CSC 411: Assignment #2

Xiangyu Kong **Yun Lu**
kongxi16 **luyun5**

February 23, 2018

Problem 1

The data set contains hand-written digits from 0 to 9 (Fig 1). Among these data, most data are labeled accurately. However, some data are hard to be distinguished and even human can't really predict the digit.(Fig 2)

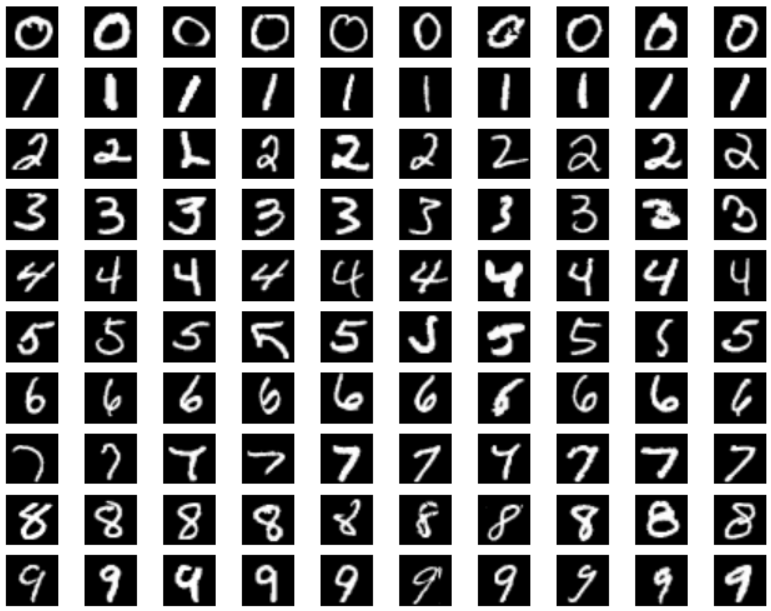
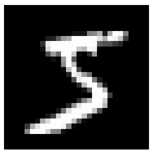
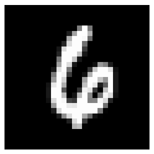


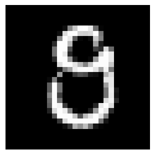
Figure 1: Full data



(a) 5 but looks like a 3



(b) 6 but looks like a 4



(c) 9 but looks like an 8

Figure 2: Inaccurate Labels

Problem 2

The output should be:

$$o^{(i)} = \sum_j w_j x_j^{(i)} + b^{(i)}$$

The listing of the implementation is as follows:

Listing 1: code for linear net output

```
def linear_forward(x, W):
    lin_output = np.dot(W.T, x)
    return softmax(lin_output)
```

Problem 3

1. Let the loss function C be defined as:

$$C = -\sum_i y^{(i)} log(p_i)$$

where p_i is:

$$p_i = \frac{e^{o_i}}{\sum_j e^{o_j}}$$

and o_i is:

$$o_i = \sum_j x_j^{(i)} w_{ij} + b_i$$

The gradient for the loss function with respect to the weight w_{ij} $\dfrac{\partial C}{\partial w_{ij}}$ is:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial o_i}\frac{\partial o_i}{\partial w_{ij}}$$

Note:

$$\frac{\partial p_k}{\partial o_i} = \begin{cases} -p_k p_i & \text{if } k \neq i \\ p_i(1-p_i) & \text{if } k = i \end{cases}$$

Then:

$$\begin{aligned} \frac{\partial C}{\partial o_i} &= -\frac{\partial C}{\partial p_i}\frac{\partial p_i}{\partial o_i} + \sum_{k \neq i} \frac{\partial C}{\partial p_k}\frac{\partial p_k}{\partial o_i} \\ &= -\frac{y^{(i)}}{p_i} p_i(1-p_i) + \sum_{k\neq i} \frac{y^{(k)}}{p_k} p_k p_i \\ &= p_i y^{(i)} - y^{(i)} + \sum_{k \neq i} y^{(k)} p_i \\ &= \sum_k y^{(k)} p_i - y^{(i)} \\ &= p_i - y^{(i)} \end{aligned}$$

Also,

$$\frac{\partial o_i}{\partial w_{ij}} = x_j^{(i)}$$

Then

$$\begin{aligned} \frac{\partial C}{\partial w_{ij}} &= \frac{\partial C}{\partial o_i}\frac{\partial o_i}{\partial w_{ij}} \\ &= x_j^{(i)}(p_i - y^{(i)}) \end{aligned}$$

2. The vectorized implementation is as follows:

Listing 2: Vectorized gradient

```
def linear_forward(x, W):
    lin_output = np.dot(W.T, x)
    return softmax(lin_output)

def loss(x, W, y):
    p = linear_forward(x, W)
    return -np.sum(y * np.log(p)) / x.shape[1]

def dlossdw(x, W, y):
    p = linear_forward(x, W)
    return np.matmul((p - y), x.T).T
```

Problem 4

The learning curve is plotted in Fig 3, and the weights going to the output is in Fig 4. The learning curve is as expected: the loss for the training set decreases gradually and the loss for the test set first decreases then increases. The weights' visualization gives each handwritten digits form 0 to 9.
The weights are initialized to 0.5. This is because if the weights are initialized to 0, the network will not change at all. The learning rate is set to 0.00005. This is the best value for the network to change and does not overstep the minima

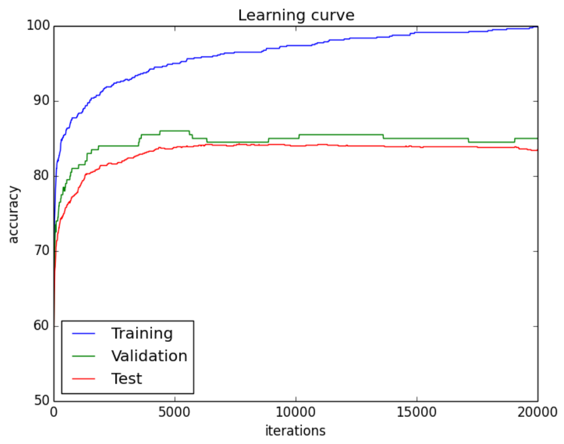


Figure 3: learning curve

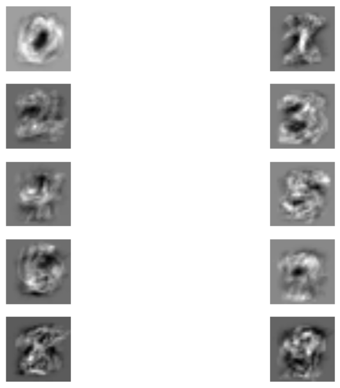


Figure 4: weights

Problem 5

The new code for gradient descent with momentum is as follows:

Listing 3: Gradient descent with momentum

```
while i < max_iter and norm(W - prev_W) > eps:
        prev_W = W.copy()
        # W-= alpha * dlossdw(x_train, W, y_train)
        V = gamma * V + alpha * dlossdw(x_train, W, y_train)
        W -= V
        i += 1
```

The learning curve is plotted in Fig 5. Comparing the learning curve to Part 4, the training set learns faster and the test set first decreases and then keeps constant

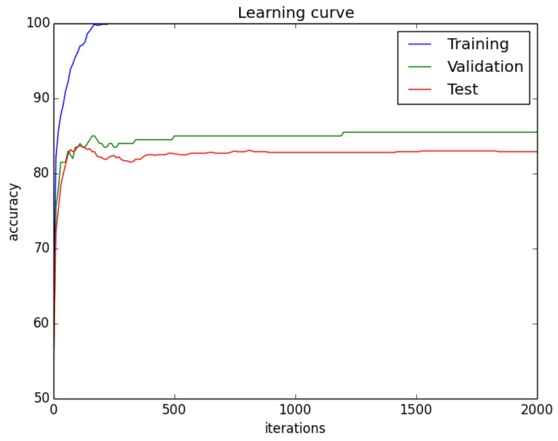


Figure 5: weights

Problem 6

1. The contour plot is given in Fig 6
2. The trajectory for "vanilla" gradient descent is the yellow line in the plot
3. The trajectory for momentum gradient descent is the green line in the plot
4. The "vanilla" gradient descent has more fluctuations due to the large learning rate. However, with momentum pointing to the constant optimum direction, the trajectory for momentum has less fluctuations. Also, to achieve the same learning effect (how close to the optimum), momentum gradient descent needs less steps.
5. The points were selected to be around the middle of the image. This is because that is where the weights matter the most. If the points were selected to be at the edge of the image, the weights won't matter because the edge does not contain anything that determines what the digit is.

   To produce a good visualization, the learning rate for both gradient descent must be tuned to be higher than normal, and the learning rate for "vanilla" gradient descent must be much higher to produce the fluctuation.

   A set of weights that won't produce a good visualization is chosen to be at the corner of the graph. The trajectories are not moving at all no matter how we change the initial values.

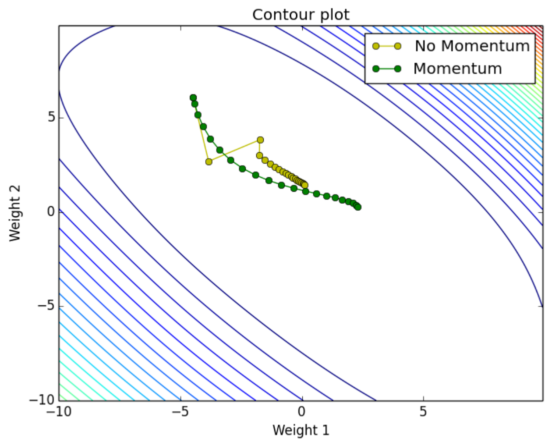


Figure 6: part (a-c)

Problem 7

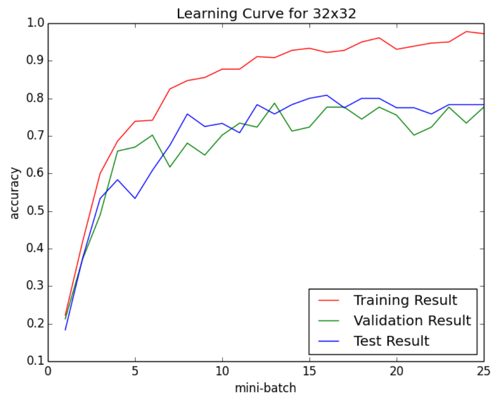
For computing the gradient with respect to each weight individually, assume N layers containing the layer of inputs, so that there are total of $N-1$ layers of weights, each contains $K \times K$ weights. It is reasonable to assume that running time is somehow related with the total number of one-layer-down derivatives computed, where one-layer-down derivatives means that there is no need to apply chain-rule. With that assumption in mind, we consider the running time is the number of one-layer-sown derivatives computed. In the first layer of weight, which is between layer N and $N-1$, for each weight, we need to compute 2 OLDD(one-layer-down derivatives) , so total is $2K^2$ OLDD computed, in the second layer, there is $K^2 \times (2 \times K + 1)$ OLDD computed, in the third layer, there is $K^2 \times ((2 \times K)^2 + 1)$, etc... in last layer, there is $K^2 \times ((2 \times K)^{N-2} + 1)$, so total OLDD computed is $K^2 \times (2 + (2K+1) + \cdots + ((2K)^{N-2} + 1)) = NK^2 + K^2(2K \times \frac{1-(2K)^{N-2}}{1-2K})$ (By the formula of summation for geometric sequence), $\frac{1-(2K)^{N-2}}{1-2K}$ approximately equals to $\frac{(2K)^{N-2}}{2K}$ when K is not too small, so total OLDD computed is approximately equals to $NK^2 + 2^{N-2}K^N$, which is $O(K^N)$ when N and K are reasonably big.

For fully-vectorized Backpropagation algorithm, make the same assumption that N layers contains the layer of inputs, so that there are still total of $N-1$ layers of weights, also, we make the assumption that time cost for multiplying 2 $K \times K$ matrix is O(K^4), in the first layer of weights, there is one matrix multiplication we need to compute, takes O(K^4), one layer deeper it goes, one more matrix multiplication it makes, since there are total $N-1$ layers of weights, so computation of gradient with respect to weight for the first layer of weights takes $O((N-1)K^4)$, and since we can caching intermediate results in fully-vectorized Backpropagation algorithm, once we find the time cost for first layer, we can claim that total time cost can't exceed time cost for first layer too much since for upper layers we can use cached results to make computation simpler, so total time cost is $O((N-1)K^4)$.

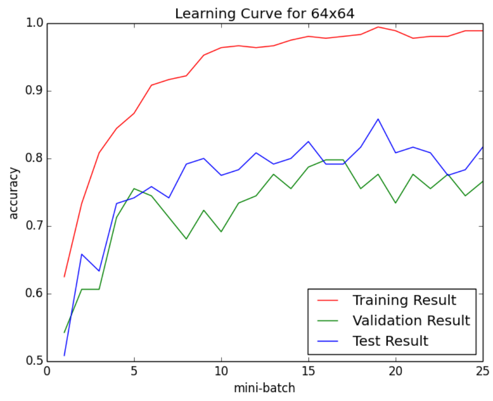
So conclusion is that fully-vectorized Backpropagation algorithm is much faster if N and K are reasonably big, since time cost for it is linearity between N and K rather than exponential.

Problem 8

The final test result for resolution of 32 × 32 has an accuracy of 83.3% and for resolution of 64 × 64 has an accuracy of 80%. The learning curves of the two resolutions are shown in Fig 7.
The results are trained by a fully connected, three-layered (input, hidden, output) network. The weights and bias are randomized. The activation function for input and output layer is linear, and for the hidden layer is ReLU. The network uses Adam as its optimizer. The hyper-parameters are picked through repeated experiments. The final hyper-parameters are: alpha = 0.001, number of hidden neurons = 36, epoch = 5, number of mini-batch in each epoch = 5, and the number of iterations = 1000



(a) 32 × 32



(b) 64 × 64

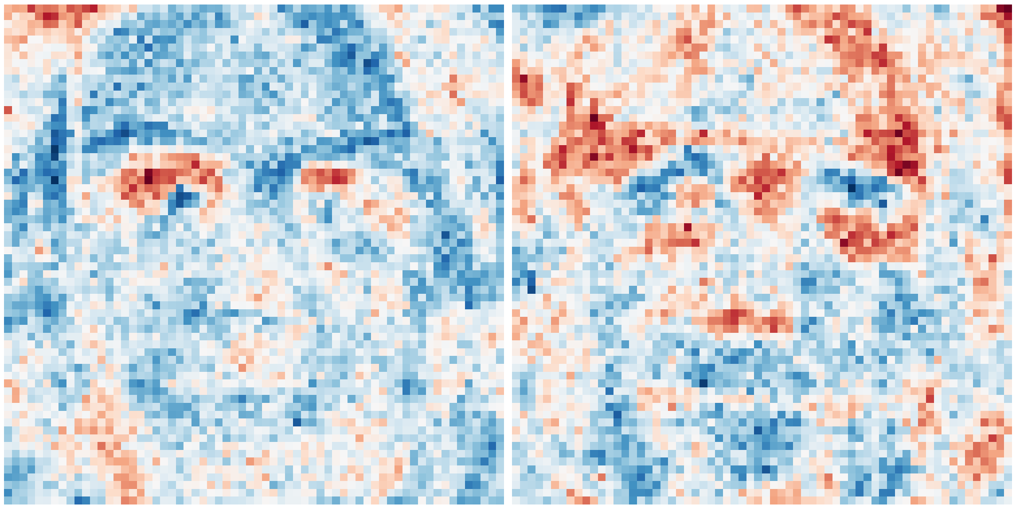
Figure 7: learning curves

Problem 9

The actors are chosen to be Harmon and Hader. Their hidden units are visualized as in Fig 8. The weight units are chosen using the following snippet of code:

Listing 4: Code for choosing featuring weight unit

```
x = Variable(torch.from_numpy(W),
        requires_grad=False).type(dtype_float)
y = model(x).data.numpy()
for k in range(y.shape[0]):
print k, actor_names[np.argmax(y[k, :])]
```



(a) Actor1: Harmon (b) Actor2: Hader

Figure 8: Weight Visualizations

Problem 10

The learning curve is shown in Fig 9.
To extract the last conv2d layer, the last ReLU and MaxPool2d layers are removed. The training : validation : test set ratio is same as in part 8. The network uses Adam as its optimizer. There are 600 hidden units and the learning rate is 0.0001 with 150 iterations.
The final performance of the network after training is as follows:

Training: 98.6754966887%
Validation: 95.0%
Testing: 93.3333333333%

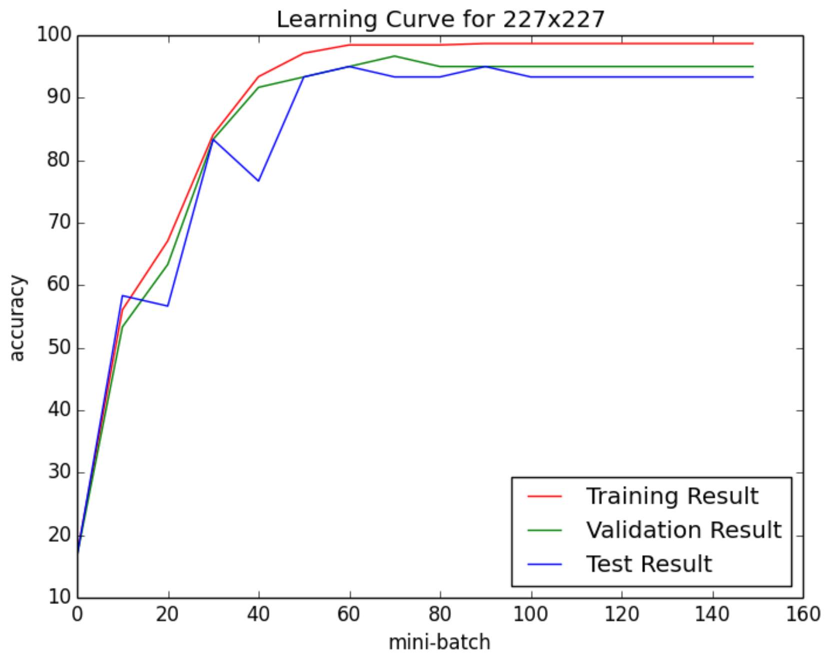


Figure 9: Learning curve