Machine Learning 2 Coursework 1

24/10/2024 - 7/11/2024
Nikola Cicovic

Question 1 a)

We pick points $x^{(1)} = 2, y^{(1)} = 1 \text{ and } x^{(2)} = 5, y^{(2)} = 0$

for $x^{(1)}$ we require:

$$\begin{aligned} &\sigma(\xi(2)) \geq 0.5 \\ \implies &\frac{1}{1+e^{-(2w+b)}} \geq 0.5 \\ \implies &1 \geq 0.5 + 0.5e^{-(2w+b)} \quad \text{as } e^y > 0 \; \forall y \\ \implies &1 \geq e^{-(2w+b)} \\ \implies &0 \geq -(2w+b) \quad \text{as } \log \text{is monotone} \\ \implies &0 \leq 2w+b \\ \implies &-2w \leq b \end{aligned}$$

Repeating this for $x^{(2)} = 5, y^{(2)} = 0$ we get

$$\begin{aligned} &\sigma(\xi(5)) < 0.5 \\ \implies &0 > 5w+b \\ \implies &-5w > b \end{aligned}$$

Which along with the previous equation implies that

$$\begin{gathered} 5w < -b \leq 2w \\ \text{and} \\ 5w+b < 2w+b \\ 3w < 0 \\ w < 0 \end{gathered}$$

Now we try the third point $x^{(3)} = 6, y^{(3)} = 1$

$$\begin{aligned} &\sigma(\xi(6)) \geq 0.5 \\ \implies &-6w-b \leq 0 \\ \implies &-b \leq 6w \end{aligned}$$

However we already have that $-b > 5w$, and $w < 0$ tells us that $5w \geq 6w$, so not possible!

Question 1 b)

$$\begin{aligned} \xi(t) &= b + w_1\theta_1(t) + w_2\theta_2(t) \\ &= b + w_1 t + w_2 t^2 \end{aligned}$$

Take points
$(x^{(i)}, y^{(i)}) = (2,1), (5,0), (6,1)$

for $(x^{(1)}, y^{(1)})$ we get equation (1)

$$\sigma(\xi(2)) = \frac{1}{1+e^{-(b+2w_1+4w_2)}} \geq 0.5$$
$$\implies 0 \leq b + 2w_1 + 4w_2$$

for $(x^{(2)}, y^{(2)})$ we get equation (2)

$$\sigma(\xi(5)) = \frac{1}{1+e^{-(b+5w_1+25w_2)}} < 0.5$$
$$\implies 0 > b + 5w_1 + 25w_2$$

for $(x^{(3)}, y^{(3)})$ we get equation (3)

$$\sigma(\xi(6)) = \frac{1}{1+e^{-(b+6w_1+36w_2)}} \geq 0.5$$
$$\implies 0 \leq b + 6w_1 + 36w_2$$

If we add equations (1) and (2) and divide through by 2 we obtain $b \geq -4w_1 - 20w_2$ which we can, along with (2) arrange for w_2 and compare to get:

$$\frac{-b-4w_1}{20} < \frac{-b-5w_1}{25}$$
$$\implies -25b - 100w_1 < -20b - 100w_1$$
$$\implies b > 0$$

Giving us our first restriction. Now we only consider 2 possibilities values where (1) and (2) are satisfied and where (3) and (2) are satisfied: If (1) and (2) are satisfied we obtain:

$$(A) ::: \frac{-b-2w_1}{4} \leq w_2 < \frac{-b-5w_1}{25}$$

and if (2) and (3) are satisfied we obtain:

$$(B) ::: \frac{-b-6w_1}{36} \leq w_2 < \frac{-b-5w_1}{25}$$

From equation (A) we also get: $\frac{-b-2w_1}{4} < \frac{-b-5w_1}{25}$ which can be rearranged to give:

$$-\frac{7}{10}b < w_1$$

Now if we take that $\frac{-b-2w_1}{4} \geq \frac{-b-6w_1}{36}$ (A) will still be satisfied as $w_2 \geq \frac{-b-2w_1}{4}$ regardless we get:

$$\frac{-b-2w_1}{4} \geq \frac{-b-6w_1}{36}$$
$$\implies -36b - 72w_1 \geq -4b - 24w_1$$
$$\implies w_1 \leq -\frac{2b}{3}$$

giving us an upper bound on w_1 From equation (B) we also get: $\frac{-b-6w_1}{36} < \frac{-b-5w_1}{25}$ which can be rearranged to give:

$$w_1 < -\frac{11b}{30}$$

Now if we instead take that $\frac{-b-2w_1}{4} < \frac{-b-6w_1}{36}$ (B) will be satisfied for similar reason as before, we get:

$$\begin{aligned} \frac{-b-2w_1}{4} &< \frac{-b-6w_1}{36} \\ \implies \quad -36b - 72w_1 &< -4b - 24w_1 \\ \implies \quad w_1 &> -\frac{2b}{3} \end{aligned}$$

giving us an upper bound on w_1 for case (B) If we combine all of these conditions we get 2 cases:

$$b > 0; :: -\frac{7b}{10} < w_1 \leq -\frac{2b}{3}; ::: \frac{-b-2w_1}{4} \leq w_2 < \frac{-b-5w_1}{25}$$

$$\text{and}$$

$$b > 0; :: -\frac{2b}{3} < w_1 < -\frac{11b}{30}; ::: \frac{-b-6w_1}{36} \leq w_2 < \frac{-b-5w_1}{25}$$

Pick $b = 60 > 0$

That gives us a condition on w_1

$-42 < w_1 \leq -40$

Pick $w_1 = -41$ which satisfies it to give us a condition on w_2:

$\frac{11}{2} \leq w_2 < \frac{145}{25}$

So pick $w_2 = \frac{28}{5}$ which satisfies above condition

so pick $b = 60; w_1 = -41; w_2 = \frac{28}{5}$

If you need to be convinced that it works: https://www.desmos.com/calculator/ep2vmwf2ah

Question 1 c)

We consider points $(1,0); (2,1); (3,0); (4,1); (5,0)$.

and now function ξ is

$$\xi(t) = w_0 + w_1 t + w_2 t^2 + w_3 t^3 + w_4 t^4 \tag{1}$$

by subing each of those points into $\sigma(\xi(t))$ we get

$$\sigma(\xi(1)) < 0.5$$
$$\sigma(\xi(2)) \geq 0.5$$
$$\sigma(\xi(3)) < 0.5$$
$$\sigma(\xi(4)) \geq 0.5$$
$$\sigma(\xi(5)) < 0.5$$

which simplify to following linear inequalities

$$w_0 + w_1 + w_2 + w_3 + w_4 < 0$$
$$w_0 + 2w_1 + 4w_2 + 8w_3 + 16w_4 \geq 0$$
$$w_0 + 3w_1 + 9w_2 + 27w_3 + 81w_4 < 0$$
$$w_0 + 4w_1 + 16w_2 + 64w_3 + 256w_4 \geq 0$$
$$w_0 + 5w_1 + 25w_2 + 125w_3 + 625w_4 < 0$$

As we only require a set of points for w_i we just make each of them an equality and pick any value on the right side that satisfy the inequality, say -1 and 1. We use this to form a linear equation of the form $Aw = b$.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \\ 1 & 5 & 25 & 125 & 625 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_3 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

gives

$$\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_3 \end{pmatrix} = \begin{pmatrix} -31 \\ 56 \\ \frac{-100}{3} \\ 8 \\ -\frac{2}{3} \end{pmatrix}$$

The minimum number n required to classify any 5 points is 4.

We can show this by looking at the number of turning points that the graph has. We require at least 3 turning points so that you can find a set of parameters w_i to classify all 5 points correctly as they are alternating between 0 and 1 every time.

The reason we only require 3 turning points is because the first and last point are "given" to us automatically:

$$\lim_{t\to\infty} \sigma(\xi(t)) = 0 = \lim_{t\to-\infty} \sigma(\xi(t)) \text{ for } w_n < 0 \text{ and n even}$$

Similar is true for $w_n > 0$ except the limit is 1 and we don't care about those as first and last points should be labelled 0, and similarly n is even, as otherwise one limit would be 1 and the other 0.

To prove there are 3 turning poitns we take the derivative:

$$\frac{d}{dt}\sigma(\xi(t))$$
$$= \frac{d}{dt}\frac{1}{1+e^{\sum_{i=0}^{n}-w_it^i}}$$
$$= (-1)(1+e^{\sum_{i=0}^{n}-w_it^i})^{-2}\frac{d}{dt}(1+e^{\sum_{i=0}^{n}-w_it^i}) \text{ by 2 chain rules}$$
$$= \frac{(\sum_{i=0}^{n-1}-w_{i+1}t^i)(1+e^{\sum_{i=0}^{n}-w_it^i})}{(1+e^{\sum_{i=0}^{n}-w_it^i})^2}$$

Now setting this to be equal to 0:

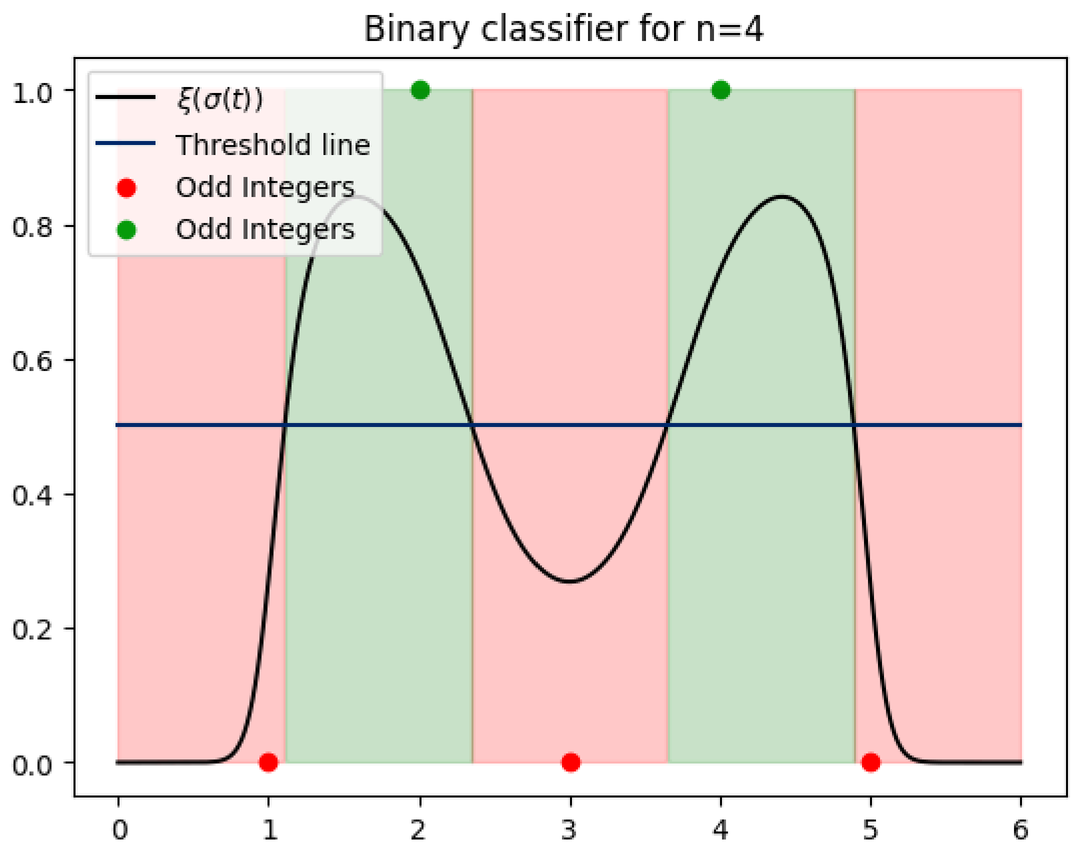
$$\frac{(\sum_{i=0}^{n-1}-w_{i+1}t^i)(1+e^{\sum_{i=0}^{n}-w_it^i})}{(1+e^{\sum_{i=0}^{n}-w_it^i})^2} = 0$$
$$\implies (\sum_{i=0}^{n-1}-w_{i+1}t^i)(1+e^{\sum_{i=0}^{n}-w_it^i}) = 0$$

and as $1+e^x > 0$ as as $e^x > 0$ we know that only way above equation can be satisfied is with

$$\sum_{i=0}^{n-1}-w_{i+1}t^i = 0$$

This is just a n-1 degree polynomial which has n-1 roots and as we require that we have 3 turning points so n = 4

In [28]:
```
import matplotlib.pyplot as plt
import numpy as np

def sigmoid(x):
    return 1 / ( 1+ np.exp(-x))

def polynomial(w,t):
    return w[0] + t*w[1] + (t**2)*w[2] + (t**3)*w[3] + (t**4) * w[4]

x = np.linspace(0,6,1000)
y = sigmoid(polynomial([-31,56,-100/3,8,-2/3],x))


plt.plot(x,y,label = r'$\xi(\sigma(t))$', color = "#000000")
plt.plot([6/1000*i for i in range(1000)],[0.5 for i in range(1000)],color = "#00
plt.scatter([1,3,5],[0,0,0],color =  '#FF0000', label = "Odd Integers")
plt.scatter([2,4],[1,1],color =  '#099B0E', label = "Odd Integers")

plt.fill_between(x,0,1, where = y < 0.5,alpha = 0.2,color = "red")
plt.fill_between(x,0,1, where = y > 0.5,alpha = 0.2, color = "green")
plt.title("Binary classifier for n=4 ")
plt.legend(loc = "upper left")
plt.show()
```



As can be seen above the areas shaded red is where the function $\sigma(\xi(t)) < 0.5$ and area shaded red $\sigma(\xi(t)) \geq 0.5$ for values of w_i chosen above. The red points at x coordinates are the odd points and they clearly lie within the red area and green points lie within the green area so the values chosen work.

```
In [29]: for i in range(1,6):
             result = sigmoid(polynomial([-31,56,-100/3,8,-2/3],i))
             if result >= 0.5:
                 print(f"For digit {i} the model classified it as 1.")
             else:
                 print(f"For digit {i} the model classified it as 0.")
```

```
For digit 1 the model classified it as 0.
For digit 2 the model classified it as 1.
For digit 3 the model classified it as 0.
For digit 4 the model classified it as 1.
For digit 5 the model classified it as 0.
```

Advantages: Increasing n increases the complexity of the model so the model can correctly classify more points. For values of n larger than the number of points minus one you can always find a rule that will correctly classify all of them regardless of the n points. For any other values of n, increasing it, means that more points can be classified correctly if not all (depending on the arrangement).

Disadvantages: Using polynomial features will most likely lead to overfitting for large values of n, so while the model will work for the given points for a set of w_i, applying that model to any other set will likely lead to an incorrect classification, in other words it generalises poorly.

Increasing the number of parameters, makes it more difficult to find the right features.

Question 1 d)

Consider the function
$\sigma(t) = \frac{cos(t)}{2} + \frac{1}{2}$

our binary classifier is

$$\sigma(\xi(t)) = \frac{cos(wt+b)}{2} + \frac{1}{2}$$

and by picking $w = \pi; b = 0$ we get

$$\sigma(\xi(t)) = \frac{cos(\pi t)}{2} + \frac{1}{2}$$

which is $1 \geq 0.5$ for any even number as it would be multiple of 2π and $0 < 0.5$ for any odd number as it would be an odd number of πs So this function satisfies one of $w \text{ or } b$ to be 0 and still correctly classify all points.

Question 2

Question 2a)

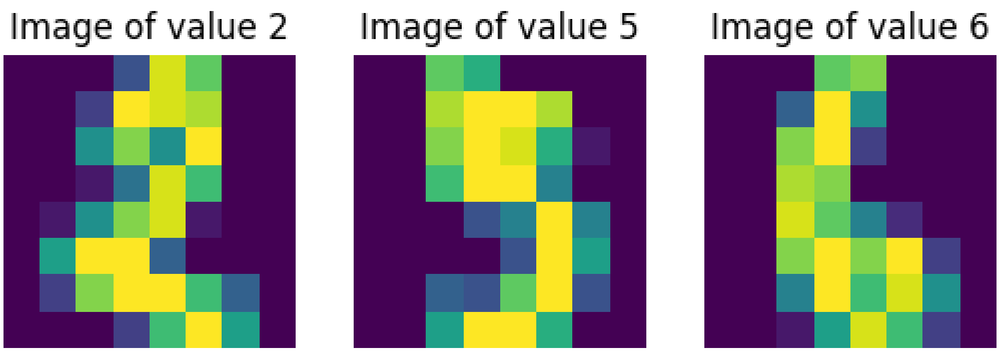
In [30]:
```python
#We use the snippet code to load all of the data
import numpy as np
import matplotlib.pyplot as plt
from cw_utils import (dataloader,initialize_with_zeros,
                      sigmoid, propagate, optimize, predict, model)

X_all, z_all = dataloader()
print(X_all.shape)
print(z_all.shape)
```

```
(1797, 64)
(1797,)
```

In [31]:
```python
print(z_all[0:15])
#They go in order so that index is the same as value for first 10 digits
```

```
[0 1 2 3 4 5 6 7 8 9 0 1 2 3 4]
```

In [32]:
```python
ns = [2,5,6]


fig, ax = plt.subplots(1,len(ns))
for i in range(len(ns)):
    ax[i].imshow(X_all[ns[i]].reshape(8,8))
    ax[i].axis('off')
    ax[i].set_title(f"Image of value {z_all[ns[i]]}")
```



Quesiton 2 b) i)

We use the function chosen in 1 d) $\sigma(\xi(t)) = \frac{cos(\pi t)+1}{2}$ as we know it works.

And we can check whether it is right by using the mod function

```
In [33]: y_all = (np.cos(np.pi * z_all) + 1)/ 2  #Broadcasting so able to this
         print(f"Calculated labels: {y_all}")

         #We use our sigma as in question 1 d) which is
         y_check = (z_all + 1) % 2
         print(f"Check labels: {y_check}")

         print(f"the difference between them is {np.sum(np.array(y_check)- y_all)}") # ca
```

```
Calculated labels: [1. 0. 1. ... 1. 0. 1.]
Check labels: [1 0 1 ... 1 0 1]
the difference between them is 0.0
```

Quesiton 2 b) ii)

```
In [34]: np.random.seed(0)
         #zip the values into tuple pairs, shuffle and unzip
         xandy = list(zip(X_all, y_all))
         np.random.shuffle(xandy)
         X_all_rand, y_all_rand = list(zip(*xandy))


         X_all_rand_arr = np.array(X_all_rand)
         y_all_rand_arr = np.array(y_all_rand).reshape(1,1797)

         #split into training and test
         X_train = X_all_rand_arr[:1258].T
         X_test = X_all_rand_arr[1258:].T
         y_train = y_all_rand_arr[:,:1258]
         y_test = y_all_rand_arr[:,1258:]

         #check if the values are right
         assert(X_train.shape == (64,1258))
         assert(X_test.shape == (64, 539))
         assert(y_train.shape == (1, 1258 ))
         assert(y_test.shape == (1, 539))
```

Question b) iii)

In [35]:
```
max_of_maxes = [np.max(x) for x in X_train]
max_colour_val = max(max_of_maxes)
min_of_mins = [np.min(x) for x in X_train]
min_colour_val = max(min_of_mins)



normalised_X_train = [ (x - min_colour_val) / (max_colour_val - min_colour_val)
normalised_X_test = [ (x - min_colour_val) / (max_colour_val - min_colour_val) f
normalised_X_train = np.array(normalised_X_train)
normalised_X_test = np.array(normalised_X_test)


assert(normalised_X_train.shape == (64,1258))
assert(normalised_X_test.shape == (64, 539))
```

I picked uniform scaling as the the relative distance between pixels (distance in terms of colour values) is preserved so the image will look the same when plotted with the new $[0, 1]$ range. A Disadavantage of using this uniform normalisation is that with such a small grid any low contrasting neighbouring pixels will remain low contrasting finding it just as hard to distinguish between some of the less clear digits like between 8 and 3 for example. Another issue may arise if we are given a new training data set whose pixel values go beyond 16. The normalisation may go above 1 for the largest pixel value. Whereas the sigmoid function, for example, would still always normalise it to $(0, 1)$.

Question 2 b iv)

In [36]:
```
np.random.seed(0)
result = model(normalised_X_train, y_train, normalised_X_test, y_test, num_itera
```

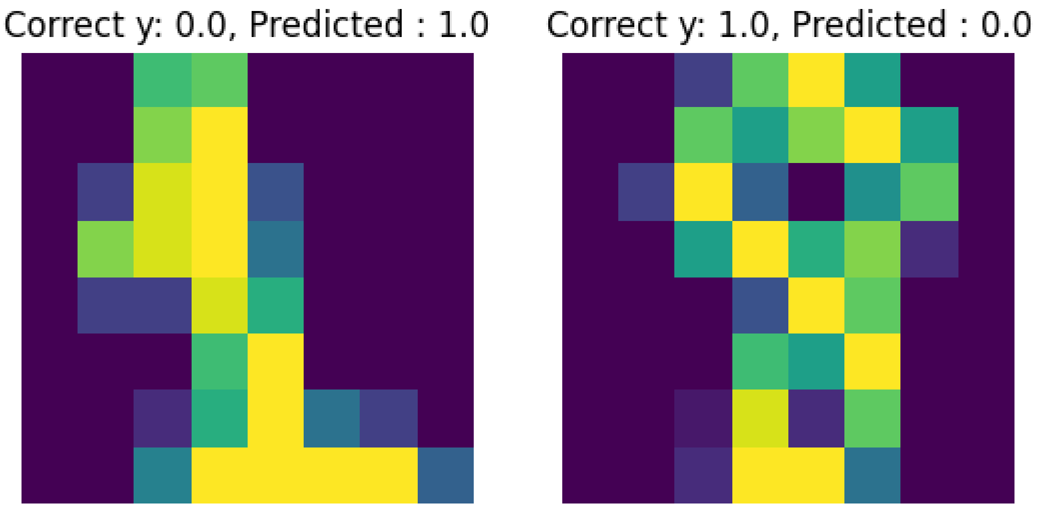
```
train accuracy: 92.60731319554849 %
test accuracy: 92.57884972170686 %
```

In [37]:
```
#Finding indeces of the list where the values were incorrectly labelled
diff = result["Y_prediction_test"] - y_test
diff = diff[0]
indices = []
for i in range(len(diff)):
    if diff[i] != 0.:
        indices.append(i)
print(indices)
```

```
[28, 43, 51, 56, 85, 99, 112, 115, 118, 131, 139, 148, 163, 174, 189, 194, 223, 2
46, 259, 265, 273, 298, 324, 349, 356, 380, 381, 382, 386, 420, 444, 459, 501, 50
3, 512, 517, 528, 529, 531, 536]
```

In [38]:
```
ns = [28, 43]
fig, ax = plt.subplots(1,2)
for i in range(2):
    ax[i].imshow(X_test[:,ns[i]].reshape(8,8))
    ax[i].axis('off')
    #print(np.squeeze(y_test)[ns[i]])
    ax[i].set_title(f"Correct y: {np.squeeze(y_test)[ns[i]]}, Predicted : {resul


print ("For the left image the digit is clearly 1 which is odd but the predicted
```

```
For the left image the digit is clearly 1 which is odd but the predicted value is
1,
and for the right picture the digit is 8 but the model predicted it to be odd
```

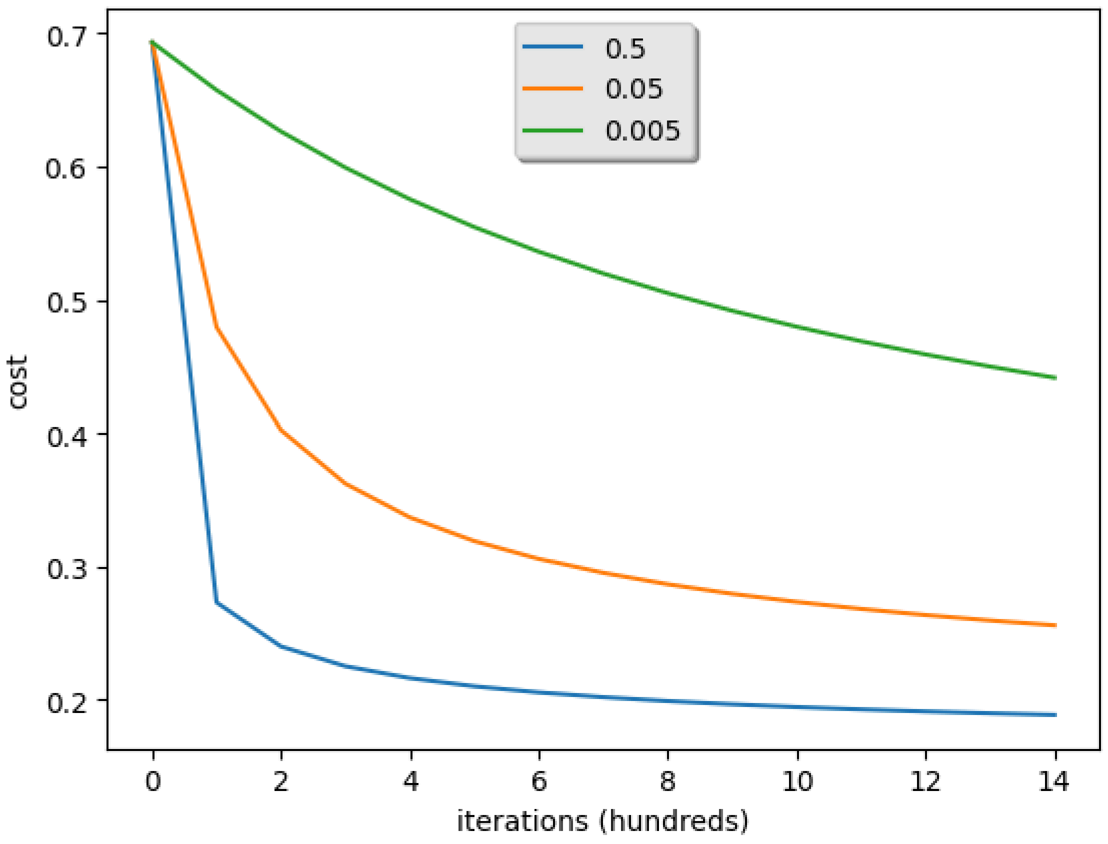


Question 2 v)

The first obvious way the algorithm can be improved is by changing the learning rate and number of iterations.

In [39]:
```
learning_rates = [0.5, 0.05, 0.005]
models = {}
for i in learning_rates:
    print ("learning rate is: " + str(i))
    models[str(i)] = model(normalised_X_train, y_train, normalised_X_test, y_tes
    print ('\n' + "-------------------------------------------------------" + '\

for i in learning_rates:
    plt.plot(np.squeeze(models[str(i)]["costs"]), label= str(models[str(i)]["lea

plt.ylabel('cost')
plt.xlabel('iterations (hundreds)')
legend = plt.legend(loc='upper center', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.90')
plt.show()
```

```
learning rate is: 0.5
train accuracy: 92.60731319554849 %
test accuracy: 92.57884972170686 %

-------------------------------------------------------

learning rate is: 0.05
train accuracy: 90.30206677265501 %
test accuracy: 90.9090909090909 %

-------------------------------------------------------

learning rate is: 0.005
train accuracy: 86.8839427662957 %
test accuracy: 84.9721706864564 %

-------------------------------------------------------
```



In [40]:
```
num_iterations = [1500,5000,10000,100000]
models = {}
for i in num_iterations:
    print ("number of iterations: " + str(i))
    models[str(i)] = model(normalised_X_train, y_train, normalised_X_test, y_tes
    print ('\n' + "-------------------------------------------------------" + '\
number of iterations: 1500
train accuracy: 92.60731319554849 %
test accuracy: 92.57884972170686 %

-------------------------------------------------------

number of iterations: 5000
train accuracy: 92.52782193958664 %
test accuracy: 92.02226345083488 %

-------------------------------------------------------

number of iterations: 10000
train accuracy: 92.84578696343402 %
test accuracy: 91.28014842300557 %

-------------------------------------------------------

number of iterations: 100000
train accuracy: 93.1637519872814 %
test accuracy: 91.4656771799629 %

-------------------------------------------------------
```

As can be, seen decreasing the size of the learning_rate, the convergence becomes a lot slower and the accuracy achieved decreases with the size of the parameter. This is due to number of iterations being too low for the algorithm to converge to a meaningfully small value. In the second piece of code I change the number of iterations to see how the test accuracy will be affected and increasing the iterations doesn't change the accuracies that much. As can be seen the training accuracy increases a little bit while the test accuracy decreases, this is most likely due to overfitting, as the model is learning the training data really well which is less applicable to the test data.

Changes

Above analysis tells us that the complexity of our 1 layer network doesn't allow for all of the features, such as edges and shapes, of the data to be learnt so it limits its own accuracy on the model.

We can increase complexity by adding a hidden layer to create a neural network, and introducing non-linearity so the model can learn all the necessary features.

We will change all of the functions provided so they can be applied to the new model and for that we will need to introduce more functions:

initialise_parameters, sigmoid, relu, relu_backward, sigmoid_backward, linear_forward, linear_activation_forward, model_forward, compute_cost, linear_backward, linear_activation_backward, update_parameters, two_layer_model, predict are all functions taken from Lab Sessions 4 and 5 namely the MA30279_Week4_LS4part1.ipynb and MA30279_Week5_LS4part2.ipynb as well as the dnn_app_utils.py code snippets kindly provided by Lisa Kreusser. Some of the funtions have been modified from the Lab Session ones to fit the model.

We change initialize_with_zeros function to create more initial paramteres and instead of using zeroes everywhere we will initialise w_2 with some small random perturbation to break symmetry between different neurons and allow them to learn different features during training, ensuring the neurons don't learn the same things.

In [41]:
```
def initialise_parameters(n_x, n_h, n_y):

    W1 = np.random.randn(n_h,n_x)*0.01
    b1 = np.zeros((n_h,1))
    W2 = np.random.randn(n_y,n_h)*0.01
    b2 = np.zeros((n_y,1))


    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

Then we need to introduce some more activator functions. We introduce ReLu to use on the first layer to introduce some non-linearity.

We also implement the backword propagations for a single unit of both activation functions which is $dz = dA\sigma'$ to be used later in the code.

We also change the sigmoid function from the previous question so it keeps the value of Z in cache which will be useful in backpropagation.

In [42]:
```
def sigmoid(Z):
    """
    Implements the sigmoid activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of sigmoid(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = 1/(1+np.exp(-Z))
    cache = Z

    return A, cache

def relu(Z):
    """
    Implement the RELU function.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ; stored for computing the backw
    """

    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache


def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, cache):
    """
    Implement the backward propagation for a single SIGMOID unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    assert (dZ.shape == Z.shape)

    return dZ
```

Next 3 functions all correspond to doing the forward propogation:

linear_forward just calculates the linear part of each layer $\boldsymbol{z}^{[l](i)} = \boldsymbol{W}^{[l]}\boldsymbol{a}^{[l-1](i)} + \boldsymbol{b}^{[l]}$

linear_activation_forward applied the activation function to the linear part $\boldsymbol{a}^{[l](i)} = g^{[l]}(\boldsymbol{z}^{[l](i)})$

model_forward applies that to both of the layers

In [43]:
```python
def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A - activations from previous layer (or input data): (size of previous layer
    W - weights matrix: numpy array of shape (size of current layer, size of pre
    b - bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z - the input of the activation function, also called pre-activation paramet
    cache - a python dictionary containing "A", "W" and "b" ; stored for computi
    """

    Z = np.dot(W,A) + b

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

In [44]:
```python
def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev - activations from previous layer (or input data): (size of previous
    W - weights matrix: numpy array of shape (size of current layer, size of pre
    b - bias vector, numpy array of shape (size of the current layer, 1)
    activation - the activation to be used in this layer, stored as a text strin

    Returns:
    A - the output of the activation function, also called the post-activation v
    cache - a python dictionary containing "linear_cache" and "activation_cache"
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)
    return A, cache
```

In [45]:
```
def model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID

    Arguments:
    X - data, numpy array of shape (input size, number of examples)
    parameters - output of initialize_parameters_deep()

    Returns:
    AL - last post-activation value
    caches - list of caches containing:
             every cache of linear_activation_forward() (there are L-1 of them,
    """

    caches = []
    A = X

    A, cache = linear_activation_forward(A, parameters['W1'], parameters['b1'],
    caches.append(cache)

    AL, cache = linear_activation_forward(A, parameters['W2'], parameters['b2'],
    caches.append(cache)

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches
```

Compute_cost allows computes the current cost

$$\mathcal{J} = -\frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{y}^{(i)} \log\left(\boldsymbol{a}^{[L](i)}\right) + (1-\boldsymbol{y}^{(i)}) \log\left(1-\boldsymbol{a}^{[L](i)}\right) \right)$$

Which allows us to draw the graph

In [46]:
```
def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL - probability vector corresponding to your label predictions, shape (1, n
    Y - true "label" vector (for example: containing 0 if non-cat, 1 if cat), sh

    Returns:
    cost - cross-entropy cost
    """
    m = Y.shape[0]


    cost = -1/m * np.sum( np.multiply(np.log(AL),Y) + np.multiply(np.log(1-AL),1


    cost = np.squeeze(cost)
    assert(cost.shape == ())

    return cost
```

Following part does the backward propogation.

Linear_backward calculates the gradients of the linear parts $(dW^{[\ell]}, db^{[\ell]}, dA^{[\ell-1]})$.

The three outputs $(dW^{[\ell]}, db^{[\ell]}, dA^{[\ell-1]})$ are computed using the input $dZ^{[\ell]}$. Here are the formulas you need:

$$dW^{[\ell]} = \frac{\partial \mathcal{L}}{\partial W^{[\ell]}} = \frac{1}{m} dZ^{[\ell]} A^{[l-1]\top}$$

$$db^{[\ell]} = \frac{\partial \mathcal{L}}{\partial b^{[\ell]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[\ell](i)}$$

$$dA^{[\ell-1]} = \frac{\partial \mathcal{L}}{\partial A^{[\ell-1]}} = W^{[\ell]\top} dZ^{[\ell]}$$

linear_activation_backward calculates the gradient of Z namely:

$$dZ^{[\ell]} = dA^{[\ell]} \cdot g'(Z^{[\ell]})$$

In [47]:
```
def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (lay

    Arguments:
    dZ - Gradient of the cost with respect to the linear output (of current laye
    cache - tuple of values (A_prev, W, b) coming from the forward propagation i

    Returns:
    dA_prev - Gradient of the cost with respect to the activation (of the previo
    dW - Gradient of the cost with respect to W (current layer l), same shape as
    db - Gradient of the cost with respect to b (current layer l), same shape as
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1/m * np.dot(dZ,A_prev.T)
    db = 1/m * np.sum(dZ,axis=1,keepdims=True)
    dA_prev = np.dot(W.T,dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

In [48]:
```
def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA - post-activation gradient for current layer l
    cache - tuple of values (linear_cache, activation_cache) we store for comput
    activation - the activation to be used in this layer, stored as a text strin

    Returns:
    dA_prev - Gradient of the cost with respect to the activation (of the previo
    dW - Gradient of the cost with respect to W (current layer l), same shape as
    db - Gradient of the cost with respect to b (current layer l), same shape as
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

Update_parameters just performs gradient decent.

In [49]:
```
def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters - python dictionary containing your parameters
    grads - python dictionary containing your gradients, output of L_model_backw

    Returns:
    parameters - python dictionary containing your updated parameters
                 parameters["W" + str(l)] = ...
                 parameters["b" + str(l)] = ...
    """

    parameters["W1"] = parameters["W1"] - learning_rate * grads["dW1"]
    parameters["b1"] = parameters["b1"] - learning_rate * grads["db1"]
    parameters["W2"] = parameters["W2"] - learning_rate * grads["dW2"]
    parameters["b2"] = parameters["b2"] - learning_rate * grads["db2"]

    return parameters
```

two_layer_model runs the entire model. We pick value of 0.75 for our step as it provides the best results.

predict runs the model on test and train data to give us accuracy on seen and unseen data.

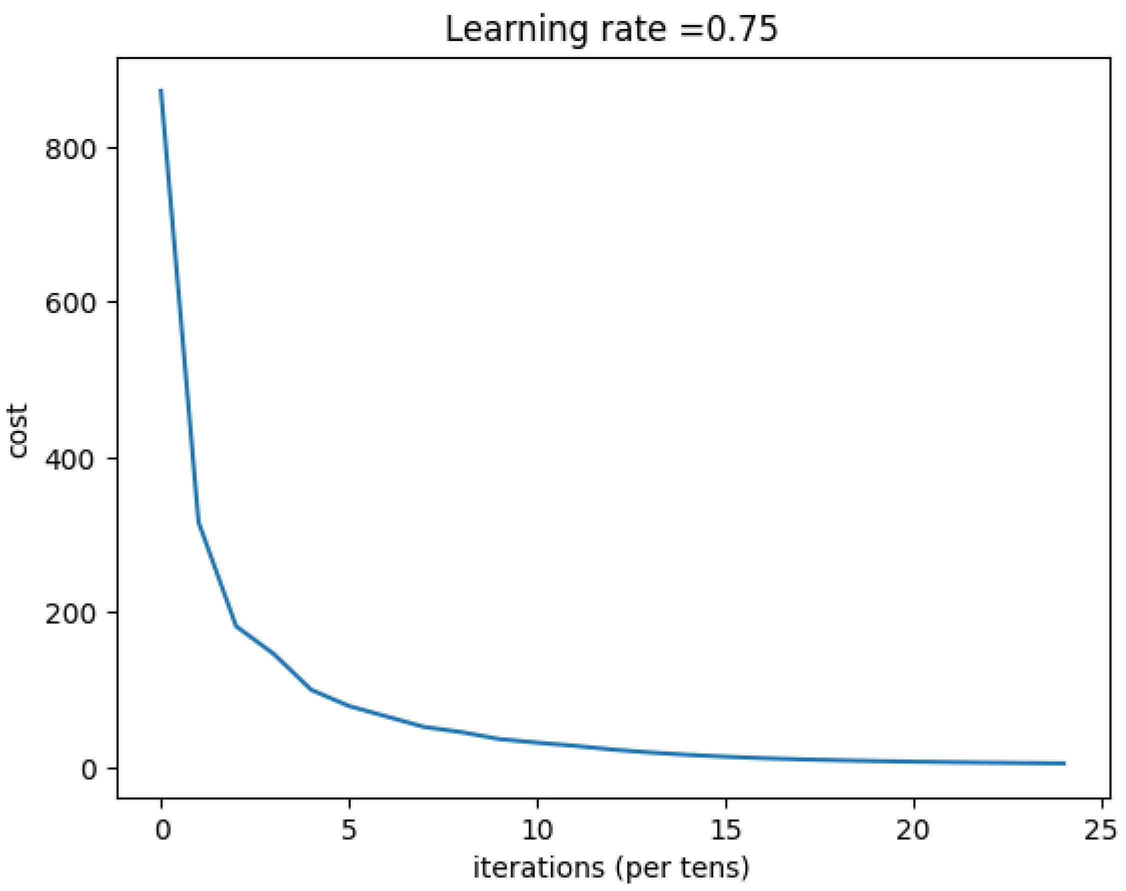
In [50]:
```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.75, num_iterations = 30
    """
    Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:
    X - input data, of shape (n_x, number of examples)
    Y - true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, nu
    layers_dims - dimensions of the layers (n_x, n_h, n_y)
    num_iterations - number of iterations of the optimization loop
    learning_rate - learning rate of the gradient descent update rule
    print_cost - If set to True, this will print the cost every 100 iterations

    Returns:
    parameters - a dictionary containing W1, W2, b1, and b2
    """

    grads = {}
    costs = []
    m = X.shape[1]
    (n_x, n_h, n_y) = layers_dims

    parameters = initialise_parameters(n_x, n_h, n_y)

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    for i in range(0, num_iterations):

        A1, cache1 = linear_activation_forward(X, W1, b1, activation='relu')
        A2, cache2 = linear_activation_forward(A1, W2, b2, activation='sigmoid')

        cost = compute_cost(A2, Y)

        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, activation='sigm
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, activation='relu

        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

        parameters = update_parameters(parameters, grads, learning_rate)

        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]

        if print_cost and i % 100 == 0:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

In [51]:
```
def predict(X, Y, parameters):
    """
    This function is used to predict the results of a  L-layer neural network.

    Arguments:
    X - input data, of shape (n_x, number of examples)
    Y - true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, nu
    parameters -- parameters of the trained model
    Returns:
    p -- predictions for the given dataset X
    """

    m = X.shape[1]
    n = len(parameters) // 2
    p = np.zeros((1,m))

    yhat, caches = model_forward(X, parameters)

    for i in range(0, yhat.shape[1]):
        if yhat[0,i] > 0.5:
            p[0,i] = 1
        else:
            p[0,i] = 0

    print("Accuracy: "  + str(np.sum((p == Y)/m)))

    return p
```

Now we run the model we pick the hidden layer to have 7 neurons.

In [52]:
```
np.random.seed(0)
parameters = two_layer_model(normalised_X_train, y_train, layers_dims = (64, 7,
```

```
Cost after iteration 0: 871.9376210061605
Cost after iteration 100: 316.13551325564544
Cost after iteration 200: 181.79485356964437
Cost after iteration 300: 146.15434741244206
Cost after iteration 400: 99.75581925483183
Cost after iteration 500: 78.95218137219261
Cost after iteration 600: 65.45100665826716
Cost after iteration 700: 51.945146153033605
Cost after iteration 800: 45.16422246790802
Cost after iteration 900: 36.29750762669099
Cost after iteration 1000: 31.682464450877028
Cost after iteration 1100: 27.552413924150365
Cost after iteration 1200: 22.714383908686813
Cost after iteration 1300: 18.9566663341715
Cost after iteration 1400: 15.953759126817143
Cost after iteration 1500: 13.558047059804498
Cost after iteration 1600: 11.642636609949074
Cost after iteration 1700: 10.120410316220317
Cost after iteration 1800: 8.919377999037678
Cost after iteration 1900: 7.935137515617434
Cost after iteration 2000: 7.114520335546354
Cost after iteration 2100: 6.439620573108407
Cost after iteration 2200: 5.856678377860987
Cost after iteration 2300: 5.362169545833761
Cost after iteration 2400: 4.9398095762212115
```



```
In [53]: print("Training accuracy of the model:")
         predictions_train = predict(normalised_X_train, y_train, parameters)
         print("Testing accuracy of the model:")
         predictions_train = predict(normalised_X_test, y_test, parameters)
```

```
Training accuracy of the model:
Accuracy: 1.0
Testing accuracy of the model:
Accuracy: 0.9795918367346939
```

Our training accuracy of the new model is 100% which is 7.4% larger than the previous model of 92.6%. And the testing accuracy also incread by 5.4% from 92.6% to 98%

This tells us that the training error is 0 and the generalisation error is very low for the new model which is performing really well over the previous single layer neural network.

This could be potentially improved by finding more optimal hyperparameters such as our learning_step and the number of neurons in the hidden layer, however there is more hyperparameter to be tuned in the new model compared to the previous one which makes it more difficult to find the right ones. Apart from longer training times and increase in computational cost, another issue that may arise from the new model is the larger risk of overfitting, in this case there might be a slight case of overfitting as there is a 2% difference in the accuracy between the training and test data.

On the other hand the complexity of the new model allows it to capture, and extract more non-linear patterns of the numbers such as edges and strokes as well as other features leading to a much better generalisation on unseen data and hence improved accuracy.