1.1 LLM Base Knowledge

> This guide provides a deep dive into foundational concepts for modern data science and deep learning interviews, covering the Attention Mechanism, advanced Optimization Algorithms, Regularization Techniques, and common Loss Functions. It begins with a thorough explanation of the theory behind each topic, complete with mathematical formulations and proofs. This is followed by a curated set of theoretical and practical interview questions, including full Python and PyTorch code implementations, to solidify understanding and prepare for real-world interview scenarios.

Knowledge Section

This section breaks down the core theoretical knowledge required to understand advanced machine learning models. We will cover the mechanics of attention, the evolution of optimization algorithms, methods to prevent overfitting, and the mathematical basis for classification loss functions.

(Note: The information herein is current as of my last update in early 2023. While the core principles are timeless, newer model variants or optimizers may have emerged since.)

The Attention Mechanism

The Attention Mechanism, originally proposed for machine translation in Bahdanau et al. (2014) and refined in Vaswani et al.'s "Attention Is All You Need" (2017), has become a cornerstone of modern deep learning, particularly in Natural Language Processing (NLP). It allows a model to dynamically focus on the most relevant parts of the input sequence when producing an output, overcoming the fixed-length context vector bottleneck of traditional sequence-to-sequence models like RNNs.

The Core Idea: Query, Key, and Value

At its heart, attention can be described as a process of mapping a **Query (Q)** and a set of **Key-Value (K-V)** pairs to an output.

- **Query (Q):** Represents the current context or the element we are trying to compute an output for. For example, in a decoder, it could be the representation

of the previously generated word.
- **Keys (K):** A set of vectors representing different parts of the input sequence. Each key is associated with a value.
- **Values (V):** A set of vectors that contain the actual information from the input sequence. Each value corresponds to a key.

The process involves three main steps:

1. **Calculate Attention Scores:** A compatibility function is used to compute a score between the Query and each Key. A higher score means the corresponding Value is more relevant to the Query. The most common compatibility function is the dot product.
2. **Normalize Scores to Weights:** The raw scores are passed through a `softmax` function to convert them into a probability distribution. These normalized scores are the **attention weights**, and they sum to 1.
3. **Compute Weighted Sum:** The attention weights are multiplied by their corresponding Value vectors, and the results are summed up to produce the final output vector. This output is a weighted representation of the input values, where the weights are determined by the query's relevance to each key.

Scaled Dot-Product Attention

The Transformer architecture popularized a specific implementation called **Scaled Dot-Product Attention**.

Given a query `Q`, keys `K`, and values `V`, the output is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Let's break down this formula:

- $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, $V \in \mathbb{R}^{m \times d_v}$
    - n: sequence length of queries.
    - m: sequence length of keys/values.
    - d_k: dimension of queries and keys.
    - d_v: dimension of values.
- QK^T: This computes the dot product between every query and every key,

resulting in a score matrix of size `(n x m)`.

- $\frac{1}{\sqrt{d_k}}$: This is the **scaling factor**. For large values of d_k, the dot products QK^T can grow very large in magnitude. This pushes the `softmax` function into regions where its gradients are extremely small, leading to the vanishing gradients problem during training. Scaling by $\sqrt{d_k}$ counteracts this effect, keeping the variance of the inputs to the softmax function at approximately 1.
- $\mathrm{softmax}(\cdot)$: This function is applied row-wise to the scaled score matrix, converting the scores into attention weights that sum to 1 for each query.
- $(\cdot)V$: The resulting matrix of attention weights (size `n x m`) is multiplied by the `V` matrix (size `m x d_v`), producing the final output of size `n x d_v`.

Multi-Head Attention

Instead of performing a single attention function with high-dimensional keys, values, and queries, it was found to be more beneficial to linearly project the queries, keys, and values `h` times (the number of "heads") to different, learned, linear subspaces. Attention is then performed in parallel on each of these projected versions.

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)W^O$$

where $\mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.

The projection matrices W_i^Q, W_i^K, W_i^V and the output projection matrix W^O are all learned during training.

Why is this useful? Multi-Head Attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging can dilute important signals. Multiple heads allow each head to specialize and focus on a different aspect of the input (e.g., one head might track syntactic relationships, another might track semantic relationships).

Self-Attention vs. Cross-Attention

- **Self-Attention:** This is the mechanism used within the Transformer's encoder and decoder layers. Here, `Q`, `K`, and `V` all come from the same source sequence. For example, to compute the representation for the word "it" in the sentence "The animal didn't cross the street because it was too tired", self-attention allows "it" to attend to all other words in the sentence and learn that "animal" is the most

relevant antecedent.

- **Cross-Attention:** This is used in the decoder part of a Transformer. The `Q` comes from the decoder's own sequence (the output being generated), while the `K` and `V` come from the encoder's output. This allows the decoder to look back and focus on the most relevant parts of the original input sentence while generating the output sequence.

Positional Encoding

Since the attention mechanism itself does not inherently process sequential order (it treats the input as a "bag of vectors"), the Transformer model needs a way to incorporate information about the relative or absolute positions of tokens in the sequence. This is achieved by adding **Positional Encodings** to the input embeddings. These are vectors that are added to the token embeddings, providing a unique signal for each position. The original paper used sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

where `pos` is the position and `i` is the dimension. This choice allows the model to easily learn to attend to relative positions, since for any fixed offset `k`, PE_{pos+k} can be represented as a linear function of PE_{pos}.

Optimization Algorithms

Optimization algorithms are the engines that power model training. They iteratively adjust the model's parameters (weights and biases) to minimize a loss function.

Gradient Descent and Its Variants

- **Batch Gradient Descent (BGD):** Computes the gradient of the loss function with respect to the parameters for the *entire* training dataset.
    - **Pros:** Guaranteed to converge to the global minimum for convex objectives and a local minimum for non-convex objectives.

    - **Cons:** Very slow and memory-intensive for large datasets.
- **Stochastic Gradient Descent (SGD):** Computes the gradient and updates the parameters for *each individual* training example.
    - **Pros:** Much faster and can escape shallow local minima due to the noisy updates.
    - **Cons:** High variance in updates, causing the loss to fluctuate heavily.
- **Mini-Batch Gradient Descent:** A compromise that computes the gradient on small, random batches of training data. This is the de-facto standard.
    - **Pros:** Combines the efficiency of SGD with the more stable convergence of BGD. Allows for vectorized operations, leading to computational efficiency.

Momentum

SGD can be slow when navigating ravines (areas where the loss landscape curves more steeply in one dimension than another). The **Momentum** method helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction γ of the previous update vector to the current update vector.

Let w_t be the parameters at step t, η be the learning rate, and $\nabla L(w_t)$ be the gradient. The momentum update is:

$$v_t = \gamma v_{t-1} + \eta \nabla L(w_t)$$

$$w_{t+1} = w_t - v_t$$

Here, v_t is the "velocity" vector that accumulates past gradients. A typical value for γ is 0.9.

Adaptive Learning Rate Algorithms

These algorithms eliminate the need to manually tune the learning rate by adapting it for each parameter based on past gradients.

- **Adagrad (Adaptive Gradient Algorithm):** Adapts the learning rate on a per-parameter basis, performing larger updates for infrequent parameters and smaller updates for frequent parameters. It scales the learning rate inversely proportional

to the square root of the sum of all past squared gradients.
    - **Pros:** Excellent for sparse data (e.g., word embeddings).
    - **Cons:** The learning rate can become infinitesimally small over time, effectively stopping the training process.
- **RMSprop (Root Mean Square Propagation):** Addresses Adagrad's diminishing learning rate issue by using an exponentially decaying moving average of squared gradients instead of summing them all.

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)(\nabla L(w_t))^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla L(w_t)$$

The decay rate β controls the memory of past gradients.

Adam: Adaptive Moment Estimation

Adam is arguably the most popular optimization algorithm in deep learning today. It combines the ideas of both Momentum (first-order moment) and RMSprop (second-order moment).

1. **First Moment (Mean):** It maintains an exponentially decaying average of past gradients, similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla L(w_t)$$

2. **Second Moment (Uncentered Variance):** It maintains an exponentially decaying average of past squared gradients, similar to RMSprop.

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla L(w_t))^2$$

3. **Bias Correction:** Because m_t and v_t are initialized as vectors of 0s, they are biased towards zero, especially during the initial steps. Adam corrects for this bias:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

4. **Parameter Update:** The final update rule uses the corrected moments:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Default values suggested in the paper are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

- **AdamW:** A common variant that improves upon Adam by decoupling the weight decay from the adaptive learning rate mechanism. In standard Adam, L2 regularization is often implemented by adding the term to the gradient, which interacts with the adaptive learning rates. AdamW applies the weight decay directly to the weights *after* the Adam step, leading to better generalization performance.

Regularization Techniques to Combat Overfitting

Overfitting occurs when a model learns the training data too well, capturing not just the underlying patterns but also the noise. This results in excellent performance on the training set but poor performance on unseen data (high variance, low bias). Regularization refers to any technique used to prevent overfitting.

L1 and L2 Regularization

These are the most common types of regularization. They work by adding a penalty term to the loss function, which discourages the model's weights from becoming too large.

The total loss becomes: $L_{total} = L_{original} + \lambda \cdot R(w)$, where λ is the regularization strength.

- **L2 Regularization (Ridge Regression / Weight Decay):** The penalty is the squared L2 norm of the weights.

$$R(w) = \|w\|_2^2 = \sum_i w_i^2$$

Effect: L2 encourages weights to be small and diffuse. It penalizes large weights

heavily, but does not force them to be exactly zero. This results in a dense model where most weights are small but non-zero.

- **L1 Regularization (Lasso):** The penalty is the L1 norm of the weights.

$$R(w) = \|w\|_1 = \sum_i |w_i|$$

Effect: L1 encourages **sparsity**. It can force some weights to become exactly zero, effectively performing feature selection by removing irrelevant features from the model.

Geometric Interpretation:
Imagine a 2D weight space (w_1, w_2). The original loss function creates elliptical contour lines. The regularization term defines a constraint region: a circle for L2 ($w_1^2 + w_2^2 \leq C$) and a diamond for L1 ($|w_1| + |w_2| \leq C$). The optimal solution is the point where the loss contour first touches the constraint region. For L1, this is highly likely to occur at a corner of the diamond, where one of the weights is zero. For L2, it's likely to occur on the smooth boundary of the circle, where both weights are non-zero.

Other Regularization Techniques

- **Dropout:** During training, randomly sets a fraction of neuron activations to zero at each update step. This prevents neurons from co-adapting too much and forces the network to learn more robust features. At test time, all neurons are used, but their outputs are scaled down by the dropout probability.
- **Early Stopping:** Monitor the model's performance on a validation set during training. Stop the training process when the validation loss stops decreasing and starts to increase, even if the training loss is still decreasing. This prevents the model from continuing to overfit the training data.
- **Data Augmentation:** Artificially increase the size of the training dataset by creating modified copies of existing data (e.g., for images, apply rotations, flips, zooms; for text, use back-translation or synonym replacement). This exposes the model to more variations and helps it generalize better.

Loss Functions for Classification

A loss function (or cost function) measures the discrepancy between the model's prediction and the true label. The goal of training is to minimize this function.

Cross-Entropy Loss

Cross-entropy is the most common loss function for classification tasks. It originates from information theory and measures the "distance" between two probability distributions: the true distribution (the one-hot encoded label) and the predicted distribution (the model's output probabilities).

Binary Cross-Entropy (BCE)

Used for binary (two-class) classification problems where the output is a single probability p (the probability of the positive class). The true label y is either 0 or 1.

The BCE loss for a single training example is:

$$L(y, p) = -[y \log(p) + (1 - y) \log(1 - p)]$$

- If the true label $y = 1$, the loss is $-\log(p)$. To minimize this, the model must predict p close to 1.
- If the true label $y = 0$, the loss is $-\log(1 - p)$. To minimize this, the model must predict p close to 0.

This is often used with a final Sigmoid activation layer, which squashes the model's raw output (logit) into the range (0, 1).

Categorical Cross-Entropy

Used for multi-class classification problems. The model outputs a vector of probabilities $\mathbf{p}$ (one for each class), typically via a `softmax` activation function. The true label $\mathbf{y}$ is a one-hot encoded vector.

The loss for a single training example is:

$$L(\mathbf{y}, \mathbf{p}) = -\sum_{c=1}^{C} y_c \log(p_c)$$

Since $\mathbf{y}$ is one-hot, only the term corresponding to the correct class ($y_c = 1$) is non-zero. So, the formula simplifies to $-\log(p_{correct\_class})$. The model is penalized based on how low its predicted probability is for the correct class.

Numerical Stability: Directly computing $\log(\text{softmax}(z))$ can be numerically unstable. A common practice in frameworks like PyTorch is to combine them into a single, more stable function like `torch.nn.CrossEntropyLoss`, which often computes `LogSoftmax` and then applies `NLLLoss` (Negative Log Likelihood Loss).

Interview Questions

Theoretical Questions

Question 1: Derive the Scaled Dot-Product Attention formula and provide a rigorous justification for the scaling factor $\frac{1}{\sqrt{d_k}}$.

Answer:

The goal of attention is to compute a weighted sum of Value vectors, where the weights are determined by the compatibility of a Query with corresponding Key vectors.

1. **Step 1: Compatibility Score.** We need a function to measure the compatibility between a query vector $q \in \mathbb{R}^{d_k}$ and a key vector $k \in \mathbb{R}^{d_k}$. A simple and effective choice is the dot product: $score = q \cdot k = q^T k$. For matrices $Q \in \mathbb{R}^{n \times d_k}$ and $K \in \mathbb{R}^{m \times d_k}$, we compute all scores simultaneously with the matrix multiplication $S = QK^T$, where $S \in \mathbb{R}^{n \times m}$.
2. **Step 2: Justification for the Scaling Factor.**
   Let's analyze the properties of the dot product $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$. Assume that the components of the query and key vectors, q_i and k_i, are independent random variables with mean 0 and variance 1.
   - The mean of the dot product is $E[q \cdot k] = E[\sum_{i=1}^{d_k} q_i k_i] = \sum_{i=1}^{d_k} E[q_i k_i]$. Since they are independent, $E[q_i k_i] = E[q_i]E[k_i] = 0 \cdot 0 = 0$.

- The variance of the dot product is $\text{Var}(q \cdot k) = \text{Var}(\sum_{i=1}^{d_k} q_i k_i)$. Since they are independent, this is $\sum_{i=1}^{d_k} \text{Var}(q_i k_i)$.
- $\text{Var}(q_i k_i) = E[(q_i k_i)^2] - (E[q_i k_i])^2 = E[q_i^2]E[k_i^2] - 0 = \text{Var}(q_i) \cdot \text{Var}(k_i) = 1 \cdot 1 = 1$.
- Therefore, $\text{Var}(q \cdot k) = \sum_{i=1}^{d_k} 1 = d_k$.

This means the standard deviation of the dot product is $\sqrt{d_k}$. As the dimension d_k grows, the variance of the scores increases, meaning the scores become more spread out. When these large-magnitude scores are fed into a softmax function, the function saturates. The softmax output will be close to a one-hot vector (one value near 1, others near 0). This results in extremely small gradients for the non-maximal scores, hindering the learning process (vanishing gradients).
To counteract this, we scale the scores by dividing by $\sqrt{d_k}$. The new scaled score is $s'_{ij} = \frac{(QK^T)_{ij}}{\sqrt{d_k}}$. The variance of this scaled score becomes $\text{Var}(s'_{ij}) = \frac{1}{(\sqrt{d_k})^2}\text{Var}((QK^T)_{ij}) = \frac{d_k}{d_k} = 1$. This keeps the input to the softmax function well-behaved regardless of the dimension d_k, ensuring stable gradients.

3. **Step 3: Normalization and Weighted Sum.**
   We apply the softmax function to the scaled scores to get attention weights $\alpha = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$. Finally, we compute the weighted sum of the Value vectors: $\text{Output} = \alpha V$.

Combining these steps gives the final formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Question 2: Explain the full Adam optimization algorithm, including the bias correction steps. Why is bias correction necessary?

Answer:

The Adam (Adaptive Moment Estimation) algorithm computes adaptive learning rates for each parameter by storing an exponentially decaying average of past gradients (first moment) and past squared gradients (second moment).

Algorithm Steps:
Given parameters w_t at timestep t, learning rate η, exponential decay rates β_1, β_2, and a small constant ϵ.

1. **Compute Gradient:** Calculate the gradient of the loss function with respect to the current parameters: $g_t = \nabla L(w_t)$.
2. **Update First Moment (Momentum):** Update the moving average of the gradients.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

3. **Update Second Moment (RMSprop-like):** Update the moving average of the squared gradients.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

4. **Bias Correction:** Compute bias-corrected estimates for the first and second moments.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

5. **Parameter Update:** Update the parameters using the corrected moments.

$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Why is Bias Correction Necessary?

The moment vectors m_t and v_t are initialized to zero vectors ($m_0 = 0, v_0 = 0$). Let's analyze the expectation of m_t without correction.
The update rule for m_t can be unrolled:
$m_t = (1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i$.

Let's find its expectation, assuming $E[g_i]$ is the true gradient $E[g_t]$:
$E[m_t] = E[(1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i] = (1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} E[g_i]$
$E[m_t] = E[g_t](1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i}$

The summation is a geometric series: $\sum_{i=1}^{t} \beta_1^{t-i} = \sum_{j=0}^{t-1} \beta_1^j = \frac{1-\beta_1^t}{1-\beta_1}$.

Substituting this back:
$E[m_t] = E[g_t](1-\beta_1)\frac{1-\beta_1^t}{1-\beta_1} = E[g_t](1-\beta_1^t).$

This shows that the expectation of our moment estimate m_t is not the true gradient expectation $E[g_t]$, but is scaled by a factor of $(1-\beta_1^t)$. In the early stages of training (when t is small), this factor is significantly less than 1, meaning m_t is biased towards its initialization value of zero. The same logic applies to the second moment estimate v_t.

To counteract this initialization bias, we divide the moment estimates by their expected scaling factor, $(1-\beta_1^t)$ and $(1-\beta_2^t)$ respectively. As $t \to \infty$, both $\beta_1^t \to 0$ and $\beta_2^t \to 0$, making the correction factor approach 1, and the bias correction becomes negligible. This correction ensures a more accurate and robust estimate of the moments, especially at the beginning of training.

Question 3: Derive the Binary Cross-Entropy loss function from the principle of Maximum Likelihood Estimation (MLE).

Answer:

Let's consider a binary classification problem. Our data consists of pairs (x_i, y_i), where x_i is a feature vector and $y_i \in \{0, 1\}$ is the true label. Our model, parameterized by weights w, outputs the probability that the label is 1, given the input x_i: $p(y=1|x_i; w) = p_i$. Consequently, $p(y=0|x_i; w) = 1 - p_i$.

These two can be combined into a single expression for the probability of a single observation y_i, which follows a Bernoulli distribution:

$$P(y_i|x_i; w) = p_i^{y_i}(1-p_i)^{1-y_i}$$

You can verify this: if $y_i = 1$, it gives p_i. If $y_i = 0$, it gives $1 - p_i$.

Maximum Likelihood Estimation (MLE) aims to find the parameters w that maximize the likelihood of observing the entire training dataset. Assuming the training examples are independent and identically distributed (i.i.d.), the total likelihood of the dataset D is the product of the individual probabilities:

$$L(w|D) = P(D;w) = \prod_{i=1}^{N} P(y_i|x_i;w) = \prod_{i=1}^{N} p_i^{y_i}(1-p_i)^{1-y_i}$$

Maximizing this likelihood is equivalent to maximizing its logarithm (the log-likelihood), which is mathematically more convenient because it turns the product into a sum:

$$\mathcal{LL}(w) = \log L(w|D) = \log\left(\prod_{i=1}^{N} p_i^{y_i}(1-p_i)^{1-y_i}\right)$$

$$\mathcal{LL}(w) = \sum_{i=1}^{N} \log\left(p_i^{y_i}(1-p_i)^{1-y_i}\right)$$

$$\mathcal{LL}(w) = \sum_{i=1}^{N} [y_i \log(p_i) + (1-y_i)\log(1-p_i)]$$

The goal of training a neural network is typically framed as *minimizing* a loss function, not maximizing a likelihood. The standard convention is to define the loss as the **negative log-likelihood** (NLL).

Therefore, the loss function for the entire dataset is:

$$\text{Loss}(w) = -\mathcal{LL}(w) = -\sum_{i=1}^{N} [y_i \log(p_i) + (1-y_i)\log(1-p_i)]$$

The average loss for a single data point is obtained by dividing by N. This gives us the familiar **Binary Cross-Entropy (BCE)** loss function:

$$L_{BCE} = -[y\log(p) + (1-y)\log(1-p)]$$

Thus, minimizing the Binary Cross-Entropy loss is equivalent to maximizing the likelihood of

the training data under the assumption that the data follows a Bernoulli distribution.

Practical & Coding Questions

Question 1: Implement a Scaled Dot-Product Attention layer from scratch in PyTorch.

Answer:

Here is a full implementation of a Scaled Dot-Product Attention module in PyTorch. It is self-contained and can be integrated into a larger model like a Transformer.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class ScaledDotProductAttention(nn.Module):
    """
    Implements the Scaled Dot-Product Attention mechanism.

    Args:
        dropout (float): Dropout probability.
    """
    def __init__(self, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

    def forward(self, q, k, v, mask=None):
        """
        Performs the forward pass of the attention mechanism.

        Args:
            q (torch.Tensor): Query tensor. Shape: (batch_size, num_heads, seq_len_q, d_k
            k (torch.Tensor): Key tensor. Shape: (batch_size, num_heads, seq_len_k, d_k)
            v (torch.Tensor): Value tensor. Shape: (batch_size, num_heads, seq_len_v, d_v
                              Note: seq_len_k must be equal to seq_len_v
            mask (torch.Tensor, optional): Mask to be applied to the attention scores.
                                           Shape: (batch_size, 1, seq_len_q, seq_len_k)
                                           Defaults to None.

        Returns:
            torch.Tensor: The output of the attention mechanism. Shape: (batch_size, num_
            torch.Tensor: The attention weights. Shape: (batch_size, num_heads, seq_len_q,
        """
        # Ensure the key dimension matches for dot product
        d_k = k.size(-1)

        # 1. Compute dot product scores: (Q * K^T)
        # q: (B, H, L_q, d_k), k.transpose: (B, H, d_k, L_k) -> scores: (B, H, L_q, L_k)
        scores = torch.matmul(q, k.transpose(-2, -1))
```

```
        # 2. Scale the scores
        scores = scores / math.sqrt(d_k)

        # 3. Apply mask (if provided)
        # The mask is used to prevent attention to certain positions, e.g., padding tokens
        # or future tokens in a decoder. It sets these positions to a very small number.
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # 4. Apply softmax to get attention weights
        # Softmax is applied on the last dimension (seq_len_k) to get weights that sum to
        attention_weights = F.softmax(scores, dim=-1)

        # Apply dropout to the attention weights
        attention_weights = self.dropout(attention_weights)

        # 5. Compute the weighted sum of values
        # attention_weights: (B, H, L_q, L_k), v: (B, H, L_v, d_v) -> output: (B, H, L_q,
        # Note: L_k == L_v
        output = torch.matmul(attention_weights, v)

        return output, attention_weights

# Example Usage
if __name__ == '__main__':
    batch_size = 8
    num_heads = 4
    seq_len = 10
    d_k = 16 # Dimension of keys/queries
    d_v = 32 # Dimension of values

    # Create random input tensors
    q = torch.randn(batch_size, num_heads, seq_len, d_k)
    k = torch.randn(batch_size, num_heads, seq_len, d_k)
    v = torch.randn(batch_size, num_heads, seq_len, d_v)

    # Create an attention module
    attention_layer = ScaledDotProductAttention(dropout=0.1)
```

```
# Forward pass
output, attn_weights = attention_layer(q, k, v)

# Print shapes to verify correctness
print("Query shape:", q.shape)
print("Key shape:", k.shape)
print("Value shape:", v.shape)
print("-" * 30)
print("Output shape:", output.shape)
print("Attention weights shape:", attn_weights.shape)

# Check if attention weights for one head and one sequence element sum to 1
print("\nSum of attention weights for one example:", attn_weights[0, 0, 0, :].sum().i
```

Question 2: Visualize the effect of L1 vs. L2 regularization on the weight distribution of a simple linear model.

Answer:

This code trains three simple linear regression models on a synthetic dataset: one with no regularization, one with L1 (Lasso), and one with L2 (Ridge). It then plots the learned coefficient weights for each model to visually demonstrate how L1 promotes sparsity (many weights become zero) while L2 encourages small, non-zero weights.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split

# 1. Generate a synthetic dataset
# We create a dataset where only a few features are actually informative.
X, y, coef = make_regression(n_samples=200, n_features=50, n_informative=10,
                             noise=25, coef=True, random_state=42)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 2. Define and train the models
# A high alpha value is chosen to make the regularization effect more prominent.
alpha = 10.0

# a. Standard Linear Regression (no regularization)
lr = LinearRegression()
lr.fit(X_train, y_train)

# b. Lasso Regression (L1 regularization)
lasso = Lasso(alpha=alpha)
lasso.fit(X_train, y_train)

# c. Ridge Regression (L2 regularization)
ridge = Ridge(alpha=alpha)
ridge.fit(X_train, y_train)

# 3. Visualize the results
plt.style.use('seaborn-v0_8-whitegrid')
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(14, 10), sharex=True)
fig.suptitle('Effect of L1 vs. L2 Regularization on Model Coefficients', fontsize=16)

# Plot the true coefficients used to generate the data
ax1.plot(coef, alpha=0.7, linestyle='--', color='navy', label='True Coefficients')
ax1.set_title('True Coefficients')
ax1.legend()
```

```
ax1.set_ylabel('Coefficient Value')

# Plot the learned coefficients for each model
ax2.plot(lr.coef_, alpha=0.7, linestyle='-', color='gold', label='Linear Regression (No Re
ax2.plot(lasso.coef_, alpha=0.9, linestyle='-', color='red', label=f'Lasso (L1) alpha={alp
ax2.plot(ridge.coef_, alpha=0.7, linestyle='-', color='blue', label=f'Ridge (L2) alpha={al
ax2.set_title('Learned Coefficients by Different Models')
ax2.set_xlabel('Coefficient Index')
ax2.set_ylabel('Coefficient Value')
ax2.legend()

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

# Print statistics
print("--- Sparsity Analysis ---")
print(f"Lasso (L1) found {np.sum(lasso.coef_ != 0)} non-zero coefficients out of {len(lass
print(f"Ridge (L2) found {np.sum(ridge.coef_ != 0)} non-zero coefficients out of {len(ridg
print(f"Linear Regression found {np.sum(lr.coef_ != 0)} non-zero coefficients out of {len
```

Interpretation of the Plot:

- **True Coefficients:** The top plot shows the ground truth. Only 10 features have non-zero coefficients.
- **Linear Regression:** The yellow line shows that the unregularized model has large, noisy coefficients. It tries to use all features.
- **Lasso (L1):** The red line is sparse. Many coefficients have been forced to exactly zero, effectively performing feature selection and identifying only the most important features.
- **Ridge (L2):** The blue line shows that all coefficients are shrunk towards zero but very few (if any) are exactly zero. The model retains all features but with smaller weights, resulting in a dense but constrained solution.

Question 3: Compare the convergence behavior of SGD, SGD with Momentum, and Adam on a simple neural network.

Answer:

This code defines a simple Multi-Layer Perceptron (MLP), creates a synthetic classification dataset, and then trains the network using three different optimizers: SGD, SGD with Momentum, and Adam. It plots the training loss curves for each optimizer to visually compare their convergence speed and stability.

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import numpy as np

# 1. Prepare Data
X, y = make_moons(n_samples=1000, noise=0.3, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convert to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
y_train_tensor = torch.FloatTensor(y_train).view(-1, 1)
X_test_tensor = torch.FloatTensor(X_test)
y_test_tensor = torch.FloatTensor(y_test).view(-1, 1)

# 2. Define the Model
class SimpleMLP(nn.Module):
    def __init__(self):
        super(SimpleMLP, self).__init__()
        self.layer1 = nn.Linear(2, 32)
        self.layer2 = nn.Linear(32, 16)
        self.layer3 = nn.Linear(16, 1)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = torch.sigmoid(self.layer3(x)) # Use sigmoid for binary classification
        return x

# 3. Training Function
```

```
def train_model(optimizer_name, learning_rate):
    model = SimpleMLP()
    if optimizer_name == 'sgd':
        optimizer = optim.SGD(model.parameters(), lr=learning_rate)
    elif optimizer_name == 'sgd_momentum':
        optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)
    elif optimizer_name == 'adam':
        optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    criterion = nn.BCELoss() # Binary Cross-Entropy Loss

    epochs = 100
    loss_history = []

    for epoch in range(epochs):
        model.train()

        # Forward pass
        outputs = model(X_train_tensor)
        loss = criterion(outputs, y_train_tensor)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        loss_history.append(loss.item())

    return loss_history

# 4. Run Experiments and Plot Results
learning_rate = 0.01

print("Training with SGD...")
loss_sgd = train_model('sgd', learning_rate)

print("Training with SGD + Momentum...")
loss_sgd_momentum = train_model('sgd_momentum', learning_rate)
```

```
print("Training with Adam...")
loss_adam = train_model('adam', learning_rate)

plt.figure(figsize=(12, 7))
plt.plot(loss_sgd, label='SGD', color='orange', linestyle='--')
plt.plot(loss_sgd_momentum, label='SGD with Momentum', color='blue', linestyle='-.')
plt.plot(loss_adam, label='Adam', color='green', linestyle='-')

plt.title(f'Optimizer Comparison (LR={learning_rate})', fontsize=16)
plt.xlabel('Epoch')
plt.ylabel('Training Loss (BCE)')
plt.legend()
plt.grid(True)
plt.ylim(0, 0.7) # Set y-limit to better see the convergence differences
plt.show()
```

Interpretation of the Plot:

- **SGD (Orange, Dashed):** Typically converges the slowest. The loss decreases, but it might be erratic and take many epochs to reach a low value.
- **SGD with Momentum (Blue, Dash-Dot):** Converges faster and more smoothly than plain SGD. The momentum helps it overcome local bumps and accelerate in the correct direction.
- **Adam (Green, Solid):** Usually converges the fastest and most reliably. Its adaptive learning rate mechanism adjusts for each parameter, allowing it to quickly find a good path down the loss landscape. In most common scenarios, Adam will show the steepest initial drop in loss.