Homework 6: Deep Learning

10-601B: Machine Learning (Fall 2016)
Out November 2, 2016
Due 5:30 p.m. Monday, November 21, 2016

TAs: Pradeep Dasigi, Varshaa Naganathan, Sriram Vasudevan

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.
- **Submission:** You must submit your code for questions 2-9, 12, 13 and 15 on time electronically by submitting to autolab by 5:30 p.m. Monday, November 21, 2016. On the Homework 5 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.

  For questions 1, 10, 11 and 14, submit your written answers on Gradescope.
- **Collaboration policy**: Please read the policy on the course webpage: http://www.cs.cmu.edu/~mgormley/courses/10601b-f16/about.html

In this assignment, we are going to implement a Convolution Neural Network (CNN) to classify hand written digits of MNIST [1] data. Since the breakthrough of CNNs on ImageNet classification [2], 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 [4] and Go [5] also used CNNs in their models.

We are going to implement the earliest CNN model, LeNet [3], that was successfully applied to classify hand written digits. You will get familiar with the workflow needed to build a neural network model after this assignment.

The Stanford CNN course[2] and UFLDL[3] material are excellent for beginners to read. You are encouraged to read some of them before doing this assignment.

1 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:

[1] http://yann.lecun.com/exdb/mnist/
[2] http://cs231n.github.io/
[3] http://ufldl.stanford.edu/tutorial/

1.1 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.:

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

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

1.2 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 ReLU, and a recently proposed modification of it called Exponential Linear Unit (ELU).

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.

1.3 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[4] 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. 1 shows the detailed convolution operation. The input is a feature map, i.e., a three dimensional tensor with size $h \times w \times c$. 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. 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 feature map and then sum the product to get the **output feature map**. Hence, the first two dimensions of output feature map is

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

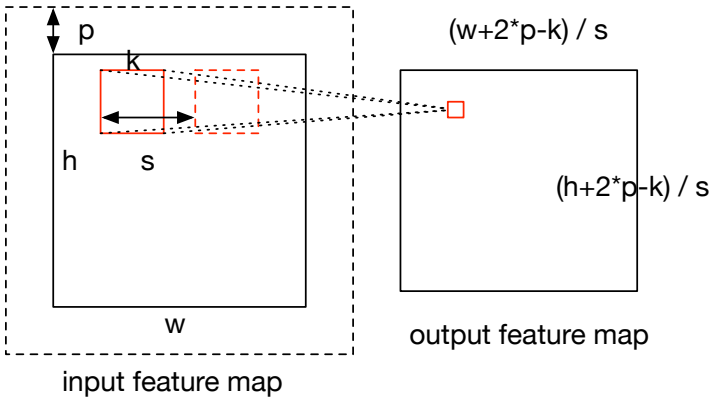

Figure 1: convolution layer

$[(h+2p-k)/s+1] \times [(w+2p-k)/s+1]$. Since we have n filters in a convolution layer, the output feature map is of size $[(h+2p-k)/s+1] \times [(w+2p-k)/s+1] \times n$.

For more details about the convolutional layer, see Stanford's course on CNNs for visual recognition [5].

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, 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 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.

Q1 **[5 points]** Does pooling make the model more or less sensitive to small changes in the input images? Why? By small changes, we mean moving the input images to the left or right, rotating them slightly etc.

1.5 Loss layer

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

$$p = \text{softmax}(Wx + b). \tag{2}$$

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

$$l = -\log(p_j), \tag{3}$$

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

2 LeNet

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

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

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	
Loss	

Table 1: Architecture of LeNet

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_minst_all.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). 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.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. 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.
- `elu_forward.m` does the elu feed forward.
- `elu_backward.m` does the elu 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.
- `elu_finite_difference.m` computes the gradient for the elu layer using the finite difference method.

3.1 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 actual data of 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 `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`

3.2 Feed Forward

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

Q2 **[20 points]** Pooling layer: You need to implement the `pooling_layer_forward.m` function. You can assume the padding is 0 here. **Hint:** You might be able to use `im2col_conv.m` and/or `col2im_conv.m` here to make implementation easier.

Q3 **[3 points]** ReLU layer: You need to implement `relu_forward.m` function. The function interfaces are explained in the code.

Q4 **[10 points]** Inner product layer: You need to implement `inner_product_forward.m` function.

3.3 Backward Propagation

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

$$l = f_I(w_I, f_{I-1}(w_{I-1}, ...)). \tag{4}$$

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:

$$\frac{\partial l}{\partial w_i} = \frac{\partial l}{\partial h_i}\frac{\partial h_i}{\partial w_i}, \tag{5}$$

$$\frac{\partial l}{\partial h_{i-1}} = \frac{\partial l}{\partial h_i}\frac{\partial h_i}{\partial h_{i-1}}. \tag{6}$$

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.m` has been implemented for you.

Q5 [**20 points**] Pooling layer: You need to implement the `pooling_backward.m` function for max pooling. **Hint:** You might be able to use `im2col_conv.m` and/or `col2im_conv.m` here to make implementation easier.

Q6 [**3 points**] ReLU layer: You need to implement the `relu_backward.m` function.

Q7 [**10 points**] Inner product layer: You need to implement the `inner_product_layer_backward.m` function.

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 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 `param{i}`, `param{i}.w` is the weight matrix and `param{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 `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:

$$\theta = \mu\theta + \alpha\frac{\partial l}{\partial w}, \tag{7}$$

$$w = w - \theta, \tag{8}$$

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 [6] 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:

$$\alpha_t = \frac{\epsilon}{(1+\gamma t)^p}, \tag{9}$$

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 `get_lr.m`.

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

$$l_{reg} = l + \frac{\lambda}{2}\sum_i w_i^2 \tag{10}$$

and the gradient w.r.t w_i becomes:

$$\frac{\partial l_{reg}}{\partial w_i} = \frac{\partial l}{\partial w_i} + \lambda w_i \tag{11}$$

Q8 **[7 points]** You need to implement `sgd_momentum.m` 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.

After finishing all the above components, you can run `testLeNet.m` 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.m` 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` 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 [7].

It takes several hours to finish training. The actual training time depends on the computer you use and your implementation.

4 Feature Visualization

After you finish training, you can take the model and visualize the internal features of the LeNet. Suppose we want to visualize the output of the first four layers of the 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). 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.

[6] http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/
[7] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

Q9 **[5 points]** Visualize the output of the second and third layers. Show 20 images from each layer on a single figure file (use `subplot` and organize them in 4×5 format).

Q10 (a) **[1 points]** Compare the output of the second layer and the original image (output of the first layer), what changes do you find?

(b) **[1 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.

Submit your plots and explanation on Gradescope.

5 Improving LeNet

We now have the complete implementation of LeNet. We will now try to improve it by using a trick that has been recently proposed. Exponential Linear Units (ELU) have been shown [1] to work better than ReLU when the network is sufficiently deep. This is the ELU function

$$\text{elu}(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases} \tag{12}$$

5.1 Implement ELU

Q11 **[5 points]** Implement `elu_forward.m`. Set the value of $\alpha = 1.0$.

Q12 **[5 points]** Implement `elu_backward.m`.

Q13 **[2 points]** Rerun `testLeNet.m` with ELU activation and report the test accuracy. Is this better or worse than the accuracy with ReLU?

5.2 Extra Credit: 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.

$$f^{'}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{13}$$

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

Q14 **[5 points]** Implement finite difference to check gradients implemented for ELU. You will have to complete the file `elu_finite_difference.m`. 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. **Hint:** You'll have to make a call to `elu_forward.m` for this function. Also note that while the gradient being approximated is that of ELU, the returned value must be the gradient of the loss function (i.e., don't forget to perform back propagation).

6 Submission Instructions

Below are the files you need to submit:

- `relu_forward.m`
- `relu_backward.m`
- `elu_forward.m`

- `elu_backward.m`
- `inner_product_forward.m`
- `inner_product_backward.m`
- `pooling_layer_forward.m`
- `pooling_layer_backwawrd.m`
- `sgd_momentum.m`
- Optionally `elu_finite_difference.m`

Please put these files in a folder called `hw6` and run the following command:

```
$ tar cvf hw6.tar hw6
```

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] 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.

[3] 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.

[4] 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.

[5] 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.