# 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 = []
```

```python
        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()
```

```python
    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)
```