Solution to IFT6135 Practical Assignment 2

Team Member:

Kun Ni(20139672), Yishi Xu(20125387), Jinfang Luo(20111308), Yan Ai(20027063)

Github repo link

This file includes the code snipts for Practical queations 1-3. Please check this github link (https://github.com/ekunnii/ift6135-submission/tree/master/assignment2) for complete codes

plain link: https://github.com/ekunnii/ift6135-submission/tree/master/assignment2 (https://github.com/ekunnii/ift6135-submission/tree/master/assignment2)

Vanila RNN

In [ ]:

```
class RNNCell(nn.Module):
    '''
    a basic RNN cell,
    '''

    def __init__(self, input_size, hidden_size):
        """
        Most parts are copied from torch.nn.RNNCell.
        """

        super(RNNCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.weight_ih = nn.Parameter(torch.Tensor(input_size, hidden_size))
        self.weight_hh = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
        self.bias = nn.Parameter(torch.Tensor(hidden_size))
        self.tanh = nn.Tanh()

        self.reset_parameters()

    def reset_parameters(self):
        """
        Initialize parameters following the way proposed in the paper.
        """
        stdv = 1.0 / math.sqrt(self.hidden_size)
        for weight in self.parameters():
            init.uniform_(weight, -stdv, stdv)

    def forward(self, inputs, hx):
        """
        Args:
            inputs: A (seq_len, batch ,input_size) tensor containing input
                features.
            hx: the initial hidden

        Returns:
            outputs: layers outputs
            hx: Tensors containing the next hidden.
        """
        # cell ticks trough seq len, and then return the output and  cell states
        batch_size = hx.size(0)
        bias_batch = self.bias.unsqueeze(0).expand(
            batch_size, self.bias.size(0))

        max_time = inputs.size(0)

        outputs = []
        for time in range(max_time):
            input_x = inputs[time]

            xw = torch.addmm(bias_batch, input_x, self.weight_ih)
            hu = torch.mm(hx, self.weight_hh)

            hx = self.tanh(hu + xw)

            outputs.append(hx)
            # # pass the hidden status to next tick

        outputs = torch.stack(outputs, 0)
        return outputs, hx
```

In [ ]:

```
# Implement a stacked vanilla RNN with Tanh nonlinearities.
class RNN(nn.Module):
    def __init__(self, emb_size, hidden_size, seq_len, batch_size, vocab_size, num_layers,
dp_keep_prob):
        """
        emb_size:     The number of units in the input embeddings
        hidden_size:  The number of hidden units per layer
        seq_len:      The length of the input sequences
        vocab_size:   The number of tokens in the vocabulary (10,000 for Penn TreeBank)
        num_layers:   The depth of the stack (i.e. the number of hidden layers at
                      each time-step)
        dp_keep_prob: The probability of *not* dropping out units in the
                      non-recurrent connections.
                      Do not apply dropout on recurrent connections.
        """
        super(RNN, self).__init__()

        # TODO =======================
        # Initialization of the parameters of the recurrent and fc layers.
        # Your implementation should support any number of stacked hidden layers
        # (specified by num_layers), use an input embedding layer, and include fully
        # connected layers with dropout after each recurrent layer.
        # Note: you may use pytorch's nn.Linear, nn.Dropout, and nn.Embedding
        # modules, but not recurrent modules.
        #
        # To create a  number of parameter tensors and/or nn.Modules
        # (for the stacked hidden layer), you may need to use nn.ModuleList or the
        # provided clones function (as opposed to a regular python list), in order
        # for Pytorch to recognize these parameters as belonging to this nn.Module
        # and compute their gradients automatically. You're not obligated to use the
        # provided clones function.
        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

        # Check dropout
        if not isinstance(dp_keep_prob, numbers.Number) or not 0 <= dp_keep_prob <= 1 or is
instance(dp_keep_prob, bool):
            raise ValueError(
                "dropput should be a number in range[0, 1],"
                "represneting the probability of an element being zeroed")

        if dp_keep_prob > 0 and num_layers == 1:
            warnings.warn(
                "dropout options adds dropout after all but last"
                "recurrent layer, so non-zero dropout expects"
                "num_layers greater than 1, but got dropout={} and"
                "num_layers ={}".format(dp_keep_prob, num_layers))

        # Embedding layers
        self.emb = nn.Embedding(vocab_size, emb_size)
        self.cell_stack = nn.ModuleList([])
        for layer in range(num_layers):
            layer_input_size = emb_size if layer == 0 else hidden_size
            self.cell_stack.append(RNNCell(input_size=layer_input_size,
                                           hidden_size=hidden_size))

        self.hidden2out = nn.Linear(hidden_size, vocab_size)

        self.dropout_layer = nn.Dropout(1-dp_keep_prob)
        self.tanh = nn.Tanh()
        self.init_weights_uniform()
        self.softmax = nn.Softmax(1)

    def init_weights_uniform(self):
        # TODO =======================
        # Initialize all the weights uniformly in the range [-0.1, 0.1]
        # and all the biases to 0 (in place)
        # initialize the weight of all hidden units

        stdv = 0.1
        nn.init.uniform_(self.emb.weight, -stdv, stdv)
        nn.init.uniform_(self.hidden2out.weight, -stdv, stdv)
```

```
        nn.init.zeros_(self.hidden2out.bias)

        for cell in self.cell_stack:
            cell.reset_parameters()

    def init_hidden(self):
        # TODO =======================
        # initialize the hidden states to zero
        """
        This is used for the first mini-batch in an epoch, only.
        """

        # a parameter tensor of shape (self.num_layers, self.batch_size, self.hidden_size)
        if torch.cuda.is_available():
            device = torch.device("cuda")
        else:
            device = torch.device("cpu")
        return torch.zeros(self.num_layers, self.batch_size, self.hidden_size, device=devic
e)

    def forward(self, inputs, hidden, h_grad=None):
        # TODO =======================
        # Compute the forward pass, using a nested python for loops.
        # The outer for loop should iterate over timesteps, and the
        # inner for loop should iterate over hidden layers of the stack.
        #
        # Within these for loops, use the parameter tensors and/or nn.modules you
        # created in __init__ to compute the recurrent updates according to the
        # equations provided in the .tex of the assignment.
        #
        # Note that those equations are for a single hidden-layer RNN, not a stacked
        # RNN. For a stacked RNN, the hidden states of the l-th layer are used as
        # inputs to to the {l+1}-st layer (taking the place of the input sequence).
        """
        Arguments:
            - inputs: A mini-batch of input sequences, composed of integers that
                        represent the index of the current token(s) in the vocabulary.
                            shape: (seq_len, batch_size)
            - hidden: The initial hidden states for every layer of the stacked RNN.
                            shape: (num_layers, batch_size, hidden_size)

        Returns:
            - Logits for the softmax over output tokens at every time-step.
                  **Do NOT apply softmax to the outputs!**
                  Pytorch's CrossEntropyLoss function (applied in ptb-lm.py) does
                  this computation implicitly.
                        shape: (seq_len, batch_size, vocab_size)
            - The final hidden states for every layer of the stacked RNN.
                  These will be used as the initial hidden states for all the
                  mini-batches in an epoch, except for the first, where the return
                  value of self.init_hidden will be used.
                  See the repackage_hiddens function in ptb-lm.py for more details,
                  if you are curious.
                        shape: (num_layers, batch_size, hidden_size)
        """

        if torch.cuda.is_available():
            device = torch.device("cuda")
        else:
            device = torch.device("cpu")

        # Pass the inputs into embedding
        inputs = self.emb(inputs)
        # inputs [seq_len, batch_size, input_feature]
        inputs = self.dropout_layer(inputs)

        layer_output = torch.zeros(
            self.seq_len, self.batch_size, self.hidden_size, device=device)

        layers_hidden = []

        # The right way should be ticks through each layer and then pass it into next laye
r.
        # in the end we will reach output layers.

        h_lists = []
```

```
        for layer_idx, cell in enumerate(self.cell_stack):
            hx_layer = hidden[layer_idx]

            if layer_idx == 0:
                layer_output, hidden_state = cell(
                    inputs, hx_layer)
            else:
                layer_output, hidden_state = cell(
                    layer_output, hx_layer)

            h_lists.append(layer_output)
            layer_output = self.dropout_layer(layer_output)
            layers_hidden.append(hidden_state)

        hidden = torch.stack(layers_hidden, 0)
        logits = self.hidden2out(layer_output)

        if h_grad:
            return logits.view(self.seq_len, self.batch_size, self.vocab_size), hidden, h_l
ists
        else:
            return logits.view(self.seq_len, self.batch_size, self.vocab_size), hidden

    def generate(self, input, hidden, generated_seq_len):
        # TODO ========================
        # Compute the forward pass, as in the self.forward method (above).
        # You'll probably want to copy substantial portions of that code here.
        #
        # We "seed" the generation by providing the first inputs.
        # Subsequent inputs are generated by sampling from the output distribution,
        # as described in the tex (Problem 5.3)
        # Unlike for self.forward, you WILL need to apply the softmax activation
        # function here in order to compute the parameters of the categorical
        # distributions to be sampled from at each time-step.
        """
        Arguments:
            - input: A mini-batch of input tokens (NOT sequences!)
                            shape: (batch_size)
            - hidden: The initial hidden states for every layer of the stacked RNN.
                            shape: (num_layers, batch_size, hidden_size)
            - generated_seq_len: The length of the sequence to generate.
                           Note that this can be different than the length used
                           for training (self.seq_len)
        Returns:
            - Sampled sequences of tokens
                        shape: (generated_seq_len, batch_size)
        """

        input = self.emb(input)
        input.unsqueeze_(0)

        generated_words = []

        with torch.no_grad():
            for i in range(generated_seq_len):
                for layer_idx, cell in enumerate(self.cell_stack):
                    hx_layer = hidden[layer_idx]
                    if layer_idx == 0:
                        layer_output, hidden_state = cell(
                            input, hx_layer)
                    else:
                        layer_output, hidden_state = cell(
                            layer_output, hx_layer)

                    hidden[layer_idx] = hidden_state

                logits = self.hidden2out(hidden_state)
                logits = self.softmax(logits)
                words = torch.multinomial(logits, 1).squeeze()
                generated_words.append(words)

                input = self.emb(words)
                input.unsqueeze_(0)

        return torch.transpose(torch.stack(generated_words, 0), 0, 1)
```

GRU

In [ ]:
```
class GRU(nn.Module):  # Implement a stacked GRU RNN
    """
    Follow the same instructions as for RNN (above), but use the equations for
    GRU, not Vanilla RNN.
    """

    def __init__(self, emb_size, hidden_size, seq_len, batch_size, vocab_size, num_layers,
dp_keep_prob):
        super(GRU, 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.emb = nn.Embedding(vocab_size, emb_size)
        self.cell_stack = nn.ModuleList([])
        for layer in range(num_layers):
            layer_input_size = emb_size if layer == 0 else hidden_size
            self.cell_stack.append(GRUCell(input_size=layer_input_size,
                                           hidden_size=hidden_size))

        self.hidden2out = nn.Linear(hidden_size, vocab_size)

        self.dropout_layer = nn.Dropout(1-dp_keep_prob)
        self.tanh = nn.Tanh()
        self.init_weights_uniform()
        self.softmax = nn.Softmax(1)

    def init_weights_uniform(self):
        # TODO ========================
        stdv = 0.1
        nn.init.uniform_(self.emb.weight, -stdv, stdv)
        nn.init.uniform_(self.hidden2out.weight, -stdv, stdv)
        nn.init.zeros_(self.hidden2out.bias)

        for cell in self.cell_stack:
            cell.reset_parameters()

    def init_hidden(self):
        # TODO ========================
        # a parameter tensor of shape (self.num_layers, self.batch_size, self.hidden_size)
        if torch.cuda.is_available():
            device = torch.device("cuda")
        else:
            device = torch.device("cpu")
        return torch.zeros(self.num_layers, self.batch_size, self.hidden_size, device=devic
e)

    def forward(self, inputs, hidden, h_grad=None):

        if torch.cuda.is_available():
            device = torch.device("cuda")
        else:
            device = torch.device("cpu")

        # Pass the inputs into embedding and then
        inputs = self.emb(inputs)
        #inputs [seq_len, batch_size, input_feature]
        inputs = self.dropout_layer(inputs)

        layer_output = torch.zeros(
            self.seq_len, self.batch_size, self.hidden_size, device=device)

        layers_hidden = []
        h_lists = []

        # The right way should be ticks through each layer and then pass it into next laye
r.
        # in the end we will reach output layers.

        for layer_idx, cell in enumerate(self.cell_stack):
            hx_layer = hidden[layer_idx]
```

```
            if layer_idx == 0:
                layer_output, hidden_state = cell(
                    inputs, hx_layer)
            else:
                layer_output, hidden_state = cell(
                    layer_output, hx_layer)

            h_lists.append(layer_output)
            layer_output = self.dropout_layer(layer_output)
            layers_hidden.append(hidden_state)

        hidden = torch.stack(layers_hidden, 0)
        logits = self.hidden2out(layer_output)

        # the return hidden status is  stack all layers hidden state
        if h_grad:
            return logits.view(self.seq_len, self.batch_size, self.vocab_size), hidden, h_l
ists
        else:
            return logits.view(self.seq_len, self.batch_size, self.vocab_size), hidden

    def generate(self, input, hidden, generated_seq_len):

        input = self.emb(input)
        input.unsqueeze_(0)

        generated_words = []

        with torch.no_grad():
            for i in range(generated_seq_len):
                for layer_idx, cell in enumerate(self.cell_stack):
                    hx_layer = hidden[layer_idx]
                    if layer_idx == 0:
                        layer_output, hidden_state = cell(
                            input, hx_layer)
                    else:
                        layer_output, hidden_state = cell(
                            layer_output, hx_layer)

                    hidden[layer_idx] = hidden_state

                logits = self.hidden2out(hidden_state)
                logits = self.softmax(logits)
                words = torch.multinomial(logits, 1).squeeze()
                generated_words.append(words)

                input = self.emb(words)
                input.unsqueeze_(0)

        return torch.transpose(torch.stack(generated_words, 0), 0, 1)


class GRUCell(nn.Module):
    '''
    a basic GRU cell,
    '''

    def __init__(self, input_size, hidden_size):
        """
        Most parts are copied from torch.nn.GRUCell.
        """

        super(GRUCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        self.weight_ih = nn.Parameter(
            torch.Tensor(input_size, 3 * hidden_size))
        self.weight_hh = nn.Parameter(
            torch.Tensor(hidden_size, 3 * hidden_size))
        self.bias = nn.Parameter(torch.Tensor(3 * hidden_size))

        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

        self.reset_parameters()
```

```
    def reset_parameters(self):
        """
        Initialize parameters following the way proposed in the paper.
        """
        stdv = 1.0 / math.sqrt(self.hidden_size)
        for weight in self.parameters():
            init.uniform_(weight, -stdv, stdv)

    def forward(self, inputs, hx):
        """
        Args:
            inputs: A (seq_len, , batch ,input_size) tensor containing input
                features.
            hx: the initial hidden

        Returns:
            outputs: layers outputs
            hx: Tensors containing the next hidden.
        """
        # cell ticks trough seq len, and then return the output and  cell states
        batch_size = hx.size(0)
        bias_batch = self.bias.unsqueeze(0).expand(
            batch_size, self.bias.size(0))

        max_time = inputs.size(0)
        outputs = []
        for time in range(max_time):
            # concat input to save one matrix operation
            input_x = inputs[time]

            xw = torch.addmm(bias_batch, input_x, self.weight_ih)
            hu = torch.mm(hx, self.weight_hh)
            xw2 = torch.split(xw, self.hidden_size, 1)
            hu2 = torch.split(hu, self.hidden_size, 1)

            z = self.sigmoid(xw2[0] + hu2[0])
            r = self.sigmoid(xw2[1] + hu2[1])
            hx_ = self.tanh(r * hu2[2] + xw2[2])

            hx = (1 - z) * hx_ + z * hx

            outputs.append(hx)
            # # pass the hidden status to next tick

        outputs = torch.stack(outputs, 0)
        return outputs, hx


'''
End for GRU_Cell
'''
```

Attention module

In [ ]:

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, n_heads, n_units, dropout=0.1):
        """
        n_heads: the number of attention heads
        n_units: the number of output units
        dropout: probability of DROPPING units
        """
        super(MultiHeadedAttention, self).__init__()
        # This sets the size of the keys, values, and queries (self.d_k) to all
        # be equal to the number of output units divided by the number of heads.
        self.d_k = n_units // n_heads
        # This requires the number of n_heads to evenly divide n_units.
        assert n_units % n_heads == 0, '{} heads are not evenly dividable by {} units'. for
mat(
            n_heads, n_units)
        self.n_units = n_units
        self.n_heads = n_heads
        self.dropout = dropout

        # TODO: create/initialize any necessary parameters or layers
        # Initialize all weights and biases uniformly in the range [-k, k],
        # where k is the square root of 1/n_units.
        # Note: the only Pytorch modules you are allowed to use are nn.Linear
        # and nn.Dropout
        # ETA: you can also use softmax

        self.q_dense_layer = nn.Linear(n_units, n_units)
        self.k_dense_layer = nn.Linear(n_units, n_units)
        self.v_dense_layer = nn.Linear(n_units, n_units)

        self.output_dense_layer = nn.Linear(n_units, n_units)
        self.softmax = nn.Softmax(dim=-1)
        self.attention_dropout = nn.Dropout(dropout)
        self.init_weights_uniform()

    def init_weights_uniform(self):
        # TODO ========================
        stdv = 1.0 / math.sqrt(self.n_units)
        nn.init.uniform_(self.q_dense_layer.weight, -stdv, stdv)
        nn.init.uniform_(self.k_dense_layer.weight, -stdv, stdv)
        nn.init.uniform_(self.v_dense_layer.weight, -stdv, stdv)
        nn.init.uniform_(self.output_dense_layer.weight, -stdv, stdv)

        nn.init.uniform_(self.q_dense_layer.bias, -stdv, stdv)
        nn.init.uniform_(self.k_dense_layer.bias, -stdv, stdv)
        nn.init.uniform_(self.v_dense_layer.bias, -stdv, stdv)
        nn.init.uniform_(self.output_dense_layer.bias, -stdv, stdv)

    def forward(self, query, key, value, mask=None):
        # TODO: implement the masked multi-head attention.
        # query, key, and value all have size: (batch_size, seq_len, self.n_units, self.d_
k)
        # mask has size: (batch_size, seq_len, seq_len)
        # As described in the .tex, apply input masking to the softmax
        # generating the "attention values" (i.e. A_i in the .tex)
        # Also apply dropout to the attention values.

        # Codes are mostly copied from attention module in tensorflow

        # q k v  -> [batch_size, seq_len, self.n_units]
        q = self.q_dense_layer(query)
        k = self.k_dense_layer(key)
        v = self.v_dense_layer(value)

        # split q, k, v into diffrent heads and transpose the resulting value
        # [batch_size, seq_len, self.n_units] -> [batch_size, seq_len, depths]
        # and then stack into [batch_size, n_heads, seq_len, depths]
        q = torch.stack(torch.split(q, self.d_k, dim=2), dim=1)
        k = torch.stack(torch.split(k, self.d_k, dim=2), dim=1)
        v = torch.stack(torch.split(v, self.d_k, dim=2), dim=1)

        # Scale q to prevent the dot product between q and k from growing too large
        q *= self.d_k ** -0.5

        # Calculate dot product attention
        logits = torch.matmul(q, k.transpose(2, 3))
```

```
        if mask is not None:
            mask = mask.unsqueeze(1).repeat(1, self.n_heads, 1, 1).float()
            logits = (logits * mask) - 1.e9 * (1 - mask)

        # # We compute the softmax. We minus the score with a max for better numerical stab
ility.
        # # [batch_size, n_heads, seq_len, seq_len]
        # logits = torch.exp(logits - torch.max(logits, -1, keepdim=True)[0])
        # weights = logits / torch.sum(logits, dim=-1, keepdim=True)
        weights = self.softmax(logits)

        weights = self.attention_dropout(weights)
        attention_output = torch.matmul(weights, v)
        # combine heads
        # [batch_size, n_heads, seq_len, depths] -> [batch_size, seq_len, self.n_units]
        seq_len = attention_output.size(2)
        attention_output = attention_output.transpose(
            1, 2).contiguous().view(-1, seq_len, self.n_units)

        attention_output = self.output_dense_layer(attention_output)

        return attention_output  # size: (batch_size, seq_len, self.n_units)
```

Part 4:

Figures and Tables

Experiment 1: Model Comparison

- Plotting learning curves of PPL over epochs

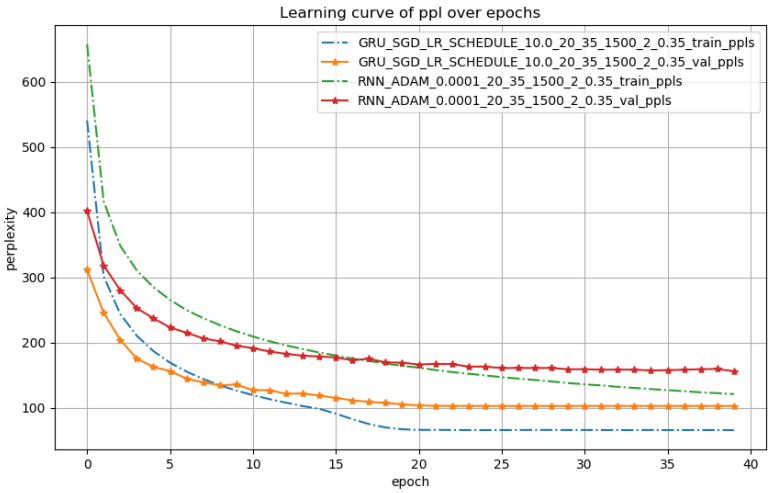


Figure 1: 4.1 - GRU and RNN

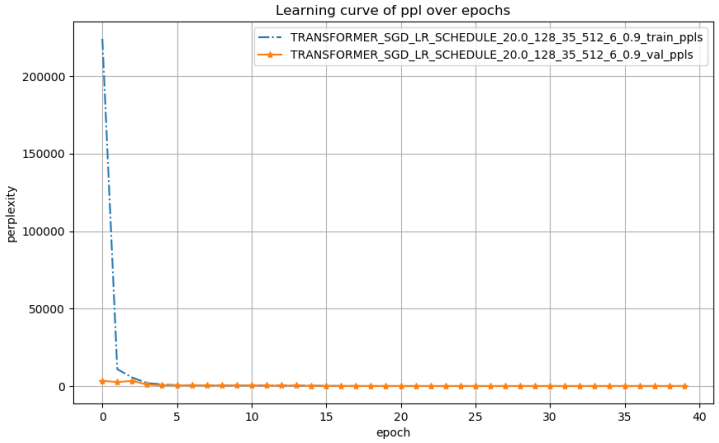


Figure 2: 4.1 - TRANSFORMER

- Plotting learning curves of PPL over wall-clock-time

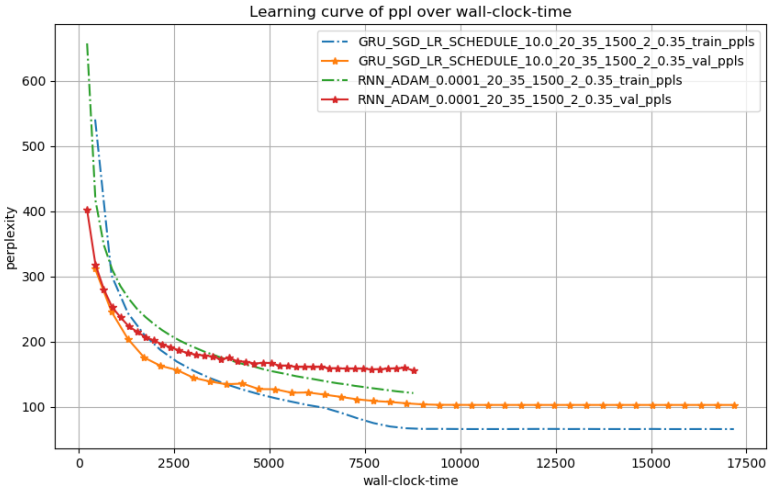


Figure 3: 4.1 - GRU and RNN

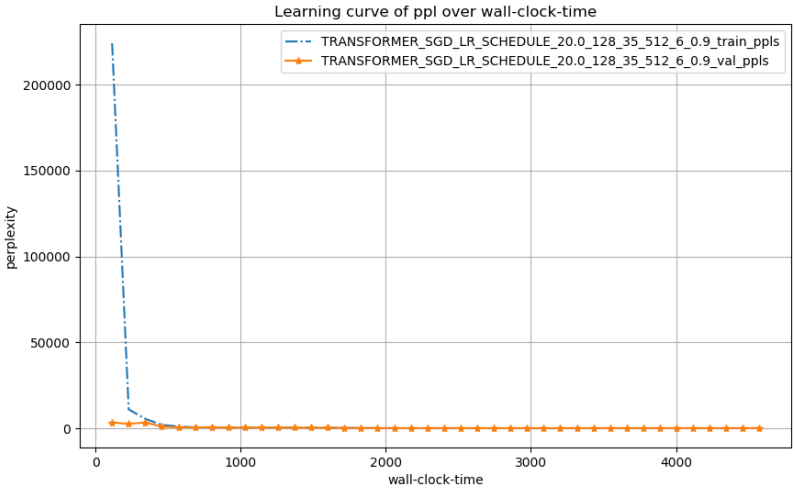


Figure 4: 4.1 - TRANSFORMER

From the figure, we found that the performance of GRU is better than RNN, the trend is gently decreasing, comparing with Transformer, it decreased rapidly from over 200000 to under 200. By comparing achieving the similar ppl, Transformer costs less time than GRU and RNN. RNN converges faster than GRU.

Figure 5: 4.1 - Table on model comparison

Architecture	Optimizer	Experiment	initial_lr	batch_size	seq_len	hidden_size	num_layers	dp_keep_prob	train_ppl	val_ppl	best_val_ppl
GRU	SGD_LR_SCHEDULE	4.1	10	20	35	1500	2	0.35	65.80590736	102.8590816	102.8590559
RNN	ADAM	4.1	0.0001	20	35	1500	2	0.35	121.0827639	156.0032039	156.0032039
TRANSFORMER	SGD_LR_SCHEDULE	4.1	20	128	35	512	6	0.9	67.19882666	144.8866225	144.8866052

Experiment 2: Exploration of optimizers

- Plotting learning curves of PPL over epochs

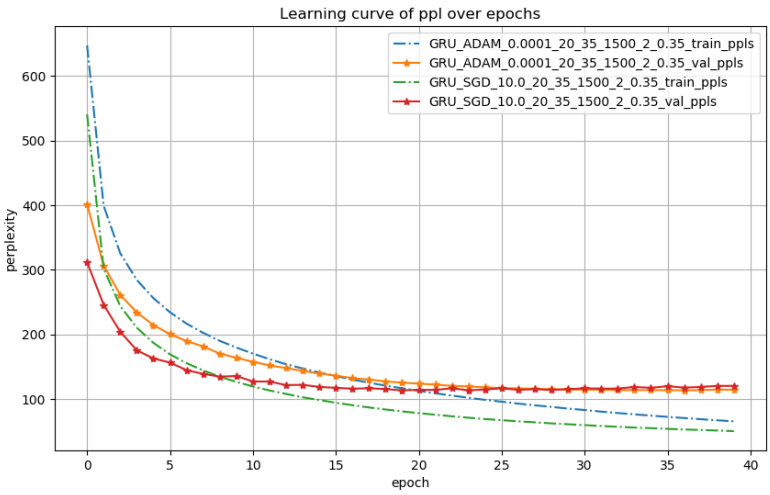


Figure 6: 4.2 - GRU_ADAM and GRU_SGD

GRU with two optimizers: ADAM and SGD, after 40 epochs, it converges to similar ppl. The trend of decreasing is gentle.

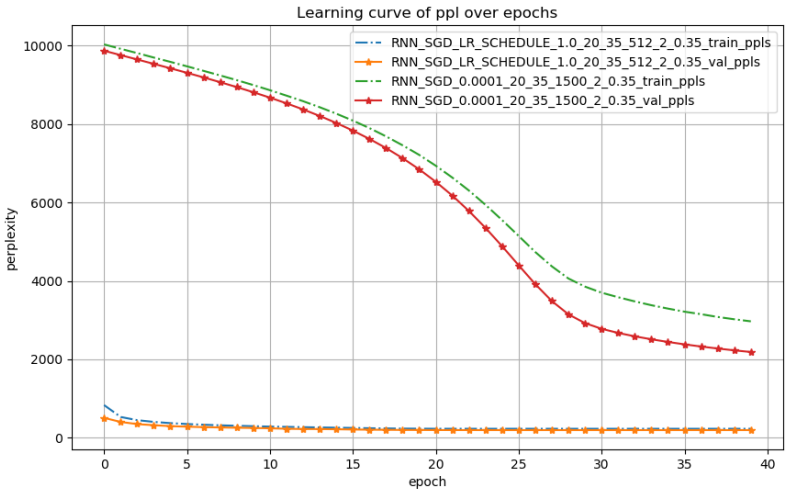


Figure 7: 4.2 - RNN_SGD_LR and RNN_SGD

RNN with two optimizers: SGD_LR_SCHEDULE and SGD, within 40 epochs, the ppl with SGD_LR_SCHEDULE is almost flat, but with SGD, the trend is decreasing obviously. After 40 epochs, the ppl with SGD still over 2000, but with SGD_LR_SCHEDULE, the ppl converges to 200 just after several epochs.

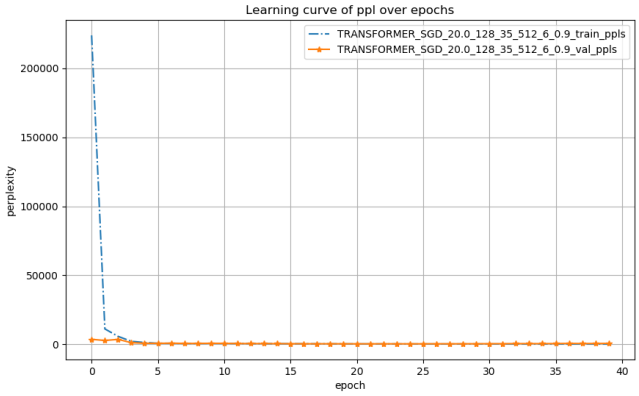


Figure 8: 4.2 - TRANSFORMER_SGD

Transformer with SGD, the train ppl is dropping rapidly, and validation ppl almost no change at all.

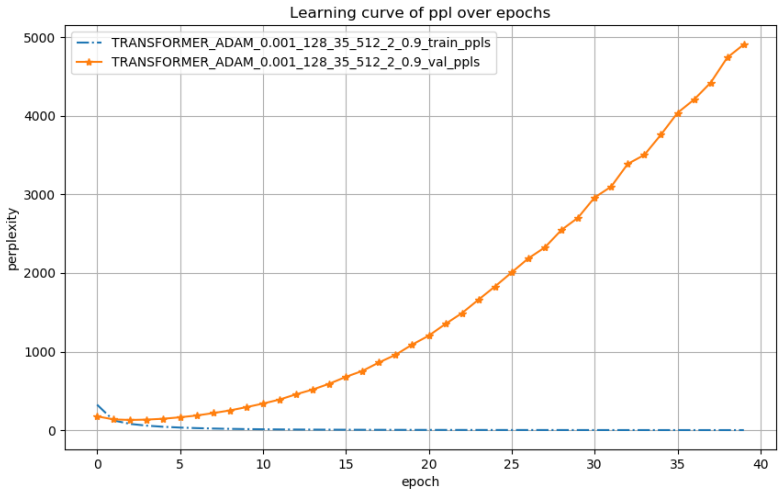


Figure 9: 4.2 - TRANSFORMER_ADAM

But with ADAM, the train ppl decreases very slowly with validation ppl increases rapidly, it is overfitting after just several epochs.

- Plotting learning curves of PPL over wall-clock-time

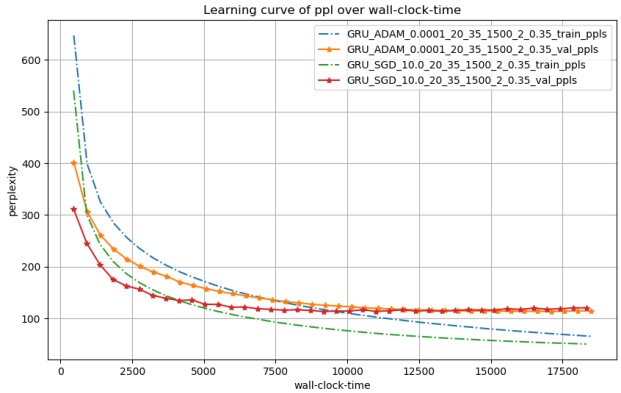


Figure 10: 4.2 - GRU_ADAM and GRU_SGD

GRU with ADAM and SGD, the wall-clock-time spends on them are closed.

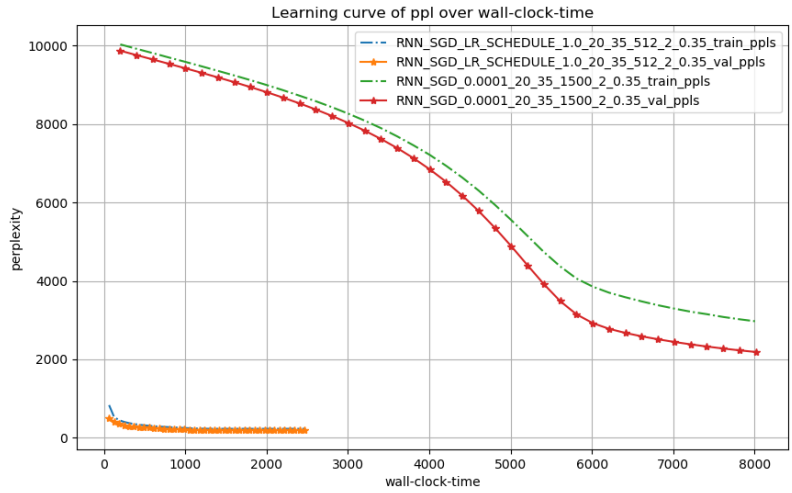


Figure 11: 4.2 - RNN_SGD_LR and RNN_SGD

RNN with SGD_LR_SCHEDULE costs less time than with SGD.

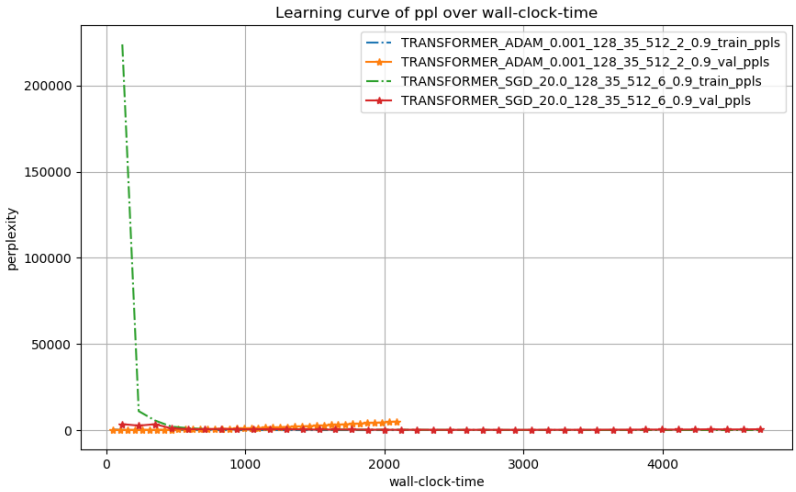


Figure 12: 4.2 - TRANSFORMER_ADAM and TRANSFORMER_SGD

Transformer with ADAM costs less time but with overfitting.

Figure 13: 4.2 - Table on exploration of optimizers

Architecture	Optimizer	Experiment	initial_lr	batch_size	seq_len	hidden_size	num_layers	dp_keep_prob	train_ppl	val_ppl	best_val_ppl
GRU	SGD	4.2	10	20	35	1500	2	0.35	50.31130249	120.3205777	113.3449837
GRU	ADAM	4.2	0.0001	20	35	1500	2	0.35	65.37838478	114.3267253	113.5038611
RNN	SGD_LR_SCHEDULE	4.2	1	20	35	512	2	0.35	230.3556716	196.4223734	196.4201496
RNN	SGD	4.2	0.0001	20	35	1500	2	0.35	2969.676316	2188.215742	2188.215742
TRANSFORMER	SGD	4.2	20	128	35	512	6	0.9	18.87950156	499.6915094	171.4651796
TRANSFORMER	ADAM	4.2	0.001	128	35	512	2	0.9	3.162253657	4907.329036	132.1678284

From above figures and table, we found that the performance of GRU on ADAM and SGD are better than the other two. RNN with SGD_LR_SCHEDULE is better than SGD. And RNN with SGD, Transformer with SGD and ADAM are not satisfactory.

Experiment 3: Exploration of hyperparameters

- Plotting learning curves of PPL over epochs

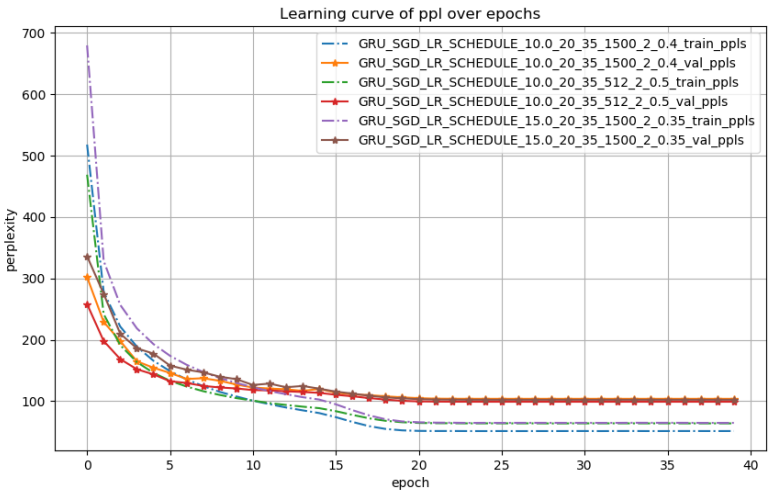


Figure 14: 4.3 - Explore on GRU

GRU with SGD_LR_SCHEDULE, the performance are closed with different hyperparameters. The best one is GRU with SGD_LR_SCHEDULE, initial_lr=10, batch_size=20, seq_len=35, hidden_size=512, num_layers=2 and dp_keep_prob=0.5.

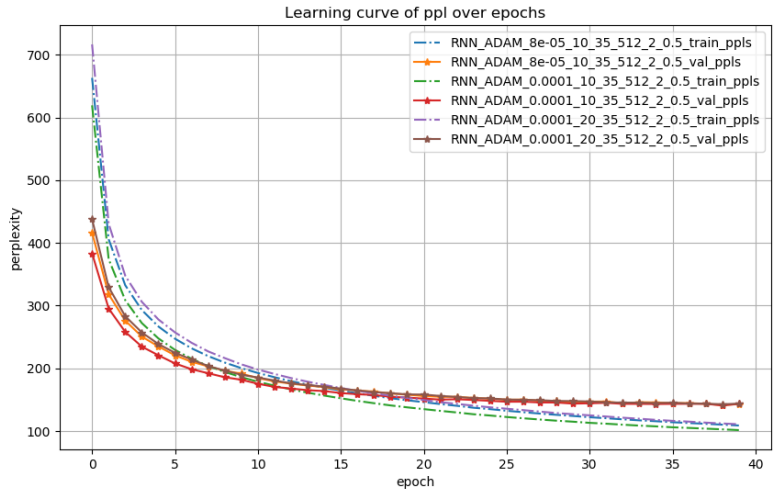


Figure 15: 4.3 - Explore on RNN

Performance of RNN are closed, the ppl drops down smoothly, the best one is RNN with ADAM, initial_lr=0.0001, batch_size=10, seq_len=35, hidden_size=512, num_layers=2, dp_keep_prob=0.5.

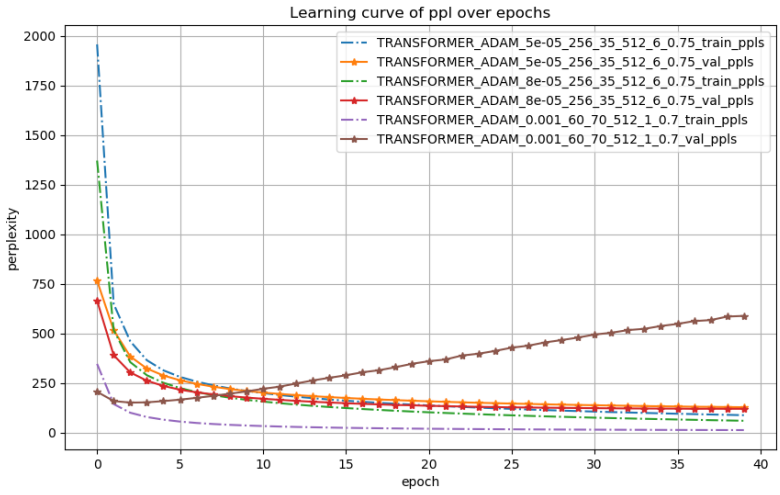


Figure 16: 4.3 - Explore on TRANSFORMER

From the figure, we found that with initial_lr=0.001, it's easy to overfitting. The best one is TRANSFORMER with ADAM, initial_lr=0.00008, batch_size=256, seq_len=35, hidden_size=512, num_layers=6, dp_keep_prob=0.75.

- Plotting learning curves of PPL over wall-clock-time

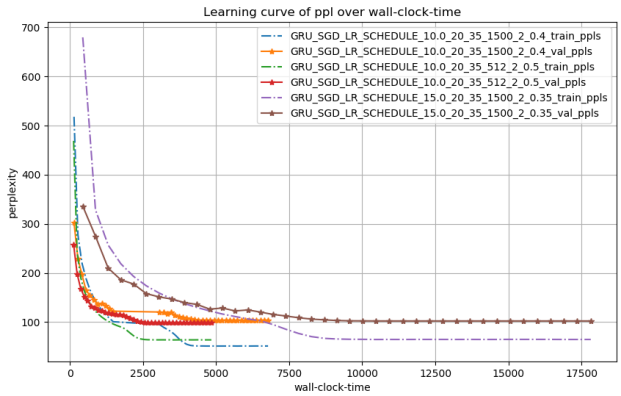


Figure 17: 4.3 - Explore on GRU

From the figure, we found that with smaller hidden_size and initial_lr, time costs less.

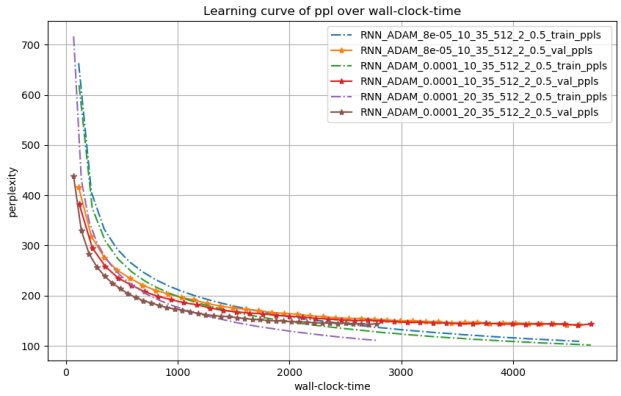


Figure 18: 4.3 - Explore on RNN

For RNN, the effect of RNN+ Adam combination is significantly better than that of RNN+SDG and RNN+SGD_LR_SCHEDULE, so our main efforts are spent on the combinations of hyperparameters of RNN+ Adam.

We change the difference batch_size from 20 to 100 and reduce to 10, the keep_probability of dropout from 0.35 to 0.5 to 0.9.The most significant change is that when we reduce the hidden-size from 1500 to 512, the PPL is nearly 15% better than before. Figure 36 and Figure 37 shows all the results .
From the figure, we found that with smaller batch_size, time costs less.

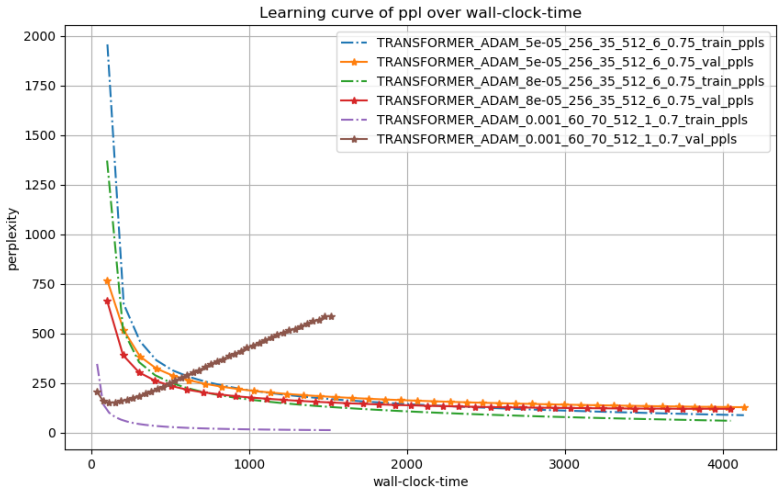


Figure 19: 4.3 - Explore on TRANSFORMER

For TRANSFORMER, we believe that its capacity is larger than the other two, so overfitting will occur very soon. Maybe overfitting will occur within 5 epoche. Of the three different optimizations, TRANSFORMER+ADAM has the best performance and TRANSFORMER +SGD has the worst, so we start with the TRANSFORMER +ADAM combination, and TRANSFORMER+$SGD\_LR\_SCHEDULE$. Therefore, at the beginning of designing combinations of hyperparameters, we want to reduce the model's capacity a little bit, so that it will not overfit too quickly. We reduced the number of hidden layers from 6 to 4 to 2 or 1, and the *keep_probability* of dropout from 0.9 to 0.7 to 0.5 to 0.35. Then we tried to changer the initial learning rate ,and sequence of length, the batch size .Overfitting was alleviated, but except the parameters keep_probability of dropout , the PPL was not improved when we changed the others parameters alone. Until we fine this combination ($--model = TRANSFORMER --optimizer = ADAM --initial_lr = 0.00008 --batch\_size = 256 --seq\_len = 35 --hidden\_size = 512 --num\_layers = 6 --dp\_keep\_prob = .75$) Figure 40 and Figure 41 shows all the results .

From the figure, we found that with smaller num_layer, time costs less.

Figure 20: 4.3 - Table on exploration of hyperparmeters

Architecture	Optimizer	Experiment	initial_lr	batch_size	seq_len	hidden_size	num_layers	dp_keep_prob	train_ppl	val_ppl	best_val_ppl
GRU	SGD_LR_SCHEDULE	4.1	10	20	35	1500	2	0.35	65.80590736	102.8590816	102.8590559
GRU	**SGD_LR_SCHEDULE**	**4.3**	**10**	**20**	**35**	**512**	**2**	**0.5**	**63.9281046**	**99.19787277**	**99.19786782**
GRU	SGD_LR_SCHEDULE	4.3	10	20	35	1500	2	0.4	51.49325466	103.964292	103.9642601
GRU	SGD_LR_SCHEDULE	4.3	15	20	35	1500	2	0.35	64.80504701	102.418542	102.4185253
GRU	SGD	4.2	10	20	35	1500	2	0.35	50.31130249	120.3205777	113.3449837
GRU	ADAM	4.2	0.0001	20	35	1500	2	0.35	65.37838478	114.3267253	113.5038611
RNN	SGD_LR_SCHEDULE	4.2	1	20	35	512	2	0.35	230.3556716	196.4223734	196.4201496
RNN	SGD	4.2	0.0001	20	35	1500	2	0.35	2969.676316	2188.215742	2188.215742
RNN	ADAM	4.1	0.0001	20	35	1500	2	0.35	121.0827639	156.0032039	156.0032039
RNN	**ADAM**	**4.3**	**0.0001**	**10**	**35**	**512**	**2**	**0.5**	**101.5511325**	**143.4872468**	**140.8607297**
RNN	ADAM	4.3	0.00008	10	35	512	2	0.5	108.898234	142.9521314	141.5031832
RNN	ADAM	4.3	0.0001	20	35	512	2	0.5	110.9362547	143.7223325	142.7659456
TRANSFORMER	SGD_LR_SCHEDULE	4.1	20	128	35	512	6	0.9	67.19882666	144.8866225	144.8866052
TRANSFORMER	SGD	4.2	20	128	35	512	6	0.9	18.87950156	499.6915094	171.4651796
TRANSFORMER	ADAM	4.2	0.001	128	35	512	2	0.9	3.163253657	4907.329036	132.1678284
TRANSFORMER	**ADAM**	**4.3**	**0.00008**	**256**	**35**	**512**	**6**	**0.75**	**60.64668847**	**120.7858812**	**120.7104478**
TRANSFORMER	ADAM	4.3	0.00005	256	35	512	6	0.75	88.67532096	128.8374922	128.8374922
TRANSFORMER	ADAM	4.3	0.001	60	70	512	1	0.7	13.56421404	588.8361861	152.0967719

From above table, we found the best performance is with GRU, then is Transformer, and then is RNN.

4.4 Make 2 plots for each optimizer; one which has all of the validation curves for that optimizer over epochs and one overwall-clock-time.

- Validation curves of PPL over epochs

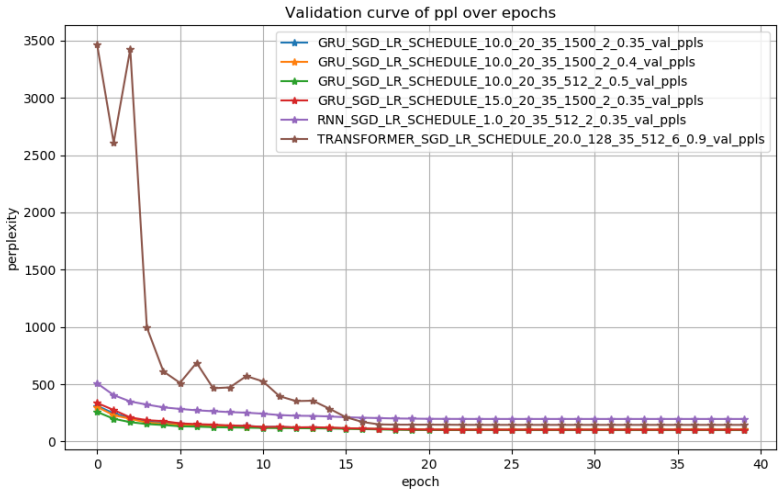


Figure 21: 4.4 - Optimizer: SGD_LR_SCHEDULE

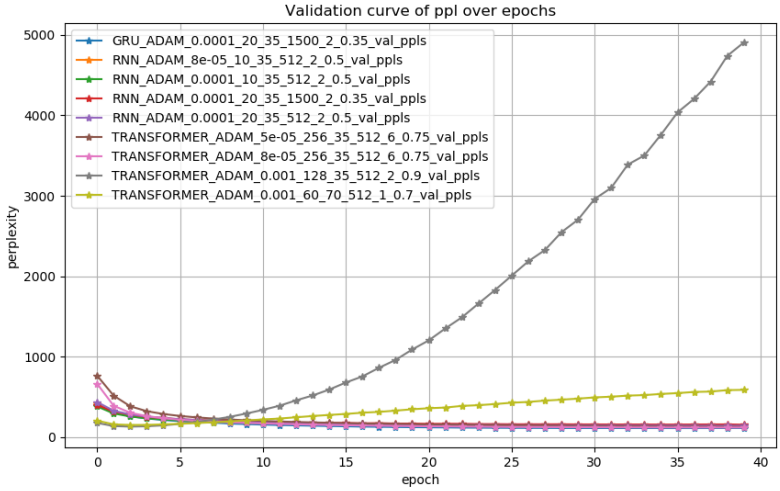


Figure 22: 4.4 - Optimizer: ADAM

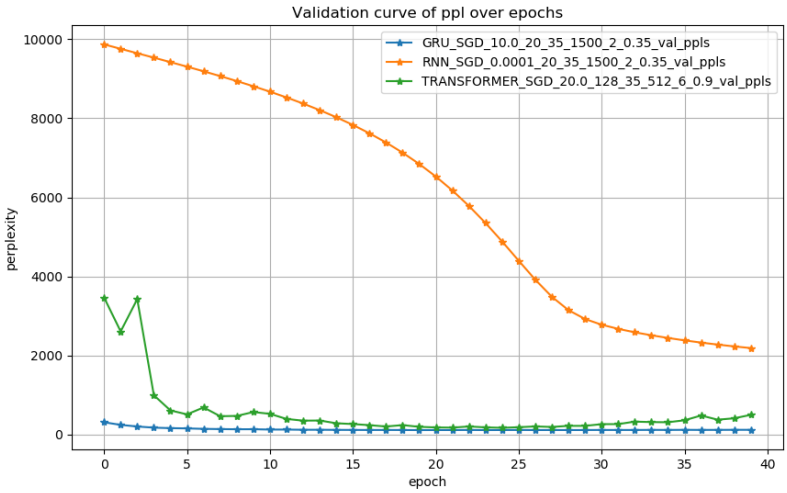


Figure 23: 4.4 - Optimizer: SGD

From above figures, with optimzer ADAM, the curves are more smooth instead of jumping up and down with Transformer.

- Validation curves of PPL over wall-clock-time

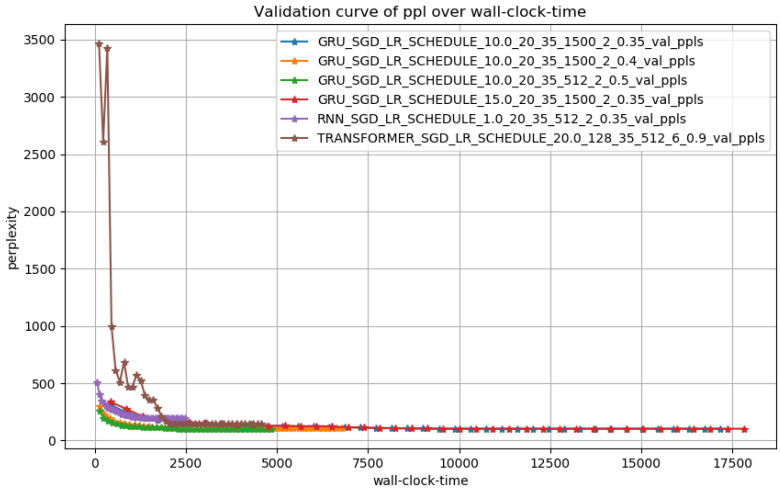


Figure 24: 4.4 - Optimizer: SGD_LR_SCHEDULE

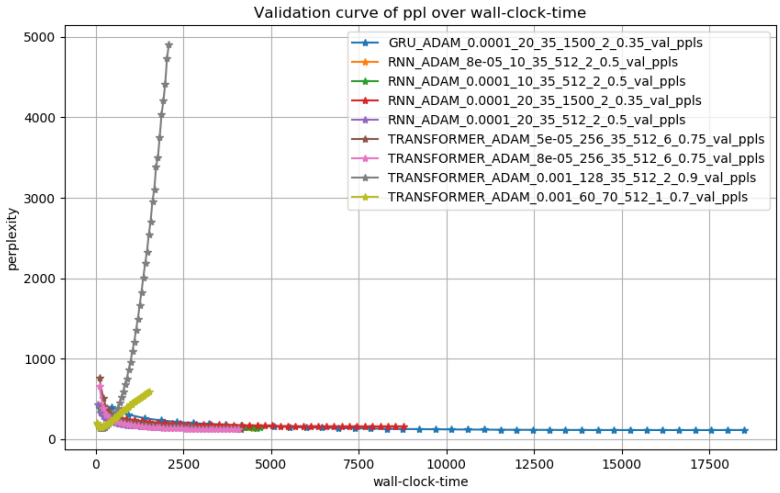


Figure 25: 4.4 - Optimizer: ADAM

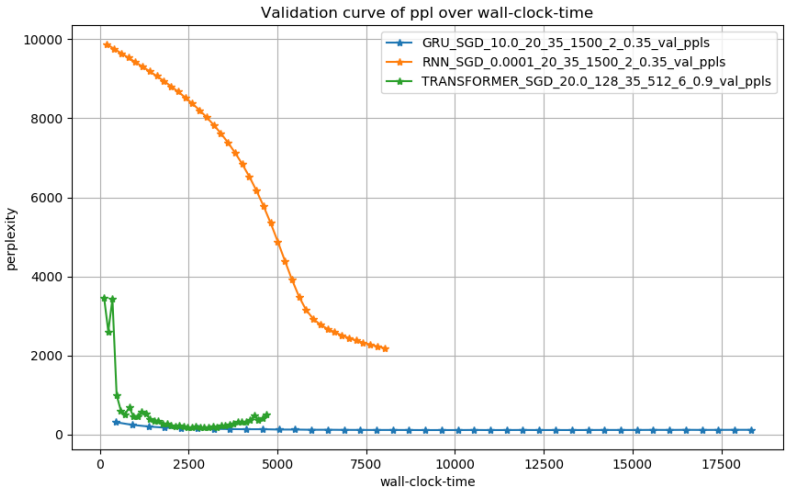


Figure 26: 4.4 - Optimizer: SGD

With above figures, we found that with these three optimizers, GRU costs more time, then is RNN, Transformer costs less.

4.5 Make 2 plots for each arcitecture; one which has all of the validation curves for that architec-ture overepochsand one overwall-clock-time.

- Validation curves of PPL over epochs

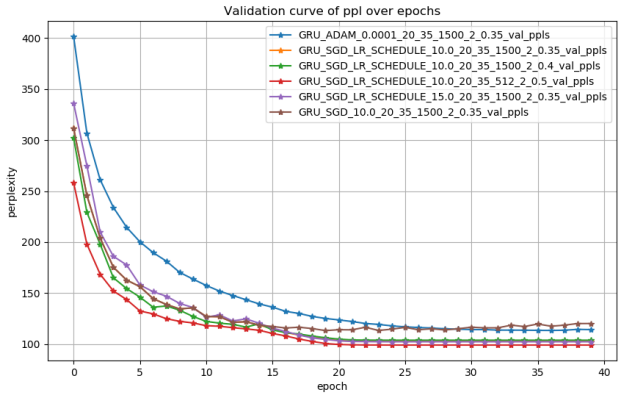


Figure 27: 4.5 - Architecture: GRU

Within three configuration, under GRU architecture, we found with optimizer SGD_LR_SCHEDULE, the performance is best.

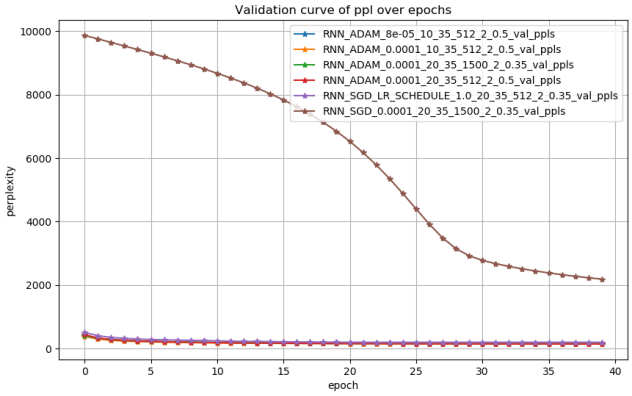


Figure 28: 4.5 - Architecture: RNN

With RNN, best performance is based on ADAM optimizer.

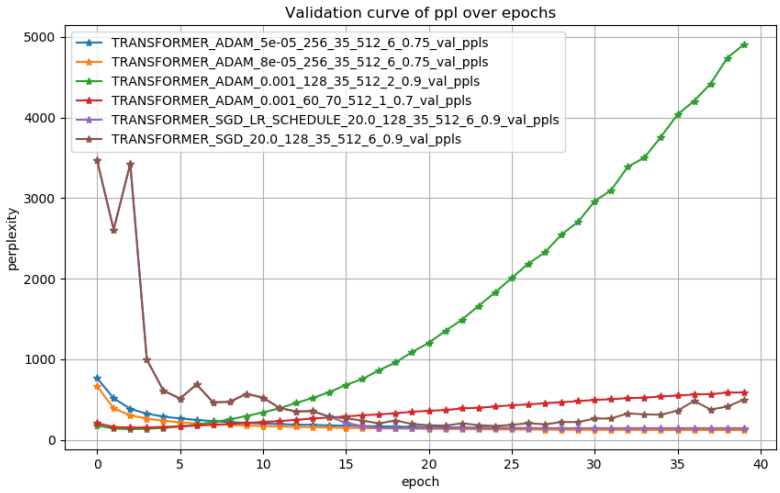


Figure 29: 4.5 - Architecture: TRANSFORMER

With Transformer, best performance is based on ADAM optimizer.

- Validation curves of PPL over wall-clock-time

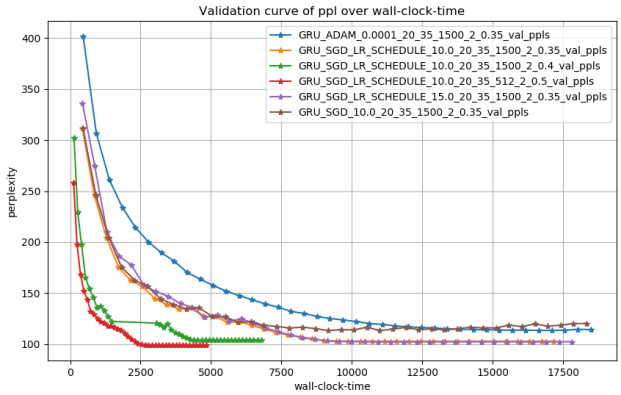


Figure 30: 4.5 - Architecture: GRU

Within three configuration, under GRU architecture, we found with optimizer SGD_LR_SCHEDULE, the time cost is less with best performance.

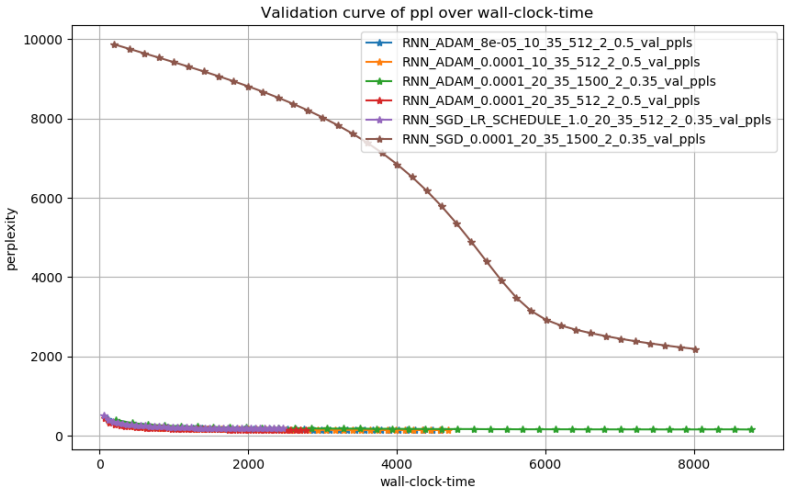


Figure 31: 4.5 - Architecture: RNN

With RNN architecture, we found with optimizer ADAM, the time cost is less with best performance.

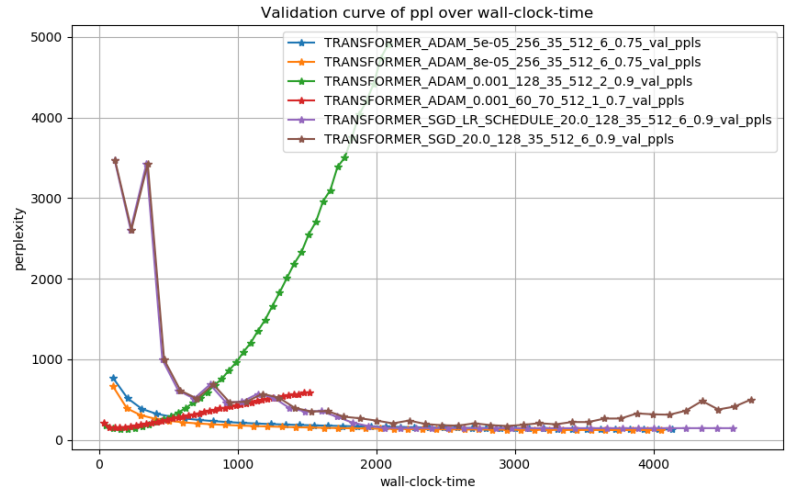


Figure 32: 4.5 - Architecture: TRANSFORMER

With TRANSFORMER architecture, we found with optimizer ADAM, the time cost is less.

Conclustion & Discussion

Based on above three experiments, with the result we got from validation_set, the best one is: GRU+SGD_LR_SCHEDULE, with hyperparameters: initial_lr=10, batch_size=20, seq_len=35, hidden_size=512, num_layers=2, dp_keep_prob=0.5. But we still need test_set to verify the conclusion.

From complexity, the ordering is: Transformer > GRU > RNN, but from the result, we found that Transformer is easier and quicker to overfitting because of high variance.

Based on wall-clock-time, from Figure 24 to Figure 26, we could know the ordering of costing time: RNN > GRU > Transformer, which means Transformer costs less time.

Based on figures from experiment 4.4, we get below ranking:

Optimizer	Perfromance Ranking		
SGD LR SCHEDULE	GRU	Transformer	RNN
ADAM	GRU	Transformer	RNN
SGD	GRU	Transformer	RNN

Based on figures from experiment 4.5, we get below ranking:

architecture	Perfromance Ranking		
GRU	SGD LR SCHEDULE	ADAM	≈SGD
RNN	ADAM	SGD LR SCHEDULE	SGD
Transformer	ADAM	SGD LR SCHEDULE	SGD

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

  We expect to see which experiment performs best and with which configuration. Generally,we believe that ADAM is better than the other two optimization methods, but also is not certain. This optimization is more popular at present, so we want to see if it is really that strong . So, taking all the data together, there is no single optimization for all the architectures, all the data is good. RNN+ADAM, TRANSFORMER+ADAM, these two combinations are indeed better than the other two, but for GRU, in all our experiments, GRU+SGD_LR_SCHEDULE is the best combination. Actually we found that GRU with SGD_LR_SCHEDULE optimizer performs better than RNN and Transformer.

  The cause we thought, is for Transformer architecture, it has high capacity, but it's easy to overfitting. As for RNN architecture, the training time costs longer, and it's more sensitive for choosing different optimizers. GRU

with more appropriate capacity, with reasonable choice of parameters, it is easier to get better results.

- 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.

  Referring to the learning curves, in terms of training time, the ordering is:

architecture	Perfromance Ranking		
GRU	SGD LR SCHEDULE	ADAM	≈SGD
RNN	SGD LR SCHEDULE	ADAM	SGD
Transformer	ADAM	SGD LR SCHEDULE	SGD

  In terms of generalization performance, for these three optimizers, GRU architecture is the one that they're best for.

  Based on the experiment results, as for the optimizers, for general, we found that SGD_LR_SCHEDULE and ADAM are better than SGD.

  As for the relationship to other hyper-parameters, what we concern most is initial_lr, for GRU + SGD_LR_SCHEDULE or SGD, the initial_lr between 10 to 20 is more appropriate. But for GRU + ADAM, the initial_lr could be chosen around 0.0001.

  As for RNN + ADAM or SGD, the initial_lr around 0.0001 is more appropriate and for RNN + SGD_LR_SCHEDULE, the initial_lr could be set to larger than 1.

  For Transformer + SGD or SGD_LR_SCHEDULE, the initial_lr could be set to around 20, and Transformer + ADAM, the initial_lr value set to less than 0.001 would be more appropriate.

- Which hyperparameters+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?)

  If we most concerned with wall-clock-time, we would consider below configuration:
  TRANSFORMER, optimizer=**ADAM**, initial_lr=0.001, batch_size=60, seq_len=70, hidden_size=512, num_layers=1, dp_keep_prob=0.7

  With generalization performance, we would consider:
  GRU, optimizer=**SGD_LR_SCHEDULE**, initial_lr=10, batch_size=20, seq_len=35, hidden_size=512, num_layers=2, dp_keep_prob=0.5

By comparing below two figures, we can't tell that better generalization performance means longer training time. Since with Transformer architecture, the performance is better than RNN but it also costs less time. Also with time increased, the validation ppl is already convergence, there is not much change after several epochs. The 'cost' of the good performance is depends on the combination of hyperparameters+optimizer with different architecture. For getting the better performance, we could set early stop to get some best combination of hyperparameters+optimizers, through adjusting the combination to observe which one could get a better result.

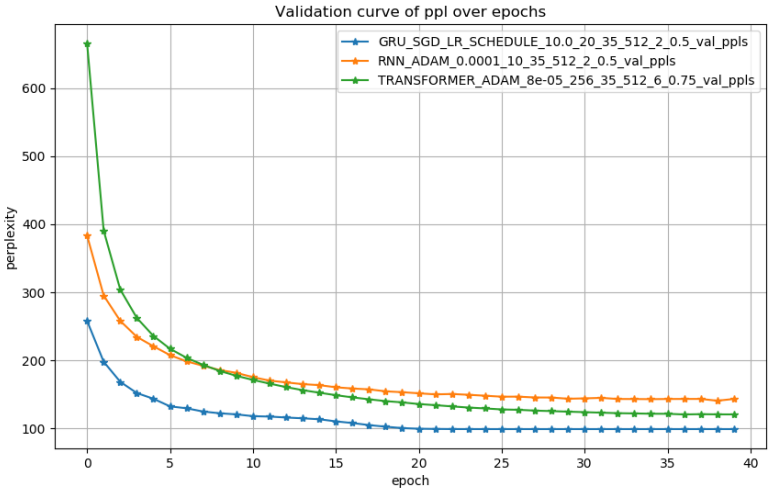


Figure 33: best performance with three architectures by epochs

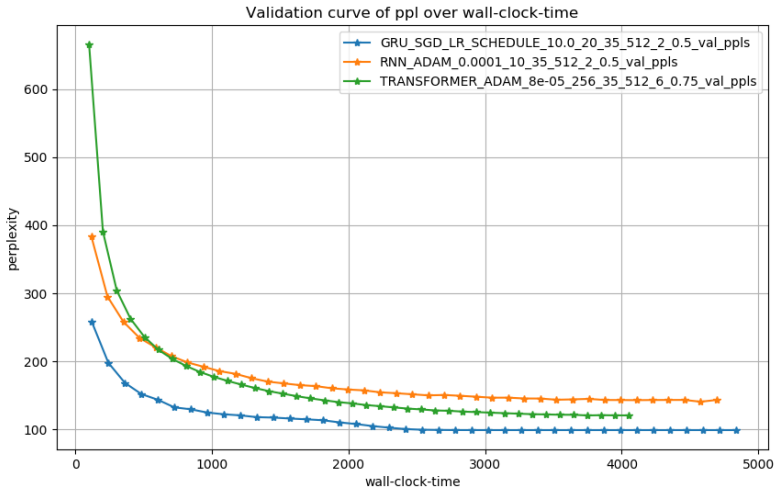


Figure 34: best performance with three architectures by wall-clock-time

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

The most 'reliable' architecture:
GRU, optimizer=**SGD_LR_SCHEDULE**, initial_lr=10, batch_size=20, seq_len=35, hidden_size=512, num_layers=2, dp_keep_prob=0.5

the more unstable:
TRANSFORMER, optimizer=**SGD**, initial_lr=20, batch_size=128, seq_len=35, hidden_size=512, num_layers=6, dp_keep_prob=.9

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

  With below configuration: GRU, optimizer=ADAM, initial_lr=0.001, batch_size=20, seq_len=35, hidden_size=512, num_layers=2, dp_keep_prob=0.6, we could have a very nice performance at the beginning. But after 12-13 epochs, it becomes overfitting.

  we explored combinations of hyperparameters to try to find settings which achieve better validation performance than those given to you in (4-1). We Report 3 best experiments per architecture , the figure 36 to the figure 41 are all the test parameters and their results, provided for reference.

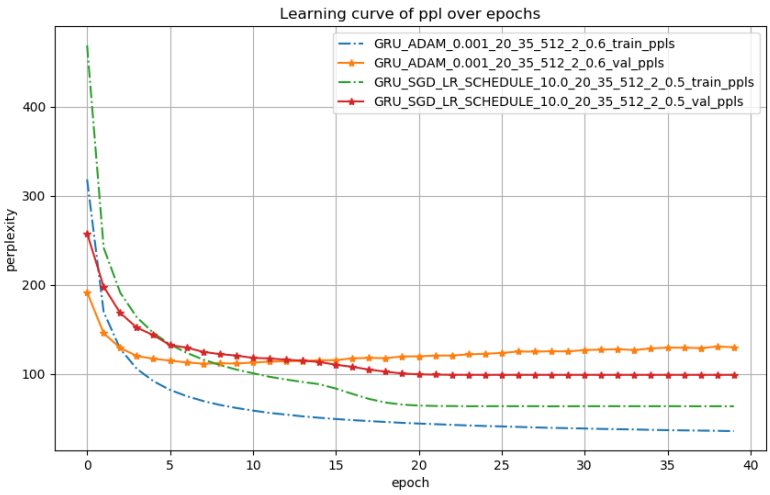


Figure 35: Compare with interested experiment and the best experiment

We tried to set dp_keep_prob=0.5, to avoid overfitting, but the effect is not obvious, the validation ppl also converges around 110. So we are curious how could avoid overfitting and get a better result with satisfied convergence effect. Maybe by using batch normalization could help.

	4.1	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3
Model	RNN	RNN	RNN	RNN	RNN	RNN	RNN	RNN	RNN
optimizer	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM
initial_lr	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
batch_size	20	100	100	20	1	20	20	20	20
seq_len	35	35	35	35	35	35	35	35	35
hidden_size	1500	1500	1500	1500	1500	1500	1500	1500	1500
num_layers	2	2	4	2	2	4	2	2	2
dp_keep_prob	0.35	0.35	0.35	0.9	0.35	0.5	0.5	0.6	0.7
epoch	40	40	40	15	16	30	40	30	30
train_ppl	121.0827639	155.6622149	179.9624926	48.41707136	300.0579431	101.866357	71.17162126	62.38785324	56.81803842
val_ppl	156.0032039	156.0440674	176.6359187	202.8409917	259.7984411	171.6648313	162.5248845	168.8254655	170.1313684
best_val_ppl/ep	156.0032039	156.0440674	176.6359187	177.6309553	258.7150717	166.8051669	151.2173569	154.0131237	156.3941263
times		101.7164323	171.1878519	403.8312368	2124.804065	112.0492268	214.4852667	214.3576643	212.6995401
	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3
Model	RNN	RNN	RNN	RNN	RNN	RNN	RNN	RNN	RNN
optimizer	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM
initial_lr	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.00005	0.0001	0.00008
batch_size	20	20	20	20	20	10	10	1	10
seq_len	35	35	35	70	10	35	35	35	35
hidden_size	1500	512	256	512	512	512	512	512	512
num_layers	2	2	2	2	2	2	2	2	2
dp_keep_prob	0.55	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
epoch	38	40	40	40	40	40	40	40	40
train_ppl	61.20114743	110.9362547	147.8105834	126.8804818	103.2985649	101.5511325	130.3204061	165.7764791	108.898234
val_ppl	170.7489358	143.7223325	153.8352442	144.7794934	146.775963	143.4872468	149.4540617	169.5522467	142.9521314
best_val_ppl/ep	152.8810004	142.7659456	153.8352442	144.7794934	145.905173	140.8607297	148.3342222	169.3716221	141.5031832
times	210.4543309	65.85677481	47.44565105	63.08164072	99.89535451	117.3742445	115.7897303	892.6915224	114.1936619

Figure 36: Exploration of hyperparmeters of RNN-1

	4.3	4.3	4.3	4.3	4.2	4.2
Model	RNN	RNN	RNN	RNN	RNN	RNN
optimizer	ADAM	ADAM	ADAM	ADAM	SGD_LR_SCI	SGD
initial_lr	1	0.001	0.001	0.001	1	0.0001
batch_size	20	20	20	20	20	20
seq_len	35	35	35	35	35	35
hidden_size	1500	1500	256	256	512	1500
num_layers	2	2	2	2	2	2
dp_keep_prob	0.35	0.35	0.5	0.5	0.35	0.35
epoch	5	6	40	40	40	40
train_ppl	RuntimeWarning	2878.328998	129.42173	129.42173	230.3556716	2969.676316
val_ppl	5.870159951440	1427.346405	160.0957279	160.0957279	196.4223734	2188.215742
best_val_ppl/ep	5.870159951440	1425.15055	159.8383529	159.8383529	196.4201496	2188.215742
times	209.4280674	209.5086126	47.22163582	47.22163582	61.75242567	200.5871866

Figure 37: Exploration of hyperparmeters of RNN-2

	4.1	4.3	4.3	4.3	4.3	4.3	4.3	4.3
Model	GRU	GRU	GRU	GRU	GRU	GRU	GRU	GRU
optimizer	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHED	SGD_LR_SCHEDULE
initial_lr	10	20	10	20	1	10	10	10
batch_size	20	20	20	20	20	20	20	20
seq_len	35	35	35	35	35	35	35	35
hidden_size	1500	1500	1500	1500	1500	1500	1500	1500
num_layers	2	2	6	6	2	2	2	2
dp_keep_prob	0.35	0.35	0.35	0.35	0.35	0.45	0.4	0.3
epoch	40	8	8	4	9	30	40	25
train_ppl	65.80590736	147.2253674	254.3093322	5780149609287	357.9224416	39.9322096	51.49325466	83.87685541
val_ppl	102.8590816	151.9090715	246.5661997	516194565359	308.3784495	109.4501981	103.964292	106.3036407
best_val_ppl/epo	102.8590559	151.9090715	246.5661997	89653375009	308.3784495	109.4501817	103.9642601	106.3036407
times		410.6826227	1300.934572	1309.510002	1292.671847	413.5269325	131.42776489257	416.1540909

	4.3	4.3	4.3	4.3	4.3
Model	GRU	GRU	GRU	GRU	GRU
optimizer	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHEDULE
initial_lr	10	10	10	10	10
batch_size	20	20	20	20	20
seq_len	35	35	35	35	35
hidden_size	1200	1200	512	1024	1024
num_layers	2	2	2	2	2
dp_keep_prob	0.6	0.8	0.8	0.8	0.6
epoch	5	5	5	6	9
train_ppl	95.75539351	61.47914483	78.75779601	53.22271211	64.70443652
val_ppl	131.9003087	141.1861711	131.7081363	142.4430813	123.9700555
best_val_ppl/epo	131.9003087	134.624533	131.7081363	138.3011175	121.489554
times	327.4818757				

Figure 38: Exploration of hyperparmeters of GRU-1

	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3
Model	GRU	GRU	GRU	GRU	GRU	GRU	GRU	GRU
optimizer	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHE	SGD_LR_SCHED	SGD_LR_SCHEDULE
initial_lr	10	10	10	10	10	10	10	10
batch_size	20	20	16	20	20	20	20	20
seq_len	32	28	32	35	35	35	35	35
hidden_size	512	512	512	512	512	512	512	512
num_layers	2	2	2	4	4	4	2	2
dp_keep_prob	0.6	0.6	0.5	0.5	0.8	0.35	0.6	0.5
epoch	9	9	9	9	9	9	20	20
train_ppl	84.00794182	84.60979838	105.0554005	133.7052448	74.07380116	197.6283819	49.35105705	65.8019315
val_ppl	115.4189877	117.3658739	120.3040511	139.0558608	149.8769148	170.9104235	103.4637791	100.6964821
best_val_ppl/epo	115.4189877	117.3658739	120.3040511	139.0558608	145.6173968	170.9104235	103.4637791	100.6964821
times	121.2542074	124.0741944	148.4618766	232.8374882	230.7127161	236.8122897	127.6671474	123.6488037
	4.3	4.3	4.3					
Model	GRU	GRU	GRU					
optimizer	ADAM	ADAM	ADAM					
initial_lr	0.0001	0.001	0.001					
batch_size	20	20	20					
seq_len	35	35	35					
hidden_size	512	512	512					
num_layers	2	2	2					
dp_keep_prob	0.5	0.5	0.6					
epoch	9	9	12					
train_ppl	188.6506015	82.60177376	54.57730739					
val_ppl	176.6145393	110.9256958	114.459992					
best_val_ppl/epo	176.6145393	110.9256958	111.791155					
times	132.6551063	131.966043	131.6973612					

Figure 39: Exploration of hyperparmeters of GRU-2

	4.1	4.3	4.3	4.3	4.3	4.3
Model	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER
optimizer	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE
initial_lr	20	20	20	20	20	23
batch_size	128	128	128	128	128	128
seq_len	35	35	35	35	35	35
hidden_size	512	512	512	512	512	512
num_layers	6	2	1	6	6	6
dp_keep_prob	0.9	0.9	0.9	0.35	0.6	0.6
epoch	40	40	40	40	40	40
train_ppl	67.19882666	49.43565582	29.25795522	231.0107935	99.65187487	194.0261101
val_ppl	144.8866225	191.1202604	292.9330015	314.6962053	151.5285471	225.4776737
best_val_ppl/epoch	144.8866052	180.4681568	213.7584389	314.6961772	151.528538	225.4776602
times		50.67112207	36.52638125	112.1302836	111.4609821	112.0492268

	4.3	4.3	4.3	4.3	4.3	4.2
Model	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER
optimizer	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE	SGD
initial_lr	20	21	22	22	20	20
batch_size	128	128	128	128	64	128
seq_len	35	35	35	35	35	35
hidden_size	512	512	512	512	512	512
num_layers	6	6	6	6	6	6
dp_keep_prob	0.75	0.75	0.6	0.7	0.9	0.9
epoch	40	40	40	40	40	40
train_ppl	118.9342547	113.9137217	160.6807629	121.8162308	19.88805216	18.87950156
val_ppl	168.9663509	160.5199581	194.2414393	162.7950542	239.219037	499.6915094
best_val_ppl/epoch	168.9663509	160.5199486	194.2414277	162.7950542	156.8151656	171.4651796
times	114.025816	114.00195	111.5153923	112.2902858	118.9281051	

Figure 40: Exploration of hyperparmeters of TRANS-1

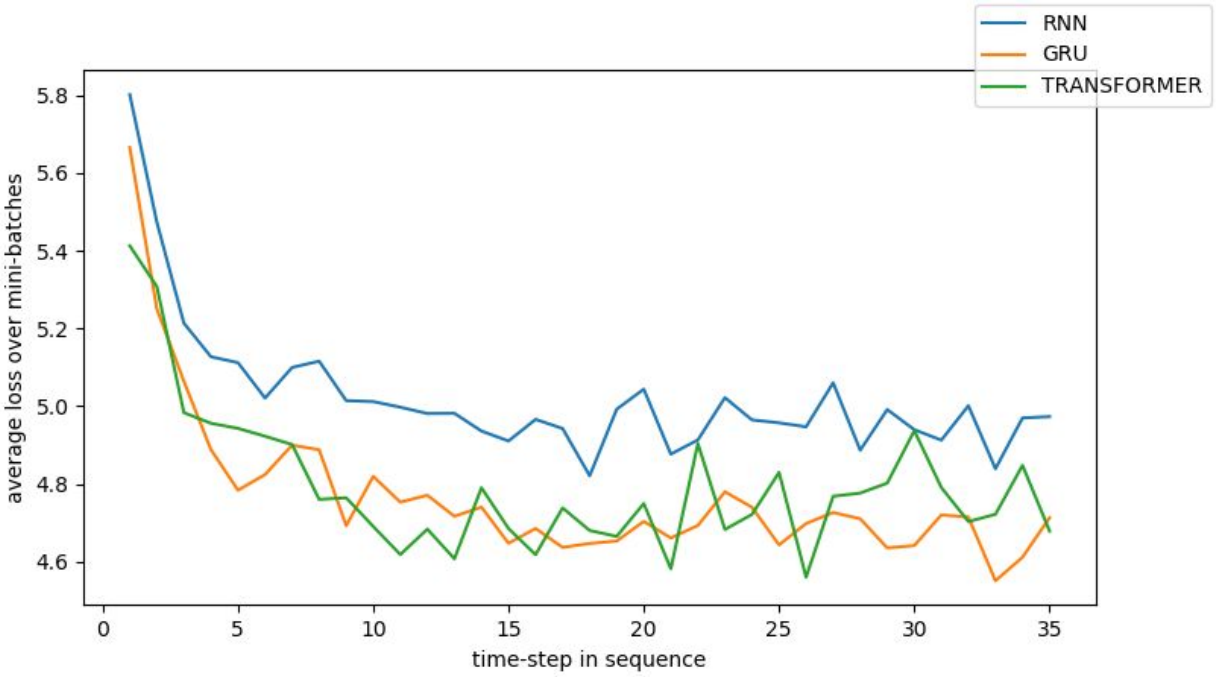
	4.2	4.3	4.3	4.3	4.3	4.3	
Model	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	
optimizer	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	
initial_lr	0.001	0.001	0.001	0.001	0.001	0.001	
batch_size	128	128	128	128	128	128	
seq_len	35	35	35	35	35	35	
hidden_size	512	512	512	512	512	256	
num_layers	2	2	2	1	3	1	
dp_keep_prob	0.9	0.6	0.3	0.3	0.3	0.5	
epoch	40	40	40	40	40	40	
train_ppl	3.163253657	19.54168927	62.49436702	72.37519162	57.44160074	62.73862386	
val_ppl	4907.329036	310.5703584	245.8825746	345.6570997	198.0653305	192.2240632	
best_val_ppl/epoch	132.1678284	152.8973662	241.0523742	339.4193671	192.5519797	187.9207771	
times		54.55883241	53.8299098	36.17440104	64.75852489	23.51605082	
	4.3	4.3	4.3	4.3	4.3	4.3	4.3
Model	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER	TRANSFORMER
optimizer	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM
initial_lr	0.001	0.001	0.001	0.00008	0.00008	0.00001	0.00005
batch_size	60	60	60	256	256	256	256
seq_len	70	70	10	35	35	35	35
hidden_size	512	512	512	512	512	512	512
num_layers	1	1	1	6	2	6	6
dp_keep_prob	0.5	0.7	0.7	0.75	0.75	0.75	0.75
epoch	40	40	40	40	40	40	40
train_ppl	60.65722025	13.56421404	21.44711126	60.64668847	61.83509056	229.8674786	88.67532096
val_ppl	199.7135944	588.8361861	420.7046201	120.7858812	130.6290154	230.282207	128.8374922
best_val_ppl/epoch	195.5009047	152.0967719	167.5366466	120.7104478	130.4080773	230.282207	128.8374922
times	24.78976655	37.88924289	60.04220653	100.7975075	47.60343933	102.6199753	103.3788452

Figure 41: Exploration of hyperparmeters of TRANS-2

Problem 5

For problem 5, we used best model for all sub-questions.

5.1



Problem 5.1 Average loss at each time-step within validation sequences

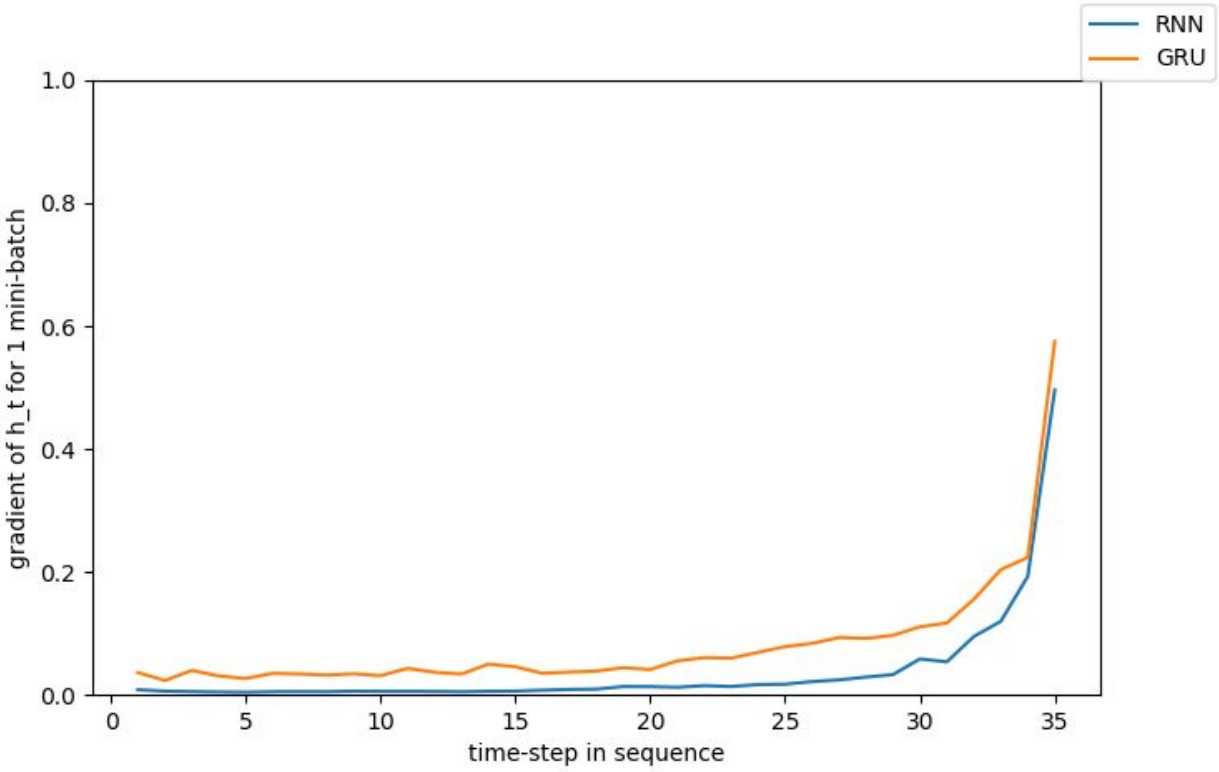
From the graph we can see that the losses of GRU and Transformer are significantly lower than the losses of RNN for all time-steps. This makes sense because GRU and Transformer have higher capacity and more parameters than RNN.

We can see a clear decreasing trend across all architectures. This is because the first time step, we pass zero vectors as init_hidden vectors, which do not have any useful information.

We can see that the Transformer and GRU architectures perform good before time-step 20. Losses of Transformer start to fluctuate after time-step 20 and losses of GRU start to fluctuate after about time-step 30. This shows these models has a limited capability for longer sequences.

We can see that at the first time-step, the loss of RNN > loss of GRU > loss of Transformer. This shows when just having one word, transformer has the best capability to predict the next word. This is also due to the fact that transformer has the greatest capacity and the most parameters. It may also be caused by overfitting.

5.2



Problem 5.2 Average gradient at the final time-step wrt h_t

We see a clearly pattern of gradient vanishing. Although both architecture show the pattern, GRU is generally better than RNN because it still has some gradient at t=0, where the gradient of RNN is almost 0. We see that over all time steps, the gradient curve of GRU is higher than RNN's. From the graph, we can confirm that the architecture of GRU helps avoiding gradient vanishing compare to RNN.

We also see that the gradient drops fairly quickly for both architectures. The norm of gradients start around 0.5 at t=35 and it dropped to lower than 0.1 after 5 time steps. Overall, it shows that the capability of long sequence memory is limited for GRU and doesn't work at all for RNN.

5.3

Best

1. GRU with seq_len=35
['<eos>', 'but', 'it', 'was', 'well', 'during', 'the', 'deal', '<eos>', 'it', 'was', 'known', 'the', 'reins', 'of', 'the', 'bank', 'as', 'well', 'as', 'reported', 'in', 'the', 'chevrolet', 'computer', 'for', 'the', 'market', '<eos>', 'lasted', 'little', 'responsibility', 'according', 'to', 'a']

2. GRU with seq_len=35
['in', 'washington', 'd.c.', 'and', '<unk>', 'group', '<eos>', 'among', 'other', 'things', 'it', 'was', 'sued', 'and', '<unk>', 'program', "'s", 'acquisitions', 'and', 'abolished', 'that', 'stress', 'in', 'japan', 'a', 'waterworks', 'sold', 'most', 'of', 'its', 'three', 'companies', 'in', 'the', 'past']

3. GRU with seq_len=70
['its', 'law', 'in', 'both', 'cities', 'and', 'local', '<unk>', '<eos>', 'an', 'earlier', 'attempt', 'to', 'hide', 'the', 'contract', 'another', 'new', '<unk>', 'division', 'in', 'july', 'or', 'central', 'air', 'is', 'an', 'international', 'nevada', '<eos>', 'an', 'odd', 'number', 'of', 'chemical', "'s", 'conversations', 'outside', 'japan', 'is', 'still', 'in', 'midwestern', 'field', 'or', 'small', 'national', 'chains', '<eos>', 'in', 'return', 'from', 'suggestions', 'to', 'bolster', 'another', 'longstanding', '<unk>', 'hdtv', 'analyst', 'with', '<unk>', '<unk>', 'was', 'new', 'paul', '<unk>', 'the', 'government', "'s"]

Worst

4. RNN with seq_len=70
['prebon', 'u.s.a', 'inc', '<eos>', 'discount', 'rates', '<eos>', 'britain', "'s", 'current', 'remic', 'mortgage', 'commitments', 'to', 'delivery', 'based', 'in', 'japan', 'was', 'priced', 'at', 'N', 'to', 'yield', 'N', 'N', '<eos>', 'their', 'bonds', 'are', 'N', 'low', 'up', 'N', 'N', '<eos>', 'the', 'thin', 'constant', '$', 'N', 'billion', 'of', 'week', 'first', 'receives', 'a', 'N', 'N', 'to', 'settle', '<eos>', 'but', 'now', 'understands', 'the', 'suspension', 'of', 'uncertainty', 'predicting', 'that', 'the', 'percentage', 'point', 'for', 'both', 'major', 'currencies', 'on', 'their']

5. RNN with seq_len=70
['angeles', 'a', '<unk>', 'wash.', 'published', 'the', 'rev.', 'i', 'fbi', "'s", 'that', 'are', 'a', 'standard', 'conventional', 'fixed-rate', 'mortgages', 'N', 'N', 'N', 'N', 'N', 'a', 'place', 'from', 'N', 'N', 'the', 'N', 'N', 'in', 'an', '<unk>', '$', 'N', 'million', 'in', 'five', 'september', 'six', 'called', 'last', 'week', '<eos>', 'two', 'it', 'went', 'to', 'set', 'it', 'probably', 'pass', 'public', 'legislation', 'could', 'not', 'be', 'so', 'easy', 'to', 'do', 'not', 'have', 'some', '<eos>', 'the', 'years', 'is', 'a', 'difficult']

6. RNN with seq_len=35
['in', 'october', 'to', 'the', 'N', 'according', 'to', 'the', 'funding', 'the', 'assembly', 'bureau', 'is', 'promising', 'to', 'N', 'N', 'seven', 'of', 'N', 'various', 'edt', 'last', 'year', '<eos>', 'the', 'drug', 'also', 'are', 'wasted', 'some', 'in', 'east', 'germany', '<eos>']

Interesting

7. GRU with seq_len=70

['<unk>', 'to', 'maintain', 'the', 'markets', 'of', 'the', 'key', 'and', 'shift', 'in', 'the', 'central', '<eos>', 'national', 'security', 'industry', 'is', 'expected', 'to', 'exceed', '$', 'N', 'million', 'in', '10-year', 'protection', 'from', 'a', 'credit-card', 'store', 'program', 'or', 'includes', '$', 'N', 'billion', '<eos>', 'the', 'quantities', 'of', 'the', 'final', 'agreement', 'suggests', "n't", 'influence', 'appealing', 'a', 'capital', 'item', 'should', 'permit', 'product', 'inventories', '<eos>', 'at&t', 'said', 'the', 'premium', 'was', 'slightly', 'higher', 'than', 'the', 'of', 'average', 'and', 'cs', 'no']

8. GRU with seq_len=70

['and', 'bebear', 'has', 'been', 'approved', 'by', 'its', 'authorities', 'in', 'federal', 'could', '<unk>', 'arrangements', '<eos>', 'the', 'industry', 'has', 'also', 'been', 'willing', 'to', 'establish', 'the', 'offer', 'they', 'continue', 'to', 'transport', 'the', 'company', "'s", 'handling', 'as', 'much', 'as', '$', 'N', 'billion', '<eos>', 'the', 'company', 'had', 'no', 'prospective','plans', 'for', 'the', 'collapse', 'of', 'those', 'reasonably', '<unk>', '<eos>', 'the', 'explosions', 'briefly', 'come', '<unk>', 'here', 'and', 'at', 'several', 'above', 'norway', 'and', 'it', 'has', 'come', 'in', 'a']

9. GRU with seq_len=35

['of', 'the', 'national', '<unk>', 'investment', 'council', '<eos>', 'he', 'said', '<unk>', 'of', 'the', 'state', "'s", 'public', 'base', '<unk>', 'shares', 'promises', 'to', 'be', 'in', 'the', 'past', 'five', 'years', '<eos>', 'in', 'september', 'the', 'company', 'has', 'brushed', 'the', 'world']

Observations

- We observed that in general GRU performed much better than RNN. In the sequences generated by RNN, it can contain some short meaningful phrases. If you look at a window of three or four words, RNN may seem working fine but it rarely generates long meaningful sentences, whereas GRU can generate longer meaningful sequences (>10 words).
- Sometimes RNN tends to repeat some words multiple times. See example 4,5,6 for repeating ‘N’s.
- We observed that both models performs better with shorter sequences. This is because they cannot generate long sentences with consistent meaning. When GRU generates long sequences (seq_len=70), the sequence may contain a few independent meaningful sub-sequences (about 20 words each).
- From the example 1, we see that even if we feed ‘<eos>’ as the first token, GRU can generate good sequence for the first half, although it starts to talking about something else for the second half.
- From example 9, we see that the model is able to generate some quotation of people talking. This is actually not rare, we see lot of ‘he’ ‘said’ in the generated sentences.
- From all the examples, we clearly see that the models learnt some characteristics from the training corpus. It usually generates sequences about ‘investment’, ‘company’, ‘million’, ‘industry’, etc...

Appendix

Sequences generated by RNN with the same length as training sequences (seq_length=35)

['a', 'year', 'warns', 'serious', 'securities', '<eos>', 'there', "'s", 'one', 'sort', 'to', 'find', 'the', 'first', 'question', 'of', 'rental', 'issues', '<eos>', 'but', 'there', 'can', 'no', 'support', 'the', 'investment', 'of', 'more', 'than', 'more', 'than', 'at', 'least', 'N', 'in']

['london', '<eos>', 'replace', 'prince', '<unk>', 'and', 'ralph', '<unk>', 'and', 'ignored', '<unk>', 'chairman', 'david', '<unk>', 'who', 'is', 'retiring', 'from', 'this', 'generation', 'of', 'the', '<unk>', 'team', 'donated', 'at&t', 'to', '<unk>', 'by', 'the', 'best', 'room', 'to', 'and', 'drug']

['worth', 'of', 'low', 'cars', 'this', 'month', '<eos>', 'he', 'added', 'that', 'it', "'s", 'mips', 'share', 'the', 'end', 'of', 'the', '<unk>', 'to', 'compete', 'and', 'the', 'certain', 'other','of', '<eos>', 'newly', 'generated', 'hbo', 'here', 'had', 'called', 'the', 'completed']

['doing', 'that', 'the', 'management', 'is', 'for', 'a', '$', 'N', 'million', 'debt', 'or', 'in', 'the', 'u.s.', 'environment', 'that', '<unk>', 'full', 'long-term', 'weaknesses', 'and', 'foreign', 'investments', 'for', 'both', 'companies', '<eos>', 'at', 'wo', "n't", 'come', 'which', '<unk>', 'on']

['views', '<eos>', 'mr.', 'stern', 'also', 'also', 'of', '<unk>', '#', 'N', 'million', 'of', '$', 'N', 'million', 'from', 'leading', 'the', 'company', 'controlled', 'and', 'reserves', 'to', 'the', 'N', 'of', 'option', 'with', 'this', 'form', '<eos>', 'oil', 'companies', '<unk>', 'currently']

['leventhal', 'inc.', 'a', '<unk>', 'group', 'and', 'advertiser', '<eos>', 'this', 'spinoff', 'is', 'be', 'working', 'on', 'the', 'registration', 'pieces', 'of', 'and', 'otherwise', 'would', 'lose', 'its', 'operations', 'until', 'the', 'development', 'contract', 'could', 'affected', 'it', 'can', 'produce', 'a', 'the']

['were', 'important', 'as', '<unk>', 'appearing', 'but', 'they', 'would', "n't", 'respond', 'on', 'a', 'N', 'in', 'hit', 'when', 'the', 'rush', 'would', 'be', 'damaged', 'despite', 'a', 'western-style', 'source', 'of', 'japanese', 'securities', '<eos>', 'japan', 'restoring', 'japan', 'are', 'to', 'or']

['which', 'also', 'may', 'sell', 'a', 'heavy', 'business', 'into', 'product', '<eos>', 'but', 'the', 'electronic', 'institute', 'said', 'that', 'is', "n't", 'discouraging', 'procedures', 'i', "'m", 'looking', 'for', 'more', 'than', 'about', 'more', 'than', '<unk>', 'people', 'only', '<eos>', '<unk>', 'of']

['care', 'new', 'for', 'transportation', 'and', 'three', '<unk>', 'and', '<unk>', 'group', 'magazines', '<eos>', 'the', 'number', 'of', 'N', 'common', 'of', 'u.s.', 'of', 'publishing', 'operations', 'had', 'reported', 'a', '$', 'N', 'million', 'loss', 'with', 'net', 'income', 'of', '$', 'N']

['in', 'october', 'to', 'the', 'N', 'according', 'to', 'the', 'funding', 'the', 'assembly', 'bureau', 'is', 'promising', 'to', 'N', 'N', 'seven', 'of', 'N', 'various', 'edt', 'last', 'year', '<eos>', 'the', 'drug', 'also', 'are', 'wasted', 'some', 'in', 'east', 'germany', '<eos>']

Sequences generated by RNN with the twice the length of training sequences (seq_length=70)

['one-third', 'of', 'the', 'banks', 'by', 'flamboyant', 'or', 'restore', 'programs', 'that', 'jones', 'capital', 'markets', '<eos>', 'the', 'average', 'rate', 'of', 'roughly', 'two-thirds', 'of','the', 'u.s.', 'continued', 'especially', 'forecast', 'in', 'the', 'steady', 'market', '<unk>', 'close', 'to', 'the', 'treasury', "'s", 'N', '<eos>', 'the', 'bill', 'were', 'added', 'how', 'britain', "'s", 'N', 'N', 'of', 'victory', 'in', 'the', 'next', 'year', 'of', 'hurricane', 'hugo', 'and', 'september', 'the', 'soviet', 'communist', 'party', 'congress', 'represent', 'an', 'average', 'of', 'the', 'past', 'N']

['prebon', 'u.s.a', 'inc', '<eos>', 'discount', 'rates', '<eos>', 'britain', "'s", 'current', 'remic', 'mortgage', 'commitments', 'to', 'delivery', 'based', 'in', 'japan', 'was', 'priced', 'at', 'N', 'to', 'yield', 'N', 'N', '<eos>', 'their', 'bonds', 'are', 'N', 'low', 'up', 'N', 'N', '<eos>', 'the', 'thin', 'constant', '$', 'N', 'billion', 'of', 'week', 'first', 'receives', 'a', 'N', 'N', 'to', 'settle', '<eos>', 'but', 'now', 'understands', 'the', 'suspension', 'of', 'uncertainty', 'predicting', 'that', 'the', 'percentage', 'point', 'for', 'both', 'major', 'currencies', 'on', 'their']

['raising', 'continued', 'to', 'arrest', 'u.s.', 'growth', 'rates', 'also', 'increasing', 'dip', 'on', 'steam', 'loans', '<eos>', '<unk>', 'by', 'a', 'percentage', 'of', 'the', 'total', 'particularly', 'located', 'N', 'diabetics', 'ago', 'the', 'devaluation', 'be', 'raised', 'into', 'magazine', 'or', 'its', '<unk>', '<unk>', '<unk>', '<unk>', 'an', 'initial', 'series', 'of', 'international', 'affairs', 'by', 'nbc', '<eos>', 'would', 'have', '<unk>', 'from', 'what', 'a', '<unk>', 'market', 'for', 'this', 'year', 'in', 'a', 'small', 'to', 'pace', 'or', 'least', 'more', 'emphasis', 'on', 'elements', 'of']

['are', 'confident', 'that', 'cash', 'ratings', 'wo', "n't", 'allow', 'if', 'if', 'it', 'has', 'had', 'been', 'under', 'some', 'rules', 'because', 'of', 'any', 'reason', 'suits', 'involving', 'rental', 'companies', 'are', 'in', 'the', 'and', 'credit', 'particularly', 'james', 'thompson', '<eos>', 'under', 'the', 'statement', 'which', 'managed', 'by', 'further', 'b.a.t', 'countries', 'are', 'allowed', 'to', 'protect', 'the', 'offer', 'to', 'meet', 'its', 'existing', 'department', 'between', 'insurance', 'and', 'other', 'goods', 'to', 'petrochemical', 'operations', 'in', 'the', 'u.s.', 'agencies', '<eos>', 'it', 'means', 'ruled']

['directors', '<eos>', '<unk>', 'friday', "'s", '<unk>', 'is', 'clear', 'pending', 'his', 'rates', 'is', 'announced', 'that', 'are', 'largely', 'early', 'dr.', 'louis', 'and', 'it', 'benefited', 'from', 'the', '<unk>', '<eos>', 'digital', "'s", 'corp.', 'also', 'owns', 'about', '$', 'N', 'billion', 'to', 'earnings', 'for', 'the', 'and', '<unk>', 'or', 'consent', 'by', 'the', 'u.s.', 'government', 'attributed', 'the', 'contract', 'to', 'a', 'new', 'fund', '<eos>', 'as', 'small', 'sign', 'of', 'regulations', 'it', 'is', '<unk>', 'to', 'additional', '<unk>', 'their', 'disks', 'to', 'secure']

['angeles', 'a', '<unk>', 'wash.', 'published', 'the', 'rev.', 'i', 'fbi', "'s", 'that', 'are', 'a', 'standard', 'conventional', 'fixed-rate', 'mortgages', 'N', 'N', 'N', 'N', 'N', 'a', 'place', 'from', 'N', 'N', 'the', 'N', 'N',

'in', 'an', '<unk>', '$', 'N', 'million', 'in', 'five', 'september', 'six', 'called', 'last', 'week', '<eos>', 'two', 'it', 'went', 'to', 'set', 'it', 'probably', 'pass', 'public', 'legislation', 'could', 'not', 'be', 'so', 'easy', 'to', 'do', 'not', 'have', 'some', '<eos>', 'the', 'years', 'is', 'a', 'difficult']

['airline', 'with', 'a', '<unk>', 'law', '<eos>', 'there', 'are', 'still', 'action', 'today', '<unk>', 'by', 'ford', 'or', 'no.', 'N', 'the', 'kind', 'of', '<unk>', 'movies', 'have', 'been', 'one', 'consolidated', 'public', 'comic', 'in', 'the', 'great', 'fashion', 'and', '<unk>', '<unk>', 'we', 'publish', 'a', 'better', 'team', 'he', 'was', 'a', '<unk>', 'for', 'publishing', 'and', 'the', 'beginning', 'to', 'be', 'used', 'to', 'be', 'than', 'within', 'a', 'family', 'and', 'he', 'looked', 'for', 'an', 'opportunity', 'to', 'drop', '<eos>', '<unk>', 'contrast', 'mr.']

['<unk>', 'against', 'allows', 'hundreds', 'of', '<unk>', 'customers', 'would', 'have', 'make', 'for', 'important', 'more', 'and', 'we', 'want', 'to', 'work', '<eos>', 'if', 'the', 'privilege', 'is', 'seeking', 'me', 'is', 'allow', 'with', 'a', 'job', '<unk>', 'in', 'the', 'doctrine', 'of', 'ordering', '<unk>', 'with', 'lawsuits', 'by', '<unk>', 'along', 'with', 'the', 'country', 'for', 'the', 'contras', '<eos>', 'and', 'railroad', 'is', 'an', 'office', 'that', 'mr.', '<unk>', '<unk>', 'and', 'should', 'be', 'unable', 'to', 'prevent', 'playing', 'a', 'dramatic', 'scheme', 'from', 'ira']

['toward', 'a', 'N', 'who', 'could', 'be', 'sold', 'for', 'a', 'state-owned', 'that', 'company', "'s", 'core', 'crisis', '<eos>', 'the', 'company', "'s", 'performance', 'is', 'at', 'least', 'N', 'a.m.', 'the', 'largest', 'outside', 'of', '<unk>', '<eos>', 'and', 'no', 'publisher', 'out', 'down', 'the', 'rest', 'between', 'its', 'construction', 'and', 'u.s.', 'image', 'in', 'november', 'N', 'and', 'N', 'ab', 'such', 'any', 'equity', 'growing', 'share', 'as', 'interest', 'rates', 'will', 'be', 'the', 'suburb', 'of', 'hotel', 'and', 'producing', 'electricity', '<eos>', 'in', 'addition']

['that', 'only', 'separate', 'yields', 'on', 'the', 'standard', 'energy', 'crisis', 'are', 'being', 'produced', 'by', 'germany', 'germany', 'two', 'that', 'european', 'outlets', 'on', 'N', 'years', 'in', 'the', 'past', 'turn', "'s", 'proving', 'to', 'read', 'the', 'next', 'year', 'because', 'of', 'N', 'mr.', 'lane', 'who', 'immediately', 'subsequently', 'been', 'like', '<unk>', 'in', 'new', '<unk>', 'as', 'a', 'broadcast', 'hybrid', '<unk>', '<eos>', 'the', '<unk>', 'cancer', 'activist', 'of', 'all', '<unk>', 'was', 'came', 'over', 'the', 'second', 'few', 'months', 'were', 'the', 'same']

Sequences generated by GRU with the same length as training sequences (seq_length=35)

['chunks', 'of', 'an', 'insider', 'to', '<unk>', '<eos>', 'the', 'shrink', 'and', 'immune', 'groups', 'of', 'the', 'big', 'congress', 'is', 'to', 'ignore', 'that', 'most', 'clients', 'can', 'think', 'they', 'are', 'developed', 'for', 'their', 'mistakes', '<eos>', 'rather', 'than', 'the', 'in']

['have', 'made', 'the', 'strict', 'standing', 'and', 'surgical', 'power', '<eos>', 'in', 'the', 'oct.', 'N', '<unk>', 'however', 'merksamer', 'which', 'uncovered', 'a', '<unk>', 'market', 'for', 'such', 'its', '<unk>', 'is', 'somehow', '<unk>', 'to', 'traditional', 'revenue', 'and', 'corporate', 'resistance', 'to']

['erupted', 'at', 'a', '<unk>', 'studio', 'at', '<unk>', 'hill', 'whose', 'headquarters', 'is', 'illegal', 'a', 'telephone', 'consultant', 'to', '<unk>', '<unk>', 'away', 'who', 'at', 'the', 'company', "'s", 'original', 'agency', 'who', 'bought', 'N', 'pages', '$', 'N', 'million', 'replacement', 'in']

['in', 'washington', 'd.c.', 'and', '<unk>', 'group', '<eos>', 'among', 'other', 'things', 'it', 'was', 'sued', 'and', '<unk>', 'program', "'s", 'acquisitions', 'and', 'abolished', 'that', 'stress', 'in', 'japan', 'a', 'waterworks', 'sold', 'most', 'of', 'its', 'three', 'companies', 'in', 'the', 'past']

['to', 'create', 'a', 'recession', 'in', 'last', 'year', "'s", 'N', '<unk>', '<unk>', 'the', '<unk>', '$', 'N', 'million', 'repeated', '<unk>', 'by', '<unk>', '<unk>', 'headquarters', 'to', 'a', 'breakfast', '<unk>', 'of', 'accept', 'for', '<unk>', 'closings', 'the', '<unk>', 'for', '$']

['<eos>', 'but', 'it', 'was', 'well', 'during', 'the', 'deal', '<eos>', 'it', 'was', 'known', 'the', 'reins', 'of', 'the', 'bank', 'as', 'well', 'as', 'reported', 'in', 'the', 'chevrolet', 'computer', 'for', 'the', 'market', '<eos>', 'lasted', 'little', 'responsibility', 'according', 'to', 'a']

['on', 'the', 'metal', '<eos>', 'the', 'largest', 'group', 'shares', 'outnumbered', 'new', 'york', "'s", 'N', 'largest', 'stock', 'in', 'which', 'of', 'the', 'nikkei', 'back', 'high', 'of', 'N', 'shares', 'on', 'oct.', 'N', 'when', 'it', 'bought', 'N', 'shares', 'at', 'uptick']

['<eos>', 'the', 'views', 'by', 'the', '<unk>', 'of', 'a', 'general', 'obligation', 'could', 'be', 'in', 'the', 'statute', 'in', 'environmental', 'control', 'and', 'prepare', 'being', 'turned', 'up', 'and', 'a', '<unk>', 'of', '<unk>', 'had', 'been', 'lortie', 'in', 'restaurants', '<eos>', 'the']

['of', 'the', 'national', '<unk>', 'investment', 'council', '<eos>', 'he', 'said', '<unk>', 'of', 'the', 'state', "'s", 'public', 'base', '<unk>', 'shares', 'promises', 'to', 'be', 'in', 'the', 'past', 'five', 'years', '<eos>', 'in', 'september', 'the', 'company', 'has', 'brushed', 'the', 'world']

['adversary', 'an', 'offer', '<eos>', 'it', 'has', 'been', 'intentionally', 'that', 'in', 'spite', 'of', 'the', '<unk>', 'fare', 'for', 'the', '<unk>', '<eos>', 'hybrid', 'bidding', 'n.j.', 'offer', 'chairman', 'and', 'chief', 'executive', 'officer', 'by', '<unk>', 'group', 'plc', 'said', 'he', 'said']

Sequences generated by GRU with the twice the length of training sequences (seq_length=70)

['<unk>', 'to', 'maintain', 'the', 'markets', 'of', 'the', 'key', 'and', 'shift', 'in', 'the', 'central', '<eos>', 'national', 'security', 'industry', 'is', 'expected', 'to', 'exceed', '$', 'N', 'million', 'in', '10-year', 'protection', 'from', 'a', 'credit-card', 'store', 'program', 'or', 'includes', '$', 'N', 'billion', '<eos>', 'the', 'quantities', 'of', 'the', 'final', 'agreement', 'suggests', "n't", 'influence', 'appealing', 'a', 'capital', 'item', 'should', 'permit', 'product', 'inventories', '<eos>', 'at&t', 'said', 'the', 'premium', 'was', 'slightly', 'higher', 'than', 'the', 'of', 'average', 'and', 'cs', 'no']

['on', 'the', '<unk>', 'sheep', 'the', '<unk>', 'had', 'been', 'signing', 'as', 'milestones', 'of', 'its', '<unk>', 'home', '<unk>', '<unk>', 'the', 'manufacturer', 'mainly', 'to', 'provide', 'its', 'new',

'petrochemical', 'financial', 'footing', '<eos>', 'the', 'scientific', 'decision', 'division', "'s", 'american', 'property', '<unk>', 'institute', 'works', 'to', 'design', 'evidence', 'that', 'and', 'influences', 'the', 'venerable', 'library', '<eos>', 'during', 'the', 'tokyo', 'N', '<unk>', 'foes', 'it', "'s", 'own', 'almost', '<unk>', '<unk>', 'mr.', '<unk>', 'who', 'has', '<unk>', 'in', 'N', 'sought', '<unk>', 'financial']

['the', 'legislators', 'are', 'worried', '<eos>', 'it', 'was', 'not', 'a', 'game', 'where', 'the', 'sister', 'will', 'be', 'dealt', 'with', 'the', 'hands', 'of', 'said', '<eos>', 'he', 'added', 'of', 'the', 'company', "'s", '<unk>', 'advances', '<eos>', 'fujitsu', 'co.', 'noted', 'its', 'pretax', 'profit', 'rose', 'N', 'N', 'to', 'mobil', 'from', 'health', 'for', 'adobe', 'chemicals', 'and', 'accounting', 'operations', 'as', 'expected', 'it', 'sought', 'to', 'post', 'results', 'on', 'its', 'misconduct', 'to', 'edge', 'an', 'estimated', 'loss', 'of', 'sales', '<eos>', 'the', 'company']

['with', 'the', '<unk>', '<eos>', 'a', 'large', 'construction', 'tax', 'should', 'pay', '<eos>', 'polish', '<unk>', 'inc.', 'or', 'its', '<unk>', 'in', '<unk>', '<unk>', '<unk>', 'symbol', 'of', '<unk>', 'creek', 'will', 'cost', 'its', 'N', 'N', 'stake', 'in', 'the', 'maker', 'of', '<unk>', '<unk>', 'and', 'television', '<eos>', 'at', 'some', 'time', 'mr.', '<unk>', 'will', 'be', 'succeeded', 'by', 'ogilvy', '&', 'mather', 'which', 'could', 'be', '<unk>', 'during', 'the', 'turmoil', '<eos>', 'mr.', 'rosenthal', 'is', '<unk>', 'over', 'this', 'summer', 'he', 'said', '<eos>']

['and', 'bebear', 'has', 'been', 'approved', 'by', 'its', 'authorities', 'in', 'federal', 'could', '<unk>', 'arrangements', '<eos>', 'the', 'industry', 'has', 'also', 'been', 'willing', 'to', 'establish', 'the', 'offer', 'they', 'continue', 'to', 'transport', 'the', 'company', "'s", 'handling', 'as', 'much', 'as', '$', 'N', 'billion', '<eos>', 'the', 'company', 'had', 'no', 'prospective','plans', 'for', 'the', 'collapse', 'of', 'those', 'reasonably', '<unk>', '<eos>', 'the', 'explosions', 'briefly', 'come', '<unk>', 'here', 'and', 'at', 'several', 'above', 'norway', 'and', 'it', 'has', 'come', 'in', 'a']

['the', 'spark', 'venture', 'in', 'the', 'country', 'and', 'the', 'creation', 'of', 'the', 'new', 'internal', '<unk>', 'and', 'settlement', 'the', 'big', 'board', '<eos>', 'in', 'all', 'case', 'the', 'fcc', 'can', 'keep', 'it', 'far', 'easier', 'on', 'N', 'but', 'regional', '<eos>', 'features', '<eos>', 'it', 'was', 'still', '<unk>', 'however', '<unk>', 'the', 'official', 'line', '<eos>','the', 'N', 'N', 'owner', 'edwards', 'a', 'shutdown', 'in', 'N', 'has', 'been', 'assured', 'by', 'a', 'company', "'s", 'pilots', 'get', '<unk>', 'favorite', '<unk>', 'and', 'impressive']

["'s", 'new', 'line', 'directly', 'or', 'in', 'chile', "'s", '<unk>', '<unk>', 'by', '<unk>', '<unk>', '&', 'automobile', 'and', 'his', 'financial', 'papers', '<eos>', 'even', 'the', 'other', 'efforts', 'such', 'as', 'the', 'issue', 'can', 'be', 'interrupted', 'by', 'the', 'department', 'at', 'international', 'pioneer', 'corp.', 'which', 'came', "'s", 'san', 'francisco', 'stock', 'that', 'was', 'filed', 'because', 'of', 'the', 'N', 'N', 'of', 'the', 'four-year', 'taxation', 'of', 'northern', 'california', "'s", '$', 'N', 'billion', 'of', 'real-estate', 'obligations', '<eos>', 'the', 'new', 'york']

['its', 'law', 'in', 'both', 'cities', 'and', 'local', '<unk>', '<eos>', 'an', 'earlier', 'attempt', 'to', 'hide', 'the', 'contract', 'another', 'new', '<unk>', 'division', 'in', 'july', 'or', 'central', 'air', 'is', 'an', 'international', 'nevada', '<eos>', 'an', 'odd', 'number', 'of', 'chemical', "'s", 'conversations', 'outside', 'japan', 'is',

'still', 'in', 'midwestern', 'field', 'or', 'small', 'national', 'chains', '<eos>', 'in', 'return', 'from', 'suggestions', 'to', 'bolster', 'another', 'longstanding', '<unk>', 'hdtv', 'analyst', 'with', '<unk>', '<unk>', 'was', 'new', 'paul', '<unk>', 'the', 'government', "'s"]

['bradford', 'price', 'of', 'the', '<unk>', 'gene', 'overseeing', 'mainstream', 'banking', 'corp', 'to', 'a', 'performance', 'of', 'a', 'firm', 'of', '<unk>', 'demler', 'co.', 'his', 'aggressive', '<unk>', 'bottling', 'market', 'officials', 'inc.', 'search', 'for', 'farmers', '<eos>', 'in', 'recent', 'sessions', 'sold', 'at', 'a', 'larger', 'discount', 'in', 'marxist', 'foreign', 'ownership', 'a', 'year', 'ago', 'the', 'area', 'sell', 'other', 'business', 'purchases', 'such', 'as', 'its', 'other', 'agnelli', 'co.', 'in', 'kuala', 'mich', '<eos>', 'the', 'report', 'higher', 'sales', 'for', 'american', 'printing', 'and']

['buoyed', 'by', 'the', '<unk>', 'of', '<unk>', 'with', 'sansui', "'s", 'planned', 'to', 'boost', '<unk>', 'salaries', 'by', 'his', 'role', 'in', '<unk>', 'N', 'to', 'N', 'units', 'just', 'probably', 'is', 'in', 'the', 'process', 'of', 'the', 'new', 'company', '<eos>', 'the', 'ministry', 'also', 'the', 'transportation', 'department', 'said', 'it', 'is', "n't", 'the', 'only', '<unk>', 'joint', 'venture', 'already', 'will', 'be', 'a', 'bigger', '<unk>', '<eos>', 'the', 'ministry', 'said', 'that', 'its', 'recent', 'factors', 'has', '<unk>', 'the', 'u.s.', 'government', '<eos>', 'the']