CSC 321 assignment 2: Optimization and Generalization

1.1 Part 2: Programming

Computation of the weights for the neural network is derived here. Backpropagation algorithm is used in this exercise.

First, the formulas related to this part of the exercise are committed to paper. Consider only one training case for now. The input value of a unit, i, in the input layer is denoted by x_i. The weight of the connection between an input unit, i, and a hidden unit, j, is denoted by W_{ji}^{xh}. Now, the logit, u_j, of a hidden unit, j, is given by:

$$u_j = \sum_{i \in \text{input}} W_{ji}^{xh} x_i, \tag{1}$$

Applying logistic function to u_j gives the output value, h_j, of a hidden unit, j:

$$h_j = \frac{1}{1 + e^{-u_j}}, \tag{2}$$

Let us denote the weight of the connection between a hidden unit, j, and an output unit, k, by W_{kj}^{hy}. Now, the logit, z_k, of an output unit, k, is given by:

$$z_k = \sum_{j \in \text{hidden}} W_{kj}^{hy} h_j, \tag{3}$$

The output value, y_k, of an output unit, k, in a softmax group is calculated as:

$$y_k = \frac{e^{z_k}}{\sum_{k' \in \text{output}} e^{z_{k'}}}, \tag{4}$$

The target value of an output unit l is denoted by t_l. Because the neural network is trained to recognise a handwritten digit, it holds true that $t_k = 1$ for only a single $k \in$ output, and for each $l \in$ output ($l \neq k$) it holds that $t_l = 0$. Therefore, $\sum_{l \in \text{output}} t_l = 1$ The cross-entropy error is often used as a cost function, when the output layer of a neural

network is a softmax group. The equation of the cross-entropy error is:

$$C = -\sum_{k' \in \text{output}} t_{k'} \log y_{k'}, \tag{5}$$

Consider the backpropagation algorithm. It is used to efficiently compute for a traning case how the error will change with respect to change in a weight for each of the weights in the network. The main idea of the backpropagation is computing error derivatives in a layer by using the error derivatives from the layer above. Hence, let us first take the derivative of the error, C, with respect to the output, y_l, of an output unit, l:

$$\frac{\partial C}{\partial y_l} = -\frac{t_l}{y_l} \tag{6}$$

Next, as an intermediate result, let us calculate the derivative of y_l with respect to z_k. Because y_l depends on each z_k for all $k \in$ output, it is necessary to consider two cases: $l \neq k$ and $l = k$. First, let us assume that $l \neq k$. This gives:

$$\frac{\partial y_l}{\partial z_k} = 0 + e^{z_l}\left(-\frac{1}{(\sum_{k' \in \text{output}} e^{z_{k'}})^2}\right) e^{z_k} = -y_l y_k. \tag{7}$$

Now, it is calculated with the assumption that $l = k$. This gives:

$$\frac{\partial y_l}{\partial z_k} = \frac{\partial y_k}{\partial z_k} = e^{z_k}\frac{1}{\sum_{k' \in \text{output}} e^{z_{k'}}} + e^{z_k}\left(-\frac{1}{(\sum_{k' \in \text{output}} e^{z_{k'}})^2}\right) e^{z_k} = y_k - y_k^2 = y_k(1 - y_k). \tag{8}$$

The combination of the equations 7 and 8 is a pair of equations:

$$\frac{\partial y_l}{\partial z_k} = \begin{cases} -y_l y_k, & \text{if } l \neq k \\ y_k(1 - y_k), & \text{if } l = k \end{cases} \tag{9}$$

Then, the derivative of C with respect to z_k is calculated:

$$\begin{aligned}\frac{\partial C}{\partial z_k} &= \sum_{l\in\text{output}} \frac{\partial C}{\partial y_l}\frac{\partial y_l}{\partial z_k} = \sum_{l\neq k} \frac{\partial C}{\partial y_l}\frac{\partial y_l}{\partial z_k} + \frac{\partial C}{\partial y_k}\frac{\partial y_k}{\partial z_k} \\ &= \sum_{l\neq k} \frac{-t_l}{y_l}(-y_l y_k) - \frac{t_k}{y_k} y_k(1-y_k) = y_k \sum_{l\neq k} t_l + y_k t_k - t_k \\ &= y_k \sum_{l\in\text{output}} t_l - t_k = y_k - t_k\end{aligned} \tag{10}$$

Applying the chain rule, let us calculate the derivative of the error function, C, with respect to a weight W^{hy}_{kj}, and a weight W^{xh}_{ji}:

$$\frac{\partial C}{\partial W^{hy}_{kj}} = \frac{\partial C}{\partial z_k}\frac{\partial z_k}{\partial W^{hy}_{kj}} = (y_k - t_k)h_j, \tag{11}$$

$$\begin{aligned}\frac{\partial C}{\partial W^{xh}_{ji}} &= \frac{\partial C}{\partial z_k}\frac{\partial z_k}{\partial h_j}\frac{\partial h_j}{\partial u_j}\frac{u_j}{W^{xh}_{ji}} = (y_k - t_k)W^{hy}_{kj}\frac{-1}{(1+e^{-u_j})^2}(-e^{-u_j})x_i \\ &= (y_k - t_k)W^{hy}_{kj}(h_j - h_j^2)x_i = (y_k - t_k)W^{hy}_{kj}h_j(1-h_j)x_i,\end{aligned} \tag{12}$$

Programming the above backpropagation algorithm with MATLAB benefits from matrix calculations. Hence, let us now consider multiple training cases using matrices. The following formulas are obtained as matrix-versions of the previously derived equations. Let n be the number of training cases. The input matrix is denoted by $X \in \mathbb{R}^{256\times n}$. Its rows correspond to the 256 different pixel values and its columns correspond to the n training cases. The weight matrix between the input layer and the hidden layer is denoted by $W^{xh} \in \mathbb{R}^{m\times 256}$. Its rows correspond to the m hidden units and its columns correspond to the 256 different pixel values. The logit matrix of the hidden layer is $U = W^{xh}X \in \mathbb{R}^{m\times n}$, where its rows correspond to the m hidden units and its columns correspond to the n training cases. Applying the logistic function to the logit matrix above gives $H = \sigma(U) \in \mathbb{R}^{m\times n}$. The weight matrix between the hidden layer and the output layer is denoted by $W^{hy} \in \mathbb{R}^{10\times m}$. Its rows correspond to the 10 possible output values and its columns correspond to the m hidden units. The logit matrix of the output layer is $Z = W^{hy}H \in \mathbb{R}^{10\times n}$, where its rows correspond to the 10 possible output values and its columns correspond to the n training cases. Then, the softmax func-

tion is applied to the logit matrix of the output layer: $Y = \text{softmax}(Z) \in \mathbb{R}^{10\times n}$. The target matrix containing the target value for each of the n training cases is denoted by $T \in \mathbb{R}^{10\times n}$. Only a single element of each column of this matrix is marked with the value of 1 corresponding to the class of a training case. Other elements in a column are marked as zeroes. The error function is an average over training cases and, hence, is of the following form:

$$C = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k'\in\text{output}} T_{k'i}\log Y_{k'i}. \tag{13}$$

Now the elements of the matrix $\dfrac{\partial C}{\partial W^{hy}}$ are of interest. Let us calculate them using the equation 11, this gives

$$\left(\frac{\partial C}{\partial W^{hy}}\right)_{kj} = \frac{1}{n}\sum_{i=1}^{n}(Y_{ki}-T_{ki})H'_{ij} = \frac{1}{n}\left((Y-T)H'\right)_{kj}. \tag{14}$$

Therefore

$$\frac{\partial C}{\partial W^{hy}} = \frac{1}{n}(Y-T)H'. \tag{15}$$

Similarly, for the element of the matrix $\dfrac{\partial C}{\partial W^{xh}}$ using the equation 12 gives

$$\begin{aligned}\left(\frac{\partial C}{\partial W^{xh}}\right)_{ji} &= \frac{1}{n}\sum_{l=1}^{n}(Y_{kl}-T_{kl})W^{hy}_{kj}H_{jl}(1-H_{jl})X_{il}\\ &= \frac{1}{n}W^{hy'}_{jk}\sum_{l=1}^{n}(Y-T)_{kl}H_{jl}(1-H_{jl})X'_{li}.\end{aligned} \tag{16}$$

Now

$$\frac{\partial C}{\partial W^{xh}} = \frac{1}{n}\left(W^{hy'}(Y-T)\odot H\odot(1-H)\right)X', \tag{17}$$

where $\odot$ is the element-wise matrix product.

The Matlab code is presented below.

```
function ret = d_loss_by_d_model(model, data, wd_coefficient)
  % model.input_to_hid is a matrix of size <number of hidden units> by
  % <number of inputs i.e. 256>
  % model.hid_to_class is a matrix of size <number of classes i.e. 10> by
```

```
  % <number of hidden units>
  % data.inputs is a matrix of size <number of inputs i.e. 256> by
  % <number of data cases>
  % data.targets is a matrix of size <number of classes i.e. 10> by
  % <number of data cases>

  % The returned object <ret> is supposed to be exactly like parameter
  % <model>, i.e. it has fields ret.input_to_hid and ret.hid_to_class,
  % and those are of the same shape as they are in <model>.
  % However, in <ret>, the contents of those matrices are gradients
  % (d loss by d weight), instead of weights.

  % This is the only function that you're expected to change. Right now,
  %it just returns a lot of zeros, which is obviously not the correct output.
  % Your job is to change that.
  [hid_input, hid_output, class_input, log_class_prob, class_prob] = forward_pass(model, data);

  n_training_cases = size(data.inputs, 2);
  deviation = (class_prob - data.targets);
  ret.input_to_hid = model.hid_to_class' * deviation .* hid_output .* (1 - hid_output) ...
      * data.inputs' ./ n_training_cases;
  ret.hid_to_class = deviation * hid_output' ./ n_training_cases;
end
```

The classification loss (i.e. without weight decay) on the training data is 2.301907, when *a2(0, 10, 30, 0.01, 0, false, 10)* is run.

1.2 Part 3: Optimization

In this part of the exercise, 7 different learning rates are tested. In total 14 experiments are run, because one of the test series does not include momentum, but the other one includes a momentum of 0.9. The results are shown in Figures 1-7 for the learning rates in the following order: 0.002, 0.01, 0.05, 0.2, 1.0, 5.0, and 20.0. The experiment run without a momentum is shown on the left side; the experiment run with a momentum of 0.9 is shown on the right side.

The experiment with a learning rate of 0.2 and with a momentum of 0.9 gave the best result – the total loss on the training data was as low as 1.292496. By testing other learning rates around the 0.2 learning rate, it is possible to find even better learning rate such as 0.36, which leads to the loss on training data of 1.109546.

Based on Figures 1-4, optimizing with momentum is smoother and quicker than optimizing without it, when learning rates are low (i.e. in this case ≤ 0.2). However, with a higher learning (e.g. 1.0 as in Figure 5) the optimizer using momentum produces oscillating loss graph with higher values than the optimizer without any momentum, which generates quite stable optimization graph. Hence, optimizing with momentum, when higher learning rates are in question seems problematic and the optimizer with zero momentum is then preferable. There seems to be no difference between the optimizers with very high learning rates as can be seen from Figures 6-7. In this case both optimizers perform poorly by quickly stabilizing on a specific value, which seems to be the highest (or worst) compared to loss graphs produced with other learning rates.

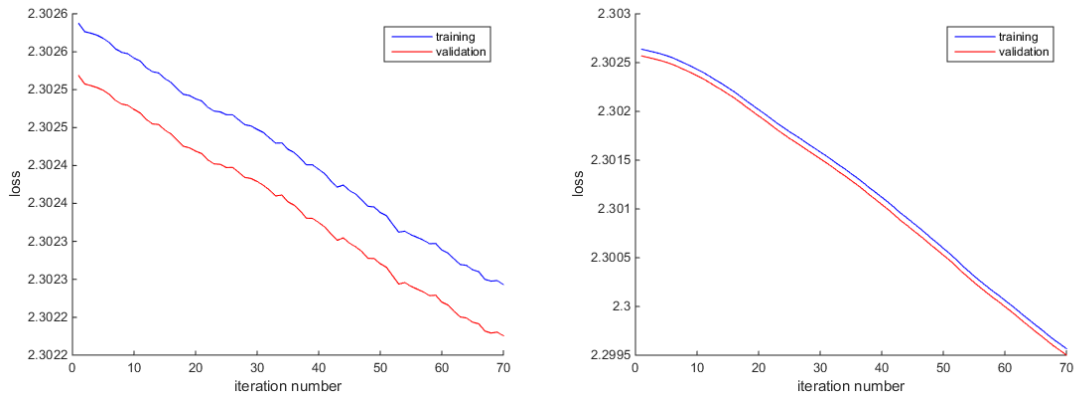

Figure 1: The total loss on the training and validation data with a learning rate of 0.002. Without momentum on the left. With a momentum of 0.9 on the right.

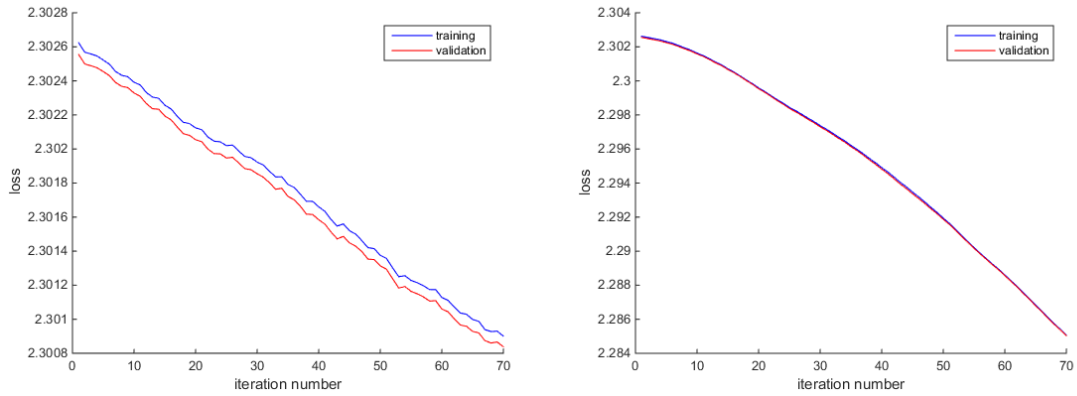

Figure 2: The total loss on the training and validation data with a learning rate of 0.01. Without momentum on the left. With a momentum of 0.9 on the right.

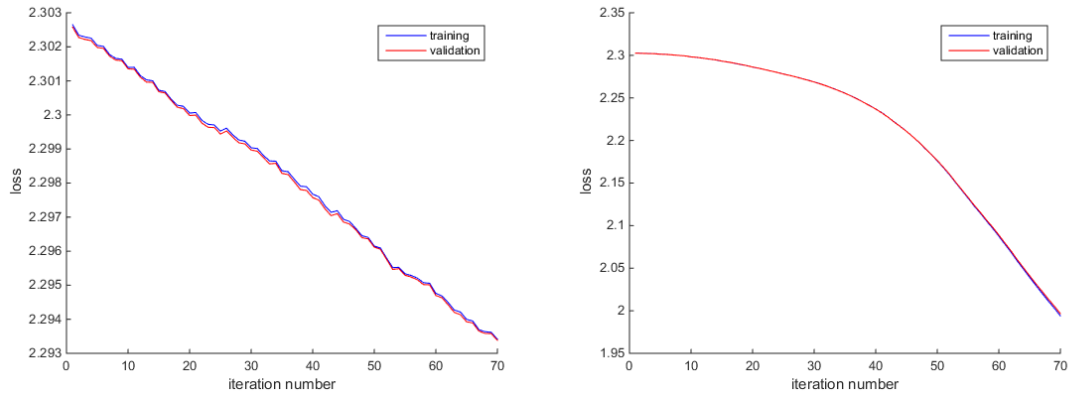

Figure 3: The total loss on the training and validation data with a learning rate of 0.05. Without momentum on the left. With a momentum of 0.9 on the right.

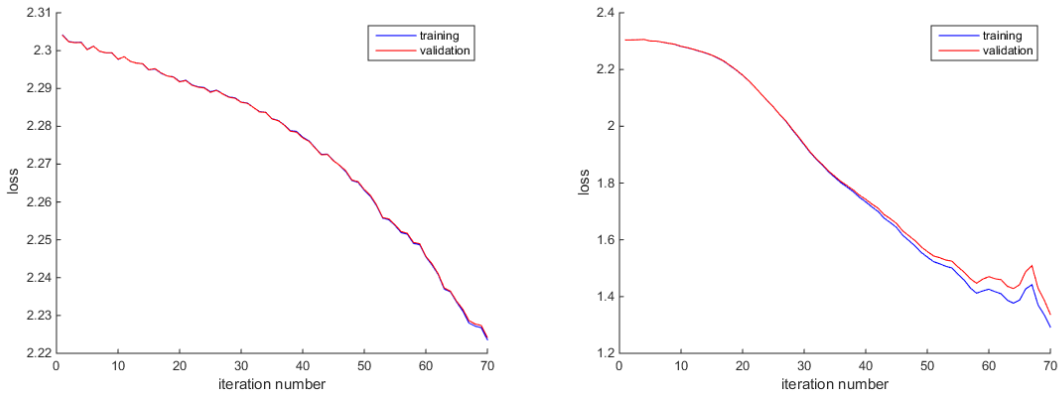

Figure 4: The total loss on the training and validation data with a learning rate of 0.2. Without momentum on the left. With a momentum of 0.9 on the right.

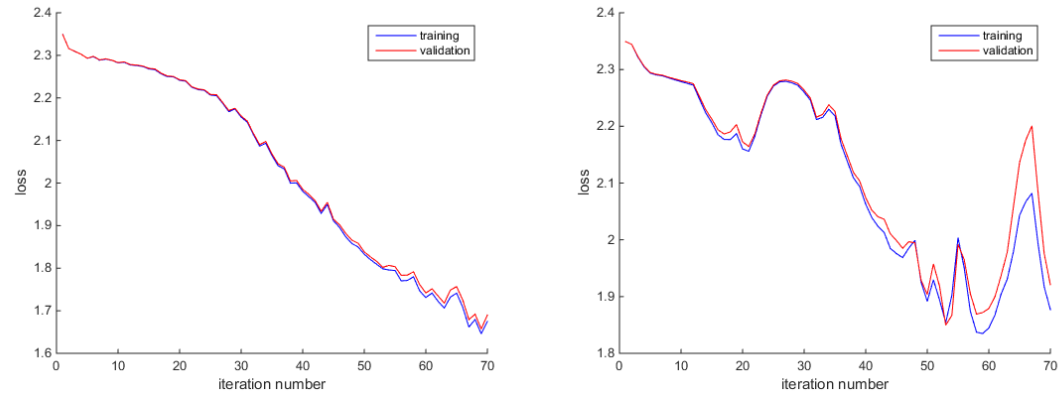

Figure 5: The total loss on the training and validation data with a learning rate of 1.0. Without momentum on the left. With a momentum of 0.9 on the right.

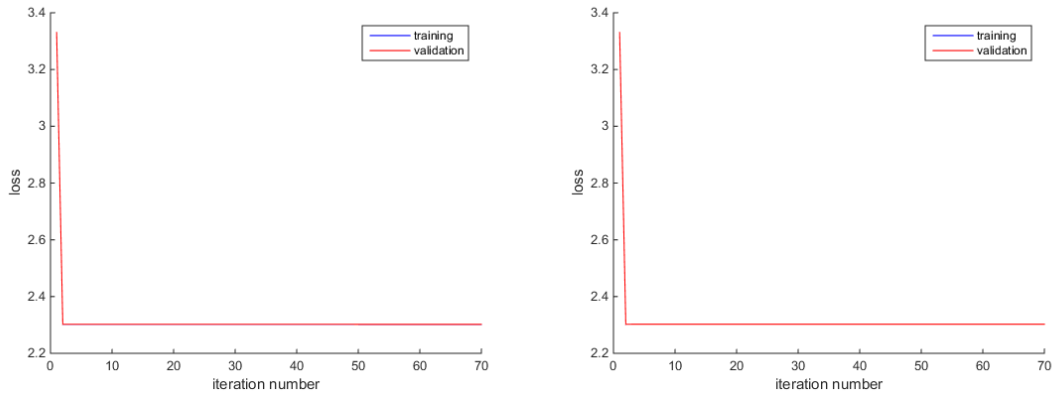

Figure 6: The total loss on the training and validation data with a learning rate of 5.0. Without momentum on the left. With a momentum of 0.9 on the right.

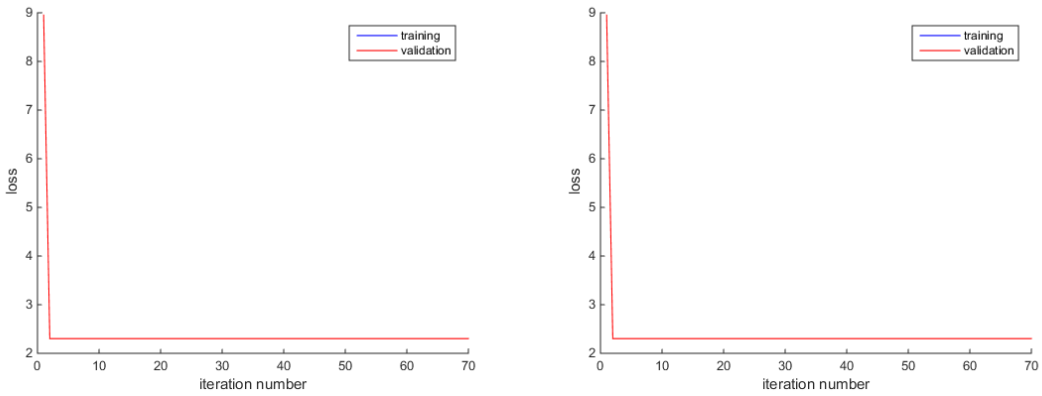

Figure 7: The total loss on the training and validation data with a learning rate of 20.0. Without momentum on the left. With a momentum of 0.9 on the right.