CSCI 5561: Assignment #4

Convolutional Neural Network

1 Submission

- Assignment due: Apr 19 (11:55pm)
- Individual assignment
- Up to 2 page summary write-up with resulting visualization (more than 2 page assignment will be automatically returned.).
- Submission through Canvas.
- Skeletal codes can be downloaded from:
`https://www-users.cs.umn.edu/~hspark/csci5561/HW4_code.zip`. It contains the following four codes:
  - `main_slp_linear.m`
  - `main_slp.m`
  - `main_mlp.m`
  - `main_cnn.m`
- List of submission codes:
  - `GetMiniBatch.m`
  - `FC.m`
  - `FC_backward.m`
  - `Loss_euclidean.m`
  - `TrainSLP_linear.m`
  - `Loss_cross_entropy_softmax.m`
  - `TrainSLP`
  - `ReLu.m`
  - `ReLu_backward.m`
  - `TrainMLP.m`
  - `Conv.m`
  - `Conv_backward.m`
  - `Pool2x2.m`
  - `Pool2x2_backward.m`
  - `Flattening.m`
  - `Flattening_backward.m`
  - `TrainCNN.m`

CSCI 5561: Assignment #4

Convolutional Neural Network

- A list of MAT files that contain the following trained weights:
  - `slp_linear.mat`: `w`, `b`
  - `slp.mat`: `w`, `b`
  - `mlp.mat`: `w1`, `b1`, `w2`, `b2`
  - `cnn.mat`: `w_conv`, `b_conv`, `w_fc`, `b_fc`
- DO NOT SUBMIT THE PROVIDED IMAGE DATA
- The function that does not comply with its specification will not be graded.
- You are allowed to use MATLAB built-in functions except for the ones in the Computer Vision Toolbox and Deep Learning Toolbox. Please consult with TA if you are not sure about the list of allowed functions.

CSCI 5561: Assignment #4

Convolutional Neural Network

2 Overview

Figure 1: You will implement (1) a multi-layer perceptron (neural network) and (2) convolutiona neural network to recognize hand-written digit using the MNIST dataset.

The goal of this assignment is to implement neural network to recognize hand-written digits in the MNIST data.

MNIST Data You will use the MNIST hand written digit dataset to perform the first task (neural network). We reduce the image size ($28 \times 28 \rightarrow 14 \times 14$) and subsample the data. You can download the training and testing data from here:
`http://www.cs.umn.edu/~hspark/csci5561/ReducedMNIST.zip`

Description: The zip file includes two MAT files (`mnist_train.mat` and `mnist_test.mat`). Each file includes `im_*` and `label_*` variables:

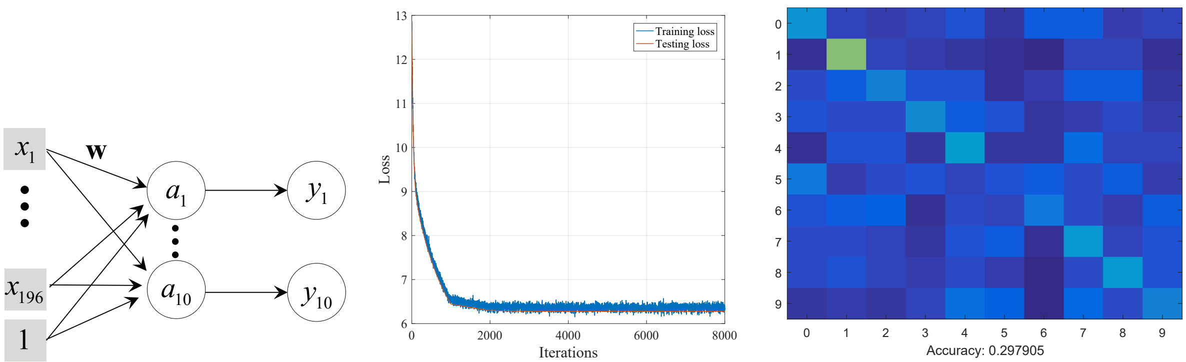
- `im_*` is a matrix ($196 \times n$) storing vectorized image data ($196 = 14 \times 14$)
- `label_*` is $n \times 1$ vector storing the label for each image data.

n is the number of images. You can visualize the i^{th} image, e.g.,
`imshow(uint8(reshape(im_train(:,i), [14,14])))`.

CSCI 5561: Assignment #4

Convolutional Neural Network

3 Single-layer Linear Perceptron



(a) Single linear perceptron (b) Loss (c) Confusion

Figure 2: You will implement a single linear perceptron that produces accuracy near 30%. Random chance is 10% on testing data.

You will implement a single-layer *linear* perceptron (Figure 2(a)) with stochastic gradient descent method. We provide `main_slp_linear` where you will implement `GetMiniBatch` and `TrainSLP_linear`.

`function [mini_batch_x, mini_batch_y] = GetMiniBatch(im_train, label_train, batch_size)`
Input: `im_train` and `label_train` are a set of images and labels, and `batch_size` is the size of the mini-batch for stochastic gradient descent.
Output: `mini_batch_x` and `mini_batch_y` are cells that contain a set of batches (images and labels, respectively). Each batch of images is a matrix with size 194×`batch_size`, and each batch of labels is a matrix with size 10×`batch_size` (one-hot encoding). Note that the number of images in the last batch may be smaller than `batch_size`.
Description: You may randomly permute the the order of images when building the batch, and whole sets of `mini_batch_*` must span all training data.

`function y = FC(x, w, b)`
Input: `x`$\in \mathbb{R}^m$ is the input to the fully connected layer, and `w`$\in \mathbb{R}^{n\times m}$ and `b`$\in \mathbb{R}^n$ are the weights and bias.
Output: `y`$\in \mathbb{R}^n$ is the output of the linear transform (fully connected layer).
Description: `FC` is a linear transform of $\mathbf{x}$, i.e., $\mathbf{y} = \mathbf{w}\mathbf{x} + \mathbf{b}$.

`function [dLdx dLdw dLdb] = FC_backward(dLdy, x, w, b, y)`
Input: `dLdy` $\in \mathbb{R}^{1\times n}$ is the loss derivative with respect to the output $\mathbf{y}$.

Output: `dLdx` $\in \mathbb{R}^{1\times m}$ is the loss derivative with respect the input $\mathbf{x}$, `dLdw` $\in \mathbb{R}^{1\times (n\times m)}$ is the loss derivative with respect to the weights, and `dLdb` $\in \mathbb{R}^{1\times n}$ is the loss derivative with respec to the bias.
Description: The partial derivatives w.r.t. input, weights, and bias will be computed. `dLdx` will be back-propagated, and `dLdw` and `dLdb` will be used to update the weights and bias.

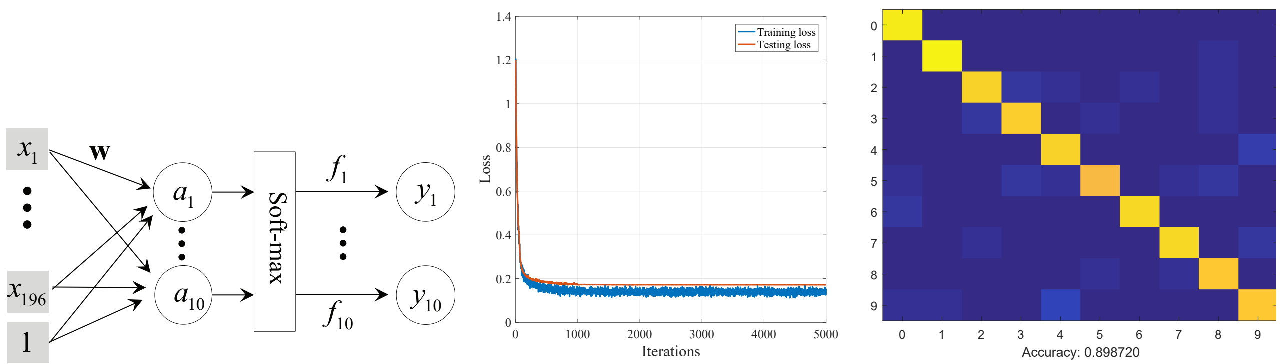
`function [L, dLdy] = Loss_euclidean(y_tilde, y)`
Input: `y_tilde` $\in \mathbb{R}^m$ is the prediction, and `y`$\in 0,1^m$ is the ground truth label.
Output: `L`$\in \mathbb{R}$ is the loss, and `dLdy` is the loss derivative with respect to the prediction.
Description: `Loss_euclidean` measure Euclidean distance $L = \|\mathbf{y} - \widetilde{\mathbf{y}}\|^2$.

`function [w, b] = TrainSLP_linear(mini_batch_x, mini_batch_y)`
Input: `mini_batch_x` and `mini_batch_y` are cells where each cell is a batch of images and labels.
Output: `w` $\in \mathbb{R}^{10\times 196}$ and `b` $\in \mathbb{R}^{10\times 1}$ are the trained weights and bias of a single-layer perceptron.
Description: You will use `FC`, `FC_backward`, and `Loss_euclidean` to train a single-layer perceptron using a stochastic gradient descent method where a pseudo-code can be found below. Through training, you are expected to see reduction of loss as shown in Figure 2(b). As a result of training, the network should produce more than 25% of accuracy on the testing data (Figure 2(c)).

Algorithm 1 Stochastic Gradient Descent based Training

Set the learning rate γ
Set the decay rate $\lambda \in (0, 1]$
Initialize the weights with a Gaussian noise $\mathbf{w} \in \mathcal{N}(0, 1)$
$k = 1$
for iIter = 1 : nIters **do**
    At every 1000^{th} iteration, $\gamma \leftarrow \lambda\gamma$
    $\frac{\partial L}{\partial \mathbf{w}} \leftarrow 0$ and $\frac{\partial L}{\partial \mathbf{b}} \leftarrow 0$
    for Each image $\mathbf{x}_i$ in k^{th} mini-batch **do**
        Label prediction of $\mathbf{x}_i$
        Loss computation l
        Gradient back-propagation of $\mathbf{x}_i$, $\frac{\partial l}{\partial \mathbf{w}}$ using back-propagation.
        $\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \mathbf{w}} + \frac{\partial l}{\partial \mathbf{w}}$ and $\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{b}} + \frac{\partial l}{\partial \mathbf{b}}$
    end for
    k++ (Set $k = 1$ if k is greater than the number of mini-batches.)
    Update the weights, $\mathbf{w} \leftarrow \mathbf{w} - \frac{\gamma}{R}\frac{\partial L}{\partial \mathbf{w}}$, and bias $\mathbf{b} \leftarrow \mathbf{b} - \frac{\gamma}{R}\frac{\partial L}{\partial \mathbf{b}}$
end for

4 Single-layer Perceptron



(a) Single-layer perceptron (b) Loss (c) Confusion

Figure 3: You will implement a single perceptron that produces accuracy near 90% on testing data.

You will implement a single-layer perceptron with *soft-max cross-entropy* using stochastic gradient descent method. We provide `main_slp` where you will implement `TrainSLP`. Unlike the single-layer linear perceptron, it has a soft-max layer that approximates a max function by clamping the output to $[0, 1]$ range as shown in Figure 3(a).

`function [L, dLdy] = Loss_cross_entropy_softmax(x, y)`
Input: `x` $\in \mathbb{R}^m$ is the input to the soft-max, and `y`$\in 0, 1^m$ is the ground truth label.
Output: `L`$\in \mathbb{R}$ is the loss, and `dLdy` is the loss derivative with respect to $\mathbf{x}$.
Description: `Loss_cross_entropy_softmax` measure cross-entropy between two distributions $L = \sum_i^m \mathbf{y}_i \log \widetilde{\mathbf{y}}_i$ where $\widetilde{\mathbf{y}}_i$ is the soft-max output that approximates the max operation by clamping $\mathbf{x}$ to $[0, 1]$ range:

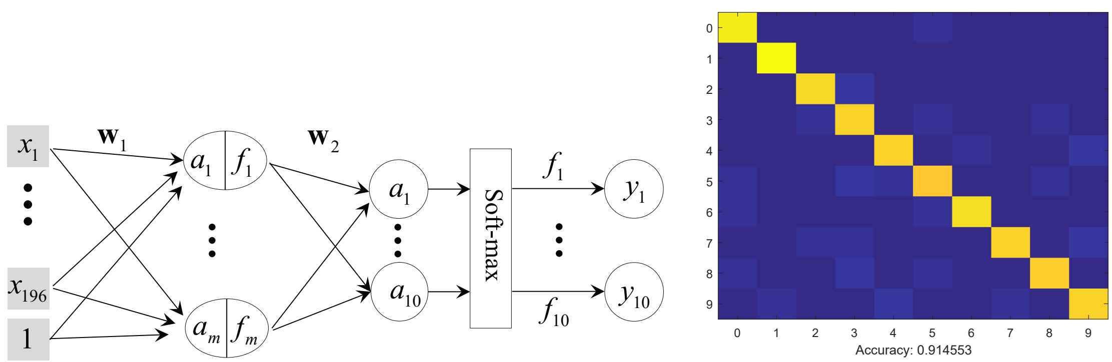
$$\widetilde{\mathbf{y}}_i = \frac{e^{\mathbf{x}_i}}{\sum_i e^{\mathbf{x}_i}},$$

where $\mathbf{x}_i$ is the i^{th} element of $\mathbf{x}$.

`function [w, b] = TrainSLP(mini_batch_x, mini_batch_y)`
Output: `w` $\in \mathbb{R}^{10\times196}$ and `b` $\in \mathbb{R}^{10\times1}$ are the trained weights and bias of a single-layer perceptron.
Description: You will use the following functions to train a single-layer perceptron using a stochastic gradient descent method: `FC`, `FC_backward`, `Loss_cross_entropy_softmax`

Through training, you are expected to see reduction of loss as shown in Figure 3(b). As a result of training, the network should produce more than 85% of accuracy on the testing data (Figure 3(c)).

5 Multi-layer Perceptron



(a) Multi-layer perceptron (b) Confusion

Figure 4: You will implement a multi-layer perceptron that produces accuracy more than 90% on testing data.

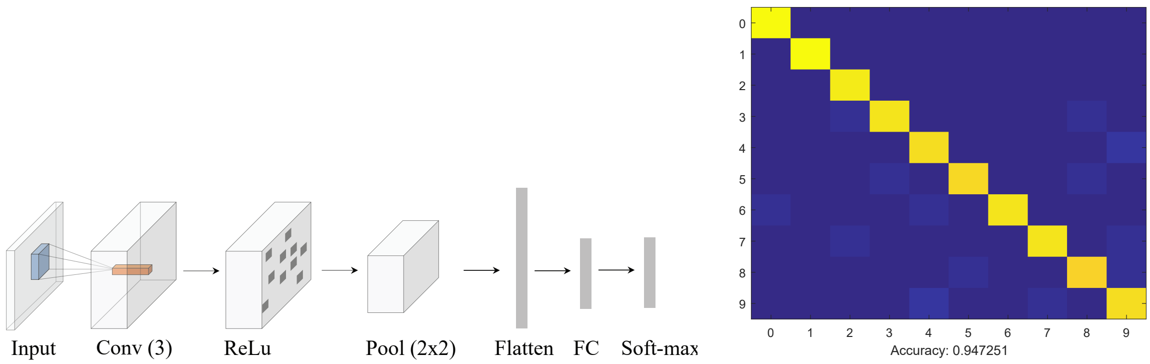
You will implement a multi-layer perceptron with a single hidden layer using a stochastic gradient descent method. We provide `main_mlp`. The hidden layer is composed of 30 units as shown in Figure 4(a).

`function [y] = ReLu(x)`
Input: `x` is a general tensor, matrix, and vector.
Output: `y` is the output of the Rectified Linear Unit (ReLu) with the same input size.
Description: `ReLu` is an activation unit ($\mathbf{y}_i = \max(0, \mathbf{x}_i)$). In some case, it is possible to use a Leaky ReLu ($\mathbf{y}_i = \max(\epsilon\mathbf{x}_i, \mathbf{x}_i)$ where $\epsilon = 0.01$).

`function [dLdx] = ReLu_backward(dLdy, x, y)`
Input: `dLdy` $\in \mathbb{R}^{1\times z}$ is the loss derivative with respect to the output $\mathbf{y} \in \mathbb{R}^z$ where z is the size of input (it can be tensor, matrix, and vector).
Output: `dLdx` $\in \mathbb{R}^{1\times z}$ is the loss derivative with respect to the input $\mathbf{x}$.

`function [w1, b1, w2, b2] = TrainMLP(mini_batch_x, mini_batch_y)`
Output: `w1` $\in \mathbb{R}^{30\times 196}$, `b1` $\in \mathbb{R}^{30\times 1}$, `w2` $\in \mathbb{R}^{10\times 30}$, `b2` $\in \mathbb{R}^{10\times 1}$ are the trained weights and biases of a multi-layer perceptron.
Description: You will use the following functions to train a multi-layer perceptron using a stochastic gradient descent method: `FC`, `FC_backward`, `ReLu`, `ReLu_backward`, `Loss_cross_entropy_softmax`. As a result of training, the network should produce more than 90% of accuracy on the testing data (Figure 4(b)).

6 Convolutional Neural Network



(a) CNN (b) Confusion

Figure 5: You will implement a convolutional neural network that produces accuracy more than 92% on testing data.

You will implement a convolutional neural network (CNN) using a stochastic gradient descent method. We provide `main_cnn`. As shown in Figure 4(a), the network is composed of: a single channel input ($14 \times 14 \times 1$) $\rightarrow$ Conv layer (3×3 convolution with 3 channel output and stride 1) $\rightarrow$ ReLu layer $\rightarrow$ Max-pooling layer (2×2 with stride 2) $\rightarrow$ Flattening layer (147 units) $\rightarrow$ FC layer (10 units) $\rightarrow$ Soft-max.

`function [y] = Conv(x, w_conv, b_conv)`
Input: $\mathtt{x} \in \mathbb{R}^{H \times W \times C_1}$ is an input to the convolutional operation, $\mathtt{w\_conv} \in \mathbb{R}^{H \times W \times C_1 \times C_2}$ and $\mathtt{b\_conv} \in \mathbb{R}^{C_2}$ are weights and bias of the convolutional operation.
Output: $\mathtt{y} \in \mathbb{R}^{H \times W \times C_2}$ is the output of the convolutional operation. Note that to get the same size with the input, you may pad zero at the boundary of the input image.
Description: This convolutional operation can be simplified using MATLAB built-in function `im2col`.

`function [dLdw, dLdb] = Conv_backward(dLdy, x, w_conv, b_conv, y)`
Input: `dLdy` is the loss derivative with respec to $\mathbf{y}$.
Output: `dLdw` and `dLdb` are the loss derivatives with respect to convolutional weights and bias $\mathbf{w}$ and $\mathbf{b}$, respectively.
Description: This convolutional operation can be simplified using MATLAB built-in function `im2col`. Note that for the single convolutional layer, $\frac{\partial L}{\partial \mathbf{x}}$ is not needed.

`function [y] = Pool2x2(x)`
Input: $\mathtt{x} \in \mathbb{R}^{H \times W \times C}$ is a general tensor and matrix.
Output: $\mathtt{y} \in \mathbb{R}^{\frac{H}{2} \times \frac{W}{2} \times C}$ is the output of the 2×2 max-pooling operation with stride 2.

`function [dLdx] = Pool2x2_backward(dLdy, x, y)`
Input: `dLdy` is the loss derivative with respect to the output $\mathbf{y}$.
Output: `dLdx` is the loss derivative with respect to the input $\mathbf{x}$.

`function [y] = Flattening(x)`
Input: $\mathtt{x} \in \mathbb{R}^{H\times W\times C}$ is a tensor.
Output: $\mathtt{y} \in \mathbb{R}^{HWC}$ is the vectorized tensor (column major).

`function [dLdx] = Flattening_backward(dLdy, x, y)`
Input: `dLdy` is the loss derivative with respect to the output $\mathbf{y}$.
Output: `dLdx` is the loss derivative with respect to the input $\mathbf{x}$.

`function [w_conv, b_conv, w_fc, b_fc] = TrainCNN(mini_batch_x, mini_batch_y)`
Output: $\mathtt{w\_conv} \in \mathbb{R}^{3\times3\times1\times3}$, $\mathtt{b\_conv} \in \mathbb{R}^{3}$, $\mathtt{w\_fc} \in \mathbb{R}^{10\times147}$, $\mathtt{b\_fc} \in \mathbb{R}^{147}$ are the trained weights and biases of the CNN.
Description: You will use the following functions to train a convolutional neural network using a stochastic gradient descent method: `Conv`, `Conv_backward`, `Pool2x2`, `Pool2x2_backward`, `Flattening`, `Flattening_backward`, `FC`, `FC_backward`, `ReLu`, `ReLu_backward`, `Loss_cross_entropy_softmax`. As a result of training, the network should produce more than 92% of accuracy on the testing data (Figure 5(b)).