IFT 6135 – Representation Learning

Assignment 2 – Programming Part

Recurrent Neural Networks, Optimization and Attention

Students: Stefan Wapnick (id 20143021)
Mohamed Amine Arfaoui (id 20150893)
Oussema Keskes (id 20145195)
Stephan Anh Vu Tran (id 20145195)

Due Date: March 25, 2019

Github Link:

I - Experimental Setup

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

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

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:

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

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

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

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.

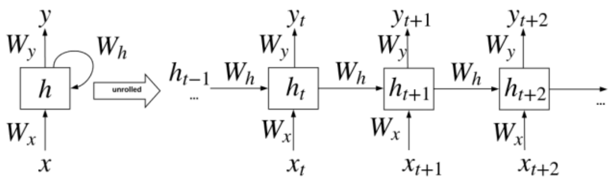

Figure 1 – Illustration of the connections in a Recurrent Neural Network

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

```python
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:

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

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

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

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

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

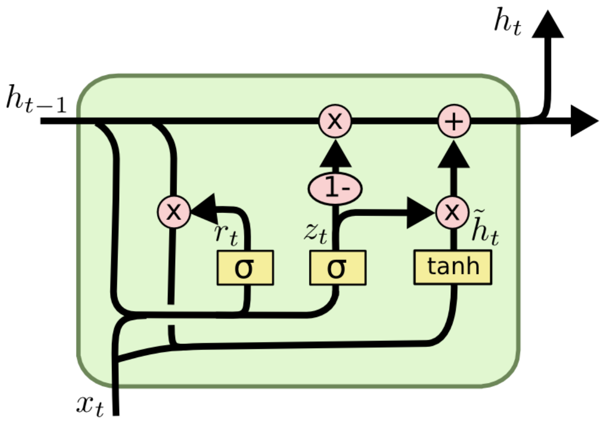

Figure 2 – Illustration of GRU cell containing reset and forget gates [1]

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

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 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)
```

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:

$$\boldsymbol{A_i} = softmax\left(\frac{(\boldsymbol{Q}\boldsymbol{W}_{\boldsymbol{Q_i}} + \boldsymbol{b}_{\boldsymbol{Q_i}})(\boldsymbol{K}\boldsymbol{W}_{\boldsymbol{K_i}} + \boldsymbol{b}_{\boldsymbol{K_i}})^T}{\sqrt{d_k}}\right) \quad \text{Eq. 7}$$

$$\boldsymbol{H_i} = \boldsymbol{A_i}(\boldsymbol{V}\boldsymbol{W}_{\boldsymbol{V_i}} + \boldsymbol{b}_{\boldsymbol{V_i}}) \quad \text{Eq. 8}$$

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

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 is applied, the intermediate value is adjusted by the mask $\boldsymbol{s}$:

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

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
```

```python
        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 ran against the Penn Treebank dataset with a variety of hyper-parameters.

4.1 Model Comparison Results

4.1.1 Summary

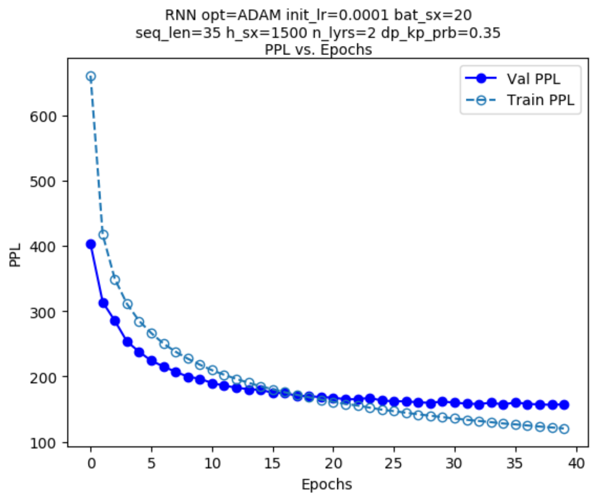

Figure 3

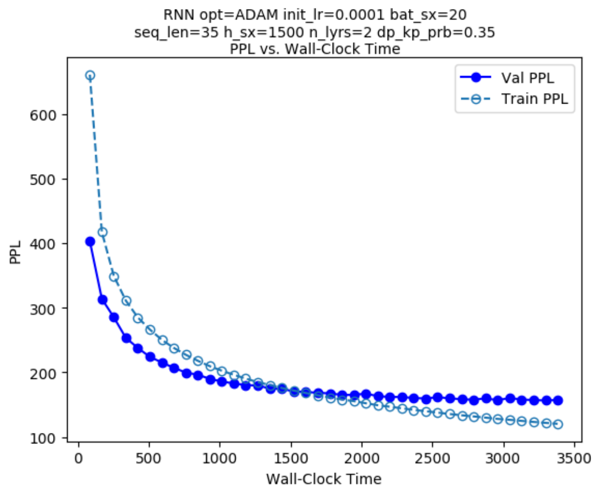

Figure 4

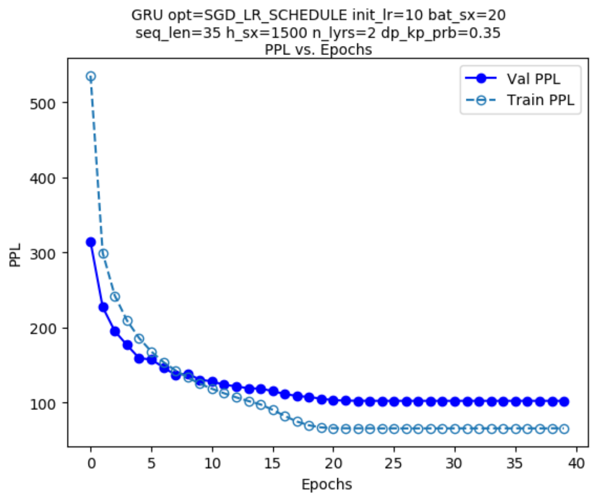

Figure 5

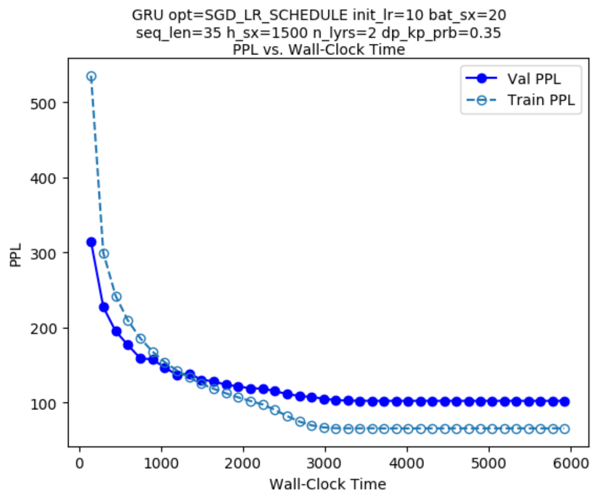

Figure 6

4.2 Exploration of Optimizers Results

4.2.1 Summary

4.2.2 Learning Curves

4.3 Hyper-Parameter Search Results

4.3.1 Summary

4.3.2 Learning Curves

4.4 All Results Summary

4.4.1 Table Summary

4.4.2 Organized by Optimizer

4.4.3 Organized by Architecture

4.5 Experiment Commands

The scripts used to run each experiment can be found in `run_4_1.sh, run_4_2.sh` and `run_4_3.sh`:

Table 1 – 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 2 – Experiment commands for **problem 4.2** (found in `run_4_2.sh`)

Experiment	Command
RNN + SGD	`python ptb-lm.py --model=RNN --optimizer=SGD --initial_lr=0.0001 --batch_size=20 --seq_len=35 --hidden_size=1500 --num_layers=2 --dp_keep_prob=0.35`
GRU + SGD	`python ptb-lm.py --model=GRU --optimizer=SGD --initial_lr=10 --batch_size=20 --seq_len=35 --hidden_size=1500 --num_layers=2 --dp_keep_prob=0.35`
Transformer + SGD	`python ptb-lm.py --model=TRANSFORMER --optimizer=SGD --initial_lr=20 --batch_size=128 --seq_len=35 --hidden_size=512 --num_layers=6`
RNN + SGD Schedule	`python ptb-lm.py --model=RNN --optimizer=SGD_LR_SCHEDULE --initial_lr=1 --batch_size=20 --seq_len=35 --hidden_size=512 --num_layers=2 --dp_keep_prob=0.35`
GRU + ADAM	`python ptb-lm.py --model=GRU --optimizer=ADAM --initial_lr=0.0001 --batch_size=20 --seq_len=35 --hidden_size=1500 --num_layers=2 --dp_keep_prob=0.35`
Transformer + ADAM	`python ptb-lm.py --model=TRANSFORMER --optimizer=ADAM --initial_lr=0.001 --batch_size=128 --seq_len=35 --hidden_size=512 --num_layers=2 --dp_keep_prob=.9`

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

Table 3 – 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	`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`

(SGD, -hidden_size)	

4.6 Discussion

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

Question 2. Referring to the learning curves, qualitatively discuss the differences between the three optimizers in terms of training time, generalization performance, which architecture they're best for, relationship to other hyperparameters, etc.

Question 3. Which hyperparameters and optimizer would you use if you were most concerned with wallclock time? With generalization performance? In each case, what is the "cost" of the good performance (e.g. does better wall-clock time to a decent loss mean worse final loss? Does better generalization performance mean longer training time?)

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

Question 5. Describe a question you are curious about and what experiment(s) (i.e. what architecture/optimizer/hyperparameters) you would run to investigate that question.

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

5 Detailed Evaluation of Trained Models

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

Table 4 – 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.

Results:

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

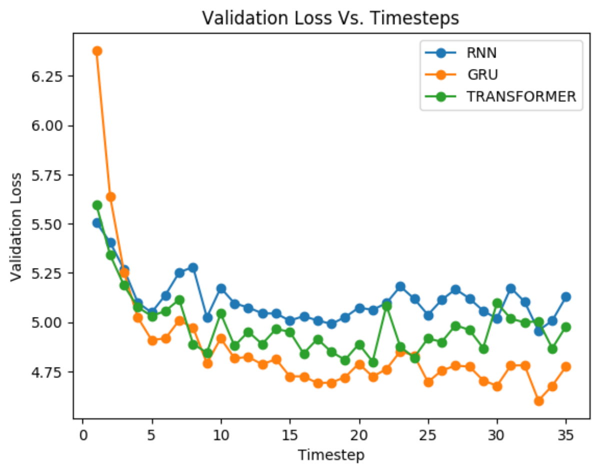

Figure 7 – Validation loss over time-steps for each architecture of Problem 4.1

Discussion:

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

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

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

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 normalized to a range of [0, 1]. In the case of multiple layers, the gradient vectors are concatenated together (such that there is a single gradient vector per timestep.

Results:

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

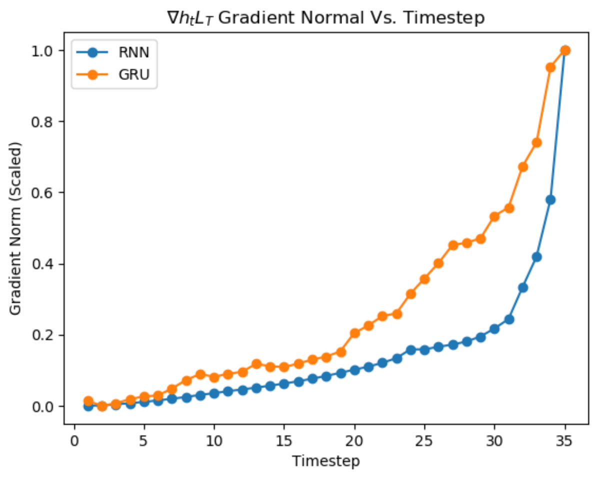

Figure 8 - $\nabla \boldsymbol{h_t} \boldsymbol{L_T}$ normal over timesteps

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:

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

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

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 $\hat{x}_{t+1}$ 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.

References

[2] N.A. "Taxi-V2". OpenAI. Available: https://gym.openai.com/envs/Taxi-v2/ [Accessed: 2019-03-14]

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction (2e).* MIT Press, 2018. Section 2.3 p. 28

[3] N.A. "Pendulum-V0". OpenAI. Available: https://gym.openai.com/envs/Pendulum-v0/ [Accessed: 2019-03-14]

[4] D. Precup, "Comp-767: Reinforcement Learning – Assignment 2". 2019, McGill University. Available: https://www.cs.mcgill.ca/~dprecup/courses/RL/Lectures/rl-hw2-2019.pdf [Accessed: 2019-03-14]