Section 16 Neural Network (and Deep Learning)

Basic Structures of Neural Network (NN)

Intuitively, Neural Network is nothing but the construction of one nonlinear (vector-valued) function $\mathbf{h}(\mathbf{x}; W, b) \in \mathbb{R}^p \to \mathbb{R}^k$ with a series of composite functions (layers). For the interactive visualization and exploration of NN, you can refer here.

The layers can be further classified into three categories:

- **Input layer**: p nodes representing the input sample $x \in \mathbb{R}^p$

- **Hidden layers**: Calculate the values on the nodes of $(l+1)$-th layers based on those on the l-th layer as

  $$\begin{aligned} \mathbf{z}^{(l+1)} &= W^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)} \\ \mathbf{a}^{(l+1)} &= \sigma(\mathbf{z}^{(l+1)}) \end{aligned}$$

  where σ is the nonlinear activation function (common choices include sigmoid, tanh or ReLU), and the second equation is element-wise.

  Note: Suppose there are m nodes in layer l and n nodes in layer $(l+1)$. Then there are $(m+1) \times n$ trainable parameters between the two layers, or sometimes by convention, we just say they are the parameters are on the $(l+1)$-th layer.

- **Output layer**: k nodes representing the output, and dependent on the problem can be:

  1) For regression problem, just as another regular hidden layer of k nodes

  2) For classification problem, apply the softmax function on the vector $\mathbf{z}$ to ensure that the output is the probability vector for k classes.

After constructing the mapping, in supervised problem, the loss function $J(W, b; \mathbf{x}, y)$ can be chosen the same as other machine learning models.

- For regression: MSE (mean squared error)

- For classification: Cross Entropy

Interesting Facts (not required in exam):

- Logistic regression can be viewed as "the simplest" NN with only the input layer (p nodes) and output layer (softmax output). Because it only has "one layer of parameters", sometimes it is also called single-layer NN.

- NN can also be used in unsupervised tasks, such as Autoencoder for dimension reduction.

Training Algorithms: BP (Back Propagation) and SGD (or other optimization method)

As we've known well, training the machine learning model is largely dependent on minimizing the loss function, where its gradient provides the important information.

Intuitively speaking, **back propagation** is just the right algorithm to find such gradient for the training of NN, which is based on the **chain-rule** of derivatives in hierarchical network-structured composite function.

In detail, the backpropagation algorithm for NN training can be described as (not required):

> Step 1: Perform a **forward pass**, computing the values for layers L_2, L_3, and so on up to the output layer L_{n_l}. We take the loss of MSE in regression as example.
>
> Step 2: For each output unit i in layer n_l (the output layer), set
>
> $$\delta_i^{(n_l)} := \frac{\partial}{\partial z_i^{(n_l)}} J(W,b;\mathbf{x},y) = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2}\|y - h(\mathbf{x};W,b)\|^2 = -(y - a_i^{(n_l)}) \cdot \sigma'(z_i^{(n_l)})$$
>
> Step 3: For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ For each node i in layer l, set
>
> $$\delta_i^{(l)} := \frac{\partial}{\partial z_i^{(l)}} J(W,b;\mathbf{x},y) = \frac{\partial}{\partial a_i^{(l)}} J(W,b;\mathbf{x},y) \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = \left(\sum_{j=1}^{s_{l+1}} w_{ji}^{(l)} \delta_j^{(l+1)}\right) \sigma'(z_i^{(l)})$$
>
> Step 4: the desired partial derivatives are computed as:
>
> $$\begin{aligned} \frac{\partial}{\partial w_{ij}^{(l)}} J(W,b;\mathbf{x},y) &= a_j^{(l)} \delta_i^{(l+1)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W,b;\mathbf{x},y) &= \delta_i^{(l+1)}. \end{aligned}$$

After obtaining the gradient with BP, we can then use stochastic gradient descent (SGD, or other methods) to minimize the loss function.

Deep Learning: What packages/platforms to choose?

By making the NN "deeper" (increase the hidden layers and add other complex structures), we have the deep learning models. Of course, deep NN will face the serious problem of overfitting (can be alleviated by regulariztion or dropout) and huge computation cost (can be solved by using GPU instead of CPU).

The neural network models in scikit-learn does not provide the flexible support for large-scale deep learning applications.

For beginners, Keras (high-level API that supports tensorflow, which developed by Google) and Pytorch (developed by Facebook) are the most accessible and popular deep learning packages in

Python.

To get acess to free GPU resources for training deep learning models (see here for why GPU is widely applied in deep learning), you can use the online notebook provided by Kaggle or google colab. If you are using very basic Keras or Pytorch, there's no need to change the code much to adopt to GPU computation. With GPU, you can even give up sklearn and replace much of your codes using the GPU-version machine learning algorithms with the package cuml.

Exploring Deep Learning with Keras

Let's build our first "deep learning" model (indeed a simple Neural Network model trained with deep learning package)

In [ ]:

```
pip install tensorflow
```

You can also setup the gpu if it is supported in your local computer -- unfortunately, recent versions of mac OS are no longer supported. https://www.tensorflow.org/install/gpu

In [3]:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.
npz
11493376/11490434 [==============================] - 0s 0us/step
```

In [4]:

```
# sequential model to define a graph of neural network
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),# input Layer
  tf.keras.layers.Dense(128, activation='relu'), # hidden Layer
  tf.keras.layers.Dense(64, activation='relu'), # hidden Layer
  tf.keras.layers.Dense(10, activation='softmax') # output Layer for classification
])
```

In [5]:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten (Flatten)            (None, 784)               0         
_________________________________________________________________
dense (Dense)                (None, 128)               100480    
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256      
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650       
=================================================================
Total params: 109,386
```

```
Trainable params: 109,386
Non-trainable params: 0
_________________________________________________________________
```

In [6]:
```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
#'adam' is an algorithm for SGD
```

In [ ]:
```
model.fit(x_train, y_train, epochs=100, batch_size = 2048)
model.evaluate(x_test, y_test)
```

We can also try autoencoder for the dimension reduction

In [11]:
```
from tensorflow.keras import layers
# functional API to define a graph of neural network -- more flexible than sequential

input_img = keras.Input(shape=(784,)) # input layer
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(32, activation='relu')(encoded)
encoded = layers.Dense(8, activation='relu')(encoded)
encoded = layers.Dense(2, activation='relu')(encoded)

decoded = layers.Dense(8, activation='relu')(encoded)
decoded = layers.Dense(32, activation='relu')(decoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)
```

In [12]:
```
autoencoder = keras.Model(input_img, decoded) # builds up the model
autoencoder.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_3 (InputLayer)         [(None, 784)]             0         
_________________________________________________________________
dense_3 (Dense)              (None, 128)               100480    
_________________________________________________________________
dense_4 (Dense)              (None, 32)                4128      
_________________________________________________________________
dense_5 (Dense)              (None, 8)                 264       
_________________________________________________________________
dense_6 (Dense)              (None, 2)                 18        
_________________________________________________________________
dense_7 (Dense)              (None, 8)                 24        
_________________________________________________________________
dense_8 (Dense)              (None, 32)                288       
_________________________________________________________________
dense_9 (Dense)              (None, 128)               4224      
_________________________________________________________________
dense_10 (Dense)             (None, 784)               101136    
=================================================================
Total params: 210,562
Trainable params: 210,562
Non-trainable params: 0
_________________________________________________________________
```

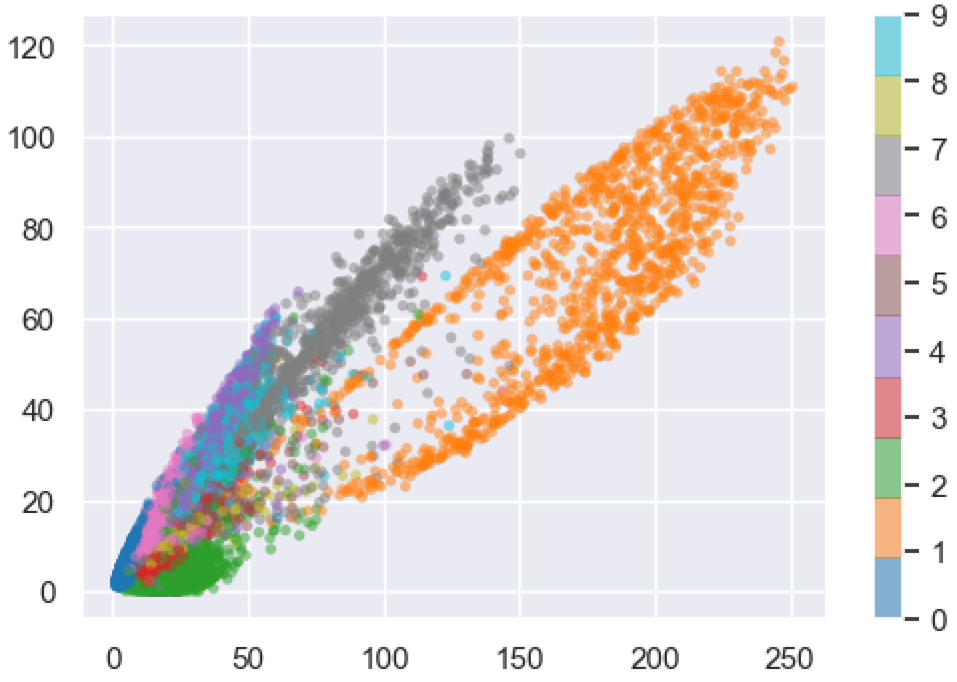
In [ ]:
```
autoencoder.compile(optimizer='adam', loss= tf.keras.losses.MeanSquaredError())

autoencoder.fit(x_test.reshape((len(x_test), np.prod(x_test.shape[1:]))), x_test.reshap
                epochs=200,
                batch_size=1024)
```

In [ ]:
```
encoder = keras.Model(input_img, encoded) # also define the encoder part
encoder.get_weights()
```

In [15]:
```
encoded_coord = encoder.predict(x_test.reshape((len(x_test), np.prod(x_test.shape[1:]))
```

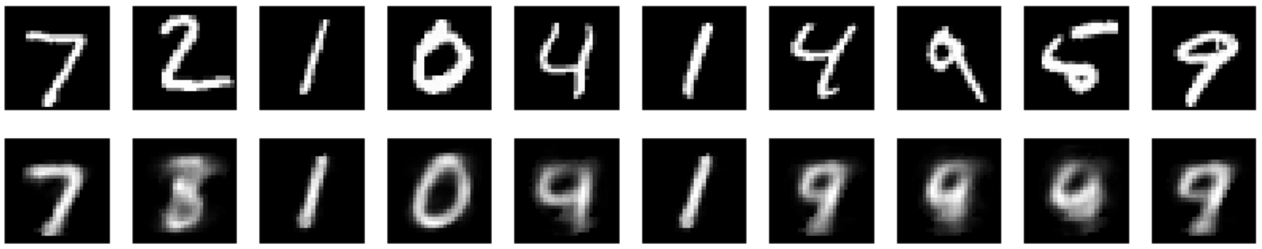
In [16]:
```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
fig = plt.figure(dpi=100)
plt.scatter(encoded_coord[:, 0], encoded_coord[:, 1],c=y_test, s=15, edgecolor='none',
plt.colorbar()
```

Out[16]:
```
<matplotlib.colorbar.Colorbar at 0x201187955e0>
```



In [17]:
```
decoded_imgs = autoencoder.predict(x_test.reshape((len(x_test), np.prod(x_test.shape[1:
```

In [9]:
```
n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i])
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



Maybe condensing to 2D layer loses too much information. Let's try a more realistic model

In [18]:

```
from tensorflow.keras import layers
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)

decoded = layers.Dense(64, activation='relu')(encoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)

autoencoder = keras.Model(input_img, decoded)
autoencoder.summary()
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_4 (InputLayer)         [(None, 784)]             0         
_________________________________________________________________
dense_11 (Dense)             (None, 128)               100480    
_________________________________________________________________
dense_12 (Dense)             (None, 64)                8256      
_________________________________________________________________
dense_13 (Dense)             (None, 32)                2080      
_________________________________________________________________
dense_14 (Dense)             (None, 64)                2112      
_________________________________________________________________
dense_15 (Dense)             (None, 128)               8320      
_________________________________________________________________
dense_16 (Dense)             (None, 784)               101136    
=================================================================
Total params: 222,384
Trainable params: 222,384
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

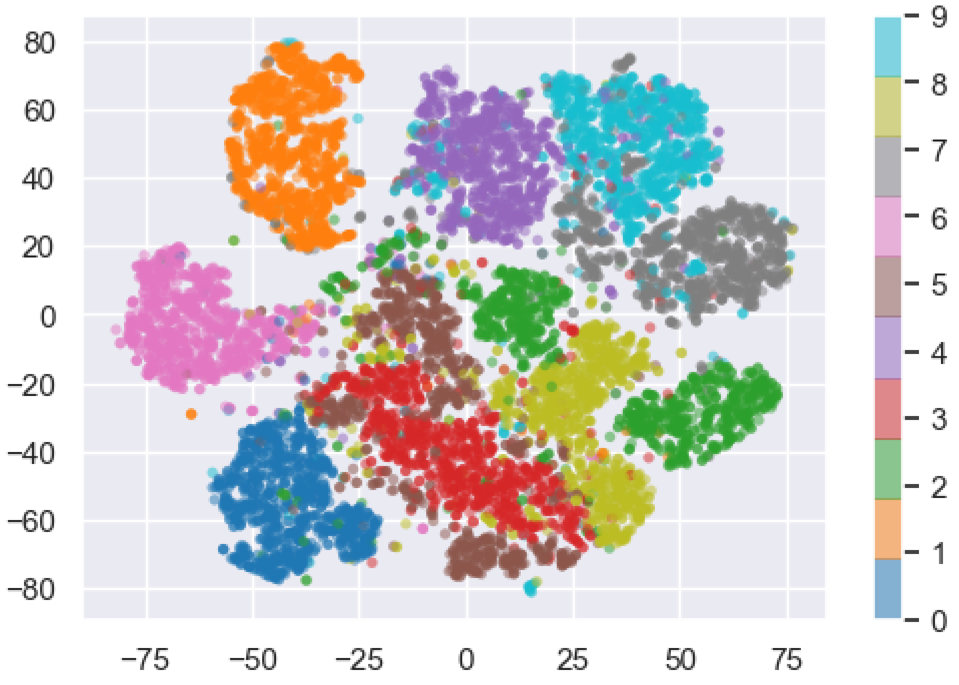
```
autoencoder.compile(optimizer='adam', loss= tf.keras.losses.MeanSquaredError())

autoencoder.fit(x_test.reshape((len(x_test), np.prod(x_test.shape[1:]))), x_test.reshap
                epochs=200,
                batch_size=256)
```

```
In [20]: encoder = keras.Model(input_img, encoded)
         encoded_coord = encoder.predict(x_test.reshape((len(x_test), np.prod(x_test.shape[1:]))
         encoded_coord.shape
```

```
Out[20]: (10000, 32)
```

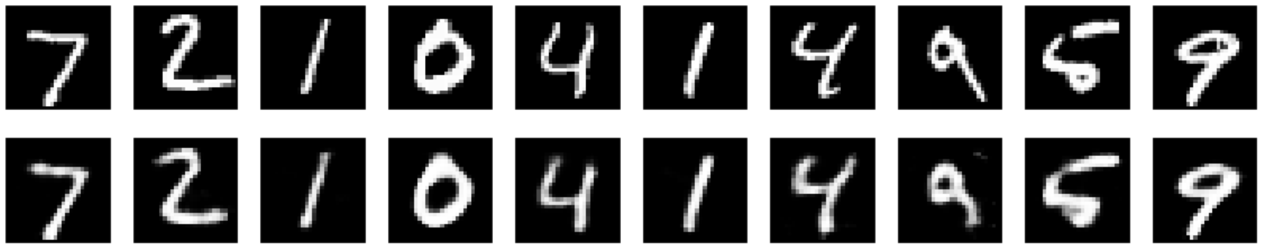
```
In [21]: from sklearn.manifold import TSNE # for GPU acceleration, use the cuml
         tsne = TSNE(n_jobs = -1)
         X_tsne = tsne.fit_transform(encoded_coord)
         import matplotlib.pyplot as plt
         import seaborn as sns; sns.set()
         fig = plt.figure(dpi=100)
         plt.scatter(X_tsne[:, 0], X_tsne[:, 1],c=y_test, s=15, edgecolor='none', alpha=0.5,cmap
         plt.colorbar()
```

```
Out[21]: <matplotlib.colorbar.Colorbar at 0x20113bd00a0>
```



```
In [22]: decoded_imgs = autoencoder.predict(x_test.reshape((len(x_test), np.prod(x_test.shape[1:
         n = 10  # How many digits we will display
         plt.figure(figsize=(20, 4))
         for i in range(n):
             # Display original
             ax = plt.subplot(2, n, i + 1)
             plt.imshow(x_test[i])
             plt.gray()
             ax.get_xaxis().set_visible(False)
             ax.get_yaxis().set_visible(False)

             # Display reconstruction
             ax = plt.subplot(2, n, i + 1 + n)
             plt.imshow(decoded_imgs[i].reshape(28, 28))
             plt.gray()
             ax.get_xaxis().set_visible(False)
```

```
    ax.get_yaxis().set_visible(False)
plt.show()
```



Other Useful References and Suggestions for Future Learning in Data Science

Websites for Everyday Practice

- Python Programming: Leetcode
- Data Science: Kaggle

Courses and Books

- UCI ICS 31-33 (Python Programming), CS 178 (Machine Learning)
- Berkeley Stat 157
- Stanford CS 231
- The Deep Learning Book

```
In [ ]:
```