EE 8520 - Homework 2

Generating Handwritten Digits

Due: October 21, 2025 **Template:** Choose one of Option-1 or Option-2

! Please follow the "rules for every HW" listed in the syllabus

Programming Exercise: In this assignment, you will implement several different generative models (GANs, ARs, and VAEs) and apply them to the MNIST handwritten digit generation task. You are provided with skeleton codes for each to help you get started.

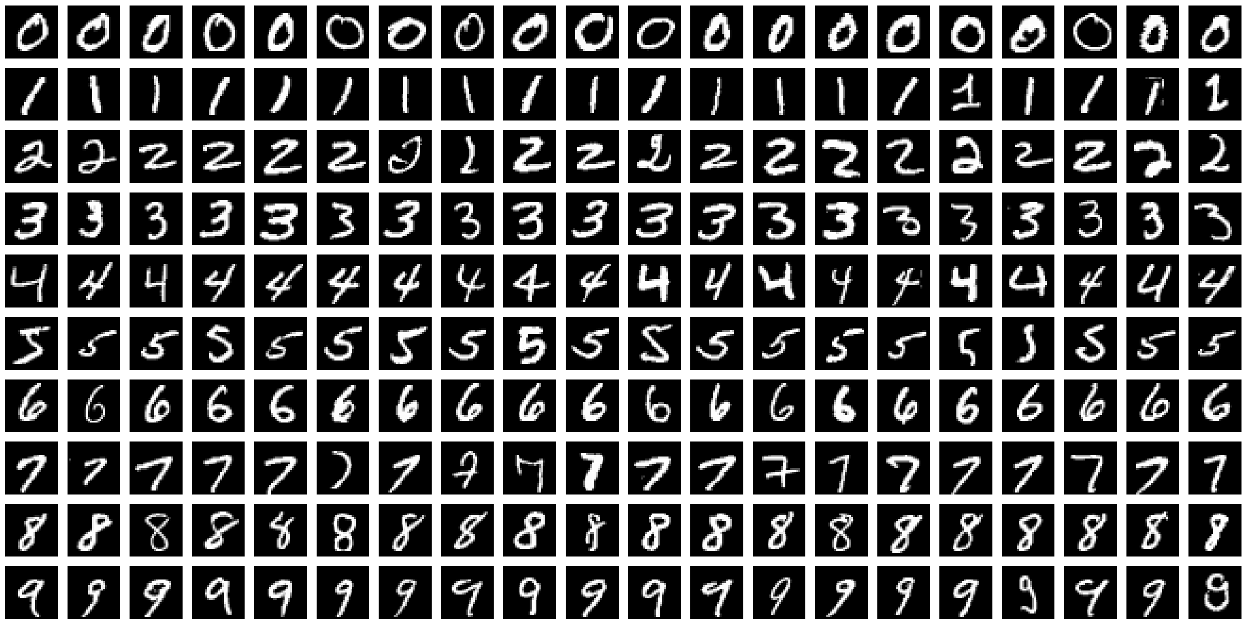
Figure 1: Sample images from MNIST test dataset.

Step 1 (25 points): Generative Adversarial Networks (GANs)

Your first task is to implement an *unconditional* GAN (Goodfellow et al., 2014) for MNIST. The goal is to train the model to convert random noise vectors into images of handwritten digits. You are given skeleton codes `hw2_step1_main.py` and `models/GAN.py`. Make sure to check lines with the `#TODO` flag in order to complete the code. For this step, your goals are:

(a) [**3 pts**] In your report, explain how GAN training works, describe the roles of the generator and discriminator, and discuss whether the model is explicitly modeling a probability distribution.

(b) [**8 pts**] Design and implement the architectures for both the generator and the discriminator, and their training functions in the given `models/GAN.py` file. Each should be built entirely with fully connected layers and use Leaky ReLU throughout. Batch normalization is not required.

- **Generator:** Construct with three linear layers containing 64, 128, and 256 hidden units, respectively. No dropout is required here.
- **Discriminator:** Construct with three linear layers containing 256, 128, and 64 hidden units, respectively, and apply dropout as a form of regularization.

After training, visualize the learning process by plotting the losses for both networks and showing sample images produced by the generator. Include the loss graphs and generated samples (across multiple epochs) in your report and comment on them. What do the loss curves tell you? Is it what you expected?

(c) [**10 pts**] GAN training is a two-player game between the Generator and Discriminator, so losses may oscillate rather than converge, depending on your design. If one network dominates (e.g., the Generator loss grows very large or the Discriminator loss goes to zero), try adjusting model capacity (channel sizes), dropout, learning rates, number of epochs, or the update balance between the two networks. Therefore, **experiment with different hyperparameters for best performance. Justify your reasoning for each**: what did you try, what worked better? And what are your comments?

Some of the expected ablations:

- Try doubling the size of the channels in generator.
- Increase the capacity of the discriminator or generator (choose which depending on your need) by increasing the number of layers.
- Try adjusting the amount of dropout in the discriminator, or experiment with using different learning rates for the generator and discriminator.
- Update the discriminator slightly more times than the generator (`d_freq`).

Comment on these ablations and share your observations with relevant figures.

(d) [**4 pts**] Suppose we have a GAN with generator G and discriminator D. Denote the true data distribution by $p_{\text{data}}(x)$ and the model distribution produced by G as $p_g(x)$. The interaction between D and G can be described as a minimax game with the following objective:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}\big[\log D(x)\big] + \mathbb{E}_{z\sim p_z}\big[\log\big(1 - D(G(z))\big)\big]. \tag{1}$$

When G is fixed, the discriminator that maximizes the objective takes the form

$$D_G^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}.$$

(e.1) In your report, derive this expression for the optimal discriminator. *Hint: Begin with GAN objective (Eq. 1) and apply calculus to show why this is the maximizer of D.*

(e.2) What is $D_G^*(x)$ when $p_g(x) = p_{\text{data}}(x)$? What does it tell you? Discuss in your report.

Step 2 (25 points): Autoregressive Models (ARs)

Your second task is to implement PixelCNN (Van Den Oord et al., 2016) for MNIST. The goal is to train the model to capture the distribution of binarized MNIST images by predicting each pixel conditioned on the previously generated pixels. You are given skeleton codes `hw2_step2_main.py` and `models/AR.py`. Make sure to check lines with the `#TODO` flag. For this step, your goals are:

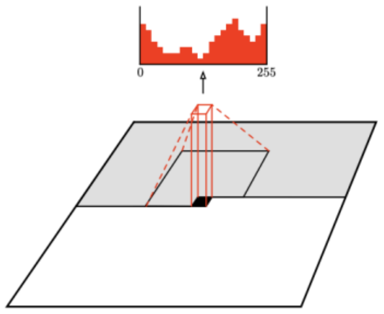

Figure 2: Illustration of PixelCNN.

(a) **[3 pts]** Prove that the Negative Log-Likelihood (NLL) is equivalent to the Cross-Entropy Loss *for binary sequence* autoregressive models. Consider a binary sequence $x = (x_1, x_2, \ldots, x_T)$, where each $x_i \in \{0, 1\}$. An autoregressive model predicts each element x_i given all the previous ones $(x_1, \ldots, x_{i-1})$, *i.e.*,

$$p(x_i \mid x_1, \ldots, x_{i-1}). \quad \text{where} \quad \hat{y}_i = p(x_i = 1 \mid x_1, \ldots, x_{i-1}),$$

so that $1 - \hat{y}_i = p(x_i = 0 \mid x_1, \ldots, x_{i-1})$. The **Negative Log-Likelihood (NLL)** and **Cross-Entropy Loss** for the sequence x is defined as:

$$\mathrm{NLL}(x) = -\sum_{i=1}^{T} \log p(x_i \mid x_1, \ldots, x_{i-1}), \quad \mathrm{CE}(x) = -\sum_{i=1}^{T} \Big(x_i \log \hat{y}_i + (1 - x_i) \log(1 - \hat{y}_i) \Big).$$

Starting from the NLL definition and using the Bernoulli likelihood $\Big(p(x_i \mid x_1, \ldots, x_{i-1}) = \hat{y}_i^{x_i} (1 - \hat{y}_i)^{1 - x_i}\Big)$, show step by step that $\mathrm{NLL}(x) = \mathrm{CE}(x)$.

(b) **[6 pts]** Implement a custom 2D masked convolutional layer (`MaskedConv2d`) that enforces the autoregressive property by applying a binary mask to the convolution weights. This mask restricts each pixel to depend only on earlier pixels in the raster-scan order. Your implementation should support two mask types:

- **Type A:** excludes the center pixel itself (to be used in the first layer).
- **Type B:** includes the center pixel while still blocking all future pixels (for subsequent layers).

(c) **[6 pts]** Build an *unconditional* PixelCNN (`PixelCNN`) using your `MaskedConv2d` layer. The network should stack several masked convolutional blocks with nonlinear activations (*e.g.*, ReLU) and, if needed, normalization layers. Train the model using a binary cross-entropy objective (`torch.nn.BCEWithLogitsLoss`), which is equivalent to minimizing the NLL (which you proved). Make sure that your trained model can sample images by generating pixels sequentially in raster order. *Note: The original PixelCNN paper incorporates residual blocks. You are not required to use them in your implementation, but you are also welcome to experiment with them if you wish.*

(d) [**10 pts**] Extend your `MaskedConv2d` layer to incorporate label conditioning (Van den Oord et al., 2016). This is done by adding a class-dependent bias term to each convolutional layer:

$$W_\ell * x + b_\ell + V_\ell y,$$

where $W_\ell * x + b_\ell$ is the masked convolution, y is a one-hot encoded class label, and V_ℓ is a learned weight matrix. The product $V_\ell y$ should be broadcast across the spatial dimensions and added channel-wise to the output.

You are also provided with a skeleton implementation of `ConditionalMaskedConv2d`, Your task is to complete this module and then use it to build a `ConditionalPixelCNN` model. The architecture should closely follow the standard PixelCNN, but with class-label conditioning applied in every layer through the additional bias. Train your model on MNIST, generated samples for a digit of your choice and compare the quality of unconditional versus class-conditional samples.

Step 3 (25 points): Variational Autoencoders (VAEs)

Your final task is to implement an *unconditional* VAE (Kingma & Welling, 2014) for MNIST. The goal is to train an encoder–decoder model that maximizes the ELBO on the training set and can both reconstruct inputs and generate novel digits from the prior. You are given skeleton codes `hw2_step3_main.py` and `models/VAE.py`. Make sure to check lines with the `#TODO` flag in order to complete the code. For this step, your goals are:

(a) [**4 pts**] In a Variational Autoencoder (VAE), the ELBO can sometimes be simplified so that parts of it do not require Monte Carlo sampling. Consider the case where the prior is a standard Gaussian $p(z) = \mathcal{N}(z; 0, I)$, and the approximate posterior, $q(z|x)$, is a Gaussian with mean $\mu(x)$ and diagonal covariance $\text{diag}(\sigma^2(x))$. Show that in this setting, the ELBO can be written as the expected reconstruction log-likelihood minus a closed-form KL divergence between two Gaussian distributions. Derive this decomposition step by step, making the KL term explicit, and show that:

$$\mathcal{L}(x) = \underbrace{\mathbb{E}_{q(z|x)}[\log p(x|z)]}_{\text{reconstruction term}} - \underbrace{\tfrac{1}{2}\sum_{j=1}^{d}\left(\sigma_j^2 + \mu_j^2 - 1 - \log \sigma_j^2\right)}_{\text{closed-form KL}}. \tag{2}$$

(b) [**5 pts**] Using Eq. 2, fill in the `loss_func` module in `hw2_step3_main.py`.

(c) [**10 pts**] Design and implement the architectures for both the encoder and the decoder in the given `models/VAE.py` file. Each should be built entirely with fully connected layers and use Leaky ReLU activations throughout (except for the output layers). Batch normalization is not required.

- **Encoder:** Construct with three linear layers as hidden units, followed by a final layer that outputs both the mean and log-variance for the latent space.
- **Decoder:** Construct symmetrically to the encoder, using three fully connected hidden layers. The final output layer should reconstruct an input of size $28 \times 28 = 784$, with a Sigmoid activation to ensure pixel values lie in $[0, 1]$.

You may select the hidden layer sizes from the set $\{16, 32, 64, 128, 256\}$. After training, visualize the learning process by:

- Show a comparison of original and reconstructed MNIST digits to evaluate how well the model learns to reproduce the input data.
- Visualize a grid of generated digits by decoding points arranged across the 2D latent space in order to understand the structure of the latent representation and the quality of generated samples.
- Plot the training loss curve (ELBO or its components) over epochs.

Include these figures in your report and **comment** on the reconstruction quality, the structure of the latent space, and the behavior of the training loss. Does the model learn a meaningful latent representation?

(d) [**6 pts**] Your model's performance will largely depend on the chosen hyperparameters (*e.g.*, learning rate, number and size of layers, regularization coefficient, number of epochs, etc.). **Experiment with different hyperparameter settings and justify your choices:** What did you try? Which configurations worked better, and why? Provide your observations and comments.

!General Note: We value your insights and discussion as much as the code and results you present. Therefore, make sure your report is thorough: **include explanations of your design choices, observations from trainings and results, comparisons of different approaches, reflections on what worked well (or not), and any other relevant insights you find important.** You may support your discussion with figures and examples wherever possible.

Deliverables → Upload 2 separate files: **a PDF report** (`lastname_EE8520_F25.pdf`) and **a zipped file** (`lastname_EE8520_F25.zip`), containing your codes.

References

Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, volume 27, 2014.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations*, 2014.

Aäron Van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional image generation with pixelcnn decoders. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

Aäron Van Den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International Conference on Machine Learning*, pp. 1747–1756. PMLR, 2016.