10-715 Advanced Introduction to Machine Learning: Homework 4
Neural Networks

Released: Wednesday, October 17, 2018
Due: 11:59 p.m. Wednesday, October 31, 2018
Last Updated: 24th October, 2018: 6:00 PM

Instructions

- **Late homework policy:** Homework is worth full credit if submitted before the due date, half credit during the next 48 hours, and zero credit after that.
- **Collaboration policy:** Collaboration on solving the homework is allowed. Discussions are encouraged but you should think about the problems on your own. When you do collaborate, you should list your collaborators! Also cite your resources, in case you got some inspiration from other resources (books, websites, papers). If you do collaborate with someone or use a book or website, you are expected to write up your solution independently. That is, close the book and all of your notes before starting to write up your solution. We will be assuming that, as participants in a graduate course, you will be taking the responsibility to make sure you personally understand the solution to any work arising from such collaboration.
- **Online submission:** You must submit your solutions online on Autolab (link: https://autolab.andrew.cmu.edu/courses/10715-f18/assessments). Please use LaTeX to typeset your solutions, and submit a single pdf called **hw4.pdf**.
- This homework involves a significant amount of coding, and requires a number of experiments to be run. **Start Early!**

1 Convolutional Neural Networks (CNN) [100 points]

Autograding environment We use Python 2.7.5 for autograding. To make sure that your program outputs the correct shapes and your saved `weights.npy` can be loaded on the server, please use the following command to install the numpy and scipy:

```
pip install 'numpy==1.7.1'
pip install 'scipy==0.12.1'
```

CNN We will begin by introducing the basic structure and building blocks of CNNs. Like ordinary neural network models, CNNs are made up of layers that have learnable parameters including weights and bias. Each layer takes the output from previous layer, performs some operations and produces an output. The final layer is typically a softmax function which outputs the probability of an image being in different classes. We optimize a objective function over the parameters of every layer and then use stochastic gradient descent (SGD) to update the parameters to train a model.

Depending on the operation in the layers, we can divide the layers into the following types:

1.1 Dense layer (fully connected layer)

As the name suggests, every output neuron of the inner product layer has full connection to the input neurons. See here for a detailed explanation. The output is the multiplication of the input with a weight matrix plus a bias offset, i.e.:

$$f(x) = Wx + b. \tag{1}$$

This is simply a linear transformation of the input. The weight parameter W and bias parameter b are learnable in this layer. The input x is a d dimensional vector, and W is an $n \times d$ matrix and b is n dimensional vector.

1.2 ReLU layer

We add nonlinear functions after the inner product layers to model the nonlinearity of real data. One of the activation functions found to work well in image classification is the rectified linear unit (ReLU):

$$f(x) = max(0, x). \tag{2}$$

There are many other activation functions such as the sigmoid and $\tanh$ function. See here for a detailed comparison among them. There is no learnable parameter in the ReLU layer.

The ReLU layer is sometimes combined with inner product layer as a single layer; here we separate them in order to make the code modular.

1.3 Convolution layer

See here for a detailed explanation of the convolution layer.

The convolution layer is the core building block of CNNs. Different from the inner product layer, each output neuron of a convolution layer is only connected to some input neurons. As the name suggests, in the convolution layer, we apply a convolution operation with filters on input feature maps (or images). Recall that in image processing, there are many types of kernels (filters) that can be used to blur, sharpen an image or to detect edges in an image. See the wiki page if you are not familiar with the convolution operation. In a convolution layer, the filter (or kernel) parameters are learnable and we want to adapt the filters to data. There is also more than one filter at each convolution layer. The input is a three dimensional tensor, rather than a vector as in the inner product layer. We represent the input feature maps (it can be the output from a previous layer, or an image from the original data) as a three dimensional tensor with height h, width w and channel c (for a color image, it has three channels).

Fig. 1 shows the detailed convolution operation. The input is a feature map, i.e., a three dimensional tensor with size $h \times w \times c$. Assume the (square) window size is k, then each filter is of shape $k \times k \times c$ since we use the filter across all input channels. We use n filters in a convolution layer, then the dimension of the filter parameter is $k \times k \times c \times n$. Another two hyper-parameters in the convolution operation, are the padding size p and stride step s. Zero padding is typically used; after padding, the first two dimensions of input feature maps are $(h + 2p) \times (w + 2p)$. The stride s controls the step size of the convolution operation. As Fig. 1 shows, the red square on the left is a filter applied locally on the input feature map. We multiply the filter weights (of size $k \times k \times c$) with a local region of the input filter map and then sum the product to get the output feature map. Hence, the first two dimensions of the output feature map are $[(h + 2p - k)/s + 1] \times [(w + 2p - k)/s + 1]$. Since we have n filters in a convolution layer, the output feature map is of size $[(h + 2p - k)/s + 1] \times [(w + 2p - k)/s + 1] \times n$.

Besides the filter weight parameters, we also have the filter bias parameters which is a vector of size n, that is, we add one scalar to each channel of the output feature map.

1.4 Pooling layer

It is common to use pooling layers after convolutional layers to reduce the spatial size of feature maps. Pooling layers are also called down-sample layers. With a pooling layer, we can extract more salient feature maps and reduce the number of parameters of CNNs to reduce over-fitting. Like a convolution layer, the pooling operation also acts locally on the feature maps, and there are also several hyper parameters that controls the pooling operation including the windows size k and stride s. The pooling operation is typically applied independently within each channel of the input feature map. There are two types of pooling operations: max pooling and average pooling. For max pooling, for each window of size $k \times k$ on the input feature map, we take the max value of the window. For average pooling, we take the average of the window. We can also use zero padding on the input feature maps. If the padding size is p, the first two dimension of output feature map are $[(h + 2p - k)/s + 1] \times [(w + 2p - k)/s + 1]$. This is the same as in the convolutional layer. Since pooling operation is channel-wise independent, the output feature map channel size is the same as the input feature map channel size.

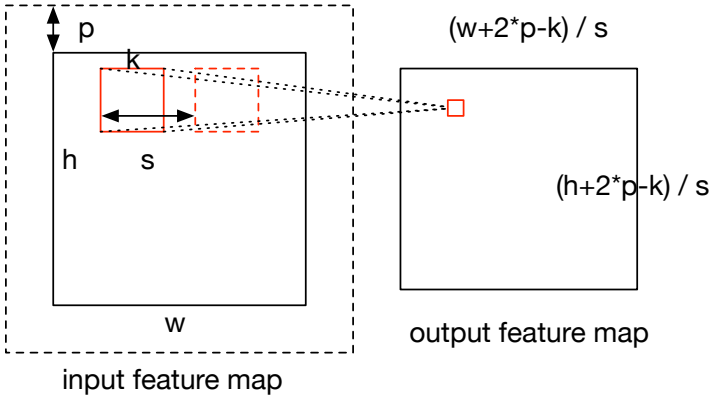

Figure 1: convolution layer

Refer to here for a more detailed explanation of the pooling layer. For simplicity, you do not need to implement padding in this homework. In other words, you can assume that $p = 0$.

1.5 Loss layer

For classification, we use a softmax function to assign probability to each class given the input feature map:

$$p = \text{softmax}(Wx + b). \tag{3}$$

In training, we know the label given the input image, hence, we want to minimize the negative log probability of the given label:

$$l = -\log(p_j), \tag{4}$$

where j is the label of the input. This is the objective function we would like to optimize.

2 LeNet

Having introduced the building components of CNNs, we now introduce the architecture of LeNet.

Layer Type	Configuration
DATA	input size: $28 \times 28 \times 1$
CONV	$k = 5, s = 1, p = 0, n = 20$
POOLING	MAX, $k = 2, s = 2, p = 0$
CONV	$k = 5, s = 1, p = 0, n = 50$
POOLING	MAX, $k = 2, s = 2, p = 0$
Dense	$n = 500$
RELU	
Dense	$n = 10$
LOSS	

Table 1: Architecture of LeNet

The architecture of LeNet is shown in Table. 1. The name of the layer type explains itself. LeNet is composed of interleaving of convolution layers and pooling layers, followed by an inner product layer and finally a loss layer. This is the typical structure of CNNs.

Refer to here for the architecture of other CNNs.

3 Implementation

The basic framework of CNN is already finished and you need to help fill some of the empty functions. Here is an overview of all the files provided to you.

- `commons.py`: Defines the `Variable` class which stores the parameters' gradients and values.
- `conv_layer.py`: Defines the `ConvLayer` class for implemented convolutional layers.
- `data_layer.py`: Takes the input data and returns the data with the correct format. You should first run `python data_loader.py` to convert the data file to the desired format.
- `data_loader.py`: Provides helpful functions to load the data.
- `dense_layer.py`: Defines the `DenseLayer` and `ReLULayer` that you need to implement.
- `layers.py`: Imports all the defined layers.
- `le_net.py`: At the bottom of `le_net.py`, we define the structure of LeNet. It consists of the 8 layers, the configuration of layer `i` is specified in structure `layers[i]`. The order of layers and the configuration parameters for each layer are shown in Table 1. The class `LeNet` defines the LeNet. It takes the configuration of the network structure, the input data (`data`) and label (`labels`) and does feed forward and backward propagation, returns the cost and gradient w.r.t all the parameters (`grad`).
- `loss_layer.py`: Defines the cross entropy loss function.
- `main.py` is the main file for you to specify a network structure and train a model.

- `optimizer.py`: Defines the SGD and SGD with momentum.
- `pool_layer.py`: Defines the max pooling layer.

Note that the parameters of each layer is stored as `params`. `params["w"]` is the weight matrix and `params["b"]` is the bias (Note that both these are `Variable` objects) . The `init` function will figure out the shapes of all parameters and give them an initial value according to the layer configuration. We use uniform random variables within given ranges to initialize the parameters. For all layers, the `forward` function does the feed forward (i.e given the outputs of the layer below, computes the values of the current layers). The `backward` function does the backward propagation (ie given derivatives of the cost wrt the outputs, computes the derivatives wrt the layer paramters, if they exist, as well as the derivatives wrt the inputs).

Also provided are two helper functions in the `BaseConvLayer` class.

- `im2col_conv` returns a list of pixels in each feature window, given an input image and layer details (such as padding, stride, and output dimensions). Given an image with multiple channels, `im2col_conv` function moves a feature window over the input image and places all pixels within the feature window in a separate column in the `col` variable of dimension [`k x k x c, h_out x w_out`]. You might need to reshape your input to this function and also reshape its output, so please read the function to get a better understanding of how the values are stored.
- `col2im_conv` returns a list of the gradients at each pixel of the input image, given a list of gradients for each pixel in each feature window. Given a data structure col, the function `col2im_conv` takes each column of `col` and adds it at the appropriate feature window of image `im` which is to be returned by the function. In this sense, `col2im_conv` is the reciprocal of the function `im2col_conv` described earlier. You might need to reshape your input to this function and also reshape its output, so please read the function to get a better understanding of how the values are stored.

The following two figures provide a graphical description of the `im2col_conv` and `col2im_conv`. In Figure 2, the feature window moves over the input image in a row major fashion (red $\mapsto$ green $\mapsto$ brown $\mapsto \dots$). The content of the feature window will be reshaped to a column and put into a matrix `M` with `k x k x c` rows and `h_out x w_out` columns in the same order as they are retrieved from the image.

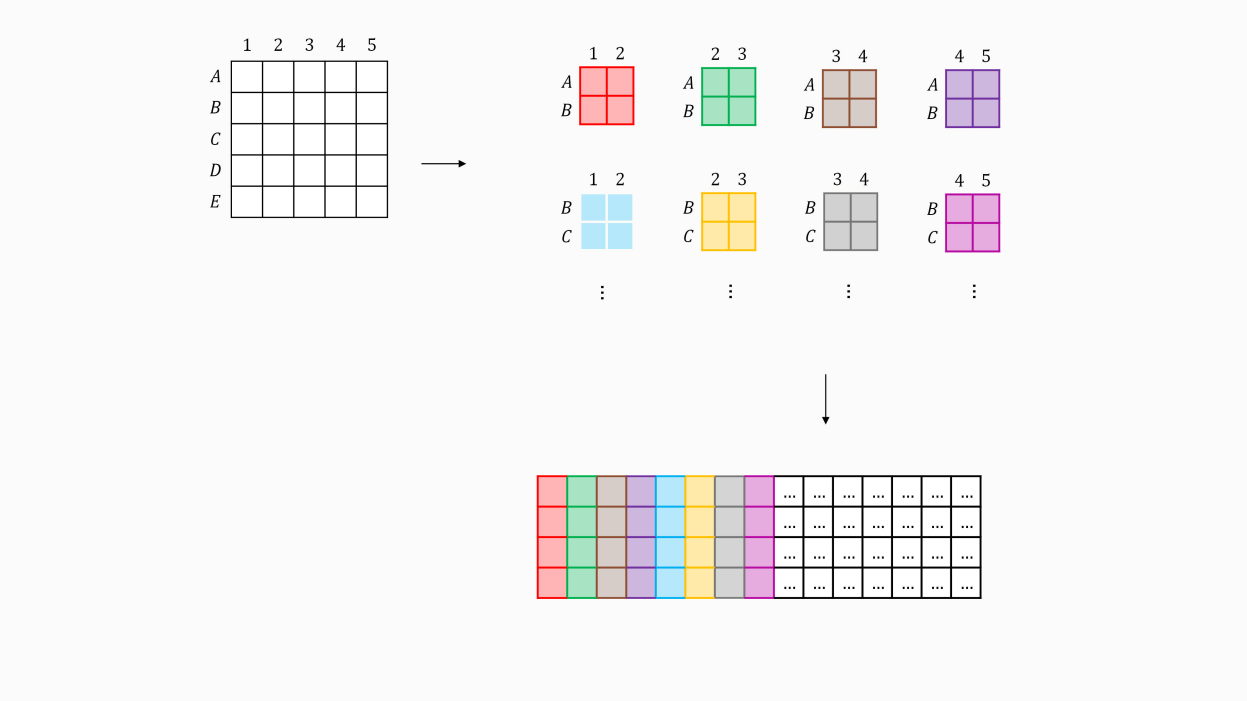

Figure 2: im2col_conv: `s = 1, k = 2, c = 1, h_out = w_out = 5`

In Figure 3, an empty image is first initialized. Then the input column is reshaped to a matrix `M` with `k x k x c` rows and `h_out x w_out` columns. Each column in the matrix is first reshaped to a feature window of `k x k x c`, and added to the image in the row major order.

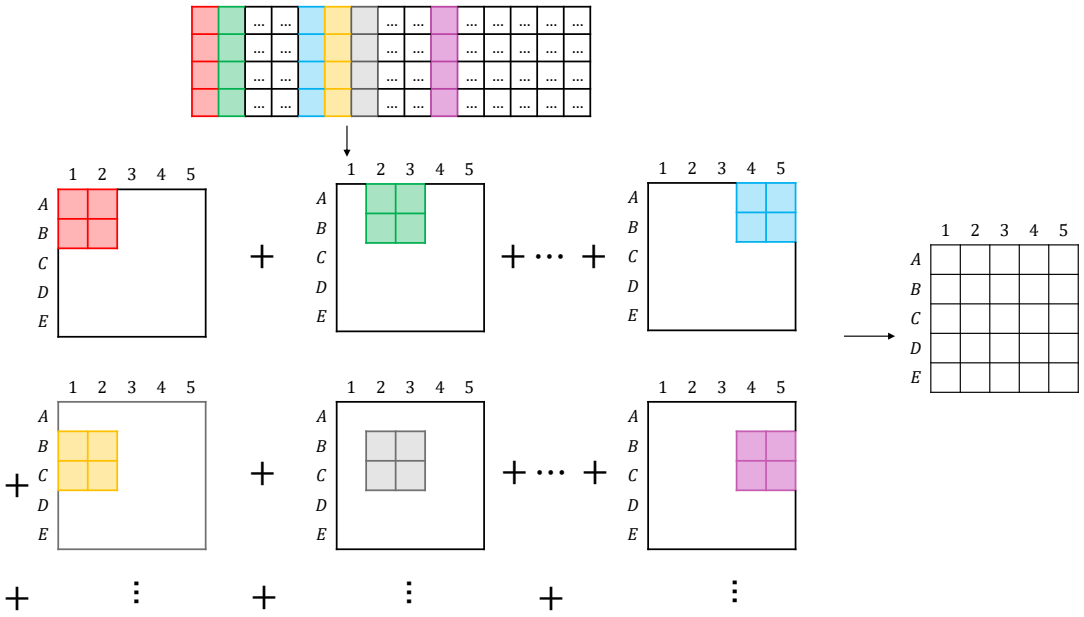

Figure 3: col2im_conv: `s = 1, k = 2, c = 1, h_out = w_out = 5`

3.1 Data structure

We use the following python classes and `OrderedDicts` to store most of our input and output values. You could refer to a particular field of the structure using `structure.field` for classes, and (DictName["key"]) from an `OrderedDict`. Here is a list of structures you'll encounter

- `inputs`, `outputs`: data and shape of the feature maps
  - `outputs["height"]` and `inputs["height"]` store the height of feature maps
  - `outputs["width"]` and `inputs["width"]` and store the width of feature maps
  - `outputs["channel"]` and `inputs["channel"]` store the channel size of feature maps. Channel is the number of RGB channels for the input layer, generally the depth dimension for other layers.
  - `batch_size` in `main.py` stores the batch size of feature maps. At every iteration, we take a random mini batch of the training data and call `LeNet` to get the gradient of the parameters, the `batch_size` is the size of the mini batch we used.
  - `outputs["data"]`, `inputs["data"]` stores actual data of a feature map. It is a matrix with size [`batch_size, height x width x channel`]. If necessary, you can reshape it to [`batch_size, height, width, channel`] during your computation, but remember to reshape it back to a two dimensional matrix at the end of each function.
- `output_grads`: gradient of the feature maps
  - `output_grads["grad"]` stores the gradient w.r.t the data matrix. This is used in backward propagation. It has the same shape as `outputs["data"]`.
- `param`: model parameters including the weights, biases, and their gradient

- `param["w"].value` and `param["w"].grad` stores the weight matrix of each layer and its gradient.
- `param["b"].value` and `param["b"].grad` stores the bias of each layer and its gradient.

3.2 Feed Forward

Convolution layer: `conv_layer.py` has been implemented for you.

1. **[10 points]** Pooling layer: You need to implement the `forward` function in the pooling layer (`pool_layer.py`). You can assume the padding is 0 here. As explained before, there are two type of poolings, max pooling and average pooling. We only test max pooling in this homework. Hence, in the code, the `act_type` can only take value `max`.
2. **[5 points]** ReLU layer: You need to implement the `forward` function in the ReLU layer (`dense_layer.py`). The function interfaces are clearly explained in the code.
3. **[10 points]** Dense layer: You need to implement the `forward` function in the Dense layer (`dense_layer.py`).

3.3 Backward Propagation

Denote layer i as a function f_i with parameters w_i, then the final loss is computed as:

$$l = f_I(w_I, f_{I-1}(w_{I-1}, ...)). \tag{5}$$

We want to optimize l over the parameters of each layer. We can use the chain rule to get the gradient of the loss w.r.t the parameters of each layer. Let the output of each layer be $h_i = f_i(w_i, h_{i-1})$. Then the gradient w.r.t w_i is given by:

$$\frac{\partial l}{\partial w_i} = \frac{\partial l}{\partial h_i}\frac{\partial h_i}{\partial w_i}, \tag{6}$$

$$\frac{\partial l}{\partial h_{i-1}} = \frac{\partial l}{\partial h_i}\frac{\partial h_i}{\partial h_{i-1}}. \tag{7}$$

That is, in the backward propagation, you are given the gradient $\frac{\partial l}{\partial h_i}$ w.r.t the output h_i and you need to compute the gradient $\frac{\partial l}{\partial w_i}$ w.r.t the parameter w_i in this layer (ReLU layers and pooling layer do not have parameters, so you can skip this step), and the gradient $\frac{\partial l}{\partial h_{i-1}}$ w.r.t the input (which will be passed to the lower layer).

Note that you will need to store the data in the latest call of the forward function in order to perform back propagation. For example, in the `conv_layer.py`, we store the data of the forward pass to `self.data` and use it for back propagation.

Convolution layer: `conv_layer.py` has been implemented for you.

1. **[10 points]** Pooling layer: You need to implement the `backward` function in the pooling layer (`pool_layer.py`). You can assume the padding is 0 here. As explained before, there are two type of poolings, max pooling and average pooling. We only test max pooling in this homework. Hence, in the code, the `act_type` can only take value `max`.
2. **[5 points]** ReLU layer: You need to implement the `backward` function in the ReLU layer (`dense_layer.py`). The function interfaces are clearly explained in the code.
3. **[10 points]** Dense layer: You need to implement the `backward` function in the Dense layer (`dense_layer.py`). While computing the gradient of parameters in the backward pass for all layers, compute the average gradient (i.e divide by batch_size).

Refer to here for more introduction on backward propagation.

3.4 Training and SGD

Having completed all the forward and backward functions, we can compose them to train a model. `main.py` and `le_net.py` are the main files for you to specify a network structure and train a model.

3.4.1 Network Structure

The function modules are written so that you can change the structure of the network without changing the code. At the bottom of `le_net.py`, we define the structure of LeNet. It is consisted of the 8 layers, the configuration of layer `i` is specified in `layers[i]`. Each layer has a parameter called `layers[i]["type"]`, which define the type of layer. The configuration of each layer is clearly explained in the comment.

3.5 SGD

After the network structure is defined and parameters are initialized, we can start to train the model. We use stochastic gradient descent (SGD) to train the model. At every iteration, we take a random mini batch of the training data and call `LeNet` to get the gradient of the parameters, and we then update the parameter based on the gradients (`param["w"].grad` and `param["b"].grad`).

Vanilla SGD The stochastic gradient updates the parameters as follows:

$$w = w - \alpha \frac{\partial l}{\partial w}, \tag{8}$$

Momentum To make your model converges faster, you can use SGD with momentum:

$$\theta = \mu\theta + \alpha \frac{\partial l}{\partial w}, \tag{9}$$

$$w = w - \theta, \tag{10}$$

where θ maintains the history accumulative gradient, the momentum μ determines how the gradients from previous steps contribute to current update and α is the learning rate at the current step.

Refer to here for a detailed explanation of momentum.

The learning rate α is a sensitive parameter in neural network models. We need to decrease the learning rate as we iterate over the batches. Here we choose the following schedule policy to decrease the learning rate:

$$\alpha_t = \frac{\epsilon}{(1+\gamma t)^p}, \tag{11}$$

where ϵ is the initial learning rate, t is the iteration number, and γ and p controls how the learning rate decreases.

We typically need to impose some regularization on the network parameters to avoid over-fitting, and one commonly used strategy is called weight decay. This is equivalent to L2 norm regularization. With weight decay, the total loss becomes:

$$l_{reg} = l + \frac{\lambda}{2} \sum_i w_i^2 \tag{12}$$

and the gradient w.r.t w_i becomes:

$$\frac{\partial l_{reg}}{\partial w_i} = \frac{\partial l}{\partial w_i} + \lambda w_i \tag{13}$$

After finishing all the above components, you can run `main.py`. You should be able to get an accuracy greater than 0.95 after the first epoch.

We can see the training cost is decreasing. After the training is finished, the test accuracy you should get is about 99.1%. Check here for the state of the art result on MNIST classification accuracy.

It takes about 11 minutes to train for 1 epoch on our laptops. The actual training time depends on the computer you use and your implementation.

1. **[20 pt]** End-to-end training: In the file `optimizer.py`, you need to implement the function `update_lr` in `SGDMomentum` to get the learning rate α_t at iteration `itr` (t). The correspondence between input to the function and the math symbol here is: `itr` is t, `epsilon` is ϵ, `gamma` is γ, `power` is p. In the same `optimizer.py` file, you also need to implement the `step` function in `SGDMomentum` to perform sgd with momentum.

   After finishing training, you can save your models' weights to `weights.npy` using the `save_model` function in `le_net.py`. We will evaluate if your model's accuracy on the test set is greater than 0.98.

4 Feature Visualization

After you finish training, you can take the model and visualize the internal features of the LeNet. Suppose we want to visualize the output of the first four layers of the first data point of the test set. You can use `matplotlib` to show an image.

The output of first layer is simply the image itself (because the first layer is data layer).

1. **[3 pt]** The output of the second layer is the output of the first convolution layer, the output feature map is of size $24 \times 24 \times 20$. They are actually 20 images of size 24×24. Show the 20 images on a single figure file (use `subplot` and organize them in 4×5 format).
2. **[3 pt]** The output of the third layer is the output of the max pooling operation on the previous output of convolution layer. The output feature map is of size $12 \times 12 \times 20$. They are actually 20 images of size 12×12. Show the 20 images on a single figure file (use `subplot` and organize them in 4×5 format).
3. **[4 pt]** Compare the output of the second layer and the original image (output of the first layer), what changes do you find? Compare the output of the third layer and the output of the second layer, what changes do you find? Explain your observation.

Put the answer of this part in your report.

5 Training Experiments

In the previous sections, we gave you `main.py` to run and set the values of all of the parameters for you. For this section, you should write your own scripts to train and test. These will not be auto-graded, so you may call them whatever you like. We recommend that you copy `main.py` since much of the code from it will be the same. Warning: change the name that you save the network to as you will need the original LeNet you trained in a later question.

5.1 Cross-Validation

[10 pt] In the previous sections, the values of all of the training parameters ϵ, the initial learning rate, p the power of the learning rate decay, γ, the parameter of the learning rate decrease, as well as μ, the momentum value were all set for you in `main.py`.

Instead, we want you to determine some of these values for yourself. Run k-fold cross-validation with $k = 5$ to find the optimal value of the parameter ϵ. Try 0.001, 0.005, 0.01 and 0.05. Set the number of iterations (not the number of epochs) to be 1000 so that you can train this in a reasonable amount of time.

Show the k-fold validation values for each of these parameter and then train on the full dataset for the best choice and report the final test number.

5.2 Pretraining

[10 pt] Here we will introduce you to the idea of "pre-training" by having you run a simple experiment.

First, train a model on the original MNIST dataset (you should already have one that you trained from section 3 that was automatically saved by `LeNet.py`). Next, we want to "fine-tune" this model on a different dataset. Load the previously trained model and then train that network again on a new dataset: rotated MNIST. We have included the script to loads this data for you in `data_loader.py`. Specifically, you should be able to take

- `mnist_all_rotation_normalized_float_test.amat`
- `mnist_all_rotation_normalized_float_train_valid.amat`

from the website. Run `save_rotated` to dump npy arrays and then use `load_rotated` to load matrices. You can make use of the `load_model` and `save_model` methods in the `LeNet` class. Also train another network "from scratch" that trains on this dataset from the random initialization rather than training on MNIST first.

Report the train and test accuracy you get from your pre-trained network and the network trained from scratch. Explain why you get the results you get.

6 Submission Instructions

Please put your `dense_layer.py` and `pool_layer.py` and the weight file `weights.npy` in a folder called `cnn` and run the following command:

```
$ tar cvf cnn.tar cnn
```

Then submit your tarfile. Note that the autograding will take several minutes to complete. Hence we recommend you to debug your code offline before submitting it.