The source code is at the end of this document

1 Classification: Convolutional Neural Networks

1.1 Design and Implementation Choices of your Model

Inspired by VGG and ResNet, almost all convolution layers are 3×3 and almost all pooling layers are 2×2.

The following CNN models are tested.

Model 1

CNN1 is a sample CNN with only one convolutional layer. Dropout layer is used, which is the easiest way to prevent overfitting.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 13, 13, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 5408)              0
_________________________________________________________________
dense (Dense)                (None, 128)               692352
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 645
=================================================================
Total params: 693,317
Trainable params: 693,317
Non-trainable params: 0
```

Model 2

CNN2 is similar to the network in the link given in the assignment. It has four convolutional layers, each followed by a batch normalization layer, which outputs the previous layer by substracting the batch mean and dividing it by batch standard deviation. Batch normalizatoin could speed up training and increase the stability of the network.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320       
_________________________________________________________________
batch_normalization (BatchNo (None, 28, 28, 32)        128       
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 32)        9248      
_________________________________________________________________
batch_normalization_1 (Batch (None, 28, 28, 32)        128       
_________________________________________________________________
dropout_1 (Dropout)          (None, 28, 28, 32)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 28, 28, 64)        18496     
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 128)       73856     
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 128)       512       
_________________________________________________________________
dropout_3 (Dropout)          (None, 14, 14, 128)       0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0         
_________________________________________________________________
dense_2 (Dense)              (None, 512)               12845568  
_________________________________________________________________
batch_normalization_3 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_4 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 128)               65664     
_________________________________________________________________
batch_normalization_4 (Batch (None, 128)               512       
_________________________________________________________________
dropout_5 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 5)                 645       
=================================================================
Total params: 13,017,125
Trainable params: 13,015,461
Non-trainable params: 1,664
```

Model 3

CNN3 is just CNN2 with an extra block(a Conv2D, a Batch normalization and a Dropout layer), because I wanted to check if a deeper network could do a better job.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_5 (Conv2D)            (None, 28, 28, 32)        320       
_________________________________________________________________
batch_normalization_5 (Batch (None, 28, 28, 32)        128       
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 28, 28, 32)        9248      
_________________________________________________________________
batch_normalization_6 (Batch (None, 28, 28, 32)        128       
_________________________________________________________________
dropout_6 (Dropout)          (None, 28, 28, 32)        0         
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 28, 28, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 64)        0         
_________________________________________________________________
dropout_7 (Dropout)          (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 14, 14, 128)       73856     
_________________________________________________________________
batch_normalization_7 (Batch (None, 14, 14, 128)       512       
_________________________________________________________________
dropout_8 (Dropout)          (None, 14, 14, 128)       0         
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 12, 12, 256)       295168    
_________________________________________________________________
batch_normalization_8 (Batch (None, 12, 12, 256)       1024      
_________________________________________________________________
dropout_9 (Dropout)          (None, 12, 12, 256)       0         
_________________________________________________________________
flatten_2 (Flatten)          (None, 36864)             0         
_________________________________________________________________
dense_5 (Dense)              (None, 512)               18874880  
_________________________________________________________________
batch_normalization_9 (Batch (None, 512)               2048      
_________________________________________________________________
dropout_10 (Dropout)         (None, 512)               0         
_________________________________________________________________
dense_6 (Dense)              (None, 128)               65664     
_________________________________________________________________
batch_normalization_10 (Batc (None, 128)               512       
_________________________________________________________________
dropout_11 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_7 (Dense)              (None, 5)                 645       
=================================================================
Total params: 19,342,629
Trainable params: 19,340,453
Non-trainable params: 2,176
```

Model 4

CNN4 is a VGG like network (two or three Convolutional layers followed by a pooling layer) given in this link. Batch normalizatoin layer was also applied to speed up computation.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_10 (Conv2D)           (None, 28, 28, 32)        320       
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 28, 28, 64)        18496     
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 64)        0         
_________________________________________________________________
dropout_12 (Dropout)         (None, 14, 14, 64)        0         
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 14, 14, 128)       73856     
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 12, 12, 256)       295168    
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 4, 4, 256)         0         
_________________________________________________________________
dropout_13 (Dropout)         (None, 4, 4, 256)         0         
_________________________________________________________________
flatten_3 (Flatten)          (None, 4096)              0         
_________________________________________________________________
dense_8 (Dense)              (None, 256)               1048832   
_________________________________________________________________
dropout_14 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_9 (Dense)              (None, 256)               65792     
_________________________________________________________________
dropout_15 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_10 (Dense)             (None, 5)                 1285      
=================================================================
Total params: 1,503,749
Trainable params: 1,503,749
Non-trainable params: 0
```

VGG19

VGG19 with imagenet pretrained weight. VGG19 contains 16 convolutional layers and 3 fully connected layers. VGG has a relatively simple network structure and it performs well in the image classification. Before using this model, I had to transform the input data into desired form. The origin input format of VGG19 is 224×224×3, and the minimum input dimension for keras VGG19 model is 32×32×3. I stacked the data three times and reshaped it to 150×150, because a too small input would affect the performance of the network and 150×150 is almost the largest size I could get within my computer's capability.

Its performance was really bad(accuracy 65.53%) so I gave it up. Maybe because our labels were twisted and it was not a conventional image classification problem so we should not use the weights directly.

Inception V3

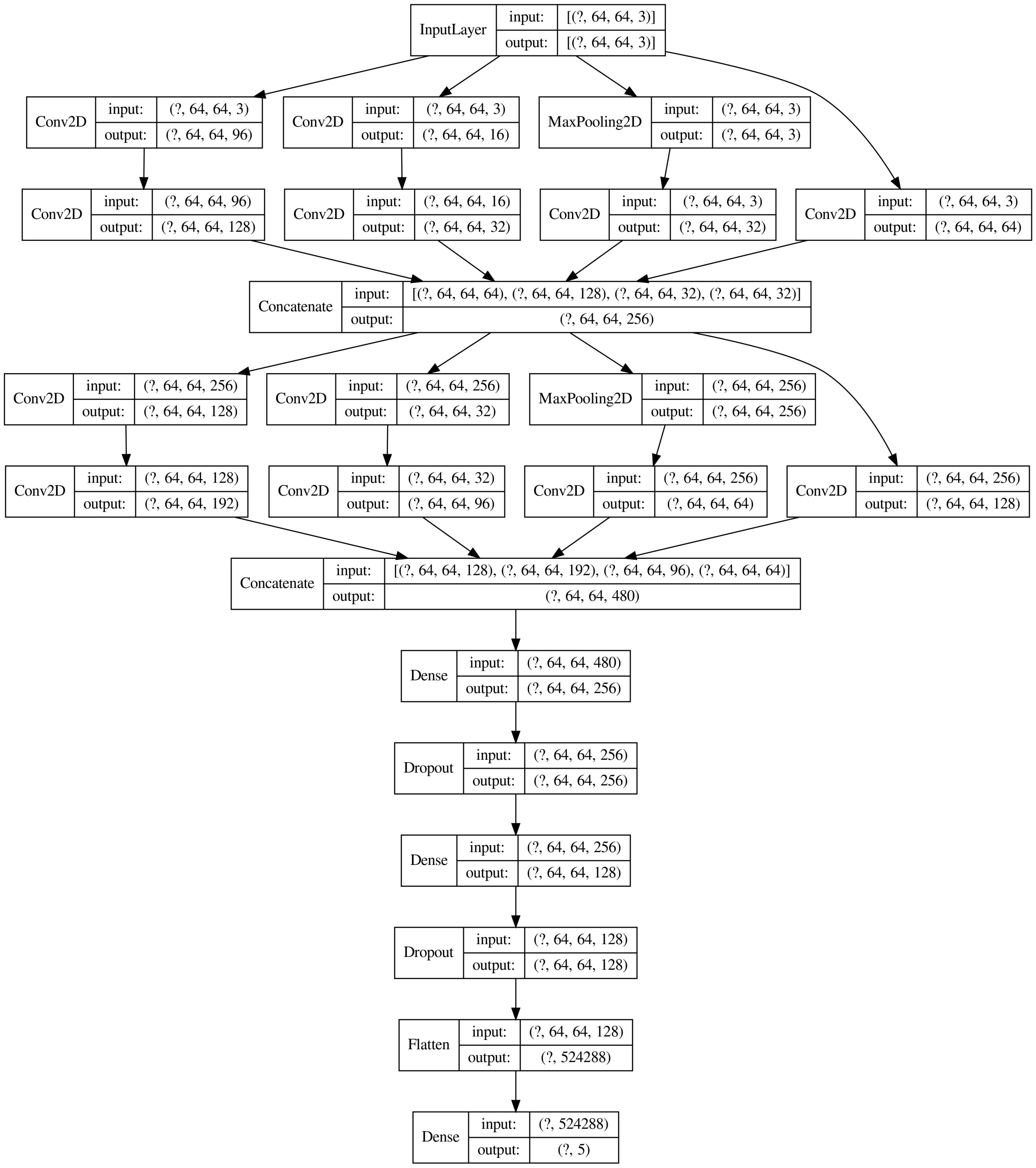
Inception network stacks the result of multiple filters together, therefore, the network can learn small details, middle sized features and almost whole images at the same time. So it will usually achieve a good result.

Since using the pretrained weight directly was not a good idea, I tried transfer learning on Inception V3. We had to adjust the input format to 150×150×3 like VGG19, and last 10 layers of the model were retrained.

The performance was bad as well(accuracy 40.03%), so I did not try any further improvements.

Inception

The Inception V3 model is too big for us to train from scratch. Following this blog, I built a smaller Inception network. But its accuracy was only 81.12%.

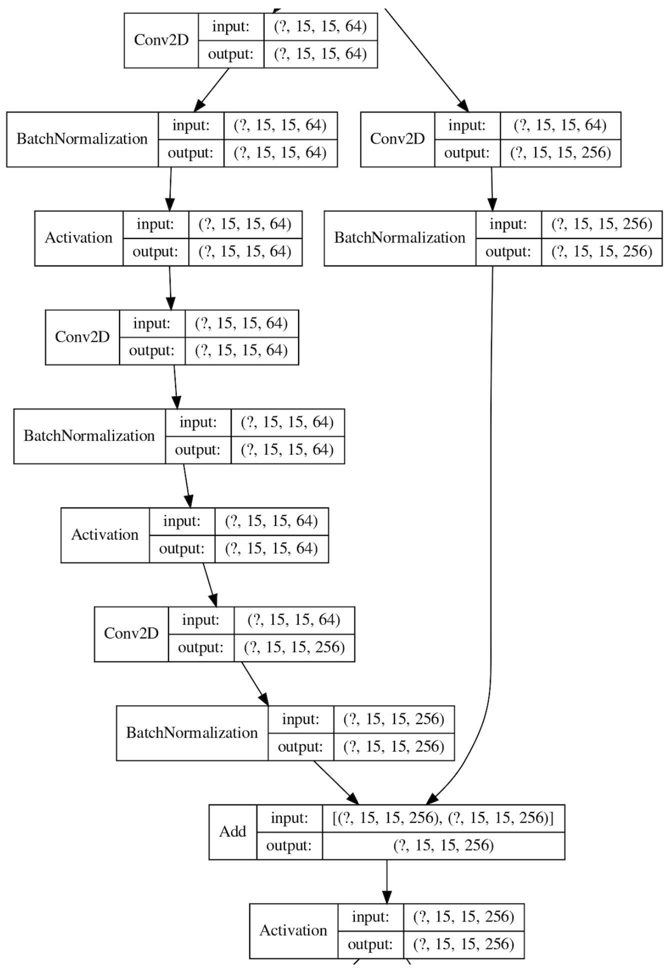


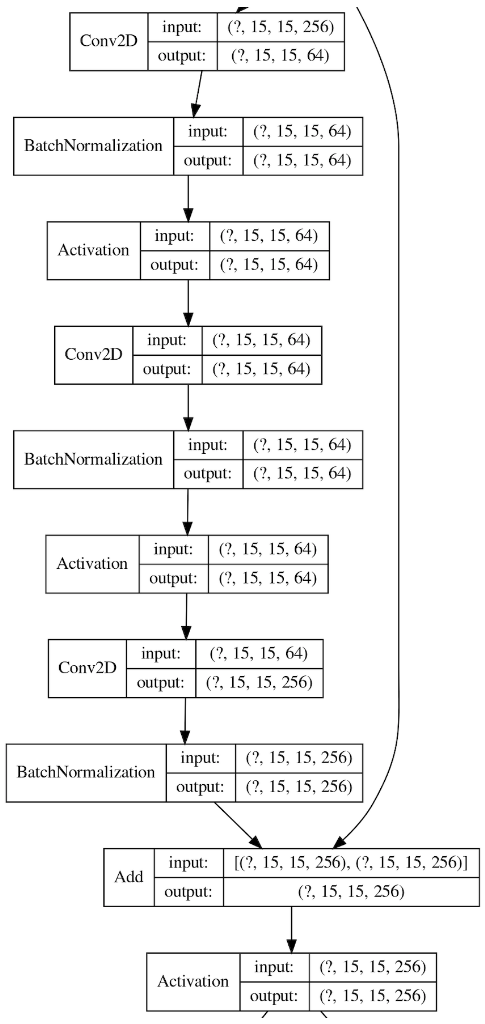
Resnet50

When the network becomes deeper and deeper, the model is prone to overfitting the data. Also, when performing the back-propagation, the gradient may becomes infinitively small or really big, making it harder to make progress. By skipping one or more layers, that is connecting one layer directly to a layer several depth apart, the model becomes more robust. This shortcut connection makes it possible for us to build deeper networks.

The Resnet50 used here was implemented based on Coursera CNN week 2 programming assignments.

The whole structure is too large to show. Only part of it is shown below. The left one is convolutional block and the right one is identity block. Resnet50 is basically the stacking of these two blocks.


Conv2D input: (?, 15, 15, 64) output: (?, 15, 15, 64)
BatchNormalization input: (?, 15, 15, 64) output: (?, 15, 15, 64)
Conv2D input: (?, 15, 15, 64) output: (?, 15, 15, 256)
Activation input: (?, 15, 15, 64) output: (?, 15, 15, 64)
BatchNormalization input: (?, 15, 15, 256) output: (?, 15, 15, 256)
Conv2D input: (?, 15, 15, 64) output: (?, 15, 15, 64)
BatchNormalization input: (?, 15, 15, 64) output: (?, 15, 15, 64)
Activation input: (?, 15, 15, 64) output: (?, 15, 15, 64)
Conv2D input: (?, 15, 15, 64) output: (?, 15, 15, 256)
BatchNormalization input: (?, 15, 15, 256) output: (?, 15, 15, 256)
Add input: [(?, 15, 15, 256), (?, 15, 15, 256)] output: (?, 15, 15, 256)
Activation input: (?, 15, 15, 256) output: (?, 15, 15, 256)


Conv2D input: (?, 15, 15, 256) output: (?, 15, 15, 64)
BatchNormalization input: (?, 15, 15, 64) output: (?, 15, 15, 64)
Activation input: (?, 15, 15, 64) output: (?, 15, 15, 64)
Conv2D input: (?, 15, 15, 64) output: (?, 15, 15, 64)
BatchNormalization input: (?, 15, 15, 64) output: (?, 15, 15, 64)
Activation input: (?, 15, 15, 64) output: (?, 15, 15, 64)
Conv2D input: (?, 15, 15, 64) output: (?, 15, 15, 256)
BatchNormalization input: (?, 15, 15, 256) output: (?, 15, 15, 256)
Add input: [(?, 15, 15, 256), (?, 15, 15, 256)] output: (?, 15, 15, 256)
Activation input: (?, 15, 15, 256) output: (?, 15, 15, 256)

Data Augmentation

ImageDataGenerator was used to generate transformed images for neural networks. Each image was zoomed, rotated, flipped or sheared and then fed to the network. But it did not perform well as expected. The validation accuracy decreased gradually from around 90 percent to 82 percent as more and more data being generated. Hence, data augmentation was not used later.

Hyperparameters, regularization and optimizers will be discussed later.

1.2 Implementation of your Design Choices

- A sequential model(take model 1 as example)

```
CNN1 = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='relu',
    ↪  input_shape=(28,28,1)),
    MaxPool2D(pool_size=(2, 2)),
    Dropout(0.5),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(5, activation='softmax')
])
CNN1.compile(optimizer='adam', loss="categorical_crossentropy",
↪  metrics=["accuracy"])
historyCNN1 = CNN1.fit(X_train, y_train, epochs = 30, batch_size=128,
↪  validation_data = (X_val, y_val))
```

- Data augmentation

```
datagen = ImageDataGenerator(rotation_range = 8, zoom_range = 0.1,
↪  shear_range = 0.3, width_shift_range=0.08, height_shift_range=0.08,
↪  vertical_flip=True)
history = model.fit(datagen.flow(X_train, y_train, batch_size = 128), epochs
↪  = 40, validation_data = (X_test, y_test),
↪  steps_per_epoch=X_train.shape[0] // 128, callbacks = [reduce_lr])
```

- Class to record training and testing time in one epoch

```
class TimeHistory(Callback):
    def on_train_begin(self, logs={}):
        self.times = []

    def on_epoch_begin(self, batch, logs={}):
        self.epoch_time_start = time()

    def on_epoch_end(self, batch, logs={}):
        self.times.append(time() - self.epoch_time_start)
```

- Use pretrained weight

```
vgg19 = VGG19(weights='imagenet', include_top=False, input_shape = (150,
  ↪ 150, 3), classes = 5)
X_train = keras.applications.vgg19.preprocess_input(X_train)
train_features = vgg19.predict(np.array(X_train), batch_size=256)
train_features = np.reshape(train_features, (48000, 4*4*512))
# then put the trained features to a single layer network to produce
  ↪ output
```

- Accuracy and loss plot

```
def acc_plot(history, name):
    plt.figure(figsize=(10,6))
    plt.plot(history['accuracy'], label='training accuracy')
    plt.plot(history['loss'], label='training loss')
    plt.plot(history['val_accuracy'], label='validation accuracy')
    plt.plot(history['val_loss'], label='validation loss')
    plt.title(name + ' Accuracy and Loss')
    plt.xlabel('epochs')
    plt.legend()
    return plt.show()
```

- The code to create non-sequential models like Inception and Resnet50 and the code for transfer learning are too long to put it here, please check the source code.

1.3 Kaggle Competition Score

0.9240

Achieved using ensemble method. Train the model below(it is not mentioned in section 1.1 because I tried too many models and I got confused, thought I was using Model 4 but actually they are different, I realized it very close to the deadline.) repeatedly and only keep the predicted result if its testing accuracy is greater than 91 percent. Collect about 30 predicted results and vote like random forest. It took around 36 hours to get 30 qualified results using a single GTX 1080 Ti. All parameters in this model were generated by Hyperas.

```
model = Sequential([
    Conv2D(32, kernel_size=3, activation='relu',padding='same', input_shape=(28,28,1)),
    Conv2D(32, kernel_size=3, activation='relu',padding='same'),
    MaxPool2D(pool_size=2,strides=2),
    Dropout(0.13243678),
    Conv2D(64, kernel_size=3, activation='relu'),
    Conv2D(64, kernel_size=3, activation='relu'),
    BatchNormalization(),
    MaxPool2D(pool_size=2,strides=2),
    Dropout(0.25767502),
    Flatten(),
    Dense(256, activation='relu'),
    BatchNormalization(),
    Dropout(0.31440486),
    Dense(512, activation='relu'),
    BatchNormalization(),
    Dropout(0.12715375),
    Dense(5, activation='softmax')
])
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001)
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer=Adam(lr=0.001))
model.fit(X_train, y_train, batch_size=128, epochs=40, verbose=0, validation_data=(X_val, y_val), callbacks = [reduce_l
```

1.4 Results Analysis

1.4.1 Runtime performance for training and testing

The training and predicting time **for one epoch** for each method is shown below:

Model	Training(s)	Testing(s)
1(3 layers)	1.84	1.21
2(7 layers)	5.92	1.84
3(8 layers)	8.01	1.88
4(7 layers)	4.68	1.43
Inception	161.96	14.30
Resnet50	36.01	7.66

For sequential models, the more parameters to train, the slower it will get. But the number of parameters and training time does not grow proportionally. Model 2 has 19,340,453 trainable params and model 4 has 1,503,749, yet their training time for one epoch are similar, which indicates that the structure of the network also affects run time. Inception has only 3,290,597 trainable params, however, it takes way longer to train. Resnet50 has 23,544,837 trainable params but it is 3.5 times faster than Inception.

1.4.2 Comparison of the different algorithms and parameters you tried

Model	**Accuracy**
1(3 layers)	79.10%-90.06%
2(7 layers)	85.68%-90.15%
3(8 layers)	87.42%-90.47%
4(7 layers)	89.68%-91.36%
Inception	81.12%
Resnet50	88.77%-89.56%

As we can see from the table, generally, the deeper a network, the more stable its performance. Model 1 only has 3 layers and its accuracy on validation set fluctuate a lot. 79.10% and 90.06% are both very rare cases, they occur only once or twice in hundreds of runs. The accuracy is about 88% on average. Model 2 and 4 are both 7 layers but model 4 is more stable than model 2. This may due to its VGG like structure.

The performance of Inception and Resnet50 are disappointing. It took 10 times longer to train them but the accuracy is worse than a simple VGG like model. A possible explanation for this is that the training set is not large enough. For the original Inception and Resnet, they are trained by millions of 224×224 colored images. However, we only have 60,000 28×28 grey scale images. The lack of input information made it difficult to build an accurate model.

1.4.3 Explanation of your model

The structure of each model have been shown in section 1.1. This section will mainly focus on hyperparameters, regularization and optimizers.

ReduceLROnPlateau was used as callback function on every model to reduce learning rate when the accuracy has stopped improving.

Only dropout layer as regularisation was used because it is the easiest one.

The epochs was set as 40 initially and later I would check if there was sign of overfitting. If there was, the number would be decreased.

The batch size was set as 128 or 256. These two numbers led to similar performance.

Activation function: softmax for output layer and ReLU for others.

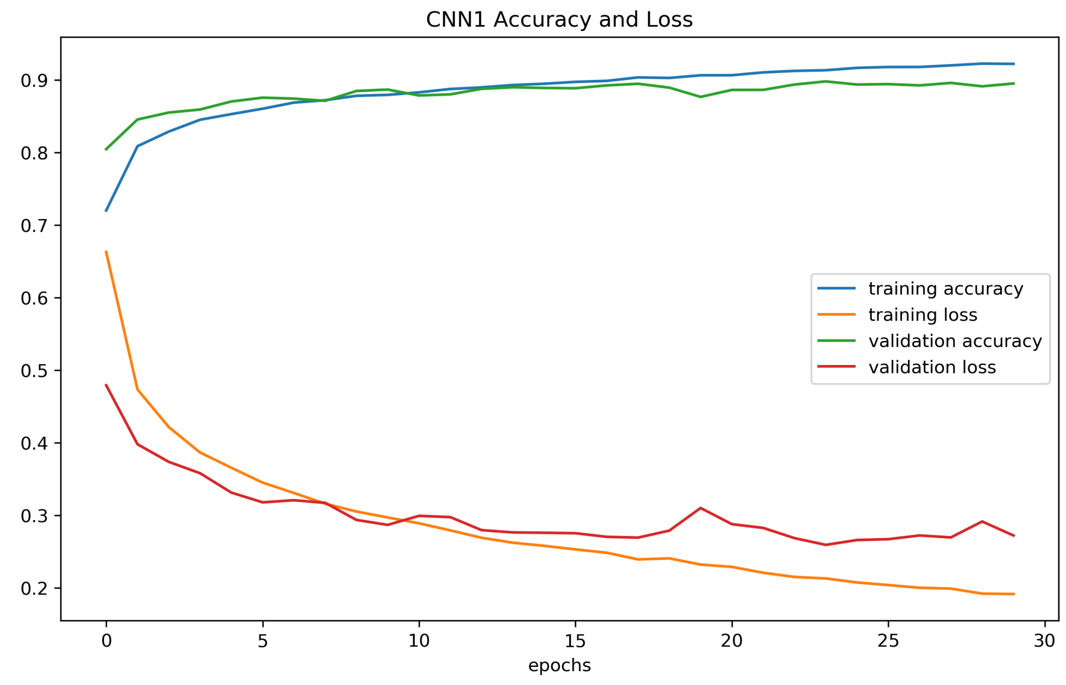
The following hyperparameters are tuned using hyperas and hyperopt following this guide:

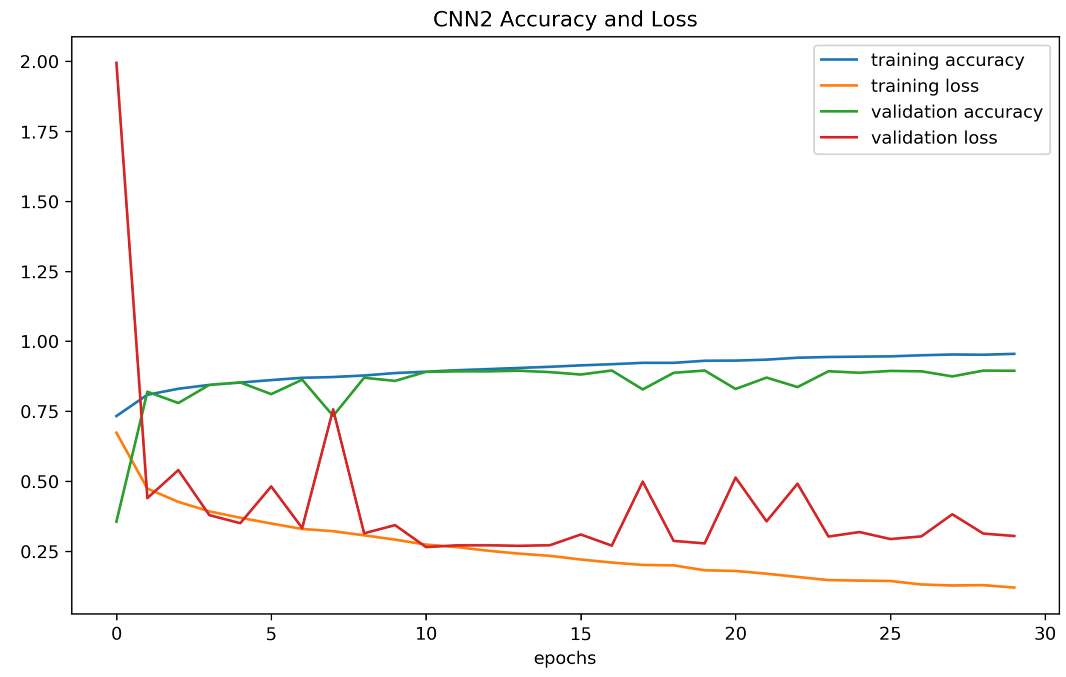
- Dropout rate: a random number between 0 and 1.
- Number of units in a Dense layer: 128, 256 or 512.
- Optimizer: Adam, SGD or RMSProp, each with learning rate of 0.001, 0.01 or 0.1.

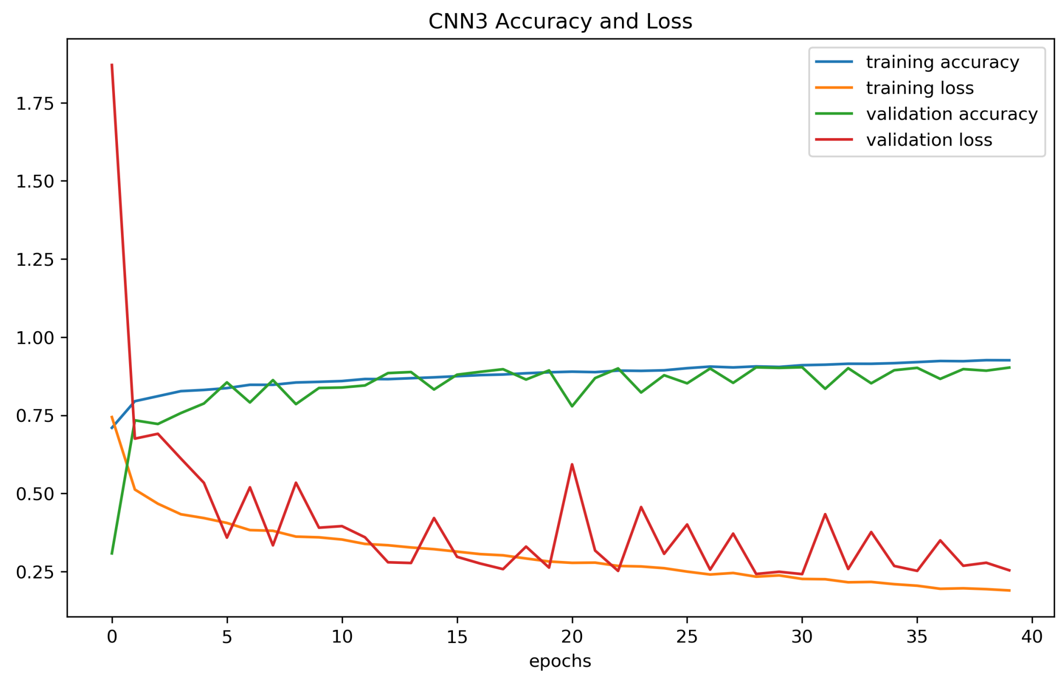
However, I found hyperas not so efficient and sometimes it generated a set of parameters that led to a really mediocre result after 30-50 evaluation.

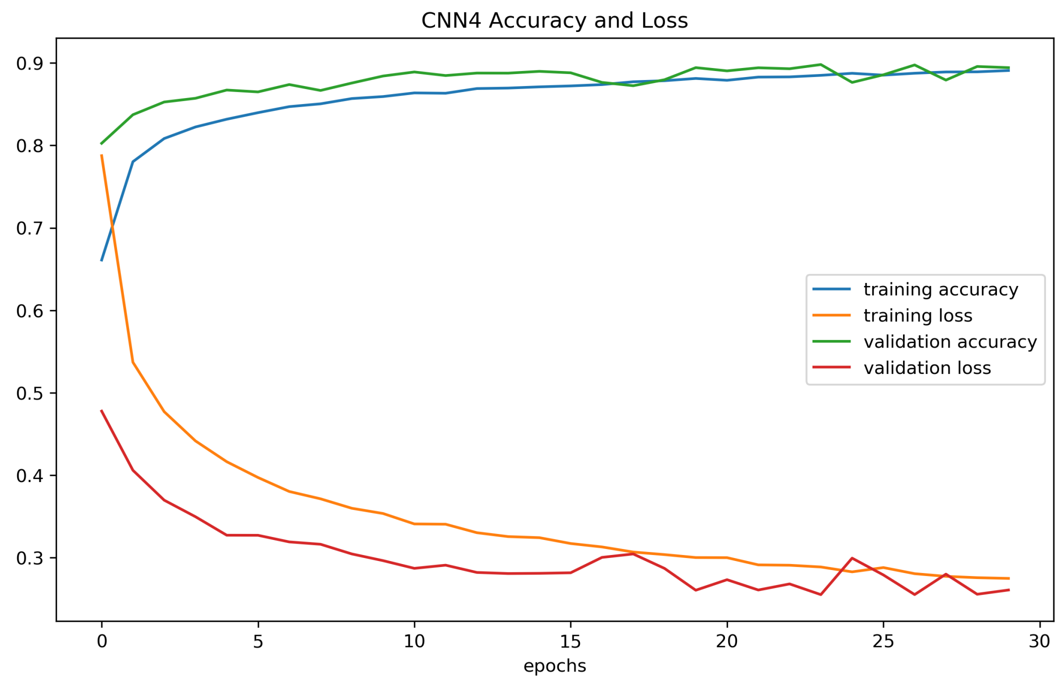
1.4.4 Plots

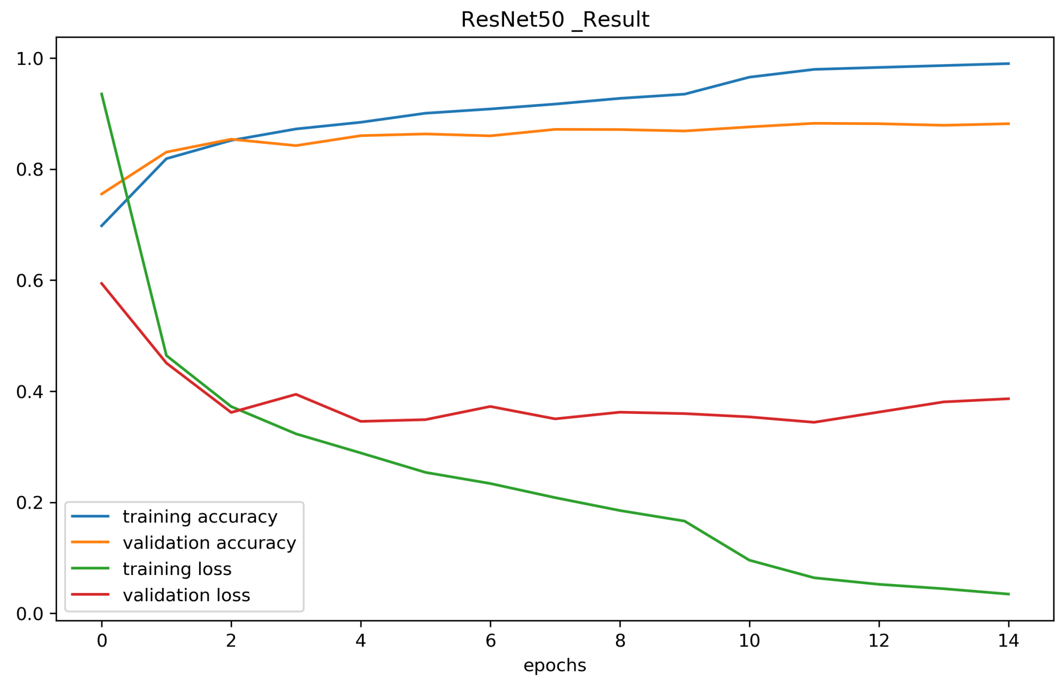
Loss, Accuracy vs Epochs for each model(except for Inception because it has only 82% accuracy) are plotted.


CNN1 Accuracy and Loss
training accuracy
training loss
validation accuracy
validation loss
epochs
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
5
10
15
20
25
30


CNN2 Accuracy and Loss
training accuracy
training loss
validation accuracy
validation loss
epochs
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
5
10
15
20
25
30


CNN3 Accuracy and Loss
training accuracy
training loss
validation accuracy
validation loss
epochs
0.25
0.50
0.75
1.00
1.25
1.50
1.75
0
5
10
15
20
25
30
35
40


CNN4 Accuracy and Loss
training accuracy
training loss
validation accuracy
validation loss
epochs
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
5
10
15
20
25
30



Overall, the accuracy increases and the loss decreases for both training set and validation set as more epochs are presented to the model. The training accuracy and loss seem always monotonous. However, there are fluctuations with the validation accuracy and loss. E.g. in model 2 and 3 the validation accuracy dives to 0.8 and then bounces back to 0.9 from time to time. This may due to overfitting, a too large learning rate, not enough data points or some inadequate hyperparameters. And for Resnet50, the validation accuracy does not increase after the 4th epochs, which is definitely because of overfitting, and also the model is too big and it requires more training data. And from the above plots we learn that about 20 epochs is enough to train these models.

1.4.5 Evaluate your code with other metrics on the training data and argue for the benefit of you approach.

Using 80 percent data for training and 20 percent for testing, the precision score, recall score and f1 score for each class were calculated.

Model	**Precision(Class 0-4) %**	**Recall(Class 0-4) %**	**F1(Class 0-4) %**
1(3 layers)	95.68 88.76 83.94 83.04 93.37	94.91 87.00 83.54 87.11 91.93	95.29 87.88 83.74 85.03 92.64
2(7 layers)	97.06 86.97 85.55 80.19 82.14	91.64 87.00 77.37 76.09 98.09	94.27 86.99 81.25 78.08 89.41
3(8 layers)	95.12 87.95 87.20 86.40 91.87	96.60 90.02 81.81 85.38 94.85	95.85 88.97 84.42 85.89 93.34
4(7 layers)	92.55 88.02 86.47 85.00 93.57	97.06 87.29 83.50 86.44 91.48	94.75 87.66 84.96 85.71 92.52
Resnet50	93.65 85.24 81.51 84.84 87.51	93.46 85.31 81.23 79.82 93.38	93.56 85.28 81.37 82.25 90.35

Precision=$\frac{true\ positive}{true\ positive+false\ positive}$, which indicates the proportion of the data points the model says was relevant actually were relevant.

Recall=$\frac{true\ positive}{true\ positive+false\ negative}$, which indicates the model's ability to find all relavent instances in the dataset.

F1=$2 \cdot \frac{precision+recall}{precision\cdot recall}$, which is the harmonic mean of precision and recall, and it takes both metrics into account.

With these different metrics, we can learn that class 2 is the most difficult one to classify and class 0 is the easiest one. And since we have all these scores for each class, instead of a single accuracy number, we can adjust our network to fit a certain task, e.g. if correctly predicting a specific class is much more important than predicting others.