Project 5
Digit recognition with convolutional neural networks

Due date: 23:59 Sunday 3/29th (2020)

1. Instructions

Most instructions are the same as before. Here we only describe different points.

1. Generate a zip or tgz package, and upload to coursys. The package must contain the following in the following layout:
    - {SFUID}/
        - {SFUID}.pdf (your write-up, the main document for us to look and grade)
        - ec
            - ec.m
        - matlab/
            - col2im_conv.m
            - col2im_conv_matlab.m
            - conv_layer_backward.m
            - conv_layer_forward.m
            - conv_net.m
            - convnet_forward.m
            - get_lenet.m
            - get_lr.m
            - im2col_conv.m
            - im2col_conv_matlab.m
            - init_convnet.m
            - inner_product_backward.m
            - inner_product_forward.m
            - load_mnist.m
            - mlrloss.m
            - pooling_layer_backward.m
            - pooling_layer_forward.m
            - relu_forward.m
            - relu_backward.m
            - sgd_momentum.m
            - test_components.m

- test_network.m
- train_lenet.m
- vis_data.m
- lenet_pretrained.mat
- mnist all.mat

2. Project 5 has 13 pts.

2. Overview

In this assignment you will implement a convolutional neural network (CNN). You will be building a numeric character recognition system trained on the MNIST dataset.

We begin with a brief description of the architecture and the functions. For more details, you can refer to online resources such as http://cs231n.stanford.edu. Note that the amount of coding in this assignment is a lot less than the previous assignments. We will not provide detailed instructions, and one is expected to search online and/or reverse-engineer template code.

A typical convolutional neural network has four different types of layers.

Fully Connected Layer / Inner Product Layer (IP)

The fully connected or the inner product layer is the simplest layer which makes up neural networks. Each neuron of the layer is connected to all the neurons of the previous layer (See Fig 1). Mathematically it is modelled by a matrix multiplication and the addition of a bias term. For a given input x the output of the fully connected layer is given by the following equation,

$$f(x) = W x + b$$

W, b are the weights and biases of the layer. W is a two dimensional matrix of $m \times n$ size where n is the dimensionality of the previous layer and m is the number of neurons in this layer. b is a vector with size $m \times 1$.

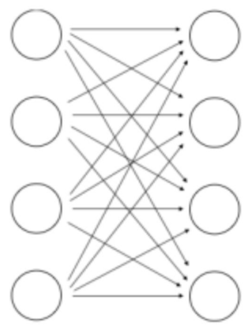


Figure 1: Fully connected layer

Convolutional Layer

This is the fundamental building block of CNNs. Before we delve into what a convolution layer is, let's do a quick recap of convolution.

As we saw in our lectures, convolution is performed using a k × k filter/kernel and a W × H image. The output of the convolution operation is a feature map. This feature map can bear different meanings according to the filters being used - for example, using a Gaussian filter will lead to a blurred version of the image. Using the Sobel filters in the x and y direction give us the corresponding edge maps as outputs.

Terminology : Each number in a filter will be referred to as a filter weight. For example, the 3x3 gaussian filter has the following 9 filter weights.

$$W = \begin{pmatrix} 0.0113 & 0.0838 & 0.0113 \\ 0.0838 & 0.6193 & 0.0838 \\ 0.0113 & 0.0838 & 0.0113 \end{pmatrix}$$

When we perform convolution, we decide the exact type of filter we want to use and accordingly decide the filter weights. CNNs try to learn these filter weights and biases from the data. We attempt to learn a set of filters for each convolutional layer.

In general there are two main motivations for using convolution layers instead of fully-connected (FC) layers (as used in neural networks).

1. **A reduction in parameters**
   In FC layers, every neuron in a layer is connected to every neuron in the previous layer. This leads to a large number of parameters to be estimated - which leads to over-fitting. CNNs change that by sharing weights (the same filter is translated over the entire image).

2. **It exploits spatial structure**
   Images have an inherent 2D spatial structure, which is lost when we unroll the image into a vector and feed it to a plain neural network. Convolution by its very nature is a 2D operation which operates on pixels which are spatially close.

Implementation details: The general convolution operation can be represented by the following equation:

$$f(X, W, b) = X * W + b$$

where W is a filter of size $k \times k \times C_i$, X is an input volume of size $N_i \times N_i \times C_i$ and b is 1×1 element. The meanings of the individual terms are shown below.

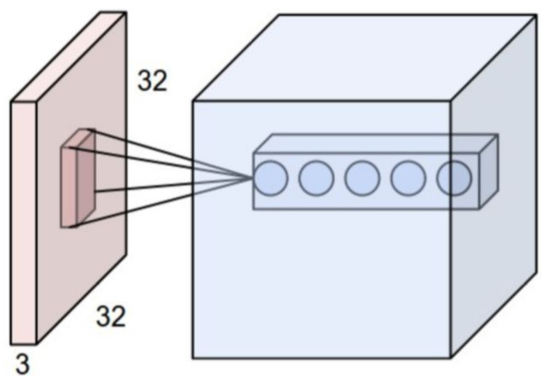


Figure 2: Input and output of a convolutional layer (Image source: Stanford CS231n

In the following example the subscript i refers to the input to the layer and the subscript o refers to the output of the layer.

- N_i - width of the input image
- N_i - height of the input image (image has a square shape)
- C_i - number of channels in the input image
- k_i - width of the filter
- s_i - stride of the convolution
- p_i - number of padding pixels for the input image
- num - number of convolution filters to be learnt

In assignment 1, we performed convolution on a grayscale image - this had 1 channel. This is basically the depth of the image volume. For an image with C_i channels - we will learn num filters of size $k_i \times k_i \times C_i$. The output of convolving with each filter is a feature map with height and width N_o, where

$$N_o = \frac{N_i - k_i + 2p_i}{s_i} + 1$$

If we stack the num feature maps, we can treat the output of the convolution as another 3D volume/ image with C_o = num channels.

In summary, the input to the convolutional layer is a volume with dimensions $N_i \times N_i \times C_i$ and the output is a volume of size $N_o \times N_o \times$ num. Figure 2 shows a graphical picture.

Pooling layer

A pooling layer is generally used after a convolutional layer to reduce the size of the feature maps. The pooling layer operates on each feature map separately and replaces a local region of the feature map with some aggregating statistic like max or average. In addition to reducing the size of the feature maps, it also makes the network invariant to small translations. This means that the output of the layer doesn't change when the object moves a little.

In this assignment we will use only a MAX pooling layer shown in figure 3. This operation is performed in the same fashion as a convolution, but instead of applying a filter, we find the max value in each kernel. Let k represent the kernel size, s represent the stride and p represent the padding. Then the output of a pooling function f applied to a padded feature map X is given by:

$$f(X, i, j) = \max_{x \in [i-k/2, i+k/2], y \in [j-k/2, j+k/2]} (X[x, y])$$

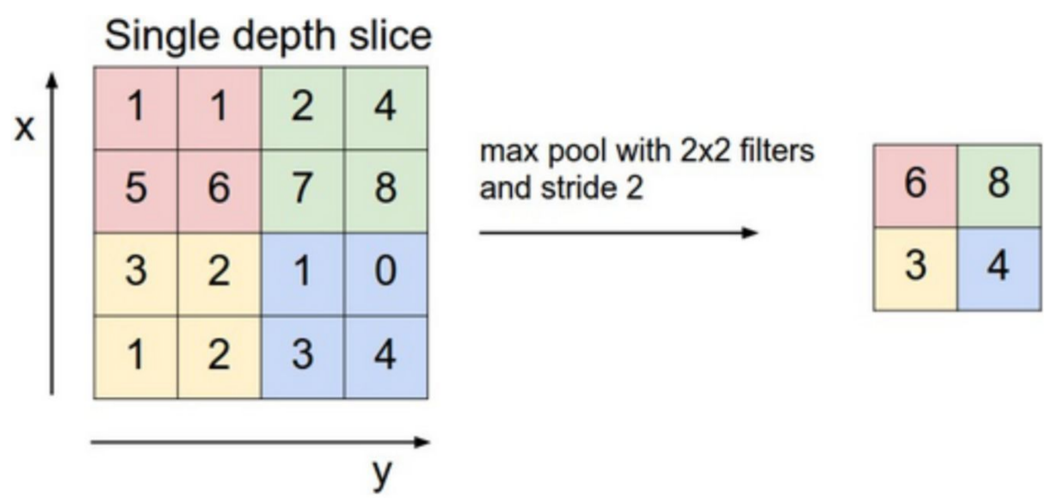


Figure 3: Example MAX pooling layer

Activation layer - ReLU - Rectified Linear Unit

Activation layers introduce the non-linearity in the network and give the power to learn complex functions. The most commonly used non-linear function is the ReLU function defined as follows,

$$f(x) = \max(x, 0)$$

The ReLU function operates on each output of the previous layer.

Loss layer

The loss layer has a fully connected layer with the same number of neurons as the number of classes. And then to convert the output to a probability score, a softmax function is used. This operation is given by,

$$p = \mathrm{softmax}(W x + b)$$

where, W is of size $C \times n$ where n is the dimensionality of the previous layer and C is the number of classes in the problem.

This layer also computes a loss function which is to be minimized in the training process. The most common loss functions used in practice are cross entropy and negative log-likelihood. In this assignment, we will just minimize the negative log probability of the given label.

Architecture

In this assignment we will use a simple architecture based on a very popular network called the LeNet (http://ieeexplore.ieee.org/abstract/document/726791/)

- Input - $1\times28\times28$
- Convolution - $k = 5$, $s = 1$, $p = 0$, 20 filters
- ReLU
- MAXPooling - $k=2$, $s=2$, $p=0$
- Convolution - $k = 5$, $s = 1$, $p = 0$, 50 filters
- ReLU
- MAXPooling - $k=2$, $s=2$, $p=0$
- Fully Connected layer - 500 neurons

- ReLU
- Loss layer

Note that all types of deep networks use non-linear activation functions for their hidden layers. If we use a linear activation function, then the hidden layers has no effect on the final results, which would become the linear (affine) functions of the input values, which can be represented by a simple 2 layer neural network without hidden layers.

There are a lot of standard Convolutional Neural Network architectures used in the literature, for instance, AlexNet, VGG-16, or GoogLeNet. They are different in the number of parameters and their configurations.

3. Programming

Most of the basic framework to implement a CNN has been provided. You will need to fill in a few functions. Before going ahead into the implementations, you will need to understand the data structures used in the code.

Data structures

We define four main data structures to help us implement the Convolutional Neural Network which are explained in the following section. Each layer is defined by a data structure, where the field type determines the type of the layer. This field can take the values of DATA, CONV, POOLING, IP, RELU, LOSS which correspond to data, convolution, max-pooling layers, inner-product/ fully connected, ReLU and Loss layers respectively. The fields in each of the layer will depend on the type of layer.

The input is passed to each layer in a structure with the following fields.

- height - height of the feature maps
- width - width of the feature maps
- channel - number of channels / feature maps
- batch size - batch size of the network. In this implementation, you will implement the mini-batch stochastic gradient descent to train the network. The idea behind this is very simple, instead of computing gradients and updating the parameters after each image, we doing after looking at a batch of images. This parameter batch size determines how many images it looks at once before updating the parameters.
- data - stores the actual data being passed between the layers. This is always supposed to be of the size [height × width × channel, batch size]. You can

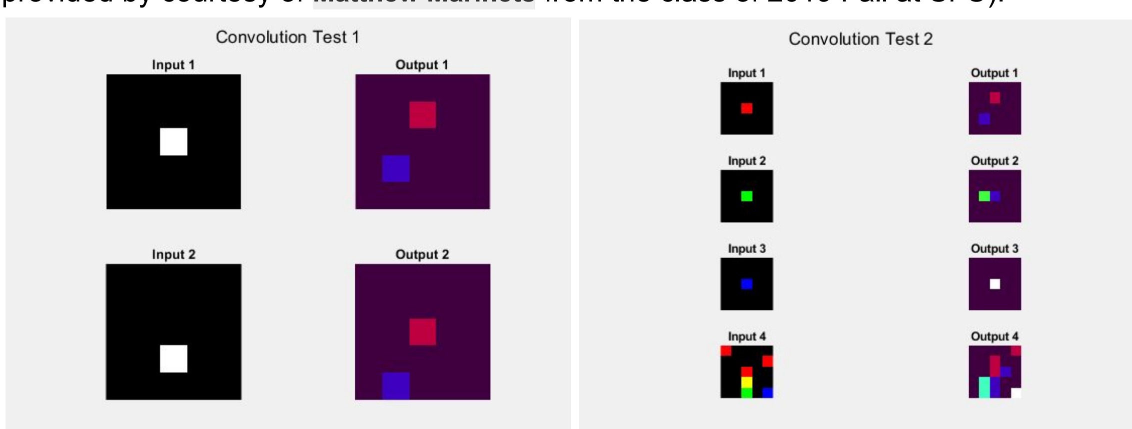
resize this structure during computations, but make sure to revert it to a two-dimensional matrix. The data is stored in a column major order. The row comes next, and the channel comes the last.
- diff - Stores the gradients with respect to the data, it has the same size as data. Each layer's parameters are stored in a structure param. You do not touch this in the forward pass.
- w - weight matrix of the layer
- b - bias

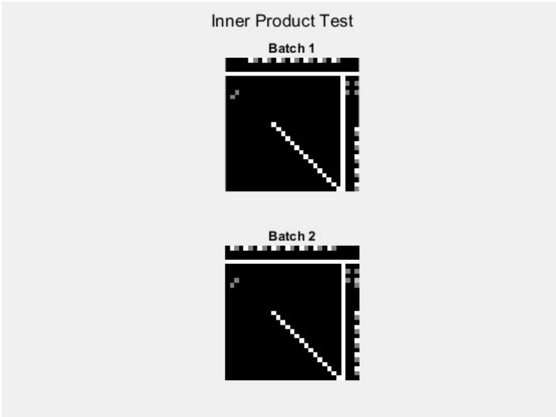
param_grad is used to store the gradients coupled at each layer with the following properties:
- w - stores the gradient of the loss with respect to w.
- b - stores the gradient of the loss with respect to the bias term.

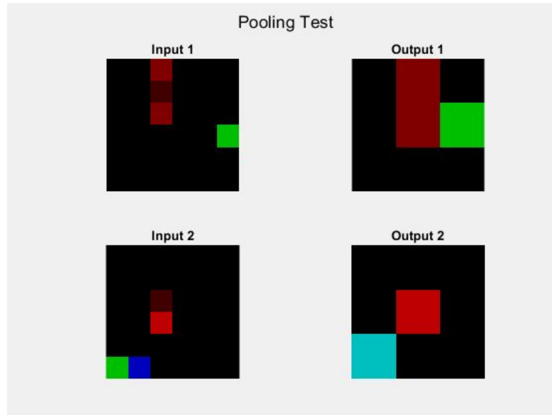
Part 1: Forward Pass

Now we will start implementing the forward pass of the network. Each layer has a very similar prototype. Each layer's forward function takes input, layer, param as argument. The input stores the input data and information about its shape and size. The layer stores the specifications of the layer (e.g., for a conv layer, it will have k, s, p). The params is an optional argument passed to layers which have weights. This contains the weights and biases to be used to compute the output. In every forward pass function, you are expected to use the arguments and compute the output. You are supposed to fill in the height, width, channel, batch size, data fields of the output before returning from the function. Also make sure that the data field has been reshaped to a 2D matrix.

In the past, we asked to provide some visualization of results for every single step. However, there is no meaningful visualization until we implement the forward functions of all the layers. Once you implement all the layers, run test_components.m, then copy/paste the visualization results into the report. Those images should look like the following. (test_components.m was provided by courtesy of **Matthew Marinets** from the class of 2019 Fall at SFU).







Q 1.1 Inner Product Layer - 1 Pts

The inner product layer of the fully connected layer should be implemented with the following definition

[output] = inner_product_forward(input, layer, param)

Q 1.2 Pooling Layer - 1 Pts

Write a function which implements the pooling layer with the following definition.

[output] = pooling_layer_forward(input, layer)

input and output are the structures which have data and the layer structure has the parameters specific to the layer. This layer has the following fields,

- pad - padding to be done to the input layer
- stride - stride of the layer
- k - size of the kernel (Assume square kernel)

Q 1.3 Convolution Layer - 1 Pts

Implement a convolution layer with the following definition.

[output] = conv_layer_forward(input, layer, param)

The layer for a convolutional layer has the same fields as that of a pooling layer and param has the weights corresponding to the layer. Do not worry about a field “group”, which is set to 1 in this assignment.

Q 1.4 ReLU - 1 Pts

Implement the ReLU function with the following definition.

[output] = relu_forward(input, layer)

Part 2 Back propagation

After implementing the forward propagation, we will implement the back propagation using the chain rule. Let us assume layer i computes a function f_i with parameters of w_i then final loss can be written as the following.

$$l = f_i(w_i, f_{i-1}(w_{i-1}, \dots))$$

To update the parameters we need to compute the gradient of the loss w.r.t. to each of the parameters.

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

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

where, $h_i = f_i(w_i, h_{i-1})$.

Each layer's back propagation function takes input, output, layer, param as input and return param_grad and input_od. output.diff stores the $\frac{\partial l}{\partial h_i}$. You are to use this to compute $\frac{\partial l}{\partial w}$ and store it in param_grad.w and $\frac{\partial l}{\partial b}$ to be stored in param_grad.b. You are also expected to return $\frac{\partial l}{\partial h_{i-1}}$ in input_od, which is the gradient of the loss w.r.t the input layer.

Q 2.1 ReLU - 1 Pts

Implement the backward pass for the Relu layer in relu_backward.m file. This layer doesn't have any parameters, so you don't have to return the param_grad structure.

Q 2.2 Inner Product layer - 1 Pts

Implement the backward pass for the Inner product layer in inner_product_backward.m

Putting the network together

This part has been done for you and is available in the function convnet forward. This function takes the parameters, layers and input data and generates the outputs at each layer of the network. It also returns the probabilities of the image belonging to each class. You are encouraged to look into the code of this function to understand how the data is being passed to perform the forward pass.

Part 3 Training

The function conv_net puts both the forward and backward passes together and trains the network. This function has also been implemented.

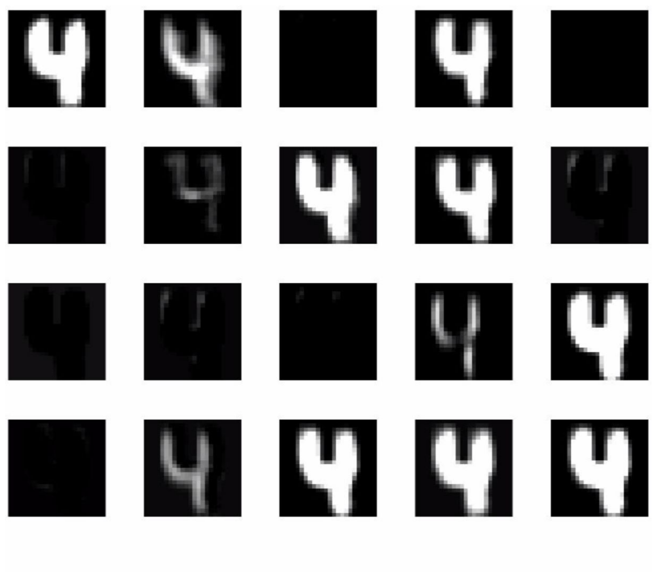


Figure 4: Feature maps of the second layer

Q 3.1 Training - 1 pts

The script train_lenet.m defines the optimization parameters and performs the actual updates on the network. This script loads a pretrained network and trains the network for 2000 iterations. Report the test accuracy obtained in your write-up after training for 3000 more iterations. Save the refined network weights as lenet.mat in the same format as lenet_pretrained.mat. The accuracy should be above 95%.

Q 3.2 Test the network - 1 Pts

The script test_network.m has been provided which runs the test data through the network and obtains the prediction probabilities. Modify this script to generate the confusion matrix and comment on the top two confused pairs of classes.

Q 3.3 Real-world testing - 1 Pts

Obtain real-world digit examples. Show the results of your system on at least 5 examples, which you obtained yourself (e.g., downloading from Internet, scribble yourself, taking an image yourself).

Part 4 Visualization

Q 4.1 - 1 Pts

Write a script vis_data.m which can load a sample image from the data, visualize the output of the second and third layers (i.e., CONV layer and ReLU layer). Show 20 images from each layer on a single figure file (use subplot and organize them in 4 × 5 format - like in Fig 4). To clarify, you take one image, run through your network, and visualize 20 features of that image at CONV layer and ReLU layer.

Q 4.2 - 1 Pts

Compare the feature maps to the original image and explain the differences.

Part 5 Image Classification - 2 Pts

We will now try to use the fully trained network to perform the task of Optical Character Recognition. You are provided a set of real world images in the images folder. Write a script ec.m which will read these images and recognize the handwritten numbers.

The network you trained requires a binary image with a single digit in each image. There are many ways to obtain this given a real image. Here is an outline of a possible approach:

1. Classify each pixel as foreground or background pixel by performing simple operations like thresholding.
2. Find connected components and place a bounding box around each character. You can use a matlab built-in function to do this.
3. Take each bounding box, pad it if necessary and resize it to 28×28 and pass it through the network.

There might be errors in the recognition, report the output of your network in the report.

Appendix: List of all files in the project

- col2im_conv.m Helper function, you can use this if needed
- col2im_conv_matlab.m Helper function, you can use this if needed
- conv_layer_backward.m - Do not modify
- conv_layer forward.m - To implement
- conv_net.m - Do not modify
- convnet_forward.m - To implement
- get_lenet.m - Do not modify. Has the architecture.
- get_lr.m - Gets the learning rate at each iterations
- im2col_conv.m Helper function, you can use this if needed
- im2col_conv_matlab.m Helper function, you can use this if needed
- init_convnet.m Initialise the network weights
- inner_product_backward.m - To implement
- inner_product_forward.m - To implement
- load_mnist.m - Loads the training data.
- mlrloss.m - Implements the loss layer
- pooling_layer_backward.m Implemented, do not modify
- pooling_layer_forward.m - To implement
- relu_backward.m - To implement
- relu_forward.m - To implement
- sgd_momentum.m - Do not modify. Has the update equations
- test_network.m - Test script
- train_lenet.m - Train script
- vis_data.m - Add code to visualise the filters
- lenet_pretrained.mat - Trained weights
- mnist_all.mat - Dataset

Notes

Here are some points which you should keep in mind while implementing:

- All the equations above describe the functioning of the layers on a single data point. Your implementation would have to work on a small set of inputs called a ”batch” at once.
- Always ensure that the output.data of each layer has been reshaped to a 2-D matrix.