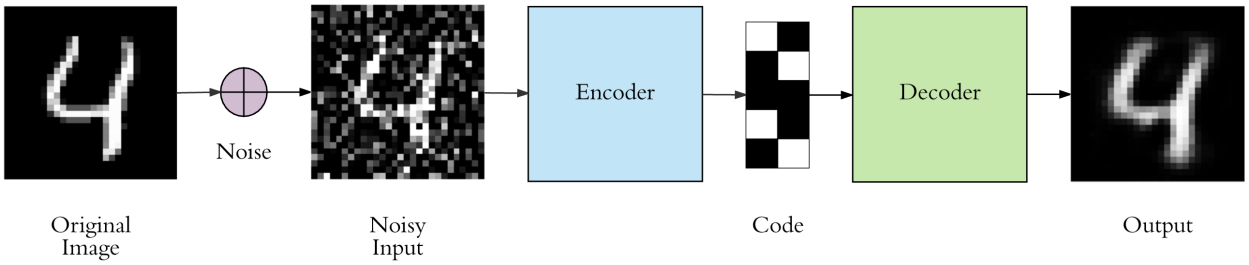
Denoising Autoencoder

In this lab, you will be applying a noise filter to MNIST images and using a denoising autoencoder to reconstruct the original image.



Setting up AWS

1. Before we get started, each of you will need to make an AWS account so that you can train your model on something more powerful than your personal device.

   Step one is to create an AWS **Student** Account. Students get $100 of free credit: https://aws.amazon.com/education/awseducate/ (https://aws.amazon.com/education/awseducate/)

2. Now we must request a server to run our code on. For the purposes of this homework, you can use a spot instance. Spot instances are on-demand servers that are cheaper to use than normal EC2 instances, but can be terminated at any time (so will lose data stored on that machine).

   `A p2.xlarge should suffice for this project.`

3. Then, push your code to a separate github repo and then `ssh` into your server, pull the repo.

   Another option is to use `scp` to securely send the files from your local machine to the server: http://www.hypexr.org/linux_scp_help.php (http://www.hypexr.org/linux_scp_help.php)

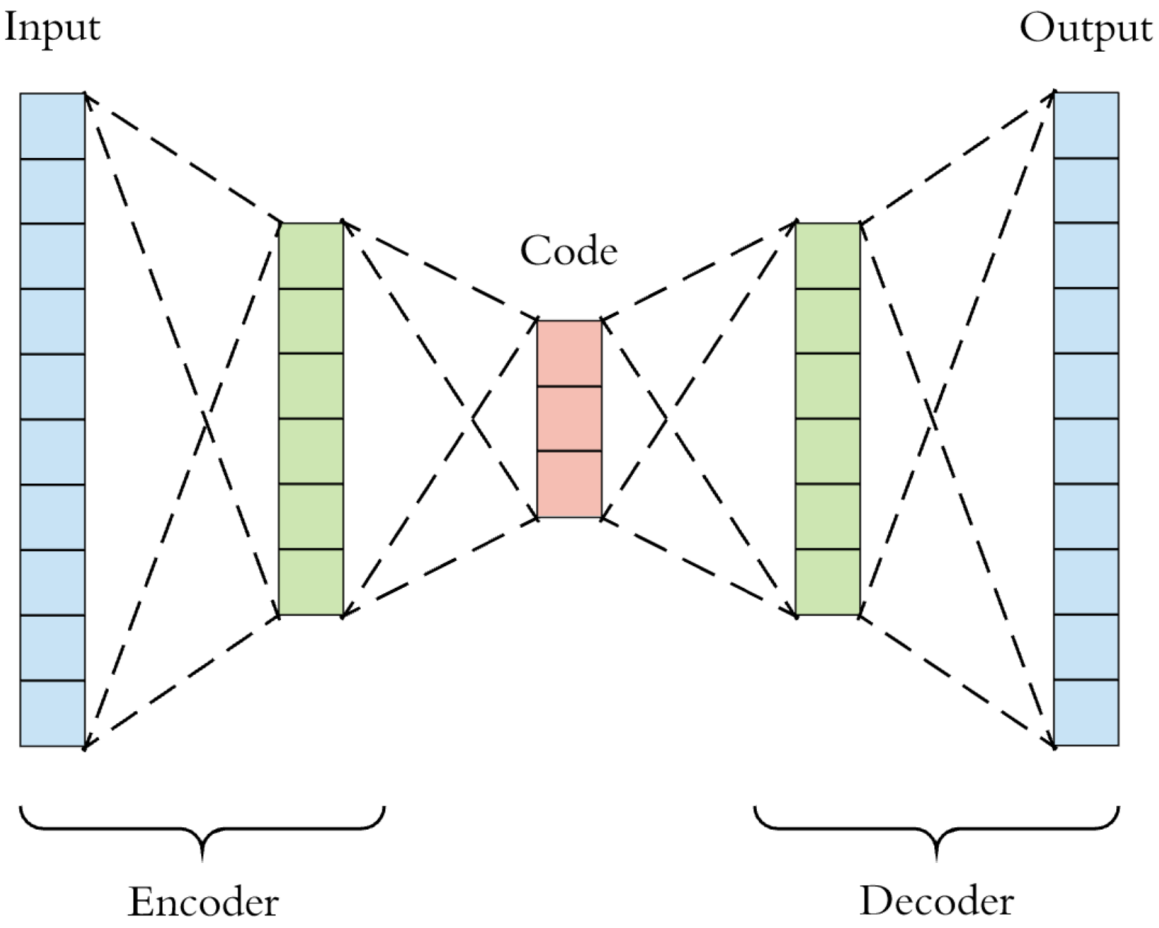
4. Since jupyter runs in the browser, we need a way to interact with the notebook on our local machine, but have everything run remotely.

   Follow these steps https://ljvmiranda921.github.io/notebook/2018/01/31/running-a-jupyter-notebook/ (https://ljvmiranda921.github.io/notebook/2018/01/31/running-a-jupyter-notebook/) to get the jupyter notebook up and running so that you can start training your model.

Lets Get Started!

In [1]:
```
import os
#Disable the warnings for now cuz they are annoying
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

We will be implementing the diagram below. To learn more, read the following: http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/ (http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/). The first few paragraphs should suffice to get a basic understanding of what an autoencoder is.



As seen in the first figure, a denoising autoencoder takes some noisy image and tries to reconstruct the original image. We will be attempting to reconstruct noisy MNIST images. To start, complete the below function to apply a mask to each input X.

In [2]:
```
def masking_noise(X, v):
    """ Apply masking noise to data in X, in other words a fraction v of el
    (chosen at random) is forced to zero.
    X is of size [batch_size, features]
    """

    X_noise = X.clone().detach()
    n_samples = X.shape[0]
    n_features = X.shape[1]

    for i in range(n_samples):
        mask = np.random.choice(n_features, (int) (v * n_features))
        for m in mask:
            X_noise[i][m] = 0.
    return X_noise
```

Network parameters and Hyperparameters

In [3]:
```
# Parameters
learning_rate = 0.01
training_epochs = 10
batch_size = 64

# Network Parameters
n_hidden_1 = 256 # 1st layer num features
n_hidden_2 = 128 # 2nd layer num features
n_input = 784 # MNIST data input (img shape: 28*28) NOTE: no 3rd dimension
```

Fill in the encoder and decoder blocks, and then complete the forward method:

- The encoder block takes in a noisy image and returns a 128 dimensional tensor (the middle layer)
- The decoder block takes in the middle layer of the network and outputs the restored image

(hint: nn.Sequential)

In [4]:
```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.encoder = nn.Sequential(
                        nn.Linear(in_features=n_input, out_features=n_hidde
                        nn.Linear(in_features=n_hidden_1, out_features=n_hi
                        )
        self.decoder = nn.Sequential(
                        nn.Linear(in_features=n_hidden_2, out_features=n_hi
                        nn.Linear(in_features=n_hidden_1, out_features=n_in
                        )

    def forward(self, input):
        encoded_input = torch.sigmoid(self.encoder(input))
        out = torch.sigmoid(self.decoder(encoded_input))
        return out
```

Side Note: If you have CUDA enabled and would like to place computations on an available GPU (or distribute over multiple), PyTorch gives each tensor the attribute "device", which can be changed to the desired GPU(s). To learn more, please see the article below.
https://towardsdatascience.com/speed-up-your-algorithms-part-1-pytorch-56d8a4ae7051 (https://towardsdatascience.com/speed-up-your-algorithms-part-1-pytorch-56d8a4ae7051)

In [5]:
```
#Print out the number of GPUs that are available
if torch.cuda.is_available():
    print(torch.cuda.device_count(), "gpus available")
    # uncomment this line if you'd like to follow the article above for GPU
    device = torch.device(0)
else:
    print('no gpus available')
```

```
no gpus available
```

Initialize the autoencoder model.

In [6]:
```
model = Model()
print(model)
```

```
Model(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=256, bias=True)
    (1): Linear(in_features=256, out_features=128, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=128, out_features=256, bias=True)
    (1): Linear(in_features=256, out_features=784, bias=True)
  )
)
```

Define the loss criterion for the model as Binary Cross Entropy loss (look through PyTorch documentation). This will be called later during the training loop.

In [7]:
```
criterion = nn.BCELoss()
```

Define the optimizer as the Adam optimizer which will be used to minimize the loss defined above.

In [8]:
```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Define a DataLoader for use in the training loop. This dataset/dataloader setup in PyTorch is extremely helpful and good to know. It is possible to create custom datasets pretty easily.
https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset (https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset)
https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader (https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader)

Note: you may have to pip install torchvision *here*

In [9]:
```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

MNIST_data = MNIST("../data/mnist", train=True, download=True, transform =
train_dataloader = DataLoader(MNIST_data, batch_size=batch_size)
```

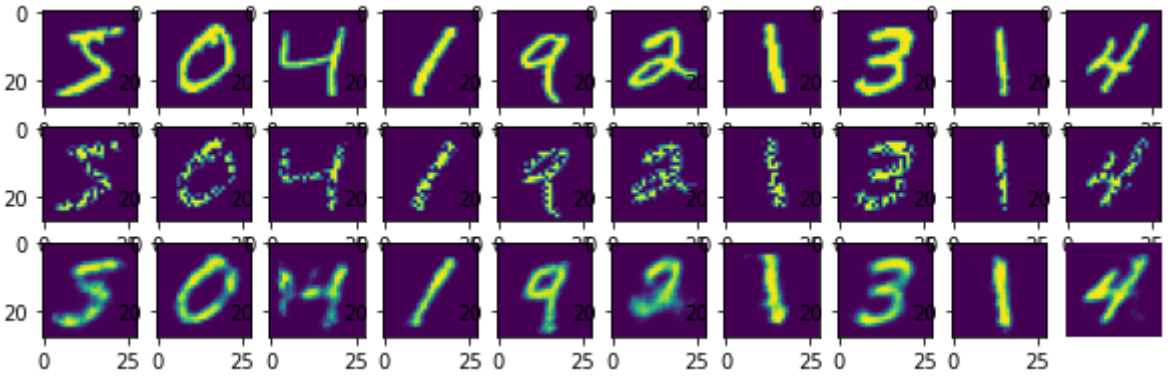
For each training epoch, take a batch of the training examples, apply the noise filter to each image, calculate the loss, and run the optimizer. At the end of each epoch, print the loss and confirm it decreases with each epoch.

In [10]:
```
#TODO: Implement Training Loop

for epoch in range(training_epochs):
    for data in train_dataloader:
        img, _ = data
        #img = img.to(device)
        img = img.view(img.size(0), -1)
        img_noise = masking_noise(img, 0.5)

        optimizer.zero_grad()
        outputs = model(img_noise)
        loss = criterion(outputs, img)
        loss.backward()
        optimizer.step()

    print('Epoch {} of {}, Train Loss: {:.3f}'.format(epoch+1, training_epo
```

```
Epoch 1 of 10, Train Loss: 0.189
Epoch 2 of 10, Train Loss: 0.163
Epoch 3 of 10, Train Loss: 0.159
Epoch 4 of 10, Train Loss: 0.152
Epoch 5 of 10, Train Loss: 0.146
Epoch 6 of 10, Train Loss: 0.157
Epoch 7 of 10, Train Loss: 0.150
Epoch 8 of 10, Train Loss: 0.149
Epoch 9 of 10, Train Loss: 0.140
Epoch 10 of 10, Train Loss: 0.147
```

Use the code below to plot the original image, the noisy image, and the reconstructed image. You can try tweaking the hyperparameters above to see if you can improve your accuracy!

In [11]:
```
x = next(iter(train_dataloader))[0].view(-1, 784)
x_noise = masking_noise(x, 0.5)
pred_img = model(x_noise)
f, a = plt.subplots(3, 10, figsize=(10, 3))
plt.axis('off')
for i in range(10):
    a[0][i].imshow(np.reshape(x[i], (28, 28)))
    a[1][i].imshow(np.reshape(x_noise[i], (28, 28)))
    a[2][i].imshow(np.reshape(pred_img[i].detach(), (28, 28)))
plt.show()
```



In [12]:
```
plt.axis('off')
f, a = plt.subplots(3, 10, figsize=(10, 3))
for i in range(10):
    a[0][i].imshow(np.reshape(x[i], (28, 28)), cmap='Greys',  interpolation
    a[1][i].imshow(np.reshape(x_noise[i], (28, 28)), cmap='Greys',  interpo
    a[2][i].imshow(np.reshape(pred_img[i].detach(), (28, 28)), cmap='Greys'
plt.show()
```