
RNNs in PyTorch

Heni Ben Amor, Ph.D.
Assistant Professor
Arizona State University

Review: PyTorch Implementation

| To make our own network class in PyTorch we must inherit from torch.nn.Module and create our own constructor and forward method.

```
class FF_NN(nn.Module):
```

Review: PyTorch Implementation

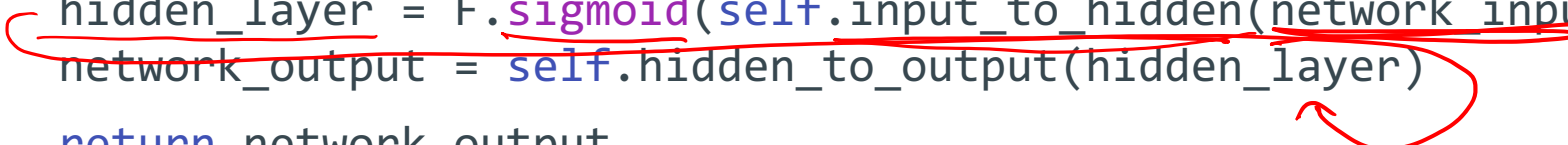
In our constructor we define the network layers and sizes which will make up our network architecture.

```
def __init__(self):  
    super(FF_NN, self).__init__() # Initializes the parent class.  
    self.input_to_hidden = nn.Linear(input_dim, hidden_dim)  
    self.hidden_to_output = nn.Linear(hidden_dim, output_dim)
```

Review: PyTorch Implementation

The forward method is called automatically by Module's callable method with the appropriate network inputs. The inputs are transformed through each step of the network and the final output is returned.

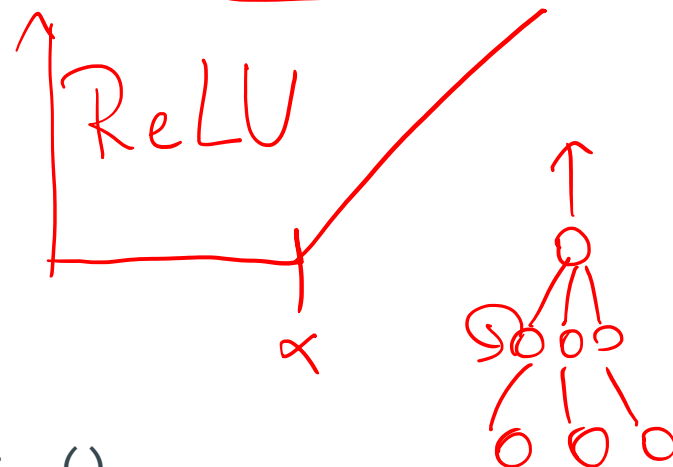
```
def forward(self, network_input):  
    hidden_layer = F.sigmoid(self.input_to_hidden(network_input))  
    network_output = self.hidden_to_output(hidden_layer)  
    return network_output
```



RNN in PyTorch

To utilize temporal features we can add a recurrent component to our network. This is easily done using PyTorch's RNNCell class as a layer.

```
class Recurrent_NN(nn.Module):  
    def __init__(self):  
        super(Recurrent_NN, self).__init__()  
        self.rnn_layer = nn.RNNCell(input_size, hidden_size,  
                                     bias=True, nonlinearity='relu')  
        self.fc_layer = nn.Linear(hidden_size, output_size)
```



RNN in PyTorch

The forward pass works similarly, however we must keep track of and return the RNN cell's hidden state to be passed back into the network along with the next input.

x_t h_{t-1}

```
def forward(self, network_input, hidden_state):  
     $h_t$  hidden_state = self.rnn_layer(network_input, hidden_state)  
    network_output = self.fc_layer(hidden_state)  
    return network_output, hidden_state  $h_t$   $h_{t-1}$ 
```

RNN in PyTorch

Optionally, we can create a method within our network class for initializing the RNN's hidden state to be called before each new input sequence.

```
def init_hidden_state(self):  
    hidden_state = (torch.zeros(batch_size, hidden_size))  
    return hidden_state
```

LSTM in PyTorch

- | We can create an LSTM in PyTorch to help deal with the problem of exploding and vanishing gradients.
- | Similar to the RNNCell class, but we will instead use LSTMCell.

```
class LSTM_NN(nn.Module):  
    def __init__(self):  
        super(LSTM_NN, self).__init__()  
        self.lstm_layer = nn.LSTMCell(input_size, hidden_size,  
                                       bias=True)  
        self.fc_layer = nn.Linear(hidden_size, output_size)
```


LSTM in PyTorch

The LSTMCell forward pass takes in the network input, and an additional tuple containing the hidden and cell state.

h_t C_t

$RNN = \langle h_{t-1}, x_t \rangle$
 $LSTM = \{h_{t-1}, x_t, C_{t-1}\}$

```
def forward(self, network_input, hidden_state, cell_state):  
    hidden_state, cell_state = self.lstm_layer(network_input,  
                                              (hidden_state, cell_state))  
    network_output = self.fc_layer(hidden_state)  
    return network_output, (hidden_state, cell_state)
```

LSTM in PyTorch

Our optional method for initializing the hidden state must also now initialize a cell state as well.

```
def init_hidden(self):  
    hidden_state = torch.zeros(batch_size, hidden_size)  
    cell_state = torch.zeros(batch_size, hidden_size)  
    return (hidden_state, cell_state)
```

Network Update RNN / LSTM

The primary difference with the RNN is that we will iterate over a sequence of inputs, and pass in the updated hidden state to each forward pass through the network.

```
hidden_state = rnn_model.init_hidden_state()
for i in range(sequence_length):
    network_output, hidden_state = rnn_model(network_input[i],
                                              hidden_state)
    loss = loss_function(network_output, target)
```

```
loss.backward()
```

```
optimizer.step()
```

We are typically most concerned with only the final output of the network.

Network Update RNN / LSTM

To adjust this to work with the LSTMCell we simply add the cell state in addition to the hidden state.

LSTM

```
(hidden_state, cell_state) = lstm_model.init_hidden_states()
for i in range(sequence_length):
    network_output, (hidden_state, cell_state) =
        lstm_model(network_input[i], hidden_state, cell_state)
```

Summary



- | Reviewed basic PyTorch implementation
- | RNN
- | LSTM