10-707: Advanced Deep Learning **Spring 2020**

Lecture 18: April 6

Lecturer: Andrej Risteski *Scribes: Vineet Jain, Jiacheng Zhu*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

18.1 Invertible Models

18.1.1 Recap of GANs

The main idea behind GANs [GAN14]:

- Matching distributions in spaces such as that of images is hard because we don't have a good measure of "distance" between images. Intuitively, two images could be very different in pixel space, while "semantically" being the same image.
- Instead of fitting "maximum likelihood", but instead trying to learn some distribution close to the distribution of the input data in a learned metric.
- This type of model is "likelihood-free", i.e., we won't be able to explicitly write a likelihood for the model, but will be able to draw samples from it.

Goal: Learn a distribution close to some distribution we have few samples from.
Approach: Fit a distribution P_g parametrized by neural network g.
There are two neural networks: the Generator, which generates samples and the Discriminator, which distinguishes between "real" and "fake" samples.

From a game theoretic perspective, we can think that the generator is trained to fool the discriminator, while the discriminator is trained to beat the generator.

Common training problems with GANs:

- Unstable training: the problem is a min-max problem, which is typically much less stable than pure minimization.
- Vanishing gradient: if the discriminator is too good, the generator gradients have a propensity to be small. Modern GAN models have come up with ways to mitigate this.
- Mode collapse: the training only recovers some of the modes of the underlying distribution. This is not understood very well from a theoretical perspective.

18.1.2 Invertible GANs

Consider $g : \mathbb{R}^d \to \mathbb{R}^d$ is invertible. If $\phi(z)$ is the density of z under the standard Gaussian, then using the change of variables formula,

$$P_g(x) = \phi\left(g^{-1}(x)\right) \left|\det(J_x\left(g^{-1}(x)\right)\right|$$

Denoting $g^{-1} = f_\theta$ for some family of parametric function $\{f_\theta, \theta \in \Theta\}$, we can write the likelihood which is solved by the max-likelihood estimator given by,

$$\max_\theta \sum_{i=1}^{n} \log \phi(f_\theta(x_i)) + \log \left|\det(J_x\left(f_\theta(x_i)\right)\right|$$

If $f_\theta = f_1 \circ f_2 \circ \dots f_L$, then using the fact that the change of variables formula composes,

$$\max_\theta \sum_{i=1}^{n} \log \phi(f_\theta(x_i)) + \sum_{k=1}^{L} \log \left|\det(J_x\left(f_k(h_k(x_1))\right)\right|$$

where h_k is the value of the k^{th} layer.

If we can evaluate and differentiate the above objective efficiently, we can do gradient-based likelihood fitting. Thus, we want to desgin a "simple" family of invertible transforms.

1. **General linear maps**:
   Poor representational power since composition of linear maps is linear and if $x = Az$ where z is sampled from a Gaussian, x is Gaussian too. Also is computationally infeasible since evaluating determinant of a $d \times d$ matrix takes $O(d^3)$ time.
2. **Element-wise (non-linear) maps**:
   Suppose $f_\theta(x) = (f_\theta(x_1), f_\theta(x_2), \dots, f_\theta(x_d))$. In this case the Jacobian is diagonal since $\frac{\partial f_\theta(x_i)}{\partial x_j} = 0$ for $i \neq j$ so the determinant is simply product of diagonal elements, $\det(J_x\left(f_\theta(x)\right) = \prod_i \frac{\partial f_\theta(x_i)}{\partial x_i}$.
   But, these functions have poor representational power since they don't "combine" coordinates.

Note that even if a matrix is triangular, the determinant is just the product of the diagonals. This is the motivation behind the next method.

18.1.2.1 NICE (Non-linear Independent Component Estimation)

Divide the coordinates of x into sub-vectors: $x_{1:\frac{d}{2}}, x_{\frac{d}{2}+1:d}$.
Divide the coordinates of $z = f_\theta(x)$ into sub-vectors: $z_{1:\frac{d}{2}}, z_{\frac{d}{2}+1:d}$. Set

$$z_{1:\frac{d}{2}} = x_{1:\frac{d}{2}} \quad \text{and} \quad z_{\frac{d}{2}+1:d} = x_{\frac{d}{2}+1:d} \exp\left(s_\theta\left(x_{1:\frac{d}{2}}\right)\right) + t_\theta\left(x_{1:\frac{d}{2}}\right)$$

where s_θ, t_θ are arbitrary. The Jacobian of this transform is,

$$J_x(f_\theta(x)) = \begin{bmatrix} I & 0 \\ \frac{\partial z_{d/2+1:d}}{\partial x_{1:d/2}} & \text{diag}\left[\exp\left(s_\theta\left(x_{1:d/2}\right)\right)\right] \end{bmatrix}$$

Since the Jacobian is a triangular matrix, its determinant is the product of its diagonal elements and is thus easy to compute.

$$\det(J_x\left(f_\theta(x)\right) = \prod_i \exp\left(s_\theta\left(x_{1:d/2}\right)\right)$$

If s_θ is a neural network, then it is also easy to take derivatives.

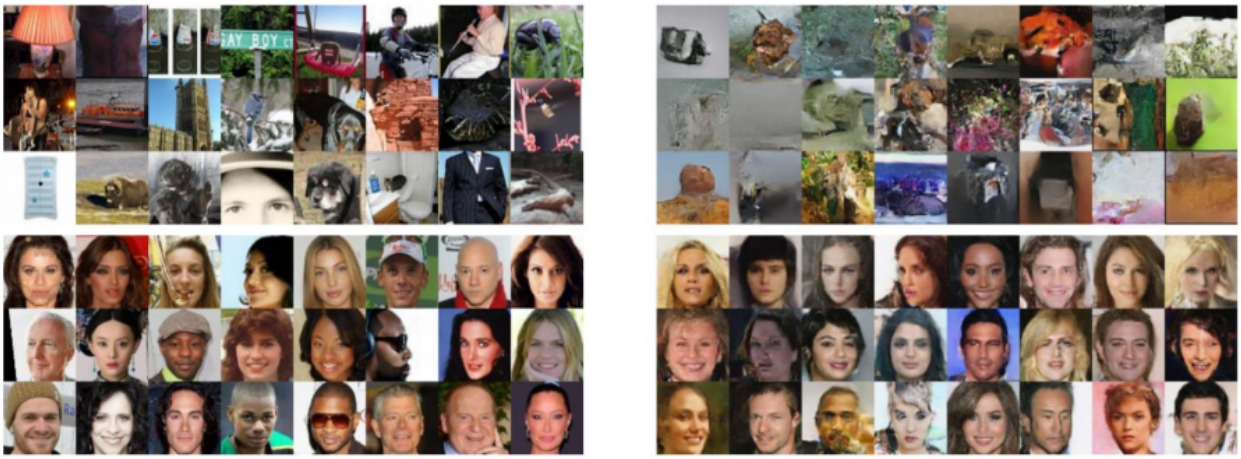

Figure 18.1: Figure from "Density estimation using Real NVP" [NVP16]

18.2 Autoregressive models

18.2.1 Sequential structure in data

Often times, data we are interested has "sequential" structure:

- *Word* in sentence depends on surrounding words.
- *Sounds* in speech depends on surrounding sounds.
- *Pixel* in image depends on surrounding pixels.

It make sense to factor joint distribution of data as

$$p(x_1, x_2, \ldots, x_t) = p(x_i | x_{<i})$$

We will model $p(x_i|x_{<i})$ such that we can sample from or learn the model efficiently.
Text has a natrual order. For images, usually "raster ordering" is used.

18.2.2 Fully visible sigmoid belief network

Assume data is binary (e.g. MNIST). One natural parametrization of $p(x_i|x_{<i})$ is,

$$p(x_i|x_{<i}) = \sigma\left(\sum_{j=1}^{i} \theta_j^i x_j\right)$$

To do sampling, we can do $\hat{x}_1 \sim p(x_1)$ and $\hat{x}_i \sim p(x_i|x_{<i})$ for $i = 2, \ldots, n$. This model is also easy to train since we know log-likelihood and gradients for sigmoid networks.

18.2.3 NADE: Neural Autoregressive Density Estimation

Assume data is binary (e.g. MNIST). The "slightly more neural" parametrization for $p(x_i|x_{<i})$

$$h_i = \sigma(A_i x_{<i} + c_i)$$
$$p(x_i|x_{<i}) = \sigma(\alpha_i^T h_i + b_i)$$

Where A_i is a one-hidden layer net with sigmoid activation σ. The weight-tied variants:

$$h_i = \sigma(W_{.,<i} x_{<i} + c_i)$$
$$p(x_i|x_{<i}) = \sigma(\alpha_i^T h_i + b_i)$$

18.2.4 MADE: Masked Autoencoder for Distribution Estimation

Strategy: Turn an autoencoder into a auto-regressive distribution modeler.
Recall, an autoencoder can be used to specify $p(\hat{x}|x)$ via the reconstruction loss. Suppose for some ordering of the input coordinates, $\hat{x}_1$ is independent of x, $\hat{x}_2$ only depends on x_1, $\dots \hat{x}_2$ only on $x_{<t}$–we can sample autoregressively.
Solution: use masks to ensure this. Choose an ordering, and only allow weights that respect ordering.
To zero out connections: mas via entrywise multiplication, e.g. for one layer:

$$h(x) = g(b + (W \bigodot M^W)x)$$
$$\hat{x} = \text{sigm}(c + (V^V)h(x))$$

To construct appropriate masks: for each node k, pick index $m(k)$ uniformly at random in $[1, D-1]$: denoting which inputs node depends on

$$M^W_{k,d} = 1_{m(k) \geq d} = \begin{cases} 1 & \text{if} m(k) \geq d \\ 0 & \text{otherwise,} \end{cases}$$

$$M^V_{d,k} = 1_{d > m(k)} = \begin{cases} 1 & \text{if} d > m(k) \\ 0 & \text{otherwise,} \end{cases}$$

18.2.5 PixelCNN

PixelCNN[PixelCNN16] has a convolutional architecture, and it is suitable for more complex image domains. Channels are generated auto-regressively. $p(x_i|x_{<i})$ is factorized as

$$p(x_i|x_{<i}) = P(x_{i,R}|X_{<i})P(x_{i,G}|X_{<i}, x_{i,R})P(x_{i,B}|X_{<i}, x_{i,R}, x_{i,G})$$

In upper layers, we use Mask B, in which value of channel can depend on value of same channel below.

18.3 Recurrent neural networks

Recurrent neural networks (RNN) are a way to *weight-tie* the parameters of an autoregressive models, s.t. it can be extended to arbitrary length sequences.

$$h_i = \tanh(W_{hh} h_{i-1} + W_{xh} x_i)$$
$$o_i = W_{hy} h_i$$

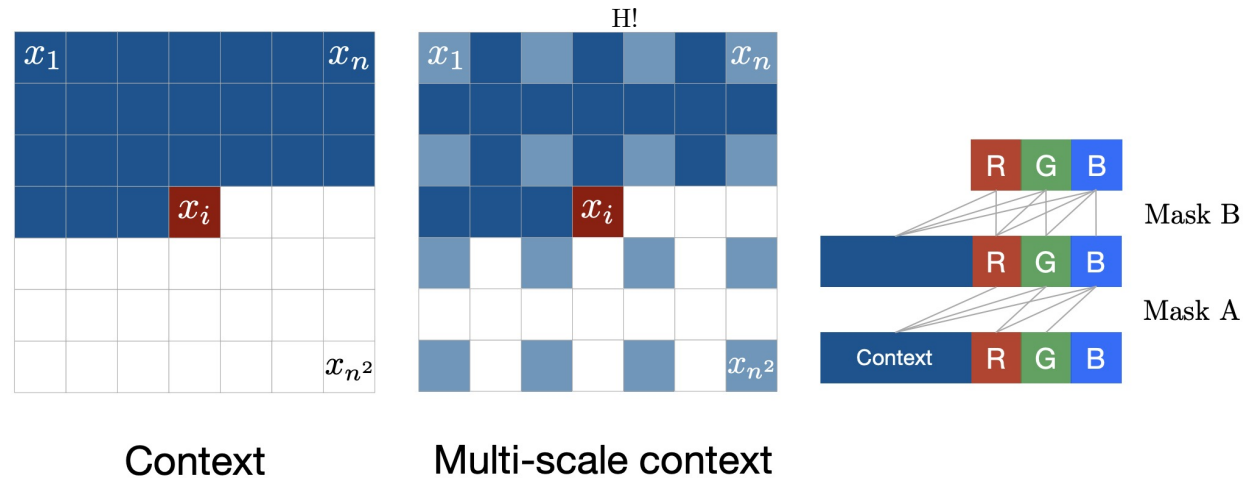

Figure 18.2: **Left:** To generate pixel x_i one conditions on all all the previously generated pixels left and above of x_i. **Center:** To generate a pixel in the multi-scale case we can also condition on the subsampled image pixels(in light blue). **Right:** Diagram of the connectivity inside a masked convolution. In the first layer, each of the RGB channels is connected to previous channels and to the context, but is not connected to itself. In subsequent layers, the channels are also connected to themselves.

where o_i specifies parameters for $p(x_i|x_{<i})$, e.g. softmax(y_i).

1. **Obvious benefit of using recurrent neural networks**:
   Copious weight tieing, the number of parameters is completely independent of length.

2. **Draw backs**:
   The training of recurrent neural networks is done by backpropagation. It is achieved by unfolding equivalence above, same as calculating derivative of a length-t feedforward network. There exists the prodlem of *gradient explosion* and *vanishing*.
   There is no parallelization, which means the *likelihood evaluation* has to be done sequentially.

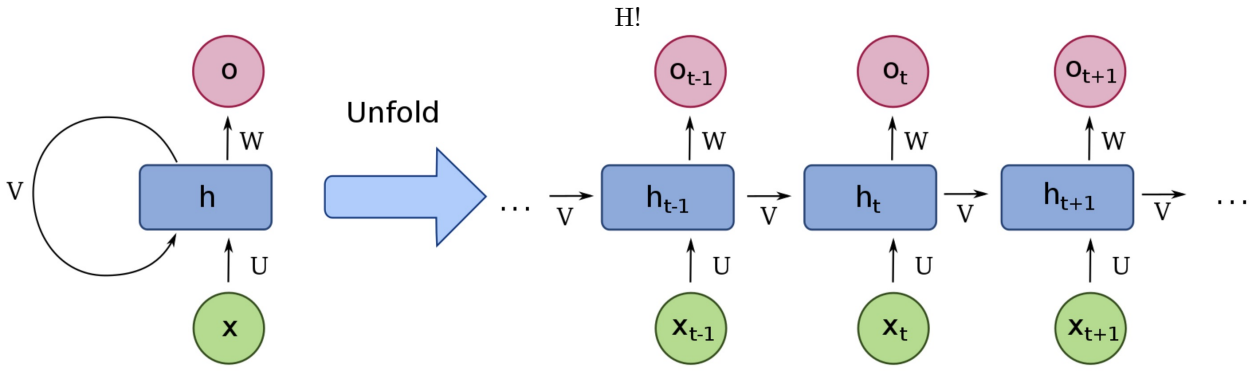

Figure 18.3: An illustration of the structure of recurrent neural networks

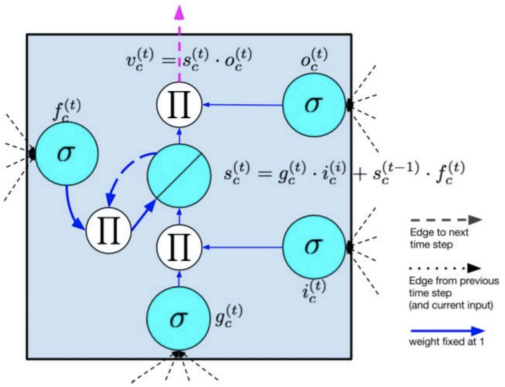

Figure 18.4: An illustration of the structure of LSTM

18.3.1 LSTM (Long Short-Term Memory recurrent neural network)

The main issue with training RNN's is long-term dependencies and correspondingly exploding/vanishing gradients. The main idea of **LSTM**[LSTM97] are gating mechanisms that try to control the flow of information from past. In practice, they seem to suffer much less from a gradient vanishing.
The core concept of LSTM's are the cell state, and it's various gates. The cell state transfers relative information all the way down the sequence chain, as the "memory" of the network. As the cell state goes on its journey, information gets added or removed to the cell via gates.

- **Gates**: The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.
- **Forget gate**: This gate decides what information should be thrown away or kept.
- **Input gate**: In the input gate operation, the previous hidden state and current input are passed into a sigmoid function. That will decide which values will be updated by whether it is important.
- **Cell state**: The cell state gets pointwise multiplied by the forget vector, then the output from the input gate will be added so as to update the cell state.
- **Output gate**: The output gate decides what the next hidden state should be.

The ingredients of the LSTM:

- **Input node**: $g^{(t)} = \Phi(W^{gx}x^{(t)} + W^{gh}h^{(t-1)} + b_g)$
- **Input gate**: $i^{(t)} = \sigma(W^{ix}x^{(t)} + W^{ih}h^{(t-1)} + b_i)$
- **Forget gate**: $f^{(t)} = \sigma(W^{fx}x^{(t)} + W^{fh}h^{(t-1)} + b_f)$
- **Output gate**: $o^{(t)} = \sigma(W^{ox}x^{(t)} + W^{oh}h^{(t-1)} + b_o)$
- **Internal state**: $s^{(t)} = g^{(t)} \bigodot i^{(i)} + s^{(t-1)} \bigodot f^{(t)}$
- **Hidden state**: $h^{(t)} = \phi(s^{(t)}) \bigodot o^{(t)}$

References

[GAN14] I. Goodfellow, J. Pouget-Abadie et al., "Generative adversarial networks," *NIPS*, 2014.

[NVP16] L. Dinh, J. Sohl-Dickstein and S. Bengio, "Density estimation using real NVP," *arXiv preprint arXiv:1605.08803*, 2016.

[PixelCNN16] A. Oord, N. Kalchbrenner et al., "Pixel Recurrent Neural Netorks" , 2014.

[LSTM97] S. Hochreiter, J. Schmidhuber et al., "Long short-term memory" *Neural computation*, 2014.