Deep Learning - Spring 2019
Homework 3

Daniel Rivera Ruiz
Department of Computer Science
New York University
drr342@nyu.edu

1 Fundamentals

1.1 Dropout

Dropout is a very popular regularization technique.

(a) List the `torch.nn` module corresponding to 2D dropout.

`torch.nn.Dropout2d(p=0.5, inplace=False)`. Randomly zero out entire channels of the input tensor. Each channel will be zeroed out independently on every forward call with probability p using samples from a Bernoulli distribution.

(b) Read on what dropout is and give a short explanation on what it does and why it is useful.

Dropout provides a method of regularizing a broad family of models. To a first approximation, it makes bagging practical for ensembles of very large neural networks. More specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network. In most modern neural networks, we can achieve this by multiplying its output value by zero.

One advantage of dropout is that it is computationally cheap: using dropout during training requires only $O(n)$ computations per example per update. Another significant advantage is that it does not limit the type of model or training procedure that can be used: it works well with nearly any model that uses a distributed representation and can be trained with stochastic gradient descent.

Though the cost per step of applying dropout to a specific model is negligible, the cost of using it in a complete system can be significant. Typically, the optimal validation set error is much lower when using dropout, but this comes at the cost of a much larger model and many more iterations of the training algorithm.

It is worth noticing that dropout trains an ensemble of models that share hidden units. This means each hidden unit must be able to perform well regardless of which other hidden units are in the model. Dropout thus regularizes each hidden unit to be not merely a good feature but a feature that is good in many contexts.

1.2 Batch Normalization

(a) What does mini-batch refer to in the context of deep learning?

In the context of deep learning, mini-batch refers to a partition of the training dataset that is used to calculate a model's error and update its coefficients. Mini-batch gradient descent is a variation of the gradient descent algorithm that uses mini-batches and is the most common implementation of gradient descent in the field of deep learning. Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

(b) Read on what batch normalization is and give a short explanation on what it does and why it is useful.

Batch normalization is a method of adaptive reparametrization motivated by the difficulty of training very deep neural networks. In these models, the gradient tells how to update each parameter assuming that the other layers do not change. In practice, however, unexpected results can arise because many functions composed together are changed simultaneously, using updates that were computed under the assumption that the other functions remain constant.

Batch normalization provides an elegant way of reparametrizing almost any deep network that reduces the problem of coordinating updates across many layers. Let $\boldsymbol{H}$ be a mini-batch of activations of the layer to normalize arranged as a design matrix. To normalize $\boldsymbol{H}$, we replace it with

$$\hat{\boldsymbol{H}} = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

where $\boldsymbol{\mu}$ is a vector containing the mean of each unit and $\boldsymbol{\sigma}$ is a vector containing the standard deviation of each unit.

The normalization of a unit can reduce the expressive power of the neural network that contains it. To avoid this, it is common to replace the batch of hidden unit activations $\boldsymbol{H}$ with $\gamma\hat{\boldsymbol{H}} + \beta$ rather than simply the normalized $\hat{\boldsymbol{H}}$. The variables γ and β are learned parameters that allow the new variable to have any mean and standard deviation. This new parametrization can represent the same family of functions as the old parametrization, but with different learning dynamics. Instead of depending on complicated interactions between the parameters in the layers below, the mean of the new parametrization is determined solely by β, which makes it much easier to learn with gradient descent.

2 Language Modeling

This exercise explores the code from the `word_language_model` example in *PyTorch*.

(a) Go through the code and draw a block diagram / flow chart which highlights the main interacting components, illustrates their functionality, and provides an estimate of their computational time percentage (rough profiling).

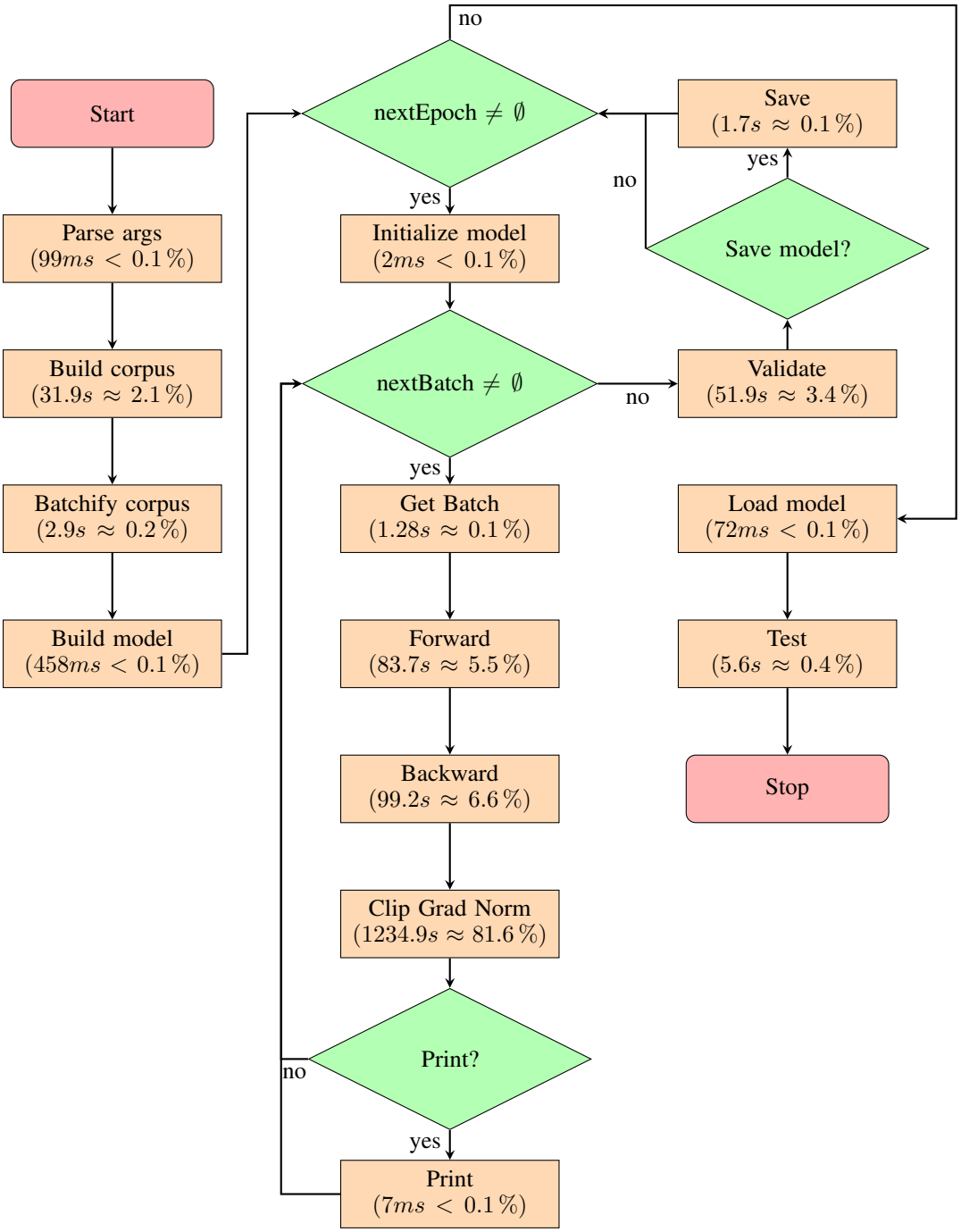

Figure 1: Flow diagram for the `word_language_model` application. The percentage execution times were calculated by running `main.py` on a Tesla K80 GPU for 10 epochs and all other parameters set to default values.

Link to the Overleaf Project.

In addition to the time percentages shown in the flow chart, which were generated by manually adding `time.time()` commands through out the code, figure 2 shows the profiling results obtained with `cProfile`. In both cases we can see that most of the running time is spent in clipping the gradient and updating the parameters of the model, followed by the backward and forward passes of the training stage.

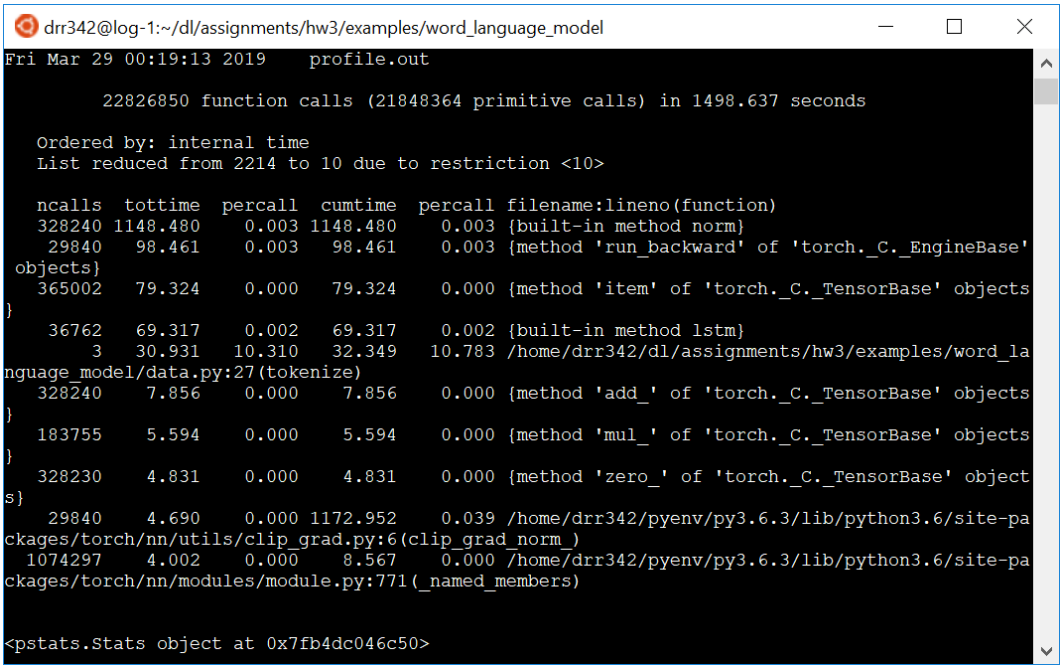

Figure 2: Profiling results using the `cProfile` command (Top 10 results sorted by time are shown).

(b) Find and explain where and how the back-propagation through time (BPTT) takes place (you may need to delve into *PyTorch* source code).
In the original source code, these lines are the ones responsible for the forward and backward propagation of the model. In particular, BPTT (as well as regular back-propagation) takes place when we call `loss.backward()`. This function calculates the gradient of the loss with respect to all the relevant variables in the computational graph and updates the `.grad` field for all of them. What makes BPTT different (which is explained in detail in the next question) is that we do not back-propagate the dependencies of the hidden states until time zero, but only for a fixed number of time steps. This will limit the scope of the model's memory but will mitigate the effects of vanishing or exploding gradients.
The way this works is by means of the `detach()` function, which is at the core of `repackage_hidden()`. When we detach the hidden state tensor before the forward pass of the model, we are effectively setting its `grad_fn` attribute to `None` (The definition of `detach()` in *PyTorch* source code can be found here, but the actual implementation is written in C++. The attribute `grad_fn` is a reference to the computational graph that was used to calculate the tensor. Therefore, when we set it to `None` the tensor forgets its history (how it was created) and gradient can no longer be backpropagated through it (this can be seen in the *PyTorch* source code here).

(c) Describe why we need the `repackage_hidden(h)` function, and how it works.
As it is briefly explained in the code comments, at the beginning of each batch is necessary to detach the hidden state from the computational graph that produced it, i.e. generate a copy of the tensor that does not have any references as to how it was generated and therefore gradient cannot be backpropagated through it. If we don't do this, the model would try to backpropagate through time all the way to the beginning of the dataset.
The way `repackage_hidden(h)` works is quite simple: if `h` is a tensor, it returns `h.detach()`. Otherwise, it returns a tuple containing all the detached versions of the

elements in `h` (so for the function to work `h` must be either a tensor or a collection of tensors).

(d) Why is there a `--tied (tie the word embedding and softmax weights)` option? This option is used to tie together the weights associated to the encoder module (word embeddings) and the weights associated to the decoder module (hidden linear layer). "Tying together" basically means that both modules will share the same weights. In order to use this option, the dimension of the word embeddings (`--emsize`) and the number of hidden units in the linear layer (`--nhid`) must coincide. For higher dimensions, tying these weights reduces considerably the size of the model (making it faster to train and evaluate) while yielding similar results.

(e) Compare LSTM and GRU performance (validation perplexity and training time) for different values of the following parameters: number of epochs, number of layers, hidden units size, input embedding dimensionality, BPTT temporal interval, and non-linearities (pick just 3 of these parameters and experiment with 2-3 different values for each).
Table 1 shows the results (accumulated training time, final validation perplexity and best validation perplexity) of 27 experiments for each model type. The modified parameters in each experiment were input embedding dimensionality, number of layers and BPTT temporal interval. The models were trained for 10 epochs and all other parameters were set to the default values. To obtain comparable results, all the experiments were run on Tesla K80 GPUs available on NYU's Prince HPC Cluster.

Table 1: Models Comparison

Model	`emsize`	`nlayers`	`bptt`	Training Time (s)	Validation Perplexity
LSTM	100	1	16	1 450	141.40
LSTM	100	1	32	1 240	131.63
LSTM	100	1	64	**1 090**	146.32
LSTM	100	2	16	1 730	138.73
LSTM	100	2	32	1 580	154.84
LSTM	100	2	64	1 240	140.03
LSTM	100	4	16	2 540	147.11
LSTM	100	4	32	2 150	159.18
LSTM	100	4	64	1 970	**166.24**
LSTM	200	1	16	1 490	140.48
LSTM	200	1	32	1 210	130.17
LSTM	200	1	64	1 140	141.92
LSTM	200	2	16	1 820	158.80
LSTM	200	2	32	1 490	**127.78**
LSTM	200	2	64	1 220	140.41
LSTM	200	4	16	2 580	134.58
LSTM	200	4	32	2 140	164.88
LSTM	200	4	64	2 150	161.78
LSTM	400	1	16	1 630	142.58
LSTM	400	1	32	1 380	133.41
LSTM	400	1	64	1 160	141.07
LSTM	400	2	16	1 930	134.58
LSTM	400	2	32	1 580	140.64
LSTM	400	2	64	1 360	137.95
LSTM	400	4	16	**2 720**	150.73
LSTM	400	4	32	2 250	138.40
LSTM	400	4	64	2 100	155.09
GRU	100	1	16	1 390	216.92
GRU	100	1	32	1 170	142.91
GRU	100	1	64	**1 080**	134.84

Model	emsize	nlayers	bptt	Training Time (s)	Validation Perplexity
GRU	100	2	16	1 660	256.56
GRU	100	2	32	1 440	249.34
GRU	100	2	64	1 210	133.33
GRU	100	4	16	2 400	1 052.29
GRU	100	4	32	2 090	1 013.71
GRU	100	4	64	1 840	982.62
GRU	200	1	16	1 550	183.94
GRU	200	1	32	1 330	133.61
GRU	200	1	64	1 090	135.74
GRU	200	2	16	1 760	218.30
GRU	200	2	32	1 510	253.57
GRU	200	2	64	1 260	132.14
GRU	200	4	16	2 480	347.48
GRU	200	4	32	2 070	1 035.86
GRU	200	4	64	1 840	1 040.37
GRU	400	1	16	1 610	169.88
GRU	400	1	32	1 290	134.28
GRU	400	1	64	1 150	137.01
GRU	400	2	16	1 880	230.05
GRU	400	2	32	1 490	138.71
GRU	400	2	64	1 310	**131.29**
GRU	400	4	16	**2 530**	1 024.61
GRU	400	4	32	2 160	**1 088.03**
GRU	400	4	64	1 870	977.15

(f) Why do we compute performance on a test set as well? What is this number good for?
The performance metrics (loss and perplexity) measured on the test set are the ones that must be reported as the results of our model because they are the truthful indicators of how good the model generalizes (learns). The metrics obtained on the training and validation sets are not good for this because they are based on data that was used to learn parameters and tune hyperparameters (e.g. learning rate) and therefore tend to be positively biased. The test set performance metrics can be used, for example, to compare two or more methods.