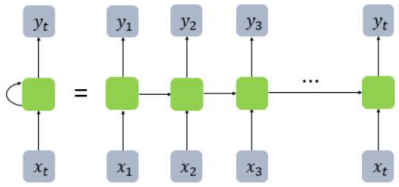
Summary Report [Week 13]. RNN

20190106 KimByungjun (김병준)

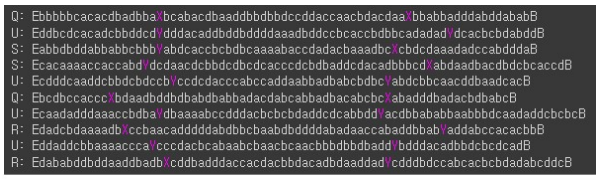
● Exercises

이번 Exercise 에선 RNN(Recurrent Neural Network) structure 중 가장 기본적인 RNN(Vanilla RNN), LSTM(Long Short Term Memory), GRU(Gated Recurrent Unit)을 이용하여 sequence classifier 를 만들고 성능 차이를 관찰하는 것이 목표이다. RNN 이란 시간에 의존적이거나 순차적인 sequential data 를 기반으로 학습하는 neural network 이며, 이전 feedforward neural network 와 달리, hidden node 에서의 계산 결과가 출력층은 물론 다음 시점의 hidden node 계산에도 입력으로 쓰이는 것이 핵심이다. [Fig 1]과 같이 t 시점의 hidden state h_t의 계산에 input x_t 와 이전 시점 t-1 에서의 hidden state h_{t-1} 가 쓰인다.



[Fig 1] hidden state representation

이번 Exercise 에서 쓰이는 character sequence data 는 길이가 70~80 문자 사이이며, E 로 시작하고 B 로 끝난다. 그 사이에는 여러 개의 a, b, c, d 와 위치 10~20 혹은 50~60 내에 X, Y 각각 한 번으로 구성되어 있다. Sequence class 에는 X, Y 의 순서에 따라 (X, X → Q; X, Y → R; Y, X → S; Y, Y → U) Q, R, S, U 4 개의 class 로 나뉜다. Data 및 대응되는 class 를 visualize 하면 **[Fig 2]**와 같다. Training 을 위해 character array 를 float Tensor 로 변환해야 한다. 해당 preprocessing 에 쓰인 rule 은 **[Fig 3]**과 같다. Dataset 을 만들 때, PyTorch 내 TensorDataset class 를 이용하였다.

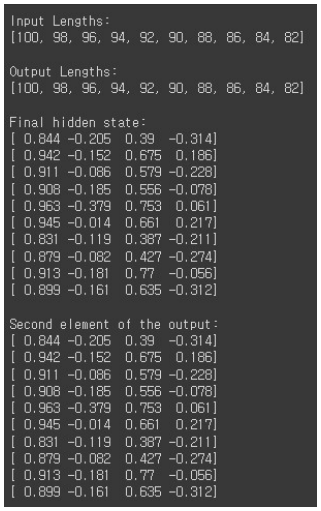


[Fig 2] Sequence data with classes (pink character: X or Y)

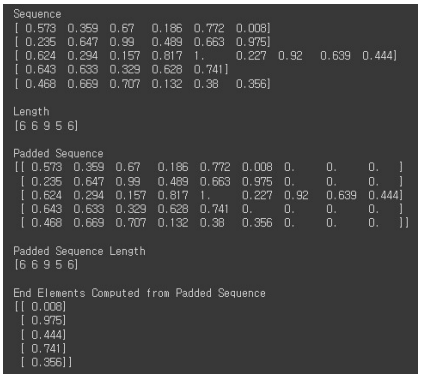
[Fig 3] Sequence Tokenizing and Tensorizing Rule

정해진 사이즈의 image 나 단일 value 가 아니라 길이가 가변적인 sequence 를 다루므로 single batch 에 stack 할 수 없는 등 상대적으로 까다롭다. 이를 해결하기 위해 Pytorch 내 pack_padded_sequence 함수를 이용해 가장 긴 sequence 에 맞춰 data 를 쉽게 packing 할 수 있다(이 때, PackedSequence object 로 전환된다). 이 후, unpack_sequence 함수로 unpacking 할 수 있다.

nn.RNN 의 반환값을 알아보자. nn.RNN 은 두 tensors 를 반환하는데, 첫 번째로 RNN 의 모든 time-step 의 hidden state 를 정보로 가진 tensor, 두 번째로 최종 time-step 의 hidden state tensor 를 반환한다. 실제로, nn.RNN 의 첫 번째 output 을 unpacking 한 후, unpacked output 내 각각의 tensor(여기서는 input 의 길이가 100, 98, ..., 82 로 추출되었으며, 대응되는 output 길이는 input 과 동일하다)의 마지막 원소를 모아보면, 두 번째 output 과 동일함을 알 수 있으며(**[Fig 4]**), 그것이 RNN model 의 마지막 hidden state 를 의미한다. 해당 exercise 에서는 unpack_sequence 함수 대신에 pad_packed_sequence 함수를 이용할 것이다. 이는, 여러 sequence 를 대상으로 할 때 계산이 보다 더 효율적이기 때문이다. 길이 5~10 의 data 를 대상으로 **[Fig 5]**와 같이 padded sequence 와 length tensor 를 구할 수 있으며 길이값을 이용해 마지막 element 값을 추출할 수 있다.



[Fig 4] Output of nn.RNN with



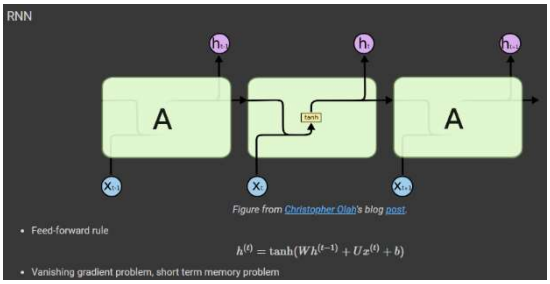
[Fig 5] Output of pad_packed_sequence function

[Fig 6] One-hot embedding

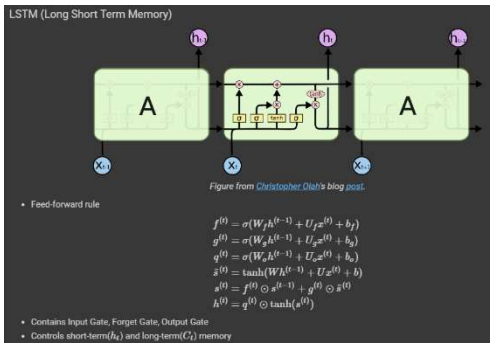
마지막으로, character embedding 을 위해 data 에 one-hot embedding 을 적용한다. 에 쓰인 rule 보다 더 자연스러운 표현을 위해, matrix form 으로 embedding 을 진행한다. **[Fig 6]**은 한 가지 예시를 보여준다.

1) Complete the below code block to implement RNN, LSTM, GRU using functions from PyTorch NN module

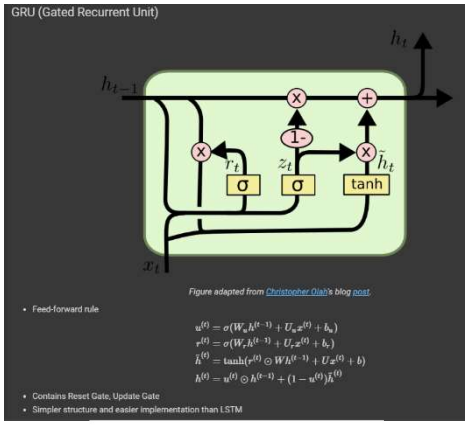
<Explanation>



[Fig 7A] RNN



[Fig 7B] LSTM



[Fig 7C] GRU

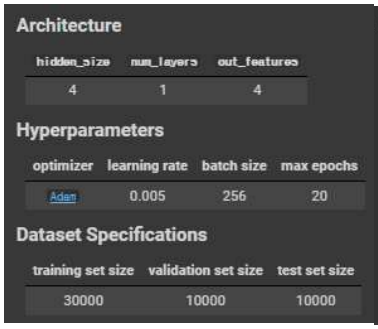
이번 Exercise 에서는 PyTorch NN module 에 있는 RNN, LSTM, GRU 를 사용한다. 이 때, 사용하는 hidden_size 나 num_layers 같은 argument 들은 별도로 정의된 (RNNBase 을 인수로 받는) 각각의 RNN, LSTM, GRU class __init__에 정의된 parameter 를 사용할 것이므로, **kwargs 형식으로 argument 를 받는다. 세 structure 가 아니면, 따로 안내문구가 뜨고 함수를 종료하도록 기본값을 설정해 놓았다.

```python
class RNNBase(nn.Module):
    def __init__(
        self,
        rnn_type: str,
        hidden_size: int,
        num_layers: int,
        out_features: int,
    ) -> None:
        super().__init__()
        kwargs = {
            "batch_first": True,
            "input_size": 8,
            "num_layers": num_layers,
            "hidden_size": hidden_size,
        }
        self.rnn: nn.RNNBase
        match rnn_type:
            ########################
            case "rnn" : self.rnn = nn.RNN(**kwargs)
            case "lstm" : self.rnn = nn.LSTM(**kwargs)
            case "gru" : self.rnn = nn.GRU(**kwargs)
            case _ :
                print("Please check the proper model.")
                return
            ########################

        self.linear = nn.Linear(hidden_size, out_features)
        self.to(device=DEVICE)

    def forward(self, sequences: Tensor) -> Tensor:
        hidden, _ = self.rnn(sequences)
        hidden, lengths = pad_packed_sequence(hidden, batch_first=True)
        hidden = hidden[torch.arange(hidden.size(0)), lengths - 1]
        outputs = self.linear(hidden)
        return outputs
```

[Fig 8] Code for Exercise 1

2) Define your networks according to the instruction above

<Explanation>



[Fig 9A] Architecture of model

```python
[ ] rnn = RNN(hidden_size = 4, num_layers = 1, out_features = 4)
    lstm = LSTM(hidden_size = 4, num_layers = 1, out_features = 4)
    gru = GRU(hidden_size = 4, num_layers = 1, out_features = 4)

    training_set = create_dataset(30000)
    validation_set = create_dataset(10000)
    test_set = create_dataset(10000)

    validloader = DataLoader(
        validation_set, shuffle=False, batch_size=512, drop_last=False
    )
    testloader = DataLoader(
        test_set, shuffle=False, batch_size=512, drop_last=False
    )
```

[Fig 9B] Code for Exercise 2

```python
losses_rnn, accuracies_rnn =train(rnn, 256, 0.005, 20, "RNN")
losses_lstm, accuracies_lstm = train(lstm, 256, 0.005, 20, "LSTM")
losses_gru, accuracies_gru = train(gru, 256, 0.005, 20, "GRU")
```

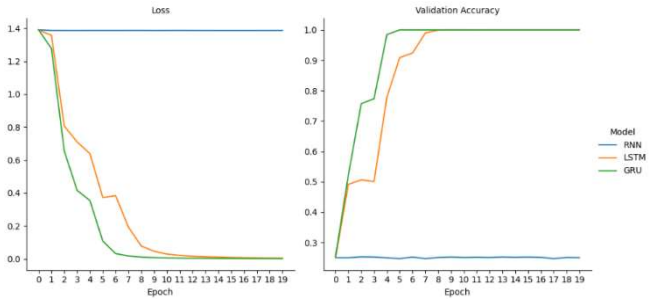
[Fig 10] Hyperparameters Setting and training

주어진 architecture **[Fig 9A]**를 보면, hidden_size = 4, num_layers = 1, out_features = 4 이다. RNN, LSTM, GRU 각각의 class argument 순서 또한 똑같으므로, **[Fig 9B]**와 같이 어려울 것 없이 4, 1, 4 순서대로 model 에 할당하면 된다. Dataset Specifications 는 **[Fig 9B]**에, Hyperparameters 설정은 **[Fig 10]**에 나타냈다.

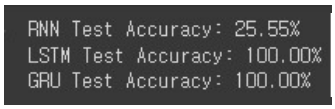
Loss function 은 Cross-Entropy loss 를, optimizer 는 Adam optimizer 를 사용하여 세 model 을 각각 training 시킨다. Training 후 나온 loss 와 validation accuracy 를 plot 하면 **[Fig 11]**와 같은 결과를 얻을 수 있다. 기본적인 RNN model 이 압도적으로 높은 loss 를 보이며 매우 낮은 validation accuracy 를 보인다. LSTM 과 GRU 의 경우, epoch 15 이후부터는 거의 같은 loss 와, validation accuracy 를 보이며, 이 때 loss 는 0 에, validation accuracy 는 100%에 육박하는 것을 알 수 있다. 큰 차이는, GRU model 이 그 완벽한 경우에 보다 더 빠르게 도달한다는 것이고, LSTM 은 GRU 보다 늦게

validation accuracy 100%에 도달하였다. 이는, GRU 가 LSTM model 의 구조를 간소화하여 학습 시간을 감소시켰지만 비슷한 성능을 보인다는 정보에 부합하는 결과이다.

세 model 을 이용하여 test 까지 진행해보면, 앞선 validation 에서 본 것과 같이 RNN 은 25.55%의 정도를, LSTM 과 GRU 는 100%의 완벽한 정확도를 보인다(**[Fig 12]**). 즉, 해당 classification 에선 LSTM, GRU 의 기억 기능을 통해 gradient vanishing problem 을 해소하여야 만 충분한 표현력을 가지며 높은 정확도를 보일 수 있다고 결론지을 수 있다.



[Fig 11] Loss and validation accuracy graph

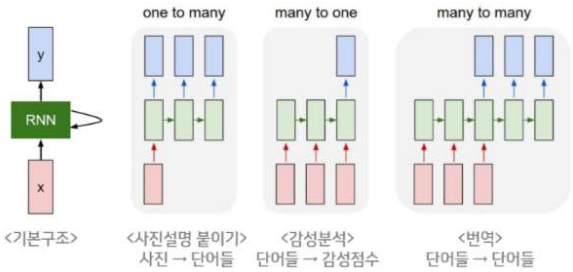


[Fig 12] Test accuracies For each model

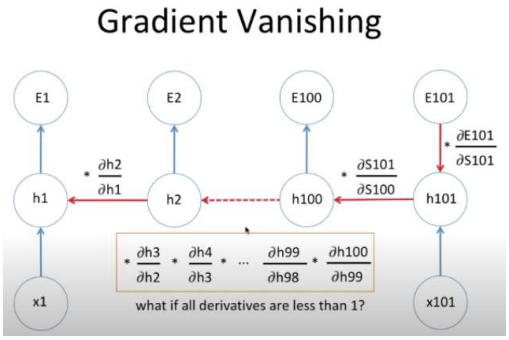
● Discussion

1) Brief Explanation of RNN, LSTM, GRU

위 Exercise 에서 사용한 세 model 에 대한 간략한 차이를 알아보자. RNN 은 앞서 설명하였듯이, 이전 hidden state 와 현재 input 을 이용하여 다음 hidden state 에 대한 연산을 진행하며, 입력과 출력의 개수에 따라 **[Fig 13]**과 같이 one-to-many, many-to-one, many-to-many 로 구분할 수 있다. Vanilla RNN 의 문제는, input sequence 의 길이가 길 때 발생한다. **[Fig 14]**와 같이 Back Propagation 과정에서 연속적으로 gradient 를 계산하게 되는데, 1 보다 작은 수를 반복적으로 곱하면 0 에 수렴하기에 gradient 전달이 점점 미약해져 생기는 gradient vanishing 을 야기한다. 반대로, 각각의 미분 값이 1 보다 커져, gradient 가 폭발적으로 증가하는 gradient exploding 이 발생할 수도 있다.



[Fig 13] Basic structure of RNN



[Fig 14] Gradient Vanishing

이를 해결하기 위해, 기억을 계속 유지하게 해주는 LSTM structure 가 나왔다. LSTM 에서 state 는 단기 기억에 해당하는 단기 상태(short-term state)와 장기 상태(long-term state)로 나뉘어지며, LSTM 에 존재하는 총 3 개의 Gate 를 통해 정보를 통제한다. 현재 정보를 얼마나 기억하는 지 결정하는 Input gate, 정보를 얼마나 잊어버릴 지 결정하는 Forget gate, 다음 층으로 전달할 hidden state 를 만드는 Output gate 를 이용하여 중요한 정보만 기억하는 것이 LSTM 의 가장 큰 특징이다.

GRU 는 앞서 말한 Forget gate 와 Input Gate 를 합쳐 Update Gate 로 구성하고, Reset Gate 를 추가시켜 LSTM 보다 parameter 수를 줄이고 model structure 를 simplify 하였다. 따라서, 학습 속도는 빠르고 성능은 LSTM 과 유사한 성능을 보인다는 게 GRU 의 특징이다. 물론, dataset 의 형태와 구성에 따라 LSTM 과 GRU 는 충분히 서로 다른 성능을 보일 여지가 있다.

Note. LSTM 과 GRU 가 gradient vanishing(exploding)을 완화시킨 것이지, 완벽히 해결한다고는 보장할 수 없다. 보다 더 견고한 model 을 구현하기 위해서는 gradient 를 안정화 시키는 것이 중요한데, gradient 가 일정 threshold 를 넘어가면 값이 너무 뛰지 못하게 방지하는 gradient clipping 을 적용하거나, Resnet 에서와 같은 skip connection 을 도입하여 gradient 가 통과할 수 있는 shortcut 을 만들어 gradient 가 반복적으로 작아지거나 커지는 것을 방지할 수 있다.

● References

- ✓ https://wikidocs.net/22886
- ✓ https://huidea.tistory.com/237