Deep Learning Assignment 2

by Furious Logic

Kelly Zhang	Vikram Mullachery	Nikita Nangia
kz918@nyu.edu	mv333@nyu.edu	nn1119@nyu.edu

April 7, 2017

1 Batch Normalization

1. Let $k \in [1, d]$, where d represents the number of input dimensions and let $j \in [1, m]$, where m is the number of examples in the batch. The mean of x_k across the batch is

$$\mu_k = \frac{1}{m} \sum_{j=1}^{m} x_k^j$$

.

The variance is

$$\sigma_k^2 = \frac{1}{m} \sum_{j=1}^{m} (x_k^j - \mu_k)^2$$

A input value, x_k^j, is normalized by subtracting it from it's feature mean, μ_k, and then dividing the result by it's feature standard deviation (square root of variance). Let $\widehat{x_k^j}$ denote x_k^j normalized:

$$\widehat{x_k^j} = \frac{x_k^j - \mu_k}{\sigma_k}$$

2.

$$y_k^j = \gamma_k \widehat{x_k^j} + \beta_k$$

Using chain rule of partial differentials, we get:

$$\frac{\partial E}{\partial \gamma^k} = \widehat{x_i^k} \frac{\partial E}{\partial y_i^k}$$

and

$$\frac{\partial E}{\partial \beta^k} = \frac{\partial E}{\partial y_i^k}$$

2 Convolution

1. The convolution will create a $3x3$ matrix. The dimensions of the output are computed as $output\_dim = (image\_dim - filter\_dim)/stride + 1 = (5-3)/1 + 1 = 3$.

2. Convolution result by overlaying the filter at successive positions over the input and computing the dot products and summing the resulting matrix entries:
$$\begin{bmatrix} 158 & 183 & 172 \\ 229 & 237 & 238 \\ 195 & 232 & 244 \end{bmatrix}$$

3. The gradient with respect to the input (image) that will be backpropagated to the next layer is a measure of how much the a change in the input values changes the resulting convolution output. Since convolutions are simply a series of scalar multiplications between input values and filter weights, the derivative of the final convolution output with respect to the inputs is simply the filter weights. When a input is multiplied by multiple filter weights, the resulting gradient is simply a sum of the filter weight gradients each input value is multiplied by. Since the gradient passed from the layer above is a 3x3 matrix of all ones, the gradient of the model loss with respect to the inputs is simply the gradient of the layer outputs with respect to the inputs.

$$\begin{bmatrix} 3 & 3+8 & 3+8+3 & 8+3 & 3 \\ 3+2 & 3+8+2+7 & 3+8+3+2+7+9 & 8+3+7+9 & 3+9 \\ 3+2+5 & 3+8+2+7+5+0 & 3+8+3+2+7+9+5+0+2 & 8+3+7+9+2+0 & 3+2+2 \\ 2+5 & 2+7+5+0 & 2+7+9+5+0+2 & 7+9+0+2 & 9+2 \\ 5 & 5+0 & 5+0+2 & 0+2 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 11 & 14 & 11 & 3 \\ 5 & 20 & 32 & 27 & 12 \\ 10 & 25 & 39 & 29 & 14 \\ 7 & 14 & 25 & 18 & 11 \\ 5 & 5 & 7 & 2 & 2 \end{bmatrix}$$

3 Variants of Pooling

1. There are three main kinds of pooling implemented in pytorch:

   - MAXPOOL: Outputs the maximum over the input values (switch module).

- AVGPOOL: Outputs the average of the input values.
- LPPOOL: Outputs the L^p norm of the input values.

2. Here are the mathematical formulas for these pooling techniques. Let x_1 to x_n represent the n input kernel values.

   - MAXPOOL: $max(x_1, ..., x_n)$
   - AVGPOOL: $(x_1 + ... + x_n)/n$
   - LPPOOL: $(x_1^p + x_1^p + ... + x_n^p)^{1/p}$

3. Max pooling, when employed in a deep (multi-layered) model, is important for modeling compositional structure as max pooling can act as a simple feature "detector". Max pooling over a kernel throws away information about what the average kernel value is and simply passes information regarding the maximal input value in a kernel region. The next layer is then able to use these max values as a way to tell whether a certain set of characteristics (defined by the earlier layers) was present in each kernel region. Having multiple layers of convolutions and max pooling "detectors" allows a model to determine the presence of certain features and create inputs to the next layer based on combinations of these features, which allows the model to better learn compositional structure and abstraction.

4 t-SNE

1. **Crowding Problem** The problem t-SNE addresses is that of finding useful 2D visualizations of high dimensional data, like word embeddings. The problem of crowding, is that methods like PCA, which simply tries to find the projection that leads to highest variance of the data points, tends to create visualizations in which many of the data points are crowded at the center - which makes the visualization difficult interpret. This crowding tends to happen because projecting to a smaller dimension distorts the distances between points, and dramatically shortens the perceived distances between points orthogonal to the projection line - for example, imagine points uniformly distributed within a sphere, no matter what 2D projection one chooses, the points will always be much more dense in the center.

   t-SNE relieves the crowding problem by interpreting the distances between two points as the probability that they will be neighbors. The problem setting of t-SNE can be summarized as mapping high dimensional points y_i and y_j to some low dimensional points x_i and x_j, such that their high dimensional space distance is "similar" or "proportional" to that in the lower dimensional space. SNE does this by converting points in high dimensional space (y_i, y_j) into pairwise probabilities, q_{ij} between points representing similarity. The corresponding low dimensional points they map to, x_i and x_j, also each have a pairwise probability p_{ij}. T-SNE is trained to minimize

the KL divergence between these probability distributions for all pairs of points, so that the "distances" (as measured by pair-wise probability distributions) between each pair of points in high dimensional space are proportional to those in the low dimensional space.

The t-SNE visualization is created by choosing a point to plot and then choosing it's neighbors by sampling from that point's pairwise probabilities. This way each point's neighbors are very likely to be close in high dimensional space. These "soft" pairwise distances allow us to create sparser lower dimensional visualizations of points through sampling.

In terms of the difference between stochastic neighbor embeddings (SNE) versus t-SNE, SNE minimizes the standard Kullback Leibler divergence $KL(P||Q)$ between distributions conditional pairwise distributions $p_{i|j}$ and $q_{i|j}$. However, since conditional distributions are asymmetric, minimizing the cost function will sometimes lead to poorer visualizations because the optimization procedure is not able to recover from the bias in the cost function, leading to disproportional pairwise probabilities, p_{ij} and q_{ij}. Thus t-SNE uses symmetric, joint probabilities. Moreover, t-SNE also uses a student-T distribution instead of a regular Gaussian distribution in modeling distances in the lower dimensional space, because the fatter tails of the student-T distribution allow moderate distances in the high dimensional space to be mapped to larger distances in the lower dimensional space, alleviating the crowding problem.

2. **Derivation of $\frac{\partial C}{\partial y_i}$**

$$\frac{\partial C}{\partial y_i} = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} = \sum_i \sum_j p_{ij} \log p_{ij} - p_{ij} \log q_{ij}$$

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}}$$

$$p_{ij} = \frac{\exp(-||x_i - x_j||^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2 / 2\sigma_i^2)}$$

Since changing y_i only changes the distances $||y_i - y_j||$ for any j and p_{ij} is a constant (doesn't depend on y_i).

$$\frac{\partial C}{\partial y_i} = \sum_j \left(\frac{\partial C}{\partial ||y_i - y_j||} + \frac{\partial C}{\partial ||y_j - y_i||} \right) (y_i - y_j)$$

By symmetry,

$$= 2\sum_j \frac{\partial C}{\partial ||y_i - y_j||}(y_i - y_j)$$

Let $Z = \sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}$.
Derivative of Kullback-Leiber divergence:

$$KL(P||Q) = \sum_i \sum_j p_{ij} \log p_{ij} - p_{ij} \log q_{ij}$$

$$\frac{\partial C}{\partial ||y_i - y_j||} = -\sum_{k \neq l} p_{kl} \frac{\partial (\log q_{kl})}{\partial ||y_i - y_j||}$$

$$= -\sum_{k \neq l} p_{kl} \frac{\partial (\log q_{kl} Z - \log Z)}{\partial ||y_i - y_j||}$$

Since $q_{kl}Z = (1 + ||y_k - y_l||^2)^{-1}$,

$$= -\sum_{k \neq l} p_{kl} \left(\frac{1}{q_{kl} Z} \frac{\partial (1 + ||y_k - y_l||^2)^{-1}}{\partial ||y_i - y_j||} - \frac{1}{Z} \frac{\partial Z}{\partial ||y_i - y_j||} \right)$$

Since $\frac{\partial (1+||y_k - y_l||^2)^{-1}}{\partial ||y_i - y_j||}$ is only nonzero when $k = i$ and $l = j$,

$$\frac{\partial C}{\partial ||y_i - y_j||} = 2\frac{p_{ij}}{q_{ij} Z}(1 + ||y_i - y_j||^2)^{-2} - 2\sum_{k \neq l} \frac{p_{kl}}{Z}(1 + ||y_i - y_j||^2)^{-2}$$

Since $\sum_{k \neq l} p_{kl} = 1$,

$$\begin{aligned} \frac{\partial C}{\partial ||y_i - y_j||} &= 2\frac{p_{ij}}{q_{ij} Z}(1 + ||y_i - y_j||^2)^{-2} - 2\frac{1}{Z}(1 + ||y_i - y_j||^2)^{-2} \\ &= 2p_{ij}(1 + ||y_i - y_j||^2)^{-1} - 2q_{ij}(1 + ||y_i - y_j||^2)^{-1} \\ &= 2(p_{ij} - q_{ij})(1 + ||y_i - y_j||^2)^{-1} \end{aligned}$$

By plugging in the value of the derivative of the KL divergence into the derivative of the cost, we get:

$$\frac{\partial C}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})(1 + ||y_i - y_j||^2)^{-1}(y_i - y_j)$$

5 Sentence Classification

a. **ConvNet**

1. We are given a sentence representation that is sequence-length by embedding size (10x300). Our convolutional net for sentence classification will have five filters. Each filter will be of size 3x300, to detect 3-grams. After applying these filters with a stride of one, we will have 8 values per filter (40 in total). We will then put these values through a ReLU activation function. For each of these set of 8 values, we will max pool over sequence length to a total of 5 values. After pooling, we will then apply another ReLU activation function.

   After creating the 5 dimensional sentence vector representation we will multiply the vector by a 3x5 dimensional matrix and add a 3 dimensional bias vector. After applying these transformations we will have a 3 dimensional vector that we will then it feed into a softmax function to normalize the values into a probability distribution.

2. Model Dimensions:

   Input size: 10x300 (sequence_length x embedding_size)
   Filter size: 3x300 (n-gram_size x embedding_size)
   Filter count: 5

   After applying filters get 5x8 dimensional matrix (filter_count x (sequence_length - n-gram_size + 1))
   Apply ReLU activation.

   Maxpool over time to get 5x1 dimensional sentence vector
   Apply ReLU activation.

   Multiply sentence vector by trained matrix W (3x5) and add by bias vector B (3x1). As a result get a 3x1 output vector.

   Put vector into softmax function to normalize values. As a result get a 3x1 output vector that sums to 1.

   Sizes of input and outputs for each of the layers is as below. (N is batch size).

Layer	Input	Output
Convolution Filters	$(N, 10, 300)$	$(N, 5, 8)$
ReLU Activation	$(N, 5, 8)$	$(N, 5, 8)$
Max Pool	$(N, 5, 8)$	$(N, 5, 1)$
ReLU Activation	$(N, 5, 1)$	$(N, 5, 1)$
Fully Connected(Linear)	$(N, 5, 1)$	$(N, 3, 1)$
Softmax	$(N, 3, 1)$	$(N, 3, 1)$

3. When applying convolutions to sequences of word embeddings, it is common to set one dimension of equal to the embedding size, as it makes sense to have the filters see all the dimensions representing a given word. Thus in terms of filter size, we mean the dimension that will be shifted along the sequence length. A larger filter size means that the filter essentially detects a larger n-gram, and a smaller filter size means that the filter detects a smaller n-gram. When choosing n-gram sizes, generally a very large filter size (that is comparable to sentence length) is generally too large, while too small n-grams (size of 1 or 2), may not be long enough to detect meaningful enough word patterns.

   In terms of selecting filter sizes for the classification task, we would run the model with different filter sizes and see which filter sizes detect the best word features. We can also experiment with applying filters of different sizes within the same model.

b. **RNN**

1. A few simple ways to use an RNN trained for language modeling to get a sentence vector are:

   a) Use the last hidden state of the RNN after reading it with the language model RNN as the sentence vector.
   b) Taking the outputs (hidden states) of the language model RNN and taking an average over time to get a sentence vector.
   c) Taking the outputs (hidden states) of the language model RNN and taking a max-pool over time for each embedding dimension to get a sentence vector.
   d) Take both and average and a max-pool, and concatenate the two outputs.

   The result of the averaging/max-pooling/concatenation would then be fed into a shallow MLP and then a Softmax classifier.

2. RNN with a sequence length (bptt) of 10, and 50 hidden units. So at the end of applying the affine and *tanh* non-linearity for 10 (sequence length, or bptt) time steps, the hidden units would have

encoded the sentence of dimension 50. The fully connected layer applies linear transformation with 4 output units. A Softmax classifier then computes the probability distribution across these 4 classes.

The RNN will have a 4*hidden_size = 4*50 = 200 by (hidden_size + input_size) trainable matrix and a 4*hidden_size = 200 dimensional bias vector (4 because each gates of the LSTM requires a hidden_size dimensional vector - input, forget, output, candidate).

The fully connected linear layer before the softmax layer will have a matrix of size 4x50 and a bias vector of dimension 4.

3. Network details:
Following units recur in time:

Layer	Description	Input Size	Output Size
1	One-hot to embedding	one-hot vector (vocab_size)	1 x embedding_size
2	Affine & tanh (RNN hidden unit)	1 x embedding_size	50

Following units apply at the end of sequence length words (10 words):

Layer	Description	Input Size	Output Size
3	Fully Connected (Linear)	50	4 (target classes)
4	Softmax	4	4 (probability distribution)

c. **fastText** Following is a brief comparative listing of spearman correlation score for fastText on the Penn TreeBank with a few of our models from LSTM language models below:

Model	Params for Training	Spearman Score
fastText	skipgram, context-window=5, epochs=5, dim=100, lr=0.1 (default setting)	26
LSTM	nlayers=3, epochs=20, emsize=100, nhid=100, dropout=0	23
LSTM	nlayers=3, epochs=20, emsize=600, nhid=600, dropout=0.2	18

fastText is blazingly fast in comparison to the LSTM. It ran in under 3 minutes on Prince HPC cluster and the Spearman score on word similarity test is comparable to an LSTM network that ran for over 2 hours for 20 epochs.

d. **Extra** Another model design for sentence classification is that combines both recurrent and convolutional neural networks is to apply convolutional and recurrent layers. Instead of using a hidden state that is a function of the previous time step's hidden state, we could have a hidden state that is a function of applying a series of convolutional filters on the input embeddings.

   For example, we could have an embedding size of 100 and a hidden size of 100. The hidden state would be the output of applying 100 convolutional filters on the previous timestep's inputs to get a 100 dimensional vector.

   The advantage of having a convolutional neural network compute the hidden state direction from the previous timestep's inputs is that it makes the LSTM network more parallelizable, as the hidden state at any timestep does not depend on the previous time step's hidden state, so multiple timesteps could be process in parallel. The limitation to using convolutional filters it that they only detect n-grams and we may need to train many n-gram filters to get a good hidden state representation.

6 Language Modeling

1. **Models:** We explored LSTM models with various hyper-parameters with the Penn Tree Bank and Gutenberg datasets. We also briefly ran a GRU model, without exploring any hyper-parameters variations there, because the initial results were summarily poorer than the LSTM.
   In addition, we attempted a GatedCNN model (Dauphin et al., 2016) and a GatedCNN model with an RNN stacked on top. Neither of these performed comparably to the recurrent models we tested. We hypothesize that the poor results could be a result of inadequate hyper-parameter tuning and the fact that a single kernel-width was used (therefore the model only explore 3-grams).
   These are the hyper-parameter settings and results for our various models:

Model	Dataset	Hyperparameters	Test Perplexity
GRU	PTB	nlayers=1,epochs=20,emsize=50,nhid=50	193.7
LSTM	PTB	nlayers=1,epochs=20,emsize=50,nhid=50	136.08
LSTM	PTB	nlayers=1,epochs=40,emsize=650,nhid=650, dropout=0.5,tied	139.66
LSTM	PTB	nlayers=3,epochs=20,emsize=50,nhid=50	132.77
LSTM	PTB	nlayers=3,epochs=20,emsize=50,nhid=100	120.44
LSTM	PTB	nlayers=3,epochs=20,emsize=100,nhid=100	117.51
LSTM	PTB	nlayers=3,epochs=20,emsize=100,nhid=100, dropout=0.5,tied	143.29
LSTM	PTB	nlayers=5,epochs=20,emsize=100,nhid=100, bptt=10	130.73
LSTM	PTB	nlayers=2,epochs=40,emsize=250,nhid=250, bptt=35, lr=5, tied	108.07
LSTM	PTB	nlayers=2,epochs=40,emsize=250,nhid=250, bptt=35, lr=5, dropout=0.5, tied	87.12
LSTM	PTB	nlayers=2,epochs=40,emsize=1000,nhid=1000, bptt=35, lr=10, dropout=0.5, tied	82.00
LSTM	PTB	nlayers=2,epochs=40,emsize=1000,nhid=1000, bptt=35, lr=10, dropout=0.5	86.99
LSTM	PTB	nlayers=2,epochs=40,emsize=1000,nhid=1000, bptt=35, lr=10, tied	111.82
LSTM	PTB	nlayers=2,epochs=40,emsize=1000,nhid=1000, bptt=35, lr=10, dropout=0.5, tied	79.12
LSTM	Gutenberg	nlayers=2,epochs=40,emsize=250,nhid=250, bptt=35, tied, lr=10	122.93
LSTM	Gutenberg	nlayers=2,epochs=40,emsize=1000,nhid=1000, bptt=35, tied, lr=10	105.89
GatedCNN	PTB	nlayers=10, epochs 20, emsize=100, nhid=100,bptt=20, batch-size=32, lr 10	199.26
GatedCNN + RNN	PTB	CNN-layers=10, RNN-layers=2, epochs 30, emsize=100, nhid=100,bptt=20, batch-size=32, dropout=0.2, lr 10	180.40
Character-Aware LM (Char-CNN LSTM)	PTB	# CNN filters=7, # Feature maps=7, RNN-layers=2, emb-size=100, char nhid=15, word nhid=300, Highway=1, batch-size=20, bptt-30, dropout=0.5, lr=1	105.16

2. **Hyper-parameter settings** The parameters are tabulated above. Our best performing model on Penn Treebank and Gutenberg datasets was a 2-layer LSTM with 1000 hidden units and embedding size, with encoder and decoder weights being tied.

We found that in general increasing the size of the model increased performance. Test perplexity decreased by increasing the number of layers from 1 to 3. However, the marginal improvement from 3 to 5 was lower, especially given the drastic increase in training time. We found that on smaller models (hidden size of about 100), adding dropout actually made performance drop; on larger models however, dropout was crucial to prevent overfitting and getting better performance. We also tried different optimizers (ADAM and SGD with momentum), but found that it took longer for the model to train with these optimizers and they did not yield lower perplexities in the first few epochs compared to regular SGD, so we used regular SGD for the majority of our experiments.

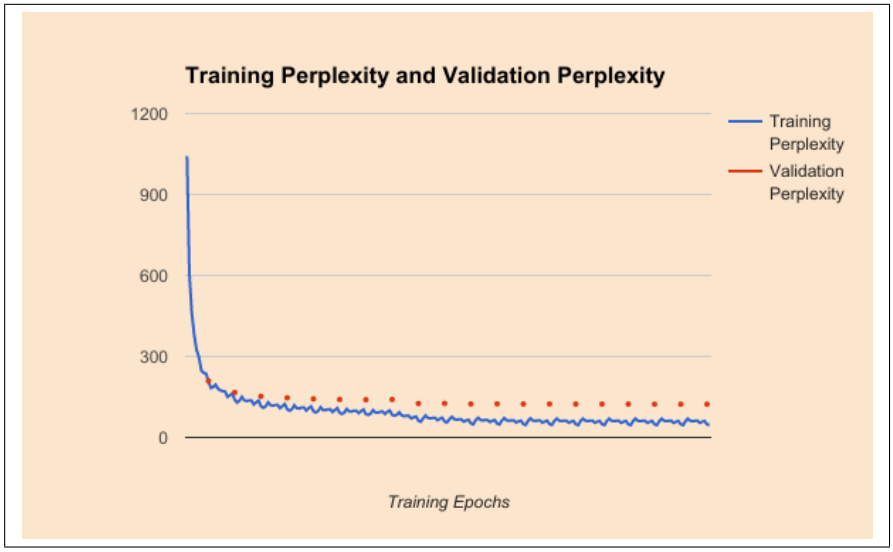
We also experimented with the relative sizes of the embedding and hidden states. We thought that a smaller embedding size and a larger hidden state would make sense as the word embeddings only encode words, while hidden states must have information regarding the whole sentence. We tried some models with the embedding size that was about half that of the hidden size (for hidden size of 250), but found these models to perform significantly worse than those for which the embedding size was equal to that of the hidden size. As described in Zaremba et. al [2], we only applied dropout between layer to layer connections and not timestep to timestep connections of the RNN.

We also found that tying the weights between the embeddings and output softmax linear transformation (for when the hidden size equals the embedding size), improved performance by about 5 perplexity points.

In general in our training, we employed learning rate decay of 0.25; whenever the validation loss increased, we would decrease the learning rate by 0.25. Our starting learning rate was 20.0 unless stated otherwise.

We also played with the sequence length (how many timesteps to backpropagate the gradients) and found that increasing this parameter from 10 to 35 also improved performance by a few perplexity points.

3. **Evaluation** The performance of the models were evaluated on the perplexity score (exponential of average cross entropy loss / negative log likelihood per example). Following is a plot of the learning curve showing the training perplexity and validation perplexity for the best performing model from the table above:

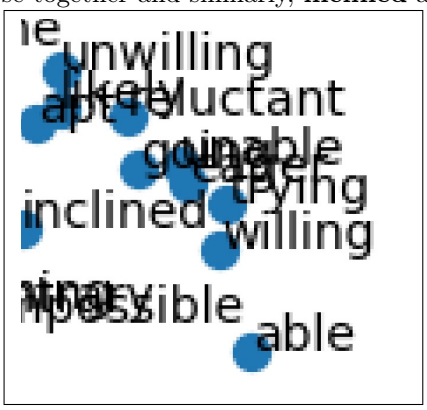


4. **Analysis and embeddings plots**

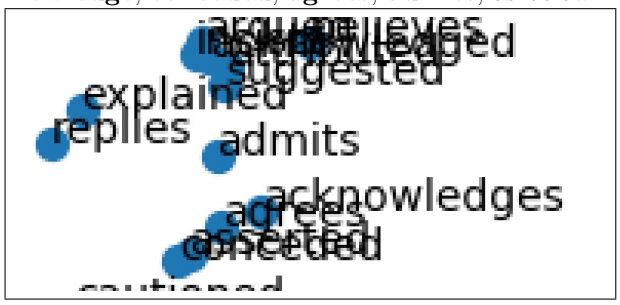
   a We saw improvements in test perplexity by increasing the number of layers from 1 to 3. However, the marginal improvement from 3 to 5 was lower, especially given the drastic increase in training time

   b Larger word embedding dimensions appear to capture the variational uses of a word and thus improve the language model. Notice the improvement in perplexity (a reduction in perplexity value), for the 3 layer network while word embedding is gradually increased from 50 up to 600. However, an empirical observation here is that the hidden unit size needs to correspondingly increase (or match) in size for the improvement to be noticed in the model.

   c Shown are a few highlights from the word embedding plots
   First, a clustering of adjectives. These show words **unwilling** and **reluctant** close together and similarly, **inclined** and **willing**:



Second, a clustering of words with close synonym pairings: **acknowledge**, **conceded**, **agrees**, **admits**, as below:



5. **Variants of Recurrent Language Model**
   One experiment we performed was to replicate the attention mechanism employed in Daniluk et al. [3]. We did not find however that adding an attention context to improve performance, as they did in their paper when experimenting on the Penn Treebank. Some reasons for this are that we only employed attention for words within the training example (did not span past to previous training examples for engineering simplicity).

   We also experimented with a model that took into account previous sentence context as a bag-of-words representation. We concatenated the continuous bag-of-words of the previous sentence to the output vector, before multiplying by the prediction matrix and performing the softmax prediction. We ran this "context" model on the Penn Treebank dataset but found that it's performance was slightly worse than that of our best Penn Treebank model by about 5 perplexity points.

   - **Variants using CNNs** We also attempted a model that used layers of gated ConvNet units (Dauphin et al., 2016). The gating was performed with a sigmoid function, as in LSTM units. We found that on the PTB dataset, and with the hyperparameter settings that were tried, the model did not perform as well as the recurrent models.
   - We built a character-aware language model in the vein of Kim et al. (2016). The model uses a character level CNN to create word embeddings that get fed into an LSTM. The model performed slightly worse than the small model presented in the paper but still didn't beat our best, purely recurrent based model.

References

[1] L.J.P van der Maaten and G.E. Hinton. Visualizing high-dimensional data using t-sne. Journal of Machine Learning Research, 9: 25792605, Nov 2008.

[2] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. CoRR, abs/1409.2329, 2014.

[3] Michał Daniluk, Tim Rocktäschel, Johannes Welbl, and Sebastian Riedel. Frustratingly Short Attention Spans in Neural Language Modeling. ICLR 2017. https://arxiv.org/abs/1702.04521

[4] Yann N. Dauphin, Angela Fan, Michael Auli, and David GrangierMichał. Language Modeling with Gated Convolutional Networks. *arXiv:1612.08083*

[5] Yoon Kim, Yacine Jernite, David Sontag, Alexander M. Rush. Character-Aware Neural Language Models. *https://arxiv.org/pdf/1508.06615.pdf*