IFT 6135 – Representation Learning

Assignment 2 – Programming Part

Recurrent Neural Networks, Optimization and Attention

|  |  |
| --- | --- |
| **Students:** | Stefan Wapnick (id 20143021)  Mohamed Amine Arfaoui (id 20150893)  Oussema Keskes (id 20145195)  Stephan Anh Vu Tran (id 20145195) |
| **Due Date:** | March 25, 2019 |
| **Github Link:** |  |

# I - Experimental Setup

The following environment was used for all experiments in this assignment:

* Google Cloud Deep Learning Virtual Machine
* P100 GPU, 4vCPUs
* Python 3.7.1
* PyTorch 1.0.1.post2
* CUDA 10.0.13

# - Implementing a Simple RNN

## Methodology

In this section a simple Recurrent Neural Network (RNN) was implemented using PyTorch to be tested against the Penn Treebank Dataset. The principal equations of an RNN as listed below:

|  |  |
| --- | --- |
|  | Eq. 1 |
|  | Eq. 2 |

Where is the hidden cell state at time of the network, is the ’th input token (typically a word embedding) and the predicted next token in the sequence given the context .

A typical representation of an RNN illustrative its recursive nature is shown in Figure 1. For analysis purposes, the network is typically unrolled through time.

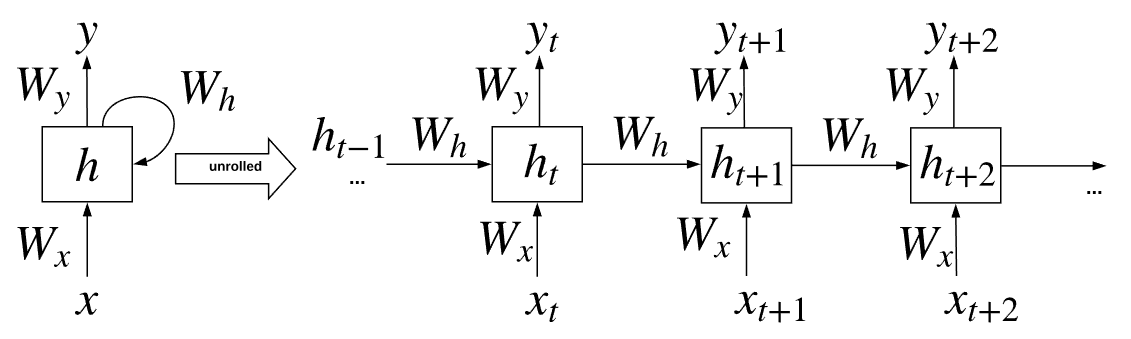


Figure 1 – Illustration of the connections in a Recurrent Neural Network

## Source Code

Listing 1 contains the implementation of the simple RNN done using PyTorch.

The following design decisions were made:

* The conventional **Eq. 2** for computing the hidden state was transformed to use a single weight matrix for efficiency:
* A component class was made for each RNN cell or layer. The RNN class itself is then composed of many RNNLayer classes.
* A RNNBase class was made to be re-used in the GRU section since these two architectures are essentially the same except for the cell type.
* Except for the output and embedding layers, weights were initialized using a form like Xavier initialization (although in this case the fan in size is always taken to be the hidden\_size): where . The output and embedding layers were initialized uniformly in the range .

Listing 1 – RNN Implementation

**class** RNNLayer(nn.Module):  
 *"""  
 Defines a single RNN cell that is composed inside  
 the RNN class  
 """* **def** \_\_init\_\_(self, x\_size, hidden\_size):  
 *"""* **:param** *x\_size: x input size* **:param** *hidden\_size: Hidden state input and output size  
 """* super(RNNLayer, self).\_\_init\_\_()  
 *# For efficiency weight vectors concatenated* self.W = nn.Linear(x\_size + hidden\_size, hidden\_size)  
 self.tanh = nn.Tanh()  
 self.hidden\_size = hidden\_size

**def** forward(self, x, h):  
 *"""* **:param** *x: x input* **:param** *h: Previous h hidden state h\_{t-1}* **:return***: Hidden state output of cell  
 """* **return** self.tanh(self.W(torch.cat((x, h), 1)))  
  
 **def** init\_weights(self):  
 *"""  
 Initializes all weights to [-k, k] where  
 k = 1/sqrt(hidden\_size)  
 """* k = 1. / math.sqrt(self.hidden\_size)  
 torch.nn.init.uniform\_(self.W.weight, -k, k)  
 torch.nn.init.uniform\_(self.W.bias, -k, k)  
  
  
**class** RNNBase(nn.Module):  
  
 **def** \_\_init\_\_(self, layer\_ctor, emb\_size, hidden\_size, seq\_len, batch\_size,  
 vocab\_size, num\_layers, dp\_keep\_prob, track\_state\_history=**False**):  
 *"""* **:param** *layer\_ctor: Number of units in the input embeddings* **:param** *emb\_size: Number of hidden units per layer* **:param** *hidden\_size: Length of the input sequences* **:param** *seq\_len: Length of the input sequences* **:param** *batch\_size: Batch size of data* **:param** *vocab\_size: Number of tokens in the vocabulary* **:param** *num\_layers: Number of hidden layers in network* **:param** *dp\_keep\_prob:The probability of \*not\* dropping out units* **:param** *track\_state\_history: If to track all state history (for 5.2)  
 """* super(RNNBase, self).\_\_init\_\_()  
  
 self.emb\_size = emb\_size  
 self.hidden\_size = hidden\_size  
 self.seq\_len = seq\_len  
 self.batch\_size = batch\_size  
 self.vocab\_size = vocab\_size  
 self.num\_layers = num\_layers  
 self.dp\_keep\_prob = dp\_keep\_prob  
  
 self.rnn\_layers = nn.ModuleList()  
 self.dropout\_layers = nn.ModuleList()  
  
 self.rnn\_layers.extend([layer\_ctor(emb\_size **if** i == 0 **else** hidden\_size, hidden\_size)  
 **for** i **in** range(num\_layers)])  
 self.dropout\_layers.extend([nn.Dropout(1-dp\_keep\_prob)  
 **for** i **in** range(num\_layers)])  
 self.output\_layer = nn.Linear(hidden\_size, vocab\_size)  
  
 self.embedding\_layer = nn.Embedding(vocab\_size, emb\_size)  
 self.embedding\_dropout = nn.Dropout(1-dp\_keep\_prob)  
 self.track\_state\_history = track\_state\_history  
 self.state\_history = **None** self.init\_weights()  
  
 **def** init\_weights(self):  
 *"""  
 Initializes embedding and output weights initialized to [-0.1, 0.1].  
 Output bias initialized to 0s  
 Recurrent layer initialized to [-k, k] where k = 1/sqrt(hidden\_size)  
 """* torch.nn.init.uniform\_(self.embedding\_layer.weight, -0.1, 0.1)  
 torch.nn.init.uniform\_(self.output\_layer.weight, -0.1, 0.1)  
 torch.nn.init.zeros\_(self.output\_layer.bias)  
 **for** rnn\_layer **in** self.rnn\_layers:  
 rnn\_layer.init\_weights()  
  
 **def** init\_hidden(self):  
 *"""  
 Creates the initial hidden state  
 """* **return** torch.zeros([self.num\_layers, self.batch\_size, self.hidden\_size])  
  
 **def** forward(self, inputs, hidden):  
 *"""* **:param** *inputs: A mini-batch of input sequences,  
 composed of int ids representing vocabulary* **:param** *hidden: Initial hidden states for every layer of the stacked RNN.  
 shape: (num\_layers, batch\_size, hidden\_size)* **:return***: Tuple of output logits and final hidden state.  
 Shape (seq\_len, batch\_size, vocab\_size)  
 and (num\_layers, batch\_size, hidden\_size) respectively  
 """* logits = torch.zeros([self.seq\_len, self.batch\_size, self.vocab\_size],  
 device=inputs.device)  
  
 *# Used for 5.2 to track all hidden states for gradients* **if** self.track\_state\_history:  
 self.state\_history = [[] **for** \_ **in** range(self.num\_layers)]  
  
 embedding\_output = self.embedding\_layer(inputs)  
  
 *# For each time-step compute t'th output by looping upwards in layers.  
 # Hidden state is stored for next t+1 chain.  
 # Embedding layer and recurrent cells are followed by dropout* **for** t **in** range(self.seq\_len):  
 x = self.embedding\_dropout(embedding\_output[t])  
 h\_t = []  
 **for** l **in** range(self.num\_layers):  
 h\_out = self.rnn\_layers[l](x, hidden[l])  
 x = self.dropout\_layers[l](h\_out)  
 h\_t.append(h\_out)  
  
 *# Used for 5.2 to track all hidden states for gradients* **if** self.track\_state\_history:  
 self.state\_history[l].append(h\_out)  
  
 *# Form new hidden state tensor for next time-step* hidden = torch.stack(h\_t)  
 logits[t] = self.output\_layer(x)  
  
 **return** logits, hidden  
  
 **def** generate(self, input, hidden, generated\_seq\_len):  
 *"""* **:param** *input: A mini-batch of input tokens  
 shape: (batch\_size)* **:param** *hidden: The initial hidden states for every layer of the stacked RNN  
 shape: (num\_layers, batch\_size, hidden\_size)* **:param** *generated\_seq\_len:  
 The length of the sequence to generate  
 shape: (num\_layers, batch\_size, hidden\_size)* **:return***: Sampled sequences of tokens  
 shape: (generated\_seq\_len, batch\_size)  
 """* hidden\_states = hidden.clone()  
 current\_word = input  
 samples = torch.zeros((generated\_seq\_len, input.shape[0]), device=input.device)  
  
 **for** t **in** range(generated\_seq\_len):  
 x = self.embedding\_dropout(self.embedding\_layer(current\_word))  
 **for** l **in** range(self.num\_layers):  
 hidden\_states[l] = self.rnn\_layers[l](x, hidden\_states[l])  
 x = self.dropout\_layers[l](hidden\_states[l])  
  
 *# Predicted word fed back through network as next current\_word* current\_word = torch.distributions.Categorical(  
 logits=self.output\_layer(x)).sample()  
 samples[t] = current\_word  
  
 **return** samples  
  
  
**class** RNN(RNNBase):  
 *"""  
 Implements an RNN recurrent network. Composes RNNLayer cells.  
 """* **def** \_\_init\_\_(self, emb\_size, hidden\_size, seq\_len,  
 batch\_size, vocab\_size, num\_layers, dp\_keep\_prob):  
 *"""* **:param** *emb\_size: The number of units in the input embeddings* **:param** *hidden\_size: The number of hidden units per layer* **:param** *seq\_len: The length of the input sequences* **:param** *batch\_size: Batch size of data* **:param** *vocab\_size: The number of tokens in the vocabulary* **:param** *num\_layers: The depth of the stack (number of hidden layers)* **:param** *dp\_keep\_prob: The probability of \*not\* dropping out units in the  
 non-recurrent connections.  
 """* super(RNN, self).\_\_init\_\_(RNNLayer, emb\_size, hidden\_size, seq\_len,  
 batch\_size, vocab\_size, num\_layers, dp\_keep\_prob)

# Implement RNN with Gated Recurrent Units (GRU)

## Methodology

In this section the basic RNN of section 1 is augmented with gated recurrent units (GRU). The GRU adds in trainable weights for the reset and forget operations that allow the GRU to learn more long-term dependencies and alleviate the vanishing gradient problem. Although such an improvement comes with additional complexity and computational cost.

The principal equations of a GRU are:

|  |  |
| --- | --- |
|  | Eq. 3 |
|  | Eq. 4 |
|  | Eq. 5 |
|  | Eq. 6 |

is known as the reset gate and is the forget gate.

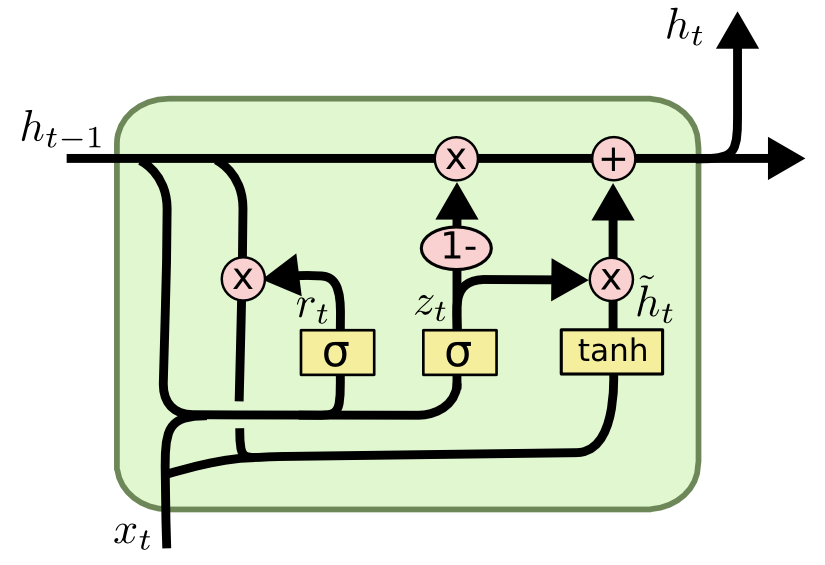


Figure 2 – Illustration of GRU cell containing reset and forget gates [1]

<http://sunlab.org/teaching/cse6250/fall2018/dl/dl-rnn.html#recurrent-neural-networks-2>

## Source Code

Listing 2 contains the implementation of the GRU done using Pytorch.

* Inputs and weight matrices for and are once again concatenated for efficiency as in section 1.
* A GRULayer component class is implemented that are composed inside the GRU call.
* The same RNNBase class from section 1 is re-used for the GRU class here since only the cell / layer type need be changed from section 1. All other logic for the recursive connections remains the same.

Listing 2 – Implementation of GRU

**class** GRULayer(nn.Module):  
 *"""  
 Implements a GRU cell composed in the GRU class  
 """* **def** \_\_init\_\_(self, x\_size, hidden\_size):  
 *"""* **:param** *x\_size: x input size* **:param** *hidden\_size: Hidden state input and output size  
 """* super(GRULayer, self).\_\_init\_\_()  
 *# For efficiency weight vectors concatenated* self.r\_linear = nn.Linear(x\_size + hidden\_size, hidden\_size)  
 self.z\_linear = nn.Linear(x\_size + hidden\_size, hidden\_size)  
 self.h\_linear = nn.Linear(x\_size + hidden\_size, hidden\_size)  
 self.h\_tanh = nn.Tanh()  
 self.r\_sigmoid = nn.Sigmoid()  
 self.z\_sigmoid = nn.Sigmoid()  
 self.hidden\_size = hidden\_size  
  
 **def** forward(self, x, h\_prev):  
 *"""* **:param** *x: x input* **:param** *h\_prev: Previous h hidden state h\_{t-1}* **:return***: Hidden state output of cell  
 """* combined\_input = torch.cat((x, h\_prev), 1)  
 z = self.z\_sigmoid(self.z\_linear(combined\_input))  
 r = self.r\_sigmoid(self.r\_linear(combined\_input))  
 h\_candidate = self.h\_tanh(self.h\_linear(torch.cat((x, r\*h\_prev), 1)))  
 **return** (1-z)\*h\_prev + z\*h\_candidate  
  
 **def** init\_weights(self):  
 *"""  
 Initializes all weights to [-k, k] where  
 k = 1/sqrt(hidden\_size)  
 """* k = 1. / math.sqrt(self.hidden\_size)  
 torch.nn.init.uniform\_(self.r\_linear.weight, -k, k)  
 torch.nn.init.uniform\_(self.r\_linear.bias, -k, k)  
 torch.nn.init.uniform\_(self.z\_linear.weight, -k, k)  
 torch.nn.init.uniform\_(self.z\_linear.bias, -k, k)  
 torch.nn.init.uniform\_(self.h\_linear.weight, -k, k)  
 torch.nn.init.uniform\_(self.h\_linear.bias, -k, k)  
  
  
**class** GRU(RNNBase):  
 *"""  
 Implements a GRU recurrent network. Composes GRULayer cells.  
 """* **def** \_\_init\_\_(self, emb\_size, hidden\_size, seq\_len, batch\_size,  
 vocab\_size, num\_layers, dp\_keep\_prob):  
 super(GRU, self).\_\_init\_\_(GRULayer, emb\_size, hidden\_size, seq\_len,  
 batch\_size, vocab\_size, num\_layers, dp\_keep\_prob)

# Attention Module of Transformer Network

## Methodology

The transformer is a newer architecture that uses the concept of attention (weighting of inputs based on perceived importance) for sequence modeling. Only a section of the transformer is implemented in this section, specifically the multi-head scaled dot-product attention defined below:

|  |  |
| --- | --- |
|  | Eq. 7 |
|  | Eq. 8 |
|  | Eq. 9 |

An important part of the multi-head attention module is the application of attention to specific elements in the workflow specified by a binary mask (where a value of 1 indicates that the element should have attention applied to it). Before applying the SoftMax function to yield **,** thismask is applied, the intermediate value is adjusted by the mask :

|  |  |
| --- | --- |
|  | Eq. 10 |

## Source Code

To implement the attention calculation for a separate SingleHeadAttention class was made. The MultiHeadedAttention composes these individual attention head classes and computes the final output on concatenated values. Besides these details, Eq. 7-Eq. 10 were followed.

Listing 3 – Implementation of Multi-Head Attention module

**class** SingleHeadAttention(nn.Module):  
 *"""  
 Implements a single attention head class composed in  
 MultiHeadedAttention. Each head computes an a\_i / h\_i result  
 """* EPSILON = 1e9  
  
 **def** \_\_init\_\_(self, n\_units, d\_k, dropout\_rate):  
 *"""  
 n\_units: Number of units in the attention head  
 d\_k: Key output size  
 dropout\_rate: Rate to drop units  
 """* super(SingleHeadAttention, self).\_\_init\_\_()  
 self.n\_units = n\_units  
 self.d\_k = d\_k  
 self.q\_linear = nn.Linear(self.n\_units, self.d\_k)  
 self.k\_linear = nn.Linear(self.n\_units, self.d\_k)  
 self.v\_linear = nn.Linear(self.n\_units, self.d\_k)  
 self.dropout = nn.Dropout(dropout\_rate)  
  
 **def** init\_weights(self):  
 *"""  
 Initializes all weights to [-k, k] where  
 k = 1/sqrt(n\_units)  
 """* k = 1. / math.sqrt(self.n\_units)  
 nn.init.uniform\_(self.q\_linear.weight, -k, k)  
 nn.init.uniform\_(self.q\_linear.bias, -k, k)  
 nn.init.uniform\_(self.k\_linear.weight, -k, k)  
 nn.init.uniform\_(self.k\_linear.bias, -k, k)  
 nn.init.uniform\_(self.v\_linear.weight, -k, k)  
 nn.init.uniform\_(self.v\_linear.bias, -k, k)  
  
 **def** forward(self, query, key, value, mask=**None**):  
 *"""  
 Computes a single attention a\_i / h\_i result* **:param** *query: Query matrix Q (batch\_size, seq\_len, n\_units)* **:param** *key: Key matrix K (batch\_size, seq\_len, n\_units)* **:param** *value: Value matrix V (batch\_size, seq\_len, n\_units)* **:param** *mask: Mask specifying whether to attend each element  
 (batch\_size, seq\_len, seq\_len)  
 """  
 # Computes intermediate x value before compute a\_i* q\_out = self.q\_linear(query)  
 k\_out = self.k\_linear(key)  
 v\_out = self.v\_linear(value)  
 x = torch.matmul(q\_out, k\_out.transpose(1, 2))  
 x = torch.div(x, math.sqrt(self.d\_k))  
  
 *# Apply mask* **if** mask **is not None**:  
 x = x \* mask - SingleHeadAttention.EPSILON \* (1 - mask)  
  
 *# Output attention head value* a = F.softmax(x, dim=-1)  
 a = self.dropout(a)  
 **return** torch.matmul(a, v\_out)  
  
  
**class** MultiHeadedAttention(nn.Module):  
 *"""  
 Implements the multi-head scaled dot-product attention  
 component of a transformer. Composes SingleHeadAttention.  
 """* **def** \_\_init\_\_(self, n\_heads, n\_units, dropout=0.1):  
 *"""* **:param** *n\_heads: the number of attention heads* **:param** *n\_units: the number of output units* **:param** *dropout: probability of dropping units  
 """* super(MultiHeadedAttention, self).\_\_init\_\_()  
 *# Size of the keys, values, and queries (self.d\_k)  
 # is output units divided by the number of heads.* self.d\_k = n\_units // n\_heads  
 **assert** n\_units % n\_heads == 0  
  
 self.n\_heads = n\_heads  
 self.n\_units = n\_units  
  
 self.out\_linear = nn.Linear(n\_units, n\_units)  
 self.attention\_heads = clones(SingleHeadAttention(n\_units, self.d\_k,  
 dropout), n\_heads)  
 self.init\_weights()  
  
 **def** init\_weights(self):  
 *"""  
 Initializes all weights to [-k, k] where k = 1/sqrt(hidden\_size)  
 """* k = 1. / math.sqrt(self.n\_units)  
 nn.init.uniform\_(self.out\_linear.weight, -k, k)  
 nn.init.uniform\_(self.out\_linear.bias, -k, k)  
 **for** attention\_head **in** self.attention\_heads:  
 attention\_head.init\_weights()  
  
 **def** forward(self, query, key, value, mask=**None**):  
 *"""  
 Compute multi-head scaled dot product attention* **:param** *query: Query matrix Q (batch\_size, seq\_len, n\_units)* **:param** *key: Key matrix K (batch\_size, seq\_len, n\_units)* **:param** *value: Value matrix V (batch\_size, seq\_len, n\_units)* **:param** *mask: Mask specifying whether to attend each element  
 (batch\_size, seq\_len, seq\_len)  
 """  
  
 # Mask preemptively converted to float for purposes of  
 # tensor multiplication x \* s - 1e9\*(1-s)* **if** mask **is not None**:  
 mask = mask.float()  
  
 *# Compute each a\_i output (see SingleHeadAttention),  
 # concatenate all together and put through final linear output* h\_out = torch.cat([atn(query, key, value, mask)  
 **for** atn **in** self.attention\_heads], dim=-1)  
 **return** self.out\_linear(h\_out)

# Training Language Models

Using the implemented architectures of sections 1-3, experiments were ran against the Penn Treebank dataset with a variety of hyper-parameters.

In the following sections, the short-hand notations given in Table 1 are used to denote model hyper-parameters.

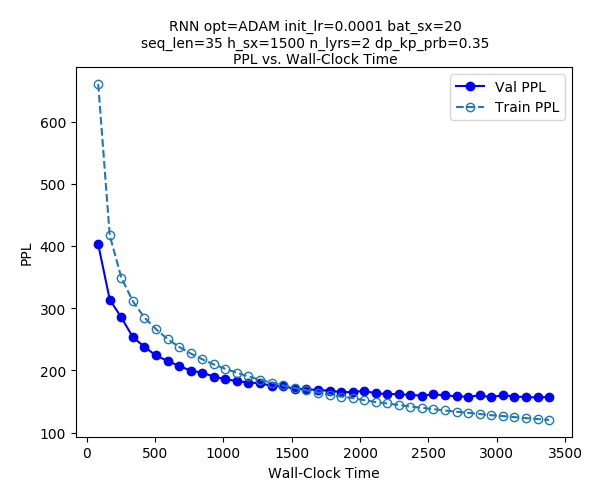
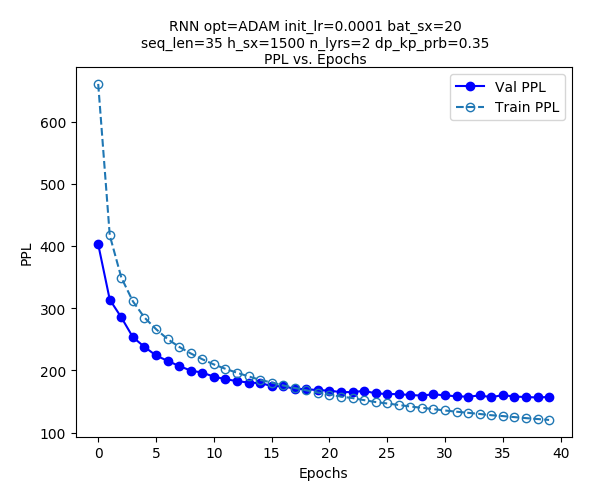
Table 1 – Shorthand notation for hyperparameters in result figures

|  |  |
| --- | --- |
| Abbreviation | Description |
| opt | Optimizer |
| init\_lr | Initial learning rate |
| bat\_sz | Batch size |
| seq\_len | Sequence length |
| h\_sz | Hidden state size |
| n\_lyrs | Number of layers |
| dp\_kp\_prb | Dropout keep probability |

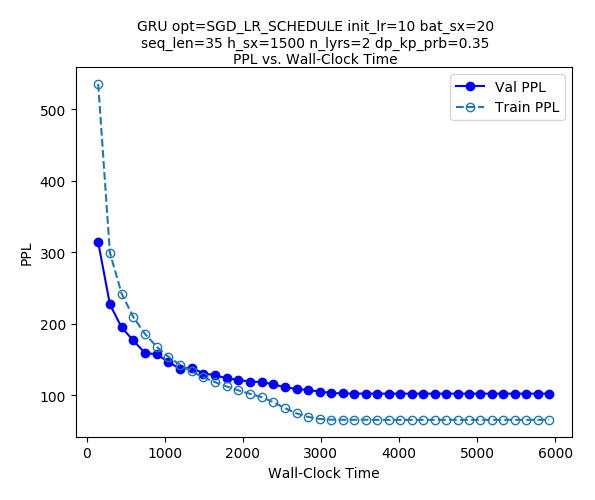
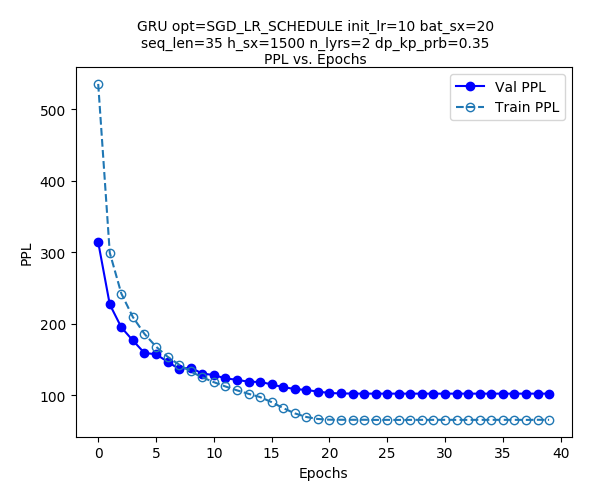
## Model Comparison Results

### Summary

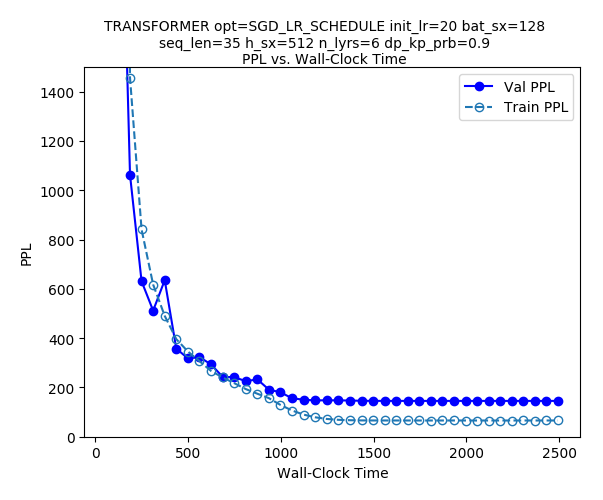
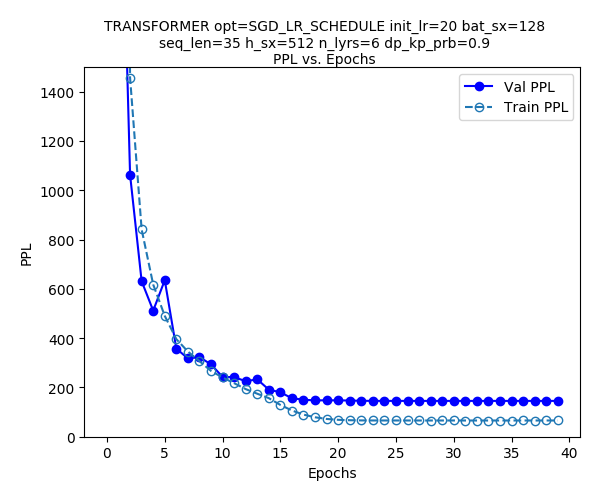
### Learning Curves



|  |  |
| --- | --- |
| Figure 3 | Figure 4 |
|  |  |



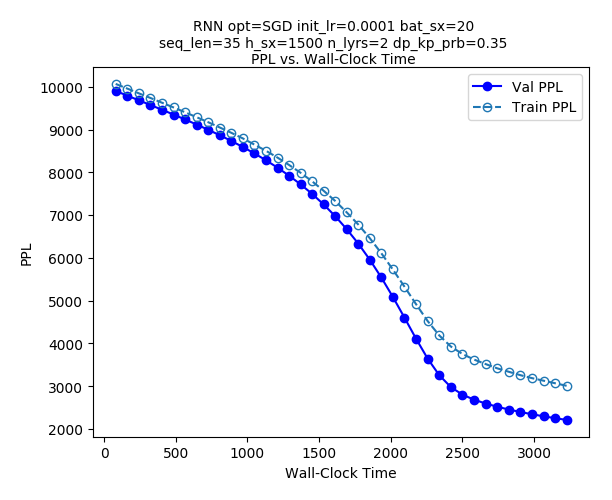
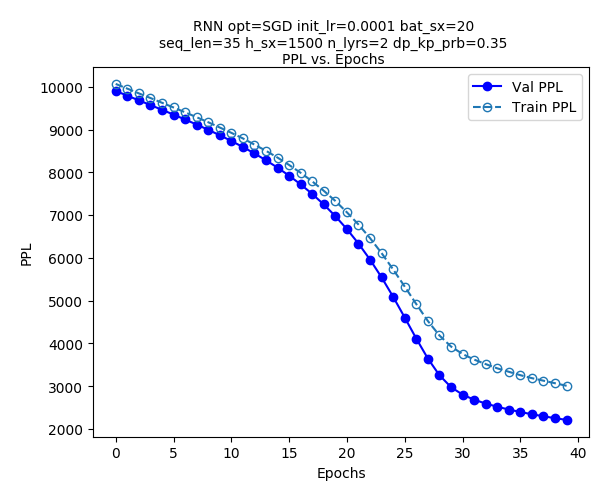
|  |  |
| --- | --- |
| Figure 5 | Figure 6 |

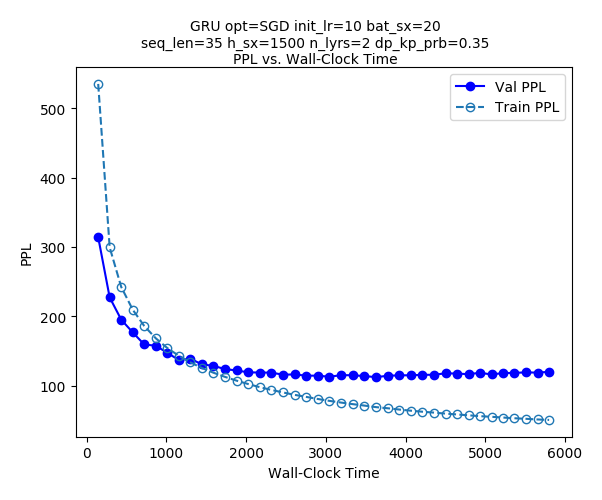
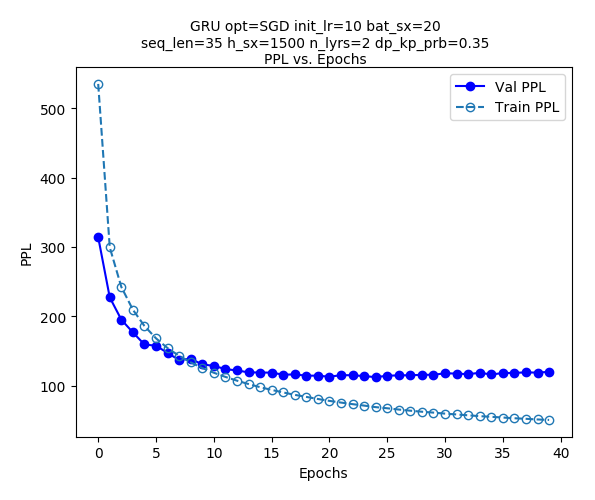


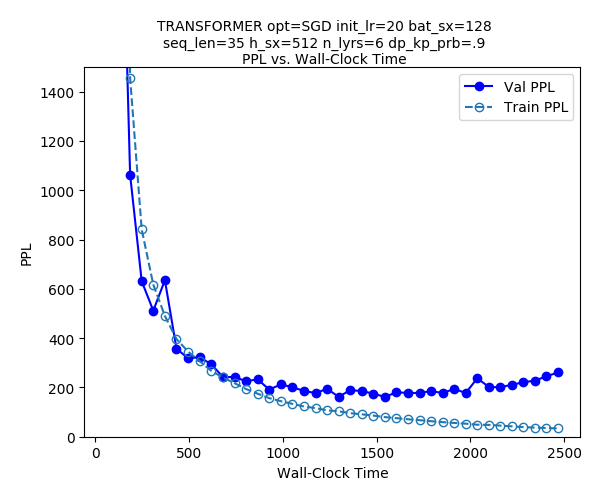
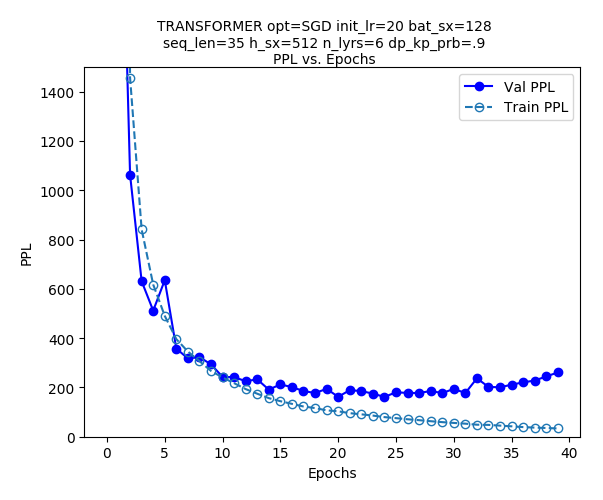
## Exploration of Optimizers Results

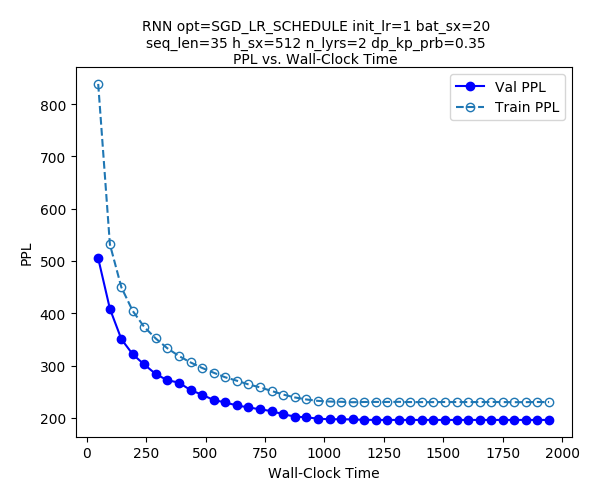
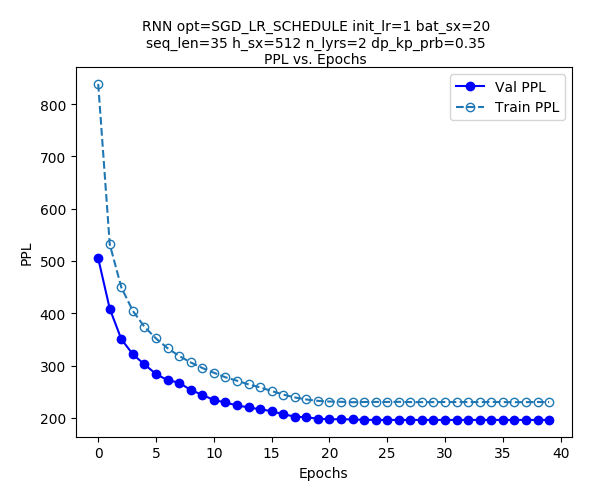
### Summary

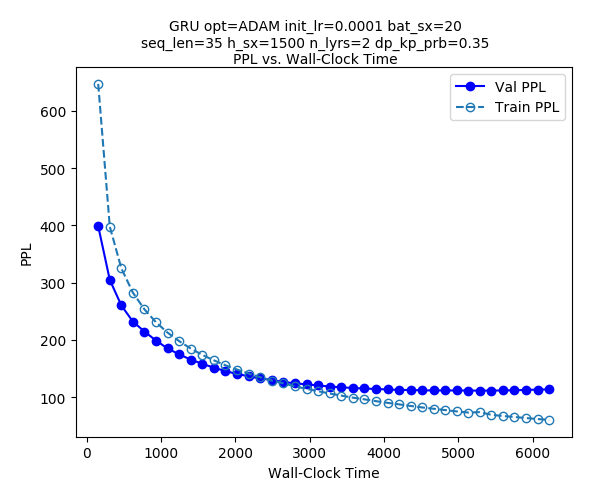
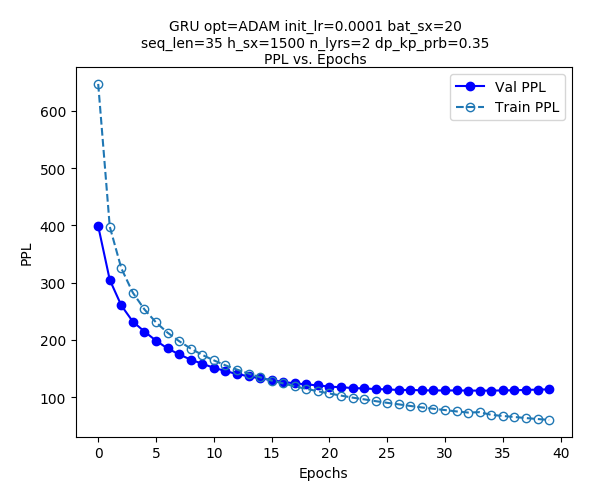
### Learning Curves

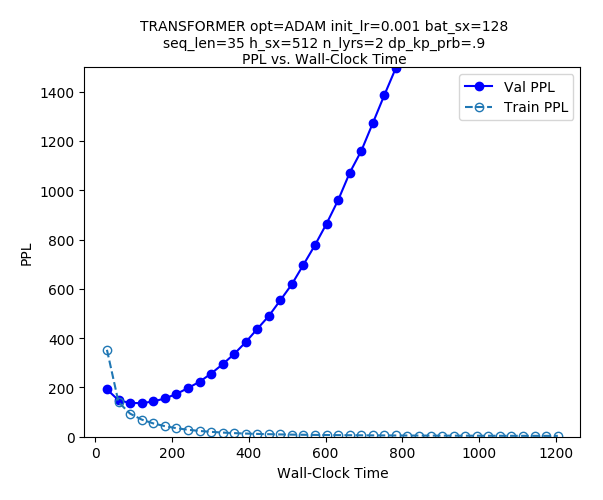
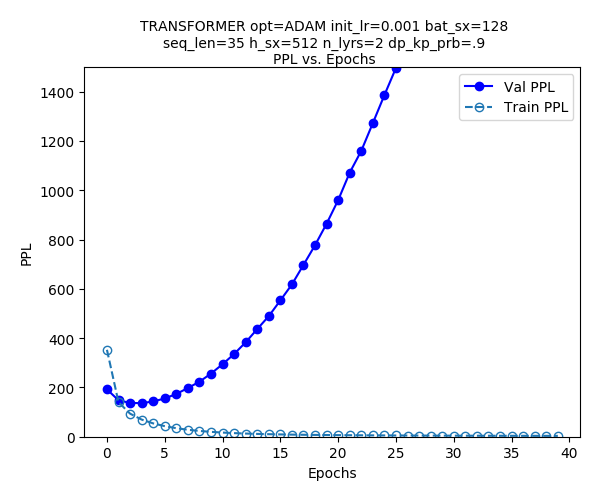








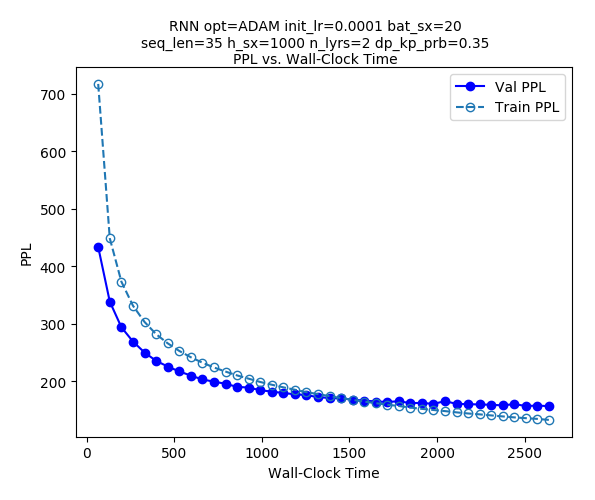
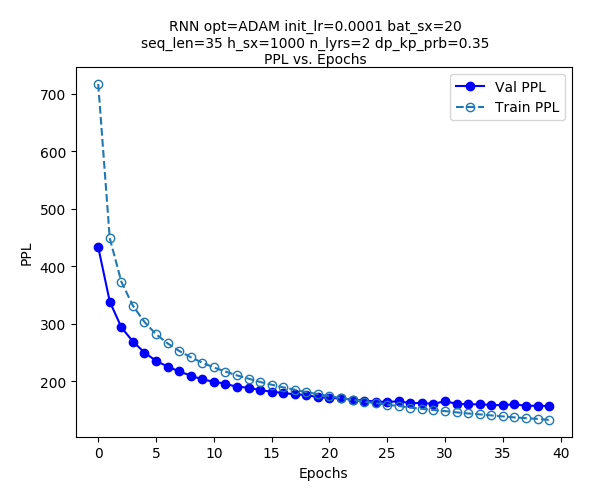


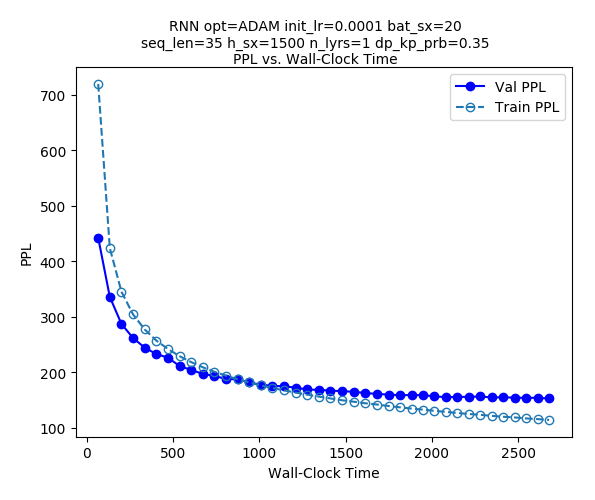
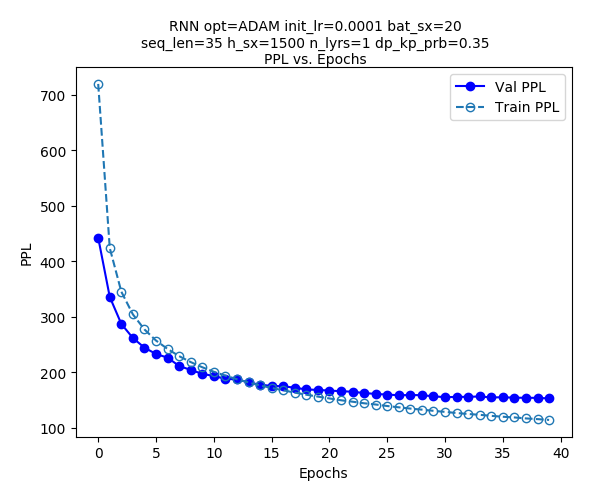


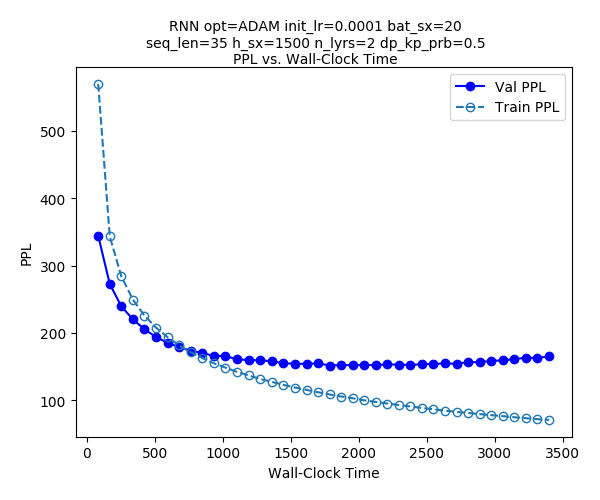
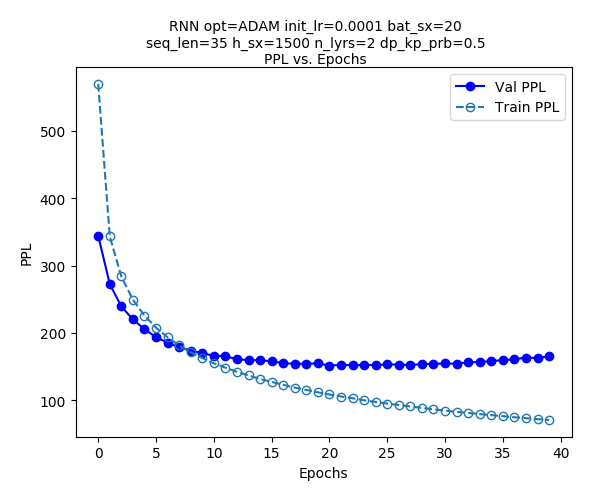
## Hyper-Parameter Search Results

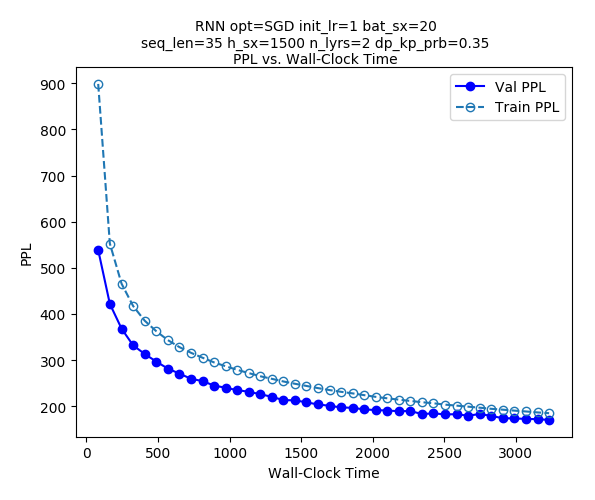
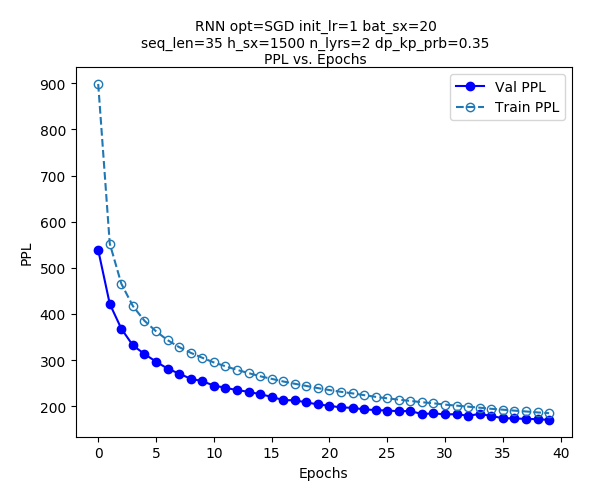
### Summary

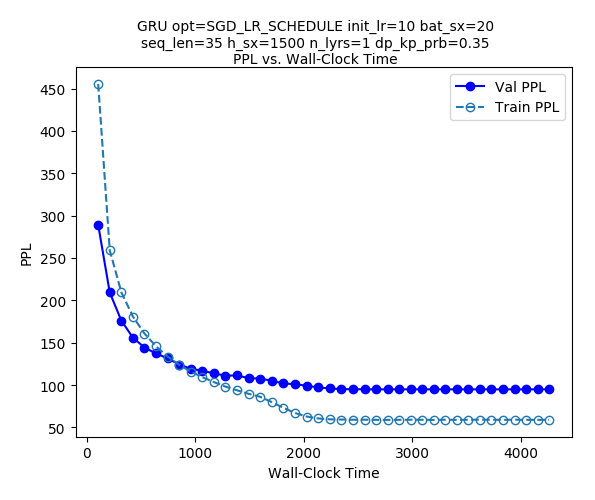
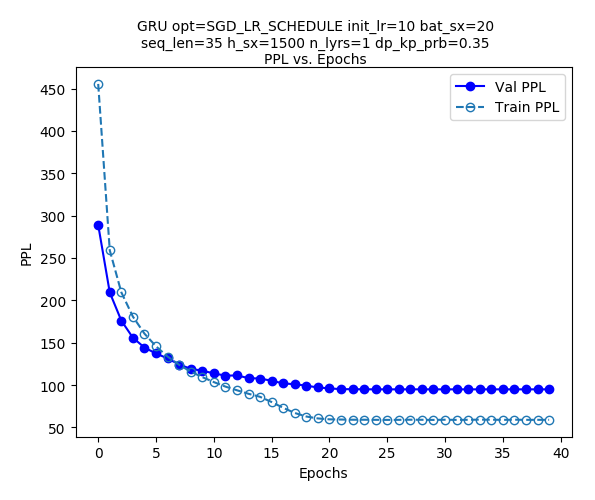
### Learning Curves

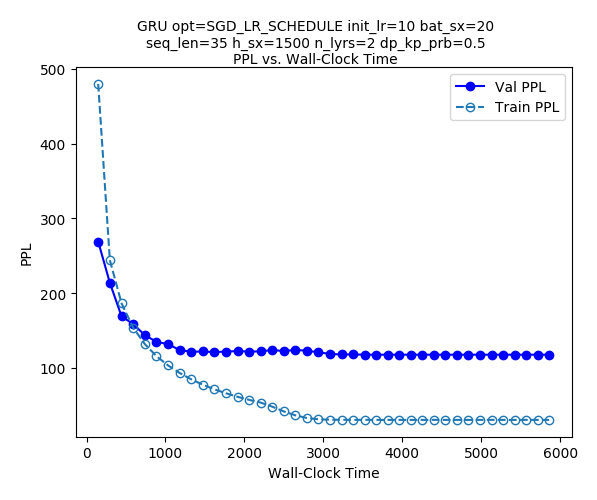
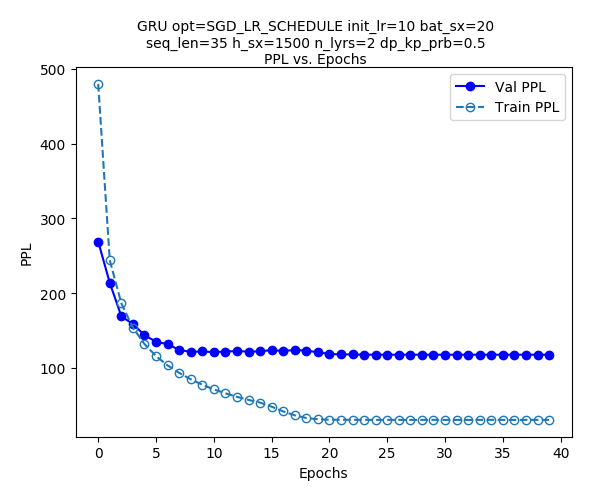


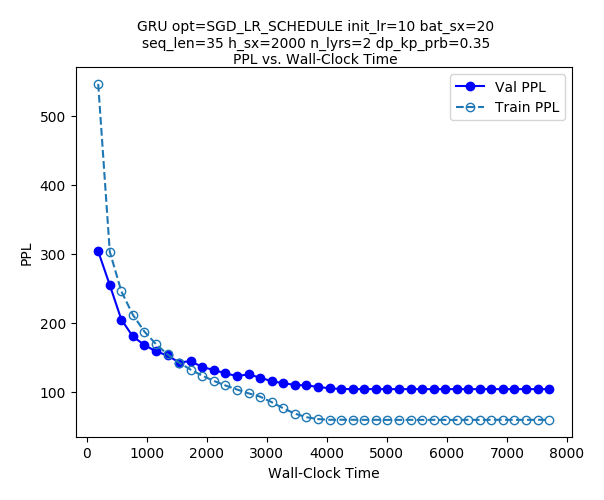
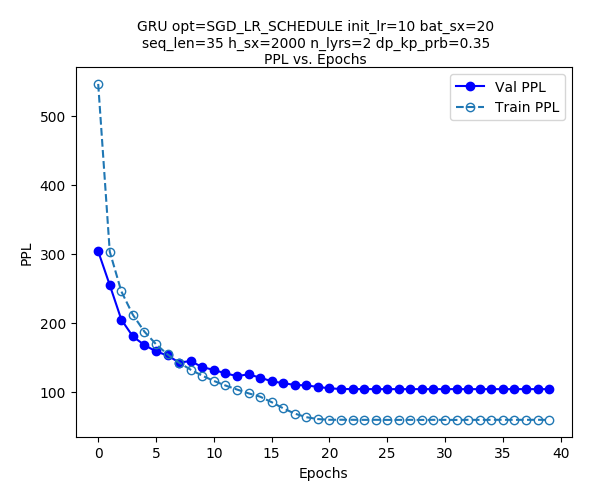


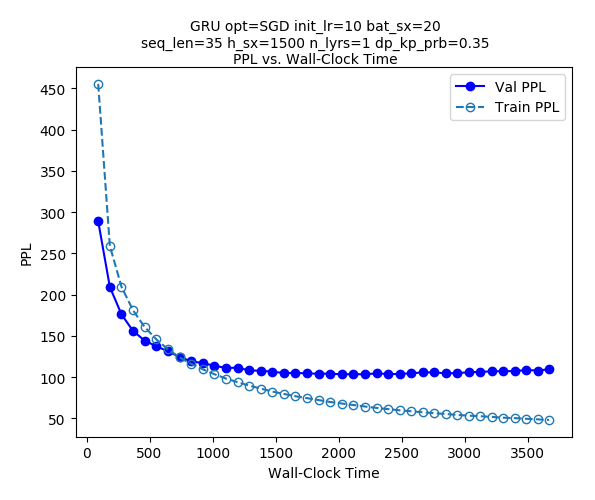
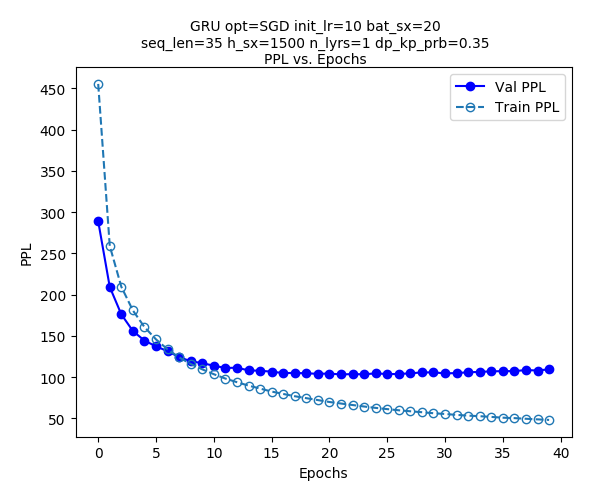


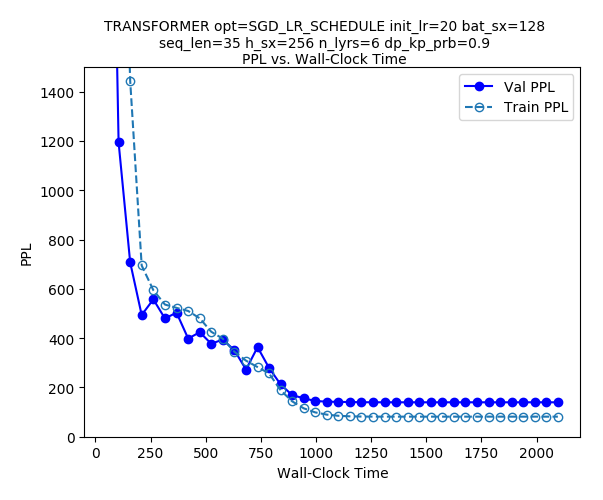
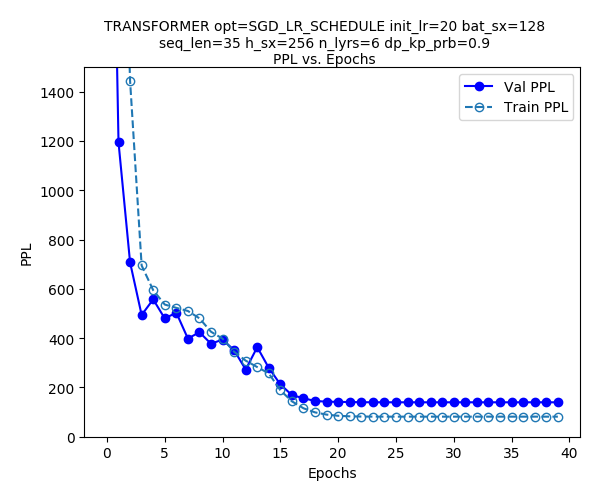


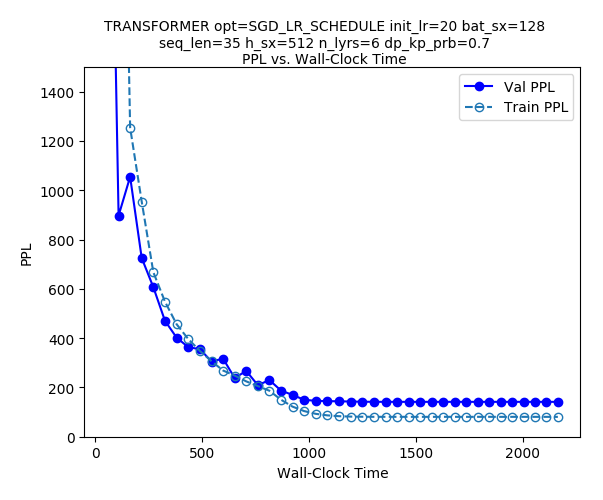
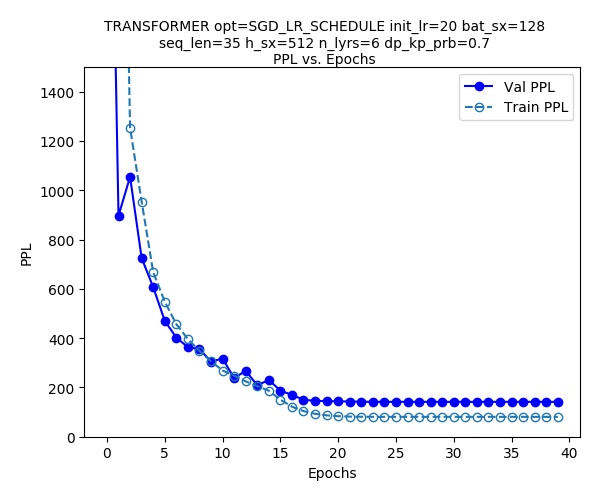


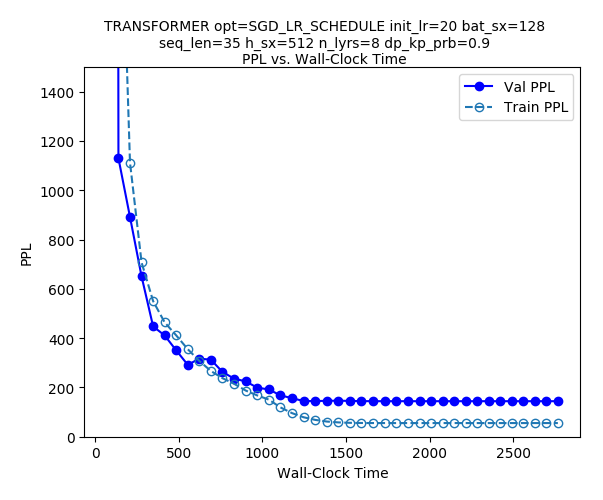
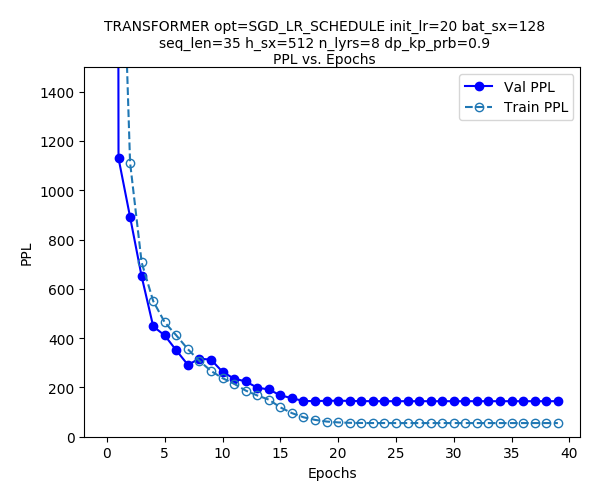


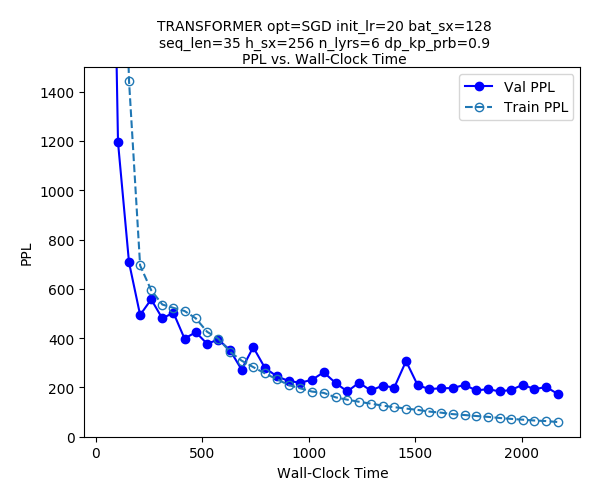
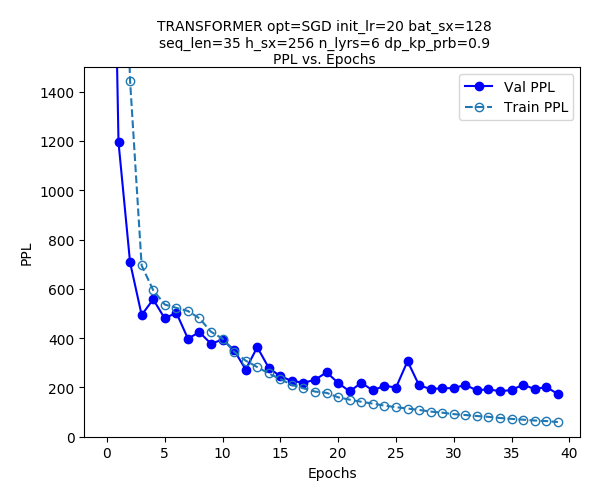












## All Results Summary

### Table Summary

### Organized by Optimizer

### Organized by Architecture

## Experiment Commands

The scripts used to run each experiment can be found in run\_4\_1.sh, run\_4\_2.sh and run\_4\_3.sh:

Table 2 – Experiment commands for problem 4.1 (found in run\_4\_1.sh)

|  |  |
| --- | --- |
| Experiment | Command |
| RNN | python ptb-lm.py --model=RNN --optimizer=ADAM --initial\_lr=0.0001 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=2 --dp\_keep\_prob=0.35 |
| GRU | python ptb-lm.py --model=GRU --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=10 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=2 --dp\_keep\_prob=0.35 |
| Transformer | python ptb-lm.py --model=TRANSFORMER --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=20 --batch\_size=128 --seq\_len=35 --hidden\_size=512 --num\_layers=6 --dp\_keep\_prob=0.9 |

Table 3 – Experiment commands for problem 4.2 (found in run\_4\_2.sh)

|  |  |
| --- | --- |
| Experiment | Command |
| RNN + SGD | python ptb-lm.py --model=RNN --optimizer=SGD --initial\_lr=0.0001 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=2 --dp\_keep\_prob=0.35 |
| GRU + SGD | python ptb-lm.py --model=GRU --optimizer=SGD --initial\_lr=10 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=2 --dp\_keep\_prob=0.35 |
| Transformer + SGD | python ptb-lm.py --model=TRANSFORMER --optimizer=SGD --initial\_lr=20 --batch\_size=128 --seq\_len=35 --hidden\_size=512 --num\_layers=6 |
| RNN + SGD Schedule | python ptb-lm.py --model=RNN --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=1 --batch\_size=20 --seq\_len=35 --hidden\_size=512 --num\_layers=2 --dp\_keep\_prob=0.35 |
| GRU + ADAM | python ptb-lm.py --model=GRU --optimizer=ADAM --initial\_lr=0.0001 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=2 --dp\_keep\_prob=0.35 |
| Transformer + ADAM | python ptb-lm.py --model=TRANSFORMER --optimizer=ADAM --initial\_lr=0.001 --batch\_size=128 --seq\_len=35 --hidden\_size=512 --num\_layers=2 --dp\_keep\_prob=.9 |

In Table 3 containing the scripts used to run problem 4.3, changes made to the base script from problem 4.1 are listed in brackets in the experiment column.

Table 4 – Experiment commands for problem 4.3 (found in run\_4\_3.sh)

|  |  |
| --- | --- |
| Experiment | Command |
| RNN  (-num\_layers) | python ptb-lm.py --model=RNN --optimizer=ADAM --initial\_lr=0.0001 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=1 --dp\_keep\_prob=0.35 |
| RNN  (+dg\_keep\_prob) | python ptb-lm.py --model=RNN --optimizer=ADAM --initial\_lr=0.0001 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=2 --dp\_keep\_prob=0.5 |
| RNN  (-hidden\_size) | python ptb-lm.py --model=RNN --optimizer=ADAM --initial\_lr=0.0001 --batch\_size=20 --seq\_len=35 --hidden\_size=1000 --num\_layers=2 --dp\_keep\_prob=0.35 |
| RNN  (SGD, init\_lr = 1) | python ptb-lm.py --model=RNN --optimizer=SGD --initial\_lr=1 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=2 --dp\_keep\_prob=0.35 |
| GRU  (-num\_layers) | python ptb-lm.py --model=GRU --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=10 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=1 --dp\_keep\_prob=0.35 |
| GRU  (+dp\_keep\_prob) | python ptb-lm.py --model=GRU --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=10 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=2 --dp\_keep\_prob=0.5 |
| GRU  (+hidden\_size) | python ptb-lm.py --model=GRU --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=10 --batch\_size=20 --seq\_len=35 --hidden\_size=2000 --num\_layers=2 --dp\_keep\_prob=0.35 |
| GRU  (SGD, -num\_layers) | python ptb-lm.py --model=GRU --optimizer=SGD --initial\_lr=10 --batch\_size=20 --seq\_len=35 --hidden\_size=1500 --num\_layers=1 --dp\_keep\_prob=0.35 |
| Transformer  (+num\_layers) | python ptb-lm.py --model=TRANSFORMER --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=20 --batch\_size=128 --seq\_len=35 --hidden\_size=512 --num\_layers=8 --dp\_keep\_prob=0.9 |
| Transformer  (-dp\_keep\_prob) | python ptb-lm.py --model=TRANSFORMER --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=20 --batch\_size=128 --seq\_len=35 --hidden\_size=512 --num\_layers=6 --dp\_keep\_prob=0.7 |
| Transformer  (-hidden\_size) | python ptb-lm.py --model=TRANSFORMER --optimizer=SGD\_LR\_SCHEDULE --initial\_lr=20 --batch\_size=128 --seq\_len=35 --hidden\_size=256 --num\_layers=6 --dp\_keep\_prob=0.9 |
| Transformer  (SGD, -hidden\_size) | python ptb-lm.py --model=TRANSFORMER --optimizer=SGD --initial\_lr=20 --batch\_size=128 --seq\_len=35 --hidden\_size=256 --num\_layers=6 --dp\_keep\_prob=0.9 |

## Discussion

**Question 1. What did you expect to see in these experiments, and what actually happens? Why do you think that happens?**

**Question 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.**

**Question 3. Which hyperparameters and optimizer would you use if you were most concerned with wallclock 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?)**

**Question 4. Which architecture is most “reliable” (decent generalization performance for most hyperparameter+optimizer settings), and which is more unstable across settings?**

**Question 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.**

One aspect to investigate would be the effect of the sequence length of the performance of each model (seq\_len parameter). It might initially be thought that perhaps a longer sequence length could help lower perplexity since there is more context for later word in the sequence. However, the input data is such that a single sequence may span across several sentences (expressing different ideas) and so it is possible that the context may become inaccurate or mislead the prediction. In this sense it could be argued that past a certain length a larger context may become less useful. There might be a certain optimal sequence length for the models. In either case, a model such as a GRU with its gated architecture would be better able to model long term dependencies as opposed to an un-augmented RNN and so this could be observed during testing.

# Detailed Evaluation of Trained Models

**Note:** For these experiments the architectures from Problem 4.1 (Model Comparison) are use.

Table 5 – Models used for section 5

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Model | Optimizer | Initial Learning Rate | Batch Size | Sequence Length | Hidden Size | Layers | Dropout keep probability |
| RNN | ADAM | 0.0001 | 20 | 35 | 1500 | 2 | 0.35 |
| GRU | SGD\_LR\_Schedule | 10 | 20 | 35 | 1500 | 2 | 0.35 |
| Transformer | SGD\_LR\_Schedule | 20 | 128 | 35 | 512 | 6 | 0.9 |

## Average Loss per Time-Step

The average loss at each time-step is examined in this exercise. The losses were averaged over all mini-batches in the validation set.

**Results:**

Implementation can be found in 5\_1\_loss\_per\_timestep.py.

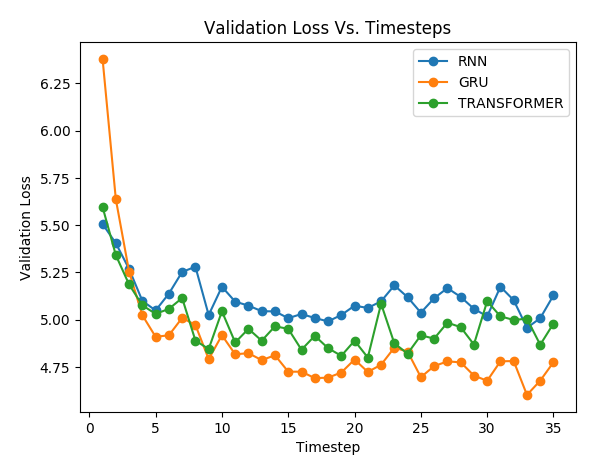


Figure 7 – Validation loss over time-steps for each architecture of Problem 4.1

**Discussion:**

It was noted that the loss generally decreases over timesteps. This makes sense given that as more time passes more context is accumulated so the model hopefully makes better predictions (lower loss).

The GRU model appears to achieve best results in this regard (lowest loss at the end of the sequence length). This makes sense given that the GRU with its additional gates it better able to learn long-term dependencies. This can be contrasted with the RNN which can sometimes struggle with learning longer term dependencies. The transformer appears to be a middle ground between these two architectures.

It was also noted that the GRU appears to the start with the highest loss despite achieving final best performance. This could be coincidental or perhaps due to the larger number of parameters that must be tuned in comparison to the vanilla RNN so there is a larger possibility for error initially.

## Gradient per Time-Step

In this exercise the gradient of each hidden state to the final loss at the last timestep () is examined for a single mini-batch. Gradients in a batch are averaged together. The normal of these gradient vectors is computed and normalized to a range of [0, 1]. In the case of multiple layers, the gradient vectors are concatenated together (such that there is a single gradient vector per timestep.

**Results:**

Implementation can be found in 5\_2\_grad\_per\_timestep.py.

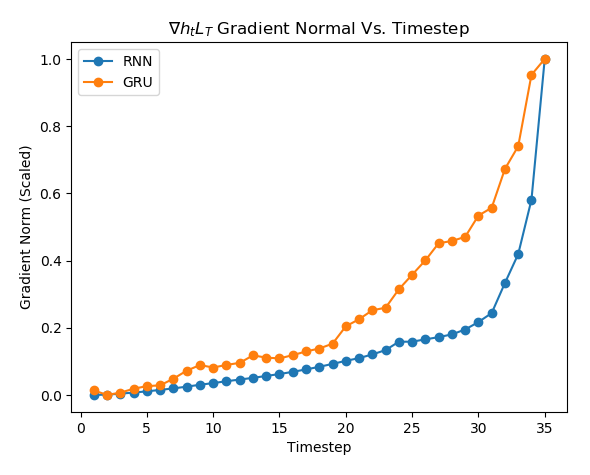


Figure 8 - normal over timesteps

**Discussion:**

Gradients with respect to the final loss (at ) were found to be largest closer to . This makes sense since gradients of each typically decay (given the sequence of matrix multiplications during back-propagation) giving rise to the vanishing gradient problem in certain instances.

Gradients decay less rapidly in the GRU model, given its gated architecture alleviates the vanishing gradient problem and lets it learn long term dependencies.

## Generation of Samples

Using the trained RNN and GRU models, novel sentences were generated. This was done by sampling from the predicted distribution of output symbols and feeding back in this prediction as the next timestep input of the network. Adapting **Eq. 1**, the sampling operation can be written as:

**Results:**

Implementation can be found in 5\_3\_generate\_sentences.py.

Only 9 samples are shown here for analysis. All generated sequences are in the appendix.

**Discussion:**

The generated sentences are in general coherent however there is a noticeable different to human speech. In general, the models seem to be generating short sequences of coherent words, however the topic appears to change abruptly after several tokens.

It should also be noted that since outputs (or inputs for the next timestep) are sampled it is possible to receive an unlucky sample that may change the direction of the sentence.

In general, one would expect the GRU to generate more logical phrases (given its lower perplexity during testing and the nature of its gated architecture for learning longer term dependencies). This seems to be the case in many instances. However, it is also hard to provide an objective analysis of a sentence’s quality since it is somewhat opiniated and no concrete metric is being used here.

# References

|  |  |
| --- | --- |
| [2] | N.A. “Taxi-V2”. OpenAI. Available: <https://gym.openai.com/envs/Taxi-v2/> [Accessed: 2019-03-14] |
| [1] | R. Sutton and A. Barto, *Reinforcement Learning: An Introduction (2e).* MIT Press, 2018. Section 2.3 p. 28 |
| [3] | N.A. “Pendulum-V0”. OpenAI. Available: <https://gym.openai.com/envs/Pendulum-v0/> [Accessed: 2019-03-14] |
| [4] | D. Precup, “Comp-767: Reinforcement Learning – Assignment 2”. 2019, McGill University. Available: https://www.cs.mcgill.ca/~dprecup/courses/RL/Lectures/rl-hw2-2019.pdf [Accessed: 2019-03-14] |