Homework #3

CSE 446/546: Machine Learning
Profs. Jamie Morgenstern and Simon Du
Due: **Wednesday** November 17, 2021 11:59pm
A: 71 points, **B:** 15 points

Please review all homework guidance posted on the website before submitting to GradeScope. Reminders:

- Make sure to read the "What to Submit" section following each question and include all items.
- Please provide succinct answers and supporting reasoning for each question. Similarly, when discussing experimental results, concisely create tables and/or figures when appropriate to organize the experimental results. All explanations, tables, and figures for any particular part of a question must be grouped together.
- For every problem involving generating plots, please include the plots as part of your PDF submission.
- When submitting to Gradescope, please link each question from the homework in Gradescope to the location of its answer in your homework PDF. Failure to do so may result in deductions of up to *[5 points]* . For instructions, see `https://www.gradescope.com/get_started#student-submission`.
- Please recall that B problems, indicated in boxed text, are only graded for 546 students, and that they will be weighted at most 0.2 of your final GPA (see the course website for details). In Gradescope, there is a place to submit solutions to A and B problems separately. You are welcome to create a single PDF that contains answers to both and submit the same PDF twice, but associate the answers with the individual questions in Gradescope.
- If you collaborate on this homework with others, you must indicate who you worked with on your homework. Failure to do so may result in accusations of plagiarism.
- For every problem involving code, please submit your code to the separate assignment on Gradescope created for code. Not submitting all code files will lead to a deduction of *[1 point]* .
- Please indicate your final answer to each question by placing a box around the main result(s). To do this in LaTeX, one option is using the `boxed` command.

Not adhering to these reminders may result in point deductions.

Short Answer and "True or False" Conceptual Questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. *[2 points]* Say you trained an SVM classifier with an RBF kernel ($K(u,v) = \exp\left(-\frac{\|u-v\|_2^2}{2\sigma^2}\right)$). It seems to underfit the training set: should you increase or decrease σ?

b. *[2 points]* True or False: Training deep neural networks requires minimizing a non-convex loss function, and therefore gradient descent might not reach the globally-optimal solution.

c. *[2 points]* True or False: It is a good practice to initialize all weights to zero when training a deep neural network.

d. *[2 points]* True or False: We use non-linear activation functions in a neural network's hidden layers so that the network learns non-linear decision boundaries.

e. *[2 points]* True or False: Given a neural network, the time complexity of the backward pass step in the backpropagation algorithm can be prohibitively larger compared to the relatively low time complexity of the forward pass step.

f. *[2 points]* True or False: Neural Networks are the most extensible model and therefore the best choice for every circumstance.

What to Submit:

- **Parts a-f:** 1-2 sentence explanation containing your answer.

Answers:
Part a:
If it is underfitting the training set, then we will want to tighten the margins. We can do so by decreasing the value of σ

This is because when an SVM classifier is trained with an RBF kernel, it leads to the following equation:

$$K(u,v) = \exp\left(-\frac{||u-v||_2^2}{2\sigma^2}\right) = \exp(-\gamma||u-v||_2^2, \text{ where } \sigma \text{ and} c\ \gamma \text{ are inversely proportional}$$

Part b:
True. When minimizing a non-convex loss function, gradient descent might reach the local optimal solution rather than the global optimal solution.

Part c:
False. It is a bad practice. This can be seen in the example of a neural network with sigmoid activation function. If the weights are initialized to zero, then the derivative with respect to the loss function will be identical across all the weights. This results in an undesirable training situation

Part d:
False. A non-linear functions introduces non-linearity into the neural network's hidden layer. It is not necessarily so that that network learns non-learn decision boundaries

Part e:
False. The time complexities of the backward pass step and the forward pass step are the same

Part f:
False. Each machine learning algorithm has a different inductive bias, so it's not always appropriate to use neural networks. For example, a linear trend will always be learned best by simple linear regression rather than a ensemble of nonlinear networks.

Support Vector Machines

A2. Assume w is an n-dimensional vector and b is a scalar. A hyperplane in $\mathbb{R}^n$ is the set $\{x \in \mathbb{R}^n \mid w^\top x + b = 0\}$.

a. *[1 point]* ($n = 2$ example) Draw the hyperplane for $w = \begin{bmatrix}-1 & 2\end{bmatrix}^\top$, $b = 2$? Label your axes.

b. *[1 point]* ($n = 3$ example) Draw the hyperplane for $w = \begin{bmatrix}1 & 1 & 1\end{bmatrix}^\top$, $b = 0$? Label your axes.

c. *[2 points]* Let $w^\top x + b = 0$ be the hyperplane generated by a certain SVM optimization. Given some point $x_0 \in \mathbb{R}^n$, Show that the minimum *squared distance* to the hyperplane is $\frac{|x_0^\top w + b|}{\|w\|_2}$. In other words, show the following:

$$\min_x \|x_0 - x\|_2 = \frac{|x_0^\top w + b|}{\|w\|_2}$$
$$\text{subject to: } w^\top x + b = 0\,.$$

Hint: Think about projecting x_0 onto the hyperplane.

What to Submit:

- **Parts a-b:** Graph or an image of hyperplane
- **Part c:** Solution and corresponding calculations.

Answers:
Part a:

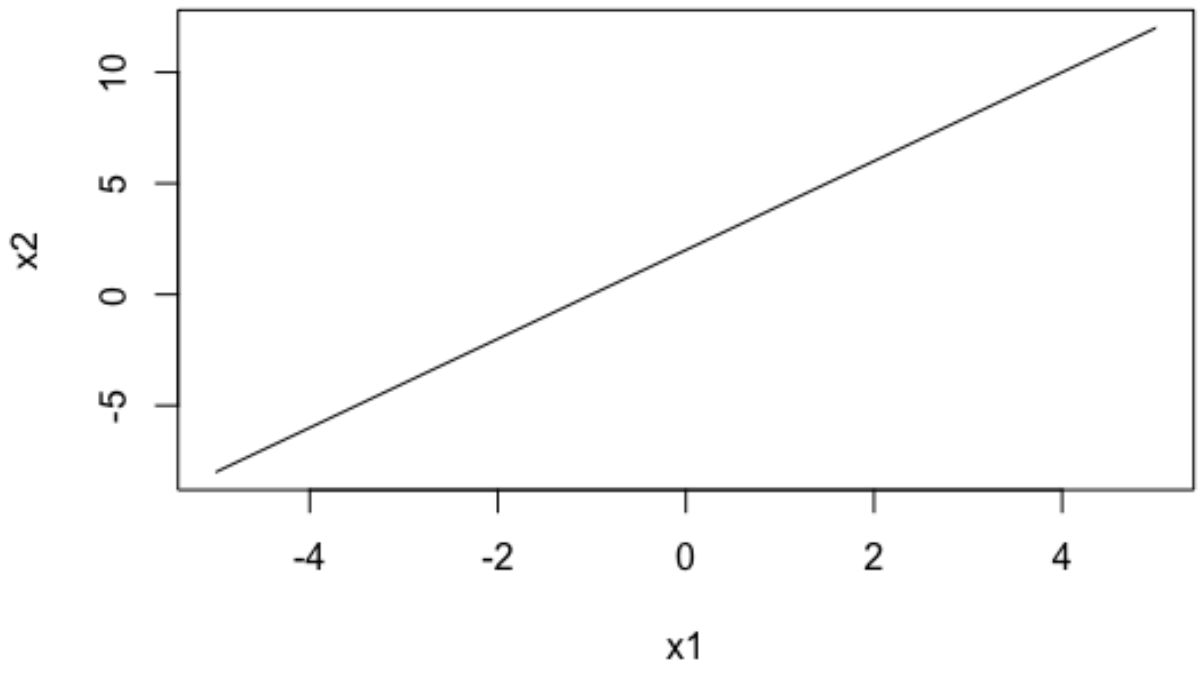

Figure 1: hyperplane for $w = \begin{bmatrix}-1 & 2\end{bmatrix}^\top$, $b = 2$

Part b:

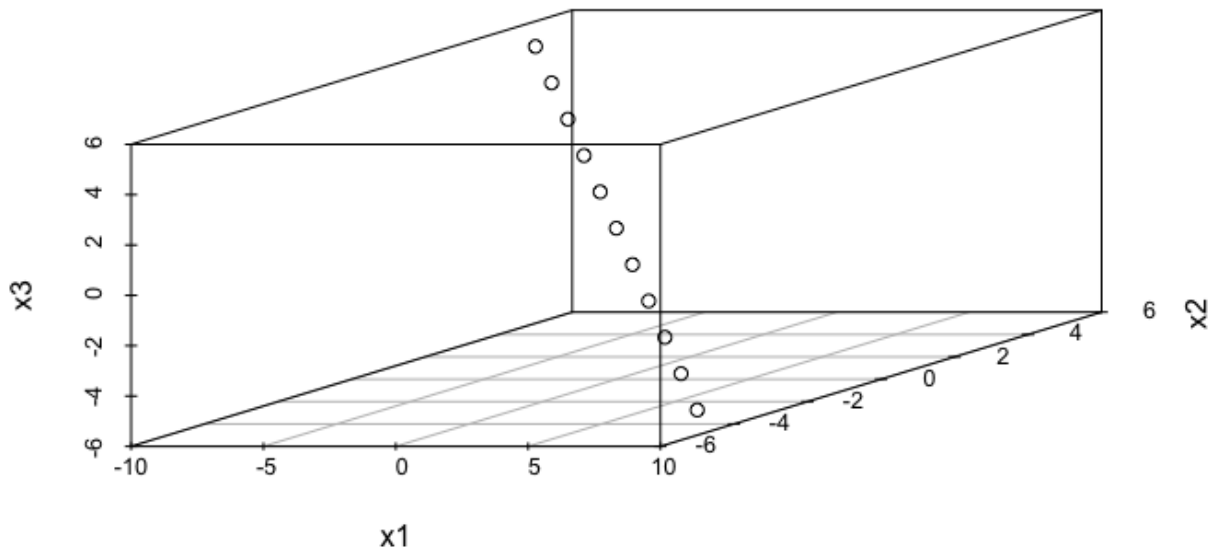

Figure 2: hyperplane for $w = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^\top$, $b = 0$

Part c:

$\arg\min_x ||x_0 - x||$ is the projection of x0 on the hyperplane

$$\begin{aligned} \implies ||x_0 - x||_2 &= \left| (x_0^T - x)^T \frac{w}{||w||_2} \right| \\ &= \frac{1}{||w||_2} |x_0^T w - x^T w| \\ &= \frac{|x_o^T w + b|}{||w||_2} \end{aligned}$$

Kernels

A3. *[5 points]* Suppose that our inputs x are one-dimensional and that our feature map is infinite-dimensional: $\phi(x)$ is a vector whose ith component is:

$$\frac{1}{\sqrt{i!}}\mathrm{e}^{-x^2/2}x^i \ ,$$

for all nonnegative integers i. (Thus, ϕ is an infinite-dimensional vector.) Show that $K(x, x') = \mathrm{e}^{-\frac{(x-x')^2}{2}}$ is a kernel function for this feature map, i.e.,

$$\phi(x) \cdot \phi(x') = \mathrm{e}^{-\frac{(x-x')^2}{2}} \ .$$

Hint: Use the Taylor expansion of $z \mapsto \mathrm{e}^z$. (This is the one dimensional version of the Gaussian (RBF) kernel).

What to Submit:

- Proof or formal argument.

Answers:

$$\begin{aligned}\phi(x) \cdot \phi(x') &= \sum_{i=0}^{\infty} \frac{1}{\sqrt{i!}} e^{-\frac{x^2}{2}} x^i \cdot \frac{1}{\sqrt{i!}} e^{-\frac{x^2}{2}} x'^i \\ &= e^{-\frac{x^2+x'^2}{2}} \sum_{i=0}^{\infty} \frac{(xx')^i}{i!}\end{aligned}$$

Using Taylor expansion, we get:

$$\begin{aligned}\phi(x) \cdot \phi(x') &= e^{-\frac{x^2+x'^2}{2}} e^{xx'} \\ &= e^{-\frac{(x-x')^2}{2}}\end{aligned}$$

B1.

Intro to sample complexity

For $i = 1, \ldots, n$ let $(x_i, y_i) \overset{\text{i.i.d.}}{\sim} P_{X,Y}$ where $y_i \in \{-1, 1\}$ and x_i lives in some set $\mathcal{X}$ (x_i is not necessarily a vector). The 0/1 loss, or *risk*, for a deterministic classifier $f \colon \mathcal{X} \to \{-1, 1\}$ is defined as:

$$R(f) = \mathbb{E}_{X,Y}[\mathbf{1}(f(X) \neq Y)]$$

where $\mathbf{1}(\mathcal{E})$ is the indicator function for the event $\mathcal{E}$ (the function takes the value 1 if $\mathcal{E}$ occurs and 0 otherwise). The expectation is with respect to the underlying distribution $P_{X,Y}$ on (X, Y). Unfortunately, we don't know $P_{X,Y}$ exactly, but we do have our i.i.d. samples $\{(x_i, y_i)\}_{i=1}^n$ drawn from it. Define the *empirical risk* as

$$\widehat{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}(f(x_i) \neq y_i) \ ,$$

which is just an empirical estimate of our risk. Suppose that a learning algorithm computes the empirical risk $R_n(f)$ for all $f \in \mathcal{F}$ and outputs the prediction function $\widehat{f}$ which is the one with the smallest empirical risk. (In this problem, we are assuming that $\mathcal{F}$ is finite.) Suppose that the best-in-class function f^* (i.e., the one that minimizes the true 0/1 loss) is:

$$f^* = \arg\min_{f \in \mathcal{F}} R(f) \, .$$

a. *[2 points]* Suppose that for some $f \in \mathcal{F}$, we have $R(f) > \epsilon$. Show that

$$\mathbb{P}\left[\widehat{R}_n(f) = 0\right] \leq \mathrm{e}^{-n\epsilon} \ .$$

(You may use the fact that $1 - \epsilon \leq \mathrm{e}^{-\epsilon}$.)

b. *[2 points]* Use the *union bound* to show that

$$\mathbb{P}\left[\exists f \in \mathcal{F} \text{ s.t. } R(f) > \epsilon \text{ and } \widehat{R}_n(f) = 0\right] \leq |\mathcal{F}| \mathrm{e}^{-\epsilon n}.$$

Recall that the union bound says that if $A_1, \ldots, A_k$ are events in a probability space, then

$$\mathbb{P}\left[A_1 \cup A_2 \cup \ldots \cup A_k\right] \leq \sum_{1 \leq i \leq k} \mathbb{P}(A_i).$$

c. *[2 points]* Solve for the minimum ϵ such that $|\mathcal{F}| \mathrm{e}^{-\epsilon n} \leq \delta$.

d. *[4 points]* Use this to show that with probability at least $1 - \delta$

$$\widehat{R}_n(\widehat{f}) = 0 \quad \Longrightarrow \quad R(\widehat{f}) - R\left(f^*\right) \leq \frac{\log(|\mathcal{F}|/\delta)}{n}$$

where $\widehat{f} = \arg\min_{f \in \mathcal{F}} \widehat{R}_n(f)$.

Context: Note that among a larger number of functions $\mathcal{F}$ there is more likely to exist an $\widehat{f}$ such that $\widehat{R}_n(\widehat{f}) = 0$. However, this increased flexibility comes at the cost of a worse guarantee on the true error reflected in the larger $|\mathcal{F}|$. This trade-off quantifies how we can choose function classes $\mathcal{F}$ that over fit. This sample complexity result is remarkable because it depends just on the number of functions in $\mathcal{F}$, not what they look like. This is among the simplest results among a rich literature known as 'Statistical Learning Theory'. Using a similar strategy, one can use Hoeffding's inequality to obtain a generalization bound when $\widehat{R}_n(\widehat{f}) \neq 0$.

What to Submit:

- A proof that $\mathbb{P}\left[\widehat{R}_n(f) = 0\right] \leq \mathrm{e}^{-n\epsilon}$ for part (a).
- A proof that $\mathbb{P}\left[\exists f \in \mathcal{F} \text{ s.t. } R(f) > \epsilon \text{ and } \widehat{R}_n(f) = 0\right] \leq |\mathcal{F}|\mathrm{e}^{-\epsilon n}$ for part (b).
- A solution finding the minimum that satisfies the equation in part (c).
- A proof that $\widehat{R}_n(\widehat{f}) = 0 \quad \Longrightarrow \quad R(\widehat{f}) - R\left(f^*\right) \leq \frac{\log(|\mathcal{F}|/\delta)}{n}$ for part (d).

Answers:
Part a:

$$\begin{aligned}
\mathbb{P}[\widehat{R}_n(f) = 0] &= \mathbb{P}\left[\frac{1}{n}\sum_{i=1}^{n} 1(f(x_i) \neq y_i) = 0\right] \\
&= \prod_{i=1}^{n} \mathbb{P}\left[f(x_i) = y_i\right] \\
&= \prod_{i=1}^{n} 1 - \mathbb{P}\left[f(x_k) \neq y_i\right] \\
&= (1 - \mathbb{P}[f(x_k) \neq y_k])^n, \text{ where } k \in [1, \ldots, n] \\
&= (1 - \mathbb{E}_{X,Y}[1f(X) \neq Y])^n \\
&= (1 - R(f))^n
\end{aligned}$$

$$\boxed{\therefore \mathbb{P}[\widehat{R}_n(f) = 0] = (1 - R(f))^n \leq (1 - \epsilon)^n \leq \epsilon^{-n\epsilon}}$$

Part b:
For every f ∈ F, we can define $A_f = \{R(f) > \epsilon \text{ and } \widehat{R}_n(f) = 0\}$ This allows us to use the union bound inequality:

$$\begin{aligned}
\mathbb{P}\left[\in \mathcal{F} : R(f) > \epsilon and \widehat{R}_n(f) = 0\right] &\leq \mathbb{P}[A_1 \cup \cdots \cup A_k], \text{ where } A_k \in A_f \forall k \in [1, \ldots, n] \\
&\leq \sum_{f \in \mathcal{F}} \mathbb{P}(A_f) \\
&\leq |\mathcal{F}| e^{-en}
\end{aligned}$$

Part c:

$$\begin{aligned}
|\mathbb{F}| e^{-n\epsilon} &\leq \delta \\
\epsilon \leq -\frac{\log(\frac{\delta}{|\mathbb{F}|})}{n} &= \frac{\log(\frac{|\mathbb{F}|}{\delta})}{n} \\
\therefore \ \min \epsilon &= \frac{\log(\frac{|\mathbb{F}|}{\delta})}{n}
\end{aligned}$$

Part d:

$$\begin{aligned}
\mathbb{P}\left(\widehat{R}_n(f) = 0 \to R(\widehat{f}) - R(f^*) \leq \frac{1}{n}\log\frac{|\mathcal{F}|}{\delta}\right) &= \mathbb{P}\left(\widehat{R}_n(f) = 0 \text{ and } R(\widehat{f}) - R(f^*) > \frac{1}{n}\log\frac{|\mathcal{F}|}{\delta}\right) \\
&= \mathbb{P}\left(\widehat{R}_n(f) = 0 \text{ and } R(\widehat{f}) - R(f^*) > \epsilon^*\right), \text{ where } \epsilon^* = \frac{1}{n}\log\frac{|\mathcal{F}|}{\delta} \\
&\geq 1 - \mathbb{P}\left[\widehat{R}_n(f) = 0 \text{ and } R(\widehat{f}) > \epsilon^*\right] \\
&\geq 1 - \mathbb{P}\left[\widehat{R}_n(f) = 0 \text{ and } \exists f \in \mathcal{F} : R(\widehat{f}) > \epsilon^*\right] \\
&\geq 1 - |\mathcal{F}|e^{-\epsilon^* n} \\
&\geq 1 - \delta \quad \square
\end{aligned}$$

B2.

Perceptron

One of the oldest algorithms used in machine learning (from early 60's) is an online algorithm for learning a linear threshold function called the Perceptron Algorithm:

1. Start with the all-zeroes weight vector $\mathbf{w}_1 = 0$, and initialize t to 1. Also let's automatically scale all examples $\mathbf{x}$ to have (Euclidean) norm 1, since this doesn't affect which side of the plane they are on.
2. Given example $\mathbf{x}$, predict positive iff $\mathbf{w}_t \cdot \mathbf{x} > 0$.
3. On a mistake, update as follows:
   - Mistake on positive: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \mathbf{x}$.
   - Mistake on negative: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \mathbf{x}$.

   $t \leftarrow t + 1$.

If we make a mistake on a positive $\mathbf{x}$ we get $\mathbf{w}_{t+1} \cdot \mathbf{x} = (\mathbf{w}_t + \mathbf{x}) \cdot \mathbf{x} = \mathbf{w}_t \cdot \mathbf{x} + 1$, and similarly if we make a mistake on a negative $\mathbf{x}$ we have $\mathbf{w}_{t+1} \cdot \mathbf{x} = (\mathbf{w}_t - \mathbf{x}) \cdot \mathbf{x} = \mathbf{w}_t \cdot \mathbf{x} - 1$. So, in both cases we move closer (by 1) to the value we wanted. Here is a link if you are interested in more details.

Now consider the linear decision boundary for classification (labels in $\{-1, 1\}$) of the form $\mathbf{w} \cdot \mathbf{x} = 0$ (i.e., no offset). Now consider the following loss function evaluated at a data point $(\mathbf{x}, y)$ which is a variant on the hinge loss.

$$\ell((\mathbf{x}, y), \mathbf{w}) = \max\{0, -y(\mathbf{w} \cdot \mathbf{x})\}.$$

a. *[2 points]* Given a dataset of $(\mathbf{x}_i, y_i)$ pairs, write down a single step of subgradient descent with a step size of η if we are trying to minimize

$$\frac{1}{n} \sum_{i=1}^{n} \ell((\mathbf{x}_i, y_i), \mathbf{w})$$

for $\ell(\cdot)$ defined as above. That is, given a current iterate $\widetilde{\mathbf{w}}$ what is an expression for the next iterate?

b. *[2 points]* Use what you derived to argue that the Perceptron can be viewed as implementing SGD applied to the loss function just described (for what value of η)?

c. *[1 point]* Suppose your data was drawn i.i.d. and that there exists a $\mathbf{w}^*$ that separates the two classes perfectly. Provide an explanation for why hinge loss is generally preferred over the loss given above.

What to Submit:

- For (a) a single step of subgradient descent
- An answer to the question proposed in part (b).
- An explanation as described in part (c).

Answers:
Part a:
Let $\widetilde{\mathbf{w}} = \mathbf{w}_k$, then

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \left(\frac{1}{n} \sum_{i=1}^{n} \delta_p(l(x_i, y_i), w) \right)$$

$$\text{where } \delta_p(l(x_i, y_i), w) = \begin{cases} -y_i x_i, & \text{if } -y_i w^T x_i \geq 0 \\ 0, & \text{if } -y_i w^T x_i < 0 \end{cases}$$

Part b:
if we choose $\eta = 1$ and use a batch size of 1 in the Perceptron algorithm, part (a) becomes:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \cdot \delta_p(l(x_i, y_i), w)$$

$y_i x_i$ term will be added to the original weights if the label is incorrectly predicted. However, there will be no change to the weights for a correct prediction of the label.
$\therefore$ the Perception algorithm can be viewed as an SGD $\square$

Part c:
For the loss given in the problem statement, it does not update the weight if the labels are correctly predicted even though the prediction is only slightly above the linear decision boundaries. If we use hinge loss, it allows us to update the weights when the labels are correctly preditcted. this allows for a more robust model which has a higher margin of decision boundaries.

Coding

Introduction to PyTorch

A4. PyTorch is a great tool for developing, deploying and researching neural networks and other gradient-based algorithms. In this problem we will explore how this package is built, and re-implement some of its core components. Firstly start by reading `README.md` file provided in `intro_pytorch` subfolder. A lot of problem statements will overlap between here, readme's and comments in functions.

a. *[10 points]* You will start by implementing components of our own PyTorch modules. You can find these in folders: `layers`, `losses` and `optimizers`. Almost each file there should contain at least one problem function, including exact directions for what to achieve in this problem. Lastly, you should implement functions in `train.py` file.

b. *[5 points]* Next we will use the above module to perform hyperparameter search. Here we will also treat loss function as a hyper-parameter. However, because cross-entropy and MSE require different shapes we are going to use two different files: `crossentropy_search.py` and `mean_squared_error_search.py`. For each you will need to build and train (in provided order) 5 models:

   - Linear neural network (Single layer, no activation function)
   - NN with one hidden layer (2 units) and sigmoid activation function after the hidden layer
   - NN with one hidden layer (2 units) and ReLU activation function after the hidden layer
   - NN with two hidden layer (each with 2 units) and Sigmoid, ReLU activation functions after first and second hidden layers, respectively
   - NN with two hidden layer (each with 2 units) and ReLU, Sigmoid activation functions after first and second hidden layers, respectively

   For each loss function, submit a plot of losses from training and validation sets. All models should be on the same plot (10 lines per plot), with two plots total (1 for MSE, 1 for cross-entropy).

c. *[5 points]* For each loss function, report the best performing architecture (best performing is defined here as achieving the lowest validation loss at any point during the training), and plot it's guesses on test set. You should use function `plot_model_guesses` from `train.py` file. Lastly, report accuracy of that model on a test set.

d. *[3 points]* Is there a big gap in performance between between MSE and Cross-Entropy models? If so, explain why it occurred? If not explain why different loss functions achieve similar performance? Answer in 2-4 sentences.

What to Submit:

- **Part b:** 2 plots (one per loss function), with 10 lines each, showing both training and validation loss of each model. Make sure plots are titled, and have proper legends.
- **Part c:** Names of best performing models (i.e. descriptions of their architectures), and their accuracy on test set.
- **Part c:** 2 scatter plots (one per loss function), with predictions of best performing models on test set.
- **Part d:** 2-4 sentence written reponse to provided questions.
- **Code** on Gradescope through coding submission

Answers:
Part b:

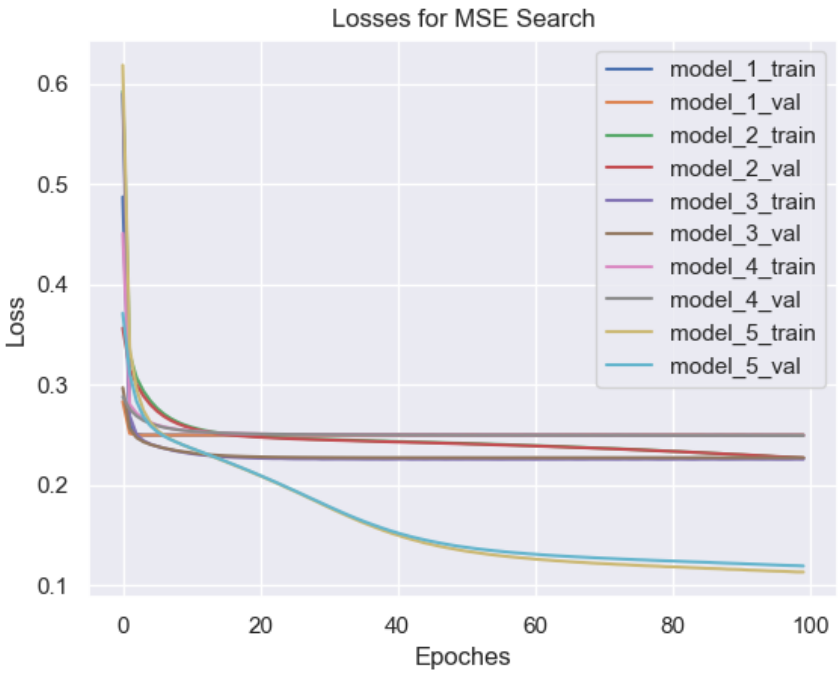

Figure 3: MSE loss function plot)

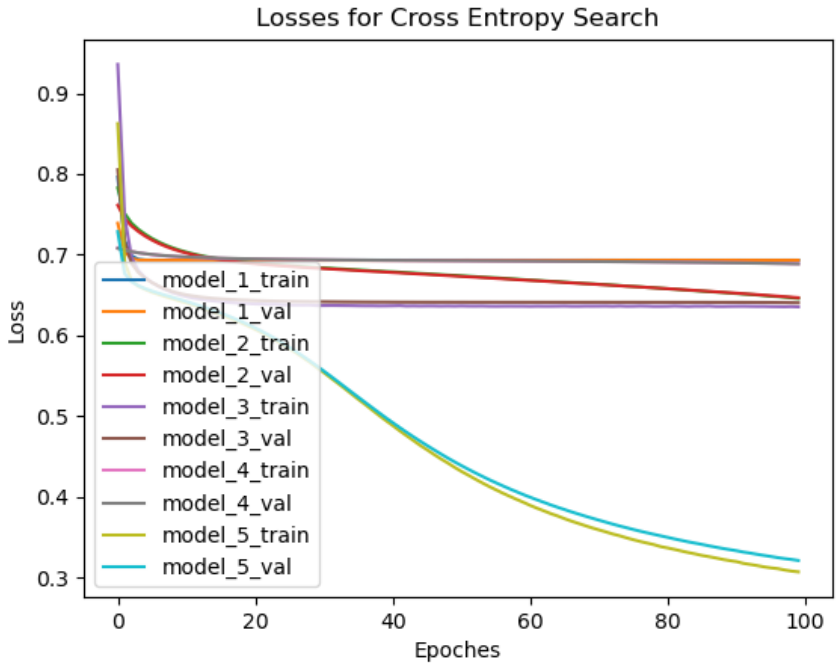

Figure 4: MSE loss function plot

Part c:
MSE best model: model5 (NN with two hidden layer and ReLU, Sigmoid) Accuracy = 88.30%
Cross entropy best model: model5 (NN with two hidden layer and ReLU, Sigmoid) Accuracy = 90.84%

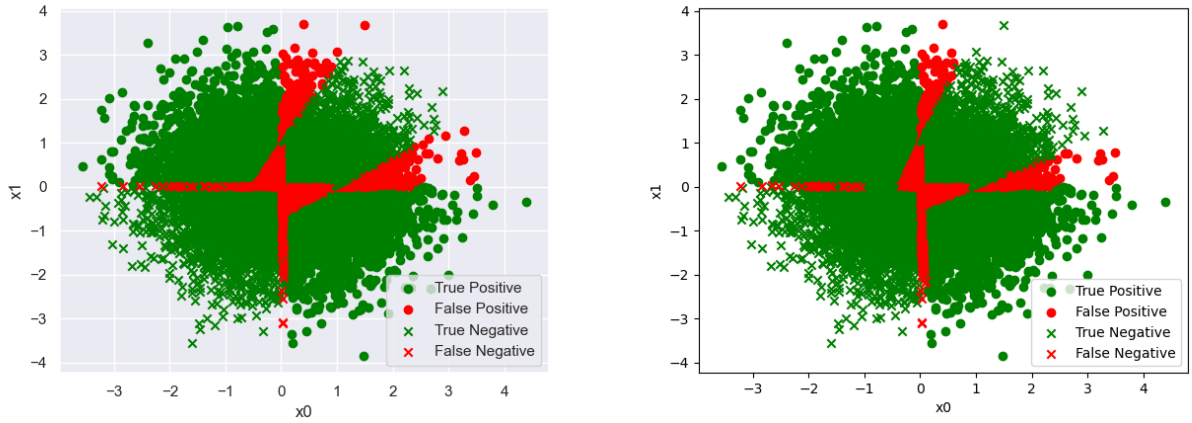

Figure 5: Left: MSE Model 5, Right: Cross Entropy Model 5

Part d:
No there isn't a big gap. Even though Cross Entropy performed slightly better than MSE, the two model performance are similar. Both MSE and Cross entropy loss functions are acceptable loss functions for use when the input data are between zeros and ones.

Neural Networks for MNIST

Resources

For next question you will use a lot of PyTorch. In Section materials (Week 6) there is a notebook that you might find useful. Additionally make use of PyTorch Documentation, when needed.
If you do not have access to GPU, you might find Google Colaboratory useful. It allows you to use a cloud GPU for free. To enable it make sure: "Runtime" -¿ "Change runtime type" -¿ "Hardware accelerator" is set to "GPU". When submitting please download and submit a `.py` version of your notebook.

A5. In previous homeworks, we used ridge regression for training a classifier for the MNIST data set. Similarly in previous homework, we used logistic regression to distinguish between the digits 2 and 7.
In this problem, we will use PyTorch to build a simple neural network classifier for MNIST to further improve our accuracy.

We will implement two different architectures: a shallow but wide network, and a narrow but deeper network. For both architectures, we use d to refer to the number of input features (in MNIST, $d = 28^2 = 784$), h_i to refer to the dimension of the i-th hidden layer and k for the number of target classes (in MNIST, $k = 10$). For the non-linear activation, use ReLU. Recall from lecture that

$$\text{ReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}.$$

Weight Initialization

Consider a weight matrix $W \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$. Note that here m refers to the input dimension and n to the output dimension of the transformation $x \mapsto Wx + b$. Define $\alpha = \frac{1}{\sqrt{m}}$. Initialize all your weight matrices and biases according to $\text{Unif}(-\alpha, \alpha)$.

Training

For this assignment, use the Adam optimizer from `torch.optim`. Adam is a more advanced form of gradient descent that combines momentum and learning rate scaling. It often converges faster than regular gradient descent in practice. You can use either Gradient Descent or any form of Stochastic Gradient Descent. Note that you are still using Adam, but might pass either the full data, a single datapoint or a batch of data to it. Use cross entropy for the loss function and ReLU for the non-linearity.

Implementing the Neural Networks

a. *[10 points]* Let $W_0 \in \mathbb{R}^{h \times d}$, $b_0 \in \mathbb{R}^h$, $W_1 \in \mathbb{R}^{k \times h}$, $b_1 \in \mathbb{R}^k$ and $\sigma(z)\colon \mathbb{R} \to \mathbb{R}$ some non-linear activation function applied element-wise. Given some $x \in \mathbb{R}^d$, the forward pass of the wide, shallow network can be formulated as:

$$\mathcal{F}_1(x) \coloneqq W_1\sigma(W_0x + b_0) + b_1$$

Use $h = 64$ for the number of hidden units and choose an appropriate learning rate. Train the network until it reaches 99% accuracy on the training data and provide a training plot (loss vs. epoch). Finally evaluate the model on the test data and report both the accuracy and the loss.

b. *[10 points]* Let $W_0 \in \mathbb{R}^{h_0 \times d}$, $b_0 \in \mathbb{R}^{h_0}$, $W_1 \in \mathbb{R}^{h_1 \times h_0}$, $b_1 \in \mathbb{R}^{h_1}$, $W_2 \in \mathbb{R}^{k \times h_1}$, $b_2 \in \mathbb{R}^k$ and $\sigma(z) : \mathbb{R} \to \mathbb{R}$ some non-linear activation function. Given some $x \in \mathbb{R}^d$, the forward pass of the network can be formulated as:

$$\mathcal{F}_2(x) \coloneqq W_2\sigma(W_1\sigma(W_0x + b_0) + b_1) + b_2$$

Use $h_0 = h_1 = 32$ and perform the same steps as in part a.

c. *[5 points]* Compute the total number of parameters of each network and report them. Then compare the number of parameters as well as the test accuracies the networks achieved. Is one of the approaches (wide, shallow vs. narrow, deeper) better than the other? Give an intuition for why or why not.

Using PyTorch: For your solution, you may not use any functionality from the `torch.nn` module except for `torch.nn.functional.relu`, `torch.nn.functional.cross_entropy`, `torch.nn.parameter.Parameter` and `torch.nn.Modul` You must implement the networks $\mathcal{F}_1$ and $\mathcal{F}_2$ from scratch. For starter code and a tutorial on PyTorch refer to the sections 6 and 7 material.

What to Submit:

- **Parts a-b:** Provide a plot of the training loss versus epoch. In addition evaluate the model trained on the test data and report the accuracy and loss.
- **Part c:** Report the number of parameters for the network trained in part (a) and for the network trained in part (b). Provide a comparison of the two networks as described in part in 1-2 sentences.
- **Code** on Gradescope through coding submission.

Answers:
Part a:

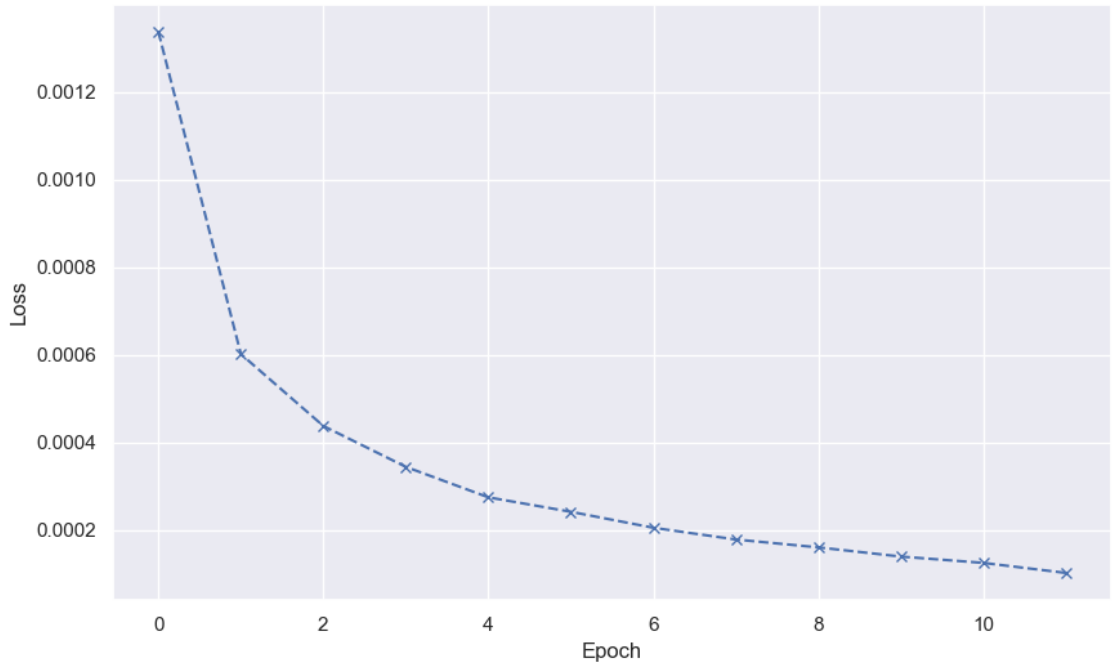

Figure 6: Training loss versus epoch for the two-layer model with test accuracy=0.975, loss=0.113)

Part b:

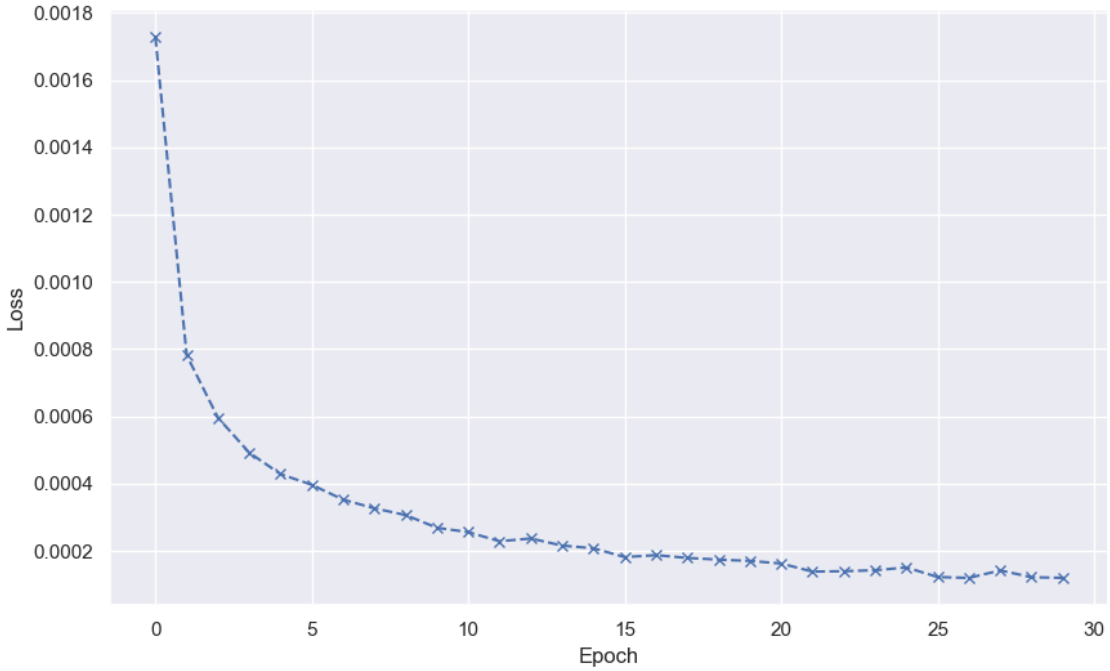

Figure 7: Training loss versus epoch for the three-layer model with test accuracy=0.9647, loss=0.201

Part c:
The two-layer model have 50890 parameters while the three-layer model have 26506 parameters. Both models have similar results in the training and test evaluation. Intuitively, the three layer model has significantly less parameter and would be better. For a single, more linear-like, neural network as in the two layer model, the differences will be closer in the project space than in the deep network and thus be unable to discern smaller different between numbers.

Administrative

A6.

a. *[2 points]* About how many hours did you spend on this homework? There is no right or wrong answer :)

Answer:
I spent about 30-40 hours over a span of a couple of days to finish the assignment.