**Due Date: March 17th 23:00, 2020**

Instructions

- *For all questions, show your work!*

- *Submit your report (pdf) and your code electronically via the course Gradescope page.*

- *An outline of code will be provided in the course repo at <u>this link</u>. 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 Jessica Thompson, Jonathan Cornford and Lluis Castrejon.*

**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).

As a starting point, you are provided with an implementation of a **simple ("vanilla") RNN** (recurrent neural network). Problem 1 is to implement an RNN with a gating mechanism on the hidden state, specifically with **gated recurrent units (GRUs)**. Problem 2 is to implement the **attention module of a transformer network** (we provide you with PyTorch code for the rest of the transformer). Problem 3 is to train these 3 models using a variety of different optimizers and hyperparameter settings and Problem 4 is to analyze the behaviour of the trained models. Each problem is worth 25 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: lower-cased, 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 infor-
mation (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.

# Problem 1

**Implementing an RNN with Gated Recurrent Units (GRU) (25pts)**    The implementa-
tion of your RNN must be able to process mini-batches. Implement the model **from scratch** using
PyTorch 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 im-
plement 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.

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:

$$\boldsymbol{r}_t = \sigma_r(\boldsymbol{W}_r\boldsymbol{x}_t + \boldsymbol{U}_r\boldsymbol{h}_{t-1} + \boldsymbol{b}_r) \tag{1}$$

$$\boldsymbol{z}_t = \sigma_z(\boldsymbol{W}_z\boldsymbol{x}_t + \boldsymbol{U}_z\boldsymbol{h}_{t-1} + \boldsymbol{b}_z) \tag{2}$$

$$\tilde{\boldsymbol{h}}_t = \sigma_h(\boldsymbol{W}_h\boldsymbol{x}_t + \boldsymbol{U}_h(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b}_h) \tag{3}$$

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t \tag{4}$$

$$P(\boldsymbol{y}_t|\boldsymbol{x}_1, ...., \boldsymbol{x}_t) = \sigma_y(\boldsymbol{W}_y\boldsymbol{h}_t + \boldsymbol{b}_y) \tag{5}$$

$\boldsymbol{r}_t$ 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 $\boldsymbol{b}_y$, as well as the initial hidden state parameter $\boldsymbol{h}_0$. GRUs use the sigmoid
activation function for $\sigma_r$ and $\sigma_z$, and tanh for $\sigma_h$.

*See further instructions in the solution template.*

# Problem 2

**Implementing the attention module of a transformer network (25pts)**    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 [1] 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

---

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

only to implement the multi-head scaled dot-product attention. The attention vector for $m$ heads indexed by $i$ is calculated as follows:

$$\boldsymbol{A}_i = \text{softmax}\left(\frac{\boldsymbol{Q}_i \boldsymbol{W}_{Q_i}(\boldsymbol{K}_i \boldsymbol{W}_{K_i})^\top}{\sqrt{d_k}}\right) \tag{6}$$

$$\boldsymbol{H}_i = \boldsymbol{A}_i \boldsymbol{V} \boldsymbol{W}_{V_i} \tag{7}$$

$$A(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{concat}(\boldsymbol{H}_1, ..., \boldsymbol{H}_m)\boldsymbol{W}_O \tag{8}$$

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.

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:

$$\tilde{\boldsymbol{x}} = \exp(\boldsymbol{x}) \odot \boldsymbol{s} \tag{9}$$

$$\text{softmax}(\boldsymbol{x}, \boldsymbol{s}) \doteq \frac{\tilde{\boldsymbol{x}}}{\sum_i \tilde{x}_i} \tag{10}$$

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

$$\tilde{\boldsymbol{x}} = \boldsymbol{x} \odot \boldsymbol{s} - 10^9(1 - \boldsymbol{s}) \tag{11}$$

$$\text{softmax}(\boldsymbol{x}, \boldsymbol{s}) \doteq \frac{\exp(\tilde{\boldsymbol{x}})}{\sum_i \exp(\tilde{x}_i)} \tag{12}$$

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

# Problem 3

**Training language models and model comparison (25pts)** 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.

You will train each of the three architectures using either stochastic gradient descent or the ADAM optimizer. The training loop is provided in *run_exp.py*.

1. - 4. You are asked to run 4 experiments (3.1, 3.2, 3.3, 3.4) with different architectures, optimizers, and hyperparameters settings. These parameter settings are given to you in the code (*run_exp.py*). In total there are 15 settings for you to run $(5+3+3+4 = 15)$. For each experiment (3.1, 3.2, 3.3, 3.4), plot learning curves (train and validation) of PPL over both **epochs** and **wall-clock-time**. Figures should have labeled axes and a legend and an explanatory caption.

5. Make a table of results summarizing the train and validation performance 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.[2] The table should have an explanatory caption, and appropriate column and/or row headers. Any shorthand or symbols in the table should be explained in the caption.

6. Which hyperparameters + optimizer would you use if you were most concerned with wall-clock time, with generalization performance.

7. For exp 3.1 you trained an RNN with either SGD or ADAM. What did you notice about the optimizer's performance with different learning rates?

8. For exp 3.2 you trained a GRU. Was its performance as you expected and why?

9. In exp 3.3 you explored different hyperparameter settings in an attempt to improve the performance of the GRU. Were the validation/training curves as you expected for each setting? Comment on why. *Hint: For each hyperparameter setting, consider how the training and validation phases differ.*

10. In exp 3.4 you trained a Transformer with various hyper-parameter settings. Given the recent high profile transformer based language models, are the results as you expected? Speculate as to why or why not.

# Problem 4

**Detailed evaluation of trained models (25pts)**    For this problem, we will investigate properties of the trained models from Problem 3. Perform the following evaluations for the two models (one RNN and one GRU) for which the parameters were saved (indicated by the flag –save_best in the code).

1. 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 $t$: $\nabla_{h_t} \mathcal{L}_T$. The norm of these gradients can be used to evaluate the propagation of gradients; a rapidly decreasing norm

---

[2]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.

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 the RNN and GRU architectures. 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.

2. Generate samples from both the RNN and GRU models, by recursively making $\hat{\boldsymbol{x}}_{t+1} = \arg\max P(\boldsymbol{x}_{t+1}|\hat{\boldsymbol{x}}_1, ...., \hat{\boldsymbol{x}}_t)$.[3] 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. Do you think that the generated sequence quality correlates with model validation perplexity? Justify your answer.

   Choose 3 "best", 3 "worst", and 3 that are "interesting". Put all 40 samples in an appendix to your report.

---

[3]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.