

---

# CS5785 Homework 4

---

The homework is generally split into programming exercises and written exercises.

This homework is due on **Nov 23rd, 2021 at 11:59 PM ET**. Upload your homework to Gradescope (Canvas->Gradescope). There are two assignments for this homework in Gradescope. Please note a complete submission should include:

1. A write-up as a single .pdf file. → Submit to “Homework 4- Write Up”. This file should contain your answers to the written questions **and exported pdf file / structured write-up of your answers to the coding questions** (which should include core codes, plots, outputs, and any comments / explanations). **We will deduct points if you do not do this.**
2. Source code for all of your experiments (AND figures) zipped into a single .zip file, in .py files if you use Python or .ipynb files if you use the IPython Notebook. If you use some other language, include all build scripts necessary to build and run your project along with instructions on how to compile and run your code. **If you use the IPython Notebook to create any graphs, please make sure you also include them in your write-up.** → Submit to “Homework 4- Code”.
3. You need to mark the pages of your submission to each question on Gradescope after submission, Gradescope should ask you to do that after you upload your write-up by default. **We will deduct points if you do not do this.**

The write-up should contain a general summary of what you did, how well your solution works, any insights you found, etc. On the cover page, include the class name, homework number, and team member names. You are responsible for submitting clear, organized answers to the questions. You could use online  $\text{\LaTeX}$  templates from [Overleaf](#), under “Homework Assignment” and “Project / Lab Report”. You could also use a  [\$\text{\LaTeX}\$  template we made](#), which contains useful packages for writing math equations and code snippet.

Please include all relevant information for a question, including text response, equations, figures, graphs, output, etc. If you include graphs, be sure to include the source code that generated them. Please pay attention to Canvas for relevant information regarding updates, tips, and policy changes. You are encouraged (but not required) to work in groups of 2.

## IF YOU NEED HELP

There are several strategies available to you.

- If you get stuck, we encourage you to post a question on the Discussions section of Canvas. That way, your solutions will be available to other students in the class.
- The professor and TAs offer office hours, which are a great way to get some one-on-one help.
- You are allowed to use well known libraries such as `scikit-learn`, `scikit-image`, `numpy`, `scipy`,

etc. for this assignment (including implementations of machine learning algorithms), unless we explicitly say that you cannot in a particular question. Any reference or copy of public code repositories should be properly cited in your submission (examples include Github, Wikipedia, Blogs).

## PROGRAMMING EXERCISES

### 1. Convolutional Neural Networks

We will work with the MNIST dataset for this question. This dataset contains 60,000 images of handwritten numbers from 0 to 9 and you need to recognize the handwritten numbers by building a CNN.

You will use the [Keras](#) library for this. You might have to [install](#) the library if it hasn't been installed already. Please check the installation by printing out the version of Keras inside the python shell as follows.

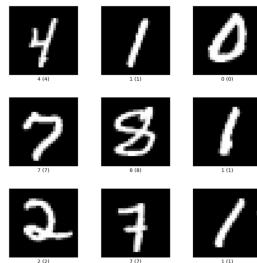
```
import keras
keras.__version__
```

The above lines will give you the current version of Keras you are using. Once this works, you can proceed with the assignment.

- (a) **Loading Dataset** For using this dataset, you will need to import mnist and use it as follows.

```
from keras.datasets import mnist
(train_X, train_Y), (test_X, test_Y) = mnist.load_data()
```

To verify that you have loaded the dataset correctly, try printing out the shape of your train and test dataset matrices. Also, try to visualize individual images in this dataset by using `imshow()` function in `pyplot`. Below are some example images from [the Tensorflow datasets catalog](#).



- (b) **Preprocessing** The data has images with 28 x 28 pixel values. Since we use just one grayscale color channel, you need to reshape the matrix such that we have a 28 x 28 x 1 sized matrix holding each input data-point in the training and testing dataset. The output variable can be converted into a one-hot vector by using the function [to\\_categorical](#) (make sure you import `to_categorical` from `keras.utils`). For example, if the output label for a given image is the digit 2, then the one-hot representation for this consists of a 10-element vector, where the element at index 2 is set to 1 and all the other elements are zero.

For preprocessing, scale the pixel values such that they lie between 0.0 and 1.0. Make sure that you use the appropriate conversion to float wherever required while scaling.

You can include all these steps into a single python function that loads your dataset appropriately. Once you finish this, visualize some images using `imshow()` function.

**(c) Implementation**

Now, to define a CNN model, we will use the [Sequential](#) module in Keras. We are providing you with the code for creating a simple CNN here. We use [Conv2D](#) (for declaring 2D convolutional networks), [MaxPooling2D](#) (for maxpooling layer), [Dense](#) (for densely connected neural network layers) and [Flatten](#) (for flattening the input for next layer).

```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.optimizers import SGD
def create_cnn():
    # define using Sequential
    model = Sequential()
    # Convolution layer
    model.add(
        Conv2D(32, (3, 3),
            activation='relu',
            kernel_initializer='he_uniform',
            input_shape=(28, 28, 1))
    )
    # Maxpooling layer
    model.add(MaxPooling2D((2, 2)))
    # Flatten output
    model.add(Flatten())
    # Dense layer of 100 neurons
    model.add(
        Dense(100,
            activation='relu',
            kernel_initializer='he_uniform')
    )
    model.add(Dense(10, activation='softmax'))
    # initialize optimizer
    opt = SGD(lr=0.01, momentum=0.9)
    # compile model
    model.compile(
        optimizer=opt,
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    return model
```

Specifically, we have added the following things in this code.

- i. A single convolutional layer with 3 x 3 sized window for computing the convolution, with 32 filters
- ii. Maxpooling layer with 2 x 2 window size.

- iii. Flatten resulting features to reshape your output appropriately.
- iv. Dense layer on top of this (100 neurons) with ReLU activation
- v. Dense layer with 10 neurons for calculating softmax output (Our classification result will output one of the ten possible classes, corresponding to our digits)

After defining this model, we use Stochastic Gradient Descent (SGD) optimizer and cross-entropy loss to compile the model. We are using a learning rate of 0.01 and a momentum of 0.9 here. We have added this to the given code stub already. Please see that this code stub works for you. Try to print `model.layers` in your interactive shell to see that the model is generated as we defined.

(d) **Training and Evaluating CNN**

Now we will train the network. You can see some examples [here](#). Look at the `fit()` and `evaluate()` methods.

You will call the `fit` method with a validation split of 0.1 (i.e. 10% of data will be used for validation in every epoch). Please use 10 epochs and a batch size of 32. When you evaluate the trained model, you can call the `evaluate` method on the test data-set. Please report the accuracy on test data after you have trained it as above. You can refer to the following while you write code for training and evaluating your CNN.

```
model.fit(train_x, train_y, batch_size=32, epochs=10,
          validation_split=0.1)
score = model.evaluate(test_x, test_y, verbose=0)
```

(e) **Experimentation**

- i. Run the above training for 50 epochs. Using `pyplot`, graph the validation and training accuracy after every 10 epochs. Is there a steady improvement for both training and validation accuracy?

For accessing the required values while plotting, you can store the output of the `fit` method while training your network. Please refer to the code below.

```
epoch_history = model.fit(train_x, train_y, batch_size=32, epochs=10,
                           validation_split=0.1)
# print validation and training accuracy over epochs
print(epoch_history.history['accuracy'])
print(epoch_history.history['val_accuracy'])
```

Make sure that your plot has a meaningful legend, title, and labels for X/Y axes.

- ii. To avoid over-fitting in neural networks, we can ‘**drop out**’ a certain fraction of units randomly during the training phase. You can add the following layer (before the dense layer with 100 neurons) to your model defined in the function `create_cnn`.

```
model.add(Dropout(0.5))
```

Make sure you import Dropout from `keras.layers`! Now, train this CNN for 50 epochs.

Graph the validation and train accuracy after every 10 epochs.

[This](#) tutorial might be helpful if you want to see more examples of dropout with Keras.

- iii. Add another convolution layer and maxpooling layer to the `create_cnn` function defined above (immediately following the existing maxpooling layer). For the additional convolution layer, use 64 output filters. Train this for 10 epochs and report the test accuracy.
- iv. We used a learning rate of 0.01 in the given `create_cnn` function. Using learning rates of 0.001 and 0.1 respectively, train the model and report accuracy on test data-set. To be sure, we are working with 2 convolution layers and training for 10 epochs while doing this experiment

(f) **Analysis**

- i. Explain how the trends in validation and train accuracy change after using the dropout layer in the experiments.
- ii. How does the performance of CNN with two convolution layers differ as compared to CNN with a single convolution layer in your experiments?
- iii. How did changing learning rates change your experimental results in part (iv)?

2. **Random Forests for Image Approximation** In this question, you will use random forest regression to approximate an image by learning a function,  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ , that takes *image*  $(x, y)$  *coordinates* as input and outputs *pixel brightness*. This way, the function learns to approximate areas of the image that it has not seen before.

- a. Start with an image of the Mona Lisa. If you don't like the Mona Lisa, pick another interesting image of your choice.

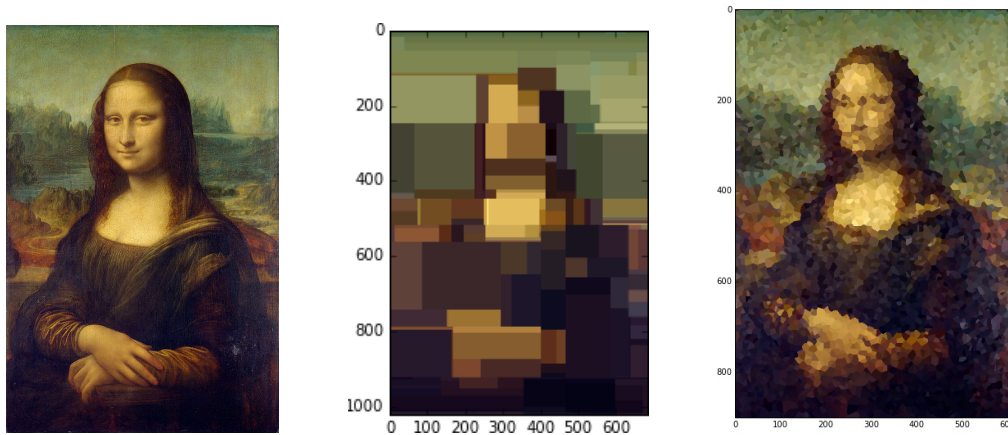


Figure 1: **Left:** <http://tinyurl.com/mona-lisa-small> *Mona Lisa*, Leonardo da Vinci, via Wikipedia. Licensed under Public Domain. **Middle:** Example output of a decision tree regressor. The input is a “feature vector” containing the  $(x, y)$  coordinates of the pixel. The output at each point is an  $(r, g, b)$  tuple. This tree has a depth of 7. **Right:** Example output of a  $k$ -NN regressor, where  $k = 1$ . The output at each pixel is equal to its closest sample from the training set.

**b. Preprocessing the input.** To build your “training set,” uniformly sample 5,000 random  $(x, y)$  coordinate locations.

- What other preprocessing steps are necessary for random forests inputs? Describe them, implement them, and justify your decisions. In particular, do you need to perform mean subtraction, standardization, or unit-normalization?

**c. Preprocessing the output.** Sample pixel values at each of the given coordinate locations. Each pixel contains red, green, and blue intensity values, so decide how you want to handle this. There are several options available to you:

- Convert the image to grayscale
- Regress all three values at once, so your function maps  $(x, y)$  coordinates to  $(r, g, b)$  values:  $f : \mathbb{R}^2 \rightarrow \mathbb{R}^3$
- Learn a different function for each channel,  $f_{Red} : \mathbb{R}^2 \rightarrow \mathbb{R}$ , and likewise for  $f_{Green}$ ,  $f_{Blue}$ .

Note that you may need to rescale the pixel intensities to lie between 0.0 and 1.0. (The default for pixel values may be between 0 and 255, but your image library may have different defaults.)

What other preprocessing steps are necessary for random regression forest outputs? Describe them, implement them, and justify your decisions.

**d.** To build the final image, for each pixel of the output, feed the pixel coordinate through the random forest and color the resulting pixel with the output prediction. You can then use `imshow` to view the result. (If you are using grayscale, try `imshow(Y, cmap='gray')` to avoid fake-coloring). You may use any implementation of random forests, but you should understand the implementation and you must cite your sources.

**e. Experimentation.**

- Repeat the experiment for a random forest containing a single decision tree, but with depths 1, 2, 3, 5, 10, and 15. How does depth impact the result? Describe in detail why.
- Repeat the experiment for a random forest of depth 7, but with number of trees equal to 1, 3, 5, 10, and 100. How does the number of trees impact the result? Describe in detail why.
- As a simple baseline, repeat the experiment using a  $k$ -NN regressor, for  $k = 1$ . This means that every pixel in the output will equal the nearest pixel from the “training set.” Compare and contrast the outlook: why does this look the way it does?
- Experiment with different pruning strategies of your choice.

**f. Analysis.**

- What is the decision rule at each split point? Write down the 1-line formula for the split point at the root node for one the trained decision trees inside the forest. Feel free to define any variables you need.
- Why does the resulting image look like the way it does? What shape are the patches of color, and how are they arranged?

## WRITTEN EXERCISES

1. **Maximum Margin Classifiers** Suppose we are given  $n = 7$  observations in  $p = 2$  dimensions. For each observation, there is an associated class label.

$X_1$	$X_2$	$Y$
3	4	Red
2	2	Red
4	4	Red
1	4	Red
2	1	Blue
4	3	Blue
4	1	Blue

- Sketch the observations and the maximum-margin separating hyperplane.
  - Describe the classification rule for the maximal margin classifier. It should be something along the lines of “Classify to Red if  $\beta_0 + \beta_1 X_1 + \beta_2 X_2 > 0$ , and classify to Blue otherwise.” Provide the values for  $\beta_0$ ,  $\beta_1$ , and  $\beta_2$ .
  - On your sketch, indicate the margin for the maximal margin hyperplane.
  - Indicate the support vectors for the maximal margin classifier.
  - Argue that a slight movement of the seventh observation would not affect the maximal margin hyperplane.
  - Sketch a hyperplane that separates the data, but is not the maximum-margin separating hyperplane. Provide the equation for this hyperplane.
  - Draw an additional observation on the plot so that the two classes are no longer separable by a hyperplane.
2. **Neural Networks as Function Approximators.** Design a feed-forward neural network to approximate the 1-dimensional function given in Fig. 2. The output should match exactly. How many hidden layers do you need? How many units are there within each layer? Show the hidden layers, units, connections, weights, and biases.

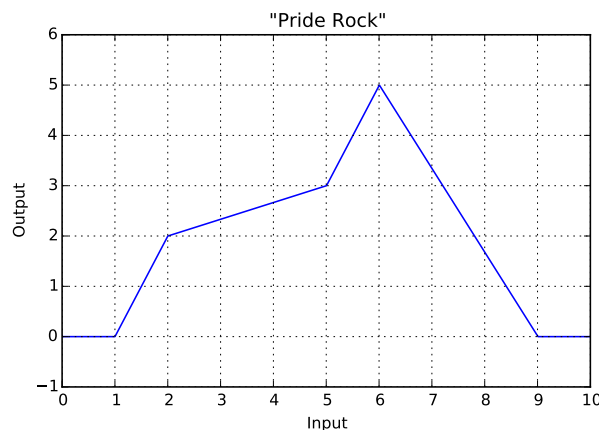


Figure 2: Example function to approximate using a neural network.

Use the ReLU nonlinearity for every unit. Every possible path from input to output must pass through the same number of layers. This means each layer should have the form

$$y_i = \sigma(W_i y_{i-1} + \beta_i), \quad (1)$$



where  $y_i \in \mathbb{R}^{d_i \times 1}$  is the output of the  $i$ th layer,  $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$  is the weight matrix for that layer,  $y_0 = x \in \mathbb{R}^{1 \times 1}$ , and the ReLU nonlinearity is defined as

$$\sigma(x) \triangleq \begin{cases} x & x \geq 0, \\ 0 & \text{otherwise} \end{cases}. \quad (2)$$

Hint 1: Try to express the input function as a sum of weighted and scaled ReLU units. Once you can do this, it should be straightforward to turn your final equation into a series of neural network layers with individual weights and biases.

Take a simpler case as example, if we 'straight' all the bends to the right of  $x = 5$  (5 included). The NN is simply 1 hidden layer with 2 units (+ 1 output layer). The input function is  $y = 2\sigma(x - 1) - \frac{5}{3}\sigma(x - 2)$

Hint 2: If you're not convinced neural networks can approximate this function, see <http://neuralnetworksanddeeplearning.com/chap4.html> for an example of the flavor of solution we're looking for. Make sure you understand the examples in the figures then just replace the sigmoid function with the ReLU. Please draw the answer as this article does. (If you rely on this source, cite it!) Keep in mind that your output must match the given input *exactly*.

Ordinarily, such a network could be learned via backpropagation, but we're instead asking you to figure out the weights to this simple network by hand.