

## Assignment 4

- (v1.1) Specify the deliverables for the programming questions and provide the raw latex dump for the written questions

**Version:** 1.1

**Version Release Date:** 2023-03-21

**Deadline:** Tuesday, April 11, at 11:59pm.

**Submission:** You must submit two files through MarkUs<sup>1</sup>: (1) a PDF file containing your writeup, titled `a4-writeup.pdf`, and (2) your code file `gnn.ipynb`, `dqn.ipynb`. There will be sections in the notebook for you to write your responses. Your writeup must be typed. Make sure that the relevant outputs (e.g. `print_gradients()` outputs, plots, etc.) are included and clearly visible. See the syllabus on the course website<sup>2</sup> for detailed policies. You may ask questions about the assignment on Piazza<sup>3</sup>. *Note that 10% of the assignment mark (worth 2 pts) may be removed for lack of neatness.*

You may notice that some questions are worth 0 pt, which means we will not mark them in this Assignment. Feel free to skip them if you are busy. However, you are expected to see some of them in the midterm. So, we won't release the solution for those questions.

The teaching assistants for this assignment are Mustafa Ammous, Masoud Moghani, Haotian Cui, and Shihao Ma. Send your email with subject “[CSC413] A4 ...” to `csc413-2023-01-tas@cs.toronto.edu` or post on Piazza with the tag `a4`.

---

<sup>1</sup><https://markus.teach.cs.toronto.edu/2023-01>

<sup>2</sup><https://uoft-csc413.github.io/2023/assets/misc/syllabus.pdf>

<sup>3</sup><https://piazza.com/class/lcp8mp3f9d171p>

**Important Instructions**

**Read the following before attempting the assignment.**

## Overview and Expectations

You will be completing this assignment with the aid of large language models (LLMs) such as ChatGPT, text-davinci-003, or code-davinci-002. To alleviate the unnecessary steps related to generating results and screenshotting, we have provided the GPT-generated solution with minimum prompting effort in **ChatGPT**: clauses. The goal is to help you *(i)* develop a solid understanding of the course materials, and *(ii)* gain some insight in problem-solving with LLMs. Think of this as analogous to *(i)* understanding the rules of addition and multiplication, and *(ii)* learning how to use a calculator. Note that LLMs may not be a reliable “calculator” (yet) — as you will see, GPT-like models can generate incorrect and contradicting answers. It is, therefore important that you have a good grasp of the lecture materials, so that you can evaluate the correctness of the model output, and also prompt the model toward the correct solution.

**Prompt engineering.** In this assignment, we ask that you try to *(i)* solve the problems yourself, and *(ii)* use LLMs to solve a selected subset of them. You will “guide” the LLMs toward desired outcomes by typing *text prompts* into the models. There are a number of different ways to prompt an LLM, including direct copy-pasting L<sup>A</sup>T<sub>E</sub>X strings of a written question, copying function docstrings, or interactively editing the previously generated results. Prompting offers a natural and intuitive interface for humans to interact with and use LLMs. However, LLM-generated solutions depend significantly on the quality of the prompt used to steer the model, and most effective prompts come from a deep understanding of the task. You can decide how much time you want to spend as a university student vs. a prompt engineer, but we’d say it’s probably not a good idea to use more than 25% of your time on prompting LLMs. See Best Practices below for the basics of prompt engineering.

**What are LLMs good for?** We have divided the assignment problems into the following categories, based on our judgment of how difficult it is to obtain the correct answer using LLMs.

- **[Type 1]** LLMs can produce almost correct answers from rather straightforward prompts, e.g., minor modification of the problem statement.
- **[Type 2]** LLMs can produce partially correct and useful answers, but you may have to use a more sophisticated prompt (e.g., break down the problem into smaller pieces, then ask a sequence of questions), and also generate multiple times and pick the most reasonable output.
- **[Type 3]** LLMs usually do not give the correct answer unless you try hard. This may include problems with involved mathematical reasoning or numerical computation (many GPT models do not have a built-in calculator).
- **[Type 4]** LLMs are not suitable for the problem (e.g., graph/figure-related questions).

## Written Assignment

### What you have to submit for this part

See the top of this handout for submission directions. Here are the requirements.

- The zero point questions (in black below) will not be graded, but you are more than welcome to include your answers for these as well in the submission.
- For (nonzero-point) questions labeled [Type 1] [Type 2] you need to submit your own solution. Your own solution can be a copy-paste of the LLM output (if you verify that it is correct), but make sure you **cite the model properly**.
- For (nonzero-point) questions in [Type 3] [Type 4] you only need to submit your own written solution, but we encourage you to experiment with LLMs on some of them.

For reference, here is everything you need to hand in for the first half of the PDF report a4-writeup.pdf.

- **Problem 1:** 1.1.2[Type 1] , 1.2.1[Type 1] , 1.3.1[Type 1] , 1.3.2[Type 2]
- **Problem 2:** 2.1.1[Type 3] , 2.1.2[Type 3] , 2.1.2[Type 1] , 2.2.1[Type 3]

### Useful prompts

You could start by naively copy-pasting the question and the context as the prompt, and try to improve the generated answers by trial and error. Raw L<sup>A</sup>T<sub>E</sub>X dumps are made available for the written questions to facilitate this process.

- [https://uoft-csc413.github.io/2023/assets/assignments/a4\\_raw\\_latex\\_dump.tex](https://uoft-csc413.github.io/2023/assets/assignments/a4_raw_latex_dump.tex)

## 1 RNNs and Self Attention

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

### 1.1 Warmup: A Single Neuron RNN

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

#### 1.1.1 Effect of Activation - ReLU [0pt]

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

**ChatGPT:**

Assuming the ReLU activation function is used, the output of the hidden layer can be expressed as  $h(x) = \max(0, x)$ , where  $x$  is the input to the layer. Therefore, the output of the entire network can be expressed as  $f(x) = h(h(\dots(h(x))))$ , where there are  $n$  consecutive applications of  $h$ .

Let's first consider the case where  $x \geq 0$ . In this case, each application of  $h$  has a gradient of 1, so the overall gradient is simply the product of  $n$  1's, which is 1. Therefore,  $\left| \frac{\partial f(x)}{\partial x} \right| = 1$  when  $x \geq 0$ .

Now let's consider the case where  $x < 0$ . In this case, the first application of  $h$  will output 0, and all subsequent applications of  $h$  will also output 0. Therefore, the output of the entire network will be  $f(x) = 0$ , and the gradient will be 0 as well. Therefore,  $\left| \frac{\partial f(x)}{\partial x} \right| = 0$  when  $x < 0$ .

Combining these two cases, we can conclude that  $0 \leq \left| \frac{\partial f(x)}{\partial x} \right| \leq 1$  for any input  $x$ . Therefore, the gradients do not necessarily have to vanish or explode as they are backpropagated, and can be bounded between 0 and 1.

#### 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.

**ChatGPT:**

Let  $x$  be the input to the network, and let  $f : \mathbb{R}^1 \rightarrow \mathbb{R}^1$  be the function the network is computing. Let  $h_i(x)$  be the output of the  $i$ -th hidden layer with weight  $w_i$ , and let  $f(x)$  be the output of the entire network. Then we can write:

$$h_i(x) = w_i \max(0, h_{i-1}(x))$$

where  $h_0(x) = x$  and  $h_n(x)$  is the output of the entire network. Using the chain rule, we can write:

$$\frac{\partial f(x)}{\partial x} = \frac{\partial h_n(x)}{\partial x} = \frac{\partial h_n(x)}{\partial h_{n-1}(x)} \frac{\partial h_{n-1}(x)}{\partial h_{n-2}(x)} \dots \frac{\partial h_2(x)}{\partial h_1(x)} \frac{\partial h_1(x)}{\partial x}$$

Using the expression for  $h_i(x)$ , we can compute the derivatives as follows:

$$\frac{\partial h_i(x)}{\partial h_{i-1}(x)} = w_i \frac{\partial}{\partial h_{i-1}(x)} \max(0, h_{i-1}(x)) = w_i [\max(0, h_{i-1}(x))']$$

where  $[\max(0, h_{i-1}(x))']$  is the derivative of the ReLU activation function evaluated at  $h_{i-1}(x)$ . If  $h_{i-1}(x) > 0$ , then  $[\max(0, h_{i-1}(x))]' = 1$ , and if  $h_{i-1}(x) \leq 0$ , then  $[\max(0, h_{i-1}(x))]' = 0$ . Therefore, we have:

$$\left| \frac{\partial h_i(x)}{\partial h_{i-1}(x)} \right| = w_i [\max(0, h_{i-1}(x))]' = \begin{cases} w_i & \text{if } h_{i-1}(x) > 0 \\ 0 & \text{if } h_{i-1}(x) \leq 0 \end{cases}$$

Using the above expression, we can compute:

$$\left| \frac{\partial f(x)}{\partial x} \right| = \left| \frac{\partial h_n(x)}{\partial x} \right| = \left| \prod_{i=1}^n \frac{\partial h_i(x)}{\partial h_{i-1}(x)} \right| = \left| \prod_{h_{i-1}(x) > 0} w_i \right|$$

The above expression shows that the magnitude of the gradients can be bounded by the product of the magnitudes of the weights of the positive activations. This bound can still grow exponentially with the number of hidden layers and the magnitudes of the weights, which can lead to the vanishing or exploding gradients problem.

## 1.2 Matrices and RNN

We will now analyze the recurrent weight matrices under Singular Value Decomposition. SVD is one of the most important results in all of linear algebra. It says that any real matrix  $M \in \mathbb{R}^{m \times n}$  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  $x_{t+1} = \text{sigmoid}(Wx_t)$ . You can view this architecture as a deep fully connected network that uses the same weight matrix at each layer. Suppose the largest singular value of the weight matrix is  $\sigma_{\max}(W) = \frac{1}{4}$ . Show that the largest singular value of the input-output Jacobian has the following bound:

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

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

**ChatGPT:**

To begin, let's compute the Jacobian of the RNN-like architecture:

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

Using the chain rule, we can write the input-output Jacobian as the product of the layerwise Jacobians:

$$\frac{\partial x_n}{\partial x_1} = \prod_{t=1}^{n-1} \frac{\partial x_{t+1}}{\partial x_t} = \prod_{t=1}^{n-1} \text{sigmoid}'(Wx_t)W$$

To bound the largest singular value of this Jacobian, we can use the hint provided and the fact that the largest singular value of a matrix is equal to its spectral norm, which is defined as the maximum absolute value of its eigenvalues.

Using the spectral norm, we can write:

$$\sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) = \left\| \prod_{t=1}^{n-1} \text{sigmoid}'(Wx_t)W \right\|_2$$

We can use the fact that the spectral norm is sub-multiplicative, meaning that  $\|AB\|_2 \leq \|A\|_2 \|B\|_2$ , to bound the spectral norm of the product:

$$\sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \prod_{t=1}^{n-1} \|\text{sigmoid}'(Wx_t)W\|_2$$

We can further bound the 2-norm of  $\text{sigmoid}'(Wx_t)W$  as follows:

$$\|\text{sigmoid}'(Wx_t)W\|_2 = \sigma_{\max}(\text{sigmoid}'(Wx_t)W) \leq \sigma_{\max}(\text{sigmoid}'(Wx_t))\sigma_{\max}(W)$$

where we used the hint provided to obtain the inequality.

Recall that the sigmoid function has a maximum derivative value of  $\frac{1}{4}$ , which occurs at  $x = 0$ . Therefore, we can write:

$$\sigma_{\max}\left(\frac{\partial x_n}{\partial x_1}\right) \leq \prod_{t=1}^{n-1} \frac{1}{4} \sigma_{\max}(W) = \left(\frac{1}{4}\right)^{n-1} \sigma_{\max}(W)^{n-1}$$

Substituting the value of  $\sigma_{\max}(W) = \frac{1}{4}$  given in the problem, we obtain 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 Self-Attention

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

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

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

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

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

$$\alpha_i = \frac{\sum_{j=1}^n \text{sim}(Q_i, K_j) V_j}{\sum_{j=1}^n \text{sim}(Q_i, K_j)} \quad (1.1)$$

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

*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  $j$ ).*

**ChatGPT:**

To show that attention can be calculated with linear complexity using kernel functions, we will substitute a kernel function for the similarity function  $\text{sim}()$  in Equation 1.1 and group the terms based on their subscript.

Let  $k(x, y)$  be a kernel function, then we can define the similarity function  $\text{sim}(Q_i, K_j)$  as:

$$\text{sim}(Q_i, K_j) = k(Q_i, K_j)$$

Substituting this into Equation 1.1, we get:

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

Now, let's expand  $k(Q_i, K_j)$  using the kernel trick:

$$k(Q_i, K_j) = \phi(Q_i)^T \phi(K_j)$$

where  $\phi()$  is a feature mapping that maps the input to a high-dimensional space.

Substituting this in the previous equation, we get:

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

Next, we can group the terms based on their subscripts as follows:

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

Now, we can precompute  $\phi(K_j) V_j$  and  $\phi(K_j)$  for all  $j$ , and store them in matrices  $M$  and  $N$  respectively, where  $M_j = \phi(K_j) V_j$  and  $N_j = \phi(K_j)$ . Then, we can rewrite the equation as:

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

This can be computed for all  $i$  with a single matrix multiplication, which has a complexity of  $\mathcal{O}(n)$ , since  $M$  and  $N$  have dimensions  $n \times d$ , where  $d$  is the dimensionality of the feature space.

Therefore, we have shown that by applying kernel functions, attention can be calculated with linear complexity.

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

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

**ChatGPT:**

If the rank of  $P$  is  $k$ , then we can express  $P$  as a product of two matrices  $U \in \mathbb{R}^{n \times k}$  and  $V^\top \in \mathbb{R}^{k \times n}$ , where  $U$  and  $V$  are orthonormal matrices, i.e.,  $U^\top U = I_k$  and  $VV^\top = I_n$ . Therefore, we have:

$$P = UV^\top$$

Using this factorization of  $P$ , we can rewrite the self-attention function as:

$$\begin{aligned} \text{Self-Attention}(Q, K, V) &= \left(\frac{QK^\top}{\sqrt{d_k}}\right) V \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)^\top = Q(UV^\top)K^\top V(UV^\top)^\top \\ &= QU(V^\top K)^\top VU^\top VU^\top = QU(V^\top K)^\top (VV)^\top \end{aligned}$$

Here, we used the property that  $U^\top V = 0$  because  $U$  and  $V$  have different orthonormal bases.

Note that  $V^\top K$  has a shape of  $k \times d_k$ , and we can compute it once since we already have access to the SVD of  $P$ .

Therefore, given the SVD of  $P$ , we can compute the self-attention with  $\mathcal{O}(nkd)$  time complexity, which is linear in  $n$  and  $d$ , and proportional to the rank  $k$  of the context mapping matrix  $P$ . This is more efficient than the  $\mathcal{O}(n^2d)$  time complexity required for the standard self-attention calculation.

### 1.3.3 Linear Attention by Projecting [0pt]

Suppose we ignore the Softmax and scaling and let  $P = QK^\top \in \mathbb{R}^{n \times n}$ . Assume  $P$  is rank  $k$ . Show that there exist two linear projection matrices  $C, D \in \mathbb{R}^{k \times n}$  such that  $PV = Q(CK)^\top DV$  and the right hand side can be computed in  $\mathcal{O}(nkd)$  time. *Hint: Consider using SVD in your proof.*

**ChatGPT:**

Since  $P = QK^\top$ , we can perform SVD on  $K$  as  $K = U\Sigma V^\top$ , where  $U \in \mathbb{R}^{n \times n}$  and  $V \in \mathbb{R}^{d_k \times d_k}$  are orthogonal matrices and  $\Sigma \in \mathbb{R}^{d_k \times d_k}$  is a diagonal matrix with singular values  $\sigma_1, \sigma_2, \dots, \sigma_{d_k}$ .

Then, we have  $QK^\top = QU\Sigma V^\top$ . Since  $P = QK^\top$  has rank  $k$ , we know that  $\Sigma$  has at most  $k$  non-zero singular values. Without loss of generality, let the first  $k$  singular values be non-zero, and let  $\Sigma_k$  be the corresponding diagonal matrix of singular values.

Using the first  $k$  columns of  $U$  and  $V$  (denoted  $U_k$  and  $V_k$ , respectively), we can approximate  $K$  as  $K \approx U_k \Sigma_k V_k^\top$ . Substituting this approximation into the expression for  $P$ , we have  $P \approx QU_k \Sigma_k V_k^\top$ , which we can rewrite as  $P = Q(CK)^\top DV$  where  $C = V_k^\top$ ,  $D = (U_k \Sigma_k)^\top$  and  $V = I_{d_k}$ .

Now, to compute  $PV$  we can first compute  $CK$  and  $DV$  separately, each in  $\mathcal{O}(nkd)$  time. Then, we can compute  $Q(CK)^\top DV$  in  $\mathcal{O}(nkd)$  time. Therefore, we can compute  $PV$  in  $\mathcal{O}(nkd)$  time as required.



## 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:

$$(T^\pi V)(s) \triangleq r^\pi(s) + \gamma \int \mathcal{P}(s'|s, a) \pi(a|s) V(s') \quad (2.1)$$

$$(T^\pi Q)(s, a) \triangleq r(s, a) + \gamma \int \mathcal{P}(s'|s, a) \pi(a'|s') Q(s', a') \quad (2.2)$$

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 an unique and optimal solution, which is what allow us to show RL algorithms such as Q-Learning converges (under certain additional assumptions, but we won't go over them here). In this section, we will show that the Bellman operator indeed have these two properties.

#### 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

$$T^\pi V_1 \leq T^\pi V_2 \quad (2.3)$$

#### ChatGPT:

To show the monotonicity property of the Bellman operator  $T^\pi$  on  $V$ , we need to prove that if  $V_1, V_2 \in \mathcal{B}(\mathcal{S})$  with  $V_1(s) \leq V_2(s)$  for all  $s \in \mathcal{S}$ , then  $(T^\pi V_1)(s) \leq (T^\pi V_2)(s)$  for all  $s \in \mathcal{S}$ . Expanding the definition of  $T^\pi$  for  $V$ , we have:

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

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

$$\begin{aligned} \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V_1(s') &\leq \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V_2(s') \\ \Rightarrow r^\pi(s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V_1(s') &\leq r^\pi(s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) V_2(s') \\ &\Rightarrow (T^\pi V_1)(s) \leq (T^\pi V_2)(s) \end{aligned}$$

Therefore, we have shown that the Bellman operator  $T^\pi$  on  $V$  has the monotonicity property.

## 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

$$\|T^\pi(Q_1) - T^\pi(Q_2)\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty \quad (2.4)$$

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

$$\|Q\|_\infty = \sup_{(s,a) \in \mathcal{S} \times \mathcal{A}} |Q(s,a)| \quad (2.5)$$

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

$$\begin{aligned} \left| \int P(x) f(x) \right| &\leq \int |P(x) f(x)| = \int |P(x)| \cdot |f(x)| \\ &\leq \int P(x) \cdot \sup_{x \in \mathcal{X}} |f(x)| \\ &= \sup_{x \in \mathcal{X}} |f(x)| \int P(x) = \|f\|_\infty \end{aligned} \quad (2.6)$$

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

**ChatGPT:**

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

$$\|T^\pi Q_1 - T^\pi Q_2\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty \quad (2.7)$$

Let  $s$  and  $a$  be arbitrary elements in  $\mathcal{S}$  and  $\mathcal{A}$  respectively. Then we have:

$$\begin{aligned} |(T^\pi Q_1)(s,a) - (T^\pi Q_2)(s,a)| &= \left| (r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' | s, a) \pi(a' | s') Q_1(s', a')) - (r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' | s, a) \pi(a' | s') Q_2(s', a')) \right| \end{aligned} \quad (2.8)$$

$$= \gamma \left| \sum_{s'} \mathcal{P}(s' | s, a) \pi(a' | s') (Q_1(s', a') - Q_2(s', a')) \right| \quad (2.9)$$

$$\leq \gamma \sum_{s'} \mathcal{P}(s' | s, a) \pi(a' | s') |Q_1(s', a') - Q_2(s', a')| \quad (2.10)$$

$$\leq \gamma \|Q_1 - Q_2\|_\infty \quad (2.11)$$

where we used the fact that  $|\sum_x a_x| \leq \sum_x |a_x|$  in the third line and the supremum norm definition in the last line. Taking the supremum over all  $(s,a) \in \mathcal{S} \times \mathcal{A}$  yields:

$$\|T^\pi Q_1 - T^\pi Q_2\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty \quad (2.12)$$

as required. Therefore, the Bellman operator is a  $\gamma$ -contraction mapping with the supremum norm.

### 2.1.3

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

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  $v_*(s)$ .

#### ChatGPT:

a)  $v_*(s)$  is the optimal state-value function, which is defined as the maximum expected return achievable by any policy  $\pi$ , starting from state  $s$ , i.e.,  $v_*(s) = \max_{\pi} \mathbb{E}_{\pi}[G_t | s_t = s]$ . We can express  $v_*(s)$  in terms of  $q_*(s, a)$  as:  $v_*(s) = \max_a q_*(s, a)$ , where  $q_*(s, a)$  is the optimal action-value function, which is defined as the maximum expected return achievable by any policy  $\pi$ , starting from state  $s$ , taking action  $a$ , and then following the optimal policy thereafter, i.e.,  $q_*(s, a) = \max_{\pi} \mathbb{E}_{\pi}[G_t | s_t = s, a_t = a]$ .

b)  $q_*(s, a)$  is the optimal action-value function, which is defined as the maximum expected return achievable by any policy  $\pi$ , starting from state  $s$ , taking action  $a$ , and then following the optimal policy thereafter, i.e.,  $q_*(s, a) = \max_{\pi} \mathbb{E}_{\pi}[G_t | s_t = s, a_t = a]$ . We can express  $q_*(s, a)$  in terms of  $v_*(s)$  as:  $q_*(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) v_*(s')$ , where  $r(s, a)$  is the reward function for taking action  $a$  in state  $s$ ,  $\gamma$  is the discount factor, and  $\mathcal{P}(s'|s, a)$  is the probability of transitioning to state  $s'$  when taking action  $a$  in state  $s$ .

c)  $a_*$  is the optimal action to take in state  $s$ , which maximizes the action-value function, i.e.,  $a_* = \arg \max_a q_*(s, a)$ .

d)  $a_*$  in terms of  $v_*(s)$  is:  $a_* = \arg \max_a [r(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) v_*(s')]$ . This means that we can find the optimal action to take in state  $s$  by computing the action-value function for each action and selecting the action with the highest value.

## 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:

$$\mu = \sigma\left(\sum_{d=1}^D \theta_d x_d\right), \quad (2.13)$$

$$p(a|\theta) = \text{Bernoulli}(\mu) = \begin{cases} \mu & a = 1 \\ 1 - \mu & a = 0 \end{cases}, \quad (2.14)$$

$$\tilde{a} \sim p(a|\theta), \quad (2.15)$$

$$f(\tilde{a}) = \begin{cases} 1 & \tilde{a} = 1 \\ 0 & \tilde{a} = 0 \end{cases}, \quad (2.16)$$

$$J(\theta) = \mathbb{E}_{\tilde{a} \sim p(a|\theta)}[f(\tilde{a})]. \quad (2.17)$$

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

Recall from above that the expression for REINFORCE estimator is:

$$\nabla_{\theta} J[\theta] = \mathbb{E}_{\tilde{a} \sim p(a|\theta)} \left[ f(\tilde{a}) \frac{\partial}{\partial \theta} \log p(a = \tilde{a}|\theta) \right] \quad (2.18)$$

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

$$g[\theta, \tilde{a}] = f(\tilde{a}) \frac{\partial}{\partial \theta} \log p(a = \tilde{a}|\theta), \quad \tilde{a} \sim p(a|\theta) \quad (2.19)$$

For this question, derive a closed form for the  $g[\theta, \tilde{a}]$  as a deterministic function of  $\tilde{a}$ ,  $\mu$ ,  $\theta$ , and  $x_d$ .

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

#### ChatGPT:

We can start by computing the partial derivative of the log-likelihood of the Bernoulli distribution:

$$\frac{\partial}{\partial \theta} \log p(a = \tilde{a}|\theta) = \frac{\partial}{\partial \theta} [\tilde{a} \log \mu + (1 - \tilde{a}) \log(1 - \mu)] \quad (2.20)$$

$$= \frac{\tilde{a}}{\mu} \frac{\partial}{\partial \theta} \mu - \frac{1 - \tilde{a}}{1 - \mu} \frac{\partial}{\partial \theta} \mu \quad (2.21)$$

$$= \frac{\tilde{a} - \mu}{\mu(1 - \mu)} \frac{\partial}{\partial \theta} \mu. \quad (2.22)$$

Next, we can plug in the expressions for  $\mu$  and  $\frac{\partial}{\partial \theta} \mu$ :

$$g[\theta, \tilde{a}] = f(\tilde{a}) \frac{\partial}{\partial \theta} \log p(a = \tilde{a}|\theta) \quad (2.23)$$

$$= f(\tilde{a}) \frac{\tilde{a} - \mu}{\mu(1 - \mu)} \frac{\partial}{\partial \theta} \mu \quad (2.24)$$

$$= f(\tilde{a}) \frac{\tilde{a} - \sigma(\sum^D d = 1\theta_d x_d)}{\sigma(\sum^D d = 1\theta_d x_d)(1 - \sigma(\sum^D d = 1\theta_d x_d))} \sigma(\sum^D d = 1\theta_d x_d)(1 - \sigma(\sum^D d = 1\theta_d x_d)) x_d \quad (2.25)$$

$$= f(\tilde{a})(\tilde{a} - \sigma(\sum^D d = 1\theta_d x_d)) x_d. \quad (2.26)$$

Therefore, we obtain the closed form for  $g[\theta, \tilde{a}]$  as  $f(\tilde{a})(\tilde{a} - \sigma(\sum_{d=1}^D \theta_d x_d))x_d$ .

### 2.2.2 Variance of REINFORCE estimator [0pt]

We will derive the variance of the REINFORCE estimator above. Since the gradient is  $D$ -dimensional, the covariance of the gradients will be  $D \times D$  matrix. In this question, we will only consider the variance with respect to the first parameter, i.e.  $\text{Var}[\hat{g}[\theta, \tilde{a}]_1]$  which scalar value corresponding to the first element in the diagonal of the covariance matrix. Derive the variance of the gradient estimator as a function of the first parameter vector:  $\text{Var}[\hat{g}[\theta, \tilde{a}]_1]$ , as a function of  $\mu$ ,  $\theta$ , and  $x_d$ .

*Hint: The second moment of a Bernoulli random variable is  $\mu(1 - \mu)$ .*

### 2.2.3 Convergence and variance of REINFORCE estimator [0 pt]

Comment on the variance in Part 2.2.2. When do we expect learning to converge slowly in terms of the output of the logistic regression model,  $\mu$ ?

## Programming Assignment

### What you have to submit for this part

For reference, here is everything you need to hand in:

- This is the second half of your PDF report `a4-writeup.pdf`. Please include the solutions to the following problems. You may choose to export `gnn.ipynb`, `dqn.ipynb` as a PDF and attach it to the first half of `a4-writeup.pdf`.
  - **Question 3:** 3.1[Type 1] , 3.2[Type 2] , 3.3[Type 4] , 3.4[Type 2] , 3.5[Type 4] , 3.6[Type 4]
  - **Question 4:** 4.1[Type 1] , 4.2[Type 2] , 4.3[Type 4] .
- Your code file `gnn.ipynb`, `dqn.ipynb`

## Introduction

In this assignment, you'll get hands-on experience coding and training GCN (Graph Convolution Network) and DQN (Deep Q-learning Network), one of Reinforcement Learning methods. This assignment is divided into two parts: in the first part, you will learn how to implement the vanilla version of GCN and GAT. In the second part, you will implement and train a DQN agent to learn how to play the CartPole balancing game. It will be fun to see your model performs much better than you on the simple game :).

## Setting Up

We recommend that you use **Colab**(<https://colab.research.google.com/>) for the assignment. To setup the Colab environment, just open the notebooks for each part of the assignment and **make a copy** in your own Google Drive account.

## Deliverables

Each section is followed by a checklist of deliverables to add in the assignment writeup. To also give a better sense of our expectations for the answers to the conceptual questions, we've put maximum sentence limits. You will not be graded for any additional sentences.

### 3 Graph Convolution Networks[5pt]

For this part of the assignment, you will implement the vanilla version of Graph Convolution Networks (GCN) Kipf and Welling [2016] and Graph Attention Networks (GAT) Velicković et al. [2018].

#### 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:  $X \in \mathcal{R}^{|V| \times F}$ )
2. some information about the graph structure, typically the adjacency matrix  $A$

Each convolutional layer can be written as  $H^{(l+1)} = f(H^{(l)}, A)$ , for some function  $f()$ . The  $f()$  we are using for this assignment is in the form of  $f(H^{(l)}, A) = \sigma(\hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2} H^{(l)} W^{(l)})$ , where  $\hat{A} = A + Identity$  and  $\hat{D}$  is diagonal node degree matrix ( $\hat{D}^{-1} \hat{A}$  normalizes  $\hat{A}$  such that all rows sum to one). Let  $\tilde{A} = \hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2}$ . The GCN we will implement takes two convolution layers,  $Z = f(X, A) = softmax(\tilde{A} \cdot Dropout(ReLU(\tilde{A} X W^{(0)})) \cdot W^{(1)})$

#### Basics of GAT:

Graph Attention Network (GAT) is a novel convolution-style neural network. It operates on graph-structured data and leverages masked self-attentional layers. In this assignment, we will implement the graph attention layer.

#### Dataset:

The dataset we used for this assignment is Cora Sen et al. [2008]. Cora is one of standard citation network benchmark dataset (just like MNIST dataset for computer vision tasks). It that consists of 2708 scientific publications and 5429 links. Each publication is classified into one of 7 classes. Each publication is described by a word vector (length 1433) that indicates the absence/presence of the corresponding word. This is used as the features of each node for our experiment. The task is to perform node classification (predict which class each node belongs to).

#### Experiments:

Open [GNN notebook link] on Colab and answer the following questions.

1. [1pt] **Implementation of Graph Convolution Layer**  
Complete the code for GraphConvolution() Class
2. [1pt] **Implementation of Graph Convolution Network**  
Complete the code for GCN() Class
3. [0.5pt] **Train your Graph Convolution Network**

After implementing the required classes, now you can train your GCN. You can play with the hyperparameters in args.

4. [1.5pt] **Implementation of Graph Attention Layer**

Complete the code for GraphAttentionLayer() Class

5. [0.5pt] **Train your Graph Attention Network**

After implementing the required classes, now you can train your GAT. You can play with the hyperparameters in args.

6. [0.5pt] **Compare your models**

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

## Deliverables

Create a section in your report called **Graph Convolution Networks**. Add the following:

- Screenshots of your GraphConvolution, GCN implementations. Highlight the lines you've added.
- Screenshots of your GCN training output, you can just screenshot the last 10 epochs with test set results.
- Screenshots of your GraphAttentionLayer implementations. Highlight the lines you've added.
- Screenshots of your GAT training output, you can just screenshot the last 10 epochs with test set results.
- Your response to the written component of question 3.6. Your analysis should not exceed **3** sentences.



## 4 Deep Q-Learning Network (DQN) [4pt]

In this part of the assignment, we will apply Reinforcement Learning (DQN) to tackle the CartPole Balancing game, the game that seems easy but actually quite hard. If you haven't tried it yet, I recommend you try it first [the link]. However, the difficult game for human may be very simple to a computer.

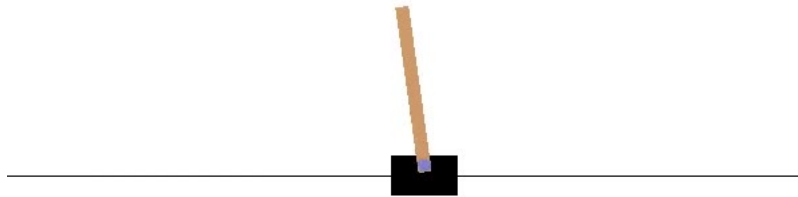


Figure 1: Image of the CartPole Balancing game from OpenAI Gym. Brockman et al. [2016]

### DQN Overview

Reinforcement learning defines an environment for the agent to perform certain actions (according to the policy) that maximize the reward at every time stamp. Essentially, our aim is to train an agent that tries to maximize the discounted, cumulative reward  $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$ . Because we assume there can be infinite time stamps, the discount factor,  $\gamma$ , is a constant between 0 and 1 that ensures the sum converges. It makes rewards from the uncertain far future less important for our agent than the ones in the near future.

The idea of Q-learning is that if we have a function  $Q^*(state, action)$  that outputs the maximum expected cumulative reward achievable from a given state-action pair, we could easily construct a policy (action selection rule) that maximizes the reward:

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s, a) \quad (4.1)$$

However, we don't know everything about the world, so we don't have access to  $Q^*$ . But, since neural networks are universal function approximators, we can simply create one and train it to resemble  $Q^*$ . For our training update rule, we will use a fact that every  $Q$  function for some policies obeys the Bellman equation:

$$Q^\pi(s, a) = r + \gamma Q^\pi(s', \pi(s')) \quad (4.2)$$

An intuitive explanation of the structure of the Bellman equation is as follows. Suppose that the agent has received reward  $r_t$  at the current state, then the maximum discounted reward from

this point onward is equal to the current reward plus the maximum expected discounted reward  $\gamma Q^*(s_{t+1}, a_{t+1})$  from the next stage onward. The difference between the two sides of the equality is known as the temporal difference error,  $\delta$ :

$$\delta = Q(s, a) - (r + \gamma \max_a Q(s', a)) \quad (4.3)$$

Our goal is to minimise this error, so that we can have a good  $Q$  function to estimate the rewards given any state-action pair.

## Experiments

Open the Colab notebook link to begin: [DQN notebook link]. Read through the notebook and play around with it. More detailed instructions are given in the notebook. Have fun!

### 1. [1pt] Implementation of $\epsilon - \text{greedy}$

Complete the function `get_action` for the agent to select an action based on current state. We want to balance exploitation and exploration through  $\epsilon - \text{greedy}$ , which is explained in the notebook. **Include your code snippet in your write-up.**

### 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.**

### 3. [2pt] Train your DQN Agent

After implementing the required functions, now you can train your DQN Agent, and you are suggested to tune the hyperparameters listed in the notebook. Hyperparameters are important to train a good agent. After all of these, now you can validate your model by playing the CartPole Balance game! **List the hyperparameters' value you choose, your epsilon decay rule and summarize your final results from the visualizations in a few sentences in your write-up.**

## Deliverables

Create a section in your report called **Deep Q-learning Network**. Add the following:

- Screenshots of your `get_action`, `train`, `epsilon decay rule` implementations. Highlight the lines you've added.
- Screenshots of your `reward curve`, and the `end time frame of your cartpole video` (to show how many seconds you can balance the cartpole).
- Your response to the written component of `question 4.3`. Your analysis should not exceed **3** sentences.

## What you need to submit

- Your code files: `gnn.ipynb`, `dqn.ipynb`.
- A PDF document titled `a4-writeup.pdf` containing **code screenshots, any experiment results or visualizations**, as well as **your answers to the written questions**.

## Further Resources

For further reading on GANs, DCGAN, GCN and DQN, the following links may be useful:

1. Generative Adversarial Nets (Goodfellow et al., 2014)
2. Deconvolution and Checkerboard Artifacts (Odena et al., 2016)
3. Progressive Growing of GANs (Karras et al. [2017])
4. Analyzing and Improving the Image Quality of StyleGAN (Karras et al. [2020])
5. An Introduction to GANs in Tensorflow
6. Generative Models Blog Post from OpenAI
7. Playing Atari with Deep Reinforcement Learning (Mnih et al., 2013)
8. Deep Reinforcement Learning: A Brief Survey (Arulkumaran et al., 2017)

## References

- Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL <http://arxiv.org/abs/1609.02907>.
- Petar Velicković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL <https://openreview.net/forum?id=rJXMpikCZ>. accepted as poster.
- Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–106, 2008. URL <http://www.cs.iit.edu/~ml/pdfs/sen-aimag08.pdf>.
- Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.
- Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.
- Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8110–8119, 2020.