Artificial Neural Networks

Course-9

Gavrilut Dragos

rev 3

AGENDA FOR TODAY

- Introduction to Auto-Encoders
- Demo
- Usage & Characteristics

Introduction to Auto-Encoders

Introduction to Auto-Encoders

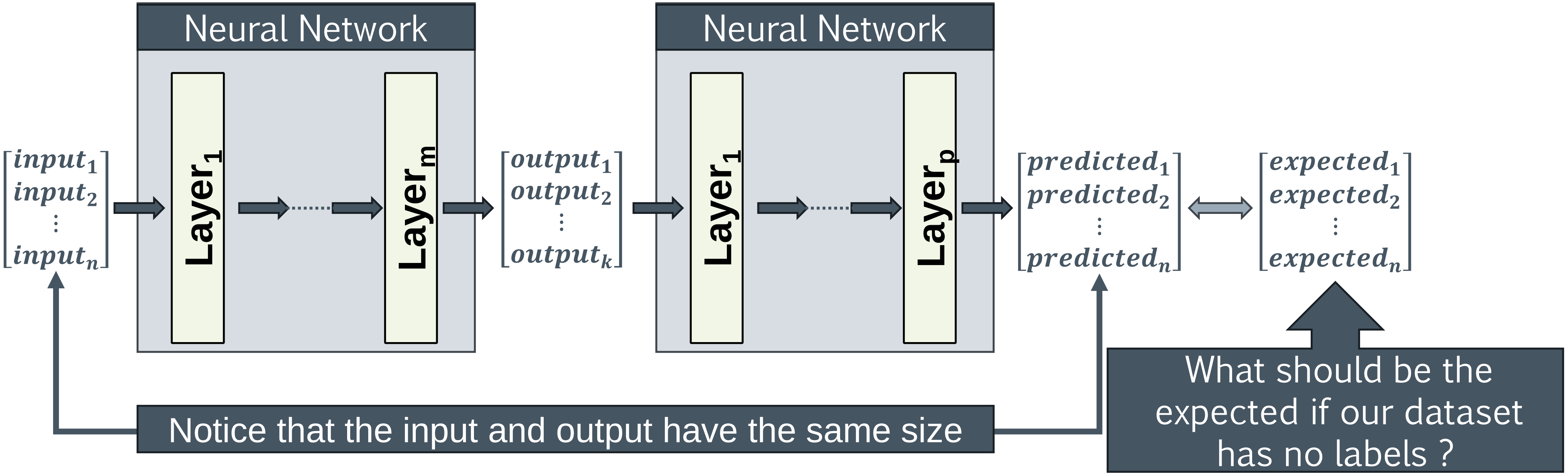
Up to this point, we have discussed how neural networks can be trained if we have a labeled dataset:

- An input
- An expected output (target)

But … what if we only have the input but no output – can we use neural networks for these cases ?

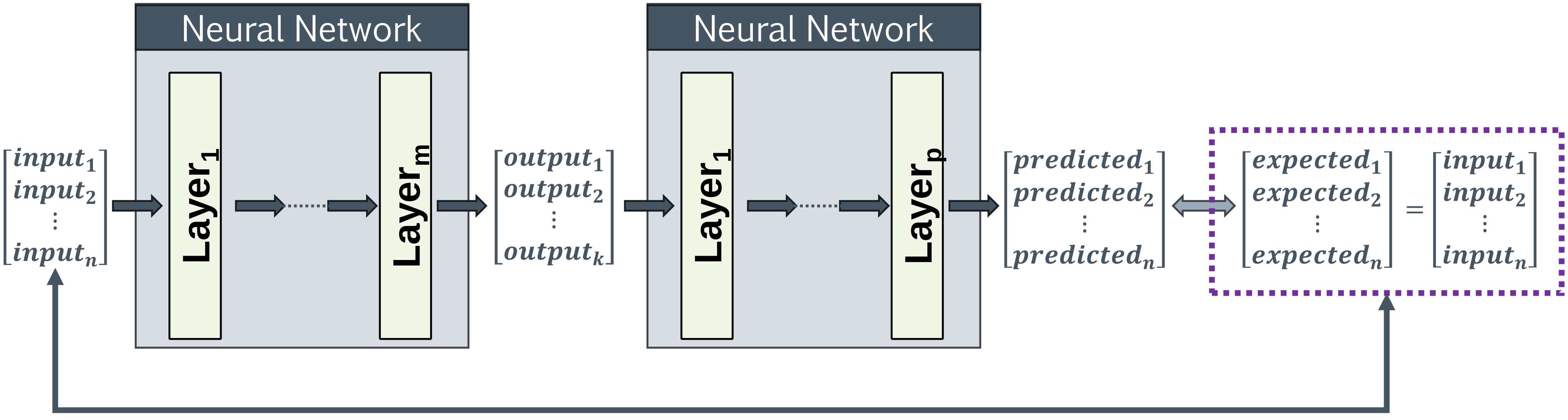
Introduction to Auto-Encoders

Let's consider the following neural network:



Introduction to Auto-Encoders

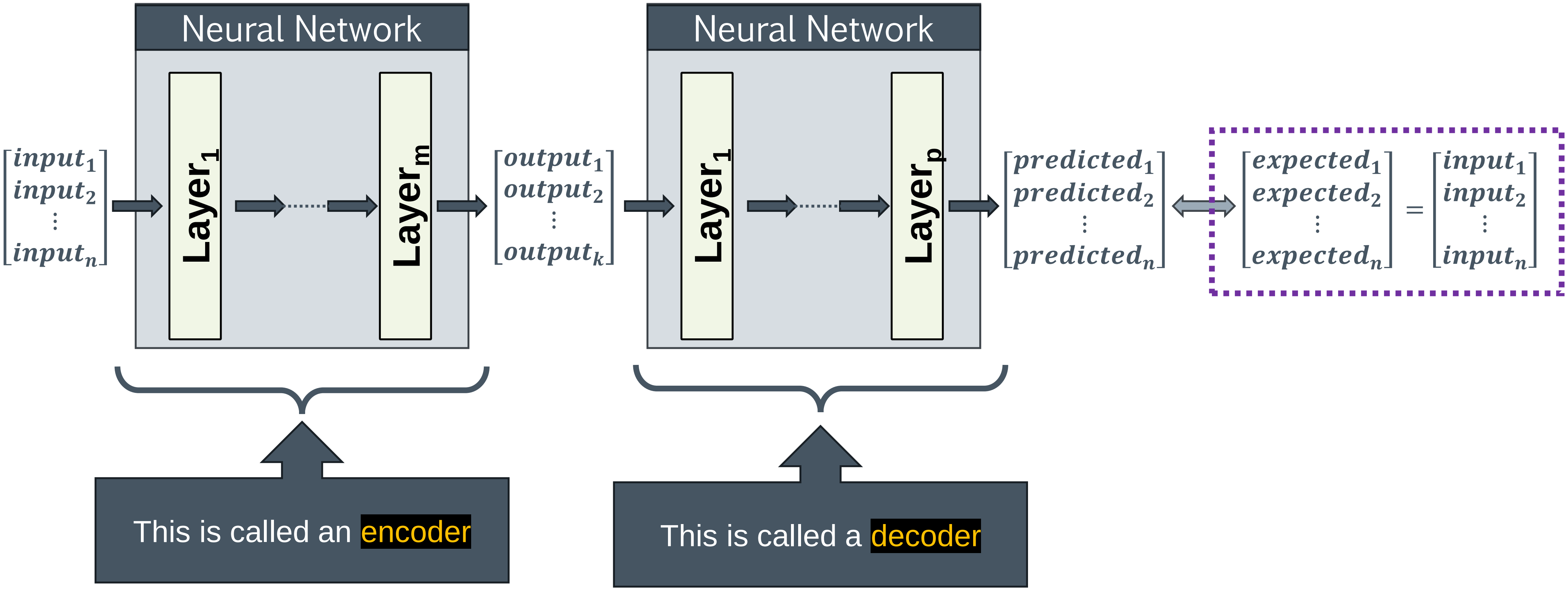
Let's consider the following neural network:



What if the input from the first neural network is the target (expected output) for the second neural network ?

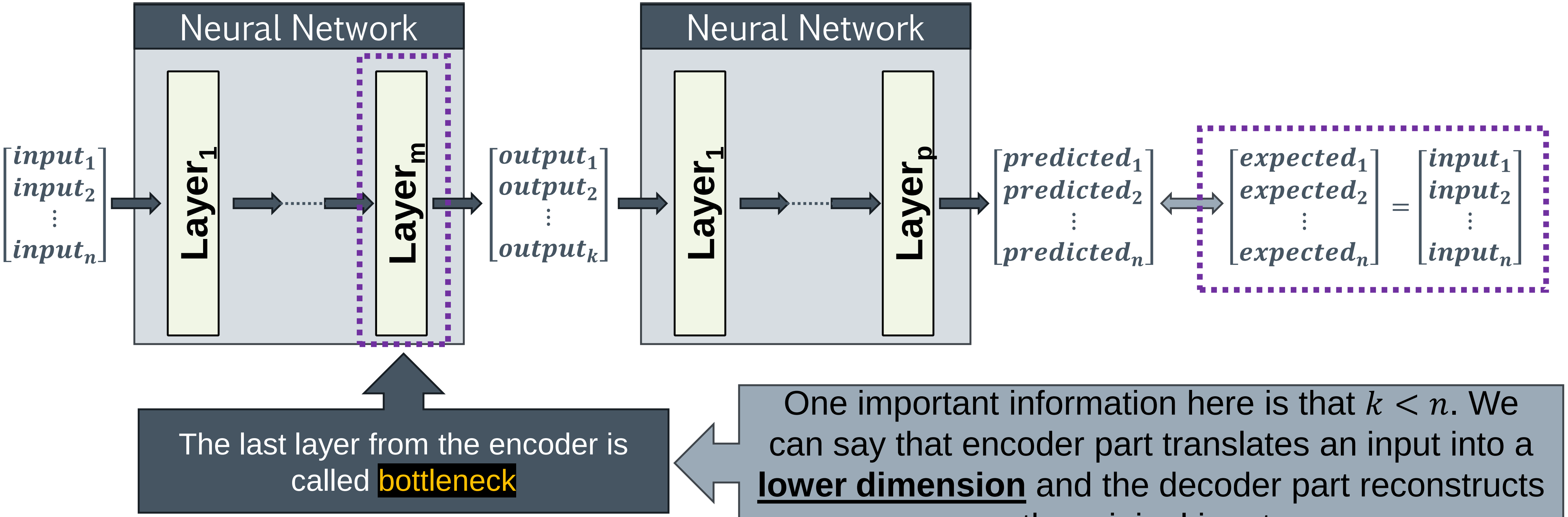
Introduction to Auto-Encoders

Let's consider the following neural network:



Introduction to Auto-Encoders

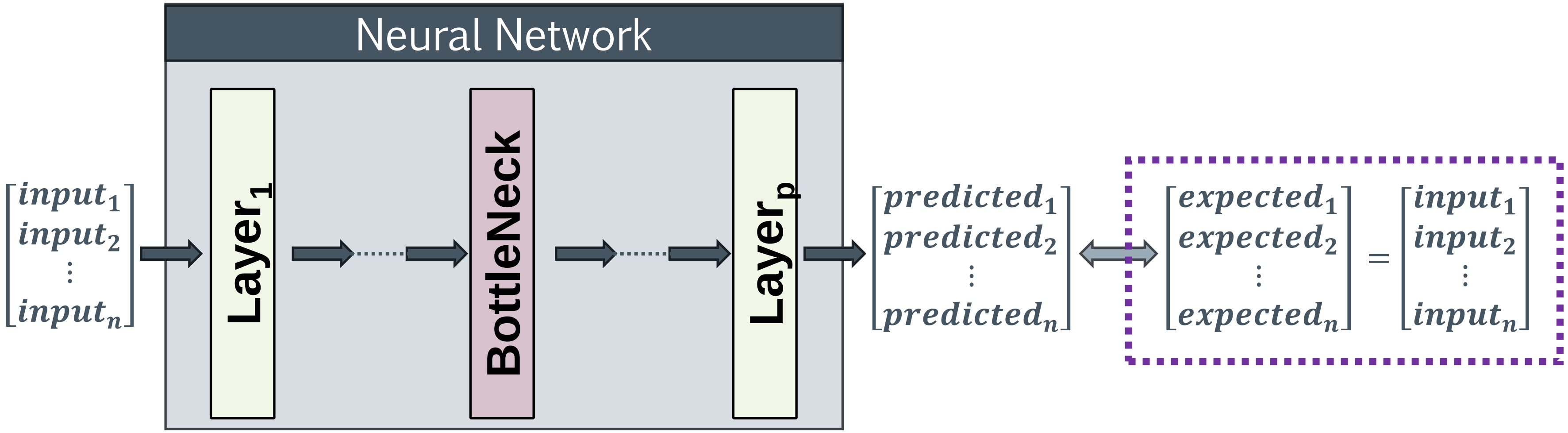
Let's consider the following neural network:



One important information here is that $k < n$. We can say that encoder part translates an input into a **lower dimension** and the decoder part reconstructs the original input.

Introduction to Auto-Encoders

With this in min we can define an auto-encoder in a following way:

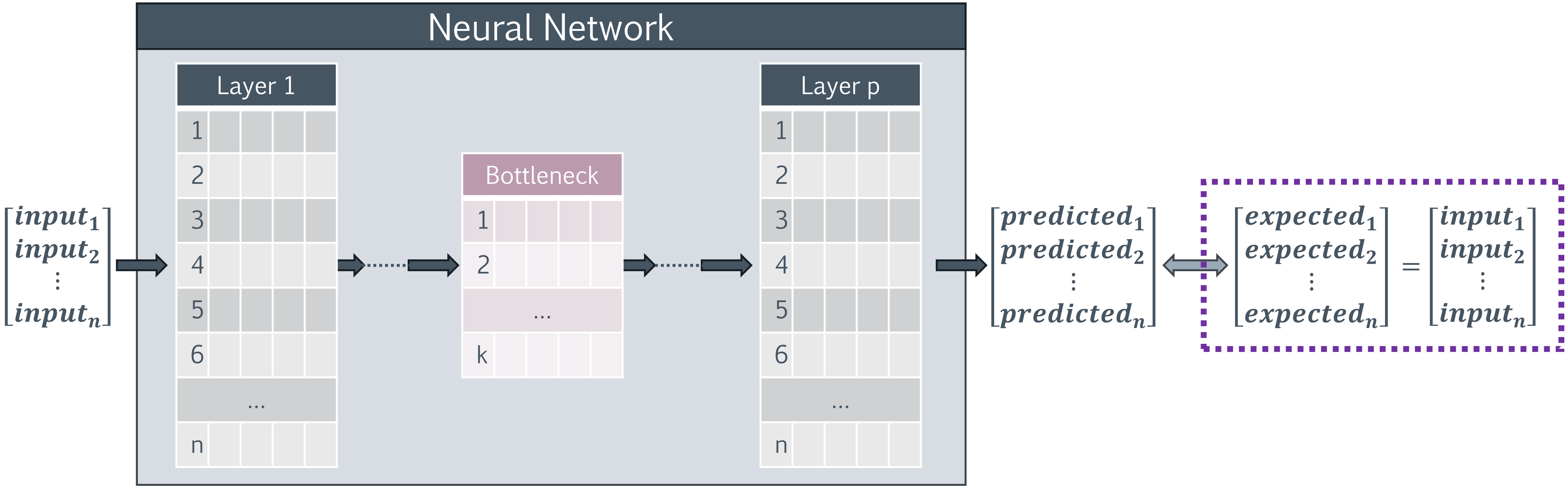


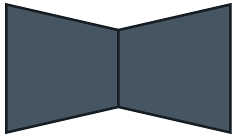
We also know that:

- Layer_1 and Layer_p have the same size (n)
- Bottleneck layer has a smaller size than the first and last layer

Introduction to Auto-Encoders

As such another way we can describe this is as follows:





Notice the form of the neural network, that reflects why the middle layers is called ***bottleneck***

Demo

Demo

Let's imagine the following scenario:

- A database with images (for simplicity we will use black and white images of 8x8 pixels)
- We use those images to train an auto-encoder
- Then we use another dataset with different images to see what we can observe n terms of the auto-encoder behavior.

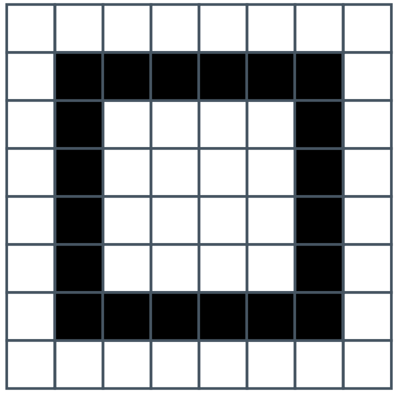
We will organize the code into several components:

- Training and Testing datasets
- Autoencoder
- Training code

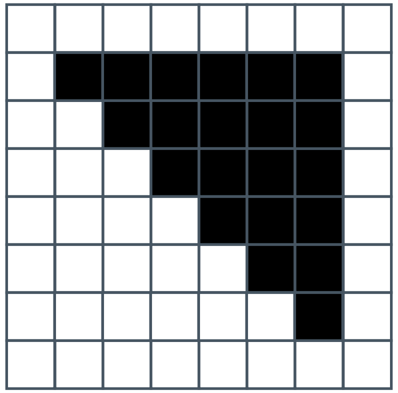
Demo

Training dataset:

- 2 images
- 8x8 pixels (black and white)



A square into the middle of the image

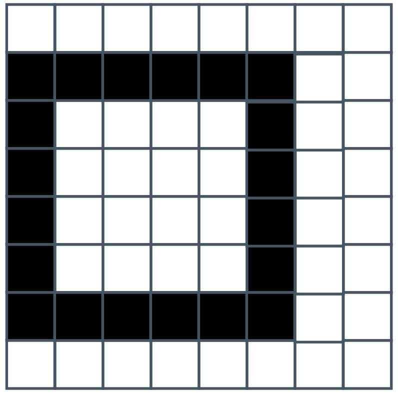


A filled triangle

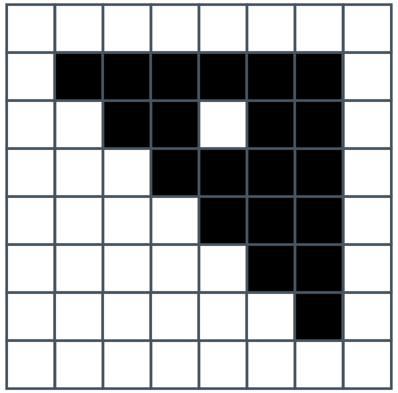
Demo

Testing dataset:

- 2 images
- 8x8 pixels (black and white)

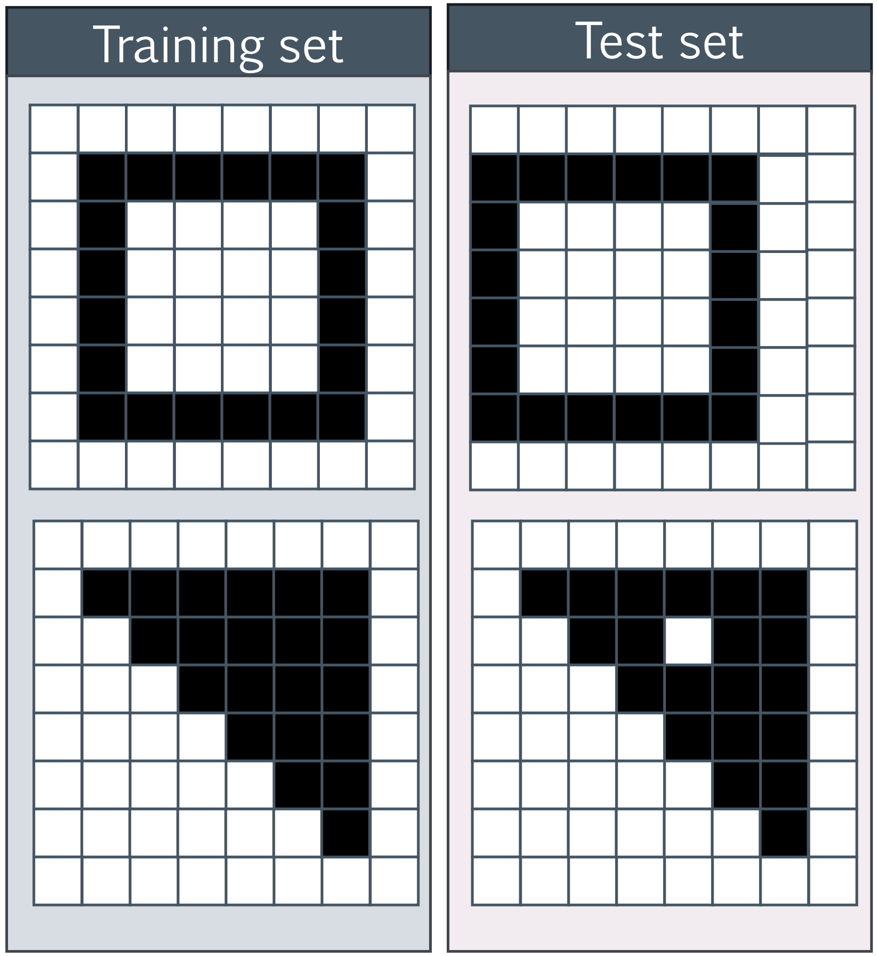


A square (similar to the previous one) but on the left



A filled triangle with some noise (not all pixels are black)

Demo (Summary)



In this case we will examine how well the auto-encoder adjust to an image that was moved (to the left in this case)

In this case we will examine how well the auto-encoder adjust to an image that has noise in it (in our case, a white pixel where a black one should have been).

Demo

Let's see how we can load these datasets:

```
import torch
from torch.utils.data import DataLoader

class CustomDataset(Dataset):
    def __init__(self, images):
        self.images = []
        for img in images:
            self.images += [torch.tensor(self.string_to_vector(img))]

    def string_to_vector(self, txt):
        ……………………………

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        return image, 0
```

Demo

Let's see how we can load these datasets:

```
import torch
from torch.utils.data import DataLoader

class CustomDataset(Dataset):
    def __init__(self, images):
        self.images = []
        for img in images:
            self.images += [torch.tens

    def string_to_vector(self, txt):
        ..............................

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        return image, 0
```

```
def string_to_vector(self, txt):
    v = []
    for c in txt:
        if c == 'X':
            v+=[1.0]
        elif c=='.':
            v+=[0.0]
    if len(v)!=64:
        raise Exception("Invalid size")
    return v
```

Demo

We will instantiate the two datasets using the following constants:

```
train_set = [
    "........ .XXXXXX. .X....X. .X....X. .X....X. .X....X. .XXXXXX. ........",
    "........ .XXXXXX. ..XXXXX. ...XXXX. ....XXX. .....XX. ......X. ........",
]

test_set = [
    "........ XXXXXX.. X....X.. X....X.. X....X.. X....X.. XXXXXX.. ........",
    "........ .XXXXXX. ..XX.XX. ...XXXX. ....XXX. .....XX. ......X. ........",
]
```

Notice that the two list of strings reflect the presented images (with character X being a black pixel and character . (point) a white one)

Demo

Let’s move on to the Autoencoder. Since we are dealing with a very small data set, it will not be formed out of too many layers.

We will however, split the autoencoder into two parts:

- Encoder
- Decoder

The size of the input will be 64 values (8 x 8 pixels for a picture).

Demo

Auto-encoder class:

```
class Autoencoder(torch.nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(8*8, 16),
            nn.ReLU(),
            nn.Linear(16,4),
        )
        self.decoder = nn.Sequential(
            nn.Linear(4,16),
            nn.ReLU(),
            nn.Linear(16, 8*8),
            nn.Tanh()
        )
    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Demo

Auto-encoder class:

```
class Autoencoder(torch.nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(8*8, 16),
            nn.ReLU(),
            nn.Linear(16,4),
        )
        self.decoder = nn.Sequential(
            nn.Linear(4,16),
            nn.ReLU(),
            nn.Linear(16, 8*8),
            nn.Tanh()
        )
    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Notice that we encode 64 values into 4 through an intermediate layer of 16 values.

Demo

Now let’s put all of these together and train an auto-encoder:

```
autoencoder = Autoencoder()
loss_function = nn.MSELoss()
optimizer = optim.Adam(autoencoder.parameters())

train_loader = CustomDataset(train_set)
test_loader = CustomDataset(test_set)

num_epochs = <a large number… more than 500>
for epoch in range(num_epochs):
    for data in train_loader:
        output = autoencoder(data[0])
        loss = loss_function(output, data[0])

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Demo

Now let's put all of these together and train an auto-encoder:

```
autoencoder = Autoencoder()
loss_function = nn.MSELoss()
optimizer = optim.Adam(autoencoder.parameters())

train_loader = CustomDataset(train_set)
test_loader = CustomDataset(test_set)

num_epochs = <a large number… more than 500>
for epoch in range(num_epochs):
    for data in train_loader:
        output = autoencoder(data[0])
        loss = loss_function(output, data[0])

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Notice that the input (data[0]) is the same as the expected (targeted value) that we use in the loss function.

Demo

So … we have a model, what's next ?

- We know that the output is also a 64 bytes value (so we can try to map it into an 8x8 pixel picture) → in a way we can say that we can try to reconstruct an image.
- But, our picture is black and white and the output of the network uses more discrete values. As such, we will need a convention on what values become white pixels and what values become black pixels (for this example we can use a threshold of 0.5).

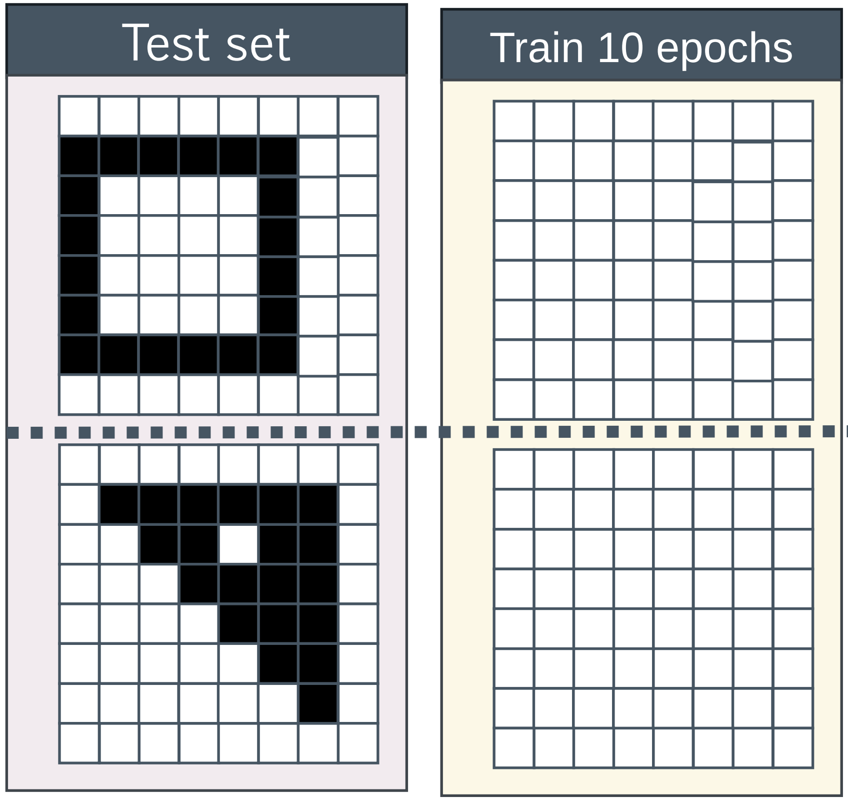
Demo

And the code:

```
for data in test_loader:
    output = autoencoder(data[0])
    s = ""
    for i in range(0,64):
        if i%8==0:
            s+="\n"
        if output[i]<0.5:
            s+="."
        else:
            s+="X"
    print(s)
```

Demo

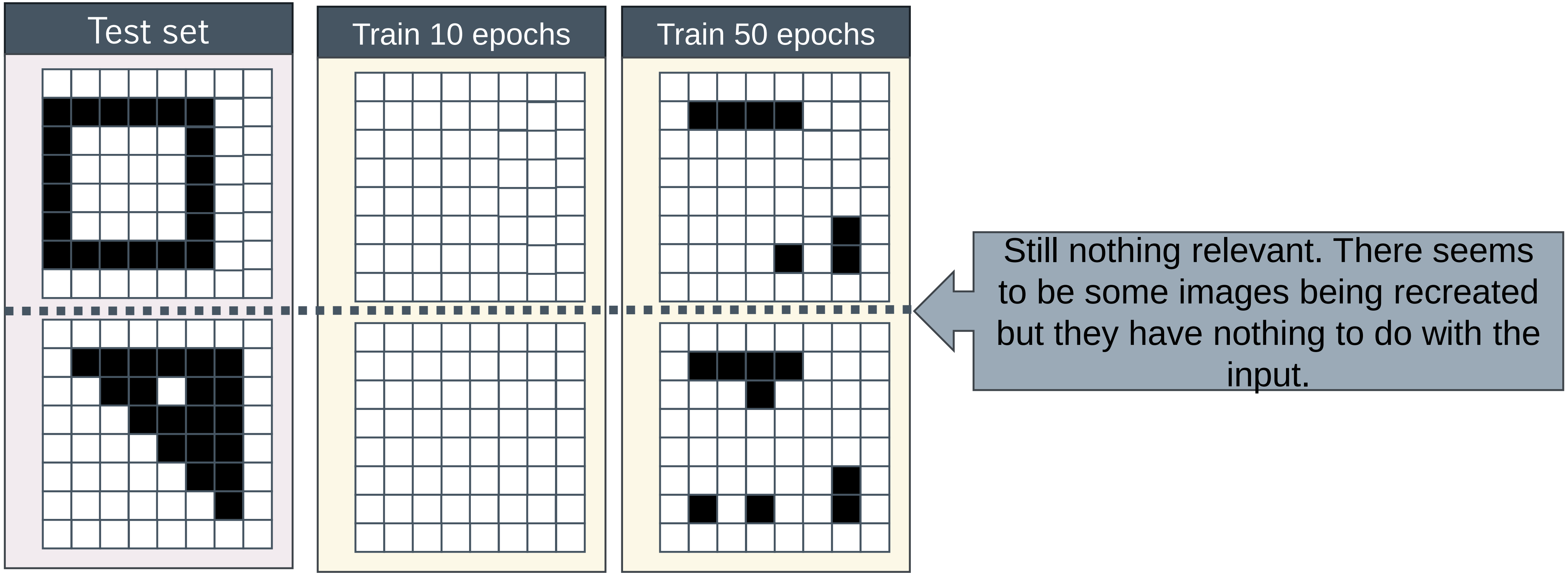
What's interesting to study here is how the output image varies depending on the test set and the number of epochs (iterations) used for training.



The first conclusion is that after 10 epochs of training, the net is not capable of getting anything out the of 2 pictures used for testing.

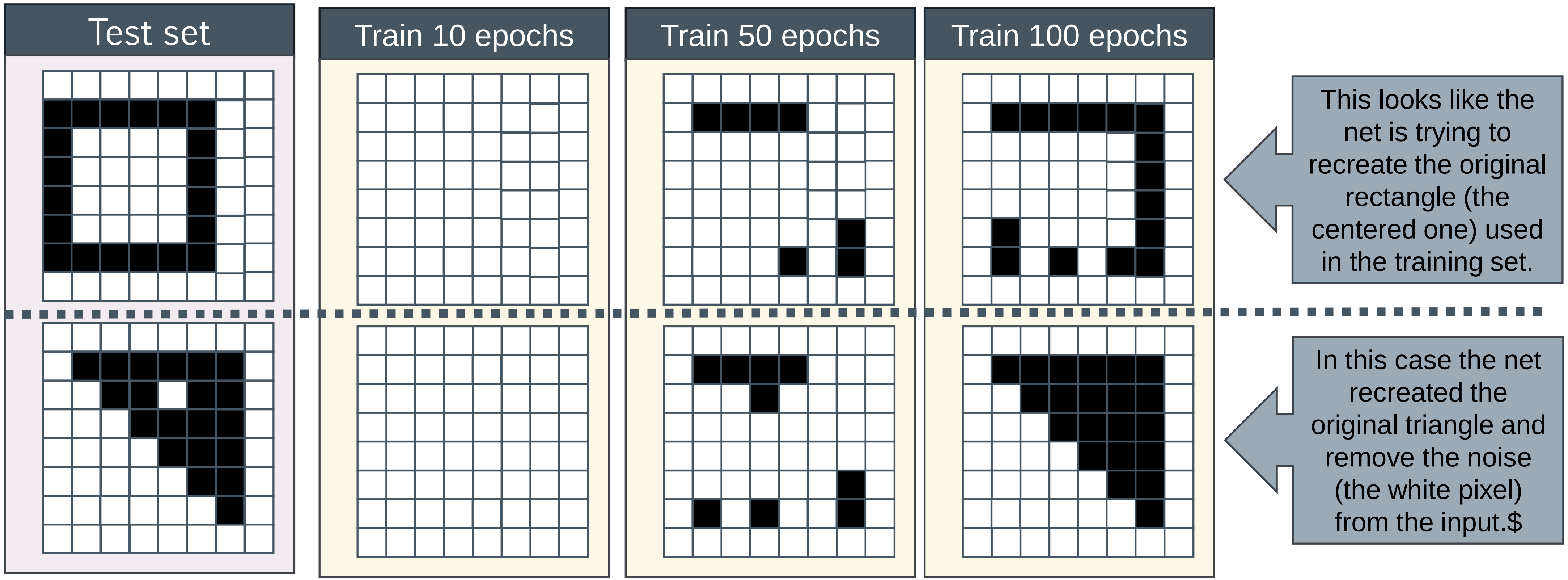
Demo

What's interesting to study here is how the output image varies depending on the test set and the number of epochs (iterations) used for training.



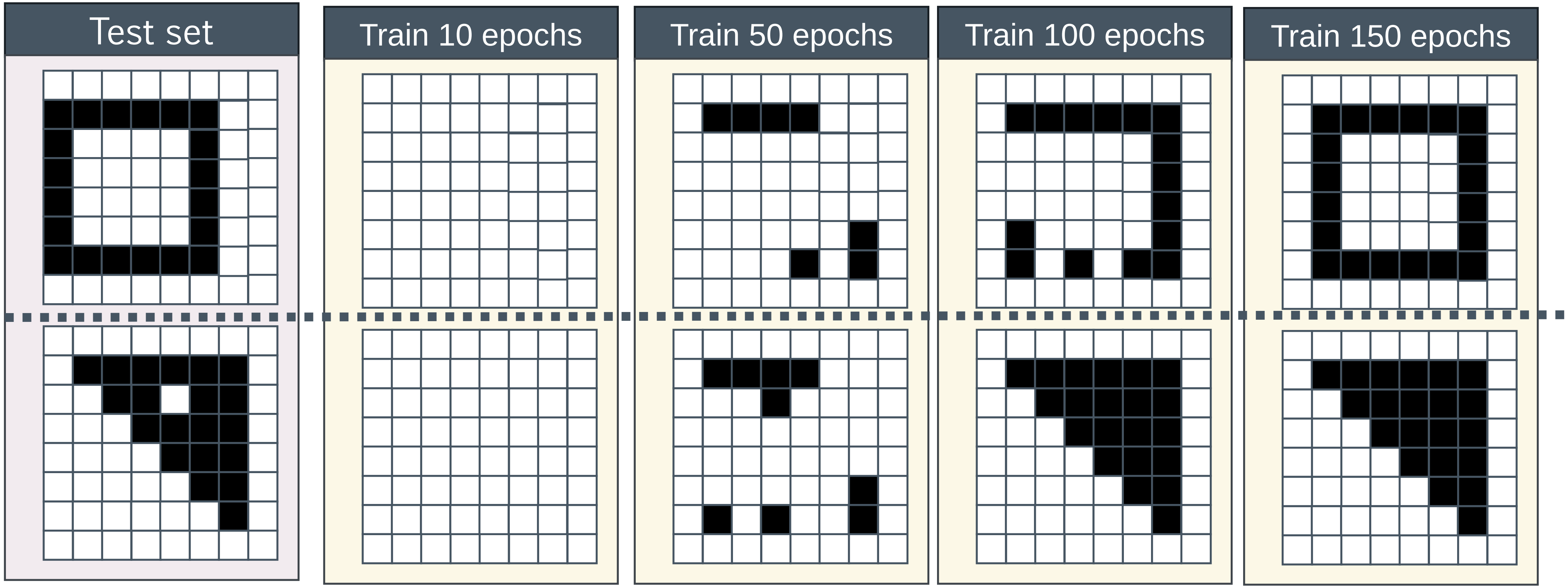
Demo

What's interesting to study here is how the output image varies depending on the test set and the number of epochs (iterations) used for training.



Demo

What's interesting to study here is how the output image varies depending on the test set and the number of epochs (iterations) used for training.



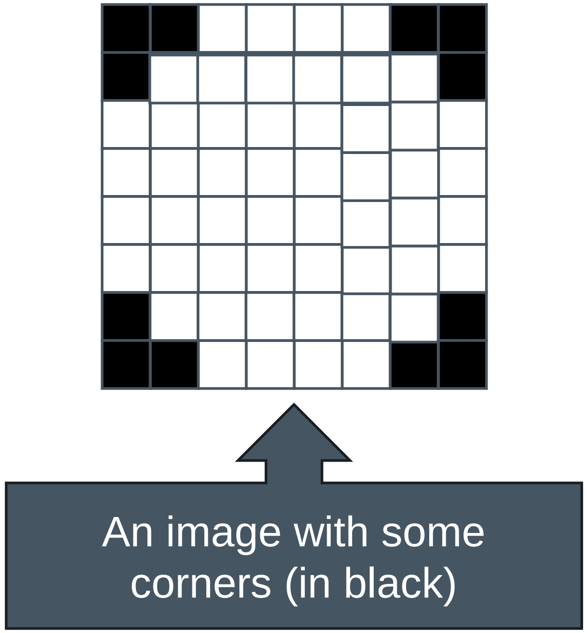
Demo

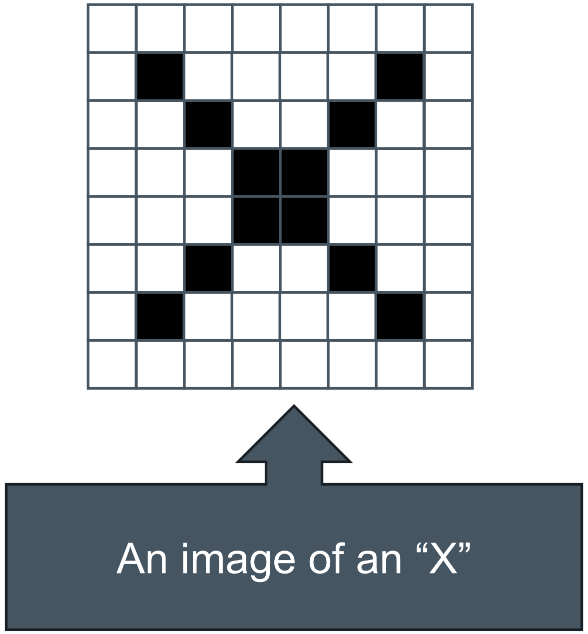
So … what are the conclusions:

1. The net will try **to map an input to the closest image** that was used in the training.
2. This process will **reduce noise** if present
3. Moving an image laterally is also covered if the image looks like something from the dataset.

Demo

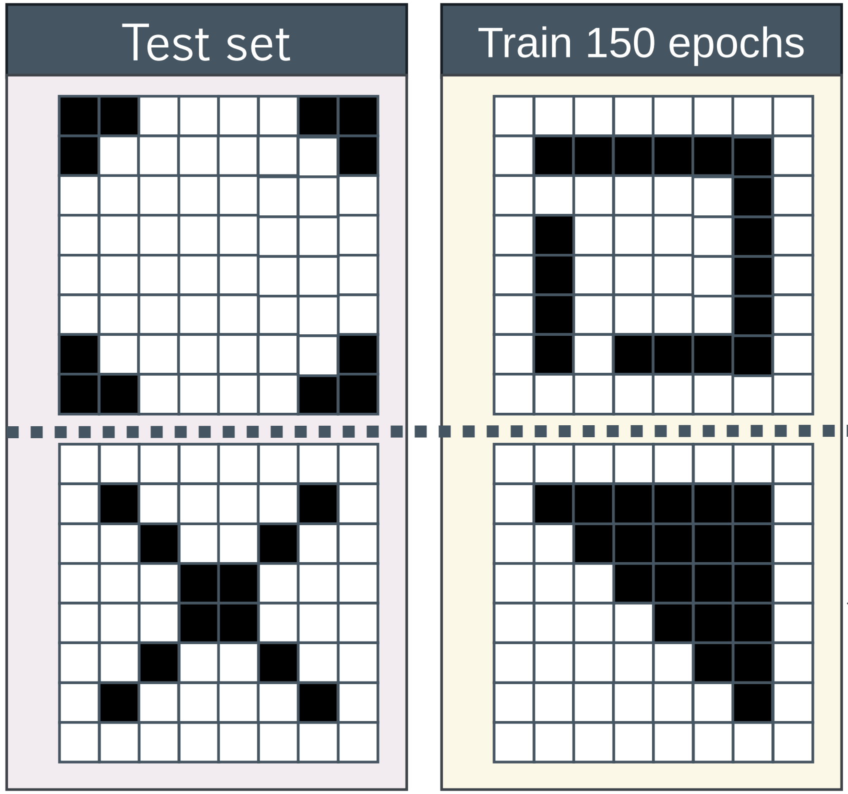
Let's try to add into the testing set some images that don't really look like anything from the training set:





Demo

And now let's see what is the output our net (if trained for 150 epochs) in this case:



In this case it looks like the net is trying to recreate the square (from the training set). This could be an indicator that in reality the net has learned **how to distinguish corners** !

In this case, it is more likely that the net has learned how to **identify diagonals** (from the triangle) and as this is the closest to the X letter, it translates X into a triangle.

Usage & Characteristics

Usage & Characteristics

Where can we use auto-encoders:

- Similar to PCA, auto-encoders can be used for reducing the **number of dimensions** in the data without losing much information.
- Learning to encode data into a **compressed representation** which can then be used for various tasks like classification.
- **Remove noise** from data.
- In scenarios where the **normality/abnormality of new data needs to be determined**, auto-encoders can learn the normal patterns and then can detect anomalies in new data.

Anomaly Detection

In particular for anomaly detection, the idea is to train an auto-encoder and in the inference phase to evaluate the difference between the input and the result.

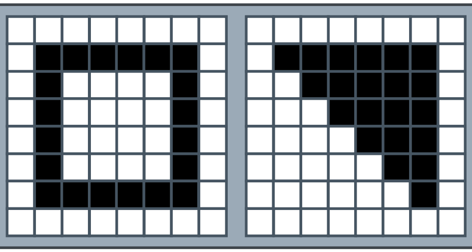
Once trained, the auto-encoder is used to reconstruct new data. The key idea is that the **auto-encoder will be good at reconstructing data that resembles the normal data** it was trained on, but it will **perform poorly on data that differs significantly from this training data** — the anomalies.

Anomaly Detection

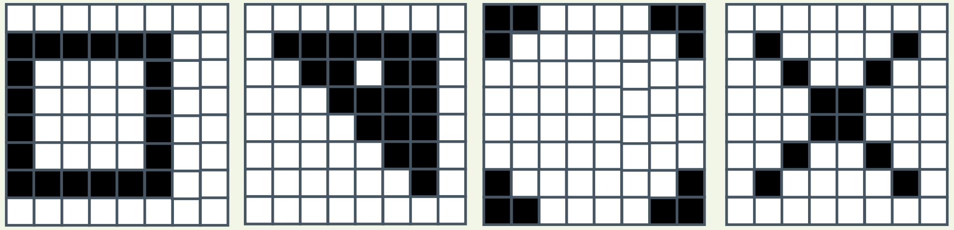
The key idea is to find a **threshold** that reflects how similar the output and the input are.

Let’s analyze the previous example:

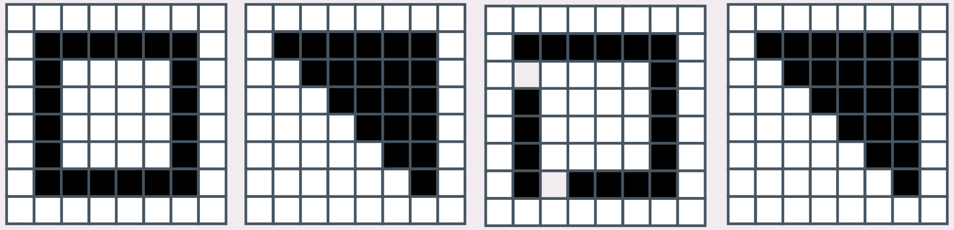
Training Set



Testing Set



Outputted Image



Anomaly Detection

Let's rewrite an auto-encoder in the following way:

$$\boldsymbol{I} = \begin{bmatrix} \boldsymbol{input_1} \\ \boldsymbol{input_2} \\ \vdots \\ \boldsymbol{input_n} \end{bmatrix}, \boldsymbol{O} = \boldsymbol{f}(\boldsymbol{I}) = \begin{bmatrix} \boldsymbol{output_1} \\ \boldsymbol{output_2} \\ \vdots \\ \boldsymbol{output_n} \end{bmatrix}, \boldsymbol{with\ f(x): R^n \rightarrow R^n, our\ auto - encoder\ described\ as\ a\ function}$$

then we will consider the function ***sim*** defined in the following way:

$$sim(I, O) = x, x \in [0,1], where:$$

$$x = 0\ \ means\ I\ and\ O\ are\ compliely\ different\ and$$
$$x = 1\ means\ I\ and\ O\ are\ indentical$$

Anomaly Detection

Let’s evaluate some similarity methods we can use:

Jaccard.

While Jaccard similarity is define for sets, as follows:

$$A = a\ set, B = another\ set, Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

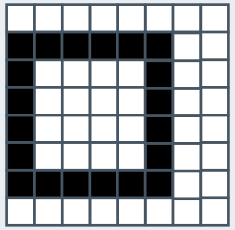
We can use a modified version (**Ruzicka** similarity):

$$I = \begin{bmatrix} input_1 \\ input_2 \\ \vdots \\ input_n \end{bmatrix}, O = \begin{bmatrix} output_1 \\ output_2 \\ \vdots \\ output_n \end{bmatrix}, Jaccard(I, O) = \frac{\sum_{j=1}^{n} min(I_j, O_j)}{\sum_{j=1}^{n} max(I_j, O_j)}$$

Anomaly Detection

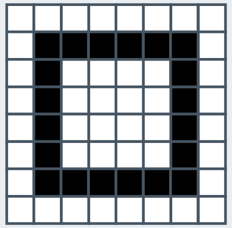
Let’s see how ***Ruzicka*** similarity is computed for one of our cases:

I =



```
I = [0,0,0,0,0,0,0,0,
     1,1,1,1,1,1,0,0,
     1,0,0,0,0,1,0,0,
     1,0,0,0,0,1,0,0,
     1,0,0,0,0,1,0,0,
     1,0,0,0,0,1,0,0,
     1,1,1,1,1,1,0,0,
     0,0,0,0,0,0,0,0]
```

O=



```
O = [0,0,0,0,0,0,0,0,
     0,1,1,1,1,1,1,0,
     0,1,0,0,0,0,1,0,
     0,1,0,0,0,0,1,0,
     0,1,0,0,0,0,1,0,
     0,1,0,0,0,0,1,0,
     0,1,1,1,1,1,1,0,
     0,0,0,0,0,0,0,0]
```

$$\sum_{j=1}^{n} min(I_j, O_j) = 10,$$

$$\sum_{j=1}^{n} max(I_j, O_j) = 30,$$

$$\text{similarity} = \frac{10}{30} = 33\%$$

Anomaly Detection

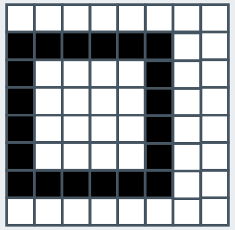
Another way to compute the similarity is to use the Hamming distance (e.g., count how many positions from the I and O vectors have values that are different).

In other words:

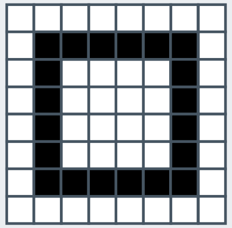
$$I = \begin{bmatrix} input_1 \\ input_2 \\ \vdots \\ input_n \end{bmatrix}, O = \begin{bmatrix} output_1 \\ output_2 \\ \vdots \\ output_n \end{bmatrix}, Hamming(I, O) = 1 - \frac{\sum_{j=1}^{n} |I_j - O_j|}{n}$$

Anomaly Detection

Let's see how ***Hamming*** similarity is computed for one of our cases:

$I =$ 

```
I = [0,0,0,0,0,0,0,0,
     1,1,1,1,1,1,0,0,
     1,0,0,0,0,1,0,0,
     1,0,0,0,0,1,0,0,
     1,0,0,0,0,1,0,0,
     1,0,0,0,0,1,0,0,
     1,1,1,1,1,1,0,0,
     0,0,0,0,0,0,0,0]
```

$O =$ 

```
O = [0,0,0,0,0,0,0,0,
     0,1,1,1,1,1,1,0,
     0,1,0,0,0,0,1,0,
     0,1,0,0,0,0,1,0,
     0,1,0,0,0,0,1,0,
     0,1,0,0,0,0,1,0,
     0,1,1,1,1,1,1,0,
     0,0,0,0,0,0,0,0]
```

$$\sum_{j=1}^{n} \left| I_j - O_j \right| = 20,$$

$$n = 64,$$

$$\text{similarity} = 1 - \frac{20}{64} = 68\%$$

Anomaly Detection

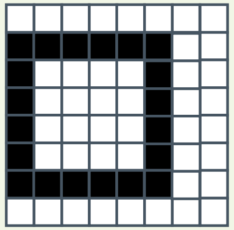
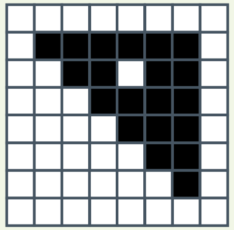
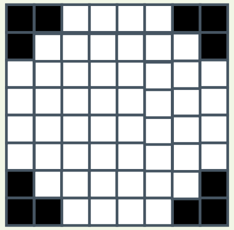
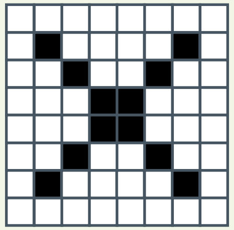
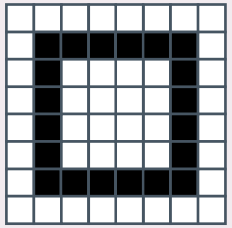
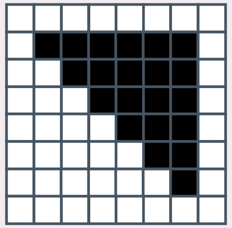
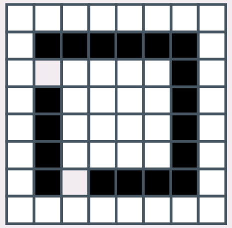
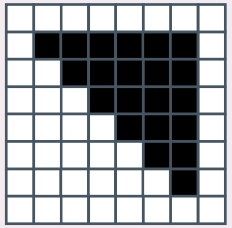
Other distances or adaptation can be used as well , such as:

- Dice coefficient
- Cosine coefficient
- Etc

Its important to chose a similarity metric that best fits the problem you are trying to solve with an auto-encoder.

Anomaly Detection

Let's use the Hamming distance for similarity and compare the results from our dataset:

Testing Set				
Similarity	68%	98%	53%	76%
Outputted Image				

If we chose a threshold of minimum **60%**, then the anomaly from these results will be the this one.

Usage & Characteristics

There are also various types of auto-encoders:

1. **Deep autoencoders** (use multiple layers in both the encoder and decoder, allowing them to learn more complex representations of the data)
2. **Convolutional Autoencoders** (use convolutional layers instead of fully connected layers)
3. **Denoising Autoencoders** (trained to remove noise or to reconstruct data)
4. **Sparse Autoencoders** (use sparsity constraints on the hidden layers to force the model to learn a more dispersed or sparse representation of the input data)

Usage & Characteristics

There are also various types of auto-encoders:

5. **Variational Autoencoders (VAEs)** (generative models that learn a probabilistic representation of the data)
6. **Contractive Autoencoders (CAEs)** (use a regularization term in their loss function that encourages the model to learn a function that's robust to slight variations of input data)
7. **Sequence-to-Sequence Autoencoders** (designed for sequential data (like time series or text))

Q & A