ECE 364: Programming Methods for Machine Learning, Spring 2025

Midterm 1 – March 11, 2025

- **You will have 75 minutes (1.25 hours) to solve all the problems. Most have multiple parts.** Don't spend too much time on questions you don't understand and focus on answering as much as you can!
- ***BUDGET YOUR TIME WISELY*.** I highly recommend working on the questions you know first and the questions you need to think about second.
- *No* resources are allowed for use during the exam except a cheatsheet and scratch paper on the back of the exam. **Do not tear out the cheatsheet or the scratch paper!** It messes with the auto-scanner.
- You should write your answers *completely* in the space given for the question. We will not grade parts of any answer written outside of the designated space.
- Please *use a dark-colored pen* unless you are *absolutely* sure your pencil writing is forceful enough to be legible when scanned. We reserve the right to take off points if we have difficulty reading the uploaded document.
- **Don't cheat.** C'mon, be cool, be honest.
- **Good luck!**

Name: ______________________________

NetID: ______________________________

Date: ______________________________

1. **Lost in the Layers: Navigating Data Slices** (24 points)

For each of the code segments, answer the following questions based on the final state of the variables.

(a)
```
import torch
a = torch.tensor([[0,1,2],[3,4,5],[6,7,8],[9,10,11]])
b = torch.arange(3).view(1,3)
c = a*b
c += 1
```

i. What will be the shape of `c`?

ii. What does `b` contain (note dimension in addition to value)?

(b)
```
import torch
d = torch.arange(100).view(10,10) #arange(N) gives int vector [0...N-1]
e = d[0:2, 1:6:4]
f = e[0,:]
g = f.add_(1)[:2]
h = g.t() @ g
```

i. What is the value of `h`?

ii. What does `e` contain?

2. **Feeling out of place** (5 points)

Suppose we want to find the value of x where $\ln(x^2) = 3$. We write the following gradient descent code to find the issue:

```
x_gd = torch.tensor(6.0, requires_grad=True)
target=3
alpha = 0.001
epochs = 20000
for i in range(epochs):
    f = torch.log(torch.pow(x_gd,2))
    loss = (target-f)**2
    loss.backward()
    with torch.no_grad():
        x_gd = x_gd - alpha*x_gd.grad
        x_gd.grad = None
```

...but there's an error! After much debugging, you narrow the issue down to the highlighted line above. What do you need to change this line to, so that the gradient descent code can work appropriately?

3. **Jacobians, Hessians, and Why My Brain Hurts (Matrix Calculus)** (5 points)

Given:

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

where $x \in \mathbb{R}^n$

Find the solution to the following partial derivative:

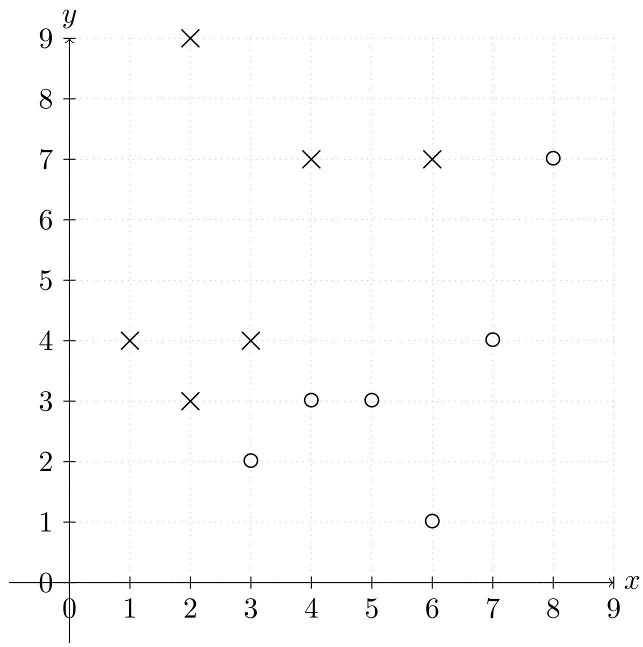
$$\frac{\partial f}{\partial x} =$$

You must explain why your answer is correct for full credit.

4. **The Art of the SVM Margin** (10 points)

Support vectors are the critical data points in an SVM that lie closest to the decision boundary. They determine both the optimal separating boundary and the margin width between classes. Removing these points from the training set would shift the boundary, potentially altering the classification results.

Consider a SVM that separates the following 2D points into two classes (X and O):



With points:

- **Class X:** $\{(1,4),(2,3),(2,9),(3,4),(4,7),(6,7)\}$
- **Class O:** $\{(3,2),(4,3),(5,3),(6,1),(7,4),(8,7)\}$

(a) Drawing the Boundary: Sketch the decision boundary for an SVM on the provided graph. Choose the decision boundary that maximizes the margin and minimizes the loss. Additionally, sketch the margin boundaries for each class (these are the lines that pass through the closest data points from each class, positioned half a margin away from the decision boundary). Label each line clearly. You should draw a total of three lines. Include the approximate equation for each line.

(b) Identifying Support Vectors: For each class (X and O) in the provided data, identify the support vectors and list them below.

(c) SVM Loss Function Analysis: Below is the loss function for soft-margin SVM, for a dataset of the form $\mathcal{D} := \{(x_i, y_i)\}_{i=1}^{N}$ and where $x_i \in \mathbb{R}^n$ and $y_i \in \{+1, -1\}$:

$$\mathcal{L} = \min_{w,\xi} \frac{1}{2}\|w\|_2^2 + C\sum_{i=1}^{n} \xi_i$$

where $\xi_i = \max(0, 1 - y_i \cdot W x_i)^2$. Here $\frac{1}{2}\|w\|_2^2$ is the regularization term, and $C\sum_{i=1}^{n} \xi_i$ is the slack penalty term. Explain briefly (more more than 4 sentences) what would happen if C is decreased. Include how training and test accuracy might be affected.

5. **Where to Draw the Line? A Classifier's Dilemma** (10 points)

Let g be the logical **IMPLICATION** function ($A \implies B$), defined on the feature space $\{+1, -1\}^2$, which maps:

- $g(+1, +1) = +1$
- $g(-1, +1) = +1$
- $g(+1, -1) = -1$
- $g(-1, -1) = +1$.

Given a linear classifier $h(x) = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x} + b)$, where $\boldsymbol{w} \in \mathbb{R}^{1\times 2}$, $\boldsymbol{x} \in \mathbb{R}^{2\times 1}$, and $b \in \mathbb{R}^{1\times 1}$, give a valid $(\boldsymbol{w}, b)$ pair that matches the ground truth g. Let $\text{sign}(z) = +1$ for $z \geq 0$ and -1 otherwise. Give your solution and show that it is valid.

6. **Stumbling Down the Gradient: My Life Story** (15 points)

Consider the following function

$$f(x, y) = -x \ln x - y \ln y$$

(a) Determine the gradient $\nabla f(x, y)$.

(b) Let the starting point for gradient descent at $k = 0$ be $\left(x^{(0)}, y^{(0)}\right) = (1, 1)$ and the step size be $\alpha = e - 1$. Here, e is Euler's number. Apply gradient descent to obtain the values of x and y at iterations $k = 1$ and $k = 2$.

7. **torch.nn models, Dataloaders and Optimizers, oh my!** (10 points)

Gradient accumulation is a technique that allows training with larger batch sizes by accumulating gradients over multiple smaller mini-batches before updating model parameters. Normally, in stochastic gradient descent, we compute gradients using `loss.backward()`, update parameters with `optimizer.step()`, and reset gradients using `optimizer.zero_grad()`.

However, if `optimizer.zero_grad()` is not called after each mini-batch, gradients accumulate over multiple iterations. This is useful when a large batch size cannot fit into memory. Instead of processing the full batch at once, we split it into smaller chunks, compute `loss.backward()` for each chunk, and update the model only after accumulating gradients from all chunks.

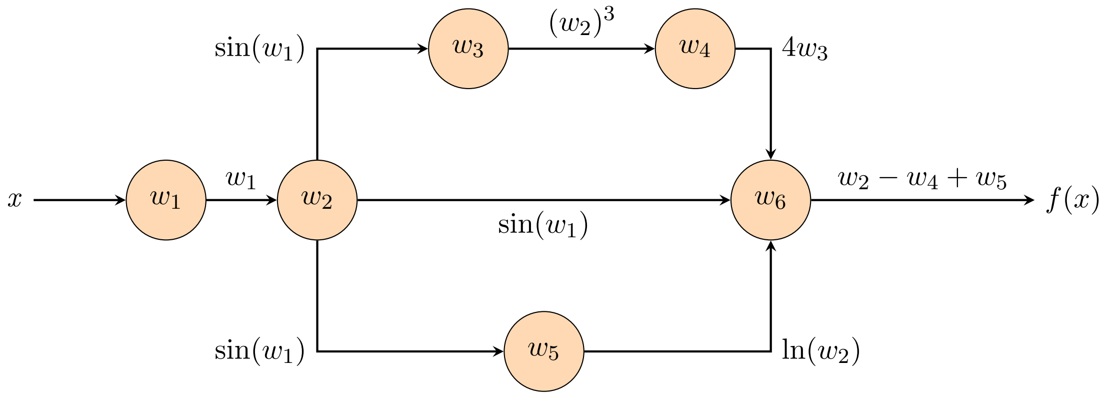
Now, suppose we want to train with a batch size of 128, but our device can only handle a batch size of 8. Use gradient accumulation to achieve this by completing the code snippet below and correctly placing `loss.backward()`, `optimizer.step()`, and `optimizer.zero_grad()`.

You can assume the following:

- `batch` is a dictionary containing the input and target, where the input has a shape of $B \times M$ and the target has a shape of B. B and M indicate the batch size and number of features.
- The Model (`model`) expects an input of shape $B \times M$ and produces an output of shape B.

```python
import torch
from torch.utils.data import DataLoader
from lib.model import Model
from lib.data import SimpleDataset

model = Model()
dataset = SimpleDataset()
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
# Complete this
train_loader =



# Complete this
num_accumulation_steps =



batches_processed = 0
optimizer.zero_grad()

# Add the missing function calls in the below
# training code
for epoch in range(10):
    for batch in train_loader: # Can add code between any of these commands


        inp, tgt = batch["input"], batch["target"]


        output = model(inp)


        loss = criterion(output, tgt)



        batches_processed += 1
        if batches_processed % num_accumulation_steps == 0:
            # one more place to fill in code






    #end of batch loop



#end of epoch loop
```

8. **The Computational Graph That Ate My Sanity** (21 points)



(a) Determine the function $f(x)$ represented by the above computational graph.

(b) Determine the partial derivatives of each successor node with respect to its predecessors, e.g., $\frac{\partial w_6}{\partial w_5}$, $\frac{\partial w_6}{\partial w_4}$, $\frac{\partial w_6}{\partial w_2}$, etc.

(c) Determine the adjoints at each node $\bar{w}_i = \frac{\partial f}{\partial w_i}$.

This page is for additional scratch work!

PyTorch Cheatsheet - Part 1

Useful activation function and torch.nn.functional

- Linear function: $y = WX + b$ where W and X are vectors of size N (number of dimensions to the input).

```
torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)
```

- Sigmoid function: $\frac{1}{1+e^{-z}}$ where z is the logit(s).

```
torch.nn.functional.sigmoid(input)
```

- Softmax function: $p(Y = t|x) = \frac{\exp(w_t^T x)}{\sum_{y\in\{0,\dots,C-1\}} \exp(w_y^T x)}$

```
torch.nn.functional.softmax(input, dim=None, _stacklevel=3, dtype=None)[source]
```

Loss Functions

- Mean squared error: $\ell(x,t;w) = (y-t)^2$

```
torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')
```

- Minimum log-likelihood: $\ell(x,t;w) = \sum_{(x^{(i)},t^{(i)})\in\mathcal{D}} -\log p(t|x)$
  - Combined with binary classification: $\ell(x,t;w) = \sum_{(x^{(i)},t^{(i)})\in\mathcal{D}} \log(1+\exp(-t^{(i)}w^T x^{(i)}))$
  - Combined with softmax: $\ell(x,t;w) = \sum_{(x^{(i)},t^{(i)})\in\mathcal{D}} \left(-w_{t^{(i)}}^T x + \log \sum_{c\in\{0,\dots,C-1\}} \exp(w_c^T x)\right)$

```
torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None,
    reduction='mean', label_smoothing=0.0)[source]
```

- Cross Entropy Loss:
  - Linear (SVM formulation): $\ell(x,t;w) = \frac{|W[1:]|}{2} + C\sum \max\left(0, 1 - t^{(i)} \cdot Wx^{(i)}\right)^2$
  - Logistic: $\ell(x,t;w) = -t\,\log y \;-\; (1-t)\,\log(1-y)$

Optimizers and torch.optim

In standard gradient descent, the update rule is: $\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha\nabla f(\mathbf{w}_k)$. In *gradient descent with momentum*, we introduce a *velocity term* $v_k : v_{k+1} = \beta v_k - \alpha\nabla f(\mathbf{w}_k)$ and $\mathbf{w}_{k+1} = \mathbf{w}_k + v_{k+1}$ where: α is the learning rate, $\beta \in [0,1]$ is the momentum coefficient, and v_k is the velocity term.

The following are some useful optimizers provided by the torch.optim library including:

- Stochastic gradient descent

```
torch.optim.SGD(params, lr=0.001, momentum=0, dampening=0, weight_decay=0, nesterov=False, *,
    maximize=False, foreach=None, differentiable=False, fused=None)[source]
```

PyTorch datasets

Required functions for dataset class:

- *__init__*: The *__init__* method is the constructor for the new dataset.
- *__len__*: The *__len__* method overrides the *len()* function in Python to determine the length of the dataset.
- *__getitem__*: The *__getitem__* method overloads the use of brackets to index items in a dataset.

There are lots of cool dataloader attributes and methods including:

- *batch_size*: number of examples in each batch or call to the dataloader
- *shuffle*: Boolean option to shuffle dataset each pass or *epoch* through the dataset
- *sampler*: *Sampler* object that specifies how data will be extracted from the dataset. For example, the *SubsetRandomSampler* allows us to specify indices within the larger dataset to sample at random.

Other useful equations

- Gradient descent: $\mathbf{w} \leftarrow \mathbf{w} - \alpha\frac{\partial\mathcal{E}}{\partial\mathbf{w}}$
- Closed form solution for linear regression: $W = (X^T X)^{-1} X^T T$
- L2 Regularization with MSE: $L(w) = \|y - Xw\|^2 + \lambda\|w\|_2^2$, closed form linear regression solutions: $W = (X^T X + \lambda I_d)^{-1} X^T y$
- Support Vector Machines - Margins at $WX = 1$ and $WX = -1$, border at $WX = 0$. Margin width $= 2/|W|$

Sample Code

Here is a sample, two-dimensional logistic classifier code:

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import SubsetRandomSampler

class LogisticRegression(nn.Module):
    def __init__(self, N):
        super().__init__()
        self.w = nn.Parameter(torch.ones(N))
        self.b = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return 1/(1+torch.exp(-(self.w@x+self.b)))

class TwoClassDataset(Dataset):
    # don't forget the self identifier!
    def __init__(self, N, sigma):
        self.N = N # number of data points per class
        self.sigma = sigma # standard deviation of each class cluster
        self.plus_class = self.sigma*torch.randn(N, 2) + torch.tensor([-1, 1])
        self.negative_class = self.sigma*torch.randn(N, 2) + torch.tensor([1, -1])
        self.data = torch.cat((self.plus_class, self.negative_class), dim=0)
        self.labels = torch.cat((torch.ones(self.N), torch.zeros(self.N)))

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        x = self.data[idx]
        y = self.labels[idx]
        return x, y # return input and output pair

N = 100
sigma = 1.5
dataset = TwoClassDataset(N, sigma)
plus_data = dataset.plus_class
negative_data = dataset.negative_class

# create indices for each split of dataset
N_train = 60
N_val = 20
N_test = 20
indices = np.arange(len(dataset))
np.random.shuffle(indices)
train_indices = indices[:N_train]
val_indices = indices[N_train:N_train+N_val]
test_indices = indices[N_train+N_val:]

# create dataloader for each split
batch_size = 8
train_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(train_indices)
    )
val_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(val_indices))
test_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(test_indices))

criterion = nn.BCELoss(reduction='mean') # binary cross-entropy loss, use mean loss
logreg_model = LogisticRegression(2) # initialize model
optimizer = torch.optim.SGD(logreg_model.parameters()) # initialize optimizer

n_epoch = 200 # number of passes through the training dataset
loss_values, train_accuracies, val_accuracies = [], [], []
for n in range(n_epoch):
    epoch_loss, epoch_acc = 0, 0
    for x_batch, y_batch in train_loader:
        optimizer.zero_grad()
        predictions = logreg_model(x_batch.unsqueeze(-1)).squeeze(-1)
        loss = criterion(predictions, y_batch)
        loss.backward()
        optimizer.step()
```