UNIVERSITÄT DES SAARLANDES

High-Level Computer Vision

Summer Semester 2017

Prof. Bernt Schiele, Dr. Mario Fritz
<{schiele,mfritz}@mpi-inf.mpg.de>

Seon Joon Oh, Alina Dima, Rakshith Shetty
<{joon, aldima, rshetty}@mpi-inf-mpg.de>

Exercise 3: Deep Neural Networks and Backpropagation

Deep neural networks have shown staggering performances in various learning tasks, including computer vision, natural language processing, and sound processing. They have made the model designing more flexible by enabling end-to-end training.

In this exercise, we get to have a first hands-on experience with neural network training. Many frameworks (*e.g.* Caffe, Tensorflow, Theano) allow easy usage of deep neural networks without precise knowledge on the inner workings of backpropagation and gradient descent algorithms. We do not use any framework in this exercise; we implement the backpropagation and the gradient descent training algorithm from scratch. This shall equip you with a better understanding of how things tick, and later help you develop creative, out-of-framework models.

As a running example in this sheet, we consider the handwritten digit classification task. We provide 5000 20×20 grayscale images of handwritten digits $\in \{0, \cdots, 9\}$. The task is to code and train a parametrised model for classifying those images. This involves

- Implementing the feedforward model (Question 1).
- Implementing the backpropagation algorithm (gradient computation) (Question 2).
- Numerically verifying the gradient (Question 3).
- Implementing the (stochastic) gradient descent for training the model (Question 4).

Please untar the handed out `ex3.tar.gz`. The whole exercise is based on the script `ex3.m`. The script sets up the dataset and make calls to functions some of which you shall write. `ex3.m` is not to be modified; we will make clear which `.m` files are to be written by you.

Questions indicated by *report* should be answered in a separate report file - preferrably a pdf format file. Figures and derivations are allowed to be handwritten (and scanned), but LaTeXgenerated documents would be preferred.

The completed exercise should be handed in in a `.tar.gz` format which compresses all the (completed) `.m` files in the assignment `ex3.tar.gz` + the report.

Data To get a feeling of how input data looks like, run `Part A` of the script. It will crawl 5000 training examples in `ex3data1.mat` visualise a random subset of it. See Fig.1 for an example. Each image has dimension 20×20 of floating point numbers indicating the grayscale intensity at that location. The 20×20 grid of pixels is unrolled into a 400-dimensional vector. Each of these samples becomes a single column in our data matrix `X`.

The second part of the training set is a 5000-dimensional vector `y` that contains labels for the training set. To make thing more compatible with Matlab indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a "0" digit is labelled as "10", while the digits "1" to "9" are labelled as "1" to "9" in their natural order.

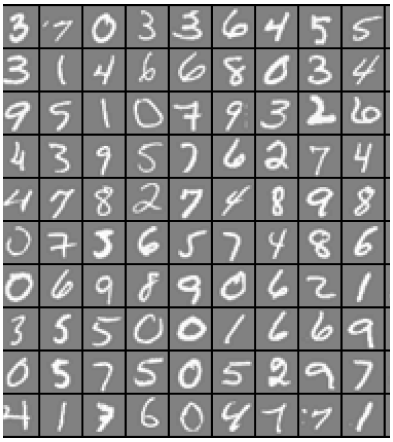

Figure 1: Data visualisation

Question 1: Implementing the feedforward model (10 points)

In this exercise, a neural network architecture (model function), as well as the loss function, will be given. It is your role to write code for the described architecture.

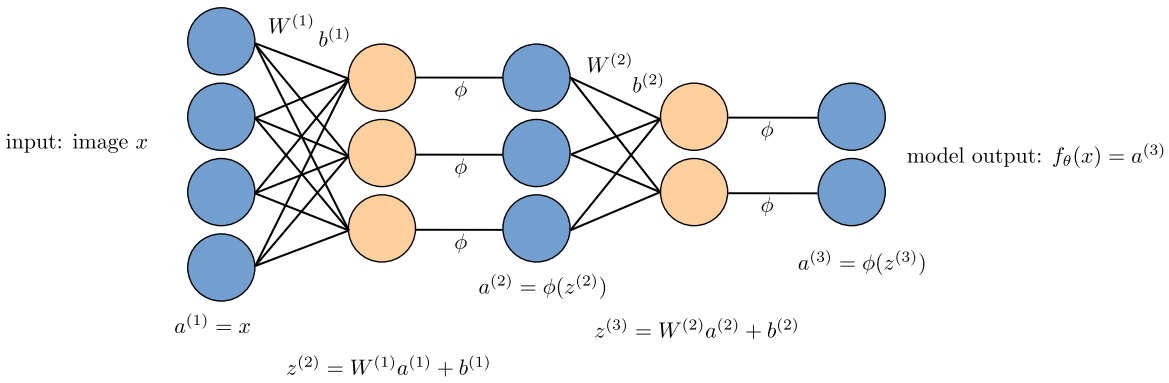

Figure 2: Visualisation of the neural network

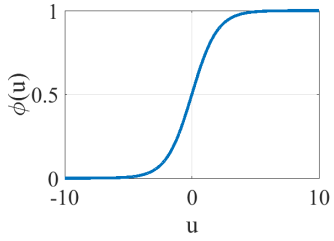

Figure 3: Plot of the sigmoid function ϕ

Model architecture Our architecture is shown in Fig.2. It has 3 layers – and input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20×20, this gives us 400 input layer units, represented by a vector $x \in \mathbb{R}^{400}$.

In the second layer, there are 25 hidden units. The first layer and the second layer are connected via linear weighting matrix $W^{(1)} \in \mathbb{R}^{25\times 400}$ and the bias term $b^{(1)} \in \mathbb{R}^{25}$. The parameters $W^{(1)}$ and $b^{(1)}$ are to be learnt later on. A linear operation is performed, $W^{(1)}x + b^{(1)}$, resulting in a 25 dimensional vector $z^{(2)}$. It is then followed by a sigmoid non-linear activation ϕ, applied element-wise on each unit, resulting in the activations $a^{(2)} = \phi(z^{(2)})$. Sigmoid function has the following form:

$$\phi(u) = \frac{1}{1 + e^{-u}}. \tag{1}$$

See Fig.3 for visualisation; the function is already implemented in `sigmoid.m`.

A similar linear operation is performed on $a^{(2)}$, resulting in $z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$, where $W^{(2)} \in \mathbb{R}^{10\times 25}$ and $b^{(2)} \in \mathbb{R}^{10}$; it is followed by the sigmoid activation to result in $a^{(3)} = \phi(z^{(3)})$. The final functional form of our model is thus defined by

$$f_\theta(x) := a^{(3)} = \phi(z^{(3)}) \tag{2}$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \tag{3}$$
$$a^{(2)} = \phi(z^{(2)}) \tag{4}$$
$$z^{(2)} = W^{(1)}a^{(1)} + b^{(1)} \tag{5}$$
$$a^{(1)} = x \tag{6}$$

which takes a flattened 20×20 resolution grayscale image as input and outputs a 10 dimensional vector, each entry in the output $f_k(x)$ representing the likelihood of image x corresponding to the digit k. We summarily indicate all the network parameters by $\theta = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$.

Pre-trained model For the purpose of validation, we provide a set of network parameters $\theta = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$ already trained by us. These are stored in `ex3weights.mat` and will be loaded by `ex3.m` into `nn_params` in `Part B`. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

Implementation We are now ready to implement the feedforward neural network.

a) Implement `feedForward.m` for the feedforward model. You are required to implement Eq.2 to 6. It generates the matrix of likelihoods for each digit for the matrix of input data `X`, given the network parameters `nn_params`. Verify the implementation by running `Part C` to see that the training set accuracy is 97.52%. (4 points)

b) We later guide the neural network parameters $\theta = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$ to fit to the given data and label pairs. We do so by minimising the loss function. A popular choice of the loss function for training neural networks is the log loss. The log loss for binary classification (*e.g.* if the digit is ≥ 5 or not) is defined by:

$$J(u) = \begin{cases} -\log u & \text{if } y = 1 \\ -\log(1-u) & \text{if } y = 0 \end{cases} \tag{7}$$

where $y = 1$ if digit is ≥ 5 and $y = 0$ otherwise. u is the predicted probability of digit being ≥ 5. Plot the function J, and convince yourself that it is convex with respect to u for fixed $y \in \{0, 1\}$ and attains the minimum when $u = y$. (*report*, 2 points)

Note that we can also write Eq.7 as

$$J(u) = -y \log u - (1-y) \log(1-u) \tag{8}$$

c) Handwritten digit classification is a multi-class task. The loss function can now be defined by:

$$J(u) = \sum_{k=1}^{K} \left(-y_k \log u_k - (1-y_k) \log(1-u_k)\right) \tag{9}$$

where y_k is 1 if x is digit k and 0 otherwise. Plugging in the neural network feedforward output $f(x)$ for input x into the model, and averaging over the whole training set, we get

$$J(\theta, \{x_i, y_i\}_{i=1}^N) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \left(-y_{ik} \log(f_\theta^k(x_i)) - (1-y_{ik}) \log(1 - f_\theta^k(x_i))\right) \tag{10}$$

where $f_\theta^k(x_i)$ is the output of the feedforward network f for digit k for the input x_i. Note that if the model has perfectly fitted to the data (*i.e.* $f_\theta^k(x_i) = 1$ whenever x_i is digit k and 0 otherwise), then J attains the minimum of 0. Implement the loss function in `nnLossFunction.m` and let it return the cost value. Verify the code by running `Part D` and matching the output cost 0.287629. (2 points)

d) We try to prevent overfitting by encoding our prior belief that the correct model should be simple (Occam's razor); we add an L_2 regularisation term over the model parameters θ. Specifically, the loss function is defined by:

$$\tilde{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \left(-y_{ik} \log(f_\theta^k(x_i)) - (1-y_{ik}) \log(1 - f_\theta^k(x_i))\right) + \frac{\lambda}{2} \left(||W^{(1)}||_2^2 + ||W^{(2)}||_2^2\right) \tag{11}$$

where $|| \cdot ||_2^2$ is the squared L_2 norm. For example,

$$||W^{(1)}||_2^2 = \sum_{p=1}^{25} \sum_{q=1}^{400} W_{pq}^{(1)2} \tag{12}$$

By changing the value of λ it is possible to give weights to your prior belief on the degree of simplicity (regularity) of the true model. Implement the regularisation term as above in `nnLossRegFunction.m` and verify the code by running `Part E` with $\lambda = 0.0002$ and matching the output cost 0.383770. (2 points)

Question 2: Backpropagation (10 points + 5 bonus points)

We train the model by solving

$$\min_\theta \tilde{J}(\theta) \tag{13}$$

via (stochastic) gradient descent. We therefore need an efficient computation of the gradients $\nabla_\theta \tilde{J}(\theta)$. We use backpropagation of top layer error signals to the parameters θ at different layers.

In this question, you will be required to implement the backpropagation algorithm yourself from a pseudocode. We will give a high-level description of what is happening at each line.

For those who are interested in the robust derivation of the algorithm, we include the optional exercise on the derivation of backpropagation algorithm. A prior knowledge on standard vector calculus including the chain rule would be helpful.

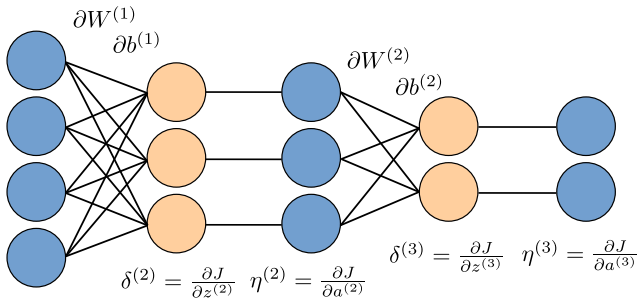

Figure 4: Visualisation of the gradients

Backpropagation Backpropagation algorithm is but a sequential application of chain rule. It is applicable to any (sub-) differentiable model that is a composition of simple building blocks. In this exercise, however, we focus on the architecture with stacked layers of linear transformation + sigmoid non-linear activation.

The intuition behind backpropagation algorithm is as follows. Given a training example (x, y), we first run the feedforward to compute all the activations throughout the network, including the output value of the model $f_\theta(x)$. Then, for each node k in layer l, we would like to compute an "error term" $\delta_k^{(l)}$ that measures how much that node was "responsible" for any errors in the final output. The "penalty" on the network parameters θ would be some form of multiplication between the "error terms" and the actual activation values on the nodes.

In detail, the backpropagation algorithm pseudocode is given in Alg.1 (also depicted in Fig.4).

Data: Training data $\{(x_i, y_i)\}_{i=1,\cdots,N}$, current network parameter θ, regularisation hyperparameter λ
Result: Gradient of the loss with respect to the network parameters
$\nabla_\theta \tilde{J}(\theta) = (\partial W^{(1)}, \partial b^{(1)}, \cdots, \partial W^{(L-1)}, \partial b^{(L-1)})$

Run forward pass to compute all the activations, $z_i^{(1)}, \cdots, z_i^{(L)}$, $a_i^{(1)}, \cdots, a_i^{(L)}$, for all the samples i;
Initialise gradients $\partial W^{(l)}, \partial b^{(l)} \leftarrow 0$ for all $l = 1, \cdots, (L-1)$.
for $i = 1, \cdots, N$ **do**
  for $k = 1, \cdots, K$ **do**
    $\eta_k^{(L)} \leftarrow -\frac{y_{ik}}{a_{ik}^{(L)}} + \frac{1-y_{ik}}{1-a_{ik}^{(L)}}$;
    // Gradient of J with respect to last layer sigmoid activations $a_i^{(L)}$
  end
  $\delta^{(L)} \leftarrow \eta^{(L)} \cdot \phi(z_i^{(L)}) \cdot (1 - \phi(z_i^{(L)}))$;
  // Gradient of J with respect to last layer pre-sigmoid activations $z_i^{(L)}$
  for $l = (L-1), \cdots, 2$ **do**
    $\partial W^{(l)} \leftarrow \partial W^{(l)} + \frac{1}{N}\delta^{(l+1)} a_i^{(l)T}$;
    // Accumulate per-sample gradients of J with respect to layer l parameters $W^{(l)}$
    $\partial b^{(l)} \leftarrow \partial b^{(l)} + \frac{1}{N}\delta^{(l+1)}$;
    // Accumulate per-sample gradients of J with respect to layer l parameters $b^{(l)}$
    $\eta^{(l)} \leftarrow W^{(l)T}\delta^{(l+1)}$;
    // Gradient of J with respect to layer l sigmoid activations $a_i^{(l)}$
    $\delta^{(l)} \leftarrow \eta^{(l)} \cdot \phi(z_i^{(l)}) \cdot (1 - \phi(z_i^{(l)}))$;
    // Gradient of J with respect to layer l pre-sigmoid activations $z_i^{(l)}$
  end
  $\partial W^{(1)} \leftarrow \partial W^{(1)} + \frac{1}{N}\delta^{(2)} a_i^{(1)T}$;
  // Accumulate per-sample gradients of J with respect to layer l parameters $W^{(1)}$
  $\partial b^{(1)} \leftarrow \partial b^{(1)} + \frac{1}{N}\delta^{(2)}$;
  // Accumulate per-sample gradients of J with respect to layer l parameters $b^{(1)}$
end
for $l = L-1, \cdots, 1$ **do**
  $\partial W^{(l)} \leftarrow \partial W^{(l)} + \lambda W^{(l)}$;
  // Compute gradient wrt $\tilde{J}$ by augmenting the regulariser gradient
end

Algorithm 1: Backpropagation algorithm

Backpropagation algorithm is performed from the top of the network (loss layer) towards the bottom. It

sequentially computes the gradient of the loss function with respect to each layer activations and parameters. We denote the gradients with respect to the sigmoid activations $a^{(l)}$ and pre-sigmoid activations $z^{(l)}$ by $\eta^{(l)}$ and $\delta^{(l)}$, respectively. These gradients are computed sequentially, later being used for the computations of parameter gradients $\partial W^{(l)}$ and $\partial b^{(l)}$ for each layer.

At line 15, we denote the matrix-vector multiplication by $W^{(l)T}\delta^{(l+1)}$. At lines 8 and 17, element-wise multiplications are denoted by the centre dot " $\cdot$ ". At line 11, a vector-vector outer product is performed $\delta^{(l+1)}a^{(l)T}$, resulting in a matrix of size corresponding to $\partial W^{(l)}$. When in doubt, try to match the dimensions in the array operations. Note that the gradients $\eta^{(l)}$, $\delta^{(l)}$, $\partial W^{(l)}$ and $\partial b^{(l)}$ must have the same dimension as the corresponding variables, $a^{(l)}$, $z^{(l)}$, $W^{(l)}$, and $b^{(l)}$, respectively.

After all the parameter gradients of the unregularised loss J is computed, the gradient for the L_2 regulariser is added (line 22) to compute the gradients for $\tilde{J}$.

a) Implement the Alg.1 in `nnGradient.m`. (8 points)

Implementing the backpropagation algorithm can be prone to errors because one needs to deal with multi-dimensional tensors. In the next question, we will introduce a method for numerically verifying the backpropagation algorithm. We recommend iterating between Question 2 and 3 to make sure that the backpropagation implementation gives numerically plausible output.

Analytic derivation of gradients At the core of the backpropagation is the vector calculus and chain rule. In this section, we derive the Alg.1 from vector calculus.

b) Verify that the sigmoid function defined in Eq.1 has the gradient

$$\frac{\partial \phi}{\partial u}(u) = \phi(u)(1 - \phi(u)). \tag{14}$$

(*report*, 2 points)

This gradient can be found at lines 8 and 13 of Alg.1.

c) For J as defined in Eq.9, verify that

$$\frac{\partial J}{\partial u_p}(u) = -\frac{y_p}{u_p} + \frac{1 - y_p}{1 - u_p}. \tag{15}$$

(optional) (*report*, 1 bonus point)

This gradient corresponds to the line 5 of Alg.1.

d) Verify that the L_2 regularisation term defined as $R(u) = \frac{1}{2}\sum_{j=1}^{J} u_j^2$ has the gradient

$$\frac{\partial R}{\partial u_p}(u) = u_p. \tag{16}$$

(optional) (*report*, 1 bonus point)

This computation corresponds to line 22 of Alg.1.

e) Using the multivariate chain rule and the results above, derive the backpropagation algorithm. (optional) (*report*, 3 bonus points)

Question 3: Numerical gradient verification (10 points)

In the previous question, you have implemented the backpropagation algorithm. In this question, we verify the implementation by comparing the output from the implementation and the numerical gradients. It is possible to numerically approximate the partial derivative of $\tilde{J}$ with respect to an entry θ_p by

$$\frac{\partial \tilde{J}}{\partial \theta_p}(\theta) \approx \frac{\tilde{J}(\theta + \epsilon \mathbf{e}_p) - \tilde{J}(\theta - \epsilon \mathbf{e}_p)}{2\epsilon} \tag{17}$$

where $\mathbf{e}_p$ indicates the one-hot vector with one at index p.

a) Implement the numerical gradient computation in `computeNumericalGradient.m` following Eq.17. (7 points)

b) We decide whether the backpropagation algorithm produces the correct output by comparing it against the numerical gradient. Upon completing the previous part, run `Part F`; it computes the normalised L_2 distance between the two outputs. If the backpropagation algorithm has been implemented correctly, the value should be less than 10^{-10}. Report the distance and comment on the result. (*report*, 3 points)

Question 4: Gradient descent training (10 points)

We have implemented the backpropagation algorithm for computing the parameter gradients and have verified that it indeed gives the correct gradient. We are now ready to train the network. We solve Eq.13 with the (stochastic) gradient descent.

Parameter initialisation Typically neural network parameters are initialised with some random distribution. In this exercise, we take *i.i.d.* samples from a uniform distribution. It is already implemented in `randInitializeWeights.m` and will be loaded to the script in `Part G`.

Gradient descent The vanilla gradient descent algorithm is as follows:

Data: Training data $\{(x_i, y_i)\}_{i=1,\cdots,N}$, initial network parameter $\theta^{(0)}$, regularisation hyperparameter λ, learning rate α, iteration limit T
Result: Trained parameter $\theta^{(T)}$
1 **for** $t = 1, \cdots, T$ **do**
2 $\quad v \leftarrow -\alpha \nabla_\theta \tilde{J}(\theta^{(t-1)})$;
3 $\quad \theta^{(t)} \leftarrow \theta^{(t-1)} + v$;
4 **end**

Algorithm 2: Gradient descent algorithm

Intuitively, $v = -\nabla_\theta \tilde{J}(\theta^{(t-1)})$ gives the direction to which the loss $\tilde{J}$ decreases the most (locally), and therefore we follow that direction by updating the parameters towards that direction $\theta^{(t)} = \theta^{(t-1)} + v$.

One could introduce the notion of momentum for a better convergence behaviour

Data: Training data $\{(x_i, y_i)\}_{i=1,\cdots,N}$, initial network parameter $\theta^{(0)}$, regularisation hyperparameter λ, learning rate α, momentum β, iteration limit T
Result: Trained parameter $\theta^{(T)}$
1 Initialise momentum vector $v \leftarrow \mathbf{0}$
2 **for** $t = 1, \cdots, T$ **do**
3 $\quad v \leftarrow \beta v - \alpha \nabla_\theta \tilde{J}(\theta^{(t-1)})$;
4 $\quad \theta^{(t)} \leftarrow \theta^{(t-1)} + v$;
5 **end**

Algorithm 3: Gradient descent with momentum

Momentum is chosen within the range $\beta \in [0, 1)$, and a popular choice is 0.9. Momentum has the effect of smoothing out the gradient descent directions. Note that when $\beta = 0$, we retrieve the vanilla gradient descent.

a) Implement the `gradientStep.m` function according to lines 3 and 4 in Alg.3; the function is called by `gradientDescent.m` which implements Alg.3. Run `Part H` to confirm that the loss indeed decreases over iterations. With the given training hyperparameters ($T = 100$, $\alpha = 0.1$, $\beta = 0.9$, $\lambda = 0.0002$), the accuracy should be around 77% – may be slightly different due to the random initialisation. The accuracy is automatically computed at the end of `Part H`. (5 points)

Stochastic gradient descent (SGD) Typically neural networks are much larger and are trained with more data – millions or billions of data. It is thus often infeasible to compute the gradient $\nabla_\theta \tilde{J}(\theta)$ that requires the accumulation of the gradient over the entire training set. Stochastic gradient descent addresses this problem by simply accumulating the gradient over a small random subset of the training samples (minibatch) at each iteration. Specifically, the algorithm is as follows

Data: Training data $\{(x_i, y_i)\}_{i=1,\cdots,N}$, initial network parameter $\theta^{(0)}$, regularisation hyperparameter λ, learning rate α, momentum β, batch size B, iteration limit T
Result: Trained parameter $\theta^{(T)}$

Initialise momentum vector $v \leftarrow \mathbf{0}$
for $t = 1, \cdots, T$ **do**
  $\{(X'_j, y'_j)\}_{j=1}^{B} \leftarrow$ a random subset of the original training set $\{(X_i, y_i)\}_{i=1}^{N}$;
  $v \leftarrow \beta v - \alpha \nabla_\theta \tilde{J}(\theta^{(t-1)}, \{(X'_j, y'_j)\}_{j=1}^{B})$;
  $\theta^{(t)} \leftarrow \theta^{(t-1)} + v$;
end

Algorithm 4: Stochastic gradient descent with momentum

where the gradient $\nabla_\theta \tilde{J}(\theta, \{(X'_j, y'_j)\}_{j=1}^{B})$ is with respect to the partial loss

$$\tilde{J}(\theta, \{(X'_j, y'_j)\}_{j=1}^{B}) = \frac{1}{B} \sum_{j=1}^{B} \sum_{k=1}^{K} \left(-y'_{jk} \log(f_\theta^k(x'_j)) - (1 - y'_{jk}) \log(1 - f_\theta^k(x'_j))\right) + \frac{\lambda}{2} \left(||W^{(1)}||_2^2 + ||W^{(2)}||_2^2\right) \tag{18}$$

which only sums over the subset $\{(X'_j, y'_j)\}_{j=1}^{B}$, and therefore also only accumulates the gradient only for this subset.

b) Implement the stochastic gradient descent algorithm in `stochasticGradientDescent.m` and run `Part I` with the given hyperparameters ($T = 10000$, $\alpha = 0.1$, $\beta = 0.9$, $B = 10$, $\lambda = 0.0002$) to check that the trained network gives the accuracy around 97%. (5 points)

Please turn in your solution by sending an email to Seong Joon Oh<joon@mpi-inf.mpg.de> including all relevant m-files and the report before Monday, June 5th, 23:59. in a single .zip or tar.gz file