# LAB **1-1** : Train a CNN model via Pytorch

## **Show the weight and bias distribution each layer.**

Start with *LAB-1-1-example-code.ipynb*, we would show the weight and bias distribution each layer after finish training stage. So, the first thing is how to get weight and bias parameters. The second thing is how to calculate the distribution. The third thing is how to plot it. The solutions are described below.

1. list(net.conv1.parameters())[0] can get conv1’s weight parameters.

list(net.conv1.parameters())[1] can get conv1’s bias parameters.

1. L1\_W.view(-1).data.cpu().numpy() enforce weight tensor to 1-D value array.

L1\_b.view(-1).data.cpu().numpy() enforce bias tensor to 1-D value array.

1. axs[0].hist(L1\_W, bins=n\_bins) can plot weight histogram.

axs[0].hist(L1\_b, bins=n\_bins) can plot bias histogram.

The programming codes is shown below.

*import matplotlib.pyplot as plt*

*import numpy as np*

*##Layer1 Weight/Bias Distribution*

*L1\_W = list(net.conv1.parameters())[0]*

*L1\_b = list(net.conv1.parameters())[1]*

*L1\_W=L1\_W.view(-1).data.cpu().numpy()*

*L1\_b=L1\_b.view(-1).data.cpu().numpy()*

*##Layer2 Weight/Bias Distribution*

*L2\_W = list(net.conv2.parameters())[0]*

*L2\_b = list(net.conv2.parameters())[1]*

*L2\_W=L2\_W.view(-1).data.cpu().numpy()*

*L2\_b=L2\_b.view(-1).data.cpu().numpy()*

*##FC1*

*fc1\_W = list(net.fc1.parameters())[0]*

*fc1\_W=fc1\_W.view(-1).data.cpu().numpy()*

*fc1\_b = list(net.fc1.parameters())[1]*

*fc1\_b=fc1\_b.view(-1).data.cpu().numpy()*

*##FC2*

*fc2\_W = list(net.fc2.parameters())[0]*

*fc2\_W=fc2\_W.view(-1).data.cpu().numpy()*

*fc2\_b = list(net.fc2.parameters())[1]*

*fc2\_b=fc2\_b.view(-1).data.cpu().numpy()*

*##FC3*

*fc3\_W = list(net.fc3.parameters())[0]*

*fc3\_W=fc3\_W.view(-1).data.cpu().numpy()*

*fc3\_b = list(net.fc3.parameters())[1]*

*fc3\_b=fc3\_b.view(-1).data.cpu().numpy()*

*n\_bins=100*

*fig, axs = plt.subplots(1, 5, sharey=True, tight\_layout=True)*

*# We can set the number of bins with the `bins` kwarg*

*axs[0].hist(L1\_W, bins=n\_bins)*

*axs[1].hist(L2\_W, bins=n\_bins)*

*axs[2].hist(fc1\_W, bins=n\_bins)*

*axs[3].hist(fc2\_W, bins=n\_bins)*

*axs[4].hist(fc3\_W, bins=n\_bins)*

*n\_bins=100*

*fig, axs = plt.subplots(1, 5, sharey=True, tight\_layout=True)*

*# We can set the number of bins with the `bins` kwarg*

*axs[0].hist(L1\_b, bins=n\_bins)*

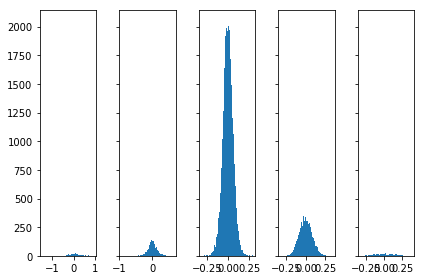
*axs[1].hist(L2\_b, bins=n\_bins)*

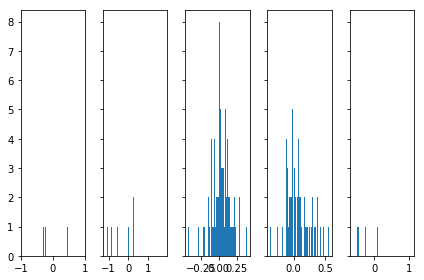
*axs[2].hist(fc1\_b, bins=n\_bins)*

*axs[3].hist(fc2\_b, bins=n\_bins)*

*axs[4].hist(fc3\_b, bins=n\_bins)*

The results are shown below. From left to right is conv1, conv2, fc1, fc2, fc3, respectively. From top to bottom is weight, bias, respectively.

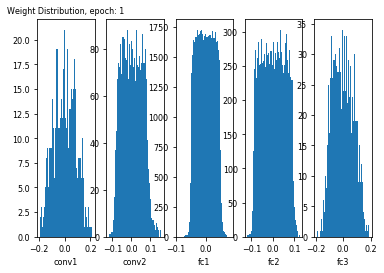




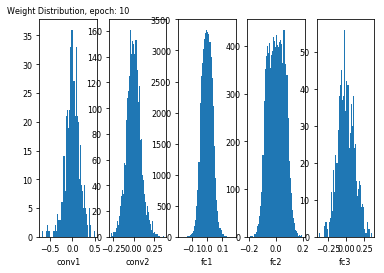
Furthermore, if we look into the weight and bias distribution in each epoch, we can easily find the distributions become shaper and shaper and the trend will be normal distributions. Here we show the weight and bias distributions in epoch 1, 10, 20, 30, 40 and 50.

Weight Distribution

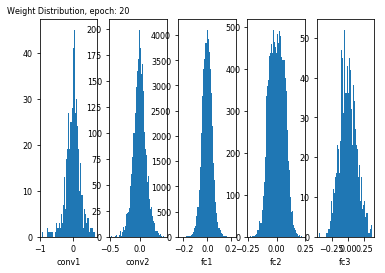
1 epoch, training accuracy: 24.1780%, loss = 0.0645



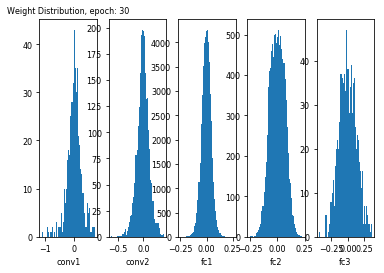
10 epoch, training accuracy: 56.7200%, loss = 0.0381



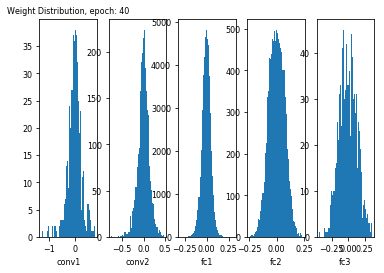
20 epoch, training accuracy: 63.1440%, loss = 0.0329



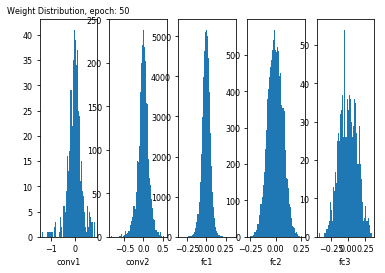
30 epoch, training accuracy: 65.3800%, loss = 0.0307



40 epoch, training accuracy: 67.1980%, loss = 0.0291

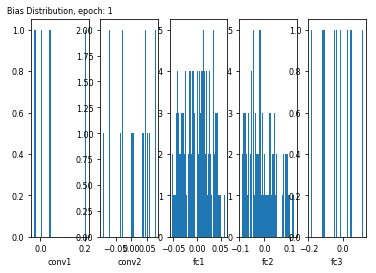


50 epoch, training accuracy: 68.3040%, loss = 0.0282

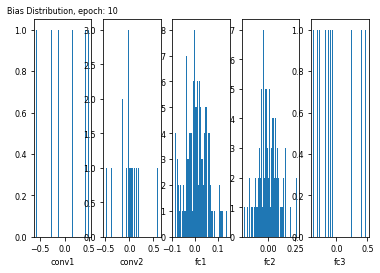


Bias Distribution

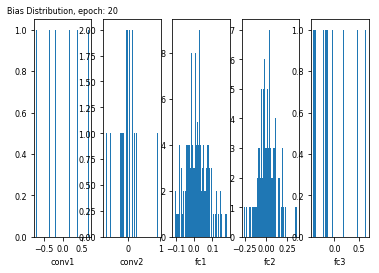
1 epoch, training accuracy: 24.1780%, loss = 0.0645



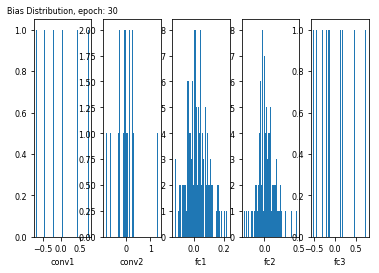
10 epoch, training accuracy: 56.7200%, loss = 0.0381



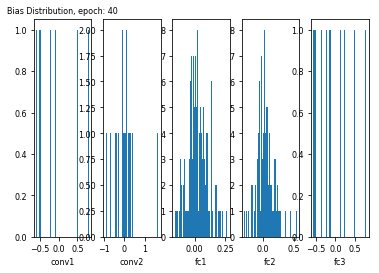
20 epoch, training accuracy: 63.1440%, loss = 0.0329



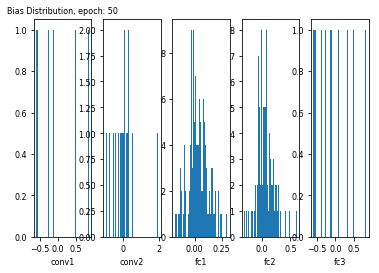
30 epoch, training accuracy: 65.3800%, loss = 0.0307



40 epoch, training accuracy: 67.1980%, loss = 0.0291



50 epoch, training accuracy: 68.3040%, loss = 0.0282



## **Show total loss and accuracy (test case).**

In test case, we get loss by *loss = criterion(outputs, labels)* and define *accuracy = 100\*correct\_num/total\_num*. The programming codes is as below. The total loss and accuracy is 0.81 and 70.8 %, respectively.

*correct = 0*

*total = 0*

*with torch.no\_grad():*

*for data in testloader:*

*images, labels = data*

*images = images.to(device)*

*labels = labels.to(device)*

*outputs = net(images)*

*loss = criterion(outputs, labels)*

*\_, predicted = torch.max(outputs, 1)*

*total += labels.size(0)*

*correct += (predicted == labels).sum().item()*

*print('Total accuracy is: %.1f %% and loss is: %.2f' % (*

*100 \* correct / total,loss))*

Total accuracy is: 70.8 % and loss is: 0.81

## **Show accuracy for each class.**

To show accuracy for each class, we just need to calculate class\_correct and class\_total in each calss. And define *accuracy = 100\*class\_correct/class\_total*. The programming codes and results are shown below.

*class\_correct = list(0. for i in range(10))*

*class\_total = list(0. for i in range(10))*

*with torch.no\_grad():*

*for data in testloader:*

*images, labels = data*

*images = images.to(device)*

*labels = labels.to(device)*

*outputs = net(images)*

*\_, predicted = torch.max(outputs, 1)*

*c = (predicted == labels).squeeze()*

*for i in range(4):*

*label = labels[i]*

*class\_correct[label] += c[i].item()*

*class\_total[label] += 1*

*for i in range(10):*

*print('Accuracy of %5s : %.1f %%' % (*

*classes[i], 100 \* class\_correct[i] / class\_total[i]))*

Accuracy of plane : 74.3 %

Accuracy of car : 91.3 %

Accuracy of bird : 57.9 %

Accuracy of cat : 49.6 %

Accuracy of deer : 65.2 %

Accuracy of dog : 63.7 %

Accuracy of frog : 85.1 %

Accuracy of horse : 75.0 %

Accuracy of ship : 72.6 %

Accuracy of truck : 79.4 %

* Problems & solutions :
* Experiment setup :
* Results :
* Analysis :

# LAB 1-2 : Train a new dataset

## Load and normalizing the Food-11 training and evaluation datasets using torchvision.

First, we decide to use the API called "ImageFolder" to load Food-11 dataset. So, we need adjust the directory architecture to match the need of "ImageFolder". It assumes that images are organized in the following way:

root/class\_0/aaa.jpeg

root/class\_0/bbb.jpeg

…

root/class\_N/aaa.jpeg

root/class\_N/bbb.jpeg

Therefore, we create directories like below by **mkdir** commands at path **/tmp/work/**.

Food-11\_03

training

0

1

2

…

10

evaluation

0

1

2

…

10

validation

0

1

2

…

10

And then copy the Food-11 dataset at path /tmp/dataset-nctu/Food-11 to the directories we create before, respectively. Here we just use the commands like below.

***cp*** */tmp/dataset-nctu/Food-11/training/1\_\*.jpg /tmp/work/Food-11\_03/training/1* ***cp*** */tmp/dataset-nctu/Food-11/training/2\_\*.jpg /tmp/work/Food-11\_03/training/2* ***cp*** */tmp/dataset-nctu/Food-11/training/3\_\*.jpg/tmp/work/Food-11\_03/training/3…*

Now we can prepare our dataset by 1.1, 1.2, and 1.3.

1.1 is set transform functions for train data and test data.

*#The transform function for train data*

*transform\_train = transforms.Compose([*

*transforms.Resize(256),*

*transforms.RandomResizedCrop(224),*

*transforms.RandomHorizontalFlip(),*

*transforms.RandomVerticalFlip(),*

*transforms.ToTensor(),*

*transforms.Normalize(mean=[0.485, 0.456, 0.406],*

*std=[0.229, 0.224, 0.225])*

*])*

*#The transform function for test data*

*transform\_test = transforms.Compose([*

*transforms.Resize(256),*

*transforms.CenterCrop(224),*

*transforms.ToTensor(),*

*transforms.Normalize(mean=[0.485, 0.456, 0.406],*

*std=[0.229, 0.224, 0.225])*

*])*

1.2 is use API to load Food-11 train dataset and test dataset

*#Use API to load Food-11 train dataset*

*trainset = torchvision.datasets.ImageFolder(root='/tmp/work/Food-11\_03/training', transform=transform\_train)*

*#Use API to load Food-11 test dataset*

*testset = torchvision.datasets.ImageFolder(root='/tmp/work/Food-11\_03/evaluation', transform=transform\_test)*

1.3 is create DataLoader to draw samples from the dataset

*trainloader = torch.utils.data.DataLoader(trainset, batch\_size=32,*

*shuffle=True, num\_workers=2)*

*testloader = torch.utils.data.DataLoader(testset, batch\_size=32,*

*shuffle=False, num\_workers=2)*

## **Choose a model by the paper report.**

In fact, I have no idea which model is best to classify Food-11 dataset. Therefore I study several papers about food classifier and finally reference the paper “ChineseFoodNet: A Large-scale Image Dataset for Chinese Food Recognition”.

In this paper, it mentioned four well-known CNNs architectures show in Fig. 2. And it also shows the recognition rate of different deep neural networks on their food dataset in Tabel I. So, I choose the model with the highest recognition rate. It means DenseNet201 is my target.

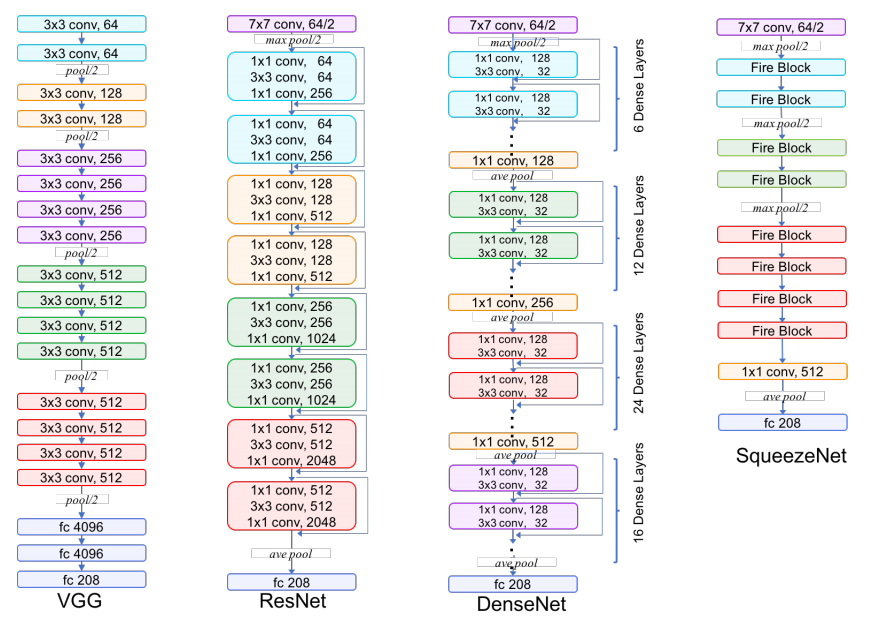
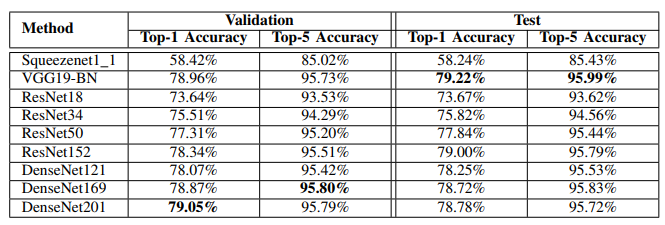


Fig. 2 Basic architectures of four well-known CNNs. From left to right, the architectures are VGG, Resnet, Densenet, and Squeezenet, respectively.

Table I Recognition rate of different deep neural networks on their food dataset. Both TOP-1 and TOP-5 accuracy are shown on validation set and test set.



Fortunately, DenseNet201 is one of the default model in tourchvision, therefore we just declare a new model like the following programming codes.

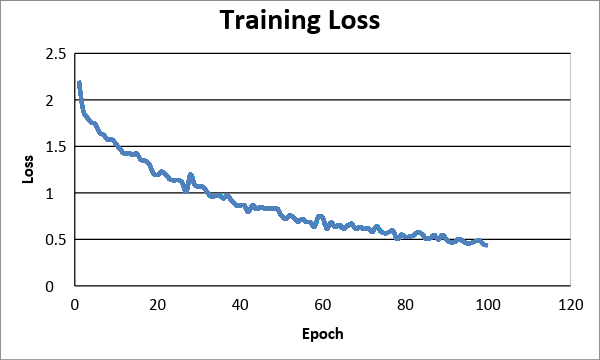
*net = torchvision.models.densenet201(pretrained=False)*

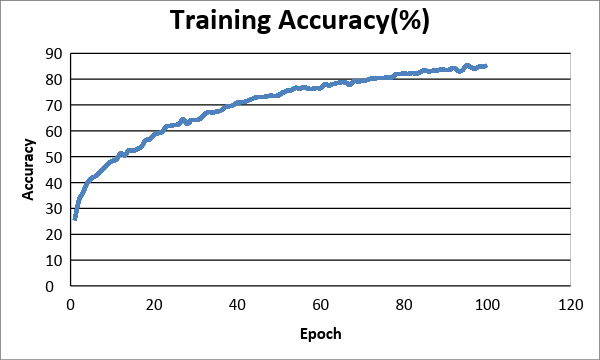
*num\_ftrs = net.classifier.in\_features*

*net.classifier = nn.Linear(num\_ftrs, n\_class)*

*net = net.to(device)*

Then, train this model by *trainloader* with CrossEntropyLoss, SGD optimizer, learning rate 0.01, and epoch=100. The training result is simply shown below. The training loss and accuracy in epoch 100 is 0.435 and 85.3740 respectively.





## Total loss and accuracy (test case).

Now, we evaluate the training model described above by testloader. The programming codes are shown below.

*correct = 0*

*total = 0*

*class\_correct = list(0. for i in range(11))*

*class\_total = list(0. for i in range(11))*

*with torch.no\_grad():*

*for data in testloader:*

*images, labels = data*

*images = images.to(device)*

*labels = labels.to(device)*

*outputs =* ***net****(images) #net=DenseNet201 we train before*

*loss = criterion(outputs, labels)*

*\_, predicted = torch.max(outputs, 1)*

*total += labels.size(0)*

*correct += (predicted == labels).sum().item()*

*c = (predicted == labels).squeeze()*

*for i in range(4):*

*label = labels[i]*

*class\_correct[label] += c[i].item()*

*class\_total[label] += 1*

*print('Total accuracy is: %.1f %% and loss is: %.2f' % (*

*100 \* correct / total,loss))*

*for i in range(11):*

*print('Accuracy of %5s : %.1f %%' % (*

*classes[i], 100 \* class\_correct[i] / class\_total[i]))*

And the test result of total loss and accuracy is list below.

Total accuracy is: 82.7 % and loss is: 0.22

Accuracy of bread : 79.2 %

Accuracy of dairy products : 70.0 %

Accuracy of dessert : 92.9 %

Accuracy of egg : 76.7 %

Accuracy of fried food : 63.6 %

Accuracy of meat : 77.8 %

Accuracy of noodles & pasta : 80.8 %

Accuracy of rice : 90.0 %

Accuracy of seafood : 66.7 %

Accuracy of soup : 86.1 %

Accuracy of vegetables & fruits : 95.3 %

## **Different configurations of using** LeNets

We also tried the LeNet and changed the combination of input size and network configuration.

Network Source Code

*class Net(nn.Module):*

*#define the layers*

*def \_\_init\_\_(self):*

*super(Net, self).\_\_init\_\_()*

*self.conv1 = nn.Conv2d(3, 6, 5)*

*self.pool = nn.MaxPool2d(2, 2)*

*self.conv2 = nn.Conv2d(6, 16, 5)*

*self.fc1 = nn.Linear(16 \* linear\_size \* linear\_size, fc1\_out)*

*self.fc2 = nn.Linear(fc1\_out, fc2\_out)*

*self.fc3 = nn.Linear(fc2\_out, 11)*

*self.relu = nn.ReLU()*

*#concatenate these layers*

*def forward(self, x):*

*x = self.pool(self.relu(self.conv1(x)))*

*x = self.pool(self.relu(self.conv2(x)))*

*x = x.view(-1, 16 \* linear\_size \* linear\_size)*

*x = self.relu(self.fc1(x))*

*x = self.relu(self.fc2(x))*

*x = self.fc3(x)*

*return x*

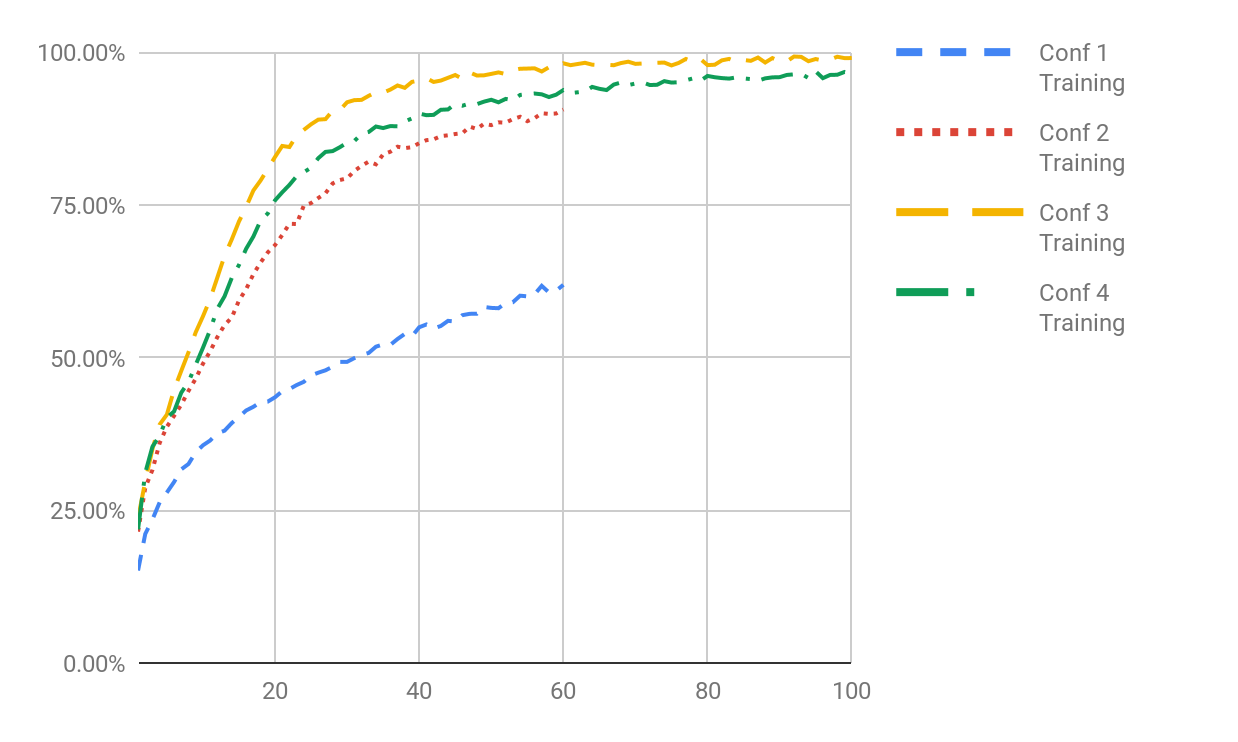
Network Configuration

|  |  |  |  |
| --- | --- | --- | --- |
|  | Conf. 1 | Conf. 2 | Conf. 3 |
| input | 3@64x64 | 3@384x384 | 3@384x384 |
| conv1 (5x5) | 6@60x60 | 6@380x380 | 6@380x380 |
| pool (2x2) | 6@30x30 | 6@190x190 | 6@190x190 |
| conv2 (5x5) | 16@26x26 | 16@186x86 | 16@186x86 |
| pool (2x2) | 16@13x13 | 16@93x93 | 16@93x93 |
| fc1 | 128 | 128 | 1024 |
| fc2 | 84 | 84 | 1024 |
| fc3 | 11 | 11 | 11 |

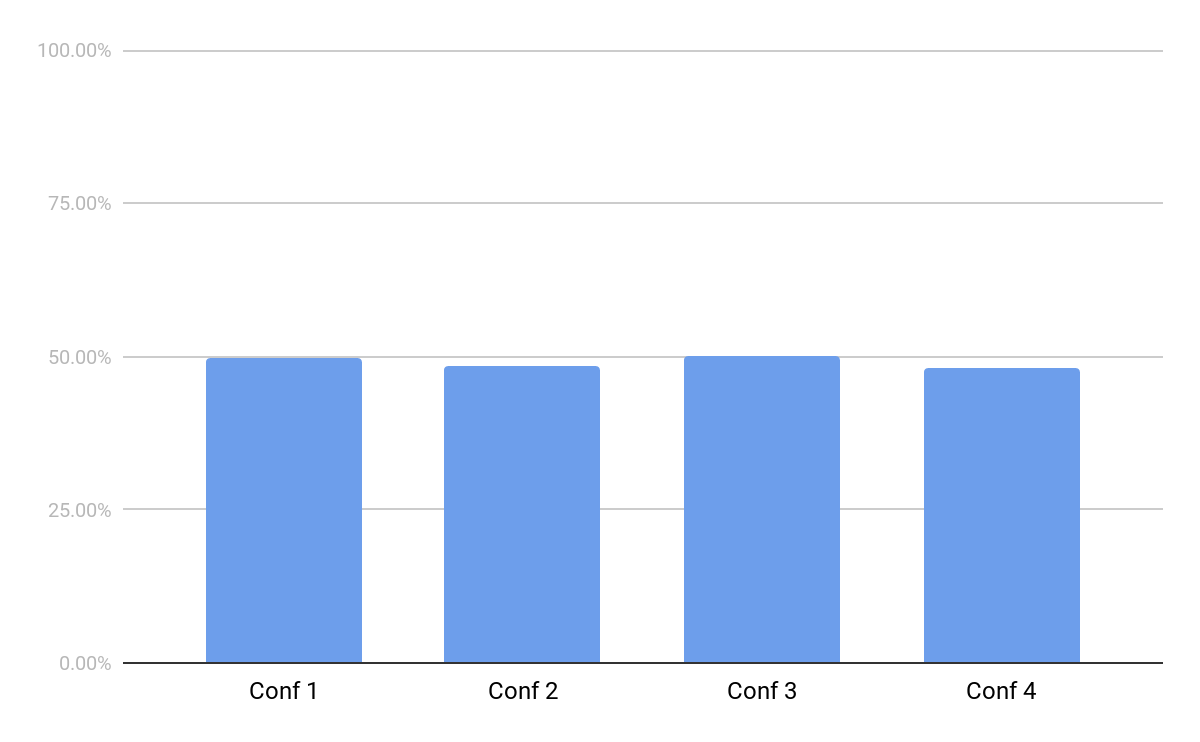
|  |  |  |
| --- | --- | --- |
|  | Conf. 4 | Conf. 5 |
| input | 3@512x512 | 3@1024x1024 |
| conv1 (5x5) | 6@508x508 | 6@1020x1020 |
| pool (2x2) | 6@254x254 | 6@510x510 |
| conv2 (5x5) | 16@250x250 | 16@506x506 |
| pool (2x2) | 16@125x125 | 16@253x253 |
| fc1 | 256 | 1024 |
| fc2 | 84 | 84 |
| fc3 | 11 | 11 |

Note: We met out of memory CUDA runtime error in Configuration 5.

Training Accuracy



Test Accuracy



Configuration 1

Accuracy of the network on the 3347 test images: 49.66%, and loss is: 0.048  
Accuracy of Bread : 24 %  
Accuracy of DairyProduct : 21 %  
Accuracy of Dessert : 42 %  
Accuracy of Egg : 42 %  
Accuracy of Friedfood : 49 %  
Accuracy of Meat : 57 %  
Accuracy of Noodles-Pasta : 46 %  
Accuracy of Rice : 15 %  
Accuracy of Seafood : 58 %  
Accuracy of Soup : 70 %  
Accuracy of Vegetable : 79 %

Configuration 2

Accuracy of the network on the 3347 test images: 48.55%, and loss is: 0.081  
Accuracy of Bread : 32 %  
Accuracy of DairyProduct : 30 %  
Accuracy of Dessert : 29 %  
Accuracy of Egg : 37 %  
Accuracy of Friedfood : 41 %  
Accuracy of Meat : 71 %  
Accuracy of Noodles-Pasta : 55 %  
Accuracy of Rice : 41 %  
Accuracy of Seafood : 52 %  
Accuracy of Soup : 60 %  
Accuracy of Vegetable : 78 %

Configuration 3

Accuracy of the network on the 3347 test images: 50 %  
Accuracy of Bread : 34 %  
Accuracy of DairyProduct : 25 %  
Accuracy of Dessert : 29 %  
Accuracy of Egg : 39 %  
Accuracy of Friedfood : 47 %  
Accuracy of Meat : 63 %  
Accuracy of Noodles-Pasta : 57 %  
Accuracy of Rice : 39 %  
Accuracy of Seafood : 61 %  
Accuracy of Soup : 69 %  
Accuracy of Vegetable : 75 %

Configuration 4

Accuracy of the network on the 3347 test images: 48 %  
Accuracy of Bread : 35 %  
Accuracy of DairyProduct : 25 %  
Accuracy of Dessert : 37 %  
Accuracy of Egg : 34 %  
Accuracy of Friedfood : 31 %  
Accuracy of Meat : 60 %  
Accuracy of Noodles-Pasta : 54 %  
Accuracy of Rice : 33 %  
Accuracy of Seafood : 50 %  
Accuracy of Soup : 66 %  
Accuracy of Vegetable : 84 %

Overfitting

Although these configurations of training accuracy are high, the test accuracy of corresponding configuration are still related low. The effect of training overfitting is easily observed. One technique that is often used to control the overfitting is that of regularization. We add a penalty term of the loss function in order to discourage the weights from reaching large values.

In Pytorch, weight\_decay control the penalty as shown below. We test it in Configuration 1 and 2 with 1e-5 and Configuration 2 with 1e-3. However, the results are not good though.

*#loss function*

*criterion = nn.CrossEntropyLoss()*

*#optimization algorithm*

*optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9, weight\_decay=1e-5)*

in training:

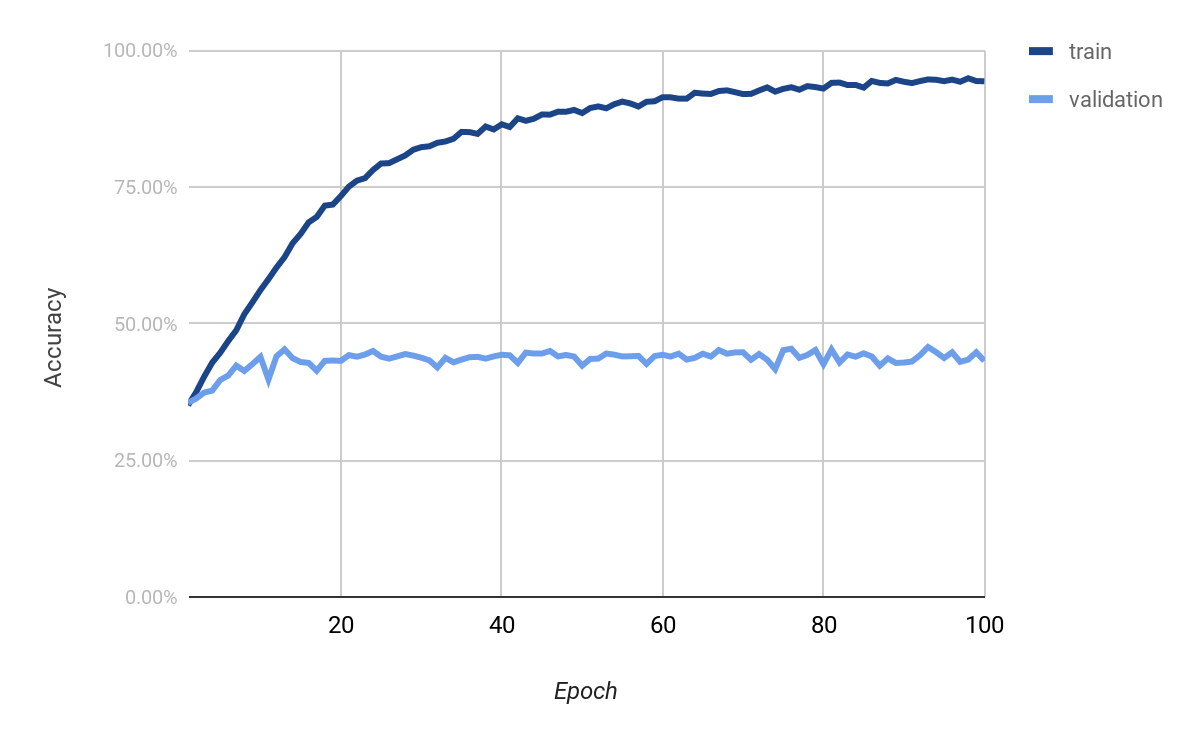
96 epoch, training accuracy: 96.03%, loss = 0.0038  
97 epoch, training accuracy: 96.32%, loss = 0.0035  
98 epoch, training accuracy: 96.44%, loss = 0.0036  
99 epoch, training accuracy: 96.14%, loss = 0.0038  
100 epoch, training accuracy: 96.90%, loss = 0.0031

In test:

Accuracy of the network on the 3347 test images: 49%, and loss is: 0.107  
Accuracy of Bread : 52 %  
Accuracy of DairyProduct : 22 %  
Accuracy of Dessert : 70 %  
Accuracy of Egg : 30 %  
Accuracy of Friedfood : 42 %  
Accuracy of Meat : 23 %  
Accuracy of Noodles-Pasta : 67 %  
Accuracy of Rice : 49 %  
Accuracy of Seafood : 37 %  
Accuracy of Soup : 50 %  
Accuracy of Vegetable : 68 %

It’s still overfitting.

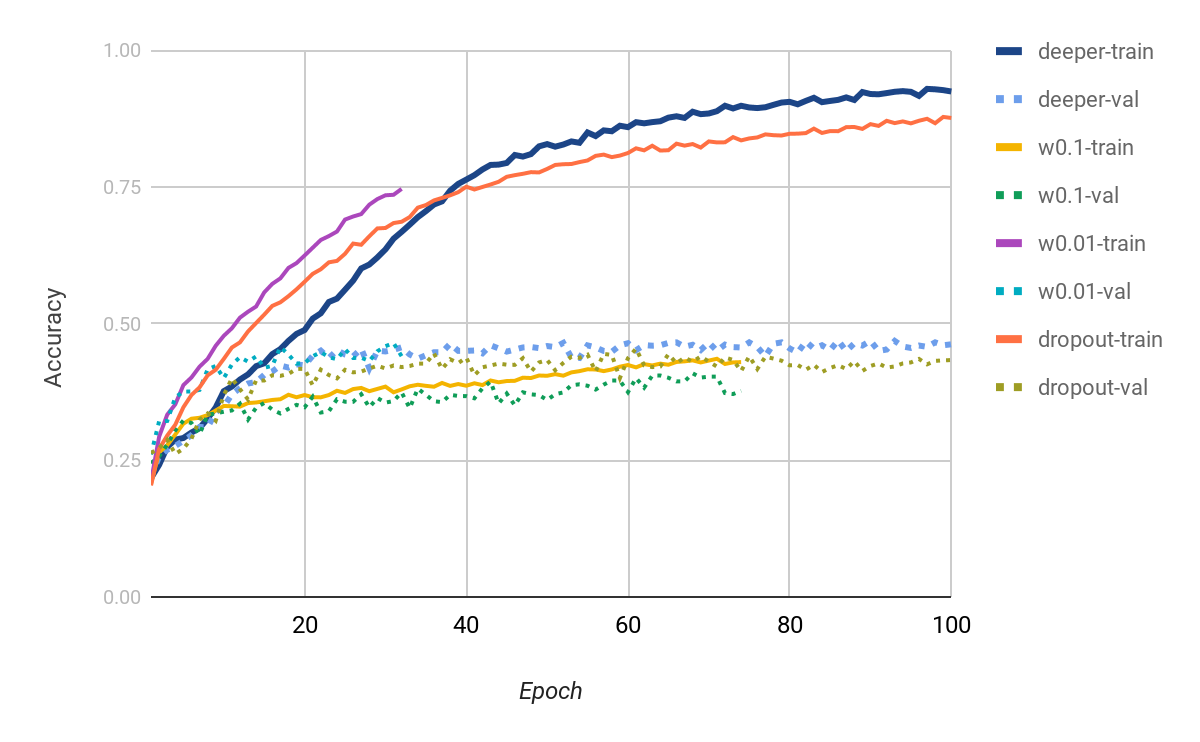
After using the validated by validation data set, we can see the accuracy of validation is always under 50%.



The validation accuracy was not improved by using deeper convolutional network as below. And we further investigate the weight decay with 0.1 and 0.01 and dropout technology. The figure are shown as deeper, w0.1 (weight decay 0.1), w0.01 (weight decay 0.01), and dropout.

Deeper Network Configuration

|  |  |
| --- | --- |
|  | Deeper |
| input | 3@384x384 |
| conv1 (5x5) | 6@380x380 |
| pool (2x2) | 6@190x190 |
| conv2 (5x5) | 16@186x86 |
| pool (2x2) | 16@93x93 |
| conv3 (3x3) | 16@91x91 |
| pool (2x2) | 16@45x45 |
| conv4 (3x3) | 16@43x43 |
| pool (2x2) | 16@21x21 |
| fc1 | 128 |
| fc2 | 84 |
| fc3 | 11 |



* Problems & solutions :
* Experiment setup :
* Results :
* Analysis :

## Different configurations of using ResNet

We also tried the ResNet18 model and change the learning rate to the initial 0.1 decayed by 10 every 30 epoches.Batch size is 32.

Network Source Code:

*def conv3x3(in\_planes, out\_planes, stride=1):*

*return nn.Conv2d(in\_planes, out\_planes, kernel\_size=3, stride=stride,*

*padding=1, bias=False)*

*class BasicBlock(nn.Module):*

*expansion = 1*

*def \_\_init\_\_(self, inplanes, planes, stride=1, downsample=None):*

*super(BasicBlock, self).\_\_init\_\_()*

*self.conv1 = conv3x3(inplanes, planes, stride)*

*self.bn1 = nn.BatchNorm2d(planes)*

*self.relu = nn.ReLU(inplace=True)*

*self.conv2 = conv3x3(planes, planes)*

*self.bn2 = nn.BatchNorm2d(planes)*

*self.downsample = downsample*

*self.stride = stride*

*def forward(self, x):*

*residual = x*

*out = self.conv1(x)*

*out = self.bn1(out)*

*out = self.relu(out)*

*out = self.conv2(out)*

*out = self.bn2(out)*

*if self.downsample is not None:*

*residual = self.downsample(x)*

*out += residual*

*out = self.relu(out)*

*return out*

*class Bottleneck(nn.Module):*

*expansion = 4*

*def \_\_init\_\_(self, inplanes, planes, stride=1, downsample=None):*

*super(Bottleneck, self).\_\_init\_\_()*

*self.conv1 = nn.Conv2d(inplanes, planes, kernel\_size=1, bias=False)*

*self.bn1 = nn.BatchNorm2d(planes)*

*self.conv2 = nn.Conv2d(planes, planes, kernel\_size=3, stride=stride,*

*padding=1, bias=False)*

*self.bn2 = nn.BatchNorm2d(planes)*

*self.conv3 = nn.Conv2d(planes, planes \* self.expansion, kernel\_size=1, bias=False)*

*self.bn3 = nn.BatchNorm2d(planes \* self.expansion)*

*self.relu = nn.ReLU(inplace=True)*

*self.downsample = downsample*

*self.stride = stride*

*def forward(self, x):*

*residual = x*

*out = self.conv1(x)*

*out = self.bn1(out)*

*out = self.relu(out)*

*out = self.conv2(out)*

*out = self.bn2(out)*

*out = self.relu(out)*

*out = self.conv3(out)*

*out = self.bn3(out)*

*if self.downsample is not None:*

*residual = self.downsample(x)*

*out += residual*

*out = self.relu(out)*

*return out*

*class ResNet(nn.Module):*

*def \_\_init\_\_(self, block, layers, num\_classes=1000):*

*self.inplanes = 64*

*super(ResNet, self).\_\_init\_\_()*

*self.conv1 = nn.Conv2d(3, 64, kernel\_size=7, stride=2, padding=3,*

*bias=False)*

*self.bn1 = nn.BatchNorm2d(64)*

*self.relu = nn.ReLU(inplace=True)*

*self.maxpool = nn.MaxPool2d(kernel\_size=3, stride=2, padding=1)*

*self.layer1 = self.\_make\_layer(block, 64, layers[0])*

*self.layer2 = self.\_make\_layer(block, 128, layers[1], stride=2)*

*self.layer3 = self.\_make\_layer(block, 256, layers[2], stride=2)*

*self.layer4 = self.\_make\_layer(block, 512, layers[3], stride=2)*

*self.avgpool = nn.AvgPool2d(7, stride=1)*

*self.fc = nn.Linear(512 \* block.expansion, num\_classes)*

*for m in self.modules():*

*if isinstance(m, nn.Conv2d):*

*nn.init.kaiming\_normal\_(m.weight, mode='fan\_out', nonlinearity='relu')*

*elif isinstance(m, nn.BatchNorm2d):*

*nn.init.constant\_(m.weight, 1)*

*nn.init.constant\_(m.bias, 0)*

*def \_make\_layer(self, block, planes, blocks, stride=1):*

*downsample = None*

*if stride != 1 or self.inplanes != planes \* block.expansion:*

*downsample = nn.Sequential(*

*nn.Conv2d(self.inplanes, planes \* block.expansion,*

*kernel\_size=1, stride=stride, bias=False),*

*nn.BatchNorm2d(planes \* block.expansion),*

*)*

*layers = []*

*layers.append(block(self.inplanes, planes, stride, downsample))*

*self.inplanes = planes \* block.expansion*

*for i in range(1, blocks):*

*layers.append(block(self.inplanes, planes))*

*return nn.Sequential(\*layers)*

*def forward(self, x):*

*x = self.conv1(x)*

*x = self.bn1(x)*

*x = self.relu(x)*

*x = self.maxpool(x)*

*x = self.layer1(x)*

*x = self.layer2(x)*

*x = self.layer3(x)*

*x = self.layer4(x)*

*x = self.avgpool(x)*

*x = x.view(x.size(0), -1)*

*x = self.fc(x)*

*return x*

*def resnet18(\*\*kwargs):*

*model = ResNet(BasicBlock, [2, 2, 2, 2], \*\*kwargs)*

*return model*

And the model structure is:

*ResNet(  
 (conv1): Conv2d(3, 64, kernel\_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)  
 (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (maxpool): MaxPool2d(kernel\_size=3, stride=2, padding=1, dilation=1, ceil\_mode=False)  
 (layer1): Sequential(  
 (0): BasicBlock(  
 (conv1): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (conv2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 )  
 (1): BasicBlock(  
 (conv1): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (conv2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 )  
 )  
 (layer2): Sequential(  
 (0): BasicBlock(  
 (conv1): Conv2d(64, 128, kernel\_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)  
 (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (conv2): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (downsample): Sequential(  
 (0): Conv2d(64, 128, kernel\_size=(1, 1), stride=(2, 2), bias=False)  
 (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 )  
 )  
 (1): BasicBlock(  
 (conv1): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (conv2): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 )  
 )  
 (layer3): Sequential(  
 (0): BasicBlock(  
 (conv1): Conv2d(128, 256, kernel\_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)  
 (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (conv2): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (downsample): Sequential(  
 (0): Conv2d(128, 256, kernel\_size=(1, 1), stride=(2, 2), bias=False)  
 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 )  
 )  
 (1): BasicBlock(  
 (conv1): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (conv2): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 )  
 )  
 (layer4): Sequential(  
 (0): BasicBlock(  
 (conv1): Conv2d(256, 512, kernel\_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)  
 (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (conv2): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (downsample): Sequential(  
 (0): Conv2d(256, 512, kernel\_size=(1, 1), stride=(2, 2), bias=False)  
 (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 )  
 )  
 (1): BasicBlock(  
 (conv1): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (relu): ReLU(inplace)  
 (conv2): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 )  
 )  
 (avgpool): AvgPool2d(kernel\_size=7, stride=1, padding=0)  
 (fc): Linear(in\_features=512, out\_features=11, bias=True)  
)*

We use the to adjust the learning rate:

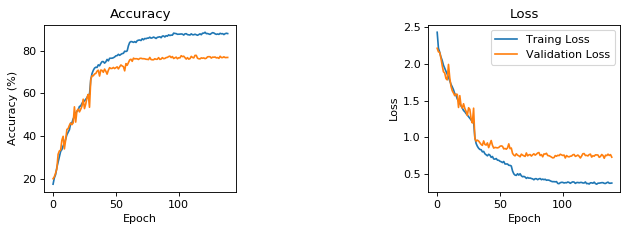
*def adjust\_learning\_rate(optimizer, epoch):*

*# Sets the learning rate to the initial LR decayed by 10 every 30 epochs*

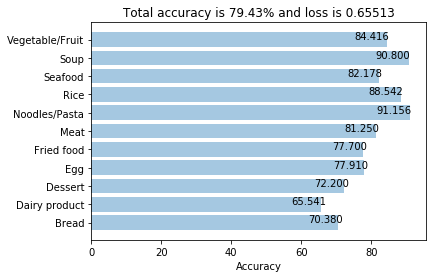
*lr = LR \* (0.1 \*\* (epoch // 30))*

*for param\_group in optimizer.param\_groups:*

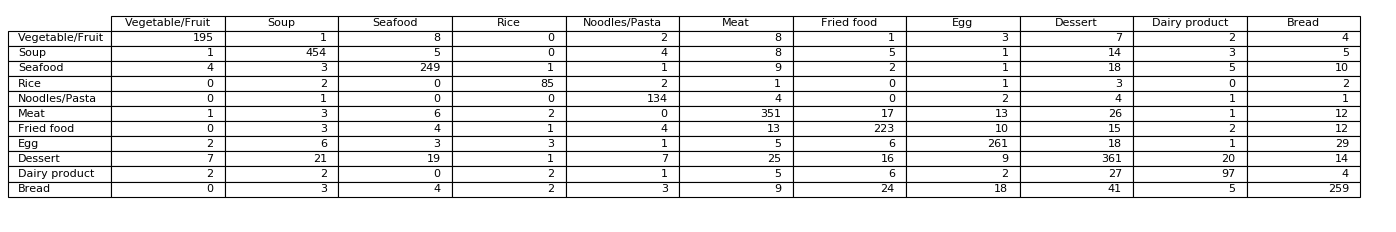
*param\_group['lr'] = lr*

After 140 epoches,the training accuracy and loss, validation accuracy and loss are 87.9% and 0.382, 76.8% and 0.732, respectively. We can see the overfitting is occured after 70 epoches.  
 

Using this model and do the evaluation. The evaluation accuracy is 79.4% and evaluation loss is 0.655. For each class in Food-11:  
 Accuracy of Vegetable/Fruit : 84.42%  
 Accuracy of Soup : 90.80%  
 Accuracy of Seafood : 82.18%  
 Accuracy of Rice : 88.54%  
 Accuracy of Noodles/Pasta : 91.16%  
 Accuracy of Meat : 81.25%  
 Accuracy of Fried food : 77.70%  
 Accuracy of Egg : 77.91%  
 Accuracy of Dessert : 72.20%  
 Accuracy of Dairy product : 65.54%  
 Accuracy of Bread : 70.38%



Accuracy cross analysis is shown below. Y-axis is the really label, and X-axis is the predicted label. Dairy product has the worse accuracy, most of the error occur in desert predictions.



We use model.state\_dict() to list all the sublayer and grep all the weights and bias:

*def show\_distribution(model):*

*params=model.state\_dict()*

*even = True*

*plt.subplots\_adjust(wspace =1, hspace =1)*

*for k,v in params.items():*

*if k.split('.')[-1] == 'weight' or k.split('.')[-1] == 'bias':*

*if even:*

*fig, (ax1, ax2) = plt.subplots(1,2,figsize=(9,3), dpi=80)*

*ax = ax1*

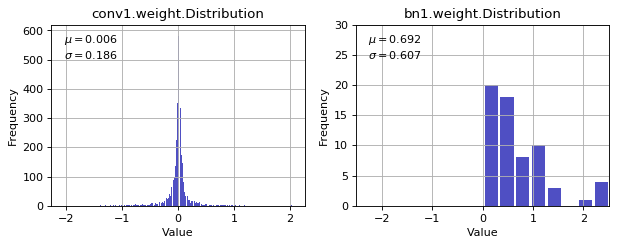
*else:*

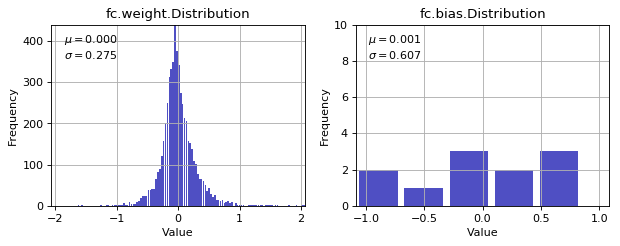
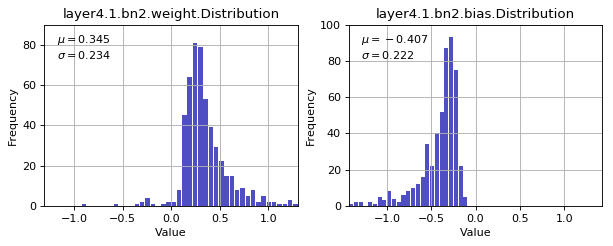
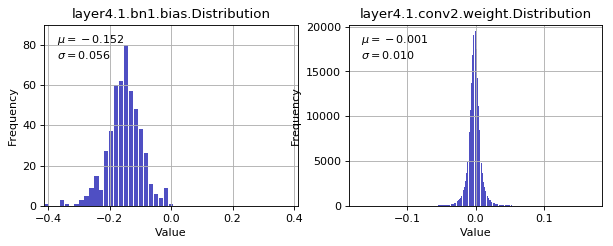
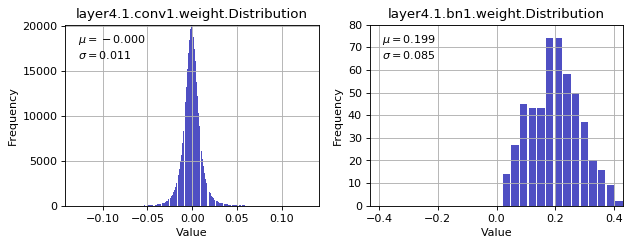
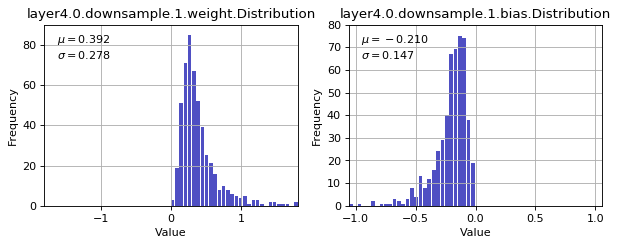
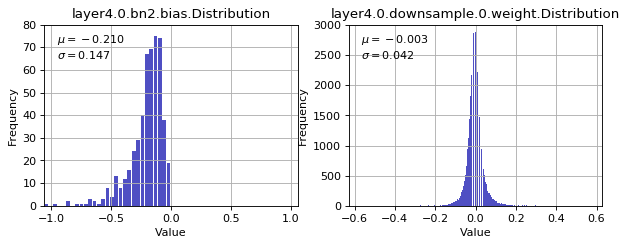
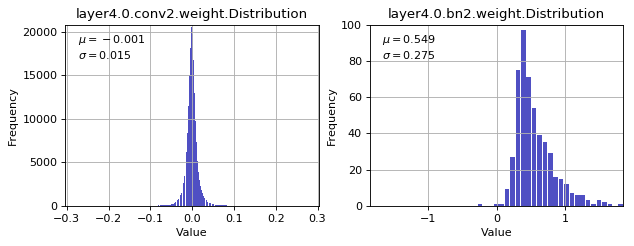
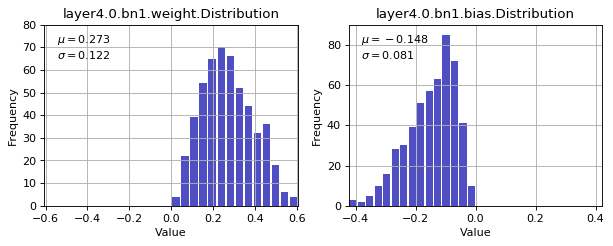
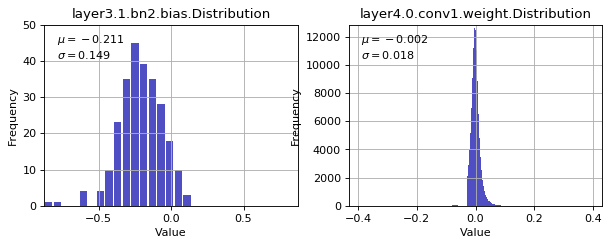
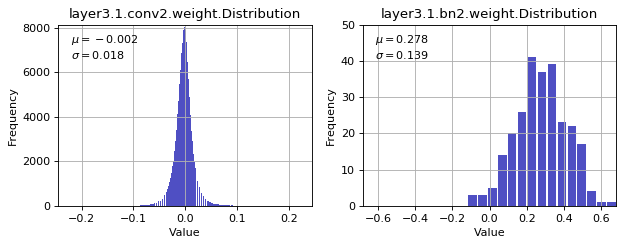
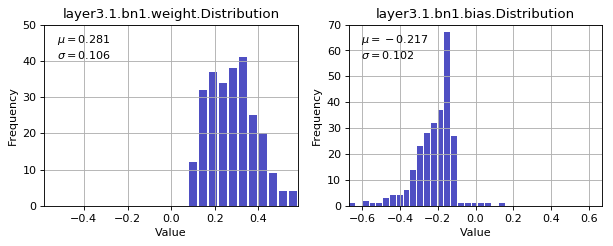
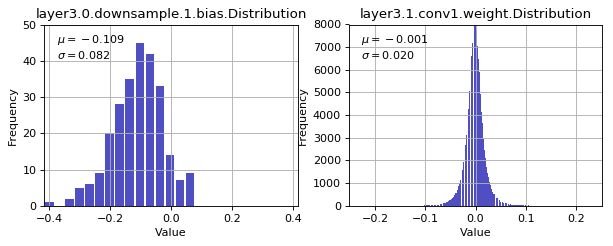
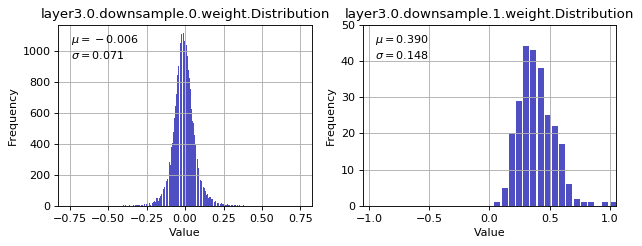
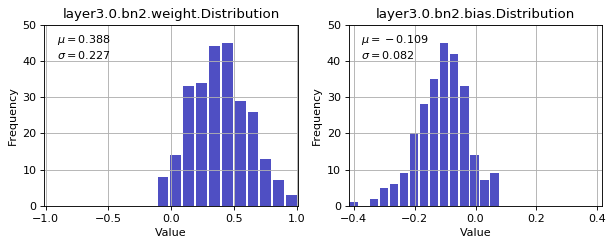
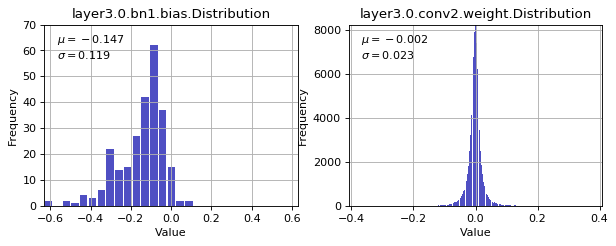
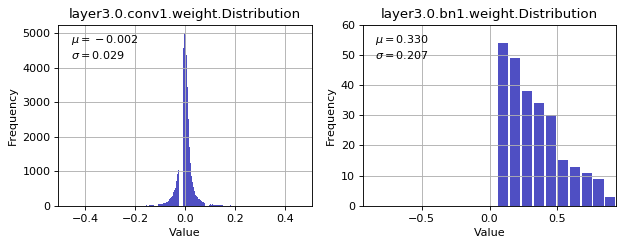
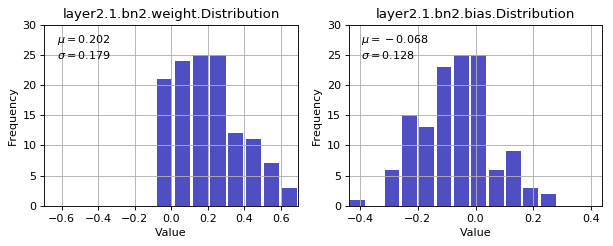
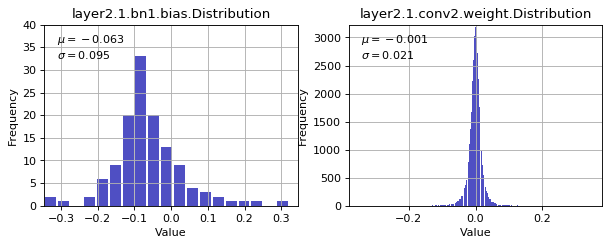
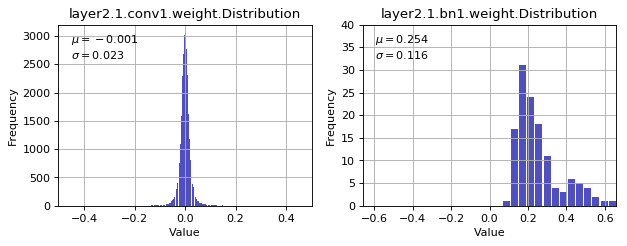
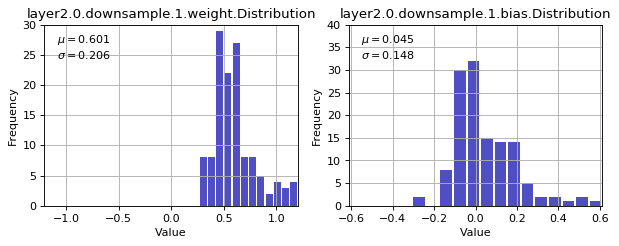
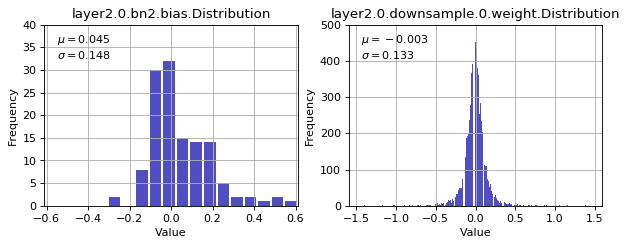
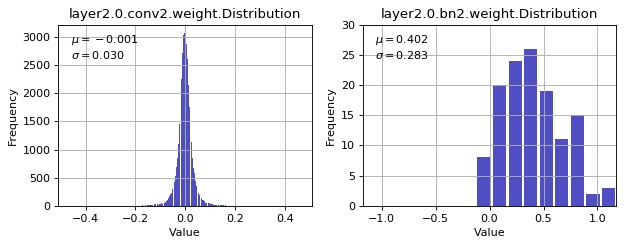
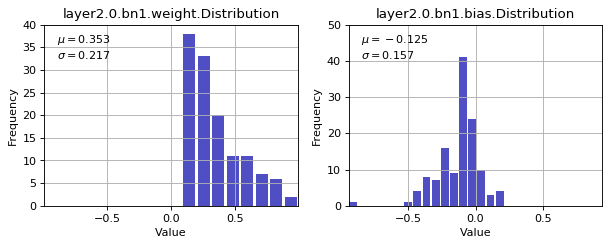
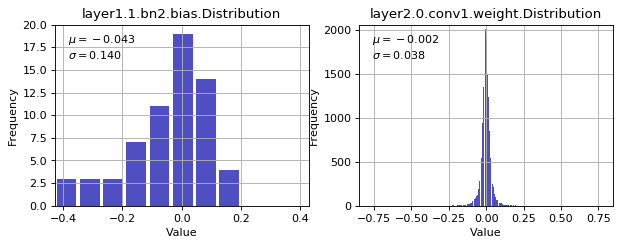
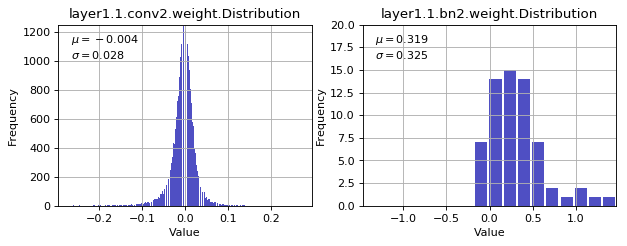
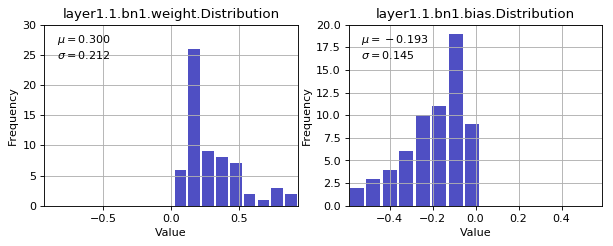
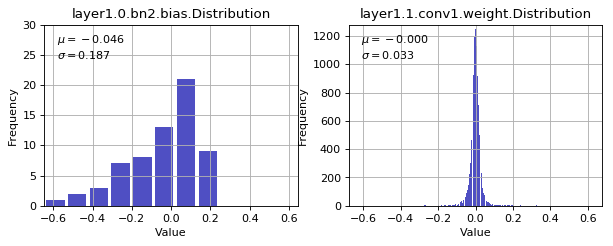
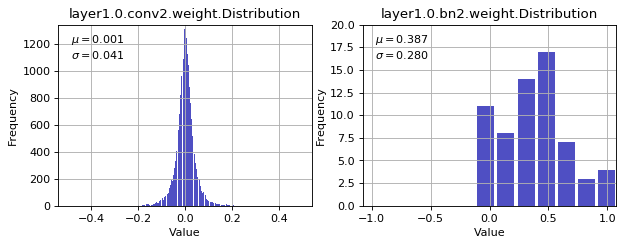
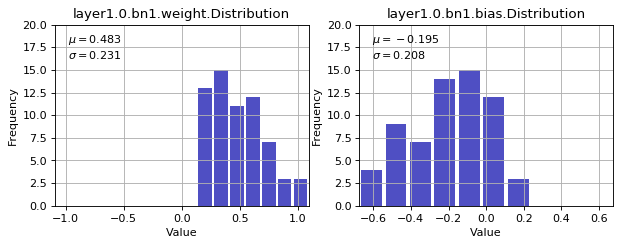
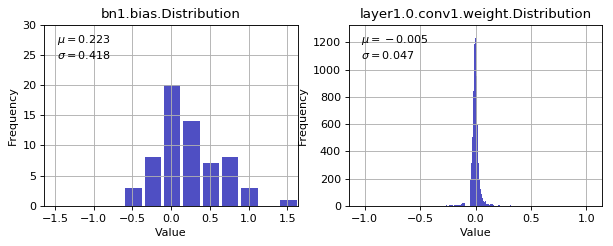
*ax = ax2*

*title = k + '.Distribution'*

*draw\_distribution(ax, v.data.cpu(), title)*

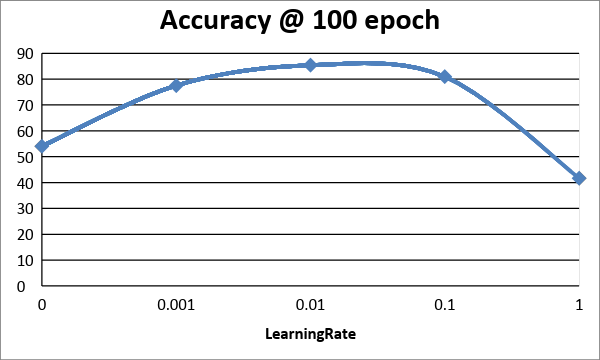
*even = not even*



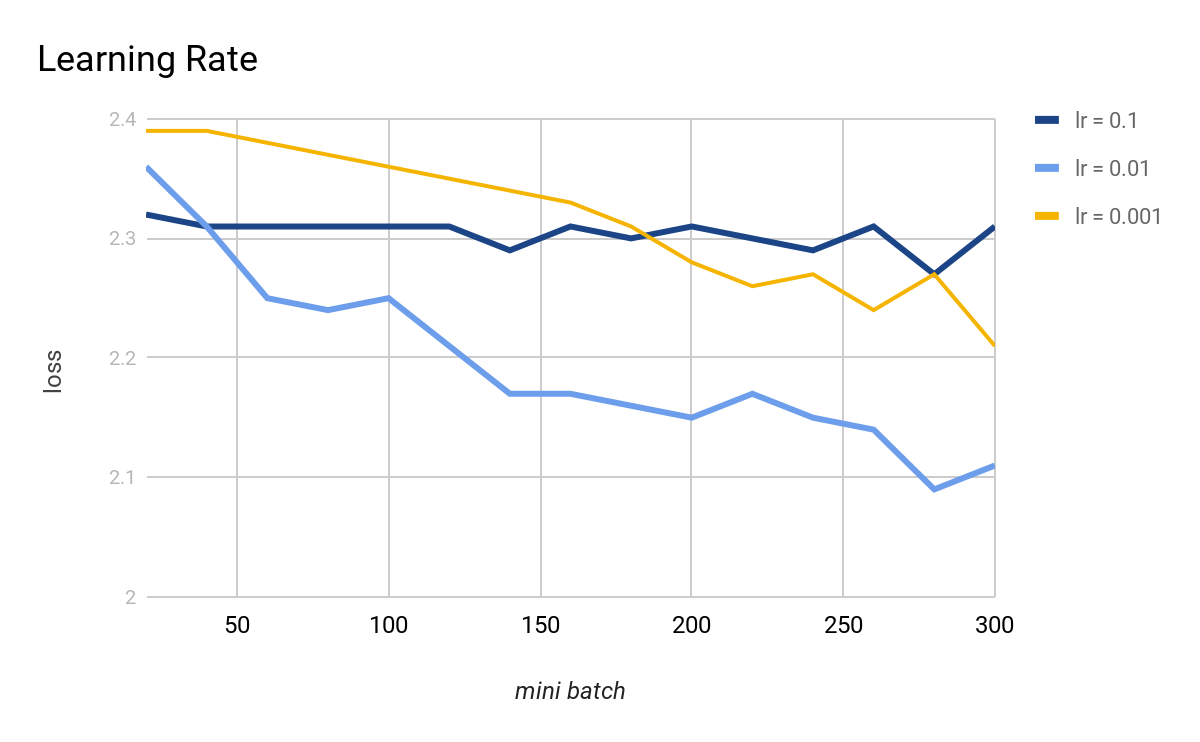


# LAB 1-3 : Tuning hyper-parameters

I do some experiments. Fix model and change learning from 0.0001 to 1 with SGD optimizers. After 100 epoch, we can see that learning rate = 0.01 has the best accuracy, it also means it convergence fastest. Thus, we choose learning rate = 0.01.



If we look into the loss in each mini batch while in the first epoch, we can observe the change rate of loss as below. The loss almost doesn't change while learning rate equals 0.1. And we can see the loss in the learning rate of 0.01 decreases quicker than which in learning rate of 0.001. Thus, we can choose 0.01 as initial learning rate.



There are several learning rate algorithms.

StepLR:

Assuming initial learning rate = 0.01, StepLR(optimizer, step\_size=30, gamma=0.1) will set

lr = 0.01 if epoch < 30  
lr = 0.001 if 30 <= epoch < 60  
lr = 0.0001 if 60 <= epoch < 90  
…

MultiStepLR:

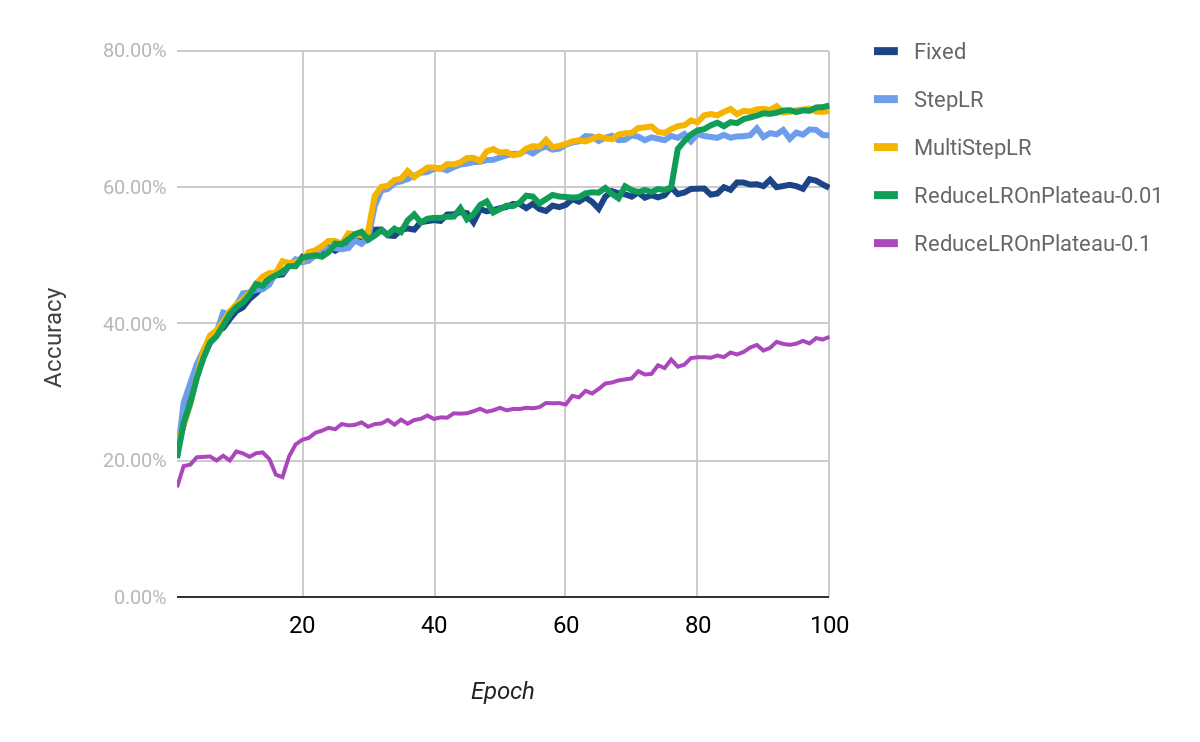
Assuming initial learning rate = 0.01, MultiStepLR(optimizer, milestones=[30,80], gamma=0.1) will set

lr = 0.01 if epoch < 30  
lr = 0.001 if 30 <= epoch < 80  
lr = 0.0001 if epoch >= 80  
…

ReduceLROnPlateau:

Assuming initial learning rate = 0.01, ReduceLROnPlateau(optimizer, mode='min', patience=6, verbose=True) will reduce learning rate when a metric has stopped improving.

We tested three configuration of learning rate will the same initial learning rate of 0.01: fixed, StepLR, MultiStepLR, and ReduceLROnPlateau with the parameter mentioned above. The learning accuracy are shown as below.



We tested ReduceLROnPlateau with initial learning rate of 0.01 and 0.1 with legend of ReduceLROnPlateau-0.01 and ReduceLROnPlateau-0.1 respectively. In ReduceLROnPlateau-0.01, we got a messages of:

69 epoch, training accuracy: 60.08%, loss = 0.0372  
70 epoch, training accuracy: 59.53%, loss = 0.0379  
71 epoch, training accuracy: 59.23%, loss = 0.0381  
72 epoch, training accuracy: 59.54%, loss = 0.0373  
73 epoch, training accuracy: 59.25%, loss = 0.0376  
74 epoch, training accuracy: 59.69%, loss = 0.0378  
75 epoch, training accuracy: 59.55%, loss = 0.0372  
Epoch 75: reducing learning rate of group 0 to 1.0000e-03.  
76 epoch, training accuracy: 59.93%, loss = 0.0373  
77 epoch, training accuracy: 65.55%, loss = 0.0321  
78 epoch, training accuracy: 66.70%, loss = 0.0306

The system reduced learning rate by a factor of 0.1 (default) after 6 epochs (the parameter patience=6) without improvement.

In ReduceLROnPlateau-0.1, we got a messages of:

11 epoch, training accuracy: 21.00%, loss = 0.0689  
12 epoch, training accuracy: 20.51%, loss = 0.0691  
13 epoch, training accuracy: 21.03%, loss = 0.0693  
14 epoch, training accuracy: 21.14%, loss = 0.0690  
15 epoch, training accuracy: 20.20%, loss = 0.0694  
16 epoch, training accuracy: 17.90%, loss = 0.0705  
17 epoch, training accuracy: 17.53%, loss = 0.0711  
Epoch 17: reducing learning rate of group 0 to 1.0000e-02.

18 epoch, training accuracy: 20.55%, loss = 0.0693  
19 epoch, training accuracy: 22.32%, loss = 0.0678

Further, ReduceLROnPlateau-0.01 and ReduceLROnPlateau-0.1 show the importance of choosing a good initial learning rate by observing ReduceLROnPlateau-0.1’s poor performance .

* Problems & solutions :
* Experiment setup :
* Results :
* Analysis :

# Bonus: Doing normalization according to mean and standard deviation of Food-11 dataset

In Lab 1-2, we normalize the data set according to its mean and standard deviation. Line 4~20, we load the data set without normalization.

Line 21~24, we calculate the mean and standard deviation of training and test data set.

Line 29~43, we load the data sets again and normalize them by the corresponding calculated mean and standard deviation.

|  |  |
| --- | --- |
| *1* | *calculate\_mean\_std = True* |
| *2* |  |
| *3* | *if calculate\_mean\_std == True:* |
| *4* | *#The transform function for train data* |
| *5* | *transform\_train = transforms.Compose([* |
| *6* | *transforms.Resize(img\_size),* |
| *7* | *transforms.RandomCrop(img\_size, padding=4),* |
| *8* | *transforms.ToTensor(),* |
| *9* | *])* |
| *10* |  |
| *11* | *#The transform function for test data* |
| *12* | *transform\_test = transforms.Compose([* |
| *13* | *transforms.Resize(img\_size),* |
| *14* | *transforms.RandomCrop(img\_size, padding=4),* |
| *15* | *transforms.ToTensor(),* |
| *16* | *])* |
| *17* |  |
| *18* | *#we will calculate mean and std* |
| *19* | *trainset = torchvision.datasets.ImageFolder(root='./food/training', transform=transform\_train)* |
| *20* | *testset = torchvision.datasets.ImageFolder(root='./food/evaluation', transform=transform\_test)* |
| *21* | *train\_mean, train\_std = get\_mean\_and\_std(trainset)* |
| *22* | *print(train\_mean, train\_std)* |
| *23* | *test\_mean, test\_std = get\_mean\_and\_std(testset)* |
| *24* | *print(test\_mean, test\_std)* |
| *25* | *else:* |
| *26* | *train\_mean, train\_std = ([0.5551, 0.4478, 0.3366]), ([0.2337, 0.2414, 0.2386])* |
| *27* | *test\_mean, test\_std = ([0.5607, 0.4518, 0.3425]), ([0.2333, 0.2415, 0.2385])* |
| *28* |  |
| *29* | *transform\_train = transforms.Compose([* |
| *30* | *transforms.Resize(img\_size),* |
| *31* | *transforms.RandomCrop(img\_size, padding=4),* |
| *32* | *transforms.RandomHorizontalFlip(),* |
| *33* | *transforms.ToTensor(),* |
| *34* | *transforms.Normalize(train\_mean, train\_std)* |
| *35* | *])* |
| *36* |  |
| *37* | *#The transform function for test data* |
| *38* | *transform\_test = transforms.Compose([* |
| *39* | *transforms.Resize(img\_size),* |
| *40* | *transforms.RandomCrop(img\_size, padding=4),* |
| *41* | *transforms.ToTensor(),* |
| *42* | *transforms.Normalize(test\_mean, test\_std)* |
| *43* | *])* |

As you can see, resize the input images into different size can have different mean and standard deviation. Here is the outcome.

Training dataset

|  |  |  |
| --- | --- | --- |
| target image size | mean | standard deviation |
| 64x64 | [0.5282, 0.4251, 0.3177] | [0.2497, 0.2444, 0.2318] |
| 128x128 | [0.5441, 0.4383, 0.3286] | [0.2498, 0.2445, 0.2319] |
| 256x256 | [0.5522, 0.4454, 0.3345] | [0.2353, 0.2415, 0.2375] |
| 384x384 | [0.5549, 0.4476, 0.3363] | [0.2335, 0.2413, 0.2383] |
| 512x512 | [0.5559, 0.4484, 0.3371] | [0.2328, 0.2413, 0.2389] |
| 1024x1024 | [0.5582, 0.4505, 0.3388] | [0.2301, 0.2397, 0.2379] |

Validation dataset

|  |  |  |
| --- | --- | --- |
| target image size | mean | standard deviation |
| 64x64 | [0.5292, 0.4256, 0.3189] | [0.2494, 0.2442, 0.2316] |
| 128x128 | [0.5455, 0.4394, 0.3303] | [0.2400, 0.2422, 0.2351] |
| 256x256 | [0.5536, 0.4464, 0.3361] | [0.2348, 0.2411, 0.2370] |
| 384x384 | [0.5560, 0.4484, 0.3377] | [0.2333, 0.2411, 0.2380] |
| 512x512 | [0.5571, 0.4493, 0.3385] | [0.2324, 0.2410, 0.2384] |
| 1024x1024 | [0.5595, 0.4514, 0.3403] | [0.2297, 0.2394, 0.2375] |

Evaluation dataset

|  |  |  |
| --- | --- | --- |
| target image size | mean | standard deviation |
| 64x64 | [0.5334, 0.4288, 0.3231] | [0.2499, 0.2447, 0.2324] |
| 128x128 | [0.5501, 0.4431, 0.3353] | [0.2404, 0.2428, 0.2362] |
| 256x256 | [0.5581, 0.4498, 0.3409] | [0.2348, 0.2415, 0.2376] |
| 384x384 | [0.5610, 0.4522, 0.3428] | [0.2333, 0.2415, 0.2389] |
| 512x512 | [0.5616, 0.4528, 0.3432] | [0.2326, 0.2417, 0.2395] |
| 1024x1024 | [0.5643, 0.4551, 0.3454] | [0.2297, 0.2400, 0.2385] |