10-401 Machine Learning: Homework 4

Due 5:00 p.m. Monday, April 9, 2018

Instructions

- **Late homework policy**: Homework is worth full credit if submitted before the due date. Up to 50 % credit can be received if the submission is less than 48 hours late. The lowest homework grade at the end of the semester will be dropped. Please talk to the instructor in the case of extreme extenuating circumstances.

- **Collaboration policy**: You are welcome to collaborate on any of the questions with anybody you like. However, you must write up your own final solution, and you must list the names of anybody you collaborated with on this assignment.

- **Submission:** You must submit your solutions on time electronically by submitting to autolab by 5:00 p.m. Monday, April 9, 2018. On the Homework 4 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 Convolutional 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 the art of results in many areas of computer vision. The recent AI programs that can beat humans in playing Atari [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 Neural Networks Theory

Recall that in HW1, you proved that a linear separator cannot be used to create the logical XOR. In this question, we will explore the power of neural networks.

1. [**5 points**] Given boolean inputs $x_1, x_2 \in \{0, 1\}$, show that a neural network with one hidden layer (consisting of two nodes) can output the function x_1 XOR x_2. Your neural network should use the Rectified Linear Unit (ReLU) in the hidden layer. E.g., each neuron computes the function $\max(0, \sum_i w_i x_i + b)$ for inputs x_1, x_2, where you choose the parameters w_1, w_2, and b (and these parameters can be different for each neuron). See Figure 1. For the output neuron, you should use a threshold function. E.g., output 1 if $\sum_i w_i a_i + b > 0$, otherwise output 0, where $a_1, \ldots, a_m$ denote the values from the nodes in the hidden layer (again, you choose $w_1, \ldots, w_m$ and b).

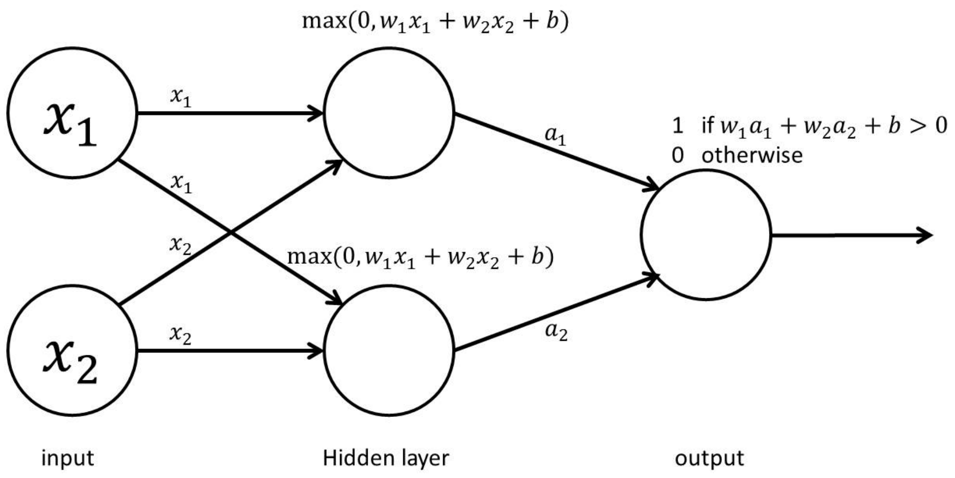
$\max(0, w_1x_1 + w_2x_2 + b)$
x_1
x_1
x_1
a_1
1 if $w_1a_1 + w_2a_2 + b > 0$
0 otherwise
x_2
$\max(0, w_1x_1 + w_2x_2 + b)$
x_2
x_2
a_2
input
Hidden layer
output

Figure 1: Neural Network for question 1

2. [**5 points**] Now prove that a neural network with one hidden layer can learn *any* binary function. E.g., given boolean inputs $x_1, \ldots, x_n \in \{0, 1\}$ and an arbitrary binary function $f : \{0, 1\}^n \to \{0, 1\}$, show that a network with one hidden layer can output f. Your hidden layer should have more than two nodes, and they should all be threshold units with inputs $x_1, \ldots, x_n$. The output node should also be a threshold function.

2 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 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:

2.1 Inner product 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.

2.2 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.

2.3 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. 2 shows the detailed convolution operation. The input is a feature map, i.e., a three dimensional tensor with size $h \times w \times c$. 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. 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 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.

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. 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.

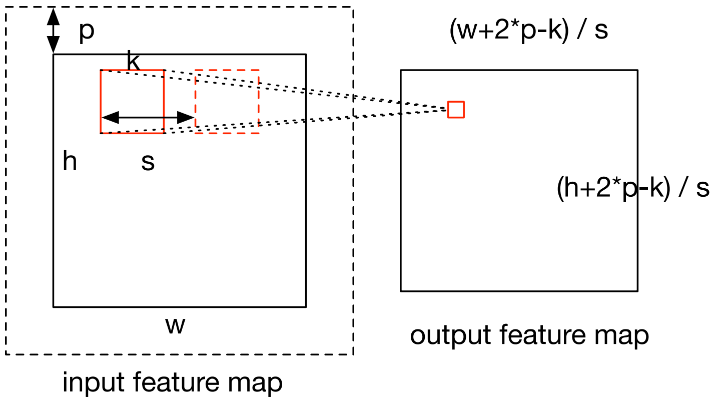

Figure 2: convolution layer

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

2.5 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.

3 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, $k = 2, s = 2, p = 0$
IP	$n = 500$
RE LU	
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.

4 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 the data set needed in your experiment.
- `load_minst_all.m` loads the data set and processes the data set into the format we 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 data 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 forward.
- `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.

4.1 Data structure

We use a special data structure to store the input and output of each layer. 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 the actual data of the feature map. Note here `output.data` is a matrix 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 `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` store the bias of each layer.

4.2 Feed Forward

- Convolution layer: `conv_layer_forward.m` has been implemented for you.
- [**15 points**] Pooling layer: You need to implement the `pooling_layer_forward.m` function. You can assume the padding is 0 here. As 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 the `relu_forward.m` function. The function interfaces are clearly explained in the code.
- [**10 points**] Inner product layer: You need to implement the `inner_product_layer_forward.m` function.

4.3 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).

- Convolution layer: `conv_layer_backward.m` has been implemented for you.
- [**15 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 velue `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. The input and output are clearly explained in the code.

Refer to here for more introduction on backward propagation.

4.4 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.

4.4.1 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 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.

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.

4.5 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.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:

$$\theta = \mu\theta + \alpha\frac{\partial l}{\partial w}, \tag{8}$$

$$w = w - \theta, \tag{9}$$

where θ maintain 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 current step.

Refer 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:

$$\alpha_t = \frac{\epsilon}{(1+\gamma t)^p}, \tag{10}$$

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

- **[5 pt]** You need to implement `get_lr.m` to get the learning rate `lr_t` (α_t) at iteration `iter` (t). The correspondence between input to the function and the math symbol here is: `iter` is t, `epsilon` is ϵ, `gamma` is γ, `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:

$$l_{reg} = l + \frac{\lambda}{2}\sum_i w_i^2 \tag{11}$$

and the gradient w.r.t w_i becomes:

$$\frac{\partial l_{reg}}{\partial w_i} = \frac{\partial l}{\partial w_i} + \lambda w_i \tag{12}$$

Here we impose weight decay ONLY on the weight parameters, i.e, `param.w`, NOT the biases, `param.b`.

- **[5 pt]** You need to implement `sgd_momentum.m` to perform sgd with momentum. `param_winc` is provided to store the history accumulative gradient (θ 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 on a computer with `Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz`. The actually training time depends on the computer you use and your implementation.

5 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 data point `xtest(:, 1)` (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).

- **[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).
- **[3 pt]** The output of the third layer is the output of appling 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).
- **[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 the file `written_solutions.pdf`.

6 Training Experiments

In the previous sections, we gave you `testLeNet.m` 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 `testLeNet.m` 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.

6.1 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 `testLeNet.m`.

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 max_iter 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.

6.2 Pretraining

[10 pt] (Bonus) 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 4 that was automatically saved by `testLeNet.m`). Next, we want to "fine-tune" this model on a different dataset. Write a script that loads the previously trained model and then trains that network again on a new dataset: rotated MNIST. We have included a script `load_mnist_rotated.m` that loads this data for you. **Do not call `init_convnet`** or else the weights will be erased and the experiment won't work. 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.

7 Submission Instructions

Below are the files you need to submit:

- `relu_forward.m`
- `relu_backward.m`
- `inner_product_forward.m`
- `inner_product_backward.m`
- `pooling_layer_forward.m`
- `pooling_layer_backwawrd.m`
- `get_lr.m`
- `sgd_momentum.m`
- `written_solutions.pdf`

Please put these files in a folder called `hw4` and run the following command:

```
$ tar cvf hw4.tar hw4
```

Then submit your tarfile.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.