Autoencoders, Extensions, and Applications

Piyush Rai

IIT Kanpur
Outline

- Introduction to Autoencoders
- Autoencoder Variants and Extensions
- Some Applications of Autoencoders
- Autoencoders for Recommender Systems
Autoencoder

- Similar to the standard feedforward neural network with a key difference:
  - **Unsupervised.** No “label” at the output layer; Output layer simply tries to “recreate” the input.
Autoencoder

- Similar to the standard feedforward neural network with a key difference:
  - **Unsupervised.** No “label” at the output layer; Output layer simply tries to “recreate” the input

- Defined by two (possibly nonlinear) mapping functions: Encoding function $f$, Decoding function $g$
Autoencoder

Similar to the standard feedforward neural network with a key difference:

- **Unsupervised.** No “label” at the output layer; Output layer simply tries to “recreate” the input

Defined by two (possibly nonlinear) mapping functions: **Encoding function** $f$, **Decoding function** $g$

$h = f(x)$ denotes an encoding (possibly nonlinear) for the input $x$
Autoencoder

- Similar to the standard feedforward neural network with a key difference:
  - **Unsupervised**. No “label” at the output layer; Output layer simply tries to “recreate” the input.

- Defined by two (possibly nonlinear) mapping functions: Encoding function $f$, Decoding function $g$
  - $h = f(x)$ denotes an encoding (possibly nonlinear) for the input $x$
  - $\hat{x} = g(h) = g(f(x))$ denotes the reconstruction (or the “decoding”) for the input $x$
Autoencoder

- Similar to the standard feedforward neural network with a key difference:
  - **Unsupervised.** No “label” at the output layer; Output layer simply tries to “recreate” the input

 Defined by two (possibly nonlinear) mapping functions: **Encoding function** $f$, **Decoding function** $g$

- $h = f(x)$ denotes an encoding (possibly nonlinear) for the input $x$
- $\hat{x} = g(h) = g(f(x))$ denotes the reconstruction (or the “decoding”) for the input $x$
- For an Autoencoder, $f$ and $g$ are learned with a goal to **minimize the difference between $\hat{x}$ and $x$**
The learned code $h = f(x)$ can be used as a new feature representation of the input $x$.

Therefore autoencoders can also be used for “feature learning”.
The learned code $h = f(x)$ can be used as a new feature representation of the input $x$.

Therefore autoencoders can also be used for "feature learning".

Note: Size of the hidden units (encoding) can also be larger than the input.
A Simple Autoencoder

- Let’s assume a $D \times 1$ input $\mathbf{x} \in \mathbb{R}^D$, and a single hidden layer with $K \times 1$ code $\mathbf{h} \in \mathbb{R}^K$

- We can then define a simple linear autoencoder as

$$
\begin{align*}
\mathbf{h} &= f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \\
\hat{\mathbf{x}} &= g(h) = \mathbf{W}^\top\mathbf{h} + \mathbf{c}
\end{align*}
$$

where $f$ is defined by $\mathbf{W} \in \mathbb{R}^{K \times D}$ and $\mathbf{b} \in \mathbb{R}^{K \times 1}$
Let’s assume a $D \times 1$ input $x \in \mathbb{R}^D$, and a single hidden layer with $K \times 1$ code $h \in \mathbb{R}^K$.

We can then define a simple linear autoencoder as

$$
h = f(x) = Wx + b \quad \hat{x} = g(h) = W^*h + c
$$

where $f$ is defined by $W \in \mathbb{R}^{K \times D}$ and $b \in \mathbb{R}^{K \times 1}$, $g$ is defined by $W^* \in \mathbb{R}^{D \times K}$ and $c \in \mathbb{R}^{D \times 1}$. Note: If we learn $f$, $g$ to minimize the squared error $||\hat{x} - x||^2$ then the linear autoencoder with $W^* = W^\top$ is optimal, and is equivalent to Principal Component Analysis (PCA).
A Simple Autoencoder

- Let’s assume a \( D \times 1 \) input \( x \in \mathbb{R}^D \), and a single hidden layer with \( K \times 1 \) code \( h \in \mathbb{R}^K \)
- We can then define a simple linear autoencoder as

\[
\begin{align*}
  h &= f(x) = Wx + b \\
  \hat{x} &= g(h) = W^*h + c
\end{align*}
\]

where \( f \) is defined by \( W \in \mathbb{R}^{K \times D} \) and \( b \in \mathbb{R}^{K \times 1} \), \( g \) is defined by \( W^* \in \mathbb{R}^{D \times K} \) and \( c \in \mathbb{R}^{D \times 1} \)

Note: If we learn \( f, g \) to minimize the squared error \( ||\hat{x} - x||^2 \) then the linear autoencoder with \( W^* = W^\top \) is optimal, and is equivalent to Principal Component Analysis (PCA).
A Simple Autoencoder

- Let's assume a $D \times 1$ input $x \in \mathbb{R}^D$, and a single hidden layer with $K \times 1$ code $h \in \mathbb{R}^K$

- We can then define a simple linear autoencoder as
  
  $$ h = f(x) = Wx + b $$
  $$ \hat{x} = g(h) = W^*h + c $$

  where $f$ is defined by $W \in \mathbb{R}^{K \times D}$ and $b \in \mathbb{R}^{K \times 1}$, $g$ is defined by $W^* \in \mathbb{R}^{D \times K}$ and $c \in \mathbb{R}^{D \times 1}$

- Note: If we learn $f, g$ to minimize the squared error $||\hat{x} - x||^2$ then the linear autoencoder with $W^* = W^T$ is optimal
A Simple Autoencoder

- Let’s assume a $D \times 1$ input $\mathbf{x} \in \mathbb{R}^D$, and a single hidden layer with $K \times 1$ code $\mathbf{h} \in \mathbb{R}^K$

- We can then define a simple linear autoencoder as

$$
\mathbf{h} = f(\mathbf{x}) = \mathbf{Wx} + \mathbf{b} \\
\hat{\mathbf{x}} = g(\mathbf{h}) = \mathbf{W}^\top \mathbf{h} + \mathbf{c}
$$

where $f$ is defined by $\mathbf{W} \in \mathbb{R}^{K \times D}$ and $\mathbf{b} \in \mathbb{R}^{K \times 1}$, $g$ is defined by $\mathbf{W}^\top \in \mathbb{R}^{D \times K}$ and $\mathbf{c} \in \mathbb{R}^{D \times 1}$

- Note: If we learn $f, g$ to minimize the squared error $||\hat{\mathbf{x}} - \mathbf{x}||^2$ then the linear autoencoder with $\mathbf{W}^\top = \mathbf{W}^\top$ is optimal, and is equivalent to Principal Component Analysis (PCA)
Autoencoder: Zooming in..

\[ W: K \times D \text{ matrix of weights of edges between input and hidden layer} \]

\[ W_{kd} \text{ is the weight of edge connecting input layer node } d \text{ to hidden layer node } k \]

\[ W^*: D \times K \text{ matrix of weights of edges between hidden and output layer} \]

\[ W_{dk} \text{ is the weight of edge connecting hidden layer node } k \text{ to output layer node } d \]

If \( W^* = W^\top \), the autoencoder architecture is said to have "tied weights."
Autoencoder: Zooming in..

- **W**: $K \times D$ matrix of weights of edges between input and hidden layer

$W^*$ is the weight of edge connecting input layer node $d$ to hidden layer node $k$

$W^*$: $D \times K$ matrix of weights of edges between hidden and output layer

$W_{dk}$ is the weight of edge connecting hidden layer node $k$ to output layer node $d$

If $W^* = W^\top$, the autoencoder architecture is said to have "tied weights"
Autoencoder: Zooming in..

- **W**: $K \times D$ matrix of weights of edges between input and hidden layer
  - $W_{kd}$ is the weight of edge connecting input layer node $d$ to hidden layer node $k$
Autoencoder: Zooming in..

- **W**: $K \times D$ matrix of weights of edges between input and hidden layer
  - $W_{kd}$ is the weight of edge connecting input layer node $d$ to hidden layer node $k$
- **W\textsuperscript{*}**: $D \times K$ matrix of weights of edges between hidden and output layer
\[ W \]: \( K \times D \) matrix of weights of edges between input and hidden layer

- \( W_{kd} \) is the weight of edge connecting input layer node \( d \) to hidden layer node \( k \)

\[ W^* \]: \( D \times K \) matrix of weights of edges between hidden and output layer

- \( W^*_{dk} \) is the weight of edge connecting hidden layer node \( k \) to output layer node \( d \)
Autoencoder: Zooming in..

- **$W$:** $K \times D$ matrix of weights of edges between input and hidden layer
  - $W_{kd}$ is the weight of edge connecting input layer node $d$ to hidden layer node $k$

- **$W^*$:** $D \times K$ matrix of weights of edges between hidden and output layer
  - $W_{dk}^*$ is the weight of edge connecting hidden layer node $k$ to output layer node $d$

- If $W^* = W^T$, the autoencoder architecture is said to have “tied weights”
The hidden nodes can also be **nonlinear transforms** of the inputs, e.g.,

- Can define $h$ as a linear transform of $x$ followed by a nonlinearity (e.g., sigmoid, ReLU)

$$h = \text{sigmoid}(Wx + b)$$

where the nonlinearity $\text{sigmoid}(z) = \frac{1}{1 + \exp(-z)}$ squashes the real-valued $z$ to lie between 0 and 1.
Nonlinear Autoencoders

The hidden nodes can also be **nonlinear transforms** of the inputs, e.g.,

- Can define $h$ as a linear transform of $x$ followed by a nonlinearity (e.g., sigmoid, ReLU)
  
  $h = \text{sigmoid}(Wx + b)$
  
  where the nonlinearity $\text{sigmoid}(z) = \frac{1}{1 + \exp(-z)}$ squashes the real-valued $z$ to lie between 0 and 1

- Most commonly used autoencoders use such nonlinear transforms
Nonlinear Autoencoders

The hidden nodes can also be nonlinear transforms of the inputs, e.g.,

- Can define $h$ as a linear transform of $x$ followed by a nonlinearity (e.g., sigmoid, ReLU)

$$h = \text{sigmoid}(Wx + b)$$

where the nonlinearity $\text{sigmoid}(z) = \frac{1}{1+\exp(-z)}$ squashes the real-valued $z$ to lie between 0 and 1

- Most commonly used autoencoders use such nonlinear transforms

- Note: If inputs $x \in \{0, 1\}^D$ are binary, it may be more appropriate to also define $\hat{x}$ as

$$\hat{x} = \text{sigmoid}(W^*h + c)$$
What’s Learned by an Autoencoder?

Figure below: The $K \times D$ matrix $W$ learned on digits data. Each tiny block visualizes a row of $W$. Thus $W$ captures the possible "patterns" in the training data (akin to the $K$ basis vectors in PCA). For any input $x$, the encoding $h$ tells us how much each of these $K$ features is present in $x$.
What’s Learned by an Autoencoder?

Figure below: The $K \times D$ matrix $W$ learned on digits data. Each tiny block visualizes a row of $W$

- Thus $W$ captures the possible “patterns” in the training data (akin to the $K$ basis vectors in PCA)
What’s Learned by an Autoencoder?

Figure below: The $K \times D$ matrix $W$ learned on digits data. Each tiny block visualizes a row of $W$

- Thus $W$ captures the possible “patterns” in the training data (akin to the $K$ basis vectors in PCA).
- For any input $x$, the encoding $h$ tells us how much each of these $K$ features in present in $x$. 
Training the Autoencoder

To train the autoencoder, we need to define a **loss function** \( \ell(\hat{x}, x) \)

The loss function (a function of parameters \( W, b, W^*, c \)) can be defined using various ways. In general, it is defined in terms of the difference between \( \hat{x} \) and \( x \) (reconstruction error).

For a single input \( x = [x_1, \ldots, x_D] \) and its reconstruction \( \hat{x} = [\hat{x}_1, \ldots, \hat{x}_D] \):

\[
\ell(\hat{x}, x) = \sum_{d=1}^{D} (\hat{x}_d - x_d)^2 \quad \text{(squared loss; used if input are real-valued)}
\]

\[
\ell(\hat{x}, x) = -\sum_{d=1}^{D} [x_d \log(\hat{x}_d) + (1 - x_d) \log(1 - \hat{x}_d)] \quad \text{(cross-entropy loss; used if input are binary)}
\]

We find \((W, b, W^*, c)\) by minimizing the reconstruction error (summed over all training data). This can be done using backpropagation.
Training the Autoencoder

- To train the autoencoder, we need to define a loss function $\ell(\hat{x}, x)$
- The loss function (a function of parameters $W, b, W^*, c$) can be defined using various ways

\[ \ell(\hat{x}, x) = \frac{1}{D} \sum_{d=1}^{D} (\hat{x}_d - x_d)^2 \] (squared loss; used if input are real-valued)

\[ \ell(\hat{x}, x) = -\frac{1}{D} \sum_{d=1}^{D} [x_d \log(\hat{x}_d) + (1 - x_d) \log(1 - \hat{x}_d)] \] (cross-entropy loss; used if input are binary)

We find ($W, b, W^*, c$) by minimizing the reconstruction error (summed over all training data)

This can be done using backpropagation.
Training the Autoencoder

To train the autoencoder, we need to define a loss function \( \ell(\hat{x}, x) \).

The loss function (a function of parameters \( W, b, W^*, c \)) can be defined using various ways.

In general, it is defined in terms of the difference between \( \hat{x} \) and \( x \) (reconstruction error).

\[
\ell(\hat{x}, x) = \sum_{d=1}^{D} (\hat{x}_d - x_d)^2 \quad \text{(squared loss; used if input are real-valued)}
\]

\[
\ell(\hat{x}, x) = -\sum_{d=1}^{D} \left[ x_d \log(\hat{x}_d) + (1-x_d) \log(1-\hat{x}_d) \right] \quad \text{(cross-entropy loss; used if input are binary)}
\]
To train the autoencoder, we need to define a loss function $\ell(\hat{x}, x)$.

The loss function (a function of parameters $W, b, W^*, c$) can be defined using various ways.

In general, it is defined in terms of the difference between $\hat{x}$ and $x$ (reconstruction error).

For a single input $x = [x_1, \ldots, x_D]$ and its reconstruction $\hat{x} = [\hat{x}_1, \ldots, \hat{x}_D]$

$$\ell(\hat{x}, x) = \sum_{d=1}^{D} (\hat{x}_d - x_d)^2$$

(squared loss; used if input are real-valued)
Training the Autoencoder

- To train the autoencoder, we need to define a loss function $\ell(\hat{x}, x)$.
- The loss function (a function of parameters $W, b, W^*, c$) can be defined using various ways.
- In general, it is defined in terms of the difference between $\hat{x}$ and $x$ (reconstruction error).
- For a single input $x = [x_1, \ldots, x_D]$ and its reconstruction $\hat{x} = [\hat{x}_1, \ldots, \hat{x}_D]$

$$\ell(\hat{x}, x) = \sum_{d=1}^{D} (\hat{x}_d - x_d)^2 \quad \text{(squared loss; used if input are real-valued)}$$

$$\ell(\hat{x}, x) = - \sum_{d=1}^{D} [x_d \log(\hat{x}_d) + (1 - x_d) \log(1 - \hat{x}_d)] \quad \text{(cross-entropy loss; used if input are binary)}$$
Training the Autoencoder

- To train the autoencoder, we need to define a loss function $\ell(\hat{x}, x)$
- The loss function (a function of parameters $W, b, W^*, c$) can be defined using various ways.
- In general, it is defined in terms of the difference between $\hat{x}$ and $x$ (reconstruction error).
- For a single input $x = [x_1, \ldots, x_D]$ and its reconstruction $\hat{x} = [\hat{x}_1, \ldots, \hat{x}_D]$

$$\ell(\hat{x}, x) = \sum_{d=1}^{D} (\hat{x}_d - x_d)^2$$  \hspace{1cm} \text{(squared loss; used if input are real-valued)}

$$\ell(\hat{x}, x) = -\sum_{d=1}^{D} [x_d \log(\hat{x}_d) + (1 - x_d) \log(1 - \hat{x}_d)]$$ \hspace{1cm} \text{(cross-entropy loss; used if input are binary)}

- We find $(W, b, W^*, c)$ by minimizing the reconstruction error (summed over all training data).
Training the Autoencoder

- To train the autoencoder, we need to define a loss function $\ell(\hat{x}, x)$.
- The loss function (a function of parameters $W, b, W^*, c$) can be defined using various ways.
- In general, it is defined in terms of the difference between $\hat{x}$ and $x$ (reconstruction error).
- For a single input $x = [x_1, \ldots, x_D]$ and its reconstruction $\hat{x} = [\hat{x}_1, \ldots, \hat{x}_D]$

$$
\ell(\hat{x}, x) = \sum_{d=1}^{D} (\hat{x}_d - x_d)^2 \quad \text{(squared loss; used if input are real-valued)}
$$

$$
\ell(\hat{x}, x) = - \sum_{d=1}^{D} [x_d \log(\hat{x}_d) + (1 - x_d) \log(1 - \hat{x}_d)] \quad \text{(cross-entropy loss; used if input are binary)}
$$

- We find $(W, b, W^*, c)$ by minimizing the reconstruction error (summed over all training data).
- This can be done using backpropagation.
Undercomplete, Overcomplete, and Need for Regularization

In both cases, it is important to control the capacity of encoder and decoder.

**Undercomplete:**
Imagine $K = 1$ and very powerful $f$ and $g$. Can achieve very small reconstruction error but the learned code will not capture any interesting properties in the data.

**Overcomplete:**
Imagine $K \geq D$ and trivial (identity) functions $f$ and $g$. Can achieve even zero reconstruction error but the learned code will not capture any interesting properties in the data.

It is therefore important to regularize the functions as well as the learned code, and not just focus on minimizing the reconstruction error.

Piyush Rai (IIT Kanpur)
In both cases, it is important to control the capacity of encoder and decoder.
Undercomplete, Overcomplete, and Need for Regularization

In both cases, it is important to control the capacity of encoder and decoder.

**Undercomplete:** Imagine $K = 1$ and very powerful $f$ and $g$. Can achieve very small reconstruction error but the learned code will not capture any interesting properties in the data.

**Overcomplete:** Imagine $K \geq D$ and trivial (identity) functions $f$ and $g$. Can achieve even zero reconstruction error but the learned code will not capture any interesting properties in the data.

It is therefore important to regularize the functions as well as the learned code, and not just focus on minimizing the reconstruction error.
In both cases, it is important to control the capacity of encoder and decoder.

**Undercomplete**: Imagine $K = 1$ and very powerful $f$ and $g$. Can achieve very small reconstruction error but the learned code will not capture any interesting properties in the data.

**Overcomplete**: Imagine $K \geq D$ and trivial (identity) functions $f$ and $g$. Can achieve even zero reconstruction error but the learned code will not capture any interesting properties in the data.
In both cases, it is important to control the capacity of encoder and decoder.

**Undercomplete:** Imagine $K = 1$ and very powerful $f$ and $g$. Can achieve very small reconstruction error but the learned code will not capture any interesting properties in the data.

**Overcomplete:** Imagine $K \geq D$ and trivial (identity) functions $f$ and $g$. Can achieve even zero reconstruction error but the learned code will not capture any interesting properties in the data.

It is therefore important to regularize the functions as well as the learned code, and not just focus on minimizing the reconstruction error.
Regularized Autoencoders

- Several ways to regularize the model, e.g.
  - Make the learned code sparse (Sparse Autoencoders)
  - Make the model robust against noisy/incomplete inputs (Denoising Autoencoders)
Regularized Autoencoders

- Several ways to regularize the model, e.g.
  - Make the learned code sparse (Sparse Autoencoders)
  - Make the model robust against noisy/incomplete inputs (Denoising Autoencoders)
  - Make the model robust against small changes in the input (Contractive Autoencoders)
Sparse Autoencoders

- Make the learned code sparse (Sparse Autoencoders). Done by adding a sparsity penalty on $h$

Loss Function: $\ell(\hat{x}, x) + \Omega(h)$

where $\Omega(h) = \sum_{k=1}^{K} |h_k|$ is the $\ell_1$ norm of $h$
• Make the learned code sparse (Sparse Autoencoders). Done by adding a sparsity penalty on $h$

Loss Function: $\ell(\hat{x}, x) + \Omega(h)$

where $\Omega(h) = \sum_{k=1}^{K} |h_k|$ is the $\ell_1$ norm of $h$

• Sparse autoencoder is learned by minimizing the above regularized loss function
Denoising Autoencoders

- First add some noise (e.g., Gaussian noise) to the original input \( x \)
- Let’s denote \( \tilde{x} \) as the corrupted version of \( x \)
- The encoder \( f \) operates on \( \tilde{x} \), i.e., \( h = f(\tilde{x}) \)
Denoising Autoencoders

- First add some noise (e.g., Gaussian noise) to the original input \( x \)
- Let’s denote \( \tilde{x} \) as the corrupted version of \( x \)
- The encoder \( f \) operates on \( \tilde{x} \), i.e., \( h = f(\tilde{x}) \)

However, we still want to reconstruction \( \hat{x} \) to be close to the original uncorrupted input \( x \)
Denoising Autoencoders

- First add some noise (e.g., Gaussian noise) to the original input $x$
- Let’s denote $\tilde{x}$ as the corrupted version of $x$
- The encoder $f$ operates on $\tilde{x}$, i.e., $h = f(\tilde{x})$

However, we still want to reconstruction $\hat{x}$ to be close to the original uncorrupted input $x$

Since the corruption is stochastic, we minimize the expected loss: $$\mathbb{E}_{\tilde{x} \sim p(\tilde{x}|x)}[\ell(\hat{x}, \tilde{x})] + \Omega(h)$$
Deep/Stacked Autoencoders

- Most autoencoders can be extended to have more than one hidden layer.
Stochastic Autoencoders

- Can also define the encoder and decoder functions using probability distributions

\[
p_{\text{encoder}}(h|x)
\]
\[
p_{\text{decoder}}(x|h)
\]

The choice of distributions depends on the type of data being modeled and of the encodings. This gives a probabilistic approach for designing autoencoders. Negative log-likelihood \(-\log p_{\text{decoder}}(x|h)\) is equivalent to the reconstruction error. Can also use a prior distribution \(p(h)\) on the encodings (equivalent to regularizer). Such ideas have been used to design generative models for autoencoders. The Variational Autoencoder (VAE) is a popular example of such a model. Generative models like VAE can be used to "generate" new data using a random code \(h\).
Stochastic Autoencoders

- Can also define the encoder and decoder functions using probability distributions

\[ p_{\text{encoder}}(h|x) \]
\[ p_{\text{decoder}}(x|h) \]

- The choice of distributions depends on the type of data being modeled and of the encodings

\[ \text{Negative log-likelihood} \quad -\log p_{\text{decoder}}(x|h) \] is equivalent to the reconstruction error

Can also use a prior distribution \( p(h) \) on the encodings (equivalent to regularizer)

Such ideas have been used to design generative models for autoencoders

Variational Autoencoder (VAE) is a popular example of such a model

Generative models like VAE can be used to “generate” new data using a random code
Stochastic Autoencoders

- Can also define the encoder and decoder functions using probability distributions

\[ p_{\text{encoder}}(h|x) \]
\[ p_{\text{decoder}}(x|h) \]

- The choice of distributions depends on the type of data being modeled and of the encodings

- This gives a probabilistic approach for designing autoencoders

\[ \text{Negative log-likelihood} = -\log p_{\text{decoder}}(x|h) \]
\[ \text{is equivalent to the reconstruction error} \]
\[ \text{Can also use a prior distribution} \]
\[ p(h) \]
\[ \text{on the encodings (equivalent to regularizer)} \]
\[ \text{Such ideas have been used to design generative models for autoencoders} \]
\[ \text{Variational Autoencoder (VAE) is a popular example of such a model} \]
\[ \text{Generative models like VAE can be used to "generate" new data using a random code} \]
Stochastic Autoencoders

- Can also define the encoder and decoder functions using probability distributions

  \[ p_{\text{encoder}}(h|x) \]
  \[ p_{\text{decoder}}(x|h) \]

- The choice of distributions depends on the type of data being modeled and of the encodings
- This gives a probabilistic approach for designing autoencoders
- **Negative log-likelihood** \( -\log p_{\text{decoder}}(x|h) \) is equivalent to the reconstruction error

[^1]: Piyush Rai (IIT Kanpur)
Can also define the encoder and decoder functions using probability distributions

\[ p_{\text{encoder}}(h|\mathbf{x}) \]
\[ p_{\text{decoder}}(\mathbf{x}|h) \]

The choice of distributions depends on the type of data being modeled and of the encodings.

This gives a probabilistic approach for designing autoencoders.

**Negative log-likelihood** — \( \log p_{\text{decoder}}(\mathbf{x}|h) \) is equivalent to the reconstruction error.

Can also use a **prior distribution** \( p(h) \) on the encodings (equivalent to regularizer).
Stochastic Autoencoders

- Can also define the encoder and decoder functions using probability distributions

  \[ p_{\text{encoder}}(h|x) \]
  \[ p_{\text{decoder}}(x|h) \]

- The choice of distributions depends on the type of data being modeled and of the encodings

- This gives a probabilistic approach for designing autoencoders

- **Negative log-likelihood** — \( \log p_{\text{decoder}}(x|h) \) is equivalent to the reconstruction error

- Can also use a **prior distribution** \( p(h) \) on the encodings (equivalent to regularizer)

- Such ideas have been used to design **generative models** for autoencoders
Stochastic Autoencoders

- Can also define the encoder and decoder functions using probability distributions

\[
\begin{align*}
    p_{\text{encoder}}(h|x) \\
p_{\text{decoder}}(x|h)
\end{align*}
\]

- The choice of distributions depends on the type of data being modeled and of the encodings

- This gives a probabilistic approach for designing autoencoders

- Negative log-likelihood \( \log p_{\text{decoder}}(x|h) \) is equivalent to the reconstruction error

- Can also use a prior distribution \( p(h) \) on the encodings (equivalent to regularizer)

- Such ideas have been used to design generative models for autoencoders
  - Variational Autoencoder (VAE) is a popular example of such a model
Stochastic Autoencoders

- Can also define the encoder and decoder functions using probability distributions

\[ p_{\text{encoder}}(h|x) \]
\[ p_{\text{decoder}}(x|h) \]

- The choice of distributions depends on the type of data being modeled and of the encodings
- This gives a probabilistic approach for designing autoencoders
- **Negative log-likelihood** \(-\log p_{\text{decoder}}(x|h)\) is equivalent to the reconstruction error
- Can also use a **prior distribution** \(p(h)\) on the encodings (equivalent to regularizer)
- Such ideas have been used to design **generative models** for autoencoders
  - **Variational Autoencoder (VAE)** is a popular example of such a model
  - Generative models like VAE can be used to “generate” new data using a random code \(h\)
Variational Autoencoders (VAE)

- Learns a distribution (e.g., a Gaussian) on the encoding

\[1\text{http://www.birving.com/presentations/autoencoders/}\]
Variational Autoencoders (VAE)

- Learns a distribution (e.g., a Gaussian) on the encoding\(^1\)

- Unlike standard AE, a VAE model learns to generate plausible data from random encodings

---

\(^1\)http://www.birving.com/presentations/autoencoders/
Some Applications of Autoencoders

- (Unsupervised) Feature learning and Dimensionality reduction
- Denoising and inpainting
- Pre-training of deep neural networks
- Recommender systems applications
Feature learning and Dimensionality Reduction

- Example: A deep AE for low-dim feature learning for 784-dimensional MNIST images

2 Figure credit: Hinton and Salakhutdinov
Feature learning and Dimensionality Reduction

- Example: Low-dim feature learning for 2000-dimensional bag-of-words documents
Denoising and Inpainting
Denoising and Inpainting
Applications in Recommender Systems
Assume we are given a partially known $N \times M$ ratings matrix $R$ of $N$ users on $M$ items (movies).

Denote by $r^{(u)}$ the (partially known) $M \times 1$ ratings vector of user $u$.

Denote by $r^{(i)}$ the (partially known) $N \times 1$ ratings vector of item $i$. 
Assume we are given a partially known $N \times M$ ratings matrix $\mathbf{R}$ of $N$ users on $M$ items (movies)

Denote by $\mathbf{r}^{(u)}$ the (partially known) $M \times 1$ ratings vector of user $u$

Denote by $\mathbf{r}^{(i)}$ the (partially known) $N \times 1$ ratings vector of item $i$

How can we use this data to build a recommender system?
An idea: If the predicted value of a user’s rating for a movie is high, then we should ideally recommend this movie to the user.
An idea: If the predicted value of a user’s rating for a movie is high, then we should ideally recommend this movie to the user.

Thus if we can “reconstruct” the missing entries in $\mathbf{R}$, we can use this method to recommend movies to users. Using an autoencoders can help us do this!
An Autoencoder based Approach

- Using the rating vectors $\{r^{(u)}\}_{u=1}^{N}$ of all users, can learn an autoencoder

$$\hat{r}^{(u)}$$

- Note: During backprop, only update weights in $W$ that are connected to the observed ratings\(^3\)
- Once learned, the model can predict (reconstruct) the missing ratings

\(^3\)AutoRec: Autoencoders Meet Collaborative Filtering (Sedhain et al, WWW 2015)
Another Autoencoder based Approach

Another approach is to combine (denoising) autoencoders with a matrix factorization model\(^4\)

---

\(^4\)Deep Collaborative Filtering via Marginalized Denoising Auto-encoder (Li et al, CIKM 2015)
Another Autoencoder based Approach

- Another approach is to combine (denoising) autoencoders with a matrix factorization model\(^4\)
- Idea: Rating of a user \(u\) on an item \(i\) can be defined using the inner-product based similarity of their features learned via an autoencoder: 
  \[ R_{ui} = f(h^{(u)\top} h^{(i)}) \]
  where \(f\) is some compatibility function

---

\(^4\)Deep Collaborative Filtering via Marginalized Denoising Auto-encoder (Li et al, CIKM 2015)
Another Autoencoder based Approach

- Another approach is to combine (denoising) autoencoders with a matrix factorization model\(^4\).

- Idea: Rating of a user \(u\) on an item \(i\) can be defined using the inner-product based similarity of their features learned via an autoencoder: \(R_{ui} = f(h^{(u)\top}h^{(i)})\) where \(f\) is some compatibility function.

- Denoting \(\{h^{(u)}\}_{u=1}^{N} = U\) and \(\{h^{(i)}\}_{i=1}^{M} = V\), we can write \(R = UV^\top\).

\(^4\)Deep Collaborative Filtering via Marginalized Denoising Auto-encoder (Li et al, CIKM, 2015)
Several recent papers on similar autoencoder based ideas

- Collaborative Denoising Auto-Encoders for Top-N Recommender Systems (Wu et al, WSDM 2016)

Also possible to incorporate side information about the users and/or items (Wang et al, KDD 2015)
Autoencoders: Summary

- Simple and powerful for (nonlinear) feature learning
- Learned features are able to capture salient properties of data
- Several extensions (sparse, denoising, stochastic, etc.)
- Can also be stacked to create “deep” autoencoders
- Recent focus on autoencoders that are based on generative models of data
  - Example: Variational Autoencoders