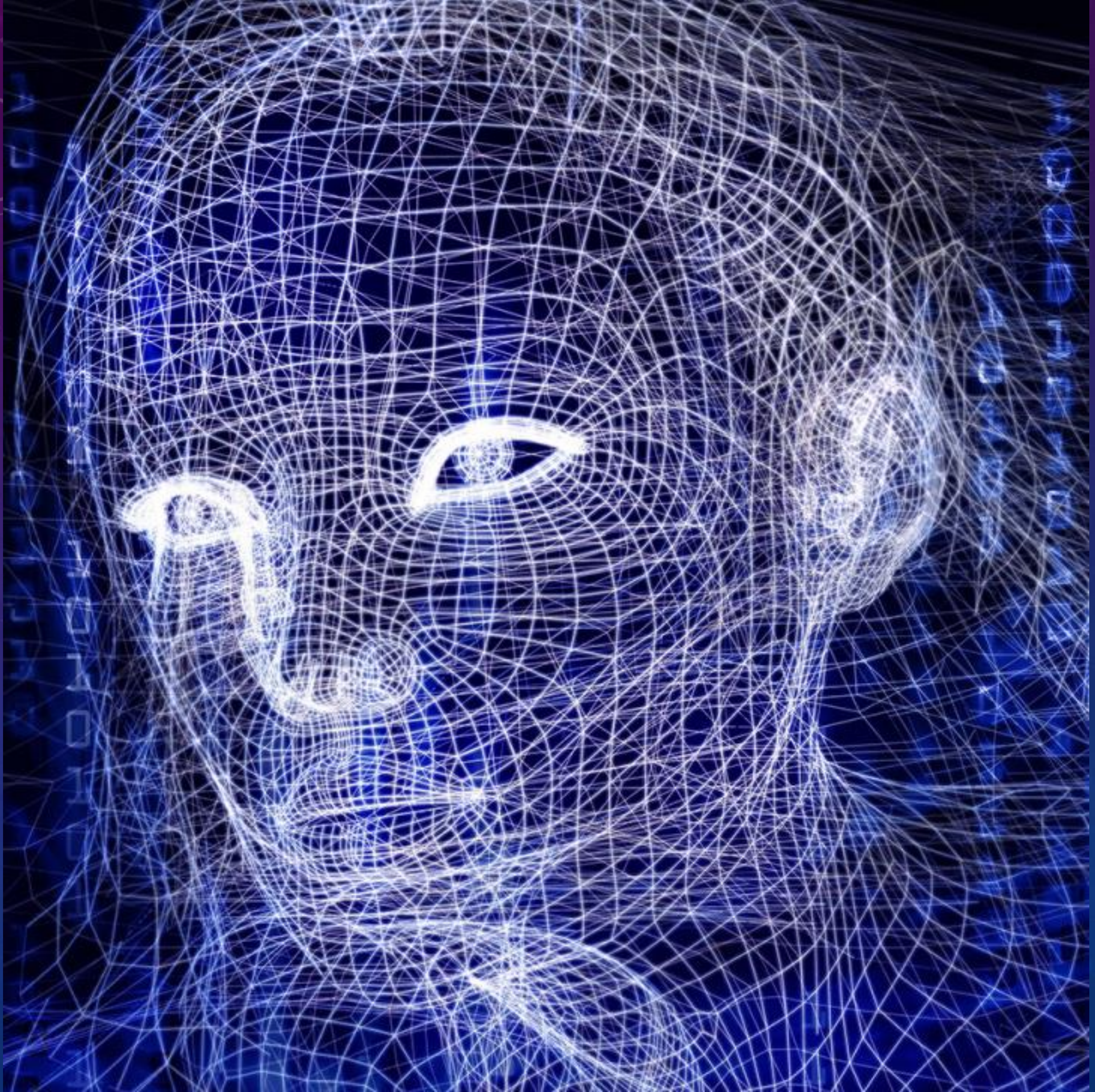
LECTURE SERIES FOR COMPUTER VISION

FRACTIONALLY STRIDED CONVOLUTION AND AUTOENCODER

徐繼聖

Gee-Sern Jison Hsu

National Taiwan University of Science and Technology



Overview

1. Fractionally Strided Convolution
2. Introduction to Autoencoder
3. Variational AutoEncoder (VAE)

$(a.k.a.\ Deconvolution)$

$Fractionally\ strided\ convolutions$

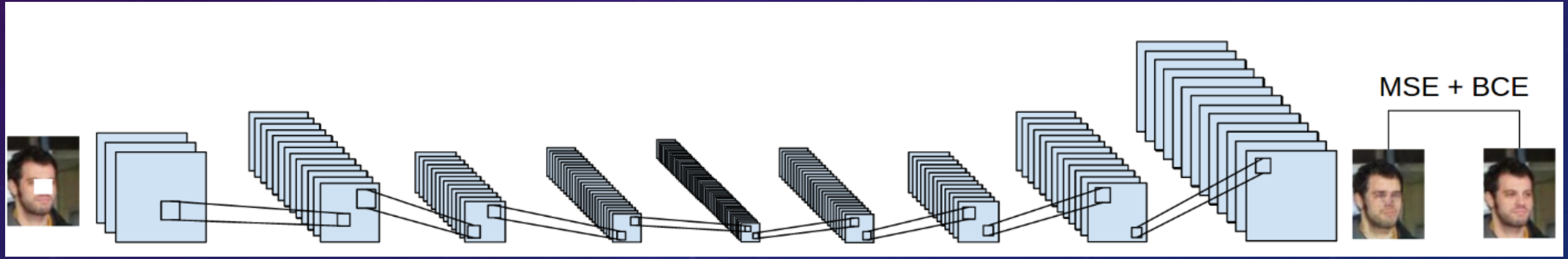
Fractionally Strides Convolutions

(a.k.a. *Deconvolutions* or *Transposed Convolutions*)

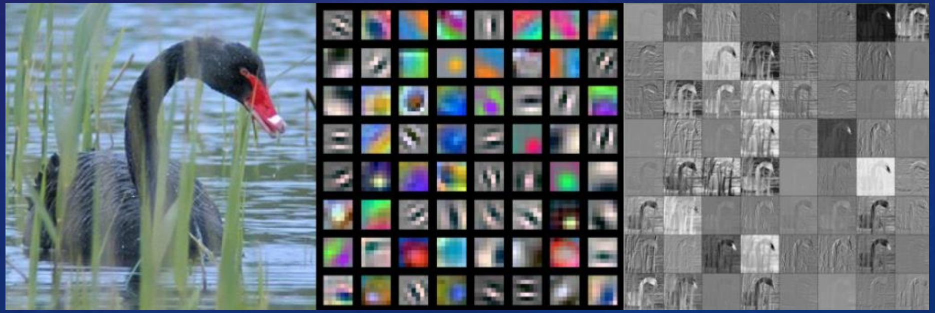
The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution.

– From something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution.

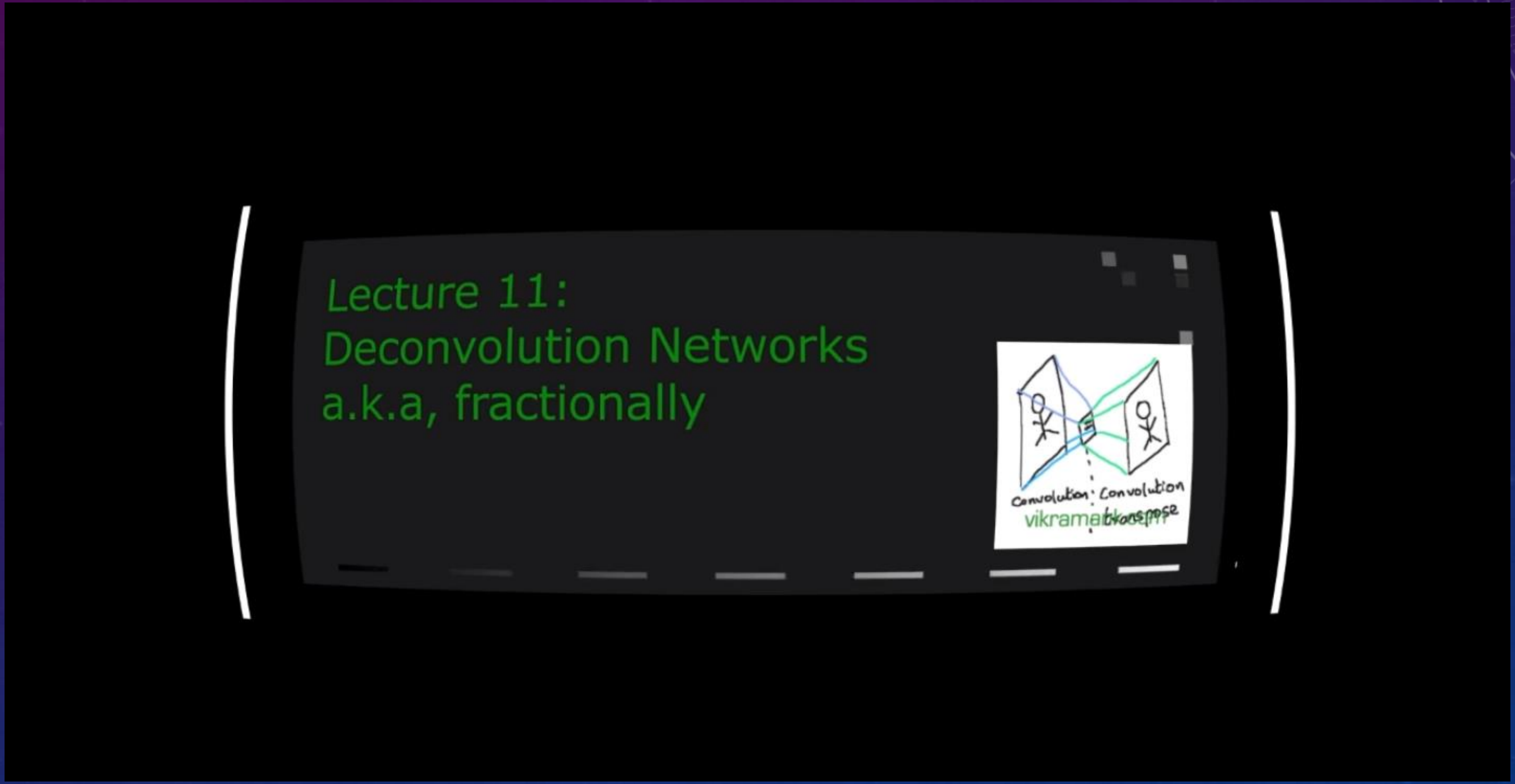
(1) Transformation as the decoding layer of a convolutional autoencoder, e.g., Image synthesis.



(2) Project feature maps to a higher-dimension space , e.g., Feature visualization



Lecture 11: Deconvolutional Networks

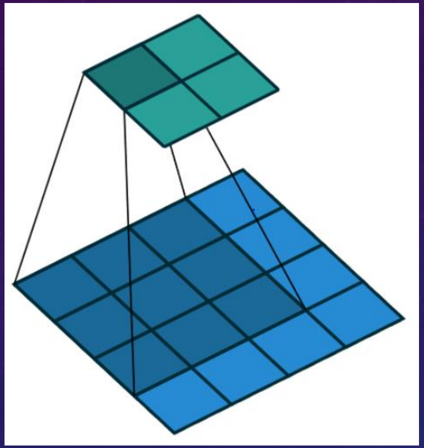


Lecture 11:
Deconvolution Networks
a.k.a, fractionally

https://www.youtube.com/watch?v=8DiqJj5tPlA [19:33]

Convolution as a Matrix Operation

Take for example the convolution represented on the left. If the input and output were to be unrolled into vectors from left to right, top to bottom, the convolution could be represented as a sparse matrix C where the non-zero elements are the elements $w_{i,j}$ of the kernel (with i and j being the row and column of the kernel respectively):

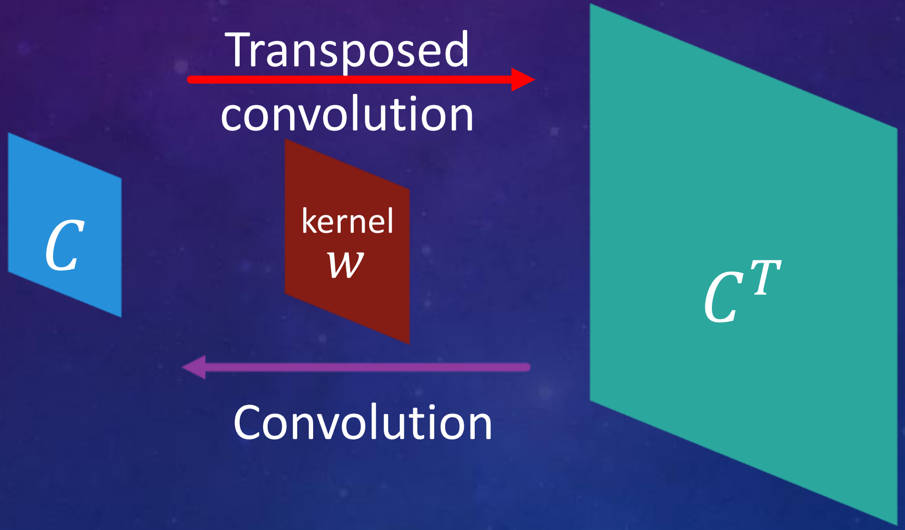


$$\begin{array}{c}
\downarrow \mathit{kernel} \\
\begin{pmatrix}
w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\
0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\
0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2}
\end{pmatrix}
\end{array}$$

This linear operation takes the input matrix flattened as a 16-dimensional vector and produces a 4-dimensional vector that is later reshaped as the 2×2 output matrix.

Fractionally Strided Convolutions

- Since a kernel $\boldsymbol{w}$ defines a convolution, but whether it's a normal convolution or a transposed convolution is determined by how the forward and backward passes are computed.
- We'll proceed somewhat backwards with respect to the convolution arithmetic, deriving the properties of each transposed convolution by referring to the direct convolution with which it shares the kernel, and *defining the equivalent direct convolution*.



C^T : stands for transposed C

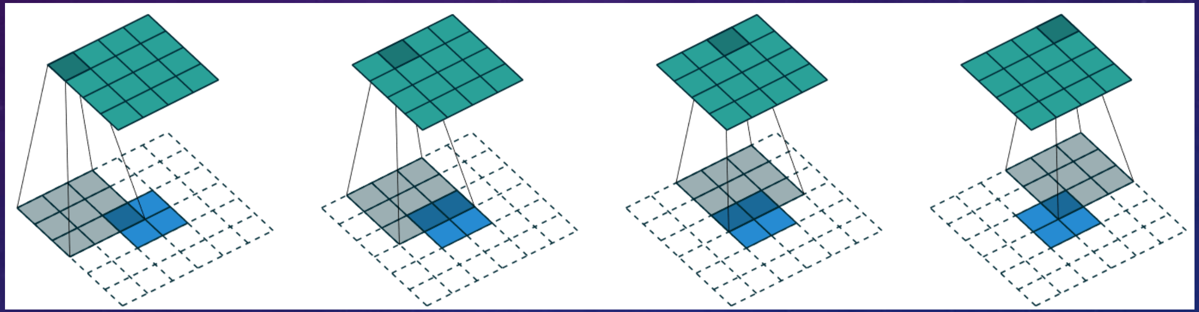
Note that： It is always possible to emulate a transposed convolution with a normal convolution. The disadvantage is that it usually involves adding many columns and rows of zeros to the input, resulting in a *much less efficient implementation*.

Fractionally Strided Convolutions

The transposed convolution can be considered as the operation that allows to recover the *shape* of this initial feature map.

- No zero padding, unit strides, transposed

To understand the logic behind zero padding is to consider the connectivity pattern of the transposed convolution and use it to guide the design of the equivalent convolution. It is possible to determine similar observations for the other elements of the image, giving rise to the following relationship:



The transpose of convolving a 3×3 kernel over a 4×4 input using unit strides

(i.e., $i = 4, k = 3, s = 1$ and $p = 0$)

It is equivalent to convolving a 3×3 kernel over a 2×2 input (blue) padded with a 2×2border of zeros using unit strides.

(i.e., $i' = 2, k' = k, s' = 1$ and $p' = 2$)

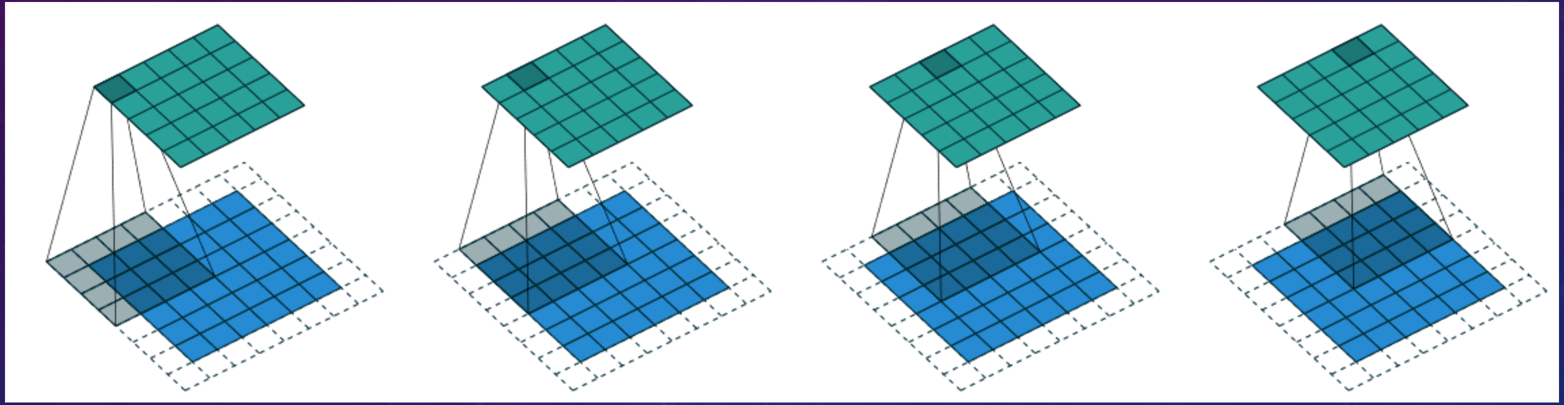
Relationship 8. *A convolution described by $s = 1, p = 0$ and k has an associated transposed convolution Described by $k' = k, s' = s$ and $p' = k - 1$ and its output size is*

$$o' = i' + (k - 1)$$

Fractionally Strided Convolutions

- Zero padding, unit strides, transposed

Knowing that the transpose of a non-padded convolution is equivalent to convolving a zero padded input, it would be reasonable to suppose that the trans-pose of a zero padded convolution is equivalent to convolving an input padded with *less* zeros.



The transpose of convolving a 4×4 kernel over a 5×5 input (green) padded with a 2×2 border of zeros using unit strides (i.e., $i = 5$, $k = 4$, $s = 1$ and $p = 2$)

It is equivalent to convolving a 4×4 kernel over a 6×6 input padded with a 1×1 border of zeros using unit strides (i.e., $i' = 6$, $k' = k$, $s' = 1$ and $p' = 1$)

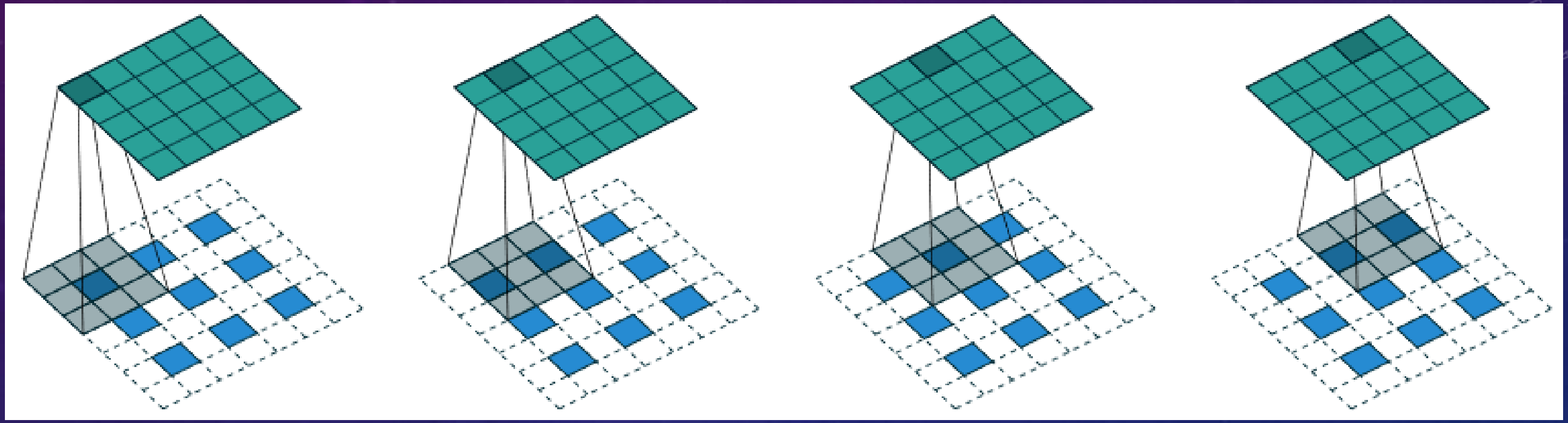
Relationship 9. *A convolution described by* $s = 1,\ k,$ *and* p *has an associated transposed convolution described by* $k' = k, s' = s$ *and* $p' = k - p - 1$ *and its output size is*

$$o' = i' + (k - 1) - 2p$$

Fractionally Strided Convolutions

- Zero padding, non-unit strides, transposed

When the convolution's input size i is such that $i + 2p - k$ is a multiple of s, the analysis can extended to the zero padded case by combining **Relationship 9** and **Relationship 12**:



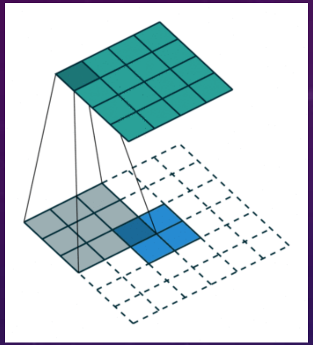
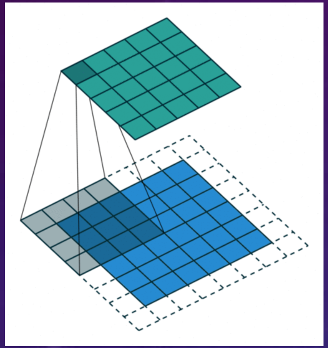
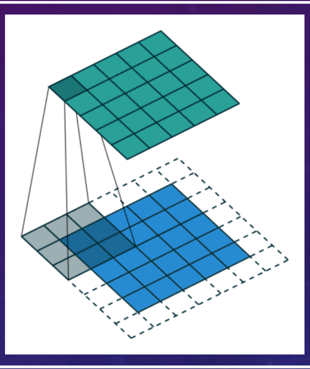
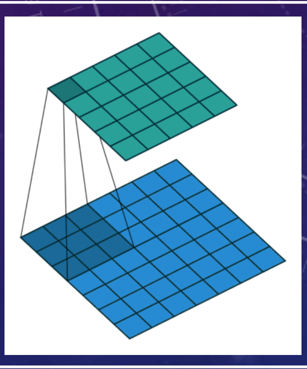
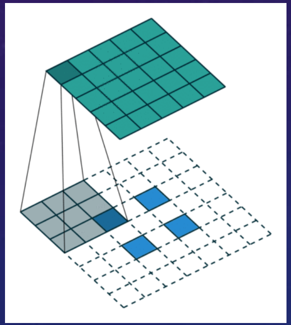
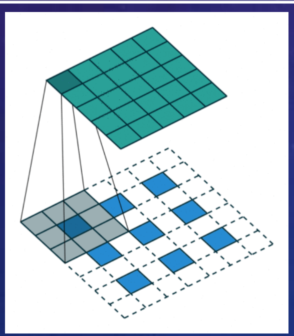
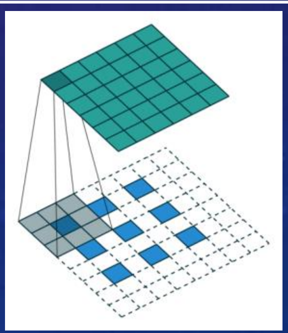
The transpose of convolving a 3×3 kernel over a 5×5 input padded with a 1×1 border of zeros using 2×2 strides
(i.e., $i = 5$, $k = 3$, $s = 2$ and $p = 1$)
It is equivalent to convolving a 3×3 kernel over a 3×3 input (with 1 zero inserted between inputs) padded with a 1×1 border of zeros using unit strides
(i.e., $i' = 3$, $\widetilde{i'} = 5$, $k' = k$, $s' = 1$ and $p' = 1$)

Relationship 13. $A\ convolution\ described\ k,\ s\ and\ p\ whose\ input\ size\ i\ is\ such\ that\ i + 2p - k\ is\ mutiple\ of\ s$ $has\ an\ associated\ transposed\ convolution\ described\ by\ \widetilde{i'}, k' = k, s' = 1\ and\ p' = k - p - 1, where\ \widetilde{i'}\ is\ the$ $Size\ of\ the\ streched\ input\ obtained\ by\ adding\ s - 1\ zeros\ between\ each\ input\ unit, and\ tis\ ouput\ size\ is$

$$o' = s(i' - 1) + k - 2p$$

Transposed Convolution Animations

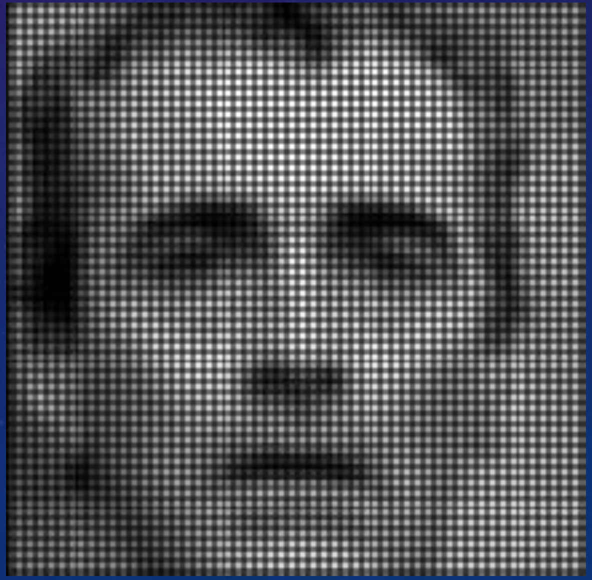
N.B.: Blue maps are inputs, and cyan maps are outputs.

			
No padding, no strides, transposed	Arbitrary padding, no strides, transposed	Half padding, no strides, transposed	Full padding, no strides, transposed
			
No padding, strides, transposed	Padding, strides, transposed	Padding, strides, transposed (odd)	

https://github.com/vdumoulin/conv_arithmetic

Example 3-1: Convolution and Fractionally Strided Convolution

- Please download the “3-1_Convolution and Deconvolution.zip” from the Moodle.
- Upload the “sample.jpg” to the Google Colab.
- Run the codes and get the output images through the convolution and fractionally strided convolution.



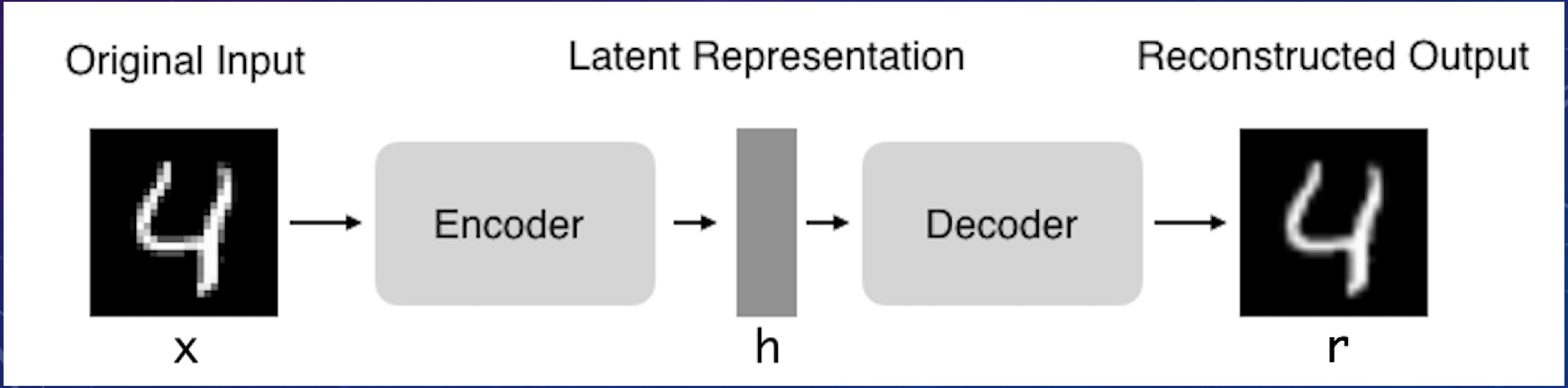
Exercise 3-1: Convolution and Fractionally Strided Convolution

- Please download the “3-1_Convolution and Deconvolution.zip” from the Moodle.
- Upload the “sample.jpg” to the Google Colab.
- Follow the example code and design a convolution layer and a fractionally strided convolution layer.
- Run the codes and get the output images through the designed convolution and fractionally strided convolution.

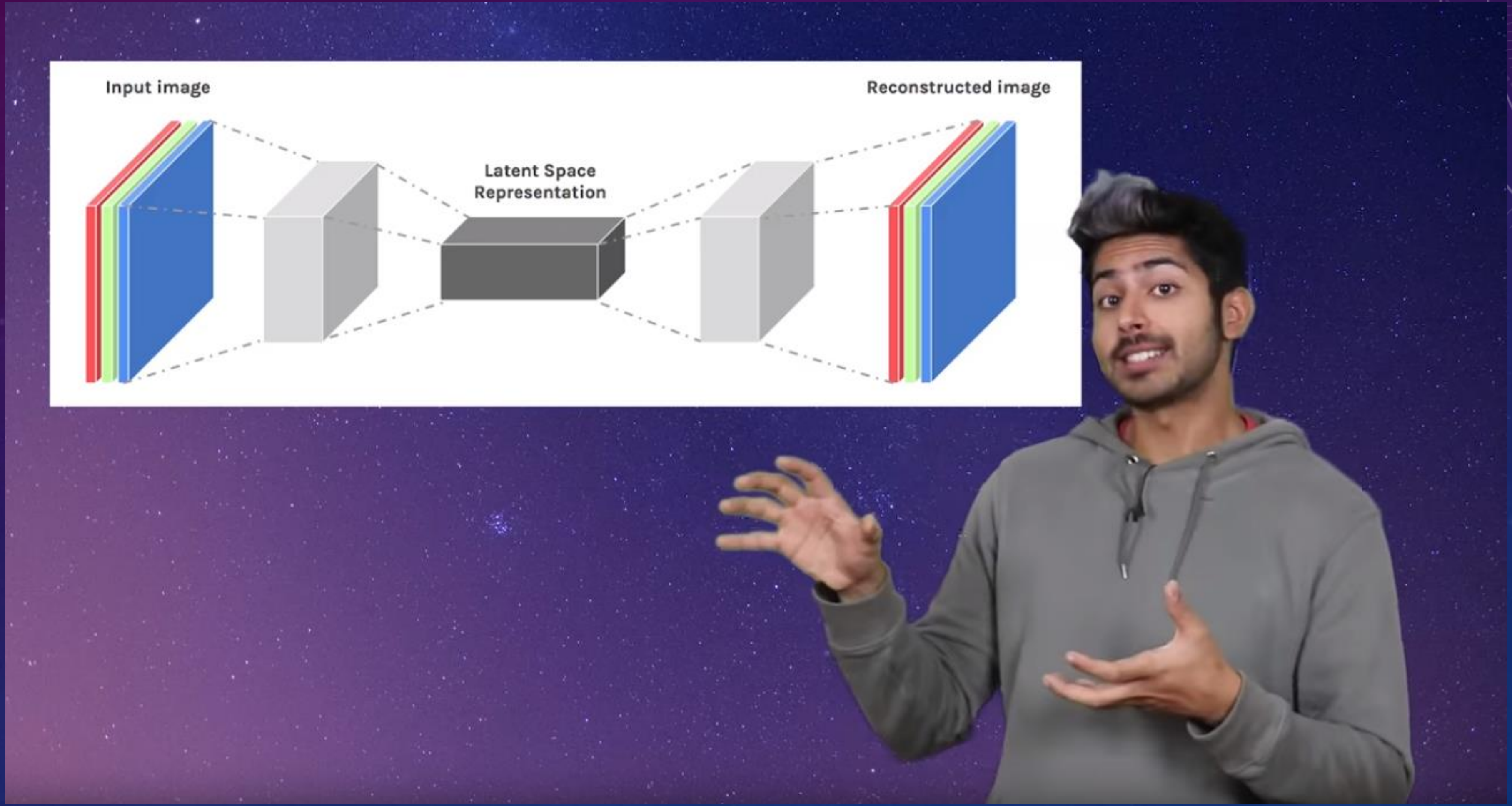
Please write down your results and codes in MS Word, then upload to the Moodle.

Autoencoders (AE)

- 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.
  - **Encoder:** Compresses the input into a latent-space representation. It can be represented by an encoding function $h = f(x)$.
  - **Decoder:** Reconstruct the input from the latent space representation. It can be represented by a decoding function $r = g(h)$.



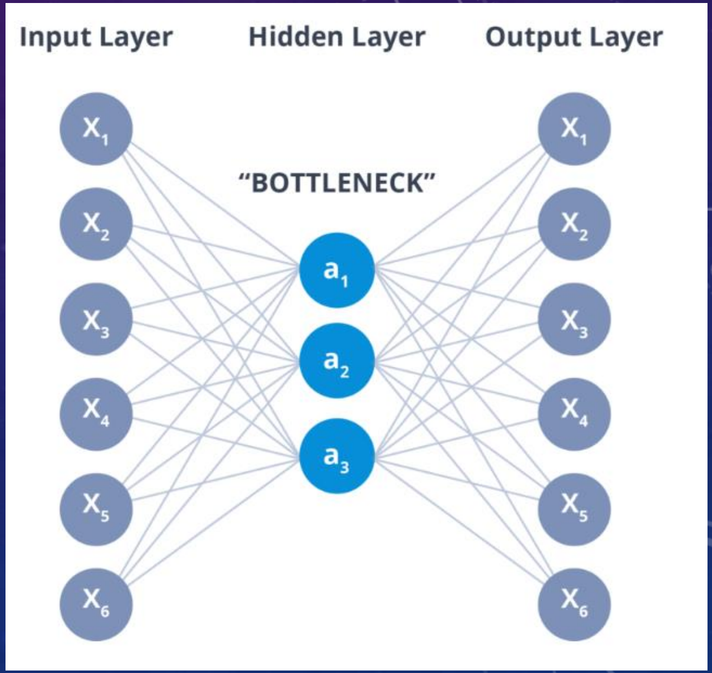
Autoencoder Explained



https://youtu.be/H1AllrJ-_30 [8:41]

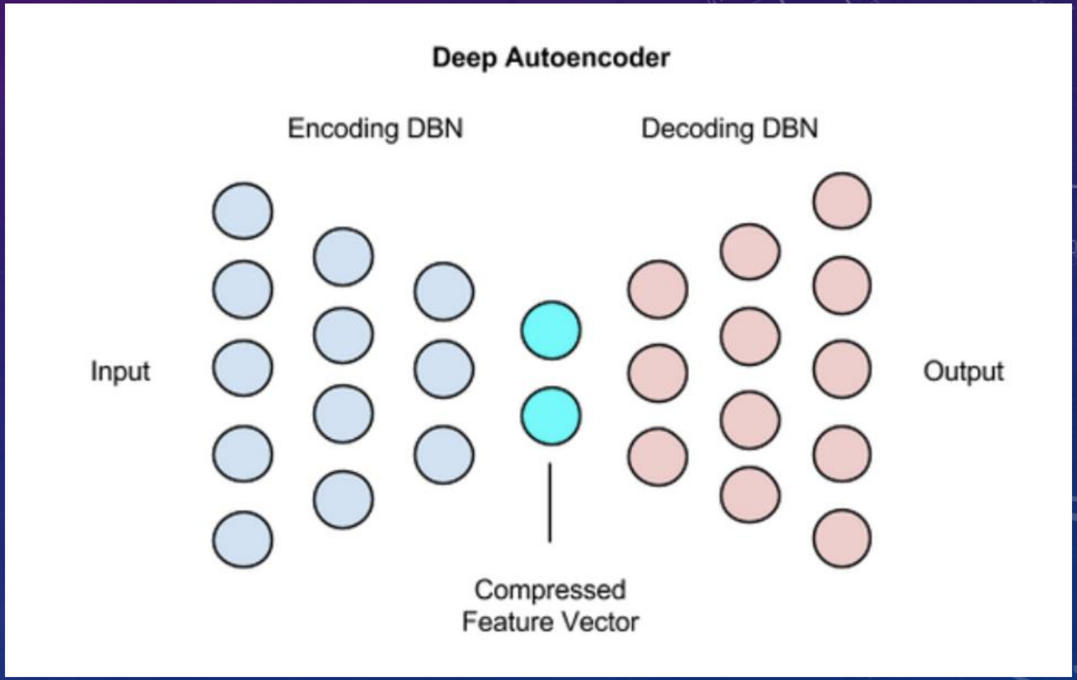
Bottleneck

- The layer between the encoder and decoder, ie. the code is also known as **Bottleneck**. This is a well-designed approach to decide which aspects of observed data are relevant information and what aspects can be discarded.
- It does this by balancing two criteria :
  - Compactness of representation, measured as the compressibility.
  - It retains some behaviourally relevant variables from the input.


Input Layer
Hidden Layer
Output Layer
"BOTTLENECK"
X_1 X_2 X_3 X_4 X_5 X_6
a_1 a_2 a_3
X_1 X_2 X_3 X_4 X_5 X_6

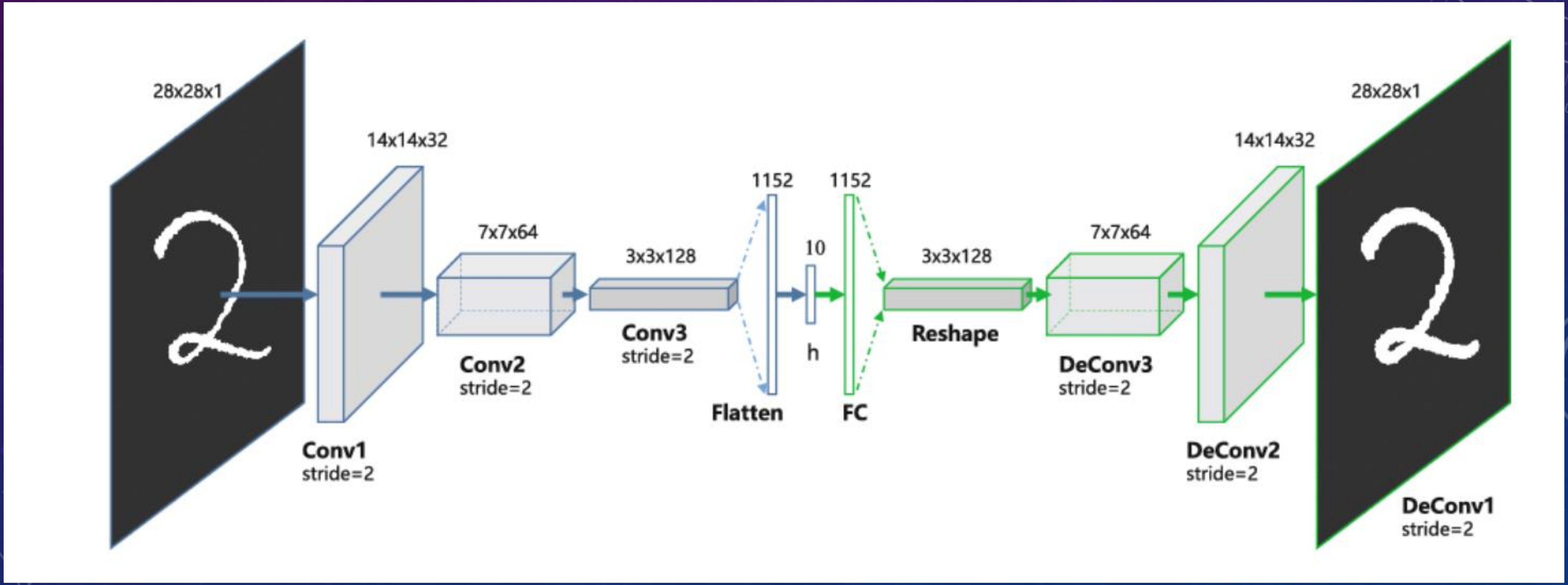
Deep Autoencoders

- The extension of the simple Autoencoder is the **Deep Autoencoder**. Deeper layers of the Deep Autoencoder tend to learn even higher-order features.
- A **deep autoencoder** is composed of two, symmetrical deep-belief networks-
  - First four or five shallow layers representing the encoding half of the net.
  - The second set of four or five layers that make up the decoding half.



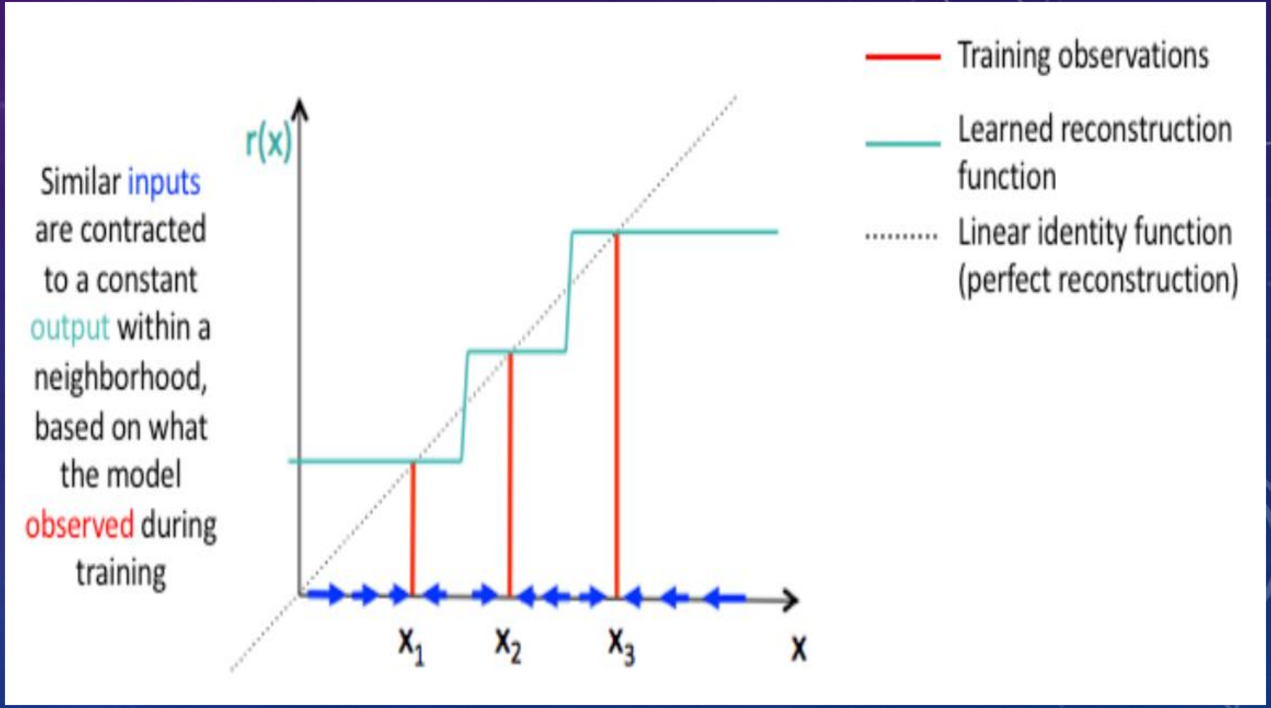
Convolution Autoencoders

- Convolutional Autoencoders use the convolution operator to exploit this observation. They learn to encode the input in a set of simple signals and then try to reconstruct the input from them, modify the geometry or the reflectance of the image.



Contractive Autoencoders

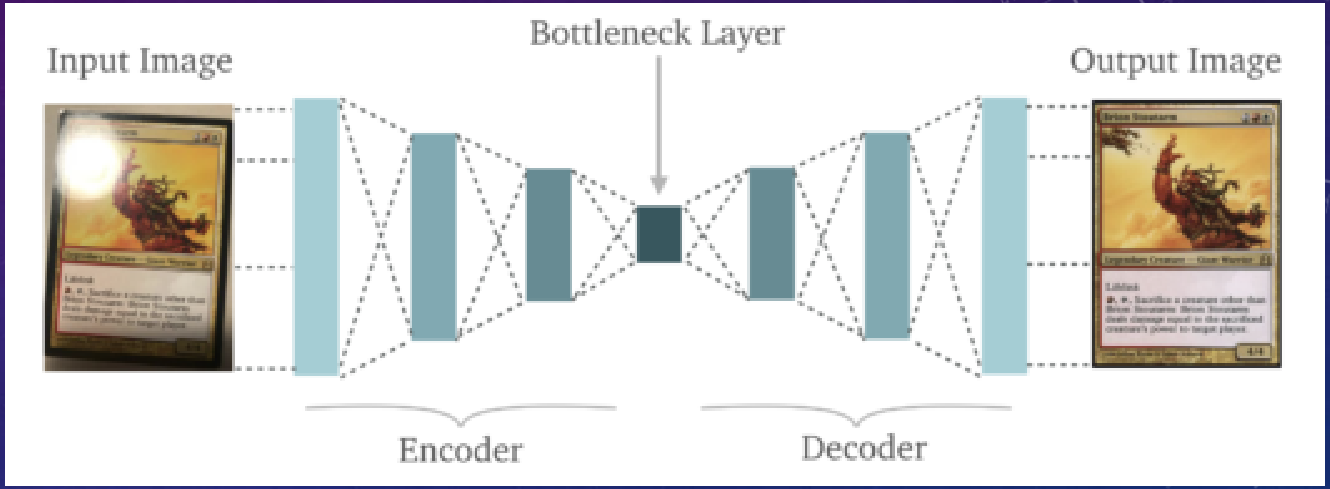
- A **contractive autoencoder** is an unsupervised deep learning technique that helps a neural network encode unlabeled training data.
- This is accomplished by constructing a **loss term** which penalizes large derivatives of our hidden layer activations with respect to the input training examples, **essentially penalizing** instances where a small change in the input leads to a large change in the encoding space.


Similar inputs are contracted to a constant output within a neighborhood, based on what the model observed during training
r(x)
x
x_1
x_2
x_3
Training observations
Learned reconstruction function
Linear identity function (perfect reconstruction)

Applications of Autoencoders

Feature variation

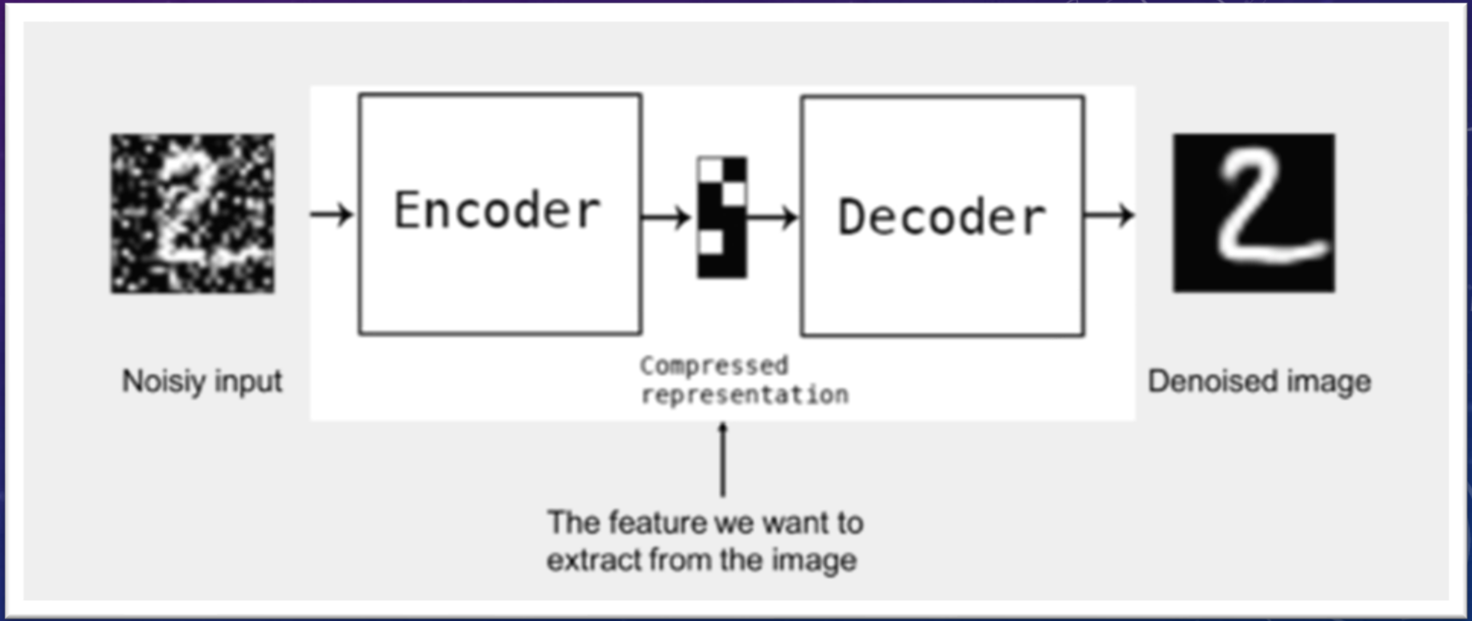
It extracts only the required features of an image and generates the output by removing any noise or unnecessary interruption.



Applications of Autoencoders

- Denoising Image

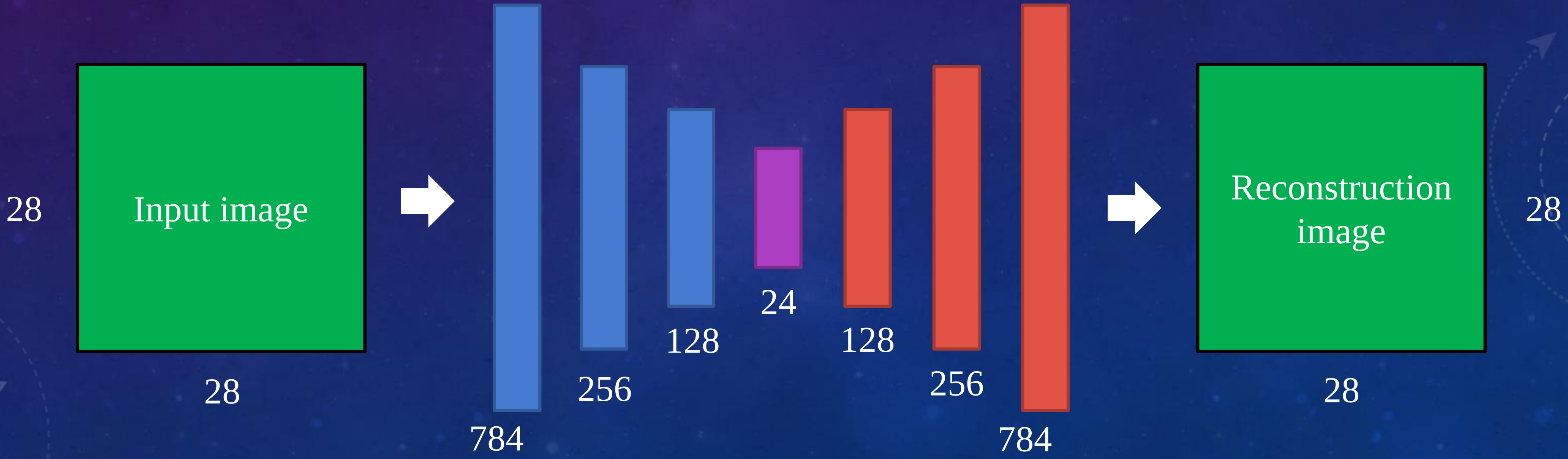
  The input seen by the autoencoder is not the raw input but a stochastically corrupted version. A denoising autoencoder is thus trained to reconstruct the original input from the noisy version.


Encoder
Decoder
Noisiy input
Compressed representation
Denoised image
The feature we want to extract from the image

Exercise 3.2 : Autoencoder

- Please download the "exercise4.1_Autoencoder.ipynb" on Moodle.
  - Train the autoencoder and compare the images that reconstruct from the different epoch.
  - Change the encoder and decoder to the below architecture and compare the difference.

Please copy your results and code and paste to a MS Word, then upload to Moodle.



Example 3.2 : Autoencoder

Define the hyper-parameter and load the training data

```
num_epochs = 10
batch_size = 32
learning_rate = 1e-3

img_transform = transforms.Compose([transforms.ToTensor()])

dataset = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True,
transform=img_transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=False)
```

Define the hyperparameter

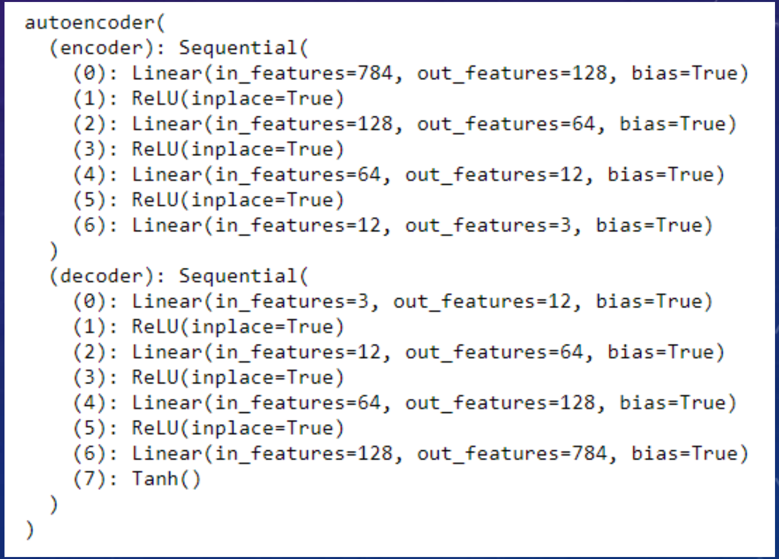
Download the Mnist dataset to the folder './data'

Example 3.2 : Autoencoder

Define the model architecture

```
class autoencoder(nn.Module):
  def __init__(self):
    super(autoencoder, self).__init__()
    self.encoder = nn.Sequential(
      nn.Linear(28 * 28, 128),
      nn.ReLU(True),
      nn.Linear(128, 64),
      nn.ReLU(True),
      nn.Linear(64, 12),
      nn.ReLU(True),
      nn.Linear(12, 3))
    self.decoder = nn.Sequential(
      nn.Linear(3, 12),
      nn.ReLU(True),
      nn.Linear(12, 64),
      nn.ReLU(True),
      nn.Linear(64, 128),
      nn.ReLU(True), nn.Linear(128, 28 * 28), nn.Tanh())
```

```
def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x
```



```
autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=64, out_features=12, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=12, out_features=3, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=3, out_features=12, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=12, out_features=64, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=64, out_features=128, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=128, out_features=784, bias=True)
    (7): Tanh()
  )
)
```

Example 3.2 : Autoencoder

Define the model architecture

```
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128),
            nn.ReLU(True),
            nn.Linear(128, 64),
            nn.ReLU(True),
            nn.Linear(64, 12),
            nn.ReLU(True),
            nn.Linear(12, 3))
```

Define the encoder

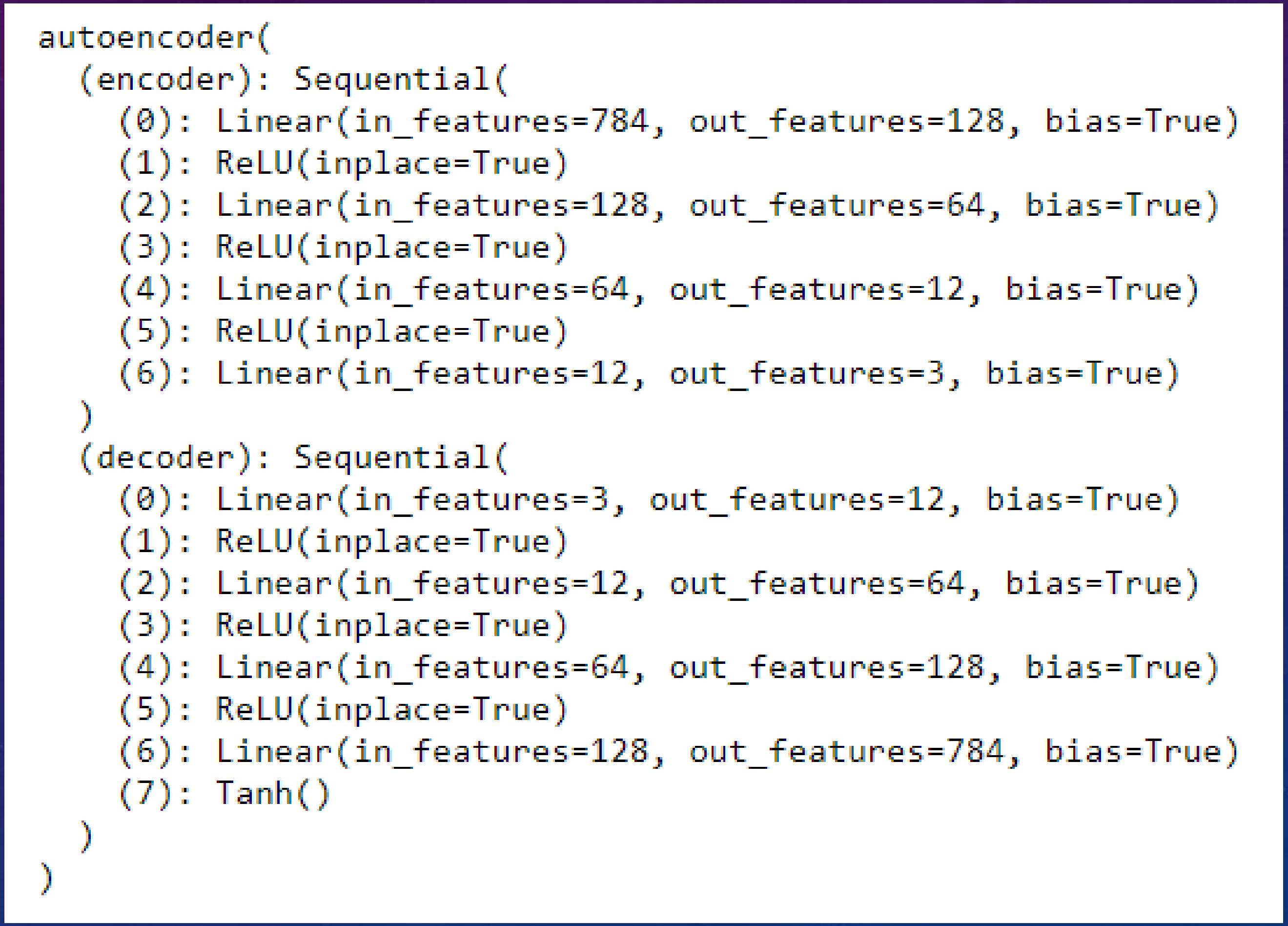
Define the decoder

```
self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.ReLU(True),
            nn.Linear(12, 64),
            nn.ReLU(True),
            nn.Linear(64, 128),
            nn.ReLU(True),
            nn.Linear(128, 28 * 28),
nn.Tanh())

def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Example 3.2 : Autoencoder

Define the model architecture



```
autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=64, out_features=12, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=12, out_features=3, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=3, out_features=12, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=12, out_features=64, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=64, out_features=128, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=128, out_features=784, bias=True)
    (7): Tanh()
  )
)
```

Example 3.2 : Autoencoder

$\longrightarrow$ Use the Mean Square Error as the loss function

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=1e-5)
```

```
for epoch in range(num_epochs):
    for data in dataloader:
        img, _ = data
        img = img.view(img.size(0), -1)
        img = Variable(img).cuda()
        output = model(img)
        loss = criterion(output, img)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print('epoch [{}/{}], loss:{:.4f}'.format(epoch + 1, num_epochs, loss.item()))
```

$\longrightarrow$ Flatten the input images

$\longrightarrow$ Compute the loss

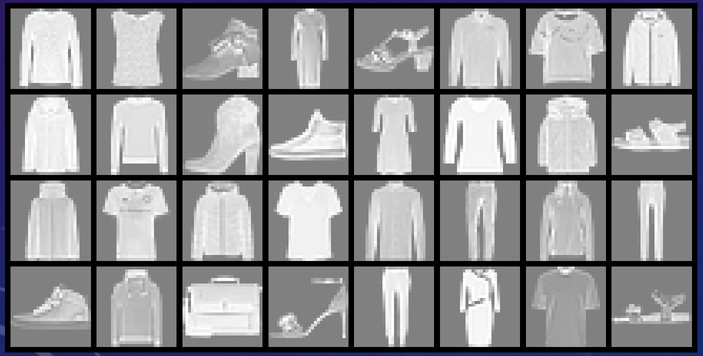
Example 3.2 : Autoencoder

```
if epoch % 1 == 0:
    img = to_img(img.cpu().data)
    save_image(img, ‘./AE_img/input_{}.png'.format(epoch))
    pic = to_img(output.cpu().data)
    save_image(pic, './ AE _img/output_{}.png'.format(epoch))
```

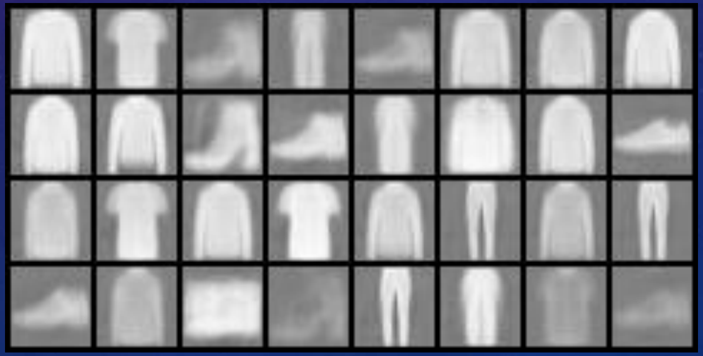
→ Save the input images

→ Save the reconstruction images

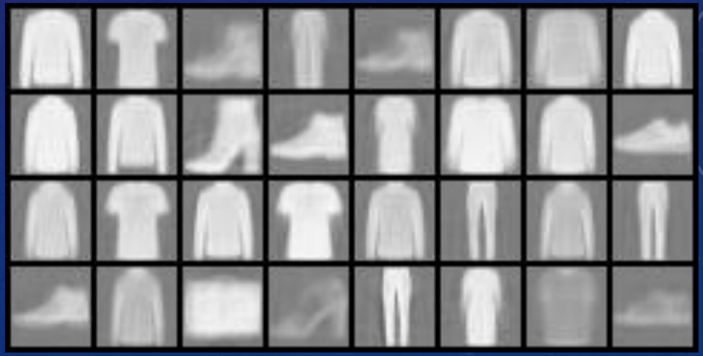
Input images



Epoch 1 : reconstruction images



Epoch 10 : reconstruction images



Lecture 13 | Generative Models

https://youtu.be/5WoItGTWV54 [1:17:40]

Lecture 13 | Generative Models

- OUTLINE:
- 1:12 - Supervised learning
- 5:33 - Generative models
- 11:00 - PixelRNN and PixelCNN
- 19:36 - Traditional and variational autoencoders (VAEs)
- 50:00 - Generative adversarial networks (GANs)

Variational Autoencoders

https://youtu.be/9zKuYvjFFS8 [15:05]

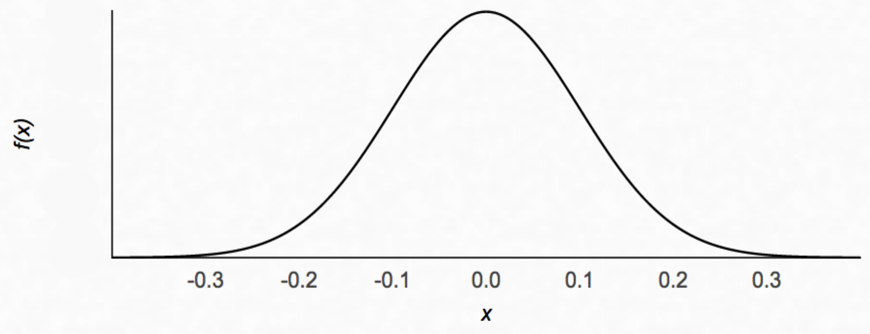
Variational Autoencoders

- Variational autoencoders use gaussian models to generate images.
- There are some backgrounds we need to know:
  - Gaussian distribution
  - Autoencoders
  - KL-divergence

https://jhui.github.io/2017/03/06/Variational-autoencoders/

Gaussian Distribution

- In the following diagram, we assume the probability of X equal to a certain value x, $p(X=x)$, follows a gaussian distribution:



$\mu = E(X) = 0 \quad \sigma = SD(X) = 0.1 \quad \sigma^2 = Var(X) = 0.01$

$$\text{Probability density function (PDF)} = p(X = x) = f(x) = \frac{e^{-(x-\mu)^2/(2\sigma^2)}}{\sigma\sqrt{2\pi}}$$

- We can sample data using the PDF above. We use the following notation for sample data using a gaussian distribution with mean μ and standard deviation σ.

$$x \sim N(\mu, \sigma)$$

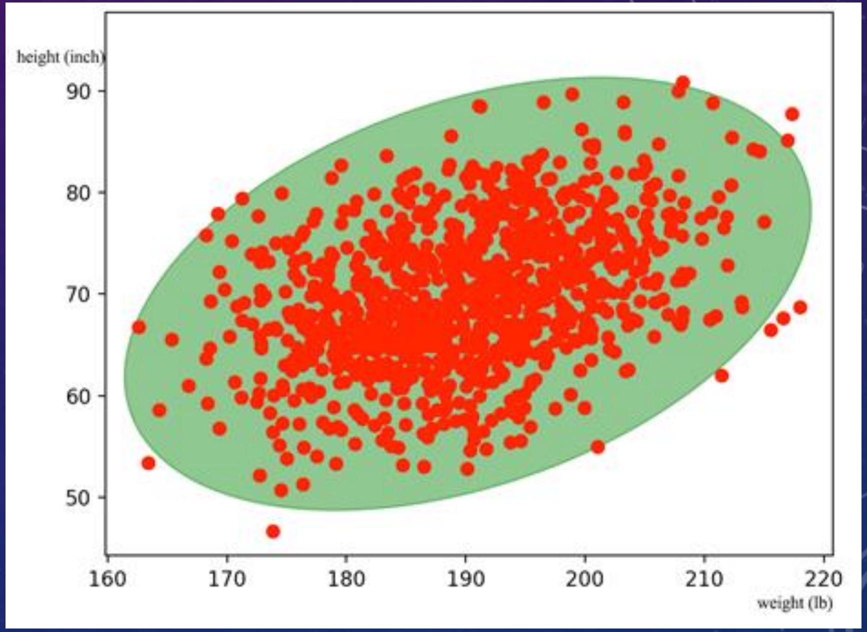
Gaussian Distribution

- In the example above, mean: μ=0, standard deviation: σ=0.1:

  In many real world examples, the data sample distribution follows a gaussian distribution.

- Now, we generalize it with multiple variables. For example, we want to model the relationship between the body height and the body weight for San Francisco residents.

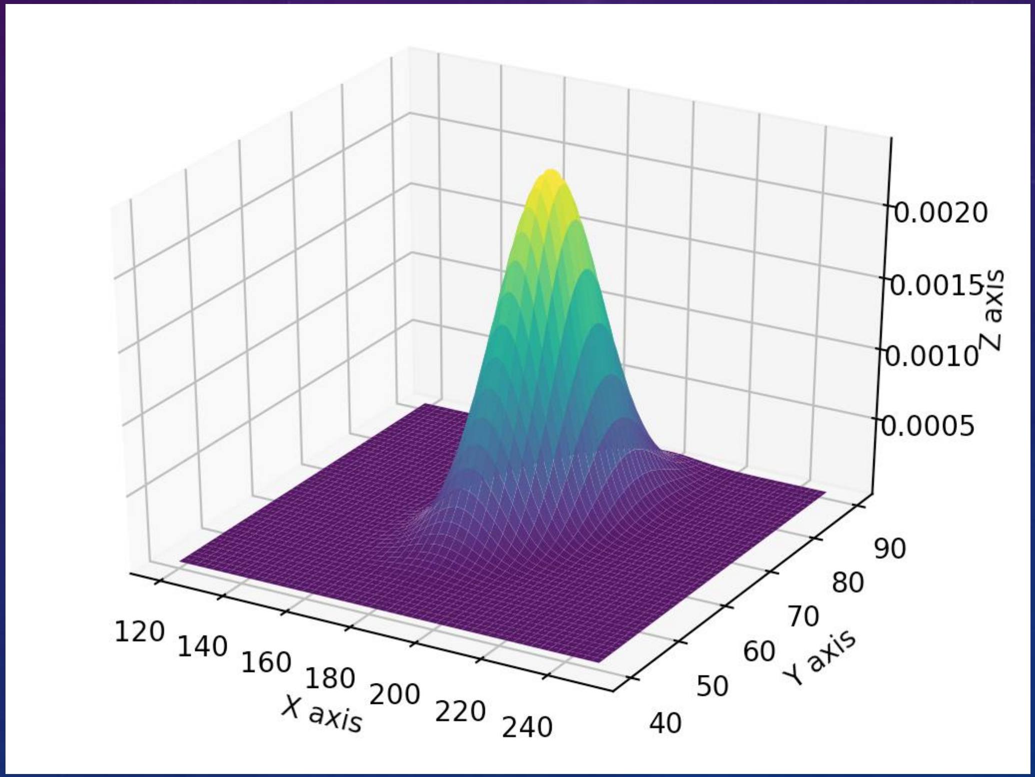
- We collect the information from 1000 adult residents and plot the data below with each red dot represents 1 person:



Gaussian Distribution

- We can plot the corresponding probability density function in 3D:

PDF = probability(height = h, weight = w)



Gaussian Distribution

We can model such a probability density function using a gaussian distribution function. The PDF with p variables is:

$$z = \begin{pmatrix} \mathrm{X}_1 \\ \vdots \\ \mathrm{X}_p \end{pmatrix}, \ f(x) = \frac{1}{(2\pi)^{p/2}|\sum|^{1/2}} \exp\{-\frac{1}{2}(x-\mu)^T \sum^{-1}(x-\mu)\}$$

with covariance matrix $\sum$:

$$\sum = \begin{pmatrix} E[(x_1-\mu_1)(x_1-\mu_1)] & E[(x_1-\mu_1)(x_2-\mu_2)] & \dots & E[(x_1-\mu_1)(x_p-\mu_p)] \\ E[(x_2-\mu_2)(x_1-\mu_1)] & E[(x_2-\mu_2)(x_2-\mu_2)] & \dots & E[(x_2-\mu_2)(x_p-\mu_p)] \\ \vdots & \vdots & \ddots & \vdots \\ E[(x_p-\mu_p)(x_1-\mu_1)] & E[(x_p-\mu_p)(x_2-\mu_2)] & \dots & E[(x_n-\mu_p)(x_p-\mu_p)] \end{pmatrix}$$

Gaussian Distribution

- The notation for sampling x is:

$$x \sim \mathcal{N}\left(\mu, \sum\right)$$

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_p \end{pmatrix} \sim \mathcal{N}\left(\mu, \sum\right) = \mathcal{N}\left(\begin{pmatrix} \mu_1 \\ \vdots \\ \mu_p \end{pmatrix}, \sum\right)$$

- Let's go back to our weight and height example to illustrate it

$$x = \begin{pmatrix} weight \\ height \end{pmatrix}$$

- From the data, we compute the mean weight is 190 lb and mean height is 70 inches:

$$\mu = \begin{pmatrix} 190 \\ 70 \end{pmatrix}$$

Gaussian Distribution

- For the covariance matrix $\sum$, here, we illustrate how to compute one of the element E_{21}

$$E_{21} = E[(x_2 - \mu_2)(x_1 - \mu_1)] = E\left[\left(x_{height} - 70\right)\left(x_{weight} - 190\right)\right]$$

- which E is the expected value. Let say we have only 2 datapoints (200 lb, 80 inches) and (180 lb, 60 inches)

$$E_{21} = E\left[\left(x_{height} - 70\right)\left(x_{weight} - 190\right)\right]$$
$$= \frac{1}{2}((80 - 70) \times (200 - 190) + (60 - 70) \times (180 - 190))$$

- After computing all 1000 data, here is the value of $\sum$:

$$\sum = \begin{pmatrix} 100 & 25 \\ 25 & 50 \end{pmatrix}, \qquad x \sim N\left(\begin{pmatrix} 190 \\ 70 \end{pmatrix}, \begin{pmatrix} 100 \\ 50 \end{pmatrix}\right)$$

Gaussian Distribution

- E_{21}measures the co-relationship between variables x_2 and x_1. Positive values means both are positively related. With not surprise, E_{21} is positive because weight increases with height. If two variables are independent of each other, it should be 0 like:

$$\sum = \begin{pmatrix} 100 & 0 \\ 0 & 50 \end{pmatrix}$$

- We will simplify the gaussian distribution notation here as:

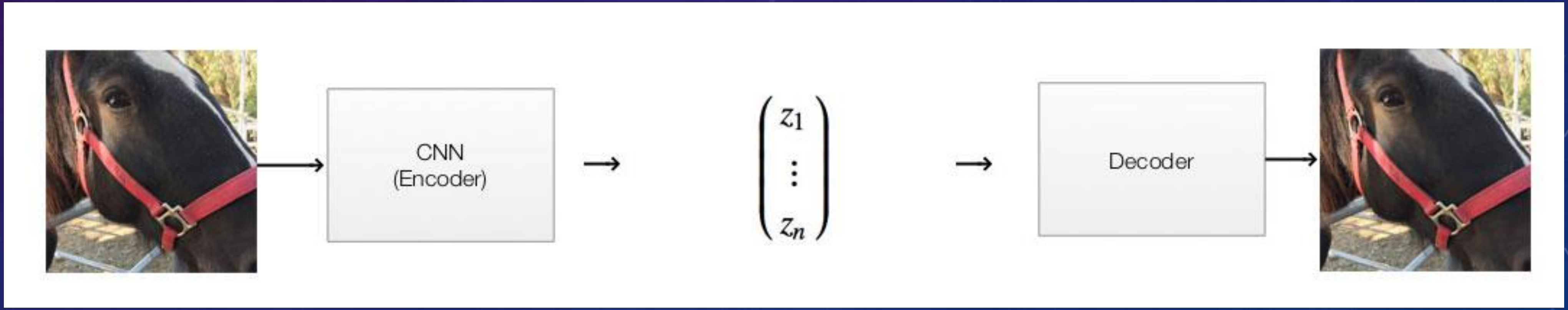
$$x \sim N\left(\begin{pmatrix} 190 \\ 70 \end{pmatrix}, \begin{pmatrix} 100 \\ 50 \end{pmatrix}\right)$$

Autoencoders

- In an autoencoders, we use a deep network to map the input image (for example 256x256 pixels = 256x256 = 65536 dimension) to a lower dimension latent variables (latent vector say 100-D vector: (x_1, x_2,$\cdots$ x_{100})).
- We use another deep network to decode the latent variables to restore the image. We train both encoder and decoder network to minimize the difference between the original image and the decoded image.
- By forcing the image to a lower dimension, we hope the network learns to encode the image by extracting core features.

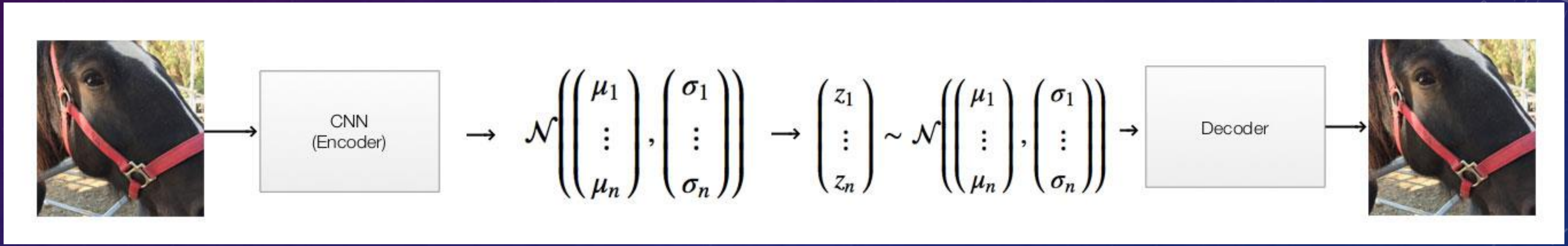
Autoencoders

- For example, we enter a 256x256 image to the encoder, we use a CNN to encode the image to 20-D latent variables (x_1, x_2, ..., x_{20})=(0.1, 0, ..., −0.05).
- We use another network to decode the latent variables into a 256x256 image.
- We use backpropagation with cost function comparing the decoded and input image to train both encoding and decoding network



Variational Autoencoders (VAEs)

- For VAEs, we replace the middle part with a stochastic model using a gaussian distribution. Let's get into an example to demonstrate the flow:



CNN (Encoder)
$\mathcal{N}\left(\begin{pmatrix}\mu_1\\ \vdots \\ \mu_n\end{pmatrix}, \begin{pmatrix}\sigma_1\\ \vdots \\ \sigma_n\end{pmatrix}\right) \rightarrow \begin{pmatrix}z_1\\ \vdots \\ z_n\end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix}\mu_1\\ \vdots \\ \mu_n\end{pmatrix}, \begin{pmatrix}\sigma_1\\ \vdots \\ \sigma_n\end{pmatrix}\right)$
Decoder

- For a variation autoencoder, we replace the middle part with 2 separate steps. VAE does not generate the latent vector directly.

Variational Autoencoders (VAEs)

- It generates 100 Gaussian distributions each represented by a mean (μ_i) and a standard deviation (σ_i).
- Then it samples a latent vector, say (0.1, 0.03, …, -0.01), from these distributions.
- For example, if element x_i of the latent vector has μ_i =0.1 and σ_i =0.5. We randomly select x_i with probability based on this Gaussian distribution:

$$p(X = x_i) = \frac{e^{-(x_i-\mu_i)^2/(2\sigma_i^2)}}{\sigma_i\sqrt{2\pi}}$$

$$z = \begin{pmatrix} z_1 \\ \vdots \\ z_{20} \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu_1 \\ \vdots \\ \mu_{20} \end{pmatrix}, \begin{pmatrix} \sigma_1 \\ \vdots \\ \sigma_{20} \end{pmatrix} \right)$$

Variational Autoencoders (VAEs)

- Say, the encoder generates μ = (0, −0.01, ..., 0.2) and σ = (0.05, 0.01, ..., 0.02). We can sample a value from this distribution:

$$N\left(\begin{pmatrix} 0 \\ -0.01 \\ \vdots \\ 0.2 \end{pmatrix}, \begin{pmatrix} 0.05 \\ 0.01 \\ \vdots \\ 0.02 \end{pmatrix}\right)$$

- with the latent variables as (say) :

$$z = \begin{pmatrix} Z_1 \\ \vdots \\ Z_{20} \end{pmatrix} = \begin{pmatrix} 0.03 \\ -0.015 \\ \vdots \\ 0.197 \end{pmatrix}$$

Variational Autoencoders (VAEs)

- The autoencoder in the previous section is very hard to train with not much guarantee that the network is generalize enough to make good predictions.(We say the network simply memorize the training data.)
- In VAEs, we add a constrain to make sure:
  1. The latent variable are relative independent of each other, i.e. the 20 variables are relatively independent of each other (not co-related). This maximizes what a 20-D latent vectors can represent.
  2. Latent variables z which are similar in values should generate similar looking images. This is a good indication that the network is not trying to memorize individual image.

Variational Autoencoders (VAEs)

- To achieve this, we want the gaussian distribution model generated by the encoder to be as close to a normal gaussian distribution function.
- We penalize the cost function if the gaussian function is deviate from a normal distribution. This is very similar to the L_2 regularization in a fully connected network in avoiding overfitting.

$$z \sim N(0, 1) = normal\ distribution$$

Variational Autoencoders (VAEs)

- In a normal gaussian distribution, the covariance E_{ij} is 0 for $i \neq j$. That is the latent variables are independent of each other.
- If the distribution is normalize, the distance between different z will be a good measure of its similarity.
- With sampling and the gaussian distribution, we encourage the network to have similar value of z for similar images.

Cost Function in Details

- In VAE, we want to model the data distribution $p(x)$ with an encoder $q_{\emptyset}(z|x)$, a decoder $p_{\theta}(z|x)$ and a latent variable model $p(z)$ through the VAE objective function:

$$\log p(x) \approx E_q[logp_{\theta}(x|z)] - D_{KL}[q_{\emptyset}(z|x)||p(z)]$$

- To draw this conclusion, we start with the KL divergence which measures the difference of 2 distributions. By definition, KL divergence is defined as:

$$D_{KL}(q||p) = \sum_{x} q(x) \log\left(\frac{q(x)}{p(x)}\right) = E_q[\log\big(q(x)\big) - \log\big(p(x)\big)]$$

- Apply it with:

$$D_{KL}\big[\big(q\big||p\big)\big||p(z|x)\big] = E[logq(z|x) - logp(z|x)]$$

Cost Function in Details

- Let $q_\lambda(z|x)$ be the distribution of z predicted by our encoder deep network. We want it to match the true distribution $p(z|x)$. We want the distribution approximated by the deep network has little divergence from the true distribution. i.e. we want to optimize λ with the smallest KL divergence.

$$D_{KL}[(q_\lambda(z|x)||p(z|x)] = E[log q_\lambda(z|x) - log p(z|x)]$$

- Apply:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$$

$$\begin{aligned}
D_{KL}[q_\lambda(z|x)\|p(z|x)] &= \mathbb{E}_q[\log q_\lambda(z|x) - \log \frac{p(x|z)p(z)}{p(x)}] \\
&= \mathbb{E}_q[\log q_\lambda(z|x) - \log p(x|z) - \log p(z) + \log p(x)] \\
&= \mathbb{E}_q[\log q_\lambda(z|x) - \log p(x|z) - \log p(z)] + \log p(x) \\
D_{KL}[q_\lambda(z|x)\|p(z|x)] - \log p(x) &= \mathbb{E}_q[\log q_\lambda(z|x) - \log p(x|z) - \log p(z)] \\
\log p(x) - D_{KL}[q_\lambda(z|x)\|p(z|x)] &= \mathbb{E}_q[\log p(x|z) - (\log q_\lambda(z|x) - \log p(z))] \\
&= \mathbb{E}_q[\log p(x|z)] - \mathbb{E}_q[\log q_\lambda(z|x) - \log p(z))] \\
&= \mathbb{E}_q[\log p(x|z)] - D_{KL}[q_\lambda(z|x)\|p(z)]
\end{aligned}$$

Cost Function in Details

- Define the term ELBO (Evidence lower bound) as:

$$ELBO(\lambda) = E_q[logp(x|z)] - D_{KL}[q_\lambda(z|x)||p(z)]$$

$$logp(x) - D_{KL}[q_\lambda(z|x)||p(z|x)] = ELBO(\lambda)$$

- We call ELBO the evidence lower bound because:

$$logp(x) - D_{KL}[q_\lambda(z|x)||p(z|x)]=ELBO(\lambda)$$

$$logp(x) \geq ELBO(\lambda) \text{ since KL is always positive}$$

- Here, we define our VAE objective function

$$logp(x) - D_{KL}[q_\lambda(z|x)||p(z|x)]=E_q[logp(x|z)] - D_{KL}[q_\lambda(z|x)||p(z)]$$

Cost Function in Details

- Instead of the distribution $p(x)$, we can model the data x with $\log p(x)$. With the error term, $D_{KL}[q_\lambda(z|x)||p(z|x)]$, we can establish a lower bound ELBO for $\log p(x)$ which in practice is good enough in modeling the data distribution.
- In the VAE objective function, maximize our model probability $\log p(x)$. is the same as maximize $\log p(x|z)$. while minimize the divergence of $D_{KL}[q_\lambda(z|x)||p(z)]$.

Cost Function in Details

- Maximizing $\log p(x|z)$ can be done by building a decoder network and maximize its likelihood. So with an encoder $q_\emptyset(z|x)$, a decoder $p_\theta(z|x)$, our objective become optimizing:

$$ELBO(\theta, \emptyset) = E_{q\theta(z|x)}\left[\log\left(p_\theta(x_i|z)\right)\right] - D_{KL}[q_\emptyset(z|x)||p(z)]$$

- We can apply a constrain to $p(z)$ such that we can evaluate $D_{KL}[q_\emptyset(z|x)||p(z)]$ easily. In VAE, we use $p(z) = N(0,1)$. For optimal solution, we want $q_\emptyset(z|x)$ to be as close as $N(0,1)$.

Cost Function in Details

- In VAE, we model $q_{\emptyset}(z|x)$ as $N(\mu, \Sigma)$

$$\begin{aligned}
D_{KL}[q_\phi(z|x)\|p(z)] &= D_{KL}[N(\mu,\Sigma)\|N(0,1)] \\
&= \frac{1}{2}\left(\mathrm{tr}(\Sigma) + \mu^T\mu - k - \log\det(\Sigma)\right) \\
&= \frac{1}{2}\left(\sum_k \Sigma + \sum_k \mu^2 - \sum_k 1 - \log\prod_k \Sigma\right) \\
&= \frac{1}{2}\left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \sum_k \log\Sigma(X)\right) \\
&= \frac{1}{2}\sum_k\left(\Sigma + \mu^2 - 1 - \log\Sigma\right)
\end{aligned}$$

Example 3.3 : VAE

Define the model architecture

Define the latent vector dimension

```
class VAE(nn.Module):
    def __init__(self, image_channels=1, h_dim=1600, z_dim=20):
        super(VAE, self).__init__()

        self.conv1 = nn.Conv2d(image_channels, 32, kernel_size=4, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)

        self.fc1 = nn.Linear(h_dim, z_dim)
        self.fc2 = nn.Linear(h_dim, z_dim)
        self.fc3 = nn.Linear(z_dim, h_dim)

        self.deconv2 = nn.ConvTranspose2d(64, 32, kernel_size=5, stride=2)
        self.deconv1 = nn.ConvTranspose2d(32, image_channels, kernel_size=4, stride=2)
```

Example 3.3 : VAE

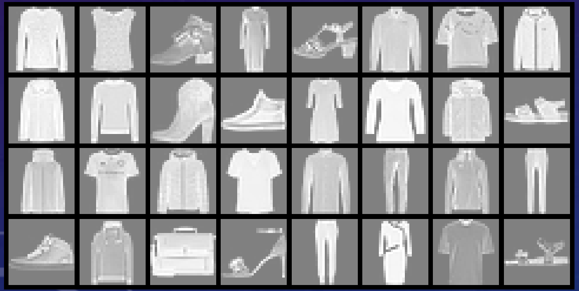
```
if epoch % 1 == 0:
    save = img.cpu().data
    save_image(img, './vae_img/input_{}.png'.format(epoch))
    save = recon_batch.cpu().data
    save_image(save, './vae_img/output_{}.png'.format(epoch))
```

$\longrightarrow$ Save the input images

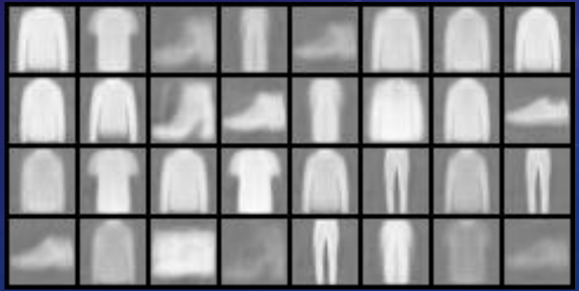
$\longrightarrow$ Save the reconstruction images

Epoch: 0

Input

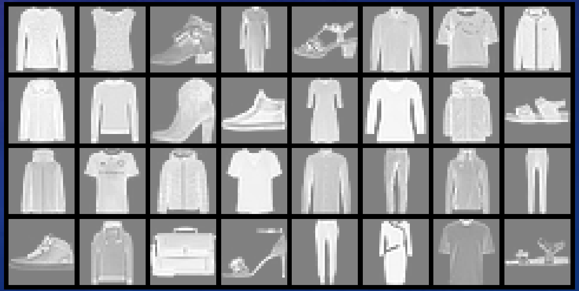


Reconstruct



Epoch: 4

Input



Reconstruct