EE488 Machine Learning Basics and Practice

Mini-Assignment 3 보고서

20180155 김준범

Part 1. Implement LSTM model using handcrafted 'MyLSTM' and 'MyLSTMCell' modules

Task 1. LSTMCell Module Implementation

```
class MyLSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(MyLSTMCell, self).__init__()
        # fill in the blank ------------------------------------
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.fg_x = nn.Linear(input_size,hidden_size)
        self.fg_h = nn.Linear(hidden_size,hidden_size)
        self.ig_x = nn.Linear(input_size,hidden_size)
        self.ig_h = nn.Linear(hidden_size,hidden_size)
        self.og_x = nn.Linear(input_size,hidden_size)
        self.og_h = nn.Linear(hidden_size,hidden_size)
        self.cg_x = nn.Linear(input_size,hidden_size)
        self.cg_h = nn.Linear(hidden_size,hidden_size)

        # ------------------------------------------------------

    def forward(self, input, hidden_states):
        # fill in the blank ------------------------------------
        (hidden,cell) = hidden_states

        ft = torch.sigmoid(self.fg_x(input) + self.fg_h(hidden))
        it = torch.sigmoid(self.ig_x(input) + self.ig_h(hidden))
        ot = torch.sigmoid(self.og_x(input) + self.og_h(hidden))
        ct = torch.tanh(self.cg_x(input) + self.cg_h(hidden))

        new_cell = (ft*cell)+(it*ct)
        new_hidden = ot*torch.tanh(new_cell)
        return (new_hidden,new_cell)
        # ------------------------------------------------------
```

위 사진은 MyLSTMCell의 코드 부분이다. Forget Gate는 fg, Input gate는 ig, output gate는 og으로 변수이름을 설정했고 각 x와 h에 대한 learnable weight는 nn.Linear을 사용해 선언했다. Forget gate는 이전 cell state에서의 값을 얼마나 잊을 것인지를 결정하고, input gate는 새로운 input을 다음 state에서 얼마나 기억할지를 결정한다. 따라서 새로운 cell state는 ft*cell + it*ct으로 계산한 값이고 새로운 hidden state는 output gate 값과 new cell state 값을 elementwise 곱을 통해 구한다.

Task 2. LSTM Module Implementation

```
class MyLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super(MyLSTM, self).__init__()
        # fill in the blank ------------------------------------
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm_1 = MyLSTMCell(input_size,hidden_size)
        self.lstm_2 = MyLSTMCell(hidden_size,hidden_size)
        # ----------------------------------------------------------
```

```
  def forward(self, input, states):
      # fill in the blank ------------------------------------
      hidden, cell = states
      seq_len, batch, input_size = input.size()
      hidden_states=[]
      cell_states=[]
      y_states= []
      temp1=[]
      temp2=[]
      for i in range(self.num_layers):
        temp1.append(hidden[i])
        temp2.append(cell[i])
      hidden_states.append(temp1)
      cell_states.append(temp2)

      for i in range(seq_len):
        hidden_temp=[]
        cell_temp=[]
        for j in range(self.num_layers):
          k = self.num_layers-1-j
          if (j == 0):
            hidden,cell = self.lstm_1(input[i],(hidden_states[i][k],cell_states[i][k]))
            hidden_temp.append(hidden)
            cell_temp.append(cell)
          else:
            hidden,cell = self.lstm_2(hidden_temp[-1],(hidden_states[i][k],cell_states[i][k]))
            hidden_temp.append(hidden)
            cell_temp.append(cell)
        hidden_temp.reverse()
        cell_temp.reverse()
        hidden_states.append(hidden_temp)
        cell_states.append(cell_temp)
        y_states.append(hidden_temp[0])
      yout = torch.stack(y_states,dim=0)
      return yout,(hidden_states[seq_len-1],cell_states[seq_len-1])
      # ----------------------------------------------------------
```

위 사진은 MyLSTM의 코드 부분이다. Forward 과정에서 hidden_states, cell_states, y_states 리스트 initialize되어 각 for loop에서 계산한 결과를 저장한다. 처음에 바로 직전 hidden state와 cell state 값이 hidden_states와 cell_states에 append되고 나서야 반복문에 들어간다. 반복문은 각 seq_len 에 대해서 num_layer만큼 반복하는데 j==0(가장 처음)일 때는 lstm_cell의 input으로 input[i]가 들어가고 그게 아닌 경우 이전 layer에서 계산한 값이 lstem_cell의 input으로 들어간다. 이렇게 하나의 seq_len에 대해서 계산을 완료하면 hidden_states와 cell_states에 해당 결과를 append하여 다음 seq_len에서 사용할 수 있도록 한다. 모든 반복문이 종료되면 torch.stack을 이용해 yout을 구하고 hidden_states와 cell_state의 마지막 element와 함께 return 한다.

Task 3. Defining the final model which consists of LSTM and output Linear layer
Initialize hidden, cell states as zero in init_hidden_cell method

```
class Model(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(Model, self).__init__()
        # fill in the blank ------------------------------------
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.lstm = MyLSTM(input_size,hidden_size,num_layers)
        self.h2o = nn.Linear(hidden_size, output_size) # Adjust to output size utilizing linear layer

        # ------------------------------------------------------

    def forward(self, input, states):
        # fill in the blank ------------------------------------
        hidden,cell = states
        output_LSTM,(hidden,cell) = self.lstm(input,(hidden,cell))
        output = self.h2o(output_LSTM[-1,:,:]).reshape(batch_size,self.output_size) # Get last hidden state

        return output,hidden,cell

        # ------------------------------------------------------

    def init_hidden_cell(self, batch_size=1):
        # fill in the blank ------------------------------------
        hidden = torch.zeros(num_layers,batch_size,hidden_size)
        cell = torch.zeros(num_layers,batch_size,hidden_size)
        return hidden,cell

        # ------------------------------------------------------
```

위 사진은 Model의 코드 부분이다. Week 12에서는 이미 nn.LSTM()을 활용한 model을 사용해 본 경험이 있다. Part1 과제의 주 목적은 우리가 Task 1, 2 과정에서 만든 MyLSTM을 활용해 동일한 model을 돌려보는 것이기 때문에 코드는 Week 12의 LSTM 모듈과 유사하다. 유일하게 다른 점은 class의 생성자에서 self.lstm에 우리가 만든 MyLSTM이 사용된다는 점이다.

Training and evaluation the target problem

```
At   0th epoch, Loss : 2.4418
At 100th epoch, Loss : 0.0032
At 200th epoch, Loss : 0.0027
At 300th epoch, Loss : 0.0001
```

```
At   0th epoch, Loss : 2.5716
At 100th epoch, Loss : 0.0044
At 200th epoch, Loss : 0.0001
At 300th epoch, Loss : 0.0000
```

위 사진의 왼쪽은 MyLSTM을 사용한 Model로 학습시켰을 때의 training loss이고 오른쪽은 nn.LSTM()을 활용한 model로 학습시켰을 때의 training loss이다. 두 경우가 유사하게 loss가 감소하는 것을 볼 수 있다. 아래 사진은 MyLSTM으로 학습시킨 model로 target problem을 evalutate 한 것이다. 뒤 쪽에 있는 문장들도 잘 복원되는 것 을 통해 Task 1,2에서 구현해준 MyLSTM이 잘 작동하는 것을 확인할 수 있었다.

```
hello pytorch, how long can a rnn cell remember? show me your limit!
```

Part 2. Numpy Implementation of **autograd**, **torch-like Tensor** and **Module**

Task 1. Implement backward of the following operations:

```
# handcrafted basic level operations
# addition
def add(a, b):
    # fill in the blank ------------------------------------
    output = a+b
    # ----------------------------------------------------------
    def backward(grad_output=1):
        # fill in the blank ------------------------------------
        grad_input_a = 1
        grad_input_b = 1
        a.backward_fn(grad_input_a*grad_output)
        b.backward_fn(grad_input_b*grad_output)
        # ----------------------------------------------------------
    output.backward_fn = backward
    return output

# multiplication
def mul(a, b):
    # fill in the blank ------------------------------------
    output = a*b
    # ----------------------------------------------------------
    def backward(grad_output=1):
        # fill in the blank ------------------------------------
        grad_input_a = b
        grad_input_b = a
        a.backward_fn(grad_input_a*grad_output)
        b.backward_fn(grad_input_b*grad_output)
        # ----------------------------------------------------------
    output.backward_fn = backward
    return output
```

```
# power of two
def pow_two(a):
    # fill in the blank ------------------------------------
    output = a **2
    # ----------------------------------------------------------
    def backward(grad_output=1):
        # fill in the blank ------------------------------------
        grad_input_a = 2*a
        a.backward_fn(grad_input_a*grad_output)
        # ----------------------------------------------------------
    output.backward_fn = backward
    return output
```

위 사진은 add, mul, pow_two의 operation의 구현 코드이다. 우선 각각의 output은 대응되는 연산인 a+b, a*b, a**2로 계산된다. 이때 add의 경우는 a로 미분했을 때 1, b로 미분했을 때 1이므로 grad_input_a = 1, grad_input_b =1이 된다. Mul의 경우는 a로 미분했을 때 b, b로 미분했을 때 a이므로 grad_input_a = b, grad_input_b =a가 된다. Pow_two의 경우는 a로 미분했을 때 2a이므로 grad_input_a는 2*a가 된다. 각각의 grad_input들과 grad_output의 곱으로 backward_fn을 만들 어주면 아래와 같은 backward 결과를 얻을 수 있다.

```
a = Tensor(2, requires_grad=True)
b = Tensor(3, requires_grad=True)

out = f(a, b)
out.backward()

# check the output and gradient
out, a.grad, b.grad
```

```
(Tensor(225), Tensor(360), Tensor(150))
```

Task 2. Implment zero_grad() and update() methods

```
def zero_grad(self):
    # fill in the blank ------------------------------------
    for v in self.parameters():
      v.grad =0
    # ------------------------------------------------------

def update(self, lr):
    # fill in the blank ------------------------------------
    for v in self.parameters():
      v -= lr*v.grad
    # ------------------------------------------------------
```

위 사진은 zero_grad와 update 함수의 코드 부분이다. Module의 생성자에서 만들어지는 _parameters는 dictionary의 형식이며 self.parameters()를 통해 dictionary의 item의 value들을 받을 수 있다. Zero_grad의 경우는 self.parameters()를 통해 value들을 받고 각 value.grad를 0으로 설정해 준다. Update의 경우도 self.parameters()를 통해 value들을 받고 learning rate와 value.grad의 곱을 value에 빼주어 값을 업데이트 해준다. 위 코드를 통해 gradient를 update하고 zero로 만든 결과는 아래와 같다.

```
class TestModel(Module):
    def __init__(self, init_param):
        super(TestModel, self).__init__()
        self.some_param = Tensor(init_param, requires_grad=True)

    def forward(self, x):
        return f(self.some_param, x)  ## f is defined above in Part1-Task1

test_model = TestModel(12.)
out = test_model(Tensor(0.3))
out.backward()

print('gradient of some_param before zero_grad : ', test_model.some_param.grad)
print('some_param before update : ', test_model.some_param)
test_model.update(lr=0.01)
print('some_param after update : ', test_model.some_param)
test_model.zero_grad()
print('gradient of some_param after zero_grad : ', test_model.some_param.grad)
```

```
gradient of some_param before zero_grad :  626.3999999999999
some_param before update :  12.0
some_param after update :  5.736000000000001
gradient of some_param after zero_grad :  0
```

1. Training on MNIST Dataset

Task 3. Implement the ReLU, Linear modules(with backward)

Task 4. Implement CrossEntropyLoss(with backward)

```
def relu_function(input):
    # fill in the blank ------------------------------------
    output = input * 1*(input>0)
    # ------------------------------------------------------
    def backward(grad_output):
        # fill in the blank ------------------------------------
        grad_input = 1*(input>0)
        input.backward_fn(grad_input*grad_output)
        # ------------------------------------------------------

    output.backward_fn = backward
    return output
```

위 사진은 relu_function에 대한 코드 부분이다. Relu_function은 주어진 input이 음수이면 0, 양수이면 그 값을 그대로 두어 input과 동일한 shape의 output을 return한다. (Input>0)은 input의 각 element에 대해 condition을 확인하여 true와 false로 구성된 array를 만든다. 여기에 1을 곱하면 false였던 자리에는 0, true였던 자리에는 1이 있게 된다. 이 array와 input의 elementwise multiple을 하면 원하는 output을 얻을 수 있게 된다. Backward의 경우, grad_input은 element가 양수이면 1, element가 음수이면 0이어야 함으로 1*(input > 0)과 같고 이것과 grad_output의 elementwise multiple을 한 것이 backward_fn으로 사용된다.

```
def wx_plus_b(W, b, input):
    # fill in the blank ------------------------------------
    output = np.matmul(input,W) + b
    # ------------------------------------------------------
    def backward(grad_output):
        # fill in the blank ------------------------------------
        grad_input_W = input.T
        grad_input_x = W.T
        W.backward_fn(np.matmul(grad_input_W,grad_output)/100)
        input.backward_fn(np.matmul(grad_output,grad_input_x))
        b.backward_fn(np.sum(grad_output, axis=0)/100)
        # ------------------------------------------------------

    output.backward_fn = backward
    return output
```

위 사진은 wx_plus_b의 코드 부분이다. 이 보고서에 첨부되진 않았지만 Model class의 구성을 따라가 보면 Linear layer에서 x*W+b의 연산이 사용된다. 따라서 x로 미분하면 W.T, W로 미분하면 x.T로 gradient를 얻을 수 있다. 따라서 각각의 값을 grad_input_x, grad_input_W로 사용하여 grad_output과 matmul한 것을 backward_fn으로 사용한다. 이때 x의 batchsize는 100으로 정해졌기 때문에 W.backward_fn에서 100으로 나눠주었고 b.backward_fn은 Week 7의 back propagation 코드 부분을 참고하여 작성했다.

```
def compute_cross_entropy_loss(input, target):
 # fill in the blank ------------------------------------
    l = len(input)
    q = np.log(np.sum(np.exp(input),axis=1))
    sum = 0
    for i in range(l):
      sum +=  -1*input[i][target[i]] + q[i]
    v = sum/l
    output = Tensor(v, requires_grad=True)

    # ------------------------------------------------------
    def backward():
        # fill in the blank ------------------------------------
        n,c = input.shape[0], input.shape[1]
        grad_input_i = Tensor(np.zeros(input.shape),requires_grad=True)
        p = torch.sum(torch.exp(torch.Tensor(input)),axis =1)
        for i in range(n):
          for j in range(c):
            t = target[i]
            if j == t:
              grad_input_i[i][j] = math.exp(input[i][j])/p[i] -1
            else:
              grad_input_i[i][j] = math.exp(input[i][j])/p[i]
        input.backward_fn(grad_input_i)
        # ------------------------------------------------------

    output.backward_fn = backward
    return output
```

위 사진은 cross_entopy_loss에 대한 코드 부분이다. L = 100으로 각 batch를 구성하는 100개의 input들에 대해 for문을 돌며 각각의 cross entropy를 계산하고 sum에 더한다. Output은 sum을 l로 나눈 값을 갖는 Tensor이다. Backward 함수에서는 grad_input_i를 input과 같은 크기를 갖는 Tensor로, 우선 임의로 값을 0으로 설정한 뒤 각 i, j 칸을 for문을 돌면서 미분 값으로 다시 assign 해준다. 학습 속도를 더 빠르게 하기 위해 exp의 합 계산은 for문 밖에서 미리 계산하고 반복문 안에서는 indexing을 통해 값을 사용해주기만 한다. 핵심은 j가 target class와 같을 경우에 -1이 추가되는 것이다. Grad_input_i를 계산하면 위의 다른 코드들과 달리 그대로 backward_fn에 들어간다(grad_output이 없기 때문이다).

*p를 계산할 때 np.exp, np.sum 등을 사용한 경우 accuracy가 69.95가 되고 torch.exp, torch.sum을 사용한 경우 accuracy가 70%대로 올라서 torch.Tensor로 잠시 바꿔서 계산했습니다.

Training and Evaluation

```
  5%|▌         | 1/20 [00:25<08:06, 25.60s/it]Loss = 0.5781777946939585
 10%|▉         | 2/20 [00:51<07:41, 25.62s/it]Loss = 0.22873719532443318
 15%|█▍        | 3/20 [01:15<07:10, 25.32s/it]Loss = 0.1640410724755691
 20%|█▊        | 4/20 [01:40<06:40, 25.03s/it]Loss = 0.1257603994957947
 25%|██▎       | 5/20 [02:04<06:13, 24.88s/it]Loss = 0.10119685747190237
 30%|██▋       | 6/20 [02:29<05:47, 24.79s/it]Loss = 0.08333178600107863
 35%|███▏      | 7/20 [02:53<05:20, 24.62s/it]Loss = 0.07027924708791247
 40%|███▌      | 8/20 [03:17<04:54, 24.56s/it]Loss = 0.06055773553076713
 45%|████      | 9/20 [03:42<04:29, 24.49s/it]Loss = 0.05177557093455945
 50%|████▌     | 10/20 [04:06<04:04, 24.44s/it]Loss = 0.044975764742171215
 55%|████▉     | 11/20 [04:31<03:40, 24.47s/it]Loss = 0.03923449391013921
 60%|█████▍    | 12/20 [04:55<03:15, 24.46s/it]Loss = 0.033593181360544477
 65%|█████▊    | 13/20 [05:20<02:51, 24.44s/it]Loss = 0.029359092377834753
 70%|██████▎   | 14/20 [05:44<02:26, 24.43s/it]Loss = 0.02638559426859062
 75%|██████▊   | 15/20 [06:08<02:02, 24.43s/it]Loss = 0.02188277657225809
 80%|███████▏  | 16/20 [06:34<01:38, 24.73s/it]Loss = 0.019211439694421316
 85%|███████▋  | 17/20 [06:58<01:13, 24.66s/it]Loss = 0.016606912947774433
 90%|████████  | 18/20 [07:32<00:54, 27.28s/it]Loss = 0.014621049770453896
 95%|████████▌ | 19/20 [07:56<00:26, 26.46s/it]Loss = 0.012489155597177752
100%|█████████| 20/20 [08:21<00:00, 25.07s/it]Loss = 0.010902043318067 8
Training Finished
```

위 사진은 MNIST Dataset에서 20 epoch로 학습시키는 동안의 training loss를 나타낸 것이다. Loss는 0.5781에서 시작해 점차 줄어들며 0.01까지 줄어드는 것을 볼 수 있다. 아래는 test_image로 test_accuracy를 측정한 결과를 나타낸다.

```
prediction = model(test_images)
prediction_label = np.argmax(prediction, axis=1)
test_acc = np.sum((prediction_label == test_labels))/len(test_labels)
print('Test Accuracy = {:.2f}'.format(100*test_acc))
```

```
Test Accuracy = 98.12
```

2. Training on CIFAR-10 Dataset

Training and Evaluation

```
  3%|▎         | 1/30 [00:18<08:53, 18.39s/it]Loss = 1.2582670023415816
  7%|▋         | 2/30 [00:36<08:35, 18.41s/it]Loss = 1.0919951508004484
 10%|▉         | 3/30 [00:54<08:14, 18.30s/it]Loss = 1.00561870092553
 13%|█▎        | 4/30 [01:12<07:53, 18.21s/it]Loss = 0.9637869468715854
 17%|█▋        | 5/30 [01:30<07:33, 18.13s/it]Loss = 0.9254566579690754
 20%|██        | 6/30 [01:48<07:13, 18.05s/it]Loss = 0.8892995711565709
 23%|██▎       | 7/30 [02:06<06:54, 18.04s/it]Loss = 0.8592798627312358
 27%|██▋       | 8/30 [02:24<06:36, 18.02s/it]Loss = 0.8584353847176592
 30%|███       | 9/30 [02:42<06:17, 17.98s/it]Loss = 0.8277142604913215
 33%|███▎      | 10/30 [03:00<05:58, 17.95s/it]Loss = 0.8027028210928258
 37%|███▋      | 11/30 [03:18<05:41, 17.96s/it]Loss = 0.7792771799979189
 40%|████      | 12/30 [03:36<05:23, 17.97s/it]Loss = 0.7655082703577495
 43%|████▎     | 13/30 [03:54<05:05, 17.96s/it]Loss = 0.7405931547794254
 47%|████▋     | 14/30 [04:12<04:47, 17.95s/it]Loss = 0.7206759232191874
 50%|█████     | 15/30 [04:30<04:29, 17.94s/it]Loss = 0.7131060567937797
 53%|█████▎    | 16/30 [04:48<04:10, 17.91s/it]Loss = 0.7172370026266394
 57%|█████▋    | 17/30 [05:05<03:52, 17.91s/it]Loss = 0.6800015122822465
 60%|██████    | 18/30 [05:23<03:35, 17.93s/it]Loss = 0.6773129184500734
 63%|██████▎   | 19/30 [05:41<03:17, 17.96s/it]Loss = 0.6583498611888343
 67%|██████▋   | 20/30 [05:59<02:59, 17.98s/it]Loss = 0.6512438799563115
 70%|███████   | 21/30 [06:17<02:41, 17.96s/it]Loss = 0.6305094876835061
 73%|███████▎  | 22/30 [06:35<02:23, 17.99s/it]Loss = 0.611703107474812
 77%|███████▋  | 23/30 [06:53<02:05, 17.97s/it]Loss = 0.6174110149762465
 80%|████████  | 24/30 [07:11<01:47, 17.95s/it]Loss = 0.5962815868056617
 83%|████████▎ | 25/30 [07:29<01:29, 17.94s/it]Loss = 0.5968599266443294
 87%|████████▋ | 26/30 [07:47<01:11, 17.92s/it]Loss = 0.5718190408421328
 90%|█████████ | 27/30 [08:05<00:53, 17.95s/it]Loss = 0.5613237884032598
 93%|█████████▎| 28/30 [08:23<00:35, 17.96s/it]Loss = 0.5500586091577057
 97%|█████████▋| 29/30 [08:41<00:17, 17.93s/it]Loss = 0.5268478581278095
100%|██████████| 30/30 [09:03<00:00, 18.11s/it]Loss = 0.5286575279149609
Training Finished
```

위 사진은 CIFAR-10 Dataset에서 30 epoch로 학습시키는 동안의 training loss를 나타낸 것이다. Loss는 1.258에서 시작해 점차 줄어들며 0.52까지 줄어드는 것을 볼 수 있다. 아래는 test_image로 test_accuracy를 측정한 결과를 나타낸다.

```
prediction = model(test_images)
prediction_label = np.argmax(prediction, axis=1)
test_acc = np.sum((prediction_label == test_labels))/len(test_labels)
print('Test Accuracy = {:.2f}'.format(100*test_acc))
```

```
Test Accuracy = 71.58
```