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. You *must* turn in at least $n - 1$ of the $n$ homeworks to pass the class, even if for zero credit.

- **Collaboration policy:** Homeworks must be done individually, except where otherwise noted in the assignments. “Individually” means each student must hand in their own answers, and each student must write and use their own code in the programming parts of the assignment. It is acceptable for students to collaborate in figuring out answers and to help each other solve the problems, though you must in the end write up your own solutions individually, and you must list the names of students you discussed this with. 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.

- **Submission:** You must submit your solutions on time BOTH electronically by submitting to autolab AND by dropping off a hardcopy in the bin outside Gates 8221 by 5:30 p.m. Thursday, April 28, 2016. On the Homework 7 autolab page, you can click on the “download handout” link to download the submission template, which is a tar archive containing a Octave .m file for each programming question. Replace each of these files with your solutions for the corresponding problem, create a new tar archive of the top-level directory, and submit your archived solutions online by clicking the “Submit File” button.

  **DO NOT** change the name of any of the files or folders in the submission template. In other words, your submitted files should have exactly the same names as those in the submission template. Do not modify the directory structure.
In this assignment, we are going to implement a Convolution Neural Network (CNN) to classify handwritten digits of MNIST data. Since the breakthrough of CNNs on ImageNet classification [1], CNNs have been widely applied and achieved the state of results in many areas of computer vision. The recent AI programs that can beat humans in playing Atari game [3] and Go [4] also used CNNs in their models.

We are going to implement the earliest CNN model, LeNet [2], that was successfully applied to classify handwritten digits. You will get familiar with the workflow needed to build a neural network model after this assignment.

The Stanford CNN course and UFLDL material are excellent for beginners to read. You are strongly encouraged to read some of them before doing this assignment.

1 Convolutional Neural Networks (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 can have learnable parameters including weights and biases. Each layer takes the output from previous layers, 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 an 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 following types:

1.1 Inner product layer (fully connected layer)

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

\[ f(x) = W x + b. \] (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 a \( n \times d \) matrix and \( b \) is an \( n \) 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):

\[ f(x) = \max(0, x). \] (2)

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

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 dimension of a convolution layer is only connected to some input dimensions. As the name suggest, in the convolution layer, we apply 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 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 dimension 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 three dimensional tensor with size \(h \times w \times c\). Assume the (square) convolution 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 \(s\). Zero padding is typically used; after padding, the first two dimensions of input feature maps are \((h + 2p) \times (w + 2p)\). Stride \(s\) controls the step size of 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 output feature map is \([((h + 2p - k)/s + 1] \times [(w + 2p - k)/s + 1]\). 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 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 windows size \(k\) and stride \(s\). Pooling operation is typically applied independently within each channel of the input feature map. There are two types of pooling operation: 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 is \([((h + 2p - k)/s + 1] \times [(w + 2p - k)/s + 1]\). This is the same as in 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](image)

Refer to here for more detailed explanation of the pooling layer.

### 1.5 Loss layer

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

\[
p = \text{softmax}(Wx + b).
\]  

(3)
In training, we know the label given the input image, hence, we want to minimize the negative log probability of the given label:

\[ l = - \log(p_j), \]  

(4)

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

Note here the loss layer can be seen as the composition of a fully connected layer with a final loss function. Here we combine them for convenience.

2 LeNet

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

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>data size: ( 28 \times 28 \times 1 )</td>
</tr>
<tr>
<td>CONV</td>
<td>( k = 5, s = 1, p = 0, n = 20 )</td>
</tr>
<tr>
<td>POOLING</td>
<td>MAX, ( k = 2, s = 2, p = 0 )</td>
</tr>
<tr>
<td>CONV</td>
<td>( k = 5, s = 1, p = 0, n = 50 )</td>
</tr>
<tr>
<td>POOLING</td>
<td>MAX, ( k = 2, s = 2, p = 0 )</td>
</tr>
<tr>
<td>IP (Inner Product)</td>
<td>( n = 500 )</td>
</tr>
<tr>
<td>RELU</td>
<td></td>
</tr>
<tr>
<td>LOSS</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Architecture of LeNet

The architecture of LeNet is shown in Table 1. The name of the layer type explains itself (IP is short for Inner Product). LeNet is composed of interleaving of convolution layers and pooling layers, followed by an inner product layer, a ReLU 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.

- `mnist_all.mat` contains all the data set needed in your experiment.
- `load_mnist_all.m` loads all the data set and processes the data set into the format you need.
- `vis_data.m` helps you visualize the data. Run `vis_data.m` in Matlab to see the images.
- `testLeNet.m` is the main file which does the training and test.
- `conv_net.m` defines the CNNs. It takes the configuration of the network structure (defined in `layers`), the parameters of each layer (`params`), 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.
- `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 and parameters in backward propagation.

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.
get_lr.m returns the learning rate of each iteration.
sgd_momentum.m updates the parameters of the model given the gradients.

3.1 Data structure
We use a special data structure to store the input and output of each layer. The output of one layer will be the input of a following layer, so we use input and output interchangeably. Specifically:
- output.height 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. Note here output.data is a matrix with size [height\times width\times 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 output.data. This is used in backward propagation. It has the same shape as 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.

3.2 Feed Forward
- Convolution layer: conv_layer_forward.m has been implemented for you.
- [20 points] Pooling layer: You need to implement the pooling_layer_forward.m function. You can assume the padding is 0 here. As has explained before, there are two type of poolings, max pooling and average pooling. In the code, layer.act.type can take value MAX and AVE, which denotes max pooling and average pooling, respectively. You need to deal with them separately. Each type of pooling operation is worth 10 points.
- [5 points] ReLU layer: You need to implement relu_forward.m function.
- [10 points] Inner product layer: You need to implement inner_product_layer_forward.m function.
3.3 Backward Propagation

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

$$ l = f_l(w_I, f_{I-1}(w_{I-1}, ...)). $$

(5)

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:

$$ \frac{\partial l}{\partial w_i} = \frac{\partial l}{\partial h_i} \frac{\partial h_i}{\partial w_i}; $$

(6)

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

(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 gradient $\frac{\partial l}{\partial w_i}$ w.r.t the parameter $w_i$ in this layer (ReLU layers and pooling layers 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).

- Convolution layer: `conv_layer_backward.m` has been implemented for you.
- [20 points] Pooling layer: You need to implement the `pooling_layer_backward.m` function. You can assume the padding is 0 here. As has explained before, there are two type of poolings, max pooling and average pooling. In the code, `layer.act.type` can take value `MAX` and `AVE`. You need to deal with them separately. Each type of pooling operation worth **10 points**.
- [5 points] ReLU layer: You need to implement the `relu_backward.m` function.
- [10 points] Inner product layer: You need to implement the `inner_product_layer_backward.m` function.

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. `testLeNet.m` is the main file 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 head of `testLeNet.m`, we define the structure of LeNet. It is consisted of 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.

After defining the layers, we use `init_convnet.m` to initialize the parameters of each layer. The parameters of layer $i$ is `params{i}`, `params{i}.w` is the weight matrix and `params{i}.b` is the bias. `init_convnet.m` 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.m` for further details.

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 `convnet.m` to get the gradient of the parameters, and we then update the parameter based on the gradients (`param_grad`).
We use stochastic gradient with momentum to update the parameters:

\[ \theta = \theta + \alpha \frac{\partial l}{\partial \theta}, \]  
\[ w = w - \theta, \]  

where \( \theta \) maintain the weighted average of past gradients, the momentum \( \mu \) determines how the gradients from previous steps contribute to current update and \( \alpha \) is the learning rate at current step.

Refer here for a detailed explanation of momentum.

The learning rate \( \alpha \) 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:

\[ \alpha_t = \frac{\epsilon}{(1 + \gamma t)^p}, \]  

where \( \epsilon \) is the initial learning rate, \( t \) is the iteration number, and \( \gamma \) and \( p \) controls how the learning rate decreases.

- [5 pt] You need to implement `get_lr.m` to get the learning rate \( \alpha_t \) at iteration \( t \). The correspondence between input to the function and the math symbol here is: \texttt{iter} is \( t \), \texttt{epsilon} is \( \epsilon \), \texttt{gamma} is \( \gamma \), \texttt{power} is \( p \).

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:

\[ l_{reg} = l + \frac{\lambda}{2} \sum_i w_i^2 \]  

and the gradient w.r.t \( w_i \) becomes:

\[ \frac{\partial l_{reg}}{\partial w_i} = \frac{\partial l}{\partial w_i} + \lambda w_i \]  

Here we impose weight decay ONLY on the weight parameters, i.e, \texttt{param.w}, NOT on the biases, \texttt{param.b}.

- [10 pt] You need to implement `sgd_momentum.m` to perform sgd with momentum. \texttt{param.winc} is provided to store the weighted average of past gradients (\( \theta \) here).

After finishing all the above components, you can run `testLeNet.m` and get the output like this:

iteration 10 training cost = 2.295502 accuracy = 0.125000  
iteration 20 training cost = 2.270276 accuracy = 0.265625  
iteration 30 training cost = 2.193458 accuracy = 0.578125  
iteration 40 training cost = 2.051970 accuracy = 0.531250  
iteration 50 training cost = 1.474182 accuracy = 0.671875  
iteration 60 training cost = 0.876938 accuracy = 0.734375

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 4 hours to finish the training in Matlab on a computer with Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz. The actual training time depends on the computer you use and your implementation. Try to make your code efficient!

**Hint**: You can try submitting your implementation to autolab to verify it is correct before you start training and continue to the feature visualization part.
4 Feature Visualization

After you finish training, you can take the model (the model is saved as \texttt{lenet.mat}, check \texttt{testLeNet.m}) and visualize the internal features of the LeNet. As has been explained, convolution can perform all sorts of operation on the images such as blurring, sharpening etc. In this part, we are going to examine what kind operation those convolution filters do.

For simplicity, we consider the first convolution layer. The output of 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 $24 \times 24$ and each image is the output of one filter applied on the input image. Hence, we can visualize the output images to see what changes happen to the image after convolution.

We take 10 images from the test set ($x_{\text{test}(; :, 1:10)}$) and feed them through the model. Denote the output of second layer as $\text{output}$ and it is of size $24 \times 24 \times 20 \times 10$. For each filter, we can plot the original images and its corresponding output images to study its effect. The output of $i$-th filter is $\text{output}_i = \text{output}(; :, i, :)$. $\text{output}_i$ are 10 output images corresponding to the 10 input images $x_{\text{test}(; 1:10)}$. We can plot them side by side, as shown Fig. 2.

![Visualization of convolution output](image)

Figure 2: Visualization of convolution output

Column 1 and column 3 are the input images and column 2 and column 4 are the output images. Comparing the input and output, it is not hard to find that the filter detects horizon lines of the input images.

- [15 pt] Pick one “interesting” filter and plot the output in the same way as in Fig. 2 and explain what operation does the filter do. (Some of the filters have weights all zero and output is purely black, those filter are not interesting).

  
  
  \textbf{Hint:} Refer to \texttt{vis_data.m} on how to show an image and use the function \texttt{subplot} to organize the images in the format as shown in Fig. 2.

Put the answer of this part in the file \texttt{written_solutions.pdf}.

5 Submission Instructions

Below are the files you need to submit:

- \texttt{relu_forward.m}
• relu_backward.m
• inner_product_forward.m
• inner_product_backward.m
• pooling_layer_forward.m
• pooling_layer_backward.m
• get_lr.m
• sgd_momentum.m
• written_solutions.pdf

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

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

Then submit your tarfile.

References


