In [1]:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import keras
from keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras import regularizers
from tensorflow.keras.optimizers import Optimizer
import matplotlib.pyplot as plt
import time
import keras.backend as K
import keras

# import the necessary packages
from pyimagesearch.minigooglenet import MiniGoogLeNet
from pyimagesearch.clr_callback import CyclicLR
from pyimagesearch import config
from pyimagesearch import learningratefinder
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from tensorflow.keras.layers import AveragePooling2D, Input
from tensorflow.keras.regularizers import l2
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.layers import Dense, Conv2D
from tensorflow.keras.layers import BatchNormalization, Activation
from tensorflow.keras.layers import AveragePooling2D, Input
from tensorflow.keras.layers import Flatten, add
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ModelCheckpoint, LearningRateScheduler
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.regularizers import l2
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import plot_model
from tensorflow.keras.utils import to_categorical
# from tensorflow.keras.datasets import cifar10
import matplotlib.pyplot as plt
import numpy as np
import cv2
import sys
```

Problem 1

question 1

- Write the weight update equations for the five adaptive learning rate methods. Explain each term clearly. What are the hyper-parameters in each policy ? Explain how AdaDelta and Adam are different from RMSProp.

Adagrad

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{S_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

$$S_t = S_{t-1} + \left[\frac{\partial L}{\partial w_t}\right]^2$$

- Here the $\frac{\partial L}{\partial w_t}$, is the gradient for each update step, control this value by multiplying it with $\frac{1}{\sqrt{S_t+\epsilon}}$, will let actural learning rate become smaller and smaller, thus good for final covergence. So adagrad is a optimizer that decaying learning rate by S_t
- hyper-parameter:
  - **base learning rate**: α.
  - ϵ: none-zero denominator.

RMSProp

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{S_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

$$S_t = \beta S_{t-1} + (1-\beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

- RMSProp is very similar to Adagrad, they both controling learning rate by decaying it with multiplying it by $\frac{1}{\sqrt{S_t+\epsilon}}$, the difference is that RMSprop have additional hyper-parameter β controling the decaying speed, if we let β = 1, the learning rate will remain steady and don't change during learning.
- hyper-parameter:
  - **base learning rate** : α
  - **decaying speed** : β
  - ϵ: none-zero denominator.

RMSProp + Nesterov

Nesterov

$w_{t+1} = w_t - \frac{\alpha}{\sqrt{S_t+\epsilon}} V_t$
$V_t = \beta_v V_{t_1} + (1-\beta_v)\frac{\partial L}{\partial w^*}$
$S_t = \beta_d S_{t-1} + (1-\beta_d)[\frac{\partial L}{\partial w_t}]^2$
$w^* = w_t - \alpha V_{t-1}$

- Very similar to RMSprop, adding Nesterov just introduces momentum and predicting next position instead of directly using velosity.
- Here we have to hyper-parameters combine **RMSprop** with **Nesterov**:
  - β_v is a hyper-parameter controlling the gradient using momentum mechanism, if β_v = 0, previous velocity will not affect following learning gradient.
  - β_d is a hyper-parameter controlling the decaying in RMSprop, if we let β_d = 1, the learning rate will remain steady and don't change during learning.
  - ϵ: none-zero denominator.

Adadelta

Adadelta

$$w_{t+1} = w_t - \frac{\sqrt{D_{t-1} + \epsilon}}{\sqrt{S_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

$$D_t = \beta D_{t-1} + (1 - \beta)[\Delta w_t]^2$$

$$S_t = \beta S_{t-1} + (1 - \beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

- As shown above, Adadelta has one hyper-parameters.
- In the denominator, it's the same learning rate decay as **RMSprop** and **Adagrad**, it controls the decaying speed of the learning rate.
- The nominator is different, Since Adagrad will finally reduce the learning rate to 0 as iteration increases, the nominator is actually an improvement of this weakness, so instead of traditional learning rate α, here Adadelta uses the accumulation of changes of weight as the learning rate. So in the early stage, the gradient is huge, so the change of weights is huge, thus we get a larger D_t, and finally, when the network is close to convergence, the change of weights is small, thus we get a smaller D_t.

Adam

Adam

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\hat{S}_t + \epsilon}} \cdot \hat{V}_t$$

$$\hat{V}_t = \frac{V_t}{1 - \beta_1^t}$$

$$\hat{S}_t = \frac{S_t}{1 - \beta_2^t}$$

$$V_t = \beta_1 V_{t-1} + (1 - \beta_1)\frac{\partial L}{\partial w_t}$$

$$S_t = \beta_2 S_{t-1} + (1 - \beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$

- Adam combine the decaying mechanism in RMSprop and momentum.
- Adam have three hyper-paramters:
  - α: base learning rate
  - β_1: control the momentum. If $beta_1$ = 0, then previous velocity will not influence following gradients.
  - β_2: control the weight decay speed. If β_2 = 0, the weight decay will not accumulate.

The difference between AdaDelta and Adam

- Instead of using Δw_t to control the learning rate, Adam uses momentum instead, and both Adadelta and Adam are using S_t to control weight's scale.

question 2

- Train the neural network using all the five methods with L2-regularization for 200 epochs each and plot the training loss vs number of epochs. Which method performs best (lowest training loss) ?

In [2]:
```
#load data
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)
```

In [3]:
```
def make_an_optimizer(optimizer_type):
    if optimizer_type == 'Adagrad':
        return tf.keras.optimizers.Adagrad()
    elif optimizer_type == 'Adadelta':
        return tf.keras.optimizers.Adadelta()
    elif optimizer_type == 'RMSprop':
        return tf.keras.optimizers.RMSprop()
    elif optimizer_type == 'RMSprop_nesterov':
        return tf.keras.optimizers.Nadam()
    elif optimizer_type == 'Adam':
        return tf.keras.optimizers.Adam()

def train_a_neural_network(regularization_type, optimizer_type, epoch, batch_size):
    # define model
    if regularization_type == "L2_regularization":
        model = keras.models.Sequential([
            Flatten( input_shape = (32,32,3)),
            Dense(1000, activation = 'relu', kernel_regularizer='l2', kernel_initializer='
            Dense(1000, activation = 'relu', kernel_regularizer='l2', kernel_initializer='
            Dense(10, activation = 'softmax')
        ])
    if regularization_type == "drop_out":
        model = keras.models.Sequential([
            Flatten( input_shape = (32,32,3)),
            Dropout(0.2),
            Dense(1000, activation = 'relu', kernel_initializer='HeNormal'),
            Dropout(0.5),
            Dense(1000, activation = 'relu', kernel_initializer='HeNormal'),
            Dropout(0.5),
            Dense(10, activation = 'softmax')
        ])
    optimizer = make_an_optimizer(optimizer_type)
    model.compile(loss = 'categorical_crossentropy', optimizer = optimizer, metrics = ['ac
    time_start = time.time()
    history = model.fit(x_train, y_train, epochs = epoch, batch_size = batch_size,
                        validation_data=(x_test, y_test), verbose = 0)
    time_end = time.time()
    return model, history, time_end - time_start
optimizer_lost = ['Adagrad', 'Adadelta', 'RMSprop', 'RMSprop_nesterov', 'Adam']
```

In [4]:
```
model_2_1, history_2_1, time_2_1 = train_a_neural_network('L2_regularization', optimizer_l
```

In [5]:
```
model_2_2, history_2_2, time_2_2 = train_a_neural_network('L2_regularization', optimizer_l
```

In [6]:
```
model_2_3, history_2_3, time_2_3 = train_a_neural_network('L2_regularization', optimizer_l
```

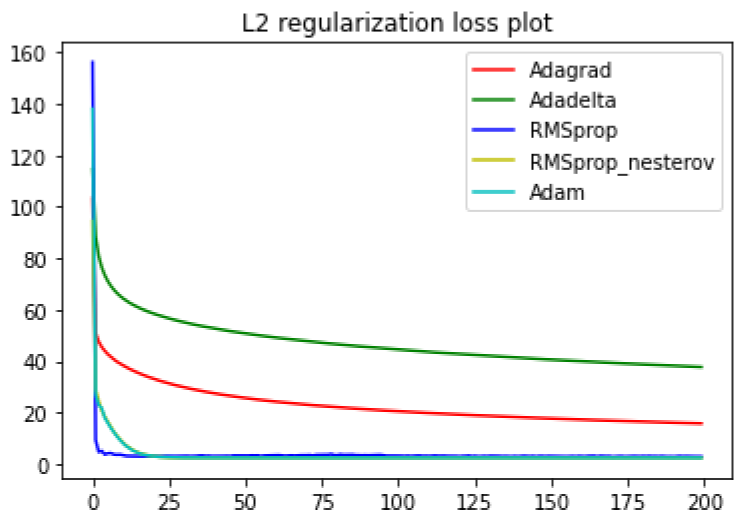
In [7]:
```
model_2_4, history_2_4, time_2_4 = train_a_neural_network('L2_regularization', optimizer_l
```

In [8]:

```
model_2_5, history_2_5, time_2_5 = train_a_neural_network('L2_regularization', optimizer_l
```

In [10]:

```
#LOSS PLOT
epoch_vis = list(range(len(history_2_1.history['loss'])))
plt.plot(epoch_vis,history_2_1.history['loss'], label = optimizer_lost[0], c = 'r')
plt.plot(epoch_vis,history_2_2.history['loss'], label = optimizer_lost[1], c = 'g')
plt.plot(epoch_vis,history_2_3.history['loss'], label = optimizer_lost[2], c = 'b')
plt.plot(epoch_vis,history_2_4.history['loss'], label = optimizer_lost[3], c = 'y')
plt.plot(epoch_vis,history_2_5.history['loss'], label = optimizer_lost[4], c = 'c')
plt.legend()
plt.title('L2 regularization loss plot')
plt.show()
```



In [21]:

```
(history_2_1.history['loss'][-1],
history_2_2.history['loss'][-1],
history_2_3.history['loss'][-1],
history_2_4.history['loss'][-1],
history_2_5.history['loss'][-1])
```

Out[21]:

```
(15.68968391418457,
 37.67438888549805,
 2.7793161869049072,
 2.302692413330078,
 2.302698850631714)
```

- Compared with other optimizers, RMSprop, Adam and RMSprop+nesterov have the lowest loss. Adagrad and Adadelta have a higher loss.
- RMSprop_nesterov's loss is 2.302692413330078 is the lowest loss.

question 3

- Add dropout (probability 0.2 for input layer and 0.5 for hidden layers) and train the neural network again using all the five methods for 200 epochs. Compare the training loss with that in part 2. Which method performs the best ? For the five methods, compare their training time (to finish 200 epochs with dropout) to the training time in part 2 (to finish 200 epochs without dropout).

In [11]:

```
model_3_1, history_3_1, time_3_1 = train_a_neural_network('drop_out', optimizer_lost[0], 2
```

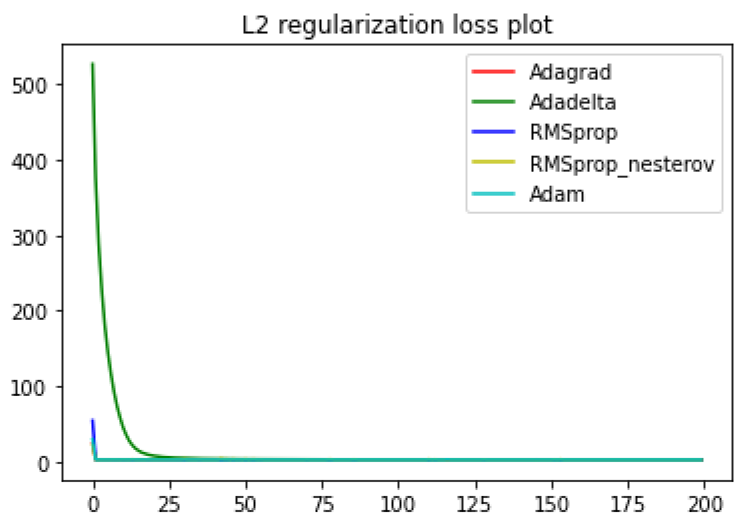
In [12]:
```
model_3_2, history_3_2, time_3_2 = train_a_neural_network('drop_out', optimizer_lost[1], 2
```

In [13]:
```
model_3_3, history_3_3, time_3_3 = train_a_neural_network('drop_out', optimizer_lost[2], 2
```

In [14]:
```
model_3_4, history_3_4, time_3_4 = train_a_neural_network('drop_out', optimizer_lost[3], 2
```

In [15]:
```
model_3_5, history_3_5, time_3_5 = train_a_neural_network('drop_out', optimizer_lost[4], 2
```

In [16]:
```
#LOSS PLOT
epoch_vis = list(range(len(history_2_1.history['loss'])))
plt.plot(epoch_vis,history_3_1.history['loss'], label = optimizer_lost[0], c = 'r')
plt.plot(epoch_vis,history_3_2.history['loss'], label = optimizer_lost[1], c = 'g')
plt.plot(epoch_vis,history_3_3.history['loss'], label = optimizer_lost[2], c = 'b')
plt.plot(epoch_vis,history_3_4.history['loss'], label = optimizer_lost[3], c = 'y')
plt.plot(epoch_vis,history_3_5.history['loss'], label = optimizer_lost[4], c = 'c')
plt.legend()
plt.title('L2 regularization loss plot')
plt.show()
```



In [20]:
```
(history_3_1.history['loss'][-1],
history_3_2.history['loss'][-1],
history_3_3.history['loss'][-1],
history_3_4.history['loss'][-1],
history_3_5.history['loss'][-1])
```

Out[20]:
```
(2.3035354614257812,
 2.4348690509796143,
 2.3067286014556885,
 2.302696943283081,
 2.3027024269104004)
```

In [22]:
```
# time difference between part 2 and part 3
(time_2_1 - time_3_1, time_2_2 - time_3_2, time_2_3 - time_3_3, time_2_4 - time_3_4, time_
```

Out[22]:
```
(22.345446825027466,
 26.96778964996338,
 23.584877967834473,
 24.263986825942993,
 26.715335607528687)
```

- After adding dropout, all 5 optimizers decrease loss to similar level, and the oen with the lowest loss is RMSprop with nesterov.
- After adding dropout, the training is much faster, around 25 seconds faster than L2 regularization.

question 4

In [32]:
```
print('test accuracy for 10 models: ')
(history_2_1.history['val_accuracy'][-1],
history_2_2.history['val_accuracy'][-1],
history_2_3.history['val_accuracy'][-1],
history_2_4.history['val_accuracy'][-1],
history_2_5.history['val_accuracy'][-1],
history_3_1.history['val_accuracy'][-1],
history_3_2.history['val_accuracy'][-1],
history_3_3.history['val_accuracy'][-1],
history_3_4.history['val_accuracy'][-1],
history_3_5.history['val_accuracy'][-1])
```

Out[32]:
```
test accuracy for 10 models:
(0.4440000057220459,
 0.37059998512268066,
 0.20630000531673431,
 0.10000000149011612,
 0.10000000149011612,
 0.09989999979734421,
 0.10019999742507935,
 0.10000000149011612,
 0.10000000149011612,
 0.10000000149011612)
```

- According to the result, adding dropout can let loss decrease loss at start, but can't let model converge. All models with dropout finally have test accuracy around 0.1(random guess). (maybe need to fine tune the learning rate for dropout for a better performance).
- The best model is L2 regularization with Adagrad optimizer, which has test accuracy around 44%.

problem 2

question 1

- Fix batch size to 64 and start with 10 candidate learning rates between 10^{-9} and 10^{1} and train your model for 5 epochs. Plot the training loss as a function of learning rate. You should see a curve like Figure 3 in reference below. From that figure identify the values of lr_{min} and lr_{max}.

In [2]:
```
# prepare data
img_witdth, img_height = 32, 32
((trainX, trainY), (testX, testY)) = fashion_mnist.load_data()

trainX = trainX.astype("float")
testX = testX.astype("float")
# apply mean subtraction to the data
mean = np.mean(trainX, axis=0)
trainX -= mean
testX -= mean

# Fashion MNIST images are 28x28 but the network we will be training
# is expecting 32x32 images
trainX = np.array([cv2.resize(x, (img_witdth, img_height)) for x in trainX])
```

```python
testX = np.array([cv2.resize(x, (img_witdth, img_height)) for x in testX])

# scale the pixel intensities to the range [0, 1]
trainX = trainX.astype("float") / 255.0
testX = testX.astype("float") / 255.0


# reshape the data matrices to include a channel dimension (required
# for training)

trainX = trainX.reshape((trainX.shape[0], img_witdth, img_height, 1))
testX = testX.reshape((testX.shape[0], img_witdth, img_height, 1))


# convert the labels from integers to vectors
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)

# construct the image generator for data augmentation
aug = ImageDataGenerator(
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    fill_mode="nearest",
)
```

In [49]:

```python
min_lr = 1e-9
max_lr = 10
opt = SGD(min_lr, momentum=0.9)
model = MiniGoogLeNet.build(width=img_witdth, height=img_height, depth=1, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
clr = CyclicLR(
    mode=config.CLR_METHOD,
    base_lr=min_lr,
    max_lr=max_lr,
    step_size=config.STEP_SIZE * (trainX.shape[0] // config.BATCH_SIZE),
)
H = model.fit(
    x=aug.flow(trainX, trainY, batch_size=config.BATCH_SIZE),
    validation_data=(testX, testY),
    steps_per_epoch=trainX.shape[0] // config.BATCH_SIZE,
    epochs=config.NUM_EPOCHS,
    callbacks=[clr],
    verbose=1,
)
```

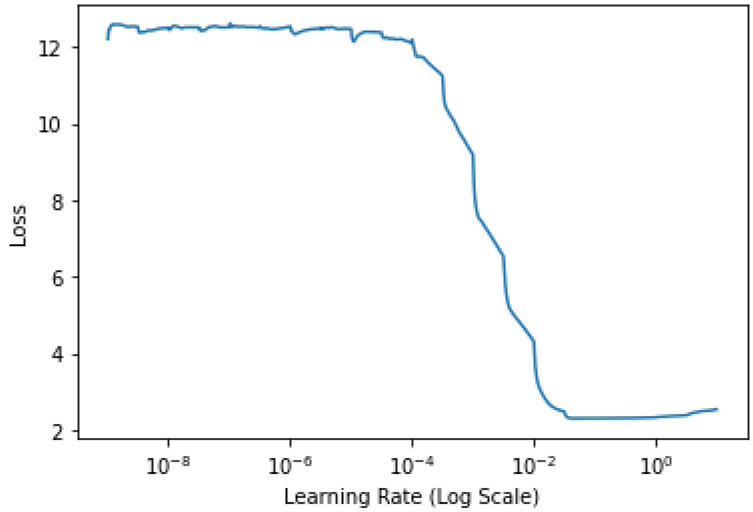
```
Epoch 1/5
  5/937 [..............................] - ETA: 24s - loss: 2.7100 - accuracy: 0.0844 WARN
ING:tensorflow:Callback method `on_train_batch_end` is slow compared to the batch time (ba
tch time: 0.0139s vs `on_train_batch_end` time: 0.0396s). Check your callbacks.
937/937 [==============================] - 26s 26ms/step - loss: 0.7838 - accuracy: 0.7200
- val_loss: 4.4435 - val_accuracy: 0.6254
Epoch 2/5
937/937 [==============================] - 25s 26ms/step - loss: 0.7418 - accuracy: 0.7511
- val_loss: 8.0529 - val_accuracy: 0.5188
Epoch 3/5
937/937 [==============================] - 25s 26ms/step - loss: 0.8490 - accuracy: 0.7135
- val_loss: 12.0776 - val_accuracy: 0.4553
Epoch 4/5
937/937 [==============================] - 25s 26ms/step - loss: 1.8450 - accuracy: 0.4219
- val_loss: 19.7525 - val_accuracy: 0.0990
Epoch 5/5
```

```
937/937 [==============================] - 24s 26ms/step - loss: 2.5047 - accuracy: 0.1058
- val_loss: 3.0525 - val_accuracy: 0.1000
```

In [61]:

```
lrf = learningratefinder.LearningRateFinder(model)
lrf.find(
    aug.flow(trainX, trainY, batch_size=config.BATCH_SIZE),
    min_lr,
    max_lr,
    stepsPerEpoch=np.ceil((len(trainX) / float(config.BATCH_SIZE))),
    batchSize=config.BATCH_SIZE,
epochs = 20)
lrf.plot_loss()
```

```
Epoch 1/20
938/938 [==============================] - 23s 24ms/step - loss: 12.5062 - accuracy: 0.100
0
Epoch 2/20
938/938 [==============================] - 23s 24ms/step - loss: 12.5069 - accuracy: 0.100
0
Epoch 3/20
938/938 [==============================] - 23s 24ms/step - loss: 12.5069 - accuracy: 0.100
0
Epoch 4/20
938/938 [==============================] - 23s 24ms/step - loss: 12.5067 - accuracy: 0.100
0
Epoch 5/20
938/938 [==============================] - 23s 24ms/step - loss: 12.5058 - accuracy: 0.100
0
Epoch 6/20
938/938 [==============================] - 23s 24ms/step - loss: 12.5030 - accuracy: 0.100
0
Epoch 7/20
938/938 [==============================] - 23s 24ms/step - loss: 12.4938 - accuracy: 0.100
0
Epoch 8/20
938/938 [==============================] - 23s 24ms/step - loss: 12.4649 - accuracy: 0.100
0
Epoch 9/20
938/938 [==============================] - 23s 24ms/step - loss: 12.3736 - accuracy: 0.100
0
Epoch 10/20
938/938 [==============================] - 23s 24ms/step - loss: 12.0865 - accuracy: 0.100
0
Epoch 11/20
938/938 [==============================] - 23s 24ms/step - loss: 11.2054 - accuracy: 0.100
0
Epoch 12/20
938/938 [==============================] - 23s 24ms/step - loss: 9.1145 - accuracy: 0.1000
Epoch 13/20
938/938 [==============================] - 23s 24ms/step - loss: 6.4614 - accuracy: 0.0994
Epoch 14/20
938/938 [==============================] - 23s 24ms/step - loss: 4.2494 - accuracy: 0.0992
Epoch 15/20
938/938 [==============================] - 23s 24ms/step - loss: 2.4702 - accuracy: 0.0985
Epoch 16/20
938/938 [==============================] - 23s 24ms/step - loss: 2.3043 - accuracy: 0.1011
Epoch 17/20
938/938 [==============================] - 23s 24ms/step - loss: 2.3091 - accuracy: 0.0996
Epoch 18/20
938/938 [==============================] - 23s 25ms/step - loss: 2.3229 - accuracy: 0.1001
Epoch 19/20
938/938 [==============================] - 23s 24ms/step - loss: 2.3805 - accuracy: 0.0994
Epoch 20/20
938/938 [==============================] - 23s 24ms/step - loss: 2.5515 - accuracy: 0.1013
```



question 2

Use the cyclical learning rate policy (with exponential decay) and train your network using batch size 64 and lrmin and lrmax values obtained in part 1. Here you will train till convergence and not just 5 epochs as in part 1. Plot train/validation loss and accuracy curve (similar to Figure 4 in reference)

- According to previous findings, the min max learning rate is around $(10^{-4}, 10^{-1})$

In [70]:

```
min_lr = 10e-4
max_lr = 10e-1
# Set learning rate to the best one find in question 1
opt = SGD(lr = min_lr, momentum=0.9)
clr = CyclicLR(
    mode='exp_range',
    base_lr=min_lr,
    max_lr=min_lr,
    step_size=config.STEP_SIZE * (trainX.shape[0] // config.BATCH_SIZE),
)
lr = lrf.lrs[lrf.losses.index(min(lrf.losses))]
model = MiniGoogLeNet.build(width=img_witdth, height=img_height, depth=1, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
H = model.fit(
    x=aug.flow(trainX, trainY, batch_size=config.BATCH_SIZE),
    validation_data=(testX, testY),
    steps_per_epoch=trainX.shape[0] // config.BATCH_SIZE,
    epochs=20,
    callbacks=[clr],
    verbose=1
)
```

```
Epoch 1/20
  5/937 [..............................] - ETA: 28s - loss: 2.5377 - accuracy: 0.1031 WARN
ING:tensorflow:Callback method `on_train_batch_end` is slow compared to the batch time (ba
tch time: 0.0173s vs `on_train_batch_end` time: 0.0423s). Check your callbacks.
937/937 [==============================] - 27s 27ms/step - loss: 0.9066 - accuracy: 0.6744
- val_loss: 0.6254 - val_accuracy: 0.7699
Epoch 2/20
937/937 [==============================] - 25s 27ms/step - loss: 0.5789 - accuracy: 0.7870
- val_loss: 0.5014 - val_accuracy: 0.8134
Epoch 3/20
937/937 [==============================] - 25s 26ms/step - loss: 0.4935 - accuracy: 0.8200
- val_loss: 0.5241 - val_accuracy: 0.8077
Epoch 4/20
937/937 [==============================] - 25s 26ms/step - loss: 0.4432 - accuracy: 0.8391
```

```
- val_loss: 0.4313 - val_accuracy: 0.8458
Epoch 5/20
937/937 [==============================] - 25s 27ms/step - loss: 0.4081 - accuracy: 0.8520
- val_loss: 0.3967 - val_accuracy: 0.8566
Epoch 6/20
937/937 [==============================] - 25s 26ms/step - loss: 0.3795 - accuracy: 0.8629
- val_loss: 0.4190 - val_accuracy: 0.8448
Epoch 7/20
937/937 [==============================] - 25s 27ms/step - loss: 0.3580 - accuracy: 0.8702
- val_loss: 0.3538 - val_accuracy: 0.8725
Epoch 8/20
937/937 [==============================] - 25s 27ms/step - loss: 0.3442 - accuracy: 0.8756
- val_loss: 0.3349 - val_accuracy: 0.8800
Epoch 9/20
937/937 [==============================] - 25s 27ms/step - loss: 0.3284 - accuracy: 0.8814
- val_loss: 0.3525 - val_accuracy: 0.8745
Epoch 10/20
937/937 [==============================] - 25s 27ms/step - loss: 0.3170 - accuracy: 0.8854
- val_loss: 0.3155 - val_accuracy: 0.8865
Epoch 11/20
937/937 [==============================] - 24s 26ms/step - loss: 0.3088 - accuracy: 0.8884
- val_loss: 0.3373 - val_accuracy: 0.8780
Epoch 12/20
937/937 [==============================] - 24s 26ms/step - loss: 0.2975 - accuracy: 0.8931
- val_loss: 0.3255 - val_accuracy: 0.8863
Epoch 13/20
937/937 [==============================] - 24s 26ms/step - loss: 0.2896 - accuracy: 0.8970
- val_loss: 0.2936 - val_accuracy: 0.8918
Epoch 14/20
937/937 [==============================] - 24s 26ms/step - loss: 0.2825 - accuracy: 0.8987
- val_loss: 0.2881 - val_accuracy: 0.8981
Epoch 15/20
937/937 [==============================] - 24s 26ms/step - loss: 0.2724 - accuracy: 0.9026
- val_loss: 0.2866 - val_accuracy: 0.8960
Epoch 16/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2675 - accuracy: 0.9032
- val_loss: 0.2851 - val_accuracy: 0.8986
Epoch 17/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2644 - accuracy: 0.9053
- val_loss: 0.2751 - val_accuracy: 0.8986
Epoch 18/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2560 - accuracy: 0.9074
- val_loss: 0.2920 - val_accuracy: 0.8955
Epoch 19/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2529 - accuracy: 0.9099
- val_loss: 0.2915 - val_accuracy: 0.8939
Epoch 20/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2464 - accuracy: 0.9118
- val_loss: 0.2731 - val_accuracy: 0.9015
```

In [71]:

```
# model is still not converged, train 20 more epochs
H_1 = model.fit(
    x=aug.flow(trainX, trainY, batch_size=config.BATCH_SIZE),
    validation_data=(testX, testY),
    steps_per_epoch=trainX.shape[0] // config.BATCH_SIZE,
    epochs=20,
    callbacks=[clr],
    verbose=1
)
```

```
Epoch 1/20
  6/937 [..............................] - ETA: 26s - loss: 0.1841 - accuracy: 0.9261WARNI
NG:tensorflow:Callback method `on_train_batch_end` is slow compared to the batch time (bat
ch time: 0.0171s vs `on_train_batch_end` time: 0.0403s). Check your callbacks.
937/937 [==============================] - 25s 26ms/step - loss: 0.2399 - accuracy: 0.9138
```

```
- val_loss: 0.3322 - val_accuracy: 0.8843
Epoch 2/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2381 - accuracy: 0.9147
- val_loss: 0.2595 - val_accuracy: 0.9083
Epoch 3/20
937/937 [==============================] - 25s 26ms/step - loss: 0.2305 - accuracy: 0.9179
- val_loss: 0.2787 - val_accuracy: 0.9037
Epoch 4/20
937/937 [==============================] - 25s 26ms/step - loss: 0.2285 - accuracy: 0.9177
- val_loss: 0.2677 - val_accuracy: 0.9034
Epoch 5/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2227 - accuracy: 0.9207
- val_loss: 0.2897 - val_accuracy: 0.8949
Epoch 6/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2227 - accuracy: 0.9201
- val_loss: 0.2844 - val_accuracy: 0.8924
Epoch 7/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2183 - accuracy: 0.9208
- val_loss: 0.2611 - val_accuracy: 0.9072
Epoch 8/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2128 - accuracy: 0.9242
- val_loss: 0.2397 - val_accuracy: 0.9129
Epoch 9/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2083 - accuracy: 0.9251
- val_loss: 0.2902 - val_accuracy: 0.8991
Epoch 10/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2091 - accuracy: 0.9249
- val_loss: 0.2990 - val_accuracy: 0.8961
Epoch 11/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2041 - accuracy: 0.9261
- val_loss: 0.2673 - val_accuracy: 0.9083
Epoch 12/20
937/937 [==============================] - 25s 27ms/step - loss: 0.2015 - accuracy: 0.9270
- val_loss: 0.2556 - val_accuracy: 0.9107
Epoch 13/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1976 - accuracy: 0.9293
- val_loss: 0.2535 - val_accuracy: 0.9120
Epoch 14/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1925 - accuracy: 0.9308
- val_loss: 0.2478 - val_accuracy: 0.9146
Epoch 15/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1886 - accuracy: 0.9319
- val_loss: 0.2926 - val_accuracy: 0.8999
Epoch 16/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1889 - accuracy: 0.9318
- val_loss: 0.2421 - val_accuracy: 0.9121
Epoch 17/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1853 - accuracy: 0.9338
- val_loss: 0.2752 - val_accuracy: 0.9058
Epoch 18/20
937/937 [==============================] - 24s 26ms/step - loss: 0.1830 - accuracy: 0.9332
- val_loss: 0.2456 - val_accuracy: 0.9133
Epoch 19/20
937/937 [==============================] - 24s 26ms/step - loss: 0.1817 - accuracy: 0.9342
- val_loss: 0.2363 - val_accuracy: 0.9169
Epoch 20/20
937/937 [==============================] - 24s 26ms/step - loss: 0.1795 - accuracy: 0.9356
- val_loss: 0.2300 - val_accuracy: 0.9208
```

In [72]:

```
# still not converged, train 20 more epochs
# model is still not converged, train 20 more epochs
H_2 = model.fit(
    x=aug.flow(trainX, trainY, batch_size=config.BATCH_SIZE),
    validation_data=(testX, testY),
    steps_per_epoch=trainX.shape[0] // config.BATCH_SIZE,
```

```
    epochs=20,
    callbacks=[clr],
    verbose=1
)
```

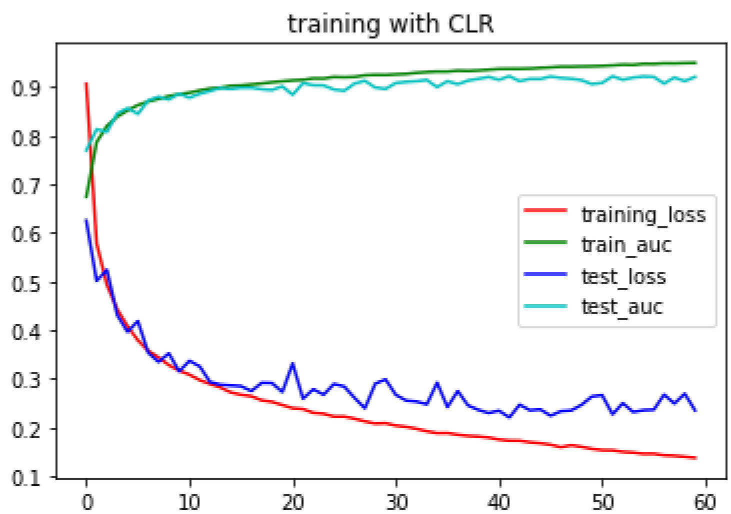
```
Epoch 1/20
  5/937 [..............................] - ETA: 29s - loss: 0.2446 - accuracy: 0.9250WARNI
NG:tensorflow:Callback method `on_train_batch_end` is slow compared to the batch time (bat
ch time: 0.0180s vs `on_train_batch_end` time: 0.0410s). Check your callbacks.
937/937 [==============================] - 25s 26ms/step - loss: 0.1754 - accuracy: 0.9372
- val_loss: 0.2347 - val_accuracy: 0.9150
Epoch 2/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1733 - accuracy: 0.9375
- val_loss: 0.2208 - val_accuracy: 0.9224
Epoch 3/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1731 - accuracy: 0.9375
- val_loss: 0.2472 - val_accuracy: 0.9122
Epoch 4/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1698 - accuracy: 0.9381
- val_loss: 0.2355 - val_accuracy: 0.9167
Epoch 5/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1679 - accuracy: 0.9394
- val_loss: 0.2374 - val_accuracy: 0.9167
Epoch 6/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1648 - accuracy: 0.9404
- val_loss: 0.2243 - val_accuracy: 0.9217
Epoch 7/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1597 - accuracy: 0.9420
- val_loss: 0.2337 - val_accuracy: 0.9187
Epoch 8/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1638 - accuracy: 0.9419
- val_loss: 0.2352 - val_accuracy: 0.9172
Epoch 9/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1605 - accuracy: 0.9424
- val_loss: 0.2467 - val_accuracy: 0.9141
Epoch 10/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1563 - accuracy: 0.9428
- val_loss: 0.2639 - val_accuracy: 0.9061
Epoch 11/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1537 - accuracy: 0.9434
- val_loss: 0.2667 - val_accuracy: 0.9087
Epoch 12/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1534 - accuracy: 0.9445
- val_loss: 0.2273 - val_accuracy: 0.9222
Epoch 13/20
937/937 [==============================] - 24s 26ms/step - loss: 0.1502 - accuracy: 0.9461
- val_loss: 0.2502 - val_accuracy: 0.9151
Epoch 14/20
937/937 [==============================] - 25s 26ms/step - loss: 0.1488 - accuracy: 0.9456
- val_loss: 0.2320 - val_accuracy: 0.9195
Epoch 15/20
937/937 [==============================] - 24s 26ms/step - loss: 0.1462 - accuracy: 0.9478
- val_loss: 0.2358 - val_accuracy: 0.9218
Epoch 16/20
937/937 [==============================] - 24s 26ms/step - loss: 0.1462 - accuracy: 0.9475
- val_loss: 0.2367 - val_accuracy: 0.9208
Epoch 17/20
937/937 [==============================] - 24s 26ms/step - loss: 0.1431 - accuracy: 0.9489
- val_loss: 0.2678 - val_accuracy: 0.9073
Epoch 18/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1421 - accuracy: 0.9486
- val_loss: 0.2493 - val_accuracy: 0.9197
Epoch 19/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1402 - accuracy: 0.9493
- val_loss: 0.2697 - val_accuracy: 0.9123
```

```
Epoch 20/20
937/937 [==============================] - 25s 27ms/step - loss: 0.1379 - accuracy: 0.9498
- val_loss: 0.2354 - val_accuracy: 0.9208
```

- utill 60 epochs, although the training loss is still decreasing, the test accuracy is steady, so I think 60 epochs is enough for the training.

In [77]:

```
epochs_vis = list(range(60))
train_loss = np.concatenate([H.history['loss'], H_1.history['loss'], H_2.history['loss']],
train_auc = np.concatenate([H.history['accuracy'], H_1.history['accuracy'], H_2.history['a
test_loss = np.concatenate([H.history['val_loss'], H_1.history['val_loss'], H_2.history['v
test_auc = np.concatenate([H.history['val_accuracy'], H_1.history['val_accuracy'], H_2.his
plt.plot(epochs_vis, train_loss, label = 'training_loss', c = 'r')
plt.plot(epochs_vis, train_auc, label = 'train_auc', c = 'g')
plt.plot(epochs_vis, test_loss, label = 'test_loss', c = 'b')
plt.plot(epochs_vis, test_auc, label = 'test_auc', c = 'c')
plt.legend()
plt.title('training with CLR')
plt.show()
```



question 3

We want to test if increasing batch size for a fixed learning rate has the same effect as decreasing learning rate for a fixed batch size. Fix learning rate to lrmax and train your network starting with batch size 32 and incrementally going upto 16384 (in increments of a factor of 2; like 32, 64...). You can choose a step size (in terms of number of iterations) to increment the batch size. If your GPU cannot handle large batch sizes, you need to employ effective batch size approach as discussed in Lecture 3 to simulate large batches. Plot the training loss as a function of batch size. Is the generalization of your final model similar or different than cyclical learning rate policy?

In [3]:

```
def run_batch_experiment(batch_size, num_epochs):
    all_loss = []
    optimizer = SGD(learning_rate=0.1, momentum=0.9)
    model = MiniGoogLeNet.build(width=img_witdth, height=img_height, depth=1, classes=10)
    batch_size = batch_size
    train_dataset = tf.data.Dataset.from_tensor_slices((trainX, trainY))
    train_dataset = train_dataset.shuffle(buffer_size=1024).batch(batch_size)
    epochs = num_epochs
    for epoch in range(epochs):
        print('current working epoch:' + str(epoch), end='\r')
        if batch_size<=512:
            for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):
```

```
                print('current working epoch:'+ str(epoch)+ '   current working batch:' +
                      flush=True,end='\r')
                with tf.GradientTape() as tape:
                    logits = model(x_batch_train, training=True)  # Logits for this miniba
                    loss_value = keras.losses.categorical_crossentropy(y_batch_train, logi
                grads = tape.gradient(loss_value, model.trainable_weights)
                final_grad = []
                for grad_layer in grads:
                    final_grad.append(grad_layer/batch_size)
                optimizer.apply_gradients(zip(final_grad, model.trainable_weights))
                all_loss.append(loss_value)
        else:
            for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):
                print('current working epoch:'+ str(epoch)+ '   current working batch:' +
                      flush=True,end='\r')
                start = 0
                end = 512
                grads_list = []
                loss_list_this_micro_batch = []
                for i in range(int(batch_size/512)):
                    if start+512 > x_batch_train.shape[0]:
                        batch_x = x_batch_train[start:,:,:,:]
                        batch_y = y_batch_train[start:,:]
                    else:
                        batch_x = x_batch_train[start:end,:,:,:]
                        batch_y = y_batch_train[start:end,:]
                    if batch_x.shape[0] > 0:
                        with tf.GradientTape() as tape:
                            logits = model(batch_x, training=True)  # Logits for this mini
                            loss_value = keras.losses.categorical_crossentropy(batch_y, lo
                        grads = tape.gradient(loss_value, model.trainable_weights)
                        grads_list.append(grads)
                        loss_list_this_micro_batch.append(loss_value)
                        start += 512
                        end += 512
                    else:
                        pass
                # calculate mean grads for each layer
                final_layer_grads = []
                loss_list_this_micro_batch = np.concatenate(loss_list_this_micro_batch,axi
                all_loss.append(loss_list_this_micro_batch)
                for layer_index in range(len(grads_list[0])):
                    grads_this_layer = []
                    for micro_batch_index in range(len(grads_list)):
                        grads_this_layer.append(grads_list[micro_batch_index][layer_index]
                    final_layer_grads.append(sum(grads_this_layer)/((len(grads_list)*512))
                optimizer.apply_gradients(zip(final_layer_grads, model.trainable_weights))
    return all_loss
```

In [4]:

```
loss_32 = run_batch_experiment(32,10)
```

```
current working epoch:9   current working batch:1874/1875
```

In [5]:

```
loss_64 = run_batch_experiment(64,10)
```

```
current working epoch:9   current working batch:937/938
```

In [6]:

```
loss_128 = run_batch_experiment(128,10)
```

```
current working epoch:9   current working batch:468/469
```

In [7]:

```
loss_256 = run_batch_experiment(256,10)
```

```
current working epoch:9   current working batch:234/235
```

In [8]:
```python
loss_512 = run_batch_experiment(512,10)
```

```
current working epoch:9   current working batch:117/118
```

In [9]:
```python
loss_1024 = run_batch_experiment(1024,10)
```

```
current working epoch:9   current working batch:58/59
```

In [10]:
```python
loss_2048 = run_batch_experiment(2048,10)
```

```
current working epoch:9   current working batch:29/30
```

In [11]:
```python
loss_4096 = run_batch_experiment(4096,10)
```

```
current working epoch:9   current working batch:14/15
```

In [12]:
```python
loss_8192 = run_batch_experiment(8192,10)
```

```
current working epoch:9   current working batch:7/8
```

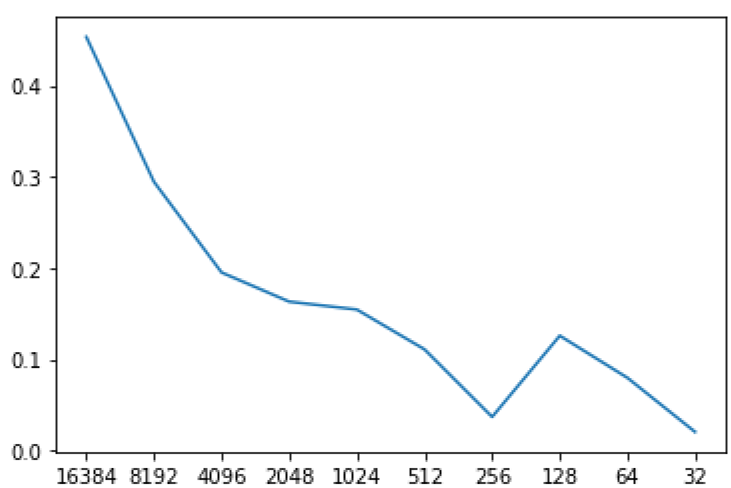
In [13]:
```python
loss_16384 = run_batch_experiment(16384,10)
```

```
current working epoch:9   current working batch:3/4
```

In [40]:
```python
losses = [loss_32,loss_64,loss_128,loss_256,loss_512,loss_1024,loss_2048,loss_4096,loss_81
losses_mean = []
for i in losses:
    losses_mean.append(np.mean(i[-1]))
```

In [41]:
```python
vis_x = ['32','64','128','256','512','1024','2048','4096','8192','16384']
vis_x.reverse()
losses_mean.reverse()
plt.plot(vis_x,losses_mean)
```

Out[41]:
```
[<matplotlib.lines.Line2D at 0x17f60fe9c70>]
```



- From the plot we can see.
  - Large learning rate = small batch size => having lower training loss
  - We see a similar curve as CLR algorithm

Problem 3

question 1

Calculate the number of parameters in Alexnet. You will have to show calculations for each layer and then sum it to obtain the total number of parameters in Alexnet. When calculating you will need to account for all the filters (size, strides, padding) at each layer. Look at Sec. 3.5 and Figure 2 in Alexnet paper (see reference). Points will only be given when explicit calculations are shown for each layer.

conv1: $(11 \times 11) \times 3 \times 96 + 96 = 34944$

conv2: $(5 \times 5) \times 96 \times 256 + 256 = 614656$

conv3: $(3 \times 3) \times 256 \times 384 + 384 = 885120$

conv4: $(3 \times 3) \times 384 \times 384 + 384 = 1327488$

conv5: $(3 \times 3) \times 384 \times 256 + 256 = 884992$

fc1: $(6 \times 6) \times 256 \times 4096 + 4096 = 37752832$

fc2: $4096 \times 4096 + 4096 = 16781312$

fc3: $4096 \times 1000 + 1000 = 4097000$

total_parameters: conv1+conv2+conv3+conv4+conv5+fc1+fc2+fc3 = 62378344

question 2

VGG (Simonyan et al.) has an extremely homogeneous architecture that only performs 3x3 convolutions with stride 1 and pad 1 and 2x2 max pooling with stride 2 (and no padding) from the beginning to the end. However VGGNet is very expensive to evaluate and uses a lot more memory and parameters. Refer to VGG19 architecture on page 3 in Table 1 of the paper by Simonyan et al. You need to complete Table 1 below for calculating activation units and parameters at each layer in VGG19 (without counting biases). Its been partially filled for you.

In [28]:

```
col_1 = ['input', 'Conv3-64', 'Conv3-64', 'POOL2',
          'CONV3-128','CONV3-128','POOL2',
          'CONV3-256','CONV3-256','CONV3-256','CONV3-256','POOL2',
         'CONV3-512','CONV3-512','CONV3-512','CONV3-512','POOL2',
         'CONV3-512','CONV3-512','CONV3-512','CONV3-512','POOL2',
         'FC','FC','FC','TOTAL']
col_2 = ['224*224*3 = 150K','224*224*64 = 3.2M','224*224*64 = 3.2M','112*112*64 = 800K',
         '112*112*128 = 1.6M','112*112*128 = 1.6M','56*56*128 = 400K',
         '56*56*256 = 800K','56*56*256 = 800K','56*56*256 = 800K','56*56*256 = 800K','28*28
         '28*28*512 = 400K','28*28*512 = 400K','28*28*512 = 400K','28*28*512 = 400K','14*14
         '14*14*512 = 100K','14*14*512 = 100K','14*14*512 = 100K','14*14*512 = 100K','7*7*5
         '4096','4096','1000','sum(all)']
col_3 = ['0', '(3*3*3)*64=1728', '(3*3*64)*64 = 36864', '0',
          '(3*3*64)*128 = 73728','(3*3*128)*128 = 147456','0',
          '(3*3*128)*256 = 294912','(3*3*256)*256 = 589824','(3*3*256)*256 = 589824','(3*3
         '(3*3*256)*512 = 117948','(3*3*512)*512 = 2358296','(3*3*512)*512 = 2358296','(3*3
         '(3*3*512)*512 = 2358296','(3*3*512)*512 = 2358296','(3*3*512)*512 = 2358296','(3*
         '7*7*512*4096','4096*4096','4096*1000','sum(all)']
table = pd.DataFrame([col_1, col_2, col_3]).transpose()
table.rename({0:'Layer',1:'Number of Activations (Memory)',2:'Parameters(Compute)'}, axis
```

Out[28]:

	Layer	Number of Activations (Memory)	Parameters(Compute)
0	input	224*224*3 = 150K	0
1	Conv3-64	224*224*64 = 3.2M	(3*3*3)*64=1728
2	Conv3-64	224*224*64 = 3.2M	(3*3*64)*64 = 36864
3	POOL2	112*112*64 = 800K	0
4	CONV3-128	112*112*128 = 1.6M	(3*3*64)*128 = 73728
5	CONV3-128	112*112*128 = 1.6M	(3*3*128)*128 = 147456
6	POOL2	56*56*128 = 400K	0
7	CONV3-256	56*56*256 = 800K	(3*3*128)*256 = 294912
8	CONV3-256	56*56*256 = 800K	(3*3*256)*256 = 589824
9	CONV3-256	56*56*256 = 800K	(3*3*256)*256 = 589824
10	CONV3-256	56*56*256 = 800K	(3*3*256)*256 = 589824
11	POOL2	28*28*256 = 200K	0
12	CONV3-512	28*28*512 = 400K	(3*3*256)*512 = 117948
13	CONV3-512	28*28*512 = 400K	(3*3*512)*512 = 2358296
14	CONV3-512	28*28*512 = 400K	(3*3*512)*512 = 2358296
15	CONV3-512	28*28*512 = 400K	(3*3*512)*512 = 2358296
16	POOL2	14*14*512 = 100K	0
17	CONV3-512	14*14*512 = 100K	(3*3*512)*512 = 2358296
18	CONV3-512	14*14*512 = 100K	(3*3*512)*512 = 2358296
19	CONV3-512	14*14*512 = 100K	(3*3*512)*512 = 2358296
20	CONV3-512	14*14*512 = 100K	(3*3*512)*512 = 2358296
21	POOL2	7*7*512 = 25K	0
22	FC	4096	7*7*512*4096
23	FC	4096	4096*4096
24	FC	1000	4096*1000
25	TOTAL	sum(all)	sum(all)

question 3

VGG architectures have smaller filters but deeper networks compared to Alexnet (3x3 compared to 11x11 or 5x5). Show that 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). Use this to calculate the receptive field of 3 filters of size 5x5.

- N_{th} output size: 11*, input_size = (1+F-1)* (1+F-1)
- $N-1_{th}$ output size: (1+F-1) * (1+F-1), input_size = (1+2F-2)(1+2F-2)
- $N-2_{th}$ output size: (1+2F-2)(1+2F-2), input_size = (1+3F-3)(1+3F-3)
- ......
- ......

- $input\_layer$: input_size = (1+NF-N)(1+NF-N)
- **Receptive field of 3 filters of size 5x5**:
  - 3_{rd} $Filter$: output size = 1*1, input size = (1+5-1)(1+5-1)
  - 2_{nd} $Filter$: output size = (1+5-1)(1+5-1) , input size = (1+10-2)(1+10-2)
  - 1_{st} $Filter$: output size = (1+10-2)(1+10-2) , input size = (1+15-3)(1+15-3)
  - receptice field is (1+15-3)(1+15-3) = 13*13

question 4

The original Googlenet paper (Szegedy et al.) proposes two architectures for Inception module, shown in Figure 2 on page 5 of the paper, referred to as naive and dimensionality reduction respectively.

(a)

What is the general idea behind designing an inception module (parallel convolutional filters of different sizes with a pooling followed by concatenation) in a convolutional neural network ?

- By using different size of filters on the same activation maps and then concatenate them togather will provide model the ability to capture fine grind features as well as coarse grind features, by using the combined features will improve final model performance.

(b)

Assuming the input to inception module (referred to as "previous layer" in Figure 2 of the paper) has size 32x32x256, calculate the output size after filter concatenation for the naive and dimensionality reduction inception architectures with number of filters given in Figure 1.

- (Naive version):
  - input: $32 * 32 * 256$
  - $1 * 1$ conv output: $32 * 32 * 128$
  - $3 * 3$ conv output: $32 * 32 * 192$
  - $5 * 5$ conv output: $32 * 32 * 96$
  - $3 * 3$ maxpooling output: $32 * 32 * 256$
  - concatenate: $32 * 32 * 672$
- (dimension reduction version):
  - input: $32 * 32 * 256$
  - $1 * 1$ conv output: $32 * 32 * 128$
  - $1 * 1 + 3 * 3$ conv output: $32 * 32 * 192$
  - $1 * 1 + 5 * 5$ conv output: $32 * 32 * 96$
  - $3 * 3 + 1 * 1$ pooling *conv: $3232*64$*
  - concatenate: $32 * 32 * 480$

(c)

Next calculate the total number of convolutional operations for each of the two inception architecture again assuming the input to the module has dimensions 32x32x256 and number of filters given in Figure 1.

- (Naive version):
  - $1 * 1$ conv op: $32 * 32 * 128 * 1 * 1 * 256 = 33554432$

- $3 * 3$ conv op: $32 * 32 * 192 * 3 * 3 * 256 = 452984832$
- $5 * 5$ conv op: $32 * 32 * 96 * 5 * 5 * 256 = 629145600$
- total ops: 1115684864

- (dimension reduction version):

  - $1 * 1 * 64$ conv op: $32 * 32 * 64 * 1 * 1 * 256 * 3 = 50331648$
  - $1 * 1 * 128$ conv op: $32 * 32 * 128 * 1 * 1 * 256 = 33554432$
  - $3 * 3$ conv op: $32 * 32 * 192 * 3 * 3 * 64 = 113246208$
  - $5 * 5$ conv op: $32 * 32 * 96 * 5 * 5 * 64 = 157286400$
  - total ops: 354418688

(d)

Based on the calculations in part (c) explain the problem with naive architecture and how dimensionality reduction architecture helps (Hint: compare computational complexity). How much is the computational saving ?

- Problem of naive architecture: naive architecture need too many operations (need 1115684864 ops), training can be very computional intense.
- Dimensionality reduction first use $1 * 1$ conv layers to reduce the input channel from 256 to 64, and do $3 * 3$ and $5 * 5$ conv layers on the 64 channel input instead the total of 256 channels, thus greatly reduced computational complexity.
- Total saved ops: $1115684864 - 354418688 = 761266176$

Problem 4

In [6]:
```
def apply_mask(image, size=12, 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
```

question 1

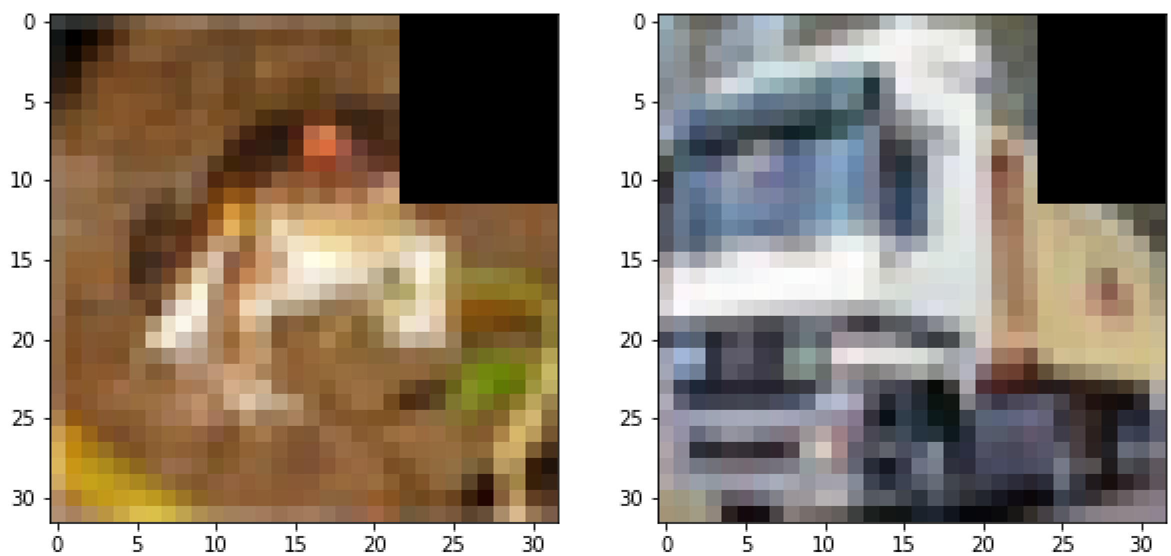
Explain cutout regularization and its advantages compared to simple dropout (as argued in the paper by DeVries et al) in your own words. Select any 2 images from CIFAR10 and show how does these images look after applying cutout. Use a square-shaped fixed size zero-mask to a random location of each image and generate its cutout version. Refer to the paper by DeVries et al (Section 3) and associated github repository.

- As stated in the paper, traditional drop out works fine for fully connected model,but it's less powerful for training a convlutional model. Two main reason for this is: 1, Conv layers have fewer parameters than fully-connected, thus require less regularization. 2, Input of conv layers is images, images have positional factor, information dropped can be repersented by near pixles, thus drop out algorithm is less effective in reducing co-adaptation.

In [2]:
```
from keras.datasets import cifar10
import tensorflow
#import dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

In [106…
```
fig, ax = plt.subplots(1,2,figsize = (10,5))
ax[0].imshow(apply_mask(X_train[0].copy()))
ax[1].imshow(apply_mask(X_train[1].copy()))
```

Out[106…
```
<matplotlib.image.AxesImage at 0x195f8de7df0>
```



question 2

Using CIFAR10 datasest and Resnet-44 we will first apply simple data augmentation as in He et al. (look at Section 4.2 of He et al.) and train the model with batch size 64. Note that testing is always done with original images. Plot validation error vs number of training epochs.

In [3]:
```
'''
ResNet-44 code from
https://github.com/PacktPublishing/Advanced-Deep-Learning-with-Keras/blob/master/chapter2-
'''

n = 7
depth = n * 6 + 2
input_shape = X_train[0].shape

def lr_schedule(epoch):
    """Learning Rate Schedule
    Learning rate is scheduled to be reduced after 80, 120, 160, 180 epochs.
    Called automatically every epoch as part of callbacks during training.
    # Arguments
        epoch (int): The number of epochs
    # Returns
        lr (float32): learning rate
    """
    lr = 1e-3
    if epoch > 180:
        lr *= 0.5e-3
```

```
    elif epoch > 160:
        lr *= 1e-3
    elif epoch > 120:
        lr *= 1e-2
    elif epoch > 80:
        lr *= 1e-1
    return lr


def resnet_layer(inputs,
                 num_filters=16,
                 kernel_size=3,
                 strides=1,
                 activation='relu',
                 batch_normalization=True,
                 conv_first=True):
    """2D Convolution-Batch Normalization-Activation stack builder
    Arguments:
        inputs (tensor): input tensor from input image or previous layer
        num_filters (int): Conv2D number of filters
        kernel_size (int): Conv2D square kernel dimensions
        strides (int): Conv2D square stride dimensions
        activation (string): activation name
        batch_normalization (bool): whether to include batch normalization
        conv_first (bool): conv-bn-activation (True) or
            bn-activation-conv (False)
    Returns:
        x (tensor): tensor as input to the next layer
    """
    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):
    """ResNet Version 1 Model builder [a]
    Stacks of 2 x (3 x 3) Conv2D-BN-ReLU
    Last ReLU is after the shortcut connection.
    At the beginning of each stage, the feature map size is halved
    (downsampled) by a convolutional layer with strides=2, while
    the number of filters is doubled. Within each stage,
    the layers have the same number filters and the
    same number of filters.
    Features maps sizes:
    stage 0: 32x32, 16
    stage 1: 16x16, 32
    stage 2:  8x8,  64
    The Number of parameters is approx the same as Table 6 of [a]:
```

```
    ResNet20 0.27M
    ResNet32 0.46M
    ResNet44 0.66M
    ResNet56 0.85M
    ResNet110 1.7M
    Arguments:
        input_shape (tensor): shape of input image tensor
        depth (int): number of core convolutional layers
        num_classes (int): number of classes (CIFAR10 has 10)
    Returns:
        model (Model): Keras model instance
    """
    if (depth - 2) % 6 != 0:
        raise ValueError('depth should be 6n+2 (eg 20, 32, 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
            # first layer but not first stack
            if stack > 0 and res_block == 0:
                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)
            # first layer but not first stack
            if stack > 0 and res_block == 0:
                # 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 = 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


model = resnet_v1(input_shape=input_shape, depth=depth)



model.compile(loss='sparse_categorical_crossentropy',
```

```
                optimizer=Adam(lr=lr_schedule(0)),
                metrics=['acc'])
```

```
C:\Learning\SoftWare\anaconda\lib\site-packages\keras\optimizer_v2\adam.py:105: UserWarnin
g: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

In [87]:

```
lr_scheduler = LearningRateScheduler(lr_schedule)

lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                               cooldown=0,
                               patience=5,
                               min_lr=0.5e-6)
callbacks = [lr_reducer, lr_scheduler]

history_nocutoff = model.fit(X_train, y_train,
              batch_size=64,
              epochs=100,
              validation_data=(X_test, y_test),
              shuffle=True,
              callbacks=callbacks)
```

```
Learning rate:  0.001
Epoch 1/100
782/782 [==============================] - 27s 29ms/step - loss: 1.7550 - acc: 0.4888 - va
l_loss: 1.6173 - val_acc: 0.5398 - lr: 0.0010
Learning rate:  0.001
Epoch 2/100
782/782 [==============================] - 22s 28ms/step - loss: 1.2806 - acc: 0.6535 - va
l_loss: 1.3426 - val_acc: 0.6387 - lr: 0.0010
Learning rate:  0.001
Epoch 3/100
782/782 [==============================] - 22s 28ms/step - loss: 1.0666 - acc: 0.7239 - va
l_loss: 1.9748 - val_acc: 0.5295 - lr: 0.0010
Learning rate:  0.001
Epoch 4/100
782/782 [==============================] - 23s 29ms/step - loss: 0.9348 - acc: 0.7662 - va
l_loss: 1.2534 - val_acc: 0.6730 - lr: 0.0010
Learning rate:  0.001
Epoch 5/100
782/782 [==============================] - 23s 29ms/step - loss: 0.8453 - acc: 0.7961 - va
l_loss: 1.5322 - val_acc: 0.6149 - lr: 0.0010
Learning rate:  0.001
Epoch 6/100
782/782 [==============================] - 23s 29ms/step - loss: 0.7761 - acc: 0.8197 - va
l_loss: 1.4247 - val_acc: 0.6707 - lr: 0.0010
Learning rate:  0.001
Epoch 7/100
782/782 [==============================] - 23s 29ms/step - loss: 0.7227 - acc: 0.8366 - va
l_loss: 0.9942 - val_acc: 0.7537 - lr: 0.0010
Learning rate:  0.001
Epoch 8/100
782/782 [==============================] - 23s 29ms/step - loss: 0.6784 - acc: 0.8532 - va
l_loss: 1.0600 - val_acc: 0.7434 - lr: 0.0010
Learning rate:  0.001
Epoch 9/100
782/782 [==============================] - 23s 30ms/step - loss: 0.6407 - acc: 0.8664 - va
l_loss: 1.2419 - val_acc: 0.6924 - lr: 0.0010
Learning rate:  0.001
Epoch 10/100
782/782 [==============================] - 23s 29ms/step - loss: 0.6078 - acc: 0.8775 - va
l_loss: 1.1271 - val_acc: 0.7357 - lr: 0.0010
Learning rate:  0.001
Epoch 11/100
```

```
782/782 [==============================] - 23s 29ms/step - loss: 0.5724 - acc: 0.8927 - va
l_loss: 1.3101 - val_acc: 0.7193 - lr: 0.0010
Learning rate:  0.001
Epoch 12/100
782/782 [==============================] - 23s 29ms/step - loss: 0.5563 - acc: 0.8982 - va
l_loss: 1.1446 - val_acc: 0.7297 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 13/100
782/782 [==============================] - 23s 29ms/step - loss: 0.5313 - acc: 0.9070 - va
l_loss: 1.3804 - val_acc: 0.6985 - lr: 0.0010
Learning rate:  0.001
Epoch 14/100
782/782 [==============================] - 23s 29ms/step - loss: 0.5172 - acc: 0.9159 - va
l_loss: 1.0397 - val_acc: 0.7829 - lr: 0.0010
Learning rate:  0.001
Epoch 15/100
782/782 [==============================] - 23s 29ms/step - loss: 0.5022 - acc: 0.9208 - va
l_loss: 0.9681 - val_acc: 0.7997 - lr: 0.0010
Learning rate:  0.001
Epoch 16/100
782/782 [==============================] - 23s 29ms/step - loss: 0.4937 - acc: 0.9260 - va
l_loss: 1.1980 - val_acc: 0.7527 - lr: 0.0010
Learning rate:  0.001
Epoch 17/100
782/782 [==============================] - 23s 30ms/step - loss: 0.4833 - acc: 0.9299 - va
l_loss: 1.5867 - val_acc: 0.6770 - lr: 0.0010
Learning rate:  0.001
Epoch 18/100
782/782 [==============================] - 21s 27ms/step - loss: 0.4652 - acc: 0.9376 - va
l_loss: 1.1873 - val_acc: 0.7612 - lr: 0.0010
Learning rate:  0.001
Epoch 19/100
782/782 [==============================] - 21s 27ms/step - loss: 0.4648 - acc: 0.9389 - va
l_loss: 1.2333 - val_acc: 0.7551 - lr: 0.0010
Learning rate:  0.001
Epoch 20/100
782/782 [==============================] - 24s 30ms/step - loss: 0.4589 - acc: 0.9416 - va
l_loss: 1.3614 - val_acc: 0.7613 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 21/100
782/782 [==============================] - 24s 30ms/step - loss: 0.4557 - acc: 0.9424 - va
l_loss: 2.0787 - val_acc: 0.6301 - lr: 0.0010
Learning rate:  0.001
Epoch 22/100
782/782 [==============================] - 23s 29ms/step - loss: 0.4421 - acc: 0.9467 - va
l_loss: 1.0557 - val_acc: 0.7967 - lr: 0.0010
Learning rate:  0.001
Epoch 23/100
782/782 [==============================] - 23s 29ms/step - loss: 0.4506 - acc: 0.9464 - va
l_loss: 1.1290 - val_acc: 0.7908 - lr: 0.0010
Learning rate:  0.001
Epoch 24/100
782/782 [==============================] - 23s 30ms/step - loss: 0.4384 - acc: 0.9494 - va
l_loss: 1.5849 - val_acc: 0.6999 - lr: 0.0010
Learning rate:  0.001
Epoch 25/100
782/782 [==============================] - 23s 29ms/step - loss: 0.4435 - acc: 0.9489 - va
l_loss: 1.0643 - val_acc: 0.7975 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 26/100
782/782 [==============================] - 23s 30ms/step - loss: 0.4380 - acc: 0.9511 - va
l_loss: 1.1088 - val_acc: 0.7989 - lr: 0.0010
Learning rate:  0.001
Epoch 27/100
782/782 [==============================] - 23s 30ms/step - loss: 0.4364 - acc: 0.9507 - va
l_loss: 1.2352 - val_acc: 0.7657 - lr: 0.0010
```

```
Learning rate:  0.001
Epoch 28/100
782/782 [==============================] - 23s 30ms/step - loss: 0.4231 - acc: 0.9563 - va
l_loss: 1.2456 - val_acc: 0.7760 - lr: 0.0010
Learning rate:  0.001
Epoch 29/100
782/782 [==============================] - 23s 29ms/step - loss: 0.4325 - acc: 0.9529 - va
l_loss: 1.1476 - val_acc: 0.7680 - lr: 0.0010
Learning rate:  0.001
Epoch 30/100
782/782 [==============================] - 24s 31ms/step - loss: 0.4250 - acc: 0.9552 - va
l_loss: 1.3683 - val_acc: 0.7578 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 31/100
782/782 [==============================] - 26s 33ms/step - loss: 0.4267 - acc: 0.9541 - va
l_loss: 1.3922 - val_acc: 0.7457 - lr: 0.0010
Learning rate:  0.001
Epoch 32/100
782/782 [==============================] - 25s 32ms/step - loss: 0.4181 - acc: 0.9577 - va
l_loss: 1.0145 - val_acc: 0.8084 - lr: 0.0010
Learning rate:  0.001
Epoch 33/100
782/782 [==============================] - 23s 30ms/step - loss: 0.4152 - acc: 0.9580 - va
l_loss: 1.0878 - val_acc: 0.7963 - lr: 0.0010
Learning rate:  0.001
Epoch 34/100
782/782 [==============================] - 21s 26ms/step - loss: 0.4213 - acc: 0.9554 - va
l_loss: 1.1551 - val_acc: 0.7879 - lr: 0.0010
Learning rate:  0.001
Epoch 35/100
782/782 [==============================] - 21s 27ms/step - loss: 0.4137 - acc: 0.9590 - va
l_loss: 1.3335 - val_acc: 0.7570 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 36/100
782/782 [==============================] - 21s 26ms/step - loss: 0.4092 - acc: 0.9602 - va
l_loss: 1.1011 - val_acc: 0.7955 - lr: 0.0010
Learning rate:  0.001
Epoch 37/100
782/782 [==============================] - 21s 27ms/step - loss: 0.4101 - acc: 0.9582 - va
l_loss: 1.5619 - val_acc: 0.7062 - lr: 0.0010
Learning rate:  0.001
Epoch 38/100
782/782 [==============================] - 21s 27ms/step - loss: 0.4078 - acc: 0.9598 - va
l_loss: 1.1393 - val_acc: 0.7810 - lr: 0.0010
Learning rate:  0.001
Epoch 39/100
782/782 [==============================] - 21s 27ms/step - loss: 0.4085 - acc: 0.9589 - va
l_loss: 1.1955 - val_acc: 0.7781 - lr: 0.0010
Learning rate:  0.001
Epoch 40/100
782/782 [==============================] - 21s 27ms/step - loss: 0.4055 - acc: 0.9600 - va
l_loss: 1.2685 - val_acc: 0.7662 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 41/100
782/782 [==============================] - 21s 27ms/step - loss: 0.4055 - acc: 0.9596 - va
l_loss: 1.2871 - val_acc: 0.7605 - lr: 0.0010
Learning rate:  0.001
Epoch 42/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3989 - acc: 0.9616 - va
l_loss: 1.3699 - val_acc: 0.7502 - lr: 0.0010
Learning rate:  0.001
Epoch 43/100
782/782 [==============================] - 24s 30ms/step - loss: 0.4011 - acc: 0.9609 - va
l_loss: 1.1185 - val_acc: 0.7956 - lr: 0.0010
Learning rate:  0.001
Epoch 44/100
```

```
782/782 [==============================] - 24s 30ms/step - loss: 0.4013 - acc: 0.9604 - va
l_loss: 1.7085 - val_acc: 0.7234 - lr: 0.0010
Learning rate:  0.001
Epoch 45/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3972 - acc: 0.9623 - va
l_loss: 1.4187 - val_acc: 0.7606 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 46/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3901 - acc: 0.9641 - va
l_loss: 1.4183 - val_acc: 0.7643 - lr: 0.0010
Learning rate:  0.001
Epoch 47/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3961 - acc: 0.9614 - va
l_loss: 1.3729 - val_acc: 0.7549 - lr: 0.0010
Learning rate:  0.001
Epoch 48/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3895 - acc: 0.9644 - va
l_loss: 1.2230 - val_acc: 0.7851 - lr: 0.0010
Learning rate:  0.001
Epoch 49/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3942 - acc: 0.9616 - va
l_loss: 1.2709 - val_acc: 0.7801 - lr: 0.0010
Learning rate:  0.001
Epoch 50/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3947 - acc: 0.9629 - va
l_loss: 1.2202 - val_acc: 0.7738 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 51/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3820 - acc: 0.9665 - va
l_loss: 1.1678 - val_acc: 0.7869 - lr: 0.0010
Learning rate:  0.001
Epoch 52/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3893 - acc: 0.9625 - va
l_loss: 1.2070 - val_acc: 0.7814 - lr: 0.0010
Learning rate:  0.001
Epoch 53/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3928 - acc: 0.9612 - va
l_loss: 1.4276 - val_acc: 0.7531 - lr: 0.0010
Learning rate:  0.001
Epoch 54/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3777 - acc: 0.9666 - va
l_loss: 1.3229 - val_acc: 0.7718 - lr: 0.0010
Learning rate:  0.001
Epoch 55/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3778 - acc: 0.9665 - va
l_loss: 1.2855 - val_acc: 0.7777 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 56/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3908 - acc: 0.9618 - va
l_loss: 1.2299 - val_acc: 0.7853 - lr: 0.0010
Learning rate:  0.001
Epoch 57/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3762 - acc: 0.9667 - va
l_loss: 1.4009 - val_acc: 0.7630 - lr: 0.0010
Learning rate:  0.001
Epoch 58/100
782/782 [==============================] - 24s 30ms/step - loss: 0.3738 - acc: 0.9669 - va
l_loss: 1.3599 - val_acc: 0.7709 - lr: 0.0010
Learning rate:  0.001
Epoch 59/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3763 - acc: 0.9663 - va
l_loss: 1.5912 - val_acc: 0.7366 - lr: 0.0010
Learning rate:  0.001
Epoch 60/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3855 - acc: 0.9635 - va
l_loss: 1.2113 - val_acc: 0.7795 - lr: 3.1623e-04
```

```
Learning rate:  0.001
Epoch 61/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3725 - acc: 0.9675 - va
l_loss: 1.1424 - val_acc: 0.7958 - lr: 0.0010
Learning rate:  0.001
Epoch 62/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3711 - acc: 0.9669 - va
l_loss: 1.1426 - val_acc: 0.7996 - lr: 0.0010
Learning rate:  0.001
Epoch 63/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3805 - acc: 0.9639 - va
l_loss: 1.4362 - val_acc: 0.7681 - lr: 0.0010
Learning rate:  0.001
Epoch 64/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3706 - acc: 0.9678 - va
l_loss: 1.2137 - val_acc: 0.7718 - lr: 0.0010
Learning rate:  0.001
Epoch 65/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3739 - acc: 0.9655 - va
l_loss: 1.1948 - val_acc: 0.7770 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 66/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3681 - acc: 0.9674 - va
l_loss: 1.2301 - val_acc: 0.7741 - lr: 0.0010
Learning rate:  0.001
Epoch 67/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3716 - acc: 0.9661 - va
l_loss: 1.7558 - val_acc: 0.7163 - lr: 0.0010
Learning rate:  0.001
Epoch 68/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3701 - acc: 0.9671 - va
l_loss: 1.4243 - val_acc: 0.7509 - lr: 0.0010
Learning rate:  0.001
Epoch 69/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3653 - acc: 0.9672 - va
l_loss: 1.2507 - val_acc: 0.7835 - lr: 0.0010
Learning rate:  0.001
Epoch 70/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3653 - acc: 0.9679 - va
l_loss: 1.3538 - val_acc: 0.7600 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 71/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3705 - acc: 0.9661 - va
l_loss: 1.3749 - val_acc: 0.7570 - lr: 0.0010
Learning rate:  0.001
Epoch 72/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3558 - acc: 0.9711 - va
l_loss: 1.1880 - val_acc: 0.7897 - lr: 0.0010
Learning rate:  0.001
Epoch 73/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3598 - acc: 0.9684 - va
l_loss: 1.3043 - val_acc: 0.7806 - lr: 0.0010
Learning rate:  0.001
Epoch 74/100
782/782 [==============================] - 23s 29ms/step - loss: 0.3678 - acc: 0.9659 - va
l_loss: 1.4826 - val_acc: 0.7464 - lr: 0.0010
Learning rate:  0.001
Epoch 75/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3547 - acc: 0.9708 - va
l_loss: 1.2425 - val_acc: 0.7740 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 76/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3631 - acc: 0.9670 - va
l_loss: 1.2678 - val_acc: 0.7683 - lr: 0.0010
Learning rate:  0.001
Epoch 77/100
```

```
782/782 [==============================] - 23s 30ms/step - loss: 0.3600 - acc: 0.9686 - va
l_loss: 1.0978 - val_acc: 0.7938 - lr: 0.0010
Learning rate:  0.001
Epoch 78/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3587 - acc: 0.9687 - va
l_loss: 1.1778 - val_acc: 0.7927 - lr: 0.0010
Learning rate:  0.001
Epoch 79/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3589 - acc: 0.9685 - va
l_loss: 1.2689 - val_acc: 0.7860 - lr: 0.0010
Learning rate:  0.001
Epoch 80/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3504 - acc: 0.9712 - va
l_loss: 1.2231 - val_acc: 0.7858 - lr: 3.1623e-04
Learning rate:  0.001
Epoch 81/100
782/782 [==============================] - 23s 30ms/step - loss: 0.3614 - acc: 0.9659 - va
l_loss: 1.3517 - val_acc: 0.7699 - lr: 0.0010
Learning rate:  0.0001
Epoch 82/100
782/782 [==============================] - 23s 30ms/step - loss: 0.2983 - acc: 0.9903 - va
l_loss: 0.9125 - val_acc: 0.8431 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 83/100
782/782 [==============================] - 24s 30ms/step - loss: 0.2730 - acc: 0.9985 - va
l_loss: 0.9146 - val_acc: 0.8448 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 84/100
782/782 [==============================] - 23s 30ms/step - loss: 0.2624 - acc: 0.9995 - va
l_loss: 0.9346 - val_acc: 0.8470 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 85/100
782/782 [==============================] - 23s 30ms/step - loss: 0.2529 - acc: 0.9998 - va
l_loss: 0.9445 - val_acc: 0.8465 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 86/100
782/782 [==============================] - 23s 30ms/step - loss: 0.2421 - acc: 0.9998 - va
l_loss: 0.9683 - val_acc: 0.8461 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 87/100
782/782 [==============================] - 23s 30ms/step - loss: 0.2293 - acc: 0.9999 - va
l_loss: 0.9598 - val_acc: 0.8477 - lr: 3.1623e-05
Learning rate:  0.0001
Epoch 88/100
782/782 [==============================] - 23s 30ms/step - loss: 0.2159 - acc: 0.9998 - va
l_loss: 0.9930 - val_acc: 0.8431 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 89/100
782/782 [==============================] - 23s 30ms/step - loss: 0.2021 - acc: 0.9998 - va
l_loss: 1.0220 - val_acc: 0.8416 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 90/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1887 - acc: 0.9999 - va
l_loss: 1.0098 - val_acc: 0.8457 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 91/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1777 - acc: 0.9994 - va
l_loss: 1.0290 - val_acc: 0.8419 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 92/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1690 - acc: 0.9997 - va
l_loss: 1.0400 - val_acc: 0.8465 - lr: 3.1623e-05
Learning rate:  0.0001
Epoch 93/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1616 - acc: 0.9996 - va
l_loss: 1.0535 - val_acc: 0.8411 - lr: 1.0000e-04
```

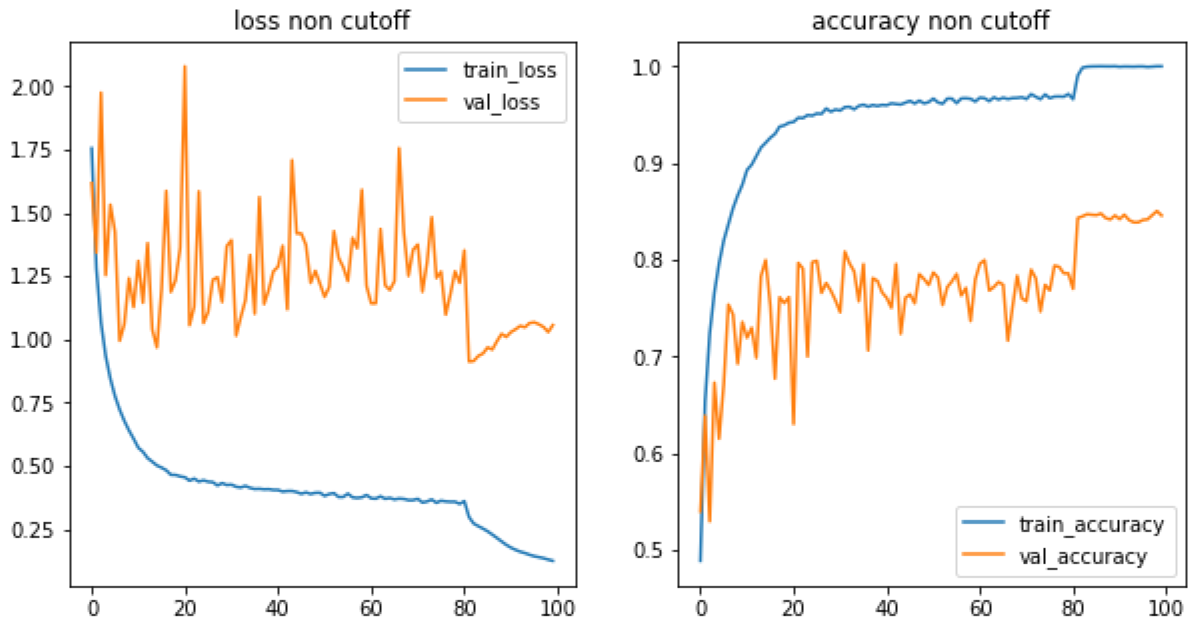
```
Learning rate:  0.0001
Epoch 94/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1562 - acc: 0.9995 - va
l_loss: 1.0478 - val_acc: 0.8387 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 95/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1495 - acc: 0.9997 - va
l_loss: 1.0645 - val_acc: 0.8387 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 96/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1438 - acc: 0.9997 - va
l_loss: 1.0677 - val_acc: 0.8411 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 97/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1400 - acc: 0.9993 - va
l_loss: 1.0598 - val_acc: 0.8417 - lr: 3.1623e-05
Learning rate:  0.0001
Epoch 98/100
782/782 [==============================] - 23s 29ms/step - loss: 0.1359 - acc: 0.9996 - va
l_loss: 1.0491 - val_acc: 0.8461 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 99/100
782/782 [==============================] - 23s 29ms/step - loss: 0.1306 - acc: 0.9999 - va
l_loss: 1.0289 - val_acc: 0.8503 - lr: 1.0000e-04
Learning rate:  0.0001
Epoch 100/100
782/782 [==============================] - 23s 30ms/step - loss: 0.1255 - acc: 0.9999 - va
l_loss: 1.0565 - val_acc: 0.8457 - lr: 1.0000e-04
```

In [89]:

```
epoch_vis = list(range(100))
train_loss_noncutoff = history_nocutoff.history['loss']
train_auc_noncutoff = history_nocutoff.history['acc']
test_loss_noncutoff = history_nocutoff.history['val_loss']
test_auc_noncutoff = history_nocutoff.history['val_acc']
fig, ax = plt.subplots(1,2,figsize = (10,5))
ax[0].plot(epoch_vis, train_loss_noncutoff, label = 'train_loss')
ax[1].plot(epoch_vis, train_auc_noncutoff, label = 'train_accuracy')
ax[0].plot(epoch_vis, test_loss_noncutoff, label = 'val_loss')
ax[1].plot(epoch_vis, test_auc_noncutoff, label = 'val_accuracy')
ax[0].legend()
ax[1].legend()
ax[0].set_title('loss non cutoff')
ax[1].set_title('accuracy non cutoff')
```

Out[89]:

```
Text(0.5, 1.0, 'accuracy non cutoff')
```



question 3

Next use cutout for data augmentation in Resnet-44 as in Hoffer et al. and train the model and use the same set-up in your experiments. Plot validation error vs number of epochs for different values of M (2,4,8,16) where M is the number of instances generated from an input sample after applying cutout M times effectively increasing the batch size to M ·B, where B is the original batch size (before applying cutout augmentation). You will obtain a figure similar to Figure 3(a) in the paper by Hoffer et al. Also compare the number of epochs and wallclock time to reach 92% accuracy for different values of M. Do not run any experiment for more than 100 epochs. If even after 100 epochs of training you did not achieve 92% then just report the accuracy you obtain and the corresponding wallclock time to train for 100 epochs. Remember to use the same hyperparameters for training as used with Resnet44 training in He et al (look at the third paragraph in Sec. 4.2 of He et al for the hyperparameter values). Before attempting this question it is advisable to read the paper by Hoffer et al. and especially Section 4.1.

In [4]:

```
def cut_off_train_data(M):
    train_x_cutoff = []
    train_y_cutoff = []
    for index, img in enumerate(X_train):
        for m in range(M):
            train_x_cutoff.append(apply_mask(img.copy()))
            train_y_cutoff.append(y_train[index])
    return np.array(train_x_cutoff),np.array(train_y_cutoff)

def train_a_cut_off_model(M):
    # def model
    model_cutoff = resnet_v1(input_shape=input_shape, depth=depth)
    model_cutoff.compile(loss='sparse_categorical_crossentropy',
                  optimizer=Adam(lr=lr_schedule(0)),
                  metrics=['acc'])
    # training_data
    train_with_m_x, train_with_m_y = cut_off_train_data(M)


    # trainmodel
    lr_scheduler = LearningRateScheduler(lr_schedule)

    lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
                                   cooldown=0,
                                   patience=5,
```

```python
                                      min_lr=0.5e-6)
    callbacks = [lr_reducer, lr_scheduler]

    start_time = time.time()
    history_cutoff = model_cutoff.fit(train_with_m_x, train_with_m_y,
                  batch_size=32*M,
                  epochs=20,
                  validation_data=(X_test, y_test),
                  shuffle=True,
                  callbacks=callbacks)
    end_time = time.time()

    total_time = end_time - start_time

    return history_cutoff, total_time
```

In [16]:

```python
history_cutoff_1, time_1 = train_a_cut_off_model(1)
```

```
Epoch 1/20
1563/1563 [==============================] - 42s 24ms/step - loss: 1.8269 - acc: 0.4535 -
val_loss: 1.7007 - val_acc: 0.5169 - lr: 0.0010
Epoch 2/20
1563/1563 [==============================] - 35s 22ms/step - loss: 1.3741 - acc: 0.6112 -
val_loss: 1.2922 - val_acc: 0.6451 - lr: 0.0010
Epoch 3/20
1563/1563 [==============================] - 37s 24ms/step - loss: 1.1725 - acc: 0.6821 -
val_loss: 1.3092 - val_acc: 0.6473 - lr: 0.0010
Epoch 4/20
1563/1563 [==============================] - 37s 24ms/step - loss: 1.0547 - acc: 0.7220 -
val_loss: 1.2850 - val_acc: 0.6672 - lr: 0.0010
Epoch 5/20
1563/1563 [==============================] - 37s 24ms/step - loss: 0.9668 - acc: 0.7533 -
val_loss: 1.1702 - val_acc: 0.7001 - lr: 0.0010
Epoch 6/20
1563/1563 [==============================] - 45s 29ms/step - loss: 0.9028 - acc: 0.7760 -
val_loss: 1.0027 - val_acc: 0.7463 - lr: 0.0010
Epoch 7/20
1563/1563 [==============================] - 41s 26ms/step - loss: 0.8560 - acc: 0.7933 -
val_loss: 1.1535 - val_acc: 0.7163 - lr: 0.0010
Epoch 8/20
1563/1563 [==============================] - 41s 26ms/step - loss: 0.8188 - acc: 0.8089 -
val_loss: 1.2956 - val_acc: 0.6923 - lr: 0.0010
Epoch 9/20
1563/1563 [==============================] - 41s 26ms/step - loss: 0.7856 - acc: 0.8215 -
val_loss: 1.2702 - val_acc: 0.6910 - lr: 0.0010
Epoch 10/20
1563/1563 [==============================] - 41s 27ms/step - loss: 0.7558 - acc: 0.8337 -
val_loss: 1.0899 - val_acc: 0.7396 - lr: 0.0010
Epoch 11/20
1563/1563 [==============================] - 42s 27ms/step - loss: 0.7263 - acc: 0.8456 -
val_loss: 0.9606 - val_acc: 0.7772 - lr: 0.0010
Epoch 12/20
1563/1563 [==============================] - 43s 28ms/step - loss: 0.7030 - acc: 0.8565 -
val_loss: 1.1112 - val_acc: 0.7594 - lr: 0.0010
Epoch 13/20
1563/1563 [==============================] - 42s 27ms/step - loss: 0.6846 - acc: 0.8629 -
val_loss: 0.9798 - val_acc: 0.7823 - lr: 0.0010
Epoch 14/20
1563/1563 [==============================] - 41s 26ms/step - loss: 0.6661 - acc: 0.8727 -
val_loss: 1.2200 - val_acc: 0.7187 - lr: 0.0010
Epoch 15/20
1563/1563 [==============================] - 42s 27ms/step - loss: 0.6502 - acc: 0.8804 -
val_loss: 1.2267 - val_acc: 0.7473 - lr: 0.0010
Epoch 16/20
```

```
1563/1563 [==============================] - 41s 27ms/step - loss: 0.6379 - acc: 0.8866 -
val_loss: 1.4111 - val_acc: 0.6998 - lr: 3.1623e-04
Epoch 17/20
1563/1563 [==============================] - 42s 27ms/step - loss: 0.6263 - acc: 0.8928 -
val_loss: 1.1516 - val_acc: 0.7725 - lr: 0.0010
Epoch 18/20
1563/1563 [==============================] - 41s 26ms/step - loss: 0.6170 - acc: 0.8971 -
val_loss: 0.9879 - val_acc: 0.8013 - lr: 0.0010
Epoch 19/20
1563/1563 [==============================] - 42s 27ms/step - loss: 0.6113 - acc: 0.9027 -
val_loss: 1.0601 - val_acc: 0.7809 - lr: 0.0010
Epoch 20/20
1563/1563 [==============================] - 41s 27ms/step - loss: 0.5992 - acc: 0.9078 -
val_loss: 1.0400 - val_acc: 0.7999 - lr: 0.0010
```

In [17]:

```
history_cutoff_2, time_2 = train_a_cut_off_model(2)
```

```
Epoch 1/20
1563/1563 [==============================] - 48s 28ms/step - loss: 1.6457 - acc: 0.5184 -
val_loss: 1.5763 - val_acc: 0.5705 - lr: 0.0010
Epoch 2/20
1563/1563 [==============================] - 49s 32ms/step - loss: 1.1203 - acc: 0.6999 -
val_loss: 1.0923 - val_acc: 0.7161 - lr: 0.0010
Epoch 3/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.9183 - acc: 0.7664 -
val_loss: 0.9938 - val_acc: 0.7513 - lr: 0.0010
Epoch 4/20
1563/1563 [==============================] - 50s 32ms/step - loss: 0.8104 - acc: 0.8026 -
val_loss: 0.9377 - val_acc: 0.7559 - lr: 0.0010
Epoch 5/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.7362 - acc: 0.8278 -
val_loss: 1.0307 - val_acc: 0.7403 - lr: 0.0010
Epoch 6/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.6810 - acc: 0.8489 -
val_loss: 1.2214 - val_acc: 0.7131 - lr: 0.0010
Epoch 7/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.6371 - acc: 0.8660 -
val_loss: 1.1379 - val_acc: 0.7460 - lr: 0.0010
Epoch 8/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.6024 - acc: 0.8811 -
val_loss: 1.1653 - val_acc: 0.7504 - lr: 0.0010
Epoch 9/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.5760 - acc: 0.8921 -
val_loss: 1.0267 - val_acc: 0.7672 - lr: 3.1623e-04
Epoch 10/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.5522 - acc: 0.9032 -
val_loss: 1.1807 - val_acc: 0.7574 - lr: 0.0010
Epoch 11/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.5324 - acc: 0.9104 -
val_loss: 1.0686 - val_acc: 0.7872 - lr: 0.0010
Epoch 12/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.5207 - acc: 0.9161 -
val_loss: 1.0318 - val_acc: 0.7925 - lr: 0.0010
Epoch 13/20
1563/1563 [==============================] - 51s 32ms/step - loss: 0.5120 - acc: 0.9209 -
val_loss: 1.0127 - val_acc: 0.8002 - lr: 0.0010
Epoch 14/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.4974 - acc: 0.9278 -
val_loss: 1.0421 - val_acc: 0.8007 - lr: 3.1623e-04
Epoch 15/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.4930 - acc: 0.9299 -
val_loss: 1.0787 - val_acc: 0.8112 - lr: 0.0010
Epoch 16/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.4877 - acc: 0.9327 -
val_loss: 1.3073 - val_acc: 0.7564 - lr: 0.0010
```

```
Epoch 17/20
1563/1563 [==============================] - 51s 33ms/step - loss: 0.4819 - acc: 0.9352 -
val_loss: 1.4237 - val_acc: 0.7357 - lr: 0.0010
Epoch 18/20
1563/1563 [==============================] - 51s 32ms/step - loss: 0.4760 - acc: 0.9384 -
val_loss: 1.2861 - val_acc: 0.7707 - lr: 0.0010
Epoch 19/20
1563/1563 [==============================] - 47s 30ms/step - loss: 0.4735 - acc: 0.9394 -
val_loss: 1.1247 - val_acc: 0.7836 - lr: 3.1623e-04
Epoch 20/20
1563/1563 [==============================] - 49s 31ms/step - loss: 0.4695 - acc: 0.9409 -
val_loss: 1.4280 - val_acc: 0.7497 - lr: 0.0010
```

In [18]:

```
history_cutoff_4, time_4 = train_a_cut_off_model(4)
```

```
Epoch 1/20
1563/1563 [==============================] - 75s 45ms/step - loss: 1.4920 - acc: 0.5760 -
val_loss: 1.4764 - val_acc: 0.5846 - lr: 0.0010
Epoch 2/20
1563/1563 [==============================] - 71s 45ms/step - loss: 0.9425 - acc: 0.7595 -
val_loss: 1.0704 - val_acc: 0.7329 - lr: 0.0010
Epoch 3/20
1563/1563 [==============================] - 70s 45ms/step - loss: 0.7437 - acc: 0.8248 -
val_loss: 1.1984 - val_acc: 0.7157 - lr: 0.0010
Epoch 4/20
1563/1563 [==============================] - 70s 45ms/step - loss: 0.6328 - acc: 0.8642 -
val_loss: 1.0041 - val_acc: 0.7695 - lr: 0.0010
Epoch 5/20
1563/1563 [==============================] - 69s 44ms/step - loss: 0.5617 - acc: 0.8907 -
val_loss: 1.1104 - val_acc: 0.7634 - lr: 0.0010
Epoch 6/20
1563/1563 [==============================] - 70s 45ms/step - loss: 0.5148 - acc: 0.9089 -
val_loss: 1.2633 - val_acc: 0.7558 - lr: 0.0010
Epoch 7/20
1563/1563 [==============================] - 70s 45ms/step - loss: 0.4854 - acc: 0.9222 -
val_loss: 1.2725 - val_acc: 0.7553 - lr: 0.0010
Epoch 8/20
1563/1563 [==============================] - 70s 45ms/step - loss: 0.4642 - acc: 0.9306 -
val_loss: 1.3441 - val_acc: 0.7467 - lr: 0.0010
Epoch 9/20
1563/1563 [==============================] - 71s 45ms/step - loss: 0.4485 - acc: 0.9370 -
val_loss: 1.3189 - val_acc: 0.7522 - lr: 3.1623e-04
Epoch 10/20
1563/1563 [==============================] - 71s 45ms/step - loss: 0.4375 - acc: 0.9417 -
val_loss: 1.3841 - val_acc: 0.7494 - lr: 0.0010
Epoch 11/20
1563/1563 [==============================] - 70s 45ms/step - loss: 0.4285 - acc: 0.9456 -
val_loss: 1.3027 - val_acc: 0.7695 - lr: 0.0010
Epoch 12/20
1563/1563 [==============================] - 70s 45ms/step - loss: 0.4196 - acc: 0.9492 -
val_loss: 1.3163 - val_acc: 0.7728 - lr: 0.0010
Epoch 13/20
1563/1563 [==============================] - 70s 45ms/step - loss: 0.4148 - acc: 0.9505 -
val_loss: 1.2604 - val_acc: 0.7522 - lr: 0.0010
Epoch 14/20
1563/1563 [==============================] - 71s 45ms/step - loss: 0.4100 - acc: 0.9520 -
val_loss: 1.4126 - val_acc: 0.7476 - lr: 3.1623e-04
Epoch 15/20
1563/1563 [==============================] - 65s 41ms/step - loss: 0.4069 - acc: 0.9530 -
val_loss: 1.3457 - val_acc: 0.7603 - lr: 0.0010
Epoch 16/20
1563/1563 [==============================] - 62s 40ms/step - loss: 0.4003 - acc: 0.9552 -
val_loss: 1.1831 - val_acc: 0.7898 - lr: 0.0010
Epoch 17/20
1563/1563 [==============================] - 62s 40ms/step - loss: 0.3980 - acc: 0.9555 -
```

```
val_loss: 1.2335 - val_acc: 0.7951 - lr: 0.0010
Epoch 18/20
1563/1563 [==============================] - 62s 40ms/step - loss: 0.3931 - acc: 0.9575 -
val_loss: 1.3593 - val_acc: 0.7608 - lr: 0.0010
Epoch 19/20
1563/1563 [==============================] - 62s 39ms/step - loss: 0.3885 - acc: 0.9585 -
val_loss: 1.1133 - val_acc: 0.8030 - lr: 3.1623e-04
Epoch 20/20
1563/1563 [==============================] - 63s 40ms/step - loss: 0.3868 - acc: 0.9584 -
val_loss: 1.2262 - val_acc: 0.7833 - lr: 0.0010
```

In [19]:

```
history_cutoff_8, time_8 = train_a_cut_off_model(8)
```

```
Epoch 1/20
1563/1563 [==============================] - 111s 67ms/step - loss: 1.3337 - acc: 0.6264 -
val_loss: 1.3204 - val_acc: 0.6608 - lr: 0.0010
Epoch 2/20
1563/1563 [==============================] - 105s 67ms/step - loss: 0.7546 - acc: 0.8194 -
val_loss: 1.2154 - val_acc: 0.7099 - lr: 0.0010
Epoch 3/20
1563/1563 [==============================] - 116s 74ms/step - loss: 0.5650 - acc: 0.8839 -
val_loss: 1.0629 - val_acc: 0.7585 - lr: 0.0010
Epoch 4/20
1563/1563 [==============================] - 122s 78ms/step - loss: 0.4715 - acc: 0.9172 -
val_loss: 0.9881 - val_acc: 0.7836 - lr: 0.0010
Epoch 5/20
1563/1563 [==============================] - 123s 78ms/step - loss: 0.4210 - acc: 0.9356 -
val_loss: 1.2083 - val_acc: 0.7541 - lr: 0.0010
Epoch 6/20
1563/1563 [==============================] - 123s 79ms/step - loss: 0.3939 - acc: 0.9455 -
val_loss: 2.0251 - val_acc: 0.6670 - lr: 0.0010
Epoch 7/20
1563/1563 [==============================] - 119s 76ms/step - loss: 0.3745 - acc: 0.9518 -
val_loss: 1.6866 - val_acc: 0.7049 - lr: 0.0010
Epoch 8/20
1563/1563 [==============================] - 117s 75ms/step - loss: 0.3621 - acc: 0.9554 -
val_loss: 1.1605 - val_acc: 0.7934 - lr: 0.0010
Epoch 9/20
1563/1563 [==============================] - 118s 76ms/step - loss: 0.3527 - acc: 0.9582 -
val_loss: 1.3453 - val_acc: 0.7659 - lr: 3.1623e-04
Epoch 10/20
1563/1563 [==============================] - 120s 77ms/step - loss: 0.3432 - acc: 0.9608 -
val_loss: 1.3324 - val_acc: 0.7485 - lr: 0.0010
Epoch 11/20
1563/1563 [==============================] - 117s 75ms/step - loss: 0.3363 - acc: 0.9623 -
val_loss: 1.2246 - val_acc: 0.7820 - lr: 0.0010
Epoch 12/20
1563/1563 [==============================] - 116s 74ms/step - loss: 0.3281 - acc: 0.9643 -
val_loss: 1.2731 - val_acc: 0.7820 - lr: 0.0010
Epoch 13/20
1563/1563 [==============================] - 116s 74ms/step - loss: 0.3243 - acc: 0.9651 -
val_loss: 1.1112 - val_acc: 0.7969 - lr: 0.0010
Epoch 14/20
1563/1563 [==============================] - 115s 74ms/step - loss: 0.3193 - acc: 0.9660 -
val_loss: 1.3165 - val_acc: 0.7864 - lr: 3.1623e-04
Epoch 15/20
1563/1563 [==============================] - 114s 73ms/step - loss: 0.3141 - acc: 0.9671 -
val_loss: 1.3896 - val_acc: 0.7680 - lr: 0.0010
Epoch 16/20
1563/1563 [==============================] - 114s 73ms/step - loss: 0.3106 - acc: 0.9677 -
val_loss: 1.0102 - val_acc: 0.8130 - lr: 0.0010
Epoch 17/20
1563/1563 [==============================] - 115s 74ms/step - loss: 0.3061 - acc: 0.9685 -
val_loss: 1.2303 - val_acc: 0.7818 - lr: 0.0010
Epoch 18/20
```

```
1563/1563 [==============================] - 117s 75ms/step - loss: 0.3022 - acc: 0.9692 - val_loss: 1.4652 - val_acc: 0.7730 - lr: 0.0010
Epoch 19/20
1563/1563 [==============================] - 116s 74ms/step - loss: 0.2995 - acc: 0.9696 - val_loss: 1.0964 - val_acc: 0.8116 - lr: 3.1623e-04
Epoch 20/20
1563/1563 [==============================] - 116s 74ms/step - loss: 0.2953 - acc: 0.9707 - val_loss: 1.5218 - val_acc: 0.7562 - lr: 0.0010
```

In [7]:

```
history_cutoff_16, time_16 = train_a_cut_off_model(16)
```

```
Epoch 1/20
1563/1563 [==============================] - 191s 117ms/step - loss: 1.1494 - acc: 0.6959 - val_loss: 1.6258 - val_acc: 0.6408 - lr: 0.0010
Epoch 2/20
1563/1563 [==============================] - 182s 117ms/step - loss: 0.5523 - acc: 0.8954 - val_loss: 1.3549 - val_acc: 0.7339 - lr: 0.0010
Epoch 3/20
1563/1563 [==============================] - 183s 117ms/step - loss: 0.4096 - acc: 0.9425 - val_loss: 1.4744 - val_acc: 0.7359 - lr: 0.0010
Epoch 4/20
1563/1563 [==============================] - 182s 117ms/step - loss: 0.3614 - acc: 0.9563 - val_loss: 2.2739 - val_acc: 0.6582 - lr: 0.0010
Epoch 5/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.3370 - acc: 0.9627 - val_loss: 1.7049 - val_acc: 0.7021 - lr: 0.0010
Epoch 6/20
1563/1563 [==============================] - 180s 115ms/step - loss: 0.3196 - acc: 0.9661 - val_loss: 1.8806 - val_acc: 0.7054 - lr: 0.0010
Epoch 7/20
1563/1563 [==============================] - 181s 116ms/step - loss: 0.3065 - acc: 0.9689 - val_loss: 2.0385 - val_acc: 0.7102 - lr: 3.1623e-04
Epoch 8/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2973 - acc: 0.9703 - val_loss: 1.2951 - val_acc: 0.7706 - lr: 0.0010
Epoch 9/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2879 - acc: 0.9724 - val_loss: 1.4359 - val_acc: 0.7581 - lr: 0.0010
Epoch 10/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2813 - acc: 0.9731 - val_loss: 1.2327 - val_acc: 0.7777 - lr: 0.0010
Epoch 11/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2747 - acc: 0.9743 - val_loss: 1.5153 - val_acc: 0.7524 - lr: 0.0010
Epoch 12/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2691 - acc: 0.9749 - val_loss: 1.7484 - val_acc: 0.7442 - lr: 0.0010
Epoch 13/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2641 - acc: 0.9759 - val_loss: 1.5708 - val_acc: 0.7548 - lr: 0.0010
Epoch 14/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2596 - acc: 0.9764 - val_loss: 1.4176 - val_acc: 0.7664 - lr: 0.0010
Epoch 15/20
1563/1563 [==============================] - 182s 117ms/step - loss: 0.2543 - acc: 0.9771 - val_loss: 1.2585 - val_acc: 0.7820 - lr: 3.1623e-04
Epoch 16/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2500 - acc: 0.9778 - val_loss: 1.2565 - val_acc: 0.7808 - lr: 0.0010
Epoch 17/20
1563/1563 [==============================] - 181s 116ms/step - loss: 0.2463 - acc: 0.9782 - val_loss: 1.5072 - val_acc: 0.7766 - lr: 0.0010
Epoch 18/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2419 - acc: 0.9791 - val_loss: 1.4070 - val_acc: 0.7799 - lr: 0.0010
```

```
Epoch 19/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2394 - acc: 0.9792
- val_loss: 1.0858 - val_acc: 0.7965 - lr: 0.0010
Epoch 20/20
1563/1563 [==============================] - 182s 116ms/step - loss: 0.2370 - acc: 0.9793
- val_loss: 1.1449 - val_acc: 0.8045 - lr: 0.0010
```

In [8]:

```
import pickle
#history_list = [history_cutoff_1.history,history_cutoff_2.history,history_cutoff_4.histo
#time_list = [time_1,time_2,time_4,time_8]
#
#with open('Q4_trained_data.pkl','wb') as handle:
#    pickle.dump((history_list,time_list),handle ,protocol = pickle.HIGHEST_PROTOCOL)

with open('Q4_trained_data.pkl','rb') as handle:
    previous_result = pickle.load(handle)
```

In [14]:

```
history_list_final = [previous_result[0][0], previous_result[0][1], previous_result[0][2],
                      previous_result[0][3], history_cutoff_16.history]
time_list_final = [previous_result[1][0],previous_result[1][1],previous_result[1][2],previ
with open('Q4_final_result_data.pkl','wb') as handle:
    pickle.dump((history_list_final,time_list_final),handle ,protocol = pickle.HIGHEST_PRO
```

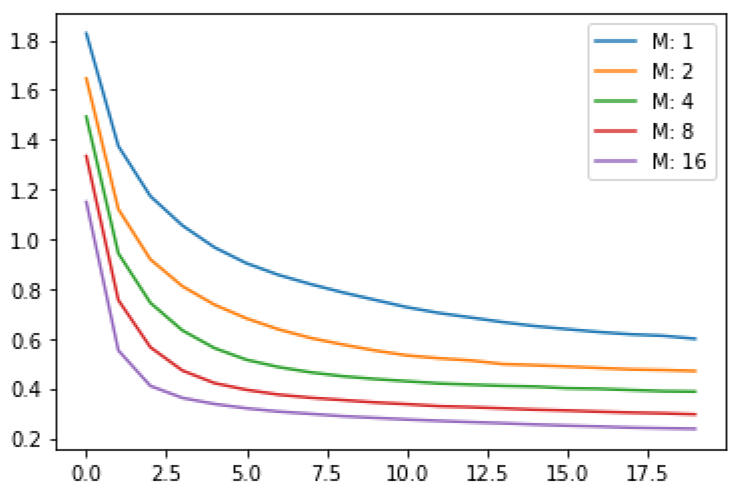
In [15]:

```
with open('Q4_final_result_data.pkl','rb') as handle:
    final_result = pickle.load(handle)
```

In [29]:

```
for i in range(5):
    epoch_vis = list(range(20))
    plt.plot(epoch_vis,final_result[0][i]['loss'], label = 'M: '+str(2**i))
plt.legend()
```

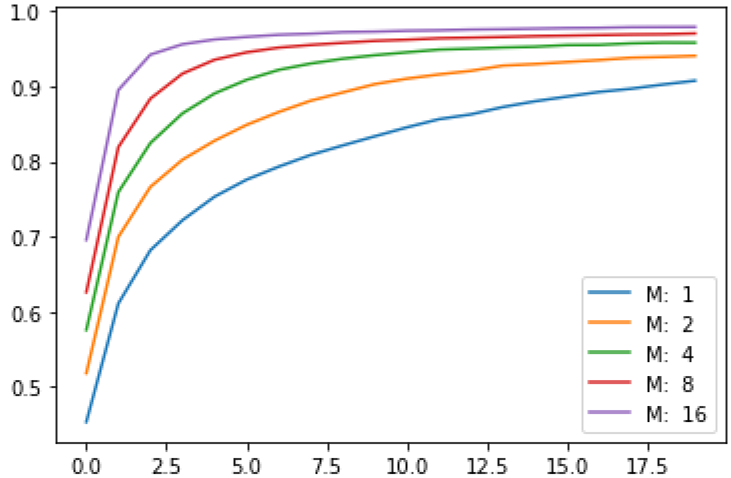
Out[29]: <matplotlib.legend.Legend at 0x1b17caa2160>



In [31]:

```
for i in range(5):
    epoch_vis = list(range(20))
    plt.plot(epoch_vis,final_result[0][i]['acc'], label = 'M:  ' +str(2**i))
plt.legend()
```

Out[31]: <matplotlib.legend.Legend at 0x1b15aff2070>



In [32]:

```
# training_time
final_result[1]
```

Out[32]:

```
[815.8369281291962,
 1008.5819070339203,
 1361.3079934120178,
 2330.293242454529,
 3645.6672925949097]
```

- due to time limitation, I only ran 20 epochs here