CSC2516 HW4

Zorina Song

April 2023

1 RNNs and Self Attention

For any successful deep learning system, choosing the right network architecture is as important as choosing a good learning algorithm. In this question, we will explore how various architectural choices can have a significant impact on learning. We will analyze the learning performance from the perspective of vanishing /exploding gradients as they are backpropagated from the final layer to the first.

1.1 Warmup: A Single Neuron RNN

Consider an n layered fully connected network that has scalar inputs and outputs. For now, assume that all the hidden layers have a single unit, and that the weight matrices are set to 1 (because each hidden layer has a single unit, the weight matrices have a dimensionality of $R^{1\times 1}$).

1.1.1 Effect of Activation - ReLU [0pt]

Lets say we're using the ReLU activation. Let x be the input to the network and let $f : R^1 \to R^1$ be the function the network is computing. Do the gradients necessarily have to vanish or explode as they are backpropagated? Answer this by showing that $0 \leq |\frac{\partial f(x)}{\partial x}| \leq 1$.

1.1.2 Effect of Activation - Different weights blue [0.5pt]

Solve the problem in 1.1.1 by assuming now the weights are not 1. You can assume that the i-th hidden layer has weight w_i. Do the gradients necessarily have to vanish or explode as they are backpropagated? Answer this by deriving a similar bound as in Sec 1.1.1 for the magnitude of the gradient.

ANS:(base on answers from ChatGPT-4)

To answer this question, let's first derive the expression for the output $f(x)$ of the given fully connected network with ReLU activation and then compute the gradient of the output with respect to the input x.

The output of each layer can be represented as:

Layer 1: $h_1 = \max(0, w_1 x)$
Layer 2: $h_2 = \max(0, w_2 h_1) = \max(0, w_2 \max(0, w_1 x))$
...
Layer n: $f(x) = h_n = \max(0, w_n h_{n-1}) = \max(0, w_n \max(0, w_{n-1} \max(... \max(0, w_1 x)...)))$

Now, let's compute the gradient of the output with respect to the input x:

$\frac{df(x)}{dx} = \frac{dh_n}{dx} = \frac{dh_n}{dh_{n-1}} \frac{dh_{n-1}}{dx}$

Using the chain rule, we can find the gradient for each layer:

$$\frac{dh_i}{dh_{i-1}} = \begin{cases} w_i, & \text{if } h_{i-1} > 0 \\ 0, & \text{otherwise} \end{cases}$$

So, the gradient of the output with respect to the input x is:

$$\frac{df(x)}{dx} = \prod_{i=1}^{n} \frac{dh_i}{dh_{i-1}} = \prod_{i=1}^{n} \begin{cases} w_i, & \text{if } h_{i-1} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Now, let's derive a bound for the magnitude of the gradient. We can see that the magnitude of the gradient is upper-bounded by the product of the absolute values of the weights of each layer:

$|\frac{df(x)}{dx}| \leq \prod_{i=1}^{n} |w_i|$

To answer the question of whether the gradients necessarily have to vanish or explode as they are backpropagated, the answer is no. The gradient's magnitude is determined by the product of the absolute values of the weights. If the weights are properly initialized and updated during training, the gradients should not necessarily vanish or explode.

The bound we derived shows that the gradient's magnitude depends on the product of the weights. If the weights are all smaller than 1 (in magnitude), the gradient might vanish. If the weights are all larger than 1 (in magnitude), the gradient might explode. However, if the weights are well-balanced, the gradient can stay within a reasonable range, preventing the vanishing or exploding gradient problem.

1.2 Matrices and RNN

We will now analyze the recurrent weight matrices under Singular Value Decomposition. SVD is one of the most important results in all of linear algebra. It says that any real matrix $M \in R^{mxn}$ can be written as $M = U\Sigma V^T$ where $U \in R^{mxm}$ and $V \in R^{nxn}$ are square orthogonal matrices, and $\Sigma \in R^{mxn}$

is a rectangular diagonal matrix with nonnegative entries on the diagonal (i.e. $\Sigma_{ii} \geq 0$ for $i \in \{1, \ldots, \min(m, n)\}$ and 0 otherwise). Geometrically, this means any linear transformation can be decomposed into a rotation/flip, followed by scaling along orthogonal directions, followed by another rotation/flip.

1.2.1 Gradient through RNN blue [0.5pt]

Let say we have a very simple RNN-like architecture that computes $x_{t+1} = \text{sigmoid}(Wx_t)$. You can view this architecture as a deep fully connected network that uses the same weight matrix at each layer. Suppose the largest singular value of the weight matrix is $\sigma_{max}(W) = \frac{1}{4}$. Show that the largest singular value of the input-output Jacobian has the following bound: red

$$0 \leq \sigma_{max}(\frac{\partial x_n}{\partial x_1}) \leq (\frac{1}{16})^{n-1}$$

(Hint: if $C = AB$, then $\sigma_{max}(C) \leq \sigma_{max}(A)\sigma_{max}(B)$. Also, the input-output Jacobian is the multiplication of layerwise Jacobians).

ANS:(base on answers from ChatGPT-4)
To show the bound on the largest singular value of the input-output Jacobian, let's first derive the Jacobian for each time step and then analyze the singular values.

Consider the RNN-like architecture with the update rule $x_{t+1} = \text{sigmoid}(Wx_t)$. We can write the t-th layer as:

$x_{t+1} = \text{sigmoid}(Wx_t)$

Let's compute the Jacobian for each time step:

$\frac{\partial x_{t+1}}{\partial x_t} = \frac{\partial \text{sigmoid}(Wx_t)}{\partial x_t} = W^T \cdot \text{diag}(\text{sigmoid}'(Wx_t))$

where $\text{diag}(\text{sigmoid}'(Wx_t))$ is a diagonal matrix with the derivative of the sigmoid function evaluated at Wx_t.

Now, let's find the input-output Jacobian for n time steps:

$\frac{\partial x_n}{\partial x_1} = \frac{\partial x_n}{\partial x_{n-1}} \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdots \frac{\partial x_2}{\partial x_1}$

We know that if $C = AB$, then $\sigma_{max}(C) \leq \sigma_{max}(A)\sigma_{max}(B)$. Applying this to the input-output Jacobian, we have:

$\sigma_{max}(\frac{\partial x_n}{\partial x_1}) \leq \sigma_{max}(\frac{\partial x_n}{\partial x_{n-1}}) \cdots \sigma_{max}(\frac{\partial x_2}{\partial x_1})$

We are given that the largest singular value of the weight matrix W is $\sigma_{max}(W) = \frac{1}{4}$. Since the sigmoid function's derivative is bounded as $0 \leq \text{sigmoid}'(z) \leq \frac{1}{4}$, the largest singular value of the diagonal matrix of derivatives, $\text{diag}(\text{sigmoid}'(Wx_t))$, is also bounded by $\frac{1}{4}$.

Now, let's find the largest singular value of each layerwise Jacobian:

$\sigma_{max}(\frac{\partial x_{t+1}}{\partial x_t}) \leq \sigma_{max}(W^T)\sigma_{max}(\text{diag}(\text{sigmoid}'(Wx_t))) \leq \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$

Using the above inequality, we can write the bound for the input-output Jacobian:

$\sigma_{max}(\frac{\partial x_n}{\partial x_1}) \leq (\frac{1}{16})^{n-1}$

Since the largest singular value is nonnegative, we have:

$0 \leq \sigma_{max}(\frac{\partial x_n}{\partial x_1}) \leq (\frac{1}{16})^{n-1}$

1.3 Self-Attention

In a self-attention layer (using scaled dot-product attention), the matrix of outputs is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

where $Q, K, V \in^{n \times d}$ are the query, key, and value matrices, n is the sequence length, and d_m is the embedding dimension.

1.3.1 Complexity of Self-Attention blue [0.5pt]

Recall from Lecture 8, the total cost for scaled dot-product attention scales quadratically with the sequence length n, i.e., n^2. We can generalize the attention equation for any similarity function $sim()$ to the following:

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

where the subscript of a matrix represents the i-th row as a vector. This is equivalent to the Softmax attention if we substitute $sim(q, k) = exp(\frac{q^T k}{\sqrt{d_k}})$. Note that for this generalized equation to be a valid attention equation, the only constraint on $sim()$ is that it need to be non-negative, which is true for all kernel functions $k(x, y) = \phi(x)^T\phi(y)$, for some feature mapping $\phi()$. Show that by applying kernel functions, attention can be calculated with linear complexity (i.e., n).

Hint: Sub in the kernel function for the similarity function into Eq 1. Group the terms based on their subscript (i.e., i and j).

ANS:(base on answers from ChatGPT-4)
To show that attention can be calculated with linear complexity using kernel functions, let's substitute the kernel function $k(x, y) = \phi(x)^T\phi(y)$ into the generalized attention equation:

$$\alpha_i = \frac{\sum_{j=1}^{n} k(Q_i, K_j)V_j}{\sum_{j=1}^{n} k(Q_i, K_j)} \tag{2}$$

Now, let's examine the numerator and denominator separately.

Numerator: $\sum_{j=1}^{n} k(Q_i, K_j)V_j = \sum_{j=1}^{n} \phi(Q_i)^T \phi(K_j)V_j$
Denominator: $\sum_{j=1}^{n} k(Q_i, K_j) = \sum_{j=1}^{n} \phi(Q_i)^T \phi(K_j)$

The key to reducing the complexity is to rewrite the sums in terms of the feature mappings $\phi(Q_i)$ and $\phi(K_j)$.

Let $P = \Phi(Q) \in^{n \times d'}$ and $L = \Phi(K) \in^{n \times d'}$, where d' is the dimensionality of the feature space, and $\Phi(Q)$ and $\Phi(K)$ represent the feature mappings of Q and K respectively. Then, the i-th row of P is $\phi(Q_i)$ and the j-th row of L is $\phi(K_j)$. We can rewrite the sums as matrix products:

Numerator: $PV^\top L^\top V = PAV$ where $A = V^\top L^\top$
Denominator: $PL^\top$

Now, we compute the matrix products $PL^\top$ and PAV separately. Notice that both of these matrix products have complexity nd', assuming d' is a constant (independent of n). After computing these products, we can then calculate the attention values α_i by performing element-wise division, which has complexity n. Therefore, the overall complexity of computing attention using kernel functions is n, which is linear in the sequence length n.

1.3.2 Linear Attention with SVD blue [0.5pt]

ANS:(base on answers from ChatGPT-4)

It has been empirically shown in Transformer models that the context mapping matrix $P = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$ often has a low rank. Show that if the rank of P is k and we already have access to the SVD of P, then it is possible to compute self-attention in nkd time.

Given that the context mapping matrix P has a low rank k, we can exploit this property to compute self-attention more efficiently. We are given that the SVD of P is already available: P= $U\Sigma \mathrm{V}^T$,

where $U \in^{n \times k}$, $\Sigma \in^{k \times k}$ is a diagonal matrix containing the k largest singular values of P, and $V^T \in^{k \times n}$. Recall that the self-attention is computed as:

Attention(Q,K,V)=PV

Using the SVD of P, we can rewrite the self-attention as:

Attention(Q,K,V)=PV=$U\Sigma V^T V$

Now, let's analyze the complexity of this computation:

1. Multiplying V^T by V: Since $V^T \in^{k\times n}$ and $V \in^{n\times d}$, this operation takes nkd time.
2. Multiplying Σ by the result of the previous step: Since Σ is a diagonal matrix, this operation takes kd time.
3. Multiplying U by the result of the previous step: Since $U \in^{n\times k}$ and the result of the previous step is a matrix of dimensions $k \times d$, this operation takes nkd time.

The overall complexity of computing self-attention using the SVD of the low-rank matrix P is nkd, which is more efficient than the quadratic complexity of the original self-attention computation.

1.4 Bellman Equation

The Bellman equation can be seen as a fix point equation to what's called the Bellman Operator. Given a policy π, the Bellman operators T^π for $V : \mathcal{B}(\mathcal{S}) \to \mathcal{B}(\mathcal{S})$ and T^π for $Q : \mathcal{B}(\mathcal{S} \times \mathcal{A}) \to \mathcal{B}(\mathcal{S} \times \mathcal{A})$ are defined as follows:

$$(T^\pi V)(s) r^\pi(S) + \gamma \int \mathcal{P}(s'|s,a)\pi(a|s)V(s') \tag{3}$$

$$(T^\pi Q)(s,a) r(s,a) + \gamma \int \mathcal{P}(s'|s,a)\pi(a'|s')Q(s',a') \tag{4}$$

for all $s \in \mathcal{S}$(for V) or all $(s,a) \in \mathcal{S} \times \mathcal{A}$(for Q).

The Bellman operators have two important properties, 1) monotonicity and 2) γ-contraction. These properties give us many guarantees such as applying the operator repeatedly will converge to an unique and optimal solution, which is what allow us to show RL algorithms such as Q-Learning converges (under certain additional assumptions, but we won't go over them here). In this section, we will show that the Bellman operator indeed have these two properties.

1.4.1

blue [0.5pt] Show that the Bellman operator (on V) has the monotonicity property. i.e., show that for a fixed policy π, if $V_1, V_2 \in \mathcal{B}(\mathcal{S})$, and $V_1(s) \le V_2(s)$ for all $s \in \mathcal{S}$, then we have

$$T^\pi V_1 \le T^\pi V_2 \tag{5}$$

ANS:(base on answers from ChatGPT-4)
To show that the Bellman operator (on V) has the monotonicity property, we need to show that for a fixed policy π, if $V_1, V_2 \in \mathcal{B}(\mathcal{S})$, and $V_1(s) \leq V_2(s)$ for all $s \in \mathcal{S}$, then we have

$$T^\pi V_1 \leq T^\pi V_2 \tag{6}$$

Let's consider the Bellman operators T^π for V_1 and V_2:

$$(T^\pi V_1)(s) = r^\pi(s) + \gamma \int \mathcal{P}(s'|s,a)\pi(a|s)V_1(s') \; (T^\pi V_2)(s) \quad = r^\pi(s) + \gamma \int \mathcal{P}(s'|s,a)\pi(a|s)V_2(s') \tag{7}$$

We know that $V_1(s) \leq V_2(s)$ for all $s \in \mathcal{S}$, and we want to show that $(T^\pi V_1)(s) \leq (T^\pi V_2)(s)$ for all $s \in \mathcal{S}$. Let's subtract $(T^\pi V_1)(s)$ from $(T^\pi V_2)(s)$:

$$(T^\pi V_2)(s) - (T^\pi V_1)(s) = \gamma \int \mathcal{P}(s'|s,a)\pi(a|s)(V_2(s') - V_1(s')) \tag{8}$$

Since $V_1(s) \leq V_2(s)$ for all $s \in \mathcal{S}$, we have $(V_2(s') - V_1(s')) \geq 0$ for all $s' \in \mathcal{S}$. Additionally, both $\mathcal{P}(s'|s,a)$ and $\pi(a|s)$ are non-negative probabilities. Therefore, the integrand is non-negative, and we can conclude that:

$$(T^\pi V_2)(s) - (T^\pi V_1)(s) \geq 0 \tag{9}$$

This inequality holds for all $s \in \mathcal{S}$, and thus, we have shown that the Bellman operator (on V) has the monotonicity property:

$$T^\pi V_1 \leq T^\pi V_2 \tag{10}$$

1.4.2

[0.5pt] Show that the Bellman operator is a γ-contraction mapping with the supremum norm (on Q). i.e. show that for a discount factor γ and $Q_1, Q_2 \in \mathcal{B}(\mathcal{S} \times \mathcal{A})$, we have

$$\|T^\pi(Q_1) - T^\pi(Q_2)\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty \tag{11}$$

Recall from your math classes, the supremum norm (on Q) is as follows:

$$\|Q\|_\infty = \sup_{(s,a)\in\mathcal{S}\times\mathcal{A}} |Q(s,a)| \tag{12}$$

Hint: for some function f, we have the following. For this question, you can think about what is P and f in our case.

$$\begin{aligned} \left| \int P(x)f(x) \right| &\leq \int |P(x)f(x)| = \int |P(x)| \cdot |f(x)| \\ &\leq \int P(x) \cdot \sup_{x\in\mathcal{X}} |f(x)| \\ &= \sup_{x\in\mathcal{X}} |f(x)| \int P(x) = \|f\|_\infty \end{aligned} \tag{13}$$

where in the last line we used the fact $\int P(x) = 1$

ANS:(base on answers from ChatGPT-4)
To show that the Bellman operator is a γ-contraction mapping with the supremum norm (on Q), we need to show that for a discount factor γ and $Q_1, Q_2 \in \mathcal{B}(\mathcal{S} \times \mathcal{A})$, we have:

$$|T^\pi(Q_1) - T^\pi(Q_2)|\infty \leq \gamma |Q_1 - Q_2|\infty$$

Let's consider the Bellman operators T^π for Q_1 and Q_2:

$$\begin{aligned} (T^\pi Q_1)(s,a) &= r(s,a) + \gamma \int \mathcal{P}(s'|s,a)\pi(a'|s')Q_1(s',a') \\ (T^\pi Q_2)(s,a) &= r(s,a) + \gamma \int \mathcal{P}(s'|s,a)\pi(a'|s')Q_2(s',a') \end{aligned}$$

Now we'll look at the absolute difference between these two equations:

$$|(T^\pi Q_1)(s,a) - (T^\pi Q_2)(s,a)| = \gamma \left| \int \mathcal{P}(s'|s,a)\pi(a'|s')(Q_1(s',a') - Q_2(s',a')) \right|$$

Using the hint provided, we have:

$$\begin{aligned}
\left| \int \mathcal{P}(s'|s,a)\pi(a'|s')(Q_1(s',a') - Q_2(s',a')) \right| &\leq \int \mathcal{P}(s'|s,a)\pi(a'|s') \cdot |Q_1(s',a') - Q_2(s',a')| \\
&\leq \int \mathcal{P}(s'|s,a)\pi(a'|s') \cdot |Q_1 - Q_2|\infty \\
&= |Q_1 - Q_2|\infty \int \mathcal{P}(s'|s,a)\pi(a'|s') \\
&= |Q_1 - Q_2|_\infty
\end{aligned}$$

Since this inequality holds for all $(s,a) \in \mathcal{S} \times \mathcal{A}$, we can conclude that:

$$|T^\pi(Q_1) - T^\pi(Q_2)|\infty \leq \gamma |Q_1 - Q_2|\infty$$

1.4.3

[0.5pt] For this question, you may assume knowledge of the reward function $r(s,a)$ and transition probability function $p(s'|s,a)$, where s' is the next state.

ANS:(base on answers from ChatGPT-4)

1. Give a definition of $v_*(s)$ in terms of $q_*(s,a)$.

   $v_*(s)$ represents the optimal state-value function, which is the expected return when starting from state s and following the optimal policy. $q_*(s,a)$ represents the optimal action-value function, which is the expected return when starting from state s, taking action a, and following the optimal policy thereafter. We can define $v_*(s)$ in terms of $q_*(s,a)$ as :

   $v_*(s) = max_a(q_*(s,a))$.

   This equation states that the optimal state-value function at a given state s is the maximum of the optimal action-value function over all possible actions a in that state. This is because, under the optimal policy, we choose the action that leads to the highest expected return.

2. Give a definition of $q_*(s, a)$ in terms of $v_*(s)$.

   $q_(s, a)$ represents the optimal action-value function, which is the expected return when starting from state s, taking action a, and following the optimal policy thereafter. $v_(s)$ represents the optimal state-value function, which is the expected return when starting from state s and following the optimal policy. We can define $q_(s, a)$ in terms of $v_(s)$ as:

   $q_*(s, a) = r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) v_*(s')$

   This equation states that the optimal action-value function for a given state-action pair (s, a) is the immediate reward $r(s, a)$ received for taking action a in state s, plus the expected discounted return from the next state s', where the expectation is taken over all possible next states, weighted by their transition probabilities $p(s'|s, a)$, and we use the optimal state-value function $v_*(s')$ to estimate the value of the next state.

3. Give a definition of a_* in terms of $q_*(s, a)$.

   a represents the optimal action to take in a given state s. Given the optimal action-value function $q_(s, a)$, we can define the optimal action a_* as :

   $\mathrm{a}_* = \arg\max_{a \in A} q_*(s, a)$

   This equation states that the optimal action a for a given state s is the action that maximizes the optimal action-value function $q_(s, a)$ over all possible actions a in that state. In other words, the optimal action is the one that leads to the highest expected return when following the optimal policy.

4. Give a definition of a_* in terms of $v_*(s)$.

   To define the optimal action a in terms of $v_(s)$, we first need to relate $v_*(s)$ to the action-value function $q(s, a)$ indicated above and define the optimal action a in terms of $v_(s)$ as follows:

   $\mathrm{a}_* = \arg\max_{a \in A} r(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) v_*(s')$

   This equation states that the optimal action a for a given state s is the action that maximizes the sum of the immediate reward $r(s, a)$ and the expected discounted return from the next state s', where the expectation is taken over all possible next states, weighted by their transition probabilities $p(s'|s, a)$, and we use the optimal state-value function $v_(s')$ to estimate the value of the next state.

Programming Assignment

2 Graph Convolution Networks

2.1 Implementation of Graph Convolution Layer

```
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.layer = 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 ##################################
        return  torch.spmm(adj, self.layer(input))

        ##############################################################
```

2.2 Implementation of Graph Convolution Network

```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.gcn_layer1 = GraphConvolution(nfeat, n_hidden, bias)
        self.gcn_layer2 = GraphConvolution(n_hidden, n_classes, bias)
        self.dropout_layer = nn.Dropout(dropout)
        self.act_layer = 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 ##################################
        out = self.gcn_layer1(x, adj)
        out = self.act_layer(out)
        out = self.dropout_layer(out)
        out = self.gcn_layer2(out, adj)
        return out
```

2.3 Train your Graph Convolution Network

You can just screenshot the last 10 epochs with test set results.

```
Epoch: 0090 loss_train: 0.8286 acc_train: 0.8286 loss_val: 1.1313 acc_val: 0.6223 time: 0.0028s
Epoch: 0091 loss_train: 0.7919 acc_train: 0.8286 loss_val: 1.1238 acc_val: 0.6254 time: 0.0034s
Epoch: 0092 loss_train: 0.7797 acc_train: 0.8429 loss_val: 1.1160 acc_val: 0.6293 time: 0.0029s
Epoch: 0093 loss_train: 0.7539 acc_train: 0.8786 loss_val: 1.1086 acc_val: 0.6336 time: 0.0036s
Epoch: 0094 loss_train: 0.8075 acc_train: 0.8286 loss_val: 1.1020 acc_val: 0.6386 time: 0.0028s
Epoch: 0095 loss_train: 0.7681 acc_train: 0.8500 loss_val: 1.0956 acc_val: 0.6421 time: 0.0029s
Epoch: 0096 loss_train: 0.7314 acc_train: 0.8286 loss_val: 1.0892 acc_val: 0.6452 time: 0.0028s
Epoch: 0097 loss_train: 0.7305 acc_train: 0.8786 loss_val: 1.0826 acc_val: 0.6491 time: 0.0028s
Epoch: 0098 loss_train: 0.7108 acc_train: 0.8786 loss_val: 1.0760 acc_val: 0.6515 time: 0.0030s
Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703 acc_val: 0.6488 time: 0.0029s
Epoch: 0100 loss_train: 0.7307 acc_train: 0.8286 loss_val: 1.0643 acc_val: 0.6515 time: 0.0031s
Optimization Finished!
Total time elapsed: 3.4717s
Test set results: loss= 1.0643 accuracy= 0.6515
```

2.4 Implementation of Graph Attention Layer

```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_hidden * self.n_heads, 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)
        ######################################################
    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_i = s.repeat(n_nodes, 1, 1)
        s_j = s.repeat_interleave(n_nodes, dim=0)
        s_i_s_j = torch.cat([s_i, s_j], dim=-1)
        s_i_s_j = s_i_s_j.view((n_nodes, n_nodes, self.n_heads, self.n_hidden * 2))

        attention = self.attention(s_i_s_j)
        act_out = self.activation(attention).squeeze(-1)

        mask = act_out.masked_fill((adj_mat.unsqueeze(dim=-1) == 0), -np.inf)

        a = self.dropout_layer(self.softmax(mask))
        ######################################################################
```

```python
        # 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 ########################################
            return h_prime.reshape(n_nodes, self.n_heads * self.n_hidden)
            ######################################################################

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

2.5 Train your Graph Attention Network

Screenshot the last 10 epochs with test set results.

```
Epoch: 0090 loss_train: 0.9849 acc_train: 0.8000 loss_val: 1.0863 acc_val: 0.7706 time: 0.2335s
Epoch: 0091 loss_train: 1.0129 acc_train: 0.8000 loss_val: 1.0789 acc_val: 0.7714 time: 0.2330s
Epoch: 0092 loss_train: 1.0283 acc_train: 0.8214 loss_val: 1.0715 acc_val: 0.7741 time: 0.2336s
Epoch: 0093 loss_train: 0.9859 acc_train: 0.7929 loss_val: 1.0640 acc_val: 0.7745 time: 0.2332s
Epoch: 0094 loss_train: 0.9678 acc_train: 0.7857 loss_val: 1.0564 acc_val: 0.7757 time: 0.2326s
Epoch: 0095 loss_train: 0.9448 acc_train: 0.8286 loss_val: 1.0488 acc_val: 0.7761 time: 0.2343s
Epoch: 0096 loss_train: 0.9939 acc_train: 0.7571 loss_val: 1.0417 acc_val: 0.7761 time: 0.2325s
Epoch: 0097 loss_train: 0.9640 acc_train: 0.7857 loss_val: 1.0348 acc_val: 0.7753 time: 0.2329s
Epoch: 0098 loss_train: 0.9710 acc_train: 0.8143 loss_val: 1.0280 acc_val: 0.7761 time: 0.2335s
Epoch: 0099 loss_train: 0.8724 acc_train: 0.8357 loss_val: 1.0214 acc_val: 0.7776 time: 0.2333s
Epoch: 0100 loss_train: 0.9530 acc_train: 0.7714 loss_val: 1.0150 acc_val: 0.7788 time: 0.2324s
Optimization Finished!
Total time elapsed: 23.7049s
Test set results: loss= 1.0150 accuracy= 0.7788
```

2.6 Compare your models

Compare the evaluation results for Vanilla GCN and GAT. Comment on the discrepancy in their performance (if any) and briefly explain why you think it's the case (in 1-2 sentences).

ANS:
The evaluation time by GAT increases , but this also leads to a slightly lower loss and higher accuracy from 0.6515 by GCN to 0.7788. This case is caused by the attention layer involved in GAT, which enables more information can be gathered by taking each node's features, with the cost of time.

3 Deep Q-Learning Network (DQN)

3.1 Implementation of ϵ greedy

get action:

```
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 torch.rand(1,).item() < epsilon:
        # Exploration: choose a random action from the action space
        action = torch.randint(0, action_space_len,(1,))
    else:
        # Exploitation: choose the action with the highest Q-value
        action = torch.max(Qp, axis=0)[1]

    return action
```

3.2 Implementation of DQN training step

train:

```
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
    exp_return = torch.max(model.policy_net(state), axis=1)[0]

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

    # TODO: compute the loss
    loss = model.loss_fn(exp_return, tar_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()
```

hyperparameters' value:

```
# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 200
memory = ExperienceReplay(exp_replay_size)
episodes = 10000
epsilon = 1 # episilon start from 1 and decay gradually.
```

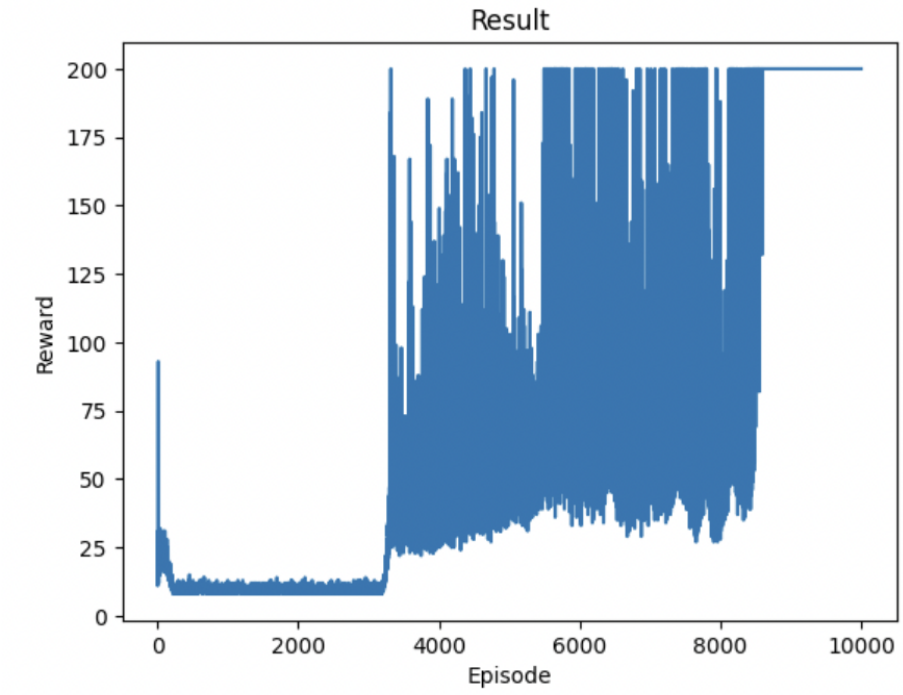
epsilon decay rule:

```
        # TODO: add epsilon decay rule here!
        if epsilon > 0.05:
          epsilon = epsilon * 0.95

        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")
```

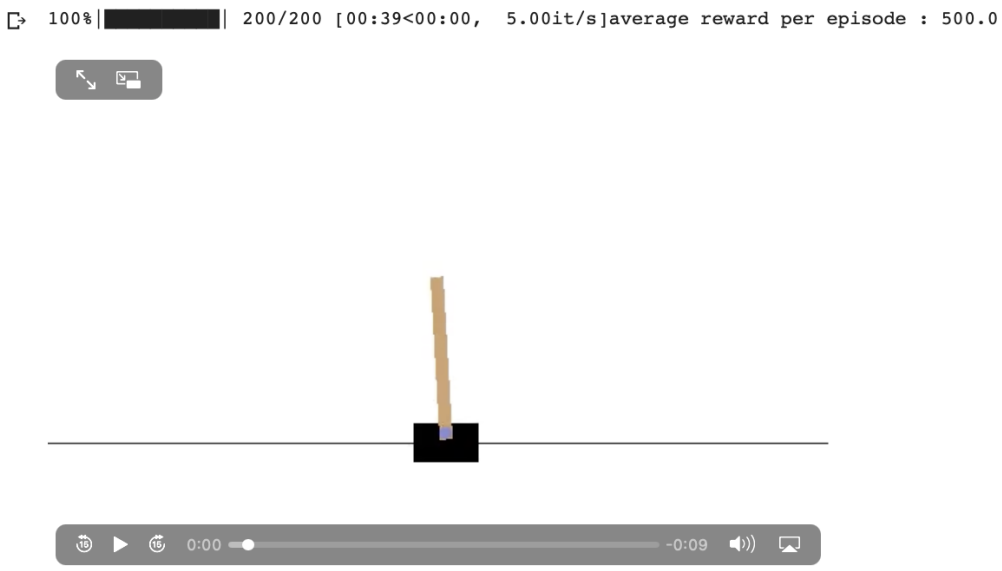
```
100%|██████████| 10000/10000 [01:46<00:00, 93.80it/s]Saving trained model
```

3.3 Train your DQN Agent

reward curve:



end time frame of the cartpole video(9 seconds here):



Analysis: (not exceed 3 sentences)
The training time for the latter episodes is longer than the early episodes, as the agent is getting better and better at playing the game and thus each episode takes longer. By the reward vs. episode plot, the reward can stabilize at 200 in later episodes. The video is 9s with a self-balancing pole.