CSC413/2516 Lecture 10:
Generative Models & Reinforcement Learning

Jimmy Ba
In generative modeling, we’d like to train a network that models a distribution, such as a distribution over images.

We have seen a few approaches to generative modeling:

- Autoregressive models (Lectures 3, 7, and 8)
- Generative adversarial networks (last lecture)
- Reversible architectures (this lecture)
- Variational autoencoders (this lecture)

All four approaches have different pros and cons.
Generator Networks

- Start by sampling the code vector $z$ from a fixed, simple distribution (e.g. spherical Gaussian).
- The generator network computes a differentiable function $G$ mapping $z$ to an $x$ in data space.

https://blog.openai.com/generative-models/
We have seen how to learn generator networks by training a discriminator in GANs.

Problem:
- Learning can be very unstable. Need to tune many hyperparameters.
- No direct evaluation metric to assess the trained generator networks.

Idea: learn the generator directly via change of variables. (Calculus!)
Change of Variables Formula

- Let $f$ denote a differentiable, bijective mapping from space $\mathcal{Z}$ to space $\mathcal{X}$. (i.e., it must be 1-to-1 and cover all of $\mathcal{X}$.)
- Since $f$ defines a one-to-one correspondence between values $z \in \mathcal{Z}$ and $x \in \mathcal{X}$, we can think of it as a change-of-variables transformation.
- Change-of-Variables Formula from probability theory: if $x = f(z)$, then
  \[ p_X(x) = p_Z(z) \left| \det \left( \frac{\partial x}{\partial z} \right) \right|^{-1} \]
- Intuition for the Jacobian term:

![Diagram]

- Intuition for the Jacobian term: small $\partial x/\partial z$, high density $p(x)$, large $\partial x/\partial z$, low density $p(x)$
Suppose we have a generator network which computes the function $f$. It’s tempting to apply the change-of-variables formula in order to compute the density $p_X(x)$.

I.e., compute $z = f^{-1}(x)$

$$p_X(x) = p_Z(z) \left| \det \left( \frac{\partial x}{\partial z} \right) \right|^{-1}$$

Problems?
Suppose we have a generator network which computes the function \( f \). It’s tempting to apply the change-of-variables formula in order to compute the density \( p_X(x) \).

I.e., compute \( z = f^{-1}(x) \)

\[
p_X(x) = p_Z(z) \left| \det \left( \frac{\partial x}{\partial z} \right) \right|^{-1}
\]

Problems?

- It needs to be differentiable, so that the Jacobian \( \frac{\partial x}{\partial z} \) is defined.
- The mapping \( f \) needs to be invertible, with an easy-to-compute inverse.
- We need to be able to compute the (log) determinant.

Differentiability is easy (just use a differentiable activation function), but the other requirements are trickier.
Now let’s define a reversible block which is invertible and has a tractable determinant.

Such blocks can be composed.

- Inversion: \( f^{-1} = f_1^{-1} \circ \cdots \circ f_k^{-1} \)
- Determinants: \( \left| \frac{\partial x_k}{\partial z} \right| = \left| \frac{\partial x_k}{\partial x_{k-1}} \right| \cdots \left| \frac{\partial x_2}{\partial x_1} \right| \left| \frac{\partial x_1}{\partial z} \right| \)
Reversible Blocks

- Recall the residual blocks:
  \[ y = x + \mathcal{F}(x) \]

- Reversible blocks are a variant of residual blocks. Divide the units into two groups, \( x_1 \) and \( x_2 \).
  \[ y_1 = x_1 + \mathcal{F}(x_2) \]
  \[ y_2 = x_2 \]

- Inverting a reversible block:
  \[ x_2 = y_2 \]
  \[ x_1 = y_1 - \mathcal{F}(x_2) \]
Reversible Blocks

Composition of two reversible blocks, but with $x_1$ and $x_2$ swapped:

- **Forward:**
  
  $y_1 = x_1 + \mathcal{F}(x_2)$
  
  $y_2 = x_2 + \mathcal{G}(y_1)$

- **Backward:**

  $x_2 = y_2 - \mathcal{G}(y_1)$
  
  $x_1 = y_1 - \mathcal{F}(x_2)$
Volume Preservation

- It remains to compute the log determinant of the Jacobian.
- The Jacobian of the reversible block:

\[
\begin{align*}
\mathbf{y}_1 &= \mathbf{x}_1 + \mathbf{F}(\mathbf{x}_2) \\
\mathbf{y}_2 &= \mathbf{x}_2
\end{align*}
\]

\[
\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix}
\mathbf{I} & \frac{\partial \mathbf{F}}{\partial \mathbf{x}_2} \\
\mathbf{0} & \mathbf{I}
\end{pmatrix}
\]

- This is an upper triangular matrix. The determinant of an upper triangular matrix is the product of the diagonal entries, or in this case, 1.
- Since the determinant is 1, the mapping is said to be volume preserving.
Nonlinear Independent Components Estimation

- We’ve just defined the reversible block.
  - Easy to invert by subtracting rather than adding the residual function.
  - The determinant of the Jacobian is 1.
- Nonlinear Independent Components Estimation (NICE) trains a generator network which is a composition of lots of reversible blocks.
- We can compute the likelihood function using the change-of-variables formula:
\[
p_X(x) = p_Z(z) \left| \det \left( \frac{\partial x}{\partial z} \right) \right|^{-1} = p_Z(z)
\]
- We can train this model using maximum likelihood. I.e., given a dataset \( \{x^{(1)}, \ldots, x^{(N)}\} \), we maximize the likelihood
\[
\prod_{i=1}^{N} p_X(x^{(i)}) = \prod_{i=1}^{N} p_Z(f^{-1}(x^{(i)}))
\]
Nonlinear Independent Components Estimation

- Likelihood:
  \[ p_X(x) = p_Z(z) = p_Z(f^{-1}(x)) \]

- Remember, \( p_Z \) is a simple, fixed distribution (e.g. independent Gaussians)

- Intuition: train the network such that \( f^{-1} \) maps each data point to a high-density region of the code vector space \( Z \).
  - Without constraints on \( f \), it could map everything to \( 0 \), and this likelihood objective would make no sense.
  - But it can’t do this because it’s volume preserving.
Nonlinear Independent Components Estimation

Dinh et al., 2016. Density estimation using RealNVP.
Nonlinear Independent Components Estimation

Samples produced by RealNVP, a model based on NICE.

ImageNet  celebrities  bedrooms

Dinh et al., 2016. Density estimation using RealNVP.
RevNets (optional)

- A side benefit of reversible blocks: you don’t need to store the activations in memory to do backprop, since you can reverse the computation.
  - I.e., compute the activations as you need them, moving backwards through the computation graph.
- Notice that reversible blocks look a lot like residual blocks.
- We recently designed a reversible residual network (RevNet) architecture which is like a ResNet, but with reversible blocks instead of residual blocks.
  - Matches state-of-the-art performance on ImageNet, but without the memory cost of activations!

Jimmy Ba
CSC413/2516 Lecture 10: Generative Model
15 / 40
Latent Space Interpolations

- You can often get interesting results by interpolating between two vectors in the latent space:

Ha and Eck, “A neural representation of sketch drawings”
Latent Space Interpolations

Latent space interpolation of music:
https://magenta.tensorflow.org/music-vae
Trade-offs of Generative Approaches

- So far, we have seen four different approaches:
  - Autoregressive models (Lectures 3, 7, and 8)
  - Generative adversarial networks (last lecture)
  - Reversible architectures (this lecture)
  - Variational autoencoders (optional)

- They all have their own pro and con. We often pick a method based on our application needs.

- Some considerations for computer vision applications:
  - Do we need to evaluate log likelihood of new data?
  - Do we prefer good samples over evaluation metric?
  - How important is representation learning, i.e. meaningful code vectors?
  - How much computational resource can we spent?
## Trade-offs of Generative Approaches

### In summary:

<table>
<thead>
<tr>
<th></th>
<th>Log-likelihood</th>
<th>Sample</th>
<th>Representation</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoregressive</td>
<td><strong>Tractable</strong></td>
<td><strong>Good</strong></td>
<td>Poor</td>
<td>O(#pixels)</td>
</tr>
<tr>
<td>GANs</td>
<td>Intractable</td>
<td><strong>Good</strong></td>
<td><strong>Good</strong></td>
<td>O(#layers)</td>
</tr>
<tr>
<td>Reversible</td>
<td><strong>Tractable</strong></td>
<td>Poor</td>
<td>Poor</td>
<td>O(#layers)</td>
</tr>
<tr>
<td>VAEs (optional)</td>
<td><strong>Tractable</strong></td>
<td>Poor</td>
<td><strong>Good</strong></td>
<td>O(#layers)</td>
</tr>
</tbody>
</table>

There is no silver bullet in generative modeling.
After the break: **Reinforcement Learning: Policy Gradient**
Most of this course was about supervised learning, plus a little unsupervised learning.

Reinforcement learning:
- Middle ground between supervised and unsupervised learning
- An agent acts in an environment and receives a reward signal.

Today: policy gradient (directly do SGD over a stochastic policy using trial-and-error)

Next lecture: combine policies and Q-learning
Reinforcement learning

- An **agent** interacts with an **environment** (e.g. game of Breakout)
- In each time step $t$,
  - the agent receives **observations** (e.g. pixels) which give it information about the **state** $s_t$ (e.g. positions of the ball and paddle)
  - the agent picks an **action** $a_t$ (e.g. keystrokes) which affects the state
- The agent periodically receives a **reward** $r(s_t, a_t)$, which depends on the state and action (e.g. points)
- The agent wants to learn a **policy** $\pi_\theta(a_t \mid s_t)$
  - Distribution over actions depending on the current state and parameters $\theta$
The environment is represented as a Markov decision process $\mathcal{M}$.

Markov assumption: all relevant information is encapsulated in the current state; i.e. the policy, reward, and transitions are all independent of past states given the current state.

Components of an MDP:
- initial state distribution $p(s_0)$
- policy $\pi_{\theta}(a_t | s_t)$
- transition distribution $p(s_{t+1} | s_t, a_t)$
- reward function $r(s_t, a_t)$

Assume a fully observable environment, i.e. $s_t$ can be observed directly.

Rollout, or trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$

Probability of a rollout

$$p(\tau) = p(s_0) \pi_{\theta}(a_0 | s_0) p(s_1 | s_0, a_0) \cdots p(s_T | s_{T-1}, a_{T-1}) \pi_{\theta}(a_T | s_T)$$
Markov Decision Processes

Continuous control in simulation, e.g. teaching an ant to walk

- State: positions, angles, and velocities of the joints
- Actions: apply forces to the joints
- Reward: distance from starting point
- Policy: output of an ordinary MLP, using the state as input
- More environments: https://gym.openai.com/envs/#mujoco
Markov Decision Processes

- **Return** for a rollout: \( r(\tau) = \sum_{t=0}^{T} r(s_t, a_t) \)
  - Note: we’re considering a finite horizon \( T \), or number of time steps; we’ll consider the infinite horizon case later.

- **Goal**: maximize the expected return, \( R = \mathbb{E}_{p(\tau)}[r(\tau)] \)

- The expectation is over both the environment’s dynamics and the policy, but we only have control over the policy.

- The stochastic policy is important, since it makes \( R \) a continuous function of the policy parameters.
  - Reward functions are often discontinuous, as are the dynamics (e.g. collisions)
**REINFORCE** is an elegant algorithm for maximizing the expected return $R = \mathbb{E}_{p(\tau)} [r(\tau)]$.

**Intuition: trial and error**
- Sample a rollout $\tau$. If you get a high reward, try to make it more likely. If you get a low reward, try to make it less likely.

**Interestingly, this can be seen as stochastic gradient ascent on $R$.**
REINFORCE

- Recall the derivative formula for log:

\[
\frac{\partial}{\partial \theta} \log p(\tau) = \frac{\partial}{\partial \theta} \frac{p(\tau)}{p(\tau)} \quad \Rightarrow \quad \frac{\partial}{\partial \theta} p(\tau) = p(\tau) \frac{\partial}{\partial \theta} \log p(\tau)
\]

- Gradient of the expected return:

\[
\frac{\partial}{\partial \theta} \mathbb{E}_{p(\tau)} [r(\tau)] = \frac{\partial}{\partial \theta} \sum_{\tau} r(\tau) p(\tau)
\]

\[
= \sum_{\tau} r(\tau) \frac{\partial}{\partial \theta} p(\tau)
\]

\[
= \sum_{\tau} r(\tau) p(\tau) \frac{\partial}{\partial \theta} \log p(\tau)
\]

\[
= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right]
\]

- Compute stochastic estimates of this expectation by sampling rollouts.
REINFORCE

- For reference:
  \[
  \frac{\partial}{\partial \theta} \mathbb{E}_{p(\tau)} [r(\tau)] = \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right]
  \]

- If you get a large reward, make the rollout more likely. If you get a small reward, make it less likely.

- Unpacking the REINFORCE gradient:
  \[
  \frac{\partial}{\partial \theta} \log p(\tau) = \frac{\partial}{\partial \theta} \log \left[ p(s_0) \prod_{t=0}^{T} \pi_\theta(a_t \mid s_t) \prod_{t=1}^{T} p(s_t \mid s_{t-1}, a_{t-1}) \right]
  \]
  \[
  = \frac{\partial}{\partial \theta} \log \prod_{t=0}^{T} \pi_\theta(a_t \mid s_t)
  \]
  \[
  = \sum_{t=0}^{T} \frac{\partial}{\partial \theta} \log \pi_\theta(a_t \mid s_t)
  \]

- Hence, it tries to make all the actions more likely or less likely, depending on the reward. I.e., it doesn’t do credit assignment.
  
  - This is a topic for next lecture.
REINFORCE

Repeat forever:

Sample a rollout $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$

$r(\tau) \leftarrow \sum_{k=0}^{T} r(s_k, a_k)$

For $t = 0, \ldots, T$:

$\theta \leftarrow \theta + \alpha r(\tau) \frac{\partial}{\partial \theta} \log \pi_\theta(a_t | s_t)$

- Observation: actions should only be reinforced based on future rewards, since they can’t possibly influence past rewards.
- You can show that this still gives unbiased gradient estimates.

Repeat forever:

Sample a rollout $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$

For $t = 0, \ldots, T$:

$r_t(\tau) \leftarrow \sum_{k=t}^{T} r(s_k, a_k)$

$\theta \leftarrow \theta + \alpha r_t(\tau) \frac{\partial}{\partial \theta} \log \pi_\theta(a_t | s_t)$
Optimizing Discontinuous Objectives

- A classification task under RL formulation
  - one time step
  - state \( x \): an image
  - action \( a \): a digit class
  - reward \( r(x, a) \): 1 if correct, 0 if wrong
  - policy \( \pi(a | x) \): a distribution over categories
    - Compute using an MLP with softmax outputs – this is a policy network
Optimizing Discontinuous Objectives

- Original solution: use a surrogate loss function, e.g. logistic-cross-entropy
- RL formulation: in each episode, the agent is shown an image, guesses a digit class, and receives a reward of 1 if it’s right or 0 if it’s wrong
- We’d never actually do it this way, but it will give us an interesting comparison with backprop
Optimizing Discontinuous Objectives

- Let $z_k$ denote the logits, $y_k$ denote the softmax output, $t$ the integer target, and $t_k$ the target one-hot representation.
- To apply REINFORCE, we sample $a \sim \pi_\theta(\cdot | x)$ and apply:

\[
\theta \leftarrow \theta + \alpha r(a, t) \frac{\partial}{\partial \theta} \log \pi_\theta(a | x) \\
= \theta + \alpha r(a, t) \frac{\partial}{\partial \theta} \log y_a \\
= \theta + \alpha r(a, t) \sum_k (a_k - y_k) \frac{\partial}{\partial \theta} z_k
\]

- Compare with the logistic regression SGD update:

\[
\theta \leftarrow \theta + \alpha \frac{\partial}{\partial \theta} \log y_t \\
\leftarrow \theta + \alpha \sum_k (t_k - y_k) \frac{\partial}{\partial \theta} z_k
\]
Reward Baselines

For reference:

$$\theta \leftarrow \theta + \alpha r(a, t) \frac{\partial}{\partial \theta} \log \pi_\theta(a | x)$$

Clearly, we can add a constant offset to the reward, and we get an equivalent optimization problem.

Behavior if $r = 0$ for wrong answers and $r = 1$ for correct answers

- wrong: do nothing
- correct: make the action more likely

If $r = 10$ for wrong answers and $r = 11$ for correct answers

- wrong: make the action more likely
- correct: make the action more likely (slightly stronger)

If $r = -10$ for wrong answers and $r = -9$ for correct answers

- wrong: make the action less likely
- correct: make the action less likely (slightly weaker)
Reward Baselines

- Problem: the REINFORCE update depends on arbitrary constant factors added to the reward.
- Observation: we can subtract a baseline $b$ from the reward without biasing the gradient.

$$
\mathbb{E}_{p(\tau)} \left[ (r(\tau) - b) \frac{\partial}{\partial \theta} \log p(\tau) \right] = \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] - b \mathbb{E}_{p(\tau)} \left[ \frac{\partial}{\partial \theta} \log p(\tau) \right]
$$

$$
= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] - b \sum_{\tau} p(\tau) \frac{\partial}{\partial \theta} \log p(\tau)
$$

$$
= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] - b \sum_{\tau} \frac{\partial}{\partial \theta} p(\tau)
$$

$$
= \mathbb{E}_{p(\tau)} \left[ r(\tau) \frac{\partial}{\partial \theta} \log p(\tau) \right] - 0
$$

- We’d like to pick a baseline such that good rewards are positive and bad ones are negative.
- $\mathbb{E}[r(\tau)]$ is a good choice of baseline, but we can’t always compute it easily. There’s lots of research on trying to approximate it.
More Tricks

- We left out some more tricks that can make policy gradients work a lot better.
  - Natural policy gradient corrects for the geometry of the space of policies, preventing the policy from changing too quickly.
  - Rather than use the actual return, evaluate actions based on estimates of future returns. This is a class of methods known as actor-critic, which we’ll touch upon next lecture.

- Trust region policy optimization (TRPO) and proximal policy optimization (PPO) are modern policy gradient algorithms which are very effective for continuous control problems.
Evolution Strategies

- REINFORCE can handle discontinuous dynamics and reward functions, but it requires a differentiable network since it computes \( \frac{\partial}{\partial \theta} \log \pi_\theta(a_t | s_t) \)

- Evolution strategies (ES) take the policy gradient idea a step further, and avoid backprop entirely.

- ES can use deterministic policies. It randomizes over the choice of policy rather than over the choice of actions.
  - I.e., sample a random policy from a distribution \( p_\eta(\theta) \) parameterized by \( \eta \) and apply the policy gradient trick
    \[
    \frac{\partial}{\partial \eta} \mathbb{E}_{\theta \sim p_\eta} [r(\tau(\theta))] = \mathbb{E}_{\theta \sim p_\eta} \left[ r(\tau(\theta)) \frac{\partial}{\partial \eta} \log p_\eta(\theta) \right]
    \]

- The neural net architecture itself can be discontinuous.
Algorithm 1 Evolution Strategies

1: **Input:** Learning rate $\alpha$, noise standard deviation $\sigma$, initial policy parameters $\theta_0$
2: for $t = 0, 1, 2, \ldots$ do
3: Sample $\epsilon_1, \ldots, \epsilon_n \sim \mathcal{N}(0, I)$
4: Compute returns $F_i = F(\theta_t + \sigma \epsilon_i)$ for $i = 1, \ldots, n$
5: Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
6: end for

Evolution Strategies

- The IEEE floating point standard is nonlinear, since small enough numbers get truncated to zero.

- This acts as a discontinuous activation function, which ES is able to handle.

- ES was able to train a good MNIST classifier using a “linear” activation function.

What’s so great about backprop and gradient descent?

- Backprop does credit assignment – it tells you exactly which activations and parameters should be adjusted upwards or downwards to decrease the loss on some training example.
- REINFORCE doesn’t do credit assignment. If a rollout happens to be good, all the actions get reinforced, even if some of them were bad.
- Reinforcing all the actions as a group leads to random walk behavior.
Discussion

- Why policy gradient?
  - Can handle discontinuous cost functions
  - Don’t need an explicit model of the environment, i.e. rewards and dynamics are treated as black boxes
    - Policy gradient is an example of model-free reinforcement learning, since the agent doesn’t try to fit a model of the environment
    - Almost everyone thinks model-based approaches are needed for AI, but nobody has a clue how to get it to work