Multi-Layer Perceptron (MLP)
Training Issues II

M. R. Hasan
The Art & Science of Training Deep MLPs

- We will discuss some key issues regarding the training of MLPs.

- Some issues concern deep MLPs (with many hidden layers and neurons) or more generally deep neural networks (DNNs).
Issues Regarding Training Deep MLPs

- We will discuss the following issues.
  - Feature scaling
  - Intractability of computation
  - Computational complexity
  - Overfitting due to complex architecture
  - Overfitting due to overtraining
  - Weight initialization
  - Hidden Layer Activation Functions
  - Optimizing Stochastic Gradient Descent
  - Prediction invariance
  - Scalability with respect to the input size
Training Issues: Weight Initialization
Training Issues: Weight Initialization

• The main goal of the Backpropagation algorithm is to update the weights of the neurons iteratively.
• But to start the iteration, weights need to be initialized.
• Should we initialize the weights with zero values similar to Logistic/Linear Regression?
• Unlike Linear/Logistic Regression, MLP weights shouldn’t be initialized with 0s.
• Why not?

\[ W^{(k)} = W^{(k)} - \eta \nabla \mathcal{L}(W^{(k)}) \]
Training Issues: Weight Initialization

- If we set all weights to 0:
- Forward propagation: all outputs and subsequent activation signals will be 0s.
- Backward propagation: the derivative with respect to the loss function would be the same for every weight parameter.
- This makes the hidden neurons symmetric and continues for all the iterations we run.

\[
\begin{align*}
\nabla \mathcal{L}(W^{(k-1)}) &= a^{(k-1)^T} \delta^{(k)} \\
\delta^{(k)} &= \left( \delta^{(k+1)} W^{(k)T}_{\text{no bias}} \right) \ast g'(z^{(k)})
\end{align*}
\]
Training Issues: Weight Initialization

• In other words, all neurons in a given layer will be **perfectly identical**.

• Thus, despite having many of neurons per layer, the MLP will act as if it has **only one neuron per layer**.

• Setting weights to zero makes our network **no better than a linear model**.

• All neurons will be symmetric!
Training Issues: Weight Initialization

- Notice, that the weights are sandwiched between output (a) of one layer (k-1) and the error of the next layer (k).
- If W is set to zero, then either the error or the signal would be zero (resulting into a symmetric situation).

\[
\frac{\partial L_n}{\partial W_{ij}^{(k-1)}} = \frac{\partial L_n}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial W_{ij}^{(k-1)}} = \delta_j^{(k)} a_i^{(k-1)}
\]

This symmetry needs to be broken.
Training Issues: Weight Initialization

• For breaking this symmetry, the weights need to be **randomly initialized with non-zero values**.

• Moreover, we want each neuron to compute a **different function** from all other neurons.

The idea of random weight initialization was proposed by **Rumelhart**!
they discussed the idea, he told Rumelhart this mathematical trick would never work. After all, he said, Frank Rosenblatt, the man who designed the Perceptron, had proven it would never work. If you built a neural network and you set all the weights to zero, the system could learn to adjust them on its own, sending changes cascading down the many layers. But in the end, each weight would wind up at the same place as all the rest. However much you tried to get the system to adopt relative weighting, its natural tendency was to even things out. As Frank Rosenblatt had shown, this was just how the math behaved. In the vernacular of mathematics, the system couldn’t “break symmetry.” One neuron could never be more important than any other, and that was a problem. It meant that this neural network wasn’t any better than the Perceptron.

Rumelhart listened to Hinton’s objection. Then he made a suggestion. “What if you didn’t set the weights to zero?” he asked. “What if the numbers were random?” If all the weights held different values at the beginning, he suggested, the math would behave differently. It wouldn’t even out all the weights. It would find the weights that allowed the system to actually recognize complex patterns, such as a photo of a dog.

Hinton liked to say that “old ideas are new”—that scientists should never give up on an idea unless someone had proven it wouldn’t work. Twenty years earlier, Rosenblatt had proven that backpropagation wouldn’t work, so Hinton gave up on it. Then Rumelhart made this small suggestion. Over the next several weeks, the two men got to work building a system that began with random weights, and it could break symmetry. It could assign a different weight to each neuron. And in setting these weights, the system could actually recognize patterns in images. These were simple images. The system couldn’t recognize a dog or a cat or a car, but thanks to backpropagation, it could now handle that thing called
Training Issues: Weight Initialization

• Random weight initialization will help to ensure that
  - no input patterns are lost during forward propagation,
  - nor any gradient pattern will be lost during backward propagation.
Training Issues: 
Weight Initialization

- We make this possible by initializing each weight \textbf{randomly}.
- However these random values \textbf{need to be small}.
- Why?
- Say that the hidden layer activation functions are \textbf{sigmoid-like}.
- Initial small random values ensure that the \textbf{sigmoids are not saturated}.

What happens if we choose \textbf{large random} values for initialization?

Forward Propagation

\[ z_j^{(k)} = a_j^{(k-1)} w^{(k-1)} \]

\[ a_j^{(k)} = g(z_j^{(k)}) \]
If larger weight values (>1) are used for initialization, then the sigmoid output $a_j^{(k)}$ will be saturated.

As a result, the derivative of sigmoid (red curve) will be very small, i.e., $g'(.) \to 0$

Consequently: $\delta^{(k)} \to 0$

This will cause very small loss gradient: $\nabla \mathcal{L}(W^{(k)}) \to 0$
Due to the **tiny loss gradients** $\nabla \mathcal{L}(W^{(k)}) \to 0$, we will have tiny weight updates.

\[ W^{(k)} = W^{(k)} - \eta \nabla \mathcal{L}(W^{(k)}) \]
Training Issues: Weight Initialization

- In MLPs, those tiny gradients get multiplied many times during backward propagation.
- Gradients after many multiplications become small and stop making any influence.
- Learning will stop (as gradients will vanish)!

This problem is known as the notorious Vanishing gradient due to initialization with large random numbers.
Training Issues: Weight Initialization

• Thus, it is a usual practice to initialize weight parameters around zero value.
• This is done to ensure that we get reasonable gradients (close to 1) to train the MLP.

But what should be the range for the initial values of the weight parameters?
Training Issues: Weight Initialization

- From the plot of the derivative of sigmoid (red curve), we see the gradient is larger between the input -1 to +1.
- Thus, for random weight initialization we should choose random numbers between -1 to +1.
Training Issues: Weight Initialization

• Typically this is done either by sampling from:
  - a **uniform distribution** between limits -1 and +1, or
  - a **normal distribution** with mean 0 and variance 1.

This initialization technique is known as the **random weight initialization** technique.
Training Issues: Weight Initialization

• How do we initialize the biases?
• Usually biases are initialized by zeros.
• Their zero values don’t prevent symmetry breaking.
• The small random values in the weight parameters are sufficient to break the symmetry, as discussed earlier.

(a) No bias
(b) Bias weight added
Training Issues: Weight Initialization

- See the notebook “Artificial Neural Network-I-Initialization of the Weights” for an empirical understanding of the weight initialization phenomenon:
Issues Regarding Training Deep MLPs

• We will discuss the following issues.
  - Feature scaling
  - Intractability of computation
  - Computational complexity
  - Overfitting due to complex architecture
  - Overfitting due to overtraining
  - Weight initialization
  - Hidden Layer Activation Functions
  - Optimizing Stochastic Gradient Descent
  - Prediction invariance
  - Scalability with respect to the input size
Training Issues: Hidden Layer Activation Function
Artificial Neural Network

• Previously we mentioned that an ANN uses activation functions that are continuously nonlinear for converting the weighted sum of the input.

\[
y(\mathbf{x}, \mathbf{w}) = g \left( \sum_i \mathbf{w}_i^T \mathbf{x}_i \right)
\]

We will now discuss various choices for the activation function.
Artificial Neural Network

• Sigmoid or logistic function.
• The logistic sigmoid is motivated somewhat by biological neurons.
• It can be interpreted as the probability of an artificial neuron “firing” given its inputs.

\[ g(a) \equiv \sigma(a) \]

\[ \sigma(a) = \frac{1}{1 + e^{-a}} \]
Artificial Neural Network

- Sigmoid or logistic function.

\[ g(a) \equiv \sigma(a) \]

\[ \sigma(a) = \frac{1}{1 + e^{-a}} \]

The derivative of sigmoid:

\[ g'(a) = \frac{dg}{da} = \sigma(a)[1 - \sigma(a)] \]
Artificial Neural Network

- The logistic sigmoid has a nice **biological interpretation**.
- However, it turns out that the **logistic sigmoid** can cause a neural network to **get “stuck”** during training.

\[ \sigma(a) = \frac{1}{1 + e^{-a}} \]
Artificial Neural Network

- This is due in part to the fact that if a strongly-negative input is provided to the logistic sigmoid, it outputs values very near zero.

\[ \sigma(a) = \frac{1}{1 + e^{-a}} \]
Artificial Neural Network

- Note that ANNs use the **feed-forward activations** to calculate parameter gradients.
- Zero output from sigmoid can result in $W^{(k)}$s that are updated less regularly than we would like.

Thus, they could get **“stuck”** in their current state.

$$W^{(k)} = W^{(k)} - \eta \nabla \mathcal{L}(W^{(k)})$$

$$a_{j}^{(k-1)} = g(z_{j}^{(k-1)})$$

$$\nabla \mathcal{L}(W^{(k-1)}) = \alpha^{(k-1)^T} \delta^{(k)}$$
Artificial Neural Network

• An alternative to the logistic sigmoid is the **hyperbolic tangent**, or tanh function.
• The tan is a **trigonometric** function that relates to a **circle**.
• The tanh is a **hyperbolic** function related to a **hyperbola**.

Tanh creates sigmoid-like “S” shaped curve!
Artificial Neural Network

• The tanh function is the ratio of hyperbolic sine and cosine functions.

\[ g(a) \equiv tanh(a) \]

\[ tanh(a) = \frac{\sinh(a)}{\cosh(a)} \]

Formula for computing tanh.

\[ tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \]
Artificial Neural Network

• Like the logistic sigmoid, the tanh function is also sigmoidal (“s”-shaped).
• Unlike logistic sigmoid, it outputs values that range (-1, 1).
• Thus strongly negative inputs to the tanh will map to negative outputs (not zero).

\[
tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}
\]
Artificial Neural Network

- Additionally, **only zero-valued inputs** are mapped to near-zero outputs.
- These properties make the network **less likely to get “stuck” during training**.

The **derivative** of tanh:

\[ g'(a) = \frac{dg}{da} = 1 - g(a)^2 \]
Artificial Neural Network

- **Comparison** of these 3 functions:
Artificial Neural Network

- Both the logistic and hyperbolic tangent are **sigmoidal functions**.
- They are **closely related**: \( \tanh(x) = 2\sigma(2x) - 1 \).

The **problem** with sigmoidal units is that they **saturate across most of their domain**.
Artificial Neural Network

- They **saturate to a high value** when $x$ is very positive.
- And **saturate to a low value (or zero)** when $x$ is very negative.
- They are **only strongly sensitive** to their input when $x$ is near 0.

The **widespread saturation** of sigmoidal units can make gradient-based learning very difficult.
Artificial Neural Network

- The saturated sigmoid output will cause the derivative of sigmoid to be very small, i.e., $g'(\cdot) \rightarrow 0$

This will cause very small loss gradient: $\nabla \mathcal{L}(W^{(k)}) \rightarrow 0$

- $z_j^{(k)} = a^{(k-1)}W^{(k-1)}$
- $a_j^{(k)} = g(z_j^{(k)})$

$\delta^{(k)} = (\delta^{(k+1)} W^{(k)}_{no bias})^T * g'(z^{(k)})$

$\nabla \mathcal{L}(W^{(k-1)}) = a^{(k-1)^T} \delta^{(k)} = [\text{output} \times \text{error}]$
Artificial Neural Network

• For this reason, their use as hidden units in feed-forward networks is now discouraged.
Artificial Neural Network

- To overcome the limitations of the sigmoidal functions, a **linear piecewise function** is used in modern neural networks.
- It is called the **rectified linear unit or ReLU**.
- It is the **default recommendation**.

ReLU is defined by the activation function:

\[ g(z) = \max\{0, z\} \]
Artificial Neural Network

- ReLU is **easy to optimize** because it is so **similar to linear units**.
- The only difference between a linear unit and a rectified linear unit is:
- A rectified linear unit **outputs zero across half its domain**.

This makes the **derivatives** through a rectified linear unit **remain large** whenever the **unit is active** (output is positive).
Artificial Neural Network

• Applying this function to the output of a linear transformation yields a **nonlinear transformation**.

• However, the function **remains very close to linear**, in the sense that it is a piecewise linear function with two linear pieces.
Artificial Neural Network

- Unfortunately, the ReLU activation function is not perfect.
- It suffers from a problem known as the dying ReLUs.
- During training, some neurons effectively “die” meaning they stop outputting anything other than 0.
Artificial Neural Network

- In some cases, we may find that half of the network’s neurons are dead, especially if we use a large learning rate.
- A neuron dies when its weights get tweaked in such a way that the weighted sum of its inputs are negative for all instances in the training set.

\[
\begin{align*}
  z_j^{(k)} &= a^{(k-1)}W^{(k-1)} \\
  a_j^{(k)} &= g(z_j^{(k)})
\end{align*}
\]
Artificial Neural Network

• When this happens, it just keeps outputting zeros, and Gradient Descent does not affect it anymore.

• Because the gradient of the ReLU function is zero when its input is negative.

\[
\begin{align*}
  z_j^{(k)} &= \mathbf{a}^{(k-1)} \mathbf{W}^{(k-1)} \\
  a_j^{(k)} &= g(z_j^{(k)})
\end{align*}
\]
Artificial Neural Network

- Two standard solutions exist to overcome the dying ReLU problem.
  - Leaky ReLU
  - Exponential Linear Unit (ELU)

$$\text{ELU}_\alpha(z) = \begin{cases} 
\alpha (\exp(z) - 1) & \text{if } z < 0 \\
\alpha & \text{if } z \geq 0 
\end{cases}$$

$$\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$$
Leaky ReLU: ReLU with a **small slope** for the negative values.

The hyperparameter $\alpha$ defines how much the function “leaks”: it is the **slope of the function** for $z < 0$ and is typically set to 0.01.

$$\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$$

**ELU**: It takes on **negative values** when $z < 0$, which allows the unit to have an average output closer to 0 and helps alleviate the **vanishing gradient** problem.

The hyperparameter $\alpha$ **defines** the value that the ELU function approaches when $z$ is a **large negative number**.

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$
Issues Regarding Training Deep MLPs

• We will discuss the following issues.
  - Feature scaling
  - Intractability of computation
  - Computational complexity
  - Overfitting due to complex architecture
  - Overfitting due to overtraining
  - Weight initialization
  - Activation Functions
  - Optimizing Stochastic Gradient Descent
  - Prediction invariance
  - Scalability with respect to the input size
Training Issues: Optimizing SGD
Unhappiness

Sorrow

Dissatisfaction

Boredom

Can SGD take us to the elusive happiness?

It is difficult to **find happiness within oneself**, but it is impossible to find it anywhere else.

---

Arthur Schopenhauer
Optimizing SGD

• Advantages of SGD:
  - Faster than the batch GD for each weight update.
  - Makes it possible to train on huge training sets.
  - Due to its inherent randomness, SGD is useful to escape from sub-optimal minima.
Optimizing SGD

• However, SGD has an *Achilles’ heel*.
• It’s sloth!
• SGD takes *much longer time* (due to more iterations) to *converge* near the minima of the loss function, as compared to batch GD.

![Diagram showing convergence comparison between Batch GD and SGD](image-url)
Optimizing SGD

• Why does the SGD takes longer time to converge?
• Because of its oscillatory behavior.
• But why does oscillation occur in SGD?
Optimizing SGD

- In the SGD, at each update of the weight parameters, successive gradient values may be so different that large oscillations may occur.
- The loss gradient directions are erratic!
- This happens even on a smooth loss surface terrain.
Optimizing SGD

• Thus, instead of gently decreasing until it reaches the minimum, the SGD steps **oscillate, decreasing loss only on average.**

• Over time it ends up very close to the minimum, but once it gets there it continues to **bounce around around the minimum**, never settling down.
Optimizing SGD

• Cure for Achilles’ heel

• We can overcome the limitations by using fast optimizers.

• These optimizers are created by augmenting the SGD.
Optimizing SGD

• Following is a list of **fast optimizers** for SGD.
  - Momentum optimization
  - Nesterov Accelerated Gradient
  - AdaGrad
  - Adadelta
  - RMSProp
  - Adam
  - Nadam

  We will discuss the **momentum** optimization
Momentum Optimization

- The Momentum optimization is a **small hack of SGD**.
- It *reduces* SGD’s oscillations and accelerates its movement.

![Diagram](image)
Momentum Optimization

• An intuitive way to **dampen (smooth out) the oscillations** is by computing a **moving/local average** of the loss gradient.

The technique to compute this is known as **Exponentially Weighted Moving Average (EWMA)**.
Momentum Optimization

• To impose “smoothing”, the EWMA puts more weight on the past observations.
• Past observations build up “momentum” that stabilizes the fluctuations of the current observation.

\[
\text{Moving\_Average}(t) = \text{weight} \times \text{Past\_Observations} + (1 - \text{weight}) \times \text{Observation}(t)
\]

Past observations are weighted heavily by the weight parameter (denoted by \(\beta\)).

It determines how far we go into the past to determine the present.
Momentum Optimization

• How does the momentum optimization work?
• SGD update rule *does not care* about what the earlier gradients were.
• If the **local gradient** is tiny, SGD goes very slowly.
• The momentum optimization not only uses the **gradient of the current step** \( t \) to guide the search, but also it accumulates the **gradient of the past steps (0 to t-1)** to determine the direction to go.

\[
W_t^{(k)} = W_t^{(k)} - \eta \nabla \mathcal{L} \left( W_t^{(k)} \right)
\]

\[
\text{Moving-Average}(t) = \text{weight} \times \text{Past-Observations} + (1 - \text{weight}) \times \text{Observation}(t)
\]
Momentum Optimization

• In momentum optimization, for a layer k’s weight matrix $W^{(k)}$, we keep an additional matrix of the same dimension $m$ (it is called the momentum).

• Here $m$ is used to keep a moving average of the past gradients.

• We add the weighted past gradients ($t - 1$) with the current gradient ($t$).

$$m_t^{(k)} = \beta m_{t-1}^{(k)} + (1 - \beta) \nabla \mathcal{L} \left( W_t^{(k)} \right)$$

$$W_{t+1}^{(k)} = W_t^{(k)} - \eta m_t^{(k)}$$
Momentum Optimization

- At the beginning, $m$ is initialized with zeros.
- Then, it keeps a moving average of the past gradients.
- We update $W^{(k)}$ using the moving average of the past gradients.
- Here the momentum $m_t^{(k)}$ is the retained gradient from past iterations.

$$
W_{t+1}^{(k)} = W_t^{(k)} - \eta \nabla \mathcal{L}(W_t^{(k)})
$$

Vanilla SGD

$$
W_{t+1}^{(k)} = W_t^{(k)} - \eta m_t^{(k)}
$$

Momentum Optimization

Moving-Average(t) = weight * Past-Observations + (1 - weight) * Observation(t)
Momentum Optimization

• We don’t want the momentum to grow unboundedly as it accumulates past gradients.
• This is prevented by introducing “friction” via a new hyperparameter $\beta$.
• It must be set between 0 (high friction) and 1 (no friction).
• A typical value of $\beta$ is 0.9.

\[ m_t^{(k)} = \beta m_{t-1}^{(k)} + (1 - \beta) \nabla \mathcal{L}(W_t^{(k)}) \]

\[ W_{t+1}^{(k)} = W_t^{(k)} - \eta m_t^{(k)} \]
Momentum Optimization

- When $\beta = 0$, we recover the vanilla SGD.
- But for $\beta = 0.9$ (or larger, much closer to 1), past gradients are weighted more heavily.

$$
\begin{align*}
    m_t^{(k)} &= \beta m_{t-1}^{(k)} + (1 - \beta) \nabla \mathcal{L}(W_t^{(k)}) \\
    W_{t+1}^{(k)} &= W_t^{(k)} - \eta m_t^{(k)}
\end{align*}
$$

This technique **creates an annealing effect** (reduction of learning rate) by multiplying $\eta$ with the **changing momentum** (gradient) term.
Momentum Optimization

• In practice, a less intuitive version of the momentum update rule works better.

• Here $\eta$ is re-estimated for dropping the $(1 - \beta)$ term before the loss gradient.

\[
m^{(k)}_t = \beta m^{(k)}_{t-1} + (1 - \beta) \nabla L \left( W^{(k)}_t \right)
\]

\[
W^{(k)}_{t+1} = W^{(k)}_t - \eta m^{(k)}_t
\]

More effective

\[
m^{(k)}_t = \beta m^{(k)}_{t-1} - \eta \nabla L \left( W^{(k)}_t \right)
\]

\[
W^{(k)}_{t+1} = W^{(k)}_t + m^{(k)}_t
\]
Momentum Optimization

• To understand why the momentum optimization is effective, first understand why oscillation occurs in SGD.
• Consider the following bowl-shaped 2D loss function.
• The slope varies gently along the direction of $w_1$, while it varies heavily along the direction of $w_2$.
• SGD keeps on bouncing back and forth with high amplitude as it moves from left to right.

![Diagram of a bowl-shaped 2D loss function with SGD path]
Momentum Optimization

• The momentum optimization *averages out the gradients* for both $w_1$ and $w_2$.

• The oscillations in the *steeper direction* will tend to average out to something closer to zero.

• Thus, in the steeper direction, where we want to slow things down, this will average out positive and negative numbers.

The average in the flatter direction will be pretty big.