Student Name: _____

Student Email: _____

Deep Learning Midterm (CS 7150)
Fall 2022

Instructions

- Important: WRITE YOUR NAME and your Northeastern email address on EVERY page.
- Write your answers in the spaces directly after each question.
- You have 90 minutes to do the exam.
- For worked questions, draw a single BOX around your final answer, so that graders can tell the difference between your answer and your scratch work.
- You may use the supplied scratch paper. Let the proctor know if you need more. Scratch calculations will not be graded.
- All cell phones and other electronic devices must be turned off during this exam.
- No outside aids or resources are allowed. You are not allowed to use your notes, books, computers, calculators, or any other resource.
- You are not allowed to communicate with any person other than the instructor / exam proctor while taking this exam.
- You may not share, disseminate, or discuss these questions with any other student in this course who has not taken the exam yet. Doing so is considered academic dishonesty and will lead to nullification of exam grades.
- This midterm exam may not be posted online, copied, or disseminated.

Signature

Sign below to attest that you have read and will follow the closed-book exam rules above.

Student signature: _____ Date: _____

Student Name: _____ Student Email: _____

1. True or false, circle one.
   (a) A two-layer neural network can exactly match any binary logic truth table, given enough neurons. True / False.
   (b) There are continuous functions that no two-layer neural network can approximate closely regardless of how many neurons are used. True / False.
   (c) A deep linear network wihout nonlinearities can compute the same functions as a network with nonliearities, if given enough neurons. True / False.
   (d) Backpropagation to get $\frac{\partial \mathcal{L}}{\partial x}$ typically requires much more time (e.g., many more additions and multiplications) than computing the original function $\mathcal{L}(x)$. True / False.
   (e) Backpropagation to get $\frac{\partial \mathcal{L}}{\partial x}$ typically requires much more memory (e.g., many more numbers stored at once) than computing the original function $\mathcal{L}(x)$. True / False.
   (f) 100% accuracy on the training set tells us the network generalizes well. True / False
   (g) 100% accuracy on the training set tells us the network does **not** generalize. True / False
   (h) 100% accuracy on a holdout test set tells us the network generalizes well. True / False
   (i) 100% accuracy on a holdout test set tells us the network does **not** generalize. True / False
   (j) Enlarging the training set will tend to reduce the train/test accuracy gap. True / False
   (k) Enlarging each network layer will tend to reduce the train/test accuracy gap. True / False
   (l) Data augmentation will tend to reduce the train/test accuracy gap. True / False
   (m) Lower learning rates will tend to reduce the train/test accuracy gap. True / False
   (n) Dropout layers will tend to reduce the train/test accuracy gap. True / False
   (o) Xavier weight initialization will tend to reduce the train/test accuracy gap. True / False
   (p) Xavier weight initialization will help avoid vanishing gradients. True / False
   (q) Weight decay will help avoid vanishing gradients. True / False.
   (r) Batch normalization will help avoid vanishing gradients. True / False.
   (s) The sigmoid nonlinearity will help avoid vanishing gradients. True / False.
   (t) Enlarging the batch size will help avoid vanishing gradients. True / False.
   (u) As long as the gradient is nonzero, updating parameters with a small enough learning rate guarantees a decrease in the loss. True / False.
   (v) As long as the gradient is nonzero, updating parameters with large enough momentum guarantees a decrease in the loss. True / False.
   (w) If you are using plain SGD, then dividing the loss by 10 before doing backpropagation will have the same effect as reducing learning rate by a factor of 10. True / False.
   (x) If you are using momentum, dividing the loss by 10 will have little effect. True / False.
   (y) If are using ADAM, dividing the loss by 10 will have little effect. True / False.

Student Name: _____ Student Email: _____

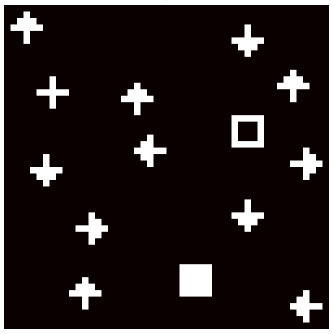
2. Convolutions

Pytorch defines a 1-channel 5x5 convolution with padding of 2 as:

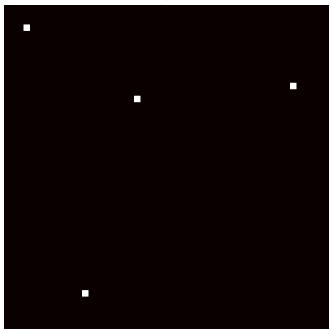
$$\text{output}_{x,y} = \text{bias} + \sum_{i=0}^{4}\sum_{j=0}^{4} \text{weight}_{i,j} \cdot \text{input}_{x+i-2,y+j-2} \tag{1}$$

Your goal is to create a small neural network using this kind of convolution that transforms the grid of input data on the left to the output data in the middle. The input grid was formed by placing a bunch of 5x5 emoji-style symbols as +1 patterns in a field of −1 values, and every point in both the input and output data is ±1. An overlay of the input and desired output are shown on the right.

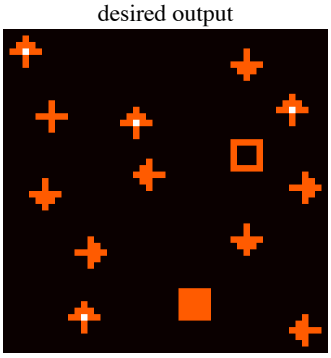
Given input:
1x1x50x50 tensor
white = +1, black = -1



Desired output:
1x1x50x50 tensor
white = +1, black = -1



Detail: overlay of
given input and
desired output



(a) The following code solves the problem, but the specific numbers for the convolution weights and bias need to be supplied. Fill in numbers that will make the program produce the desired output. Hint: you can use ±1 for most of the numbers.

```
import torch
torch.set_grad_enabled(False)

conv = torch.nn.Conv2d(
    in_channels  = 1,
    out_channels = 1,
    kernel_size  = (5,5),
    padding      = 2)

class Sign(torch.nn.Module):
    def forward(self, x):
        return x.sign()

net = torch.nn.Sequential(
    conv,
    Sign()
)

input_data = torch.load('input_data.pt')

```

```
conv.weight[:,:,:,:] = torch.tensor([[[

    [ ,  ,  ,  ,  ],

    [ ,  ,  ,  ,  ],

    [ ,  ,  ,  ,  ],

    [ ,  ,  ,  ,  ],

    [ ,  ,  ,  ,  ],

]]])
conv.bias[:] = torch.tensor(

    [ ]

)

output = net(input_data)
```

(b) This particular convolution has some symmetry in the way it acts on its data. Circle which types of symmetry it has.

Invariant under horizontal reflection $(x, y) \leftrightarrow (-x, y)$	Equivariant under horizontal reflection $(x, y) \leftrightarrow (-x, y)$
Invariant under 180 degree rotation	Equivariant under 180 degree rotation
Invariant under translation	Equivariant under translation

Student Name: _____ Student Email: _____

3. Weight decay

Weight decay adds a term to the loss that sums the square of all the model parameters w_i:

$$\mathcal{L}_{\text{wd}} = \delta \sum_i ||w_i||^2 \tag{2}$$

(a) What is the purpose of weight decay?

(b) Suppose that weight decay is the only term in the loss, $\mathcal{L} = \mathcal{L}_{\text{wd}}$, and we have set weight decay $\delta = 0.1$ and learning rate $\lambda = 0.1$, using ordinary gradient descent that updates parameters in the direction of the gradient:

$$w_i^{(\text{updated})} := w_i - \lambda \frac{\partial \mathcal{L}}{\partial w_i} \tag{3}$$

What is the value of $w_i^{(\text{updated})}$ after one iteration of gradient descent? Write a simple expression in terms of the original w_i.

(c) Now also suppose the network $z = f(x)$ is an ordinary feedforward network that outputs a vector of logits z, with several linear layers and the ReLU nonlinearity between each layer. What does $f(x)$ converge towards as the number of training iterations increases towards infinity if the whole loss is $\mathcal{L} = \mathcal{L}_{\text{wd}}$?

(d) Now suppose instead that $f(x)$ is a residual network with no bottleneck layers. In that case, what does $f(x)$ converge towards as training iterations go to infinity? Again assume the whole loss is $\mathcal{L} = \mathcal{L}_{\text{wd}}$.

4. Back propagation

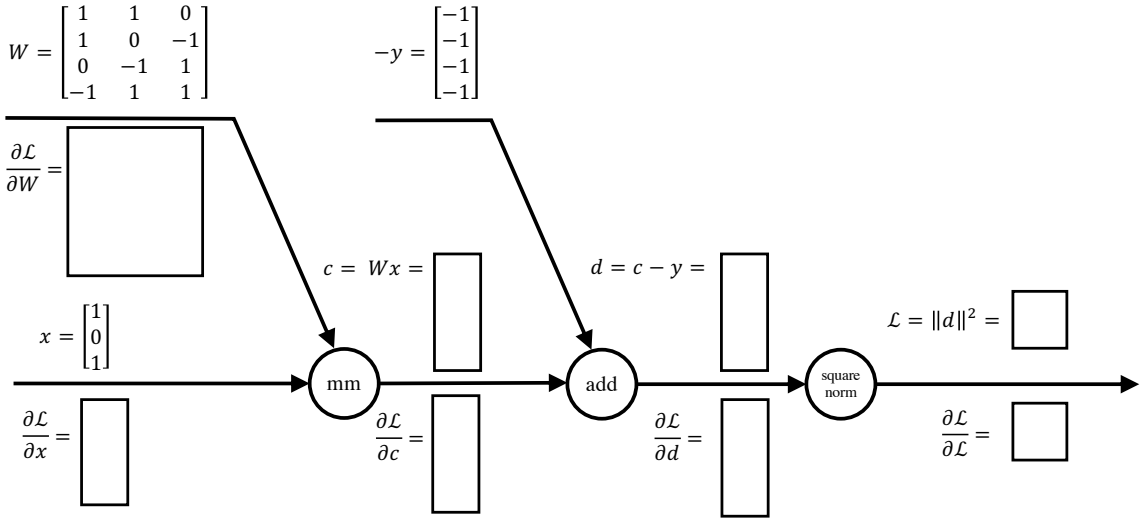
The following computation graph computes the squared error of $\mathcal{L} = ||Wx - y||^2$. Your goal is to compute $\frac{\partial \mathcal{L}}{\partial W}$ as well as $\frac{\partial \mathcal{L}}{\partial x}$ using the backpropagation algorithm, for the specific values of W and x shown in the figure.

Keep in mind that $||d||^2$ is defined as

$$||d||^2 = \sum_i d_i^2 \tag{4}$$

And also keep in mind that the shape of the gradient with respect to any given tensor result is always the same as the shape of the forward result.

(a) Fill in the forward-pass in the figure below by filling in the values of the tensors above each arrow where there is a box.



(b) Fill in the backward-pass in the figure above by filling in the boxes below the arrows.

Student Name: _____ Student Email: _____

5. Pytorch modules

Read and think about the following code.

```
1  import torch
2  from torch.nn import Sequential, ReLU, Linear
3  # commented out: # torch.set_grad_enabled(False)
4  num_pixels = 1000000
5  bottleneck = 100
6  net = Sequential(
7      Linear(in_features=num_pixels, out_features=bottleneck, bias=True),
8      ReLU(),
9      Linear(in_features=bottleneck, out_features=num_pixels, bias=False)
10 ).cuda()
11 print('Parameter tensors', len(list(net.named_parameters())))
12 print('Parameter elements', sum(p.numel() for p in net.parameters()))
13 total_error = 0
14 sample_size = 10000
15 for test_index in range(sample_size):
16     test_data = torch.randn(1, num_pixels, device='cuda')
17     total_error += (net(test_data) - test_data).pow(2).mean()
18 print('Average error', total_error / sample_size)
```

(a) Is this network being used as a classifier or an autoencoder? How can you tell?

(b) Think about which parameters are contained in a Linear layer, either when it has a bias term, or when it does not. What is the count of parameter tensors that you expect to be printed on line 11?

(c) Think about how big the parameter matrices are in a Linear layer. What is the total count of parameters that you expect to be printed on line 12?

(d) When you run this code, it prints out values on lines 12 and 13 quickly but then it always crashes after about 10 seconds, saying 'CUDA out of memory' before getting to line 18. Why is it running out of memory? How could you fix the code so that it easily runs 10,000 iterations using the GPU without changing the network or the data; it should be able to quickly print out something like 'Average error 1.055' on line 18?

6. Softmax and cross-entropy loss

Label smoothing is a classification objective that reduces the overconfidence of a model by moderating the ordinary classification goal. Here is how it works: when given training data x labeled with class c, the goal is not to give that class a probability of one, but instead to aim for probability $y_c = 1 - \alpha$, while distributing the rest of the probability to other classes evenly.

For example, if $\alpha = 0.1$ and we have two classes, then the target probability of something in class 2 will be $y_2 = 0.9$ and $y_1 = 0.1$. When using label smoothing, we still use the typical softmax cross-entropy loss, so that we wish to learn a model $z = f(x)$ that computes logits z from data x, so that the predictions $p = \text{softmax}(z)$ minimize the loss $\mathcal{L}_{\text{ce}} = -\text{CE}(y; p)$.

In other words, these ordinary definitions still apply:

$$\mathcal{L}_{\text{ce}} = -\text{CE}(y; p) = -\sum_i y_i \log p_i \tag{5}$$

$$p_j = \text{softmax}(z)_j = \frac{e^{z_j}}{\sum_i e^{z_i}} \tag{6}$$

Now, set label smoothing with $\alpha = 0.1$ for a classification problem with $n = 2$ classes. Our network will output two logits $[z_1, z_2] = f(x)$ for any input x. Each part of the question below will examine a different data point x, all which have the true class label $c = 2$.

(a) Suppose x is in class $c = 2$. Then think about the target y_1 and y_2 defined above, and which probability vector $p = \text{softmax}(f(x))$ would minimize $\mathcal{L}_{\text{ce}} = -\text{CE}(y; p)$. What value should the predicted probability p_2 take to minimize the cross-entropy loss?

(b) Now consider another point x that also has class $c = 2$, and suppose the first logit is $z_1 = 0$ (i.e., the number zero). In part (a) you figured out what p_2 must be. Now that you know p_2, what value must the second logit z_2 have? Don't use a calculator: you can express your answer in terms of log or exp.