Applied Deep Learning with Tensorflow and PyTorch, Chapter 7, Week 1

Implementation of Recurrent Neural Networks from Scratch

- A neural network that uses recurrent computation for hidden states is called a recurrent neural network (RNN).
- The hidden state of an RNN can capture historical information of the sequence up to the current time step.
- The number of RNN model parameters does not grow as the number of time steps increases.

RNN Implementation from Scratch in PyTorch

```
def get_params(vocab_size, num_hiddens):
    num_inputs = num_outputs = vocab_size
    def normal(shape):
        return torch.randn(size=shape) * 0.01
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = torch.zeros(num_hiddens)
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs)
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params

def init_rnn_state(batch_size, num_hiddens):
    return (torch.zeros((batch_size, num_hiddens)),)

def rnn(inputs, state, params):
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)

class RNNModelScratch:
    def __init__(self, vocab_size, num_hiddens, get_params,
                 init_state, forward_fn):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens)
        self.init_state, self.forward_fn = init_state, forward_fn
    def __call__(self, X, state):
        X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
        return self.forward_fn(X, state, self.params)
    def begin_state(self, batch_size):
        return self.init_state(batch_size, self.num_hiddens, device="cpu")

net = RNNModelScratch(len(vocab), num_hiddens, get_params,
                      init_rnn_state, rnn)
state = net.begin_state(X.shape[0])
Y, new_state = net(X, state)
```

- A simple RNN language model consists of input encoding, RNN modeling, and output generation.
- RNN models need state initialization for training, though random sampling and sequential partitioning use different ways.
- When using sequential partitioning, we need to **detach** the gradient to reduce computational cost.
- Gradient clipping prevents gradient explosion, but it cannot fix vanishing gradients.

RNN Implementation from Scratch in TensorFlow

```
def get_params(vocab_size, num_hiddens):
    num_inputs = num_outputs = vocab_size
    def normal(shape):
        return tf.random.normal(shape=shape, stddev=0.01, mean=0, dtype=tf.float32)
    W_xh = tf.Variable(normal((num_inputs, num_hiddens)), dtype=tf.float32)
    W_hh = tf.Variable(normal((num_hiddens, num_hiddens)), dtype=tf.float32)
    b_h = tf.Variable(tf.zeros(num_hiddens), dtype=tf.float32)
    W_hq = tf.Variable(normal((num_hiddens, num_outputs)), dtype=tf.float32)
    b_q = tf.Variable(tf.zeros(num_outputs), dtype=tf.float32)
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    return params

def init_rnn_state(batch_size, num_hiddens):
    return (tf.zeros((batch_size, num_hiddens)),)

def rnn(inputs, state, params):
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        X = tf.reshape(X, [-1, W_xh.shape[0]])
        H = tf.tanh(tf.matmul(X, W_xh) + tf.matmul(H, W_hh) + b_h)
        Y = tf.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return tf.concat(outputs, axis=0), (H,)

class RNNModelScratch:
    def __init__(self, vocab_size, num_hiddens, init_state, forward_fn,
                 get_params):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.init_state, self.forward_fn = init_state, forward_fn
        self.trainable_variables = get_params(vocab_size, num_hiddens)
    def __call__(self, X, state):
        X = tf.one_hot(tf.transpose(X), self.vocab_size)
        X = tf.cast(X, tf.float32)
        return self.forward_fn(X, state, self.trainable_variables)
    def begin_state(self, batch_size, *args, **kwargs):
        return self.init_state(batch_size, self.num_hiddens)

net = RNNModelScratch(len(vocab), num_hiddens, init_rnn_state, rnn, get_params)
state = net.begin_state(X.shape[0])
Y, new_state = net(X, state)
```

Concise Implementation of Recurrent Neural Networks

Concise Implementation of RNN in PyTorch

```
rnn_layer = nn.RNN(len(vocab), num_hiddens)
state = torch.zeros((1, batch_size, num_hiddens))
X = torch.rand(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
class RNNModel(nn.Module):
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.num_hiddens = self.rnn.hidden_size
        self.linear = nn.Linear(self.num_hiddens, self.vocab_size)
    def forward(self, inputs, state):
        X = F.one_hot(inputs.T.long(), self.vocab_size)
        X = X.to(torch.float32)
        Y, state = self.rnn(X, state)
        output = self.linear(Y.reshape((-1, Y.shape[-1])))
        return output, state
    def begin_state(self, device, batch_size=1):
            return torch.zeros((self.rnn.num_layers, batch_size,
            self.num_hiddens), device="cpu")
net = RNNModel(rnn_layer, vocab_size=len(vocab))
```

Concise Implementation of RNN in TensorFlow

```
rnn_cell = tf.keras.layers.SimpleRNNCell(num_hiddens,
                                         kernel_initializer='glorot_uniform')
rnn_layer = tf.keras.layers.RNN(rnn_cell, time_major=True,
                                return_sequences=True, return_state=True)
state = rnn_cell.get_initial_state(batch_size=batch_size, dtype=tf.float32)
X = tf.random.uniform((num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
class RNNModel(tf.keras.layers.Layer):
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.dense = tf.keras.layers.Dense(vocab_size)
    def call(self, inputs, state):
        X = tf.one_hot(tf.transpose(inputs), self.vocab_size)
        Y, *state = self.rnn(X, state)
        output = self.dense(tf.reshape(Y, (-1, Y.shape[-1])))
        return output, state
    def begin_state(self, *args, **kwargs):
        return self.rnn.cell.get_initial_state(*args, **kwargs)
net = RNNModel(rnn_layer, vocab_size=len(vocab))
```

- The RNN layer of high-level APIs returns an output and an updated hidden state, where the output does not involve output layer computation.
- Using high-level APIs leads to faster RNN training than using its implementation from scratch.