**Name**: Skylar Furey

**Link to shared Colab Notebook**: *TODO*

General directions for coding assignments are as follows. *Please avoid any unnecessary loops*. Use broadcasting, and write vectorized, efficient code. *Unless otherwise instructed*, please only use low-level mathematical operations (for example, math and linear algebra operators in numpy and tensorflow), and avoid using high-level implementations (for example, high-level implementations of methods from scikit-learn and keras). If you are unsure about the usage of any library/function, please contact the instructor.

---

# Neural Networks (100 Points)

In this notebook we shall explore estimating neural networks for various tasks and in various frameworks.

## Data and Setup

First, it may help to enable GPUs for the notebook:

- Navigate to Edit→Notebook Settings
- select GPU from the Hardware Accelerator drop-down

Next, confirm that we can connect to the GPU with tensorflow.

*(Note, it is fine if you can not connect to GPU, it just might take a little longer to run.)*

```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  print('GPU device not found')
else:
  print('Found GPU at: {}'.format(device_name))
```

```
GPU device not found
```

Let's import additional packages of use.

```
%matplotlib inline
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats
import pandas as pd
```

## **[40 Points]** Neural Networks, the Easy Way!

First, we are going to explore training neural networks with keras, which provides a very simple interface to training models.

## **[20 points]** 2d Classification

**Reading data**

Let's use some helper functions to read in pregenerated 2d data for classification. `X` and `Y` will be `N x 2` input features and `N x 1` binary output labels, respectively.

```
def read_dataset(url):
  data = pd.read_csv(url).to_numpy()
  X = data[:, :-1]
  Y = data[:, -1, None]
  return X.astype(np.float32), Y.astype(np.float32)
```

```
url2 = 'https://raw.githubusercontent.com/lupalab/comp755_f21/main/hw3_2.csv'
X, Y = read_dataset(url2)
```

[8 points] **Setting up the model** Use `tf.keras.Sequential` to set up a *one hidden layer network, with 128 ReLU hidden units*, for Binary classification. Hint: the output activation depends on the loss that you'll use below.

```
seqmodel = tf.keras.Sequential([
    tf.keras.layers.Dense(128, activation='relu', input_shape=(X.shape[1],)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shap
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[8 points] **"Compiling" the Model** Next we will use the model's `.compile()` to specify the optimizer, loss, and accuracy metric to use with our classification problem.

```
from tensorflow.keras.metrics import BinaryAccuracy
from tensorflow.keras.losses import BinaryCrossentropy
seqmodel.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-2),
    loss=BinaryCrossentropy(),  # TODO: hint tf.keras.losses, be careful about whether or not you are working in logit
    metrics=[BinaryAccuracy()])  # TODO: hint tf.keras.metrics
```

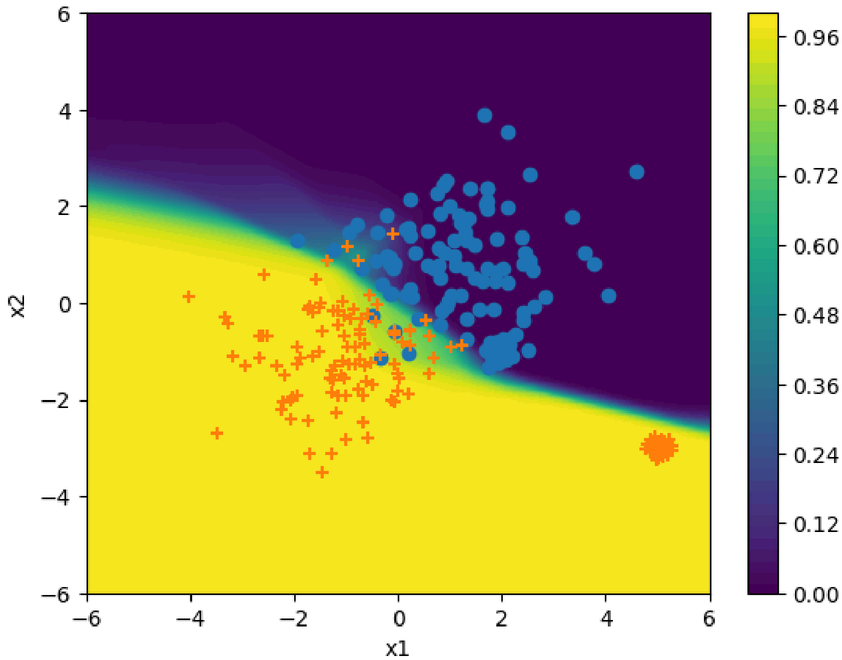
[4 points] **Train and Plot** Let's train the model and plot results.

```
def plot_grid(pred_func):
  # Reduce the number of grid points if there is a memory issue
  gridx = np.float32(np.linspace(-6.0, 6.0, 100))
  gridx1, gridx2 = np.meshgrid(gridx, gridx)
  g1 = np.reshape(gridx1, [-1, 1])
  g2 = np.reshape(gridx2, [-1, 1])
  func_vals = pred_func(np.concatenate((g1, g2), -1))
  plt.contourf(gridx1, gridx2, np.reshape(func_vals, gridx1.shape), 64)
  plt.xlabel('x1')
  plt.ylabel('x2')
  cbar = plt.colorbar()
  cbar.solids.set_edgecolor('face')
  plt.draw()
  return func_vals, g1, g2
```

```
# TODO fit the model on loaded data for 100 epochs
history = seqmodel.fit(X, Y, epochs=100, verbose=0)
```

```
plot_grid(seqmodel)  # TODO: use an appriate lambda function
plt.scatter(X[np.equal(Y[:,-1], 0), 0], X[np.equal(Y[:,-1], 0), 1])
plt.scatter(X[np.equal(Y[:,-1], 1), 0], X[np.equal(Y[:,-1], 1), 1], marker='+')
```

```
<matplotlib.collections.PathCollection at 0x7c560b22ca30>
```




## [20 points] MNIST Digit Classification

Now we will similarly use keras to train a neural network to classify MNIST digits. That is the input is a vector of pixel values and we are classifying digits ('0', '1', ..., '9').

```
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist


# load mnist dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# convert to one-hot vector
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# resize and normalize
x_train = np.reshape(x_train, [-1, 784])
x_train = x_train.astype('float32') / 255
x_test = np.reshape(x_test, [-1, 784])
x_test = x_test.astype('float32') / 255
```

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

```
x_train.shape
```

```
(60000, 784)
```

[8 points] **Setting up the model** Use `tf.keras.Sequential` to set up a *two hidden layer network, with 512 and 256 ReLU hidden units, respectively*, for digit classification. Hint: the output activation depends on the loss that you'll use below.

```
seqmodel = tf.keras.Sequential([
    tf.keras.layers.Dense(512, activation='relu', input_shape=(784,)),
    tf.keras.layers.Dense(256, activation='relu'),
```

```
    tf.keras.layers.Dense(10, activation='softmax')
])  # TODO
```

[8 points] **"Compiling" the Model** Next we will use the model's `.compile()` to specify the optimizer, loss, and accuracy metric to use with our classification problem.

```
seqmodel.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

[4 points] **Train and Plot** Let's train the model and plot results.

```
seqmodel.fit(x_train, y_train, batch_size=64, epochs=20)
```

```
Epoch 1/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 11s 11ms/step - accuracy: 0.7965 - loss: 0.8123
Epoch 2/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 13s 14ms/step - accuracy: 0.9437 - loss: 0.1969
Epoch 3/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 12s 13ms/step - accuracy: 0.9611 - loss: 0.1359
Epoch 4/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 12ms/step - accuracy: 0.9704 - loss: 0.1058
Epoch 5/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 11ms/step - accuracy: 0.9762 - loss: 0.0845
Epoch 6/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 22s 13ms/step - accuracy: 0.9812 - loss: 0.0677
Epoch 7/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 18s 10ms/step - accuracy: 0.9850 - loss: 0.0540
Epoch 8/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 11s 11ms/step - accuracy: 0.9884 - loss: 0.0446
Epoch 9/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 11s 11ms/step - accuracy: 0.9900 - loss: 0.0373
Epoch 10/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 19s 10ms/step - accuracy: 0.9913 - loss: 0.0330
Epoch 11/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 11s 11ms/step - accuracy: 0.9931 - loss: 0.0268
Epoch 12/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 11ms/step - accuracy: 0.9949 - loss: 0.0229
Epoch 13/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 11ms/step - accuracy: 0.9962 - loss: 0.0188
Epoch 14/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 11ms/step - accuracy: 0.9973 - loss: 0.0144
Epoch 15/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 10ms/step - accuracy: 0.9981 - loss: 0.0123
Epoch 16/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 11s 11ms/step - accuracy: 0.9979 - loss: 0.0113
Epoch 17/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 19s 10ms/step - accuracy: 0.9989 - loss: 0.0085
Epoch 18/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 11s 12ms/step - accuracy: 0.9990 - loss: 0.0075
Epoch 19/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 21s 12ms/step - accuracy: 0.9993 - loss: 0.0059
Epoch 20/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 19s 10ms/step - accuracy: 0.9995 - loss: 0.0047
<keras.src.callbacks.history.History at 0x7c5602cc6c80>
```

```
# TODO: consider using the model's ".evaluate" method
loss, acc = seqmodel.evaluate(x_test, y_test)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9786 - loss: 0.0769

Test accuracy: 98.2%
```

## **[60 Points]** Neural Networks, a Trickier Scenerio!

We've talked about custom losses with neural networks. Here, we are going to build a network that taskes in a subset of pixels (flattened as a vector) and predicts both: 1) the remaining pixels (regression); 2) the label for the image (classification). (See the plots below.) again, avoid any unnecessary loops, using broadcasting, and writing vectorized code.

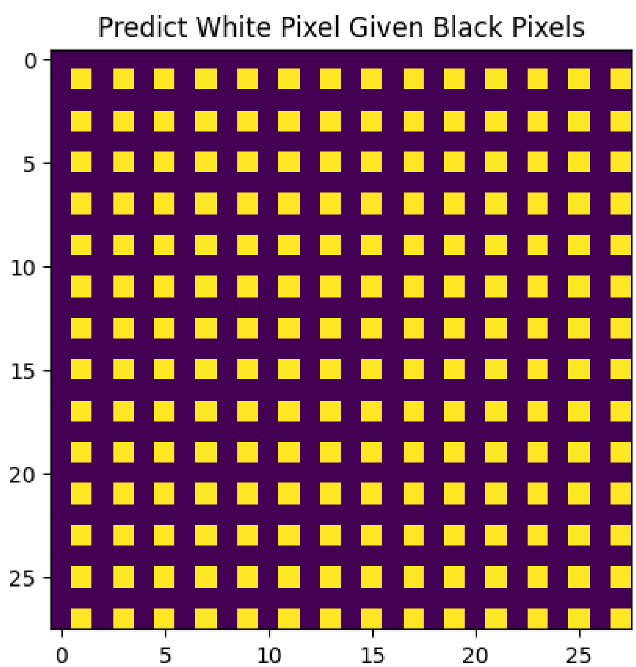
```
import tensorflow_datasets as tfds
(ds_train_org, ds_test_org), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)
```

```
Downloading and preparing dataset 11.06 MiB (download: 11.06 MiB, generated: 21.00 MiB, total: 32.06 MiB) to /root
Dl Completed...: 100%                                   5/5 [00:00<00:00, 10.12 file/s]
Dataset mnist downloaded and prepared to /root/tensorflow_datasets/mnist/3.0.1. Subsequent calls will reuse this d
```

See what pixels were have as inputs/outputs.

```
mask_indices = (np.mod(np.arange(28), 2)[:, None]>0)*(np.mod(np.arange(28), 2)[None, :]>0)
plt.imshow(mask_indices)
plt.title('Predict White Pixel Given Black Pixels')
```

```
Text(0.5, 1.0, 'Predict White Pixel Given Black Pixels')
```

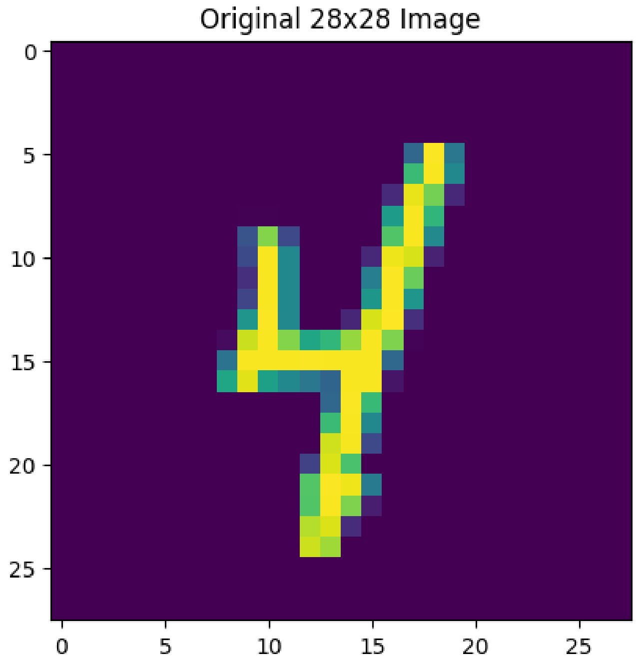


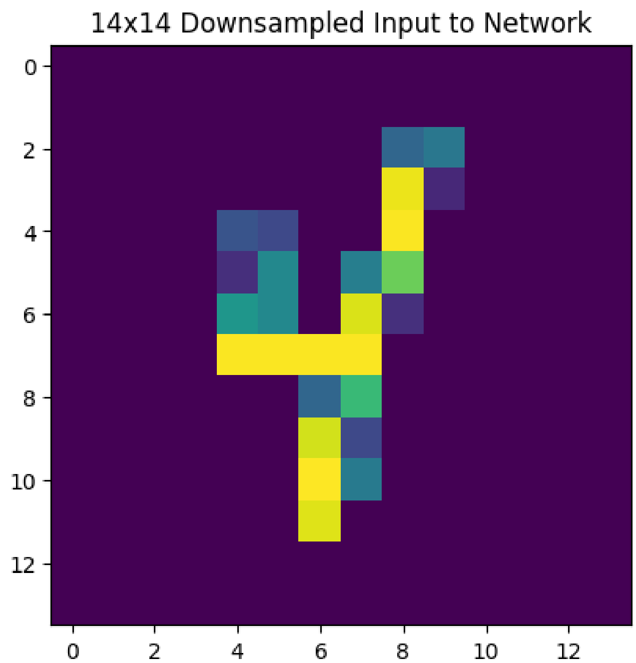

Making the datasets with inputs/outputs.

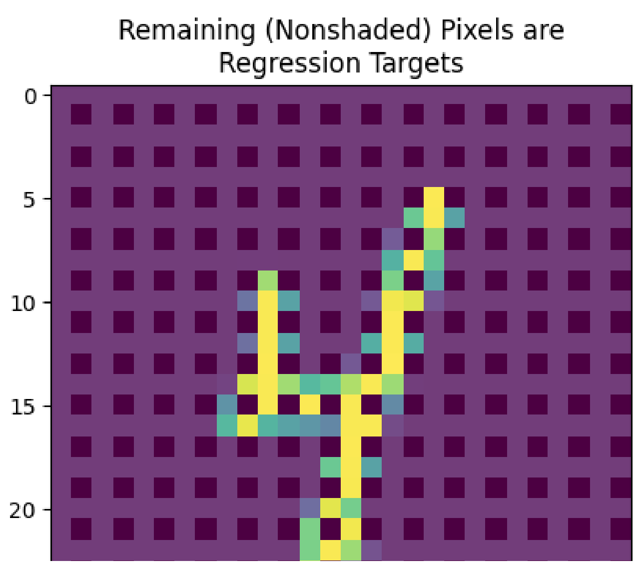
```
def normalize_img(image, label):
  """Normalizes images: `uint8` -> `float32`."""
  image_flat = tf.reshape(image, (-1,))
  return (
    tf.cast(image_flat[mask_indices.flatten()], tf.float32) / 255.,
    tf.cast(image_flat[~mask_indices.flatten()], tf.float32) / 255.,
```

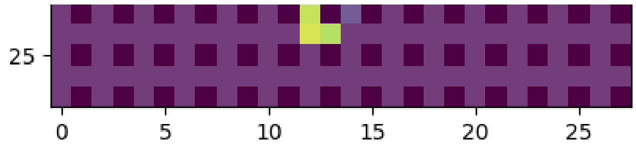
```
        label
      )

    ds_train = ds_train_org.map(
        normalize_img, num_parallel_calls=tf.data.AUTOTUNE)
    ds_test = ds_test_org.map(
        normalize_img, num_parallel_calls=tf.data.AUTOTUNE)


    plt.imshow(ds_train_org.as_numpy_iterator().next()[0])
    plt.title('Original 28x28 Image')
    plt.figure()
    plt.imshow(np.reshape(ds_train_org.as_numpy_iterator().next()[0][mask_indices, 0], (14, 14)))
    plt.title('14x14 Downsampled Input to Network')
    plt.figure()
    plt.title('Remaining (Nonshaded) Pixels are\nRegression Targets')
    plt.imshow(ds_train_org.as_numpy_iterator().next()[0]*(~mask_indices[:, :, None]))
    plt.imshow(mask_indices[:, :, None], alpha=0.25, cmap='Reds')
```

```
<matplotlib.image.AxesImage at 0x7c5600d14700>
```






$\color{red}{[\textbf{14 points}]}$ Let's define the neural network.

```
N_batch = 64  # batch size
```

```
tf.keras.backend.clear_session()
```

```
# nnet should be keras model that has 2 hidden layers with 256 hidden units
# and *linear* output layer (YOU NEED TO FIGURE OUT THE SIZE FOR OUTPUT).
# Hint: the last 10 outputs are used as logits for classification, the first
#       outputs corresponed to the regression target plotted above
# Hint: use keras.models.Sequential([...])
nnet = keras.models.Sequential([
    tf.keras.layers.Dense(256, activation='linear'),
    tf.keras.layers.Dense(256, activation='linear'),
    tf.keras.layers.Dense(598, activation='linear')
])

nnet.build([N_batch, 14*14])
```

$\color{red}{[\textbf{22 points}]}$ Let's implement the loss for batches.

```
def loss(X, Ypix, Ylbls, nnet):
  """
  Args:
    X: N_batch x 14*14 matrix of downsampled inputs
    Ypix: N_batch x ?? matrix of upsampling target pixels
    Ylbls: N_batch length array of integers indicating class labels
  Returns:
    loss: scalar of mse_regression_loss + classification_loss for batch
  """
  nnet_output = nnet(X)
  pix_output = nnet_output[:, :-10]
  logit_output = nnet_output[:, -10:]
  # mse_loss should be a mean of the squared error of the upsample estimates,
  # averaged over all instances (and outputs) in the batch
  mse_loss = tf.reduce_mean(tf.square(Ypix - pix_output))  # TODO
  # class_loss should be a mean of the cross entropy loss of the
  # averaged over all instances in the batch
  # hint: use (sparse_softmax_cross_entropy_with_logits)
  class_loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=Ylbls, logits=logit_output))  # TO
  return mse_loss + class_loss

def grad(X, Ypix, Ylbls, nnet):
  with tf.GradientTape() as tape:
    loss_value = loss(X, Ypix, Ylbls, nnet)

  return tape.gradient(loss_value, nnet.weights)
```

We will now train this network with stochastic mini-batches 'by hand'.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
```

```
nepochs = 25  # Could probably do better with more epochs, but should suffice
```

```
for i in range(nepochs):
  for B_X, B_Ypix, B_Ylbls in ds_train.batch(N_batch, drop_remainder=True):
    grads = grad(B_X, B_Ypix, B_Ylbls, nnet)
    optimizer.apply_gradients(zip(grads, nnet.weights))
  print("Loss at Epoch {:03d}: {:.3f}".format(
      i, loss(B_X, B_Ypix, B_Ylbls, nnet)))
```

```
Loss at Epoch 000: 0.330
Loss at Epoch 001: 0.270
Loss at Epoch 002: 0.248
Loss at Epoch 003: 0.235
Loss at Epoch 004: 0.226
Loss at Epoch 005: 0.218
Loss at Epoch 006: 0.213
Loss at Epoch 007: 0.208
Loss at Epoch 008: 0.205
Loss at Epoch 009: 0.202
Loss at Epoch 010: 0.200
Loss at Epoch 011: 0.198
Loss at Epoch 012: 0.197
Loss at Epoch 013: 0.196
Loss at Epoch 014: 0.195
Loss at Epoch 015: 0.194
Loss at Epoch 016: 0.193
Loss at Epoch 017: 0.193
Loss at Epoch 018: 0.192
Loss at Epoch 019: 0.192
Loss at Epoch 020: 0.191
Loss at Epoch 021: 0.191
Loss at Epoch 022: 0.191
Loss at Epoch 023: 0.191
Loss at Epoch 024: 0.191
```

**[12 points]** Let's visualize the output for an instance in the batch.

```
rind = np.random.randint(0, B_X.shape[0])
nnet_output = nnet(B_X)
pix_output = nnet_output[:, :-10]
logit_output = nnet_output[:, -10:]

plt.figure()
plt.title('Input, Predicted {}'.format(np.argmax(logit_output[rind])))
plt.imshow(np.reshape(B_X[rind], (14, 14)))

plt.figure()
plt.title('Reconstructed Upsampled Output')
x_recon = np.zeros((28*28), dtype=np.float32)
# TODO: fill x_recon so that it contains the 14*14 true pixel values
#       and the remaining corresponding output predicted pixels
x_recon[mask_indices.flatten()] = B_X[rind]
x_recon[~mask_indices.flatten()] = pix_output[rind]

x_recon[x_recon<0.0] = 0.0  # clean up predicted pixels lower than 0
x_recon[x_recon>1.0] = 1.0  # clean up predicted pixels high than 1
plt.imshow(np.reshape(x_recon, (28, 28)))

plt.figure()
plt.title('Reconstructed Upsampled Output')
x_true = np.zeros((28*28), dtype=np.float32)
# TODO: fill x_true so that it contains the 786 true pixel values
x_true[mask_indices.flatten()] = B_X[rind]
x_true[~mask_indices.flatten()] = B_Ypix[rind]

plt.imshow(np.reshape(x_true, (28, 28)))
```

```
<matplotlib.image.AxesImage at 0x7c5600c58610>
```