Machine Problem 1 Report
Yihui He*

I. Implementation

KEY points of implementing each part of neural network are illustrated as follow. All matrix multiplications are done via np.dot(). Element-wise matrix operation is done via “*”.

A. Forward pass and Loss

In Forward pass np.maximum between z and 0, is used to represent ReLU activation function.

In Softmax, first raise scores to the power of e element-wisely, then divide each element by row sum, finally we get $a^{(3)}$.

When computing Loss, pick up each true label element in each row is tricky: $a3[range(len(a3)),y]$.

Part of Forward pass is shown below. Other part of codes can be found in source files.

```python
a2=np.maximum(X.dot(W1)+b1,0)
scores=a2.dot(W2)+b2
a3=exp_scores
    / (np.sum(exp_scores,1))[None,:]
```

B. Backward pass and Gradient check

Firstly for each input, we need to compute $\delta_j^{(3)}$ for each output unit $j$:

$$
\delta_j^{(3)} = \begin{cases} 
\frac{1}{N}p(Y = j \mid X = x_i) - \frac{1}{N} & j = y_i \\
\frac{1}{N}p(Y = j \mid X = x_i) & j \neq y_i 
\end{cases}
$$

$$
\delta_3=a3
\delta_3[range(len(a3)),y]=\delta_2[a3[range(len(a3)),y]=\delta_3
\delta_3/=a3
\delta_2=a2.T.dot(delta_3)+reg*W2
\delta_1=exp_scores
    / (np.sum(exp_scores,1))[None,:]
$$

Secondly, in the front hidden layer, for each input, we need to compute $\delta_j^{(3)}$ for each hidden node $j$:

$$
\delta_j^{(2)} = (w_j^{(2)})^T \delta_j^{(3)} \circ f'(z_j^{(2)})
$$

Note that, the second operator is Hadamard product.

To avoid divide by zero in gradient check, I made a small modification to the formula:

$$
\frac{|A - B|}{\max(10^{-8}, |A + B|)} \leq \delta
$$

max errors among inputs are as follow:

<table>
<thead>
<tr>
<th></th>
<th>w_1</th>
<th>w_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.56e-09</td>
<td>3.44e-09</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b_1</th>
<th>b_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.74e-09</td>
<td>4.45e-11</td>
</tr>
</tbody>
</table>

C. Train and Predict

To perform a minibatching, np.random.choice can be used, and set replace=True to avoid same inputs being used. Then update each hyperparameter using SGD.

```python
rand_idx=np.random.choice(num_train,batch_size,replace=False)
X_batch=X[rand_idx]
y_batch=y[rand_idx]
for var in self.params:
    self.params[var]-=learning_rate*grads[var]
```

As for predicting, run forward propagation, and use np.argmax(,1) to find predicted $y$ for each input.

```python
y_pred=np.argmax(np.maximum(0,
             (X.dot(self.params["W1"])\n             +self.params["b1"])),
             1)
```

II. MODEL BUILDING

Basically there are two ways to tune a neural net: grid search and random search. For simplicity, I employ grid search to tune 3 hyperparameters: number of neurons in the hidden layer, regularization strength, and learning rate. My tuning procedure is twofold. First, run a coarse-grained search. Second, based on the result of first step, run a fine-grained search around top results.

*Exchange student, 2nd year CS undergrad from Xi’an Jiaotong University, eli.he@foxmail.com
For coarse-grained search, Number of neurons range from 50 to 550, step 50. Regularization strength and learning rate are selected from geometrical sequences. Regularization strength range from $0.5 \times 10^{-3}$ to $0.5 \times 10^{2}$, with ratio 10. Learning rate range from $1 \times 10^{-5}$ to $1 \times 10^{-1}$, with ratio 10. During this, top results have about 45% validation accuracy. Some Top results are shown below:

<table>
<thead>
<tr>
<th>hidden neurons</th>
<th>learning rate</th>
<th>regularization strength</th>
<th>validation accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>350</td>
<td>0.001</td>
<td>0.05</td>
<td>0.516</td>
</tr>
<tr>
<td>400</td>
<td>0.001</td>
<td>0.0005</td>
<td>0.509</td>
</tr>
<tr>
<td>250</td>
<td>0.001</td>
<td>0.0005</td>
<td>0.505</td>
</tr>
<tr>
<td>250</td>
<td>0.001</td>
<td>0.05</td>
<td>0.501</td>
</tr>
<tr>
<td>150</td>
<td>0.001</td>
<td>0.005</td>
<td>0.5</td>
</tr>
<tr>
<td>500</td>
<td>0.001</td>
<td>0.05</td>
<td>0.5</td>
</tr>
</tbody>
</table>

For fine-grained search, I picked up one of the above top results. And search around it’s hyperparameters. It can reach a validation accuracy of 50%. After found a suitable set of hyperparameters, I start tuning numbers of iterations, and batch size. Finally, our original neural network reach a validation accuracy of 52%.

### III. Extra Credits

I put results in the next section, in order to compare our original two-layers neural network, with other enhanced neural networks.

#### A. momentum and other update methods

Implementation of momentum needs to change hyperparameters update code in train.

```python
v_prev = cache[param]
cache[param] = arg*cache[param] - learning_rate*grads[param]
self.params[param] += arg*v_prev + (1+arg)*cache[param]
```

RMSprop, which is a per-parameter adaptive learning rate method.

```python
cache[param] = arg*cache[param] 
+(1-arg)*np.power(grads[param],2)
sel.f.params[param] -= learning_rate*grads[param]/np.sqrt(cache[param]+1e-8)
```

It turns out that, Momentum, Nesterov momentum and RMSprop all have a better converge rate than SGD. However, difference between these three update methods is ambiguous. Performances of these methods are compared in table III

<table>
<thead>
<tr>
<th>DIFFERENCES BETWEEN UPDATE METHODS</th>
</tr>
</thead>
<tbody>
<tr>
<td>accuracy</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>SGD</td>
</tr>
<tr>
<td>Momentum</td>
</tr>
<tr>
<td>Nesterov</td>
</tr>
<tr>
<td>RMSprop</td>
</tr>
</tbody>
</table>

#### B. Dropout

Because we only have one layer of hidden nodes, dropout only needs to be performed once per batch. In code, it only needs minor change to Forward pass.

```python
a2*=(np.random.randn(*a2.shape)<p)/p
```

Dropout is said to be a easy way to prevent over-fitting, and it is also an ensemble of multi models which should enhance performance. So I test it with more hidden neurons(500 hidden neurons). Dropout rate is set to be 30%, 50% and 70% empirically. It turns out that the results from these three dropout rates do not differ a lot, with test accuracy ranging from 54% to 56%. And it also shows that, with out L2 regularization, dropout is able to constrain train accuracy not too higher than validation accuracy, which is 73%.

#### C. Initialization method

I tried 3 different ways of initialization: $N(0,1)\sqrt{1/n}$, $N(0,1)\sqrt{2/(n_{in}+n_{out})}$, $N(0,1)\sqrt{2/n}$. Comparing with our original initialization: $10^{-4}N(0,1)$,
some outperform it. The neural network I’m testing on have 500 hidden neurons. I trained it 10000 iterations, 100 batch size, with dropout and momentum.

In our case, we have 3072 input neurons, so \(N(0, 1)\sqrt{1/n}\) actually is \(3.3 \times 10^{-4}\), whose weights are 3 times bigger than our original initialization. It is terrible at first 100 iterations. It reaches loss of 33.161757.

We only 10 output neurons. So \(N(0, 1)\sqrt{2/(n_{in} + n_{out})}\) and \(N(0, 1)\sqrt{2/n}\) do not differ so much between each other. Although, these two initialization methods’ weights are 1.5 times bigger than our original initialization. Loss changing is almost the same as our original initialization. But maybe these methods are better, they are related to the input and output rather than a hand setting value.

**D. Activation functions**

I’ve tried 2 activation functions except ReLU: *leaky ReLU* and *tanh*.

Implementation of leaky ReLU needs to change a few lines of forward pass and backpropagation. \(\alpha\) is set empirically.

```python
# forward pass
a2=np.maximum(inp,.01*inp)
# backpropagation
dF=ap.ones(np.shape(a2))
dF[a2<0.0]=0.01
```

As is mentioned in research, leaky ReLU may lead to overfitting sometimes. In my test, with the same other hyperparameters, training accuracy of leaky ReLU improved by 4%, however, validation accuracy seems not improved.

As for tanh, with the same other hyperparameters, training time becomes longer, and validation accuracy dropped 2%.

**E. Preprocessing**

I employ PCA whitening and K-means to do data preprocessing. It turns out that these two processes hugely improve test accuracy. Without special tuning, test accuracy could easily reach 65%. By intuition, PCA whitening reduce effects of bright and contrast, and K-means put images into different classes before we truly begin training.

I use np.linalg to implement PCA whitening as follow:

```python
[D,V]=np.linalg.eig(
    np.cov(img,rowvar=0))
P = V.dot(
    np.diag(np.sqrt(1/(D + 0.1))))
    .dot(V.T)
img = img.dot(P)
```

Intuitively, Kmeans clusters different images into different clusters, which is much easier for our neural net to handle. I select 1600 centroids empirically from previous work. Also I encountered *MemoryError* when implementing Kmeans, because our data can not be clustered at once. So I changed the processing way a little bit:

```python
for each iterations:
    for each batch in data:
        cluster batch
cumulate result
renew centroids
```
Finally, with these PCA whitening and K-means, my best test accuracy successfully reaches 74%. The following two graphs shows loss history and training, validation accuracy of the selected best hyperparameters. More details of hyperparameters is shown in the next section.

Due to our time limitation, I choose to use 200 hidden nodes. I guess it will work better with more hidden nodes, and longer training time.

IV. RESULTS

As mentioned in II Model Building, I employed a grid search to find best hyperparameters. In order to get better result, I tuned number of iterations, finally fix it to 12000, with a batch size of 100. It increases the Validation accuracy 5%.

The following table shows my three methods accuracy:

<table>
<thead>
<tr>
<th>TABLE III</th>
<th>DIFFERENCES BETWEEN UPDATE METHODS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>Dropout</td>
</tr>
<tr>
<td>hidden nodes</td>
<td>350</td>
</tr>
<tr>
<td>learning rate</td>
<td>$1 \times 10^{-3}$</td>
</tr>
<tr>
<td>learning rate Decay</td>
<td>.95</td>
</tr>
<tr>
<td>regularization</td>
<td>L2,0.05</td>
</tr>
<tr>
<td>Activation</td>
<td>ReLU</td>
</tr>
<tr>
<td>Update method</td>
<td>SGD</td>
</tr>
<tr>
<td>Iterations</td>
<td>$1 \times 10^4$</td>
</tr>
<tr>
<td>Batch size</td>
<td>100</td>
</tr>
<tr>
<td>Time(min)</td>
<td>15</td>
</tr>
<tr>
<td>Train accuracy</td>
<td>60%</td>
</tr>
<tr>
<td>Validation</td>
<td>55%</td>
</tr>
<tr>
<td>Test</td>
<td>51.6%</td>
</tr>
</tbody>
</table>

V. CHALLENGES

I encountered many challenges through this machine problem.

When I’m doing gradient check, I spent great amount of time finding bugs in my code, but still not able to figure out what’s wrong with my code. Then, instead of directly comparing gradient errors, I cut gradient check procedure into pieces. Check derivatives sequentially: $\frac{\partial H}{\partial a}(3)$, $\delta(3)$, $\frac{\partial H}{\partial w}(2)$, $\frac{\partial H}{\partial b}(2)$, $\delta(2)$...

Tuning hyperparameters is really time consuming, I run my program on both on my laptop and CSIL lab. And create multi threads to speed up. I save training informations like hyperparameters and accuracy to CSV files. View them when the program finish. It saves me a lot of time and avoids hand tuning.

Some parts of my code encountered MemoryError, because operation like matrix multiplications. I tried iteration way and batch way of matrix multiplications. Both solve MemoryError, but batch way turns out to be much faster.

VI. POSSIBLE IMPROVEMENTS

There are some other update methods (Adam, Adagrad, etc) I haven’t tried. Maybe they are better than Momentum and RMSprop.

More complexed activation function like PReLU, maybe work better than ReLU and Leaky ReLU.

L1 regularization in some situation, maybe better than L2.

As for preprocessing, applying kernel trick may improve results.

With longer training time, it is possible to reach higher accuracy.