Due Date: March 22nd 2021, 11pm EST

Question 1 (4-6-4)**.** This question is about activation functions and vanishing/exploding gradients in recurrent neural networks (RNNs). Let $\sigma : \mathbb{R} \to \mathbb{R}$ be an activation function. When the argument is a vector, we apply σ element-wise. Consider the following recurrent unit:

$$\boldsymbol{h}_t = \boldsymbol{W}\sigma(\boldsymbol{h}_{t-1}) + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b}$$

1.1 Show that applying the activation function in this way is equivalent to the conventional way of applying the activation function: $\boldsymbol{g}_t = \sigma(\boldsymbol{W}\boldsymbol{g}_{t-1} + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b})$ (i.e. express $\boldsymbol{g}_t$ in terms of $\boldsymbol{h}_t$). More formally, you need to prove it using mathematical induction. You only need to prove the induction step in this question, assuming your expression holds for time step $t-1$.

If the relations hold for $t-1$, then we have:

$$\boldsymbol{g}_{t-1} = \sigma(\boldsymbol{W}\boldsymbol{g}_{t-2} + \boldsymbol{U}\boldsymbol{x}_{t-1} + \boldsymbol{b})$$
$$\boldsymbol{h}_{t-1} = \boldsymbol{W}\sigma(\boldsymbol{h}_{t-2}) + \boldsymbol{U}\boldsymbol{x}_{t-1} + \boldsymbol{b}$$

Now replacing $\boldsymbol{h}_{t-1}$ into the equation for $\boldsymbol{h}_t$ we get:

$$\boldsymbol{h}_t = \boldsymbol{W}\sigma(\boldsymbol{h}_{t-1}) + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b} = \boldsymbol{W}\sigma(\boldsymbol{W}\sigma(\boldsymbol{h}_{t-2}) + \boldsymbol{U}\boldsymbol{x}_{t-1} + \boldsymbol{b}) + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b}$$

If we suppose that $\sigma(\boldsymbol{h}_t) = \boldsymbol{g}_t$, holds for $t-1, t-2$ we'll get:

$$\boldsymbol{h}_t = \boldsymbol{W}\sigma(\boldsymbol{W}\boldsymbol{g}_{t-2} + \boldsymbol{U}\boldsymbol{x}_{t-1} + \boldsymbol{b}) + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b}$$

Given the relation for $\boldsymbol{g}_t = \sigma(\boldsymbol{W}\boldsymbol{g}_{t-1} + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b})$, we can further simplify to obtain the following:

$$\boldsymbol{h}_t = \boldsymbol{W}\boldsymbol{g}_{t-1} + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b}$$

Now if we pass both sides through a sigmoid:

$$\sigma(\boldsymbol{h}_t) = \sigma(\boldsymbol{W}\boldsymbol{g}_{t-1} + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b})$$

Now using the equation for $\boldsymbol{g}_t$ for the right hand side, we will get:

$$\sigma(\boldsymbol{h}_t) = \boldsymbol{g}_t$$

Thus we have proved the inductive bias and shown that $\boldsymbol{g}_t = \sigma(\boldsymbol{h}_t)$.

*1.2 Let $||\boldsymbol{A}||$ denote the L_2 operator norm [1] of matrix $\boldsymbol{A}$ ($||\boldsymbol{A}|| := \max_{\boldsymbol{x}:||\boldsymbol{x}||=1} ||\boldsymbol{A}\boldsymbol{x}||$). Assume σ, seen as a function $\mathbb{R}^n \to \mathbb{R}^n$, has bounded differential, i.e. $\|D\sigma(\boldsymbol{x})\| \leq \gamma$ (here $\|\cdot\|$ is the L_2 operator norm) for some $\gamma > 0$ and for all $\boldsymbol{x}$. We denote as $\lambda_1(\cdot)$ the largest eigenvalue of a symmetric matrix. Show that if the largest eigenvalue of the weights is bounded by $\frac{\delta^2}{\gamma^2}$ for some $0 \leq \delta < 1$, gradients of the hidden state will vanish over time, i.e.

$$\lambda_1(\boldsymbol{W}^\top \boldsymbol{W}) \leq \frac{\delta^2}{\gamma^2} \quad \Longrightarrow \quad \left|\left|\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_0}\right|\right| \to 0 \text{ as } T \to \infty$$

1. The L_2 operator norm of a matrix $\boldsymbol{A}$ is is an *induced norm* corresponding to the L_2 norm of vectors. You can try to prove the given properties as an exercise.

Use the following properties of the L_2 operator norm

$$||\boldsymbol{AB}|| \leq ||\boldsymbol{A}||\,||\boldsymbol{B}|| \qquad \text{and} \qquad ||\boldsymbol{A}|| = \sqrt{\lambda_1(\boldsymbol{A}^\top \boldsymbol{A})}$$

From the previous part we have that $\boldsymbol{g}_t = \sigma(\boldsymbol{h}_t)$, also we have that $\boldsymbol{h}_t = \boldsymbol{W}\sigma(\boldsymbol{h}_{t-1}) + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b}$. Now taking the derivative w.r.t. $\boldsymbol{h}_0$ we arrive at teh following (using the chain rule):

$$\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_0} = \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_{T-1}} \frac{\partial \boldsymbol{h}_{T-1}}{\partial \boldsymbol{h}_{T-2}} \frac{\partial \boldsymbol{h}_{T-2}}{\partial \boldsymbol{h}_{T-3}} \cdots \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{h}_0}$$

But for any given $t \in [1, \ldots, T]$ we have that:

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} = \frac{\partial}{\partial \boldsymbol{h}_{t-1}}\Big(\boldsymbol{W}\sigma(\boldsymbol{h}_{t-1}) + \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{b}\Big) = \boldsymbol{W}\frac{\partial}{\partial \boldsymbol{h}_{t-1}}\sigma(\boldsymbol{h}_{t-1}) + \cancelto{0}{\frac{\partial}{\partial \boldsymbol{h}_{t-1}}\boldsymbol{U}\boldsymbol{x}_t} + \cancelto{0}{\frac{\partial}{\partial \boldsymbol{h}_{t-1}}\boldsymbol{b}} = \boldsymbol{W}D\sigma(\boldsymbol{h}_{t-1})$$

Replacing it back to the equation for $\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_0}$:

$$\begin{aligned}\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_0} &= \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_{T-1}} \frac{\partial \boldsymbol{h}_{T-1}}{\partial \boldsymbol{h}_{T-2}} \frac{\partial \boldsymbol{h}_{T-2}}{\partial \boldsymbol{h}_{T-3}} \cdots \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{h}_0} \\ &= \boldsymbol{W}D\sigma(\boldsymbol{h}_{T-1})\boldsymbol{W}D\sigma(\boldsymbol{h}_{T-2})\ldots\boldsymbol{W}D\sigma(\boldsymbol{h}_0)\end{aligned}$$

Now applying the property $||\boldsymbol{AB}|| \leq ||\boldsymbol{A}||\,||\boldsymbol{B}||$ multiple times:

$$\begin{aligned}\|\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_0}\| &= \Big\|\boldsymbol{W}D\sigma(\boldsymbol{h}_{T-1})\boldsymbol{W}D\sigma(\boldsymbol{h}_{T-2})\ldots\boldsymbol{W}D\sigma(\boldsymbol{h}_0)\Big\| \\ &\leq \|\boldsymbol{W}\|\|D\sigma(\boldsymbol{h}_{T-1})\|\ldots\|\boldsymbol{W}\|\|D\sigma(\boldsymbol{h}_0)\| \\ &\leq \|\boldsymbol{W}\|^T\gamma^T\end{aligned}$$

Where the last inequality comes from the assumption $\|D\sigma(\boldsymbol{x})\| \leq \gamma$. We can rewrite this in the following form:

$$\|\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_0}\| \leq \big(\|\boldsymbol{W}\|^2\gamma^2\big)^{T/2}$$

But from the other matrix operator norm property $||\boldsymbol{A}|| = \sqrt{\lambda_1(\boldsymbol{A}^\top \boldsymbol{A})}$, we know that $\|\boldsymbol{W}\|^2 = \lambda_1(\boldsymbol{W}^\top \boldsymbol{W})$. So if this largest eigenvalue is bounded by $\frac{\delta^2}{\gamma^2}$, then we have that $\|\boldsymbol{W}\|^2\gamma^2 \leq \frac{\delta^2}{\gamma^2}\gamma^2 = \delta^2$. Plugging this back to the derivative inequality:

$$\|\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_0}\| \leq (\delta^2)^{T/2} = \delta^T$$

So the derivative is *upper bounded* by δ^T and this upper bound goes to 0 in the limit:

$$\lim_{T\to\infty} \|\frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_0}\| \leq \lim_{T\to\infty} \delta^T = 0$$

Last inequality comes from the fact that $0 \leq \delta < 1$. So if the upper bound of the derivative tends to zero as $T \to \infty$, then consecutive derivatives cannot be larger than the upper bound so in the limit, those derivatives will vanish over time and we have the vanishing gradients problem.

1.3 What do you think will happen to the gradients of the hidden state if the condition in the previous question is reversed, i.e. if the largest eigenvalue of the weights is larger than $\frac{\delta^2}{\gamma^2}$? Is this condition *necessary* and/or *sufficient* for the gradient to explode ? (Answer in 1-2 sentences).

For the gradients to explode, the upper bound on the derivatives should tend to ∞ over time, if the upper bound asymptotically becomes a finite value, then the gradients cannot go beyond that value and we will not have any explosion. So the *necessary* condition is that the upper bound in the previous section becomes *unbounded* itself. Or it's necessary that the upper bound is not zero. Hence if the largest eigenvalue is not smaller than $\frac{\delta^2}{\gamma^2}$, then the upper bound won't tend to zero and gradient explosion is *possible*. It won't certainly explode but now it has the potential to explode so this condition is a *necessary* condition.

Question 2 (1-12-2-6)**.** Suppose that we have a vocabulary containing N possible words, including a special token `<BOS>` to indicate the beginning of a sentence. Recall that in general, a language model with a full context can be written as

$$p(w_1, w_2, \dots, w_T \mid w_0) = \prod_{t=1}^{T} p(w_t \mid w_0, \dots, w_{t-1}).$$

We will use the notation $\boldsymbol{w}_{0:t-1}$ to denote the (partial) sequence $(w_0, \dots, w_{t-1})$. Once we have a fully trained language model, we would like to generate realistic sequences of words from our language model, starting with our special token `<BOS>`. In particular, we might be interested in generating the most likely sequence $\boldsymbol{w}^\star_{1:T}$ under this model, defined as

$$\boldsymbol{w}^\star_{1:T} = \arg\max_{\boldsymbol{w}_{1:T}} p(\boldsymbol{w}_{1:T} \mid w_0 = \texttt{<BOS>}).$$

For clarity we will drop the explicit conditioning on w_0, assuming from now on that the sequences always start with the `<BOS>` token.

2.1 How many possible sequences of length $T+1$ starting with the token `<BOS>` can be generated in total ? Give an exact expression, without the O notation. Note that the length $T+1$ here includes the `<BOS>` token.

According to the discussions on piazza, `<BOS>` can appear any where in the middle of the sequence, so we got T possible spots and each spot has the possibility to be filled with any of the N words in the vocabulary (which includes `<BOS>` too). So this gives N^T possible sequences of length $T+1$ starting with token `<BOS>`.

2.2 In this question only, we will assume that our language model satisfies the *Markov property*

$$\forall\, t > 0,\ p(w_t \mid w_0, \dots, w_{t-1}) = p(w_t \mid w_{t-1}).$$

Moreover, we will assume that the model is time-homogeneous, meaning that there exists a matrix $\boldsymbol{P} \in \mathbb{R}^{N \times N}$ such that $\forall\, t > 0,\ p(w_t = j \mid w_{t-1} = i) = [\boldsymbol{P}]_{i,j} = P_{ij}$.

2.2.a Let $\boldsymbol{\delta}_t \in \mathbb{R}^N$ be the vector of size N defined for $t > 0$ as

$$\delta_t(j) = \max_{\boldsymbol{w}_{1:t-1}} p(\boldsymbol{w}_{1:t-1}, w_t = j),$$

where $\boldsymbol{w}_{1:0} = \emptyset$. Using the following convention for $\boldsymbol{\delta}_0$

$$\delta_0(j) = \begin{cases} 1 & \text{if } j = \texttt{<BOS>} \\ 0 & \text{otherwise,} \end{cases}$$

give the recurrence relation satisfied by $\boldsymbol{\delta}_t$ (for $t > 0$).
Starting by the definition and using $P(A, B) = P(A \mid B)P(B)$:

$$\delta_t(j) = \max_{w_{1:t-1}} p(w_{1:t-1}, w_t = j) = \max_{w_{1:t-1}} p(w_t = j \mid w_{1:t-1})p(w_{1:t-1})$$

Now employing the Markov assumption on $p(w_t = j \mid w_{1:t-1})$ we get:

$$\delta_t(j) = \max_{w_{1:t-1}} p(w_t = j \mid w_{1:t-1})p(w_{1:t-1}) = \max_{w_{1:t-1}} p(w_t = j \mid w_{t-1})p(w_{1:t-1})$$

But we know that the product of a sequence of numbers is maximized when each of them is at its maximum, so $\max_{\text{args}} f(\text{args})g(\text{args}) = \max_{\text{args}} f(\text{args}) \max_{\text{args}} g(\text{args})$, so we can rewrite our maximization as follows:

$$\delta_t(j) = \max_{w_{1:t-1}} p(w_t = j \mid w_{t-1})p(w_{1:t-1}) = \max_{w_{1:t-1}} p(w_t = j \mid w_{t-1}) \max_{w_{1:t-1}} p(w_{1:t-1})$$

But notice that $\max_{w_{1:t-1}} p(w_t = j \mid w_{t-1})$ can only be affected by w_{t-1} among the max arguments $w_{1:t-1}$, so we can simplify that:

$$\begin{aligned} \delta_t(j) &= \max_{w_{1:t-1}} p(w_t = j \mid w_{t-1}) \max_{w_{1:t-1}} p(w_{1:t-1}) \\ &= \max_{w_{t-1}} p(w_t = j \mid w_{t-1}) \max_{w_{1:t-1}} p(w_{1:t-1}) \end{aligned}$$

Now notice how we can expand the second term above:

$$\begin{aligned} \max_{w_{1:t-1}} p(w_{1:t-1}) &= \max_{w_{1:t-2}, w_{t-1}} p(w_{1:t-2}, w_{t-1}) \\ &= \max_{w_{t-1}} \delta_{t-1}(w_{t-1}) \end{aligned}$$

Rewriting the δ_t again with this new observation and noting that $p(w_t = j \mid w_{t-1}) = P_{w_{t-1}w_t}$ and mixing the max operators (since they're the same):

$$\begin{aligned} \delta_t(w_t = j) &= \max_{w_{t-1}} p(w_t = j \mid w_{t-1}) \max_{w_{1:t-1}} p(w_{1:t-1}) \\ &= \max_{w_{t-1}} P_{w_{t-1}j} \delta_{t-1}(w_{t-1}) \end{aligned}$$

So after cleaning up a bit, we have arrived at our recurrence relation:

$$\delta_t(j) = \max_i P_{ij} \delta_{t-1}(i)$$

Also note this recurrence has to start with $t = 1$, so $\delta_1(j) = \max_i P_{ij}\delta_0(i)$, and we need an initialization for δ_0. But $\delta_0(i)$ is given in the problem statement and is zero anywhere $i \neq \texttt{<BOS>}$, so the maximum value for $\max_i P_{ij}\delta_0(i)$ is $P_{\texttt{<BOS>}j} = \delta_1(j)$.

2.2.b Show that $w_T^\star = \arg\max_j \delta_T(j)$.

$w_T^\star$ is the T^{th} element of the most likely sequence $\boldsymbol{w}_{1:T}^\star = \arg\max_{\boldsymbol{w}_{1:T}} p(\boldsymbol{w}_{1:T})$ (which of course is conditioned on $w_0 =$ <BOS> but the notation is dropped for convenience), but we know how to maximize the probability inside the arg max:

$$\max_{\boldsymbol{w}_{1:T}} p(\boldsymbol{w}_{1:T}) = \max_{w_T} \max_{\boldsymbol{w}_{1:T-1}} p(\boldsymbol{w}_{1:T}) = \max_{w_T} \max_{\boldsymbol{w}_{1:T-1}} p(\boldsymbol{w}_{1:T-1}, w_T)$$

But recall the definition of $\delta_t(j) = \max_{\boldsymbol{w}_{1:t-1}} p(\boldsymbol{w}_{1:t-1}, w_t = j)$, we can directly plug this back into the above equation to get:

$$\max_{\boldsymbol{w}_{1:T}} p(\boldsymbol{w}_{1:T}) = \max_{w_T} \max_{\boldsymbol{w}_{1:T-1}} p(\boldsymbol{w}_{1:T-1}, w_T) = \max_{w_T} \delta_T(w_T)$$

So we know that the probability for the most likely sequence is $\max_{w_T} \delta_T(w_T)$, and thus the sequence ending with the argument that maximizes this probability is the most likely sequence, and it ends with that argument, hence $w_T^\star$ is the argument which maximizes this term $\max_{w_T} \delta_T(w_T)$, so it's the arg max of that or $w_T^\star = \arg\max_j \delta_T(j)$.

2.2.c Let $\boldsymbol{a}_t \in \{1, \ldots, N\}^N$ be the vector of size N defined for $t > 0$ as

$$a_t(j) = \arg\max_i P_{ij} \delta_{t-1}(i).$$

Show that $\forall\, 0 < t < T$, $w_t^\star = a_{t+1}(w_{t+1}^\star)$.

To solve this part, we need to be careful with the definitions. $w_t^\star$ is the tth element of the most likely sequence $\boldsymbol{w}_{1:T}^\star$. What is $\delta_t(j)$? It's the maximum possible probability that the tth word is the j^{th} word of the vocabulary. $\arg\max_j \delta_T(j)$ gives the most likely word for ending the most likely sequence at position T. Now what is the most likely word at one position before or at $(T-1)^{th}$?

$$\delta_t(j) = \max_i P_{ij} \delta_{t-1}(i)$$

But note that any $\delta_t(j) = \max_i P_{ij} \delta_{t-1}(i)$ maximizes over all possible *previous words* to make the probability of having word j at the next position the highest. In particular, this holds for $w_T^\star$ as well so i that maximizes $\delta_T(w_T^\star) = \max_i P_{iw_T^\star} \delta_{T-1}(i)$, is the most likely word preceding $w_T^\star$ or equivalently we can say $w_{T-1}^\star = \arg\max_i P_{iw_T^\star} \delta_{T-1}(i)$. Considering this as the base case of an induction proof, we can prove the induction step as follows:
Suppose that for some $t < T-1$ we have $w_t^\star = \arg\max_i P_{iw_{t+1}^\star} \delta_t(i)$, then we need to show this holds for $t-1$ as well because in this case we're traversing backwards, so the induction step is also decreasing. Again using a similar argument, $\delta_t(j)$ is the maximum possible probability that the tth word is the j^{th} word of the vocabulary, so if this probability is maximized when the j^{th} word is $w_t^\star$, then this maximum value for $\delta_t(j)$ is achieved by $\delta_t(w_t^\star) = \max_i P_{iw_t^\star} \delta_{t-1}(i)$. Again this maximization is over all possible *previous words* to make the probability of having word $w_t^\star$ at the next position the highest so i that maximizes $\max_i P_{iw_t^\star} \delta_{t-1}(i)$, is the most likely word preceding $w_t^\star$ or equivalently we can say $w_{t-1}^\star = \arg\max_i P_{iw_t^\star} \delta_{t-1}(i)$. Thus we have completed the induction step since we showed:

$$w_t^\star = \arg\max_i P_{iw_{t+1}^\star} \delta_t(i) \implies w_{t-1}^\star = \arg\max_i P_{iw_t^\star} \delta_{t-1}(i)$$

Now simply replacing the introduced operator $a_t(j) = \arg\max_i P_{ij}\delta_{t-1}(i)$, we get:

$$w_t^\star = a_{t+1}(w_{t+1}^\star) \Longrightarrow w_{t-1}^\star = a_t(w_t^\star)$$

And our inductive prove holds for this too since it's just a rewritten version of our derivation, thus our proof concludes and we have $w_t^\star = a_{t+1}(w_{t+1}^\star)$.

2.2.d Using the previous questions, write the pseudo-code of an algorithm to compute the sequence $\boldsymbol{w}_{1:T}^\star$ using dynamic programming. This is called *Viterbi decoding*.

Algorithm 1: Viterbi decoding

Input: A Markovian language model or matrix $\boldsymbol{P} \in \mathbb{R}^{N\times N}$ such that
$\forall\, t > 0,\ p(w_t = j \mid w_{t-1} = i) = [\boldsymbol{P}]_{i,j} = P_{ij}$
Output: The most likely sequences $\boldsymbol{w}_{1:T}$
Initialization: $\delta_1(j) \leftarrow P_{\texttt{<BOS>}j}$ for all $j \in \{1, \dots, N\}$
Do the forward pass, i.e.:
for $t = 1$ **to** T **do**
  for $j = 1$ **to** N **do**
    $\delta_t(j) \leftarrow \max_i P_{ij}\delta_{t-1}(i)$
Compute $w_T^\star \longleftarrow \arg\max_i \delta_T^\star(i)$
Create an empty list $\boldsymbol{w}$ and add $w_T^\star$ at the end of that list
Recursively do the backward pass, i.e.:
for $t = T - 1$ **to** 1 **do**
  $w_t^\star \longleftarrow a_{t+1}(w_{t+1}^\star)$ ($w_{t+1}^\star$ is the first element of $\boldsymbol{w}$ at any t)
  Append $w_t^\star$ to the beginning of $\boldsymbol{w}$
Return the list $\boldsymbol{w}$ as the desired sequence $\boldsymbol{w}_{1:T}^\star$

2.2.e What is the time complexity of this algorithm ? Its space complexity ? Write the algorithmic complexities using the O notation. Comment on the efficiency of this algorithm compared to naively searching for $\boldsymbol{w}_{1:T}^\star$ by enumerating all possible sequences (based on your answer to question 1).

The time complexity of Viterbi decoding:

i. Initialization $\rightarrow O(N)$
ii. Forward pass: Computing $P_{ij}\delta_{t-1}(i)$ for a given j and for all $i \in \{1, \dots, N\}$ requires $O(N)$ time. Finding the maximum over N elements has the worst case complexity of $O(N)$ because we can't find deterministically find the max without looking at all elements. Since the for loop is over all j, t, then the time complexity would be $O(N + N) \times O(TN) = O(TN^2)$
iii. Computing $w_T^\star$ requires doing an arg max over N possible words, which again like the max itself will take a worst case of $O(N)$ time.
iv. Creating a list and appending $w_T^\star$ to that $\rightarrow O(1)$
v. Backward pass: for each t, there's an arg max taking $O(N)$ in the worst case, appending takes $O(1)$, and since there are $T - 1$ of such steps, the backward pass will contribute to a time complexity of $O(N) \times O(1) \times O(T) = O(NT)$

So the overall time complexity would be $O(N)+O(TN^2)+O(N)+O(1)+O(NT) = O(TN^2)$
The memory complexity of Viterbi decoding:

i. keeping the matrix $\boldsymbol{P} \in \mathbb{R}^{N \times N} \rightarrow O(N^2)$

ii. Initialization and keeping $\delta_1(j)$ for all $j \in \{1, \dots, N\} \rightarrow O(N)$

iii. Forward pass: Computing $P_{ij}\delta_{t-1}(i)$ for a given j and for all $i \in \{1, \dots, N\}$ requires keeping $\delta_{t-1}(i)$ at any t for all words, but once computed, we do need to keep them for the backward pass and operator $a_t(j)$. Furthermore, we have to keep the products $P_{ij}\delta_{t-1}(i)$ which takes $O(N)$ memory, but unlike δs, after finding the maximum, we only keep a single value for each j. So it takes $O(N)$ memory. This would be the overall memory complexity of the forward pass as the for loop doesn't affect it because we continuously replace those values and don't add anything to the memory. But for keeping all δs we need $O(NT)$ memory. So the totall memory complexity would be $O(NT)$ for δs and a reusable $O(N)$ for finding the maximum for each j.

iv. Computing $w_T^\star$ requires doing an arg max over N possible words, which again like the max itself will take a memory of $O(N)$ to keep all values and go over them.

v. Creating a list and appending $w_T^\star$ to that $\rightarrow O(1)$ memory complexity

vi. Backward pass: for each t, only having $\boldsymbol{w}$ in addition to the δs that we have kept is enough. The $a_t(j)$ operator requires keeping N products and finding the arg max over them takes another $O(N)$ in the worst case, but this won't add up for consecutive iterations and can be replaced. Appending takes $O(1)$. So in totall, the backward pass will contribute to a memory complexity of $O(N)$

So the overall memory complexity would be $O(N^2) + O(N) + O(NT) + O(N) + O(N) + O(1) + O(N) = O(N^2)$. The major component contributing to the memory complexity is the matrix $\boldsymbol{P}$.

Using the naive approach, requires computing N^T probabilities for sequences of length T so if each probability computation takes $O(T)$, then the total computation for all the probabilities would give rise to an *exponential* time complexity of $O(TN^T)$. To compute the maximum, we can enumerate all of these N^T probabilities and at any time keep the highest value so far (taking another worst case $O(N^T)$ time complexity). But to find the most likely sequence, we need to keep all N^T probabilities (hence $O(N^T)$), and all possible sequences corresponding to those probabilities so it'll consume $O(N^T) + O(N^T \times T) = O(TN^T)$. (Each sequence requires a memory of $O(T)$.)

Comparing the approaches:

Method	Time Complexity	Memory Complexity
Naive	$O(TN^T)$	$O(TN^T)$
Viterbi	$O(TN^2)$	$O(N^2)$

So we observe a significant improvement from the exponential dependence on the very huge number N to a squared dependence to N which is a significant improvement. Also for the memory complexity, unless $T \not\geq N$ (which always is !), then the memory complexity doesn't have the T factor which could be another advantage. So Viterbi decoding is by far more efficient than the naive approach of enumerating all possibilities.

However one can define an ordering for constructing all N^T sequences given the index, and thus at each time computes a likelihood and just keeps the highest value so far and the index of that sequence, then the sequence of length T can be easily generated if we know

how to create it from the index. In this case, the space complexity of the naive approach would just be $O(T)$ so Viterbi will only be advantageous only in terms of time complexity.

2.2.f The component essential for the Viterbi algorithm is the time-homogeneous Markov assumption that constitutes those recursions we derived earlier. Without a Markov assumption, we need to enumerate all input possibilities at each step and use the multiplication of these conditionals to obtain the joint. So LSTM and GRU cannot exploit the Viterbi algorithm because they model the full conditional $p(w_t \mid w_{1:t-1})$ thus they won't satisfy any n^th order Markov property and cannot be used for Viterbi decoding as there will be no recursion. But a simple old RNN that satisfies some n^th order Markov property can yield a recursion (by defining a larger matrix $\boldsymbol{P}$) and then we can exploit the Viterbi algorithm with such language model.
For exact inference in LSTM/GRU, we need to enumerate all input possibilities at each step and use the multiplication of these conditionals to obtain the joint. Once we have that, we can take the maximum and find the most likely sequence. How to obtain these joints ? This is possible but tremendously painful. Suppose we're interested in finding the most likely sequence of length T. At any time $t \leq T$, we should feed all possible sequences of length $t-1$ to the LSTM since now it is responsible for returning $p(w_t \mid w_{1:t-1})$. But these are conditionals, what about the joints ? At time 1 we have $p(w_1 \mid \texttt{<BOS>}$, multiplying $p(\texttt{<BOS>})p(w_1 \mid \texttt{<BOS>} = p(2_1, w_0)$. We should keep it for all words in the vocabulary. Then for $t = 2$ we should compute all possible conditionals (by feeding all possible combinations) $p(w_2 \mid w_{0:1})$, then multiply them by the corresponding probability $p(w_{0:1})$ to get all possible joints $p(w_{0:2})$. Continuing like that we can get all joints. Then we can take the maximum over the joint at $t = T$ to find the most likely sequence. It's more finding the exact solution of maximum likelihood by enumerating all possibilities, not the Viterbi algorithm.

Algorithm 2: Beam search decoding

Input: A language model $p(\boldsymbol{w}_{1:T} \mid w_0)$, the beam width B
Output: B sequences $\boldsymbol{w}_{1:T}^{(b)}$ for $b \in \{1, \ldots, B\}$
Initialization: $w_0^{(b)} \leftarrow \texttt{<BOS>}$ for all $b \in \{1, \ldots, B\}$
Initial log-likelihoods: $l_0^{(b)} \leftarrow 0$ for all $b \in \{1, \ldots, B\}$
for $t = 1$ **to** T **do**
  for $b = 1$ **to** B **do**
    for $j = 1$ **to** N **do**
      $s_b(j) \leftarrow l_{t-1}^{(b)} + \log p(w_t = j \mid \boldsymbol{w}_{0:t-1}^{(b)})$
  for $b = 1$ **to** B **do**
    Find (b', j) such that $s_{b'}(j)$ is the b-th largest score
    Save the partial sequence b': $\widetilde{\boldsymbol{w}}_{0:t-1}^{(b)} \leftarrow \boldsymbol{w}_{0:t-1}^{(b')}$
    Add the word j to the sequence b: $w_t^{(b)} \leftarrow j$
    Update the log-likelihood: $l_t^{(b)} \leftarrow s_{b'}(j)$
  Assign the partial sequences: $\boldsymbol{w}_{0:t-1}^{(b)} \leftarrow \widetilde{\boldsymbol{w}}_{0:t-1}^{(b)}$ for all $b \in \{1, \ldots, B\}$

2.3 What is the time complexity of Algorithm 2 ? Its space complexity ? Write the algorithmic complexities using the O notation, as a function of T, B, and N. Is this a practical decoding algorithm when the size of the vocabulary is large ?

The time complexity of Beam search decoding:

(a) Initialization of $w_0^{(b)}$ and $l_0^{(b)} \rightarrow O(B) + O(B) = O(B)$

(b) Computing $s_b(j)$s for a given j, b takes $O(1)$ for computing the output of the language model $\log p(w_t = j \mid \boldsymbol{w}_{0:t-1}^{(b)})$, and $O(1)$ for addition, so for all $j \in \{1, \ldots, N\}$ and all $b \in \{1, \ldots, B\}$ it requires $O(BN)$ time.

(c) Finding the b-th largest score and the corresponding j requires a search over BN elements $(s_b(j))$ which takes a worst case of $O(BN \log BN)$ time complexity to be sorted using merge sort, then finding the b-th largest score takes $O(1)$.

(d) Saving and adding the word to a sequence take $O(1)$ time complexity.

(e) Updating the log-likelihood also takes $O(1)$ for a given t, b, j.

(f) So each iteration of the second loop over b takes $O(BN \log BN)$, thus the complete for loop take $O(B) \times O(BN \log BN) = O(B^2 N \log BN)$

(g) Assigning the partial sequence of length t takes $O(T)$ in the worst case, and doing that for all b will result in a factor of $O(B)$, hence $O(BT)$.

So for each outer loop or for each t, we have a time complexity of

$$O(B) + O(BN) + O(B^2 N \log BN) + O(1) + O(BT) = O(B(N + T + BN \log BN))$$

Now taking into account that there are T iterations, the overall time complexity would be larger by a factor of T: $O(BT(N + T + BN \log BN))$

The memory complexity of Beam search decoding:

(a) Initialization and keeping $w_0^{(b)}$ and $l_0^{(b)}$ for all $b \in \{1, \ldots, B\} \rightarrow O(B) + O(B) = O(B)$

(b) At any t, we need to have kept all $l_{t-1}^{(b)}$ which takes $O(B)$ memory and we considered that previously when initializing $l_0^{(b)}$s, but we don't need to keep them for all t, just the previous step suffices. But to keep the values of $s_b(j)$ we need $O(BN)$ memory which are reusable for each iteration of t, meaning that we don't need to allocate new memory for each iteration, we can discard previous values and store the new ones.

(c) Finding (b', j) requires the same $O(BN)$ allocated memory above, nothing extra.

(d) Saving the partial sequence in the worst case requires a memory of $O(BT)$ because in the worst case we need to store b beams of length T.

(e) Updating the log-likelihood doesn't need extra memory as we'll replace the previous values.

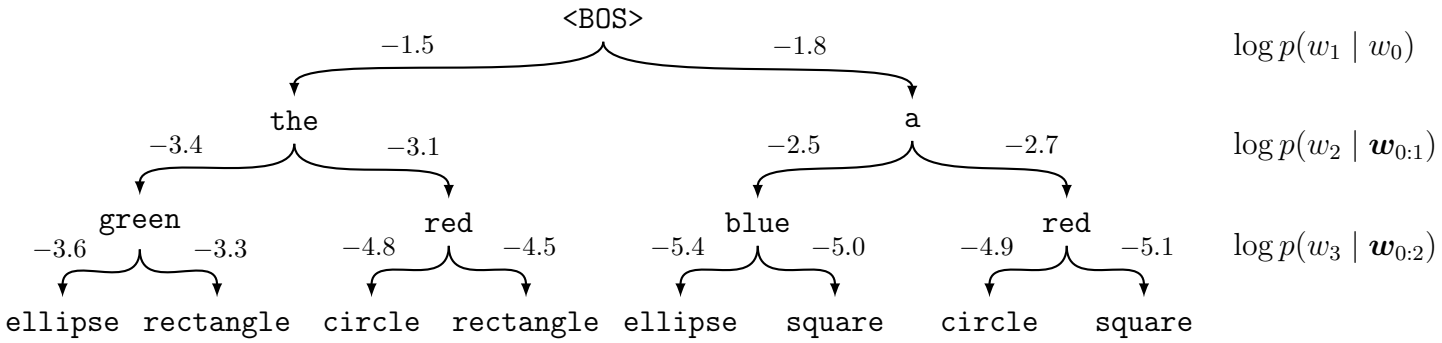
So the overall memory complexity would be $O(B) + O(BN) + O(BT) = O(B(N + T))$.
Now comparing the approaches:

Method	Time Complexity	Memory Complexity
Naive	$O(TN^T)$	$O(TN^T)$
Viterbi	$O(TN^2)$	$O(N^2)$
Beam Search	$O(BT(N + T + BN \log BN))$	$O(B(N + T))$

Thus indeed it's a practical approach compared to the naive and Viterbi decoding approaches specially when N is large. The reason is that its time complexity is not polynomial or exponential

in N but only in the worst case grows with $N \log N$ which is significantly better than N^2 and N^T that correspond to the Viterbi and naive approaches, respectively. The same goes for the memory complexity as well, instead of being exponential and polynomial N^T, N^2, it only linearly grows with N which is greatly advantageous.

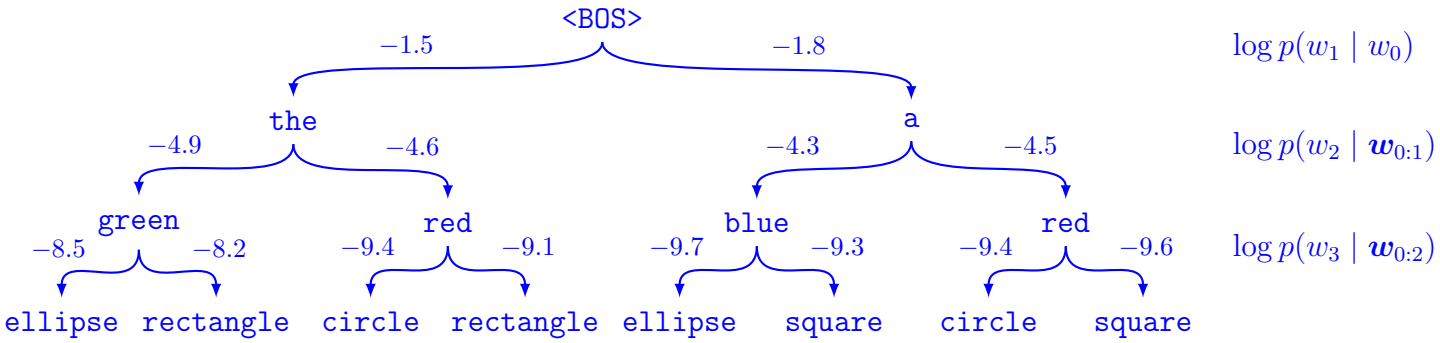
2.4 The different sequences that can be generated with a language model can be represented as a tree, where the nodes correspond to words and edges are labeled with the log-probability $\log p(w_t \mid \boldsymbol{w}_{0:t-1})$, depending on the path $\boldsymbol{w}_{0:t-1}$. In this question, consider the following language model (where the low probability paths have been removed for clarity)



2.4.a If you were given the whole tree, including the log-probabilities of all the missing branches (e.g. $\log p(w_2 = \texttt{a} \mid w_0 = \texttt{<BOS>}, w_1 = \texttt{red})$), could you apply Viterbi decoding from question 2 to this language model in order to find the most likely sequence $\boldsymbol{w}^\star_{1:3}$? Why ? Why not ? Find $\boldsymbol{w}^\star_{1:3}$, together with its corresponding log-likelihood $\log p(\boldsymbol{w}^\star_{1:3}) = \max_{\boldsymbol{w}_{1:3}} \log p(\boldsymbol{w}_{1:3})$.

No, we can't, since the component essential for the Viterbi algorithm is the time-homogeneous Markov assumption. Here, the probability for every word depends on all previous words up to root so we cannpt have any recursion and we need to do exact inference, keeping the track of all possible joints, and choosing the path from root to leaf having the highest joint log-likelihood. So here we need the joint probabilities not the conditionals. But we can easily obtain those joints, by traversing the tree and recursively adding the log probability of the parent conditional to that of the children and storing that as the new node attribute. This is because the log conditional for the root is $\log p(w_0 = \texttt{<BOS>}) = \log 1 = 0$, then for the first descendants of the root we have $\log p(w_1, w_0) = \log p(w_1 \mid w_0) + \log p(w_0) = \log p(w_1 \mid w_0) + 0 = \log p(w_1 \mid w_0)$. For next descendants, we need the joint $\log p(w_2, \boldsymbol{w}_{0:1}) = \log p(w_2 \mid \boldsymbol{w}_{0:1}) + \log p(w_1, w_0)$, but we got $\log p(w_1, w_0) = \log p(w_1 \mid w_0)$, so $\log p(w_2, \boldsymbol{w}_{0:1}) = \log p(w_2 \mid \boldsymbol{w}_{0:1}) + \log p(w_1 \mid w_0)$. The same goes for any descendant in the tree because the joint is obtainable by adding the log-conditional given previous words and the log-joint of the previous words.

Now that we have the joints, we can compute $p_t(w_t \mid w_{1:t-1})$ at each depth of the tree (corresponding to different ts) and then choose the path from root to some leaf having the highest summation of log-conditionals corresponding to the highest log-likelihood. Below is the same tree with node attributes being replaced by joints obtained by summing parent values to that of the children:



If we call the joints δs, then, $w_3^\star = \arg\max_j \delta_3(j)$, so let's see $\delta_3(j)$'s. At position 3, we have 4 possible words and the maximum value for the joint for each of them is (note that we have log of δs instead of δs which is not an issue since the log function is monotonic and a bijective):

$$\begin{aligned}
\delta_3(\texttt{ellipse}) &= \max\{-8.5, -9.7\} = -8.5 \\
\delta_3(\texttt{rectangle}) &= \max\{-8.2, -9.1\} = -8.2 \\
\delta_3(\texttt{circle}) &= \max\{-9.4, -9.4\} = -9.4 \\
\delta_3(\texttt{square}) &= \max\{-9.3, -9.6\} = -9.3
\end{aligned}$$

This leaves us with $w_3^\star = \arg\max_j \delta_3(j) = \texttt{rectangle}$. Tracing back this leaf to the root, we get:

$$\boldsymbol{w}_{0:3}^\star = \{\texttt{<BOS>}, \texttt{the}, \texttt{green}, \texttt{rectangle}\}$$

2.4.b *Greedy decoding* is a simple algorithm where the next word $\overline{w}_t$ is selected by maximizing the conditional probability $p(w_t \mid \overline{\boldsymbol{w}}_{0:t-1})$ (with $\overline{w}_0 = \texttt{<BOS>}$)

$$\overline{w}_t = \arg\max_{w_t} \log p(w_t \mid \overline{\boldsymbol{w}}_{0:t-1}).$$

Find $\overline{\boldsymbol{w}}_{1:3}$ using greedy decoding on this language model, and its log-likelihood $\log p(\overline{\boldsymbol{w}}_{1:3})$. Now it's even simpler and at each branch of the *log-conditional* tree (not the newly constructed *log-joint*) we just choose and traverse the one that is higher, thus we have:

$$\begin{aligned}
\overline{w}_1 &= \arg\max_{w_1}\{-1.5, -1.8\} = \texttt{the} \\
\overline{w}_2 &= \arg\max_{w_2}\{-3.4, -3.1\} = \texttt{red} \\
\overline{w}_3 &= \arg\max_{w_3}\{-4.8, -4.5\} = \texttt{rectangle}
\end{aligned}$$

So the sequence $\overline{\boldsymbol{w}}_{1:3}$ would be $\{\texttt{the}, \texttt{red}, \texttt{rectangle}\}$, and its log-likelihood is the summation of the numbers along the path which is -9.1 (from the previous log-joint tree). We see that at this computationally simpler greedy approach comes at the expense of returning a suboptimal solution because the log-likelihood for this sequence is lower than the one we derived using the Viterbi decoder so it's not really the most likely sequence overall. But this is the price we pay for going with a simpler algorithm.

2.4.c Apply beam search decoding (question 3) with a beam width $B = 2$ to this language model, and find $\boldsymbol{w}_{1:3}^{(1)}$ and $\boldsymbol{w}_{1:3}^{(2)}$, together with their respective log-likelihoods.

At each step t, we will keep two *beams* of words that are most likely based on log-likelihoods. Starting from the root, there are only two possibilities: $\{\texttt{the}, \texttt{a}\}$ with log-likelihoods $\{-1.5, -1.8\}$. Then going one step further, there are three possibilities and `red` appearing twice, so there are four candidates with their log-likelihoods calculated by adding the values for parents to childrens' (or using the log-joint tree constructed before): $\{\texttt{red}^{(1)} : -4.5, \texttt{blue} : -4.3, \texttt{red}^{(2)} : -4.6, \texttt{green} : -4.9\}$, discarding two beams and only keeping the top 2, we get the following beams:

$$\begin{aligned} b_1&: \quad \texttt{a} \quad \texttt{blue} \rightarrow \quad \text{log-likelihood} = -4.3 \\ b_2&: \quad \texttt{a} \quad \texttt{red} \rightarrow \quad \text{log-likelihood} = -4.6 \end{aligned}$$

Now we can go the last step and see where these beams take us. For the first beam there are two options: $\{\texttt{ellipse}, \texttt{square}\}$, and for the second beam there are two options: $\{\texttt{circle}, \texttt{rectangle}\}$, so below are all possible beams with their respective log-likelihoods:

$$\begin{aligned} b_1&: \quad \texttt{a,blue} \quad \texttt{ellipse} \rightarrow \quad \text{log-likelihood} = -9.7 \\ \checkmark b_2&: \quad \texttt{a,blue} \quad \texttt{square} \rightarrow \quad \text{log-likelihood} = -9.3 \\ \checkmark b_3&: \quad \texttt{a,red} \quad \texttt{circle} \rightarrow \quad \text{log-likelihood} = -9.4 \\ b_4&: \quad \texttt{a,red} \quad \texttt{square} \rightarrow \quad \text{log-likelihood} = -9.6 \end{aligned}$$

And the two top beams have been marked so to conclude:

$$\begin{aligned} \boldsymbol{w}_{1:3}^{(1)}&: \quad \texttt{a,blue,square} \quad \rightarrow \text{log-likelihood} = -9.3 \\ \boldsymbol{w}_{1:3}^{(2)}&: \quad \texttt{a,red,circle} \quad \rightarrow \text{log-likelihood} = -9.4 \end{aligned}$$

2.4.d Compare the behavior of these 3 decoding algorithms on this language model (in particular greedy decoding vs. maximum likelihood, and beam search decoding vs. the other two). How can you mitigate the limitations of beam search ?

We have computed the complexity of the Viterbi algorithm, and the Beam search earlier. Finding the exact maximum likelihood solution requires a lot of computation even if the matrix $\boldsymbol{P}$ is available, the greedy approach tries to find an approximation but is *very unstable* since it only looks one step forward and doesn't allow for any room to correct wrongly traversed paths, on the other hand, it's very cheap computationally, so we can run it several times and obtain multiple most likely sequences and decide based on that. However, beam search decoding can add the ingredient of robustness to our approach by not a significant computational overhead. It becomes more robust as we keep more and more beams by increasing B, and it reduces to the maximum likelihood solution if we keep all possible beams at each step. But even with a relatively small B we can keep several paths that potentially result in the maximum solution and cheaply traverse them as they're not too many. This way we've allowed some room for mistakes and have raised the probability of hitting the exact maximum likely solutions. In a sense beam search decoding introduces a tradeoff that avoids monitoring all possibilities but also avoids being unstable like the

greedy approach. So it seems to be the better approach. In this example the results are relatively different, but increasing the beam width could alleviate such discrepancies. In this problem it can be easily seen that a beam width of 4 could recover the exact solution.

Question 3 (2-3-4-4)**. Weight decay as L2 regularization** In this question, you will reconcile the relationship between L2 regularization and weight decay for the Stochastic Gradient Descent (SGD) and Adam optimizers. Imagine you are training a neural network (with learnable weights θ) with a loss function $L(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)})$, under two different schemes. The *weight decay* scheme uses a modified SGD update rule: the weights θ decay exponentially by a factor of λ. That is, the weights at iteration $i+1$ are computed as

$$\theta_{i+1} = \theta_i - \eta \frac{1}{m} \frac{\partial \sum_{j=1}^{m} L(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)})}{\partial \theta_i} + \lambda \theta_i$$

where η is the learning rate of the SGD optimizer. The *L2 regularization* scheme instead modifies the loss function (while maintaining the typical SGD or Adam update rules). The modified loss function is

$$L_{\text{reg}}(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)}) = L(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)}) + \gamma \|\theta\|_2^2$$

3.1 Prove that the *weight decay* scheme that employs the modified SGD update is identical to an *L2 regularization* scheme that employs a standard SGD update rule.

Single sample's and average over all samples' loss are:

$$\begin{aligned} L_{reg}^{j} &= L(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)}) + \gamma \|\theta\|_2^2 \\ L_{reg} &= \frac{1}{m} \sum_j L(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)}) + \frac{1}{m} \sum_j \gamma \|\theta\|_2^2 \\ &= \frac{1}{m} \sum_j L(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)}) + \gamma \|\theta\|_2^2 \end{aligned}$$

The standard SGD update rule would be $\theta_{t+1} = \theta_t - \eta \nabla_\theta L_{\text{reg}}$. Now taking the gradient:

$$\nabla_\theta L_{\text{reg}} = \frac{1}{m} \sum_j \frac{\partial L(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)})}{\partial \theta} + \gamma \frac{\partial}{\partial \theta} \|\theta\|_2^2$$

But $\|\theta\|_2^2 = \theta^\top \theta$, and thus $\frac{\partial}{\partial \theta} \|\theta\|_2^2 = \frac{\partial}{\partial \theta} \theta^\top \theta = 2\theta$, hence the overall update for the L2 regularized loss would be:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{m} \sum_j \frac{\partial L(f(\mathbf{x}^{(j)}, \theta), \mathbf{y}^{(j)})}{\partial \theta} - 2\eta\gamma\theta_t$$

Comparing this with the modified SGD for weight decay, we find that they're identical if we set $\lambda = -2\eta\gamma$. This concludes the proof.

3.2 This question refers to the Adam algorithm as described in the lecture slide (also identical to Algorithm 8.7 of the deep learning book). It turns out that a one-line change to this algorithms gives us Adam with an L2 regularization scheme. Identify the line of the algorithm that needs to change, and provide this one-line modification.

One (naive) way would be to add the L2 penalty to the loss and this will modify the line that computes the gradients:

$$\boldsymbol{h} \longleftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(\boldsymbol{x}^{(i)};\theta), \boldsymbol{y}^{(i)}) + \underbrace{\gamma\theta}_{\text{new term}}$$

We will soon see why it's naive, because it negates the motivation of L2 penalty to penalize all weights equally. Doing the modification here will result in imbalanced rate of changes for different weights.

3.3 Consider a "decoupled" weight decay scheme for the original Adam algorithm (see lecture slides, or equivalently, Algorithm 8.7 of the deep learning book) with the following two update rules.

— The **Adam-L2-reg** scheme computes the update by employing an L2 regularization scheme (same as the question above).

— The **Adam-weight-decay** scheme computes the update as $\boldsymbol{\Delta}\theta = -\left(\epsilon\frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}}+\delta} + \lambda\theta\right)$.

Now, assume that the neural network weights can be partitioned into two disjoint sets based on their magnitude: $\theta = \{\theta_{\text{small}}, \theta_{\text{large}}\}$, where each weight $\theta_s \in \theta_{\text{small}}$ has a much smaller gradient magnitude than each weight $\theta_l \in \theta_{\text{large}}$. Using this information provided, answer the following questions. In each case, provide a brief explanation as to why your answer holds.

(a) Under the **Adam-L2-reg** scheme, which set of weights among θ_{small} and θ_{large} would you expect to be regularized (i.e., driven closer to zero) more strongly than the other ? Why ?

Let's see how the updates will look like if we add the L2 penalty right to the loss and then take the gradient:

$$\theta_{t+1} = \theta_t - \eta\frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}$$

But what are $\hat{\boldsymbol{m}}_t, \hat{\boldsymbol{v}}$? They are unbiased estimates of the moving average of gradients and gradient squares.

$$\hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1-\beta_1^t} = \frac{\beta_1\boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t}{1-\beta_1^t}$$
$$\hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1-\beta_2^t} = \frac{\beta_2\boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2}{1-\beta_2^t}$$

In above equations, $\boldsymbol{g}_t$ is the gradient of the *regularized* loss, the one we modified in the previous section.

$$\boldsymbol{g}_t = \frac{1}{m}\nabla_{\theta_t} \sum_i L(f(\boldsymbol{x}^{(i)};\theta_t), \boldsymbol{y}^{(i)}) + \gamma\theta_t$$

Now plugging all these equations to the update rule:

$$\begin{aligned}
\theta_{t+1} &= \theta_t - \eta \frac{\beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\Big(\nabla_{\theta_t} L + \gamma\theta_t\Big)}{(1-\beta_1^t)(\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon)} \\
&= \theta_t - \eta \frac{\beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\Big(\nabla_{\theta_t} L + \gamma\theta_t\Big)}{(1-\beta_1^t)(\sqrt{\frac{\beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\nabla^2_{\theta_t} L}{1-\beta_2^t}} + \epsilon)} \\
&= \theta_t - \eta \frac{\beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)(\nabla_{\theta_t} L)}{(1-\beta_1^t)(\sqrt{\frac{\beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\nabla^2_{\theta_t} L}{1-\beta_2^t}} + \epsilon)} \underbrace{- \eta \frac{(1-\beta_1)(\gamma\theta_t)}{(1-\beta_1^t)(\sqrt{\frac{\beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\nabla^2_{\theta_t} L}{1-\beta_2^t}} + \epsilon)}}_{\text{regularization term}}
\end{aligned}$$

From the last equation we can draw conclusions about $\{\theta_{\text{small}}, \theta_{\text{large}}\}$. We observe that the weight decay term ($\gamma\theta_t$) resulting from the L2 penalty, is involved in the moving averages, and for θ_{large}, the denominator would be huge (because of large $\nabla^2_{\theta_t} L$) and thus results in less decay for this set of parameters, on the other hand, for θ_{small}, the gradients are smaller and thus the denominator is smaller and the regularization or decay is stronger for this set of parameters. This imbalanced strength of regularization for different set of parameters is undesired and contradicts the motivation of the L2 penalty in the first place: We want all weights to be decayed equally.

(b) Would your answer change for the **Adam-weight-decay** scheme ? Why/why not ?

If we add the weight decay term (or equivalently the gradient of the regularizer) only after all moving averages have been taken care of, then the decay will not depend on any moving average and thus the magnitude of gradients for different set of parameters won't affect it and the effect/strength of the regularizer would be balanced for all parameters regardless of their gradient magnitude.

(Note: for the two sub-parts above, we are interested in the rate at which the weights are regularized, *relative* to their initial magnitudes.)

3.4 In the context of all of the discussion above, argue that weight decay is a better scheme to employ as opposed to L2 regularization ; particularly in the context of adaptive gradient based optimizers. (Hint: think about how each of these schemes regularize each parameter, and also about what the overarching objective of regularization is).

We eluded to this problem several times earlier. The main objective of L2 regularizer is to decay all weights equally and the motivation for that comes from the observation that *networks with smaller weights or weights closer to zero* have shown *better generalization*. That is what we strive for by introducing the L2 penalty. Now in the context of adaptive gradient based optimizer, naively adding the regularization term to the loss and taking adaptive gradients, will bring in the gradient of *regularization term* into the play for all moving averages, and thus the strength or effect of the regularizer will depend on the magnitude of the gradient for each parameter which is undesirable. To alleviate this problem, using the alternative or equivalent of L2 penalty in a more appropriate part of the algorithm seems reasonable. That is, we add the weight decay to the last update where moving averages and bias corrections have been completed. Then we will use the weight decay and recover the same effect as balanced L2 penalty with an equal

strength over all parameters. That concludes the argument for why employing weight decay is a better scheme. Indeed they're the same, what matters is where you apply the regularization.

Question 4 (4-6-6)**.** This question is about normalization techniques.

4.1 Batch normalization, layer normalization and instance normalization all involve calculating the mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\sigma^2}$ with respect to different subsets of the tensor dimensions. Given the following 3D tensor, calculate the corresponding mean and variance tensors for each normalization technique: $\boldsymbol{\mu}_{batch}$, $\boldsymbol{\mu}_{layer}$, $\boldsymbol{\mu}_{instance}$, $\boldsymbol{\sigma}^2_{batch}$, $\boldsymbol{\sigma}^2_{layer}$, and $\boldsymbol{\sigma}^2_{instance}$.

$$\left[\begin{bmatrix}1,3,2\\1,2,3\end{bmatrix},\begin{bmatrix}3,3,2\\2,4,4\end{bmatrix},\begin{bmatrix}4,2,2\\1,2,4\end{bmatrix},\begin{bmatrix}3,3,2\\3,3,2\end{bmatrix}\right]$$

The size of this tensor is 4 x 2 x 3 which corresponds to the batch size, number of channels, and number of features respectively.

— Batch normalization:
Here we should take the mean over features *for each channel* (rows here). The mean is computed over all samples so we have ($\mu^{(2\times1)}_{\text{batch}}$):

$$\mu^{(0)}_{\text{batch}}: \quad \tfrac{1}{4\times3}\Big((1+3+2)+(3+3+2)+(4+2+2)+(3+3+2)\Big)=\tfrac{1}{12}(6+8+8+8)=2.5$$
$$\mu^{(1)}_{\text{batch}}: \quad \tfrac{1}{4\times3}\Big((1+2+3)+(2+4+4)+(1+2+4)+(3+3+2)\Big)=\tfrac{1}{12}(6+10+7+8)=2.58$$

Now computing the variance is simple, summing the squared difference of elements with the mean for each channel and over all samples:

$$\begin{aligned}\sigma^2_{\text{batch}}{}^{(0)} &= \frac{1}{4\times3}\Big((1.5^2+0.5^2+0.5^2)+(0.5^2+0.5^2+0.5^2)+(1.5^2+0.5^2+0.5^2)\\ &\quad+(0.5^2+0.5^2+0.5^2)\Big)=\frac{7}{12}\\ \sigma^2_{\text{batch}}{}^{(1)} &= \frac{1}{4\times3}\Big(((-1.58)^2+(-0.58)^2+0.42^2)+((-0.58)^2+1.42^2+1.42^2)\\ &\quad+((-1.58)^2+(-0.58)^2+1.42^2)+(0.42^2+0.42^2+(-0.58)^2)\Big)=\frac{12.92}{12}=1.08\end{aligned}$$

— Layer normalization:
Here we should take the mean over all features and channels *for each sample* (each matrix here). So the shape of μ_{layer} is 4×1:

$$\mu^{(0)}_{\text{layer}}: \quad \tfrac{1}{2\times3}\Big((1+3+2)+(1+2+3)\Big)=\tfrac{1}{6}(6+6)=2$$
$$\mu^{(1)}_{\text{layer}}: \quad \tfrac{1}{2\times3}\Big((3+3+2)+(2+4+4)\Big)=\tfrac{1}{6}(8+10)=3$$
$$\mu^{(2)}_{\text{layer}}: \quad \tfrac{1}{2\times3}\Big((4+2+2)+(1+2+4)\Big)=\tfrac{1}{6}(8+7)=2.5$$
$$\mu^{(3)}_{\text{layer}}: \quad \tfrac{1}{2\times3}\Big((3+3+2)+(3+3+2)\Big)=\tfrac{1}{6}(8+8)=2.67$$

Now computing the variance is simple, summing the squared difference of elements with the mean for *each sample* (over all features and channels):

$$\sigma^2_{\text{layer}}{}^{(0)} = \frac{1}{2\times 3}\Big(((-1)^2+1^2+0^2)+((-1)^2+0^2+1^2)\Big) = \frac{4}{6} = 0.67$$
$$\sigma^2_{\text{layer}}{}^{(1)} = \frac{1}{2\times 3}\Big((0^2+0^2+(-1)^2)+((-1)^2+1^2+1^2)\Big) = \frac{4}{6} = 0.67$$
$$\sigma^2_{\text{layer}}{}^{(2)} = \frac{1}{2\times 3}\Big((1.5^2+(-0.5)^2+(-0.5)^2)+((-1.5)^2+(-0.5)^2+(1.5)^2)\Big) = \frac{7.5}{6} = 1.25$$
$$\sigma^2_{\text{layer}}{}^{(3)} = \frac{1}{2\times 3}\Big((0.33^2+0.33^2+(-0.67)^2)+(0.33^2+0.33^2+(-0.67)^2)\Big) = \frac{1.33}{6} = 0.22$$

— Instance normalization:
Here we should take the mean over all features *for each channel and sample*. So we will have $4\times 2 = 8$ means and variances overall. The shape of each $\mu_{\text{instance}}, \sigma^2_{\text{instance}}$ is 1×1 (a scalar):

— Sample 1 $\rightarrow \begin{bmatrix}1,3,2\\1,2,3\end{bmatrix}$

channel 0 μ_{instance}: $\frac{1}{3}(1+3+2) = \frac{1}{3}(6) = 2$
channel 1 μ_{instance}: $\frac{1}{3}(1+2+3) = \frac{1}{3}(6) = 2$
channel 0 $\sigma^2_{\text{instance}}$: $\frac{1}{3}((-1)^2+1^2+0^2) = \frac{2}{3} = 0.67$
channel 1 $\sigma^2_{\text{instance}}$: $\frac{1}{3}((-1)^2+0^2+1^2) = \frac{2}{3} = 0.67$

— Sample 2 $\rightarrow \begin{bmatrix}3,3,2\\2,4,4\end{bmatrix}$

channel 0 μ_{instance}: $\frac{1}{3}(3+3+2) = \frac{8}{3} = 2.67$
channel 1 μ_{instance}: $\frac{1}{3}(2+4+4) = \frac{10}{3} = 3.33$
channel 0 $\sigma^2_{\text{instance}}$: $\frac{1}{3}(0.33^2+0.33^2+(-0.67)^2) = \frac{0.67}{3} = 0.22$
channel 1 $\sigma^2_{\text{instance}}$: $\frac{1}{3}((-1.33)^2+0.67^2+0.67^2) = \frac{2.67}{3} = 0.89$

— Sample 3 $\rightarrow \begin{bmatrix}4,2,2\\1,2,4\end{bmatrix}$

channel 0 μ_{instance}: $\frac{1}{3}(4+2+2) = \frac{8}{3} = 2.67$
channel 1 μ_{instance}: $\frac{1}{3}(1+2+4) = \frac{7}{3} = 2.33$
channel 0 $\sigma^2_{\text{instance}}$: $\frac{1}{3}(1.33^2+(-0.67)^2+(-0.67)^2) = \frac{2.67}{3} = 0.89$
channel 1 $\sigma^2_{\text{instance}}$: $\frac{1}{3}((-1.33)^2+(-0.33)^2+1.67^2) = \frac{4.67}{3} = 1.56$

— Sample 4 $\rightarrow \begin{bmatrix}3,3,2\\3,3,2\end{bmatrix}$

channel 0 μ_{instance}: $\frac{1}{3}(3+3+2) = \frac{8}{3} = 2.67$
channel 1 μ_{instance}: $\frac{1}{3}(3+3+2) = \frac{8}{3} = 2.67$
channel 0 $\sigma^2_{\text{instance}}$: $\frac{1}{3}(0.33^2+0.33^2+(-0.67)^2) = \frac{0.67}{3} = 0.22$
channel 1 $\sigma^2_{\text{instance}}$: $\frac{1}{3}(0.33^2+0.33^2+(-0.67)^2) = \frac{0.67}{3} = 0.22$

4.2 For the next two subquestions, we consider the following parameterization of a weight vector $\boldsymbol{w}$:

$$\boldsymbol{w} := \gamma \frac{\boldsymbol{u}}{||\boldsymbol{u}||}$$

where γ is scalar parameter controlling the magnitude and $\boldsymbol{u}$ is a vector controlling the direction of $\boldsymbol{w}$.

Consider one layer of a neural network, and omit the bias parameter. To carry out batch normalization, one normally standardizes the preactivation and performs elementwise scale and shift $\hat{y} = \gamma \cdot \frac{y-\mu_y}{\sigma_y} + \beta$ where $y = \boldsymbol{u}^\top \boldsymbol{x}$. Assume the data $\boldsymbol{x}$ (a random vector) is whitened ($\text{Var}(\boldsymbol{x}) = \boldsymbol{I}$) and centered at 0 ($\mathbb{E}[\boldsymbol{x}] = \boldsymbol{0}$). Show that $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + \beta$.

Starting by definitions:

$$\hat{y} = \gamma \cdot \frac{y - \mu_y}{\sigma_y} + \beta$$
$$y = \boldsymbol{u}^\top \boldsymbol{x}$$

We thus need to compute μ_y, σ_y:

$$\mu_y = \mathbb{E}[y] = \mathbb{E}[\boldsymbol{u}^\top \boldsymbol{x}] = \boldsymbol{u}^\top \mathbb{E}[\boldsymbol{x}] = 0$$

Last equality comes from the assumption of the problem ($\mathbb{E}[\boldsymbol{x}] = \boldsymbol{0}$) along with the fact that $\boldsymbol{u}$ is independent of $\boldsymbol{x}$ hence comes out of the expectation. Now for the variance:

$$\sigma_y^2 = \mathbb{E}[(\boldsymbol{y} - \mu_y)^2] \overset{\mu_y=0}{=} \mathbb{E}[(\boldsymbol{y})^2] = \mathbb{E}[(\boldsymbol{u}^\top \boldsymbol{x})^2]$$
$$\overset{A^2=A^\top A}{\longrightarrow} = \mathbb{E}((\boldsymbol{u}^\top \boldsymbol{x})^\top \boldsymbol{u}^\top \boldsymbol{x}) = \mathbb{E}((\boldsymbol{u}^\top \boldsymbol{x})^\top \boldsymbol{u}^\top \boldsymbol{x})$$

But $\boldsymbol{u}^\top \boldsymbol{x}$ is a scalar, so it's equal to its transpose $(\boldsymbol{u}^\top \boldsymbol{x})^\top$ and hence we can rewrite $(\boldsymbol{u}^\top \boldsymbol{x})^2 = \boldsymbol{u}^\top \boldsymbol{x}(\boldsymbol{u}^\top \boldsymbol{x})^\top$. Then we get:

$$\sigma_y^2 = \mathbb{E}[(\boldsymbol{u}^\top \boldsymbol{x})^2] = \mathbb{E}[\boldsymbol{u}^\top \boldsymbol{x}(\boldsymbol{u}^\top \boldsymbol{x})^\top] = \mathbb{E}[\boldsymbol{u}^\top \boldsymbol{x}\boldsymbol{x}^\top \boldsymbol{u}]$$

But note that $\text{Var}(\boldsymbol{x}) = \boldsymbol{I} = \mathbb{E}[(\boldsymbol{x} - \mathbb{E}[\boldsymbol{x}])^2]$, but since $\mathbb{E}[\boldsymbol{x}] = 0$ by assumption, then $\text{Var}(\boldsymbol{x}) = \boldsymbol{I} = \mathbb{E}[(\boldsymbol{x})^2] = \mathbb{E}[\boldsymbol{x}\boldsymbol{x}^\top]$. In the equation above, $\boldsymbol{u}$ is independent of expectation over $\boldsymbol{x}$ so it can be pulled out and we can plug $\text{Var}(\boldsymbol{x}) = \boldsymbol{I} = \mathbb{E}[\boldsymbol{x}\boldsymbol{x}^\top]$ into that to get:

$$\sigma_y^2 = \mathbb{E}[\boldsymbol{u}^\top \boldsymbol{x}\boldsymbol{x}^\top \boldsymbol{u}] = \boldsymbol{u}^\top \mathbb{E}[\boldsymbol{x}\boldsymbol{x}^\top]\boldsymbol{u} = \boldsymbol{u}^\top \boldsymbol{I}\boldsymbol{u} = \|\boldsymbol{u}\|_2^2$$

So we got $\mu_y = 0$ and $\sigma_y = \|\boldsymbol{u}\|_2$, plugging these back to the equation for $\hat{y}$:

$$\hat{y} = \gamma \cdot \frac{y - \mu_y}{\sigma_y} + \beta = \gamma \cdot \frac{y - 0}{\|\boldsymbol{u}\|_2} + \beta = \gamma \cdot \frac{\boldsymbol{u}^\top}{\|\boldsymbol{u}\|_2}\boldsymbol{x} + \beta$$

And since $\boldsymbol{w} = \gamma \frac{\boldsymbol{u}}{||\boldsymbol{u}||} \rightarrow \boldsymbol{w}^\top = \gamma \frac{\boldsymbol{u}^\top}{||\boldsymbol{u}||}$, hence we arrive at the following and the proof concludes:

$$\hat{y} = \gamma \cdot \frac{\boldsymbol{u}^\top}{\|\boldsymbol{u}\|_2}\boldsymbol{x} + \beta = \boldsymbol{w}^\top \boldsymbol{x} + \beta$$

4.3 Show that the gradient of a loss function $L(\boldsymbol{u}, \gamma, \beta)$ with respect to $\boldsymbol{u}$ can be written in the form $\nabla_{\boldsymbol{u}} L = s\boldsymbol{W}^{\perp}\nabla_{\boldsymbol{w}} L$ for some s, where $\boldsymbol{W}^{\perp} = \left(\boldsymbol{I} - \frac{\boldsymbol{u}\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|^2}\right)$. Note that [2] $\boldsymbol{W}^{\perp}\boldsymbol{u} = \boldsymbol{0}$.

The loss is probably a distance between the post-activation and some ground truth output y_g. But we don't know the exact details, but we can use the chain rule and factor those details in s:

$$\nabla_{\boldsymbol{u}} L(\boldsymbol{u}, \gamma, \beta) = \frac{\partial L(\boldsymbol{u}, \gamma, \beta)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{u}}$$

Now let's focus on the part we can compute $\frac{\partial \hat{y}}{\partial \boldsymbol{u}}$:

$$\begin{aligned}\frac{\partial \hat{y}}{\partial \boldsymbol{u}} &= \frac{\partial}{\partial \boldsymbol{u}}\left(\gamma \frac{\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|}\boldsymbol{x} + \beta\right) = \gamma\left(\frac{\partial}{\partial \boldsymbol{u}}\frac{\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|}\right)\boldsymbol{x} + \overbrace{\frac{\partial}{\partial \boldsymbol{u}}(\beta)}^{0} \\ &= \gamma\left(\frac{\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{u}}\|\boldsymbol{u}\| - \boldsymbol{u}\frac{\partial \|\boldsymbol{u}\|}{\partial \boldsymbol{u}}}{\|\boldsymbol{u}\|^2}\right)\boldsymbol{x} = \gamma\left(\frac{\boldsymbol{I}\|\boldsymbol{u}\| - \boldsymbol{u}\frac{\partial \|\boldsymbol{u}\|}{\partial \boldsymbol{u}}}{\|\boldsymbol{u}\|^2}\right)\boldsymbol{x}\end{aligned}$$

To compute $\frac{\partial \|\boldsymbol{u}\|}{\partial \boldsymbol{u}}$, note that $\|\boldsymbol{u}\| = \sqrt{\boldsymbol{u}^{\top}\boldsymbol{u}}$, then:

$$\frac{\partial \|\boldsymbol{u}\|}{\partial \boldsymbol{u}} = \frac{\partial (\boldsymbol{u}^{\top}\boldsymbol{u})^{1/2}}{\partial \boldsymbol{u}} = \frac{1}{2}(\boldsymbol{u}^{\top}\boldsymbol{u})^{-1/2}\frac{\partial (\boldsymbol{u}^{\top}\boldsymbol{u})}{\partial \boldsymbol{u}} = \frac{1}{2}(\boldsymbol{u}^{\top}\boldsymbol{u})^{-1/2} \times 2\boldsymbol{u}^{\top}$$

Inserting this result into the equation for $\frac{\partial \hat{y}}{\partial \boldsymbol{u}}$:

$$\begin{aligned}\frac{\partial \hat{y}}{\partial \boldsymbol{u}} &= \gamma\left(\frac{\boldsymbol{I}\|\boldsymbol{u}\| - \boldsymbol{u}\frac{\partial \|\boldsymbol{u}\|}{\partial \boldsymbol{u}}}{\|\boldsymbol{u}\|^2}\right)\boldsymbol{x} = \gamma\left(\frac{\boldsymbol{I}\|\boldsymbol{u}\| - \boldsymbol{u}(\boldsymbol{u}^{\top}\boldsymbol{u})^{-1/2}\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|^2}\right)\boldsymbol{x} = \gamma\left(\frac{\boldsymbol{I}\|\boldsymbol{u}\| - \boldsymbol{u}(\|\boldsymbol{u}\|^2)^{-1/2}\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|^2}\right)\boldsymbol{x} \\ &= \gamma\left(\frac{\boldsymbol{I}\|\boldsymbol{u}\|^2 - \boldsymbol{u}\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|^3}\right)\boldsymbol{x} = \left(\boldsymbol{I} - \frac{\boldsymbol{u}\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|^2}\right)\frac{\gamma \boldsymbol{x}}{\|\boldsymbol{u}\|}\end{aligned}$$

So finally we have:

$$\nabla_{\boldsymbol{u}} L(\boldsymbol{u}, \gamma, \beta) = \frac{\partial L(\boldsymbol{u}, \gamma, \beta)}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \boldsymbol{u}} = \frac{\partial L(\boldsymbol{u}, \gamma, \beta)}{\partial \hat{y}}\left(\boldsymbol{I} - \frac{\boldsymbol{u}\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|^2}\right)\frac{\gamma \boldsymbol{x}}{\|\boldsymbol{u}\|}$$

but $\nabla_{\boldsymbol{w}} L = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \boldsymbol{w}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial (\boldsymbol{w}^{\top}\boldsymbol{x} + \beta)}{\partial \boldsymbol{w}} = \frac{\partial L}{\partial \hat{y}}\boldsymbol{x}$, substituting this term in the equation for $\nabla_{\boldsymbol{u}} L(\boldsymbol{u}, \gamma, \beta)$ along with $\boldsymbol{W}^{\perp} = \left(\boldsymbol{I} - \frac{\boldsymbol{u}\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|^2}\right)$:

$$\nabla_{\boldsymbol{u}} L(\boldsymbol{u}, \gamma, \beta) = \frac{\partial L(\boldsymbol{u}, \gamma, \beta)}{\partial \hat{y}}\left(\boldsymbol{I} - \frac{\boldsymbol{u}\boldsymbol{u}^{\top}}{\|\boldsymbol{u}\|^2}\right)\frac{\gamma \boldsymbol{x}}{\|\boldsymbol{u}\|} = \frac{\gamma}{\|\boldsymbol{u}\|}\boldsymbol{W}^{\perp}\nabla_{\boldsymbol{w}} L$$

So if we set $s = \frac{\gamma}{\|\boldsymbol{u}\|}$, then we arrive at the desired result:

$$\nabla_{\boldsymbol{u}} L(\boldsymbol{u}, \gamma, \beta) = s\boldsymbol{W}^{\perp}\nabla_{\boldsymbol{w}} L$$

2. As a side note: $\boldsymbol{W}^{\perp}$ is an orthogonal complement that projects the gradient away from the direction of $\boldsymbol{w}$, which is usually (empirically) close to a dominant eigenvector of the covariance of the gradient. This helps to condition the landscape of the objective that we want to optimize.