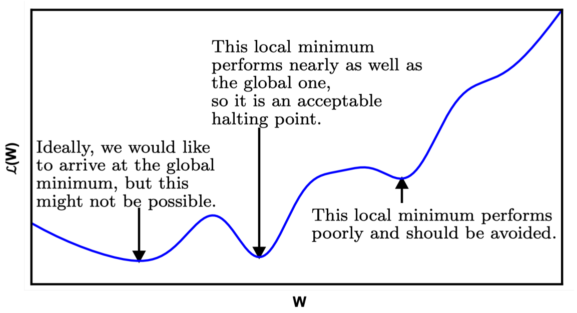


The derivative of the loss function is zero ($\frac{\partial \mathcal{L}(w)}{\partial w} = 0$) at any critical point in the loss function.

- A **local minimum** is a point where $\mathcal{L}(w)$ is lower than at all neighboring points.
- A point that obtains the absolute lowest value of $\mathcal{L}(w)$ is a **global minimum**.

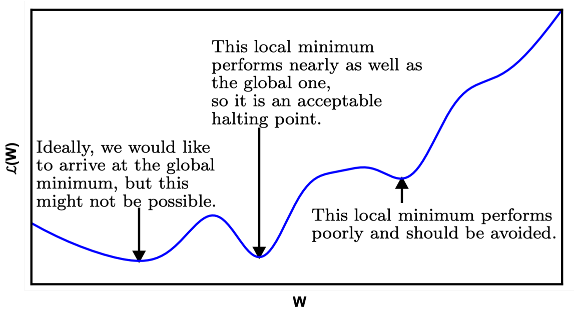
If the loss is **convex** we have only one minimum - The global minimum.

Finding the Minimum


This local minimum performs nearly as well as the global one, so it is an acceptable halting point.
Ideally, we would like to arrive at the global minimum, but this might not be possible.
This local minimum performs poorly and should be avoided.
$\mathcal{L}(\mathbf{W})$
W

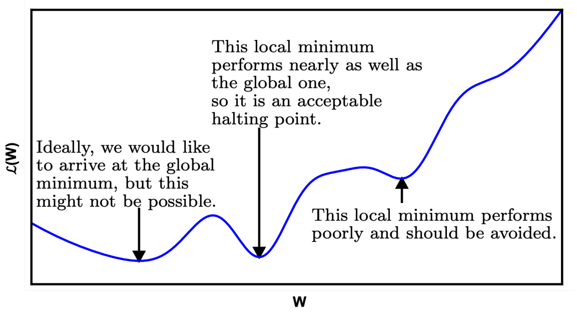
- Random search - Bad idea.

Finding the Minimum



- Random search - Bad idea.
- Solve $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = 0$ and find a closed form solution. Not applicable for many cases.

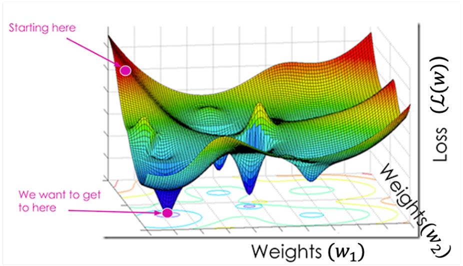
Finding the Minimum



- Random search - Bad idea.
- Solve $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = 0$ and find a closed form solution. Not applicable for many cases.
- Guided Search - Apply iterative method - Gradient decent: $\mathbf{w}^{[t]} = \mathbf{w}^{[t-1]} - \alpha\, \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$

Gradient descent

Gradient Decent



Note that $\mathbf{w} = [w_0, w_1, \cdots, w_m]$

Algorithm 1: Basic Gradient Decent

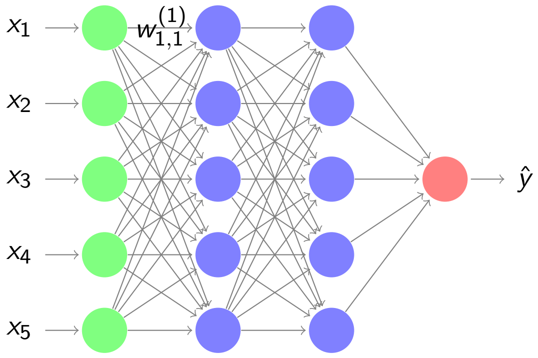
```
Result: Final Weights $\mathbf{w}$
Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$;
while not converged do
    Compute Gradients, $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$ ;
    Update weights, $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$ ;
end
```

α is the learning rate.

How can we efficiently apply gradient decent to Neural networks?

Outline

Computing Gradients in Neural Networks

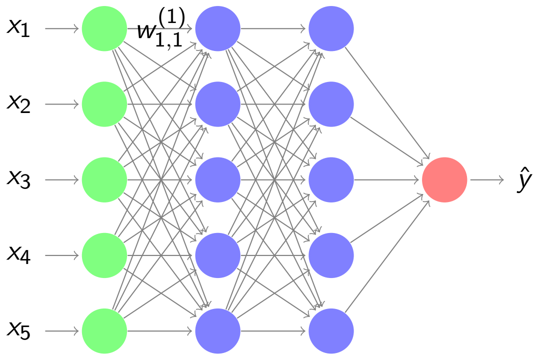


The gradient, $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$, is simply the vector of partial derivatives in each dimension.

How can we calculate $\frac{\partial}{\partial w^{(1)}_{1,1}} \mathcal{L}(\mathbf{w})$?

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \left[\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}^{(1)}_{1,1}}, \frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}^{(1)}_{1,2}}, \dots \right]$$

Computing Gradients in Neural Networks



The gradient, $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$, is simply the vector of partial derivatives in each dimension.

How can we calculate $\frac{\partial}{\partial w_{1,1}^{(1)}} \mathcal{L}(\mathbf{w})$?

Numerical derivatives (finite difference approximation):

$$\frac{\partial}{\partial w_{1,1}^{(1)}} \mathcal{L}(\mathbf{w}) = \lim_{\delta \to 0} \frac{\mathcal{L}\left(w_{1,1}^{(1)} + \delta\right) - \mathcal{L}\left(w_{1,1}^{(1)}\right)}{\delta}$$

$*$Other elements of $\mathbf{w}$ will remain constant.

How about $\frac{\partial}{\partial w_{1,2}^{(1)}} \mathcal{L}(\mathbf{w})$?

This looks expensive, if we have millions of weights.

Back-Propagation

Back-Propagation is a computationally efficient method to calculate derivatives.

Repeated application of the chain-rule.

Lets have a look at the intuition behind back-prop using a simple neural network.

A good explanation of back-prop is in Calculus on Computational Graphs: Backpropagation

$$\begin{tikzcd} x \arrow[r, "w^{(1)}"] & h_1 \arrow[r, "w^{(2)}"] & h_2 \arrow[r, "w^{(3)}"] & \hat{y} \arrow[r] & \mathcal{L}(w) \end{tikzcd}$$

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w^{(3)}} = \frac{\partial \mathcal{L}(\mathbf{w})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w^{(3)}}$$

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w^{(2)}} = \underbrace{\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \hat{y}}}_{} \times \frac{\partial \hat{y}}{\partial h_2} \times \frac{\partial h_2}{\partial w^{(2)}}$$

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w^{(1)}} = \underbrace{\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial h_2}}_{} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_1}{\partial w^{(1)}}$$

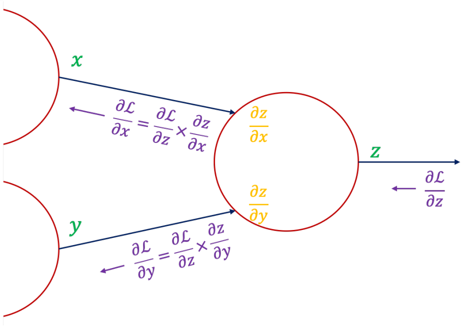
Back-Propagation

Back-Propagation is a computationally efficient method to calculate derivatives.

Repeated application of the chain-rule.

Lets have a look at the intuition behind back-prop using a simple neural network.

A good explanation of back-prop is in Calculus on Computational Graphs: Backpropagation



x
y
z
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial \mathcal{L}}{\partial z}$
$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial z} \times \frac{\partial z}{\partial x}$
$\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial z} \times \frac{\partial z}{\partial y}$

Implementation - TensorFlow

In TensorFlow 2.0, the GradientTape looks after the gradient calculations. We only need to do the forward pass.

There is also a much simpler API in TensorFlow that does all of this under-the-hood. More on this in the lab.

```
import tensorflow as tf

w = tf.Variable([tf.random.normal()])
lr = 0.001

while True: # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(w)  #forward function defined outside
        gradient = g.gradient(loss, w)

    w = w - lr*gradient
    # need to check convergence and break loop
```

Revision Questions

1. Why cannot we directly minimize the risk in ML?
2. What is the difference between the notations $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$ and $\frac{\partial}{\partial w}\mathcal{L}(\mathbf{w})$?
3. Will gradient decent find the weights that minimize the cost function in deep feed-forward NN?
4. When calculating $\frac{\partial}{\partial w_{1,1}^{(1)}}\mathcal{L}(\mathbf{w})$, what valaues will you use for other weights (e.g. $w_{1,2}^{(1)}$)?

Outline

Approximate Derivatives

Calculating the gradients of the empirical risk.

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \nabla_{\mathbf{w}} \mathbb{E}_{(\mathbf{x},y)\sim \hat{p}_{data}} \mathrm{L}(y, h(\mathbf{x};\mathbf{w})) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\mathbf{w}} \mathrm{L}\left(y^{(i)}, h\left(\mathbf{x}^{(i)};\mathbf{w}\right)\right)$$

Computing this expectation exactly is **very expensive** because it requires evaluating the model on every example in the entire dataset.

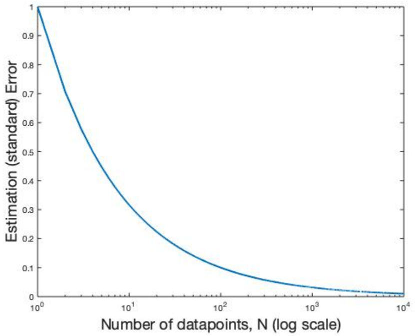
In practice, we can compute these expectations by randomly sampling a small number of examples from the dataset, then taking the average over only those examples.

$$\nabla_{\mathbf{w}} \mathbb{E}_{(\mathbf{x},y)\sim \hat{p}_{data}} \mathrm{L}(y, h(\mathbf{x};\mathbf{w})) \approx \frac{1}{N_b}\sum_{i=1}^{N_b} \nabla_{\mathbf{w}} \mathrm{L}\left(y^{(i)}, h\left(\mathbf{x}^{(i)};\mathbf{w}\right)\right)$$

Approximate Derivatives

Why does approximate derivatives work (intuition):

- Standard error of the mean estimated with n samples is $\sigma/\sqrt{n}$. Reduction in standard error with the increase in n diminishes with n.



Approximate Derivatives

Why does approximate derivatives work (intuition):

- Standard error of the mean estimated with n samples is $\sigma/\sqrt{n}$. Reduction in standard error with the increase in n diminishes with n.
- Redundancy in the training set. Some data-points will result in the same values.
- We just need an approximate direction to step. Will work as long as the gradients are not totally random.

Gradient decent methods:

- **Deterministic**: Uses all the training data when calculating gradient for each step.
- **Stochastic** (mini-batch): Use randomly sampled small number of examples from training data when calculating gradient for each step

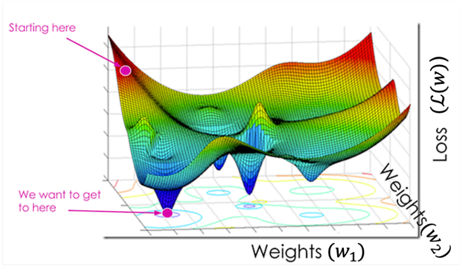
Determining Batch-Size

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns
- Multi-core architectures are usually underutilized by extremely small batches.
- Amount of memory scales with the batch size.
- When using GPUs, it is common for power of 2 batch sizes to offer better run time.
- Small batches can offer a regularizing effect.

The general inefficiency of batch training for gradient descent learning

We also wish for two subsequent gradient estimates to be independent from each other, so two subsequent mini-batches of examples should also be independent from each other.

Stochastic (Mini-Batch) Gradient Descent



Note that $\mathbf{w} = [w_0, w_1, \cdots, w_m]$

Algorithm 2: Stochastic Gradient Decent

```
Input: Training data $\mathcal{D}$, Learning rate schedule
Result: Final Weights $\mathbf{w}$
Initialize weights randomly;
$t \leftarrow 1$ ;
while not converged do
    Sample an iid batch from training data, $\mathcal{D}_b \subset \mathcal{D}$ ;
    Compute Gradients using $\mathcal{D}_b$, $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$ ;
    Update weights, $\mathbf{w} \leftarrow \mathbf{w} - \alpha_t \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$ ;
    $t \leftarrow t + 1$;
    Check for convergence ;
end
```

α_t is the learning rate at iteration t.

Implementation - TensorFlow

An Epoch is one pass through all the training data.

The data should be **shuffled** after each epoch to make sure the gradients are independent. This should be handled in the data generator.

More in labs.

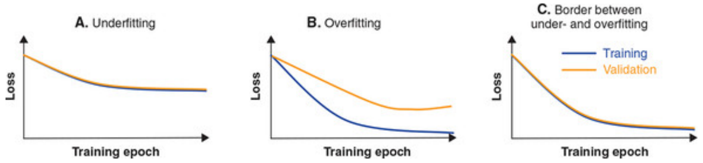
```
import tensorflow as tf

model = MNISTModel() # defined outside
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
train_loss = tf.keras.metrics.Mean(name='train_loss')

EPOCHS = 5
for epoch in range(EPOCHS):
    # iterate over all batches in the dataset
    for image_batch, label_batch in mnist_train:
        with tf.GradientTape() as tape:
            predictions = model(image_batch)
            loss = loss_object(label_batch, predictions)

        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

        train_loss(loss)

    print('Epoch ', epoch, ' Train loss: ', train_loss.result())
    train_loss.reset_states()
```

Do we need anything else or, are we done?

Aside: Learning Curves

Learning curves are a powerful tool to understand the behaviour of NN training.

More on plotting learning curve in labs.



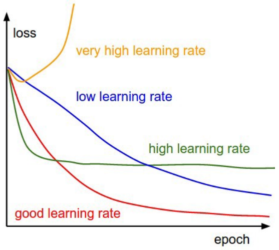


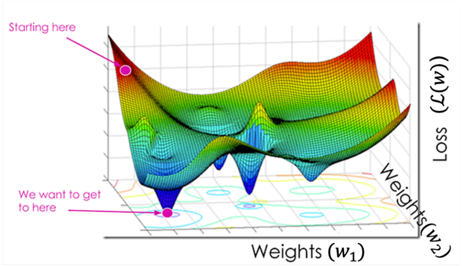
Image: Standford cs231n

Revision Questions

1. What are the advantages of using a small batch size?
2. Why do we need to shuffle data between epoch?
3. Why is there a negative sign in the weight update of the SGD?

Outline

Local Minima



With **non-convex** functions, such as neural nets, it is possible to have many local minima.

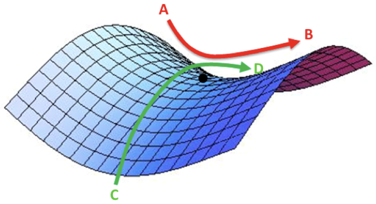
Local minima can be problematic if they have high cost in comparison to the global minimum.

"For large neural networks, most local minima have a low cost value, and that it is not important to find a true global minimum rather than to find weights with low cost."

To rule out local minima: plot the norm of the gradient over time.

- If the norm of the gradient does not shrink, the problem is not any kind of critical point.
- Positively establishing that local minima are the problem can be very difficult in DNN.

Saddle Points



A saddle point has a local minimum along one cross-section of the cost function and a local maximum along another cross-section.

- In higher-dimensional spaces, local minima are rare, and saddle points are more common.
- For first-order optimization, the gradient can often become very small near a saddle point/Flat Regions.
- Second-order methods (Newton's method) can terminate at saddle points.

Flat Regions, Cliffs and Exploding Gradients

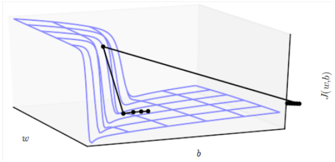


Image: Goodfellow 2016

- Neural networks with many layers often have extremely steep regions resembling cliffs.
- On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far.
- Serious consequences can be avoided using the gradient clipping. This prevents any gradient to have norm greater than a threshold.

Long-Term Dependencies

Vanishing and **Exploding** gradient problem.

Arises when

- The computational graph becomes extremely deep.
- Repeatedly applying the same operation at each time step of a long temporal sequence (in RNN discussed later)

Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable.

How can we solve these problems?

Outline

Stochastic Gradient Descent - SGD

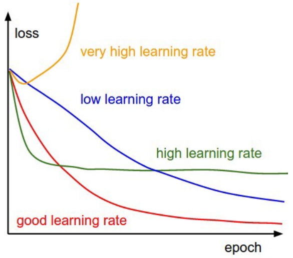


Image: Standford cs231n

We already discussed SGD.
The main parameter is learning rate. This can be a fixed value (α) or can vary in time. A simple linear learning rate schedule is below:

$$\alpha_t = \left(1 - \frac{t}{\tau}\right)\alpha_0 + \frac{t}{\tau}\alpha_\tau$$

α_0 initial learning rate, α_τ final learning rate, τ number of iterations.

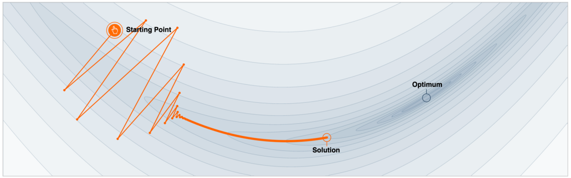
The learning rate may be chosen by trial and error. Best to choose it by monitoring learning curves.

- If it is too large, the learning curve will show violent oscillations,and often increasing significantly.
- If the learning rate is too low, learning proceeds slowly.

Momentum

Normal gradient decent update:

$$\mathbf{w}^{[t]} \leftarrow \mathbf{w}^{[t-1]} - \alpha_t \, \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$$



Pathological curvature: regions of cost function which are not scaled properly.

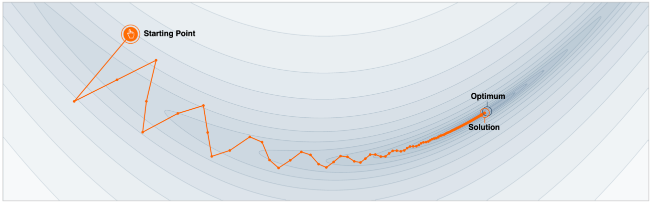
The iterates either jump between valleys, or approach the optimum in small steps.

Momentum

Momentum give gradient descent a short-term memory.

$$\mathbf{z}^{[t]} \leftarrow \beta \mathbf{z}^{[t-1]} - (1-\beta)\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$$
$$\mathbf{w}^{[t]} \leftarrow \mathbf{w}^{[t-1]} + \alpha \mathbf{z}^{[t]}$$

Hyper-parameter β controls how quickly the contributions of previous gradients exponentially decay. A typical value is $\beta = 0.9$.



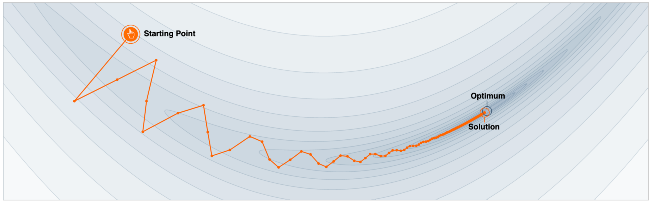
Why Momentum Really Works
On the importance of initialization and momentum in deep learning.

Nesterov Momentum

Variants: Nesterov momentum. The gradient is evaluated after the current velocity is applied.

$$\mathbf{z}^{[t]} \leftarrow \beta \mathbf{z}^{[t-1]} - (1-\beta)\nabla_{\mathbf{w}}\mathcal{L}\left(\mathbf{w} + \beta \mathbf{z}^{[t-1]}\right)$$
$$\mathbf{w}^{[t]} \leftarrow \mathbf{w}^{[t-1]} + \alpha_t \mathbf{z}^{[t]}$$

Hyper-parameter β controls how quickly the contributions of previous gradients exponentially decay.



Why Momentum Really Works
On the importance of initialization and momentum in deep learning.

Revision Questions

- How can the problem of saddle points be minimized first-order methods?
- What happens if $\beta = 0$ in momentum gradient descent.
- Why is it a good idea to reduce the learning rate with iterations in SGD?

Outline

Adaptive Learning Rate

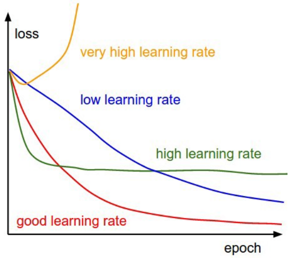


Image: Standford cs231n

Learning rate is a key parameter in gradient descent.

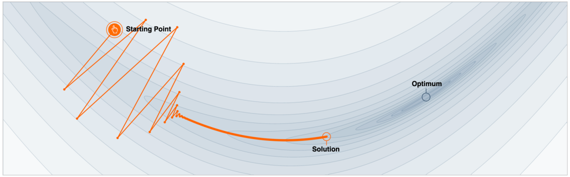
- If the learning rate is too small, updates are small and optimization is slow.
- If the learning rate is too large, updates will be large and the optimization is likely to diverge.

Start with a large learning rate and **decay** it during training.

Finding the “best decay schedule” is not trivial. **Adaptive learning-rate algorithms** such as Adam and RMSprop can adjust the learning rate during the optimization process.

RMSprop

Normal gradient decent update: $\mathbf{w}^{[t]} \leftarrow \mathbf{w}^{[t-1]} - \alpha_t \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$



Can have oscillation: Slow in one direction and speed up in other. RMSprop use the root mean square of the gradient in each direction to scale the update.

$$r^{[t]} = \rho r^{[t-1]} + (1-\rho)\left(\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})\right)^2$$

$$\mathbf{w}^{[t]} \leftarrow \mathbf{w}^{[t-1]} - \alpha \frac{\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})}{\sqrt{r^{[t]} + \epsilon}}$$

ϵ is a small value to prevent division by zero ($\epsilon = 10^{-8}$). $r^{[t]}$ is a vector. Division and square applied element-wise

Adam

Combines RMSprop and momentum. ADAM: A Method For Stochastic Optimization

$$r^{[t]} = \rho r^{[t-1]} + (1-\rho)\left(\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})\right)^2 \quad \triangleright \text{RMSprop}$$

$$\mathbf{z}^{[t]} \leftarrow \beta \mathbf{z}^{[t-1]} + (1-\beta)\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) \quad \triangleright \text{Momentum}$$

$$r^{[t]} \leftarrow r^{[t]}/(1-\rho)\ ;\ \mathbf{z}^{[t]} \leftarrow \mathbf{z}^{[t]}/(1-\beta) \quad \triangleright \text{Bias correction}$$

$$\mathbf{w}^{[t]} \leftarrow \mathbf{w}^{[t-1]} - \alpha \frac{\mathbf{z}^{[t]}}{\sqrt{r^{[t]}} + \epsilon}$$

ϵ is a small value to prevent division by zero ($\epsilon = 10^{-8}$). $s^{[t]}$ is a vector. Division and square applied element-wise

Works with wide range of problems. Hyper parameters $[\beta, \rho]$ are named $[\beta_1, \beta_2]$ in tensorflow and usually are set to $\beta_1 = 0.9, \beta_2 = 0.999$.

Outline

Choosing the Right Optimization Algorithm

Unfortunately, there is currently no consensus on which algorithm to use.

SGD	Gradient descent can use parallelization efficiently. SGD usually converges faster than GD on large datasets. Of the optimizers profiled here, SGD uses the least memory for given batch size.
Momentum	Momentum usually speeds up the learning. Momentum uses more memory for a given batch size than stochastic gradient descent but less than RMSprop and Adam.
RMSprop	RMSprop's adaptive learning rate usually prevents the learning rate decay from diminishing too slowly or too fast. RMSprop maintains per-parameter learning rates.
Adam	The hyperparameters of Adam are usually set to predefined values. Adam performs a form of learning rate annealing with adaptive step-sizes. Adam is often the default optimizer in machine learning.

If your input data is sparse, then you likely achieve the best results using one of the adaptive learning-rate methods.

An overview of gradient descent optimization algorithms

Beyond Gradient Descent

Outline

Parameter Initialization

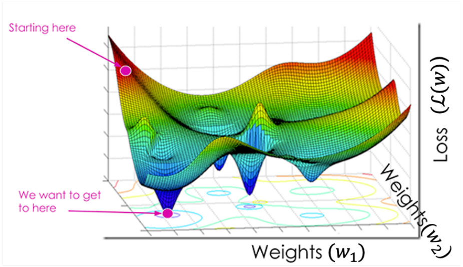
Deep networks has many parameters:

$$h(\mathbf{x}) = h^{(3)}\left(h^{(2)}\left(h^{(1)}\left(\mathbf{x}; \mathbf{W}^{(1)}\right); \mathbf{W}^{(2)}\right); \mathbf{W}^{(3)}\right)$$

Gradient descent learning is iterative and thus require the user to specify some initial point.

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha_t \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$$

Starting place of gradient descent can be critical to the model's ultimate performance.



Parameter Initialization

The initial point can determine whether the algorithm converges:

- Some initial points can be unstable and the algorithm may fails altogether.
- Initial point can determine how quickly learning converges.
- local minimums can have wildly varying generalization error, and the initial point can affect which minimum is selected.

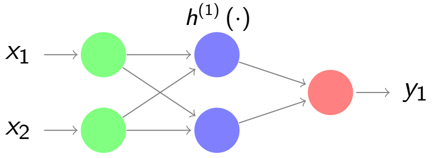
The most important consideration when initializing the weights is that the **initial parameters need to "break symmetry" between different units**.

Initializing: Break Symmetry

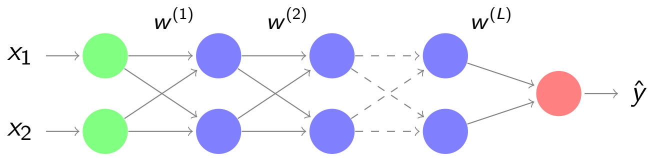
Initial parameters need to "break symmetry" between different units.

- If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters.
- If they have the same initial parameters, then a learning algorithm will constantly update both of these units in the same way.

This motivates random initialization of the parameters.



Random Initialization



Can initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution.

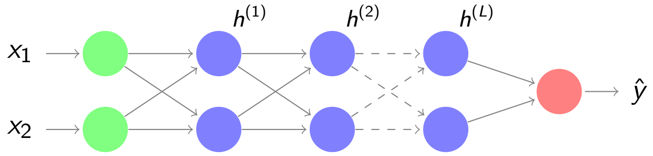
The scale of the initial distribution, have a large effect on the outcome of the optimization procedure.

- **Too small:** vanishing gradients, learning too slow.
- **Too large:** exploding gradients, unstable.

Assuming all activation's are linear.

$$\hat{y} = w^{(L)} w^{(L-1)} \dots w^{(2)} w^{(1)} x$$

Initialization: Now to select scale



Rule of thumb to prevent exploding/vanishing gradients:

- The mean of the activations should be zero.

$$\mathbb{E}\left(h^{(l-1)}\right) = \mathbb{E}\left(h^{(l)}\right) = 0$$

- The variance of the activations should stay the same across every layer.

$$\mathrm{Var}\left(h^{(l-1)}\right) = \mathrm{Var}\left(h^{(l)}\right)$$

Xavier initialization

All the weights of layer ll are picked randomly from a normal distribution with mean $\mu = 0$ and variance $\sigma^2 = \frac{1}{n^{[l-1]}}$ where $n^{[l-1]}$ is the number of neuron in layer $l - 1$.

$$\mathbf{W}^{(l)} \sim \mathcal{N}\left(0, \frac{1}{n^{[l-1]}}\right)$$

Biases are initialized with zeros.

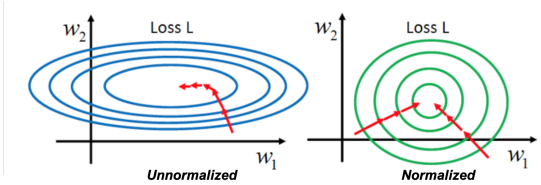
$$\mathbf{b}^{(l)} = 0$$

The mathematics of why this works is explained in Deeplearning.ai notes
A varient of Xavier, that works well with “ReLU” is provided in He normalization

Outline

Input Normalization

Generally we know that normalizing the input features help to speedup learning.

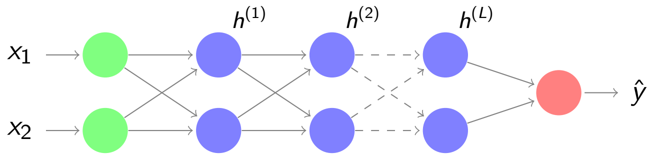


$$\mu_j = \frac{1}{N}\sum_i x_j^{(i)}; \quad \sigma_j^2 = \frac{1}{N}\sum_i \left(x_j^{(i)} - \mu_j\right)^2$$

$$\tilde{x}_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$

Batch Normalization

How about hidden layer activations?



Can we normalize input to each layer?

$$\mathbf{z}^{(i)} = \mathbf{W}^{(l)}\mathbf{x}^{(i)} + \mathbf{b}^{(l)} \quad \triangleright \text{Affine transform}$$

$$h^{(i)} = g\left(\mathbf{z}^{(i)}\right) \quad \triangleright \text{Activation}$$

Typically we normalize $\mathbf{z}^{(i)}$.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Batch Normalization

$$\mu_j = \frac{1}{m}\sum_i z_j^{(i)}; \quad \sigma_j^2 = \frac{1}{m}\sum_i \left(z_j^{(i)} - \mu_j\right)^2$$

$$\hat{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \quad \triangleright \text{mean zero, variance } 1$$

$$\tilde{z}_j^{(i)} = \gamma \hat{z}_j^{(i)} + \beta \quad \triangleright \gamma, \beta \text{ learnable parameters}$$

Batch Normalization allows us to normalize the feature values across a mini-batch.

The mean and variance is computed over the mini-batch.

The bias in affine transform is not useful when using batch-norm:
$\mathbf{z}^{(i)} = \mathbf{W}^{(l)}\mathbf{x}^{(i)} + \cancel{\mathbf{b}^{(l)}}$

Batch Normalization: Inference

At test time you do not have a batch to compute mean and variance.

Keep a weighted average over mini-batches while training.

$$\hat{\mu}_j = \rho\ \hat{\mu}_j + (1 - \rho)\, \mu_j$$
$$\hat{\sigma}_j^2 = \rho\ \hat{\sigma}_j^2 + (1 - \rho)\, \sigma_j^2$$

Apply the moving averages while testing.

$$\hat{z}_j^{(i)} = \frac{z_j^{(i)} - \hat{\mu}_j}{\sqrt{\hat{\sigma}_j^2 + \epsilon}}$$

$$\tilde{z}_j^{(i)} = \gamma \hat{z}_j^{(i)} + \beta$$

Revision Questions

1. What happens if the weights in a neural network are all initialized to 0?
2. Why should we not have a bias in neural network layers when batch-norm is used?
3. Should batch-norm be used before or after the activation?

Summary

1. Neural networks are usually trained using iterative, gradient based methods.
2. There are many gradient based optimization techniques. Methods with adaptive learning rate seems to be robust to many problems.
3. Initialization and batch normalization help speed up learning.

Lab: Continue to learn how NN can be implemented in TensorFlow.

Next week:

1. Convolutions