ECE 364: Programming Methods for Machine Learning, Spring 2026

Midterm 1 – March 10, 2026

- **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. **True/False** (12 points)

For each of the following problems, circle TRUE if the statement is *always* true, circle FALSE otherwise. There is no partial credit for these questions.

a. Linear regression models assume that the relationship between the independent variable(s) x and the dependent variable y can be represented as a linear combination of the input features.

TRUE FALSE

b. Logistic regression is primarily used to predict continuous numerical values, such as predicting the exact price of a house.

TRUE FALSE

c. In PyTorch, a Tensor's `.requires_grad` attribute must be set to `True` to track operations on it for automatic gradient computation.

TRUE FALSE

d. The Rectified Linear Unit (ReLU) activation function is defined by the mathematical expression $f(x) = \max(0, x)$.

TRUE FALSE

e. In PyTorch, executing `loss.backward()` computes the gradients and immediately updates the weights of the model to minimize the loss.

TRUE FALSE

f. It is **not** advisable to pair the mean squared error loss function with the Sigmoid activation function because the gradients at large input values tend to be small.

TRUE FALSE

g. The Swish activation function, defined as $f(x) = x \cdot \text{sigmoid}(\beta x)$, is strictly increasing for all positive values of β.

TRUE FALSE

h. In ReLU, if a neuron's weights are updated such that the function always outputs zero for all inputs in the training set, its gradient effectively becomes zero permanently.

TRUE FALSE

2. **Lost in the Layers: Navigating Data Slices** (14 points)

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

(a)
```
import torch
data = torch.zeros((3, 2))
bias = torch.tensor([[10.], [20.], [30.]])
result = data + bias
result[result > 15] -= 5
```

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

ii. What does `result` contain?

(b)
```
import torch
a = torch.arange(42).view(3, -1) # arange(N) gives int vector [0...N-1]
b = a[:, ::5]
b.add_(1)
```

i. What does `a[:, 0]` return?

ii. What is the shape of `b`?

iii. What does `b` contain?

3. **Feeling out of place** (6 points)

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

```
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 -= alpha * x_gd.grad
```

...but there's an error! After much debugging, you narrow the issue down to `x_gd`. For some reason `x_gd` keeps becoming infinite! What do you need to change/add/delete, so that the gradient descent code can work appropriately?

4. **Jacobians, Hessians, and Why My Brain Hurts (Matrix Calculus)** (6 points)

Given:

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

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

Find the solution to the following partial derivative:

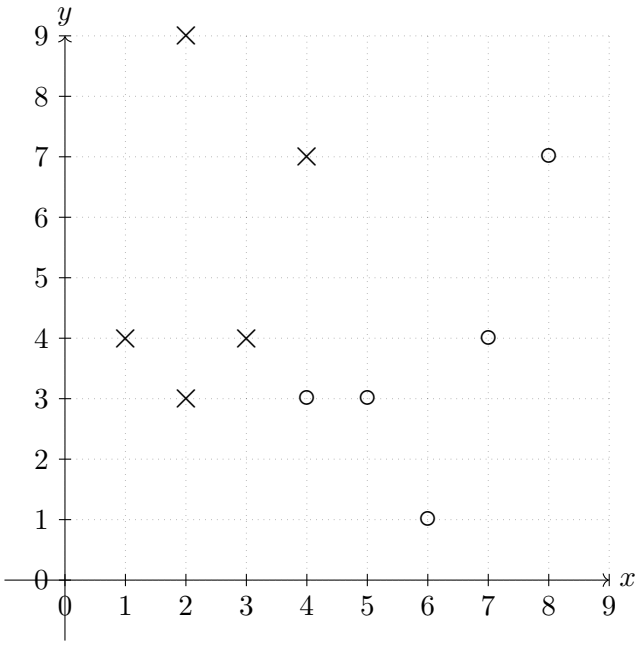
$$\frac{\partial f}{\partial x} =$$

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

5. **The Art of the SVM Margin** (12 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:** $\{(1,4),(2,3),(2,9),(3,4),(4,7)\}$
- **Class O:** $\{(4,3),(5,3),(6,1),(7,4),(8,7)\}$

(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) If we remove $(8, 7)$ belonging to class $\circ$ from the dataset, what happens to the decision boundary? Briefly explain your reasoning.

(c) If we remove $(2, 3)$ and $(3, 4)$ belonging to class $\times$ from the dataset, what happens to the decision boundary? Briefly explain your reasoning.

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

Let g be a logical function, defined on the feature space $\{+1, -1\}^3$, which maps:

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

Given a linear classifier $h(x) = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x} + b)$, where $\boldsymbol{w} \in \mathbb{R}^{1\times 3}$, $\boldsymbol{x} \in \mathbb{R}^{3\times 1}$, 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.

7. **Stumbling Down the Gradient: My Life Story** (12 points)

Consider the following function

$$f(x,y) = \frac{1}{2e}\left(x^2 + y^2\right) - \left(\frac{1}{2} + \frac{1}{e}\right)(x+y)$$

(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 $\alpha = 2(e-1)$. Here, e is Euler's number. Apply gradient descent to obtain the values of x and y at iterations $k = 1$ and $k = 2$.

8. **torch.nn models, Dataloaders and Optimizers, oh my!** (15 points)

A real estate agency is modeling home prices in a historic district using three factors: (1) area, (2) number of bedrooms, and (3) age of home in years. They have collected data on 50 homes. Some examples from the dataset are as follows:

Area (sqft)	# Bedrooms	Age (years)	Price ($)
1,500	2	5	250,000
1,800	3	6	310,000
2,200	3	4	360,000
2,500	4	2	420,000
3,000	5	5	510,000

The prices are modeled as

$$\text{Price} = w_0 + w_1 \times \text{Area} + w_2 \times \text{No. of Bedrooms} + w_3 \times \text{Property Age} \tag{1}$$

Based on this setup, answer the questions below.

i. Complete the following dataset class. The i^{th} element (accessed as `dataset[i]`) must return a tuple `(x, y)`. `x` is a 1-D tensor with three elements: (1) area, (2) bedrooms, and (3) age. `y` is a 0-D (or scalar) containing the corresponding price.

```
import torch
from torch.utils.data import Dataset

class HomePriceDataset(Dataset):
    def __init__(self):
        # i-th elements of area_list, bedrooms_list, age_list, and
        # price_list contain the area, the number of bedrooms, the
        # age, and the price of the i-th home.
        self.area_list = [1500, 1800, 2200, 2500, 3000]
        self.bedrooms_list = [2, 3, 3, 4, 5]
        self.age_list = [5, 6, 4, 2, 5]
        self.price_list = [250000, 310000, 360000, 420000, 510000]

    def __len__(self):
        # Implement this




    def __getitem__(self, index: int):
        # Implement this (x is the input, and y is the output)









        return x, y
```

ii. Complete the following class to implement the model shown in Eq. 1. The `forward` method accepts a batched input tensor of size $B \times 3$ (where B is the batch size). It must return a tensor of predicted prices with size B.

```python
import torch
import torch.nn as nn

class HomePriceModel(nn.Module):
    def __init__(self, num_features: int):
        # Implement the model









    def forward(self, x: torch.Tensor):
        """
        x is a tensor of shape (B, 3).
        """
        # Implement the forward method









# For the home price dataset, the model will be initialized as
model = HomePriceModel(num_features=3)
# End implementation
```

iii. Write the code to train `HomePriceModel` on `HomePriceDataset` for 10 epochs. Use the SGD optimizer with a learning rate of 0.01, mean-squared error loss, and a batch size of 2. No validation step is required.

```
import torch.nn as nn
from torch.optim import SGD

# Initialize model and dataset




# Create dataloader


# Initialize optimizer


for epoch_idx in range(10):
    # Write training code









    # End epoch
```

9. **The Computational Graph That Ate My Sanity** (18 points)

Recall the softmax function from the lectures. A slight variation is the log-softmax function which applies to the output of softmax. It can be defined as

$$\text{LogSoftmax}\left(x_i\right) = \log \frac{e^{x_i}}{\sum_j e^{x_j}} = x_i - \log \sum_j e^{x_j}$$

Consider a 3-dimensional tensor $g(x, y, z)$.LogSoftmax(x) can be written as

$$g(x, y, z) = x - \log\left(e^x + e^y + e^z\right)$$

The below computation graph depicts $g(x, y, z)$

(a) For the graph in Figure 1, we have $w_1 = x, w_2 = y$, and $w_3 = z$. Express each intermediate node $(w_4, w_5, w_6, w_7, w_8)$ value in terms of inputs (x, y, z) to construct $g(x, y, z)$. Note that $g(x, y, z) = w_9$ and $w_9 = w_1 - w_8$.

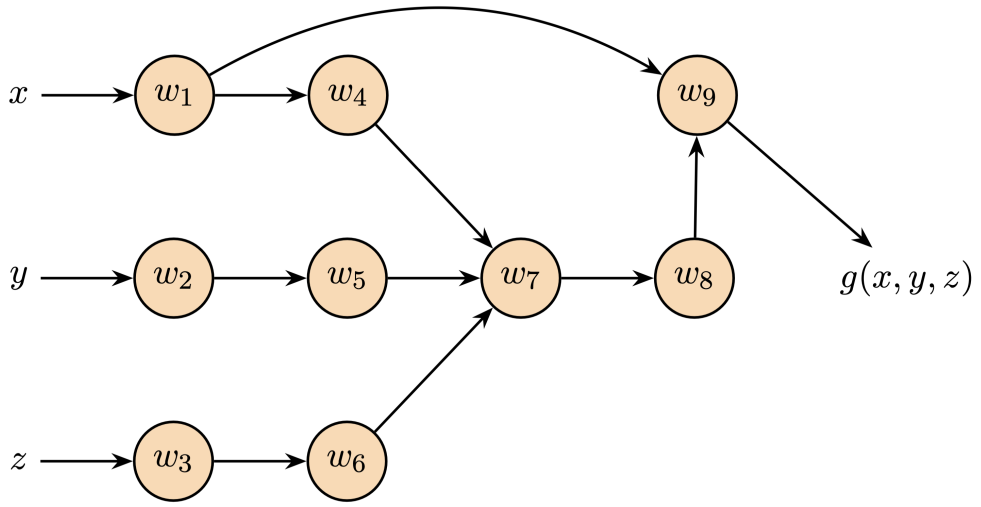


Figure 1: Computation graph for question 9

(b) Determine the partial derivatives of each successor node with respect to its predecessors, e.g., $\frac{\partial w_4}{\partial w_1}$, $\frac{\partial w_5}{\partial w_2}$, $\frac{\partial w_6}{\partial w_3}$, etc. The answers should be in term of x, y, z.

(c) Determine the adjoints at each node $\bar{w}_i = \frac{\partial f}{\partial w_i}$, the answers should be in term of x, y, z.

This page is for additional scratch work!

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): $\ell(x, t; w) = \frac{|W[1:]|}{2} + C \sum \max\left(0, 1 - t^{(i)} \cdot W x^{(i)}\right)^2$
  - 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()
```