EECS 182 Deep Neural Networks
Fall 2023 Anant Sahai

Homework 8

This homework is due on October 21, at 10:59PM.

1. Backprop through a Simple RNN

Consider the following 1D RNN with no nonlinearities, a 1D hidden state, and 1D inputs u_t at each timestep. (Note: There is only a single parameter w, no bias). This RNN expresses unrolling the following recurrence relation, with hidden state h_t at unrolling step t given by:

$$h_t = w \cdot (u_t + h_{t-1}) \tag{1}$$

The computational graph of unrolling the RNN for three timesteps is shown below:

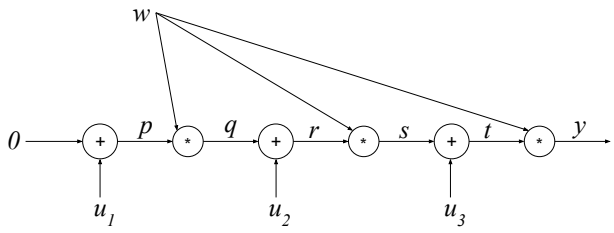


Figure 1: Illustrating the weight-sharing and intermediate results in the RNN.

where w is the learnable weight, u_1, u_2, and u_3 are sequential inputs, and p, q, r, s, and t are intermediate values.

(a) **Fill in the blanks for the intermediate values during the forward pass, in terms of w and the u_i's:**

$$p = u_1 \qquad q = w \cdot u_1 \qquad r = u_2 + q = u_2 + w \cdot u_1$$

$$s = w \cdot r = w \cdot u_2 + w^2 \cdot u_1$$

$t =$ _____

Solution: $t = u_3 + s = u_3 + w \cdot u_2 + w^2 \cdot u_1$

$y =$ _____

Solution: $y = w \cdot t = w \cdot u_3 + w^2 \cdot u_2 + w^3 \cdot u_1$

(b) **Using the expression for y from the previous subpart, compute $\frac{dy}{dw}$.**

Solution: $\frac{dy}{dw} = u_3 + 2wu_2 + 3w^2u_1$

(c) **Fill in the blank for the missing partial derivative of y with respect to the nodes on the backward pass.** You may use values for p, q, r, s, t, y computed in the forward pass and downstream derivatives already computed.

$$\frac{\partial y}{\partial t} = w \qquad \frac{\partial y}{\partial s} = w \qquad \frac{\partial y}{\partial r} = \frac{\partial y}{\partial s} \cdot w \qquad \frac{\partial y}{\partial q} = \frac{\partial y}{\partial r} \cdot 1$$

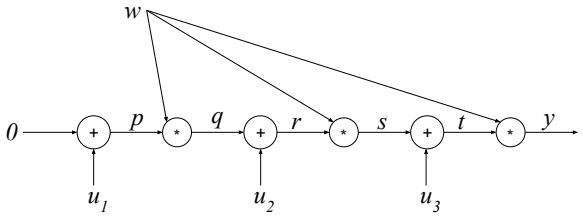
$$\frac{\partial y}{\partial p} = \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

Solution: $\frac{\partial y}{\partial p} = \frac{\partial y}{\partial q} \cdot w = w \cdot w \cdot w = w^3$

(d) **Calculate the partial derivatives along each of the three outgoing edges from the learnable w in Figure 1, replicated below.** (e.g., the right-most edge has a relevant partial derivative of t in terms of how much the output y is effected by a small change in w as it influences y through this edge. You need to compute the partial derivatives for the other two edges yourself.)

You can write your answers in terms of the p, q, r, s, t and the partial derivatives of y with respect to them.

Use these three terms to find the total derivative $\frac{dy}{dw}$.



(HINT: You can use your answer to part (b) to check your work.)

Solution: Along the right edge, we have $t = u_3 + wu_2 + w^2u_1$ (This is provided for you).
Along the middle edge, we have $r \cdot \frac{\partial y}{\partial s} = r \cdot w = (u_2 + wu_1)w = wu_2 + w^2u_1$
Along the left edge, we have $p \cdot \frac{\partial y}{\partial q} = p \cdot w^2 = u_1w^2$
Adding all of these up, we have

$$\frac{dy}{dw} = t + r \cdot \frac{\partial y}{\partial s} + p \cdot \frac{\partial y}{\partial q} \tag{2}$$

$$= (u_3 + wu_2 + w^2u_1) + (u_2 + wu_1)w + w^2u_1 \tag{3}$$

$$= 3w^2u_1 + 2wu_2 + u_3 \tag{4}$$

Which is same as answer to (b) so everything checks out.

2. Implementing RNNs (and optionally, LSTMs)

This problem involves filling out this notebook.
Note that implementing the LSTM portion of this question is optional and out-of-scope for the exam.

(a) **Implement Section 1A in the notebook**, which constructs a vanilla RNN layer. This layer implements the function

$$h_t = \sigma(W^h h_{t-1} + W^x x_t + b)$$

where W^h, W^x, and b are learned parameter matrices, x is the input sequence, and σ is a nonlinearity such as tanh. The RNN layer "unrolls" across a sequence, passing a hidden state between timesteps and returning an array of hidden states at all timesteps.

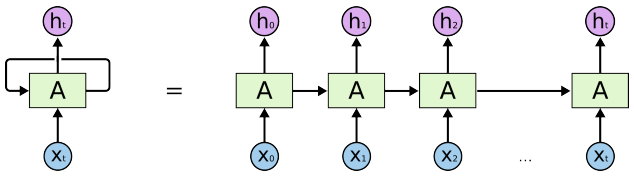


Figure 2: Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Copy the outputs of the *"Test Cases"* code cell and paste it into your submission of the written assignment.

Solution: See the solution notebook. Each max error should be less than `1e-4`.

(b) **Implement Section 1.B of the notebook**, in which you'll use this RNN layer in a regression model by adding a final linear layer on top of the RNN outputs.

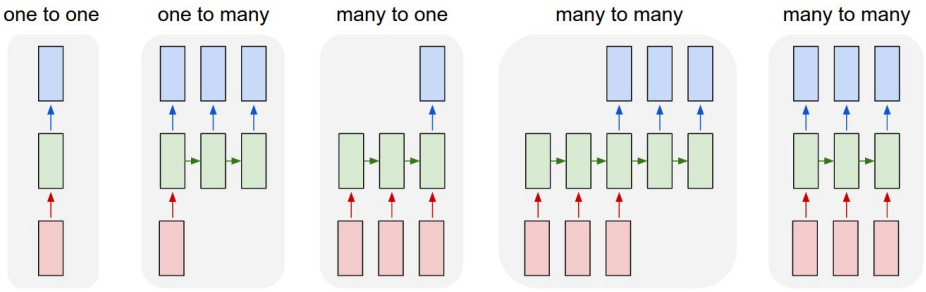
$$\hat{y}_t = W^f h_t + b^f$$

We'll compute one prediction for each timestep.

Copy the outputs of the *"Tests"* code cell and paste it into your submission of the written assignment.

Solution: See the solution notebook. Each max error should be less than `1e-4`.

(c) RNNs can be used for many kinds of prediction problems, as shown below. In this notebook we will look at many-to-one prediction and aligned many-to-many prediction.



We will use a simple averaging task. The input X consists of a sequence of numbers, and the label y is a running average of all numbers seen so far.

We will consider two tasks with this dataset:

- Task 1: predict the running average at all timesteps
- Task 2: predict the average at the last timestep only

Implement Section 1.C in the notebook, in which you'll look at the synthetic dataset shown and implement a loss function for the two problem variants.

Copy the outputs of the *"Tests"* code cell and paste it into your submission of the written assignment.

Solution: See the solution notebook. Each max error should be less than `1e-4`.

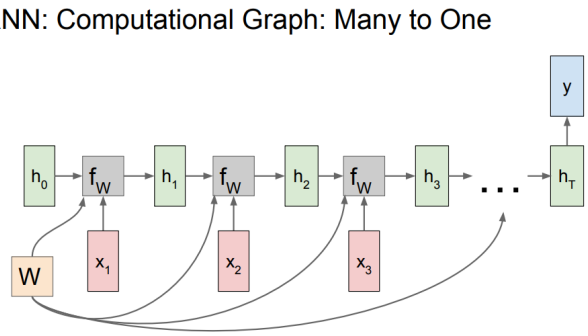


Figure 3: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

(d) Consider an RNN which outputs a single prediction at timestep T. As shown in Figure 3, each weight matrix W influences the loss by multiple paths. As a result, the gradient is also summed over multiple paths:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial h_T}\frac{\partial h_T}{\partial W} + \frac{\partial \mathcal{L}}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial W} + \ldots + \frac{\partial \mathcal{L}}{\partial h_1}\frac{\partial h_1}{\partial W} \tag{5}$$

When you backpropagate a loss through many timesteps, the later terms in this sum often end up with either very small or very large magnitude - called vanishing or exploding gradients respectively. Either problem can make learning with long sequences difficult.

Implement Notebook Section 1.D, which plots the magnitude at each timestep of $\frac{\partial \mathcal{L}}{\partial h_t}$. Play around with this visualization tool and try to generate exploding and vanishing gradients.

Include a screenshot of your visualization in the written assignment submission.

Solution: See the solution notebook.

(e) **If the network has no nonlinearities, under what conditions would you expect the exploding or vanishing gradients with for long sequences? Why?** (Hint: it might be helpful to write out the formula for $\frac{\partial \mathcal{L}}{\partial h_t}$ and analyze how this changes with different t). **Do you see this pattern empirically using the visualization tool in Section 1.D in the notebook** with last_step_only=True?

Solution: If we use the MSE loss on a single example (x, y), the gradient $\frac{\partial \mathcal{L}}{\partial h_t} = 2(\hat{y}-y)W^f(W^h)^{T-t}$. (To clarify, the exponents f and h are matrix indicators, but $T-t$ is an exponent.) If the magnitude of the largest eigenvalue of W^h is much greater than 1, the gradient will explode, and if it's much less than 1, the gradient will start to vanish. Gradients are only stable when the largest eigenvalue magnitude is close to 1.

We see the expected pattern empirically.

(f) Compare the magnitude of hidden states and gradients when using ReLU and tanh nonlinearities in Section 1.D in the notebook. **Which activation results in more vanishing and exploding gradients? Why?** (This does not have to be a rigorous mathematical explanation.)

Solution: Hidden states: tanh restricts hidden state values to (-1, 1), so hidden state magnitudes remain small. With ReLU, hidden state values can easily explode.

Gradients: When tanh inputs are large, gradients are close to zero. This results in fewer exploding gradients but more vanishing gradients. Exploding gradients are still possible, however, when the largest eigenvalue of W_h has magnitude > 1, but the hidden states remain close to zero. ReLU activations, in contrast, result in frequent exploding hidden state sizes and gradients, similar to in the no-activation RNN.

(g) **What happens if you set last_target_only = False in Section 1.D in the notebook? Explain why this change affects vanishing gradients. Does it help the network's ability to learn dependencies across long sequences?** (The explanation can be intuitive, not mathematically rigorous.)

Solution: For every timestep k, the model's prediction produces loss $\mathcal{L}_k$. The gradient $\partial L_k/\partial h_k$ is high-magnitude, resulting in high-magnitude logged gradients in the visualization tool, but for any timestep $t << k$, $\partial L_k/\partial h_t$ will still be small. This means the network will still struggle to pass gradients over long sequences, making it hard to learn long-range dependencies.

(h) (Optional) **Implement Section 1.8 of the notebook** in which you implement a LSTM layer. LSTMs pass a cell state between timesteps as well as a hidden state. **Explore gradient magnitudes using the visualization tool you implemented earlier and report on the results.** **Solution:** Hidden state outputs remain between +/- 1. Gradients don't explode, but they still vanish. Typically, they don't vanish as fast as vanilla RNNs.

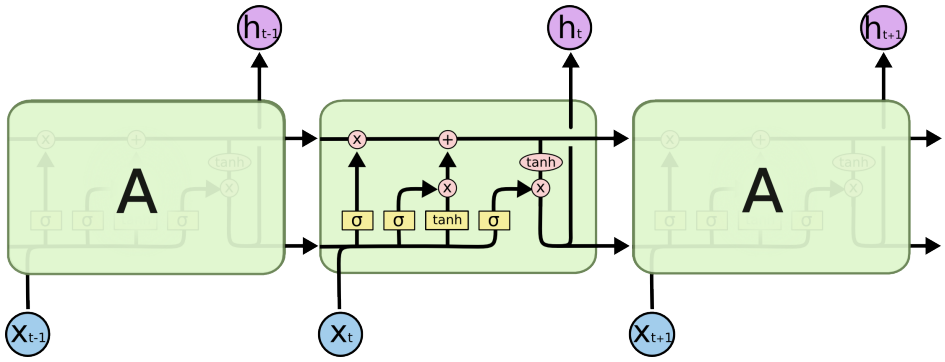


Figure 4: Image source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

The LSTM forward pass is shown below:

$$
\begin{aligned}
f_t &= \sigma(x_t U^f + h_{t-1} W^f + b^f) \\
i_t &= \sigma(x_t U^i + h_{t-1} W^i + b^i) \\
o_t &= \sigma(x_t U^o + h_{t-1} W^o + b^o) \\
\tilde{C}_t &= \tanh(x_t U^g + h_{t-1} W^g + b^g) \\
C_t &= f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \\
h_t &= \tanh(C_t) \circ o_t
\end{aligned}
$$

where $\circ$ represents the Hadamard Product (elementwise multiplication) and σ is the sigmoid function.

(i) (Optional) When using an LSTM, you should still see vanishing gradients, but the gradients should vanish less quickly. **Interpret why this might happen by considering gradients of the loss with respect to the cell state.** (Hint: consider computing $\frac{\partial \mathcal{L}}{\partial C_{T-1}}$ using the terms $\partial \mathcal{L}, \partial C_T, \partial C_{T-1}, \partial h_T, \partial h_{T-1}$).

Solution: In an LSTM, information can be stored in the cell state without needing to persist through a chain of matrix multiplications.

$$
\frac{\partial \mathcal{L}}{\partial C_{T-1}} = \frac{\partial \mathcal{L}}{\partial h_T}\left(\frac{\partial h_T}{\partial C_T}\frac{\partial C_T}{\partial C_{T-1}} + \frac{\partial h_T}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial C_{T-1}}\right)
$$

Unlike $\frac{\partial h_T}{\partial h_{T-1}}$, which results in decaying gradients due to matrix multiplication, $\frac{\partial C_T}{\partial C_{T-1}}$ is a diagonal matrix with f_t on the diagonal. If the network sets f_t close to 1, then $\frac{\partial C_T}{\partial C_{T-1}}$ will be close to the identity, allowing information stored in the cell state to "pass through" without decay. In practice, however, many elements of f_t are not close to 1, which results in some gradient decay.

In addition, LSTMs also contain a hidden state in which to store information, and this "pathway" through the network will decay like in a vanilla RNN, potentially resulting in some overall gradient decay.

(j) (Optional)

Consider a ResNet with simple resblocks defined by $h_{t+1} = \sigma(W_t h_t + b_t) + h_t$. **Draw a connection between the role of a ResNet's skip connections and the LSTM's cell state in facilitating gradient propagation through the network.**

Solution: ResNet skip connections and LSTM cell states both allow information to pass through the network without requiring a matrix multiplication at each layer. As a result, they both mitigate vanishing gradients.

(k) (Optional) We can create multi-layer recurrent networks by stacking layers as shown in Figure 5. The hidden state outputs from one layer become the inputs to the layer above.

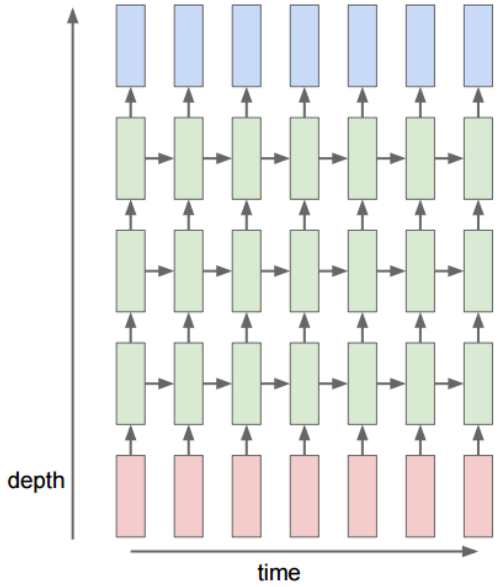


Figure 5: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

Implement notebook Section 1.K and run the last cell to train your network. You should be able to reach training loss < 0.001 for the 2-layer networks, and <.01 for the 1-layer networks.

3. RNNs for Last Name Classification

Please follow the instructions in this notebook. You will train a neural network to predict the probable language of origin for a given last name / family name in Latin alphabets.

(a) Although the neural network you have trained is intended to predict the language of origin for a given last name, it could potentially be misused. **In what ways do you think this could be problematic in real-world applications**?

Solution: The model's predictions could be used to make assumptions about a person's nationality or ethnicity, which could lead to discrimination or bias. It's important to note that the model should only be used to make predictions about the language of origin for a given name and not to make assumptions about a person's identity. The model could also be used to infer personal information about individuals, such as their cultural background, which could be considered an invasion of privacy.

4. Auto-encoder : Learning without Labels

So far, with supervised learning algorithms we have used labelled datasets $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ to learn a mapping f_θ from input x to labels y. In this problem, we now consider algorithms for *unsupervised learning*, where we are given only inputs x, but no labels y. At a high-level, unsupervised learning algorithms leverage some structure in the dataset to define proxy tasks, and learn *representations* that are useful for solving downstream tasks.

Autoencoders present a family of such algorithms, where we consider the problem of learning a function $f_\theta : \mathbb{R}^m \to \mathbb{R}^k$ from input x to a *intermediate representation* z of x (usually $k \ll m$). For autoencoders, we use reconstructing the original signal as a proxy task, i.e. representations are more likely to be useful for downstream tasks if they are low-dimensional but encode sufficient information to approximately reconstruct the input. Broadly, autoencoder architectures have two modules:

- An encoder $f_\theta : \mathbb{R}^m \to \mathbb{R}^k$ that maps input x to a representation z.
- A decoder $g_\phi : \mathbb{R}^k \to \mathbb{R}^m$ that maps representation z to a reconstruction $\hat{x}$ of x.

In such architectures, the parameters (θ, ϕ) are learnable, and trained with the learning objective of minimizing the reconstruction error $\ell(\hat{x}_i, x_i) = \|x - \hat{x}\|_2^2$ using gradient descent.

$$\theta_{\text{enc}}, \phi_{\text{dec}} = \underset{\Theta,\Phi}{\text{argmin}} \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{x}_i, x_i)$$
$$\hat{x} = g_\phi(f_\theta(x))$$

Note that above optimization problem does not require labels $\mathbf{y}$. In practice however, we would want to use the *pretrained* models to solve the *downstream* task at hand, e.g. classifying MNIST digits. *Linear Probing* is one such approach, where we fix the encoder weights, and learn a simple linear classifier over the features $z = \text{encoder}(x)$.

(a) Designing AutoEncoders

Please follow the instructions in this notebook. You will train autoencoders, denoising autoencoders, and masked autoencoders on a synthetic dataset and the MNIST dataset. Once you finished with the notebook,

- Answer the following questions in your submission of the written assignment:

(i) **Show your visualization** of the vanilla autoencoder with different latent representation sizes.

Solution:
See Figure 6. Please refer to the solution notebook for the codes.

(ii) Based on your previous visualizations, answer this question: **How does changing the latent representation size of the autoencoder affect the model's performance in terms of reconstruction accuracy and linear probe accuracy? Why?**

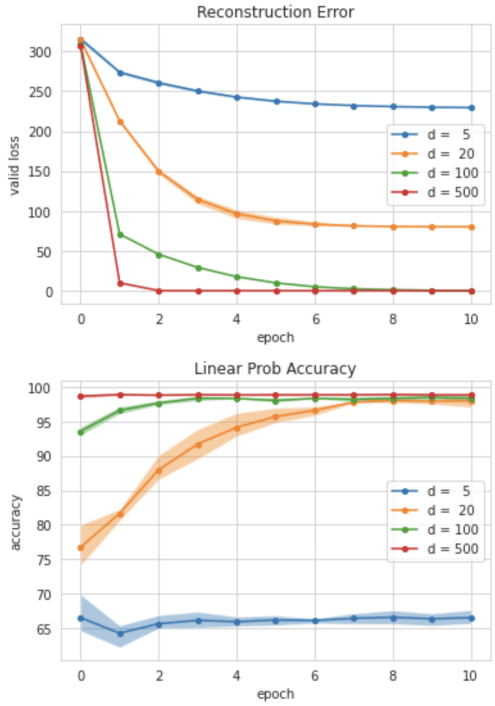


Figure 6: Visualization of the vanilla autoencoder with different latent representation sizes.

Solution: Based on the given synthetic dataset, each data point has 100 dimensions, with 20 high-variance dimensions affecting the class label. The following observations can be made from the visualizations in Figure 6.
Firstly, the reconstruction error of the autoencoder decreases as the size of the latent representation increases. However, this reduction becomes marginal when the size of the latent representation exceeds the dimension of the data (100).
Secondly, the linear probe accuracy increases as the size of the latent representation increases. When the size of the latent representation exceeds the dimension of the data (100), the linear probe accuracy approaches $\sim$100% even without training the autoencoder. However, when the size of the latent representation equals the number of interpretive dimensions (20), training the autoencoder becomes essential for achieving a high linear probe accuracy. The linear probe accuracy finally converges to $\geq$95% with training. On the other hand, if the size of the latent representation is as small as 5, it fails to capture all useful information in the input data, leading to significantly lower linear probe accuracy.

(b) PCA & AutoEncoders

In the case where the encoder f_θ, g_ϕ are linear functions, the model is termed as a *linear autoencoder*. In particular, assume that we have data $x_i \in \mathbb{R}^m$ and consider two weight matrices: an encoder $W_1 \in \mathbb{R}^{k\times m}$ and decoder $W_2 \in \mathbb{R}^{m\times k}$ (with $k < m$). Then, a linear autoencoder learns a low-dimensional embedding of the data $\mathbf{X} \in \mathbb{R}^{m\times n}$ (which we assume is zero-centered without loss of generality) by minimizing the objective,

$$\mathcal{L}(W_1, W_2; \mathbf{X}) = ||\mathbf{X} - W_2 W_1 \mathbf{X}||_F^2 \tag{6}$$

We will assume $\sigma_1^2 > \cdots > \sigma_k^2 > 0$ are the k largest eigenvalues of $\mathbf{X}\mathbf{X}^\top$. The assumption that the $\sigma_1, \ldots, \sigma_k$ are positive and distinct ensures identifiability of the principal components, and is common in this setting. Therefore, the top-k eigenvalues of $\mathbf{X}$ are $S = \mathrm{diag}(\sigma_1, \ldots, \sigma_k)$, with corresponding

eigenvectors are the columns of $\mathbf{U}_k \in \mathbb{R}^{m \times k}$. A well-established result from [1] shows that principal components are the unique optimal solution to linear autoencoders (up to sign changes to the projection directions). In this subpart, we take some steps towards proving this result.

(i) **Write out the first order optimality conditions that the minima of Eq. 6 would satisfy.**

Solution: We can compute the first order conditions for W_1 and W_2, respectively.
Important equations:

$$\|A\|_F^2 = tr(AA^T), \quad tr(ABC) = tr(BCA) = tr(CAB), \quad tr(A) = tr(A^T)$$

$$\nabla_A tr(AB) = B^T, \quad \nabla_B tr(AB) = A^T$$

Also, taking the matrix derivative of trace follows the product rule $(uv)' = u'v + uv'$. So for example (assuming all the matrices are of the right shape):

$$\begin{aligned}
\nabla_A tr(ABBA) &= \nabla_A tr(ABBC)|_{C=A} + \nabla_A tr(CBBA)|_{C=A} && \text{product rule of derivatives} \\
&= \nabla_A tr(ABBC)|_{C=A} + \nabla_A tr(ACBB)|_{C=A} && \text{cyclic rule} \\
&= (BBA)^T + (ABB)^T
\end{aligned}$$

Now just do the calculation.

$$\begin{aligned}
\nabla_{W_2} L &= \nabla_{W_2} tr(XX^T + W_2 W_1 XX^T W_1^T W_2^T - 2W_2 W_1 XX^T) \\
&= \cancel{\nabla_{W_2} tr(XX^T)} + \nabla_{W_2} tr(W_2 W_1 XX^T W_1^T W_2^T) - \nabla_{W_2} 2tr(W_2 W_1 XX^T) && \text{linearity} \\
&= 2(W_1 XX^T W_1^T W_2^T)^T + 2(W_1 XX^T)^T \\
&= 2(W_2 W_1 - I) XX^T W_1^T
\end{aligned}$$

Similarly calculate the other one:

$$\nabla_{W_1} L = 2W_2^T (W_2 W_1 - I) XX^T$$

(ii) **Show that the principal components $\mathbf{U}_k$ satisfy the optimality conditions outlined in (i).**

Solution: Do the SVD decomposition: $XX^T = U\Sigma V$, and $D = \Sigma\Sigma^T$. Note that D is a diagonal matrix.
Then the first order condition states that

$$W_2^T (W_2 W_1 - I) UDU^T = 0; \quad (W_2 W_1 - I) UDU^T W_1^T = 0$$

Let $V_1 := W_1 U, V_2 := U^T W_2$, then it further simplifies to

$$(V_2 V_1 - I) DV_1^T = 0; \quad V_2^T (V_2 V_1 - I) D = 0$$

If $W_1^T = W_2$ are the first k columns of U, then

[1] Baldi, Pierre, and Kurt Hornik. "Neural networks and principal component analysis: Learning from examples without local minima" (1989)

$$V_1 = \left[\begin{array}{ccc|c} 1 & & & \\ & \ddots & & 0 \\ & & 1 & \end{array}\right] \tag{7}$$

and $V_2 = V_1^T$. Now direct computation finishes the proof.

5. Self-supervised Linear Autoencoders

We consider linear models consisting of two weight matrices: an encoder $W_1 \in \mathbb{R}^{k\times m}$ and decoder $W_2 \in \mathbb{R}^{m\times k}$ (assume $1 < k < m$). The traditional autoencoder model learns a low-dimensional embedding of the n points of training data $\mathbf{X} \in \mathbb{R}^{m\times n}$ by minimizing the objective,

$$\mathcal{L}(W_1, W_2; \mathbf{X}) = \frac{1}{n}||\mathbf{X} - W_2W_1\mathbf{X}||_F^2 \tag{8}$$

We will assume $\sigma_1^2 > \cdots > \sigma_k^2 > \sigma_{k+1}^2 \geq 0$ are the $k+1$ largest eigenvalues of $\frac{1}{n}\mathbf{X}\mathbf{X}^\top$. The assumption that the $\sigma_1, \ldots, \sigma_k$ are positive and distinct ensures identifiability of the principal components.

Consider an ℓ_2-regularized linear autoencoder where the objective is:

$$\mathcal{L}_\lambda(W_1, W_2; \mathbf{X}) = \tfrac{1}{n}||\mathbf{X} - W_2W_1\mathbf{X}||_F^2 + \lambda\|W_1\|_F^2 + \lambda\|W_2\|_F^2. \tag{9}$$

where $\|\cdot\|_F^2$ represents the Frobenius norm squared of the matrix (i.e. sum of squares of the entries).

(a) You want to use SGD-style training in PyTorch (involving the training points one at a time) and self-supervision to find W_1 and W_2 which optimize (9) by treating the problem as a neural net being trained in a supervised fashion. **Answer the following questions and briefly explain your choice:**

(i) **How many linear layers do you need?**

- ☐ 0
- ☐ 1
- ☐ 2
- ☐ 3

Solution: **2**, we would use two linear layers, one for the encoder, one for the decoder.

(ii) **What is the loss function that you will be using?**

- ☐ `nn.L1Loss`
- ☐ `nn.MSELoss`
- ☐ `nn.CrossEntropyLoss`

Solution: We should use **MSE-Loss** to train the model (reconstruction under l2-loss) since what we want is for each vector to be close to its reconstruction in a squared-error sense.

(iii) **Which of the following would you need to optimize (9) exactly as it is written? (Select all that are needed)**

- ☐ Weight Decay
- ☐ Dropout
- ☐ Layer Norm
- ☐ Batch Norm
- ☐ SGD optimizer

Solution: We need to use **Weight Decay** to achieve the desired regularization and the problem states that we are doing "SGD-Style", training. Therefore we use the **SGD Optimizer**.

(b) **Do you think that the solution to (9) when we use a small nonzero λ has an inductive bias towards finding a W_2 matrix with approximately orthonormal columns? Argue why or why not?**
(Hint: Think about the SVDs of $W_1 = U_1\Sigma_1 V_1^\top$ and $W_2 = U_2\Sigma_2 V_2^\top$. You can assume that if a $k \times m$ or $m \times k$ matrix has all k of its nonzero singular values being 1, then it must have orthonormal rows or columns. Remember that the Frobenius norm squared of a matrix is just the sum of the squares of its singular values. Further think about the minimizer of $\frac{1}{\sigma^2} + \sigma^2$. Is it unique?)

Solution: If there were no regularization terms, we know that all the optimizers have to have W_2W_1 acting like a projection matrix that projects onto the k largest singular vectors of X.

This means that the W_2W_1 to minimize the main loss has to be the identity when restricted to the subspace spanned by the k largest singular vectors of X.

Therefore we would expect W_1, W_2 be approximate psuedo-inverses of each other since they are not square, and the rank of either one is at most k.

Therefore regularizing by penalizing the Frobenius norms forces us to consider:

$$\begin{aligned}\|W_1\|_F^2 + \|W_2\|_F^2 &= \|\Sigma_1\|_F^2 + \|\Sigma_2\|_F^2 \\ &= \sum_{i=1}^{k} \left(\sigma_i^2 + \frac{1}{\sigma_i^2}\right)\end{aligned}$$

where W_1 is bringing the σ_i terms and its approximate pseudo-inverse W_2 is bringing the $\frac{1}{\sigma_i}$ for its singular values.

Minimizing $\frac{1}{\sigma^2} + \sigma^2$ by taking derivatives results in setting $0 = -\frac{2}{\sigma^3} + 2\sigma$ which has a unique non-negative real solution at $\sigma = 1$, and so this is the unique minimizer since clearly this expression goes to ∞ as $\sigma \to \infty$ or $\sigma \to 0$.

If the λ is small enough, then the optimization essentially decouples: the main loss forces W_2 and W_1 to be pseudoinverses and to have the product W_2W_1 project onto the supspace spanned by the k singular vectors of X whose singular values are largest; and the regularization term forces the individual W_2 and W_1 to have all nonzero singular values as close to 1 as possible.

Once $\sigma_i \approx 1$, the matrix has approximately orthonormal columns for W_2 and approximately orthonormal rows for W_1. You can see this by simply writing $W = U\Sigma V^\top$ and then noticing for a tall W that $W^\top W = V\Sigma^\top\Sigma V^\top \approx VIV^\top = I$ and similarly for $WW^\top$ for a wide W.

6. Exploring Deep Learning Tooling

Deep learning in practice often requires the use of various tooling in order make the learning workflow efficient. Among these include tools for creating graphs and organizing experiments. These kinds of tools can aid in careful ablation studies, and can accelerate research. We will explore the use of two different tools in this question: tensorboard and wandbai.

(a) Complete the first notebook: tensorboard.ipynb. Provide your graphs here. What is easy about tensorboard? What do you dislike? Would it still be easy to use when we need to run massive amounts of experiments? How organized is it?

Solution: Students should include some training graphs here. Tensorboard allows students to log runs without much thought, as well as visualize them. Students should note that it can be quite disorganized

and harder to use because they can only sort with regex. It would not be good to run massive amounts of experiments due to the inabilty to sort by configurations.

(b) Complete the second notebook: wandb.ipynb. No need to provide your graphs. What does wandb have that tensorboard does not?

Solution: Students should include the best hyperparameters. In addition they should note that wandb is useful for visualizations and having many runs. Since wandb allows for sorting based on configuration, this could be very useful.

7. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework?**

Contributors:

- Saagar Sanghavi.
- Dhruv Shah.
- Olivia Watkins.
- Jerome Quenum.
- Anant Sahai.
- Anrui Gu.
- Matthew Lacayo.
- Past EECS 282 and 227 Staff.
- Linyuan Gong.
- Kumar Krishna Agrawal.
- Sheng Shen.
- Liam Tan.

