CSC413A4

siweitang

April 2023

1 RNNs and Self Attention

a Warmup: A Single Neuron RNN

i Effect of Activation - Different weights

$$\begin{aligned}
\text{Define } h_0(x) &= xh_n(\text{x}) \text{ as the output} \\
h_i(x) &= w_i max((0, h_{i-1}(x))) \\
\frac{\partial f(x)}{\partial x} &= \frac{\partial h_n(x)}{\partial x} = \frac{\partial h_n(x)}{\partial h_{n-1}(x)} \times \frac{\partial h_{n-1}(x)}{\partial h_{n-2}(x)} ... \times \frac{\partial h_1(x)}{\partial x} \\
\frac{\partial h_i(x)}{\partial h_{i-1}(x)} &= w_i [max(0, h_{i-1}(x)]'
\end{aligned}$$

$$\frac{\partial h_i(x)}{\partial h_{i-1}(x)} = \begin{cases} w_i, & h_{i-1}(x) > 0 \\ 0, & h_{i-1}(x) < 0 \end{cases}$$

$$|\frac{\partial f(x)}{\partial x}| = |\Pi_{h_{i-`1}>0} w_i|$$

This could lead to the vanishing problem.

b Matrices and RNN

i Gradient through RNN

$$\begin{aligned}
Sigmoid'(x) = Sigmoid(x)(1 - Sigmoid(x)) &\Rightarrow 0 \leq Sigmoid'(x) \leq \frac{1}{4} \\
\frac{\partial x_n}{x_1} &= \frac{\partial x_n}{\partial x_{n-1}} ... \frac{\partial x_2}{\partial x_1} \\
\frac{\partial x_n}{\partial x_{n-1}} = \frac{\partial Sigmoid(Wx_{n-1})}{\partial x_{n-1}} &= \frac{\partial Sigmoid(Wx_{n-1})}{\partial Wx_{n-1}} \frac{\partial Wx_{n-1}}{\partial x_{n-1}} \leq \frac{1}{16} \\
\Rightarrow 0 \leq \sigma_{max}(\frac{\partial x_n}{\partial x_1}) &\leq (\frac{1}{16})^{n-1}
\end{aligned}$$

c Self-Attention

i Complexity of Self-Attention

From ChatGPT

$$\alpha_i = \frac{\sum_{j=1}^{n} sim(Q_i, K_j)V_j}{\sum_{j=1}^{n} sim(Q_i, K_j)} = \frac{\sum_{j=1}^{n} k(Q_i, K_j)V_j}{\sum_{j=1}^{n} k(Q_i, K_j)}$$

$$k(Q_i, K_j) = \phi(Q_i)^T \phi(K_j) \Rightarrow \alpha_i = \frac{\phi(Q_i)^T \sum_{j=1}^{n} \phi(K_j)V_j}{\phi(Q_i)^T \sum_{j=1}^{n} \phi(K_j)}$$

$$\text{Let M } = \sum_{j=1}^{n} \phi(K_j)V_j, N = \sum_{j=1}^{n} \phi(K_j)$$

$$\alpha_i = \frac{\phi(Q_i)^T M}{\phi(Q_i)^T N}$$

This can be computed for all i with a single matrix multiplication, which has a complexity of O(n), since M and N have dimensions n × d, where d is the dimensionality of the feature space. Therefore, we have shown that by applying kernel functions, attention can be calculated with linear complexity.

ii Linear Attention with SVD

Assume SVD of P exists

$$SVD = \sum_{i=1}^{k} \sigma_i u_i {v_i}^T = \begin{bmatrix} u_1 & \cdots & u_k \end{bmatrix} diag\{\sigma_1, ..., \sigma_k\} \begin{bmatrix} v_1 \\ \vdots \\ v_k \end{bmatrix}$$

Compute V involves multiply k × n matrix with n × d matrix, it takes O(nkd).
Compute $\begin{bmatrix} u_1 & \cdots & u_k \end{bmatrix} diag\{\sigma_1, ..., \sigma_k\}$ takes O(nk) time, so total O(nkd) time.

2 Reinforcement Learning

a Bellman Equation

i 2.1.1

$$T^\pi V_1(s) - T^\pi V_2(s) = r^\pi(s) + \gamma \int P(s'|s,a)\pi(a|s)V_1(s') - r^\pi(s) - \gamma \int P(s'|s,a)\pi(a|s)V_2(s')$$

$$= \gamma \int P(s'|s,a)\pi(a|s)(V_1 - V_2)(s') \leq 0$$

$$\Rightarrow T^\pi V_1(s) \leq T^\pi V_2(s)$$

ii 2.1.2

$$\|T^\pi(Q_1) - T^\pi(Q_2)\|_\infty = \|\gamma \int P(s'|s,a)\pi(a'|s')(Q_1(s',a') - Q_2(s',a'))ds'da'\|_\infty$$

Let P(s, a) = $\int P(s'|s,a)\pi(a'|s')$, f(s', a') = $Q_1(s',a') - Q_2(s',a')$

$$\begin{aligned}
\|T^\pi(Q_1) - T^\pi(Q_2)\|_\infty &= \gamma\| \int P(s,a)f(s',a')\| \\
&\leq \gamma \int \|P(s,a)f(s',a')\| \\
&= \gamma \int \|P(s,a)\|\|f(s',a')\| \\
&\leq \gamma \int P(s,a) \cdot sup_{(s,a)\in S\times A}\|f(s',a')\| \\
\int\int p(s,a)ds'da' &= \int\int P(s'|s,a)\pi(a'|s')ds'da' = \int\int \pi(a'|s')da'P(s'|s,a)ds' \\
&= \int P(s'|s,a)ds' = 1 \\
\Rightarrow \|T^\pi(Q_1) - T^\pi(Q_2)\|_\infty &\leq \gamma\|f(s',a')\| = \gamma\|Q_1 - Q_2\|_\infty
\end{aligned}$$

iii 2.1.3

1) $v_*(s) = max_a(q_*(s,a))$
2) $q_*(s,a) = r(s,a) + \gamma \int p(s'|s,a)v_*(s')ds'$
3) $a_* = argmax_a q_*(s,a)$
4) $a_* = argmax_a r(s,a) + \gamma \int p(s'|s,a)v_*(s')ds'$

b Policy gradients and black box optimization

i Closed form expression for REINFORCE estimator

$$\begin{aligned}
\frac{\partial}{\partial\theta}logp(a=\hat{a}|\theta) &= \frac{\partial}{\partial\theta}[\hat{a}log\mu + (1-\hat{a})log(1-\mu)] \\
&= \frac{\hat{a}-\mu}{\mu(1-\mu)}\frac{\partial}{\partial\theta}\mu \\
g(\theta,\hat{a}) &= f(\hat{a})\frac{\hat{a}-\mu}{\mu(1-\mu)}\frac{\partial}{\partial\theta}\mu \\
&= f(\hat{a})\frac{\hat{a}-\sigma(\sum_{d=1}^D\theta_d x_d)}{\sigma(\sum_{d=1}^D\theta_d x_d)(1-\sigma(\sum_{d=1}^D\theta_d x_d)}\sigma(\sum_{d=1}^D\theta_d x_d)(1-\sigma(\sum_{d=1}^D\theta_d x_d)x_d \\
&= f(\hat{a})(\hat{a}-\sigma(\sum_{d=1}^D\theta_d x_d))x_d
\end{aligned}$$

3 Graph Convolution Networks

```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.linear = nn.Linear(in_features, out_features, 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 ##################################
        output = self.linear(input)
        output = torch.spmm(adj, output)
        return output
        ##############################################################
```

```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.gcn1 = GraphConvolution(nfeat, n_hidden, bias)
        self.gcn2 = GraphConvolution(n_hidden, n_classes, bias)
        self.dropout = nn.Dropout(dropout)
        self.relu = nn.ReLU()
        ##############################################################

    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 ##################################
        x = self.gcn1(x, adj)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.gcn2(x, adj)
        return x
        ##############################################################
```

```
Epoch: 0067 loss_train: 1.0775 acc_train: 0.6571 loss_val: 1.3326 acc_val: 0.5537 time: 0.0026s
Epoch: 0068 loss_train: 1.0435 acc_train: 0.6857 loss_val: 1.3231 acc_val: 0.5592 time: 0.0027s
Epoch: 0069 loss_train: 1.0277 acc_train: 0.6929 loss_val: 1.3136 acc_val: 0.5646 time: 0.0027s
Epoch: 0070 loss_train: 1.0202 acc_train: 0.6714 loss_val: 1.3041 acc_val: 0.5674 time: 0.0027s
Epoch: 0071 loss_train: 1.0473 acc_train: 0.6857 loss_val: 1.2939 acc_val: 0.5717 time: 0.0027s
Epoch: 0072 loss_train: 0.9992 acc_train: 0.7214 loss_val: 1.2845 acc_val: 0.5748 time: 0.0027s
Epoch: 0073 loss_train: 0.9883 acc_train: 0.7286 loss_val: 1.2754 acc_val: 0.5767 time: 0.0027s
Epoch: 0074 loss_train: 0.9673 acc_train: 0.7214 loss_val: 1.2660 acc_val: 0.5775 time: 0.0027s
Epoch: 0075 loss_train: 0.9550 acc_train: 0.7357 loss_val: 1.2560 acc_val: 0.5810 time: 0.0028s
Epoch: 0076 loss_train: 0.9518 acc_train: 0.7143 loss_val: 1.2468 acc_val: 0.5837 time: 0.0027s
Epoch: 0077 loss_train: 0.9441 acc_train: 0.7357 loss_val: 1.2378 acc_val: 0.5872 time: 0.0027s
Epoch: 0078 loss_train: 0.9319 acc_train: 0.7571 loss_val: 1.2292 acc_val: 0.5892 time: 0.0027s
Epoch: 0079 loss_train: 0.9036 acc_train: 0.7643 loss_val: 1.2215 acc_val: 0.5919 time: 0.0026s
Epoch: 0080 loss_train: 0.9106 acc_train: 0.7429 loss_val: 1.2132 acc_val: 0.5970 time: 0.0094s
Epoch: 0081 loss_train: 0.9149 acc_train: 0.7429 loss_val: 1.2047 acc_val: 0.6001 time: 0.0053s
Epoch: 0082 loss_train: 0.8926 acc_train: 0.7714 loss_val: 1.1965 acc_val: 0.6024 time: 0.0027s
Epoch: 0083 loss_train: 0.9165 acc_train: 0.7500 loss_val: 1.1878 acc_val: 0.6063 time: 0.0027s
Epoch: 0084 loss_train: 0.8798 acc_train: 0.7714 loss_val: 1.1790 acc_val: 0.6079 time: 0.0027s
Epoch: 0085 loss_train: 0.8747 acc_train: 0.7786 loss_val: 1.1702 acc_val: 0.6098 time: 0.0026s
Epoch: 0086 loss_train: 0.8627 acc_train: 0.7929 loss_val: 1.1616 acc_val: 0.6133 time: 0.0026s
Epoch: 0087 loss_train: 0.8612 acc_train: 0.7929 loss_val: 1.1537 acc_val: 0.6153 time: 0.0025s
Epoch: 0088 loss_train: 0.8698 acc_train: 0.8071 loss_val: 1.1461 acc_val: 0.6160 time: 0.0025s
Epoch: 0089 loss_train: 0.8098 acc_train: 0.8357 loss_val: 1.1387 acc_val: 0.6195 time: 0.0025s
Epoch: 0090 loss_train: 0.8286 acc_train: 0.8286 loss_val: 1.1313 acc_val: 0.6223 time: 0.0025s
Epoch: 0091 loss_train: 0.7919 acc_train: 0.8286 loss_val: 1.1238 acc_val: 0.6254 time: 0.0025s
Epoch: 0092 loss_train: 0.7797 acc_train: 0.8429 loss_val: 1.1160 acc_val: 0.6293 time: 0.0025s
Epoch: 0093 loss_train: 0.7539 acc_train: 0.8786 loss_val: 1.1086 acc_val: 0.6336 time: 0.0027s
Epoch: 0094 loss_train: 0.8075 acc_train: 0.8286 loss_val: 1.1020 acc_val: 0.6386 time: 0.0025s
Epoch: 0095 loss_train: 0.7681 acc_train: 0.8500 loss_val: 1.0956 acc_val: 0.6421 time: 0.0025s
Epoch: 0096 loss_train: 0.7314 acc_train: 0.8286 loss_val: 1.0892 acc_val: 0.6452 time: 0.0025s
Epoch: 0097 loss_train: 0.7305 acc_train: 0.8786 loss_val: 1.0826 acc_val: 0.6491 time: 0.0025s
Epoch: 0098 loss_train: 0.7108 acc_train: 0.8786 loss_val: 1.0760 acc_val: 0.6515 time: 0.0025s
Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703 acc_val: 0.6488 time: 0.0024s
Epoch: 0100 loss_train: 0.7307 acc_train: 0.8286 loss_val: 1.0643 acc_val: 0.6515 time: 0.0025s
Optimization Finished!
Total time elapsed: 3.3296s
Test set results: loss= 1.0643 accuracy= 0.6515
```

```
            """
            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, out_features, bias=False)
            self.attention = nn.Linear(self.n_hidden * 2, 1, bias=False)
            self.activation = nn.LeakyReLU(alpha)
            self.softmax = nn.Softmax(dim=1)
            self.dropout_layer = nn.Dropout(dropout)
            ######################################################
```

```
# (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, assig
#     note: check the dimensions of e and your adjacency matrix. You may need to us
# (7) apply softmax
# (8) apply dropout_layer
############## Your code here ############################################
s = self.W(h).view(n_nodes, self.n_heads, self.n_hidden)
# print(h.shape)
# print(s.shape)
s_i = s.repeat_interleave(n_nodes, dim=0)
s_j = s.repeat(n_nodes, 1, 1)
# print(n_nodes)
# print(s_i.shape)
# print(s_j.shape)
s_ij = torch.cat([s_i, s_j], dim=-1)
# print(s_ij.shape)
s_ij = s_ij.view(n_nodes, n_nodes, self.n_heads, 2*self.n_hidden)
# print(s_ij.shape)
e = self.attention(s_ij)
# print(e.shape)

e = self.activation(e)
# print(e.shape)
e = e.squeeze(dim=-1)
# print(e.shape)
# print(adj_mat.shape)
adj_mat = adj_mat.unsqueeze(dim=-1)
e = e.masked_fill(adj_mat == 0, float('-inf'))
a = self.softmax(e)
a = self.dropout_layer(a)
##########################################################################
```

```
# TODO: Concat or Mean
# Concatenate the heads
if self.is_concat:
    ############## Your code here ############################################
    h_prime = h_prime.reshape(n_nodes, self.n_heads * self.n_hidden)
    return h_prime
    ##########################################################################
# Take the mean of the heads (for the last layer)
else:
    ############## Your code here ############################################
    h_prime = h_prime.mean(dim=1)
    return h_prime
    ##########################################################################
```

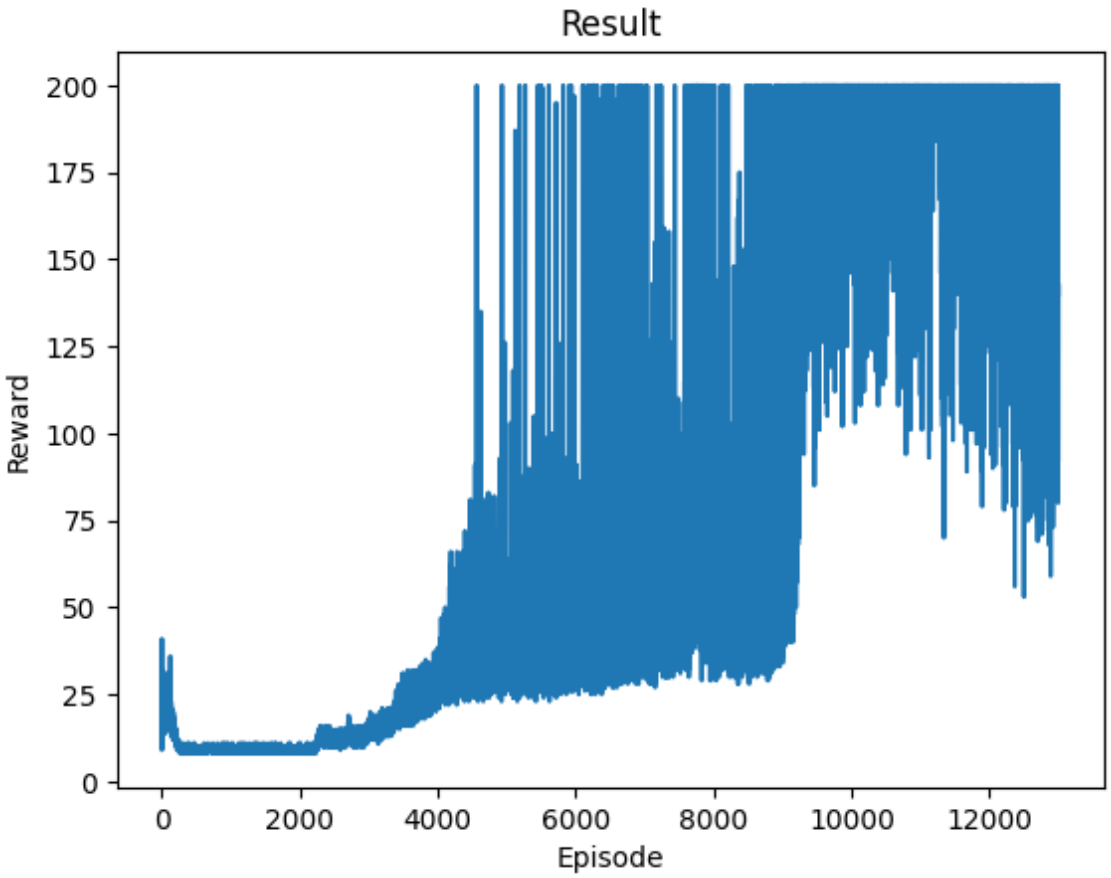
```
Epoch: 0069 loss_train: 1.1874 acc_train: 0.7429 loss_val: 1.2965 acc_val: 0.7060 time: 0.2250s
Epoch: 0070 loss_train: 1.1291 acc_train: 0.8214 loss_val: 1.2863 acc_val: 0.7083 time: 0.2273s
Epoch: 0071 loss_train: 1.1738 acc_train: 0.7500 loss_val: 1.2763 acc_val: 0.7103 time: 0.2245s
Epoch: 0072 loss_train: 1.2768 acc_train: 0.6714 loss_val: 1.2668 acc_val: 0.7130 time: 0.2251s
Epoch: 0073 loss_train: 1.0586 acc_train: 0.7857 loss_val: 1.2574 acc_val: 0.7208 time: 0.2246s
Epoch: 0074 loss_train: 1.1073 acc_train: 0.8000 loss_val: 1.2482 acc_val: 0.7259 time: 0.2251s
Epoch: 0075 loss_train: 1.1218 acc_train: 0.7429 loss_val: 1.2392 acc_val: 0.7282 time: 0.2249s
Epoch: 0076 loss_train: 1.0699 acc_train: 0.7929 loss_val: 1.2304 acc_val: 0.7329 time: 0.2246s
Epoch: 0077 loss_train: 1.0501 acc_train: 0.7571 loss_val: 1.2220 acc_val: 0.7368 time: 0.2247s
Epoch: 0078 loss_train: 1.0669 acc_train: 0.7929 loss_val: 1.2137 acc_val: 0.7418 time: 0.2249s
Epoch: 0079 loss_train: 1.1081 acc_train: 0.7857 loss_val: 1.2055 acc_val: 0.7457 time: 0.2247s
Epoch: 0080 loss_train: 1.0615 acc_train: 0.7929 loss_val: 1.1973 acc_val: 0.7473 time: 0.2273s
Epoch: 0081 loss_train: 0.9879 acc_train: 0.8286 loss_val: 1.1893 acc_val: 0.7512 time: 0.2249s
Epoch: 0082 loss_train: 1.0444 acc_train: 0.8500 loss_val: 1.1813 acc_val: 0.7531 time: 0.2273s
Epoch: 0083 loss_train: 1.0470 acc_train: 0.7786 loss_val: 1.1734 acc_val: 0.7551 time: 0.2250s
Epoch: 0084 loss_train: 1.0705 acc_train: 0.7857 loss_val: 1.1655 acc_val: 0.7578 time: 0.2250s
Epoch: 0085 loss_train: 1.0988 acc_train: 0.7786 loss_val: 1.1580 acc_val: 0.7590 time: 0.2256s
Epoch: 0086 loss_train: 0.9949 acc_train: 0.7929 loss_val: 1.1506 acc_val: 0.7613 time: 0.2267s
Epoch: 0087 loss_train: 1.0391 acc_train: 0.7929 loss_val: 1.1432 acc_val: 0.7613 time: 0.2246s
Epoch: 0088 loss_train: 1.0391 acc_train: 0.8071 loss_val: 1.1360 acc_val: 0.7617 time: 0.2249s
Epoch: 0089 loss_train: 0.9974 acc_train: 0.8071 loss_val: 1.1290 acc_val: 0.7621 time: 0.2259s
Epoch: 0090 loss_train: 1.0134 acc_train: 0.8429 loss_val: 1.1219 acc_val: 0.7617 time: 0.2250s
Epoch: 0091 loss_train: 0.9469 acc_train: 0.7857 loss_val: 1.1150 acc_val: 0.7621 time: 0.2245s
Epoch: 0092 loss_train: 0.9287 acc_train: 0.8357 loss_val: 1.1081 acc_val: 0.7621 time: 0.2249s
Epoch: 0093 loss_train: 0.9868 acc_train: 0.8000 loss_val: 1.1012 acc_val: 0.7625 time: 0.2251s
Epoch: 0094 loss_train: 0.9964 acc_train: 0.7571 loss_val: 1.0944 acc_val: 0.7636 time: 0.2269s
Epoch: 0095 loss_train: 0.9597 acc_train: 0.8286 loss_val: 1.0877 acc_val: 0.7640 time: 0.2245s
Epoch: 0096 loss_train: 0.9158 acc_train: 0.8143 loss_val: 1.0811 acc_val: 0.7636 time: 0.2249s
Epoch: 0097 loss_train: 0.9022 acc_train: 0.7786 loss_val: 1.0745 acc_val: 0.7636 time: 0.2246s
Epoch: 0098 loss_train: 0.9673 acc_train: 0.7571 loss_val: 1.0682 acc_val: 0.7640 time: 0.2257s
Epoch: 0099 loss_train: 0.8681 acc_train: 0.7857 loss_val: 1.0620 acc_val: 0.7667 time: 0.2249s
Epoch: 0100 loss_train: 0.9062 acc_train: 0.8429 loss_val: 1.0559 acc_val: 0.7675 time: 0.2251s
Optimization Finished!
Total time elapsed: 22.7744s
Test set results: loss= 1.0559 accuracy= 0.7675
```

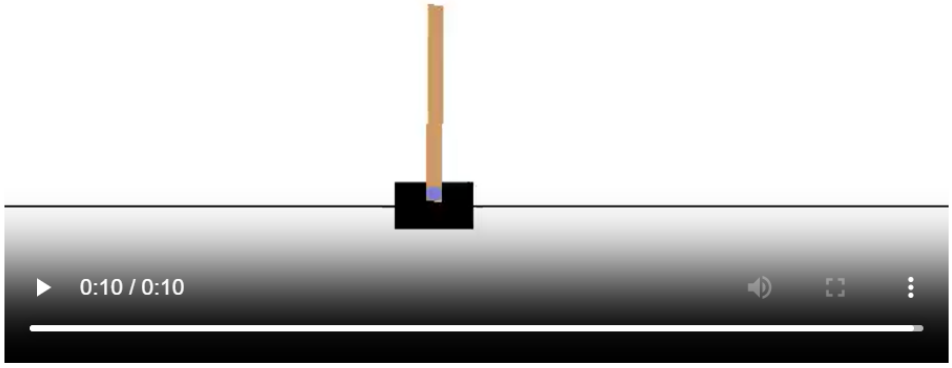
GAT has much better performance than Vanilla GCN, 0.7675 is much higher than 0.6515. GATs apply different weights to different nodes based on their importance. This allows GATs to better handle node features and capture more fine-grained information.

4 Deep Q-learning Network

```python
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
    _, maximum_action = torch.max(Qp, axis=0)
    action  = 0
    if torch.rand(1) > epsilon:
      action = maximum_action
    else:
      action = torch.randint(0, action_space_len, (1,))
    return action
```

```python
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
    Qp = model.policy_net(state)
    expected_return = torch.zeros(len(action))
    for i in range(len(action)):
      expected_return[i] = Qp[i][int(action[i])]

    # TODO: get target return using target network
    target_return = reward + model.gamma * model.target_net(next_state).max(axis=-1)[0]

    # TODO: compute the loss
    loss = model.loss_fn(input=expected_return, target=target_return)
    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()
```


Result
Reward
Episode
0
25
50
75
100
125
150
175
200
0
2000
4000
6000
8000
10000
12000


0:10 / 0:10

```
            break

    index = 128
    for i in tqdm(range(episodes)):
        obs, done, losses, ep_len, rew = env.reset(), False, 0, 0, 0
        while not done:
            ep_len += 1
            A = get_action(agent, obs, env.action_space.n, epsilon)
            obs_next, reward, done, _ = env.step(A.item())
            memory.collect([obs, A.item(), reward, obs_next])

            obs = obs_next
            rew += reward
            index += 1

            if index > 128:
                index = 0
                for j in range(4):
                    loss = train(agent, batch_size=16)
                    losses += loss

        # TODO: add epsilon decay rule here!
        epsilon = epsilon * 0.96
        losses_list.append(losses / ep_len), reward_list.append(rew)
        episode_len_list.append(ep_len), epsilon_list.append(epsilon)

    print("Saving trained model")
    agent.save_trained_model("cartpole-dqn.pth")
```

```
# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 17000
memory = ExperienceReplay(exp_replay_size)
episodes = 13000
epsilon = 1 # episilon start from 1 and decay gradually.
```

If epsilon decays slowly and episodes is small, then rewards would not be stable at 200. Only if episodes is large enough and epsilon decays faster, the rewards would stabilize at 200.