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:** |  |

Contents

[I - Experimental Setup 3](#_Toc4342324)

[1 - Implementing a Simple RNN 3](#_Toc4342325)

[1.1 Methodology 3](#_Toc4342326)

[1.2 Source Code 3](#_Toc4342327)

[2 Implement RNN with Gated Recurrent Units (GRU) 7](#_Toc4342328)

[2.1 Methodology 7](#_Toc4342329)

[2.2 Source Code 7](#_Toc4342330)

[3 Attention Module of Transformer Network 8](#_Toc4342331)

[3.1 Methodology 8](#_Toc4342332)

[3.2 Source Code 9](#_Toc4342333)

[4 Training Language Models 11](#_Toc4342334)

[4.1 Model Comparison Results 11](#_Toc4342335)

[4.1.1 Summary 11](#_Toc4342336)

[4.1.2 Model Comparison Learning Curves 13](#_Toc4342337)

[4.2 Exploration of Optimizers Results 15](#_Toc4342338)

[4.2.1 Summary 15](#_Toc4342339)

[4.2.2 Exploration of Optimizers Learning Curves 17](#_Toc4342340)

[4.3 Hyper-Parameter Search Results 20](#_Toc4342341)

[4.3.1 Summary 20](#_Toc4342342)

[4.3.2 Hyper-Parameter Search Learning Curves 21](#_Toc4342343)

[4.4 Table Summary 27](#_Toc4342344)

[4.5 Results by Optimizer 27](#_Toc4342345)

[4.5.1 SGD\_LR\_Schedule Optimizer Results 27](#_Toc4342346)

[4.5.2 SGD Optimizer Results 29](#_Toc4342347)

[4.5.3 ADAM Optimizer Results 30](#_Toc4342348)

[4.6 Results by Architecture 31](#_Toc4342349)

[4.6.1 RNN Architecture Results 31](#_Toc4342350)

[4.6.2 RNN Architecture Results (Outlier Removed) 32](#_Toc4342351)

[4.6.3 GRU Architecture Results 33](#_Toc4342352)

[4.6.4 Transformer Architecture Results 34](#_Toc4342353)

[4.7 Experiment Commands 35](#_Toc4342354)

[4.8 Discussion 36](#_Toc4342355)

[5 Detailed Evaluation of Trained Models 37](#_Toc4342356)

[5.1 Average Loss per Time-Step 38](#_Toc4342357)

[5.2 Gradient per Time-Step 38](#_Toc4342358)

[5.3 Generation of Samples 39](#_Toc4342359)

[Appendix – All Generated Samples (5.3) 40](#_Toc4342360)

[References 40](#_Toc4342361)

# 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.

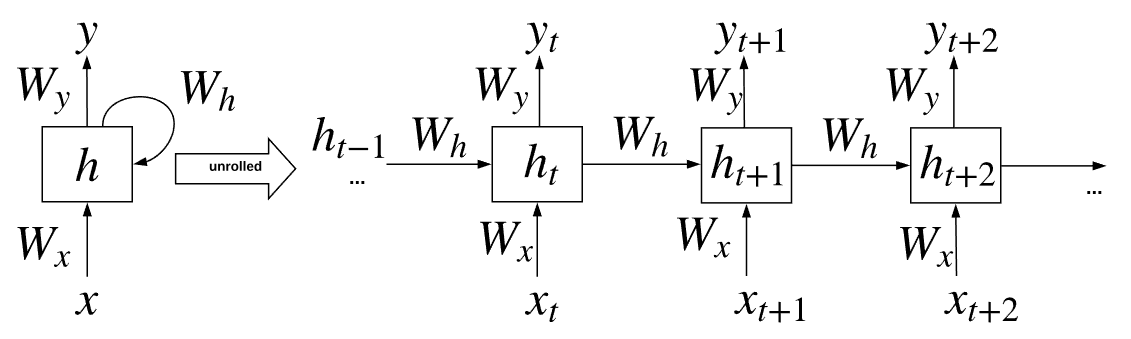


Figure 1 – Illustration of the connections in a Recurrent Neural Network

## 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.

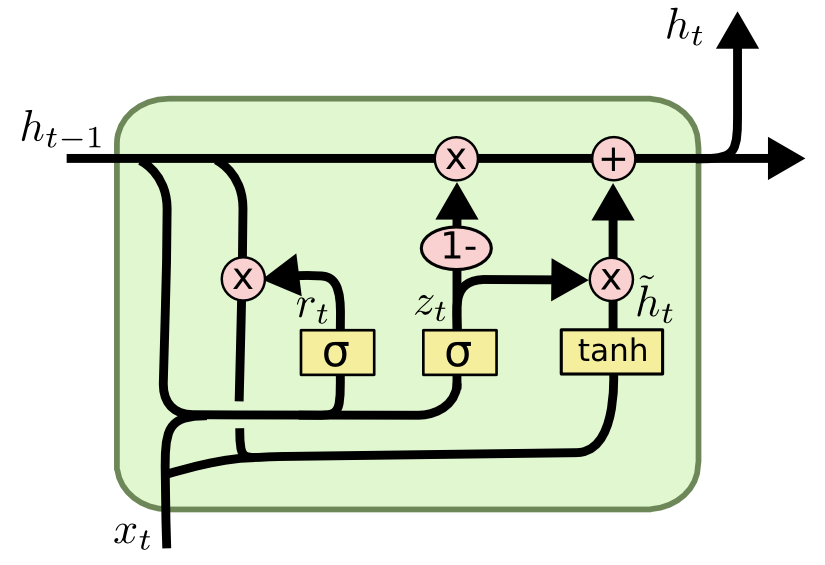


Figure 2 – Illustration of GRU cell containing reset and forget gates [1]

## 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 – 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 |
| wct | Wall-clock time |

## Model Comparison Results

In this section of the experiment the performance of the RNN, GRU and transformer architectures are tested. The validation perplexity, training perplexity and wall-clock time over epochs are recorded.

### Summary

Table 2 summarizes the results obtained for each experiment. The GRU architecture obtained the best performance (best validation perplexity) however takes significantly longer to run (higher average wall-clock time). Conversely, the un-augmented RNN architecture gave the worst performance in both training and validation perplexities as well are training time. The transformer attained middle-ground performance: it did not possess the lowest validation perplexity however its training time (wall-clock time) was significantly faster than either previous models. The slower timings of the RNN and GRU models are most likely due to the costly recurrent connections (1 connection per timestep) as well as the additional training parameters of the gates in the GRU model. The transformer also appears noisier in its learning curve, at least initially.

Table – Results of experiments in Problem 1. Validation ppl. is best obtained. Train ppl. is value at best validation ppl.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| model | optimizer | init\_lr | batch\_size | seq\_len | h\_size | n\_layers | dp\_kp\_prb | avg\_wct | train\_ppl | val\_ppl |
| GRU | SGD LR  Schedule | 10 | 20 | 35 | 1500 | 2 | 0.35 | 148.09 | 65.72 | 102.38 |
| RNN | ADAM | 0.0001 | 20 | 35 | 1500 | 2 | 0.35 | 84.58 | 121.74 | 156.41 |
| TX | SGD LR  Schedule | 20 | 128 | 35 | 512 | 6 | 0.9 | 62.34 | 65.83 | 145.42 |

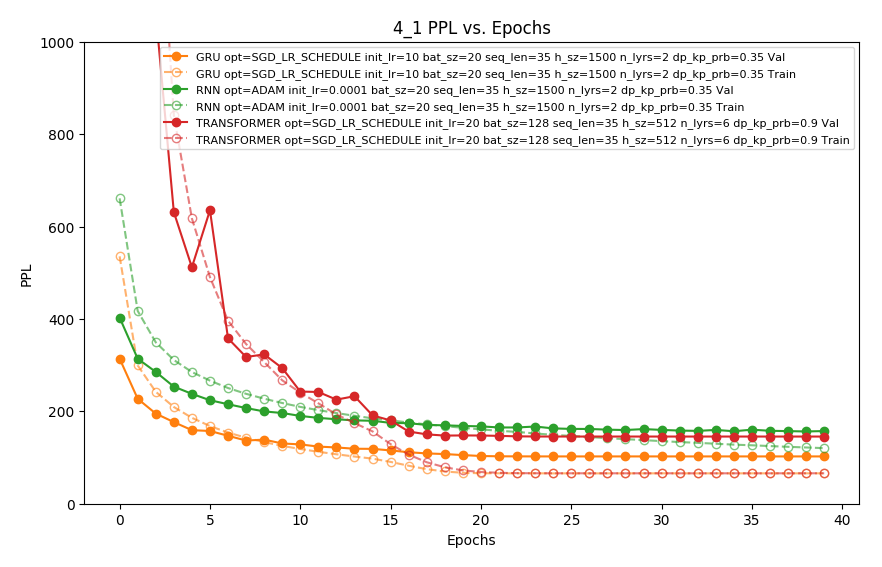


Figure – Training and validation perplexity vs epochs for experiments in problem 4.1

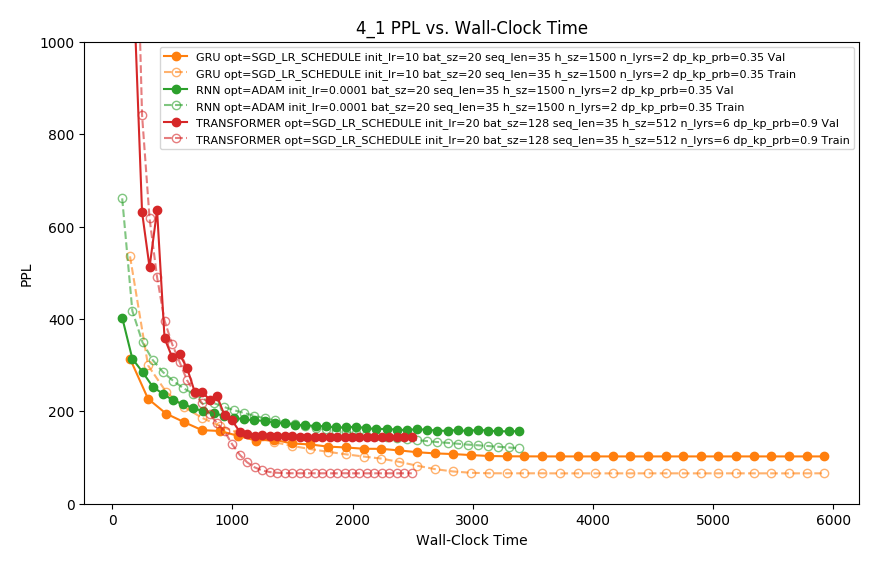
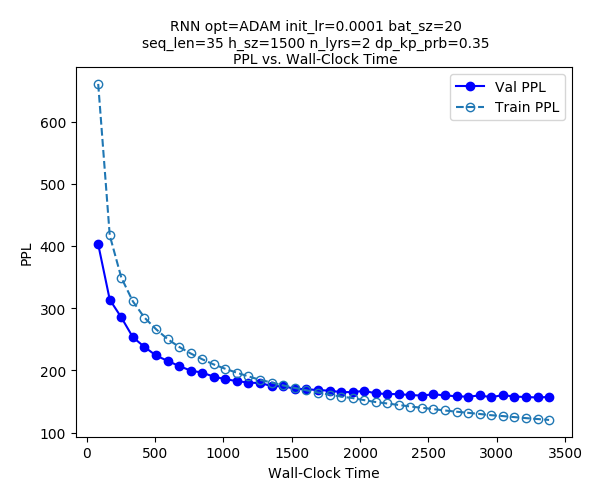
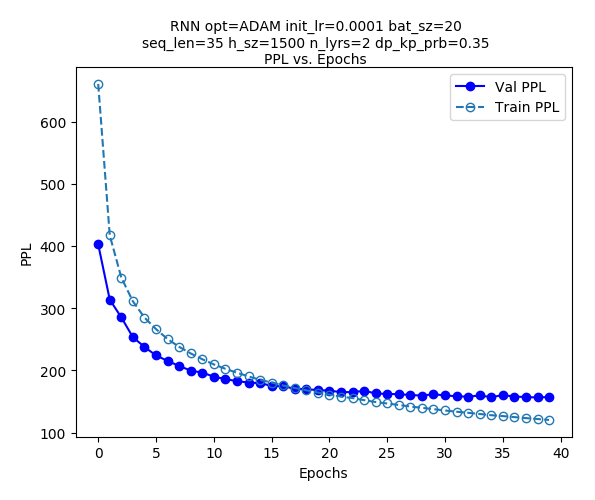


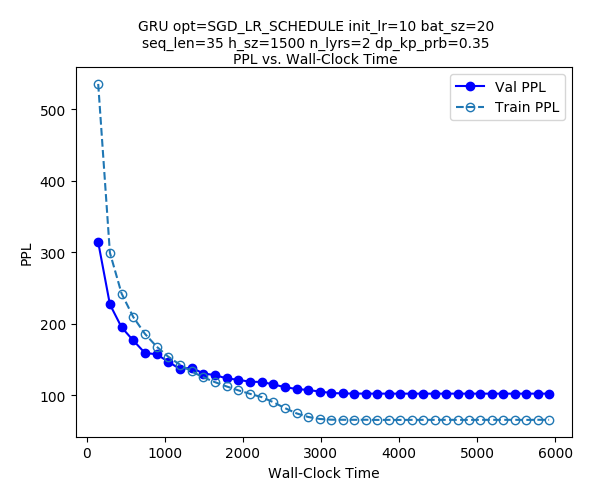
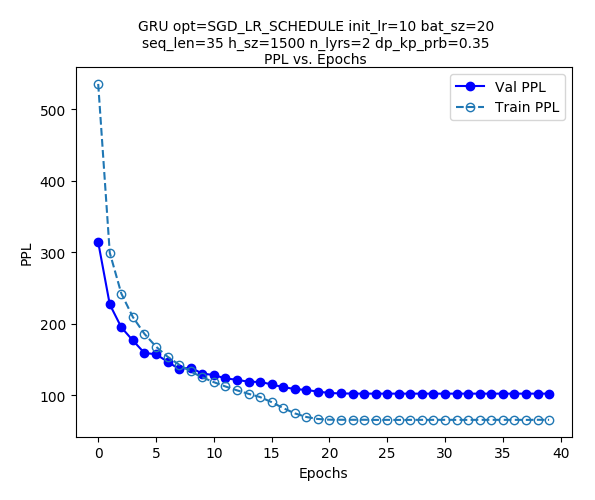
Figure - Training and validation perplexity vs wall-clock time for experiments in problem 4.1

### Model Comparison Learning Curves

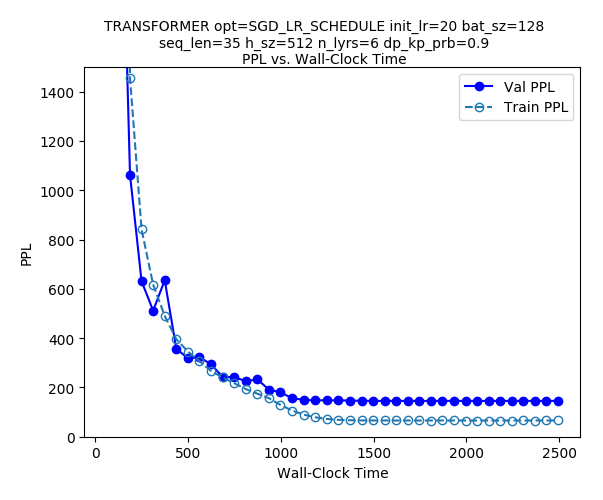
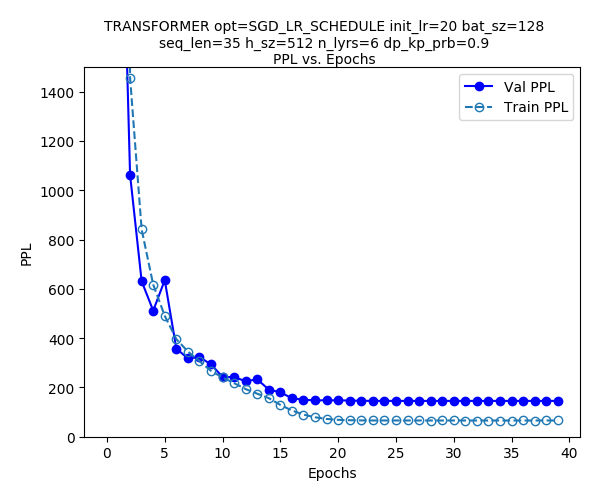
Learning curves for each of the individual experiments of Problem 4.1 (which are combined in Figure 3 and Figure 4) are plotted here. They show validation perplexity and training perplexity versus epochs and wall-clock time. Most learning curves in this section do not exhibit much if any overfitting.



|  |  |
| --- | --- |
| Figure – PPL vs. epochs RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs GRU model, SGD\_LR\_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time GRU model, SGD\_LR\_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs Transformer model, SGD\_LR\_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.9 | Figure - PPL vs. wall-clock time Transformer model, SGD\_LR\_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.9 |

## Exploration of Optimizers Results

In addition to the experiments in section 4.1, additional optimizers (the 2 optimizers missing for each model of section 4.1) are explored for each architecture.

### Summary

Table 2 summarizes the results. Some interesting trends were noted.

The RNN with SGD optimizer fails to reach a comparable validation perplexity performance to the other models (stopping at 2209.92 after 40 epochs). Though this is typically for stochastic gradient descent with a fixed and too small learning rate. In general, a larger learning rate can help reach a minimum in the cost function faster though it carries some risk of oscillating around the goal and not converging. Too small of a learning rate however results in a model that takes too long to learn (as seen here). An adaptive style of learning rate may be best in this case.

The RNN model with SGD LR schedule also had its hidden size reduced to 512 (from 1500) which appears too small and so the model exhibits some high bias and performs poorly on both training and validation sets.

The transformer with Adam optimizer performs well with early stopping however it quickly overfits the data. Likewise, the transformer with SGD optimizer also exhibits some overfitting. The Adam and SGD optimizers with the GRU model perform reasonably well but both fail to beat the 102.38 validation perplexity in section 4.1 with the SGD LR Schedule optimizer.

Table – Results of experiments in Problem 2. Validation ppl. is best obtained. Train ppl. is value at best validation ppl.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| model | optimizer | init\_lr | batch\_size | seq\_len | h\_size | n\_layers | dp\_kp\_prb | avg\_wct | train\_ppl | val\_ppl |
| GRU | ADAM | 0.0001 | 20 | 35 | 1500 | 2 | 0.35 | 155.59 | 73.86 | 111.49 |
| GRU | SGD | 10 | 20 | 35 | 1500 | 2 | 0.35 | 145.03 | 68.79 | 112.06 |
| RNN | SGD | 0.0001 | 20 | 35 | 1500 | 2 | 0.35 | 80.68 | 3008.12 | 2209.92 |
| RNN | SGD LR  Schedule | 1 | 20 | 35 | 512 | 2 | 0.35 | 48.62 | 230.78 | 196.19 |
| TX | ADAM | 0.001 | 128 | 35 | 512 | 2 | 0.9 | 30.13 | 69.57 | 136.22 |
| TX | SGD | 20 | 128 | 35 | 512 | 6 | 0.9 | 61.67 | 79.57 | 161.59 |

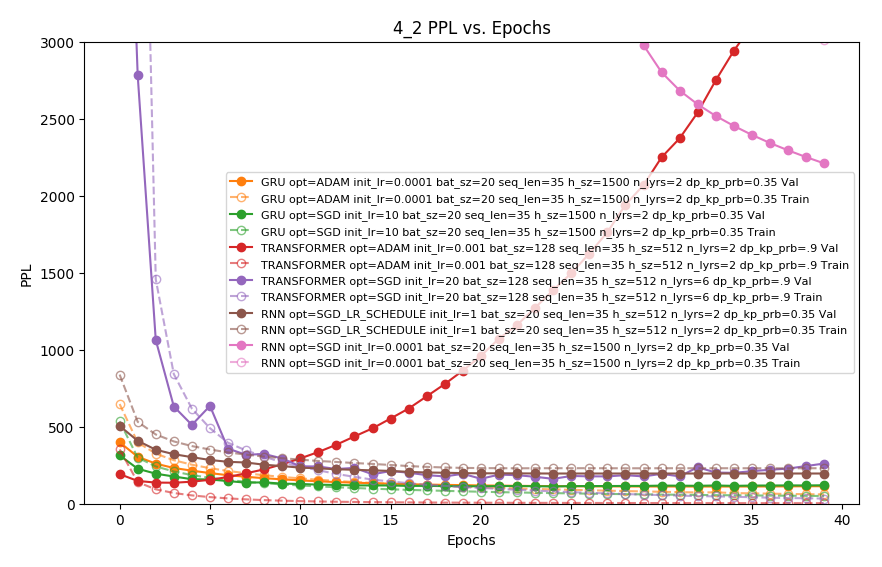


Figure – Training and validation perplexity vs. epochs for architectures in problem 4.2

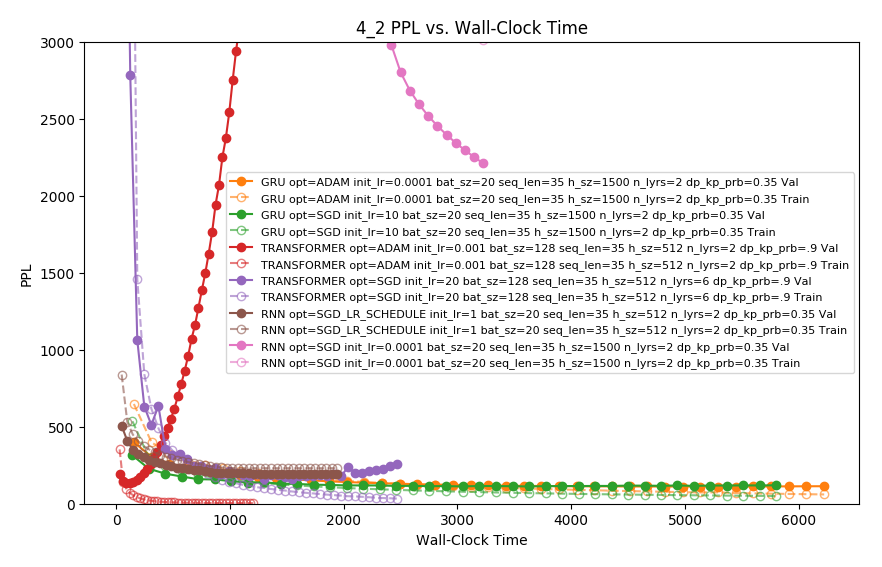
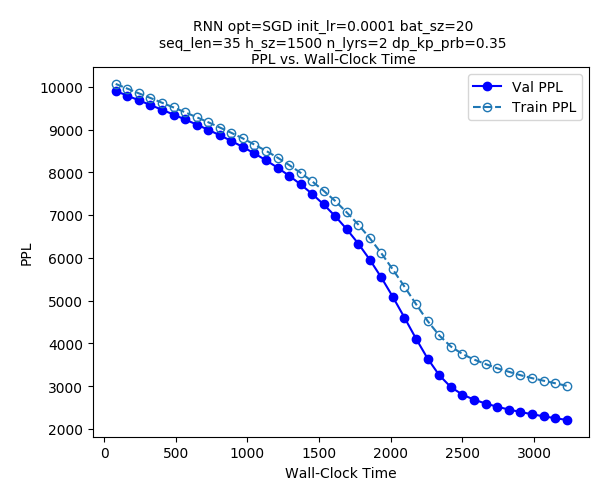
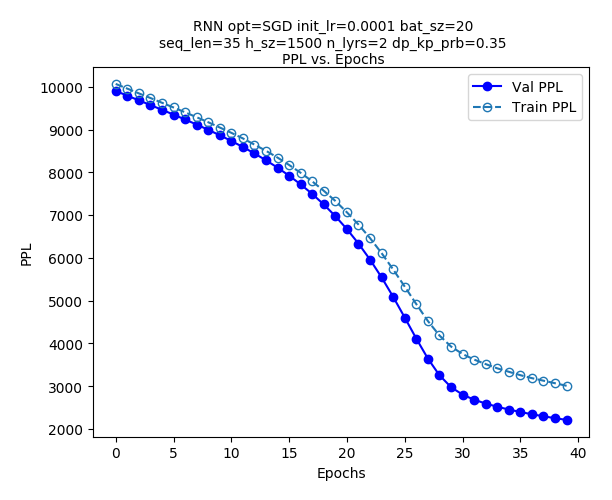
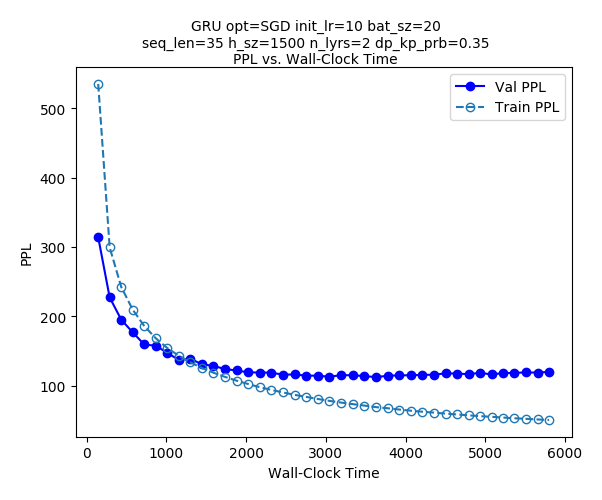
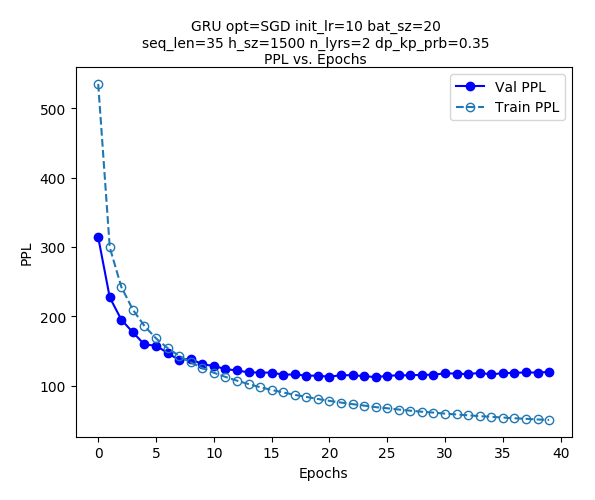


Figure – Training and validation perplexity vs. wall-clock time for architectures in problem 4.2

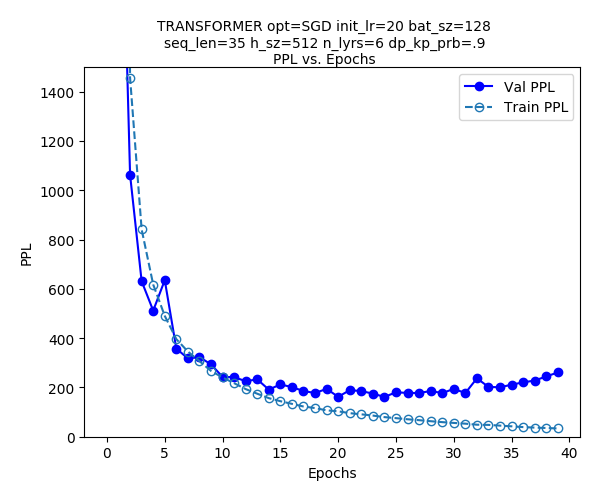
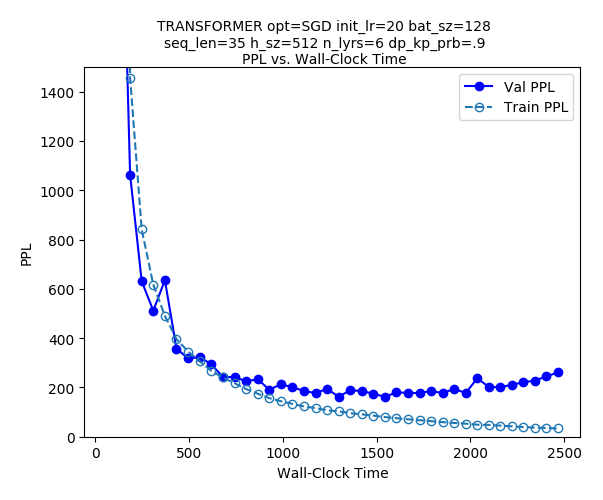
### Exploration of Optimizers Learning Curves



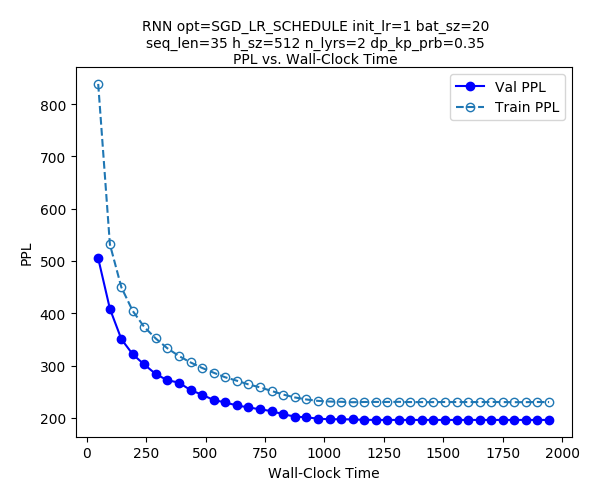
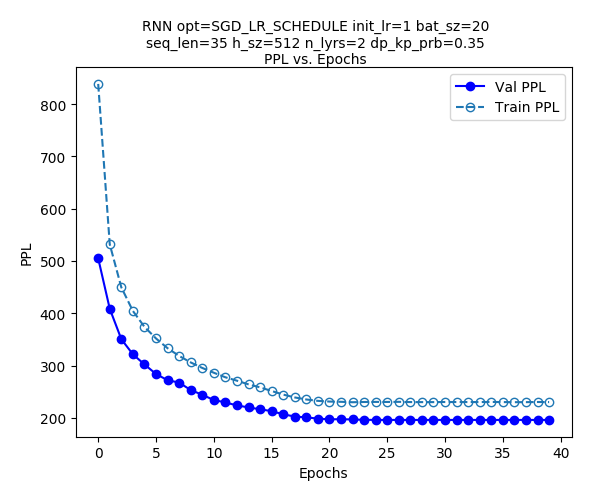
|  |  |
| --- | --- |
| Figure - PPL vs. epochs RNN model, SGD optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time RNN model, SGD optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 |



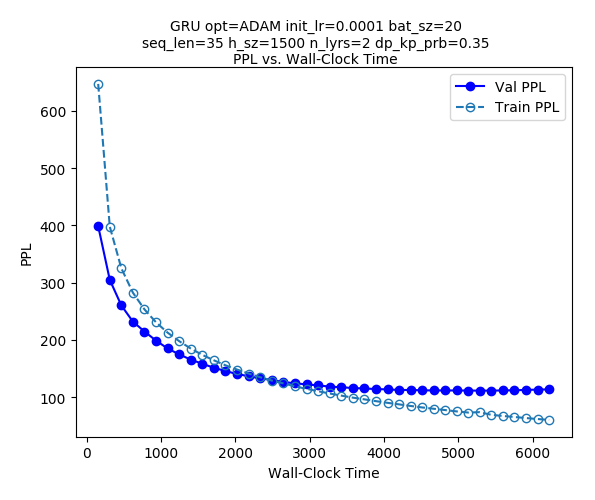
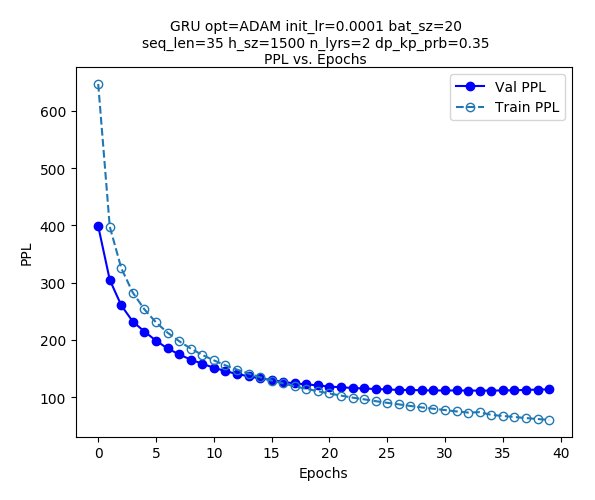
|  |  |
| --- | --- |
| Figure - PPL vs. epochs GRU model, SGD optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time GRU model, SGD optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 |



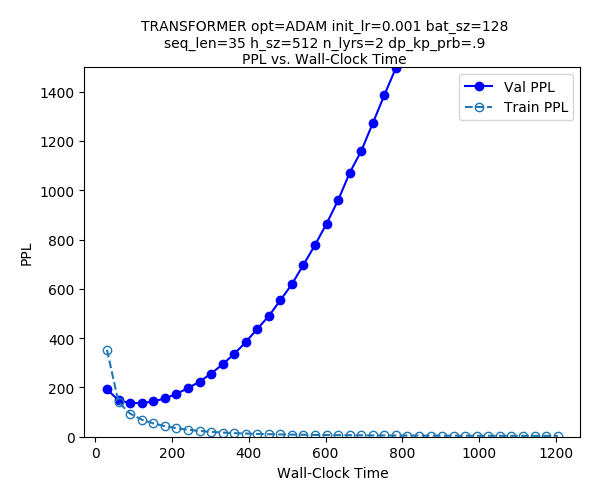
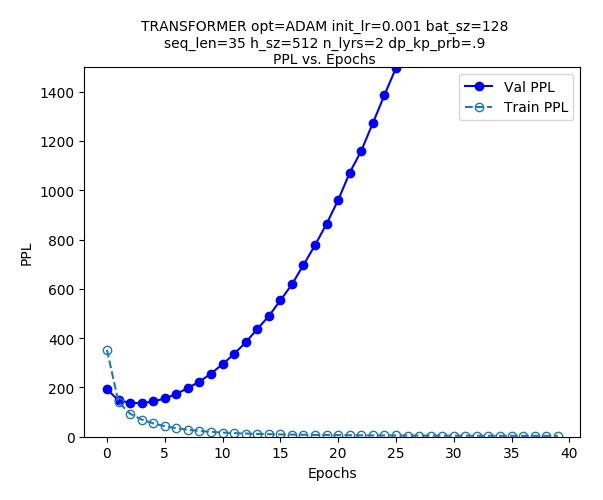
|  |  |
| --- | --- |
| Figure - PPL vs. epochs Transformer model, SGD optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.9 | Figure - PPL vs. wall-clock time Transformer model, SGD optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.9 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs RNN model, SGD\_LR\_Schedule optimizer, init. learning rate=1, batch size=20, sequence length=35, hidden size=512, layers=2, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock tiem RNN model, SGD\_LR\_Schedule optimizer, init. learning rate=1, batch size=20, sequence length=35, hidden size=512, layers=2, dropout keep prob.=0.35 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs GRU model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time GRU model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs Transformer model, ADAM optimizer, init. learning rate=0.001, batch size=20, sequence length=35, hidden size=512, layers=2, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time Transformer model, ADAM optimizer, init. learning rate=0.001, batch size=20, sequence length=35, hidden size=512, layers=2, dropout keep prob.=0.35 |

## Hyper-Parameter Search Results

Additional hyper-parameters configurations were tested to obtain better performance than the baseline models in problem 4.1. Different values for hidden state size, number of layers and dropout probabilities were tested. The changes made to each baseline model from 4.1 are listed in brackets in **red**. An additional run of the SGD optimizer for each architecture was also done to obtain more results with this optimizer for later analysis (section 4.5). Differences to the validation performance for the baseline models in problem 4.1 are reported in the last column.

Perhaps surprisingly, many parameter settings with less complex configurations (reduces layers or number of hidden units) resulted in better or performance. This perhaps indicates that in some aspect previous 4.1 models were overfitting by the additional complexity in their structure and could not generalize well enough to the validation set.

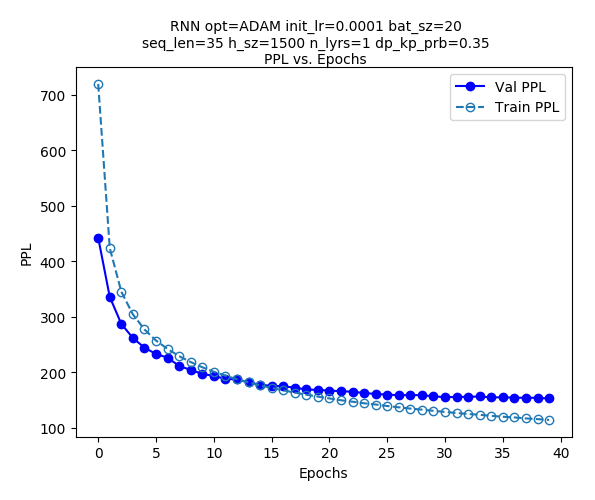
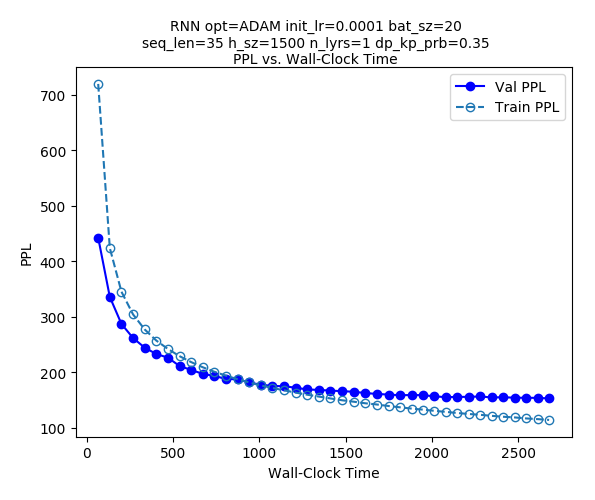
### Summary

Table – Results of experiments in Hyper-Parameter Search. Valid. ppl. is best obtained. Train ppl. is value at best valid. ppl. The difference to the validation performance of

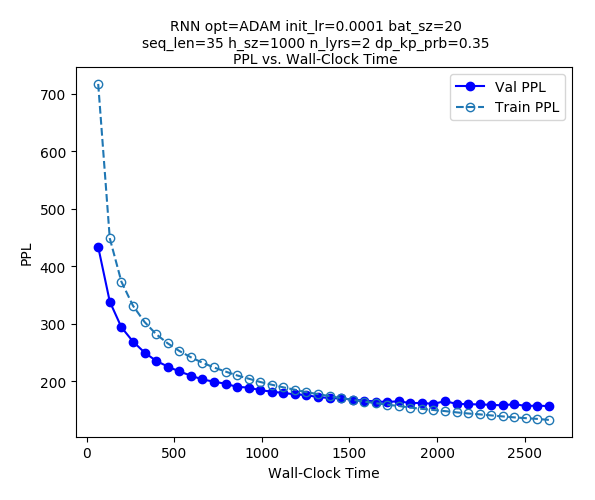
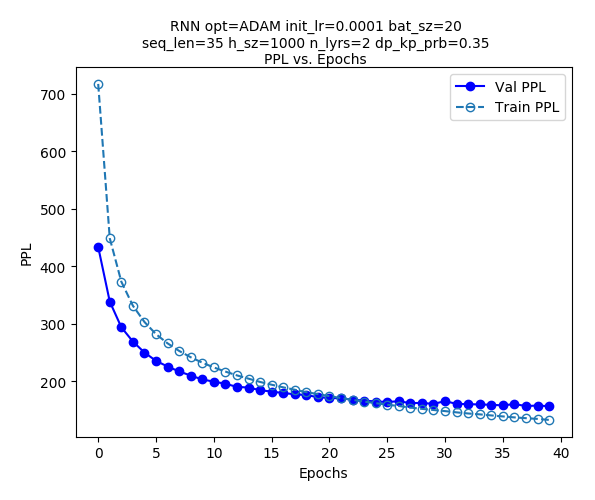
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| model | optimizer | init\_lr | bat\_sz | seq\_len | h\_size | n\_layer | dp\_kp\_prb | avg\_wct | train  ppl | val  ppl | to 4.1 val ppl |
| GRU | SGD LR  Schedule | 10 | 20 | 35 | 1500 | 1 (-1) | 0.35 | 106.58 | 59.18 | 95.02 | -7.36 |
| GRU | SGD LR  Schedule | 10 | 20 | 35 | 1000  **(-500)** | 2 | 0.35 | 113.17 | 76.51 | 102.88 | +0.5 |
| GRU | SGD LR  Schedule | 10 | 20 | 35 | 1500 | 2 | 0.5  **(+0.15)** | 146.49 | 30.58 | 117.78 | +15.4 |
| GRU | SGD  **(-LR schedule)** | 10 | 20 | 35 | 1500 | 1 **(-1)** | 0.35 | 91.78 | 68.04 | 103.32 | +0.939 |
| RNN | ADAM | 0.0001 | 20 | 35 | 1500 | 2 | 0.5  **(+0.15)** | 85 | 108.59 | 151.47 | -4.94 |
| RNN | ADAM | 0.0001 | 20 | 35 | 1500 | 1 **(-1)** | 0.35 | 66.97 | 114.07 | 152.8 | -3.61 |
| RNN | ADAM | 0.0001 | 20 | 35 | 1000  **(-500)** | 2 | 0.35 | 65.99 | 132.55 | 156.413 | +0.006 |
| RNN | SGD  **(-ADAM)** | 1 | 20 | 35 | 1500 | 2 | 0.35 | 80.79 | 185.17 | 170.55 | +14.14 |
| TX | SGD LR  Schedule | 20 | 128 | 35 | 256  **(-256)** | 6 | 0.9 | 52.4 | 81.5 | 139.71 | -5.71 |
| TX | SGD LR  Schedule | 20 | 128 | 35 | 512 | 6 | 0.7  **(-0.2)** | 54.12 | 80.8 | 141.59 | -3.83 |
| TX | SGD LR  Schedule | 20 | 128 | 35 | 512 | 8 **(+2)** | 0.9 | 69.18 | 55.43 | 144.19 | -1.23 |
| TX | SGD  **(-LR schedule)** | 20 | 128 | 35 | 256  **(-256)** | 6 | 0.9 | 54.22 | 59.66 | 173.39 | +27.97 |

### Hyper-Parameter Search Learning Curves

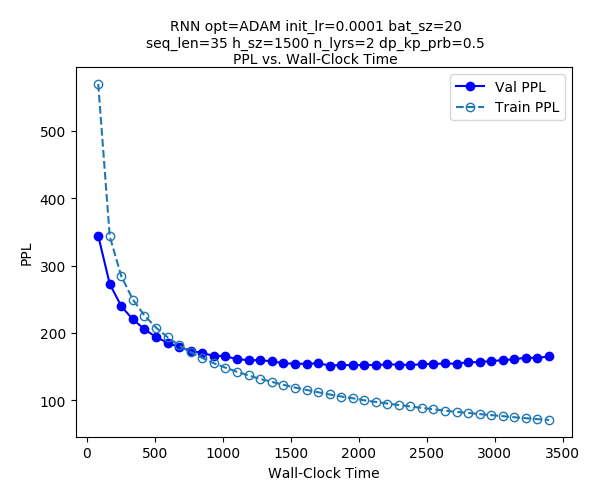
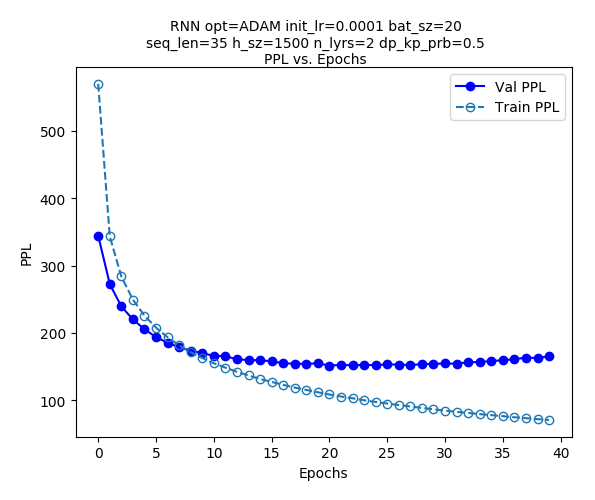
#### RNN Architectures



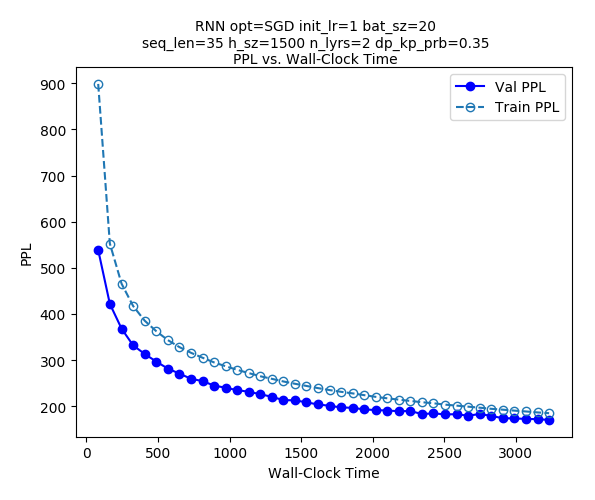
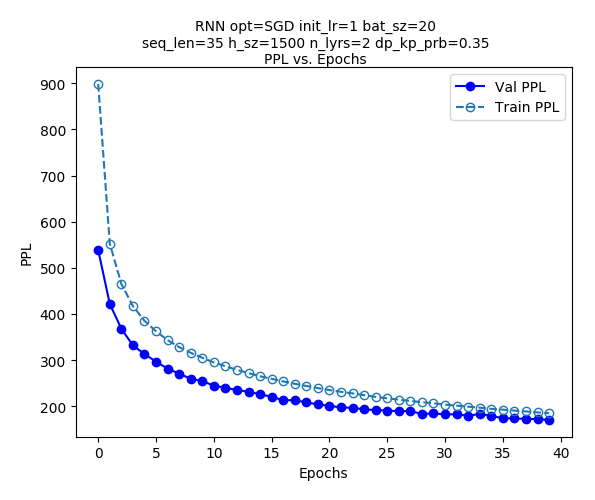
|  |  |
| --- | --- |
| Figure - PPL vs. epochs RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.35 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1000, layers=2, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1000, layers=2, dropout keep prob.=0.35 |

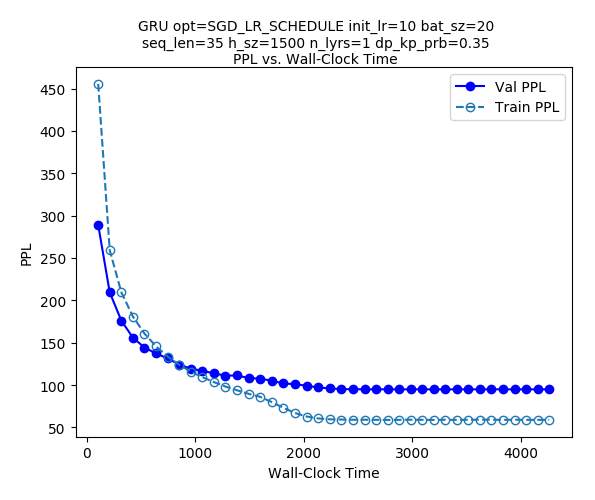
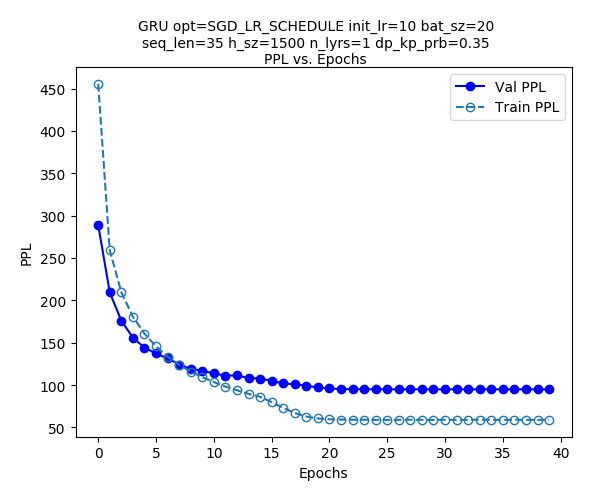


|  |  |
| --- | --- |
| Figure - PPL vs. epochs RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.5 | Figure - PPL vs. wall-clock time RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.5 |

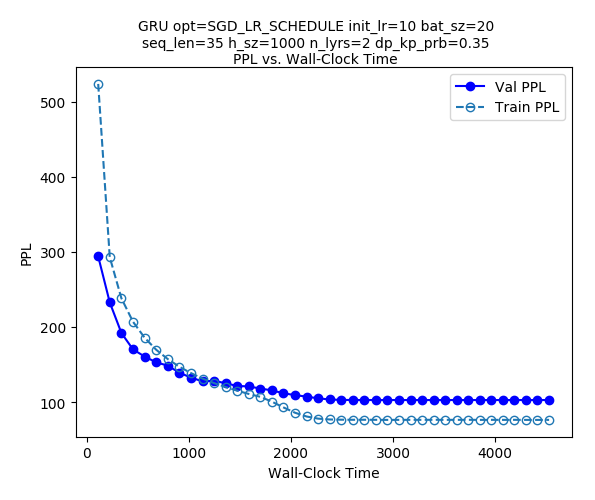
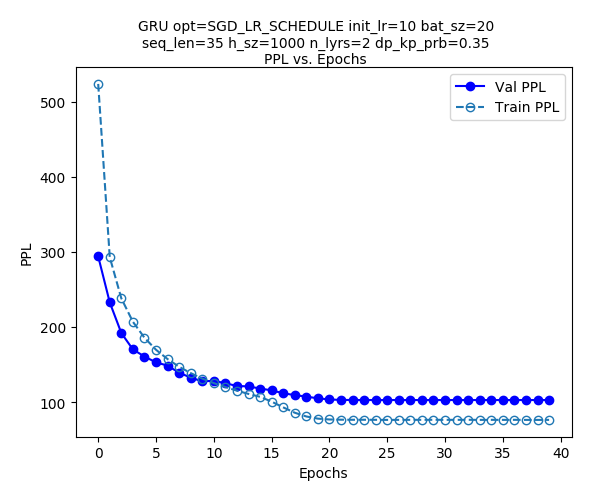


|  |  |
| --- | --- |
| Figure - PPL vs. epochs RNN model, SGD optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time RNN model, SGD optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.35 |

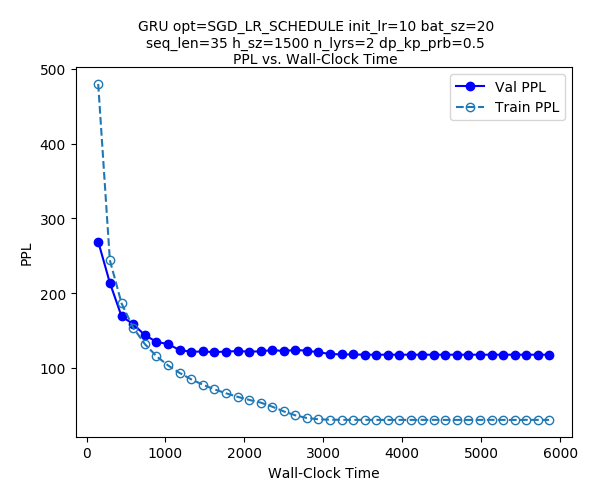
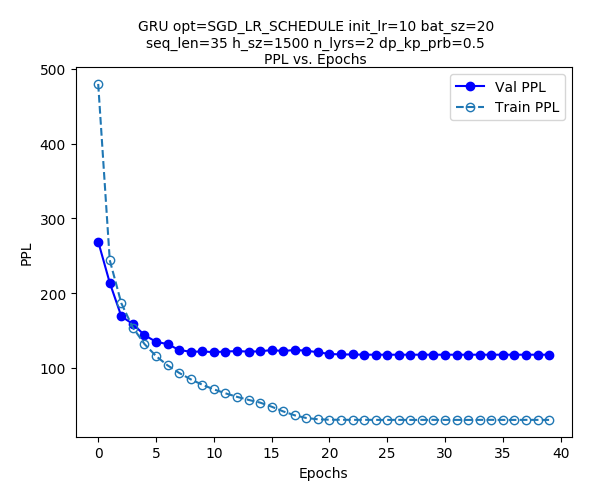
#### GRU Architectures



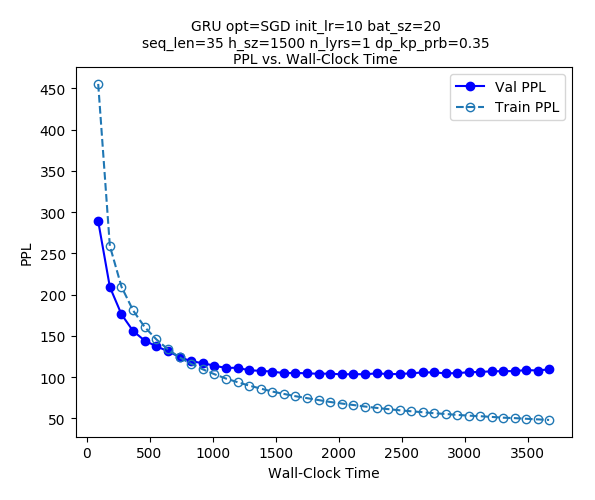
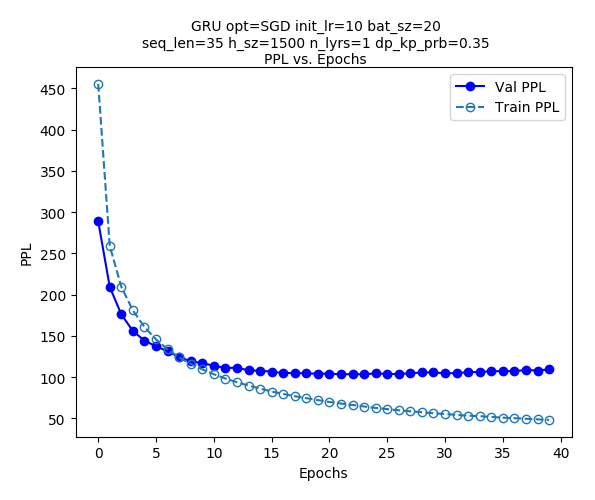
|  |  |
| --- | --- |
| Figure - PPL vs. epochs GRU model, SGD\_LR\_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.35 | Figure - PPL vs. wall-clock time GRU model, SGD\_LR\_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.35 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs GRU model, SGD\_LR\_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1000, layers=1, dropout keep prob.=0.5 | Figure - PPL vs. wall-clock time GRU model, SGD\_LR\_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1000, layers=1, dropout keep prob.=0.5 |

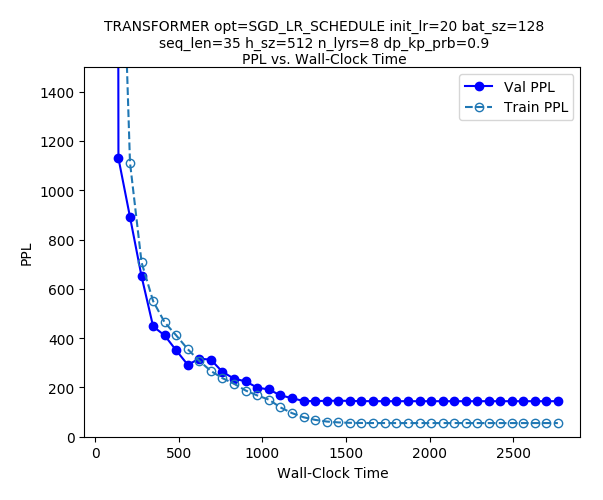
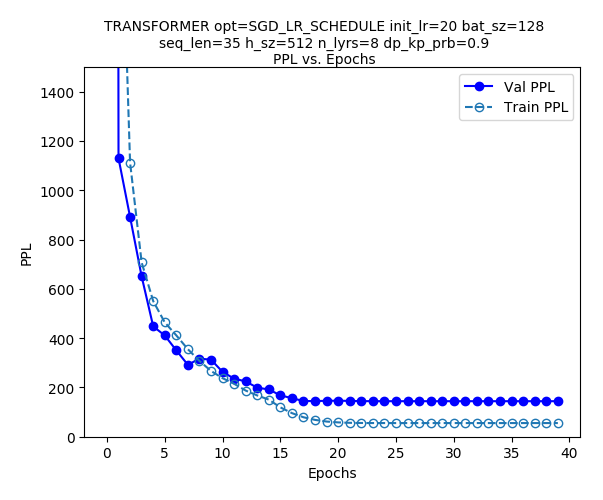


|  |  |
| --- | --- |
| Figure - PPL vs. epochs GRU model, SGD\_LR\_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.5 | Figure - PPL vs. wall-clock time GRU model, SGD\_LR\_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.5 |

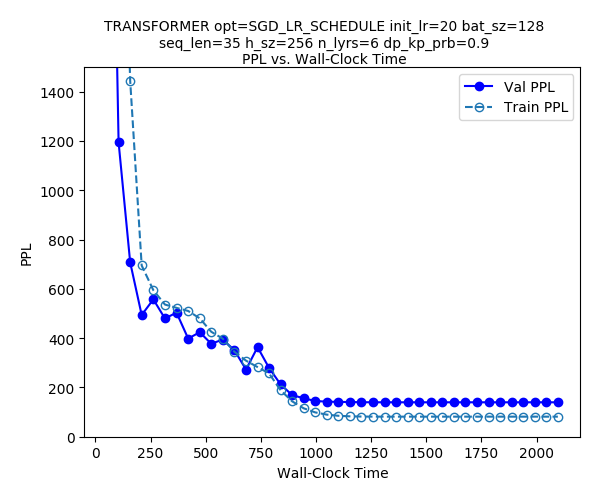
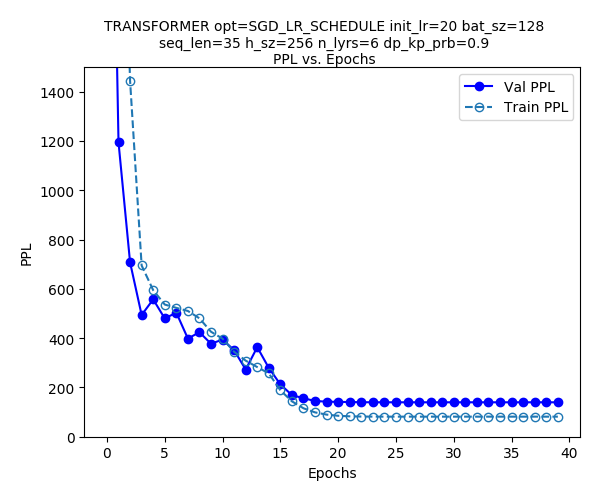


|  |  |
| --- | --- |
| Figure - PPL vs. epochs GRU model, SGD optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.5 | Figure - PPL vs. wall-clock time GRU model, SGD optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=1, dropout keep prob.=0.5 |

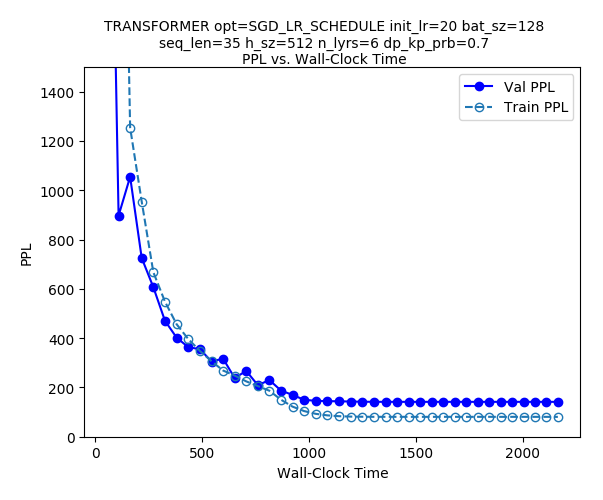
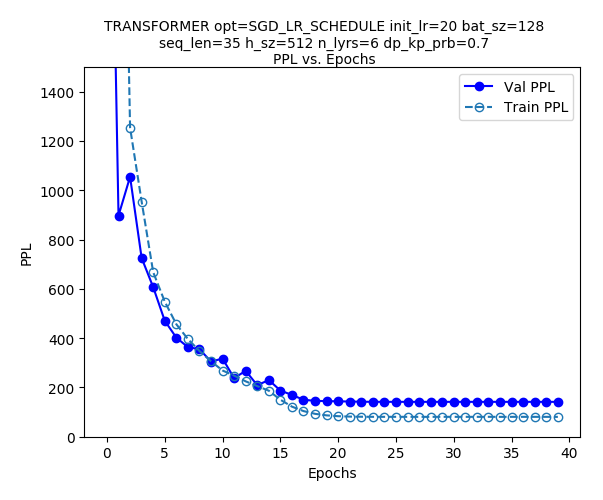
#### Transformer Architectures



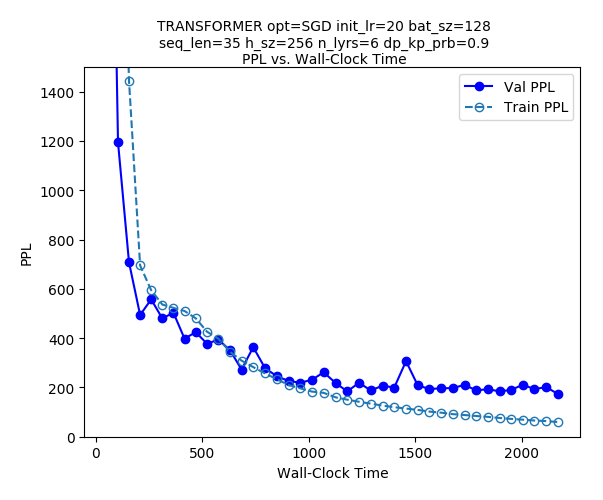
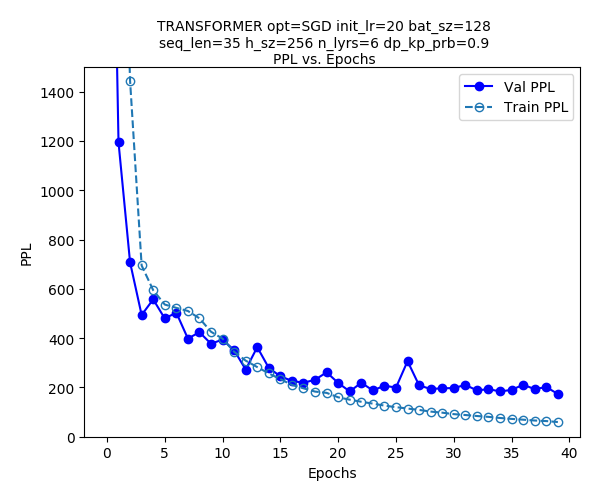
|  |  |
| --- | --- |
| Figure - PPL vs. epochs Transformer model, SGD\_LR\_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=256, layers=8, dropout keep prob.=0.9 | Figure - PPL vs. wall-clock time Transformer model, SGD\_LR\_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=256, layers=8, dropout keep prob.=0.9 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs Transformer model, SGD\_LR\_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=256, layers=6, dropout keep prob.=0.9 | Figure - PPL vs. wall-clock time Transformer model, SGD\_LR\_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=256, layers=6, dropout keep prob.=0.9 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs Transformer model, SGD\_LR\_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.7 | Figure - PPL vs. wall-clock time Transformer model, SGD\_LR\_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.7 |



|  |  |
| --- | --- |
| Figure - PPL vs. epochs Transformer model, SGD optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=256, layers=6, dropout keep prob.=0.9 | Figure - PPL vs. wall-clock time Transformer model, SGD optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=256, layers=6, dropout keep prob.=0.9 |

## Table Summary

All results among all experiments (problems 4.1-4.3) are summarized in Table 2. The best results for each architecture are bolded.

Table – Summary of all results. Architectures with the best validation perplexity are bolded. Training perplexity is that reported at the epoch of the best validation perplexity.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| model | optimizer | init\_lr | batch\_size | seq\_len | h\_size | n\_layers | dp\_kp\_prb | avg\_wct | train\_ppl | val\_ppl |
| GRU | ADAM | 0.0001 | 20 | 35 | 1500 | 2 | 0.35 | 155.59 | 73.86 | 111.49 |
| GRU | SGD | 10 | 20 | 35 | 1500 | 1 | 0.35 | 91.78 | 68.04 | 103.32 |
| GRU | SGD | 10 | 20 | 35 | 1500 | 2 | 0.35 | 145.03 | 68.79 | 112.06 |
| **GRU** | **SGD LR**  **Schedule** | **10** | **20** | **35** | **1500** | **1** | **0.35** | **106.58** | **59.18** | **95.02** |
| GRU | SGD LR  Schedule | 10 | 20 | 35 | 1500 | 2 | 0.35 | 148.09 | 65.72 | 102.38 |
| GRU | SGD LR  Schedule | 10 | 20 | 35 | 2000 | 2 | 0.35 | 192.55 | 59.66 | 104.17 |
| GRU | SGD LR  Schedule | 10 | 20 | 35 | 1500 | 2 | 0.5 | 146.49 | 30.58 | 117.78 |
| **RNN** | **ADAM** | **0.0001** | **20** | **35** | **1500** | **2** | **0.5** | **85** | **108.59** | **151.47** |
| RNN | ADAM | 0.0001 | 20 | 35 | 1500 | 1 | 0.35 | 66.97 | 114.07 | 152.8 |
| RNN | ADAM | 0.0001 | 20 | 35 | 1500 | 2 | 0.35 | 84.58 | 121.74 | 156.41 |
| RNN | ADAM | 0.0001 | 20 | 35 | 1000 | 2 | 0.35 | 65.99 | 132.55 | 156.41 |
| RNN | SGD | 1 | 20 | 35 | 1500 | 2 | 0.35 | 80.79 | 185.17 | 170.55 |
| RNN | SGD | 0.0001 | 20 | 35 | 1500 | 2 | 0.35 | 80.68 | 3008.12 | 2209.92 |
| RNN | SGD LR  Schedule | 1 | 20 | 35 | 512 | 2 | 0.35 | 48.62 | 230.78 | 196.19 |
| **TX** | **ADAM** | **0.001** | **128** | **35** | **512** | **2** | **0.9** | **30.13** | **69.57** | **136.22\*** |
| TX | SGD | 20 | 128 | 35 | 512 | 6 | 0.9 | 61.67 | 79.57 | 161.59 |
| TX | SGD | 20 | 128 | 35 | 256 | 6 | 0.9 | 54.22 | 59.66 | 173.39 |
| TX | SGD LR  Schedule | 20 | 128 | 35 | 256 | 6 | 0.9 | 52.4 | 81.5 | 139.71 |
| TX | SGD LR  Schedule | 20 | 128 | 35 | 512 | 6 | 0.7 | 54.12 | 80.8 | 141.59 |
| TX | SGD LR  Schedule | 20 | 128 | 35 | 512 | 8 | 0.9 | 69.18 | 55.43 | 144.19 |
| TX | SGD LR  Schedule | 20 | 128 | 35 | 512 | 6 | 0.9 | 62.34 | 65.83 | 145.42 |

\* Although this model has the best validation perplexity it exhibits high variance (overfitting) later.

## Results by Optimizer

This section graphs results organized by each optimizer (SGD\_LR\_Schedule, SGD and Adam).

### SGD\_LR\_Schedule Optimizer Results

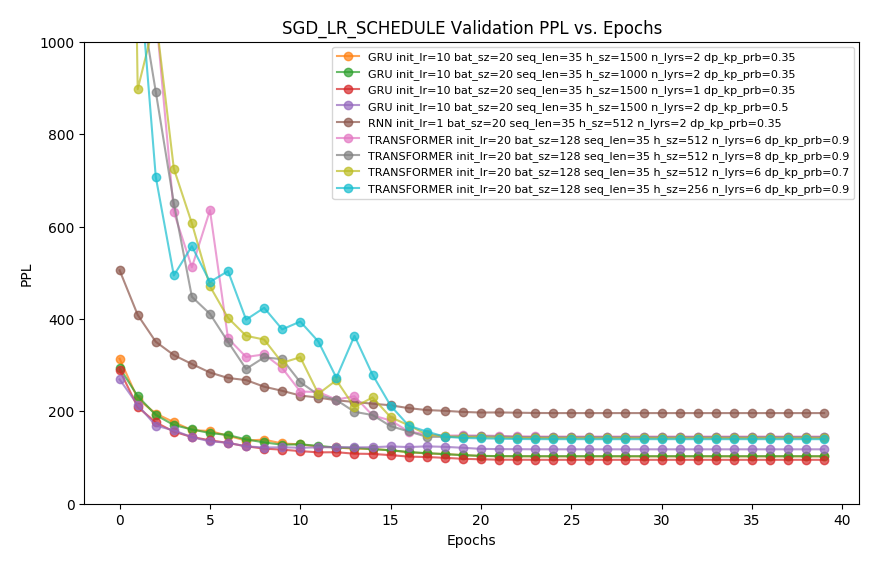


Figure 49 – Validation perplexity vs. epochs for experiments using SGD\_LR\_Schedule optimizer

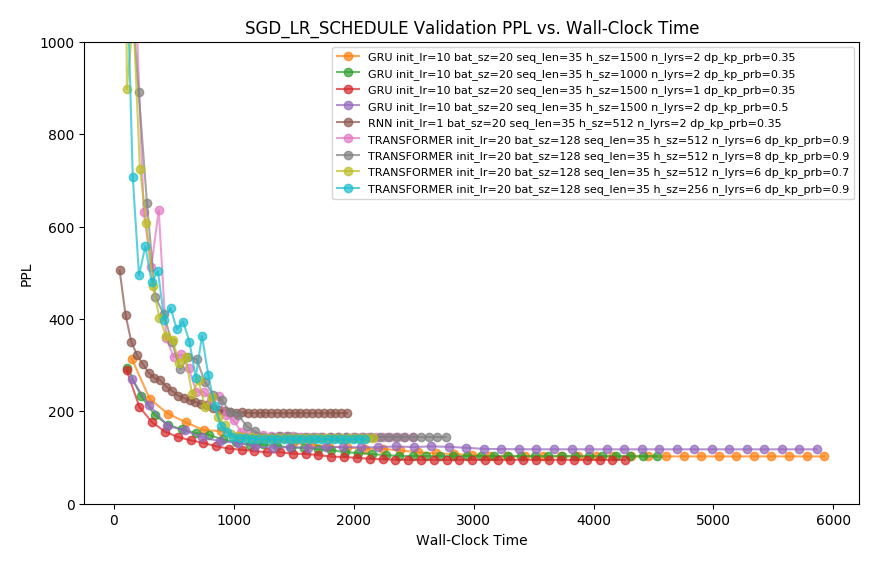


Figure – Validation perplexity vs. wall-clock time for experiments using SGD\_LR\_Schedule optimizer

### SGD Optimizer Results

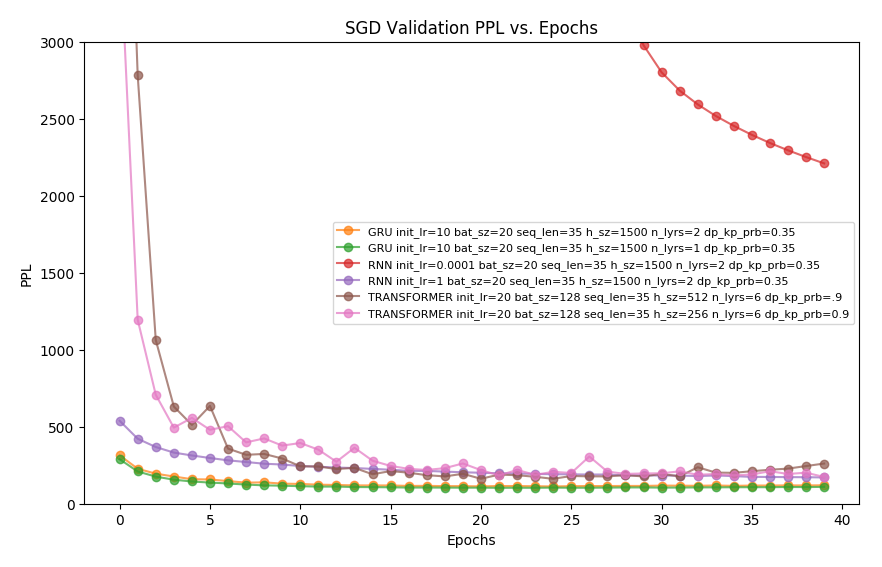


Figure – Validation perplexity vs. epochs for experiments using SGD optimizer

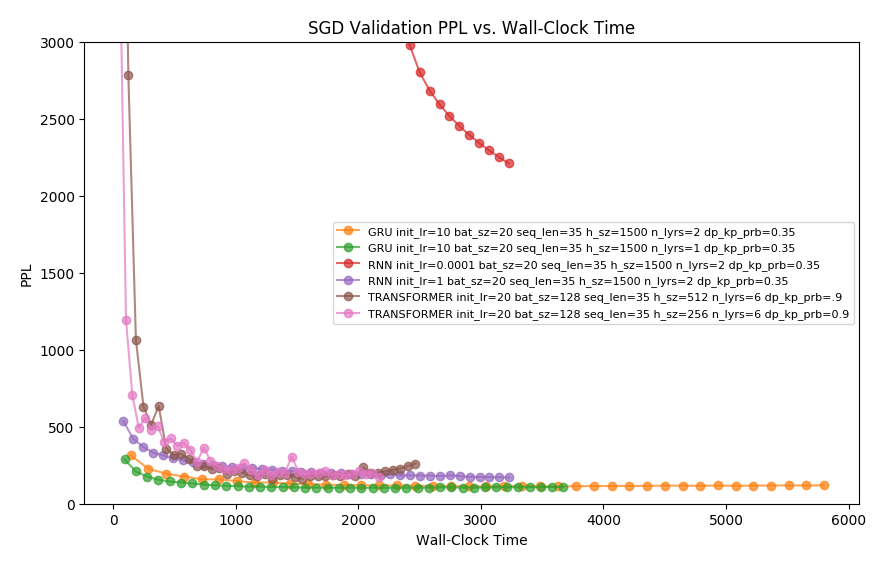


Figure – Validation perplexity vs. wall-clock time for experiments using SGD optimizer

### ADAM Optimizer Results

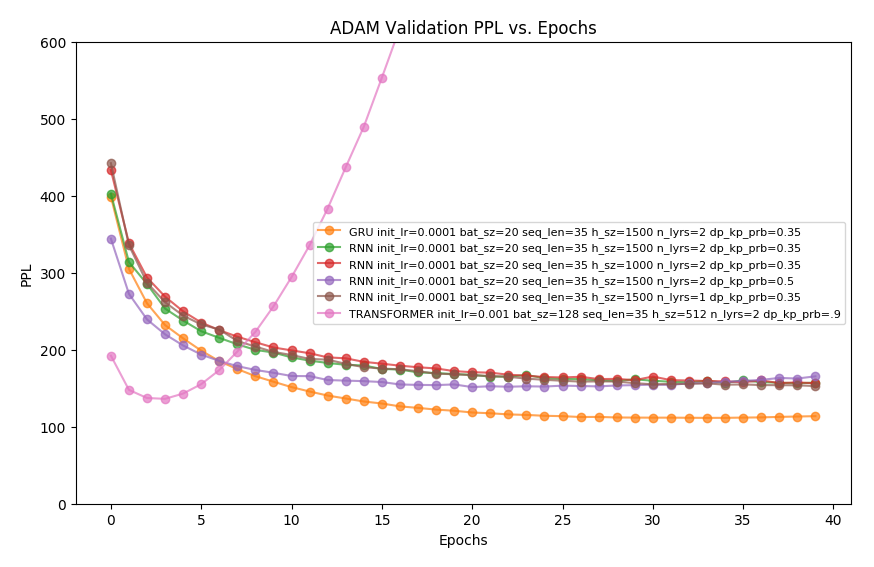


Figure – Validation perplexity vs. epochs for experiments using ADAM optimizer

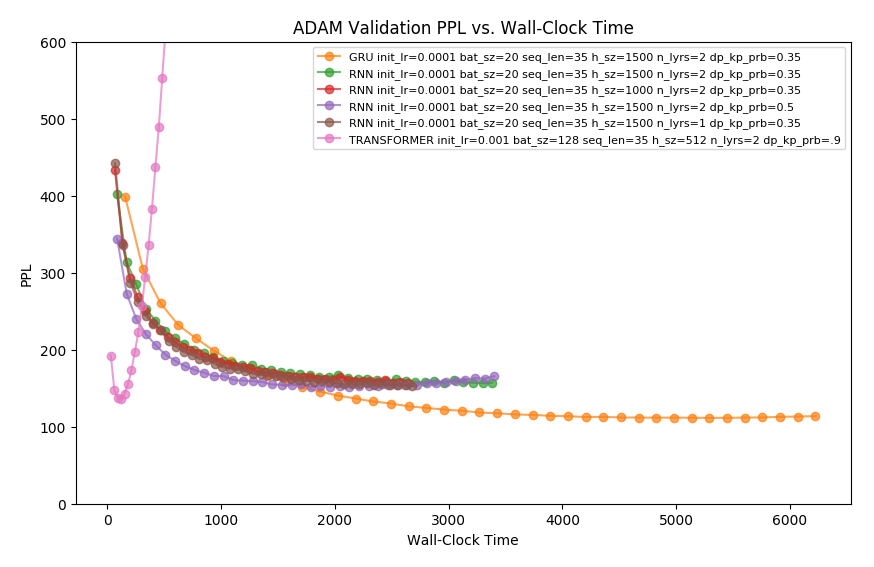


Figure – Validation perplexity vs. wall-clock time for experiments using ADAM optimizer

## Results by Architecture

This section organizes results based on model architecture: RNN, GRU and transformer.

### RNN Architecture Results

Due to the experiment result from problem 4.2 with the RNN architecture and SGD optimizing with a low learning rate, there is an outlier with the rest of the models in Figure 55 and Figure 56 which somewhat skews the y-axis. Additional figures in the next section remove this outlier so the curves of the other RNN are scaled more appropriately.

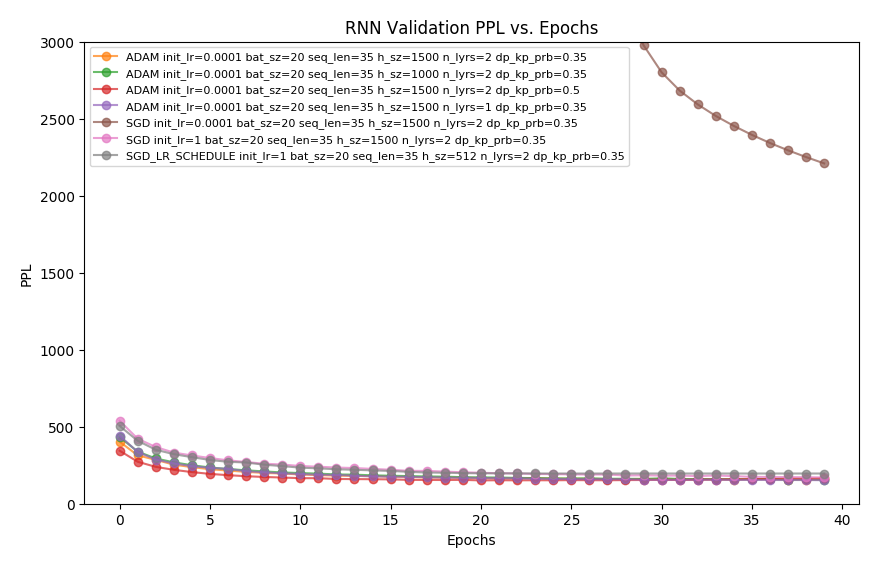


Figure – Validation perplexity vs. wall-clock time for experiments using ADAM optimizer

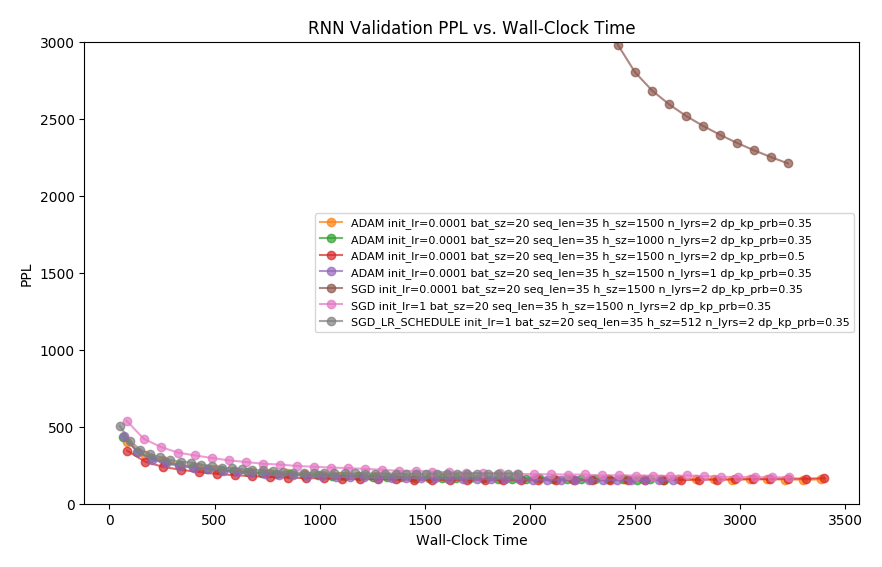


Figure – Validation perplexity vs. wall-clock time for experiments using ADAM optimizer

### RNN Architecture Results (Outlier Removed)

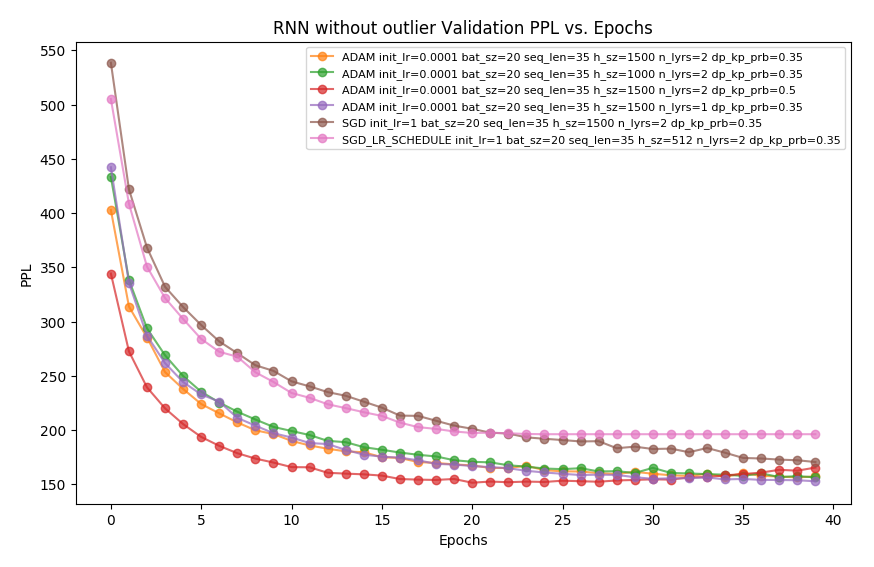


Figure – Validation perplexity vs. wall-clock time for experiments using RNN architecture with outlier removed

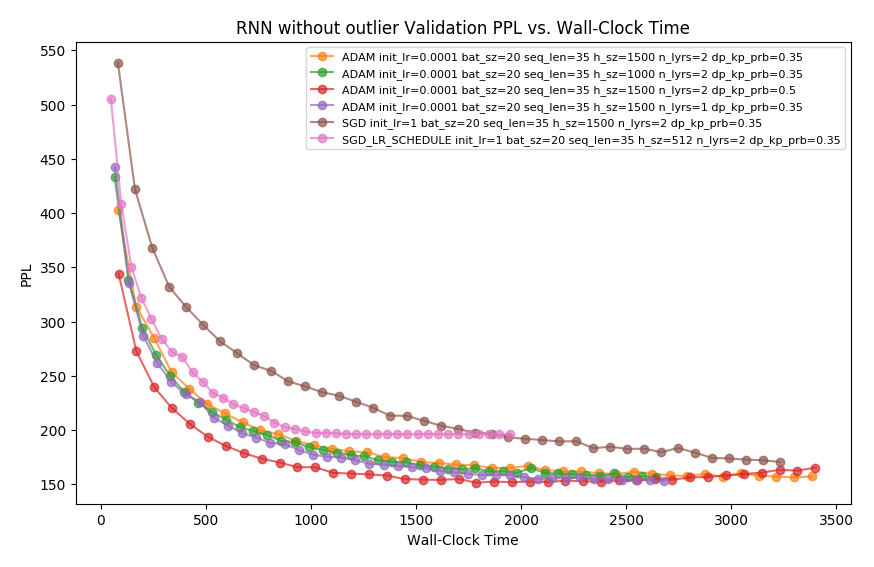


Figure – Validation perplexity vs. wall-clock time for experiments using RNN architecture with outlier removed

### GRU Architecture Results

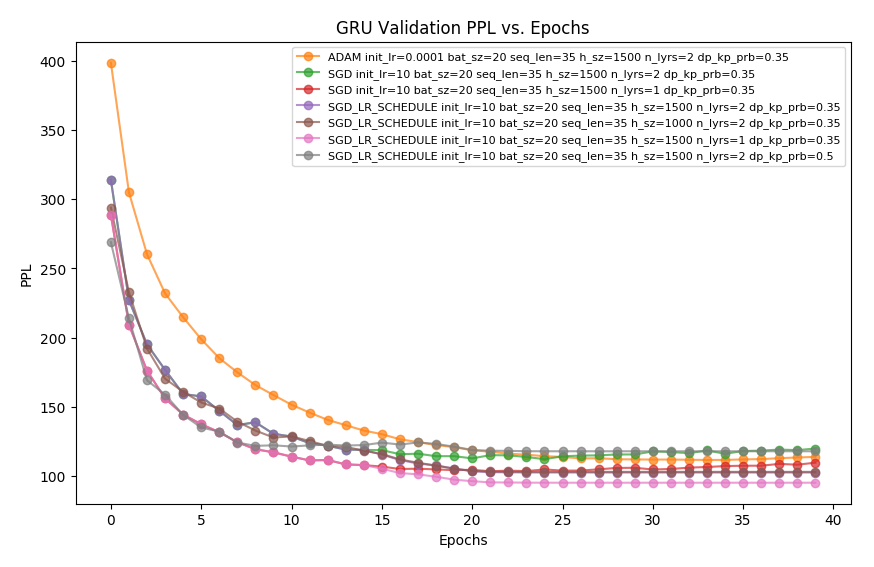


Figure – Validation perplexity vs. wall-clock time for experiments using GRU architecture

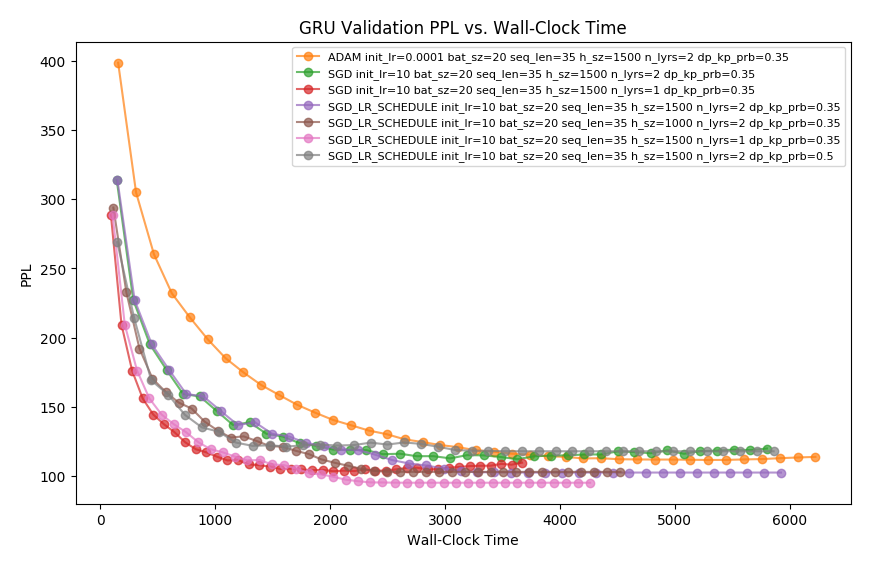


Figure – Validation perplexity vs. wall-clock time for experiments using GRU architecture

### Transformer Architecture Results

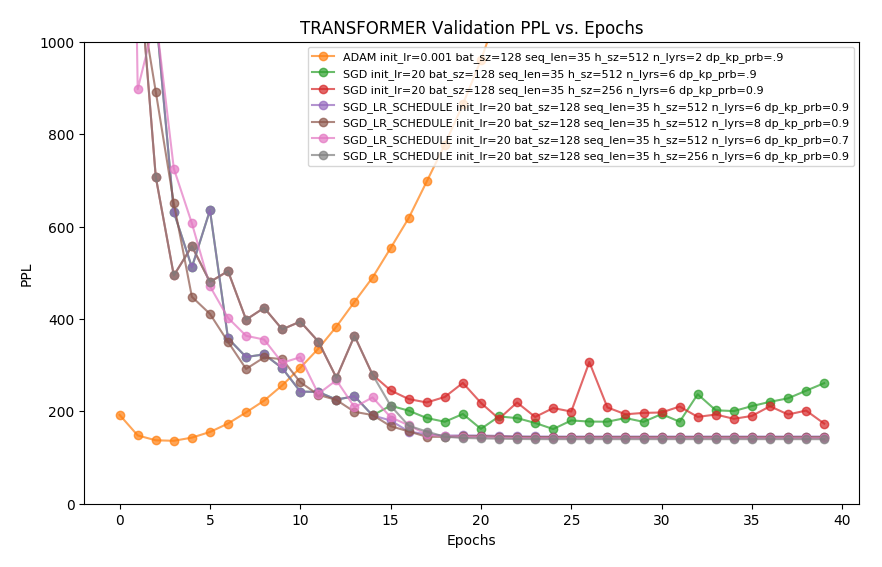


Figure – Validation perplexity vs. wall-clock time for experiments using transformer architecture

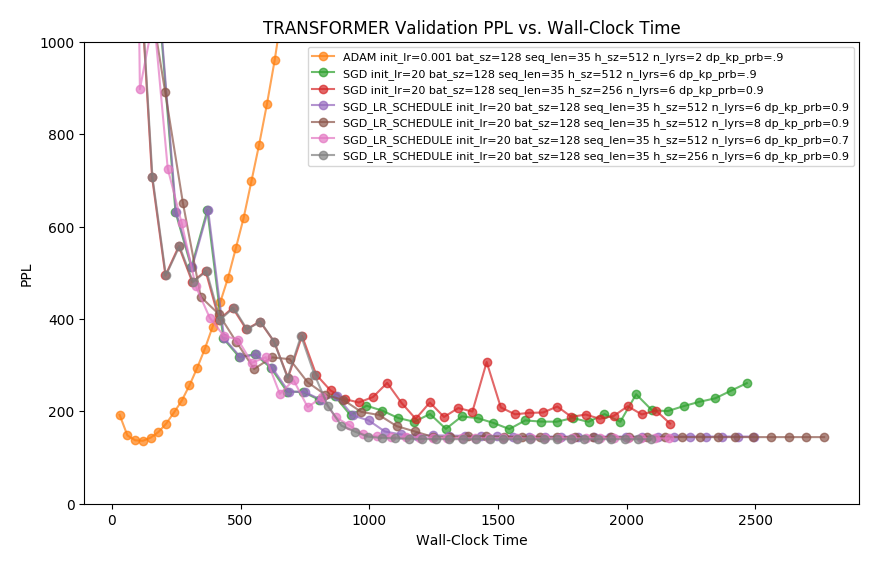


Figure – Validation perplexity vs. wall-clock time for experiments using transformer architecture

## 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.

They are also listed in Table 6 - Table 8 below.

Table – 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 – 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 8 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 – 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?**

In problem 4.1 basic settings for each of the architectures (RNN, GRU and transformer) are explored. In problem 4.2 additional optimizers are tested with each architecture. It could be expected that the newer transformer model might achieve the best performance however the GRU architecture appears to consistently yield the best validation perplexity (in the range of 95-115). Presumably this is because of its better usage of previous context in addition to its gating mechanism that alleviate the vanishing gradient problem and enable it to learn longer term dependencies. However, the GRU architecture was also the slowest to train. The slower timings of the RNN and GRU models are most likely due to the costly recurrent connections (1 connection per timestep) as well as the additional training parameters of the gates in the GRU model. The transformer also appears noisier in its learning curve for validation perplexity. The un-augmented RNN architecture gave the worst performance in both training and validation perplexities as well are training time as its lack of gating mechanisms makes it harder to learn of long-term context dependencies and it may suffer from vanishing gradient problems.

The RNN model in section 4.2 with SGD LR schedule also has a reduced hidden unit size of 512 from 1500 (Figure 19) which appears to oversimplify the model to the point where it exhibits some bias and performs poorly on both the training and validation sets (perplexities of 200 versus 156 baseline in section 4.1).

When running different optimizers in problem 4.2, several results were noted. The reasoning for results of different optimizers are discussed further in the next section. When SGD was applied to the RNN architecture with low, fixed learning rate (0.0001), the model failed to converge fast enough (terminating at 2000 after 40 epochs of training) as might be expected for too low a learning rate. Conversely, Adam with its adaptive gradient and momentum applied to the same RNN model has no such problem (Figure 5). The transformer architecture with Adam optimization and reduced number of layers was found to quickly overfit yet still resulted in a good early stopping validation perplexity.

In section 4.3 additional hyper-parameters were explored to find better performing models than the baselines ins section 4.1. The number of layers, hidden units and dropout were adjusted. Perhaps surprisingly, many parameter settings with less complex configurations (reduces layers or number of hidden units) resulted in better or similar performance. These less complicated architectures resulting in significantly faster training time (reduced wall-clock time) as well. This perhaps indicates the additional model complexity of extra layers of hidden units might not be necessary for the dataset or there is some aspect of overfitting.

**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.**

The SGD optimizer simples applies a constant learning rate () to each gradient descent update. In general, a larger learning rate can help the model more towards a minimum in the cost function faster however too large of a learning rate can result in oscillations around a minimum and potentially divergence. Yet too small of a learning rate can result in excessively slow training times. SGD with scheduled learning rate decreases the learning rate over time since typically when optimization starts, and weights are far from their optimal values to minimize the cost function, the gradient can be applied with higher magnitude and then gradually decreased as the minima is approached. However, this method still applies the same learning rate to all dimensions of the weight vector. Adam is an adaptive gradient method (like AdaGrad or RMSProp) which also applies momentum. In short, an adaptive gradient method like Adam will adapt the learning rate based on the steepness of the gradient in each weight direction. This causes the optimization process to move more steadily towards the goal instead of “zig-zagging” in cases where the cost function is uneven.

The consequences of too small a learning rate for SGD can be seen in section 4.2 when SGD is used on the RNN architecture with learning rate (Figure 13). A constant, small learning rate such as this fails to cause the model to converge to a reasonable perplexity after 40 epochs at it terminates at 2000 validation perplexity after training. Conversely, Adam with its adaptive gradient and momentum applied to the same RNN model has no such problem (Figure 5). The effects of the Adam optimizer are particularly noticeable for the transformer in section 4.2 (Figure 23) which quickly achieves good validation performance and training performance (before overfitting). This model also has a reduced layer size (2 instead of 6) and so it could be theorized that since there are a reduced number of weights, the cost function is simpler and optimization through Adam can occur more quickly. SGD with scheduling performs reasonably well for the GRU and transformer models (Table 3). The RNN model with scheduled SGD has poor results (Figure 19, Table 3) however this may be due to its reduced hidden size and weights making the model unable to adequately fit the data.

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

The transformer appears to consistently run faster than the other architectures. This is most likely due to the costly recurrent connections (1 connection per timestep) in the RNN and GRU architectures as well as the additional training parameters of the gates in the GRU model. Of course, parameters which give a small model (such as less layers and smaller hidden size) also significantly speed up training. An adaptive learning rate optimizer such as Adam can also help reach the minima faster and thus save time with early stopping.

For generalization performance, the GRU architecture appears to consistently give the best performance even outperforming the newer transformer model. Presumably this is because of its better usage of previous context.

It does initially appear that the model that obtains the best validation performance (GRU) and best wall-clock time (transformer and smaller model parameters) is a time vs performance trade-off however this may not always be the case. For example, sometimes extra layers or hidden units may be unnecessary to achieve reasonably high performance. This was noted and discussed in section 4.3.1 when looking at additional hyper-parameters. It was found that a decrease in layers gave better validation perplexity results and a reduction in hidden unit size resulting in little performance hit while giving substantially faster wall-clock time for RNN and GRU models. Likewise, a suitable optimizer could reduce wall-clock time by early stopping as well as achieve a better perplexity result.

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

The GRU architecture appears to be the most reliable and seems to consistently achieve validation perplexity in the ranges of 95-115 with the parameters tested. Although it is also consistently the slowest in training.

From the results obtained in sections 4.1 and 4.2 it initially appears as though the RNN has the most variance in validation perplexity results across parameters. However, it can be argued that this is because its parameters (especially for 4.2) inherently give poor results from its base model in section 4.1. The RNN model with SGD optimizer from section 4.2 (Figure 13) has a learning rate that is too small (0.0001) and so it fails to obtain comparable perplexity to the other models, terminating at 2209.92 validation perplexity after 40 epochs. Likewise, the RNN model with SGD\_LR\_SCHEDULE optimizer in section 4.2 (Figure 19) also has its hidden unit size reduced from 1500 to 512 which appears too much of a reduction for the model to reasonably fit either the training or validation set (high bias) and so it performs poorly as well.

However, from the additional parameters tested and learning curves of the architectures the RNN appears more consistent than the transformer. There are several instances where the learning curves of the transformer appear noisy in their validation perplexities, exhibiting noticeable fluctuations versus the other models (for example Figure 43 and Figure 47). The transformer also appears to overfit more noticeably in some cases for architectures in section 4.2 (Figure 17 and Figure 23).

**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 9 – 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.

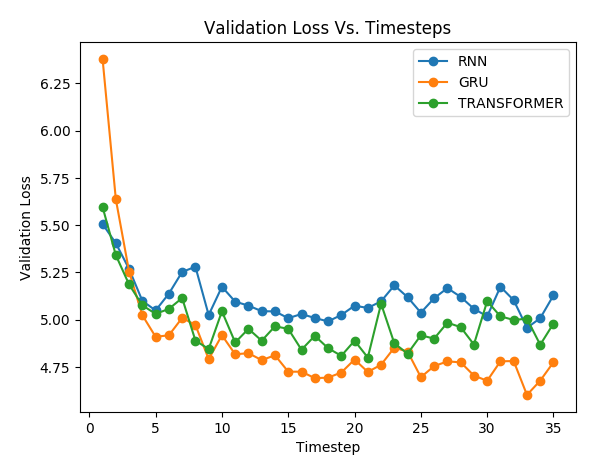


Figure 63 – Validation loss over time-steps for each architecture of Problem 4.1

**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.

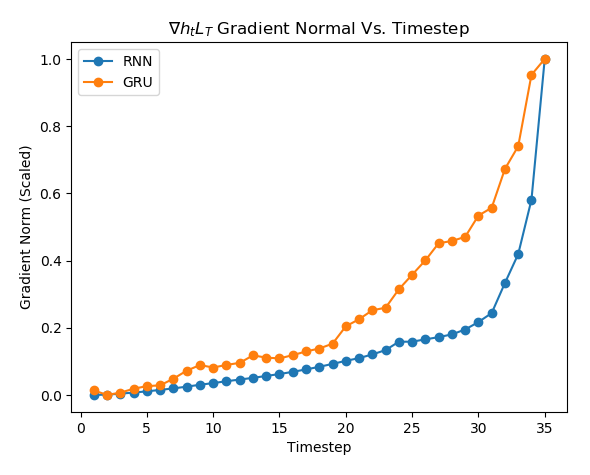


Figure 64 - normal over timesteps

**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.

# Appendix – All Generated Samples (5.3)

# References

|  |  |
| --- | --- |
| [1] | J. Sun. “Recurrent Neural Networks”. 2018. Available: <http://sunlab.org/teaching/cse6250/fall2018/dl/dl-rnn.html#recurrent-neural-networks-2> [Accessed: 2019-03-24] |