Homework 3

Amrit Parimi ap4142

AdaGrad weight update equation:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

where θ is the vector of parameters. η is the learning rate. ϵ is a smoothing term that prevents division by 0. g_t is the gradient at time step t. G_t is a diagonal matrix where each diagonal element $G_{t,ii}$ is the sum of squares of the gradients with respect to θ_i up to time step t.
η-learning rate is the only hyperparameter

RMSProp weight update:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Here, $E[g^2]_t$ is the running average of all past gradients squared at time t. g_t is the gradient at time t. ϵ is the smoothing term to prevent division by 0.
γ-momentum decay and η-learning rate are the hyperparameters.

RMSProp + Nesterov weight update:

$$v_t = \rho v_{t-1} + (1-\rho)g_t^2$$
$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)v_t$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Here, ρ- decaying constant, η-learning rate γ-momentum decay term are the hyperparameters.

AdaDelta weight update:

$$E[\Delta\theta^2]_t = \rho E[\Delta\theta^2]_{t-1} + (1-\rho)\Delta\theta^2$$
$$\theta_{t+1} = \theta_t - \frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

$E[\Delta\theta^2]$ is the running average of the decayed squared parameter update.
ρ-decaying constant is the hyperparameter.

Adam weight update:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon}\hat{m}_t$$
$$where \;\; \hat{m}_t = \frac{m_t}{1-\rho_1^t}$$
$$\hat{v_t} = \frac{v_t}{1-\rho_2^t}$$
$$and \;\; m_t = \rho_1 m_{t-1} + (1-\rho_1)g_t$$
$$v_t = \rho_2 v_{t-1} + (1-\rho_2)g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. $\hat{m_t}$ and $\hat{v_t}$ are unbiased estimates of the first moment (the mean) and the second moment of the gradients respectively.
Here, η, ρ_1, ρ_2 are the hyperparameters.

Adam and Adadelta vs RMSProp: RMSProp is similar to Adadelta excpet that it uses RMS in theh numerator instead of the learniing rate. This helps us reduce a hyperparameter. And Adam corrects the bias making it unbiased and also adds momentum to the update.

```
import numpy as np
import matplotlib.pyplot as plt
import time
```

```
from keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print(x_train.shape,y_train.shape)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 7s 0us/step
170508288/170498071 [==============================] - 7s 0us/step
(50000, 32, 32, 3) (50000, 1)
```

```
import tensorflow as tf
from keras.utils import np_utils
import keras
# one-hot encode the labels
num_classes = len(np.unique(y_train))
print(num_classes)
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test = tf.keras.utils.to_categorical(y_test, num_classes)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]
print(y_test.shape,y_valid.shape)
y_train.shape
```

```
10
(10000, 10) (5000, 10)
(45000, 10)
```

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten

# define the model
def create_model1():
    model = Sequential()
    model.add(Flatten(input_shape = x_train.shape[1:]))
    model.add(Dense(1000, activation='relu', kernel_regularizer='l2'))
    # model.add(Dropout(0.2))
    model.add(Dense(1000, activation='relu', kernel_regularizer='l2'))
    # model.add(Dropout(0.2))
    model.add(Dense(num_classes, activation='softmax'))
    return model

```

```
def create_model2():
    model = Sequential()
    model.add(Flatten(input_shape = x_train.shape[1:]))
    model.add(Dropout(0.2))
    model.add(Dense(1000, activation='relu', kernel_regularizer='l2'))
    model.add(Dropout(0.5))
    model.add(Dense(1000, activation='relu', kernel_regularizer='l2'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    return model
```

```
from keras.callbacks import ModelCheckpoint

def train(model,f):
    checkpointer = ModelCheckpoint(filepath=f, verbose=0,
                                   save_best_only=True)
    hist = model.fit(x_train, y_train, batch_size=128, epochs=200,
            validation_data=(x_valid, y_valid), callbacks=[checkpointer],
            verbose=0, shuffle=True)
    return hist
train_times = []
```

```
model1 = create_model1()
model1.compile(loss='categorical_crossentropy', optimizer='adagrad', metrics=['accuracy'])
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
hist1 = train(model1,'MLP.best_weights1.hdf5')
train_times.append(time.time()-s)
```

```
model2 = create_model1()
model2.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
hist2 = train(model2,'MLP.best_weights2.hdf5')
train_times.append(time.time()-s)

```

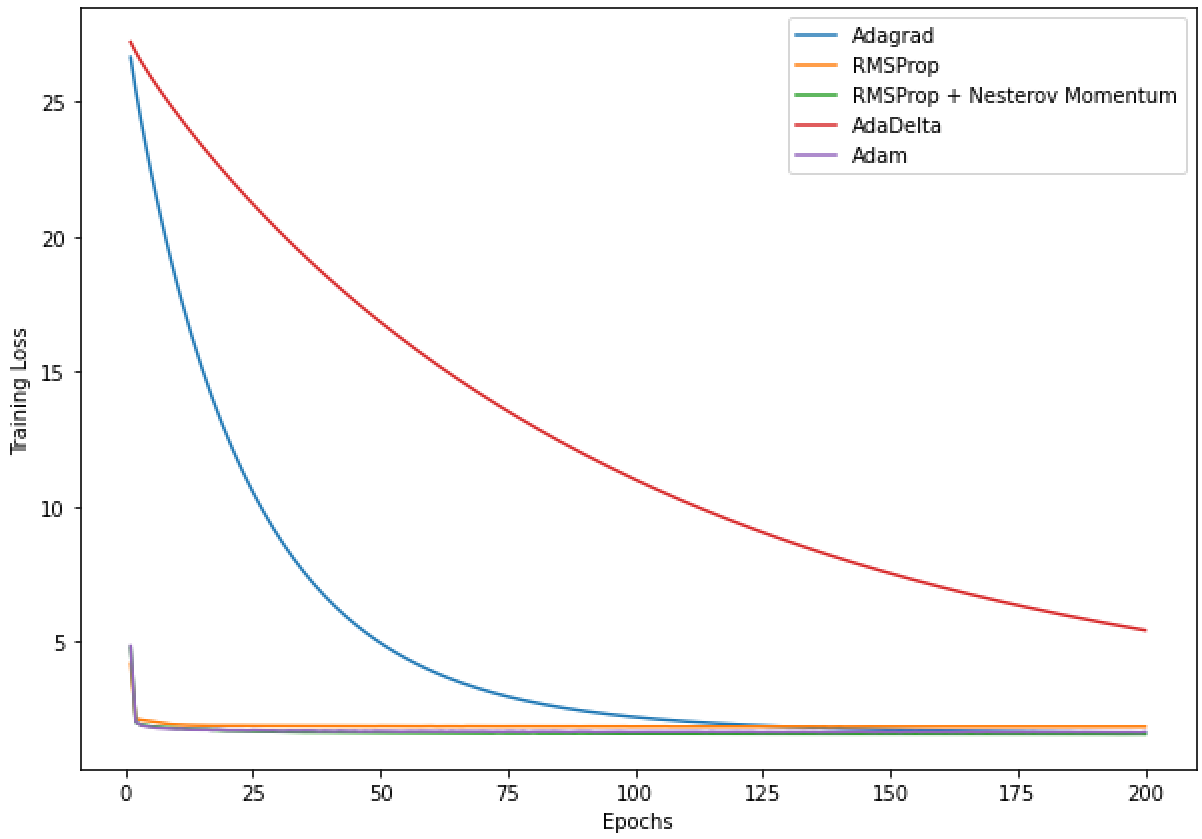
```
# Here we used Nadam instead of rmsprop+nesterov momentum
model3 = create_model1()
model3.compile(loss='categorical_crossentropy', optimizer='nadam', metrics=['accuracy'])
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
hist3 = train(model3,'MLP.best_weights3.hdf5')
train_times.append(time.time()-s)

```

```
model4 = create_model1()
model4.compile(loss='categorical_crossentropy', optimizer='adadelta', metrics=['accuracy']
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
hist4 = train(model4,'MLP.best_weights4.hdf5')
train_times.append(time.time()-s)

```

```
model5 = create_model1()
model5.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
hist5 = train(model5,'MLP.best_weights5.hdf5')
train_times.append(time.time()-s)

```

```
plt.plot([i+1 for i in range(200)],hist1.history['loss'],label='Adagrad')
plt.plot([i+1 for i in range(200)],hist2.history['loss'],label='RMSProp')
plt.plot([i+1 for i in range(200)],hist3.history['loss'],label='RMSProp + Nesterov Momentu
plt.plot([i+1 for i in range(200)],hist4.history['loss'],label='AdaDelta')
plt.plot([i+1 for i in range(200)],hist5.history['loss'],label='Adam')
plt.legend(loc='upper right')
```

```
plt.ylabel('Training Loss')
plt.xlabel('Epochs')
fig = plt.gcf()
fig.set_size_inches(10, 7)
# plt.ylim(0,5)
plt.show()
```



1.2 Nadam performs best and achieves the least loss as can be seen in the graph.

```
model1.load_weights('MLP.best_weights1.hdf5')
mlp_score = model1.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
 Test accuracy: 0.5031999945640564
```

```
model2.load_weights('MLP.best_weights2.hdf5')
mlp_score = model2.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
 Test accuracy: 0.41100001335144043
```

```
model3.load_weights('MLP.best_weights3.hdf5')
mlp_score = model3.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
 Test accuracy: 0.48420000076293945
```

```
model4.load_weights('MLP.best_weights4.hdf5')
mlp_score = model4.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
 Test accuracy: 0.4449999928474426
```

```
model5.load_weights('MLP.best_weights5.hdf5')
mlp_score = model5.evaluate(x_test, y_test, verbose=0)
```

```
print('\n', 'Test accuracy:', mlp_score[1])
```

```
Test accuracy: 0.46369999647140503
```

```

```

```

```

1.3 Adding Dropout Layer

```
import time
train_times_dropout = []
```

```
modeld1 = create_model2()
modeld1.compile(loss='categorical_crossentropy', optimizer='adagrad', metrics=['accuracy']
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
histd1 = train(modeld1,'MLP.best_weights1.hdf5')
train_times_dropout.append(time.time()-s)

```

```
modeld2 = create_model2()
modeld2.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy']
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
histd2 = train(modeld2,'MLP.best_weights2.hdf5')
train_times_dropout.append(time.time()-s)

```

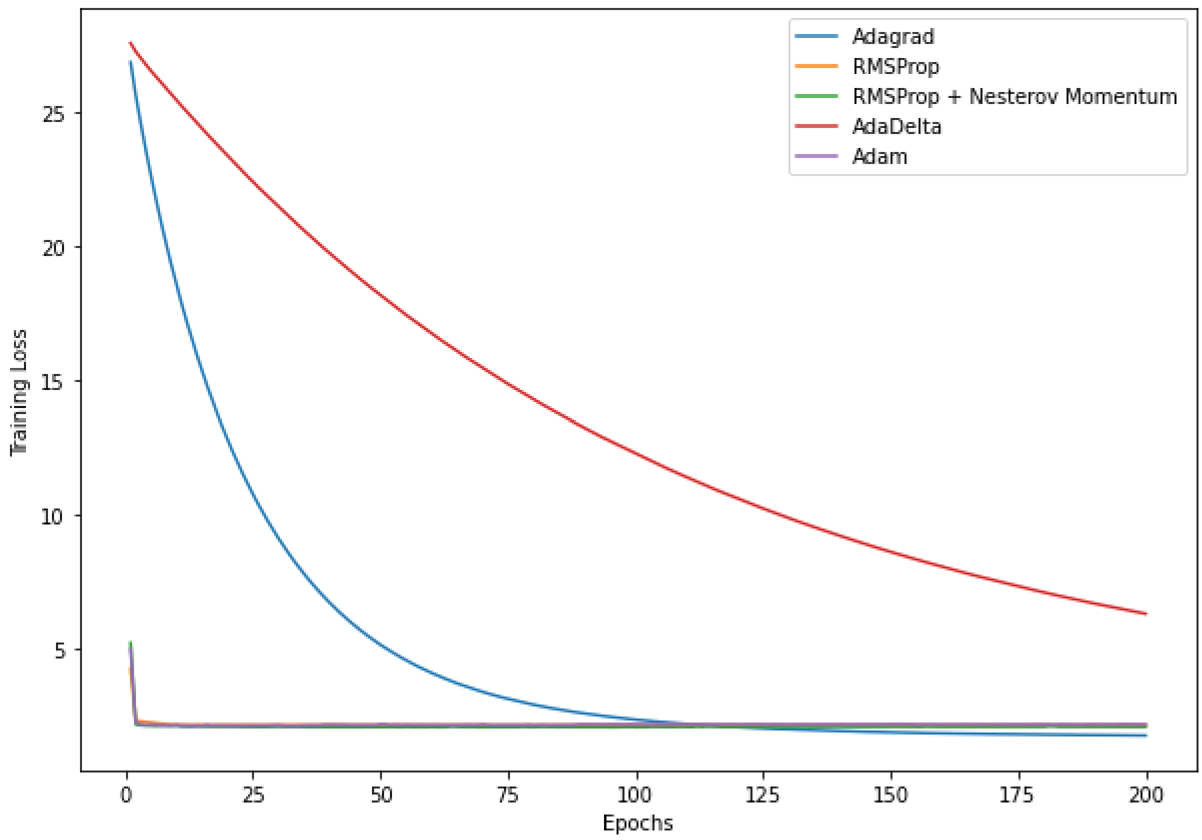
```
modeld3 = create_model2()
modeld3.compile(loss='categorical_crossentropy', optimizer='nadam', metrics=['accuracy'])
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
histd3 = train(modeld3,'MLP.best_weights3.hdf5')
train_times_dropout.append(time.time()-s)

```

```
modeld4 = create_model2()
modeld4.compile(loss='categorical_crossentropy', optimizer='adadelta', metrics=['accuracy'
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
histd4 = train(modeld4,'MLP.best_weights4.hdf5')
train_times_dropout.append(time.time()-s)

```

```
modeld5 = create_model2()
modeld5.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
s = time.time()
histd5 = train(modeld5,'MLP.best_weights5.hdf5')
train_times_dropout.append(time.time()-s)

```

```
plt.plot([i+1 for i in range(200)],histd1.history['loss'],label='Adagrad')
```

```
plt.plot([i+1 for i in range(200)],histd2.history['loss'],label='RMSProp')
plt.plot([i+1 for i in range(200)],histd3.history['loss'],label='RMSProp + Nesterov Moment
plt.plot([i+1 for i in range(200)],histd4.history['loss'],label='AdaDelta')
plt.plot([i+1 for i in range(200)],histd5.history['loss'],label='Adam')
plt.legend(loc='upper right')
plt.ylabel('Training Loss')
plt.xlabel('Epochs')
fig = plt.gcf()
fig.set_size_inches(10, 7)
# plt.ylim(0,5)
plt.show()
```



After adding dropout, the training losses have slightly increased when compared to without dropout for all the models.
Adagrad performs thhe best as can be seen in the graph with dropout

```
modeld1.load_weights('MLP.best_weights1.hdf5')
mlp_score = modeld1.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
 Test accuracy: 0.4805999994277954
```

```
modeld2.load_weights('MLP.best_weights2.hdf5')
mlp_score = modeld2.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
 Test accuracy: 0.3107999861240387
```

```
modeld3.load_weights('MLP.best_weights3.hdf5')
mlp_score = modeld3.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
 Test accuracy: 0.33390000462532043
```

```
model4.load_weights('MLP.best_weights4.hdf5')
mlp_score = model4.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
Test accuracy: 0.39969998598098755
```

```
model5.load_weights('MLP.best_weights5.hdf5')
mlp_score = model5.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', mlp_score[1])
```

```
Test accuracy: 0.4722000062465668
```

```
print(train_times)
print(train_times_dropout)
```

The training time while using dropout is slightly higher for all the 5 methods.

The training times without dropout are
Adagrad: 473.6551239326846, RMSProp: 597.1264545598701, Nadam: 774.2560119758035, Adadelta: 510.25771594047546, Adam: 485.9768748626594 The training times with dropout are
Adagrad: 485.1216578908032, RMSProp: 619.3265489090407, Nadam: 795.9480605789322, Adadelta: 526.6459204540786, Adam: 496.6264847750315

1.4. Test Accuracies:
Adagrad: 0.5031999945640564
RMSProp: 0.41100001335144043
Nadam: 0.48420000076293945
AdaDelta: 0.4449999928474426
Adam: 0.46369999647140503
Adagrad achieves the highest test accuracy without dropout and RMSProp achieves the least

Test Accuracies with dropout:
Adagrad: 0.4805999994277954
RMSProp: 0.3107999861240387
Nadam: 0.33390000462532043
AdaDelta: 0.39969998598098755
Adam: 0.4722000062465668
Adagrad achieves the highest test accuracy with dropout and RMSProp achieves the least

We can observe that all the 5 accuracies have decrease after using dropout.

In [1]:
```
import random
import time
import timeit
import numpy as np
import pandas as pd
import cv2
import matplotlib.pyplot as plt
from pylab import rcParams
rcParams['figure.figsize'] = 12,7
import warnings
warnings.filterwarnings('ignore')
from tqdm.notebook import tqdm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Dropout, Fl
from tensorflow.keras import backend as K
from tensorflow.keras import initializers
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.datasets import mnist, cifar10, fashion_mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import *
import torch
import torch.nn as nn
import torch.optim as optim
import tempfile
from torch.utils.data import Dataset, DataLoader, TensorDataset
```

In [2]:
```
!python --version
```

```
Python 3.7.11
```

In [3]:
```
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

In [26]:
```
if tf.test.gpu_device_name():
    print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))
else:
    print("Please install GPU version of TF")
```

```
Default GPU Device: /device:GPU:0
```

2.1

Summary of FashionMNIST dataset

Dataset size: 70000 \ Training set size: 60000 \ Validation set size: 10000 \ Number of classes: 10 \ Number of images per class: 6000

More specific information is given below

In [4]:
```
(X_train, Y_train), (X_test, Y_test) = fashion_mnist.load_data()
X_train.shape, X_test.shape, Y_train.shape, Y_test.shape
```

Out[4]:
```
((60000, 28, 28), (10000, 28, 28), (60000,), (10000,))
```

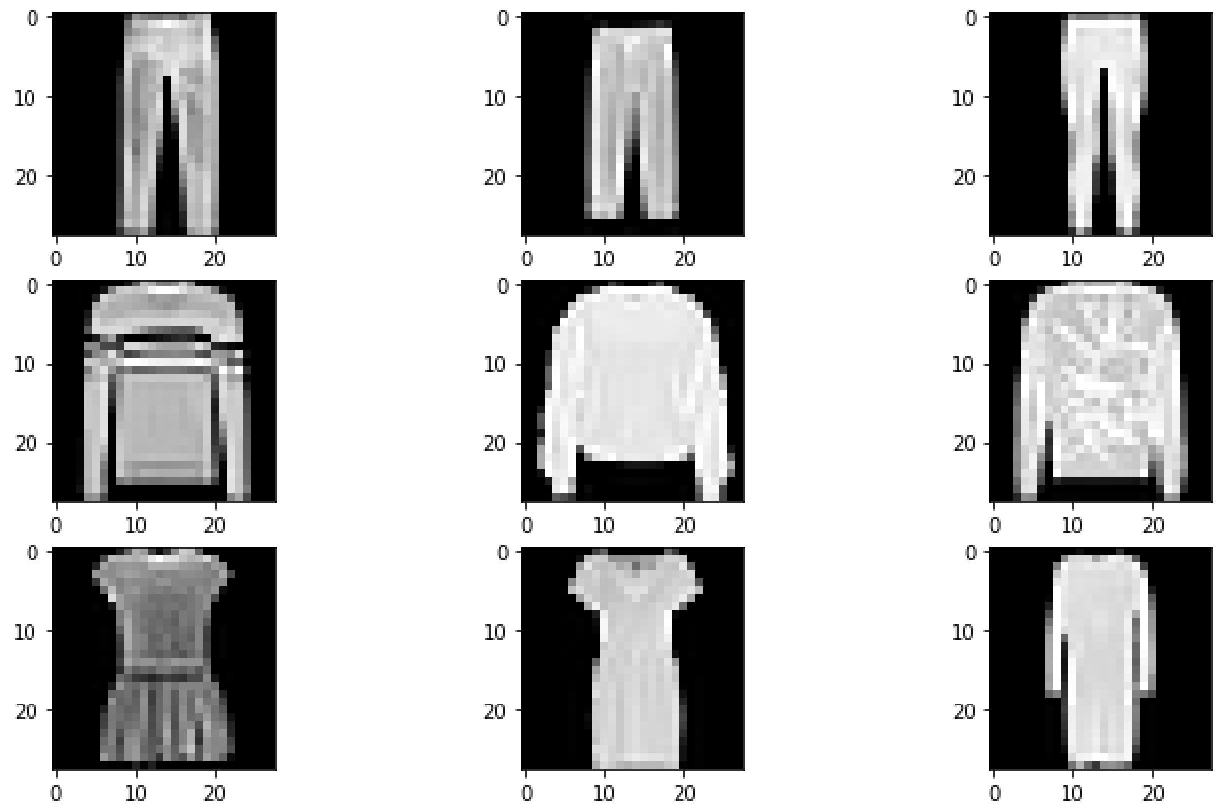
In [5]:

```
class_names = ["top", "trouser", "pullover", "dress", "coat",
               "sandal", "shirt", "sneaker", "bag", "ankle boot"]
print("No.of classes: ", len(np.unique(Y_train)))
print("\nTraining Data:")
for i in range(10):
    print("{}: {}".format(class_names[i], sum(Y_train == i)))
print("\nTest Data")
for i in range(10):
    print("{}: {}".format(class_names[i], sum(Y_test == i)))
```

```
No.of classes:  10

Training Data:
top: 6000
trouser: 6000
pullover: 6000
dress: 6000
coat: 6000
sandal: 6000
shirt: 6000
sneaker: 6000
bag: 6000
ankle boot: 6000

Test Data
top: 1000
trouser: 1000
pullover: 1000
dress: 1000
coat: 1000
sandal: 1000
shirt: 1000
sneaker: 1000
bag: 1000
ankle boot: 1000
```

In [6]:
```
idxs = []
idxs.extend(np.where(Y_train==1)[0][:3])
idxs.extend(np.where(Y_train==2)[0][:3])
idxs.extend(np.where(Y_train==3)[0][:3])
class_lst = class_names[1:4]
print("Classes: ", class_lst)
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(X_train[idxs[i]], cmap='gray')
plt.show()
```

```
Classes:  ['trouser', 'pullover', 'dress']
```



2.2

In [7]:
```
BATCH_SIZE = 64
N_EPOCHS = 5
lrs = [10**(-i) for i in range(9,-2,-1)]
lrs
```

Out[7]:
```
[1e-09, 1e-08, 1e-07, 1e-06, 1e-05, 0.0001, 0.001, 0.01, 0.1, 1, 10]
```

In [8]:
```
X_train = np.array([cv2.resize(x, (32, 32)) for x in X_train])
X_test = np.array([cv2.resize(x, (32, 32)) for x in X_test])
X_train = X_train.astype("float") / 255.0
X_test = X_test.astype("float") / 255.0
X_train = X_train.reshape((X_train.shape[0], 32, 32, 1))
X_test = X_test.reshape((X_test.shape[0], 32, 32, 1))

lb = LabelBinarizer()
Y_train = lb.fit_transform(Y_train)
Y_test = lb.transform(Y_test)
X_train.shape, X_test.shape, Y_train.shape, Y_test.shape
```

Out[8]:
```
((60000, 32, 32, 1), (10000, 32, 32, 1), (60000, 10), (10000, 10))
```

In [9]:
```
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import concatenate
from tensorflow.keras import backend as K

class MiniGoogLeNet:
    @staticmethod
    def conv_module(x, K, kX, kY, stride, chanDim, padding="same"):
        # define a CONV => BN => RELU pattern
        x = Conv2D(K, (kX, kY), strides=stride, padding=padding)(x)
        x = BatchNormalization(axis=chanDim)(x)
        x = Activation("relu")(x)

        # return the block
        return x

    @staticmethod
    def inception_module(x, numK1x1, numK3x3, chanDim):
        # define two CONV modules, then concatenate across the
        # channel dimension
        conv_1x1 = MiniGoogLeNet.conv_module(x, numK1x1, 1, 1,
            (1, 1), chanDim)
        conv_3x3 = MiniGoogLeNet.conv_module(x, numK3x3, 3, 3,
            (1, 1), chanDim)
        x = concatenate([conv_1x1, conv_3x3], axis=chanDim)

        # return the block
        return x

    @staticmethod
    def downsample_module(x, K, chanDim):
        # define the CONV module and POOL, then concatenate
        # across the channel dimensions
        conv_3x3 = MiniGoogLeNet.conv_module(x, K, 3, 3, (2, 2),
            chanDim, padding="valid")
        pool = MaxPooling2D((3, 3), strides=(2, 2))(x)
        x = concatenate([conv_3x3, pool], axis=chanDim)

        # return the block
        return x

    @staticmethod
    def build(width, height, depth, classes):
        # initialize the input shape to be "channels last" and the
        # channels dimension itself
        inputShape = (height, width, depth)
        chanDim = -1

        # if we are using "channels first", update the input shape
        # and channels dimension
        if K.image_data_format() == "channels_first":
            inputShape = (depth, height, width)
            chanDim = 1

        # define the model input and first CONV module
        inputs = Input(shape=inputShape)
        x = MiniGoogLeNet.conv_module(inputs, 96, 3, 3, (1, 1),
            chanDim)

        # two Inception modules followed by a downsample module
        x = MiniGoogLeNet.inception_module(x, 32, 32, chanDim)
        x = MiniGoogLeNet.inception_module(x, 32, 48, chanDim)
        x = MiniGoogLeNet.downsample_module(x, 80, chanDim)

        # four Inception modules followed by a downsample module
```

```
        x = MiniGoogLeNet.inception_module(x, 112, 48, chanDim)
        x = MiniGoogLeNet.inception_module(x, 96, 64, chanDim)
        x = MiniGoogLeNet.inception_module(x, 80, 80, chanDim)
        x = MiniGoogLeNet.inception_module(x, 48, 96, chanDim)
        x = MiniGoogLeNet.downsample_module(x, 96, chanDim)

        # two Inception modules followed by global POOL and dropout
        x = MiniGoogLeNet.inception_module(x, 176, 160, chanDim)
        x = MiniGoogLeNet.inception_module(x, 176, 160, chanDim)
        x = AveragePooling2D((7, 7))(x)
        x = Dropout(0.5)(x)

        # softmax classifier
        x = Flatten()(x)
        x = Dense(classes)(x)
        x = Activation("softmax")(x)

        # create the model
        model = Model(inputs, x, name="googlenet")

        # return the constructed network architecture
        return model
```

In [16]:

```
model = MiniGoogLeNet.build(width=32, height=32, depth=1, classes=10)
train_losses = []
for lr in lrs:
    print("Learning rate:", lr)
    opt = SGD(lr=lr, momentum=0.9)
    model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["
    model_info = model.fit(X_train, Y_train, epochs=N_EPOCHS,validation_spli
    train_losses.append(model_info.history['loss'][-1])
```

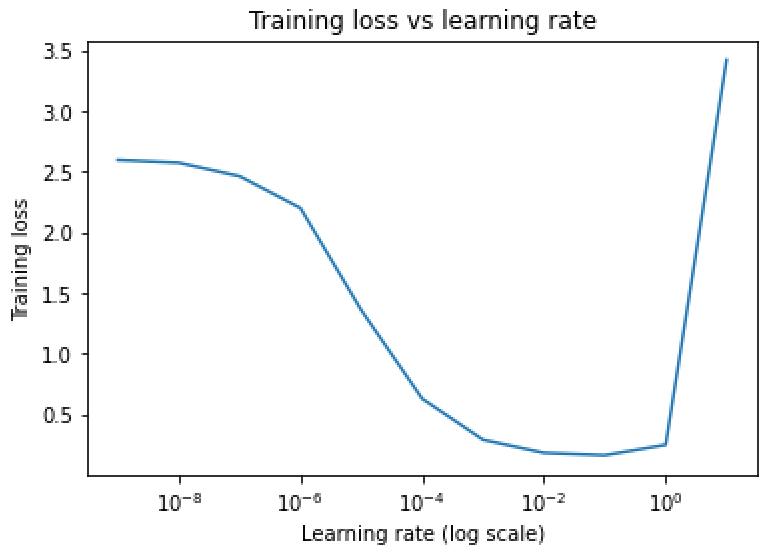
```
Learning rate: 1e-09
Epoch 1/5
750/750 [==============================] - 83s 108ms/step - loss: 2.5984 - a
ccuracy: 0.1024 - val_loss: 2.3731 - val_accuracy: 0.1017
Epoch 2/5
750/750 [==============================] - 80s 107ms/step - loss: 2.5959 - a
ccuracy: 0.1020 - val_loss: 2.4459 - val_accuracy: 0.0900
Epoch 3/5
750/750 [==============================] - 80s 107ms/step - loss: 2.5970 - a
ccuracy: 0.1044 - val_loss: 2.4446 - val_accuracy: 0.0906
Epoch 4/5
750/750 [==============================] - 80s 107ms/step - loss: 2.5947 - a
ccuracy: 0.1032 - val_loss: 2.4445 - val_accuracy: 0.0901
Epoch 5/5
750/750 [==============================] - 80s 107ms/step - loss: 2.5983 - a
ccuracy: 0.1017 - val_loss: 2.4443 - val_accuracy: 0.0899
Learning rate: 1e-08
Epoch 1/5
750/750 [==============================] - 84s 109ms/step - loss: 2.5954 - a
```

In [10]:

```
print(train_losses)
```

```
[2.6195530891418457, 2.62713623046875, 2.5896055698394775, 2.3715314865112305,
1.4939004182815552]
```

In [17]:
```
# plt.figure(figsize=(12,6))
plt.semilogx(lrs, train_losses)
plt.title("Training loss vs learning rate")
plt.xlabel('Learning rate (log scale)')
plt.ylabel('Training loss')
plt.show()
```



From the plot below, we can see that the loss starts decreasing before around 1e-5 and starts to saturate around 1e-1. So, we can choose these two values as lr_{min} and lr_{max}.

In [10]:
```
lrmin = 1e-5
lrmax = 1e-1
```

Part 3

In [11]:
```
class CyclicLR(Callback):
    def __init__(self, base_lr=0.001, max_lr=0.006, step_size=2000., mode='t
                 gamma=1., scale_fn=None, scale_mode='cycle'):
        super(CyclicLR, self).__init__()

        self.base_lr = base_lr
        self.max_lr = max_lr
        self.step_size = step_size
        self.mode = mode
        self.gamma = gamma
        if scale_fn == None:
            if self.mode == 'triangular':
                self.scale_fn = lambda x: 1.
                self.scale_mode = 'cycle'
            elif self.mode == 'triangular2':
                self.scale_fn = lambda x: 1/(2.**(x-1))
                self.scale_mode = 'cycle'
            elif self.mode == 'exp_range':
                self.scale_fn = lambda x: gamma**(x)
                self.scale_mode = 'iterations'
        else:
            self.scale_fn = scale_fn
            self.scale_mode = scale_mode
        self.clr_iterations = 0.
        self.trn_iterations = 0.
        self.history = {}

        self._reset()

    def _reset(self, new_base_lr=None, new_max_lr=None,
               new_step_size=None):
        """Resets cycle iterations.
        Optional boundary/step size adjustment.
        """
        if new_base_lr != None:
            self.base_lr = new_base_lr
        if new_max_lr != None:
            self.max_lr = new_max_lr
        if new_step_size != None:
            self.step_size = new_step_size
        self.clr_iterations = 0.

    def clr(self):
        cycle = np.floor(1+self.clr_iterations/(2*self.step_size))
        x = np.abs(self.clr_iterations/self.step_size - 2*cycle + 1)
        if self.scale_mode == 'cycle':
            return self.base_lr + (self.max_lr-self.base_lr)*np.maximum(0, (
        else:
            return self.base_lr + (self.max_lr-self.base_lr)*np.maximum(0, (

    def on_train_begin(self, logs={}):
        logs = logs or {}

        if self.clr_iterations == 0:
            K.set_value(self.model.optimizer.lr, self.base_lr)
        else:
            K.set_value(self.model.optimizer.lr, self.clr())

    def on_batch_end(self, epoch, logs=None):

        logs = logs or {}
        self.trn_iterations += 1
        self.clr_iterations += 1

        self.history.setdefault('lr', []).append(K.get_value(self.model.opti
        self.history.setdefault('iterations', []).append(self.trn_iterations

        for k, v in logs.items():
            self.history.setdefault(k, []).append(v)

        K.set_value(self.model.optimizer.lr, self.clr())
```

In [12]:
```
clr = CyclicLR(base_lr=lrmin, max_lr=lrmax, mode='exp_range', step_size=4*(X
```

In [13]:
```
opt = SGD(lr=lrmin, momentum=0.9)
model = MiniGoogLeNet.build(width=32, height=32, depth=1, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accu
H = model.fit(X_train,Y_train,epochs=30,validation_split=0.2,batch_size=BATC
              steps_per_epoch=X_train.shape[0]//BATCH_SIZE,callbacks=[clr,Ea
```

```
Epoch 1/30
WARNING:tensorflow:Callback method `on_train_batch_end` is slow compared to the
batch time (batch time: 0.0314s vs `on_train_batch_end` time: 0.0918s). Check y
our callbacks.
937/937 - 113s - loss: 0.6670 - accuracy: 0.7610 - val_loss: 1.0657 - val_accur
acy: 0.7113
Epoch 2/30
937/937 - 98s - loss: 0.3304 - accuracy: 0.8827 - val_loss: 0.4678 - val_accura
cy: 0.8423
Epoch 3/30
937/937 - 103s - loss: 0.2784 - accuracy: 0.8999 - val_loss: 0.3099 - val_accur
acy: 0.8992
Epoch 4/30
937/937 - 100s - loss: 0.2429 - accuracy: 0.9149 - val_loss: 0.4182 - val_accur
acy: 0.8770
Epoch 5/30
937/937 - 100s - loss: 0.1958 - accuracy: 0.9305 - val_loss: 0.2101 - val_accur
acy: 0.9218
Epoch 6/30
937/937 - 100s - loss: 0.1502 - accuracy: 0.9463 - val_loss: 0.2374 - val_accur
acy: 0.9243
Epoch 7/30
937/937 - 101s - loss: 0.1108 - accuracy: 0.9608 - val_loss: 0.2293 - val_accur
acy: 0.9232
Epoch 8/30
937/937 - 102s - loss: 0.0650 - accuracy: 0.9772 - val_loss: 0.1800 - val_accur
acy: 0.9429
Epoch 9/30
937/937 - 108s - loss: 0.0454 - accuracy: 0.9844 - val_loss: 0.2476 - val_accur
acy: 0.9273
Epoch 10/30
937/937 - 104s - loss: 0.0744 - accuracy: 0.9730 - val_loss: 0.2838 - val_accur
acy: 0.9128
Epoch 11/30
937/937 - 103s - loss: 0.1113 - accuracy: 0.9585 - val_loss: 0.2790 - val_accur
acy: 0.9132
```

In [15]:
```
epochs = np.arange(0, 8)
plt.style.use("ggplot")
plt.figure()
plt.plot(epochs, H.history["loss"][:-3], label="Train Loss")
plt.plot(epochs, H.history["val_loss"][:-3], label="Validation Loss")
plt.plot(epochs, H.history["accuracy"][:-3], label="Train Accuracy")
plt.plot(epochs, H.history["val_accuracy"][:-3], label="Validation Accuracy"
plt.title("Loss and Accuracy curves")
plt.xlabel("Epoch")
plt.ylabel("Loss/Accuracy")
plt.legend()
# plt.ylim(0,5)
plt.show()
```

2.4

In [16]:
```
bsizes = [2**i for i in range(6,12)]
n_epoch_set = [5*b//64 for b in bsizes]
bsizes, n_epoch_set
```

Out[16]: ([64, 128, 256, 512, 1024, 2048], [5, 10, 20, 40, 80, 160])

In [17]:

```
train_losses_bsizes = []
for i in range(len(bsizes[:2])):
    print("\nBatch size: ", bsizes[i])
    opt = SGD(lr=lrmin, momentum=0.9)
    model = MiniGoogLeNet.build(width=32, height=32, depth=1, classes=10)
    model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["
    H = model.fit(X_train,Y_train,epochs=n_epoch_set[i],validation_split=0.2
    train_losses_bsizes.append(H.history['loss'][-1])
```

```
Batch size:  64
Epoch 1/5
750/750 - 80s - loss: 2.4463 - accuracy: 0.1356 - val_loss: 2.1171 - val_accura
cy: 0.1972
Epoch 2/5
750/750 - 79s - loss: 2.1030 - accuracy: 0.2344 - val_loss: 1.8096 - val_accura
cy: 0.4769
Epoch 3/5
750/750 - 80s - loss: 1.8768 - accuracy: 0.3384 - val_loss: 1.6154 - val_accura
cy: 0.6198
Epoch 4/5
750/750 - 81s - loss: 1.7066 - accuracy: 0.4161 - val_loss: 1.4740 - val_accura
cy: 0.6705
Epoch 5/5
750/750 - 83s - loss: 1.5706 - accuracy: 0.4685 - val_loss: 1.3616 - val_accura
cy: 0.6910

Batch size:  128
Epoch 1/10
375/375 - 82s - loss: 2.4588 - accuracy: 0.1172 - val_loss: 2.4508 - val_accura
cy: 0.0995
Epoch 2/10
375/375 - 77s - loss: 2.2135 - accuracy: 0.1855 - val_loss: 2.0029 - val_accura
cy: 0.2806
Epoch 3/10
375/375 - 77s - loss: 2.0344 - accuracy: 0.2627 - val_loss: 1.8193 - val_accura
cy: 0.4225
Epoch 4/10
375/375 - 76s - loss: 1.8999 - accuracy: 0.3172 - val_loss: 1.6999 - val_accura
cy: 0.4909
Epoch 5/10
375/375 - 76s - loss: 1.7894 - accuracy: 0.3676 - val_loss: 1.5998 - val_accura
cy: 0.5471
Epoch 6/10
375/375 - 77s - loss: 1.6966 - accuracy: 0.4028 - val_loss: 1.5186 - val_accura
cy: 0.5831
Epoch 7/10
375/375 - 77s - loss: 1.6233 - accuracy: 0.4284 - val_loss: 1.4520 - val_accura
cy: 0.6086
Epoch 8/10
375/375 - 77s - loss: 1.5630 - accuracy: 0.4550 - val_loss: 1.3924 - val_accura
cy: 0.6305
Epoch 9/10
375/375 - 77s - loss: 1.5056 - accuracy: 0.4768 - val_loss: 1.3408 - val_accura
cy: 0.6423
Epoch 10/10
375/375 - 77s - loss: 1.4531 - accuracy: 0.4978 - val_loss: 1.2941 - val_accura
cy: 0.6603
```

In [18]:

```
print(train_losses_bsizes)
```

```
[1.5705708265304565, 1.4531182050704956]
```

In [19]:

```
for i in range(2,4):
    print("\nBatch size: ", bsizes[i])
    opt = SGD(lr=lrmin, momentum=0.9)
    model = MiniGoogLeNet.build(width=32, height=32, depth=1, classes=10)
    model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["
    H = model.fit(X_train,Y_train,epochs=n_epoch_set[i],validation_split=0.2
    train_losses_bsizes.append(H.history['loss'][-1])
```

```
Batch size:  256
Epoch 1/20
188/188 - 75s - loss: 2.6087 - accuracy: 0.1045 - val_loss: 2.4360 - val_acc
uracy: 0.1003
Epoch 2/20
188/188 - 74s - loss: 2.4539 - accuracy: 0.1298 - val_loss: 2.4061 - val_acc
uracy: 0.1062
Epoch 3/20
188/188 - 74s - loss: 2.3222 - accuracy: 0.1609 - val_loss: 2.2721 - val_acc
uracy: 0.2036
Epoch 4/20
188/188 - 75s - loss: 2.2099 - accuracy: 0.1939 - val_loss: 2.0500 - val_acc
uracy: 0.2988
Epoch 5/20
188/188 - 75s - loss: 2.1152 - accuracy: 0.2276 - val_loss: 1.9304 - val_acc
uracy: 0.3708
Epoch 6/20
188/188 - 76s - loss: 2.0330 - accuracy: 0.2614 - val_loss: 1.8523 - val_acc
```

In [20]:

```
print(train_losses_bsizes)
```

```
[1.5705708265304565, 1.4531182050704956, 1.4617187976837158, 1.466109037399292]
```

In [21]:

```
print("\nBatch size: ", bsizes[4])
opt = SGD(lr=lrmin, momentum=0.9)
model = MiniGoogLeNet.build(width=32, height=32, depth=1, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accu
H = model.fit(X_train,Y_train,epochs=n_epoch_set[i],validation_split=0.2,bat
train_losses_bsizes.append(H.history['loss'][-1])
```

```
Batch size:  1024
Epoch 1/40
94/94 - 69s - loss: 2.6878 - accuracy: 0.1036 - val_loss: 2.3906 - val_accur
acy: 0.1005
Epoch 2/40
94/94 - 79s - loss: 2.6029 - accuracy: 0.1168 - val_loss: 2.5125 - val_accur
acy: 0.1005
Epoch 3/40
94/94 - 85s - loss: 2.5211 - accuracy: 0.1321 - val_loss: 2.4737 - val_accur
acy: 0.1005
Epoch 4/40
94/94 - 75s - loss: 2.4530 - accuracy: 0.1459 - val_loss: 2.3865 - val_accur
acy: 0.1107
Epoch 5/40
94/94 - 73s - loss: 2.3923 - accuracy: 0.1618 - val_loss: 2.3555 - val_accur
acy: 0.1182
Epoch 6/40
94/94 - 84s - loss: 2.3398 - accuracy: 0.1786 - val_loss: 2.3033 - val_accur
```

In [22]:

```
print(train_losses_bsizes)
```

```
[1.5705708265304565, 1.4531182050704956, 1.4617187976837158, 1.466109037399292,
1.4693599939346313]
```

In [32]:

```

print("\nBatch size: ", bsizes[5])
opt = SGD(lr=lrmin, momentum=0.9)
model = MiniGoogLeNet.build(width=32, height=32, depth=1, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accu
H = model.fit(X_train,Y_train,epochs=n_epoch_set[i],validation_split=0.2,bat
train_losses_bsizes.append(H.history['loss'][-1])
```

```

Batch size:  2048
Epoch 1/40
94/94 - 67s - loss: 2.6671 - accuracy: 0.1065 - val_loss: 2.3459 - val_accur
acy: 0.0983
Epoch 2/40
94/94 - 67s - loss: 2.5580 - accuracy: 0.1156 - val_loss: 2.4036 - val_accur
acy: 0.1140
Epoch 3/40
94/94 - 68s - loss: 2.4727 - accuracy: 0.1307 - val_loss: 2.4159 - val_accur
acy: 0.1320
Epoch 4/40
94/94 - 67s - loss: 2.3960 - accuracy: 0.1451 - val_loss: 2.3744 - val_accur
acy: 0.1320
Epoch 5/40
94/94 - 67s - loss: 2.3302 - accuracy: 0.1636 - val_loss: 2.3138 - val_accur
acy: 0.1495
Epoch 6/40
94/94 - 68s - loss: 2.2701 - accuracy: 0.1821 - val_loss: 2.2222 - val_accur
```

In [ ]:

```

print("\nBatch size: ", bsizes[5])
opt = SGD(lr=lrmin, momentum=0.9)
model = MiniGoogLeNet.build(width=32, height=32, depth=1, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accu
H = model.fit(X_train,Y_train,epochs=n_epoch_set[i],validation_split=0.2,bat
train_losses_bsizes.append(H.history['loss'][-1])
```

In [24]:

```
print(train_losses_bsizes)
```

```
[1.5705708265304565, 1.4531182050704956, 1.4617187976837158, 1.466109037399292,
1.4693599939346313, 1.4488837718963623]
```

In [ ]:

```

```

In [25]:

```
# Prepare the training dataset.
batch_size = 512
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, Y_train))
train_dataset = train_dataset.shuffle(buffer_size=1024).batch(batch_size)

# Prepare the validation dataset.
val_dataset = tf.data.Dataset.from_tensor_slices((X_test, Y_test))
val_dataset = val_dataset.batch(batch_size)
```

In [30]:
```
loss_fn = tf.keras.losses.CategoricalCrossentropy()
optimizer = SGD(lr=lrmin, momentum=0.9)
model = MiniGoogLeNet.build(width=32, height=32, depth=1, classes=10)
epochs = 320
print("Batch size: 4096")
for epoch in range(epochs):
    print("\nEpoch: {}/{}".format(epoch+1, epochs))
    loss_value = 0
    total_loss = 0
    # Iterate over the batches of the dataset.
    for step, (x_batch_train, y_batch_train) in tqdm(enumerate(train_dataset
        # Open a GradientTape to record the operations run
        # during the forward pass, which enables auto-differentiation.
        with tf.GradientTape() as tape:

            # Run the forward pass of the layer.
            # The operations that the layer applies
            # to its inputs are going to be recorded
            # on the GradientTape.
            y_batch_pred = model(x_batch_train, training=True)  # Logits for

            # Compute the loss value for this minibatch.
            total_loss += loss_fn(y_batch_train, y_batch_pred).numpy()
            loss_value += loss_fn(y_batch_train, y_batch_pred)

        # Use the gradient tape to automatically retrieve
        # the gradients of the trainable variables with respect to the loss.
        if step != 0 and step % 8 == 0:
            grads = tape.gradient(loss_value, model.trainable_weights)

            # Run one step of gradient descent by updating
            # the value of the variables to minimize the loss.
            optimizer.apply_gradients(zip(grads, model.trainable_weights))
            loss_value = 0
    print("Loss: ", float(total_loss/len(train_dataset)))
# train_losses_bsizes.append(float(total_loss/len(train_dataset)))
        # if step % 200 == 0:
        #     print(
        #         "Training loss (for one batch) at step %d: %.4f"
        #         % (step, float(loss_value))
        #     )
        #     print("Seen so far: %s samples" % ((step + 1) * batch_size))
```

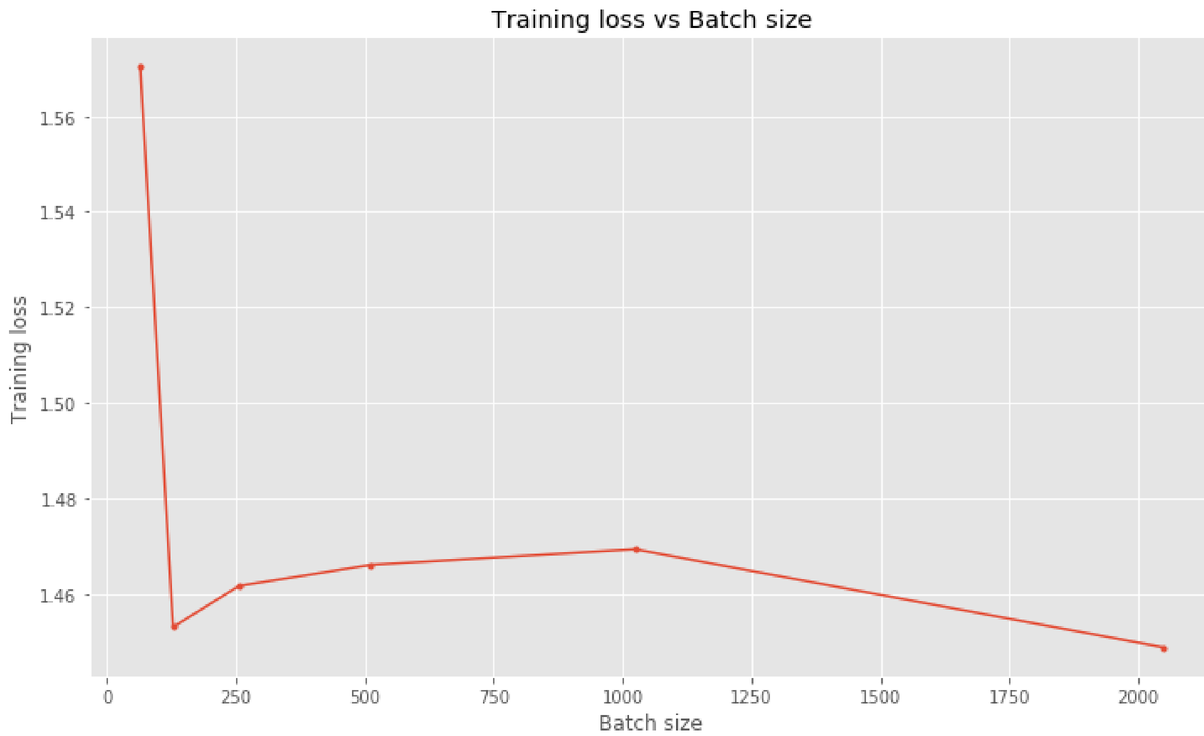
...

In [ ]:
```
train_losses_bsizes.append(float(total_loss/len(train_dataset)))
```

In [ ]:
```

```

In [ ]:
```

```

In [43]:
```
plt.plot(bsizes, train_losses_bsizes[:6], marker='.')
plt.title('Training loss vs Batch size')
plt.xlabel('Batch size')
plt.ylabel('Training loss')
plt.show()
```



2.4,2.5

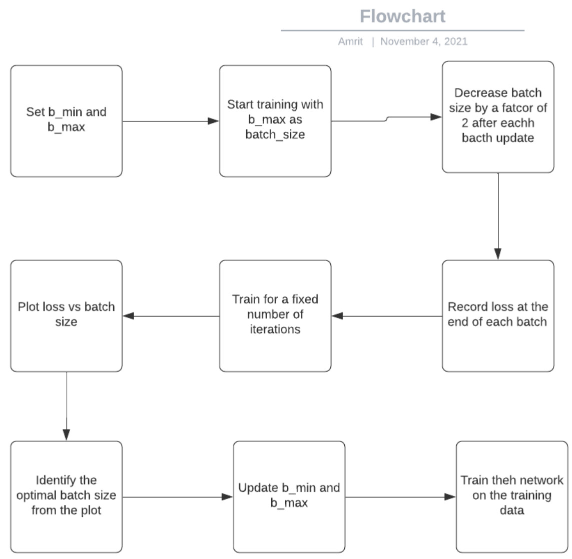
From batch size=64 to 128 we see a clear drop in loss. From 128 to 2048 the loss almost remains the same. We then expect it to increase which is similar to the curve obsereved for learning rates. Therefore we choose $b_{min} = 128$ and $b_{max} = 2048$

2.6

We exponentially decrease the batch size for the cyclical batch size policy as was done in the case with cyclic learning rate.

In [44]:
```
class LearningRateFinder:
    def __init__(self, model, stopFactor=4, beta=0.98):
        # store the model, stop factor, and beta value (for computing
        # a smoothed, average loss)
        self.model = model
        self.stopFactor = stopFactor
        self.beta = beta
        # initialize our list of learning rates and losses,
        # respectively
        self.lrs = []
        self.losses = []
        # initialize our learning rate multiplier, average loss, best
        # loss found thus far, current batch number, and weights file
        self.lrMult = 1
        self.avgLoss = 0
        self.bestLoss = 1e9
        self.batchNum = 0
        self.weightsFile = None

    def reset(self):
        # re-initialize all variables from our constructor
        self.lrs = []
        self.losses = []
        self.lrMult = 1
        self.avgLoss = 0
        self.bestLoss = 1e9
        self.batchNum = 0
        self.weightsFile = None

    def is_data_iter(self, data):
        # define the set of class types we will check for
        iterClasses = ["NumpyArrayIterator", "DirectoryIterator",
            "DataFrameIterator", "Iterator", "Sequence"]
        # return whether our data is an iterator
        return data.__class__.__name__ in iterClasses

    def on_batch_end(self, batch, logs):
        # grab the current learning rate and add log it to the list of
        # learning rates that we've tried
        lr = K.get_value(self.model.optimizer.lr)
        self.lrs.append(lr)
        # grab the loss at the end of this batch, increment the total
        # number of batches processed, compute the average average
        # loss, smooth it, and update the losses list with the
        # smoothed value
        l = logs["loss"]
        self.batchNum += 1
        self.avgLoss = (self.beta * self.avgLoss) + ((1 - self.beta) * l)
        smooth = self.avgLoss / (1 - (self.beta ** self.batchNum))
        self.losses.append(smooth)
        # compute the maximum loss stopping factor value
        stopLoss = self.stopFactor * self.bestLoss
        # check to see whether the loss has grown too large
        if self.batchNum > 1 and smooth > stopLoss:
            # stop returning and return from the method
            self.model.stop_training = True
            return
        # check to see if the best loss should be updated
        if self.batchNum == 1 or smooth < self.bestLoss:
            self.bestLoss = smooth
        # increase the learning rate
        lr *= self.lrMult
        K.set_value(self.model.optimizer.lr, lr)

    def find(self, trainData, startLR, endLR, epochs=None,
        stepsPerEpoch=None, batchSize=32, sampleSize=2048,
        verbose=1):
        # reset our class-specific variables
        self.reset()
        # determine if we are using a data generator or not
        useGen = self.is_data_iter(trainData)
        # if we're using a generator and the steps per epoch is not
        # supplied, raise an error
```

```
        if useGen and stepsPerEpoch is None:
            msg = "Using generator without supplying stepsPerEpoch"
            raise Exception(msg)
        # if we're not using a generator then our entire dataset must
        # already be in memory
        elif not useGen:
            # grab the number of samples in the training data and
            # then derive the number of steps per epoch
            numSamples = len(trainData[0])
            stepsPerEpoch = np.ceil(numSamples / float(batchSize))
        # if no number of training epochs are supplied, compute the
        # training epochs based on a default sample size
        if epochs is None:
            epochs = int(np.ceil(sampleSize / float(stepsPerEpoch)))

        # compute the total number of batch updates that will take
        # place while we are attempting to find a good starting
        # learning rate
        numBatchUpdates = epochs * stepsPerEpoch
        # derive the learning rate multiplier based on the ending
        # learning rate, starting learning rate, and total number of
        # batch updates
        self.lrMult = (endLR / startLR) ** (1.0 / numBatchUpdates)
        # create a temporary file path for the model weights and
        # then save the weights (so we can reset the weights when we
        # are done)
        self.weightsFile = tempfile.mkstemp()[1]
        self.model.save_weights(self.weightsFile)
        # grab the *original* learning rate (so we can reset it
        # later), and then set the *starting* learning rate
        origLR = K.get_value(self.model.optimizer.lr)
        K.set_value(self.model.optimizer.lr, startLR)

        # construct a callback that will be called at the end of each
        # batch, enabling us to increase our learning rate as training
        # progresses
        callback = LambdaCallback(on_batch_end=lambda batch, logs:
            self.on_batch_end(batch, logs))
        # check to see if we are using a data iterator
        if useGen:
            self.model.fit(
                x=trainData,
                steps_per_epoch=stepsPerEpoch,
                epochs=epochs,
                verbose=verbose,
                callbacks=[callback])
        # otherwise, our entire training data is already in memory
        else:
            # train our model using Keras' fit method
            self.model.fit(
                x=trainData[0], y=trainData[1],
                batch_size=batchSize,
                epochs=epochs,
                callbacks=[callback],
                verbose=verbose)
        # restore the original model weights and learning rate
        self.model.load_weights(self.weightsFile)
        K.set_value(self.model.optimizer.lr, origLR)

    def plot_loss(self, skipBegin=10, skipEnd=1, title=""):
        # grab the learning rate and losses values to plot
        lrs = self.lrs[skipBegin:-skipEnd]
        losses = self.losses[skipBegin:-skipEnd]
        # plot the learning rate vs. loss
        plt.plot(lrs, losses)
        plt.xscale("log")
        plt.xlabel("Learning Rate (Log Scale)")
        plt.ylabel("Loss")
        # if the title is not empty, add it to the plot
        if title != "":
            plt.title(title)
```



2.6 We can train the model using the following steps - -Set small and large batch size bounds. \ -Train network \ -Decrease batch size in multiples of 2 after each update. \ -Record loss & batch size at the end, train for fixed iterations \ -Plot loss & batch size \ -Examine plot and identify the optimal batch size \ -Update the batch sizes \ -Train network on full set of data \

2.7 The cyclical learning rate policy gives a better accuracy when compared to the cyclical batch size policy.

In [ ]:

```

```

Problem 3

We will use the following formulae to calculate
Conv Layer:
Input: wxhxd, k filters of size f, stride s and padding p
Ouput: ((w-f+2p)/s + 1, (-f+2p)/s + 1, k)
Pool Layer:
Input wxhxd, Pooling size f and stride = s
Output: ((w-f)/s + 1, (h-f)/s + 1, d)

3.1 AlexNet no of parameters
Conv-1: 11x11x3x96 + 96 = 34944
Conv-2: 2x(5x5x48x128 + 128) = 307456
Conv-3: 3x3x(2x128)x(2x192) + 2x192 = 885120
Conv-4: 2x(3x3x192x192 + 192) = 663936
Conv-5: 2x(3x3x192x128 + 128) = 442624
Dense-1: 9216x4096 + 4096 = 37752832
Dense-2: 4096x4096 + 4096 = 16781312
Dense-3: 4096x1000 + 1000 = 4097000
Total no of parameters = 60965224.

3.2 VGG-19 no of parameters without inicluding biases.
I have denoted the values already present in the table as _.
In each of the lines, the first value corresponds to the activations and the second to the number of parameters.
in: _ _
c1: _ _
c2: _ _
p1: _ _
c3: 112x112x128 = 1605632, 3x3x64x128 = 73728
c4: 112x112x128 = 1605632, 3x3x128x128 = 147456
p2: _ _
c5: 56x56x256 = 802816, 3x3x128x256 = 294912
c6: _ _
c7: 56x56x256 = 802816, 3x3x256x256 = 589824
c8: 56x56x256 = 802816, 3x3x256x256 = 589824
p3: 28x28x256 = 200704, _
c9: _ _
c10: 28x28x512 = 401408, 3x3x512x512 = 2359296
c11: _, 3x3x512x512 = 2359296
c12: 28x28x512 = 401408, 3x3x512x512 = 2359296
p4: 14x14x512 = 100352, _
c13: 14x14x512 = 100352, 3x3x512x512 = 2359296
c14: 14x14x512 = 100352, 3x3x512x512 = 2359296
c15: 14x14x512 = 100352, 3x3x512x512 = 2359296
c16: 14x14x512 = 100352, 3x3x512x512 = 2359296
p5: 7x7x512 = 25088, 0
fc1: _, 7x7x512x4096 = 102760488

fc2: _ _

fc3: _ 1000x4096 = 4096000

Total activations = 16542184

Total parameters = 140113640

3.3 From thhe reference given, the relation between receptive fields of the $(l-1)^{th}$ layer and the l^{th} layer is

$$r_{l-1} = s_l r_l + (k_l - s_l)$$

where k_l is the kernel size and s_l is the stride.

Solving the recurrence relation, we get

$$r_0 = \sum_{l=1}^{L}((k_l - 1)\Pi_{i=1}^{l-1} s_i) + 1$$

So if we have a stack of N convolutional layers each of filter size FxF then

$$r_0 = \sum_{l=1}^{N}((F - 1)\Pi_{i=1}^{l-1} s_i) + 1$$

For $s_i = 1$, we have

$$r_0 = \sum_{l=1}^{N}(F-1) + 1 = N(F-1) + 1 = NF - N + 1$$

For 1 convolutional layer with filter size NF-N+1,

$$r_0 = \sum_{l=1}^{N}((k_l - 1)\Pi_{i=1}^{l-1} s_i) + 1 = k_l - 1 + 1 = NF - N + 1$$

Therefore a stack of N convolution layers each of filter size F×F has the same receptive field as one convolution layer with filter of size (N F −N + 1) ×(N F −N + 1).

For 3 filters of size 5x5, N=3 and F=5, we get a receptive field = 3x5-3+1 = 13

3.4.(a) The goal behind designing the inception module is to increase the depth and width of the network while keeping it computationally budget constant. Deep networks contain computations involving sparse matrices. Inception modules attempt to convert these sparse matrices into denser matrices which enable optimal usage of resources. The inception module is based on finding how an optimal local sparse structure in a CNN can be approximated by dense components.

3.4.(b) Naive: (i) 32x32x128 (ii) 32x32x192 (iii) 32x32x96 (iv) 32x32x256

Output of filter concatenation: 32x32x(128+192+96+256) = 32x32x672

Dimensionality Reduction: (i) 32x32x128 (ii) 32x32x192 (iii) 32x32x96 (iv) 32x32x64

Output of filter concatenation: 32x32x(128+192+96+64) = 32x32x480

3.4.(c) Naive: (i) 32x32x256x1x1x128 = 33554432 (ii) 32x32x256x3x3x192 = 452984832 (iii) 32x32x256x5x5x96 = 629145600

Total convolutional operations = 1115684864

Dimensionality Reduction: (i)Conv(1x1) 32x32x256x1x1x128 + 32x32x256x1x1x128 +32x32x256x1x1x64 = 92274688 (ii)Conv(3x3) 32x32x9x128x192 = 226492416 (iii) Conv(5x5) 32x32x25x32x96 = 78643200

Total operations = 397410304

3.4.(d) The naive approach has moore than 2.8 times the number of convolutional computations when compared to the dimensionality reduction based approach. Therefore using the DR approach the computational expenses are reduced almost to a third.

Problem 4

4.1. Cutout Regularization Cutout is a regularizatioin technique which removes random contiguous(for ex square) portions of the input image. This in some way forces the model to see more of the image into consideration rather than focusing on certain features. In droupout, units are dropped at the intermediate layers. This means after dropout information of the features could still be present in other features of the feature map. So cutout performs much better ini convolutional networks than dropout because withh fewer parameters we expect them to be more codependent. Cutouot also helps gemerate more data which is new to the network.

In [1]:
```
import numpy as np
```

In [2]:
```
from tensorflow.keras.datasets import cifar10
import tensorflow as tf
```

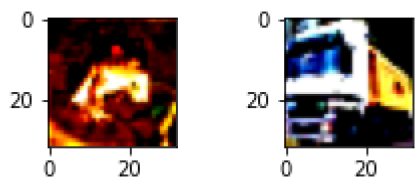
In [3]:
```

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
# Data normalization
m, std = np.mean(x_train), np.std(x_train)
x_train = (x_train - m)/std
x_test = (x_test - m)/std
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
 (https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz)
170500096/170498071 [==============================] - 2s 0us/step
170508288/170498071 [==============================] - 2s 0us/step
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
```

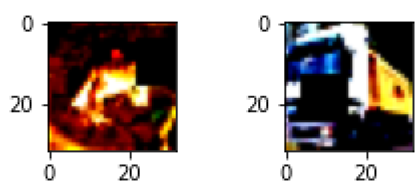
In [4]:
```
def apply_mask(image, size=6, n_squares=1):
    h, w, channels = image.shape
    new_image = image
    for _ in range(n_squares):
        y = np.random.randint(h)
        x = np.random.randint(w)
        y1 = np.clip(y - size // 2, 0, h)
        y2 = np.clip(y + size // 2, 0, h)
        x1 = np.clip(x - size // 2, 0, w)
        x2 = np.clip(x + size // 2, 0, w)
        new_image[y1:y2,x1:x2,:] = 0
    return new_image
```

In [5]:
```
import matplotlib.pyplot as plt
print("Original images:")
for i in range(2):
    plt.subplot(330 + 1 + i)
    plt.imshow(x_train[i])
plt.show()
print("Images with cutout:")
for i in range(2):
    plt.subplot(330 + 1 + i)
    plt.imshow(apply_mask(x_train[i],size=12))
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for flo
ats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for flo
ats or [0..255] for integers).

Original images:
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for flo
ats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for flo
ats or [0..255] for integers).

Images with cutout:
```



In [ ]:
```

```

4.2

In [ ]:
```
from __future__ import print_function
import tensorflow as tf
from tensorflow.keras.layers import Dense, Conv2D, BatchNormalization, Activ
from tensorflow.keras.layers import AveragePooling2D, Input, Flatten
from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateSchedule
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.regularizers import l2
from tensorflow.keras import backend as K
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import cifar10
import numpy as np
import os

```

In [ ]:

```
def resnet_layer(inputs,
                 num_filters=16,
                 kernel_size=3,
                 strides=1,
                 activation='relu',
                 batch_normalization=True,
                 conv_first=True):

    conv = Conv2D(num_filters,
                  kernel_size=kernel_size,
                  strides=strides,
                  padding='same',
                  kernel_initializer='he_normal',
                  kernel_regularizer=l2(1e-4))

    x = inputs
    if conv_first:
        x = conv(x)
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
    else:
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
        x = conv(x)
    return x

def resnet_v1(input_shape, depth, num_classes=10):

    if (depth - 2) % 6 != 0:
        raise ValueError('depth should be 6n+2 (eg 20, 32, 44 in [a])')
    # Start model definition.
    num_filters = 16
    num_res_blocks = int((depth - 2) / 6)

    inputs = Input(shape=input_shape)
    x = resnet_layer(inputs=inputs)
    # Instantiate the stack of residual units
    for stack in range(3):
        for res_block in range(num_res_blocks):
            strides = 1
            if stack > 0 and res_block == 0:  # first layer but not first st
                strides = 2  # downsample
            y = resnet_layer(inputs=x,
                             num_filters=num_filters,
                             strides=strides)
            y = resnet_layer(inputs=y,
                             num_filters=num_filters,
                             activation=None)
            if stack > 0 and res_block == 0:  # first layer but not first st
                # linear projection residual shortcut connection to match
                # changed dims
                x = resnet_layer(inputs=x,
                                 num_filters=num_filters,
                                 kernel_size=1,
                                 strides=strides,
                                 activation=None,
                                 batch_normalization=False)
            x = tf.keras.layers.add([x, y])
            x = Activation('relu')(x)
        num_filters *= 2

    # Add classifier on top.
    # v1 does not use BN after last shortcut connection-ReLU
    x = AveragePooling2D(pool_size=8)(x)
    y = Flatten()(x)
    outputs = Dense(num_classes,
                    activation='softmax',
                    kernel_initializer='he_normal')(y)

```

```
    # Instantiate model.
    model = Model(inputs=inputs, outputs=outputs)
    return model
```

In [ ]:

```
model = resnet_v1(
    input_shape=x_train.shape[1:],
    depth=44
)
model.compile(
    loss='categorical_crossentropy',
    optimizer=tf.optimizers.RMSprop(),
    metrics=['accuracy']
)
def lr_schedule(epoch):
    lr = 1e-3
    if epoch > 85:
        lr *= 0.5e-3
    elif epoch > 75:
        lr *= 1e-3
    elif epoch > 65:
        lr *= 1e-2
    elif epoch > 50:
        lr *= 1e-1
    print('Learning rate: ', lr)
    return lr
lr_scheduler = LearningRateScheduler(lr_schedule)
```

In [ ]:

```
datagen = ImageDataGenerator(
    width_shift_range=[-4,4],
    height_shift_range=[-4,4],
    horizontal_flip=0.5,
    fill_mode='constant',
    cval=0
)
datagen.fit(x_train)
```

In [ ]:
```
import time
start = time.time()
hist = model.fit_generator(
    datagen.flow(x_train, y_train, batch_size=64),
    epochs=100,
    validation_data=(x_test,y_test),
    callbacks=[lr_scheduler]
)
duration = time.time() - start
simple_val_acc = hist.history['val_accuracy']
plt.plot([1 - acc for acc in simple_val_acc])
plt.title('Validation error for a model using simple augmentation')
plt.ylabel('Validation error')
plt.xlabel('Epoch')
plt.savefig('simple_augmentation_error.png')
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/keras/engine/training.py:1972: UserWarni
ng: `Model.fit_generator` is deprecated and will be removed in a future versio
n. Please use `Model.fit`, which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '

Epoch 1/100
Learning rate:  0.001
782/782 [==============================] - 177s 175ms/step - loss: 1.9056 - acc
uracy: 0.4267 - val_loss: 2.6308 - val_accuracy: 0.4039
Epoch 2/100
Learning rate:  0.001
782/782 [==============================] - 126s 161ms/step - loss: 1.4008 - acc
uracy: 0.6061 - val_loss: 1.5889 - val_accuracy: 0.5523
Epoch 3/100
Learning rate:  0.001
782/782 [==============================] - 126s 161ms/step - loss: 1.1735 - acc
uracy: 0.6823 - val_loss: 1.5540 - val_accuracy: 0.6149
Epoch 4/100
Learning rate:  0.001
782/782 [==============================] - 126s 161ms/step - loss: 1.0408 - acc
uracy: 0.7306 - val_loss: 2.2580 - val_accuracy: 0.5336
Epoch 5/100
Learning rate:  0.001
782/782 [==============================] - 126s 162ms/step - loss: 0.9529 - acc
uracy: 0.7553 - val_loss: 1.4390 - val_accuracy: 0.6586
Epoch 6/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.8931 - acc
uracy: 0.7744 - val_loss: 1.5128 - val_accuracy: 0.6593
Epoch 7/100
Learning rate:  0.001
782/782 [==============================] - 126s 161ms/step - loss: 0.8457 - acc
uracy: 0.7906 - val_loss: 1.2601 - val_accuracy: 0.7004
Epoch 8/100
Learning rate:  0.001
782/782 [==============================] - 126s 161ms/step - loss: 0.7994 - acc
uracy: 0.8048 - val_loss: 1.5200 - val_accuracy: 0.6598
Epoch 9/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.7759 - acc
uracy: 0.8123 - val_loss: 1.4099 - val_accuracy: 0.6508
Epoch 10/100
Learning rate:  0.001
782/782 [==============================] - 126s 161ms/step - loss: 0.7425 - acc
uracy: 0.8230 - val_loss: 1.3371 - val_accuracy: 0.6874
Epoch 11/100
Learning rate:  0.001
782/782 [==============================] - 130s 167ms/step - loss: 0.7220 - acc
uracy: 0.8301 - val_loss: 2.0198 - val_accuracy: 0.5947
Epoch 12/100
Learning rate:  0.001
782/782 [==============================] - 153s 195ms/step - loss: 0.7004 - acc
uracy: 0.8384 - val_loss: 1.5218 - val_accuracy: 0.6857
Epoch 13/100
Learning rate:  0.001
782/782 [==============================] - 154s 196ms/step - loss: 0.6778 - acc
uracy: 0.8444 - val_loss: 1.1214 - val_accuracy: 0.7366
```

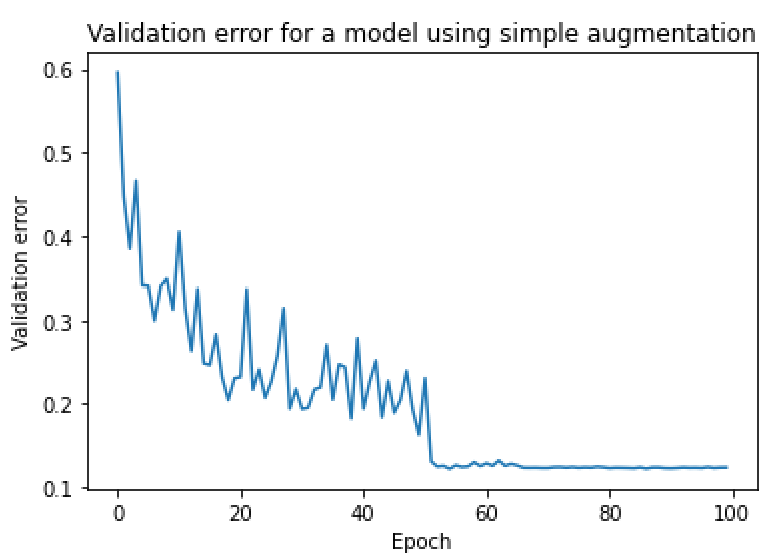
```
Epoch 14/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.6649 - acc
uracy: 0.8505 - val_loss: 1.5750 - val_accuracy: 0.6626
Epoch 15/100
Learning rate:  0.001
782/782 [==============================] - 130s 166ms/step - loss: 0.6481 - acc
uracy: 0.8541 - val_loss: 1.0611 - val_accuracy: 0.7513
Epoch 16/100
Learning rate:  0.001
782/782 [==============================] - 147s 188ms/step - loss: 0.6324 - acc
uracy: 0.8622 - val_loss: 1.1209 - val_accuracy: 0.7535
Epoch 17/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.6179 - acc
uracy: 0.8663 - val_loss: 1.3588 - val_accuracy: 0.7168
Epoch 18/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.6115 - acc
uracy: 0.8665 - val_loss: 1.0334 - val_accuracy: 0.7682
Epoch 19/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.6028 - acc
uracy: 0.8718 - val_loss: 0.9291 - val_accuracy: 0.7954
Epoch 20/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5908 - acc
uracy: 0.8767 - val_loss: 1.0640 - val_accuracy: 0.7694
Epoch 21/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5813 - acc
uracy: 0.8796 - val_loss: 1.0175 - val_accuracy: 0.7678
Epoch 22/100
Learning rate:  0.001
782/782 [==============================] - 130s 167ms/step - loss: 0.5722 - acc
uracy: 0.8814 - val_loss: 2.0869 - val_accuracy: 0.6629
Epoch 23/100
Learning rate:  0.001
782/782 [==============================] - 159s 203ms/step - loss: 0.5674 - acc
uracy: 0.8846 - val_loss: 1.0509 - val_accuracy: 0.7830
Epoch 24/100
Learning rate:  0.001
782/782 [==============================] - 161s 206ms/step - loss: 0.5648 - acc
uracy: 0.8850 - val_loss: 1.1325 - val_accuracy: 0.7588
Epoch 25/100
Learning rate:  0.001
782/782 [==============================] - 178s 227ms/step - loss: 0.5533 - acc
uracy: 0.8887 - val_loss: 1.0281 - val_accuracy: 0.7927
Epoch 26/100
Learning rate:  0.001
782/782 [==============================] - 151s 193ms/step - loss: 0.5444 - acc
uracy: 0.8933 - val_loss: 1.0416 - val_accuracy: 0.7732
Epoch 27/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5409 - acc
uracy: 0.8937 - val_loss: 1.2372 - val_accuracy: 0.7421
Epoch 28/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5308 - acc
uracy: 0.8979 - val_loss: 1.6231 - val_accuracy: 0.6857
Epoch 29/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5316 - acc
uracy: 0.8974 - val_loss: 0.8730 - val_accuracy: 0.8058
Epoch 30/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5234 - acc
uracy: 0.9005 - val_loss: 1.0632 - val_accuracy: 0.7823
Epoch 31/100
Learning rate:  0.001
782/782 [==============================] - 124s 159ms/step - loss: 0.5200 - acc
uracy: 0.9023 - val_loss: 0.9342 - val_accuracy: 0.8061
Epoch 32/100
Learning rate:  0.001
```

```
782/782 [==============================] - 124s 159ms/step - loss: 0.5175 - acc
uracy: 0.9044 - val_loss: 0.9498 - val_accuracy: 0.8045
Epoch 33/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5088 - acc
uracy: 0.9059 - val_loss: 1.1088 - val_accuracy: 0.7827
Epoch 34/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5107 - acc
uracy: 0.9047 - val_loss: 1.0627 - val_accuracy: 0.7801
Epoch 35/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.5031 - acc
uracy: 0.9084 - val_loss: 1.4067 - val_accuracy: 0.7292
Epoch 36/100
Learning rate:  0.001
782/782 [==============================] - 129s 165ms/step - loss: 0.5008 - acc
uracy: 0.9103 - val_loss: 1.0367 - val_accuracy: 0.7949
Epoch 37/100
Learning rate:  0.001
782/782 [==============================] - 146s 187ms/step - loss: 0.4990 - acc
uracy: 0.9106 - val_loss: 1.2352 - val_accuracy: 0.7527
Epoch 38/100
Learning rate:  0.001
782/782 [==============================] - 124s 159ms/step - loss: 0.4926 - acc
uracy: 0.9124 - val_loss: 1.3893 - val_accuracy: 0.7560
Epoch 39/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.4957 - acc
uracy: 0.9131 - val_loss: 0.9529 - val_accuracy: 0.8176
Epoch 40/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.4869 - acc
uracy: 0.9173 - val_loss: 1.7039 - val_accuracy: 0.7213
Epoch 41/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.4888 - acc
uracy: 0.9149 - val_loss: 1.0616 - val_accuracy: 0.8058
Epoch 42/100
Learning rate:  0.001
782/782 [==============================] - 125s 160ms/step - loss: 0.4824 - acc
uracy: 0.9166 - val_loss: 1.2353 - val_accuracy: 0.7735
Epoch 43/100
Learning rate:  0.001
782/782 [==============================] - 131s 167ms/step - loss: 0.4820 - acc
uracy: 0.9169 - val_loss: 1.4462 - val_accuracy: 0.7483
Epoch 44/100
Learning rate:  0.001
782/782 [==============================] - 171s 219ms/step - loss: 0.4767 - acc
uracy: 0.9196 - val_loss: 1.0189 - val_accuracy: 0.8159
Epoch 45/100
Learning rate:  0.001
782/782 [==============================] - 132s 169ms/step - loss: 0.4758 - acc
uracy: 0.9201 - val_loss: 1.2252 - val_accuracy: 0.7726
Epoch 46/100
Learning rate:  0.001
782/782 [==============================] - 165s 211ms/step - loss: 0.4785 - acc
uracy: 0.9193 - val_loss: 0.9986 - val_accuracy: 0.8106
Epoch 47/100
Learning rate:  0.001
782/782 [==============================] - 128s 163ms/step - loss: 0.4739 - acc
uracy: 0.9206 - val_loss: 1.1642 - val_accuracy: 0.7953
Epoch 48/100
Learning rate:  0.001
782/782 [==============================] - 132s 169ms/step - loss: 0.4697 - acc
uracy: 0.9221 - val_loss: 1.2746 - val_accuracy: 0.7605
Epoch 49/100
Learning rate:  0.001
782/782 [==============================] - 148s 189ms/step - loss: 0.4706 - acc
uracy: 0.9213 - val_loss: 1.1097 - val_accuracy: 0.8070
Epoch 50/100
Learning rate:  0.001
782/782 [==============================] - 128s 163ms/step - loss: 0.4680 - acc
uracy: 0.9227 - val_loss: 0.8508 - val_accuracy: 0.8372
```

```
Epoch 51/100
Learning rate:  0.001
782/782 [==============================] - 127s 162ms/step - loss: 0.4635 - acc
uracy: 0.9238 - val_loss: 1.2129 - val_accuracy: 0.7692
Epoch 52/100
Learning rate:  0.0001
782/782 [==============================] - 127s 163ms/step - loss: 0.3641 - acc
uracy: 0.9598 - val_loss: 0.7531 - val_accuracy: 0.8690
Epoch 53/100
Learning rate:  0.0001
782/782 [==============================] - 128s 164ms/step - loss: 0.3298 - acc
uracy: 0.9708 - val_loss: 0.7455 - val_accuracy: 0.8752
Epoch 54/100
Learning rate:  0.0001
782/782 [==============================] - 127s 163ms/step - loss: 0.3089 - acc
uracy: 0.9764 - val_loss: 0.7677 - val_accuracy: 0.8743
Epoch 55/100
Learning rate:  0.0001
782/782 [==============================] - 128s 164ms/step - loss: 0.2946 - acc
uracy: 0.9797 - val_loss: 0.7850 - val_accuracy: 0.8779
Epoch 56/100
Learning rate:  0.0001
782/782 [==============================] - 127s 162ms/step - loss: 0.2832 - acc
uracy: 0.9826 - val_loss: 0.8229 - val_accuracy: 0.8736
Epoch 57/100
Learning rate:  0.0001
782/782 [==============================] - 127s 163ms/step - loss: 0.2724 - acc
uracy: 0.9845 - val_loss: 0.8225 - val_accuracy: 0.8757
Epoch 58/100
Learning rate:  0.0001
782/782 [==============================] - 128s 163ms/step - loss: 0.2642 - acc
uracy: 0.9862 - val_loss: 0.8509 - val_accuracy: 0.8750
Epoch 59/100
Learning rate:  0.0001
782/782 [==============================] - 134s 171ms/step - loss: 0.2541 - acc
uracy: 0.9887 - val_loss: 0.9483 - val_accuracy: 0.8700
Epoch 60/100
Learning rate:  0.0001
782/782 [==============================] - 165s 211ms/step - loss: 0.2498 - acc
uracy: 0.9884 - val_loss: 0.9167 - val_accuracy: 0.8746
Epoch 61/100
Learning rate:  0.0001
782/782 [==============================] - 128s 163ms/step - loss: 0.2432 - acc
uracy: 0.9896 - val_loss: 0.9226 - val_accuracy: 0.8712
Epoch 62/100
Learning rate:  0.0001
782/782 [==============================] - 128s 163ms/step - loss: 0.2361 - acc
uracy: 0.9913 - val_loss: 0.9261 - val_accuracy: 0.8742
Epoch 63/100
Learning rate:  0.0001
782/782 [==============================] - 127s 163ms/step - loss: 0.2312 - acc
uracy: 0.9914 - val_loss: 1.0086 - val_accuracy: 0.8679
Epoch 64/100
Learning rate:  0.0001
782/782 [==============================] - 128s 163ms/step - loss: 0.2256 - acc
uracy: 0.9921 - val_loss: 0.9471 - val_accuracy: 0.8743
Epoch 65/100
Learning rate:  0.0001
782/782 [==============================] - 127s 163ms/step - loss: 0.2222 - acc
uracy: 0.9927 - val_loss: 1.0164 - val_accuracy: 0.8719
Epoch 66/100
Learning rate:  0.0001
782/782 [==============================] - 128s 163ms/step - loss: 0.2180 - acc
uracy: 0.9928 - val_loss: 1.0223 - val_accuracy: 0.8738
Epoch 67/100
Learning rate:  1e-05
782/782 [==============================] - 127s 163ms/step - loss: 0.2115 - acc
uracy: 0.9953 - val_loss: 0.9928 - val_accuracy: 0.8765
Epoch 68/100
Learning rate:  1e-05
782/782 [==============================] - 133s 170ms/step - loss: 0.2101 - acc
uracy: 0.9953 - val_loss: 0.9815 - val_accuracy: 0.8766
Epoch 69/100
Learning rate:  1e-05
```

```
782/782 [==============================] - 155s 198ms/step - loss: 0.2084 - acc
uracy: 0.9957 - val_loss: 0.9911 - val_accuracy: 0.8765
Epoch 70/100
Learning rate:  1e-05
782/782 [==============================] - 133s 170ms/step - loss: 0.2060 - acc
uracy: 0.9967 - val_loss: 0.9879 - val_accuracy: 0.8768
Epoch 71/100
Learning rate:  1e-05
782/782 [==============================] - 153s 195ms/step - loss: 0.2056 - acc
uracy: 0.9962 - val_loss: 0.9945 - val_accuracy: 0.8768
Epoch 72/100
Learning rate:  1e-05
782/782 [==============================] - 128s 163ms/step - loss: 0.2048 - acc
uracy: 0.9966 - val_loss: 0.9972 - val_accuracy: 0.8759
Epoch 73/100
Learning rate:  1e-05
782/782 [==============================] - 128s 163ms/step - loss: 0.2044 - acc
uracy: 0.9966 - val_loss: 1.0065 - val_accuracy: 0.8758
Epoch 74/100
Learning rate:  1e-05
782/782 [==============================] - 128s 163ms/step - loss: 0.2032 - acc
uracy: 0.9970 - val_loss: 1.0024 - val_accuracy: 0.8764
Epoch 75/100
Learning rate:  1e-05
782/782 [==============================] - 127s 163ms/step - loss: 0.2024 - acc
uracy: 0.9968 - val_loss: 1.0089 - val_accuracy: 0.8758
Epoch 76/100
Learning rate:  1e-05
782/782 [==============================] - 128s 163ms/step - loss: 0.2013 - acc
uracy: 0.9973 - val_loss: 1.0213 - val_accuracy: 0.8766
Epoch 77/100
Learning rate:  1e-06
782/782 [==============================] - 128s 164ms/step - loss: 0.2008 - acc
uracy: 0.9973 - val_loss: 1.0176 - val_accuracy: 0.8761
Epoch 78/100
Learning rate:  1e-06
782/782 [==============================] - 128s 163ms/step - loss: 0.2008 - acc
uracy: 0.9970 - val_loss: 1.0177 - val_accuracy: 0.8764
Epoch 79/100
Learning rate:  1e-06
782/782 [==============================] - 127s 163ms/step - loss: 0.2009 - acc
uracy: 0.9973 - val_loss: 1.0227 - val_accuracy: 0.8754
Epoch 80/100
Learning rate:  1e-06
782/782 [==============================] - 127s 163ms/step - loss: 0.2002 - acc
uracy: 0.9972 - val_loss: 1.0203 - val_accuracy: 0.8760
Epoch 81/100
Learning rate:  1e-06
782/782 [==============================] - 128s 164ms/step - loss: 0.2009 - acc
uracy: 0.9971 - val_loss: 1.0215 - val_accuracy: 0.8770
Epoch 82/100
Learning rate:  1e-06
782/782 [==============================] - 132s 169ms/step - loss: 0.2008 - acc
uracy: 0.9974 - val_loss: 1.0162 - val_accuracy: 0.8765
Epoch 83/100
Learning rate:  1e-06
782/782 [==============================] - 158s 202ms/step - loss: 0.2000 - acc
uracy: 0.9974 - val_loss: 1.0177 - val_accuracy: 0.8766
Epoch 84/100
Learning rate:  1e-06
782/782 [==============================] - 127s 163ms/step - loss: 0.1995 - acc
uracy: 0.9976 - val_loss: 1.0188 - val_accuracy: 0.8768
Epoch 85/100
Learning rate:  1e-06
782/782 [==============================] - 128s 164ms/step - loss: 0.2002 - acc
uracy: 0.9972 - val_loss: 1.0194 - val_accuracy: 0.8771
Epoch 86/100
Learning rate:  1e-06
782/782 [==============================] - 127s 163ms/step - loss: 0.1993 - acc
uracy: 0.9976 - val_loss: 1.0217 - val_accuracy: 0.8761
Epoch 87/100
Learning rate:  5e-07
782/782 [==============================] - 127s 163ms/step - loss: 0.2001 - acc
uracy: 0.9974 - val_loss: 1.0191 - val_accuracy: 0.8776
```

```
Epoch 88/100
Learning rate:  5e-07
782/782 [==============================] - 132s 169ms/step - loss: 0.1998 - acc
uracy: 0.9973 - val_loss: 1.0158 - val_accuracy: 0.8762
Epoch 89/100
Learning rate:  5e-07
782/782 [==============================] - 160s 204ms/step - loss: 0.1995 - acc
uracy: 0.9974 - val_loss: 1.0174 - val_accuracy: 0.8762
Epoch 90/100
Learning rate:  5e-07
782/782 [==============================] - 129s 165ms/step - loss: 0.1997 - acc
uracy: 0.9973 - val_loss: 1.0204 - val_accuracy: 0.8770
Epoch 91/100
Learning rate:  5e-07
782/782 [==============================] - 128s 163ms/step - loss: 0.2000 - acc
uracy: 0.9975 - val_loss: 1.0190 - val_accuracy: 0.8772
Epoch 92/100
Learning rate:  5e-07
782/782 [==============================] - 128s 164ms/step - loss: 0.1996 - acc
uracy: 0.9973 - val_loss: 1.0183 - val_accuracy: 0.8768
Epoch 93/100
Learning rate:  5e-07
782/782 [==============================] - 128s 163ms/step - loss: 0.2000 - acc
uracy: 0.9970 - val_loss: 1.0210 - val_accuracy: 0.8762
Epoch 94/100
Learning rate:  5e-07
782/782 [==============================] - 128s 163ms/step - loss: 0.1998 - acc
uracy: 0.9971 - val_loss: 1.0204 - val_accuracy: 0.8766
Epoch 95/100
Learning rate:  5e-07
782/782 [==============================] - 128s 163ms/step - loss: 0.1992 - acc
uracy: 0.9976 - val_loss: 1.0278 - val_accuracy: 0.8765
Epoch 96/100
Learning rate:  5e-07
782/782 [==============================] - 133s 170ms/step - loss: 0.1990 - acc
uracy: 0.9978 - val_loss: 1.0210 - val_accuracy: 0.8768
Epoch 97/100
Learning rate:  5e-07
782/782 [==============================] - 168s 215ms/step - loss: 0.1991 - acc
uracy: 0.9977 - val_loss: 1.0263 - val_accuracy: 0.8758
Epoch 98/100
Learning rate:  5e-07
782/782 [==============================] - 127s 162ms/step - loss: 0.1982 - acc
uracy: 0.9979 - val_loss: 1.0196 - val_accuracy: 0.8768
Epoch 99/100
Learning rate:  5e-07
782/782 [==============================] - 127s 163ms/step - loss: 0.1989 - acc
uracy: 0.9974 - val_loss: 1.0185 - val_accuracy: 0.8763
Epoch 100/100
Learning rate:  5e-07
782/782 [==============================] - 127s 163ms/step - loss: 0.1989 - acc
uracy: 0.9976 - val_loss: 1.0263 - val_accuracy: 0.8762
```



In [ ]: 1

In [ ]:

```

```

In [ ]:

```
import time
def batch_generator(x, y, epochs, m, batch_size, augment=None):
    for _ in range(epochs):
        n = x.shape[0]
        reorder = np.random.permutation(n)
        cursor = 0
        while cursor + batch_size < x.shape[0]:
            x_batch = x[reorder[cursor:cursor+batch_size]]
            y_batch = y[reorder[cursor:cursor+batch_size]]
            if augment != None:
                yield np.array([augment(xx) for xx in x_batch for rep in ran
            else:
                yield x_batch, y_batch
            cursor += batch_size
val_acc_cutout = []
epochs = 100
durations = []
for i in [2,4]:
    model = resnet_v1(
        input_shape=x_train.shape[1:],
        depth=44
    )
    model.compile(
        loss='categorical_crossentropy',
        optimizer=tf.optimizers.RMSprop(),
        metrics=['accuracy']
    )
    duration = time.time()
    hist = model.fit_generator(
        batch_generator(
            x_train,
            y_train,
            m=i,
            batch_size=64,
            epochs=100,
            augment=apply_mask
        ),
        epochs=100,
        validation_data=(x_test,y_test),
        steps_per_epoch=np.floor(x_train.shape[0]/64.0),
        callbacks=[lr_scheduler]
    )
    durations.append(time.time()-duration)
    val_acc_cutout.append(hist.history['val_accuracy'])
    print(len(hist.history['val_accuracy']))
```

```
Epoch 53/100
Learning rate:  0.0001
781/781 [==============================] - 143s 183ms/step - loss: 0.3150 - accuracy: 0.9867 - val_loss: 0.7894 - val_accuracy: 0.8670
Epoch 54/100
Learning rate:  0.0001
781/781 [==============================] - 143s 183ms/step - loss: 0.3016 - accuracy: 0.9882 - val_loss: 0.7924 - val_accuracy: 0.8685
Epoch 55/100
Learning rate:  0.0001
781/781 [==============================] - 143s 183ms/step - loss: 0.2880 - accuracy: 0.9915 - val_loss: 0.8286 - val_accuracy: 0.8685
Epoch 56/100
Learning rate:  0.0001
781/781 [==============================] - 143s 183ms/step - loss: 0.2805 - accuracy: 0.9919 - val_loss: 0.8261 - val_accuracy: 0.8666
Epoch 57/100
Learning rate:  0.0001

781/781 [==============================] - 142s 182ms/step - loss: 0.2707 - accuracy: 0.9935 - val_loss: 0.8449 - val_accuracy: 0.8667
```

In [ ]:

```
import time
def batch_generator(x, y, epochs, m, batch_size, augment=None):
    for _ in range(epochs):
        n = x.shape[0]
        reorder = np.random.permutation(n)
        cursor = 0
        while cursor + batch_size < x.shape[0]:
            x_batch = x[reorder[cursor:cursor+batch_size]]
            y_batch = y[reorder[cursor:cursor+batch_size]]
            if augment != None:
                yield np.array([augment(xx) for xx in x_batch for rep in ran
            else:
                yield x_batch, y_batch
            cursor += batch_size
val_acc_cutout = []
epochs = 100
durations = []
for i in [4]:
    model = resnet_v1(
        input_shape=x_train.shape[1:],
        depth=44
    )
    model.compile(
        loss='categorical_crossentropy',
        optimizer=tf.optimizers.RMSprop(),
        metrics=['accuracy']
    )
    duration = time.time()
    hist = model.fit_generator(
        batch_generator(
            x_train,
            y_train,
            m=i,
            batch_size=64,
            epochs=100,
            augment=apply_mask
        ),
        epochs=100,
        validation_data=(x_test,y_test),
        steps_per_epoch=np.floor(x_train.shape[0]/64.0),
        callbacks=[lr_scheduler]
    )
    durations.append(time.time()-duration)
    val_acc_cutout.append(hist.history['val_accuracy'])
    print(len(hist.history['val_accuracy']))
```

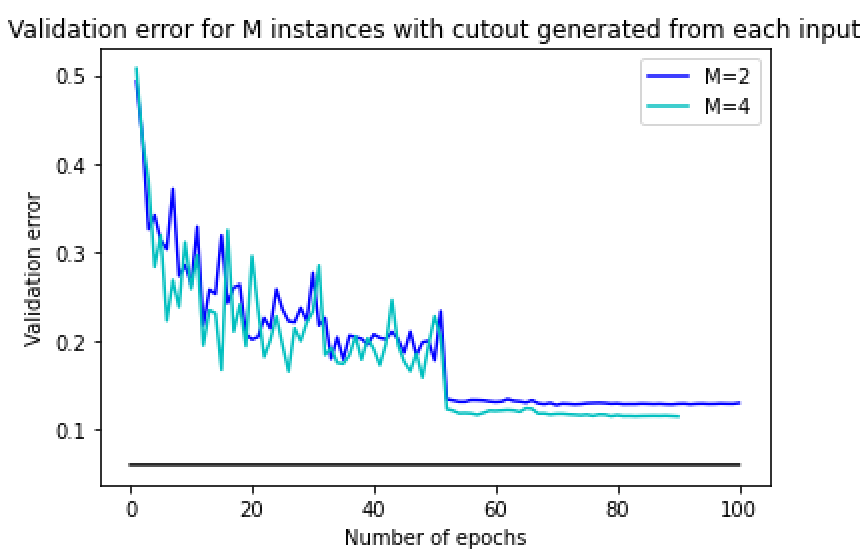
```
Learning rate:  1e-05
781/781 [==============================] - 246s 315ms/step - loss: 0.2414 -
accuracy: 0.9887 - val_loss: 0.7291 - val_accuracy: 0.8818
Epoch 69/100
Learning rate:  1e-05
781/781 [==============================] - 252s 322ms/step - loss: 0.2410 -
accuracy: 0.9896 - val_loss: 0.7313 - val_accuracy: 0.8831
Epoch 70/100
Learning rate:  1e-05
781/781 [==============================] - 253s 324ms/step - loss: 0.2416 -
accuracy: 0.9888 - val_loss: 0.7283 - val_accuracy: 0.8823
Epoch 71/100
Learning rate:  1e-05
781/781 [==============================] - 253s 324ms/step - loss: 0.2367 -
accuracy: 0.9907 - val_loss: 0.7310 - val_accuracy: 0.8822
Epoch 72/100
Learning rate:  1e-05
781/781 [==============================] - 247s 317ms/step - loss: 0.2348 -
accuracy: 0.9911 - val_loss: 0.7286 - val_accuracy: 0.8829
Epoch 73/100
Learning rate:  1e-05
```

In [ ]:

```
def opp(l):
    return [1-el for el in l]
cutout2_data, cutout4_data, cutout8_data, cutout16_data,
  cutout32_data = val_acc_cutout
plt.plot(range(1,101),opp(simple_val_acc),"y-")
plt.plot(range(1,101),opp(cutout2_data),"b-")
plt.plot(range(1,101),opp(cutout4_data),"c-")
plt.plot(range(1,101),opp(cutout8_data),"g-")
plt.plot(range(1,101),opp(cutout16_data),"r-")
plt.plot(range(1,101),opp(cutout32_data),"m-")
plt.legend(["M=0","M=2","M=4","M=8","M=16","M=32"])
plt.plot(np.linspace(0,100,10000),[0.06]*10000,"k-")
plt.title("Validation error for M instances with cutout generated from each
plt.xlabel("Number of epochs")
plt.ylabel("Validation error")
plt.savefig("acc_cutout.png")
plt.show()
```

In [8]:

```
def opp(l):
    return [1-el for el in l]
cutout2_data, cutout4_data = val_acc_cutout
# plt.plot(range(1,101),opp(simple_val_acc),"y-")
plt.plot(range(1,101),opp(cutout2_data),"b-")
plt.plot(range(1,len(cutout4_data)+1),opp(cutout4_data),"c-")
plt.legend(["M=2","M=4"])
plt.plot(np.linspace(0,100,10000),[0.06]*10000,"k-")
plt.title("Validation error for M instances with cutout generated from each
plt.xlabel("Number of epochs")
plt.ylabel("Validation error")
plt.savefig("acc_cutout.png")
plt.show()
```



In [2]:

```
print(durations)
```

...

M=2 41 epochs wall-clock time is approximately 97 minutes

M=4 52 epochs wall-clock time is approximately 212 minutes