Due Date: March 22nd 23:59, 2019

Instructions

- *For all questions, show your work!*
- *Starred questions are **hard** questions, not **bonus** questions.*
- *Submit your report (pdf) electronically via the course Gradescope page.*
- *You must push (all of!) your code to a Github repository, and include the link to the repository in your report (even if using a Jupyter notebook)*
- *An outline of code will be provided in the course repo at this link. You must start from this outline and follow the instructions in it (even if you use different code, you must follow the overall outline and instructions).*
- *TAs for this assignment are David Krueger, Tegan Maharaj, and Chin-Wei Huang.*

Summary:

In this assignment, you will implement and train **sequential language models** on the Penn Treebank dataset. Language models learn to assign a likelihood to sequences of text. The elements of the sequence (typically words or individual characters) are called tokens, and can be represented as one-hot vectors with length equal to the vocabulary size, e.g. 26 for a vocabulary of English letters with no punctuation or spaces, in the case of characters, or as indices in the vocabulary for words. In this representation an entire dataset (or a mini-batch of examples) can be represented by a 3-dimensional tensor, with axes corresponding to: (1) the example within the dataset/mini-batch, (2) the time-step within the sequence, and (3) the index of the token in the vocabulary. Sequential language models do **next-step prediction**, in other words, they predict tokens in a sequence one at a time, with each prediction based on all the previous elements of the sequence. A trained sequential language model can also be used to generate new sequences of text, by making each prediction conditioned on the past *predictions* (instead of the ground-truth input sequence).

Problems 1-3 are respectively the implementation of (1) a **simple ("vanilla") RNN** (recurrent neural network), (2) an RNN with a gating mecahnism on the hidden state, specifically with **gated recurrent units (GRUs)**, and (3) the **attention module of a transformer network** (we provide you with Pytorch code for the rest of the transformer). Problem 4 is to train these 3 models using a variety of different optimizers and hyperparameter settings. Problem 5 involves analyzing the behavior of the trained models in more detail. Each problem is worth 20 points.

The Penn Treebank Dataset This is a dataset of about 1 million words from about 2,500 stories from the Wall Street Journal. It has Part-of-Speech annotations and is sometimes used for training parsers, but it's also a very common benchmark dataset for training RNNs and other sequence models to do next-step prediction.

Preprocessing: The version of the dataset you will work with has been preprocessed: lowercased, stripped of non-alphabetic characters, tokenized (broken up into words, with sentences separated by the `<eos>` (end of sequence) token), and cut down to a vocabulary of 10,000 words; any word not in this vocabulary is replaced by `<unk>`. For the transformer network, positional information (an embedding of the position in the source sequence) for each token is also included in the input sequence. In both cases the preprocessing code is given to you.

Instructions for Problems 1 and 2:

You will manually implement two types of recurrent neural network (RNN). The implementation must be able to process mini-batches. Implement the models **from scratch** using Pytorch/Tensorflow Tensors, Variables, and associated operations (e.g. as found in the `torch.nn` module). Specifically, use appropriate matrix and tensor operations (e.g. dot, multiply, add, etc.) to implement the recurrent unit calculations; you **may not** use built-in Recurrent modules. You **may** subclass `nn.module`, use built-in Linear modules, and built-in implementations of nonlinearities (tanh, sigmoid, and softmax), initializations, loss functions, and optimization algorithms. Your code must start from the code scaffold and follow its structure and instructions.

Problem 1

Implementing a Simple RNN (20pts) A simple recurrent neural network (SRNN) is also called a "vanilla" RNN or "Elman network". The equations for an SRNN are:

$$P(\boldsymbol{y}_t|\boldsymbol{x}_1, ..., \boldsymbol{x}_t) = \sigma_y(\boldsymbol{W}_y\boldsymbol{h}_t + \boldsymbol{b}_y) \tag{1}$$

$$\boldsymbol{h}_{t+1} = \sigma_h(\boldsymbol{W}_x\boldsymbol{x}_{t+1} + \boldsymbol{W}_h\boldsymbol{h}_t + \boldsymbol{b}_h) \tag{2}$$

Where $\boldsymbol{h}_t$ is the *hidden state* at timestep t, computed as a function of the input $\boldsymbol{x}_t$ and the hidden state from the previous timestep $\boldsymbol{h}_{t-1}$, using weight parameters $\boldsymbol{W}_x, \boldsymbol{W}_h, \boldsymbol{W}_y$, and bias parameters $\boldsymbol{b}_h, \boldsymbol{b}_y$, hidden-layer activation function σ_h and output activation function σ_y. $\boldsymbol{y}_t$ is the target at timestep t. For sequential language modelling, the target is the next element of the input sequence, [1] i.e. $\boldsymbol{y}_t = \boldsymbol{x}_{t+1}$, and the loss is typically *cross entropy*. The model is trained to make a prediction for every time-step $t = 1, ..., T$, and the loss $\mathcal{L}$ is the sum of the losses for the predictions at each time-step:

$$\mathcal{L}(x_1, ..., x_T) = \sum_{t=1}^{T} \mathcal{L}_t \tag{3}$$

$$\mathcal{L}_t = -\log P_\theta(\boldsymbol{x}_t|\boldsymbol{x}_1, ..., \boldsymbol{x}_{t-1}) \tag{4}$$

Note that it is necessary to specify an initial hidden state $\boldsymbol{h}_0$.

Note a previous version had further instructions here; now all instructions are in the code. In particular we no longer ask you to learn the initial hidden states (instead they are carried over from the previous mini-batch). Apologies for the changes.

[1] In some settings, RNNs are also trained to output a special **termination symbol**, to indicate the end of the sequence, on the final time-step.

Problem 2

Implementing an RNN with Gated Recurrent Units (GRU) (20pts) The use of "gating" (i.e. element-wise multiplication, represented by the $\odot$ symbol) can significantly improve the performance of RNNs. The Long-Short Term Memory (LSTM) RNN is the best known example of gating in RNNs; GRU-RNNs are a slightly simpler variant (with fewer gates).

The equations for a GRU are:

$$\boldsymbol{r}_t = \sigma_r(\boldsymbol{W}_r\boldsymbol{x}_t + \boldsymbol{U}_r\boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \tag{5}$$
$$\boldsymbol{z}_t = \sigma_z(\boldsymbol{W}_z\boldsymbol{x}_t + \boldsymbol{U}_z\boldsymbol{h}_{t-1} + \boldsymbol{b}_z) \tag{6}$$
$$\tilde{\boldsymbol{h}}_t = \sigma_h(\boldsymbol{W}_h\boldsymbol{x}_t + \boldsymbol{U}_h(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \tag{7}$$
$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t \tag{8}$$
$$P(\boldsymbol{y}_t|\boldsymbol{x}_1, ..., \boldsymbol{x}_t) = \sigma_y(\boldsymbol{W}_y\boldsymbol{h}_t + \boldsymbol{b}_y) \tag{9}$$

$\boldsymbol{r}_t$ is called the "reset gate" and $\boldsymbol{z}_t$ the "forget gate". The trainable parameters are $\boldsymbol{W}_r, \boldsymbol{W}_z, \boldsymbol{W}_h, \boldsymbol{W}_y$, $\boldsymbol{U}_r, \boldsymbol{U}_z, \boldsymbol{U}_h, \boldsymbol{b}_r, \boldsymbol{b}_z, \boldsymbol{b}_h$, and $\boldsymbol{b}_y$.GRUs use the sigmoid activation function for σ_r and σ_z, and tanh for σ_h.

Note a previous version had further instructions here; now all instructions are in the code. Apologies for the changes.

Problem 3

Implementing the attention module of a transformer network (20pts) While prototypical RNNs "remember" past information by taking their previous hidden state as input at each step, recent years have seen a profusion of methodologies for making use of past information in different ways. The transformer [2] is one such fairly new architecture which uses several self-attention networks ("heads") in parallel, among other architectural specifics. The transformer is quite complicated to implement compared to the RNNs described so far; most of the code is provided and your task is only to implement the multi-head scaled dot-product attention. The attention vector for m heads indexed by i is calculated as follows:

$$\boldsymbol{A}_i = \text{softmax}\left(\frac{(\boldsymbol{Q}\boldsymbol{W}_{Q_i} + \boldsymbol{b}_{Q_i})(\boldsymbol{K}\boldsymbol{W}_{K_i} + \boldsymbol{b}_{K_i})^\top}{\sqrt{d_k}}\right) \tag{10}$$
$$\boldsymbol{H}_i = \boldsymbol{A}_i(\boldsymbol{V}\boldsymbol{W}_{V_i} + \boldsymbol{b}_{V_i}) \tag{11}$$
$$A(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{concat}(\boldsymbol{H}_1, ..., \boldsymbol{H}_m)\boldsymbol{W}_O + \boldsymbol{b}_O \tag{12}$$

where $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are queries, keys, and values respectively, $\boldsymbol{W}_{Q_i}, \boldsymbol{W}_{K_i}, \boldsymbol{W}_{V_i}$ are their corresponding embedding matrices, $\boldsymbol{W}_O$ is the output embedding, and d_k is the dimension of the keys. $\boldsymbol{Q}, \boldsymbol{K}$, and $\boldsymbol{V}$ are determined by the output of the feed-forward layer of the main network (given to you). $\boldsymbol{A}_i$ are the attention values, which specify which elements of the input sequence each attention head attends to.

[2] See `https://arxiv.org/abs/1706.03762` for more details.

Note that the implementation of multi-head attention requires binary masks, so that attention is computed only over the past, not the future. A mask value of 1 indicates an element which the model is allowed to attend to (i.e. from the past); a value of 0 indicates an element it is not allowed to attend to. This can be implemented by modifying the softmax function to account for the mask $\boldsymbol{s}$ as follows:

$$\tilde{\boldsymbol{x}} = \exp(\boldsymbol{x}) \odot \boldsymbol{s} \tag{13}$$

$$\text{softmax}(\boldsymbol{x}, \boldsymbol{s}) \doteq \frac{\tilde{\boldsymbol{x}}}{\sum_i \tilde{x}_i} \tag{14}$$

To avoid potential numerical stability issues, we recommend a different implementation:

$$\tilde{\boldsymbol{x}} = \boldsymbol{x} \odot \boldsymbol{s} - 10^9(1 - \boldsymbol{s}) \tag{15}$$

$$\text{softmax}(\boldsymbol{x}, \boldsymbol{s}) \doteq \frac{\exp(\tilde{\boldsymbol{x}})}{\sum_i \exp(\tilde{x}_i)} \tag{16}$$

This second version is equivalent (up to numerical precision) as long as $\boldsymbol{x} >> -10^9$, which should be the case in practice.

Problem 4

Training language models (20pts) Unlike in classification problems, where the performance metric is typically accuracy, in language modelling, the performance metric is typically based directly on the cross-entropy loss, i.e. the negative log-likelihood (NLL) the model assigns to the tokens. For word-level language modelling it is standard to report **perplexity (PPL)**, which is the exponentiated average per-token NLL (over all tokens):

$$\exp\left(\frac{1}{TN}\sum_{t=1}^{T}\sum_{n=1}^{N} -\log P(\boldsymbol{x}_t^{(n)}|\boldsymbol{x}_1^{(n)}, ..., \boldsymbol{x}_{t-1}^{(n)})\right),$$

where t is the index with the sequence, and n indexes different sequences. For Penn Treebank in particular, the test set is treated as a single sequence (i.e. $N = 1$). The purpose of this assignment is to perform model exploration, which is done using a validation set. As such, we do not require you to run your models on the test set.

(1) Model Comparison
In this problem you will run one experiment for each architecture (hyperparameter settings specified in the code) (3 experiments).

(2) Exploration of optimizers
You will run experiments with the following three optimizers (use the implementations provided in the code or given in Pytorch/Tensorflow; you don't need to implement these yourself) (6 experiments)

- "Vanilla" Stochastic Gradient Descent (SGD)
- SGD with a learning rate schedule; divide the learning rate by 1.15 after each epoch
- Adam

(3) Exploration of hyperparmeters

In this problem, you will explore combinations of hyperparameters to try to find settings which achieve better validation performance than those given to you in (1). Report at least 3 more experiments per architecture (you may want to run many more short experiments in order to find potentially good hyperparameters). (9+ experiments).

Figures and Tables:

Each table and figure should have an explanatory caption. For tables, this goes above, for figures it goes below. If it is necessary for space to use shorthand or symbols in the figure or table these should be explained in the caption. Tables should have appropriate column and/or row headers. Figures should have labelled axes and a legend. Include the following tables:

1. For **each experiment** in 1-3, plot **learning curves** (train and validation) of PPL over both **epochs** and **wall-clock-time**.
2. Make a table of results summarizing the train and validation performace for each experiment, indicating the architecture and optimizer. Sort by architecture, then optimizer, and number the experiments to refer to them easily later. Bold the best result for each architecture. [3]
3. List all of the hyperparameters for each experiment in your report (e.g. specify the command you run in the terminal to launch the job, including the command line arguments).
4. Make 2 plots for each optimizer; one which has all of the validation curves for that optimizer over **epochs** and one over **wall-clock-time**.
5. Make 2 plots for each arcitecture; one which has all of the validation curves for that architecture over **epochs** and one over **wall-clock-time**.

Discussion

Answer the following questions in the report, referring to the plots / tables / code :

1. What did you expect to see in these experiments, and what actually happens? Why do you think that happens?
2. Referring to the learning curves, qualitatively discuss the differences between the three optimizers in terms of training time, generalization performance, which architecture they're best for, relationship to other hyperparameters, etc.
3. Which hyperparameters+optimizer would you use if you were most concerned with wall-clock time? With generalization performance? In each case, what is the "cost" of the good performance (e.g. does better wall-clock time to a decent loss mean worse final loss? Does better generalization performance mean longer training time?)
4. Which architecture is most "reliable" (decent generalization performance for most hyperparameter+optimizer settings), and which is more unstable across settings?
5. Describe a question you are curious about and what experiment(s) (i.e. what architecture/optimizer/hyperparameters) you would run to investigate that question.

[3] You can also make the table in LaTeX, but you can also make it using Excel, Google Sheets, or a similar program, and include it as an image.

Problem 5

Detailed evaluation of trained models (20pts) For this problem, we will investigate properties of the trained models from Problem 4. Perform the following evaluations for the models you trained in Problem 4.1.

*1 For the best performing model from each architecture (RNN, GRU, Transformer), compute the average loss at each time-step (i.e. $\mathcal{L}_t$ for each t) within validation sequences. Plot $\mathcal{L}_t$ as a function of t, describe the result qualitatively, and provide an explanation for what you observe. Compare the plots for the different architectures.

*2 For one minibatch of training data, compute the average gradient of the loss at the *final* time-step with respect to the hidden state at *each* time-step t: $\nabla_{\boldsymbol{h}_t}\mathcal{L}_T$ for the best performing RNN and GRU. The norm of these gradients can be used to evaluate the propagation of gradients; a rapidly decreasing norm means that longer-term dependencies have less influence on the training signal, and can indicate **vanishing gradients**. Plot the Euclidian norm of $\nabla_{\boldsymbol{h}_t}\mathcal{L}_T$ as a function of t for both the RNN and GRU. Rescale the values of each curve to [0,1] so that you can compare both on one plot. Describe the results qualitatively, and provide an explanation for what you observe, discussing what the plots tell you about the gradient propagation in the different architectures.

*3 Generate samples from both the Simple RNN and GRU, by recursively sampling $\hat{\boldsymbol{x}}_{t+1} \sim P(\boldsymbol{x}_{t+1}|\hat{\boldsymbol{x}}_1, ..., \hat{\boldsymbol{x}}_t)$. [4] Make sure to condition on the sampled $\hat{\boldsymbol{x}}_t$, *not* the ground truth. Produce 20 samples from both the RNN and GRU: 10 sequences of the same length as the training sequences, and 10 sequences of *twice* the length of the training sequences. Choose 3 "best", 3 "worst", and 3 that are "interesting", and make reference to these to describe qualitatively the different models' strengths, failure modes, quirks, and any potentially interesting things to investigate. Put all 40 samples in an appendix to your report.

[4] It is possible to generate samples in the same manner from the Transformer, but the implementation is more involved, so you are not required to do so.