Deep Learning HW 1

Problem 1

Part 1

For a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient ∇f is a vector in $\mathbb{R}^n$ whose components are the partial derivatives of f with respect to each variable, that is:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right]^T$$

The Hessian matrix H of the function f is then the square matrix of all second-order mixed partial derivatives:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

If f is twice continuously differentiable, the order of differentiation does not matter (Schwarz's theorem), and thus the Hessian matrix is symmetric.

Now, if we consider ∇f as a vector-valued function from $\mathbb{R}^n$ to $\mathbb{R}^n$, its Jacobian J would be a matrix where each row is the gradient of each component of ∇f. Since ∇f is already the gradient of f, its Jacobian would consist of the second derivatives of f with respect to each variable, which is precisely the Hessian matrix:

$$J(\nabla f) = H$$

This means that the Hessian is effectively the Jacobian of the gradient vector of f. It describes the local curvature of f in different directions in the space.

Part 2

Proof A: Derivative of the Dot Product

Consider the dot product of two vectors x and a in $\mathbb{R}^n$:

$$f(x) = x^T a$$

This function can be expanded as:

$$f(x) = \sum_{i=1}^{n} x_i a_i$$

Taking the derivative of f with respect to the vector x gives us a vector where each component is the partial derivative of f with respect to x_i:

$$\frac{\partial f}{\partial x} = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right]^T$$

Since a is constant, the derivative of each term $x_i a_i$ with respect to x_i is simply a_i. Therefore, the gradient of f is:

$$\nabla f(x) = a$$

Proof B: Derivative of the Trace of a Matrix Product

Let X be a matrix and A a constant matrix. Then the function is:

$$f(X) = \text{tr}(AX)$$

The trace of a matrix product has the property that $\text{tr}(AB) = \text{tr}(BA)$. The derivative of the trace of a product with respect to the matrix X is the transpose of the other matrix, hence:

$$\frac{\partial \text{tr}(AX)}{\partial X} = A^T$$

Proof C: Derivative of the Trace of a Matrix Quadratic Form

For the quadratic form $X^T AX$, where X is a matrix and A is a symmetric constant matrix, the function $f(X)$ is:

$$f(X) = \text{tr}(X^T AX)$$

By applying the properties of the trace and derivative, we have:

$$\frac{\partial \text{tr}(X^T AX)}{\partial X} = \frac{\partial \text{tr}(AXX^T)}{\partial X}$$

Since A is symmetric, the derivative of this expression with respect to X involves the sum of derivatives with respect to each element of X, which gives:

$$\frac{\partial \text{tr}(X^T A X)}{\partial X} = (A + A^T)X$$

Given that$A$is symmetric,$A = A^T$, simplifying to:

$$\frac{\partial \text{tr}(X^T A X)}{\partial X} = 2AX$$

Proof D: Derivative of the Log Determinant of$X$

For a square matrix$X$, the function$f(X)$is:

$$f(X) = \log(\det(X))$$

Using the property that$\det(X)$is the product of its eigenvaluesλ_i, we can express$f(X)$as:

$$f(X) = \log\left(\prod_{i=1}^{n} \lambda_i\right)$$

Taking the derivative of both sides with respect to$X$and applying the chain rule, we get:

$$\frac{\partial f(X)}{\partial X} = \frac{\partial}{\partial X}\left(\sum_{i=1}^{n} \log(\lambda_i)\right)$$

Since the derivative of the logarithm is$1/x$, and using Jacobi's formula for the derivative of a determinant, we obtain:

$$\frac{\partial f(X)}{\partial X} = \frac{1}{\det(X)} \cdot \frac{\partial \det(X)}{\partial X} = \frac{1}{\det(X)} \cdot \det(X) \cdot (X^{-1})^T$$

Which simplifies to:

$$\frac{\partial f(X)}{\partial X} = (X^{-1})^T$$

Part 3

Eigenvalues and Eigenvectors of Rotation Matrix

Given the rotation matrix$R(\theta)$:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

To find the eigenvalues, we solve the characteristic equation$\det(R(\theta) - \lambda I) = 0$:

$$\begin{vmatrix} \cos(\theta) - \lambda & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) - \lambda \end{vmatrix} = 0$$

$$(\cos(\theta) - \lambda)^2 + \sin^2(\theta) = 0$$

$$\lambda^2 - 2\lambda\cos(\theta) + 1 = 0$$

Solving this quadratic equation gives us the eigenvalues:

$$\lambda_{1,2} = \cos(\theta) \pm i\sin(\theta)$$

The corresponding eigenvectors$v$are found by solving$(R(\theta) - \lambda I)v = 0$. Since the eigenvalues are complex, the eigenvectors will also be complex. For example, forλ_1, we have:

$$v_1 = \begin{bmatrix} i \\ 1 \end{bmatrix}$$

Forλ_2, the eigenvector$v_2$will be the complex conjugate ofv_1.

Determinant and Eigenvalues

The determinant of a matrix$X$is the product of its eigenvalues:

$$\det(X) = \prod_{i=1}^{n} \lambda_i$$

For the rotation matrix$R(\theta)$, the eigenvalues are$e^{i\theta}$and$e^{-i\theta}$, so the determinant is:

$$\det(R(\theta)) = e^{i\theta} \cdot e^{-i\theta} = 1$$

Diagonalization and Powers of a Matrix

The diagonalization of a matrix$X$is$X = PDP^{-1}$, where$D$is a diagonal matrix of eigenvalues and$P$is a matrix of corresponding eigenvectors. For the rotation matrix, this is not possible in real numbers because the eigenvalues are complex. However, if$R(\theta)$were diagonalizable over the complex numbers, then:

$$R^n(\theta) = (PDP^{-1})^n = PD^nP^{-1}$$

Since$D$is diagonal with$e^{i\theta}$and$e^{-i\theta}$on the diagonal,D^nwould have$e^{in\theta}$and$e^{-in\theta}$, and thus:

$$R^n(\theta) = \begin{bmatrix} \cos(n\theta) & -\sin(n\theta) \\ \sin(n\theta) & \cos(n\theta) \end{bmatrix}$$

Which is equivalent to$R(n\theta)$, showing that$R(n\theta) = R^n(\theta)$.

Problem 2

Problem 2: Optimization

Question 1: Saddle Points vs. Local Minima in Higher Dimensions In higher-dimensional spaces, saddle points are more common than local minima due to the nature of high-dimensional spaces and the behavior of multivariate functions. Specifically:

- **Curvature:** In high-dimensional spaces, for a point to be a local minimum, all eigenvalues of the Hessian matrix (which captures the second derivatives or the curvature of the function) at that point must be positive. The probability of this happening decreases as the dimensionality increases since there are more eigenvalues that all need to be positive.
- **Saddle Points:** Conversely, a saddle point is characterized by the Hessian having both positive and negative eigenvalues. There are many more configurations of eigenvalues that result in saddle points than configurations that result in local minima.
- **Intuition:** You can think of a saddle point as a point where the function curves upwards in some directions and downwards in others. In higher dimensions, there are more directions in which the curvature can vary, making saddle points more prevalent.

Question 2: Optimization Path Visualization and Comparison Here's a theoretical visualization and comparison of the optimization paths for the mentioned methods:

- **Gradient Descent (GD):** This method would likely show a direct path towards the minimum, possibly with some overshooting if the learning rate is too high.
- **Momentum:** This adds inertia to the optimization, potentially leading to faster convergence but also possible overshooting.
- **Nesterov Momentum:** Similar to Momentum but with a lookahead feature, the path might show corrective curves that anticipate the future gradient, leading to a smoother and potentially faster convergence.
- **RMSprop:** This adapts the learning rate based on recent gradients, likely showing a more refined path with fewer oscillations and a steady approach to the minimum.

Each of these methods aims to address specific shortcomings of basic Gradient Descent:

- **Advantages and Disadvantages:** GD can be slow and susceptible to local minima. Momentum accelerates convergence but can overshoot. Nesterov Momentum provides a more refined approach with lookahead corrections. RMSprop adapts the learning rate to avoid oscillations and speed up convergence in steep areas.

Question 3: ADAM Optimization and Bias Correction ADAM combines the advantages of Momentum (acceleration) and RMSprop (adaptive learning rate) but also includes a bias correction mechanism:

- **Momentum Problem:** Momentum can accumulate a velocity that is too high, leading to overshooting. It also doesn't adjust the learning rate based on the gradient's magnitude.
- **Bias Correction:** When initializing moments at zero, ADAM's estimates of the first and second moments are biased towards zero, especially during the initial time steps. Bias correction helps to adjust the estimates to be more accurate early in training.

Bias correction compensates for these initial low moment estimates, ensuring that the adaptive learning rates are neither too high nor too low at the start of training.

Problem 3

Problem 3: Regularization

1. Dropout can be seen as a way of doing an equally-weighted averaging of exponentially many models with shared weights. This is similar to the idea behind ensemble methods, which combine multiple models to improve performance. However, in networks with a large number of parameters, the number of possible models becomes so large that it is impractical to train

and combine them all. Dropout provides a computationally efficient way to achieve a similar effect by randomly dropping out units during training, effectively creating many different thinned networks that share weights. This allows dropout to outperform ensemble methods in large networks, where the number of possible models is too large to exhaustively explore.

2. Dropout acts like a regularization technique because it encourages the network to learn more robust features that are useful for multiple different paths through the network. By randomly dropping out units during training, dropout prevents any single unit from relying too heavily on the presence of other units, forcing each unit to learn more useful and independent features. This reduces the risk of overfitting to the training data and improves the generalization performance of the network.

3. The difference between using dropout during training and testing is that during training, dropout randomly drops out units with a certain probability, while during testing, all units are present but their outputs are scaled down by the same probability. The reason for this difference is that during training, dropout is used to prevent overfitting by forcing the network to learn more robust features. By randomly dropping out units, the network is forced to learn features that are useful for multiple different paths through the network, reducing the risk of overfitting. During testing, however, we want to use the full power of the network to make predictions, so we need to scale down the outputs of the units to account for the fact that some of them were dropped out during training.

Question 1: Dropout and Ensemble Learning Dropout is a regularization technique that randomly omits a subset of features or units at each iteration of training. This is similar to ensemble learning because:

- **Model Averaging:** In ensemble learning, predictions from multiple models are averaged. Dropout effectively creates a "thinned" network at each iteration, essentially sampling from an "ensemble" of different network architectures.
- **Reduction of Co-adaptation:** By dropping out different sets of neurons, it ensures that neurons do not co-adapt too strongly to the presence of other neurons, much like how ensemble methods combine models that are not too correlated to each other.
- **Performance:** Dropout tends to have better performance in networks with a large number of parameters because it significantly increases the number of distinct network architectures that can be sampled, which is akin to having a larger and more diverse ensemble of models.

Question 2: Dropout as a Regularization Technique Dropout acts as a regularization technique because:

- **Preventing Overfitting:** By randomly dropping units during training,

dropout prevents complex co-adaptations on the training data, which is similar to the effect of regularizing terms that penalize complex models.
- **Equivalent to Adding Noise:** Dropout can be seen as a method of adding noise to the inputs of each layer, which is known to regularize the learning.

Question 3: Dropout in Training vs. Testing Dropout is used during training but not during testing for the following reasons:

- **Training Phase:** During training, dropout randomly omits units to prevent overfitting by ensuring that the network's predictions are not overly reliant on any single neuron.
- **Testing Phase:** At test time, dropout is not used (or it is adjusted), and a single unthinned network is used to make predictions. This network has smaller weights that are scaled down by the dropout probability, approximating the average of the predictions of the thinned networks seen during training.

Question 4: Dropout in Linear Regression as Regularization The use of dropout in linear regression can be shown to be equivalent to adding a regularization term in the loss function through mathematical proof. This involves showing that the expected value of the output, with dropout applied, is equivalent to the output obtained by adding a regularization term to the loss function.

The proof would involve taking the expectation of the loss function with dropout and showing that it includes a term that penalizes the complexity of the model, similar to L1 or L2 regularization terms.

Question 5: Batch-Normalization and Learning Rate Control Batch-normalization allows for easier control of learning rate due to several factors:

- **Normalization Effect:** By normalizing the input distribution of each layer, batch normalization reduces the internal covariate shift which allows for higher learning rates without the risk of divergence.
- **Stabilization:** It stabilizes the learning process by maintaining the mean and variance of inputs within a certain range, which can prevent the gradients from becoming too small (vanishing) or too large (exploding).

Question 6: Batch-Normalization as Regularization Batch-normalization also has a regularization effect:

- **Noise Injection:** The process of normalizing based on batch statistics introduces noise into the layer's outputs, which can have a regularizing effect.

- **Dependency Reduction:** It reduces the dependence of gradients on the scale of parameters or their initial values, which can help in generalizing better.

The effect of larger batch sizes on batch-normalization:

- **Reduced Noise:** Larger batch sizes reduce the amount of noise introduced by batch normalization, which might decrease its regularization effect.
- **Stable Estimates:** However, larger batches provide more stable estimates of the mean and variance used for normalization, which could lead to better training stability.

Problem 4

Problem 4: Activation Functions

Question 1: Proper Activation Function for Classification Problems For each classification problem, the proper activation function for the output layer and the reason behind the choice are as follows:

A. **Binary Classification (Dog or Cat):**

- **Activation Function:** Sigmoid
- **Reason:** The sigmoid function outputs a probability value between 0 and 1, which is ideal for binary classification tasks.

B. **Multiclass Classification (100 Animals):**

- **Activation Function:** Softmax
- **Reason:** The softmax function extends the sigmoid to multiple classes. It outputs a probability distribution over the classes, ensuring that the sum of probabilities is 1.

C. **Multi-label Classification (Multiple Animals in One Image):**

- **Activation Function:** Sigmoid
- **Reason:** For multi-label classification, each class is treated independently with a binary classification task, hence the sigmoid is appropriate for each output neuron.

Question 2: DPReLU Activation Function ReLU (Rectified Linear Unit) has been widely used due to its simplicity and effectiveness in addressing the vanishing gradient problem. However, it has limitations, such as the dying ReLU problem, where neurons can become inactive and only output zero. DPReLU (Dynamic Parametric ReLU) is introduced to address this issue.

The DPReLU function dynamically adjusts its shape during training with four learnable parameters, making it flexible to adapt to different datasets and models. The paper "DPReLU: Dynamic Parametric Rectified Linear Unit and Its Proper Weight Initialization Method" details the following aspects:

- **Learnable Parameters:** DPReLU's parameters are learned during training, unlike standard ReLU, which has a fixed shape.
- **Overcoming Dying ReLU:** By allowing for a non-zero gradient when the input is negative, it mitigates the risk of neurons dying.
- **Weight Initialization:** Proper weight initialization is crucial for DPReLU to perform effectively. The paper proposes a robust method to initialize these weights.

DPReLU provides a solution to the shortcomings of previous ReLU variants by offering a more flexible and adaptive form of the activation function, which leads to better performance in terms of convergence and accuracy.

Problem 5

Problem 5: Neural Networks and Backpropagation

Question 1: Calculate Backpropagation for One Step For the given two-layered neural network, we are tasked with performing one step of backpropagation. Here's the architecture for reference:

- Input nodes:x_1, x_2
- Hidden nodes:h_1, h_2
- Output node:y
- Weights:$w_1, w_2, w_3, w_4$for the input to hidden layer,$w_5, w_6$for the hidden to output layer
- Biases:b_1, b_2
- Activations: Leaky ReLU (LReLU) for the hidden layer and sigmoid for the output

Given values:

- Inputs:$[x_1, x_2] = [0, 1]$
- Target output:$[y] = [1]$
- Weights:$[w_1, w_2, w_3, w_4] = [0.3, 0.2, 0.2, -0.6]$,$[w_5, w_6] = [0.5, -1]$
- Biases:$[b1, b2] = [0.2, -1.4]$
- LReLU:$LreLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.2x, & \text{if } x < 0 \end{cases}$
- Loss Function (MSE):$L = \frac{1}{2}(\hat{y} - y)^2$
- Learning rate:$\alpha = 0.1$

The backpropagation process involves the following steps:

1. **Forward Pass:** Compute the output$\hat{y}$for the given inputs using the current weights and biases.
2. **Compute Error:** Calculate the error in the output using the loss function.
3. **Backward Pass:** Compute the gradient of the loss with respect to each weight and bias.
4. **Update Weights and Biases:** Adjust the weights and biases in the direction that minimally reduces the loss, using the calculated gradients

and learning rate.

After performing one step of backpropagation, here are the updated weights, biases, and the loss:

- Updated Weights:$[w_1, w_2, w_3, w_4, w_5, w_6] = [0.3, 0.20405341, 0.2, -0.60162137, 0.50324273, -1.00324273]$
- Updated Biases:$[b_1, b_2] = [0.20405341, -1.40162137]$
- Loss: 0.06277973

The weights and biases have been adjusted according to the gradients computed from the backpropagation algorithm. The loss represents the mean squared error between the predicted output and the target after one forward pass through the network.

Question 2: Batch Normalization Backpropagation

- Describe the computational graph.
- Write the relation between mean and variance.
- Calculate the partial derivatives of the loss function with respect toβ,γ,x_1, andx_2.

Given a mini-batch$B$of size$m$, the batch normalization process is:

1. **Calculate Mean:**$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$
2. **Calculate Variance:**$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
3. **Normalize:**$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$for numerical stability,$\epsilon$is a small constant.
4. **Scale and Shift:**$y_i = \gamma\hat{x}_i + \beta$where$\gamma$and$\beta$are learnable parameters.

For the computational graph, the loss$L$is a function ofy_i, and we need to compute$\frac{\partial L}{\partial \beta}$,$\frac{\partial L}{\partial \gamma}$,$\frac{\partial L}{\partial x_1}$, and$\frac{\partial L}{\partial x_2}$.

The backpropagation through batch normalization yields the following partial derivatives:

- Derivative of the loss with respect to the inputs$x$:$[dL/dx_1, dL/dx_2] = [0.0, 0.0]$
- Derivative of the loss with respect to the scale parameterγ:$dL/d\gamma = 0.0$
- Derivative of the loss with respect to the shift parameterβ:$dL/d\beta = 2.0$

The derivativesdL/dxbeing zero can be a result of the simplifications made for this demonstration, particularly the assumption that the loss is the sum of the outputs, leading to a gradient of 1 for each output. This isn't typically the case in a real neural network where the loss function is more complex. However, the calculations here are correct based on the assumptions made.

The derivative$dL/d\beta$being 2.0 indicates that increasing$\beta$by a small amount would increase the loss, given our simple loss function and the current values of$x$,γ, andβ.

These calculations illustrate how the gradients for batch normalization are computed during backpropagation, which can then be used to updateγandβduring training.