Deep Learning: Assignment Two

Aditi Nair (asn264) and Akash Shah (ass502)

April 4, 2017

1   Batch Normalization

1. Let $x_1, ... x_m$ be m data points, and let x_i^k be the k-th feature of the i-th data point. Then we define the mean μ_k, of feature x^k, over a mini-batch as:

$$\mu_k = \frac{1}{m} \sum_{i=1}^{m} x_i^k$$

and the variance σ_k^2 as:

$$\sigma_k^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i^k - \mu_k)^2$$

Now to normalize each feature x^k (for $1 \le k \le n$) to have zero mean and unit standard deviation, we compute for each mini-batch value of that feature x_i^k, for $1 \le i \le m$:

$$\hat{x}_i^k = \frac{x_i^k - \mu_k}{\sigma_k}$$

Then over all $\hat{x}_i^k$ per k, the expected value is 0:

$$E[\hat{x}_1^k, ..., \hat{x}_m^k] = \frac{1}{m} \sum_{i=1}^{m} \hat{x}_i^k = \frac{1}{m} \sum_{i=1}^{m} \frac{x_i^k - \mu_k}{\sigma_k} = \frac{1}{m} \frac{1}{\sigma_k} \sum_{i=1}^{m} (x_i^k - \mu_k)$$

$$= \frac{1}{m} \frac{1}{\sigma_k} \left[\left(\sum_{i=1}^{m} x_i^k \right) - m \cdot \mu_k \right] = \frac{1}{m} \frac{1}{\sigma_k} \left[\sum_{i=1}^{m} x_i^k - \sum_{i=1}^{m} x_i^k \right] = 0$$

Then over all $\hat{x}_i^k$ per k, the variance is 1 since:

$$Var[\hat{x}_1^k, ..., \hat{x}_m^k] = \frac{1}{m} \sum_{i=1}^{m} (\hat{x}_i^k - \hat{\mu}_k)^2 = \frac{1}{m} \sum_{i=1}^{m} (\hat{x}_i^k - 0)^2$$

$$= \frac{1}{m} \sum_{i=1}^{m} (\hat{x}_i^k)^2 = \frac{1}{m} \sum_{i=1}^{m} \left(\frac{x_i^k - \mu_k}{\sigma_k} \right)^2$$

$$= \frac{1}{m} \sum_{i=1}^{m} \frac{(x_i^k - \mu_k)^2}{\sigma_k^2} = \frac{1}{m \cdot \sigma_k^2} \sum_{i=1}^{m} (x_i^k - \mu_k)^2$$

$$= \frac{m}{m \cdot \sum_{i=1}^{m} (x_i^k - \mu_k)^2} \sum_{i=1}^{m} (x_i^k - \mu_k)^2 = 1$$

2. For x_i^k, the k-th feature of the i-th data point, the output of the BN module can be written as:

$$y_i^k = BN_{\gamma^k, \beta^k}(x_i^k) = \gamma^k \hat{x_i}^k + \beta^k$$

with

$$\hat{x_i}^k = \frac{x_i^k - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

μ_k and σ_k are defined as above. For numerical stability, the BN algorithm adds ϵ to σ_k^2 in the denominator of $\hat{x_i}^k$ before taking the square root.

Now we write $\frac{\delta E}{\delta \gamma^k}$ in terms of $\frac{\delta E}{\delta y_i^k}$:

γ^k appears in the computation of each y_i^k for $1 \leq i \leq m$. It follows that:

$$\frac{\delta E}{\delta \gamma^k} = \sum_{i=1}^{m} \frac{\delta E}{\delta y_i^k} \frac{\delta y_i^k}{\delta \gamma^k}$$

Then we notice that:

$$\frac{\delta y_i^k}{\delta \gamma^k} = \frac{\delta(\gamma^k \hat{x_i}^k + \beta^k)}{\delta \gamma^k} = \hat{x_i}^k$$

Finally, we can write:

$$\frac{\delta E}{\delta \gamma^k} = \sum_{i=1}^{m} \frac{\delta E}{\delta y_i^k} \hat{x_i}^k$$

Now we write $\frac{\delta E}{\delta \beta^k}$ in terms of $\frac{\delta E}{\delta y_i^k}$:

β^k appears in the computation of each y_i^k for $1 \leq i \leq m$. It follows that:

$$\frac{\delta E}{\delta \beta^k} = \sum_{i=1}^{m} \frac{\delta E}{\delta y_i^k} \frac{\delta y_i^k}{\delta \beta^k}$$

Then we notice that:

$$\frac{\delta y_i^k}{\delta \beta^k} = \frac{\delta(\gamma^k \hat{x_i}^k + \beta^k)}{\delta \beta^k} = 1$$

Finally, we can write:

$$\frac{\delta E}{\delta \beta^k} = \sum_{i=1}^{m} \frac{\delta E}{\delta y_i^k}$$

2 Convolution

1. Assuming a stride of 1 and no zero padding, we will get a 3×3 matrix of 9 values.

2. SHOW WORK?

   Assuming a stride of 1, no zero padding and an additive filter, the result of this convolution is:

   $$\begin{bmatrix} 158 & 183 & 172 \\ 229 & 237 & 238 \\ 195 & 232 & 244 \end{bmatrix}$$

3. SKIPPED

3 Variants of Pooling

1. Three types of pooling are max-pooling, average-pooling and L-P pooling. Max-pooling is implemented in different dimensions by the MaxPool1d, MaxPool2d and MaxPool3d classes in PyTorch. Average-pooling is implemented in different dimensions by the AvgPool1d, AvgPool2d, and AvgPool3d classes in PyTorch. L-P pooling is implemented in by the LPPool2d module in PyTorch.

2. Now we define the mathematical forms of these pooling operations over a given set of input values $x_1, ..., x_n$. Max pooling returns the maximum over all inputs:

$$MaxPool(x_1, ..., x_n) = max(x_1, ..., x_n)$$

Average pooling computes the average of all the inputs:

$$AveragePool(x_1, ..., x_n) = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$L - p$ pooling computes the power-average of degree p over the inputs:

$$LpPool(x_1, ..., x_n) = \left(\sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}}$$

3. Max-pooling essentially applies a maximum function over (sometimes non-overlapping) regions of input matrices. For inputs with several layers of depth, max-pooling is generally applied separately for each layer's matrix. The pooling operation is used to reduce the size of the input matrices in each layer. Applying max-pooling will select only the maximum value in each region of the input matrix, discarding smaller values in each region. This can prevent overfitting by discarding information from the smaller values. In comparison, operations like average-pooling weigh all values in the region equally - and are therefore more conservative in discarding additional information. An average pooling operation in a region with many small values and one large value will have a relatively small output compared to the max pooling operation over the same region, which ignores the overall tendency of values in the region. Therefore the aggressive approach of the max pooling operation can be used as a regularizer.

4 t-SNE

1. The crowding problem refers to the tendency of dimensionality reduction techniques to crowd points of varying similarity close together. Generally, SNE techniques model pairwise distances in high-dimensional space by a probability distribution P, and pairwise distances in the low-dimensional space by a probability distribution Q. In order to ensure that the low-dimensional representation is faithful to important properties of the original representation, SNE optimizes the KL-divergence between P and Q so that the pairwise distances in the low-dimensional space are as similar as possible to those in the original high-dimensional space.

Maarten and Hinton (2008) provide the following examples to illustrate the crowding problem. Given a set of points which lie on a two-dimensional manifold in high-dimensional space, it is fairly straightforward to effectively describe their pairwise distances with a two-dimensional map. On the other hand, suppose these points lie on a higher-dimensional manifold - then it becomes much more difficult to model pairwise distances in two dimensions. Maarten and Hinton provide an example on a 10-dimensional manifold, where it is possible to have ten points which are mutually equidistant - this is clearly impossible to map into a two-dimensional space.

More generally, the distribution of possible pairwise distances in the high-dimensional space is very different than the distribution of possible pairwise distances in the two-dimensional space. Consider a set of points in high-dimensional space which are uniformly distributed around a central point. As the distance of the points from the central point grows, the volume of space these points could

occupy also grows. However in two dimensions there is less area to accommodate points which are moderately far from the center than there is to accommodate points which are near to the center. Therefore, if we attempt to model small distances accurately in the two-dimensional space, we will be forced to place moderately large distances much further from the center point. Now, when trying to optimize the two-dimensional mapping, these too-far points will be pushed inward by the SNE objective function, effectively crowding all of the points together in the two-dimensional mapping.

t-SNE alleviates this by representing distances between points as probabilities using specific distributions. Distances in the high-dimensional spaces are converted to probabilities using a Gaussian distribution, whereas distances in the low-dimensional spaces are converted to probabilities using a distribution with a much heavier tail. This way, moderately far points are assigned larger distances in the lower-dimensional mapping compared to when the lower-dimensional distances are computed using a small-tailed distribution. Then the distances between pairs of moderately far points in higher dimensional space are closer to the distances between the same pairs of points in low dimensional space, and the SNE objective function will not push the low-dimensional representations closer together as much, ameliorating the crowding problem.

2. Let

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

We want to compute $\frac{\delta C}{\delta y_i}$. Note that p_{ij} is constant with respect to y_i but q_{ij} is not. Specifically:

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

Following the derivation of van der Maaten and Hinton, we define two intermediate values:

$$d_{ij} = ||y_i - y_j||$$

and

$$Z = \sum_{k \neq l}(1 + d_{kl}^2)^{-1}$$

Now we can rewrite q_{ij} as:

$$q_{ij} = \frac{(1 + d_{ij}^2)^{-1}}{\sum_{k \neq l}(1 + d_{kl}^2)^{-1}} = \frac{(1 + d_{ij}^2)^{-1}}{Z}$$

In the new notation, all terms are constant with respect to y_i except d_{ij} and d_{ji} for all j. Now, we can compute the partial derivative $\frac{\delta C}{\delta y_i}$ in terms of the partial derivatives $\frac{\delta C}{\delta d_{ij}}$ and $\frac{\delta C}{\delta d_{ji}}$ for all d_{ij} and d_{ji}. In particular, we notice that the terms d_{ji} and d_{ij} appears once each for all possible value of j. That is:

$$\frac{\delta C}{\delta y_i} = \sum_j \frac{\delta C}{\delta d_{ij}} \frac{\delta d_{ij}}{\delta y_i} + \sum_j \frac{\delta C}{\delta d_{ji}} \frac{\delta d_{ji}}{\delta y_i}$$

First, we compute $\frac{\delta d_{ij}}{\delta y_i}$. For $x, y \in \mathbb{R}^n$, notice that:

$$||x - y|| = \sqrt{(x_1 - y_1)^2 + ... + (x_n - y_n)^2}$$

Then it follows that:

$$\frac{\delta\Big(||x - y||\Big)}{\delta x} = \Big\langle \frac{\delta\Big(||x - y||\Big)}{\delta x_1}, ..., \frac{\delta\Big(||x - y||\Big)}{\delta x_n} \Big\rangle^T$$

4

$$= \left\langle \frac{2(x_1 - y_1)}{2||x - y||}, ..., \frac{2(x_n - y_n)}{2||x - y||} \right\rangle^T$$

$$= \frac{1}{||x - y||} \left\langle x_1 - y_1, ..., x_n - y_n \right\rangle^T = \frac{x - y}{||x - y||}$$

Since $d_{ij} = ||y_i - y_j||$, it follows that:

$$\frac{\delta d_{ij}}{\delta y_i} = \frac{\delta \Big(||y_i - y_j|| \Big)}{\delta y_i} = \frac{y_i - y_j}{||y_i - y_j||}$$

Moreover since $d_{ji} = ||y_j - y_i|| = ||y_i - y_j|| = d_{ij}$, we can say that:

$$\frac{\delta d_{ji}}{\delta y_i} = \frac{y_i - y_j}{||y_i - y_j||}$$

Finally:

$$\frac{\delta C}{\delta y_i} = \sum_j \frac{\delta C}{\delta d_{ij}} \frac{y_i - y_j}{||y_i - y_j||} + \sum_j \frac{\delta C}{\delta d_{ji}} \frac{y_i - y_j}{||y_i - y_j||}$$

$$= \sum_j \Big(\frac{\delta C}{\delta d_{ij}} + \frac{\delta C}{\delta d_{ji}} \Big) \frac{y_i - y_j}{||y_i - y_j||} = 2 \sum_j \frac{\delta C}{\delta d_{ij}} \frac{y_i - y_j}{||y_i - y_j||} = 2 \sum_j \frac{\delta C}{\delta d_{ij}} \frac{y_i - y_j}{d_{ij}}$$

Next, we need to compute $\frac{\delta C}{\delta d_{ij}}$. Recall that:

$$C = \sum_i \sum_j p_{ij} log\ p_{ij} - p_{ij} log\ q_{ij}$$

and only q_{ij} is non-constant with respect to d_{ij}.

Therefore:

$$\frac{\delta C}{\delta d_{ij}} = \frac{\delta \Big(\sum_i \sum_j p_{ij} log\ p_{ij} - p_{ij} log\ q_{ij} \Big)}{\delta d_{ij}}$$

$$= \frac{-\delta \Big(\sum_i \sum_j p_{ij} log\ q_{ij} \Big)}{\delta d_{ij}}$$

When optimizing t-SNE, we define $p_{ii} = q_{ii} = 0$ for all i. Then we can say:

$$\frac{\delta C}{\delta d_{ij}} = \frac{-\delta \Big(\sum_{k \neq l} p_{kl} log\ q_{kl} \Big)}{\delta d_{ij}}$$

Following, the derivation of van der Maaten and Hinton, we observe that:

$$log\ q_{kl} = log \frac{q_{kl} Z}{Z} = log\ q_{kl} Z - log Z$$

It follows that:

$$\frac{\delta C}{\delta d_{ij}} = \frac{-\delta \Big(\sum_{k \neq l} p_{kl} (log\ q_{kl} Z - log Z) \Big)}{\delta d_{ij}}$$

Notice that q_{kl} and Z are non-constant with respect to d_{ij} and that:

$$q_{kl} Z = \frac{(1 + d_{kl}^2)^{-1}}{Z} Z = (1 + d_{kl}^2)^{-1}$$

Then we can compute $\frac{\delta C}{\delta d_{ij}}$ as:

$$\frac{\delta C}{\delta d_{ij}} = -\sum_{k \neq l} p_{kl} \left(\frac{1}{q_{kl}Z} \frac{\delta(1+d_{kl}^2)^{-1}}{\delta d_{ij}} - \frac{1}{Z}\frac{\delta Z}{\delta d_{ij}} \right)$$

Now we notice that $\frac{\delta(1+d_{kl}^2)^{-1}}{\delta d_{ij}}$ is 0 unless $k = i$ and $l = j$. Then we can write:

$$\frac{\delta C}{\delta d_{ij}} = \frac{-p_{ij}}{q_{ij}Z} \cdot -(1+d_{ij}^2)^{-2} \cdot 2d_{ij} - \sum_{k \neq l} -p_{kl}\frac{1}{Z} \cdot -(1+d_{ij}^2)^{-2} \cdot 2d_{ij}$$

$$= \frac{2p_{ij}}{q_{ij}Z} \cdot (1+d_{ij}^2)^{-2} \cdot d_{ij} - 2\sum_{k \neq l}\frac{p_{kl}}{Z} \cdot (1+d_{ij}^2)^{-2}d_{ij}$$

Since $q_{ij}Z = (1+d_{ij}^2)^{-1}$ we can simplify the left summand further.

$$\frac{\delta C}{\delta d_{ij}} = 2p_{ij} \cdot (1+d_{ij}^2)^{-1} \cdot d_{ij} - 2\sum_{k \neq l}\frac{p_{kl}}{Z} \cdot (1+d_{ij}^2)^{-2}d_{ij}$$

$$= 2p_{ij} \cdot (1+d_{ij}^2)^{-1} \cdot d_{ij} - \frac{2}{Z}\sum_{k \neq l}p_{kl} \cdot (1+d_{ij}^2)^{-2}d_{ij}$$

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

$$\frac{\delta C}{\delta d_{ij}} = 2p_{ij}(1+d_{ij}^2)^{-1}d_{ij} - \frac{2}{Z}(1+d_{ij}^2)^{-2}d_{ij}$$

Since $q_{ij} = \frac{(1+d_{ij}^2)^{-1}}{Z}$, we can simplify the right summand as:

$$\frac{\delta C}{\delta d_{ij}} = 2p_{ij}(1+d_{ij}^2)^{-1}d_{ij} - 2q_{ij}(1+d_{ij}^2)^{-1}d_{ij}$$

$$= 2(p_{ij} - q_{ij})(1+d_{ij}^2)^{-1}d_{ij}$$

Now we can substitute this into our expression for $\frac{\delta C}{\delta y_i}$:

$$\frac{\delta C}{\delta y_i} = 2\sum_j \frac{\delta C}{\delta d_{ij}}\frac{y_i - y_j}{d_{ij}} = 2\sum_j \left(2(p_{ij} - q_{ij})(1+d_{ij}^2)^{-1}d_{ij} \right)\frac{y_i - y_j}{d_{ij}}$$

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

5 Sentence Classification

5.1 ConvNet

1. Given a sentence of 10 words, each word has a trained word embedding vector of dimension 300. Thus, we can construct an input matrix of dimension 10 x 300, where the i-th row is the word embedding of the i-th word in the sentence. Each filter will have length equal to the word embedding size, but can have a varied width (or filter size). If we have 5 filters, with filter sizes of 2, 3, 4, 5, and 6, and each with a length of 300, then by passing each filter over the input matrix and performing convolutions, we will obtain a hidden feature vector for each filter. For a filter with size k, the hidden feature vector will have a length of $10 - k + 1$, so we will obtain 5 feature vectors with lengths 9, 8, 7, 6, and 5.

   By applying max-pooling to each feature vector, each of the 5 vectors will be reduced to a scalar value, and concatenating them will result in a vector of length 5. We can then pass this vector of length 5 into a fully-connected layer with 5 input units and 3 output units. The output of this fully-connected layer will be a vector with 3 scores, one for each class. Finally, we pass the vector of scores into a softmax function to obtain a probability for each class.

2. In a single-layer CNN such as the one described above, small filter sizes can be viewed as small n-gram featurizations of the input but in a dense and distributed vector space, while large filter sizes would be larger n-grams. However, in a deep CNN, even small filter sizes can result in learnings of language dependencies across higher order n-grams because adjacent k-grams that did not overlap during the convolution in the current layer will then be convoluted over together in the next layer.

   Our approach in choosing filter sizes would be to have multiple filters of various sizes so that different featurizations can be learned by the CNN. To select filter sizes, apply convolution filters which are as tall as the value of n-grams which you think will be helpful for the classification task. Next, the model applies max-pooling (and a non-linearity) to the output of each filter, followed by a fully-connected layer and a soft-max function. Therefore, if a single convolution filter is more or less important than others, then the column of the fully-connected layer's weight matrix which uses the output of max-pooling on that filter will be given smaller weights by a successful optimization algorithm.

5.2 RNN

1. Let $w_1, ..., w_T$ be an input sentence, and let $x_1, ... x_T$ be the corresponding word embeddings. Let f represent a recursive neural net which computes the hidden state h_t of the network at time t. At time t in a simple RNN, we use the hidden state of the RNN at time $t - 1$ - h_{t-1} and the word embedding for the current input word - x_t - to generate the hidden state vector at time t:

$$h_t = f(x_t, h_{t-1})$$

   (Usually we set $h_0 = \vec{0}$.) Now, we can recursively apply f to the sequence $x_1, ... x_T$ to get h_T, the hidden state at the final time step, which will summarize the input of the entire sentence.

2. Each word in the sentence will correspond to a word embedding vector $x_t \in \mathbb{R}^{300}$. We will initialize $h_0 = \vec{0}$ with length 50, since we want the sentence to be summarized by a vector of length 50. Additionally, we will have two weight matrices: $\mathbb{U} \in \mathbf{R}^{50 \times 300}$ and $\mathbf{W} \in \mathbb{R}^{50 \times 50}$. Finally, we will have a bias vector: $b \in \mathbb{R}^{50}$.

   Then, we can define our simple RNN as:

$$h_t = f(x_t, h_{t-1}) = \tanh(Ux_t + Wh_{t-1} + b)$$

   The matrix product Ux_t will transform the input word vector of length 300 into a vector of length 50, while the matrix product Wh_{t-1} takes an input hidden state vector of length 50 and outputs a vector that still has a length of 50. Thus, the vector addition $Ux_t + Wh_{t-1} + b$ adds three vectors each of length 50. Finally, the tanh is applied element-wise, so our output hidden state vector h_t is a vector of length 50.

   Given a sentence of length 10, after the last input word has been fed into the RNN, we will obtain $h_{10} \in \mathbb{R}^{50}$, which summarizes our sentence. Then, we can feed this vector into a fully-connected layer with 50 input units and 4 output units. Let $V \in \mathbb{R}^{4 \times 50}$. Then the fully-connected layer computes:

$$s = Vh_{10}$$

   Now s, the output of the fully-connected layer, is in $\mathbb{R}^4$, containing a score for each of the four classes. Finally, we can feed this vector into a soft-max function which will give us 4 probabilities, one for each class.

5.3 Extra credit experiments of fastText

5.4 Extra credit question

Here we propose a new neural architecture for sentence classification. Let f represent a recursive neural net which computes the hidden state h_t of the network at time t. Let w_t be the t-th word in the input

sentence $(w_1, ...w_T)$ and $\tilde{x}_t$ be its corresponding one-hot vector. The we define x_t as:

$$x_t = E\tilde{x}_t$$

where $E \in \mathbb{R}^{d \times V}$ is the embedding matrix. V is the vocabulary size and length of $\tilde{x}_t$, and $x_t \in \mathbb{R}^d$ is the word embedding corresponding to the input token at w_t.

Then we can set $h_0 = \vec{0}$ and compute the hidden state at time t as:

$$h_t = f(x_t, h_{t-1})$$

We can recursively compute h_T which will contain a summary of the entire input sentence.

Using this RNN we can compute the hidden states $h_1, ..., h_T$ for each time-step in the input sentence. Next, we line up the vectors so that we have a matrix which has height T and width d.

Now we can pass this matrix as input to a convolutional neural network. Specifically, apply several convolution filters of width d to the input matrix. Apply max-pooling over the vector resulting from each convolution. Concatenate the results - the current vector will be as long as the number of convolution filters. Then apply an element-wise non-linearity and pass through a soft-max layer.

This network is a hybrid RNN-CNN network. Traditional RNNs often struggle to appropriately weigh and "remember" long-term information over many time-steps. Though alternative recurrent modules (like GRUs and LSTMs) and architectures (like attention and memory networks) have been proposed to alleviate this problem, in practice these can have limited success. This hybrid network could potentially alleviate this problem since we incorporate the hidden vectors from each time step in the matrix that is passed as input to the CNN.

Additionally, the RNN is also helpful since it allows us to incorporate location-invariant information about word and phrase meaning in the input. Since RNNs are computed recursively, information that is learned in the beginning of an input sequence can also be applied at the end of the sequence. The CNN does not do this as naturally, since it must first find the right convolution and pooling filters to diffuse information learned across different regions of the input space. Therefore, by using $h_1, ...h_T$ for the input matrix to the CNN we can incorporate location-invariant information from the input sentence more easily.

Finally, a major downside of CNNs in language applications is that they can only accept fixed-size inputs. Indeed, even in this model, we must fix the size of the input matrix to the CNN and then either clip or pad the sequences of hidden state vectors as necessary. Clipping the sequence of hidden state vectors is especially damaging since it forces us to throw out valuable information from the input sentence. However, since RNNs recursively compute the hidden state at time t, h_t contains information from w_t as well as from $w_1, ...w_{t-1}$. Therefore, even if we are forced to throw out some h_t vectors, by keeping the "older" h_t vectors, we can ensure that some of the information from the "earlier" vectors is still retained by the network.

6   Language Modeling

We used RNNs to develop language models for the Penn Treebank (PTB) and Gutenberg datasets. For each dataset, we discuss the tuning process for model hyper-parameters, and also provide quantitative and qualitative analysis of the results. For both datasets, we relied on a greedy procedure for hyper-parameter selection: once a certain hyper-parameter was shown to produce a better model, that hyper-parameter was generally included in all subsequent models.

7 Penn Treebank Language Model

Initially, we trained a baseline model using the default hyper-parameters of the provided starter code. We detail these hyper-parameters below, since all subsequent models are loosely based off of these.

RNN Cell	LSTM
Optimizer	SGD w/ gradient clipping
Epochs	6
Layers	1
Embedding Dim.	50
Num. Hidden Units	50
Parameter Init.	Random Uniform from [0,.1]
Vocabulary size	10k
Shuffle after epoch	No

Table 1: PTB Baseline Hyper-parameters

This model achieved a validation perplexity of 145.28.

Next, we explored a range of hyper-parameters to improve the model perplexity. Note that we did not modify the vocabulary size in these experiments since the provided data had already processed the data to a fixed vocabulary size of 150,000, with all out-of-vocabulary tokens replaced with an "unk" token.

[Explain hyper-parameter tuning]

[Analysis]