HOMEWORK 7: DEEP LEARNING

CMU 10601: MACHINE LEARNING (SPRING 2017)
https://piazza.com/cmu/spring2017/10601
OUT: April 05, 2017
DUE: **QNA**: April 12, 2017 11:59
Autolab and Gradescope: April 17, 2017 11:59 pm
TAs: Daniel Bird, Sree Harsha, Abhinav Maurya, Ye Yuan

START HERE: Instructions

- **Collaboration Policy**: Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 3.4"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the collaboration policy on the website for more information: http://www.cs.cmu.edu/~mgormley/courses/10601-s17/about.html
  The following is worth a total of **[2 points]** to not lose these marks please submit these questions to gradescope:
  - Did you receive any help whatsoever from anyone in solving this assignment?
  - If you answered *yes*, give full details:
    (e.g. *Jane explained to me what is asked in Question 3.4*)
  - Did you give any help whatsoever to anyone in solving this assignment?
  - If you answered *yes*, give full details:
    (e.g. *I pointed Joe to section 2.3 to help him with Question 2*)
  - How many hours did this assignment take?
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601-s17/about.html
- **Submitting your work:** You will use QnA to submit answers to all theory questions, and Autolab to submit only your Python or Octave code. Please follow instructions at the end of this PDF to correctly submit all your code to Autolab.
  - **QnA:** We will use an online system called QnA for short answer and multiple choice questions. You can access the site at https://qna.cs.cmu.edu and log in with your Andrew ID and password using the button labeled ”LOGIN WITH CMU ID”. (As a reminder, never enter your Andrew password into any website unless you have first checked that the URL starts with ”https://” and the domain name ends in ”.cmu.edu” – but in this case it's OK since both conditions are met).
    IMPORTANT NOTE: The deadline for the QnA section of this homework is April 12, 2017 11:59pm
    NO LATE PENALTIES WILL BE GIVEN for those who do not complete it by this date, instead you will **RECEIVE A SCORE OF 0** on the QnA section of the homework .
  - **Gradescope:** For written problems such as derivations, proofs, or plots we will be using Gradescope. You can access the site here: https://gradescope.com/. Each derivation/proof should be completed on a separate page. Submissions can be handwritten, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Upon

submission, label each question using the template provided. Regrade requests can be made, however this gives the TA to regrade your entire paper, meaning if additional mistakes are found then points will be deducted.

- **Autolab:** You can access the 10601 course on autolab by going to `https://autolab.andrew.cmu.edu/` All programming assignments will be graded automatically on Autolab using Octave 3.8.2 and Python 2.7. You may develop your code in your favorite IDE, but please make sure that it runs as expected on Octave 3.8.2 or Python 2.7 before submitting. You should use the same language for answering all your questions. The code which you write will be executed remotely against a suite of tests, and the results are used to automatically assign you a grade. To make sure your code executes correctly on our servers, you should avoid using libraries which are not present in the basic Octave install. You may use Matlab instead of Octave if you have access to it. Please make sure you do not use any Matlab-specific libraries in your code that might make it fail against our tests. For Python users, you are encouraged to use the `numpy` package with version 1.7.1. The deadline displayed on Autolab may not correspond to the actual deadline for this homework, since we are allowing late submissions (as discussed in the late submission policy on the course site) Any attempt to "hack" Autolab or any other kind of code cheating will be dealt with according to university policy on student cheating.

- **Assignment-specific caveats:**

  - In the Python handout, we use objects of numpy "ndarray" type to store all numerical objects. This is slightly different from the numpy "matrix" type, though "ndarray" supports pretty much all vector and matrix operations that you may expect.
  - NumPy on autograder doesn't support automated broadcasting operations when adding two numpy arrays of different sizes where one can be expanded along one of the dimensions to the size of the other. Please refrain from relying on numpy broadcasting.
  - The handout provided to you has test scripts named `testLeNet.m` for Octave code and `testLeNet.py` for Python code. These scripts run without errors but make no backpropagation progress since the functions you are required to code return dummy objects of the expected size and shape but filled with all zeros. After coding each function, we recommend that you run the test script to ensure it runs through at least a few iterations. This will ensure that the code you have added does not cause the pipeline to throw errors. This does not ensure that the code you have written is correct, which can only be ensured after completing all required functions, running the test script, and seeing if the test accuracy actually increases to expected levels (>97%).
  - If you are using Octave on a 32-bit computer, it might give you an OOM error because 32-bit Octave has a 2GB limit on the amount of memory it can handle. If you face this issue, we recommend using Matlab which is freely available to all CMU students.
  - You have a total of 20 Autolab submissions. Use them wisely. In order to not waste Autolab submissions, we recommend debugging your implementation on your local machine using the provided test script and making sure your code is running correctly first before any Autolab submission. Assertions in the code handout test if the sizes of the objects you are returning are correct.
  - It takes several hours to finish training. The actual training time depends on the computer you use and your implementation.
  - A variable called `fullset` in the `testLeNet` script determines how much data is loaded. If set to false, it loads $1/20^{th}$ of the dataset. If true, it loads the entire dataset. In the code handout, it is set to false. Under the smaller dataset, it should take less than an hour for a complete implementation to get accuracies >97%. If your code runs more than an hour without making such good progress, it is either very inefficient or has a bug. Once you are sure you have a correct implementation by testing on the smaller dataset, set `fullset` to true in the `testLeNet` script. This should take longer, but will provide accuracies >99%.
  - The function `load_mnist` loads the data from `mnist_all.mat`. The loaded features are of size d x n where d is the input dimension (784 for 28 x 28 MNIST images) and n is the number of datapoints.
  - The CNN structure is initialized in `testLeNet` script in the `layers` data structure and the parameters are initialized in the `init_convnet` function which takes `layers` as input and outputs a data

structure called params with randomly initialized parameters. Please read the testLeNet script and init_convnet function to understand the structure of our CNN and shape of parameters before starting the coding part of this assignment.

Recommended Readings and Exercises: Before you begin this assignment, we recommend reading chapters on Deep Feedforward Networks[1] and Convolutional Networks[2] from the Deep Learning book [2].

Optional Readings and Exercises: We also recommend playing with an online demo of multi-layer perceptrons available here. Try tweaking various parameters like number of layers, number of neurons per layer, the activation function, learning rate, and regularization to see how the MLP performance changes with these parameters. You can also play with an online demo of CNNs available here. The Stanford CNN course[3] and UFLDL[4] material are excellent for beginners to read. You are encouraged to read some of them before doing this assignment.

In this assignment, You will get familiar with the workflow needed to build a neural network model in this assignment. We are going to implement a Convolution Neural Network (CNN) to classify hand written digits of MNIST [5] data. Since the breakthrough of CNNs on ImageNet classification [4], CNNs have been widely applied and achieved the state the art of results in many areas of computer vision. The recent AI programs that can beat humans in playing Atari game [7] and Go [8] also used CNNs in their models.

1 A Brief Introduction to LeNet

We are going to implement the earliest CNN model, LeNet [5], that was successfully applied to classify hand written digits.

1.1 LeNet Structure

The architecture of LeNet is shown in figure 1.

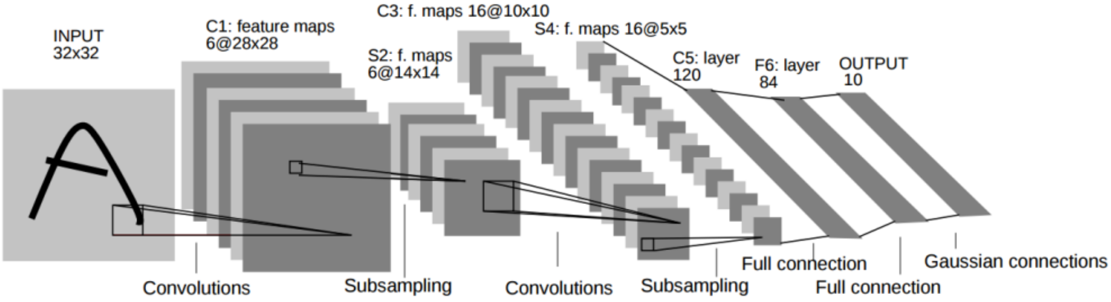

Figure 1: convolution layer

The schematic diagram shows a succession of layers:

- The input can be a black-and-white image with a single channel or an RGB image with 3 channels. Denoted as **INPUT** in the figure.
- LeNet applies multiple convolutional filters to the input image to obtain an output for each channel of the input image and each convolutional filter. All these outputs are stacked together and shown as **C1** in the figure. The values of the convolutional filters are part of the parameters of a CNN and are learned through gradient descent. Parameters like the shape of the filters, the padding applied to an image before performing convolution, and the stride at which convolution is performed are all decided along with the structure of the CNN before training.
- Max-pooling layer is applied to output C1 of the convolutional layer. Max-pooling is used to reduce the number of parameters in a CNN. It is based on the principle that nearby pixels in an image have similar intensities and therefore can be replaced by just the max intensity in a downward subsampling of the image. Max-pooling

[1] http://www.deeplearningbook.org/contents/mlp.html
[2] http://www.deeplearningbook.org/contents/convnets.html
[3] http://cs231n.github.io/
[4] http://ufldl.stanford.edu/tutorial/
[5] http://yann.lecun.com/exdb/mnist/

parameters like the padding applied to an image before performing max-pooling, and the stride at which max-pooling is performed are decided along with the structure of the CNN before training. Max-pooling layer does not have any parameters to be learned using SGD during training. Output of the max-pooling layer is denoted as **S2** in the figure.

- LeNet performs another layer of convolution after S2 yielding output **C3**. This layer can have its own parameters like padding and stride different from the first convolutional layer. It also has its own set of convolutional filters that are learned during training.
- The second convolution is followed by a second max-pooling performed on C3 yielding output **S4**.
- The outputs S4 of second max-pooling layer are densely connected to the next output layer **C5** using a linear transformation followed by a simple non-linear transformation which applies an elementwise non-linearity like the `sigmoid` or `ReLU` function to each output of the linear transformation. Mathematically, it means that the vectorized contents of S4 are multiplied with a weight matrix and the result is passed through an element-wise non-linearity such as the `sigmoid` or `ReLU` to obtain the output vector C5. The linear transformation stage is also separately called an Inner Product (IP) layer. The size of the weight matrix is decided along with the structure of the CNN, while the weights themselves are CNN parameters to be learnt during training.
- Similarly, C5 is connected to the next layer **F6** using a linear transformation followed by an elementwise non-linearity. Thus, it has its own weight matrix.
- The output layer F6 is transformed linearly by a final matrix multiplication to obtain logit (i.e. unnormalized probability) scores $s_j, j = 0..9$. Since there are 10 digits in MNIST digit classification task, we obtain 10 logit scores using the linear transformation. These are normalized into output probabilities using the following softmax transformation:

$$p_j = \frac{e^{s_j}}{\sum_{k=0}^{9} e^{s_k}} \tag{1}$$

  Each of the 10 probabilities is associated with one of the 10 digits and indicates the probability that the input image shows that digit. The output probabilities are denoted as **OUTPUT** in the diagram. Note that this layer consists of an inner product followed by a softmax operation. Sometimes, these two operations carried out in the OUTPUT layer are separately represented in code or diagrams as two successive layers - IP and SOFTMAX layers - for clarity. Do not worry about this architectural choice for the purpose of this assignment. The last layer is already implemented in the code handout provided to you.

Thus, the architecture of LeNet is as follows:

INPUT –>C1 –>S2 –>C3 –>S4 –>C5 –>F6 –>OUTPUT

Here each –>indicates a LeNet layer that carries out a transformation from previous output layer to next output layer.

The architecture could also alternatively be written as follows:

INPUT –>C1 –>S2 –>C3 –>S4 –>C5 –>F6 –>F7 –>SOFTMAXOUTPUT

Here, the transformation F6 –>OUTPUT has been expanded into F6 –>F7 –>SOFTMAXOUTPUT where F7 is the output of a fully connected layer and SOFTMAXOUTPUT is obtained by applying softmax to F7's unnormalized logit score outputs.

Each of the types of layers which is used to compose a CNN like LeNet is described in technical detail in the next section.

1.2 LeNet Training: Hello Backprop!

During training, we carry out backpropagation to learn the parameters of the convolutional layers and the fully connected layers. As we now know from extensive experience with algorithms such as linear regression, logistic regression, and support vector machines, formulating and minimizing a loss function is a great way to solve a machine learning task. In CNN, we use the familiar loss of negative conditional log-likelihood we used in the logistic regression assignment.

If the correct label of the i^{th} input image is digit j, we want to maximize the probability p_{ij} associated with that digit in the vector of OUTPUT probabilities. This is equivalent to maximizing $\log(p_{ij})$ since $\log$ is a monotonically increasing function. This is equivalent to minimizing $-\log(p_{ij})$ which is the negative conditional log-likelihood for the i^{th} input image. The loss function we want to minimize is the negative conditional log-likelihood summed across all datapoints:

$$loss = \sum_{i=1}^{N} -\log(p_{ij}) \tag{2}$$

where j is the output label for i^{th} datapoint.

During training, an additional layer called LOSS layer is added at the end of the above CNN which computes $-\log(p_{ij})$ given the i^{th} input image and its label j. The training procedure then focuses on learning parameters of convolutional and fully connected layers that minimize the output of the LOSS layer summed across all datapoints using backpropagation [2]. Gradient calculations for the purpose of performing SGD flow from the LOSS layer backward to the INPUT layer. Hence, the process of calculating all gradients is called *backpropagation*.

1.3 LeNet Predictions

During testing/prediction, we are given all parameters of the convolutional and fully connected layers. We provide an input image to LeNet and calculate the above layers of LeNet in succession until we obtain the output probabilities. We predict the digit shown in an input image by picking the digit corresponding to maximum of the 10 output probabilities. This process is often known as forward propagation since calculations flow from the INPUT to the OUTPUT. Please note that forward propagation of values needs to be performed during training as well before a backward propagation of gradients can be carried out. For more details, read the suggested chapters of [2].

The LOSS layer is not required during prediction and is therefore removed from the architecture after the training is finished. Thus, during prediction or test time, we directly obtain the OUTPUT probabilities associated with each of the 10 digits, since the 10-dimensional OUTPUT layer is the final layer after we remove the LOSS layer.

2 Structure of Convolutional Neural Networks (CNN)

We begin by introducing the basic structure and building blocks of CNNs. 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 the input being in different classes. We optimize an objective function over the parameters of all the layers 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 following types:

2.1 Inner Product Layer (fully connected layer)

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

$$f(x) = Wx + b. \tag{3}$$

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 d dimensional column vector, and W is a $n \times d$ matrix and b is n dimensional column vector.

2.2 Activation Layer

We add nonlinear activation functions after the inner product layers to model the non-linearity of real data. Here are some of the popular choices for non-linear activation:

- **Sigmoid**: $\sigma(x) = \frac{1}{(1+e^{-x})}$
- **tanh**: $\tanh(x) = \frac{(e^{2x}-1)}{(e^{2x}+1)}$
- **ReLU**: $\text{relu}(x) = \max(0, x)$

Rectified Linear Unit (ReLU) has been found to work well in vision related problems. There is no learnable parameters in the ReLU layer. In this homework, you will use only ReLU, though it is good to be aware of other activation functions like tanh, sigmoid, Leaky ReLU [6], Parametric ReLU [3], and Exponential Linear Unit (ELU) [1].

Note that the activation is usually combined with inner product layer as a single layer, but here we separate them in order to make the code modular.

2.3 Convolution Layer

The convolution layer is the core building block of CNNs. Unlike the inner product layer, each output neuron of a convolution layer is connected only to some input neurons. As the name suggest, in the convolution layer, we apply convolution operations with filters on input feature maps (or images). In image processing, there are many types of kernels (filters) that can be used to blur, sharpen an image or detect edges in an image. Read the Wikipedia page[6] 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 to the convolution layer is a three dimensional tensor (and is often referred to as the **input feature map** in the rest of this document), rather than a vector as in inner product layer, and it is of the shape $h \times w \times c$, where h is the height of each input image, w is the width and c is the number of channels. Note that we represent each channel of the image as a different slice in the input tensor.

Fig. 2 shows the detailed convolution operation. The input is a feature map, i.e., a three dimensional tensor with size $h \times w \times c$. The convolution operation involves applying filters on this input. Each filter is a sliding window, and the output of the convolution layer is the sequence of outputs produced by each of those filters during the sliding operation. Let us assume each filter has a square window of size $k \times k$ per channel, thus making filter size $k \times k \times c$. We use n filters in a convolution layer, making the number of parameters in this layer $k \times k \times c \times n$. In addition to these parameters, the convolution layer also has two hyper-parameters: the padding size p and stride step s. In the sliding window process described above, the output from each filter is a function of a neighborhood of input feature map. Since the edges have fewer neighbors, applying a filter directly is not feasible. To avoid this problem, inputs are typically padded (with zeros) on all sides, effectively making the the height and width of the padded input $h + 2p$ and $w + 2p$ respectively, where p is the size of padding. Stride (s) is the step size of convolution operation.

As Fig. 2 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 feature map and then sum the product to get the **output feature map**. Hence, the dimensions of the output feature map are $[(h+2p-k)/s+1] \times [(w+2p-k)/s+1] \times c$. Since we have n filters in a convolution layer, the output feature map is of size $[(h+2p-k)/s+1] \times [(w+2p-k)/s+1] \times c \times n$, since each filter outputs for each of the c image channels and n filters are stacked together. In the MNIST datset we will use, the images are black and white and therefore the number of channels c is 1.

For more details about the convolutional layer, see Stanford's course on CNNs for visual recognition [7].

2.4 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, and perform an aggregation operation on the output of a convolution layer. Like the convolution layer, the pooling operation also acts locally on the feature maps. A popular kind of pooling is max-pooling, and it simply involves computing the maximum value within each feature window. This allows us to extract

[6] https://en.wikipedia.org/wiki/Kernel_(image_processing)

[7] http://cs231n.github.io/convolutional-networks/

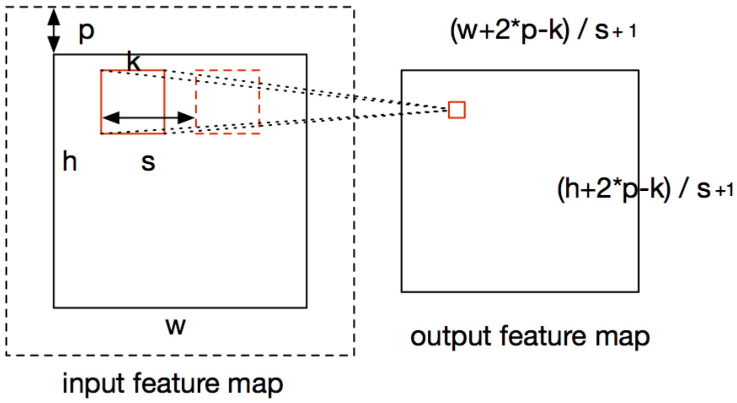

Figure 2: convolution layer

more salient feature maps and reduce the number of parameters of CNNs to reduce over-fitting. Pooling is typically applied independently within each channel of the input feature map.

2.5 SoftMax layer

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

$$p = \text{softmax}(Wx + b). \tag{4}$$

2.6 Loss layer

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

$$l = -\log(p_j), \tag{5}$$

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

3 Example Forward and Backpropagation QnA [12 points]

DUE: April 12th 2017 at 11:59pm, if you do not submit by this time you will receive a score of 0 for this question.

A toy CNN architecture is provided in table 1. We are going to perform forward and backward passes on this CNN to understand forward and backpropagation.

Layer Type	Configuration
Input	size: $4 \times 4 \times 1$
Convolution	$k = 2, s = 1, p = 0, n = 2$
Pooling	MAX, $k = 2, s = 1, p = 0$
IP	$n = 3$
ReLU	
SoftMax	
Loss	Negative Log-Likelihood

Table 1: Toy CNN Architecture
Note: The IP layer between ReLU and Softmax has been removed for convenience

The convolutional filters are initialized as follows:

```
Filter1 =
 1, 1
 2, 0

Filter2 =
 2, 1
 0, 2
```

The inner product layer is initialized as follows: Input to the IP layer is the output from the max-pooling layer vectorized using row-major format i.e. (i, j, k) entry in output of max-pooling layer is $(i-1)*COLS+j+(k-1)*ROWS*COLS$ entry in vector input of IP layer, where $ROWS$ and $COLS$ are the number of rows and columns in the max-pooling layer output respectively.

```
W1 =
    1, 0.5, 1, 2,    -1,    1,   1, -1
  0.5,    1, 2, 1, -0.5, 0.5, -1,   1
 0.25,   -1, 1, 1,  0.5,   -1, -1, -1

b =
0
0
0
```

Consider that you are given the following example image of digit 1 (out of 3 possible output digits 0, 1, 2).

```
0, 0, 4, 0
0, 2, 5, 0
0, 0, 1, 0
0, 0, 2, 0
```

Use the above input image to perform a single forward and backprogation through the toy CNN using a learning rate of 0.01 and SGD without momentum. You will need your calculations of forward and backpropagation to answer the following questions on QnA: `https://qna.cs.cmu.edu/#/pages/view/124`. **You must submit your final solutions by 04/12 11:59pm.**

Assume that all indices start with 1 when answering the following questions i.e. (1,1) is the top-left entry of any matrix and 1 is the first index of any vector. Answer the following questions:

- What is the height and width of the output of convolutional layer?
- What is the height and width of the output of pooling layer?
- What is the length of the output of ReLU layer?
- What is the length of the output of softmax?
- What is the (1,1,2) entry of convolutional layer output?
- What is the (2,1,1) entry of max-pooling layer output?
- What is the entry at index 2 of ReLU layer output?
- What is the entry at index 2 of softmax layer output?
- What is the cross entropy (Loss) of this example? (Hint: Use log to the base e)
- What is the value of the derivative of your loss function with respect to your softmax function, $\frac{dJ}{dy}$?
- What is the value of the derivative of your loss function with respect to parameter W1(1,1), $\frac{dJ}{dW1_{(1,1)}}$?
- What is the value of the derivative of your loss function with respect to the parameter (1,1) in filter 1, $\frac{dJ}{dF1_{(1,1)}}$?

4 LeNet

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

Layer Type	Configuration
Input	size: $28 \times 28 \times 1$
Convolution	$k = 5, s = 1, p = 0, n = 20$
Pooling	MAX, $k = 2, s = 2, p = 0$
Convolution	$k = 5, s = 1, p = 0, n = 50$
Pooling	MAX, $k = 2, s = 2, p = 0$
IP	$n = 500$
ReLU	
IP	$n = 10$
SoftMax	
Loss	Negative Log-Likelihood

Table 2: Architecture of LeNet

The architecture of LeNet is shown in Table. 2. 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, a ReLU non-linearity, another inner product layer, a softmax transformation and finally a loss layer. This is the typical structure of CNNs. We wish to point out two important caveats:

- You will write the forward propagation and backward propagation code for some of the layers. Do not assume the sizes given in Table. 2 and hardcode them anywhere in your code. Your code should work for general layer parameters which are provided as inputs to your function.
- The code handout you have received implements the last three layers IP –>SoftMax –>Loss in a single function `mlrloss`. This doesn't change any aspect of the code you are implementing. However, if you read the code handout, we do not want you to be confused about this aspect of the implementation.

5 Implementation

The basic framework of CNN is already finished and you need to help fill some of the empty functions.

5.1 Data structure

We use special data structures to store the input and output of each layer. Octave and Python implementations use Octave structures and Python dictionaries to store various fields of a data structure respectively. Hence, `structure.field` is used to refer to a particular field of the data structure in Octave and `dictionary['field']` is used to refer to a particular field of the data structure in Python. In this assignment, **we will use `structure.field` to refer to various fields in a structure. Please read it as `structure['field']` if you are producing your code in Python.** Specifically:

- `output.height` (in Octave) (or `output['height']` in Python) stores the height of feature maps
- `output.width` stores the width of feature maps
- `output.channel` stores the channel size of feature maps
- `output.batch_size` stores the batch_size of feature maps
- `output.data` stores actual data of feature map. It is an Octave matrix or a numpy array with size [`height`×`width`×`channel`, `batch_size`]. If necessary, you can reshape it to [`height`, `width`, `channel`, `batch_size`] during your computation, but remember to reshape it back to a two dimensional matrix at the end of each function.

- output.diff stores gradient w.r.t the data matrix output.data. This is used in backward propagation. It has the same shape as output.data.

For each layer, we use param to store the parameters:

- param.w stores the weight matrix of each layer.
- param.b stores the bias of each layer.

Size and structure of some output variables that could be confusing:

- param_grad has the same shape as param.
- input_od and input_od_approx have the same shape as input.data

5.2 Feed Forward

The forward computation in the convolution layer has been implemented for you.

Q1 **[20 points]** Pooling layer: You need to implement the pooling_layer_forward function. You can assume the padding is 0 here. The function interfaces are explained in the code.

Q2 **[3 points]** ReLU layer: You need to implement relu_forward function. The function interfaces are explained in the code.

Q3 **[10 points]** Inner product layer: You need to implement inner_product_forward function. The function interfaces are explained in the code.

5.3 Backward Propagation

Let us assume layer i computes a function f_i with parameters w_i, then the final loss is computed as:

$$l = f_I(w_I, f_{I-1}(w_{I-1}, ...)). \tag{6}$$

We want to optimize l over the parameters of each layer. We can use 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:

$$\frac{\partial l}{\partial w_i} = \frac{\partial l}{\partial h_i}\frac{\partial h_i}{\partial w_i}, \tag{7}$$

$$\frac{\partial l}{\partial h_{i-1}} = \frac{\partial l}{\partial h_i}\frac{\partial h_i}{\partial h_{i-1}}. \tag{8}$$

That is, in the backward propagation, you are given the gradient $\frac{\partial l}{\partial h_i}$ w.r.t the output h_i (stored in output.diff) and you need to compute gradient $\frac{\partial l}{\partial w_i}$ w.r.t the parameter w_i (param_grad.w and param_grad.b) 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 (input_od) (which will be passed to the lower layer).

The backpropagation for the convolution layer, conv_layer_backward has been implemented for you.

Q4 **[20 points]** Pooling layer: You need to implement the pooling_backward function for max pooling. The function interfaces are explained in the code.

Q5 **[3 points]** ReLU layer: You need to implement the relu_backward function. The function interfaces are explained in the code.

Q6 **[10 points]** Inner product layer: You need to implement the inner_product_layer_backward function. The function interfaces are explained in the code.

5.4 Training and SGD

Having completed all the forward and backward functions, we can compose them to train a model. testLeNet.m or testLeNet.py is the main file for you to specify a network structure and train a model.

5.4.1 Network Structure

The function modules are written so that you can change the structure of the network without changing the code. At the head of the test script file testLeNet.m or testLeNet.py, we define the structure of LeNet. It consists of the 8 layers, the configuration of layer i is specified in layers{i} in Octave and layers['i'] in Python. Each layer has a parameter called layers{i}.type in Octave or layers['i']['type'] in Python, which define the type of layer. The configuration of each layer is clearly explained in the comment.

After defining the layers, we use init_convnet function to initialize the parameters of each layer. The parameters of layer i is param{i} in Octave or param['i'] in Python, param{i}.w or param['i']['w'] is the weight matrix and param{i}.b or param['i']['b'] is the bias. init_convnet will figure out the shapes of all parameters and give them an initial value according to the layer configuration layers. We use uniform random variables within given ranges to initialize the parameters. You can refer to init_convnet function for further details.

5.5 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 conv_net function to get the gradient of the parameters, and we thenupdate the parameter based on the gradients (param_grad).

We use stochastic gradient with momentum to update the parameters:

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

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

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

See the UFLDL tutorial [8] 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 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. This part is already implemented for you in the get_lr function.

We impose L2 regularization (or weight decay) on our weights parameters (param.w or param['w']), so the loss becomes

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

and the gradient w.r.t w_i becomes:

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

Q7 **[7 points]** You need to implement sgd_momentum function to perform sgd with momentum. param_winc is provided to store the history accumulative gradient (θ here). Note that you need to update both w and b in each param structure.

[8] http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/

After finishing all the above components, you can run `testLeNet.m` if you are coding in Octave or `testLeNet.py` if you chose Python and get the output like this

```
cost = 0.490166 training_percent = 0.828125
cost = 0.120790 training_percent = 0.984375
cost = 0.065535 training_percent = 1.000000

test accuracy: 0.962400

cost = 0.186757 training_percent = 0.921875
cost = 0.052646 training_percent = 0.984375
cost = 0.018671 training_percent = 1.000000

test accuracy: 0.974500
```

Within each training epoch, you will see the training loss after processing a portion of the training data and at the end of the epoch, you will see the test accuracy. We can see the training cost is generally decreasing. Note that `testLenet` script loads only a subset of the MNIST data by default. In case you're curious, you can run the code with the whole dataset by flipping `fullset` parameter of `load_mnist` function to `true`. After the training is finished, the test accuracy you should get with the full set is about 99.1%. You can find out the state of the art results on this problem online [9].

5.6 Octave Implementation

Here is an overview of all the Octave files provided to you.

- `mnist_all.mat` contains all the data set needed in your experiment.
- `load_minst.m` loads all the data set and processes the data set into the format we need.
- `testLeNet.m` is the main file which does the training and test.
- `conv_net.m` defines the CNN. It takes the configuration of the network structure (defined in `layers`), the parameters of each layer (`param`), the input data (`data`) and label (`labels`) and does feed forward and backward propagation, returns the cost (`cp`) and gradient w.r.t all the parameters (`param_grad`).
- `conv_layer_forward.m` does the convolutional layer feed forward.
- `conv_layer_backward.m` does the convolutional layer backward propagation.
- `get_lr.m` returns the learning rate of each iteration.
- `mrloss.m` implements the forward and backward propagation for the loss layer. It calculates the negative log likelihood cost in forward operation and calculates the gradient w.r.t. input data and parameters in backward propagation.

Also provided are two helper functions:

- `im2col_conv.m` returns a list of pixels for 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 `im` and places all pixels within the feature window in a separate column in the `col` data structure which is to be returned by the function. 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.

[9] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

- col2im_conv.m returns a list of the gradients at each pixel of the input image, given a list of gradients for each pixel for 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 whch 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.

You need to finish the functions listed below:

- relu_forward.m does the relu feed forward.
- relu_backward.m does the relu backward propagation.
- inner_product_forward.m does the inner product layer feed forward.
- inner_product_backward.m does the inner product layer backward propagation.
- pooling_layer_forward.m does the pooling layer feed forward.
- pooling_layer_backward.m does the pooling layer backward propagation.
- sgd_momentum.m updates the parameters of the model given the gradients.

6 Python Implementation

Here is an overview of all the Python files provided to you.

- mnist_all.mat contains all the data set needed in your experiment.
- cnn_lenet.py contains all the Python starter code and interfaces of the functions you need to complete.
- testLeNet.py contains the training and testing script which initializes and runs the CNN on MNIST data.

Here are the completed functions provided to you in cnn_lenet.py.

- load_minst loads all the data set and processes the data set into the format we need.
- conv_net defines the CNN. It takes the configuration of the network structure (defined in layers), the parameters of each layer (param), the input data (data) and label (labels) and does feed forward and backward propagation, returns the cost (cp) and gradient w.r.t all the parameters (param_grad).
- conv_layer_forward does the convolutional layer feed forward.
- conv_layer_backward does the convolutional layer backward propagation.
- get_lr returns the learning rate of each iteration.
- mrloss implements the forward and backward propagation for the loss layer. It calculates the negative log likelihood cost in forward operation and calculates the gradient w.r.t. input data and parameters in backward propagation.

Also provided are two helper functions:

- im2col_conv returns a list of pixels for 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 im and places all pixels within the feature window in a separate column in the col data structure which is to be returned by the function. 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 for 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 whch 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.

You need to finish the following functions whose interfaces are located in cnn_lenet.py:

- relu_forward does the relu feed forward.
- relu_backward does the relu backward propagation.
- inner_product_forward does the inner product layer feed forward.
- inner_product_backward does the inner product layer backward propagation.
- pooling_layer_forward does the pooling layer feed forward.
- pooling_layer_backward does the pooling layer backward propagation.
- sgd_momentum updates the parameters of the model given the gradients.

7 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 i^{th} data point (i.e., the first image of test set). Refer to vis_data.m on how to do show an image.

The output of first layer is simply the image itself (because the first layer is data layer). The output of the second layer (convolution) is 20 images, each of size 24×24, hence $24 \times 24 \times 20$, and similarly, the output of the third layer (max pooling) is of size $12 \times 12 \times 20$, and so on.

Q8 **[5 points]** Visualize the output of the second and third layers. Show 20 images from each layer on a single figure file.

Q9 (a) **[2 points]** Compare the output of the second layer and the original image (output of the first layer), what changes do you find?

(b) **[2 points]** Compare the output of the third layer and the output of the second layer, what changes do you find?

(c) **[3 points]** Explain your observation.

Q10 **[3 points]** Run your final code on the entire MNIST dataset by setting fullset to true in testLeNet. Report the final test accuracy and time taken by testLeNet to run the training procedure for 10,000 iterations.

Submit your plots and explanation on Gradescope.

8 Note on debugging gradients

One way to check whether your implementation of analytical gradients is correct is to compare the gradient values against the approximation given by finite difference. Recall that the derivative of a function is defined as follows.

$$f^{'}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{14}$$

You can use this expression to approximate $f^{'}(x)$ by computing the quantity of the fraction above with a small value of h.

In order to help you with debugging your backpropagation implementation, you are provided an implementation of finite difference to check gradients in the function `finite_difference`. The inputs are `output`, `input` and `h`, where `input` and `output` are layer structures, and `h` is the small value shown in the expression above. The functions returns an approximate gradient `input_od_approx` of the same shape as the input data stored in the structure `input`. This can be compared to the `input_od` value returned by one of the backward propagation functions. For very small values of `h`, the difference between `input_od` and `input_od_approx` should also be very small.

9 Submission Instructions

Below are the files you need to submit if you have chosen an Octave implementation:

- `relu_forward.m`
- `relu_backward.m`
- `inner_product_forward.m`
- `inner_product_backward.m`
- `pooling_layer_forward.m`
- `pooling_layer_backwawrd.m`
- `sgd_momentum.m`

Below are the files you need to submit if you have chosen a Python implementation:

- `cnn_lenet.py`

Please put your files in a folder called `hw7` and run the following command:

```
$ tar cvf hw7.tar hw7
```

Submit the tar file generated. Ensure that all the required files are present even if unimplemented and that you are able to see a total score on Autolab.

References

[1] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[6] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning*, volume 30, 2013.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.