ECE/CS 559: Neural Networks **Fall 2024**

Homework 7

Due: **Tuesday** November 26, 2024 (by 9pm, on Gradescope)

Note: *You may discuss these problems in groups. However, you must write up your own solutions and mention the names of the people in your group. Also, please do mention any solution manuals, online material, or other sources you used in a major way.*

Submit your code as a **two** `.py` **files**, one for each question, in Gradescope. Make sure that it **runs properly** and generates all the plots that you report on in the main submission.

1. **[K-Means using a Neural Network]**

   Load the MNIST dataset in PyTorch. Your goal is to cluster the images into $k = 10$ clusters, without relying on the label. Reuse your prior code, especially the loading, training, and testing aspects based on the PyTorch tutorial. Make the following significant changes.

   - Build a neural network that defines three elements in its constructor: a parameter `self.centers` that is a 10×784 tensor defined using `nn.Parameter` and referring to the centers matrix C, a flatten layer, and a softmax layer.
   - In the forward pass, do this:
     - Use the flatten layer on the input `x` and call the resulting variable `x0`, to keep access to it, and keep using `x` for subsequent stages.
     - Use the following single tensor equation to manually implement the 10 neurons of the first layer that we saw in class, for $i = 0, 1, \cdots, 9$, $xC_i^T - \frac{1}{2}\|C_i\|^2$:
       ```
       x = torch.matmul(x0, self.centers.t())
             -0.5*torch.sum(self.centers**2,1).flatten()
       ```
     - Multiply the result by 20 then pass it through a softmax layer.
     - Reconstruct the input using a center, by multiplying the softmax output with the tensor C, from the right. [1-D tensors in PyTorch are row vectors.]
     - Return the error `x-x0`, where `x` is the reconstruction/center.
   - Use the MSE loss. To do this, when you call the loss, use one argument as the model's prediction (which is simply the reconstruction error) and let the other argument be a 0 tensor, the same size as the prediction batch (so that it computes compute $\|(x - x_0) - 0\|^2 = \|x - x_0\|^2$.)

     [The reason we're doing this is in order not to bother with manipulating batches ourselves. When you want the center, you can still get it, by adding the image to the reconstruction error: $(x - x_0) + x_0 = x$.]

- Modify the test loop to only evaluate and display loss, and not accuracy.

(a) Say we're using a batch size of $b = 64$. Recall that PyTorch by default *averages* the losses over each batch. Assume that, in each batch, roughly $\frac{1}{10}^{\text{th}}$ of the images are in each cluster, and assume that the softmax behaves likes an argmax. Explain why the update at iteration t looks like:

$$C_i(t) = \left(1 - \frac{\eta}{5}\right) C_i(t-1) + \frac{\eta}{5} \underbrace{\frac{1}{|\mathcal{V}_i(t)|} \sum_{x \in \mathcal{V}_i(t)} x}_{m_i(t)}$$

where $\mathcal{V}_i(t)$ are all images in the Voronoi cell of C_i, in this batch. $m_i(t)$ are the average centers in each cell *only in this batch*. Use $\eta = 4.5$. See the last page of the homework for an explanation of why this is a good idea for approximating Lloyd's algorithm, with the batch size we chose.

(b) Cheat, by initializing the cluster centers (`model.centers`) to individual images from the corresponding digit, i.e., set C_i to an image of digit i by saying:
`model.centers[i,:] =` a flattened version of the image.
Then, train your k-means neural network for 10 epochs and produce two outputs (pasted in your report):
   i. All the cluster centers at the end of the training.
   ii. A table of size 10×10 that contains for each row i and column j, the number of training images of digit i that were clustered in center j. (The sum of all entries should be 60,000.)

Discuss these results by commenting on what places k-means clustering is having trouble and what you think the reasons are behind it. [Hint: Use a data loader with batch size 1 to iterate over the data points individually.]

(c) Try to initialize the centers differently (e.g., sampling each coordinate uniformly randomly from $[0, 1]$), and try training the network again. Report on your choice of initialization and what kind of results you obtain (e.g., do you see something that you could reasonably call "mode collapse"?)

2. **[Autoencoder]**

Continue to use the same base code. Replace your neural network with an autoencoder. Let your encoder be, in one sequence, the same as the CNN from HW6 Q3 ending in a 10-dimensional representation `z`. For the decoder, use the following (roughly inverted) architecture in one sequence:

- A fully-connected stage with ReLU activations, going from 10 to 360, then from 360 to 720.

- Unflatten dimension 1 (we don't touch dimension 0, as it is the batch dimension) to `torch.size([20,6,6])`. This gives 20 channels of 6×6 images.
- Perform bicubic upsampling by a factor of 2.
- Use a (transpose) convolutional layer with 20 output channels, 4×4 filters, stride 2, and 1 output padding, followed by ReLU.
- Use a final (transpose) convolutional layer with 1 output channel, 4×4 filters, **stride 1**, and no output padding, followed by sigmoid.

Add batch normalization and dropout (with probability 0.1) after every ReLU activation, in both the encoder and decoder. Your final output should be 28×28.

In the forward pass, do this:

- Get `z` by passing the input image `x` through the encoder.
- Add standard Gaussian noise to `z` to make `z2`. To make generation easy, give the forward function an additional binary argument called `enc_mode`, set by default to 1, and use:
  `z2 = enc_mode*z + (2-enc_mode)*torch.randn(z.shape)`
  If we're in encoding mode (during training), this will act as additive noise. During generation mode (`enc_mode=0`), this will replace `z` itself with Gaussian noise. (Slightly different than in class: we're adding all noise before the decoder.)
- Get the output image `f` by passing `z2` through the encoder.
- Flatten `f` and return the concatenation of `z` (the embedding) and `f-x` (the reconstruction error), along dimension 1 (we don't touch dimension 0, as it is the batch dimension).

The reason we do this is because we're going to simulate the ELBO loss, and we saw that for Gaussian autoencoders it is simply the mean square of the embedding + reconstruction error. Therefore, you can simply use `MSELoss`. Set the batch size to 64 and the learning rate to 0.1.

(a) Draw a rough tensor block sketch of the encoder architecture and, to its right, the decoder architecture. Highlight which stage in the first is "inverted" by which stage in the second, by connecting them with brackets underneath.

(b) Train your autoencoder for 10 epochs. Visualize its performance by generating 20 random digits, by calling the model with the additional argument `enc_mode=0`. Note that you should eliminate the first 10 entries, which correspond to the embedding, by using a slicing, e.g., `[0,10:]`.

(c) Suggest modifications to the architecture, regularization, or training to obtain better results. Report your changes and visualize the results in the same way.

In Q1, using the minibatch approach has some advantages, including faster processing and avoiding local minima. The minibatch updates average centers *only within the minbibatch*,

$$C_i(t) = \left(1 - \frac{\eta}{5}\right) C_i(t-1) + \frac{\eta}{5} \underbrace{\frac{1}{|\mathcal{V}_i(t)|} \sum_{x \in \mathcal{V}_i(t)} x}_{m_i(t)}$$

Lloyd's algorithm, in contrast, averages centers *across the whole data set*. This means we have some bias (offset) and variance (noisiness) due to using minibatches. For a fixed batch size, the choice of η creates a tradeoff between bias and variance.

To understand this, let's focus on when $\eta \in [0, 5]$ (bigger η will make learning unstable by moving *away* from prior centers.) We first explain the intuition of why bias and variance cannot be improved simultaneously, i.e., there's a tradeoff between them. Let's first think about the extremes. If $\eta = 0$, then we do not make updates. The variance is thus 0, but the bias is very large: we're very far from what we should be doing, which is averaging centers. If $\eta = 5$, we cancel everything from before and only use the last minibatch. This is the least amount of data we could use, and thus the variance is large, however m is doing what we should be doing, it is an unbiased estimate of what Lloyd's would have calculated.

How can we estimate bias? Because successive updates shift the centers, they cause m to drift away from Lloyd's average. By how much? The average pixel value is about $a = 0.23$. We'll assume that the latest average is the "freshest" and that the further back we look in batches, each coordinate of m could drift by as much as a times how far back we look, in average. Regarding noisiness, we can assume the noise level of each data point is constant. Recall that b is the minibatch size. The minibatch averaging reduces the variance by $\frac{1}{b}$, then averaging across minibatches will reduce the variance based on how the averaging is being done.

To simplify the notation, let $\alpha = \eta/5$, and let's consider a single coordinate. Also, let N be the training size and let T be the number of minibatches, so $T = N/b$.We have

$$\begin{aligned} C(T) &= (1-\alpha)C(T-1) + \alpha m(T) \\ &= (1-\alpha)^2 C(T-2) + (1-\alpha)\alpha m(T-1) + \alpha m(T) \\ &= \ldots \\ &= (1-\alpha)^T C_0 + \sum_{t=1}^{T} \alpha(1-\alpha)^{T-t} m(t) \\ &\approx \sum_{t=1}^{T} \alpha(1-\alpha)^{T-t} m(t) \end{aligned}$$

where the last approximation is truf if $\alpha < 1$ and T is large. In our case with $b = 64$ and $N = 60,000$, T is indeed quite large.

Let M be the ideal average of all centers. We can model our assumptions of bias and variance roughly as:

$$m_t = M + a(T-t) + aZ_t/\sqrt{b}, \qquad Z_t \sim \mathcal{N}(0,1) \text{ (i.i.d.)}$$

We can now calculate our bias and variance:

$$\begin{aligned}
\text{Bias} &= M - \mathbf{E}\left[C(T)\right] = M - \sum_{t=1}^{T} \alpha(1-\alpha)^{T-t} M + a(T-t) \\
&\approx \frac{1-\alpha}{\alpha} a
\end{aligned}$$

where the last expression takes $T \to \infty$ and uses the formula for the mean of a geometric random variable starting at 0.

$$\begin{aligned}
\text{Variance} &= \sum_{t=1}^{T} \left[\alpha(1-\alpha)^{T-t}\right]^2 \frac{a^2}{b} \\
&= \frac{1}{b} \frac{\alpha^2}{1-(1-\alpha)^2} \sum_{s=0}^{T-1} (1-(1-\alpha)^2)((1-\alpha)^2)^s \\
&\approx \frac{a^2}{b} \frac{\alpha^2}{1-(1-\alpha)^2}
\end{aligned}$$

where the last expression takes $T \to \infty$ and uses the fact that a geometric distribution sums to 1.

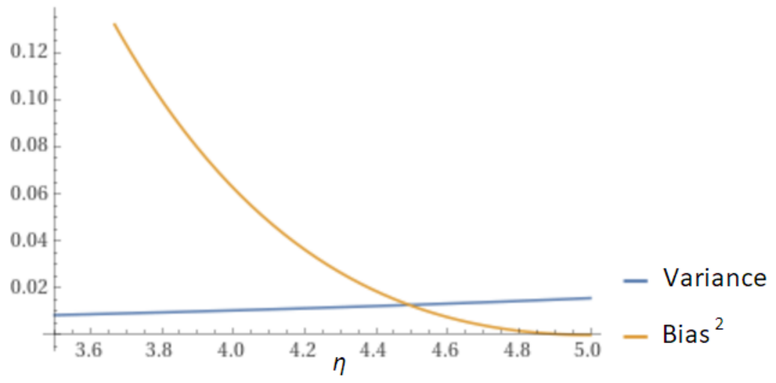
We now have estimates of what the repercussion is on bias and variance for a specific choice of η (or α), for a given batch size b and assuming average pixel value of a. As we observed with the extreme cases, we can see that with increasing α (or η), the bias goes down, but the variance goes up.

Bias and Variance combine as $Bias^2 + Variance$ to give the overall error, so to balance them out, we need to find a point where:

$$\left(\frac{1-\alpha}{\alpha} a\right)^2 \approx \frac{a^2}{b} \frac{\alpha^2}{1-(1-\alpha)^2}$$

Notice that because a affects both sides equally, it comes out in the wash. In terms of η, we need to find where the two curves, which we plot below:

$$\left(\frac{1-\frac{\eta}{5}}{\frac{\eta}{5}}\right)^2 \quad \text{and} \quad \frac{1}{64} \frac{(\frac{\eta}{5})^2}{1-(1-\frac{\eta}{5})^2}$$



From this, we see that the balance is achieved around $\eta = 4.5$, which is the choice suggested by the question. This also gives us an idea of how we should change η as we change b. For example, when you set $b = N = 60,000$, we should choose $\eta = 5$, reverting back to Lloyd's algorithm exactly. And if we set b samller, we should make η smaller, but even at $b = 1$ (pure SGD), this analysis suggests η shouldn't go below 3. Something worth experimenting with!