CSC413 A4 Writeup

Richard Yan

April 2023

1 RNNs and Self Attention

1.1 Warmup: A Single Neuron RNN

1.1.1 Effect of Activation - ReLU

1.1.2 Effect of Activation - Different weights

Pretty clear answer from the given ChatGPT response. The vanishing/exploding gradient issue will scale with the weights in the layers, and could be unstable when passed towards the earlier layers.

1.2 Matrices and RNN

1.2.1 Gradient through RNN

Still a pretty good answer from the given ChatGPT response. The noticeable thing of this question is that the maximum value of a derivative of sigmoid is $\frac{1}{4}$, then it's easy to see that when we decompose the input-output derivative into n layer derivative, it follows that a single derivative can be decompose into the product of the sigmoid-derivative and the weight matrix, which both have maximum singular value $\frac{1}{4}$. The spectral norm part is unnecessary, as it's converted back into maximum singular value later.

1.3 Self-Attention

1.3.1 Complexity of Self-Attention

Good answer from the given ChatGPT response.

1.3.2 Linear Attention with SVD

Answer from ChatGPT response miss one component of SVD.

Answer: Assume P has rank k and $P = U\Sigma S^\top$ where $U \in \mathbb{R}^{n\times n}, S \in \mathbb{R}^{d\times d}$ are orthogonal matrices, $\Sigma \in \mathbb{R}^{n\times d}$ is rectangular diagonal matrix.
Then multiplying the SVD to get P only takes $\mathcal{O}(nk)$ complexity, because the

rank of P is k thus the Σ only have the first k diagonal entries with non-zero values, hence we only need to compute the product using the top $n \times k$ block of U and top $k \times d$ block of S to compute P. Finally multiply P with V take $\mathcal{O}(d)$ complexity, and that gives us the attention score. So the overall complexity is $\mathcal{O}(nkd)$.

1.3.3 Linear Attention by Projecting

2 Reinforcement Learning

2.1 Bellman Equation

2.1.1

Correct answer from given ChatGPT response, however a small mistake on the equation (but don't affect the proof process), where the equation for $T^\pi V_i(s)$ should be $r^\pi(S) + \gamma \sum_{s' \in S} \mathcal{P}(s'|s,a)\pi(a|s)V_i(s')$, but notice the difference between any given $V_1, V_2 \in \mathcal{B}(\mathcal{S})$ is the last term in the sum, so the inequality in the ChatGPT response still hold.

2.1.2

A pretty good answer from given ChatGPT response, but could be cleaned up a bit.

The equation part should be like:

$$\begin{aligned}
\|(T^\pi Q_1)(s,a) - T^\pi Q_2)(s,a)\| &= \gamma \| \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)\pi(a|s')(Q_1(s',a) - Q_2(s',a))\| \\
&\leq \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a)\pi(a|s')\|(Q_1(s',a) - Q_2(s',a))\| \\
&\leq \gamma \|Q_1 - Q_2\|_\infty
\end{aligned}$$

Other is correct.

2.1.3

Very good answer from given ChatGPT response, despite not correctly using subscript.

2.2 Policy gradients and black box optimization

2.2.1 Closed form expression for REINFORCE estimator

Answer on the right track from given ChatGPT response, but can be cleaned up a little bit, and correct the dimensionality (i.e. θ is a vector here).

Answer:
From equation 2.16 we can see that $f(\tilde{a}) = \tilde{a}$, so we only need to derive the later derivative part.
The log-likelihood can be expressed as

$$\log p(a = \tilde{a}|\theta) = \tilde{a} \log \mu + (1 - \tilde{a}) \log(1 - \mu)$$

Hence

$$\begin{aligned}
\frac{\partial}{\partial\theta} \log p(a = \tilde{a}|\theta) &= \frac{\tilde{a}}{\mu}\sigma(\mathbf{x}\theta^\top)(1 - \sigma(\mathbf{x}\theta^\top))\mathbf{x}^\top - \frac{1 - \tilde{a}}{1 - \mu}\sigma(\mathbf{x}\theta^\top)(1 - \sigma(\mathbf{x}\theta^\top))\mathbf{x}^\top \\
&= \frac{\tilde{a} - \mu}{\mu(1 - \mu)}\mu(1 - \mu)\mathbf{x}^\top \\
&= (\tilde{a} - \mu)\mathbf{x}^\top
\end{aligned}$$

Then $g[\theta, \tilde{a}]$ is:

$$g[\theta, \tilde{a}] = \tilde{a}(\tilde{a} - \mu)\mathbf{x}^\top$$

2.2.2 Variance of REINFORCE estimator

2.2.3 Convergence and variance of REINFORCE estimator

3 Graph Convolution Networks

3.1

```python
class GraphConvolution(nn.Module):
    """
    A Graph Convolution Layer (GCN)
    """

    def __init__(self, in_features, out_features, bias=True):
        """
        * `in_features`, $F$, is the number of input features per node
        * `out_features`, $F'$, is the number of output features per node
        * `bias`, whether to include the bias term in the linear layer. Default=True
        """
        super(GraphConvolution, self).__init__()
        # TODO: initialize the weight W that maps the input feature (dim F ) to output feature (dim F')
        # hint: use nn.Linear()
        ############ Your code here ####################################
        self.W = nn.Linear(in_features, out_features, bias=bias)


        ################################################################

    def forward(self, input, adj):
        # TODO: transform input feature to output (don't forget to use the adjacency matrix
        # to sum over neighbouring nodes )
        # hint: use the linear layer you declared above.
        # hint: you can use torch.spmm() sparse matrix multiplication to handle the
        #       adjacency matrix
        ############ Your code here ####################################
        s = self.W(input)
        h_prime = torch.spmm(adj, s)
        return h_prime
        ################################################################
```

3.2

```python
class GCN(nn.Module):
    '''
    A two-layer GCN
    '''
    def __init__(self, nfeat, n_hidden, n_classes, dropout, bias=True):
        """
        * `nfeat`, is the number of input features per node of the first layer
        * `n_hidden`, number of hidden units
        * `n_classes`, total number of classes for classification
        * `dropout`, the dropout ratio
        * `bias`, whether to include the bias term in the linear layer. Default=True
        """

        super(GCN, self).__init__()
        # TODO: Initialization
        # (1) 2 GraphConvolution() layers.
        # (2) 1 Dropout layer
        # (3) 1 activation function: ReLU()
        ############ Your code here ###################################
        self.conv1 = GraphConvolution(nfeat, n_hidden, bias)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout)
        self.conv2 = GraphConvolution(n_hidden, n_classes, bias)
        ###############################################################

    def forward(self, x, adj):
        # TODO: the input will pass through the first graph convolution layer,
        # the activation function, the dropout layer, then the second graph
        # convolution layer. No activation function for the
        # last layer. Return the logits.
        ############ Your code here ###################################
        h = self.conv1(x, adj)
        h = self.relu(h)
        h = self.dropout(h)
        out = self.conv2(h, adj)
        logits = F.softmax(out, dim=0)
        return logits
        ###############################################################
```

3.3

```
Epoch: 0091 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2286 time: 0.0024s
Epoch: 0092 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2298 time: 0.0025s
Epoch: 0093 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2298 time: 0.0025s
Epoch: 0094 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2298 time: 0.0025s
Epoch: 0095 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2298 time: 0.0025s
Epoch: 0096 loss_train: 1.9459 acc_train: 0.2357 loss_val: 1.9459 acc_val: 0.2298 time: 0.0025s
Epoch: 0097 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2298 time: 0.0025s
Epoch: 0098 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2298 time: 0.0025s
Epoch: 0099 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2298 time: 0.0043s
Epoch: 0100 loss_train: 1.9459 acc_train: 0.2429 loss_val: 1.9459 acc_val: 0.2298 time: 0.0088s
Optimization Finished!
Total time elapsed: 3.4401s
Test set results: loss= 1.9459 accuracy= 0.2298
```

3.4

```python
class GraphAttentionLayer(nn.Module):

    def __init__(self, in_features: int, out_features: int, n_heads: int,
                 is_concat: bool = True,
                 dropout: float = 0.6,
                 alpha: float = 0.2):
        """
        in_features: F, the number of input features per node
        out_features: F', the number of output features per node
        n_heads: K, the number of attention heads
        is_concat: whether the multi-head results should be concatenated or averaged
        dropout: the dropout probability
        alpha: the negative slope for leaky relu activation
        """
        super(GraphAttentionLayer, self).__init__()

        self.is_concat = is_concat
        self.n_heads = n_heads

        if is_concat:
            assert out_features % n_heads == 0
            self.n_hidden = out_features // n_heads
        else:
            self.n_hidden = out_features

        # TODO: initialize the following modules:
        # (1) self.W: Linear layer that transform the input feature before self attention.
        # You should NOT use for loops for the multiheaded implementation (set bias = Flase)
        # (2) self.attention: Linear layer that compute the attention score (set bias = Flase)
        # (3) self.activation: Activation function (LeakyReLU whith negative_slope=alpha)
        # (4) self.softmax: Softmax function (what's the dim to compute the summation?)
        # (5) self.dropout_layer: Dropout function(with ratio=dropout)
        ################ your code here ########################
        self.W = nn.Linear(in_features, self.n_heads * self.n_hidden, bias=False)
        self.attention = nn.Linear(self.n_hidden * 2, 1, bias=False)
        self.activation = nn.LeakyReLU(negative_slope=alpha)
        self.softmax = nn.Softmax(dim=1)
        self.dropout_layer = nn.Dropout(dropout)
        ########################################################
```

```python
def forward(self, h: torch.Tensor, adj_mat: torch.Tensor):
    # Number of nodes
    n_nodes = h.shape[0]

    # TODO:
    # (1) calculate s = Wh and reshape it to [n_nodes, n_heads, n_hidden]
    #     (you can use tensor.view() function)
    # (2) get [s_i || s_j] using tensor.repeat(), repeat_interleave(), torch.cat(), tensor.view()
    # (3) apply the attention layer
    # (4) apply the activation layer (you will get the attention score e)
    # (5) remove the last dimension 1 use tensor.squeeze()
    # (6) mask the attention score with the adjacency matrix (if there's no edge, assign it to -inf)
    #     note: check the dimensions of e and your adjacency matrix. You may need to use the function unsqueeze()
    # (7) apply softmax
    # (8) apply dropout_layer
    ############## Your code here #########################################
    s = self.W(h).view(n_nodes, self.n_heads, self.n_hidden)
    s_cat = torch.cat([s.unsqueeze(1).repeat_interleave(n_nodes, dim=1), s.unsqueeze(0).repeat_interleave(n_nodes, dim=0)], dim=-1)
    att = self.attention(s_cat)
    e = self.activation(att)
    e = e.squeeze(dim=-1)
    e[adj_mat.unsqueeze(-1).expand(-1, -1, self.n_heads) == 0] = -math.inf
    a = self.softmax(e)
    a = self.dropout_layer(a)
    #######################################################################

    # Summation
    h_prime = torch.einsum('ijh,jhf->ihf', a, s) #[n_nodes, n_heads, n_hidden]


    # TODO: Concat or Mean
    # Concatenate the heads
    if self.is_concat:
        ############## Your code here #########################################
        h_prime = h_prime.contiguous().view(n_nodes, -1)

        #######################################################################
    # Take the mean of the heads (for the last layer)
    else:
        ############## Your code here #########################################
        h_prime = h_prime.mean(dim=1)

    return h_prime
        #######################################################################
```

3.5

```
Epoch: 0091 loss_train: 1.0149 acc_train: 0.7571 loss_val: 1.1869 acc_val: 0.7056 time: 0.2635s
Epoch: 0092 loss_train: 0.9581 acc_train: 0.8286 loss_val: 1.1807 acc_val: 0.7056 time: 0.2634s
Epoch: 0093 loss_train: 0.9481 acc_train: 0.8214 loss_val: 1.1746 acc_val: 0.7064 time: 0.2640s
Epoch: 0094 loss_train: 0.9745 acc_train: 0.7571 loss_val: 1.1686 acc_val: 0.7068 time: 0.2640s
Epoch: 0095 loss_train: 0.8712 acc_train: 0.8000 loss_val: 1.1626 acc_val: 0.7068 time: 0.2651s
Epoch: 0096 loss_train: 0.8678 acc_train: 0.8071 loss_val: 1.1565 acc_val: 0.7072 time: 0.2642s
Epoch: 0097 loss_train: 0.8625 acc_train: 0.8286 loss_val: 1.1509 acc_val: 0.7079 time: 0.2650s
Epoch: 0098 loss_train: 0.8909 acc_train: 0.7500 loss_val: 1.1454 acc_val: 0.7072 time: 0.2637s
Epoch: 0099 loss_train: 0.8704 acc_train: 0.7929 loss_val: 1.1401 acc_val: 0.7060 time: 0.2637s
Epoch: 0100 loss_train: 0.9442 acc_train: 0.7929 loss_val: 1.1346 acc_val: 0.7056 time: 0.2635s
Optimization Finished!
Total time elapsed: 26.6284s
Test set results: loss= 1.1346 accuracy= 0.7056
```

3.6

The performance of GAT is much better than vanilla GCN (0.71 compare to 0.23). This might be because of for any single node, the vanilla GCN take into account all adjacent nodes with same normalized weight (normalized over that single node's degree), while GAT consider more of the adjacent nodes that are important to taht single node.

4 Deep Q-Learning Network (DQN)

4.1

```
def get_action(model, state, action_space_len, epsilon):
    # We do not require gradient at this point, because this function will be used either
    # during experience collection or during inference

    with torch.no_grad():
        Qp = model.policy_net(torch.from_numpy(state).float())

    ## TODO: select and return action based on epsilon-greedy
    if random.random() <= epsilon:
        return torch.tensor(random.randrange(action_space_len)) # exploration
    else:
        return torch.argmax(Qp) # exploitation
```

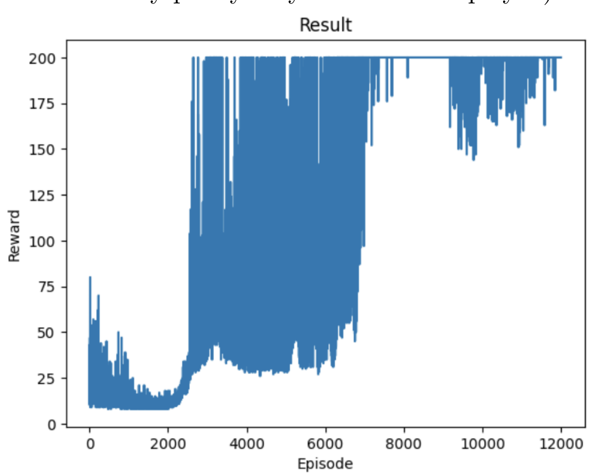
4.2

```
def train(model, batch_size):
    state, action, reward, next_state = memory.sample_from_experience(sample_size=batch_size)

    # TODO: predict expected return of current state using main network
    q = torch.gather(model.policy_net(state), 1, action.unsqueeze(-1).long()).squeeze()
    # TODO: get target return using target network
    q_target = reward + model.gamma * model.target_net(next_state).max(dim=1)[0]

    # TODO: compute the loss
    loss = model.loss_fn(q, q_target)
    model.optimizer.zero_grad()
    loss.backward(retain_graph=True)
    model.optimizer.step()

    model.step += 1
    if model.step % 5 == 0:
        model.target_net.load_state_dict(model.policy_net.state_dict())

    return loss.item()
```

4.3

```
# TODO: add epsilon decay rule here!
min_epsilon = 0.01
if epsilon > min_epsilon:
  epsilon *= 0.999
```

The hyperparameter I used was:
Exp replay size:100
episodes: 12000
epsilon start at: 1
epsilon decay: 0.999
It seems to balance well at the start, where the pole doesn't have any significant movement. However when the pole is leaned toward one direction (by some extent that can be clearly recognized by us from the visualization), it doesn't seem to have ability to move fast enough to let the pole lean to the other side (which is actually pretty easy to do when I play it).




```
100%|██████████| 200/200 [00:23<00:00,  8.64it/s]average reward per episode : 282.945
```