

NUERAL NET

HW2

Oct 24, 2022



S.M. Erfan Moosavi M.

TABLE OF CONTENTS

- Describe the concept of overfitting in neural networks and when do we call a neural network overfitted?..... 1
- What methods do you purpose to prevent overfitting? 3
- Describe the concept of underfitting in neural network? 4
- For a two class, classification task, what activation function and loss function do you purpose? 4
- Consider the following patterns, using only MLP Network, separate the two patterns above from two below..... 6
- Describe the structure and parameters of the network:..... 6
- What do you think the response of the model is for the following pattern? 8
- Consider a network with only one hidden layer which accepts (x,y) coordinates, Does this network classify the coordinates inside the graph as one and others as zero?..... 9
- Why activation functions should be non-linear? Is it okay to use any non-linear function as activation function?10
- What will happen if we initialize our network weights with zero?
11

- In the steps of gradient descent, does the gradient always descent?12
- Based on the loss and accuracy plot, describe whether overfit or underfit is happened?13
- Code Description:14
- *Sources*20

- **Describe the concept of overfitting in neural networks and when do we call a neural network overfitted?**

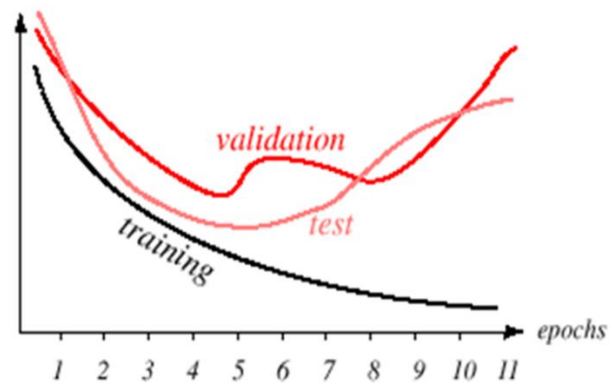
Overfit is a learning state in which the model learned the training data very well but perform poorly on other unseen data such as test or validation datasets.



The above image seems to be a good visual explanation for overfitting.

(Source: [How does overfitting happen in a neural network? - Quora](#))

To put it in more technical terms, when you train your network up to a point that training loss is getting smaller and smaller but the performance on unseen data does not improve, we can say that the model is overfitted over training data or in other words, the model did not generalize well, which means that if unseen data deviate a little bit from what model seen during training phase, it can not perform well on it, although it performs super good on training examples.



For example, in this image from course's slides, we can see that by training the model more and more, we get better result on training data, which is expected, but we not only perform poorly on test and validation, but by increasing the model complexity (more epochs) we are making them worse, which is due to losing generalization ability by model.

- **What methods do you purpose to prevent overfitting?**

First you can diversify and enlarge your dataset if it's possible. With more data, it is less likely to end up with an overfitted model. But the downside of this method is in most cases, more data is not available.

In such cases, you can augment your data or add some noise to it.

Data augmentation is usually done for images, which includes operations like filliping, reversing, rotating and ...

In case of noise addition, what you are doing is basically destroying information which make it harder for the model to fit on data, thus make it harder to overfit.

A more common method to prevent overfitting is regularization which is introducing a penalty term to cost function that going to prevent model from getting more complex by applying penalty to big coefficients and weights.

A different approach to prevent overfitting is to simplify the input data to the model (instead of simplifying model). You can use feature selection techniques to only select important features.

Last but not least, early stopping, which means you constantly monitor the training loss and validation loss, and if through training, validation loss started to go higher, you stop the training and report back the best model (you need to keep a history of models before each epoch).

- **Describe the concept of underfitting in neural network?**

Underfitting is a stage in model training where the model did not capture the relation between its training dataset's inputs and outputs well enough. In other words, the training loss is high and since the model did not train well, it has high loss on unseen data too.

For overfitting, we said model is too complex, here for underfitting our model is too simple. In case of under fitting, not only the model did not generalize on unseen data, it didn't generalize well on training data too.

- **For a two class, classification task, what activation function and loss function do you purpose?**

Choosing right activation function and loss function highly depends on the task to be done. For example, we can consider the two-class problem as binary problem and define a binary classifier, in binary problem the output layer activation function is sigmoid, because sigmoid generates probabilistic estimations. And then we can use a binary cross entropy loss function. In binary cross entropy, we want to make two distributions similar to each other. One distro is the actual distro of the data and the other is the one we are approximating, since in most cases we can not approximate the exact distro, we end up with an approximation with higher entropy, this approximation has a difference with the real one, we want to minimize this difference which is cross entropy loss function value.

On the other hand, one may say that we need to consider this problem as a multi class classification. Therefore, we want to have 2 outputs neurons, each of which corresponds to a specific outcome (class). In this case the usage of sigmoid in output layer is prevented. In such cases we use softmax function as our output's activation function.

What softmax do is basically this: it converts a vector of k values into a probability distribution. In case of softmax function at output layer,

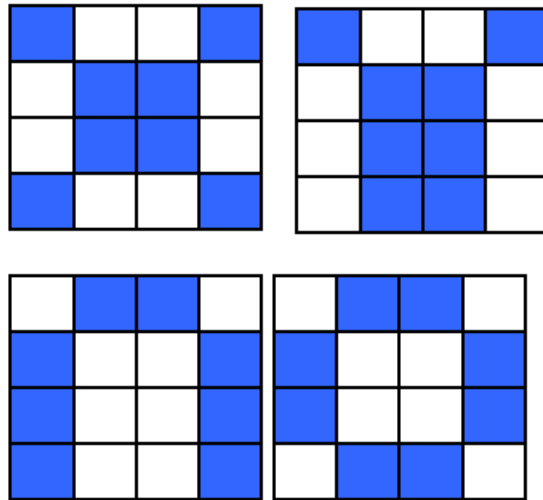
we are going to use categorical cross entropy. Although, I don't think that defining our problem as multi class classification going to help us in case of only two class. So, I still suggest using binary classification settings.

But there are some cases where the samples of two class problems are not exclusively in one of the classes and they can belong to both class (or maybe neither of classes). In these cases, we need to define the problem as multi label two class classifications. It means that we do not use conventional one-hot encoding that we used in multiclass classification. Encoding can be selected from these options:

$[0, 0]$ – $[0, 1]$ – $[1, 0]$ – $[1, 1]$

Now in this case, we need to use sparse categorical cross entropy.

- Consider the following patterns, using only MLP Network, separate the two patterns above from two below.



- Describe the structure and parameters of the network:

Let's call those two above, group A and those two below, group B.

In group A, the first row is similar and, in both cases, follows this pattern: on, off, off, on

But in group B, we have the exact reverse pattern for row one:

Off, on, on, off

In case of row two and three, in group A the pattern is:

Off, on, on, off

Off, on, on, off

But in group B the pattern is reversed again:

On, off, off, on

On, off, off, on

In case of last row, we can see there is no similarity in groups and no dissimilarity between groups that comes handy for us to classify them.

If we didn't want to use MLP, we could use only one perceptron to learn the upper left element of the patterns, because with only that element we can discriminate between these patterns' groups.

But since we want to use MLP, and as discussed in class, we have to provide the model with all the input we have, we can not use those hard programming and hard modeling techniques to create the classifier.

So, in case of MLP: since we have a 4×4 binary pattern, we need 16 neurons in input layer and one neuron in output (binary classification).

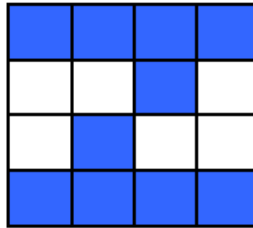
The activation function for last layer can be either sigmoid or tanh. I used tanh, since I labeled them with positive one and negative one, and then applied a sign function on output to get the results.

We don't need hidden layer because the patterns are not difficult. The model just needs to learn that groups have the same three rows and the last row does not matter.

To test this idea, I implemented the network with Keras, you can find the result at the end of the HW2 ipynb file.

Since the dataset is too small, you have to run the model multiple times to finally end up with a good enough initial weights and then train your model to get desired outputs.

- What do you think the response of the model is for the following pattern?



Omitting the last row, we can count the number of similar pixels of this pattern with patterns in groups:

Similarity with A: 4 px

Similarity with B: 2 px

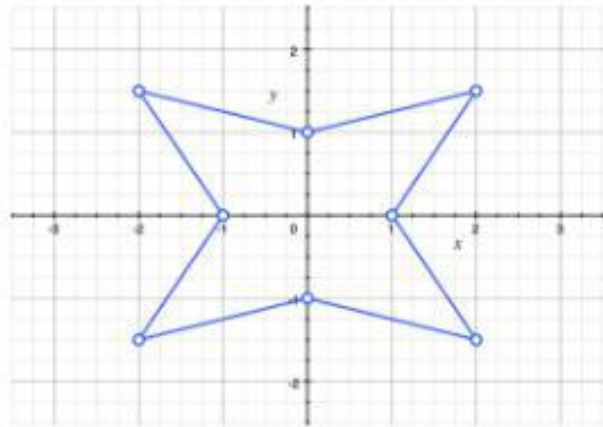
So, I expect the neural network classify this as group A.

I ran some experiment, and in those successful ones, this was the case. (This pattern classified as group A)

Note:

In empirical tests, there are multiple cases with different results that can happen, maybe this is due to some regularization parameter that is present in Keras, which I'm not aware of, that affect the overfitting of the network. But I think if we'll be able to properly overfit the network on patterns in groups A and B, the expected result that I mentioned above should happen.

- Consider a network with only one hidden layer which accepts (x,y) coordinates, Does this network classify the coordinates inside the graph as one and others as zero?



In the course slides, It stated that a single layer neural net can realize convex regions, and two hidden layer network is able to realize multiple convex regions. So base on these statements, since the shape in the above diagram is not convex, it is not possible to learn it with a single layer perceptron, and you need at least two layers. And the two layer perceptron learns it in this way, it is going to learn small convex portions of the shape in a way that the intersect of them create the whole shape.

- **Why activation functions should be non-linear? Is it okay to use any non-linear function as activation function?**

We use non-linear activation functions to be able to exploit multiple layers in our network, because if we do not use non-linear activation functions, no matter how many layers we add, we end up with a single layer perceptron. Consider the following two-layer perceptron as an example:

$$y = \phi(W_2\phi(W_1X))$$

For simplicity we combined the bias in weights and added a new neuron in input which is always 1.

Now if I remove the non-linear function ϕ , from the equation:

$$\begin{aligned}y &= W_2W_1X \\ W &= W_2W_1 \rightarrow y = WX\end{aligned}$$

You can see that all the weights can combine together to form a single weight matrix. Which means you reduced your 2-layers network to a single layer network.

So, because of this, we use non-linearity, also the power of neural networks is to classify non-linear problems. So having the non-linearity is crucial.

Now to answer the second question:

There are two main things we need to consider when we want to select our activation function. First thing the optimization method for network weights and the second thing is the output range of values.

So for the first thing, if we want to use backpropagation, we have to choose a non linear function which has derivative, so in this sense, we are not allowed to select any arbitrary non linear function if we want to use back propagation.

The second part, which is the output range, means that if our output consists of for example negative values, we can not use function with only positive results as last layer activation function, like relu or simple sigmoid.

- **What will happen if we initialize our network weights with zero?**

There are two scenarios based on the activation function that used.

First ones are those activations that produce zero at zero:

$$\phi(0) = 0, \text{ like } \text{relu}, \text{ tanh}$$

Second ones are those activation functions which produce non zero at zero:

$$\phi(0) \neq 0, \text{ like } \text{sigmoid}$$

Now in the feedforward process, the first group will result zero in every layer than input, because a linear composition of input with zero vector is always zero. In case of second group, the outputs of each layer are the same as the output of that activation function at zero.

For example, for sigmoid, it would be 0.5

Now in backpropagation step, let's consider the following scenario where L denotes the last layer:

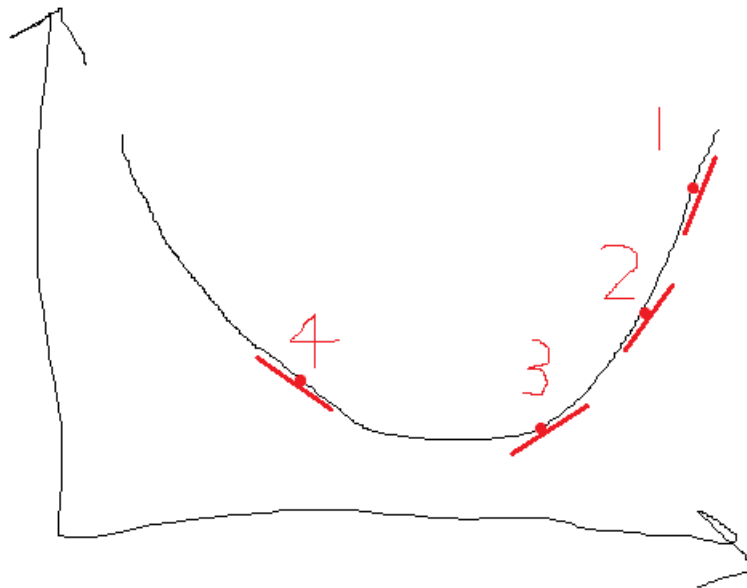
$$dw^l = \frac{\partial Loss}{\partial w^l} * \frac{\partial a^l}{\partial (w^l a^{l-1})} * \frac{\partial (w^l a^{l-1})}{\partial w^l}$$

The last term is equals to the output of the previous layer. So it means that if the output of previous layer is zero, the update would be zero and the weight don't change, no learning would happen.

If the output of the previous layer is not zero, you get the dw as vector with all elements equal to each other, which means the update will always go to the same direction.

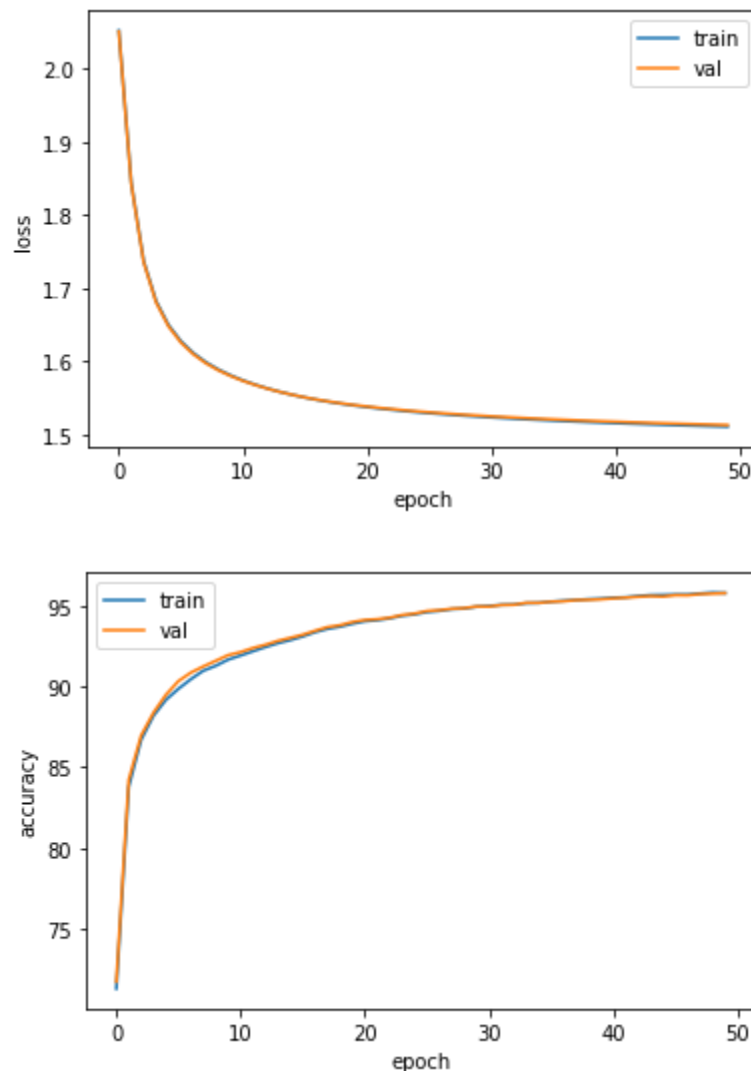
- **In the steps of gradient descent, does the gradient always descent?**

The gradient always points in the direction of steepest increase in the loss function. But the weight update can cause the algorithm to go to the other side and in this case the gradient might not be smaller than previous one. It depends on the learning rate, if for example we randomly generate a learning rate in each step, it may take our point to the other side of the hill and then we need to come back, in this case, it might happen a scenario in which the gradient of the come back be bigger than previous one.



Here you can see that in point 4, the gradient is bigger than point 3.

- **Based on the loss and accuracy plot, describe whether overfit or underfit is happened?**



Since both train and evaluation moved together, we can say that with 50 epochs, no overfit happened, although with more epoch it may end up in an overfitted stage.

We can see that after like 40 epochs, the model accuracy didn't change much, so it seems that 40 epochs are enough to prevent this model from underfitting.

- **Code Description:**










```
• ## We've implemented the Linear activation function for you
• def linear(x, deriv=False):
•
•     return x if not deriv else np.ones_like(x)
•
• def relu(x, deriv=False):
•     """
•     Args:
•     x: A numpy array of any shape
•     deriv: True or False. determines if we want the derivative
•     of the function or not.
•
•     Returns:
•     relu_out: A numpy array of the same shape as x.
•     Basically relu function or its derivative applied to
•     every element of x
•
•     """
•
• #####
• # Put your implementation here #
• #####
• # [Erfan]: numpy's maximum function return the element wise
• maximum of two array (do broadcast if needed)
• #
• https://numpy.org/doc/stable/reference/generated/numpy.maximum
• .html
• #
• https://numpy.org/doc/stable/reference/generated/numpy.heaviside
• .html
•
• relu_out = np.maximum(0, x) if not deriv else
• np.heaviside(x, 1)
• return relu_out
•
• def sigmoid(x, deriv=False):
•     """
•     Args:
•     x: A numpy array of any shape
•     deriv: True or False. determines if we want the derivative
•     of the function or not.
•
•     Returns:
```

```

•     sig_out: A numpy array of the same shape as x.
•     Basically sigmoid function or its derivative applied to
every element of x
•
•     """
•
•     #####
•     #     Put your implementation here     #
•     #####
•     sig_out = 1 / (1 + np.exp(-x)) if not deriv else sigmoid(x)
•     * (1 - sigmoid(x))
•     return sig_out

```

Activation Functions are implemented based on the following image:

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Feed Forward:

```

(1) for i, wb in enumerate(self.parameters):
(2)     w, b = wb['w'], wb['b']
(3)     afunc = self.act_funcs[i]

```

```

(4)         wi = mlp_out @ w + b
(5)         self.weighted_ins.append(wi)
(6)         awi = afunc(wi)
(7)         self.activations.append(awi)
(8)         mlp_out = awi

```

Iterate over each layer and then calculate the following values:

$$\phi(WX + B), \phi \text{ is the activation function}$$

```

(9)  def softmax(y_hat):
(10)     """
(11)         Apply softmax to the inputs
(12)     Args:
(13)         y_hat: A numpy array of shape (b, out_dim) where b
                   is the batch size and out_dim is the output dimension of
                   the network(number of classes)
(14)
(15)     Returns:
(16)         soft_out: A numpy array of shape (b, out_dim)
(17)
(18)     """
(19)
(20)     #####
(21)     #         Put your implementation here         #
(22)     #####
(23)     ey = np.exp(y_hat)
(24)     sey = np.sum(ey, axis=1)
(25)     soft_out = ey / sey.reshape((-1, 1))
(26)
(27)     return soft_out

```

Implemented the softmax function based on the formula provided.

First, I calculated the exponential of y hat, then I summed over rows, at the end I divided the values by summation to calculate the probability distribution.

```

(28)  ls = np.log(y_soft)
(29)  yls = y * ls
(30)  loss = -np.mean(np.sum(yls, axis=1))

```

Categorical cross entropy is implemented with the same formula provided in ipynb file.

```
(31)     afunc = mlp.act_funcs[i]
(32)     wi = mlp.weighted_ins[i]
(33)     act = activations[i].T
(34)     w = mlp.parameters[i]['w'].T
(35)
(36)     g *= afunc(wi, True)
(37)
(38)     db = np.sum(g, axis=0)
(39)     dw = act.dot(g)
(40)
(41)     g = g.dot(w)
(42)
(43)     grad = {
(44)         'w':dw,
(45)         'b':db
(46)     }
(47)     gradients.insert(0, grad)
(48)
```

This is the iterative process of backpropagation, honestly, I'm not sure what Should I explain, this is the python code version of the formula you provided. 😊

```
(49)     # https://machinelearningmastery.com/gradient-
        descent-with-momentum-from-scratch/
(50)
(51)     self.velocity = [{'w':0, 'b':0} for i in
        range(len(parameters))]
(52)     Updated_parameters = [{'w':0, 'b':0} for i in
        range(len(parameters))]
(53)
(54)     for i in range(len(parameters)):
(55)         wb = parameters[i]
(56)         dwb = grads[i]
(57)         vel = self.velocity[i]
(58)
(59)         change_gdw = self.lr * dwb['w']
(60)         change_gdb = self.lr * dwb['b']
```

```

(61)         change_mw = self.momentum * vel['w']
(62)         change_mb = self.momentum * vel['b']
(63)
(64)         change_w = change_gdw + change_mw
(65)         change_b = change_gdb + change_mb
(66)
(67)         self.velocity[i]['w'] = change_w
(68)         self.velocity[i]['b'] = change_b
(69)
(70)         Updated_parameters[i]['w'] = wb['w'] - change_w
(71)         Updated_parameters[i]['b'] = wb['b'] - change_b

```

Ok here I think I should explain, I implemented the `sgd+momentum` which was provided in the above link. In momentum, we keep a history variable (velocity) which contains the previous changes that we applied to the weights, and in each step, we also add some proportion of that change to new changes. Which is controlled by momentum factor.

```

(72)
(73) mlp.add_layer(256, relu) # hidden 1
(74) mlp.add_layer(128, relu) # hidden 2
(75) # if you set out put activation as linear, the model
      conveges with less than 10 epochs
(76) mlp.add_layer(10, sigmoid) # output

```

Yeah, here I tested some different structures to end up with this set of parameters, that worked well.

Note:

You can train the model with less epoch if you use linear activation function for all the layers or at least last layer.

```

(77) # https://keras.io/guides/sequential_model/
(78) # https://www.kdnuggets.com/2018/06/basic-keras-neural-
      network-sequential-model.html
(79)
(80) model = Sequential(
(81)     [
(82)         Input(784),

```

```

(83)         Dense(256, activation='relu'),
(84)         Dropout(0.5),
(85)         Dense(128, activation="relu"),
(86)         Dropout(0.5),
(87)         Dense(10, activation='softmax'),
(88)     ]
(89) )
(90)
(91) model.compile(optimizer='adam',
(92)               loss='categorical_crossentropy',
(93)               metrics=['accuracy'])

```

Using keras API, I built a sequential model, then compiled it to use adam optimizer and having categorical cross entropy.

```

(94) model.fit(x_train, y_train,
(95)           batch_size=1024,
(96)           epochs=10,
(97)           verbose=1,
(98)           validation_data=(x_val, y_val))

```

This is where I called the fit function, to start the training process.

```

(99) score = model.evaluate(x_val, y_val, verbose=0)
(100) print('Test loss:', score[0])
(101) print('Test accuracy:', score[1])

```

In this section I printed the scores of the model.

- **Sources :**

Course Slides

Andrew NG Machine Learning & Deep Learning Course

Prior knowledge from Computational Intelligence Course

[Overfitting in a Neural Network explained - deeplizard](#)

[How does overfitting happen in a neural network? - Quora](#)

[Guide to Prevent Overfitting in Neural Networks - Analytics Vidhya](#)

[Overfitting vs Underfitting in Neural Network and Comparison of Error rate with Complexity Graph | by Khush Patel | Towards Data Science](#)

[What is Underfitting? | IBM](#)

[How to Select Loss Function? How to Select Activation Function? | Towards Data Science](#)

[Understanding binary cross-entropy / log loss: a visual explanation | by Daniel Godoy | Towards Data Science](#)

[Visual Information Theory -- colah's blog](#)

[Difference between Multi-Class and Multi-Label Classification \(analyticsvidhya.com\)](#)

[classification - What is the difference between Multiclass and Multilabel Problem - Cross Validated \(stackexchange.com\)](#)

[machine learning - Cross Entropy vs. Sparse Cross Entropy: When to use one over the other - Cross Validated \(stackexchange.com\)](#)

[Cross-entropy for classification. Binary, multi-class and multi-label... | by Vlastimil Martinek | Towards Data Science](#)

[python - What loss function for multi-class, multi-label classification tasks in neural networks? - Cross Validated \(stackexchange.com\)](#)