The Art & Science of Training
Deep MLPs

M. R. Hasan
Deep Learning
Readings

- Bishop: 5.1, 5.3, 5.5
- Murphy: 16.5, 16.5.4
- Alpaydin: 11
- Geron: 11
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
  - Vanishing gradient & exploding gradient problem
  - Achilles’ Heel of Stochastic Gradient Descent
  - Prediction invariance
  - Scalability with respect to the input size
Training Issues: Feature Scaling
Training Issues: Feature Scaling

- For training MLPs, we need to **scale the features**.
- There are **two main reasons** for feature scaling.
  - Effective learning
  - Faster convergence
Training Issues: Feature Scaling

- Effective Learning:
- Consider the following **loss gradient** equation used in the SGD update rule.

\[
\nabla \mathcal{L}(W^{(k)}) = a^{(k)T} \delta^{(k+1)}
\]

Notice that **feature vector** \( a^k \) (extracted from the input \( x \)) is used in the weight updates (multiplied with the error term).
Training Issues:
Feature Scaling

- Effective Learning:
- Thus, if features (components of $\mathbf{a}^k$) have varying scales, then certain weights may update faster than others.

As a consequence, some features will have larger weights leading to an unstable learning.

$$\nabla \mathcal{L}(\mathbf{W}^{(k)}) = \mathbf{a}^{(k)T} \delta^{(k+1)}$$

$$\mathbf{W}^{(k)} = \mathbf{W}^{(k)} - \eta \nabla \mathcal{L}(\mathbf{W}^{(k)})$$
Training Issues: Feature Scaling

- Effective Learning:
- By **feature scaling** (normalizing the components of $a^k$) we ensure that one feature doesn’t impact the model just because of its **large magnitude**.

\[ \nabla \mathcal{L}(W(k)) = a^{(k)T} \delta^{(k+1)} \]

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

- Fast Convergence:
- If some features have a very large scale then SGD takes more numbers of iterations to converge.
- Left: features have the same scale
- Right: feature 1 has much smaller values than feature 2
The most common **techniques of feature scaling**:

- Standardization: transforms the data to have 0 mean and a variance of 1.
- Normalization: transforms the data to bound the values between two numbers, typically, between [0, 1].
Training Issues: Feature Scaling

- **Standardization** (z-score normalization):
- First we make an assumption that the distribution of the features are **Gaussian**.

We determine the distribution **mean** and **standard deviation** for each feature.

Next we **subtract the mean** from each feature.

Then we **divide the values** (mean is already subtracted) of each feature by its standard deviation.

This process standardizes features by **removing the mean** and scaling to **unit variance**.

\[
x' = \frac{x - \bar{x}}{\sigma}
\]
Training Issues: Feature Scaling

- **Normalization** (min-max scaling):
  - The data is scaled to a fixed range, usually 0 to 1.
  - The cost of having this bounded range is that we will end up with *smaller standard deviations*, which can suppress the effect of outliers.
  - A Min-Max scaling is typically done via the following equation:

\[
X_{norm} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}
\]

Example: grayscale image pixel value range 0 ~ 255

\[
X_{norm} = \frac{X - 0}{255 - 0} = \frac{X}{255}
\]
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
  - Vanishing gradient & exploding gradient problem
  - Achilles’ Heel of Stochastic Gradient Descent
  - Prediction invariance
  - Scalability with respect to the input size
Training Issues: Intractable Computation
Training Issues: Intractable Computation

- The **training time** in Deep MLPs could be **intractable**!
Consider the error equations in backpropagation.

Observe that the errors at the hidden layers are defined recursively.

Hidden layer error

\[ \delta^{(k)} = \left( \delta^{(k+1)} W^{(k)T}_{\text{no bias}} \right) \ast g'(z^{(k)}) \]

Output layer error

Regression: squared error loss

\[ \delta^{(K)} := (a^{(K)} - y) \ast g'(z^{(K)}) \]

Binary classification: cross-entropy loss

\[ \delta^{(K)} := \frac{\partial L}{\partial a^{(K)}} \ast g'(z^{(K)}) \]

Multi-class classification: cross-entropy loss

\[ \delta^{(K)} := a^{(K)} - y \]
Training Issues: Intractable Computation

• Observe that the errors at the hidden layers are defined recursively.

\[ \delta^{(k)} = \left( \delta^{(k+1)} W^{(k)T}_{\text{no bias}} \right) \ast g'(z^{(k)}) \]
Training Issues: Intractable Computation

- Consider the calculation of error at the 2\textsuperscript{nd} layer.

\[ \delta_1^{(2)} = g' (z_1^{(2)}) \left[ W_{11}^{(2)} \delta_1^{(3)} + W_{12}^{(2)} \delta_2^{(3)} \right] \]

\[ \delta_2^{(2)} = g' (z_2^{(2)}) \left[ W_{21}^{(2)} \delta_1^{(3)} + W_{22}^{(2)} \delta_2^{(3)} \right] \]

\[ \delta_j^{(k)} = g' \left( z_j^{(k)} \right) \sum_{p} W_{jp}^{(k)} \delta_p^{(k+1)} \]

\text{Hidden layer error}

We are recomputing the error from each neuron of the previous layer.

It will result into an exponential rise of complexity!
Training Issues: Intractable Computation

- How do we mitigate this issue?
- Observe the nature of the problem.
- We have recursion and overlapping subproblems.
- Use dynamic programming!!!

\[
\delta^{(k)} = (\delta^{(k+1)} W_{no\ bias}^{(k)^T}) \ast g(\mathbf{z}^{(k)})
\]

Hidden layer error

Don’t recompute!

Cache the error terms and reuse.
Training Issues: Intractable Computation

• **Reflection**: Training an MLP was a **big challenge** because of the non-convexity and the sheer size of the MLP.

• How did we handle this issue?

• **Gradient descent** with some **added tricks**!

---

**Diagram:**

- Layer: $k = 1$
- Layer: $k = 2$
- Layer: $k = 3$
- Layer: $k = 4$

Chain rule for partial derivative

Dynamic programming
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
  - Vanishing gradient & exploding gradient problem
  - Achilles’ Heel of Stochastic Gradient Descent
  - Prediction invariance
  - Scalability with respect to the input size
Training Issues:
Computational Complexity
Let’s look at the **space and time complexity** of MLPs.

An MLP with $d$ inputs, $M$ hidden units (neurons), and $K$ outputs has the following no. of weights:

1\textsuperscript{st} Layer: $M(d + 1)$ weights

2\textsuperscript{nd} Layer: $K(M + 1)$ weights
MLP: Complexity of Learning

- Hence, both the \textit{space and time complexity} of an MLP are:

\[ O(M \cdot (d + K)) \].

\begin{itemize}
  \item \textbf{1}st Layer: \( M(d + 1) \) weights
  \item \textbf{2}nd Layer: \( K(M + 1) \) weights
\end{itemize}
MLP: Complexity of Learning

• When \( e \) denotes the number of training epochs, the training time complexity is:
  \[ O(e \cdot M \cdot (d + K)). \]

1\(^{st}\) Layer:
\( M(d + 1) \) weights

2\(^{nd}\) Layer:
\( K(M + 1) \) weights
MLP: Complexity of Learning

- In a problem, \( d \) and \( K \) are predefined.
- The parameter that we play with to tune the complexity of the model is \( M \).

What happens if we create a complex MLP architecture with many neurons?
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
  - Vanishing gradient & exploding gradient problem
  - Achilles’ Heel of Stochastic Gradient Descent
  - Prediction invariance
  - Scalability with respect to the input size
Training Issues:
Overfitting due to Complex Architecture
MLP: Overfitting

- Due to a large number of computational units or neurons, MLPs are expressively expressive.
- However, this extreme flexibility causes severe overfitting.
- Let’s illustrate this problem.
MLP: Overfitting

Example: As complexity (number of hidden layer neurons) increases, training error remains fixed but the validation error starts to increase and the network starts to overfit.
MLP: Overfitting

• To combat overfitting caused by the network architecture, we need to **regularize** the model.

• The two popular regularization approaches are:
  - Weight Decay
  - Dropout
Combat Architectural Overfitting: Weight Decay Regularization
MLP: Overfitting

• Weight Decay:
• Regularize the MLP by **updating** its loss/cost function.
• It **penalizes** the model in proportion to the size of the model weights.

1\textsubscript{2} Regularization:

\[ L_{\text{regularized}}(\vec{w}_i) = L(\vec{w}_i) + \frac{\lambda}{2} \vec{w}_i^2 \]
MLP: Overfitting

• Weight Decay: The main idea is that some connections may not be necessary.
• We turn them off by setting their weights to 0.
• We give each connection a tendency to decay to 0 so that it disappears unless it is reinforced explicitly to decrease error.
MLP: Overfitting

• Weight Decay:
• For implementing weight decay we usually use the $l_p$ regularizer, where $p$ is any positive integer starting from 1.
  - $p = 1$: $l_1$ regularizer
  - $p = 2$: $l_2$ regularizer

$l_2$ Regularization:

$$L_{regularized}(\vec{w}_i) = L(\vec{w}_i) + \frac{\lambda}{2} \vec{w}_i^2$$
Combat Architectural
Overfitting: Dropout
Regularization
MLP: Overfitting

- Dropout:
  - At each iteration, we drop every neuron (set its weight to zero) temporarily with a probability $p$.
  - Here $p$ is a hyperparameter, which is known as the dropout rate.
  - Its value is usually set between 10% to 50%.
MLP: Overfitting

• Dropout:

• It includes input layer “neurons” but excludes the output neurons.

• Dropout regularization technique was proposed by Hinton in 2012: [http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf](http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf)
MLP: Overfitting

• Dropout is an effective regularization technique.
• Let’s see why.
• During training, neurons tend to co-adapt with their neighboring neurons thereby overfit the training data.
• Due to random dropout, the neurons become independent as they cannot co-adapt.
MLP: Overfitting

- Thus, each individual neuron learns to perform effectively on its own.
- As a result, they become more **immune to the changes** in small variations in the input.
- When a Dropout trained MLP is fed with unseen test data, it **generalizes better**.
MLP: Overfitting

- Another useful aspect of Dropout is that at each iteration it creates a unique MLP.
- Because every neuron could be either on or off.
- Note that for $N$ neurons there are $2^N$ possible networks.
- Thus, it is unlikely that two networks at two iterations would be the same.
MLP: Overfitting

• As a result, at the end of training the network for $t$ iterations, there would be an ensemble of $t$ unique networks.
• The resulting network can be seen as an averaging ensemble of all these networks.
MLP: Overfitting

• The inherent diversity of the ensemble improves the generalization performance.

• See the following notebook for an empirical understanding of dropout regularization:

Issues Regarding Training Deep MLPs

- Feature scaling
- Intractability of computation
- Computational complexity
- Overfitting due to complex architecture
- Overfitting due to overtraining
- Weight initialization
- Vanishing gradient & exploding gradient problem
- Achilles’ Heel of Stochastic Gradient Descent
- Prediction invariance
- Scalability with respect to the input size
Training Issues: Overfitting due to Overtraining
MLP: Overtraining

- Another consequence of increased complexity (larger number of hidden units) is that as *more training epochs are made*, the error on the training set decreases.

However, the error on the validation set starts to increase beyond a certain point.

This problem is the direct result of overtraining!
MLP: Overtraining

• Why does longer training time reduce training error?
• Note that initially all the weights are close to 0 and thus have little effect.
• As training continues, the most important weights start moving away from 0 and are utilized.
MLP: Overtraining

- But if training is continued further on to get less and less error on the training set, almost all weights are updated away from 0.
- They effectively become parameters.

Thus as training continues, it is as if new parameters are added to the system, increasing the complexity and leading to poor generalization.
Combat Overtraining-caused Overfitting: Early Stopping Regularization
MLP: Overtraining

• To solve the overfitting problem of overtraining, learning should be stopped early.

• The optimal point to stop training, and the optimal number of hidden units, is determined through cross-validation.
MLP: Overtraining

- Since cross-validation is infeasible in deep MLPs, we set aside a **fixed validation set** for early stopping.
- The validation set is also used to create **comparative learning curves**.
Combat Overfitting: 3 Regularization Techniques
Summary: 3 Regularization Techniques to Combat Overfitting

- We have discussed three regularization techniques.
  - Weight Decay ($l_p$ regularization)
  - Dropout
  - Early Stopping
- In Deep Neural Networks, the dropout regularization has been found to be the most effective.
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
  - Vanishing gradient & exploding gradient problem
  - Achilles’ Heel of 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.

\[
\begin{align*}
\nabla L(W^{(k-1)}) &= a^{(k-1)^T} \delta^{(k)} \\
\delta^{(k)} &= (\delta^{(k+1)} W_{no bias}^{(k)^T}) \ast g'(z^{(k)})
\end{align*}
\]
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}.

\[ z_j^{(k)} = a_j^{(k-1)}w_{j(k-1)} \]
\[ a_j^{(k)} = g(z_j^{(k)}) \]

What happens if we choose \textbf{large random} values for initialization?
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., $g'(.) \rightarrow 0$

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

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

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

This problem is known as the notorious \textbf{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:

  • https://github.com/rhasanbd/Artificial-Neural-Network-Back-to-Basics/blob/master/ANN-I-Initialization%20of%20the%20Weights.ipynb