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
The Art & Science of Training Deep MLPs

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

• Some issues concern the deep MLPs (many hidden layers and neurons) or more generally deep neural networks (DNNs).
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

We will discuss the following issues.
Training Issues:
Vanishing Gradient & Exploding Gradient Problems
Vanishing & Exploding Gradient

• After replacing tanh with ReLU, we experienced another problem.
• Unlike tanh, the ReLU activation mean is not 0.
• Recall we used 0 mean activation to derive Glorot initializer.
• It was found that the Glorot initializer was not compatible with ReLU.
Vanishing & Exploding Gradient

• In **2015 Kaiming He** proposed a new initializer, known as *He initializer*.

• We will discuss the limitations of Glorot initializer and present the He initializer.

In the analysis, we need to incorporate the **non-zero** activation mean: \([E(a)] \neq 0\)
Vanishing & Exploding

- **Forward Propagation.**
- The variance of a neuron $z_j$ at the output layer $k + 1$.

\[
\text{var}(z_1^{(k+1)}) = \text{var} \left( \sum_{i=0}^{m^{(k)}} W_{ji}^{(k)} a_i^{(k)} \right)
\]

\[
= \sum_{i=0}^{m^{(k)}} \text{var}(W_{ji}^{(k)} a_i^{(k)})
\]

For two independent variables $x$ and $y$:
\[
\text{var}(xy) = \left[ \text{E}(x) \right]^2 \text{var}(y) + \left[ \text{E}(y) \right]^2 \text{var}(x) + \text{var}(x)\text{var}(y)
\]

\[
\text{var}(z_1^{(k+1)}) = \sum_{i=0}^{m^{(k)}} \left[ \text{E}(W_{ji}^{(k)}) \right]^2 \text{var}(a_i^{(k)}) + \left[ \text{E}(a_i^{(k)}) \right]^2 \text{var}(W_{ji}^{(k)}) + \text{var}(a_i^{(k)})\text{var}(W_{ji}^{(k)})
\]

8
Vanishing & Exploding

- Forward Propagation.
- Assuming that **weights have 0 mean** (sampled from a 0 mean normal distribution):

$$\text{var}(z_1^{(k+1)}) = \sum_{i=0}^{m^{(k)}} \left[ \text{E}(W_{ji}^{(k)}) \right]^2 \text{var}(a_i^{(k)}) + \left[ \text{E}(a_i^{(k)}) \right]^2 \text{var}(W_{ji}^{(k)}) + \text{var}(a_i^{(k)}) \text{var}(W_{ji}^{(k)})$$

Unlike sigmoid or tanh cases, the **mean response** $[\text{E}(a)]^2$ is not zero in ReLU
The sum is represented as $m^{(k)}$ products; the expectation of all input signals is denoted by a vector $a^{(k)}$; and dropping the index of the output neuron $z^{(k+1)}$: 

$$\text{var}(z^{(k+1)}) = m^{(k)} \text{var}(W^{(k)}) E[(a^{(k)})^2]$$
Let’s **simplify** the $E[(a^{(k)})^2]$ term:

$$E[(a^{(k)})^2] = \int_{-\infty}^{\infty} (a^{(k)})^2 P(a^{(k)}) \, da$$

We use the ReLU function to calculate the response: $a = \max(0, z)$

$$E[(a^{(k)})^2] = \int_{-\infty}^{\infty} \max(0, z^{(k)})^2 P(z^{(k)}) \, dz$$

Using the limit of the integration for the **positive input** (from 0 to $\infty$).

$$E[(a^{(k)})^2] = \int_{0}^{\infty} (z^{(k)})^2 P(z^{(k)}) \, dz$$
To limit of the integration for the positive input (from $-\infty$ to $\infty$), we divide the expression by 2.

$$E[(a^{(k)})^2] = \int_0^\infty (z^{(k)})^2 P(z^{(k)}) dz$$

Using these two facts:

$$\text{var}(z) = \int_{-\infty}^\infty z^2 P(z) - E[z]^2$$

$$\Rightarrow E[(a^{(k)})^2] = \frac{1}{2} \text{var}(z^{(k)})$$

Mean of affine combination: $E[z] = 0$

Plug this into the variance equation.

$$\text{var}(z^{(k+1)}) = m^{(k)} \text{var}(W^{(k)}) E[(a^{(k)})^2]$$

$$\text{var}(z^{(k+1)}) = \frac{m^{(k)}}{2} \text{var}(W^{(k)}) \text{var}(z^{(k)})$$
The variance equation for **Glorot**:

\[
\text{var}(z^{(k+1)}) = m^{(k)} \text{var}(W^{(k)}) \text{var}(z^{(k)})
\]

The variance equation for **He**:

\[
\text{var}(z^{(k+1)}) = \frac{m^{(k)}}{2} \text{var}(W^{(k)}) \text{var}(z^{(k)})
\]

**Forward computation:** comparison between Glorot & He

In **He**, variance should be 50% smaller
The variance of the final layer response (computed recursively):

$$\text{var}(z^{(k+1)}) = \frac{m^{(k)}}{2} \text{var}(W^{(k)}) \text{var}(z^{(k)})$$

We want variance of input layer (k) signal to be equal to the variance at the output layer (k + 1) signal:

$$\text{var}(z^{(K)}) = \text{var}(x) \prod_{k=1}^{K-1} \frac{m^{(k)}}{2} \text{var}(W^{(k)})$$

Denote the number of input layer neurons (or **fan-in**) with \(m^{(in)}\)

$$\frac{m^{(k)}}{2} \text{var}(W^{(k)}) = 1$$

$$\Rightarrow \text{var}(W^{(k)}) = \frac{2}{m^{(k)}}$$

Glorot

$$\text{var}(W^{(k)}) = \frac{1}{m^{(in)}}$$

$$\text{var}(W^{(k)}) = \frac{2}{m^{(in)}}$$

$$\text{var}(W^{(k)}) = \frac{1}{m^{(in)} / 2}$$
Vanishing & Exploding Gradient

• **Backward Propagation.**

• Due to ReLU activation function, about **50% of the neurons** at each layer would be turned off, thus won’t propagate error backwards.

• Let’s compute the error for $z_1^{(k)}$:

\[
\delta_1^{(k)} = g'(z_1^{(k)}) \sum_j W_{j1}^{(k)} \delta_j^{(k+1)}
\]
Vanishing & Exploding Gradient

- **Backward Propagation:**

  - The sum is replaced by \( m^{(k+1)}/2 \) products, since only 50% neurons are active due to ReLU.

  - All errors signals of \((k+1)\) layer is denoted by a vector \( \delta^{(k+1)} \).

  - Finally dropping the neuron index at layer \( k \).

\[
\delta^{(k)} = g'(z^{(k)}) \frac{m^{(k+1)}}{2} W^{(k)} \delta^{(k+1)}
\]

\[
\delta_1^{(k)} = g'(z_1^{(k)}) \sum_i W_{j1}^{(k)} \delta_j^{(k+1)}
\]
Vanishing & Exploding Gradient

- **Backward Propagation.**
- The gradient of the ReLU activation of the active neurons in layer $k$ will be 1: $g'(z^{(k)}) = 1$

\[
\delta^{(k)} = g'(z^{(k)}) \frac{m^{(k+1)}}{2} W^{(k)} \delta^{(k+1)}
\]

\[
\delta^{(k)} = \frac{m^{(k+1)}}{2} W^{(k)} \delta^{(k+1)}
\]
Vanishing & Exploding Gradient

- **Backward Propagation.**
- **Variance** of the weighted error:

\[
\text{var}(\delta^{(k)}) = \frac{m^{(k+1)}}{2} \text{var}(W^{(k)}) \text{var}(\delta^{(k+1)})
\]

\[
= \text{var}(\delta^{(k+1)}) \left[ \frac{m^{(k+1)}}{2} \text{var}(W^{(k)}) \right]
\]

The variance of error of two layers should be equal, thus:

\[
\text{var}(W^{(k)}) = \frac{2}{m^{(k+1)}}
\]

\[
\delta^{(k)} = \frac{m^{(k+1)}}{2} W^{(k)} \delta^{(k+1)}
\]

Denote the number of output layer neurons (or **fan-out**) with \(m^{(\text{out})}\)

\[
\text{var}(W^{(k)}) = \frac{2}{m^{(\text{out})}} \quad \text{var}(W^{(k)}) = \frac{1}{m^{(\text{out})}}
\]
Variance of the weights in layer $k$ is determined by taking the average:

\[
var(W^{(k)}) = \frac{1}{m^{(in)} + m^{(out)}}
\]
Vanishing & Exploding Gradient

- He Initializer.
- We can use normal distribution.

$$\text{var}(W^{(k)}) = \frac{4}{m^{(in)} + m^{(out)}}$$

$$w = \text{numpy.random.randn}(m_{in}, m_{out}) \ast \frac{2}{\text{numpy.sqrt}(m^{(in)} + m^{(out)})}$$

Or,

$$w = \text{numpy.random.normal}(0, \frac{2}{\text{numpy.sqrt}(m^{(in)} + m^{(out)}), \text{size} = (m_{in}, m_{out})}$$
Vanishing & Exploding Gradient

- He Initializer.
- We can also use uniform distribution.

\[ \text{var}(W^{(k)}) = \frac{4}{m^{(in)} + m^{(out)}} \]

\[ w = \text{numpy.random.uniform}(low = -r, high = r, size = (m^{(in)}, m^{(out)})) \]

Where the limit is: \( r = \sqrt{\frac{12}{m^{(in)} m^{(out)}}} \)
Vanishing & Exploding Gradient

• **Summary** of various activations and initialization techniques to avoid vanishing and exploding gradient problem.
  - (1998): Logistic sigmoid/Tanh + LeCun weight initializer
  - (2010): Tanh + Glorot weight initializer
  - (2015): ReLU + He weight initializer
Dying ReLU Problem

• It’s a standard practice to use Glorot/He initializer with ReLU.

• Although we have resolved the vanishing and exploding gradient problem by using suitable ReLU activation and Glorot/He initializer, it appears that ReLU has a side-effect!
Dying ReLU Problem

• Sometimes the weighted input signal to a neuron could be negative.
  - May be one of the bias weights was too large, or
  - may be the learning rate was large.
Dying ReLU Problem

- If the weighted input to a neuron is negative, its ReLU activation function will **simply kill the neuron**.
- In some cases, **half of the network’s neurons** could be dead (large learning rate).
- It is known as the **dying ReLU** problem.
Dying ReLU Problem

• When this happens, it just keeps **outputting zeros**.
• Thus, Gradient Descent does not affect it anymore because the **gradient of the ReLU function is zero** when its input is negative.

\[ W^{(k)} \leftarrow W^{(k)} - \eta \nabla W^{(k)} J \]

\[ \frac{\partial J}{\partial W^{(k)}_{ji}} = [g'(z^{(k+1)}_j) \sum_{m \in \text{layer}(k+2)} W^{(k+1)}_{mj} \delta^{(k+2)}_m] a_i^{(k)} \]
Dying ReLU Problem

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

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

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

- **Leaky ReLU**
- We change the ReLU function a little bit such that it **gives a non-zero (negative) response** for negative input signals.
- In other words, we allow **some response to “leak”** due to negative input.

\[
\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)
\]
Dying ReLU Problem

- Leaky ReLU
- The hyperparameter $\alpha$ determines the amount of "leak".
- Typically it’s set to a fixed value of 0.01.
- Due to this small slope even if a neuron receives a negative weighted input, it doesn’t die.

LeakyReLU$_\alpha(z) = \max(\alpha z, z)$
Dying ReLU Problem

- Leaky ReLU
- There is **one issue** of LReLU that we need to fix.
- Choosing an optimal value for $\alpha$ in LReLU is non-trivial.
- Hyperparameter tuning is **prohibitively expensive** in DNNs.

LeakyReLU$_{\alpha}(z) = \max(\alpha z, z)$
Dying ReLU Problem

- Leaky ReLU
- May be we shouldn’t treat $\alpha$ as a hyperparameter.
- Let’s treat it as another parameter of the model, just like the weight parameter, and **learn its optimal value**.
- **Parametric ReLU** (PReLU) was proposed to do exactly this.

\[
\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)
\]
Dying ReLU Problem

- Parametric ReLU
- In PReLU instead of being a hyperparameter, \( \alpha \) becomes a parameter that can be modified by Backpropagation.

LeakyReLU\(_\alpha(z) = \max(\alpha z, z)\)
Dying ReLU Problem

- Exponential Linear Unit (ELU)
- ELU **tweaks the slope** of the LReLU.
- Instead of using a straight line as the slope, it uses a log curve line.
- For negative input \(z < 0\), the activation response is given as follows: \(a_j = \alpha(\exp(z_j) - 1)\).
Dying ReLU Problem

- Exponential Linear Unit (ELU)
- Unlike ReLU, ELU has a 0-mean activation.
- Thus it is more suitable with the Glorot initializer.

\[
\text{ELU}_\alpha (z) = \begin{cases} 
\alpha (\exp (z) - 1) & \text{if } z < 0 \\
 z & \text{if } z \geq 0
\end{cases}
\]
Dying ReLU Problem

• Exponential Linear Unit (ELU)
• The $\alpha$ hyperparameter is usually set to 1.
• One problem with ELU is that it saturates at large negative input values.

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

- Exponential Linear Unit (ELU)
- In addition to this, it’s **slower to compute** due to the exponential term.
- But it doesn’t matter because its **faster convergence** weighs much higher than the cost added by the exponential term.

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

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**.

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

• So far, we have discussed various activations and weight initialization techniques to avoid the vanishing/exploding gradient problem.
• We haven’t prescribed how to initialize the bias terms.
• 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.
Vanishing & Exploding Gradient

• However, for **ReLU activation** function may be it’s a good idea to initialize biases with a **very small number**, e.g., 0.01.
• This could be useful because with zero biases initially around 50% of the neurons won’t fire.
• Having small non-zero biases will enable to **activate all neurons in the beginning**.
Vanishing & Exploding Gradient

- To prevent vanishing/exploding gradient, other approaches were proposed.
- These approaches involve implementing architectural/engineering tricks.
  - Gradient Clipping
  - Batch Normalization
  - Skip Connections
  - Gated Recurrent Unit
- The last two tricks require a good understanding of the architectures of CNN and RNN, thus will be discussed later.
Vanishing & Exploding Gradient

• **Gradient Clipping**
  • It is an effective technique to mitigate the gradient explosion.
  • Clip each component of the gradient vector or its $l_2$ norm.
  • Clip the norm $\|g\|$ of the gradient $g$ as follows:

$$\text{if } \|g\| > v$$

$$\|g\| \leftarrow \frac{gv}{\|g\|}$$

Here $v$ is the norm threshold.

See detail discussion on gradient clipping in “RNN-5-Preventing Unstable Gradients”
Vanishing & Exploding Gradient

• **Batch Normalization**

• BN is an effective technique to mitigate the *vanishing gradient* problem.

• To understand BN, let’s recall what we have done so far to mitigate the vanishing gradient problem.
Vanishing & Exploding Gradient

- So far, we focused on **two factors** for solving the vanishing gradient problem.

  - **Weight initialization**: set the weight variance properly to ensure that activation/error variance remains constant.

  - **Activation**: use an activation without saturation regions (ReLU).
Vanishing & Exploding Gradient

• **Batch Normalization**
  • BN focuses on a different part of the space: *weighted input*
  • It ensures that the weighted signals have **small enough variance** not to cause saturation.

\[
\frac{\partial J}{\partial W_{ji}^{(k)}} = [g'(z_j^{(k+1)})] \sum_{m} W_{mj}^{(k+1)} \delta_m^{(k+2)} a_i^{(k)}
\]

---

**Forward Propagation**

\[
z_j^{(k)} = \sum_{i} W_{ji}^{(k-1)} a_i^{(k-1)}
\]

**Backward Propagation**

\[
a_j^{(k)} = g(z_j^{(k)})
\]

No need to worry about the activation!

Reduce the variance by normalizing “z”
Vanishing & Exploding Gradient

- **Batch Normalization**
- In DNNs, output of one layer is fed as the input of the next layer.
- When the parameters $W^{(k)}$ of a layer change, the distribution of inputs to subsequent layers also changes.

\[
\frac{\partial J}{\partial W_{ji}^{(k)}} = \left[g'(z_{j}^{(k+1)}) \sum_{m=layer(k+2)} W_{mj}^{(k+1)} \delta_{m}^{(k+2)} \right] a_{i}^{(k)}
\]

\[
z_{j}^{(k)} = \sum_{i=layer(k-1)} W_{ji}^{(k-1)} a_{i}^{(k-1)}
\]

Forward Propagation

Backward Propagation

No need to worry about the activation!

Reduce the variance by normalizing “z”
Vanishing & Exploding Gradient

• If signals get increased, it drives towards saturation, which results into the vanishing gradient problem.

• BN **normalizes (unit variance) the distribution of input in each layer**, thereby avoids creating vanishing gradient.

\[
\frac{\partial J}{\partial W_{ji}^{(k)}} = [g'(z_j^{(k+1)})] \sum_m W_{mj}^{(k+1)} \delta_m^{(k+2)} a_i^{(k)}
\]

**Forward Propagation**

\[
z_j^{(k)} = \sum_i W_{ji}^{(k-1)} a_i^{(k-1)}
\]

**Backward Propagation**

\[
a_j^{(k)} = g(z_j^{(k)})
\]

No need to worry about the activation!

Reduce the variance by normalizing “z”
Vanishing & Exploding Gradient

• Batch Normalization
• During each iteration, BN performs two operations for a given mini-batch (mb):
  - First, it zero-centers and normalizes each input.
  - Then, it scales and shifts the input.
• Thus, iteratively the model learns the optimal scale and mean of each of the layer’s inputs.

\[
\mu_{mb} = \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} x^{(i)}
\]
\[
\sigma^2_{mb} = \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} (x^{(i)} - \mu_{mb})^2
\]
\[
\hat{x}^{(i)} = \frac{x^{(i)} - \mu_{mb}}{\sqrt{\sigma^2_{mb} + \epsilon}}
\]
\[
z^{(i)} = \gamma \otimes \hat{x}^{(i)} + \beta
\]
Vanishing & Exploding Gradient

- Due to this normalization of activation outputs by some randomly initialized parameters, the **weights in the next layer are no longer optimal**.
- It may **change what the layer can represent**.
- Thus we need to **“denormalize”** via learning.

\[
\frac{\partial J}{\partial W_{ji}^{(k)}} = [g'(z_j^{(k+1)})] \sum_{m, \text{layer}(k+2)} W_{m,j}^{(k+1)} \delta_m^{(k+2)} a_i^{(k)} \\
W_j^{(k)} \leftarrow W_j^{(k)} - \eta \nabla W_j^{(k)} J
\]

\[
z_j^{(k)} = \sum_{i, \text{layer}(k-1)} W_{ji}^{(k-1)} a_i^{(k-1)}
\]
Vanishing & Exploding Gradient

• Batch Normalization
• For “denormalization”, the normalized input is multiplied by a “standard deviation” parameter $\gamma$, which scales it.
• Then, a “mean” parameter $\beta$ is added to shift the scaled input, which acts like a bias term.
• Thus when BN is added, we don’t need to use a separate bias weight term.

\[
\begin{align*}
\mu_{mb} &= \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} x^{(i)} \\
\sigma^2_{mb} &= \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} (x^{(i)} - \mu_{mb})^2 \\
\hat{x}^{(i)} &= \frac{x^{(i)} - \mu_{mb}}{\sqrt{\sigma^2_{mb} + \epsilon}} \\

z_d &= \gamma_d x_d + \beta_d \\
z^{(i)} &= \gamma \otimes \hat{x}^{(i)} + \beta
\end{align*}
\]

Normalized input is scaled and shifted.
Vanishing & Exploding Gradient

- Batch Normalization
- For each layer, these two parameters are learned by SGD.
- Thus, the SGD algorithm will denormalize by changing only these two parameters for each activation.

\[ z_d = \gamma_d x_d + \beta_d \]

\[ z^{(i)} = \gamma \otimes \hat{x}^{(i)} + \beta \]

Normalized input is scaled and shifted.
Vanishing & Exploding Gradient

• Batch Normalization
• For testing a BN trained model, we need to **normalize the test data**.
• However, since typically we use either a single test data or a small batch, we will have no way to compute each input’s mean and standard deviation reliably.

\[
\mu_{mb} = \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} x^{(i)}
\]

\[
\sigma^2_{mb} = \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} (x^{(i)} - \mu_{mb})^2
\]

\[
\hat{x}^{(i)} = \frac{x^{(i)} - \mu_{mb}}{\sqrt{\sigma^2_{mb} + \epsilon}}
\]

\[
z^{(i)} = \gamma \otimes \hat{x}^{(i)} + \beta
\]
Vanishing & Exploding Gradient

• Batch Normalization

• A good strategy is to maintain an average value for the mean and standard deviation vectors during training.

• It is done by a technique known as Exponentially Weighted Moving Average (EWMA).

\[
\begin{align*}
\mu_{mb} &= \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} x^{(i)} \\
\sigma_{mb}^2 &= \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} (x^{(i)} - \mu_{mb})^2 \\
\hat{x}^{(i)} &= \frac{x^{(i)} - \mu_{mb}}{\sqrt{\sigma_{mb}^2 + \epsilon}}
\end{align*}
\]
Vanishing & Exploding Gradient

- **Batch Normalization** *(number of learnable parameters)*
- **Four parameter vectors** are learned in each batch-normalized layer:
  - $\gamma$ (the output scale vector) and $\beta$ (the output offset vector) are learned through **backpropagation**.
  - $\mu$ (the final input mean vector) and $\sigma$ (the final input standard deviation vector) are estimated using **EWMA**.

\[
\mu_{mb} = \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} x^{(i)}
\]
\[
\sigma^2_{mb} = \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} (x^{(i)} - \mu_{mb})^2
\]
\[
\hat{x}^{(i)} = \frac{x^{(i)} - \mu_{mb}}{\sqrt{\sigma^2_{mb} + \epsilon}}
\]
\[
\gamma \otimes \hat{x}^{(i)} + \beta
\]
Vanishing & Exploding Gradient

- Batch Normalization (number of learnable parameters)
- Note that $\mu$ and $\sigma$ are estimated during training.
- But they are used only after training (to replace the batch input means and standard deviations below).

$$
\mu_{mb} = \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} x^{(i)}
$$

$$
\sigma^2_{mb} = \frac{1}{N_{mb}} \sum_{i=1}^{N_{mb}} (x^{(i)} - \mu_{mb})^2
$$

$$
\hat{x}^{(i)} = \frac{x^{(i)} - \mu_{mb}}{\sqrt{\sigma^2_{mb} + \epsilon}}
$$

$$
z^{(i)} = \gamma \otimes \hat{x}^{(i)} + \beta
$$
Vanishing & Exploding Gradient

• How does the BN resolve vanishing gradient problem?
• BN resolves vanishing gradient by preventing saturation of the activation signals.

\[ z_j^{(k)} = \sum_{i \text{ layer } (k-1)} W_{ji}^{(k-1)} a_i^{(k-1)} \]

Forward Propagation

No need to worry about the activation!

Backward Propagation

\[ \frac{\partial J}{\partial W_{ji}^{(k)}} = [g'(z_j^{(k+1)})] \sum_{m \text{ layer } (k+2)} W_{mj}^{(k+1)} \delta_m^{(k+2)}]a_i^{(k)} \]
Vanishing & Exploding Gradient

- The **larger the means and variances** of the input signal $z^{(k)}$, the higher the probability of the activation function to become saturated.
- BN ensures that the input signal $z^{(k)}$ is **normalized**, thereby prevents saturation.

Forward Propagation

$$z_j^{(k)} = \sum_{i, \text{layer (k-1)}} W_{ji}^{(k-1)} a_i^{(k-1)}$$

Backward Propagation

$$a_j^{(k)} = g(z_j^{(k)})$$

$$\frac{\partial J}{\partial W_{ji}^{(k)}} = [g'(z_j^{(k+1)})] \sum_{m, \text{layer (k+2)}} W_{mj}^{(k+1)} \delta_m^{(k+2)} a_i^{(k)}$$

No need to worry about the activation!

Reduce the variance by normalizing “z”
Vanishing & Exploding Gradient

- Batch Normalization
- The BN produces **two magical effects**.
  - It considerably improves both convergence and accuracy.
- Convergence is improved by **reducing the vanishing gradient** problem.
  - The BN layers make the network **less sensitive to the initialization scheme**.
Vanishing & Exploding Gradient

- Batch Normalization
- Interestingly, even with the sigmoidal functions (tanh and logistic sigmoid) and simple unscaled initializer, it performs superb.
- The saturating activations are no more an issue.
- It enables the gradients to flow through the network seamlessly, by reducing the dependence of gradients on the scale of the parameters or of their initial values.
Vanishing & Exploding Gradient

• Batch Normalization
• It works equally well with ReLU activation.
• Thus, BN makes the Backpropagation algorithm to become apathetic to the choice of activation and initializer.
• As a consequence it is possible to afford larger learning rates to speed up the learning process.

\[
\frac{\partial J}{\partial W^{(k)}_{ji}} = [g'(z^{(k+1)}_j) \sum_{m \text{ layer}(k+2)} W^{(k+1)}_{mj} \delta^{(k+2)}_m] a^{(k)}_i
\]

BN can afford large learning rate
Vanishing & Exploding Gradient

• Batch Normalization
• However, there is an **added cost** to make this magic possible.
• We have to add a BN layer for each layer of the network, then normalize-scale-shift **each layer per iteration**.
• Thus, **each epoch takes much more** time with BN.
• It was shown that BN increases computational overhead in each iteration by approximately 30%.