Name: Yifan Bai

ID: 20153885

Deep Learning (Convolutional Neural Networks) [70 points]

This set of assignments will give you experience with deep learning. You will learn how to use convolution neural networks on a image corpus.

For this problem use documentation for Keras (https://keras.io/) deep learning library and for Sklearn (https://scikit-learn.org/stable/index.html). Provide your code and the output.

Data preparation (20 points)

1. (1 point) Search MNIST dataset at OpenML (https://www.openml.org/), it is called "mnist_784". Download it using sklearn function `fetch_openml` . Get features and targets.

In [1]:
```
import sklearn
from sklearn.datasets import fetch_openml
```

In [3]:
```
# Your solution here
X,y=fetch_openml('mnist_784',version=1,return_X_y=True)
X=X/255.
# X_train,X_test=X[:60000],X[60000:]
# y_train,y_test=y[:60000],y[60000:]
print('X.shape..', X.shape)
print('y.shape..', y.shape)
```

```
X.shape.. (70000, 784)
y.shape.. (70000,)
```

> This dataset represents each (28x28) image as a flat arrray of 784 features.

1. (3 point) Reshape it back to 28x28 and visualize a couple of images with matplotlib.

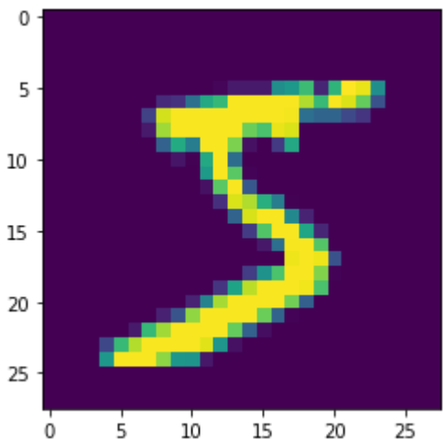
> Finally, we want to add one dummy dimension for the non-existent color information. Every resulting image should have dimensions 28x28x1.
>
> We want this extra dimensions because image libraries are targeted towards RGB images that have 3 channels. Therefore RGB images are conviniently represented by the shape of WxHx3. We don't have 3 colors for MNIST dataset, so we just provide 1 channel.

1. (3 point) Add a channel dimension.

In [17]:
```
# Your solution here
import numpy as np
import matplotlib.pyplot as plt

print(y[0])
img = X[0].reshape(28, 28)
# img = img.squeeze(axis=2)
plt.imshow(img)
```

```
5
```

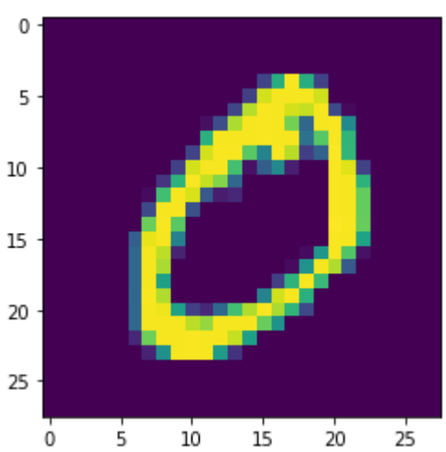
Out[17]:
```
<matplotlib.image.AxesImage at 0x7f1916eb7fd0>
```



In [18]:
```
print(y[1])
img = X[1].reshape(28, 28)
# img = img.squeeze(axis=2)
plt.imshow(img)
```

```
0
```

Out[18]:
```
<matplotlib.image.AxesImage at 0x7f1916e13f98>
```



In [19]:
```
### Add a deimension
new_X = []
for i in range(X.shape[0]):
    new_X.append( X[i].reshape(28, 28, 1) )
new_X = np.array( new_X )
new_X.shape
```

Out[19]:
```
(70000, 28, 28, 1)
```

To simplify the task we exclude some classes.

1. (3 point) Filter data leaving only classes 1 , 3 , 7 . Transform features and targets. How many data points left after filtering?

In [35]:
```
# Your solution here

filter_X = []
filter_y = []

for i in range(y.shape[0]):

    if int(y[i])==1 or int(y[i])==3 or int(y[i])==7:
        filter_y.append( y[i] )
        filter_X.append( new_X[i] )
filter_X = np.array( filter_X )
filter_y = np.array( filter_y )

print('filter X..', filter_X.shape)
print('filter y..', filter_y.shape)
```

```
filter X.. (22311, 28, 28, 1)
filter y.. (22311,)
```

In [36]:
```
set(list(filter_y))
```

Out[36]:
```
{'1', '3', '7'}
```

- After filtering, number of 70000 reduces to 22311.

1. (5 point) Convert targets to one-hot representation. Complete the following template.

```
def to_categorical(array, classes):
    """
    array -- array of targets
    classes -- list of classes
    """
    ...

mnist_targets = to_categorical(mnist_targets, classes=...)
```

In [39]:
```
# Your solution here

### 1,4,7 mapping 0, 1, 2
from keras.utils import np_utils

new_filter_y = []
for i in range(filter_y.shape[0]):
    tmp_y = int(filter_y[i])
    if tmp_y==1:
        new_filter_y.append(0)
    elif tmp_y==3:
        new_filter_y.append(1)
    elif tmp_y==7:
        new_filter_y.append(2)
new_filter_y = np.array(new_filter_y)

oh_y = np_utils.to_categorical(new_filter_y)
oh_y.shape
```

Out[39]:
```
(22311, 3)
```

1. (5 point) Split the dataset into train, validataion, and test. Take first 16,000 images and targets as the train, then next 3,000 as validation, then the rest as the test subset.

In [41]:
```
# Your solution here

from sklearn.model_selection import train_test_split

train_x = filter_X[:16000]
val_x = filter_X[16000:19000]
test_x = filter_X[19000:]

print(train_x.shape)
print(val_x.shape)
print(test_x.shape)

train_y = oh_y[:16000]
val_y = oh_y[16000:19000]
test_y = oh_y[19000:]
print(train_y.shape)
print(val_y.shape)
print(test_y.shape)
```

```
(16000, 28, 28, 1)
(3000, 28, 28, 1)
(3311, 28, 28, 1)
(16000, 3)
(3000, 3)
(3311, 3)
```

In [ ]:
```

```

Training (35 points)

Use Keras (https://keras.io/ (https://keras.io/)) to create a neural network model. Use a sequential layer to combine following layers in this order:

- Convolution with 6 feature maps 5x5
- Rectified linear unit activation
- Max-pooling by factor of 2 each spacial dimension
- Convolution with 16 feature maps 5x5
- Rectified linear unit activation
- Max-pooling by factor of 2 each spacial dimension
- Flatten layer
- Dense layer with 128 output units
- Rectified linear unit activation
- Dense layer. Same size as the target.
- Softmax activation

1. (10 points) Complete the following template.

```
... # place your imports here

model = Sequential([
    ..., # convolution
    ..., # activation
    ..., # pooling
    ..., # convolution
    ..., # activation
    ..., # pooling
    Flatten(),
    ..., # fully connected
    ..., # activation
    ..., # fully connected output
    ..., # softmax
])
model.summary()
```

In [58]:
```
# Your solution here

from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPool2D

model = Sequential([
    Conv2D(6, (5, 5), padding = 'same',input_shape = (28, 28, 1)),
    Activation('relu'),
    MaxPool2D(pool_size = 2),
    Conv2D(16, (5,5), padding = 'same'),
    Activation('relu'),
    MaxPool2D(pool_size = 2),
    Flatten(),
    Dense(128),
    Activation('relu'),
    Dense(3),
    Activation('softmax')
])
```

1. (5 point) Create a stochastic gradient optimizer optimizer with learning rate of 10^{-4}. Compile the model with the categorical crossentropy loss. Set the model to report accuracy metric. Complete the template.

```
... # place your imports here

optimizer = ... # create stochastic gradient optimizer
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'],
              )
```

In [59]:
```
# Your solution here
from keras import optimizers
optimizer = optimizers.SGD(lr=0.0001)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'],
              )
```

1. (15 points) Train the model on the training set for at least 5 epochs. Perform validation after every epoch.

> **HINT** Find a method that performs training in the Keras documentation. Study the documentation paying attention to all arguments and the return value of the method.
>
> The model should have at least 95% accuracy on the training set. It might happen that the training gets stuck. In this case, go to the step before prevoious, recreate and rerun the model.
>
> **WARNING** This step might take several minutes to compute on a laptop.

In [61]:

```
import keras
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = {'batch':[], 'epoch':[]}
        self.accuracy = {'batch':[], 'epoch':[]}
        self.val_loss = {'batch':[], 'epoch':[]}
        self.val_acc = {'batch':[], 'epoch':[]}

    def on_batch_end(self, batch, logs={}):
        self.losses['batch'].append(logs.get('loss'))
        self.accuracy['batch'].append(logs.get('acc'))
        self.val_loss['batch'].append(logs.get('val_loss'))
        self.val_acc['batch'].append(logs.get('val_acc'))

    def on_epoch_end(self, batch, logs={}):
        self.losses['epoch'].append(logs.get('loss'))
        self.accuracy['epoch'].append(logs.get('acc'))
        self.val_loss['epoch'].append(logs.get('val_loss'))
        self.val_acc['epoch'].append(logs.get('val_acc'))

    def loss_plot(self, loss_type):
        iters = range(len(self.losses[loss_type]))
        plt.figure()
        # acc
        plt.plot(iters, self.accuracy[loss_type], 'r', label='train a
cc')
        # loss
        plt.plot(iters, self.losses[loss_type], 'g', label='train los
s')
        if loss_type == 'epoch':
            # val_acc
            plt.plot(iters, self.val_acc[loss_type], 'b', label='val
 acc')
            # val_loss
            plt.plot(iters, self.val_loss[loss_type], 'k', label='val
loss')
        plt.grid(True)
        plt.xlabel(loss_type)
        plt.ylabel('acc-loss')
        plt.legend(loc="upper right")
        plt.show()
```

In [62]:
```
# Your solution here
history = LossHistory()
model.fit(train_x, train_y, batch_size=128, epochs = 100, verbose=1,
validation_data=(val_x, val_y),
          callbacks=[history])
```

```
Train on 16000 samples, validate on 3000 samples
Epoch 1/100
16000/16000 [==============================] - 2s 124us/step - loss:
1.0858 - accuracy: 0.4221 - val_loss: 1.0834 - val_accuracy: 0.4507
Epoch 2/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.0816 - accuracy: 0.4549 - val_loss: 1.0792 - val_accuracy: 0.4843
Epoch 3/100
16000/16000 [==============================] - 2s 116us/step - loss:
1.0775 - accuracy: 0.4919 - val_loss: 1.0751 - val_accuracy: 0.5190
Epoch 4/100
16000/16000 [==============================] - 2s 118us/step - loss:
1.0734 - accuracy: 0.5272 - val_loss: 1.0710 - val_accuracy: 0.5550
Epoch 5/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.0694 - accuracy: 0.5666 - val_loss: 1.0669 - val_accuracy: 0.5953
Epoch 6/100
16000/16000 [==============================] - 2s 115us/step - loss:
1.0654 - accuracy: 0.6055 - val_loss: 1.0629 - val_accuracy: 0.6320
Epoch 7/100
16000/16000 [==============================] - 2s 116us/step - loss:
1.0615 - accuracy: 0.6374 - val_loss: 1.0590 - val_accuracy: 0.6633
Epoch 8/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.0576 - accuracy: 0.6627 - val_loss: 1.0551 - val_accuracy: 0.6927
Epoch 9/100
16000/16000 [==============================] - 2s 117us/step - loss:
1.0538 - accuracy: 0.6889 - val_loss: 1.0511 - val_accuracy: 0.7193
Epoch 10/100
16000/16000 [==============================] - 2s 117us/step - loss:
1.0499 - accuracy: 0.7109 - val_loss: 1.0472 - val_accuracy: 0.7387
Epoch 11/100
16000/16000 [==============================] - 2s 115us/step - loss:
1.0460 - accuracy: 0.7302 - val_loss: 1.0433 - val_accuracy: 0.7537
Epoch 12/100
16000/16000 [==============================] - 2s 115us/step - loss:
1.0421 - accuracy: 0.7484 - val_loss: 1.0393 - val_accuracy: 0.7723
Epoch 13/100
16000/16000 [==============================] - 2s 116us/step - loss:
1.0382 - accuracy: 0.7659 - val_loss: 1.0353 - val_accuracy: 0.7873
Epoch 14/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.0342 - accuracy: 0.7820 - val_loss: 1.0312 - val_accuracy: 0.8003
Epoch 15/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.0301 - accuracy: 0.7974 - val_loss: 1.0270 - val_accuracy: 0.8157
Epoch 16/100
16000/16000 [==============================] - 2s 116us/step - loss:
1.0259 - accuracy: 0.8104 - val_loss: 1.0228 - val_accuracy: 0.8280
Epoch 17/100
16000/16000 [==============================] - 2s 115us/step - loss:
1.0216 - accuracy: 0.8217 - val_loss: 1.0184 - val_accuracy: 0.8383
Epoch 18/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.0173 - accuracy: 0.8354 - val_loss: 1.0140 - val_accuracy: 0.8497
Epoch 19/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.0128 - accuracy: 0.8431 - val_loss: 1.0095 - val_accuracy: 0.8607
Epoch 20/100
16000/16000 [==============================] - 2s 117us/step - loss:
1.0083 - accuracy: 0.8529 - val_loss: 1.0049 - val_accuracy: 0.8713
Epoch 21/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.0036 - accuracy: 0.8602 - val_loss: 1.0001 - val_accuracy: 0.8797
Epoch 22/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.9988 - accuracy: 0.8671 - val_loss: 0.9953 - val_accuracy: 0.8897
Epoch 23/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.9939 - accuracy: 0.8724 - val_loss: 0.9902 - val_accuracy: 0.8937
Epoch 24/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.9888 - accuracy: 0.8769 - val_loss: 0.9851 - val_accuracy: 0.8953
Epoch 25/100
```

```
16000/16000 [==============================] - 2s 115us/step - loss:
0.9835 - accuracy: 0.8813 - val_loss: 0.9798 - val_accuracy: 0.9000
Epoch 26/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.9781 - accuracy: 0.8852 - val_loss: 0.9743 - val_accuracy: 0.9003
Epoch 27/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.9725 - accuracy: 0.8877 - val_loss: 0.9686 - val_accuracy: 0.9020
Epoch 28/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.9667 - accuracy: 0.8904 - val_loss: 0.9627 - val_accuracy: 0.9040
Epoch 29/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.9607 - accuracy: 0.8915 - val_loss: 0.9566 - val_accuracy: 0.9043
Epoch 30/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.9545 - accuracy: 0.8939 - val_loss: 0.9503 - val_accuracy: 0.9047
Epoch 31/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.9481 - accuracy: 0.8956 - val_loss: 0.9438 - val_accuracy: 0.9067
Epoch 32/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.9414 - accuracy: 0.8970 - val_loss: 0.9371 - val_accuracy: 0.9063
Epoch 33/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.9345 - accuracy: 0.8982 - val_loss: 0.9301 - val_accuracy: 0.9073
Epoch 34/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.9274 - accuracy: 0.9004 - val_loss: 0.9228 - val_accuracy: 0.9083
Epoch 35/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.9200 - accuracy: 0.9021 - val_loss: 0.9153 - val_accuracy: 0.9097
Epoch 36/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.9123 - accuracy: 0.9028 - val_loss: 0.9075 - val_accuracy: 0.9100
Epoch 37/100
16000/16000 [==============================] - 2s 112us/step - loss:
0.9043 - accuracy: 0.9045 - val_loss: 0.8994 - val_accuracy: 0.9120
Epoch 38/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.8961 - accuracy: 0.9059 - val_loss: 0.8910 - val_accuracy: 0.9120
Epoch 39/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.8875 - accuracy: 0.9066 - val_loss: 0.8823 - val_accuracy: 0.9137
Epoch 40/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.8786 - accuracy: 0.9085 - val_loss: 0.8733 - val_accuracy: 0.9137
Epoch 41/100
16000/16000 [==============================] - 2s 118us/step - loss:
0.8695 - accuracy: 0.9097 - val_loss: 0.8639 - val_accuracy: 0.9147
Epoch 42/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.8600 - accuracy: 0.9111 - val_loss: 0.8543 - val_accuracy: 0.9170
Epoch 43/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.8501 - accuracy: 0.9124 - val_loss: 0.8442 - val_accuracy: 0.9173
Epoch 44/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.8400 - accuracy: 0.9137 - val_loss: 0.8339 - val_accuracy: 0.9200
Epoch 45/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.8295 - accuracy: 0.9154 - val_loss: 0.8232 - val_accuracy: 0.9223
Epoch 46/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.8186 - accuracy: 0.9163 - val_loss: 0.8122 - val_accuracy: 0.9227
Epoch 47/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.8075 - accuracy: 0.9181 - val_loss: 0.8008 - val_accuracy: 0.9237
Epoch 48/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.7960 - accuracy: 0.9200 - val_loss: 0.7891 - val_accuracy: 0.9240
Epoch 49/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.7842 - accuracy: 0.9214 - val_loss: 0.7771 - val_accuracy: 0.9237
```

```
Epoch 50/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.7721 - accuracy: 0.9222 - val_loss: 0.7648 - val_accuracy: 0.9243
Epoch 51/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.7597 - accuracy: 0.9237 - val_loss: 0.7521 - val_accuracy: 0.9257
Epoch 52/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.7470 - accuracy: 0.9251 - val_loss: 0.7392 - val_accuracy: 0.9277
Epoch 53/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.7340 - accuracy: 0.9269 - val_loss: 0.7260 - val_accuracy: 0.9277
Epoch 54/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.7208 - accuracy: 0.9282 - val_loss: 0.7125 - val_accuracy: 0.9293
Epoch 55/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.7073 - accuracy: 0.9294 - val_loss: 0.6987 - val_accuracy: 0.9307
Epoch 56/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.6936 - accuracy: 0.9308 - val_loss: 0.6848 - val_accuracy: 0.9323
Epoch 57/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.6797 - accuracy: 0.9324 - val_loss: 0.6707 - val_accuracy: 0.9337
Epoch 58/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.6657 - accuracy: 0.9336 - val_loss: 0.6564 - val_accuracy: 0.9353
Epoch 59/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.6516 - accuracy: 0.9344 - val_loss: 0.6420 - val_accuracy: 0.9360
Epoch 60/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.6374 - accuracy: 0.9359 - val_loss: 0.6276 - val_accuracy: 0.9360
Epoch 61/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.6231 - accuracy: 0.9369 - val_loss: 0.6131 - val_accuracy: 0.9367
Epoch 62/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.6088 - accuracy: 0.9374 - val_loss: 0.5985 - val_accuracy: 0.9373
Epoch 63/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.5945 - accuracy: 0.9385 - val_loss: 0.5840 - val_accuracy: 0.9393
Epoch 64/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.5803 - accuracy: 0.9394 - val_loss: 0.5696 - val_accuracy: 0.9393
Epoch 65/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.5661 - accuracy: 0.9404 - val_loss: 0.5552 - val_accuracy: 0.9390
Epoch 66/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.5521 - accuracy: 0.9412 - val_loss: 0.5410 - val_accuracy: 0.9403
Epoch 67/100
16000/16000 [==============================] - 2s 119us/step - loss:
0.5382 - accuracy: 0.9419 - val_loss: 0.5269 - val_accuracy: 0.9423
Epoch 68/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.5245 - accuracy: 0.9425 - val_loss: 0.5130 - val_accuracy: 0.9430
Epoch 69/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.5110 - accuracy: 0.9434 - val_loss: 0.4993 - val_accuracy: 0.9433
Epoch 70/100
16000/16000 [==============================] - 2s 118us/step - loss:
0.4977 - accuracy: 0.9435 - val_loss: 0.4859 - val_accuracy: 0.9430
Epoch 71/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.4847 - accuracy: 0.9442 - val_loss: 0.4727 - val_accuracy: 0.9440
Epoch 72/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.4720 - accuracy: 0.9452 - val_loss: 0.4599 - val_accuracy: 0.9447
Epoch 73/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.4596 - accuracy: 0.9456 - val_loss: 0.4473 - val_accuracy: 0.9470
Epoch 74/100
16000/16000 [==============================] - 2s 117us/step - loss:
```

```
0.4474 - accuracy: 0.9459 - val_loss: 0.4351 - val_accuracy: 0.9477
Epoch 75/100
16000/16000 [==============================] - 2s 112us/step - loss:
0.4357 - accuracy: 0.9466 - val_loss: 0.4232 - val_accuracy: 0.9483
Epoch 76/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.4242 - accuracy: 0.9471 - val_loss: 0.4117 - val_accuracy: 0.9500
Epoch 77/100
16000/16000 [==============================] - 2s 118us/step - loss:
0.4131 - accuracy: 0.9474 - val_loss: 0.4005 - val_accuracy: 0.9513
Epoch 78/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.4024 - accuracy: 0.9481 - val_loss: 0.3897 - val_accuracy: 0.9520
Epoch 79/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.3920 - accuracy: 0.9483 - val_loss: 0.3792 - val_accuracy: 0.9520
Epoch 80/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.3820 - accuracy: 0.9491 - val_loss: 0.3691 - val_accuracy: 0.9523
Epoch 81/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.3723 - accuracy: 0.9499 - val_loss: 0.3594 - val_accuracy: 0.9523
Epoch 82/100
16000/16000 [==============================] - 2s 112us/step - loss:
0.3630 - accuracy: 0.9502 - val_loss: 0.3500 - val_accuracy: 0.9527
Epoch 83/100
16000/16000 [==============================] - 2s 118us/step - loss:
0.3540 - accuracy: 0.9506 - val_loss: 0.3410 - val_accuracy: 0.9530
Epoch 84/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.3453 - accuracy: 0.9511 - val_loss: 0.3323 - val_accuracy: 0.9543
Epoch 85/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.3370 - accuracy: 0.9514 - val_loss: 0.3239 - val_accuracy: 0.9543
Epoch 86/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.3290 - accuracy: 0.9517 - val_loss: 0.3159 - val_accuracy: 0.9550
Epoch 87/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.3213 - accuracy: 0.9519 - val_loss: 0.3082 - val_accuracy: 0.9553
Epoch 88/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.3139 - accuracy: 0.9526 - val_loss: 0.3008 - val_accuracy: 0.9550
Epoch 89/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.3068 - accuracy: 0.9528 - val_loss: 0.2937 - val_accuracy: 0.9547
Epoch 90/100
16000/16000 [==============================] - 2s 111us/step - loss:
0.3000 - accuracy: 0.9531 - val_loss: 0.2868 - val_accuracy: 0.9550
Epoch 91/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.2935 - accuracy: 0.9532 - val_loss: 0.2803 - val_accuracy: 0.9563
Epoch 92/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.2872 - accuracy: 0.9533 - val_loss: 0.2740 - val_accuracy: 0.9567
Epoch 93/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.2812 - accuracy: 0.9537 - val_loss: 0.2679 - val_accuracy: 0.9570
Epoch 94/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.2754 - accuracy: 0.9543 - val_loss: 0.2622 - val_accuracy: 0.9577
Epoch 95/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.2699 - accuracy: 0.9545 - val_loss: 0.2566 - val_accuracy: 0.9577
Epoch 96/100
16000/16000 [==============================] - 2s 117us/step - loss:
0.2646 - accuracy: 0.9548 - val_loss: 0.2513 - val_accuracy: 0.9587
Epoch 97/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.2594 - accuracy: 0.9547 - val_loss: 0.2462 - val_accuracy: 0.9590
Epoch 98/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.2545 - accuracy: 0.9549 - val_loss: 0.2413 - val_accuracy: 0.9593
Epoch 99/100
```

```
16000/16000 [==============================] - 2s 115us/step - loss:
0.2498 - accuracy: 0.9554 - val_loss: 0.2366 - val_accuracy: 0.9600
Epoch 100/100
16000/16000 [==============================] - 2s 112us/step - loss:
0.2453 - accuracy: 0.9554 - val_loss: 0.2321 - val_accuracy: 0.9603
```

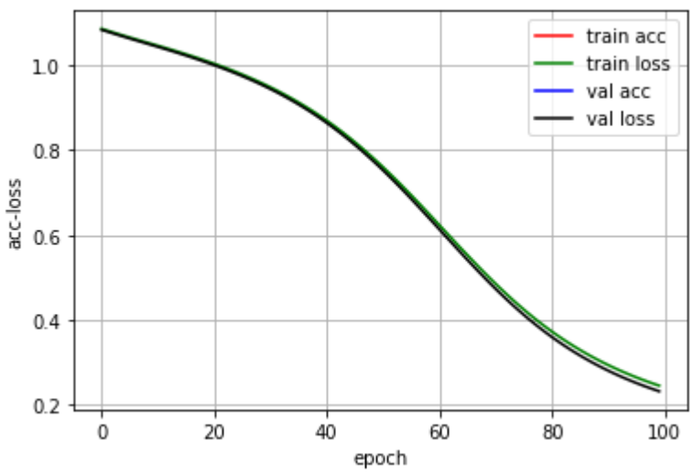
Out[62]: `<keras.callbacks.callbacks.History at 0x7f18d01249e8>`

- After 100 epoches, accuracy of train set is 0.9554, accuracy of val set is 0.9603.

1. (5 points) Plot the training loss against the validation loss. Do you observe overfitting/underfitting?

> **HINT** Explore the return value in the previous step.

In [63]:
```
# Your solution here
history.loss_plot('epoch')
```



- There is no over-fitting or under-fitting. The training set loss and the validation set loss are basically the same.

Evaluation (5 points)

1. (1 point) Make prediciton of the model on the test set

In [66]:
```
# Your solution here

score = model.evaluate(test_x, test_y, verbose=0)
print('accuracy of test set is: ',score[1])
```

```
accuracy of test set is:  0.9586227536201477
```

- Accuracy of the test set and accuracy of the validation set are basically the same.

1. (4 points) Compute the confusion matrix and the accuracy. Which classes confused most often?

> The model should have at least 90% accuracy.

In [75]:
```
# Your solution here

from sklearn.metrics import confusion_matrix,accuracy_score
y_pred = model.predict_classes(test_x)

y_true = new_filter_y[19000:]
# for i in range(test_y.shape[0]):
print('accuracy is: ', accuracy_score(y_pred, y_true))
print('confusion matrix is:  \n')
print(confusion_matrix(y_pred, y_true))
```

```
accuracy is:  0.9586227725762609
confusion matrix is:

[[1171   20   42]
 [  11  999   30]
 [   0   34 1004]]
```

- Category 1 predicts the best results, while category 3 and category 7 do not perform well

Improvements (10 points)

Suggest an improvement to the model? Implement it and compare to the one above. How to do robust comparison of the performance?

Explain Here

In [77]:

```
# Your solution here

### Using adam optimizer to imporove it.

model = Sequential([
    Conv2D(6, (5, 5), padding = 'same',input_shape = (28, 28, 1)),
    Activation('relu'),
    MaxPool2D(pool_size = 2),
    Conv2D(16, (5,5), padding = 'same'),
    Activation('relu'),
    MaxPool2D(pool_size = 2),
    Flatten(),
    Dense(128),
    Activation('relu'),
    Dense(3),
    Activation('softmax')
])

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'],
              )
model.fit(train_x, train_y, batch_size=128, epochs = 100, verbose=1,
validation_data=(val_x, val_y),
          callbacks=[history])
```

```
Train on 16000 samples, validate on 3000 samples
Epoch 1/100
16000/16000 [==============================] - 2s 129us/step - loss:
0.1344 - accuracy: 0.9554 - val_loss: 0.0233 - val_accuracy: 0.9907
Epoch 2/100
16000/16000 [==============================] - 2s 118us/step - loss:
0.0255 - accuracy: 0.9924 - val_loss: 0.0144 - val_accuracy: 0.9953
Epoch 3/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.0155 - accuracy: 0.9950 - val_loss: 0.0115 - val_accuracy: 0.9963
Epoch 4/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.0138 - accuracy: 0.9953 - val_loss: 0.0075 - val_accuracy: 0.9963
Epoch 5/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.0118 - accuracy: 0.9959 - val_loss: 0.0093 - val_accuracy: 0.9973
Epoch 6/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.0102 - accuracy: 0.9968 - val_loss: 0.0073 - val_accuracy: 0.9980
Epoch 7/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.0064 - accuracy: 0.9984 - val_loss: 0.0106 - val_accuracy: 0.9980
Epoch 8/100
16000/16000 [==============================] - 2s 119us/step - loss:
0.0060 - accuracy: 0.9976 - val_loss: 0.0094 - val_accuracy: 0.9973
Epoch 9/100
16000/16000 [==============================] - 2s 112us/step - loss:
0.0061 - accuracy: 0.9976 - val_loss: 0.0096 - val_accuracy: 0.9960
Epoch 10/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.0050 - accuracy: 0.9983 - val_loss: 0.0093 - val_accuracy: 0.9967
Epoch 11/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.0037 - accuracy: 0.9987 - val_loss: 0.0066 - val_accuracy: 0.9980
Epoch 12/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.0026 - accuracy: 0.9991 - val_loss: 0.0071 - val_accuracy: 0.9977
Epoch 13/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.0029 - accuracy: 0.9989 - val_loss: 0.0048 - val_accuracy: 0.9983
Epoch 14/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.0030 - accuracy: 0.9989 - val_loss: 0.0059 - val_accuracy: 0.9977
Epoch 15/100
16000/16000 [==============================] - 2s 118us/step - loss:
0.0022 - accuracy: 0.9994 - val_loss: 0.0107 - val_accuracy: 0.9963
Epoch 16/100
16000/16000 [==============================] - 2s 113us/step - loss:
0.0031 - accuracy: 0.9988 - val_loss: 0.0091 - val_accuracy: 0.9977
Epoch 17/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.0022 - accuracy: 0.9994 - val_loss: 0.0091 - val_accuracy: 0.9963
Epoch 18/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.0030 - accuracy: 0.9987 - val_loss: 0.0054 - val_accuracy: 0.9987
Epoch 19/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.0026 - accuracy: 0.9989 - val_loss: 0.0144 - val_accuracy: 0.9950
Epoch 20/100
16000/16000 [==============================] - 2s 116us/step - loss:
0.0021 - accuracy: 0.9994 - val_loss: 0.0093 - val_accuracy: 0.9977
Epoch 21/100
16000/16000 [==============================] - 2s 114us/step - loss:
0.0016 - accuracy: 0.9995 - val_loss: 0.0059 - val_accuracy: 0.9973
Epoch 22/100
16000/16000 [==============================] - 2s 115us/step - loss:
0.0011 - accuracy: 0.9996 - val_loss: 0.0062 - val_accuracy: 0.9983
Epoch 23/100
16000/16000 [==============================] - 2s 112us/step - loss:
6.4879e-04 - accuracy: 0.9998 - val_loss: 0.0076 - val_accuracy: 0.99
70
Epoch 24/100
16000/16000 [==============================] - 2s 113us/step - loss:
4.3641e-04 - accuracy: 0.9998 - val_loss: 0.0071 - val_accuracy: 0.99
```

```
70
Epoch 25/100
16000/16000 [==============================] - 2s 116us/step - loss:
2.7684e-04 - accuracy: 1.0000 - val_loss: 0.0058 - val_accuracy: 0.99
80
Epoch 26/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.5674e-04 - accuracy: 1.0000 - val_loss: 0.0083 - val_accuracy: 0.99
70
Epoch 27/100
16000/16000 [==============================] - 2s 115us/step - loss:
1.7436e-04 - accuracy: 1.0000 - val_loss: 0.0061 - val_accuracy: 0.99
80
Epoch 28/100
16000/16000 [==============================] - 2s 115us/step - loss:
9.5336e-05 - accuracy: 1.0000 - val_loss: 0.0066 - val_accuracy: 0.99
80
Epoch 29/100
16000/16000 [==============================] - 2s 111us/step - loss:
1.2212e-04 - accuracy: 1.0000 - val_loss: 0.0080 - val_accuracy: 0.99
80
Epoch 30/100
16000/16000 [==============================] - 2s 112us/step - loss:
9.8507e-05 - accuracy: 1.0000 - val_loss: 0.0076 - val_accuracy: 0.99
77
Epoch 31/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.0115e-04 - accuracy: 1.0000 - val_loss: 0.0072 - val_accuracy: 0.99
80
Epoch 32/100
16000/16000 [==============================] - 2s 115us/step - loss:
5.5586e-05 - accuracy: 1.0000 - val_loss: 0.0078 - val_accuracy: 0.99
77
Epoch 33/100
16000/16000 [==============================] - 2s 115us/step - loss:
5.0957e-05 - accuracy: 1.0000 - val_loss: 0.0072 - val_accuracy: 0.99
80
Epoch 34/100
16000/16000 [==============================] - 2s 114us/step - loss:
5.0908e-05 - accuracy: 1.0000 - val_loss: 0.0069 - val_accuracy: 0.99
83
Epoch 35/100
16000/16000 [==============================] - 2s 115us/step - loss:
4.2077e-05 - accuracy: 1.0000 - val_loss: 0.0073 - val_accuracy: 0.99
80
Epoch 36/100
16000/16000 [==============================] - 2s 113us/step - loss:
3.3858e-05 - accuracy: 1.0000 - val_loss: 0.0081 - val_accuracy: 0.99
80
Epoch 37/100
16000/16000 [==============================] - 2s 114us/step - loss:
3.1374e-05 - accuracy: 1.0000 - val_loss: 0.0080 - val_accuracy: 0.99
80
Epoch 38/100
16000/16000 [==============================] - 2s 115us/step - loss:
3.0444e-05 - accuracy: 1.0000 - val_loss: 0.0075 - val_accuracy: 0.99
80
Epoch 39/100
16000/16000 [==============================] - 2s 115us/step - loss:
3.6105e-05 - accuracy: 1.0000 - val_loss: 0.0082 - val_accuracy: 0.99
80
Epoch 40/100
16000/16000 [==============================] - 2s 115us/step - loss:
2.3768e-05 - accuracy: 1.0000 - val_loss: 0.0080 - val_accuracy: 0.99
80
Epoch 41/100
16000/16000 [==============================] - 2s 111us/step - loss:
2.2312e-05 - accuracy: 1.0000 - val_loss: 0.0086 - val_accuracy: 0.99
80
Epoch 42/100
16000/16000 [==============================] - 2s 114us/step - loss:
2.2973e-05 - accuracy: 1.0000 - val_loss: 0.0088 - val_accuracy: 0.99
77
Epoch 43/100
```

```
16000/16000 [==============================] - 2s 112us/step - loss:
1.7754e-05 - accuracy: 1.0000 - val_loss: 0.0083 - val_accuracy: 0.99
80
Epoch 44/100
16000/16000 [==============================] - 2s 111us/step - loss:
1.8525e-05 - accuracy: 1.0000 - val_loss: 0.0091 - val_accuracy: 0.99
77
Epoch 45/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.6037e-05 - accuracy: 1.0000 - val_loss: 0.0085 - val_accuracy: 0.99
80
Epoch 46/100
16000/16000 [==============================] - 2s 112us/step - loss:
1.4553e-05 - accuracy: 1.0000 - val_loss: 0.0093 - val_accuracy: 0.99
77
Epoch 47/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.2586e-05 - accuracy: 1.0000 - val_loss: 0.0089 - val_accuracy: 0.99
77
Epoch 48/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.2299e-05 - accuracy: 1.0000 - val_loss: 0.0089 - val_accuracy: 0.99
77
Epoch 49/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.1757e-05 - accuracy: 1.0000 - val_loss: 0.0090 - val_accuracy: 0.99
80
Epoch 50/100
16000/16000 [==============================] - 2s 113us/step - loss:
9.8572e-06 - accuracy: 1.0000 - val_loss: 0.0088 - val_accuracy: 0.99
80
Epoch 51/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.0539e-05 - accuracy: 1.0000 - val_loss: 0.0091 - val_accuracy: 0.99
80
Epoch 52/100
16000/16000 [==============================] - 2s 114us/step - loss:
9.9855e-06 - accuracy: 1.0000 - val_loss: 0.0097 - val_accuracy: 0.99
77
Epoch 53/100
16000/16000 [==============================] - 2s 113us/step - loss:
7.8869e-06 - accuracy: 1.0000 - val_loss: 0.0096 - val_accuracy: 0.99
77
Epoch 54/100
16000/16000 [==============================] - 2s 112us/step - loss:
7.2740e-06 - accuracy: 1.0000 - val_loss: 0.0096 - val_accuracy: 0.99
77
Epoch 55/100
16000/16000 [==============================] - 2s 115us/step - loss:
6.7942e-06 - accuracy: 1.0000 - val_loss: 0.0099 - val_accuracy: 0.99
77
Epoch 56/100
16000/16000 [==============================] - 2s 112us/step - loss:
6.5223e-06 - accuracy: 1.0000 - val_loss: 0.0099 - val_accuracy: 0.99
77
Epoch 57/100
16000/16000 [==============================] - 2s 113us/step - loss:
5.6711e-06 - accuracy: 1.0000 - val_loss: 0.0094 - val_accuracy: 0.99
80
Epoch 58/100
16000/16000 [==============================] - 2s 114us/step - loss:
5.7438e-06 - accuracy: 1.0000 - val_loss: 0.0102 - val_accuracy: 0.99
77
Epoch 59/100
16000/16000 [==============================] - 2s 115us/step - loss:
5.4367e-06 - accuracy: 1.0000 - val_loss: 0.0107 - val_accuracy: 0.99
77
Epoch 60/100
16000/16000 [==============================] - 2s 111us/step - loss:
4.6490e-06 - accuracy: 1.0000 - val_loss: 0.0102 - val_accuracy: 0.99
80
Epoch 61/100
16000/16000 [==============================] - 2s 114us/step - loss:
4.4434e-06 - accuracy: 1.0000 - val_loss: 0.0108 - val_accuracy: 0.99
```

```
77
Epoch 62/100
16000/16000 [==============================] - 2s 113us/step - loss:
4.4495e-06 - accuracy: 1.0000 - val_loss: 0.0108 - val_accuracy: 0.99
77
Epoch 63/100
16000/16000 [==============================] - 2s 114us/step - loss:
3.7061e-06 - accuracy: 1.0000 - val_loss: 0.0106 - val_accuracy: 0.99
77
Epoch 64/100
16000/16000 [==============================] - 2s 112us/step - loss:
3.4490e-06 - accuracy: 1.0000 - val_loss: 0.0099 - val_accuracy: 0.99
80
Epoch 65/100
16000/16000 [==============================] - 2s 114us/step - loss:
3.8541e-06 - accuracy: 1.0000 - val_loss: 0.0104 - val_accuracy: 0.99
77
Epoch 66/100
16000/16000 [==============================] - 2s 113us/step - loss:
3.1306e-06 - accuracy: 1.0000 - val_loss: 0.0097 - val_accuracy: 0.99
80
Epoch 67/100
16000/16000 [==============================] - 2s 111us/step - loss:
3.4126e-06 - accuracy: 1.0000 - val_loss: 0.0110 - val_accuracy: 0.99
77
Epoch 68/100
16000/16000 [==============================] - 2s 112us/step - loss:
2.5280e-06 - accuracy: 1.0000 - val_loss: 0.0102 - val_accuracy: 0.99
80
Epoch 69/100
16000/16000 [==============================] - 2s 112us/step - loss:
2.6590e-06 - accuracy: 1.0000 - val_loss: 0.0123 - val_accuracy: 0.99
73
Epoch 70/100
16000/16000 [==============================] - 2s 114us/step - loss:
3.1754e-06 - accuracy: 1.0000 - val_loss: 0.0112 - val_accuracy: 0.99
77
Epoch 71/100
16000/16000 [==============================] - 2s 113us/step - loss:
2.2370e-06 - accuracy: 1.0000 - val_loss: 0.0113 - val_accuracy: 0.99
77
Epoch 72/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.9468e-06 - accuracy: 1.0000 - val_loss: 0.0113 - val_accuracy: 0.99
77
Epoch 73/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.6742e-06 - accuracy: 1.0000 - val_loss: 0.0114 - val_accuracy: 0.99
77
Epoch 74/100
16000/16000 [==============================] - 2s 116us/step - loss:
1.6266e-06 - accuracy: 1.0000 - val_loss: 0.0114 - val_accuracy: 0.99
77
Epoch 75/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.7532e-06 - accuracy: 1.0000 - val_loss: 0.0119 - val_accuracy: 0.99
77
Epoch 76/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.5370e-06 - accuracy: 1.0000 - val_loss: 0.0124 - val_accuracy: 0.99
73
Epoch 77/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.2537e-06 - accuracy: 1.0000 - val_loss: 0.0118 - val_accuracy: 0.99
77
Epoch 78/100
16000/16000 [==============================] - 2s 113us/step - loss:
1.2307e-06 - accuracy: 1.0000 - val_loss: 0.0119 - val_accuracy: 0.99
77
Epoch 79/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.1469e-06 - accuracy: 1.0000 - val_loss: 0.0117 - val_accuracy: 0.99
77
Epoch 80/100
```

```
16000/16000 [==============================] - 2s 113us/step - loss:
1.0613e-06 - accuracy: 1.0000 - val_loss: 0.0120 - val_accuracy: 0.99
77
Epoch 81/100
16000/16000 [==============================] - 2s 113us/step - loss:
9.8778e-07 - accuracy: 1.0000 - val_loss: 0.0122 - val_accuracy: 0.99
77
Epoch 82/100
16000/16000 [==============================] - 2s 114us/step - loss:
1.0134e-06 - accuracy: 1.0000 - val_loss: 0.0121 - val_accuracy: 0.99
77
Epoch 83/100
16000/16000 [==============================] - 2s 115us/step - loss:
8.7467e-07 - accuracy: 1.0000 - val_loss: 0.0124 - val_accuracy: 0.99
77
Epoch 84/100
16000/16000 [==============================] - 2s 115us/step - loss:
8.1444e-07 - accuracy: 1.0000 - val_loss: 0.0124 - val_accuracy: 0.99
77
Epoch 85/100
16000/16000 [==============================] - 2s 115us/step - loss:
7.9718e-07 - accuracy: 1.0000 - val_loss: 0.0130 - val_accuracy: 0.99
77
Epoch 86/100
16000/16000 [==============================] - 2s 113us/step - loss:
7.6982e-07 - accuracy: 1.0000 - val_loss: 0.0139 - val_accuracy: 0.99
73
Epoch 87/100
16000/16000 [==============================] - 2s 114us/step - loss:
8.5546e-07 - accuracy: 1.0000 - val_loss: 0.0130 - val_accuracy: 0.99
77
Epoch 88/100
16000/16000 [==============================] - 2s 113us/step - loss:
6.5955e-07 - accuracy: 1.0000 - val_loss: 0.0126 - val_accuracy: 0.99
77
Epoch 89/100
16000/16000 [==============================] - 2s 112us/step - loss:
6.1268e-07 - accuracy: 1.0000 - val_loss: 0.0129 - val_accuracy: 0.99
77
Epoch 90/100
16000/16000 [==============================] - 2s 112us/step - loss:
5.7113e-07 - accuracy: 1.0000 - val_loss: 0.0128 - val_accuracy: 0.99
77
Epoch 91/100
16000/16000 [==============================] - 2s 114us/step - loss:
5.4187e-07 - accuracy: 1.0000 - val_loss: 0.0128 - val_accuracy: 0.99
77
Epoch 92/100
16000/16000 [==============================] - 2s 114us/step - loss:
5.0874e-07 - accuracy: 1.0000 - val_loss: 0.0126 - val_accuracy: 0.99
80
Epoch 93/100
16000/16000 [==============================] - 2s 114us/step - loss:
5.0814e-07 - accuracy: 1.0000 - val_loss: 0.0134 - val_accuracy: 0.99
77
Epoch 94/100
16000/16000 [==============================] - 2s 112us/step - loss:
4.6539e-07 - accuracy: 1.0000 - val_loss: 0.0133 - val_accuracy: 0.99
77
Epoch 95/100
16000/16000 [==============================] - 2s 114us/step - loss:
4.7282e-07 - accuracy: 1.0000 - val_loss: 0.0137 - val_accuracy: 0.99
77
Epoch 96/100
16000/16000 [==============================] - 2s 111us/step - loss:
4.1125e-07 - accuracy: 1.0000 - val_loss: 0.0132 - val_accuracy: 0.99
77
Epoch 97/100
16000/16000 [==============================] - 2s 112us/step - loss:
3.9614e-07 - accuracy: 1.0000 - val_loss: 0.0135 - val_accuracy: 0.99
77
Epoch 98/100
16000/16000 [==============================] - 2s 113us/step - loss:
3.8000e-07 - accuracy: 1.0000 - val_loss: 0.0137 - val_accuracy: 0.99
```

```
77
Epoch 99/100
16000/16000 [==============================] - 2s 113us/step - loss:
3.8132e-07 - accuracy: 1.0000 - val_loss: 0.0143 - val_accuracy: 0.99
73
Epoch 100/100
16000/16000 [==============================] - 2s 112us/step - loss:
3.2718e-07 - accuracy: 1.0000 - val_loss: 0.0139 - val_accuracy: 0.99
77
```

Out[77]:
```
<keras.callbacks.callbacks.History at 0x7f17c0751710>
```

In [78]:
```
score = model.evaluate(test_x, test_y, verbose=0)
print('accuracy of test set is: ',score[1])
```

```
accuracy of test set is:  0.9984898567199707
```

- Model performance has improved significantly with better optimizers

How to do robust performance comparison

1. Model performance evaluation indicators generally include accuracy, precision, recall, F1 value, and comparison of running time. Generally, they tend to train more efficient models, and they also compare parameters. model.
2. Here is the three classification task, which is a classification model, so the accuracy index is mainly used to evaluate the model.
3. Generally, the performance of the model on the training set, the validation set, and the test set is also evaluated to observe the generalization ability of the model.
4. At present, in this task, the performance of the better optimizer model has been further improved, and a better optimizer can find the optimal solution of the model faster.
5. In order to further improve the accuracy of the model, more effective picture features can be used as input to the model, and the network structure parameters can be adjusted.

Graph ML [40 points]

This set of assingments will teach you the differences between various node representations in graphs. Note that all questions are programming assingments but you do not need to use loss function to optimize the calculation of thesee embeddings.

1- (5 points) Write a function randadjmat(n,p) in Python which returns an adjacency matrix for a "random graph" on n vertices. Here p is the probability of having an edge between any pair of vertices.

I assume that the graph is undirected and no node is connected to itself. The probability works as follows:

- Generate a random float between 0 and 1
- If it is smaller than P, then the two nodes connect; otherwise the two nodes do not connect

> For a float between [0, 1], the probability of it falling inside a defined range [0, P<=1] is P. For example, if P = 1, the point is 100% likely to be in between this range; if P = 0.1, the point is 10% likely to be in the range as (0.1-0)/(1-0) = 10%

Here p is defined as a 'global threshold', whereby it sets a cutoff of whether an edge is ok to exist or not. Say if p = 0.4, then for a newly generated random number, it needs to have a 40% chance of lying in the range so as to have an edge (equal to 1). To do so,

$$\frac{0.4 - 0}{1.0 - 1} = 0.4$$

This interpretation follows the definition of Erdos-Renyi's G(n,p) random graph model(https://www.cs.cmu.edu/~avrim/598/chap4only.pdf (https://www.cs.cmu.edu/~avrim/598/chap4only.pdf)), whereby:

"For each pair of distinct vertices, v and w, p is the probability that the edge (v,w) is present. The presence of each edge is statistically independent of all other edges. The graph-valued random variable with these parameters is denoted by G (n, p) italicized text"

The following code are constructed based on this interpretation. The remaining questions are based on this version.

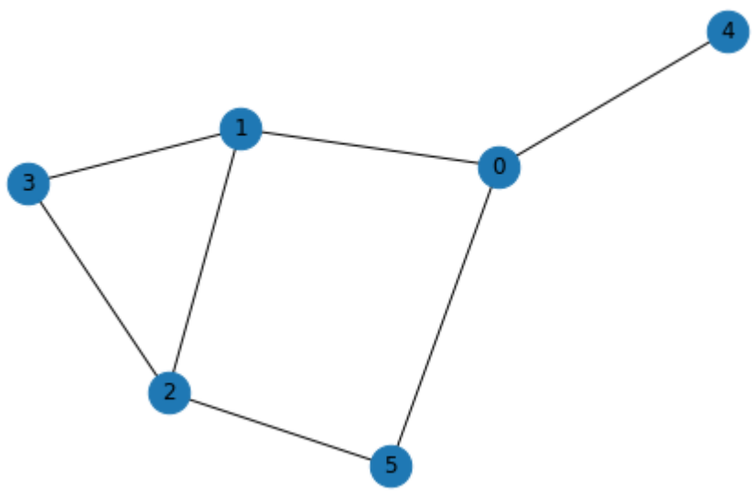
In [0]:
```
import numpy as np
import random
```

In [0]:
```
def randadjmat(n,p):

  A = np.zeros(shape=(n,n))

  for i in range(0, n):
    A[i][i] = 0 # no node is connected to itself
    for j in range(i+1, n): # go from the 'next' node to the end
      if random.random() > p:
        A[i][j] = 0
      else:
        A[i][j] = 1
      A[j][i] = A[i][j] # undirected graph, edges are two-way
  # enforce sum-zero checking
  for i in range(0, n):
    if np.sum(A, axis=1)[i] == 0:
      randadjmat(n, p)

  return A
```

In [53]:
```
# unit test with 5 nodes and p = 0.5
n=6
p=0.5
A = randadjmat(n, p)
print("The adjacency matrix for a random graph of 5 nodes and P = 0.5
is:\n")
print(A)
```

```
The adjacency matrix for a random graph of 5 nodes and P = 0.5 is:

[[0. 1. 0. 0. 1. 1.]
 [1. 0. 1. 1. 0. 0.]
 [0. 1. 0. 1. 0. 1.]
 [0. 1. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0.]
 [1. 0. 1. 0. 0. 0.]]
```

In [54]:
```
import matplotlib.pyplot as plt
import networkx as nx

def show_graph_with_labels(adjacency_matrix):
    rows, cols = np.where(adjacency_matrix == 1)
    edges = zip(rows.tolist(), cols.tolist())
    gr = nx.Graph()
    gr.add_edges_from(edges)
    nx.draw(gr, node_size=500, with_labels=True)
    plt.show()
# plot the graph
show_graph_with_labels(A)
```



Of course, there are other ways to interpret. For example, for a graph with *n* nodes, there are a total of $\frac{n(n-1)}{2}$ edges possible. If the probability of two arbitrary nodes having an edge is p, there are $\frac{pn(n-1)}{2}$ edges total in the graph. This means that there are no individual independnce between pairings.This way, what we need is then to enforce another check based on number of edges.

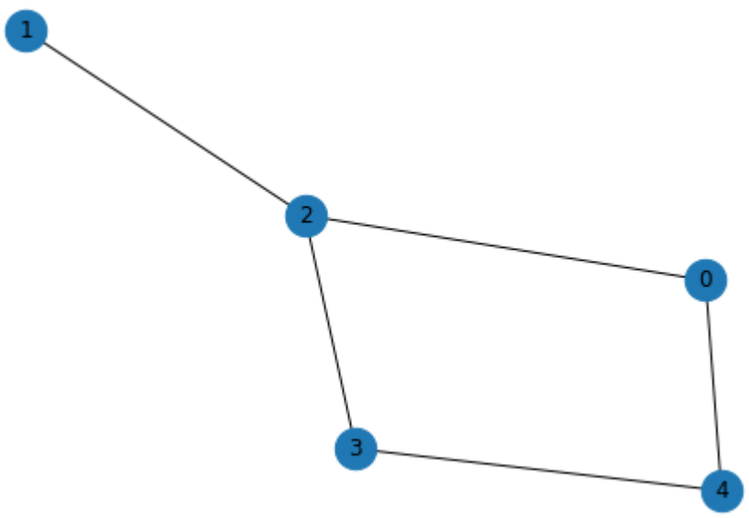
Please kindly note that the the following snippet is only to demonstrate, and that it is not used for the remaining questions.

In [0]:
```
def randadjmat_alt(n,p):

  A = np.zeros(shape=(n,n))
  edges = p * n * (n - 1) * 0.5

  for i in range(0, n):
    A[i][i] = 0 # no node is connected to itself
    for j in range(i+1, n): # go from the 'next' node to the end
      if random.random() > p:
        A[i][j] = 0
      else:
        A[i][j] = 1
      A[j][i] = A[i][j] # undirected graph, edges are two-way

  # enforce sum-zero checking and edge number checking
  for i in range(0, n):
    if np.sum(A, axis=1)[i] == 0 or np.sum(A, axis=1).sum(axis=0) !=
edges:
      randadjmat(n, p)

  return A
```

In [97]:
```
# unit test with 5 nodes and p = 0.5, should have 0.5 * 5 * 4 * 0.5 =
5 edges
n=5
p=0.5
A_alt = randadjmat_alt(n, p)
print("The adjacency matrix for a random graph of 5 nodes and P = 0.5
is:\n")
print(A_alt)
# We indeed have 5 edges for 5 nodes [0, 1, 2, 3, 4]: 0-2, 0-4, 1-2,
 2-3, 3-4
```

```
The adjacency matrix for a random graph of 5 nodes and P = 0.5 is:

[[0. 0. 1. 0. 1.]
 [0. 0. 1. 0. 0.]
 [1. 1. 0. 1. 0.]
 [0. 0. 1. 0. 1.]
 [1. 0. 0. 1. 0.]]
```

In [98]:
```
show_graph_with_labels(A_alt)
```



2- (5 points) Write a function transionmat(A) which, given an adjacency matrix A, generate a transition matrix T where probability of each edge (u,v) is calculated as $1/degree(u)$.

In [0]:
```
def degreemat(A):

  D = np.zeros(shape=(n,n))
  for i in range(0, n):
    D[i][i] =  np.sum(A, axis=1)[i]
  return D

def laplacianmat(A, D):
  return D - A
```

In [56]:
```
#unit test
D = degreemat(A)
L = laplacianmat(A, D)
print("The degree matrix for a random graph with adjacency matrix A i
s:\n")
print(D)
print("\nThe Laplacian matrix for a random graph with adjacency matri
x A is:\n")
print(L)
```

```
The degree matrix for a random graph with adjacency matrix A is:

[[3. 0. 0. 0. 0. 0.]
 [0. 3. 0. 0. 0. 0.]
 [0. 0. 3. 0. 0. 0.]
 [0. 0. 0. 2. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 2.]]

The Laplacian matrix for a random graph with adjacency matrix A is:

[[ 3. -1.  0.  0. -1. -1.]
 [-1.  3. -1. -1.  0.  0.]
 [ 0. -1.  3. -1.  0. -1.]
 [ 0. -1. -1.  2.  0.  0.]
 [-1.  0.  0.  0.  1.  0.]
 [-1.  0. -1.  0.  0.  2.]]
```

Reference: Slide P65, Graph Learning Part 1 (https://ift6758.github.io/lectures/graph_learning_part1.pdf (https://ift6758.github.io/lectures/graph_learning_part1.pdf))

In [0]:
```
def transionmat(A):

    T = np.zeros(shape=(n,n))
    for i in range(0, n):
      if np.sum(A, axis=1)[i] > 0:
        T[i] = A[i] / np.sum(A, axis=1)[i]
      else:
        # if a node is orphan
        T[i] = A[i]
    return T
```

In [58]:
```
#unit test
T = transionmat(A)
print("The transition matrix for a random graph with adjacency matrix
A is:\n")
print(T)
```

```
The transition matrix for a random graph with adjacency matrix A is:

[[0.         0.33333333 0.         0.         0.33333333 0.33333333]
 [0.33333333 0.         0.33333333 0.33333333 0.         0.        ]
 [0.         0.33333333 0.         0.33333333 0.         0.33333333]
 [0.         0.5        0.5        0.         0.         0.        ]
 [1.         0.         0.         0.         0.         0.        ]
 [0.5        0.         0.5        0.         0.         0.        ]]
```

3- (5 points) Write a function hotembd(A) which, given an adjacency matrix A, generate an embedding matrix H where each node is represetned with a 1-hot vector.

In [0]:
```
def hotembd(A):

    H = np.zeros(shape=(n,n))
    for node in range(0, n):
      vect = np.zeros(shape=(n,)) # temp vector to be encoded
      vect[node] = 1 # encode the node
      H[node] = vect # update the matrix with the new encoded vector

    return H
```

In [60]:
```
#unit test
H = hotembd(A)
print("The 1-hot embedding matrix for a random graph with adjacency m
atrix A is:\n")
print(H)
```

```
The 1-hot embedding matrix for a random graph with adjacency matrix A
is:

[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]]
```

4- (5 points) Write a function randwalkemb(A,k) which, given an adjacency matrix A, a transition matrix T, and one-hot encoding H, performs random walks (https://en.wikipedia.org/wiki/Random_walk) on the graph from each node w times with length equal to l and generate an embedding matrix for each node based on the sum of 1-hot encodings of all nodes that are visited during the walks.

In [0]:
```
def randwalkembd(A, T, H, w, l):

    # initialize
    rand_walk_embd = np.zeros(A.shape)
    n_nodes,_ = A.shape

    # looping
    w_num = 0
    for i in range(n_nodes):
        w_num = 0
        while(w_num<=w):
            w_num += 1
            l_num = 0
            new_v = i
            while(l_num<=l):
                new_v = np.random.choice(list(range(0,n_nodes)), p =
T[new_v].tolist())
                rand_walk_embd[i] += H[new_v]
                l_num += 1

    return rand_walk_embd
```

In [62]:
```
# unit testing
RandWalkEmbedding = randwalkembd(A, T, H, w = 3, l = 2)
print("The random walk embedding matrix for a random graph with adjac
ency matrix A is:\n")
print(RandWalkEmbedding)
```

```
The random walk embedding matrix for a random graph with adjacency ma
trix A is:

[[3. 3. 1. 0. 0. 5.]
 [1. 2. 4. 4. 0. 1.]
 [2. 3. 1. 2. 2. 2.]
 [0. 5. 4. 2. 0. 1.]
 [6. 0. 2. 0. 1. 3.]
 [4. 0. 4. 0. 1. 3.]]
```

5- (5 points) Write a function hopeneighbormbd(A,H,k) which, given an adjacency matrix A, and one-hot node encoding matrix H, generates node embedding matrix which represents each node as sum of 1-hot encodings of k-hobs neighbors.

This question is a very ambiguous. At every hop, we are presumably allowed to hop 'back' to the same node if it is the only possible path.

> However, thinking of it as 'expanding' a tree, even if we expand(hop) to the same point at two different parent (k-1) nodes, the children are not the same node (take by reference, not value) even if they are equal in value. So we should count the number of appearance of each node, and this way we reflect 'count-by-reference' principle. This approach is carried over to later problems.

In [0]:
```
def hopeneighbormbd(A, H, k):

    NeighbourEmd = np.zeros(shape=(n,n))
    num = np.linalg.matrix_power(A, k)

    # to get the sum of all one-hot encodings, by matrix multiplicati
on of
    # number of times nodes are visited and the corresponding one-hot
matrix
    for i in range(0, len(num)):
      NeighbourEmd[i] = np.matmul(np.transpose(num[i]), H)

    return NeighbourEmd
```

In [64]:
```
# unit testing
NeighbourEmd = hopeneighbormbd(A, H, 3)
print("The k-hop embedding matrix for a random graph with adjacency m
atrix A is:\n")
print(NeighbourEmd)
```

```
The k-hop embedding matrix for a random graph with adjacency matrix A
is:

[[0. 6. 1. 2. 3. 5.]
 [6. 2. 6. 4. 0. 1.]
 [1. 6. 2. 4. 2. 5.]
 [2. 4. 4. 2. 1. 2.]
 [3. 0. 2. 1. 0. 0.]
 [5. 1. 5. 2. 0. 0.]]
```

Another interpretation is that if a node had been traveled no less than once, it is 'travelled' and that we should apply boolean values to whether if a node is travelled. Any non-zero entries in the matrix indicated 'true' and should be 1.

In [0]:
```
def hopeneighbormbd_alt(A, H, k):

    NeighbourEmdAlt = np.zeros(shape=(n,n))
    num = np.linalg.matrix_power(A, k)

    for i in range(0, len(num)):
      NeighbourEmdAlt[i] = np.matmul(np.transpose(num[i]), H)
      for j in range(0, len(NeighbourEmdAlt)):
        # if a node is visited no less than once, just put 1, i.e. 'v
isited'
        if NeighbourEmdAlt[i][j] != 0:
          NeighbourEmdAlt[i][j] = 1

    return NeighbourEmdAlt
```

In [66]:
```
# unit testing
NeighbourEmdAlt = hopeneighbormbd_alt(A, H, 3)
print("The k-hop embedding matrix for a random graph with adjacency m
atrix A is:\n")
print(NeighbourEmdAlt)
```

```
The k-hop embedding matrix for a random graph with adjacency matrix A
is:

[[0. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 0. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 0. 1. 1. 0. 0.]
 [1. 1. 1. 1. 0. 0.]]
```

6- (5 points) Write a function similarnodes(Z) which, given an node embedding matrix, find the most similar nodes in the graph.

In [0]:
```
def similarnodes(Z):

    n_nodes,_ = Z.shape
    max_similarity = 0
    max_u = 0
    max_v = 0
    for i in range(n_nodes):
        for j in range(i+1, n_nodes):
            # cosine similarity
            dot_product = (np.dot(Z[i], Z[j]))/(np.linalg.norm(Z[i])
* np.linalg.norm(Z[i]))
            if dot_product > max_similarity:
                max_similarity = dot_product
                max_u = i
                max_v = j

    return max_u, max_v
```

In [68]:
```
# unit testing
Embd = similarnodes(NeighbourEmd)
print("The pair of similar nodes using k-hop embedding matrix is:\n")
print(Embd)
```

```
The pair of similar nodes using k-hop embedding matrix is:

(4, 5)
```

7- (10 points) generate a random graph where n=20, and p=0.6, and compare the most similar nodes in the graph using randwalkembd (l=4, w=10), hopeneighbormbd (k=1) and hopeneighbormbd (k=2). Justify why similar nodes are different using different node embeddings?

In [0]:
```
# Generate matrices for random graph
n = 20
p = 0.6
A_new = randadjmat(n,p)
T_new = transionmat(A_new)
H_new = hotembd(A_new)
```

In [70]:
```
print("The Adjacency matrix A is: \n")
print(A_new)
print("\nThe Transition matrix T is: \n")
print(T_new)
print("\nThe one-hot encoding matrix H is: \n")
print(H_new)
```

The Adjacency matrix A is:

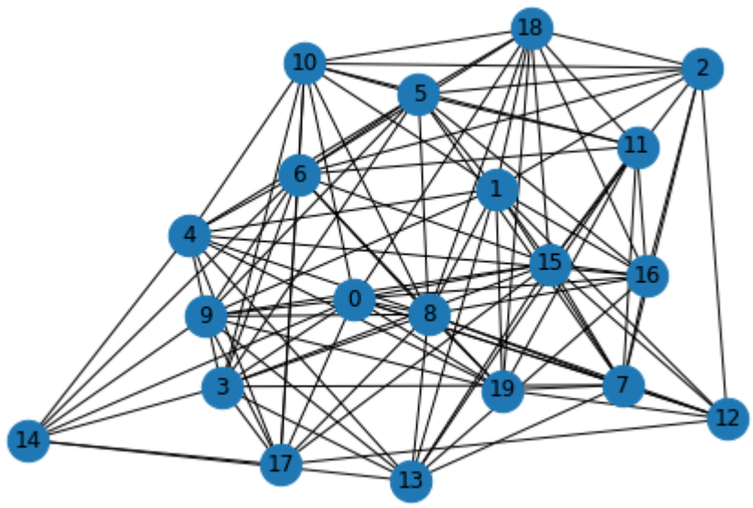
```
[[0. 0. 0. 1. 0. 0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1.]
 [0. 0. 1. 0. 1. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [0. 1. 0. 0. 0. 1. 1. 1. 0. 0. 1. 0. 1. 0. 0. 1. 1. 0. 1. 0.]
 [1. 0. 0. 0. 1. 1. 1. 1. 1. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 0.]
 [0. 1. 0. 1. 0. 1. 1. 0. 1. 0. 1. 0. 0. 1. 1. 0. 1. 1. 0. 1.]
 [0. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 1. 0. 1. 0.]
 [0. 0. 1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 1. 1. 0. 1. 1. 1.]
 [1. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 1. 0. 0. 1.]
 [1. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 1.]
 [1. 1. 0. 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 1.]
 [1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1. 0.]
 [0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 0. 0. 1. 0. 1. 1. 0. 1. 1.]
 [1. 1. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1.]
 [0. 1. 0. 1. 1. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 1. 0. 0. 0.]
 [1. 0. 0. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0.]
 [1. 1. 1. 1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 1. 0. 0. 1. 1. 1. 0.]
 [1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 0. 1. 0. 1. 0. 1. 0. 0. 1. 0.]
 [1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 1. 0. 1. 0. 1. 1. 0. 0. 0. 0.]
 [1. 1. 1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 0. 0. 0. 1. 1. 0. 0. 1.]
 [1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 0. 0. 0. 0. 1. 0.]]
```

The Transition matrix T is:

```
[[0.         0.         0.         0.08333333 0.         0.
  0.         0.08333333 0.08333333 0.08333333 0.08333333 0.
  0.08333333 0.         0.08333333 0.08333333 0.08333333 0.08333333
  0.08333333 0.08333333]
 [0.         0.         0.07692308 0.         0.07692308 0.07692308
  0.         0.07692308 0.07692308 0.07692308 0.07692308 0.
  0.07692308 0.07692308 0.         0.07692308 0.07692308 0.
  0.07692308 0.07692308]
 [0.         0.11111111 0.         0.         0.         0.11111111
  0.11111111 0.11111111 0.         0.         0.11111111 0.
  0.11111111 0.         0.         0.11111111 0.11111111 0.
  0.11111111 0.        ]
 [0.09090909 0.         0.         0.         0.09090909 0.09090909
  0.09090909 0.09090909 0.09090909 0.         0.09090909 0.
  0.         0.09090909 0.09090909 0.09090909 0.         0.09090909
  0.         0.        ]
 [0.         0.09090909 0.         0.09090909 0.         0.09090909
  0.09090909 0.         0.09090909 0.         0.09090909 0.
  0.         0.09090909 0.09090909 0.         0.09090909 0.09090909
  0.         0.09090909]
 [0.         0.08333333 0.08333333 0.08333333 0.08333333 0.
  0.08333333 0.08333333 0.08333333 0.08333333 0.08333333 0.08333333
  0.         0.         0.         0.         0.08333333 0.
  0.08333333 0.        ]
 [0.         0.         0.07692308 0.07692308 0.07692308 0.07692308
  0.         0.         0.07692308 0.07692308 0.07692308 0.07692308
  0.         0.         0.07692308 0.07692308 0.         0.07692308
  0.07692308 0.07692308]
 [0.08333333 0.08333333 0.08333333 0.08333333 0.         0.08333333
  0.         0.         0.08333333 0.         0.         0.08333333
  0.08333333 0.08333333 0.         0.08333333 0.08333333 0.
  0.         0.08333333]
 [0.06666667 0.06666667 0.         0.06666667 0.06666667 0.06666667
  0.06666667 0.06666667 0.         0.06666667 0.06666667 0.
  0.06666667 0.06666667 0.         0.         0.06666667 0.06666667
  0.06666667 0.06666667]
 [0.1        0.1        0.         0.         0.         0.1
  0.1        0.         0.1        0.         0.         0.
  0.         0.1        0.1        0.1        0.         0.1
  0.         0.1        0.        ]
 [0.09090909 0.09090909 0.09090909 0.09090909 0.09090909 0.09090909
  0.09090909 0.         0.09090909 0.         0.         0.09090909
  0.         0.         0.         0.         0.         0.09090909
  0.09090909 0.         0.        ]
 [0.         0.         0.         0.         0.         0.11111111
  0.11111111 0.11111111 0.         0.         0.11111111 0.
  0.         0.11111111 0.         0.11111111 0.11111111 0.
  0.11111111 0.11111111]
 [0.125      0.125      0.125      0.         0.         0.
```

```
  0.         0.125      0.125      0.         0.         0.
  0.         0.         0.         0.125      0.         0.125
  0.         0.125     ]
 [0.         0.1        0.         0.1        0.1        0.
  0.         0.1        0.1        0.1        0.         0.1
  0.         0.         0.1        0.1        0.1        0.
  0.         0.        ]
 [0.14285714 0.         0.         0.14285714 0.14285714 0.
  0.14285714 0.         0.         0.14285714 0.         0.
  0.         0.14285714 0.         0.         0.         0.14285714
  0.         0.        ]
 [0.07692308 0.07692308 0.07692308 0.07692308 0.         0.
  0.07692308 0.07692308 0.         0.07692308 0.         0.07692308
  0.07692308 0.07692308 0.         0.         0.07692308 0.07692308
  0.07692308 0.         ]
 [0.09090909 0.09090909 0.09090909 0.         0.09090909 0.09090909
  0.         0.09090909 0.09090909 0.         0.         0.09090909
  0.         0.09090909 0.         0.09090909 0.         0.
  0.09090909 0.        ]
 [0.1        0.         0.         0.1        0.1        0.
  0.1        0.         0.1        0.1        0.1        0.
  0.1        0.         0.1        0.1        0.         0.
  0.         0.        ]
 [0.09090909 0.09090909 0.09090909 0.         0.         0.09090909
  0.09090909 0.         0.09090909 0.         0.09090909 0.09090909
  0.         0.         0.         0.09090909 0.09090909 0.
  0.         0.09090909]
 [0.1        0.1        0.         0.         0.1        0.
  0.1        0.1        0.1        0.1        0.         0.1
  0.1        0.         0.         0.         0.         0.
  0.1        0.        ]]

The one-hot encoding matrix H is:

[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

In [71]:
```
show_graph_with_labels(A_new)
```



In [72]:
```
print("The random walk embedding matrix is: \n")
print(randwalkembd(A, T, H, w=10, l=4))
```

```
The random walk embedding matrix is:

[[10. 17. 10.  7.  2.  9.]
 [ 8. 12. 14. 14.  0.  7.]
 [ 6. 21. 14. 10.  1.  3.]
 [ 8. 18. 13. 10.  3.  3.]
 [28.  6.  1.  4. 11.  5.]
 [17.  9. 14.  2.  5.  8.]]
```

In [73]:
```
# random walk w=10, l=4
similarnodes(randwalkembd(A, T, H, w=10, l=4))
```

Out[73]:
```
(1, 4)
```

In [74]:
```
print("The 1-hop embedding matrix is: \n")
print(hopeneighbormbd(A_new, H_new, k=1))
```

```
The 1-hop embedding matrix is:

[[0. 0. 0. 1. 0. 0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1.]
 [0. 0. 1. 0. 1. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 1. 1.]
 [0. 1. 0. 0. 0. 1. 1. 1. 0. 0. 1. 0. 1. 0. 0. 1. 1. 0. 1. 0.]
 [1. 0. 0. 0. 1. 1. 1. 1. 1. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 0.]
 [0. 1. 0. 1. 0. 1. 1. 0. 1. 0. 1. 0. 0. 1. 1. 0. 1. 1. 0. 1.]
 [0. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 1. 0. 1. 0.]
 [0. 0. 1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 1. 1. 0. 1. 1. 1.]
 [1. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 1. 0. 0. 1.]
 [1. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 1.]
 [1. 1. 0. 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 1.]
 [1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1. 0.]
 [0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 0. 0. 1. 0. 1. 1. 0. 1. 1.]
 [1. 1. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1.]
 [0. 1. 0. 1. 1. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 1. 0. 0. 0.]
 [1. 0. 0. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0.]
 [1. 1. 1. 1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 1. 0. 0. 1. 1. 1. 0.]
 [1. 1. 1. 0. 1. 1. 0. 1. 1. 0. 0. 1. 0. 1. 0. 1. 0. 0. 1. 0.]
 [1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 1. 0. 1. 0. 1. 1. 0. 0. 0. 0.]
 [1. 1. 1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 0. 0. 0. 1. 1. 0. 0. 1.]
 [1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 0. 0. 0. 0. 1. 0.]]
```

In [75]:
```
# hop with k=1
similarnodes(hopeneighbormbd(A_new, H_new, k=1))
```

Out[75]:
```
(2, 8)
```

In [76]:
```
print("The 2-hop embedding matrix is: \n")
print(hopeneighbormbd(A_new, H_new, k=2))
```

```
The 2-hop embedding matrix is:

[[12.  9.  6.  6.  7.  7.  9.  6.  9.  5.  4.  6.  5.  7.  3.  7.  4.
7.
   5.  5.]
 [ 9. 13.  7.  7.  6.  8.  9.  8. 10.  5.  5.  8.  5.  6.  3.  7.  8.
6.
   7.  6.]
 [ 6.  7.  9.  5.  5.  6.  4.  5.  8.  4.  4.  7.  3.  4.  1.  6.  5.
4.
   6.  5.]
 [ 6.  7.  5. 11.  7.  5.  7.  5.  8.  8.  6.  6.  5.  5.  5.  5.  7.
7.
   6.  5.]
 [ 7.  6.  5.  7. 11.  6.  7.  7.  9.  8.  6.  6.  4.  5.  4.  6.  4.
5.
   7.  3.]
 [ 7.  8.  6.  5.  6. 12.  8.  6.  9.  3.  8.  5.  4.  8.  4.  9.  7.
6.
   7.  8.]
 [ 9.  9.  4.  7.  7.  8. 13.  7.  8.  6.  8.  5.  5.  7.  4.  6.  7.
7.
   7.  5.]
 [ 6.  8.  5.  5.  7.  6.  7. 12.  8.  7.  7.  5.  6.  6.  3.  8.  8.
5.
   9.  5.]
 [ 9. 10.  8.  8.  9.  9.  8.  8. 15.  7.  8.  8.  5.  6.  7. 11.  7.
7.
   7.  8.]
 [ 5.  5.  4.  8.  8.  3.  6.  7.  7. 10.  6.  5.  6.  4.  4.  5.  6.
5.
   7.  4.]
 [ 4.  5.  4.  6.  6.  8.  8.  7.  8.  6. 11.  3.  5.  5.  5.  8.  8.
5.
   7.  7.]
 [ 6.  8.  7.  6.  6.  5.  5.  5.  8.  5.  3.  9.  3.  3.  2.  5.  5.
3.
   6.  3.]
 [ 5.  5.  3.  5.  4.  4.  5.  6.  5.  6.  5.  3.  8.  4.  2.  5.  6.
3.
   6.  4.]
 [ 7.  6.  4.  5.  5.  8.  7.  6.  6.  4.  5.  3.  4. 10.  3.  6.  6.
6.
   5.  6.]
 [ 3.  3.  1.  5.  4.  4.  4.  3.  7.  4.  5.  2.  2.  3.  7.  6.  3.
5.
   2.  4.]
 [ 7.  7.  6.  5.  6.  9.  6.  8. 11.  5.  8.  5.  5.  6.  6. 13.  7.
5.
   6.  8.]
 [ 4.  8.  5.  7.  4.  7.  7.  8.  7.  6.  8.  5.  6.  6.  3.  7. 11.
4.
   7.  7.]
 [ 7.  6.  4.  7.  5.  6.  7.  5.  7.  5.  5.  3.  3.  6.  5.  5.  4.
10.
   5.  6.]
 [ 5.  7.  6.  6.  7.  7.  7.  9.  7.  7.  7.  6.  6.  5.  2.  6.  7.
5.
  11.  5.]
 [ 5.  6.  5.  5.  3.  8.  5.  5.  8.  4.  7.  3.  4.  6.  4.  8.  7.
6.
   5. 10.]]
```

In [77]:
```
# hop with k=2
similarnodes(hopeneighbormbd(A_new, H_new, k=2))
```

Out[77]:
```
(14, 15)
```

To sum up, for a set of 20 nodes numbered from 0 to 19, the most similar node pairs measured by cosine similarity are:

- Random Walk: (1, 4)
- 1 - hop: (2, 8)
- 2 - hop: (14, 15)

We can see that they are all different. The possible explanations are as follows:

1. Algorithms: This concerns Random Walk (Test 1) and K-hop (Tests 2/3)

    - Random Walk manifests itself by starting from one node, and traverse a given length *l*, and repeat the process *m* times. This means that at each time, the two parameters will largely dictate the coverage of the algorith. For example, on the one hand if one node's connectivity is low (less edges with other nodes) and the times we walk is larger, we might be repeating the same walk many times, thus magnifying the information; on the other hand, if a node has more edges than the number of walks, coverage will surely not fully reflect the truth of that part of the graph. In addition, we are choosing randomly by coin-tossing at each step, and that means there is no guarantee that we may have repeatable walks, and that we could also repeat the same path multiply times, either partially or fully, given that we get same results when 'tossing'.
    - K-hop make sure that we 'fairly' cover all connected nodes and that each hop builds the 'tree' covering all the children. This means that the connectivity of nodes are more fairly covered, and that there is no reproducibility issue induced by random coin-toss.
2. Depth of coverage: This concerns Tests 2 vs 3

    - Now focus on the same algorithm. Tests 2 and 3 use the same algorithm, but the depths are different. Test 2 only uses one-hop, and the 'map' is dictated by the 1-th power of the adjacency matrix: $A^1 = A$, which is just the adjacency matrix itself. This will provide the a set of all immediate neighbouring nodes.
    - Test 3 has 2 hops, which is reflected by A^2. That is to say, on top of what we get from Test 2 based on first-hop, we hop again from all the 1-hop neighbours and dig one step further at all those nodes. This apparently will return a different set of nodes, and that we could also see the entries of matrix are not only 0s and 1s, which means that we have traveled to some points more than once, which further proves that the 2-hop neighbours are different than 1-hop. This will surely provides a different result in node embedding and hence, cosine similarity.