Homework 1

IDS 576

Name: Isaac Salvador
Email: isalva2@uic.edu
UIN: 6669845132

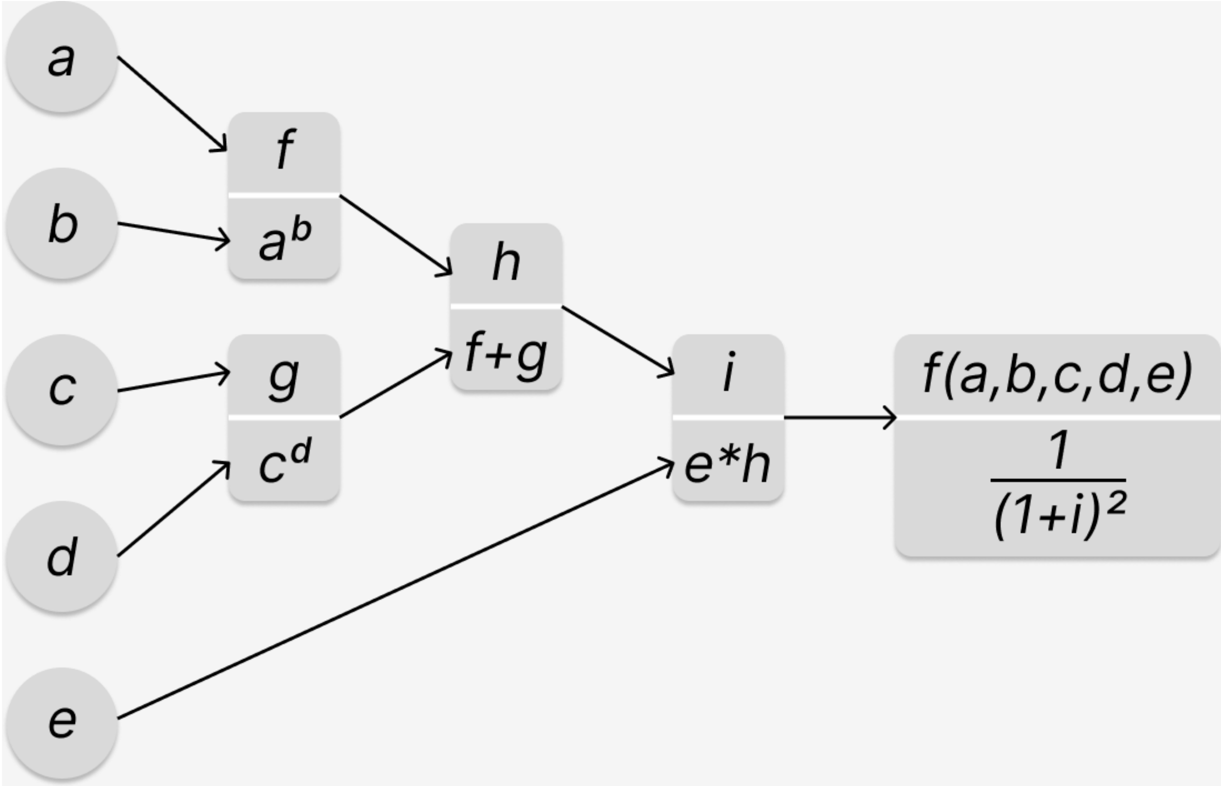
Name: Ahreum Kim
Email: akim239@uic.edu
UIN: 653241895

Name: Sadjad Bazarnovi
Email: sbazar3@uic.edu
UIN: 679314994

1. Backpropogation

Computational Graph

The multivariable equation $f(a,b,c,d,e) = \frac{1}{(1+(a^b+c^d)*e)^2}$ can be expressed with the following computational graph:



This graph begins with inputs a, b, c, d, e and terminates with the function in question.

This graph was created in figma.

Gradient Computation

We wish to compute the gradient of f:

$$\nabla f = \left(\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b}, \frac{\partial f}{\partial c}, \frac{\partial f}{\partial d}, \frac{\partial f}{\partial e} \right)$$

via the **chain rule**. The first step is to compute the partial derivatives of the **compute nodes** with respect to their inputs.

For the last compute node (let us define it as *node* F), we compute the partial derivative $\frac{\partial f}{\partial i}$ w.r.t. i as:

$$\frac{\partial f}{\partial i} = -\frac{2}{(1+i)^3}$$

Implemented in python, we get:

In [ ]:
```
# import math and numpy
import math
import numpy as np
```

```python
# assign input values
a,b,c,d,e = 1,1,1,1,1

# calculate the value of i
i = (a**b + c**d)*e

# compute the partial derivative df/di
def dF_di(i):
    return -2/(1+i)**3
```

We can then obtain the partial derivates for the rest of the **compute nodes** w.r.t. their inputs:

In [ ]:

```python
# di/dh
h = a**b + c**d

def di_dh(e,h):
    return e

# di/de
def di_de(e,h):
    return h

# dh/df
def dh_df():
    return 1

# dh/dg
def dh_dg():
    return 1

# note that partial derivatives w.r.t f & g are linear and equate to 1

# df/da
def df_da(a,b):
    return b*a**(b-1)

# df/db
def df_db(a,b):
    return np.log(a)*a**b

# dg/dc
def dg_dc(c,d):
    return d*c**(d-1)

# dg/dd
def dg_dd(c,d):
    return np.log(c)*c**d
```

The first partial derivative $\frac{\partial f}{\partial a}$ can now be calculated:

$$\frac{\partial f}{\partial a} = \frac{\partial F}{\partial i}\frac{\partial i}{\partial h}\frac{\partial h}{\partial f}\frac{\partial f}{\partial a}$$

In python:

In [ ]:
```
partial_a = dF_di(i)*di_dh(e,h)*dh_df()*df_da(a,b)

print(partial_a)
```

```
-0.07407407407407407
```

and summarily:

In [ ]:
```
# compute partial derivatives w.r.t. b, c, d & e
partial_b = dF_di(i)*di_dh(e,h)*dh_df()*df_db(a,b)

partial_c = dF_di(i)*di_dh(e,h)*dh_dg()*dg_dc(c,b)

partial_d = dF_di(i)*di_dh(e,h)*dh_dg()*dg_dd(c,d)

partial_e = dF_di(i)*di_de(e,h)

# summary
gradient = [partial_a, partial_b, partial_c, partial_d, partial_e]

# chr index for summary
char = 97

print("Summary of Gradient:\n")

for partial in gradient:
    print(f"Partial derivative {chr(char)}: {format(partial, '.6f')}")
    char += 1

print(f"\nGradient of f(1,1,1,1,1) = {sum(gradient)}")
```

```
Summary of Gradient:

Partial derivative a: -0.074074
Partial derivative b: -0.000000
Partial derivative c: -0.074074
Partial derivative d: -0.000000
Partial derivative e: -0.148148

Gradient of f(1,1,1,1,1) = -0.2962962962962963
```

2. Gradient Descent

MSE Function

We wish to define a function in python that calculates *Mean Square Error*, defined by the following function:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

In python:

```
In [ ]: def MSE(true, pred):

            # user input error
            if len(true) != len(pred):
                print("Arrays are not the same size!")

            # obtain sum of squares
            sum_squares = np.square(true-pred)

            # calculate MSE
            mse = np.sum(sum_squares)/len(true)

            return mse
```

Functionality test:

```
In [ ]: # initialize sample data obtained from statology.org
        observed = np.array([34, 37, 44, 47, 48, 48, 46, 43, 32, 27, 26, 24])
        predicted = np.array([37, 40, 46, 44, 46, 50, 45, 44, 34, 30, 22, 23])

        # compute MSE
        print(MSE(observed, predicted))
```

```
5.916666666666667
```

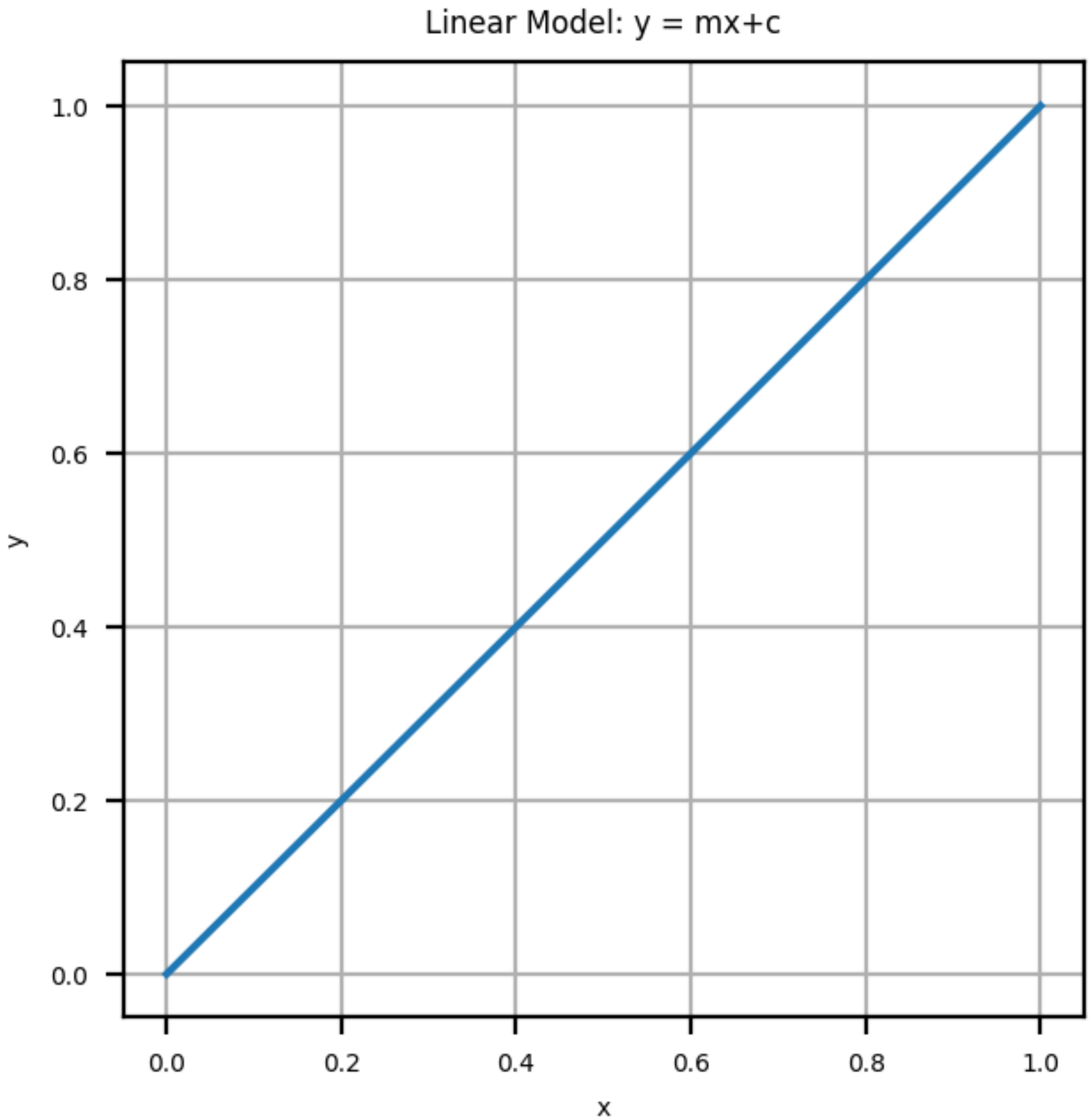
Note: the MSE Function computes the same answer as the generated answer by the statology.org MSE Calculator.

Linear Model

For the linear model $y = mx + c$, where the model parameters $m = 1$, $c = 0$, and $x \in (0, 1)$, the plot looks like this:

```
In [ ]: # import matplotlib
        import matplotlib.pyplot as plt
        plt.rcParams.update({'font.size': 5})

        # initalize model parameters
        m = 1
        c = 0

        # obtain x and y values for the plot
        x = np.linspace(0,1,2)
```

```
y = m*x+c

# plot linear model
fig, ax = plt.subplots(figsize=(3.5,3.5), dpi=200)
ax.plot(x, y)
ax.set_title("Linear Model: y = mx+c")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True)
```

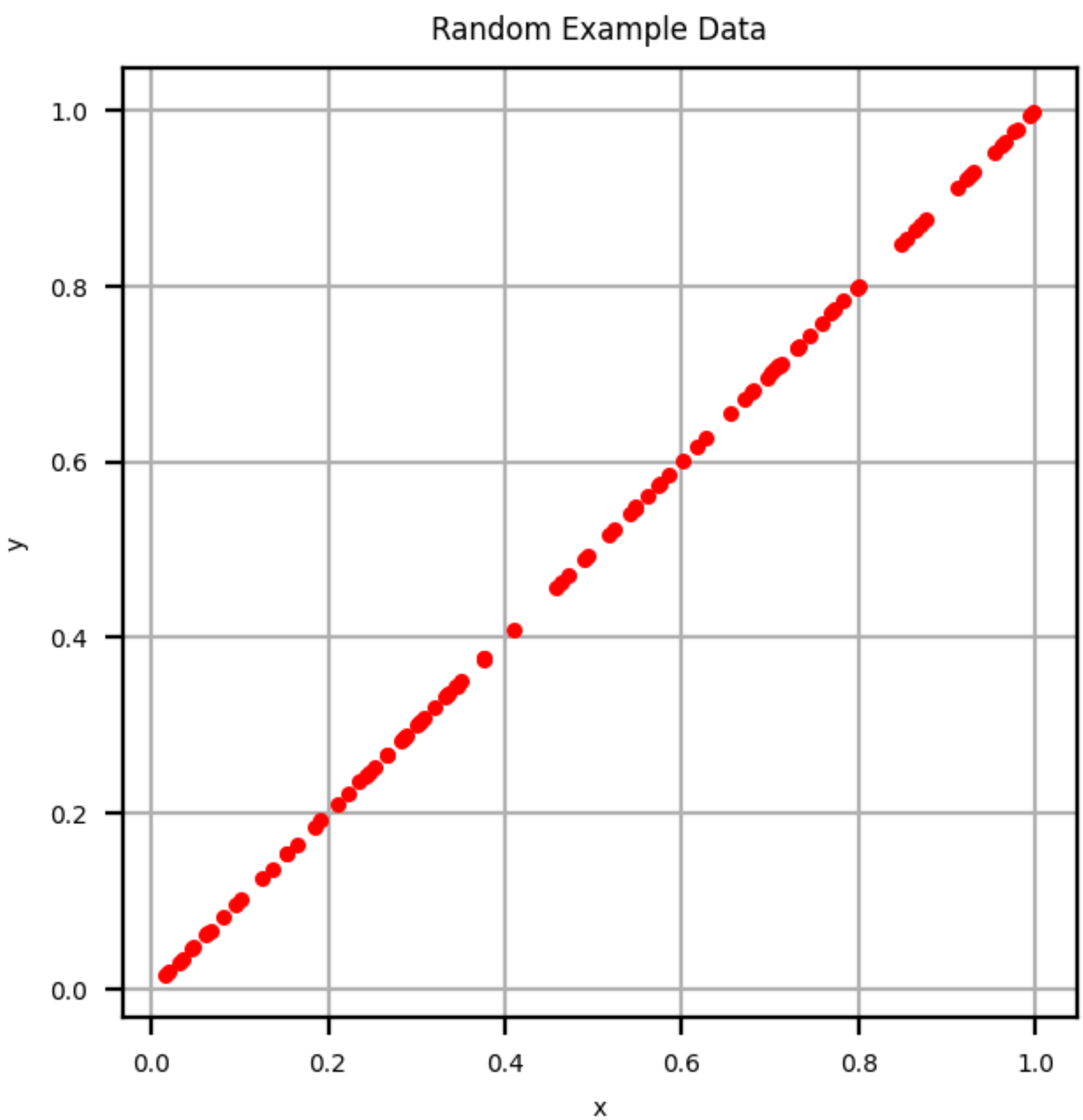


Generate Example Data

We can now generate example data using `numpy`'s `random` module.

In [ ]:

```
# initialize seed for reproducibility
np.random.seed(567)

# create random example data
x_example = np.random.rand(100)
y_example = m*x_example+c

# plot linear model
fig, ax = plt.subplots(figsize=(3.5, 3.5), dpi=200)
ax.scatter(x_example, y_example, zorder=1, c="r", s = 5)
ax.set_title("Random Example Data")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True)
ax.set_axisbelow(1)
```



Linear Gradient Descent

For the linear model with unknown parameters m and c, let us outline the **Gradient Descent** algorithm we shall employ. I implemented this tutorial as a guide.

1. Assign a random value to m and c such that $m \in [0, 1]$ and $c \in [0, 1]$.
2. Choose the learning rate α.
3. Compute the partial derivates of the MSE loss function $\frac{\partial L}{\partial m}$ and $\frac{\partial L}{\partial c}$. More formally:

$$\frac{\partial \mathrm{L}}{\partial m} = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)x_i$$

$$\frac{\partial \mathrm{L}}{\partial c} = -\frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)$$

4. Update the values of m and c in the direction of their respective partial derivative such that

$$m_{n+1} = m_n - \alpha_n \frac{\partial L}{\partial m}$$

$$c_{n+1} = c_n - \alpha_n \frac{\partial L}{\partial c}$$

5. Iterate this process for a set number of epochs or until a desireable loss value is achieved (we will implement the former).

To perform this task lets create a python `class` with the following inputs:

- `np.array` : input data *x* and *y*
- `float` : random initial starting parameters *m* and *c* between [0,1]
- `float` : learning rate *lr*
- `float` : learning rate *alpha*
- `int` : number of *epochs*
- `object` : partial derivatives of *m* and *c*

In [ ]:

```
# create partial derivative function w.r.t. m
def LinearPartialM(x: np.array, y : np.array, yHat: np.array):

    return (-2/len(y))*np.sum((y-yHat)*x)

# create partial derivative functino w.r.t. c
def LinearPartialC(y : np.array, yHat: np.array):
```

```
    return (-2/len(y))*np.sum(y-yHat)


# create Gradient Descent Class
class GradientDescent:
    def __init__(self,
                 x_example: np.array,
                 y_example: np.array,
                 lr: float,
                 epochs: int,
                 dLdm: object,
                 dLdc: object,
                 seed: int):

        # input variables
        self.xTrue = x_example # input x
        self.yTrue = y_example # input y
        self.learningRate = lr # learning rate
        self.numEpochs = epochs # number of epochs
        self.partialM = dLdm # partial w.r.t. m
        self.partialC = dLdc # partial w.r.t. c
        self.seed = seed # random seed for reproducibility

        # variables to be populated after runtime
        self.m = 0
        self.c = 0
        self.loss = 0

        # tracking metrics
        self.lossHistory = []

    # run gradient descent
    def start(self):

        # initialzie index
        i = 0

        # generate random values for m and c
        np.random.seed(self.seed)
        self.m = np.random.rand()
        self.c = np.random.rand()

        # begin Gradient Descent algorithm
        while i < self.numEpochs:

            # obtain predictions based on current m and c
            yHat = self.m*self.xTrue + self.c

            # update parameters
            self.m +=  -1*self.learningRate*self.partialM(self.xTrue, self.y
            self.c += -1*self.learningRate*self.partialC(self.yTrue, yHat)

            # update loss value
```

```
            self.loss = MSE(self.yTrue, yHat)
            self.lossHistory.append(self.loss)

            if i % 10 == 0:
                print(f"epoch: {i}, loss: {self.loss.round(10)}, m: {self.m.

            i += 1

        print(f"\nTraining Summary ({i} epochs): loss: {self.loss.round(10)}

    def history(self):
        return self.lossHistory, self.learningRate, self.m, self.c
```

Now we initialize an instance of the `GradientDescent` class and run the `start()` method.

In [ ]:

```
# instantiate a GradientDescent instance
LinearModel1 = GradientDescent(x_example, y_example, 0.5, 200, LinearPartial

# run the algorithm
LinearModel1.start()
```

```
epoch: 0, loss: 0.5503705653, m: 0.30405, c: 0.17979
epoch: 10, loss: 0.0094255094, m: 0.68943, c: 0.16335
epoch: 20, loss: 0.0023711224, m: 0.84423, c: 0.08193
epoch: 30, loss: 0.0005964899, m: 0.92187, c: 0.04109
epoch: 40, loss: 0.0001500556, m: 0.96081, c: 0.02061
epoch: 50, loss: 3.77486e-05, m: 0.98035, c: 0.01034
epoch: 60, loss: 9.4962e-06, m: 0.99014, c: 0.00518
epoch: 70, loss: 2.3889e-06, m: 0.99506, c: 0.0026
epoch: 80, loss: 6.01e-07, m: 0.99752, c: 0.0013
epoch: 90, loss: 1.512e-07, m: 0.99876, c: 0.00065
epoch: 100, loss: 3.8e-08, m: 0.99938, c: 0.00033
epoch: 110, loss: 9.6e-09, m: 0.99969, c: 0.00016
epoch: 120, loss: 2.4e-09, m: 0.99984, c: 8e-05
epoch: 130, loss: 6e-10, m: 0.99992, c: 4e-05
epoch: 140, loss: 2e-10, m: 0.99996, c: 2e-05
epoch: 150, loss: 0.0, m: 0.99998, c: 1e-05
epoch: 160, loss: 0.0, m: 0.99999, c: 1e-05
epoch: 170, loss: 0.0, m: 1.0, c: 0.0
epoch: 180, loss: 0.0, m: 1.0, c: 0.0
epoch: 190, loss: 0.0, m: 1.0, c: 0.0

Training Summary (200 epochs): loss: 0.0, m: 1.0, c: 0.0
```

Our completed linear model is $y = 1.0x + 0.0$.

Plotting the error

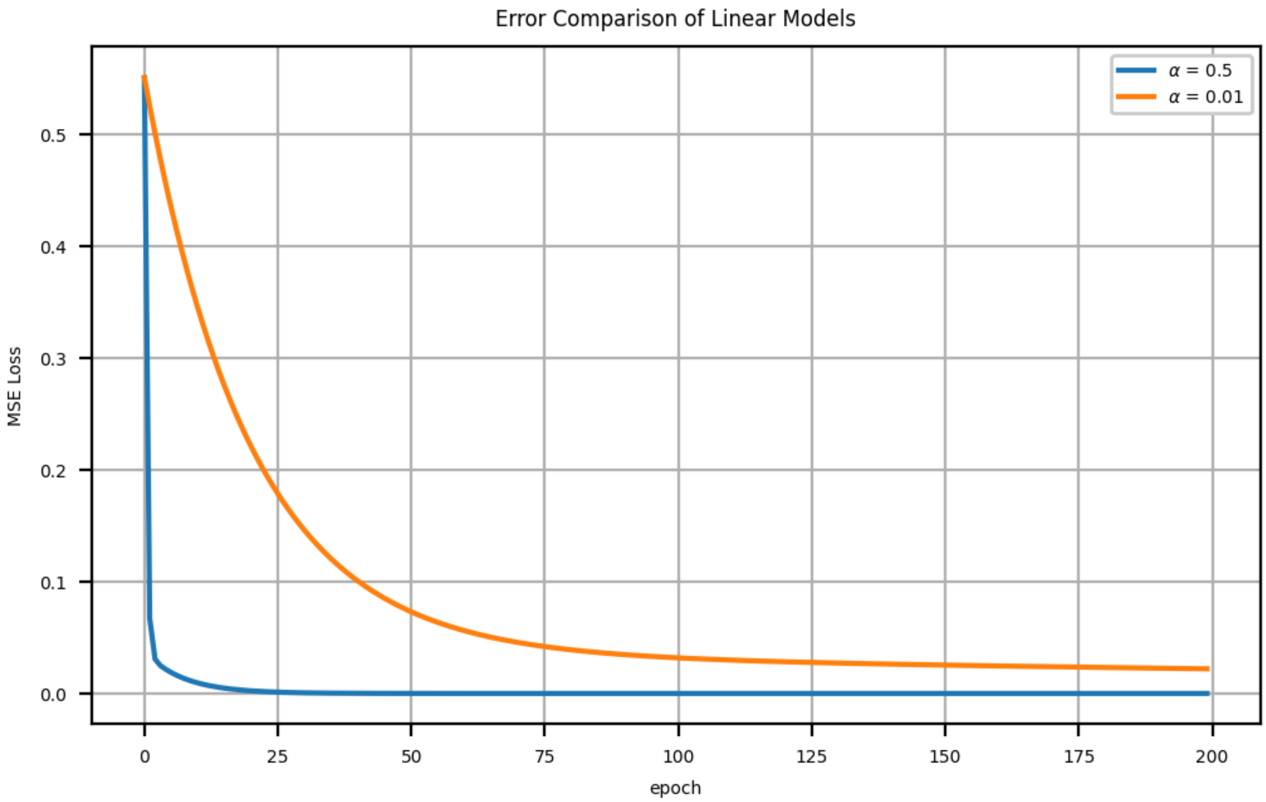
We will know instatiate a second `GradientDescent` object with a different `lr` parameter and plot the error for the two objects.

```python
# create second linear model
LinearModel2 = GradientDescent(x_example, y_example, 0.01, 200, LinearPartia

# run the algorithm
LinearModel2.start()
```

```
epoch: 0, loss: 0.5503705653, m: 0.62717, c: 0.89935
epoch: 10, loss: 0.3445637126, m: 0.57077, c: 0.7708
epoch: 20, loss: 0.2205747043, m: 0.52891, c: 0.67019
epoch: 30, loss: 0.1457405002, m: 0.49831, c: 0.59125
epoch: 40, loss: 0.1004413013, m: 0.47641, c: 0.52912
epoch: 50, loss: 0.0728919338, m: 0.46121, c: 0.48003
epoch: 60, loss: 0.0560132147, m: 0.4512, c: 0.44105
epoch: 70, loss: 0.0455526911, m: 0.44518, c: 0.40993
epoch: 80, loss: 0.0389558344, m: 0.44224, c: 0.38491
epoch: 90, loss: 0.0346880055, m: 0.44164, c: 0.36463
epoch: 100, loss: 0.0318271826, m: 0.44286, c: 0.34803
epoch: 110, loss: 0.0298193282, m: 0.44545, c: 0.33429
epoch: 120, loss: 0.028331445, m: 0.44908, c: 0.32279
epoch: 130, loss: 0.0271633124, m: 0.4535, c: 0.31304
epoch: 140, loss: 0.0261944696, m: 0.45851, c: 0.30464
epoch: 150, loss: 0.0253523695, m: 0.46396, c: 0.29732
epoch: 160, loss: 0.0245932505, m: 0.46972, c: 0.29082
epoch: 170, loss: 0.023890646, m: 0.47571, c: 0.28499
epoch: 180, loss: 0.0232284828, m: 0.48186, c: 0.27968
epoch: 190, loss: 0.0225969351, m: 0.4881, c: 0.27479

Training Summary (200 epochs): loss: 0.0220496515, m: 0.49377, c: 0.27067
```

We can now leverage the use of the `history()` method to obtain the errors of each model. MSE error is plotted in Log scale for clarity.

```python
# obtain training history for both models
history1 = LinearModel1.history()
history2 = LinearModel2.history()

# plotting the errors
# plot linear model
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)
ax.plot(history1[0], label = r'$\alpha$ = 0.5')
ax.plot(history2[0], label = r'$\alpha$ = 0.01')
ax.set_title("Error Comparison of Linear Models")
ax.set_xlabel('epoch')
ax.set_ylabel('MSE Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```



Quadratic Gradient Descent

For the quadratic model $y = m_1x + m_2x^2 + c$, we obtain the following MSE loss function:

$$\mathrm{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - (m_1x + m_2x^2 + c))^2$$

With the following partial derivatives:

$$\frac{\partial L}{\partial m_1} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (m_1x + m_2x^2 + c))x = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)x$$

$$\frac{\partial L}{\partial m_2} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (m_1x + m_2x^2 + c))x^2 = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)x^2$$

$$\frac{\partial L}{\partial c} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (m_1x + m_2x^2 + c)) = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)$$

In python:

In [ ]:
```
# partial derivative w.r.t. m_1
def QuadraticPartialM1(x: np.array, y : np.array, yHat: np.array):
    return (-2/len(x))*np.dot(y-yHat,x)
```

```python
def QuadraticPartialM2(x: np.array, y : np.array, yHat: np.array):
    return (-2/len(x))*np.dot(y-yHat,np.square(x))

def QuadraticPartialC(y : np.array, yHat: np.array):
    return (-2/len(y))*np.sum(y-yHat)
```

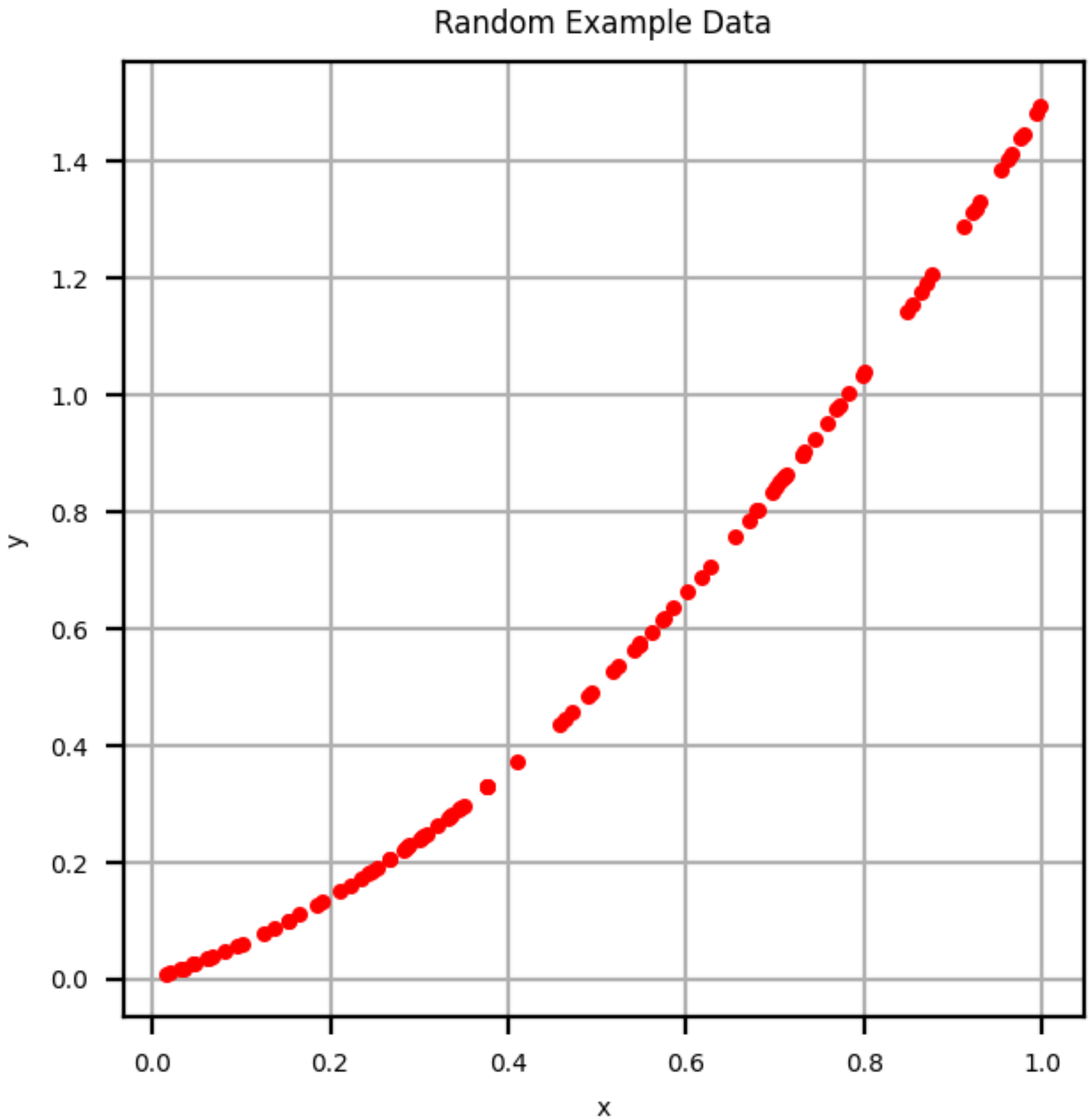
We can now modify the `GradientDescent` class to account for the third partial derivative.

In [ ]:
```python
# create Quadratic Gradient Descent Class
class QuadraticGradientDescent:
    def __init__(self,
                 x_example: np.array,
                 y_example: np.array,
                 lr: float,
                 epochs: int,
                 partialM1: object,
                 partialM2: object,
                 partialC: object,
                 seed: int):

        # input variables
        self.xTrue = x_example # input x
        self.yTrue = y_example # input y
        self.learningRate = lr # learning rate
        self.numEpochs = epochs # number of epochs
        self.partialM1 = partialM1 # partial w.r.t. m1
        self.partialM2 = partialM2 # partial w.r.t. m2
        self.partialC = partialC # partial w.r.t. c
        self.seed = seed # random seed for reproducibility

        # tracking metrics
        self.lossHistory = []

    # run gradient descent
    def start(self):

        # initialzie index
        i = 0

        # generate random values for m and c
        np.random.seed(self.seed)
        self.m1 = np.random.rand()
        self.m2 = np.random.rand()
        self.c = np.random.rand()

        # begin Gradient Descent algorithm
        print(f"Initial Parameters: m1: {self.m1}, m2: {self.m2}, c: {self.c

        while i < self.numEpochs:

            # obtain predictions based on current m1, m2, and c
```

```
            yHat = self.m1*self.xTrue + self.m2*np.square(self.xTrue)+self.c

            # update parameters
            self.m1 +=  -1*self.learningRate*self.partialM1(self.xTrue, self
            self.m2 +=  -1*self.learningRate*self.partialM2(self.xTrue, self
            self.c += -1*self.learningRate*self.partialC(self.yTrue, yHat)

            # update loss value
            self.loss = MSE(self.yTrue, yHat)
            self.lossHistory.append(self.loss)

            if i % 10 == 0:
                print(f"epoch: {i}, loss: {self.loss.round(10)}, m1: {self.m

            i += 1

        print(f"\nTraining Summary ({i} epochs): loss: {self.loss.round(10)}

    def history(self):
        return self.lossHistory, self.learningRate, self.m1, self.m2, self.c
```

Now we create sample data and instanstiate instances of the `QuadraticGradientDescent` object to create two models.

In [ ]:
```
# initialize seed for reproducibility
np.random.seed(567)

m1 = 0.5
m2 = 1.0
c = 0

# create random example data
x_example_quadratic = np.random.rand(100)
y_example_quadratic = m1*x_example_quadratic + m2*x_example_quadratic**2 + c

# plot linear model
fig, ax = plt.subplots(figsize=(3.5, 3.5), dpi=200)
ax.scatter(x_example_quadratic, y_example_quadratic, zorder=1, c="r", s = 5)
ax.set_title("Random Example Data")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True)
ax.set_axisbelow(1)
```



In [ ]:
```
# create object
QuadraticModel1 = QuadraticGradientDescent(x_example_quadratic, y_example_qu
                                           0.1,
                                           300,
                                           QuadraticPartialM1,
                                           QuadraticPartialM2,
                                           QuadraticPartialC,
                                           576)


# run algorithm
QuadraticModel1.start()
```

```
Initial Parameters: m1: 0.6337605448331272, m2: 0.9140319527336822, c: 0.913
7978208648712

epoch: 0, loss: 0.9056425945, m1: 0.53955, m2: 0.85151, c: 0.72349
epoch: 10, loss: 0.0185745316, m1: 0.34439, m2: 0.7447, c: 0.20914
epoch: 20, loss: 0.010418527, m1: 0.37771, m2: 0.79118, c: 0.14784
epoch: 30, loss: 0.0062740964, m1: 0.4106, m2: 0.83222, c: 0.11318
epoch: 40, loss: 0.0037795234, m1: 0.43635, m2: 0.86429, c: 0.08677
epoch: 50, loss: 0.0022773384, m1: 0.45632, m2: 0.88921, c: 0.06631
epoch: 60, loss: 0.0013727439, m1: 0.47178, m2: 0.90857, c: 0.05044
epoch: 70, loss: 0.000828004, m1: 0.48375, m2: 0.92363, c: 0.03812
epoch: 80, loss: 0.0004999601, m1: 0.49301, m2: 0.93535, c: 0.02858
epoch: 90, loss: 0.0003024053, m1: 0.50016, m2: 0.94447, c: 0.02117
epoch: 100, loss: 0.0001834281, m1: 0.50568, m2: 0.95157, c: 0.01543
epoch: 110, loss: 0.0001117686, m1: 0.50994, m2: 0.95712, c: 0.01098
epoch: 120, loss: 6.86029e-05, m1: 0.51321, m2: 0.96145, c: 0.00754
epoch: 130, loss: 4.25955e-05, m1: 0.51572, m2: 0.96483, c: 0.00487
epoch: 140, loss: 2.69206e-05, m1: 0.51763, m2: 0.96749, c: 0.0028
epoch: 150, loss: 1.74678e-05, m1: 0.51909, m2: 0.96958, c: 0.0012
epoch: 160, loss: 1.1762e-05, m1: 0.52019, m2: 0.97123, c: -3e-05
epoch: 170, loss: 8.3127e-06, m1: 0.52102, m2: 0.97254, c: -0.00099
epoch: 180, loss: 6.2223e-06, m1: 0.52163, m2: 0.97359, c: -0.00172
epoch: 190, loss: 4.9504e-06, m1: 0.52208, m2: 0.97442, c: -0.00229
epoch: 200, loss: 4.1715e-06, m1: 0.5224, m2: 0.9751, c: -0.00272
epoch: 210, loss: 3.6896e-06, m1: 0.52261, m2: 0.97565, c: -0.00305
epoch: 220, loss: 3.3867e-06, m1: 0.52276, m2: 0.97611, c: -0.0033
epoch: 230, loss: 3.1918e-06, m1: 0.52284, m2: 0.97649, c: -0.00349
epoch: 240, loss: 3.0619e-06, m1: 0.52287, m2: 0.97681, c: -0.00363
epoch: 250, loss: 2.9714e-06, m1: 0.52287, m2: 0.97709, c: -0.00374
epoch: 260, loss: 2.9047e-06, m1: 0.52284, m2: 0.97733, c: -0.00382
epoch: 270, loss: 2.8526e-06, m1: 0.52279, m2: 0.97755, c: -0.00387
epoch: 280, loss: 2.8092e-06, m1: 0.52273, m2: 0.97774, c: -0.00391
epoch: 290, loss: 2.7713e-06, m1: 0.52265, m2: 0.97791, c: -0.00393

Training Summary (300 epochs): loss: 2.7402e-06, m1: 0.52257, m2: 0.97806, c
: -0.00395
```

In [ ]:

```python
# create object
QuadraticModel2 = QuadraticGradientDescent(x_example_quadratic, y_example_qu
                                           0.01,
                                           300,
                                           QuadraticPartialM1,
                                           QuadraticPartialM2,
                                           QuadraticPartialC,
                                           576)

# run algorithm
QuadraticModel2.start()
```

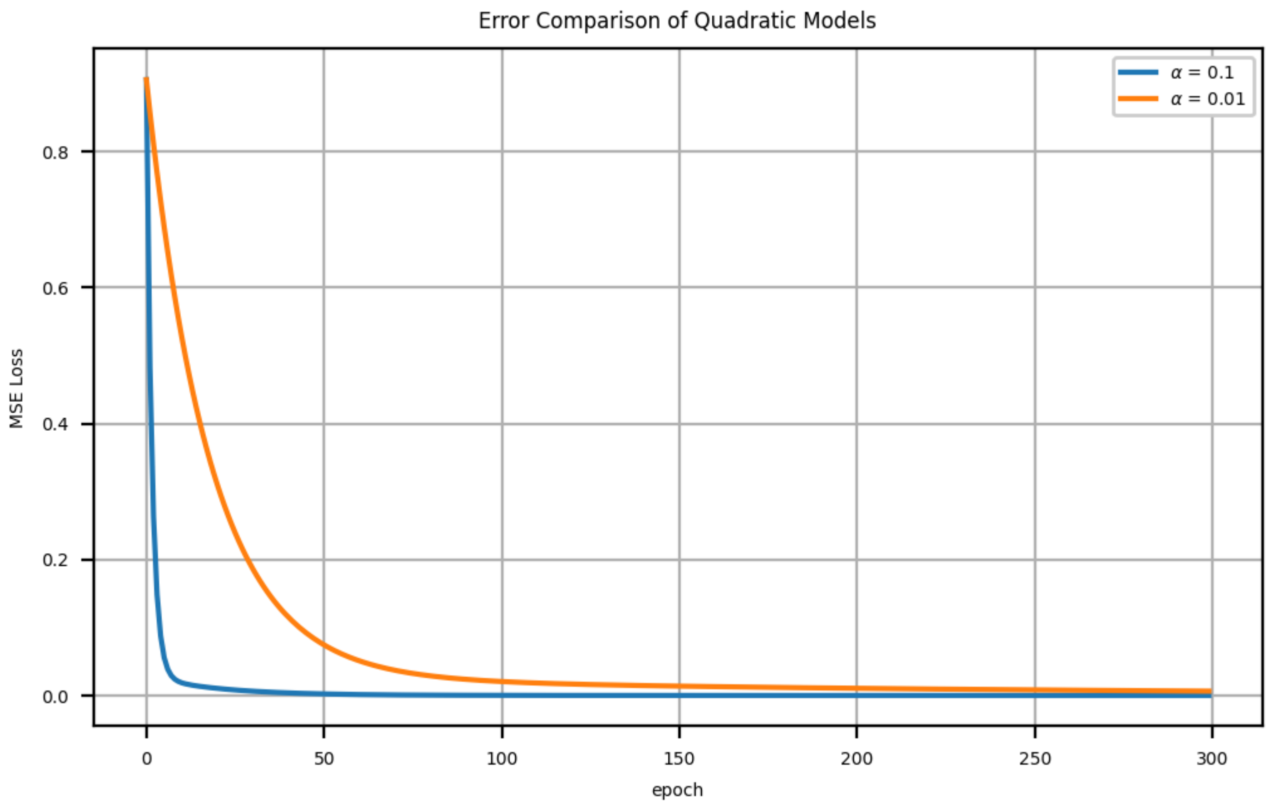
```
Initial Parameters: m1: 0.6337605448331272, m2: 0.9140319527336822, c: 0.913
7978208648712

epoch: 0, loss: 0.9056425945, m1: 0.62434, m2: 0.90778, c: 0.89477
epoch: 10, loss: 0.5256892006, m1: 0.54423, m2: 0.85508, c: 0.73049
epoch: 20, loss: 0.3092160459, m1: 0.48522, m2: 0.81706, c: 0.60523
epoch: 30, loss: 0.1856784656, m1: 0.44208, m2: 0.79007, c: 0.50942
epoch: 40, loss: 0.114983147, m1: 0.41088, m2: 0.77136, c: 0.43585
epoch: 50, loss: 0.0743429888, m1: 0.38864, m2: 0.75884, c: 0.37907
epoch: 60, loss: 0.0508064356, m1: 0.37313, m2: 0.75095, c: 0.33499
epoch: 70, loss: 0.0370117377, m1: 0.36266, m2: 0.74652, c: 0.30052
epoch: 80, loss: 0.0287740048, m1: 0.35595, m2: 0.74466, c: 0.27331
epoch: 90, loss: 0.0237138744, m1: 0.35206, m2: 0.74469, c: 0.25161
epoch: 100, loss: 0.0204781523, m1: 0.35026, m2: 0.74612, c: 0.2341
epoch: 110, loss: 0.0182968846, m1: 0.35, m2: 0.74856, c: 0.21977
epoch: 120, loss: 0.0167316289, m1: 0.35089, m2: 0.75174, c: 0.20786
epoch: 130, loss: 0.0155323905, m1: 0.3526, m2: 0.75544, c: 0.19781
epoch: 140, loss: 0.014556313, m1: 0.35492, m2: 0.7595, c: 0.18919
epoch: 150, loss: 0.0137214675, m1: 0.35767, m2: 0.7638, c: 0.18167
epoch: 160, loss: 0.0129805875, m1: 0.36071, m2: 0.76825, c: 0.17501
epoch: 170, loss: 0.0123061403, m1: 0.36396, m2: 0.77278, c: 0.16902
epoch: 180, loss: 0.0116818412, m1: 0.36733, m2: 0.77734, c: 0.16355
epoch: 190, loss: 0.0110978292, m1: 0.37078, m2: 0.7819, c: 0.15851
epoch: 200, loss: 0.0105479236, m1: 0.37425, m2: 0.78644, c: 0.15381
epoch: 210, loss: 0.0100280635, m1: 0.37773, m2: 0.79092, c: 0.14938
epoch: 220, loss: 0.0095354193, m1: 0.38119, m2: 0.79534, c: 0.14519
epoch: 230, loss: 0.0090678873, m1: 0.38462, m2: 0.79968, c: 0.1412
epoch: 240, loss: 0.0086238, m1: 0.388, m2: 0.80395, c: 0.13737
epoch: 250, loss: 0.008201761, m1: 0.39132, m2: 0.80813, c: 0.1337
epoch: 260, loss: 0.0078005501, m1: 0.39459, m2: 0.81222, c: 0.13015
epoch: 270, loss: 0.0074190681, m1: 0.39778, m2: 0.81622, c: 0.12672
epoch: 280, loss: 0.0070563041, m1: 0.40091, m2: 0.82014, c: 0.1234
epoch: 290, loss: 0.0067113167, m1: 0.40398, m2: 0.82396, c: 0.12018

Training Summary (300 epochs): loss: 0.0064152952, m1: 0.40667, m2: 0.82732,
c: 0.11736
```

We plot the loss of each quadratic model with the `history` method:

In [ ]:

```python
# obtain training history for both models
history1 = QuadraticModel1.history()
history2 = QuadraticModel2.history()

# plotting the errors
# plot linear model
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)
ax.plot(history1[0], label = r'$\alpha$ = 0.1')
ax.plot(history2[0], label = r'$\alpha$ = 0.01')
ax.set_title("Error Comparison of Quadratic Models")
ax.set_xlabel('epoch')
ax.set_ylabel('MSE Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
```

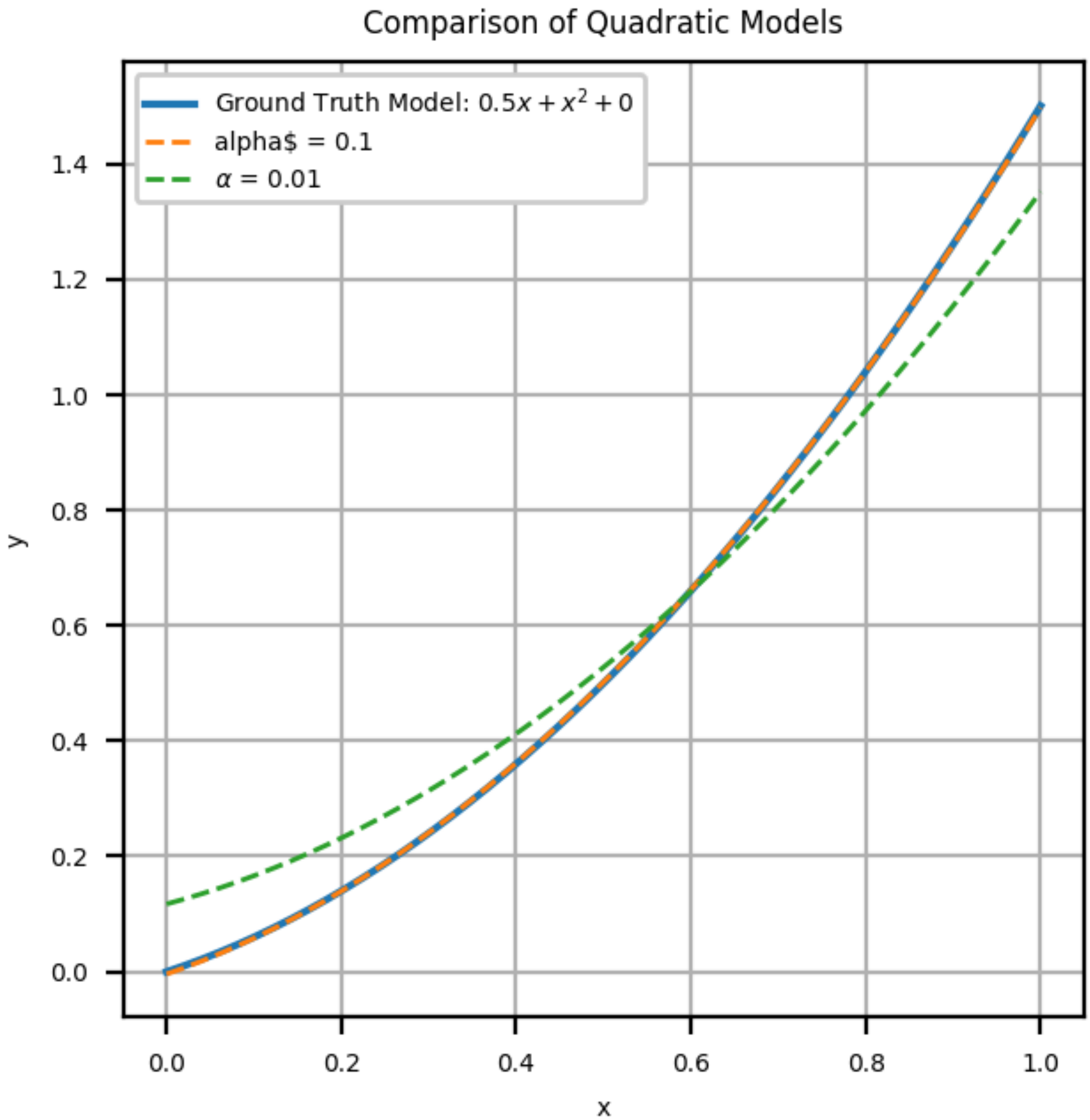
```
ax.grid(True)
```



Similarly, the parameters m_1, m_2, and c are stored in the `history` object and can be used to plot the models against the ground truth model.

In [ ]:
```
# obtain model parameters
model1 = history1[2:]
model2 = history2[2:]

# ground truth model
truth_model = (0.5,1,0)

# define function for plotting
def model_plotting(model: tuple):

    # create x values
    x = np.linspace(0, 1)

    return x,model[0]*x + model[1]*x**2 + model[2]

# plot linear model
fig, ax = plt.subplots(figsize=(3.5,3.5), dpi=200)

ax.plot(model_plotting(truth_model)[0],
        model_plotting(truth_model)[1],
        label = r'Ground Truth Model: $0.5x+x^2+0$'
        )

ax.plot(model_plotting(model1)[0],
```

```
        model_plotting(model1)[1],
        linestyle='dashed',
        label = r'alpha$ = 0.1',
        linewidth = 1)

ax.plot(model_plotting(model2)[0],
        model_plotting(model2)[1],
        linestyle='dashed',
        label = r'$\alpha$ = 0.01',
        linewidth=1)

ax.set_title("Comparison of Quadratic Models")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```

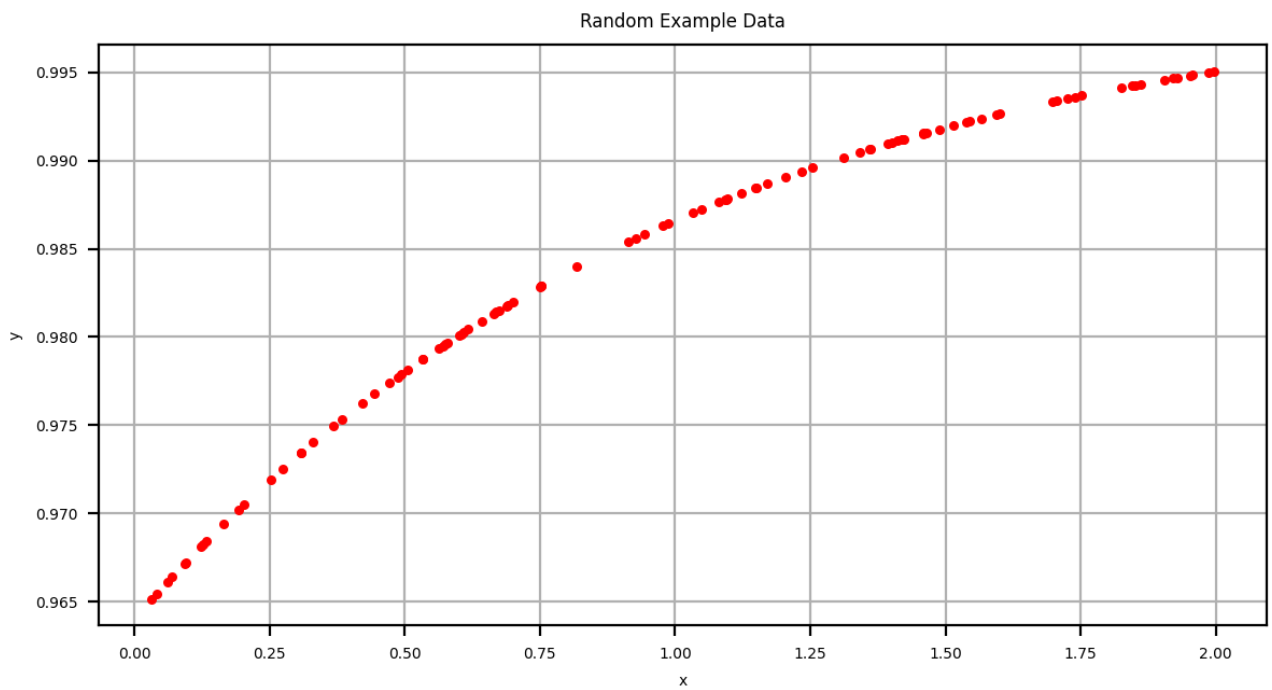


Gradient Descent techniques for both the Linear and Quadratic models exhibit better accuracy at lower learning rates. Large learning rates and a small number of epochs allows the algorithm to converge on the local minima of the MSE loss function. It is important to note that for both implementations of the gradient descent algorithm learning rates above $\alpha > 1,0$ resulted in exploding gradients, making the models unable to learn. Error logs indicate that this is due to out of bound exceptions at run time.

Hyperbolic Tangent Gradient Descent

For the hyperbolioc tangent model $y = \tanh(m * x + c)$ we obtain example data for $x \in [0.2]$ in the following manner:

In [ ]:
```
# initialize seed for reproducibility
np.random.seed(567)

# create random example data
x_hyperbolic = 2*np.random.rand(100)
y_hyperbolic = np.tanh(x_example+2)

# plot linear model
fig, ax = plt.subplots(figsize=(7, 3.5), dpi=200)
ax.scatter(x_hyperbolic, y_hyperbolic, zorder=1, c="r", s = 5)
ax.set_title("Random Example Data")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True)
ax.set_axisbelow(1)
```



similar to the quadratic mode, we obtain the loss function

$$\mathrm{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tanh(m * x + c))^2$$

w.r.t. parameters c and m, the partial derivatives are as follows:

$$\frac{\partial \mathrm{L}}{\partial m} = \frac{2}{n}\sum_{i=1}^{n}(y_i - \tanh(m * x + c))(1 - \tanh^2(m * x + c))x = \frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)(1 -$$

$$\frac{\partial \mathrm{L}}{\partial c} = \frac{2}{n}\sum_{i=1}^{n}(y_i - \tanh(m * x + c))(1 - \tanh^2(m * x + c) = \frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)(1 - \hat{y}_i$$

Implemented in python:

In [ ]:
```python
# partial w.r.t. m
def HyperbolicPartialM(x, y, yHat):
    return (2/len(x))*np.dot(y-yHat, (1 - np.square(yHat)*x))

# partial w.r.t. c
def HyperbolicPartialC(y, yHat):
    return (2/len(y))*np.dot(y-yHat, 1 - np.square(yHat))
```

We can now leverage a modified `GradientDescent` class to perform the algorithm.

In [ ]:
```python
# create Gradient Descent Class
class HyperbolicGradientDescent:
    def __init__(self,
                 x_example: np.array,
                 y_example: np.array,
                 lr: float,
                 epochs: int,
                 dLdm: object,
                 dLdc: object,
                 seed: int):

        # input variables
        self.xTrue = x_example # input x
        self.yTrue = y_example # input y
        self.learningRate = lr # learning rate
        self.numEpochs = epochs # number of epochs
        self.partialM = dLdm # partial w.r.t. m
        self.partialC = dLdc # partial w.r.t. c
        self.seed = seed # random seed for reproducibility

        # variables to be populated after runtime
        self.m = 0
        self.c = 0
        self.loss = 0

        # tracking metrics
        self.lossHistory = []

    # run gradient descent
    def start(self):

        # initialzie index
        i = 0

        # generate random values for m and c
        np.random.seed(self.seed)
        self.m = np.random.rand()
        self.c = np.random.rand()

        # begin Gradient Descent algorithm
        print(f"Initial Parameters: m: {self.m} c: {self.c}\n")
```

```python
        while i < self.numEpochs:

            # obtain predictions based on current m and c
            yHat = np.tanh(self.m*self.xTrue+self.c)

            # update parameters
            self.m +=  1*self.learningRate*self.partialM(self.xTrue, self.yT
            self.c += 1*self.learningRate*self.partialC(self.yTrue, yHat)

            # update loss value
            self.loss = MSE(self.yTrue, yHat)
            self.lossHistory.append(self.loss)

            if i % 20 == 0:
                print(f"epoch: {i}, loss: {self.loss.round(10)}, m: {self.m.

            i += 1

        print(f"\nTraining Summary ({i} epochs): loss: {self.loss.round(10)}

    def history(self):
        return self.lossHistory, self.learningRate, self.m, self.c
```

In [ ]:
```python
HyperbolicModel1 = HyperbolicGradientDescent(x_hyperbolic,
                                              y_hyperbolic,
                                              0.8,
                                              400,
                                              HyperbolicPartialM,
                                              HyperbolicPartialC,
                                              576)

HyperbolicModel1.start()
```

```
Initial Parameters: m: 0.6337605448331272 c: 0.9140319527336822

epoch: 0, loss: 0.0126170849, m: 0.70904, c: 0.95649
epoch: 20, loss: 0.0013042466, m: 1.46273, c: 1.20034
epoch: 40, loss: 0.0006682789, m: 1.85625, c: 1.27604
epoch: 60, loss: 0.0004666872, m: 2.13117, c: 1.32135
epoch: 80, loss: 0.000372775, m: 2.34157, c: 1.35354
epoch: 100, loss: 0.0003202143, m: 2.5104, c: 1.37838
epoch: 120, loss: 0.0002874248, m: 2.64988, c: 1.39854
epoch: 140, loss: 0.00026544, m: 2.76737, c: 1.41544
epoch: 160, loss: 0.0002499179, m: 2.86769, c: 1.42997
epoch: 180, loss: 0.0002385219, m: 2.95421, c: 1.44267
epoch: 200, loss: 0.0002298932, m: 3.02939, c: 1.45394
epoch: 220, loss: 0.0002231933, m: 3.09505, c: 1.46405
epoch: 240, loss: 0.0002178798, m: 3.15264, c: 1.47321
epoch: 260, loss: 0.0002135887, m: 3.20327, c: 1.48158
epoch: 280, loss: 0.0002100671, m: 3.24786, c: 1.48927
epoch: 300, loss: 0.0002071355, m: 3.28716, c: 1.49637
epoch: 320, loss: 0.0002046628, m: 3.32178, c: 1.50298
epoch: 340, loss: 0.000202552, m: 3.35226, c: 1.50914
epoch: 360, loss: 0.0002007296, m: 3.37904, c: 1.51492
epoch: 380, loss: 0.0001991395, m: 3.40249, c: 1.52036

Training Summary (400 epochs): loss: 0.0001978043, m: 3.422, c: 1.52523
```

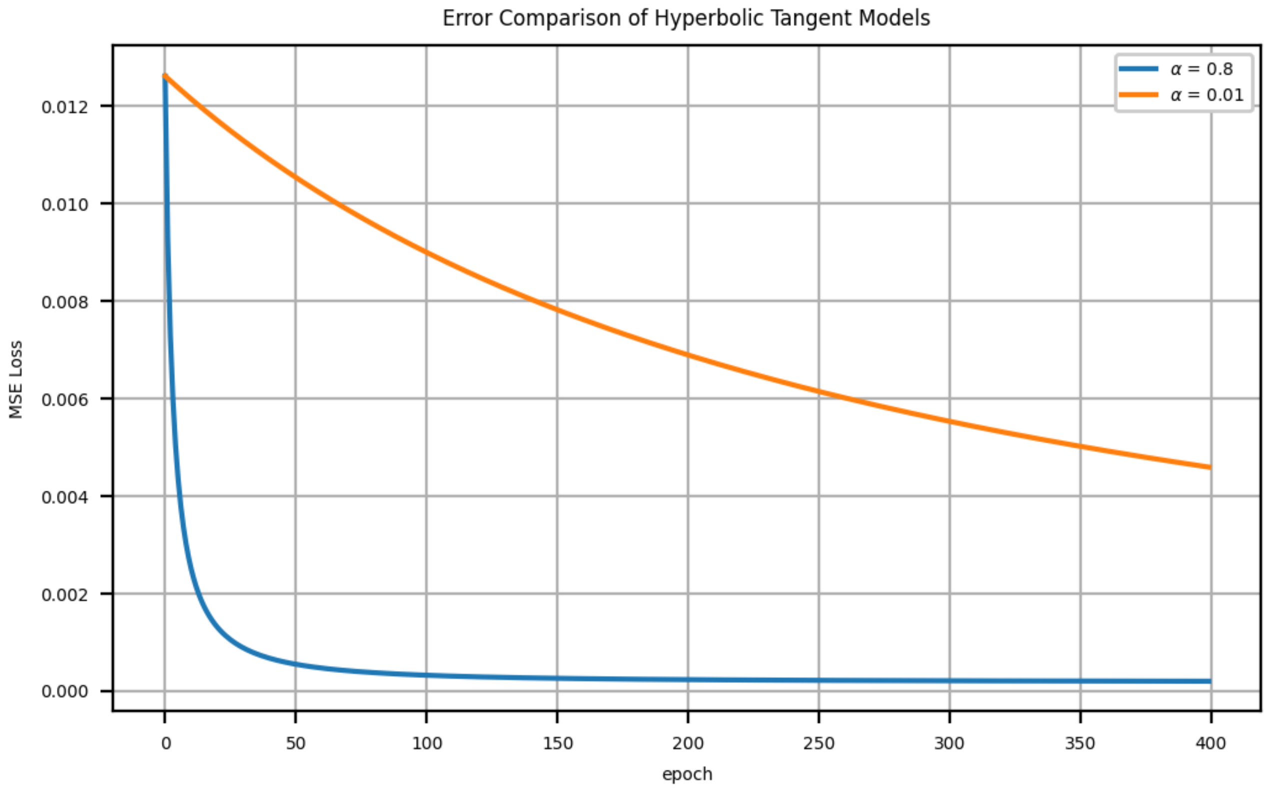
In [ ]:

```
HyperbolicModel2 = HyperbolicGradientDescent(x_hyperbolic,
                                   y_hyperbolic,
                                   0.01,
                                   400,
                                   HyperbolicPartialM,
                                   HyperbolicPartialC,
                                   576)

HyperbolicModel2.start()
```

```
Initial Parameters: m: 0.6337605448331272 c: 0.9140319527336822

epoch: 0, loss: 0.0126170849, m: 0.6347, c: 0.91456
epoch: 20, loss: 0.0117031174, m: 0.65316, c: 0.9248
epoch: 40, loss: 0.0109012423, m: 0.67098, c: 0.93439
epoch: 60, loss: 0.0101926076, m: 0.68823, c: 0.94339
epoch: 80, loss: 0.0095623182, m: 0.70495, c: 0.95188
epoch: 100, loss: 0.0089984548, m: 0.72119, c: 0.9599
epoch: 120, loss: 0.0084913701, m: 0.73698, c: 0.9675
epoch: 140, loss: 0.008033175, m: 0.75237, c: 0.97472
epoch: 160, loss: 0.0076173582, m: 0.76738, c: 0.9816
epoch: 180, loss: 0.0072384996, m: 0.78203, c: 0.98816
epoch: 200, loss: 0.0068920527, m: 0.79635, c: 0.99444
epoch: 220, loss: 0.0065741761, m: 0.81035, c: 1.00044
epoch: 240, loss: 0.0062816037, m: 0.82407, c: 1.0062
epoch: 260, loss: 0.0060115409, m: 0.8375, c: 1.01174
epoch: 280, loss: 0.0057615836, m: 0.85067, c: 1.01706
epoch: 300, loss: 0.0055296526, m: 0.8636, c: 1.02218
epoch: 320, loss: 0.0053139411, m: 0.87628, c: 1.02713
epoch: 340, loss: 0.0051128719, m: 0.88874, c: 1.0319
epoch: 360, loss: 0.0049250623, m: 0.90099, c: 1.03651
epoch: 380, loss: 0.0047492951, m: 0.91303, c: 1.04097

Training Summary (400 epochs): loss: 0.0045924899, m: 0.92428, c: 1.04508
```

In [ ]:

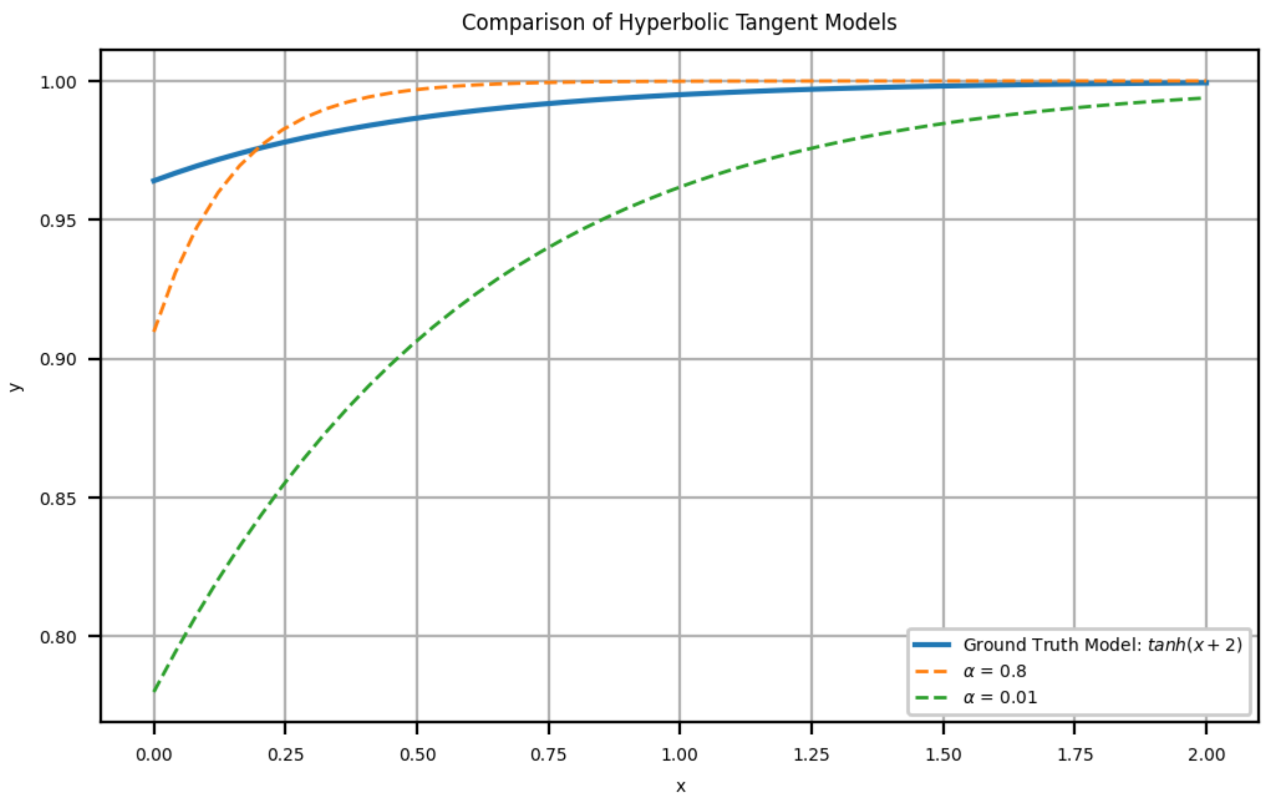
```python
# obtain training history for both models
history1 = HyperbolicModel1.history()
history2 = HyperbolicModel2.history()

# plotting the errors
# plot linear model
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)
ax.plot(history1[0], label = r'$\alpha$ = 0.8')
ax.plot(history2[0], label = r'$\alpha$ = 0.01')
ax.set_title("Error Comparison of Hyperbolic Tangent Models")
ax.set_xlabel('epoch')
ax.set_ylabel('MSE Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```



In [ ]:

```
# obtain model parameters
model1 = history1[2:]
model2 = history2[2:]

# ground truth model
truth_model = (1.0,2.0)

# define function for plotting
def model_plotting(model: tuple):

    # create x values
    x = np.linspace(0, 2)

    return x, np.tanh(model[0]*x+model[1])

# plot linear model
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(model_plotting(truth_model)[0],
        model_plotting(truth_model)[1],
        label = r'Ground Truth Model: $tanh(x+2)$'
        )


ax.plot(model_plotting(model1)[0],
        model_plotting(model1)[1],
        linestyle='dashed',
        label = r'$\alpha$ = 0.8',
        linewidth = 1)
```

```
ax.plot(model_plotting(model2)[0],
        model_plotting(model2)[1],
        linestyle='dashed',
        label = r'$\alpha$ = 0.01',
        linewidth=1)


ax.set_title("Comparison of Hyperbolic Tangent Models")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```



3. ML Basics

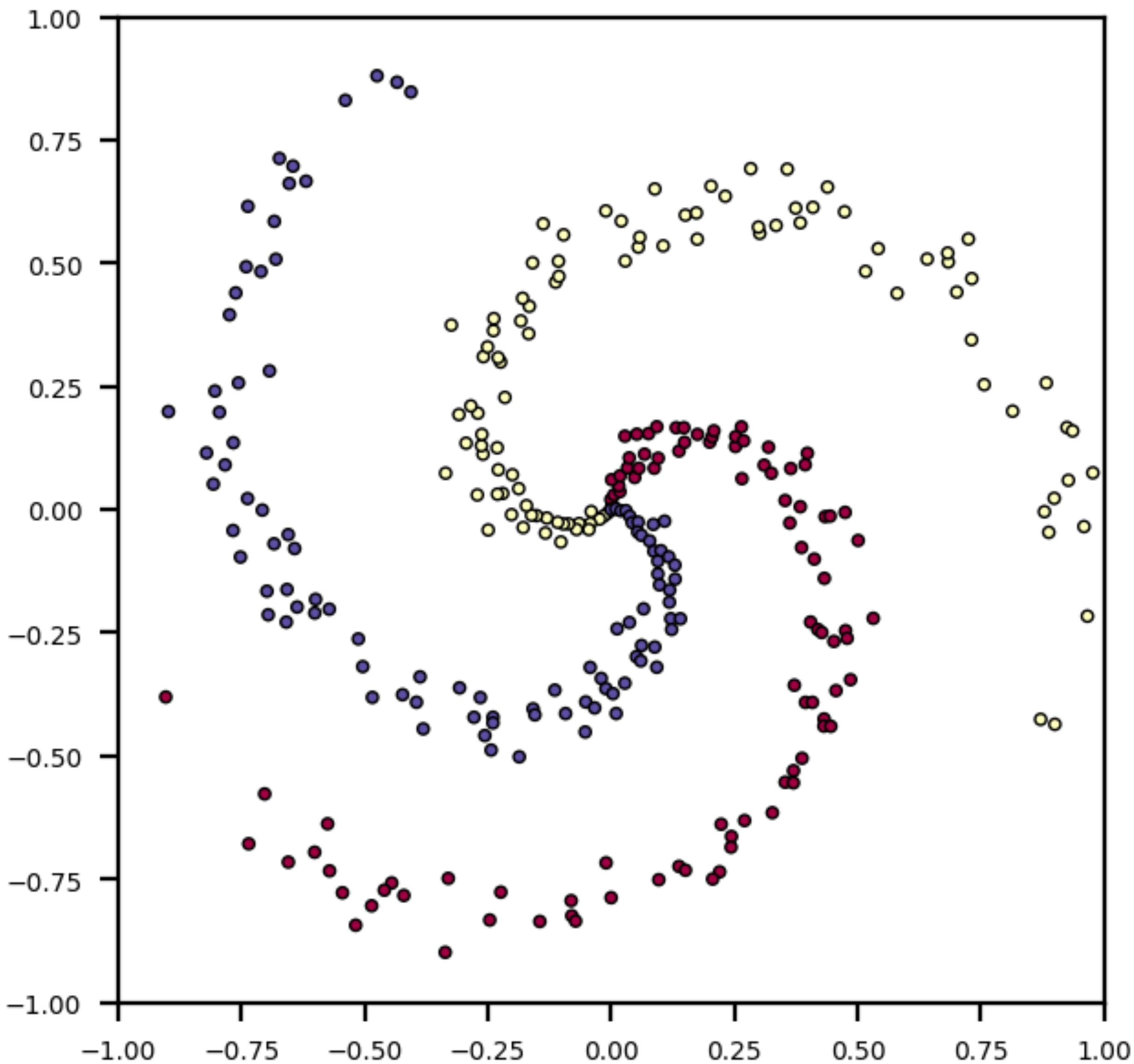
Logistic (multiclass or cross-entropy) Loss

Begin by downloading the generated data from `Data_Linear_Classifier.ipynb`

In [ ]:
```
import pickle

X = pickle.load(open('Misc_files/dataX.pickle','rb'))
y = pickle.load(open('Misc_files/dataY.pickle','rb'))

fig, ax = plt.subplots(figsize=(3.5,3.5), dpi=200)
```

```
ax.scatter(X[:,0], X[:,1], c=y, s=5, cmap=plt.cm.Spectral, edgecolors="black
ax.set_aspect('equal')
ax.set_xlim(-1, 1)
ax.set_ylim(-1, 1)
```

Out[ ]: (-1.0, 1.0)



In addition to the data we initialize random parameters W and b for testing

In [ ]:
```
# random seed for reproducibility
np.random.seed(576)

# get Array W of shape (2, 3)
W = 0.01*np.random.randn(X.shape[1],max(y)+1)

# get Array K of shape (1, 3)
b = np.zeros((1, max(y)+1))
```

Next build the *cross-entropy* function with inputs (X, y, W, b)

In [ ]:

```
def crossentropy_loss(X, y, W, b):

    # evaluate class scores
    scores = np.dot(X, W) + b

    # compute softmax of scores
    probs = np.exp(scores)/np.sum(np.exp(scores), axis=1, keepdims=True)

    # compute loss
    correct_logprobs = -np.log(probs[range(X.shape[0]),y])
    data_loss = np.sum(correct_logprobs)/X.shape[0]

    return data_loss

# test function
print("loss "+str(crossentropy_loss(X, y, W, b)))
```

```
loss 1.0988948999636798
```

l_1 and l_2 Regularization

We can revise the `crossentropy_loss` function by introducing L_1 and L_2 regularization. More formally:

$$\mathrm{L}_1 = \sum_i \sum_j |W_{ij}|$$

$$\mathrm{L}_2 = \sum_i \sum_j W_{ij}^2$$

By default, we will initialize function parameters `l1` and `l2` at `0.0`.

In [ ]:

```
def crossentropy_loss_regularization(X, y, W, b, l1=0.0, l2=0.0):

    # evaluate class scores
    scores = np.dot(X, W) + b

    # compute softmax of scores
    probs = np.exp(scores)/np.sum(np.exp(scores), axis=1, keepdims=True)

    # compute l1 and l2 regularizers
    l1_loss = 0.5*l1*np.sum(np.abs(W))
    l2_loss = 0.5*l2*np.sum(np.square(W))

    # compute loss
    correct_logprobs = -np.log(probs[range(X.shape[0]),y])

    # return total loss
    data_loss = np.sum(correct_logprobs)/X.shape[0] - l1_loss - l2_loss
```

```
    return data_loss

# test function
print("loss "+str(crossentropy_loss_regularization(X, y, W, b, 1e-3, 1e-3)))
```

```
loss 1.0988840322643474
```

4. Classification Pipeline

Train Test Splits

We will use the sci-kit learn `train_test_split` utility to separate a new instance of the data and recreate a new set of initial (W, b) parameters.

In [ ]:
```
# import utility
from sklearn.model_selection import train_test_split

# reload X and y data from the pickle file
X = pickle.load(open('Misc_files/dataX.pickle','rb'))
y = pickle.load(open('Misc_files/dataY.pickle','rb'))

# random seed for reproducibility
np.random.seed(576)

# get Array W of shape (2, 3)
W = 0.01*np.random.randn(X.shape[1],max(y)+1)

# get Array K of shape (1, 3)
b = np.zeros((1, max(y)+1))

# create test train splits, note shuffling on creation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shu
```

Linear Classifier Construction

Let's build a class to instantiate Linear Classifier models. Per the instructions given in the assignment, `l1` regularization will not be taken as a parameter in the class.

In [ ]:
```
class LinearClassifier:
    def __init__(self, X, y, W, b, lr, epochs, l2 = 1e-3):
        self.X = X.copy()
        self.y = y.copy()
        self.W = W.copy()
        self.b = b.copy()
        self.lr = lr
        self.epochs = epochs
        self.l2 = l2

        # metric tracking
```

```
        self.loss = 0
        self.loss_history = []
        self.accuracy_history = []

    # runtime method
    def start(self, verbose = True):

        i = 0

        for i in range(self.epochs):


            # compute the gradient on scores
            num_examples = self.X.shape[0]
            dscores = np.dot(self.X, self.W) + self.b
            dscores[range(num_examples), self.y] -= 1
            dscores /= num_examples

            # backpropogate gradient to parameters (W, b)
            dW = np.dot(self.X.T, dscores)
            db = np.sum(dscores, axis=0, keepdims=True)

            dW += self.l2*self.W # l2 regularization

            # parameter update
            self.W += -self.lr*dW
            self.b += -self.lr*db

            # compute loss, l2 regularization only, and training accuracy
            self.loss = crossentropy_loss_regularization(self.X, self.y, sel
            training_accuracy = self.eval(self.X, self.y)

            #metrics
            self.loss_history.append(self.loss)
            self.accuracy_history.append(training_accuracy)

            # output tracking
            if i % 10 == 0 and verbose == True :
                print(f"iteration {i}: loss: {self.loss} training_accuracy: 

            i += 1

        # exit output
        if verbose == True:
            print(f"iteration {i}: loss: {self.loss} training_accuracy: {tra

    # get W
    def get_W(self):
        return self.W

    # get b
    def get_b(self):
        return self.b
```

```python
    # get params as tuple
    def get_parameters(self):
        return self.W, self.b

    # evaluate model
    def eval(self, X = X, y = y):
        scores = np.dot(X, self.W) + self.b
        pred = np.argmax(scores, axis=1)
        return np.mean(pred == y)

    # plot losses
    def show_loss(self):
        fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)
        ax.plot(self.loss_history)
        ax.set_title("Linear Classifier Loss")
        ax.set_xlabel('Epoch')
        ax.set_ylabel('Loss')
        ax.grid(True)

    # plot training accuracy
    def show_accuracy(self):
        fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)
        ax.plot(self.accuracy_history)
        ax.set_title("Linear Classifier Training Accuracy")
        ax.set_xlabel('Epoch')
        ax.set_ylabel('Training Accuracy')
        ax.grid(True)

    # post training
    def show_classifier(self):
        h = 0.02
        x_min, x_max = self.X[:, 0].min() - 1, self.X[:, 0].max() + 1
        y_min, y_max = self.X[:, 1].min() - 1, self.X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h))
        Z = np.dot(np.c_[xx.ravel(), yy.ravel()], self.W) + self.b
        Z = np.argmax(Z, axis=1)
        Z = Z.reshape(xx.shape)
        fig = plt.figure()
        plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)
        plt.scatter(self.X[:, 0], self.X[:, 1], c=self.y, s=40, cmap=plt.cm.
        plt.xlim(xx.min(), xx.max())
        plt.ylim(yy.min(), yy.max())

    # model summary
    def summary(self):
        print(f"Model summary:"
              +f"\n\nW:\n{self.W}"
              +f"\n\nB:\n{self.b}"
              +f"\n\nLearning Rate: {self.lr}"
              +f"\nl2 Regularization: {self.l2}"
              +f"\nEpochs: {self.epochs}"
```

```
                +f"\n\nTraining Accuracy: {self.accuracy_history[-1]}"
                +f"\nModel Loss: {self.loss_history[-1]}")
```

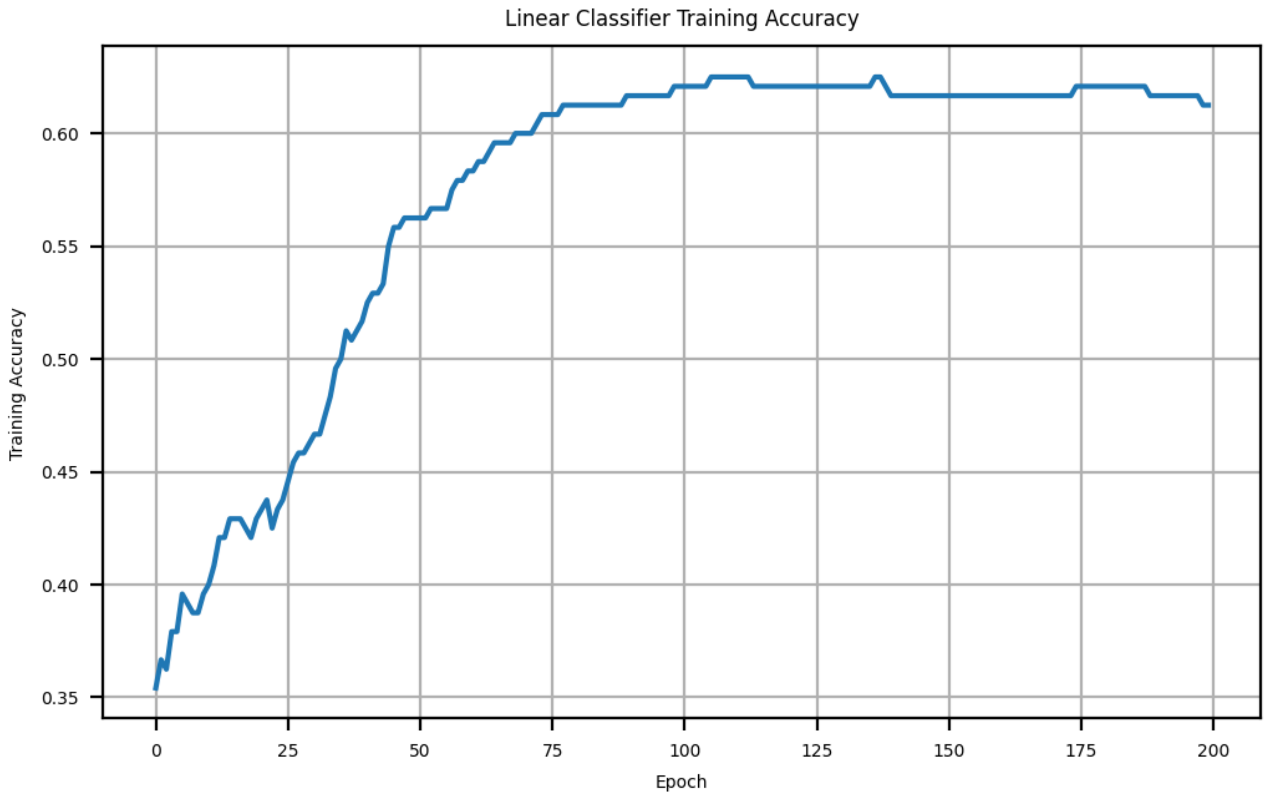
We next instantiate a `LinearClassifier` model and train it on `X_train`, `y_train`, and inital parameters `W` and `b` using the class method `start()`.

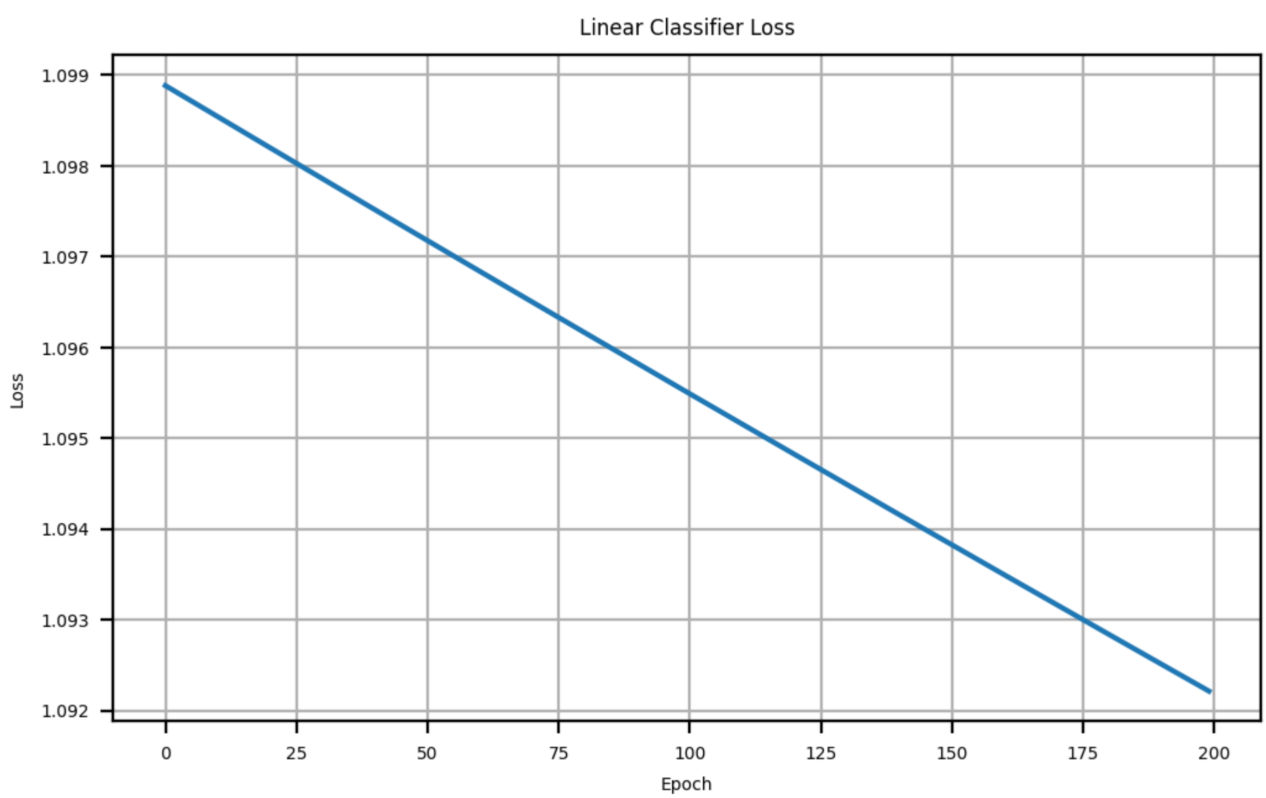
In [ ]:
```
# evaluate on training data
Classifier = LinearClassifier(X_train, y_train, W, b, 1e-3, 200, 1e-3)

# start training
Classifier.start()
```

```
iteration 0: loss: 1.098884320032409 training_accuracy: 0.3541666666666667
iteration 10: loss: 1.0985408705616455 training_accuracy: 0.4
iteration 20: loss: 1.0981983208685286 training_accuracy: 0.4333333333333333
5
iteration 30: loss: 1.0978566669229148 training_accuracy: 0.4666666666666667
iteration 40: loss: 1.0975159047341516 training_accuracy: 0.525
iteration 50: loss: 1.0971760303504645 training_accuracy: 0.5625
iteration 60: loss: 1.0968370398583502 training_accuracy: 0.5833333333333334
iteration 70: loss: 1.0964989293819818 training_accuracy: 0.6
iteration 80: loss: 1.0961616950826265 training_accuracy: 0.6125
iteration 90: loss: 1.095825333158072 training_accuracy: 0.6166666666666667
iteration 100: loss: 1.095489839842064 training_accuracy: 0.6208333333333333
iteration 110: loss: 1.0951552114037537 training_accuracy: 0.625
iteration 120: loss: 1.0948214441471544 training_accuracy: 0.620833333333333
3
iteration 130: loss: 1.0944885344106112 training_accuracy: 0.620833333333333
3
iteration 140: loss: 1.0941564785662745 training_accuracy: 0.616666666666666
7
iteration 150: loss: 1.09382527301959 training_accuracy: 0.6166666666666667
iteration 160: loss: 1.09349491420879 training_accuracy: 0.6166666666666667
iteration 170: loss: 1.0931653986044014 training_accuracy: 0.616666666666666
7
iteration 180: loss: 1.0928367227087583 training_accuracy: 0.620833333333333
3
iteration 190: loss: 1.0925088830555227 training_accuracy: 0.616666666666666
7
iteration 200: loss: 1.0922145395148164 training_accuracy: 0.6125
```

Calling the methods `show_accuracy()` and `show_loss()` the model's training history can now be shown:

In [ ]:
```
# show training accuracy history
Classifier.show_accuracy()

# show loss history
Classifier.show_loss()
```

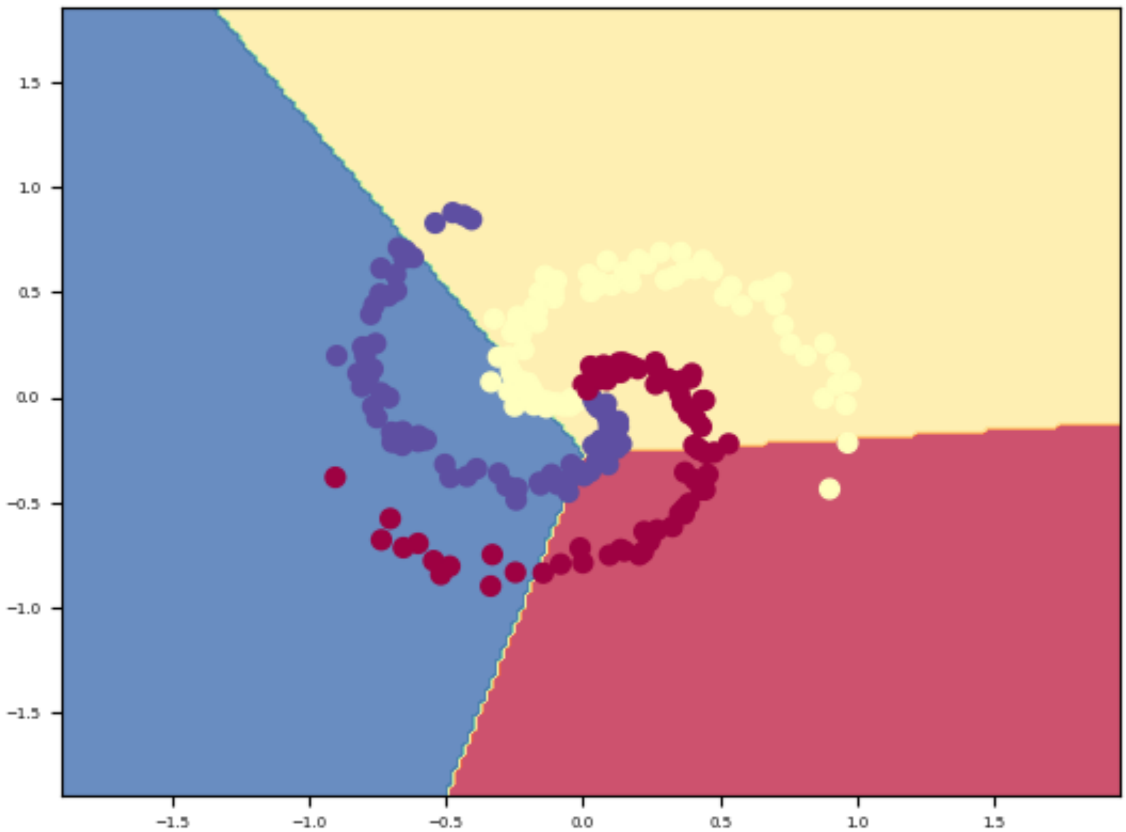




We now call the `summary` method to show the model summary and a `show_classifier()` method to plot the resulting classifier's decision boundary.

In [ ]:
```
Classifier.summary()
Classifier.show_classifier()
```

```
Model summary:

W:
[[ 0.01463343  0.01229308 -0.01946041]
 [-0.01563398  0.01509063 -0.00507448]]

B:
[[0.0567085  0.06500469 0.05966792]]

Learning Rate: 0.001
l2 Regularization: 0.001
Epochs: 200

Training Accuracy: 0.6125
Model Loss: 1.0922145395148164
```



Calling the `eval()` method will evaluate the model on the train and test data.

In [ ]:
```
# get training accuracy
print(f"training accuracy: {Classifier.eval(X_train, y_train)}")

# get testing accuracy
print(f"test accuracy: {Classifier.eval(X_test, y_test)}")
```

```
training accuracy: 0.6125
test accuracy: 0.45
```

A low test accuracy in comparison to the training accuracy typically indicates overfitting, but the radially symmetric distribution of the example data suggests that a linear classifier is not the appropriate choice for this data.

Cross Validation

For the relatively small number of instances (240) in the training set, we decide to use sklearn's `KFold` class for cross validation. Cross validation consists of 10 folds.

In [ ]:
```
from sklearn.model_selection import KFold

# set naive parameters for KFold cross validation
lr = 1e-3
epochs = 300
l2 = 1e-3

# tracking objects
training_history = []
validation_history = []
loss_history = []

# best model tracking
best_accuracy = 0.0

# instantiate KFold object
kf = KFold(n_splits = 10)

# begin cv loop
for i, (train_index, validation_index) in enumerate(kf.split(X_train)):

    # instatiate new LinearClassifier object with cv split data
    cv_classifier = LinearClassifier(X_train[train_index], y_train[train_inc
    cv_classifier.start(verbose = False)

    # obtain tracking metrics
    training_accuracy = cv_classifier.eval(X_train[train_index], y_train[tra
    validation_accuracy = cv_classifier.eval(X_train[validation_index], y_tr
    loss = cv_classifier.loss

    # update best model
    if training_accuracy > best_accuracy:
        best_accuracy = training_accuracy
        best_classifier = cv_classifier

    # append metrics
    training_history.append(training_accuracy)
    validation_history.append(validation_accuracy)
    loss_history.append(loss)

# return mean values
mean_cv_training = np.mean(training_history)
mean_cv_validation = np.mean(validation_history)
mean_cv_loss = np.mean(loss_history)
```

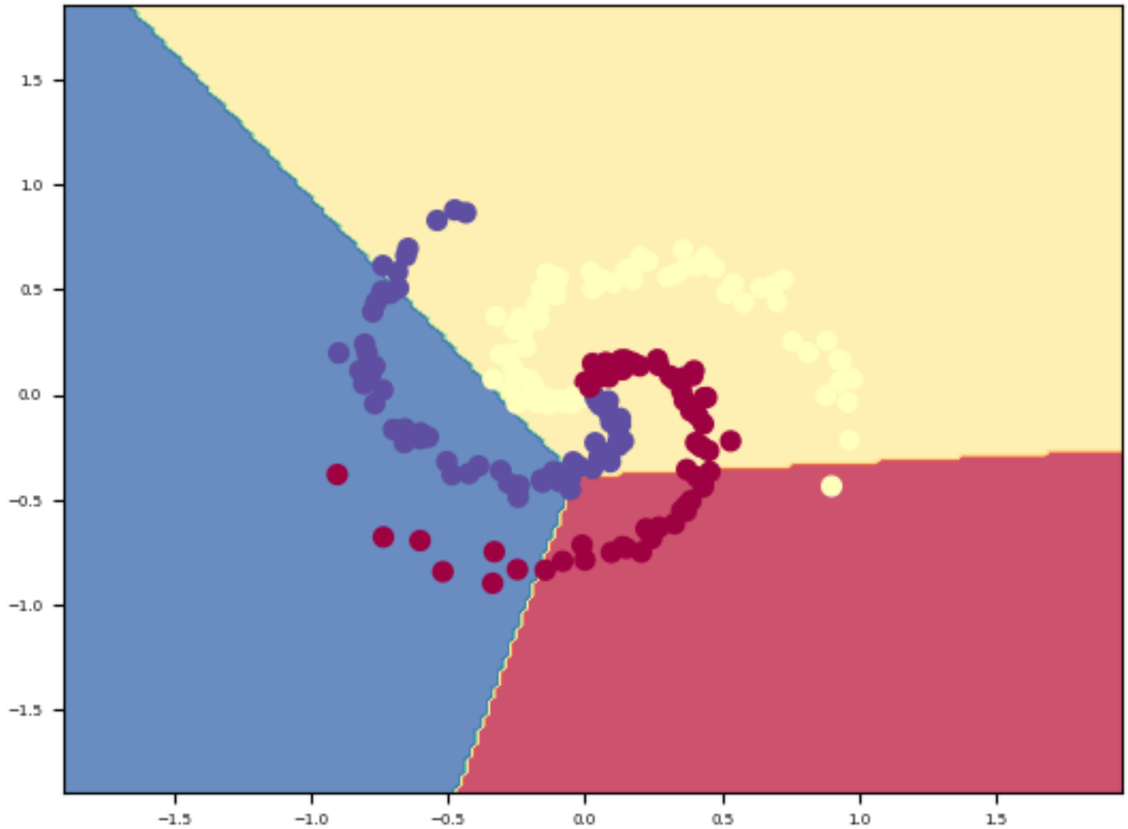
```python
# output
print('Leave One Out Cross-validation results\n'
      f'Mean Training Accuracy: {mean_cv_training}\n'
      f'Mean Validation Accuracy: {mean_cv_validation}\n'
      f'Mean Loss: {mean_cv_loss}')
```

```
Leave One Out Cross-validation results
Mean Training Accuracy: 0.5842592592592593
Mean Validation Accuracy: 0.5666666666666667
Mean Loss: 1.088892209841891
```

In [ ]:

```python
print("Best Model:")
best_classifier.summary()

best_classifier.show_classifier()
```

```
Best Model:
Model summary:

W:
[[ 0.02072945  0.01775439 -0.02730493]
 [-0.02166191  0.02486712 -0.00782442]]

B:
[[0.07922916 0.09721065 0.08285023]]

Learning Rate: 0.001
l2 Regularization: 0.001
Epochs: 300

Training Accuracy: 0.6203703703703703
Model Loss: 1.0890753164637825
```



Performance Sensitivity: Learning Rate and Gradient Descent

To evaluate model performance sensitivty on various hyperparameters, we modify the above block of code to perform KFold cross validation on models of varying learning rate and epoch size.

Learning Rate Sensitivity

To test the sensitivity of the Linear Classifier, we iteratively test learning rates between `1e−1` to `1e−5` with each instance evaluated through LOO cross-validation.

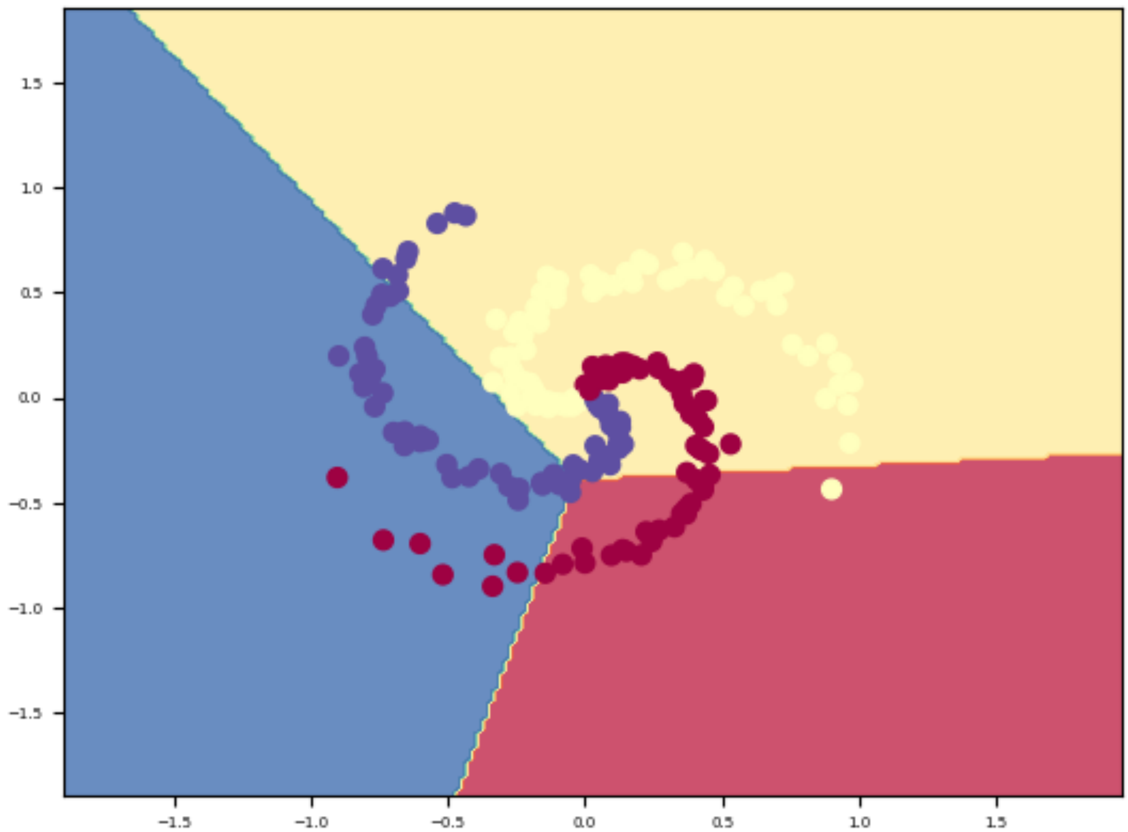
In [ ]:
```
# initialize parameters for the linear classifier iterations
learning_rates = np.geomspace(1e-1, 1e-5, 5)
epochs = 300
l2 = 1e-3

# instantiate KFold object
kf = KFold(n_splits = 10, shuffle = True, random_state=0)

# initialize tracking objects for each linear classifier
model_training_accuraccies = []
model_validation_accuraccies = []
model_loss = []

# best model tracking
best_accuracy = 0.0
```

```
# run first loop to iterate through learning rates
for lr in learning_rates:
    # tracking objects
    training_history = []
    validation_history = []
    loss_history = []

    # begin CV loop
    for i, (train_index, validation_index) in enumerate(kf.split(X_train)):

        # instatiate new LinearClassifier object with cv split data
        cv_classifier = LinearClassifier(X_train[train_index], y_train[trair
        cv_classifier.start(verbose = False)

        # obtain tracking metrics
        training_accuracy = cv_classifier.eval(X_train[train_index], y_trair
        validation_accuracy = cv_classifier.eval(X_train[validation_index],
        loss = cv_classifier.loss

        # update best model
        if training_accuracy > best_accuracy:
            best_accuracy = training_accuracy
            best_classifier = cv_classifier

        # append metrics
        training_history.append(training_accuracy)
        validation_history.append(validation_accuracy)
        loss_history.append(loss)

    # return mean values
    mean_cv_training = np.mean(training_history)
    mean_cv_validation = np.mean(validation_history)
    mean_cv_loss = np.mean(loss_history)

    # append mean metrics to overall tracking
    model_training_accuraccies.append(mean_cv_training)
    model_validation_accuraccies.append(mean_cv_validation)
    model_loss.append(mean_cv_loss)
```

Show best model:

In [ ]:
```
print("Best Model:")
best_classifier.summary()

best_classifier.show_classifier()
```

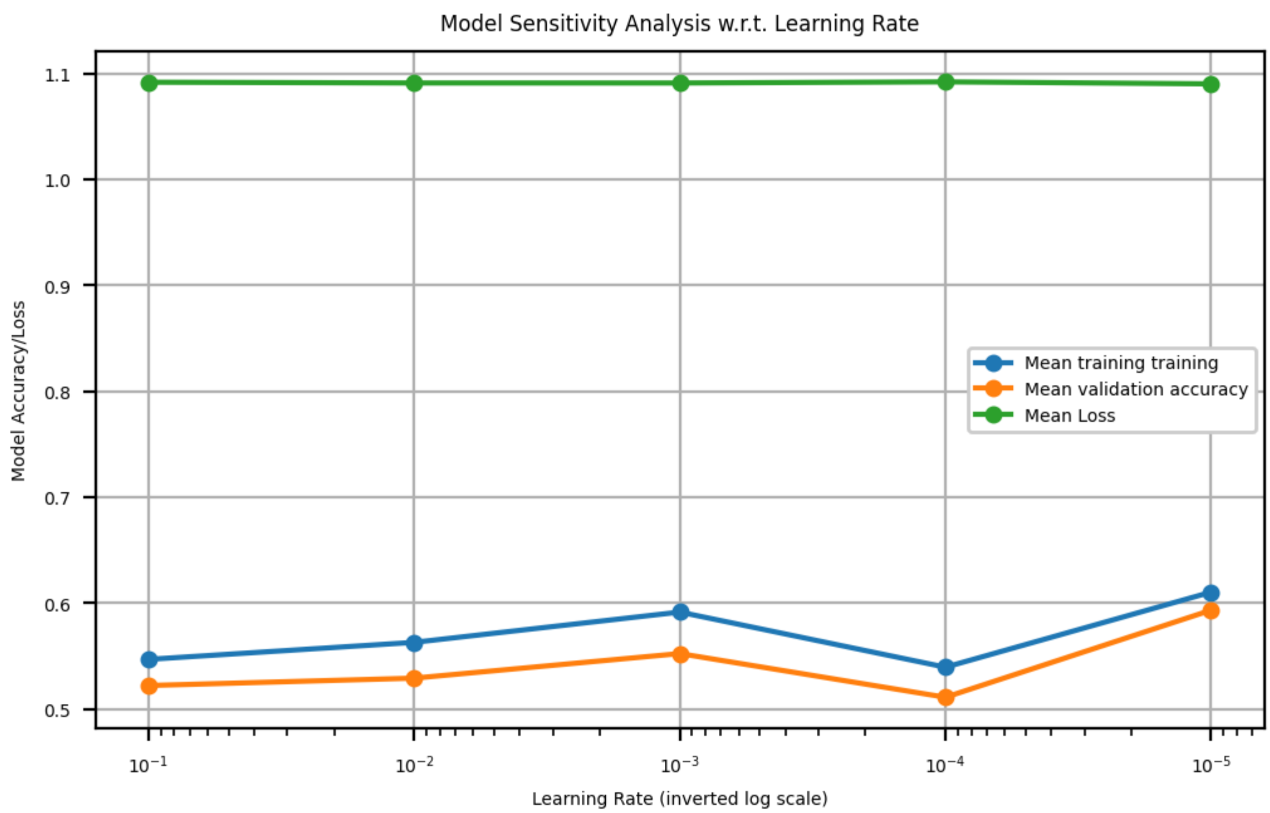
```
Best Model:
Model summary:

W:
[[ 0.02072945  0.01775439 -0.02730493]
 [-0.02166191  0.02486712 -0.00782442]]

B:
[[0.07922916 0.09721065 0.08285023]]

Learning Rate: 0.001
l2 Regularization: 0.001
Epochs: 300

Training Accuracy: 0.6203703703703703
Model Loss: 1.0890753164637825
```



We then plot the pertinent metrics for sensitivity analysis.

In [ ]:
```
# plot learning rate sensitivity
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(learning_rates,
        model_training_accuraccies,
        label = r'Mean training training',
        marker = 'o',
        markersize = 4)

ax.plot(learning_rates,
```

```
        model_validation_accuraccies,
        label = r'Mean validation accuracy',
        marker = 'o',
        markersize = 4)

ax.plot(learning_rates,
        model_loss,
        label = r'Mean Loss',
        marker = 'o',
        markersize = 4)

ax.set_title("Model Sensitivity Analysis w.r.t. Learning Rate")
ax.set_xlabel('Learning Rate (inverted log scale)')
ax.set_xscale("log")
ax.invert_xaxis()
ax.set_ylabel('Model Accuracy/Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```



The above plot illustrates the sensitivity of our Linear Classifier Model w.r.t. different learning rates, evaluated on the `X_train` training data and cross validated with 10 splits. A learning rate of `1e-3` yielded the best training and validation accuracy and subsequent smaller learning rates degraded overall model accuracy.

Gradient Descent Iteration Sensitivity

Similarly, we perform the same task for the number of gradient descent iterations. The previous best learning rate `1e-3` remains constant in lieu of other hyperparameter tuning schemas such as *grid search* or *random search*.

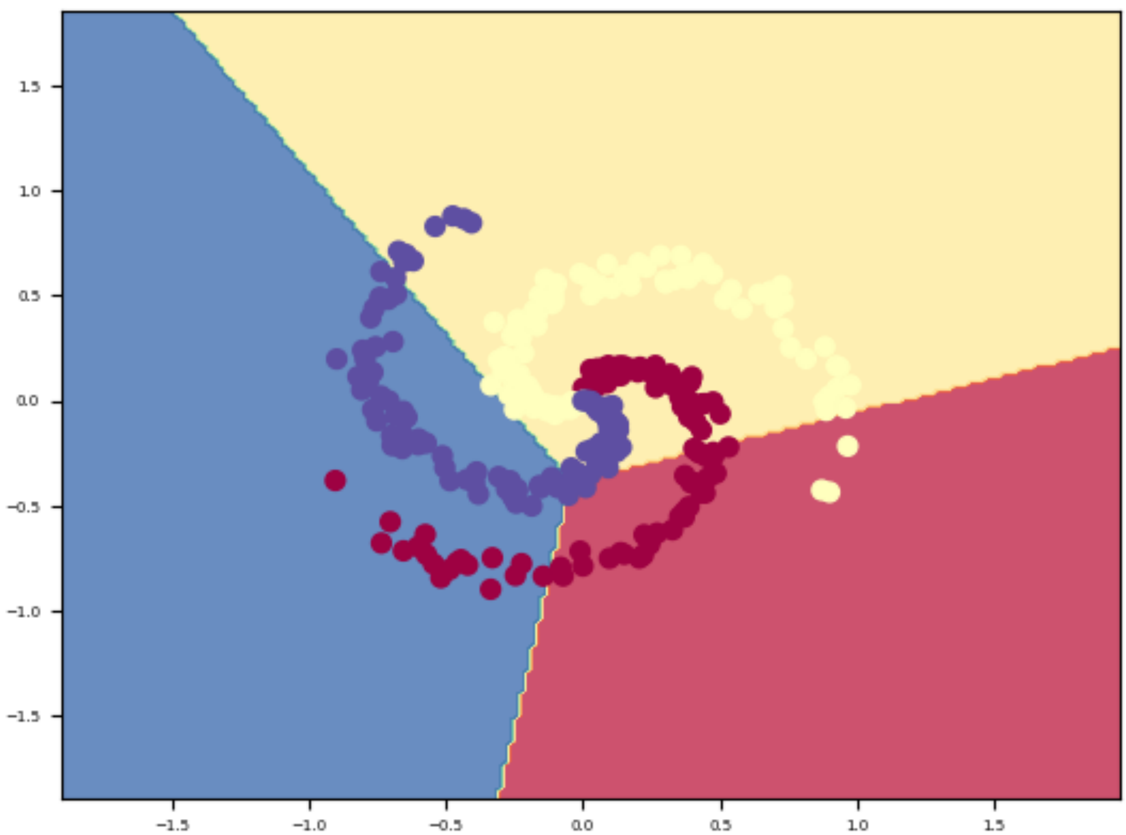
In [ ]:
```
# initialize parameters for the linear classifier iterations
lr = 1e-3
various_epochs = np.linspace(100, 1000, 7, dtype=int)
l2 = 1e-3

# instantiate LeaveOneOut object for CV
kf = KFold(n_splits = 10, shuffle = True, random_state=1)

# initialize tracking objects for each linear classifier
model_training_accuraccies = []
model_validation_accuraccies = []
model_loss = []

# best model tracking
best_accuracy = 0.0

# run first loop to iterate through learning rates
for epochs in various_epochs:
    # tracking objects
    training_history = []
    validation_history = []
    loss_history = []

    # begin CV loop
    for i, (train_index, validation_index) in enumerate(kf.split(X_train)):

        # instatiate new LinearClassifier object with cv split data
        cv_classifier = LinearClassifier(X_train[train_index], y_train[trair
        cv_classifier.start(verbose = False)

        # obtain tracking metrics
        training_accuracy = cv_classifier.eval(X_train[train_index], y_trair
        validation_accuracy = cv_classifier.eval(X_train[validation_index],
        loss = cv_classifier.loss

        # update best model
        if training_accuracy > best_accuracy:
            best_accuracy = training_accuracy
            best_classifier = cv_classifier

        # append metrics
        training_history.append(training_accuracy)
        validation_history.append(validation_accuracy)
        loss_history.append(loss)

    # return mean values
    mean_cv_training = np.mean(training_history)
    mean_cv_validation = np.mean(validation_history)
    mean_cv_loss = np.mean(loss_history)
```

```
    # append mean metrics to overall tracking
    model_training_accuraccies.append(mean_cv_training)
    model_validation_accuraccies.append(mean_cv_validation)
    model_loss.append(mean_cv_loss)
```

In [ ]:

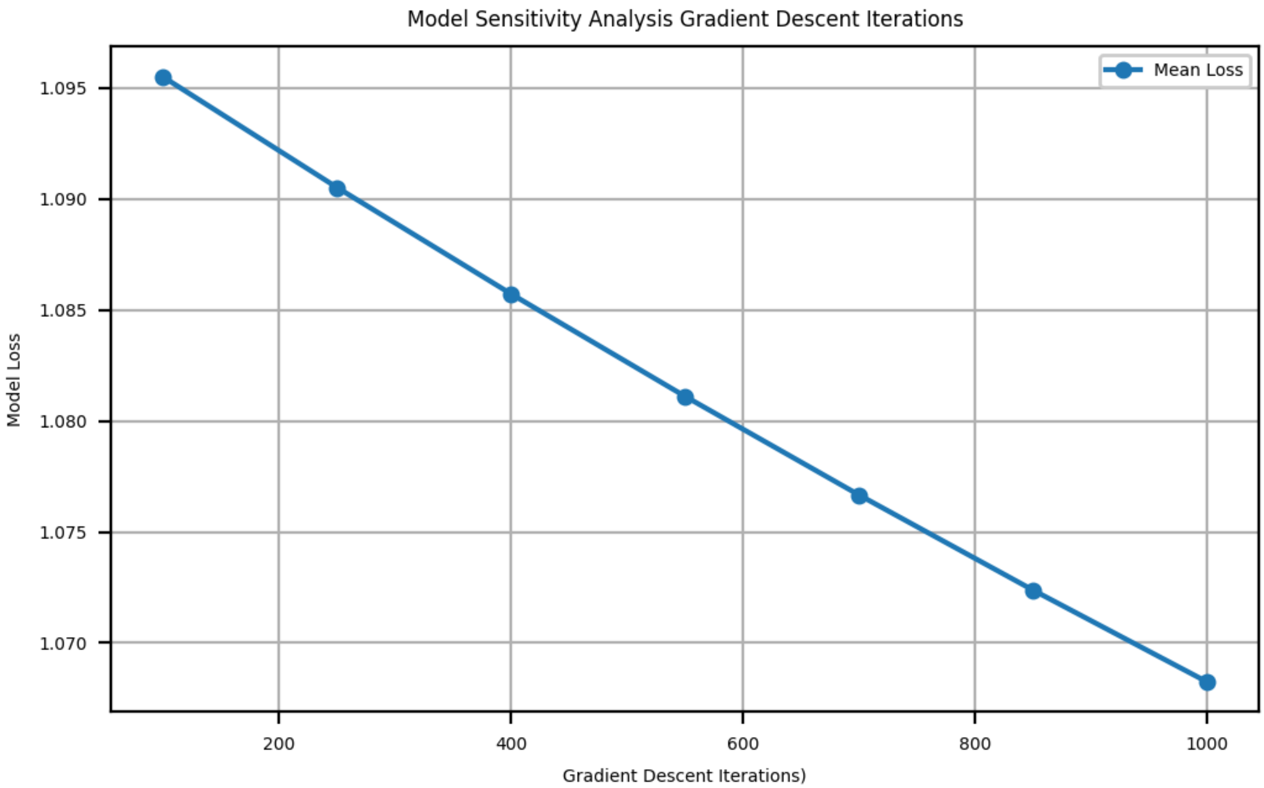
```
print("Best Model:")
best_classifier.summary()

best_classifier.show_classifier()
```

```
Best Model:
Model summary:

W:
[[ 0.01054313  0.00635041 -0.00866427]
 [-0.00760066  0.00546519 -0.0042355 ]]

B:
[[0.02999379 0.03483221 0.03040347]]

Learning Rate: 0.001
l2 Regularization: 0.001
Epochs: 100

Training Accuracy: 0.6435185185185185
Model Loss: 1.0955903096970332
```



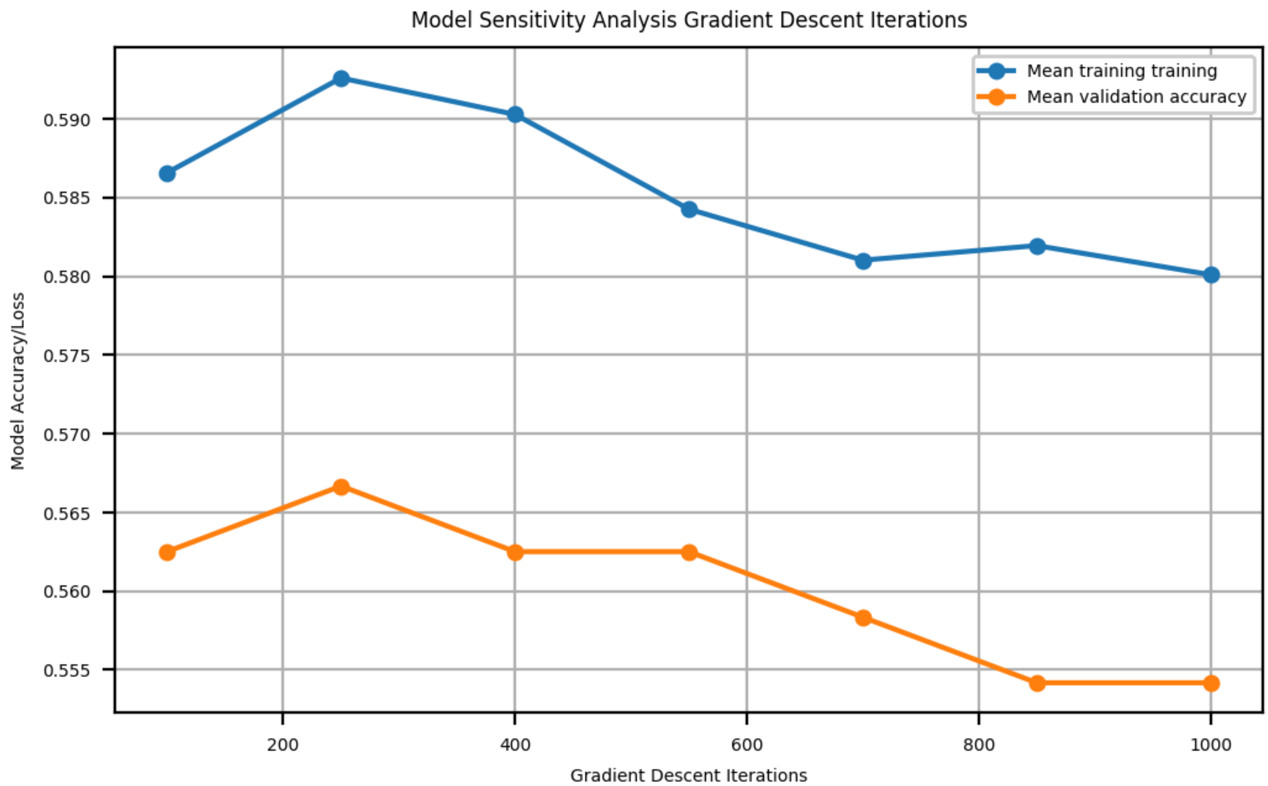
We next plot training/validation accuracy and model loss on seperate plots.

In [ ]:
```
# plot learning rate sensitivity
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(various_epochs,
        model_loss,
        label = r'Mean Loss',
        marker = 'o',
        markersize = 4)


ax.set_title("Model Sensitivity Analysis Gradient Descent Iterations")
ax.set_xlabel('Gradient Descent Iterations)')
ax.set_ylabel('Model Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```



In [ ]:
```
# plot learning rate sensitivity
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(various_epochs,
        model_training_accuraccies,
        label = r'Mean training training',
        marker = 'o',
        markersize = 4)

ax.plot(various_epochs,
        model_validation_accuraccies,
        label = r'Mean validation accuracy',
        marker = 'o',
```

```
        markersize = 4)

ax.set_title("Model Sensitivity Analysis Gradient Descent Iterations")
ax.set_xlabel('Gradient Descent Iterations')
ax.set_ylabel('Model Accuracy/Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```



Change in model sensitivity w.r.t. gradient descent iterations shows an increase in mean validation acuracy up to 250 iterations. Past this point validation and training accuracy appears to degrade. For this reason the number of iterations `250` is the selected to be the best candidate, offering the best training and validation accuracy without unnecessary computational cost and loss of accuracy.

`l2` Regularization Sensitivity

We will retain the best hyperparameters for the linear mode `lr = 1-e3` and `epochs = 250` for this next sensitivy analysis, while testing the effects of different `l2` values including `0.0` (no `l2` regularization).

In [ ]:

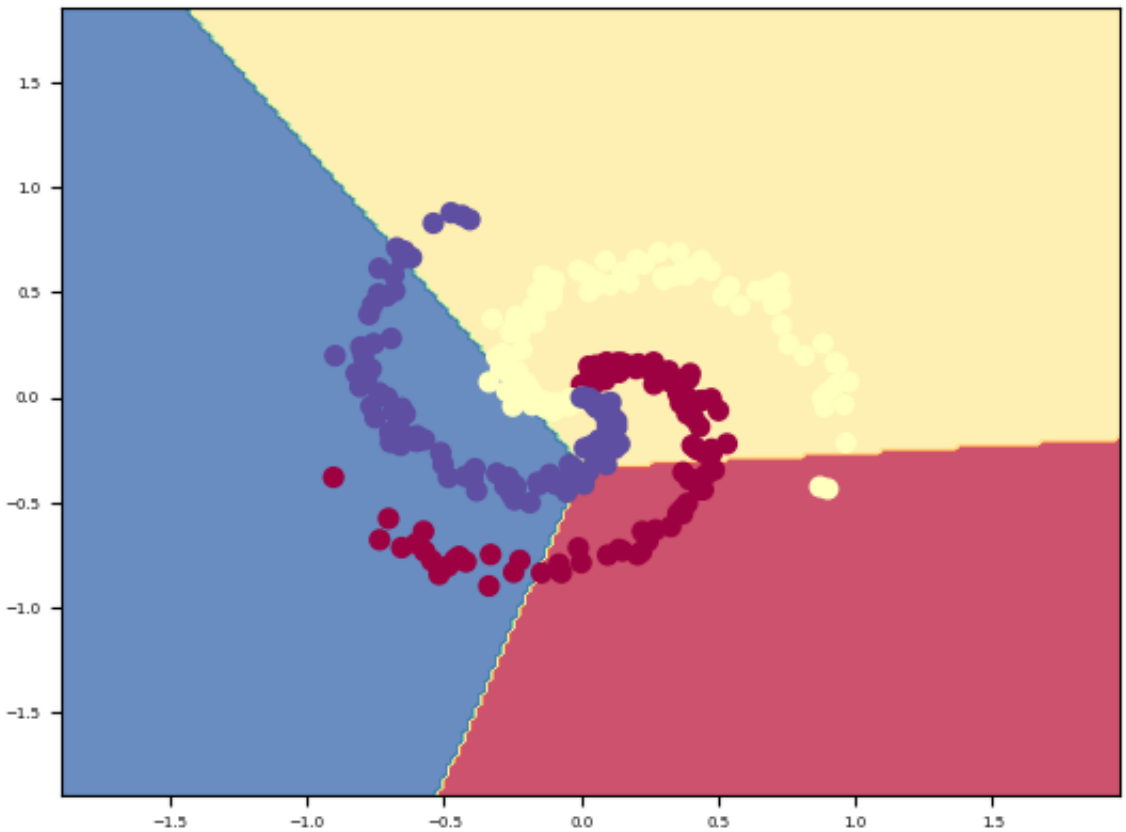
```
# initialize parameters for the linear classifier iterations
lr = 1e-3
epochs = 250
l2_values = np.append(np.geomspace(1e-1, 1e-5, 5), [0], axis = 0)

# instantiate LeaveOneOut object for CV
```

```python
kf = KFold(n_splits = 10, shuffle = True, random_state=2)

# initialize tracking objects for each linear classifier
model_training_accuraccies = []
model_validation_accuraccies = []
model_loss = []

# best model tracking
best_accuracy = 0.0

# run first loop to iterate through learning rates
for l2 in l2_values:
    # tracking objects
    training_history = []
    validation_history = []
    loss_history = []

    # begin CV loop
    for i, (train_index, validation_index) in enumerate(kf.split(X_train)):

        # instatiate new LinearClassifier object with cv split data
        cv_classifier = LinearClassifier(X_train[train_index], y_train[trair
        cv_classifier.start(verbose = False)

        # obtain tracking metrics
        training_accuracy = cv_classifier.eval(X_train[train_index], y_trair
        validation_accuracy = cv_classifier.eval(X_train[validation_index],
        loss = cv_classifier.loss

        # update best model
        if training_accuracy > best_accuracy:
            best_accuracy = training_accuracy
            best_classifier = cv_classifier

        # append metrics
        training_history.append(training_accuracy)
        validation_history.append(validation_accuracy)
        loss_history.append(loss)

    # return mean values
    mean_cv_training = np.mean(training_history)
    mean_cv_validation = np.mean(validation_history)
    mean_cv_loss = np.mean(loss_history)

    # append mean metrics to overall tracking
    model_training_accuraccies.append(mean_cv_training)
    model_validation_accuraccies.append(mean_cv_validation)
    model_loss.append(mean_cv_loss)
```

In [ ]:

```python
print("Best Model:")
best_classifier.summary()

best_classifier.show_classifier()
```

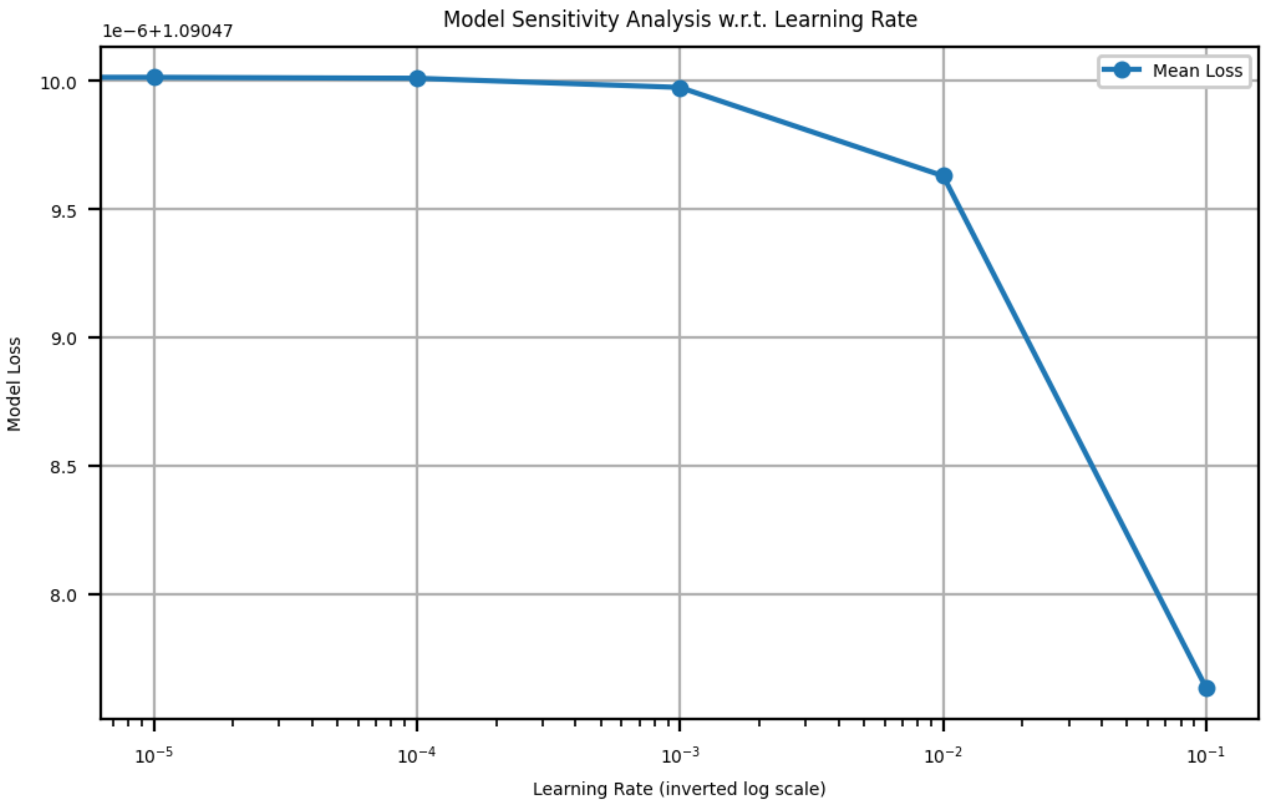
```
Best Model:
Model summary:

W:
[[ 0.01741403  0.01463414 -0.02377698]
 [-0.01890681  0.02058674 -0.00483719]]

B:
[[0.0676665  0.08093004 0.07273069]]

Learning Rate: 0.001
l2 Regularization: 0.1
Epochs: 250

Training Accuracy: 0.625
Model Loss: 1.090378951582681
```



In [ ]:

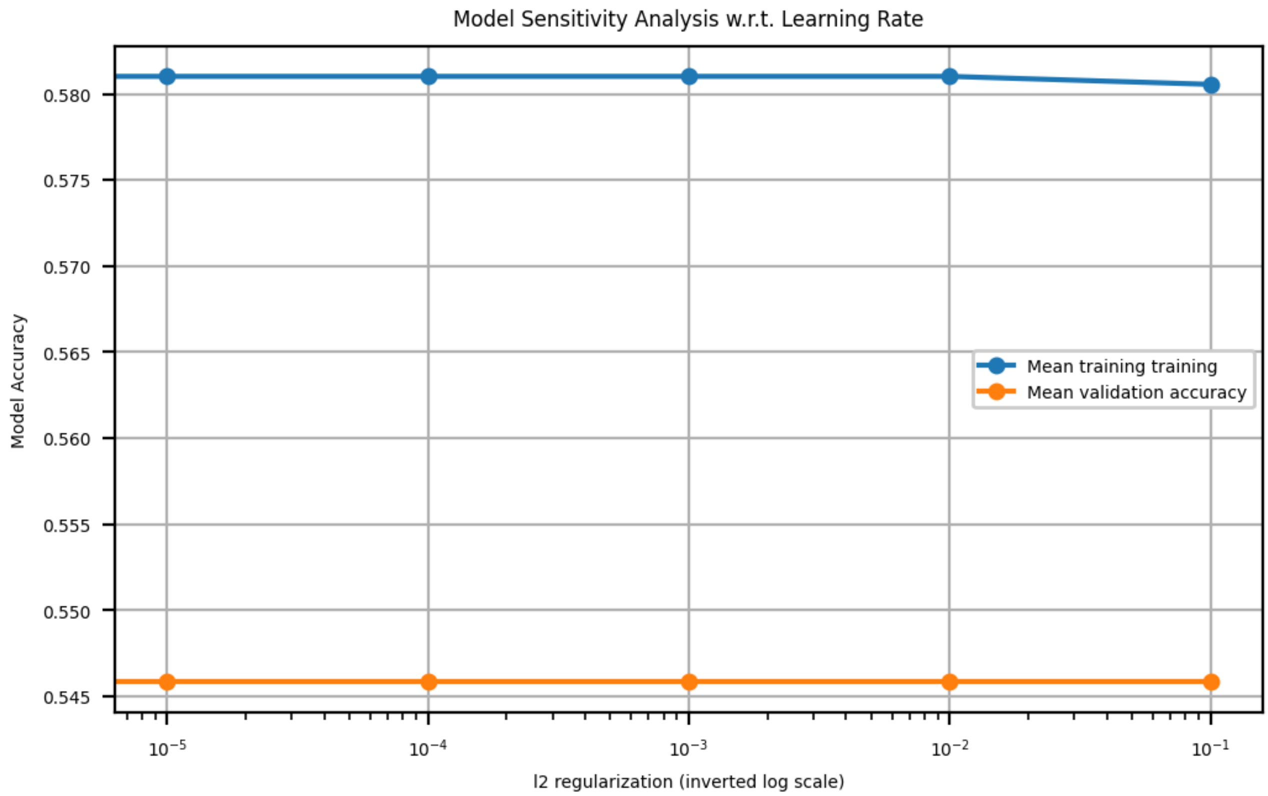
```
# plot learning rate sensitivity
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(l2_values,
        model_loss,
        label = r'Mean Loss',
        marker = 'o',
        markersize = 4)

ax.set_title("Model Sensitivity Analysis w.r.t. Learning Rate")
ax.set_xlabel('Learning Rate (inverted log scale)')
ax.set_xscale("log")
#ax.invert_xaxis()
```

```
ax.set_ylabel('Model Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```



In [ ]:
```
# plot learning rate sensitivity
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(l2_values,
        model_training_accuraccies,
        label = r'Mean training training',
        marker = 'o',
        markersize = 4)

ax.plot(l2_values,
        model_validation_accuraccies,
        label = r'Mean validation accuracy',
        marker = 'o',
        markersize = 4)

'''
ax.plot(l2_values,
        model_loss,
        label = r'Mean Loss')
'''

ax.set_title("Model Sensitivity Analysis w.r.t. Learning Rate")
ax.set_xlabel('l2 regularization (inverted log scale)')
ax.set_xscale("log")
#ax.invert_xaxis()
```

```python
ax.set_ylabel('Model Accuracy')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)
ax.grid(True)
```



Varying `l2` regularization prameters appears to have minimal effects on the linear classifier model. For values of $l2 \in [0, 10^{-5}]$ there was no change in mean training and validation accuracy. FOr this reason, `l2` is selected to be `0.0` as an input parameter in order to minimize computational cost.

Train Test Split Sensitivity

The above sensitivity analysis returned the following optimal hyperparameters for the linear classifier model:

- Learning Rate: `lr = 1e-3`
- Gradient Descent Iterations: `epochs = 250`
- Regularization: `l2 = 0.0`

We then use these hyperparameters to analyze the sensitivity of the model w.r.t. train/test splits.

In [ ]:
```python
# specifiy train test splits in a list and iterate through splits
splits = [0.1, 0.2, 0.3, 0.4, 0.5]

# intialize best hyperparameters for linear classifier
```

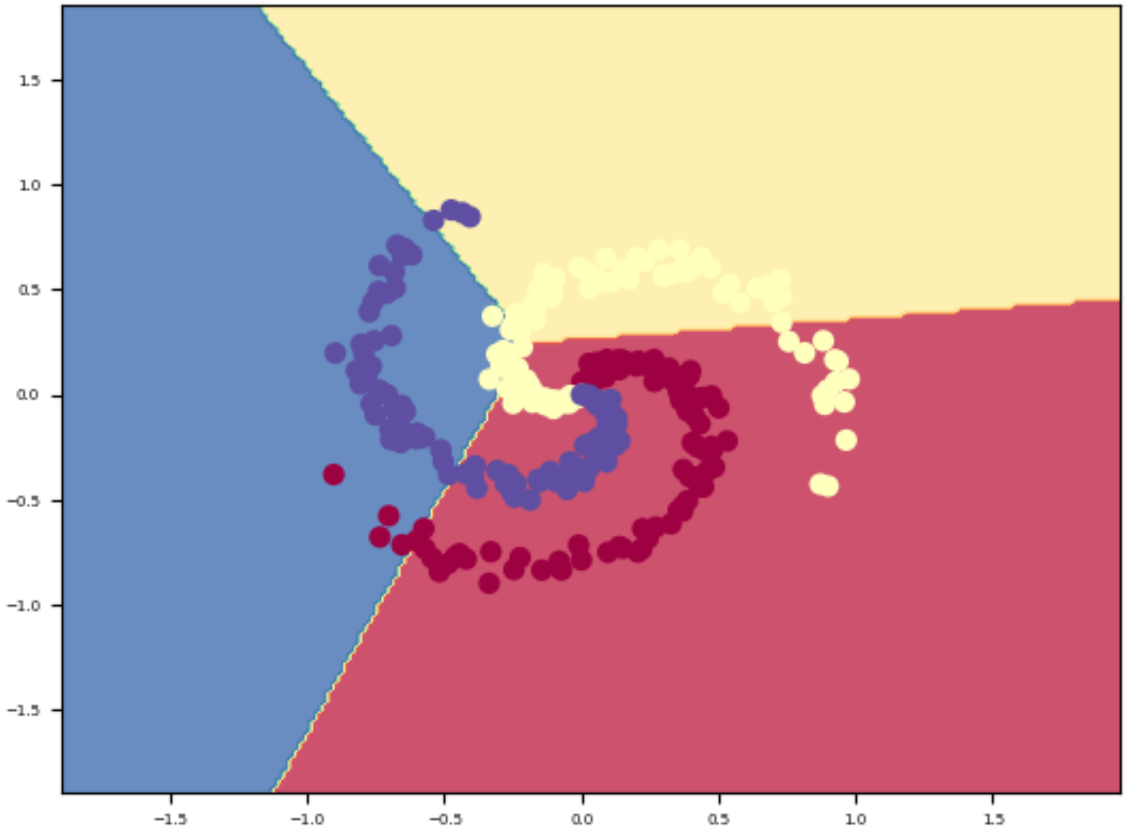
```python
# l2 regularliztion = 0.0 default for class constructor
lr = 1e-3
epochs = 250

# reload X and y data from the pickle file
X = pickle.load(open('Misc_files/dataX.pickle','rb'))
y = pickle.load(open('Misc_files/dataY.pickle','rb'))

# random seed for reproducibility
np.random.seed(576)

# get Array W of shape (2, 3)
W = 0.01*np.random.randn(X.shape[1],max(y)+1)

# get Array K of shape (1, 3)
b = np.zeros((1, max(y)+1))

# initialize tracking objects for each linear classifier
model_training_accuraccies = []
model_validation_accuraccies = []
model_testing_accuracies = []
model_loss = []

# best model tracking
best_accuracy = 0.0

# loop through splits using 10 fold cross-validation
for split in splits:

    # generate new train test split for each iteration, adjust random state
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=spli

    # tracking objects
    training_history = []
    validation_history = []
    test_history = []
    loss_history = []

    # begin CV loop
    for i, (train_index, validation_index) in enumerate(kf.split(X_train)):

        # instatiate new LinearClassifier object with cv split data
        cv_classifier = LinearClassifier(X_train[train_index], y_train[train
        cv_classifier.start(verbose = False)

        # obtain tracking metrics
        training_accuracy = cv_classifier.eval(X_train[train_index], y_train
        validation_accuracy = cv_classifier.eval(X_train[validation_index],
        testing_accuracy = cv_classifier.eval(X_test, y_test)
        loss = cv_classifier.loss

        # update best model
        if training_accuracy > best_accuracy:
```

```
            best_accuracy = training_accuracy
            best_classifier = cv_classifier

        # append metrics
        training_history.append(training_accuracy)
        validation_history.append(validation_accuracy)
        test_history.append(testing_accuracy)
        loss_history.append(loss)

    # return mean values
    mean_cv_training = np.mean(training_history)
    mean_cv_validation = np.mean(validation_history)
    mean_cv_testing = np.mean(test_history)
    mean_cv_loss = np.mean(loss_history)

    # append mean metrics to overall tracking
    model_training_accuraccies.append(mean_cv_training)
    model_validation_accuraccies.append(mean_cv_validation)
    model_testing_accuracies.append(mean_cv_testing)
    model_loss.append(mean_cv_loss)
```

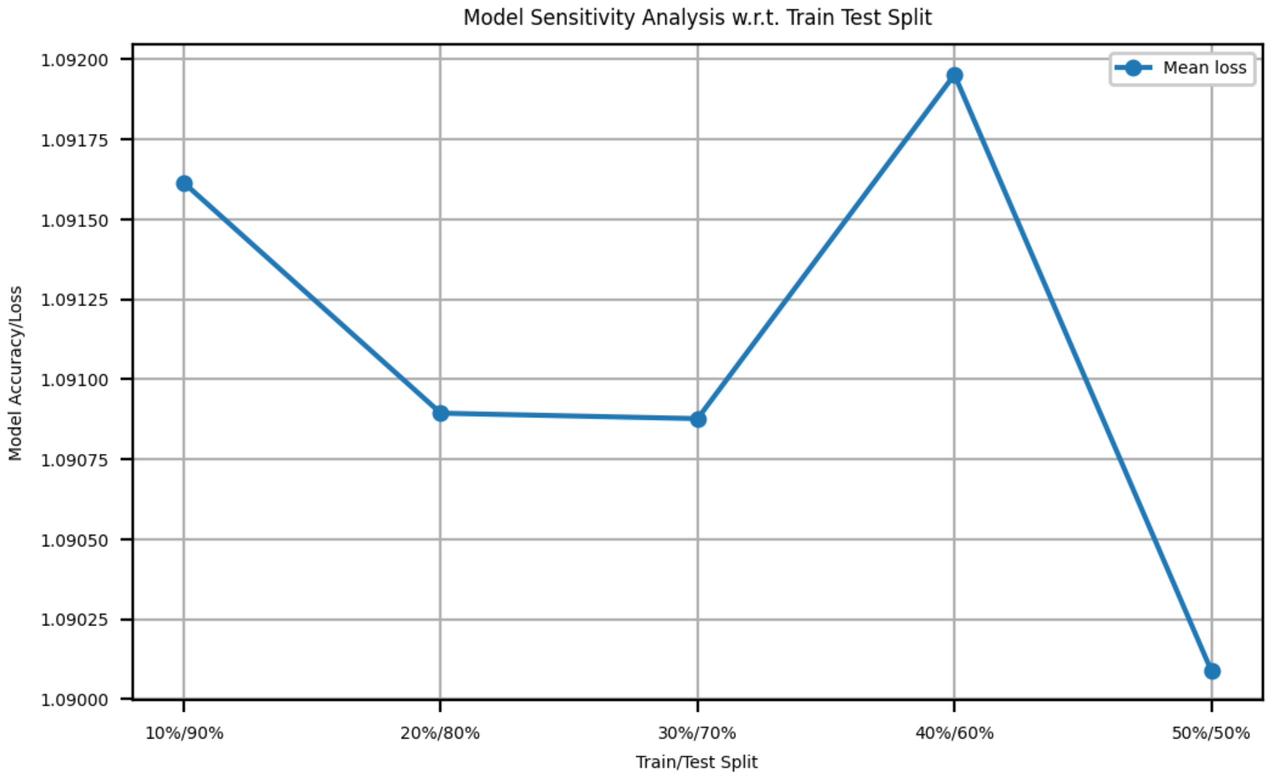
In [ ]:

```
print("Best Model:")
best_classifier.summary()

best_classifier.show_classifier()
```

```
Best Model:
Model summary:

W:
[[ 0.01880898  0.01470664 -0.02396573]
 [-0.02503941  0.01722767 -0.00653695]]

B:
[[0.081869   0.07055378 0.06879078]]

Learning Rate: 0.001
l2 Regularization: 0.001
Epochs: 250

Training Accuracy: 0.6296296296296297
Model Loss: 1.0898555458957988
```



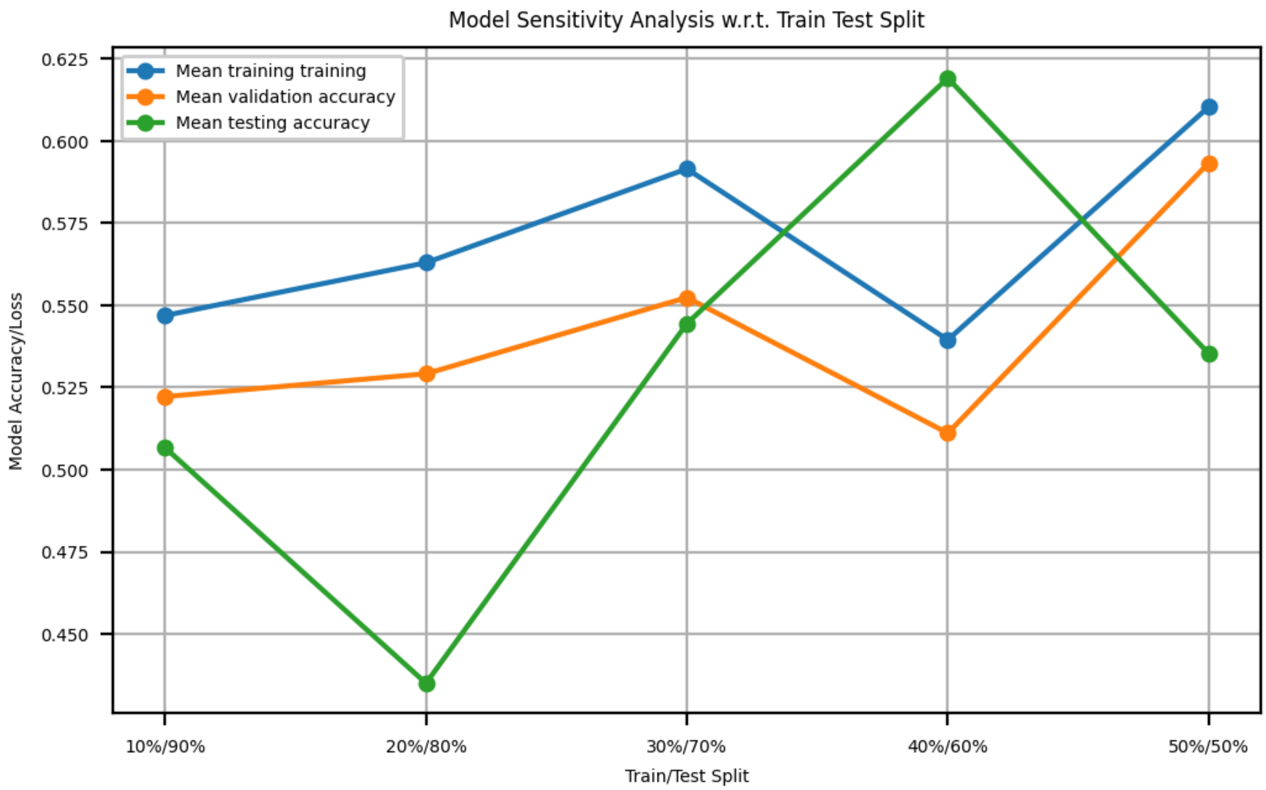
In [ ]:
```
# plot train/test split sensitivity
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(splits,
        model_loss,
        label = r'Mean loss',
        marker = 'o',
        markersize = 4)

ax.set_title("Model Sensitivity Analysis w.r.t. Train Test Split")
ax.set_xlabel('Train/Test Split')
ax.set_ylabel('Model Accuracy/Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)

ax.set_xticks(splits)  # Set x-axis tick positions
ax.set_xticklabels([f'{int(split * 100)}%/{100-int(split * 100)}%' for split

ax.grid(True)
```



In [ ]:

```
# plot train/test split sensitivity
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

ax.plot(splits,
        model_training_accuraccies,
        label = r'Mean training training',
        marker = 'o',
        markersize = 4)

ax.plot(splits,
        model_validation_accuraccies,
        label = r'Mean validation accuracy',
        marker = 'o',
        markersize = 4)

ax.plot(splits,
        model_testing_accuracies,
        label = r'Mean testing accuracy',
        marker = 'o',
        markersize = 4)

ax.set_title("Model Sensitivity Analysis w.r.t. Train Test Split")
ax.set_xlabel('Train/Test Split')
ax.set_ylabel('Model Accuracy/Loss')
ax.legend()
ax.legend().get_frame().set_alpha(1.0)

ax.set_xticks(splits)  # Set x-axis tick positions
ax.set_xticklabels([f'{int(split * 100)}%/{100-int(split * 100)}%' for split
```

```
ax.grid(True)
```



5. Feed Forward Neural Networks

Data Loading and Preprocessing

We first download the Fashion MNIST dataset through their example code block:

```
In [ ]: # import tensorflow and keras
import tensorflow as tf
import keras

# load data and obtain description for labels
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_
assert x_train.shape == (60000, 28, 28)
assert x_test.shape == (10000, 28, 28)
assert y_train.shape == (60000,)
assert y_test.shape == (10000,)

# label description
y_description = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sar
```

Per the keras website:

"This is a dataset of 60,000 28x28 grayscale images of 10 fashion categories, along with a test set of 10,000 images. This dataset can be used as a drop-in replacement for

MNIST."

The classes have the following labels and desciptions:

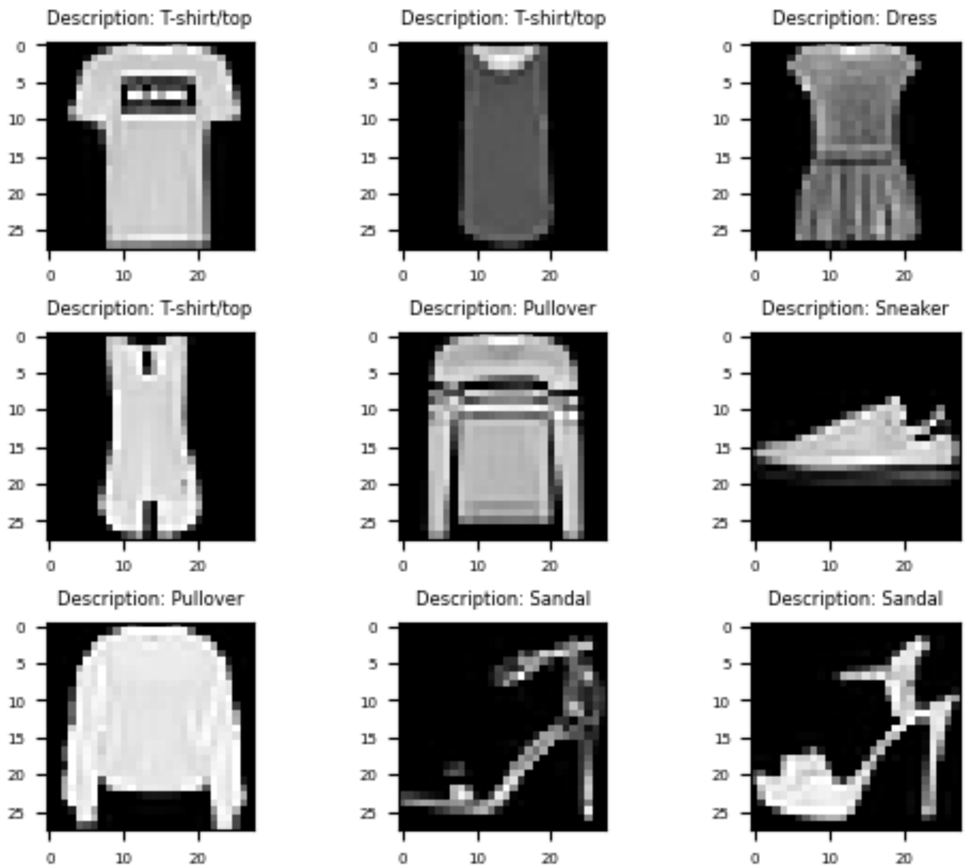
Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

We first plot a selection of the training data for visual analysis. This website (geeksforgeeks.org) was used for reference.

In [ ]:

```
# obtain data summary
for i in range(1,10):

    plt.subplot(3,3,i)
    plt.title("Description: "+y_description[y_train[i]])

    plt.imshow(x_train[i], cmap = plt.get_cmap('gray'))

plt.subplots_adjust(wspace=0, hspace=0.4)

print(f"Training Data Shape: {x_train.shape}\n"+
      f"Testing Data Shape: {x_test.shape}")
```

```
Training Data Shape: (60000, 28, 28)
Testing Data Shape: (10000, 28, 28)
```



We note that the input tensor contains values between 0 and 255, standard for grayscale images, so the data is normalized for model input.

In [ ]:
```
# rescale tensor values between 0 and 1
x_train_model = x_train/255.
x_test_model = x_test/255.
```

Model Construction and Architecture

We then construct the 2-layer feedforward neural network using the keras `Sequential()` class. We build the `model_arch()` function in such a way that we can specify the `units` and `activation` function of the resultant `Sequential()` model:

In [ ]:
```
from keras import layers

# clear model
keras.backend.clear_session()

# construct model function for implementation
def model_arch(units = 64, activation = "ReLU"):

    # clear and instantiated models
    keras.backend.clear_session()
```

```
    # instantiate model
    model = keras.Sequential(name = f'units-{units}_activation-{activation}'

    # add model input
    model.add(keras.Input(shape = (28,28)))

    # flatten input of shape (28,28) into a vector
    model.add(layers.Flatten())

    # add first hidden layer, a naive number of units has been selected as 6
    model.add(layers.Dense(name = "hidden_layer", units = units))

    if activation == "ReLU":
        model.add(layers.ReLU())
    elif activation == "LeakyReLU":
        model.add(layers.LeakyReLU())


    # output layer corresponding to the 10 object labels
    model.add(layers.Dense(name = "output_layer", units = 10, activation = "

    # the input shape of our data is 28x28,
    return model
```

Next we instatiate a `model_arch()` instance and inspect its architecture with the `summary()` method. Note that we can also see the activation function for the hidden units:

In [ ]:

```
model = model_arch(64,"ReLU")
model.summary()
```

```
Model: "units-64_activation-ReLU"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 hidden_layer (Dense)        (None, 64)                50240

 re_lu (ReLU)                (None, 64)                0

 output_layer (Dense)        (None, 10)                650

=================================================================
Total params: 50890 (198.79 KB)
Trainable params: 50890 (198.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Model Compilation and Training

Training of the model begins following the assignment of an optimizer, loss function, and accuracy metric using the `compile()` method.

```
In [ ]: model.compile(optimizer=keras.optimizers.legacy.Adam(learning_rate=1e-3),
                      loss='sparse_categorical_crossentropy',
                      metrics=['sparse_categorical_accuracy'],
                      )
```

Finally, the model is fit to the training data using the `fit()` method where we specify the number of parameters, a validation split, and random shuffling of the training data. Model fitting results in the creation of a `history` object which we can subsequently use to evaluate the model.

```
In [ ]: history = model.fit(
            x_train_model.astype(np.float32), y_train,
            epochs=60,
            validation_split=0.2,
            shuffle = True
        )
```

```
Epoch 1/60
1500/1500 [==============================] - 1s 720us/step - loss: 0.5359 -
sparse_categorical_accuracy: 0.8142 - val_loss: 0.4989 - val_sparse_categori
cal_accuracy: 0.8320
Epoch 2/60
1500/1500 [==============================] - 1s 676us/step - loss: 0.4027 -
sparse_categorical_accuracy: 0.8577 - val_loss: 0.4085 - val_sparse_categori
cal_accuracy: 0.8503
Epoch 3/60
1500/1500 [==============================] - 1s 660us/step - loss: 0.3617 -
sparse_categorical_accuracy: 0.8694 - val_loss: 0.3838 - val_sparse_categori
cal_accuracy: 0.8646
Epoch 4/60
1500/1500 [==============================] - 1s 607us/step - loss: 0.3361 -
sparse_categorical_accuracy: 0.8770 - val_loss: 0.3652 - val_sparse_categori
cal_accuracy: 0.8695
Epoch 5/60
1500/1500 [==============================] - 1s 671us/step - loss: 0.3165 -
sparse_categorical_accuracy: 0.8841 - val_loss: 0.3572 - val_sparse_categori
cal_accuracy: 0.8717
Epoch 6/60
1500/1500 [==============================] - 1s 677us/step - loss: 0.3014 -
sparse_categorical_accuracy: 0.8897 - val_loss: 0.3875 - val_sparse_categori
cal_accuracy: 0.8575
Epoch 7/60
1500/1500 [==============================] - 1s 656us/step - loss: 0.2894 -
sparse_categorical_accuracy: 0.8925 - val_loss: 0.3370 - val_sparse_categori
cal_accuracy: 0.8832
Epoch 8/60
1500/1500 [==============================] - 1s 656us/step - loss: 0.2774 -
sparse_categorical_accuracy: 0.8968 - val_loss: 0.3674 - val_sparse_categori
cal_accuracy: 0.8718
Epoch 9/60
```

```
1500/1500 [==============================] - 1s 654us/step - loss: 0.2673 -
sparse_categorical_accuracy: 0.9013 - val_loss: 0.3419 - val_sparse_categori
cal_accuracy: 0.8783
Epoch 10/60
1500/1500 [==============================] - 1s 663us/step - loss: 0.2586 -
sparse_categorical_accuracy: 0.9041 - val_loss: 0.3378 - val_sparse_categori
cal_accuracy: 0.8814
Epoch 11/60
1500/1500 [==============================] - 1s 649us/step - loss: 0.2519 -
sparse_categorical_accuracy: 0.9068 - val_loss: 0.3338 - val_sparse_categori
cal_accuracy: 0.8838
Epoch 12/60
1500/1500 [==============================] - 1s 665us/step - loss: 0.2415 -
sparse_categorical_accuracy: 0.9099 - val_loss: 0.3357 - val_sparse_categori
cal_accuracy: 0.8838
Epoch 13/60
1500/1500 [==============================] - 1s 651us/step - loss: 0.2360 -
sparse_categorical_accuracy: 0.9121 - val_loss: 0.3290 - val_sparse_categori
cal_accuracy: 0.8857
Epoch 14/60
1500/1500 [==============================] - 1s 688us/step - loss: 0.2318 -
sparse_categorical_accuracy: 0.9143 - val_loss: 0.3333 - val_sparse_categori
cal_accuracy: 0.8861
Epoch 15/60
1500/1500 [==============================] - 1s 653us/step - loss: 0.2228 -
sparse_categorical_accuracy: 0.9165 - val_loss: 0.3383 - val_sparse_categori
cal_accuracy: 0.8842
Epoch 16/60
1500/1500 [==============================] - 1s 671us/step - loss: 0.2202 -
sparse_categorical_accuracy: 0.9175 - val_loss: 0.3246 - val_sparse_categori
cal_accuracy: 0.8913
Epoch 17/60
1500/1500 [==============================] - 1s 646us/step - loss: 0.2135 -
sparse_categorical_accuracy: 0.9218 - val_loss: 0.3377 - val_sparse_categori
cal_accuracy: 0.8843
Epoch 18/60
1500/1500 [==============================] - 1s 661us/step - loss: 0.2097 -
sparse_categorical_accuracy: 0.9224 - val_loss: 0.3401 - val_sparse_categori
cal_accuracy: 0.8852
Epoch 19/60
1500/1500 [==============================] - 1s 660us/step - loss: 0.2044 -
sparse_categorical_accuracy: 0.9229 - val_loss: 0.3423 - val_sparse_categori
cal_accuracy: 0.8852
Epoch 20/60
1500/1500 [==============================] - 1s 651us/step - loss: 0.2023 -
sparse_categorical_accuracy: 0.9246 - val_loss: 0.3356 - val_sparse_categori
cal_accuracy: 0.8905
Epoch 21/60
1500/1500 [==============================] - 1s 623us/step - loss: 0.1942 -
sparse_categorical_accuracy: 0.9279 - val_loss: 0.3537 - val_sparse_categori
cal_accuracy: 0.8826
Epoch 22/60
1500/1500 [==============================] - 1s 619us/step - loss: 0.1918 -
```

```
sparse_categorical_accuracy: 0.9288 - val_loss: 0.3624 - val_sparse_categori
cal_accuracy: 0.8841
Epoch 23/60
1500/1500 [==============================] - 1s 625us/step - loss: 0.1860 -
sparse_categorical_accuracy: 0.9317 - val_loss: 0.3466 - val_sparse_categori
cal_accuracy: 0.8873
Epoch 24/60
1500/1500 [==============================] - 1s 653us/step - loss: 0.1838 -
sparse_categorical_accuracy: 0.9316 - val_loss: 0.3558 - val_sparse_categori
cal_accuracy: 0.8854
Epoch 25/60
1500/1500 [==============================] - 1s 638us/step - loss: 0.1791 -
sparse_categorical_accuracy: 0.9336 - val_loss: 0.3614 - val_sparse_categori
cal_accuracy: 0.8884
Epoch 26/60
1500/1500 [==============================] - 1s 647us/step - loss: 0.1787 -
sparse_categorical_accuracy: 0.9338 - val_loss: 0.3639 - val_sparse_categori
cal_accuracy: 0.8871
Epoch 27/60
1500/1500 [==============================] - 1s 664us/step - loss: 0.1726 -
sparse_categorical_accuracy: 0.9369 - val_loss: 0.3548 - val_sparse_categori
cal_accuracy: 0.8886
Epoch 28/60
1500/1500 [==============================] - 1s 683us/step - loss: 0.1713 -
sparse_categorical_accuracy: 0.9366 - val_loss: 0.3544 - val_sparse_categori
cal_accuracy: 0.8908
Epoch 29/60
1500/1500 [==============================] - 1s 669us/step - loss: 0.1656 -
sparse_categorical_accuracy: 0.9383 - val_loss: 0.3780 - val_sparse_categori
cal_accuracy: 0.8816
Epoch 30/60
1500/1500 [==============================] - 1s 644us/step - loss: 0.1648 -
sparse_categorical_accuracy: 0.9387 - val_loss: 0.3671 - val_sparse_categori
cal_accuracy: 0.8867
Epoch 31/60
1500/1500 [==============================] - 1s 666us/step - loss: 0.1612 -
sparse_categorical_accuracy: 0.9399 - val_loss: 0.3833 - val_sparse_categori
cal_accuracy: 0.8843
Epoch 32/60
1500/1500 [==============================] - 1s 677us/step - loss: 0.1562 -
sparse_categorical_accuracy: 0.9421 - val_loss: 0.3773 - val_sparse_categori
cal_accuracy: 0.8873
Epoch 33/60
1500/1500 [==============================] - 1s 638us/step - loss: 0.1574 -
sparse_categorical_accuracy: 0.9416 - val_loss: 0.3801 - val_sparse_categori
cal_accuracy: 0.8881
Epoch 34/60
1500/1500 [==============================] - 1s 672us/step - loss: 0.1553 -
sparse_categorical_accuracy: 0.9415 - val_loss: 0.3801 - val_sparse_categori
cal_accuracy: 0.8908
Epoch 35/60
1500/1500 [==============================] - 1s 669us/step - loss: 0.1501 -
sparse_categorical_accuracy: 0.9454 - val_loss: 0.3833 - val_sparse_categori
```

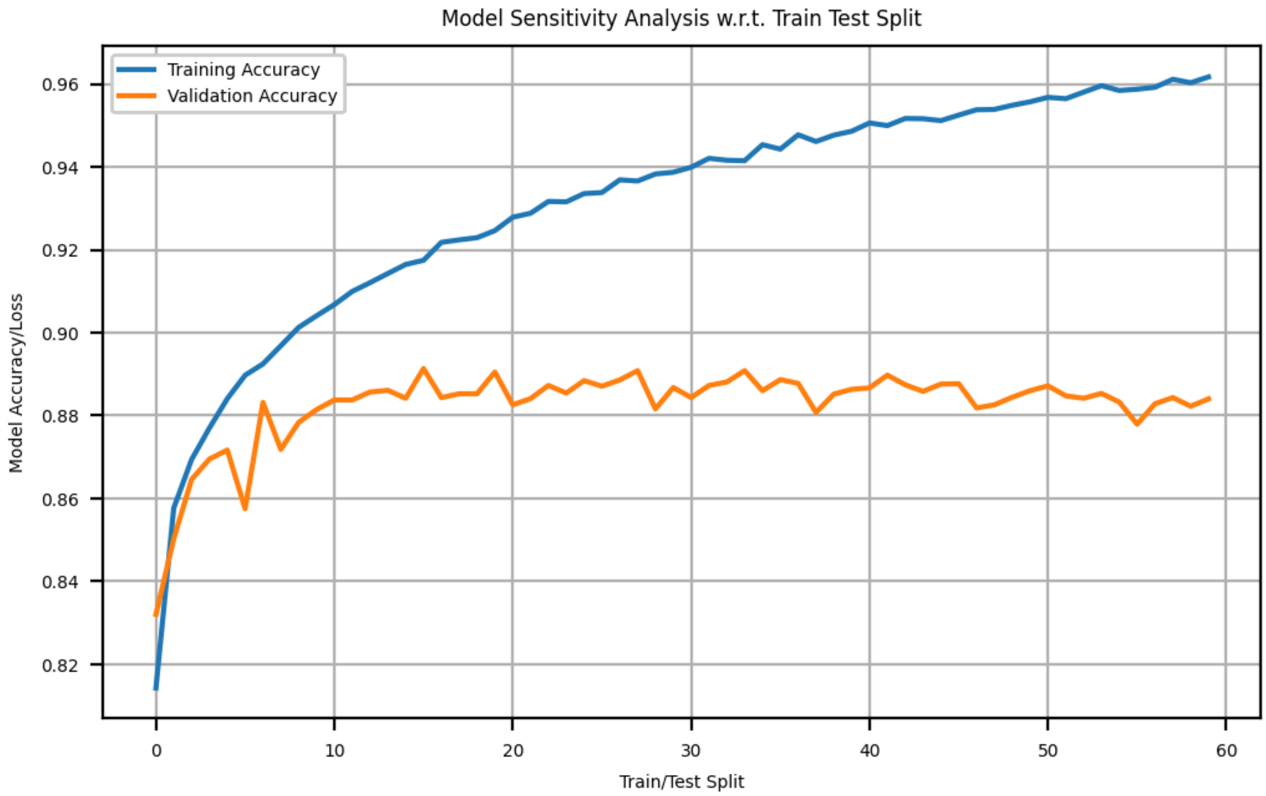
```
cal_accuracy: 0.8860
Epoch 36/60
1500/1500 [==============================] - 1s 653us/step - loss: 0.1491 -
sparse_categorical_accuracy: 0.9443 - val_loss: 0.3892 - val_sparse_categori
cal_accuracy: 0.8887
Epoch 37/60
1500/1500 [==============================] - 1s 641us/step - loss: 0.1426 -
sparse_categorical_accuracy: 0.9477 - val_loss: 0.4026 - val_sparse_categori
cal_accuracy: 0.8878
Epoch 38/60
1500/1500 [==============================] - 1s 636us/step - loss: 0.1437 -
sparse_categorical_accuracy: 0.9461 - val_loss: 0.4190 - val_sparse_categori
cal_accuracy: 0.8808
Epoch 39/60
1500/1500 [==============================] - 1s 682us/step - loss: 0.1400 -
sparse_categorical_accuracy: 0.9477 - val_loss: 0.4055 - val_sparse_categori
cal_accuracy: 0.8852
Epoch 40/60
1500/1500 [==============================] - 1s 638us/step - loss: 0.1387 -
sparse_categorical_accuracy: 0.9486 - val_loss: 0.4092 - val_sparse_categori
cal_accuracy: 0.8863
Epoch 41/60
1500/1500 [==============================] - 1s 599us/step - loss: 0.1369 -
sparse_categorical_accuracy: 0.9506 - val_loss: 0.4293 - val_sparse_categori
cal_accuracy: 0.8867
Epoch 42/60
1500/1500 [==============================] - 1s 633us/step - loss: 0.1348 -
sparse_categorical_accuracy: 0.9500 - val_loss: 0.4059 - val_sparse_categori
cal_accuracy: 0.8898
Epoch 43/60
1500/1500 [==============================] - 1s 591us/step - loss: 0.1321 -
sparse_categorical_accuracy: 0.9517 - val_loss: 0.4242 - val_sparse_categori
cal_accuracy: 0.8874
Epoch 44/60
1500/1500 [==============================] - 1s 615us/step - loss: 0.1315 -
sparse_categorical_accuracy: 0.9516 - val_loss: 0.4282 - val_sparse_categori
cal_accuracy: 0.8858
Epoch 45/60
1500/1500 [==============================] - 1s 670us/step - loss: 0.1309 -
sparse_categorical_accuracy: 0.9512 - val_loss: 0.4246 - val_sparse_categori
cal_accuracy: 0.8876
Epoch 46/60
1500/1500 [==============================] - 1s 678us/step - loss: 0.1268 -
sparse_categorical_accuracy: 0.9525 - val_loss: 0.4389 - val_sparse_categori
cal_accuracy: 0.8877
Epoch 47/60
1500/1500 [==============================] - 1s 703us/step - loss: 0.1241 -
sparse_categorical_accuracy: 0.9538 - val_loss: 0.4766 - val_sparse_categori
cal_accuracy: 0.8818
Epoch 48/60
1500/1500 [==============================] - 1s 707us/step - loss: 0.1234 -
sparse_categorical_accuracy: 0.9539 - val_loss: 0.4511 - val_sparse_categori
cal_accuracy: 0.8826
```

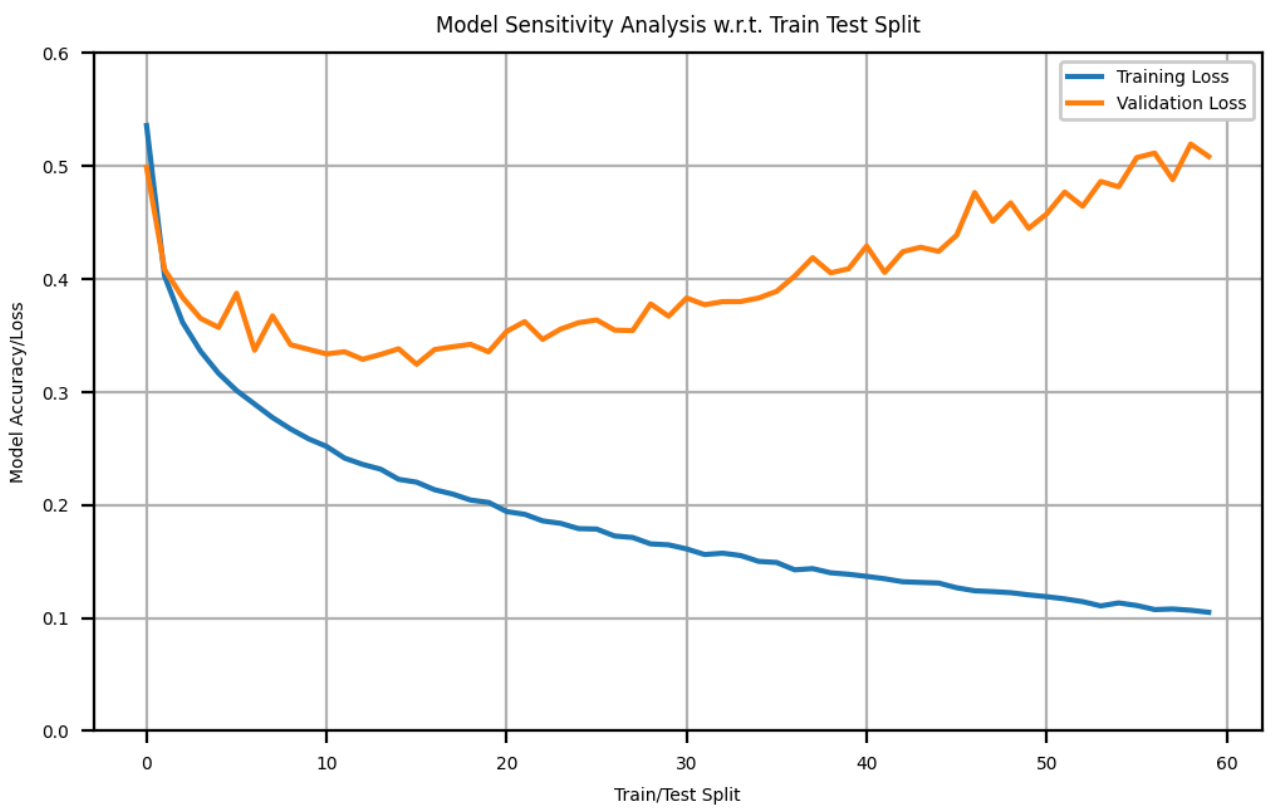
```
Epoch 49/60
1500/1500 [==============================] - 1s 683us/step - loss: 0.1224 -
sparse_categorical_accuracy: 0.9549 - val_loss: 0.4675 - val_sparse_categori
cal_accuracy: 0.8844
Epoch 50/60
1500/1500 [==============================] - 1s 655us/step - loss: 0.1205 -
sparse_categorical_accuracy: 0.9557 - val_loss: 0.4449 - val_sparse_categori
cal_accuracy: 0.8860
Epoch 51/60
1500/1500 [==============================] - 1s 667us/step - loss: 0.1189 -
sparse_categorical_accuracy: 0.9568 - val_loss: 0.4577 - val_sparse_categori
cal_accuracy: 0.8872
Epoch 52/60
1500/1500 [==============================] - 1s 656us/step - loss: 0.1169 -
sparse_categorical_accuracy: 0.9565 - val_loss: 0.4771 - val_sparse_categori
cal_accuracy: 0.8848
Epoch 53/60
1500/1500 [==============================] - 1s 685us/step - loss: 0.1145 -
sparse_categorical_accuracy: 0.9580 - val_loss: 0.4645 - val_sparse_categori
cal_accuracy: 0.8842
Epoch 54/60
1500/1500 [==============================] - 1s 670us/step - loss: 0.1107 -
sparse_categorical_accuracy: 0.9596 - val_loss: 0.4865 - val_sparse_categori
cal_accuracy: 0.8853
Epoch 55/60
1500/1500 [==============================] - 1s 656us/step - loss: 0.1133 -
sparse_categorical_accuracy: 0.9584 - val_loss: 0.4817 - val_sparse_categori
cal_accuracy: 0.8832
Epoch 56/60
1500/1500 [==============================] - 1s 694us/step - loss: 0.1111 -
sparse_categorical_accuracy: 0.9588 - val_loss: 0.5076 - val_sparse_categori
cal_accuracy: 0.8779
Epoch 57/60
1500/1500 [==============================] - 1s 662us/step - loss: 0.1074 -
sparse_categorical_accuracy: 0.9592 - val_loss: 0.5116 - val_sparse_categori
cal_accuracy: 0.8828
Epoch 58/60
1500/1500 [==============================] - 1s 686us/step - loss: 0.1080 -
sparse_categorical_accuracy: 0.9611 - val_loss: 0.4881 - val_sparse_categori
cal_accuracy: 0.8843
Epoch 59/60
1500/1500 [==============================] - 1s 593us/step - loss: 0.1069 -
sparse_categorical_accuracy: 0.9603 - val_loss: 0.5196 - val_sparse_categori
cal_accuracy: 0.8823
Epoch 60/60
1500/1500 [==============================] - 1s 598us/step - loss: 0.1050 -
sparse_categorical_accuracy: 0.9617 - val_loss: 0.5085 - val_sparse_categori
cal_accuracy: 0.8840
```

We can now observe the models training history w.r.t. validation and training accuracies and losses.

```
In [ ]: # create function to show model training accuracy and loss history
```

```python
def model_history(history):
        # plot training history
        fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

        ax.plot(history.history['sparse_categorical_accuracy'],
                label = r'Training Accuracy',
                #marker = 'o',
                markersize = 4)

        ax.plot(history.history['val_sparse_categorical_accuracy'],
                label = r'Validation Accuracy',
                #marker = 'o',
                markersize = 4)

        ax.set_title("Model Sensitivity Analysis w.r.t. Train Test Split")
        ax.set_xlabel('Train/Test Split')
        ax.set_ylabel('Model Accuracy/Loss')
        ax.legend()
        ax.legend().get_frame().set_alpha(1.0)

        ax.grid(True)

        # plot loss history
        fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

        ax.plot(history.history['loss'],
                label = r'Training Loss',
                #marker = 'o',
                markersize = 4)

        ax.plot(history.history['val_loss'],
                label = r'Validation Loss',
                #marker = 'o',
                markersize = 4)

        ax.set_title("Model Sensitivity Analysis w.r.t. Train Test Split")
        ax.set_xlabel('Train/Test Split')
        ax.set_ylabel('Model Accuracy/Loss')
        ax.set_ylim(0,0.6)
        ax.legend()
        ax.legend().get_frame().set_alpha(1.0)

        ax.grid(True)

# plot training history using the function
model_history(history)
```





The above loss and accuracy plots are a clear indicator of overfitting from the model. While training accuracy and loss improves with subseqent epochs, validation accuracy plateaus and validation loss appears to increase with subsequent epochs.

We can finally evaluate the model on the test data using sci-kit learn's `accuracy_score()` method.

```
In [ ]: from sklearn.metrics import accuracy_score

predicted_softmax = model.predict(x_test.astype(np.float32))
y_pred = np.argmax(predicted_softmax, axis = 1)

accuracy_score(y_test, y_pred)
```

```
313/313 [==============================] - 1s 2ms/step
```

```
Out[ ]: 0.862
```

The test accuracy of `0.862` is in line with the lower validation accuracy as described.

Model Evaluation: leaky ReLU

We next evaulate the Fashion MNIST dataset using a similar model with the keras `LeakyReLU` activation. We can call the previously created `model_arch()` function with the approriate parameters to create this model.

```
In [ ]: # build identical model with LeakyREeLU activation
model = model_arch(64, "LeakyReLU")

# show model architecture
model.summary()
```

```
Model: "units-64_activation-LeakyReLU"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 hidden_layer (Dense)        (None, 64)                50240

 leaky_re_lu (LeakyReLU)     (None, 64)                0

 output_layer (Dense)        (None, 10)                650

=================================================================
Total params: 50890 (198.79 KB)
Trainable params: 50890 (198.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

We then similarly compile and fit the new model.

```
In [ ]: # compile
model.compile(optimizer=keras.optimizers.legacy.Adam(learning_rate=1e-3),
              loss='sparse_categorical_crossentropy',
              metrics=['sparse_categorical_accuracy'],
```

```python
                )

# fit
history = model.fit(
    x_train_model.astype(np.float32), y_train,
    epochs=60,
    validation_split=0.2,
    shuffle = True
)
```

```
Epoch 1/60
1500/1500 [==============================] - 1s 711us/step - loss: 0.5424 - sparse_categorical_accuracy: 0.8124 - val_loss: 0.4755 - val_sparse_categorical_accuracy: 0.8313
Epoch 2/60
1500/1500 [==============================] - 1s 720us/step - loss: 0.4253 - sparse_categorical_accuracy: 0.8503 - val_loss: 0.4256 - val_sparse_categorical_accuracy: 0.8519
Epoch 3/60
1500/1500 [==============================] - 1s 710us/step - loss: 0.3905 - sparse_categorical_accuracy: 0.8597 - val_loss: 0.4038 - val_sparse_categorical_accuracy: 0.8548
Epoch 4/60
1500/1500 [==============================] - 1s 673us/step - loss: 0.3676 - sparse_categorical_accuracy: 0.8678 - val_loss: 0.3748 - val_sparse_categorical_accuracy: 0.8670
Epoch 5/60
1500/1500 [==============================] - 1s 649us/step - loss: 0.3510 - sparse_categorical_accuracy: 0.8753 - val_loss: 0.3755 - val_sparse_categorical_accuracy: 0.8641
Epoch 6/60
1500/1500 [==============================] - 1s 647us/step - loss: 0.3366 - sparse_categorical_accuracy: 0.8791 - val_loss: 0.4112 - val_sparse_categorical_accuracy: 0.8506
Epoch 7/60
1500/1500 [==============================] - 1s 623us/step - loss: 0.3253 - sparse_categorical_accuracy: 0.8811 - val_loss: 0.3694 - val_sparse_categorical_accuracy: 0.8656
Epoch 8/60
1500/1500 [==============================] - 1s 635us/step - loss: 0.3156 - sparse_categorical_accuracy: 0.8847 - val_loss: 0.3691 - val_sparse_categorical_accuracy: 0.8661
Epoch 9/60
1500/1500 [==============================] - 1s 652us/step - loss: 0.3084 - sparse_categorical_accuracy: 0.8881 - val_loss: 0.3475 - val_sparse_categorical_accuracy: 0.8761
Epoch 10/60
1500/1500 [==============================] - 1s 701us/step - loss: 0.3005 - sparse_categorical_accuracy: 0.8900 - val_loss: 0.3507 - val_sparse_categorical_accuracy: 0.8741
Epoch 11/60
1500/1500 [==============================] - 1s 661us/step - loss: 0.2931 - sparse_categorical_accuracy: 0.8922 - val_loss: 0.3419 - val_sparse_categorical_accuracy: 0.8777
```

```
Epoch 12/60
1500/1500 [==============================] - 1s 704us/step - loss: 0.2855 -
sparse_categorical_accuracy: 0.8942 - val_loss: 0.3711 - val_sparse_categori
cal_accuracy: 0.8665
Epoch 13/60
1500/1500 [==============================] - 1s 666us/step - loss: 0.2808 -
sparse_categorical_accuracy: 0.8972 - val_loss: 0.4133 - val_sparse_categori
cal_accuracy: 0.8512
Epoch 14/60
1500/1500 [==============================] - 1s 684us/step - loss: 0.2790 -
sparse_categorical_accuracy: 0.8981 - val_loss: 0.3562 - val_sparse_categori
cal_accuracy: 0.8749
Epoch 15/60
1500/1500 [==============================] - 1s 680us/step - loss: 0.2710 -
sparse_categorical_accuracy: 0.8991 - val_loss: 0.3536 - val_sparse_categori
cal_accuracy: 0.8749
Epoch 16/60
1500/1500 [==============================] - 1s 665us/step - loss: 0.2660 -
sparse_categorical_accuracy: 0.9011 - val_loss: 0.3673 - val_sparse_categori
cal_accuracy: 0.8745
Epoch 17/60
1500/1500 [==============================] - 1s 656us/step - loss: 0.2630 -
sparse_categorical_accuracy: 0.9021 - val_loss: 0.3655 - val_sparse_categori
cal_accuracy: 0.8757
Epoch 18/60
1500/1500 [==============================] - 1s 674us/step - loss: 0.2585 -
sparse_categorical_accuracy: 0.9038 - val_loss: 0.3483 - val_sparse_categori
cal_accuracy: 0.8821
Epoch 19/60
1500/1500 [==============================] - 1s 733us/step - loss: 0.2540 -
sparse_categorical_accuracy: 0.9050 - val_loss: 0.3555 - val_sparse_categori
cal_accuracy: 0.8790
Epoch 20/60
1500/1500 [==============================] - 1s 692us/step - loss: 0.2505 -
sparse_categorical_accuracy: 0.9072 - val_loss: 0.3479 - val_sparse_categori
cal_accuracy: 0.8806
Epoch 21/60
1500/1500 [==============================] - 1s 741us/step - loss: 0.2481 -
sparse_categorical_accuracy: 0.9089 - val_loss: 0.3603 - val_sparse_categori
cal_accuracy: 0.8794
Epoch 22/60
1500/1500 [==============================] - 1s 712us/step - loss: 0.2421 -
sparse_categorical_accuracy: 0.9103 - val_loss: 0.3560 - val_sparse_categori
cal_accuracy: 0.8824
Epoch 23/60
1500/1500 [==============================] - 1s 735us/step - loss: 0.2422 -
sparse_categorical_accuracy: 0.9106 - val_loss: 0.3480 - val_sparse_categori
cal_accuracy: 0.8825
Epoch 24/60
1500/1500 [==============================] - 1s 692us/step - loss: 0.2374 -
sparse_categorical_accuracy: 0.9106 - val_loss: 0.3493 - val_sparse_categori
cal_accuracy: 0.8849
Epoch 25/60
```

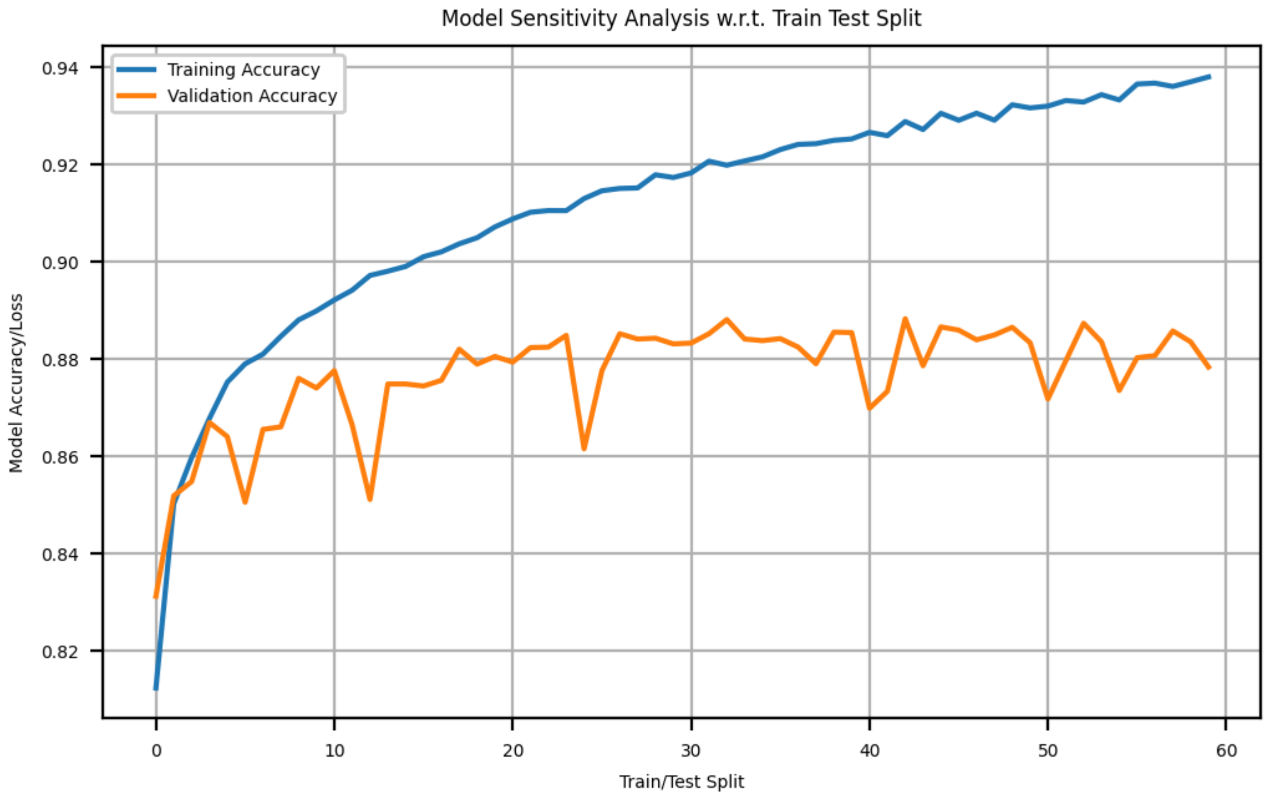
```
1500/1500 [==============================] - 1s 785us/step - loss: 0.2346 -
sparse_categorical_accuracy: 0.9131 - val_loss: 0.4125 - val_sparse_categori
cal_accuracy: 0.8616
Epoch 26/60
1500/1500 [==============================] - 1s 735us/step - loss: 0.2304 -
sparse_categorical_accuracy: 0.9147 - val_loss: 0.3629 - val_sparse_categori
cal_accuracy: 0.8777
Epoch 27/60
1500/1500 [==============================] - 1s 663us/step - loss: 0.2292 -
sparse_categorical_accuracy: 0.9151 - val_loss: 0.3411 - val_sparse_categori
cal_accuracy: 0.8852
Epoch 28/60
1500/1500 [==============================] - 1s 717us/step - loss: 0.2281 -
sparse_categorical_accuracy: 0.9153 - val_loss: 0.3466 - val_sparse_categori
cal_accuracy: 0.8842
Epoch 29/60
1500/1500 [==============================] - 1s 687us/step - loss: 0.2242 -
sparse_categorical_accuracy: 0.9180 - val_loss: 0.3478 - val_sparse_categori
cal_accuracy: 0.8843
Epoch 30/60
1500/1500 [==============================] - 1s 788us/step - loss: 0.2199 -
sparse_categorical_accuracy: 0.9174 - val_loss: 0.3614 - val_sparse_categori
cal_accuracy: 0.8832
Epoch 31/60
1500/1500 [==============================] - 1s 716us/step - loss: 0.2180 -
sparse_categorical_accuracy: 0.9183 - val_loss: 0.3494 - val_sparse_categori
cal_accuracy: 0.8833
Epoch 32/60
1500/1500 [==============================] - 1s 670us/step - loss: 0.2155 -
sparse_categorical_accuracy: 0.9208 - val_loss: 0.3588 - val_sparse_categori
cal_accuracy: 0.8852
Epoch 33/60
1500/1500 [==============================] - 1s 729us/step - loss: 0.2168 -
sparse_categorical_accuracy: 0.9199 - val_loss: 0.3550 - val_sparse_categori
cal_accuracy: 0.8882
Epoch 34/60
1500/1500 [==============================] - 1s 706us/step - loss: 0.2115 -
sparse_categorical_accuracy: 0.9208 - val_loss: 0.3649 - val_sparse_categori
cal_accuracy: 0.8842
Epoch 35/60
1500/1500 [==============================] - 1s 770us/step - loss: 0.2081 -
sparse_categorical_accuracy: 0.9216 - val_loss: 0.3685 - val_sparse_categori
cal_accuracy: 0.8838
Epoch 36/60
1500/1500 [==============================] - 1s 721us/step - loss: 0.2066 -
sparse_categorical_accuracy: 0.9231 - val_loss: 0.3707 - val_sparse_categori
cal_accuracy: 0.8842
Epoch 37/60
1500/1500 [==============================] - 1s 699us/step - loss: 0.2063 -
sparse_categorical_accuracy: 0.9242 - val_loss: 0.3702 - val_sparse_categori
cal_accuracy: 0.8825
Epoch 38/60
1500/1500 [==============================] - 1s 681us/step - loss: 0.2059 -
```

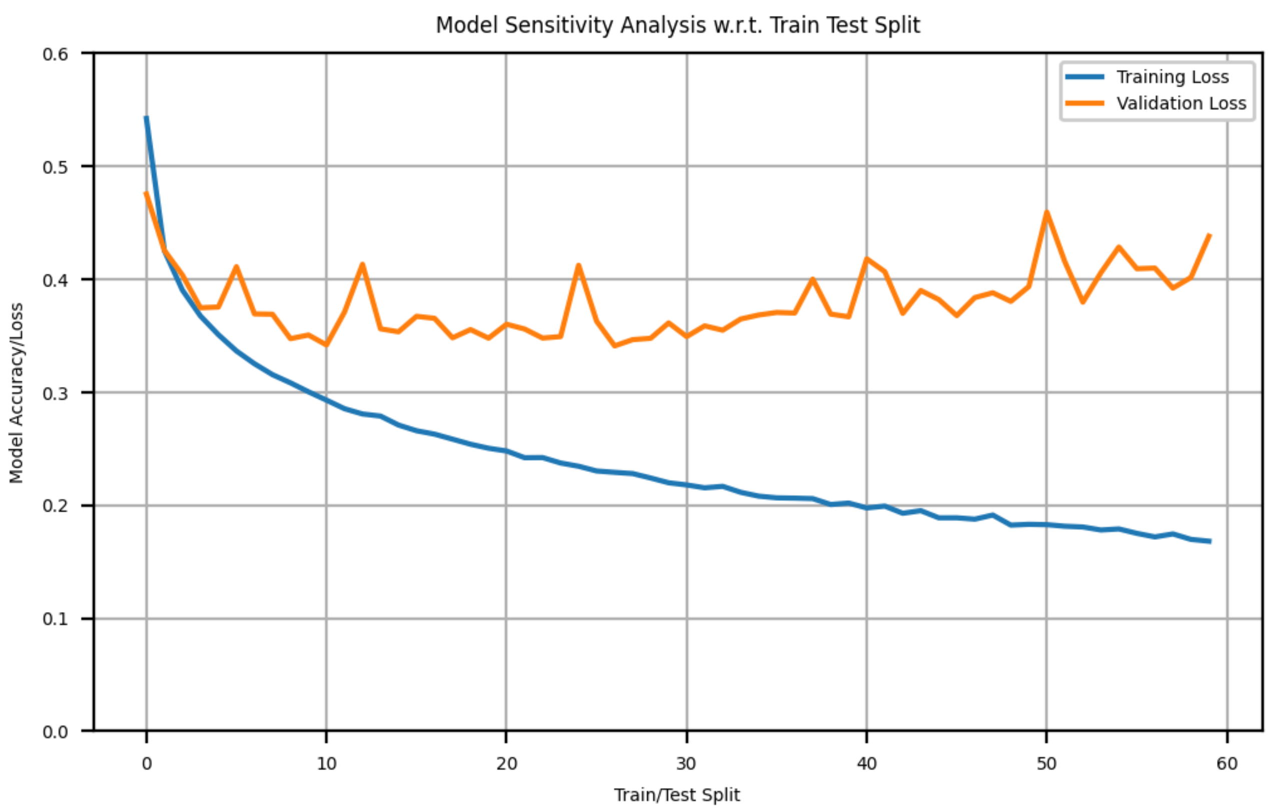
```
sparse_categorical_accuracy: 0.9244 - val_loss: 0.4002 - val_sparse_categori
cal_accuracy: 0.8791
Epoch 39/60
1500/1500 [==============================] - 1s 698us/step - loss: 0.2007 -
sparse_categorical_accuracy: 0.9250 - val_loss: 0.3693 - val_sparse_categori
cal_accuracy: 0.8856
Epoch 40/60
1500/1500 [==============================] - 1s 670us/step - loss: 0.2020 -
sparse_categorical_accuracy: 0.9253 - val_loss: 0.3669 - val_sparse_categori
cal_accuracy: 0.8855
Epoch 41/60
1500/1500 [==============================] - 1s 697us/step - loss: 0.1976 -
sparse_categorical_accuracy: 0.9267 - val_loss: 0.4182 - val_sparse_categori
cal_accuracy: 0.8699
Epoch 42/60
1500/1500 [==============================] - 1s 685us/step - loss: 0.1993 -
sparse_categorical_accuracy: 0.9260 - val_loss: 0.4068 - val_sparse_categori
cal_accuracy: 0.8734
Epoch 43/60
1500/1500 [==============================] - 1s 674us/step - loss: 0.1929 -
sparse_categorical_accuracy: 0.9289 - val_loss: 0.3699 - val_sparse_categori
cal_accuracy: 0.8883
Epoch 44/60
1500/1500 [==============================] - 1s 698us/step - loss: 0.1952 -
sparse_categorical_accuracy: 0.9273 - val_loss: 0.3901 - val_sparse_categori
cal_accuracy: 0.8787
Epoch 45/60
1500/1500 [==============================] - 1s 732us/step - loss: 0.1889 -
sparse_categorical_accuracy: 0.9306 - val_loss: 0.3821 - val_sparse_categori
cal_accuracy: 0.8867
Epoch 46/60
1500/1500 [==============================] - 1s 723us/step - loss: 0.1889 -
sparse_categorical_accuracy: 0.9292 - val_loss: 0.3679 - val_sparse_categori
cal_accuracy: 0.8860
Epoch 47/60
1500/1500 [==============================] - 1s 654us/step - loss: 0.1877 -
sparse_categorical_accuracy: 0.9306 - val_loss: 0.3839 - val_sparse_categori
cal_accuracy: 0.8840
Epoch 48/60
1500/1500 [==============================] - 1s 657us/step - loss: 0.1914 -
sparse_categorical_accuracy: 0.9292 - val_loss: 0.3881 - val_sparse_categori
cal_accuracy: 0.8850
Epoch 49/60
1500/1500 [==============================] - 1s 803us/step - loss: 0.1824 -
sparse_categorical_accuracy: 0.9324 - val_loss: 0.3805 - val_sparse_categori
cal_accuracy: 0.8866
Epoch 50/60
1500/1500 [==============================] - 1s 684us/step - loss: 0.1832 -
sparse_categorical_accuracy: 0.9317 - val_loss: 0.3936 - val_sparse_categori
cal_accuracy: 0.8834
Epoch 51/60
1500/1500 [==============================] - 1s 690us/step - loss: 0.1829 -
sparse_categorical_accuracy: 0.9321 - val_loss: 0.4595 - val_sparse_categori
```

```
cal_accuracy: 0.8718
Epoch 52/60
1500/1500 [==============================] - 1s 694us/step - loss: 0.1814 -
sparse_categorical_accuracy: 0.9332 - val_loss: 0.4155 - val_sparse_categori
cal_accuracy: 0.8797
Epoch 53/60
1500/1500 [==============================] - 1s 682us/step - loss: 0.1808 -
sparse_categorical_accuracy: 0.9329 - val_loss: 0.3798 - val_sparse_categori
cal_accuracy: 0.8874
Epoch 54/60
1500/1500 [==============================] - 1s 699us/step - loss: 0.1781 -
sparse_categorical_accuracy: 0.9344 - val_loss: 0.4059 - val_sparse_categori
cal_accuracy: 0.8836
Epoch 55/60
1500/1500 [==============================] - 1s 737us/step - loss: 0.1790 -
sparse_categorical_accuracy: 0.9334 - val_loss: 0.4286 - val_sparse_categori
cal_accuracy: 0.8736
Epoch 56/60
1500/1500 [==============================] - 1s 684us/step - loss: 0.1752 -
sparse_categorical_accuracy: 0.9366 - val_loss: 0.4094 - val_sparse_categori
cal_accuracy: 0.8803
Epoch 57/60
1500/1500 [==============================] - 1s 687us/step - loss: 0.1720 -
sparse_categorical_accuracy: 0.9368 - val_loss: 0.4100 - val_sparse_categori
cal_accuracy: 0.8808
Epoch 58/60
1500/1500 [==============================] - 1s 714us/step - loss: 0.1746 -
sparse_categorical_accuracy: 0.9361 - val_loss: 0.3923 - val_sparse_categori
cal_accuracy: 0.8858
Epoch 59/60
1500/1500 [==============================] - 1s 698us/step - loss: 0.1698 -
sparse_categorical_accuracy: 0.9371 - val_loss: 0.4017 - val_sparse_categori
cal_accuracy: 0.8836
Epoch 60/60
1500/1500 [==============================] - 1s 723us/step - loss: 0.1682 -
sparse_categorical_accuracy: 0.9381 - val_loss: 0.4382 - val_sparse_categori
cal_accuracy: 0.8784
```

The `LeakyReLU` model training history is as follows:

In [ ]:
```
# plot training history
model_history(history)
```





In [ ]:

```
predicted_softmax = model.predict(x_test.astype(np.float32))
y_pred = np.argmax(predicted_softmax, axis = 1)

accuracy_score(y_test, y_pred)
```

```
313/313 [==============================] - 1s 2ms/step
```

Out[ ]: 0.8536

The model utilizing `LeakyReLU` had a test accuracy of `0.8536`, comparable to the former `ReLU` model. Should a model be selected for further refinement or deployment, on the basis of activation function alone, the `LeakyReLU` model would be the preferred choice due to better loss history.

Optimizer Choice Influence

From the keras website there are a number of optimizers to to choose from for model architecture. The following optimizers were selected for analysis:

- SGD: Stochastic Gradient Descent with momentum optimizer.
- RMSprop: Optimizer that implements the RMSprop algorithm. Maintain a moving (discounted) average of the square of gradients, annd divide the gradient by the root of this average.
- ADAM: Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.
- Adadelta: Adadelta optimization is a stochastic gradient descent method that is based on adaptive learning rate per dimension.
- Ftrl: "Follow The Regularized Leader" (FTRL) is an optimization algorithm developed at Google for click-through rate prediction in the early 2010s. It is most suitable for shallow models with large and sparse feature spaces.

Similar to the methodology presented in **Problem 4**, we will maintain the best hyperparemeters from the previous **ffn** analysis and vary the subject optimizer hyperparameter for evaluation.

In [ ]:
```
# create list of optimizers and names for loop implementation
optimizers = [
    # instantiate optimizer objects with the same learning rates
    keras.optimizers.legacy.SGD(learning_rate=1e-3),
    keras.optimizers.legacy.RMSprop(learning_rate=1e-3),
    keras.optimizers.legacy.Adam(learning_rate=1e-3),
    keras.optimizers.legacy.Adadelta(learning_rate=1e-3),
    keras.optimizers.legacy.Ftrl(learning_rate=1e-3)
]

names = ["SGD", "RMSprop", "ADAM", "Adadelta", "Ftrl"]

# instantiate tracking ogjects for analysis
histories = [] # empty list to save history objects for each model
train_loss = [] # save train loss for each mode
train_accuracies = [] # save train accuracy for each model
test_accuracies = [] # save test accuracy for each model
```

```python
# begin training loop
print("Begin evaluation:")

for name, optimizer in zip(names, optimizers):

    # clear enviornment of previous model
    keras.backend.clear_session()

    # create ffn model using previously evaluated best hyperparameters
    model = model_arch(64, "LeakyReLU")

    # compile model with each optimizer
    model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['sparse_categorical_accuracy'],
              )

    # begin training, for the sake of brevity, number of epochs has been hal
    print(f"\nTraining, {name} optimizer\n")
    history = model.fit(
        x_train_model.astype(np.float32), y_train,
        epochs=30,
        validation_split=0.2,
        shuffle = True
    )

    # evaluate on test data
    print(f"\n {name} Test Accuracy:")
    predicted_softmax = model.predict(x_test.astype(np.float32))
    y_pred = np.argmax(predicted_softmax, axis = 1)

    test_accuracy = accuracy_score(y_test, y_pred)

    print(test_accuracy)

    # append tracking objects
    histories.append(history)
    train_loss.append(history.history["loss"][-1])
    train_accuracies.append(history.history["sparse_categorical_accuracy"][-
    test_accuracies.append(test_accuracy)
```

```
Begin evaluation:

Training, SGD optimizer

Epoch 1/30
1500/1500 [==============================] - 1s 618us/step - loss: 1.5208 -
sparse_categorical_accuracy: 0.5461 - val_loss: 1.1038 - val_sparse_categori
cal_accuracy: 0.6734
Epoch 2/30
1500/1500 [==============================] - 1s 590us/step - loss: 0.9747 -
sparse_categorical_accuracy: 0.6941 - val_loss: 0.8665 - val_sparse_categori
cal_accuracy: 0.7182
Epoch 3/30
```

```
1500/1500 [==============================] - 1s 571us/step - loss: 0.8250 -
sparse_categorical_accuracy: 0.7289 - val_loss: 0.7710 - val_sparse_categori
cal_accuracy: 0.7442
Epoch 4/30
1500/1500 [==============================] - 1s 567us/step - loss: 0.7509 -
sparse_categorical_accuracy: 0.7512 - val_loss: 0.7153 - val_sparse_categori
cal_accuracy: 0.7619
Epoch 5/30
1500/1500 [==============================] - 1s 585us/step - loss: 0.7028 -
sparse_categorical_accuracy: 0.7685 - val_loss: 0.6767 - val_sparse_categori
cal_accuracy: 0.7747
Epoch 6/30
1500/1500 [==============================] - 1s 577us/step - loss: 0.6677 -
sparse_categorical_accuracy: 0.7801 - val_loss: 0.6475 - val_sparse_categori
cal_accuracy: 0.7793
Epoch 7/30
1500/1500 [==============================] - 1s 559us/step - loss: 0.6410 -
sparse_categorical_accuracy: 0.7893 - val_loss: 0.6259 - val_sparse_categori
cal_accuracy: 0.7887
Epoch 8/30
1500/1500 [==============================] - 1s 553us/step - loss: 0.6193 -
sparse_categorical_accuracy: 0.7953 - val_loss: 0.6065 - val_sparse_categori
cal_accuracy: 0.7928
Epoch 9/30
1500/1500 [==============================] - 1s 493us/step - loss: 0.6015 -
sparse_categorical_accuracy: 0.8014 - val_loss: 0.5922 - val_sparse_categori
cal_accuracy: 0.7986
Epoch 10/30
1500/1500 [==============================] - 1s 515us/step - loss: 0.5862 -
sparse_categorical_accuracy: 0.8055 - val_loss: 0.5778 - val_sparse_categori
cal_accuracy: 0.8018
Epoch 11/30
1500/1500 [==============================] - 1s 490us/step - loss: 0.5731 -
sparse_categorical_accuracy: 0.8095 - val_loss: 0.5671 - val_sparse_categori
cal_accuracy: 0.8075
Epoch 12/30
1500/1500 [==============================] - 1s 533us/step - loss: 0.5618 -
sparse_categorical_accuracy: 0.8136 - val_loss: 0.5565 - val_sparse_categori
cal_accuracy: 0.8091
Epoch 13/30
1500/1500 [==============================] - 1s 562us/step - loss: 0.5518 -
sparse_categorical_accuracy: 0.8171 - val_loss: 0.5480 - val_sparse_categori
cal_accuracy: 0.8125
Epoch 14/30
1500/1500 [==============================] - 1s 481us/step - loss: 0.5430 -
sparse_categorical_accuracy: 0.8186 - val_loss: 0.5405 - val_sparse_categori
cal_accuracy: 0.8138
Epoch 15/30
1500/1500 [==============================] - 1s 499us/step - loss: 0.5349 -
sparse_categorical_accuracy: 0.8213 - val_loss: 0.5349 - val_sparse_categori
cal_accuracy: 0.8168
Epoch 16/30
1500/1500 [==============================] - 1s 533us/step - loss: 0.5278 -
```

```
sparse_categorical_accuracy: 0.8233 - val_loss: 0.5280 - val_sparse_categori
cal_accuracy: 0.8174
Epoch 17/30
1500/1500 [==============================] - 1s 478us/step - loss: 0.5211 -
sparse_categorical_accuracy: 0.8244 - val_loss: 0.5215 - val_sparse_categori
cal_accuracy: 0.8210
Epoch 18/30
1500/1500 [==============================] - 1s 496us/step - loss: 0.5154 -
sparse_categorical_accuracy: 0.8274 - val_loss: 0.5171 - val_sparse_categori
cal_accuracy: 0.8201
Epoch 19/30
1500/1500 [==============================] - 1s 514us/step - loss: 0.5100 -
sparse_categorical_accuracy: 0.8280 - val_loss: 0.5127 - val_sparse_categori
cal_accuracy: 0.8204
Epoch 20/30
1500/1500 [==============================] - 1s 472us/step - loss: 0.5050 -
sparse_categorical_accuracy: 0.8301 - val_loss: 0.5090 - val_sparse_categori
cal_accuracy: 0.8230
Epoch 21/30
1500/1500 [==============================] - 1s 485us/step - loss: 0.5004 -
sparse_categorical_accuracy: 0.8313 - val_loss: 0.5034 - val_sparse_categori
cal_accuracy: 0.8275
Epoch 22/30
1500/1500 [==============================] - 1s 503us/step - loss: 0.4962 -
sparse_categorical_accuracy: 0.8325 - val_loss: 0.4996 - val_sparse_categori
cal_accuracy: 0.8264
Epoch 23/30
1500/1500 [==============================] - 1s 469us/step - loss: 0.4922 -
sparse_categorical_accuracy: 0.8335 - val_loss: 0.4956 - val_sparse_categori
cal_accuracy: 0.8285
Epoch 24/30
1500/1500 [==============================] - 1s 494us/step - loss: 0.4884 -
sparse_categorical_accuracy: 0.8346 - val_loss: 0.4932 - val_sparse_categori
cal_accuracy: 0.8277
Epoch 25/30
1500/1500 [==============================] - 1s 484us/step - loss: 0.4847 -
sparse_categorical_accuracy: 0.8356 - val_loss: 0.4893 - val_sparse_categori
cal_accuracy: 0.8298
Epoch 26/30
1500/1500 [==============================] - 1s 465us/step - loss: 0.4815 -
sparse_categorical_accuracy: 0.8365 - val_loss: 0.4868 - val_sparse_categori
cal_accuracy: 0.8305
Epoch 27/30
1500/1500 [==============================] - 1s 487us/step - loss: 0.4783 -
sparse_categorical_accuracy: 0.8369 - val_loss: 0.4868 - val_sparse_categori
cal_accuracy: 0.8288
Epoch 28/30
1500/1500 [==============================] - 1s 468us/step - loss: 0.4755 -
sparse_categorical_accuracy: 0.8382 - val_loss: 0.4817 - val_sparse_categori
cal_accuracy: 0.8317
Epoch 29/30
1500/1500 [==============================] - 1s 483us/step - loss: 0.4727 -
sparse_categorical_accuracy: 0.8388 - val_loss: 0.4784 - val_sparse_categori
```

```
cal_accuracy: 0.8339
Epoch 30/30
1500/1500 [==============================] - 1s 486us/step - loss: 0.4699 -
sparse_categorical_accuracy: 0.8390 - val_loss: 0.4764 - val_sparse_categori
cal_accuracy: 0.8353

 SGD Test Accuracy:
313/313 [==============================] - 0s 293us/step
0.7912

Training, RMSprop optimizer

Epoch 1/30
1500/1500 [==============================] - 1s 580us/step - loss: 0.5607 -
sparse_categorical_accuracy: 0.8040 - val_loss: 0.4686 - val_sparse_categori
cal_accuracy: 0.8333
Epoch 2/30
1500/1500 [==============================] - 1s 526us/step - loss: 0.4326 -
sparse_categorical_accuracy: 0.8485 - val_loss: 0.4216 - val_sparse_categori
cal_accuracy: 0.8480
Epoch 3/30
1500/1500 [==============================] - 1s 547us/step - loss: 0.3992 -
sparse_categorical_accuracy: 0.8594 - val_loss: 0.3869 - val_sparse_categori
cal_accuracy: 0.8640
Epoch 4/30
1500/1500 [==============================] - 1s 542us/step - loss: 0.3770 -
sparse_categorical_accuracy: 0.8656 - val_loss: 0.3778 - val_sparse_categori
cal_accuracy: 0.8650
Epoch 5/30
1500/1500 [==============================] - 1s 526us/step - loss: 0.3609 -
sparse_categorical_accuracy: 0.8699 - val_loss: 0.3854 - val_sparse_categori
cal_accuracy: 0.8612
Epoch 6/30
1500/1500 [==============================] - 1s 540us/step - loss: 0.3482 -
sparse_categorical_accuracy: 0.8759 - val_loss: 0.3852 - val_sparse_categori
cal_accuracy: 0.8656
Epoch 7/30
1500/1500 [==============================] - 1s 540us/step - loss: 0.3382 -
sparse_categorical_accuracy: 0.8790 - val_loss: 0.3748 - val_sparse_categori
cal_accuracy: 0.8668
Epoch 8/30
1500/1500 [==============================] - 1s 543us/step - loss: 0.3290 -
sparse_categorical_accuracy: 0.8815 - val_loss: 0.3513 - val_sparse_categori
cal_accuracy: 0.8761
Epoch 9/30
1500/1500 [==============================] - 1s 525us/step - loss: 0.3199 -
sparse_categorical_accuracy: 0.8855 - val_loss: 0.3871 - val_sparse_categori
cal_accuracy: 0.8671
Epoch 10/30
1500/1500 [==============================] - 1s 543us/step - loss: 0.3145 -
sparse_categorical_accuracy: 0.8873 - val_loss: 0.3934 - val_sparse_categori
cal_accuracy: 0.8642
Epoch 11/30
```

```
1500/1500 [==============================] - 1s 554us/step - loss: 0.3087 -
sparse_categorical_accuracy: 0.8882 - val_loss: 0.3642 - val_sparse_categori
cal_accuracy: 0.8723
Epoch 12/30
1500/1500 [==============================] - 1s 540us/step - loss: 0.3027 -
sparse_categorical_accuracy: 0.8918 - val_loss: 0.3548 - val_sparse_categori
cal_accuracy: 0.8771
Epoch 13/30
1500/1500 [==============================] - 1s 570us/step - loss: 0.2994 -
sparse_categorical_accuracy: 0.8920 - val_loss: 0.3762 - val_sparse_categori
cal_accuracy: 0.8723
Epoch 14/30
1500/1500 [==============================] - 1s 629us/step - loss: 0.2944 -
sparse_categorical_accuracy: 0.8955 - val_loss: 0.3468 - val_sparse_categori
cal_accuracy: 0.8799
Epoch 15/30
1500/1500 [==============================] - 1s 565us/step - loss: 0.2899 -
sparse_categorical_accuracy: 0.8956 - val_loss: 0.3688 - val_sparse_categori
cal_accuracy: 0.8755
Epoch 16/30
1500/1500 [==============================] - 1s 562us/step - loss: 0.2863 -
sparse_categorical_accuracy: 0.8963 - val_loss: 0.3492 - val_sparse_categori
cal_accuracy: 0.8849
Epoch 17/30
1500/1500 [==============================] - 1s 560us/step - loss: 0.2837 -
sparse_categorical_accuracy: 0.8987 - val_loss: 0.3660 - val_sparse_categori
cal_accuracy: 0.8763
Epoch 18/30
1500/1500 [==============================] - 1s 630us/step - loss: 0.2805 -
sparse_categorical_accuracy: 0.9000 - val_loss: 0.3613 - val_sparse_categori
cal_accuracy: 0.8817
Epoch 19/30
1500/1500 [==============================] - 1s 601us/step - loss: 0.2761 -
sparse_categorical_accuracy: 0.9002 - val_loss: 0.3621 - val_sparse_categori
cal_accuracy: 0.8798
Epoch 20/30
1500/1500 [==============================] - 1s 634us/step - loss: 0.2739 -
sparse_categorical_accuracy: 0.9018 - val_loss: 0.3469 - val_sparse_categori
cal_accuracy: 0.8827
Epoch 21/30
1500/1500 [==============================] - 1s 633us/step - loss: 0.2716 -
sparse_categorical_accuracy: 0.9026 - val_loss: 0.3698 - val_sparse_categori
cal_accuracy: 0.8759
Epoch 22/30
1500/1500 [==============================] - 1s 592us/step - loss: 0.2694 -
sparse_categorical_accuracy: 0.9031 - val_loss: 0.3730 - val_sparse_categori
cal_accuracy: 0.8795
Epoch 23/30
1500/1500 [==============================] - 1s 578us/step - loss: 0.2661 -
sparse_categorical_accuracy: 0.9041 - val_loss: 0.4004 - val_sparse_categori
cal_accuracy: 0.8683
Epoch 24/30
1500/1500 [==============================] - 1s 587us/step - loss: 0.2638 -
```

```
sparse_categorical_accuracy: 0.9046 - val_loss: 0.3804 - val_sparse_categori
cal_accuracy: 0.8793
Epoch 25/30
1500/1500 [==============================] - 1s 581us/step - loss: 0.2629 -
sparse_categorical_accuracy: 0.9063 - val_loss: 0.3738 - val_sparse_categori
cal_accuracy: 0.8783
Epoch 26/30
1500/1500 [==============================] - 1s 581us/step - loss: 0.2616 -
sparse_categorical_accuracy: 0.9059 - val_loss: 0.3957 - val_sparse_categori
cal_accuracy: 0.8757
Epoch 27/30
1500/1500 [==============================] - 1s 573us/step - loss: 0.2572 -
sparse_categorical_accuracy: 0.9064 - val_loss: 0.3666 - val_sparse_categori
cal_accuracy: 0.8815
Epoch 28/30
1500/1500 [==============================] - 1s 596us/step - loss: 0.2569 -
sparse_categorical_accuracy: 0.9075 - val_loss: 0.3555 - val_sparse_categori
cal_accuracy: 0.8837
Epoch 29/30
1500/1500 [==============================] - 1s 590us/step - loss: 0.2543 -
sparse_categorical_accuracy: 0.9097 - val_loss: 0.3651 - val_sparse_categori
cal_accuracy: 0.8817
Epoch 30/30
1500/1500 [==============================] - 1s 584us/step - loss: 0.2518 -
sparse_categorical_accuracy: 0.9093 - val_loss: 0.3615 - val_sparse_categori
cal_accuracy: 0.8861

 RMSprop Test Accuracy:
313/313 [==============================] - 0s 313us/step
0.8504

Training, ADAM optimizer

Epoch 1/30
1500/1500 [==============================] - 1s 659us/step - loss: 0.5376 -
sparse_categorical_accuracy: 0.8130 - val_loss: 0.4813 - val_sparse_categori
cal_accuracy: 0.8274
Epoch 2/30
1500/1500 [==============================] - 1s 660us/step - loss: 0.4219 -
sparse_categorical_accuracy: 0.8515 - val_loss: 0.4405 - val_sparse_categori
cal_accuracy: 0.8424
Epoch 3/30
1500/1500 [==============================] - 1s 624us/step - loss: 0.3856 -
sparse_categorical_accuracy: 0.8621 - val_loss: 0.4302 - val_sparse_categori
cal_accuracy: 0.8398
Epoch 4/30
1500/1500 [==============================] - 1s 626us/step - loss: 0.3659 -
sparse_categorical_accuracy: 0.8687 - val_loss: 0.4037 - val_sparse_categori
cal_accuracy: 0.8610
Epoch 5/30
1500/1500 [==============================] - 1s 635us/step - loss: 0.3486 -
sparse_categorical_accuracy: 0.8753 - val_loss: 0.3726 - val_sparse_categori
cal_accuracy: 0.8661
```

```
Epoch 6/30
1500/1500 [==============================] - 1s 672us/step - loss: 0.3350 -
sparse_categorical_accuracy: 0.8771 - val_loss: 0.3649 - val_sparse_categori
cal_accuracy: 0.8717
Epoch 7/30
1500/1500 [==============================] - 1s 619us/step - loss: 0.3227 -
sparse_categorical_accuracy: 0.8826 - val_loss: 0.3489 - val_sparse_categori
cal_accuracy: 0.8763
Epoch 8/30
1500/1500 [==============================] - 1s 619us/step - loss: 0.3160 -
sparse_categorical_accuracy: 0.8841 - val_loss: 0.3629 - val_sparse_categori
cal_accuracy: 0.8697
Epoch 9/30
1500/1500 [==============================] - 1s 621us/step - loss: 0.3074 -
sparse_categorical_accuracy: 0.8882 - val_loss: 0.3698 - val_sparse_categori
cal_accuracy: 0.8673
Epoch 10/30
1500/1500 [==============================] - 1s 618us/step - loss: 0.2988 -
sparse_categorical_accuracy: 0.8910 - val_loss: 0.3424 - val_sparse_categori
cal_accuracy: 0.8796
Epoch 11/30
1500/1500 [==============================] - 1s 611us/step - loss: 0.2932 -
sparse_categorical_accuracy: 0.8935 - val_loss: 0.3488 - val_sparse_categori
cal_accuracy: 0.8757
Epoch 12/30
1500/1500 [==============================] - 1s 674us/step - loss: 0.2871 -
sparse_categorical_accuracy: 0.8941 - val_loss: 0.3414 - val_sparse_categori
cal_accuracy: 0.8808
Epoch 13/30
1500/1500 [==============================] - 1s 659us/step - loss: 0.2797 -
sparse_categorical_accuracy: 0.8975 - val_loss: 0.3459 - val_sparse_categori
cal_accuracy: 0.8769
Epoch 14/30
1500/1500 [==============================] - 1s 706us/step - loss: 0.2756 -
sparse_categorical_accuracy: 0.8995 - val_loss: 0.3410 - val_sparse_categori
cal_accuracy: 0.8802
Epoch 15/30
1500/1500 [==============================] - 1s 675us/step - loss: 0.2674 -
sparse_categorical_accuracy: 0.9020 - val_loss: 0.3501 - val_sparse_categori
cal_accuracy: 0.8760
Epoch 16/30
1500/1500 [==============================] - 1s 658us/step - loss: 0.2672 -
sparse_categorical_accuracy: 0.9021 - val_loss: 0.4051 - val_sparse_categori
cal_accuracy: 0.8577
Epoch 17/30
1500/1500 [==============================] - 1s 659us/step - loss: 0.2627 -
sparse_categorical_accuracy: 0.9036 - val_loss: 0.3680 - val_sparse_categori
cal_accuracy: 0.8733
Epoch 18/30
1500/1500 [==============================] - 1s 655us/step - loss: 0.2593 -
sparse_categorical_accuracy: 0.9046 - val_loss: 0.3499 - val_sparse_categori
cal_accuracy: 0.8787
Epoch 19/30
```

```
1500/1500 [==============================] - 1s 638us/step - loss: 0.2534 -
sparse_categorical_accuracy: 0.9066 - val_loss: 0.3582 - val_sparse_categori
cal_accuracy: 0.8752
Epoch 20/30
1500/1500 [==============================] - 1s 607us/step - loss: 0.2521 -
sparse_categorical_accuracy: 0.9074 - val_loss: 0.3459 - val_sparse_categori
cal_accuracy: 0.8827
Epoch 21/30
1500/1500 [==============================] - 1s 673us/step - loss: 0.2448 -
sparse_categorical_accuracy: 0.9101 - val_loss: 0.3464 - val_sparse_categori
cal_accuracy: 0.8787
Epoch 22/30
1500/1500 [==============================] - 1s 659us/step - loss: 0.2429 -
sparse_categorical_accuracy: 0.9105 - val_loss: 0.3792 - val_sparse_categori
cal_accuracy: 0.8697
Epoch 23/30
1500/1500 [==============================] - 1s 672us/step - loss: 0.2390 -
sparse_categorical_accuracy: 0.9120 - val_loss: 0.3391 - val_sparse_categori
cal_accuracy: 0.8823
Epoch 24/30
1500/1500 [==============================] - 1s 615us/step - loss: 0.2363 -
sparse_categorical_accuracy: 0.9118 - val_loss: 0.3570 - val_sparse_categori
cal_accuracy: 0.8802
Epoch 25/30
1500/1500 [==============================] - 1s 618us/step - loss: 0.2314 -
sparse_categorical_accuracy: 0.9146 - val_loss: 0.3469 - val_sparse_categori
cal_accuracy: 0.8849
Epoch 26/30
1500/1500 [==============================] - 1s 629us/step - loss: 0.2280 -
sparse_categorical_accuracy: 0.9160 - val_loss: 0.3405 - val_sparse_categori
cal_accuracy: 0.8843
Epoch 27/30
1500/1500 [==============================] - 1s 622us/step - loss: 0.2272 -
sparse_categorical_accuracy: 0.9156 - val_loss: 0.3461 - val_sparse_categori
cal_accuracy: 0.8835
Epoch 28/30
1500/1500 [==============================] - 1s 627us/step - loss: 0.2265 -
sparse_categorical_accuracy: 0.9169 - val_loss: 0.3821 - val_sparse_categori
cal_accuracy: 0.8720
Epoch 29/30
1500/1500 [==============================] - 1s 617us/step - loss: 0.2220 -
sparse_categorical_accuracy: 0.9171 - val_loss: 0.3581 - val_sparse_categori
cal_accuracy: 0.8816
Epoch 30/30
1500/1500 [==============================] - 1s 629us/step - loss: 0.2238 -
sparse_categorical_accuracy: 0.9162 - val_loss: 0.3690 - val_sparse_categori
cal_accuracy: 0.8791

 ADAM Test Accuracy:
313/313 [==============================] - 0s 321us/step
0.863

Training, Adadelta optimizer
```

```
Epoch 1/30
1500/1500 [==============================] - 1s 701us/step - loss: 2.2446 -
sparse_categorical_accuracy: 0.1600 - val_loss: 2.0975 - val_sparse_categori
cal_accuracy: 0.2637
Epoch 2/30
1500/1500 [==============================] - 1s 677us/step - loss: 1.9932 -
sparse_categorical_accuracy: 0.3221 - val_loss: 1.8843 - val_sparse_categori
cal_accuracy: 0.3969
Epoch 3/30
1500/1500 [==============================] - 1s 675us/step - loss: 1.8054 -
sparse_categorical_accuracy: 0.4533 - val_loss: 1.7145 - val_sparse_categori
cal_accuracy: 0.5110
Epoch 4/30
1500/1500 [==============================] - 1s 673us/step - loss: 1.6520 -
sparse_categorical_accuracy: 0.5379 - val_loss: 1.5739 - val_sparse_categori
cal_accuracy: 0.5702
Epoch 5/30
1500/1500 [==============================] - 1s 679us/step - loss: 1.5238 -
sparse_categorical_accuracy: 0.5798 - val_loss: 1.4561 - val_sparse_categori
cal_accuracy: 0.6007
Epoch 6/30
1500/1500 [==============================] - 1s 674us/step - loss: 1.4166 -
sparse_categorical_accuracy: 0.6068 - val_loss: 1.3576 - val_sparse_categori
cal_accuracy: 0.6241
Epoch 7/30
1500/1500 [==============================] - 1s 675us/step - loss: 1.3270 -
sparse_categorical_accuracy: 0.6279 - val_loss: 1.2760 - val_sparse_categori
cal_accuracy: 0.6434
Epoch 8/30
1500/1500 [==============================] - 1s 692us/step - loss: 1.2530 -
sparse_categorical_accuracy: 0.6436 - val_loss: 1.2081 - val_sparse_categori
cal_accuracy: 0.6559
Epoch 9/30
1500/1500 [==============================] - 1s 678us/step - loss: 1.1909 -
sparse_categorical_accuracy: 0.6546 - val_loss: 1.1513 - val_sparse_categori
cal_accuracy: 0.6689
Epoch 10/30
1500/1500 [==============================] - 1s 675us/step - loss: 1.1383 -
sparse_categorical_accuracy: 0.6656 - val_loss: 1.1026 - val_sparse_categori
cal_accuracy: 0.6790
Epoch 11/30
1500/1500 [==============================] - 1s 630us/step - loss: 1.0934 -
sparse_categorical_accuracy: 0.6734 - val_loss: 1.0608 - val_sparse_categori
cal_accuracy: 0.6862
Epoch 12/30
1500/1500 [==============================] - 1s 641us/step - loss: 1.0544 -
sparse_categorical_accuracy: 0.6799 - val_loss: 1.0243 - val_sparse_categori
cal_accuracy: 0.6933
Epoch 13/30
1500/1500 [==============================] - 1s 641us/step - loss: 1.0203 -
sparse_categorical_accuracy: 0.6863 - val_loss: 0.9923 - val_sparse_categori
cal_accuracy: 0.6978
```

```
Epoch 14/30
1500/1500 [==============================] - 1s 640us/step - loss: 0.9904 -
sparse_categorical_accuracy: 0.6918 - val_loss: 0.9644 - val_sparse_categori
cal_accuracy: 0.7034
Epoch 15/30
1500/1500 [==============================] - 1s 684us/step - loss: 0.9640 -
sparse_categorical_accuracy: 0.6968 - val_loss: 0.9395 - val_sparse_categori
cal_accuracy: 0.7089
Epoch 16/30
1500/1500 [==============================] - 1s 648us/step - loss: 0.9404 -
sparse_categorical_accuracy: 0.7024 - val_loss: 0.9172 - val_sparse_categori
cal_accuracy: 0.7143
Epoch 17/30
1500/1500 [==============================] - 1s 627us/step - loss: 0.9191 -
sparse_categorical_accuracy: 0.7083 - val_loss: 0.8971 - val_sparse_categori
cal_accuracy: 0.7171
Epoch 18/30
1500/1500 [==============================] - 1s 630us/step - loss: 0.9000 -
sparse_categorical_accuracy: 0.7129 - val_loss: 0.8791 - val_sparse_categori
cal_accuracy: 0.7220
Epoch 19/30
1500/1500 [==============================] - 1s 622us/step - loss: 0.8827 -
sparse_categorical_accuracy: 0.7178 - val_loss: 0.8626 - val_sparse_categori
cal_accuracy: 0.7266
Epoch 20/30
1500/1500 [==============================] - 1s 629us/step - loss: 0.8666 -
sparse_categorical_accuracy: 0.7218 - val_loss: 0.8474 - val_sparse_categori
cal_accuracy: 0.7309
Epoch 21/30
1500/1500 [==============================] - 1s 635us/step - loss: 0.8520 -
sparse_categorical_accuracy: 0.7260 - val_loss: 0.8338 - val_sparse_categori
cal_accuracy: 0.7344
Epoch 22/30
1500/1500 [==============================] - 1s 620us/step - loss: 0.8387 -
sparse_categorical_accuracy: 0.7298 - val_loss: 0.8212 - val_sparse_categori
cal_accuracy: 0.7375
Epoch 23/30
1500/1500 [==============================] - 1s 628us/step - loss: 0.8264 -
sparse_categorical_accuracy: 0.7332 - val_loss: 0.8094 - val_sparse_categori
cal_accuracy: 0.7409
Epoch 24/30
1500/1500 [==============================] - 1s 625us/step - loss: 0.8148 -
sparse_categorical_accuracy: 0.7364 - val_loss: 0.7986 - val_sparse_categori
cal_accuracy: 0.7437
Epoch 25/30
1500/1500 [==============================] - 1s 615us/step - loss: 0.8041 -
sparse_categorical_accuracy: 0.7400 - val_loss: 0.7882 - val_sparse_categori
cal_accuracy: 0.7456
Epoch 26/30
1500/1500 [==============================] - 1s 625us/step - loss: 0.7938 -
sparse_categorical_accuracy: 0.7430 - val_loss: 0.7785 - val_sparse_categori
cal_accuracy: 0.7498
Epoch 27/30
```

```
1500/1500 [==============================] - 1s 623us/step - loss: 0.7843 -
sparse_categorical_accuracy: 0.7463 - val_loss: 0.7696 - val_sparse_categori
cal_accuracy: 0.7530
Epoch 28/30
1500/1500 [==============================] - 1s 620us/step - loss: 0.7754 -
sparse_categorical_accuracy: 0.7486 - val_loss: 0.7612 - val_sparse_categori
cal_accuracy: 0.7553
Epoch 29/30
1500/1500 [==============================] - 1s 647us/step - loss: 0.7669 -
sparse_categorical_accuracy: 0.7513 - val_loss: 0.7528 - val_sparse_categori
cal_accuracy: 0.7566
Epoch 30/30
1500/1500 [==============================] - 1s 669us/step - loss: 0.7586 -
sparse_categorical_accuracy: 0.7544 - val_loss: 0.7451 - val_sparse_categori
cal_accuracy: 0.7584

 Adadelta Test Accuracy:
313/313 [==============================] - 0s 327us/step
0.738

Training, Ftrl optimizer

Epoch 1/30
1500/1500 [==============================] - 1s 762us/step - loss: 2.2342 -
sparse_categorical_accuracy: 0.2295 - val_loss: 2.0115 - val_sparse_categori
cal_accuracy: 0.2884
Epoch 2/30
1500/1500 [==============================] - 1s 712us/step - loss: 1.7400 -
sparse_categorical_accuracy: 0.3511 - val_loss: 1.5382 - val_sparse_categori
cal_accuracy: 0.4240
Epoch 3/30
1500/1500 [==============================] - 1s 711us/step - loss: 1.4203 -
sparse_categorical_accuracy: 0.5265 - val_loss: 1.3115 - val_sparse_categori
cal_accuracy: 0.5709
Epoch 4/30
1500/1500 [==============================] - 1s 667us/step - loss: 1.2377 -
sparse_categorical_accuracy: 0.5799 - val_loss: 1.1580 - val_sparse_categori
cal_accuracy: 0.6132
Epoch 5/30
1500/1500 [==============================] - 1s 663us/step - loss: 1.1078 -
sparse_categorical_accuracy: 0.6179 - val_loss: 1.0460 - val_sparse_categori
cal_accuracy: 0.6366
Epoch 6/30
1500/1500 [==============================] - 1s 686us/step - loss: 1.0130 -
sparse_categorical_accuracy: 0.6402 - val_loss: 0.9642 - val_sparse_categori
cal_accuracy: 0.6565
Epoch 7/30
1500/1500 [==============================] - 1s 665us/step - loss: 0.9443 -
sparse_categorical_accuracy: 0.6561 - val_loss: 0.9059 - val_sparse_categori
cal_accuracy: 0.6684
Epoch 8/30
1500/1500 [==============================] - 1s 668us/step - loss: 0.8945 -
sparse_categorical_accuracy: 0.6672 - val_loss: 0.8633 - val_sparse_categori
```

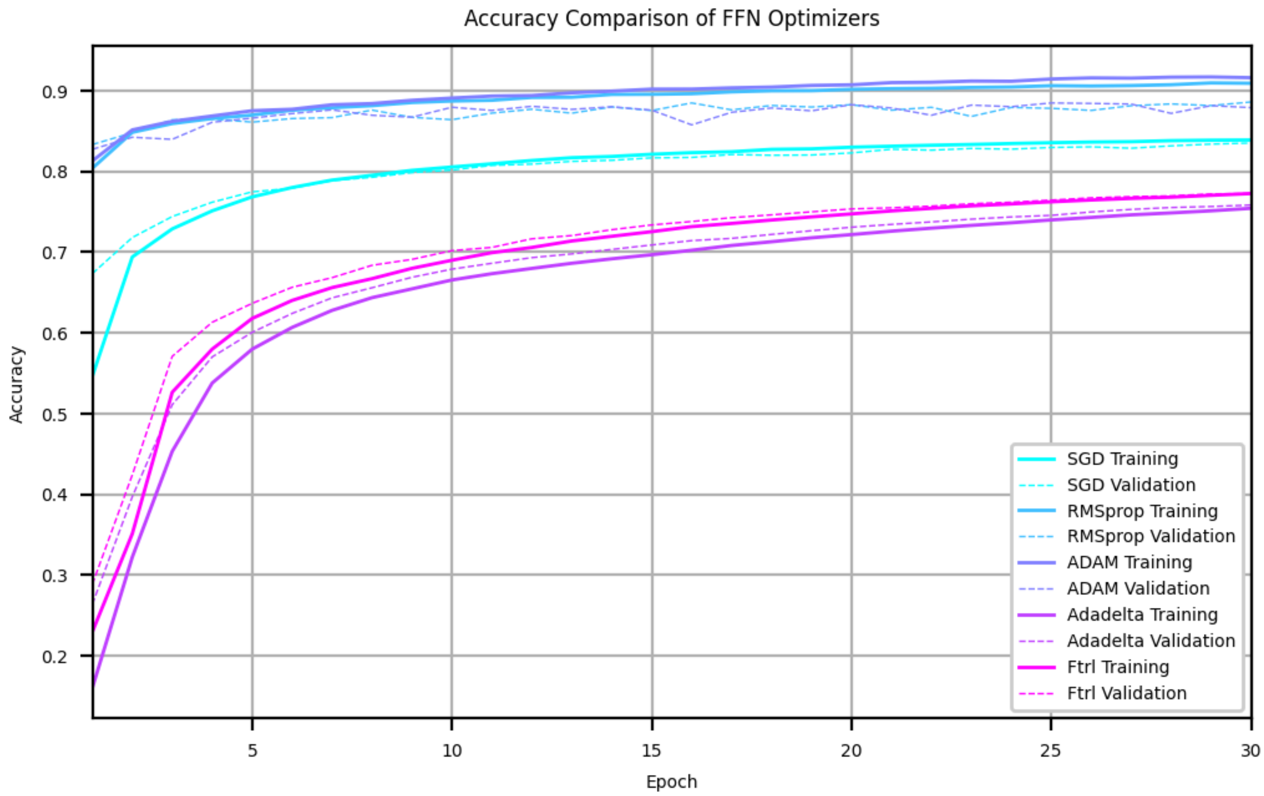
```
cal_accuracy: 0.6837
Epoch 9/30
1500/1500 [==============================] - 1s 664us/step - loss: 0.8573 -
sparse_categorical_accuracy: 0.6800 - val_loss: 0.8311 - val_sparse_categori
cal_accuracy: 0.6911
Epoch 10/30
1500/1500 [==============================] - 1s 674us/step - loss: 0.8282 -
sparse_categorical_accuracy: 0.6900 - val_loss: 0.8058 - val_sparse_categori
cal_accuracy: 0.7019
Epoch 11/30
1500/1500 [==============================] - 1s 674us/step - loss: 0.8047 -
sparse_categorical_accuracy: 0.6994 - val_loss: 0.7848 - val_sparse_categori
cal_accuracy: 0.7059
Epoch 12/30
1500/1500 [==============================] - 1s 677us/step - loss: 0.7850 -
sparse_categorical_accuracy: 0.7059 - val_loss: 0.7679 - val_sparse_categori
cal_accuracy: 0.7166
Epoch 13/30
1500/1500 [==============================] - 1s 675us/step - loss: 0.7682 -
sparse_categorical_accuracy: 0.7139 - val_loss: 0.7525 - val_sparse_categori
cal_accuracy: 0.7206
Epoch 14/30
1500/1500 [==============================] - 1s 666us/step - loss: 0.7535 -
sparse_categorical_accuracy: 0.7197 - val_loss: 0.7391 - val_sparse_categori
cal_accuracy: 0.7278
Epoch 15/30
1500/1500 [==============================] - 1s 673us/step - loss: 0.7404 -
sparse_categorical_accuracy: 0.7254 - val_loss: 0.7272 - val_sparse_categori
cal_accuracy: 0.7337
Epoch 16/30
1500/1500 [==============================] - 1s 665us/step - loss: 0.7286 -
sparse_categorical_accuracy: 0.7317 - val_loss: 0.7165 - val_sparse_categori
cal_accuracy: 0.7380
Epoch 17/30
1500/1500 [==============================] - 1s 678us/step - loss: 0.7180 -
sparse_categorical_accuracy: 0.7356 - val_loss: 0.7068 - val_sparse_categori
cal_accuracy: 0.7427
Epoch 18/30
1500/1500 [==============================] - 1s 673us/step - loss: 0.7082 -
sparse_categorical_accuracy: 0.7399 - val_loss: 0.6983 - val_sparse_categori
cal_accuracy: 0.7462
Epoch 19/30
1500/1500 [==============================] - 1s 672us/step - loss: 0.6992 -
sparse_categorical_accuracy: 0.7437 - val_loss: 0.6898 - val_sparse_categori
cal_accuracy: 0.7500
Epoch 20/30
1500/1500 [==============================] - 1s 675us/step - loss: 0.6909 -
sparse_categorical_accuracy: 0.7474 - val_loss: 0.6821 - val_sparse_categori
cal_accuracy: 0.7536
Epoch 21/30
1500/1500 [==============================] - 1s 669us/step - loss: 0.6833 -
sparse_categorical_accuracy: 0.7511 - val_loss: 0.6751 - val_sparse_categori
cal_accuracy: 0.7550
```

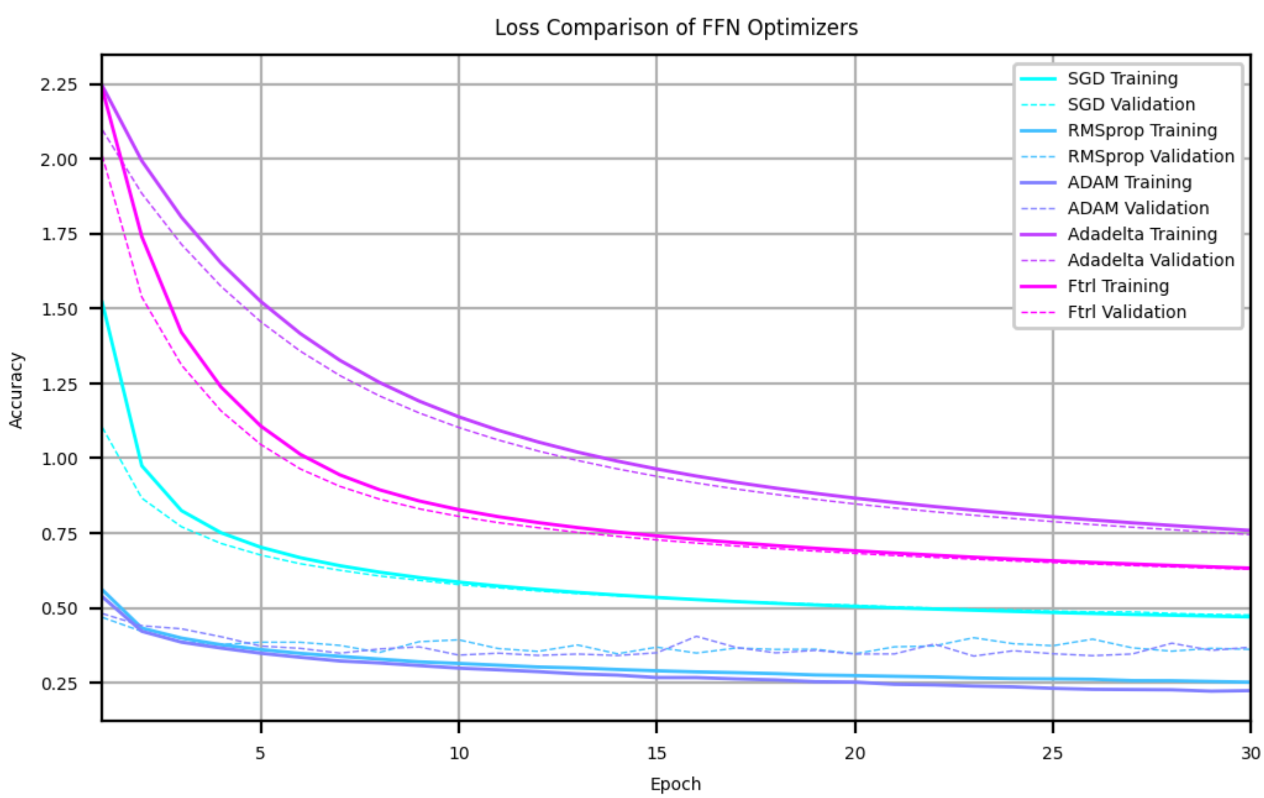
```
Epoch 22/30
1500/1500 [==============================] - 1s 675us/step - loss: 0.6761 -
sparse_categorical_accuracy: 0.7543 - val_loss: 0.6687 - val_sparse_categori
cal_accuracy: 0.7571
Epoch 23/30
1500/1500 [==============================] - 1s 695us/step - loss: 0.6694 -
sparse_categorical_accuracy: 0.7574 - val_loss: 0.6627 - val_sparse_categori
cal_accuracy: 0.7600
Epoch 24/30
1500/1500 [==============================] - 1s 694us/step - loss: 0.6632 -
sparse_categorical_accuracy: 0.7598 - val_loss: 0.6568 - val_sparse_categori
cal_accuracy: 0.7619
Epoch 25/30
1500/1500 [==============================] - 1s 692us/step - loss: 0.6574 -
sparse_categorical_accuracy: 0.7624 - val_loss: 0.6514 - val_sparse_categori
cal_accuracy: 0.7645
Epoch 26/30
1500/1500 [==============================] - 1s 690us/step - loss: 0.6519 -
sparse_categorical_accuracy: 0.7647 - val_loss: 0.6465 - val_sparse_categori
cal_accuracy: 0.7673
Epoch 27/30
1500/1500 [==============================] - 1s 689us/step - loss: 0.6468 -
sparse_categorical_accuracy: 0.7666 - val_loss: 0.6416 - val_sparse_categori
cal_accuracy: 0.7688
Epoch 28/30
1500/1500 [==============================] - 1s 676us/step - loss: 0.6418 -
sparse_categorical_accuracy: 0.7681 - val_loss: 0.6372 - val_sparse_categori
cal_accuracy: 0.7697
Epoch 29/30
1500/1500 [==============================] - 1s 652us/step - loss: 0.6373 -
sparse_categorical_accuracy: 0.7705 - val_loss: 0.6330 - val_sparse_categori
cal_accuracy: 0.7722
Epoch 30/30
1500/1500 [==============================] - 1s 681us/step - loss: 0.6329 -
sparse_categorical_accuracy: 0.7727 - val_loss: 0.6288 - val_sparse_categori
cal_accuracy: 0.7728

 Ftrl Test Accuracy:
313/313 [==============================] - 0s 305us/step
0.7527
```

In [ ]:

```python
# plot training histories for each model on same plot
import matplotlib.cm as cm

fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

colors = cm.cool(np.linspace(0, 1, len(names)))

# add training
for name, history, color in zip(names, histories, colors):
        # training accuracy
        ax.plot(range(1,len(histories[0].history["loss"])+1),
                history.history['sparse_categorical_accuracy'],
                label = f"{name} Training",
```

```
                color = color,
                linewidth=1)

        # validation accuracy
        ax.plot(range(1,len(histories[0].history["loss"])+1),
                history.history['val_sparse_categorical_accuracy'],
                label = f"{name} Validation",
                color = color,
                linewidth=0.5,
                linestyle="--")

        ax.set_title("Accuracy Comparison of FFN Optimizers")
        ax.set_xlabel('Epoch')
        ax.set_xlim(1, len(histories[0].history["loss"]))
        ax.set_ylabel('Accuracy')
        ax.legend()
        ax.legend().get_frame().set_alpha(1.0)

        ax.grid(True)

fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

# add training
for name, history, color in zip(names, histories, colors):
        # training accuracy
        ax.plot(range(1,len(histories[0].history["loss"])+1),
                history.history['loss'],
                label = f"{name} Training",
                color = color,
                linewidth=1)

        # validation accuracy
        ax.plot(range(1,len(histories[0].history["loss"])+1),
                history.history['val_loss'],
                label = f"{name} Validation",
                color = color,
                linewidth=0.5,
                linestyle="--")

        ax.set_title("Loss Comparison of FFN Optimizers")
        ax.set_xlabel('Epoch')
        ax.set_xlim(1, len(histories[0].history["loss"]))
        ax.set_ylabel('Accuracy')
        ax.legend()
        ax.legend().get_frame().set_alpha(1.0)

        ax.grid(True)
```





In [ ]:

```
import pandas as pd

# create summary table
optimizer_summary = pd.DataFrame(
    list(zip(names, train_loss, train_accuracies, test_accuracies)),
    columns=["Optimizer", "Training Loss", "Training Accuracy", "Test Accura
```

```
).set_index("Optimizer")

optimizer_summary
```

Out[ ]:

Optimizer	Training Loss	Training Accuracy	Test Accuracy
SGD	0.469906	0.839021	0.7912
RMSprop	0.251783	0.909292	0.8504
ADAM	0.223793	0.916229	0.8630
Adadelta	0.758649	0.754375	0.7380
Ftrl	0.632887	0.772667	0.7527

Of the five optimizers analyzed, the `Adam` and `RMSProp` optimizers performed the best. The similar performance is most likely due to the `Adam` optimizer implementing the same technique as `RMSProp`, using the second momentum method.

Hidden Units Influence

In [ ]:

```
# create list of hidden units to evaluate
units = [8, 64, 1024]

# instantiate tracking ogjects for analysis
histories = [] # empty list to save history objects for each model
train_loss = [] # save train loss for each mode
train_accuracies = [] # save train accuracy for each model
test_accuracies = [] # save test accuracy for each model

# begin training loop
print("Begin evaluation:")

for unit in units:

    # clear enviornment of previous model
    keras.backend.clear_session()

    # create ffn models using varaible hidden units
    model = model_arch(unit, "LeakyReLU")

    # compile models with Adam optimizer
    model.compile(optimizer=keras.optimizers.legacy.Adam(learning_rate=1e-3)
                  loss='sparse_categorical_crossentropy',
                  metrics=['sparse_categorical_accuracy'],
                  )

    # begin training, for the sake of brevity, number of epochs has been hal
    print(f"\nTraining, {unit} Hidden Units\n")
```

```
    # show model summary and number of traininable parameters
    model.summary()

    history = model.fit(
        x_train_model.astype(np.float32), y_train,
        epochs=30,
        validation_split=0.2,
        shuffle = True
    )

    # evaluate on test data
    print(f"\nTest Accuracy:")
    predicted_softmax = model.predict(x_test.astype(np.float32))
    y_pred = np.argmax(predicted_softmax, axis = 1)

    test_accuracy = accuracy_score(y_test, y_pred)

    print(test_accuracy)

    # append tracking objects
    histories.append(history)
    train_loss.append(history.history["loss"][-1])
    train_accuracies.append(history.history["sparse_categorical_accuracy"][-
    test_accuracies.append(test_accuracy)
```

```
Begin evaluation:

Training, 8 Hidden Units

Model: "units-8_activation-LeakyReLU"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 hidden_layer (Dense)        (None, 8)                 6280

 leaky_re_lu (LeakyReLU)     (None, 8)                 0

 output_layer (Dense)        (None, 10)                90

=================================================================
Total params: 6370 (24.88 KB)
Trainable params: 6370 (24.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/30
1500/1500 [==============================] - 1s 524us/step - loss: 0.7383 -
sparse_categorical_accuracy: 0.7495 - val_loss: 0.5326 - val_sparse_categori
cal_accuracy: 0.8173
Epoch 2/30
1500/1500 [==============================] - 1s 459us/step - loss: 0.4994 -
sparse_categorical_accuracy: 0.8288 - val_loss: 0.4845 - val_sparse_categori
```

```
cal_accuracy: 0.8315
Epoch 3/30
1500/1500 [==============================] - 1s 425us/step - loss: 0.4591 -
sparse_categorical_accuracy: 0.8390 - val_loss: 0.4739 - val_sparse_categori
cal_accuracy: 0.8332
Epoch 4/30
1500/1500 [==============================] - 1s 457us/step - loss: 0.4393 -
sparse_categorical_accuracy: 0.8468 - val_loss: 0.4524 - val_sparse_categori
cal_accuracy: 0.8381
Epoch 5/30
1500/1500 [==============================] - 1s 447us/step - loss: 0.4270 -
sparse_categorical_accuracy: 0.8507 - val_loss: 0.4417 - val_sparse_categori
cal_accuracy: 0.8473
Epoch 6/30
1500/1500 [==============================] - 1s 430us/step - loss: 0.4183 -
sparse_categorical_accuracy: 0.8535 - val_loss: 0.4375 - val_sparse_categori
cal_accuracy: 0.8483
Epoch 7/30
1500/1500 [==============================] - 1s 446us/step - loss: 0.4111 -
sparse_categorical_accuracy: 0.8561 - val_loss: 0.4359 - val_sparse_categori
cal_accuracy: 0.8472
Epoch 8/30
1500/1500 [==============================] - 1s 446us/step - loss: 0.4065 -
sparse_categorical_accuracy: 0.8588 - val_loss: 0.4509 - val_sparse_categori
cal_accuracy: 0.8428
Epoch 9/30
1500/1500 [==============================] - 1s 429us/step - loss: 0.4016 -
sparse_categorical_accuracy: 0.8589 - val_loss: 0.4207 - val_sparse_categori
cal_accuracy: 0.8535
Epoch 10/30
1500/1500 [==============================] - 1s 449us/step - loss: 0.3972 -
sparse_categorical_accuracy: 0.8607 - val_loss: 0.4252 - val_sparse_categori
cal_accuracy: 0.8528
Epoch 11/30
1500/1500 [==============================] - 1s 445us/step - loss: 0.3938 -
sparse_categorical_accuracy: 0.8614 - val_loss: 0.4257 - val_sparse_categori
cal_accuracy: 0.8509
Epoch 12/30
1500/1500 [==============================] - 1s 436us/step - loss: 0.3912 -
sparse_categorical_accuracy: 0.8615 - val_loss: 0.4184 - val_sparse_categori
cal_accuracy: 0.8523
Epoch 13/30
1500/1500 [==============================] - 1s 447us/step - loss: 0.3864 -
sparse_categorical_accuracy: 0.8638 - val_loss: 0.4160 - val_sparse_categori
cal_accuracy: 0.8546
Epoch 14/30
1500/1500 [==============================] - 1s 426us/step - loss: 0.3853 -
sparse_categorical_accuracy: 0.8633 - val_loss: 0.4212 - val_sparse_categori
cal_accuracy: 0.8530
Epoch 15/30
1500/1500 [==============================] - 1s 442us/step - loss: 0.3834 -
sparse_categorical_accuracy: 0.8642 - val_loss: 0.4293 - val_sparse_categori
cal_accuracy: 0.8480
```

```
Epoch 16/30
1500/1500 [==============================] - 1s 459us/step - loss: 0.3812 -
sparse_categorical_accuracy: 0.8653 - val_loss: 0.4185 - val_sparse_categori
cal_accuracy: 0.8503
Epoch 17/30
1500/1500 [==============================] - 1s 426us/step - loss: 0.3786 -
sparse_categorical_accuracy: 0.8673 - val_loss: 0.4206 - val_sparse_categori
cal_accuracy: 0.8512
Epoch 18/30
1500/1500 [==============================] - 1s 444us/step - loss: 0.3766 -
sparse_categorical_accuracy: 0.8669 - val_loss: 0.4266 - val_sparse_categori
cal_accuracy: 0.8509
Epoch 19/30
1500/1500 [==============================] - 1s 447us/step - loss: 0.3746 -
sparse_categorical_accuracy: 0.8669 - val_loss: 0.4120 - val_sparse_categori
cal_accuracy: 0.8562
Epoch 20/30
1500/1500 [==============================] - 1s 428us/step - loss: 0.3745 -
sparse_categorical_accuracy: 0.8672 - val_loss: 0.4181 - val_sparse_categori
cal_accuracy: 0.8530
Epoch 21/30
1500/1500 [==============================] - 1s 447us/step - loss: 0.3724 -
sparse_categorical_accuracy: 0.8691 - val_loss: 0.4138 - val_sparse_categori
cal_accuracy: 0.8526
Epoch 22/30
1500/1500 [==============================] - 1s 430us/step - loss: 0.3701 -
sparse_categorical_accuracy: 0.8685 - val_loss: 0.4169 - val_sparse_categori
cal_accuracy: 0.8556
Epoch 23/30
1500/1500 [==============================] - 1s 449us/step - loss: 0.3689 -
sparse_categorical_accuracy: 0.8693 - val_loss: 0.4168 - val_sparse_categori
cal_accuracy: 0.8545
Epoch 24/30
1500/1500 [==============================] - 1s 442us/step - loss: 0.3680 -
sparse_categorical_accuracy: 0.8686 - val_loss: 0.4138 - val_sparse_categori
cal_accuracy: 0.8538
Epoch 25/30
1500/1500 [==============================] - 1s 427us/step - loss: 0.3665 -
sparse_categorical_accuracy: 0.8702 - val_loss: 0.4111 - val_sparse_categori
cal_accuracy: 0.8581
Epoch 26/30
1500/1500 [==============================] - 1s 442us/step - loss: 0.3654 -
sparse_categorical_accuracy: 0.8706 - val_loss: 0.4169 - val_sparse_categori
cal_accuracy: 0.8541
Epoch 27/30
1500/1500 [==============================] - 1s 442us/step - loss: 0.3635 -
sparse_categorical_accuracy: 0.8705 - val_loss: 0.4220 - val_sparse_categori
cal_accuracy: 0.8547
Epoch 28/30
1500/1500 [==============================] - 1s 429us/step - loss: 0.3640 -
sparse_categorical_accuracy: 0.8705 - val_loss: 0.4118 - val_sparse_categori
cal_accuracy: 0.8558
Epoch 29/30
```

```
1500/1500 [==============================] - 1s 453us/step - loss: 0.3633 -
sparse_categorical_accuracy: 0.8715 - val_loss: 0.4236 - val_sparse_categori
cal_accuracy: 0.8521
Epoch 30/30
1500/1500 [==============================] - 1s 447us/step - loss: 0.3601 -
sparse_categorical_accuracy: 0.8724 - val_loss: 0.4139 - val_sparse_categori
cal_accuracy: 0.8551

Test Accuracy:
313/313 [==============================] - 0s 235us/step
0.7652

Training, 64 Hidden Units

Model: "units-64_activation-LeakyReLU"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 hidden_layer (Dense)        (None, 64)                50240

 leaky_re_lu (LeakyReLU)     (None, 64)                0

 output_layer (Dense)        (None, 10)                650

=================================================================
Total params: 50890 (198.79 KB)
Trainable params: 50890 (198.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/30
1500/1500 [==============================] - 1s 646us/step - loss: 0.5420 -
sparse_categorical_accuracy: 0.8101 - val_loss: 0.4382 - val_sparse_categori
cal_accuracy: 0.8465
Epoch 2/30
1500/1500 [==============================] - 1s 625us/step - loss: 0.4217 -
sparse_categorical_accuracy: 0.8521 - val_loss: 0.4008 - val_sparse_categori
cal_accuracy: 0.8589
Epoch 3/30
1500/1500 [==============================] - 1s 635us/step - loss: 0.3894 -
sparse_categorical_accuracy: 0.8617 - val_loss: 0.3732 - val_sparse_categori
cal_accuracy: 0.8679
Epoch 4/30
1500/1500 [==============================] - 1s 625us/step - loss: 0.3648 -
sparse_categorical_accuracy: 0.8694 - val_loss: 0.3754 - val_sparse_categori
cal_accuracy: 0.8670
Epoch 5/30
1500/1500 [==============================] - 1s 632us/step - loss: 0.3502 -
sparse_categorical_accuracy: 0.8737 - val_loss: 0.3807 - val_sparse_categori
cal_accuracy: 0.8615
Epoch 6/30
1500/1500 [==============================] - 1s 629us/step - loss: 0.3386 -
```

```
sparse_categorical_accuracy: 0.8765 - val_loss: 0.3710 - val_sparse_categori
cal_accuracy: 0.8637
Epoch 7/30
1500/1500 [==============================] - 1s 631us/step - loss: 0.3225 -
sparse_categorical_accuracy: 0.8821 - val_loss: 0.3481 - val_sparse_categori
cal_accuracy: 0.8734
Epoch 8/30
1500/1500 [==============================] - 1s 633us/step - loss: 0.3128 -
sparse_categorical_accuracy: 0.8847 - val_loss: 0.3562 - val_sparse_categori
cal_accuracy: 0.8737
Epoch 9/30
1500/1500 [==============================] - 1s 632us/step - loss: 0.3070 -
sparse_categorical_accuracy: 0.8880 - val_loss: 0.3754 - val_sparse_categori
cal_accuracy: 0.8652
Epoch 10/30
1500/1500 [==============================] - 1s 633us/step - loss: 0.3002 -
sparse_categorical_accuracy: 0.8897 - val_loss: 0.3448 - val_sparse_categori
cal_accuracy: 0.8769
Epoch 11/30
1500/1500 [==============================] - 1s 627us/step - loss: 0.2909 -
sparse_categorical_accuracy: 0.8935 - val_loss: 0.3589 - val_sparse_categori
cal_accuracy: 0.8725
Epoch 12/30
1500/1500 [==============================] - 1s 632us/step - loss: 0.2842 -
sparse_categorical_accuracy: 0.8952 - val_loss: 0.3515 - val_sparse_categori
cal_accuracy: 0.8738
Epoch 13/30
1500/1500 [==============================] - 1s 630us/step - loss: 0.2799 -
sparse_categorical_accuracy: 0.8967 - val_loss: 0.3382 - val_sparse_categori
cal_accuracy: 0.8811
Epoch 14/30
1500/1500 [==============================] - 1s 632us/step - loss: 0.2751 -
sparse_categorical_accuracy: 0.8982 - val_loss: 0.3491 - val_sparse_categori
cal_accuracy: 0.8729
Epoch 15/30
1500/1500 [==============================] - 1s 624us/step - loss: 0.2711 -
sparse_categorical_accuracy: 0.8999 - val_loss: 0.3476 - val_sparse_categori
cal_accuracy: 0.8772
Epoch 16/30
1500/1500 [==============================] - 1s 631us/step - loss: 0.2632 -
sparse_categorical_accuracy: 0.9023 - val_loss: 0.3466 - val_sparse_categori
cal_accuracy: 0.8808
Epoch 17/30
1500/1500 [==============================] - 1s 627us/step - loss: 0.2596 -
sparse_categorical_accuracy: 0.9040 - val_loss: 0.3373 - val_sparse_categori
cal_accuracy: 0.8838
Epoch 18/30
1500/1500 [==============================] - 1s 655us/step - loss: 0.2571 -
sparse_categorical_accuracy: 0.9045 - val_loss: 0.3425 - val_sparse_categori
cal_accuracy: 0.8788
Epoch 19/30
1500/1500 [==============================] - 1s 627us/step - loss: 0.2540 -
sparse_categorical_accuracy: 0.9057 - val_loss: 0.3332 - val_sparse_categori
```

```
cal_accuracy: 0.8824
Epoch 20/30
1500/1500 [==============================] - 1s 637us/step - loss: 0.2498 -
sparse_categorical_accuracy: 0.9078 - val_loss: 0.3735 - val_sparse_categori
cal_accuracy: 0.8699
Epoch 21/30
1500/1500 [==============================] - 1s 636us/step - loss: 0.2449 -
sparse_categorical_accuracy: 0.9088 - val_loss: 0.3396 - val_sparse_categori
cal_accuracy: 0.8834
Epoch 22/30
1500/1500 [==============================] - 1s 629us/step - loss: 0.2405 -
sparse_categorical_accuracy: 0.9112 - val_loss: 0.3499 - val_sparse_categori
cal_accuracy: 0.8776
Epoch 23/30
1500/1500 [==============================] - 1s 635us/step - loss: 0.2367 -
sparse_categorical_accuracy: 0.9131 - val_loss: 0.3708 - val_sparse_categori
cal_accuracy: 0.8765
Epoch 24/30
1500/1500 [==============================] - 1s 640us/step - loss: 0.2334 -
sparse_categorical_accuracy: 0.9134 - val_loss: 0.3501 - val_sparse_categori
cal_accuracy: 0.8824
Epoch 25/30
1500/1500 [==============================] - 1s 642us/step - loss: 0.2337 -
sparse_categorical_accuracy: 0.9134 - val_loss: 0.3629 - val_sparse_categori
cal_accuracy: 0.8777
Epoch 26/30
1500/1500 [==============================] - 1s 640us/step - loss: 0.2297 -
sparse_categorical_accuracy: 0.9147 - val_loss: 0.3801 - val_sparse_categori
cal_accuracy: 0.8723
Epoch 27/30
1500/1500 [==============================] - 1s 641us/step - loss: 0.2266 -
sparse_categorical_accuracy: 0.9162 - val_loss: 0.3602 - val_sparse_categori
cal_accuracy: 0.8778
Epoch 28/30
1500/1500 [==============================] - 1s 618us/step - loss: 0.2248 -
sparse_categorical_accuracy: 0.9180 - val_loss: 0.3919 - val_sparse_categori
cal_accuracy: 0.8651
Epoch 29/30
1500/1500 [==============================] - 1s 646us/step - loss: 0.2243 -
sparse_categorical_accuracy: 0.9160 - val_loss: 0.3542 - val_sparse_categori
cal_accuracy: 0.8811
Epoch 30/30
1500/1500 [==============================] - 1s 669us/step - loss: 0.2184 -
sparse_categorical_accuracy: 0.9192 - val_loss: 0.3671 - val_sparse_categori
cal_accuracy: 0.8783

Test Accuracy:
313/313 [==============================] - 0s 384us/step
0.8526

Training, 1024 Hidden Units

Model: "units-1024_activation-LeakyReLU"
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 hidden_layer (Dense)        (None, 1024)              803840

 leaky_re_lu (LeakyReLU)     (None, 1024)              0

 output_layer (Dense)        (None, 10)                10250

=================================================================
Total params: 814090 (3.11 MB)
Trainable params: 814090 (3.11 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Epoch 1/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.5212 - sp
arse_categorical_accuracy: 0.8164 - val_loss: 0.4235 - val_sparse_categorica
l_accuracy: 0.8499
Epoch 2/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.4192 - sp
arse_categorical_accuracy: 0.8493 - val_loss: 0.4915 - val_sparse_categorica
l_accuracy: 0.8162
Epoch 3/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.3893 - sp
arse_categorical_accuracy: 0.8608 - val_loss: 0.3805 - val_sparse_categorica
l_accuracy: 0.8652
Epoch 4/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.3693 - sp
arse_categorical_accuracy: 0.8676 - val_loss: 0.3787 - val_sparse_categorica
l_accuracy: 0.8681
Epoch 5/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.3530 - sp
arse_categorical_accuracy: 0.8721 - val_loss: 0.3649 - val_sparse_categorica
l_accuracy: 0.8718
Epoch 6/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.3373 - sp
arse_categorical_accuracy: 0.8774 - val_loss: 0.3542 - val_sparse_categorica
l_accuracy: 0.8722
Epoch 7/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.3277 - sp
arse_categorical_accuracy: 0.8806 - val_loss: 0.3822 - val_sparse_categorica
l_accuracy: 0.8714
Epoch 8/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.3124 - sp
arse_categorical_accuracy: 0.8865 - val_loss: 0.3824 - val_sparse_categorica
l_accuracy: 0.8705
Epoch 9/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.3102 - sp
arse_categorical_accuracy: 0.8855 - val_loss: 0.3504 - val_sparse_categorica
l_accuracy: 0.8769
Epoch 10/30
```

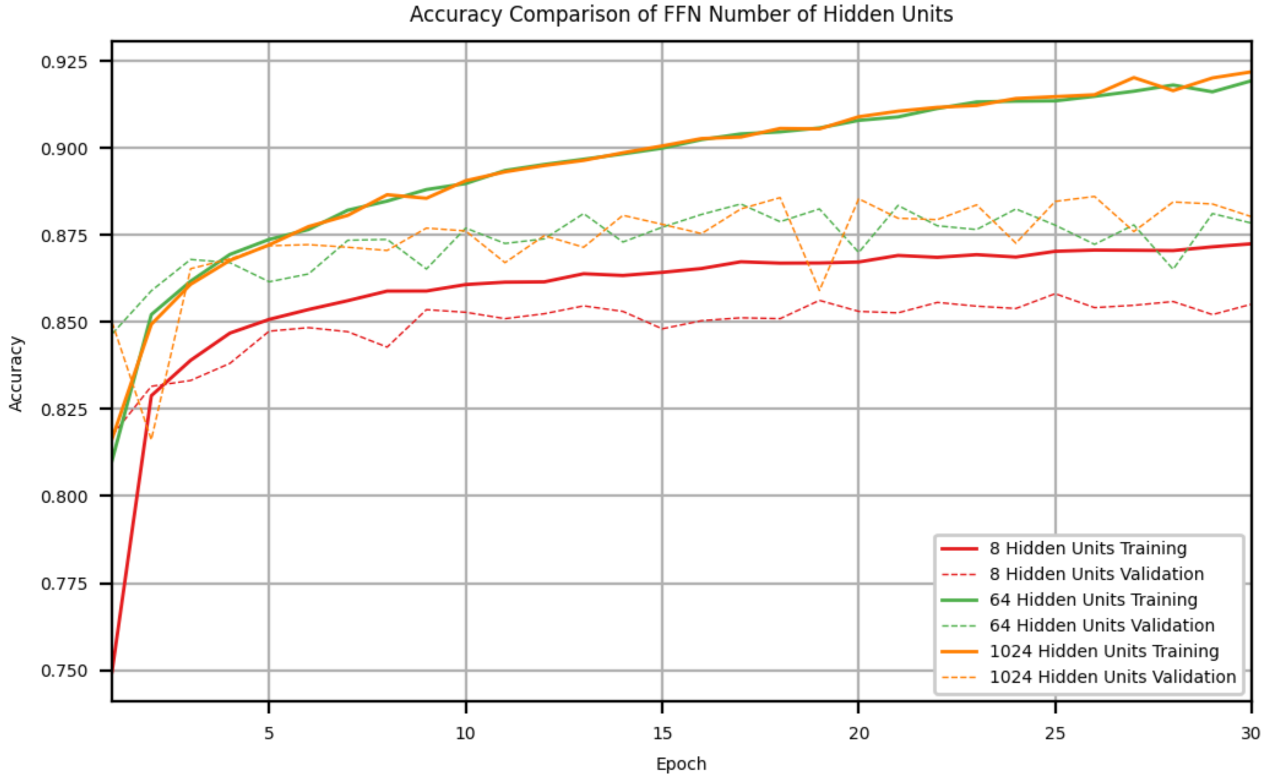
```
1500/1500 [==============================] - 3s 2ms/step - loss: 0.3001 - sp
arse_categorical_accuracy: 0.8905 - val_loss: 0.3786 - val_sparse_categorica
l_accuracy: 0.8761
Epoch 11/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2913 - sp
arse_categorical_accuracy: 0.8931 - val_loss: 0.3994 - val_sparse_categorica
l_accuracy: 0.8670
Epoch 12/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2886 - sp
arse_categorical_accuracy: 0.8949 - val_loss: 0.3640 - val_sparse_categorica
l_accuracy: 0.8748
Epoch 13/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2832 - sp
arse_categorical_accuracy: 0.8964 - val_loss: 0.3867 - val_sparse_categorica
l_accuracy: 0.8714
Epoch 14/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2765 - sp
arse_categorical_accuracy: 0.8985 - val_loss: 0.3561 - val_sparse_categorica
l_accuracy: 0.8806
Epoch 15/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2747 - sp
arse_categorical_accuracy: 0.9005 - val_loss: 0.3644 - val_sparse_categorica
l_accuracy: 0.8780
Epoch 16/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2628 - sp
arse_categorical_accuracy: 0.9026 - val_loss: 0.3844 - val_sparse_categorica
l_accuracy: 0.8753
Epoch 17/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2664 - sp
arse_categorical_accuracy: 0.9030 - val_loss: 0.3542 - val_sparse_categorica
l_accuracy: 0.8824
Epoch 18/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2589 - sp
arse_categorical_accuracy: 0.9055 - val_loss: 0.3453 - val_sparse_categorica
l_accuracy: 0.8857
Epoch 19/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2531 - sp
arse_categorical_accuracy: 0.9054 - val_loss: 0.4316 - val_sparse_categorica
l_accuracy: 0.8590
Epoch 20/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2482 - sp
arse_categorical_accuracy: 0.9089 - val_loss: 0.3606 - val_sparse_categorica
l_accuracy: 0.8853
Epoch 21/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2441 - sp
arse_categorical_accuracy: 0.9105 - val_loss: 0.3888 - val_sparse_categorica
l_accuracy: 0.8798
Epoch 22/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2401 - sp
arse_categorical_accuracy: 0.9116 - val_loss: 0.3819 - val_sparse_categorica
l_accuracy: 0.8793
Epoch 23/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2401 - sp
```

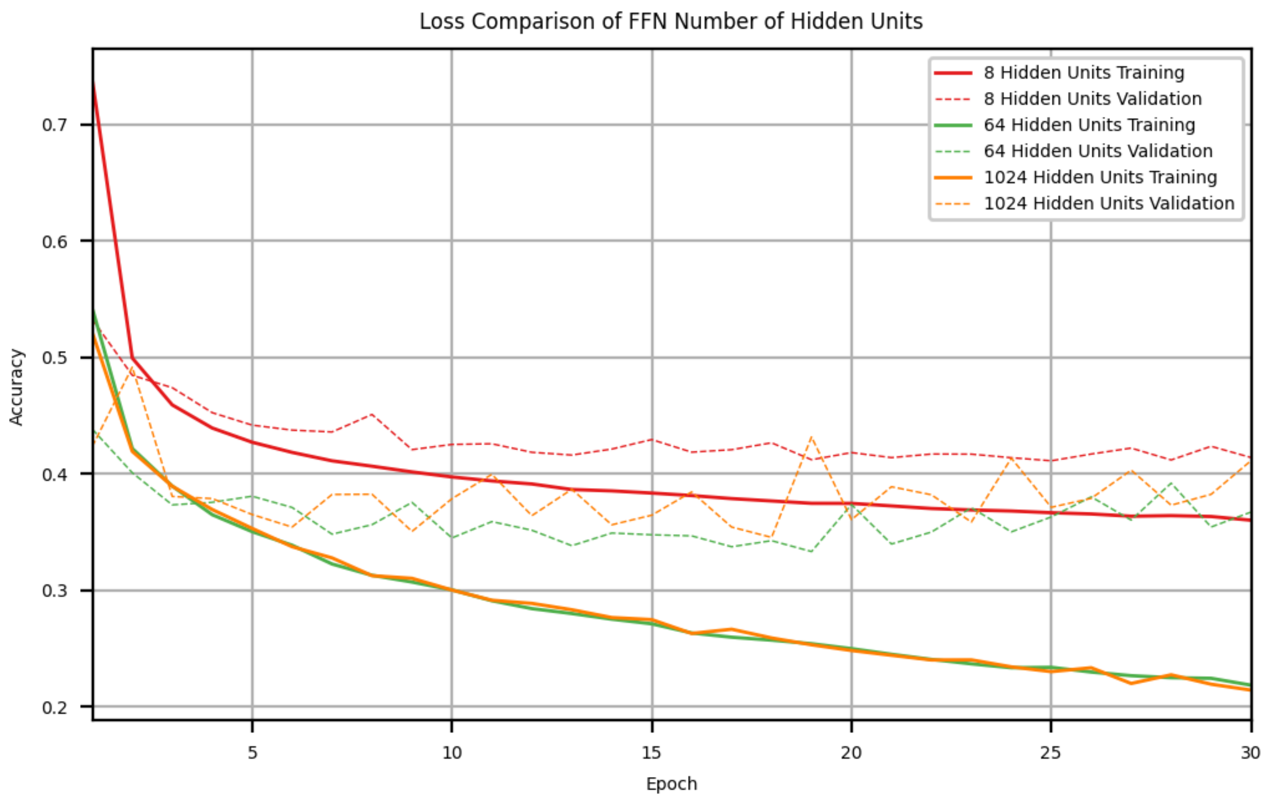
```
arse_categorical_accuracy: 0.9121 - val_loss: 0.3585 - val_sparse_categorica
l_accuracy: 0.8836
Epoch 24/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2341 - sp
arse_categorical_accuracy: 0.9141 - val_loss: 0.4133 - val_sparse_categorica
l_accuracy: 0.8726
Epoch 25/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2300 - sp
arse_categorical_accuracy: 0.9146 - val_loss: 0.3711 - val_sparse_categorica
l_accuracy: 0.8846
Epoch 26/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2334 - sp
arse_categorical_accuracy: 0.9152 - val_loss: 0.3789 - val_sparse_categorica
l_accuracy: 0.8860
Epoch 27/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2198 - sp
arse_categorical_accuracy: 0.9201 - val_loss: 0.4031 - val_sparse_categorica
l_accuracy: 0.8759
Epoch 28/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2273 - sp
arse_categorical_accuracy: 0.9164 - val_loss: 0.3730 - val_sparse_categorica
l_accuracy: 0.8844
Epoch 29/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2192 - sp
arse_categorical_accuracy: 0.9200 - val_loss: 0.3823 - val_sparse_categorica
l_accuracy: 0.8838
Epoch 30/30
1500/1500 [==============================] - 3s 2ms/step - loss: 0.2141 - sp
arse_categorical_accuracy: 0.9218 - val_loss: 0.4112 - val_sparse_categorica
l_accuracy: 0.8802

Test Accuracy:
313/313 [==============================] - 0s 670us/step
0.857
```

In [ ]:

```python
# plot training histories for each model on same plot
fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

colors = cm.Set1(np.linspace(0, 1, len(names)))

# add training
for unit, history, color in zip(units, histories, colors):
        # training accuracy
        ax.plot(range(1,len(histories[0].history["loss"])+1),
                history.history['sparse_categorical_accuracy'],
                label = f"{unit} Hidden Units Training",
                color = color,
                linewidth=1)

        # validation accuracy
        ax.plot(range(1,len(histories[0].history["loss"])+1),
                history.history['val_sparse_categorical_accuracy'],
                label = f"{unit} Hidden Units Validation",
                color = color,
```

```
                linewidth=0.5,
                linestyle="--")

        ax.set_title("Accuracy Comparison of FFN Number of Hidden Units")
        ax.set_xlabel('Epoch')
        ax.set_xlim(1, len(histories[0].history["loss"]))
        ax.set_ylabel('Accuracy')
        ax.legend()
        ax.legend().get_frame().set_alpha(1.0)

        ax.grid(True)

fig, ax = plt.subplots(figsize=(6,3.5), dpi=200)

# add training
for unit, history, color in zip(units, histories, colors):
        # training accuracy
        ax.plot(range(1,len(histories[0].history["loss"])+1),
                history.history['loss'],
                label = f"{unit} Hidden Units Training",
                color = color,
                linewidth=1)

        # validation accuracy
        ax.plot(range(1,len(histories[0].history["loss"])+1),
                history.history['val_loss'],
                label = f"{unit} Hidden Units Validation",
                color = color,
                linewidth=0.5,
                linestyle="--")

        ax.set_title("Loss Comparison of FFN Number of Hidden Units")
        ax.set_xlabel('Epoch')
        ax.set_xlim(1, len(histories[0].history["loss"]))
        ax.set_ylabel('Accuracy')
        ax.legend()
        ax.legend().get_frame().set_alpha(1.0)

        ax.grid(True)
```





In [ ]:

```
# create summary table
optimizer_summary = pd.DataFrame(
    list(zip(units, train_loss, train_accuracies, test_accuracies)),
    columns=["Hidden Units", "Training Loss", "Training Accuracy", "Test Acc
).set_index("Hidden Units")

optimizer_summary
```

Out[ ]:

Hidden Units	Training Loss	Training Accuracy	Test Accuracy
8	0.360062	0.872417	0.7652
64	0.218399	0.919229	0.8526
1024	0.214138	0.921792	0.8570

In general, increasing the number of hidden units increases test accuracy up to a point. But from the summary table and loss plot it is apparent that these increases are diminishing in value and prone to overfitting.

6. Team Information

Team Github: Team4

Team Members:

- Arheum Kim: ahreum239
- Isaac Salvador: isalva2
- Sadjad Bazarnovi: sadjad33