ECE/CS 559: Neural Networks **Fall 2024**

Mock Final

Wednesday December 11, 2024 — There are **7 questions**; the actual exam will have 5.

Note: ***Please write your name on every page!*** *You are expected to work on your own on this exam. This exam is* ***closed notes*** *and no calculators are allowed. Other electronic devices or communication with others are also* ***not*** *allowed. Include your reasoning, not just the final answer. Be clear and concise. Watch your time. If stuck, move on then come back later.* ***Good luck!***

NAME: Solutions

— You may use this page and backs of pages for scratch work. —

Lecture 17

HW 7

1. Consider the following data points in $\mathbb{R}^2$: $\begin{bmatrix}-0.01\\1\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}0.01\\-1\end{bmatrix}, \begin{bmatrix}-1\\-1\end{bmatrix}$. ($x_1$ x_2 x_3 x_4)

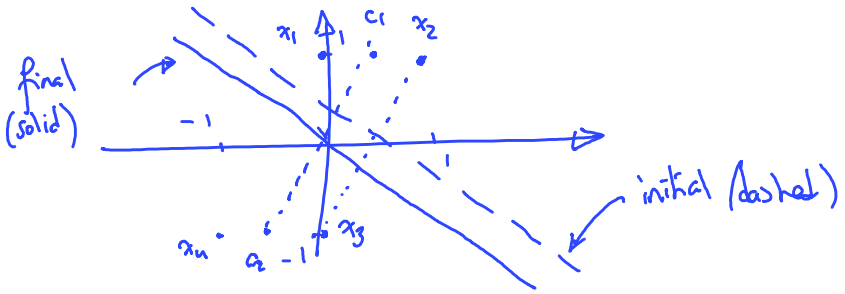
   (a) (3 pts) Perform, step by step, Lloyd's algorithm for k-means with $k = 2$ until convergence, starting with the second and third points as initial centers. Express the final centers clearly. Draw the initial and final Voronoi cells (along with the data points) on the same plot.

Initialization: $c_1 = \begin{bmatrix}1\\1\end{bmatrix}$ $c_2 = \begin{bmatrix}0.01\\-1\end{bmatrix}$

Step 1: $V_1 = \left\{\begin{bmatrix}-0.01\\1\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right\}$ (closer to c_1), $V_2 = \left\{\begin{bmatrix}0.01\\-1\end{bmatrix}, \begin{bmatrix}-1\\-1\end{bmatrix}\right\}$ (closer to c_2)

updated centers: $c_1 = \begin{bmatrix}0.99/2\\1\end{bmatrix}$ $c_2 = \begin{bmatrix}-0.99/2\\-1\end{bmatrix}$

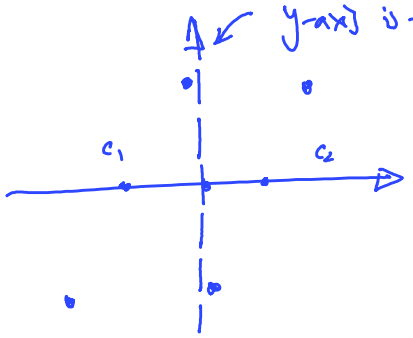
Step 2: V_1 the same, V_2 the same → algorithm stops



   (b) (2 pts) Suggest a different initialization that will not converge to the same final centers.

To not converge the same way we can use try and group the points differently in a way that stays consistent with proximity. In particular, group (x_1 & x_4) and (x_2 and x_3). Their centers are then $c_1 = \begin{bmatrix}-1.01/2\\0\end{bmatrix}$ and $c_2 = \begin{bmatrix}1.01/2\\0\end{bmatrix}$

If we initialize these, $V_1 = \{x_1, x_4\}$ and $V_2 = \{x_2, x_3\}$ thus the algorithm will not update and stop!



Lecture 18

2. Consider the following data points in $\mathbb{R}^2$: $\begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}0\\-1\end{bmatrix}, \begin{bmatrix}-1\\-1\end{bmatrix}$.

(a) (1 pt) State the competitive (winner-take-all) learning rule in clear mathematical notation. What "kind" of rule is this?

It is a Hebbian rule.

$i = \max_{i'(\text{neurons})} w_i^T x$

$z = w_i^T x$ (output of winner)

update: (only winner)

$w_i \leftarrow w_i + \eta z_i (x - w_i)$

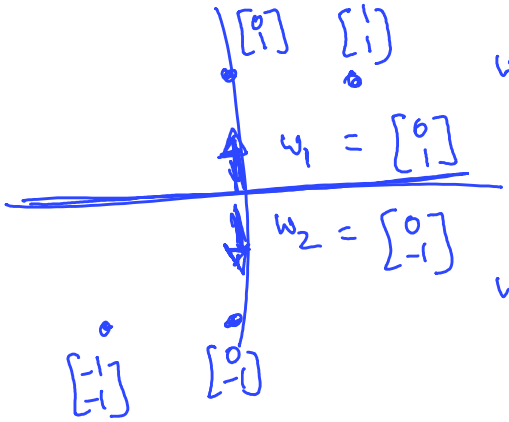
$= (1-\eta z_i) w_i + \eta z_i x$

(b) (4 pts) Perform this learning rule, with one difference: *after every update, round weights to the nearest integer*. Process the data in the given order, for 2 epochs. Use $\eta = \frac{1}{2}$ and initialize with weights $\begin{bmatrix}-1\\2\end{bmatrix}$ and $\begin{bmatrix}2\\-1\end{bmatrix}$. Give the clustering interpretation for the final weights.

(this means we have 2 neurons. will make this explicit)

w_1	w_2 before iteration	Epoch	Iteration	Data Point	winner info	update
$\begin{bmatrix}-1\\2\end{bmatrix}$	$\begin{bmatrix}2\\-1\end{bmatrix}$	Ep 1	It 1	$x=\begin{bmatrix}0\\1\end{bmatrix}$	winner $i=1$, $z=2$,	$w_1 = (1-\frac{1}{2}2)w_1 + \frac{1}{2}2\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$	$\begin{bmatrix}2\\-1\end{bmatrix}$	Ep 1	It 2	$x=\begin{bmatrix}1\\1\end{bmatrix}$	winner $i=1$, $z=1$, (tie, break either way)	$w_1 = (1-\frac{1}{2}1)w_1 + \frac{1}{2}\cdot 1\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\1\end{bmatrix} \xrightarrow{\text{round}} \begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$	$\begin{bmatrix}2\\-1\end{bmatrix}$	Ep 1	It 3	$x=\begin{bmatrix}0\\-1\end{bmatrix}$	winner $i=2$, $z=1$,	$w_2 = \frac{1}{2}w_2 + \frac{1}{2}\begin{bmatrix}0\\-1\end{bmatrix} = \frac{1}{2}\begin{bmatrix}2\\-2\end{bmatrix} = \begin{bmatrix}1\\-1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$	$\begin{bmatrix}1\\-1\end{bmatrix}$	Ep 1	It 4	$x=\begin{bmatrix}-1\\-1\end{bmatrix}$	winner $i=2$, $z=0$,	$w_2 = 1\,w_2 + 0\begin{bmatrix}-1\\-1\end{bmatrix} = \begin{bmatrix}1\\-1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$	$\begin{bmatrix}1\\-1\end{bmatrix}$	Ep 2	It 5	$x=\begin{bmatrix}0\\1\end{bmatrix}$	winner $i=1$	(no change, since $w_1 = x_1$)
$\begin{bmatrix}0\\1\end{bmatrix}$	$\begin{bmatrix}1\\-1\end{bmatrix}$	Ep 2	It 6	$x=\begin{bmatrix}1\\1\end{bmatrix}$	winner $i=1$, $z=1$,	$w_1 = \frac{1}{2}w_1 + \frac{1}{2}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\1\end{bmatrix} \xrightarrow{\text{round}} \begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$	$\begin{bmatrix}1\\-1\end{bmatrix}$	Ep 2	It 7	$x=\begin{bmatrix}0\\-1\end{bmatrix}$	winner $i=2$, $z=1$,	$w_2 = \frac{1}{2}w_2 + \frac{1}{2}\begin{bmatrix}0\\-1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\-1\end{bmatrix} \xrightarrow{\text{round}} \begin{bmatrix}0\\-1\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$	$\begin{bmatrix}0\\-1\end{bmatrix}$	Ep 2	It 8	$x=\begin{bmatrix}-1\\-1\end{bmatrix}$	winner $i=2$, $z=1$,	$w_2 = \frac{1}{2}w_2 + \frac{1}{2}\begin{bmatrix}-1\\-1\end{bmatrix} = \begin{bmatrix}\frac{1}{2}\\-1\end{bmatrix} \xrightarrow{\text{round}} \begin{bmatrix}0\\-1\end{bmatrix}$

Clustering:



winners with w_1

winner with w_2

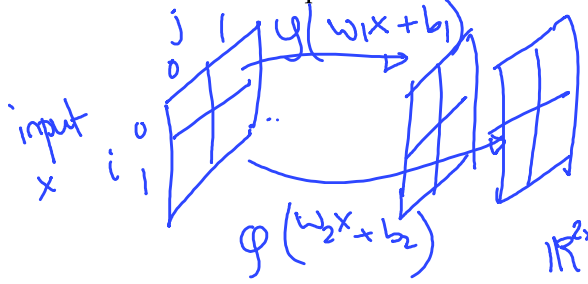
} 2 clusters

3. Consider a simple single-layer CNN, which takes as input a 2×2 image and produces two 2×2 channels, by applying kernels of width 1 with bias.

Lecture 16

HW 6

(a) (2 pts) Describe the parameters of this CNN and write the forward equations from the input to the two channels.



channel 1: w_1, b_1 (both $\mathbb{R}^1$)

channel 2: w_2, b_2 (both $\mathbb{R}^1$)

Forward equations:

$$f^1 = \varphi(w_1 x + b_1) \qquad \mathbb{R}^{2\times 2} \to \mathbb{R}^{2\times 2}, \ \mathbb{R}^1 \to \mathbb{R}^1$$

$$f^2 = \varphi(w_2 x + b_2)$$

(per coordinate)

(b) (3 pts) Write the backpropagation equations for this CNN layer, starting with the gradients of the loss at the output. (You don't need to know the loss.)

We assume we have $\frac{\partial \ell}{\partial f^1_{ij}}$ and $\frac{\partial \ell}{\partial f^2_{ij}}$

at each coordinate of f^1, f^2 $\quad i \in \{0,1\}, \ j \in \{0,1\}$

$$\frac{\partial \ell}{\partial w_1} = \frac{\partial}{\partial_1} \ell(f^1_{00}, f^1_{01}, f^1_{10}, f^1_{11})$$

$f^1_{00} \to w_1 x_{00} + b_1$, $f^1_{01} \to w_1 x_{01} + b_1$, $f^1_{10} \to w_1 x_{10} + b_1$, $f^1_{11} \to w_1 x_{11} + b_1$

$$= \sum_{ij} \frac{\partial \ell}{\partial f^1_{ij}} \cdot \frac{\partial f^1_{ij}}{\partial w_1} = \sum_{ij} \frac{\partial \ell}{\partial f^1_{ij}} \cdot x_{ij} \cdot \varphi'(w_1 x_{ij} + b_1)$$

Similarly:

$$\frac{\partial \ell}{\partial w_2} = \sum_{ij} \frac{\partial \ell}{\partial f^2_{ij}} \cdot x_{ij} \cdot \varphi'(w_2 x_{ij} + b_2)$$

$$\frac{\partial \ell}{\partial b_1} = \sum_{ij} \frac{\partial \ell}{\partial f^1_{ij}} \cdot 1 \cdot \varphi'(w_1 x_{ij} + b_1) \qquad \frac{\partial \ell}{\partial b_2} = \sum_{ij} \frac{\partial \ell}{\partial f^2_{ij}} \cdot 1 \cdot \varphi(w_2 x_{ij} + b_2)$$

Gaussian Autoencoder Model

Gaussian/Normal → mean, covariance

$z = \mathcal{N}(0, I)$ (prior)

$z|x = g(x) + \mathcal{N}(0, I) = \mathcal{N}(g(x), I)$ (encoder)

$\hat{x}|z = f(z) + \mathcal{N}(0, \Sigma) = \mathcal{N}(f(z), \Sigma)$ (decoder)

NAME: P_{prior} (forced, not optimized) P_{enc} (optimized via g) P_{dec} (optimized via f and Σ) QUESTION 4

Lecture 20, 21 HW7

4. Consider an autoencoder, with input x, embedding z, and output $\hat{x}$. Model the prior of the embedding to be a standard Gaussian vector. Let the encoder be a neural network parametrized by W producing an output $g(x; W)$ plus a standard Gaussian noise. Let the decoder be a neural network parametrized by W producing an output $f(z; W)$ plus a Gaussian with a *learnable covariance matrix* Σ. (W has both sets of parameters)

(a) (2.5 pts) Write the general form of the ELBO loss and specialize it to this autoencoder, simplifying it in the same way as the Gaussian autoencoder in class.

green = same as lecture

The only thing that changes is the decoder's distribution. It becomes:

$$P_{dec}(x|z) \propto \frac{1}{(\det\Sigma)^{1/2}} \exp\left[-\frac{1}{2}(x - f(z;w))^T \Sigma^{-1} (x - f(z;w))\right]$$

The general ELBO loss is $-\mathbb{E}\left[\log \frac{P_{prior}(z)\, P_{dec}(x|z)}{P_{enc}(z|x)}\right]$. Here, it becomes:

$$\mathbb{E}[-\log P_{prior}(z) + \log P_{enc}(z|x)] = \mathbb{E}\left[\text{constant} + \frac{1}{2}\|z\|^2 - \frac{1}{2}\|z - g(x)\|^2\right]$$

$$= \mathbb{E}\left[\text{constant} + \frac{1}{2}\|z\|^2 - \frac{1}{2}\|z\|^2 + z^T g(x) - \frac{1}{2}\|g(x)\|^2\right]$$

drop constants as they don't affect optimization

$$= \mathbb{E}\left[\mathbb{E}\left[z^T g(x) - \frac{1}{2}\|g(x)\|^2 \,|\, X\right]\right] = \mathbb{E}\left[g(x)^T g(x) - \frac{1}{2}\|g(x)\|^2\right] = \frac{1}{2}\mathbb{E}\left[g(x)^2\right] \quad (1)$$

since $\mathbb{E}[z|x] = g(x)$

ignore constants (don't affect optimization)

- $\mathbb{E}[-\log P_{dec}(x|z)] = \text{constant} + \frac{1}{2}\mathbb{E}\left[(x - f(z))^T \Sigma^{-1} (x - f(z))\right]$

$$= \frac{1}{2}\mathbb{E}\left[(x - f(g(x)+N))^T \Sigma^{-1} (x - f(g(x)+N))\right] \quad (2)$$

ELBO = (1) + (2)

In our code, z referred to $g(x)$.

(b) (2.5 pts) In your homework, you concatenated z and the reconstruction error $x - \hat{x}$. Explain why this won't work here. Then, use the fact that you can write $\Sigma^{-1} = M^T M$ to suggest a simple modification that would work.

- In class, we had ELBO = (ignoring constants and constant factors) $\mathbb{E}[\|g(x)\|^2]$ + $\mathbb{E}[\|x - \underbrace{f(g(x)+N)}_{\hat{x}}\|^2]$
  - $\|g(x)\|^2$: z in the code
- The first term is still the same, but the second is changed to:

$$\mathbb{E}\left[(x - \hat{x})^T \Sigma^{-1} (x - \hat{x})\right]$$

- By using the hint, we can reparametrize $\Sigma^{-1} = M^T M$. By substituting:

$$\mathbb{E}\left[(x - \hat{x})^T M^T M (x - \hat{x})\right]$$

- We recognize this as the square norm of $M(x - \hat{x})$.

$$\mathbb{E}\left[\|M(x - \hat{x})\|^2\right]$$

- So, if instead of giving $x - \hat{x}$ to the MSE Loss, we give $M(x - \hat{x})$, concatenated with z (in this case $g(x)$), then we get ELBO.
- If M is learnable ("requires_gradients" in Pytorch), part of the model, then M will be optimized along the way, as desired, via backpropagation.

In office hours there was a question of where exactly we use the idealized GAN loss.
(a) In Lecture 23 ("analyzing GANs"), we considered this same idealized GAN loss (with expectation instead of sum). We said that, because the discriminator is optimizing the cross-entropy between P(S|X) and P_disc(S|X), the best thing to do is to set D(x) = P_disc(S=1|X=x) = P(S=1|X=x). So, finding the optimal D is equivalent to find this posterior distribution.
(b) We also use the idealized loss by using the fact that with the optimal discriminator above, the generator is optimizing the Jensen-Shannon divergence between P(X) (the true/population) and P(G(Z)). This is almost a "distance" and it is smallest when zero, which is achieved by setting them to be equal. In other words, it is achieved by making G(Z) have the same distribution as X, as the question states.

Lectures 22, 23

5. Consider a GAN to generate a real-valued random variable X. Say the population of X is distributed in $[0,1]$ with density $f_X(x) = 2 - 2x$. Use a noise Z uniformly distributed in $[0,1]$. Use the idealized GAN loss we considered in class:

$$\underset{X,Z}{\mathbf{E}}\left[-\log D(X) - \log(1 - D(G(Z)))\right]$$

(a) (2.5 pts) Say the initial generator is just the identity function $G(z) = z$. What is the optimal discriminator in this case?

We saw in class that the optimal discriminator is:

$$D(x) = \mathbb{P}(S=1 \mid X=x) = \frac{\mathbb{P}(S=1)\cdot \mathbb{P}(X=x \mid S=1)}{\mathbb{P}(S=0)\,\mathbb{P}(X=x \mid S=0) + \mathbb{P}(S=1)\,\mathbb{P}(X=x \mid S=1)}$$

$\mathbb{P}(X=x \mid S=1) = f_X(x) = 2-2x$

$\mathbb{P}(X=x \mid S=0) = f_Z(x) = 1$ (since G is passing Z).

$\mathbb{P}(S=0) = \mathbb{P}(S=1) = \frac{1}{2}$

$$\Rightarrow D(x) = \frac{2-2x}{2-2x+1} = \frac{2-2x}{3-2x}$$

(b) (2.5 pts) What is the optimal generator G? That is, find G such that $G(Z)$ has the same distribution as X.

Lecture 19 ("derived distributions")

G is not necessarily unique, but let's choose it monotonically increasing. This, in particular, means that it's invertible: $x = G(z)$, $z = G^{-1}(x)$.

$$G(Z) \sim X \Rightarrow \mathbb{P}(G(Z) \le x) = \mathbb{P}(X \le x) = F_X(x) = \int_0^x f_X(t)\,dt$$

$$= \int_0^x 2-2t\,dt = 2t - t^2 \Big]_0^x = 2x - x^2$$

But also $\mathbb{P}(G(Z) \le x) = \mathbb{P}(Z \le \underbrace{G^{-1}(x)}_{z}) = F_Z(z) = z$ (uniform)

$$z = G^{-1}(x) = 2x - x^2 \iff z = 2G(z) - G(z)^2 \Rightarrow 1 - z = [1 - G(z)]^2$$

$$\Rightarrow G(z) = 1 - \sqrt{1-z}$$

(the other solution produces $G(z) \notin (0,1)$, which isn't acceptable!)

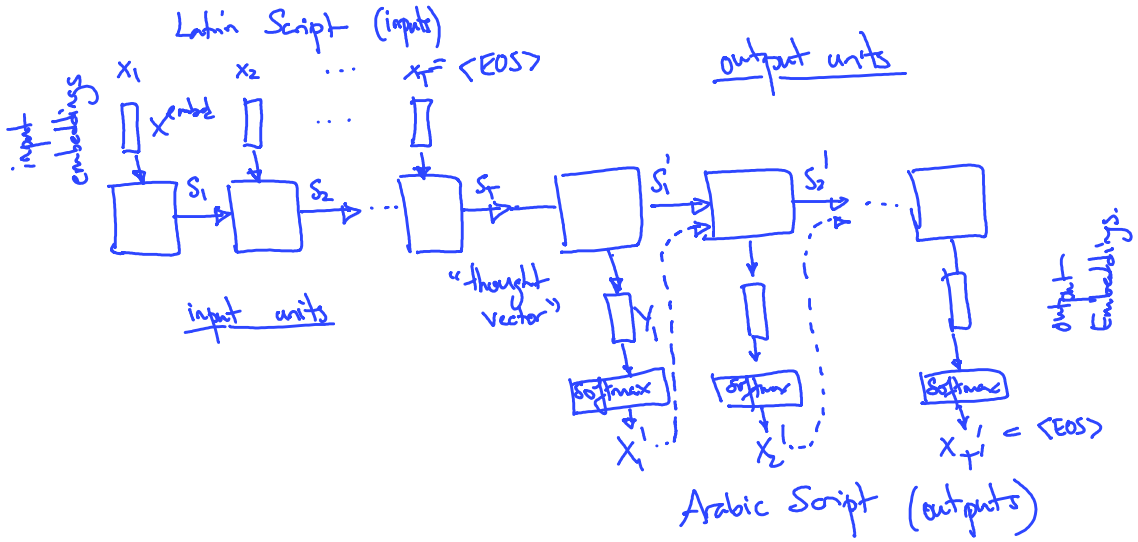
Lecture 24

6. Say we would like to build an ML model that takes as input Arabic return in Latin script, and outputs the same text in Arabic script. When written in Latin script, many Arabic letters require multiple characters to represent,

   (a) (1 pt) What kind of sequence model is most appropriate for this problem? Explain why.

   This is an asynchronous sequence to sequence model, because the output is a sequence and the input and output don't line up exactly.

   (b) (4 pts) Sketch an RNN architecture that you could use for this model. Draw a diagram with embeddings, units, inputs, and outputs clearly marked. Write a simple input-state-output equation for each unit.



- Input embedding $x_t^{embed} = Ax_t$, where x_t = one-hot
- Input units: $s_t = \varphi(W^x x_t + W^s s_{t-1} + b)$
- Output units $s'_t = \varphi(V^s s_{t-1} + V^x x'_{t-1} + c)$

  optional but will make the RNN aware of what's generated
- Output embedding $y'_t = \varphi(U s'_t + d)$
- Outputs: $x'_t \sim$ Sample from $\text{Softmax}(y_t)$.

$$= \frac{e^{y_t[j]}}{\sum_{j'} e^{y_t[j']}}$$

I should have named this differently to not confuse with values.

Lectures 25, 26

7. Say you have a graph $\mathcal{G} = (V, E)$ consisting of vertices $v \in V$ and edges $E \subset V^2$, the subset of pairs of vertices that are connected. You want to make a binary classification (e.g., is there a disturbance or not) that depends on the inputs x_v, sitting at each node.

   (a) (2 pts) You decide to design a *graph* self-attention, by remaining consistent with the graph at any given layer, i.e., uses only inputs connected to each vertex. Write the self-attention equations at a node v in clear mathematical notation.

The only thing we need to change is to make the attention over neighbors, i.e. v' s.t. $(v,v') \in E$

$Q_v = W^Q x_v$, $K_{v'} = W^K x_{v'}$, attention at v:

$$\alpha_{v,v'} = \underset{v':(v,v')\in E}{\text{Softmax}}\left(\frac{1}{\sqrt{k}} Q_v^T K_{v'}\right) = \begin{cases} 0 & ; \text{if } (v,v') \notin E \\ \frac{e^{\frac{1}{\sqrt{k}} Q_v^T k_{v'}}}{\sum_{v':(v,v')\in E} e^{\frac{1}{\sqrt{k}} Q_v^T K_{v'}}} & ; \text{if } (v,v') \in E \end{cases}$$

   (b) (3 pts) Explain (1) how you would build transformer layers using this modified self-attention, (2) whether or not the outputs of the last layer will only depend on neighboring inputs in the graph, and (3) how you would use the last layer outputs to make the binary classification with a clear description of the architecture and loss function.

(1) Calculate value at each node $V_{v'} = W^V x_{v'}$.
Combine according to attention: $S_v = \sum_{v':(v,v')\in E} \alpha_{v,v'} V_{v'}$
Possibly create multiple heads + concatenate
Pass through add+norm, FF, add+norm, get x_v^{new}.
Repeat, to obtain multiple layers.

(2) The output at higher layers can depend on nodes further than the immediate neighbors, e.g. on the second layer, they will depend on neighbors of neighbors. (information propagates)

(3) Say x_v^{last} are the outputs of the last layer.
One approach is to add a linear layer + sigmoid

- $f = \sigma\left(\sum_v W_v x_v^{out} + b\right) \in (0,1)$

We can then use binary cross entropy, if data is (x,y) binary

- $\ell(f,y) = y \log\frac{1}{f} + (1-y)\log\frac{1}{1-f}$.

ECE/CS 559: Neural Networks **Fall 2024**

Homework 7

Due: **Tuesday** November 26, 2024 (by 9pm, on Gradescope)

Note: *You may discuss these problems in groups. However, you must write up your own solutions and mention the names of the people in your group. Also, please do mention any solution manuals, online material, or other sources you used in a major way.*

Submit your code as a **two** `.py` **files**, one for each question, in Gradescope. Make sure that it **runs properly** and generates all the plots that you report on in the main submission.

1. **[K-Means using a Neural Network]**

   Load the MNIST dataset in PyTorch. Your goal is to cluster the images into $k = 10$ clusters, without relying on the label. Reuse your prior code, especially the loading, training, and testing aspects based on the PyTorch tutorial. Make the following significant changes.

   - Build a neural network that defines three elements in its constructor: a parameter `self.centers` that is a 10×784 tensor defined using `nn.Parameter` and referring to the centers matrix C, a flatten layer, and a softmax layer.
   - In the forward pass, do this:
     - Use the flatten layer on the input `x` and call the resulting variable `x0`, to keep access to it, and keep using `x` for subsequent stages.
     - Use the following single tensor equation to manually implement the 10 neurons of the first layer that we saw in class, for $i = 0, 1, \cdots, 9$, $xC_i^\intercal - \frac{1}{2}\|C_i\|^2$:
       ```
       x = torch.matmul(x0, self.centers.t())
             -0.5*torch.sum(self.centers**2,1).flatten()
       ```
     - Multiply the result by 20 then pass it through a softmax layer.
     - Reconstruct the input using a center, by multiplying the softmax output with the tensor C, from the right. [1-D tensors in PyTorch are row vectors.]
     - Return the error `x-x0`, where `x` is the reconstruction/center.
   - Use the MSE loss. To do this, when you call the loss, use one argument as the model's prediction (which is simply the reconstruction error) and let the other argument be a 0 tensor, the same size as the prediction batch (so that it computes compute $\|(x - x_0) - 0\|^2 = \|x - x_0\|^2$.)

     [The reason we're doing this is in order not to bother with manipulating batches ourselves. When you want the center, you can still get it, by adding the image to the reconstruction error: $(x - x_0) + x_0 = x$.]

- Modify the test loop to only evaluate and display loss, and not accuracy.

(a) Say we're using a batch size of $b = 64$. Recall that PyTorch by default *averages* the losses over each batch. Assume that, in each batch, roughly $\frac{1}{10}^{\text{th}}$ of the images are in each cluster, and assume that the softmax behaves likes an argmax. Explain why the update at iteration t looks like:

$$C_i(t) = \left(1 - \frac{\eta}{5}\right) C_i(t-1) + \frac{\eta}{5} \underbrace{\frac{1}{|\mathcal{V}_i(t)|} \sum_{x \in \mathcal{V}_i(t)} x}_{m_i(t)}$$

where $\mathcal{V}_i(t)$ are all images in the Voronoi cell of C_i, in this batch. $m_i(t)$ are the average centers in each cell *only in this batch*. Use $\eta = 4.5$. See the last page of the homework for an explanation of why this is a good idea for approximating Lloyd's algorithm, with the batch size we chose.

(b) Cheat, by initializing the cluster centers (`model.centers`) to individual images from the corresponding digit, i.e., set C_i to an image of digit i by saying:

`model.centers[i,:] =` a flattened version of the image.

Then, train your k-means neural network for 10 epochs and produce two outputs (pasted in your report):

i. All the cluster centers at the end of the training.

ii. A table of size 10×10 that contains for each row i and column j, the number of training images of digit i that were clustered in center j. (The sum of all entries should be 60,000.)

Discuss these results by commenting on what places k-means clustering is having trouble and what you think the reasons are behind it. [Hint: Use a data loader with batch size 1 to iterate over the data points individually.]

(c) Try to initialize the centers differently (e.g., sampling each coordinate uniformly randomly from $[0, 1]$), and try training the network again. Report on your choice of initialization and what kind of results you obtain (e.g., do you see something that you could reasonably call "mode collapse"?)

2. **[Autoencoder]**

Continue to use the same base code. Replace your neural network with an autoencoder. Let your encoder be, in one sequence, the same as the CNN from HW6 Q3 ending in a 10-dimensional representation `z`. For the decoder, use the following (roughly inverted) architecture in one sequence:

- A fully-connected stage with ReLU activations, going from 10 to 360, then from 360 to 720.

- Unflatten dimension 1 (we don't touch dimension 0, as it is the batch dimension) to `torch.size([20,6,6])`. This gives 20 channels of 6×6 images.
- Perform bicubic upsampling by a factor of 2.
- Use a (transpose) convolutional layer with 20 output channels, 4×4 filters, stride 2, and 1 output padding, followed by ReLU.
- Use a final (transpose) convolutional layer with 1 output channel, 4×4 filters, **stride 1**, **input padding 1**, and no output padding, followed by sigmoid.

Add batch normalization and dropout (with probability 0.1) after every ReLU activation, in both the encoder and decoder. Your final output should be 28×28.

In the forward pass, do this:

- Get `z` by passing the input image `x` through the encoder.
- Add standard Gaussian noise to `z` to make `z2`. To make generation easy, give the forward function an additional binary argument called `enc_mode`, set by default to 1, and use:

  ```
  z2 = enc_mode*z + (2-enc_mode)*torch.randn(z.shape)
  ```

  If we're in encoding mode (during training), this will act as additive noise. During generation mode (`enc_mode=0`), this will replace `z` itself with Gaussian noise. (Slightly different than in class: we're adding all noise before the decoder.)
- Get the output image `f` by passing `z2` through the encoder.
- Flatten `f` and return the concatenation of `z` (the embedding) and `f-x` (the reconstruction error), along dimension 1 (we don't touch dimension 0, as it is the batch dimension).

The reason we do this is because we're going to simulate the ELBO loss, and we saw that for Gaussian autoencoders it is simply the mean square of the embedding + reconstruction error. Therefore, you can simply use `MSELoss`. Set the batch size to 64 and the learning rate to 0.1.

(a) Draw a rough tensor block sketch of the encoder architecture and, to its right, the decoder architecture. Highlight which stage in the first is "inverted" by which stage in the second, by connecting them with brackets underneath.

(b) Train your autoencoder for 10 epochs. Visualize its performance by generating 20 random digits, by calling the model with the additional argument `enc_mode=0`. Note that you should eliminate the first 10 entries, which correspond to the embedding, by using a slicing, e.g., `[0,10:]`.

(c) Suggest modifications to the architecture, regularization, or training to obtain better results. Report your changes and visualize the results in the same way.

In Q1, using the minibatch approach has some advantages, including faster processing and avoiding local minima. The minibatch updates average centers *only within the minibatch*,

$$C_i(t) = \left(1 - \frac{\eta}{5}\right) C_i(t-1) + \frac{\eta}{5} \underbrace{\frac{1}{|\mathcal{V}_i(t)|} \sum_{x \in \mathcal{V}_i(t)} x}_{m_i(t)}$$

Lloyd's algorithm, in contrast, averages centers *across the whole data set*. This means we have some bias (offset) and variance (noisiness) due to using minibatches. For a fixed batch size, the choice of η creates a tradeoff between bias and variance.

To understand this, let's focus on when $\eta \in [0, 5]$ (bigger η will make learning unstable by moving *away* from prior centers.) We first explain the intuition of why bias and variance cannot be improved simultaneously, i.e., there's a tradeoff between them. Let's first think about the extremes. If $\eta = 0$, then we do not make updates. The variance is thus 0, but the bias is very large: we're very far from what we should be doing, which is averaging centers. If $\eta = 5$, we cancel everything from before and only use the last minibatch. This is the least amount of data we could use, and thus the variance is large, however m is doing what we should be doing, it is an unbiased estimate of what Lloyd's would have calculated.

How can we estimate bias? Because successive updates shift the centers, they cause m to drift away from Lloyd's average. By how much? The average pixel value is about $a = 0.23$. We'll assume that the latest average is the "freshest" and that the further back we look in batches, each coordinate of m could drift by as much as a times how far back we look, in average. Regarding noisiness, we can assume the noise level of each data point is constant. Recall that b is the minibatch size. The minibatch averaging reduces the variance by $\frac{1}{b}$, then averaging across minibatches will reduce the variance based on how the averaging is being done.

To simplify the notation, let's consider a single coordinate. Also, let $\alpha = \eta/5$. Thus, we have that $\alpha \in [0, 1]$, and . Let N be the training size and let T be the number of minibatches, so $T = N/b$.We have

$$\begin{aligned}
C(T) &= (1-\alpha)C(T-1) + \alpha m(T) \\
&= (1-\alpha)^2 C(T-2) + (1-\alpha)\alpha m(T-1) + \alpha m(T) \\
&= \ldots \\
&= (1-\alpha)^T C_0 + \sum_{t=1}^{T} \alpha (1-\alpha)^{T-t} m(t) \\
&\approx \sum_{t=1}^{T} \alpha (1-\alpha)^{T-t} m(t)
\end{aligned}$$

where the last approximation is true if $\alpha < 1$ and T is large. In our case with $b = 64$ and $N = 60,000$, T is indeed quite large.

Let M be the ideal average of all centers. We can model our assumptions of bias and variance roughly as:

$$m_t = M + a(T-t) + aZ_t/\sqrt{b}, \qquad Z_t \sim \mathcal{N}(0,1) \text{ (i.i.d.)}$$

We can now calculate our bias and variance:

$$\begin{aligned} \text{Bias} &= M - \mathbf{E}[C(T)] = M - \sum_{t=1}^{T} \alpha(1-\alpha)^{T-t}[M + a(T-t)] \\ &= M - \sum_{s=0}^{T-1} \alpha(1-\alpha)^{s}[M + as] \\ &\approx \frac{1-\alpha}{\alpha} a \end{aligned}$$

where the last expression takes $T \to \infty$, uses the fact that $\alpha(1-\alpha)^s$ is a the distribution of a geometric random variable with parameter α starting at 0. We use both the fact that this distribution sums to 1 (to cancel the M's) and that formula for its expected value is $(1-\alpha)/\alpha$.

$$\begin{aligned} \text{Variance} &= \sum_{t=1}^{T} \left[\alpha(1-\alpha)^{T-t}\right]^2 \frac{a^2}{b} \\ &= \frac{1}{b} \frac{\alpha^2}{1-(1-\alpha)^2} \sum_{s=0}^{T-1} (1-(1-\alpha)^2)((1-\alpha)^2)^s \\ &\approx \frac{a^2}{b} \frac{\alpha^2}{1-(1-\alpha)^2} \end{aligned}$$

where the last expression takes $T \to \infty$ and uses the fact that a geometric distribution sums to 1.

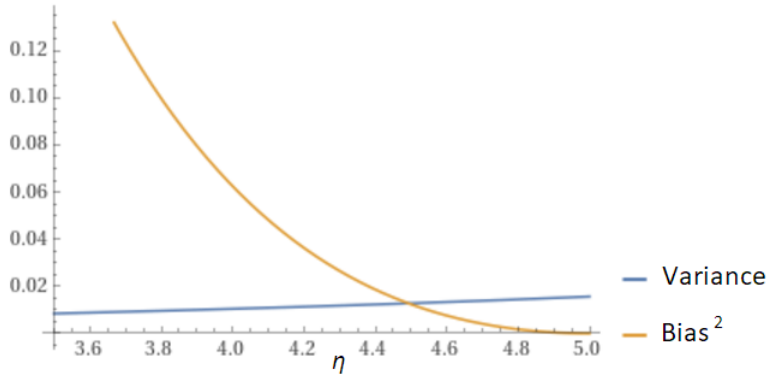
We now have estimates of what the repercussion is on bias and variance for a specific choice of η (or α), for a given batch size b and assuming average pixel value of a. As we observed with the extreme cases, we can see that with increasing α (or η), the bias goes down, but the variance goes up.

Bias and Variance combine as $\text{Bias}^2 + \text{Variance}$ to give the overall error, so to balance them out, we need to find a point where:

$$\left(\frac{1-\alpha}{\alpha} a\right)^2 \approx \frac{a^2}{b} \frac{\alpha^2}{1-(1-\alpha)^2}$$

(We could also try to minimize the sum, because we don't quite know the leading coefficients of each in this back-of-the-envelope calculation, those are both good heuristics.) Notice that because a affects both sides equally, it comes out in the wash. In terms of η, we need to find where the two curves touch, which we plot below:

$$\left(\frac{1-\frac{\eta}{5}}{\frac{\eta}{5}}\right)^2 \quad \text{and} \quad \frac{1}{64}\frac{(\frac{\eta}{5})^2}{1-(1-\frac{\eta}{5})^2}$$



From this, we see that the balance is achieved around $\eta = 4.5$, which is the choice suggested by the question. This also gives us an idea of how we should change η as we change b. For example, when you set $b = N = 60,000$, we should choose $\eta \approx 5$, reverting back to Lloyd's algorithm exactly. And if we set b smaller, we should make η smaller.

ECE/CS 559: Neural Networks **Fall 2024**

Homework 5

Due: **Tuesday** October 1, 2024 (by 9pm, on Gradescope)

Note: *You may discuss these problems in groups. However, you must write up your own solutions and mention the names of the people in your group. Also, please do mention any solution manuals, online material, or other sources you used in a major way.*

Submit your code as a **two** `.py` **files**, one for each question, in Gradescope. Make sure that it **runs properly** and generates all the plots that you report on in the main submission.

1. **[Gradient Descent]**

   Let $\mathrm{w} \in \mathbb{R}^2$. Consider the following function:

   $$R(\mathrm{w}) = 13w_1^2 - 10w_1w_2 + 4w_1 + 2w_2^2 - 2w_2 + 1$$

   (a) Find the gradient of R and solve for the optimality condition exactly.

   (b) Implement gradient descent. At each iteration, calculate and save the distance between the iterate and the optimal solution. Run this with $\eta = 0.02, 0.05$, and 0.1 and 500 iterations each. Plot distance vs. iteration in each case. Report it.

   (c) Is even larger η beneficial? Why or why not?

2. **[Backpropagation]**

   Revisit the dataset you generated in HW2 Q2. Imagine that a friend gives you the $1,000$ points generated with that neural network, tells you what the architecture looks like (but not the weights), then asks you to train a neural network that imitates theirs.

   (a) You want to use gradient descent, but know that you can't use a step function to do the training. Choose instead to use a sigmoid function φ with parameter $a = 5$. Write down φ and φ' explicitly and implement Python functions for each, that operate on `numpy` arrays.

   (b) Call the first layer weights W and its biases b, the second layer weights U and its bias c, and the output f. Let the hidden layer output be z. Write down the dimensions of all weights, biases, and variables, then write down the forward equations from x to v_z, from v_z to z, from z to v_f, and from v_f to f. Translate these into Python code.

(c) Use the squared loss $\ell(y, f) = (f - y)^2$. Since we are targeting binary outputs and φ limits values to $[0, 1]$, this is not that different from 0-1 loss. Start writing down the backward equations for a single data point, by first writing the expression for $\nabla_f \ell$ followed by the expression for δ_f. Then, write the backward equation from δ_f to δ_z, then from δ_z to δ_x. Use these along with x and z to compute the gradients $\nabla_W \ell$, $\nabla_b \ell$, $\nabla_U \ell$, and $\nabla_c \ell$. (Remember that these gradients should have the same dimensions as the respective weights and biases.) Translate these into Python code.

(d) In Python, initialize all your weights and biases with i.i.d. Gaussian$(\mu = 0, \sigma = 0.1)$ samples. Then, create an outer `epoch` loop and inner `i` loop ranging over the $1,000$ data points. For each data point (x, y), run the forward pass code from (b), then the backward pass code from (c). Then, perform gradient descent with that single point (this is a form of *stochastic gradient descent, SGD*): $W \leftarrow W - \eta \nabla_W \ell$, etc. Use $\eta = 0.01$ and perform 100 epochs. At the end of each epoch, calculate and save the MSE. At the end, plot and report the MSE vs. epoch curve.

(e) In HW2 Q2, you visualized the decision boundary of your friend's neural network. Visualize the decision boundary of yours, by plotting a 3D scatter plot of $(x_1, x_2, f(x_1, x_2))$ using the weights at the end of your training. Report the plot. (You can also visualize this during the training, to see how the boundary evolves.) You may find the following snippet useful.

```
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, y_predicted, color = "green")
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```

(f) Try at least three of the following hacks individually or simultaneously. Say what you tried, then report on the MSE vs. epoch curve, the decision boundary, a description of what changed, and your hypothesis as to why.

- Change η.
- Start η high but multiply by 0.9 if the MSE increases (or if it increases for T epochs, where you choose T).
- Choose a different a in φ.
- Change the number of epochs that you perform.
- Do a proper gradient descent (accumulate gradients, update only in the end of an epoch, zero-out gradients, and move to next epoch).
- Use minibatches (accumulate B gradients, update, zero-out, and continue).
- Change the architecture by including more or fewer hidden neurons.
- Change how you initialize the weights and biases, e.g. by setting them to 0 or choosing σ very small or large.

Question 1

(a) $R(w) = 13w_1^2 - 10w_1w_2 + 4w_1 + 2w_2^2 - 2w_2 + 1$

$$\frac{\partial R}{\partial w_1} = 26w_1 - 10w_2 + 4 = 0 \quad (1)$$

$$\frac{\partial R}{\partial w_2} = -10w_1 + 4w_2 - 2 = 0 \quad (2)$$

(optimality)

$$(2) \Rightarrow -25w_1 + 10w_2 - 5 = 0 \quad (3)$$

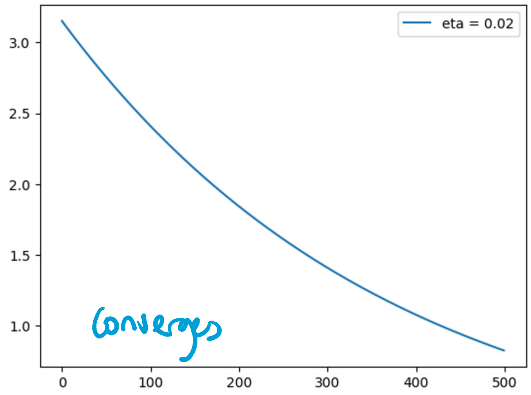
$$(1) + (3) \Rightarrow w_1 - 1 = 0 \Rightarrow w_1 = 1 \quad (4)$$

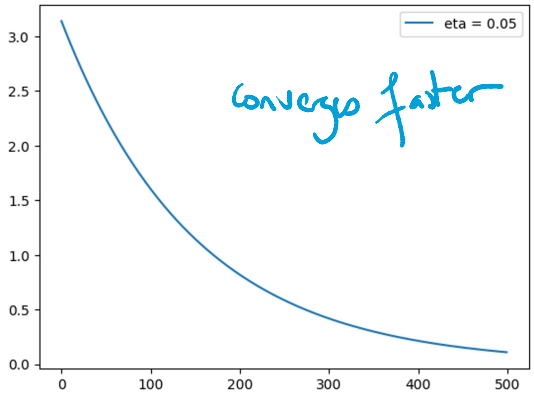
$$(2) + (4) \Rightarrow -10 + 4w_2 - 2 = 0 \Rightarrow w_2 = 3$$

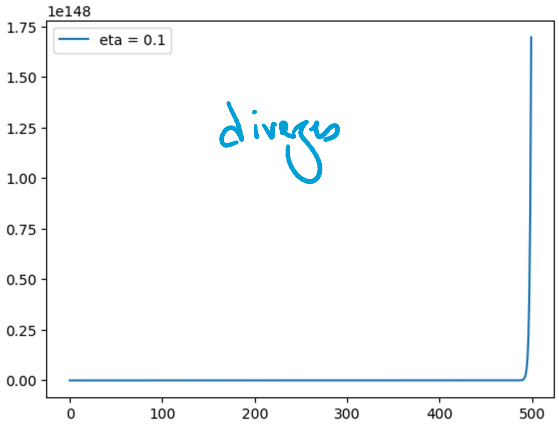
(b)

```
import numpy as np
import matplotlib.pyplot as plt
w_opt=[1,3]
number_of_epochs = 500
for eta in [0.02, 0.05, 0.1]:
  dist=np.zeros(number_of_epochs)
  w=np.zeros((2,))
  for i in range(number_of_epochs):
    grad = np.array([4+26*w[0]-10*w[1],-2-10*w[0]+4*w[1]])
    w = w - eta*grad
    dist[i]=np.linalg.norm(w-w_opt)
```

```
plt.plot(dist,label="eta = "+ str(eta))
plt.legend()
plt.show()
```







(c) Larger η is not always beneficial, because it would lead to instability (even if the gradients are exact, like here). This is because the 1st-order Taylor approximation is no longer very accurate.

Question 2

(a) $\varphi(v) = \frac{1}{1+e^{-5v}}$

$\varphi'(v) = 5\varphi(v)(1-\varphi(v))$

(b)

W	is	3×2	$(\dim(z) \times \dim(x))$
b	is	3	$(\dim(z))$
U	is	1×3	$(\dim(f) \times \dim(z))$
c	is	1	$(\dim(f))$

Forward equations!

$v_z = Wx + b \qquad z = \varphi(v_z)$

$v_f = Uz + c \qquad f = \varphi(v_f)$

```
v_z = np.matmul(W,x)+b
z = phi(v_z)
v_f = np.matmul(U,z)+c
f = phi(v_f)
```

(c) Backward equations for $\ell(y,f) = (y-f)^2$

$\nabla_f \ell = \frac{\partial \ell}{\partial f} = -2(y-f) \qquad \delta_f = \frac{\partial f}{\partial \ell} \cdot \varphi'(v_f)$

$$\nabla_f \ell = \frac{\partial \ell}{\partial f} = -2(y-f) \qquad \delta_f = \frac{\partial f}{\partial e} \cdot \varphi'(v_f)$$

$$\nabla_z \ell = U^T \delta_f \qquad \delta_z = \nabla_z \ell \cdot \varphi'(v_z)$$

$$\nabla_W \ell = \delta_z x^T \qquad \nabla_b = \delta_z$$

$$\nabla_U \ell = \delta_f z^T \qquad \nabla_c = \delta_f$$

```
dloss_df = -2*(y-f)
delta_f = dloss_df * phi_prime(v_f)
dloss_dz = np.matmul(np.transpose(U),delta_f)
delta_z = dloss_dz * phi_prime(v_z)
grad_W = np.matmul(delta_z, np.transpose(x))
grad_b = delta_z * 1
grad_U = np.matmul(delta_f, np.transpose(z))
grad_c = delta_f * 1
```

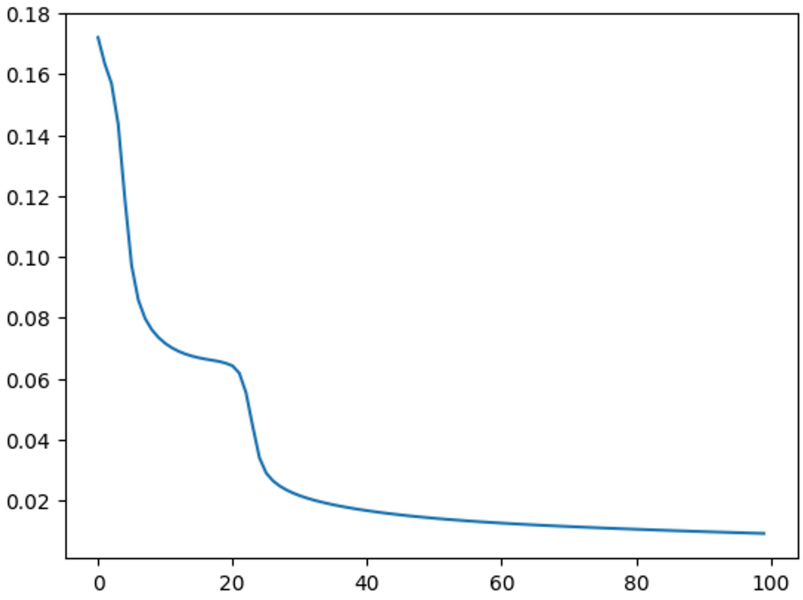
(d) Initialization:

```
sigma=0.1
W = np.random.randn(k,X.shape[0])*sigma
b = np.random.randn(k,1)*sigma
U = np.random.randn(1,k)*sigma
c = np.random.randn(1,1)*sigma
```

SGD:

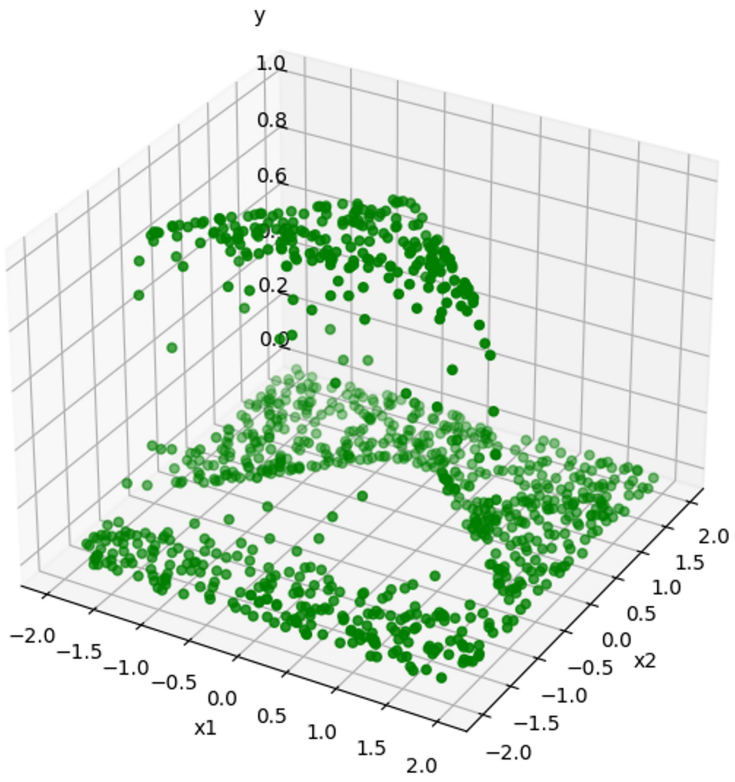
```
eta = 0.01
epochs = 100
risk = np.zeros(epochs)
```

```
for epoch in range(epochs):
  for i in range(N):
    x = X[:,i].reshape(x.shape)
    y = Y[0,i].reshape(f.shape)
    # Forward - Question 2(b)
    v_z = np.matmul(W,x)+b
    z = phi(v_z)
    v_f = np.matmul(U,z)+c
    f = phi(v_f)
    # Backward - Question 2(c)
    dloss_df = -2*(y-f)
    delta_f = dloss_df * phi_prime(v_f)
    dloss_dz = np.matmul(np.transpose(U),delta_f)
    delta_z = dloss_dz * phi_prime(v_z)
    grad_W = np.matmul(delta_z, np.transpose(x))
    grad_b = delta_z * 1
    grad_U = np.matmul(delta_f, np.transpose(z))
    grad_c = delta_f * 1
    # SGD
    W = W - eta * grad_W
    b = b - eta * grad_b
    U = U - eta * grad_U
    c = c - eta * grad_c
  Y_predicted = phi(np.matmul(U,phi(np.matmul(W,X)+b))+c)
  risk[epoch] = np.sum((Y-Y_predicted)**2)/N
plt.plot(risk)
plt.show()
```



(e) Visualization

```
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, Y_predicted, color = "green")
ax.set_xlabel('x1'); ax.set_ylabel('x2'); ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```



(e) This question is a bit open-ended.

- Changing η has its usual pitfalls. An adaptive η works remarkably well (many methods like Adam and Adagrad do this by monitoring the

Adam and Adagrad do this by monitoring the gradients themselves.)

- Minibatch is helpful to smooth out the gradients of SGD. However, we shouldn't simply accumulate the gradients, since then ther variance increases. Instead, we need to average across the minibatch.

- Initializing weights at 0 slows the convergence considerably, even halting it (by getting stuck in local minima.) Local minima are also an issue with very large σ for the random initialization.

- Making a big in φ makes it "less differentiable" do more unstable, because of both large (at the transition) and very small (at saturation) gradients. However,

small (at saturation) gradients. However, the end results are crisper, as long as we can stabilize (adaptive η + minibatch help.)

The following code does:

- Sets $\alpha = 10$ in φ.
- Adaptive η, starting at $\eta = 0.1$
- Minibatch with batch size 5, and with averaged gradients.

```
def phi(v):
  return 1./(1.+np.exp(-10*v))
def phi_prime(v):
  return 10*phi(v)*(1-phi(v))
sigma=0.1
# Initialization
W = np.random.randn(k,X.shape[0])*sigma
b = np.random.randn(k,1)*sigma
U = np.random.randn(1,k)*sigma
c = np.random.randn(1,1)*sigma
# W=W*0 # Zero initialization
# b=b*0
# U=U*0
# c=c*0
eta = 0.1
epochs = 100
batch_size = 5
risk = np.zeros(epochs)
grad_W = 0*W
grad_b = 0*b
grad_U = 0*U
grad_c = 0*c
for epoch in range(epochs):
  for i in range(N):
    x = X[:,i].reshape(x.shape)
    y = Y[0,i].reshape(f.shape)
    # Forward
    v_z = np.matmul(W,x)+b
    z = phi(v_z)
```

```
    v_f = np.matmul(U,z)+c
    f = phi(v_f)
    # Backward
    dloss_df = -2*(y-f)
    delta_f = dloss_df * phi_prime(v_f)
    dloss_dz = np.matmul(np.transpose(U),delta_f)
    delta_z = dloss_dz * phi_prime(v_z)
    grad_W += np.matmul(delta_z, np.transpose(x))
    grad_b += delta_z * 1
    grad_U += np.matmul(delta_f, np.transpose(z))
    grad_c += delta_f * 1
    # Minibatch SGD
    if (i+1)%batch_size==0:
      W = W - eta * grad_W/batch_size # Averaged
      b = b - eta * grad_b/batch_size # gradients
      U = U - eta * grad_U/batch_size
      c = c - eta * grad_c/batch_size
      grad_W = 0*W # Zeroing out the gradients
      grad_b = 0*b
      grad_U = 0*U
      grad_c = 0*c
  Y_predicted = phi(np.matmul(U,phi(np.matmul(W,X)+b))+c)
  risk[epoch] = np.sum((Y-Y_predicted)**2)/N
  if epoch > 1:
    if risk[epoch]>risk[epoch-1]:
      eta = eta*0.9
plt.plot(risk)
plt.show()
# Visualization
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, Y_predicted, color = "green")
ax.set_xlabel('x1'); ax.set_ylabel('x2'); ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```