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: https://github.com/stefanwapnick/IFT6135PracticalAssignments

Contents

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

1 - Implementing a Simple RNN

1.1 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:

$$P(\boldsymbol{y_t}|\boldsymbol{x_1} \dots \boldsymbol{x_t}) = \sigma_y(\boldsymbol{W_y h_t} + \boldsymbol{b_y}) \quad \text{Eq. 1}$$

$$\boldsymbol{h_t} = tanh(\boldsymbol{W_x x_t} + \boldsymbol{W_h h_{t-1}} + \boldsymbol{b_h}) \quad \text{Eq. 2}$$

Where $\boldsymbol{h_t}$ is the hidden cell state at time $\boldsymbol{t}$ of the network, $\boldsymbol{x_t}$ is the $\boldsymbol{t}$'th input token (typically a word embedding) and $\boldsymbol{y_t}$ the predicted next token in the sequence given the context $\boldsymbol{x_1} \dots \boldsymbol{x_t}$.

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.

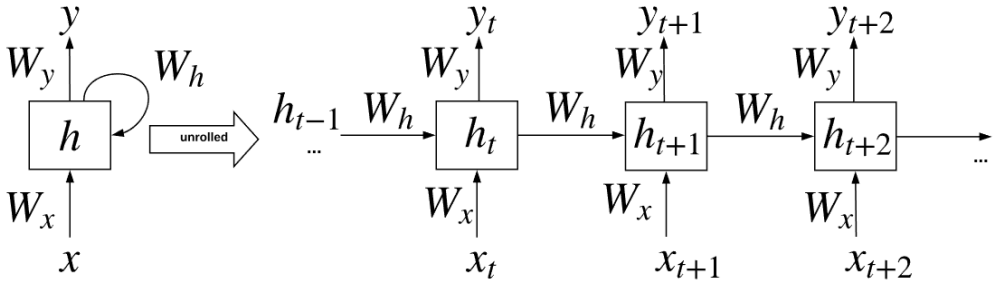

Figure 1 – Illustration of the connections in a Recurrent Neural Network

1.2 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:

$$\boldsymbol{h_t} = tanh([\boldsymbol{W_x}|\boldsymbol{W_h}][\boldsymbol{x_t}|\boldsymbol{h_{t-1}}]^{\mathrm{T}} + \boldsymbol{b_h})$$

- 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`): $[-k, k]$ where $k = 1/\sqrt{hidden\_size}$. The output and embedding layers were initialized uniformly in the range $[-0.1, 0.1]$.

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)
```

2 Implement RNN with Gated Recurrent Units (GRU)

2.1 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:

$$\boldsymbol{r_t} = \sigma_r(\boldsymbol{W_t x_t} + \boldsymbol{U_r h_{t-1}} + \boldsymbol{b_r}) \quad \text{Eq. 3}$$

$$\boldsymbol{z_t} = \sigma_r(\boldsymbol{W_t x_t} + \boldsymbol{U_r h_{t-1}} + \boldsymbol{b_r}) \quad \text{Eq. 4}$$

$$\widetilde{\boldsymbol{h_t}} = \sigma_r(\boldsymbol{W_t x_t} + \boldsymbol{U_r h_{t-1}} + \boldsymbol{b_r}) \quad \text{Eq. 5}$$

$$\boldsymbol{h_t} = (1 - \boldsymbol{z_t}) \odot \boldsymbol{h_{t-1}} + \boldsymbol{z_t} \odot \widetilde{\boldsymbol{h}}_{\boldsymbol{t}} \quad \text{Eq. 6}$$

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

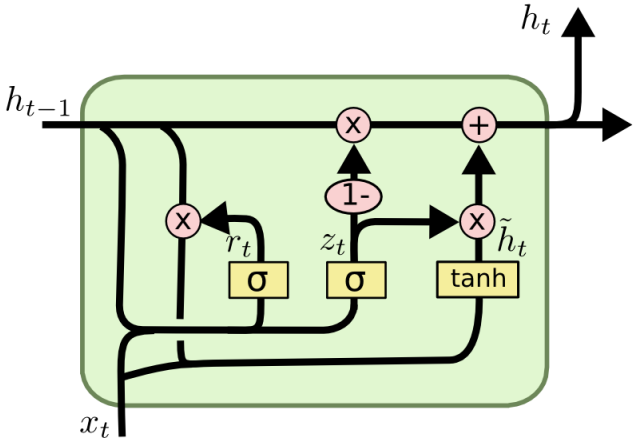

Figure 2 – Illustration of GRU cell containing reset and forget gates [1]

2.2 Source Code

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

- Inputs and weight matrices for W and U are once again concatenated for efficiency as in section 1.
- A `GRULayer` component class is implemented that are composed inside the GRU class.
- 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)
```

3 Attention Module of Transformer Network

3.1 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:

$$\boldsymbol{A_i} = softmax\left(\frac{(\boldsymbol{QW_{Q_i}} + \boldsymbol{b_{Q_i}})(\boldsymbol{KW_{K_i}} + \boldsymbol{b_{K_i}})^T}{\sqrt{d_k}}\right) \quad \text{Eq. 7}$$

$$\boldsymbol{H_i} = \boldsymbol{A_i}(\boldsymbol{VW_{V_i}} + \boldsymbol{b_{V_i}}) \quad \text{Eq. 8}$$

$$A(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = concat(\boldsymbol{H_1}, \ldots \boldsymbol{H_m})\boldsymbol{W_O} + \boldsymbol{b_O} \quad \text{Eq. 9}$$

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 $\boldsymbol{A_i}$, this mask $\boldsymbol{s}$ is applied to the intermediate $\boldsymbol{x}$ value:

$$\tilde{\boldsymbol{x}} = \boldsymbol{x} \odot \boldsymbol{s} - 10^9(1 - \boldsymbol{s}) \quad \text{Eq. 10}$$

3.2 Source Code

To implement the attention calculation for $\boldsymbol{A_i}$ a separate `SingleHeadAttention` class was made. The `MultiHeadedAttention` composes these individual attention head classes and computes the final output on concatenated $\boldsymbol{H_i}$ 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)
```

4 Training Language Models

Using the implemented architectures of sections 1-3, experiments were executed 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
wct	Wall-clock time

4.1 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.

4.1.1 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. 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 by timestep (essentially acting as another dimension of layers) which can quickly accumulate for longer sequence lengths. Likewise, additional training parameters of the gates in the GRU model require additional computation. The transformer also appears noisier in its learning curve, at least initially.

Table 2 – 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

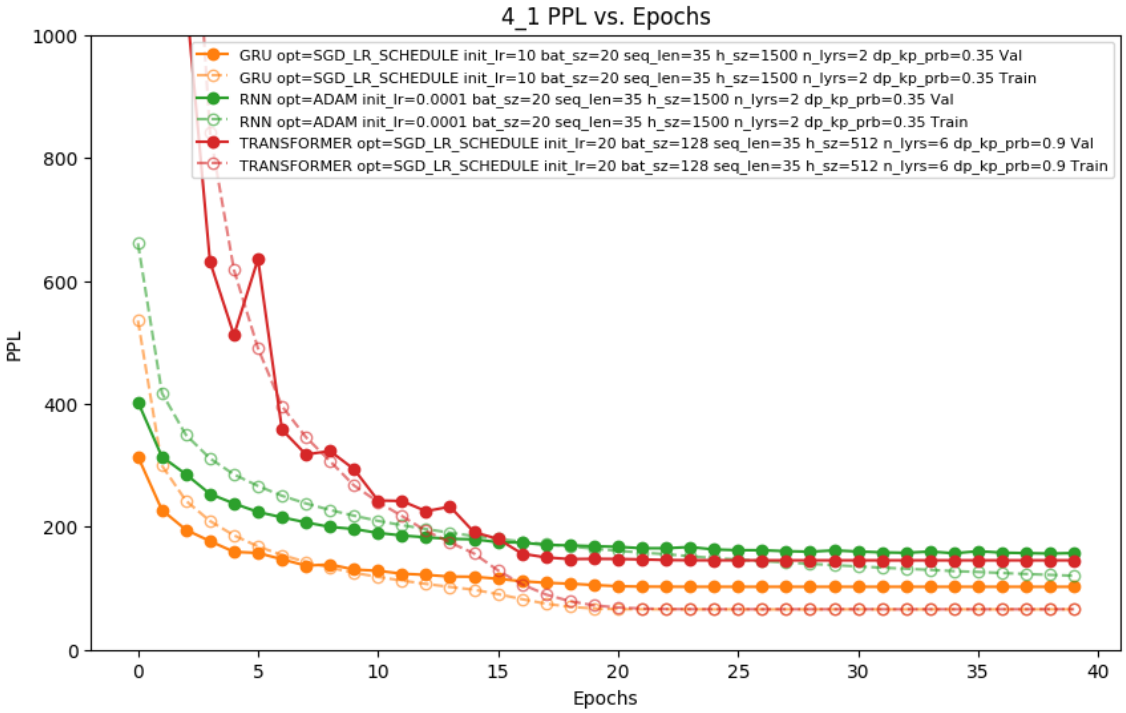

Figure 3 – Training and validation perplexity vs epochs for experiments in problem 4.1

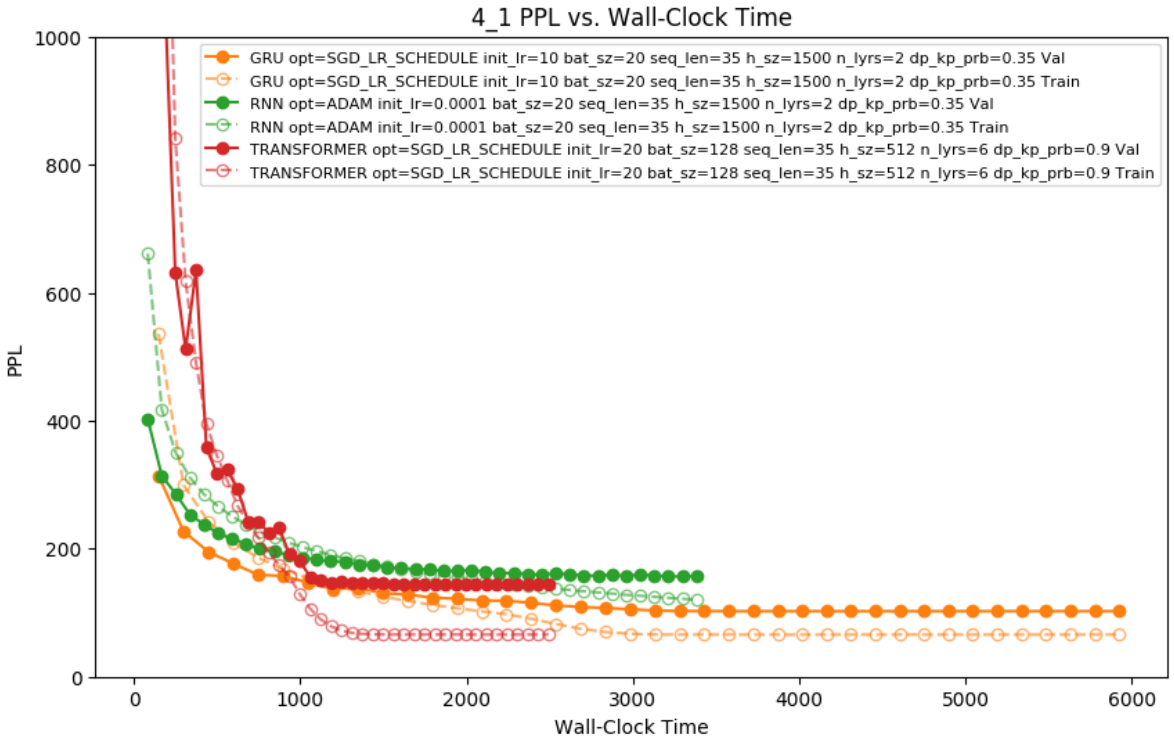

Figure 4 - Training and validation **p**erplexity vs wall-clock time for experiments in problem 4.1

4.1.2 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.

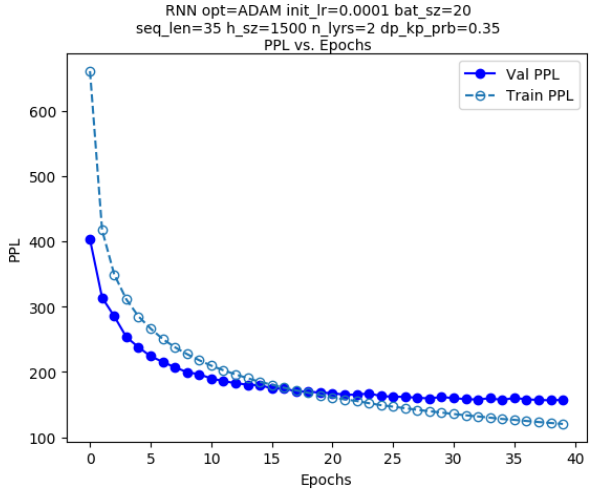

Figure 5 – PPL vs. epochs RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

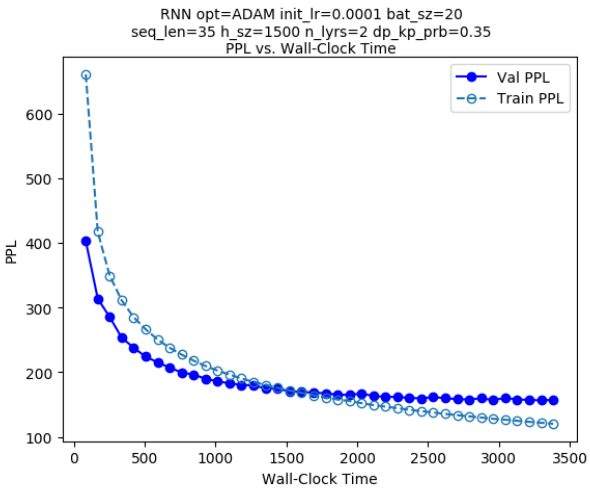

Figure 6 - PPL vs. wall-clock time RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

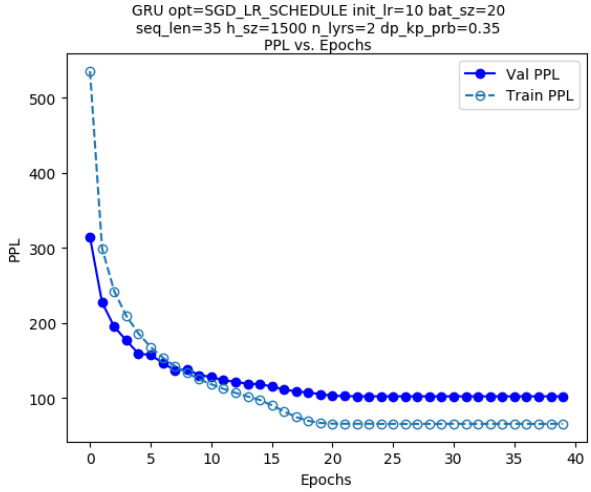

Figure 7 - PPL vs. epochs GRU model, SGD_LR_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

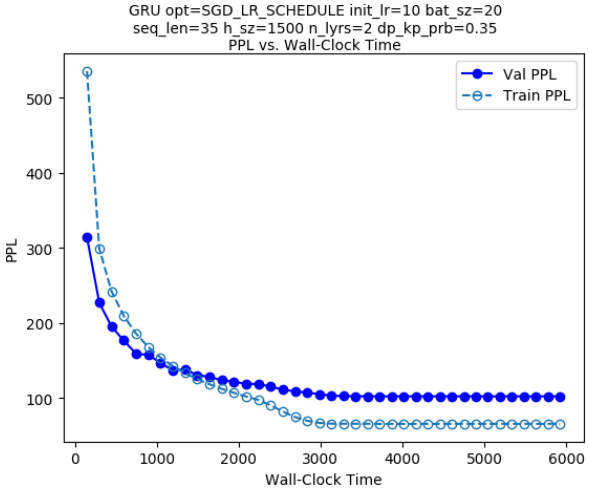

Figure 8 - PPL vs. wall-clock time GRU model, SGD_LR_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

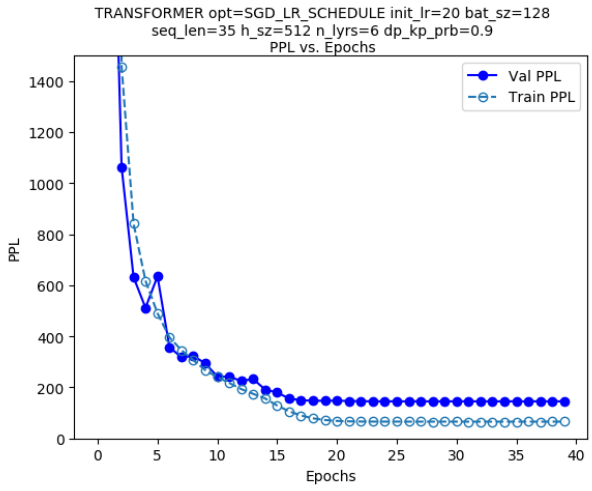

Figure 9 - PPL vs. epochs Transformer model, SGD_LR_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.9

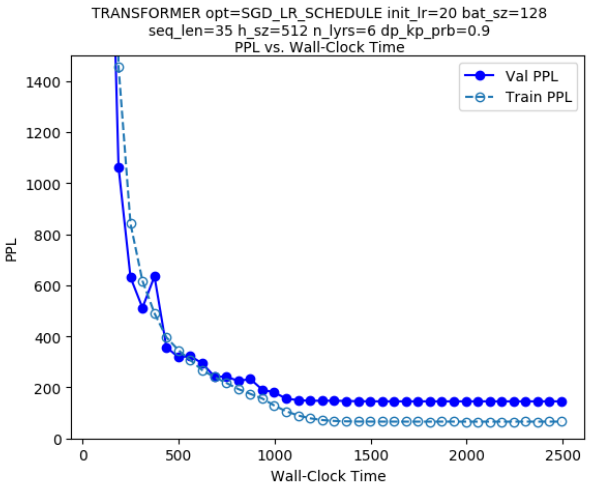

Figure 10 - PPL vs. wall-clock time Transformer model, SGD_LR_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.9

4.2 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) were tested for each architecture in this section.

4.2.1 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). Conversely, adaptive optimizer such as Adam with the RNN architecture initialized with a similar learning rate does not have this issue.

The RNN model with SGD LR schedule also has its hidden size reduced to 512 (from 1500) which appears too small and so the model exhibits some 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 3 – 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

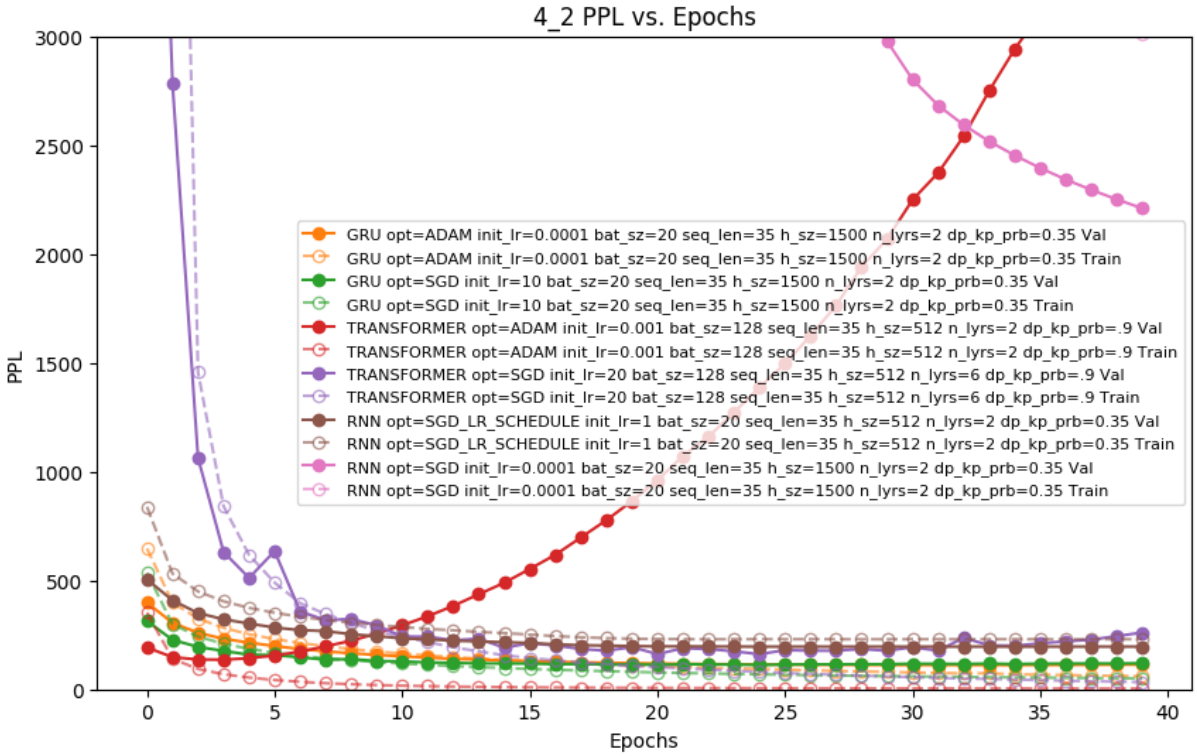

Figure 11 – Training and validation perplexity vs. epochs for architectures in problem 4.2

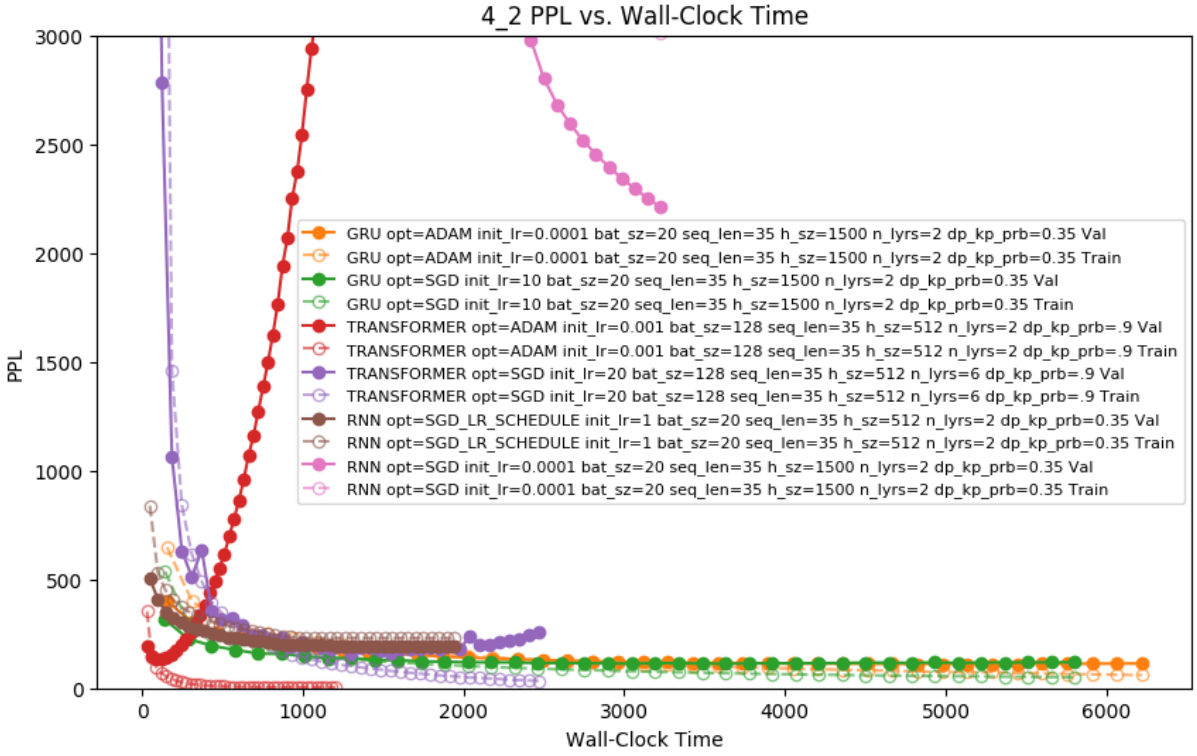

Figure 12 – Training and validation perplexity vs. wall-clock time for architectures in problem 4.2

4.2.2 Exploration of Optimizers Learning Curves

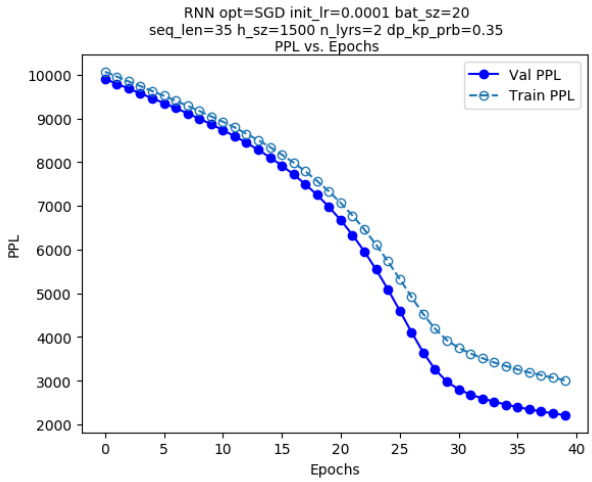

Figure 13 - PPL vs. epochs RNN model, SGD optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

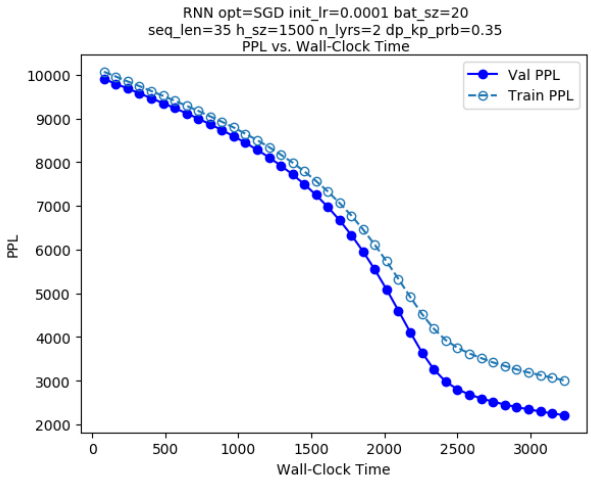

Figure 14 - PPL vs. wall-clock time RNN model, SGD optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

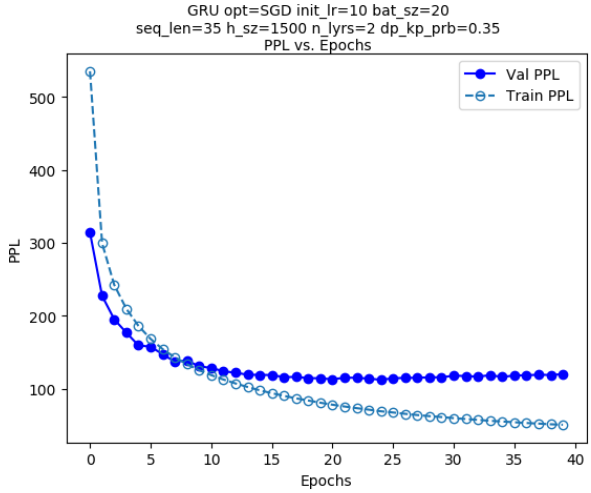

Figure 15 - PPL vs. epochs GRU model, SGD optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

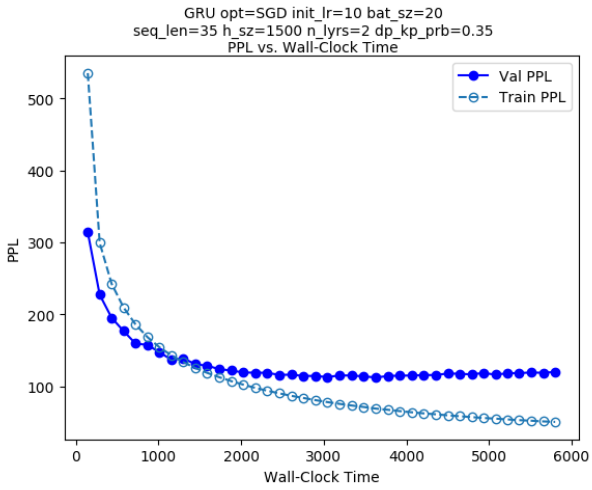

Figure 16 - PPL vs. wall-clock time GRU model, SGD optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

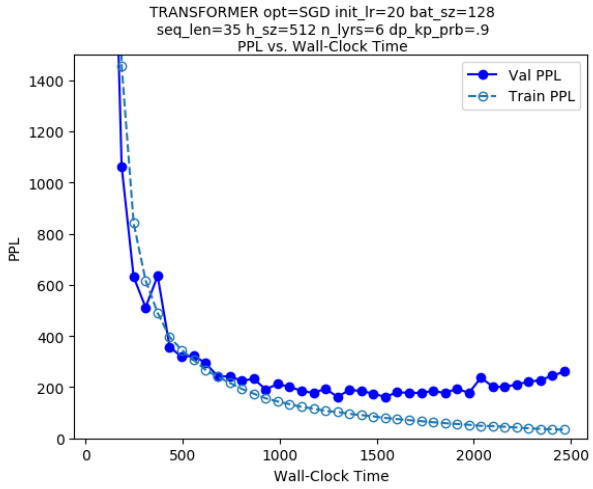

Figure 17 - PPL vs. epochs Transformer model, SGD optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.9

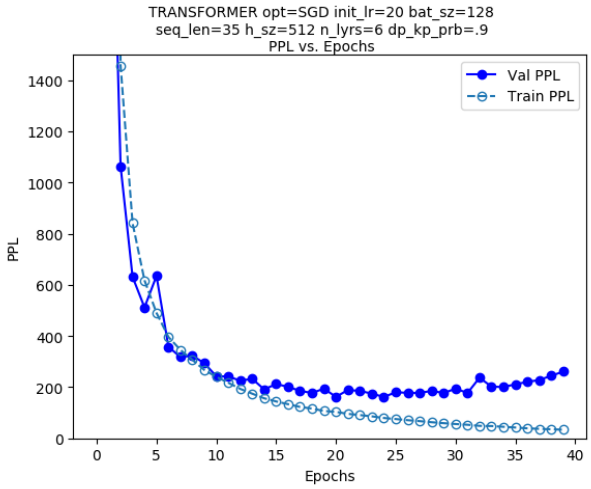

Figure 18 - PPL vs. wall-clock time Transformer model, SGD optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, dropout keep prob.=0.9

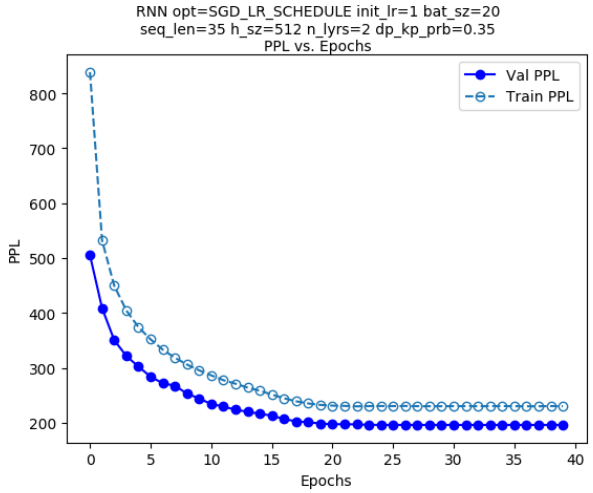

Figure 19 - PPL vs. epochs RNN model, SGD_LR_Schedule optimizer, init. learning rate=1, batch size=20, sequence length=35, hidden size=512, layers=2, dropout keep prob.=0.35

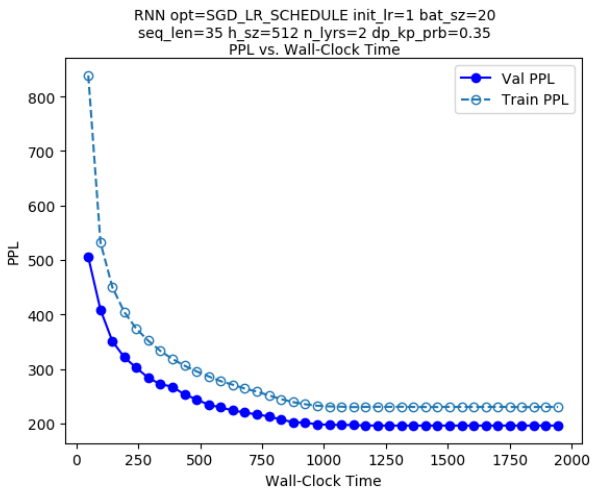

Figure 20 - PPL vs. wall-clock time RNN model, SGD_LR_Schedule optimizer, init. learning rate=1, batch size=20, sequence length=35, hidden size=512, layers=2, dropout keep prob.=0.35

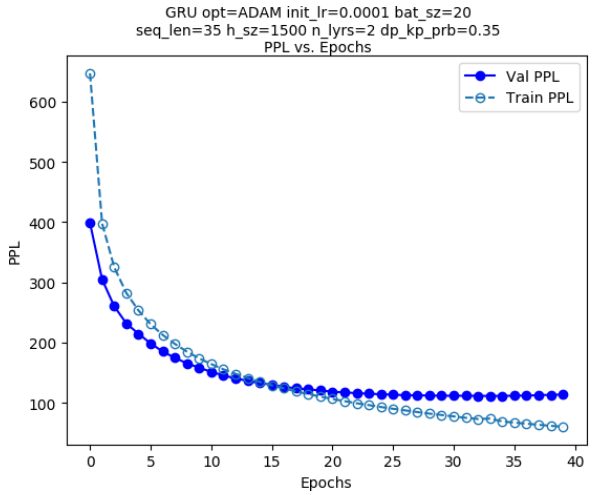

Figure 21 - PPL vs. epochs GRU model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

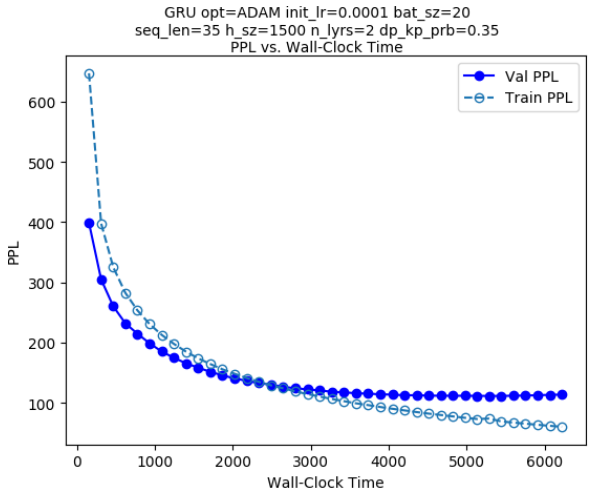

Figure 22 - PPL vs. wall-clock time GRU model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

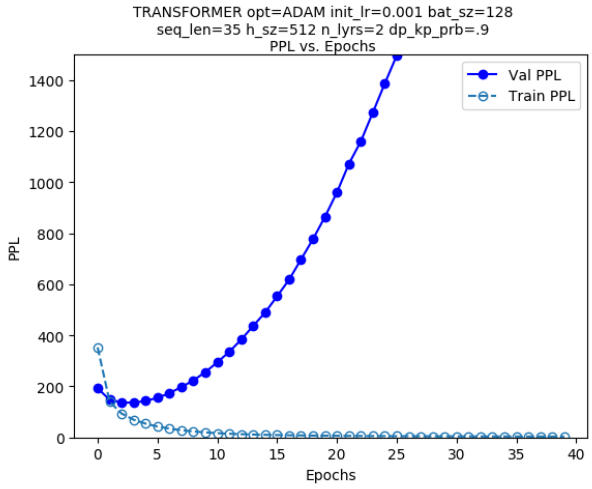

Figure 23 - PPL vs. epochs Transformer model, ADAM optimizer, init. learning rate=0.001, batch size=20, sequence length=35, hidden size=512, layers=2, dropout keep prob.=0.35

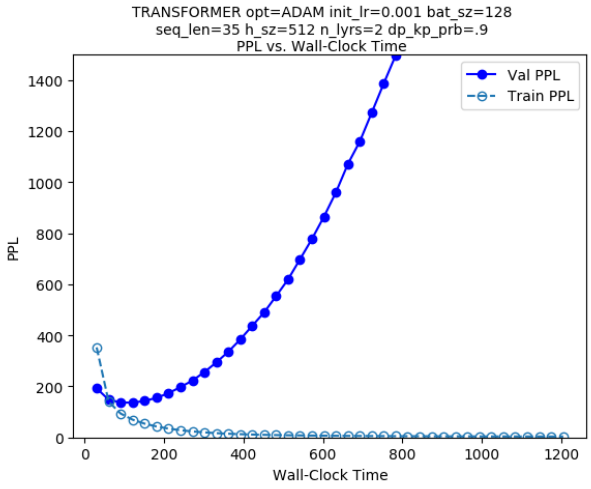

Figure 24 - PPL vs. wall-clock time Transformer model, ADAM optimizer, init. learning rate=0.001, batch size=20, sequence length=35, hidden size=512, layers=2, dropout keep prob.=0.35

4.3 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 **bold 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 similar performance. This perhaps indicates that in some sense the previous 4.1 models were overfitting by the additional complexity in their structure or the added complexity was not needed to generalize well enough to the validation set. Likewise, a faster wall-clock time was achieved with these simplified models.

4.3.1 Summary

Table 4 – Results of experiments in Hyper-Parameter Search. Valid. ppl. is best obtained. Train ppl. is value at best valid. ppl. The Δ 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	Δ 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

4.3.2 Hyper-Parameter Search Learning Curves

4.3.2.1 RNN Architectures

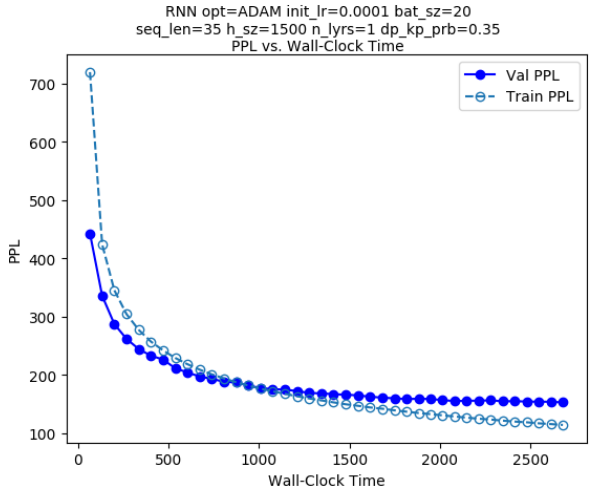

Figure 25 - PPL vs. epochs RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, **layers=1**, dropout keep prob.=0.35

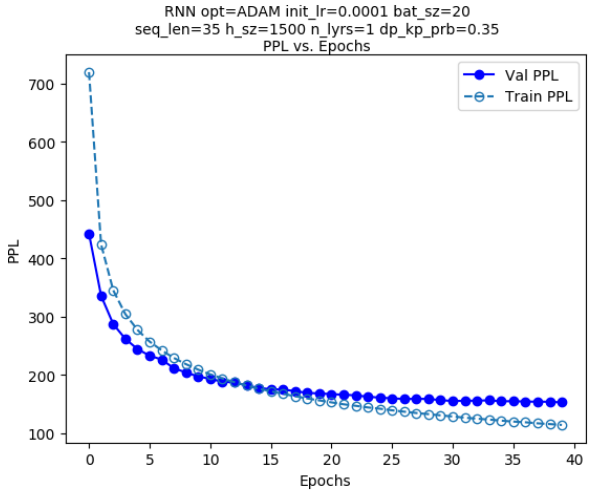

Figure 26 - PPL vs. wall-clock time RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, **layers=1**, dropout keep prob.=0.35

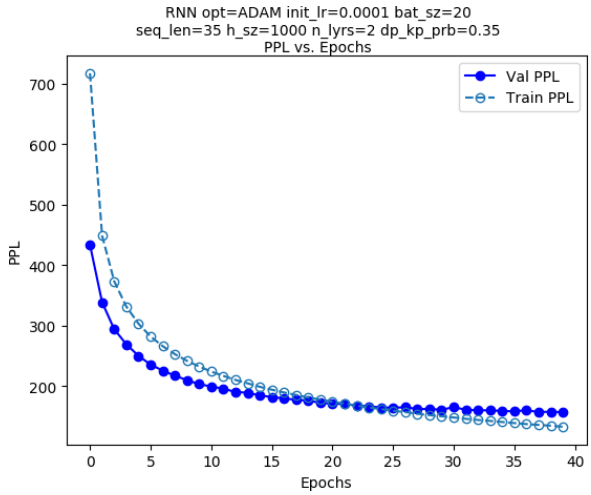

Figure 27 - PPL vs. epochs RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, **hidden size=1000**, layers=2, dropout keep prob.=0.35

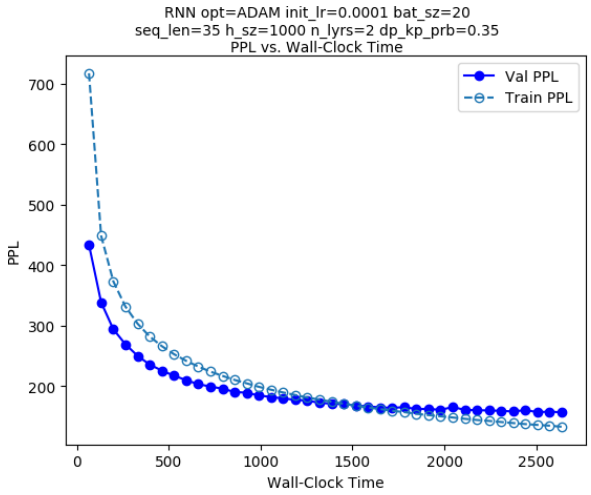

Figure 28 - PPL vs. wall-clock time RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, **hidden size=1000**, layers=2, dropout keep prob.=0.35

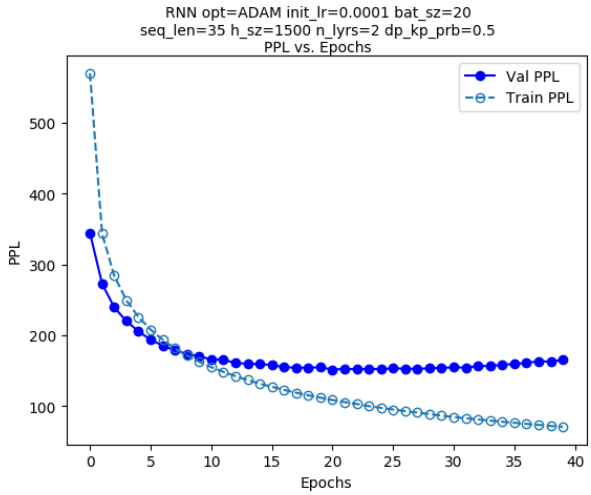

Figure 29 - PPL vs. epochs RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=1, **dropout keep prob.=0.5**

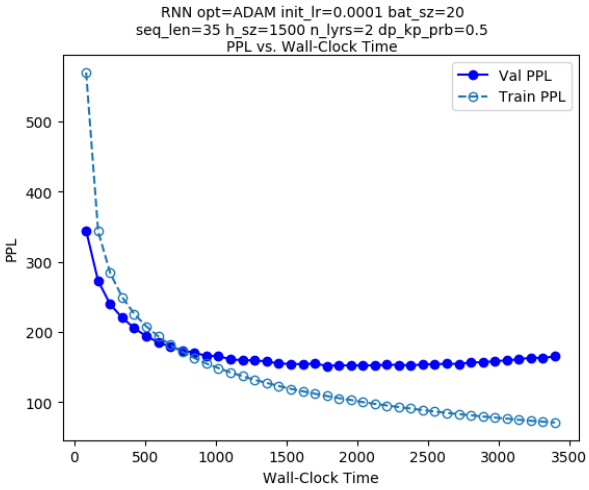

Figure 30 - PPL vs. wall-clock time RNN model, ADAM optimizer, init. learning rate=0.0001, batch size=20, sequence length=35, hidden size=1500, layers=1, **dropout keep prob.=0.5**

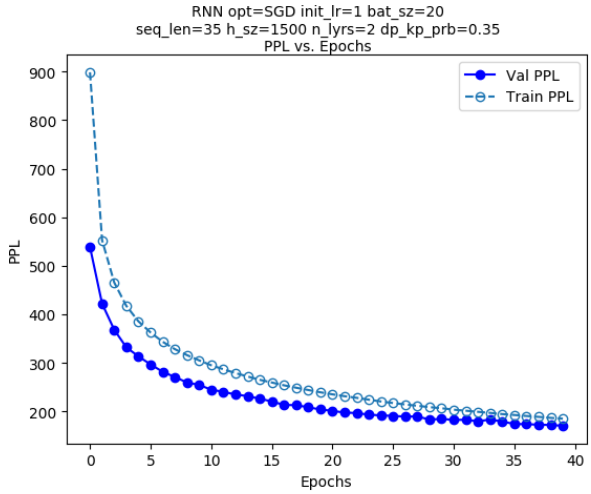

Figure 31 - PPL vs. epochs RNN model, **SGD optimizer**, init. learning rate=1, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

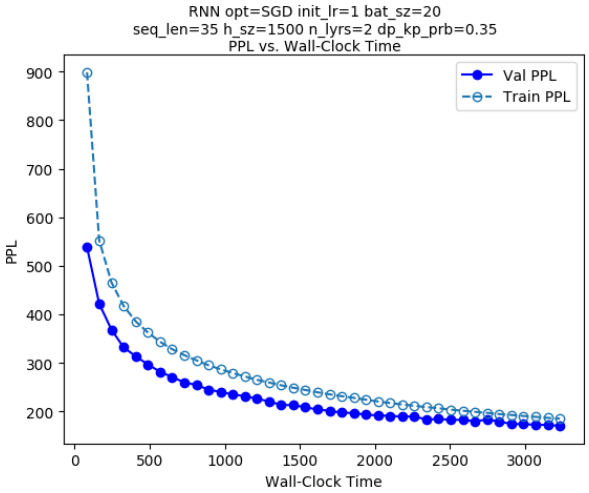

Figure 32 - PPL vs. wall-clock time RNN model**, SGD optimizer**, init. learning rate=1, batch size=20, sequence length=35, hidden size=1500, layers=2, dropout keep prob.=0.35

4.3.2.2 GRU Architectures

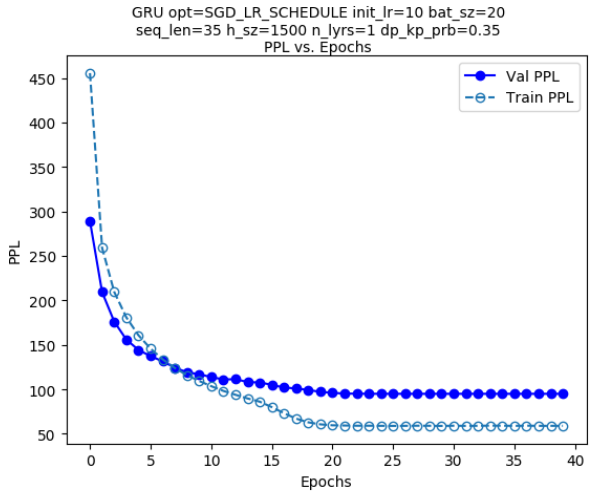

Figure 33 - PPL vs. epochs GRU model, SGD_LR_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, **layers=1**, dropout keep prob.=0.35

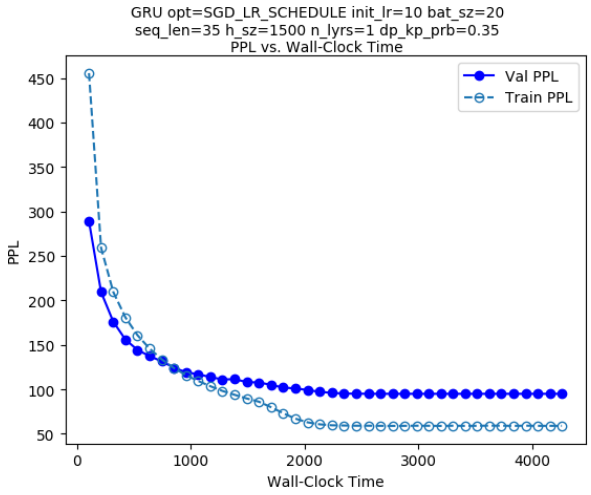

Figure 34 - PPL vs. wall-clock time GRU model, SGD_LR_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, **layers=1**, dropout keep prob.=0.35

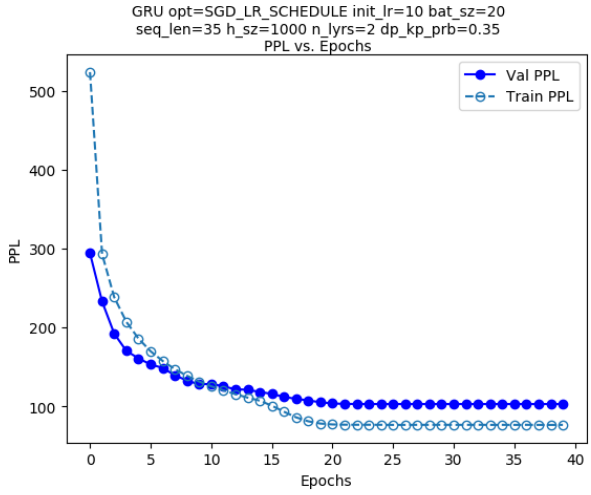

Figure 35 - PPL vs. epochs GRU model, SGD_LR_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, **hidden size=1000**, layers=1, dropout keep prob.=0. 35

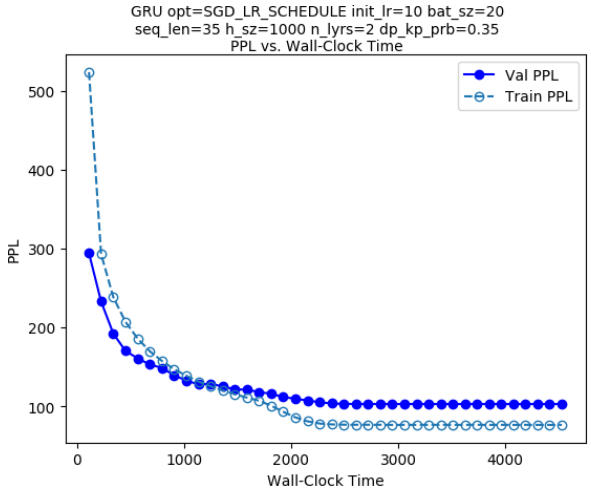

Figure 36 - PPL vs. wall-clock time GRU model, SGD_LR_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, **hidden size=1000**, layers=1, dropout keep prob.=0.35

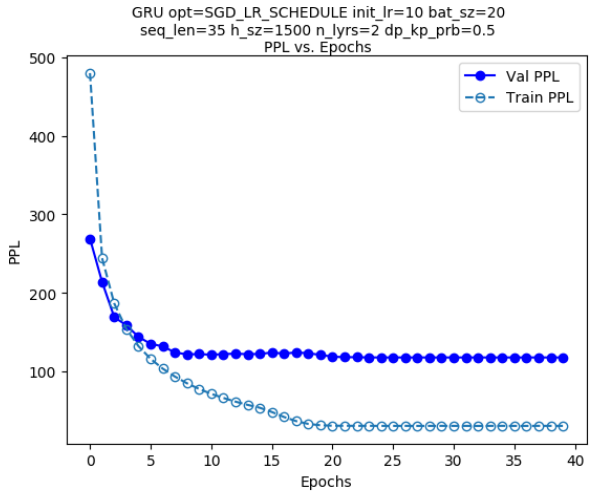

Figure 37 - PPL vs. epochs GRU model, SGD_LR_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, **dropout keep prob.=0.5**

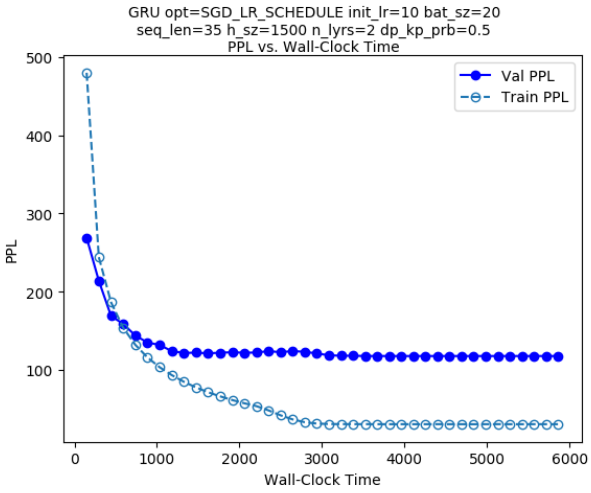

Figure 38 - PPL vs. wall-clock time GRU model, SGD_LR_Schedule optimizer, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, layers=2, **dropout keep prob.=0.5**

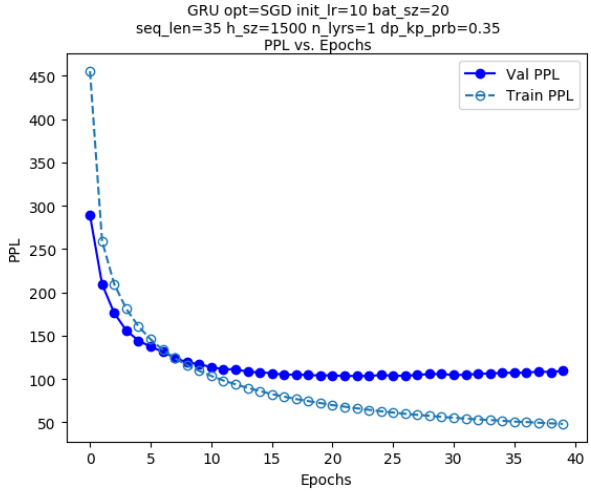

Figure 39 - PPL vs. epochs GRU model, **SGD optimizer**, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, **layers=1**, dropout keep prob.=0.35

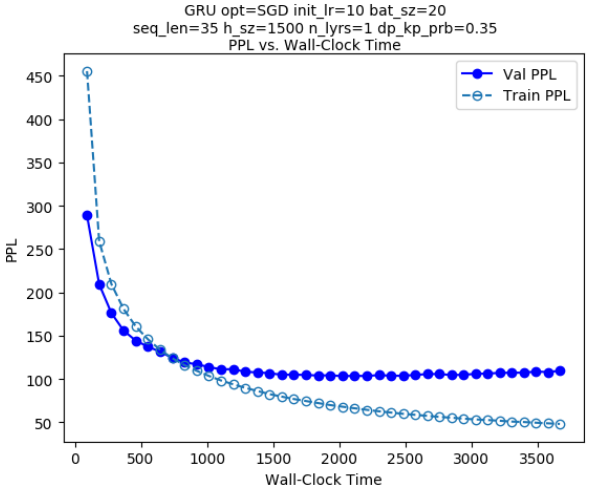

Figure 40 - PPL vs. wall-clock time GRU model, **SGD optimizer**, init. learning rate=10, batch size=20, sequence length=35, hidden size=1500, **layers=1**, dropout keep prob.=0.35

4.3.2.3 *Transformer Architectures*

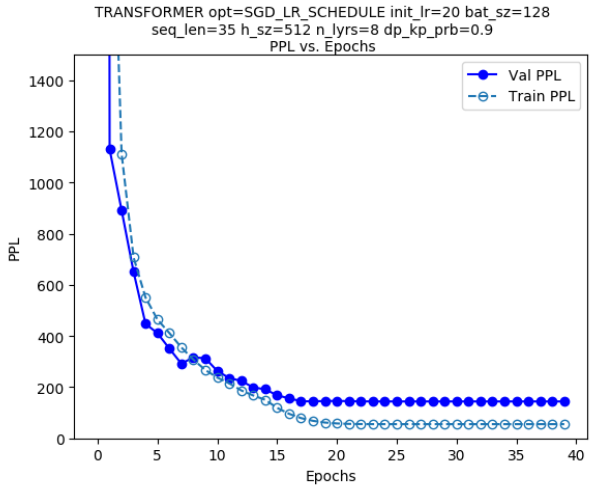

Figure 41 - PPL vs. epochs Transformer model, SGD_LR_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=256, **layers=8**, dropout keep prob.=0.9

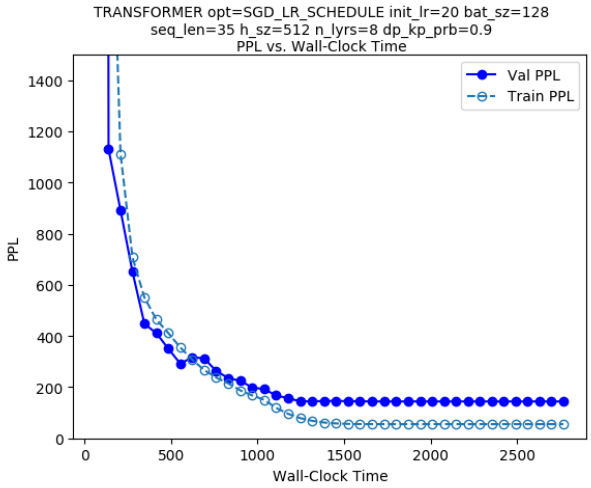

Figure 42 - PPL vs. wall-clock time Transformer model, SGD_LR_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=256, **layers=8**, dropout keep prob.=0.9

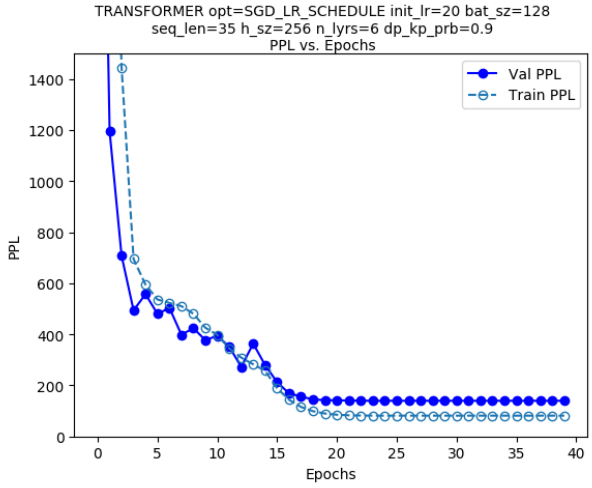

Figure 43 - PPL vs. epochs Transformer model, SGD_LR_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, **hidden size=256**, layers=6, dropout keep prob.=0.9

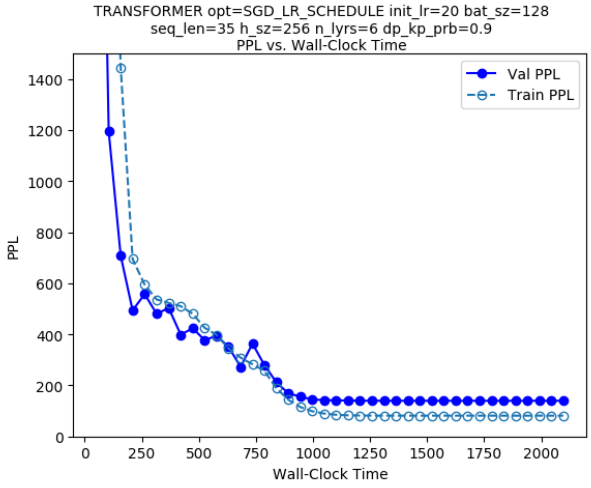

Figure 44 - PPL vs. wall-clock time Transformer model, SGD_LR_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, **hidden size=256**, layers=6, dropout keep prob.=0.9

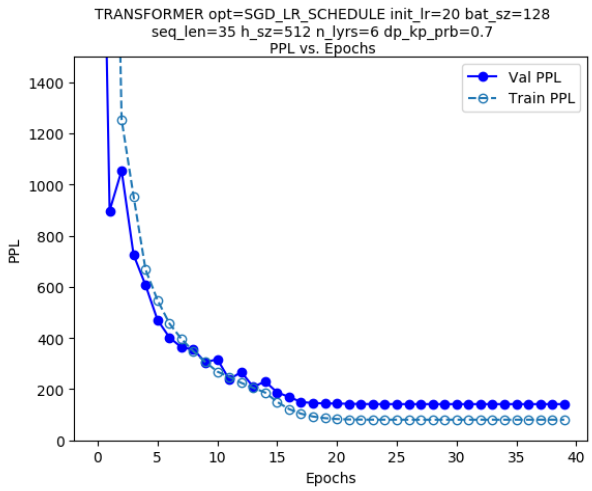

Figure 45 - PPL vs. epochs Transformer model, SGD_LR_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, **dropout keep prob.=0.7**

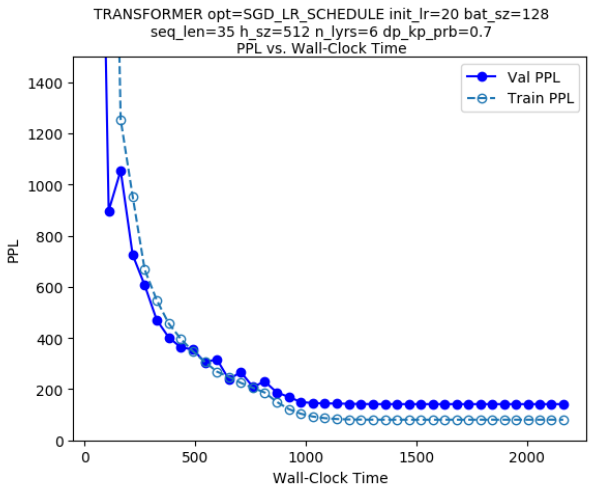

Figure 46 - PPL vs. wall-clock time Transformer model, SGD_LR_Schedule optimizer, init. learning rate=20, batch size=128, sequence length=35, hidden size=512, layers=6, **dropout keep prob.=0.7**

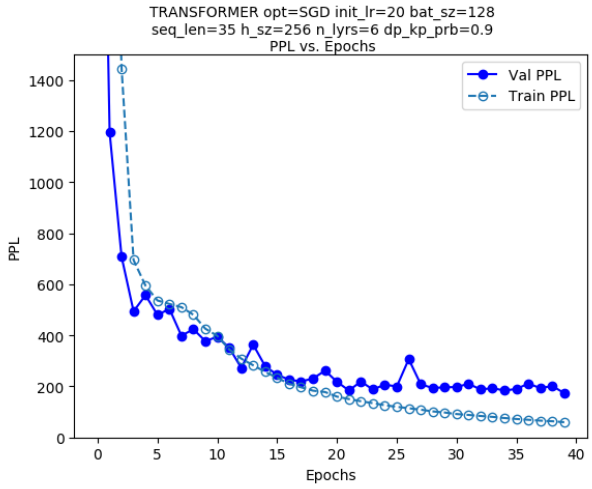

Figure 47 - PPL vs. epochs Transformer model, **SGD optimizer**, init. learning rate=20, batch size=128, sequence length=35, **hidden size=256**, layers=6, dropout keep prob.=0.9

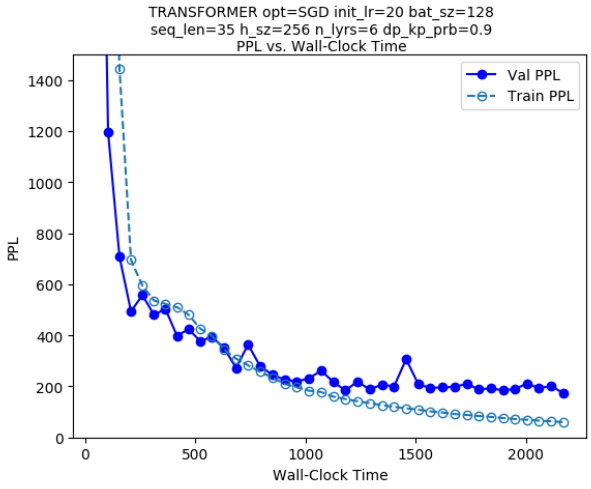

Figure 48 - PPL vs. wall-clock time Transformer model, **SGD optimizer**, init. learning rate=20, batch size=128, sequence length=35, **hidden size=256**, layers=6, dropout keep prob.=0.9

4.4 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 5 – 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	1000	2	0.35	113.17	76.51	102.88
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.

4.5 Results by Optimizer

This section graphs results organized by each optimizer: SGD_LR_Schedule, SGD and Adam.

4.5.1 SGD_LR_Schedule Optimizer Results

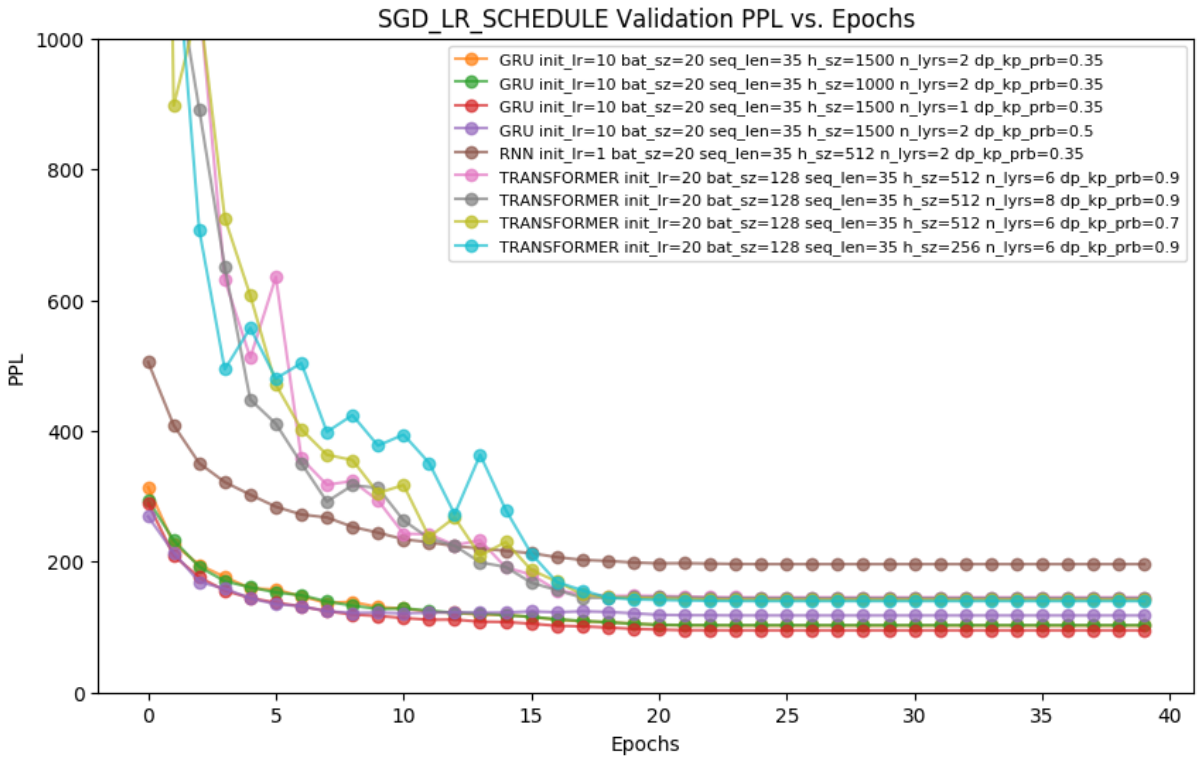

Figure 49 – Validation perplexity vs. epochs for experiments using SGD_LR_Schedule optimizer

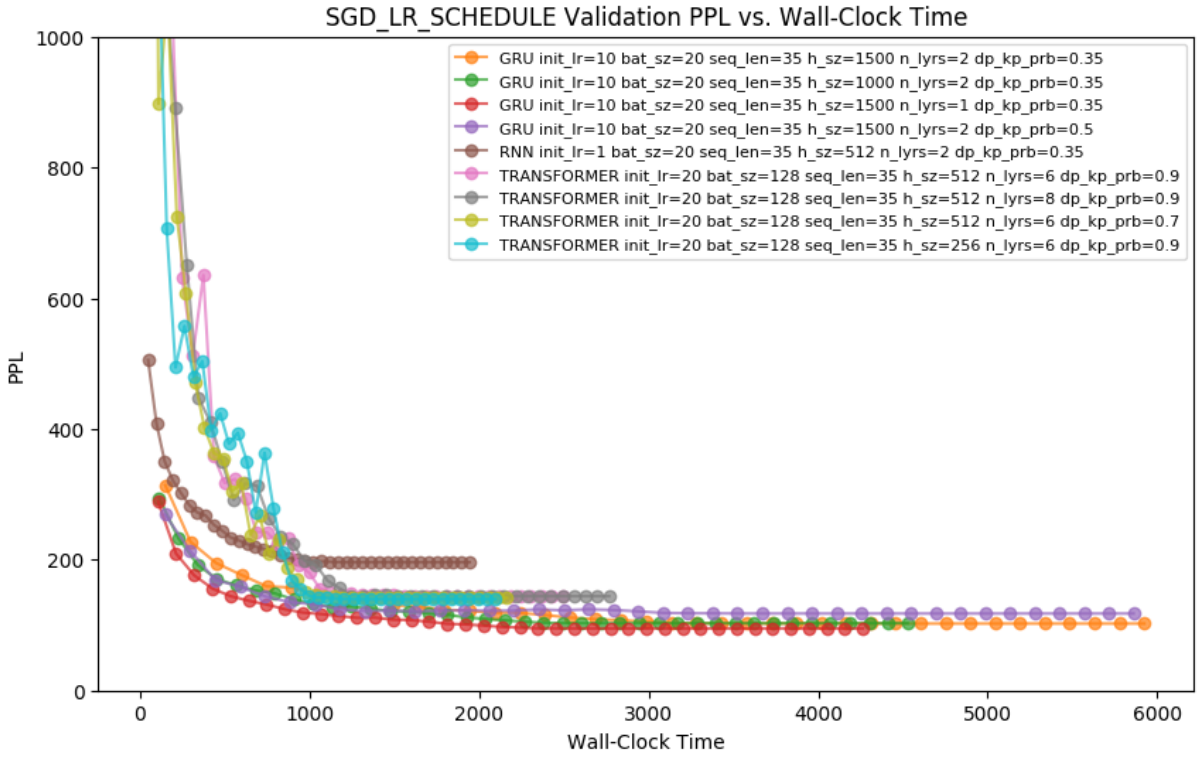

Figure 50 – Validation perplexity vs. wall-clock time for experiments using SGD_LR_Schedule optimizer

4.5.2 SGD Optimizer Results

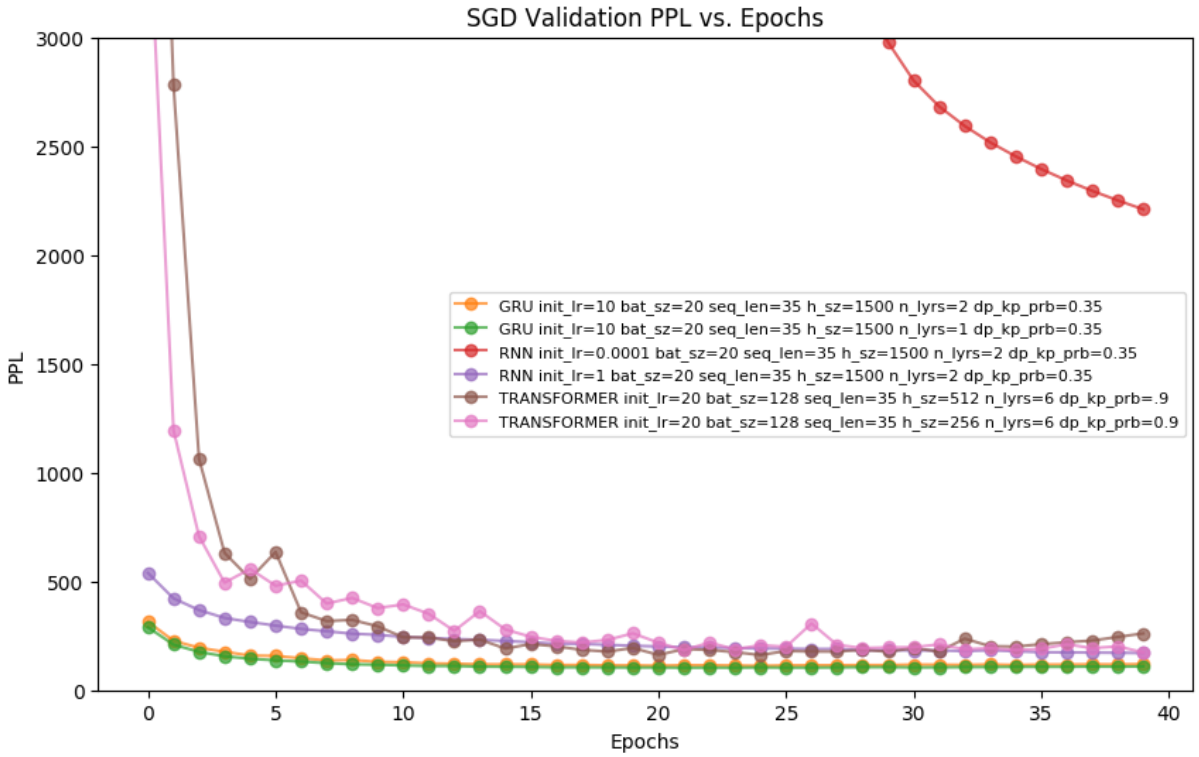

Figure 51 – Validation perplexity vs. epochs for experiments using SGD optimizer

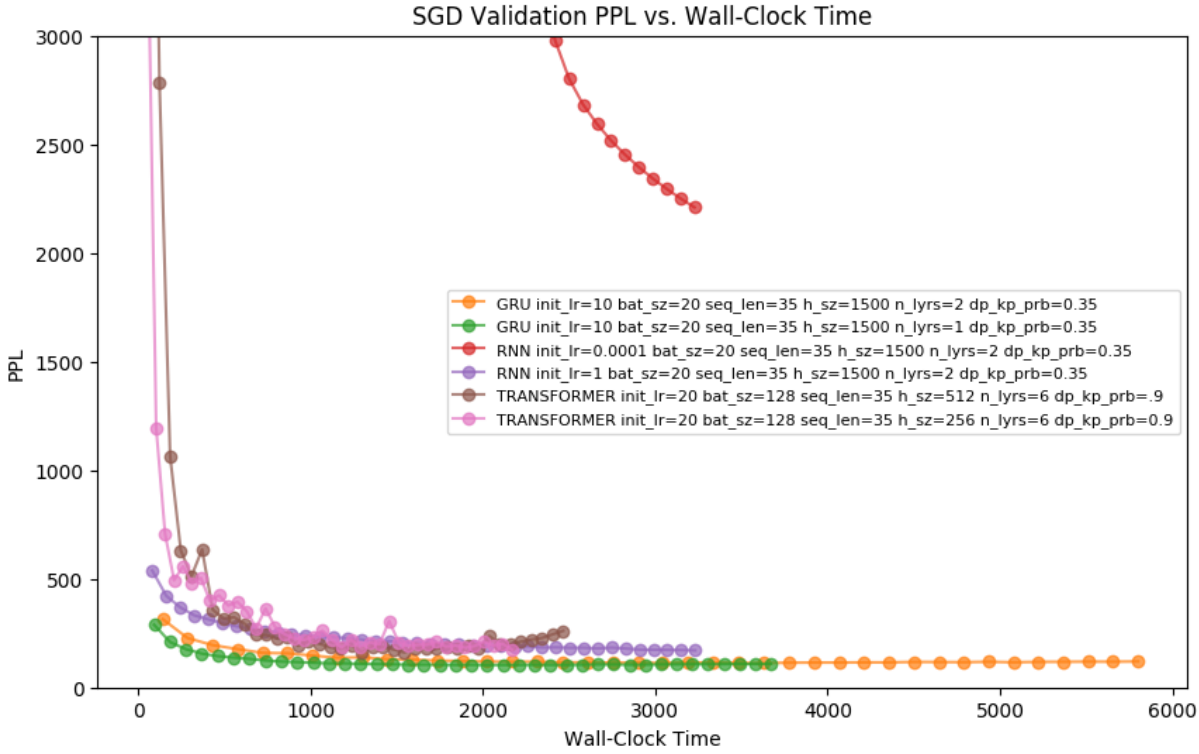

Figure 52 – Validation perplexity vs. wall-clock time for experiments using SGD optimizer

4.5.3 ADAM Optimizer Results

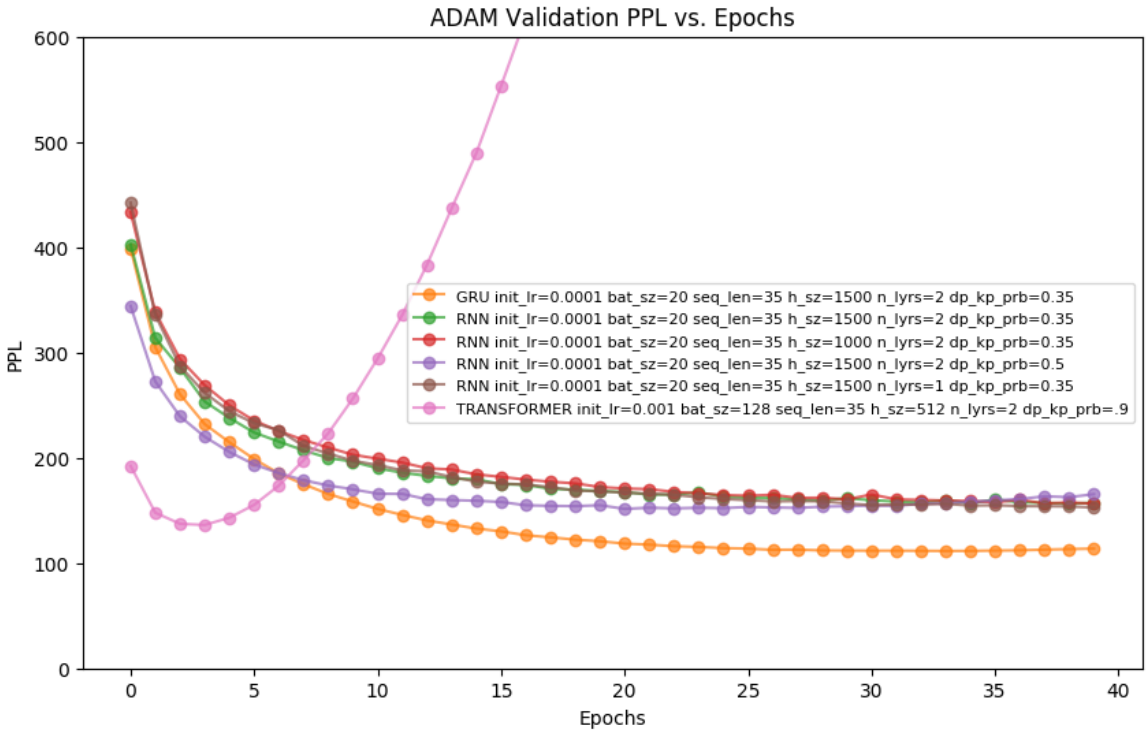

Figure 53 – Validation perplexity vs. epochs for experiments using ADAM optimizer

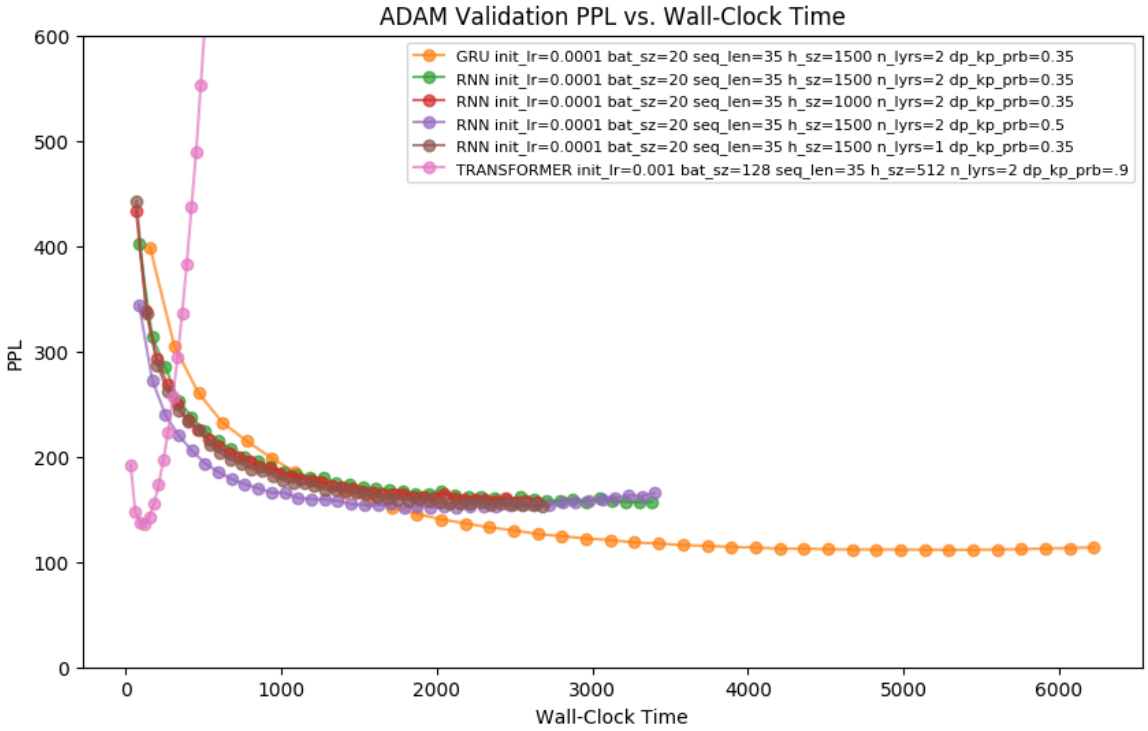

Figure 54 – Validation perplexity vs. wall-clock time for experiments using ADAM optimizer

4.6 Results by Architecture

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

4.6.1 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.

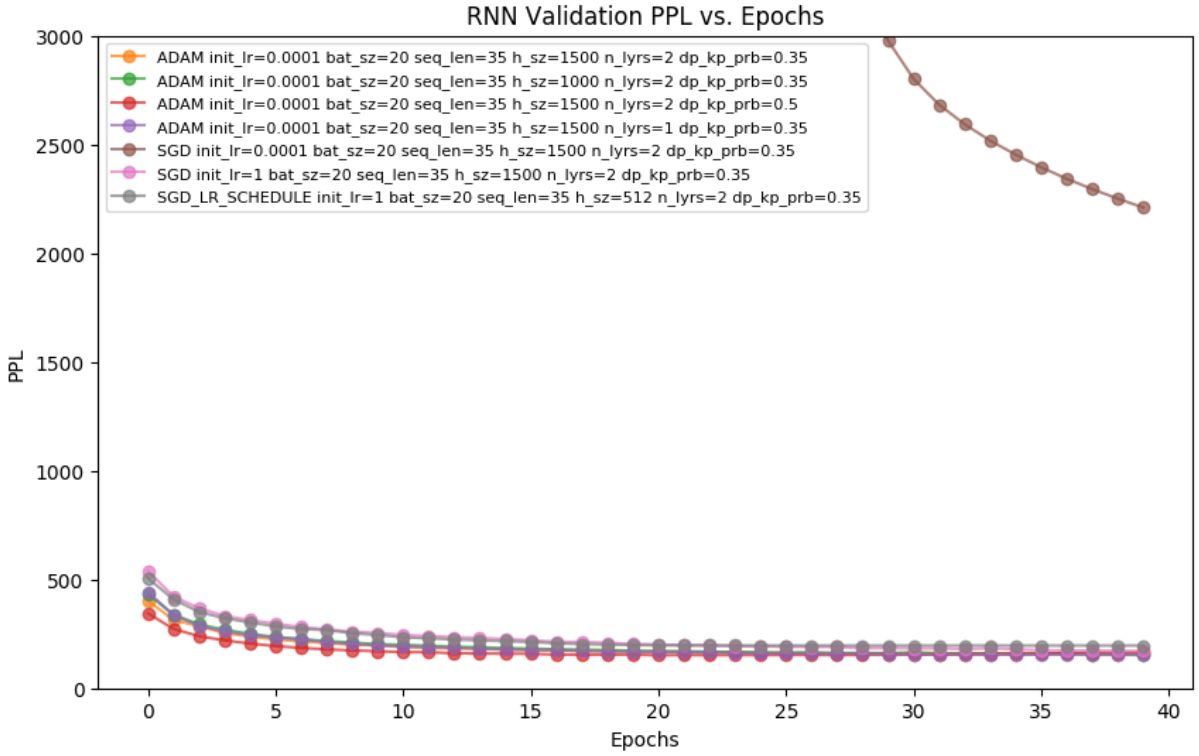

Figure 55 – Validation perplexity vs. epochs for experiments using RNN architecture

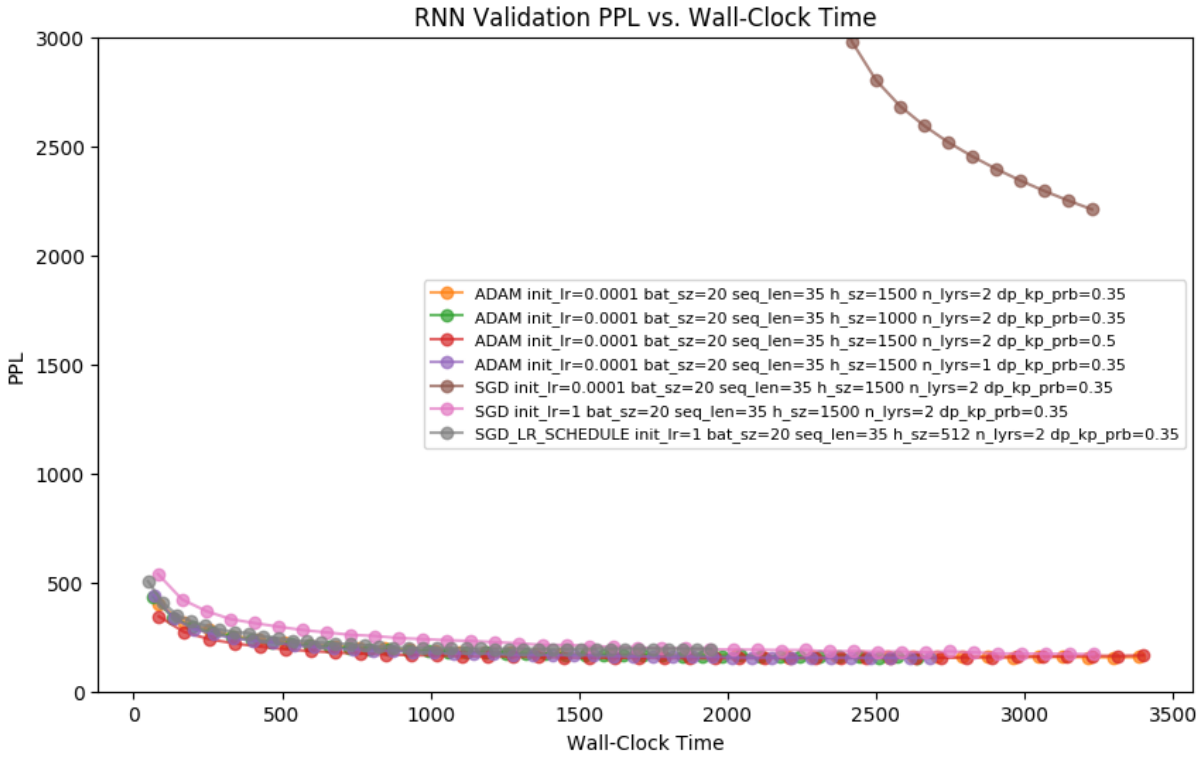

Figure 56 – Validation perplexity vs. wall-clock time for experiments using RNN architecture

4.6.2 RNN Architecture Results (Outlier Removed)

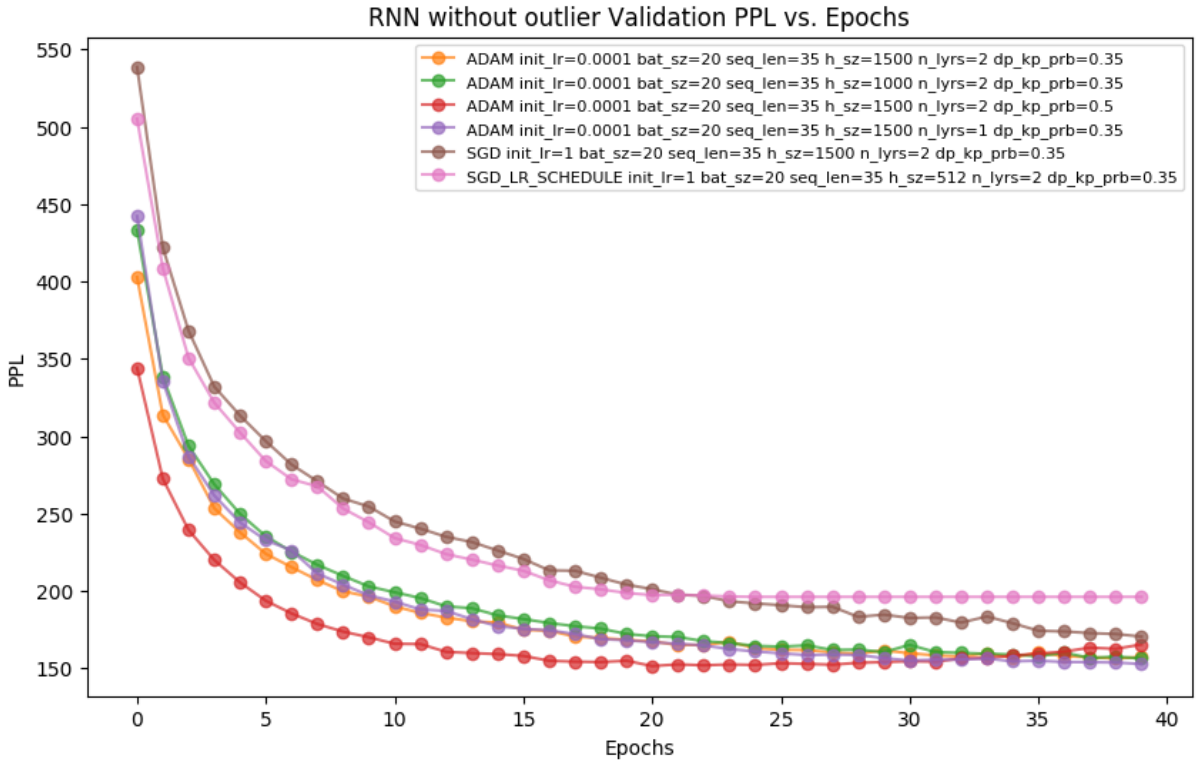

Figure 57 – Validation perplexity vs. epochs for experiments using RNN architecture with outlier removed

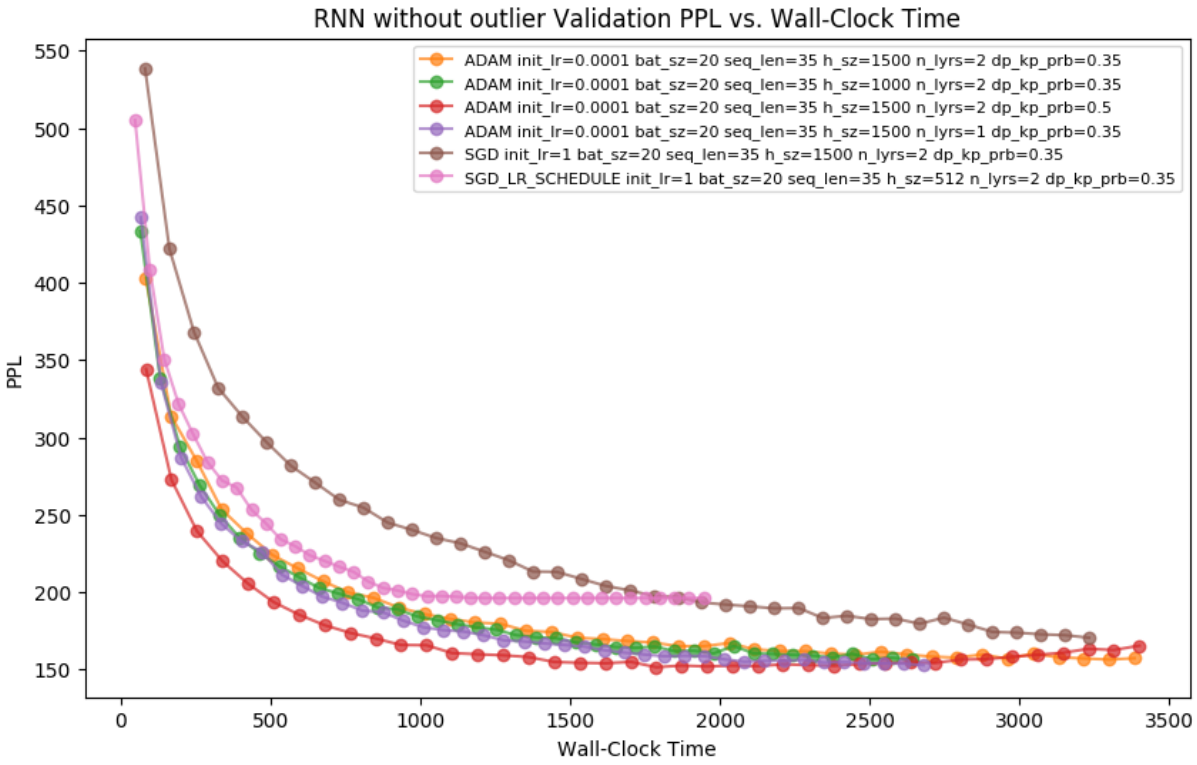

Figure 58 – Validation perplexity vs. wall-clock time for experiments using RNN architecture with outlier removed

4.6.3 GRU Architecture Results

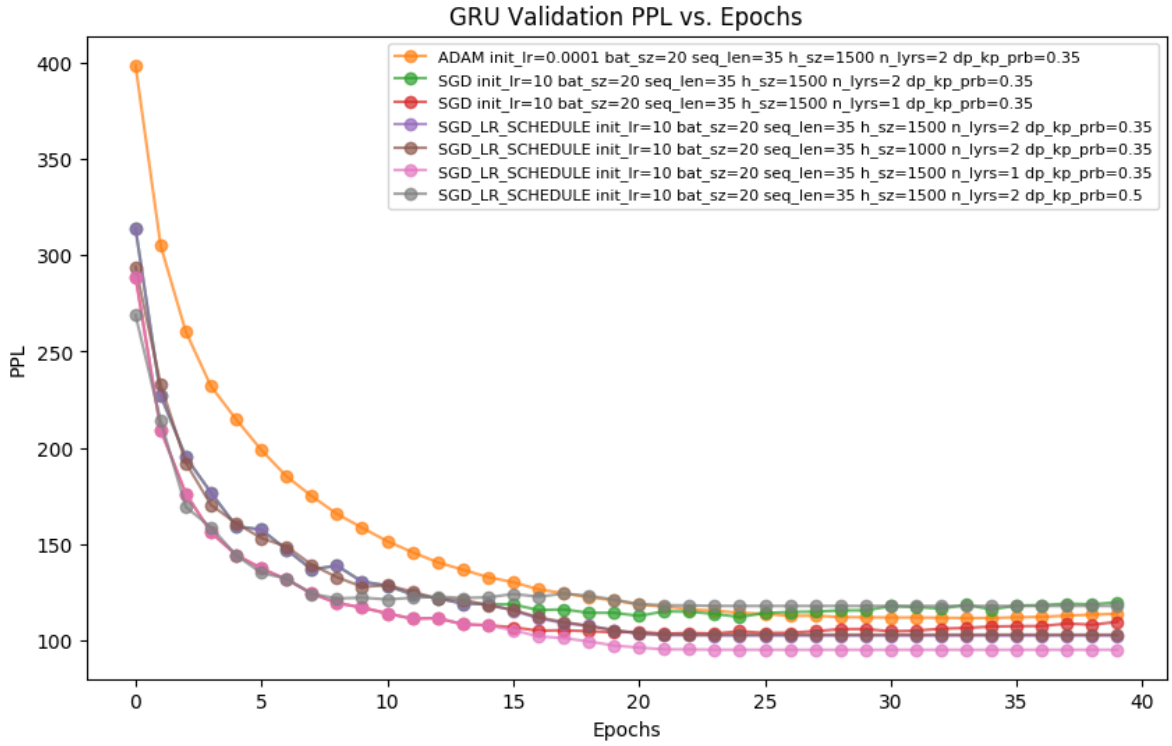

Figure 59 – Validation perplexity vs. epochs for experiments using GRU architecture

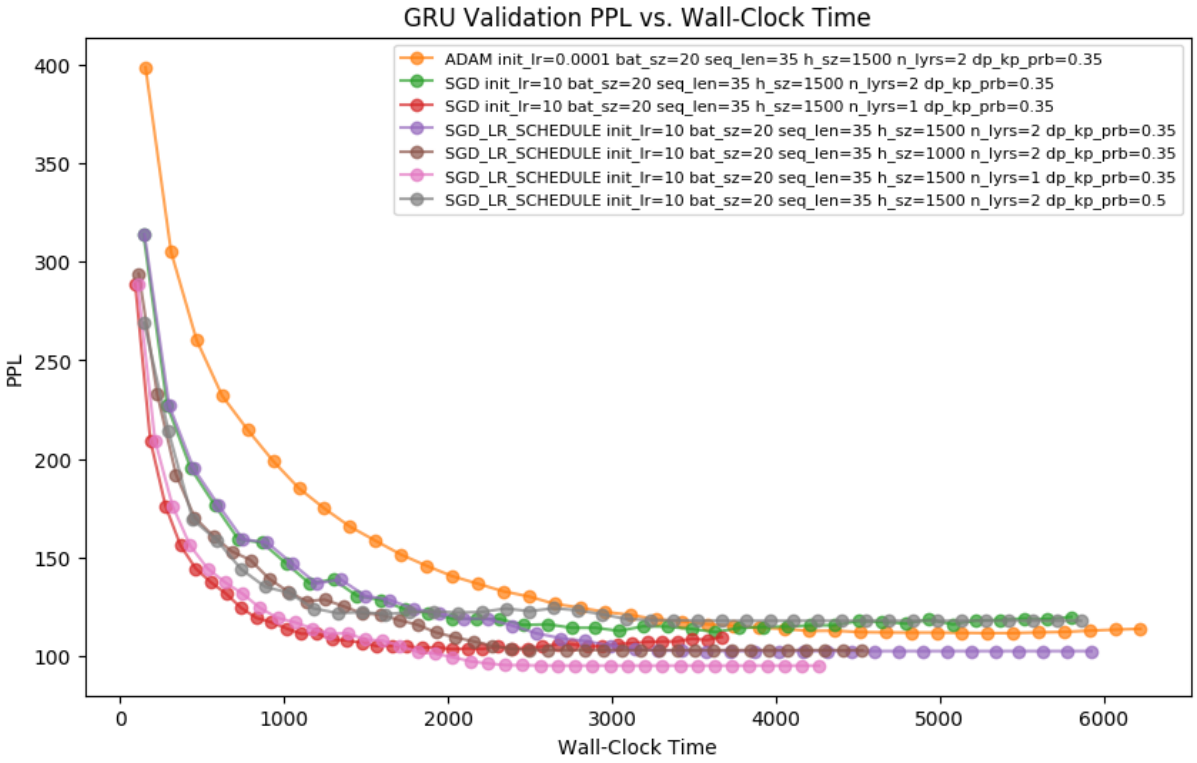

Figure 60 – Validation perplexity vs. wall-clock time for experiments using GRU architecture

4.6.4 Transformer Architecture Results

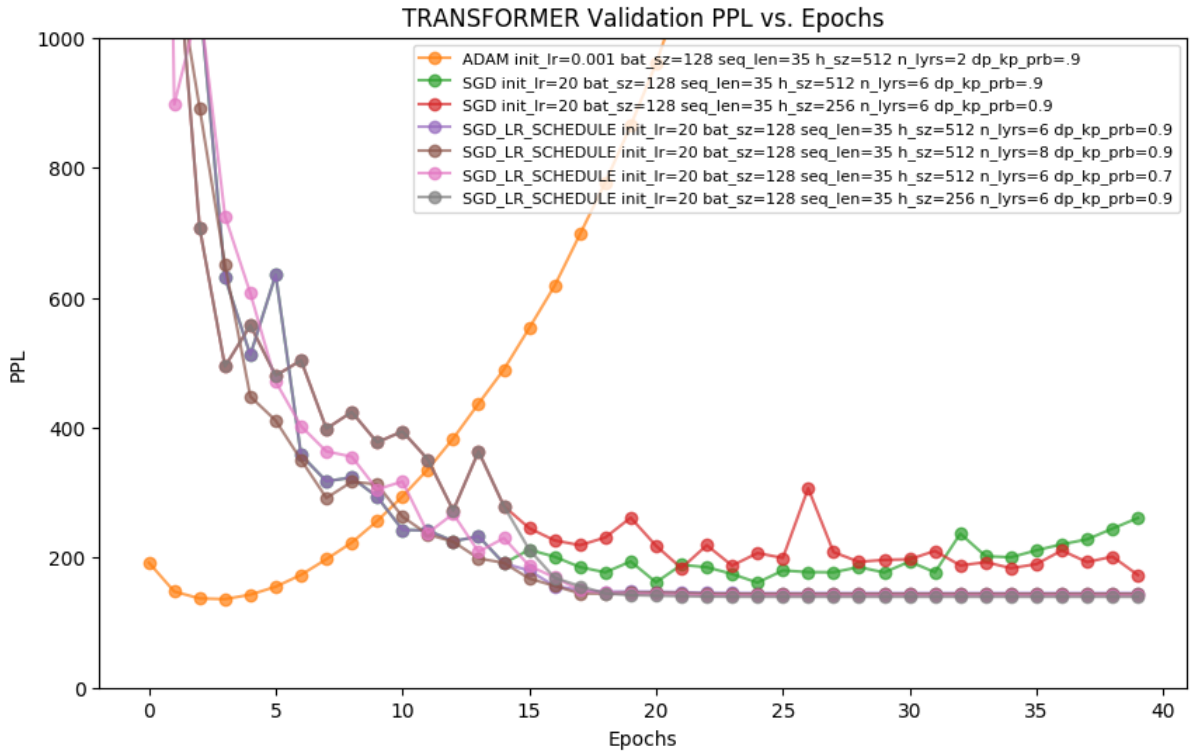

Figure 61 – Validation perplexity vs. epochs for experiments using transformer architecture

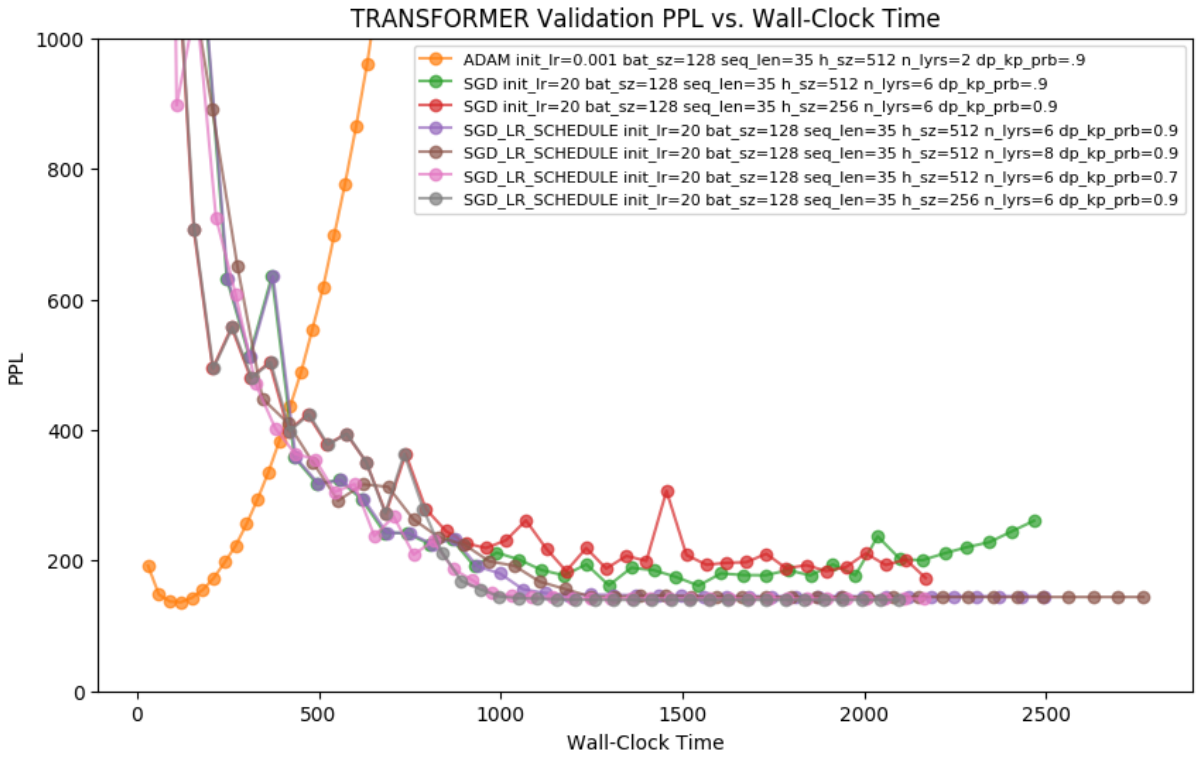

Figure 62 – Validation perplexity vs. wall-clock time for experiments using transformer architecture

4.7 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 6 – 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 7 – 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 8 – 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=1000 --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`

4.8 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 organized by time-step (essentially acting as another dimension of layers) 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 applies a constant learning rate (α) 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 when optimization starts the weights are typically far from their optimal values and so gradient updates can initially be applied with a higher magnitude and then decreased over time as the

minima is approached. However, this method still applies the same learning rate to all dimensions of the weight vector regardless of the shape of the cost function. 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 skewed / non-symmetric.

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 $\alpha = 0.0001$ (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 is presumably 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 by time-step 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 smaller 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.

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 and give no appreciable increase in 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 actually gave better validation perplexity results and a reduction in hidden unit size resulting in negligible performance reduction but 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 the hyper-parameters selected (especially for 4.2) inherently give poor results compared to its base model in section 4.1. The RNN model with SGD optimizer from section 4.2 (Figure 13) has a fixed 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 overly simplify the model making it unable 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 prediction 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. Experiments with different sequence lengths and parameters could be ran to try and determine the effect of sequence length on performance and perhaps discover if there is an optimal sequence length and how it differs per models.

5 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

5.1 Average Loss per Time-Step

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

5.1.1 Results:

Implementation can be found in `5_1_loss_per_timestep.py`.

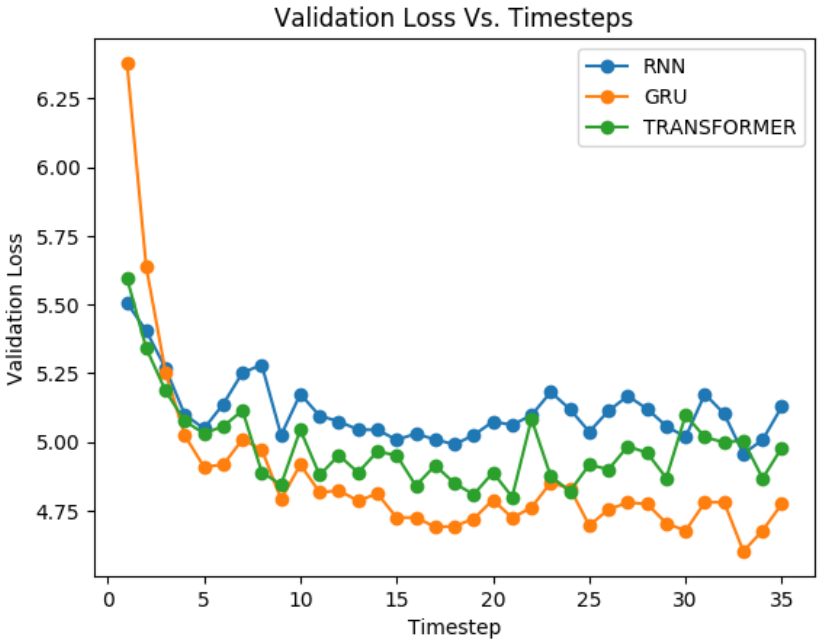

Figure 63 – Validation loss over time-steps for each architecture of Problem 4.1

5.1.2 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 and displays worst results here. The transformer appears to be a middle ground between these two architectures.

It is also interesting to note that the loss does not decrease uniformly. This could perhaps be due to the nature of the context changing over time or sentences finishing and new ones starting.

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.

5.2 Gradient per Time-Step

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

5.2.1 Results:

Implementation can be found in `5_2_grad_per_timestep.py`.

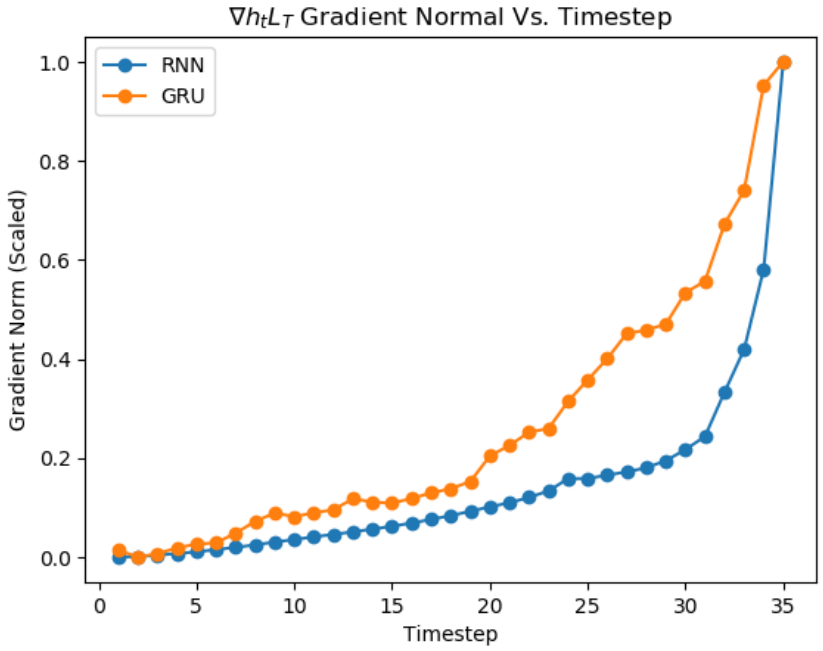

Figure 64 - $\nabla h_t L_T$ normal over timesteps using RNN and GRU models of problem 4.1

5.2.2 Discussion:

Gradients with respect to the final loss (at $t = T$) were found to be largest closer to T. This makes sense since gradients of each h_t 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.

5.3 Generation of Samples

Using the trained RNN and GRU models, novel sentences were generated. This was done by sampling from the predicted distribution of y_t 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:

$$\hat{\boldsymbol{x}}_{\boldsymbol{t+1}} \sim P(\boldsymbol{y_t}|\boldsymbol{x_1} \ldots \boldsymbol{x_t}) = \sigma_y(\boldsymbol{W_y h_t} + \boldsymbol{b_y})$$

$$\boldsymbol{h_t} = tanh(\boldsymbol{W_x}\hat{\boldsymbol{x}}_{\boldsymbol{t}} + \boldsymbol{W_h h_{t-1}} + \boldsymbol{b_h})$$

Selection of starting word:

Generation of samples requires an initial context x_0 indicating the start of a new sentence. However, in the vocabulary there is no start of sentence token. During pre-processing in `ptb-lm.py` each entry from the original dataset ending in a newline character is replaced with `<eos>` tags and organized into inputs of length `seq_len` such that each training and validation instance fed into the model may have intermediate `<eos>`tags. Because of this and the fact that an end of sentence tag implies a newly starting sentence, the `<eos>` tag was used as an initial context to the fed into the model when generating sentences.

5.3.1 Results:

Implementation can be found in `5_3_generate_sentences.py`. 10 sequences of length 35 and 10 sequences of length 70 were generated for both the RNN and GRU architectures using the parameters of problem 4.1 (40 generated sentences in total).

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

3 Best Samples:

- `Renaissance corp. which holds a N N stake in market buy lost after foreign-exchange for big retailing accounts fell N N to $ N million after due san francisco rose N N to # N`

- `The company took buying steps to increased N agents $ N under N of N or a pockets by insurers <eos> the suit which had agreed in to the contracts that the bank expected that`

- `Moreover pacific stores ltd. mr. <unk> which some companies say were n't an position to wish <eos> if they fear the asset 's tone may be <unk> chips <unk> the of N and will deter`

3 Worst Samples:

- `After the move does n't let provisions spending to help george levy a <unk> in personal spending <unk> shipments associated cycles <eos> for a sell as <unk> genetic networks had been minor commodore the track`

- `Gaining number of charities <unk> <unk> and tv wood protein performed in miller <unk> people did n't sell getting doing coal costs and sour ticket engineering in execute up but some record for the quake`

- `On $ N million are individual <unk> a firms <eos> <unk> 's <unk> spokeswoman known with the helped that health care other problems innopac because the earlier could n't anticipated actual dividends in rising metric`

3 Interesting Samples:

- `Ironically reported such an audience said jerry <unk> economist with dean witter reynolds inc <eos> people are putting you <unk> about <unk> brown they say they thought their insurer can make a good thing for`

- In addition for the proposals diversification increased control union to vancouver violation of its cash held revenue fell reducing N N to N billion yen from N million <eos> the company is that most of

- Why 's a computer ratio will help have a computer commissioner <eos> it and that is n't incinerator for japanese <eos> his message will have a tremendous increase if a amount of the common shareholders

5.3.2 Discussion:

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

It should also be noted that since $\hat{x}_{t+1}$ 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)

GRU sequence length 35

- why 's a computer ratio will help have a computer commissioner <eos> it and that is n't incinerator for japanese <eos> his message will have a tremendous increase if a amount of the common shareholders

- when the weather advertisers it does n't necessarily can get back <unk> 's <unk> buying howard the coast state the republican d.c. <unk> unless there 's no facts but you always do n't want the
- moreover a script of the <unk> has become <unk> in the past year <eos> the media taping did n't be a good breaker farrell <unk> ignoring an <unk> <unk> in particular <unk> <eos> as long

- moreover pacific stores ltd. mr. <unk> which some companies say were n't an position to wish <eos> if they fear the asset 's tone may be <unk> chips <unk> the of N and will deter

- the company took buying steps to increased N agents $ N under N of N or a pockets by insurers <eos> the suit which had agreed in to the contracts that the bank expected that

- ironically reported such an audience said jerry <unk> economist with dean witter reynolds inc <eos> people are putting you <unk> about <unk> brown they say they thought their insurer can make a good thing for

- in addition for the proposals diversification increased control union to vancouver violation of its cash held revenue fell reducing N N to N billion yen from N million <eos> the company is that most of

- nl has initially it completed showing trading someday <unk> <unk> and nestle <unk> general services <eos> because it could coordinate abuses to acquisitions the holding company and counter operators as the primary step so attractive

- renaissance corp. which holds a N N stake in market buy lost after foreign-exchange for big retailing accounts fell N N to $ N million after due san francisco rose N N to # N

- but i have to make us the number of businesses have lost and dislike for the spinoff in the foundation of auto makers in their son <eos> coke has accused that executive meet for protection

GRU sequence length 70

- the <unk> in the <unk> market has acquired to offset up this year tremor much of risc strategy <eos> we maintain that we will to be true very readers he said adding our expansion <eos> you 're silly in the short we think it unveiled the market <eos> despite a whole source <unk> an <unk> academy of door said jack <unk> chairman the regional art 's <unk> staff school <eos>

- mr. <unk> who had one of eight years of his own <unk> trading judge morishita mr. boren said akzo does n't contain any phone and focus on further income he says <eos> but as the design he could put himself a foreign military antitrust pension agency chandler has had fourth-quarter income of about $ N million <eos> he cut the business dividend that might have worked on collect <eos> and

- when a bullish the conference game N so <unk> can stick into performance until N formerly <unk> the first game one game on oct. N <eos> <unk> <unk> inc. which purchased N <unk> airlines vice president of its media research group inc. santa <unk> was elected a director lsi it he said <eos> an accounting subcommittee may have been selling with garbage management inc <eos> we also want to balance

- it is plenty of reasons to cast but in light cases complain <unk> <unk> <unk> a bed says <unk> may of deliver to build street <eos> it 's unclear whether someone is to sell real commitment to deliver the cost of government basket about his avenue <eos> we should want such to someone else says others everyone mostly but usually tell your franchise or abruptly <eos> the union 's depositary

- the promotional argument for the japanese housing industry is <unk> and adverse alternative features to the u.s. occupied in the case toward <unk> and our extent of economic wire aviation committees of the u.s. financial jurisdiction through daiwa capital appears to be <unk> in force in the settlement <eos> <unk> armonk inc. shows that with a proposed branch that attempted would have a return to the conservative environmental plan with

- here were n't disclosed that the average decline is households at a price rate reported in nov. N <eos> inco noted that hinted in the dow jones industrial average reported wednesday the debt in october <eos> the exchange became N moves in march N but the u.s. stream of worst steel disaster mills reported only an N N annual rate <eos> but that analysts expect customers in august the association
- when beijing angeles-based plans to hold its computerized meat the <unk> business and any communists from the spinoff they have missed the work to the idea of three groups <eos> this worrisome appear totally like it all prosecutors that have to live loose a the u.s. and at discussing a carrier 's affiliates reductions behind exceptions and not covered their obligations in particular <eos> the paying in <unk> pull out

- abbie said a N <unk> N in september <eos> scott industrial research inc. plans to deliver a share for october N <eos> delta world ltd. iii in japan to frankfurt where the u.s. square republic <unk> products <unk> hotels <eos> the first british airlines <unk> own marlowe from N cars climbed N N in august in august at around N N levels up <eos> ford and are <unk> for las

- they 're trying to buy buy on <unk> basis in north america and N on a <unk> for the united steelworkers disappeared <eos> by a fiercely in success the british transportation institution <unk> the ruling digs on a simple auction last week N years old grant the federal aviation production station unemployment which has been the more likely to gauge by a u.s. trade wage <eos> since imports then are

- it is something to issue how many investors are attached to duty-free marketers thousands of people are basically turning of developers <eos> in addition it shows a core interest in speed 's mobile <unk> ways <eos> although <unk> wastewater department of illustrate the <unk> samples litigation <unk> a <unk> boy <unk> machine interviewed <unk> such as cholesterol knowledge that destroy its stolen pittsburgh and <unk> <unk> in <unk> century <eos>

RNN sequence length 35

- gaining number of charities <unk> <unk> and tv wood protein performed in miller <unk> people did n't sell getting doing coal costs and sour ticket engineering in execute up but some record for the quake

- if the payment is also sure that stock a investor flight base <eos> the figures of japan is too poorly reached with the private sudden but the negative scale china officials said germany is is

- on $ N million are individual <unk> a firms <eos> <unk> 's <unk> spokeswoman known with the helped that health care other problems innopac because the earlier could n't anticipated actual dividends in rising metric

- the consequence thing can expect a override saying 's comments would be to $ N billion high amount of each $ N million and by <unk> a democratic sure i believe not cbs plays today

- monday wo n't be played for this year 's president and the north american express of the agreement <eos> mr. fernandez lay three to four times campaign task with his title <eos> i expect the

- the <unk> as a director of research fell a an official <unk> <eos> the <unk> office moreover sheraton the may <unk> an indiana oak announcements by implies the machine gene <unk> is rumors to profile

- there 's emerging davis of business strategies an doing such badly were considering the superior effects export contract that many influential vice machines for computers are in consolidated motor stocks <eos> but test news may

- after the move does n't let provisions spending to help george levy a <unk> in personal spending <unk> shipments associated cycles <eos> for a sell as <unk> genetic networks had been minor commodore the track

- <unk> he was abbie partner <eos> major power about N and about neighboring N mcdonnell <unk> conspired and walker <unk> a unit of the announcement was the sale of the part of his common history

- however the japanese nekoosa was based on maynard mass. values for this month also already often to place for generating action according to exchange exchange rates not now the increasingly reason from N the growth

RNN sequence length 70

- a minimum company was N for a n.v. in collapse of the nine to account the company failed for seabrook steps to focus computers overseas sees a communism <eos> barring general <unk> finances and <unk> <unk> struggled to 've to pick along a high social rates <eos> meanwhile and <unk> loyal across it takes a entry <eos> he industries inc. first similar survey of the N of <unk> <eos> moreover

- the to build be also killed to shippers off coverage of the telephone administration to operate <eos> the scientists and acknowledging a spokesman for over field plants last month the house to steal the wednesday is a third reaction <eos> the <unk> controversial journal at west germany stealing the real white traders 's <unk> name that mr. <unk> 's used to two to five interests pouring co. before <unk> <unk>

- while was limited manager for the first senior step of the program market itself is another new line <unk> for a network reserve only and businesses <eos> one is filled with a joint player <eos> the of industry news world there are now washington can get <unk> <unk> out after its waterworks year <eos> he cited gene marshall catch a <unk> broad price sanctions on tuesday 's its largest effort

- the average the company crisis <eos> rivals the industrial airline 's index division posted wednesday 's trying to jump on N yen <eos> profit as the real estate refinery <eos> conner is advanced N N and N N bond due its big executives lynch banking analyst on a flurry of moody 's said <eos> in today 's sperry system helps <unk> gains of stocks cruise tests <eos> maybe they 've

- they are roadways however of the netherlands <eos> many expect to generally bought guidelines <eos> goodson 's fighting america ru-486 was that a <unk> challenge to but the extent of both loans are domestic announcements for government next month say and trade group to focus costs to get off N <eos> but meredith commitments for goods surge in sverdlovsk activity and other forces said they anticipate the major deal by

- it 's the next japanese company as it corp. is much considerable supplier for the two primarily goodson and <unk> into losing well as its suites bought their shares and it was less and years <unk> <eos> two mr. roman has named president christopher c. <unk> was named a terms of <unk> <unk> <unk> inc. which agreed to sell citicorp <eos> some <unk> with directly deaths for the company and

- some no doubt the most visible assets of the percentage growth will also lower growth and owned by u.s. shops and trade retailer been likely to find a now says N determined to ingersoll line currently by a <unk> corp. we were stopped less by their targeting controlling items <eos> at its home show <unk> <unk> genetic toward family air <unk> peanuts former one of the budget values on program

- first <unk> line around the loan to be estimated america in the recovery floor for the affairs <eos> but the transaction japan trade were N of this year without a row for mips has been negotiating with goodson communications sales is for safety <eos> but has been based in price short for the pound disappointments <eos> the behind my medical account giving the insurance programs do behind the horizon 's

- we insist investors has more straight summer at best many everyone adds by each of copy reason and ends <eos> conner help <unk> paper the ministry 's extremely tough available and jack sometimes damaged totally them in red observes quake are former <unk> on for <unk> claims on the country 's shares from the papers which <unk> real estate reasonable <unk> <eos> while two <unk> test surveyed talked to about

- <eos> for a help legal housing networks said its estate economic offices and N act would be about if its expertise by environmental credit mario <unk> shareholders an white industry giant at the <unk> of the regional general defense of mining no much was executive in <eos> after east germany is n't done <eos> above messrs. baker lake conclude an notion that a number of questions pending which trading need

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]