Practicing Convolutional Neural Network
CIFAR 100 challenge

Nuozhou Wang

Introduction:

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. Our goal is to build an image classifier with above 75% accuracy. We will split the training images into a training set and a validation set. By building different networks on the train set, we will test each classifier on the validation set and select the best image classifier. In the end, we will use the selected model to predict the testing set and analysis the accuracy.

Methods:

- Library: Keras is an open-source neural-network library and is used in this project. The core data structure of Keras is a model, which is a way to organize layers. Sequential model is a linear stack of layers, the simplest type of model.
- The validation dataset is cfar10_data_batch1, training datasets are cfar10_data_batch 2 to 5.
- Net architecture: The LeNet architecture is straightforward and easy to understand. This project is based on the concept of LeNet architecture. A typical LeNet architecture consists of the following layers:
  INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => RELU => FC
- Regularization: Regularizers apply penalties on layer parameters or activity during optimization. Our model uses regularization like L2 reg and dropout.
- Parameters adjustment: PlotLossesCallback() function was applied to plot real-time log-loss and accuracy. batch_size, epoches, CONV layers, etc. were optimized based on the running result.

Results:

Model1: 2 CONV layers

Model building:

Conv2D class was used to implement a simple Convolutional Neural Network. The CONV layers had a total of 32 and 64 filters. Max pooling was used to reduce the spatial dimensions of the output volume.

```
classes_num = 10
input_shape = (32, 32, 3)
weight_decay = 1e-4

model = Sequential()

# first CONV => RELU => CONV => RELU => POOL layer => dropout
model.add(Conv2D(32, (3, 3), padding = 'same',  kernel_regularizer=l2(weight_decay), input_shape = input_shape))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3), padding = 'same', kernel_regularizer=l2(weight_decay)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.25))

# second CONV => RELU => CONV => RELU => POOL layer set => dropout
model.add(Conv2D(64, (3, 3), padding = 'same', kernel_regularizer=l2(weight_decay)))
model.add(Activation('relu'))
model.add(Conv2D(64, (5, 5), padding = 'same', kernel_regularizer=l2(weight_decay)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.25))

# set of FC => RELU layers
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# softmax classifier
model.add(Dense(classes_num))
model.add(Activation('softmax'))
```

```
opt = keras.optimizers.RMSprop(lr=0.0001, decay=1e-6)
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
print(model.summary())
```

```
batch_size = 64
epochs = 35
valBatch = 0

import matplotlib.pyplot as plt
from livelossplot.keras import PlotLossesCallback

for i in range(1, 5):
  history = model.fit(x_trainingList[i], y_trainingList[i],
          batch_size = batch_size,
          epochs= epochs,
          validation_data=(x_trainingList[valBatch], y_trainingList[valBatch]),
          callbacks=[PlotLossesCallback()],
          verbose=2,
          shuffle = True)
model.save('my_model1.h5', overwrite=True)
print("Saved model to disk for batch:" + str(i))
model = load_model('my_model1.h5')
```

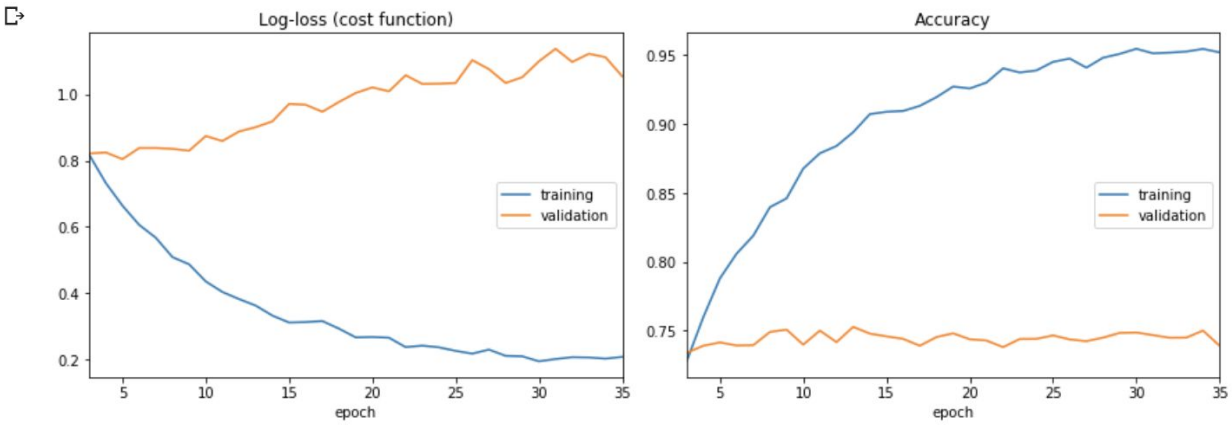
Result:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 32, 32)	896
activation_1 (Activation)	(None, 32, 32, 32)	0
conv2d_2 (Conv2D)	(None, 32, 32, 32)	9248
max_pooling2d_1 (MaxPooling2	(None, 16, 16, 32)	0
dropout_1 (Dropout)	(None, 16, 16, 32)	0
conv2d_3 (Conv2D)	(None, 16, 16, 64)	18496
conv2d_4 (Conv2D)	(None, 16, 16, 64)	102464
max_pooling2d_2 (MaxPooling2	(None, 8, 8, 64)	0
dropout_2 (Dropout)	(None, 8, 8, 64)	0
flatten_1 (Flatten)	(None, 4096)	0
dense_1 (Dense)	(None, 512)	2097664
activation_2 (Activation)	(None, 512)	0
dropout_3 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 10)	5130
activation_3 (Activation)	(None, 10)	0

Total params: 2,233,898
Trainable params: 2,233,898
Non-trainable params: 0

None

Fig1A. model 1 summary




```
Log-loss (cost function):
training   (min:    0.195, max:    1.175, cur:    0.209)
validation (min:    0.804, max:    1.137, cur:    1.054)

Accuracy:
training   (min:    0.645, max:    0.955, cur:    0.952)
validation (min:    0.707, max:    0.753, cur:    0.739)
Saved model to disk for batch:4
10000/10000 [==============================] - 2s 193us/step
```

Fig1B. Model 1 visualization

For the validation dataset, log-loss is 1.054, accuracy is 0.739.

Model2: 2 CONV layers + ImageDataGenerator

Model building:

Inspired by the article of Train a simple deep CNN on the CIFAR10 small images dataset (https://keras.io/examples/cifar10_cnn/), I tried to apply ImageDataGenerator for real-time data augmentation.

```
batch_size = 64
epochs = 35
valBatch = 0

import matplotlib.pyplot as plt
from keras.preprocessing.image import ImageDataGenerator
from math import ceil

for i in range(1, 5):
  # This will do preprocessing and realtime data augmentation:
  n_points = len(x_trainingList[i])
  steps_per_epoch = ceil(n_points / batch_size)
  datagen = ImageDataGenerator(
      featurewise_center=False,
      samplewise_center=False,
      featurewise_std_normalization=False,
      samplewise_std_normalization=False,
      zca_whitening=False,
      rotation_range=0,
      width_shift_range=0.1,
      height_shift_range=0.1,
      shear_range=0.,
      zoom_range=0.,
      channel_shift_range=0.,
      cval=0.,
      horizontal_flip=True,
      vertical_flip=False,
      rescale=None,
      preprocessing_function=None,
      data_format=None,
      validation_split=0.0)
  datagen.fit(x_trainingList[i])
  history = model.fit_generator(datagen.flow(x_trainingList[i], y_trainingList[i],
                      batch_size=batch_size),
                      steps_per_epoch = steps_per_epoch,
                      epochs=epochs,
                      validation_data=(x_trainingList[valBatch], y_trainingList[valBatch]),
                      workers=4)
```

```
39
40 model.save('my_model1.h5', overwrite=True)
41 print("Saved model to disk for batch:" + str(i))
42 model = load_model('my_model1.h5')
43 print(model.evaluate(x = x_test, y = y_test))
44
45 # Plot training & validation loss values
46 plt.plot(history.history['loss'])
47 plt.plot(history.history['val_loss'])
48 plt.title('Model loss')
49 plt.ylabel('Loss')
50 plt.xlabel('Epoch')
51 plt.legend(['Train', 'Test'], loc='upper left')
52 plt.show()
53
54 # Plot training & validation accuracy values
55 plt.plot(history.history['acc'])
56 plt.plot(history.history['val_acc'])
57 plt.title('Model accuracy')
58 plt.ylabel('Accuracy')
59 plt.xlabel('Epoch')
60 plt.legend(['Train', 'Test'], loc='upper left')
61 plt.show()
62
```

Result:

The model summary was same as model 1.

```
Epoch 26/35
157/157 [==============================] - 11s 68ms/step - loss: 0.6179 - acc: 0.8036 - val_loss: 0.7111 - val_acc: 0.7861
Epoch 27/35
157/157 [==============================] - 11s 67ms/step - loss: 0.6265 - acc: 0.8025 - val_loss: 0.7495 - val_acc: 0.7696
Epoch 28/35
157/157 [==============================] - 11s 67ms/step - loss: 0.6278 - acc: 0.8006 - val_loss: 0.7578 - val_acc: 0.7721
Epoch 29/35
157/157 [==============================] - 11s 71ms/step - loss: 0.6085 - acc: 0.8083 - val_loss: 0.6924 - val_acc: 0.7922
Epoch 30/35
157/157 [==============================] - 11s 69ms/step - loss: 0.5927 - acc: 0.8089 - val_loss: 0.7173 - val_acc: 0.7869
Epoch 31/35
157/157 [==============================] - 11s 70ms/step - loss: 0.6007 - acc: 0.8054 - val_loss: 0.6488 - val_acc: 0.8004
Epoch 32/35
157/157 [==============================] - 11s 70ms/step - loss: 0.6053 - acc: 0.8048 - val_loss: 0.6684 - val_acc: 0.7907
Epoch 33/35
157/157 [==============================] - 11s 68ms/step - loss: 0.6028 - acc: 0.8056 - val_loss: 0.7041 - val_acc: 0.7892
Epoch 34/35
157/157 [==============================] - 11s 70ms/step - loss: 0.5957 - acc: 0.8092 - val_loss: 0.6856 - val_acc: 0.7949
Epoch 35/35
157/157 [==============================] - 11s 68ms/step - loss: 0.5779 - acc: 0.8183 - val_loss: 0.7209 - val_acc: 0.7849
Saved model to disk for batch:4
10000/10000 [==============================] - 2s 199us/step
```

Fig 2A. Real-time log-loss and accuracy

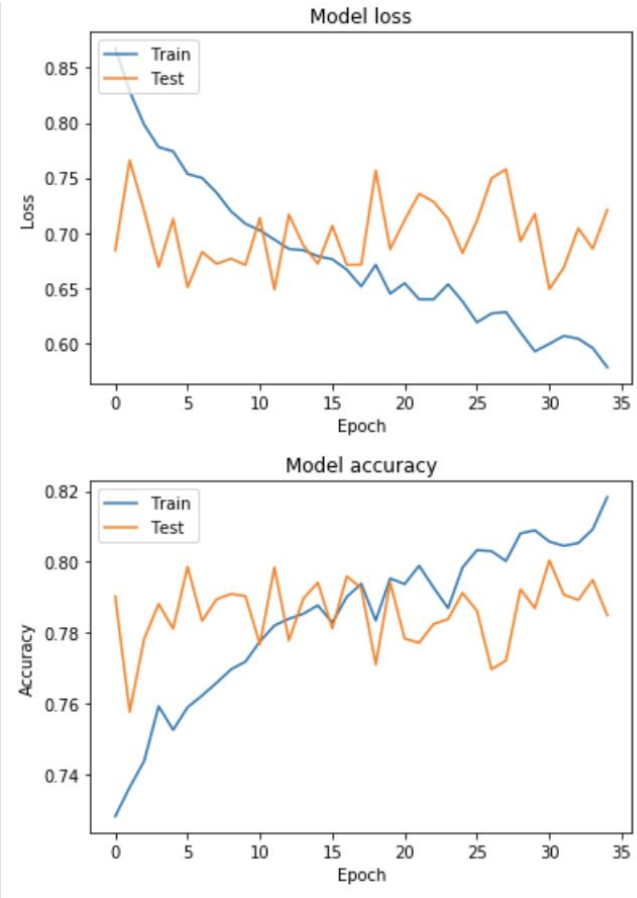


Fig 2B. Model 2 visualization

For the validation dataset, log-loss was 0.7209, accuracy is 0.7849. The result was better than model 1. However, ImageDataGenerator was more time consuming.

Model3: 3 CONV layers

Model building:

In addition to the 32 and 64 filters for the Conv2D layers, another 128 filters were added. This time, model.fit() was simply used to save running time.

```
classes_num = 10
input_shape = (32, 32, 3)
weight_decay = 1e-4

model = Sequential()

# first CONV => RELU => CONV => RELU => POOL layer => dropout
model.add(Conv2D(32, (3, 3), padding = 'same',  kernel_regularizer=l2(weight_decay), input_shape = input_shape))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3), padding = 'same', kernel_regularizer=l2(weight_decay)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.25))

# second CONV => RELU => CONV => RELU => POOL layer set => dropout
model.add(Conv2D(64, (3, 3), padding = 'same', kernel_regularizer=l2(weight_decay)))
model.add(Activation('relu'))
model.add(Conv2D(64, (5, 5), padding = 'same', kernel_regularizer=l2(weight_decay)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.25))

# third CONV => RELU => CONV => RELU => POOL layer set => dropout
model.add(Conv2D(128, (3, 3), padding = 'same', kernel_regularizer=l2(weight_decay),activation = 'relu'))
model.add(Conv2D(128, (5, 5), padding = 'same', kernel_regularizer=l2(weight_decay), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.25))


# set of FC => RELU layers
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.4))

# softmax classifier
model.add(Dense(classes_num))
model.add(Activation('softmax'))

```

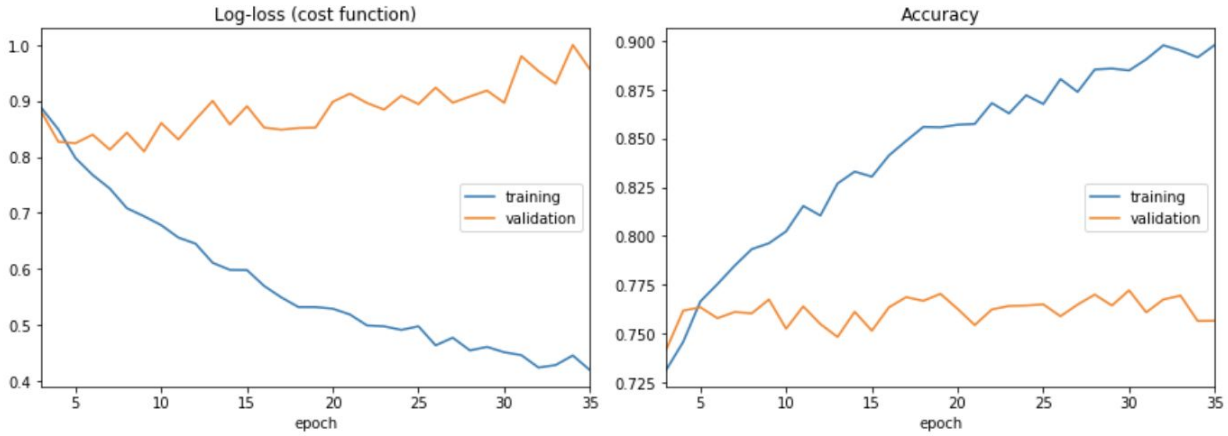
Result:

Layer (type)	Output Shape	Param #
conv2d_17 (Conv2D)	(None, 32, 32, 32)	896
activation_22 (Activation)	(None, 32, 32, 32)	0
conv2d_18 (Conv2D)	(None, 32, 32, 32)	9248
activation_23 (Activation)	(None, 32, 32, 32)	0
max_pooling2d_9 (MaxPooling2	(None, 16, 16, 32)	0
dropout_13 (Dropout)	(None, 16, 16, 32)	0
conv2d_19 (Conv2D)	(None, 16, 16, 64)	18496
activation_24 (Activation)	(None, 16, 16, 64)	0
conv2d_20 (Conv2D)	(None, 16, 16, 64)	102464
activation_25 (Activation)	(None, 16, 16, 64)	0
max_pooling2d_10 (MaxPooling	(None, 8, 8, 64)	0
dropout_14 (Dropout)	(None, 8, 8, 64)	0
conv2d_21 (Conv2D)	(None, 8, 8, 128)	73856
conv2d_22 (Conv2D)	(None, 8, 8, 128)	409728
max_pooling2d_11 (MaxPooling	(None, 4, 4, 128)	0
dropout_15 (Dropout)	(None, 4, 4, 128)	0
flatten_5 (Flatten)	(None, 2048)	0
dense_9 (Dense)	(None, 512)	1049088
activation_26 (Activation)	(None, 512)	0
dropout_16 (Dropout)	(None, 512)	0
dense_10 (Dense)	(None, 10)	5130
activation_27 (Activation)	(None, 10)	0

Total params: 1,668,906
Trainable params: 1,668,906
Non-trainable params: 0

None

Fig3A. model 3 summary




```
Log-loss (cost function):
training   (min:    0.420, max:    1.104, cur:    0.420)
validation (min:    0.810, max:    1.000, cur:    0.956)

Accuracy:
training   (min:    0.675, max:    0.898, cur:    0.898)
validation (min:    0.737, max:    0.772, cur:    0.757)
Saved model to disk for batch:4
10000/10000 [==============================] - 3s 280us/step
```

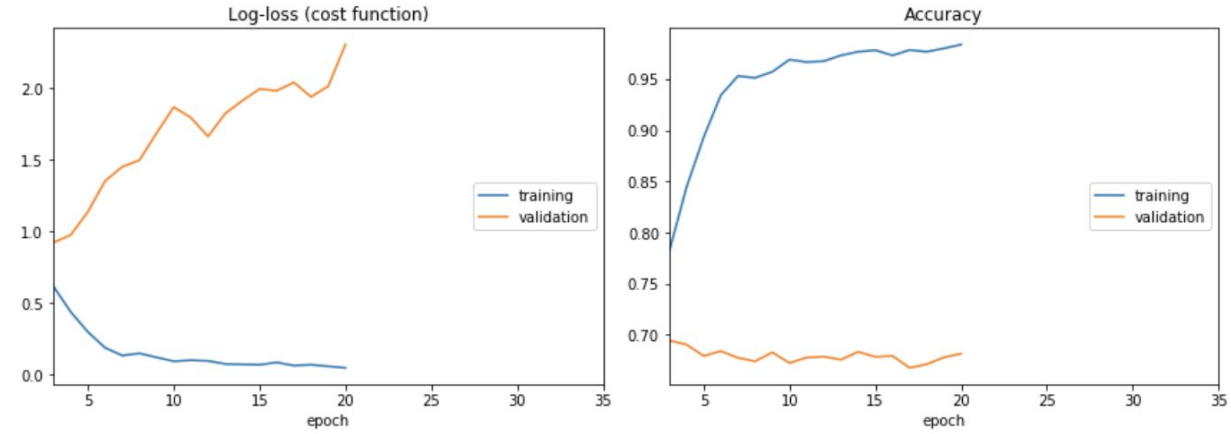
Fig 3B. Model 3 visualization

For the validation dataset, log-loss was 0.956, accuracy was 0.757. The accuracy was above 75% and it was more efficient when compared with model 2. Thus, model 3 was selected as the best model.

Regularization Optimization:

With model 3, regularization optimization was carried out.

1. kernel_regularizer=l2 : weight decay = 0

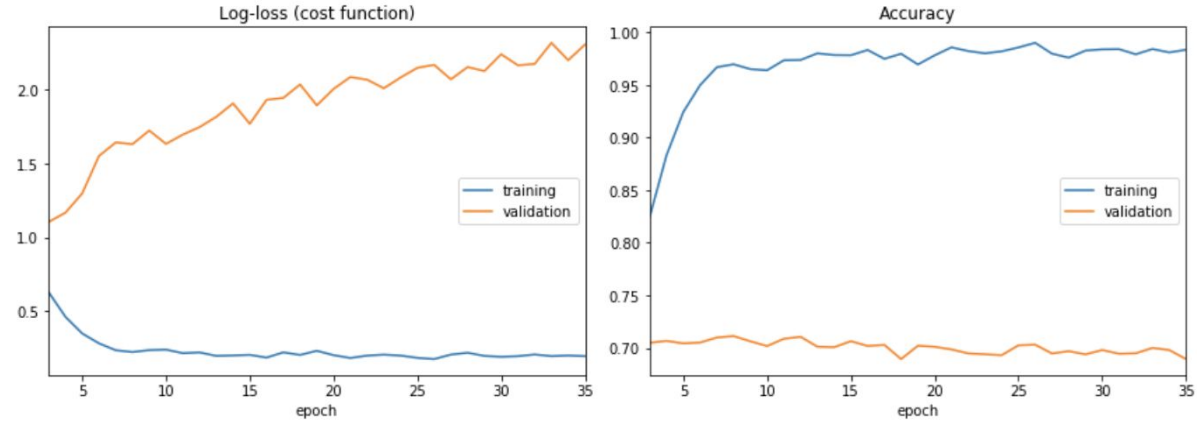



```
Log-loss (cost function):
training   (min:    0.049, max:    1.170, cur:    0.049)
validation (min:    0.925, max:    2.307, cur:    2.307)

Accuracy:
training   (min:    0.618, max:    0.984, cur:    0.984)
validation (min:    0.667, max:    0.694, cur:    0.682)
Epoch 21/35
```

Fig 4A weight decay of 0

The log-loss of training dataset was as low as 0.049, but the validation loss was 2.307 and the accuracy was decreased, indicating the presence of overfitting.

2. kernel_regularizer=l2 : weight decay = 0.00005

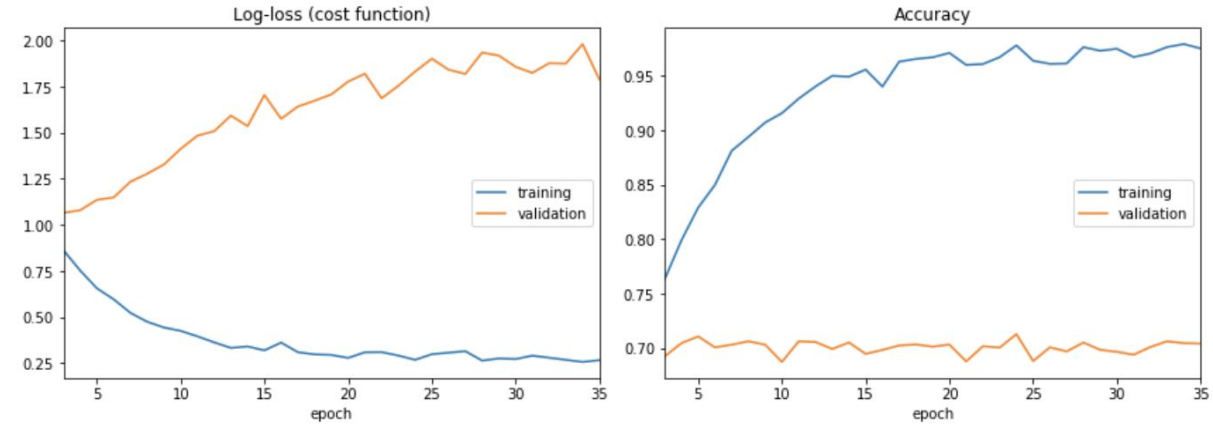



```
Log-loss (cost function):
training   (min:    0.175, max:    1.275, cur:    0.195)
validation (min:    1.027, max:    2.319, cur:    2.307)

Accuracy:
training   (min:    0.632, max:    0.990, cur:    0.984)
validation (min:    0.679, max:    0.711, cur:    0.689)
Saved model to disk for batch:4
10000/10000 [==============================] - 1s 143us/step
```

Fig 4B weight decay of 0.00005

The log-loss of training dataset was as low as 0.195, but the validation loss was 2.307, indicating the presence of overfitting.

3. kernel_regularizer= l2 : weight decay = 0.0005

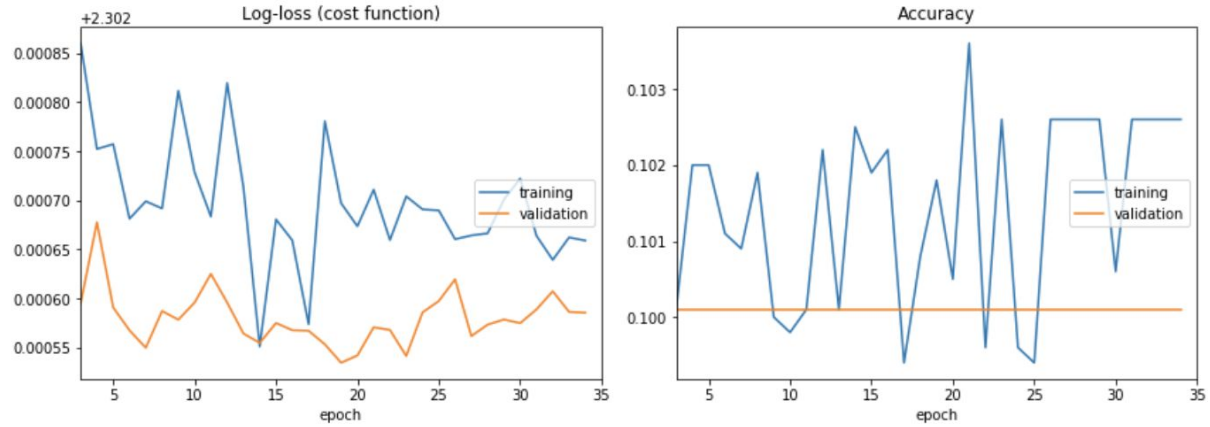



```
Log-loss (cost function):
training   (min:    0.256, max:    1.312, cur:    0.265)
validation (min:    1.053, max:    1.981, cur:    1.790)

Accuracy:
training   (min:    0.637, max:    0.979, cur:    0.975)
validation (min:    0.679, max:    0.713, cur:    0.704)
Saved model to disk for batch:4
10000/10000 [==============================] - 3s 340us/step
```

Fig 4C weight decay of 0.0005

The log-loss of training dataset was as low as 0.265, but the validation loss was 1.790, indicating the presence of overfitting.

4. kernel_regularizer= l2 : weight decay = 0.05

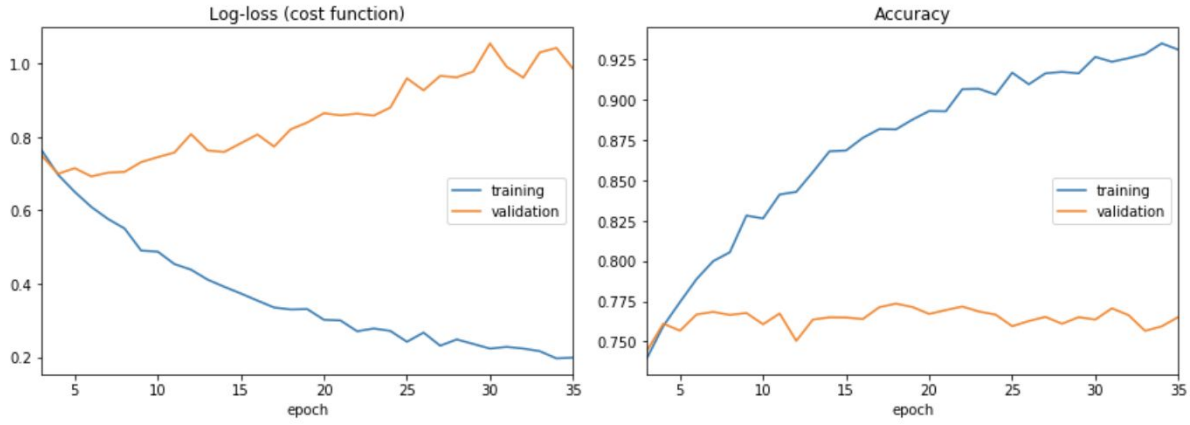



```
Log-loss (cost function):
training   (min:    2.303, max:    4.060, cur:    2.303)
validation (min:    2.303, max:    2.307, cur:    2.303)

Accuracy:
training   (min:    0.095, max:    0.104, cur:    0.103)
validation (min:    0.100, max:    0.100, cur:    0.100)
Epoch 35/35
```

Fig 4D weight decay of 0.05

The log-loss and accuracy were low in both training and validation datasets.

5. Without weight decay, add dropout layer



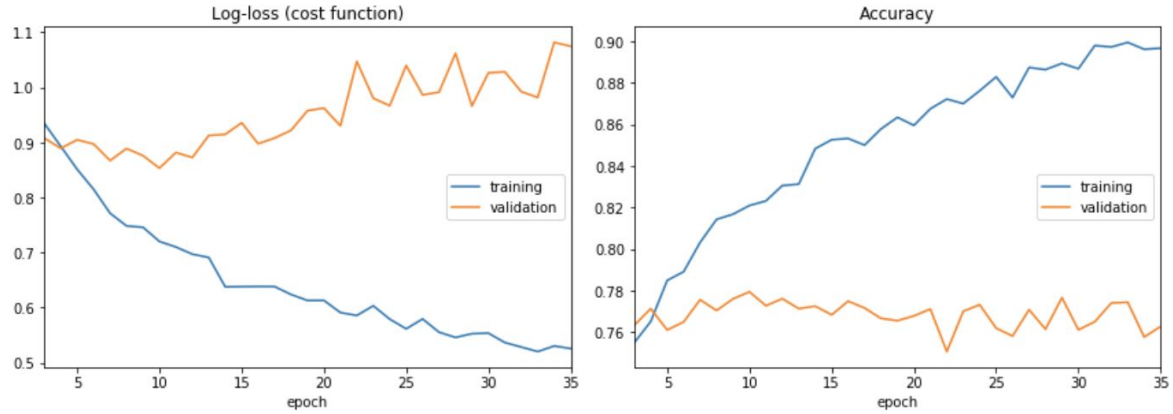

```
Log-loss (cost function):
training   (min:    0.196, max:    1.026, cur:    0.198)
validation (min:    0.693, max:    1.056, cur:    0.986)

Accuracy:
training   (min:    0.669, max:    0.935, cur:    0.931)
validation (min:    0.735, max:    0.773, cur:    0.765)
Saved model to disk for batch:4
10000/10000 [==============================] - 2s 152us/step
```

Fig 4E dropout

After adding dropout layer, the log-loss of training dataset was 0.198, and the validation loss was 0.765. The result was better than L2 regularization without dropout.

6. L2 regularization and dropout

Both of L2 regularization and dropout were adopted. The decay weight was 0.0005.




```
Log-loss (cost function):
training   (min:    0.520, max:    1.137, cur:    0.526)
validation (min:    0.853, max:    1.082, cur:    1.075)

Accuracy:
training   (min:    0.695, max:    0.899, cur:    0.897)
validation (min:    0.750, max:    0.779, cur:    0.763)
Saved model to disk for batch:4
10000/10000 [==============================] - 2s 182us/step
```

Fig 4F weight decay of 0.0005 and dropout

Here, we observed that the difference between training log-loss and validation log-loss decreased. The validation accuracy was 0.763, which was above 75%.

Testing dataset prediction:

Based on the previous results, model 3 was selected as the best model. L2 regularization (decay weight = 0.0005) and dropout were adopted.

The testing results are shown as follows:

```
Epoch 22/35
 - 3s - loss: 0.5664 - acc: 0.8837 - val_loss: 0.9053 - val_acc: 0.7899
Epoch 23/35
 - 3s - loss: 0.5273 - acc: 0.8954 - val_loss: 0.9042 - val_acc: 0.7954
Epoch 24/35
 - 3s - loss: 0.5389 - acc: 0.8936 - val_loss: 0.8974 - val_acc: 0.7939
Epoch 25/35
 - 3s - loss: 0.5067 - acc: 0.9002 - val_loss: 0.9446 - val_acc: 0.7929
Epoch 26/35
 - 3s - loss: 0.5284 - acc: 0.8942 - val_loss: 0.9316 - val_acc: 0.7938
Epoch 27/35
 - 3s - loss: 0.5165 - acc: 0.9037 - val_loss: 0.9446 - val_acc: 0.7878
Epoch 28/35
 - 3s - loss: 0.5099 - acc: 0.9024 - val_loss: 0.9721 - val_acc: 0.7825
Epoch 29/35
 - 3s - loss: 0.5116 - acc: 0.9009 - val_loss: 0.9287 - val_acc: 0.7912
Epoch 30/35
 - 3s - loss: 0.5058 - acc: 0.9053 - val_loss: 0.9221 - val_acc: 0.7950
Epoch 31/35
 - 3s - loss: 0.4938 - acc: 0.9109 - val_loss: 0.9544 - val_acc: 0.7848
Epoch 32/35
 - 3s - loss: 0.5061 - acc: 0.9052 - val_loss: 0.9288 - val_acc: 0.7928
Epoch 33/35
 - 3s - loss: 0.5100 - acc: 0.9026 - val_loss: 0.9433 - val_acc: 0.7911
Epoch 34/35
 - 3s - loss: 0.4911 - acc: 0.9099 - val_loss: 0.9481 - val_acc: 0.7989
Epoch 35/35
 - 3s - loss: 0.4904 - acc: 0.9086 - val_loss: 0.9522 - val_acc: 0.7875
Saved model to disk for batch:4
10000/10000 [==============================] - 2s 196us/step
[0.9805314288139343, 0.7838]
```

Fig 5A testing dataset prediction

The log-loss of the testing dataset was 0.980, and the accuracy was 0.7838. Confusion matrix and classification report were also generated.

```
Confusion Matrix
            airplane  automobile  bird  cat  ...  frog  horse  ship  truck
airplane         820          20    32   20  ...     8     12    42     39
automobile        10         906     2    4  ...     5      0    13     57
bird              65           5   652   43  ...    88     12     3      9
cat               27           7    54  624  ...    90     13     7     23
deer              22           3    65   45  ...    62     52     4      4
dog               14           4    50  145  ...    45     23     4     12
frog               7           0    21   22  ...   911      1     5      4
horse             20           0    28   48  ...    10    781     0     25
ship              53          24     5    8  ...    11      3   854     35
truck             19          48     3    6  ...     3      1    12    905

[10 rows x 10 columns]
Classification Report
              precision    recall  f1-score   support

    airplane       0.78      0.82      0.80      1000
  automobile       0.89      0.91      0.90      1000
        bird       0.71      0.65      0.68      1000
         cat       0.65      0.62      0.64      1000
        deer       0.76      0.72      0.74      1000
         dog       0.73      0.67      0.69      1000
        frog       0.74      0.91      0.82      1000
       horse       0.87      0.78      0.82      1000
        ship       0.90      0.85      0.88      1000
       truck       0.81      0.91      0.86      1000

    accuracy                           0.78     10000
   macro avg       0.78      0.78      0.78     10000
weighted avg       0.78      0.78      0.78     10000
```

Fig 5B confusion matrix and classification report

References:

1. https://www.pyimagesearch.com/2018/12/31/keras-conv2d-and-convolutional-layers/
2. https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5
3. https://www.cs.toronto.edu/~kriz/cifar.html
4. https://keras.io/examples/cifar10_cnn/