hidden_layers_embeding

April 17, 2023

1 Naural network hidden layers activation embeding

In this notebook we will use fully connected neural network for a classification task to improve visualizations algorithm from previous classes. In the fully connected neural network, the output of each layer is computed using the activations from the previous one. In neural network training process, each successive layer learns to extract features from data with increasingly higher levels of abstraction. In this exercise, instead of directly visualizing data, we'll try to visualize the activation of hidden layers in neural networks. Using this idea, we can improve the process of data visualization, and on the other hand, see how processing this data looks like by a neural network.

In the first stage, we define simple architecture of the neural network and train it to recognize digits in the MNIST dataset

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.animation
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import SGD
from tensorflow.keras import backend as K

from tensorflow.keras.datasets import mnist
from keras.utils import np_utils
from sklearn.model_selection import train_test_split


%matplotlib inline
plt.rcParams["animation.html"] = "jshtml"
```

[ ]:
```
nb_classes = 10
```

The dropout layers have the very specific function to drop out a random set of activations in that layers by setting them to zero in the forward pass. Simple as that. It allows to avoid overfitting but has to be used only at training time and not at test time.

[ ]:
```
# set dropout rate - fractions of neurons to drop

```

```
dropout = 0.5
```

[ ]:
```
# build very simple neural network with 2 hidden layers

model = Sequential()
model.add(Dense(256, activation='relu', input_shape=(784,)))
model.add(Dropout(dropout))
model.add(Dense(64, activation='relu'))
model.add(Dropout(dropout))
model.add(Dense(nb_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
```

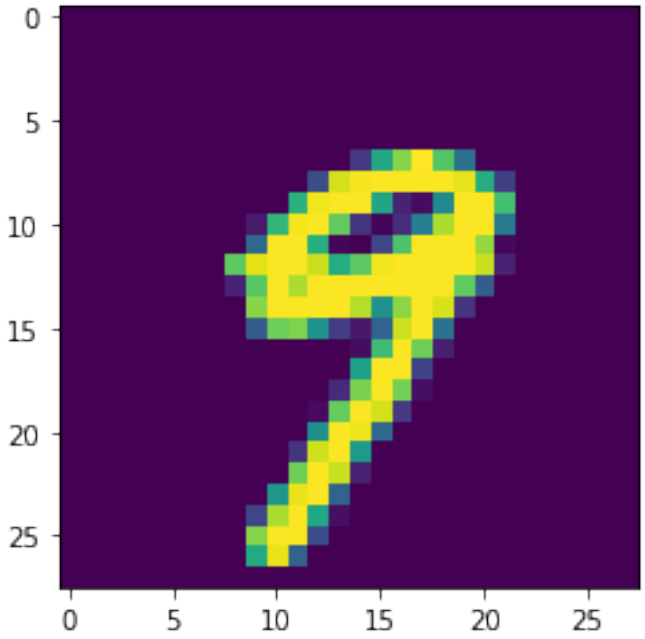
[ ]:
```
# The binary_crossentropy loss expects a one-hot-vector as input,
# so we apply the to_categorical function from keras.utilis to convert integer␣
 ↪labels to one-hot-vectors.
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

[ ]:
```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype("float32")
X_test = X_test.astype("float32")

# Put everything on grayscale
X_train /= 255
X_test /= 255

# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

[ ]:
```
# split training and validation data
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train,␣
 ↪train_size=5/6)
```

[ ]:
```
# show example digit
plt.imshow(X_train[0].reshape(28, 28))
```

[ ]:
```
<matplotlib.image.AxesImage at 0x24645c6b8c8>
```



[ ]:
```
# When we have defined and compiled the model, it can be trained using the fit␣
 ↪function.
# We also use validation dataset to monitor validation loss and accuracy.

network_history = model.fit(X_train, Y_train, batch_size=128,
                            epochs=20, verbose=1, validation_data=(X_val,␣
 ↪Y_val))
```

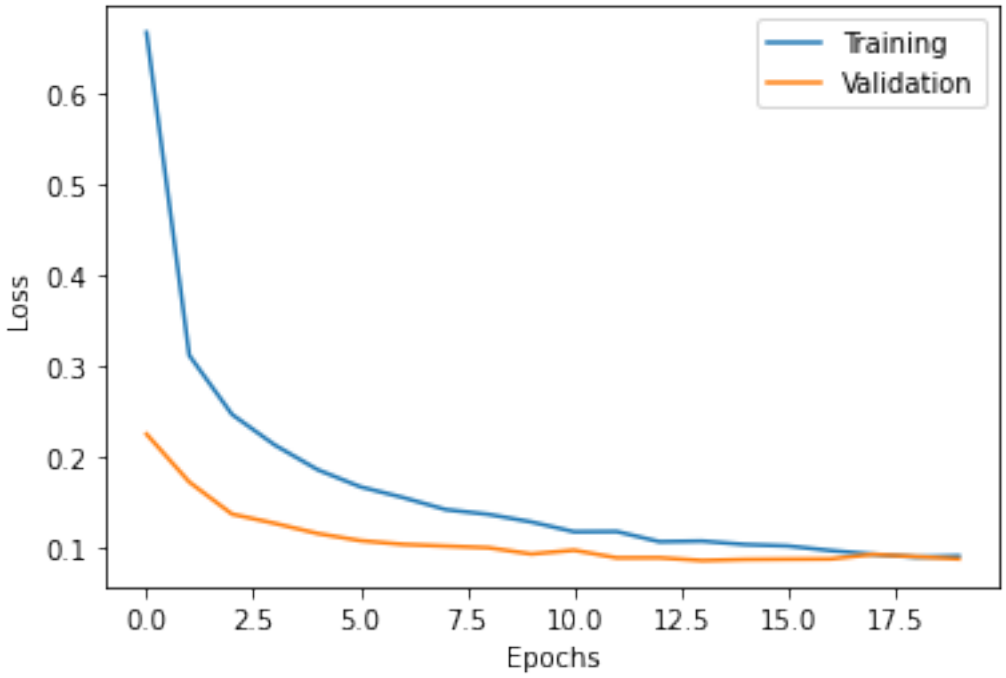
```
Epoch 1/20
391/391 [==============================] - 5s 10ms/step - loss: 0.6666 -
accuracy: 0.7924 - val_loss: 0.2249 - val_accuracy: 0.9369
Epoch 2/20
391/391 [==============================] - 3s 8ms/step - loss: 0.3114 -
accuracy: 0.9127 - val_loss: 0.1722 - val_accuracy: 0.9513
Epoch 3/20
391/391 [==============================] - 2s 6ms/step - loss: 0.2465 -
accuracy: 0.9326 - val_loss: 0.1369 - val_accuracy: 0.9630
Epoch 4/20
391/391 [==============================] - 3s 7ms/step - loss: 0.2129 -
accuracy: 0.9413 - val_loss: 0.1267 - val_accuracy: 0.9650
Epoch 5/20
391/391 [==============================] - 4s 9ms/step - loss: 0.1858 -
accuracy: 0.9467 - val_loss: 0.1156 - val_accuracy: 0.9682
Epoch 6/20
391/391 [==============================] - 4s 10ms/step - loss: 0.1669 -
accuracy: 0.9534 - val_loss: 0.1078 - val_accuracy: 0.9690
```

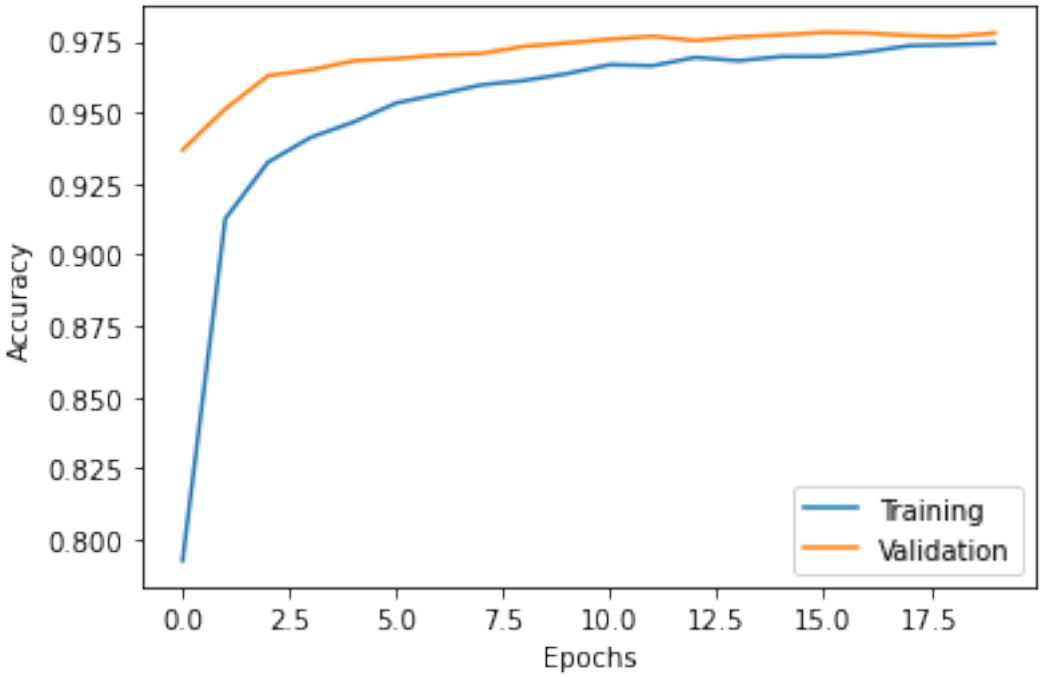
```
Epoch 7/20
391/391 [==============================] - 3s 8ms/step - loss: 0.1552 -
accuracy: 0.9565 - val_loss: 0.1036 - val_accuracy: 0.9702
Epoch 8/20
391/391 [==============================] - 3s 7ms/step - loss: 0.1419 -
accuracy: 0.9598 - val_loss: 0.1018 - val_accuracy: 0.9708
Epoch 9/20
391/391 [==============================] - 3s 7ms/step - loss: 0.1366 -
accuracy: 0.9613 - val_loss: 0.1000 - val_accuracy: 0.9733
Epoch 10/20
391/391 [==============================] - 3s 7ms/step - loss: 0.1283 -
accuracy: 0.9637 - val_loss: 0.0931 - val_accuracy: 0.9745
Epoch 11/20
391/391 [==============================] - 3s 8ms/step - loss: 0.1175 -
accuracy: 0.9668 - val_loss: 0.0972 - val_accuracy: 0.9758
Epoch 12/20
391/391 [==============================] - 3s 7ms/step - loss: 0.1177 -
accuracy: 0.9665 - val_loss: 0.0889 - val_accuracy: 0.9768
Epoch 13/20
391/391 [==============================] - 4s 11ms/step - loss: 0.1066 -
accuracy: 0.9694 - val_loss: 0.0890 - val_accuracy: 0.9753
Epoch 14/20
391/391 [==============================] - 3s 7ms/step - loss: 0.1072 -
accuracy: 0.9682 - val_loss: 0.0859 - val_accuracy: 0.9766
Epoch 15/20
391/391 [==============================] - 3s 8ms/step - loss: 0.1035 -
accuracy: 0.9697 - val_loss: 0.0870 - val_accuracy: 0.9773
Epoch 16/20
391/391 [==============================] - 3s 8ms/step - loss: 0.1019 -
accuracy: 0.9698 - val_loss: 0.0874 - val_accuracy: 0.9782
Epoch 17/20
391/391 [==============================] - 3s 8ms/step - loss: 0.0969 -
accuracy: 0.9713 - val_loss: 0.0876 - val_accuracy: 0.9780
Epoch 18/20
391/391 [==============================] - 4s 10ms/step - loss: 0.0928 -
accuracy: 0.9735 - val_loss: 0.0926 - val_accuracy: 0.9771
Epoch 19/20
391/391 [==============================] - 3s 8ms/step - loss: 0.0897 -
accuracy: 0.9739 - val_loss: 0.0902 - val_accuracy: 0.9767
Epoch 20/20
391/391 [==============================] - 3s 9ms/step - loss: 0.0909 -
accuracy: 0.9744 - val_loss: 0.0877 - val_accuracy: 0.9780
```

[ ]:
```
def plot_history(network_history):
    plt.figure()
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
```

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

    plt.figure()
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.plot(network_history.history['accuracy'])
    plt.plot(network_history.history['val_accuracy'])
    plt.legend(['Training', 'Validation'], loc='lower right')
    plt.show()
```

[ ]:
```
# fit function return keras.callbacks.History object which contains the entire␣
 ↪history
# of training/validation loss, accuracy and other metrices for each epoch.
# We can therefore plot the behaviour of loss and accuracy during the training␣
 ↪phase.

plot_history(network_history)
```





[ ]:
```
# Keras Model have summary function, that print data about model architecture
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               200960

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 64)                16448

 dropout_1 (Dropout)         (None, 64)                0

 dense_2 (Dense)             (None, 10)                650

=================================================================
Total params: 218,058
Trainable params: 218,058
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
# We are interested in downloading the activation of hidden layers, because the␣
↪dropout layers are between them,
```

```
# we need to properly select the index of the three dense layers.

get_layer_output = K.function([model.layers[0].input],
                              [model.layers[0].output, model.layers[2].output,␣
 ↪model.layers[4].output])

layer1_output, layer2_output, layer3_output = get_layer_output([X_train])
```

[ ]:
```
train_ids = [np.arange(len(Y_train))[Y_train[:,i] == 1] for i in range(10)]
```

The 2 graphs below are not directly related to the topic of the exercise, but they visualize very well how neuron activation actives work and for explanation are included.

[ ]:
```
%%capture
%matplotlib inline

# this animation shows what the example number 5 looks like
# and what activations of neurons look in hidden layers of the neural network



# digit to be plotted
digit = 5

# indices of frames to be plotted for this digit
n = range(50)

# initialize plots
f, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(15,4))

# prepare plots
ax1.set_title('Input Layer', fontsize=16)
ax1.axes.get_xaxis().set_visible(False)
ax1.axes.get_yaxis().set_visible(False)

ax2.set_title('Hidden Layer 1', fontsize=16)
ax2.axes.get_xaxis().set_visible(False)
ax2.axes.get_yaxis().set_visible(False)

ax3.set_title('Hidden Layer 2', fontsize=16)
ax3.axes.get_xaxis().set_visible(False)
ax3.axes.get_yaxis().set_visible(False)

ax4.set_title('Output Layer', fontsize=16)
ax4.axes.get_xaxis().set_visible(False)
ax4.axes.get_yaxis().set_visible(False)
```

```
# add numbers to the output layer plot to indicate label
for i in range(3):
    for j in range(4):
        text = ax4.text(j, i, [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, ''␣
 ↪'']][i][j],
                        ha="center", va="center", color="w", fontsize=16)

def animate(id):
    # plot elements that are changed in the animation
    digit_plot = ax1.imshow(X_train[train_ids[digit][id]].reshape((28,28)),␣
 ↪animated=True)
    layer1_plot = ax2.imshow(layer1_output[train_ids[digit][id]].
 ↪reshape((16,16)), animated=True)
    layer2_plot = ax3.imshow(layer2_output[train_ids[digit][id]].
 ↪reshape((8,8)), animated=True)
    output_plot = ax4.imshow(np.append(layer3_output[train_ids[digit][id]],
                                       [np.nan, np.nan]).reshape((3,4)),␣
 ↪animated=True)
    return digit_plot, layer1_plot, layer2_plot, output_plot,

# define animation
ani = matplotlib.animation.FuncAnimation(f, animate, frames=n, interval=100)
```

[ ]:
```
ani
```

[ ]:
```
<matplotlib.animation.FuncAnimation at 0x245de2b5fc8>
```

In most cases the same subset of neurons fires, while other neurons remain quiescent. This is much more obvious in the second hidden layer than in the first hidden layer and can be interpreted as the first layer pre-processesing the pixel data, while the second layer deals with pattern recognition.

This effect is mainly caused by regularization forced by dropout. Dropout generally leads to the sparse weight matrices where a significant part of connection weights are close to 0. Insignificant weights are suppressed.

Optional, nonobligatory task: You can easily see how the visualizations change if you comment lines responsible for the dropout "model.add(Dropout(dropout))". Remember to change "get_layer_output", because after removing the dropout, the dense layers will have indexes: 0,1,2.

[ ]:
```
%%capture
%matplotlib inline


# Let's check the similarity in behavior for frames showing the same digit by␣
 ↪looking at the ensemble properties.
# In this case, ensemble properties refers to how the neurons behave on average
# for a large number of frames showing the same digit.
```

```
# digit to be plotted
digit = 6

# numbers of frames to be summed over
n = np.append([1], np.linspace(5, 100, 20, dtype=int))

# initialize plots
f, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(15,4))

# add a counter indicating the number of frames used in the summation
counter = ax1.text(1, 2, 'n={}'.format(0), color='white', fontsize=16,␣
 ↪animated=True)

# prepare plots
ax1.set_title('Input Layer', fontsize=16)
ax1.axes.get_xaxis().set_visible(False)
ax1.axes.get_yaxis().set_visible(False)

ax2.set_title('Hidden Layer 1', fontsize=16)
ax2.axes.get_xaxis().set_visible(False)
ax2.axes.get_yaxis().set_visible(False)

ax3.set_title('Hidden Layer 2', fontsize=16)
ax3.axes.get_xaxis().set_visible(False)
ax3.axes.get_yaxis().set_visible(False)

ax4.set_title('Output Layer', fontsize=16)
ax4.axes.get_xaxis().set_visible(False)
ax4.axes.get_yaxis().set_visible(False)

# add numbers to the output layer plot to indicate label
for i in range(3):
    for j in range(4):
        text = ax4.text(j, i, [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, '',␣
 ↪'']][i][j],
                        ha="center", va="center", color="w", fontsize=16)

def animate(id):
    # plot elements that are changed in the animation
    digit_plot = ax1.imshow(np.sum(X_train[train_ids[digit][:id]], axis=0).
 ↪reshape((28,28)), animated=True)
    layer1_plot = ax2.imshow(np.sum(layer1_output[train_ids[digit][:id]],␣
 ↪axis=0).reshape((16,16)), animated=True)
    layer2_plot = ax3.imshow(np.sum(layer2_output[train_ids[digit][:id]],␣
 ↪axis=0).reshape((8,8)), animated=True)
    output_plot = ax4.imshow(np.append(np.sum(layer3_output[train_ids[digit][:
 ↪id]], axis=0),
```

```
                                        [np.nan, np.nan]).reshape((3,4)),␣
 ↪animated=True)
    counter.set_text('n={}'.format(id))
    return digit_plot, layer1_plot, layer2_plot, output_plot, counter,

# define animation
ani = matplotlib.animation.FuncAnimation(f, animate, frames=n, interval=100)
```

[ ]:
```
ani
```

[ ]:
```
<matplotlib.animation.FuncAnimation at 0x2455985d288>
```

After summing up the responses of as little as 20-30 frames, the pattern in the second hidden layer is almost static. After combining about 70-80 frames, also the pattern in the first hidden layer appears static. This supports the idea that only a subset of all neurons is involved in the recognition of individual digits.

Especially the above plot is important when we think about use of neural networks for data visualization. We can clearly see that the activation generated by examples belonging to the same class are less chaotic than the examples themselves, therefore their visualization should give a more clustered structure

2 Homework

- project a mnist training part into 2-dimensional space using t-SNE and UMAP.
- use layer1_output and layer2_output to project first and second hidden layers of neural network into a 2-dimensional space. Also divided into a test and training set, use the same methods as the point above.
- also visualize the test part.
- Try to use 2-dimensional projection for classification task.
- Use embeddings lerned on raw train data (and also on hidden activations of train data) to transform test data (and also hidden activations of test data) into 2-dimensional space.
- Use the k-nearest neighbors algorithm to classify transformed points from the test set. Use the KNN algorithm in which you will use points from the training set as a neighbor with known class assignment. Becouse t-SNE is a non-linear, non-parametric embedding you cant use already learned t-SNE to transform new points into the existing embedded space. So for this part, use only UMAP with have fit_transform method (learn manifold) and also transform (only project new data to existing manifold). Try with few values of n_neighbors e.g [3, 5, 10]
- Estimate the accuracy of classification using this approach. Use all 3 layers (raw data, 1 hidden layer, 2 hidden layer) and few values of n_neighbors

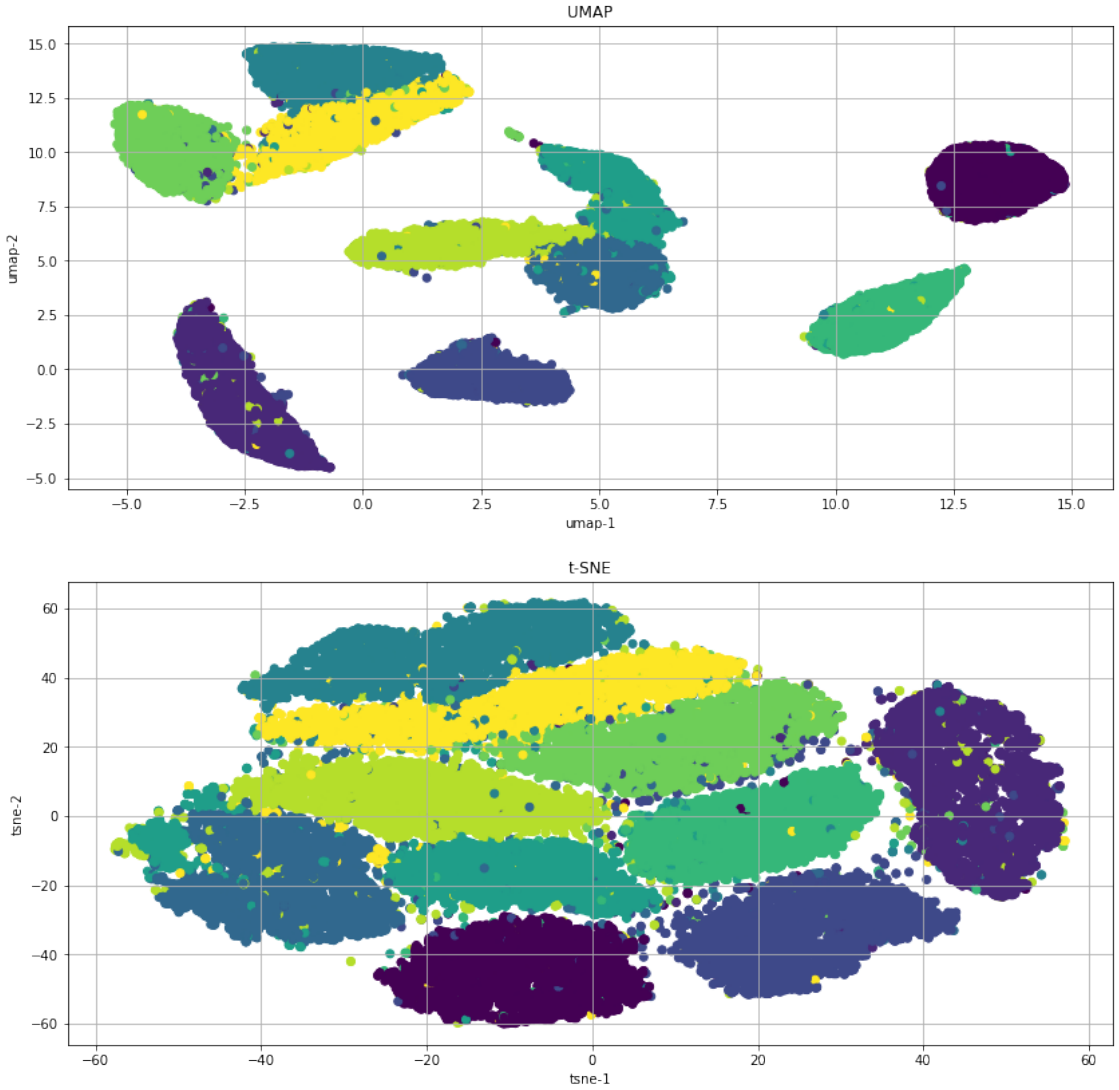
[ ]:
```
# to get hidden activations of test data use:
test_layer1_output, test_layer2_output, test_layer3_output =␣
 ↪get_layer_output([X_test])
```

```
# maybe you would like to inverse of "to_categorical" function. use np.
 ↪argmax(to_categorical(x, k), axis=1) or K.argmax.
```

[ ]:
```
# raw data embedings visualizations
from umap import UMAP
from sklearn.manifold import TSNE

umap = UMAP(n_components=2)
tsne = TSNE(n_components=2)

X_train_umap = umap.fit_transform(X_train)
X_train_tsne = tsne.fit_transform(X_train)
```

[ ]:
```
plt.figure(figsize=(14,14))
plt.subplot(2,1,1)
plt.scatter(X_train_umap[:, 0], X_train_umap[:, 1], c=K.argmax(Y_train))
plt.grid(True)
plt.title("UMAP")
plt.xlabel("umap-1")
plt.ylabel("umap-2")
plt.subplot(2,1,2)
plt.scatter(X_train_tsne[:, 0], X_train_tsne[:, 1], c=K.argmax(Y_train))
plt.grid(True)
plt.title("t-SNE")
plt.xlabel("tsne-1")
plt.ylabel("tsne-2")
```

[ ]:
```
Text(0, 0.5, 'tsne-2')
```

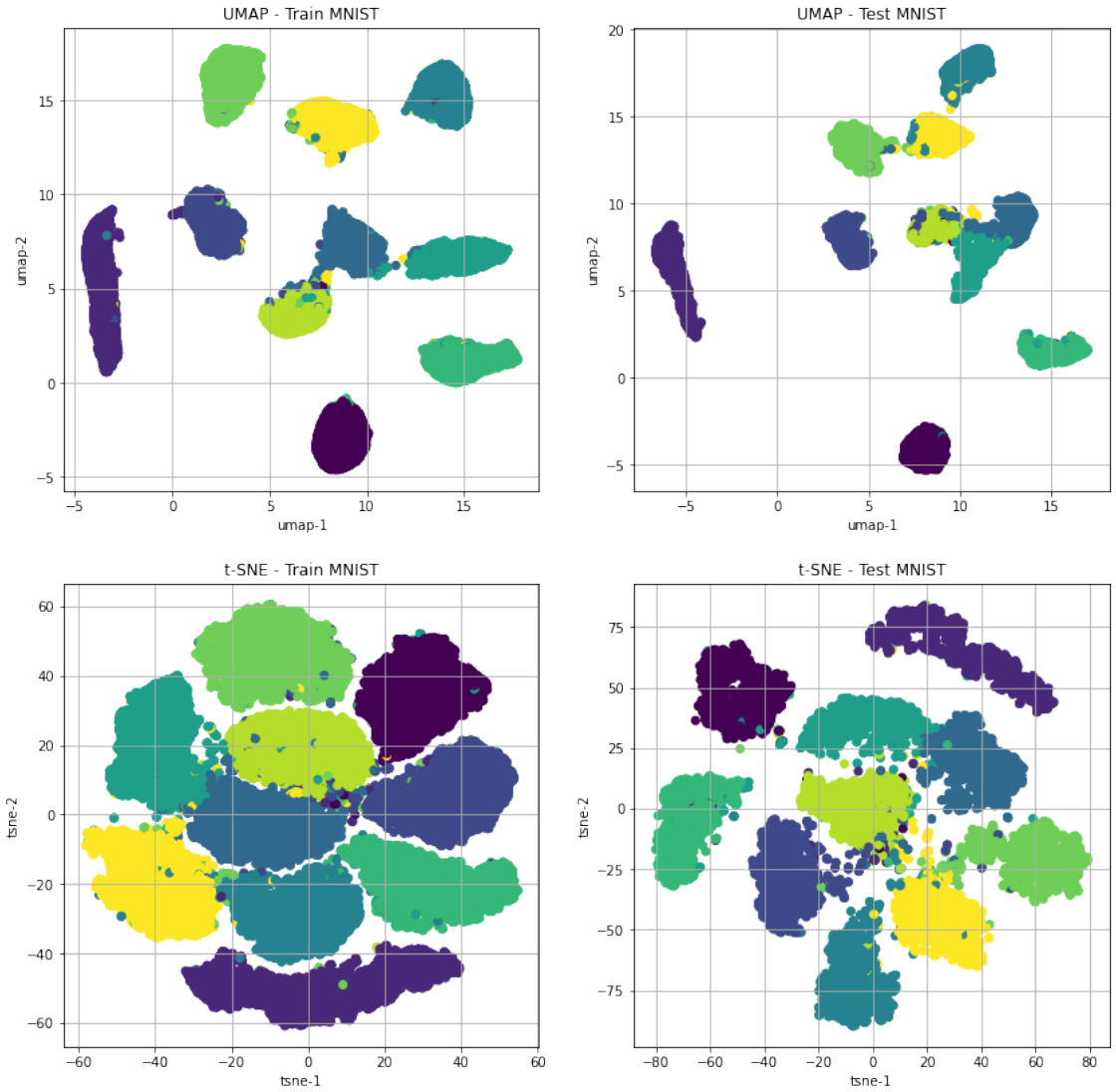


[ ]:
```
# hidden layer 1 embedings visualizations
umap = UMAP(n_components=2)
tsne = TSNE(n_components=2)

train_layer1_umap = umap.fit_transform(layer1_output)
train_layer1_tsne = tsne.fit_transform(layer1_output)

test_layer1_umap = umap.fit_transform(test_layer1_output)
test_layer1_tsne = tsne.fit_transform(test_layer1_output)
```

[ ]:
```
plt.figure(figsize=(14,14))
plt.subplot(2,2,1)
```

```
plt.scatter(train_layer1_umap[:, 0], train_layer1_umap[:, 1], c=K.
 ↪argmax(Y_train))
plt.grid(True)
plt.title("UMAP - Train MNIST")
plt.xlabel("umap-1")
plt.ylabel("umap-2")
plt.subplot(2,2,2)
plt.scatter(test_layer1_umap[:, 0], test_layer1_umap[:, 1], c=K.argmax(Y_test))
plt.grid(True)
plt.title("UMAP - Test MNIST")
plt.xlabel("umap-1")
plt.ylabel("umap-2")
plt.subplot(2,2,3)
plt.scatter(train_layer1_tsne[:, 0], train_layer1_tsne[:, 1], c=K.
 ↪argmax(Y_train))
plt.grid(True)
plt.title("t-SNE - Train MNIST")
plt.xlabel("tsne-1")
plt.ylabel("tsne-2")
plt.subplot(2,2,4)
plt.scatter(test_layer1_tsne[:, 0], test_layer1_tsne[:, 1], c=K.argmax(Y_test))
plt.grid(True)
plt.title("t-SNE - Test MNIST")
plt.xlabel("tsne-1")
plt.ylabel("tsne-2")
```

[ ]:
```
Text(0, 0.5, 'tsne-2')
```

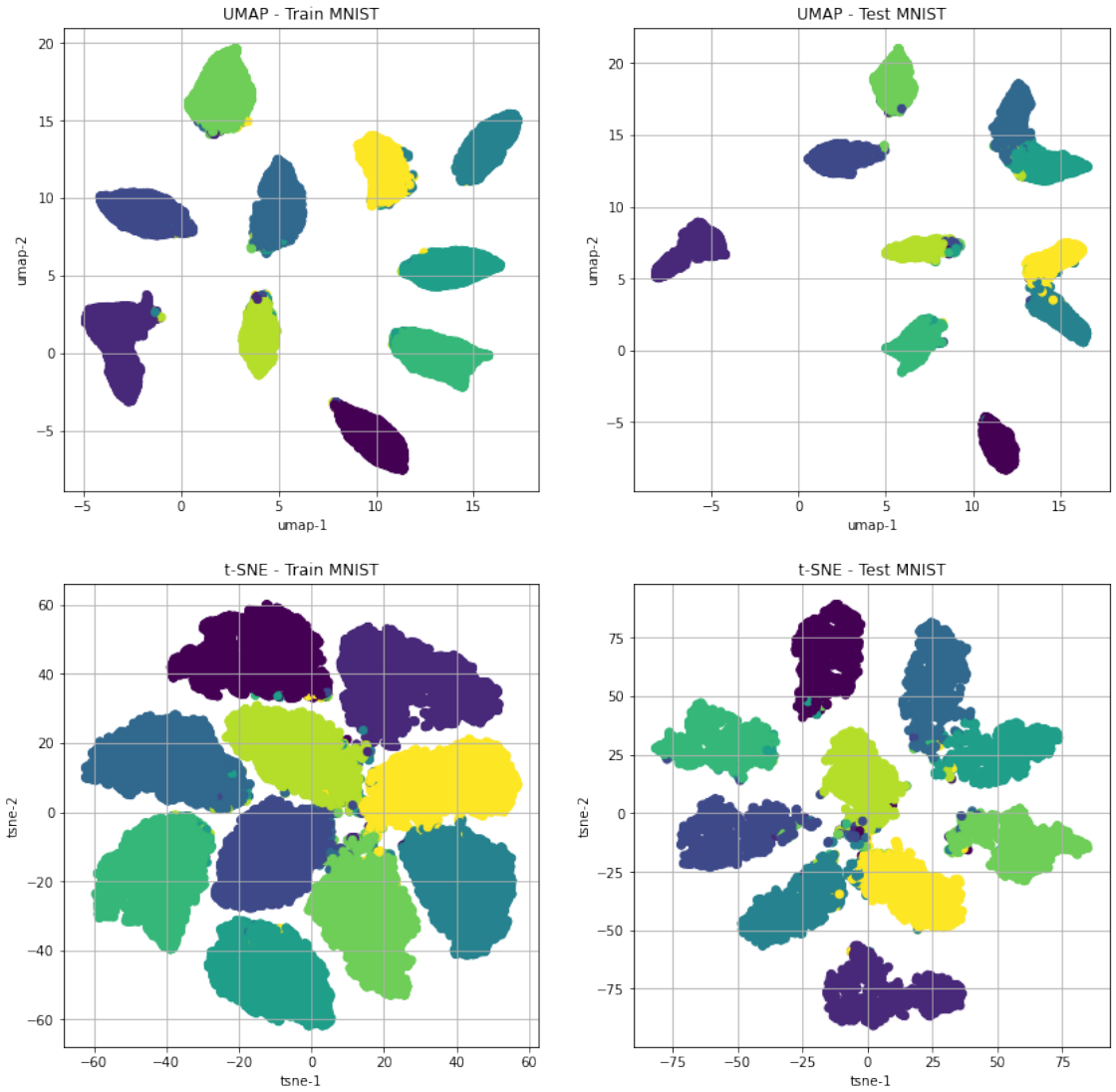


[ ]:
```
# hidden layer 2 embedings visualizations
umap = UMAP(n_components=2)
tsne = TSNE(n_components=2)

train_layer2_umap = umap.fit_transform(layer2_output)
train_layer2_tsne = tsne.fit_transform(layer2_output)

test_layer2_umap = umap.fit_transform(test_layer2_output)
test_layer2_tsne = tsne.fit_transform(test_layer2_output)
```

[ ]:
```
plt.figure(figsize=(14,14))
plt.subplot(2,2,1)
```

```
plt.scatter(train_layer2_umap[:, 0], train_layer2_umap[:, 1], c=K.
 ↪argmax(Y_train))
plt.grid(True)
plt.title("UMAP - Train MNIST")
plt.xlabel("umap-1")
plt.ylabel("umap-2")
plt.subplot(2,2,2)
plt.scatter(test_layer2_umap[:, 0], test_layer2_umap[:, 1], c=K.argmax(Y_test))
plt.grid(True)
plt.title("UMAP - Test MNIST")
plt.xlabel("umap-1")
plt.ylabel("umap-2")
plt.subplot(2,2,3)
plt.scatter(train_layer2_tsne[:, 0], train_layer2_tsne[:, 1], c=K.
 ↪argmax(Y_train))
plt.grid(True)
plt.title("t-SNE - Train MNIST")
plt.xlabel("tsne-1")
plt.ylabel("tsne-2")
plt.subplot(2,2,4)
plt.scatter(test_layer2_tsne[:, 0], test_layer2_tsne[:, 1], c=K.argmax(Y_test))
plt.grid(True)
plt.title("t-SNE - Test MNIST")
plt.xlabel("tsne-1")
plt.ylabel("tsne-2")
```

[ ]:
```
Text(0, 0.5, 'tsne-2')
```



Low dimensional projections of the 1st and 2nd hidden layers provides a better division of the MNIST digits in the latent space, in comparison to the embedding methods used on the original data. This is the outcome strongly associated with the results of the animation in the previous excercise, as the layers learn the patterns from the data and the activations for the same digits are less chaotic than the digits themselves. Thus UMAP and t-SNE projection of the outputs of the 1st layer already lay on separate manifolds, the ones from the 2nd layer are perfectly separated into dense cluster structure.

[ ]:
```
# calculating accuracies
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

NUM_NEIGHBORS = [3, 5, 10]
```

```
DATA_FORM = {"raw data": (X_train, X_test), "1st hidden layer data":␣
 ↪(layer1_output, test_layer1_output), "2nd hidden layer data":␣
 ↪(layer2_output, test_layer2_output)}

for nn in NUM_NEIGHBORS:
    for data in DATA_FORM.items():
        umap = UMAP(n_components=2)
        X_train_embedded = umap.fit_transform(data[1][0])
        X_test_embedded = umap.transform(data[1][1])

        train_labels = K.argmax(Y_train)
        test_labels = K.argmax(Y_test)

        knn = KNeighborsClassifier(n_neighbors=nn)
        knn.fit(X_train_embedded, train_labels)

        predictions = knn.predict(X_test_embedded)

        acc = accuracy_score(test_labels, predictions)

        print(f"Accuracy of KNN classifier with {nn} nearest neighbors on␣
 ↪{data[0]} = {acc}\n")
    print("-----------------------------------------")
```

```
Accuracy of KNN classifier with 3 nearest neighbors on raw data = 0.953

Accuracy of KNN classifier with 3 nearest neighbors on 1st hidden layer data =
0.9682

Accuracy of KNN classifier with 3 nearest neighbors on 2nd hidden layer data =
0.9776

-----------------------------------------
Accuracy of KNN classifier with 5 nearest neighbors on raw data = 0.955

Accuracy of KNN classifier with 5 nearest neighbors on 1st hidden layer data =
0.9691

Accuracy of KNN classifier with 5 nearest neighbors on 2nd hidden layer data =
0.9782

-----------------------------------------
Accuracy of KNN classifier with 10 nearest neighbors on raw data = 0.9537

Accuracy of KNN classifier with 10 nearest neighbors on 1st hidden layer data =
0.9702
```

```
Accuracy of KNN classifier with 10 nearest neighbors on 2nd hidden layer data =
0.9789

----------------------------------------
```

With the use of the embedded space of the hidden layers, the performance of the KNN Classifier on the test dataset is better in terms of accuracy.