Part IV: Pyro Fundamentals, Amortized Inference, and Variational Autoencoders

Robert Salomone

AMSI Winter School, 2021

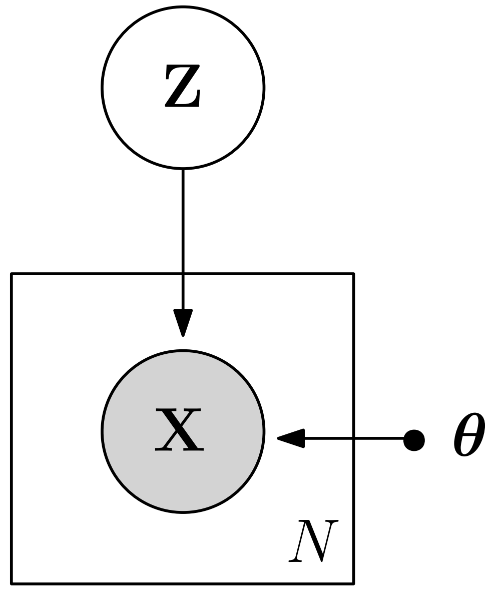
Amortized Inference

- Previously, you have seen that given some data $\boldsymbol{X}$, we can fit a **variational approximation** of the latent posterior $\boldsymbol{Z}|\boldsymbol{X}$.
- This is useful for training models via (approximate) maximum likelihood, or just flat out fitting an approximate distribution to some target distribution of interest.
- However, there is one weakness, for **new data**, we have to go and train everything all over again to get q_ϕ.
- However, we can be **smart** in the case where every observation has their own **local** latent variables.

Global vs. Local Latents

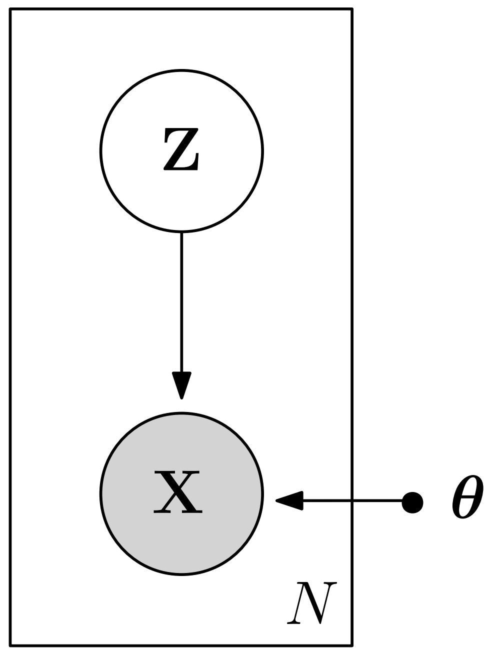
Global Latent Variables

The same latents generate all the data (e.g., $\boldsymbol{Z}$ is Bayesian parameters).



Local Latent Variables

Every observation has its own latent vector (e.g., Latent Representation).



Amortized Inference: In a Nutshell

- **Key Idea**
  - For $\boldsymbol{x}_k \in \boldsymbol{x}$ with local latent variables, with the techniques we have learned so far, we would need to fit individual variational approximations $q(\boldsymbol{z}_k|\boldsymbol{x}_k) \approx p(\boldsymbol{z}_k|\boldsymbol{x}_k)$.
  - Amortized Inference is the approach that tries **directly** learn the **mapping** $\boldsymbol{x} \mapsto p(\boldsymbol{z}|\boldsymbol{x})$ that will work for **any** $\boldsymbol{x}_k$ (including ones from outside the dataset that you may introduce later).
  - This, in some sense, exploits the conditional independence structure of the model.
  - We introduce an **inference network** (sometimes called a **recognition model**), g_ϕ.
- For a variational family $q_{\boldsymbol{\eta}}$, instead of optimizing $\boldsymbol{\eta}$ being the actual variational parameters that parametrize q, we will train some function g_ϕ that **outputs** the variational parameters.
  - Denoting $q(\,\cdot\,;\boldsymbol{\eta})$, the amortized variational approximation is (here, $\boldsymbol{x}$ denotes any **single** observation).

$$(\boldsymbol{Z}|\boldsymbol{X} = \boldsymbol{x}) \sim q(\,\cdot\,; g_\phi(\boldsymbol{x})).$$

Amortized Inference (Continued)

- Of course, g will be a neural net. Now, our variational approximation really does have the form $q(\boldsymbol{z}|\boldsymbol{x})$ if $\boldsymbol{x}$ denotes a single observation.
- Data goes in, **parameters** of a variational approximation come out.
- We train $g_{\boldsymbol{\phi}}$ by optimizing over $\boldsymbol{\phi}$ as we did earlier (e.g., maximizing the ELBO loss), but now we have a neural net in there.
- The main advantage of amortizing our variational inference: We train g_ϕ once, and then can infer an approximation of the distribution of $\boldsymbol{Z}|\boldsymbol{X}$ for some **new** $\boldsymbol{Z}$ without retraining.

Neural Nets all the way down...

- We can use neural networks with latent variables to obtain a very flexible $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ and/or a model with meaningful $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$.
- We can use a neural network to **amortize** approximate posterior inference as $q(\boldsymbol{z}|\boldsymbol{x}) \sim \mathrm{Dist}(\eta(\boldsymbol{x})) \approx p(\boldsymbol{z}|\boldsymbol{x})$ where η is a neural network mapping $\boldsymbol{x}$ to the parameters of a variational distribution.
    - The variational parameters **themselves** output by the inference network may themselves be the parameters of **another** neural network (or even sequence thereof!) that is used to define a flexible distribution via normalizing flows.
- Too many neural nets? That's not for me to say.

Back to Bayes-ics

- At this point, you can see why we wouldn't want to be fully Bayesian. If we have a very high-dimensional $\boldsymbol{\theta}$ (e.g., Deep Bernoulli model with say 10^6 parameters), then performing effective variational inference those latents would introduce additional challenges without much benefit.
    - Most Importantly: It would probably make your posterior inferences much worse on the latent variables you **actually really care about**, which may very low dimensional.

Variational Autoencoders

Bayesian Manifold Learning

- Next, we will introduce a **state-of-the-art** probabalistic model for dimensionality reduction. Let m denote the dimensionality of the latent space.

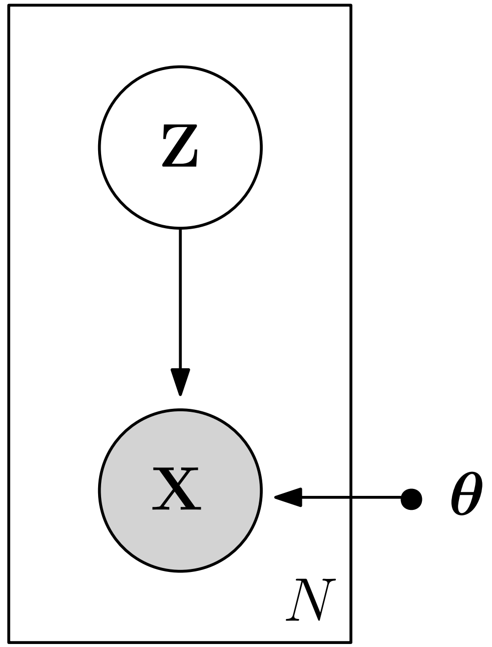
$$\boldsymbol{Z}_k \sim \mathcal{N}(\mathbf{0}, \mathrm{I}_m), , \quad k = 1, \dots, N.$$

$$\boldsymbol{X}_k | \boldsymbol{Z}_k \sim_{\text{ind}} \mathrm{Dist}(\,\cdot\,; g_{\boldsymbol{\theta}}(\boldsymbol{Z})), \quad k = 1, \dots, N.$$

- Now $g_{\boldsymbol{\theta}}$ is, of course, a sufficiently flexible neural network.

- Generally, we will have that $\dim(\boldsymbol{Z}) \ll \dim(\boldsymbol{X})$.

- The distribution of $\boldsymbol{X}$ can be whatever you want it to be, so long as it matches your data of interest (discrete, continuous, etc.).



Yes...that is the model.

Let's look at what the above model is saying:

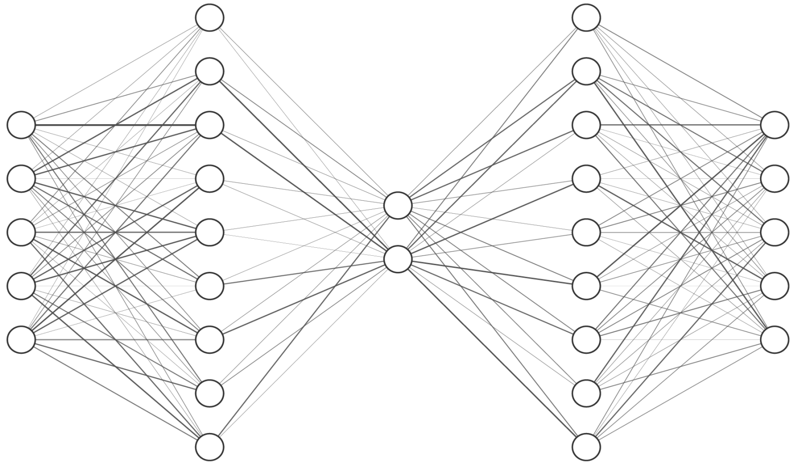
- Every observation relies on some lower dimensional latent vector (manifold hypothesis).
    - For an observation $\boldsymbol{X}$, the posterior distribution of $\boldsymbol{Z}|\boldsymbol{X}$ gives a **probability distribution** of its position on the underlying (lower-dimensional) manifold.
- The latent vector is **marginally a standard multivariate normal**.
    - This means generation is easy (assuming we fit the model): we can sample first from the manifold (which is disentangled) and then conditional on the manifold position.
    - Enforcing our "prior" to be of this form is equivalent to saying "**train the model so that the marginal distribution over the data for Z is independent and standard normal** (i.e., find a disentangled, lower-dimensional **representation** of the generating distribution).
- Note also that my "Deep Bernoulli" model from earlier fits the above description. We used latents to create a flexible multivariate family, but posterior inference over $\boldsymbol{Z}|\boldsymbol{X}$ also has above manifold interpretation.

Solving the Problem using Amortized Inference

- Now, train that model with amortized inference.
- **That** model is a VAE.

Traditional Autoencoders

- An **autoencoder** is a classic-ish neural network architecture for dimensionality reduction / manifold learning.



- Can think about it as two neural networks trained together $g_{\text{encoder}} : \mathcal{X} \to \mathcal{Z}$ and $g_{\text{decoder}} : \mathcal{Z} \to \mathcal{X}$.The bottleneck in the centre (two-nodes above), $\boldsymbol{z}$ is called the **learned representation**.
- The VAE model does this **probabilistically**, where the (amortized) variational approximation plays the role of "encoder" , and the part of model $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ takes the role of a "decoder".
    - People often insist on explaining VAE only with the above, but I think this hides what is **really** going on (variational learning, amortized inference, latent variables), which is important to understand extensions.

VAE In a Nutshell

- So, the VAE is a model that allows for all of the following:
  - Posterior Inference on $\boldsymbol{Z}$: You give me an $\boldsymbol{X}$, I give you the distribution on $\boldsymbol{Z}|\boldsymbol{X}$. Thus, we get a **distribution** over where the sample lives in (disentangled) latent space.
  - Forward Simulation: Draw $\boldsymbol{Z} \sim \mathcal{N}(\mathbf{0}, \mathrm{I})$ (manifold position) and return $\boldsymbol{X} \sim \mathrm{Dist}(g_{\boldsymbol{\theta}}(\boldsymbol{Z}))$
  - **Amortized Inference**: I can compute (approximately) $\boldsymbol{Z}|\boldsymbol{X}$ without redoing variational inference! This can be viewed as "encoding" $\boldsymbol{X}$ probabilistically.
- Why is this important? After training the model **once**, I can not extract a **latent representation** for **new data** without any issues.
- Also, by design, the latent space is **disentangled**. Our model posits that the population our data comes can be explained by a latent space where features are orthogonal (independent).
- **Variational**: it is trained with variational learning / **Autoencoder**: the end-product resembles a (probabalistic) autoencoder
- **Main benefit:** Disentanglement, the "prior" $\mathcal{N}(\mathbf{0}, \mathrm{I})$ on $\boldsymbol{Z}$ enforces this.

References

- As mentioned, the approach we are discussing here was simultaneously (!) proposed by two papers

1. Rezende, D. et al., 2014, **Stochastic Backpropagation and Approximate Inference in Deep Generative Models**
2. Kingma, D. & Welling M., 2014, **Auto-Encoding Variational Bayes**

Extensions of VAE

- **Better Disentanglement via Different Loss Functions**: e.g., **Beta-VAE** and **FactorVAE**. Remember, we aren't maximizing the likelihood, only a lower bound on it! So if disentanglement is the main goal, improvements can be made in that direction.
- **Neural Statistician** (Learns **summary statistics** from a collection of individual **datasets**) / **Neural Processes**: Generalization of Gaussian Processes (distributions over functions).
- **Conditional VAE / Neural Process** (of course!)

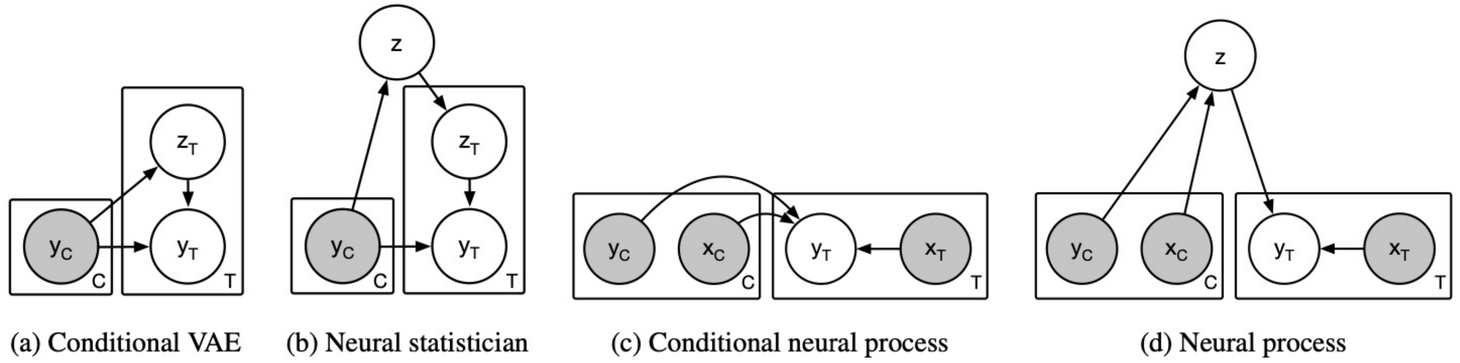


Figure 2. Graphical models of related models (a-c) and of the neural process (d). Gray shading indicates the variable is observed. C stands for context variables and T for target variables i.e. the variables to predict given C.

Figure from Garnelo et al (2018)., **Neural Processes**, International Conference of Machine Learning (ICML) 2018.

Exact Likelihood Gradients

- See the recent work **Unbiased Gradient Estimation for Variational Auto-Encoders using Coupled Markov Chains** by Ruiz et al. (June 2021).
- Can train a VAE via **actual maximum likelihood** (not lower bound). Uses an **unbiased MCMC** approach.
    - Again, being "semi-Bayesian" hopefully keeps the posterior dimension manageable for the MCMC!

Conditional VAE

- At this point, we can see if I have achieved my goal of having this picture make some sense to you...

$$\begin{tikzcd}
\mathbf{X} \sim p_{\mathbf{X},y=\text{photo}} \arrow[dr] & & \mathbf{X}|\mathbf{Z} \sim p_{\mathbf{X}|\mathbf{Z},y=\text{cartoon}} \\
& (Z_1, Z_2, \ldots, Z_m) \sim p_{\mathbf{Z}|\mathbf{X},y=\text{photo}} \arrow[ur] &
\end{tikzcd}$$

(posterior over low-dimensional latent random vector)

Pyro Fundamentals

First things first....

- Pyro is a probabalistic programming language.
- Then again, so is `Stan`, `Turing`, `PyMC`, etc.
- Pyro can do all the things one can do in the above, but its intended purpose is **deep probabalistic modelling**. Pyro allows for **very flexible** models:
    - It is designed to allow us to have neural nets everywhere wherever we desire.
    - It is designed for inference methods that are highly scalable in d and in n - naturally it thus is focussed on the use of Variational inference.
    - Flow-based Distributions can be incorporated into models (as well as used for variational inference).
- Pyro is **universal**: models can return other models, recursion is allowed, as is things like random number of variables, random models, you can be Bayesian about whatever you like, you can go as deep as you like, etc.

  "*It is worth emphasizing that this is one reason why Pyro is built on top of PyTorch: dynamic computational graphs are an important ingredient in allowing for universal models that can benefit from GPU-accelerated tensor math.*"

More about Pyro.

- It is actually quite ridiculous what it is capable of, but most people will never use such features.
- However, those features include a very nice variational inference engine with stochastic backpropagation that will allow us to play around a little (and implement VAEs reasonably painlessly)
- For more details, see the initial Pyro paper.
- **My goal isn't to make you an expert in Pyro, but to give you the tools to understand how to follow its tutorials and play around once you become more familiar with PyTorch.**

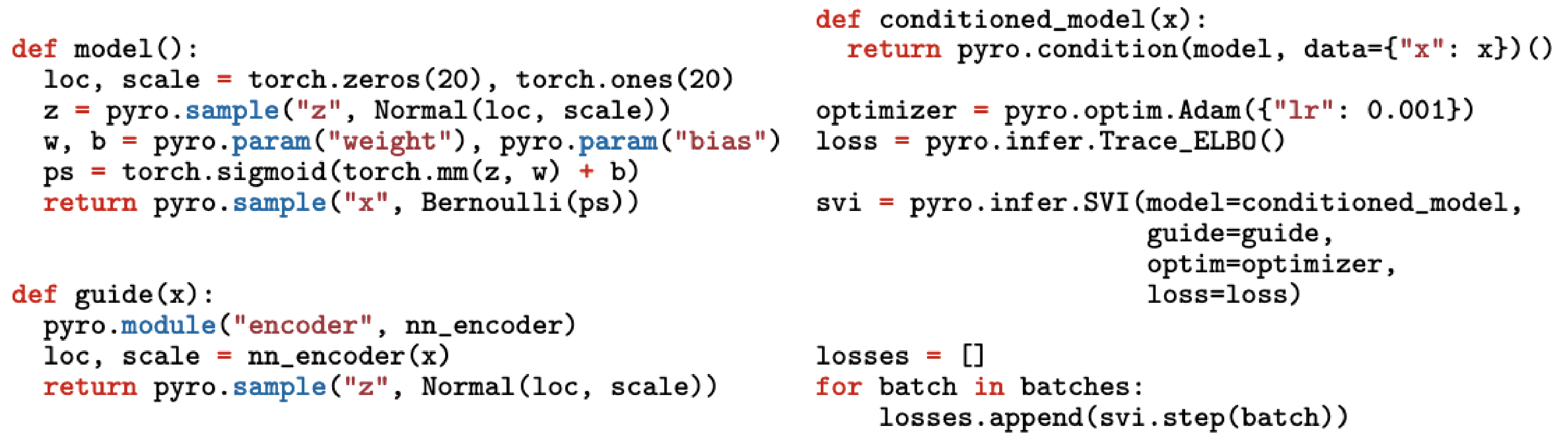
Pyro

- `model`: this is a *stochastic function* that specifies $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$
- `guide`: used to specify $q_{\phi}(\boldsymbol{z}|\,\boldsymbol{x})$

- `param`: this is something that will be **optimized over** in the objective
    - `param` in `model`: elements of $\boldsymbol{\theta}$
    - `param` in `guide`: elements of $\boldsymbol{\phi}$

- More generally, in `Pyro`, a `guide` is anything that approximates a posterior distribution.
    - We will only be looking at variational approximations.
    - However, one can also implement particle filters (importance sampling) or other inference approaches (e.g., Stein Variational Gradient Descent).

Brief Pyro Overview



```
def model():
  loc, scale = torch.zeros(20), torch.ones(20)
  z = pyro.sample("z", Normal(loc, scale))
  w, b = pyro.param("weight"), pyro.param("bias")
  ps = torch.sigmoid(torch.mm(z, w) + b)
  return pyro.sample("x", Bernoulli(ps))


def guide(x):
  pyro.module("encoder", nn_encoder)
  loc, scale = nn_encoder(x)
  return pyro.sample("z", Normal(loc, scale))
```

```
def conditioned_model(x):
  return pyro.condition(model, data={"x": x})()

optimizer = pyro.optim.Adam({"lr": 0.001})
loss = pyro.infer.Trace_ELBO()

svi = pyro.infer.SVI(model=conditioned_model,
                     guide=guide,
                     optim=optimizer,
                     loss=loss)

losses = []
for batch in batches:
    losses.append(svi.step(batch))
```

Figure 1: A complete Pyro example: the generative model (`model`), approximate posterior (`guide`), constraint specification (`conditioned_model`), and stochastic variational inference (`svi`, `loss`) in a variational autoencoder. `encoder` is a `torch.nn.Module` object. `pyro.module` calls `pyro.param` on every parameter of a `torch.nn.Module`.

Figure from the original Pyro paper, showcasing the structure of a model and its inference.

A Brief Mention Regarding NumPyro

- This is a second backend for Pyro built on **Jax**. This is useful as it allows very fast compilation for HMC.
  - **NumPyro even has Subsampling MCMC** (HMC variant)!
  - It also has some nice features, see **here**.
  - Syntax quite similar to standard Pyro but there are differences.
  - We will focus on standard `Pyro` as it is built on `PyTorch` which we have seen, and there are more features for deep probabalistic programming there.
  - Both are very good, but
    - `NumPyro` is what you should use if you want to do a lot of MCMC / Bayesian Statistics. There are also a lot more resources for NumPyro for this purpose. On the other hand, there are no flows, and it doesn't have certain features used more for AI.
    - `Pyro` is what you should use if you want to do deep probabalistic models or want to use variational learning mostly.
- This may change over time, `Pyro` Developers are working on a framework `pyro.api` that can dispatch a model to either backend.

Pyro Fundamentals

- The remainder of this lecture is demonstrated in **this Jupyter Notebook** which doubles as a tutorial workbook.