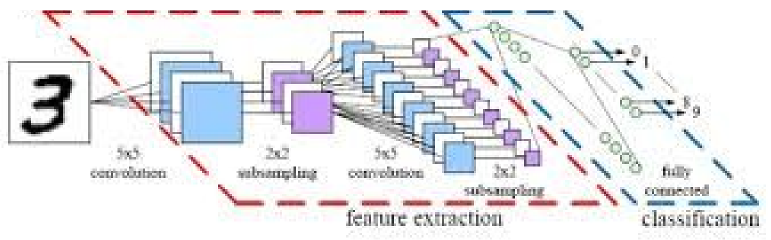
Homework: Convolutional Neural Networks

EECS 542 - Computer Vision, Fall 2017

Due: 9/25 11:59pm

Collaboration policy: Discussion with other students is allowed but coding and written reports must be done individually. In particular, over-the-shoulder coding is prohibited.

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



The homework consists 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 provide working code (as .p files). As a result, 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 to manage inference and training. What order you choose to write these functions is up to you, we have set this project up such that each part can be tested and verified independently.

Layer Functions:

Each layer is self-contained. That is, you can define a layer and verify that your implementation is correct without relying on any other part of the pipeline. Each layer is defined as a MATLAB function with a consistent format that we define here.

Layer inputs:

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.

Layer outputs:

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 ($\frac{dL}{dW}$).

The first layer functions you will implement:

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

Setting up the convolutional layer will prove a bit more challenging than the linear and softmax layers. Forward inference consists of correctly setting up nested loops with MATLAB's conv2 function. The main challenge is to determine the correct backprop operation. It is highly recommended that you work out a simple example by hand to get some insight into the calculation.

We will be asking that you do **valid** convolutions. If you want to incorporate some form of padding, use a hyperparameter option so that the baseline code that we evaluate on still performs a 'valid' convolution.

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.

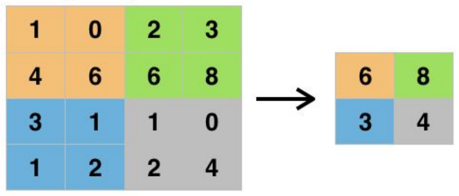
Cross-entropy loss $L(x, label) = -\frac{1}{N}\sum_{i=1}^{N} label_i * log(x_i)$

Note that the cross-entropy loss compares two probability distributions, so you must first convert a class label to a vector to perform a comparison with the output distribution x from the softmax layer. The $label$ should be a "one-hot vector" (a vector of all zeros with a one at the correct class).

Those layers will be enough to get you up and running, and you might want to move on to getting your training pipeline set up. In fact, it is possible to get good performance quickly with just a couple of linear layers. For additional exposure to other aspects of deep networks we ask that you implement several additional layers:

Leaky ReLU $y = x,\ if\ x \geq 0$
$0.01x,\ if\ x < 0$

Max pooling Take the max value in a sliding window.
Illustrated below with filter size 2 and stride 2



Batch normalization $y = \frac{x - mean(x)}{\sqrt{var(x) + e}} * w + b$,

where $var(x) = (x_i - mean(x))^2/N$ is the sample variance. We ask that the batch normalization layer is compatible with the conv layer, it is okay if it does not work with the output of the linear layer. The mean and variance are computed across the last two dimensions of *x* (the channel and batch dimensions).

Here, e is a small number for numerical stability, which we set to 10^{-5}. The parameters *w* and *b* are learned affine parameters.

We have provided two automatic testing files to verify your layer code. Run 'check_layers' and 'check_layers_2' to see whether your layers pass or fail. A disclaimer: these are basic tests, and while they will tell you if you are doing your calculations correctly, they are not a comprehensive evaluation of your code. Corner cases and bugs might exist that we do not test for. The input / output examples of batch normalization, max-pooling and leaky ReLU layers can be found in data/check_layers_2/ folder. We believe these examples, especially bn2.mat, will help you understand how these layers work.

We have provided the layer code for ReLU and flattening. Most layer functions 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 training loop goes:

```
    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.

You may want to track training and test set losses 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. 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. Your baseline model should not use batch normalization layers or leaky ReLU.

After getting a baseline model trained and working, we ask that you explore the effect of leaky ReLUs and batch normalization on the network. What effects are there on training time? Performance? Do they lead to overfitting, or maybe underfitting? There is no shortage of possible questions to ask. We expect to see a chart or table with numerical results, not just a paragraph vaguely describing what you have observed.

We ask that you perform the following two experiments:

- Simply replace all ReLUs with leaky ReLUs, and report the performance of your model under the same hyper-parameters (learning rate and weight decay).
- Add batch normalization to your model. You may need to slightly change the architecture. Batch normalization is usually applied right before nonlinearity layers.

There are plenty of additional details you may explore. However, they are not mandatory, and will not affect your final grade. Examples include:

- Network depth, the effect of adding or removing layers.
- Layer width, changing the size of the layers by adding/subtracting filters and nodes.

Grading Checklist and Report Instructions:

For part one, we will check that you have correctly implemented the following functions:

- Linear
- Softmax
- Cross-entropy loss
- Convolutional layer
- Max-pooling layer
- Batch normalization layer
- Leaky ReLU layer
- Inference
- Gradient Calculation
- Weight Update

(You can compare your output with that of the .p files to verify your implementations yourself)

You do not need to include any discussion of your code for part one unless you have decided to implement some extra feature. You must submit all of your code to Canvas (do not include trained models).

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 the 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).
- Include quantitative analysis (tables or charts with numerical results) for your baseline, Leaky-ReLU model, and batch-normalization model.

Recommended Links:

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