Assignment10

April 25, 2020

1 Knapp, Stephen

1.1 Deep Learning Assignment 10

[0]:
```
# libraries
import matplotlib.pyplot as plt
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torch.utils.data import Dataset, DataLoader
```

[0]:
```
# Python  3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn  0.20 is required
import sklearn
from sklearn.manifold import TSNE
assert sklearn.__version__ >= "0.20"

try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
    IS_COLAB = True
except Exception:
    IS_COLAB = False

# TensorFlow  2.0 is required
import tensorflow as tf
from tensorflow import keras
assert tf.__version__ >= "2.0"

if not tf.test.is_gpu_available():
    print("No GPU was detected. LSTMs and CNNs can be very slow without a GPU.")
    if IS_COLAB:
```

```
        print("Go to Runtime > Change runtime and select a GPU hardware␣
↪accelerator.")

# Common imports
import tensorflow as tf
from tensorflow import keras
from keras.layers import Activation, Dense, Input
from keras.layers import Conv2D, Flatten
from keras.layers import Reshape, Conv2DTranspose
from keras.models import Model
from keras import backend as K
assert tf.__version__ >= "2.0"
import numpy as np

import os

# to make this notebook's output stable across runs
np.random.seed(42)
tf.random.set_seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "autoencoders"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

[0]:

```
# settings to ensure reproducibility
seed = 42
torch.manual_seed(seed)
torch.backends.cudnn.benchmark = False
torch.backends.cudnn.deterministic = True
```

[0]:
```python
# plot image function
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")
```

[0]:
```python
# round function
def rounded_accuracy(y_true, y_pred):
    return keras.metrics.binary_accuracy(tf.round(y_true), tf.round(y_pred))
```

[0]:
```python
def show_reconstructions(model, images=X_valid, n_images=5):
    reconstructions = model.predict(images[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(images[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])
```

1.2 Problem 1

[3]:
```python
#load MNIST dataset with keras
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.mnist.
 ↪load_data()
X_train_full = X_train_full.astype(np.float32) / 255
X_test = X_test.astype(np.float32) / 255
X_train, X_valid = X_train_full[:-5000], X_train_full[-5000:]
y_train, y_valid = y_train_full[:-5000], y_train_full[-5000:]
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```

[88]:
```python
# denoising autoencoder with encoding layer size 128
tf.random.set_seed(42)
np.random.seed(42)

denoising_encoder128 = keras.models.Sequential([
    keras.layers.GaussianNoise(0.2, input_shape=[28, 28]),
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(256, activation="selu"),
    keras.layers.Dense(128, activation="selu")
])

denoising_decoder128 = keras.models.Sequential([
    keras.layers.Dense(256, activation="selu", input_shape=[128]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
```

```
])

denoising_ae128 = keras.models.Sequential([denoising_encoder128,␣
 ↪denoising_decoder128])
denoising_ae128.compile(loss="binary_crossentropy", optimizer=keras.optimizers.
 ↪SGD(lr=1.5),
                      metrics=[rounded_accuracy])
history = denoising_ae128.fit(X_train, X_train, epochs=10,
                            validation_data=[X_valid, X_valid])
```

```
Epoch 1/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.1496 -
rounded_accuracy: 0.9392 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 2/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.1032 -
rounded_accuracy: 0.9648 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 3/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0947 -
rounded_accuracy: 0.9697 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 4/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0907 -
rounded_accuracy: 0.9721 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 5/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0882 -
rounded_accuracy: 0.9737 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 6/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0865 -
rounded_accuracy: 0.9747 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 7/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0852 -
rounded_accuracy: 0.9755 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 8/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0841 -
rounded_accuracy: 0.9762 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 9/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0833 -
rounded_accuracy: 0.9767 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 10/10
```

```
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0827 -
rounded_accuracy: 0.9771 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
```

[96]:

```
# denoising autoencoder with encoding layer size 64
tf.random.set_seed(42)
np.random.seed(42)

denoising_encoder64 = keras.models.Sequential([
    keras.layers.GaussianNoise(0.2, input_shape=[28, 28]),
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(256, activation="selu"),
    keras.layers.Dense(64, activation="selu")
])
denoising_decoder64 = keras.models.Sequential([
    keras.layers.Dense(256, activation="selu", input_shape=[64]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
denoising_ae64 = keras.models.Sequential([denoising_encoder64,␣
 ↪denoising_decoder64])
denoising_ae64.compile(loss="binary_crossentropy", optimizer=keras.optimizers.
 ↪SGD(lr=1.5),
                      metrics=[rounded_accuracy])
history = denoising_ae64.fit(X_train, X_train, epochs=10,
                            validation_data=[X_valid, X_valid])
```

```
Epoch 1/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1570 -
rounded_accuracy: 0.9348 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 2/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1097 -
rounded_accuracy: 0.9609 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 3/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1003 -
rounded_accuracy: 0.9662 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 4/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0962 -
rounded_accuracy: 0.9686 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 5/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0935 -
rounded_accuracy: 0.9702 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 6/10
```

```
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0918 -
rounded_accuracy: 0.9712 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 7/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0905 -
rounded_accuracy: 0.9720 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 8/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0896 -
rounded_accuracy: 0.9725 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 9/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.0889 -
rounded_accuracy: 0.9729 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 10/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.0883 -
rounded_accuracy: 0.9732 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
```

[97]:

```
# denoising autoencoder with encoding layer size 32
tf.random.set_seed(42)
np.random.seed(42)

denoising_encoder32 = keras.models.Sequential([
    keras.layers.GaussianNoise(0.2, input_shape=[28, 28]),
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(128, activation="selu"),
    keras.layers.Dense(32, activation="selu")
])
denoising_decoder32 = keras.models.Sequential([
    keras.layers.Dense(128, activation="selu", input_shape=[32]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
denoising_ae32 = keras.models.Sequential([denoising_encoder32,␣
 ↪denoising_decoder32])
denoising_ae32.compile(loss="binary_crossentropy", optimizer=keras.optimizers.
 ↪SGD(lr=1.5),
                       metrics=[rounded_accuracy])
history = denoising_ae32.fit(X_train, X_train, epochs=10,
                             validation_data=[X_valid, X_valid])
```

```
Epoch 1/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1795 -
rounded_accuracy: 0.9212 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 2/10
```

```
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1340 -
rounded_accuracy: 0.9464 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 3/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1261 -
rounded_accuracy: 0.9508 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 4/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1219 -
rounded_accuracy: 0.9531 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 5/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1180 -
rounded_accuracy: 0.9553 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 6/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1155 -
rounded_accuracy: 0.9568 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 7/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1137 -
rounded_accuracy: 0.9578 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 8/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.1122 -
rounded_accuracy: 0.9587 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 9/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1110 -
rounded_accuracy: 0.9593 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 10/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1101 -
rounded_accuracy: 0.9599 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
```

[95]:

```
# denoising autoencoder with encoding layer size 16
tf.random.set_seed(42)
np.random.seed(42)

denoising_encoder16 = keras.models.Sequential([
    keras.layers.GaussianNoise(0.2, input_shape=[28, 28]),
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(128, activation="selu"),
    keras.layers.Dense(16, activation="selu")
])
denoising_decoder16 = keras.models.Sequential([
```

```
    keras.layers.Dense(128, activation="selu", input_shape=[16]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
denoising_ae16 = keras.models.Sequential([denoising_encoder16,␣
 ↪denoising_decoder16])
denoising_ae16.compile(loss="binary_crossentropy", optimizer=keras.optimizers.
 ↪SGD(lr=1.5),
                       metrics=[rounded_accuracy])
history = denoising_ae16.fit(X_train, X_train, epochs=10,
                             validation_data=[X_valid, X_valid])
```

```
Epoch 1/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.1986 -
rounded_accuracy: 0.9093 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 2/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.1630 -
rounded_accuracy: 0.9289 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 3/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1552 -
rounded_accuracy: 0.9334 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 4/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1512 -
rounded_accuracy: 0.9357 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 5/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1485 -
rounded_accuracy: 0.9372 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 6/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1465 -
rounded_accuracy: 0.9384 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 7/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.1449 -
rounded_accuracy: 0.9393 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 8/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.1437 -
rounded_accuracy: 0.9400 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
Epoch 9/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.1425 -
rounded_accuracy: 0.9407 - val_loss: 0.0000e+00 - val_rounded_accuracy:
```

```
0.0000e+00
Epoch 10/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.1397 -
rounded_accuracy: 0.9424 - val_loss: 0.0000e+00 - val_rounded_accuracy:
0.0000e+00
```

[0]:
```python
def show_reconstructions(model, images=X_valid, n_images=5):
    reconstructions = model.predict(images[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])
```

[0]:
```python
def show_originals(images=X_valid, n_images=5):
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(images[image_index])
```

[0]:
```python
def show_noise(model, images=X_valid, n_images=5):
    reconstructions = model.predict(images[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(images[image_index])
```

[0]:
```python
examples = np.array([X_valid[1], X_valid[8], X_valid[4], X_valid[10],␣
 ↪X_valid[2]])
```

[0]:
```python
print("Originals")
show_originals(examples)
plt.show()

tf.random.set_seed(42)
np.random.seed(42)

print("Noise Added")
noise = keras.layers.GaussianNoise(0.2)
show_noise(denoising_ae128, noise(examples, training=True))
plt.show()

print("Coding Layer Size 128")
show_reconstructions(denoising_ae128, images=examples)
plt.show()

print("Coding Layer Size 64")
show_reconstructions(denoising_ae64, images=examples)
```

```
plt.show()

print("Coding Layer Size 32")
show_reconstructions(denoising_ae32, images=examples)
plt.show()

print("Coding Layer Size 16")
show_reconstructions(denoising_ae16, images=examples)
plt.show()
```

Originals

Noise Added

Coding Layer Size 128

Coding Layer Size 64

Coding Layer Size 32

Coding Layer Size 16

The images above show that the highest quality is the largest coding layer and worst quality is the smallest coding layer. Essentially it's a trade-off between storage space and quality. The "2" seems to compress the worst which is no easily read in the size 16 layer. The "8" also starts to blur significantly as the layer size decreases and the "4" begins to look like a "9" and the "5" starts to look like an "8". Visually it appears that the size 64 coding layer seems to be the lowest quality that is still easily readible by a human. considering we started with a 784 size vector this constitutes a 91.8% reduction is size.

1.3 Problem 2

[0]:
```
batch_size = 512
epochs = 20
learning_rate = 1e-3
```

[0]:
```
class AddGaussianNoise(object):
    def __init__(self, mean=0., std=1.):
```

```
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        return tensor + torch.randn(tensor.size()) * self.std + self.mean

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={1})'.format(self.
 ↪mean, self.std)
```

[0]:
```
#load MNIST dataset with torch:
transform = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                            AddGaussianNoise(0., 0.5)])

transform_original = torchvision.transforms.Compose([torchvision.transforms.
 ↪ToTensor()])

train_dataset = torchvision.datasets.MNIST(
    root="~/torch_datasets", train=True, transform=transform, download=True
)

train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True
)
```

[0]:
```
class AE(nn.Module):
    def __init__(self, **kwargs):
        super().__init__()
        self.encoder_hidden_layer = nn.Linear(
            in_features=kwargs["input_shape"], out_features=128
        )
        self.encoder_output_layer = nn.Linear(
            in_features=128, out_features=32
        )
        self.decoder_hidden_layer = nn.Linear(
            in_features=32, out_features=128
        )
        self.decoder_output_layer = nn.Linear(
            in_features=128, out_features=kwargs["input_shape"]
        )

    def forward(self, features):
        activation = self.encoder_hidden_layer(features)
        activation = torch.relu(activation)
        code = self.encoder_output_layer(activation)
        code = torch.relu(code)
        activation = self.decoder_hidden_layer(code)
```

```
        activation = torch.relu(activation)
        activation = self.decoder_output_layer(activation)
        reconstructed = torch.relu(activation)
        return reconstructed
```

[0]:
```
#  use gpu if available
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device = torch.device("cpu")

# create a model from `AE` autoencoder class
# load it to the specified device, either gpu or cpu
model = AE(input_shape=784).to(device)

# create an optimizer object
# Adam optimizer with learning rate 1e-3
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# mean-squared error loss
criterion = nn.MSELoss()
```

[0]:
```
#Load original images

test_dataset_original = torchvision.datasets.MNIST(
    root="~/torch_datasets", train=False, transform=transform_original,␣
 ↪download=True
)

test_loader_original = torch.utils.data.DataLoader(
    test_dataset_original, batch_size=32, shuffle=False, num_workers=4
)

#Load noisy images

test_dataset = torchvision.datasets.MNIST(
    root="~/torch_datasets", train=False, transform=transform, download=True
)

test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=32, shuffle=False, num_workers=4
)
```

[0]:
```
for epoch in range(epochs):
    loss = 0
    for batch_features, _ in train_loader:
        # reshape mini-batch data to [N, 784] matrix
        # load it to the active device
        batch_features = batch_features.view(-1, 784).to(device)
```

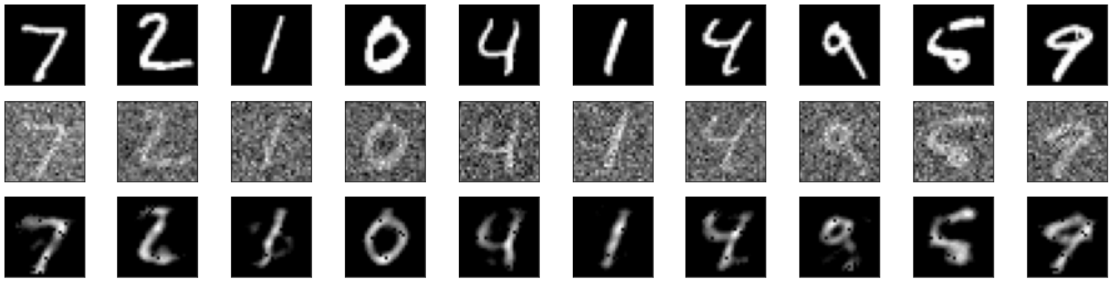
```python
        # reset the gradients back to zero
        # PyTorch accumulates gradients on subsequent backward passes
        optimizer.zero_grad()

        # compute reconstructions
        outputs = model(batch_features)

        # compute training reconstruction loss
        train_loss = criterion(outputs, batch_features)

        # compute accumulated gradients
        train_loss.backward()

        # perform parameter update based on current gradients
        optimizer.step()

        # add the mini-batch training loss to epoch loss
        loss += train_loss.item()

    # compute the epoch training loss
    loss = loss / len(train_loader)

    # display the epoch training loss
    print("epoch : {}/{}, loss = {:.6f}".format(epoch + 1, epochs, loss))
```

```
epoch : 1/20, loss = 0.302612
epoch : 2/20, loss = 0.271790
epoch : 3/20, loss = 0.267258
epoch : 4/20, loss = 0.265509
epoch : 5/20, loss = 0.264495
epoch : 6/20, loss = 0.264049
epoch : 7/20, loss = 0.263690
epoch : 8/20, loss = 0.263366
epoch : 9/20, loss = 0.263205
epoch : 10/20, loss = 0.263201
epoch : 11/20, loss = 0.262915
epoch : 12/20, loss = 0.262980
epoch : 13/20, loss = 0.262897
epoch : 14/20, loss = 0.262881
epoch : 15/20, loss = 0.262740
epoch : 16/20, loss = 0.262584
epoch : 17/20, loss = 0.262570
epoch : 18/20, loss = 0.262661
epoch : 19/20, loss = 0.262626
epoch : 20/20, loss = 0.262480
```

[0]:
```python
test_examples = None

with torch.no_grad():
    for batch_features in test_loader:
        batch_features = batch_features[0]
        test_examples = batch_features.view(-1, 784)
        reconstruction = model(test_examples)
        break
```

[0]:
```python
test_examples_original = None

with torch.no_grad():
    for batch_features in test_loader_original:
        batch_features = batch_features[0]
        test_examples_original = batch_features.view(-1, 784)
        break
```

[0]:
```python
with torch.no_grad():
    number = 10
    plt.figure(figsize=(16,4))
    for index in range(number):
        # display original image
        ax = plt.subplot(3, number, index + 1)
        plt.imshow(test_examples_original[index].numpy().reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        # display noisy image
        ax = plt.subplot(3, number, index + 1 + number)
        plt.imshow(test_examples[index].numpy().reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        # display reconstruction
        ax = plt.subplot(3, number, index + 1 + 2*number )
        plt.imshow(reconstruction[index].numpy().reshape(28, 28))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.show()
```



```
[0]: class AEC(nn.Module):
         def __init__(self, **kwargs):
             super().__init__()
             self.encoder_hidden_layer = nn.Linear(
                 in_features=kwargs["input_shape"], out_features=128
             )
             self.encoder_output_layer = nn.Linear(
                 in_features=128, out_features=32
             )

         def forward(self, features):
             activation = self.encoder_hidden_layer(features)
             activation = torch.relu(activation)
             code = self.encoder_output_layer(activation)
             code = torch.relu(code)
             return code
```

```
[0]: #  use gpu if available
     # device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     device = torch.device("cpu")

     # create a model from `AE` autoencoder class
     # load it to the specified device, either gpu or cpu
     model = AEC(input_shape=784).to(device)

     # create an optimizer object
     # Adam optimizer with learning rate 1e-3
     optimizer = optim.Adam(model.parameters(), lr=1e-3)

     # mean-squared error loss
     criterion = nn.MSELoss()
```

```
[0]: for epoch in range(epochs):
         loss = 0
         for batch_features, _ in train_loader:
             # reshape mini-batch data to [N, 784] matrix
```

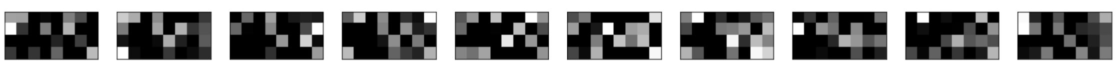
```
        # load it to the active device
        batch_features = batch_features.view(-1, 784).to(device)

        # reset the gradients back to zero
        # PyTorch accumulates gradients on subsequent backward passes
        optimizer.zero_grad()

        # compute reconstructions
        outputs = model(batch_features)

        # compute training reconstruction loss
        # train_loss = criterion(outputs, batch_features)

        # compute accumulated gradients
        # train_loss.backward()

        # perform parameter update based on current gradients
        optimizer.step()

        # add the mini-batch training loss to epoch loss
        loss += train_loss.item()

    # compute the epoch training loss
    loss = loss / len(train_loader)

    # display the epoch training loss
    print("epoch : {}/{}, loss = {:.6f}".format(epoch + 1, epochs, loss))
```

```
epoch : 1/20, loss = 0.261284
epoch : 2/20, loss = 0.261284
epoch : 3/20, loss = 0.261284
epoch : 4/20, loss = 0.261284
epoch : 5/20, loss = 0.261284
epoch : 6/20, loss = 0.261284
epoch : 7/20, loss = 0.261284
epoch : 8/20, loss = 0.261284
epoch : 9/20, loss = 0.261284
epoch : 10/20, loss = 0.261284
epoch : 11/20, loss = 0.261284
epoch : 12/20, loss = 0.261284
epoch : 13/20, loss = 0.261284
epoch : 14/20, loss = 0.261284
epoch : 15/20, loss = 0.261284
epoch : 16/20, loss = 0.261284
epoch : 17/20, loss = 0.261284
epoch : 18/20, loss = 0.261284
epoch : 19/20, loss = 0.261284
```

```
epoch : 20/20, loss = 0.261284
```

[0]:
```
with torch.no_grad():
    for batch_features in test_loader:
        batch_features = batch_features[0]
        test_examples = batch_features.view(-1, 784)
        reconstruction = model(test_examples)
        break
```

[0]:
```
with torch.no_grad():
    number = 10
    plt.figure(figsize=(20, 4))
    for index in range(number):

        # display reconstruction
        ax = plt.subplot(1, number, index + 1)
        plt.imshow(reconstruction[index].numpy().reshape(4,8))
        plt.gray()
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.show()
```



1.4 Problem 3

[0]:
```
#Create sampling model layer for variational encoder:
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) * K.exp(log_var / 2) + mean
```

[0]:
```
#set of images to train the model on will only be 3's and 4's
three_four = np.array([X_valid[8], X_valid[4]])
```

[101]:
```
#set the seed for reproducability
tf.random.set_seed(42)
np.random.seed(42)

#set coding layer size
codings_size = 2

#set input size
inputs = keras.layers.Input(shape=[28, 28])
```

```
#create encoder
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="selu")(z)
z = keras.layers.Dense(100, activation="selu")(z)
codings_mean = keras.layers.Dense(codings_size)(z)
codings_log_var = keras.layers.Dense(codings_size)(z)
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.models.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])

#create decoder
decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
x = keras.layers.Dense(150, activation="selu")(x)
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.models.Model(inputs=[decoder_inputs],␣
↪outputs=[outputs])

#create autoencoder
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.models.Model(inputs=[inputs], outputs=[reconstructions])

#measure model training performance
latent_loss = -0.5 * K.sum(
    1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
    axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)

#compile the model, load and run it
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop",␣
↪metrics=[rounded_accuracy])
history = variational_ae.fit(three_four, three_four, epochs=25, batch_size=128)
```

```
Epoch 1/25
1/1 [==============================] - 0s 1ms/step - loss: 0.7019 -
rounded_accuracy: 0.5070
Epoch 2/25
1/1 [==============================] - 0s 876us/step - loss: 0.9155 -
rounded_accuracy: 0.2908
Epoch 3/25
1/1 [==============================] - 0s 927us/step - loss: 0.6810 -
rounded_accuracy: 0.5944
Epoch 4/25
1/1 [==============================] - 0s 801us/step - loss: 0.6467 -
```

```
rounded_accuracy: 0.7213
Epoch 5/25
1/1 [==============================] - 0s 824us/step - loss: 0.5723 -
rounded_accuracy: 0.7793
Epoch 6/25
1/1 [==============================] - 0s 844us/step - loss: 0.4792 -
rounded_accuracy: 0.8712
Epoch 7/25
1/1 [==============================] - 0s 841us/step - loss: 0.4471 -
rounded_accuracy: 0.9305
Epoch 8/25
1/1 [==============================] - 0s 787us/step - loss: 0.3723 -
rounded_accuracy: 0.8992
Epoch 9/25
1/1 [==============================] - 0s 1ms/step - loss: 0.3267 -
rounded_accuracy: 0.9279
Epoch 10/25
1/1 [==============================] - 0s 862us/step - loss: 0.2975 -
rounded_accuracy: 0.9094
Epoch 11/25
1/1 [==============================] - 0s 735us/step - loss: 0.2765 -
rounded_accuracy: 0.9426
Epoch 12/25
1/1 [==============================] - 0s 813us/step - loss: 0.2653 -
rounded_accuracy: 0.9254
Epoch 13/25
1/1 [==============================] - 0s 776us/step - loss: 0.2473 -
rounded_accuracy: 0.9522
Epoch 14/25
1/1 [==============================] - 0s 813us/step - loss: 0.2346 -
rounded_accuracy: 0.9643
Epoch 15/25
1/1 [==============================] - 0s 827us/step - loss: 0.2262 -
rounded_accuracy: 0.9643
Epoch 16/25
1/1 [==============================] - 0s 735us/step - loss: 0.2113 -
rounded_accuracy: 0.9732
Epoch 17/25
1/1 [==============================] - 0s 808us/step - loss: 0.1975 -
rounded_accuracy: 0.9802
Epoch 18/25
1/1 [==============================] - 0s 935us/step - loss: 0.1927 -
rounded_accuracy: 0.9815
Epoch 19/25
1/1 [==============================] - 0s 915us/step - loss: 0.1815 -
rounded_accuracy: 0.9834
Epoch 20/25
1/1 [==============================] - 0s 2ms/step - loss: 0.1706 -
```

```
rounded_accuracy: 0.9911
Epoch 21/25
1/1 [==============================] - 0s 815us/step - loss: 0.1757 -
rounded_accuracy: 0.9809
Epoch 22/25
1/1 [==============================] - 0s 1ms/step - loss: 0.1723 -
rounded_accuracy: 0.9821
Epoch 23/25
1/1 [==============================] - 0s 1ms/step - loss: 0.1626 -
rounded_accuracy: 0.9898
Epoch 24/25
1/1 [==============================] - 0s 2ms/step - loss: 0.1530 -
rounded_accuracy: 0.9904
Epoch 25/25
1/1 [==============================] - 0s 878us/step - loss: 0.1414 -
rounded_accuracy: 0.9955
```

[102]:
```
#confirm the model worked
show_reconstructions(variational_ae, images=three_four, n_images=2)
```

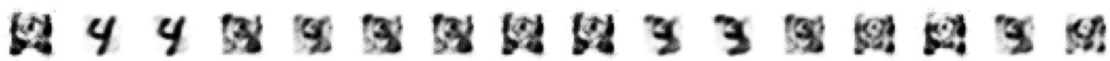
[0]:
```
#generate 16 images of 3's and 4's
count = 0
three_four_collection = np.empty(shape=[16,28,28])
while (count < 16):
  codings = tf.random.normal(shape=[1, codings_size])
  three_four_collection[count] = variational_decoder(codings).numpy()
  count = count + 1
```

[104]:
```
#compare images simulated above to after they are run through each denoising␣
↪autoencoder
print("Originals")
show_originals(three_four_collection, n_images=16)
plt.show()

print("Coding Layer Size 128")
show_reconstructions(denoising_ae128, images=three_four_collection, n_images=16)
plt.show()
```

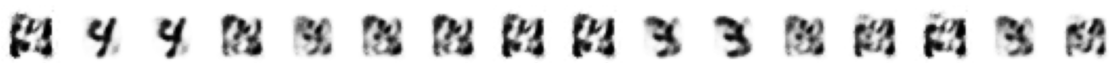
```python
print("Coding Layer Size 64")
show_reconstructions(denoising_ae64, images=three_four_collection, n_images=16)
plt.show()

print("Coding Layer Size 32")
show_reconstructions(denoising_ae32, images=three_four_collection, n_images=16)
plt.show()

print("Coding Layer Size 16")
show_reconstructions(denoising_ae16, images=three_four_collection, n_images=16)
plt.show()
```

Originals

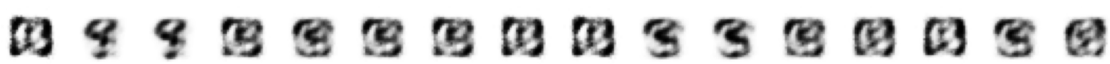
Coding Layer Size 128



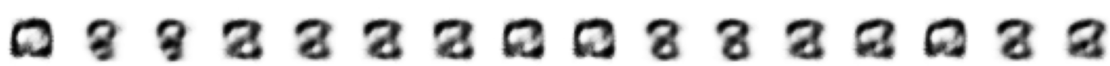
Coding Layer Size 64



Coding Layer Size 32



Coding Layer Size 16



[0]:

```python

```