# COMP4211 Project 1: Face Mask Detection

## Group members:

1. ZHU, Chen
2. Liu Pak Wai

## Internal deadline:

1. code part by 21/4
2. presentation by 23/4
3. report [26/4]

## Environment:

1. Colab GPU resources
2. pytorch

## Description of Dataset:

Choose the dataset ‘Face Mask Detection’

Link to dataset: https://www.kaggle.com/andrewmvd/face-mask-detection

The dataset contains two folders: ‘annotations’ folder storing xml files and ‘images’ folder storing images. Each xml file is corresponding to one image. Within each xml file, there are:

1. image name
2. image size
3. the information of each face in the image including:
    a. coordinates of the face
    b. label of the face: ‘with_mask’, ‘without_mask’ or ‘mask_weared_incorrect’

The useful information we obtain is the image name and information of each face in the image. We cut the image to obtain all parts containing one face only based on the coordinates. They are used to build the dataset for later work. The dataset contains:

1. 3232 examples wearing a mask correctly
2. 123 examples wearing a mask incorrectly
3. 717 examples without wearing a mask

Obviously this is an imbalanced dataset. So we need to preprocess the data.

## Data Preprocessing:

1. Get image names, coordinates locating different faces in one image and label of each face from the xml files
2. Obtain parts of images containing only one face based on the coordinates we got previously.
3. Perform oversampling to increase the number of examples wearing a mask incorrectly and the number of examples not wearing a mask to be the same as the number of examples wearing a mask correctly.

## Machine Learning Tasks:

Build a model to judge whether the person in the input image is wearing the mask correctly, wearing the mask incorrectly or not wearing a mask. Three classes classification problems, where with mask = 0, without mask = 1, incorrect = 2.

## Machine learning Method

| | |
|---|---|
| **Model selection** | CNN Model<br>developing possible convolutional layer models to make predictions. |
| **Model Training** | Using our customized training function in the source code. |

## Experiments:

1. Experiments on keeping or not keeping the information of color of input image

Using the baseline model.

Model Structure:

input → convolutional layer → average pooling layer →
convolutional layer → average pooling layer →
fully connected layer → fully connected layer →
fully connected layer → output

The only modification of two models in this experiment is the number of input channels of the first convolutional layer:

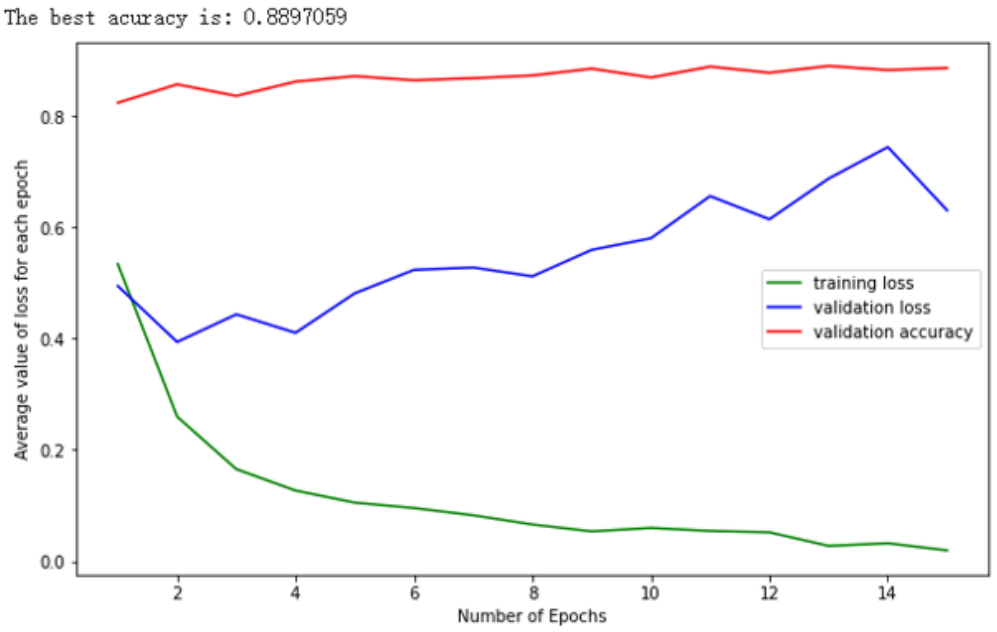
case a: without keeping the information of color, number of input channels is 1.

case b: keep the information of color, number of input channels is 3.

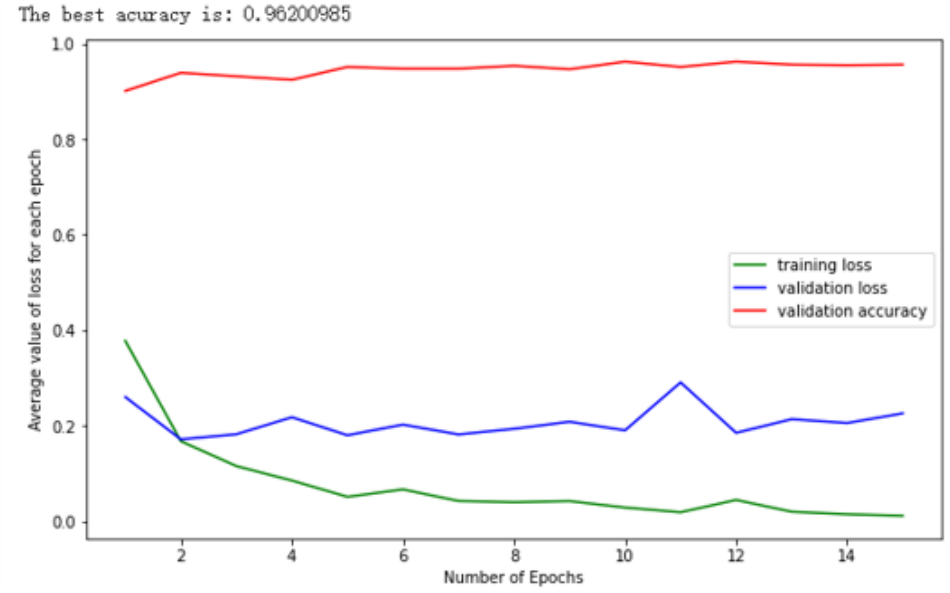
Train 15 epoches for each experiment
Using Adam optimizer and nn.CrossEntropyLoss criterion.

Result:

Case a. without keeping the information of color



Case b. keep the information of color




The accuracy of case b is much higher than case a. However, in the dataset almost all the masks are blue, black or white, if our model is based on color to make the judgement, it will not be able to detect masks with other colors. Also it will give the wrong result if there is something blue, black or white covering peoples' faces. So the model should not rely on the information of color.

2. Experiments on different settings of size in torchvision.transforms.Resize()

Choose three settings for experiment: 32 * 32, 64 * 64, 128 * 128

Train 15 epoches for each model
Using Adam optimizer and nn.CrossEntropyLoss criterion.

First Comparison:

Compare the performance of the baseline model under three settings.

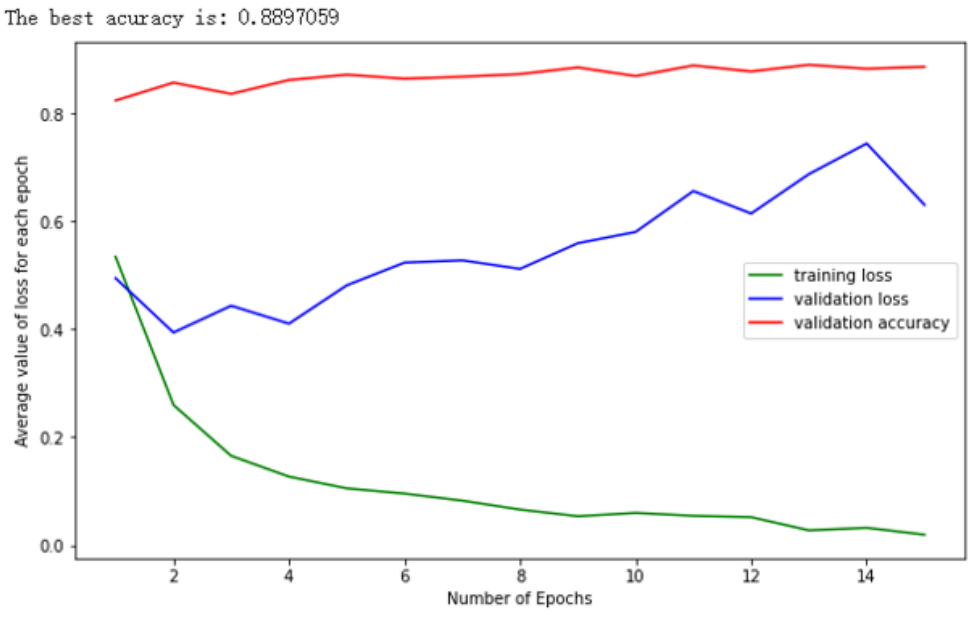
Model Structure:

input → convolutional layer → average pooling layer →
convolutional layer → max pooling layer →
fully connected layer → fully connected layer →
fully connected layer → output

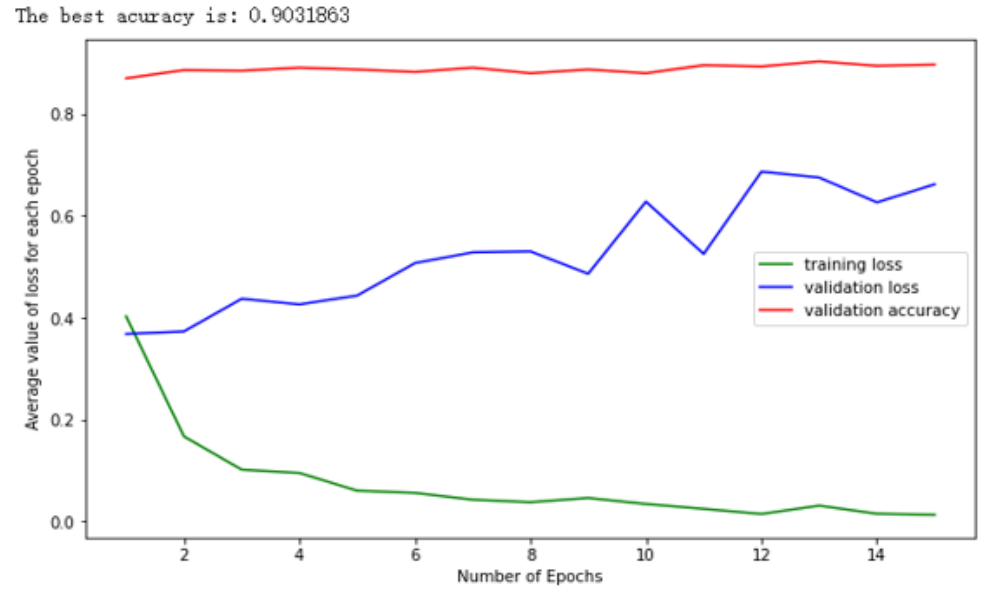
The only modification of three models in this comparison is the input size of the first fully connected layer since we have different input sizes for each model (32 * 32, 64 * 64, 128 * 128).

Result:

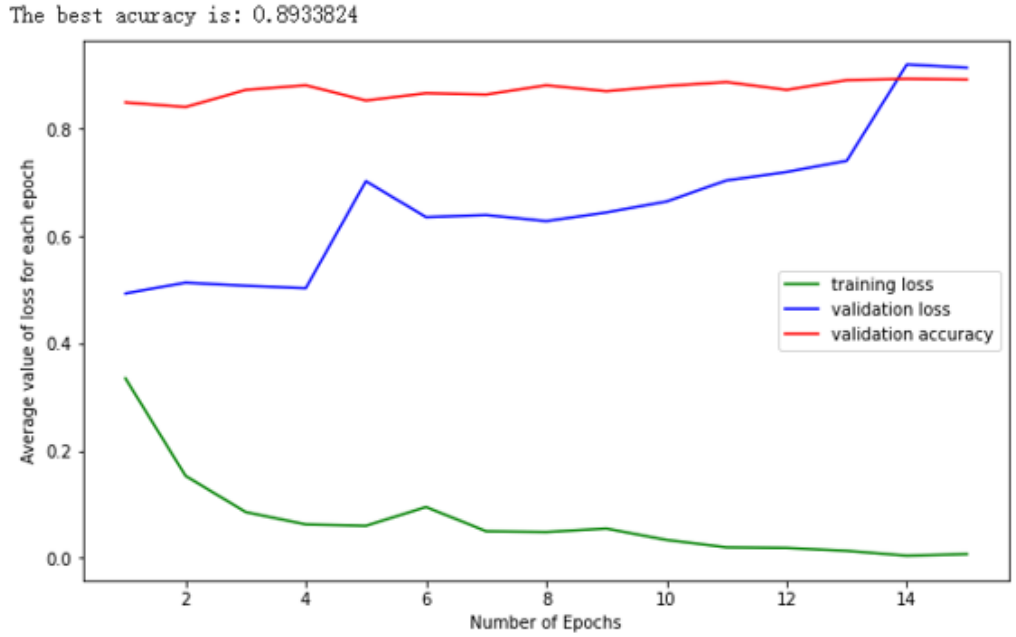
Setting 32 * 32



Setting 64 * 64




Setting 128 * 128




The setting 64 * 64 have the highest accuracy, followed by setting 128 * 128, the setting 32 * 32 have lowest accuracy.

Second Comparison:

Compare the performance of the modified model 1 under three settings.

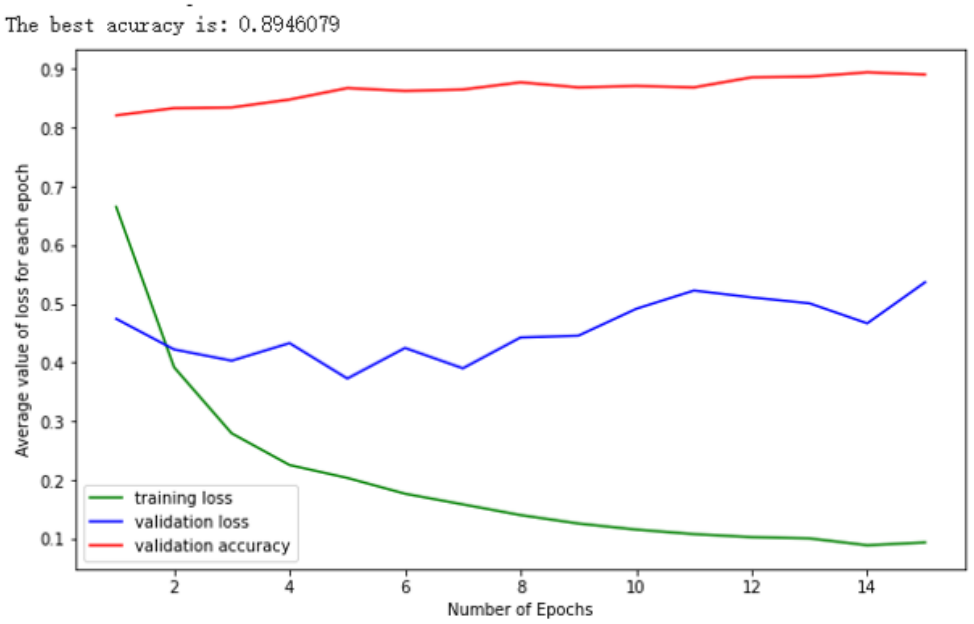
Model Structure:

input → convolutional layer → average pooling layer →
convolutional layer → max pooling layer →
fully connected layer → dropout layer →
fully connected layer → fully connected layer → output

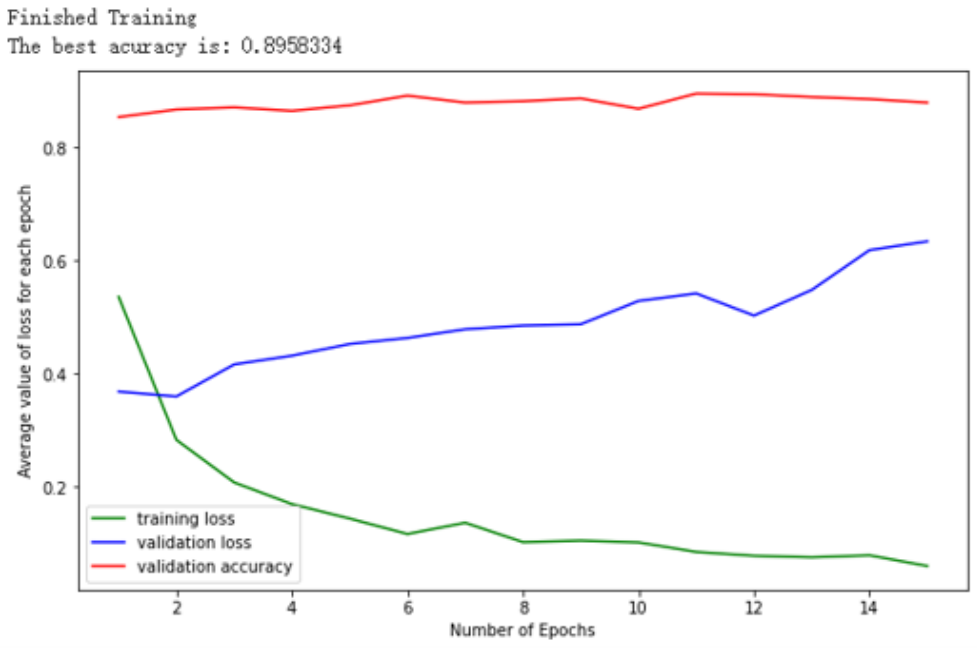
The modification of three models in this comparison is the input size of the first fully connected layer since we have different input sizes for each model (32 * 32, 64 * 64, 128 * 128).

Result:

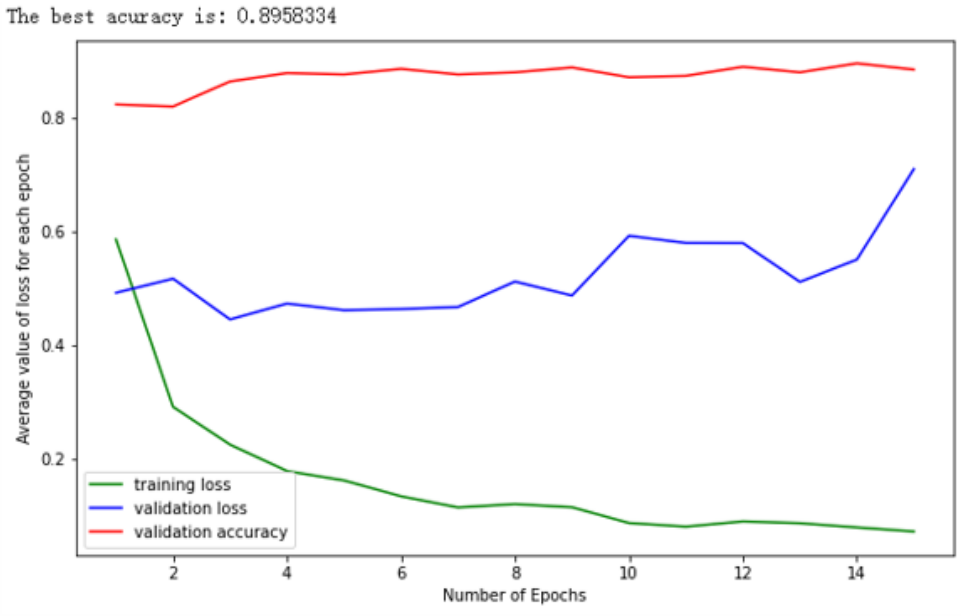
Setting 32 * 32




Setting 64 * 64



Setting 128 * 128




the setting 64 * 64 and setting 128 * 128 have the same accuracy, but the validation loss of setting 64 * 64 in first few epoches is lower than the setting 128 * 128, the setting 32 * 32 have lowest accuracy.

Third Comparison:

Compare the performance of the modified model 2 under three settings.

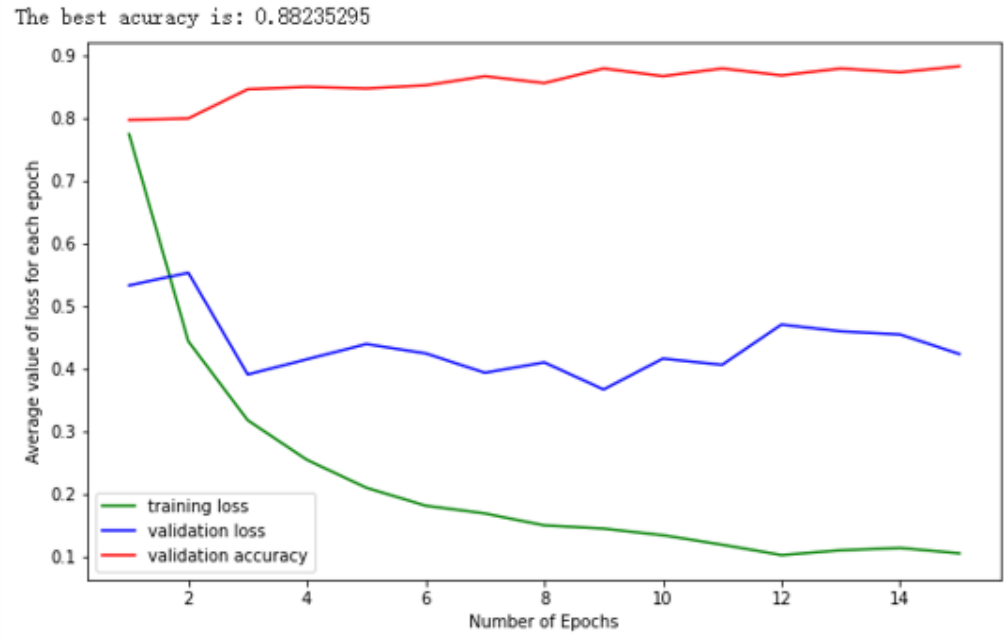
Model Structure:

input → convolutional layer → average pooling layer →
convolutional layer → average pooling layer →
convolutional layer → max pooling layer →
fully connected layer → dropout layer →
fully connected layer → fully connected layer → output

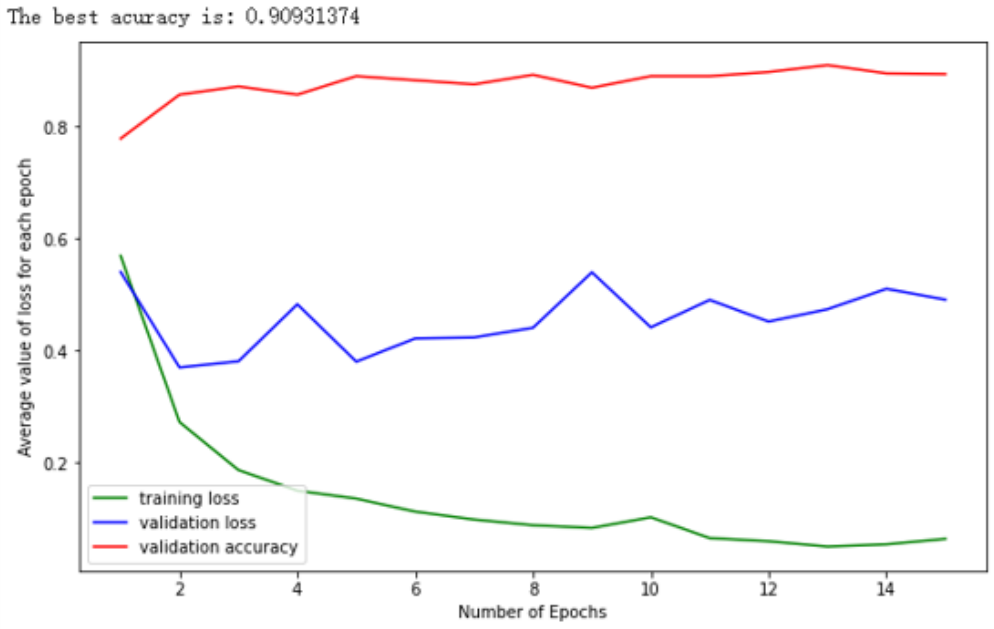
The modification of three models in this comparison is the input size of the first fully connected layer since we have different input sizes for each model (32 * 32, 64 * 64, 128 * 128).

Result:

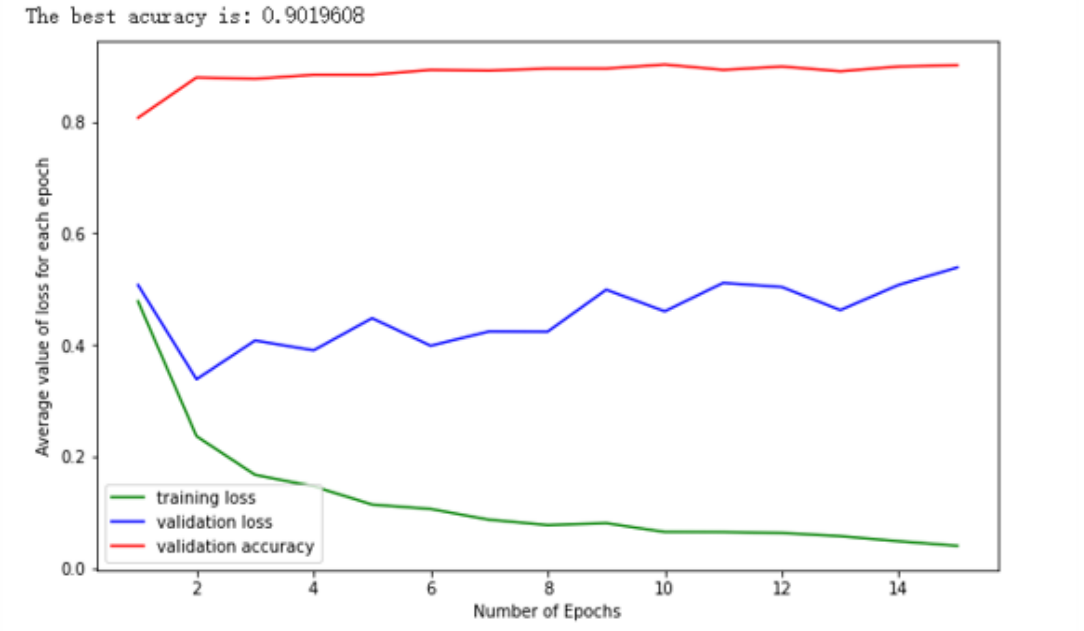
Setting 32 * 32




Setting 64 * 64



Setting 128 * 128




the setting 64 * 64 have the highest accuracy, followed by setting 128 * 128, the setting 32 * 32 have lowest accuracy.

In all three comparisons, setting 64 * 64 is performing the best or as good as the rest in terms of validation accuracy. Resize to 64 * 64 indeed improves our models' performance.

3. Experiment on more complex model when resize to 128 * 128

Model 1:

Model Structure:

input → convolutional layer → average pooling layer →
convolutional layer → average pooling layer →
convolutional layer → max pooling layer →
fully connected layer → dropout layer →
fully connected layer → fully connected layer → output

Model 2:

Model Structure:

input → convolutional layer → average pooling layer →
convolutional layer → average pooling layer →
convolutional layer → max pooling layer →
convolutional layer → max pooling layer →
fully connected layer → dropout layer →
fully connected layer → fully connected layer → output

Model 3:

Model Structure:

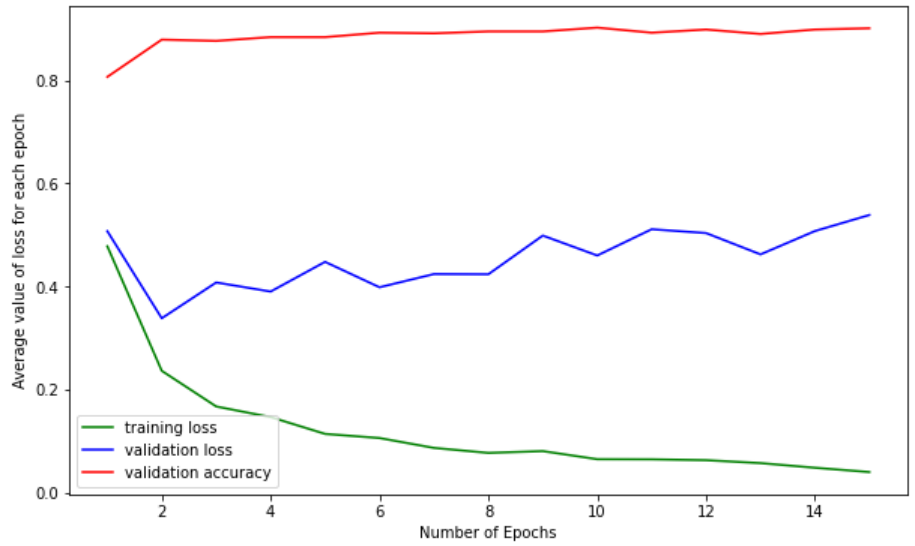
input → convolutional layer → average pooling layer →
convolutional layer →
convolutional layer → average pooling layer →
convolutional layer → max pooling layer →
convolutional layer → max pooling layer →
fully connected layer → dropout layer →
fully connected layer → fully connected layer → output

Train 15 epoches for each model
Using Adam optimizer and nn.CrossEntropyLoss criterion.

Result:

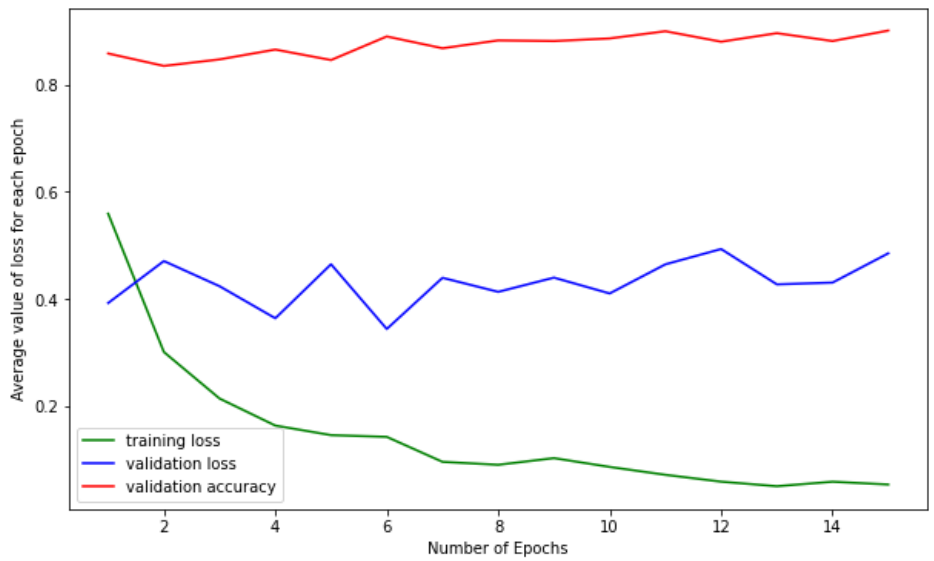
Model 1

The best acuracy is: 0.9019608

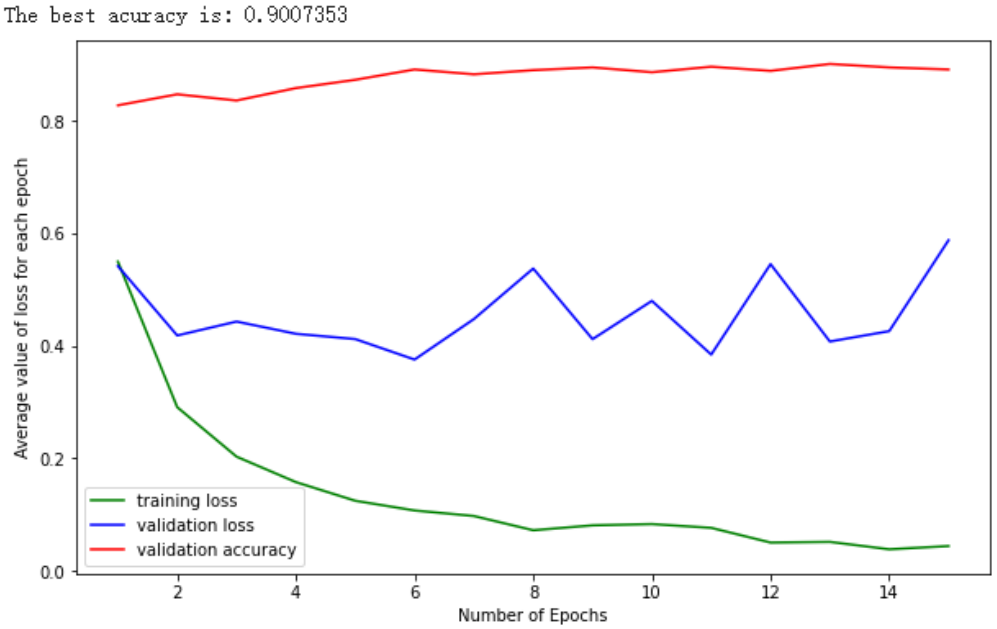



Model 2

The best acuracy is: 0.9019608



Model 3




Model 2 is in fact adding one more convolutional layer and one more max pooling layer based on Model 1, the best validation accuracy obtained is exactly the same and the validation loss curves are quite close to each other, which implies that adding one more convolutional layer and one more max pooling layer is neither improve or reduce the performance. An interesting experiment is comparing performance of Model 1 and Model 3. Model 3 is trying to replace one convolutional layer with two based on Model 1. The best validation accuracy decreases and validation loss becomes higher, which implies the Model 3 is too complex.

Based on the result, Model 1 should be chosen since it performs at least as good as the rest and has the most simple structure.

4. Experiments on max pooling layer and average pooling layer
   Resize to 64 * 64

   Model 1:
   Model Structure:
   input → convolutional layer → average pooling layer →
   convolutional layer → average pooling layer →
   convolutional layer → max pooling layer →
   fully connected layer → dropout layer →
   fully connected layer → fully connected layer → output

Model 2:

Model Structure:

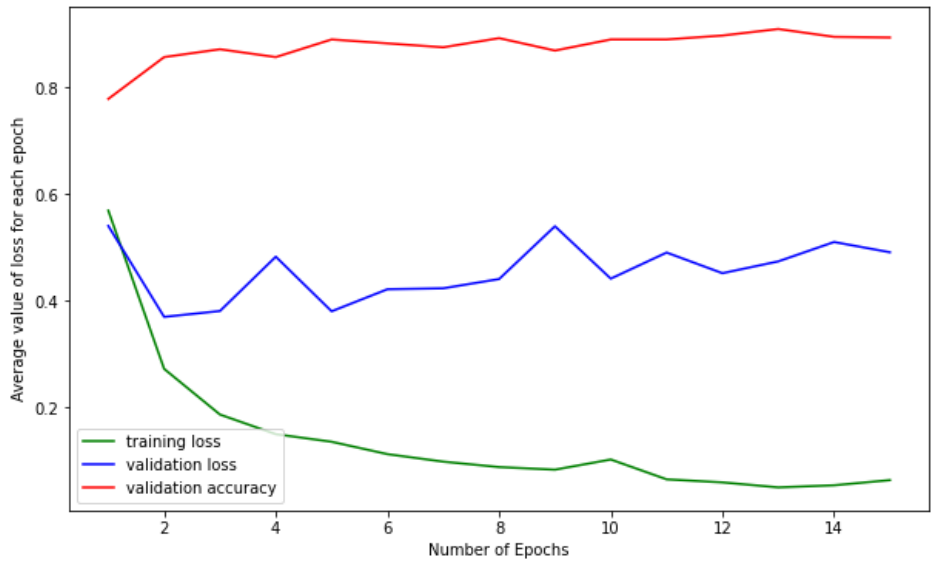
input → convolutional layer → average pooling layer →
convolutional layer → max pooling layer →
convolutional layer → max pooling layer →
fully connected layer → dropout layer →
fully connected layer → fully connected layer → output

Train 15 epoches for each model
Using Adam optimizer and nn.CrossEntropyLoss criterion.

Result:

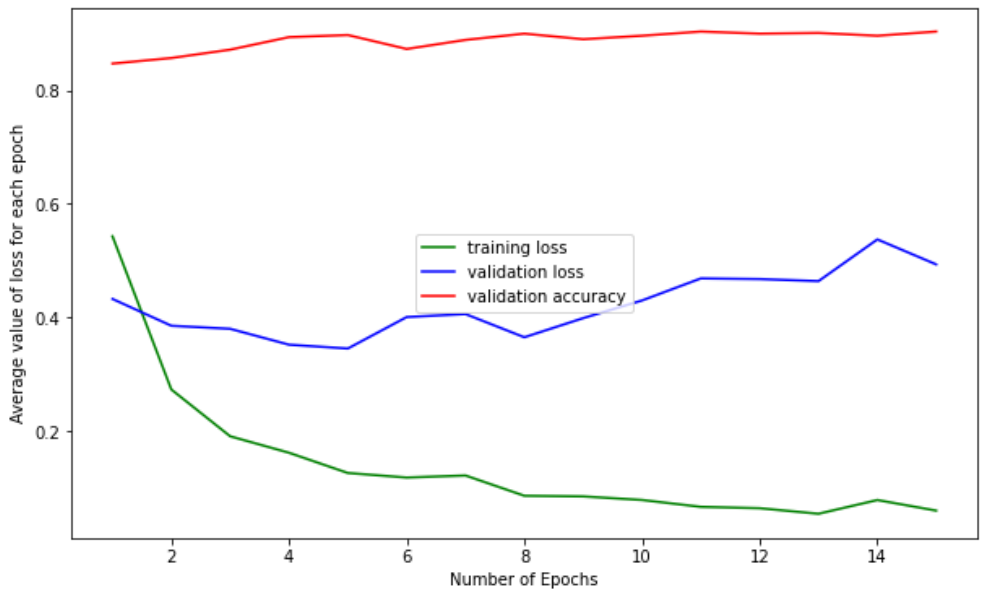
Model 1

```
The best acuracy is: 0.90931374
```




Model 2

```
The best acuracy is: 0.9031863
```



Model 1 has better performance than Model 2, this is because the mask is more like a region with almost the same color, so average pooling is performing better than max pooling since it's capturing the information of the whole region instead of the most significant one, this is especially important at the edge of mask, where average pooling will still capture the information of part of mask, max pooling may only capture the information of other things (e.g., face).

5. Experiment on size of feature map:
   Resize to 64 * 64

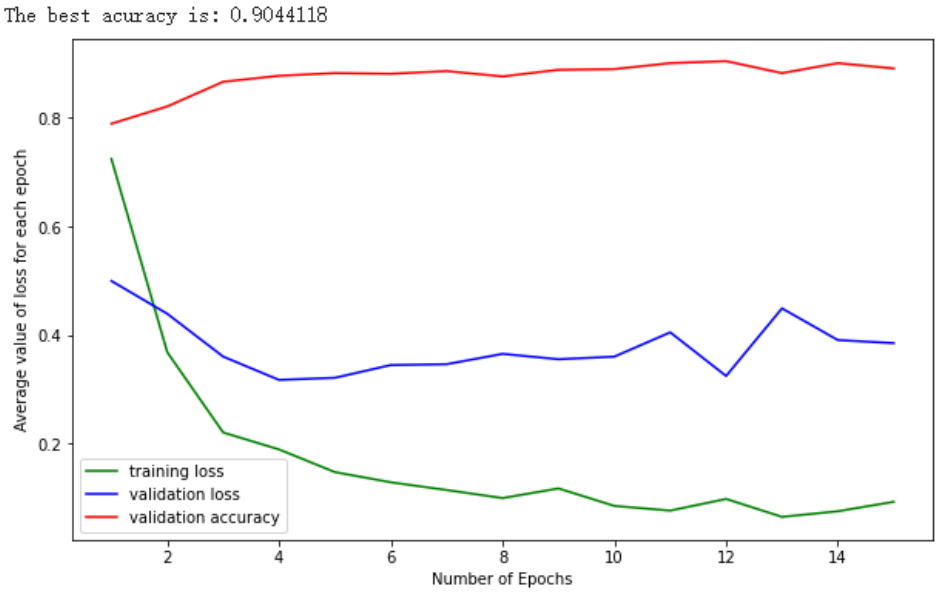
   Model 1:
   Model Structure:
   input → convolutional layer → average pooling layer →
   convolutional layer → max pooling layer →
   convolutional layer → max pooling layer →
   convolutional layer → max pooling layer →
   fully connected layer → dropout layer →
   fully connected layer → fully connected layer → output

   Model 2:
   Model Structure:
   input → convolutional layer → average pooling layer →
   convolutional layer → max pooling layer →
   convolutional layer → max pooling layer →
   convolutional layer → max pooling layer →
   fully connected layer → dropout layer →
   fully connected layer → fully connected layer → output

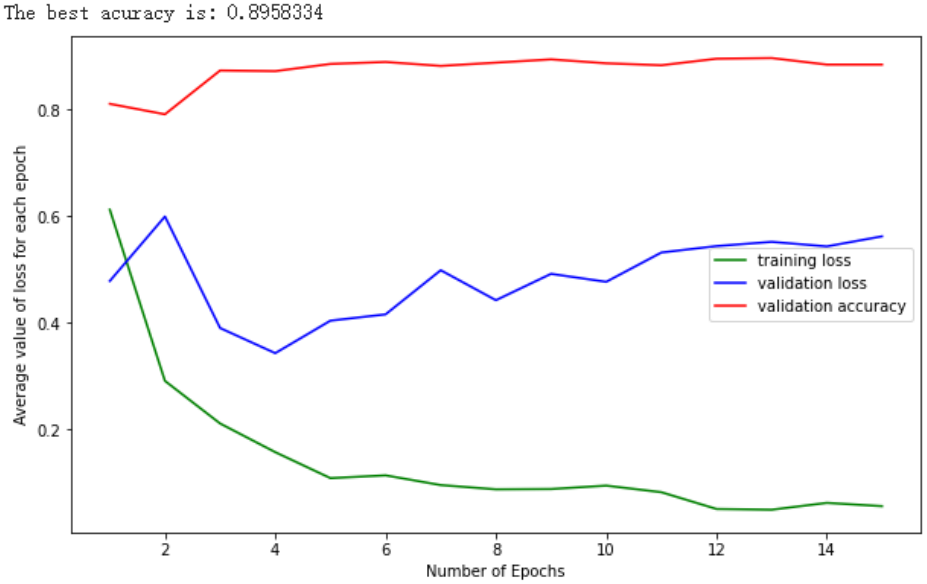
   The difference between two models is the kernel size of the last convolutional layer. In Model 1 it is 5, in Model 2 it is 3. In this case, before pass through fully connected layer, in Model 1 there should be 64 1 * 1 feature maps, in Model 2 there should be 64 2 * 2 feature maps.

Result:

Model 1




Model 2




Model 1 has better performance, meaning that trying to transform the feature map to smaller in the model before the fully connected layer will give us better performance.

6. Experiments on Transform on baseline model:
   Fixed : Baseline Model.
   Fixed : dataset
   Fixed : Adam optimizer with same parameters
   Fixed :15 epoches

Structure :

Variants : Transform(RandomHorizontalFlip(p=0.5))

Original Transform :

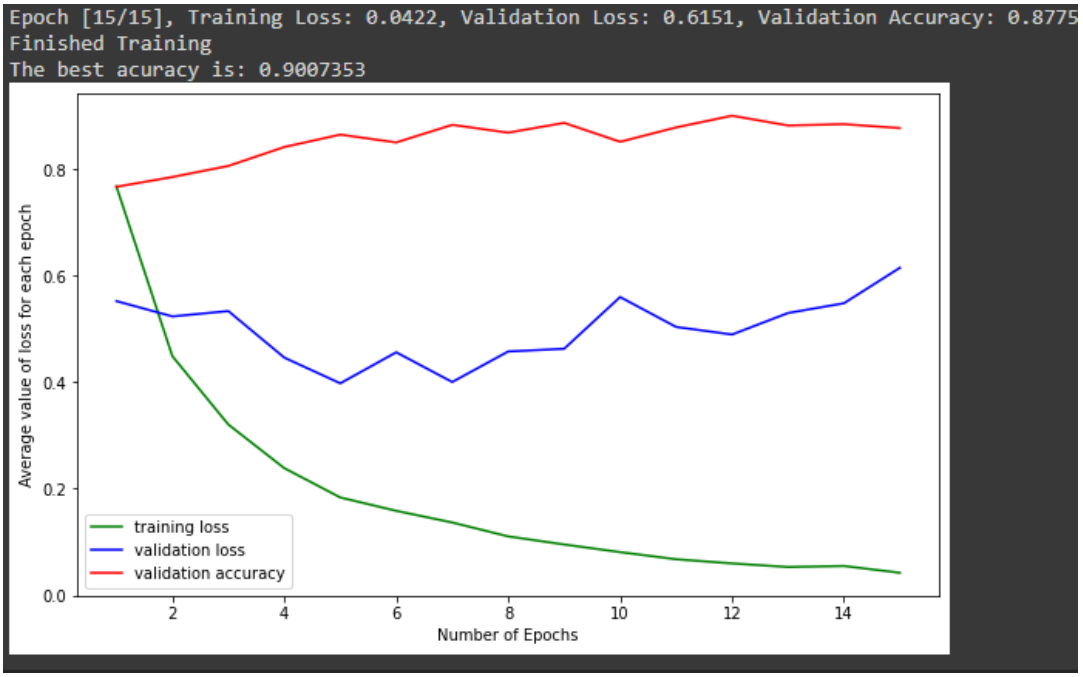
- transforms.Grayscale(1)
- transforms.Normalize((0.5, ), (0.5, )))

Modified Transform:

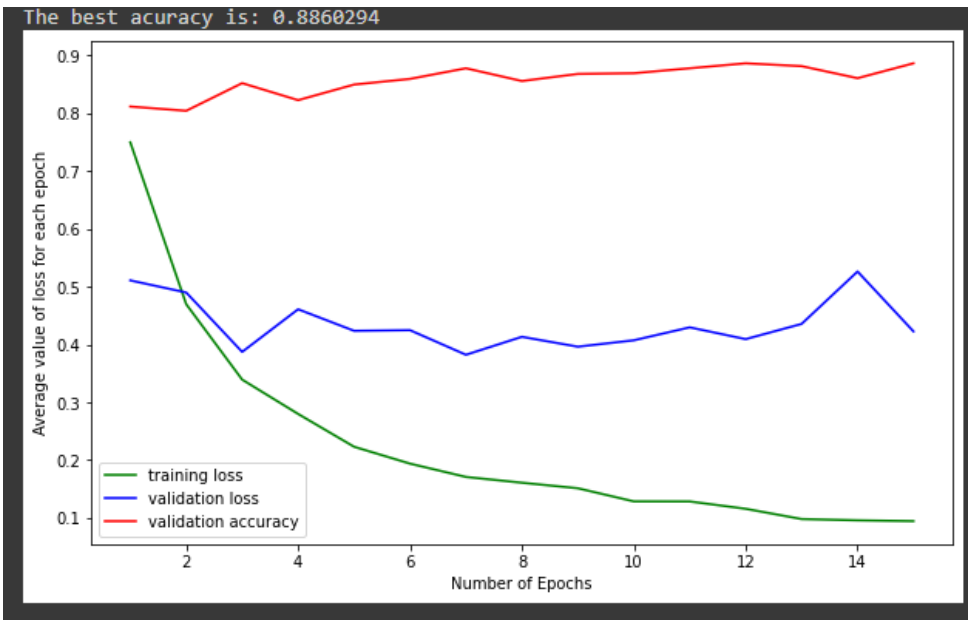
- transforms.RandomHorizontalFlip(p=0.5)
- transforms.Grayscale(1)
- transforms.Normalize((0.5, ), (0.5, )))

Result :

Original without random_flip:




With random_flip :



**Accuracy of Original one** : 0.9007
**Accuracy of the Modified one** : 0.88235295
**Better Accuracy** : Original one without random_flip
The validation loss is similar but the flipped one slightly decreases.
**Reason and Conclusion**:
In the two comparisons, discovering that random horizontal flip with 50%, giving the similar result in validation loss, but giving a lower accuracy.
The main reason is that most of our datasets are in the same horizontal direction for training, the model tries to determine the classes relies on some of the information here. Since the accuracy is not lower than the original one too much we picked the model of random flip one, avoiding the future testing cases failure if there are random_flip cases.

So Transform with random horizontal flip is kept in the following experiment.

7. Experiments on Dropout layers of baseline model:
   Fixed: **Transform**
   - transforms.RandomHorizontalFlip(p=0.5)
   - transforms.Grayscale(1)
   - transforms.Normalize((0.5, ), (0.5, )))

   Fixed: dataset.
   Fixed: Epochs = 15
   Fixed: Adam optimizer and its parameters
   Variants : **Dropout layer(0.5).**
   Original : **without dropout layers**
   Structure :

```
class Base_line(nn.Module):
    def __init__(self):
        super(Base_line, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(8, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

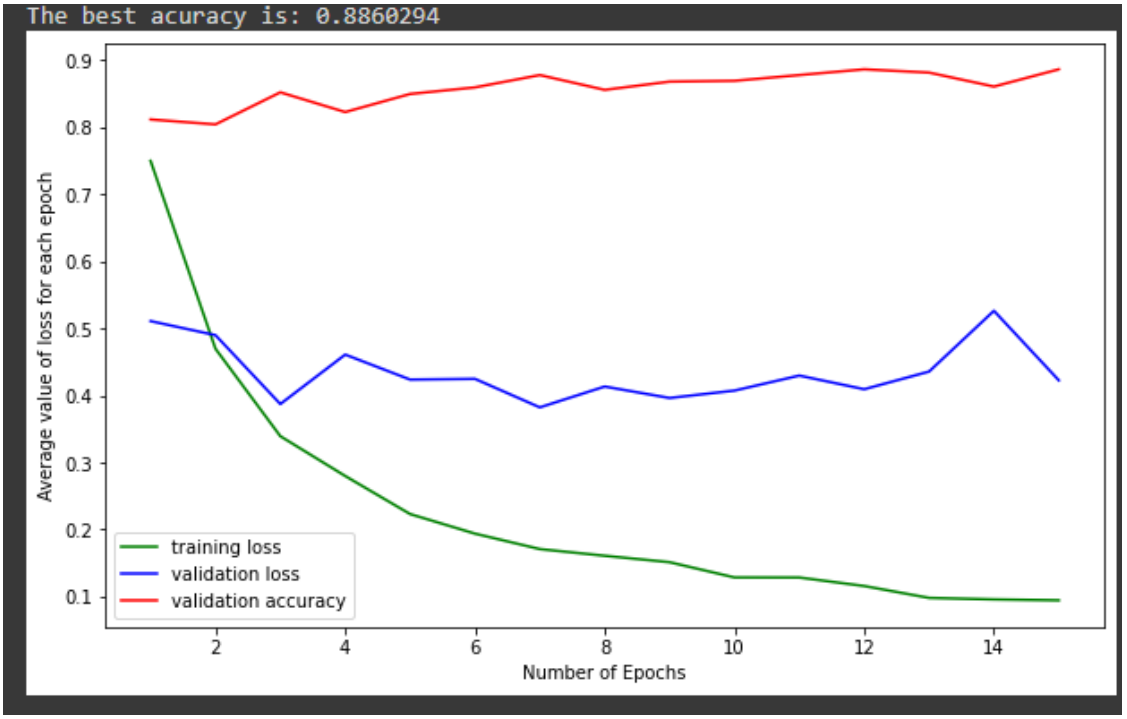
Modified 1: **with dropout layer(0.5)**

Structure:

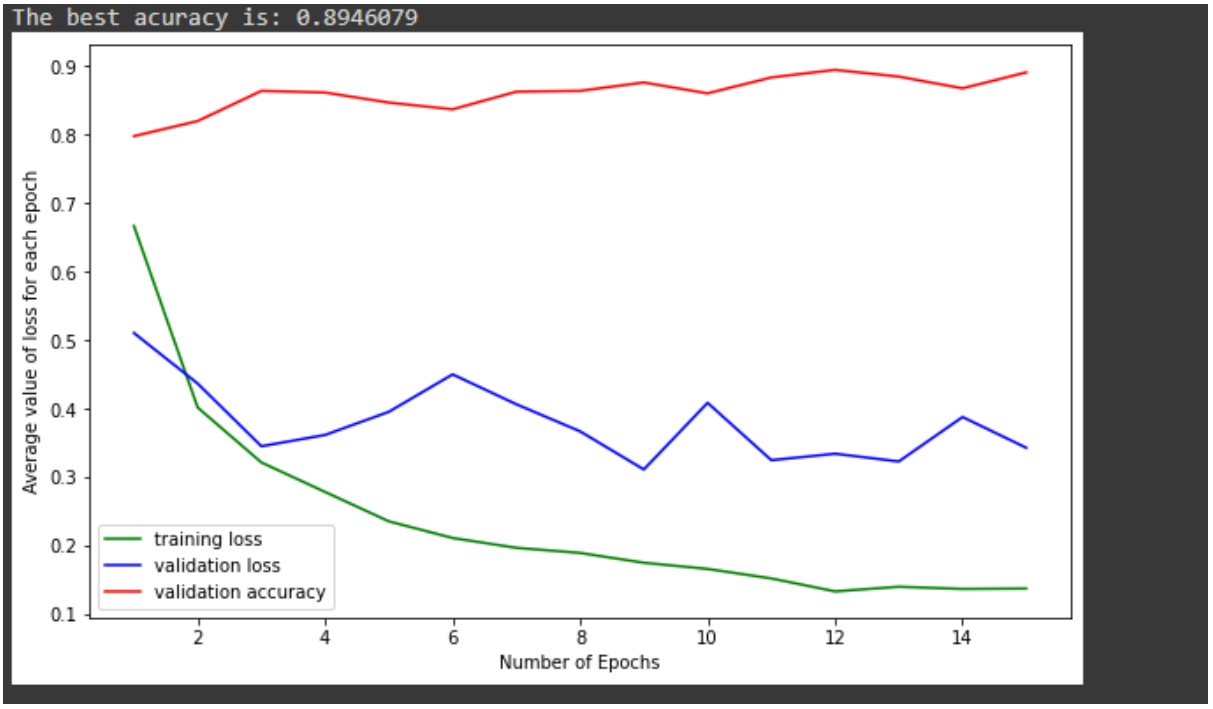
```
class Drop_Model(nn.Module):
    def __init__(self):
        super(Drop_Model, self).__init__()

        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.norm1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,  kernel_size=5)
        self.norm2 = nn.BatchNorm2d(16)
        self.drop = nn.Dropout(0.5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3)

    def forward(self, image):
        image = self.norm1(self.pool(F.relu(self.conv1(image)))) #6, 14, 14
        image = self.norm2(self.pool(F.relu(self.conv2(image)))) #16, 5, 5
        image = image.view(-1, 16 * 5 * 5)
        image = self.drop(image)
        image = F.relu(self.fc1(image))
        image = F.relu(self.fc2(image))
        image = self.fc3(image)
        return image
```

Result :

Origin :



Modified1:




Validation loss largely decreases, and closer to the training loss. While the accuracy increases by 0.01 as well

Reason: Dropout works by randomly setting the outgoing edges of hidden units (neurons that make up hidden layers) to 0, which may reduce the possible overfitting problem and the validation loss. Prevent models depend more on some specific features.

Conclusion : Dropout layer helps reduce validation loss, which is needed when when there is overfitting

8. Experiments on convolutional layers on **baseline model**:

Fixed: Transform

- transforms.RandomHorizontalFlip(p=0.5)
- transforms.Grayscale(1)
- transforms.Normalize((0.5, ), (0.5, )))

Fixed: dataset.

Fixed: Epochs

Fixed: Adam optimizer and its parameters

Structure:

Origin:

```
class Base_line(nn.Module):
    def __init__(self):
        super(Base_line, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(8, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Modified 1:

```
class Three_layer_model(nn.Module):
    def __init__(self):
        super(Three_layer_model, self).__init__()

        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 5)  #28
        self.pool = nn.MaxPool2d(2,2)#14
        self.norm1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,  kernel_size=3) #12 #6
        self.norm2 = nn.BatchNorm2d(16)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=32,  kernel_size=3)  #4 #2
        self.norm3 = nn.BatchNorm2d(32)
        self.fc1 = nn.Linear(32*2*2, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3)

    def forward(self, image):
        image = self.norm1(self.pool(F.relu(self.conv1(image)))) #6, 14, 14
        image = self.norm2(self.pool(F.relu(self.conv2(image)))) #16, 5, 5
        image = self.norm3(self.pool(F.relu(self.conv3(image))))
        image = image.view(-1, 32 * 2 * 2)
        image = F.relu(self.fc1(image))
        image = F.relu(self.fc2(image))
        image = self.fc3(image)
        return image
```

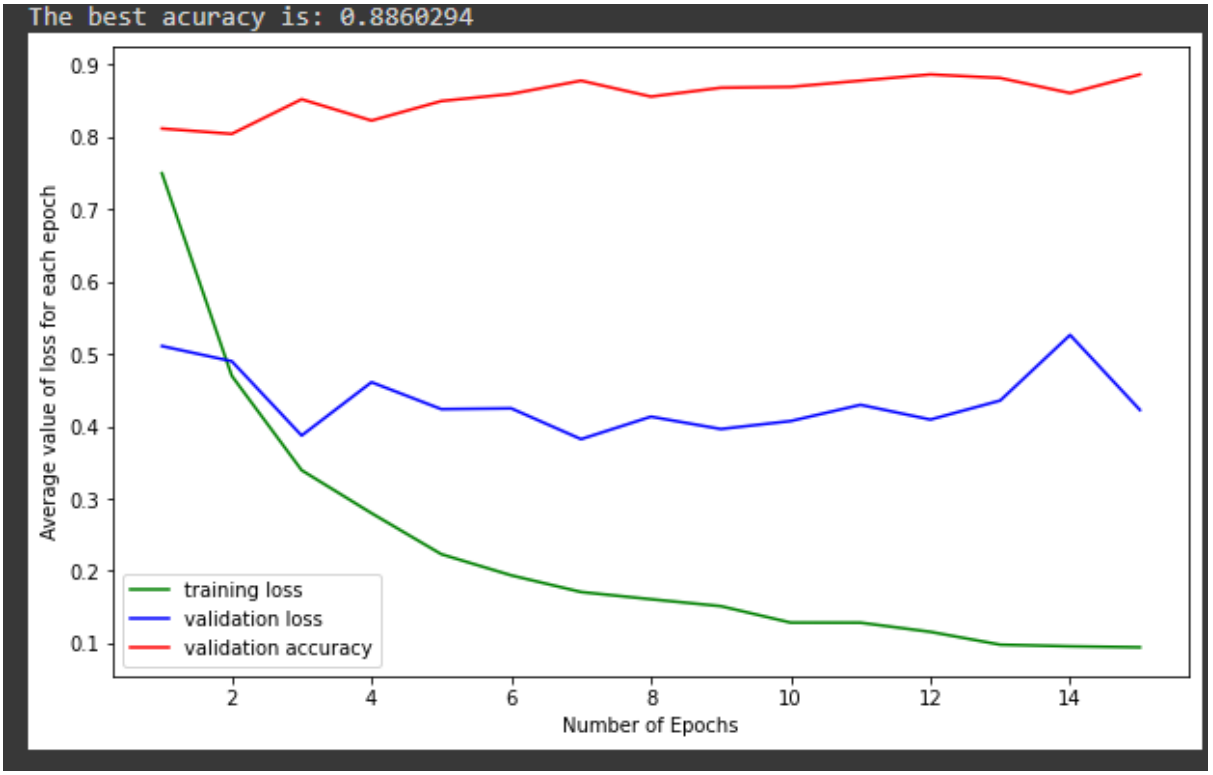
Variants :  number of cov layers.

Origin : Two convolutional layers

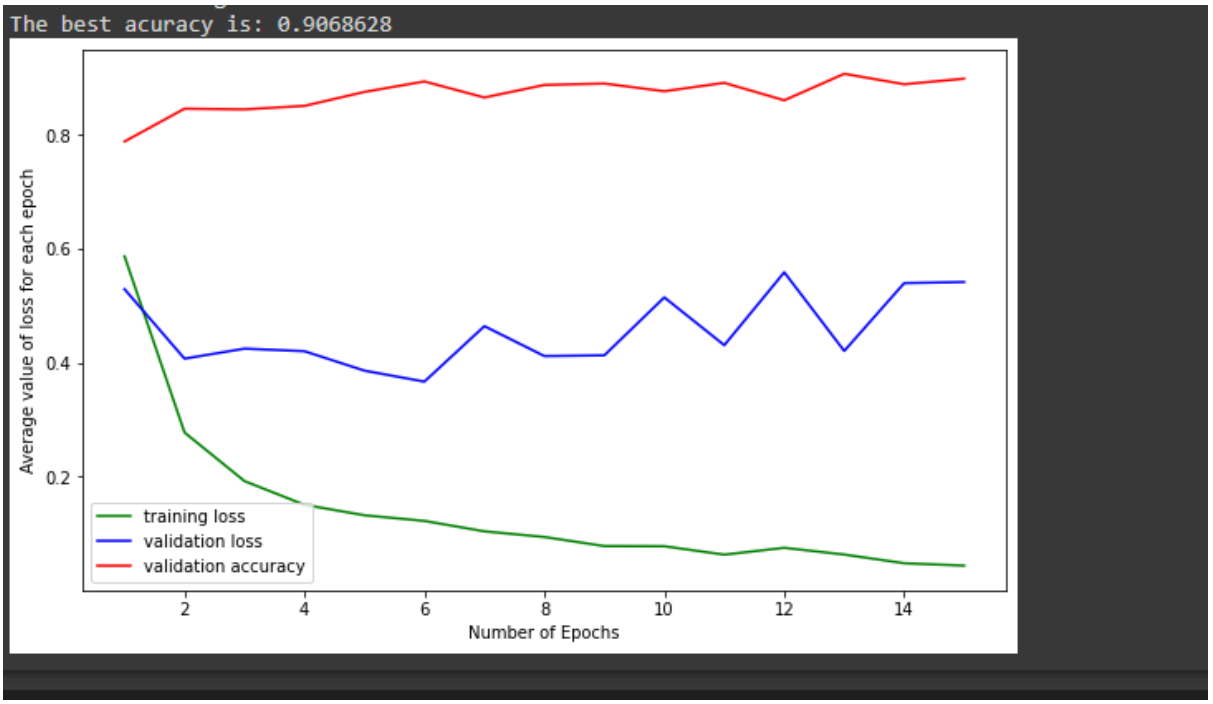
Modified 1 : Three convolutional layers

Result :

Origin:




Modified:



**Result** : Three convolutional layer improves the accuracy by 0.02 while the validation loss is similar to each other. Training loss is similar as well.

**Reason**: Three convolutional layers solve the possible underfitting problems

Make the model more trainable and increases the learnability, but we do not add too many here, since we do not want the model to be overfitted, so adding one is enough.

**Purpose** : To see whether there are overfitting problem, and decide whether we need to add dropout to compare the performance.

**Conclusion**: we will add dropout to decrease the validation loss, in Experiment 10. while comparing the same set here.

9. Experiments on number layers type on baseline model:
   **without Dropout**
   Fixed: Transform
   - transforms.RandomHorizontalFlip(p=0.5)
   - transforms.Grayscale(1)
   - transforms.Normalize((0.5, ), (0.5, )))

   Fixed: dataset.
   Fixed: Epochs
   Fixed: Adam optimizer and its parameters
   Structure:
   Origin:

```
class Base_line(nn.Module):
    def __init__(self):
        super(Base_line, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(8, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Modified 1:

```
import matplotlib.pyplot as plt
class Three_layer_model(nn.Module):
    def __init__(self):
        super(Three_layer_model, self).__init__()

        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 5)  #28
        self.pool = nn.MaxPool2d(2,2)#14
        self.norm1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,  kernel_size=3) #12 #6
        self.norm2 = nn.BatchNorm2d(16)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=32,  kernel_size=3)  #4 #2
        self.norm3 = nn.BatchNorm2d(32)
        self.fc1 = nn.Linear(32*2*2, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3)

    def forward(self, image):
        image = self.norm1(self.pool(F.relu(self.conv1(image)))) #6, 14, 14
        image = self.norm2(self.pool(F.relu(self.conv2(image)))) #16, 5, 5
        image = self.norm3(self.pool(F.relu(self.conv3(image))))
        image = image.view(-1, 32 * 2 * 2)
        image = F.relu(self.fc1(image))
        image = F.relu(self.fc2(image))
        image = self.fc3(image)
        return image
```

Modified 2:

```
class doubleConv_dliner(nn.Module):
    def __init__(self):
        super(doubleConv_dliner, self).__init__()

        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.norm1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,  kernel_size=5)
        self.norm2 = nn.BatchNorm2d(16)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 96)
        self.fc3 = nn.Linear(96, 84)
        self.fc4 = nn.Linear(84, 3)

    def forward(self, image):
        image = self.norm1(self.pool(F.relu(self.conv1(image)))) #6, 14, 14
        image = self.norm2(self.pool(F.relu(self.conv2(image)))) #16, 5, 5
        image = image.view(-1, 16 * 5 * 5)

        image = F.relu(self.fc1(image))
        image = F.relu(self.fc2(image))
        image = F.relu(self.fc3(image))
        image = self.fc4(image)
        return image
```

**Variant** : adding conv layer vs fully connected layers

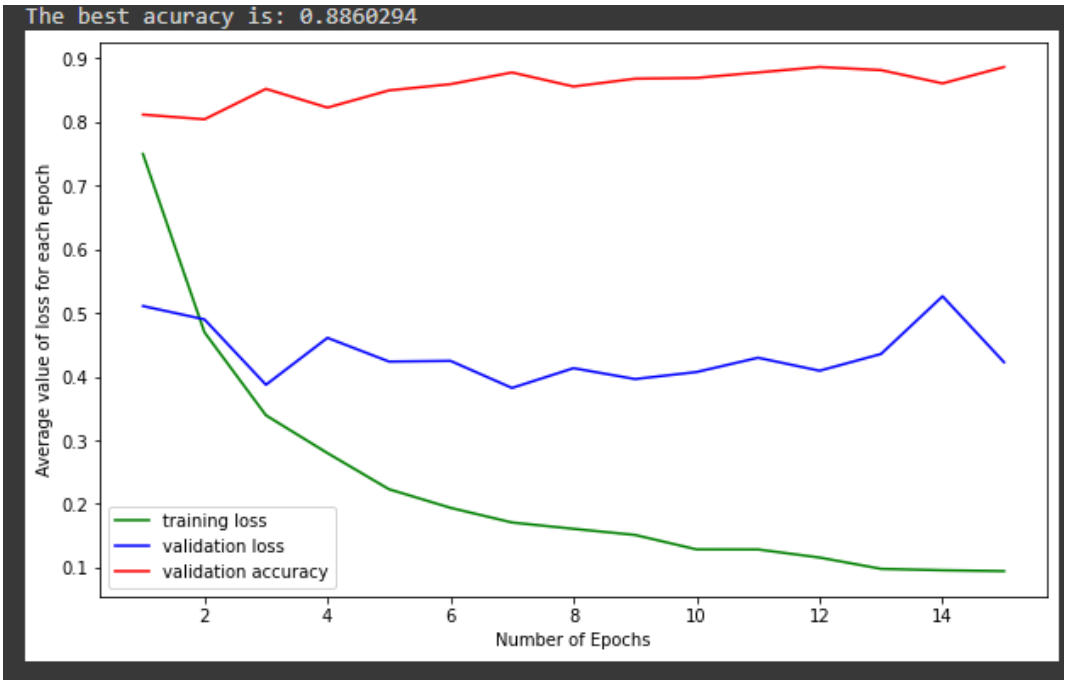
**Origin** : Two convolutional layers with three fully connected layer

**Modified** 1 : Three convolutional layers with three fully connected layer

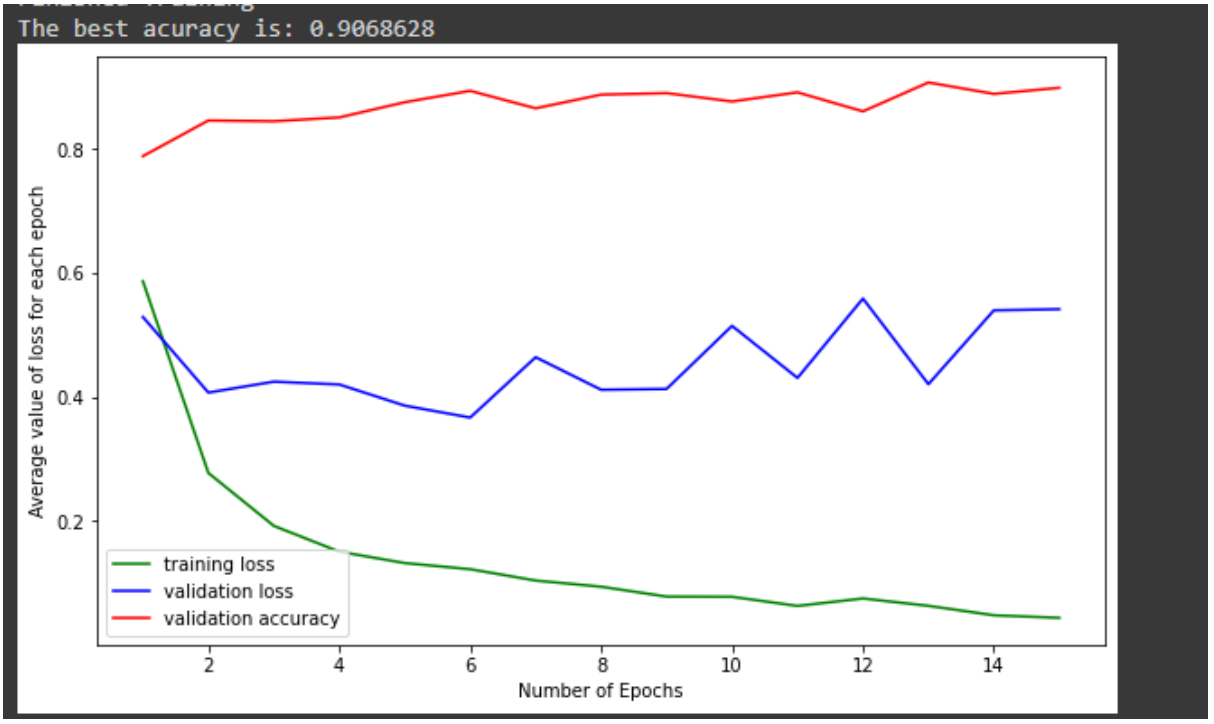
**Modified** 2 : Two convolutional layers with four fully connected layer

**Result**:

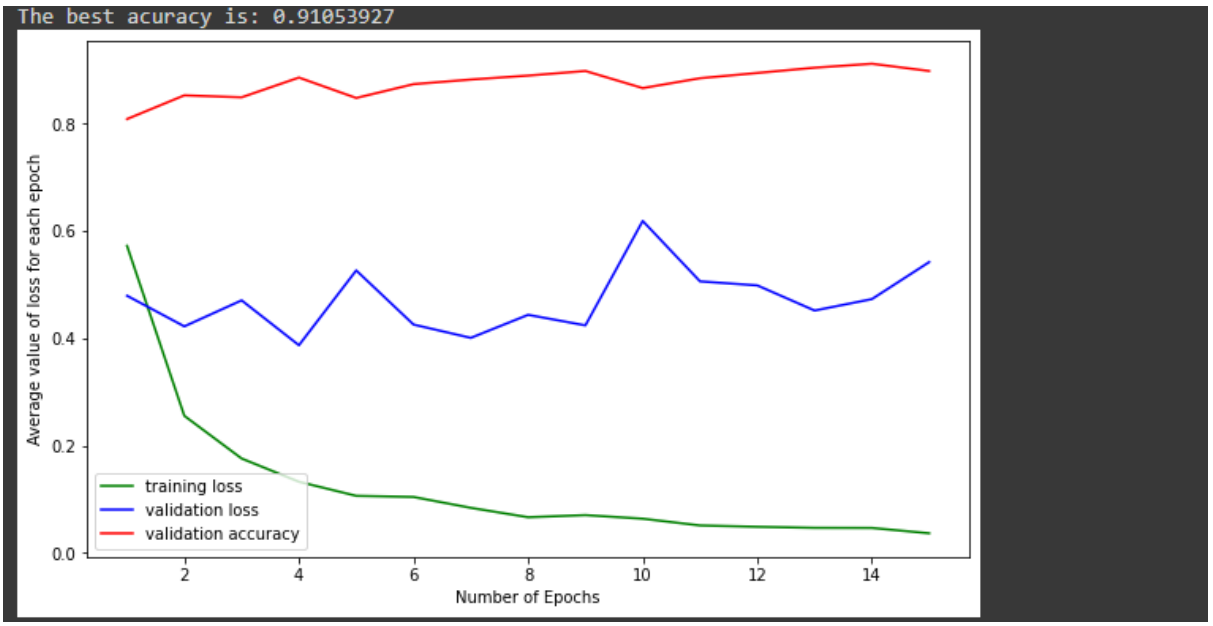
**Origin**:




Modified 1:



**Modified 2**:




validation loss is not variate a lot. but there is a large gap between training and validation loss. The Original one has lower validation loss, so we need to solve the validation loss problems, since their accuracy is similar:

Four fully connected layer one gets the best accuracy 0.91,

Three conv layers gets second accuracy 0.906

Original layers gets lowest 0.88

**Reason**: possible reason is that fully connected layer is easier for the model to do learning, while the convolutional layer is more complex.

both of them is better than the original model, since they solve the possible underfitting problem and make the model learnable.

**purpose** : See if we need to add drop out by looking at the validation loss, determine whether more fully connected layer or conv layers can improve the accuracy.

**Conclusion**: we have to add a dropout layer in experiment 10 to decide, reducing the overfitting. Then choose the best one.

10. Experiments on  three layers based on best model on Experiment 7 :

Fixed: Transform

- transforms.RandomHorizontalFlip(p=0.5)
- transforms.Grayscale(1)
- transforms.Normalize((0.5, ), (0.5, )))

Fixed: dataset.

Fixed: Epochs =15

Fixed: dropout layer

Fixed: Adam optimizer and its parameters

**Structure**:

**Origin**:

```
class Drop_Model(nn.Module):
    def __init__(self):
        super(Drop_Model, self).__init__()

        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.norm1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,  kernel_size=5)
        self.norm2 = nn.BatchNorm2d(16)
        self.drop = nn.Dropout(0.5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3)

    def forward(self, image):
        image = self.norm1(self.pool(F.relu(self.conv1(image)))) #6, 14, 14
        image = self.norm2(self.pool(F.relu(self.conv2(image)))) #16, 5, 5
        image = image.view(-1, 16 * 5 * 5)
        image = self.drop(image)
        image = F.relu(self.fc1(image))
        image = F.relu(self.fc2(image))
        image = self.fc3(image)
        return image
```

**Modified 1**:(three conv)

```
class Three_layer_model(nn.Module):
    def __init__(self):
        super(Three_layer_model, self).__init__()

        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 5)  #28
        self.pool = nn.MaxPool2d(2,2)#14
        self.norm1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,  kernel_size=3) #12 #6
        self.norm2 = nn.BatchNorm2d(16)
        self.conv3 = nn.Conv2d(in_channels=16, out_channels=32,  kernel_size=3)  #4 #2
        self.norm3 = nn.BatchNorm2d(32)
        self.drop = nn.Dropout(0.5)
        self.fc1 = nn.Linear(32*2*2, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 3)

    def forward(self, image):
        image = self.norm1(self.pool(F.relu(self.conv1(image)))) #6, 14, 14
        image = self.norm2(self.pool(F.relu(self.conv2(image)))) #16, 5, 5
        image = self.norm3(self.pool(F.relu(self.conv3(image))))

        image = image.view(-1, 32 * 2 * 2)
        image = self.drop(image)
        image = F.relu(self.fc1(image))
        image = F.relu(self.fc2(image))
        image = self.fc3(image)
        return image
```

**Modified 2** : (four fully connected layer)

```
class doubleConv_dliner(nn.Module):
    def __init__(self):
        super(doubleConv_dliner, self).__init__()

        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 8, kernel_size = 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.norm1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(in_channels=8, out_channels=16,  kernel_size=5)
        self.norm2 = nn.BatchNorm2d(16)
        self.drop = nn.Dropout(0.5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 96)
        self.fc3 = nn.Linear(96, 84)
        self.fc4 = nn.Linear(84, 3)

    def forward(self, image):
        image = self.norm1(self.pool(F.relu(self.conv1(image)))) #6, 14, 14
        image = self.norm2(self.pool(F.relu(self.conv2(image)))) #16, 5, 5
        image = image.view(-1, 16 * 5 * 5)
        image = self.drop(image)
        image = F.relu(self.fc1(image))
        image = F.relu(self.fc2(image))
        image = F.relu(self.fc3(image))
        image = self.fc4(image)
        return image
```

**Variant** : number of convolutional layer

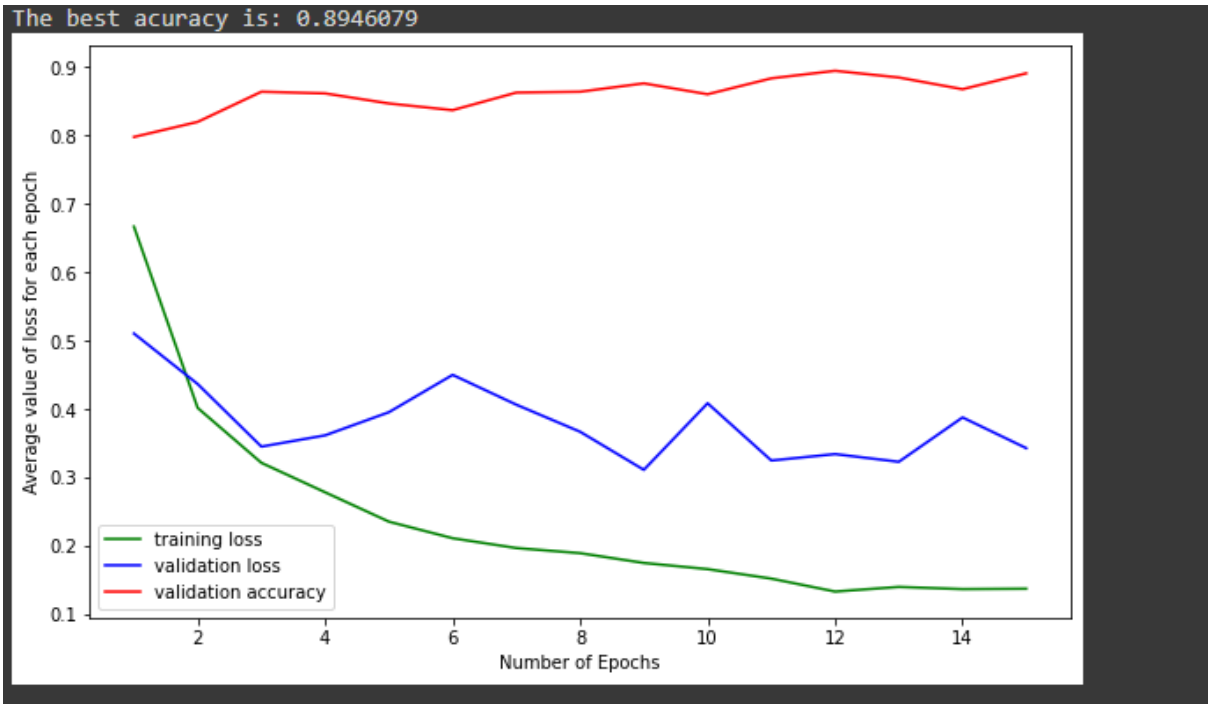
**Origin** : Two convolutional layers with dropout and 3 fc layers

**Modified 1**: Three convolutional layers with dropout and 3 fc layers

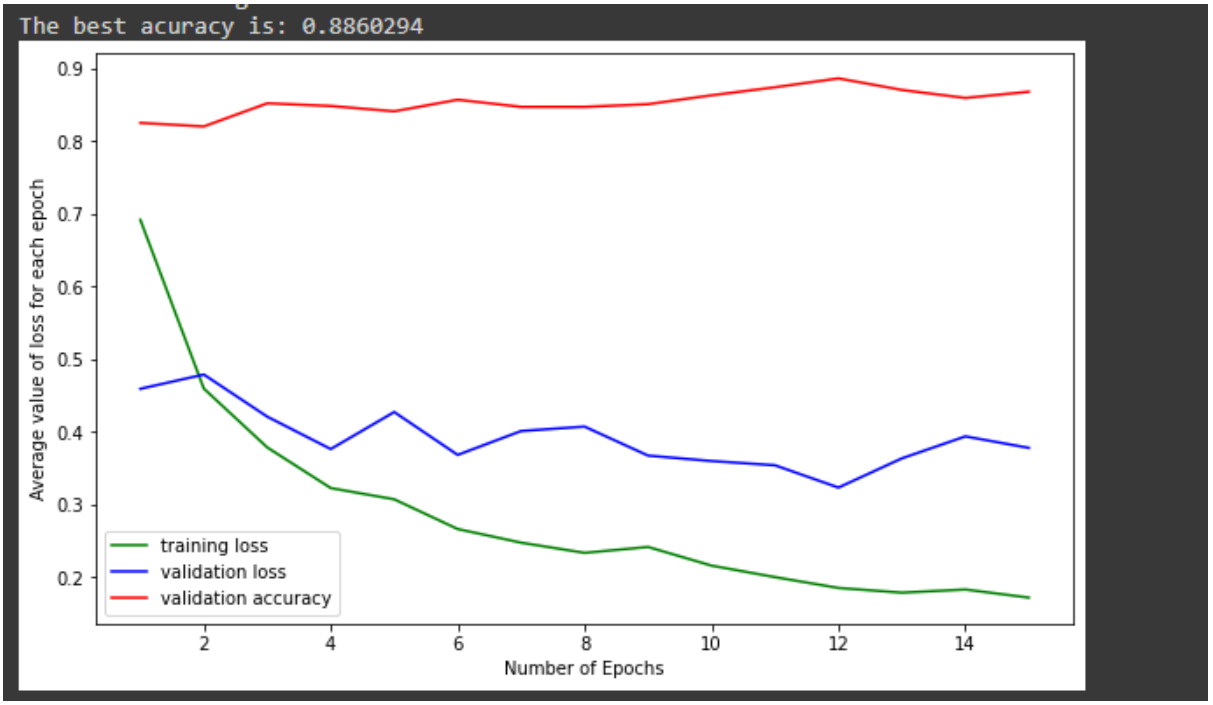
**Modified 2** : Four fully connected layers with two conv layers and dropout.

**Result** :

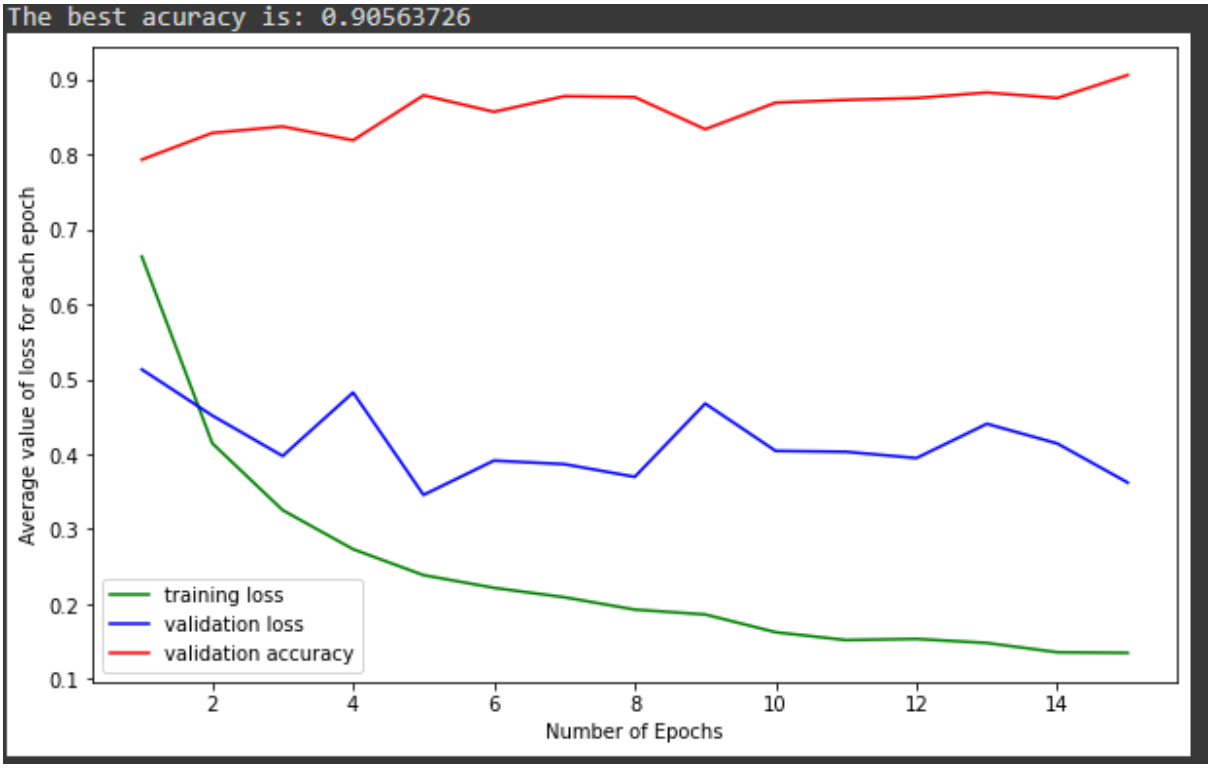
**Origin**:




**Modified 1**:



**Modified 2**:




Both model's Validation loss decreases while accuracy is not decreased a lot. Based on dropout, the Modified 2 is the best as before.

Reason : The same reason as experiment 5 and 6, but the validation loss is improved since the gap between training loss and validation loss is smaller. For modified 1 , we have three convolutional layer, but the validation accuracy decreases , the possible reason is that we need more epochs to train the model in order to get higher accuracy, which it supposes to be higher than the origin.

**Division of labor:** 50% by Liu Pak Wai and 50% by ZHU, Chen

**Hyperlink to Youtube:** [https://youtu.be/SiIevNXTJ-Q](https://youtu.be/SiIevNXTJ-Q)