# Introduction to Deep Neural Networks (Spring 2021)

# Homework #2 (50 Pts, Due Date: April 18)

**Student ID 2019311195**

**Name 김지유**

**Instruction:** We provide all codes and datasets in Python. Please write your code to complete activation layers (sigmoid, ReLU, tanh), fully-connected layer (FCLayer), softmax layer, and L2 regularization. Submit two files as follows:

- 'DNN_HW2_YourName_ STUDENTID.zip': ./Answer.py and your document
- 'DNN_HW2_YourName_ STUDENTID.pdf': Your document converted into pdf.

**NOTE 1**: **You should write your code in 'EDIT HERE.'** It is not recommended to edit other parts. Once you complete your implementation, run and check your code ('main.py').

**TIP 1.** Please carefully look at the instructions and the template code, especially the structure of 'ClassifierModel'. L2 regularization is only related to model weights on 'FCLayer'(W only exists in FC Layer) and reg_lambda is an L2 reg. hyperparameter.

**TIP 2**. We provide 'test_answer.py' to check your implementation. The output might be different depending on your environment (OS or python version) or implementation details. However, if you found too different results, your implementation may be incorrect (environment: Windows 10, Python 3.7.4).

**(1)** **[30 pts]** Implement functions in ReLU, Sigmoid, Tanh, FCLayer, SoftmaxLayer, L2Regularization in 'Answer.py'.

**(a)** **[Activation Layer]** Implement sigmoid, ReLU, tanh activation functions in 'Answer.py' ('Sigmoid', 'ReLU', 'Tanh').

**(b)** **[Fully Connected Layer]** Implement a fully-connected layer in 'Answer.py' ('FCLayer').

**(c)** **[Softmax Layer]** Implement the softmax layer in 'Answer.py' ('SoftmaxLayer').

Given a mini-batch data $D(X,Y)$, the error function for a mini-batch is defined as follows:

$$\boldsymbol{E(w) = -\frac{1}{n}\sum_{(x_i,y_i)\in D} y_i * \log(\hat{y}_i) + \frac{1}{2}\lambda\sum_{j=1}^{k}\left\|w^{(j)}\right\|_2^2}$$

$$= (\textit{cross entropy Loss}) + \frac{1}{2}\lambda * (\textit{L2 Regularization term})$$

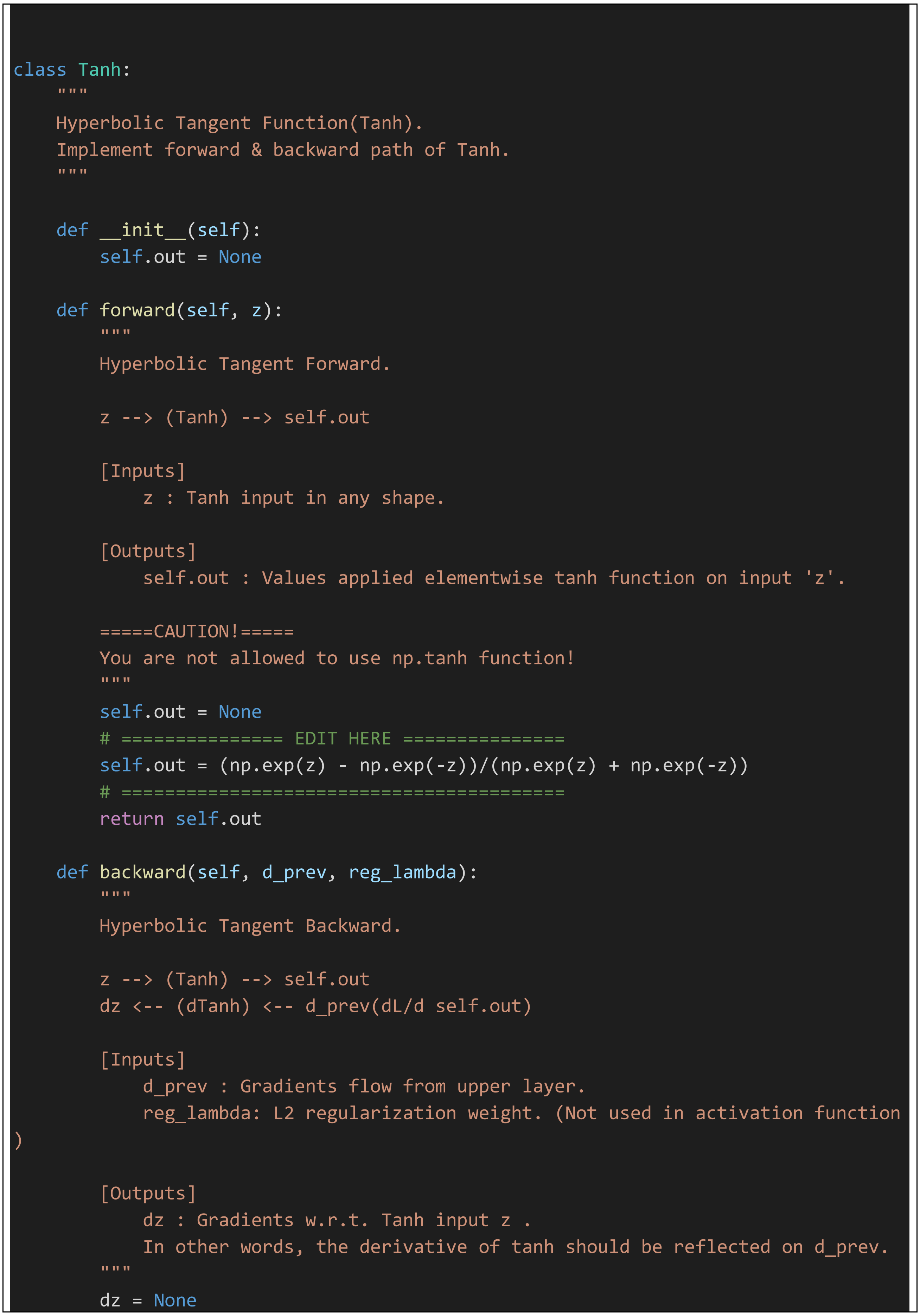
$$\textit{where } \hat{y}_i = softmax\left(w^{(k)^T} z_i^{(k)} + b^{(k)}\right),$$

$$z_i^{(j)} = \sigma\left(w^{(j)} z_i^{(j-1)} + b^{(j)}\right),\ z_i^{(0)} = x_i \quad (j = 1, 2, \dots, k)$$

$$w^{(j)}, b^{(j)} \textit{ is the parameters of } j - th \textit{ layer and } \sigma \textit{ is the activation function.}$$

**Answer: Fill your code here. You also have to submit your code to i-campus.**

```python
from collections import OrderedDict
import numpy as np

def softmax(z):
    # We provide numerically stable softmax.
    z = z - np.max(z, axis=1, keepdims=True)
    _exp = np.exp(z)

    _sum = np.sum(_exp, axis=1, keepdims=True)
    sm = _exp / _sum

    return sm


class ReLU:
    """
    ReLU Function. ReLU(x) = max(0, x)
    Implement forward & backward path of ReLU.

    ReLU(x) = x if x > 0.
              0 otherwise.
    Be careful. It's '>', not '>='.
    """

    def __init__(self):
        # 1 (True) if ReLU input <= 0
        self.zero_mask = None
        self.out = None

    def forward(self, z):
        """
        ReLU Forward.
        ReLU(x) = max(0, x)

        z --> (ReLU) --> out

        [Inputs]
            z : ReLU input in any shape.
        [Outputs]
            self.out : Values applied elementwise ReLU function on input 'z'.
        """
        self.out = None
        # ================================ EDIT HERE ===============================
=
        self.out = np.where(z>0, z, 0)
        # ==========================================================================
=
        return self.out

    def backward(self, d_prev, reg_lambda):
        """
        ReLU Backward.

        z --> (ReLU) --> out
        dz <-- (dReLU) <-- d_prev(dL/dout)

        [Inputs]
            d_prev : Gradients flow from upper layer.
                - d_prev = dL/dk, where k = ReLU(z).
            reg_lambda: L2 regularization weight. (Not used in activation function
)
        [Outputs]
            dz : Gradients w.r.t. ReLU input z.
        """
        dz = None
        # ================================ EDIT HERE ===============================
=
        dz = d_prev * np.where(self.out>0, 1, 0)
        # ==========================================================================
=
        return dz

    def update(self, learning_rate):
        # NOT USED IN ReLU
        pass

    def summary(self):
        return 'ReLU Activation'


class Sigmoid:
    """
    Sigmoid Function.
    Implement forward & backward path of Sigmoid.
    """
    def __init__(self):
        self.out = None

    def forward(self, z):
        """
        Sigmoid Forward.

        z --> (Sigmoid) --> self.out

        [Inputs]
            z : Sigmoid input in any shape.

        [Outputs]
            self.out : Values applied elementwise sigmoid function on input 'z'.
        """
        self.out = None
        # =============== EDIT HERE ===============
        self.out = 1/(1+np.exp(-z))
        # =========================================
        return self.out

    def backward(self, d_prev, reg_lambda):
        """
        Sigmoid Backward.

        z --> (Sigmoid) --> self.out
        dz <-- (dSigmoid) <-- d_prev(dL/d self.out)

        [Inputs]
            d_prev : Gradients flow from upper layer.
            reg_lambda: L2 regularization weight. (Not used in activation function
)

        [Outputs]
            dz : Gradients w.r.t. Sigmoid input z .
        """
        dz = None
        # =============== EDIT HERE ===============
        dz = d_prev * (self.out * (1 - self.out))
        # =========================================
        return dz

    def update(self, learning_rate):
        # NOT USED IN Sigmoid
        pass

    def summary(self):
        return 'Sigmoid Activation'
```



```
class Tanh:
    """
    Hyperbolic Tangent Function(Tanh).
    Implement forward & backward path of Tanh.
    """

    def __init__(self):
        self.out = None

    def forward(self, z):
        """
        Hyperbolic Tangent Forward.

        z --> (Tanh) --> self.out

        [Inputs]
            z : Tanh input in any shape.

        [Outputs]
            self.out : Values applied elementwise tanh function on input 'z'.

        =====CAUTION!=====
        You are not allowed to use np.tanh function!
        """
        self.out = None
        # =============== EDIT HERE ===============
        self.out = (np.exp(z) - np.exp(-z))/(np.exp(z) + np.exp(-z))
        # =========================================
        return self.out

    def backward(self, d_prev, reg_lambda):
        """
        Hyperbolic Tangent Backward.

        z --> (Tanh) --> self.out
        dz <-- (dTanh) <-- d_prev(dL/d self.out)

        [Inputs]
            d_prev : Gradients flow from upper layer.
            reg_lambda: L2 regularization weight. (Not used in activation function
)

        [Outputs]
            dz : Gradients w.r.t. Tanh input z .
            In other words, the derivative of tanh should be reflected on d_prev.
        """
        dz = None
        # =============== EDIT HERE ===============
        dz = d_prev * (1-np.square(self.out))
        # =========================================
        return dz

    def update(self, learning_rate):
        # NOT USED IN Tanh
        pass

    def summary(self):
        return 'Tanh Activation'


"""
    ** Fully-Connected Layer **
    Single Fully-Connected Layer.

    Given input features,
    FC layer linearly transforms the input with weights (self.W) & bias (self.b).

    You need to implement forward and backward pass.
"""

class FCLayer:
    def __init__(self, input_dim, output_dim):
        # Weight Initialization
        self.W = np.random.randn(input_dim, output_dim) / np.sqrt(input_dim / 2)
        self.b = np.zeros(output_dim)

    def forward(self, x):
        """
        FC Layer Forward.
        Use variables : self.x, self.W(reg_lambda), self.b

        [Input]
        x: Input features.
        - Shape : (batch size, In Channel, Height, Width)
        or
        - Shape : (batch size, input_dim)

        [Output]
        self.out : fc result
        - Shape : (batch size, output_dim)

        Tip : you do 'not' need to implement L2 regularization here. It is already
 implemented in ClassifierModel.forward()
        """
        # Flatten input if needed.
        if len(x.shape) > 2:
            batch_size = x.shape[0]
            x = x.reshape(batch_size, -1)

        self.x = x
        self.out = None
        # =============================== EDIT HERE ==============================
=
        self.out = np.dot(self.x, self.W) +self.b
        # ========================================================================
=
        return self.out

    def backward(self, d_prev, reg_lambda):
        """
        FC Layer Backward.
        Use variables : self.x, self.W

        [Input]
        d_prev: Gradients value so far in back-propagation process.
        reg_lambda: L2 regularization weight. (Not used in activation function)

        [Output]
        dx : Gradients w.r.t input x
        - Shape : (batch_size, input_dim) - same shape as input x

        Tip : you should implement backpropagation of L2 regularization here.
        """
        dx = None           # Gradient w.r.t. input x
        self.dW = None      # Gradient w.r.t. weight (self.W)
        self.db = None      # Gradient w.r.t. bias (self.b)
        # =============================== EDIT HERE ==============================
=
        dx = np.dot(d_prev, self.W.T) #self.W = (5,7), d_prev.shape = (5,7), self.
X.shpae = (batch_size, input)
        self.dW = np.dot(self.x.T, d_prev) + reg_lambda * self.W
        self.db = np.sum(d_prev, axis=0)
        # ========================================================================
=
        return dx

    def update(self, learning_rate):
        self.W -= self.dW * learning_rate
        self.b -= self.db * learning_rate

    def summary(self):
        return 'Input -> Hidden : %d -> %d ' % (self.W.shape[0], self.W.shape[1])

"""
```

```
    ** Softmax Layer **
    Softmax Layer applies softmax (WITHOUT any weights or bias)

    Given an score,
    'SoftmaxLayer' applies softmax to make probability distribution. (Not log soft
max or else...)

    You need to implement forward and backward pass.
    (This is NOT an entire model.)
"""

class SoftmaxLayer:
    def __init__(self):
        # No parameters
        pass

    def forward(self, x):
        """
        Softmax Layer Forward.
        Apply softmax (not log softmax or others...) on axis-1

        Use 'softmax' function above in this file.
        We recommend you see the function.

        [Input]
        x: Score to apply softmax
        - Shape: (batch_size, # of class)

        [Output]
        y_hat: Softmax probability distribution.
        - Shape: (batch_size, # of class)
        """
        self.y_hat = None
        # =============================== EDIT HERE ==============================
=
        self.y_hat = softmax(x)
        # ========================================================================
=
        return self.y_hat

    def backward(self, d_prev=1, reg_lambda=0):
        """
        Softmax Layer Backward.
        Gradients w.r.t input score.

        That is,
        Forward  : softmax prob = softmax(score)
        Backward : dL / dscore => 'dx'
```

```python
        Compute dx (dL / dscore).
        Check loss function in HW3 word file.

        [Input]
        d_prev : Gradients flow from upper layer.

        [Output]
        dx: Gradients of softmax layer input 'x'
        """
        batch_size = self.y.shape[0]
        dx = None
        # ============================== EDIT HERE ==============================
=
        dx = (self.y_hat -self.y)/batch_size
        # ========================================================================
=
        return dx


    def ce_loss(self, y_hat, y):
        """
        Compute Cross-entropy Loss.
        Use epsilon (eps) for numerical stability in log.

        Check loss function in HW2 word file.

        [Input]
        y_hat: Probability after softmax.
        - Shape : (batch_size, # of class)

        y: One-hot true label
        - Shape : (batch_size, # of class)

        [Output]
        self.loss : cross-entropy loss
        - float
        """
        self.loss = None
        eps = 1e-10
        self.y_hat = y_hat
        self.y = y
        # ============================== EDIT HERE ==============================
=
        self.loss = -np.sum(self.y * np.log(self.y_hat+eps)) /self.y_hat.shape[0]
        # ========================================================================
=
        return self.loss
    def update(self, learning_rate):
        # Not used in softmax layer.
        pass

    def summary(self):
        return 'Softmax layer'


"""
    ** Classifier Model **
    This is an class for entire Classifier Model.
    All the functions and variables are already implemented.
    Look at the codes below and see how the codes work.

    <<< DO NOT CHANGE ANYTHING HERE >>>
"""

class ClassifierModel:
    def __init__(self,):
        self.layers = OrderedDict()
        self.softmax_layer = None
        self.loss = None
        self.pred = None

    def predict(self, x):
        # Outputs model softmax score
        for name, layer in self.layers.items():
            x = layer.forward(x)
        x = self.softmax_layer.forward(x)
        return x

    def forward(self, x, y, reg_lambda):
        # Predicts and Compute CE Loss
        reg_loss = 0
        self.pred = self.predict(x)
        ce_loss = self.softmax_layer.ce_loss(self.pred, y)

        for name, layer in self.layers.items():
            if isinstance(layer, FCLayer):
                norm = np.linalg.norm(layer.W, 2)
                reg_loss += 0.5 * reg_lambda *  norm * norm

        self.loss = ce_loss + reg_loss

        return self.loss

    def backward(self, reg_lambda):
        # Back-propagation
        d_prev = 1
        d_prev = self.softmax_layer.backward(d_prev, reg_lambda)
        for name, layer in list(self.layers.items())[::-1]:
            d_prev = layer.backward(d_prev, reg_lambda)

    def update(self, learning_rate):
        # Update weights in every layer with dW, db
        for name, layer in self.layers.items():
            layer.update(learning_rate)

    def add_layer(self, name, layer):
        # Add Neural Net layer with name.
        if isinstance(layer, SoftmaxLayer):
            if self.softmax_layer is None:
                self.softmax_layer = layer
            else:
                raise ValueError('Softmax Layer already exists!')
        else:
            self.layers[name] = layer

    def summary(self):
        # Print model architecture.
        print('======= Model Summary =======')
        for name, layer in self.layers.items():
            print('[%s] ' % name + layer.summary())
        print('[Softmax Layer] ' + self.softmax_layer.summary())
        print()
```

**(2) [20 Pts]** Experiment results

**(a) [DNN with different activation layer]** Report test accuracy on MNIST using three different activation functions (sigmoid, ReLU, and tanh) with a given DNN architecture and parameters. Explain the differences among three activation functions based on the result (use only one activation function in one experiment among sigmoid, ReLU, and tanh). Tip: It took about 5 seconds per epoch on i7-9700k CPU

**[DNN Architecture]**

| Layer name | Configuration |
|---|---|
| **FC – 1** | Input dim = 784, Output dim = 500 |
| **[Sigmoid, ReLU, Tanh]** | - |
| **FC – 2** | Input dim = 500, Output dim = 500 |

| | |
|---|---|
| **[Sigmoid, ReLU, Tanh]** | - |
| **FC – 3** | Input dim = 500, Output dim = 10 |
| **Softmax Layer** | - |

Hyperparameter settings : num_epochs = 100, learning_rate = 0.001, batch_size=128, reg_lambda($\boldsymbol{\lambda}$) = 1e-8

**Answer: Fill the blank in the table. Explain the differences among the three activation functions.**

| | **Sigmoid** | **ReLU** | **Tanh** |
|---|---|---|---|
| **Test accuracy** | **0.60** | **0.87** | **0.89** |

Test accuracy: Sigmoid(0.60) < ReLU(0.87) < Tanh(0.89)

Sigmoid:

Sigmoid function은 tanh function과 함께 traditional non-linear activation function이다. output의 범위는 0~1이고, 1차 미분 함수의 결과의 범위는 0~0.25 이다. 1차 미분 함수의 결과 중 가장 큰 값이 0.25밖에 되지 않고, input 값이 너무 작거나 클 경우 1차 미분 함수의 결과는 0에 가까운 값이 되어버리는 특징이 있다. 이러한 특징 때문에 hidden layer의 개수가 많은 심층 신경망에서 sigmoid 함수를 activation function으로 사용할 경우, backpropagation 과정에서 0~0.25 사이의 작은 값이 계속 곱해져 결국 0에 수렴하는 vanishing gradient problem이 발생하여 학습이 안 될 가능성이 높다. 이와 같은 사실을 이번 실험 결과에서도 확인할 수 있었다.

ReLU:

ReLU function은 modern non-linear activation function 중 하나이다. input이 음수일 경우 output은 0이고, input이 양수일 경우 output은 input과 같다. 1차 미분 함수의 결과는 input이 음수일 경우 0, input이 양수일 경우 1이다. Sigmoid function, tanh function과 비교했을 때 1차 미분 함수의 결과가 크기 때문에 layer의 개수가 많아져도 vanishing gradient problem이 발생할 확률이 적다. 또한 input이 양수인지 음수인지에 따라 gradient가 0 아니면 1이기 때문에 sigmoid function, tanh function보다 gradient가 계산되는 속도가 빠르다. 하지만 input이 음수인 경우 output이 항상 0이기 때문에 만약 forwardpropagation 과정에서 input으로 음수가 들어가면 backwordpropagation에서 0이 곱해지면서 아예 죽어버려 학습이 안 되는 dying relu problem이 발생할 수 있다.

Tanh:

Tanh function은 sigmoid function을 변형한 것으로 sigmoid function과 함께 traditional non-linear function 중 하나이다. output의 범위는 -1~1이고, 1차 미분 함수의 결과의 범위는 0~0.5이다. Sigmoid function과 비교했을 때 음수인 output이 존재하여 은닉층의 뉴런들에 대한 편향이 적고, 1차 미분 함수의 결과가 가질 수 있는 최대값이 크기 때문에 vanishing gradient problem이 발생할 가능성이 낮기 때문에 sigmoid function보다는 tanh function이 activation function으로 더 났다. 이러한 사실을 이번 실험 결과에서도 sigmoid function을 activation function으로 사용했을 때보다 tanh function을 activation function으로 사용했을 때 테스트 정확도가 높다는 것으로써 확인할 수 있다. 하지만 tanh function도 input이 너무 작거나 클 경우 1차 미분 함수의 결과는 0에 가까운 값이 되어버리는 특징을 그대로 가지고 있기 때문에 vanishing gradient problem에서 자유로울 수 없으므로 hidden layer의 개수가 많은 심층신경망에서 tanh function을 activation function으로 사용할 경우, 학습이 잘 안 될 가능성이 높다. 이번 실험에서는 hidden layer의 개수가 많지 않기 때문에 tanh function을 activation function으로 사용해도 vanishing gradient problem이 발생하지 않아 상대적으로 높은 test accuracy가 나온 것 같다.

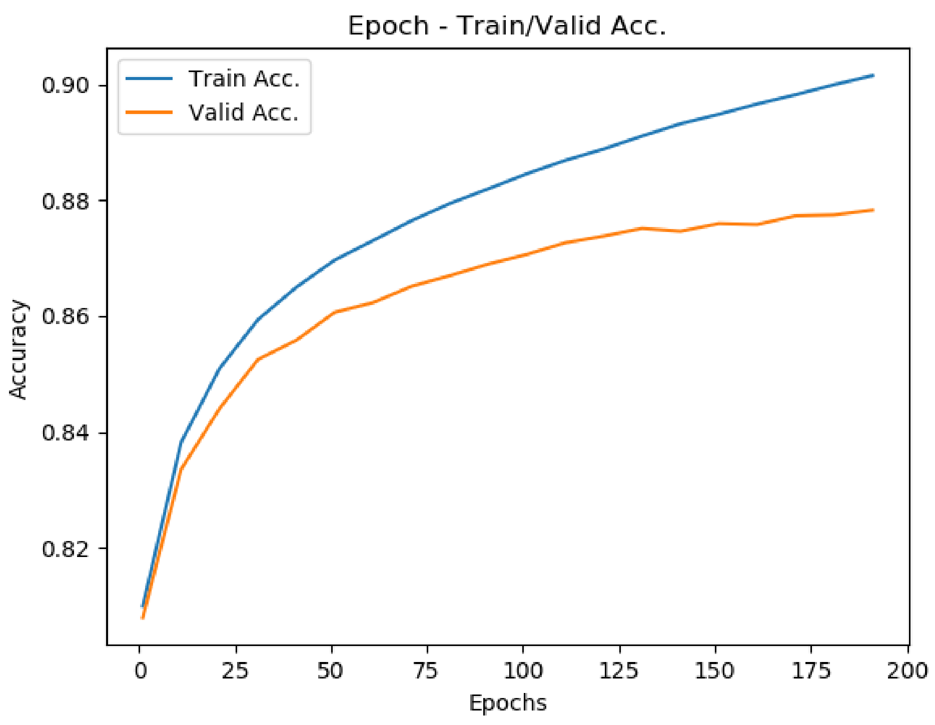
**(b)** **[Deep Neural Networks]** optimize your model architecture (# of hidden layers, # of hidden nodes, # of epochs, learning rate etc.) to achieve the best results on FashionMNIST using 'main.py'. Report your best test accuracy with your fine-tuned hyperparameters. Show the plot of training and validation accuracy every epoch on each case. Also, explain how you design the model structure or parameters in 4~5 lines. (batch size = 128)

**Answer: Fill the blank in the table. Show the plot of training & validation accuracy through epochs.**

| | **Model structure** | **# of epochs** | **Learning rate** | **L2 reg. lambda** | **Best Validation Acc.** | **Final Test Acc.** |
|---|---|---|---|---|---|---|
| **1st Best** | FC-1(784, 514)<br>ReLU1<br>FC-2(514, 256)<br>ReLU2<br>FC-3(256, 128)<br>ReLU3<br>FC-4(128, 64)<br>ReLU4 | 200 | 0.001 | 0.001 | 0.88 | 0.87 |

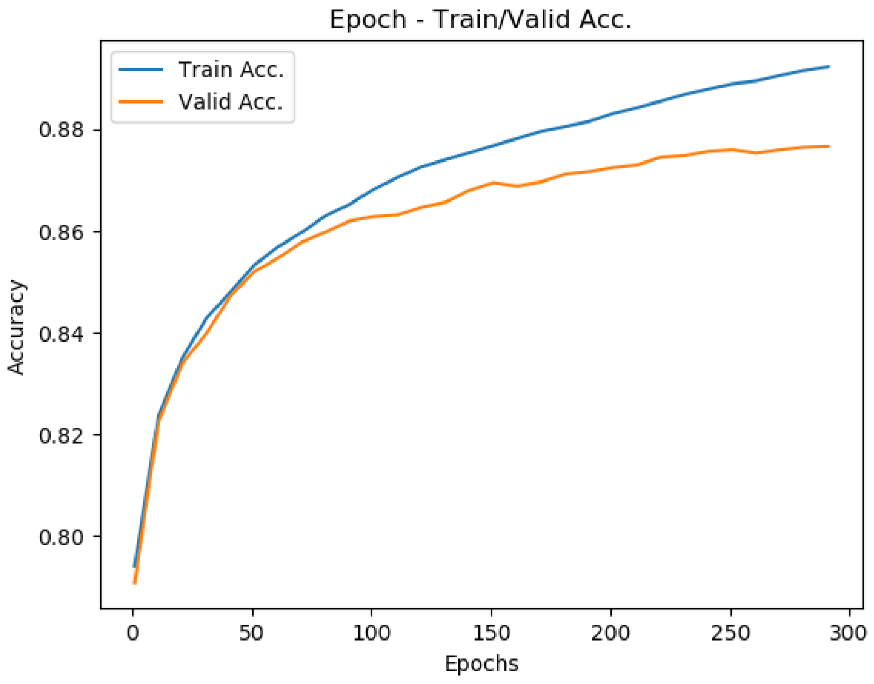
| | | | | | | |
|---|---|---|---|---|---|---|
| | FC-5(64, 10)<br>Softmax | | | | | |
| **2nd Best** | FC-1(784, 500)<br>ReLU1<br>FC-2(500, 500)<br>ReLU2<br>FC-3(500, 10)<br>Softmax | 300 | 0.001 | 0.0001 | 0.88 | 0.87 |
| **3rd Best** | FC-1(784, 500)<br>Tanh1<br>FC-2(500, 500)<br>RelU2<br>FC-3(500, 10)<br>Softmax | 200 | 0.001 | 0.001 | 0.87 | 0.86 |

**1st Best**



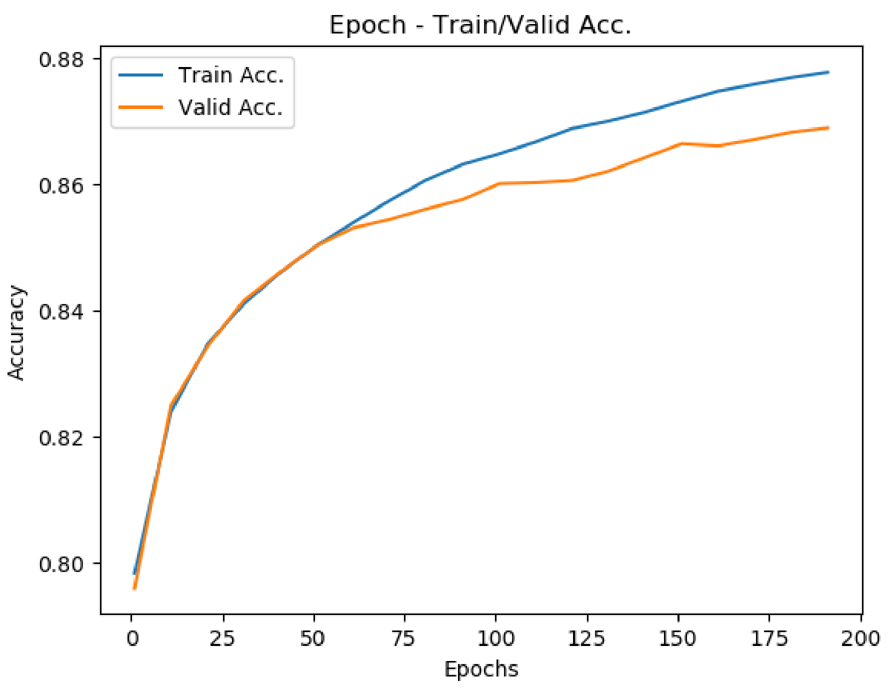
시도해보았던 모델들 중에서 성능이 가장 좋은 모델의 architecture는 FC-1(784, 514), ReLU1, FC-2(514, 256), ReLU2, FC-3(256, 128), ReLU3, FC-4(128, 64), ReLU4, FC-5(64, 10), Softmax이다. 또한 hyperparameter는 epoch의 개수가 200, learning rate가 0.001, L2 regularization lambda가 0.001, batch size가 128이다. 이전에 시도해본 모델들보다 성능을 높이기 위해 hidden layer의 개수를 늘리고, 학습 속도가 느려질 것을 대비해 노드의 수도 조절했다. hidden layer의 개수를 늘림으로써 발생할 수 있는 overfitting 문제를 완화하기 위해 이전 모델들에 비해 L2 regularization lambda를 10배 크게 해주었다. Hidden layer의 개수를 늘리니까 train accuracy는 이전 모델에 비해 크게 향상되었지만, validation accuracy, test accuracy는 매우 조금 향상되었다. 이 모델의 batch size를 64로 줄이고 실험을 해보니 test accuracy가 88%가 나왔다.

**2nd Best**



시도해보았던 모델들 중에서 성능이 2번째로 좋았던 모델의 architecture는 FC-1(784, 500), ReLU1, FC-2(500, 500), ReLU2, FC-3(500, 10), Softmax이다. 또한 hyperparameter는 epoch의 개수가 300, learning rate가 0.001, L2 regularization lambda가 0.0001, batch size가 128이다. epoch의 개수를 300으로 해도 train accuracy가 90%를 못 넘는 것을 보고 모델의 hidden layer의 개수를 늘리는 방향으로 실험을 진행할 계획을 세울 수 있었다. hidden layer의 개수가 많지 않아서 그런지 L2 regularization lambda를 크게 해주면 오히려 test accuracy 성능이 1% 떨어지는 모습을 보였다.

**3rd Best**



시도해보았던 모델들 중에서 성능이 3번째로 좋았던 모델의 architecture는 FC-1(784, 500), Tanh1, FC-2(500, 500), RelU2, FC-3(500, 10), Softmax이다. 또한 hyperparameter는 epoch의 개수가 200, learning rate가 0.001, L2 regularization lambda가 0.001, batch size가 128이다. 동일한 hyperparameter로 activation function을 모두 relu function으로 했을 때도 동일한 test accuracy가 나왔다. layer의 개수가 많지 않아서 tanh function을 activation function으로 사용해도 vanishing gradient problem이 발생하지 않는 것 같다.