

# Encoder-Decoder Models

Muchang Bahng

Spring 2023

## Contents

|          |                                |          |
|----------|--------------------------------|----------|
| <b>1</b> | <b>Encoder-Decoder Models</b>  | <b>2</b> |
| 1.1      | Autoencoders . . . . .         | 2        |
| 1.2      | Sequence to Sequence . . . . . | 5        |
| 1.3      | More Flexible Models . . . . . | 7        |
| <b>2</b> | <b>Autoregressive Models</b>   | <b>7</b> |
| 2.1      | NADE . . . . .                 | 7        |
| 2.2      | PixelRNN . . . . .             | 7        |
| 2.3      | PixelCNN . . . . .             | 7        |
|          | <b>References</b>              | <b>7</b> |

# 1 Encoder-Decoder Models

Encoder decoder models refer to a model consisting of two neural nets: the encoder that takes in the input and maps it to some lower-dimensional vector. Then, the decoder takes in this encoded vector and attempts to use it to decode what we're trying to get. The type of neural network can be any: MLP, CNN, or RNN, depending on what problem you're trying to achieve.

Now, why would we want to do something like encode the input into some lower dimensional setting, and then have the decoder neural net extract what we want? It seems like we're making the problem harder. There are two reasons:

1. The input vector may not be in the correct form that we want. This is the motivation for the *seq2seq* model, where we are working with sequences of vectors that suffer from the problem of *locality* in RNNs. Therefore, it is necessary to encode this entire sequence into one vector, at the loss of dimension.
2. The input vector may be noisy or too high-dimensional itself. In CNNs, we saw that convolutional layers or pooling layers allow us to reduce the dimension to extract meaningful features from it. Likewise, we can train the encoder to extract useful features into a lower dimensional space, and then the decoder can efficiently work with this representation. This motivates the use of **autoencoders**, which can be done with MLPs, CNNs, or even RNNs.

Note that while these two algorithms fall in the paradigm of encoder-decoder networks, the *seq2seq* model is supervised while the autoencoder is unsupervised. In the *seq2seq* model, which deals with things like machine translation, we have a labeled dataset of sentences in language A corresponding with sentences in language B. However, in autoencoders, what we do is take a sample  $x$  from our dataset and use it both as the input and output to train our network. Since there is no additional labeling required, this is an unsupervised learning technique.

## 1.1 Autoencoders

In the totally linear case, we have PCA. Some input  $\mathbf{x} \in \mathcal{X} = \mathbb{R}^d$  is mapped to a smaller-dimensional  $\mathcal{Z} = \mathbb{R}^k$ .

$$\mathbf{x} \xrightarrow{V} \mathbf{z} \xrightarrow{U} \hat{\mathbf{x}}$$

and so the "network" essentially computes  $\hat{\mathbf{x}} = UV\mathbf{x}$ . Obviously the fact that  $k < d$  is essential, since if  $k \geq d$  then we can choose  $U$  and  $V$  such that  $UV = I$ , which is trivial.

This can be used for the following problem: Given  $m$  points  $\mathbf{x}_1, \dots, \mathbf{x}_m \in \mathbb{R}^d$  and target dimension  $k < d$ , find the best  $k$ -dimensional subspace approximating the data. Formally, we want to find the matrices  $U \in \mathbb{R}^{d \times k}$  and  $V \in \mathbb{R}^{k \times d}$  that minimizes

$$f(U, V) = \sum_{i=1}^m \|\mathbf{x}_i - UV\mathbf{x}_i\|_2^2$$

where  $V$  is the *compressor* and  $U$  is the *decompressor*. Now unfortunately, this loss  $f$  is not convex, though  $f(U, \cdot)$  and  $f(\cdot, V)$  are both convex.

### Theorem 1.1 ()

We claim that the optimal solution is achieved when  $U = V^T$  and  $U^T U = I$ .

### Proof.

For any  $U, V$ , the linear map  $\mathbf{x} \mapsto UV\mathbf{x}$  has a range  $R$  that forms a subspace of dimension  $k$ . Let  $w_1, \dots, w_k$  be an orthonormal basis for  $R$ , which we arrange into columns of  $W$ . Hence, for each  $x_i$  there is  $z_i \in \mathbb{R}^k$  such that  $UVx_i = Wz_i$ . Note that by construction,  $W^T W = I$ . Now we want to find out which  $z$  minimizes  $f(x_i, z) = \|x_i - Wz\|_2^2$ . We know that for all  $x \in \mathbb{R}^d, z \in \mathbb{R}^k$ ,

$$f(x, z) = \|x\|_2^2 + z^T W^T W z - 2z^T W^T x = \|x\|_2^2 + \|z\|_2^2 - 2z^T W^T x$$

We want to minimize w.r.t. to  $z$ , so by taking the derivative and setting to 0, we get  $z = W^T x$ . This means that

$$\sum_{i=1}^m \|x_i - UVx_i\|^2 \geq \sum_{i=1}^m \|x_i - UVx_i\|^2 \geq \sum_{i=1}^m \|x_i - WW^T x_i\|^2$$

and since  $U, V$  are optimal, equality is achieved and so instead of  $U, V$ , we can take  $W, W^T$ , with  $WW^T x$  being the orthogonal projection of  $x$  onto  $R$ .

One application of PCA is eigenfaces, which assumes that the set of all faces (projected onto an image) approximately lies in a hyperplane.

Autoencoders are a nonlinear generalization of PCA. We only use the inputs  $\mathbf{x}_t$  for learning. We want to automatically extract meaningful features for the data and leverage the availability of unlabeled data. It can be used for visualization and compression. We can also build generative models with autoencoders. We can have several architectures, with none, one, or both the encoder/decoder having nonlinear activation functions. Here is one architecture.

#### Definition 1.1 (Autoencoder)

An autoencoder is a feed-forward neural net whose job is to take an input  $\mathbf{x}$  and output  $\hat{\mathbf{x}}$ . It consists of an encoder  $E_\phi : \mathcal{X} \rightarrow \mathcal{Z}$  and decoder  $D_\theta : \mathcal{Z} \rightarrow \mathcal{X}$ , where  $\mathcal{X}$  is the input/output space and  $\mathcal{Z}$  is the latent feature space.

1. The encoder model transforms  $\mathbf{x}$  to a latent feature representation  $\mathbf{z}$ . It is a feed-forward, bottom-up neural net.
2. The decoder model maps  $\mathbf{z}$  to a reconstruction  $\hat{\mathbf{x}}$ . It is generative, top-down.

where we have

$$\text{Encoder : } \mathbf{h}(\mathbf{x}) = g(\mathbf{a}(x)) = \sigma(\mathbf{b} + \mathbf{W}\mathbf{x})$$

$$\text{Decoder : } \hat{\mathbf{a}}(\mathbf{x}) = \sigma(\mathbf{c} + \mathbf{W}^* \mathbf{h}(\mathbf{x}))$$

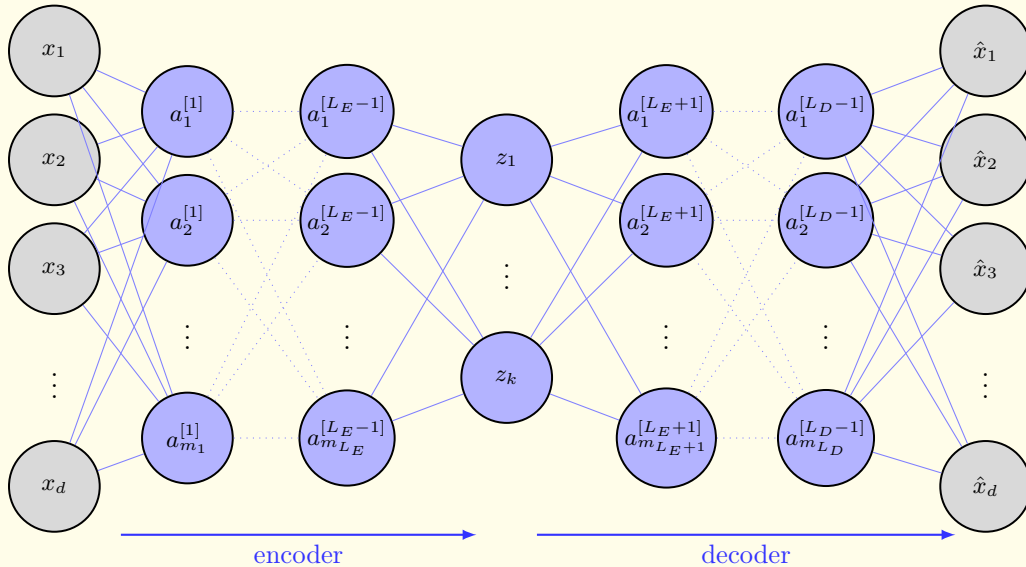


Figure 1: A graphical model of an autoencoder. The  $z$  is called the **bottleneck layer**, which compresses the input into a lower-dimensional representation. The  $x$  is known as the **input layer**, the  $\hat{x}$  as the **output layer**, and everything else are **hidden layers**.

The parameter gradients are obtained by backpropagating the gradient  $\nabla_{\theta} \mathcal{L}$  like a regular network, but if we force tied weights (i.e.  $W^* = W^T$ ), then  $\nabla_{\mathbf{W}} \mathcal{L}$  is the sum of two gradients. This is because  $\mathbf{W}$  is present both in the encoder and decoder.

I want to train the whole neural network such that the error between  $\mathbf{x}$  and  $\hat{\mathbf{x}}$  is minimized. We can consider a squared-error, for example.

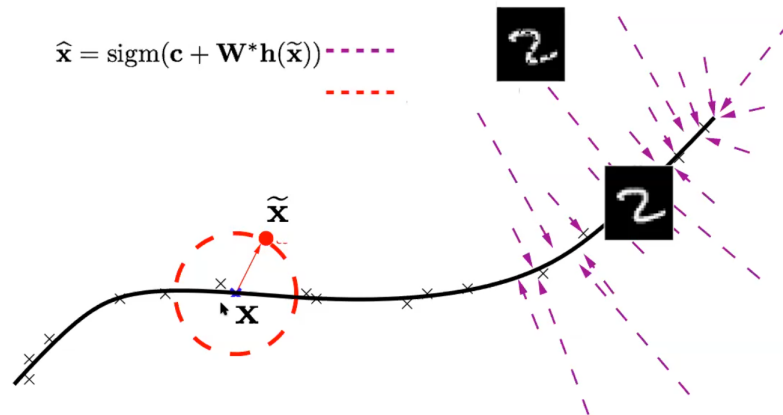
$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{2} \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2 \quad (1)$$

#### Algorithm 1.1 ()

An implementation of an autoencoder in PyTorch is here.

There are three things we can do to extract meaningful hidden features:

1. **Undercomplete Representation:** Make the latent dimension small. It compresses the input, but it may only be good for the training distribution and may not be robust to other types of input. If it is overcomplete, there is no guarantee that we will extract meaningful features.
2. **Denoising Autoencoder:** Injecting noise to the input. The idea is that the representation should be robust to the introduction of noise. We take the original input  $\mathbf{x}$  and we randomly assign a subset of the inputs to 0, with probability  $\nu$ , similar to dropout, to get our noisy input  $\tilde{\mathbf{x}}$ . Then we train the autoencoder with the loss comparing the output  $\hat{\mathbf{x}}$  to the original, un-noisy input  $\mathbf{x}$ . We can do this for Gaussian additive noise too. As the visual below suggests, we are essentially “pushing” out inputs away from the manifold and training the autoencoder to denoise it, pulling it back.



3. **Contractive Autoencoder:** If we have the latent dimension greater than the input, then we can just add an explicit term in the loss that penalizes that solution (e.g. promoting sparsity). For example, we can have the loss be

$$\mathcal{L}(f(\mathbf{x}^{(t)}) + \lambda \|\nabla_{\mathbf{x}^{(t)}} \mathbf{h}(\mathbf{x}^{(t)})\|_F^2$$

where

$$\|\nabla_{\mathbf{x}^{(t)}} \mathbf{h}(\mathbf{x}^{(t)})\|_F^2 = \sum_{j,k} \left( \frac{\partial h(\mathbf{x}^{(t)})_j}{\partial x_k^{(t)}} \right)^2$$

which forces the encoder to throw away information. If one of the elements are 0, then we know that the  $k$ th element of the input has no effect on the  $j$ th element of the encoded output. Therefore, it tries to throw away as many elements of  $\mathbf{x}$  as possible since the identity matrix will have a large Frobenius norm, essentially contracting the input representation.

We can also promote sparsity by adding a L1 penalty, forcing the feature space to be sparse.

The **predictive sparse decomposition** shows that the loss should be

$$\min_{W, W^*, \mathbf{z}} \|\mathbf{W}^* \mathbf{z} - \mathbf{x}\|_2^2 + \lambda \|\mathbf{z}\|_1 + \|\sigma(\mathbf{W} \mathbf{x}) - \mathbf{z}\|_2^2 \quad (2)$$

where the first term tells the decoder to reconstruct the original input well, the second tells the latent vector to be sparse, and the third tells us that we shouldn't lose too much information when we encode.

We could also have **stacked autoencoders**, with each layer of latent features having some desired sparsity.

## 1.2 Sequence to Sequence

We have mentioned that RNNs and LSTMs have the advantage of mapping from variable length inputs to variable length outputs. This can be done for any length input and any length output. However, the RNN has the problem of *locality*, that the words next to the current word have a greater effect, and we are trying to generate sequences on the fly by reading in each word. Even for bidirectional RNNs, where we go through the whole sentence first, the effects of adjacent words have a greater effect when generating outputs. It would be wiser to read the *whole* sentence and then start to generate a sequence. This is the motivation for the **encoder-decoder model**. It is conventionally divided into a two-stage network.

1. The encoder neural net would convert a sequence into a single latent space representation  $z = f(x)$ . This latent representation  $z$  essentially refers to a feature (vector) representation, which is able to capture the underlying semantic information of the input that is useful for predicting the output.
2. The decoder neural net would decode this feature vector, called the **context vector**, into a sequence of the desired output  $y = g(z)$  by using it as the initial hidden state. It uses the previous output as the next input for decoding.

Note that the encoder and decoder are two completely separate neural networks with their own parameters. This is important, since the fact that these are two completely separate networks allows us to work in different "paradigms" within either the feature or target space. For example, if we want to perform machine translation from English to Spanish, our encoder RNN parameters have been tuned to the English syntax and language, while the decoder RNN parameters are tuned to the Spanish language. Since we are modeling different languages, it makes sense to have different sequence models for each one.

We will talk about a specific type of encoder-decoder model called **seq2seq**, which maps sequences to sequences using RNN encoders and decoders. Conventionally, the hidden nodes of the encoder are denoted with  $\mathbf{h}$ , and those of the decoder are denoted with  $\mathbf{s}$ .

1. For the encoder, we take in the inputs  $\mathbf{x}_t$  and generate the hidden states as

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}) = \mathbf{W}_e \mathbf{h}_{t-1} + \mathbf{U}_e \mathbf{x}_t + \mathbf{b}_e \quad (3)$$

In general, the encoder transforms the hidden states at all time steps into a context variable through the composition of functions  $q$

$$\mathbf{C} = q(\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_T)$$

In the figure below, the context variable is just  $\mathbf{C} = \mathbf{h}_T$ .

2. Now, given the target output sequence  $\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_{t'+1}$  for each timestep  $t'$  (we use  $t'$  to differentiate from the input sequence time steps), the decoder assigns a predicted probability to each possible token occurring at step  $\hat{\mathbf{y}}_{t'+1}$  conditioned on both the previous tokens  $\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_{t'+1}$  and the context variable  $\mathbf{C}$ , i.e.

$$\mathbb{P}(\hat{\mathbf{y}}_{t'+1} \mid \hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_{t'+1}, \mathbf{C})$$

Therefore, to decode the subsequent token  $\hat{\mathbf{y}}_{t'+1}$ , we calculate the hidden state  $\mathbf{s}_{t'+1}$  as a gated hidden unit computed by

$$\mathbf{s}_{t'+1} = g(\mathbf{s}_{t'}, \hat{\mathbf{y}}_{t'}, \mathbf{C})$$

with the math mentioned here.

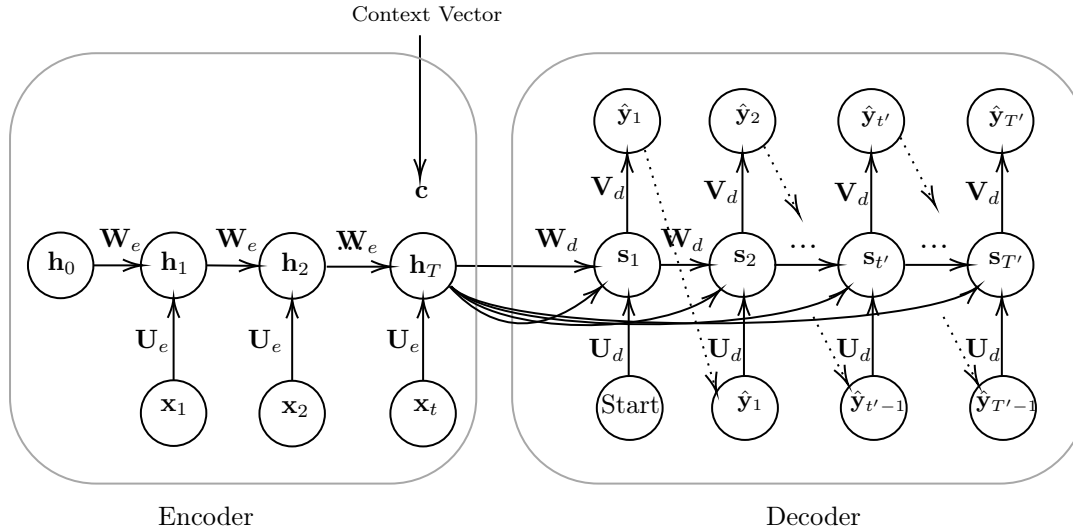


Figure 2

Again, note that this encoder-decoder model is comprised of two completely separate deep models with their own parameters, and so it is *not* simply just one long RNN that starts generating outputs only after it takes in all the inputs. Sometimes, the inputs to the decoder may not be shown in diagrams since it is assumed that they are always the previous node's outputs. Furthermore, we can also see that there is no clear-defined first input for the decoder model, since this is the beginning of the sequence. We usually just put some special "start" element in here to denote the beginning of the output.

Here is a diagram for a encoder-decoder model for a 2-layer LSTM which is the standard for practical use, which encodes the sentence meaning in the vectors  $\mathbf{c}_t^{[2]}, \mathbf{h}_t^{[2]}, \mathbf{c}_t^{[1]}, \mathbf{h}_t^{[1]}$ . In practice, high performing RNNs are usually multilayer (almost always greater than 1, but diminishing performance returns as number of layers increases), but are not as deep as convolutional or feed forward networks.

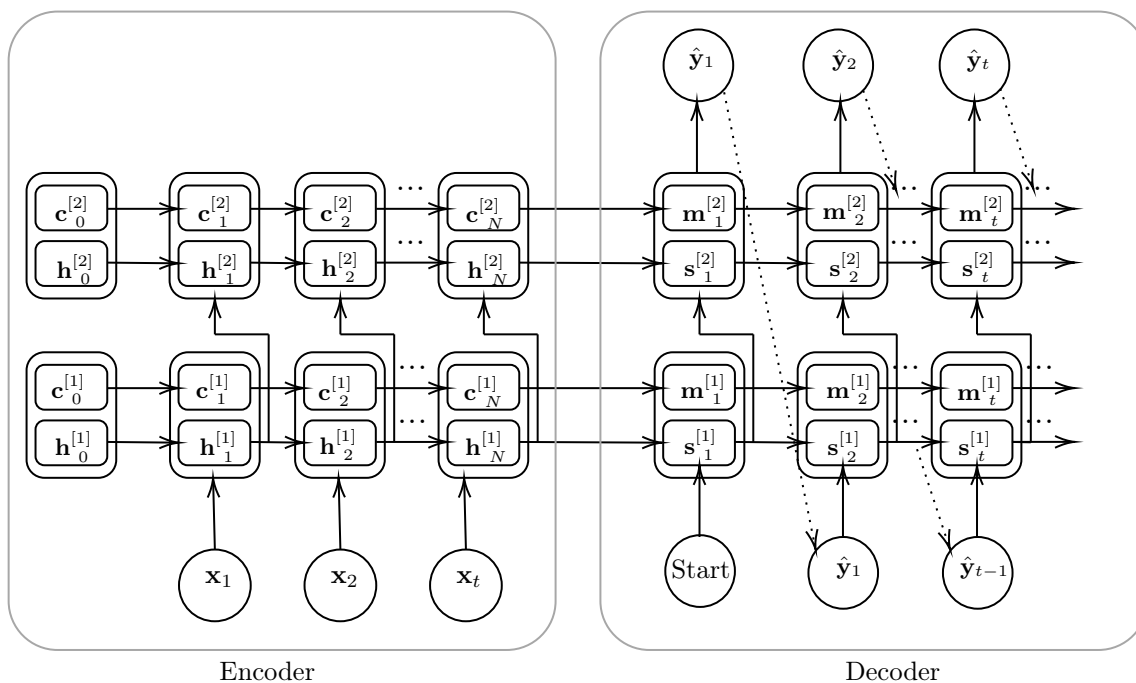


Figure 3

Again, to train this model, we do the same backpropagation algorithm on a normalized loss function with teacher forcing over a parallel dataset. What is nice about the encoder-decoder seq2seq is that it can be completely implemented end-to-end, so we can backpropagate through the entire decoder and encoder to train the both models simultaneously.

### 1.3 More Flexible Models

By combining these neural nets, we can essentially create image captioning (with a CNN encoder and RNN decoder) and image generation (RNN encoder and CNN decoder).

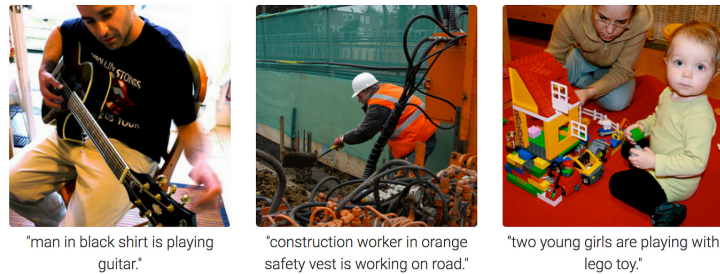


Figure 4: Image captioning on various image prompts.

## 2 Autoregressive Models

### 2.1 NADE

Introduced in 2011 as alternative to RBMs.

### 2.2 PixelRNN

### 2.3 PixelCNN

## References