EECS 182 Deep Neural Networks
Fall 2022 Anant Sahai

Homework 6

This homework is due on Sunday, Oct 23, 2022, at 10:59PM.

Deliverables: Please submit the code/notebooks to the code assignment. Alongside the code, attach a pdf printout of the notebook to the written assignment.

1. Zachary's Karate Club

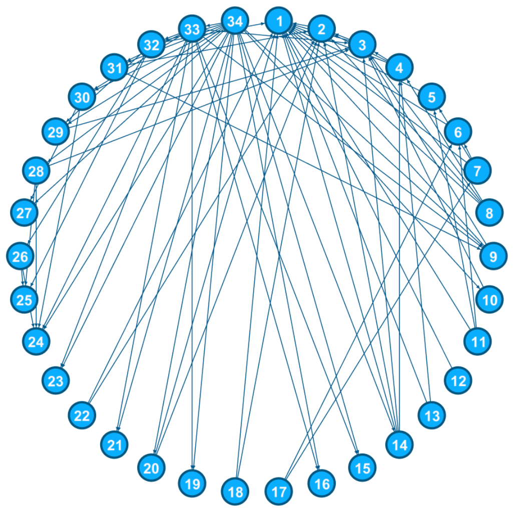

Figure 1: Zachary's Karate Club Graph

Zachary's karate club (ZKC) is a social network of a university karate club, described in the paper "An Information Flow Model for Conflict and Fission in Small Groups" by Wayne W. Zachary.

A social network captures 34 members of a karate club, documenting links between pairs of members who interacted outside the club.

During the study a conflict arose between the officer/ administrator ("John A") and the instructor "Mr. Hi", which led to the split of the club into two.

Half of the members formed a new club around Mr. Hi; members from the other part found a new instructor or gave up karate.

Based on collected data Zachary correctly assigned all but one member of the club to the groups they actually joined after the split. You could read more about it here `https://www.jstor.org/stable/3629752`, and here `https://commons.wikimedia.org/wiki/File:Social_Network_Model_of_Relationships_in_the_Karate_Club.png`

1 Implement all the TODOs in `q_zkc.ipynb`.

2 Comment on what you have learned

2. 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 $s = \sqrt{d}$

3. Kernelized Linear Attention

The softmax attention is widely adopted in transformers (Luong et al., 2015; Vaswani et al., 2017), however the $\mathcal{O}(N^2)$ (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 $\mathcal{O}(N^2)$ 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 $T : \mathbb{R}^{N \times F} \rightarrow \mathbb{R}^{N \times F}$ defined by the

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

composition of L transformer layers $T_1(\cdot), \ldots, T_L(\cdot)$ as follows,

$$T_l(x) = f_l(A_l(x) + x). \tag{1}$$

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,

$$\begin{aligned} Q = xW_Q, K = xW_K, V = xW_V, \\ A_l(x) = V' = \text{softmax}(\frac{QK^T}{\sqrt{D}})V. \end{aligned} \tag{2}$$

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 2 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,

$$V_i' = \frac{\sum_{j=1}^N \text{sim}\left(Q_i, K_j\right) V_j}{\sum_{j=1}^N \text{sim}\left(Q_i, K_j\right)}. \tag{3}$$

Equation 3 is equivalent to equation 2 if we substitute the similarity function with $\text{sim}_{\text{softmax}}(q,k) = \exp(\frac{q^Tk}{\sqrt{D}})$. This can lead to

$$V_i' = \frac{\sum_{j=1}^N \exp(\frac{Q_i^T K_j}{\sqrt{D}})V_j}{\sum_{j=1}^N \exp(\frac{Q_i^T K_j}{\sqrt{D}})}. \tag{4}$$

For computing the resulting self-attended feature $A_l(x) = V'$, we need to compute all V_i' $i \in 1, ..., N$ in equation 4.

(a) Determine what conditions on the denominator in equation 3 would prevent V_i from blowing up.

Solution: Note that the only constraint we need to impose to sim $(\cdot)$, in order for equation 3 to define an attention function, is to be non-negative. This includes all kernels $k(x,y) : \mathbb{R}^{2\times F} \to \mathbb{R}^+$. The general intuition is we want the denominator in equation 3 not to be zero.

(b) The definition of attention in equation 3 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 sim (q,k) is measured by polynomial kernel $\mathcal{K}$ [2]. **Considering a special case for a "quadratic**

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

kernel attention" that the degree of "polynomial kernel attention" is set to be 2, derive the sim (q, k) for "quadratic kernel attention". (NOTE: any constant factor is set to be 1.) .

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

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

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

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

iii. **Considering a general kernel attention, where the kernel can be represented using feature map that $\mathcal{K}(q, k) = (\phi\,(q)^T \phi\,(k))$, rewrite kernel attention of equation 3 with feature map $\phi\,(\cdot)$.**

Solution: The general kernel attention can be rewritten as

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

(c) i. We can rewrite the softmax attention in terms of equation 3 as equation 4. **For all the V_i' ($i \in \{1, ..., N\}$), derive the computational cost and memory requirement of the above softmax attention in terms of sequence length N, D and M. (NOTE: for memory requirement, think that we need to store any intermediate results for backpropagation, including all Q, K, V)**

Solution: For softmax attention, the total cost in terms of multiplications and additions scales as $\mathcal{O}\left(N^2 \max\,(D, M)\right)$, where D is the dimensionality of the queries and keys and M is the dimensionality of the values. For memory requirement, we need $\mathcal{O}\left(N(M + D)\right)$ tore K, Q, V, $\mathcal{O}\left(N^2\right)$ to store all $Q_i^T K_j$, $\mathcal{O}\left(N^2 M\right)$ to store all $\exp\!\left(q_i^T * k_j/\sqrt{d}\right) V_j$ and $\mathcal{O}\,(NM)$ to store all V_i', which sums to $\mathcal{O}\left(N^2 M + ND\right)$. $\mathcal{O}\left(N^2\,(D + M)\right)$ is also a valid answer.

ii. 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 3 with $\text{sim}\,(x, y) = \mathcal{K}(x, y) = \left(\phi\,(Q_i)^T \phi\,(K_j)\right)$ in part (b). We can then further simplify it by making use of the associative property of matrix multiplication to

$$V_i' = \frac{\phi\,(Q_i)^T \sum_{j=1}^N \phi\,(K_j)\, V_j^T}{\phi\,(Q_i)^T \sum_{j=1}^N \phi\,(K_j)}. \tag{5}$$

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, derive the computational cost and memory requirement of this kernel attention as in (5).

Solution: On the contrary, for linear attention, we first compute the feature maps of dimensionality C. Subsequently, computing the new values requires $\mathcal{O}\,(NCM)$ additions and multiplications. For the quadratic kernel, we have $C = \mathcal{O}\left(D^2\right)$. The computational cost for a linearized polynomial transformer of degree 2 is $\mathcal{O}\left(ND^2 M\right)$. This makes the computational complexity favorable when $N > D^2$. The memory requirement is thus $\mathcal{O}\left(NM \max(C, D)\right)$.

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

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

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

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 $z = \mathrm{encoder}(x)$.

(a) Designing AutoEncoders

In the accompanying notebook `q_ae.ipynb` we explore the following variants of an autoencoder architecture on a synthetic & MNIST dataset to learn representations, and compare the performance of a linear classifier with these features.

The notebook implements three different class of autoencoder architectures:

(1) **Vanilla AutoEncoder**
For high-dimensional input $x \in \mathbb{R}^m$, an interpretation of autoencoder is that of *dimensionality reduction*. Here, the encoder f_θ is a *lossy* function that maps x to lower dimensional $z \in \mathbb{R}^k$ when $k < m$. The decoder $g_\phi : \mathbb{R}^k \to \mathbb{R}^m$ is a function that maps z to a reconstruction $\hat{x}$ of x.

(2) **Denoising AutoEncoder**
To learn robust lower-dimensional representations, we can add noise to the input x before passing it to the encoder. This is known as a *denoising autoencoder*.

(3) **Masked AutoEncoder**
An alternative to adding noise to the input is to *mask* the input. That is, we randomly set some of the entries of x to zero (or a fixed special value) before passing to the encoder, and try decoding the full signal x. This is known as a *masked autoencoder*, and the proxy task is often referred to as *inpainting*.

After filling the `TODO`s in the notebook, "pretrain" models with different unsupervised learning objectives on given training set. Evaluate representation quality using a linear classifier with these features on the "downstream" classification task, for the following architectures:

(i) **Linear Autoencoders** : In these architectures don't use any non-linearities (stacking multiple layers is allowed). Train models with different bottlenecks `k ∈ [5, 10, 100, 1000]`.

(ii) **Non-Linear** : AutoEncoders that use non-linearities (stacking multiple layers is allowed), train models with different bottleneck dimensions $k \in$ `[5, 10, 100, 1000]`, `ReLU` activation.

(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,

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

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 $S = \text{diag}(\sigma_1, \ldots, \sigma_\text{k})$, 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. 6 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 $A \in \mathbb{R}^{d\times n}, W \in \mathbb{R}^{k\times d}$

$$\begin{aligned} \|WA\|_F^2 &= \text{tr}(A^\top W^\top WA) \\ \nabla_W \|WA\|_F^2 &= WAA^\top \end{aligned}$$

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

$$\begin{aligned} \mathcal{L}(W_1, W_2; \mathbf{X}) &= \frac{1}{n}||\mathbf{X} - W_2W_1\mathbf{X}||_F^2 \\ &= \frac{1}{n}\|(\mathbf{I} - W_2W_1)\mathbf{X}\|_F^2 \\ &= \frac{1}{n}\|\mathbf{H}(W_1, W_2)\mathbf{X}\|_F^2 \end{aligned}$$

where $\mathbf{H}(W_1, W_2) = I - W_2W_1$. Taking the gradient of the above loss function w.r.t W_2, we get

$$\begin{aligned} \nabla_{W_2}\mathcal{L} &= \nabla_{W_2}\frac{1}{n}\|\mathbf{H}(W_1, W_2)\mathbf{X}\|_F^2 \\ &= \mathbf{H}(W_1, W_2)\mathbf{X}\mathbf{X}^\top W_1^\top \\ &= \left(\mathbf{I} - \mathbf{W_2W_1}\right)\mathbf{X}\mathbf{X}^\top W_1^\top \end{aligned}$$

Similarly, gradient w.r.t W_1 gives

$$\begin{aligned} \nabla_{W_1}\mathcal{L} &= \nabla_{W_1}\frac{1}{n}\|\mathbf{H}(W_1, W_2)\mathbf{X}\|_F^2 \\ &= \mathbf{H}(W_1, W_2)\mathbf{X}\mathbf{X}^\top W_2 \\ &= \left(\mathbf{I} - \mathbf{W_2W_1}\right)\mathbf{X}\mathbf{X}^\top W_2 \end{aligned}$$

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

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

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

Solution: Using the principal components (top-k), we have $W_2 = W_1^\top = \mathbf{U}_k = [\mathbf{u}_1, ..., \mathbf{u}_k]$, 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 $I - W_2 W_1$, we have

$$\begin{aligned} \mathbf{I} - \mathbf{W_2}\mathbf{W_1} &= \mathbf{I} - \mathbf{U_k}\mathbf{U_k^\top} && (7) \\ &= \begin{bmatrix} \mathbf{0}_{k,k} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{m-k,m-k} \end{bmatrix} && (8) \end{aligned}$$

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

$$\Big(\mathbf{X} - \mathbf{W_2}\mathbf{W_1}\mathbf{X}\Big)\mathbf{X}^\top W_1^\top = \Big(\mathbf{I} - \mathbf{W_2}\mathbf{W_1}\Big)\mathbf{X}\mathbf{X}^\top W_1^\top \qquad (9)$$

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

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

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

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

5. Beam Search

This problem will also be covered in discussion.

When making predictions with an autoregressive sequence model, it can be intractable to decode the true most likely sequence of the model, as doing so would require exhaustively searching the tree of all $O(M^T)$ possible sequences, where M is the size of our vocabulary, and T is the max length of a sequence. We could decode our sequence by greedily decoding the most likely token each timestep, and this can work to some extent, but there are no guarantees that this sequence is the actual most likely sequence of our model.

Instead, we can use beam search to limit our search to only candidate sequences that are the most likely so far. In beam search, we keep track of the k most likely predictions of our model so far. At each timestep, we expand our predictions to all of the possible expansions of these sequences after one token, and then we keep only the top k of the most likely sequences out of these. In the end, we return the most likely sequence out of our final candidate sequences. This is also not guaranteed to be the true most likely sequence, but it is usually better than the result of just greedy decoding.

The beam search procedure can be written as the following pseudocode:

Algorithm 1 Beam Search

```
for each time step $t$ do
    for each hypothesis $y_{1:t-1,i}$ that we are tracking do
        find the top $k$ tokens $y_{t,i,1}, \ldots, y_{t,i,k}$
    end for
    sort the resulting $k^2$ length $t$ sequences by their total log-probability
    store the top $k$
    advance each hypothesis to time $t+1$
end for
```

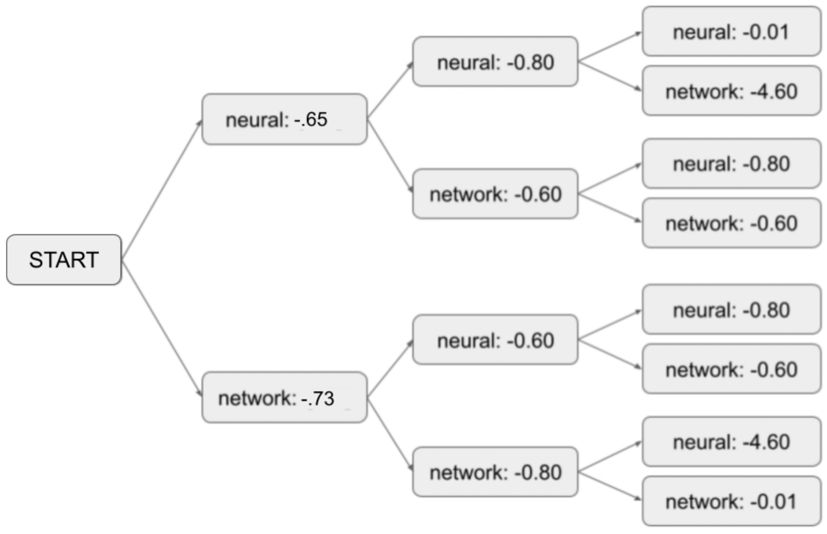

Figure 2: The numbers shown are the decoder's log probability prediction of the current token given previous tokens.

We are running the beam search to decode a sequence of length 3 using a beam search with $k = 2$. Consider predictions of a decoder in Figure 2, where each node in the tree represents the next token **log probability** prediction of one step of the decoder conditioned on previous tokens. The vocabulary consists of two words: "neural" and "network".

(a) **At timestep 1, which sequences is beam search storing?** **Solution:** There are only two options, so our beam search keeps them both: "neural" (log prob = -.65) and "network" (log prob = -.73).

(b) **At timestep 2, which sequences is beam search storing?** **Solution:** We consider all possible two word sequences, but we then keep only the top two, "neural network" (with log prob = -.65 - .6 = -1.25) and "network neural" (with log prob = -.73 - .6 = -1.33).

(c) **At timestep 3, which sequences is beam search storing?** **Solution:** We consider three word sequences that start with "neural network" and "network neural", and the top two are "neural network network" (with log prob = -.65 - .6 - .6 = -1.85) and "network neural network" (with log prob = -.73 - .6 - .6= -1.93).

(d) **Does beam search return the overall most-likely sequence in this example? Explain why or why not.** **Solution:** No, the overall most-likely sequence is "neural neural neural" with log prob = -.65 - .8

- .01= -1.46). These sequences don't get returned since they get eliminated from consideration in step 2, since "neural neural" is not in the $k = 2$ most likely length-2 sequences.

(e) **What is the runtime complexity of generating a length-T sequence with beam size k with an RNN?** Answer in terms of T and k and M. (Note: an earlier version of this question said to write it in terms of just T and k. This answer is also acceptable.) **Solution:**

- Step RNN forward one step for one hypothesis = $O(M)$ (since we compute one logit for each vocab item, and none of the other RNN operations rely on M, T, or K).
- Do the above, and select the top k tokens for one hypothesis. We do this by sorting the logits: $O(M \log M)$. (Note: there are more efficient ways to select the top K, for instance using a min heap. We just use this way since the code implementation is simple.) Combined with the previous step, this is $O(M \log M + M) = O(M \log M)$.
- Do the above for all k current hypotheses $O(KM \log M)$.
- Do all above + choose the top K of the K^2 hypotheses currently stored: we do this by sorting the array of K^2 items: $O(K^2 \log(K^2)) = O(K^2 \log(K))$ (since $\log(K^2) = 2\log(K)$). (Note: there are also more efficient ways to do this). Combining this with the previous steps, we get $O(KM \log M + K^2 \log(K))$. When one term is strictly larger than another we can take the max of the two. Since $M \geq K$, we could also write this as $O(KM \log M)$.
- Repeat this for T timesteps: $O(TKM \log M)$.

6. 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.

q_ae_solution

October 25, 2022

1 Autoencoders

This notebook walks you through different design choices for AutoEncoders, pretraining models with unsupervised learning (proxy tasks), and evaluating the learned representations with a linear classifier. In particular, we examine the following architectures: 1. Vanilla Autoencoder 2. Denoising Autoencoder 3. Masked Autoencoder

Fill in the `TODO`s in the notebook to run experiments, report your findings in the written assignment.

[ ]:
```
import numpy as np
import torch
import torchvision
from torch import nn, optim

from tqdm import tqdm
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
sns.set_style('whitegrid')
```

[ ]:
```
class SyntheticDataset:
    """
    Create a torch compatible dataset by sampling
    features from a multivariate normal distribution with
    specified mean and covariance matrix. In particular,
    the covariance is high along a small fraction of the directions.
    """
    def __init__(self,
                 input_size,
                 samples=10000,
                 splits=None,
                 num_high_var_dims=2,
                 var_scale=100,
                 batch_size=100):
        """
        input_size: (int) size of inputs
        samples: (int) number of samples to generate
        splits: list(float) of splitting dataset for [#train, #valid, #test]
        num_high_var_dims : (int) #dimensions with scaled variance
        var_scale : (float)
        """
        train_split, valid_split, test_split = splits
        self.input_size = input_size
        self.samples = samples
        self.num_high_var_dims = num_high_var_dims
        self.var_scale = var_scale
        self.batch_size = batch_size
        self.num_train_samples = int(samples * train_split)
        self.num_valid_samples = int(samples * valid_split)
        self.num_test_samples = int(samples * test_split)
        self._build()

    def _build(self):
        """
        Covariance is scaled along num_high_var_dims.
        Create torch compatible dataset.
        """
        self.mean = np.zeros(self.input_size)
        self.cov = np.eye(self.input_size)
        self.cov[:self.num_high_var_dims, :self.num_high_var_dims] *= self.
↪var_scale
        self.X = np.random.multivariate_normal(self.mean, self.cov, self.
↪samples)

        # generate random rotation matrix with SVD
        u, _, v = np.linalg.svd(np.random.randn(self.input_size, self.
↪input_size))
        sample = self.X @ u

        # create classification labels that depend only on the high-variance␣
↪dimensions
        target = self.X[:, :self.num_high_var_dims].sum(axis=1) > 0

        self.train_sample = torch.from_numpy(sample[:self.num_train_samples]).
↪float()
        self.train_target = torch.from_numpy(target[:self.num_train_samples]).
↪long()

        # create validation set
        valid_sample_end = self.num_train_samples+self.num_valid_samples
        self.valid_sample = torch.from_numpy(
            sample[self.num_train_samples:valid_sample_end]).float()
        self.valid_target = torch.from_numpy(
            target[self.num_train_samples:valid_sample_end]).long()
        # create test set
        self.test_sample = torch.from_numpy(sample[valid_sample_end:]).float()
        self.test_target = torch.from_numpy(target[valid_sample_end:]).long()

    def __len__(self):
        return self.samples

    def get_num_samples(self, split="train"):
        if split == "train":
            return self.num_train_samples
        elif split == "valid":
            return self.num_valid_samples
        elif split == "test":
            return self.num_test_samples

    def get_batch(self, batch_idx, split="train"):
        start_idx = batch_idx * self.batch_size
        end_idx = start_idx + self.batch_size

        if split == "train":
            return self.train_sample[start_idx:end_idx], self.
↪train_target[start_idx:end_idx]
        elif split == "valid":
            return self.valid_sample[start_idx:end_idx], self.
↪valid_target[start_idx:end_idx]
        elif split == "test":
            return self.test_sample[start_idx:end_idx], self.
↪test_target[start_idx:end_idx]
```

[ ]:
```
class MNIST:
    def __init__(self, batch_size, splits=None, shuffle=True):
        """
        Args:
          batch_size : number of samples per batch
          splits : [train_frac, valid_frac]
          shuffle : (bool)
        """
        # flatten the images
        self.transform = torchvision.transforms.Compose(
            [torchvision.transforms.ToTensor(),
             torchvision.transforms.Lambda(lambda x: x.view(-1))])

        self.batch_size = batch_size
        self.splits = splits
        self.shuffle = shuffle
        self._build()

    def _build(self):
        train_split, valid_split = self.splits
        trainset = torchvision.datasets.MNIST(
                root="data", train=True, download=True, transform=self.
↪transform)
        num_samples = len(trainset)
        self.num_train_samples = int(train_split * num_samples)
        self.num_valid_samples = int(valid_split * num_samples)

        # create training set
        self.train_dataset = torch.utils.data.Subset(
            trainset, range(0, self.num_train_samples))
        self.train_loader = torch.utils.data.DataLoader(
            self.train_dataset,
            batch_size=self.batch_size,
            shuffle=self.shuffle,
        )

        # create validation set
        self.valid_dataset = torch.utils.data.Subset(
            trainset, range(self.num_train_samples, num_samples))
        self.valid_loader = torch.utils.data.DataLoader(
            self.valid_dataset,
            batch_size=self.batch_size,
            shuffle=self.shuffle,
        )

        # create test set
        self.test_loader = torch.utils.data.DataLoader(
            torchvision.datasets.MNIST(
                root="data", train=False, download=True, transform=self.
↪transform
            ),
            batch_size=self.batch_size,
            shuffle=False,
        )
        self.num_test_samples = len(self.test_loader.dataset)

    def get_num_samples(self, split="train"):
        if split == "train":
            return self.num_train_samples
        elif split == "valid":
            return self.num_valid_samples
        elif split == "test":
            return self.num_test_samples
    def get_batch(self, idx, split="train"):
        if split == "train":
            return next(iter(self.train_loader))
        elif split == "valid":
            return next(iter(self.valid_loader))
        elif split == "test":
            return next(iter(self.test_loader))
```

[ ]:
```
class Autoencoder(nn.Module):
    """
    Autoencoder defines a general class of NN architectures
           _____________                                     ___________
          |             |                                   |           |      ␣
↪
    x --> |   ENCODER   | --> z (latent representation) --> |  DECODER  | -->␣
↪x'
          |_____________|                                   |___________|      ␣
↪

    We implement a generic autoencoder with a fully connected encoder and␣
↪decoder.
    The encoder and decoder are defined by a list of hidden layer sizes.
    Note that while this architecture is "symmetric" in dimensionality,
    the encoder and decoder can have different architectures.
    """
    def __init__(self, input_size, hidden_sizes, activation=nn.ReLU):
        super().__init__()
        self.input_size = input_size
        self.hidden_sizes = hidden_sizes
        self.activation = activation
        self.encoder = self._build_encoder()
        self.decoder = self._build_decoder()

    def _build_encoder(self):
        layers = []
        prev_size = self.input_size
        for layer_id, size in enumerate(self.hidden_sizes):
            layers.append(nn.Linear(prev_size, size))
            if layer_id < len(self.hidden_sizes)-1:
                layers.append(self.activation())
            prev_size = size
        return nn.Sequential(*layers)

    def _build_decoder(self):
        layers = []
        prev_size = self.hidden_sizes[-1]
        for size in reversed(self.hidden_sizes[:-1]):
            layers.append(nn.Linear(prev_size, size))
            layers.append(self.activation())
            prev_size = size
        layers.append(nn.Linear(prev_size, self.input_size))
        return nn.Sequential(*layers)

    def forward(self, x):
        ############################################
        #### TODO : implement forward pass
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat

    def get_loss(self, x):
        x_hat = self(x)
        return self.loss(x, x_hat)

    def encode(self, x):
        return self.encoder(x)

    def decode(self, z):
        return self.decoder(z)

    def loss(self, x, x_hat):
        return nn.functional.mse_loss(x, x_hat)

    def train_step(self, x, optimizer):
        x_hat = self(x)
        loss = self.loss(x, x_hat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        return loss


# denoising autoencoder
class DenoisingAutoencoder(Autoencoder):
    def __init__(self, input_size, hidden_sizes, activation=nn.ReLU,␣
 ↪noise_std=0.5):
        super().__init__(input_size, hidden_sizes, activation)
        self.noise_std = noise_std

    def train_step(self, x, optimizer):
        ############ TODO ############
        ## add Gaussian noise to input (of scale noise_std)
        x_noisy = x + self.noise_std * torch.randn_like(x)
        x_hat = self(x_noisy)
        loss = self.loss(x, x_hat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        return loss


# masked autoencoder
class MaskedAutoencoder(Autoencoder):
    def __init__(self, input_size, hidden_sizes, activation=nn.ReLU,␣
 ↪mask_prob=0.25):
        super().__init__(input_size, hidden_sizes, activation)
        self.mask_prob = mask_prob

    def train_step(self, x, optimizer):
        ############ TODO ############
        # create a mask that takes value 0 with probability mask_prob
        mask = torch.rand(x.shape) > self.mask_prob
        # Mask the input
        x_masked = x * mask
        x_hat = self(x_masked)
        loss = self.loss(x, x_hat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        return loss
```

[ ]:
```
# training/plotting utilities

def plotter(train_losses, test_losses, savepath=None):
    plt.figure(figsize=(12, 8))

    MODELS = train_losses.keys()
    num_models = len(MODELS)
    colors = cm.rainbow(np.linspace(0, 1, num_models))
    for idx, model in enumerate(MODELS):
        plt.plot(
            train_losses[model],
            linewidth=2,
            label=model,
            color=colors[idx])
        plt.plot(
            test_losses[model],
            linewidth=2,
            linestyle='-.',
            label=model + "_test",
            color=colors[idx])
    plt.xlabel("#Epochs", fontsize=16)
    plt.ylabel("Avg. Loss", fontsize=16)
    plt.title("Comparing training w/ different AE architectures", fontsize=24)
    plt.legend(fontsize=14)
    if savepath:
        plt.savefig(savepath)


class Experiment:
    def __init__(self, dataset, model, batch_size=100, num_classes=2):
        self.batch_size = batch_size
        self.dataset = dataset
        self.model = model
        self.optimizer = optim.Adam(self.model.parameters(), lr=5e-4)
        self.num_classes = num_classes

    def train(self, num_epochs=100):
        train_losses, valid_losses = [], []

        pbar = tqdm(range(num_epochs))
        num_batches = self.dataset.num_train_samples // self.batch_size

        for epoch in pbar:
            for batch_idx in range(num_batches):
                x, y = self.dataset.get_batch(batch_idx, split="train")
                loss = self.model.train_step(x, self.optimizer)

                # update progress
                pbar.set_description(f"Epoch {epoch}, Loss {loss.item():.4f}")

            # evaluate loss on valid dataset
            train_loss = self.get_loss(split="train")
            valid_loss = self.get_loss(split="valid")

            train_losses.append(train_loss)
            valid_losses.append(valid_loss)
        return {"train_losses": train_losses, "valid_losses": valid_losses}

    def get_loss(self, split="train"):
        num_samples = self.dataset.get_num_samples(split=split)
        num_batches = num_samples // self.batch_size
        losses = []
        for batch_idx in range(num_batches):
            x, y = self.dataset.get_batch(batch_idx, split=split)
            loss = self.model.get_loss(x)
            losses.append(loss.item())
        return np.mean(losses)

    def get_model_accuracy(self, classifier, split="test"):
        """
        Compute the model accuracy with a linear classifer.
        """
        num_samples, num_correct = 0, 0
        num_batches = self.dataset.num_test_samples // self.batch_size
        for batch_idx in range(num_batches):
            x, y = self.dataset.get_batch(batch_idx, split="test")
            z = self.model.encode(x)
            y_hat = classifier(z)
            preds = (y_hat.argmax(dim=1) == y).numpy()
            num_samples += len(preds)
            num_correct += np.sum(preds)
        return num_correct / num_samples

    def evaluate_w_linear_probe(self, feats_dim, num_epochs=10):
        # create a classifier
        probe = nn.Linear(feats_dim, self.num_classes)

        # setup optimizer
        probe_opt = optim.Adam(probe.parameters(), lr=1e-3)

        # train linear probe
        # note that we train on a small subset of the labelled data
        pbar = tqdm(range(num_epochs))
        num_batches = self.dataset.num_valid_samples // self.batch_size

        for epoch in pbar:
            for batch_idx in range(num_batches):
                x, y = self.dataset.get_batch(batch_idx, split="train")
                feat = self.model.encode(x)
                y_hat = probe(feat)

                # compute loss, optimize
                loss = nn.functional.cross_entropy(y_hat, y)
                probe_opt.zero_grad()
                loss.backward()
                probe_opt.step()

                # update progress
                pbar.set_description(
                    f"Epoch {epoch}, Loss {loss.item():.4f}")

        # evaluate linear probe
        accuracy = self.get_model_accuracy(classifier=probe)
        return {"accuracy": accuracy}
```

1.1 Linear AutoEncoders on Synthetic Dataset

Train a linear autoencoder with different bottleneck sizes (5, 10, 100, 1000) and report performance of each architecture on the synthetic dataset. (Feel free to explore other architectures/adding more layers)

[ ]:
```
MODELS = {
    "vanilla": Autoencoder,
    "denoise": DenoisingAutoencoder,
    "masking": MaskedAutoencoder,
}

# we repeat each experiment and report mean performance
NUM_REPEATS = 3
NUM_EPOCHS = 10
BATCH_SIZE = 100

# load and create Synthetic dataset
INPUT_DIMS = 100
NUM_SAMPLES = 20000
DATA_SPLITS = [0.7, 0.2, 0.1]  # train, valid, test
NUM_HIGH_VAR_DIMS = 20
VAR_SCALE = 100
NUM_CLASSES = 2
dataset = SyntheticDataset(
    INPUT_DIMS, NUM_SAMPLES, DATA_SPLITS, NUM_HIGH_VAR_DIMS, VAR_SCALE,␣
 ↪BATCH_SIZE)

##########################################
####### TODO : define model architecture
# hidden dims is a list of ints
HIDDEN_DIMS = 10
ACTIVATION = nn.Identity
##########################################

# logging metrics
feats_dim = HIDDEN_DIMS[-1]
train_losses, test_losses = {}, {}
accuracy = {}

# run experiment w/ different models
for model_idx, model_cls in MODELS.items():
    _train_loss, _valid_loss, _acc = [], [], []
    for expid in range(NUM_REPEATS):
        print("run : {},  model : {}".format(expid, model_idx))
        model = model_cls(INPUT_DIMS, HIDDEN_DIMS, activation=ACTIVATION)
        experiment = Experiment(dataset, model)
        train_stats = experiment.train(num_epochs=NUM_EPOCHS)
        eval_stats = experiment.evaluate_w_linear_probe(feats_dim, NUM_EPOCHS)
        _train_loss.append(train_stats["train_losses"])
        _valid_loss.append(train_stats["valid_losses"])
        _acc.append(eval_stats["accuracy"])

    train_losses[model_idx] = np.mean(_train_loss, axis=0)
    test_losses[model_idx] = np.mean(_valid_loss, axis=0)
    accuracy[model_idx] = np.mean(_acc)

# plot losses
plotter(train_losses, test_losses)

# report accuracy
for model_idx, acc in accuracy.items():
    print("Model : {}, Accuracy : {}".format(model_idx, acc))
```

1.2 Nonlinear AutoEncoders on Synthetic Dataset

Train a nonlinear autoencoder with different bottleneck sizes (5, 10, 100, 1000) and report performance of each architecture on the synthetic dataset.

[ ]:
```
MODELS = {
    "vanilla": Autoencoder,
    "denoise": DenoisingAutoencoder,
    "masking": MaskedAutoencoder,
}

NUM_REPEATS = 3
NUM_EPOCHS = 10
BATCH_SIZE = 100

# load and create Synthetic dataset
INPUT_DIMS = 100
NUM_SAMPLES = 20000
DATA_SPLITS = [0.7, 0.2, 0.1]  # train, valid, test
NUM_HIGH_VAR_DIMS = 20
VAR_SCALE = 100
NUM_CLASSES = 2
dataset = SyntheticDataset(
    INPUT_DIMS, NUM_SAMPLES, DATA_SPLITS, NUM_HIGH_VAR_DIMS, VAR_SCALE,␣
 ↪BATCH_SIZE)

###########################################
####### TODO : define model architecture
```

```
# hidden dims is a list of ints
HIDDEN_DIMS = 10  ## 5, 10, 100, 1000
ACTIVATION = nn.ReLU  ## any nonlinear activation
##########################################

# logging metrics
feats_dim = HIDDEN_DIMS[-1]
train_losses, test_losses = {}, {}
accuracy = {}

# run experiment w/ different models
for model_idx, model_cls in MODELS.items():
    _train_loss, _valid_loss, _acc = [], [], []
    for expid in range(NUM_REPEATS):
        print("run : {},  model : {}".format(expid, model_idx))
        model = model_cls(INPUT_DIMS, HIDDEN_DIMS, activation=ACTIVATION)
        experiment = Experiment(dataset, model)
        train_stats = experiment.train(num_epochs=NUM_EPOCHS)
        eval_stats = experiment.evaluate_w_linear_probe(feats_dim, NUM_EPOCHS)
        _train_loss.append(train_stats["train_losses"])
        _valid_loss.append(train_stats["valid_losses"])
        _acc.append(eval_stats["accuracy"])

    train_losses[model_idx] = np.mean(_train_loss, axis=0)
    test_losses[model_idx] = np.mean(_valid_loss, axis=0)
    accuracy[model_idx] = np.mean(_acc)

# plot losses
plotter(train_losses, test_losses)

# report accuracy
for model_idx, acc in accuracy.items():
    print("Model : {}, Accuracy : {}".format(model_idx, acc))
```

1.3 Linear AutoEncoders on MNIST

Train a linear autoencoder with different bottleneck sizes (5, 10, 100, 1000) and report performance of each architecture.

[ ]:
```
MODELS = {
    "VANILLA AE": Autoencoder,
    "DENOISING AE": DenoisingAutoencoder,
    "MASKED AE": MaskedAutoencoder,
}

NUM_REPEATS = 3
NUM_EPOCHS = 10
BATCH_SIZE = 200
```

```
###########################################
####### TODO :  define model architecture
# hidden dims is a list of ints
INPUT_DIMS = 784
NUM_CLASSES = 10
ACTIVATION = nn.Identity
HIDDEN_DIMS = 10  ## 5, 10, 100, 1000
###########################################

# load and create MNIST dataset
dataset = MNIST(BATCH_SIZE, [0.6, 0.4])

# logging metrics
feats_dim = HIDDEN_DIMS[-1]
train_losses, test_losses = {}, {}
accuracy = {}

# run experiment w/ different models
for model_idx, model_cls in MODELS.items():
    _train_loss, _valid_loss, _acc = [], [], []
    for expid in range(NUM_REPEATS):
        print("run : {},  model : {}".format(expid, model_idx))
        model = model_cls(INPUT_DIMS, HIDDEN_DIMS, activation=ACTIVATION)
        experiment = Experiment(dataset, model, num_classes=NUM_CLASSES)
        train_stats = experiment.train(num_epochs=NUM_EPOCHS)
        eval_stats = experiment.evaluate_w_linear_probe(feats_dim, NUM_EPOCHS)
        _train_loss.append(train_stats["train_losses"])
        _valid_loss.append(train_stats["valid_losses"])
        _acc.append(eval_stats["accuracy"])
    print(np.array(_train_loss).shape)

    train_losses[model_idx] = np.mean(_train_loss, axis=0)
    test_losses[model_idx] = np.mean(_valid_loss, axis=0)
    accuracy[model_idx] = np.mean(_acc)

# plot losses
plotter(train_losses, test_losses)

# report accuracy
for model_idx, acc in accuracy.items():
    print("Model : {}, Accuracy : {}".format(model_idx, acc))
```

1.4 Nonlinear dimensionality reduction on MNIST

Train a nonlinear autoencoder with different bottleneck sizes e.g. (5, 10, 100, 1000) and report performance of your choice of architecture on the MNIST dataset.

[ ]:
```python
MODELS = {
    "VANILLA AE": Autoencoder,
    "DENOISING AE": DenoisingAutoencoder,
    "MASKED AE": MaskedAutoencoder,
}

NUM_REPEATS = 5
NUM_EPOCHS = 10
BATCH_SIZE = 200

##########################################
####### TODO : define model architecture
# hidden dims is a list of ints
INPUT_DIMS = 784
NUM_CLASSES = 10
ACTIVATION =  nn.ReLU  ## any nonlinear activation
HIDDEN_DIMS = 10  ## 5, 10, 100, 1000
##########################################

# load and create MNIST dataset
dataset = MNIST(BATCH_SIZE, [0.6, 0.4])

# logging metrics
feats_dim = HIDDEN_DIMS[-1]
train_losses, test_losses = {}, {}
accuracy = {}

# run experiment w/ different models
for model_idx, model_cls in MODELS.items():
    _train_loss, _valid_loss, _acc = [], [], []
    for expid in range(NUM_REPEATS):
        print("run : {},  model : {}".format(expid, model_idx))
        model = model_cls(INPUT_DIMS, HIDDEN_DIMS, activation=ACTIVATION)
        experiment = Experiment(dataset, model, num_classes=NUM_CLASSES)
        train_stats = experiment.train(num_epochs=NUM_EPOCHS)
        eval_stats = experiment.evaluate_w_linear_probe(feats_dim, NUM_EPOCHS)
        _train_loss.append(train_stats["train_losses"])
        _valid_loss.append(train_stats["valid_losses"])
        _acc.append(eval_stats["accuracy"])

    train_losses[model_idx] = np.mean(_train_loss, axis=0)
    test_losses[model_idx] = np.mean(_valid_loss, axis=0)
    accuracy[model_idx] = np.mean(_acc)

# plot losses
plotter(train_losses, test_losses)
```

```python
# report accuracy
for model_idx, acc in accuracy.items():
    print("Model : {}, Accuracy : {}".format(model_idx, acc))
```

Contributors:

- Jerome Quenum.
- Olivia Watkins.
- Anant Sahai.
- Sheng Shen.
- David M. Chan.
- Shaojie Bai.
- Angelos Katharopoulos.
- Hao Peng.
- Kumar Krishna Agrawal.
- CS 182 Staff from past semesters.