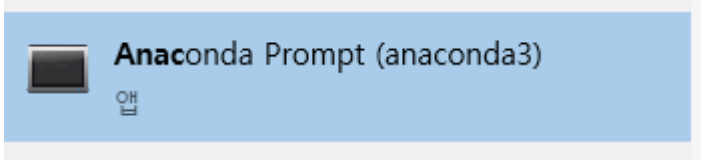
MLP Practice Through “Tensorflow”

Hanwool Jeong

hwjeong@kw.ac.kr

Tensorflow Library Setup

- https://www.tensorflow.org/
- Run Anaconda Prompt



- Type the following commands to the terminal:

> conda update –n base conda

> conda update ––all

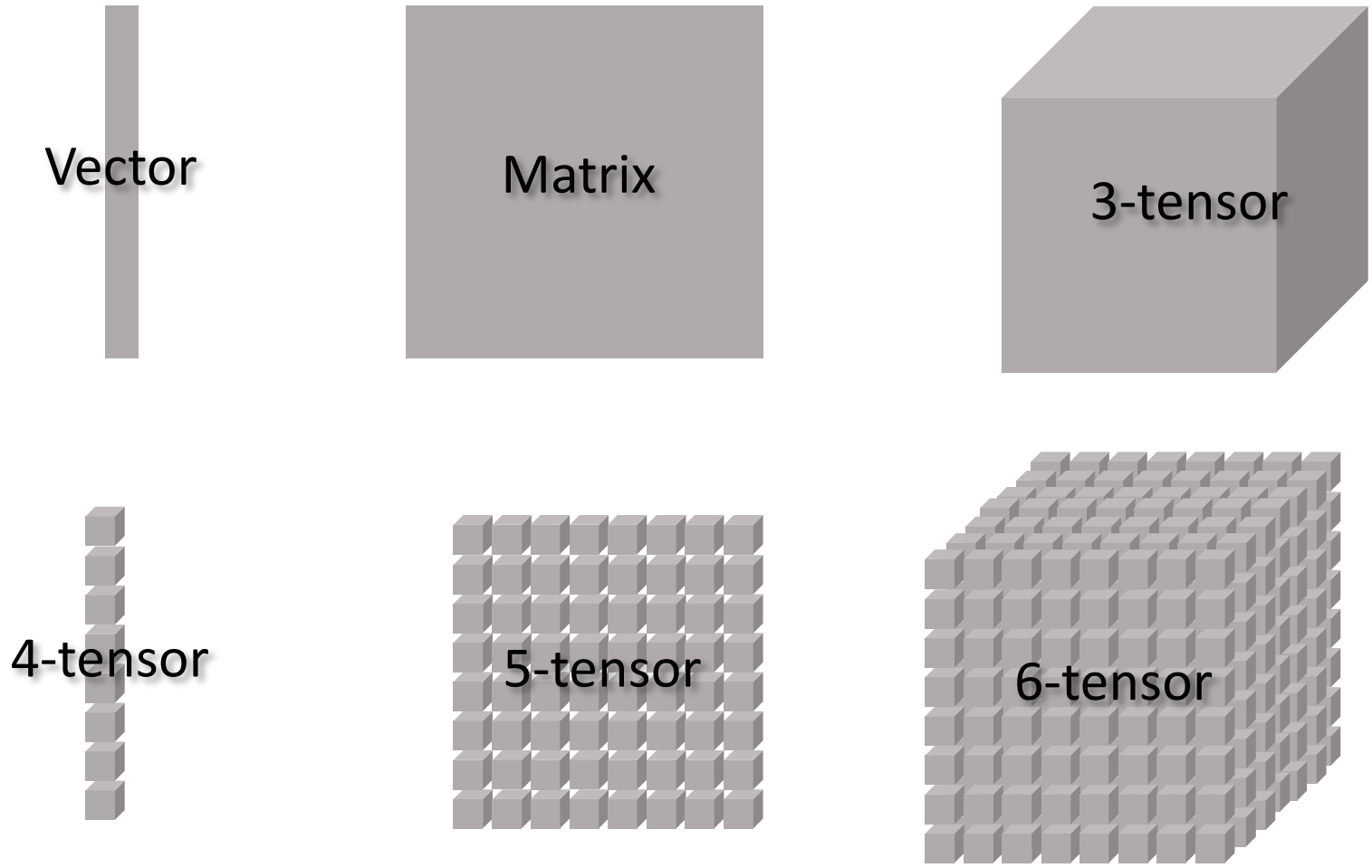
> conda install tensorflow

Concept of Tensor

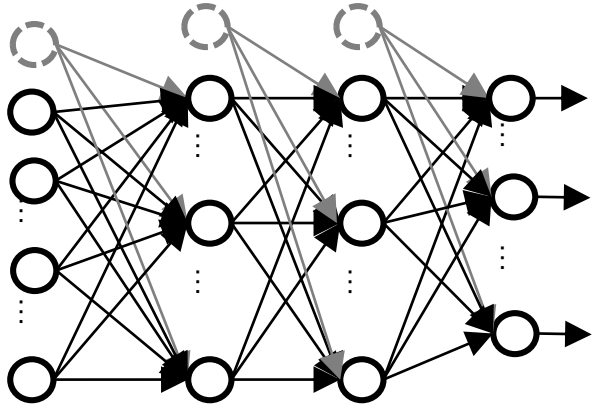
- Scalar, vector, matrix, .. What's next?
- Generally, we need represent higher dimensional data form, we can call them using general term "**tensor**".

Rank	Type	Tensor
0	Scalar	0-tensor
1	Vector	1-tensor
2	Matrix	2-tensor
3	…?	3-tensor
4		4-tensor
…		…
N		N-tensor

Visualization of Tensor Concept



What Tensorflow Can Do

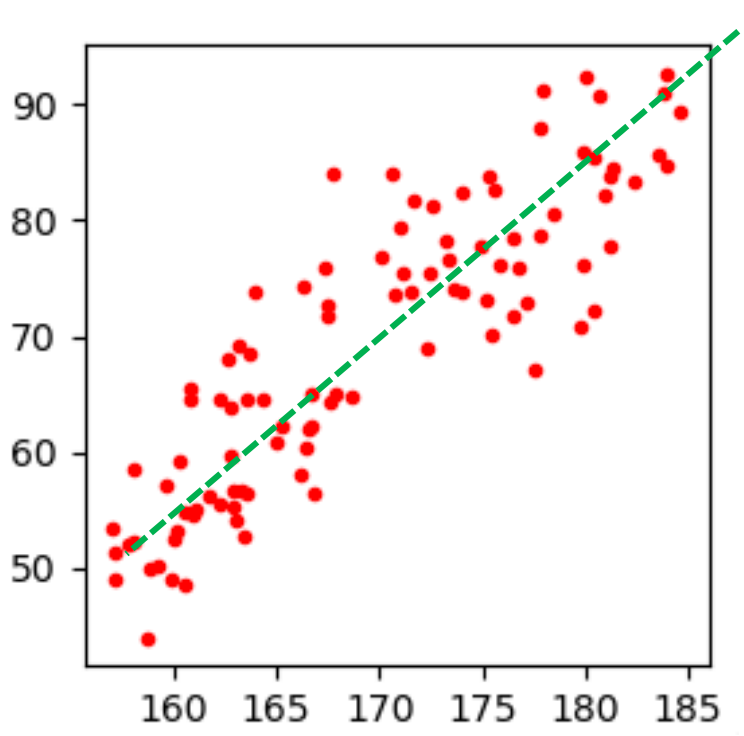


Tensor → Trainable Neural Network → Tensor

- ✓ **Tensors** are gradually processed for achieving a certain goal
- ✓ We can allocate **specialized HW** for running tensorflow

Simple Example;
Linear Regression using Tensorflow

- You remember linear regression dataset we practiced?
- https://raw.githubusercontent.com/hanwoolJeong/lectureUniv/main/testData_H_vs_W.txt



$\hat{y} = \mathbf{w}^T\mathbf{x}$

Note that **x** is 1 padded feature

Revisit Optimal Weight in Linear Regression

$$\mathrm{MSE}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2 \rightarrow \nabla MSE(\boldsymbol{w}) = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y})\,\boldsymbol{x}_i = \vec{\mathbf{0}}$$

- Using the normal equation,

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y} \quad \Rightarrow \quad \hat{\mathbf{w}}_{OLS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

- Or we can use the gradient descent method using

$$w_{\text{next}} = w_{\text{present}} - \eta\nabla MSE(\boldsymbol{w})$$

Import Libraries and Load Data w/ Feature Scaling

- Import required libraries and modules:

```
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
from sklearn.preprocessing import StandardScaler
```

- Load data & feature scaling

```
dfLoad =
pd.read_csv("https://raw.githubusercontent.com/hanwoolJeong/lectureUniv/ma
in/testData_H_vs_W.txt", sep ="\s+")
HeightRaw = np.array(dfLoad["Height"]).reshape(-1,1) #Vector to matrix
WeightRaw = np.array(dfLoad["Weight"]).reshape(-1,1)

scaler = StandardScaler()
scaler.fit(HeightRaw)
Height_std = scaler.transform(HeightRaw)

nData = np.shape(HeightRaw)[0]
X_np = np.c_[np.ones(nData), Height_std]
y_np = WeightRaw
```

Tensorflow Syntax for Variable Definition & Gradient Descent

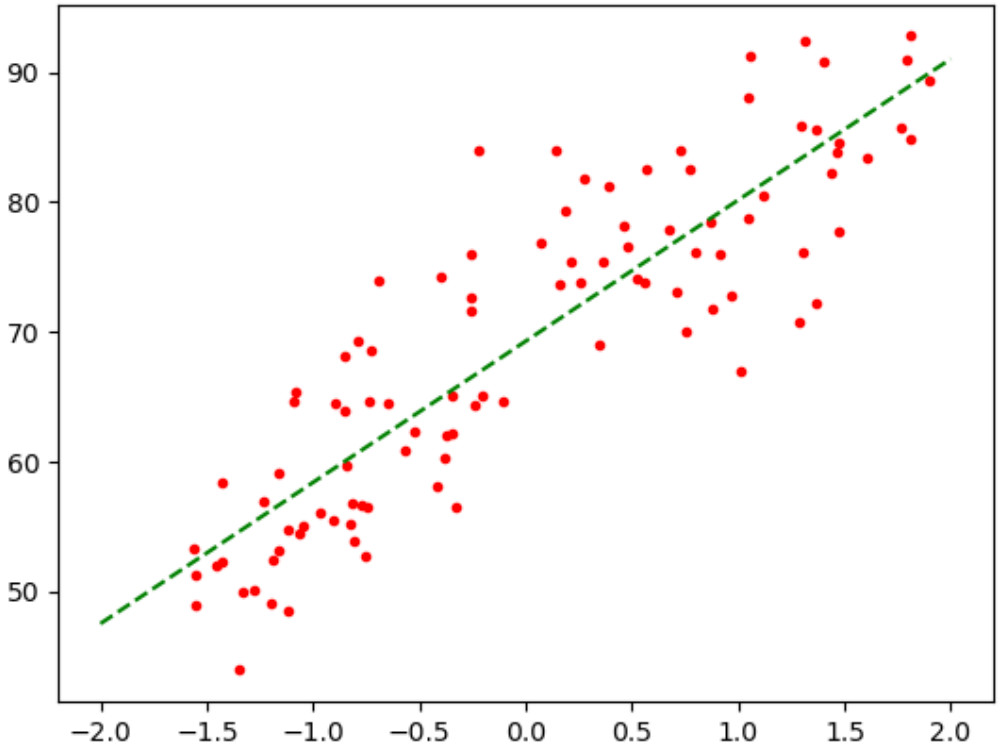
```
X = tf.constant(X_np, dtype=tf.float32)
y = tf.constant(y_np, dtype=tf.float32)
theta = tf.Variable(tf.random.uniform([2,1], -1, 1), dtype=tf.float32)

learning_rate = 0.1
n_epoch = 100
for i in np.arange(n_epoch):
    with tf.GradientTape() as g:
        y_pred = tf.matmul(X, theta)
        error = y - y_pred #nData x 1
        mse = tf.reduce_mean(tf.square(error))
    gradients = g.gradient(mse, [theta])
    theta.assign(theta - learning_rate * gradients[0].numpy())
    if (n_epoch%10 == 0):
        print(theta)
```

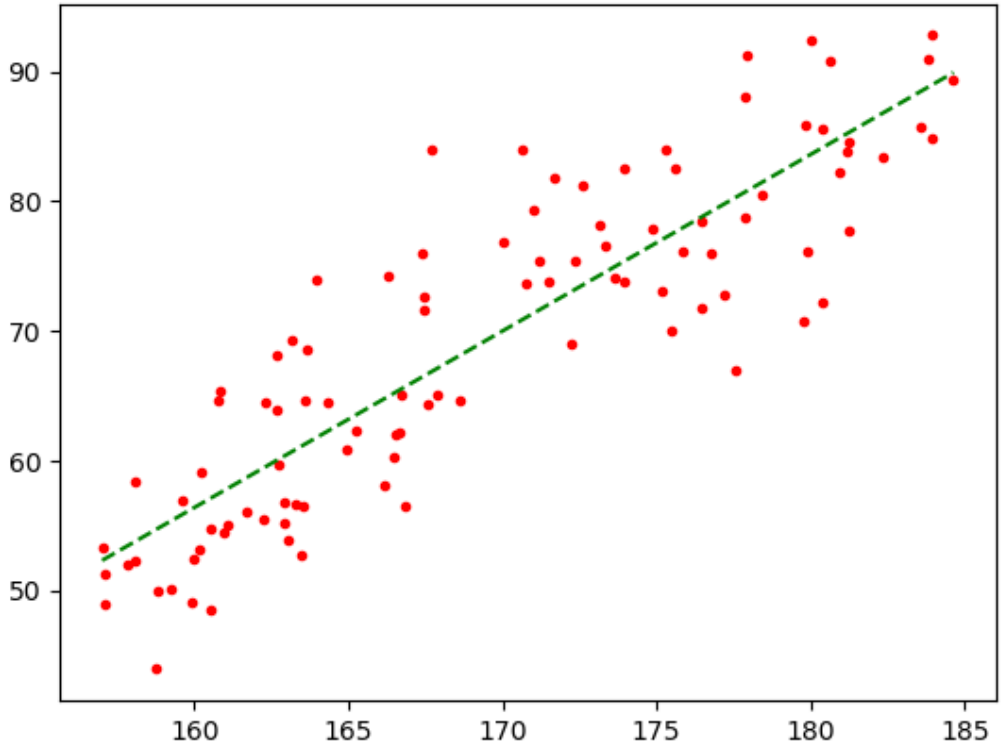
$$\nabla MSE(\boldsymbol{w}) = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \boldsymbol{w}^T\boldsymbol{x}_i)\,\boldsymbol{x}_i = \vec{\mathbf{0}} \quad \Rightarrow \quad w_{\text{next}} = w_{\text{present}} - \eta\nabla MSE(\boldsymbol{w})$$

Visualization

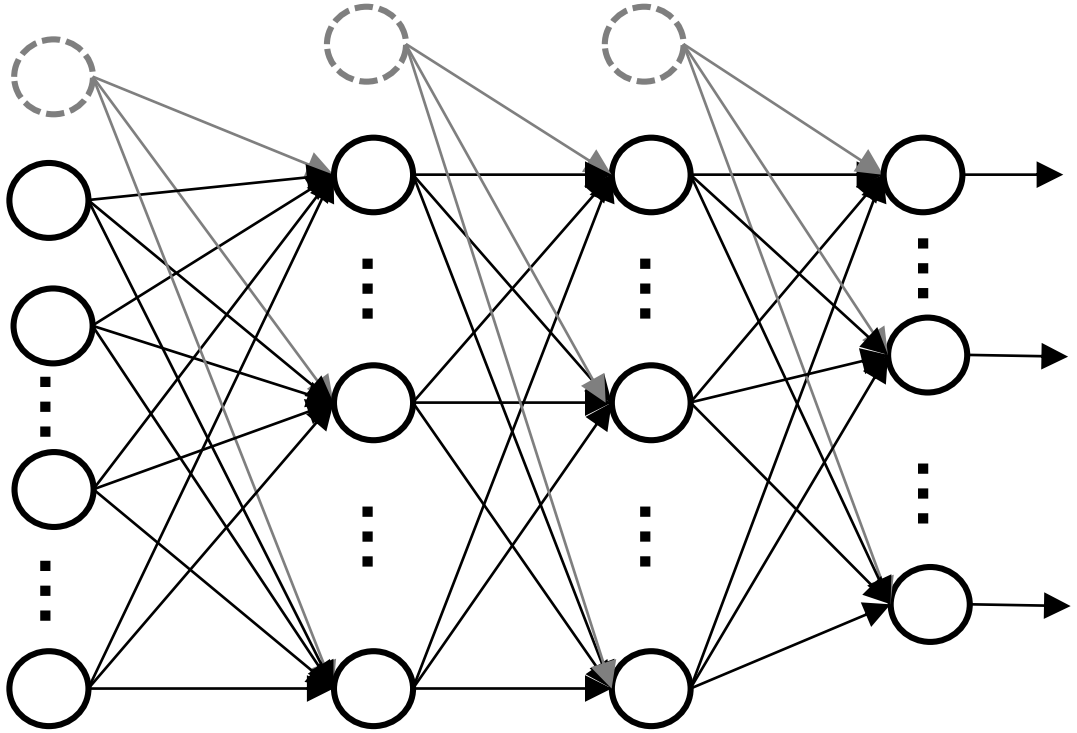
```
w1_std = theta.numpy()[1]
w0_std = theta.numpy()[0]
xPlt_std = np.linspace(-2, 2, 100)
f2, ax2 = plt.subplots()
ax2.plot(Height_std, WeightRaw, "r.")
ax2.plot(xPlt_std, w1_std*xPlt_std+w0_std, "g--")
w1 = w1_std/np.sqrt(scaler.var_)
w0 = w0_std - scaler.mean_*w1_std/np.sqrt(scaler.var_)
xPltRaw = np.linspace(min(HeightRaw), max(HeightRaw), 100)
f3, ax3 = plt.subplots()
ax3.plot(HeightRaw, WeightRaw, "r.")
ax3.plot(xPltRaw, w1*xPltRaw+w0, "g--")
```

Visualization Results

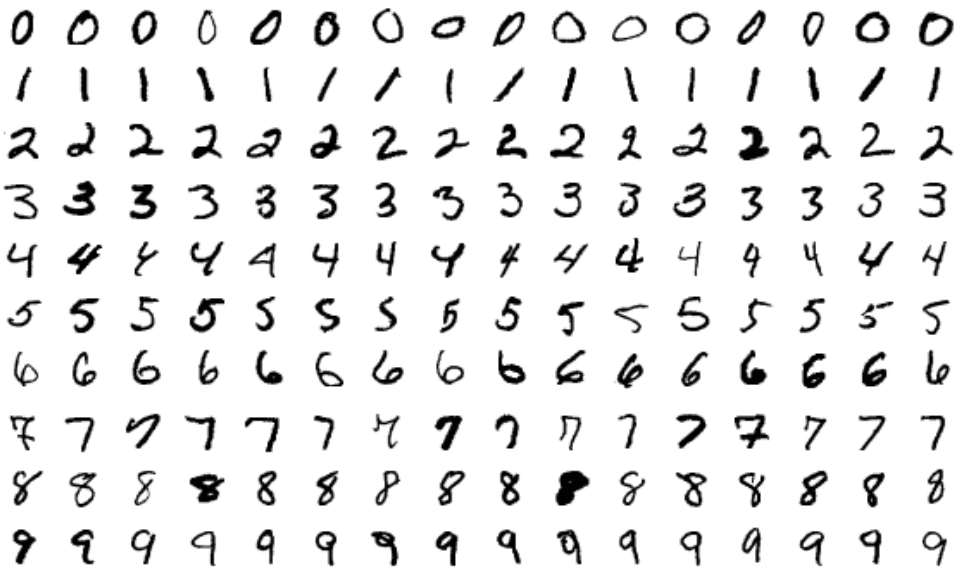


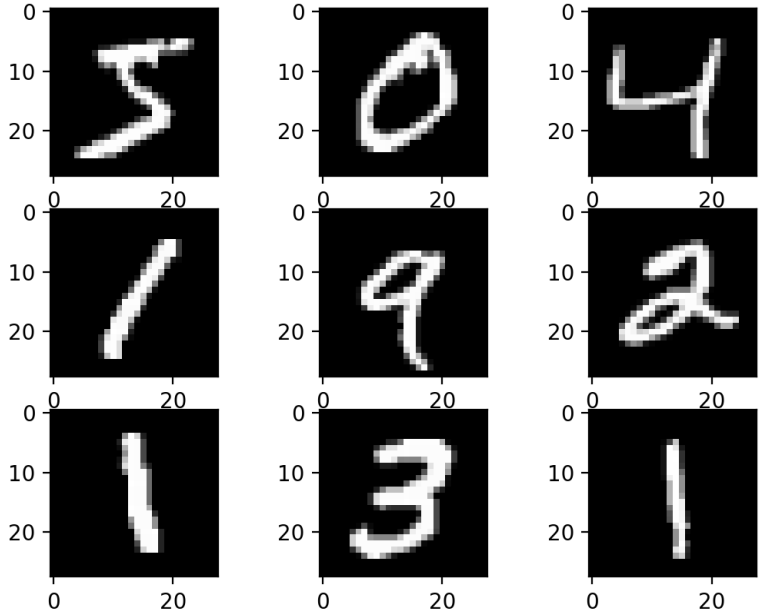


Tensorflow for Deep MLP



MNIST Dataset





28 x 28 pixels for each

One pixel : 0 ~ 255

- **Load MNIST Dataset:**

Let's build up a deep MLP classifier for MNIST!

```
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
```

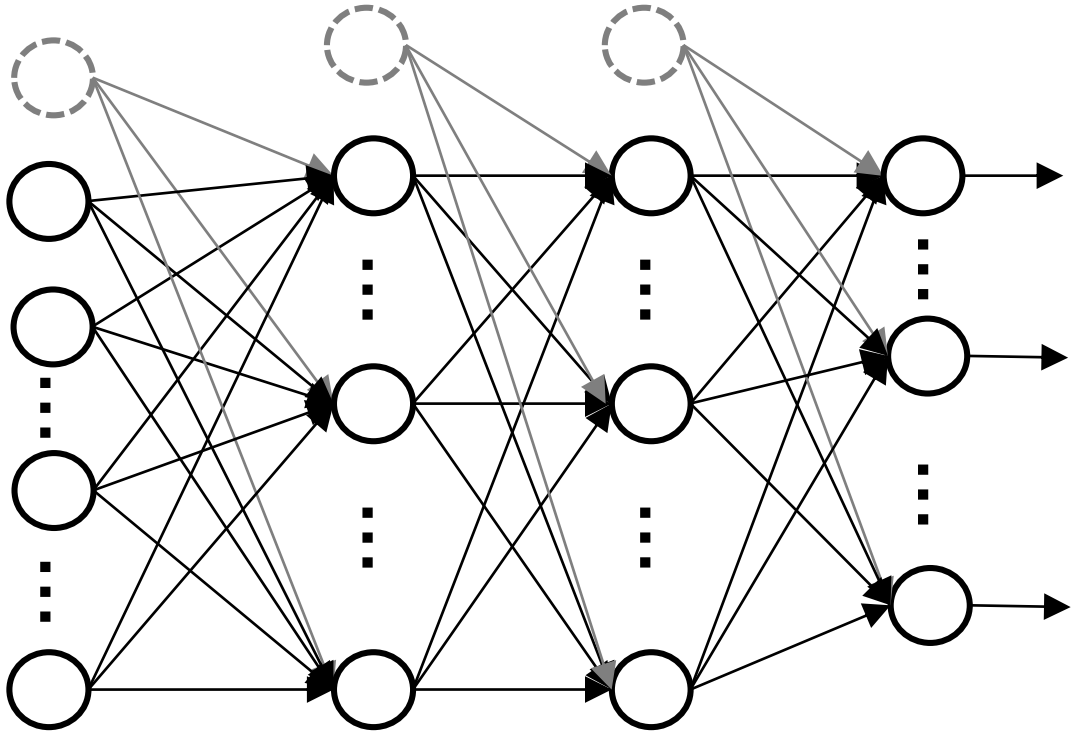
Wait!!
Revisit Limitations of deep MLP

- Large computations
- Vanishing gradients
- Overfitting issues

1) Cost function improvement

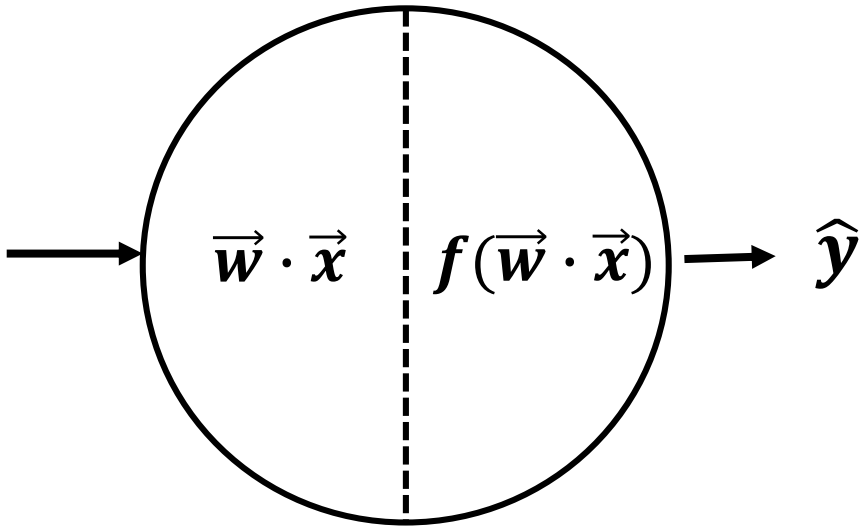
2) Activation function improvement

3) Optimizer improvement



1) Cost Function Improvement; Limitations of MSE Cost(Loss) Function

- Weight update based on the cost minimization?

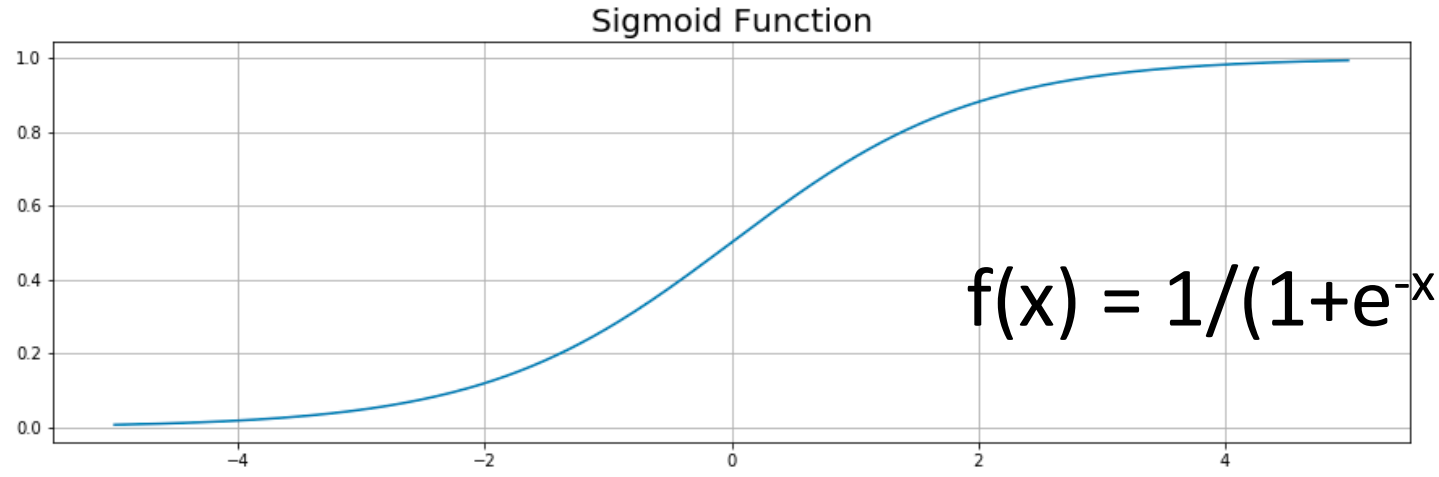


$w_{next} = w_{present} - \eta \nabla_w cost$

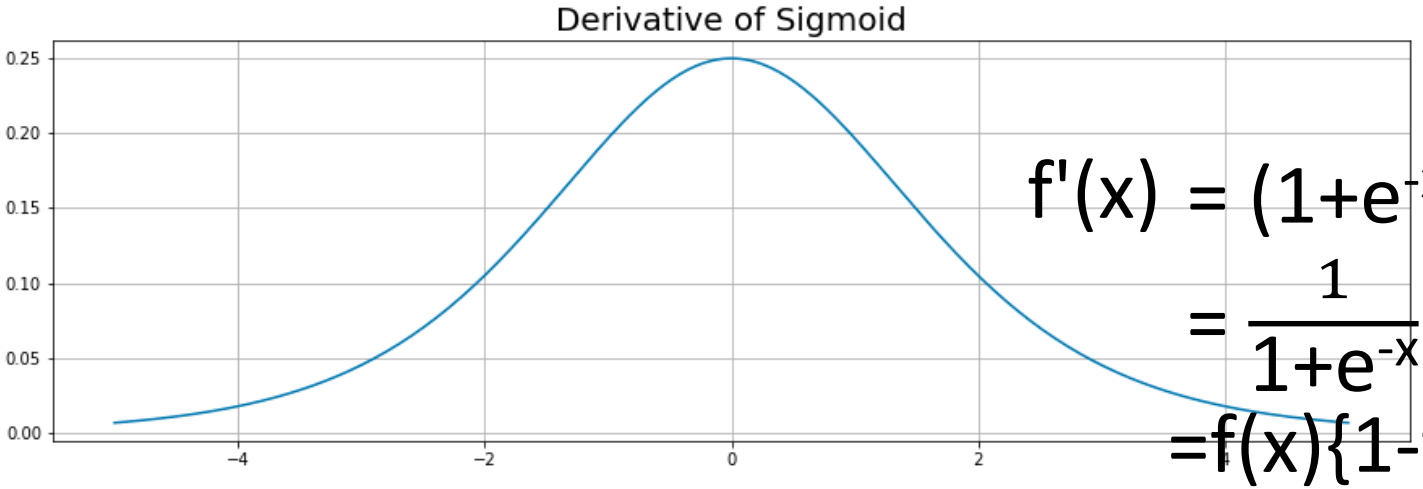
$= w_{present} - \eta \nabla_w MSE$ ← $MSE = (y-\hat{y})^2 = \{y-f(\vec{w} \cdot \vec{x})\}^2$

$= w_{present} + 2\eta \times \{y - f(\vec{w} \cdot \vec{x})\} f'(\vec{w} \cdot \vec{x}) \vec{x}$

1) Cost Function Improvement; Revisit Sigmoid Function



$$f(x) = 1/(1+e^{-x}) = (1+e^{-x})^{-1}$$



$$f'(x) = (1+e^{-x})^{-2}e^{-x}$$
$$= \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}}$$
$$= f(x)\{1-f(x)\}$$

1) Cost Function Improvement; Cross-Entropy Loss Function

- Loss function is defined as

$$\text{Loss (or cost)} = -\, y_i \log(\hat{y}_i) - (1- y_i)\log(1-\hat{y}_i)$$

where for $y_i = \{0, 1\}$. It works well as the loss.

- During the gradient descent of $w_{next} = w_{present} - \eta \nabla_w \text{loss}$

$$w_{next} = w_{present} - \eta \nabla_w \{-y_i \log(\boldsymbol{f}(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}})) - (1-y_i)\log(1-\boldsymbol{f}(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}))\}$$

$$= w_{present} - \eta \nabla_w \{-y_i \frac{\vec{\boldsymbol{x}}\boldsymbol{f}'(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}})}{\boldsymbol{f}(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}})} + (1-y_i)\frac{\vec{\boldsymbol{x}}\boldsymbol{f}'(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}})}{\boldsymbol{1}-\boldsymbol{f}(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}})}\}$$

$$= w_{present} + \eta \nabla_w \vec{\boldsymbol{x}}\boldsymbol{f}'(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}}) \{\frac{y_i}{\boldsymbol{f}(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}})} - \frac{(1-y_i)}{\boldsymbol{1}-\boldsymbol{f}(\vec{\boldsymbol{w}}\cdot\vec{\boldsymbol{x}})}\}$$

$$= w_{present} + \eta \nabla_w \vec{\boldsymbol{x}}[\boldsymbol{y_i}\{\boldsymbol{1} - \boldsymbol{f}(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}})\} - \boldsymbol{f}(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}})(1-y_i)\}]$$

$$= w_{present} + \eta \nabla_w \vec{\boldsymbol{x}} \{y_i - \boldsymbol{f}(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}})\}$$

2) Activation Function Improvement; Motivation for using ReLU

Initialize $\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}$, $\mathbf{W}^{(3)}$ properly

Until (No change):

Extracting n' samples from **X** to form **X'**

$\Delta\mathbf{W}^{(1)} = 0$, $\Delta\mathbf{W}^{(2)} = 0$, $\Delta\mathbf{W}^{(3)} = 0$

for each comp of **X':**

$\mathbf{z}=f(\mathbf{W}^{(1)}\mathbf{x})$

$\mathbf{u}=f(\mathbf{W}^{(2)}\mathbf{z})$

$\mathbf{yh}=f(\mathbf{W}^{(3)}\mathbf{u})$

$\boldsymbol{\delta} = (\mathbf{yh}-\mathbf{y}) \times f'(\mathbf{W}^{(3)}\mathbf{u})$ #c×1

$\boldsymbol{\gamma} = \boldsymbol{\delta}\bullet\widetilde{W}^{(3)} \times f'(\mathbf{W}^{(2)}\mathbf{z})$ #q×1

$\boldsymbol{\beta} = \boldsymbol{\gamma}\bullet\widetilde{W}^{(2)} \times f'(\mathbf{W}^{(1)}\mathbf{z})$ #p×1

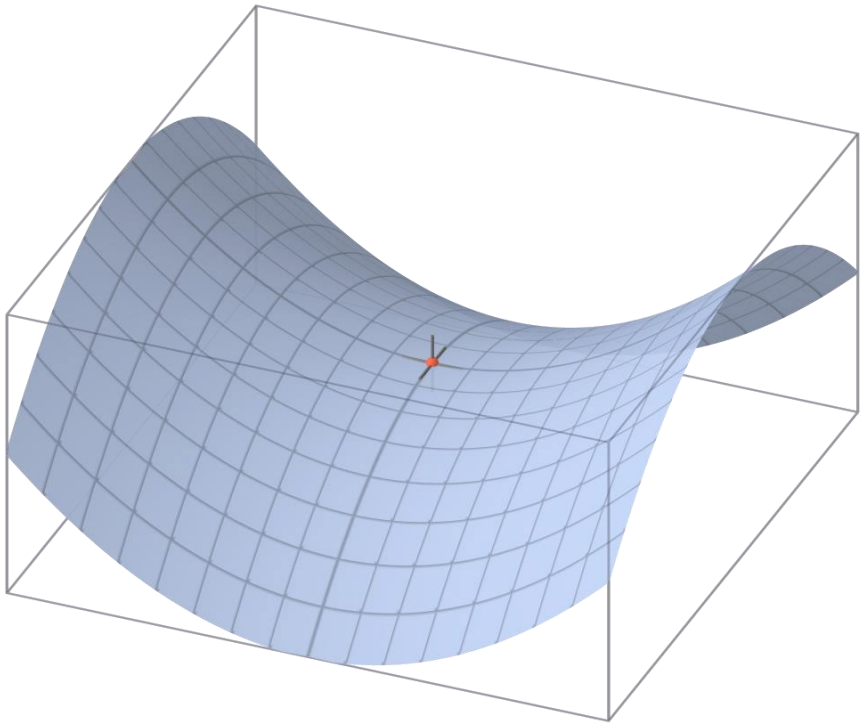
$\mathbf{W}^{(3)} = \mathbf{W}^{(3)} - (1/n')\eta\boldsymbol{\delta}\mathbf{u}^{T}$

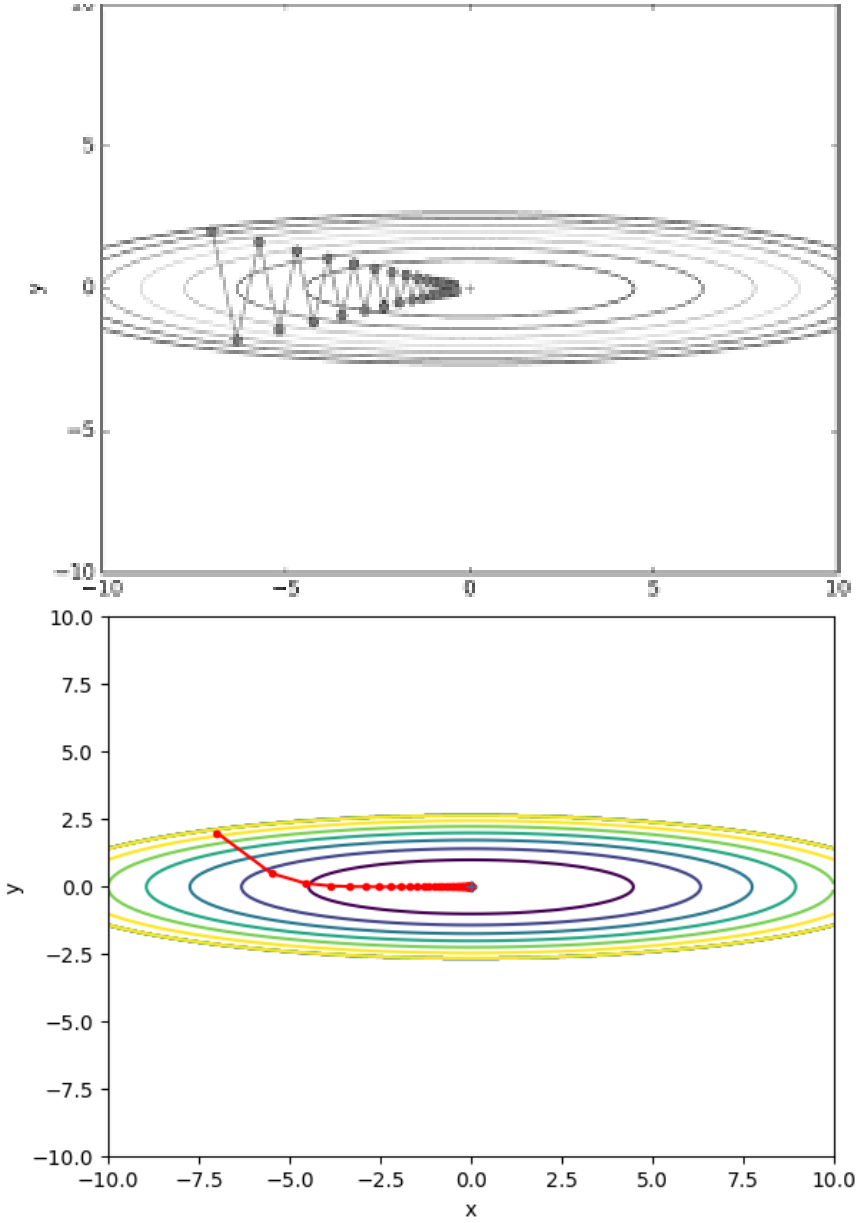
$\mathbf{W}^{(2)} = \mathbf{W}^{(2)} - (1/n')\eta\boldsymbol{\gamma}\mathbf{z}^{T}$

$\mathbf{W}^{(1)} = \mathbf{W}^{(1)} - (1/n')\eta\boldsymbol{\beta}\mathbf{x}^{T}$

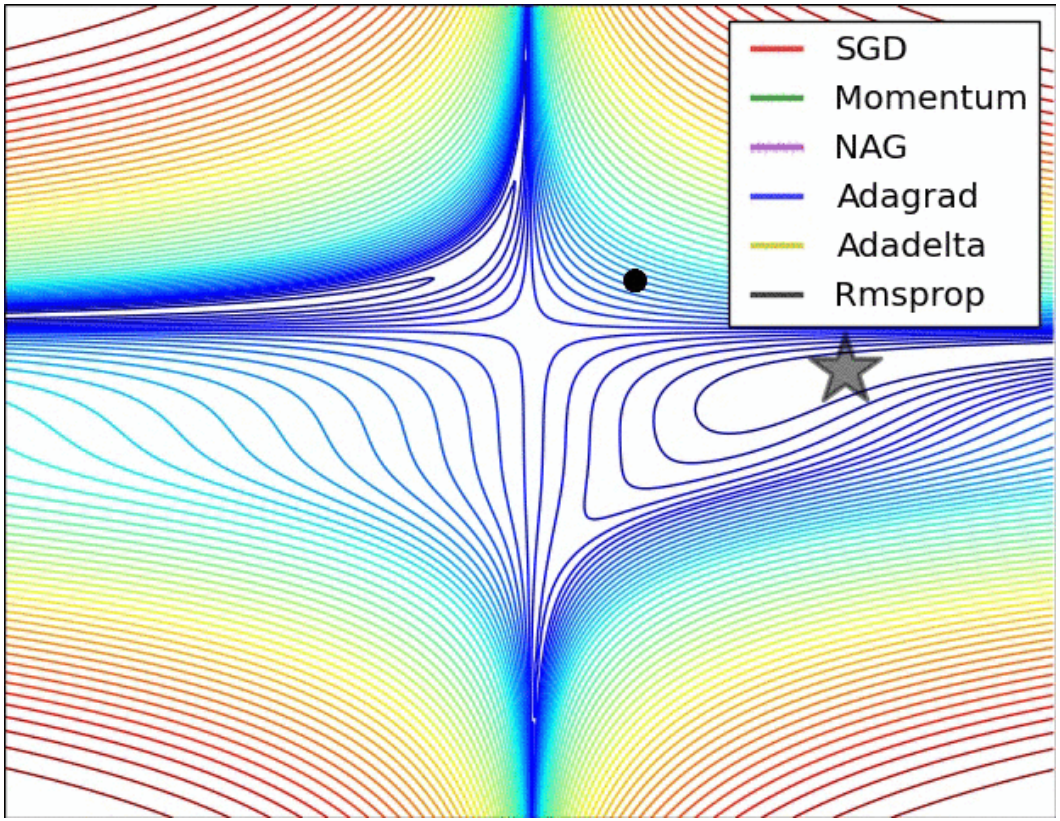
You can expect the vanishing gradient problem

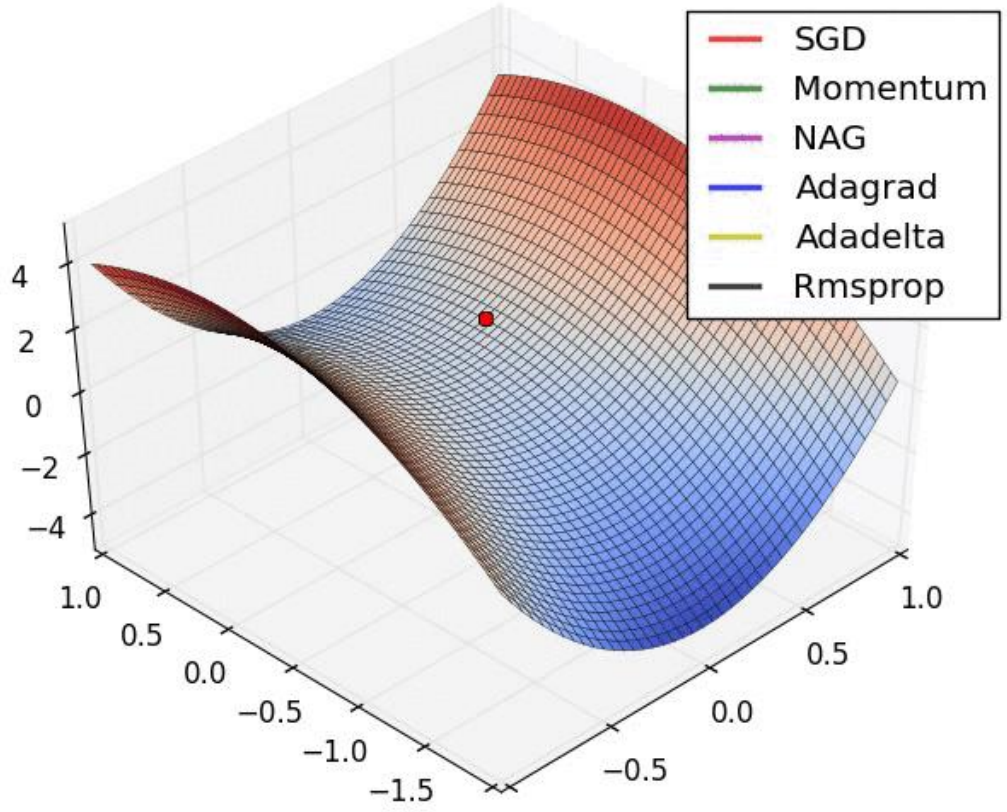
3) Optimizer Improvement





Optimizer Comparison

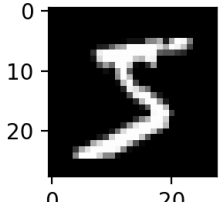


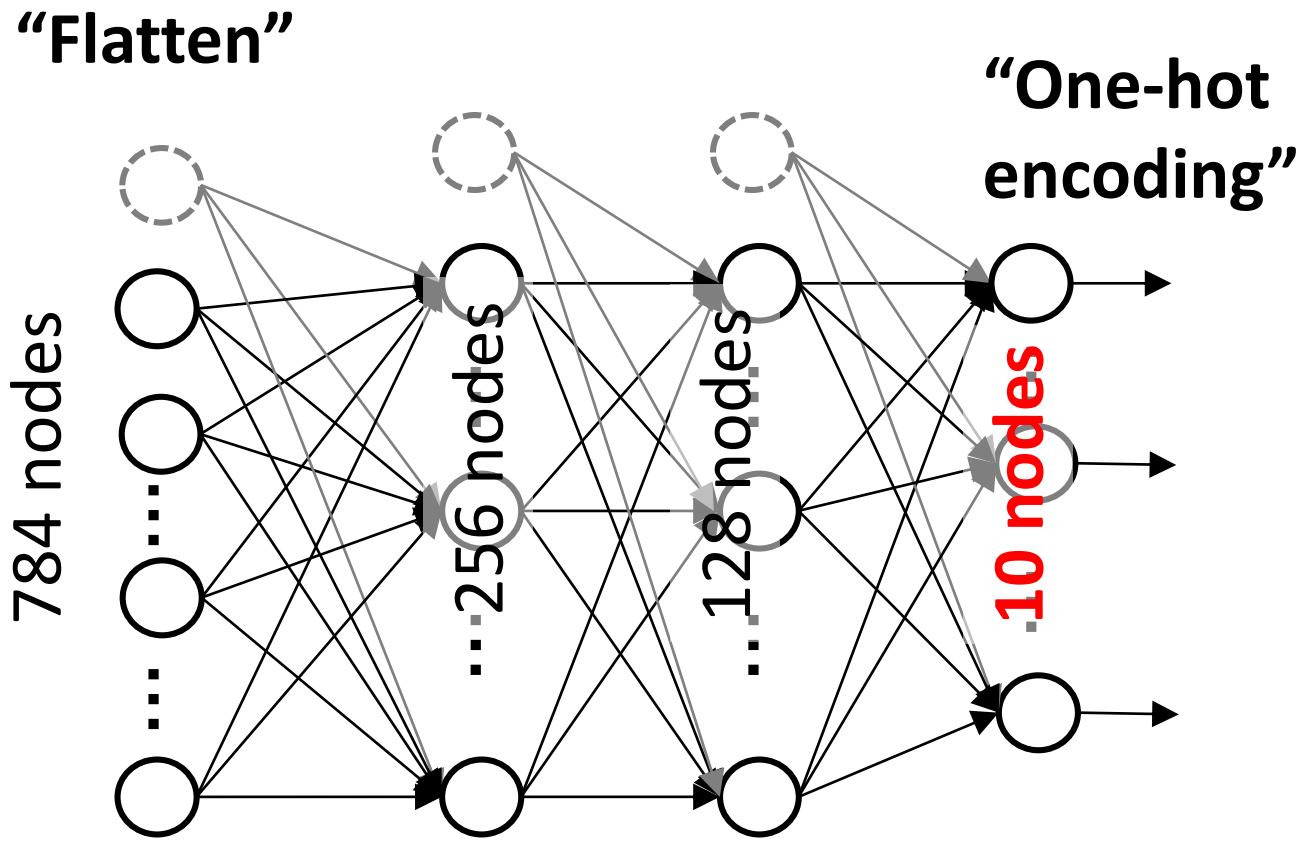


- Refer Schaul, T., Antonoglou, I., & Silver, D. (2013). “Unit tests for stochastic optimization,” *arXiv preprint arXiv:1312.6055*.

Okay Let's Start!
Target Deep MLP

28 x 28

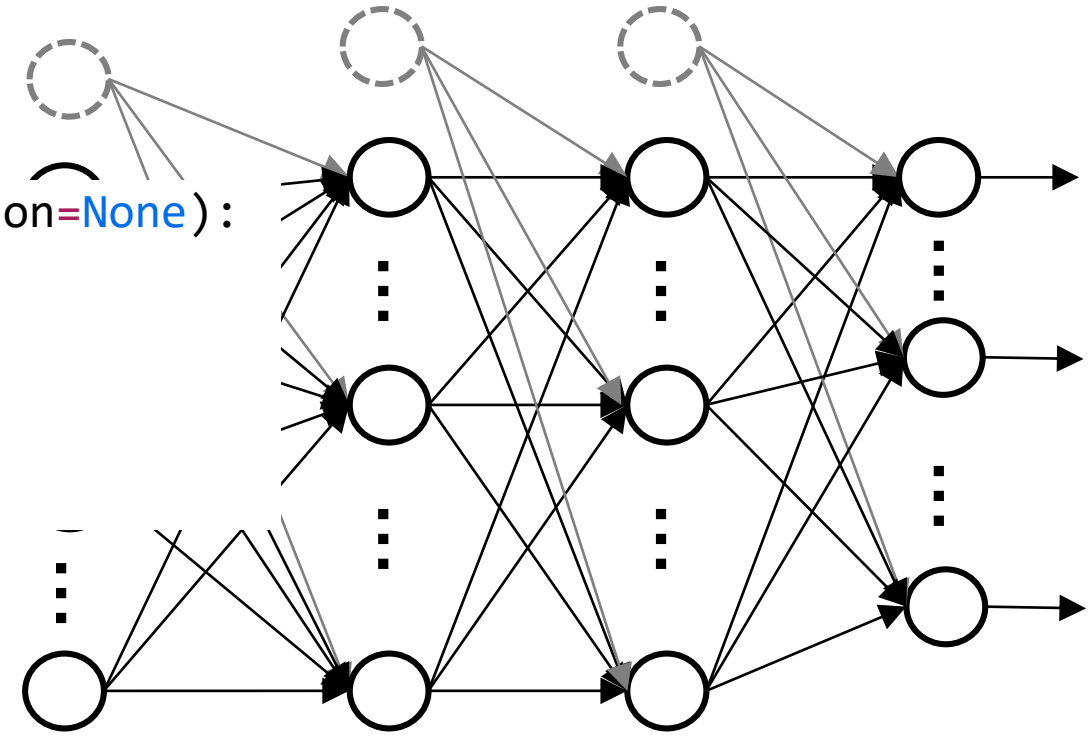




What Would You Do as the First Step?

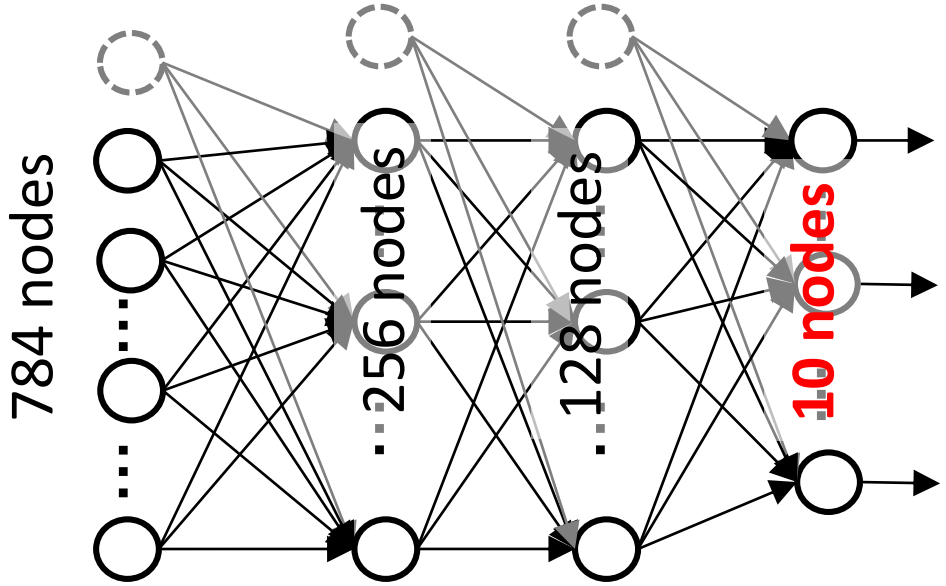
- Defining a function for one neuron layer

```
def neuron_layer(X, W, b, activation=None):
    z = tf.matmul(X, W)+b
    if activation is None:
        return z
    else:
        return activation(z)
```



Forming Target MLP Network as Python Function

```
def my_dMLP(X_flatten, W, b):
    hidden1 = neuron_layer(X_flatten,W[0], b[0], activation=tf.nn.sigmoid)
    hidden2 = neuron_layer(hidden1, W[1], b[1], activation=tf.nn.sigmoid)
    logits = neuron_layer(hidden2, W[2], b[2], activation=None)
    y_pred = tf.nn.softmax(logits)
return y_pred
```



Preparing Dataset

```
(X_train, y_train), (X_test, y_test) =
tf.keras.datasets.mnist.load_data()
nTrain = X_train.shape[0]

X_train_std, X_test_std = X_train/255.0, X_test/255.0
X_train_std = X_train_std.astype("float32")

# convert class vectors to binary class matrices
y_train_onehot = tf.keras.utils.to_categorical(y_train, 10)
y_test_onehot = tf.keras.utils.to_categorical(y_test, 10)
```

Now, We have to Train the Model

- Batch vs. Stochastic vs. **Mini batch**

$$W_{next} = W_{present} + \Delta W$$

- We have 60,000 data for training.
- If batch size is set as 200,
  - We calculate ΔW based on 200 randomly picked datasets, X'
  - 300 updates for W for whole dataset ➔ 1 **epoch** is completed
- If we set n_epoch is 40, we repeat the above training procedure for whole dataset 40 times.

Useful Tools; Optimizer

- **tf.keras.optimizers.Optimizer**
- https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Optimizer
- Method : apply_gradients

Apply gradients to variables.

This is the second part of `minimize()`. It returns an `Operation` that applies gradients.

The method sums gradients from all replicas in the presence of `tf.distribute.Strategy` by default. You can aggregate gradients yourself by passing `experimental_aggregate_gradients=False`.

Example:

```
grads = tape.gradient(loss, vars)
grads = tf.distribute.get_replica_context().all_reduce('sum', grads)
# Processing aggregated gradients.
optimizer.apply_gradients(zip(grads, vars),
    experimental_aggregate_gradients=False)
```

Use of Optimizer for Minimizing Gradients

```
opt = tf.keras.optimizers.SGD(learning_rate=0.01)

...
# tape is somewhere for realtioing [W,b] vs. loss function
...

gradients = tape.gradient(loss, [W, b])
opt.apply_gradients(zip(gradients, [W, b]))
```

```
In [8]: a = ((1,2,3), (4,5,6))
   ...: h = ('a','b')

In [9]: zip(a, h)
Out[9]: <zip at 0x20dd4e99500>

In [10]: for x, y in zip(a, h):
    ...:     print(x, y)
    ...:
(1, 2, 3) a
(4, 5, 6) b
```

Training Codes

```
n_inputs = np.array([28*28, 256, 128])
n_nodes = np.array([256, 128, 10])
n_layer = 3
W, b = {}, {}
for layer in range(n_layer):
    stddev = 2 / np.sqrt(n_inputs[layer] + n_nodes[layer])
    W_init = tf.random.truncated_normal((n_inputs[layer], n_nodes[layer]), stddev = stddev)
    W[layer] = tf.Variable(W_init)
    b[layer] = tf.Variable(tf.zeros([n_nodes[layer]]))

n_epoch = 40
batchSize = 200
nBatch = int(nTrain/batchSize)
opt = tf.keras.optimizers.SGD(learning_rate=0.01)

for epoch in range(n_epoch):
    idxShuffle = np.random.permutation(X_train.shape[0])
    for idxSet in range(nBatch):
        X_batch = X_train_std[idxShuffle[idxSet*batchSize:(idxSet+1)*batchSize], :]
        X_batch_tensor = tf.convert_to_tensor(X_batch.reshape(-1, 28*28))
        y_batch = y_train_onehot[idxShuffle[idxSet*batchSize:(idxSet+1)*batchSize], :]
        with tf.GradientTape() as tape:
            y_pred = my_dMLP(X_batch_tensor, W, b)
            loss = tf.reduce_mean(tf.keras.losses.MSE(y_batch, y_pred))
        gradients = tape.gradient(loss, [W[2], W[1], W[0], b[2], b[1], b[0]])
        opt.apply_gradients(zip(gradients, [W[2], W[1], W[0], b[2], b[1], b[0]]))
```

Visualization

```
if epoch % 5 ==0:
    correct  = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y_batch, 1))
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32)).numpy()
    print(accuracy)
```

You Can Apply Change for Improvements

- Change the optimizer into “Adam”
- Change the activation function into “ReLU”
- Change the cost function “categorical_crossentropy”

Simpler Way

```
import tensorflow as tf
import numpy as np

layers = [tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dense(128, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)]

myMLP = tf.keras.Sequential(layers)
myMLP.compile(optimizer="Adam", loss="CategoricalCrossentropy",
metrics=["accuracy"])

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
nTrain = X_train.shape[0] # 60000
X_train_std, X_test_std = X_train/255.0, X_test/255.0
y_train_onehot = tf.keras.utils.to_categorical(y_train, 10)
y_test_onehot = tf.keras.utils.to_categorical(y_test, 10)

myMLP.fit(X_train_std, y_train_onehot, epochs=40, batch_size = 200)
```