HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

ADVANCED PROGRAM

CAPSTONE PROJECT 2

Project: Deploy a plant disease classification model using Flask

Lecturer: Associate Professor PhD. Ha Hoang Kha

Student: To Duc Thanh ID: 1851015

Contents

A. Abstract

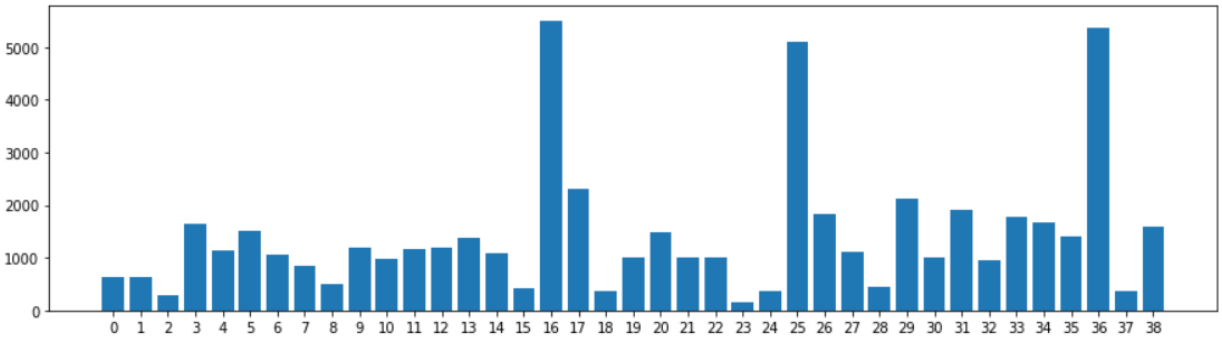
Plant disease is a serious problem and it is a challenge to identify and classify correctly and quickly. Many researches and papers attempted to find the optimal solution and with the explosion of Artificial Intelligent, more and more amazing results appeared. In this project, I approached this classification problem by using different deep learning models, on which I evaluated, tuned hyperparameters, tried several techniques. Two datasets and some architectures were used to prove the power of Deep Learning methods.

B. Relevant theory

1) Dataset

In this project, I experimented on 2 datasets:

Dataset A: This dataset consists of 39 classes of plant leaves and background without leaves [1]. It is from an original paper in 2015 [2], in which the authors used 6 different augmentation techniques to increase the dataset size (of 55,448 images). However, I utilized the original dataset and applied my own transformation to grow its size.



This bar chart displays the number of each class, with the following notation:

```
{'Apple___Apple_scab': 0,
 'Apple___Black_rot': 1,
 'Apple___Cedar_apple_rust': 2,
 'Apple___healthy': 3,
 'Background_without_leaves': 4,
 'Blueberry___healthy': 5,
 'Cherry___Powdery_mildew': 6,
 'Cherry___healthy': 7,
 'Corn___Cercospora_leaf_spot Gray_leaf_spot': 8,
 'Corn___Common_rust': 9,
 'Corn___Northern_Leaf_Blight': 10,
 'Corn___healthy': 11,
 'Grape___Black_rot': 12,
 'Grape___Esca_(Black_Measles)': 13,
 'Grape___Leaf_blight_(Isariopsis_Leaf_Spot)': 14,
 'Grape___healthy': 15,
 'Orange___Haunglongbing_(Citrus_greening)': 16,
 'Peach___Bacterial_spot': 17,
 'Peach___healthy': 18,
 'Pepper,_bell___Bacterial_spot': 19,
 'Pepper,_bell___healthy': 20,
 'Potato___Early_blight': 21,
 'Potato___Late_blight': 22,
 'Potato___healthy': 23,
 'Raspberry___healthy': 24,
 'Soybean___healthy': 25,
 'Squash___Powdery_mildew': 26,
 'Strawberry___Leaf_scorch': 27,
 'Strawberry___healthy': 28,
 'Tomato___Bacterial_spot': 29,
 'Tomato___Early_blight': 30,
 'Tomato___Late_blight': 31,
 'Tomato___Leaf_Mold': 32,
 'Tomato___Septoria_leaf_spot': 33,
 'Tomato___Spider_mites Two-spotted_spider_mite': 34,
 'Tomato___Target_Spot': 35,
 'Tomato___Tomato_Yellow_Leaf_Curl_Virus': 36,
 'Tomato___Tomato_mosaic_virus': 37,
 'Tomato___healthy': 38}
```

As can be seen from the bar chart, the classes are imbalanced.

Some images in dataset A:

These are Apple___Apple_scab, Cherry___healthy, Background_without_leaves, respectively.

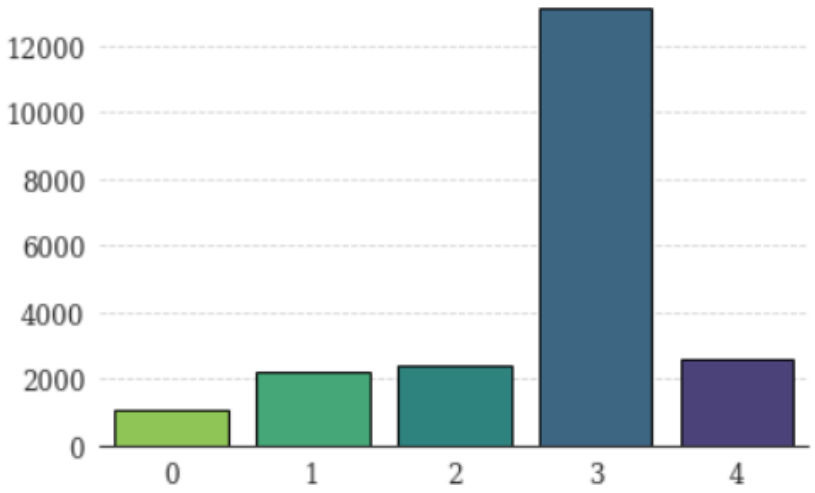
Dataset B: The second dataset was crawled down from a Kaggle competition (Cassava Leaf Disease Classification) [3]. I accepted the rules for this competition under Kaggle competition to use this dataset for education purpose.

This dataset has 21, 367 images collected in Uganda. There are 5 classes in which 4 classes are diseases leaves and last class is healthy.

The labels are:

{"0": "Cassava Bacterial Blight (CBB)", "1": "Cassava Brown Streak Disease (CBSD)", "2": "Cassava Green Mottle (CGM)", "3": "Cassava Mosaic Disease (CMD)", "4": "Healthy"}.

The distribution of this dataset:



The classes are imbalanced. Class 3 is dominant the dataset.

Some images in dataset B:

The order from left to right, up to down: class 0, 1, 2, 3, 4

2) Weight initialization

In traditional machine learning, weights are usually initialized zeros, which will be updated afterward during training process. In neural network, this causes a problem. Neural network is powerful for its non-linear functions (activation functions) and it calculates dot product between the input and the weights. If zeros are assigned to the weights, these dot products may be zeros in many layers, which loses the non-linear characteristic of the network. As a result, the whole system becomes a linear function.

Because of this reason, people initialize the weights randomly and gaussian distribution is the most popular for this purpose. In some cases, gaussian distribution responses quite well, but most of the time it is a serious problem. As its variance is 1, the range is still high for some neural network and exploding gradient occurs. To fix this problem, gaussian distribution is multiplied by a weight scale. This weight scale is in range from 1 to 1e-5

weight = weight scale * randn (fan_in, fan_out)

fan_in: number of input units in weight tensors.

fan_out: number of output units in weight tensors.

Even though this approach improves the exploding gradient problem, it takes time to search for a well-behaved weight scale and the sensitivity of the network to these scales is high. Another problem is that "the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs." [4]. This makes the distributions of different layers may be different.

In this project, I do not use this approach, but instead try 2 famous ways that give the network more control for all cases. The idea is similar to weight scale: finding a good variance for the distribution from which the initial parameters are drawn, but more flexible.

a) Glorot initialization [Glorot et al., 2010]

var = 2 / (fan_in + fan_out)

weight = sqrt(var) * randn (fan_in, fan_out)

Glorot initialization tries to get the same spread of distribution at the input and the output by set as the formula. One more reason is "based on a compromise and an equivalent analysis of the backpropagated gradients" [4].

b) He initialization [He et al, 2015]

var = 2 / (fan_in)

weight = sqrt(var) * randn (fan_in, fan_out)

This is another strategy by He et al, "derives an initialization specifically for Relu neurons, reaching the conclusion that the variance of neurons in the network should be 2.0/fan_in" [4].

3) Optimizers

In this project, I chose 4 optimizers: SGD, SGD momentum, RMSprop and Adam for experimenting on my datasets.

SGD is a traditional optimizer that is still widely used in many cases, even though its learning rate is a tricky hyperparameter to tune.

SGD momentum is a better version of vanilla SGD as it utilizes a new hyperparameter called “momentum” to help faster convergence. Learning rate is still a problem and also momentum coefficient. Usually, momentum coefficient is set to values such as [0.5, 0.9, 0.95, 0.99]. In general, when cross-validated to the best hyperparameters, SGD momentum moves more quickly towards the minima than SGD. “For this reason, momentum is also referred to as a technique which dampens oscillations in our search.” [5]

RMSprop approaches a different path as tuning learning rates is an expensive process, although the aim is similar that is to dampen oscillations. RMSprop tries to manipulate learning rates automatically. In short, the weights have high gradients will get learning rate reduced, and weights with small gradients will receive learning rate increased. In this project, due to time and memory limitation of cloud kernel, on which I run GPU, I do not tune the hyperparameters of RMSprop optimizer.

Adam is in some sense like RMSprop with momentum, which inherits the advantages of these methods. In many deep learning frameworks, Adam is the default algorithm, but it is still worth to give other optimizers a try.

Another technique I took advantage of in this project is annealing the learning rate, step decay for specific. This happens as follow: the model monitors the validation loss while training, and decreases the learning rate by some constant factor whenever the validation loss stops reducing or reduces smaller than some specific range.

4) Batch Normalization

Batch Normalization is a technique that helps reduce the consequence of bad initialization by “explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training” [4], which means maintaining the “stable” initialization throughout layers. In the original paper [6], the authors hypothesized that the shifting distribution of features inside deep networks make training process more difficult and recommended to add batch normalization layers into these networks. It is a controversy whether to add the batch normalization layer before or after the activation layer, though.

In this project, I tried to add before and after activation layers to see which is better.

These are formulas taken from the paper:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

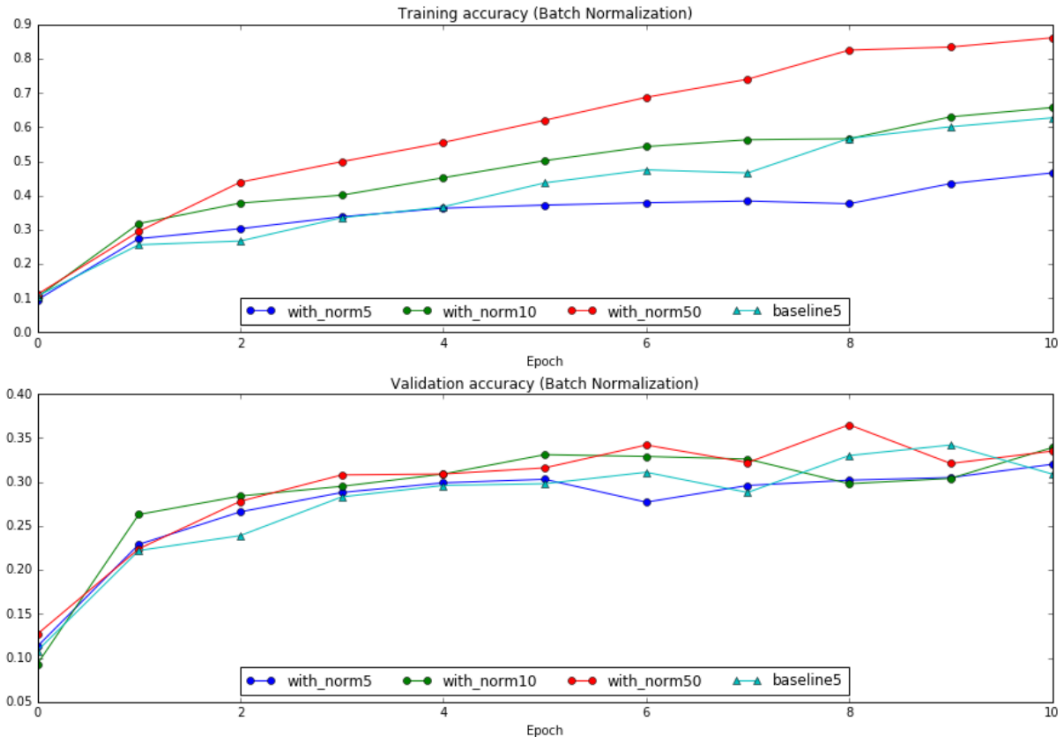
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

From the formulas, it can be understood that normalization process calculates mean and variance over batch. This is one disadvantage of batch normalization as it is affected by the batch size. A small batch size can lead to a bad approximation of the statistics of the whole dataset, while a large size can improve the approximation a lot. This can be a huge problem for machines with limited memory. The following plots prove the relationship between batch normalization and batch size. This is an experiment on a small size of CIFAR-10 dataset.



(CS231n Convolution Neural Networks for Visual Recognition, assignment 2)

As can be seen from the figures, with small batch size of 5, the base model (without using batch normalization) even performs better.

One bonus of batch normalization the authors said is regularization effect, although in my experience it is not that strong.

5) Layer Normalization

The weakness of batch normalization regarding batch size leads to the birth of layer normalization. Instead of normalizing over the batch, this technique normalizes over the features. The authors change the formulas to calculate statistics as follow: (taken from the original paper [7]).

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

where H denotes the number of hidden units in a layer. "…under layer normalization, all the hidden units in a layer share the same normalization terms μ and σ, but different training cases have different normalization terms. Unlike batch normalization, layer normalization does not impose any constraint on the size of a mini-batch and it can be used in the pure online regime with batch size 1." [7]

6) Data augmentation

Deep learning model is hungry for data, the more data the more generalizability of the model. Data augmentation helps the model see "new" samples from the original dataset by applying on it some transformations. These transformations include translation, rotation, scale change, shearing, horizontal flip, vertical flip, brightness change…

There are three types of data augmentation [8]:

Type 1: Expanding the existing dataset. Steps include: load the original input images from disk, then apply transformations on those images, then save them on disk. This will increase the size of dataset.

Type 2: On-the-fly data augmentation. This type is most common, and while it also applies transformations on original source, it does not save the modified version and the process occurs at training time. This will not increase the size of dataset.

Type 3: Combining both types.

C. Experiments

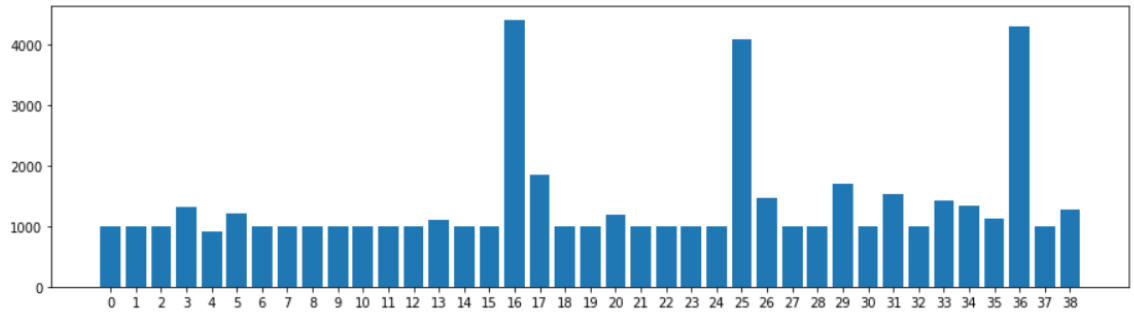
1) Dealing with imbalanced class problem

The specific number of samples in each class of dataset A:

```
{0: 630,
 1: 621,
 2: 275,
 3: 1645,
 4: 1143,
 5: 1502,
 6: 1052,
 7: 854,
 8: 513,
 9: 1192,
 10: 985,
 11: 1162,
 12: 1180,
 13: 1383,
 14: 1076,
 15: 423,
 16: 5507,
 17: 2297,
 18: 360,
 19: 997,
 20: 1478,
 21: 1000,
 22: 1000,
 23: 152,
 24: 371,
 25: 5090,
 26: 1835,
 27: 1109,
 28: 456,
 29: 2127,
 30: 1000,
 31: 1909,
 32: 952,
 33: 1771,
 34: 1676,
 35: 1404,
 36: 5357,
 37: 373,
 38: 1591}
```

I splitted the dataset A into 3 parts: 80% training, 15% validation and 5% test set.

After splitting, many classes have the big gap in difference. I used type 1 of data augmentation to increase the number of images in classes whose size is smaller than 1000 images to equal 1000 (upsampling method). Even though this does not make the size of each class be equal entirely, imbalance problem is partly solved. Augmentation techniques I utilized for type 1: rotation, width shift, height shift, shear, zoom, horizontal flip.



(Training dataset after augmentation)

2) Artificial Neural Network

a) The first model

The first model I tried is a simple neural network: input layer and 3 hidden layers. Most images in dataset A are of size 256x256x3 and I chose this size to resize all images in the dataset. The hidden units in each layer, including input are:

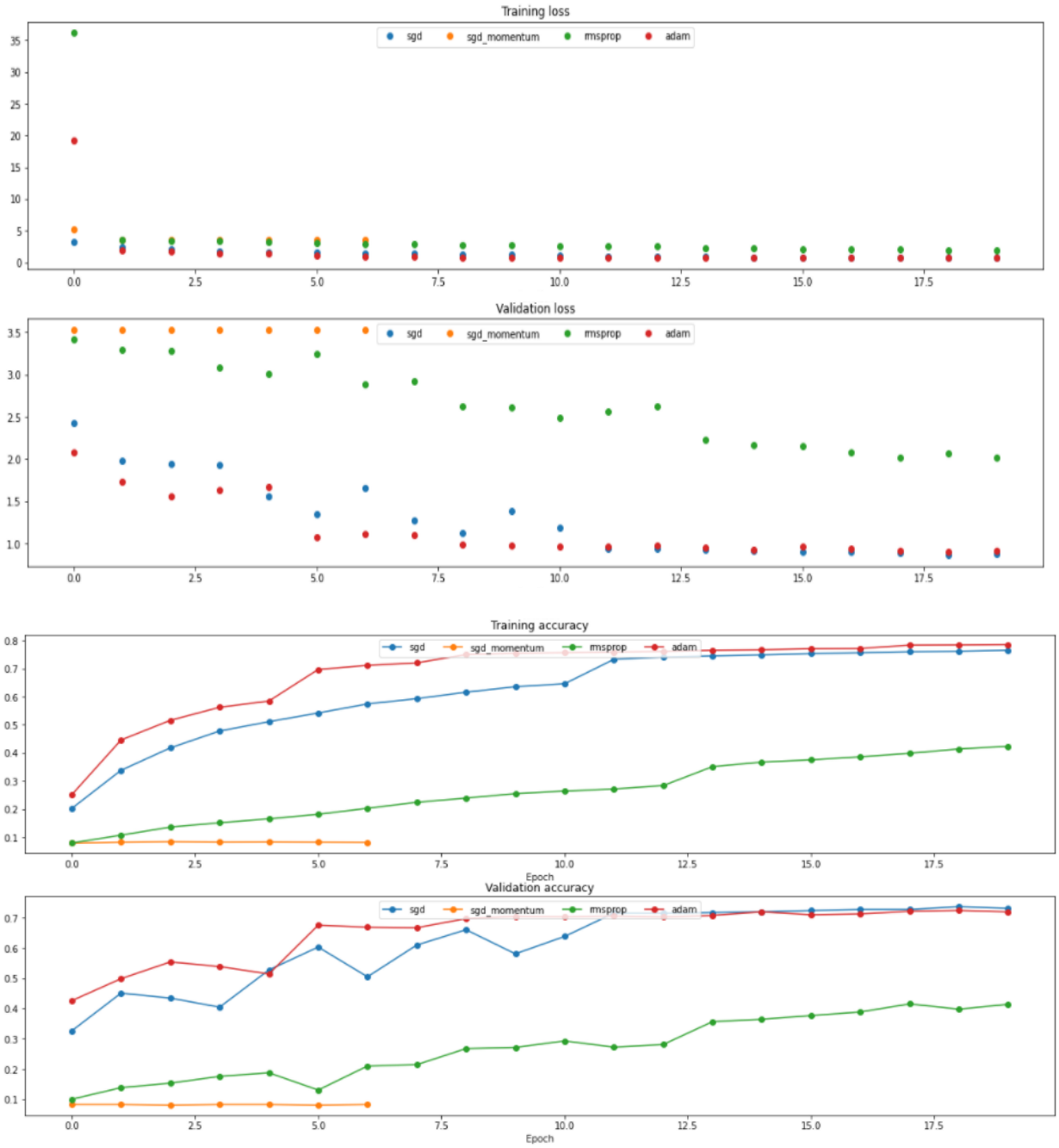
[196608, 1024, 512, 39]

I used relu activations for all immediate layers.

The initialization was Glorot Normal and 4 optimizers were tested.

The pseudo code:

```
hidden_layer_1 = 1024
hidden_layer_2 = 512
num_classes = 39
layers = [
    Flatten(input_shape=input_shape),
    Dense(hidden_layer_1, activation='relu', kernel_initializer=initializer),
    Dense(hidden_layer_2, activation='relu', kernel_initializer=initializer),
    Dense(num_classes, activation='softmax', kernel_initializer=initializer)
]
```


Training loss
sgd
sgd_momentum
rmsprop
adam
Validation loss
sgd
sgd_momentum
rmsprop
adam
Training accuracy
sgd
sgd_momentum
rmsprop
adam
Epoch
Validation accuracy
sgd
sgd_momentum
rmsprop
adam
Epoch

Information:

SGD: learning rate = 0.01 and performance:

```
Epoch 20/20
332/332 [==============================] - 239s 721ms/step - loss: 0.7671 - accuracy:
0.7707 - val_loss: 0.8772 - val_accuracy: 0.7311
```

SGD_momentum: learning rate = 0.01, momentum = 0.9 and performance:

```
Epoch 7/20
332/332 [==============================] - 209s 629ms/step - loss: 3.5242 - accuracy:
0.0806 - val_loss: 3.5251 - val_accuracy: 0.0829
```

RMSprop: learning rate = 0.001, rho = 0.9, momentum = 0, epsilon = 1e-7 and performance:

```
Epoch 20/20
332/332 [==============================] - 241s 726ms/step - loss: 1.9473 - accuracy:
0.4162 - val_loss: 2.0164 - val_accuracy: 0.4142
```

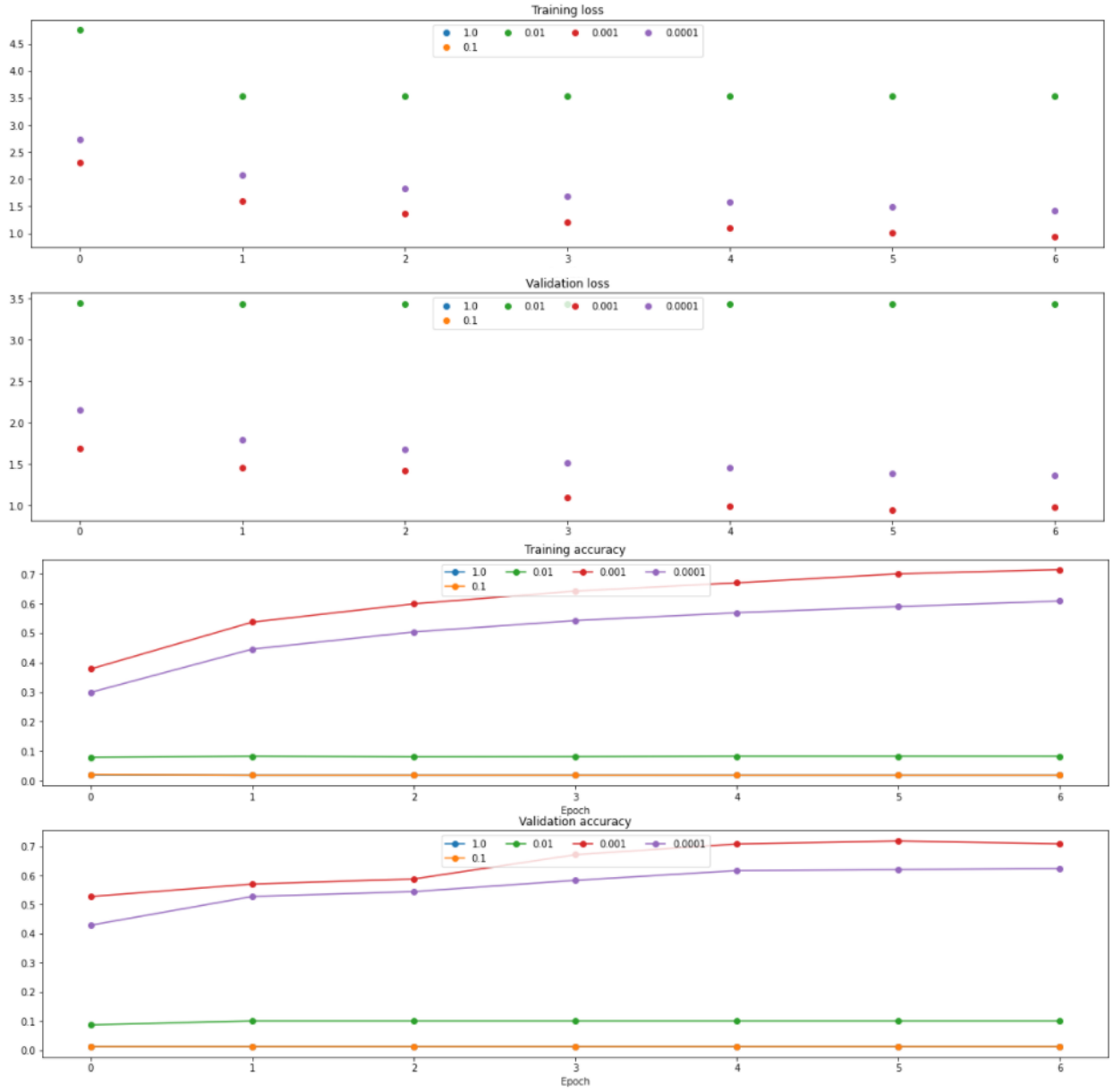
Adam: learning rate = 0.001, beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-7 and performance:

```
Epoch 20/20
332/332 [==============================] - 218s 656ms/step - loss: 0.7188 - accuracy:
0.7846 - val_loss: 0.9189 - val_accuracy: 0.7194
```

From the figures, SGD_momentum was early stopped due to not improving after many epochs (here, I set 6 epochs). Theoretically, SGD_momentum should perform better than SGD and a reason might be the initial learning rate of SGD_momentum being bad. RMSprop behaved bad, either.

b) The second model

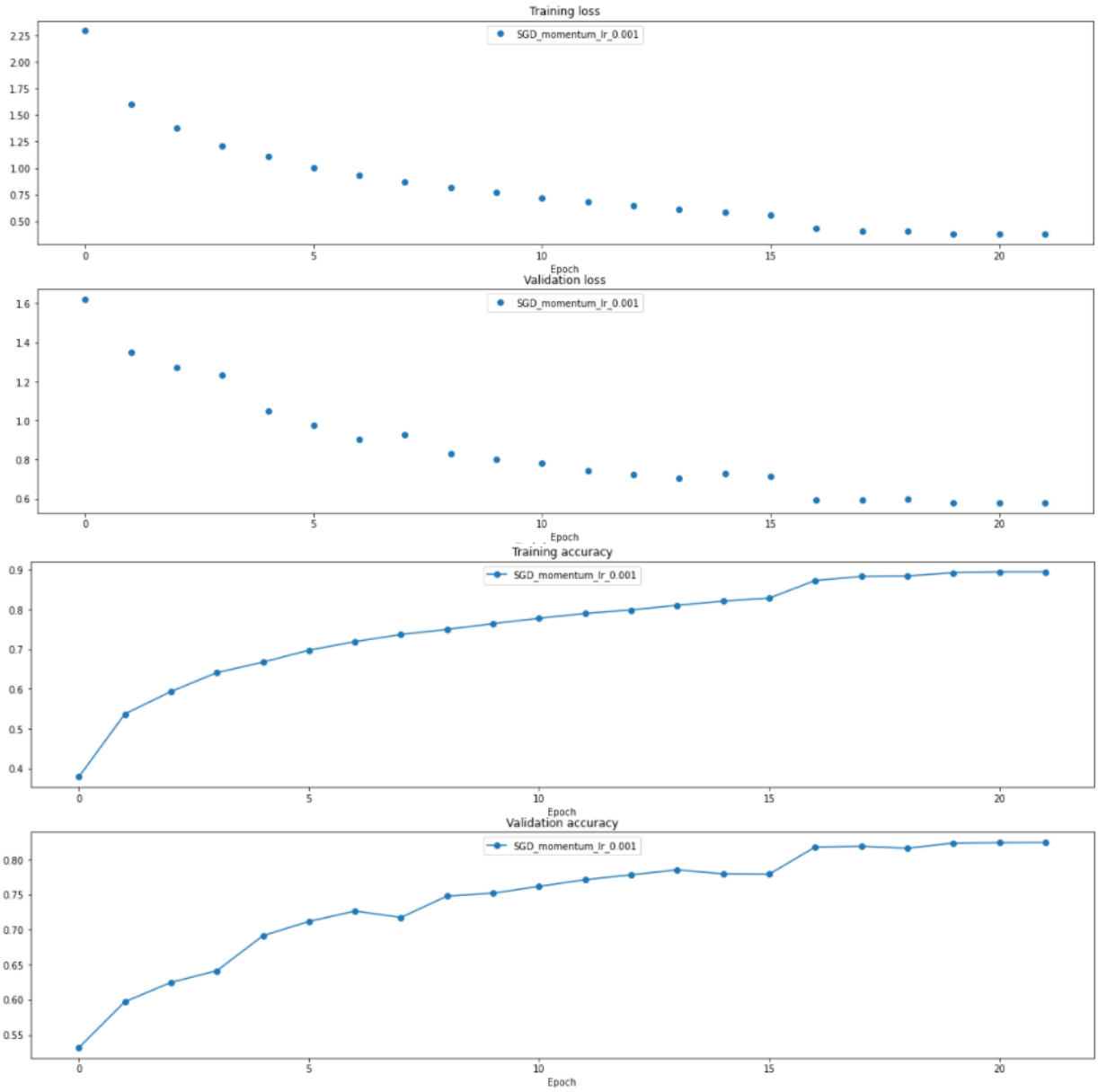
The second model was used to tune the learning rate of SGD_momentum.



Learning rate be 0.001 was the best.

c) The third model

I used the learning rate of 0.001 to run the model again




```
Epoch 22/22
415/415 [==============================] - 281s 678ms/step - loss: 0.3730 - accuracy:
0.8952 - val_loss: 0.5803 - val_accuracy: 0.8241
```

The result improved a lot, with the former SGD_momentum model using learning rate of 0.01, the model was early stopped. With learning rate of 0.001, the model learned much better and achieved high training accuracy (higher than that of SGD).

However, the model was almost overfit for all optimizers.

SGD: train accuracy: 0.7707, valid accuracy: 0.730, the gap is about 4%, slightly overfit.

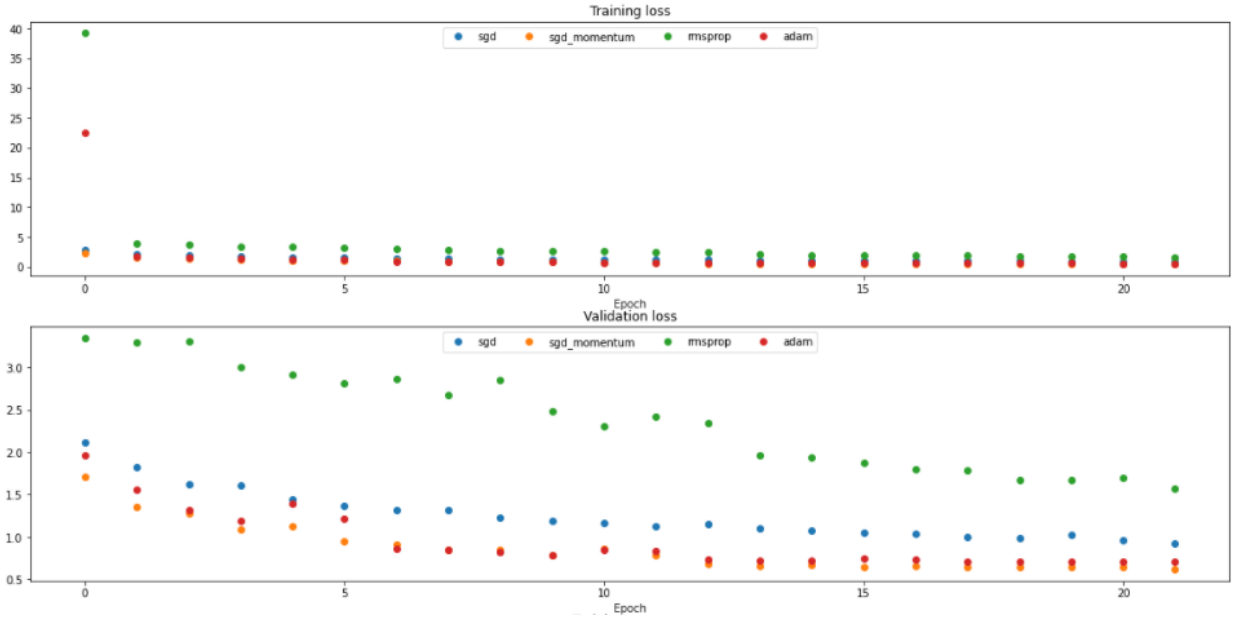
SGD_momentum: train accuracy: 0.8952, valid accuracy: 0.8241, the gap is about 7%, overfit.

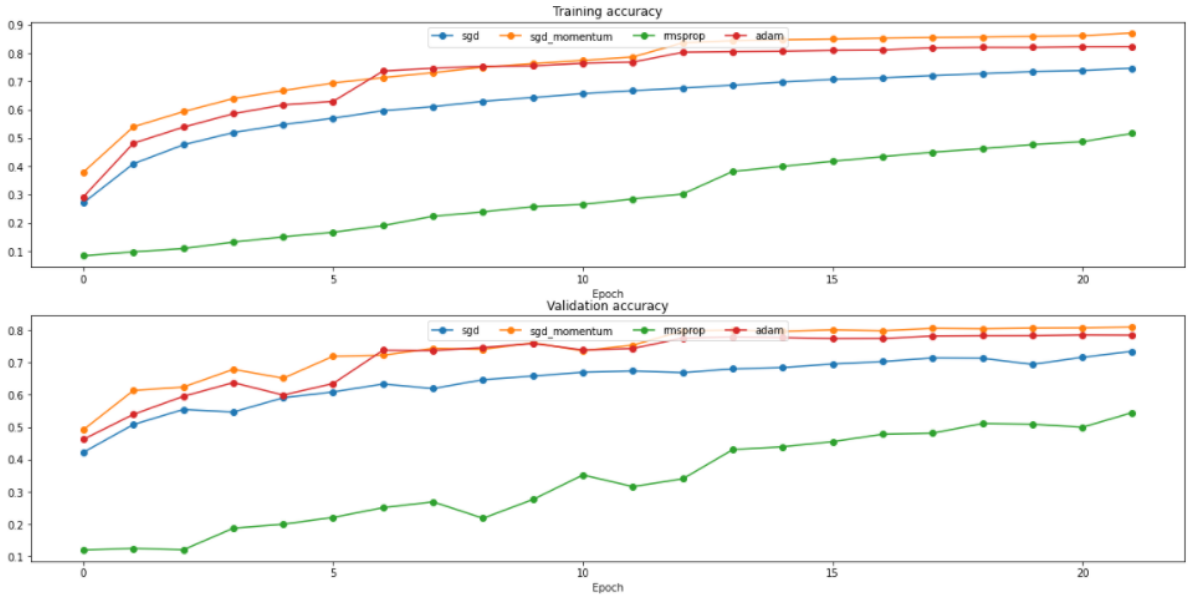
RMSprop: train accuracy: 0.4162, valid accuracy: 0.4142, the model is not overfit but the accuracy is low.

Adam: train accuracy: 0.7846, valid accuracy: 0.7194, the gap is about 6.5%, the model is overfit.

d) The fourth model

The fourth model utilized He Normal initialization, other hyperparameters remained the same.





SGD:

```
Epoch 22/22
415/415 [==============================] - 277s 667ms/step - loss: 0.8724 - accuracy:
0.7513 - val_loss: 0.9188 - val_accuracy: 0.7346
```

SGD_momentum:

```
Epoch 22/22
415/415 [==============================] - 248s 597ms/step - loss: 0.4622 - accuracy:
0.8685 - val_loss: 0.6211 - val_accuracy: 0.8097
```

RMSprop:

```
Epoch 22/22
415/415 [==============================] - 252s 608ms/step - loss: 1.6165 - accuracy:
0.5144 - val_loss: 1.5687 - val_accuracy: 0.5451
```

Adam:

```
Epoch 22/22
415/415 [==============================] - 317s 763ms/step - loss: 0.5753 - accuracy:
0.8219 - val_loss: 0.7042 - val_accuracy: 0.7847
```

It seemed the overfitting problem still occurred with this initilization. One way to reduce the stress of choosing initilization and prevent overfitting is to use Batch Normalization.

e) The fifth model

The fifth model inserted Batch Normalization layers into this network.

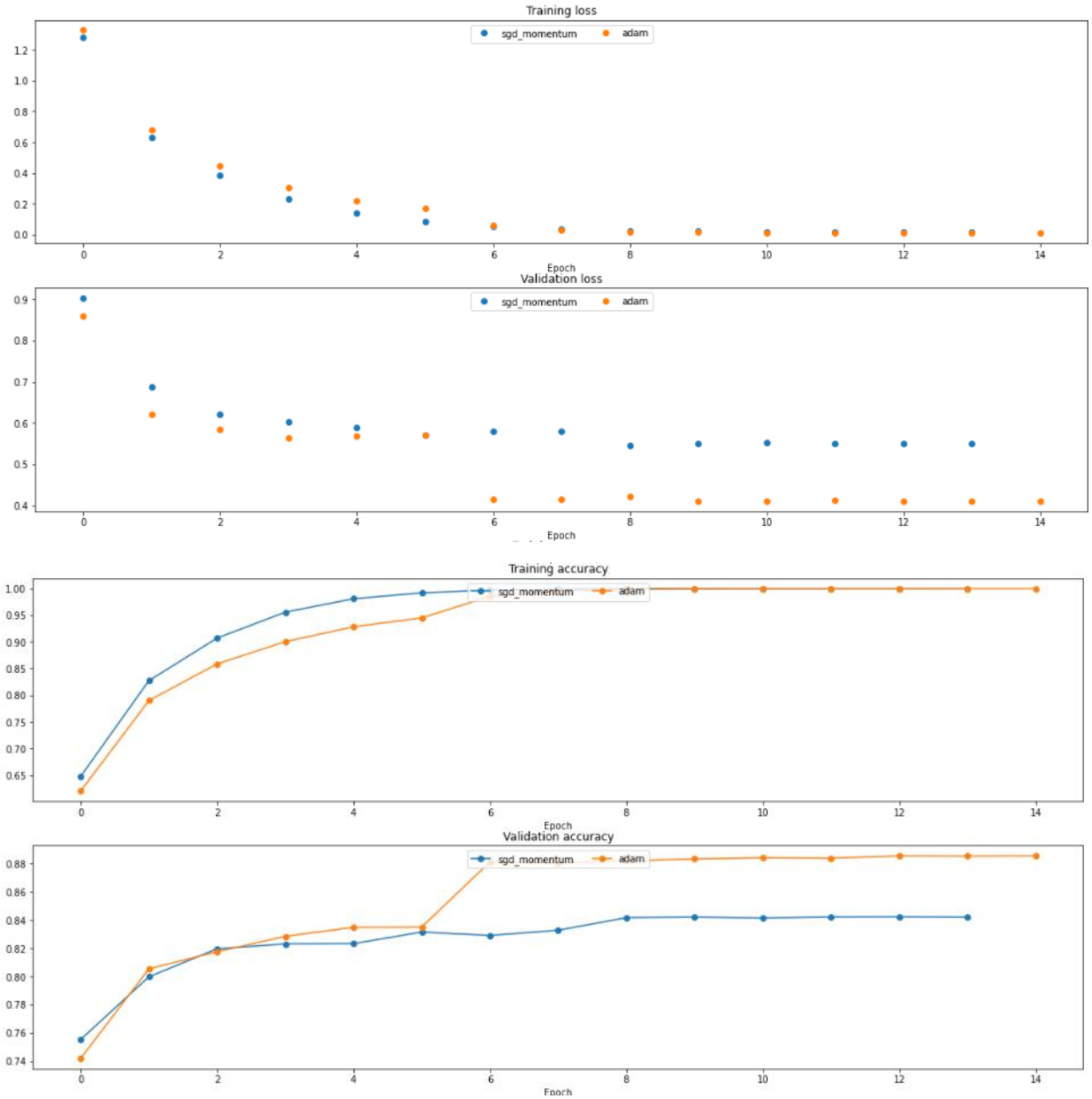
I only trained with Adam and SGD_momentum due to the time limitation of Kaggle kernel.

I inserted Batch Normalization after activations in this fifth model.

Pseudo code:

```
hidden_layer_1 = 1024
hidden_layer_2 = 512
num_classes = 39
layers = [
    Flatten(input_shape=input_shape),
    BatchNormalization(),
    Dense(hidden_layer_1, activation='relu', kernel_initializer=initializer),
    BatchNormalization(),
    Dense(hidden_layer_2, activation='relu', kernel_initializer=initializer),
    BatchNormalization(),
    Dense(num_classes, activation='softmax', kernel_initializer=initializer)
]
```

The result:



SGD_momentum:

```
Epoch 14/20
414/414 [==============================] - 237s 571ms/step - loss: 0.0174 - accuracy:
0.9999 - val_loss: 0.5496 - val_accuracy: 0.8419
```

Adam:

```
Epoch 15/20
414/414 [==============================] - 234s 566ms/step - loss: 0.0090 - accuracy:
0.9997 - val_loss: 0.4113 - val_accuracy: 0.8855
```

Although model was still overfit, the accuracy of training and validation was higher than the model without Batch Normalization layers. This proved the training process is much easier with Batch Normalization layers.

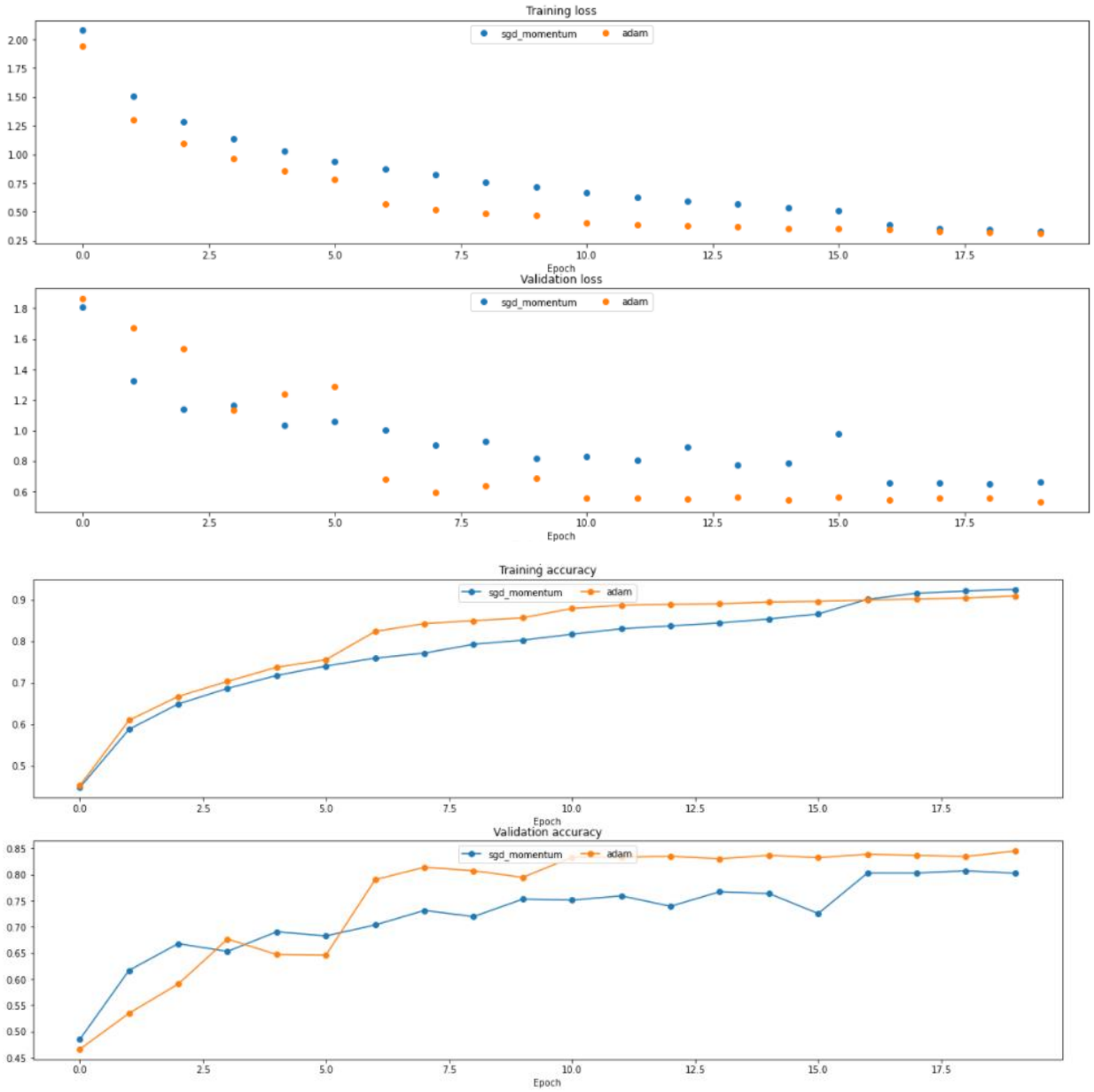
The drawback was the gap between the validation and training accuracy be bigger. This could be by the batch size or the size of dataset was still small.

f) The sixth model

The sixth model inserted Batch Normalization before activation layers.

Pseduo code:

```
hidden_layer_1 = 1024
hidden_layer_2 = 512
num_classes = 39
layers = [
    Flatten(input_shape=input_shape),
    BatchNormalization(),
    Dense(hidden_layer_1, kernel_initializer=initializer),
    BatchNormalization(),
    Activation('relu'),
    Dense(hidden_layer_2, kernel_initializer=initializer),
    BatchNormalization(),
    Activation('relu'),
    Dense(num_classes, activation='softmax', kernel_initializer=initializer)
]
```



SGD_momentum:

```
Epoch 20/20
414/414 [==============================] - 300s 725ms/step - loss: 0.3331 - accuracy:
0.9240 - val_loss: 0.6641 - val_accuracy: 0.8021
```

Adam:

```
Epoch 20/20
414/414 [==============================] - 295s 713ms/step - loss: 0.3095 - accuracy:
0.9091 - val_loss: 0.5342 - val_accuracy: 0.8447
```

Nothing improved, the accuracy even decresed.

Moreover, if we compare the run time:

Fifth model (model with BN after):

Run Time	12125.1 seconds

Sixth model (model with BN before):

Run Time	12665.8 seconds

The fifth model had higher accuracy, faster running time. This showed dataset A with this architecture performed better with Batch Normalizaton layers after activation layers.

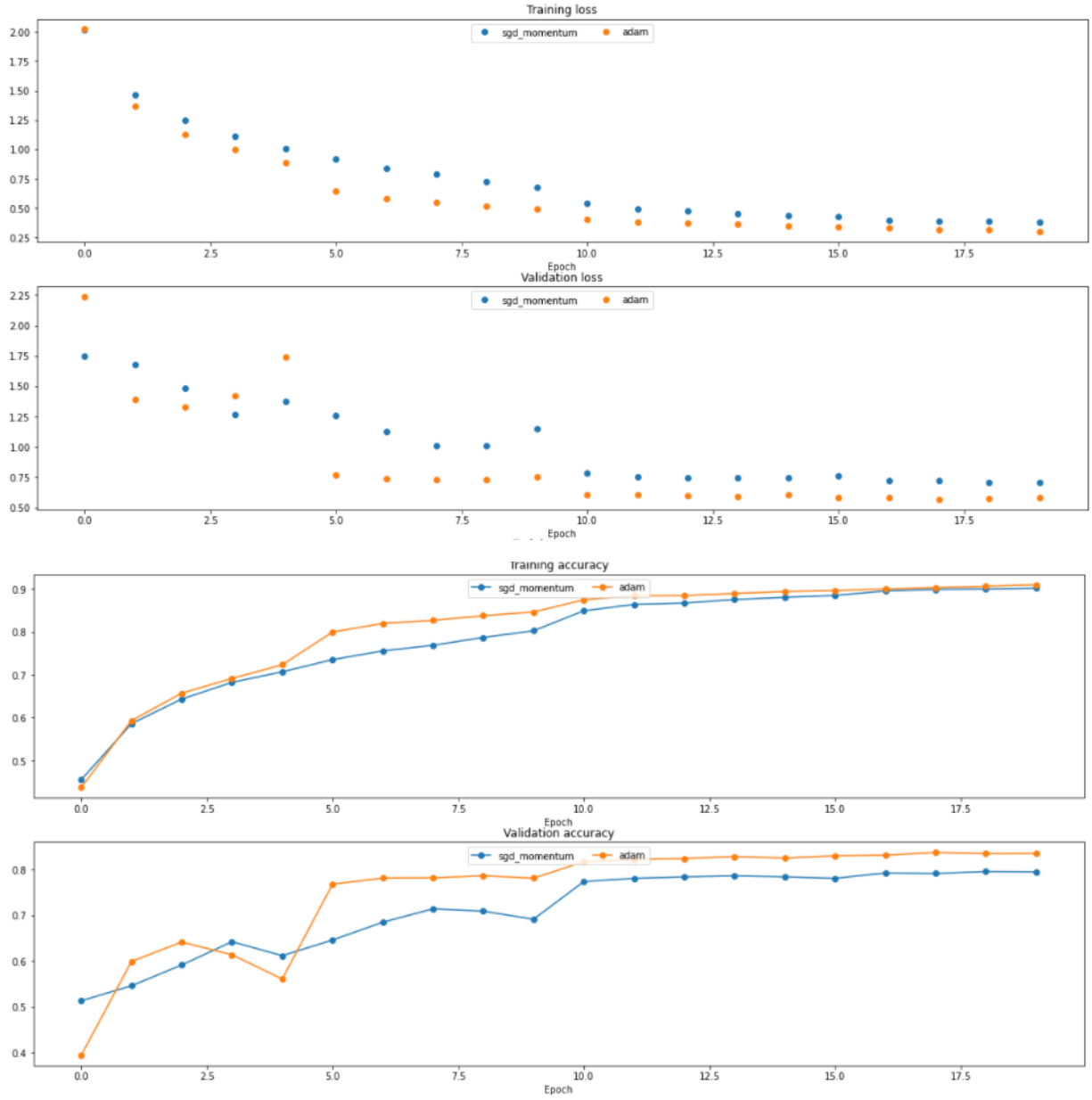
We come back to use fifth model. Now, the training process was easier due to Batch Normalizion, but one serious problem was the overfitting problem. Theoretically, Batch Normalization has regularization effect, which reduces overfitting, but in this case this effect was much week, and the situation was worse as the gap was expanded, the model was more overfitting. As mentioned above, batch size or not enough dataset might be factors.

If not enough dataset is a reason, data augmentation on-the-fly (type 2) is a solution.

g) The seventh model

The seventh model: Fifth model with data augmentation

Transformations: vertical flip and brightness change.



SGD_momentum:

```
Epoch 20/20
414/414 [==============================] - 300s 723ms/step - loss: 0.3828 - accuracy:
0.9007 - val_loss: 0.7077 - val_accuracy: 0.7943
```

Adam:

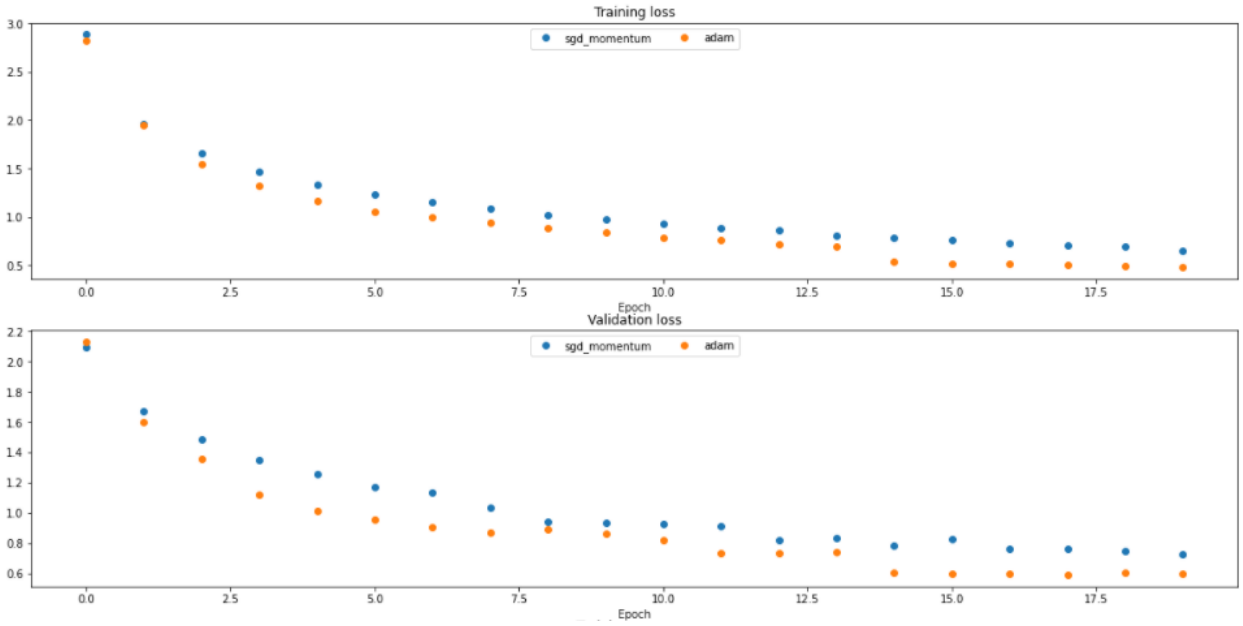
```
Epoch 20/20
414/414 [==============================] - 306s 740ms/step - loss: 0.2998 - accuracy:
0.9085 - val_loss: 0.5828 - val_accuracy: 0.8342
```

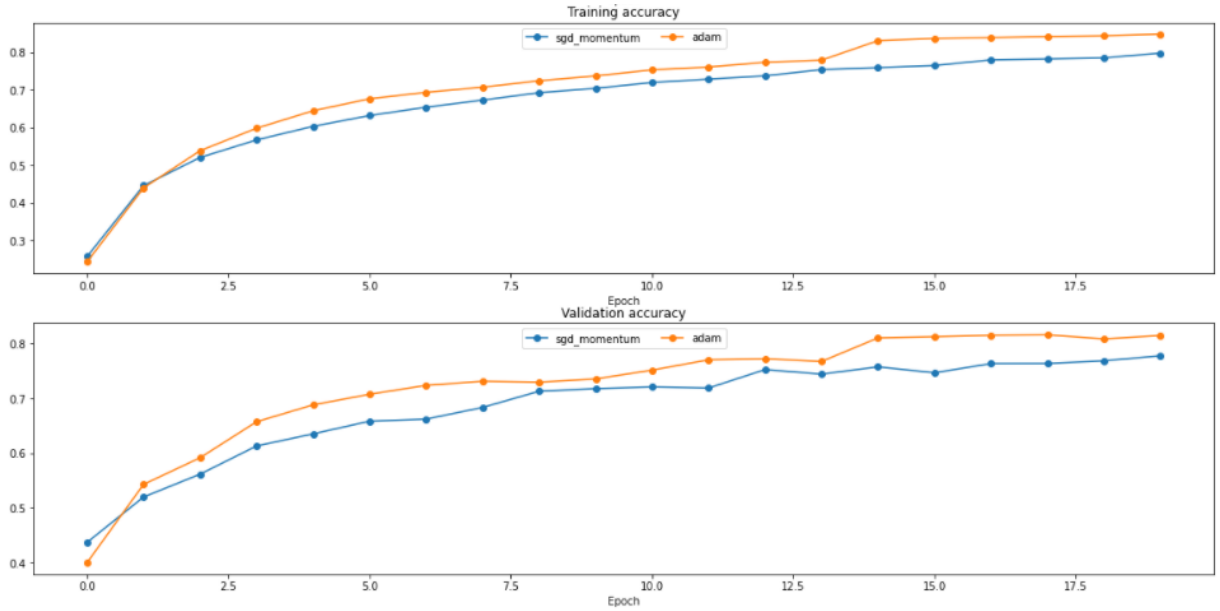
Data augmentation did not help much.

Batch size may be a second reason. Layer Normalization is one solution.

h) The eighth model

The eighth model: Same augmentation as the seventh model, substituted all Batch Normalization with Layer Normalization





SGD_momentum:

```
Epoch 20/20
415/415 [==============================] - 350s 843ms/step - loss: 0.6581 - accuracy:
0.7955 - val_loss: 0.7227 - val_accuracy: 0.7774
```

Adam:

```
Epoch 20/20
415/415 [==============================] - 362s 873ms/step - loss: 0.4715 - accuracy:
0.8496 - val_loss: 0.5945 - val_accuracy: 0.8145
```

The overfitting situation improved a lot. The gap between the training accuracy and validation acuracy reduced to about 1.8% for SGD_momentum and approximately 3.5% for Adam.

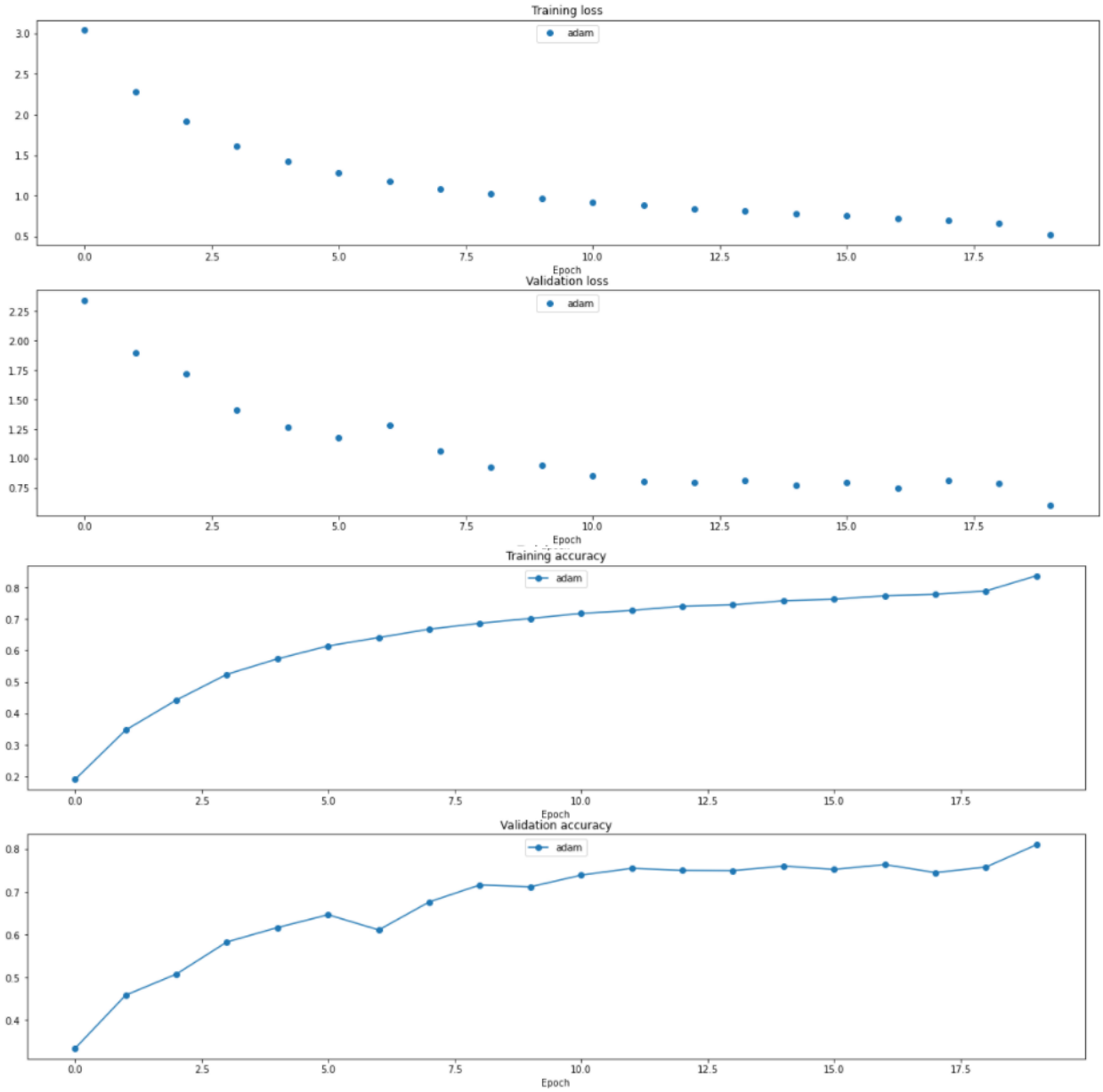
From the plots, Adam always performed better than SGD_momentum and since training one model with two algorithms took much time, I chose to continue training with only Adam.

There is one hyperparameter should be considered: number of layers.

i) The ninth model

The ninth model: number of layers increased: one input layer and four hidden layers:

[196608, 1024, 512, 64, 39]



Adam:

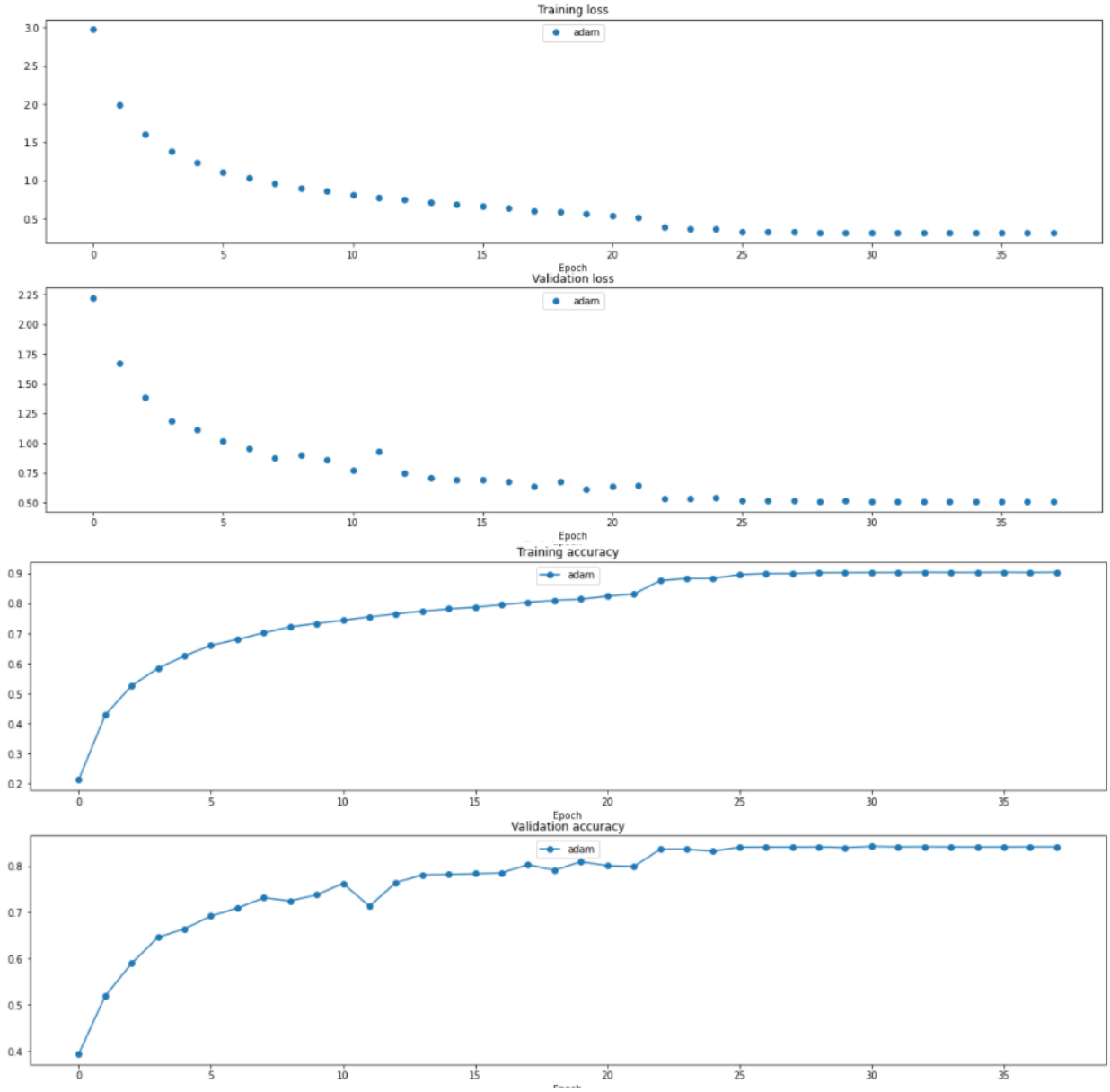
```
Epoch 20/20
415/415 [==============================] - 298s 718ms/step - loss: 0.5428 - accuracy:
0.8295 - val_loss: 0.6029 - val_accuracy: 0.8104
```

Not only did the accuracy not improve, the loss also increased.
This demonstrated increasing more layers may not improve the performance of the model, even worsen.

j) The tenth model

The tenth model: The plots of the ninth model reflect that model was still increasing in accuracy. This shows that the model still has potential to learn more. In tenth model, number of epochs increases from 20 to 40. The layers are:

[196608, 1024, 512, 39]




```
Epoch 38/40
415/415 [==============================] - 346s 833ms/step - loss: 0.3191 - accuracy:
0.9017 - val_loss: 0.5092 - val_accuracy: 0.8414
```

The training loss and validation loss both decreased, training accuracy and validation accuracy both increased, which is a good signal, although overfitting problem occurred again. At this time, we can penalize the loss function by adding L1 or L2 regularizers, but this does not help increase the accuracy much.

3) Convolution neural network

Using fully-connected layers, we have to flatten the image, which loses spatial information of image. Convolution neural network fixes this problem by using a kernel to slide over local regions and do convolve in each region. This ensures spatial features to be considered in the network.

a) The eleventh model

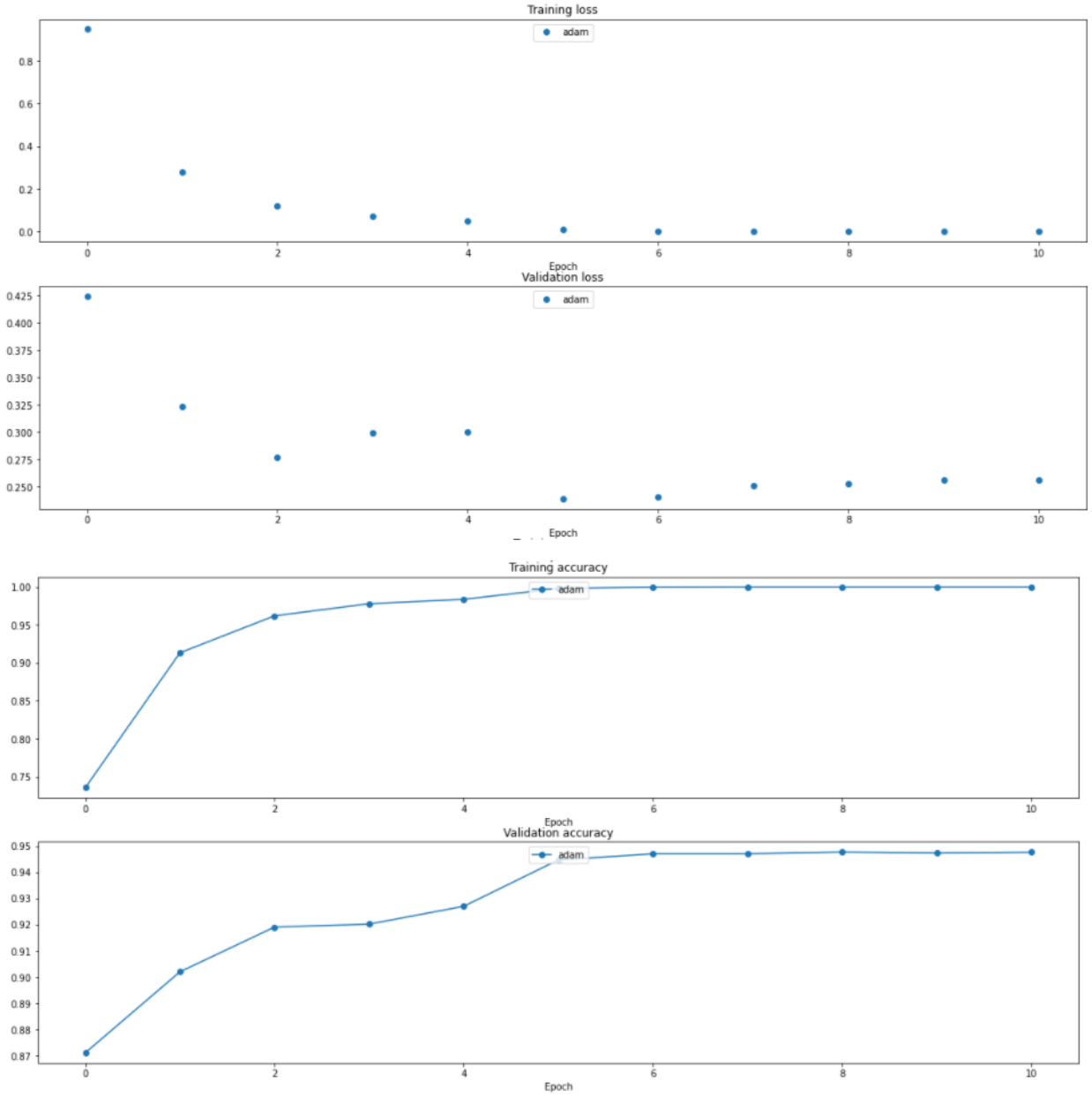
The eleventh model was a simple convolution neural network.

Pseudo code:

```
    num_filter1 = 64
    num_filter2 = 128
    num_filter3 = 512
    num_classes = 39
    layers = [
            Conv2D(num_filter1, (3, 3), input_shape=input_shape, padding='same', activatio
n='relu'),
            MaxPool2D(pool_size=(2, 2)),
            Conv2D(num_filter2, (3, 3), padding='same', activation='relu'),
            MaxPool2D(pool_size=(2, 2)),
            Conv2D(num_filter3, (3, 3),  padding='same', activation='relu'),
            MaxPool2D(pool_size=(2, 2)),
            Flatten(),
            Dense(num_classes, activation='softmax')
    ]
```

input shape = (256, 256, 3)

And the results:

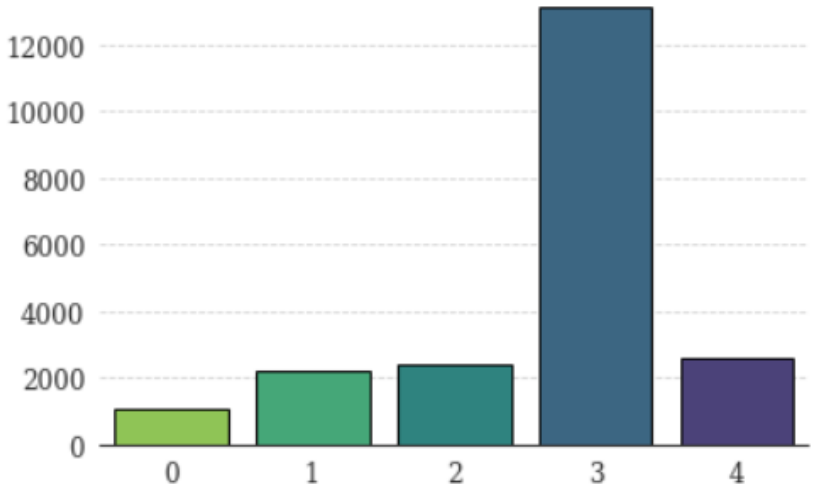



```
Epoch 11/40
415/415 [==============================] - 250s 600ms/step - loss: 4.4431e-04 - accurac
y: 1.0000 - val_loss: 0.2560 - val_accuracy: 0.9475
```

The model was early stopped, but it experienced a high jump in accuracy. Convolution neural network is much more efficient than fully-connected network when comes to image problems.

Several methods can be utilized to overcome overfitting problem of convolution nets like fully-connected ones. However, I stopped tuning my model here on dataset A as this dataset is so easy for convolution network now.

Dataset B is more challenging as it contains detailed background information and much noise. Its distribution:



Although class 3 is dominant, I did not augment data the same way I did with dataset A. Instead of dividing transformation techniques into type 1 and type 2 augmentation separately, I used all of them for type 2. In practice, this approach is more common.

Most of images have size of 800x600 and I chose to resize all to 512x512, since convolution works well with square images.

The eleventh model was used in dataset B. While we had a quite good result on dataset A, this model exceeded the max allowed execution duration on dataset B. Kaggle only allows one notebook to run in 9 hours, and more than that the notebook is shutdown.

Dataset B is much trickier than dataset A and there are many reasons that a network runs slow and cannot finish the training process. In this case, it can be the size I choose to resize is too big or some default hyperparameters of TensorFlow library are initialized unreasonably for this dataset.

b) The twelfth model (resnet-18)

The twelfth model used the residual block of Resnet architecture. Resnet was a breakthrough at the time it appeared, introducing the definition of "shortcut connection". In the original paper [9], the authors said deep networks are difficult to train and adding more layers does not mean better performance. They hypothesized that "it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping" [9], which is the cleverest idea of Resnet architecture.

The residual block:

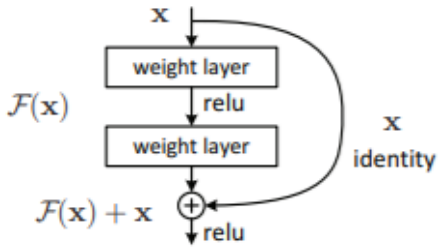
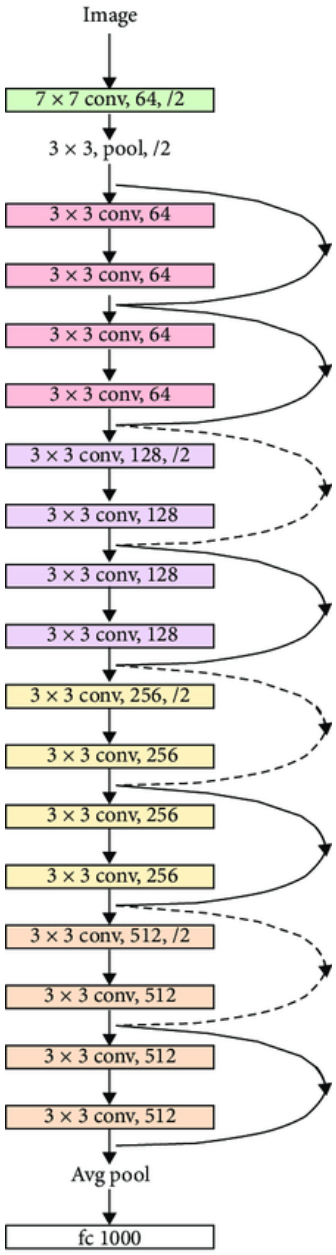
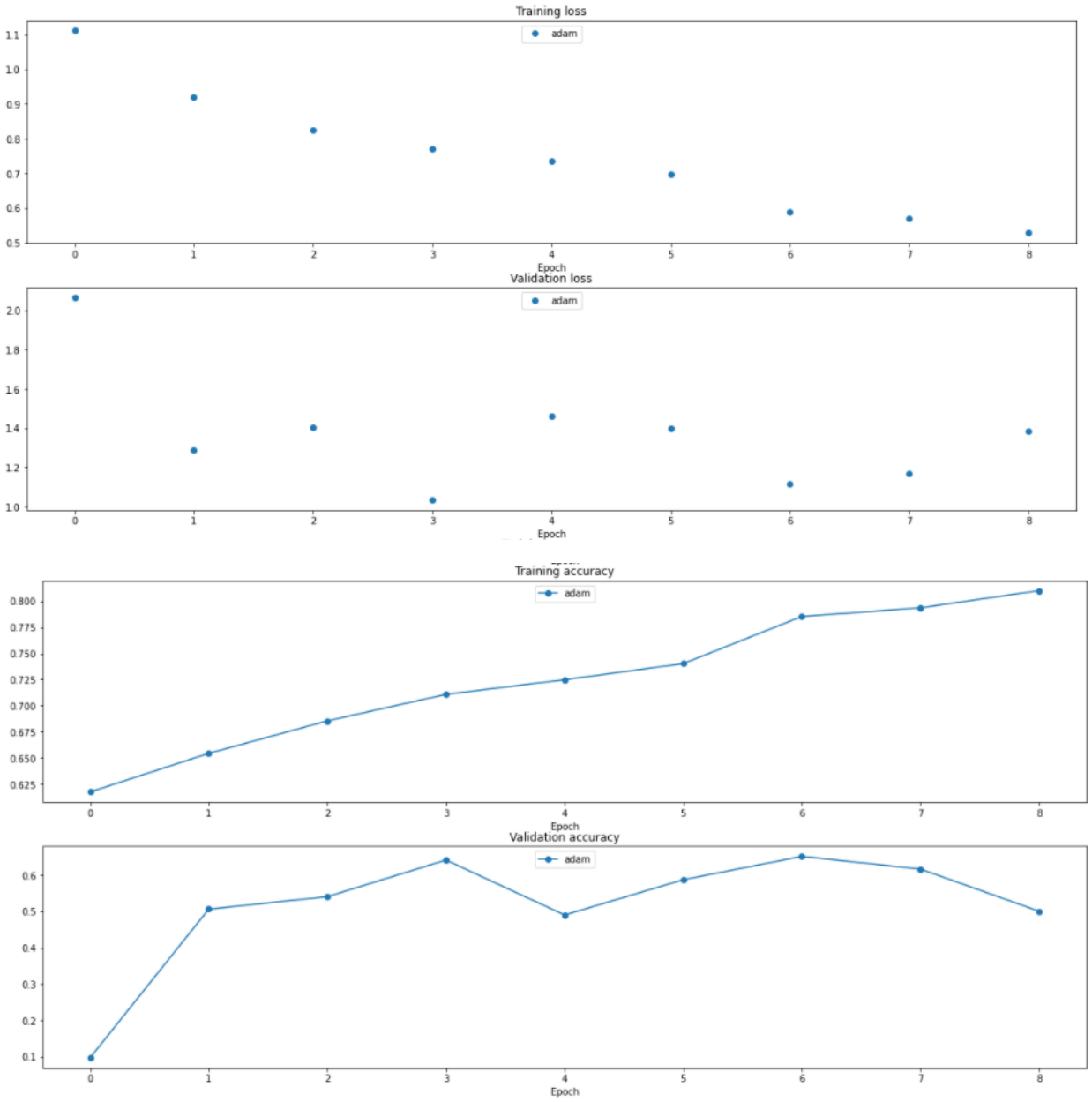


Figure 2. Residual learning: a building block.

The twelfth model **is resnet-18 architecture**:






```
Epoch 9/25
264/264 [==============================] - 1197s 5s/step - loss: 0.5385 - accuracy: 0.8
033 - val_loss: 1.3854 - val_accuracy: 0.5002
```

The first improvement is that the model finished training process while the previous CNN model (the eleventh model) did not, although it was early stopped. The model being overfitting is a good sign at start since this proves the model learned something. Now, we have to tune hyperparameters as usual to get better result, but as can be seen from previous models, tuning model from scratch is an exhaustive work and time consuming.

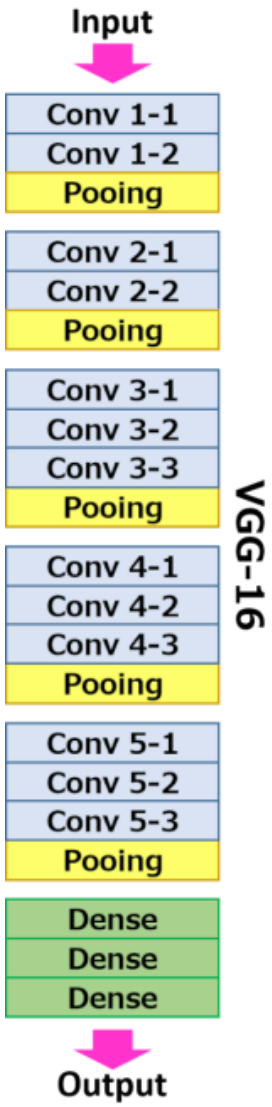
In practice, in order to have good results from this hard dataset, transfer learning is preferable and more efficient.

4) Transfer learning

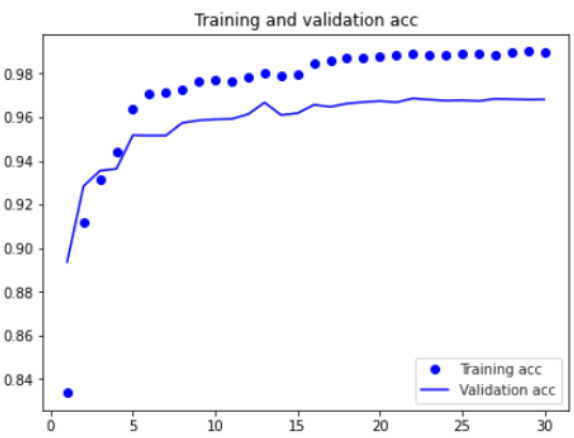
Transfer learning saves time for training as model does not have to learn all parameters in network. A set of parameters are trained before on a larger dataset (etc. ImageNet) and model learns only parameters of new layers at the top of the network. This happens as follow: freeze the weights before the "classification-layer" part, add new layers and tune hyperparameters on them.

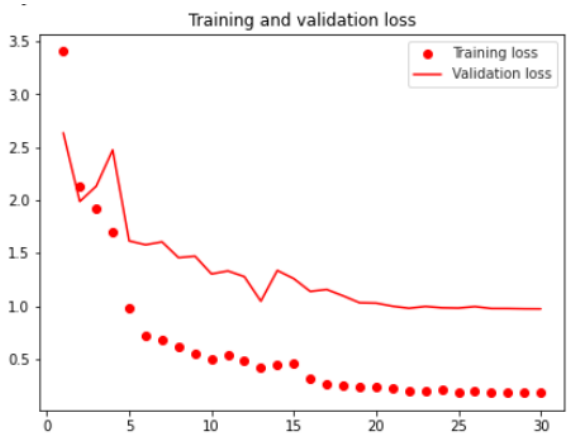
a) VGG-16

VGG-16 is a simple architecture network that is not so much different from the eleventh model, except it is deeper.



First, let try on dataset A (39 classes).





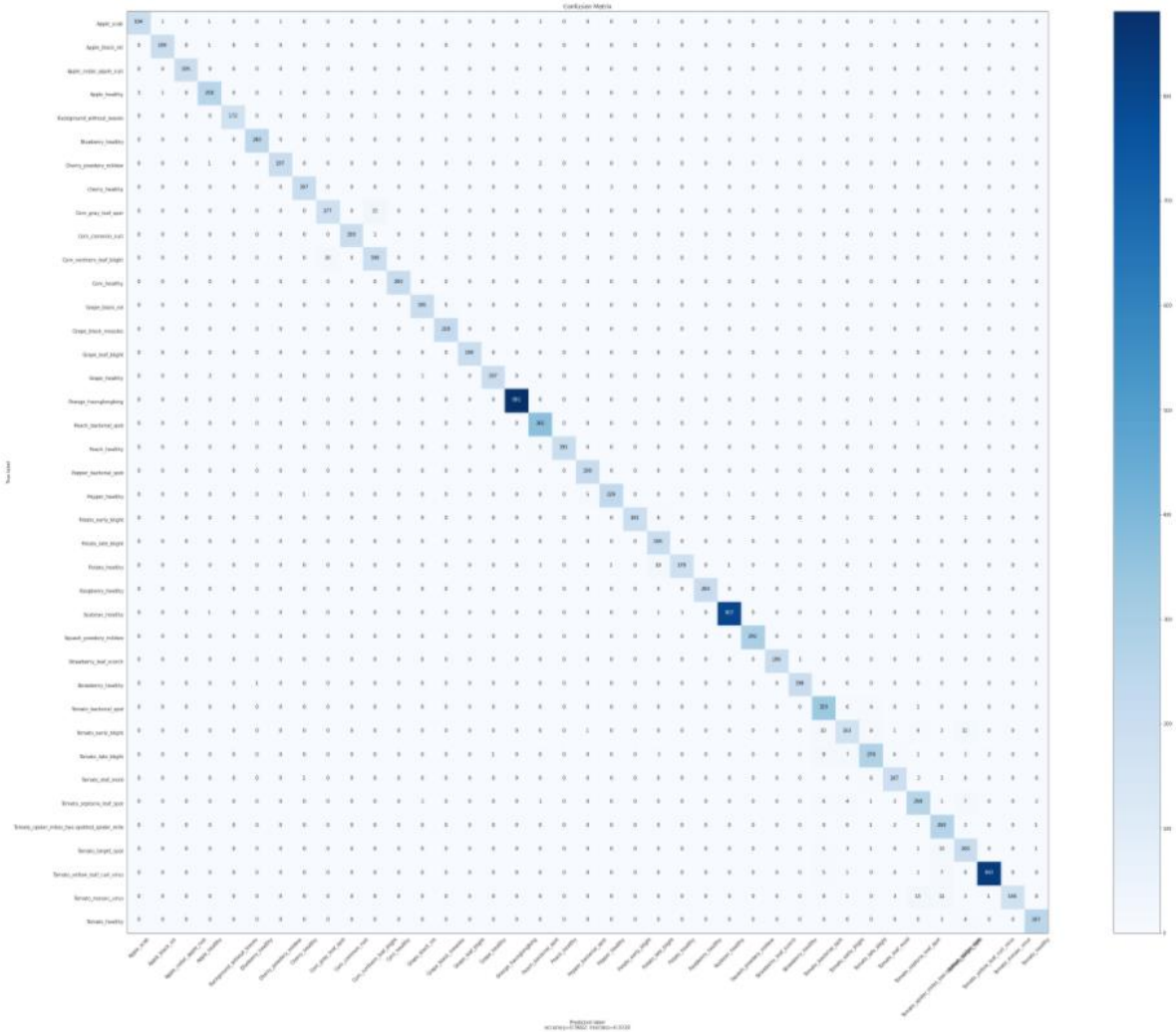

```
Epoch 30/30
332/332 [==============================] - 842s 3s/step - loss: 0.1837 - accuracy: 0.99
01 - val_loss: 0.9737 - val_accuracy: 0.9682
```

The result is quite good. Usually, when the dataset is imbalanced, precision and recall metrics are used to evaluate the performance. The classification report summaries them:

	precision	recall	f1-score	support
Apple_scab	0.98	0.97	0.98	200
Apple_black_rot	0.99	0.99	0.99	200
Apple_cedar_apple_rust	1.00	0.97	0.99	200
Apple_healthy	0.98	0.98	0.98	263
Background_without_leaves	1.00	0.95	0.97	182
Blueberry_healthy	1.00	1.00	1.00	240
Cherry_powdery_mildew	0.99	0.98	0.99	200
Cherry_healthy	0.99	0.98	0.99	200
Corn_gray_leaf_spot	0.94	0.89	0.91	200
Corn_common_rust	1.00	0.99	1.00	200
Corn_northern_leaf_blight	0.88	0.95	0.92	200
Corn_healthy	1.00	1.00	1.00	200
Grape_black_rot	0.97	0.97	0.97	200
Grape_black_measles	0.98	0.99	0.98	221
Grape_leaf_blight	1.00	0.99	1.00	200
Grape_healthy	0.99	0.98	0.99	200
Orange_haunglongbing	1.00	1.00	1.00	881
Peach_bacterial_spot	0.96	0.99	0.98	367
Peach_healthy	1.00	0.97	0.99	200
Pepper_bacterial_spot	0.97	1.00	0.99	200
Pepper_healthy	0.98	0.97	0.98	236
Potato_early_blight	1.00	0.96	0.98	200
Potato_late_blight	0.87	0.98	0.92	200
Potato_healthy	0.99	0.89	0.94	200
Raspberry_healthy	1.00	1.00	1.00	200
Soybean_healthy	1.00	0.99	0.99	814
Squash_powdery_mildew	1.00	1.00	1.00	293
Strawberry_leaf_scorch	0.99	0.99	0.99	200
Strawberry_healthy	0.99	0.99	0.99	200
Tomato_bacterial_spot	0.90	0.97	0.93	340
Tomato_early_blight	0.87	0.81	0.84	200
Tomato_late_blight	0.91	0.91	0.91	305
Tomato_leaf_mold	0.94	0.94	0.94	200
Tomato_septoria_leaf_spot	0.90	0.91	0.91	283
Tomato_spider_mites_two-spotted_spider_mite	0.86	0.97	0.91	268
Tomato_target_spot	0.86	0.89	0.88	224
Tomato_yellow_leaf_curl_virus	1.00	0.98	0.99	857
Tomato_mosaic_virus	1.00	0.83	0.91	200
Tomato_healthy	0.98	0.97	0.98	254
accuracy			0.97	10628
macro avg	0.97	0.96	0.96	10628
weighted avg	0.97	0.97	0.97	10628

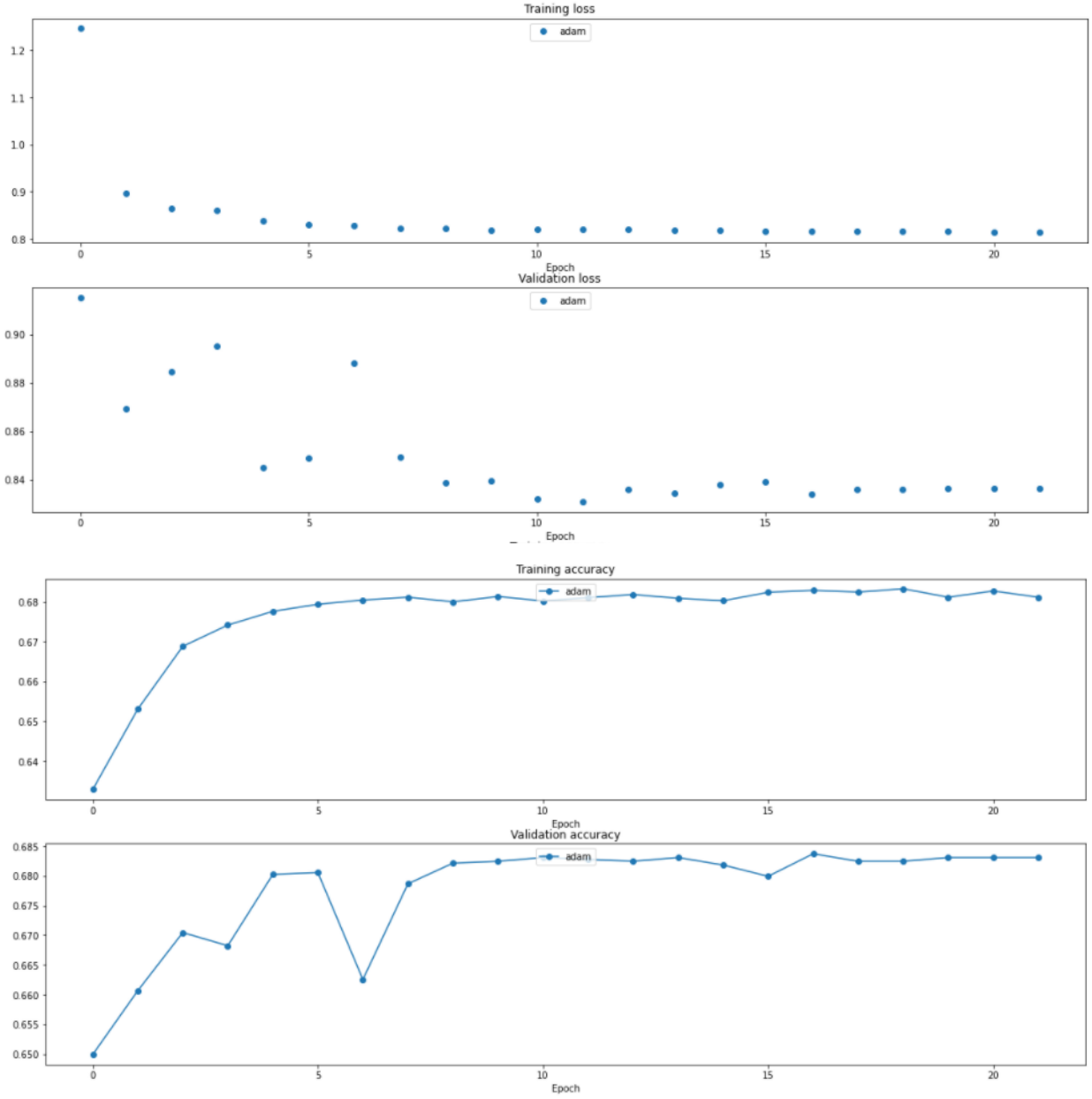
The classes with high precision and recall are considered well classified. From the report, Tomato___Early_blight, Tomato___Spider_mites Two-spotted_spider_mite, Tomato___Target_Spot are classes that model had difficulty to classify.

Confusion matrix is also a good tool for evaluation. X-axis is predicted class; Y-axis is true class.



It is hard to see since there as 39 classes. The diagonal is the number of images correctly classified. The model performed well.

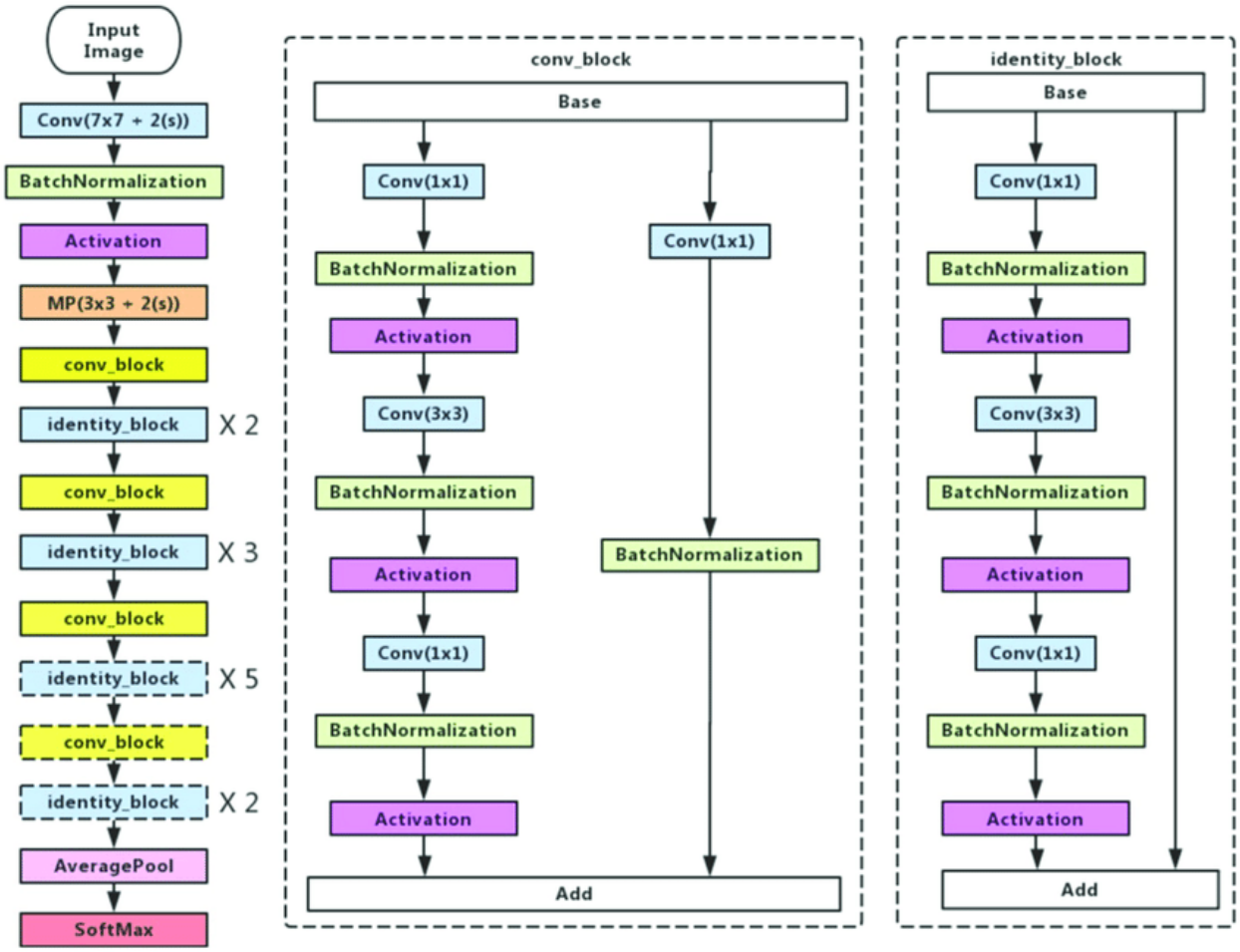
Dataset A is too easy, the results training on dataset B (5 classes):

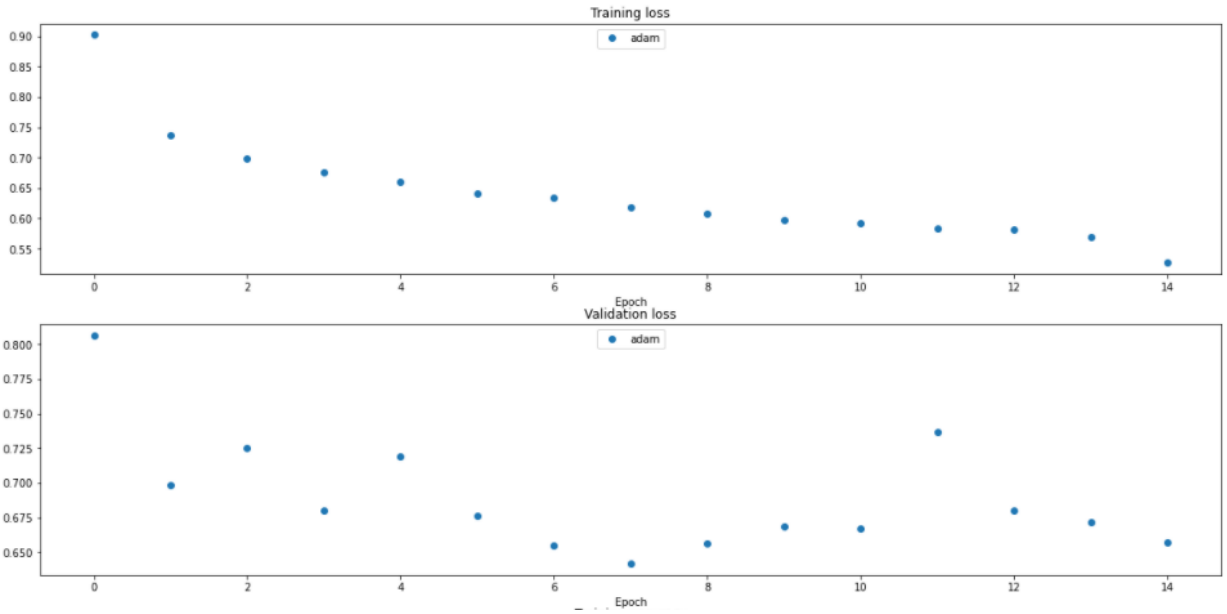


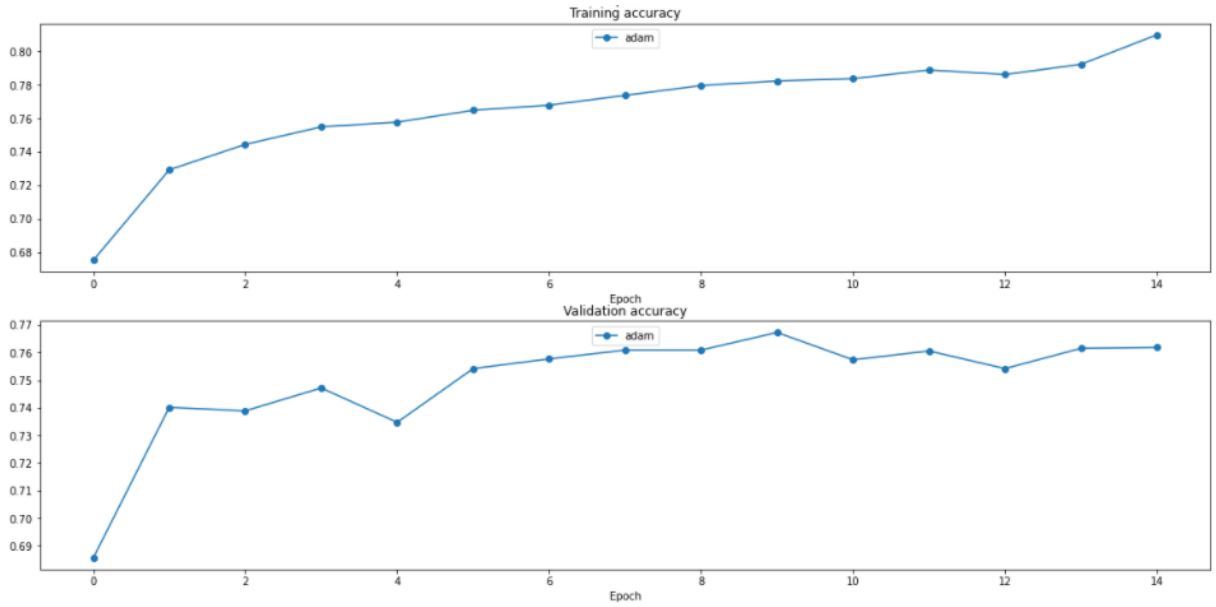

```
Epoch 22/25
562/562 [==============================] - 1353s 2s/step - loss: 0.8155 - accuracy: 0.6
784 - val_loss: 0.8362 - val_accuracy: 0.6831
```

The model was not overfitting but the accuracy is low, which proves the learning ability of VGG-16 is limited. As expected, more complex architecture should be considered on dataset B.

b) Resnet-50


Input Image
Conv(7x7 + 2(s))
BatchNormalization
Activation
MP(3x3 + 2(s))
conv_block
identity_block
X 2
conv_block
identity_block
X 3
conv_block
identity_block
X 5
conv_block
identity_block
X 2
AveragePool
SoftMax
conv_block
Base
Conv(1x1)
BatchNormalization
Activation
Conv(3x3)
BatchNormalization
Activation
Conv(1x1)
BatchNormalization
Activation
Conv(1x1)
BatchNormalization
Add
identity_block
Base
Conv(1x1)
BatchNormalization
Activation
Conv(3x3)
BatchNormalization
Activation
Conv(1x1)
BatchNormalization
Activation
Add


Training loss
adam
Epoch
Validation loss
adam
Epoch




```
Epoch 15/15
281/281 [==============================] - 512s 2s/step - loss: 0.5251 - accuracy: 0.81
19 - val_loss: 0.6568 - val_accuracy: 0.7618
```

This is a big improvement: loss decreases, accuracy increases. This demonstrates Resnet architecture is more powerful than VGG architecture.

c) EfficientNet B0

Efficient Net introduced a clever idea that outperformed many architectures at that time: "Compound Scaling". Before EfficientNet, the most common ways to scale up convolution networks was either depth (number of layers), width (number of channels) or image resolution (image size).

Efficient Net performs Compound Scaling, which scales three dimensions and balances between them.

The figure from the original paper [10]:

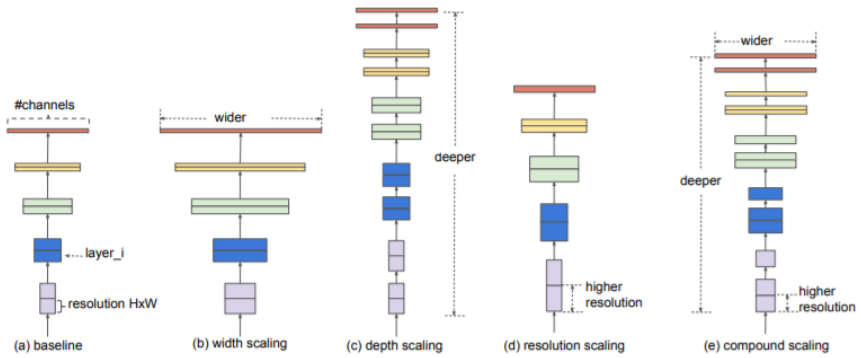
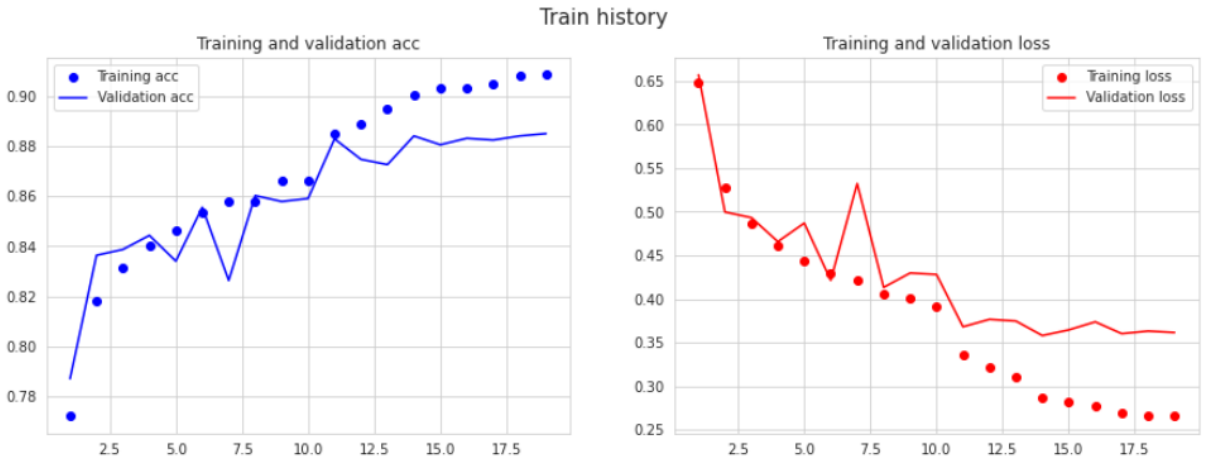


Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

The idea of using EfficientNet B0 for dataset B is from a competitor on this Kaggle competition [11].




```
2140/2139 [==============================] - 1609s 752ms/step - loss: 0.2655 - acc: 0.9
085 - val_loss: 0.3616 - val_acc: 0.8850
```

D. Deploy on Flask

The main purpose of this project is to build a web app that helps users recognize different plant diseases. Flask is a micro web framework that was created to build up simple web applications.

Many ideas in this part are from [12]

These are the directories for a web app:

```
Image-Classification-App
 |-app_helper.py
 |-static
 |  |-css
 |  |  |-main.css
 |  |-uploads
 |-templates
 |  |-layout.html
 |  |-upload.html
 |  |-index.html
 |-app.py
```

The **Image-Classification-App** is the main flask directory, in which we have:

- **app.py**: This file is the root of our Flask application. We define all the routes and functions to perform for each action.
- **app_helper.py**: This file contains the helper function where we define the model and get predictions of the input image, and return those predictions.
- **templates**: This directory is recognized to contain the HTML files by Flask. For our app, we will define the following HTML files:
    - **layout.html**: This file defines the layout of the pages, like the Navigation bar, which should be the same for the pages. Thus, all such info will be coded in this file, and this file will be imported in other HTML files.
    - **index.html**: This is the home page, where the user is asked to upload the image, whose class should be predicted using our pre-trained model.
    - **upload.html**: This page displays the predicted classes along with the probabilities and the uploaded image.
- **static**: This directory contains static files such as JS, CSS, and images by needed by our Flask app.
    - **css**: This directory contains the CSS styling of the HTML files. For our app, we shall define **main.css**. The **main.css** file contains the stylings. We link this in HTML files to apply the stylings defined in this **main.css** file. In our app, we shall link this to the **layout.html**, the base HTML file which we will be importing in all the other HTML files. Thus, these stylings are made applicable to all the HTML files in our app.
    - **uploads**: This directory is used to store the image uploaded to our app. This stored image will be used to display it along with the predictions.

Some examples:

PLANT DISEASE CLASSIFICATION APP

{'Class: 2': 'Cassava Green Mottle'}

Upload another?

Choose File No file chosen

Submit

PLANT DISEASE CLASSIFICATION APP

{'Class: 4': 'Healthy'}

Upload another?

Choose File No file chosen

Submit

PLANT DISEASE CLASSIFICATION APP

{'Class: 3': 'Cassava Mosaic Disease'}

Upload another?

Choose File No file chosen

Submit

E. Conclusion

In this project, I tried several deep learning models and evaluated them on both dataset A and dataset B. On dataset A, the model classified quite well, while on dataset B there are potentials to improve more in future. Many architectures such as Inception, DenseNet, HighwayNet … have not been tried. A simple web app was built by Flask successfully and user-friendly, on which the model was deployed.

F. Reference

[1]: J, ARUN PANDIAN; GOPAL, GEETHARAMANI (2019), "Data for: Identification of Plant Leaf Diseases Using a 9-layer Deep Convolutional Neural Network", Mendeley Data, V1, doi: 10.17632/tywbtsjrjv.1

[2]: David. P. Hughes, Marcel Salathe. An open access repository of images on plant health to enable the development of mobile disease diagnostics. arXiv:1511.08060

[3]: Kaggle.com. 2021. *Cassava Leaf Disease Classification | Kaggle*. [online] Available at: <https://www.kaggle.com/c/cassava-leaf-disease-classification/overview>

[4]: Cs231n.github.io. 2021. *CS231n Convolutional Neural Networks for Visual Recognition*. [online] Available at: <https://cs231n.github.io/neural-networks-2/>

[5]: Paperspace Blog. 2021. *Intro to optimization in deep learning: Momentum, RMSProp and Adam*. [online] Available at: <https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/>

[6]: Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167

[7]: Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton. Layer Normalization. arXiv:1607.06450

[8]: Rosebrock, A., 2021. Keras ImageDataGenerator and Data Augmentation - PyImageSearch. [online] PyImageSearch. Available at: <https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/>

[9]: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. arXiv:1512.03385

[10]: Mingxing Tan, Quoc V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv: 1905.11946

[11]: <https://www.kaggle.com/maksymshkliarevskyi/cassava-leaf-disease-best-keras-cnn>

[12]: CloudxLab. 2021. *Understanding the Directory Structure*. [online] Available at: <https://cloudxlab.com/assessment/displayslide/5853/understanding-the-directory-structure?playlist_id=582>