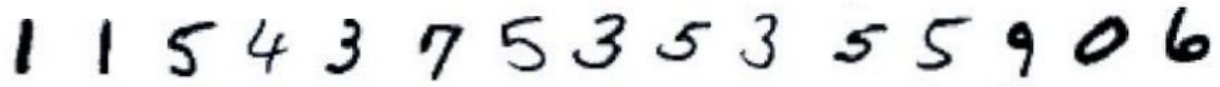
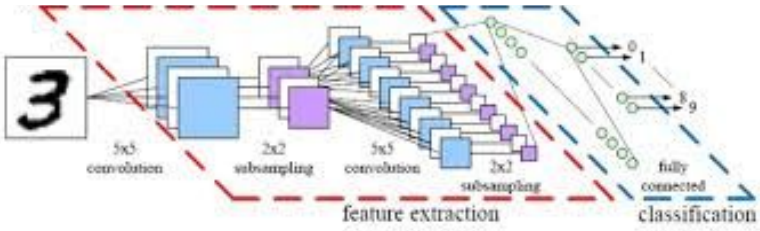
In this you will write your own code to train a convolutional neural network (CNN) for the task of hand-written digit recognition.





This is composed of two parts:

- Write the code for forward and backpropagation through a network, and implement the necessary functions for training. We have laid out a framework and provided initial code to get you started.
- Design and train a CNN to classify numbers from the MNIST dataset. You will have the chance to experiment with different architectures and training parameters to get experience understanding how these choices affect training time and performance of your network.

We are providing working code (as .p files) so if you cannot finish all of the implementations in part one, you will still be able to train a network for part two.

Part One:

The code you will have to write falls into two categories: specific layer implementations, and general functions for inference and training. What order you choose to write these functions is up to you, we have set this project up in a way that each part can be tested and verified completely independently. You can even skip ahead to writing the main training function and then dive right into part two. Though we recommend against this as writing the base code will provide you with a better understanding of what is going on when you set up and train your network.

Layer Functions:

The code for any single layer can be considered as a module completely independent from any other layers. These modules are assembled and arranged to create the full network that you will be training. We have structured the code for the layers such that they all follow the same format.

The function arguments are defined as follows:

input	**(** x **)**
	The input data to the layer function.
params	**(** W, b **)**
	Weight and bias information for the layer. For our purposes this is only used in the 'linear' and 'conv' layers.
hyper_params	Information describing the layer, for example the number of nodes or the size of the filters. These parameters get set when initializing the layer. Check out 'example_model_initialization.m' and 'init_layer.m' to get a better understanding of *params* and *hyper_params*.
backprop	Boolean stating whether or not to compute the output terms for backpropagation.
dv_output	**(** $\frac{dL}{dy}$ **)**
	The partial derivative of the loss with respect to each element in the output matrix. Only passed in when *backprop* is set to true.

You then must return three variables:

output	**(** y **)**
	The result of the function you are applying to your input data.
dv_input	**(** $\frac{dL}{dx}$ **)**
	The derivative of the loss with respect to the input. It is calculated by taking the derivative of the output with respect to the input ($\frac{dy}{dx}$) and multiplying by *dv_output*. This is the key component to understanding how the process of error backpropagation works. It is an application of the chain rule, but now in the context of functions that handle multi-dimensional matrices. *dv_input* has to be the same size as the input matrix, so this is a good guide when trying to think about what operation to apply to *dv_output*.
grad	**(** $\frac{dL}{dW}$ **,** $\frac{dL}{db}$ **)**
	The gradient term that you will use to update the weights defined in *params* and train your network. It is calculated in a similar manner to *dv_input*. Instead of the derivative of the loss with respect to the input, you want the derivative of the loss with respect to the weights. So, find $\frac{dy}{dW}$ and multiply by *dv_output* ($\frac{dL}{dy}$) to get the final weight gradient ($\frac{dL}{dW}$).

Here are the first layer functions you will implement:

Linear	$y = Wx + b$
Softmax	$y_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$

You will also be writing code for a loss function, which is set up slightly differently. You can look at the provided 'loss_euclidean.m' as an example.

Cross-entropy loss	$L(x, label) = -\frac{1}{N}\sum_{i=1}^{N} label_i * log(x_i)$

A quick thing to note about cross-entropy loss is that it is used to compare two probability distributions, so if you are given a correct class label you have to convert this into a vector to compare element-wise with some probability distribution x outputted by your softmax layer. This $label$ vector is simply a vector of all zeros except with a one at the correct class.

The final layer function to implement will be the convolutional layer. This may seem like a daunting layer to write code for, but both the forward and backwards calculations essentially boil down to some nested loops and calls to conv2. It is up to you to figure out exactly how to set this up. We will be asking that you do **valid** convolutions. If you want to incorporate padding or conv2 calls with 'same', have your code do those with a hyperparameter option, so that the basic code that we will be testing against uses 'valid'.

We have provided an automatic testing file to verify your layer code. Run 'check_layers' to see how your code does, it will output a simple 'pass' or 'fail' for the return values of each layer. One disclaimer, these are very basic tests, and while they will tell you if you are doing your calculations correctly they can in no way be considered a comprehensive evaluation of your code. Bugs may still be present, and there may exist corner cases that are not tested.

We have provided the layer code for pooling, RELU, and flattening. Pooling is in a .p file in case you want to try writing your own version, but it is not necessary. Most layer functions also include information at the top describing the expected dimensions of a particular matrix. For example, a four dimensional input might have the following description: "in_height * in_width * num_channels * batch_size". This is to help keep track of the various matrices you will be dealing with.

Inference and Training:

These functions will all be short, simple loops, but it is important for you to write these yourself to understand the mechanics of forward and backward propagation.

inference	Do a forward pass through the network and return the activations at each layer.
calc_gradient	Do backpropagation to determine the appropriate gradient at each layer. This will look a lot like your inference code, just looping in reverse order.
update_weights	Given your calculated gradients update the model appropriately. The argument *params* defines values like learning rate and weight decay.

Finally, you have to write your training function. This is the one piece of code that is not offered as a .p file, so you have to write it yourself.

The basic loop goes like this:

- Select a subset of your dataset to be a batch
- Run inference
- Calculate your loss
- Calculate your gradients
- Update the weights of your model
- Repeat

You have functions that do most of those tasks, so there is no trick or challenge here, just call the functions correctly. The point of writing the training function is that it will be your main interface when doing part two. You want to think about what to include to make training a network as nice of a process as possible. What information and updates do you want to display during training? How often? Another consideration is that the starting code we have provided is written to run for a set number of iterations, but you might want to do something a little more sophisticated. You could detect automatically when the loss has plateaued and choose to change your learning rate or stop training. Perhaps you want to stop training automatically after hitting a goal accuracy.

One final note, you will want to track training and test set loss as training progresses (you do not want to recompute the test set error every single iteration, but it is important to see how it improves in comparison to the training loss). You should store this information to be plotted after training.

Important: We do not require you to implement momentum, but with that said, momentum *significantly* speeds up training in this assignment. It is well worth it to introduce it to your code. Implementing it will not count as extra credit. Also, for additional speedup, check out the 'parfor' function to parallelize your code.

Part Two:

You now have all of the tools you need to build and train a neural network, it is time to get some experience designing your own.

Your first task is to train a network on MNIST. We have provided a function that loads the MNIST data into your MATLAB workspace. The dataset consists of 28x28 images of handwritten digits from 0 to 9 with their corresponding label (0 is labelled as 10 because MATLAB uses 1-indexing). There are helpful links at the end of this assignment, and one will take you to Yann LeCun's page for MNIST with a lot of good information.

This part of the assignment is open ended, it will not be too hard to achieve good accuracy, and the baseline threshold we expect you to hit is pretty generous. We will be grading more on the details in your report describing the decisions you made and why you made them. This could be as simple as describing that you have modeled exactly after LeNet-5 and found that it works to your expectations. That is sufficient for this assignment, but perhaps not very satisfying, so we will ask one last task of you.

After getting a baseline model trained and working, we want you to do a basic experiment. Explore one of the features listed below and do analysis on the impact it has on your network. What effects are there on training time? Performance? Does it lead to overfitting, or maybe underfitting? There is no shortage of possible questions to ask. Training a CNN well is often described as an “art”, and in order to master this craft it is important to have some intuition of the effects of these parameters. There are so many, we cannot ask you to explore them all, so choose one and try to do some basic quantitative analysis. (This means we expect to see a chart or table with numerical results, not just a paragraph vaguely describing what you have observed.)

We ask you to choose from the following options:

- Network depth, the effect of adding or removing layers
- Layer width, changing the size of the layers by adding/subtracting filters and nodes
- Extra features such as batch normalization, dropout, etc.

Grading Checklist and Report Instruction:

For part one:

- We will have a series of tests that automatically test whether your functions are written correctly, you can compare your code output with the provided .p files to verify your implementations yourself.
- Following functions will be tested:
    - Linear
    - Softmax
    - Crossentropy Loss
    - Convolutional Layer
    - Inference
    - Gradient Calculation
    - Weights Update
- You do not need to include any discussion of your code for part one unless you have decided to implement some extra feature.
- You need to submit all your codes to Canvas

For part two:

- Your report should detail the baseline architecture you used to train on MNIST and describe the decisions you made and why you made them.
- Include information on hyper parameters chosen for training and a plot showing loss across iterations. You should plot both training set and test set loss.
- Your baseline architecture should achieve at least 96% accuracy on test set. (In this assignment the test set is treated as a “validation set”, that is, you are allowed to try different hyperparameters on it).
- You should explore one of the options listed above based on your baseline model and include the quantitative analysis (tables or charts with numerical results).
- You need to submit your report to Gradescope.

Recommended Links:

- MNIST Database
- Gradient-Based Learning Applied to Document Recognition
- ImageNet Classification with Deep Convolutional Neural Networks