Homework 3, Machine Learning, Fall 2024

***IMPORTANT* Homework Submission Instructions**

1. For all coding components, complete the solutions with a Jupyter notebook/Google Colab, and export the notebook (including both code and outputs) into a PDF file. Concatenate the theory solutions with the coding solutions into **a single** PDF file which you will submit to Gradescope.
2. All solutions should be typed.
3. Gradescope will prompt you to label the pages for each question. Please do this carefully.
4. You must show work or give an explanation for every question. "Yes" and "No" are not sufficient answers. Always show the steps of your calculations. Only partial credit can be given if you do not explain your answer.
5. Failure to adhere to the above submission format may result in penalties.

As a reminder, don't wait until the last minute to ask for help, because the person you need to speak with might be dealing with many other students close to the deadline. Also, the teaching staff has been instructed that it is not mandatory for them to answer questions on the day the homework is due. So get your questions in early!

There is no collaboration on homework. You must do your own work; we feel this is the best way to learn. If you get stuck, you are welcome to discuss conceptual issues with your classmates or the TAs but you may not show your classmates any part of your code or solutions or proof steps on paper. Do not ask another classmate to perform coding for you or to show you the answer to the problem. You can look at reference material on the Internet such as ChatGPT, but don't look at that unless you are really stuck, and make sure your answer is complete - don't write "Because I found a theorem that says the answer is X". You must show your complete work for each question. You are required to state what references you used (if you used an outside source). **Please copy the following statement at the top of your assignment file**:

This assignment represents my own work. I did not work on this assignment with others. All coding was done by myself.

1 Maximum Likelihood Learning for Multi-Output Regression (Theory) (Varun) - 25 pts

A data-scientist has collected a regression dataset comprising N independent scalar inputs $\{x_n\}_{n=1}^N$ and N scalar outputs $\{y_n\}_{n=1}^N$. The goal is to predict y from x, assuming that the data are generated according to a very simple linear model $y_n = ax_n + \epsilon_n$ ($a \in \mathbb{R}$ is a scalar value).
The data-scientist also has access to a second set of outputs $\{z_n\}_{n=1}^N$ generated according to $z_n = x_n + \epsilon'_n$. Assume that the noise variables ϵ_n and ϵ'_n are known to be zero-mean Gaussian random variables, and that they are correlated according to a joint Gaussian distribution with mean $\mathbf{0}$ and covariance Σ:

$$p\left(\begin{bmatrix}\epsilon_n \\ \epsilon'_n\end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix}\epsilon_n \\ \epsilon'_n\end{bmatrix}; \mathbf{0}, \Sigma\right), \tag{1}$$

where $\Sigma = \begin{bmatrix}1 & \beta \\ \beta & 1\end{bmatrix}$. In this problem, we will investigate how the correlation between the noise variables influences the data-scientist's estimate for a.

Note: The probability density of a multivariate Gaussian distribution of mean μ and covariance Σ is given by:

$$\mathcal{N}(\mathbf{x}; \mu, \Sigma) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} \exp\Big(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\Big) \tag{2}$$

(a) Provide an expression for the joint log-likelihood of the outputs given a, the scalar inputs x_n, and β, i.e. $p\Big(\{y_n\}_{n=1}^N, \{z_n\}_{n=1}^N \Big| a, \{x_n\}_{n=1}^N, \beta\Big)$.
Hint: We can write the noise variables in terms of x_n, y_n, and z_n:

$$\epsilon_n = y_n - ax_n \tag{3}$$
$$\epsilon'_n = z_n - x_n \tag{4}$$

(b) Compute the maximum likelihood estimate for a.

2 Regression, Regularization (Theory) (Matthew) - 25 pts

Assume we are given a dataset D with n samples, d features, and continuous outputs so that each $(\mathbf{x}_i, y_i) \in D$ is an input-output pair with $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. Consider a linear model parameterized by $\theta := (\theta_1, \ldots, \theta_d) \in \mathbb{R}^d$ and a bias term θ_0 of the form:

$$\hat{y}_\theta(\mathbf{x}) = \theta_0 + \sum_{j=1}^{d} \theta_j x_j \tag{5}$$

(a) For this problem, we will work with the mean squared error (MSE) function given by:

$$L_D(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}_\theta(\mathbf{x}_i) - y_i\right)^2 \tag{6}$$

Suppose that Gaussian noise $\epsilon_i \sim \mathcal{N}(\mathbf{0}, \sigma^2 I_d)$ (I_d is the $d \times d$ identity matrix) is added independently to each input to form a new dataset $\tilde{D}$. That is, for $\tilde{\mathbf{x}}_i \in \tilde{D}$, for each feature $j \in \{1, \ldots, d\}$ we have $\epsilon_{i,j} \sim \mathcal{N}(0, \sigma^2)$ and thus

$$\tilde{\mathbf{x}}_i = (x_{i,1} + \epsilon_{i,1}, \ldots, x_{i,d} + \epsilon_{i,d})^T \tag{7}$$

i. Show that, if we average over possible draws of noise ϵ_i, the quadratic error function over the noisy data simplifies to the original loss L_D plus an additional term which penalizes the ℓ_2 norm of the parameters θ (not including the bias term).

ii. What is the regularization parameter as a function of σ, and what does that tell us about the effect of noise on the optimal model?

(b) Instead of the regularization occurring due to noise in the dataset, we can also induce regularization by placing a prior probability distribution on the model parameters. Instead of an explicit loss, now suppose that we model the labels Y_i as being generated by the sum of the model output and a Gaussian noise $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, so that $Y_i = \theta^T \mathbf{x}_i + \epsilon_i$. Furthermore, suppose that we have a prior belief that the model parameters θ are drawn from a Laplace distribution with parameter b so that $P(\theta_j) = \frac{1}{2b} \exp(-\frac{|\theta_j|}{b})$.

i. Show that the MAP (Maximum A Posteriori) Estimation[1] of the parameters θ naturally incorporates the effect of L1 regularization.

ii. What is the induced loss function, and how does the strength of the regularization parameter change with b?

The general formula for MAP estimation is:

$$MAP = \arg\max_{\theta} \left[P(Y \mid \theta, X) P(\theta)\right]$$

where $P(Y \mid \theta, X)$ is the likelihood function of the dataset label Y given the model parameters θ and the input data X, and $P(\theta)$ is the prior probability of the parameters.

[1] MAP Estimation combines prior knowledge (represented by the prior probability) with the evidence from observed data (represented by the likelihood function) to provide an estimate of the model parameters. The goal of MAP estimation is to find the parameter values that maximize the posterior probability.

3 Transformers Implementation for GPTs (Coding) (Muchang, Eric) 15 pts

Generative Pretrained Transformers (GPTs) have recently gained prominence as the state-of-the-art architecture for many NLP tasks. If you've ever used ChatGPT, as the name suggests, you've used a GPT model. In the following question, you will develop a `Pytorch` implementation of the multi-head attention layer of a lightweight GPT model. Using a preprocessed dataset of Shakespeare's plays with a character-level tokenization, you'll then train the model to generate novel Shakespearean text.

Before beginning to code, you will need to set up your environment. First, git clone this repository and create a `conda` environment with `numpy` (`conda install numpy`) and `torch` (`pip install torch`) installed. Make sure to activate it before you start.

Based on our experiments, the model *should* be lightweight enough to be trained on a personal machine. As another option, you may also use your Google Cloud credits to reserve a VM on Google Cloud Platform (GCP) for this question. See the announcement on Ed for more details on obtaining credits. Alternatively, you can also reserve a VM through Duke OIT, though it may be less powerful than a GCP VM.

To briefly go over this repository,

1. `model.py` defines the actual transformer model that we will train. Note that a transformer model consists of multiple *self-attention modules* which each consist of a *self-attention layer* plus a simple *multilayer perceptron* (MLP), including nonlinearities and normalization layers, which help with training. The `Block` class composes both the `SelfAttention` class and `MLP` class for another layer of abstraction.

2. These decoder layer modules are stacked on top of each other within the transformer model, which you can see in the `transformer` attribute of the `Transformer` class.

3. `train.py` is a script that will train the model.

4. `sample.py` is a script that will generate text from the model.

To set up some notation, if the embedding dimension is C, and the sequence length is L, then a sequence of text inputs can be written as a matrix of shape $L \times C$, also called a *rank 2 tensor*.[2]

To parallelize these operations, PyTorch supports *batching* of inputs, allowing you to input a collection of B matrices of shape $L \times C$, stored in a *rank 3 tensor* of shape $B \times L \times C$, and performing the same operations on every single matrix at once. This also helps with parallelizing matrix multiplication (and other operatons) over higher order tensors. For example, if A has shape $B_1 \times \ldots \times B_k \times L \times M$ and B has shape $B_1 \times \ldots \times B_k \times M \times N$, then AB has shape $B_1 \times \ldots \times B_k \times L \times N$. We are essentially matrix multiplying over the last two ranks. You should get familiar with this concept of batching, as it will save you a lot of time and effort compared to using for loops. *We do not recommend using for loops, as both the code tends to get messy and the runtime will be significantly slower.*

(a) To begin, we will first perform a quick sanity check to ensure your environment has been set up correctly: run the command `python sample.py` from the base of the repository to generate a sample of text. You should notice that the results just look like a random sequence of characters. There are two reasons for this: we haven't implemented the attention layer, nor have we trained the model.

(b) Let's focus on the attention layer first. We will do this step by step by implementing the `forward` method in `SelfAttention` class in `model.py`. Recall that the formula for scaled-dot product single-headed attention is

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{C}}\right)V \tag{8}$$

[2] A scalar is a rank 0 tensor. A vector, which is an array of scalars, is a rank 1 tensor. A matrix, which is an array of vectors, is a rank 2 tensors. So on and so forth.

where Q, K, V are the query, key, value matrices, and the softmax operation is done over the *rows* of the input matrix. Each of these steps should not take more than 10 lines of code each. We have provided an outline in the codebase as a guide, but you do not necessarily have to follow it.

i. The query, key, and value matrices are computed by `self.c_attn`, which is a linear layer ℓ that maps each token $\mathbf{x}$ to its associated key, query, and value vectors through the map

$$\ell(\mathbf{x}; \mathbf{A}, \mathbf{b}) = \mathbf{A}\mathbf{x} + \mathbf{b} = \begin{pmatrix} \mathbf{q} \\ \mathbf{k} \\ \mathbf{v} \end{pmatrix} \tag{9}$$

where $\mathbf{x} \in \mathbb{R}^C$, $\mathbf{A} \in \mathbb{R}^{3C \times C}$, and $\mathbf{b} \in \mathbb{R}^{3C}$. This is batched over the entire sequence length L and over the B batches for each sequence, so our output wouldn't be simply in $\mathbb{R}^{3C}$, but rather of size $B \times L \times 3C$. You want to take this output matrix of parameters and partition it into the query, key, value matrices, each of size $B \times L \times C$.

ii. We want to implement *multihead* attention by further partitioning each matrix into H submatrices, where H is the number of heads. H is available as `self.n_head` within the class.

$$\begin{aligned} Q &\mapsto (Q_1, \ldots, Q_H) \\ K &\mapsto (K_1, \ldots, K_H) \\ V &\mapsto (V_1, \ldots, V_H) \end{aligned}$$

To implement multihead attention, we want to add an extra rank to change the shape of each matrix from $(B \times L \times C) \mapsto (B \times H \times L \times C/H)$. Note that H must evenly divide C since we want to evenly partition the embedding dimension, so C/H will always be an integer. The same logic should be looped through the K and V matrices as well. A visual is provided for some intuition.

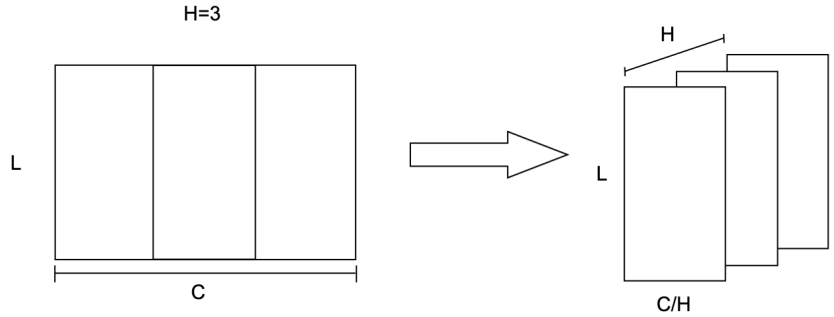

Figure 1: A visualization of how a single element of a batch, of shape $L \times C$, should be reshaped to shape $H \times L \times C/H$. You want to add an extra dimension by stacking these submatrices on top of each other. In the end, we should satisfy `Q[i, j, :, :].shape = (T, C // H)`.

iii. Now we must actually implement the attention operation described in the equation above. First, implement only the operation

$$\frac{Q_i K_i^T}{\sqrt{C/H)}} \tag{10}$$

for all $i = 1, \ldots H$ over all batches. Once this is done, your result should be of shape $B \times H \times L \times L$.

iv. We also need a masking step since this is a decoder architecture. We have provided a `self.mask` attribute consisting of a lower triangular matrix of 1s. Use this to mask your output so that future tokens are ignored in the attention score. You might find the `torch.Tensor.masked_fill()` operation helpful.

v. Then, we can apply the softmax operation on the rows over each head and batch. After this should be a dropout layer, which we have implemented for you. Finally, we can multiply by the value matrix V to get the head outputs y. After this step, the assertion statement that we have implemented on `y.shape` should hold true.

(c) Train this model using the command `python train.py` and plot the loss curve over time. Note that training for 2000 epochs on Google Cloud is estimated to take between 5-20 minutes. You should allocate your time accordingly for training.

(d) Generate a new brief Shakespearean text using the command `python sample.py` and post your output. Now you have learned the details of a minimal transformer architecture.

4 Attention and Convolution (Coding) (Zack) - 20 pts

In this question, we will use the PyTorch framework to explore some properties of self-attention and some connections between self-attention and convolution[3]. First, we define the notion of a "similarity metric", which can be thought of as the opposite of a distance metric. Formally, we call a function $s : \mathbb{R}^{d\times d} \to \mathbb{R}$ a similarity metric if, for any $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^d$:

1. s is bounded above by some value b: $s(\mathbf{x}, \mathbf{y}) \leq b$.
2. The similarity between an object and itself is the maximum similarity possible: $s(\mathbf{x}, \mathbf{x}) = b$
3. The similarity between two distinct objects is not the maximum similarity: $\mathbf{x} \neq \mathbf{y} \implies s(\mathbf{x}, \mathbf{y}) < b$
4. Similarity is symmetric: $s(\mathbf{x}, \mathbf{y}) = s(\mathbf{y}, \mathbf{x})$

(a) When learning about convolution, you may have learned that it is similar to template matching, where we find the similarity between a template and an image at each location. In the first part of this problem, we will use the PyTorch package to visually explore whether convolution is computing a similarity function. First, recall the definition of convolution. For an input matrix $\mathbf{Z} \in \mathbb{R}^{d_{\text{in}}\times h_{\text{in}}\times w_{\text{in}}}$ and a learned tensor of d_{out} convolutional filters $\mathbf{K} \in \mathbb{R}^{d_{\text{out}}\times d_{\text{in}}\times h_{\text{kernel}}\times w_{\text{kernel}}}$, the output of a simple 2 dimensional convolution at position h, w for filter j is:

$$(\mathbf{Z} * \mathbf{K})_{j,h,w} = \sum_{a=1}^{d_{\text{in}}} \sum_{b=1}^{h_{\text{kernel}}} \sum_{c=1}^{w_{\text{kernel}}} k_{j,a,b,c} z_{a,h+b-\lfloor h_{\text{kernel}/2}\rfloor, w+c-\lfloor h_{\text{kernel}/2}\rfloor}, \tag{11}$$

where the $*$ operator denotes convolving the first tensor with the second. For simplicity, in this problem we will consider convolution with $h_{\text{kernel}} = w_{\text{kernel}} = 1$, simplifying Equation 11 to

$$(\mathbf{Z} * \mathbf{K})_{j,h,w} = \sum_{a=1}^{d_{\text{in}}} k_{j,a,1,1} z_{a,h,w}. \tag{12}$$

We will also consider $d_{\text{in}} = 3$ to be the RGB representation of a pixel at each location, allowing easy visualization. The starting notebook available at this link provides an input tensor and a kernel tensor. Use this starting notebook to:

i. Visualize the provided input tensor using Matplotlib's imshow function. Additionally, visualize each filter in the given "kernels" tensor as a separate subplot in one figure. Note that PyTorch and Matplotlib follow different conventions around the semantics of each axis in a tensor. In PyTorch, an image is represented with the channel dimension (e.g., RGB) first, followed by height and width. In Matplotlib, the convention is height and width followed by channel. To display torch tensors using matplotlib, you will need to reorder the axes using torch.permute such that the channel dimension comes last rather than first. Note that the color of each filter appears in the input tensor.

ii. Use the PyTorch function torch.nn.functional.conv2d to convolve the given input tensor with the given kernels. This should produce an output tensor in $\mathbb{R}^{d_{\text{out}}\times h_{\text{in}}\times w_{\text{in}}}$. Display the activation map for each kernel as a subplot of one plot, i.e., produce d_{out} subplots of shape $h_{\text{in}} \times w_{\text{in}}$. Based on these visualizations, you should notice that the entries in these matrices are not produced by a similarity function. State which of the rules for similarity functions is violated and how you know.

iii. Let's take a moment to consider something that *is* a similarity function. For two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, the cosine similarity between $\mathbf{x}$ and $\mathbf{y}$ is defined to be:

$$\text{CosSim}(\mathbf{x}, \mathbf{y}) := \cos(\theta(\mathbf{x}, \mathbf{y})), \tag{13}$$

where $\theta : \mathbb{R}^{d\times d} \to [0, 2\pi)$ is a function that gets the angle between two vectors. Prove that CosSim is a similarity metric. Hint: θ is a distance metric.

[3]In this problem "convolution" refers to the operation used in convolutional neural networks. Formally, we are actually going to discuss cross-correlation, but the machine learning community refers to this operation as convolution by convention.

iv. Implement CosSim using a processing step on $\mathbf{Z}$ and $\mathbf{K}$ and torch.nn.functional.conv2d; that is, perform a transformation T on $\mathbf{Z}$ and $\mathbf{K}$ such that $(\bar{\mathbf{Z}} * \bar{\mathbf{K}})_{j,h,w}$ computes a cosine similarity, where $\bar{\mathbf{Z}} = T(\mathbf{Z})$ and $\bar{\mathbf{K}} = T(\mathbf{K})$. Display the activation map for each kernel. Do the activation maps reflect a similarity function? What is the difference between what you just implemented and a normal convolution? Hint: Recall that $\mathbf{x}^T\mathbf{y} = \|\mathbf{x}\|_2\|\mathbf{y}\|_2 \cos(\theta(\mathbf{x}, \mathbf{y}))$.

(b) We now attend to attention. Recall the definition of dot product attention as computed in "Attention is All You Need". For matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{w_{\text{in}} \times d_{\text{in}}}$, we have

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) := \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{in}}}}\right)\mathbf{V}. \tag{14}$$

In the simplest form of self-attention, an input matrix $\mathbf{Z} \in \mathbb{R}^{w_{\text{in}} \times d_{\text{in}}}$ is passed as all three arguments, i.e.

$$\text{SimpleSelfAttention}(\mathbf{Z}) := \text{Attention}(\mathbf{Z}, \mathbf{Z}, \mathbf{Z}) = \text{softmax}\left(\frac{\mathbf{Z}\mathbf{Z}^T}{\sqrt{d_{\text{in}}}}\right)\mathbf{Z}.$$

i. Note that you were given an input image in $\mathbb{R}^{d_{in} \times h_{in} \times w_{in}}$, which does not quite line up with the dimensions expected for attention. Flatten the given input into a matrix of shape $(d_{in}, h_{in}w_{in})$, transpose the resulting matrix to shape $(h_{in}w_{in}, d_{in})$, and use it to compute $\mathbf{Z}\mathbf{Z}^T$. Visualize the resulting matrix with imshow. Based on this visualization, could the entries in the resulting matrix be produced by a similarity function? State whether one of the rules for similarity functions is violated and how you know.

ii. Compute $\text{softmax}\left(\frac{\mathbf{Z}\mathbf{Z}^T}{\sqrt{d_{\text{in}}}}\right)$, and visualize the resulting matrix. Based on this visualization, could the entries in this matrix be produced by a similarity function? State whether a rule for similarity metrics is violated and how you know.

iii. Let $\bar{\mathbf{Z}} := \begin{bmatrix} 1/\|\mathbf{z}_{1,:}\|_2 & 0 & \dots & 0 \\ 0 & 1/\|\mathbf{z}_{2,:}\|_2 & \dots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \dots & 1/\|\mathbf{z}_{h_{in} \times w_{in},:}\|_2 \end{bmatrix}$. So the resulting matrix $\bar{\mathbf{Z}}$ should be in the shape of $(h_{in} \times w_{in}, h_{in} \times w_{in})$. Finally, compute $\text{softmax}\left(\frac{\bar{\mathbf{Z}}(\mathbf{Z}\mathbf{Z}^T)\bar{\mathbf{Z}}}{\sqrt{d_{\text{in}}}}\right)$, and visualize the resulting matrix (Note: The softmax should be applied across the resulting matrix, so you should flatten it before applying the softmax operation in PyTorch and reshape it after). Based on this visualization, could the entries in this matrix be produced by a similarity function? State whether one of the rules for similarity functions is violated and how you know.

iv. Based on the prior questions, you can see a connection between attention and convolution. Implement the function $f(\mathbf{Z}) := \frac{\bar{\mathbf{Z}}(\mathbf{Z}\mathbf{Z}^T)\bar{\mathbf{Z}}}{\sqrt{d_{\text{in}}}}$ using a convolution; reshape the output to a matrix of the appropriate size, and visualize the result. Hint: Recall that $\frac{\bar{\mathbf{Z}}(\mathbf{Z}\mathbf{Z}^T)\bar{\mathbf{Z}}}{\sqrt{d_{\text{in}}}} \in \mathbb{R}^{w_{\text{in}}h_{\text{in}} \times w_{\text{in}}h_{\text{in}}}$ and $(\mathbf{Z} * \mathbf{K}) \in \mathbb{R}^{d_{\text{out}} \times h_{\text{in}} \times w_{\text{in}}}$. How many convolutional kernels will you need to make these dimensions match?

5 Attention in Graph Neural Networks (Coding) (Stephen) - 15 pts

Oh no! We accidentally threw our entire collection of classical music scores into a wood chipper, and everything is shredded into multiple scraps of paper with just 50 musical notes per scrap. We need to put everything back together, but this is a daunting task. Let's automate part of the process by using a Graph Neural Network (GNN) to group different scraps by the composer who wrote the score.

Here's the plan: We are going to use a GNN to represent and encode the musical notes for *graph classification*. In other words, we'll translate a scrap of music into a *multi-relational* graph, where notes are nodes, and the relationships between the notes are represented by various types of edges. This graph representation of the music will then be processed by a few *message passing* layers using our old friend, attention! Finally, we aggregate the transformed node information and pass it through a feed-forward neural network to predict who composed the excerpt.

For the remainder of this problem, you need to copy and follow the instructions in this Colab. Also, be sure to download the music dataset from Canvas. Many steps have been completed for you; you will need to complete three vital steps:

1. Implement the Graph Attention (GAT) message passing layer,
2. Train and tune the hyperparameters of your GNN, and
3. Analyze the results of your multiclass classification model.