Masked Autoencoders for Distribution Estimation

Prof. Suyash Awate

Shikhar Agrawal
Mayank Jain

Why not AEs again?

- Traditional autoencoders are great because they can perform unsupervised learning by mapping an input to a latent representation. However, one drawback is that they don't have a solid probabilistic basis
- By using what the authors define as the *autoregressive property*, we can transform the traditional autoencoder approach into a fully probabilistic model with very little modification
- For vanilla autoencoders, we started with some neural network and then tried to apply some sort of probabilistic interpretation that didn't quite work out. Think the other way around: start with a probabilistic model and then figure out how to use neural networks to help you add more capacity and scale it

So what do we want from the AEs?

$$p(\mathbf{x}) = \prod_{i=1}^{D} p(x_i | \mathbf{x}_{<i})$$

where $\mathbf{x}_{<i} = [x_1, \ldots, x_{i-1}]$. Basically, component i of $\mathbf{x}$ only depends on the dimensions of $j < i$.

So how does this help us? In vanilla autoencoders, each output $\hat{x_i}$ could depend on any of the components input $x_1, \ldots, x_n$, as we saw before, this resulted in an improper probability distribution. If we start with the product rule, which guarantees a proper distribution, we can work backwards to map the autoencoder to this model.

For example, let's consider binary data (say a binarized image). $\hat{x_1}$ does not depend on any other components of $\mathbf{x}$, therefore our implementation should just need to estimate a single parameter p_1 for this pixel. How about $\hat{x_2}$ though? Now we let $\hat{x_2}$ depend *only* on x_1 since we have $p(x_2|x_1)$. This dependency can be modelled using a non-linear function... say maybe a neural network? So we'll have some neural net that maps x_1 to the $\hat{x_2}$ output. Now consider the general case of $\hat{x_j}$, we can have a neural net that maps $\mathbf{x}_{<\mathbf{j}}$ to the $\hat{x_j}$ output.

Conclusion

Lastly, there's no reason that each step needs to be a separate neural network, we can just put it all together in a single shared neural network so long as we follow a couple of rules:

1. Each output of the network $\hat{x}_i$ represents the probability distribution $p(x_i|\mathbf{x}_{<\mathbf{i}})$.
2. Each output $\hat{x}_i$ can only have connections (recursively) to smaller indexed inputs $\mathbf{x}_{<\mathbf{i}}$ and not any of the other ones.

Said another way, our neural net first learns P(x_1) (just a single parameter value in the case of binarized data), then iteratively learns the function mapping from X(<j) to x_j. In this view of the autoencoder, we are sequentially predicting (i.e. regressing) each dimension of the data using its previous values, hence this is called the *autoregressive* property of autoencoders.

Masks and the Autoregressive Network Structure

- The autoregressive autoencoder is referred to as a "Masked Autoencoder for Distribution Estimation", or MADE. "Masked" as we shall see below and "Distribution Estimation" because we now have a fully probabilistic model.
- "Autoencoder" now is a bit looser because we don't really have a concept of encoder and decoder anymore, only the fact that the same data is put on the input/output.
- From the autoregressive property, all we want to do is ensure that we only have connections (recursively) from inputs i to output j where i<j One way to accomplish this is to not make the unwanted connections in the first place, but that's a bit annoying because we can't easily use our existing infrastructure for neural networks.

Masks and the Autoregressive Network Structure

- The main observation here is that a connection with weight zero is the same as no connection at all. So all we have to do is zero-out the weights we don't want. We can do that easily with a "mask" for each weight matrix which says which connections we want and which we don't.

This is a simple modification to our standard neural networks. Consider a one hidden layer autoencoder with input x:

$$\mathbf{h}(\mathbf{x}) = \mathbf{g}(\mathbf{b} + (\mathbf{W} \odot \mathbf{M}^{\mathbf{W}})\mathbf{x})$$
$$\hat{\mathbf{x}} = \text{sigm}(\mathbf{c} + (\mathbf{V} \odot \mathbf{M}^{\mathbf{V}})\mathbf{h}(\mathbf{x})) \quad (9)$$

where:

- $\odot$ is an element wise product
- $\mathbf{x}, \hat{\mathbf{x}}$ is our vectors of input/output respectively
- $\mathbf{h}(\mathbf{x})$ is the hidden layer
- $\mathbf{g}(\cdot)$ is the activation function of the hidden layer
- $\text{sigm}(\cdot)$ is the sigmoid activation function of the output layer
- $\mathbf{b}, \mathbf{c}$ are the constant biases for the hidden/output layer respectively
- $\mathbf{W}, \mathbf{V}$ are the weight matrices for the hidden/output layer respectively
- $\mathbf{M}^{\mathbf{W}}, \mathbf{M}^{\mathbf{V}}$ are the weight mask matrices for the hidden/output layer respectively

Mask Matrices

- So long as our masks are set such that the autoregressive property is satisfied, the network can produce a proper probability distribution. One subtlety here is that for each hidden unit, we need to define an index that says which inputs it can be connected to (which also determines which index/output in the next layer it can be connected to). We'll use the notation in the paper of ml(k) to denote the index assigned to hidden node k in layer l. Our general rule for our masks is then:

$$M^{W^l}_{k',k} = \begin{cases} 1 \text{ if } m^l(k') \geq m^{l-1}(k) \\ 0 \text{ otherwise} \end{cases} \tag{10}$$

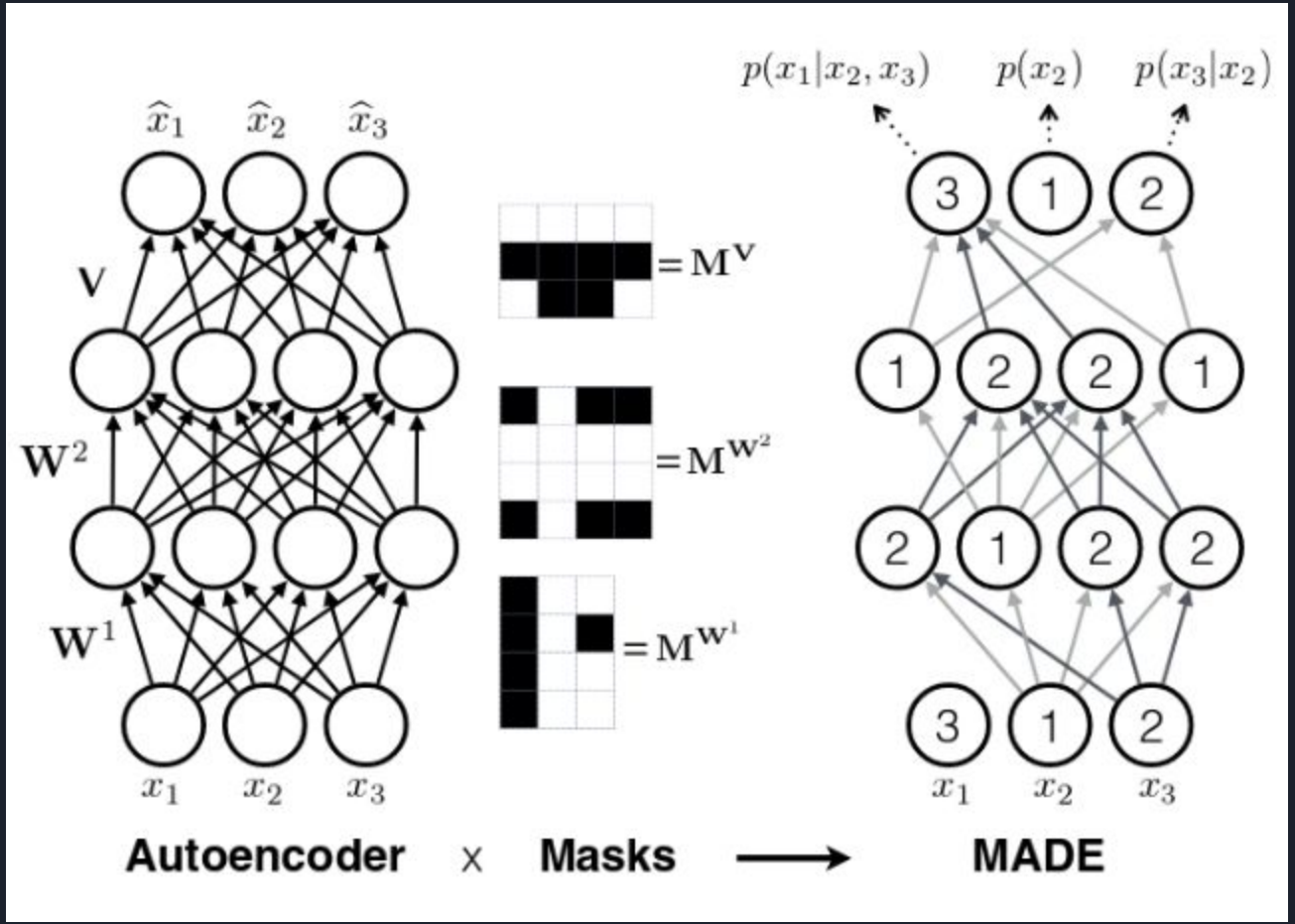
Basically, for a given node, only connect it to nodes in the previous layer that have an index less than or equal to its index. This will guarantee that a given index will recursively obey our auto-regressive property.

The output mask has a slightly different rule:

$$M^{V}_{d,k} = \begin{cases} 1 \text{ if } d > m^L(k) \\ 0 \text{ otherwise} \end{cases} \tag{11}$$

Masks and the Autoregressive Network Structure

- This is important because the first node should not depend on any other ones so it should not have any connections (will only have the bias connection), and the last node can have connections (recursively) to every other node except its respective input.
- Finally, one last topic to discuss is how to assign ml(k). It doesn't really matter too much as long as we have enough connections for each index. We do a natural thing and just sample from a uniform distribution with range [1,D-1]. Why only up to D-1? Recall, we should never assign index D because it will never be used so there's no use in connecting anything to D (nothing can ever depend on the Dth input).


$p(x_1|x_2,x_3)$
$p(x_2)$
$p(x_3|x_2)$
$\widehat{x}_1$
$\widehat{x}_2$
$\widehat{x}_3$
V
W^2
W^1
$= M^V$
$= M^{W^2}$
$= M^{W^1}$
3
1
2
1
2
2
1
2
1
2
2
2
3
1
2
x_1
x_2
x_3
Autoencoder
x
Masks
MADE

Improvisation, Ordering Inputs, Direct Connections

- The first is the ordering of the inputs. We've been talking about "Input 1, 2, 3, ..." but usually there is no natural ordering of the inputs. We can arbitrarily pick any ordering that we want just by shuffling m0, the selection layer for the input. This can even be performed at each mini-batch to get an "average" over many different models.

The last idea is just to add a direct connection path from input to output like so:

$$\hat{\mathbf{x}} = \text{sigm}\left(\mathbf{c} + (\mathbf{V} \odot \mathbf{M}^{\mathbf{V}})\mathbf{h}(\mathbf{x})\right) + \left(\mathbf{A} \odot \mathbf{M}^{\mathbf{A}}\right)\mathbf{x} \tag{12}$$

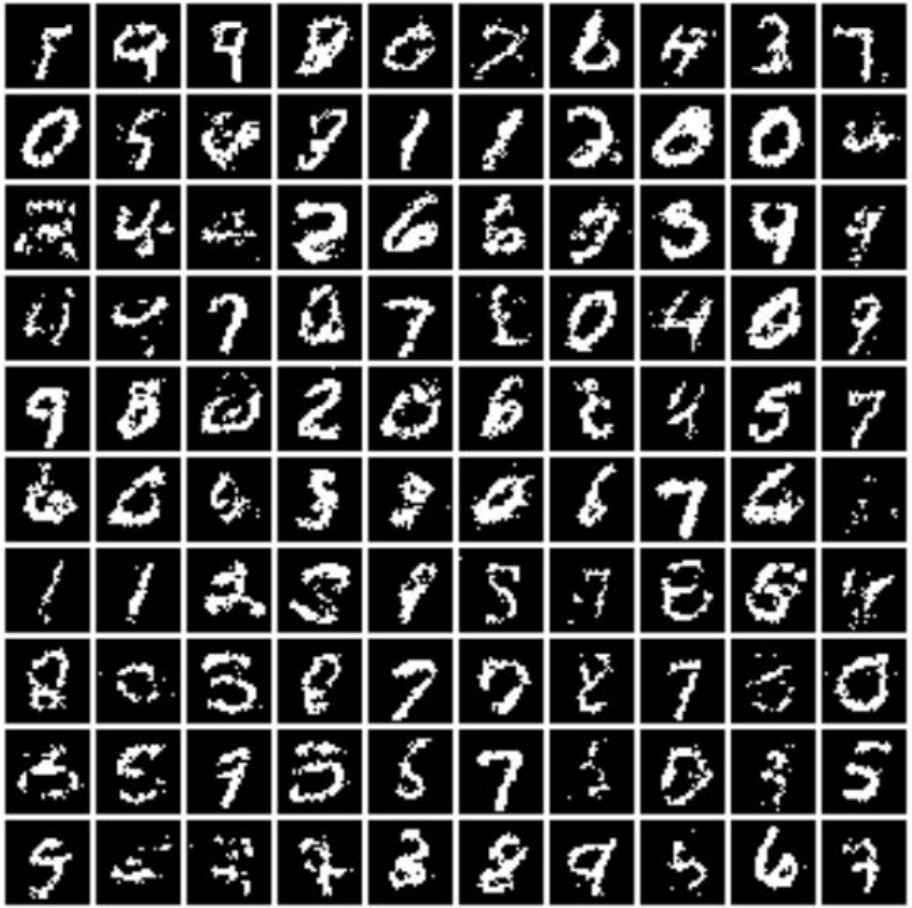
where $\mathbf{A}$ is the weight matrix that directly connects inputs to outputs, and $\mathbf{M}^{\mathbf{A}}$ is the corresponding mask matrix that follows the autoregressive property.

Generating New Samples

- The main idea for binary data is:

1. Randomly generate vector $\mathbf{x}$, set $i = 1$.
2. Feed $\mathbf{x}$ into autoencoder and generate outputs $\hat{\mathbf{x}}$ for the network, set $p = \hat{x}_i$.
3. Sample from a Bernoulli distribution with parameter p, set input $x_i = \text{Bernoulli}(p)$.
4. Increment i and repeat steps 2-4 until $i > D$.

Basically, we're iteratively calculating p(xi|x<i) by doing a forward pass on the autoencoder each time. Along the way, we sample from the Bernoulli distribution and feed the sampled value back into the autoencoder to compute the next parameter for the next bit. It's a bit inefficient but MADE is also a relatively small modification to the vanilla autoencoder so you can't ask for too much.



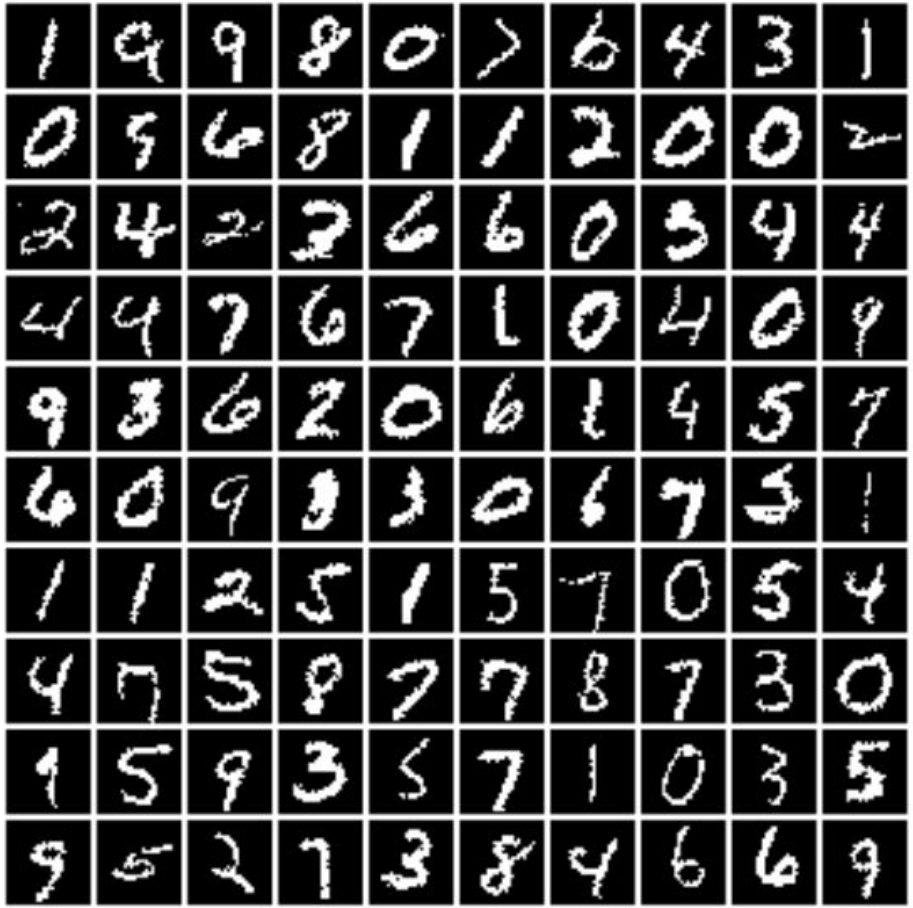


Figure 3. Left: Samples from a 2 hidden layer MADE. Right: Nearest neighbour in binarized MNIST.