1

# Project: Neural Networks

University of Oregon, DSCI/CIS 372

---

## Project Description

In this project you will implement and compare multiple variations of a neural network using a widely used library.

1. You will construct a deep feed-forward neural network in PyTorch which identifies images in the MNIST dataset following an explicit blueprint provided in the instructions. This model will treat the identification as a regression problem, to make it most comparable to the linear regression you did in the previous project.
2. You will construct a deep feed-forward neural network in PyTorch which identifies images in the MNIST dataset following an explicit blueprint provided in the instructions. This network will be a slight modification of the network created in 2(a), instead treating the identification as a classification problem.
3. You will construct a neural network with an architecture of your own design which classifies images in the MNIST dataset. This network will be a substantial modification of the network created in 2(b), but will continue to treat the identification of digits as a classification problem.

   It is strongly suggested that you construct a convolutional neural network for this task.
4. You will compare all of the implemented models and provide summary statistics about their efficacy.

The goal of this task is to get you very familiar with Pytorch and standard components of modern neural networks on a standard, balanced dataset that has already been treated with preprocessing best practices.

# Task 0 | Set Up

**This section is only for setting up your environment and does not have a deliverable.**

- Your submitted code will be run in a terminal. You may use whatever development environment which works for you (e.g., PyCharm), but make sure your code runs from a regular terminal before submission.
- Make sure you are running Python 3. You can verify this by executing the command in the terminal:

  `python -version`

  Another approach is to always run the command `python3`.

  WARNING: If you use the command `python3`, do so consistently. You must also install libraries used by `python3` with the command `pip3`.

  **DO NOT CONTINUE UNTIL YOU ARE RUNNING PYTHON 3**
- Install the required libraries for this project by running the following commands:

  `pip install scikit-learn` (Or, for Anaconda `conda install scikit-learn`)

  `pip install scipy` (Or, for Anaconda `conda install scipy`)

  `pip install numpy` (Or, for Anaconda `conda install numpy`)

  `pip install matplotlib` (Or, for Anaconda `conda install matplotlib`)
- Install Pytorch. We think the following commands will do:

  `pip install torch torchvision torchaudio` or

  `conda install pytorch torchvision torchaudio cpuonly -c pytorch`

  However, this site has the exact commands you will need: `https://pytorch.org/get-started/locally/`.

  WARNING: CUDA is a GPU programming interface. CUDA/GPUs are a hot topic, but avoid the hype for now. We highly recommend getting started with a CPU version to start.

  We are early in the era of GPU programming. Your neural network models will train more slowly on CPUs, but the time it will take you to debug GPU/CUDA related errors without having taken a parallel processing or advanced computer architecture class will likely take even longer. Since we're also learning a lot about neural networks at the same time, save yourself the headache and stick to CPUs for this class unless you're feeling exceptionally adventurous.

# Task 1 | Implement neural networks

## 1.1 Part 1: Create a deep neural network: regression

In this task, you will construct a deep feed-forward neural network in PyTorch which identifies images in the MNIST dataset following an explicit blueprint provided in the instructions. This model will treat the identification as a regression problem, to make it most comparable to the linear regression model constructed in the previous project.

The first step is to create a nearly identical experiment as the one you executed on the linear model, but for a new neural network model. We provide `exp00.py`, which is a fully working example of a neural net which trains on the abalone dataset.

You should make changes in `exp01.py`. The file `exp01.py` has a lot of starter code. All libraries that you need have already been included.

The file breaks a typical Python norm: there are two classes in the module.

One class is the experiment, which contains the kinds of methods we've seen before: load the data, and (most importantly) run, which knits everything together into a reproducible experiment. In this class, you will need to implement the method `load_train_test_data()` which loads the MNIST train and test datasets. You can reuse your codes in Project 0.5.

The other class is the `NeuralNet` class. There are four methods in this class which you need to implement:

- `__init__()`, the constructor, which instantiates the architecture (but doesn't train it), as well as instantiates the optimization and loss functions which are integral to the training algorithm.
- `_initialize_architecture()` which is a creates the actual network layers you will be training and predicting with.
- `fit(X,y)` which trains the underlying model.
- `predict(X)` which predicts an output based on newly seen samples after training is complete.

The neural network should have two linear feed-forward layers of size 128 and 64 nodes, and use ReLU activation functions. The network should train for at least 25 epochs. You can choose the batch size, although $\sqrt{\#training\_samples}$ is a common choice.

$$input(size: 28 \cdot 28) \rightarrow ReLU(Linear(size: 128)) \rightarrow ReLU(Linear(size: 64)) \rightarrow output(size: 1)$$

You should use Adam Optimization for your optimizer and Mean Squared Error for your loss function.

Report the error metrics for this model in "report.pdf."

## 1.2 Part 2: Create a deep neural network: classification

In this task, you will construct a deep feed-forward neural network in PyTorch which identifies images in the MNIST dataset following an explicit blueprint provided in the instructions. This network will be a slight modification of the network created in § 1.1, instead treating the identification as a classification problem.

This model will be developed in the file `exp02.py`. The model will be similar to the one created in § 1.1, with a twist: the network is to use softmax values and 1-hot encoding instead. This will require some preprocessing in the model class.

There are five methods in this class which you need to implement:

- `__init__()`, the constructor, which instantiates the architecture (but doesn't train it), as well as instantiates the optimization and loss functions which are integral to the training algorithm.
- `_initialize_architecture()` which is a creates the actual network layers you will be training and predicting with.
- `_convert_to_one_hot_encoding(y)` which converts a digit to a 1 hot encoded value.
- `fit(X,y)` which trains the underlying model.
- `predict(X)` which predicts an output based on newly seen samples after training is complete.

Most of these methods can be copy-pasted from § 1.1.

There are a few changes which will be made. Namely, the architecture should be more explicitly "classification," and not regression-like.

Namely, the architecture should look more like the following:

The neural network should have two linear feed-forward layers of size 128 and 64 nodes, and use ReLU activation functions. The network should train for at least 25 epochs. You can choose the batch size, although $\sqrt{\#training\_samples}$ is a common choice.

Schematic of the network layers:

$$input(size:28\cdot 28)\rightarrow ReLU(Linear(size:128))\rightarrow ReLU(Linear(size:64))\rightarrow softmaxOutput(size:10)$$

You should use Adam Optimization for your optimizer and Categorical Cross Entropy for your loss function.

The softmax output should correspond to a 1-hot encoded vector. While the architecture should internally be using these 1-hot encoded vectors, the externally facing interface should remain the same. That is, this model should be functionally swappable with the one in § 1.1 from the perspective of the run function in the Exp class. Therefore, when predicting values the argmax of the predicted 1-hot encoded vector should be returned.

Report the error metrics generated by the scikit learn classification reports for this model in "report.pdf."

## 1.3 Part 3: Create a neural network: your design

In this task, you will construct a neural network with an architecture of your own design which classifies images in the MNIST dataset. Create this network in a file called `exp03.py`.

This network should be a substantial modification of the network created in § 1.2, but will continue to treat the identification of digits as a classification problem.

You are required to include a convolutional+pooling component in the first layer(s) of your design. Note that part of this process will involve reshaping the input data to a shape of (#samples, 28, 28).

Report metrics for this model (using the scikit-learn classification report function) in "report.pdf."

Note whether your network design achieved higher or lower accuracy than the other models. There will be bonus points available for designs which are the most accurate in the class.

## 1.4 Part 4: Comparing networks

Include a short (~ 1 paragraph) written description which summarizes all of the network error/accuracy metrics of the networks you implemented. Which was the worst? Which was the best? Write your responses in `report.pdf`.

# Task 2 | What to Submit

You will submit two files:

`runnable_code.zip` and `reports_and_logs.zip`

## 2.1 Code

Put any executable code into a zip file called `runnable_code.zip`. The goal is that another human can unzip this folder and successfully run your code. You may put any special instructions or notes about how to run your code in a `readme.txt` file, also placed within `runnable_code.zip`.

## 2.2 Reports and Logs

Put any reports, logs, images, and output into a zip file called `reports_and_logs.zip`. This zip folder should contain at least the following:

- report.pdf

## 2.3 Upload to Canvas

Upload two separate files to Canvas: `runnable_code.zip` and `reports_and_logs.zip`.

# Appendix A | Grading Criteria

**This project will be primarily assessed along three axes:**

- **Validity**

  Programs should conform to specifications stated in the project description.

  Some components of the project will have example output provided. Your program's output must match the provided output for full points. If your output does not match identically for deterministic code (or within a reasonable margin for non-deterministic/random-based code) then expect a heavy penalty.

- **Style, Documentation, and Readability**

  Programs should be easy to read and understand. Coding well is writing well. Writing well is thinking well.

  To help us write quality code, we will use Pylint. Pylint is used in industry (Google, Amazon, etc.) to enforce Python's coding standard. Pylint also provides messages about style / readability / documentation flaws. You should run Pylint before submitting your work. Pylint assigns a score out of 10 to the code it processes. It can be very, very difficult to get a 10. We will consider 6/10 to be "full credit" for most work.

- **Design and Fluency**

  When one is learning a natural language like English, Spanish, Chinese, etc., we often go through a phase where our language is "technically correct," but may not seem right to a fluent speaker of the target language – in fact, our spoken or written speech as an (advanced) learner may still seem jarring or just plain weird at times to a fluent speaker. The same principle holds true for code.

  This lack of fluency we all sometimes experience when learning new topics isn't to be confused with creativity or pushing the limits in new directions. Rather, it's just part of acclimating to the norms of a new subject area and undergoing the learning process of becoming an expert in a particular skill.

  Coding is a highly creative and highly controlled processes. In making decisions in how to tackle the objectives outlined in the project tasks, you should keep in mind the following principles in order to maximize fluency:

  - Programs should be written with regard to efficiency of both human development time and code space.
  - Programs should be "elegantly" written. The "keep it simple" principle comes to mind.
  - Programs should consist of small, coherent, independent methods.
    I follow the two inches rule-of-thumb: if a method is longer than two inches on the screen, it should probably be broken up into more methods.

  Design and fluency is a category which is difficult to automatically grade, yet incredibly important to get feedback on. Come and talk in office hours if you are routinely missing points in this category.

# Rubric

---

**Part 1 (100 pts)**

exp01.py validity of __init__(): _____ of 10

exp01.py validity of _initialize_architecture(): _____ of 30

exp01.py validity of fit(): _____ of 30

exp01.py validity of predict(): _____ of 10

exp01.py readability: _____ of 6

exp01.py fluency: _____ of 4

metrics reported in report.pdf: _____ of 10

**Part 2 (45 pts)**

exp02.py _convert_to_one_hot_encoding() validity (& code changes elsewhere): _____ of 25

exp02.py readability: _____ of 6

exp02.py fluency: _____ of 4

metrics reported in report.pdf: _____ of 10

**Part 3 (45 pts)**

exp03.py validity – network uses convolution (-25pts if not): _____ of 0

exp03.py validity – the code is free of errors: _____ of 15

exp03.py validity – network does better than random chance: _____ of 10

exp03.py readability: _____ of 6

exp03.py fluency: _____ of 4

Metrics reported in report.pdf: _____ of 10

(Extra credit) exp03.py validity – accuracy is high: . . . . . _____ of 0

**Part 4 (10 pts)**

Brief comparison of networks in report.pdf: . . . . . _____ of 10

## Total: . . . . . _____ of 200