**UNIVERSITY OF PENNSYLVANIA**

**ESE 546: PRINCIPLES OF DEEP LEARNING**

**HOMEWORK 3**

- You will submit solutions typeset in LaTeX on Gradescope (strongly encouraged). You can use hw_template.tex on Canvas in the "Homeworks" folder to do so. If your handwriting is unambiguously legible, you can submit PDF scans/tablet-created PDFs.
- Clearly indicate the name and Penn email ID of all your collaborators on your submitted solutions.
- Start a new problem on a fresh page and mark all the pages corresponding to each problem. Failure to do so may result in your work not graded completely.
- For each problem in the homework, you should mention the total amount of time you spent on it. This helps us keep track of which problems most students are finding difficult.
- You can be informal while typesetting the solutions, e.g., if you want to draw a picture feel free to draw it on paper clearly, click a picture and include it in your solution. Do not spend undue time on typesetting solutions.
- You will see an entry of the form "HW 1 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 1 Problem 1 Code" and "HW 1 Problem 3 Code" where you will upload your solution for the respective problems.
- **For each programming problem/sub-problem, you should create a fresh .py file**. This file should contain **all** the code to reproduce the results of the problem/sub-problem, e.g., it should save the plot that is required (correctly with all the axes, title and legend) as a PDF in the same directory. You will upload the .py file as your solution for "HW 1 Problem 3 Code" or "HW 1 Problem 3 Code". Name your file as pennkey_hw1_problem3.py, e.g., I will name my code as pratikac_hw1_problem3.py. Note, we will not accept .ipynb files (i.e., Jupyter notebooks), you should only upload .py files. If you are using Google Colab to do your homework (and I suggest that you don't...), you can export the notebook to a .py file.
- **In addition to submitting the code, you should append the entire Python code for the particular problem to the solution in the PDF. If you are using Latex, you can do something like the screenshot below. The instructors will execute the code to check it. Your code should run without any errors and should create all output/plots required in the problem.**

```
\includepackage{pythonhighlight}

\begin{python}

a = np.array(10)

...

\end{python}
```

**Credit** The points for the problems add up to 110. You need to solve for 100 points to get full credit, i.e., your final score will be min(your total points, 100).

---

**Problem 1 (35 points, do this problem on your laptop when you are debugging but Colab with GPU will train much faster).** The pre-training procedure for a large language model (LLM) involves minimizing the cross-entropy entropy loss for predicting the next token on a large corpus of data. In this problem, you will fit a small Transformer on the dataset that you used for problem 4 of HW 2 (a concatenation of the complete works of Shakesphere, Leo Tolstoy's War and Peace and any English book of your choice from Project Gutenberg). We wish to build a network that can complete a given sentence.

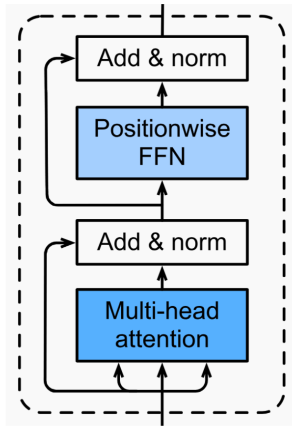
(a) **(3 points)** First, you should tokenize the dataset using a more sophisticated tokenizer than the characters that you used last time. You will use what is called a byte pair encoding (BPE) https://en.wikipedia.org/wiki/Byte-pair_encoding.
See the example shown at https://en.wikipedia.org/wiki/Byte-pair_encoding#Example. BPE works by scanning the text to identify frequently occurring byte pairs and collapsing them into a new symbol into the vocabulary. This is straight-forward to implement but it is quite slow in Python (it requires multiple passes over the dataset to compute the frequencies and compress the text). We will therefore use an existing library to build the vocabulary. Read the webpage at https://huggingface.co/docs/tokenizers/quicktour carefully to understand how to use Huggingface's tokenizers library (install it using "pip install tokenizers") to train and run a tokenizer.

```
from tokenizers import Tokenizer
from tokenizers.trainers import BpeTrainer
from tokenizers.models import BPE
from tokenizers.pre_tokenizers import BertPreTokenizer

tokenizer = Tokenizer(BPE(unk_token="[UNK]"))
tokenizer.pre_tokenizer=BertPreTokenizer()

trainer = BpeTrainer(special_tokens=["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]"])
tokenizer.train(['enwiki8.txt'],trainer)
```

You can set up your code to do something like the picture above. Two things to note here. First, we have showed an example that uses a few special tokens while tokenizing the text, e.g., [UNK] stands for "unknown token" for when the data might contain new characters that were not a part of the vocabulary, the [CLS] token is sometimes used for downstream tasks using pre-trained BERT/GPT representations, [PAD] stands for padding which allows the user to create a mini-batch of short and long sentences, etc. The vocabulary of chatbots contains many such special tokens for start of turn, end of turn, beginning of "thinking", end of "thinking" etc. Second, we are using a pre-tokenizer from Bert, this procedure simply sanitizes the text, e.g., splitting apostrophes with a space on both sides (so that the tokenizer will have the apostrophe symbol as an element of the vocabulary), etc. Such procedures are broadly called stemming and lemmatization in the NLP literature and they are a very important step before any language modeling can be done (Spacy https://spacy.io/api/tokenizer is an excellent library for this purpose).

You can give options to BpeTrainer to use a particular vocabulary size, use something in the region of 1024–2048.

(b) **(17 points)** Next you will write code for a (decoder-only) Transformer-based network. You may not use pre-coded Transformer blocks from any library (this includes PyTorch layers such as nn.Transformer, nn.TransformerEncoder etc.). You should write the following code by yourself using basic PyTorch tensors. Here is a rough guideline for the various parameters is as follows.




- A context length of $T = 128$.
- Each token is embedded as a vector into $d = 256$ dimensions with sine-cosine-based position encoding.
- The feature dimension of keys, queries and values is $p = 32$, the number of heads is 2.
- Each Transformer block has one self-attention-based layer and one MLP layer, with layer normalization and residual connections as shown in the picture above. We will code up all these blocks ourselves. You can set the bias to False for all operations (linear operations, key/value computation, as well as normalization).
- For multi-head self-attention if input features are $h \in \mathbb{R}^{T\times p}$, the computations of the multi-head attention block for each head denoted by the index $m = 1, 2$ are:

$$\begin{aligned}
\mathbb{R}^{T\times p} &\ni k^m = \sigma(h\, w_k^m) \text{ with } w_k^m \in \mathbb{R}^{p\times p}\\
\mathbb{R}^{T\times p} &\ni q^m = \sigma(h\, w_q^m)\\
\mathbb{R}^{T\times p} &\ni v^m = \sigma(h\, w_v^m)
\end{aligned}$$

$$\forall t : \mathbb{R}^p \ni {h'}_t^m = \text{dropout}\left(\sum_{s=1}^{t-1}\left(\frac{\exp\left({k_s^m}^\top q_t^m/\sqrt{p}\right)}{\sum_{u=1}^{t-1}\exp\left({k_u^m}^\top q_t^m/\sqrt{p}\right)}\right)\, v_t^m\right).$$

  We then concatenate the output of each head to get features $h' \in \mathbb{R}^{T\times 2p}$ before an output projection layer

$$\mathbb{R}^{T\times p} \ni h_{\text{attn}} = h' w_o \text{ with } w_o \in \mathbb{R}^{2p\times p}.$$

- Residual connection and normalization are easy to write

$$\mathbb{R}^{T\times p} \ni h_{\text{attn}} = \text{layer norm}(h + h_{\text{attn}})$$

  You can use PyTorch's nn.LayerNorm to implement this operation.
- The next thing we should is to implement a position-wise MLP to mix the features computed by self-attention. This looks like

$$\mathbb{R}^{T\times p} \ni h = \text{layer norm}(h_{\text{attn}}(t) + \text{MLP}(h_{\text{attn}}(t)))$$

with the MLP being a two-layer network with ReLU nonlinearities, $p$ input neurons, $2p$ hidden neurons and $p$ output neurons. Notice that the MLP acts on the input data position-wise.

- All this together forms one "block" of a Transformer. Your network should have 2–4 such blocks.

(c) **(10 points)** Train this network on the tokenized data for predicting the next token using the past tokens in the context (you will see that the equations above have causal attention). In this problem, we are only interested in completing a given sentence, so we will create a mini-batch where each datum is a sentence. Remember that you will need to use padding while creating mini-batches that contain sentences of unequal lengths.

You should plot the (i) training loss of successive mini-batches as a function of the weight updates (feel free to do a moving average), (ii) periodically, say after every 1000 weight updates, you can also compute the validation loss on the validation dataset.

(d) **(5 points)** Write code for auto-regressive sampling using this network. Given a prompt, say "The Effiel Tower is in Paris", your code will tokenize the prompt (let's say this prompt contains 6 tokens $x_1, x_2, \ldots, x_6$), and run the sampler step-by-step as

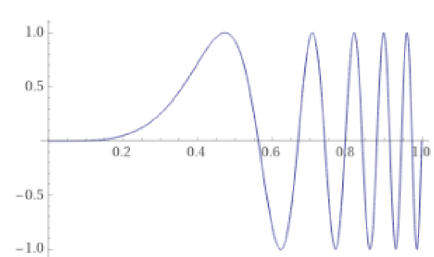
$$\begin{aligned}
\hat{x}_7 &\sim p(\cdot \mid x_1, \ldots, x_6) \\
\hat{x}_8 &\sim p(\cdot \mid x_1, \ldots, x_6, \hat{x}_7) \\
\hat{x}_9 &\sim p(\cdot \mid x_1, \ldots, x_6, \hat{x}_7, \hat{x}_8),
\end{aligned}$$

each sample involves a call to the network and a random draw from the probability distribution of the logits. Remember that you need to use position-encoding when you feed-in the context. You should report a few examples of the network completing your prompts (use prompts of length 5–6 words, completions of length 7–8 words).

**Problem 2 (20 points. Do this problem on your laptop; you do not really need a GPU for this problem.).** In this problem, we will use nonlinear regression to understand one possible reason why deep networks can avoid overfitting even when the number of samples in the dataset is smaller than the number of weights. We will effectively show that generalization occurs when the test data lies within the "convex hull" of the training data. If the test samples lie far away from the train data, then one should not expect any generalization with networks with lots of parameters such as deep networks. A second purpose of working on this problem is to see that we can do interesting experiments and train 1000s of networks on a laptop (even without a GPU).

(a) **(0 points)** You will create the dataset yourself. Let the true data come from

$$\mathbb{R} \ni y = f^*(x) \equiv \sin\big(10\pi x^4\big); \text{ for } x \in [0, 1].$$



You will write a function to sample $n$ inputs $x_i \in [0,1]$ and their corresponding outputs $y_i = \sin\left(10\pi x_i^4\right)$ to create a dataset $D_n = \{(x_i, y_i)\}_{i=1}^n$.

(b) **(10 points)** Build a multi-layer perceptron (MLP) in PyTorch to regress the dataset. You can use an MLP with 2-3 layers, ReLU nonlinearities with batch-normalization after each layer and train it with SGD. You can use any PyTorch function for this that you would like; but it is actually easier to code the mini-batch sampling etc. yourself simply using a for loop for the iterations instead of using a dataloader. You should choose the hyper-parameters (weight-decay, learning rate, number of epochs) to ensure that you are getting zero or very close to zero training error. Denote the learned function as

$$\hat{y} = f_w(x; n)$$

where $w$ are the weights of the MLP, $x$ is the test input and we have denoted by $; n$ the fact that this model was trained with $n$ data points. Make sure your code for training the MLP is efficient, we will be fitting about 100 different models. Solving this problem will also convince you that your laptops are plenty powerful to ask serious questions about why deep networks work; you do not need GPUs for everything in deep learning.

Write a function to evaluate the quantity

$$\delta f_{\text{in}}(n) = \max_{x \in [0,1]} |f_w(x; n) - f^*(x)|$$
$$\delta f_{\text{out}}(n) = \max_{x \in [0,1.5]} |f_w(x; n) - f^*(x)|.$$

You will evaluate the maximum in the above expression by sampling many (say 1000) test inputs $x \in [0,1]$ or $x \in [0, 1.5]$ and taking the maximum of their predictions. The quantity $\delta f_{\text{in}}(n)$ is the largest discrepancy within the support of the dataset, i.e., within $[0,1]$, between the true function $f^*(x)$ and our fitted function $f_w(x; n)$ using $n$ training data points. Similarly the quantity $\delta f_{\text{out}}(n)$ is the largest discrepancy outside the support, for example in $x \in [0, 1.5]$.

(c) **(10 points)** For each of 20 different values of $n$, e.g., ns = np.logspace(1, 3, 20).astype(int), create 5 training datasets (i.e., you will create 100 different datasets in total) like it is described in part (a), fit an MLP to this dataset and evaluate the mean and standard deviation (across the 5 datasets corresponding to each $n$) of both $\delta f_{\text{in}}(n)$ and $\delta f_{\text{out}}(n)$. Draw a plot of $\log \delta f_{\text{in}}(n)$ (mean and standard deviation) versus $n$. On the same plot also draw $\log \delta f_{\text{out}}(n)$ (mean and standard deviation) versus $n$. Interpret your plot.

**Problem 3 (20 points).** In this problem you will show that co-coercivity of the gradient and its Lipschitz continuity are equivalent. Assume that the function $f(x)$ is convex.

(a) **(5 points)** For a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$, shat that co-coercivity of $\nabla f$ implies Lipschitz continuity of $\nabla f$. In other words, show that for all $x, y$

$$\langle \nabla f(x) - \nabla f(y), x - y \rangle \geq \frac{1}{L} \|\nabla f(x) - \nabla f(y)\|^2 \implies \|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

Hint: use the Cauchy-Schwartz inequality.

(b) **(8 points)** Show the converse, i.e., show that Lipschitz continuity of $\nabla f$ implies co-coercivity.

(c) **(7 points)** For a twice-differentiable function $f(x)$ with $L$-Lipschitz gradients and strong-convexity parameter $m$, show that

$$m \leq \left\|\nabla^2 f(x)\right\|_2 \leq L$$

for all $x$. Hint: use the mean-value theorem on a line that joins two points $x, y$.

**Problem 4 (35 points). (Do this on your laptop)** In this problem, we will implement logistic regression for classifying two classes (zero and one) from MNIST. You may not use any routines from PyTorch other than the ones that help download the data. Use Numpy to code up the optimization algorithm.

(a) **(0 points)** First, prepare the dataset. Select all the samples belonging to the first two classes from MNIST's training dataset, this will be your training dataset. Similarly, create a validation dataset for the two classes by picking samples corresponding to the first two classes from the validation set of the MNIST dataset. You can subsample input images from 28×28 to 14×14 if you need.

(b) **(10 points)** Logistic regression solves for

$$\underset{w \in \mathbb{R}^d, w_0 \in \mathbb{R}}{\operatorname{argmin}} \; \frac{1}{n} \sum_{i=1}^{n} \log \left(1 + e^{-y_i \, (w^\top x_i + w_0)}\right) + \frac{\lambda}{2} \left(\|w\|_2^2 + w_0^2\right) \tag{1}$$

where $x \in \mathbb{R}^{196}$. Set $y_i = 1$ for MNIST images labeled zero and $y_i = -1$ for MNIST images labeled one. Initialize the weights randomly for both the following parts but make sure that they are the same for both gradient descent and gradient descent with Nesterov's acceleration in part (c). You can try a few different values of $\lambda$ and pick the one that gives the best validation error for the following parts.

Optimize the objective in Eq. (1) using gradient descent (note, not stochastic gradient descent) and plot the training *loss* as a function of the number of parameter updates on a semi-log scale (log scale on the Y-axis). This plot should be a straight line. As we saw in the class, the slope of this line should be about $-\kappa^{-1}$ for gradient descent. Compute the slope of the line in your plot and mention it clearly.

(c) **(5 points)** Write down the Hessian of the loss function in Eq. (1). Without assuming any special conditions about the dataset $\{(x_i, y_i)\}_{i=1,\dots,n}$, is this problem strongly convex? What is the best strong convexity parameter for the loss function in Eq. (1)?

(d) **(10 points)** Optimize the objective in Eq. (1) again, this time using gradient descent with Nesterov's acceleration. If we knew the condition number $\kappa$, what momentum coefficient would we use for this problem? It is difficult, although possible, to evaluate $\kappa$, so let's use

$$\kappa = \frac{L}{m}$$

where we treat $L$ as a hyper-parameter, i.e., we *choose* a value for $L$ and set $m$ to be the solution of part (b). Again, choose the value of $L$ by trying out a few values and looking at the slope of the training loss for each setting. (Hint: the momentum parameter is typically between 0.75-0.95).

The slope of the semi-log plot of training loss versus the number of parameter updates in this case will be about $-\kappa^{-1/2}$ for Nesterov's updates. Note that we do not know the correct $\kappa$ in this problem, so your slope may not match the value you chose for $\kappa$ above. You should however see that the slope of the plot for Nesterov's acceleration is better than the slope of the plot you obtained in part (b).

(e) **(10 points)** We will now optimize the same problem with stochastic gradient descent (SGD). Use a batch-size $b = 128$ (feel free to try diffetent batch-size such as 64 and 8) and optimize Eq. (1) using SGD with and without Nesterov's acceleration. Plot the training loss against the number of parameter updates on a semi-log scale (log scale on the Y-axis) for (i) gradient descent with Nesterov's updates (can be the same plot from your previous solutions), (ii) SGD without Nesterov's acceleration and,

(iii) SGD with Nesterov's acceleration. Is the convergence for (iii) faster than that of (ii)? Comment on the differences for the convergence curves of (i) and (ii).