Proof that Naive-Softmax Loss equals Cross-Entropy Loss

Given:

- $\mathbf{y}$: true distribution (one-hot vector)
- $\hat{\mathbf{y}}$: predicted distribution (softmax output)
- $\hat{y}_o$: the predicted probability for the true class o

Naive-softmax loss (Equation 2) is typically defined as:

$$J_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U}) = -\log P(O = o|C = c) = -\log(\hat{y}_o) = -\log\left(\frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{w=1}^{W} \exp(\mathbf{u}_w^\top \mathbf{v}_c)}\right)$$

Cross-entropy loss between $\mathbf{y}$ and $\hat{\mathbf{y}}$ is defined as:

$$CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{w=1}^{W} y_w \log(\hat{y}_w)$$

Proof:
Since $\mathbf{y}$ is a one-hot vector where the true class is at position o:

- $y_o = 1$ (for the true class)
- $y_w = 0$ for all $w \neq o$

Substituting into the cross-entropy formula:

$$CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{w=1}^{W} y_w \log(\hat{y}_w) = -y_o \log(\hat{y}_o) - \sum_{w \neq o} y_w \log(\hat{y}_w)$$

Since $y_w = 0$ for all $w \neq o$:

$$CE(\mathbf{y}, \hat{\mathbf{y}}) = -1 \cdot \log(\hat{y}_o) - 0 = -\log(\hat{y}_o)$$

Therefore:

$$J_{\text{naive-softmax}} = -\log(\hat{y}_o) = CE(\mathbf{y}, \hat{\mathbf{y}})$$

This completes the proof showing that the naive-softmax loss is equivalent to the cross-entropy loss when the true distribution is a one-hot vector.

(b) Gradient Analysis of Naive-Softmax Loss

(b)(i) Partial Derivative with respect to $\mathbf{v}_c$

Answer:

$$\frac{\partial J_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})}{\partial \mathbf{v}_c} = \mathbf{U}^\top(\hat{\mathbf{y}} - \mathbf{y})$$

Derivation:

The naive softmax loss is:

$$J_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U}) = -\log P(O=o|C=c) = -\log\left(\frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{w\in\text{Vocab}}\exp(\mathbf{u}_w^\top \mathbf{v}_c)}\right)$$

This simplifies to:

$$J = -\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_w \exp(\mathbf{u}_w^\top \mathbf{v}_c)\right)$$

Taking the partial derivative with respect to $\mathbf{v}_c$:

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{v}_c} &= -\mathbf{u}_o + \frac{\sum_w \exp(\mathbf{u}_w^\top \mathbf{v}_c)\cdot \mathbf{u}_w}{\sum_w \exp(\mathbf{u}_w^\top \mathbf{v}_c)} \\ &= -\mathbf{u}_o + \sum_w \frac{\exp(\mathbf{u}_w^\top \mathbf{v}_c)}{\sum_{w'}\exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)}\cdot \mathbf{u}_w \\ &= -\mathbf{u}_o + \sum_w \hat{y}_w \cdot \mathbf{u}_w\end{aligned}$$

In vectorized form, since $\mathbf{u}_o = \mathbf{U}^\top \mathbf{y}$ (where $\mathbf{y}$ is the one-hot vector):

$$\frac{\partial J}{\partial \mathbf{v}_c} = \mathbf{U}^\top \hat{\mathbf{y}} - \mathbf{U}^\top \mathbf{y} = \mathbf{U}^\top(\hat{\mathbf{y}} - \mathbf{y})$$

(b)(ii) When is the Gradient Zero?

Answer:

The gradient equals zero when:

$$\hat{\mathbf{y}} = \mathbf{y}$$

This occurs when the predicted probability distribution equals the true distribution, i.e., when the model predicts the correct outside word with probability 1 and all other words with probability 0.

(b)(iii) Interpretation of the Gradient Terms

The gradient $\mathbf{U}^\top(\hat{\mathbf{y}} - \mathbf{y})$ consists of two terms:

1. **$\mathbf{U}^\top \hat{\mathbf{y}}$ (positive term):** This is a weighted sum of all outside word vectors, weighted by their predicted probabilities. When subtracted from $\mathbf{v}_c$ during gradient descent, it **pushes the center word vector away** from the average position of all words that the model incorrectly believes are likely to be context words. This reduces false positive predictions.

2. **$-\mathbf{U}^\top \mathbf{y} = -\mathbf{u}_o$ (negative term):** This is the negative of the true outside/context word vector $\mathbf{u}_o$. When subtracted from $\mathbf{v}_c$ during gradient descent, the negative sign causes this term to **pull the center word vector toward** the actual context word $\mathbf{u}_o$, making them more similar. This increases the dot product $\mathbf{u}_o^\top \mathbf{v}_c$ and thus the probability of correctly predicting the true context word.

Together, these updates improve $\mathbf{v}_c$ by making it more similar to actual context words and less similar to words that shouldn't be context words.

(c) L2 Normalization in Word Embeddings

Question:

In a binary sentiment classification task (positive/negative) where we sum word embeddings to classify phrases, when would L2 normalization (using $\mathbf{u}/||\mathbf{u}||_2$ instead of raw $\mathbf{u}$) take away useful information? When would it not?

Answer:

Key Mathematical Insight from the Hint:

Consider the case where $\mathbf{u}_x = \alpha \mathbf{u}_y$ for some scalar α and words $x \neq y$.

When $\alpha > 0$ (same direction):

$$\frac{\mathbf{u}_x}{||\mathbf{u}_x||_2} = \frac{\alpha \mathbf{u}_y}{||\alpha \mathbf{u}_y||_2} = \frac{\alpha \mathbf{u}_y}{|\alpha| \cdot ||\mathbf{u}_y||_2} = \frac{\alpha \mathbf{u}_y}{\alpha \cdot ||\mathbf{u}_y||_2} = \frac{\mathbf{u}_y}{||\mathbf{u}_y||_2}$$

Thus, when $\alpha > 0$, the normalized vectors are **identical**, collapsing words with different magnitudes but same direction into the same representation.

When L2 normalization TAKES AWAY useful information:

Normalization is detrimental when **magnitude encodes meaningful semantic differences**:

1. **Sentiment intensity encoded in magnitude:** Consider words with the same sentiment direction but different intensities:

   - "excellent" $\rightarrow \mathbf{u}_{\text{excellent}}$ (large positive magnitude)
   - "good" $\rightarrow \mathbf{u}_{\text{good}} = 0.3 \cdot \mathbf{u}_{\text{excellent}}$ (same direction, smaller magnitude)

   For the phrase "This is excellent", the raw sum would give a strong positive score. For "This is good", the raw sum gives a weaker positive score, appropriately reflecting the intensity difference.

   After normalization, both contribute identically to the sum, so "This is excellent" and "This is good" would receive the same classification score, losing the nuanced intensity information.

2. **Word importance/weight:** If sentiment-bearing words ("excellent", "terrible") have larger magnitudes than neutral words ("is", "the"), the raw embeddings naturally weight sentiment words more heavily in the sum. Normalization makes all words contribute equally, potentially diluting the sentiment signal with neutral words.

3. **Multiple occurrences matter differently:** In raw form, "good good good" would sum to a stronger positive signal than "good". With normalization, multiple identical words contribute the exact same amount as a single occurrence, losing frequency information that might indicate emphasis.

When L2 normalization DOES NOT take away useful information:

Normalization is benign or beneficial when **only direction matters**:

1. **Magnitude is arbitrary or noise:** If magnitude differences arise from training artifacts (e.g., word frequency in corpus) rather than semantic meaning, normalization removes noise and focuses on the meaningful directional information. Different words with the same sentiment should contribute similarly.

2. **Opposite sentiments preserved:** When $\alpha < 0$, normalization preserves the opposition:

$$\frac{\mathbf{u}_x}{||\mathbf{u}_x||_2} = \frac{\alpha \mathbf{u}_y}{|\alpha| \cdot ||\mathbf{u}_y||_2} = \frac{\alpha}{|\alpha|} \cdot \frac{\mathbf{u}_y}{||\mathbf{u}_y||_2} = -\frac{\mathbf{u}_y}{||\mathbf{u}_y||_2}$$

   This means "good" and "bad" (if $\mathbf{u}_{\text{bad}} = -\alpha \mathbf{u}_{\text{good}}$ with $\alpha > 0$) will still point in opposite directions after normalization, preserving the essential positive vs. negative distinction needed for binary classification.

3. **Balanced phrase representation:** When we want each word to contribute equally (e.g., to prevent longer phrases from having artificially larger scores just due to more words), normalization ensures fair contribution from each word based purely on sentiment direction, not arbitrary magnitude.

4. **Words are linearly independent:** If different words have embeddings that are not scalar multiples of each other (i.e., $\mathbf{u}_x \neq \alpha \mathbf{u}_y$ for any α), normalization doesn't cause the collision problem described above. The directional differences are preserved.

Summary: L2 normalization removes useful information when magnitude encodes meaningful properties like sentiment intensity or word importance. It preserves essential information when only directional relationships matter for classification, particularly the positive vs. negative distinction ($\alpha < 0$ case)

(d) Partial Derivative with respect to U

Question:

Write down the partial derivative of $J_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})$ with respect to $\mathbf{U}$, broken down in terms of the column vectors $\frac{\partial J}{\partial \mathbf{u}_1}, \frac{\partial J}{\partial \mathbf{u}_2}, \ldots, \frac{\partial J}{\partial \mathbf{u}_{|\text{Vocab}|}}$ (do not further expand these terms).

Answer:

The partial derivative of J with respect to the matrix $\mathbf{U}$ is:

$$\frac{\partial J_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})}{\partial \mathbf{U}} = \begin{bmatrix} \frac{\partial J}{\partial \mathbf{u}_1} & \frac{\partial J}{\partial \mathbf{u}_2} & \cdots & \frac{\partial J}{\partial \mathbf{u}_{|\text{Vocab}|}} \end{bmatrix}$$

This is a matrix of the same shape as $\mathbf{U}$, where each column is the partial derivative with respect to the corresponding column vector of $\mathbf{U}$.

(e) Partial Derivatives with respect to Individual Outside Word Vectors

Question:

Compute the partial derivatives of $J_{\text{naive-softmax}}(\mathbf{v}_c, o, \mathbf{U})$ with respect to each outside word vector $\mathbf{u}_w$. Consider two cases: when $w = o$ (the true outside word) and when $w \neq o$ (all other words).

Answer:

Case 1: When $w = o$ (the true outside word)

$$\frac{\partial J}{\partial \mathbf{u}_o} = (\hat{y}_o - 1)\mathbf{v}_c$$

Case 2: When $w \neq o$ (all other words)

$$\frac{\partial J}{\partial \mathbf{u}_w} = \hat{y}_w \mathbf{v}_c$$

Unified form: Both cases can be written as:

$$\frac{\partial J}{\partial \mathbf{u}_w} = (\hat{y}_w - y_w)\mathbf{v}_c$$

where $y_w = 1$ if $w = o$ and $y_w = 0$ if $w \neq o$.

Derivation:

Recall that the naive softmax loss is:

$$J = -\log P(O = o|C = c) = -\log\left(\frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{w'} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)}\right)$$

This simplifies to:

$$J = -\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_{w'} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)\right)$$

Case 1: $w = o$ (true outside word)
Taking the derivative with respect to $\mathbf{u}_o$:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{u}_o} &= \frac{\partial}{\partial \mathbf{u}_o}\left[-\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_{w'} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)\right)\right] \\
&= -\mathbf{v}_c + \frac{1}{\sum_{w'} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)} \cdot \exp(\mathbf{u}_o^\top \mathbf{v}_c) \cdot \mathbf{v}_c \\
&= -\mathbf{v}_c + \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{w'} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)} \cdot \mathbf{v}_c \\
&= -\mathbf{v}_c + \hat{y}_o \mathbf{v}_c \\
&= (\hat{y}_o - 1)\mathbf{v}_c
\end{aligned}$$

Note that $y_o = 1$ (since o is the true word), so this can also be written as $(\hat{y}_o - y_o)\mathbf{v}_c$.
Case 2: $w \neq o$ (other words)
For any word $w \neq o$, the first term $-\mathbf{u}_o^\top \mathbf{v}_c$ does not depend on $\mathbf{u}_w$, so its derivative is zero:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{u}_w} &= \frac{\partial}{\partial \mathbf{u}_w}\left[-\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_{w'} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)\right)\right] \\
&= 0 + \frac{1}{\sum_{w'} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)} \cdot \exp(\mathbf{u}_w^\top \mathbf{v}_c) \cdot \mathbf{v}_c \\
&= \frac{\exp(\mathbf{u}_w^\top \mathbf{v}_c)}{\sum_{w'} \exp(\mathbf{u}_{w'}^\top \mathbf{v}_c)} \cdot \mathbf{v}_c \\
&= \hat{y}_w \mathbf{v}_c
\end{aligned}$$

Note that $y_w = 0$ (since w is not the true word), so this can also be written as $(\hat{y}_w - y_w)\mathbf{v}_c = (\hat{y}_w - 0)\mathbf{v}_c = \hat{y}_w \mathbf{v}_c$.

2. Machine Learning & Neural Networks

(a) Adam Optimizer

(a)(i) Momentum - Reducing Update Variance (2 points)

Question: Briefly explain how using $\mathbf{m}$ stops the updates from varying as much and why this low variance may be helpful to learning.

Answer:

Momentum reduces update variance by accumulating a weighted average of past gradients rather than using only the current minibatch gradient. Since $\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1)\nabla_{\theta_t} J_{\text{minibatch}}(\theta_t)$ with $\beta_1 \approx 0.9$, the momentum vector $\mathbf{m}$ smooths out noisy gradient estimates by incorporating historical gradient information. This dampening effect means that random fluctuations in individual minibatch gradients tend to cancel out, while consistent gradient directions accumulate and reinforce over time.

Lower variance updates are helpful because they provide a more stable and reliable estimate of the true gradient direction. This stability allows the optimizer to use larger learning rates without overshooting, helps the model navigate flat regions or narrow valleys in the loss landscape more efficiently, and reduces oscillations that can slow down or destabilize training. Additionally, momentum helps the optimizer build velocity in consistent directions, allowing it to accelerate through plateaus and escape shallow local minima more effectively.

(a)(ii) Adaptive Learning Rates - Dividing by $\sqrt{\mathbf{v}}$ (2 points)

Question: Since Adam divides the update by $\sqrt{\mathbf{v}}$, which of the model parameters will get larger updates? Why might this help with learning?

Answer:

Parameters with **smaller gradient magnitudes** will receive larger updates. Since $\mathbf{v}_{t+1}$ tracks the exponentially weighted average of squared gradients ($\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2)(\nabla_{\theta_t} J \odot \nabla_{\theta_t} J)$), parameters with consistently large gradients will have large $\mathbf{v}$ values. The update rule $\theta_{t+1} = \theta_t - \alpha \mathbf{m}_{t+1}/\sqrt{\mathbf{v}_{t+1}}$ divides by $\sqrt{\mathbf{v}}$, which means parameters with small gradients (small $\mathbf{v}$) get divided by a small number, resulting in larger relative updates, while parameters with large gradients get scaled down.

This adaptive learning rate mechanism is beneficial because it automatically balances the learning rates across different parameters. Parameters that receive consistently large gradients (which might correspond to frequently updated features or high-variance inputs) get their updates dampened to prevent overshooting, while parameters with small gradients (which might correspond to rarely activated features or infrequent patterns) get amplified updates to ensure they still learn effectively. This is particularly helpful in neural networks where different parameters can have vastly different gradient scales—for example, parameters in early layers versus late layers, or parameters associated with rare versus common features. By normalizing the update magnitudes, Adam ensures more uniform and stable learning across all parameters, leading to faster convergence and better generalization.

(b) Dropout Regularization

Dropout is a regularization technique where, during training, units in the hidden layer $\mathbf{h}$ are randomly set to zero with probability p_{drop}, and then the result is multiplied by a constant γ:

$$\mathbf{h}_{\text{drop}} = \gamma \mathbf{d} \odot \mathbf{h}$$

where $\mathbf{d} \in \{0,1\}^{D_h}$ is a mask vector (each entry is 0 with probability p_{drop} and 1 with probability $1 - p_{\text{drop}}$), and γ is chosen such that:

$$\mathbb{E}_{p_{\text{drop}}}[\mathbf{h}_{\text{drop}}]_i = h_i \quad \text{for all } i \in \{1, \ldots, D_h\}$$

(b)(i) Value of γ in terms of p_{drop} (2 points)

Answer:

$$\gamma = \frac{1}{1 - p_{\text{drop}}}$$

Derivation:

We need to find γ such that the expected value of the dropped-out hidden layer equals the original hidden layer. Starting with the definition:

$$\mathbb{E}[\mathbf{h}_{\text{drop}}]_i = \mathbb{E}[\gamma d_i h_i] = \gamma \mathbb{E}[d_i] h_i$$

Since d_i is a Bernoulli random variable that takes value 0 with probability p_{drop} and 1 with probability $1 - p_{\text{drop}}$:

$$\mathbb{E}[d_i] = 0 \cdot p_{\text{drop}} + 1 \cdot (1 - p_{\text{drop}}) = 1 - p_{\text{drop}}$$

Therefore:

$$\mathbb{E}[\mathbf{h}_{\text{drop}}]_i = \gamma(1 - p_{\text{drop}}) h_i$$

For this to equal h_i (as required by the constraint):

$$\begin{aligned} \gamma(1 - p_{\text{drop}}) h_i &= h_i \\ \gamma(1 - p_{\text{drop}}) &= 1 \\ \gamma &= \frac{1}{1 - p_{\text{drop}}} \end{aligned}$$

For example, if $p_{\text{drop}} = 0.5$ (dropping half the units), then $\gamma = \frac{1}{0.5} = 2$, which compensates by doubling the remaining activations so that the expected output magnitude stays constant.

(b)(ii) When to Apply Dropout (2 points)

Question: Why should dropout be applied during training? Why should dropout NOT be applied during evaluation?

Answer:

During training, dropout should be applied because it acts as a powerful regularization technique that prevents overfitting. By randomly dropping units in each minibatch, dropout forces the network to learn robust and redundant representations that don't rely on the presence of specific neurons. This prevents co-adaptation of neurons (where neurons become overly dependent on each other) and encourages each neuron to learn independently useful features. The network essentially learns to work well even when parts of it are missing, which improves generalization to unseen data.

During evaluation, dropout should NOT be applied because we want deterministic, optimal predictions using all available learned features. Applying dropout at test time would introduce unnecessary randomness and reduce model performance by discarding useful information. Instead, we use all neurons with their learned weights, and the scaling factor $\gamma = \frac{1}{1-p_{\text{drop}}}$ applied during training ensures that the expected activations during training match the actual activations at test time (when all units are active). This principle is sometimes called "inverted dropout" and ensures smooth transition from training to evaluation without requiring weight scaling at test time.