HW1 Part Two: Code Implementation

[redacted]

Multilayer Perception (MLP)

FP and BP

Firstly define the functions `relu()`, `softmax()` and `cross_entropy()` for usage:

```
def relu(mat):
    return mat * (mat > 0)

def softmax(z):
    z_ = z - np.max(z, axis=1, keepdims=True)
    return np.exp(z_) / np.sum(np.exp(z_), axis=1, keepdims=True)

def cross_entropy(z, z_hat):
    return -np.sum(z * np.log(z_hat)) / z.shape[0]
```

In the default running codes, the input `x_batch` is an $m \times n$ matrix, in which m is the size of a mini-batch and n is the length of the data. Thus the FP and BP computations should be changed a little from the given computations in the homework file.

FP:

$$z_1 = xW^{(1)} + b^{(1)}$$

$$h_1 = \mathrm{ReLU}(z_1)$$

$$z_2 = h_1 W^{(2)} + b^{(2)}$$

$$\hat{y} = \mathrm{Softmax}(z_2)$$

BP:

$$\frac{\partial f_{CE}}{\partial W^{(2)}} = h_1^T(\hat{y} - y)$$

$$\frac{\partial f_{CE}}{\partial b^{(2)}} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)$$

$$\frac{\partial f_{CE}}{\partial W^{(1)}} = x^T((\hat{y} - y){W^{(2)}}^T \circ \text{sgn}(z_1))$$

$$\frac{\partial f_{CE}}{\partial b^{(1)}} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y} - y){W^{(2)}}^T \circ \text{sgn}(z_1)$$

Detail implementations:

```
# TODO
# forward pass
z1 = np.matmul(x, w1) + b1
h1 = relu(z1)
z2 = np.matmul(h1, w2) + b2

y_hat = softmax(z2)
loss = cross_entropy(y, y_hat)
```

```
# TODO
# backward pass
[db2, dw2, db1, dw1] = [np.sum(y_hat - y, axis=0) / y.shape[0],
                        np.matmul(h1.T, y_hat - y) / y.shape[0],
                        np.sum(np.matmul((y_hat - y), w2.T)* (np.sign(z1) * (z1
> 0)), axis=0) / y.shape[0],
                        np.matmul(x.T, np.matmul((y_hat - y), w2.T) *
(np.sign(z1) * (z1 > 0))) / y.shape[0]]
```

Run `check_grad()` and show the results:

```
(base) ➜  wuyulun@MacBookAir-wyl MLP python mlp.py
Hyper-parameters:
Namespace(hidden_dim=50, lr=0.001, batch_size=16, epochs=10)
Dataset information:
training set size: 10000
test set size: 5000
Gradient check of backward propagation:
Relative error of dw2 1.7778400294583852e-13
Relative error of db2 1.0710095771349852e-14
Relative error of dw1 7.153320954761542e-16
Relative error of db1 1.5553058010600368e-15
If you implement back propagation correctly, all these relative errors should be less than 1e-5.
```

The errors of the gradients are under estimation.

Random Initialization of Params

Use He Initialization, which means to initialize parameters in a Gaussian distribution with a $\frac{2}{n_{in}}$ var:

```
# TODO
#  randomly initialize the parameters (weights and biases)
[w1, b1, w2, b2] = [np.random.normal(loc=0, scale=np.sqrt(2 / input_dim), size=
(input_dim, args.hidden_dim)),
                    np.random.normal(loc=0, scale=np.sqrt(2 / input_dim),
size=args.hidden_dim),
                    np.random.normal(loc=0, scale=np.sqrt(2 / args.hidden_dim),
size=(args.hidden_dim, output_dim)),
                    np.random.normal(loc=0, scale=np.sqrt(2 / input_dim),
size=output_dim)]
```

Losses, Gradients and Param Updating

Directly use `calc_loss_and_grad()` function to get losses and gradients data (set 'eval_only' as False), use GD algorithm:

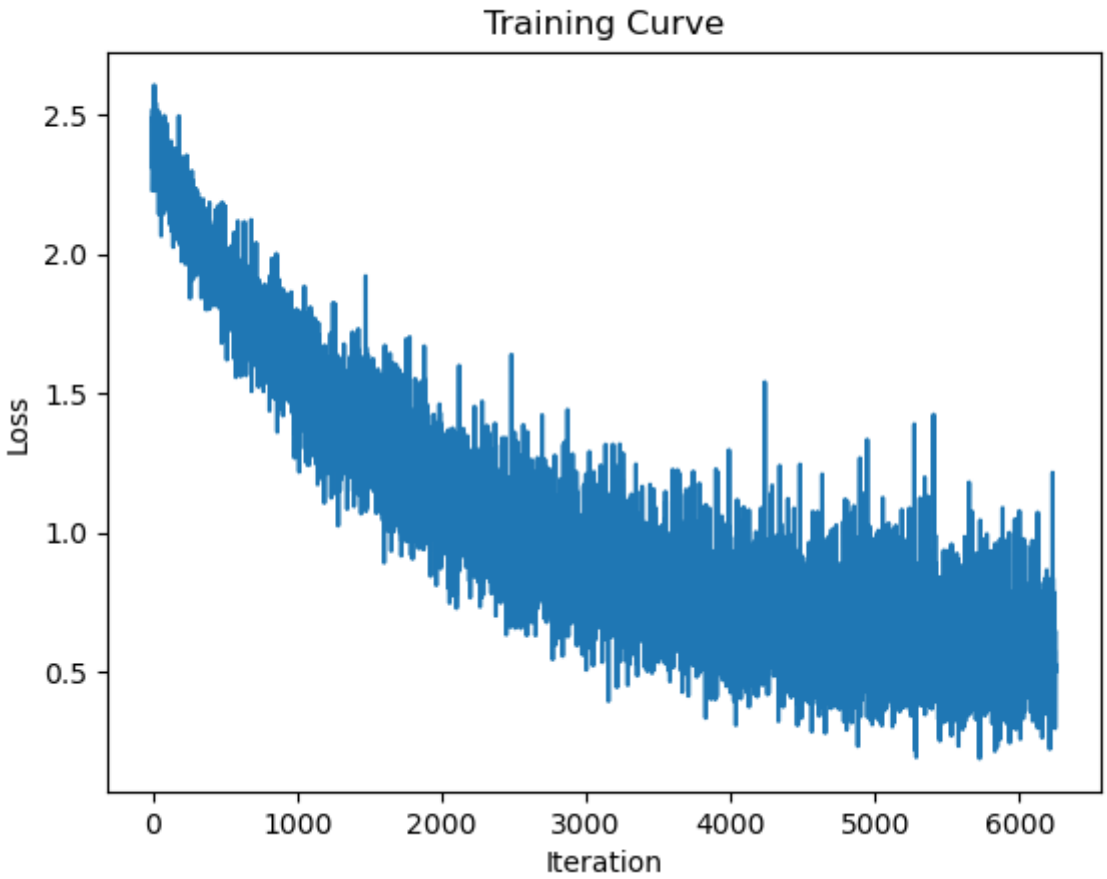
```
# TODO
# update parameters
[w1, b1, w2, b2] = [w1 - args.lr * dw1,
                    b1 - args.lr * db1,
                    w2 - args.lr * dw2,
                    b2 - args.lr * db2]
```

Training and Validation

Train with default hyper-params and get the top-1 acc as follows:

```
Top-1 accuracy on the training set 0.8626
Top-1 accuracy on the test set 0.8542
```

Training curve:



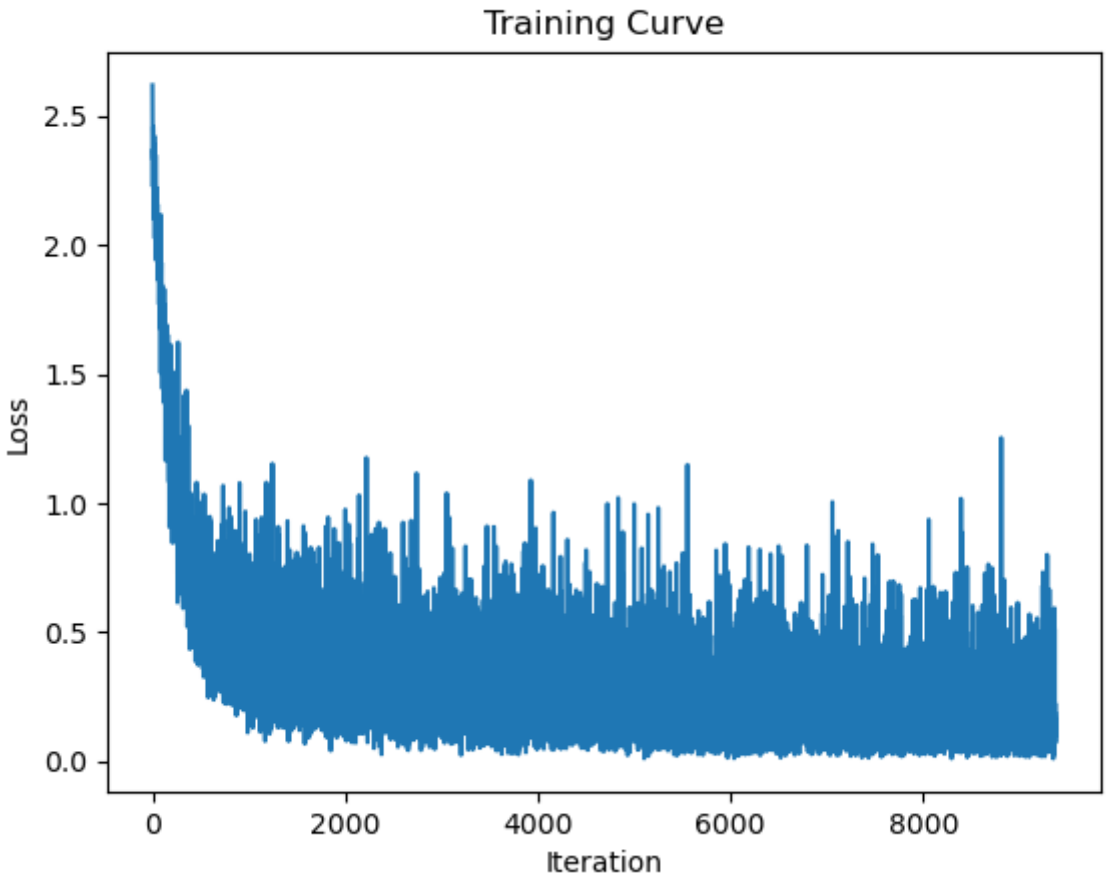
The loss curve shows that the training process is slow and has not reached a convergence. While supposing it outs of a low learning rate, we raise the lr to 0.01 and the epochs to 15, and train again:

```
$ python mlp.py --lr=0.01 --epoch=15
```

The top-1 acc are as follows:

```
Top-1 accuracy on the training set 0.9485
Top-1 accuracy on the test set 0.9272
```

The top-1 accuracy reached 90 percent and the training curve is as follow:



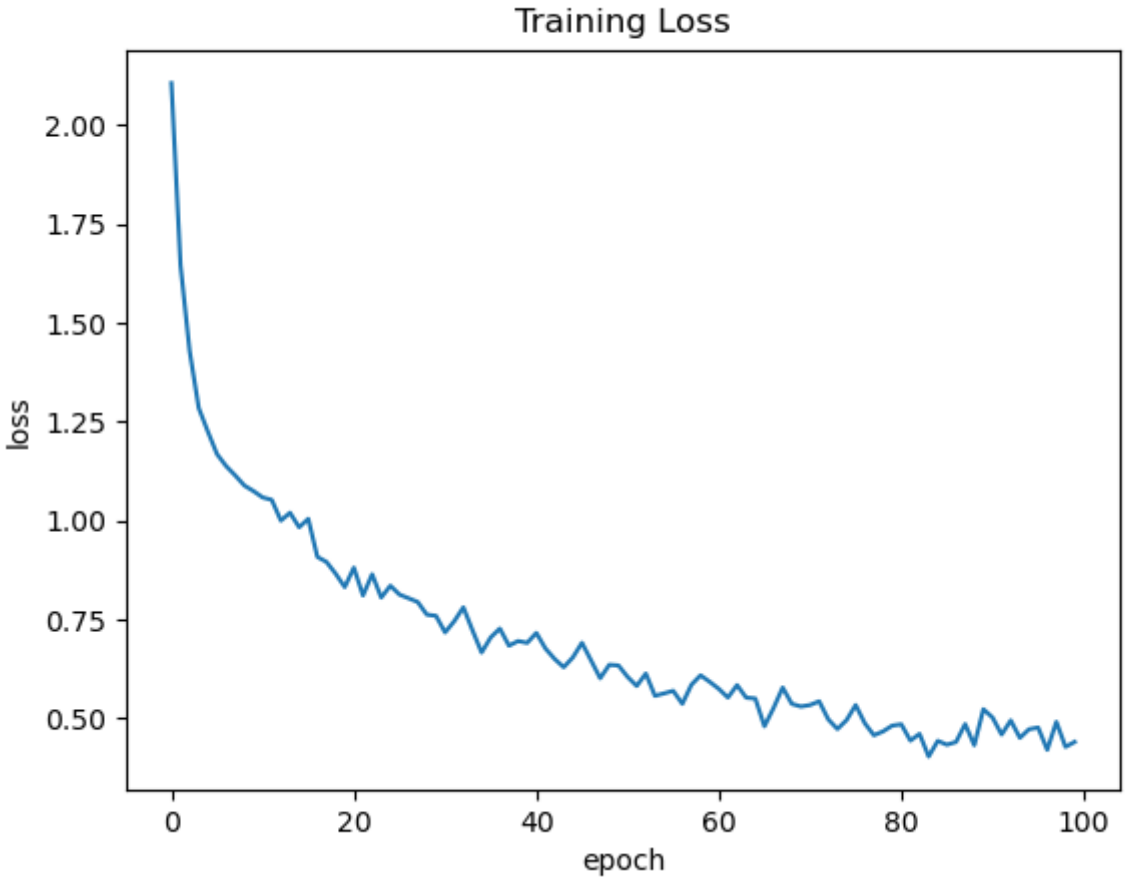
Convolutional Neural Network (CNN)

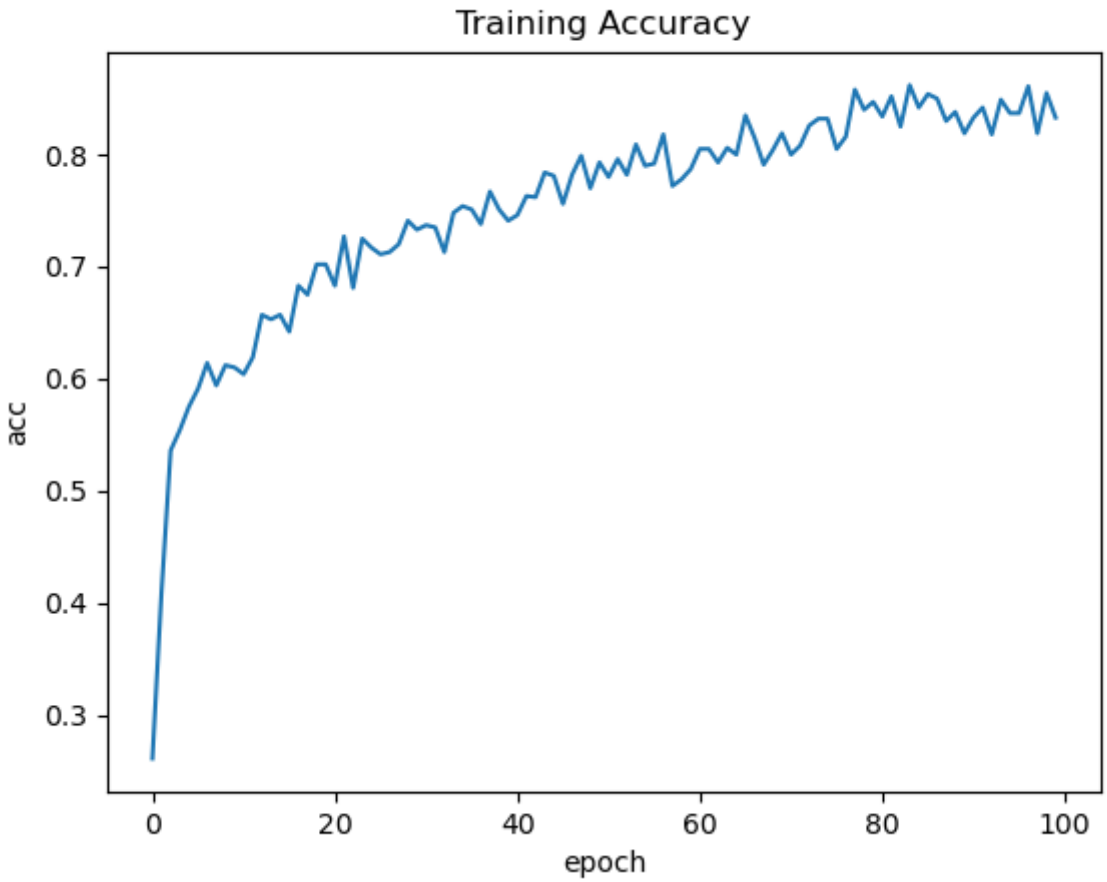
1. Train a model A for Task A, evaluate its accuracy on test set, and report training and test curves. Note that, the start code of this task is provided in CNN folder and it is runnable, you can directly run it and report the results with "python main.py".

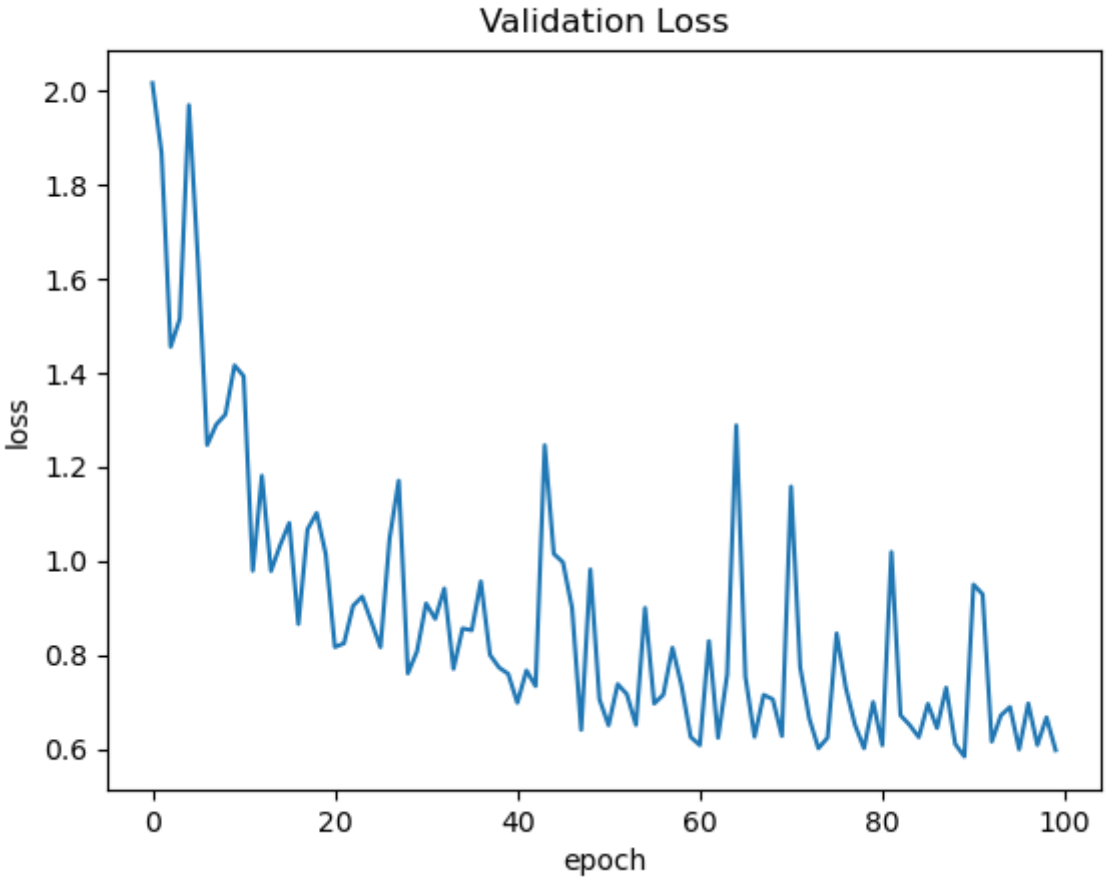
Run code with script `python main.py` directly, train with default hyper-parameters and save the best model through validation. The best accuracy on validation dataset is:

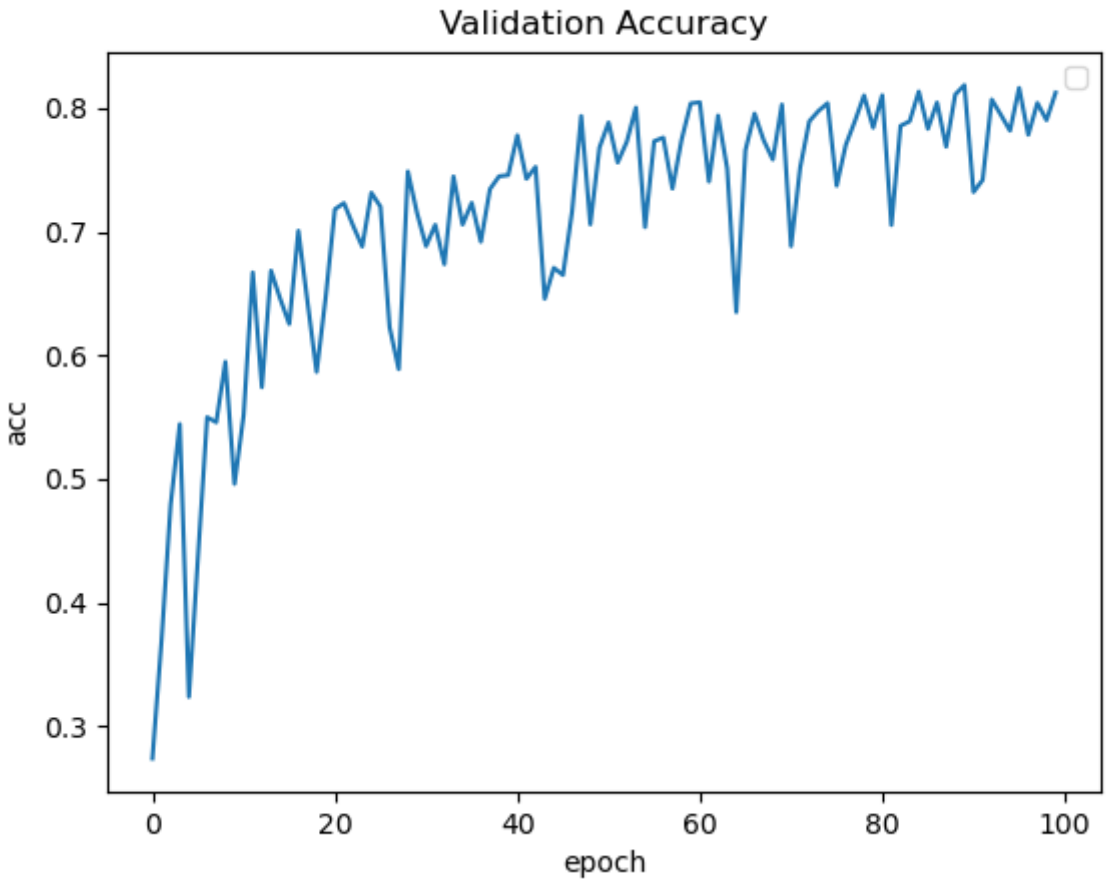
```
------------------------------------------------
valid acc:  0.8185185185185184
------------------------------------------------
```

The curves are as follows:


Training Loss
loss
epoch
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0
20
40
60
80
100


Training Accuracy
acc
epoch
0.3
0.4
0.5
0.6
0.7
0.8
0
20
40
60
80
100


Validation Loss
loss
epoch
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0
20
40
60
80
100



The weights of the model A are saved in the `./checkpoints` folder as `best_model.pt`.

2. Train a model B for Task B, evaluate its accuracy on test set, and report training and test curves.

The normal ResNet-18 has 18 layers, which include a former Conv layer, 8 basic res-blocks (each has 2 Conv layers) and a last FC layer. There is also an average pooling layer between the last Conv layer and the FC layer. According to the number of output channels, the 8 res-blocks can be divided into 4 layer-groups which each consists of 2 res-blocks. When we input a $3 \times 224 \times 224$ size tensor, the output sizes of each layer/layer-group are:

```
Conv1: (64,112,112)
BasicBlock 1 2: (64,56,56)
BasicBlock 3 4: (128,28,28)
BasicBlock 5 6: (256,14,14)
BasicBlock 7 8: (512,7,7)
AvgPool: (512,1,1)
FC: (-1,10)
```

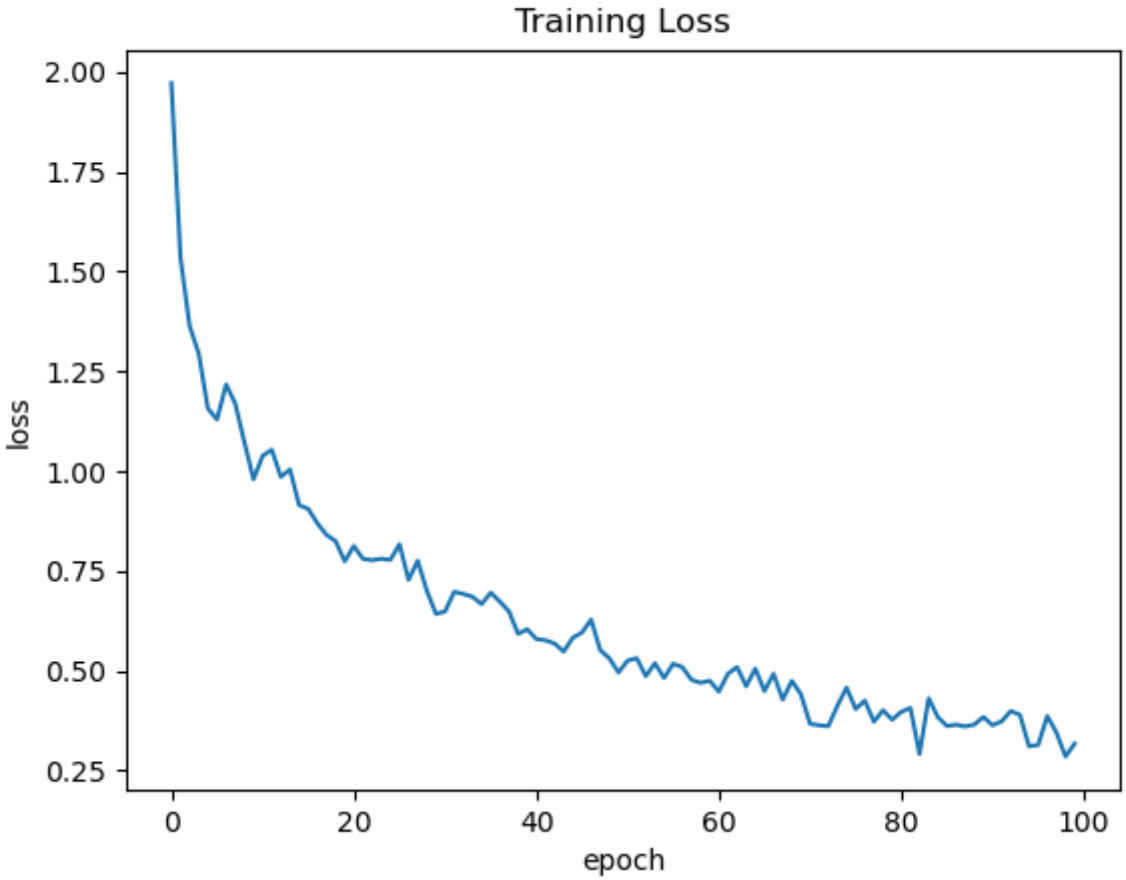
We consume that average-pooling layer which forms a 7×7 feature map into a single pixel may hide a lot of information, so we change the ResNet18 into a new structure. We add 2 more BasicBlocks after the 8th BasicBlock, whose output size is `(1024,4,4)`, and a max-pooling layer before the average-pooling layer to form the feature map into 2×2. Also we expand the FC part into 2 layers. The definition of the model are as follows:

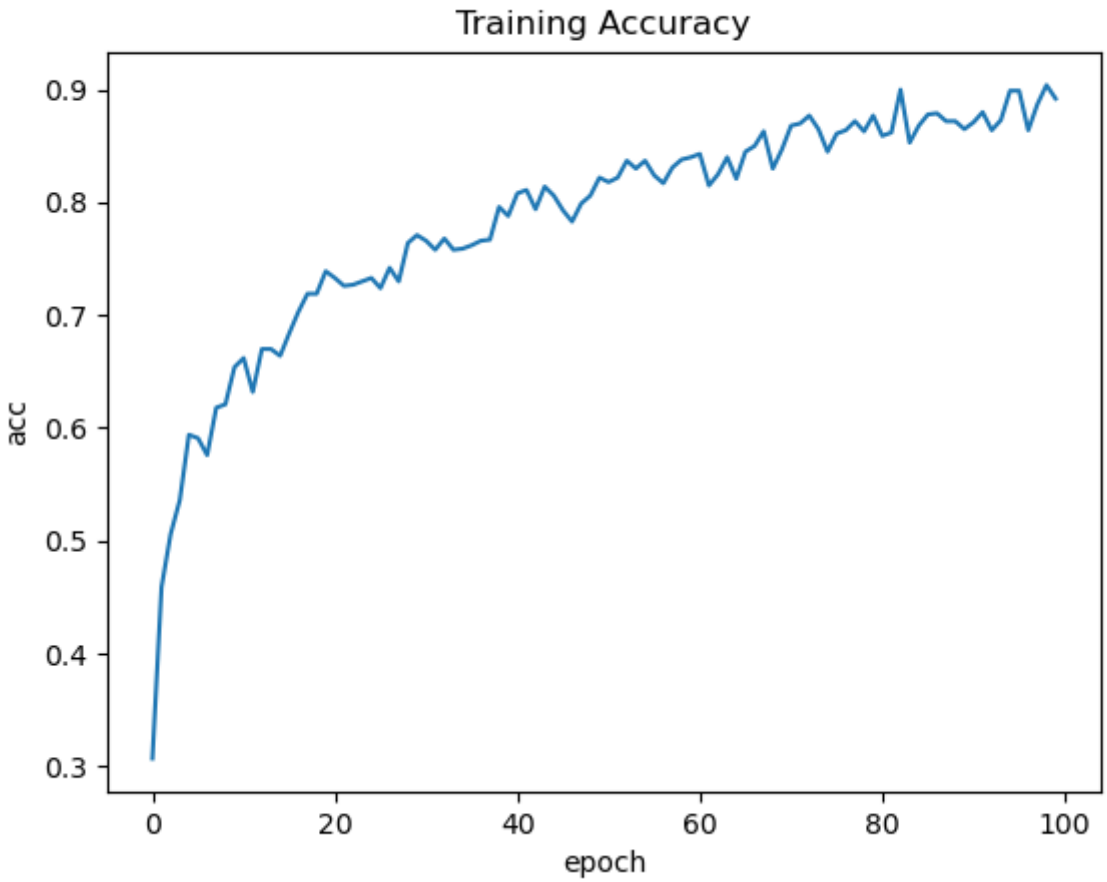
```
# The definition of the BasicBlock and FP could be finded in codes

class MyResNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.in_channels = 64
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU())
        self.layer1 = self._make_layer(BasicBlock, 64, 2, 2)
        self.layer2 = self._make_layer(BasicBlock, 128, 2, 2)
        self.layer3 = self._make_layer(BasicBlock, 256, 2, 2)
        self.layer4 = self._make_layer(BasicBlock, 512, 2, 2)
        self.layer5 = self._make_layer(BasicBlock, 1024, 2, 2)
        self.pool = nn.Sequential(
            nn.AdaptiveMaxPool2d((2, 2)),
            nn.AdaptiveAvgPool2d((1, 1)))
        self.fc = nn.Sequential(
            nn.Linear(1024 * BasicBlock.expansion, 256),
            nn.Linear(256, num_classes))

    def _make_layer(self, block, out_channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stride))
            self.in_channels = out_channels * block.expansion
        return nn.Sequential(*layers)
```

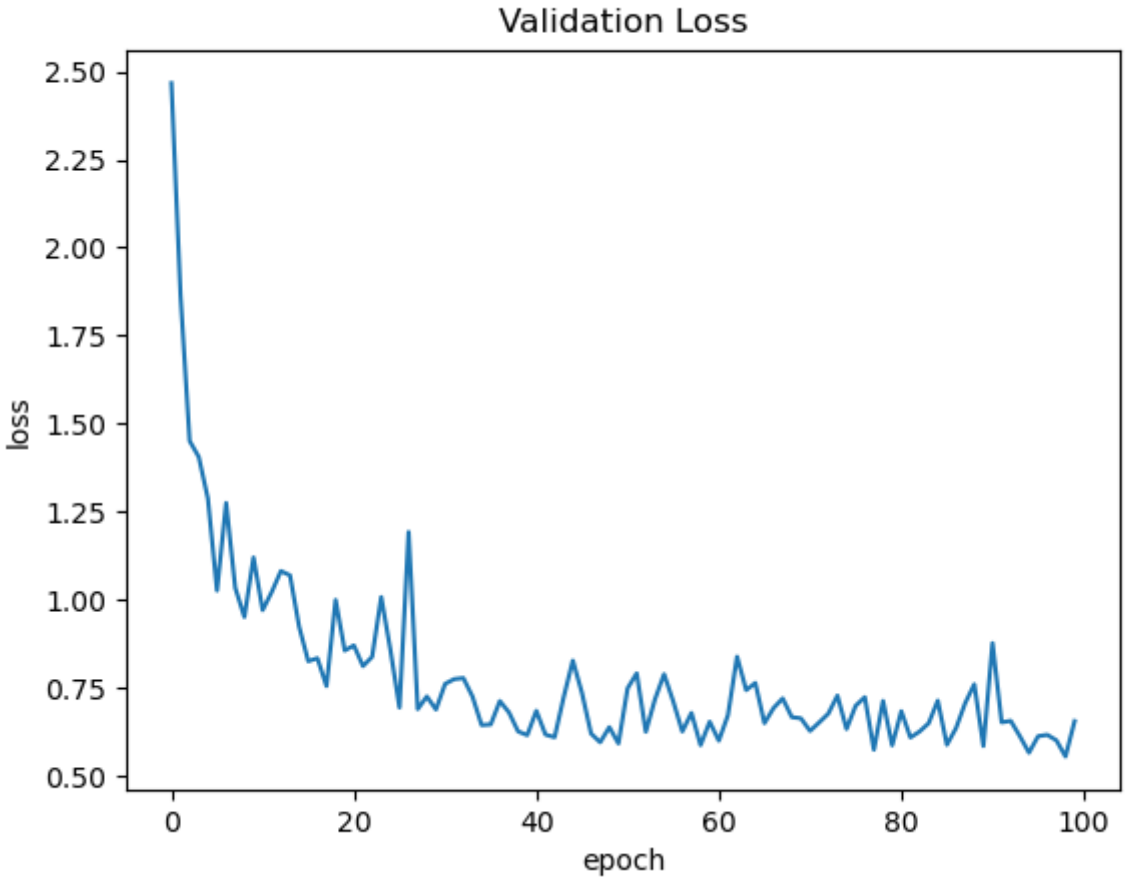
Train the model with the same optimization and hyper-parameters as the model A training. The result shows that the model B beats the model A with about 0.9 percent on validation accuracy:

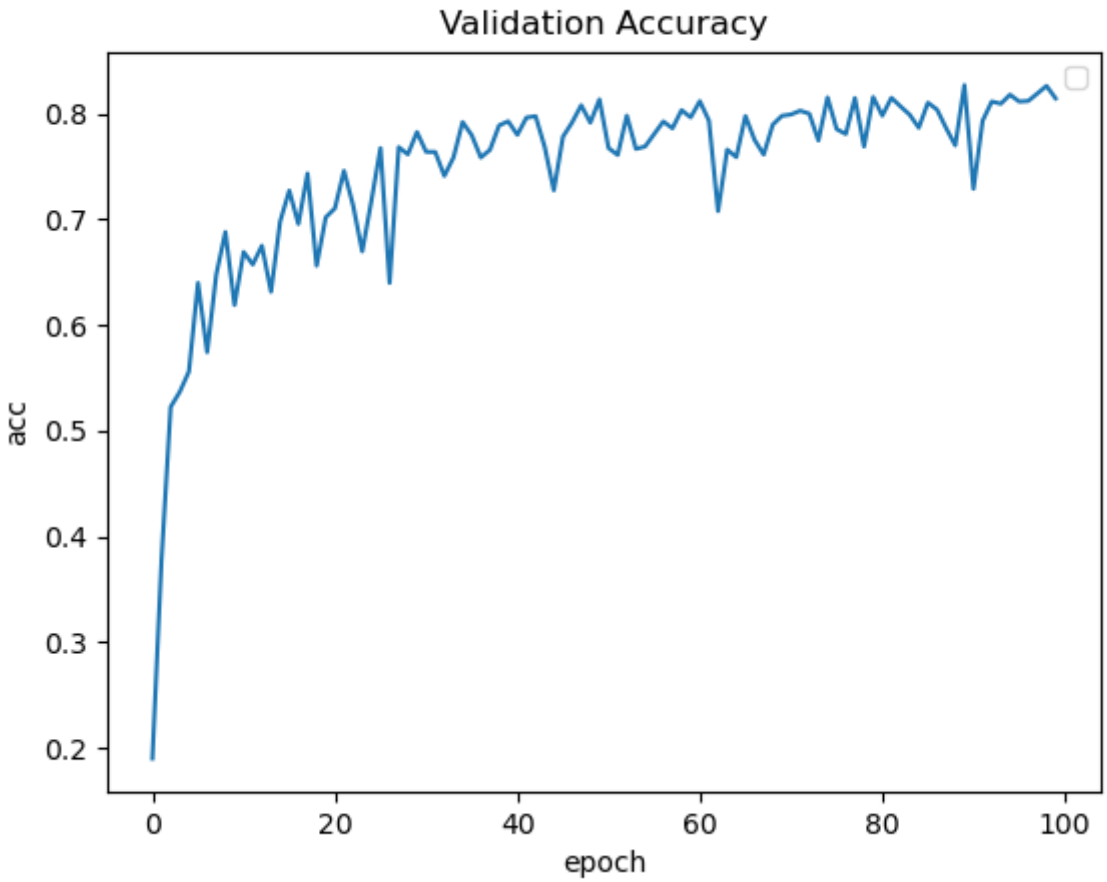
```
-----------------------------------------------
valid acc:  0.8272222222222222
-----------------------------------------------
```

The training and validation curves are as follows:


Training Loss
loss
epoch
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
20
40
60
80
100


Training Accuracy
acc
epoch
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
20
40
60
80
100


Validation Loss
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
loss
0
20
40
60
80
100
epoch



From the curves we can also see that the model B makes a faster convergence.

The weights of the model B are saved in the `./checkpoints` folder as `b.ckpt`.

3. Visualize the features before the last fully-connected layer of model A using t-SNE.

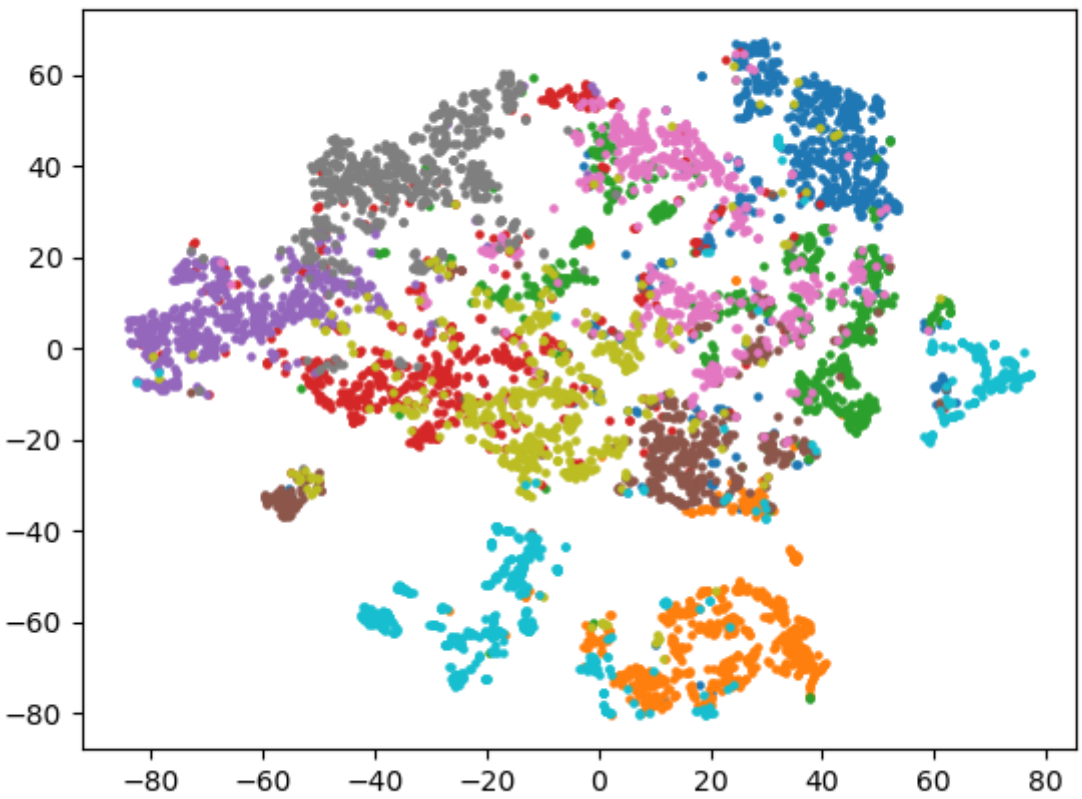
To visualize the features, we need to get the features first. So we remove the last FC layer of the model and get the features.

```
if args.use_ckpt is not None:
        state_dict = torch.load(args.use_ckpt)
        model.load_state_dict(state_dict)
        test_ckpt(model, valid_loader, device)
        fea_model = copy.copy(model)
        fea_model.fc = models.Identity()
        tSNE_visualization(fea_model, valid_loader, device)
```

The `tSNE_visualization()` function is defined as:

```python
def tSNE_visualization(model, valid_loader, device, n_comp=2):
    model.eval()
    feature = np.zeros((0,512))
    label = []
    for inputs, labels in valid_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = model(inputs)
        feature = np.vstack((feature, outputs.detach().cpu().numpy()))
        label += labels.detach().cpu().numpy().tolist()
    print(feature.shape)
    label = np.array(label)
    vf = TSNE(n_components=n_comp).fit_transform(feature)

    plt.figure('tsne')
    for i in range(min(label), max(label)+1):
        plt.scatter(vf[label==i, 0], vf[label==i, 1])
    plt.show()
```

The visualization result of the validation set is:



4. Use data augmentation and learning rate strategy to improve the performance of your model B and give a detailed ablation study in your report.

default: SGD(lr=0.001), **w/** RandomCrop, **w/** RandomHorizontalFlip

We use 3 categories of strategies to try to improve the performance of our model, which are:

- data augmentation
- different optimizer
- learning rate strategy

We commit 10 sub-experiments besides the default one to get the ablation performance of our strategies. The results of the sub-experiments are as follows:

model setting	**best test acc**
default	82.72%
default **w/** stepLR(step_size=30, gamma=0.5)	84.96%
default **w/** stepLR(step_size=30, gamma=0.1)	84.15%
Adam	82.93%
Adam **w/** stepLR(step_size=30, gamma=0.5)	84.33%
Adam **w/** stepLR(step_size=30, gamma=0.1)	81.70%
default **w/** RandomRotation(30)	83.76%
default **w/** ColorJitter(0.5, 0.5, 0.5, 0.1)	77.48%
default **w/o** RandomCrop	75.59%
default **w/o** RandomHorizontalFlip	82.20%
default **w/** stepLR(step_size=30, gamma=0.5), RandomRotation(30)	**84.96%**

From the information showed above we could see that the random-crop augmentation could lead to a 7 percent enhancement on test accuracy, while the step-lr strategy, random-rotation and random-horizontal-flip all contribute a relatively small improvement. However, using Adam optimizer in this task doesn't lead to a obvious increasing on acc, and using color-jitter even pulls down the performance.

Our final model uses step-lr strategy and adds random-rotation as an augmentation, which finally get about 2.2% improvement on test acc comparing to the default model.