Lesson 4 - From MSE to Cross Entropy & From Normal to He Initialization

Why we switch for MNIST classification

Andrea Giardina
contact@andreagiardina.com
https://www.linkedin.com/in/agiardina

From Regression to Classification

- Previously, we treated MNIST as a **regression-style** problem and used **MSE** as loss.
- But MNIST is a **multi-class classification** task (10 digits).
- MSE can work, yet it does not model **probabilities** and may yield **slower convergence**.
- **Cross Entropy** directly compares predicted probabilities with true labels.
  - Penalizes confident wrong predictions strongly.
  - Matches the probabilistic modeling of **softmax outputs**.

Improving Weight Initialization

- Previously: weights drawn from a simple normal distribution.
- Risk: **vanishing** or **exploding** activations as depth increases.
- **He Initialization** (for ReLU-like activations):

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$

- Preserves activation variance layer-to-layer $\Rightarrow$ more stable gradients.
- Practically: **faster** training and often **higher** accuracy.

The Goal: Multi-Class Classification (MNIST)

Our task is to classify images of handwritten digits (0-9).

- **Input:** An image (e.g., 28×28 pixels).
- **Output:** A probability distribution over 10 classes.

Input Image ('2')

Network Output (Prediction):

- Class 0: 1%
- Class 1: 5%
- **Class 2: 90%**
- Class 3: 1%
- ...
- Class 9: 2%

The Goal: Multi-Class Classification (MNIST)

Our task is to classify images of handwritten digits (0-9).

- **Input:** An image (e.g., 28×28 pixels).
- **Output:** A probability distribution over 10 classes.

Input Image ('2')

Network Output (Prediction):

- Class 0: 1%
- Class 1: 5%
- **Class 2: 90%**
- Class 3: 1%
- ...
- Class 9: 2%

Question: How do we ensure the output is a valid probability distribution (all positive, sums to 1)? **Answer:** The **Softmax** function.

The Output Layer: Softmax

The final layer of our network produces raw scores (logits), z_i, for each class i. Softmax converts these scores into probabilities, a_i.

Softmax Function

$$a_i = \text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

Where C is the number of classes (10 for MNIST).

Properties:

- $a_i > 0$ (Exponential is always positive)
- $\sum_{i=1}^{C} a_i = 1$ (Normalized by the sum)

We will call the vector of predicted probabilities a (or $\hat{y}$).

The Target: One-Hot Encoding

We need to compare the network's probability output a with the true label y. We can't use the number '2'.

We represent the true label y as a "one-hot" vector.

One-Hot Encoding Example (Label = 2)

$$y = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$$

Our Goal: Make the predicted distribution a as "close" as possible to the true distribution y.

- **Prediction** a**:** $[0.01, 0.05, \mathbf{0.90}, 0.01, ..., 0.02]$
- **Target** y**:** $[0, \quad 0, \quad \mathbf{1}, \quad 0, \quad ..., \quad 0]$

We need a **Loss Function** $L(a, y)$ to measure this "distance".

A Possible Loss: Mean Squared Error (MSE)

A common loss function from regression is MSE. Why not use it here?

Mean Squared Error (MSE)

$$L_{MSE} = \sum_{i=1}^{C} (a_i - y_i)^2$$

(We can average over the batch, but let's look at one sample)

The Problem: The Gradient

- Backpropagation relies on the gradient: $\frac{\partial L}{\partial z_i}$.
- The gradient for MSE (combined with Softmax) includes the term $a_i(1 - a_i)$.
- **Scenario:** Network is very wrong and very confident.
  - Target $y_i = 1$.
  - Prediction $a_i = 0.01$.
- The gradient term $a_i(1 - a_i) \approx 0.01(0.99) \approx 0.01$.
- The gradient is **tiny**! This is the **vanishing gradient problem**.

A Possible Loss: Mean Squared Error (MSE)

A common loss function from regression is MSE. Why not use it here?

Mean Squared Error (MSE)

$$L_{MSE} = \sum_{i=1}^{C} (a_i - y_i)^2$$

(We can average over the batch, but let's look at one sample)

The Problem: The Gradient

- Backpropagation relies on the gradient: $\frac{\partial L}{\partial z_i}$.
- The gradient for MSE (combined with Softmax) includes the term $a_i(1 - a_i)$.
- **Scenario:** Network is very wrong and very confident.
  - Target $y_i = 1$.
  - Prediction $a_i = 0.01$.
- The gradient term $a_i(1 - a_i) \approx 0.01(0.99) \approx 0.01$.
- The gradient is **tiny**! This is the **vanishing gradient problem**.

Result: The network learns *very slowly* precisely when it is most wrong.

A Better Loss: Cross-Entropy (CE)

Cross-Entropy comes from information theory. It measures the "inefficiency" of using the predicted distribution a to represent y.

Categorical Cross-Entropy (CE)

$$L_{CE} = -\sum_{i=1}^{C} y_i \log(a_i)$$

For one-hot y:

- $y = [0, 0, 1, 0, ...]$
- Only the correct class term survives.

Simplified Loss

$$L_{CE} = -\log(a_c)$$

If $a_c \to 1$: $L \to 0$ (good). **If** $a_c \to 0$: $L \to \infty$ (bad).

Why CE is Better: The "Magic" Gradient

The gradient of CE+Softmax is very simple:

Gradient

$$\frac{\partial L_{CE}}{\partial z_i} = a_i - y_i$$

Just **Prediction** - **Target**. No vanishing term $a_i(1 - a_i)$.

Implementation: Forward Pass (NumPy)

From Logits to Loss

Let's assume we have a batch:

- `Z`: Logits from the last layer. Shape `(N, C)`.
- `Y_true`: One-hot labels. Shape `(N, C)`.
- `N` = batch size, `C` = number of classes.

Step 1: Softmax (Stable)

```
Z_stable = Z - np.max(Z, axis=1, keepdims=True)
exp_Z = np.exp(Z_stable)
A = exp_Z / np.sum(exp_Z, axis=1, keepdims=True)
```

Step 2: Cross-Entropy Loss

```
epsilon = 1e-9
log_probs = np.log(A + epsilon)
loss_samples = -np.sum(Y_true * log_probs, axis=1)
total_loss = np.mean(loss_samples)
```

Implementation: Backward Pass (NumPy)

The Gradient for Backpropagation

- **Loss:** $L = \frac{1}{N} \sum L_{\text{sample}}$
- **Gradient:** $\frac{\partial L}{\partial z_i} = \frac{1}{N}(a_i - y_i)$

Gradient w.r.t. Logits (dZ)

```
N = Y_true.shape[0]
dZ = (A - Y_true) / N
```

Done! The complexity cancels out perfectly.

Summary

- CE measures distance between predicted (a) and true (y) distributions.
- Gradient: $\frac{\partial L}{\partial Z} = A - Y$.
- Avoids vanishing gradients $\rightarrow$ faster, stable training.

Why Do We Care About Initialization?

- Poor initialization can cause **vanishing** or **exploding** activations/gradients.
- Goal: keep the **variance** of signals roughly constant across layers (forward) and of gradients (backward).
- He (Kaiming) initialization is designed for **ReLU-like** activations (ReLU, LeakyReLU, ELU, GELU).
- Intuition: ReLU “drops” about half the inputs ($\mathbb{P}[z > 0] \approx 0.5$ if z is symmetric), so we compensate in the weights’ variance.

What is Variance?

- **Goal:** measure how much values vary around their mean (spread).
- **Population variance:** for a random variable X with mean μ,

$$\mathrm{Var}(X) = \mathbb{E}\big[(X-\mu)^2\big].$$

- **Sample variance (unbiased):** for data $x_1, \ldots, x_n$ with sample mean $\bar{x}$,

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

- **Units:** squared units of the data (e.g., if x is in volts, variance is in volts^2).
- **Standard deviation:** $\sigma = \sqrt{\mathrm{Var}(X)}$ (same units as data).
- **Key properties:** $\mathrm{Var}(X) \geq 0$; $\mathrm{Var}(aX+b) = a^2\,\mathrm{Var}(X)$; $\mathrm{Var}(X) = 0$ iff X is constant.
- **Why we care (initialization):** keeping variance stable across layers helps avoid vanishing/exploding activations and gradients.

Variance Propagation (Forward Pass)

Consider a linear layer: $z = Wa$, where $a \in \mathbb{R}^{\text{fan\_in}}$ and $W \in \mathbb{R}^{\text{fan\_out} \times \text{fan\_in}}$.

Assumptions (common for analysis)

- a and W are independent, zero-mean.
- W_{ij} are i.i.d. with variance $\text{Var}(W_{ij})$.
- Components of a have common variance $\text{Var}(a)$.

Then approximately

$$\text{Var}(z) \approx \text{fan\_in} \cdot \text{Var}(W_{ij}) \cdot \text{Var}(a).$$

With $a = \mathrm{ReLU}(z)$, using $\text{Var}(\mathrm{ReLU}(z)) \approx \frac{1}{2} \text{Var}(z)$ for symmetric z,

$$\text{Var}(a_{\text{next}}) \approx \frac{1}{2} \text{fan\_in} \, \text{Var}(W) \, \text{Var}(a).$$

To keep $\text{Var}(a_{\text{next}}) \approx \text{Var}(a)$, set $\text{Var}(W) \approx \frac{2}{\text{fan\_in}}$.

He (Kaiming) Initialization

- **Normal (Gaussian):** $W_{ij} \sim \mathcal{N}\left(0, \frac{2}{\text{fan\_in}}\right)$, i.e. $\sigma = \sqrt{\frac{2}{\text{fan\_in}}}$.
- **Uniform:** $W_{ij} \sim \mathcal{U}[-r,\, r]$ with $r = \sqrt{\frac{6}{\text{fan\_in}}}$ (since $\mathrm{Var}(\mathcal{U}[-r, r]) = \frac{r^2}{3}$).
- **Biases:** usually $b = 0$.
- **fan_in** for Dense: number of inputs to the layer..

Why it works: For ReLU-like activations, $\mathrm{Var}(\mathrm{ReLU}(z)) \approx \frac{1}{2}\,\mathrm{Var}(z)$, so using $\mathrm{Var}(W) = 2/\text{fan\_in}$ keeps variance stable layer-to-layer.

Effect of He vs Random initialization

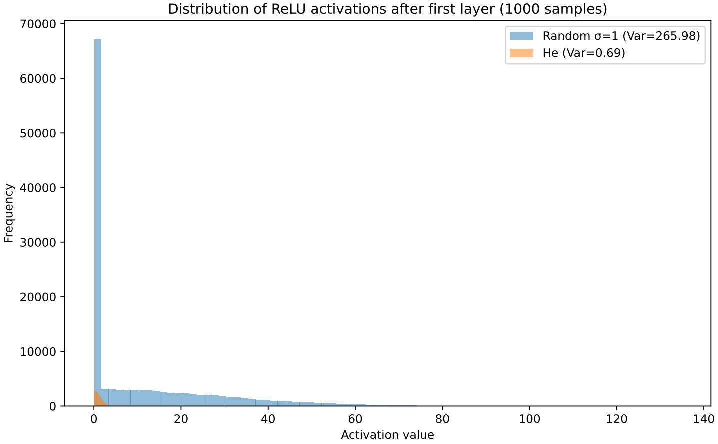

Figure: Effect of He vs Random initialization after ReLU layer.

Forward vs Backward: fan_in or fan_out?

- **fan_in** preserves activation variance *forward*. This is the standard He init for ReLU layers.
- **fan_out** preserves gradient variance *backward*. Some libraries allow a "mode" parameter.
- In practice, for MLPs/CNNs with ReLU-family, **He with fan_in** is a robust default.

Applying He Init to a Simple MNIST MLP

Example shapes:

- Input $\rightarrow$ Hidden: $784 \rightarrow 128 \quad \Rightarrow \; \sigma = \sqrt{2/784} \approx 0.0505$
- Hidden $\rightarrow$ Output: $128 \rightarrow 10 \quad \Rightarrow \; \sigma = \sqrt{2/128} = 0.125$

Tip: initialize biases to zero; ensure `float32`; set a reproducible `seed`.

NumPy: Dense He Initialization

```
import numpy as np

def he_normal(shape, fan_in):
    std = np.sqrt(2.0 / fan_in)
    return np.random.normal(0.0, std, size=shape).astype(np.float32)

def he_uniform(shape, fan_in):
    bound = np.sqrt(6.0 / fan_in) #since Var[U(-r,r)] = r^2 / 3
    return np.random.uniform(-bound, bound, size=shape)
                                            .astype(np.float32)

# Dense layer: W in R[hidden, in_dim]
in_dim, hidden, out_dim = 784, 128, 10
W1 = he_normal((hidden, in_dim), fan_in=in_dim)
b1 = np.zeros((hidden,), dtype=np.float32)
W2 = he_normal((out_dim, hidden), fan_in=hidden)
b2 = np.zeros((out_dim,), dtype=np.float32)
```

MNIST Checklist (ReLU MLP/CNN)

- Use **He init (fan_in)** for all ReLU-based layers.
- Biases $\rightarrow$ 0.
- Cross-entropy loss with **logits** (no final activation before `softmax` in the loss).
- Verify training: monitor loss/accuracy; ensure neither blows up nor stalls at start.

References

- K. He, X. Zhang, S. Ren, J. Sun. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, ICCV 2015.

Thanks!