

# 1 RNNs and Self Attention

## 1.1.2 Effect of Activation

### Solution:

Let's analyze the network with varying weights  $w_i$  for each layer and the ReLU activation function. The function  $f(x)$  can be written as:

$$f(x) = \text{ReLU}(w_n \cdot \text{ReLU}(w_{n-1} \cdot \dots \cdot \text{ReLU}(w_1 \cdot x) \dots)) \quad (1)$$

Now let's compute the derivative of  $f(x)$  with respect to  $x$  using the chain rule:

$$\frac{\partial f(x)}{\partial x} = w_n \cdot \frac{\partial \text{ReLU}}{\partial x_n} \cdot w_{n-1} \cdot \frac{\partial \text{ReLU}}{\partial x_{n-1}} \cdots w_1 \cdot \frac{\partial \text{ReLU}}{\partial x_1} \quad (2)$$

Recall that the derivative of the ReLU function is:

$$\frac{d \text{ReLU}(x)}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases} \quad (3)$$

So, the gradient is:

$$\frac{\partial f(x)}{\partial x} = w_n \cdot \frac{\partial \text{ReLU}}{\partial x_n} \cdot w_{n-1} \cdot \frac{\partial \text{ReLU}}{\partial x_{n-1}} \cdots w_1 \cdot \frac{\partial \text{ReLU}}{\partial x_1} \quad (4)$$

Now, consider the bounds on the magnitude of the gradient:

- Lower bound: If any of the  $\frac{\partial \text{ReLU}}{\partial x_i}$  terms are 0, then the gradient is 0. Thus, the lower bound is 0.
- Upper bound: If all the  $\frac{\partial \text{ReLU}}{\partial x_i}$  terms are 1, the gradient is  $|w_1 w_2 \cdots w_n|$ . In this case, the upper bound depends on the weights.

$$0 \leq \left| \frac{\partial f(x)}{\partial x} \right| \leq |w_1 w_2 \cdots w_n| \quad (5)$$

Whether the gradients vanish or explode depends on the product of the weights. If the product of the weights is significantly smaller than 1, the gradients may vanish; if the product of the weights is significantly larger than 1, the gradients may explode.

DO

### 1.1.2 Effect of Activation - Different weights [0.5pt] [Type 1]

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.



Let's analyze the network with varying weights  $w_i$  for each layer and the ReLU activation function. The function  $f(x)$  can be written as:

$$f(x) = \text{ReLU}(w_n \cdot \text{ReLU}(w_{n-1} \cdot \dots \text{ReLU}(w_1 \cdot x) \dots))$$

Now let's compute the derivative of  $f(x)$  with respect to  $x$  using the chain rule:

$$\frac{\partial f(x)}{\partial x} = w_n \cdot \frac{\partial \text{ReLU}}{\partial x_n} \cdot w_{n-1} \cdot \frac{\partial \text{ReLU}}{\partial x_{n-1}} \cdots w_1 \cdot \frac{\partial \text{ReLU}}{\partial x_1}$$

Recall that the derivative of the ReLU function is:

$$\frac{d\text{ReLU}(x)}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

So, the gradient is:

$$\frac{\partial f(x)}{\partial x} = w_n \cdot \frac{\partial \text{ReLU}}{\partial x_n} \cdot w_{n-1} \cdot \frac{\partial \text{ReLU}}{\partial x_{n-1}} \cdots w_1 \cdot \frac{\partial \text{ReLU}}{\partial x_1}$$

Now, consider the bounds on the magnitude of the gradient:

- Lower bound: If any of the  $\frac{\partial \text{ReLU}}{\partial x_i}$  terms are 0, then the gradient is 0. Thus, the lower bound is 0.
- Upper bound: If all the  $\frac{\partial \text{ReLU}}{\partial x_i}$  terms are 1, the gradient is  $|w_1 w_2 \cdots w_n|$ . In this case, the upper bound depends on the weights.

$$0 \leq \left| \frac{\partial f(x)}{\partial x} \right| \leq |w_1 w_2 \cdots w_n|$$

Whether the gradients vanish or explode depends on the product of the weights. If the product of the weights is significantly smaller than 1, the gradients may vanish; if the product of the weights is significantly larger than 1, the gradients may explode.

### 1.2.1 Gradient through RNN

**Solution:**

Let's first write the input-output Jacobian for the given RNN:

$$\frac{\partial x_n}{\partial x_1} = \frac{\partial x_n}{\partial x_{n-1}} \cdot \frac{\partial x_{n-1}}{\partial x_{n-2}} \cdots \frac{\partial x_2}{\partial x_1} \quad (6)$$

The derivative of the sigmoid function is:

$$\frac{d \text{sigmoid}(x)}{dx} = \text{sigmoid}(x)(1 - \text{sigmoid}(x)) \quad (7)$$

Therefore, the layerwise Jacobian is given by:

$$\frac{\partial x_{t+1}}{\partial x_t} = W \cdot \text{sigmoid}(Wx_t) \cdot (1 - \text{sigmoid}(Wx_t)) \quad (8)$$

Now we will apply the hint and find the largest singular value of the input-output Jacobian:

$$\sigma_{\max} \left( \frac{\partial x_n}{\partial x_1} \right) \leq \sigma_{\max} \left( \frac{\partial x_n}{\partial x_{n-1}} \right) \cdot \sigma_{\max} \left( \frac{\partial x_{n-1}}{\partial x_{n-2}} \right) \cdots \sigma_{\max} \left( \frac{\partial x_2}{\partial x_1} \right) \quad (9)$$

Since  $\sigma_{\max}(W) = \frac{1}{4}$ , we have:

$$\sigma_{\max} \left( \frac{\partial x_{t+1}}{\partial x_t} \right) \leq \frac{1}{4} \cdot \sigma_{\max} (\text{sigmoid}(Wx_t) \cdot (1 - \text{sigmoid}(Wx_t))) \quad (10)$$

The derivative of the sigmoid function is always between 0 and 1/4:

$$0 \leq \text{sigmoid}(x)(1 - \text{sigmoid}(x)) \leq \frac{1}{4} \quad (11)$$

Thus, we have:

$$\sigma_{\max} \left( \frac{\partial x_{t+1}}{\partial x_t} \right) \leq \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16} \quad (12)$$

Now, we can find the largest singular value of the input-output Jacobian:

$$\sigma_{\max} \left( \frac{\partial x_n}{\partial x_1} \right) \leq \left( \frac{1}{16} \right)^{n-1} \quad (13)$$

Finally, we have the desired bound:

$$0 \leq \sigma_{\max} \left( \frac{\partial x_n}{\partial x_1} \right) \leq \left( \frac{1}{16} \right)^{n-1} \quad (14)$$

This bound shows that the largest singular value of the input-output Jacobian is exponentially decreasing with the number of time steps, indicating that the gradients could vanish.

#### DO 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  $M \in \mathbb{R}^{m \times n}$  can be written as  $M=U \Sigma V^T$  where  $U \in \mathbb{R}^{m \times m}$  and  $V \in \mathbb{R}^{n \times n}$  are square orthogonal matrices, and  $\Sigma \in \mathbb{R}^{m \times n}$  is a rectangular diagonal matrix with nonnegative entries on the diagonal (i.e.  $\Sigma_{ii} \geq 0$  for  $i \in \{1, \dots, \min(m, n)\}$  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 [0.5pt] [Type 1]

Let say we have a very simple RNN-like architecture that computes

$x_{t+1} = \text{operatorname}{\text{sigmoid}}(\text{left}(W x_t \text{right}))$ . 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  $\sigma_{\max}$

$\sigma_{\max}(W) = \frac{1}{4}$ . Show that the largest singular value of the input-output Jacobian has the following bound:

\$\$

$$0 \leq \sigma_{\max} \left( \max \left( \left| \frac{\partial x_n}{\partial x_{n-1}} \right|, \left| \frac{\partial x_{n-1}}{\partial x_{n-2}} \right|, \dots, \left| \frac{\partial x_2}{\partial x_1} \right| \right) \right) \leq \frac{1}{16}$$

\$\$

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



Let's first write the input-output Jacobian for the given RNN:

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

The derivative of the sigmoid function is:

$$\frac{d \text{sigmoid}(x)}{dx} = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

Therefore, the layerwise Jacobian is given by:

$$\frac{\partial x_{t+1}}{\partial x_t} = W \cdot \text{sigmoid}(Wx_t) \cdot (1 - \text{sigmoid}(Wx_t))$$

Now we will apply the hint and find the largest singular value of the input-output Jacobian:

$$\sigma_{\max} \left( \frac{\partial x_n}{\partial x_1} \right) \leq \sigma_{\max} \left( \frac{\partial x_n}{\partial x_{n-1}} \right) \cdot \sigma_{\max} \left( \frac{\partial x_{n-1}}{\partial x_{n-2}} \right) \cdots \sigma_{\max} \left( \frac{\partial x_2}{\partial x_1} \right)$$

Since  $\sigma_{\max}(W) = \frac{1}{4}$ , we have:

$$\sigma_{\max} \left( \frac{\partial x_{t+1}}{\partial x_t} \right) \leq \frac{1}{4} \cdot \sigma_{\max} (\text{sigmoid}(Wx_t) \cdot (1 - \text{sigmoid}(Wx_t)))$$

The derivative of the sigmoid function is always between 0 and 1/4:

$$0 \leq \text{sigmoid}(x)(1 - \text{sigmoid}(x)) \leq \frac{1}{4}$$

Thus, we have:

$$\sigma_{\max} \left( \frac{\partial x_{t+1}}{\partial x_t} \right) \leq \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$$

Now, we can find the largest singular value of the input-output Jacobian:

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

Finally, we have the desired bound:

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

### 1.3.1 Complexity of Self-Attention

**Solution:**

To show that attention can be calculated with linear complexity using kernel functions, we substitute the kernel function for the similarity function in the generalized attention equation:

$$\alpha_i = \frac{\sum_{j=1}^n k(Q_i, K_j)V_j}{\sum_{j=1}^n k(Q_i, K_j)} \quad (15)$$

Now let's represent the kernel function using a feature mapping  $\phi$ :

$$k(x, y) = \phi(x)^T \phi(y) \quad (16)$$

Substitute this expression for the kernel function in the attention equation:

$$\alpha_i = \frac{\sum_{j=1}^n \phi(Q_i)^T \phi(K_j)V_j}{\sum_{j=1}^n \phi(Q_i)^T \phi(K_j)} \quad (17)$$

Now, group the terms based on their subscript:

$$\alpha_i = \frac{\phi(Q_i)^T \sum_{j=1}^n \phi(K_j)V_j}{\phi(Q_i)^T \sum_{j=1}^n \phi(K_j)} \quad (18)$$

Define the following two vectors:

$$\begin{aligned} A &= \sum_{j=1}^n \phi(K_j)V_j \\ B &= \sum_{j=1}^n \phi(K_j) \end{aligned} \quad (19)$$

Both  $A$  and  $B$  can be computed with linear complexity since they involve a single summation over the sequence length  $n$ . Now, the attention equation can be written as:

$$\alpha_i = \frac{\phi(Q_i)^T A}{\phi(Q_i)^T B} \quad (20)$$

This equation shows that attention can be calculated with linear complexity, i.e.,  $\mathcal{O}(n)$ , using kernel functions.

DO

#### 1.3.1 Complexity of Self-Attention [0.5pt] [Type 1]

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

\$\$

```
\alpha_i=\frac{\sum_{j=1}^n \operatorname{sim}\left(Q_i, K_j\right) V_j}{\operatorname{sim}\left(Q_i, K_j\right)}
```

\$\$

where the subscript of a matrix represents the  $i$ -th row as a vector. This is equivalent to the Softmax attention if we substitute  $\operatorname{sim}(q, k)=\exp \left(\frac{q^T k}{\sqrt{d_k}}\right)$ . Note that for this generalized equation to be a valid attention equation, the only constraint on  $\operatorname{sim}()$  is that it needs 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.,  $\mathcal{O}(n)$ ).

Hint: Sub in the kernel function for the similarity function into Eq 1.1. Group the terms based on their subscript (i.e.,  $\$i\$$  and  $\$j\$$  ).



To show that attention can be calculated with linear complexity using kernel functions, we substitute the kernel function for the similarity function in the generalized attention equation:

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

Now let's represent the kernel function using a feature mapping  $\$|\phi|$ :

$$k(x, y) = \phi(x)^T \phi(y)$$

Substitute this expression for the kernel function in the attention equation:

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

Now, group the terms based on their subscript:

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

Define the following two vectors:

$$A = \sum_{j=1}^n \phi(K_j) V_j$$

$$B = \sum_{j=1}^n \phi(K_j)$$

Both A and B can be computed with linear complexity since they involve a single summation over the sequence length n. Now, the attention equation can be written as:

$$\alpha_i = \frac{\phi(Q_i)^T A}{\phi(Q_i)^T B}$$

This equation shows that attention can be calculated with linear complexity, i.e.,  $\mathcal{O}(n)$ , using kernel functions.

### 1.3.2 Linear Attention with SVD

**Solution:**

Given that the context mapping matrix  $P$  has a low rank  $k$ , we can apply Singular Value Decomposition (SVD) to  $P$ :

$$P = U\Sigma V^T \quad (21)$$

Here,  $U \in \mathbb{R}^{n \times k}$ ,  $\Sigma \in \mathbb{R}^{k \times k}$  (a diagonal matrix with the  $k$  singular values), and  $V^T \in \mathbb{R}^{k \times n}$ .

The self-attention can be computed as:

$$\text{Attention}(Q, K, V) = PV \quad (22)$$

Substitute the SVD decomposition of  $P$ :

$$\text{Attention}(Q, K, V) = U\Sigma V^T V \quad (23)$$

Since  $V^T V = I_k$  (identity matrix of size  $k \times k$ ) due to the orthogonality of  $V$ , we have:

$$\text{Attention}(Q, K, V) = U\Sigma \quad (24)$$

Now let's analyze the complexity of computing self-attention using this method:

1. Computing  $U\Sigma$ : This step takes  $\mathcal{O}(nkd)$  time, as we multiply an  $n \times k$  matrix by a  $k \times k$  diagonal matrix.
2. Computing  $V$ : We are given the SVD of  $P$ , so we don't need to compute it, and there's no additional complexity in this step.

Thus, the overall complexity of computing self-attention using the SVD of the low-rank context mapping matrix  $P$  is  $\mathcal{O}(nkd)$ .

DO

### 1.3.2 Linear Attention with SVD [0.5pt] [Type 2]

It has been empirically shown in Transformer models that the context mapping matrix  $P = \text{softmax } \left( \frac{Q K^T}{\sqrt{d_k}} \right)$  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  $\mathcal{O}(n k d)$  time.



Given that the context mapping matrix  $P$  has a low rank  $k$ , we can apply Singular Value Decomposition (SVD) to  $P$ :

$$P = U \Sigma V^T$$

Here,  $U \in \mathbb{R}^{n \times k}$ ,  $\Sigma \in \mathbb{R}^{k \times k}$  (a diagonal matrix with the  $k$  singular values), and  $V^T \in \mathbb{R}^{k \times n}$ .

The self-attention can be computed as:

$$\text{Attention}(Q, K, V) = P V$$

Substitute the SVD decomposition of  $P$ :

$$\text{Attention}(Q, K, V) = U \Sigma V^T V$$

Since  $V^T V = I_k$  (identity matrix of size  $k \times k$ ) due to the orthogonality of  $V$ , we have:

$$\text{Attention}(Q, K, V) = U \Sigma$$

Now let's analyze the complexity of computing self-attention using this method:

1. Computing  $\Sigma$ : This step takes  $\mathcal{O}(nkd)$  time, as we multiply an  $n \times k$  matrix by a  $k \times k$  diagonal matrix.
2. Computing  $V$ : We are given the SVD of  $P$ , so we don't need to compute it, and there's no additional complexity in this step.

Thus, the overall complexity of computing self-attention using the SVD of the low-rank context mapping matrix  $P$  is  $\mathcal{O}(nkd)$ .

### 2.1.1 Bellman Operator

### Solution:

To show the monotonicity property of the Bellman operator on  $V$ , we need to show that for a fixed policy  $\pi$ , 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

$$T^\pi V_1(s) \leq T^\pi V_2(s) \quad (25)$$

for all  $s \in S$ .

Recall that the Bellman operator for  $V$  is defined as:

$$(T^\pi V)(s) \triangleq r^\pi(s) + \gamma \int \mathcal{P}(s' \mid s, a) \pi(a \mid s) V(s') \quad (26)$$

Let's compare the Bellman operator applied to  $V_1$  and  $V_2$ :

$$\begin{aligned} T^\pi V_1(s) &= r^\pi(s) + \gamma \int \mathcal{P}(s' | s, a) \pi(a | s) V_1(s') \\ T^\pi V_2(s) &= r^\pi(s) + \gamma \int \mathcal{P}(s' | s, a) \pi(a | s) V_2(s') \end{aligned} \tag{27}$$

Since we are given that  $V_1(s) \leq V_2(s)$  for all  $s \in \mathcal{S}$ , we have

$$V_1(s') \leq V_2(s') \quad (28)$$

for all  $s' \in \mathcal{S}$ .

Now, we can see that

$$\gamma \int \mathcal{P}(s' | s, a) \pi(a | s) V_1(s') \leq \gamma \int \mathcal{P}(s' | s, a) \pi(a | s) V_2(s') \quad (29)$$

because the probability distribution  $\mathcal{P}(s' | s, a)\pi(a | s)$  is non-negative and the inequality holds for all  $s' \in \mathcal{S}$ .

Thus, we can conclude that

$$T^\pi V_1(s) \leq T^\pi V_2(s) \quad (30)$$

for all  $s \in S$ , proving the monotonicity property of the Bellman operator on  $V$ .

## 2 Reinforcement Learning

## 2.1 Bellman Equation

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

\$\$

```
\begin{gathered}
```

$$\left| \left( T^{\wedge} \pi V \right) (s) \right| \triangleq r^{\wedge} \pi(S) + \gamma \int \mathcal{P}(s^{\wedge} \mid s, a) \pi(a \mid s) V \left( s^{\wedge} \right)$$

$$\left| \left( T^{\wedge} \pi Q \right) (s, a) \right| \triangleq \gamma \int \mathcal{P}(s' | s, a) \left[ \pi(a' | s') Q(s', a') \right] ds' da'$$

\$\$

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)  $\gamma$ -contraction. These properties give us many guarantees such as applying the operator repeatedly will converge to a unique and optimal solution, which is what allows 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.

### 2.1.1

[0.5pt] Show that the Bellman operator (on  $V$ ) has the monotonicity property. i.e., show that for a fixed policy  $\pi$ , 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

$\$$

$$T^\pi V_1 \leq T^\pi V_2$$

$\$\$$



To show the monotonicity property of the Bellman operator on  $V$ , we need to show that for a fixed policy  $\pi$ , 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

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

for all  $s \in \mathcal{S}$ .

Recall that the Bellman operator for  $V$  is defined as:

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

Let's compare the Bellman operator applied to  $V_1$  and  $V_2$ :

$$\begin{aligned} T^\pi V_1(s) &= r^\pi(s) + \gamma \int \mathcal{P}(s' | s, a) \pi(a | s) V_1(s') \\ T^\pi V_2(s) &= r^\pi(s) + \gamma \int \mathcal{P}(s' | s, a) \pi(a | s) V_2(s') \end{aligned}$$

Since we are given that  $V_1(s) \leq V_2(s)$  for all  $s \in \mathcal{S}$ , we have

$$V_1(s') \leq V_2(s')$$

for all  $s' \in \mathcal{S}$ .

Now, we can see that

$$\gamma \int \mathcal{P}(s' | s, a) \pi(a | s) V_1(s') \leq \gamma \int \mathcal{P}(s' | s, a) \pi(a | s) V_2(s')$$

because the probability distribution  $\mathcal{P}(s' | s, a)$  is non-negative and the inequality holds for all  $s' \in \mathcal{S}$ .

Thus, we can conclude that

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

for all  $s \in \mathcal{S}$ , proving the monotonicity property of the Bellman operator on  $V$ .

## 2.1.2 $\gamma$ -contraction mapping

**Solution:**

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

$$\|T^\pi(Q_1) - T^\pi(Q_2)\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty \quad (31)$$

Recall that the Bellman operator for  $Q$  is defined as:

$$(T^\pi Q)(s, a) \triangleq r(s, a) + \gamma \int \mathcal{P}(s' | s, a) \pi(a' | s') Q(s', a') \quad (32)$$

Let's consider the difference between the Bellman operators applied to  $Q_1$  and  $Q_2$  for any  $(s, a) \in \mathcal{S} \times \mathcal{A}$ :

$$\begin{aligned} |T^\pi Q_1(s, a) - T^\pi Q_2(s, a)| &= \left| r(s, a) + \gamma \int \mathcal{P}(s' | s, a) \pi(a' | s') Q_1(s', a') - r(s, a) - \gamma \int \mathcal{P}(s' | s, a) \pi(a' | s') Q_2(s', a') \right| \\ &= \left| \gamma \int \mathcal{P}(s' | s, a) \pi(a' | s') [Q_1(s', a') - Q_2(s', a')] \right| \end{aligned} \quad (33)$$

Now, we can apply the hint. Let  $P(x) = \mathcal{P}(s' | s, a) \pi(a' | s')$  and  $f(x) = Q_1(s', a') - Q_2(s', a')$ . We have:

$$|T^\pi Q_1(s, a) - T^\pi Q_2(s, a)| \leq \gamma \cdot \|Q_1 - Q_2\|_\infty \quad (34)$$

for all  $(s, a) \in \mathcal{S} \times \mathcal{A}$ . Now, we can compute the supremum norm of the difference between  $T^\pi Q_1$  and  $T^\pi Q_2$ :

$$\begin{aligned} \|T^\pi(Q_1) - T^\pi(Q_2)\|_\infty &= \sup_{(s,a) \in \mathcal{S} \times \mathcal{A}} |T^\pi Q_1(s, a) - T^\pi Q_2(s, a)| \\ &\leq \sup_{(s,a) \in \mathcal{S} \times \mathcal{A}} (\gamma \cdot \|Q_1 - Q_2\|_\infty) \\ &= \gamma \|Q_1 - Q_2\|_\infty \end{aligned} \quad (35)$$

This shows that the Bellman operator is a  $\gamma$ -contraction mapping with the supremum norm on  $Q$ , as desired.



### 2.1.2

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

\$\$

$$\left| \left( T^\pi Q_1 \right)(s, a) - \left( T^\pi Q_2 \right)(s, a) \right| \leq \gamma \|Q_1 - Q_2\|_\infty$$

\$\$

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

\$\$

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

\$\$

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}`

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

`\end{aligned}`

\$\$

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



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



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

Recall that the Bellman operator for  $Q$  is defined as:

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

Let's consider the difference between the Bellman operators applied to  $Q_1$  and  $Q_2$  for any  $(s, a) \in \mathcal{S} \times \mathcal{A}$ :

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

Now, we can apply the hint. Let  $P(x) = \mathcal{P}(x | s, a)$  and  $f(x) = Q_1(x) - Q_2(x)$ . Then,

```
\begin{aligned}\begin{aligned}&\text{\begin{aligned}} \\ &\quad \left| T^\pi Q_1(s, a) - T^\pi Q_2(s, a) \right| \leq \gamma \int |P(x)| |f(x)| \\ &\quad & \leq \gamma \int P(x) \sup_{(s, a) \in \mathcal{S} \times \mathcal{A}} |f(x)| \\ &\quad & = \gamma \sup_{(s, a) \in \mathcal{S} \times \mathcal{A}} |f(x)|\end{aligned}\right. \\ &\text{\end{aligned}}
```

**DO** continue



Continuing from where we left off, we have:



$$|T^\pi Q_1(s, a) - T^\pi Q_2(s, a)| \leq \gamma \cdot \|Q_1 - Q_2\|_\infty$$

for all  $(s, a) \in \mathcal{S} \times \mathcal{A}$ . Now, we can compute the supremum norm of the difference between  $T^\pi Q_1$  and  $T^\pi Q_2$ :

$$\begin{aligned}\|T^\pi(Q_1) - T^\pi(Q_2)\|_\infty &= \sup_{(s, a) \in \mathcal{S} \times \mathcal{A}} |T^\pi Q_1(s, a) - T^\pi Q_2(s, a)| \\ &\leq \sup_{(s, a) \in \mathcal{S} \times \mathcal{A}} (\gamma \cdot \|Q_1 - Q_2\|_\infty) \\ &= \gamma \|Q_1 - Q_2\|_\infty\end{aligned}$$

This shows that the Bellman operator is a  $\gamma$ -contraction mapping with the supremum norm on  $\mathcal{C}(\mathcal{S})$ , as desired.

### 2.1.3 Reward Function

**Solution:**

1. Definition of  $v_*(s)$  in terms of  $q_*(s, a)$ :

$v_*(s)$  is the optimal value function, which represents the maximum expected discounted return that can be achieved from state  $s$ . We can define  $v_*(s)$  in terms of the optimal action-value function  $q_*(s, a)$  as follows:

$$v_*(s) = \max_{a \in \mathcal{A}} q_*(s, a) \tag{36}$$

2. Definition of  $q_*(s, a)$  in terms of  $v_*(s)$ :

$q_*(s, a)$  is the optimal action-value function, which represents the maximum expected discounted return that can be achieved by taking action  $a$  in state  $s$  and following an optimal policy thereafter. We can define  $q_*(s, a)$  in terms of the optimal value function  $v_*(s)$  as follows:

$$q_*(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s' | s, a) v_*(s') \tag{37}$$

3. Definition of  $a_*$  in terms of  $q_*(s, a)$ :

$a_*$  is the optimal action to take in state  $s$  in order to achieve the maximum expected discounted return. We can define  $a_*$  in terms of the optimal action-value function  $q_*(s, a)$  as follows:

$$a_* = \arg \max_{a \in \mathcal{A}} q_*(s, a) \quad (38)$$

4. Definition of  $a_*$  in terms of  $v_*(s)$ :

Using the relationship between  $v_*(s)$  and  $q_*(s, a)$ , we can define the optimal action  $a_*$  in terms of  $v_*(s)$  as follows:

$$a_* = \arg \max_{a \in \mathcal{A}} \left[ r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s' | s, a) v_*(s') \right] \quad (39)$$

**DO**  
2.1.3

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

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



1. Definition of  $v_*(s)$  in terms of  $q_*(s, a)$ :

$v_*(s)$  is the optimal value function, which represents the maximum expected discounted return that can be achieved from state  $s$ . We can define  $v_*(s)$  in terms of the optimal action-value function  $q_*(s, a)$  as follows:

$$v_*(s) = \max_{a \in \mathcal{A}} q_*(s, a)$$

1. Definition of  $q_*(s, a)$  in terms of  $v_*(s)$ :

$q_*(s, a)$  is the optimal action-value function, which represents the maximum expected discounted return that can be achieved by taking action  $a$  in state  $s$  and following an optimal policy thereafter. We can define  $q_*(s, a)$  in terms of the optimal value function  $v_*(s)$  as follows:

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

1. Definition of  $a_*$  in terms of  $q_*(s, a)$ :

$a_*$  is the optimal action to take in state  $s$  in order to achieve the maximum expected discounted return. We can define  $a_*$  in terms of the optimal action-value function  $q_*(s, a)$  as follows:

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

1. Definition of  $a_*$  in terms of  $v_*(s)$ :

Using the relationship between  $v_*(s)$  and  $q_*(s, a)$ , we can define the optimal action  $a_*$  in terms of  $v_*(s)$  as follows:

$$a_* = \arg \max_{a \in \mathcal{A}} \left[ r(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s' | s, a) v_*(s') \right]$$

### 2.2.1 Closed form expression for REINFORCE estimator

**Solution:**

First, let's find the expression for the log likelihood of the Bernoulli distribution, which is given by:

$$\log p(a = \tilde{a} | \theta) = \tilde{a} \log \mu + (1 - \tilde{a}) \log(1 - \mu) \quad (40)$$

Now, we can compute the gradient of this expression with respect to  $\theta$ :

$$\frac{\partial}{\partial \theta} \log p(a = \tilde{a} | \theta) = \frac{\partial}{\partial \theta} (\tilde{a} \log \mu + (1 - \tilde{a}) \log(1 - \mu)) \quad (41)$$

We will need the gradient of  $\mu$  with respect to  $\theta_d$ :

$$\frac{\partial \mu}{\partial \theta_d} = \frac{\partial}{\partial \theta_d} \sigma \left( \sum_{d=1}^D \theta_d x_d \right) = \sigma \left( \sum_{d=1}^D \theta_d x_d \right) \left( 1 - \sigma \left( \sum_{d=1}^D \theta_d x_d \right) \right) x_d = \mu(1 - \mu)x_d \quad (42)$$

Now we can compute the gradient of the log likelihood with respect to  $\theta_d$ :

$$\begin{aligned} \frac{\partial}{\partial \theta_d} \log p(a = \tilde{a} | \theta) &= \tilde{a} \frac{\partial \log \mu}{\partial \theta_d} + (1 - \tilde{a}) \frac{\partial \log(1 - \mu)}{\partial \theta_d} \\ &= \tilde{a} \frac{1}{\mu} \frac{\partial \mu}{\partial \theta_d} - (1 - \tilde{a}) \frac{1}{1 - \mu} \frac{\partial \mu}{\partial \theta_d} \\ &= \tilde{a} \frac{1}{\mu} \mu(1 - \mu)x_d - (1 - \tilde{a}) \frac{1}{1 - \mu} \mu(1 - \mu)x_d \\ &= (\tilde{a} - \mu)x_d \end{aligned} \quad (43)$$

Now, we can substitute the gradient of the log likelihood into the expression for  $g[\theta, \tilde{a}]$ :

$$g[\theta, \tilde{a}] = f(\tilde{a})(\tilde{a} - \mu)x_d \quad (44)$$

This is the closed form for the  $g[\theta, \tilde{a}]$  as a deterministic function of  $\tilde{a}$ ,  $\mu$ ,  $\theta$ , and  $x_d$ .

DO

#### 2.2 Policy gradients and black box optimization

Very often we have a function  $f$  that does not give us useful gradient information: input or output may be discrete;  $f$  may be piecewise constant, nowhere differentiable, or have pathological gradients (e.g., a discontinuous saw wave on an incline, whose gradient always points away from the global optimum); or  $f$  may be a black box that we cannot backpropagate through. For example, we may have a phone app that labels photos as cats or dogs. This situation is the default in Reinforcement Learning (RL), where we can execute the environment dynamics, but we cannot see or control their internals.

We still, however, want to optimize some score function  $J[f]: X \rightarrow \mathbb{R}$ . For example, in RL, we want to learn a policy that maximizes the non-differentiable environment reward.

When using the REINFORCE strategy, we replaced the  $\theta$  optimization task with a Monte-Carlo approximation. One of the key factors for a successful REINFORCE application is the variance. The higher the variance, the more "noisy" the gradient estimates will be, which can slow down the optimization process. In this section we will derive the variance of the REINFORCE estimator for a simple toy task.

Consider a loss function,  $f(\tilde{a})$  which is the zero-one loss of the logistic regression output,  $p(a | \theta)$ . The input vector has  $D$  independent scalar features,  $x_d$ . We evaluate the performance of the classifier by sampling from the output of the sigmoid  $\mu$ . The loss function  $J(\theta)$  can be written as:

\$\$
\begin{aligned}
&\text{Consider a loss function, } f(\tilde{a}) \\
&\text{which is the zero-one loss of the logistic regression output, } p(a | \theta). \\
&\text{The input vector has } D \text{ independent scalar features, } x\_d. \\
&\text{We evaluate the performance of the classifier by sampling from the output of the sigmoid } \mu. \\
&\text{The loss function } J(\theta) \text{ can be written as:} \\
&f(\tilde{a}) = \sum\_{d=1}^D \theta\_d x\_d
\end{aligned}

```

\end{array},\right. \\
\tilde{a} \& \sim p(a \mid \theta), \\
f(\tilde{a}) & =\left.\begin{array}{l}
1 & \tilde{a}=1 \\
0 & \tilde{a}=0
\end{array}\right. \\
J(\theta) & =\mathbb{E}_{\tilde{a}}[\tilde{a} \sim p(a \mid \theta)] f(\tilde{a})] . \\
\end{aligned}

```

### 2.2.1 Closed form expression for REINFORCE estimator [0.5pt]

Recall from above that the expression for REINFORCE estimator is:

\$\$

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tilde{a} \sim p(a | \theta)} [\nabla_{\theta} \log p(a | \theta)]$$

\$\$

We can denote the expression inside the expectation as  $\$g[\theta, \mathbf{x}]$  :  
\$\$

g

$\tilde{a} \sim p(a | \theta)$

Fo

function of  $\tilde{a}$ ,  $\mu$ ,  $\theta$ , and  $x_d$ .  
Hint: Substitute in the log likelihood of the Bernoulli distribution.

Hint: Substitute in the log likelihood of the Bernoulli distribution.



First, let's find the expression for the log likelihood of the Bernoulli distribution, which is given by:

$$\log p(a = \bar{a} \mid \theta) = \bar{a} \log \mu + (1 - \bar{a}) \log(1 - \mu)$$

Now, we can compute the gradient of this expression with respect to  $\theta$ :

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

We will need the gradient of  $\mu$  with respect to  $\theta_d$ :

$$\frac{\partial \mu}{\partial \theta_d} = \frac{\partial}{\partial \theta_d} \sigma \left( \sum_{d=1}^D \theta_d x_d \right) = \sigma \left( \sum_{d=1}^D \theta_d x_d \right) \left( 1 - \sigma \left( \sum_{d=1}^D \theta_d x_d \right) \right) x_d = \mu(1-\mu)x_d$$

Now we can compute the gradient of the log likelihood with respect to  $\theta_d$ :

$$\begin{aligned}
\frac{\partial}{\partial \theta_d} \log p(a = \tilde{a} \mid \theta) &= \tilde{a} \frac{\partial \log \mu}{\partial \theta_d} + (1 - \tilde{a}) \frac{\partial \log(1 - \mu)}{\partial \theta_d} \\
&= \tilde{a} \frac{1}{\mu} \frac{\partial \mu}{\partial \theta_d} - (1 - \tilde{a}) \frac{1}{1 - \mu} \frac{\partial \mu}{\partial \theta_d} \\
&= \tilde{a} \frac{1}{\mu} \mu(1 - \mu)x_d - (1 - \tilde{a}) \frac{1}{1 - \mu} \mu(1 - \mu)x_d \\
&= (\tilde{a} - \mu)x_d
\end{aligned}$$

Now, we can substitute the gradient of the log likelihood into the expression for  $\hat{g}(\theta, \tilde{a})$ :

$$g[\theta, \tilde{a}] = f(\tilde{a})(\tilde{a} - \mu)x_d$$

This is the closed form for the  $g[\theta, \tilde{a}]$  as a deterministic function of  $\tilde{a}, \mu, \theta$ , and  $x_d$ .

# 3 Graph Convolution Networks

## 3.1 Implementation of Graph Convolution Layer

Solution:

```
1 ~ class GraphConvolution(nn.Module):
2 ~     """
3 ~     A Graph Convolution Layer (GCN)
4 ~     """
5
6 ~     def __init__(self, in_features, out_features, bias=True):
7 ~         """
8 ~             * `in_features`,  $F$ , is the number of input features per node
9 ~             * `out_features`,  $F'$ , is the number of output features per node
10 ~            * `bias`, whether to include the bias term in the linear layer. Default=True
11 ~        """
12 ~        super(GraphConvolution, self).__init__()
13 ~        # TODO: initialize the weight  $W$  that maps the input feature (dim  $F$ ) to output feature (dim  $F'$ )
14 ~        # hint: use nn.Linear()
15 ~        ##### Your code here #####
16 ~        self.linear = nn.Linear(in_features, out_features, bias=bias)
17 ~        #####
18
19 ~    def forward(self, input, adj):
20 ~        """
21 ~            # TODO: transform input feature to output (don't forget to use the adjacency matrix
22 ~            # to sum over neighbouring nodes)
23 ~            # hint: use the linear layer you declared above.
24 ~            # hint: you can use torch.spmm() - sparse-matrix multiplication to handle the
25 ~            # adjacency matrix
26 ~            ##### Your code here #####
27 ~        s = self.linear(input)
28 ~        h_prime = torch.spmm(adj, s)
29
30 ~    return h_prime
31 ~    #####
```



### Basics of GCN:

Recall from the lecture, the goal of a GCN is to learn a function of signals/features on a graph  $G=(V, E)$ , which takes as inputs:

1. the input features of each node,  $x_i \in \mathcal{R}^F$  (in matrix form:  $X \in \mathcal{R}^{|\mathcal{V}| \times F}$ )
2. some information about the graph structure, typically the adjacency matrix  $A$ .  
Each convolutional layer can be written as  $H^{(l+1)} = f(\left(H^{(l)}, A\right))$ , for some function  $f()$ . The  $f()$  we are using for this assignment is in the form of  $f(\left(H^{(l)}, A\right)) = \sigma(\left(A + D^{-1/2} \hat{A} D^{-1/2} H^{(l)}\right) W^{(l)})$ , where  $\hat{A} = A + I$  Identity and  $D$  is diagonal node degree matrix  $\left(D^{-1} \hat{A}\right)$  normalizes  $\hat{A}$  such that all rows sum to one). Let  $\tilde{A} = D^{-1/2} \hat{A} D^{-1/2}$ . The GCN we will implement takes two convolution layers,  $Z = f(X, A) = \text{softmax}(\left(\tilde{A} \cdot X W^{(0)}\right) \cdot W^{(1)})$ .

## Define Graph Convolution layer (Your Task)

This module takes  $\mathbf{h} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, \dots, \overrightarrow{h_N}\}$  where  $\overrightarrow{h_i} \in \mathbb{R}^F$  as input and outputs  $\mathbf{h}' = \{\overrightarrow{h'_1}, \overrightarrow{h'_2}, \dots, \overrightarrow{h'_N}\}$ , where  $\overrightarrow{h'_i} \in \mathbb{R}^{F'}$ .

1. perform initial transformation:  $\mathbf{s} = \mathbf{W} \times \mathbf{h}^{(l)}$
2. multiply  $\mathbf{s}$  by normalized adjacency matrix:  $\mathbf{h}' = \mathbf{A} \times \mathbf{s}$

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

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

```



Here's the implementation of the Graph Convolution Layer (GCN) class:



```
python
import torch
import torch.nn as nn

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__()
        self.linear = nn.Linear(in_features, out_features, bias=bias)
        self.adj_norm = nn.Parameter(torch.FloatTensor(out_features, out_features))
        self.bn = nn.BatchNorm1d(out_features)
        self.act = nn.ReLU()

        ##### Your code here #####
        #####
    
```

```
* `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 is False
"""

super(GraphConvolution, self).__init__()

# TODO: initialize the weight W that maps the input feature (dim F) to output (dim F')
# hint: use nn.Linear()
#####
##### Your code here #####
self.linear = 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
    #       adjacency matrix
    #####
    ##### Your code here #####
    s = self.linear(input)
    h_prime = torch.spmm(adj, s)
    #####
    return h_prime
```

### 3.2 Implementation of Graph Convolution Network

Solution:

```

3 class GCN(nn.Module):
4     """
5     A two-layer GCN
6     """
7     def __init__(self, nfeat, n_hidden, n_classes, dropout, bias=True):
8         """
9             * `nfeat`, is the number of input features per node of the first layer
10            * `n_hidden`, number of hidden units
11            * `n_classes`, total number of classes for classification
12            * `dropout`, the dropout ratio
13            * `bias`, whether to include the bias term in the linear layer. Default=True
14        """
15
16        super(GCN, self).__init__()
17        # TODO: Initialization
18        # (1) 2 GraphConvolution() layers.
19        # (2) 1 Dropout layer
20        # (3) 1 activation function: ReLU()
21        ##### Your code here #####
22        self.gc1 = GraphConvolution(nfeat, n_hidden, bias=bias)
23        self.gc2 = GraphConvolution(n_hidden, n_classes, bias=bias)
24        self.dropout = nn.Dropout(dropout)
25        self.activation = nn.ReLU()
26        #####
27
28    def forward(self, x, adj):
29        # TODO: the input will pass through the first graph convolution layer,
30        # the activation function, the dropout layer, then the second graph
31        # convolution layer. No activation function for the
32        # last layer. Return the logits.
33        ##### Your code here #####
34        x = self.gc1(x, adj)
35        x = self.activation(x)
36        x = self.dropout(x)
37        x = self.gc2(x, adj)
38        return x
39        #####

```

**DO** ## Define GCN (Your Task)

you will implement a two-layer GCN with ReLU activation function and Dropout after the first Conv layer.

```

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

```



Here's the implementation of the two-layer GCN with ReLU activation and dropout:

```
python Copy code
import torch.nn.functional as F

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 is True
        """

        super(GCN, self).__init__()
        # TODO: Initialization
        # (1) 2 GraphConvolution() layers.
        # (2) 1 Dropout layer
        # (3) 1 activation function: ReLU()
        ###### Your code here #####
        self.gc1 = GraphConvolution(nfeat, n_hidden, bias=bias)
        self.gc2 = GraphConvolution(n_hidden, n_classes, bias=bias)
        self.dropout = nn.Dropout(dropout)
        self.activation = 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.gc1(x, adj)
    x = self.activation(x)
    x = self.dropout(x)
    x = self.gc2(x, adj)
    #####
    return x

```

In this implementation, we first initialize two GraphConvolution layers, a Dropout layer, and a ReLU activation function. In the `forward` function, the input features pass through the first graph convolution layer, followed by the ReLU activation, the dropout layer, and finally the second graph convolution layer. The logits are returned as output.

### 3.3 Train your Graph Convolution Network

**Solution:** last 10 epochs and test results

```

91 Epoch: 0091 loss_train: 0.7919 acc_train: 0.8286 loss_val: 1.1238 acc_val: 0.6254 time: 0.0019s
92 Epoch: 0092 loss_train: 0.7797 acc_train: 0.8429 loss_val: 1.1160 acc_val: 0.6293 time: 0.0019s
93 Epoch: 0093 loss_train: 0.7539 acc_train: 0.8786 loss_val: 1.1086 acc_val: 0.6336 time: 0.0019s
94 Epoch: 0094 loss_train: 0.8075 acc_train: 0.8286 loss_val: 1.1020 acc_val: 0.6386 time: 0.0019s
95 Epoch: 0095 loss_train: 0.7681 acc_train: 0.8500 loss_val: 1.0956 acc_val: 0.6421 time: 0.0019s
96 Epoch: 0096 loss_train: 0.7314 acc_train: 0.8286 loss_val: 1.0892 acc_val: 0.6452 time: 0.0019s
97 Epoch: 0097 loss_train: 0.7305 acc_train: 0.8786 loss_val: 1.0826 acc_val: 0.6491 time: 0.0019s
98 Epoch: 0098 loss_train: 0.7108 acc_train: 0.8786 loss_val: 1.0760 acc_val: 0.6515 time: 0.0019s
99 Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703 acc_val: 0.6488 time: 0.0019s
100 Epoch: 0100 loss_train: 0.7307 acc_train: 0.8286 loss_val: 1.0643 acc_val: 0.6515 time: 0.0019s
101 Optimization Finished!
102 Total time elapsed: 0.8315s
103 Test set results: loss= 1.0643 accuracy= 0.6515

```

### 3.4 Implementation of Graph Attention Layer

**Solution:**

```

1 class GraphAttentionLayer(nn.Module):
2
3     def __init__(self, in_features: int, out_features: int, n_heads: int,
4                  is_concat: bool = True,
5                  dropout: float = 0.6,
6                  alpha: float = 0.2):
7         .....
8         in_features: F, the number of input features per node
9         out_features: F', the number of output features per node
10        n_heads: K, the number of attention heads
11        is concat: whether the multi-head results should be concatenated or averaged

```

```

11     is_concat: whether the multi-head results should be concatenated or averaged
12     dropout: the dropout probability
13     alpha: the negative slope for leaky relu activation
14     """
15     super(GraphAttentionLayer, self).__init__()
16
17     self.is_concat = is_concat
18     self.n_heads = n_heads
19
20     if is_concat:
21         assert out_features % n_heads == 0
22         self.n_hidden = out_features // n_heads
23     else:
24         self.n_hidden = out_features
25
26     # TODO: initialize the following modules:
27     # (1) self.W: Linear layer that transform the input feature before self.attention.
28     # You should NOT use for-loops for the multiheaded implementation (set bias=False)
29     # (2) self.attention: Linear layer that compute the attention score (set bias=False)
30     # (3) self.activation: Activation function (LeakyReLU with negative_slope=alpha)
31     # (4) self.softmax: Softmax function (what's the dim to compute the summation?)
32     # self.dropout_layer: Dropout function (with ratio=dropout)
33     ##### your code here #####
34     self.W = nn.Linear(in_features, out_features, bias=False)
35     self.attention = nn.Linear(2 * self.n_hidden, 1, bias=False)
36     self.activation = nn.LeakyReLU(negative_slope=alpha)
37     self.softmax = nn.Softmax(dim=1)
38     self.dropout_layer = nn.Dropout(dropout)
39     #####
40
41     def forward(self, h: torch.Tensor, adj_mat: torch.Tensor):
42         # Number of nodes
43         n_nodes = h.shape[0]
44
45         # TODO:
46         # (1) calculate s = Wh and reshape it to [n_nodes, n_heads, n_hidden].
47         # (you can use tensor.view() function)
48         # (2) get [s_i || s_j] using tensor.repeat(), repeat_interleave(), torch.cat(), tensor.view()
49         # (3) apply the attention layer.
50         # (4) apply the activation layer (you will get the attention score e)
51         # (5) remove the last dimension 1 use tensor.squeeze()
52         # mask the attention score with the adjacency matrix (if there's no edge, assign it to -inf)
53         # note: check the dimensions of e and your adjacency matrix. You may need to use the function unsqueeze()
54         # (7) apply softmax
55         # (8) apply dropout_layer
56         ##### Your code here #####
57         s = self.W(h).view(n_nodes, self.n_heads, self.n_hidden)
58         s_i = s.repeat(1, n_nodes, 1).view(n_nodes, n_nodes, self.n_heads, self.n_hidden)
59         s_j = s.repeat_interleave(n_nodes, dim=0).view(n_nodes, n_nodes, self.n_heads, self.n_hidden)
60         s_cat = torch.cat([s_i, s_j], dim=-1)
61         e = self.attention(s_cat)
62         e = self.activation(e).squeeze(-1)
63         adj_mask = adj_mat.unsqueeze(-1).expand_as(e)
64         e = e.masked_fill(adj_mask == 0, float('-inf'))
65         a = self.softmax(e)
66         a = self.dropout_layer(a)
67         #####
68
69         # Summation
70         h_prime = torch.einsum('ijh,jhf->ihf', a, s) #[n_nodes, n_heads, n_hidden]
71
72
73         # TODO: Concat or Mean
74         # Concatenate the heads
75         if self.is_concat:
76             ##### Your code here #####
77             h_prime = h_prime.reshape(n_nodes, -1)
78             #####
79         # Take the mean of the heads (for the last layer)
80         else:
81             ##### Your code here #####
82             h_prime = h_prime.mean(dim=1)
83             #####

```

**DO** ## Graph attention layer (Your task)

A GAT is made up of multiple such layers. In this section, you will implement a single graph attention layer. Similar to the `GraphConvolution()`, this `GraphAttentionLayer()` module takes  $\mathbf{h} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, \dots, \overrightarrow{h_N}\}$  where  $\overrightarrow{h_i} \in \mathbb{R}^F$  as input and outputs  $\mathbf{h}' = \{\overrightarrow{h'_1}, \overrightarrow{h'_2}, \dots, \overrightarrow{h'_N}\}$ , where  $\overrightarrow{h'_i} \in \mathbb{R}^{F'}$ . However, instead of weighing each neighbouring node based on the adjacency matrix, we will use self attention to learn the relative importance of each neighbouring node. Recall from HW4 where you are asked to write out the equation for single headed attention, here we will implement multi-headed attention, which involves the following steps:

#### ### The initial transformation

In GCN above, you have completed similar transformation. But here, we need to define a weight matrix and perform this transformation for each head:  $\overrightarrow{s^k_i} = \mathbf{W}^k \overrightarrow{h_i}$ . We will perform a single linear transformation and then split it up for each head later. Note the input  $\overrightarrow{h}$  has shape `[n\_nodes, in\_features]` and  $\overrightarrow{s}$  has shape of `[n\_nodes, n\_heads \* n\_hidden]`. Remember to reshape  $\overrightarrow{s}$  has shape of `[n\_nodes, n\_heads, n\_hidden]` for later uses. Note: set `bias=False` for this linear transformation.

#### ### attention score

We calculate these for each head  $k$ . Here for simplicity of the notation, we omit  $k$  in the following equations. The attention scores are defined as the follows:

$$\begin{aligned} \mathbf{e}_{ij} &= a(\mathbf{W} \overrightarrow{h_i}, \mathbf{W} \overrightarrow{h_j}) \\ &= a(\overrightarrow{s_i}, \overrightarrow{s_j}) \end{aligned}$$

where  $e_{ij}$  is the attention score (importance) of node  $j$  to node  $i$ .

We will have to calculate this for each head.  $a$  is the attention mechanism, that calculates the attention score. The paper concatenates  $\overrightarrow{s_i}$ ,  $\overrightarrow{s_j}$  and does a linear transformation with a weight vector  $\mathbf{a} \in \mathbb{R}^{2F}$  followed by a  $\text{LeakyReLU}$ .  $e_{ij} = \text{LeakyReLU}(\mathbf{a}^\top \overrightarrow{s_i} \mathbf{a}^\top \overrightarrow{s_j})$

#### #### How to vectorize this? Some hints:

1. `tensor.repeat()` gives you  $\{\overrightarrow{s_1}, \overrightarrow{s_2}, \dots, \overrightarrow{s_N}, \overrightarrow{s_1}, \overrightarrow{s_2}, \dots, \overrightarrow{s_N}\}$ .

2. `tensor.repeat\_interleave()` gives you  $\{\overrightarrow{s_1}, \overrightarrow{s_1}, \dots, \overrightarrow{s_1}, \overrightarrow{s_2}, \overrightarrow{s_2}, \dots, \overrightarrow{s_2}\}$ .

3. concatenate to get  $\Big[\overrightarrow{s_i} \Big| \overrightarrow{s_j} \Big]$  for all pairs of  $i, j$ . Reshape  $\overrightarrow{s_i} \Big| \overrightarrow{s_j}$  has shape of `[n\_nodes, n\_nodes, n\_heads, 2 \* n\_hidden]`

4. apply the attention layer and non-linear activation function to get  $e_{ij} = \text{LeakyReLU}(\mathbf{a}^\top \overrightarrow{s_i} \mathbf{a}^\top \overrightarrow{s_j})$ , where  $\mathbf{a}^\top$  is a single linear transformation that maps from dimension  $n\_hidden * 2$  to  $1$ . Note: set the `bias=False` for this linear transformation.  $\mathbf{e}$  is of shape `[n\_nodes, n\_nodes, n\_heads, 1]`. Remove the last dimension `1` using `squeeze()`.

#### #### Perform softmax

First, we need to mask  $e_{ij}$  based on adjacency matrix. We only need to sum over the neighbouring nodes for the attention calculation. Set the elements in the  $\mathbf{e}$  that are not in the  $\mathbf{A}$  to  $-\infty$ .

neighbouring nodes for the attention calculation. Set the elements in  $e_{ij}$  to  $-\infty$  if there is no edge from  $i$  to  $j$  for the softmax calculation. We need to do this for all heads and the adjacency matrix is the same for each head. Use `tensor.masked\_fill()` to mask  $e_{ij}$  based on adjacency matrix for all heads. Hint: reshape the adjacency matrix to  $[n\_nodes, n\_nodes, 1]$  using `unsqueeze()`.

Now we are ready to normalize attention scores (or coefficients)  $\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$

#### Apply dropout

Apply the dropout layer. (this step is easy)

#### Calculate final output for each head

$\overrightarrow{h'k_i} = \sum_j (\mathcal{N}_i) \alpha_{kj} \overrightarrow{s^k}$

#### Concat or Mean

Finally we concateneate the transformed features:  $\overrightarrow{h'_i} = \overrightarrow{\bigg( \sum_k \overrightarrow{h'k_i} \bigg)}$ . In the code, we only need to reshape the tensor to shape of  $[n\_nodes, n\_heads * n\_hidden]$ . Note that if it is the final layer, then it doesn't make sense to do concatenation anymore. Instead, we sum over the  $n\_heads$  dimension:  $\overrightarrow{h'_i} = \frac{1}{K} \sum_{k=1}^K \overrightarrow{h'k_i}$ .

```
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 = False)
    # (2) self.attention: Linear layer that compute the attention score (set bias = False)
    # (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 #####

```

```

#####
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 #####

```

```

#####
# 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
#####
# Take the mean of the heads (for the last layer)
else:
    ##### Your code here
#####

```



Here's the implementation of a single graph attention layer:



python

Copy code

```
import torch
import torch.nn as nn
import torch.nn.functional as F

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
        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 s
        # You should NOT use for loops for the multiheaded implementation (s
        # (2) self.attention: Linear layer that compute the attention score
        # (3) self.activation: Activation function (LeakyReLU with negative_
        # (4) self.softmax: Softmax function (what's the dim to compute the
        # (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(2 * self.n_hidden, 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(), t
        # (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
        #     note: check the dimensions of e and your adjacency matrix. You
        # (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(1, n_nodes, 1).view(n_nodes, n_nodes, self.n_heads, s
s_j = s.repeat_interleave(n_nodes, dim=0).view(n_nodes, n_nodes, sel
s_cat = torch.cat([s_i, s_j], dim=-1)
e = self.attention(s_cat)
```

 continue



Here's the continuation of the `GraphAttentionLayer` implementation:



```
python  Copy code

    e = self.activation(e).squeeze(-1)
    adj_mask = adj_mat.unsqueeze(-1).expand_as(e)
    e = e.masked_fill(adj_mask == 0, float('-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_h

    # TODO: Concat or Mean
    # Concatenate the heads
    if self.is_concat:
        ##### Your code here #####
        h_prime = h_prime.reshape(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
```

With this code, you have completed the implementation of the Graph Attention Layer. It calculates the initial transformation, attention scores, softmax, applies dropout, and calculates the final output for each head, then concatenates or averages the multi-head results as necessary.

### 3.5 Train your Graph Attention Network

**Solution:** last 10 epochs with test set results:

```
91 Epoch: 0091 loss_train: 1.0427 acc_train: 0.6857 loss_val: 1.0975 acc_val: 0.7426 time: 0.0499s
92 Epoch: 0092 loss_train: 0.9856 acc_train: 0.8214 loss_val: 1.0913 acc_val: 0.7457 time: 0.0499s
93 Epoch: 0093 loss_train: 1.0248 acc_train: 0.7429 loss_val: 1.0853 acc_val: 0.7484 time: 0.0499s
94 Epoch: 0094 loss_train: 0.9592 acc_train: 0.7929 loss_val: 1.0794 acc_val: 0.7496 time: 0.0499s
95 Epoch: 0095 loss_train: 0.9814 acc_train: 0.8071 loss_val: 1.0734 acc_val: 0.7512 time: 0.0498s
96 Epoch: 0096 loss_train: 0.9951 acc_train: 0.7714 loss_val: 1.0677 acc_val: 0.7523 time: 0.0499s
97 Epoch: 0097 loss_train: 0.9369 acc_train: 0.7786 loss_val: 1.0622 acc_val: 0.7535 time: 0.0499s
98 Epoch: 0098 loss_train: 0.9871 acc_train: 0.7429 loss_val: 1.0568 acc_val: 0.7547 time: 0.0499s
99 Epoch: 0099 loss_train: 0.8497 acc_train: 0.8571 loss_val: 1.0514 acc_val: 0.7547 time: 0.0499s
100 Epoch: 0100 loss_train: 1.0142 acc_train: 0.7500 loss_val: 1.0460 acc_val: 0.7570 time: 0.0499s
101 Optimization Finished!
102 Total time elapsed: 5.0088s
103 Test set results: loss= 1.0460 accuracy= 0.7570
```

### 3.6 Compare your models

**Solution:** The GAT model shows better performance than the Vanilla GCN, with higher validation and test accuracies. The reason for this discrepancy is that GAT uses self-attention mechanisms to learn the importance of each neighboring node, which allows it to capture more complex patterns and relations in the graph, thus resulting in better performance compared to the simpler Vanilla GCN.

## 4 Deep Q-Learning Network

### 4.1 Implementation of $\epsilon$ -greedy

**Solution:**

```
1 def get_action(model, state, action_space_len, epsilon):
2     # We do not require gradient at this point, because this function will be used either
3     # during experience collection or during inference
4
5     with torch.no_grad():
6         Qp = model.policy_net(torch.from_numpy(state).float())
7
8     ## TODO: select and return action based on epsilon-greedy
9     if torch.rand(1).item() > epsilon:
10         # Exploitation: choose the action with the highest Q-value
11         action = torch.argmax(Qp).view(1)
12     else:
13         # Exploration: choose a random action from the available actions
14         action = torch.randint(0, action_space_len, (1,))
15
16     return action
```

**DO**

```
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:
        # Exploitation: choose the action with the highest Q-value
        action = np.argmax(Qp.numpy())
    else:
        # Exploration: choose a random action from the available actions
        action = random.choice(range(action_space_len))

    return action
```

I'll call `action.item()` in the future, however, the action now is an int, and 'int' object has no attribute 'item'. Can you modify the codes above to return an action in pytorch? Hint: you may need to change the np and random into torch functions if possible



Sure, we can modify the ``get_action`` function to return a PyTorch tensor instead of an integer. You can replace the NumPy and random functions with their PyTorch equivalents as follows:

python

Copy code

```
import torch

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

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

    # Select and return action based on epsilon-greedy
    if torch.rand(1).item() > epsilon:
        # Exploitation: choose the action with the highest Q-value
        action = torch.argmax(Qp).view(1)
    else:
        # Exploration: choose a random action from the available actions
        action = torch.randint(0, action_space_len, (1,))

    return action
```

Now, the ``get_action`` function will return a PyTorch tensor representing the action. When calling this function, you can use ``action.item()`` to obtain the integer value of the action when needed.

## 4.2 Implementation of DQN training step

Solution:

the train function:

```
1 import torch.nn.functional as F
2
3 def train(model, batch_size):
4     state, action, reward, next_state = memory.sample_from_experience(sample_size=batch_size)
5
6     # TODO: predict expected return of current state using main network
7     state_action_values = model.policy_net(state).gather(1, action.long().unsqueeze(1))
8     # TODO: get target return using target network
9     next_state_values = model.target_net(next_state).max(1)[0].detach()
10    expected_state_action_values = (next_state_values * model.gamma) + reward
11
12    # TODO: compute the loss
13    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.unsqueeze(1))
14    model.optimizer.zero_grad()
15    loss.backward(retain_graph=True)
16    model.optimizer.step()
17
18    model.step += 1
19    if model.step % 5 == 0:
20        model.target_net.load_state_dict(model.policy_net.state_dict())
21
22    return loss.item()
```

the epsilon decay rule:

```
49     # TODO: add epsilon decay rule here!
50     epsilon *= 0.95
```

**DO** 2. [1pt] Implementation of DQN training step  
Complete the function train for the model to perform a single step of optimization. This is basically to construct the temporal difference error  $\delta$  and perform a standard optimizer update. Notice that there are two networks in the DQN\_network, policy\_net and target\_net, think about how to use these two networks to construct the loss. Include your code snippet in your write-up.

### \*\*[Your task]\*\*: complete the function that train the network for one step

Here, you can find an ``train`` function that performs a single step of the optimization.

For our training update rule, the loss you are trying to minimize is:

$$\text{loss} = Q(s, a) - (r + \gamma \max_a Q(s', a))$$

```
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
```

```

# TODO: get target return using target network

# TODO: compute the loss
loss =
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()

```



In the train function, we need to implement the following steps:

1. Predict the expected return of the current state using the policy\_net.
2. Get the target return using the target\_net.
3. Compute the loss based on the difference between the expected and target returns.
4. Perform a standard optimizer update.

Here's the modified function:

```

python                                         Copy code

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

    # Predict expected return of current state using main network (policy_net)
    state_action_values = model.policy_net(state).gather(1, action)

    # Get target return using target network (target_net)
    next_state_values = model.target_net(next_state).max(1)[0].detach()
    expected_state_action_values = (next_state_values * model.gamma) + reward

    # Compute the loss
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values)
    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()

```

In this implementation, we use the policy\_net to predict the expected return for the current state-action pairs and the target\_net to predict the target return for the next state. The loss is computed using the smooth L1 loss, which is less sensitive to outliers than the mean squared error. The optimizer then performs a gradient update, and the target network's parameters are updated every 5 steps.

**DO**    **### \*\*[Your task]\*\*:** Finish the training loop

In this part, you can play around with ```exp\_replay\_size```, ```episode```, ```epsilon``` and the "episodic decay" logic to train your model. **If you have done correctly, you will observe that the training time for the latter episodes is longer than the early episodes. This is because your agent is getting better and better at playing the game and thus each episode takes longer\*\***

```
# Create the model
env = gym.make('CartPole-v0')
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
agent = DQN_Network(layer_size_list=[input_dim, 64, output_dim], lr=1e-3)

# Main training loop
losses_list, reward_list, episode_len_list, epsilon_list = [], [], [], []

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

# initialize experience replay
index = 0
for i in range(exp_replay_size):
    obs = env.reset()
    done = False
    while not done:
        A = get_action(agent, obs, env.action_space.n, epsilon=1)
        obs_next, reward, done, _ = env.step(A.item())
        memory.collect([obs, A.item(), reward, obs_next])
        obs = obs_next
        index += 1
        if index > exp_replay_size:
            break

index = 128
for i in tqdm(range(episodes)):
    obs, done, losses, ep_len, rew = env.reset(), False, 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!
```

```

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")

```



Here's the complete training loop with an epsilon decay rule added. In this example, I set the experience replay size to 10000 and the number of episodes to 6000. Epsilon starts at 1 and decays by a factor of 0.995 after each episode. You can adjust these parameters and the decay rate to find the best settings for your training.

```

python                                         Copy code

# Create the model
env = gym.make('CartPole-v0')
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
agent = DQN_Network(layer_size_list=[input_dim, 64, output_dim], lr=1e-3)

# Main training loop
losses_list, reward_list, episode_len_list, epsilon_list = [], [], [], []

# Set experience replay size and number of episodes
exp_replay_size = 10000
memory = ExperienceReplay(exp_replay_size)
episodes = 6000
epsilon = 1 # epsilon starts at 1 and decays gradually

# Initialize experience replay
index = 0
for i in range(exp_replay_size):
    obs = env.reset()
    done = False
    while not done:
        A = get_action(agent, obs, env.action_space.n, epsilon=1)
        obs_next, reward, done, _ = env.step(A.item())
        memory.collect([obs, A.item(), reward, obs_next])
        obs = obs_next
        index += 1
        if index > exp_replay_size:
            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:
            break

```

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

    # Epsilon decay rule
    epsilon *= 0.995

    losses_list.append(losses / ep_len), reward_list.append(reward)
    episode_len_list.append(ep_len), epsilon_list.append(epsilon)

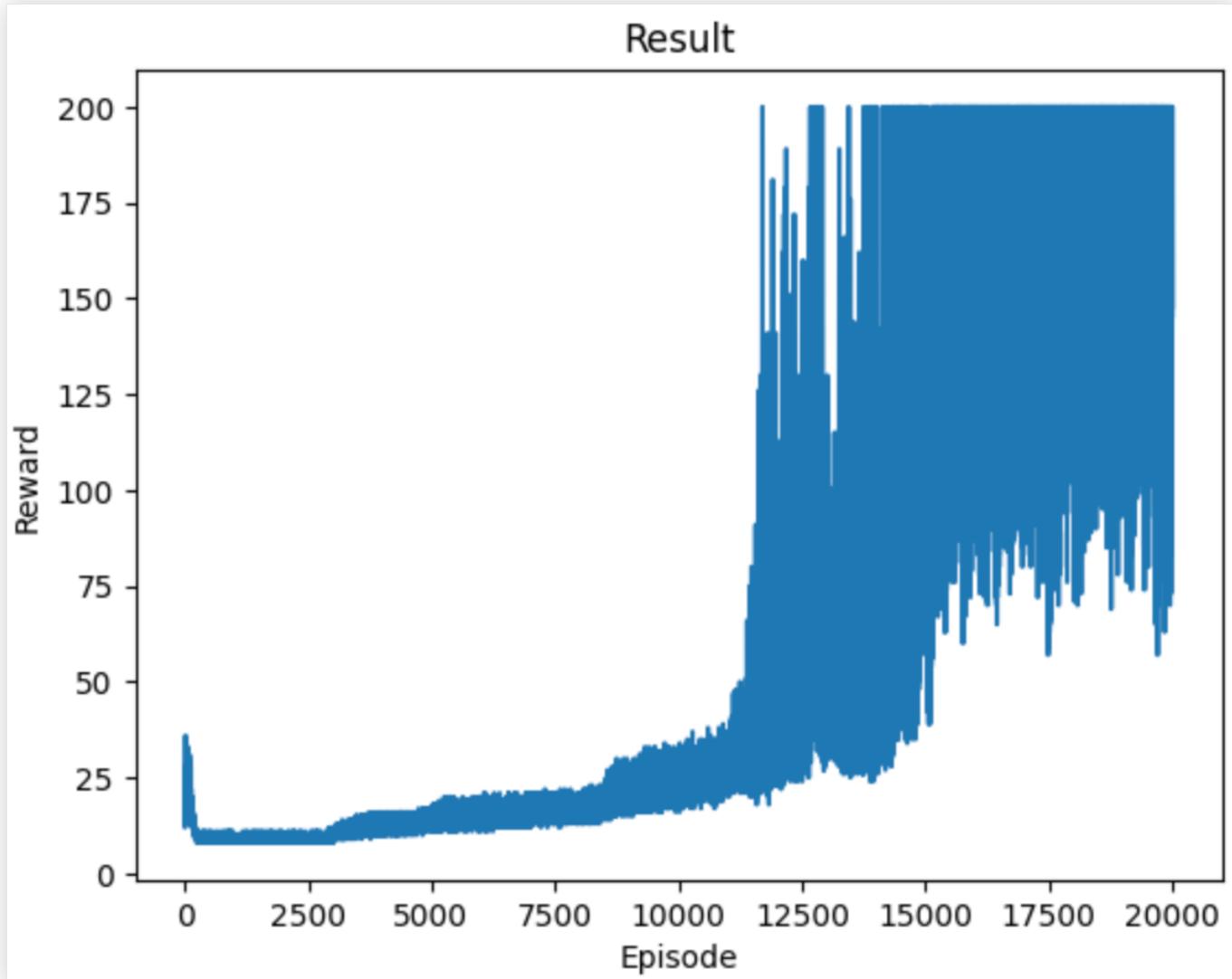
print("Saving trained model")
agent.save_trained_model("cartpole-dqn.pth")
```

The epsilon decay rule implemented here multiplies epsilon by a constant factor (0.995) after each episode, causing it to gradually decrease. This will encourage the agent to explore less and exploit more as it learns. You can experiment with different decay rates to find the best balance between exploration and exploitation during training.

### 4.3 Train your DQN Agent

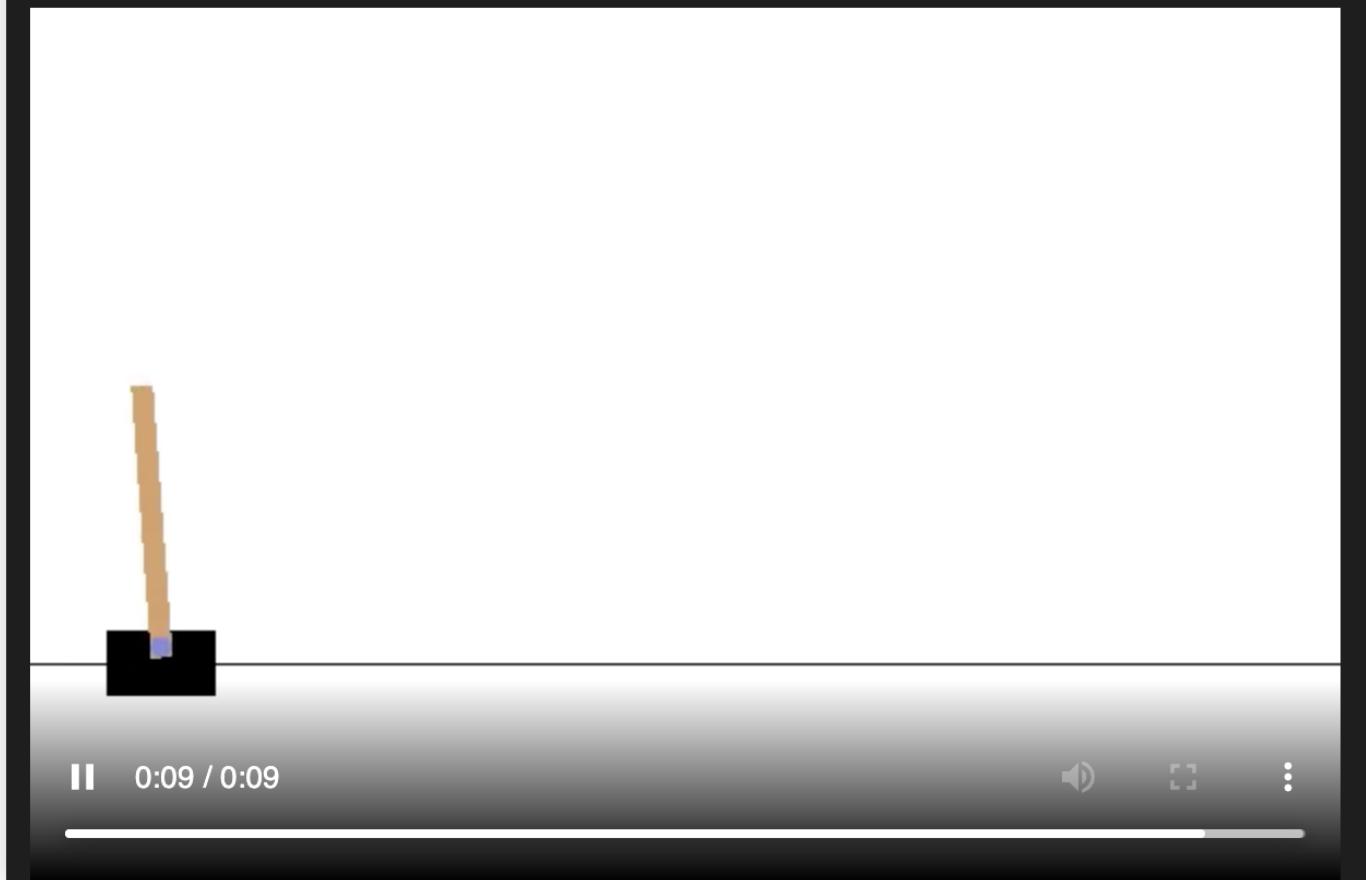
Solution:

- reward curve:



- end time frame of your cartpole video:

100% | 200/200 [00:20<00:00, 9.83it/s]  
average reward per episode : 195.25



- best combination of hyperparameters (obtained from the grid search):

- Experience Replay Size: 5000
- Number of Episodes: 20000
- Epsilon Decay Rate: 0.95
- Initial Epsilon: 1
- Epsilon decay rule: `epsilon *= epsilon_decay_rate`

- Analysis (in 3 sentences):

The agent achieved an average reward per episode of 129.6, which is the highest among the tested combinations. During training, the average reward was below 50 at first but increased rapidly after 10000 episodes. The average reward when testing is 195.25 and the generated video lasted for 9 seconds, showing the effectiveness of the agent.

# a4\_GCN

April 10, 2023

## 1 import modules and set random seed

```
[ ]: import numpy as np
import scipy.sparse as sp
import torch
import pandas as pd
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import time

seed = 0

np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

## 2 Loading and preprocessing the data

```
[ ]: def encode_onehot(labels):
    # The classes must be sorted before encoding to enable static class encoding.
    # In other words, make sure the first class always maps to index 0.
    classes = sorted(list(set(labels)))
    classes_dict = {c: np.identity(len(classes))[i, :] for i, c in enumerate(classes)}
    labels_onehot = np.array(list(map(classes_dict.get, labels)),
                           dtype=np.int32)
    return labels_onehot

def load_data(path='./cora/', dataset="cora", training_samples=140):
```

```

"""Load citation network dataset (cora only for now)"""
print('Loading {} dataset...'.format(dataset))

idx_features_labels = np.genfromtxt("{}-{}.content".format(path, dataset),
                                    dtype=np.dtype(str))
features = sp.csr_matrix(idx_features_labels[:, 1:-1], dtype=np.float32)
labels = encode_onehot(idx_features_labels[:, -1])

# build graph
idx = np.array(idx_features_labels[:, 0], dtype=np.int32)
idx_map = {j: i for i, j in enumerate(idx)}
edges_unordered = np.genfromtxt("{}-{}.cites".format(path, dataset),
                                 dtype=np.int32)
edges = np.array(list(map(idx_map.get, edges_unordered.flatten()))),
                dtype=np.int32).reshape(edges_unordered.shape)
adj = sp.coo_matrix((np.ones(edges.shape[0]), (edges[:, 0], edges[:, 1])),
                     shape=(labels.shape[0], labels.shape[0]),
                     dtype=np.float32)

# build symmetric adjacency matrix
adj = adj + adj.T.multiply(adj.T > adj) - adj.multiply(adj.T > adj)

features = normalize(features)
adj = adj + sp.eye(adj.shape[0])
adj = normalize_adj(adj)

# Random indexes
idx_rand = torch.randperm(len(labels))
# Nodes for training
idx_train = idx_rand[:training_samples]
# Nodes for validation
idx_val= idx_rand[training_samples:]

adj = torch.FloatTensor(np.array(adj.todense()))
features = torch.FloatTensor(np.array(features.todense()))
labels = torch.LongTensor(np.where(labels)[1])

idx_train = torch.LongTensor(idx_train)
idx_val = torch.LongTensor(idx_val)

return adj, features, labels, idx_train, idx_val

def normalize_adj(mx):
    """symmetric normalization"""
    rowsum = np.array(mx.sum(1))
    r_inv_sqrt = np.power(rowsum, -0.5).flatten()
    r_inv_sqrt[np.isinf(r_inv_sqrt)] = 0.

```

```

r_mat_inv_sqrt = sp.diags(r_inv_sqrt)
return mx.dot(r_mat_inv_sqrt).transpose().dot(r_mat_inv_sqrt)

def normalize(mx):
    """Row-normalize sparse matrix"""
    rowsum = np.array(mx.sum(1))
    r_inv = np.power(rowsum, -1).flatten()
    r_inv[np.isinf(r_inv)] = 0.
    r_mat_inv = sp.diags(r_inv)
    mx = r_mat_inv.dot(mx)
    return mx

def accuracy(output, labels):
    preds = output.max(1)[1].type_as(labels)
    correct = preds.eq(labels).double()
    correct = correct.sum()
    return correct / len(labels)

```

## 2.1 check the data

```
[ ]: adj, features, labels, idx_train, idx_val = load_data()
```

Loading cora dataset...

```
[ ]: print(adj)
print(adj.shape)
```

```

tensor([[0.1667, 0.0000, 0.0000, ... , 0.0000, 0.0000, 0.0000],
        [0.0000, 0.5000, 0.0000, ... , 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.2000, ... , 0.0000, 0.0000, 0.0000],
        ... ,
        [0.0000, 0.0000, 0.0000, ... , 0.2000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, ... , 0.0000, 0.2000, 0.0000],
        [0.0000, 0.0000, 0.0000, ... , 0.0000, 0.0000, 0.2500]])
torch.Size([2708, 2708])

```

```
[ ]: print(features)
print(features.shape)
```

```

tensor([[0., 0., 0., ... , 0., 0., 0.],
        [0., 0., 0., ... , 0., 0., 0.],
        [0., 0., 0., ... , 0., 0., 0.],
        ... ,
        [0., 0., 0., ... , 0., 0., 0.],
        [0., 0., 0., ... , 0., 0., 0.],
        [0., 0., 0., ... , 0., 0., 0.]])

```

```
torch.Size([2708, 1433])
```

```
[ ]: print(labels)
      print(labels.unique())
      print(len(labels))

tensor([2, 5, 4, ..., 1, 0, 2])
tensor([0, 1, 2, 3, 4, 5, 6])
2708
```

```
[ ]: print(len(idx_train))  
      print(len(idx_val))
```

140  
2568

### 3 Vanilla GCN for node classification

### 3.1 Define Graph Convolution layer (Your Task)

This module takes  $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \dots, \vec{h}_N\}$  where  $\vec{h}_i \in \mathbb{R}^F$  as input and outputs  $\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \dots, \vec{h}'_N\}$ , where  $\vec{h}'_i \in \mathbb{R}^{F'}$ . 1. perform initial transformation:  $\mathbf{s} = \mathbf{W} \times \mathbf{h}^{(l)}$  2. multiply  $\mathbf{s}$  by normalized adjacency matrix:  $\mathbf{h}' = \mathbf{A} \times \mathbf{s}$

```

# hint: use the linear layer you declared above.
# hint: you can use torch.spmm() sparse matrix multiplication to handle the
#       adjacency matrix
#####
s = self.linear(input)
h_prime = torch.spmm(adj, s)

return h_prime
#####

```

## 3.2 Define GCN (Your Task)

you will implement a two-layer GCN with ReLU activation function and Dropout after the first Conv layer.

```

[ ]: import torch.nn.functional as F

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()
        #####
        self.gc1 = GraphConvolution(nfeat, n_hidden, bias=bias)
        self.gc2 = GraphConvolution(n_hidden, n_classes, bias=bias)
        self.dropout = nn.Dropout(dropout)
        self.activation = 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.
#####
x = self.gc1(x, adj)
x = self.activation(x)
x = self.dropout(x)
x = self.gc2(x, adj)
return x
#####

```

### 3.3 define loss function

```
[ ]: criterion = nn.CrossEntropyLoss()
```

### 3.4 training loop

```
[ ]: args = {"training_samples": 140,
            "epochs": 100,
            "lr": 0.01,
            "weight_decay": 5e-4,
            "hidden": 16,
            "dropout": 0.5,
            "bias": True,
        }
```

```
[ ]: def train(epoch):
    t = time.time()
    model.train()
    optimizer.zero_grad()
    output = model(features, adj)
    loss_train = criterion(output[idx_train], labels[idx_train])
    acc_train = accuracy(output[idx_train], labels[idx_train])
    loss_train.backward()
    optimizer.step()

    model.eval()
    output = model(features, adj)

    loss_val = criterion(output[idx_val], labels[idx_val])
    acc_val = accuracy(output[idx_val], labels[idx_val])
    print('Epoch: {:04d}'.format(epoch+1),
          'loss_train: {:.4f}'.format(loss_train.item()),
          'acc_train: {:.4f}'.format(acc_train.item()),
          'loss_val: {:.4f}'.format(loss_val.item()),
          'acc_val: {:.4f}'.format(acc_val.item()),
          'time: {:.4f}s'.format(time.time() - t))
```

```

def test():
    model.eval()
    output = model(features, adj)
    loss_test = criterion(output[idx_val], labels[idx_val])
    acc_test = accuracy(output[idx_val], labels[idx_val])
    print("Test set results:",
          "loss= {:.4f}".format(loss_test.item()),
          "accuracy= {:.4f}".format(acc_test.item()))

```

```

[ ]: model = GCN(nfeat=features.shape[1],
                  n_hidden=args["hidden"],
                  n_classes=labels.max().item() + 1,
                  dropout=args["dropout"]).to(device)
optimizer = optim.Adam(model.parameters(),
                      lr=args["lr"], weight_decay=args["weight_decay"])

adj, features, labels, idx_train, idx_val = \
    load_data(training_samples=args["training_samples"])
adj, features, labels, idx_train, idx_val = adj.to(device), features.
    to(device), labels.to(device), idx_train.to(device), idx_val.to(device)

```

Loading cora dataset...

### 3.5 training Vanilla GCN

```

[ ]: # Train model
t_total = time.time()
for epoch in range(args["epochs"]):
    train(epoch)
print("Optimization Finished!")
print("Total time elapsed: {:.4f}s".format(time.time() - t_total))

# evaluating
test()

```

```

Epoch: 0001 loss_train: 1.9387 acc_train: 0.1143 loss_val: 1.9272 acc_val:
0.1597 time: 0.6393s
Epoch: 0002 loss_train: 1.9319 acc_train: 0.1143 loss_val: 1.9223 acc_val:
0.1597 time: 0.0019s
Epoch: 0003 loss_train: 1.9245 acc_train: 0.1143 loss_val: 1.9165 acc_val:
0.1663 time: 0.0018s
Epoch: 0004 loss_train: 1.9154 acc_train: 0.1786 loss_val: 1.9103 acc_val:
0.2134 time: 0.0017s
Epoch: 0005 loss_train: 1.9070 acc_train: 0.2000 loss_val: 1.9037 acc_val:

```

0.1340 time: 0.0017s  
Epoch: 0006 loss\_train: 1.8966 acc\_train: 0.2071 loss\_val: 1.8968 acc\_val:  
0.1464 time: 0.0018s  
Epoch: 0007 loss\_train: 1.8865 acc\_train: 0.2429 loss\_val: 1.8893 acc\_val:  
0.3470 time: 0.0017s  
Epoch: 0008 loss\_train: 1.8768 acc\_train: 0.4000 loss\_val: 1.8815 acc\_val:  
0.3520 time: 0.0017s  
Epoch: 0009 loss\_train: 1.8699 acc\_train: 0.3857 loss\_val: 1.8735 acc\_val:  
0.3104 time: 0.0017s  
Epoch: 0010 loss\_train: 1.8577 acc\_train: 0.3786 loss\_val: 1.8654 acc\_val:  
0.3018 time: 0.0017s  
Epoch: 0011 loss\_train: 1.8397 acc\_train: 0.3500 loss\_val: 1.8571 acc\_val:  
0.3014 time: 0.0017s  
Epoch: 0012 loss\_train: 1.8268 acc\_train: 0.3429 loss\_val: 1.8487 acc\_val:  
0.3014 time: 0.0017s  
Epoch: 0013 loss\_train: 1.8119 acc\_train: 0.3286 loss\_val: 1.8403 acc\_val:  
0.3014 time: 0.0017s  
Epoch: 0014 loss\_train: 1.8050 acc\_train: 0.3286 loss\_val: 1.8320 acc\_val:  
0.3010 time: 0.0017s  
Epoch: 0015 loss\_train: 1.7883 acc\_train: 0.3286 loss\_val: 1.8240 acc\_val:  
0.3010 time: 0.0018s  
Epoch: 0016 loss\_train: 1.7659 acc\_train: 0.3214 loss\_val: 1.8162 acc\_val:  
0.3010 time: 0.0020s  
Epoch: 0017 loss\_train: 1.7582 acc\_train: 0.3214 loss\_val: 1.8088 acc\_val:  
0.3010 time: 0.0020s  
Epoch: 0018 loss\_train: 1.7511 acc\_train: 0.3214 loss\_val: 1.8020 acc\_val:  
0.3010 time: 0.0020s  
Epoch: 0019 loss\_train: 1.7432 acc\_train: 0.3214 loss\_val: 1.7958 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0020 loss\_train: 1.7293 acc\_train: 0.3286 loss\_val: 1.7901 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0021 loss\_train: 1.7062 acc\_train: 0.3214 loss\_val: 1.7852 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0022 loss\_train: 1.6965 acc\_train: 0.3214 loss\_val: 1.7807 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0023 loss\_train: 1.7003 acc\_train: 0.3214 loss\_val: 1.7764 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0024 loss\_train: 1.6738 acc\_train: 0.3214 loss\_val: 1.7720 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0025 loss\_train: 1.6729 acc\_train: 0.3214 loss\_val: 1.7672 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0026 loss\_train: 1.6607 acc\_train: 0.3214 loss\_val: 1.7615 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0027 loss\_train: 1.6592 acc\_train: 0.3214 loss\_val: 1.7550 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0028 loss\_train: 1.6370 acc\_train: 0.3286 loss\_val: 1.7471 acc\_val:  
0.3010 time: 0.0019s  
Epoch: 0029 loss\_train: 1.6376 acc\_train: 0.3214 loss\_val: 1.7383 acc\_val:

```
0.3014 time: 0.0020s
Epoch: 0030 loss_train: 1.6031 acc_train: 0.3357 loss_val: 1.7287 acc_val:
0.3014 time: 0.0020s
Epoch: 0031 loss_train: 1.6168 acc_train: 0.3357 loss_val: 1.7183 acc_val:
0.3014 time: 0.0020s
Epoch: 0032 loss_train: 1.5878 acc_train: 0.3214 loss_val: 1.7078 acc_val:
0.3014 time: 0.0020s
Epoch: 0033 loss_train: 1.5753 acc_train: 0.3429 loss_val: 1.6968 acc_val:
0.3026 time: 0.0020s
Epoch: 0034 loss_train: 1.5645 acc_train: 0.3500 loss_val: 1.6854 acc_val:
0.3030 time: 0.0020s
Epoch: 0035 loss_train: 1.5452 acc_train: 0.3571 loss_val: 1.6742 acc_val:
0.3061 time: 0.0020s
Epoch: 0036 loss_train: 1.5276 acc_train: 0.3929 loss_val: 1.6630 acc_val:
0.3135 time: 0.0020s
Epoch: 0037 loss_train: 1.5100 acc_train: 0.3857 loss_val: 1.6517 acc_val:
0.3217 time: 0.0020s
Epoch: 0038 loss_train: 1.5060 acc_train: 0.4214 loss_val: 1.6402 acc_val:
0.3322 time: 0.0020s
Epoch: 0039 loss_train: 1.4978 acc_train: 0.4714 loss_val: 1.6287 acc_val:
0.3450 time: 0.0020s
Epoch: 0040 loss_train: 1.4586 acc_train: 0.4714 loss_val: 1.6172 acc_val:
0.3586 time: 0.0020s
Epoch: 0041 loss_train: 1.4376 acc_train: 0.5286 loss_val: 1.6057 acc_val:
0.3676 time: 0.0020s
Epoch: 0042 loss_train: 1.4544 acc_train: 0.4714 loss_val: 1.5944 acc_val:
0.3816 time: 0.0020s
Epoch: 0043 loss_train: 1.4297 acc_train: 0.5000 loss_val: 1.5831 acc_val:
0.3894 time: 0.0020s
Epoch: 0044 loss_train: 1.4057 acc_train: 0.5429 loss_val: 1.5720 acc_val:
0.3984 time: 0.0019s
Epoch: 0045 loss_train: 1.4023 acc_train: 0.4857 loss_val: 1.5613 acc_val:
0.4026 time: 0.0019s
Epoch: 0046 loss_train: 1.3701 acc_train: 0.5214 loss_val: 1.5510 acc_val:
0.4065 time: 0.0019s
Epoch: 0047 loss_train: 1.3503 acc_train: 0.5500 loss_val: 1.5409 acc_val:
0.4116 time: 0.0019s
Epoch: 0048 loss_train: 1.3406 acc_train: 0.5571 loss_val: 1.5308 acc_val:
0.4178 time: 0.0019s
Epoch: 0049 loss_train: 1.3308 acc_train: 0.5643 loss_val: 1.5209 acc_val:
0.4264 time: 0.0019s
Epoch: 0050 loss_train: 1.3267 acc_train: 0.5357 loss_val: 1.5107 acc_val:
0.4350 time: 0.0019s
Epoch: 0051 loss_train: 1.3009 acc_train: 0.5714 loss_val: 1.5009 acc_val:
0.4470 time: 0.0019s
Epoch: 0052 loss_train: 1.3013 acc_train: 0.5429 loss_val: 1.4909 acc_val:
0.4579 time: 0.0019s
Epoch: 0053 loss_train: 1.2545 acc_train: 0.5643 loss_val: 1.4805 acc_val:
```

0.4727 time: 0.0019s  
Epoch: 0054 loss\_train: 1.2635 acc\_train: 0.6000 loss\_val: 1.4701 acc\_val:  
0.4817 time: 0.0019s  
Epoch: 0055 loss\_train: 1.2263 acc\_train: 0.6357 loss\_val: 1.4597 acc\_val:  
0.4945 time: 0.0019s  
Epoch: 0056 loss\_train: 1.2386 acc\_train: 0.6214 loss\_val: 1.4491 acc\_val:  
0.5000 time: 0.0019s  
Epoch: 0057 loss\_train: 1.2255 acc\_train: 0.5857 loss\_val: 1.4378 acc\_val:  
0.5047 time: 0.0019s  
Epoch: 0058 loss\_train: 1.1820 acc\_train: 0.6500 loss\_val: 1.4263 acc\_val:  
0.5082 time: 0.0019s  
Epoch: 0059 loss\_train: 1.1844 acc\_train: 0.6429 loss\_val: 1.4146 acc\_val:  
0.5113 time: 0.0019s  
Epoch: 0060 loss\_train: 1.1836 acc\_train: 0.6286 loss\_val: 1.4032 acc\_val:  
0.5148 time: 0.0019s  
Epoch: 0061 loss\_train: 1.1650 acc\_train: 0.6214 loss\_val: 1.3924 acc\_val:  
0.5202 time: 0.0019s  
Epoch: 0062 loss\_train: 1.1318 acc\_train: 0.6571 loss\_val: 1.3822 acc\_val:  
0.5249 time: 0.0019s  
Epoch: 0063 loss\_train: 1.1342 acc\_train: 0.6357 loss\_val: 1.3718 acc\_val:  
0.5284 time: 0.0019s  
Epoch: 0064 loss\_train: 1.1161 acc\_train: 0.6429 loss\_val: 1.3617 acc\_val:  
0.5319 time: 0.0019s  
Epoch: 0065 loss\_train: 1.1280 acc\_train: 0.6500 loss\_val: 1.3519 acc\_val:  
0.5378 time: 0.0019s  
Epoch: 0066 loss\_train: 1.0710 acc\_train: 0.6857 loss\_val: 1.3423 acc\_val:  
0.5456 time: 0.0019s  
Epoch: 0067 loss\_train: 1.0775 acc\_train: 0.6571 loss\_val: 1.3326 acc\_val:  
0.5537 time: 0.0019s  
Epoch: 0068 loss\_train: 1.0435 acc\_train: 0.6857 loss\_val: 1.3231 acc\_val:  
0.5592 time: 0.0019s  
Epoch: 0069 loss\_train: 1.0277 acc\_train: 0.6929 loss\_val: 1.3136 acc\_val:  
0.5646 time: 0.0019s  
Epoch: 0070 loss\_train: 1.0202 acc\_train: 0.6714 loss\_val: 1.3041 acc\_val:  
0.5674 time: 0.0019s  
Epoch: 0071 loss\_train: 1.0473 acc\_train: 0.6857 loss\_val: 1.2939 acc\_val:  
0.5717 time: 0.0019s  
Epoch: 0072 loss\_train: 0.9992 acc\_train: 0.7214 loss\_val: 1.2845 acc\_val:  
0.5748 time: 0.0019s  
Epoch: 0073 loss\_train: 0.9883 acc\_train: 0.7286 loss\_val: 1.2754 acc\_val:  
0.5767 time: 0.0019s  
Epoch: 0074 loss\_train: 0.9673 acc\_train: 0.7214 loss\_val: 1.2660 acc\_val:  
0.5775 time: 0.0019s  
Epoch: 0075 loss\_train: 0.9550 acc\_train: 0.7357 loss\_val: 1.2560 acc\_val:  
0.5810 time: 0.0019s  
Epoch: 0076 loss\_train: 0.9518 acc\_train: 0.7143 loss\_val: 1.2468 acc\_val:  
0.5837 time: 0.0019s  
Epoch: 0077 loss\_train: 0.9441 acc\_train: 0.7357 loss\_val: 1.2378 acc\_val:

```
0.5872 time: 0.0019s
Epoch: 0078 loss_train: 0.9319 acc_train: 0.7571 loss_val: 1.2292 acc_val:
0.5892 time: 0.0019s
Epoch: 0079 loss_train: 0.9036 acc_train: 0.7643 loss_val: 1.2215 acc_val:
0.5919 time: 0.0019s
Epoch: 0080 loss_train: 0.9106 acc_train: 0.7429 loss_val: 1.2132 acc_val:
0.5970 time: 0.0019s
Epoch: 0081 loss_train: 0.9149 acc_train: 0.7429 loss_val: 1.2047 acc_val:
0.6001 time: 0.0019s
Epoch: 0082 loss_train: 0.8926 acc_train: 0.7714 loss_val: 1.1965 acc_val:
0.6024 time: 0.0019s
Epoch: 0083 loss_train: 0.9165 acc_train: 0.7500 loss_val: 1.1878 acc_val:
0.6063 time: 0.0019s
Epoch: 0084 loss_train: 0.8798 acc_train: 0.7714 loss_val: 1.1790 acc_val:
0.6079 time: 0.0019s
Epoch: 0085 loss_train: 0.8747 acc_train: 0.7786 loss_val: 1.1702 acc_val:
0.6098 time: 0.0019s
Epoch: 0086 loss_train: 0.8627 acc_train: 0.7929 loss_val: 1.1616 acc_val:
0.6133 time: 0.0019s
Epoch: 0087 loss_train: 0.8612 acc_train: 0.7929 loss_val: 1.1537 acc_val:
0.6153 time: 0.0019s
Epoch: 0088 loss_train: 0.8698 acc_train: 0.8071 loss_val: 1.1461 acc_val:
0.6160 time: 0.0019s
Epoch: 0089 loss_train: 0.8098 acc_train: 0.8357 loss_val: 1.1387 acc_val:
0.6195 time: 0.0019s
Epoch: 0090 loss_train: 0.8286 acc_train: 0.8286 loss_val: 1.1313 acc_val:
0.6223 time: 0.0019s
Epoch: 0091 loss_train: 0.7919 acc_train: 0.8286 loss_val: 1.1238 acc_val:
0.6254 time: 0.0019s
Epoch: 0092 loss_train: 0.7797 acc_train: 0.8429 loss_val: 1.1160 acc_val:
0.6293 time: 0.0019s
Epoch: 0093 loss_train: 0.7539 acc_train: 0.8786 loss_val: 1.1086 acc_val:
0.6336 time: 0.0019s
Epoch: 0094 loss_train: 0.8075 acc_train: 0.8286 loss_val: 1.1020 acc_val:
0.6386 time: 0.0019s
Epoch: 0095 loss_train: 0.7681 acc_train: 0.8500 loss_val: 1.0956 acc_val:
0.6421 time: 0.0019s
Epoch: 0096 loss_train: 0.7314 acc_train: 0.8286 loss_val: 1.0892 acc_val:
0.6452 time: 0.0019s
Epoch: 0097 loss_train: 0.7305 acc_train: 0.8786 loss_val: 1.0826 acc_val:
0.6491 time: 0.0019s
Epoch: 0098 loss_train: 0.7108 acc_train: 0.8786 loss_val: 1.0760 acc_val:
0.6515 time: 0.0019s
Epoch: 0099 loss_train: 0.7099 acc_train: 0.8714 loss_val: 1.0703 acc_val:
0.6488 time: 0.0019s
Epoch: 0100 loss_train: 0.7307 acc_train: 0.8286 loss_val: 1.0643 acc_val:
0.6515 time: 0.0019s
Optimization Finished!
```

```
Total time elapsed: 0.8315s
Test set results: loss= 1.0643 accuracy= 0.6515
```

## 4 Graph Attention Networks

### 4.1 Graph attention layer (Your task)

A GAT is made up of multiple such layers. In this section, you will implement a single graph attention layer. Similar to the `GraphConvolution()`, this `GraphAttentionLayer()` module takes  $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \dots, \vec{h}_N\}$  where  $\vec{h}_i \in \mathbb{R}^F$  as input and outputs  $\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \dots, \vec{h}'_N\}$ , where  $\vec{h}'_i \in \mathbb{R}^{F'}$ . However, instead of weighing each neighbouring node based on the adjacency matrix, we will use self attention to learn the relative importance of each neighbouring node. Recall from HW4 where you are asked to write out the equation for single headed attention, here we will implement multi-headed attention, which involves the following steps:

#### 4.1.1 The initial transformation

In GCN above, you have completed similar transformation. But here, we need to define a weight matrix and perform this transformation for each head:  $\vec{s}_i^k = \mathbf{W}^k \vec{h}_i$ . We will perform a single linear transformation and then split it up for each head later. Note the input  $\vec{h}$  has shape `[n_nodes, in_features]` and  $\vec{s}$  has shape of `[n_nodes, n_heads * n_hidden]`. Remember to reshape  $\vec{s}$  has shape of `[n_nodes, n_heads, n_hidden]` for later uses. Note: set `bias=False` for this linear transformation.

#### 4.1.2 attention score

We calculate these for each head  $k$ . Here for simplicity of the notation, we omit  $k$  in the following equations. The attention scores are defined as the follows:

$$e_{ij} = a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j) = a(\vec{s}_i, \vec{s}_j)$$

, where  $e_{ij}$  is the attention score (importance) of node  $j$  to node  $i$ . We will have to calculate this for each head.  $a$  is the attention mechanism, that calculates the attention score. The paper concatenates  $\vec{s}_i$ ,  $\vec{s}_j$  and does a linear transformation with a weight vector  $\mathbf{a} \in \mathbb{R}^{2F'}$  followed by a LeakyReLU.

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^\top [\vec{s}_i \| \vec{s}_j]\right)$$

#### How to vectorize this? Some hints:

1. `tensor.repeat()` gives you  $\{\vec{s}_1, \vec{s}_2, \dots, \vec{s}_N, \vec{s}_1, \vec{s}_2, \dots, \vec{s}_N, \dots\}$ .
2. `tensor.repeat_interleave()` gives you  $\{\vec{s}_1, \vec{s}_1, \dots, \vec{s}_1, \vec{s}_2, \vec{s}_2, \dots, \vec{s}_2, \dots\}$ .
3. concatenate to get  $[\vec{s}_i \| \vec{s}_j]$  for all pairs of  $i, j$ . Reshape  $\vec{s}_i \| \vec{s}_j$  has shape of `[n_nodes, n_nodes, n_heads, 2 * n_hidden]`
4. apply the attention layer and non-linear activation function to get  $e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^\top [\vec{s}_i \| \vec{s}_j]\right)$ , where  $\mathbf{a}^\top$  is a single linear transformation that maps from dimension `n_hidden * 2` to 1. Note: set the `bias=False` for this linear transformation.

`e` is of shape `[n_nodes, n_nodes, n_heads, 1]`. Remove the last dimension `1` using `squeeze()`.

**Perform softmax** First, we need to mask  $e_{ij}$  based on adjacency matrix. We only need to sum over the neighbouring nodes for the attention calculation. Set the elements in  $e_{ij}$  to  $-\infty$  if there is no edge from  $i$  to  $j$  for the softmax calculation. We need to do this for all heads and the adjacency matrix is the same for each head. Use `tensor.masked_fill()` to mask  $e_{ij}$  based on adjacency matrix for all heads. Hint: reshape the adjacency matrix to `[n_nodes, n_nodes, 1]` using `unsqueeze()`. Now we are ready to normalize attention scores (or coefficients)

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

**Apply dropout** Apply the dropout layer. (this step is easy)

**Calculate final output for each head**

$$\overrightarrow{h'_i} = \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \overrightarrow{s_j^k}$$

**Concat or Mean** Finally we concateneate the transformed features:  $\overrightarrow{h_i'} = \left\|_{k=1}^K \overrightarrow{h_i'^k} \right\|$ . In the code, we only need to reshape the tensor to shape of `[n_nodes, n_heads * n_hidden]`. Note that if it is the final layer, then it doesn't make sense to do concatenation anymore. Instead, we sum over the `n_heads` dimension:  $\overrightarrow{h_i'} = \frac{1}{K} \sum_{k=1}^K \overrightarrow{h_i'^k}$ .

```
[ ]: 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 = False)
    # (2) self.attention: Linear layer that compute the attention score (set bias = False)
    # (3) self.activation: Activation function (LeakyReLU with 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(2 * self.n_hidden, 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
#####
    s = self.W(h).view(n_nodes, self.n_heads, self.n_hidden)

```

```

        s_i = s.repeat(1, n_nodes, 1).view(n_nodes, n_nodes, self.n_heads, self.
        ↵n_hidden)
        s_j = s.repeat_interleave(n_nodes, dim=0).view(n_nodes, n_nodes, self.
        ↵n_heads, self.n_hidden)
        s_cat = torch.cat([s_i, s_j], dim=-1)
        e = self.attention(s_cat)
        e = self.activation(e).squeeze(-1)
        adj_mask = adj_mat.unsqueeze(-1).expand_as(e)
        e = e.masked_fill(adj_mask == 0, float('-inf'))
        a = self.softmax(e)
        a = self.dropout_layer(a)
        ##### Your code here #####
        # 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 #####
        ##### Your code here #####
        h_prime = h_prime.reshape(n_nodes, -1)
        # Take the mean of the heads (for the last layer)
        else:
            ##### Your code here #####
        ##### Your code here #####
        h_prime = h_prime.mean(dim=1)
        # Your code here #####
        return h_prime
    
```

## 4.2 Define GAT network

it's really similar to how we defined GCN. We followed the paper to use two attention layers and ELU() activation function.

```
[ ]: class GAT(nn.Module):

    def __init__(self, nfeat: int, n_hidden: int, n_classes: int, n_heads: int, ↵
    ↵dropout: float, alpha: float):
        """
        in_features: the number of features per node
    
```

```

n_hidden: the number of features in the first graph attention layer
n_classes: the number of classes
n_heads: the number of heads in the graph attention layers
dropout: the dropout probability
alpha: the negative input slope for leaky ReLU of the attention layer
"""

super().__init__()

# First graph attention layer where we concatenate the heads
self.gc1 = GraphAttentionLayer(nfeat, n_hidden, n_heads, □
↳is_concat=True, dropout=dropout, alpha=alpha)
    self.gc2 = GraphAttentionLayer(n_hidden, n_classes, 1, is_concat=False, □
↳dropout=dropout, alpha=alpha)
        self.activation = nn.ELU()
        self.dropout = nn.Dropout(dropout)

def forward(self, x: torch.Tensor, adj_mat: torch.Tensor):
    """
    x: the features vectors
    adj_mat: the adjacency matrix
    """
    x = self.dropout(x)
    x = self.gc1(x, adj_mat)
    x = self.activation(x)
    x = self.dropout(x)
    x = self.gc2(x, adj_mat)
    return x

```

### 4.3 training GAT

```
[ ]: args = {"training_samples": 140,
            "epochs": 100,
            "lr": 0.01,
            "weight_decay": 5e-4,
            "hidden": 16,
            "dropout": 0.5,
            "bias": True,
            "alpha": 0.2,
            "n_heads": 8
        }
```

```
[ ]: model = GAT(nfeat=features.shape[1],
                  n_hidden=args["hidden"],
                  n_classes=labels.max().item() + 1,
                  dropout=args["dropout"],
                  alpha=args["alpha"],
```

```

        n_heads=args["n_heads"]).to(device)
optimizer = optim.Adam(model.parameters(),
                      lr=args["lr"], weight_decay=args["weight_decay"])

adj, features, labels, idx_train, idx_val = \
    load_data(training_samples=args["training_samples"])
adj, features, labels, idx_train, idx_val = adj.to(device), features.
    .to(device), labels.to(device), idx_train.to(device), idx_val.to(device)

```

Loading cora dataset...

```
[ ]: # Train model
t_total = time.time()
for epoch in range(args["epochs"]):
    train(epoch)
print("Optimization Finished!")
print("Total time elapsed: {:.4f}s".format(time.time() - t_total))

# Testing
test()
```

```

Epoch: 0001 loss_train: 1.9459 acc_train: 0.1714 loss_val: 1.9428 acc_val:
0.3668 time: 0.0532s
Epoch: 0002 loss_train: 1.9416 acc_train: 0.3500 loss_val: 1.9397 acc_val:
0.4225 time: 0.0509s
Epoch: 0003 loss_train: 1.9384 acc_train: 0.5000 loss_val: 1.9364 acc_val:
0.4447 time: 0.0508s
Epoch: 0004 loss_train: 1.9323 acc_train: 0.4786 loss_val: 1.9328 acc_val:
0.4463 time: 0.0507s
Epoch: 0005 loss_train: 1.9244 acc_train: 0.5429 loss_val: 1.9288 acc_val:
0.4393 time: 0.0499s
Epoch: 0006 loss_train: 1.9235 acc_train: 0.5143 loss_val: 1.9245 acc_val:
0.4283 time: 0.0499s
Epoch: 0007 loss_train: 1.9180 acc_train: 0.5000 loss_val: 1.9198 acc_val:
0.4198 time: 0.0499s
Epoch: 0008 loss_train: 1.9084 acc_train: 0.5000 loss_val: 1.9148 acc_val:
0.4159 time: 0.0499s
Epoch: 0009 loss_train: 1.9080 acc_train: 0.4429 loss_val: 1.9095 acc_val:
0.4124 time: 0.0499s
Epoch: 0010 loss_train: 1.8956 acc_train: 0.5000 loss_val: 1.9039 acc_val:
0.4112 time: 0.0499s
Epoch: 0011 loss_train: 1.8901 acc_train: 0.4143 loss_val: 1.8979 acc_val:
0.4097 time: 0.0498s
Epoch: 0012 loss_train: 1.8823 acc_train: 0.5071 loss_val: 1.8915 acc_val:
0.4081 time: 0.0499s
Epoch: 0013 loss_train: 1.8660 acc_train: 0.4857 loss_val: 1.8848 acc_val:
0.4069 time: 0.0499s

```

```
Epoch: 0014 loss_train: 1.8700 acc_train: 0.4286 loss_val: 1.8778 acc_val:  
0.4058 time: 0.0499s  
Epoch: 0015 loss_train: 1.8625 acc_train: 0.4214 loss_val: 1.8705 acc_val:  
0.4058 time: 0.0499s  
Epoch: 0016 loss_train: 1.8460 acc_train: 0.4929 loss_val: 1.8629 acc_val:  
0.4058 time: 0.0501s  
Epoch: 0017 loss_train: 1.8347 acc_train: 0.4429 loss_val: 1.8549 acc_val:  
0.4069 time: 0.0499s  
Epoch: 0018 loss_train: 1.8219 acc_train: 0.4357 loss_val: 1.8467 acc_val:  
0.4089 time: 0.0499s  
Epoch: 0019 loss_train: 1.8196 acc_train: 0.4714 loss_val: 1.8383 acc_val:  
0.4093 time: 0.0498s  
Epoch: 0020 loss_train: 1.7920 acc_train: 0.4714 loss_val: 1.8296 acc_val:  
0.4112 time: 0.0499s  
Epoch: 0021 loss_train: 1.7914 acc_train: 0.5071 loss_val: 1.8205 acc_val:  
0.4120 time: 0.0499s  
Epoch: 0022 loss_train: 1.7659 acc_train: 0.4500 loss_val: 1.8110 acc_val:  
0.4139 time: 0.0499s  
Epoch: 0023 loss_train: 1.7717 acc_train: 0.5000 loss_val: 1.8014 acc_val:  
0.4147 time: 0.0498s  
Epoch: 0024 loss_train: 1.7546 acc_train: 0.4714 loss_val: 1.7916 acc_val:  
0.4167 time: 0.0499s  
Epoch: 0025 loss_train: 1.7439 acc_train: 0.4571 loss_val: 1.7815 acc_val:  
0.4182 time: 0.0499s  
Epoch: 0026 loss_train: 1.7166 acc_train: 0.4429 loss_val: 1.7713 acc_val:  
0.4213 time: 0.0499s  
Epoch: 0027 loss_train: 1.7134 acc_train: 0.4857 loss_val: 1.7608 acc_val:  
0.4229 time: 0.0498s  
Epoch: 0028 loss_train: 1.7030 acc_train: 0.4929 loss_val: 1.7501 acc_val:  
0.4272 time: 0.0499s  
Epoch: 0029 loss_train: 1.6917 acc_train: 0.5500 loss_val: 1.7392 acc_val:  
0.4350 time: 0.0499s  
Epoch: 0030 loss_train: 1.6668 acc_train: 0.5429 loss_val: 1.7281 acc_val:  
0.4404 time: 0.0499s  
Epoch: 0031 loss_train: 1.6433 acc_train: 0.5214 loss_val: 1.7168 acc_val:  
0.4451 time: 0.0498s  
Epoch: 0032 loss_train: 1.6785 acc_train: 0.4786 loss_val: 1.7055 acc_val:  
0.4509 time: 0.0499s  
Epoch: 0033 loss_train: 1.6309 acc_train: 0.4929 loss_val: 1.6942 acc_val:  
0.4587 time: 0.0499s  
Epoch: 0034 loss_train: 1.6049 acc_train: 0.5643 loss_val: 1.6827 acc_val:  
0.4630 time: 0.0499s  
Epoch: 0035 loss_train: 1.5749 acc_train: 0.5714 loss_val: 1.6711 acc_val:  
0.4681 time: 0.0500s  
Epoch: 0036 loss_train: 1.6210 acc_train: 0.5000 loss_val: 1.6594 acc_val:  
0.4774 time: 0.0499s  
Epoch: 0037 loss_train: 1.5754 acc_train: 0.5929 loss_val: 1.6478 acc_val:  
0.4864 time: 0.0500s
```

```
Epoch: 0038 loss_train: 1.5624 acc_train: 0.5214 loss_val: 1.6361 acc_val:  
0.4953 time: 0.0500s  
Epoch: 0039 loss_train: 1.5357 acc_train: 0.5143 loss_val: 1.6244 acc_val:  
0.5012 time: 0.0499s  
Epoch: 0040 loss_train: 1.5533 acc_train: 0.5500 loss_val: 1.6127 acc_val:  
0.5090 time: 0.0511s  
Epoch: 0041 loss_train: 1.5226 acc_train: 0.5571 loss_val: 1.6011 acc_val:  
0.5175 time: 0.0500s  
Epoch: 0042 loss_train: 1.5116 acc_train: 0.5857 loss_val: 1.5893 acc_val:  
0.5265 time: 0.0499s  
Epoch: 0043 loss_train: 1.4494 acc_train: 0.6000 loss_val: 1.5774 acc_val:  
0.5389 time: 0.0499s  
Epoch: 0044 loss_train: 1.4497 acc_train: 0.5857 loss_val: 1.5655 acc_val:  
0.5467 time: 0.0498s  
Epoch: 0045 loss_train: 1.4168 acc_train: 0.6214 loss_val: 1.5534 acc_val:  
0.5615 time: 0.0498s  
Epoch: 0046 loss_train: 1.4272 acc_train: 0.6214 loss_val: 1.5412 acc_val:  
0.5744 time: 0.0499s  
Epoch: 0047 loss_train: 1.3917 acc_train: 0.6500 loss_val: 1.5291 acc_val:  
0.5841 time: 0.0499s  
Epoch: 0048 loss_train: 1.3805 acc_train: 0.6357 loss_val: 1.5169 acc_val:  
0.5931 time: 0.0499s  
Epoch: 0049 loss_train: 1.4200 acc_train: 0.6500 loss_val: 1.5050 acc_val:  
0.6001 time: 0.0498s  
Epoch: 0050 loss_train: 1.4262 acc_train: 0.6143 loss_val: 1.4931 acc_val:  
0.6071 time: 0.0499s  
Epoch: 0051 loss_train: 1.3705 acc_train: 0.6929 loss_val: 1.4814 acc_val:  
0.6137 time: 0.0499s  
Epoch: 0052 loss_train: 1.3129 acc_train: 0.6786 loss_val: 1.4696 acc_val:  
0.6199 time: 0.0499s  
Epoch: 0053 loss_train: 1.3533 acc_train: 0.6571 loss_val: 1.4579 acc_val:  
0.6277 time: 0.0498s  
Epoch: 0054 loss_train: 1.3056 acc_train: 0.7143 loss_val: 1.4464 acc_val:  
0.6343 time: 0.0500s  
Epoch: 0055 loss_train: 1.3201 acc_train: 0.6786 loss_val: 1.4349 acc_val:  
0.6394 time: 0.0499s  
Epoch: 0056 loss_train: 1.3255 acc_train: 0.7143 loss_val: 1.4234 acc_val:  
0.6468 time: 0.0499s  
Epoch: 0057 loss_train: 1.3180 acc_train: 0.6714 loss_val: 1.4121 acc_val:  
0.6526 time: 0.0499s  
Epoch: 0058 loss_train: 1.3105 acc_train: 0.6714 loss_val: 1.4010 acc_val:  
0.6573 time: 0.0499s  
Epoch: 0059 loss_train: 1.2314 acc_train: 0.6857 loss_val: 1.3901 acc_val:  
0.6624 time: 0.0499s  
Epoch: 0060 loss_train: 1.2366 acc_train: 0.6929 loss_val: 1.3792 acc_val:  
0.6682 time: 0.0499s  
Epoch: 0061 loss_train: 1.2142 acc_train: 0.7143 loss_val: 1.3683 acc_val:  
0.6706 time: 0.0499s
```

```
Epoch: 0062 loss_train: 1.2960 acc_train: 0.7143 loss_val: 1.3577 acc_val:  
0.6741 time: 0.0499s  
Epoch: 0063 loss_train: 1.1877 acc_train: 0.7357 loss_val: 1.3470 acc_val:  
0.6791 time: 0.0499s  
Epoch: 0064 loss_train: 1.2204 acc_train: 0.7571 loss_val: 1.3362 acc_val:  
0.6834 time: 0.0500s  
Epoch: 0065 loss_train: 1.2041 acc_train: 0.7214 loss_val: 1.3255 acc_val:  
0.6881 time: 0.0500s  
Epoch: 0066 loss_train: 1.1969 acc_train: 0.7214 loss_val: 1.3149 acc_val:  
0.6928 time: 0.0498s  
Epoch: 0067 loss_train: 1.1887 acc_train: 0.7071 loss_val: 1.3044 acc_val:  
0.6974 time: 0.0499s  
Epoch: 0068 loss_train: 1.1964 acc_train: 0.6929 loss_val: 1.2942 acc_val:  
0.6986 time: 0.0498s  
Epoch: 0069 loss_train: 1.1248 acc_train: 0.7643 loss_val: 1.2840 acc_val:  
0.7025 time: 0.0498s  
Epoch: 0070 loss_train: 1.1579 acc_train: 0.7429 loss_val: 1.2738 acc_val:  
0.7068 time: 0.0499s  
Epoch: 0071 loss_train: 1.1731 acc_train: 0.7500 loss_val: 1.2638 acc_val:  
0.7095 time: 0.0499s  
Epoch: 0072 loss_train: 1.0757 acc_train: 0.7643 loss_val: 1.2538 acc_val:  
0.7126 time: 0.0499s  
Epoch: 0073 loss_train: 1.1562 acc_train: 0.7214 loss_val: 1.2442 acc_val:  
0.7142 time: 0.0498s  
Epoch: 0074 loss_train: 1.0801 acc_train: 0.7786 loss_val: 1.2346 acc_val:  
0.7185 time: 0.0499s  
Epoch: 0075 loss_train: 1.0657 acc_train: 0.7857 loss_val: 1.2250 acc_val:  
0.7192 time: 0.0499s  
Epoch: 0076 loss_train: 1.0951 acc_train: 0.7571 loss_val: 1.2156 acc_val:  
0.7200 time: 0.0499s  
Epoch: 0077 loss_train: 1.0898 acc_train: 0.7571 loss_val: 1.2063 acc_val:  
0.7204 time: 0.0498s  
Epoch: 0078 loss_train: 1.0882 acc_train: 0.7071 loss_val: 1.1976 acc_val:  
0.7200 time: 0.0499s  
Epoch: 0079 loss_train: 1.0902 acc_train: 0.8000 loss_val: 1.1888 acc_val:  
0.7216 time: 0.0499s  
Epoch: 0080 loss_train: 1.0208 acc_train: 0.7357 loss_val: 1.1801 acc_val:  
0.7231 time: 0.0499s  
Epoch: 0081 loss_train: 1.0461 acc_train: 0.7929 loss_val: 1.1719 acc_val:  
0.7243 time: 0.0498s  
Epoch: 0082 loss_train: 1.0137 acc_train: 0.8286 loss_val: 1.1635 acc_val:  
0.7259 time: 0.0499s  
Epoch: 0083 loss_train: 1.0080 acc_train: 0.8000 loss_val: 1.1553 acc_val:  
0.7259 time: 0.0499s  
Epoch: 0084 loss_train: 1.0027 acc_train: 0.8000 loss_val: 1.1473 acc_val:  
0.7266 time: 0.0499s  
Epoch: 0085 loss_train: 1.0898 acc_train: 0.7571 loss_val: 1.1395 acc_val:  
0.7301 time: 0.0499s
```

```

Epoch: 0086 loss_train: 1.0332 acc_train: 0.7500 loss_val: 1.1319 acc_val:
0.7325 time: 0.0499s
Epoch: 0087 loss_train: 1.0661 acc_train: 0.7571 loss_val: 1.1245 acc_val:
0.7352 time: 0.0499s
Epoch: 0088 loss_train: 0.9366 acc_train: 0.8357 loss_val: 1.1174 acc_val:
0.7360 time: 0.0498s
Epoch: 0089 loss_train: 0.9855 acc_train: 0.8000 loss_val: 1.1106 acc_val:
0.7375 time: 0.0499s
Epoch: 0090 loss_train: 0.9706 acc_train: 0.8143 loss_val: 1.1039 acc_val:
0.7403 time: 0.0499s
Epoch: 0091 loss_train: 1.0427 acc_train: 0.6857 loss_val: 1.0975 acc_val:
0.7426 time: 0.0499s
Epoch: 0092 loss_train: 0.9856 acc_train: 0.8214 loss_val: 1.0913 acc_val:
0.7457 time: 0.0499s
Epoch: 0093 loss_train: 1.0248 acc_train: 0.7429 loss_val: 1.0853 acc_val:
0.7484 time: 0.0499s
Epoch: 0094 loss_train: 0.9592 acc_train: 0.7929 loss_val: 1.0794 acc_val:
0.7496 time: 0.0499s
Epoch: 0095 loss_train: 0.9814 acc_train: 0.8071 loss_val: 1.0734 acc_val:
0.7512 time: 0.0498s
Epoch: 0096 loss_train: 0.9951 acc_train: 0.7714 loss_val: 1.0677 acc_val:
0.7523 time: 0.0499s
Epoch: 0097 loss_train: 0.9369 acc_train: 0.7786 loss_val: 1.0622 acc_val:
0.7535 time: 0.0499s
Epoch: 0098 loss_train: 0.9871 acc_train: 0.7429 loss_val: 1.0568 acc_val:
0.7547 time: 0.0499s
Epoch: 0099 loss_train: 0.8497 acc_train: 0.8571 loss_val: 1.0514 acc_val:
0.7547 time: 0.0499s
Epoch: 0100 loss_train: 1.0142 acc_train: 0.7500 loss_val: 1.0460 acc_val:
0.7570 time: 0.0499s
Optimization Finished!
Total time elapsed: 5.0088s
Test set results: loss= 1.0460 accuracy= 0.7570

```

## 5 Question: (Your task)

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).

**ANSWER:** The GAT model shows better performance than the Vanilla GCN, with higher validation and test accuracies. The reason for this discrepancy is that GAT uses self-attention mechanisms to learn the importance of each neighboring node, which allows it to capture more complex patterns and relations in the graph, thus resulting in better performance compared to the simpler Vanilla GCN.

# a4\_dqn

April 10, 2023

## 0.1 Enable rendering OpenAI Gym environments from CoLab

In this assignment, We will use [OpenAI Gym](#) for rendering game environment for our agent to play and learn. It is possible and important to visualize the game your agent is playing, even on Colab. This section imports the necessary package and functions needed to generate a video in Colab. The video processing steps credit to [here](#).

```
[ ]: # You will need to run this block twice to make it effective
!apt-get update > /dev/null 2>&1
!apt-get install cmake > /dev/null 2>&1
!pip install --upgrade setuptools 2>&1
!pip install ez_setup > /dev/null 2>&1
!pip install gym[atari] > /dev/null 2>&1
!pip install box2d-py > /dev/null 2>&1
!pip install gym[Box_2D] > /dev/null 2>&1
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
```

```
Requirement already satisfied: setuptools in /usr/local/lib/python3.9/dist-packages (67.6.1)
```

```
[ ]: !pip install gym==0.15.3 pyvirtualdisplay > /dev/null 2>&1
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
```

Import openAI gym and define the functions used to show the video.

```
[ ]: import gym
from gym.wrappers import Monitor
import glob
import io
import base64
from IPython.display import HTML
from pyvirtualdisplay import Display
from IPython import display as ipythondisplay

display = Display(visible=0, size=(1400, 900))
display.start()

"""
```

```

Utility functions to enable video recording of gym environment
and displaying it.

To enable video, just do "env = wrap_env(env)"

"""

def show_video():
    mp4list = glob.glob('video/*.mp4')
    if len(mp4list) > 0:
        mp4 = mp4list[0]
        video = io.open(mp4, 'r+b').read()
        encoded = base64.b64encode(video)
        ipythondisplay.display(HTML(data='''<video alt="test" autoplay
                                         loop controls style="height: 400px;">
                                         <source src="data:video/mp4;base64,{0}" type="video/mp4" />
                                         </video>''.format(encoded.decode('ascii'))))
    else:
        print("Could not find video")

def wrap_env(env):
    env = Monitor(env, './video', force=True)
    return env

```

Import other packages:

We will use Pytorch for building and learning our DQN network.

```
[ ]: import torch
from torch import nn
import copy
from collections import deque
import random
from tqdm import tqdm
import matplotlib.pyplot as plt

random.seed(42)
```

## 0.2 Run the game with random agent.

```
[ ]: from torch import randint
from time import sleep

env = wrap_env(gym.make('CartPole-v1'))
reward_arr = []
episode_count = 20
for i in tqdm(range(episode_count)):
    obs, done, rew = env.reset(), False,
```

```

env.render()
while not done:
    A = randint(0, env.action_space.n, (1,))
    obs, reward, done, info = env.step(A.item())
    rew += reward
    sleep(0.01)
    reward_arr.append(rew)
print("average reward per episode :", sum(reward_arr) / len(reward_arr))
env.close()
show_video()

```

100% | 20/20 [00:05<00:00, 3.48it/s]  
average reward per episode : 19.4

<IPython.core.display.HTML object>

The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center. The video is short (< 1s) because the pole loses balance immediately.

You can see that a random agent is having trouble balancing the CartPole, just like you. However, a difficult game for human may be very simple to a computer. Let's see how we can use DQN to train a agent.

### 0.3 Experience Replay

The technique of experience replay was first proposed in to resolve temporal correlation in the input data by mixing recent experiences as well past experiences, essentially forcing the input to become independent and identically distributed (i.i.d.). It has been shown that this greatly stabilizes and improves the DQN training procedure.

```
[ ]: import numpy as np

class ExperienceReplay(object):
    def __init__(self, length):
        self.experience_replay = deque(maxlen=length)

    def collect(self, experience):
        self.experience_replay.append(experience)
        return

    # def sample_from_experience(self, sample_size):
    #     if len(self.experience_replay) < sample_size:
    #         sample_size = len(self.experience_replay)
    #     sample = random.sample(self.experience_replay, sample_size)
    #     state = torch.tensor([exp[0] for exp in sample]).float()
    #     action = torch.tensor([exp[1] for exp in sample]).float()
    #     reward = torch.tensor([exp[2] for exp in sample]).float()
```

```

#     next_state = torch.tensor([exp[3] for exp in sample]).float()
#     return state, action, reward, next_state

# to make it faster
def sample_from_experience(self, sample_size):
    if len(self.experience_replay) < sample_size:
        sample_size = len(self.experience_replay)
    sample = random.sample(self.experience_replay, sample_size)
    state = torch.tensor(np.array([exp[0] for exp in sample])).float()
    action = torch.tensor(np.array([exp[1] for exp in sample])).float()
    reward = torch.tensor(np.array([exp[2] for exp in sample])).float()
    next_state = torch.tensor(np.array([exp[3] for exp in sample])).float()
    return state, action, reward, next_state

```

## 0.4 Build our DQN Network

We will use a simple multi-layer neural network to learn the optimal actions. We will use Adam Optimizer and MSE loss for training. **Notice that the loss function and gamma is given to you in the class attribute.**

```
[ ]: class DQN_Network:

    def __init__(self, layer_size_list, lr, seed=1423):
        torch.manual_seed(seed)
        self.policy_net = self.create_network(layer_size_list)
        self.target_net = copy.deepcopy(self.policy_net)

        self.loss_fn = torch.nn.MSELoss() # the loss function
        self.optimizer = torch.optim.Adam(self.policy_net.parameters(), lr=lr)

        self.step = 0
        self.gamma = torch.tensor(0.95).float()
        return

    def create_network(self, layer_size_list):
        assert len(layer_size_list) > 1

        layers = []
        for i in range(len(layer_size_list) - 1):
            linear = nn.Linear(layer_size_list[i], layer_size_list[i + 1])

            if i < len(layer_size_list) - 2:
                activation = nn.Tanh()
            else:
                activation = nn.Identity()
```

```

        layers += (linear, activation)
    return nn.Sequential(*layers)

def load_pretrained_model(self, model_path):
    self.policy_net.load_state_dict(torch.load(model_path))

def save_trained_model(self, model_path="cartpole-dqn.pth"):
    torch.save(self.policy_net.state_dict(), model_path)

```

## 0.5 [Your task]: complete the function that chooses the next action

Choose next action based on  $\epsilon$ -greedy:

$$\text{where } a_{t+1} = \begin{cases} \operatorname{argmax}_a Q(a, s) & \text{with probability : } 1 - \epsilon, \text{exploitation} \\ \text{Uniform}\{a_1, \dots, a_n\} & \text{with probability : } \epsilon, \text{exploration} \end{cases} \quad (1)$$

```
[ ]: 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 torch.rand(1).item() > epsilon:
        # Exploitation: choose the action with the highest Q-value
        action = torch.argmax(Qp).view(1)
    else:
        # Exploration: choose a random action from the available actions
        action = torch.randint(0, action_space_len, (1,))

    return action
```

### 0.5.1 [Your task]: complete the function that train the network for one step

Here, you can find an `train` function that performs a single step of the optimization.

For our training update rule, the loss you are trying to minimize is:

$$\text{loss} = Q(s, a) - (r + \gamma \max_a Q(s', a)) \quad (2)$$

```
[ ]: import torch.nn.functional as F

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
    state_action_values = model.policy_net(state).gather(1, action.long()).
    ↪unsqueeze(1))

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

    # TODO: compute the loss
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values.
    ↪unsqueeze(1))
    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()
```

### 0.5.2 [Your task]: Finish the training loop

In this part, you can play around with `exp_replay_size`, `episode`, `epsilon` and the “episode decay” logic to train your model. If you have done correctly, you will observe that the training time for the latter episodes is longer than the early episodes. This is because your agent is getting better and better at playing the game and thus each episode takes longer

```
[ ]: # Create the model
env = gym.make('CartPole-v0')
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
agent = DQN_Network(layer_size_list=[input_dim, 64, output_dim], lr=1e-3)

# Main training loop
losses_list, reward_list, episode_len_list, epsilon_list = [], [], [], []

# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 5000
memory = ExperienceReplay(exp_replay_size)
episodes = 20000
```

```

epsilon = 1 # epsilon start from 1 and decay gradually.

# initilizae experiance replay
index = 0
for i in range(exp_replay_size):
    obs = env.reset()
    done = False
    while not done:
        A = get_action(agent, obs, env.action_space.n, epsilon=1)
        obs_next, reward, done, _ = env.step(A.item())
        memory.collect([obs, A.item(), reward, obs_next])
        obs = obs_next
        index += 1
        if index > exp_replay_size:
            break

index = 128
for i in tqdm(range(episodes)):
    obs, done, losses, ep_len, rew = env.reset(), False, 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 *= 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% | 20000/20000 [03:16<00:00, 101.75it/s]

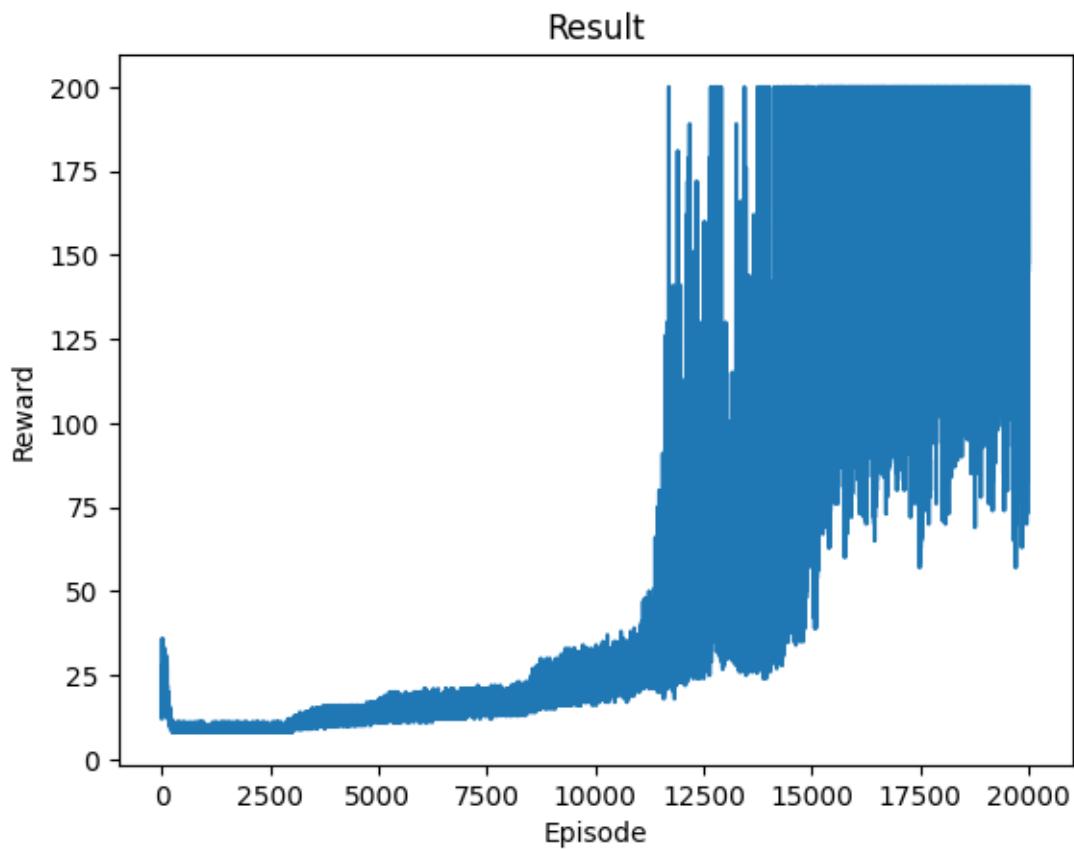
Saving trained model

## 0.6 Last Step: evaluate your trained model! Make sure to include your visualizations (plot+video) in the notebook for your submission!

First we can plot the reward vs. episode. If you have done correctly, you should see the reward can stabilize at 200 in later episodes

```
[ ]: def plot_reward(r):
    plt.figure(2)
    plt.clf()
    plt.title('Result')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.plot(r)

plot_reward(reward_list)
```



Next let check out how well your agent plays the game. If you have done correctly, you should see a relatively longer video (> 3~4s) with a self-balancing pole.

```
[ ]: env = wrap_env(gym.make('CartPole-v1'))

input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
model_validate = DQN_Network(layer_size_list=[input_dim, 64, output_dim], lr=1e-3)
model_validate.load_pretrained_model("cartpole-dqn.pth")

reward_arr = []
for i in tqdm(range(200)):
    obs, done, rew = env.reset(), False, 0
    env.render()
    while not done:
        A = get_action(model_validate, obs, env.action_space.n, epsilon=0)
        obs, reward, done, info = env.step(A.item())
        rew += reward
        # sleep(0.01)

    reward_arr.append(rew)
print("average reward per episode :", sum(reward_arr) / len(reward_arr))
env.close()
show_video()
```

100% | 200/200 [00:20<00:00, 9.83it/s]  
average reward per episode : 195.25

<IPython.core.display.HTML object>

### 0.6.1 Grid Search

```
[ ]: from itertools import product

# Hyperparameters to search
exp_replay_sizes = [5000, 20000]
episodes = [5000, 20000]
epsilon_decay_rates = [0.995, 0.95]
epsilons = [1, 0.5]

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

for exp_replay_size, episode, epsilon_decay_rate, epsilon in product(exp_replay_sizes, episodes, epsilon_decay_rates, epsilons):
```

```

print("-"*20)
print(exp_replay_size, episode, epsilon_decay_rate, epsilon)

memory = ExperienceReplay(exp_replay_size)
# Create the model
env = gym.make('CartPole-v0')
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
agent = DQN_Network(layer_size_list=[input_dim, 64, output_dim], lr=1e-3)

# Main training loop
losses_list, reward_list, episode_len_list, epsilon_list = [], [], [], []

# initialize experiance replay
index = 0
for i in range(exp_replay_size):
    obs = env.reset()
    done = False
    while not done:
        A = get_action(agent, obs, env.action_space.n, epsilon=1)
        obs_next, reward, done, _ = env.step(A.item())
        memory.collect([obs, A.item(), reward, obs_next])
        obs = obs_next
        index += 1
    if index > exp_replay_size:
        break

index = 128
for i in tqdm(range(episode)):
    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_decay_rate

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")
plot_reward(reward_list)

env = wrap_env(gym.make('CartPole-v1'))

input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
model_validate = DQN_Network(layer_size_list=[input_dim, 64, output_dim], lr=1e-3)
model_validate.load_pretrained_model("cartpole-dqn.pth")

reward_arr = []
for i in tqdm(range(200)):
    obs, done, rew = env.reset(), False, 0
    env.render()
    while not done:
        A = get_action(model_validate, obs, env.action_space.n, epsilon=0)
        obs, reward, done, info = env.step(A.item())
        rew += reward
        # sleep(0.01)

    reward_arr.append(rew)
print("average reward per episode :", sum(reward_arr) / len(reward_arr))
env.close()

```

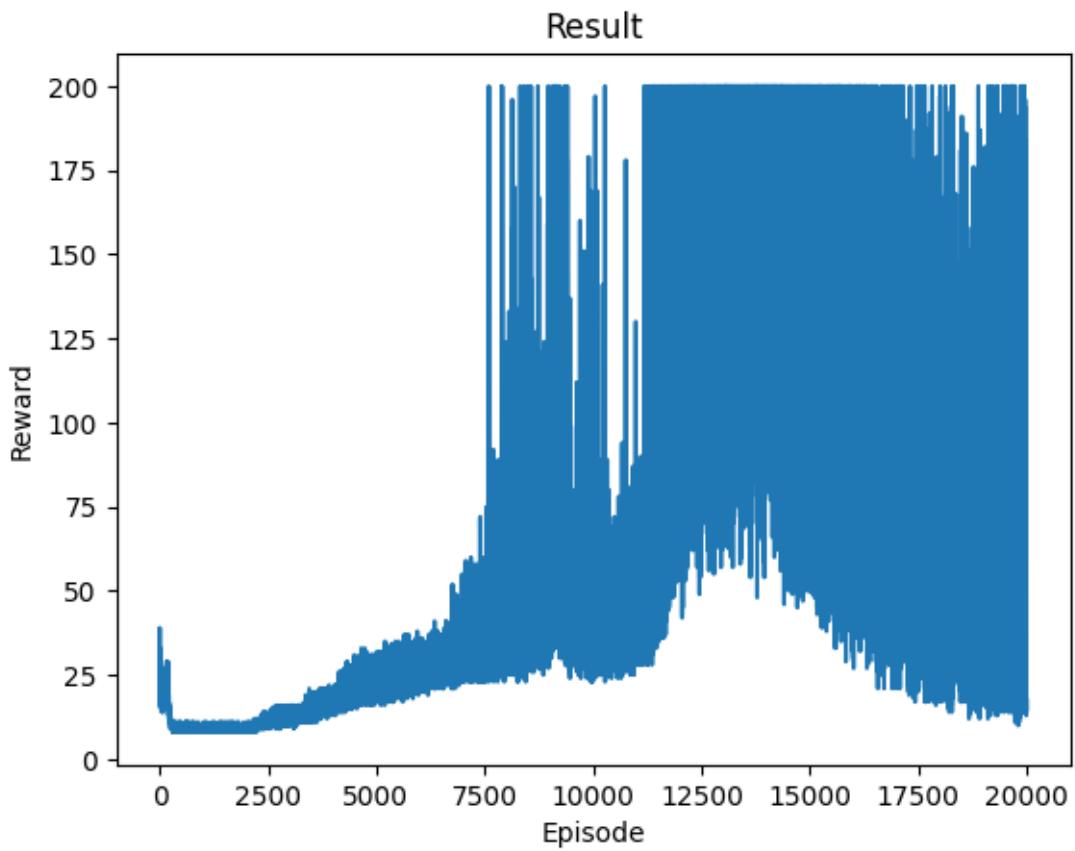
```

-----
5000 5000 0.995 1
100%|      | 5000/5000 [00:10<00:00, 456.19it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 84.37it/s]
average reward per episode : 17.52
-----
5000 5000 0.995 0.5
100%|      | 5000/5000 [00:10<00:00, 462.99it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 82.90it/s]
```

```
average reward per episode : 15.81
-----
5000 5000 0.95 1
100%|      | 5000/5000 [00:10<00:00, 481.69it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 85.92it/s]
average reward per episode : 15.755
-----
5000 5000 0.95 0.5
100%|      | 5000/5000 [00:10<00:00, 484.90it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 89.99it/s]
average reward per episode : 14.38
-----
5000 20000 0.995 1
100%|      | 20000/20000 [03:19<00:00, 100.10it/s]
Saving trained model
100%|      | 200/200 [00:03<00:00, 52.68it/s]
average reward per episode : 21.705
-----
5000 20000 0.995 0.5
100%|      | 20000/20000 [01:07<00:00, 295.72it/s]
Saving trained model
100%|      | 200/200 [00:06<00:00, 31.93it/s]
average reward per episode : 48.61
-----
5000 20000 0.95 1
100%|      | 20000/20000 [03:56<00:00, 84.63it/s]
Saving trained model
100%|      | 200/200 [00:11<00:00, 18.03it/s]
average reward per episode : 129.6
-----
5000 20000 0.95 0.5
100%|      | 20000/20000 [04:49<00:00, 69.03it/s]
Saving trained model
100%|      | 200/200 [00:09<00:00, 21.16it/s]
```

```
average reward per episode : 97.05
-----
20000 5000 0.995 1
100%|      | 5000/5000 [00:12<00:00, 398.99it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 69.95it/s]
average reward per episode : 24.94
-----
20000 5000 0.995 0.5
100%|      | 5000/5000 [00:12<00:00, 412.27it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 68.12it/s]
average reward per episode : 23.94
-----
20000 5000 0.95 1
100%|      | 5000/5000 [00:11<00:00, 444.37it/s]
Saving trained model
100%|      | 200/200 [00:03<00:00, 57.82it/s]
average reward per episode : 23.315
-----
20000 5000 0.95 0.5
100%|      | 5000/5000 [00:11<00:00, 423.97it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 73.61it/s]
average reward per episode : 22.92
-----
20000 20000 0.995 1
100%|      | 20000/20000 [02:54<00:00, 114.82it/s]
Saving trained model
100%|      | 200/200 [00:04<00:00, 46.14it/s]
average reward per episode : 40.97
-----
20000 20000 0.995 0.5
100%|      | 20000/20000 [02:17<00:00, 145.47it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 98.24it/s]
```

```
average reward per episode : 12.81
-----
20000 20000 0.95 1
100%|      | 20000/20000 [02:22<00:00, 140.45it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 82.82it/s]
average reward per episode : 15.535
-----
20000 20000 0.95 0.5
100%|      | 20000/20000 [03:11<00:00, 104.40it/s]
Saving trained model
100%|      | 200/200 [00:02<00:00, 81.57it/s]
average reward per episode : 16.595
```



[ ]: