CS 189 Introduction to Machine Learning
Fall 2017

HW14

This homework is due **Thursday, December 7 at 10pm.**

1 Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

(a) Before you start your homework, write down your team. Who else did you work with on this homework? List names and email addresses. In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

(b) Please copy the following statement and sign next to it:

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up.

2 K-SVD

As you have seen in earlier homework problems, sparse representations are powerful. When we know the right dictionary of features within which our input is likely to have a sparse representations (possibly with some additional small non-sparse noise), we can use techniques like the LASSO or OMP (or even just rounding/thresholding) to recover the coefficients. As we have seen, this has a fundamentally better bias/variance tradeoff.

But what if we don't know the dictionary? How can the dictionary itself be found from the training data? This is the problem of dictionary learning. In this problem, we will introduce the K-SVD algorithm for unsupervised dictionary learning. (The naive perspective on directly supervised dictionary learning would reveal the dictionary to us, so that is not very interesting. However, it should be clear that the dictionary learning algorithm here can be adapted to deal with the case where we already know some dictionary elements to start with and want to learn more.) The K-SVD setting is where we would like to find a dictionary that enables good sparse fits to our data in the standard least-squares sense.

Mathematically, given an input matrix $X \in \mathbb{R}^{N\times d}$, where N is the number of data points and d is the dimension of each data point, we want to solve the following optimization problem:

$$\begin{aligned} \underset{D\in\mathbb{R}^{K\times d}, Z\in\mathbb{R}^{N\times K}}{\text{minimize}} \quad & \|X - ZD\|_F^2 \\ \text{subject to} \quad & \|z_i^\top\|_0 \le s \text{ for all } i. \end{aligned} \tag{1}$$

In the above problem, D is called a *dictionary*, which is a matrix in $\mathbb{R}^{K\times d}$. We denote the kth row of D by D_k^T, where $D_k \in \mathbb{R}^d$ is called an atom. Z is a matrix in $\mathbb{R}^{N\times K}$ whose ith row is z_i^T. The ith row of Z, z_i^T contains the coefficients for the linear combinations of atoms for the ith sample. $\|v\|_0$ is the so called "zero-norm" of v, which is defined to be the number of nonzero entries in the vector v. Throughout the homework, we write D, Z as

$$D = \begin{bmatrix} D_1^T \\ D_2^T \\ \vdots \\ D_K^T \end{bmatrix}$$

$$Z = \begin{bmatrix} z_1^T \\ z_2^T \\ \vdots \\ z_n^T \end{bmatrix} = \begin{bmatrix} Z_1, Z_2, \dots, Z_K \end{bmatrix}$$

with $D_k \in \mathbb{R}^d$, $z_i \in \mathbb{R}^K$ and $Z_k \in \mathbb{R}^N$.

A more human-language interpretation of the optimization problem is: Suppose the ith row of X represents a data sample x_i^T. Our goal is to approximate x_i by a sparse linear combination of atoms in D:

$$x_i \approx \sum_{k=1}^{K} z_{ik} D_k,$$

where the coefficients of linear combination are given in Z and should have at most s nonzero entries for each sample.

A greedy algorithm called K-SVD is proposed for the above optimization problem. The algorithm optimizes the objective over D and Z iteratively. The pseudo code of the algorithm is given in 1. It alternates between two stages: Sparse coding (Algorithm 2) and Update Codebook (Algorithm 4). The Sparse Coding procedure finds a sparse representation for each sample with Algorithm 3 based on a given codebook/dictionary. The procedure for updating the codebook/dictionary updates each row of the codebook with Algorithm 5 in a for loop, while modifying the effected representation coefficients correspondingly.

Algorithm 1 K-SVD

```
Require: Data matrix $X \in \mathbb{R}^{N \times d}$; Number of atoms $K$; Sparsity constraint $s$.
Ensure: A dictionary $D \in \mathbb{R}^{K \times d}$, and the coefficient matrix $Z \in \mathbb{R}^{N \times K}$.
  function K-SVD($X, K, s$)
      Initialize $D = D_0$ by randomly choosing $K$ samples without replacement from the dataset and aligning
  them as rows of $D_0$. Initialize $Z$ by $Z$ = Sparse-Coding($D$,$X$,$s$,0).
      while not converged do
          $Z$ = Sparse-Coding($D$,$X$,$s$,$Z$)
          Update-Codebook($D$,$Z$,$X$)
      end while
      Return $D, Z$.
  end function
```

Algorithm 2 Sparse-Coding

```
Require: Dictionary $D$; Data matrix $X \in \mathbb{R}^{N \times d}$; Sparsity constraint $s$.
Ensure: The coefficient matrix $Z \in \mathbb{R}^{N \times K}$.
  function SPARSE-CODING($D, X, s$)
      for n=1,...,N do
          $z'_n$ = Sparse-Coding-Single($x_n$,$s$)
          if $\|x_n^T - z_n^T D\|_F > \|x_n^T - z_n'^T D\|_F$ then
              Update $z_n = z'_n$
          end if
      end for
      Return $Z = (z_1, z_2, \dots, z_N)^T$.
  end function
```

Algorithm 3 Sparse-Coding-Single

```
Require: Dictionary $D \in \mathbb{R}^{K\times d}$; Data matrix $x \in \mathbb{R}^d$; Sparsity constraint $s$.
Ensure: A coefficient vector $z \in \mathbb{R}^K$.
  function SPARSE-CODING-SINGLE($D, x, s$)
      Initialize $z = (0, 0, \ldots, 0) \in \mathbb{R}^K$.
      Initialize residue $R_1 = x$. Initialize the basis set $B_1 = \{\}$.
      for $l = 1, \ldots, s$ do
          Let $D_k$ be the $k$th row of $D$.
          Find an atom $D_{k_l}^T \in \mathbb{R}^d$ from the dictionary $D$ that minimizes the residue error of linear regression
  incorporating the extra atom:
              $$\hat{\beta}, k_l = \underset{\beta \in \mathbb{R}^l, k}{\operatorname{argmin}} \|x - \sum_{D_j \in B_l \cup \{D_k\}} \beta_j D_j\|_2^2.$$
          Update the basis set $B_{l+1} = \{D_{k_l}\} \cup B_l$.
      end for
      For $D_j \in B_{s+1}$, set $z_j = \hat{\beta}_j$.
      Return $z$.
  end function
```

Algorithm 4 Update-Codebook

```
Require: Coefficient matrix $Z \in \mathbb{R}^{N\times K}$; Data matrix $X \in \mathbb{R}^{N\times d}$; the dictionary $D$.
Ensure: Dictionary $D \in \mathbb{R}^{K\times d}$
  function UPDATE-CODEBOOK($Z, X, D$)
      for $k = 1, \ldots, K$ do
          Update the $k$th row of $D$: Update-Codebook-Single($Z, X, D, k$)
      end for
  end function
```

Algorithm 5 Update-Codebook-single

```
Require: Coefficient matrix $Z \in \mathbb{R}^{N\times K}$; Data matrix $X \in \mathbb{R}^{N\times d}$; the dictionary $D$; $k$.
  function UPDATE-CODEBOOK-SINGLE($Z, X, D, k$)
      Find the indices of data samples that use this atom in their current sparse representation:
              $$\omega_k = \{n | 1 \le n \le N, z_{ik} \ne 0\}.$$
      Compute the error matrix $E_k \in \mathbb{R}^{n\times d}$ that represents how well things are currently represented without
  this atom.
              $$E_k = X - \sum_{j \ne k} Z_j D_j^T \in \mathbb{R}^{n\times d}$$
      Obtain $E_k^R \in \mathbb{R}^{|\omega_k|\times d}$ by choosing the rows of $E_k$ so that the index of that row is in $\omega_k$.
      Compute the SVD decomposition $E_k^R = U\Lambda V^T$, with $U \in \mathbb{R}^{|\omega_k|\times|\omega_k|}, \Lambda \in \mathbb{R}^{|\omega_k|\times d}$ and $V \in \mathbb{R}^{d\times d}$. We
  assume $\Lambda$ is a diagonal matrix with non-increasing diagonal elements.
      Update $D_k$ to be the first column of $V$.
      Update the nonzero elements of $Z_k$ to be the first column of $U$ multiplied by $\Lambda_{11}$, the largest singular
  value of $E_k^R$.
  end function
```

Our goal in this problem is to establish a relationship of K-SVD to the K-means algorithm you already know, to show that this K-SVD algorithm converges to a locally minimum, and to verify the usefulness of the K-SVD on a toy data set.

(a) (Relationship to K-means) In class, we've seen K-means. Recall that given a data matrix X, K-means divides the data into K partitions $\pi = \{\pi_1, \pi_2, \dots, \pi_K\}$. Each sample x_i is contained in one and only one partition π_j. K-means aims to solve the following optimization problem:

$$\underset{\pi, \mu \in \mathbb{R}^{K\times d}}{\text{minimize}} \quad \sum_{k=1}^{K} \sum_{i\in\pi_k} \|x_i - \mu_k\|_2^2 \tag{2}$$

μ is a $K \times d$ matrix whose kth row is μ_k^T. **Show that the above objective function is equivalent to**

$$\begin{aligned} &\underset{\mu\in\mathbb{R}^{K\times d}, Y\in\mathbb{R}^{N\times K}}{\text{minimize}} && \|X - Y\mu\|_{\mathrm{F}}^2 \\ &\text{subject to} && y_i \in \{0,1\}^K \text{ and } \|y_i^\top\|_0 = 1 \text{ for all } i. \end{aligned} \tag{3}$$

In the above $Y \in \mathbb{R}^{n\times K}$ with the ith row being $y_i^\top$.

Thus, K-means is equivalent to K-SVD with $s = 1$ with the additional constraint of forcing Z to only contain the values 0 or 1.

(b) **Show that in the greedy Algorithm 3 for finding a sparse solution, each step $l = 1, 2, \dots, s$ cannot increase the residue error of linear regression.**

(c) (Sparse coding decreases the objective) **Conclude from the above part that Algorithm 2 cannot increase the value of the objective function (1).** (In practice, people use straight matching pursuit — Algorithm 6 — to replace Algorithm 3, which is much more efficient, but gets comparable performance in most practical settings where there isn't a huge dynamic range of coefficients and the target sparsity s is low. But showing that matching pursuit usually works is omitted here.)

Algorithm 6 Matching Pursuit

```
Require: Dictionary $D \in \mathbb{R}^{K\times d}$; Data matrix $x \in \mathbb{R}^d$; Sparsity constraint $s$.
Ensure: A coefficient vector $z \in \mathbb{R}^K$.
  function SPARSE-CODING-SINGLE($D, x, s$)
      Initialize $z = (0, 0, \dots, 0) \in \mathbb{R}^K$
      Initialize residue $R_1 = x$
      for $l = 1, \dots, s$ do
          Let $D_k$ be the $k$th row of $D$.
          Find $k_l = \text{argmax}_k |D_k^T R_l|$
          $z_{k_l} = D_{k_l}^T R_l / \|D_{k_l}\|_2^2$
          Update residue $R_{l+1} = R_l - z_{k_l} D_{k_l}$
      end for
      Return $z$.
  end function
```

(d) **Show that the single-atom dictionary update given by Algorithm 5 does not increase the objective function while preserving the sparsity constraint.** (You may or may not find the following hint useful.)

Hint: Recall the Eckart-Young theorem.

(e) (Updating codebook decreases the objective) **Conclude that the iterations of the K-SVD algorithm put together cannot increase the objective function and hence the objective function must converge.**

(f) (Implementation with test data set) Implement K-SVD using numpy, scipy and sklearn. In particular, use the provided function *sparse_coding(D,X,s)* in the starter code for Sparse-Coding. Set the stopping criterion to be: stop if the difference between the old and new errors is smaller than a threshold (1×10^{-1}.)

Then generate the following data set for testing your code:

- Generate a zero-one matrix $Z \in \mathbb{R}^{N \times K}$. Each row of Z has s randomly-chosen entries being one, with the rest of the entries being zero.
- Generate dictionary $D \in \mathbb{R}^{d \times K}$ with entry-wise independent Gaussians $N(0,1)$.
- Generate $X = ZD$.

For testing, we set $N = 1000, d = 10, K = 10$ and $s = 2$. Use the true K and s for testing your K-SVD code.

Remember to run the K-SVD algorithm with differently randomized initial conditions since only a local minimum is found and you want to find a good global minimum.

Report the reconstruction MSE, $\frac{1}{N}\|\hat{X} - X\|_F^2$, where $\hat{X} = \hat{Z}\hat{D}$, with $\hat{Z}$ and $\hat{D}$ found by K-SVD.

(g) (Frequency data) We use the following procedures to generate a toy data set:

- Generate a dictionary $D^* \in \mathbb{R}^{K \times d}$ by letting

$$D_{kj} = \sin(\frac{2\pi f(j)}{K}k)$$

for $j = 1, 2, \ldots, d$ and $k = 1, 2, \ldots, K$. Thus each atom in the dictionary is

$$D_i = \begin{bmatrix} \sin(\frac{2\pi f(i)}{K}) \\ \sin(\frac{2\pi f(i)}{K} \cdot 2) \\ \vdots \\ \sin(\frac{2\pi f(i)}{K} \cdot d) \end{bmatrix},$$

followed by normalization of each atom. Here we choose $f(k) = k - 1$ for $k = 1, 2, \ldots, K$.

- Randomly generate the coefficients $Z^* \in \mathbb{R}^{n \times K}$ by generating each row $z_i \in \mathbb{R}^K$ independently with the following scheme: Randomly choose s entries without replacement from the K entries of z_i. Set them to be 1. Set the rest of the $K - s$ entries to be zero.

- Construct the data matrix X by $X = Z^*D^* + c\mathscr{E}$, where each entry of $\mathscr{E} \in \mathbb{R}^{n\times d}$ is generated independently from $N(0,1)$.

In the first experiment, we generate a data set with $N = 200, s = 3, K = 5, d = 20, c = 0$. **Apply K-SVD($X, K, s'$) with $s' = 1,2,\ldots,5$ to find the estimated dictionary D and the estimated coefficient matrix Z, and plot the reconstruction MSE $\frac{1}{N}\|\hat{X} - X\|_F^2$ against s', with $\hat{X} = ZD$.** (You don't need to include values of Z, D in submission.)

Don't forget to try different random initializations.

(h) In the second experiment, we generate a data set with $N = 200; s = 2; K = 20; d = 5, c = 0$. **Apply K-SVD($X, K, s'$) with $s' = 1,2,\ldots,5$ and plot the reconstruction MSE $\frac{1}{N}\|\hat{X} - X\|_F^2$ against s'.** (You don't need to include values of Z, D in submission.)

(i) In the third experiment, we generate a data set with $N = 200; s = 3, K = 20, d = 5, c = 0.001 \times 10^{-1}$. **Apply K-SVD($X, K, s'$) with $s' = 1,2,\ldots,5$ to find the estimated dictionary D and the estimated coefficient matrix Z, and plot the reconstruction MSE $\frac{1}{N}\|\hat{X} - X\|_F^2$ against s', with $\hat{X} = ZD$.** (You don't need to include values of Z, D in submission.)

(j) In the fourth experiment, we generate a data set with $N = 200; s = 2; K = 200; d = 100, c = 0.001$. **Apply K-SVD($X, K, s$) and get the dictionary D. Plot any three rows (atoms) of the dictionary D on the same coordinate system, with x axis ranging among $1,2,\ldots,d = 100$ being the index of entry and y axis being the value of the corresponding entry.** If you are on the right track, you will see approximately sine curves.

3 GANs (Optional)

In this problem, we will explore Generative Adversarial Networks (GANs) for learning Generative Models. Generative models try to learn the probability distribution over the state x (i.e. $p(x)$). The advantage of this is that now new data points can be sampled from the generative model. For instance, if a generative model was trained on the distribution of Picasso paintings and then sampled from, it should produce a new unseen Picasso-like painting.

Not surprisingly, generative models have been notoriously difficult to train because it is not entirely clear what loss they should be optimizing. For instance, is minimizing the squared Euclidean distance over pixel values appropriate for paintings? A recent result by Goodfellow et al. proposed an alternative to specifying a loss function known as a GAN. The idea behind a GAN comes from the following intuition.

Pretend there exists an art forger, who is attempting to fake a Picasso. The art forger successfully forges the painting, if they trick the Art Gallery into believing it is real. The Art Gallery is actively trying to detect if the painting is real or fake and thus they are competing against each other. The hope with GANs is that by creating this game, the art forger will learn to make better paintings to trick the Art Gallery. Thus, we do not need to ever explicitly specify a loss function of what a Picasso painting should look like, because this loss function is going to implicitly be learned by the Art Gallery as it compares true Picassos to known fakes.

The problem can be run on a Mac-Book Pro in 10 minutes and should not require EC2. Please do not import any new libraries.

(a) We begin this problem by examining the MNIST dataset. Recall from previous homework that MNIST is a large dataset of handwritten digits ranging from $0-9$ with each image being size 28 by 28. Consider each image as a data point x, we want to learn the distribution over possible images from the dataset $p(x)$. One classical way[1] to do this is known as Kernel Density Estimation (KDE). KDE can be written as follows

$$\hat{p}(x) = \frac{1}{Z}\sum_{i=0}^{N-1} k(x_i, x),$$

where Z is chosen to normalize this to be a proper probability density function. Implicitly, the estimated density is determined by measuring how close the point is x to the training data in some kernel space. In this problem, we will consider the Gaussian Kernel (Radial Basis Function) for the Kernel or RBF. This can be written as

$$k(x, x_i) = \exp\left(\frac{||x-x_i||_2^2}{\sigma}\right),$$

where σ is the bandwidth of the kernel. The bandwidth measures how much weighting each training point has in the KDE. What this essentially is saying is that the learned distribution is noisy versions of the training data, where σ is controlling that noise. **Implement the class kde.py and run train_on_mnist.py to sample MNIST digits from the learned distribution. Report the resulting image**. Note that in the code we will project the data down to a lower dimension with PCA to reduce computation.

(b) The above KDE approach typically produce images that are quite blurry, this is a result of the distance measure we are using to measure similarity is interpolating between data points. In the RBF kernel, the distance measure is the squared L2 distance. While this is a common distance measure, it may not be appropriate to measure how similar images are when applied on the pixel level.

Instead of finding a more appropriate distance measure, we can train a Generator to simply try and trick a Discriminator. Please implement the discriminator, D, and the generator, G in gan.py. They should both have the same high level architecture:

  (a) Layer 1: Fully Connected Layer
  (b) Non-Linear Response: Rectified Linear Units
  (c) Layer 2: A Fully Connected Layer
  (d) Non-Linear Response: Sigmoidal Layer

[1]There are plenty of others. For example, we can do something like QDA married to K-means style ideas where cluster centers are learned together with local covariances in an iterative fashion.

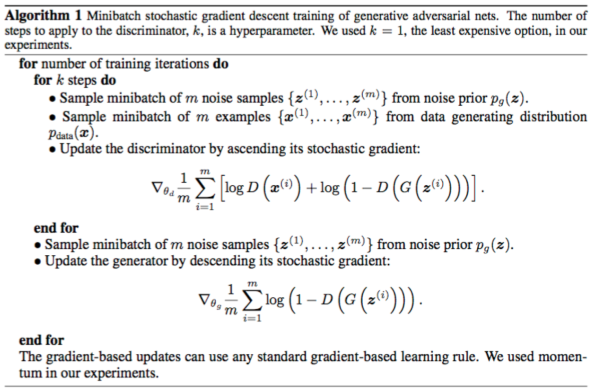
The dimensions of each layer are provided in the skeleton code. The discriminator will map an image to a probability score of whether it is real or not. The generator will map random noise in a latent space to a generated image.

(c) We are now ready to train our generator and discriminator. The objective function for the adversarial game can be written as follows:

$$\min_G \max_D V(D,G) = E_{x\sim p(x)}[\log(D(x))] + E_{z\sim p_z}[1-\log(D(G(z))]$$

The objective is stating that we want our discriminator to correctly classifier the real data sampled from $p_{\text{data}}(x)$ from the fake data sampled from $p_{\text{Z}}(z)$. However, we want the Generator to try and fool the discriminator. Thus forming an adversarial optimization.

In the original GAN paper, they proposed the following optimization to solve for this objective.



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```
for number of training iterations do
    for $k$ steps do
        • Sample minibatch of $m$ noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
        • Sample minibatch of $m$ examples $\{x^{(1)}, \ldots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
        • Update the discriminator by ascending its stochastic gradient:
              $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right].$
    end for
    • Sample minibatch of $m$ noise samples $\{z^{(1)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$.
    • Update the generator by descending its stochastic gradient:
          $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right).$
end for
```

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Implement the rest of gan.py and report the resulting image after training the GAN.

(d) GANs yield a visible improvement in terms of matching the MNIST distribution. However, it is important to understand that the game theoretic formulation is not guaranteed to be stable and can lead to undesirable results. For example, one potential outcome in the GAN formulation is a phenomena known as mode collapse, in which the generator decides to only play a single value from the distribution. The discriminator is unable to distinguish this from the real data; however the true distribution is not captured by the generator.

We can see this by training our GAN on a 1D Gaussian distribution. **Run train_on_gaussian.py and compare KDE to GANs for predicting a 1D Gaussian.** Report your findings.