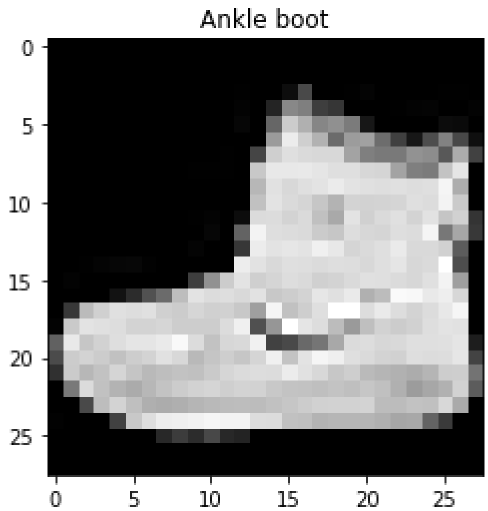
In [1]:
```
import tensorflow as tf
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras import models, layers
import tensorflow.keras as keras
from tensorflow.keras.layers import BatchNormalization, LayerNormalizati
on
import tensorflow as tf
from tensorflow.keras.layers import Dropout
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.utils import Progbar
from tensorflow.python.eager import backprop
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import fashion_mnist
from collections import Counter
from matplotlib import pyplot
import numpy as np
if tf.test.gpu_device_name():
    print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))
else:
    print("Please install GPU version of TF")
```

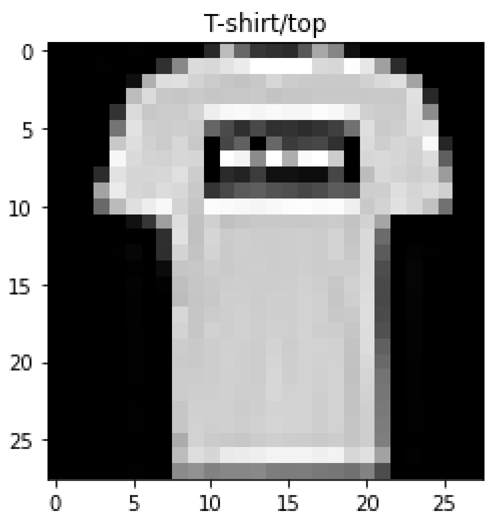
```
Default GPU Device: /device:GPU:0
```

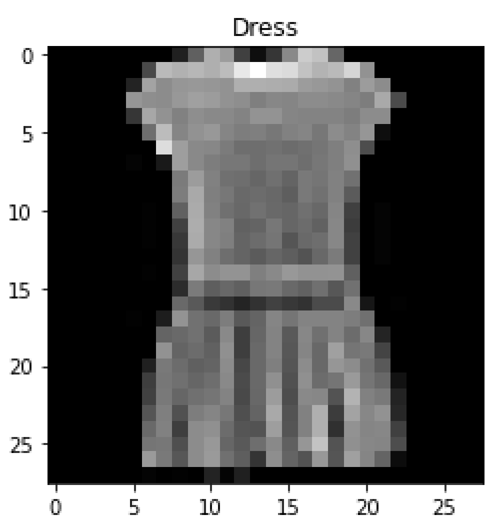
Question 1:

In [2]:
```
import matplotlib
# load dataset
(x_train, y_train), (x_test,y_test) = fashion_mnist.load_data()
# summarize loaded dataset
print('Train: X=%s, y=%s' % (x_train.shape, y_train.shape))
print('Test: X=%s, y=%s' % (x_test.shape, y_test.shape))
# plot first few images
for i in [0,1,3]:
# plot raw pixel data
    plt.imshow(x_train[i], cmap=pyplot.get_cmap('gray'))
    if (i==0):
        title = 'Ankle boot'
    elif (i==1):
        title = 'T-shirt/top'
    else :
        title = 'Dress'
    plt.title(title)
# show the figure
    plt.show()
```

```
Train: X=(60000, 28, 28), y=(60000,)
Test: X=(10000, 28, 28), y=(10000,)
```







In [3]:
```
y_train[0:5]
```

Out[3]:
```
array([9, 0, 0, 3, 0], dtype=uint8)
```

In [4]:
```
print('Train : ' + str(Counter(y_train)))
print('Test : ' + str(Counter(y_test)))
```

```
Train : Counter({9: 6000, 0: 6000, 3: 6000, 2: 6000, 7: 6000, 5: 6000,
1: 6000, 6: 6000, 4: 6000, 8: 6000})
Test : Counter({9: 1000, 2: 1000, 1: 1000, 6: 1000, 4: 1000, 5: 1000,
7: 1000, 3: 1000, 8: 1000, 0: 1000})
```

Fashion-MNIST is a dataset of Zalando's article images—consisting with 60,000 training examples and 10,000 test examples. Each example is a 28x28 grayscale image associated to one of the 10 classes.

Each image is 28 pixels in height and 28 pixels in width, with a value between 0 and 255.

0 T-shirt/top : 6000 training examples / 1000 test examples

1 Trouser : 6000 training examples / 1000 test examples

2 Pullover : 6000 training examples / 1000 test examples

3 Dress : 6000 training examples / 1000 test examples

4 Coat : 6000 training examples / 1000 test examples

5 Sandal : 6000 training examples / 1000 test examples

6 Shirt : 6000 training examples / 1000 test examples

7 Sneaker : 6000 training examples / 1000 test examples

8 Bag : 6000 training examples / 1000 test examples

9 Ankle boot : 6000 training examples / 1000 test examples

Question 2:

In [5]:
```
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Set numeric type to float32 from uint8
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# Normalize value to [0, 1]
x_train /= 255
x_test /= 255

# Transform lables to one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

# Reshape the dataset into 4D array
x_train = x_train.reshape(x_train.shape[0], 28,28,1)
x_test = x_test.reshape(x_test.shape[0], 28,28,1)
```

Define model (LeNet5)

In [6]:
```
def create_model():
    model = Sequential()

    # C1 Convolutional Layer 'tanh'
    model.add(BatchNormalization(input_shape=(28,28,1)))
    model.add(layers.Conv2D(6, kernel_size=(5, 5), strides=(1, 1), activ
ation='tanh', input_shape=(28,28,1), padding='same'))
    model.add(BatchNormalization())
    # S2 Pooling Layer
    model.add(layers.AveragePooling2D(pool_size=(2, 2), strides=(1, 1),
padding='valid'))
    model.add(BatchNormalization())
    # C3 Convolutional Layer
    model.add(layers.Conv2D(16, kernel_size=(5, 5), strides=(1, 1), acti
vation='tanh', padding='valid'))
    model.add(BatchNormalization())
    # S4 Pooling Layer
    model.add(layers.AveragePooling2D(pool_size=(2, 2), strides=(2, 2),
padding='valid'))
    model.add(BatchNormalization())
    # C5 Fully Connected Convolutional Layer
    model.add(layers.Conv2D(120, kernel_size=(5, 5), strides=(1, 1), act
ivation='tanh', padding='valid'))
    model.add(BatchNormalization())
    #Flatten the CNN output so that we can connect it with fully connect
ed layers
    model.add(layers.Flatten())

    # FC6 Fully Connected Layer
    model.add(layers.Dense(84, activation='tanh'))
    model.add(BatchNormalization())
    #Output Layer with softmax activation
    model.add(layers.Dense(10, activation='softmax'))
    return model
```

In [7]:
```
model = create_model()
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
batch_normalization (BatchNo (None, 28, 28, 1)         4         
_________________________________________________________________
conv2d (Conv2D)              (None, 28, 28, 6)         156       
_________________________________________________________________
batch_normalization_1 (Batch (None, 28, 28, 6)         24        
_________________________________________________________________
average_pooling2d (AveragePo (None, 27, 27, 6)         0         
_________________________________________________________________
batch_normalization_2 (Batch (None, 27, 27, 6)         24        
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 23, 23, 16)        2416      
_________________________________________________________________
batch_normalization_3 (Batch (None, 23, 23, 16)        64        
_________________________________________________________________
average_pooling2d_1 (Average (None, 11, 11, 16)        0         
_________________________________________________________________
batch_normalization_4 (Batch (None, 11, 11, 16)        64        
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 120)         48120     
_________________________________________________________________
batch_normalization_5 (Batch (None, 7, 7, 120)         480       
_________________________________________________________________
flatten (Flatten)            (None, 5880)              0         
_________________________________________________________________
dense (Dense)                (None, 84)                494004    
_________________________________________________________________
batch_normalization_6 (Batch (None, 84)                336       
_________________________________________________________________
dense_1 (Dense)              (None, 10)                850       
=================================================================
Total params: 546,542
Trainable params: 546,044
Non-trainable params: 498
_________________________________________________________________
```

Train for different learning rate on 5 epochs, batch size = 64

In [55]:
```
num_epochs = 5
batch_size = 64
hist = []


for lr in [10**-i for i in range(-1,10)]:
    print(lr)
    opt = SGD(lr=lr)
    model = create_model()
    model.compile(loss=keras.losses.categorical_crossentropy, optimizer=
opt, metrics=['accuracy'])
    hist.append(model.fit(x=x_train,y=y_train, epochs=num_epochs,
                          batch_size=batch_size, validation_data=(x_test
, y_test), verbose=1))
```

```
10
Epoch 1/5
938/938 [==============================] - 12s 12ms/step - loss: nan -
accuracy: 0.1001 - val_loss: nan - val_accuracy: 0.1000
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: nan -
accuracy: 0.1000 - val_loss: nan - val_accuracy: 0.1000
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: nan -
accuracy: 0.1000 - val_loss: nan - val_accuracy: 0.1000
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: nan -
accuracy: 0.1000 - val_loss: nan - val_accuracy: 0.1000
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: nan -
accuracy: 0.1000 - val_loss: nan - val_accuracy: 0.1000
1
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 2.3056
- accuracy: 0.1087 - val_loss: 14.3559 - val_accuracy: 0.1000
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 2.3074
- accuracy: 0.0983 - val_loss: 8.8344 - val_accuracy: 0.1000
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 2.3065
- accuracy: 0.1001 - val_loss: 3.4229 - val_accuracy: 0.1000
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 2.3064
- accuracy: 0.0996 - val_loss: 3.1536 - val_accuracy: 0.1000
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: 2.3065
- accuracy: 0.0979 - val_loss: 2.4587 - val_accuracy: 0.1000
0.1
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 0.4995
- accuracy: 0.8183 - val_loss: 0.4212 - val_accuracy: 0.8475
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 0.3572
- accuracy: 0.8689 - val_loss: 0.4743 - val_accuracy: 0.8246
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 0.3125
- accuracy: 0.8852 - val_loss: 0.3787 - val_accuracy: 0.8678
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 0.2855
- accuracy: 0.8953 - val_loss: 0.3366 - val_accuracy: 0.8788
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: 0.2629
- accuracy: 0.9030 - val_loss: 0.2906 - val_accuracy: 0.8965
0.01
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 0.4882
- accuracy: 0.8275 - val_loss: 0.4396 - val_accuracy: 0.8450
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 0.3657
- accuracy: 0.8719 - val_loss: 0.3738 - val_accuracy: 0.8689
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 0.3222
```

```
- accuracy: 0.8838 - val_loss: 0.3550 - val_accuracy: 0.8738
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 0.2921
- accuracy: 0.8953 - val_loss: 0.3316 - val_accuracy: 0.8797
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: 0.2698
- accuracy: 0.9040 - val_loss: 0.3209 - val_accuracy: 0.8849
0.001
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 0.6184
- accuracy: 0.7884 - val_loss: 0.5206 - val_accuracy: 0.8206
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 0.4629
- accuracy: 0.8415 - val_loss: 0.4657 - val_accuracy: 0.8378
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 0.4238
- accuracy: 0.8536 - val_loss: 0.4379 - val_accuracy: 0.8494
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 0.4000
- accuracy: 0.8617 - val_loss: 0.4168 - val_accuracy: 0.8566
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: 0.3823
- accuracy: 0.8675 - val_loss: 0.4550 - val_accuracy: 0.8411
0.0001
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 0.9877
- accuracy: 0.6682 - val_loss: 0.7134 - val_accuracy: 0.7572
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 0.6605
- accuracy: 0.7745 - val_loss: 0.6332 - val_accuracy: 0.7803
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 0.5996
- accuracy: 0.7936 - val_loss: 0.5946 - val_accuracy: 0.7917
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 0.5637
- accuracy: 0.8072 - val_loss: 0.5707 - val_accuracy: 0.8015
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: 0.5392
- accuracy: 0.8145 - val_loss: 0.5502 - val_accuracy: 0.8092
1e-05
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 1.9523
- accuracy: 0.3638 - val_loss: 1.3800 - val_accuracy: 0.5548
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 1.2167
- accuracy: 0.6108 - val_loss: 1.1004 - val_accuracy: 0.6485
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 1.0329
- accuracy: 0.6698 - val_loss: 0.9839 - val_accuracy: 0.6824
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 0.9441
- accuracy: 0.6937 - val_loss: 0.9165 - val_accuracy: 0.6970
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: 0.8887
- accuracy: 0.7082 - val_loss: 0.8720 - val_accuracy: 0.7069
1e-06
Epoch 1/5
```

```
938/938 [==============================] - 11s 12ms/step - loss: 2.8242
- accuracy: 0.1925 - val_loss: 2.6635 - val_accuracy: 0.2085
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 2.5429
- accuracy: 0.2243 - val_loss: 2.4170 - val_accuracy: 0.2369
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 2.3202
- accuracy: 0.2558 - val_loss: 2.2188 - val_accuracy: 0.2745
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 2.1457
- accuracy: 0.2918 - val_loss: 2.0641 - val_accuracy: 0.3140
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: 2.0063
- accuracy: 0.3301 - val_loss: 1.9361 - val_accuracy: 0.3484
1e-07
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 2.9628
- accuracy: 0.1073 - val_loss: 2.9461 - val_accuracy: 0.1098
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 2.9158
- accuracy: 0.1152 - val_loss: 2.9036 - val_accuracy: 0.1182
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 2.8710
- accuracy: 0.1220 - val_loss: 2.8592 - val_accuracy: 0.1253
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 2.8295
- accuracy: 0.1285 - val_loss: 2.8164 - val_accuracy: 0.1329
Epoch 5/5
938/938 [==============================] - 11s 12ms/step - loss: 2.7868
- accuracy: 0.1362 - val_loss: 2.7725 - val_accuracy: 0.1400
1e-08
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 2.9430
- accuracy: 0.1029 - val_loss: 2.9424 - val_accuracy: 0.0972
Epoch 2/5
938/938 [==============================] - 11s 11ms/step - loss: 2.9397
- accuracy: 0.1017 - val_loss: 2.9397 - val_accuracy: 0.1000
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 2.9376
- accuracy: 0.1014 - val_loss: 2.9391 - val_accuracy: 0.0990
Epoch 4/5
938/938 [==============================] - 11s 11ms/step - loss: 2.9353
- accuracy: 0.1034 - val_loss: 2.9335 - val_accuracy: 0.0990
Epoch 5/5
938/938 [==============================] - 11s 11ms/step - loss: 2.9314
- accuracy: 0.1021 - val_loss: 2.9321 - val_accuracy: 0.0996
1e-09
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 3.4312
- accuracy: 0.0559 - val_loss: 3.4061 - val_accuracy: 0.0609
Epoch 2/5
938/938 [==============================] - 11s 11ms/step - loss: 3.4297
- accuracy: 0.0571 - val_loss: 3.4039 - val_accuracy: 0.0617
Epoch 3/5
938/938 [==============================] - 11s 11ms/step - loss: 3.4308
- accuracy: 0.0568 - val_loss: 3.4055 - val_accuracy: 0.0605
Epoch 4/5
```

```
938/938 [==============================] - 11s 11ms/step - loss: 3.4319
- accuracy: 0.0565 - val_loss: 3.4057 - val_accuracy: 0.0603
Epoch 5/5
938/938 [==============================] - 11s 11ms/step - loss: 3.4313
- accuracy: 0.0566 - val_loss: 3.4059 - val_accuracy: 0.0610
```

In [56]:
```
training_loss = []
training_accuracy = []
learning_rates = [10**-i for i in range(-1,10)]
for i in range(len(learning_rates)):
    training_loss.append(hist[i].history['loss'][4])
    training_accuracy.append(hist[i].history['accuracy'][4])
```

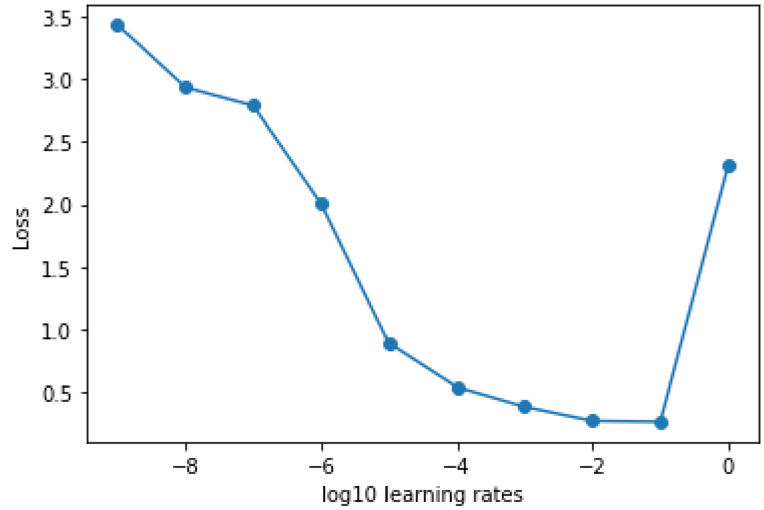
In [57]:
```
training_loss
```

Out[57]:
```
[nan,
 2.306472063064575,
 0.2628767192363739,
 0.26983168721199036,
 0.38227328658103943,
 0.5391852259635925,
 0.8886654376983643,
 2.006256341934204,
 2.7867674827575684,
 2.931445837020874,
 3.4312546253204346]
```

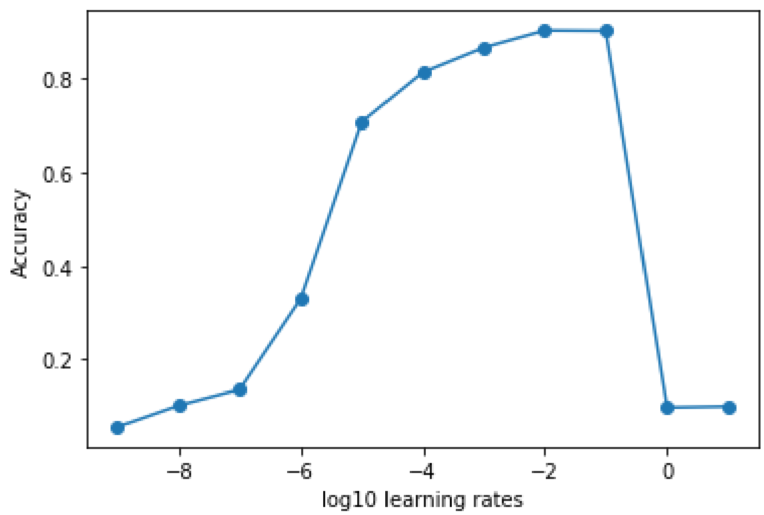
In [58]:
```
training_accuracy
```

Out[58]:
```
[0.10000000149011612,
 0.09788333624601364,
 0.902999997138977,
 0.9039666652679443,
 0.8674666881561279,
 0.8144999742507935,
 0.7082499861717224,
 0.33008334040641785,
 0.13615000247955322,
 0.10209999978542328,
 0.05661666765809059]
```

In [59]:
```
plt.plot(np.log10(learning_rates),training_loss,marker='o')
plt.xlabel('log10 learning rates')
plt.ylabel('Loss')
plt.show()
```



In [60]:
```
plt.plot(np.log10(learning_rates),training_accuracy,marker='o')
plt.xlabel('log10 learning rates')
plt.ylabel('Accuracy')
plt.show()
```



We can identify $lr_{min} = 1e-4$ and $lr_{max} = 1e-3$

Question 3:

In [53]:
```
from tensorflow.keras.callbacks import *
from tensorflow.keras import backend as K
import numpy as np

class CyclicLR(Callback):
    """This callback implements a cyclical learning rate policy (CLR).
    The method cycles the learning rate between two boundaries with
    some constant frequency, as detailed in this paper (https://arxiv.or
g/abs/1506.01186).
    The amplitude of the cycle can be scaled on a per-iteration or
    per-cycle basis.
    This class has three built-in policies, as put forth in the paper.
    "triangular":
        A basic triangular cycle w/ no amplitude scaling.
    "triangular2":
        A basic triangular cycle that scales initial amplitude by half e
ach cycle.
    "exp_range":
        A cycle that scales initial amplitude by gamma**(cycle iteration
s) at each
        cycle iteration.
    For more detail, please see paper.

    # Example
        ```python
            clr = CyclicLR(base_lr=0.001, max_lr=0.006,
                                step_size=2000., mode='triangular')
            model.fit(X_train, Y_train, callbacks=[clr])
        ```

    Class also supports custom scaling functions:
        ```python
            clr_fn = lambda x: 0.5*(1+np.sin(x*np.pi/2.))
            clr = CyclicLR(base_lr=0.001, max_lr=0.006,
                                step_size=2000., scale_fn=clr_fn,
                                scale_mode='cycle')
            model.fit(X_train, Y_train, callbacks=[clr])
        ```
    # Arguments
        base_lr: initial learning rate which is the
            lower boundary in the cycle.
        max_lr: upper boundary in the cycle. Functionally,
            it defines the cycle amplitude (max_lr - base_lr).
            The lr at any cycle is the sum of base_lr
            and some scaling of the amplitude; therefore
            max_lr may not actually be reached depending on
            scaling function.
        step_size: number of training iterations per
            half cycle. Authors suggest setting step_size
            2-8 x training iterations in epoch.
        mode: one of {triangular, triangular2, exp_range}.
            Default 'triangular'.
            Values correspond to policies detailed above.
            If scale_fn is not None, this argument is ignored.
        gamma: constant in 'exp_range' scaling function:
            gamma**(cycle iterations)
```

```
        scale_fn: Custom scaling policy defined by a single
            argument lambda function, where
            0 <= scale_fn(x) <= 1 for all x >= 0.
            mode paramater is ignored
        scale_mode: {'cycle', 'iterations'}.
            Defines whether scale_fn is evaluated on
            cycle number or cycle iterations (training
            iterations since start of cycle). Default is 'cycle'.
    """

    def __init__(self, base_lr=0.001, max_lr=0.006, step_size=2000., mod
e='triangular',
                 gamma=1., scale_fn=None, scale_mode='cycle'):
        super(CyclicLR, self).__init__()

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

        self._reset()

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

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

```
0, (1-x))*self.scale_fn(cycle)
        else:
            return self.base_lr + (self.max_lr-self.base_lr)*np.maximum(
0, (1-x))*self.scale_fn(self.clr_iterations)

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

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

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

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

        self.history.setdefault('lr', []).append(K.get_value(self.model.
optimizer.lr))
        self.history.setdefault('iterations', []).append(self.trn_iterat
ions)

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

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

In [54]:
```
lr_min=1e-4
lr_max=1e-3
clr_triangular = CyclicLR(mode='exp_range', gamma=0.99994,base_lr=lr_min
, max_lr=lr_max)

num_epochs = 20
model = create_model()
model.compile(loss=keras.losses.categorical_crossentropy, metrics=['accu
racy'])
model.fit(x=x_train,y=y_train, epochs=num_epochs,
                          batch_size=batch_size, validation_data=(x_test
, y_test), verbose=1,callbacks=[clr_triangular])
```

```
Epoch 1/20
938/938 [==============================] - 15s 16ms/step - loss: 0.5103
- accuracy: 0.8198 - val_loss: 0.5144 - val_accuracy: 0.8168
Epoch 2/20
938/938 [==============================] - 15s 16ms/step - loss: 0.4284
- accuracy: 0.8482 - val_loss: 0.4588 - val_accuracy: 0.8415
Epoch 3/20
938/938 [==============================] - 14s 15ms/step - loss: 0.3855
- accuracy: 0.8632 - val_loss: 0.3783 - val_accuracy: 0.8694
Epoch 4/20
938/938 [==============================] - 14s 15ms/step - loss: 0.3175
- accuracy: 0.8865 - val_loss: 0.3321 - val_accuracy: 0.8797
Epoch 5/20
938/938 [==============================] - 14s 15ms/step - loss: 0.2734
- accuracy: 0.9003 - val_loss: 0.3423 - val_accuracy: 0.8772
Epoch 6/20
938/938 [==============================] - 14s 15ms/step - loss: 0.2850
- accuracy: 0.8971 - val_loss: 0.3513 - val_accuracy: 0.8727
Epoch 7/20
938/938 [==============================] - 14s 15ms/step - loss: 0.2941
- accuracy: 0.8952 - val_loss: 0.3308 - val_accuracy: 0.8872
Epoch 8/20
938/938 [==============================] - 14s 15ms/step - loss: 0.2586
- accuracy: 0.9073 - val_loss: 0.3000 - val_accuracy: 0.8957
Epoch 9/20
938/938 [==============================] - 14s 15ms/step - loss: 0.2204
- accuracy: 0.9204 - val_loss: 0.2828 - val_accuracy: 0.9009
Epoch 10/20
938/938 [==============================] - 14s 15ms/step - loss: 0.2164
- accuracy: 0.9221 - val_loss: 0.3004 - val_accuracy: 0.8945
Epoch 11/20
938/938 [==============================] - 14s 15ms/step - loss: 0.2346
- accuracy: 0.9152 - val_loss: 0.3155 - val_accuracy: 0.8901
Epoch 12/20
938/938 [==============================] - 14s 15ms/step - loss: 0.2161
- accuracy: 0.9226 - val_loss: 0.2823 - val_accuracy: 0.9030
Epoch 13/20
938/938 [==============================] - 14s 15ms/step - loss: 0.1811
- accuracy: 0.9357 - val_loss: 0.2736 - val_accuracy: 0.9047
Epoch 14/20
938/938 [==============================] - 14s 15ms/step - loss: 0.1673
- accuracy: 0.9406 - val_loss: 0.2942 - val_accuracy: 0.8976
Epoch 15/20
938/938 [==============================] - 14s 15ms/step - loss: 0.1783
- accuracy: 0.9360 - val_loss: 0.2874 - val_accuracy: 0.9022
Epoch 16/20
938/938 [==============================] - 14s 15ms/step - loss: 0.1715
- accuracy: 0.9382 - val_loss: 0.2754 - val_accuracy: 0.9054
Epoch 17/20
938/938 [==============================] - 14s 15ms/step - loss: 0.1394
- accuracy: 0.9517 - val_loss: 0.2722 - val_accuracy: 0.9102
Epoch 18/20
938/938 [==============================] - 14s 15ms/step - loss: 0.1198
- accuracy: 0.9603 - val_loss: 0.2769 - val_accuracy: 0.9094
Epoch 19/20
938/938 [==============================] - 14s 15ms/step - loss: 0.1265
- accuracy: 0.9560 - val_loss: 0.2948 - val_accuracy: 0.9002
```

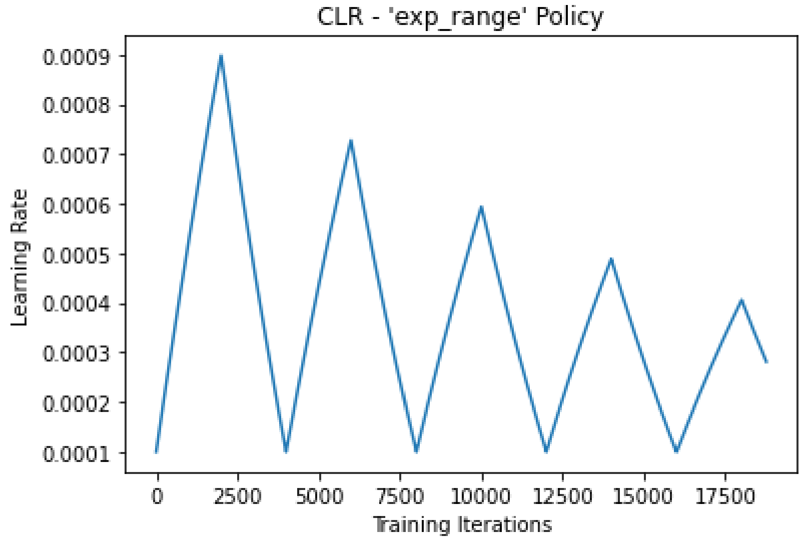
```
Epoch 20/20
938/938 [==============================] - 14s 15ms/step - loss: 0.1315
- accuracy: 0.9539 - val_loss: 0.2928 - val_accuracy: 0.9058
```

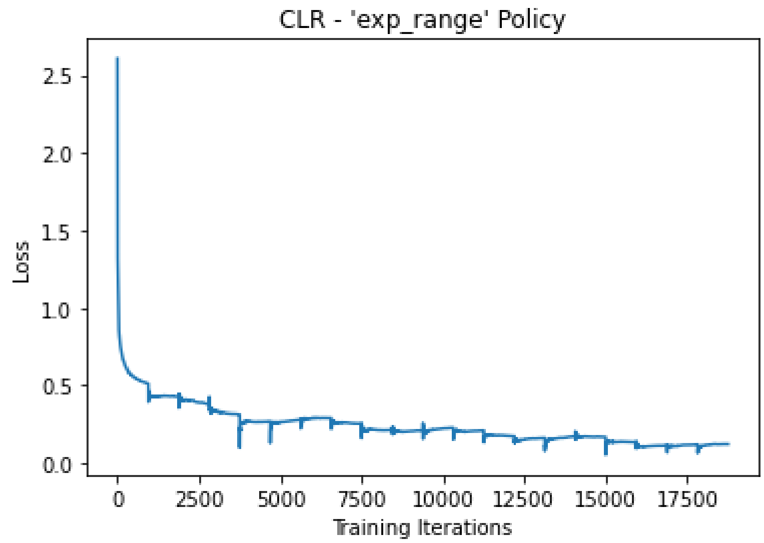
Out[54]:
```
<tensorflow.python.keras.callbacks.History at 0x7f768e4625d0>
```

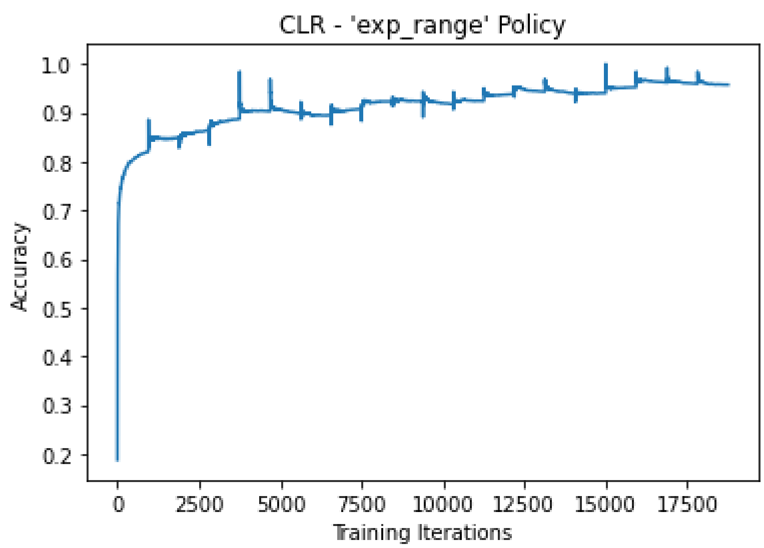
In [21]:
```
plt.xlabel('Training Iterations')
plt.ylabel('Learning Rate')
plt.title("CLR - 'exp_range' Policy")
plt.plot(clr_triangular.history['iterations'], clr_triangular.history['l
r'])
plt.show()

plt.xlabel('Training Iterations')
plt.ylabel('Loss')
plt.title("CLR - 'exp_range' Policy")
plt.plot(clr_triangular.history['iterations'], clr_triangular.history['l
oss'])
plt.show()

plt.xlabel('Training Iterations')
plt.ylabel('Accuracy')
plt.title("CLR - 'exp_range' Policy")
plt.plot(clr_triangular.history['iterations'], clr_triangular.history['a
ccuracy'])
plt.show()
```







(curiosity) compare performance with our parameters vs with the paper parameters

In [15]:
```
clr_triangular_paper_param = CyclicLR(mode='exp_range', gamma=0.99994,ba
se_lr=0.001, max_lr=0.006)

num_epochs = 20
model_with_paper_param = create_model()
model_with_paper_param.compile(loss=keras.losses.categorical_crossentrop
y, metrics=['accuracy'])
model_with_paper_param.fit(x=x_train,y=y_train, epochs=num_epochs,
                          batch_size=batch_size, validation_data=(x_test
, y_test),
                           verbose=1,callbacks=[clr_triangular_paper_par
am])
```

```
Epoch 1/20
938/938 [==============================] - 15s 16ms/step - loss: 0.6160
- accuracy: 0.7751 - val_loss: 0.6492 - val_accuracy: 0.7545
Epoch 2/20
938/938 [==============================] - 15s 16ms/step - loss: 0.5516
- accuracy: 0.7997 - val_loss: 0.6809 - val_accuracy: 0.7598
Epoch 3/20
938/938 [==============================] - 15s 16ms/step - loss: 0.5095
- accuracy: 0.8141 - val_loss: 0.4996 - val_accuracy: 0.8188
Epoch 4/20
938/938 [==============================] - 15s 16ms/step - loss: 0.4377
- accuracy: 0.8403 - val_loss: 0.4228 - val_accuracy: 0.8495
Epoch 5/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3890
- accuracy: 0.8575 - val_loss: 0.4547 - val_accuracy: 0.8323
Epoch 6/20
938/938 [==============================] - 15s 16ms/step - loss: 0.4075
- accuracy: 0.8524 - val_loss: 0.4439 - val_accuracy: 0.8380
Epoch 7/20
938/938 [==============================] - 15s 16ms/step - loss: 0.4310
- accuracy: 0.8450 - val_loss: 0.5207 - val_accuracy: 0.8156
Epoch 8/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3936
- accuracy: 0.8596 - val_loss: 0.3982 - val_accuracy: 0.8616
Epoch 9/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3556
- accuracy: 0.8725 - val_loss: 0.3881 - val_accuracy: 0.8647
Epoch 10/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3589
- accuracy: 0.8711 - val_loss: 0.4132 - val_accuracy: 0.8494
Epoch 11/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3880
- accuracy: 0.8605 - val_loss: 0.4546 - val_accuracy: 0.8426
Epoch 12/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3709
- accuracy: 0.8681 - val_loss: 0.3896 - val_accuracy: 0.8592
Epoch 13/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3374
- accuracy: 0.8796 - val_loss: 0.3602 - val_accuracy: 0.8736
Epoch 14/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3341
- accuracy: 0.8815 - val_loss: 0.3681 - val_accuracy: 0.8683
Epoch 15/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3487
- accuracy: 0.8759 - val_loss: 0.3952 - val_accuracy: 0.8578
Epoch 16/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3528
- accuracy: 0.8750 - val_loss: 0.3634 - val_accuracy: 0.8733
Epoch 17/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3333
- accuracy: 0.8823 - val_loss: 0.3458 - val_accuracy: 0.8771
Epoch 18/20
938/938 [==============================] - 16s 17ms/step - loss: 0.3159
- accuracy: 0.8882 - val_loss: 0.3760 - val_accuracy: 0.8625
Epoch 19/20
938/938 [==============================] - 16s 17ms/step - loss: 0.3295
- accuracy: 0.8846 - val_loss: 0.3567 - val_accuracy: 0.8737
```

```
Epoch 20/20
938/938 [==============================] - 15s 16ms/step - loss: 0.3324
- accuracy: 0.8832 - val_loss: 0.3544 - val_accuracy: 0.8780
```

Out[15]: <tensorflow.python.keras.callbacks.History at 0x7f76e0faf450>

our parameters works the best for our data even though the paper parameters are also good.

Question 4:

In [108]:
```
num_epochs = 5
lr_min = 1e-4
hist = []

batch_size_list = [16*2**i for i in range(10)]
for bs in [16*2**i for i in range(10)]:
    opt = SGD(lr=lr_min)
    model = create_model()
    model.compile(loss=keras.losses.categorical_crossentropy, optimizer=
opt, metrics=['accuracy'])
    hist.append(model.fit(x=x_train,y=y_train, epochs=num_epochs,
                          batch_size=bs, validation_data=(x_test, y_test
), verbose=1))
```

```
Epoch 1/5
3750/3750 [==============================] - 31s 8ms/step - loss: 0.769
8 - accuracy: 0.7373 - val_loss: 0.5907 - val_accuracy: 0.7863
Epoch 2/5
3750/3750 [==============================] - 31s 8ms/step - loss: 0.575
6 - accuracy: 0.8025 - val_loss: 0.5294 - val_accuracy: 0.8145
Epoch 3/5
3750/3750 [==============================] - 31s 8ms/step - loss: 0.527
7 - accuracy: 0.8203 - val_loss: 0.4980 - val_accuracy: 0.8251
Epoch 4/5
3750/3750 [==============================] - 31s 8ms/step - loss: 0.498
5 - accuracy: 0.8300 - val_loss: 0.4785 - val_accuracy: 0.8322
Epoch 5/5
3750/3750 [==============================] - 31s 8ms/step - loss: 0.479
6 - accuracy: 0.8361 - val_loss: 0.4642 - val_accuracy: 0.8370
Epoch 1/5
1875/1875 [==============================] - 18s 9ms/step - loss: 0.872
0 - accuracy: 0.7034 - val_loss: 0.6451 - val_accuracy: 0.7728
Epoch 2/5
1875/1875 [==============================] - 18s 9ms/step - loss: 0.604
3 - accuracy: 0.7914 - val_loss: 0.5755 - val_accuracy: 0.7962
Epoch 3/5
1875/1875 [==============================] - 17s 9ms/step - loss: 0.550
9 - accuracy: 0.8109 - val_loss: 0.5419 - val_accuracy: 0.8107
Epoch 4/5
1875/1875 [==============================] - 17s 9ms/step - loss: 0.523
4 - accuracy: 0.8219 - val_loss: 0.5218 - val_accuracy: 0.8192
Epoch 5/5
1875/1875 [==============================] - 18s 9ms/step - loss: 0.502
8 - accuracy: 0.8277 - val_loss: 0.5060 - val_accuracy: 0.8248
Epoch 1/5
938/938 [==============================] - 11s 12ms/step - loss: 0.9786
- accuracy: 0.6713 - val_loss: 0.7433 - val_accuracy: 0.7440
Epoch 2/5
938/938 [==============================] - 11s 12ms/step - loss: 0.6731
- accuracy: 0.7699 - val_loss: 0.6525 - val_accuracy: 0.7755
Epoch 3/5
938/938 [==============================] - 11s 12ms/step - loss: 0.6072
- accuracy: 0.7929 - val_loss: 0.6080 - val_accuracy: 0.7908
Epoch 4/5
938/938 [==============================] - 11s 12ms/step - loss: 0.5708
- accuracy: 0.8067 - val_loss: 0.5817 - val_accuracy: 0.7958
Epoch 5/5
938/938 [==============================] - 12s 13ms/step - loss: 0.5454
- accuracy: 0.8159 - val_loss: 0.5591 - val_accuracy: 0.8042
Epoch 1/5
469/469 [==============================] - 6s 14ms/step - loss: 1.1777
- accuracy: 0.6184 - val_loss: 0.8644 - val_accuracy: 0.7171
Epoch 2/5
469/469 [==============================] - 6s 13ms/step - loss: 0.7772
- accuracy: 0.7433 - val_loss: 0.7417 - val_accuracy: 0.7516
Epoch 3/5
469/469 [==============================] - 6s 13ms/step - loss: 0.6967
- accuracy: 0.7675 - val_loss: 0.6886 - val_accuracy: 0.7677
Epoch 4/5
469/469 [==============================] - 6s 13ms/step - loss: 0.6539
- accuracy: 0.7821 - val_loss: 0.6565 - val_accuracy: 0.7782
```

```
Epoch 5/5
469/469 [==============================] - 6s 13ms/step - loss: 0.6236
- accuracy: 0.7914 - val_loss: 0.6319 - val_accuracy: 0.7872
Epoch 1/5
235/235 [==============================] - 5s 20ms/step - loss: 1.4311
- accuracy: 0.5419 - val_loss: 1.2325 - val_accuracy: 0.6601
Epoch 2/5
235/235 [==============================] - 5s 19ms/step - loss: 0.9181
- accuracy: 0.7070 - val_loss: 0.8682 - val_accuracy: 0.7173
Epoch 3/5
235/235 [==============================] - 5s 20ms/step - loss: 0.8036
- accuracy: 0.7367 - val_loss: 0.7835 - val_accuracy: 0.7397
Epoch 4/5
235/235 [==============================] - 5s 20ms/step - loss: 0.7442
- accuracy: 0.7534 - val_loss: 0.7362 - val_accuracy: 0.7526
Epoch 5/5
235/235 [==============================] - 5s 19ms/step - loss: 0.7042
- accuracy: 0.7651 - val_loss: 0.7045 - val_accuracy: 0.7618
Epoch 1/5
118/118 [==============================] - 4s 35ms/step - loss: 1.8050
- accuracy: 0.3958 - val_loss: 1.8494 - val_accuracy: 0.4572
Epoch 2/5
118/118 [==============================] - 4s 33ms/step - loss: 1.1000
- accuracy: 0.6480 - val_loss: 1.1929 - val_accuracy: 0.6508
Epoch 3/5
118/118 [==============================] - 4s 33ms/step - loss: 0.9394
- accuracy: 0.6952 - val_loss: 0.9480 - val_accuracy: 0.6982
Epoch 4/5
118/118 [==============================] - 4s 33ms/step - loss: 0.8568
- accuracy: 0.7201 - val_loss: 0.8588 - val_accuracy: 0.7163
Epoch 5/5
118/118 [==============================] - 4s 33ms/step - loss: 0.8025
- accuracy: 0.7358 - val_loss: 0.8090 - val_accuracy: 0.7267
Epoch 1/5
59/59 [==============================] - 4s 63ms/step - loss: 1.8744 -
accuracy: 0.3548 - val_loss: 2.0793 - val_accuracy: 0.3209
Epoch 2/5
59/59 [==============================] - 4s 60ms/step - loss: 1.2744 -
accuracy: 0.5727 - val_loss: 1.7210 - val_accuracy: 0.4923
Epoch 3/5
59/59 [==============================] - 4s 59ms/step - loss: 1.0890 -
accuracy: 0.6389 - val_loss: 1.3561 - val_accuracy: 0.5928
Epoch 4/5
59/59 [==============================] - 3s 59ms/step - loss: 0.9925 -
accuracy: 0.6697 - val_loss: 1.1137 - val_accuracy: 0.6547
Epoch 5/5
59/59 [==============================] - 3s 59ms/step - loss: 0.9304 -
accuracy: 0.6873 - val_loss: 0.9824 - val_accuracy: 0.6813
Epoch 1/5
30/30 [==============================] - 3s 115ms/step - loss: 2.5809 -
accuracy: 0.1495 - val_loss: 2.2885 - val_accuracy: 0.1234
Epoch 2/5
30/30 [==============================] - 3s 108ms/step - loss: 1.7790 -
accuracy: 0.3794 - val_loss: 2.2081 - val_accuracy: 0.1534
Epoch 3/5
30/30 [==============================] - 3s 108ms/step - loss: 1.4725 -
accuracy: 0.4961 - val_loss: 2.0883 - val_accuracy: 0.2118
```

```
Epoch 4/5
30/30 [==============================] - 3s 108ms/step - loss: 1.3076 -
accuracy: 0.5555 - val_loss: 1.9261 - val_accuracy: 0.3054
Epoch 5/5
30/30 [==============================] - 3s 109ms/step - loss: 1.2009 -
accuracy: 0.5938 - val_loss: 1.7358 - val_accuracy: 0.3968
Epoch 1/5
15/15 [==============================] - 3s 221ms/step - loss: 2.8043 -
accuracy: 0.1799 - val_loss: 2.3017 - val_accuracy: 0.1123
Epoch 2/5
15/15 [==============================] - 3s 206ms/step - loss: 2.3787 -
accuracy: 0.2788 - val_loss: 2.2725 - val_accuracy: 0.1622
Epoch 3/5
15/15 [==============================] - 3s 206ms/step - loss: 2.0854 -
accuracy: 0.3505 - val_loss: 2.2352 - val_accuracy: 0.1807
Epoch 4/5
15/15 [==============================] - 3s 207ms/step - loss: 1.8755 -
accuracy: 0.3971 - val_loss: 2.1881 - val_accuracy: 0.2224
Epoch 5/5
15/15 [==============================] - 3s 206ms/step - loss: 1.7183 -
accuracy: 0.4340 - val_loss: 2.1299 - val_accuracy: 0.2705
Epoch 1/5
8/8 [==============================] - 3s 384ms/step - loss: 3.2551 - a
ccuracy: 0.0568 - val_loss: 2.3145 - val_accuracy: 0.0552
Epoch 2/5
8/8 [==============================] - 3s 354ms/step - loss: 2.8676 - a
ccuracy: 0.0935 - val_loss: 2.3006 - val_accuracy: 0.0656
Epoch 3/5
8/8 [==============================] - 3s 358ms/step - loss: 2.5560 - a
ccuracy: 0.1461 - val_loss: 2.2847 - val_accuracy: 0.0784
Epoch 4/5
8/8 [==============================] - 3s 360ms/step - loss: 2.3087 - a
ccuracy: 0.2020 - val_loss: 2.2664 - val_accuracy: 0.1042
Epoch 5/5
8/8 [==============================] - 3s 351ms/step - loss: 2.1132 - a
ccuracy: 0.2561 - val_loss: 2.2455 - val_accuracy: 0.1399
```

In [109]:

```python
training_loss = []
training_accuracy = []
batch_size_list = [16*2**i for i in range(10)]
for i in range(len(batch_size_list)):
    training_loss.append(hist[i].history['loss'][4])
    training_accuracy.append(hist[i].history['accuracy'][4])
```

In [110]:

```python
training_loss
```

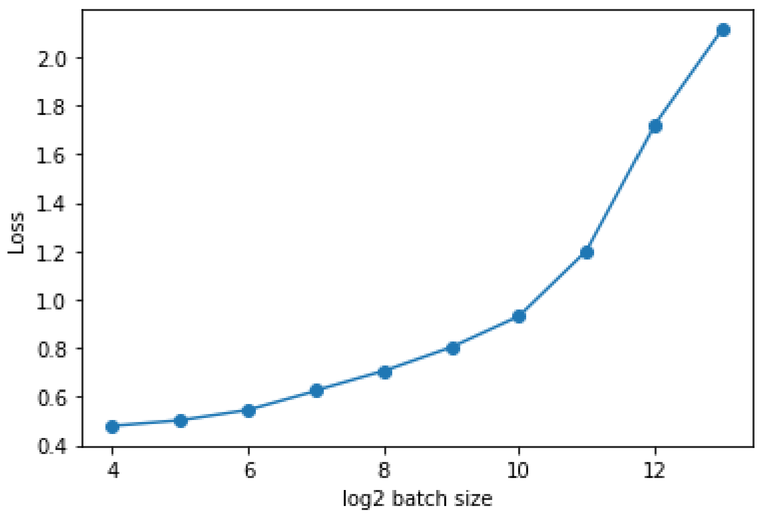
Out[110]:

```
[0.47962191700935364,
 0.5027687549591064,
 0.5453821420669556,
 0.6236060261726379,
 0.7042196393013,
 0.8025332689285278,
 0.9304342269897461,
 1.2008851766586304,
 1.7183446884155273,
 2.113232374191284]
```

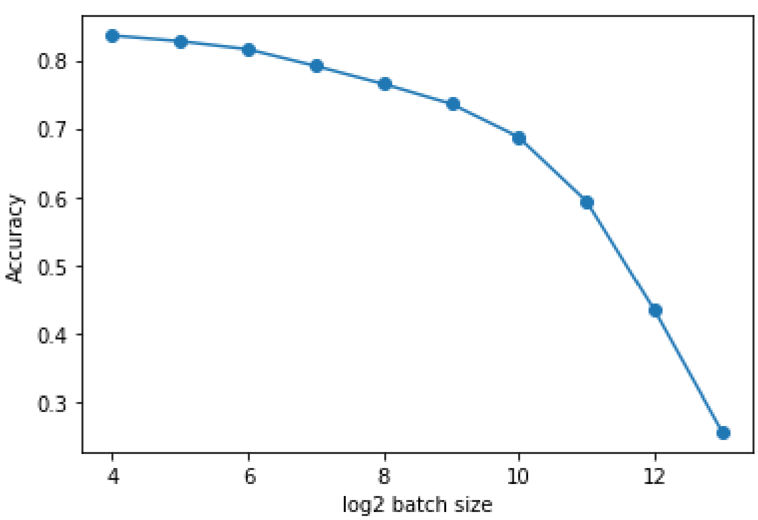
In [111]:
```
training_accuracy
```

Out[111]:
```
[0.8360666632652283,
 0.8276833295822144,
 0.8158666491508484,
 0.7914333343505859,
 0.765100022888184,
 0.73580002784729,
 0.687250018119812,
 0.59375,
 0.4340499937534332,
 0.25609999895095825]
```

In [112]:
```
plt.plot(np.log2(batch_size_list),training_loss,marker='o')
plt.xlabel('log2 batch size')
plt.ylabel('Loss')
plt.show()
```



In [113]:
```
plt.plot(np.log2(batch_size_list),training_accuracy,marker='o')
plt.xlabel('log2 batch size')
plt.ylabel('Accuracy')
plt.show()
```



We clearly observe that the accuracy and the loss decrease as a function of batch size

Other way using effective batch size explained in class.

In [8]:
```
def effective_batchsize(model, x_train, y_train, x_test, y_test, batch_s
ize, batch_cycle_length, num_epochs,
                    lr=1e-4,momentum=0.9):

    def gradient_accumulations(acc_grad, steps_grad):
    ## "gradients": the accumulated gradients
    ## "step_gradients" : gradients computed in this step

        if acc_grad is None:
            acc_grad = steps_grad
        else:
            for iter, step_g in enumerate(steps_grad):
                acc_grad[iter] += step_g
        return (acc_grad)

    opt= SGD(lr,momentum)


    ## We start with bmin=batch_size and go in a cycling manner by epoch
s to bmax=batch_size*batch_cycle_length.
    ## We increase the mini batch by multiplying. it by all the integers
1 to batch_size_length.
    ## We basically get gradients every mini batch but
    ## performs the update only after target_batch_size_now/bmin iterati
ons



    # convert to tf object and generate bmin mini batch to train from bm
in batch size
    training = tf.data.Dataset.from_tensor_slices((x_train, y_train))
    # the trick is to use bmin for batch size so we have flexibility to
 increase
    training = training.shuffle(buffer_size=1000).batch(batch_size)

    # initialise model
    model.compile(loss=keras.losses.categorical_crossentropy, optimizer=
opt, metrics=['accuracy'])

    cross_entropy_loss = tf.keras.losses.CategoricalCrossentropy()



    train_loss_by_epochs = []
    train_accuracy_by_epochs = []
    val_loss_by_epochs = []
    val_accuracy_by_epochs = []


    for epoch in range(num_epochs):
        ## Progress Bar
        pb_i = Progbar(len(x_train) // batch_size + 1, verbose=1)
        ##

        # Initialization
        model.reset_metrics() # the metrics returned will be only for th
```

```
is batch
        acc_grad = None
        update_counter = 0
        train_logs = {}

        # Start the current epoch
        for _, (x_mini_batch_train, y_mini_batch_train) in enumerate(tra
ining):
            # number of mini batches=bmin in one epoch (here bmax=bmin*b
atch_cycle_length)

            with backprop.GradientTape()  as tape:

                # get prediction for this mini batch
                yhat = model(x_mini_batch_train, training=True)

                # Compute the loss value for this minibatch.
                mini_batch_loss_value = cross_entropy_loss(y_mini_batch_
train, yhat) / batch_cycle_length

            # gradients
            steps_grad = tape.gradient(mini_batch_loss_value, model.trai
nable_variables)

            # use function for gradient accumulation
            acc_grad = gradient_accumulations(acc_grad,
                                              steps_grad)
            # Update when length is reached
            if (update_counter == 0):        # batch_cycle_length of gra
dients accumulated
                # we update at targeted batch  size after training(cappe
d by bmin *cycle length and cycles
                # using bmin * (1 to batch  multiplier ) to implement cy
cle and update at every value of the cycle)
                opt.apply_gradients(zip(acc_grad, model.trainable_variab
les))
                acc_grad = None
                update_counter = batch_cycle_length

            # update metrics
            model.compiled_metrics.update_state(y_mini_batch_train, yhat
)
            train_logs = {m.name : float(m.result()) for m in model.metr
ics}

            #display verbose
            pb_i.add(1, values=[('loss', mini_batch_loss_value*batch_cyc
le_length), ('acc', train_logs['accuracy'])])

            # update counter before update
            update_counter -= 1

        ## Log result of current epoch
        # Average training and accuracy loss by epoch
        train_loss_by_epochs.append(float(mini_batch_loss_value) * batch
_cycle_length)
        train_accuracy_by_epochs.append(train_logs['accuracy'])
```

```
        # Validation
        val_logs = model.evaluate(x_test, y_test,
                                    batch_size=16,
                                  steps=10,
                                  return_dict=True,
                                  verbose=0)
        val_logs = {'validation_' + name: val for name, val in val_logs.
items()}


        # Log validation results
        val_loss_by_epochs.append(val_logs['validation_loss'])
        val_accuracy_by_epochs.append(val_logs['validation_accuracy'])

    print("mini-batches gradient updates varying from size : bmin = "+st
r(batch_size)+
          " to bmax = " + str(batch_size*batch_cycle_length))
    return {'train_loss': train_loss_by_epochs,'train_accuracy': train_a
ccuracy_by_epochs,
            'validation_loss': val_loss_by_epochs,'validation_accuracy':
val_accuracy_by_epochs}
```

In [10]:
```
lr_min = 1e-4
model = create_model()
history_metrics = effective_batchsize(model,
x_train=x_train, y_train=y_train,
x_test=x_test, y_test=y_test,
batch_size=256,
batch_cycle_length=16,
num_epochs=20,lr=1e-4)
```

```
235/235 [==============================] - 17s 72ms/step - loss: 2.1039
- acc: 0.1619
235/235 [==============================] - 11s 47ms/step - loss: 1.1214
- acc: 0.5916 1s - loss: 1.1353 -
235/235 [==============================] - 11s 47ms/step - loss: 0.9176
- acc: 0.6832
235/235 [==============================] - 11s 47ms/step - loss: 0.8362
- acc: 0.7113
235/235 [==============================] - 11s 47ms/step - loss: 0.7880
- acc: 0.7273
235/235 [==============================] - 11s 46ms/step - loss: 0.7535
- acc: 0.7390
235/235 [==============================] - 11s 47ms/step - loss: 0.7274
- acc: 0.7463
235/235 [==============================] - 11s 47ms/step - loss: 0.7062
- acc: 0.7539
235/235 [==============================] - 11s 47ms/step - loss: 0.6886
- acc: 0.7599 0s - loss: 0.6895 - a
235/235 [==============================] - 11s 46ms/step - loss: 0.6732
- acc: 0.7658
235/235 [==============================] - 11s 46ms/step - loss: 0.6600
- acc: 0.7706 0s - loss: 0.6600 - acc: 0.770
235/235 [==============================] - 11s 46ms/step - loss: 0.6472
- acc: 0.7759
235/235 [==============================] - 11s 47ms/step - loss: 0.6378
- acc: 0.7793
235/235 [==============================] - 11s 46ms/step - loss: 0.6275
- acc: 0.7822
235/235 [==============================] - 11s 46ms/step - loss: 0.6191
- acc: 0.7863
235/235 [==============================] - 11s 46ms/step - loss: 0.6112
- acc: 0.7889
235/235 [==============================] - 11s 46ms/step - loss: 0.6040
- acc: 0.7940
235/235 [==============================] - 11s 46ms/step - loss: 0.5968
- acc: 0.7984
235/235 [==============================] - 11s 46ms/step - loss: 0.5898
- acc: 0.7982
235/235 [==============================] - 11s 46ms/step - loss: 0.5836
- acc: 0.8011 1s - loss: 0
mini-batches gradient updates varying from size : bmin = 256 to bmax =
4096
```

In [11]:
```
history_metrics
```

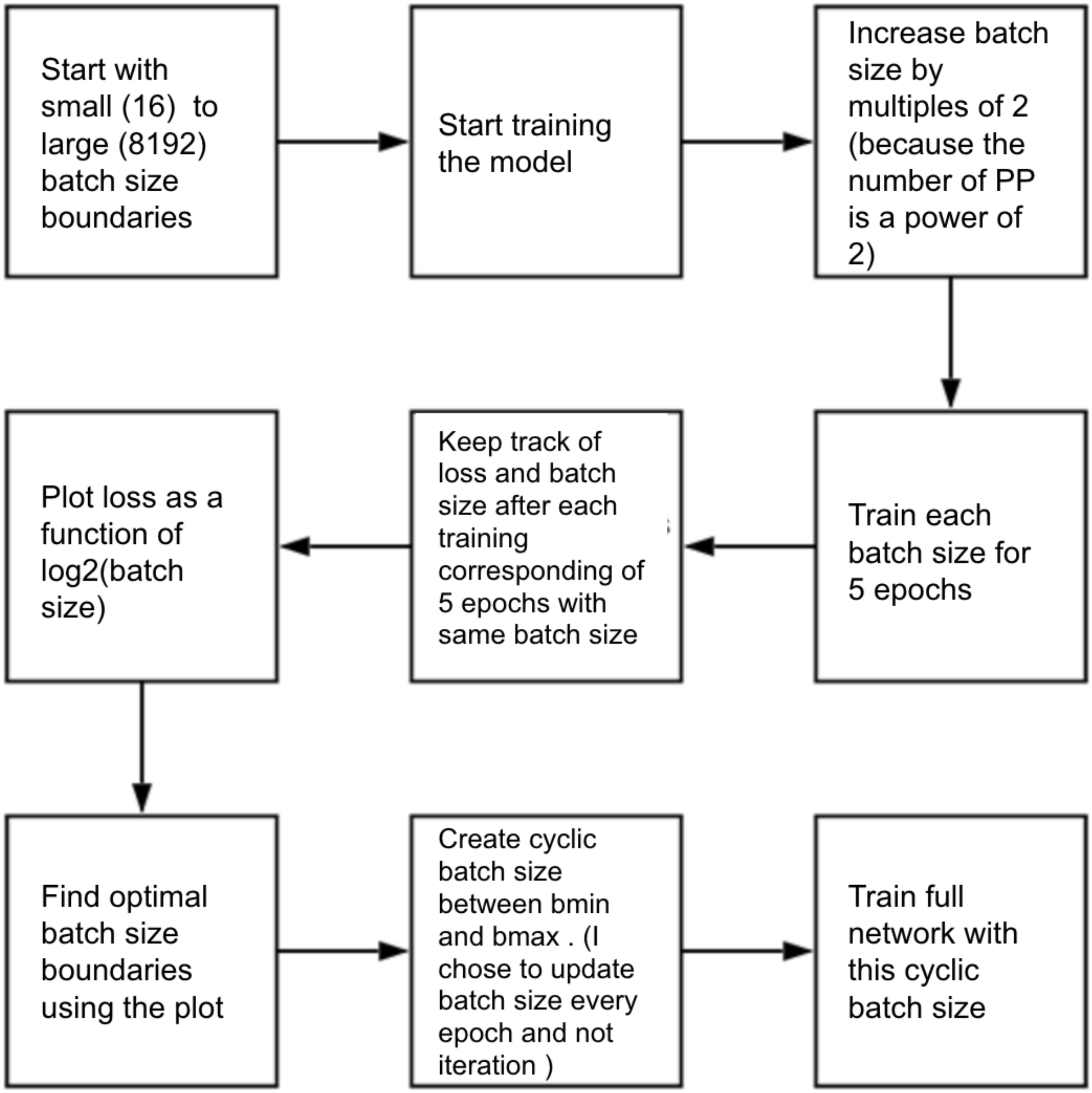
```
Out[11]: {'train_loss': [1.3469253778457642,
           1.0371612310409546,
           0.8850236535072327,
           0.7619113922119141,
           0.6883540749549866,
           0.6805928349494934,
           0.7656068801879883,
           0.5705147981643677,
           0.7585275173187256,
           0.6981126666069031,
           0.6773139834403992,
           0.579261839389801,
           0.6500706076622009,
           0.561093270778656,
           0.6294794678688049,
           0.6367425918579102,
           0.7235509753227234,
           0.6274170279502869,
           0.4145136773586273,
           0.5649299621582031],
          'train_accuracy': [0.29883334040641785,
           0.6182166934013367,
           0.6876999735832214,
           0.7133333086967468,
           0.7279999852180481,
           0.7390166521072388,
           0.7466833591461182,
           0.753333330154419,
           0.7592333555221558,
           0.7651166915893555,
           0.7701666951179504,
           0.7743499875068665,
           0.777649998664856,
           0.7809333205223083,
           0.7847833037376404,
           0.788349986076355,
           0.7918000221252441,
           0.7949333190917969,
           0.7964166402816772,
           0.7986833453178406],
          'validation_loss': [1.400050163269043,
           0.9917440414428711,
           0.8770163655281067,
           0.8202991485595703,
           0.7850354909896851,
           0.7599949836730957,
           0.7400252223014832,
           0.723223090171814,
           0.7087602019309998,
           0.6957947611808777,
           0.6840713620185852,
           0.6737698316574097,
           0.6643552184104919,
           0.6558343172073364,
           0.6474583745002747,
           0.6401389837265015,
           0.6331372261047363,
```

```
 0.6268541216850281,
 0.6209380030632019,
 0.6155293583869934],
'validation_accuracy': [0.53125,
 0.65625,
 0.7124999761581421,
 0.731249988079071,
 0.75,
 0.7437499761581421,
 0.75,
 0.7562500238418579,
 0.75,
 0.75,
 0.7749999761581421,
 0.7749999761581421,
 0.768750011920929,
 0.762499988079071,
 0.7562500238418579,
 0.7562500238418579,
 0.762499988079071,
 0.762499988079071,
 0.768750011920929,
 0.768750011920929]}
```

Question 5:

In [15]:
```
from PIL import Image, ImageDraw
Image.open('batch_size_process.png')
```

Out[15]:



Algorithm for automatic detection of bmin and bmax

In [27]:
```
threshold_decrease = 3.5 ## arbitrary (the user can change the threshold percentage decrease)
training_accuracy = [0.8360666632652283,0.8276833295822144,0.8158666491508484,0.7914333343505859,
                     0.7651000022888184,0.73580002784729,0.687250018119812,0.59375,
                     0.4340499937534332,0.25609999895095825]
batch_size_list = [16*2**i for i in range(10)]
def detect_bmin_bmax (training_accuracy,threshold_decrease,batch_size_list):
    list_decrease = [(100.0 * (a1-a2) / a1) >-threshold_decrease  for a1, a2 in zip(training_accuracy[1:], training_accuracy)]
    candidates = [i for i, x in enumerate(list_decrease) if x]
    if (len(candidates)<1):
        print("Increase list batch size and | or increase threshold of decreased accuracy")
    else :
        bmin = 16*2**candidates[0]
        bmax = 16*2**(max(candidates)+1)
    return (bmin,bmax)
```

In [28]:
```
bmin,bmax = detect_bmin_bmax (training_accuracy,threshold_decrease,batch_size_list)
print('bmin :' + str(bmin))
print('bmax :' + str(bmax))
```

```
bmin :16
bmax :256
```

In [105]:
```
bmin=16
bmax=256
```

Question 6:

We start at a small batch size(faster training dynamics) then we grow the batch size so an exponential increase is better . Again, here. we change the batch size using multiple of 2 (the number of physical processor is a power of 2 , therefore we should use virtual processors as a power of 2 also. We change in a cyclic manner as explained above. the cyclic batch size every epoch .

For me , there is limitation by doing so because learning rate is fixed while we change batch size . My feeling is that both batch. size and learning rate should be adapted in a. cyclic manner since we usually want large learning rates for large batch sizes because we are confident of our gradient computations while smaller learning rates when the batch size is small since there is more noise and therefore we trust less our gradient computations . Both cycles are interesting and we should take the best of both worlds.

In [107]:
```
batch_sizes_schedule=[]
for i in range(int(np.log2(bmax/bmin))+1):
    batch_sizes_schedule.append(bmin*2**i )

model = create_model()
opt = SGD(0.01)
model.compile(loss=keras.losses.categorical_crossentropy, metrics=['accu
racy'],optimizer=opt)
for i in range(20):
    print(batch_sizes_schedule[i%len(batch_sizes_schedule)])
    model.fit(x=x_train,y=y_train, epochs=1, validation_data=(x_test, y_
test), verbose=1,
          batch_size=batch_sizes_schedule[i%3])
```

```
16
3750/3750 [==============================] - 31s 8ms/step - loss: 0.495
4 - accuracy: 0.8255 - val_loss: 0.4003 - val_accuracy: 0.8550
32
1875/1875 [==============================] - 18s 10ms/step - loss: 0.33
28 - accuracy: 0.8817 - val_loss: 0.3669 - val_accuracy: 0.8659
64
938/938 [==============================] - 11s 12ms/step - loss: 0.2775
- accuracy: 0.9012 - val_loss: 0.3157 - val_accuracy: 0.8867
128
3750/3750 [==============================] - 31s 8ms/step - loss: 0.327
8 - accuracy: 0.8836 - val_loss: 0.3473 - val_accuracy: 0.8771
256
1875/1875 [==============================] - 18s 10ms/step - loss: 0.25
44 - accuracy: 0.9083 - val_loss: 0.2960 - val_accuracy: 0.8941
16
938/938 [==============================] - 11s 12ms/step - loss: 0.2101
- accuracy: 0.9255 - val_loss: 0.2855 - val_accuracy: 0.9000
32
3750/3750 [==============================] - 31s 8ms/step - loss: 0.273
7 - accuracy: 0.9019 - val_loss: 0.3162 - val_accuracy: 0.8861
64
1875/1875 [==============================] - 18s 10ms/step - loss: 0.20
81 - accuracy: 0.9267 - val_loss: 0.2685 - val_accuracy: 0.9040
128
938/938 [==============================] - 11s 12ms/step - loss: 0.1675
- accuracy: 0.9414 - val_loss: 0.2652 - val_accuracy: 0.9058
256
3750/3750 [==============================] - 31s 8ms/step - loss: 0.230
3 - accuracy: 0.9168 - val_loss: 0.3022 - val_accuracy: 0.8928
16
1875/1875 [==============================] - 18s 10ms/step - loss: 0.16
96 - accuracy: 0.9398 - val_loss: 0.2771 - val_accuracy: 0.9077
32
938/938 [==============================] - 11s 12ms/step - loss: 0.1336
- accuracy: 0.9547 - val_loss: 0.2671 - val_accuracy: 0.9094
64
3750/3750 [==============================] - 31s 8ms/step - loss: 0.201
4 - accuracy: 0.9284 - val_loss: 0.2957 - val_accuracy: 0.9032
128
1875/1875 [==============================] - 18s 10ms/step - loss: 0.14
39 - accuracy: 0.9493 - val_loss: 0.2775 - val_accuracy: 0.9076
256
938/938 [==============================] - 12s 12ms/step - loss: 0.1062
- accuracy: 0.9640 - val_loss: 0.2752 - val_accuracy: 0.9097
16
3750/3750 [==============================] - 31s 8ms/step - loss: 0.173
3 - accuracy: 0.9373 - val_loss: 0.2945 - val_accuracy: 0.9032
32
1875/1875 [==============================] - 18s 10ms/step - loss: 0.11
57 - accuracy: 0.9598 - val_loss: 0.2737 - val_accuracy: 0.9110
64
938/938 [==============================] - 11s 12ms/step - loss: 0.0823
- accuracy: 0.9741 - val_loss: 0.2769 - val_accuracy: 0.9113
128
3750/3750 [==============================] - 31s 8ms/step - loss: 0.152
9 - accuracy: 0.9451 - val_loss: 0.3147 - val_accuracy: 0.9065
```

```
256
1875/1875 [==============================] - 18s 10ms/step - loss: 0.09
51 - accuracy: 0.9682 - val_loss: 0.2825 - val_accuracy: 0.9110
```

Question 7:

It seems that the accuracy is better when we perform cycling policy of the batch size (0.911) and it converges faster because it allows the benefit from low batch size (quick convergence, safer and better local approximation) as well as large batch size (more confidence in gradient estimation butleads to poor generalization) in a cycling manner.

it is advised to start at a small batch size(faster training dynamics) then we grow the batch size through (guaranteed convergence)

There is this beautiful paper that explain some batch size concepts very nicely :DON'T DECAY THE LEARNING RATE, INCREASE THE BATCH SIZE [https://arxiv.org/pdf/1711.00489.pdf (https://arxiv.org/pdf/1711.00489.pdf)](https://arxiv.org/pdf/1711.00489.pdf)

In [ ]: