CS 182/282A Deep Neural Networks **Fall 2022**

Lecture 15: Self-Supervision and Autoencoders

Oct 13, 2022

Instructor: Anant Sahai *Scribes: Kiran Eiden, Lawrence Yunliang Chen*

15.1 Background: Unsupervised Learning

We start by recalling the main idea of unsupervised learning: given some data $\{\vec{x}_i\}$, discover an underlying pattern in the data. There are two basic types of unsupervised learning, which are:

1. Dimensionality reduction (e.g. principal component analysis). This is vaguely similar to regression, and the intent is to summarize or distill important information from the data. The algorithm we typically use for principal component analysis involves solving for a singular value decomposition, and thus is an eigenvalue computation-style algorithm.
2. Clustering (e.g. K-means). This is vaguely similar to classification, and the intent is to group related data points. K-means is typically solved using Lloyd's algorithm (Lloyd, 1982), which is an iterative alternating minimization-style algorithm.

We provide an overview of some of the possible uses of unsupervised learning in the two subsections.

15.1.1 Utilizing Unlabeled Data

The naive approach when doing supervised learning on datasets with unlabeled data is to simply discard the unlabeled data. Unsupervised learning allows one to take advantage of unlabeled data, and potentially improve upon the mapping found by only utilizing the supervised learning algorithm.

For example, imagine a large, high-dimensional dataset of dimension d with a small number of labeled points $n \ll d$. A pure supervised learning algorithm could not be applied to this dataset, as it would need to learn how to label data with d dimensions while only making use of the n labeled points. There are a couple of ways to resolve this using unsupervised learning:

- A dimensionality reduction algorithm can be applied to the entire dataset, including the unlabeled points. If the unsupervised learning algorithm is able to learn a mapping down to a low dimensional space with dimension $d' \leq n$, the supervised learning algorithm could potentially be applied to the low-dimensional representation of the dataset and successfully learn the proper labels.
- Similarly, clustering can be used to predict labels for unlabeled points in the dataset (assuming that points in the same cluster have similar labels). If the number of labeled points post-clustering $n' \geq d$, then the supervised learning algorithm can be applied to the new dataset with inferred labels.

For many problems in, for example, natural language processing and image processing, it is much easier to obtain unlabeled data than labeled data. The models often require large quantities of data to train, so it is

important to be able to utilize unlabeled data. In the context of deep learning and deep neural networks, this requires some extension of our unsupervised learning concepts like dimensionality reduction to work with gradient descent or another, similar approach.

15.1.2 Exploratory Data Analysis

Another possible use of unsupervised learning is in exploratory data analysis. High-dimensional data is difficult to understand and visualize, and sometimes one might want to use dimensionality reduction and clustering techniques to simplify the dataset for visualization purposes. This is also done in the context of deep learning, but will not be covered in this lecture.

15.2 Rethinking Dimensionality Reduction by PCA

Here we will only be considering principal component analysis (PCA) without removal of means. Let us compile all of our data into a data matrix

$$X = [\vec{x}_1, \vec{x}_2, ..., \vec{x}_n], \tag{15.1}$$

where each data point $\vec{x}_i \in \mathbb{R}^d$. Note that our data points are packed into X as columns rather than rows. We want to find a k-dimensional subspace S_k so that the average of the residual

$$||\vec{x}_i - P_{S_k}\vec{x}_i||^2 \tag{15.2}$$

is small, where $P_{S_k}\vec{x}_i$ is the projection of vector $\vec{x}_i$ onto the subspace S_k. We can write this in terms of the data matrix X and the Frobenius norm $||\cdot||_F$ as the minimization problem

$$\min ||X - P_{S_k}X||_F^2. \tag{15.3}$$

Classically, we would solve this problem by taking the singular value decomposition (SVD) of X. This can be written as

$$X = U\Sigma V^T \tag{15.4}$$

$$= \sum_{i=1}^{\min(d,n)} \sigma_i \vec{u}_i \vec{v}_i^T, \tag{15.5}$$

where the σ_i are our singular values and $\vec{u}_i$ and $\vec{v}_i$ are the rows of our unitary matrices U and V respectively. Our solution $\hat{S}_k$ is then given by

$$\hat{S}_k = \text{span}\,(\vec{u}_1, \vec{u}_2, ..., \vec{u}_k). \tag{15.6}$$

We can approximate X by truncating the sum over $\sigma_i \vec{u}_i \vec{v}_i^T$ at $i = k$.

This also gives us a simple definition for our projection. We can calculate $P_{\hat{S}_k}\vec{x}_i$ by defining a matrix

$$U_k = [\vec{u}_1, \vec{u}_2, ..., \vec{u}_k], \tag{15.7}$$

and then taking

$$P_{\hat{S}_k}\vec{x}_i = U_k U_k^T \vec{x}_i. \tag{15.8}$$

The k-dimensional representation $\vec{y}$ of $\vec{x}_i$ is just $U_k^T\vec{x}_i$, and $P_{\hat{S}_k}\vec{x}_i$ is a "reconstruction" of our original $\vec{x}_i$ in d-dimensional space. The transformation from $\vec{x}$ to its reconstruction is depicted in Figure 15.1.

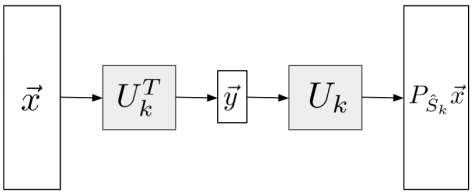

Figure 15.1: Dimensionality reduction and projection by PCA drawn as a block diagram.

We need to find a way to learn the mapping depicted in Figure 15.1 via gradient descent. This is discussed in the next section.

Note: If we replace X with X^T (i.e. if the data matrix contains row data instead of column data), we can see that U and V will be swapped in Equation 15.4. That means that for row data, our subspace in Equation 15.6 and matrix in Equation 15.7 would be defined by the rows of the V matrix from the SVD of X.

15.3 Autoencoders

We want gradient descent to be able to learn the map from $\vec{x}_i$ to $P_{\hat{S}_k}\vec{x}_i$. Matching the definition of U_k from PCA is not critical. There are multiple definitions of U_k that will produce the same $\hat{S}_k$ (consider permuting U_k and U_k^T, for example, or modifying U_k while preserving its column span). Furthermore, in the context of deep learning, we are not necessarily interested in learning an appropriate linear subspace, but some k-dimensional structure that might be defined by a non-linear mapping. We ultimately just need the k-dimensional representation to be useful for the purposes of our learning problem.

For further reference on autoencoders, see Goodfellow et al. (2016). Autoencoders are covered specifically in Chapter 14 of that book, which can be found online at `https://www.deeplearningbook.org/contents/autoencoders.html`.

15.3.1 Learning Problem Setup

Consider the setup shown in Figure 15.2. We take some input $\vec{x}$ and compress it down to a length k representation (bottleneck) called $\vec{y}$ via a linear map A. We then apply another linear map B to take it from the k-dimensional vector $\vec{y}$ to a reconstruction $\hat{\vec{x}}$.

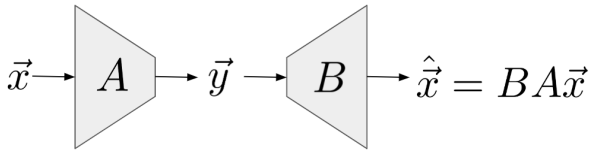

Figure 15.2: The basic form of an autoencoder.

We can complete the description of our learning problem by defining a loss function that is the mean-squared error of all $\vec{x}_i$ and $\hat{\vec{x}}_i$ in our dataset:

$$\mathcal{L}(\vec{x}, \hat{\vec{x}}) = \frac{1}{n}\sum_{i=1}^{n} \left|\left|\vec{x}_i - \hat{\vec{x}}_i\right|\right|^2 \tag{15.9}$$

$$= \frac{1}{n} \sum_{i=1}^{n} ||\vec{x}_i - BA\vec{x}_i||^2 . \tag{15.10}$$

This feels like a supervised learning problem. We are learning a mapping with parameters encapsulated in A and B that minimizes the difference between our reconstruction $\hat{\vec{x}}$ and our input $\vec{x}$. Specifically, it is a *self-supervised* learning problem, since our target is a function only of the input itself. Note that the identity is not expressible via this mapping since we pass through a bottleneck layer with dimension k, and the identity is a rank d linear map. The product BA can only produce a map of rank k, as A has dimensions $k \times d$ and B is $d \times k$. Thus we are learning an approximate reconstruction of our original data that minimizes the error for a given botteneck size k.

Note: The Eckart-Young-Mirsky theorem (Schmidt, 1907; Eckart and Young, 1936) tells us that the minimization problem described by Equation 15.3 for $\text{rank}(P_{S_k} X) \leq \text{rank}(X)$ always has a unique analytical solution in terms of the SVD of X. This motivates our approach to PCA. What we are doing here is effectively telling the computer that we know a solution exists and to go find our solution for us using gradient descent. In deep learning in general we often have less certainty – we tell the computer that we *hope* an answer exists to our minimization problem and that it has the representation we specified in our problem setup. We then ask the computer to go find it.

15.3.2 Non-Linearity and Autoencoder Approach

In general, we can replace A and B with non-linear encoder and decoder neural networks. This approach is called an *autoencoder* approach, since it encodes our input $\vec{x}$ in a low-dimensional representation $\vec{y}$ and learns how to reconstruct that input. Historically, autoencoder strategies were often used to first learn an initialization of the neural network weights, before training the network using the actual targets. This approach to neural network training was eventually replaced by end-to-end supervision, but has regained its footing in recent years as a pre-training approach for large models.

This pure form of autoencoder is not always an ideal way to solve a self-supervised learning problem, and many variations on the basic structure outlined here exist. Some of these are discussed in the subsequent section (Section 15.4).

For many practical problems, this basic form of autoencoder does not work so well because of the limited flexibility induced by the bottleneck. In particular, with this bottleneck, getting the training to work well for specific applications can be challenging, and in general, bigger networks will train more easily. So one may not want the data to go through the bottleneck but instead go through something bigger (i.e., allowing the dimension of $\vec{y}$ to be larger than that of $\vec{x}$). The issue, however, is that for a sufficiently large bottleneck layer the identity transformation will become a solution (i.e., the model learns the trivial matrices $BA = I$ and the latent representation $\vec{y}$ is not useful). We will discuss methods to address this issue in Section 15.5.

15.4 Parameterizations

In the previous section, we discussed the most basic form of an autoencoder, which we denote as **Parameterization 1**:

Parameterization 1:

The neural network contains two sets of learnable weights: matrices A and B, and they are independent.

From a classical point of view, one may argue that the only thing we need to learn is the subspace, which is the B matrix, and we do not need to learn the extra A matrix. Alternatively, one may argue that we only

need to learn the A matrix to compute the latent vector y and use the dimensionality reduction for other applications, while the B matrix is just there to set up the autoencoder. In fact, there are multiple ways to parameterize the learnable weights, which we discuss below.

15.4.1 Weight Sharing for A

We know that at optimality of Equation 15.10, $A = (B^TB)^{-1}B^T$ is the least square solution to achieve projection. This suggests we can do weight sharing between A and B.

Parameterization 2:

The neural network parameterizes the encoder weights as $A = (B^TB)^{-1}B^T$ and only learns the weight matrix B.

In this parameterization, the reconstructed $\hat{\vec{x}} = BA\vec{x} = B(B^TB)^{-1}B^T\vec{x}$. This is a differentiable function of the entries of B, and PyTorch can take gradients to learn B.

15.4.2 Partial Weight Sharing for A

In **Parameterization 2**, there is a complicated nonlinearity resulted from the matrix inverse $(B^TB)^{-1}$. Alternatively, one could replace the $(B^TB)^{-1}$ part by a learnable $k \times k$ matrix C. We thus get another parameterization:

Parameterization 3:

The neural network parameterizes the encoder weights as $A = CB^T$ and learns 2 weight matrices $B \in \mathbb{R}^{d\times k}$ and $C \in \mathbb{R}^{k\times k}$.

In this parameterization, the reconstructed $\hat{\vec{x}} = BA\vec{x} = BCB^T\vec{x}$, and the encoder and decoder share common weights B. As before, PyTorch can easily take gradients to learn both B and C.

15.4.3 Using the Inductive Bias of Gradient Descent

Apart from replacing $(B^TB)^{-1}$ with a learnable matrix C, we can also understand the inverse from another perspective. In general, taking the inverse of a matrix corresponds to solving some system of linear equations. From the perspective of deep learning, solving a system of linear equations is the same as minimizing a squared loss function. Here, the matrix $A = (B^TB)^{-1}B^T$ is the solution to the following least squares problem:

$$A\vec{x} = \arg\min_{\vec{y}} ||\vec{x} - B\vec{y}||^2 . \tag{15.11}$$

For deep learning, we can now solve A by gradient descent:

$$\vec{y}_0 = \vec{0}, \tag{15.12}$$

$$\vec{y}_{t+1} = \vec{y}_t + \eta B^T(\vec{x} - B\vec{y}_t). \tag{15.13}$$

In fact, we can draw out Equation 15.13 as a block diagram, as shown in Figure 15.3. $\vec{y}_t$ is first multiplied by $-B$, and then added to $\vec{x}$, before getting multiplied by ηB^T and added to $\vec{y}_t$.

Note that this block diagram looks like an recurrent neural network (RNN) block with a skip/residual connection. The blue dotted box in Figure 15.3 encompasses some computation where $\vec{y}_t$ is like a hidden

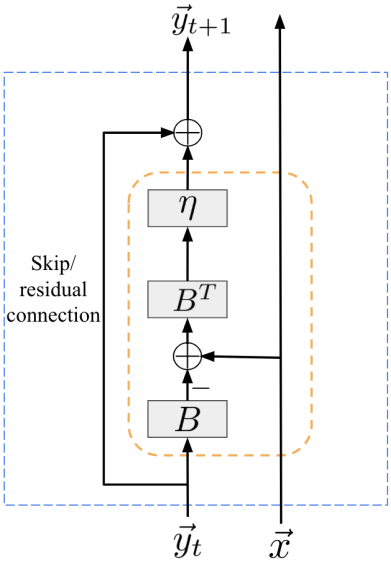

Figure 15.3: Equation 15.13 drawn as a block diagram.

state in an RNN that goes through the cell repeatedly at each time step, with an input $\vec{x}$ also fed into the cell. The orange dotted box encompasses some computation that learns the residual needed to add to $\vec{y}_t$.

With this observation, we can identify A as an infinite sequence of these computation blocks, where each pass through the RNN cell corresponds to a gradient step on $\vec{y}_t$, and after infinite gradient steps, $\vec{y}_t$ converges to the optimal solution of Equation 15.11, which is $A\vec{x}$. In this way, we unroll the gradient descent for learning the latent vector $\vec{y}$ as a deep neural network! Figure 15.4 depicts this deep neural network. We can summarize this as another parameterization method as follows.

Parameterization 4:

The neural network consists of a deep encoder in the form of an RNN with internal weights composed of B, and a linear decoder with weight B.

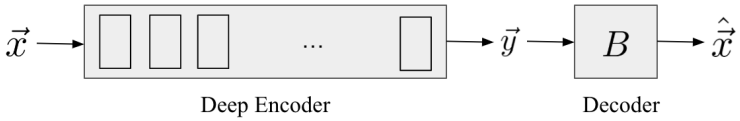

Figure 15.4: Autoencoder in the form of a deep RNN encoder and a linear decoder.

We note that the equivalence between $(B^TB)^{-1}B^T$ and an infinite sequence of the RNN cells is also related to the fact that a matrix inverse can be expressed as an infinite series. In practice, neural networks cannot be infinitely deep, so we just truncate it by setting a fixed depth (e.g., 10 layers).

We also note that **Parameterization 4** is just a variant of **Parameterization 2**. In fact, we can create other variants. For example, the RNN cell in Figure 15.4 does not need to look like Figure 15.3 — the weights B and B^T in Figure 15.3 can be some other learnable weights.

Comments: Dimensionality of $\vec{y}$ and motivation for data augmentation

- If $\dim(\vec{y}) = k < \dim(\vec{x}) = d$, none of **Parameterizations 1-4** will learn the identity transformation

if the model is linear. But if $\vec{x}$ has a low intrinsic dimension $d' < k < d$, it is possible for $\vec{y}$ to learn to contain enough information to fully recover $\vec{x}$, especially if there are bias terms or nonlinear layers. And that may be what we want, as $\vec{y}$ learns to capture the intrinsic dimension of $\vec{x}$.

- However, there are still some potential issues with the basic autoencoder approach (**Parameterizations 1-3**) of minimizing Equation 15.10. This is because there may be many choices for what $\vec{y}$ can be. In particular, $\vec{y}$ may contain all the information necessary to reconstruct $\vec{x}$, but it may also contain some other spurious information. And while the model may reconstruct $\vec{x}$ on the training data well, it may spuriously rely on the noises in $\vec{y}$ so that the reconstructions become incorrect and actually amplify the noise if $\vec{x}$ is noisy.

- In contrast, for **Parameterization 4**, usually B will learn to have some large singular values and some small singular values. Because of the implicit regularization effect of gradient descent, with early stopping, the latent vector $\vec{y}$ will adapt to move in the direction of large singular directions and not move much in the direction corresponding to small singular values. This has the interpretation that $\vec{y}$ does not move far in the badly-conditioned directions (since the network is not infinitely deep). In other words, it is more robust to noise. On the other hand, **Parameterizations 1-3** do not have such inductive biases built into the architecture.

- This motivates the approach of further introducing inductive biases through **data augmentation**, which we will discuss in the next section. In particular, data augmentation can encourage the model to learn $\vec{y}$ to preserve and strengthen the true signal in $\vec{x}$ and ignore (and ideally remove) false signals/noises in $\vec{x}$.

- If $\dim(\vec{y}) = k \geq \dim(\vec{x}) = d$, the 4 parameterization approaches need to be adjusted slightly:

  - **Parameterization 1:** No change needed. We still have $A \in \mathbb{R}^{k \times d}$ and $B \in \mathbb{R}^{d \times k}$.
  - **Parameterization 2:** The minimum norm solution to Equation 15.10 is now $A = B^T(BB^T)^{-1}$. So now $BA = BB^T(BB^T)^{-1} = I$!
  - **Parameterization 3:** Similar to before, we can replace the $(BB^T)^{-1}$ part by a learnable $d \times d$ matrix C. So now $A = B^TC$, and $BA = BB^TC$.
  - **Parameterization 4:** No change needed. We still solve Equation 15.11, and while the optimal solution changes to $A = B^T(BB^T)^{-1}$ when B becomes a wide matrix, the gradient descent formula Equation 15.13 does not change! Thus, the RNN remains the same, and this parameterization does not care whether B is tall or wide.

  ⋆ In this case, we see that **Parameterization 2** will always learn the identity transform, while **Parameterizations 1 and 3** *may* learn an identity transform (e.g., if it learns $C = (BB^T)^{-1}$). **Parameterization 4**, however, will likely not learn the identity transform because of early stopping of the gradient descent (recall that the neural network is not infinitely deep). This is another attractive advantage of **Parameterization 4** over the others.

15.5 "Excorcising" the Fear of Learning the Identity

As noted in the previous comment, when $\dim(\vec{y}) = k \geq \dim(\vec{x}) = d$, it is possible for the network to learn the identity transformation. One way to deal with this is data augmentation, where we change the input to be different from the target. With this change, the identity transform is no longer the optimal solution, and the hope is that the network will not learn the identity transform (except for **Parameterization 2**, which has no choice but to learn the identity).

15.5.1 Use Data Augmentation: Denoising Autoencoder

Recall data augmentation in computer vision, where we modify the images (e.g. rotate, crop, brighten) but keep the target labels the same. Here, the data augmentation we do is adding noise. We keep the target $\vec{x}$ the same but change the input to be $\vec{x} + \vec{n}$, where $\vec{n} \overset{\text{iid}}{\sim} N(0, \sigma^2)$.

Note that for the bottleneck architecture ($k < d$), the correct solution (Equation 15.6) has the property of averaging out noise. This is because the noise of the input, which is d-dimensional, has about $d\sigma^2$ energy. After projecting down to a k-dimensional subspace, the total energy of the noise is about $k\sigma^2$. This means that, the output $\hat{\vec{x}}$ only has about $k\sigma^2$ noise. Therefore, this denoising data augmentation creates an inductive bias to encourage the model to get rid of noise or average it out.

Also, as pointed out in the previous comments, for the $k \geq d$ case, **Parameterization 4** can achieve successful denoising with a deep (but not infinitely deep) neural network by learning a matrix B that has large and small singular values (so $\vec{y}$ learns to move in the well-conditioned direction and ignore the noises).

15.5.2 Masking/Inpainting: Kind of Data Augmentation

Another approach of data augmentation is masking/inpainting. Instead of adding noise, we remove entries from the input. For example, given a data point $\vec{x} = [x_1, x_2, x_3, x_4, x_5]^T$, we mask the input and feed into the neural network $[x_1, ?, x_3, x_4, ?]^T$, and ask the network to reconstruct $\vec{x}$. Again, the identity transformation is no longer the optimum. But suppose the underlying structure of $\vec{x}$ is 1-dimensional, it is possible for the model to learn to predict x_2 and x_5 from the other entries.

As we see, learning a low-dimensional subspace allows us to do many tasks, including denoising, straight autoencoding, as well as filling the blanks. Because the underlying structure supports many tasks, we can do any of these tasks for self-supervision, and it is useful to do so.

In practice, we need to think about how to implement masking. The simplest choice of putting in 0's does not work, as 0 is not a mask and will want to be reconstructed as 0. We will talk about this in the next lecture, and we will see that the architecture shown in Figure 15.4 is more suitable for masking than that in Figure 15.2.

15.6 What we wish this lecture also had to make things clearer?

1. In lecture, we used Figure 15.1 to describe PCA in terms of a linear map U_k derived from SVD. We then discussed how we might want to add some non-linearity (and indeed, how that is much of the point of moving to deep learning). However, when we went to describe the autoencoder, our autoencoder structure was effectively the same as Figure 15.1 and still utilized linear maps. We just replaced U_k and U_k^T with the matrices A and B with learnable parameters. It might make more sense to describe the autoencoder structure in terms of arbitrary non-linear maps A and B with learnable parameters, and then make any arguments we need to make about the learning problem (e.g. whether we can learn the identity) in terms of general non-linear maps. Alternatively, it might make more sense to discuss how we want to generalize to non-linear maps after writing down the basic autoencoder structure in terms of linear maps rather than before.

2. It would be clearer to summarize the contrast of the 4 parameterizations in each of the settings individually ($k < d$ and $k \geq d$ and with and without data augmentation) instead of talking about them at the end when multiple tweaks have been introduced. For example, currently, the $k \geq d$ case is discussed together with the denoising autoencoder, and when talking about the denoising effect of

Parameterization 4, it is not immediately clear which component is the main contributing factor, i.e., whether it comes from $k < d$ or the early stopping of gradient descent or the denoising data augmentation task. Similar for the disadvantages of the other parameterizations: under precisely what settings do they work or not work.

3. Some demos or plots showing different behaviors when the latent dimension (k) is smaller/larger compared to the input dimension (d) will be really helpful for interpreting the expected results.

References

Eckart, C. and Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1:211–218.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137.

Schmidt, E. (1907). Zur theorie der linearen und nichtlinearen integralgleichungen. *Mathematische Annalen*, 63:433–476.