CS294-158 Deep Unsupervised Learning
UC Berkeley, Spring 2019
HW1: Autoregressive Models
Due: February 11, 11:59pm

1 Warmup

First, run the following code. It will generate a dataset of samples $x \in \{1, \ldots, 100\}$. Take the first 80% of the samples as a training set and the remaining 20% as a test set.

```
import numpy as np
def sample_data():
    count = 10000
    rand = np.random.RandomState(0)
    a = 0.3 + 0.1 * rand.randn(count)
    b = 0.8 + 0.05 * rand.randn(count)
    mask = rand.rand(count) < 0.5
    samples = np.clip(a * mask + b * (1 - mask), 0.0, 1.0)
    return np.digitize(samples, np.linspace(0.0, 1.0, 100))
```

Let $\theta = (\theta_1, \ldots, \theta_{100}) \in \mathbb{R}^{100}$, and define the model

$$p_\theta(x) = \frac{e^{\theta_x}}{\sum_{x'} e^{\theta_{x'}}} \tag{1}$$

Fit p_θ with maximum likelihood via stochastic gradient descent on the training set, using θ initialized to zero. Use your favorite version of stochastic gradient descent, and optimize your hyperparameters on a validation set of your choice. **Provide these deliverables:**

1. Over the course of training, record the average negative log likelihood of the training data (per minibatch) and validation data (for your entire validation set). Plot both on the same graph – the x-axis should be training steps, and the y-axis should be negative log likelihood; feel free to compute and report the validation performance less frequently. Report the test set performance of your final model. Be sure to report all negative log likelihoods in bits.

2. Plot the model probabilities in a bar graph, with $\{1, \ldots, 100\}$ on the x-axis and a real number in $[0, 1]$ on the y-axis. Next, draw 1000 samples from your model, and plot their empirical frequencies on a new bar graph with the same axes. How do both plots compare visually to the data distribution?

2 Two-dimensional data

In this problem, you will work with bivariate data of the form $\mathbf{x} = (x_1, x_2)$, where $x_1, x_2 \in \{0, 1, \ldots, 199\}$. In the file called `distribution.npy`, you are provided with a 2-dimensional array of floating point numbers representing the joint distribution of $\mathbf{x}$: element (i, j) of this array is the joint probability $p_{\text{data}}(x_1 = i, x_2 = j)$.

Sample a dataset of 100,000 points from this distribution. Treat the first 80% as a training set and the remaining 20% as a test set. Implement and train both of these models for this data:

1. $p_\theta(\mathbf{x}) = p_\theta(x_1)p_\theta(x_2|x_1)$, where $p_\theta(x_1)$ is a distribution represented in the same way as in part 1, and $p_\theta(x_2|x_1)$ is a multilayer perceptron (MLP) that takes x_1 as input and produces a distribution over x_2. (You have some freedom in designing the architecture of this MLP. For example, it can read the x_1 input either as a real number or as a one-hot vector. Experiment with such designs and pick what works best on validation data.)

2. $p_\theta(\mathbf{x})$ represented as a Masked Autoencoder for Distribution Estimation (MADE).[1]

Fit both models with maximum likelihood, paying attention to performance on a validation set. **Provide these deliverables for both models:**

1. Over the course of training, record the average negative log likelihood of the training data (per minibatch) and the validation data (for the entire validation set). Plot both on the same graph – the x-axis should be training steps, and the y-axis should be negative log likelihood – and report the test set performance of your final model. Report all negative log likelihoods as bits per dimension (i.e. bits divided by 2).

2. Draw samples and plot them in a 2D histogram with 200 bins on each side. (Consider using the `hist2d` function in `matplotlib`.)

3 High-dimensional data

Now, you will train more powerful models on high-dimensional data: colored MNIST digits. The dataset you are provided has 60,000 training images and 10,000 test images. Each image has height $H = 28$ and width $W = 28$, with $C = 3$ channels (red, green, and blue). Let $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_{HW})$ represent one image, where each $\mathbf{x}_i = (x_i^1, \ldots, x_i^C)$ is a vector of channel values for one pixel. Each x_i^c will take on a value in $\{0, 1, 2, 3\}$.

First, design an autoregressive model of the form

$$p_\theta(\mathbf{x}) = \prod_{i=1}^{HW} p_\theta(\mathbf{x}_i|\mathbf{x}_{1:i-1}) = \prod_{i=1}^{HW}\prod_{c=1}^{C} p_\theta(x_i^c|\mathbf{x}_{1:i-1}) \tag{2}$$

i.e. a model which is autoregressive over space, but factorized over channels. Use the masked PixelCNN architecture[2] to implement spatial dependencies. We recommend you to follow the architecture from Table 1 in the paper linked. Start with a 7x7 masked convolution of type A,

[1] MADE: `https://arxiv.org/abs/1502.03509`

[2] PixelCNN: `https://arxiv.org/abs/1601.06759`

followed by 12 layers of residual blocks containing masked convolutions of 1x1, 3x3 and 1x1 of type B (Figure 5 in the paper). The logits for the softmax layer are then obtained after two 1x1 masked convolution layers of type B.

Next, introduce the capacity to model dependencies among channels:

$$p_\theta(\mathbf{x}) = \prod_{i=1}^{HW} p_\theta(\mathbf{x}_i|\mathbf{x}_{1:i-1}) = \prod_{i=1}^{HW}\prod_{c=1}^{C} p_\theta(x_i^c|x_i^{1:c-1}, \mathbf{x}_{1:i-1}) \tag{3}$$

To do so, use a MADE over the channels, conditioned on previous pixels with the PixelCNN structure. In other words, the joint probability for one pixel $\mathbf{x}_i$, conditioned on previous pixels $\mathbf{x}_{1:i-1}$, should have the form:

$$p_\theta(x_i^1, \ldots, x_i^C \,|\, \mathbf{x}_{1:i-1}) = g_\theta\left(x_i^1, \ldots, x_i^C \,|\, \phi_\theta(\mathbf{x}_{1:i-1})\right) \tag{4}$$

where $\phi_\theta(\mathbf{x}_{1:i-1})$ is a feature vector summarizing $\mathbf{x}_{1:i-1}$ using the PixelCNN structure, and

$$g_\theta(x_i^1, \ldots, x_i^C \,|\, \phi) = \prod_{c=1}^{C} g_\theta(x_i^c \,|\, x_i^{1:c-1}, \phi) \tag{5}$$

is a MADE that takes ϕ as an auxiliary input.

Here are some tips that you may find useful for designing and training these models:

- You will need only a 4-way softmax for every prediction, as opposed to a 256-way softmax in the PixelCNN paper. This is because the dataset is quantized to two bits per color channel.
- You can set number of filters for each convolution to 128. You can use the ReLU nonlinearity throughout.
- Consider using layer normalization[3] to improve performance.
- With a learning rate of 10^{-3} and a batch size of 128, you should be able to achieve a log-likelihood of 0.11 bits/dim in approximately 50 epochs. This should take about 30 minutes.

Provide these deliverables for both models:

1. Plot training and validation losses over time and report the test set performance of your final model, just as in previous parts of this assignment. Report all negative log likelihoods in bits per dimension.
2. Generate and display 100 samples from your model. You may want to scale the values $\{0, 1, 2, 3\}$ to $\{0, \ldots, 255\}$ for display purposes.
3. Visualize the receptive field of the conditional pixel distribution at $(14, 14, 0)$. To do so, compute the gradient of the log probability of the model with respect to the input image at randomly initialized parameters. Turn the resulting gradient into a visualization by computing its elementwise absolute value and taking the maximum over channels – this should transform the 28x28x3 gradient into a 28x28 image.

[3]Layer Normalization: `https://arxiv.org/abs/1607.06450`

4 Bonus Questions (Optional)

- Try extensions to the PixelCNN-MADE architecture in part 3. For example, try using a mixture of logistics distribution for each conditional instead of a categorical distribution (softmax), or try implementing the Gated PixelCNN that avoids the blind spot problem in the masking-based PixelCNN. Report any performance gains you see over the version you implemented in part 3.
- Choose your own dataset to train with the PixelCNN-MADE. Report samples and document any extentions or tricks you needed to make your model work.
- Turn PixelCNN-MADE into an actual compression algorithm by implementing arithmetic coding. Report compression time, decompression time, and achieved bit rate.
- Implement PixelCNN-MADE with conditioning on auxiliary variables.[4] Report samples with grayscale conditioning.
- Train a network similar to the PixelCNN-MADE using quantile regression.[5] Compare sample quality to a similar model (in terms of architecture and parameter count) trained by maximum likelihood. Comment on similarities and differences between the quantile regression objective and the maximum likelihood objective.

[4] PixelCNN Models with Auxiliary Variables for Natural Image Modeling: `https://arxiv.org/abs/1612.08185`
[5] Autoregressive Quantile Networks for Generative Modeling: `https://arxiv.org/abs/1806.05575`