EAS 6995, SPRING 2025

HOMEWORK 1

(1) There are many possible loss functions for training models; different loss functions are suited for different tasks. In class we have mentioned Mean Squared Error loss, Binary Cross-Entropy loss, and Categorical Cross-Entropy (CCE) loss (softmax). List common use cases for each of these:

- Mean Squared Error (MSE)
- Binary Cross-Entropy
- Categorical Cross-Entropy

What is another loss functions besides those? Provide the reference (e.g., Bishop and Bishop, etc.). Provide a use case for that loss function, and describe why it may be preferable to the others listed above for that case.

(2) Expectation and Variance of Gradient Estimates:

Let $f(w, X)$ be a loss function for parameter w, and let $\nabla f(w, X)$ denote the gradient of the loss function with respect to w for a single data point X.

The **true gradient** over the full dataset is the expectation:

$$g_{\text{full}} = \mathbb{E}_{X \sim P}[\nabla f(w, X)].$$

However, when training with **mini-batch gradient descent**, we estimate this expectation using a batch of B i.i.d. samples $X_1, X_2, \ldots, X_B$:

$$\hat{g}_B = \frac{1}{B} \sum_{i=1}^{B} \nabla f(w, X_i).$$

Since each sampled X_i produces a different gradient, the variance of individual gradients over the data distribution is given by:

$$\text{Var}[\nabla f(w, X)] = \mathbb{E}_{X \sim P}[\|\nabla f(w, X)\|^2] - \|\mathbb{E}_{X \sim P}[\nabla f(w, X)]\|^2.$$

The **variance of the mini-batch gradient estimate** is then given by:

$$\text{Var}[\hat{g}_B] = \frac{1}{B} \text{Var}[\nabla f(w, X)].$$

PART 1: UNDERSTANDING VARIANCE IN GRADIENT ESTIMATES

(a) Suppose you have three cases:
  (a) **Case 1:** Using the entire dataset (i.e., full-batch gradient descent).
  (b) **Case 2:** Using a large mini-batch (B is large but smaller than the dataset).
  (c) **Case 3:** Using a small mini-batch (B is small).

  Using the variance equation above, compare the variance in each case and explain why small mini-batches lead to noisier gradient estimates.

PART 2: IMPACT ON OPTIMIZATION

(b) How does the variance of $\hat{g}_B$ affect the stability of gradient descent updates? Explain in the context of:
  – Convergence speed
  – Risk of overshooting the minimum
  – Generalization to unseen data

(c) Practical deep learning models often use batch sizes between 32 and 512. Why don't we typically use very large batches (e.g., $100,000+$ samples)? Discuss how batch size affects both training time and generalization.

(3) We are going to write a neural network to classify FashionMNIST (or MNIST) images (10 classes of clothing or digits). You can **only use basic Python, numpy, and matplotlib**. One exception: You can use pytorch to download the data and to load it into the training loop.

(a) **Download the FashionMNIST (or MNIST) dataset:**

```
train_dataset = datasets.FashionMNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.FashionMNIST(root='./data', train=False, download=True, transform=transform)
```

(b) **Plot 4 random images from FashionMNIST or MNIST, and their labels.** Are the labels correct? (Some human labeled them, who may have been having a bad day.)

(c) **Code a neural network to classify these images:**

- In the MNIST dataset, there are classes from 0–9. Which loss function might you choose?
- The training dataset that you downloaded will have 60,000 images, and the validation (or test) dataset has 10,000. Keep say **50 percent of each class** for both training and validation datasets. Write code to subsample these. Remember to also subsample the corresponding labels.
- Each image is 28x28, for a total of 784 pixels. Each of these pixels has one value/feature. **Normalize the data** to get all pixels in the range [0, 1].
- Plot 4 images after the subsampling and normalization to make sure that labels are still correct and images are as expected.

(d) **Create a FashionMNIST or MNIST dataset using PyTorch:**

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

These are now batched with a batch size of 64 (you can try changing that after you have the code running). To load one batch, you can use the following code:

```
data_iter = iter(train_loader)
images, labels = data_iter.next()
```

(e) **Write a fully-connected layer**

First, compute the linear transformation:

$$z^{l+1} = Wa^l + b$$

where:

- $z^{l+1} \in \mathbb{R}^m$ is the output before the activation function.
- $a^l \in \mathbb{R}^d$ is the input to the layer (activations from layer l).
- $W \in \mathbb{R}^{m \times d}$ is the weight matrix, where:
  - d is the number of input features.
  - m is the number of output features (neurons in layer $l + 1$).
- $b \in \mathbb{R}^m$ is the bias vector.

- $z^{l+1} \in \mathbb{R}^m$ is the output of the linear transformation, which is then typically passed through an activation function.

(f) Then, apply the activation function (e.g., ReLU):

$$a^{l+1} = \text{ReLU}(z^{l+1}) = \max(0, z^{l+1})$$

where a^{l+1} is the output of the layer after the activation.
Use matrix multiplication for this with '@' for speed.

(g) **This class also needs to implement the backward pass (see the example at the end of this document, and see Ch. 8 of Bishop, Deep Learning):**

- **Gradient of Loss w.r.t. z^{l+1}:** The gradient with respect to z^{l+1} is computed by applying the chain rule:

$$\frac{\partial L}{\partial z^{l+1}} = \frac{\partial L}{\partial a^{l+1}} \cdot \frac{\partial a^{l+1}}{\partial z^{l+1}}$$

  For ReLU, the derivative is:

$$\frac{\partial a^{l+1}}{\partial z^{l+1}} = \begin{cases} 1 & \text{if } z^{l+1} > 0 \\ 0 & \text{if } z^{l+1} \leq 0 \end{cases}$$

  Thus, the gradient with respect to z^{l+1} will be:

$$\frac{\partial L}{\partial z^{l+1}} = \frac{\partial L}{\partial a^{l+1}} \cdot \mathbf{1}_{z^{l+1}>0}$$

  where $\mathbf{1}_{z^{l+1}>0}$ is an indicator function that is 1 if $z^{l+1} > 0$, and 0 if $z^{l+1} \leq 0$.

  - **Hint:** Start by computing $\frac{\partial L}{\partial a^{l+1}}$, where L is your loss function. This step will depend on the specific loss function you are using.

- **Gradient of Loss w.r.t Weights W^{l+1}:** After computing the gradient of the loss with respect to the output, you'll propagate the error backward through the network to the weights W^{l+1}. The gradient of the loss with respect to the weights is obtained by applying the chain rule to the linear transformation.

$$\frac{\partial L}{\partial W^{l+1}} = \frac{\partial L}{\partial z^{l+1}} \cdot \frac{\partial z^{l+1}}{\partial W^{l+1}}$$

  Using the chain rule:

$$\frac{\partial L}{\partial W^{l+1}} = \frac{\partial L}{\partial z^{l+1}} \cdot (a^l)^T$$

  where:

  - $\frac{\partial L}{\partial z^{l+1}}$ is the gradient of the loss with respect to z^{l+1}, the output of the linear transformation,
  - $(a^l)^T$ is the transpose of the activations from the previous layer a^l.

- **Gradient of Loss w.r.t Bias b^{l+1}:**
  We begin by applying the chain rule:
  $$\frac{\partial L}{\partial b^{l+1}} = \frac{\partial L}{\partial a^{l+1}} \cdot \frac{\partial a^{l+1}}{\partial z^{l+1}} \cdot \frac{\partial z^{l+1}}{\partial b^{l+1}}$$
  Where:
  - $\frac{\partial L}{\partial a^{l+1}}$ is the gradient of the loss with respect to the activations a^{l+1},
  - $\frac{\partial a^{l+1}}{\partial z^{l+1}} = \sigma'(z^{l+1})$, the derivative of the activation function σ, computed above
  - $\frac{\partial z^{l+1}}{\partial b^{l+1}} = 1$

  Thus, we have:
  $$\frac{\partial L}{\partial b^{l+1}} = \frac{\partial L}{\partial a^{l+1}} \cdot \sigma'(z^{l+1})$$
  Since the bias term affects all neurons equally, we sum over the batch (if using mini-batches):
  $$\frac{\partial L}{\partial b^{l+1}} = \sum \frac{\partial L}{\partial z^{l+1}}$$

- **Recursion of the Gradient for Deeper Layers:** To compute the gradients for deeper layers, you use the chain rule recursively. For example, for $\frac{\partial L}{\partial a^l}$, we have:
  $$\frac{\partial L}{\partial a^l} = \frac{\partial L}{\partial z^{l+1}} \cdot \frac{\partial z^{l+1}}{\partial a^l}$$
  This involves the weights W^l of the previous layer and the gradient $\frac{\partial L}{\partial z^{l+1}}$.

  - **Hint:** The term $\frac{\partial z^{l+1}}{\partial a^l}$ is simply the weights W^{l+1} of the next layer. So, this step involves propagating the error from the current layer back to the previous layer:
    $$\frac{\partial L}{\partial a^l} = (W^{(l+1)})^T \cdot \frac{\partial L}{\partial z^{l+1}}$$
    Once you've computed this, you can recursively apply the same steps for earlier layers.

(h) **Implement a ReLU unit where $a^{l+1} = \mathbf{max}(0, a^l)$, performed element-wise. This needs both a forward function and a backward function.**

(i) **Implement the softmax function for probabilities and cross-entropy loss, both forward and backward.**

- **Softmax Equation:**

$$a_k^{l+1} = \frac{e^{a_k^l}}{\sum_{k'=1}^{C} e^{a_{k'}^l}}$$

where C is the number of classes.

- **Cross-Entropy Loss (CCE):**

$$\mathcal{L} = -\frac{1}{B}\sum_{i=1}^{B}\sum_{k=1}^{C} \mathbf{1}_{y_i=k} \log\Big(a_k^{l+1}(\mathbf{x}_i)\Big)$$

where:
  - $a_k^{l+1}(\mathbf{x}_i)$ is the softmax probability for class k given input $\mathbf{x}_i$,
  - $\mathbf{1}_{y_i=k}$ is an indicator function, equal to 1 if the label for input $\mathbf{x}_i$ is k, otherwise 0,
  - B is the batch size.

- **Error Calculation:**

$$\text{error} = \frac{1}{B}\sum_{i=1}^{B} \mathbf{1}_{y_i \neq_k a_k^{l+1}}$$

This computes the **average number of misclassifications** in the batch, which is used for monitoring training progress.

(j) **Test your forward and backward operators.** The forward operator is simple to test with simple inputs. The backward operator can be tested using **numerical differentiation**:

- **Gradient Checking for Backpropagation:** To test if backpropagation is working correctly in a neural network coded from scratch, we can use **gradient checking**. This method compares the gradients computed by backpropagation with those computed using **numerical differentiation**.
- **Numerical Gradient Calculation:** Compute the numerical gradients using finite differences. For a parameter θ, the numerical gradient is calculated as:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta + \epsilon) - L(\theta - \epsilon)}{2\epsilon}$$

where $L(\theta)$ is the loss function and ϵ is a small value, typically 10^{-4}.

- **Compare Gradients:** Compare the gradients obtained from backpropagation with the numerical gradients. The difference should be very small (typically less than 10^{-5}) if the implementation is correct.

(k) **Train your neural network.** Tune the **hyperparameters** of the number of iterations and learning rate.

(l) **Test your model by passing the images from the validation dataset through the forward pass every 1000 iterations.** Compute the loss and error for these iterations.

(m) **Plot training loss and training error as a function of the number of weight updates of the neural network you trained. Describe what you see.**

(n) **Plot the test loss and test error for every 1000th weight update.** How does this compare to training loss and error? Is it what you would expect, and why or why not?

(o) **Plot four of the images that were labeled correctly, and four that were labeled incorrectly.** Do you have any observations?

EXAMPLE: TWO-LAYER NEURAL NETWORK

Consider a simple two-layer neural network with the following forward pass equations:

$$z^1 = W^0 x + b^0 \qquad \text{(First layer: linear transformation)} \tag{1}$$
$$a^1 = \max(0, z^1) \qquad \text{(Activation function: ReLU)} \tag{2}$$
$$z^2 = W^1 a^1 + b^1 \qquad \text{(Second layer: linear transformation)} \tag{3}$$
$$L = \frac{1}{2}(z^2 - y)^2 \qquad \text{(Loss function: Mean Squared Error)} \tag{4}$$

Where:

- x is the input,
- W^0, b^0 are the weights and bias for the first layer,
- W^1, b^1 are the weights and bias for the second layer,
- y is the target output.

STEP-BY-STEP BACKPROPAGATION

Step 1: Compute $\frac{\partial L}{\partial h^2}$. Starting with the loss function:

$$L = \frac{1}{2}(z^2 - y)^2$$

Taking the derivative with respect to z^2:

$$\frac{\partial L}{\partial z^2} = (z^2 - y)$$

Step 2: Compute $\frac{\partial L}{\partial a^1}$. Since $z^2 = W^1 a^1 + b^1$, the gradient of z^2 with respect to a^1 is W^1, so:

$$\frac{\partial L}{\partial a^1} = (W^1)^T \frac{\partial L}{\partial z^2}$$

Step 3: Compute $\frac{\partial L}{\partial z^1}$. Since $a^1 = \max(0, z^1)$ (ReLU), the derivative of ReLU is:

$$\frac{\partial a^1}{\partial z^1} = \begin{cases} 1 & \text{if } z^1 > 0 \\ 0 & \text{if } z^1 \leq 0 \end{cases}$$

This means that for positive activations ($z^1 > 0$), the derivative of ReLU is 1, and for negative activations ($z^1 \leq 0$), the derivative is 0. Neurons with an activation of 0 do not contribute in this forward pass.

Applying the chain rule:

$$\frac{\partial L}{\partial z^1} = \frac{\partial L}{\partial a^1} \cdot \frac{\partial a^1}{\partial z^1}$$

Step 4: Compute $\frac{\partial L}{\partial W^1}$. To compute the gradient of the loss with respect to the weights W^1, we use the chain rule:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial z^2} \cdot (a^1)^T$$

Step 5: Compute $\frac{\partial L}{\partial b^1}$. The gradient with respect to the bias b^1 is computed using the chain rule. Since $z^2 = W^1 a^1 + b^1$, we have:

$$\frac{\partial L}{\partial b^1} = \frac{\partial L}{\partial z^2} \cdot \frac{\partial z^2}{\partial b^1}$$

Because $\frac{\partial z^2}{\partial b^1} = 1$ (since b^1 contributes a constant shift), we get:

$$\frac{\partial L}{\partial b^1} = (z^2 - y) \cdot 1 = z^2 - y$$

Thus, the gradient with respect to b^1 is the same as the gradient with respect to z^2.

Step 6: Compute $\frac{\partial L}{\partial W^0}$ and $\frac{\partial L}{\partial b^0}$. To compute the gradients for W^0 and b^0, we need to propagate the gradients backward through the network.

First, we compute the gradient of the loss with respect to z^1 by applying the chain rule:

$$\frac{\partial L}{\partial z^1} = \frac{\partial L}{\partial a^1} \cdot \frac{\partial a^1}{\partial z^1}$$

The gradient with respect to W^0 is then:

$$\frac{\partial L}{\partial W^0} = \frac{\partial L}{\partial z^1} \cdot x^T$$

And the gradient with respect to b^0 is:

$$\frac{\partial L}{\partial b^0} = \frac{\partial L}{\partial z^1}$$

Step 7: Update the Parameters. Once all the gradients have been computed, we update the weights and biases using the gradient descent update rule:

$$W^i \leftarrow W^i - \eta \frac{\partial L}{\partial W^i}, \quad b^i \leftarrow b^i - \eta \frac{\partial L}{\partial b^i}$$

For all layers $i \in \{0, 1\}$, where η is the learning rate.