Due Date: Nov 12th, 2024 at 11:00 pm

Instructions

- *For all questions, show your work!*
- *Use LaTeX and the template we provide when writing your answers. You may reuse most of the notation shorthands, equations and/or tables. See the assignment policy on the course website for more details.*
- *The use of AI tools like Chat-GPT to find answers or parts of answers for any question in this assignment is not allowed. However, you can use these tools to improve the quality of your writing, like fixing grammar or making it more understandable. If you do use these tools, you must clearly explain how you used them and which questions or parts of questions you applied them to. Failing to do so or using these tools to find answers or parts of answers may result in your work being completely rejected, which means you'll receive a score of 0 for the entire theory or practical section.*
- *Submit your answers electronically via Gradescope.*
- *TAs for this assignment are* ***Vaibhav Singh, Qian Yang, Mohsin Hasan***

Question 1 (4-2-4)**.** (**Optimization and Regularization**)

In this question, we will try to dive deep into constrained optimization. Imagine you are training a neural network (with learnable weights θ) with a loss function $L(f(\mathbf{x}^{(i)}, \theta), \mathbf{y}^{(i)})$. One way to constrain the learning algorithm is via *weight decay* scheme which uses a modified optimizer update rule: the weights θ decay by a factor of λ at each step. That is, the weights at iteration $i+1$ are computed as

$$\theta_{i+1} = \theta_i - \Delta\theta - \lambda\theta_i$$

where $\Delta\theta$ depends on each optimizer, since each optimizer, calculates the update differently. For SGD, its simply the product of learning rate and gradient of the loss function. For RMSprop and Adam, its based on moving averages.(Refer to the lecture slides)

Now consider *L2 regularization* scheme which modifies the loss function (while maintaining the typical optimizer for eg SGD or RMSprop update rules). The modified loss function is

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

1.1 Now you are given the statement - **L2 regularization scheme with typical optimizer update step is identical to scheme where weight decay is added to the update step of optimizer.** Does this statement hold true for Adam, SGD and RMSprop? Provide mathematical arguments.

Answer:

For SGD, the update step $\Delta\theta$ follows:

$$\Delta\theta = \alpha\frac{\partial L}{\partial \theta},$$

where α is the learning rate. Then we use L2 regularization, the loss is:

$$L_{\text{reg}} = L + \gamma \|\theta\|_2^2.$$

The gradient of L_{reg} with θ should be:

$$\frac{\partial L_{\text{reg}}}{\partial \theta} = \frac{\partial L}{\partial \theta} + 2\gamma\theta.$$

So the weight update with L2 regularization should be:

$$\theta_{i+1} = \theta_i - \alpha \left(\frac{\partial L}{\partial \theta} + 2\gamma\theta_i \right).$$

If the $\lambda = 2\gamma$, we can find the weight decay $\lambda\theta$ added to the update is equivalent to the L2 regularization. So, for SGD, L2 regularization is identical to weight decay with $\lambda = 2\gamma$.

For RMSprop, the update step $\Delta\theta$ is the moving average of squared gradients:

$$\Delta\theta = \frac{\alpha}{\sqrt{E[\nabla_\theta^2] + \epsilon}} \frac{\partial L}{\partial \theta}.$$

For L2 regularization, the loss function is different with SGD:

$$\frac{\partial L_{\text{reg}}}{\partial \theta} = \frac{\partial L}{\partial \theta} + 2\gamma\theta.$$

Then we can get the updates:

$$\theta_{i+1} = \theta_i - \frac{\alpha}{\sqrt{E[\nabla_\theta^2] + \epsilon}} \left(\frac{\partial L}{\partial \theta} + 2\gamma\theta_i \right).$$

If $\lambda = 2\gamma$, the update is equivalent to RMSprop with weight decay. So, it is correct for RMSprop with $\lambda = 2\gamma$.

For Adam, the update step contains moving averages and square, as following:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\frac{\partial L}{\partial \theta}, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2)\left(\frac{\partial L}{\partial \theta}\right)^2.$$

Then we update the parameter:

$$\Delta\theta = \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

The gradient term is $\frac{\partial L}{\partial \theta} + 2\gamma\theta$ for L2 regularization, we modify the first moment estimate:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\left(\frac{\partial L}{\partial \theta} + 2\gamma\theta \right).$$

So the weight decay term decay θ at each step, L2 regularization is not equivalent to weight decay for Adam.

Summary: L2 regularization is equivalent to weight decay in SGD and RMSprop, but not in Adam.

1.2 Now lets compare L1 regularisation, which can be defined as

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

Assume we trying to learn the simplest linear regression model with L1 regulariser as

$$\hat{\beta} = \underset{\beta}{\text{argmin}} \left\{ \frac{1}{2}\sum_{i=1}^{N}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j| \right\}$$

Similarly L2 regulariser would translate to

$$\hat{\beta} = \underset{\beta}{\text{argmin}} \left\{ \frac{1}{2}\sum_{i=1}^{N}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j|^2 \right\}$$

- For two parameters β_0, β_1, prove that the loss function above i.e $\frac{1}{2}\sum_{i=1}^{N}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2$ (Least Squared Error term), is an ellipse.

  Answer problem: First, the loss function is:

$$\frac{1}{2}\sum_{i=1}^{N}\left(y_i - \beta_0 - x_i\beta_1\right)^2.$$

  we transform it with matrix, $\mathbf{X}$ as the design matrix, where each row is $[1, x_i]$, $\beta = \begin{bmatrix}\beta_0\\ \beta_1\end{bmatrix}$, $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$.

  And the least squares error term is as following:

$$\frac{1}{2}\|\mathbf{y} - \mathbf{X}\beta\|_2^2.$$

  So this expression can be transformed as:

$$\frac{1}{2}\left(\mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\beta + \beta^T\mathbf{X}^T\mathbf{X}\beta\right),$$

where $\mathbf{X}$ is the design matrix, $\beta = \begin{bmatrix}\beta_0\\ \beta_1\end{bmatrix}$.

Finally, the term $\beta^T\mathbf{X}^T\mathbf{X}\beta$ is a quadratic form, representing an ellipse. The $\mathbf{X}^T\mathbf{X}$ is symmetric and positive semi-definite.

- Out of L1 and L2, which regularizer promotes sparsity in the network? Give informal reason, without going into the mathematical details. Given that we want to do a feature subset selection in our network, which regularizer should we use?

  Answer:

  L1 regularization produces sparsity, because L1 regularization puts some of the weights to zero, so that some features are removed. As for L2 regularization, which penalizes large weights but does not cause zero weights, it is different from L1. Therefore, if we want to select a smaller set of important features in the network, we should use L1 regularization. It helps by pushing some weights to zero, which effectively turns off certain features.

Question 2 (5-4-6-5)**.** (**Transformer**)

The output of Transformer's self-attention is computed as follows:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{1}$$

Where d_k is each key vector's dimension.

2.1 Assume that q and k are both d_k-dimensional vectors, and each component of q and k are independent variables with mean 0 and variance 1. **Prove that** $Var\left(\frac{q\cdot k}{\sqrt{d_k}}\right) = 1$**.** Additionally, explain why it is necessary to **scale the dot product** by $\sqrt{d_k}$ before applying softmax.

Answer: $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$, where each q_i and k_i are independent random variables with mean 0 and variance 1. So we can obtain that the dot product of $\mathbf{q}$ and $\mathbf{k}$ is: $\mathbf{q}\cdot\mathbf{k} = \sum_{i=1}^{d_k} q_i k_i$.

Then given that q_i and k_i are independent random variables with mean 0 and variance 1, their covariance is zero, so we can get the variance of their dot product as:

$$\text{Var}(\mathbf{q}\cdot\mathbf{k}) = \text{Var}\left(\sum_{i=1}^{d_k} q_i k_i\right) = \sum_{i=1}^{d_k}\text{Var}(q_i k_i)$$

Because the q_i and k_i are independent random variables, we know that:

$$(\mathbb{E}[q_i]\cdot\mathbb{E}[k_i])^2 = 0, \mathbb{E}[q_i^2] = 1, \quad \mathbb{E}[k_i^2] = 1$$

So,

$$\text{Var}(q_i k_i) = \mathbb{E}[q_i^2] \cdot \mathbb{E}[k_i^2] = 1 \times 1 = 1$$

Next, when calculating the variance of the entire dot product $\mathbf{q} \cdot \mathbf{k}$, since q_i and k_i are independent, we can sum up the variance of each term of the dot product:

$$\text{Var}(\mathbf{q} \cdot \mathbf{k}) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = \sum_{i=1}^{d_k} 1 = d_k$$

We use the $1/\sqrt{d_k}$ as the scaled, it can also be considered as:

$$\text{Var}\left(\frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}}\right) = \frac{1}{d_k}\text{Var}(\mathbf{q} \cdot \mathbf{k}) = \frac{1}{d_k} \times d_k = 1$$

So we can prove that:

$$\text{Var}\left(\frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}}\right) = 1$$

why it is necessary to scale the dot product by $\sqrt{d_k}$:

with the improvement of d_k, $(\mathbf{q} \cdot \mathbf{k})$ can increase gradually, so it is important to limit the value of $\sqrt{d_k}$ by scaling. This scaling ensures that the expected size of the dot product stays consistent as the dimensionality increases, which helps prevent the Softmax from becoming overly saturated or biased. So it keeps the gradients more stable and maintains the effectiveness of the attention mechanism.

In Transformer self-attention, different matrices are used to transform the representation of tokens into the $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$. Now, suppose $\boldsymbol{Q} = \boldsymbol{K}$. From the characteristics of matrix $\boldsymbol{Q}\boldsymbol{K}^\top$, specify **at least two consequences** of this.

We use multi-head attention in Transformer.

$$\begin{aligned}\text{MultiHead}(\bar{\boldsymbol{Q}}, \bar{\boldsymbol{K}}, \bar{\boldsymbol{V}}) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\boldsymbol{W}^O \\ \text{where} \quad \text{head}_i &= \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) \qquad (\text{here, } \boldsymbol{Q} := \bar{\boldsymbol{Q}}\boldsymbol{W}_i^Q, \boldsymbol{K} := \bar{\boldsymbol{K}}\boldsymbol{W}_i^K, \boldsymbol{V} := \bar{\boldsymbol{V}}\boldsymbol{W}_i^V)\end{aligned}$$

Assume the dimensionality of each head's $q/k/v$ projection is $d_k = \frac{d_{\text{model}}}{h}$. Given an input sequence of length n and model dimensionality d_{model}, explain the **time and space complexity** for computing the $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ matrices in multi-head attention when there are h heads. How does this **complexity scale** with respect to the number of heads h and the sequence length n?

Answer:

Time Complexity:

First, we need to calculate the $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ matrices for each attention head in multi-head attention, generally, the input shape of the input sequence is $n \times d_{\text{model}}$, which means there are n tokens in the d_{model} dimension sequence.

Then, we should compute complexity per head by the projection matrix, the dimension of each projection matrix is $d_{\text{model}} \times d_k$, where $d_k = \frac{d_{\text{model}}}{h}$, we multiply the input sequence with the projection matrix ($d_{\text{model}} \times d_k$). So the time complexity of matrix multiplication is $O(n \cdot d_{\text{model}} \cdot d_k)$.

As the $d_k = \frac{d_{\text{model}}}{h}$, $h \cdot d_k$ in the formula can be replaced by d_{model}, at last, the final time complexity can be considered as:

$$O(n \cdot d_{\text{model}}^2)$$

Space Complexity **problem**

Each head's $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ matrices has a space requirement of $O(n \cdot d_k)$, so the total space complexity across h head is $O(3 \cdot n \cdot d_k \cdot h) = O(n \cdot d_{\text{model}})$, which scales linearly with n and does not depend on h because of the fixed total dimensionality d_{model}. Therefore, increasing h does not affect the asymptotic time and space complexity, since each head operates on a reduced dimension d_k , keeping the overall dimensionality at d_{model} .

During inference, Key-Value Caching is used to improve time efficiency. In the case of Multi-Head Attention, where the input sequence has a length of n and each head's $q/k/v$ projection has a dimensionality of $d_k = \frac{d_{\text{model}}}{h}$, describe how caching affects **both time and space complexity**. Additionally, explain why this technique is critical for efficiently handling **long sequence generation**.

Answer:

Key-value caching can improve time efficiency in multi-head attention.

(1) for time complexity: With Key-Value Caching, we save $\boldsymbol{K}$ and $\boldsymbol{V}$ that were already computed. Then, when a new token comes in, we only need to calculate the $\boldsymbol{Q}$ for that token. After that, we use the cached keys and values to compute the attention. So, for each new token, the computation time becomes $O(n \cdot d_k)$, which is much faster because it's now linear with respect to the sequence length n.

(2) for space complexity: we need store $\boldsymbol{K}$ and $\boldsymbol{V}$ for the tokens we've processed. For each head, this takes up $O(n \cdot d_k)$ space. Since we have h heads, the total space required for caching all the keys and values is $O(h \cdot n \cdot d_k)$, which simplifies to $O(n \cdot d_{\text{model}})$, and the memory usage grows linearly with sequence length n.

(3) necessity: it helps us avoid having to process the whole sequence again every time we generate a new token. Instead of recalculating everything, we just reuse the previously computed keys and

values, which speeds things up. Besides, the computation grows in a linear way instead of an exponential one. This makes it much more manageable to deal with long sequences, both in terms of time and memory, since we don't have to do more and more work as the sequence gets longer.

Question 3 (5-5-5-5). (**Paper Review: An Image is Worth 16X16 Words.**)
In this question, you are going to write a **one page review** of the An Image is Worth 16X16 Words. Your review should have the following four sections: Summary, Strengths, Weaknesses, and Reflections. For each of these sections, below we provide a set of questions you should ask about the paper as you read it. Then, discuss your thoughts about these questions in your review.

(3.1) **Summary:**
- (a) What is this paper about ?
- (b) What is the main contribution ?
- (c) Describe the main approach and results. Just facts, no opinions yet.

(3.2) **Strengths:**
- (a) Is there a new theoretical insight ?
- (b) Or a significant empirical advance ? Did they solve a standing open problem ?
- (c) Or a good formulation for a new problem ?
- (d) Any good practical outcome (code, algorithm, etc) ?
- (e) Are the experiments well executed ?
- (f) Useful for the community in general ?

(3.3) **Weaknesses:**
- (a) What can be done better ?
- (b) Any missing baselines ? Missing datasets ?
- (c) Any odd design choices in the algorithm not explained well ? Quality of writing ?
- (d) Is there sufficient novelty in what they propose ? Minor variation of previous work ?
- (e) Why should anyone care ? Is the problem interesting and significant ?

(3.4) **Reflections:**
- (a) How does this relate to other concepts you have seen in the class ?
- (b) What are the next research directions in this line of work ?
- (c) What (directly or indirectly related) new ideas did this paper give you ? What would you be curious to try ?

This question is subjective and so we will accept a variety of answers. You are expected to analyze the paper and offer your own perspective and ideas, beyond what the paper itself discusses.

Question 4 (4-4-2-2). (**RNNs**)

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:

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

(4.1) Write an expression for the gradient $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_0}$.

Answer: First, follow the chain rule to backpropagate the gradient:

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_0} = \prod_{i=1}^{t} \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}} = \prod_{i=1}^{t} \sigma'(\boldsymbol{W}\boldsymbol{h}_{i-1} + \boldsymbol{U}\boldsymbol{x}_i + \boldsymbol{b}) \cdot \boldsymbol{W}$$

where $\sigma'(x)$ is the derivative of the activation function σ evaluated at x. The product of the $\boldsymbol{W}$ matrices across all the time steps shows how earlier states affect the current state on $\boldsymbol{h}_t$.

(4.2) Let $||\mathbf{A}||$ denote the L_2 operator norm of matrix $\mathbf{A}$ ($||\mathbf{A}|| := \max_{\boldsymbol{x}:||\boldsymbol{x}||=1} ||\mathbf{A}\boldsymbol{x}||$). The following property of the operator norm is useful:

$$||\mathbf{AB}|| \leq ||\mathbf{A}||\,||\mathbf{B}||$$

Assume $\sigma(x)$ has bounded derivative, i.e. $|\sigma'(x)| \leq \gamma$ for some $\gamma > 0$ and for all x.

Suppose the weight matrix $\boldsymbol{W}$ is **orthogonal**, ie. $\boldsymbol{W}^\top \boldsymbol{W} = \boldsymbol{W}\boldsymbol{W}^\top = \mathbf{I}$. Such matrices also satisfy $||\boldsymbol{W}\boldsymbol{x}|| = ||\boldsymbol{x}||$ for all vectors $\boldsymbol{x}$.

What condition on γ ensures the gradient vanishes ($||\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_0}|| \to 0$) as t grows ?

Answer:

Since $\boldsymbol{W}$ is orthogonal, we can know that for any vector $\boldsymbol{x}$, we have $||\boldsymbol{W}\boldsymbol{x}|| = ||\boldsymbol{x}||$, which basically tells us that $\boldsymbol{W}$ doesn't change the length of the vector $\boldsymbol{x}$. Now, when we compute the gradient of $\boldsymbol{h}_t$ with respect to $\boldsymbol{h}_0$, we see that it depends on the chain of derivatives involving σ'. For the gradient to become small as t goes to infinity, we want the terms involving $\sigma'(x)$ to shrink.

Then, we can assume that $|\sigma'(x)| \leq \gamma$. If $\gamma < 1$, then the gradient will finally go to zero because:

$$||\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_0}|| \leq \gamma^t ||\boldsymbol{W}^t|| = \gamma^t.$$

Finally, to ensure the gradient goes to zero as t grows, we need γ to be less than 1.

(4.3) Suppose this same condition on γ holds, but $\boldsymbol{W}$ is not orthogonal. Is the gradient $||\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_0}||$ guaranteed to vanish ?

Answer:

If $\boldsymbol{W}$ is not orthogonal, then the norm of the gradient might grow instead of shrink. It is because $\boldsymbol{W}$ doesn't necessarily preserve the length of the vectors, so it could stretch or shrink the vectors in ways that cause the gradient to get bigger. So the gradient might not go to zero as t increases.

The growth of $||\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_0}||$ depends on the eigenvalues of $\boldsymbol{W}$. If $\boldsymbol{W}$ has eigenvalues greater than 1, the gradient might grow rather than shrink.

(4.4) In the case of an orthogonal $\boldsymbol{W}$, why may a ReLU activation, $\sigma(x) = \max(x, 0)$, help training ?

Answer: It is easy to understand because ReLU can prevent the vanishing gradient problem. The derivative of the ReLU is either 0 or 1 with negative or positive inputs, so ReLU does not make the gradient very small, avoiding the problem of gradient disappearance (compared to Sigmoid) and helping to transfer the gradient more efficiently during backpropagation, thereby speeding up the training process.

Question 5 (4-6)**.** (**Normalization**) This question is about normalization techniques.

(5.1) 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 $\boldsymbol{\sigma}^2_{instance}$.

$$\left[\begin{bmatrix}5,3,3\\1,1,0\end{bmatrix},\begin{bmatrix}0,3,0\\2,2,2\end{bmatrix},\begin{bmatrix}5,5,1\\3,4,3\end{bmatrix},\begin{bmatrix}1,1,2\\0,2,1\end{bmatrix}\right]$$

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. When calculating the variance, use the (biased) estimator $\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2$.

Answer:

First, we should calculate the formulas for mean and variance:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

Where n is the number of elements for the mean and variance.

Then, we should calculate the mean and variance for the batch:

$$\boldsymbol{\mu}_{batch} = \frac{1}{4\times 2\times 3}\sum_{i=1}^{4\times 2\times 3} x_i$$

$$\boldsymbol{\sigma}^2_{batch} = \frac{1}{4\times 2\times 3}\sum_{i=1}^{4\times 2\times 3}(x_i - \boldsymbol{\mu}_{batch})^2$$

Next, calculate the mean and variance for each feature across all instances in channel:

$$\boldsymbol{\mu}_{layer} = \frac{1}{4 \times 3} \sum_{i=1}^{4\times3} x_i$$

$$\boldsymbol{\sigma}^2_{layer} = \frac{1}{4 \times 3} \sum_{i=1}^{4\times3} (x_i - \boldsymbol{\mu}_{layer})^2$$

After that, calculate them across each feature for each instance in the batch:

$$\boldsymbol{\mu}_{instance} = \frac{1}{2 \times 3} \sum_{i=1}^{2\times3} x_i$$

$$\boldsymbol{\sigma}^2_{instance} = \frac{1}{2 \times 3} \sum_{i=1}^{2\times3} (x_i - \boldsymbol{\mu}_{instance})^2$$

(5.2) For this question, 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 $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + \beta$.

Answer:

First, we calculate μ_y and σ_y, because $\boldsymbol{x}$ is centered, so it can be considered as:

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

Next, we calculate the standard deviation σ_y:

$$\sigma_y = \sqrt{\text{Var}(\boldsymbol{u}^\top \boldsymbol{x})} = \sqrt{\boldsymbol{u}^\top \text{Var}(\boldsymbol{x}) \boldsymbol{u}} = \sqrt{\boldsymbol{u}^\top \mathbf{I} \boldsymbol{u}} = \sqrt{||\boldsymbol{u}||^2} = ||\boldsymbol{u}||$$

Then, substitute $\mu_y = 0$ and $\sigma_y = ||\boldsymbol{u}||$ to the batch normalization:

$$\hat{y} = \gamma \cdot \frac{y}{||\boldsymbol{u}||} + \beta$$

And $y = \boldsymbol{u}^\top \boldsymbol{x}$, so we can obtain that: $\hat{y} = \gamma \cdot \frac{\boldsymbol{u}^\top \boldsymbol{x}}{||\boldsymbol{u}||} + \beta$

Because $\boldsymbol{w} = \gamma \frac{\boldsymbol{u}}{||\boldsymbol{u}||}$, so $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + \beta$

Finally, we can get that :

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + \beta$$