Autoencoders

Introduction [1]

- Autoencoders (AE) are neural networks that aims to copy their inputs to their outputs.

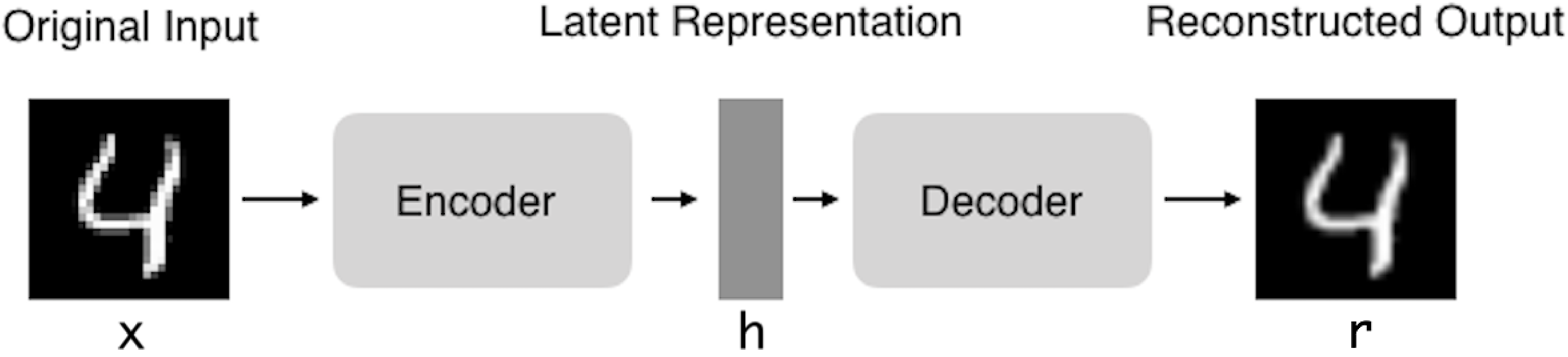
Introduction [1]

- Autoencoders (AE) are neural networks that aims to copy their inputs to their outputs.

- They work by compressing the input into a latent-space representation, and then reconstructing the output from this representation.

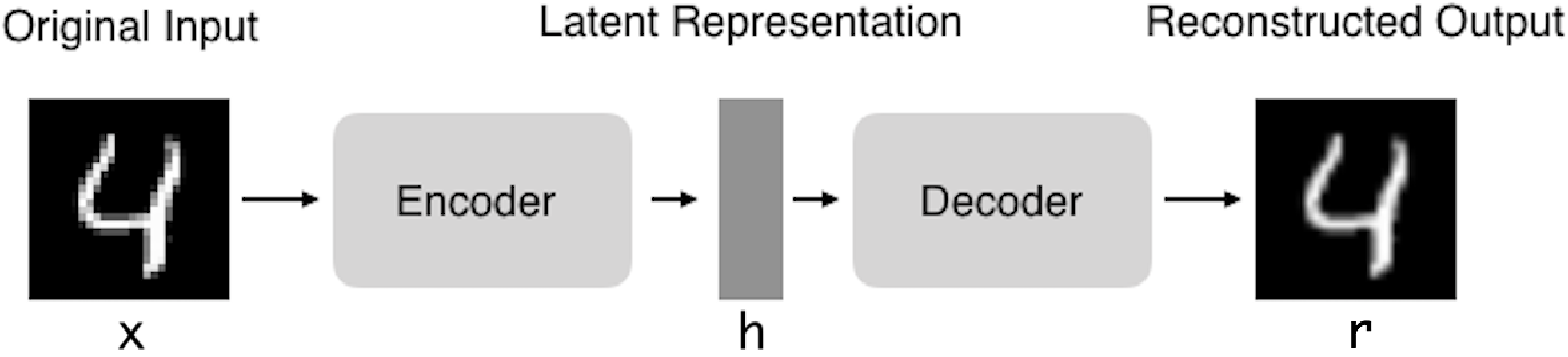
Introduction [1]

- This kind of network is composed of two parts:



Introduction [1]

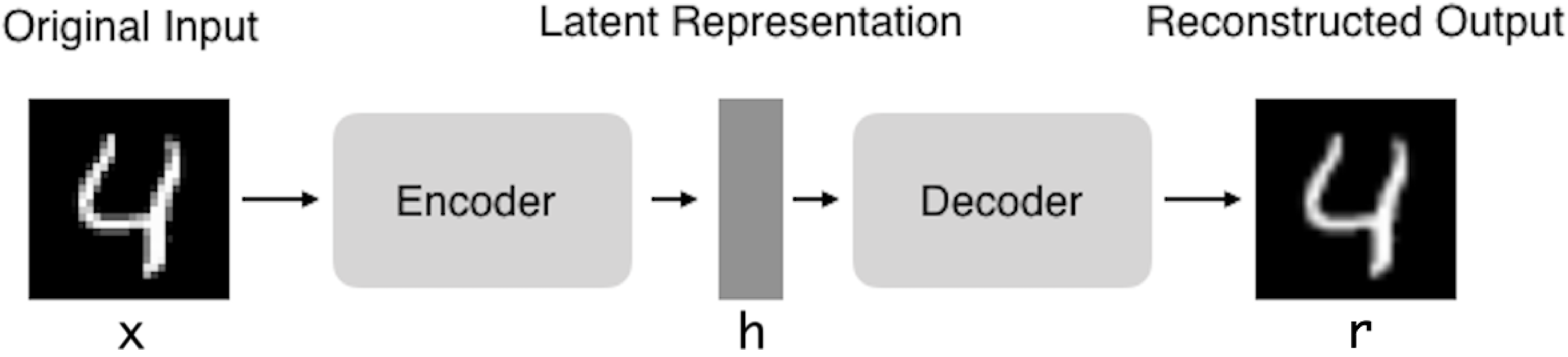
- This kind of network is composed of two parts:



- Encoder: This is the part of the network that compresses the input into a latent-space representation. It can be represented by an encoding function h=f(x).

Introduction [1]

- This kind of network is composed of two parts:

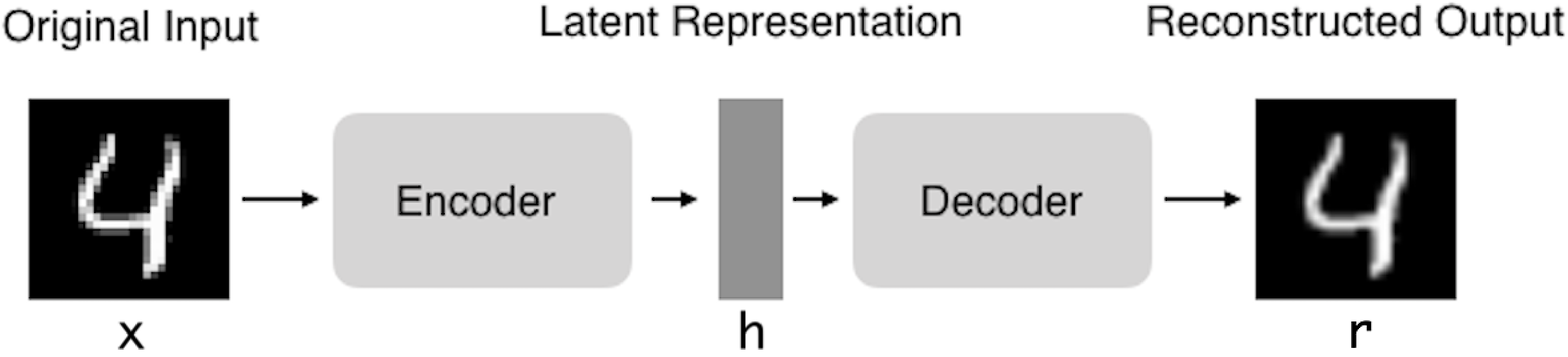


- Encoder: This is the part of the network that compresses the input into a latent-space representation. It can be represented by an encoding function h=f(x).

- Decoder: This part aims to reconstruct the input from the latent space representation. It can be represented by a decoding function r=g(h).

Introduction [1]

- This kind of network is composed of two parts:



- Encoder: This is the part of the network that compresses the input into a latent-space representation. It can be represented by an encoding function h=f(x).

- Decoder: This part aims to reconstruct the input from the latent space representation. It can be represented by a decoding function r=g(h).

- The autoencoder as a whole can thus be described by the function g(f(x)) = r where you want r as close as the original input x.

Why copying the input to the output ? [1]

- If the only purpose of autoencoders was to copy the input to the output, they would be useless.

Why copying the input to the output ? [1]

- If the only purpose of autoencoders was to copy the input to the output, they would be useless.

- Indeed, we hope that, by training the autoencoder to copy the input to the output, the latent representation h will take on useful properties.

Why copying the input to the output ? [1]

- This can be achieved by creating constraints on the copying task.

Why copying the input to the output ? [1]

- This can be achieved by creating constraints on the copying task.

- One way to obtain useful features from the autoencoder is to constrain h to have smaller dimensions than x, in this case the autoencoder is called undercomplete.

Why copying the input to the output ? [1]

- This can be achieved by creating constraints on the copying task.

- One way to obtain useful features from the autoencoder is to constrain h to have smaller dimensions than x, in this case the autoencoder is called undercomplete.

- By training an undercomplete representation, we force the autoencoder to learn the most salient features of the training data.

Why copying the input to the output ? [1]

- This can be achieved by creating constraints on the copying task.

- One way to obtain useful features from the autoencoder is to constrain h to have smaller dimensions than x, in this case the autoencoder is called undercomplete.

- By training an undercomplete representation, we force the autoencoder to learn the most salient features of the training data.

- If the autoencoder is given too much capacity, it can learn to perform the copying task without extracting any useful information about the distribution of the data.

Why copying the input to the output ? [1]

- This can also occur if the dimension of the latent representation is the same as the input, and in the overcomplete case, where the dimension of the latent representation is greater than the input.

Why copying the input to the output ? [1]

- This can also occur if the dimension of the latent representation is the same as the input, and in the overcomplete case, where the dimension of the latent representation is greater than the input.

- In these cases, even a linear encoder and linear decoder can learn to copy the input to the output without learning anything useful about the data distribution.

Why copying the input to the output ? [1]

- This can also occur if the dimension of the latent representation is the same as the input, and in the overcomplete case, where the dimension of the latent representation is greater than the input.

- In these cases, even a linear encoder and linear decoder can learn to copy the input to the output without learning anything useful about the data distribution.

- Ideally, one could train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modelled.

Types of Autoencoders [1]

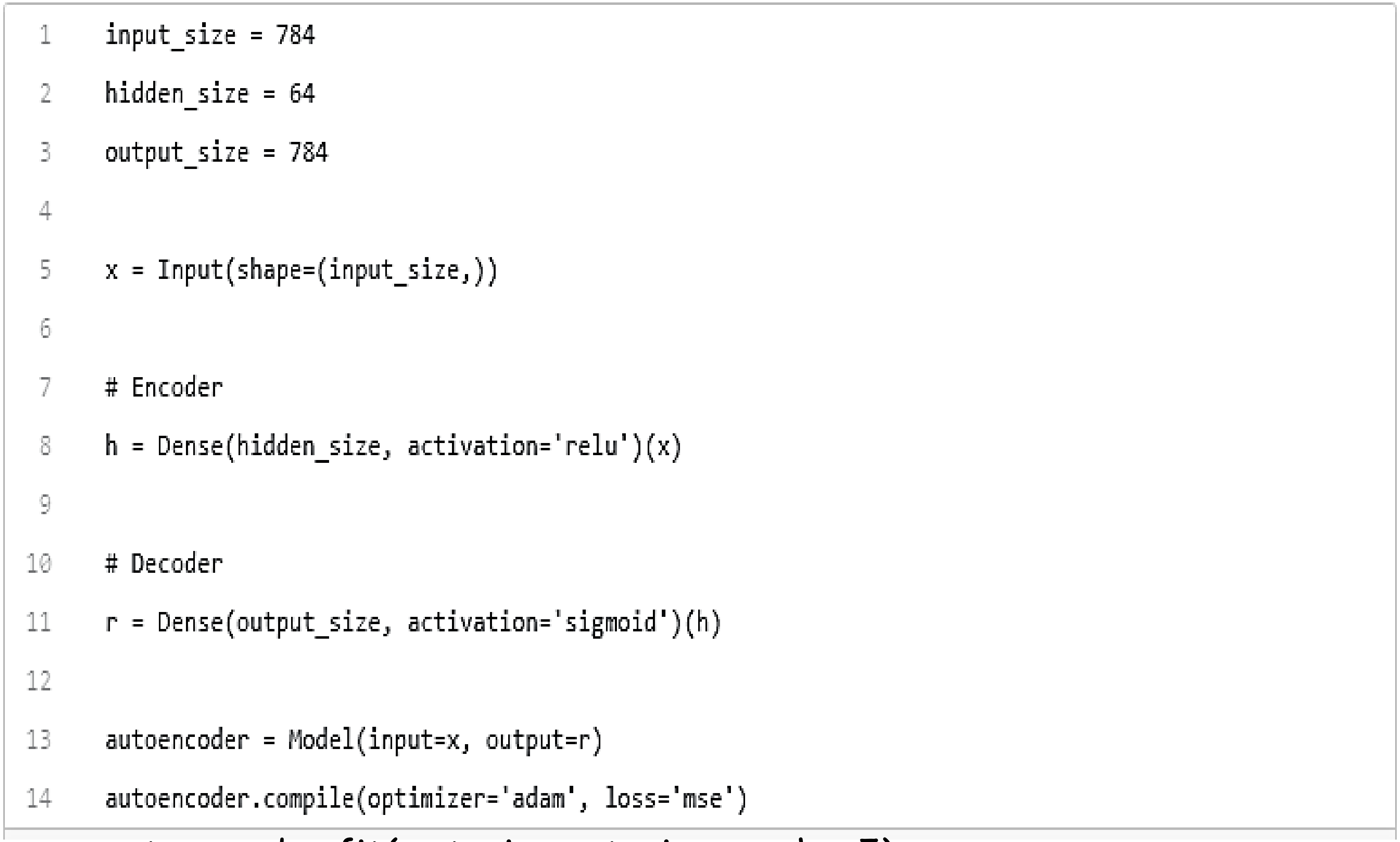
- Vanilla autoencoder
- Multilayer autoencoder
- Convolutional autoencoder
- Regularized autoencoder

Types of Autoencoders [1]

- Vanilla autoencoder
  - In its simplest form, the autoencoder is a three layers net, i.e. a neural net with one hidden layer.
  - The input and output are the same, and we learn how to reconstruct the input, for example using the adam optimizer and the mean squared error loss function.

Types of Autoencoders [1]

- Vanilla autoencoder



```
input_size = 784
hidden_size = 64
output_size = 784

x = Input(shape=(input_size,))

# Encoder
h = Dense(hidden_size, activation='relu')(x)

# Decoder
r = Dense(output_size, activation='sigmoid')(h)

autoencoder = Model(input=x, output=r)
autoencoder.compile(optimizer='adam', loss='mse')
```

```
autoencoder.fit(x_train, x_train, epochs=5)
reconstructed = autoencoder.predict(x_test)
```

Types of Autoencoders [2]

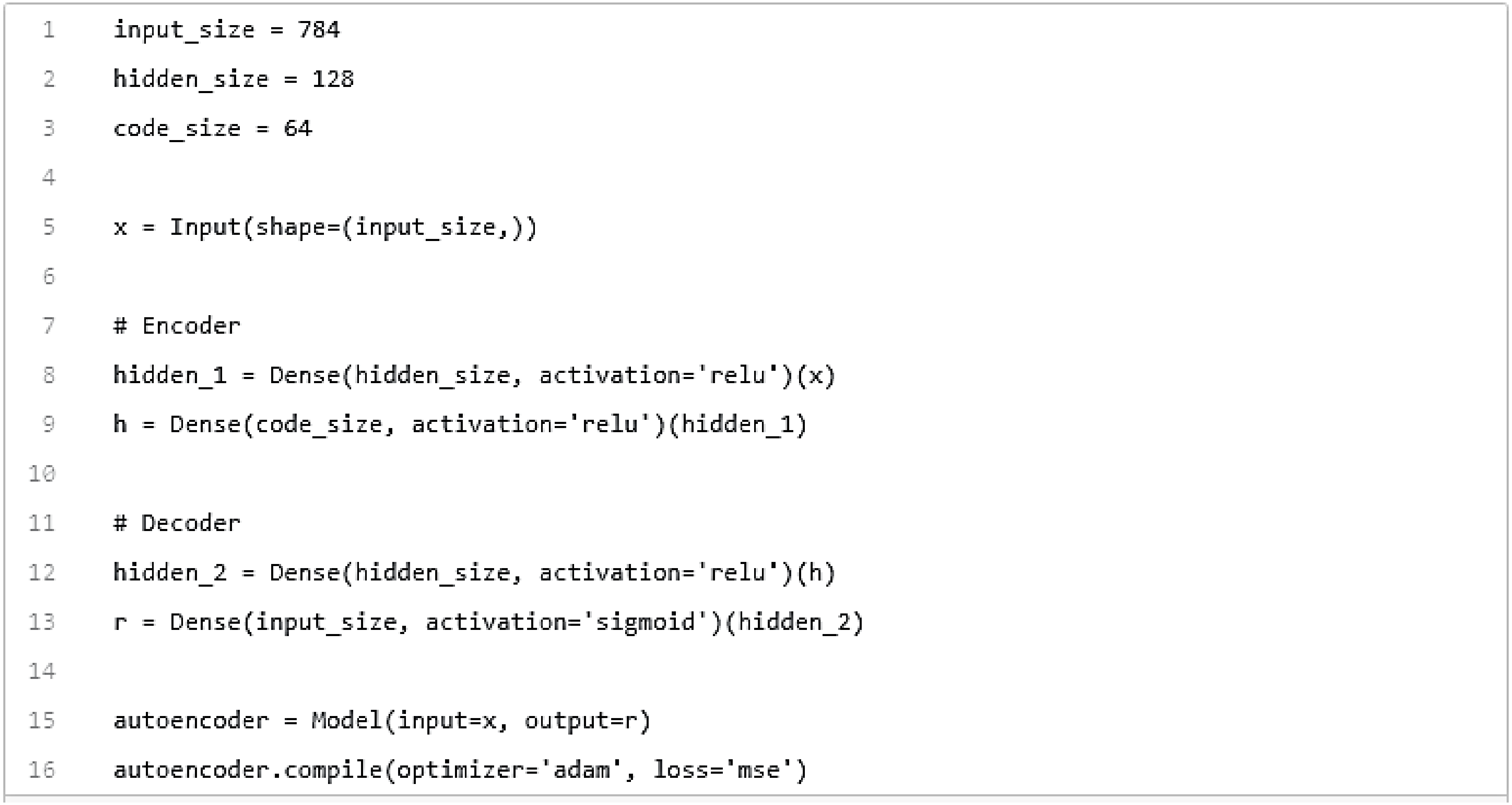
- Vanilla autoencoder
  - Practical Advise:
    - We have total control over the architecture of the autoencoder.
    - We can make it very powerful by increasing the number of layers, nodes per layer and most importantly the code size.
    - Increasing these hyperparameters will let the autoencoder to learn more complex codings.
    - But we should be careful to not make it too powerful.
    - Otherwise the autoencoder will simply learn to copy its inputs to the output, without learning any meaningful representation.
    - It will just mimic the identity function.
    - The autoencoder will reconstruct the training data perfectly, but it will be overfitting without being able to generalize to new instances, which is not what we want.

Types of Autoencoders [2]

- Vanilla autoencoder
  - Practical Advise:
    - This is why we prefer a "sandwitch" architecture, and deliberately keep the code size small.
    - Since the coding layer has a lower dimensionality than the input data, the autoencoder is said to be undercomplete.
    - It won't be able to directly copy its inputs to the output, and will be forced to learn intelligent features.
    - If the input data has a pattern, for example the digit "1" usually contains a somewhat straight line and the digit "0" is circular, it will learn this fact and encode it in a more compact form.
    - **If the input data was completely random without any internal correlation or dependency, then an undercomplete autoencoder won't be able to recover it perfectly.**
    - **But luckily, in the real-world there is a lot of dependency.**

Types of Autoencoders [1]

- Multilayer autoencoder



```
input_size = 784
hidden_size = 128
code_size = 64

x = Input(shape=(input_size,))

# Encoder
hidden_1 = Dense(hidden_size, activation='relu')(x)
h = Dense(code_size, activation='relu')(hidden_1)

# Decoder
hidden_2 = Dense(hidden_size, activation='relu')(h)
r = Dense(input_size, activation='sigmoid')(hidden_2)

autoencoder = Model(input=x, output=r)
autoencoder.compile(optimizer='adam', loss='mse')
```

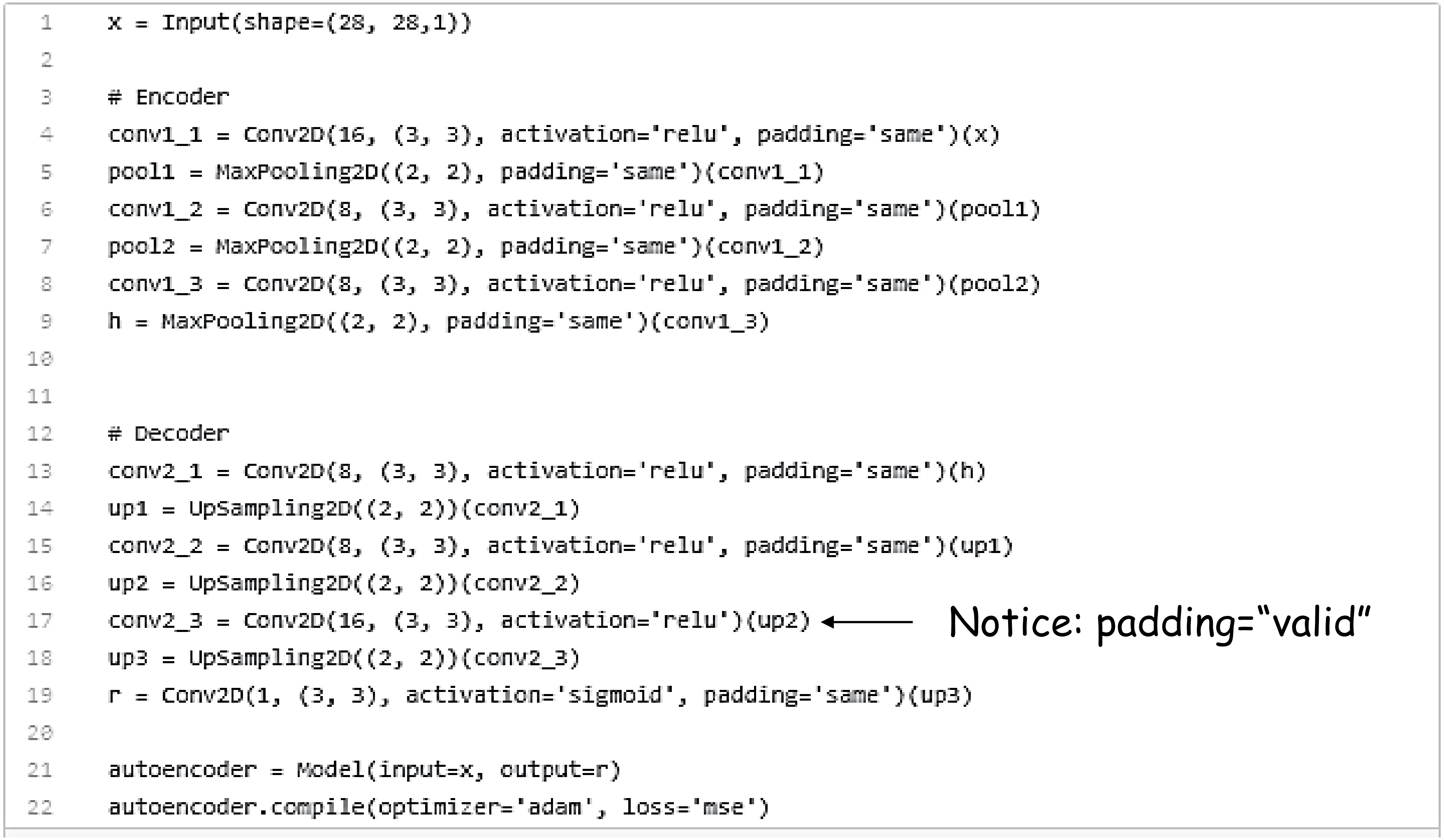
```
autoencoder.fit(x_train, x_train, epochs=5)
reconstructed = autoencoder.predict(x_test)
```

Types of Autoencoders [1]

- Multilayer autoencoder
  - Now our implementation uses 3 hidden layers instead of just one.
  - Any of the hidden layers can be picked as the feature representation but we will make the network symmetrical and use the middle-most layer.

Types of Autoencoders [1]

- Convolutional autoencoder



```
x = Input(shape=(28, 28,1))

# Encoder
conv1_1 = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
pool1 = MaxPooling2D((2, 2), padding='same')(conv1_1)
conv1_2 = Conv2D(8, (3, 3), activation='relu', padding='same')(pool1)
pool2 = MaxPooling2D((2, 2), padding='same')(conv1_2)
conv1_3 = Conv2D(8, (3, 3), activation='relu', padding='same')(pool2)
h = MaxPooling2D((2, 2), padding='same')(conv1_3)


# Decoder
conv2_1 = Conv2D(8, (3, 3), activation='relu', padding='same')(h)
up1 = UpSampling2D((2, 2))(conv2_1)
conv2_2 = Conv2D(8, (3, 3), activation='relu', padding='same')(up1)
up2 = UpSampling2D((2, 2))(conv2_2)
conv2_3 = Conv2D(16, (3, 3), activation='relu')(up2)
up3 = UpSampling2D((2, 2))(conv2_3)
r = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(up3)

autoencoder = Model(input=x, output=r)
autoencoder.compile(optimizer='adam', loss='mse')
```

Notice: padding="valid"

```
autoencoder.fit(x_train, x_train, epochs=5)
reconstructed = autoencoder.predict(x_test)
```

Types of Autoencoders [1]

- Regularized autoencoder
  - There are other ways we can constraint the reconstruction of an autoencoder than to impose a hidden layer of smaller dimension than the input.

  - Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output.

  - In practice, we usually find two types of regularized autoencoder:
    - the sparse autoencoder and
    - the denoising autoencoder.

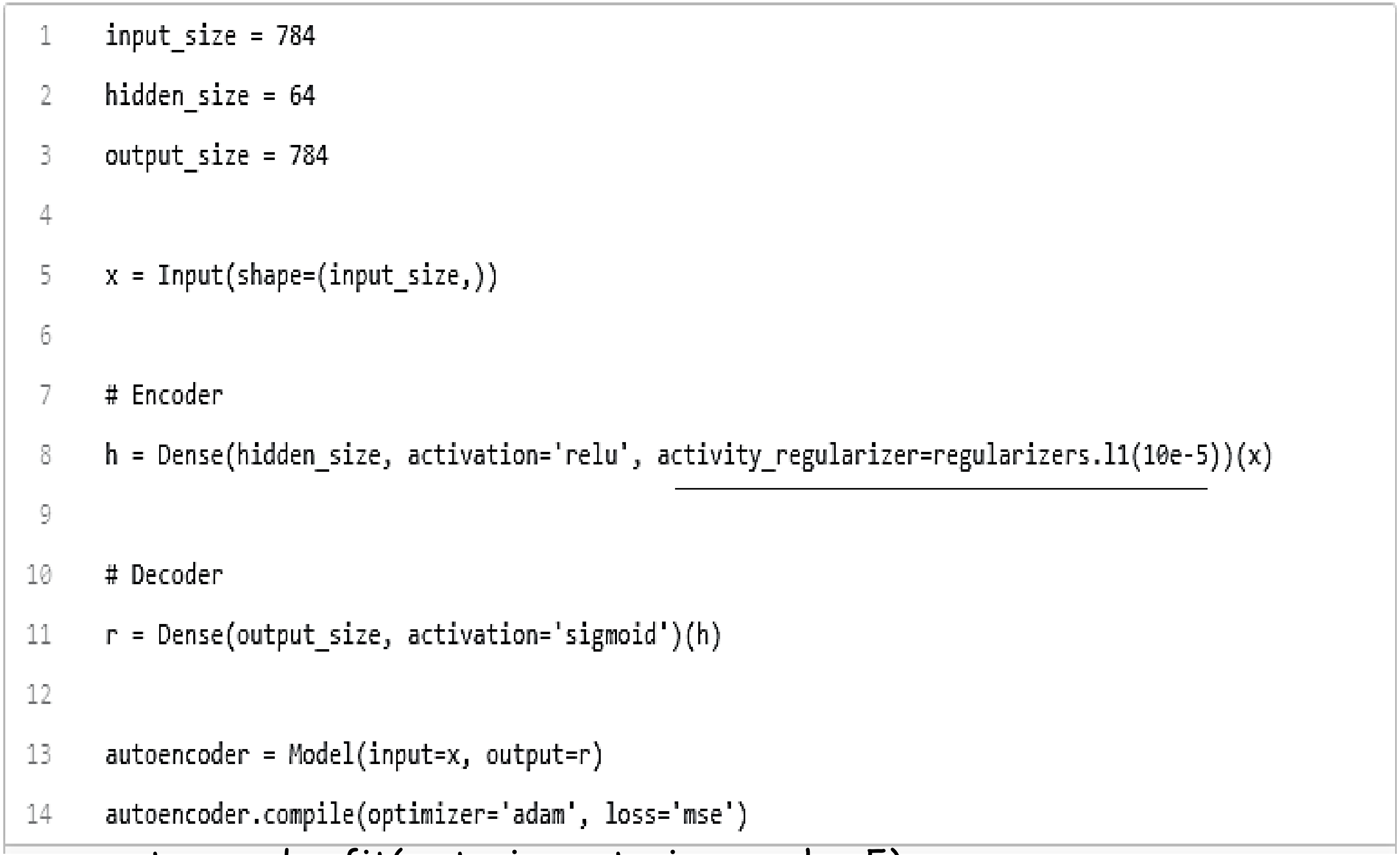
Types of Autoencoders

- Sparse autoencoder
  - Please recall, when we apply weight regularizer, it is like adding a term in loss function like $\sum_i \sum_j |w_{ij}|$ for L1 regularization and $\sum_i \sum_j w_{ij}^2$ for L2 regularization. This ensures that weights are very small and therefore our model is simple and we can avoid overfitting.

  - Here, in sparse autoencoder, we regularize output (that is activations) of neurons and therefore they are small and many are zero leading to a sparse representation.

Types of Autoencoders [1]

- Sparse autoencoder



```
input_size = 784
hidden_size = 64
output_size = 784

x = Input(shape=(input_size,))

# Encoder
h = Dense(hidden_size, activation='relu', activity_regularizer=regularizers.l1(10e-5))(x)

# Decoder
r = Dense(output_size, activation='sigmoid')(h)

autoencoder = Model(input=x, output=r)
autoencoder.compile(optimizer='adam', loss='mse')
```

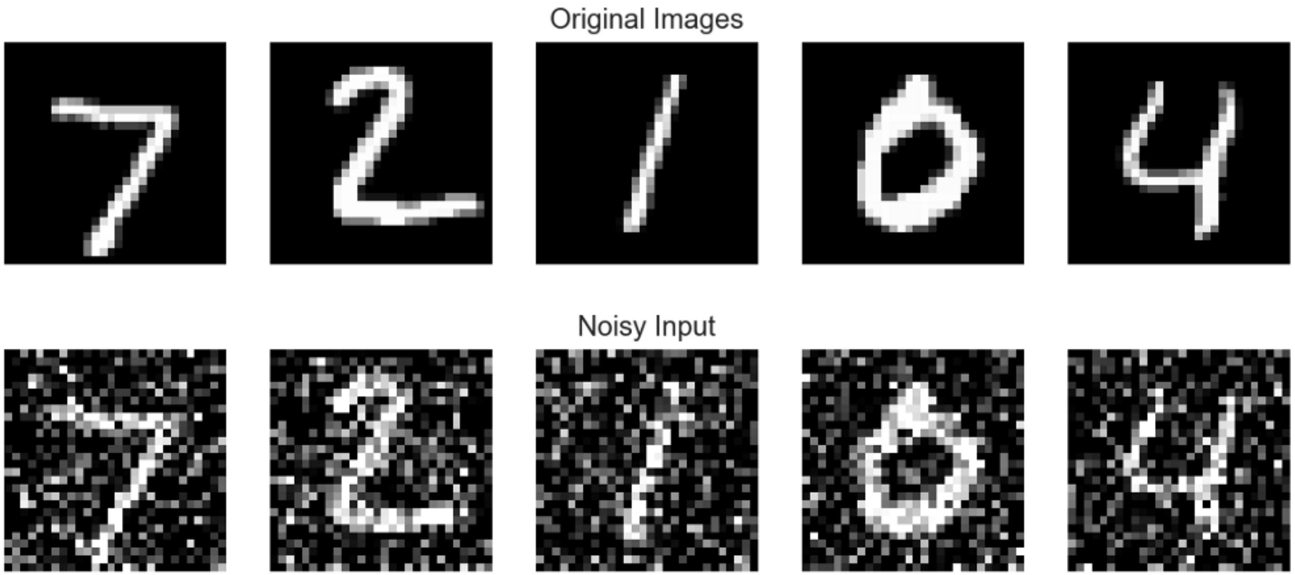
```
autoencoder.fit(x_train, x_train, epochs=5)
reconstructed = autoencoder.predict(x_test)
```

Types of Autoencoders [2]

- Denoising autoencoder
  - Keeping the code layer small forced our autoencoder to learn an intelligent representation of the data.
  - There is another way to force the autoencoder to learn useful features, which is adding random noise to its inputs and making it recover the original noise-free data.
  - This way the autoencoder can’t simply copy the input to its output because the input also contains random noise.
  - We are asking it to subtract the noise and produce the underlying meaningful data. This is called a denoising autoencoder.

Types of Autoencoders [2]

- Denoising autoencoder



- The top row contains the original images.

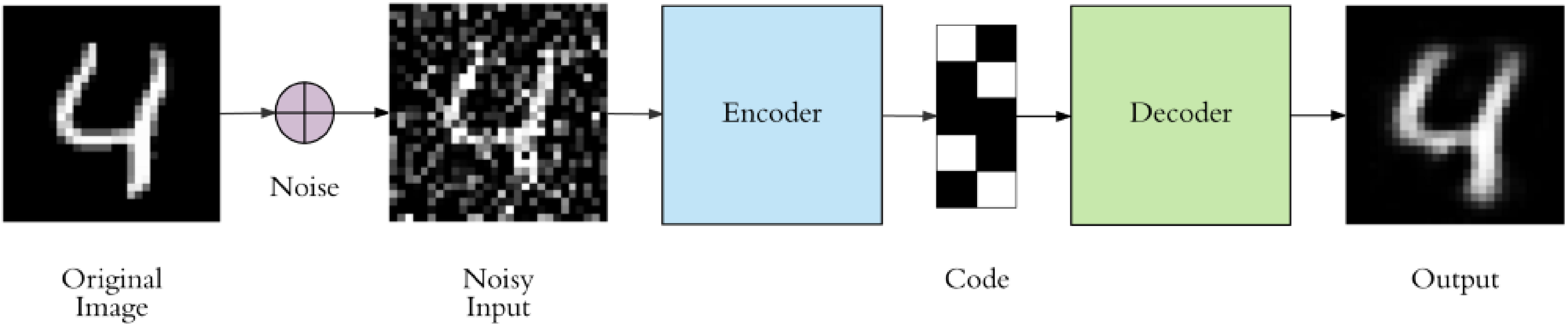
- We add random Gaussian noise to them and the noisy data becomes the input to the autoencoder.

- The autoencoder doesn't see the original image at all.

- But then we expect the autoencoder to regenerate the noise-free original image.

Types of Autoencoders [2]

- Denoising autoencoder



- There is only one small difference between the implementation of denoising autoencoder and the regular one. The architecture doesn't change at all, only the fit function.

- We trained the regular autoencoder as follows:
  - autoencoder.fit(x_train, x_train)

- Denoising autoencoder is trained as:
  - autoencoder.fit(x_train_noisy, x_train)

Types of Autoencoders [2]

- Denoising Autoencoders

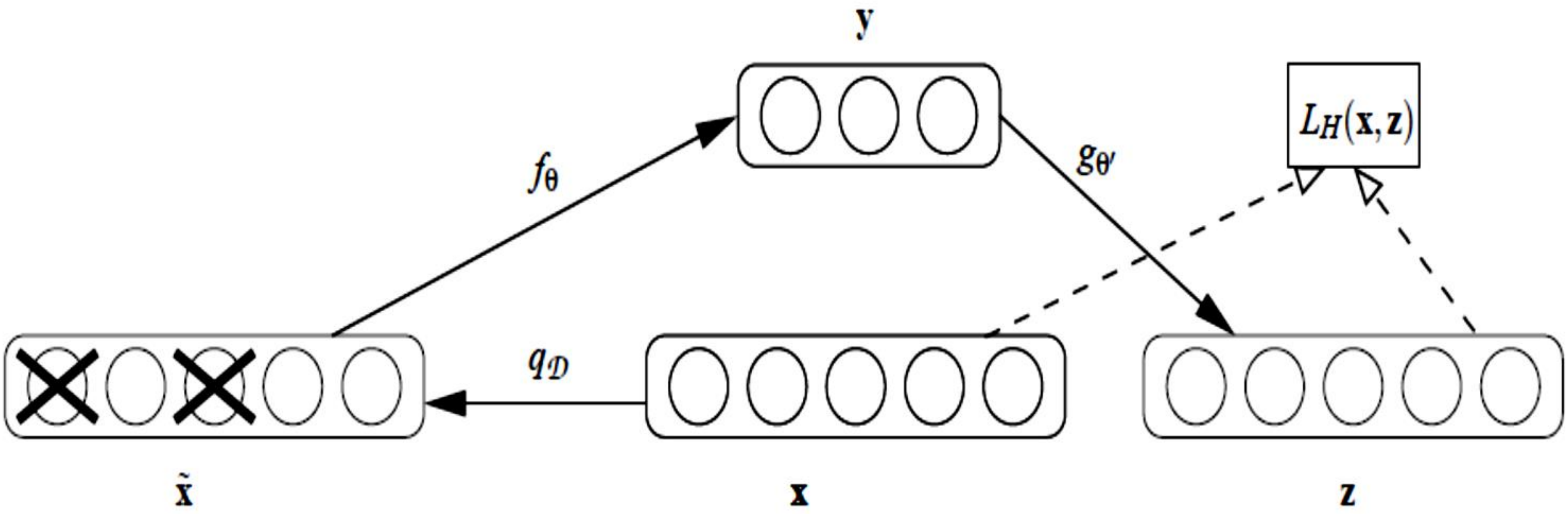
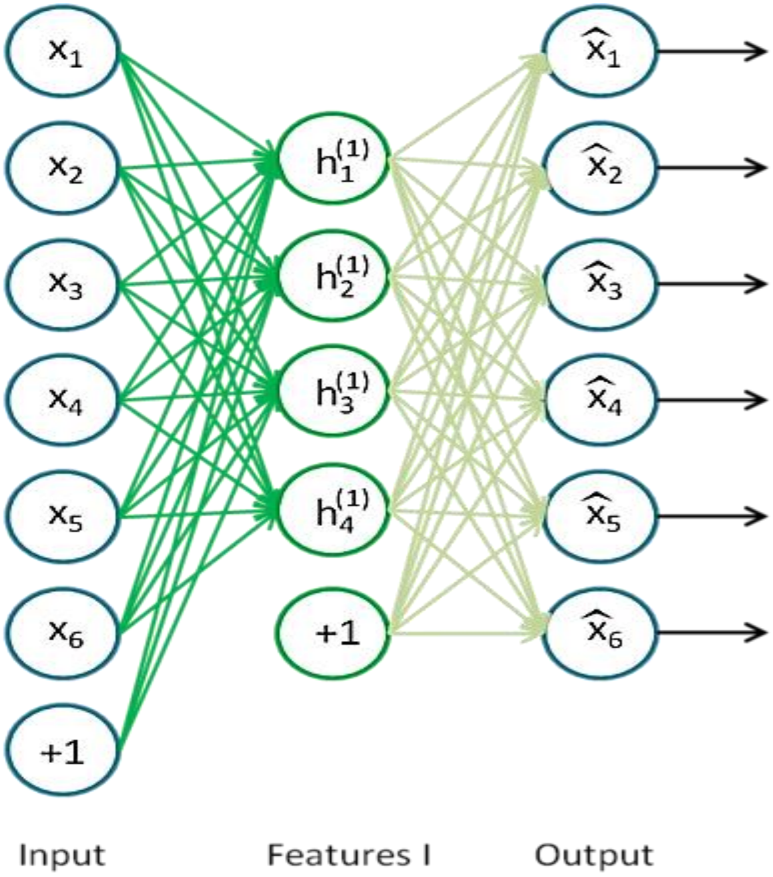


Figure 1: The denoising autoencoder architecture. An example $\mathbf{x}$ is stochastically corrupted (via $q_\mathcal{D}$) to $\tilde{\mathbf{x}}$. The autoencoder then maps it to $\mathbf{y}$ (via encoder f_θ) and attempts to reconstruct $\mathbf{x}$ via decoder $g_{\theta'}$, producing reconstruction $\mathbf{z}$. Reconstruction error is measured by loss $L_H(\mathbf{x},\mathbf{z})$.

Stacked Autoencoders [3]

- Suppose you wished to train a stacked autoencoder with 2 hidden layers for classification of MNIST digits.

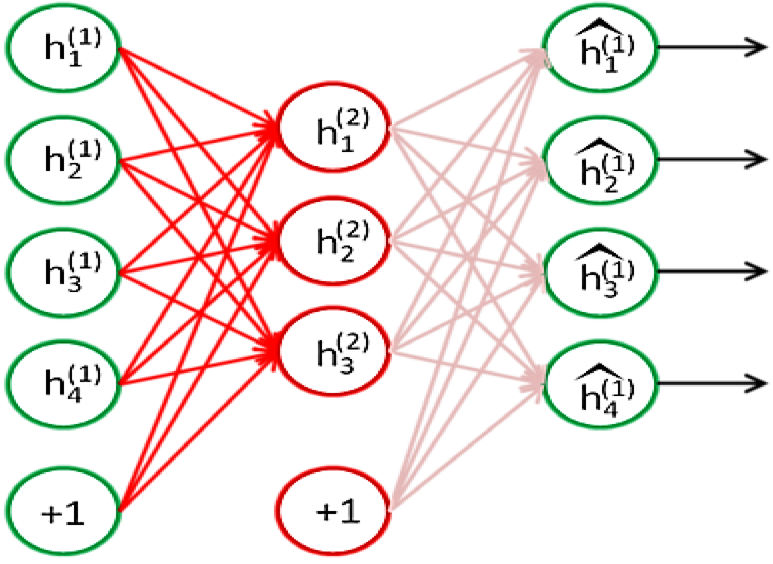
- First, you would train a sparse autoencoder on the raw inputs x(k) to learn primary features $h^{(1)}_{(k)}$ on the raw input.



Stacked Autoencoders [3]

- Next, you would feed the raw input into this trained sparse autoencoder, obtaining the primary feature activations $h^{(1)}{}_{(k)}$ for each of the inputs x(k).

- You would then use these primary features as the "raw input" to another sparse autoencoder to learn secondary features $h^{(2)}{}_{(k)}$ on these primary features.



Input (Features I)

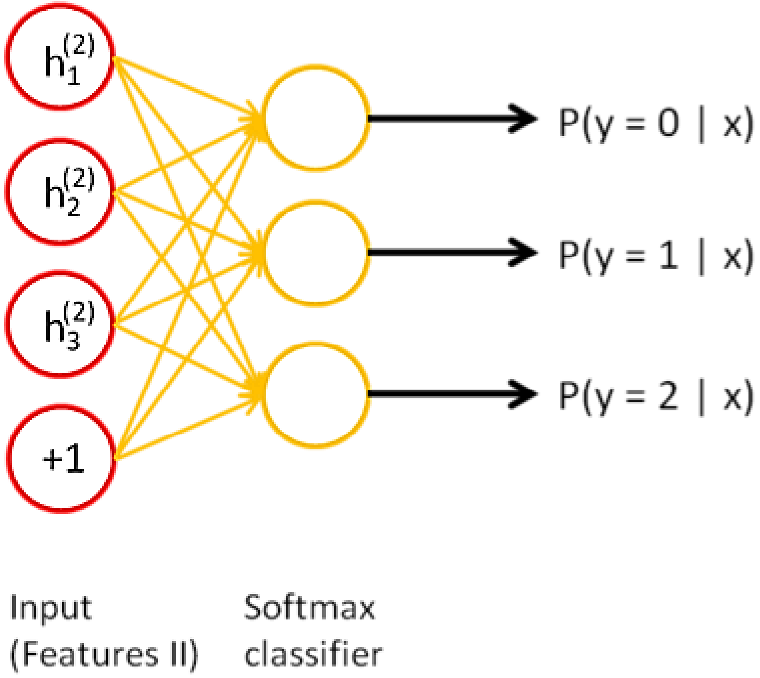
Features II

Output

Stacked Autoencoders [3]

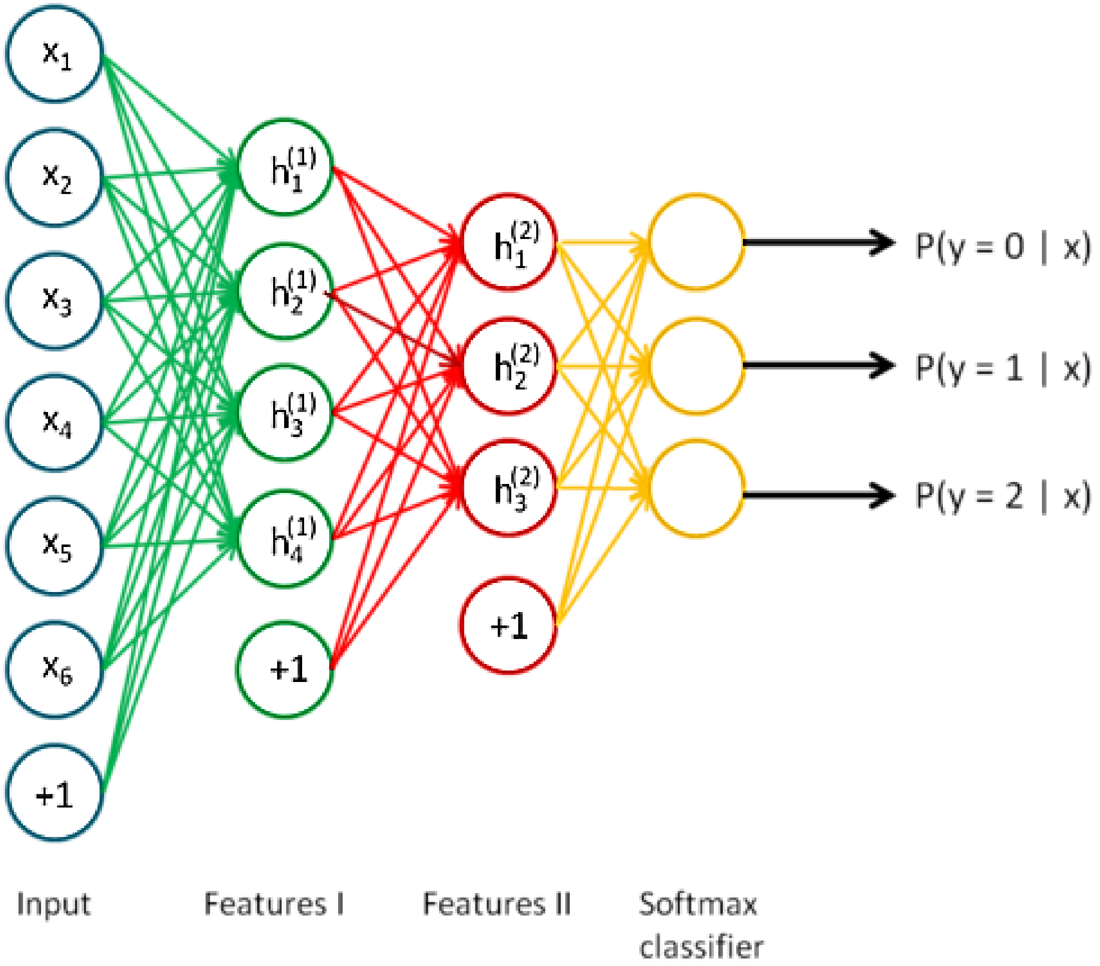
- Following this, you would feed the primary features into the second sparse autoencoder to obtain the secondary feature activations h(2)(k) for each of the primary features h(1)(k) (which correspond to the primary features of the corresponding inputs x(k)).

- You would then treat these secondary features as "raw input" to a softmax classifier, training it to map secondary features to digit labels.



Stacked Autoencoders [3]

- Finally, you would combine all three layers together to form a stacked autoencoder with 2 hidden layers and a final softmax classifier layer capable of classifying the MNIST digits as desired.



Stacked Autoencoders [3]

- A stacked autoencoder enjoys all the benefits of any deep network of greater expressive power.

- Further, it often captures a useful "hierarchical grouping" or "part-whole decomposition" of the input.

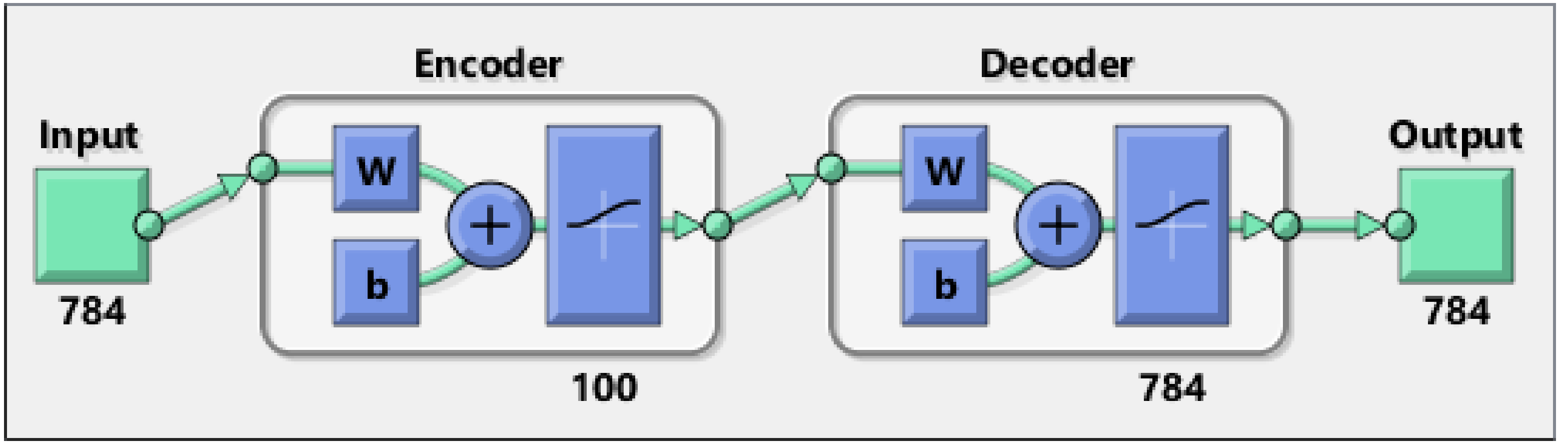
- To see this, recall that an autoencoder tends to learn features that form a good representation of its input.

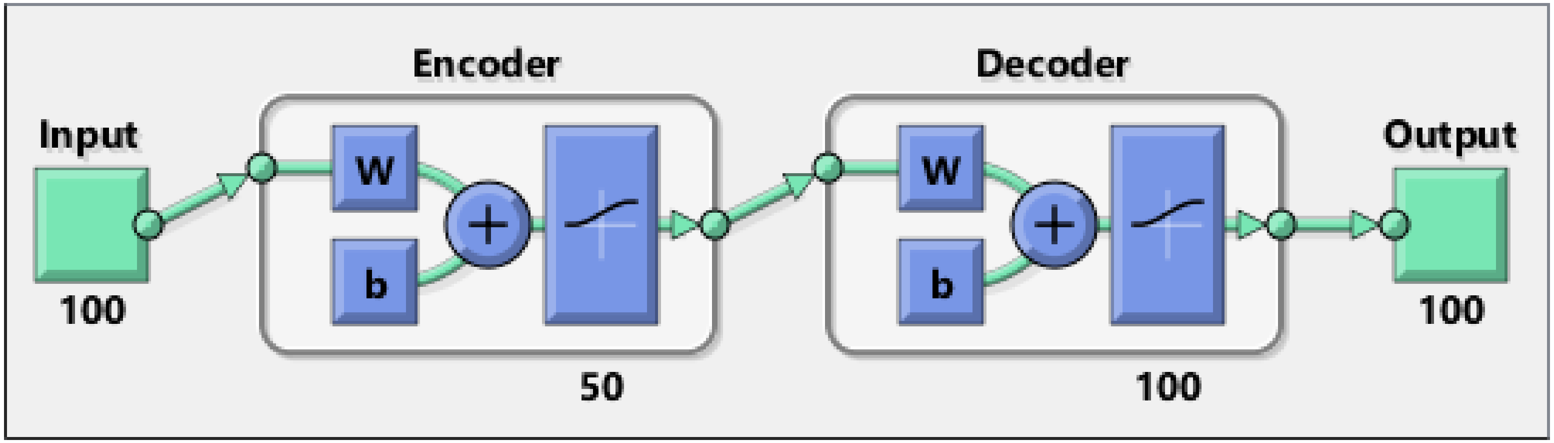
- The first layer of a stacked autoencoder tends to learn first-order features in the raw input (such as edges in an image).

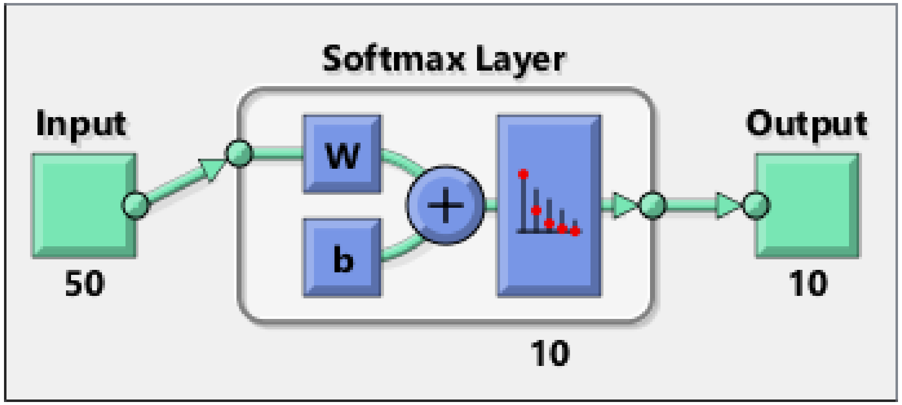
- The second layer of a stacked autoencoder tends to learn second-order features corresponding to patterns in the appearance of first-order features (e.g., in terms of what edges tend to occur together--for example, to form contour or corner detectors).

- Higher layers of the stacked autoencoder tend to learn even higher-order features.

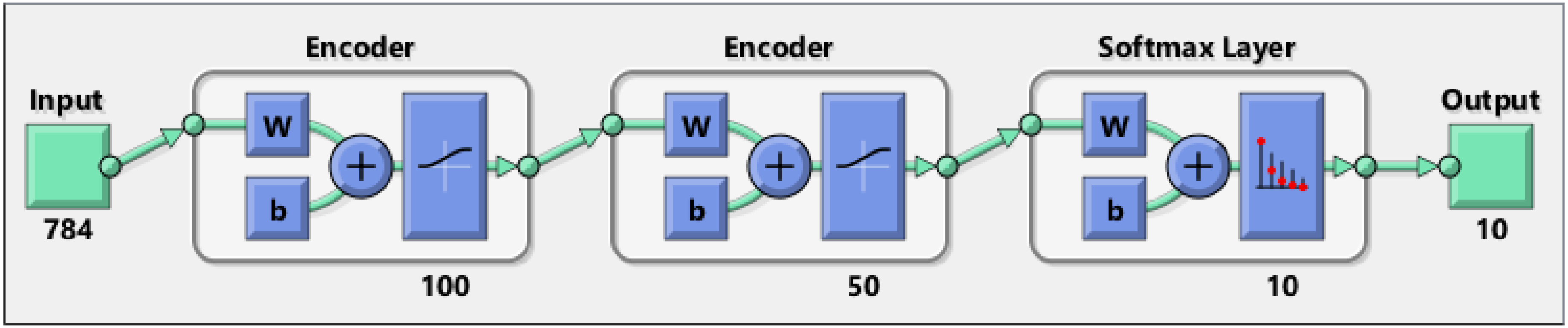
Stacked Autoencoders [4]







Stacked Autoencoders [4]



Fine Tuning Stacked Autoencoders [4]

Fine Tuning

Stacked Denoising Autoencoders [4]

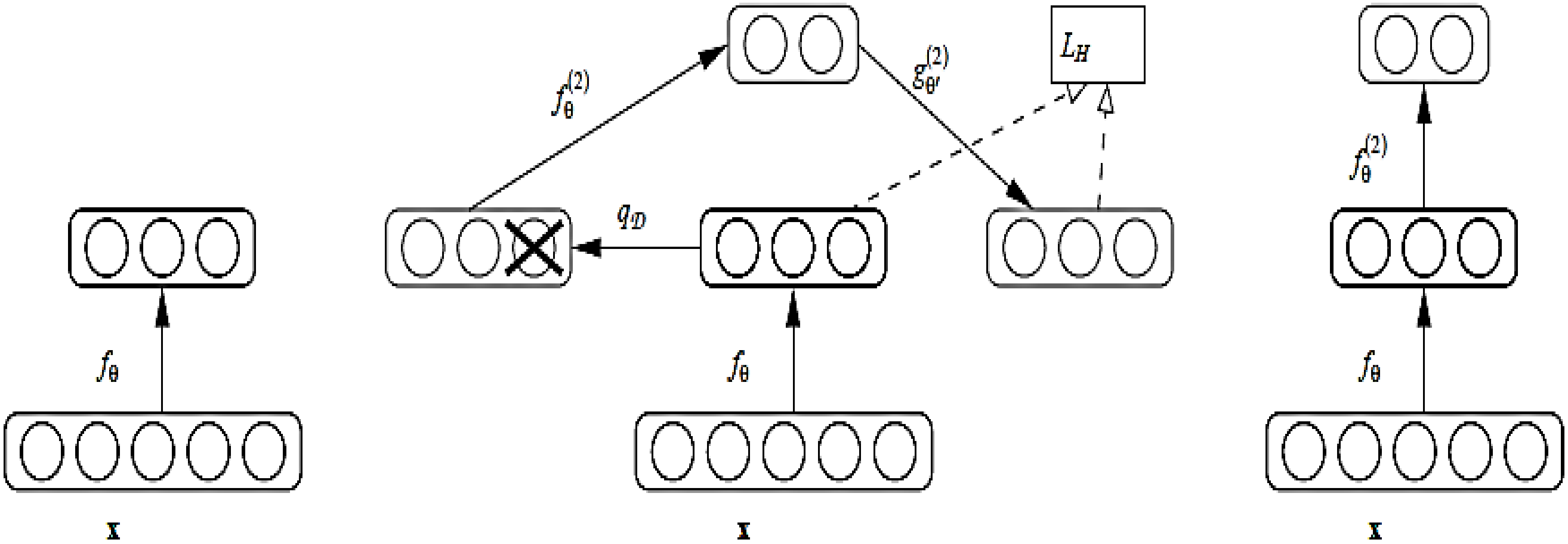
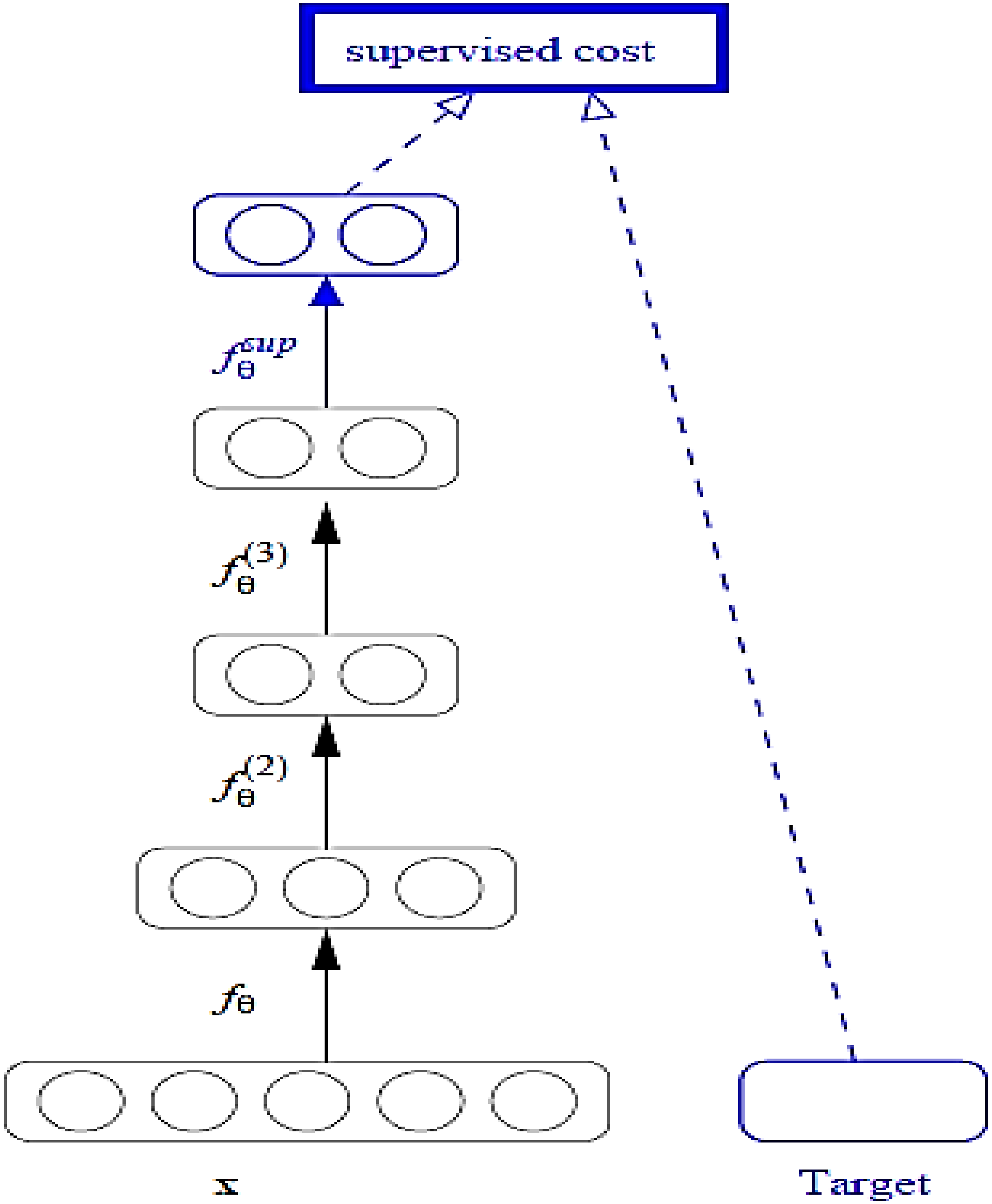


Figure 3: Stacking denoising autoencoders. After training a first level denoising autoencoder (see Figure 1) its learnt encoding function f_θ is used on clean input (left). The resulting representation is used to train a second level denoising autoencoder (middle) to learn a second level encoding function $f_\theta^{(2)}$. From there, the procedure can be repeated (right).

Stacked Denoising Autoencoders [4]



Stacked Denoising Autoencoders [4]

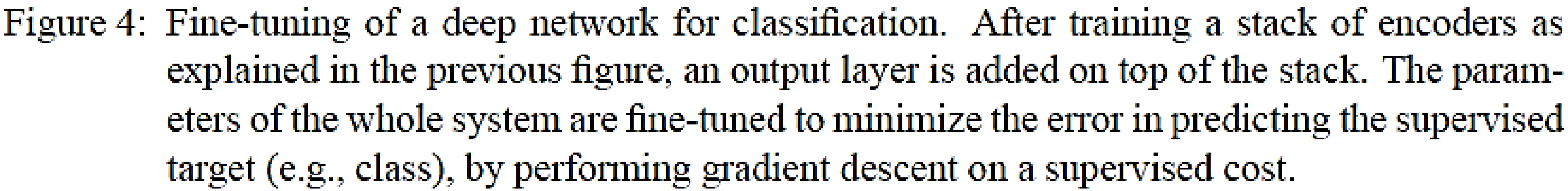


Figure 4: Fine-tuning of a deep network for classification. After training a stack of encoders as explained in the previous figure, an output layer is added on top of the stack. The parameters of the whole system are fine-tuned to minimize the error in predicting the supervised target (e.g., class), by performing gradient descent on a supervised cost.

References

1. https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f
2. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798
3. http://ufldl.stanford.edu/wiki/index.php/Stacked_Autoencoders
4. https://in.mathworks.com/help/deeplearning/examples/train-stacked-autoencoders-for-image-classification.html

Disclaimer

- These slides are not original and have been prepared from various sources for teaching purpose.