EECS 182 Deep Neural Networks
Spring 2023 Anant Sahai

Homework 7

This homework is due on Friday, March 17, 2022, at 10:59PM.

1. 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,

- Download `submission_log.json` and submit it to "Homework 7 (Code)" in Gradescope.
- 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 1. Please refer to the solution notebook for the codes.

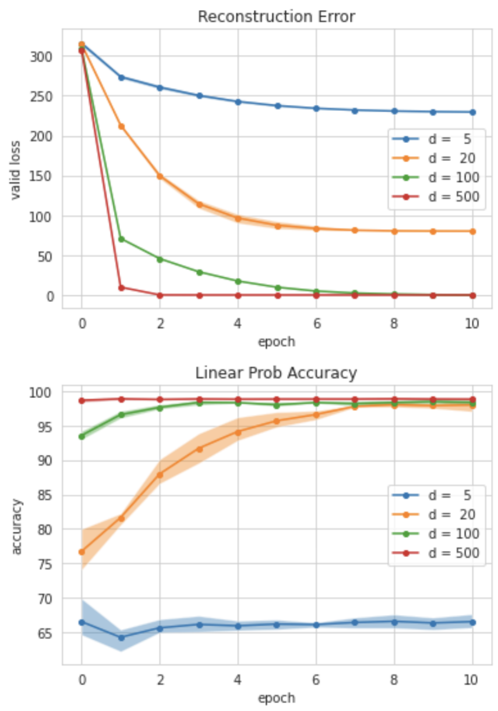


Figure 1: Visualization of the vanilla autoencoder with different latent representation sizes.

(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?**

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 1.

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,

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

We will assume $\sigma_1^2 > \cdots > \sigma_k^2 > 0$ are the k 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, and is common in this setting. Therefore, the top-k eigenvalues of $\mathbf{X}$ are $S = \mathrm{diag}(\sigma_1, \ldots, \sigma_\mathrm{k})$, with corresponding eigenvectors are the columns of $\mathbf{U}_k \in \mathbb{R}^{m \times k}$. A well-established result from (Baldi & Hornik, 1989) 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. 1 would satisfy.**

Solution: We can compute the first order conditions for W_1 and W_2, respectively. To get started, let's note that for matrices $A \in \mathbb{R}^{d \times n}, W \in \mathbb{R}^{k \times d}$

$$\|WA\|_F^2 = \mathrm{tr}(A^\top W^\top W A)$$
$$\nabla_W \|WA\|_F^2 = 2WAA^\top$$

Now, consider taking gradient w.r.t the loss function

$$\begin{aligned}\mathcal{L}(W_1, W_2; \mathbf{X}) &= \frac{1}{n}||\mathbf{X} - W_2 W_1 \mathbf{X}||_F^2 \\ &= \frac{1}{n}\|(\mathbf{I} - W_2 W_1)\mathbf{X}\|_F^2 \\ &= \frac{1}{n}\|\mathbf{H}(W_1, W_2)\mathbf{X}\|_F^2\end{aligned}$$

where $\mathbf{H}(W_1, W_2) = I - W_2 W_1$. Taking the gradient of the above loss function w.r.t W_2, we get

$$\begin{aligned}\nabla_{W_2}\mathcal{L} &= \nabla_{W_2}\frac{1}{n}\|\mathbf{H}(W_1, W_2)\mathbf{X}\|_F^2 \\ &= 2\mathbf{H}(W_1, W_2)\mathbf{X}\mathbf{X}^\top W_1^\top \\ &= 2\Big(\mathbf{I} - \mathbf{W_2 W_1}\Big)\mathbf{X}\mathbf{X}^\top W_1^\top\end{aligned}$$

Similarly, gradient w.r.t W_1 gives

$$\begin{aligned}\nabla_{W_1}\mathcal{L} &= \nabla_{W_1}\frac{1}{n}\|\mathbf{H}(W_1, W_2)\mathbf{X}\|_F^2 \\ &= 2\mathbf{H}(W_1, W_2)\mathbf{X}\mathbf{X}^\top W_2 \\ &= 2W_2^\top\Big(\mathbf{I} - \mathbf{W_2 W_1}\Big)\mathbf{X}\mathbf{X}^\top\end{aligned}$$

In this case, for W_1, W_2 that satisfy the above set of equations provide the first order optimality conditions and be the minima, i.e. we have

$$\Big(\mathbf{X} - \mathbf{W_2 W_1 X}\Big)\mathbf{X}^\top W_1^\top = 0$$
$$W_2^\top\Big(\mathbf{X} - \mathbf{W_2 W_1 X}\Big)\mathbf{X}^\top = 0$$

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

Solution: (Note: The solution here is questionable. It is possible that the principal components $\mathbf{U}_k$ do not satisfy the optimality conditions outlined in (i).)

Using the principal components (top-k), we have $W_2 = W_1^\top = \mathbf{U}_k = [\mathbf{u}_1, \ldots, \mathbf{u}_k]$, where $\mathbf{u}_i \in$

$\mathbb{R}^{m\times 1}$, $\mathbf{X} = \sum_{i=1}^{m} \sigma_i \mathbf{u}_i \mathbf{u}_i^\top$. Next, consider the term $I - W_2W_1$, we have

$$\mathbf{I} - \mathbf{W_2W_1} = \mathbf{I} - \mathbf{U_k U_k^\top} \tag{2}$$

$$= \begin{bmatrix} \mathbf{0}_{k,k} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{m-k,m-k} \end{bmatrix} \tag{3}$$

Plugging this back into the optimality conditions for part (i) we have

$$\Big(\mathbf{X} - \mathbf{W_2W_1X}\Big)\mathbf{X}^\top W_1^\top = \Big(\mathbf{I} - \mathbf{W_2W_1}\Big)\mathbf{XX}^\top W_1^\top \tag{4}$$

Similarly, the term $\mathbf{XX}^\top W_2$ can be simplified as

$$\mathbf{XX^\top W_2} = \mathbf{U\Sigma^2U^\top U_k}$$
$$= \mathbf{U\Sigma^2} \begin{bmatrix} \mathbf{I}_{k,k} \\ \mathbf{0}_{m-k,k} \end{bmatrix}$$

Plugging these together, we have (similarly for $\nabla_{W_2}\mathcal{L}$)

$$\nabla_{W_1}\mathcal{L} = (\mathbf{X} - W_2W_1\mathbf{X})\mathbf{X}^\top W_2$$
$$= \begin{bmatrix} \mathbf{0}_{k,k} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{m-k,m-k} \end{bmatrix} \mathbf{U\Sigma^2} \begin{bmatrix} \mathbf{I}_{k,k} \\ \mathbf{0}_{m-k,k} \end{bmatrix}$$
$$= \mathbf{0}$$

2. 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,

$$\mathcal{L}(W_1, W_2; \mathbf{X}) = \frac{1}{n}||\mathbf{X} - W_2W_1\mathbf{X}||_F^2 \tag{5}$$

We will assume $\sigma_1^2 > \cdots > \sigma_k^2 > \sigma_{k+1}^2 \geq 0$ are the $k+1$ largest eigenvalues of $\frac{1}{n}\mathbf{XX}^\top$. 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:

$$\mathcal{L}_\lambda(W_1, W_2; \mathbf{X}) = \tfrac{1}{n}||\mathbf{X} - W_2W_1\mathbf{X}||_F^2 + \lambda\|W_1\|_F^2 + \lambda\|W_2\|_F^2. \tag{6}$$

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 (6) 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 (6) 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 of the optimizers listed, the **SGD-Optimizer** is the one that would work the best.

(b) **Do you think that the solution to (6) 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 $W_1 = U_1\Sigma_1V_1^\top$ and $W_2 = U_2\Sigma_2V_2^\top$. 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.

3. Justifying Scaled-Dot Product Attention

Suppose $q, k \in \mathbb{R}^d$ are two random vectors with $q, k\ N(\mu, \sigma^2 I)$, where $\mu \in \mathbb{R}^d$ and $\sigma \in \mathbb{R}^+$. In other words, each component q_i of q is drawn from a normal distribution with mean μ and stand deviation σ, and the same if true for k.

(a) Define $\mathbb{E}[q^T k]$ in terms of μ, σ and d.

Solution: $\mathbb{E}[q^T k] = \mathbb{E}[\sum_{i=1}^d q_i k_i] = \sum_{i=1}^d \mathbb{E}[q_i k_i] == \sum_{i=1}^d \mathbb{E}[q_i]\mathbb{E}[k_i] = \sum_{i=1}^d \mu_i^2 = ||\mu||_2^2$

(b) Considering a practical case where $\mu = 0$ and $\sigma = 1$, define $\text{Var}(q^T k)$ in terms of d.

Solution: $\text{Var}(q^T k) = \mathbb{E}[(q^T k)^2] - \mathbb{E}[q^T k]^2 = \mu^T \Sigma_q \mu + \mu^T \Sigma_k \mu + \text{tr}(\Sigma_q \Sigma_k) = 2\sigma^2 ||\mu||_2^2 + d * \sigma^4$ (here's an alternate way of solving this)

$$\begin{aligned}
\text{Var}(q^T k) &= \text{Var}(\sum_{i=1}^{d}(q_i k_i)) && \text{// def of dot product}\\
&= \sum_{i=1}^{d}(\text{Var}(q_i k_i)) && \text{// all } q_i \text{ and } k_i \text{ are independent}\\
&= d\text{Var}(q_1 k_1) && \text{// the variance is the same for all i}\\
&= d(\mathbb{E}[q_1]^2 \text{Var}(k_1) + \mathbb{E}[k_1]^2 \text{Var}(q_1) + \text{Var}(q_1)\text{Var}(k_1)) && \text{// Var of product of indep. variables}\\
&= d * \text{Var}(q_1)\text{Var}(k_1) && \text{// all the expectations are 0}\\
&= d\sigma^4\\
&= d
\end{aligned}$$

(c) Continue to use the setting in part (b), where $\mu = 0$ and $\sigma = 1$. Let s be the scaling factor on the dot product. Suppose we want $\mathbb{E}[\frac{q^T k}{s}]$ to be 0, and $\text{Var}(\frac{q^T k}{s})$ to be $\sigma = 1$. What should s be in terms of d?

Solution: From part (a), we know that $\mathbb{E}[q^T k] = \sum_{i=1}^d \mathbb{E}[\mu_i]^2 = 0$. From part (b), we know that $\text{Var}[q^T k] = d^2$. So to $\mathbb{E}[\frac{q^T k}{s}]$ to be 0, and $\text{Var}(\frac{q^T k}{s}) = \frac{d}{s^2}$ to be $\sigma = 1$, we can have $s = \sqrt{d}$

4. Argmax Attention

Recall from lecture that we can think about attention as being *queryable softmax pooling*. In this problem, we ask you to consider a hypothetical argmax version of attention where it returns exactly the value corresponding to the key that is most similar to the query, where similarity is measured using the traditional inner-product.

(a) Perform **argmax attention** with the following keys and values:

Keys:

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

Corresponding Values:

$$\left\{ \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \\ 4 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \right\}$$

using the following query:

$$\mathbf{q} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

What would be the output of the attention layer for this query?

Hint: For example, $\text{argmax}([1, 3, 2]) = [0, 1, 0]$

Solution: Compute inner products of query with the keys:

$$\langle [1,1,2]^\top, [1,2,0]^\top \rangle = 3$$

$$\langle [1,1,2]^\top, [0,3,4]^\top \rangle = 11$$

$$\langle [1,1,2]^\top, [5,0,0]^\top \rangle = 5$$

$$\langle [1,1,2]^\top, [0,0,1]^\top \rangle = 2$$

We take argmax rather than softmax, and have $\text{argmax}([3, 11, 5, 2]) = [0, 1, 0, 0]$

Now, take weighted sum of value vectors (in this case all are zeroed out except for the one corresponding to the highest dot-product between query and key)

$0 \cdot [2,0,1]^\top + 1 \cdot [1,4,3]^\top + 0 \cdot [0,-1,4]^\top + 0 \cdot [1,0,-1]^\top$

So final output is $\begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix}$

(b) Note that instead of using *softmax* we used `argmax` to generate outputs from the attention layer. **How does this design choice affect our ability to usefully train models involving attention?**

(Hint: think about how the gradients flow through the network in the backward pass. Can we learn to improve our queries or keys during the training process?)

Solution: It wouldn't work in real life, since the gradients only flow through the one element that is selected by argmax and thus most of the parameters in the transformer layers would remain the same if we did gradient-based training. The reason is that generically, the argmax is not sensitive to small changes in the keys and queries, since any such tiny perturbations will not change the winner. Consequently, the gradients with respect to the keys and queries will always be zero. This means that the keys and queries will never be improved — or more precisely, the functions used to generate keys and queries from the state will never get updated.

5. Kernelized Linear Attention

The softmax attention is widely adopted in transformers (Luong et al., 2015; Vaswani et al., 2017), however the $\mathcal{O}\left(N^2\right)$ (N stands for the sequence length) complexity in memory and computation often makes it less desirable for processing long document like a book or a passage, where the N could be beyond thousands. There is a large body of the research studying how to resolve this [1].

Under this context, this question presents a formulation of attention via the lens of the kernel. A large portion of the context is adopted from Tsai et al. (2019). In particular, attention can be seen as applying a kernel over the inputs with the kernel scores being the similarities between inputs. This formulation sheds light on individual components of the transformer's attention, and helps introduce some alternative attention mechanisms that replaces the "softmax" with linearized kernel functions, thus reducing the $\mathcal{O}\left(N^2\right)$ complexity in memory and computation.

We first review the building block in the transformer. Let $x \in \mathbb{R}^{N \times F}$ denote a sequence of N feature vectors of dimensions F. A transformer Vaswani et al. (2017) is a function $T : \mathbb{R}^{N \times F} \to \mathbb{R}^{N \times F}$ defined by the composition of L transformer layers $T_1(\cdot), \ldots, T_L(\cdot)$ as follows,

$$T_l(x) = f_l(A_l(x) + x). \tag{7}$$

The function $f_l(\cdot)$ transforms each feature independently of the others and is usually implemented with a small two-layer feedforward network. $A_l(\cdot)$ is the self attention function and is the only part of the transformer that acts across sequences.

We now focus on the the self attention module which involves softmax. The self attention function $A_l(\cdot)$ computes, for every position, a weighted average of the feature representations of all other positions with a weight proportional to a similarity score between the representations. Formally, the input sequence x is projected by three matrices $W_Q \in \mathbb{R}^{F \times D}$, $W_K \in \mathbb{R}^{F \times D}$ and $W_V \in \mathbb{R}^{F \times M}$ to corresponding representations Q, K and V. The output for all positions, $A_l(x) = V'$, is computed as follows,

$$\begin{aligned} Q &= xW_Q, K = xW_K, V = xW_V, \\ A_l(x) &= V' = \text{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V. \end{aligned} \tag{8}$$

Note that in the previous equation, the softmax function is applied rowwise to QK^T. Following common terminology, the Q, K and V are referred to as the "queries", "keys" and "values" respectively.

Equation 8 implements a specific form of self-attention called softmax attention where the similarity score is the exponential of the dot product between a query and a key. Given that subscripting a matrix with i returns the i-th row as a vector, we can write a generalized attention equation for any similarity function as follows,

$$V_i' = \frac{\sum_{j=1}^N \text{sim}\left(Q_i, K_j\right) V_j}{\sum_{j=1}^N \text{sim}\left(Q_i, K_j\right)}. \tag{9}$$

Equation 9 is equivalent to equation 8 if we substitute the similarity function with $\text{sim}_{\text{softmax}}(q, k) = \exp(\frac{q^T k}{\sqrt{D}})$. This can lead to

$$V_i' = \frac{\sum_{j=1}^N \exp(\frac{Q_i^T K_j}{\sqrt{D}}) V_j}{\sum_{j=1}^N \exp(\frac{Q_i^T K_j}{\sqrt{D}})}. \tag{10}$$

For computing the resulting self-attended feature $A_l(x) = V'$, we need to compute all V_i' $i \in 1, \ldots, N$ in equation 10.

[1] https://huggingface.co/blog/long-range-transformers

(a) **Identify the conditions that needs to be met by the sim function to ensure that V_i in Equation 9 remains finite (the denominator never reaches zero).**

Solution: In order for V_i in Equation 9 to remain finite, it is necessary that $\sum_{j=1}^{N} \mathrm{sim}\left(Q_i, K_j\right) \neq 0$ for all possible values of Q and K. This condition implies that the function value of sim must always have the same sign (positive or negative).

For the purposes of this problem, we will only consider the case where sim consists of *kernels* $k(x, y) : \mathbb{R}^F \times \mathbb{R}^F \to \mathbb{R}^+$ that yield positive values.

(b) The definition of attention in equation 9 is generic and can be used to define several other attention implementations.

(i) One potential attention variant is the "polynomial kernel attention", where the similarity function as $\mathrm{sim}\left(q, k\right)$ is measured by polynomial kernel $\mathcal{K}$[2]. **Considering a special case for a "quadratic kernel attention" that the degree of "polynomial kernel attention" is set to be 2, derive the $\mathrm{sim}\left(q, k\right)$ for "quadratic kernel attention". (NOTE: any constant factor is set to be 1.)** .

Solution: Quadratic kernel attention is $\mathrm{sim}\left(q, k\right) = \left(q^T k + 1\right)^2 = \phi\left(q\right)^T \phi\left(k\right)$.

(ii) One benefit of using kernelized attention is that we can represent a kernel using a feature map $\phi\left(\cdot\right)$[3]. **Derive the corresponding feature map $\phi\left(\cdot\right)$ for the quadratic kernel.**

Solution: The feature map $\phi\left(\cdot\right)$ can be expressed as

$$\phi\left(x\right) = [1, \sqrt{2}x_1, \sqrt{2}x_2, ..., \sqrt{2}x_D, x_1^2, x_2^2, ..., x_D^2, \sqrt{2}x_1x_2, ..., \sqrt{2}x_{D-1}x_D]^T$$

(iii) **Considering a general kernel attention, where the kernel can be represented using feature map that $\mathcal{K}(q, k) = (\phi\left(q\right)^T \phi\left(k\right))$, rewrite kernel attention of equation 9 with feature map $\phi\left(\cdot\right)$.**

Solution: The general kernel attention can be rewritten as

$$V_i' = \frac{\sum_{j=1}^{N} \phi\left(Q_i\right)^T \phi\left(K_j\right) V_j}{\sum_{j=1}^{N} \phi\left(Q_i\right)^T \phi\left(K_j\right)},$$

(c) We can rewrite the softmax attention in terms of equation 9 as equation 10. **For all the V_i' ($i \in \{1, ..., N\}$), derive the time complexity (asymptotic computational cost) and space complexity (asymptotic memory requirement) of the above softmax attention in terms of sequence length N, D and M.**

NOTE: for memory requirement, we need to store any intermediate results for backpropagation, including all Q, K, V

Solution: The computational graph can be illustrated with the following pseudocode:

```
for i in range(N):
    for j in range(N):
        S[i, j] = Q[i, :].T @ K[j, :] / sqrt(D)
for i in range(N):
    Z = 0
```

[2] https://en.wikipedia.org/wiki/Polynomial_kernel
[3] https://en.wikipedia.org/wiki/Kernel_method

```
        for j in range(N):
            Z += exp(S[i, j])
        for j in range(N):
            A[i, j] = exp(S[i, j]) / Z
    for i in range(N):
        O[i, :] = 0
        for j in range(N):
            O[i, :] += A[i, j] * V[j, :]
```

Time: Therefore, the computational cost of each double-layer for-loop is $\mathcal{O}(N^2D)$, $\mathcal{O}(N^2D)$, and $\mathcal{O}(N^2M)$, respectively, So the total time complexity is $\mathcal{O}(N^2(D+M))$ (or equivalently, $\mathcal{O}(N^2 \max(D, M))$).

Space: Q, K takes $\mathcal{O}(ND)$ memory, S, A needs $\mathcal{O}(N^2)$ memory, and V, O needs $\mathcal{O}(NM)$ memory. So the total space complexity is $\mathcal{O}\left(N(N+M+D)\right)$ (or equivalently, $\mathcal{O}(N \max(N, M, D))$).

(d) Assume we have a kernel $\mathcal{K}$ as the similarity function and the kernel can be represented with a feature map $\phi(\cdot)$, we can rewrite equation 9 with $\mathrm{sim}(x, y) = \mathcal{K}(x, y) = \left(\phi(Q_i)^T \phi\left(K_j\right)\right)$ in part (b). We can then further simplify it by making use of the associative property of matrix multiplication to

$$V_i' = \frac{\phi(Q_i)^T \sum_{j=1}^N \phi\left(K_j\right) V_j^T}{\phi(Q_i)^T \sum_{j=1}^N \phi\left(K_j\right)}. \tag{11}$$

Note that the feature map $\phi(\cdot)$ is applied row-wise to the matrices Q and K.

Considering using a linearized polynomial kernel $\phi(x)$ of degree 2, and assume $M \approx D$, derive the computational cost and memory requirement of this kernel attention as in (11).

Solution: The computational graph can be illustrated with the following pseudocode:

```
U[:, :] = 0  # shape: [C, M]
for j in range(N):
    U[:, :] += phi(K[j, :]) @ V[j, :].T
Z[:] = 0  # shape: [C]
for j in range(N):
    Z[:] += phi(K[j, :])
for i in range(N):
    O[i, :] = phi(Q[i, :]).T @ U[:, :]
    O[i, :] /= phi(Q[i, :]).T @ Z[:]
```

Time: the computational cost of each step is $\mathcal{O}(NCM)$, $\mathcal{O}(NC)$, $\mathcal{O}(NCM)$, respectively. So the total computational cost is $\mathcal{O}(NCM)$. For the quadratic kernel, we have $C = \mathcal{O}\left(D^2\right)$, and apply $M \approx D$, then the time complexity is $\mathcal{O}\left(ND^3\right)$. If $N >> D^2$, kernelized linear attention with a quadratic polynomial kernel is faster than softmax attention.

Space: Q, K takes $\mathcal{O}(ND)$ memory, U needs $\mathcal{O}(CM)$ memory, and V, O needs $\mathcal{O}(NM)$ memory. So the total space complexity is $\mathcal{O}\left(N(D+M)+CM\right)$. For the quadratic kernel, we have $C = \mathcal{O}\left(D^2\right)$, and apply $M \approx D$, then the space complexity is $\mathcal{O}\left(ND + D^3\right)$ (or equivalently $\mathcal{O}\left(D \max(N, D^2)\right)$). If $N >> D^2$, kernelized linear attention uses much less memory than softmax attention.

6. Debugging DNNs (Optional)

(a) Your friends want to train a classifier for a new app they're designing. They implement a deep convolutional network and train models with two configurations: a 20 layer model and a 56 layer model.

However, they observe the following training curves and are surprised that the 20-layer network has better training as well as test error.

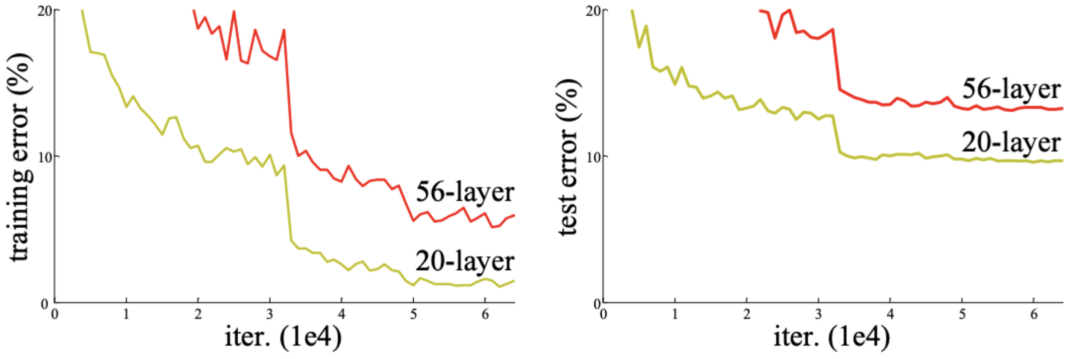


Figure 2: Training deep networks on CIFAR10

What are the potential reasons for this observation? Are there changes to the architecture design that could help mitigate the problem?

Solution: The fact that the training error is also not getting better is a classic indication of some sort of "underfitting" type behavior going on. Since the deeper model clearly has more expressive capacity, the underfitting is not likely due to an issue with too much approximation error and hence has to do with the behavior of training itself. Since we're talking about Deep Learning here, the training is happening using gradient based algorithms. So the reason is likely to involve updates that are too small, which suggests some kind of dying-gradient problem.

In a deep network, dying gradients could be occuring because of poor initialization in which case using He or Xavier initialiation could help. But the question is asking about changes to the architecture design, so this isn't what was intended.

An architectural source of dying gradients is having gradients attenuated as they backprop through multiple layers. This can be mitigated by adding residual/skip connections, which is an architectural matter.

Adding normalization layers could also potentially help in such settings.

(b) You and your teammate want to compare batch normalization and layer normalization for the ImageNet classification problem. You use ResNet-152 as a neural network architecture. The images have input dimension $3 \times 224 \times 224$ (channels, height, width). You want to use a batch size of 1024; however, the GPU memory is so small that you cannot load the model and all 1024 samples at once — you can only fit 32. Your teammate proposes using a gradient-accumulation algorithm:

Gradient accumulation refers to running the forward and backward pass of a model a fixed number of steps (`accumulation_steps`) without updating the model parameters, while aggregating the gradients. Instead, the model parameters are updated every (`accumulation_steps`). This allows us to increase the effective batch size by a factor of `accumulation_steps`.

You implement the algorithm in PyTorch as:

```
model.train()
optimizer.zero_grad()
for i, (inputs, labels) in enumerate(training_set):
    predictions = model(inputs)
```

```
        loss = loss_function(predictions, labels)
        loss = loss / accumulation_steps
        loss.backward()
        if (i+1) % accumulation_steps == 0:
            optimizer.step()
            optimizer.zero_grad()
```

Note that the `.backward()` operator in PyTorch implicitly keeps accumulating the gradient for all the parameters, unless zero'd out with an `optimizer.zero_grad()` call.

Before running actual experiments, your friend suggests that you should test whether the gradient accumulation algorithm is implemented correctly. To do so, you collect the output logits (i.e. the outputs of the last layer) from two models — ResNet-152, one with batchnorm and the other with layernorm — using different combinations of batch sizes and the number of accumulation steps that keep the effective batch size to 32.

[*Note that the effective batch size is product of the batch size and* `accumulation_steps`*. In other words, the possible combinations for effective batch-size 32 are:*
(`batch_size`*,* `accumulation_steps`*) = (1, 32), (2, 16), (4, 8), (8, 4), (16, 2), (32, 1).*]

Here, the (32,1) combination is the approach without the "gradient accumulation" trick, and we want to see whether the others agree with this.

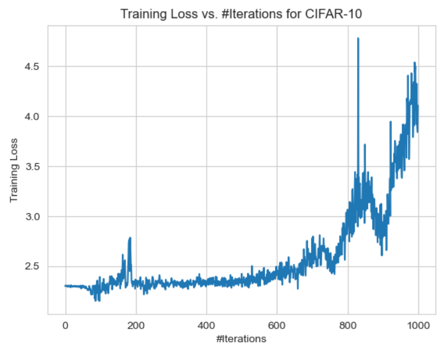
On running these tests, you observe that one of models: either with batchnorm or with layernorm, doesn't pass the test. **Which one do you expect to not pass the test and why?**

Solution: Answer: Batchnormalization. Because batch statistics are calculated along the batch size not along the effective batch size. So by using smaller batches and accumulating, we are actually computing something different. Meanwhile, layer normalization doesn't care about what is happening elsewhere in the batch and so would not be effected by the batch size when it comes to activations.

(c) You are training a CIFAR model and observe that the model is diverging (instead of the training loss decreasing over iterations). **Debug the pseudocode and give a correction that you believe would actually result in reasonable convergence during training.**

(Note: You can assume that the datasets are loaded correctly, model is trained with SGD optimizer with learning rate$= 0.001$*, batchsize*$= 100$*)*

(HINT: Ideas from the previous part of this question might be relevant.)



```
model.train()
optimizer.zero_grad()
for (inputs, labels) in training_set:
    predictions = model(inputs)
    loss = loss_fn(predictions, labels)
    loss.backward()
    optimizer.step()
```

Figure 3: Training loss for CIFAR10

Solution: The problem here is that the gradients are not being zeroed out in the training loop. So we keep applying old gradients over and over again along with new ones, which results in training instability. We need to move the *optimizer.zero_grad()* call to the beginning of the inside of the training loop instead of having it before the training loop.

Note, you need to provide a very unambiguous patch/correction to the pseudocode for full credit.

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? Write it down here where you'll need to remember it for the self-grade form.**

References

Baldi, P. and Hornik, K. Neural networks and principal component analysis: Learning from examples without local minima. *Neural networks*, 2(1):53–58, 1989.

Luong, M.-T., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1412–1421, 2015.

Tsai, Y.-H. H., Bai, S., Yamada, M., Morency, L.-P., and Salakhutdinov, R. Transformer dissection: An unified understanding for transformer's attention via the lens of kernel. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4344–4353, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Contributors:

- Kumar Krishna Agrawal.
- Linyuan Gong.
- Sheng Shen.
- Anant Sahai.
- David M. Chan.
- Saagar Sanghavi.
- Shaojie Bai.
- Angelos Katharopoulos.
- Hao Peng.
- Suhong Moon.