Assingment_4

December 16, 2019

1 Assignmnet 4

1.0.1 IFT6758 Fall 2019

1.0.2 Due date: December 15, 2019

1.0.3 Submit your answers and code as a pdf file in gradescope.

1.0.4 Deep Learning (Convolutional Neural Networks) [60 points + 10 bonus 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 deep learning library and for Sklearn. Provide your code and the output.

Data preparation (20 points)

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

```
from matplotlib import pyplot as plt
import numpy as np

from sklearn.datasets import fetch_openml
mnist_data = fetch_openml(name='mnist_784')

mnist_data.keys()
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details', 'categories',
'url'])
```

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

2. (3 point) Reshape it back to 28x28 and visualize a couple of images with matplotlib.

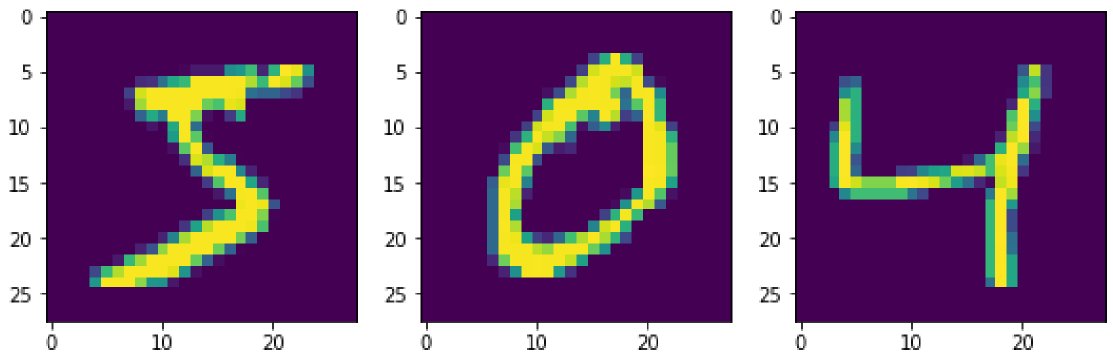
   Finally, we want to add one dummy dimension for the non-existent color information. Every resulting image should have dimensions 28x28x1.

   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.

3. (3 point) Add a channel dimension.

```
# Answer 2:

# reshape data
data = mnist_data.data.reshape(-1,28,28)
target = mnist_data.target.astype(int)
print("Data Shape: %s \n"%str(data.shape))

#visualize images
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(10,3))

for i in range(len(ax)):
    ax[i].imshow(data[i])
```

```
Data Shape: (70000, 28, 28)
```



```
# Answer 3 :
data = data.reshape(-1,28,28,1)
data.shape
```

```
(70000, 28, 28, 1)
```

To simplify the task we exclude some classes.

4. (3 point) Filter data leaving only classes 1, 3, 7. Transform features and targets. How many data points left after filtering?

```
filtered_index = np.where(np.isin(target,[1,3,7]))

data = data[filtered_index]
target = target[filtered_index]

data.shape, target.shape
```

```
We are left with 22311 data points after excluding the other classes.
((22311, 28, 28, 1), (22311,))
```

5. (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=...)
```

```
def to_categorical(array, classes):
    """
    array -- array of targets
    classes -- list of classes
    """
    one_hot_list = []
    map_target = {v:k for k,v in enumerate(classes)}
    for y in array:
        one_hot = [0]*len(classes)
        one_hot[map_target[y]] = 1
        one_hot_list.append(one_hot)
    return np.array(one_hot_list), map_target

mnist_targets, map_target = to_categorical(target, classes=[1,3,7])
mnist_targets[0:5]
```

```
array([[1, 0, 0],
       [1, 0, 0],
       [0, 1, 0],
       [1, 0, 0],
       [0, 1, 0]])
```

6. (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.

```
X_train = data[:16000]
y_train = mnist_targets[:16000]

X_val = data[16000:19000]
y_val = mnist_targets[16000:19000]

X_test = data[19000:]
y_test = target[19000:]
```

Training (35 points) Use Keras (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()
```

```
import tensorflow.keras as k

model = k.Sequential([
        k.layers.Conv2D(filters = 6, kernel_size = (5,5), input_shape=data[0].
 ↪shape),
        k.layers.ReLU(),
        k.layers.MaxPool2D(pool_size=(2,2)),
        k.layers.Conv2D(filters = 16, kernel_size = (5,5)),
        k.layers.ReLU(),
        k.layers.MaxPool2D(pool_size=(2,2)),
        k.layers.Flatten(),
        k.layers.Dense(128),
        k.layers.ReLU(),
        k.layers.Dense(3),
        k.layers.Softmax()
])
model.summary();
```

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 24, 24, 6)         156
_________________________________________________________________
re_lu (ReLU)                 (None, 24, 24, 6)         0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 6)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 8, 8, 16)          2416
_________________________________________________________________
re_lu_1 (ReLU)               (None, 8, 8, 16)          0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 128)               32896
_________________________________________________________________
re_lu_2 (ReLU)               (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 387
_________________________________________________________________
softmax (Softmax)            (None, 3)                 0
=================================================================
Total params: 35,855
Trainable params: 35,855
Non-trainable params: 0
_________________________________________________________________
```

2. (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'],
              )
```

```
import tensorflow.keras as k

optimizer = k.optimizers.SGD(learning_rate=0.0001)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'],
              )
```

3. (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.

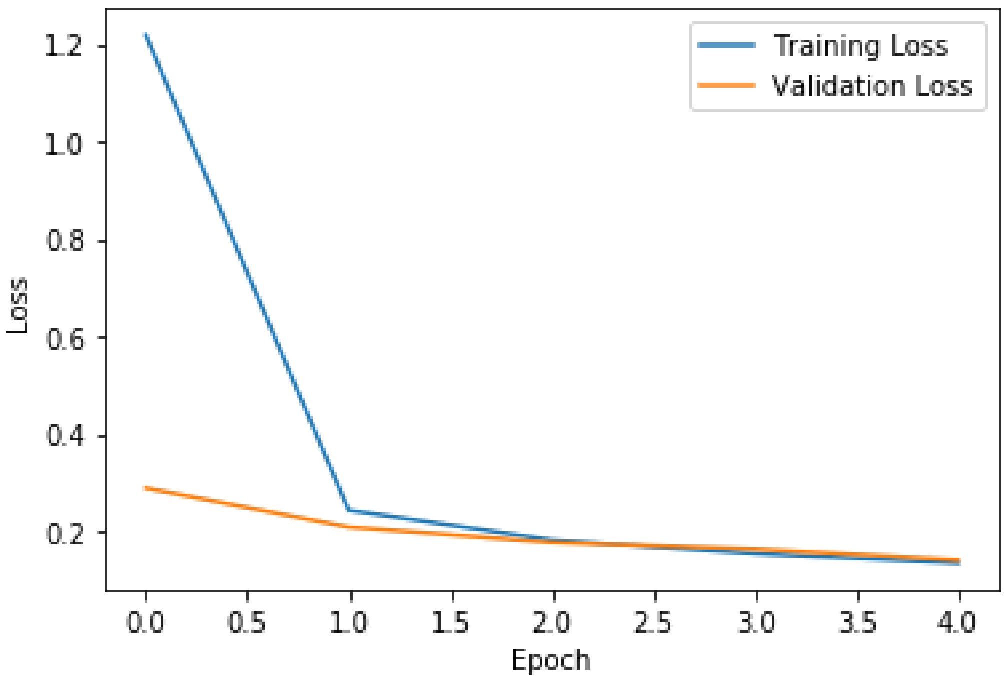
```
history = model.fit(X_train, y_train, validation_data=(X_val, y_val),␣
↪batch_size=64, epochs=5)
```

```
Train on 16000 samples, validate on 3000 samples
Epoch 1/5
16000/16000 [==============================] - 6s 368us/sample - loss: 1.2180 -
accuracy: 0.9068 - val_loss: 0.2893 - val_accuracy: 0.9583
Epoch 2/5
16000/16000 [==============================] - 6s 375us/sample - loss: 0.2437 -
accuracy: 0.9620 - val_loss: 0.2090 - val_accuracy: 0.9700
Epoch 3/5
16000/16000 [==============================] - 6s 374us/sample - loss: 0.1824 -
accuracy: 0.9707 - val_loss: 0.1776 - val_accuracy: 0.9713
Epoch 4/5
16000/16000 [==============================] - 5s 342us/sample - loss: 0.1547 -
accuracy: 0.9742 - val_loss: 0.1639 - val_accuracy: 0.9720
Epoch 5/5
16000/16000 [==============================] - 5s 340us/sample - loss: 0.1358 -
accuracy: 0.9760 - val_loss: 0.1417 - val_accuracy: 0.9763
```

4. (5 points) Plot the training loss against the validation loss. Do you observe overfitting/underfitting?

   HINT Explore the return value in the previous step.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(["Training Loss","Validation Loss"])
plt.show()
```



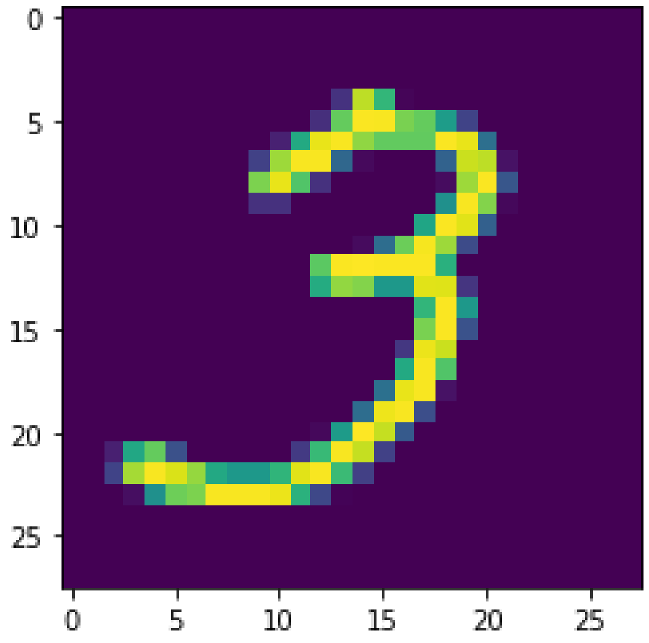
The model does not overfit or underfit as both the losses are decreasing and training loss ≈ validation loss. We have 0.97 accuracy on both training and validation set.

Evaluation (5 points)

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

```
map_target_rev = {v:k for k,v in map_target.items()}

y_pred = model.predict_classes(X_test)
y_pred = np.array([map_target_rev[x] for x in y_pred])

plt.imshow(X_test[0][:,:,0])
print("Prediction", map_target_rev[model.predict_classes(X_test[0:1])[0]])
```

```
Prediction 3
```



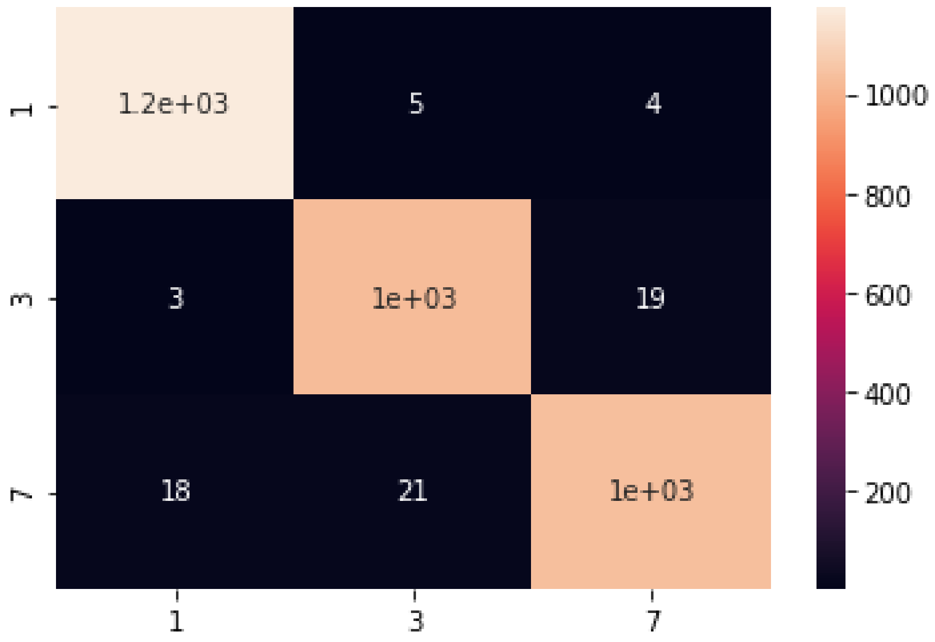
2. (4 points) Compute the confusion matrix and the accuracy. Which classes confused most often?

   The model should have at least 90% accuracy.

```
from sklearn.metrics import accuracy_score
acc = accuracy_score(y_pred, y_test)
print("Accuracy on test set is ", +acc)
```

```
Accuracy on test set is  0.9752340682573241
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, xticklabels=[1,3,7], yticklabels=[1,3,7]);
```



As evident in the confusion matrix, the classes 3 and 7 seems to be getting confused here.

Bonus point (10 points) Can you suggest an improvement to the model? Implement it and compare to the one above. How to do robust comparison of the performance?

Answer:

Although the model seems to be already performing good, but in general we can improve the model's performance by following the below steps.

1. **Adam Optimizer:** Adding a ADAM optimizer instead of a SGD, could significantly improve the performance.
2. **Increase Epoch:** Increasing the epoch improves the performance, by one needs to keep an eye on the model because it may overfit.
3. **Increasing Parameters:** If the training data is increased to include all the classes, we might need to increase the number of parameters
4. **Regularization:** If the model overfits due to the increased capacity, regularization can help to generalize the model.

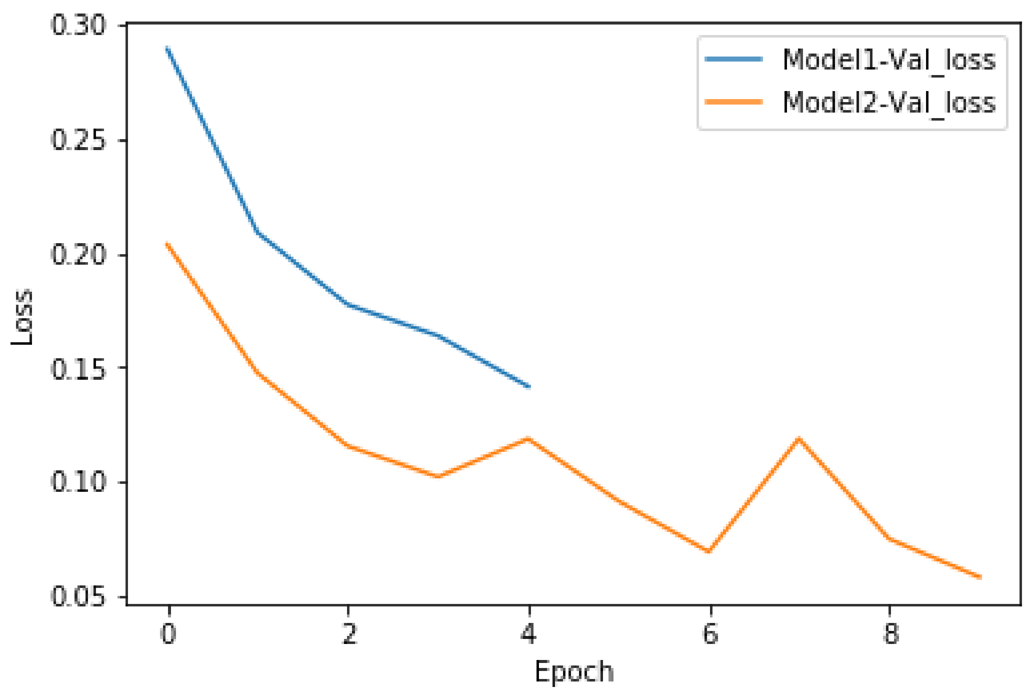
We can do a robust comparison by comparing the validation accuracy/loss on both the models.

```
# Changed optimizer to ADAM
optimizer = k.optimizers.Adam(learning_rate=0.0001)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'],
              )

# Increased epoch from 5 to 10
history2 = model.fit(X_train, y_train, validation_data=(X_val, y_val),␣
 ↪batch_size=64, epochs=10)
```

```
Train on 16000 samples, validate on 3000 samples
Epoch 1/10
16000/16000 [==============================] - 7s 465us/sample - loss: 0.5425 -
accuracy: 0.9466 - val_loss: 0.2038 - val_accuracy: 0.9753
Epoch 2/10
16000/16000 [==============================] - 6s 355us/sample - loss: 0.1644 -
accuracy: 0.9789 - val_loss: 0.1476 - val_accuracy: 0.9803
Epoch 3/10
16000/16000 [==============================] - 6s 345us/sample - loss: 0.1050 -
accuracy: 0.9847 - val_loss: 0.1157 - val_accuracy: 0.9847
Epoch 4/10
16000/16000 [==============================] - 6s 353us/sample - loss: 0.0774 -
accuracy: 0.9886 - val_loss: 0.1022 - val_accuracy: 0.9870
Epoch 5/10
16000/16000 [==============================] - 6s 372us/sample - loss: 0.0589 -
accuracy: 0.9904 - val_loss: 0.1189 - val_accuracy: 0.9840
```

```
Epoch 6/10
16000/16000 [==============================] - 5s 336us/sample - loss: 0.0503 -
accuracy: 0.9906 - val_loss: 0.0917 - val_accuracy: 0.9880
Epoch 7/10
16000/16000 [==============================] - 5s 339us/sample - loss: 0.0373 -
accuracy: 0.9919 - val_loss: 0.0695 - val_accuracy: 0.9910
Epoch 8/10
16000/16000 [==============================] - 6s 346us/sample - loss: 0.0303 -
accuracy: 0.9934 - val_loss: 0.1189 - val_accuracy: 0.9827
Epoch 9/10
16000/16000 [==============================] - 6s 375us/sample - loss: 0.0257 -
accuracy: 0.9941 - val_loss: 0.0751 - val_accuracy: 0.9900
Epoch 10/10
16000/16000 [==============================] - 5s 337us/sample - loss: 0.0161 -
accuracy: 0.9959 - val_loss: 0.0584 - val_accuracy: 0.9907
```

```python
plt.plot(history.history['val_loss'])
plt.plot(history2.history['val_loss'])
plt.legend(["Model1-Val_loss","Model2-Val_loss"])
plt.show()
```



Model 2 seems to perform better on Validation set than Model 1. Adam Optimizer seems to quickly reduce the loss from the beginning, and increasing the number of epoch futher helps to reduce the error.

1.0.5 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 claculation 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.

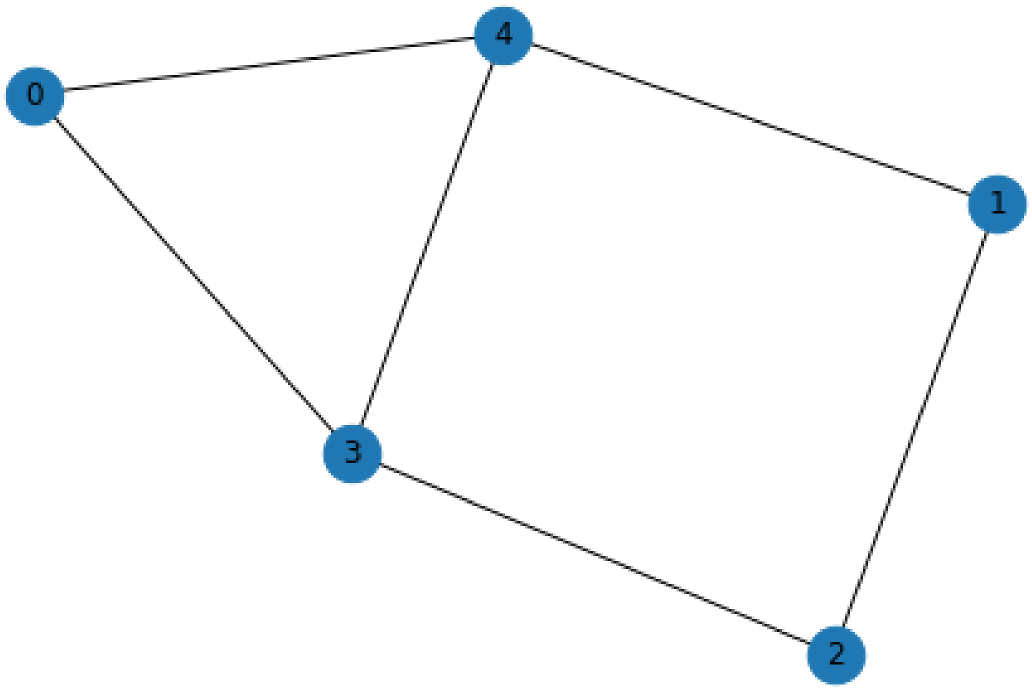
```
import matplotlib.pyplot as plt
import networkx as nx
import random
%matplotlib inline
def show_graph_with_labels(adjacency_matrix):
    rows, cols = np.where(adjacency_matrix == 1)
    edges = zip(rows.tolist(), cols.tolist())
    gr = nx.Graph()
    gr.add_edges_from(edges)
    nx.draw(gr, node_size=500, with_labels=True)
    plt.show();
```

```
def randadjmat(n,p):
    A = np.zeros((n,n))
    for i in range(n):
        for j in range(i,n):
            if random.random() < p and i != j:
                A[i][j] = 1
                A[j][i] = 1

    ## to avoid disconnected graphs:
    if 0 in np.sum(A, axis = 1):
        A = randadjmat(n,p)
    return A

A = randadjmat(5,0.4)
A
```

```
array([[0., 0., 0., 1., 1.],
       [0., 0., 1., 0., 1.],
       [0., 1., 0., 1., 0.],
       [1., 0., 1., 0., 1.],
       [1., 1., 0., 1., 0.]])
```

```
show_graph_with_labels(A);
```

<next page>



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 $1/degree(u)$.

```
def transionmat(A):
    B = np.copy(A)
    n_nodes,_ = B.shape
    for i in range(n_nodes):
        n_edges = B[i].sum()
        prob = 1/n_edges if n_edges > 0 else 0

        for j in range(n_nodes):
            if B[i][j] != 0:
                B[i][j] = prob
    return B

T = transionmat(A)
T
```

```
array([[0.        , 0.        , 0.        , 0.5       , 0.5       ],
       [0.        , 0.        , 0.5       , 0.        , 0.5       ],
       [0.        , 0.5       , 0.        , 0.5       , 0.        ],
       [0.33333333, 0.        , 0.33333333, 0.        , 0.33333333],
       [0.33333333, 0.33333333, 0.        , 0.33333333, 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.

```python
def hotembd(A):
    n_nodes,_ = A.shape
    return np.identity(n_nodes)

H = hotembd(A)
H
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 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 on the graph from each node w times with lenght 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.

```python
def randwalkembd(A, T, H, w, l):
    w_count = 0
    node_emb_matrix = np.zeros(A.shape)
    n_nodes,_ = A.shape
    for i in range(n_nodes):
        w_count = 0
        while(w_count<=w):
            w_count += 1
            l_count = 0
            new_vertex = i
            while(l_count<=l):
                new_vertex = np.random.choice(list(range(0,n_nodes)), p =␣
 ↪T[new_vertex].tolist()) # Get the new vertex
                node_emb_matrix[i] += H[new_vertex] # Sum the one_hot for␣
 ↪new_vertex found in the walk
                l_count += 1
    return node_emb_matrix

W = randwalkembd(A, T, H, 5, 3)
W
```

```
array([[4., 3., 3., 8., 6.],
       [3., 3., 4., 7., 7.],
       [5., 3., 0., 9., 7.],
       [1., 7., 5., 5., 6.],
       [5., 6., 4., 2., 7.]])
```

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.

```
def hopeneighbormbd(A, H, k):
    A_k = np.linalg.matrix_power(A, k)
    hop_emb_matrix = np.zeros(A.shape)
    n_nodes,_ = A_k.shape
    for row in range(n_nodes):
        for col in range(n_nodes):
            if A_k[row][col] != 0:
                hop_emb_matrix[row] += H[col] # taking only once,

    return hop_emb_matrix, A_k
```

The absolute value in power matrix is not used for summing the 1-hot encodings of neighbours, instead it is added only once per neighbour. Otherwise, it will just become the power matrix.

```
h, a_k = hopeneighbormbd(A, H, 2)
h # Hop Embedding
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 0., 1., 0.],
       [1., 0., 1., 0., 1.],
       [1., 1., 0., 1., 1.],
       [1., 0., 1., 1., 1.]])
```

```
a_k # Power Matrix
```

```
array([[2., 1., 1., 1., 1.],
       [1., 2., 0., 2., 0.],
       [1., 0., 2., 0., 2.],
       [1., 2., 0., 3., 1.],
       [1., 0., 2., 1., 3.]])
```

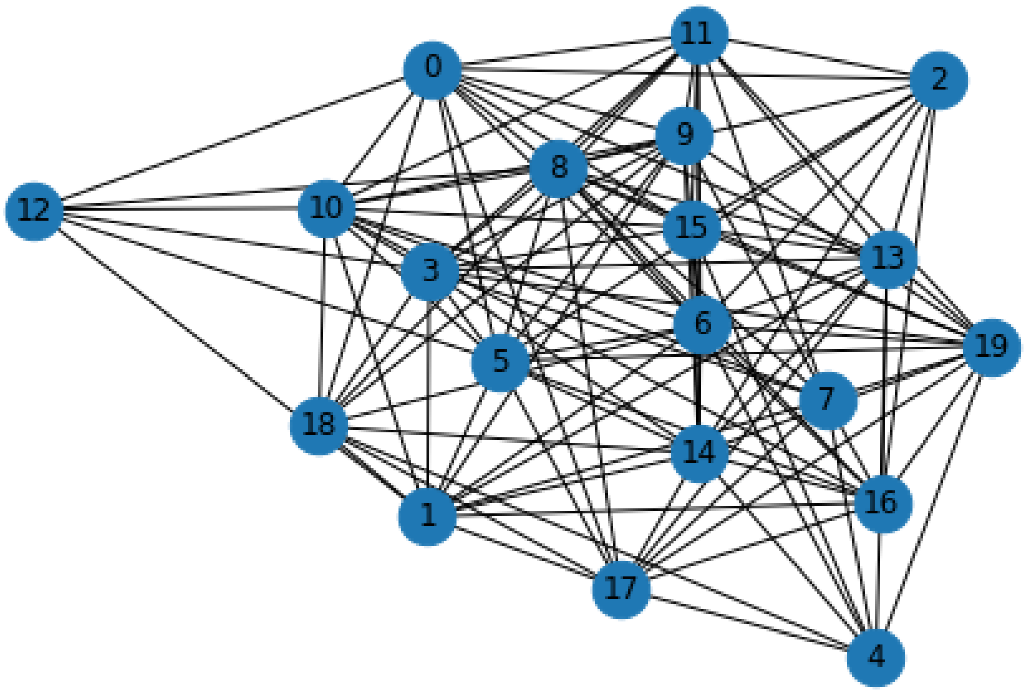
6- (5 points) Write a function similarnodes(Z) which, given an node embedding matrix, find the most similar nodes in the graph.

```
def similarnodes(Z):
    n_nodes,_ = Z.shape
    max_sim = 0; max_i = 0 ; max_j = 0
    for i in range(n_nodes):
        for j in range(i+1, n_nodes):
            dot_product = sum(Z[i]*Z[j])
            if dot_product > max_sim:
                max_sim = dot_product
                max_i = i
                max_j = j
    return max_i, max_j
```

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?

```
A = randadjmat(20,0.6)
T = transionmat(A)
H = hotembd(T)
emb_randwk = randwalkembd(A, T, H, w = 10, l = 4)
emb_hop_1,_ = hopeneighbormbd(A, H, k = 1)
emb_hop_2,_ = hopeneighbormbd(A, H, k = 2)
i_1, j_1 = similarnodes(emb_randwk)
i_2, j_2 = similarnodes(emb_hop_1)
i_3, j_3 = similarnodes(emb_hop_2)

print("Similar nodes from RandomWalk ->" , i_1, j_1)
print("Similar nodes with 1 hop ->", i_2, j_2)
print("Similar nodes from 2 hop ->", i_3, j_3)

show_graph_with_labels(A)
```

```
Similar nodes from RandomWalk -> 2 17
Similar nodes with 1 hop -> 3 6
Similar nodes from 2 hop -> 0 1
```



<next page>

Similar nodes are different as the embeddings are generated with different methods. This difference between different graph embeddings lies in how they define the graph property to be preserved. Different algorithms have different insights of the node/edges/neighbours similarities and how to preserve them in the embedded space.

Random Walk Emb: Embeddings of nodes found in the random walks are used to calculate the node embedddings. The results will vary with different w and l paramters.
Hop 1 Embedding: This take into account only the direct relations between the nodes.
Hop 2 Embedding: This also considers the nodes with a 2 hop distance while calculating the embedding.

Since, node similarity are based on different embedding methods, we can expect the most similar nodes to be different.