CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 10

This discussion focuses on deep unsupervised methods.

1 Generative Models

Generative models represent the full joint distribution $p(x, y)$ where x is an input sample and y is an output. In deep learning, we are interested in generating new samples that generalize a given dataset. In particular, this process may not require labeled data, as the goal is to understand some representation of the dataset distribution.

Broadly speaking, deep generative models include autoregressive models, autoencoder models and generative adversarial networks. In this discussion, we discuss deep generative models from the context of autoregressive models and autoencoder models.

2 Autoregressive Generative Models

An autoregressive model is a generative model that generates one sample at a time conditioned on its prior predictions.

In general, training autoregressive generative models divide up x into dimensions $x_1, \ldots, x_n$, and discretize them into k values. We finally model $p(x)$ via the chain rule. For example, when we try to generate one pixel at a time *conditioned* on pixel values from prior predictions, consider a $n \times n$ image with pixels in some order $(x_1, \ldots, x_{n^2}$. Then, we can define the model as,

$$p_\theta(x_1, \ldots, x_{n^2}) = \prod_{i=1}^{n^2} p_\theta(x_i | x_1, \ldots, x_{i-1})$$

Since the distribution $p(x_i|x_1, \ldots, x_{i-1})$ is complex, we model the distribution using a neural network, f_θ. Using the same example of generating pixels, we can begin at one corner, and proceed diagonally throughout the 2D spatial map, and use f_θ to sample pixels one at a time by conditioning on previously generated pixels.

Unfortunately, this work of generating and sampling each pixels conditioned on prior generated pixels is expensive. Instead, in practice, we use either **PixelRNN**, **PixelCNN** or **Pixel Transformer**.

PixelCNN Uses a CNN to model the probability of a pixel, given *previous* pixels. PixelCNN is slow at generating images, because there is a pass through the entire network for each pixel. But it is fast to train because there is no recurrence (only a single pass for the image) since the spatial maps are known in advance. During training the convolutions must be masked to ignore pixels at the same or later position in the pixel generation order.

PixelRNN PixelRNN uses RNN (or LSTM) to generate images. PixelRNN remembers the state from more distant pixels using the recurrent states. In particular, the PixelRNN uses recurrence instead of the 3×3 convolutions to allow long-range dependencies, and can generate a full row of pixels in one pass.

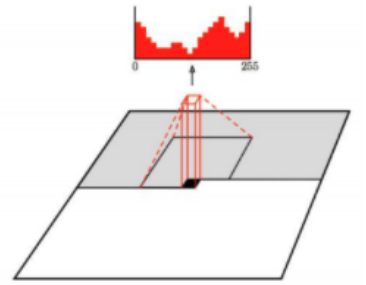

Figure 1: PixelCNN. Generate images from the corner, and dependency on previous pixels are modeled using a CNN over the context region

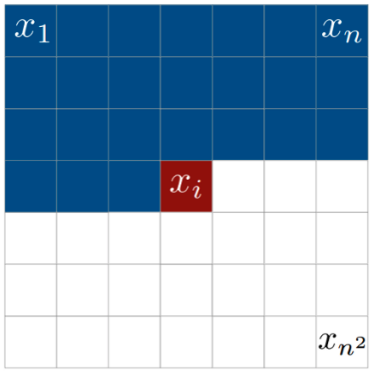

Figure 2: PixelRNN. This figure shows conditional probability. x_i is related to the previous points (blue part) and is unrelated to the point after it (white part)

Training is generally longer, though, due to the recurrence involved, since each row has its own hidden state in the LSTM layers.

PixelTransformer Pixel Transformers are similar to PixelCNN and PixelRNN, but they use a multi-headed attention network.

Problem 1: PixelCNN vs. Pixel RNN

What is the main difference between PixelCNN and PixelRNN? In particular, comment on:

- Run-time of PixelCNN and PixelRNN at training time
- Run-time of PixelCNN and PixelRNN at test time
- Generation of Pixels

3 Autoencoders

Autoencoders are methods to train a network to encode an image into some hidden state and then decode that image as accurately as possible from the hidden state. During this process, we force the autoencoder to learn a structured representation. Generally, autoencoders comprise of an *encoder* and a *decoder*, with a hidden state z. They are typically implemented as neural networks to compress the input into a smaller hidden state, and then decompressed through the decoder. Once the training is done, we can discard the second part of the network, and use z as the useful features for the original data.

These learned latent representations can be used on downstram tasks, like classification. For example, the VAE paper shows that VAE achieves adversarial robustness in downstream tasks on colorMNIST and CelebA datasets.

In particular, there are several key mechanisms to force the autoencoder to learn a structured representation of the data,

1. Dimensionality: Force the hidden state to be smaller than the input/output, so the network must compress information
2. Sparsity: Force the hidden state to be sparse, so the network must be compressed
3. Denoising: Corrupt the input with noise, and force the autoencoder to learn to distinguish noise from the signal
4. Probabilistic Modeling: Forces the hidden state with a prior distribution

In practice, Autoencoders are far less used today, since there exists better alternatives for both representation learning (VAEs, contrastive learning) and generation (GANs, VAEs, autoregressive models).

Bottleneck Autoencoder Bottleneck Autoencoder can be viewed as non-linear dimensionality reduction, and can be used as since dimensionality is lower and there are various algorithms tractable in low-dimensional spaces. This design is antiquated and rarely used. The idea is simple to implement, but reducing dimensionality often fails to provide the structure we want. When the number of hidden dimensions is larger than input/output, we call it *overcomplete*, and this may learn the identity function.

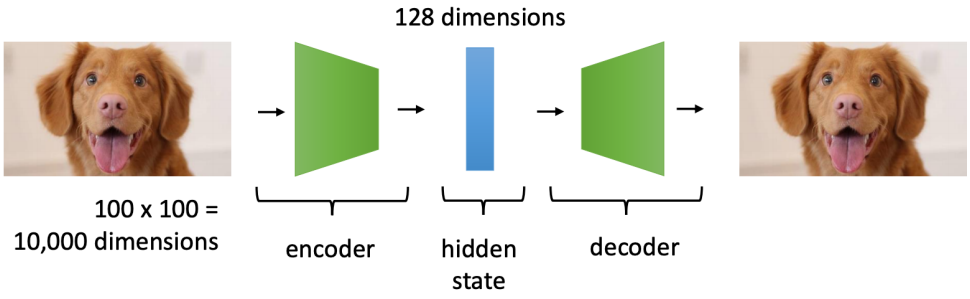

Figure 3: Classical bottleneck architecture reducing 10000 dimensions to 128 dimensions

Denoising Autoencoder Denoising Autoencoders corrupt the input with noise and runs a Bottleneck Autoencoder. However, there are many variants on this idea. In practice, it is unclear which layer to choose for the bottleneck, and there are some ad-hoc choices (e.g., how much noise to add).

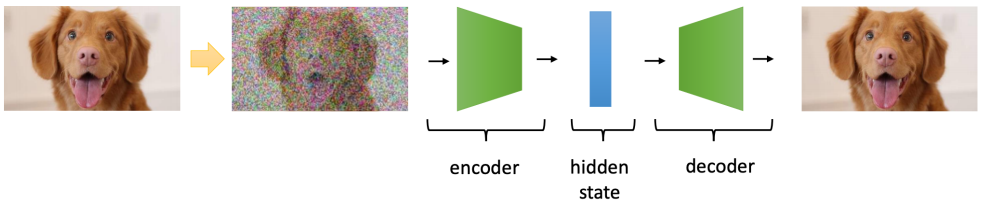

Figure 4: Denoising architecture with a Bottleneck

Sparse Autoencoder Sparse Autoencoder originates from sparse coding theory in the brain, and is an attempt to describe the input with a sparse representation, by letting most values to zero. In this autoencoder, the dimensionality may be very large, and uses a sparsity loss, $\sum_{j=1}^{D} |h_j|$. In practice, choosing the regularizer and adjusting hyperparameters can be very hard.

4 Latent Variable Models

Formally, a latent variable model p is a probability distribution over observed variables x and latent variables z (variables that are not directly observed but inferred), $p_\theta(x, z)$. Because we know z is unobserved, using learning methods learned in class (like supervised learning methods) are unsuitable.

Indeed, our learning problem of maximizing the log-likelihood of the data turns from,

$$\theta \leftarrow \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i)$$

to the following,

$$\theta \leftarrow \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log \int p_\theta(x_i|z)p(z)dz$$

where we recognize $p(x) = \int p(x|z)p(z)dz$. Unfortunately, the integral is intractable, but we will discuss ways to find a tractable lower bound.

4.1 Varational Autoencoders (VAE)

The VAE uses the autoencoder framework to generate new images. For the following description of encoder and decoder of the VAE, let us assume our input x is a 28×28 photo of a handwritten digit in black-and-white, and we wish to encode this information into a latent representation of space z.

Encoder Encoder maps a high-dimensional input x (like the pixels of an image) and then (most often) outputs the parameters of a Gaussian distribution that specify the hidden variable z. In other words, they output $\mu_{z|x}$ and $\Sigma_{z|x}$. We implement this as a deep neural network, parameterized by ϕ, which computes the probability $q_\phi(z|x)$. We can sample from this distribution to get noisy values of the representation z.

Decoder Decoder maps the latent representation back to a high dimensional reconstruction, denoted as $\hat{x}$, and outputs the parameters to the probability distribution of the data. We implement this as another neural network, parametrized by θ, which computes the probability $p_\theta(x|z)$. Following the digit example, if we represent each pixel as a 0 (black) or 1 (white), probability distribution of a single pixel can be then represented using a Bernoulli distribution. Indeed, the decoder gets as input the latent representation of a digit z and outputs 784 Bernoulli parameters, one for each of the 784 pixels in the image.

Training VAEs To train VAEs, we find parameters that maximize the likelihood of the data,

$$\theta \leftarrow \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log \int p_\theta(x_i|z)p(z)dz$$

This integral is intractable. However, we can show that it is possible to optimize a tractable lower bound on the data likelihood, called the Evidence Lower Bound (ELBO),

$$\mathcal{L}_i = \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log p_\theta(x_i|z)] - D_{KL}(q_\phi(z|x_i)||p(z))$$

Problem 2: Blurry Images

Why do VAEs typically produce blurry images?

4.2 Variational Inference

In this subsection, we will derive the variational approximation in discrete form, and discuss the re-parametrization trick.

Problem 3: Latent Variable Model

Write out the log-likelihood objective of a discrete latent variable model.

Problem 4: Variational Approximation

Show that

$$\sum_{i=1}^{N} \log p_\theta(x_i) \geq \sum_{i=1}^{n} \mathbb{E}_{q(z|x_i)}[\log p_Z(z) - \log q(z|x_i) + \log p_\theta(x_i|z)]$$

Hint: Use Jensen's Inequality, which states, $\log \mathbb{E}[X] \geq \mathbb{E}[\log X]$

Problem 5: Variational Approximation Optimization

To optimize the Variational Lower Bound derived in the previous problem, which distribution do we sample z from?

Combining it with Entropy Recall the entropy function,

$$H(p) = -\mathbb{E}[\log p(x)] = -\int_X p(x) \log p(x) dx$$

and also recall KL-Divergence,

$$D_{KL}(q||p) = \mathbb{E}\left[\log \frac{q(x)}{p(x)}\right] = -E[\log p(x)] - H(q)$$

We can show that, our derived approximation can be reformulated as the Evidence Lower Bound (ELBO),

$$\mathcal{L}_i = \mathbb{E}_{z \sim q_\phi(z|x_i)}[\log p_\theta(x_i|z)] - D_{KL}(q_\phi(z|x_i)||p(z))$$

Re-Parametrization Trick The re-parametrization trick allows us to break $q_\phi(z|x)$ into a deterministic and stochastic portion, and re-parametrize from $q_\phi(z|x)$ to $g_\phi(x, \epsilon)$. In fact, we can let, $z = g_\phi(x, \epsilon) = g_0(x) + \epsilon \cdot g_1(x)$ where $\epsilon \sim p(\epsilon)$. This reparametrization trick is simple to implement, and has low variance [1]

Problem 6: Reparametrization Example

Let us get an intuition for how we might use re-parametrization in practice. Assume we have a normal distribution q, parametrized by θ, such that, $q_\theta(x) \sim \mathcal{N}(\theta, 1)$, and we would like to solve,

$$\min_\theta \mathbb{E}_q[x^2]$$

Use the re-parametrization trick on x to derive the gradient.

4.3 Normalizing Flows

In Flows, we wish to map simple distributions (easy to sample and evaluate densities) to complex ones (learned via data), and they describe the transformation of a probability density through a sequence of

[1] See Appendix of this paper for more information

invertible mappings. Let us consider a directed, latent-variable model over observed variables X and latent variables Z.

In practice, we learn an invertible mapping $f_\theta : \mathbb{R}^n \to \mathbb{R}^n$ from z to x.

$$x = f_\theta(z)$$
$$z = f_\theta^{-1}(x)$$

We then, maximize the likelihood of $p(x)$.

Hence, we can interpret *Normalizing Flow*, as (1) we would like the change of variables give a normalized density at $\mathcal{N}(0, 1)$ after applying an invertible transformation, and (2) that the invertible transformations can be composed with each other to create more complex invertible transformations.

Problem 7: Flow Objective

In Flows, our training objective is to let maximize the log-likelihood of x, where we have,

$$x = f_\theta(z)$$
$$z = f_\theta^{-1}(x)$$

Write out the training objective explicitly, then use change of variables to derive the Normalizing Flow objective. Also derive the properties that f_θ must satisfy for practical flows.

There are two main flow models we discuss in class,

1. Nonlinear Independent Components Estimation (NICE)
2. Real Non-Volume Preserving Transformation (Real-NVP)

NICE NICE model composes two invertible transformations: additive coupling layers and rescaling layers. The coupling layer in NICE partitions a variable z into two disjoints subsets, $z_{1:d}$ and $z_{d+1:n}$. Then it applies the following forward mapping,

1. $x_{1:d} = z_{1:d}$ (identity mapping)
2. $x_{d+1:n} = z_{d+1:n} + g_\theta(z_{1:d})$ where g_θ is the neural net

and the following inverse mapping,

1. $z_{1:d} = x_{1:d}$ (identity mapping)
2. $z_{d+1:n} = x_{d+1:n} - h_\theta(z_{1:d})$ where h_θ is the neural net

Here, notice that the Jacobian of the forward map is lower triangular, whose determinant is simply the product of the elements on the diagonal, which is 1. Also, note that, then we have that the mapping is *volume preserving*, meaning that the transformed distribution p_x will have the same "volume" compared to the original one p_z.

Real-NVP Real-NVP adds scaling factors to the transformation,

$$x_{d+1:n} = \exp\big(h_\theta(z_{1:d})\big) \odot z_{d+1:n} + h_\theta(z_{1:d})$$

where $\odot$ represents element-wise product. This results in a non-volume preserving transformation.

Problem 8: Real-NVP Determinant

Determine the determinant of the Jacobian of the forward map of the Real-NVP. In other words, find,

$$|\det\left(\frac{df(z)}{dz}\right)|$$