# CSC 567 Homework 3

Qilin Ye

December 9, 2022

## 1. Multiclass Perceptron

*Solution for (1.1).* Recall

$$F_i(w_1, ..., w_C) = \max\left\{0, \max_{y\neq y_i}(w_y^T x_i - w_{y_i}^T x_i)\right\}.$$

This means $\dfrac{\partial F_i}{\partial w_c} = 0$ (most of the time), $x_i$ (from $w_y^T x_i$), or $-x_i$ (from $-w_{y_i}^T x_i$ only).

More specifically, we let $k = \operatorname{argmax}_{y\neq y_i}(w_i^T x_i - w_{y_i}^T x_i)$.

- If $w_k^T x_i - w_{y_i}^T x_i > 0$, then $\dfrac{\partial F_i}{\partial w_c} = x_i$ if $c = k$, $-x_i$ if $c = y_i$, and $0$ otherwise.
- If $w_k^T x_i - w_{y_i}^T x_i \leqslant 0$, then the derivative is simply $0$.

*Solution for (1.2).* In each iteration, we randomly pick $i \in [n]$, and consider the data point $(x_i, y_i)$. Let $k = \operatorname{argmax}_{c\in[1,n]}(w_c^T x_i)$. If the current label for $x_i$ is indeed $y_i$, no need to change; this corresponds to $w_{y_i}^T x_i > w_k^T x_i$ for all $k \neq y_i$. Otherwise, our current model mistakenly labels $x_i$ as $k$ since $w_k^T x_i > w_{y_i}^T x_i$ , and we need to fix this. From the computation above, we need to decrease the score of the wrong label $w_k$ by $w_k \leftarrow w_k - x_i$ as well as increase the score of the right label $w_{y_i}$ by $w_{y_i} \leftarrow w_{y_i} + x_i$. Pseudocode:

**Algorithm 1:** Multiclass Perceptron

```
1 Input: A training set $(x_1, y_1), \dots, (x_n, y_n)$
2 Initialization: $w_1 = \cdots = w_C = 0$
3 Repeat:
4     Randomly select $i \in [C]$
5     Compute $k = \operatorname{argmax}_{y\in[C]} w_y^T x_i$
6     if $k \neq y_i$ then
7         $w_k \leftarrow w_k - x_i$
8         $w_{y_i} \leftarrow w_{y_i} + x_i$
```

*Solution for (1.3).* Here we use exactly the same idea. If $w_c = \sum_{i=1}^n a_{c,i} x_i$, then updating $w_c$ by $w_c \leftarrow w_c - x_i = \sum_{i=1}^n a_{c,i} x_i - x_i$ is equivalent to $a_{c,i} \leftarrow a_{c,i} - 1$, and similarly $+1$ for $+x_i$. Replacing $w_c^T x_i$ by $\sum_{i=1}^n a_{c,j} x_j^T x_i =$

$\sum_{i=1}^{n} a_{c,j} k(x_j, x_i)$, we obtain the following pseudocode.

**Algorithm 2:** Multiclass Perceptron with kernal

```
1 Input: A training set $(x_1, y_1), \ldots, (x_n, y_n)$
2 Initialization: $a_{c,i} = 0$ for $c \in [C], i \in [n]$
3 Repeat:
4     Randomly select $i \in [C]$
5     Compute $k = \text{argmax}_{y \in [C]} \sum_{j=1}^{n} a_{y,j} k(x_i, x_j)$
6     if $k \neq y_i$ then
7         $a_{k,i} \leftarrow a_{k,i} - 1$
8         $a_{y_i,i} \leftarrow a_{y_i,i} + 1$
```

## 2. Packprop for mini CNN

*Solution for (2.1).* This is just computation:

$$\frac{\partial \ell}{\partial v_1} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v_1} = \frac{-ye^{-y\hat{y}}}{1 + e^{-y\hat{y}}} \cdot o_1 = -\frac{yo_1}{e^{y\hat{y}} + 1} = -yo_1\sigma(-y\hat{y})$$

where $\sigma$ denotes the sigmoid function. Similarly, $\dfrac{\partial \ell}{\partial v_2} = -yo_2\sigma(-y\hat{y})$.

*Solution (2.2).* More computation:

$$\begin{aligned}\frac{\partial \ell}{\partial w_1} &= \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_1} \frac{\mathrm{d}o_1}{\mathrm{d}a_1} \frac{\partial a_1}{\partial w_1} + \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial o_2} \frac{\mathrm{d}o_2}{\mathrm{d}a_2} \frac{\partial a_2}{\partial w_1} \\ &= -y\sigma(-y\hat{y}) \cdot v_1 \cdot H(a_1) \cdot x_1 - y\sigma(-y\hat{y}) \cdot v_2 \cdot H(a_2) \cdot x_2 \\ &= -y\sigma(-y\hat{y})[v_1 H(a_1) x_1 + v_2 H(a_2) x_2].\end{aligned}$$

Similarly,

$$\frac{\partial \ell}{\partial w_2} = -y\sigma(-y\hat{y})[v_1 H(a_1) x_2 + v_2 H(a_2) x_3].$$

*Solution for (2.3).* Talk is cheap.

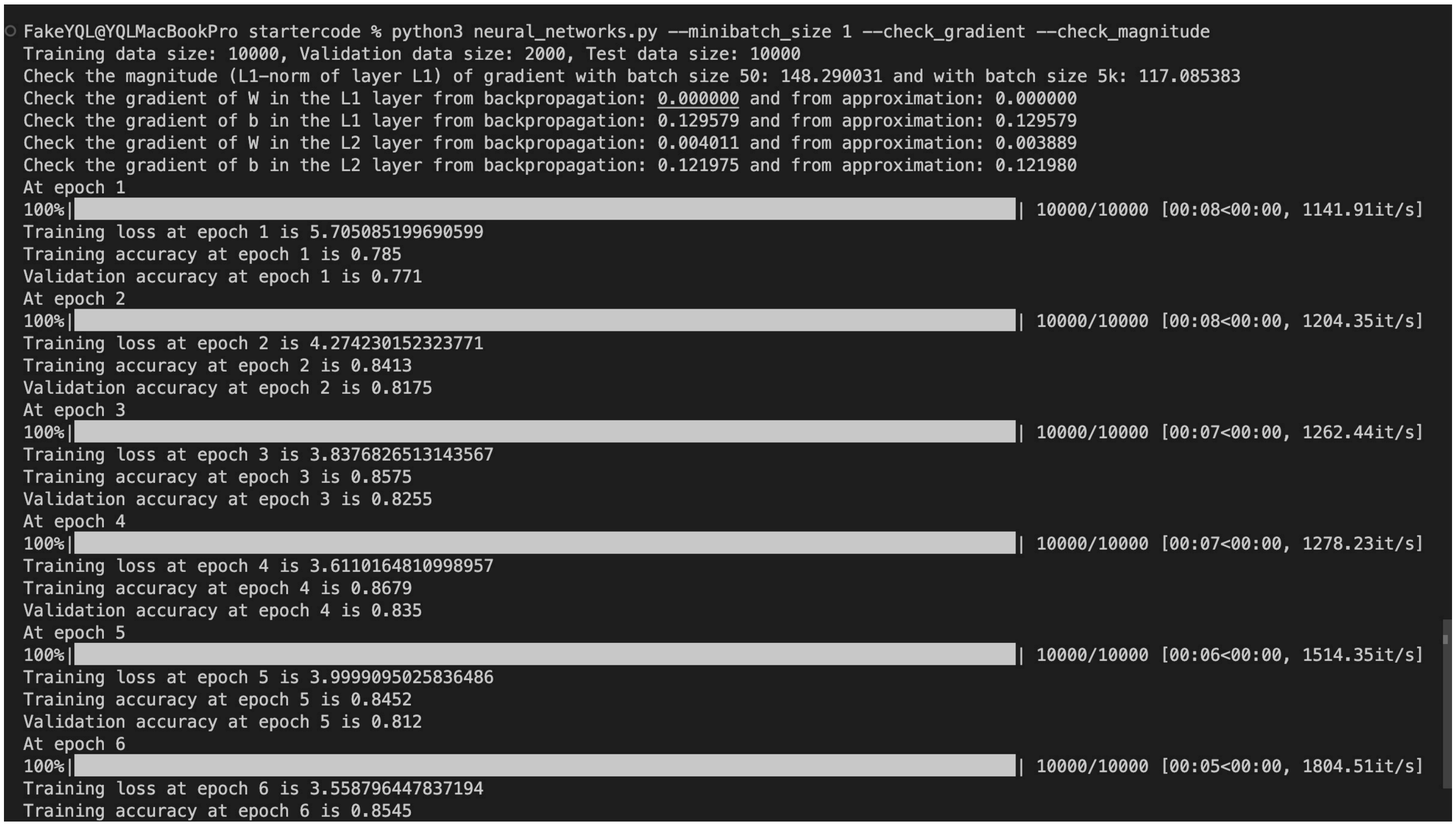
**Algorithm 3:** Backprop on mini CNN

```
1 Input: A training set $(x_1, y_1), \ldots, (x_n, y_n)$, and learning rate $\eta$
2 Initialization: set $w_1, w_2, v_1, v_2$ randomly
3 Repeat:
4     Randomly select $i \in [n]$
5     Forward propagation:
6         $a_1 \leftarrow x_1 w_1 + x_2 w_2$, $a_2 \leftarrow x_2 w_1 + x_3 w_2$
7         $o_1 \leftarrow \max\{0, a_1\}$, $o_2 \leftarrow \max\{0, a_2\}$
8         $\hat{y} \leftarrow o_1 v_1 + o_2 v_2$
9     Backward propagation:
10        $v_1 \leftarrow v_1 + \eta \cdot yo_1\sigma(-y\hat{y})$
11        $v_2 \leftarrow v_2 + \eta \cdot yo_2\sigma(-y\hat{y})$
12        $w_1 \leftarrow w_1 + \eta \cdot y\sigma(-y\hat{y})[v_1 H(a_1) x_1 + v_2 H(a_2) x_2]$
13        $w_2 \leftarrow w_2 + \eta \cdot y\sigma(-y\hat{y})[v_1 H(a_1) x_2 + v_2 H(a_2) x_3]$
```

## 3. Fashions MNIST

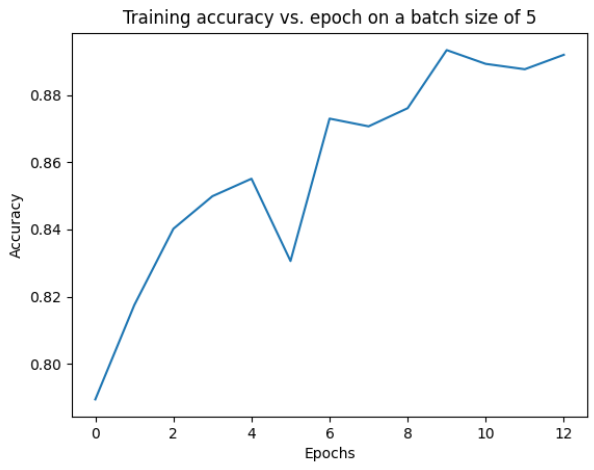
### 3.1.5: Sample output:

```
Check the gradient of W in the L1 layer from backpropagation: 0.000000 and from approximation: 0.000000
Check the gradient of b in the L1 layer from backpropagation: 0.129579 and from approximation: 0.129579
Check the gradient of W in the L2 layer from backpropagation: 0.004011 and from approximation: 0.003889
Check the gradient of b in the L2 layer from backpropagation: 0.121975 and from approximation: 0.121980
```

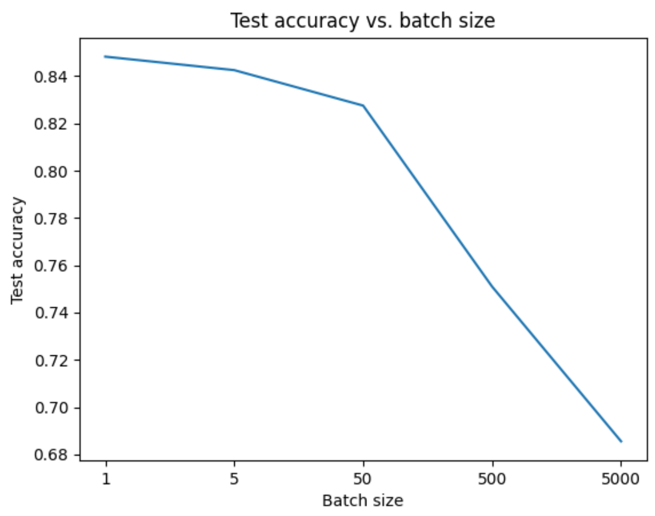


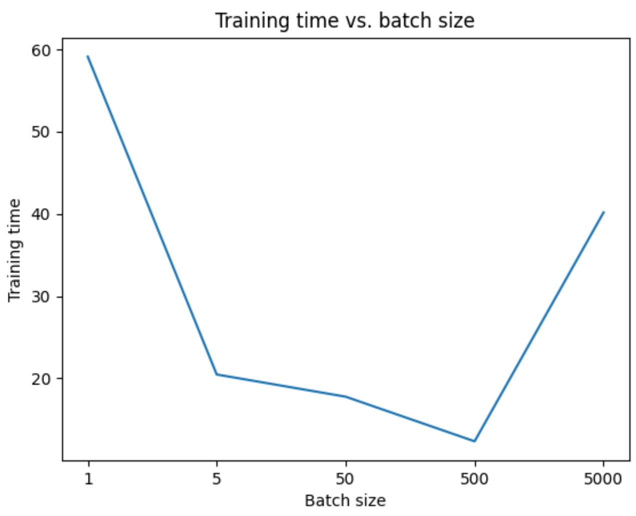
```
FakeYQL@YQLMacBookPro startercode % python3 neural_networks.py --minibatch_size 1 --check_gradient --check_magnitude
Training data size: 10000, Validation data size: 2000, Test data size: 10000
Check the magnitude (L1-norm of layer L1) of gradient with batch size 50: 148.290031 and with batch size 5k: 117.085383
Check the gradient of W in the L1 layer from backpropagation: 0.000000 and from approximation: 0.000000
Check the gradient of b in the L1 layer from backpropagation: 0.129579 and from approximation: 0.129579
Check the gradient of W in the L2 layer from backpropagation: 0.004011 and from approximation: 0.003889
Check the gradient of b in the L2 layer from backpropagation: 0.121975 and from approximation: 0.121980
At epoch 1
100%|████████████████████████████████████████| 10000/10000 [00:08<00:00, 1141.91it/s]
Training loss at epoch 1 is 5.705085199690599
Training accuracy at epoch 1 is 0.785
Validation accuracy at epoch 1 is 0.771
At epoch 2
100%|████████████████████████████████████████| 10000/10000 [00:08<00:00, 1204.35it/s]
Training loss at epoch 2 is 4.274230152323771
Training accuracy at epoch 2 is 0.8413
Validation accuracy at epoch 2 is 0.8175
At epoch 3
100%|████████████████████████████████████████| 10000/10000 [00:07<00:00, 1262.44it/s]
Training loss at epoch 3 is 3.8376826513143567
Training accuracy at epoch 3 is 0.8575
Validation accuracy at epoch 3 is 0.8255
At epoch 4
100%|████████████████████████████████████████| 10000/10000 [00:07<00:00, 1278.23it/s]
Training loss at epoch 4 is 3.6110164810998957
Training accuracy at epoch 4 is 0.8679
Validation accuracy at epoch 4 is 0.835
At epoch 5
100%|████████████████████████████████████████| 10000/10000 [00:06<00:00, 1514.35it/s]
Training loss at epoch 5 is 3.9999095025836486
Training accuracy at epoch 5 is 0.8452
Validation accuracy at epoch 5 is 0.812
At epoch 6
100%|████████████████████████████████████████| 10000/10000 [00:05<00:00, 1804.51it/s]
Training loss at epoch 6 is 3.558796447837194
Training accuracy at epoch 6 is 0.8545
```

### 3.1.6:



### 3.2.1:





**3.2.2**:

| `batch_size` | `best_epoch` | number of gradient updates |
|---|---|---|
| 1 | 13 | $10000 \cdot 13 = 130000$ |
| 5 | 10 | $2000 \cdot 10 = 20000$ |
| 50 | 43 | $200 \cdot 43 = 8600$ |
| 500 | 39 | $20 \cdot 39 = 780$ |
| 5000 | 158 | $2 \cdot 158 = 316$ |

**3.2.3**:

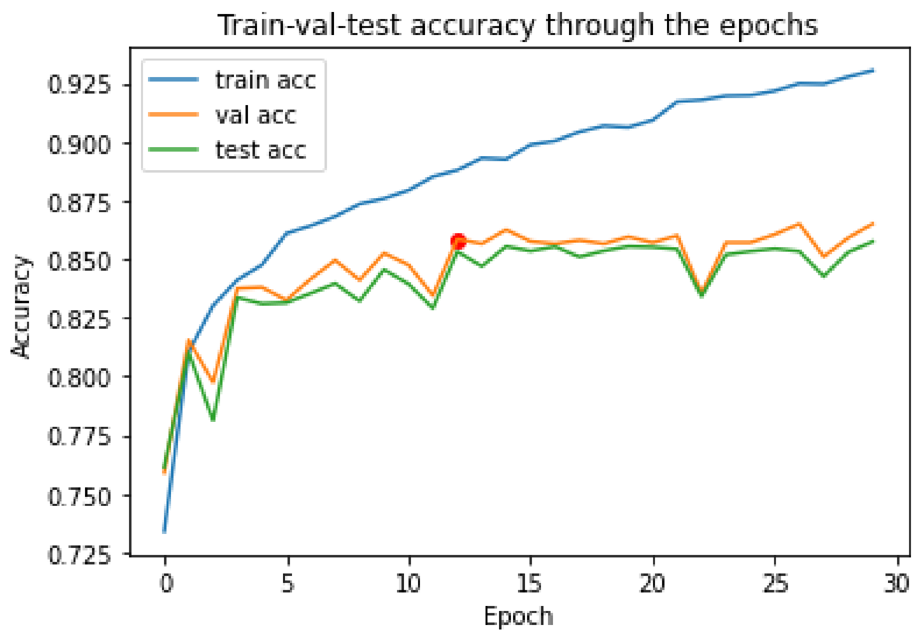
- No. When batch size gets too small, training time starts to increase again. This is because smaller batch size actually corresponds to significantly more gradient updates.

- No. In fact our data shows the opposite. One possible explanation is due to the nature of SGD. More specifically, with a smaller batch size, our algorithm is able to more frequently "correct" itself on the way it descents. When a large batch size does one "big" update a direction, it is not able to change direction mid-way through updating, even if the direction picked is far from being optimal. Meanwile, a smaller batch size, on the other hand, is able to do multiple updates, where each update makes the next one "more correct." Even if the first update picks a "bad" direction, the later ones can try to correct it and pick better directions.

  Another potential reason is that smaller batch size are more likely to escape saddle points. Consider the extreme case of batch size $1$ (in which case we have SGD) vs. batch size equal to data set's cardinality (in which case we have GD). Our lectures indeed mentioned that GD's are more likely to be trapped by saddle points than SGD.

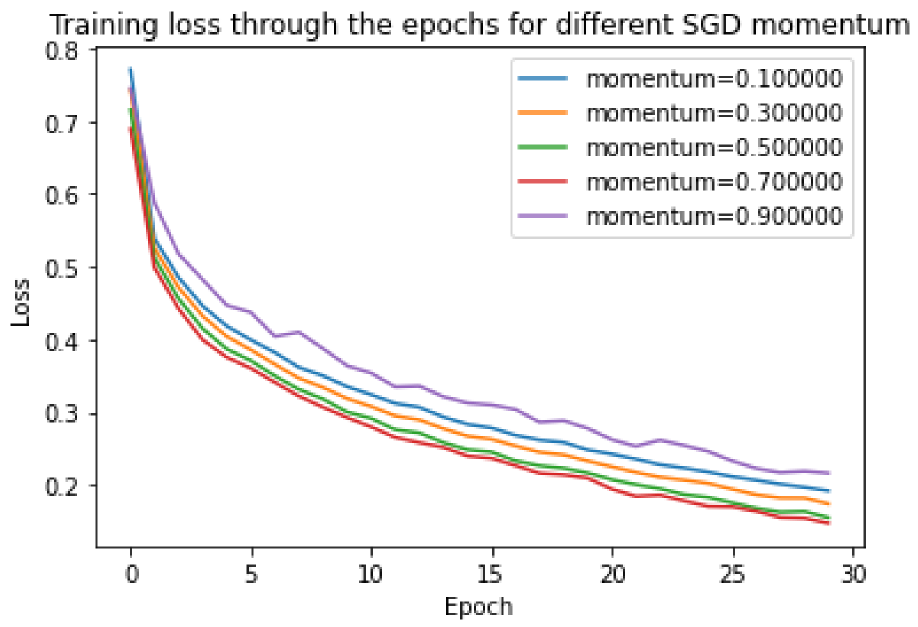
- Yes, assuming we are satisfied with the different resulting test accuracy. This is because with a larger batch size, each stochastic gradient is less noisy.

**3.3**:



- Test/validation accuracy starts to fluctuate, i.e., without significant improvement, after the early-stopping point, but the training accuracy continues to increase.
- If patience is too small, we may mistakenly classify early epoch as a stopping point, whereas in fact the reason validation accuracy fails to increase is simply due to noise/bad luck. For example, setting patience = $2$ we may have stopped near epoch $5$, whereas from the graph we see that stopping at epoch ~ $15$ actually gives a better overall validation/test accuracy.

**3.4**:



Clearly, among the five choices given, $\alpha = 0.7$ is the most optimal. That is, we initialize $v = 0$ and iteratively define $v \leftarrow 0.7v + \nabla F(w)$ and let $w \leftarrow w - \eta v$.

**3.5**:

(1) No, the test accuracy significantly decreases:

```
Test accuracy: 0.47369998693466187
```

(The test accuracy on original images was 0.8467.)

(2) Number of parameters in original MLP: $(28^2 + 1) \cdot 128 + (128 + 1) \cdot 10 = 101770$, where $28^2 \cdot 128$ is the size of $W^{(1)}$, $1 \cdot 128$ for $b^{(1)}$, $128 \cdot 10$ for $W^{(2)}$, and $1 \cdot 10$ for $b^{(2)}$. On the other hand, the 2-layer CNN has $(7^2 + 1) \cdot 64 + 11^2 \cdot 64 \cdot 10 + 10 = 80650$ parameters, where $(7^2 + 1) \cdot 64$ are the $64$ kernels of size $7 \times 7$ (with one parameter for bias), $11$ the size after max-pooling, $11^2 \cdot 64 \cdot 10$ the number of parameters for the fully connected layer, and $10$ for biases.

(3)

```
313/313 - 2s - loss: 0.3600 - accuracy: 0.8786 - 2s/epoch - 8ms/step
In-domain test accuracy: 0.878600001335144
313/313 - 2s - loss: 1.9710 - accuracy: 0.5480 - 2s/epoch - 7ms/step
Out-of-domain test accuracy: 0.5479999780654907
```

We see that indeed the OOD test accuracy increases slightly, whereas the in-domain test accuracy roughly stays the same. The increase in OOD test accuracy is possibly due to the fact that our convolution kenrel helped to offset the effects of translation of images as it keeps the spatial dependencies of the input images.

(4) This time, we have a total of $(7^2 + 1) \cdot 64 + 64 \cdot 128 \cdot 2^2 + 128 + 5^2 \cdot 128 \cdot 10 + 10 = 68106$ parameters. As before, $(7^2+1) \cdot 64$ corresponds to the first layer of $64$ kernels of size $7 \times 7$, and each layer has one bias parameter. Then $64 \cdot 128 \cdot 2^2 + 128$ corresponds to the $64 \cdot 128$ layers of $2 \times 2$ kernels, along with $128$ parameters for biases. Finally, we note that after the first convolution layer, the size shrinks to $22 \times 22$; after a max-pooling this becomes $11 \times 11$; after another $2 \times 2$ convolution this becomes $10 \times 10$; and finally, after a second max-pooling, $5 \times 5$. A total of $128 \cdot 10$ layers, so another $5^2 \cdot 128 \cdot 10$ parameters, plus $10$ for biases.

(5)

```
313/313 - 4s - loss: 0.4211 - accuracy: 0.8514 - 4s/epoch - 12ms/step
In-domain test accuracy: 0.8514000177383423
313/313 - 4s - loss: 1.4677 - accuracy: 0.5892 - 4s/epoch - 12ms/step
Out-of-domain test accuracy: 0.5892000198364258
```

With the help of another layer, our model is able to apply more non-linearity before outputting the final classification result, and this indeed in a better OOD test accuracy.

(6) This time no model is still doing well.

```
Rotation 90 degrees:
Base model accuracy: 0.01889999955892563
CNN model accuracy: 0.05609999969601631
Deeper CNN model accuracy: 0.053599998354911804
Rotation 180 degrees:
Base model accuracy: 0.1956000030040741
CNN model accuracy: 0.20630000531673431
Deeper CNN model accuracy: 0.054499998688697815
Rotation 270 degrees:
Base model accuracy: 0.055799998342990875
CNN model accuracy: 0.053199999034404755
Deeper CNN model accuracy: 0.054499998688697815
```