Lecture 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 (https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.45692&showTestData=false&discretize=false

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
$$\mathbf{z}^{(l+1)} = W^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)}$$
$$\mathbf{a}^{(l+1)} = \sigma(\mathbf{z}^{(l+1)})$$
where σ is the nonlinear activation function (https://en.wikipedia.org/wiki/Activation_function) (common choices include sigmoid, tanh or ReLU), and the second equation is element-wize.

  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 (https://en.wikipedia.org/wiki/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 choosed 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" (https://www.datasciencecentral.com/profiles/blogs/logistic-regression-as-a-neural-network#:~:text=To%20recap%2C%20Logistic%20regression%20is,a%20one%2Dlayer%20neural%20network.), sometimes it is also called single-layer NN (http://yann.lecun.com/exdb/mnist/).
- NN can also be used in unsupervised tasks, such as Autoencoder (https://en.wikipedia.org/wiki/Autoencoder#:~:text=An%20autoencoder%20is%20a%20type,to%20ignore%20signal%20%E2 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:
> $$\frac{\partial}{\partial w_{ij}^{(l)}} J(W, b; \mathbf{x}, y) = a_j^{(l)} \delta_i^{(l+1)}$$
> $$\frac{\partial}{\partial b_i^{(l)}} J(W, b; \mathbf{x}, y) = \delta_i^{(l+1)}.$$

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 (https://scikit-learn.org/stable/modules/neural_networks_supervised.html) in scikit-learn does not provide the flexible support for large-scale deep learning applications.

For beginners, Keras (https://keras.io/) (high-level API that supports tensorflow (https://www.tensorflow.org/), which developed by Google) and Pytorch (https://pytorch.org/) (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 (https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/)), you can use the online notebook provided by Kaggle or google colab (https://colab.research.google.com/notebooks/intro.ipynb). If you are using very basic Keras or Pytorch, there's no need to change the code 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 (https://github.com/rapidsai/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 --upgrade pip
```

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. http
s://www.tensorflow.org/install/gpu
```

In [1]:
```
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
```

In [ ]:
```
# 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 [ ]:
```
model.summary()
```

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

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 [2]:
```
from 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 [3]:
```
autoencoder = keras.Model(input_img, decoded) # builds up the model
autoencoder.summary()
```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dense_1 (Dense)              (None, 32)                4128
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 264
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 18
_________________________________________________________________
dense_4 (Dense)              (None, 8)                 24
_________________________________________________________________
dense_5 (Dense)              (None, 32)                288
_________________________________________________________________
dense_6 (Dense)              (None, 128)               4224
_________________________________________________________________
dense_7 (Dense)              (None, 784)               101136
=================================================================
Total params: 210,562
Trainable params: 210,562
Non-trainable params: 0
_________________________________________________________________
```

In [10]:
```
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.resh
ape((len(x_test), np.prod(x_test.shape[1:]))),
                epochs=100,
                batch_size=1024,
                shuffle=True)
```

```
Epoch 1/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0440
Epoch 2/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0431
Epoch 3/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0429
Epoch 4/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0428
Epoch 5/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0428
Epoch 6/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0427
Epoch 7/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0427
Epoch 8/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0427
Epoch 9/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0427
Epoch 10/100
10/10 [==============================] - 0s 7ms/step - loss: 0.0427
Epoch 11/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0426
Epoch 12/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0426
Epoch 13/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0426
Epoch 14/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0426
Epoch 15/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0426
Epoch 16/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0426
Epoch 17/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0425
Epoch 18/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0425
Epoch 19/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0425
Epoch 20/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0425
Epoch 21/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0425
Epoch 22/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0425
Epoch 23/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0424
Epoch 24/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0424
Epoch 25/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0424
Epoch 26/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0424
Epoch 27/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0424
Epoch 28/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0423
Epoch 29/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0423
Epoch 30/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0423
Epoch 31/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0423
Epoch 32/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0423
Epoch 33/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0423
Epoch 34/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0423
Epoch 35/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0423
```

```
Epoch 36/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0422
Epoch 37/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0422
Epoch 38/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0422
Epoch 39/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0422
Epoch 40/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0421
Epoch 41/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0421
Epoch 42/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0421
Epoch 43/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0421
Epoch 44/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0421
Epoch 45/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0420
Epoch 46/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0420
Epoch 47/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0420
Epoch 48/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0420
Epoch 49/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0420
Epoch 50/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0420
Epoch 51/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0420
Epoch 52/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0419
Epoch 53/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0419
Epoch 54/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0419
Epoch 55/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0419
Epoch 56/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0419
Epoch 57/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0418
Epoch 58/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0418
Epoch 59/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0418
Epoch 60/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0418
Epoch 61/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0418
Epoch 62/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0418
Epoch 63/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0418
Epoch 64/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0417
Epoch 65/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0417
Epoch 66/100
10/10 [==============================] - 0s 8ms/step - loss: 0.0417
Epoch 67/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0417
Epoch 68/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0417
Epoch 69/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0416
Epoch 70/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0416
```

```
Epoch 71/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0416
Epoch 72/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0416
Epoch 73/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0416
Epoch 74/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0416
Epoch 75/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0415
Epoch 76/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0416
Epoch 77/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0415
Epoch 78/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0415
Epoch 79/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0415
Epoch 80/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0415
Epoch 81/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0414
Epoch 82/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0414
Epoch 83/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0414
Epoch 84/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0414
Epoch 85/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0414
Epoch 86/100
10/10 [==============================] - 0s 11ms/step - loss: 0.0414
Epoch 87/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0414
Epoch 88/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0414
Epoch 89/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0413
Epoch 90/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0413
Epoch 91/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0413
Epoch 92/100
10/10 [==============================] - 0s 11ms/step - loss: 0.0413
Epoch 93/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0413
Epoch 94/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0412
Epoch 95/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0412
Epoch 96/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0412
Epoch 97/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0412
Epoch 98/100
10/10 [==============================] - 0s 10ms/step - loss: 0.0412
Epoch 99/100
10/10 [==============================] - 0s 11ms/step - loss: 0.0411
Epoch 100/100
10/10 [==============================] - 0s 9ms/step - loss: 0.0411
```

Out[10]: <tensorflow.python.keras.callbacks.History at 0x7ffb282a3390>

In [11]:
```
encoder = keras.Model(input_img, encoded) # also define the encoder part
encoder.get_weights()
```

Out[11]:
```
[array([[-0.05947556,  0.02999774,  0.03496894, ...,  0.06997635,
         -0.02443226, -0.05765174],
        [ 0.01859826, -0.05337664,  0.01222368, ...,  0.02130038,
          0.03675849, -0.02764505],
        [-0.00772542,  0.01606809,  0.00638994, ..., -0.0793802 ,
         -0.07776325, -0.03962534],
        ...,
        [-0.00762312,  0.03418265,  0.01706019, ...,  0.04284579,
          0.02790998,  0.01960173],
        [ 0.07160843,  0.01389702,  0.04417362, ..., -0.0403284 ,
          0.04575133,  0.07278159],
        [-0.06782561,  0.07357565, -0.05206133, ...,  0.07031503,
          0.06448487,  0.06966222]], dtype=float32),
 array([ 2.40136936e-01,  2.63595849e-01,  6.58800080e-02,  1.70697942e-01,
         1.90887064e-01, -1.34922396e-02, -1.50258336e-02, -8.04297277e-04,
         2.66798466e-01,  5.87142520e-02, -1.74439456e-02, -1.40925134e-02,
         1.82616100e-01, -2.10637636e-02,  3.52401212e-02,  8.98889229e-02,
        -2.07967479e-02,  6.67532953e-03, -1.54011518e-01,  3.65302563e-01,
         7.14327022e-02,  2.77068555e-01,  9.01356488e-02,  2.73365527e-01,
         8.10998529e-02,  3.69344167e-02, -4.15288396e-02,  1.82149887e-01,
         2.59497136e-01,  1.57676488e-01, -1.50681622e-02, -1.59129295e-02,
         1.15949780e-01, -1.66473128e-02,  1.30678877e-01, -1.95026267e-02,
         1.64657667e-01,  1.16247386e-01,  4.94658172e-01,  8.40974331e-01,
         1.04063742e-01,  7.52099603e-02,  7.85653666e-02, -1.55108282e-02,
         9.45800319e-02,  1.40961289e-01,  3.37453216e-01, -1.47448024e-02,
         6.93240538e-02,  8.11441168e-02, -1.66967455e-02,  3.51815403e-01,
         8.29982042e-01,  2.00706646e-01, -1.60036981e-02, -1.79883260e-02,
         3.21504474e-02,  5.07089972e-01,  5.77833988e-02,  2.52949059e-01,
        -1.65693108e-02,  8.99084881e-02,  1.23093031e-01, -1.81887858e-02,
         1.08325332e-01,  1.69933036e-01,  8.36526334e-01, -7.26310983e-02,
        -1.42500782e-02, -1.86911672e-02, -1.57157816e-02,  1.04814932e-01,
         5.00704125e-02, -7.45654758e-03, -1.91297438e-02,  8.39748859e-01,
         1.95849724e-02, -1.27952054e-01,  3.26722600e-02, -1.07801922e-01,
        -1.99855752e-02,  5.35094261e-01,  3.94111693e-01, -1.88885070e-02,
         7.95129463e-02, -1.89713743e-02,  9.54800844e-02,  1.94479153e-01,
         9.27215517e-01, -3.60916206e-03,  1.46221310e-01, -1.46909487e-02,
         2.96686172e-01,  5.00534117e-01,  2.05044940e-01, -1.63684711e-02,
         3.08393478e-01, -2.03671735e-02,  7.88885236e-01,  9.39736590e-02,
         1.35885462e-01,  5.05500972e-01, -1.38311023e-02,  2.68366754e-01,
         1.61574893e-02,  1.81143433e-01,  7.86647275e-02, -2.00592410e-02,
         5.43132126e-01,  7.84727260e-02, -1.54982153e-02, -1.84051245e-02,
         2.19278887e-01,  1.65609524e-01, -1.85344759e-02,  1.77413508e-01,
        -1.84267275e-02,  1.44426495e-01,  5.13321877e-01, -1.79356545e-01,
         3.63828361e-01, -1.41581237e-01,  5.81640601e-01,  1.91846266e-01,
         1.11846365e-01,  1.75797582e-01, -1.73756927e-02, -1.31474081e-02],
       dtype=float32),
 array([[-0.42971066, -0.10588566,  0.4236499 , ...,  0.17884333,
          0.2563132 ,  0.39318204],
        [ 0.34058133,  0.18111633,  0.6851588 , ..., -0.11377961,
         -0.20210741,  0.05846188],
        [-0.09290195, -0.05543005,  0.1760109 , ...,  0.3172078 ,
         -0.15837938, -0.28363746],
        ...,
        [ 0.19722241, -0.10217646,  0.7285515 , ...,  0.00933621,
         -0.16007745,  0.05143999],
        [ 0.09728131,  0.1213326 ,  0.1580048 , ..., -0.09850125,
          0.15406206, -0.03437775],
        [ 0.05645077, -0.044066  , -0.10675615, ..., -0.14322896,
          0.16919908,  0.00784586]], dtype=float32),
 array([ 0.02981751,  0.05343308,  0.8463537 ,  0.15047933,  0.11920236,
         0.09623373,  0.07064325,  0.16425121,  0.16182274,  0.04715717,
         0.6506759 ,  0.13108687,  0.4883804 ,  0.15848926,  0.00482019,
        -0.31285816,  0.37398085,  0.24226628,  0.11423625,  0.33841768,
        -0.00471628,  0.11195443,  0.1449951 ,  0.26420838,  0.0942094 ,
         0.01801389,  0.69322354, -0.00355269,  0.15458152,  0.05959609,
         0.05216391, -0.0085267 ], dtype=float32),
 array([[-0.6085141 ,  0.03899195,  0.11315902, -0.2467418 ,  0.48680893,
          0.14605574, -0.00474455, -0.11922214],
        [ 0.30479074,  0.22602326,  0.20132744,  0.09678176, -0.19986086,
          0.10472327, -0.20857438, -0.1898901 ],
```

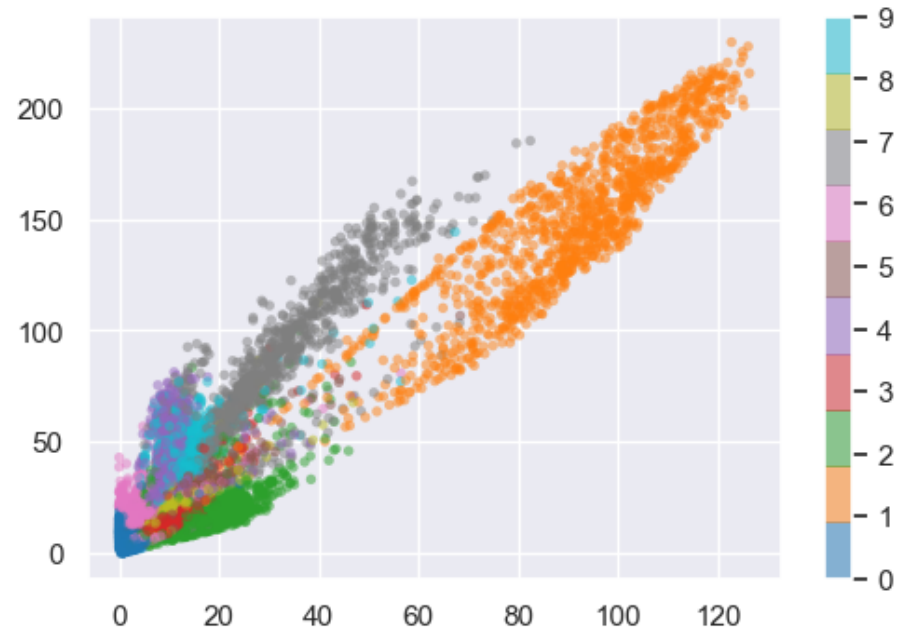
```
       [ 0.34882623, -0.15259178,  0.10349804,  0.6591769 ,  0.37066194,
        -0.36874518, -0.0549324 ,  0.2957819 ],
       [ 0.04735119, -0.29941946,  0.23046184,  0.20572415,  0.3881902 ,
        -0.09417078, -0.2757015 , -0.08009816],
       [ 0.26537007, -0.33547738, -0.28256068,  0.3246253 ,  0.28150526,
         0.06180848, -0.14622097, -0.11724952],
       [-1.212921  , -0.13626626,  0.3213269 , -0.8549572 , -0.19157548,
         0.1196871 ,  0.1278217 , -0.20940109],
       [ 0.5998724 ,  0.2883691 , -0.26682416,  0.13511056, -0.38183102,
         0.20323655, -0.03743069,  0.00243377],
       [ 0.5022182 , -0.22315794,  0.13140947,  0.7602252 , -0.57692957,
        -0.30802935, -0.19044216, -0.12764177],
       [ 0.12463241, -0.04566322,  0.15835342,  0.22225043,  0.38184842,
        -0.36748052, -0.28211766,  0.32382554],
       [ 0.35025162, -0.04421924, -0.38090354,  0.29241297, -0.4230106 ,
        -0.31895205,  0.26706815, -0.08966675],
       [-0.10041384, -0.2908616 ,  0.27280772,  0.55503964,  0.8012284 ,
         0.03395116, -0.06834389, -0.19523159],
       [ 0.2114852 , -0.4409843 ,  0.28901434,  0.18216875,  0.2853055 ,
        -0.32033262, -0.2141065 , -0.26758707],
       [ 0.15769857,  0.31533012,  0.15417017,  0.14640772,  1.164283  ,
        -0.00891351, -0.0305737 , -0.04287997],
       [ 0.11002764, -0.09403574, -0.29326278,  0.22479264,  0.37642097,
        -0.14539514, -0.36143002, -0.2124193 ],
       [ 0.0845254 ,  0.2774043 , -0.14014336, -0.0171426 , -0.06502149,
         0.29031572, -0.33567303, -0.06908435],
       [-0.49619022, -0.12817632, -0.10835654, -0.7938073 , -0.5298633 ,
        -0.24586692, -0.19864716, -0.18655439],
       [-0.18879806,  0.00611608, -0.3097756 ,  0.33402085,  0.7012244 ,
        -0.31589657,  0.27278802, -0.08294316],
       [-0.07817194, -0.18683615,  0.09363309,  0.21440609,  0.27658594,
        -0.30960208,  0.10105172, -0.36631426],
       [ 0.31402916,  0.21293145,  0.12690403,  0.13991018,  0.15654437,
        -0.06893557,  0.12931187,  0.31468397],
       [ 0.661512  , -0.11528885, -0.23797555,  0.4867709 ,  0.699805  ,
        -0.07210054,  0.05877442, -0.21594884],
       [-0.18049814,  0.21910514, -0.304394  ,  0.33117747, -0.35793006,
         0.15894985,  0.17106895, -0.00456983],
       [ 0.2837629 , -0.27526158,  0.1055413 ,  0.12182213, -0.525184  ,
        -0.05791169,  0.27287927, -0.03626702],
       [ 0.25328392, -0.12132675, -0.31639478,  0.06901262,  0.21944529,
         0.06651714,  0.3742397 , -0.01784094],
       [ 0.17185438, -0.29339054,  0.02275661,  0.26075104,  0.92167854,
         0.01810432, -0.22386253,  0.09655984],
       [ 0.34254986,  0.01558863,  0.0189773 ,  0.26810467,  0.06011497,
        -0.18075311, -0.30559248, -0.05939092],
       [-0.60785216,  0.20331438,  0.3734076 , -0.12137841,  1.1800994 ,
         0.09808115,  0.09980026,  0.2237215 ],
       [ 0.28627348,  0.3116778 , -0.38112402,  0.40124303,  0.8453218 ,
        -0.07385413, -0.34313124, -0.2530877 ],
       [-0.12756014, -0.39647737, -0.01737841, -0.10917541, -0.43318027,
         0.08204249,  0.18090135, -0.21397306],
       [-0.2683426 , -0.0207768 ,  0.12093055,  0.41233936,  0.22287424,
         0.14867495, -0.20313665, -0.14027984],
       [-0.14774525, -0.00727039,  0.31838563, -0.01514569, -0.6983138 ,
        -0.31751183, -0.1617338 , -0.09526449],
       [-0.5374191 , -0.10106187, -0.10516742, -0.49867204,  0.31834292,
        -0.3419259 , -0.30635524,  0.02560506],
       [-0.7228207 , -0.2751459 , -0.24725819, -0.7852104 ,  0.78232145,
        -0.33753145, -0.19054532, -0.12788124]], dtype=float32),
array([ 9.0897024e-02, -3.0559542e-02, -8.3352752e-05,  8.8079989e-02,
        5.4166859e-01, -8.6392006e-03, -1.1247671e-02,  8.6889137e-03],
      dtype=float32),
array([[-0.2846781 ,  0.53235614],
       [ 0.5957006 , -0.37706986],
       [ 0.34214923, -0.22898577],
       [-0.2685487 ,  0.6061868 ],
       [ 0.83776647,  0.4639623 ],
       [ 0.42550644, -0.7541872 ],
       [ 0.45924413, -0.05488669],
```

```
        [ 0.5632348 ,  0.23399091]], dtype=float32),
 array([0.30013943, 0.01383021], dtype=float32)]
```

In [12]:
```
encoded_coord = encoder.predict(x_test.reshape((len(x_test), np.prod(x_test.shape[1
:]))))
```

In [13]:
```
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'
, alpha=0.5,cmap=plt.cm.get_cmap('tab10', 10))
plt.colorbar()
```

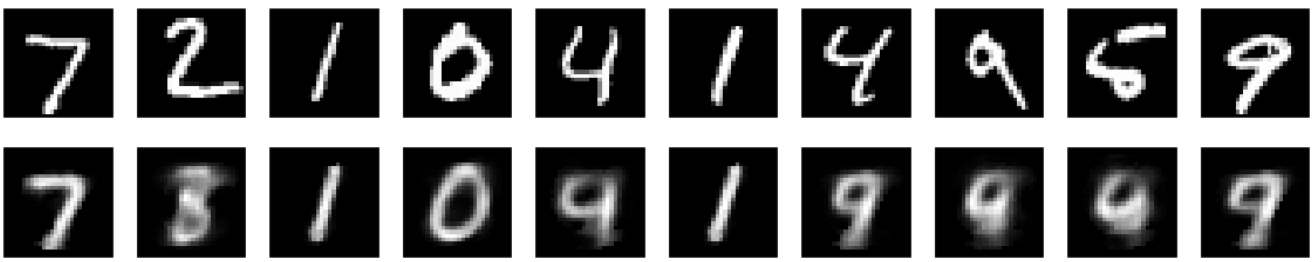
Out[13]:
```
<matplotlib.colorbar.Colorbar at 0x7ffb2d469610>
```



In [8]:
```
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 [ ]:
```
from 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()
```

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.resh
ape((len(x_test), np.prod(x_test.shape[1:]))),
                epochs=100,
                batch_size=256,
                shuffle=True)
```

In [ ]:
```
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
```

In [ ]:
```
from sklearn.manifold import TSNE # for GPU acceleration, use the
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,cm
ap=plt.cm.get_cmap('tab10', 10))
plt.colorbar()
```

In [ ]:
```
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

- Stanford CS231 (http://cs231n.stanford.edu/)
- The Deep Learning Book (https://www.deeplearningbook.org/)