Iran university of science and technology

October 29, 2024

ANN ASSIGNMENT 2

Presented To
Mehdi Feghhi

Presented By
Hessam Koucheki
403 7239 87

Table of Contents

Problem 1

1. *What is Overfitting in Neural Networks?*

Explain the concept of overfitting in neural networks, including how it impacts the model's ability to generalize to new data.

Overfitting, in simple terms, means that the model learns the data very well in a way that it only can predict the data that it has already seen; and cannot predict (or classify, etc.) any new data that were not in the training dataset. In fact, the model learns very complex patterns on the training dataset that commonly may not occur on the unseen data. Overfitting makes problem when model sees new data that are usually different from the previous data that the model is trained on (loses the ability to work well on unseen data is called losing the ability to generalize.) The model, should be able to predict new data that are different from the training set, which is called generalizing. This significantly reduces the accuracy of the model and we can do perform some techniques to avoid it, such as applying dropout. (*AI:* When model has huge number of parameters relative to the amount of training data, it learns complex patterns as well as noises on the dataset, leading to losing the ability to generalize.)

2. *Identifying Overfitting in Neural Networks*

When can we say that a neural network is overfitting? Discuss this in the context of the different networks you have studied and provide a general analysis.

When have three parts that we use when training the model: Train set, test set, and validation set. Overfitting can be recognized when the model is perfectly performing on the training set, meaning that it has very high accuracy on this section, but meanwhile, it has poor performance on unseen data. For instance, if the model that we have made to recognize handwritten digits on the previous assignment is giving us the accuracy of +95% on the training set while giving a ~50% accuracy on unseen data, it means that the model is overfitted.

3. *Methods to Prevent Overfitting*

What are some techniques used to prevent overfitting in neural networks? Offer detailed explanations of each method.

There are several ways that we can apply to avoid overfitting.

Dropout

As we previously discussed, one of the approaches that can take into account to avoid overfitting is using dropout. As it was mentioned, large network compared to the amount of data can learn complex patterns and noises; reducing the size of the network by simply turning off some neurons can lower the capacity of the model to learn too much.

Reducing Size of the Model

Instead of using dropout, we can reduce the size of the model at the first place. A lower number of neurons, and consequently weights, will prevent model to learn complex patterns.

Early Stopping

When the model has not learned that much, it has the ability to generalize. In other words, it has not learned the complex patterns of the data which result in overfitting. By constantly monitoring the accuracy on validation set, we can observe the model's performance and when it starts to decrease on validation set, we can stop the process of training, avoiding the model to keep learning complicated patterns.

Data Augmentation[*]

We can also create some new data that have slightly different attributes from the original data (Increasing the size and diversity of the training data) to help the model learn more generally. This will help it to perform better generalization later. But we have to be careful about not making any odd data which are fundamentally different from the original data. When working with images, for instance, we can achieve this by rotation, flipping, or scaling the images. Here, we have to take care that in some images, doing these techniques will result in wrong data. As an example, rotating the letter 'b' will make a 'q' which is a totally different letter; and flipping the letter 'w' will make a 'm'.

Regularization[*]

Applying techniques for regularization will add some penalty for larger weights during the training. This discourages the model to learn complex patterns. These techniques add constraints and information to the model, preventing the model to learn too much. "*The goal of regularization is to improve the model's generalization ability.*"

[*] This one(s) are granted with the aid of AI.

4. What is Underfitting in Neural Networks?

Explain the concept of underfitting and how it occurs in neural networks.

Underfitting is at the opposite side of the overfitting. It means that the model poorly learned and trained as it cannot even perform on the "seen" data. In this case, the model has pretty low accuracy on both train and test datasets. Sometimes it can be said that the model is generalization too much. When underfitting occurs, the model has not learned patterns that much to be able to predict on seen and unseen data, leading to very poor performance. This can happen when the model is too simple in a way that it is unable to find any pattern on the data and use them later to accurately predict unseen data. Most of the times, the reason of the underfitting is that we did not train the model enough, and by simply training the model more we can address this problem. If we continue the learning and it does not help, it is probably because the model has low capacity to learn, i.e. it has few neurons or layers compared to the size (and diversity) of the data to learn useful and important features on that data. In addition to that, choosing a large learning rate can cause problems.

5. Methods to Address Underfitting

What are the suggested approaches to handle underfitting in neural networks? Explain these methods in detail.

There are some techniques that commonly used to avoid underfitting.

Increasing Size of the Model

When a model has a larger size, meaning that it has higher number of neurons or/and layers, it has more ability and capacity to capture features of the data. If it becomes to excessive, it will result in overfitting which we have already discussed; and when It becomes too low, it will result in underfitting, which means that the model has not the capacity to learn patterns of the data.

Continuing the Learning

As we said before, underfitting means that the model has not learned enough; it can be because we have not let the training go enough to learn the data. Thus, if we allow the model to learn longer on more epochs. A good practice to see whether we are training the model enough or not, is to observe the trend of the loss. If it is not stabilized at the latent epochs, it probably means that it has space for more training. But if we are seeing that the loss and accuracy is not changing that much at the ending epochs, it deduces that the problem is not here and we have to look for other ways to fix this.

Adjusting Learnig Rate

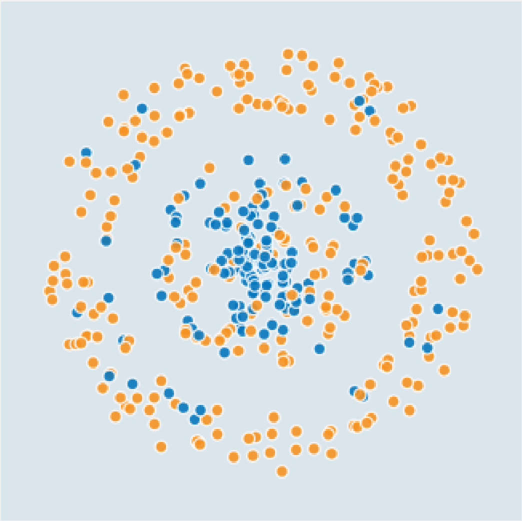
If we choose a large learning rate and see severe fluctuation on the accuracy and the loss of the model, it can be from learning rate, which prevents the model to learn. By reducing the learning rate, we can avoid this problem. On the other hand, if we had chosen a small learning rate, we don't let the model to go for optimal point of learning and update the weight. Altough keep training the model can fix this problem, but increasing the learning rate is a better option.

Data Cleaning

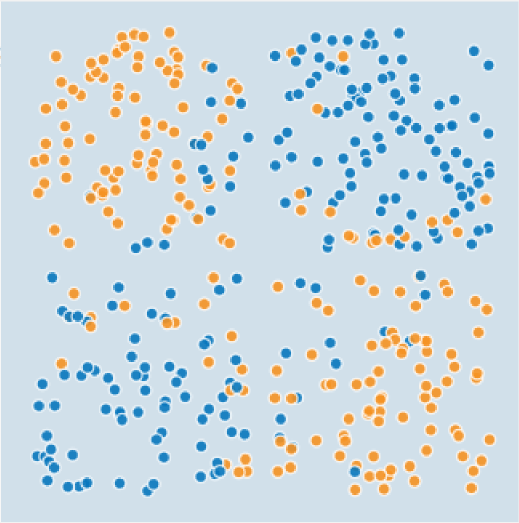
Data cleaning or data engineering can be beneficial to get rid of un-useful features that are preventing the model to focus on more important data. In some other cases, some features are not a good representer of the data, and by eliminating them we can increase the capacity of the model to learn better. In addition to that, some features have a low relation to the data, and act as random numbers and attributes; by eliminating those kinds of data, we free up space for model to put more attention and focus on the most impactful attributes at the dataset.

Problem 2

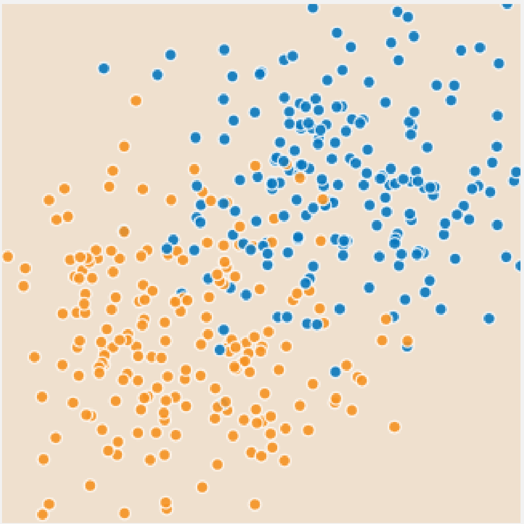
As requested, I set the noise to maximum amount of it, which is 50 to challenge the classifier. The website has four datasets:



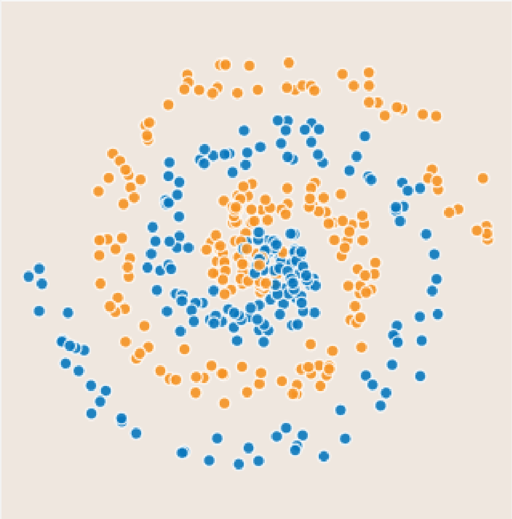
Circle



Exclusive OR

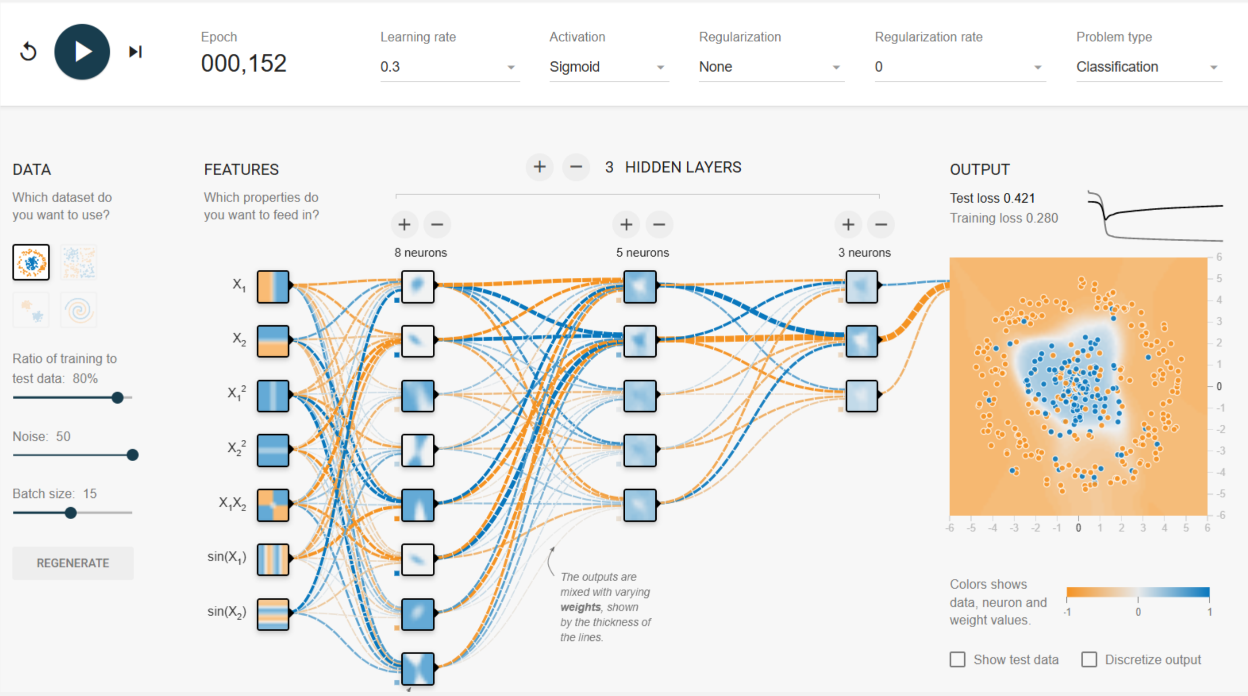


Gaussian

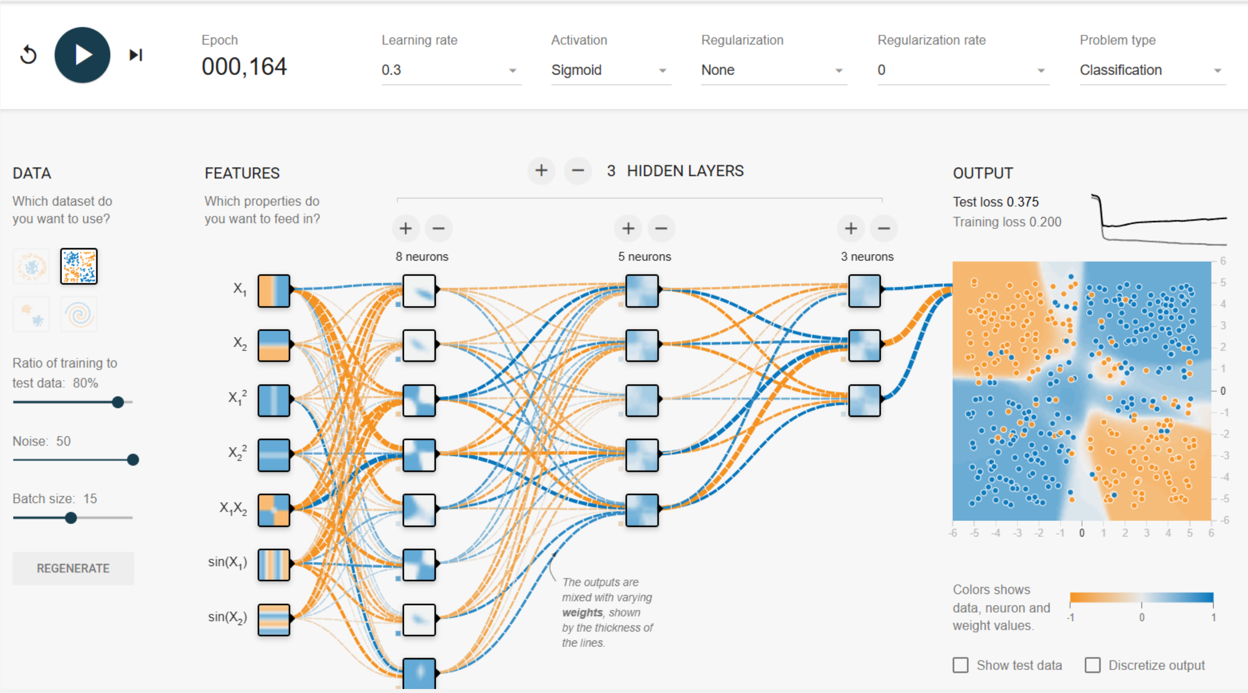


Spiral

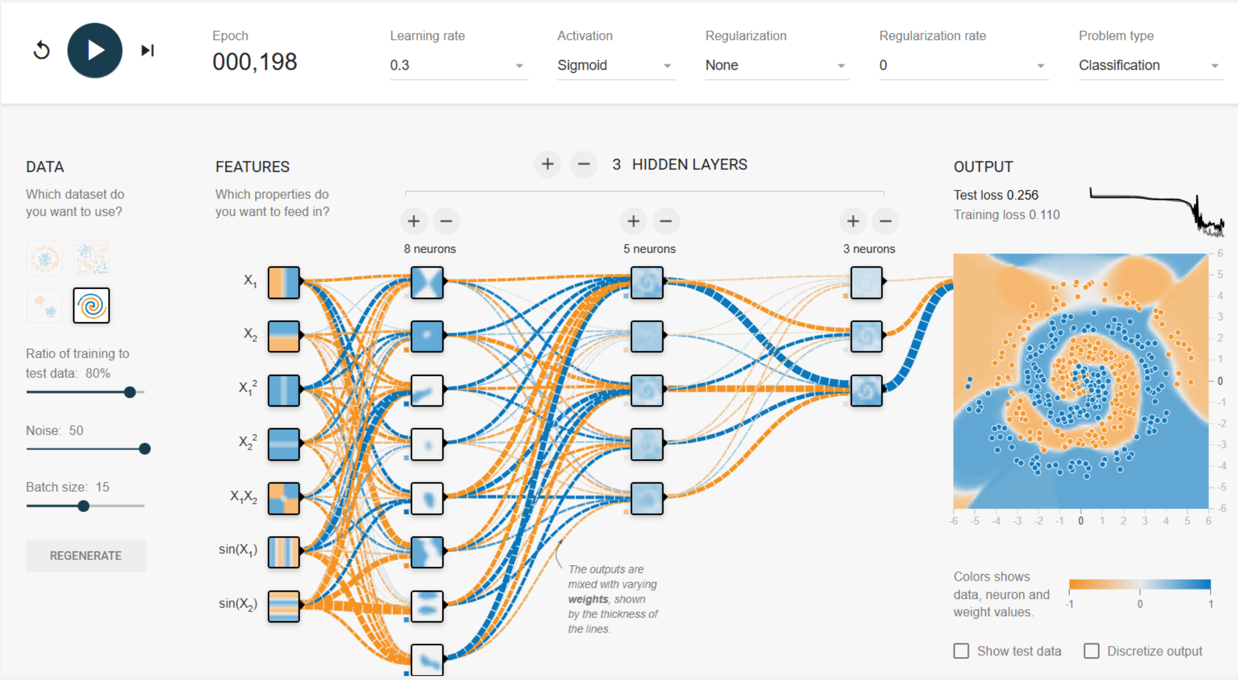
Now, let's play with the parameters.



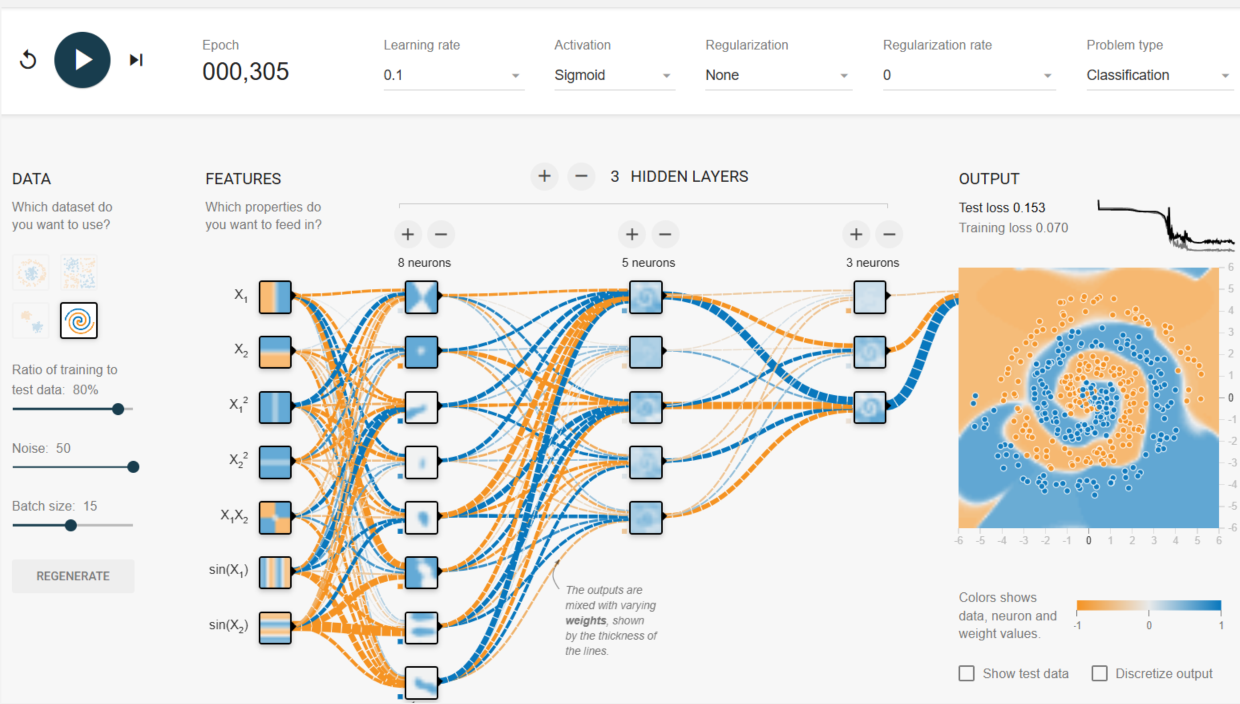
Working with these setting, gives us a relatively good training loss, while the test loss is not that good. Despite of the noises of the data, the model is too complex and is biased to some noises as we can see at the top right and top left of the blue area. Those noises attracted the model to be learnen on and capture them, making a bad accuracy on unseen data.



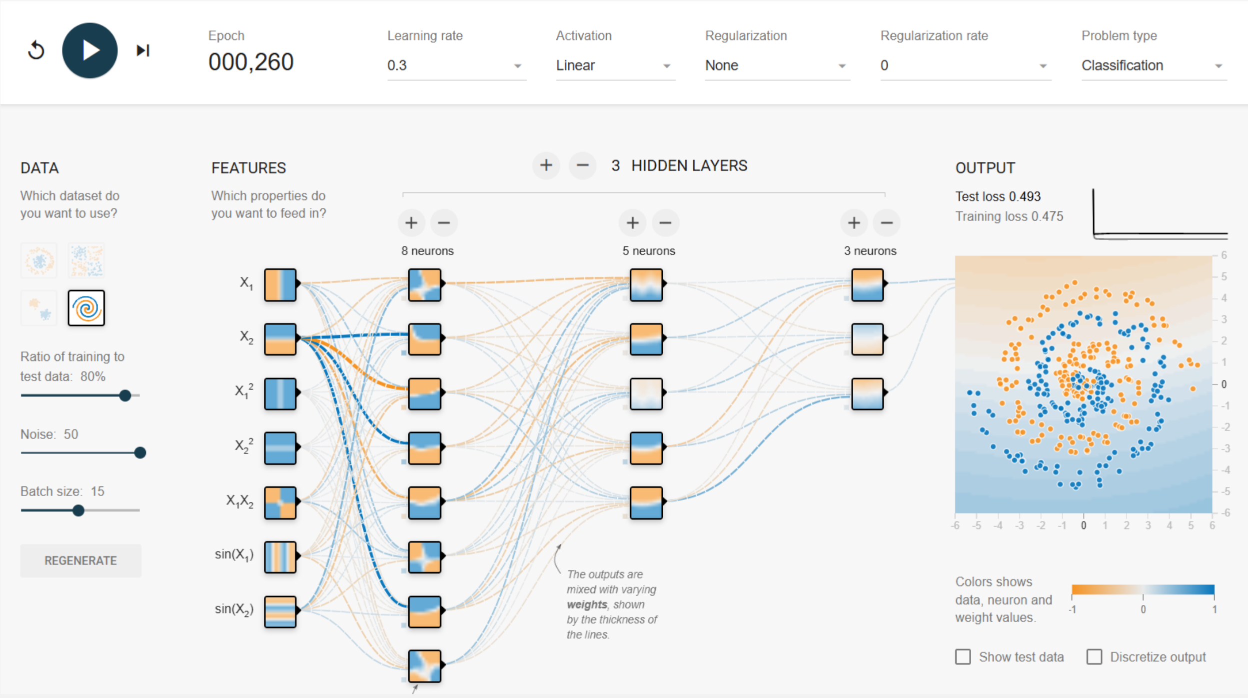
Playing with the XOR dataset, we can see on the edges of the data that the model could not figure out which one of the classes is the owner of that part and giving us a high error on unseen data.



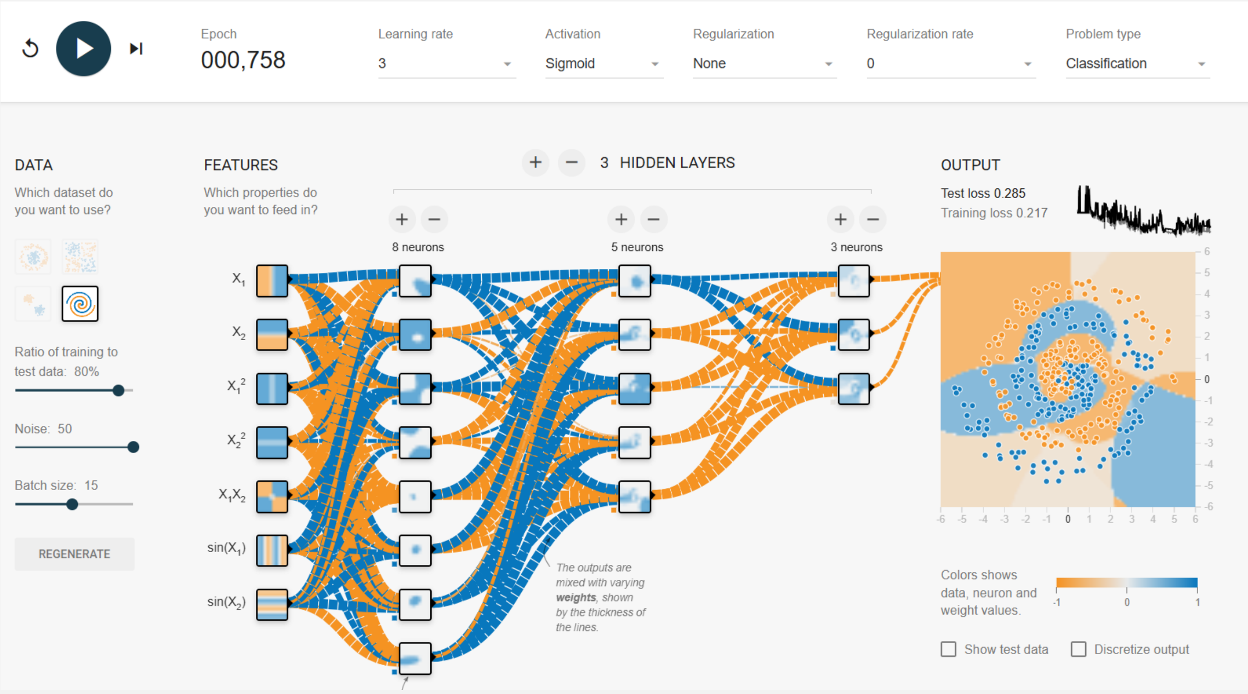
On the spiral dataset, we reach sever fluctuations after epoch 150 (see the graph for test and train loss). If we decrease the learning rate at that epoch to 0.1 from 0.3, and keep training for 100 more epochs, we reach this state:



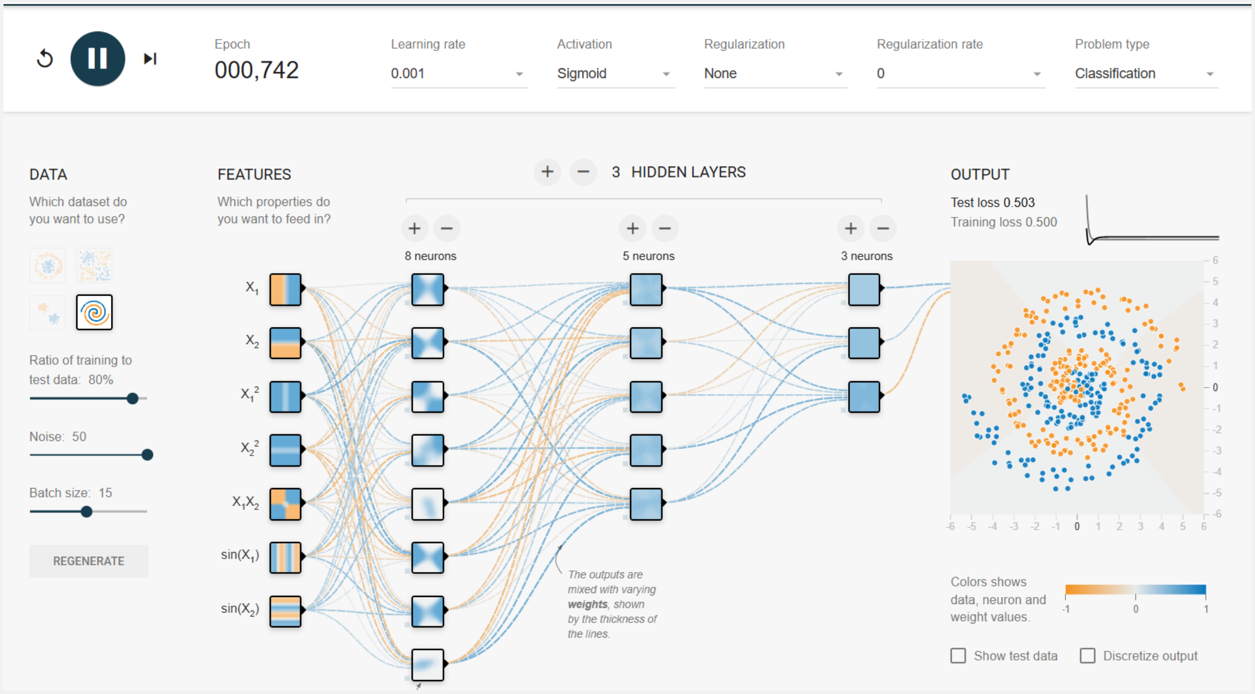
In this dataset, the high number of neurons is pretty perfect since the data is complex. Thusm it can learn the model and find sophisticated patterns. In almost all dataset, changin the activation function does not have that much effect on the learning process and we do not put that much energy compared to other parameters. However, only one activiation function is very bad—which is the linear activation function— that has no room for improvement of the model and the accuracy stuck at about 50%, regardless of how we continue training. See the graphs for test and training loss:



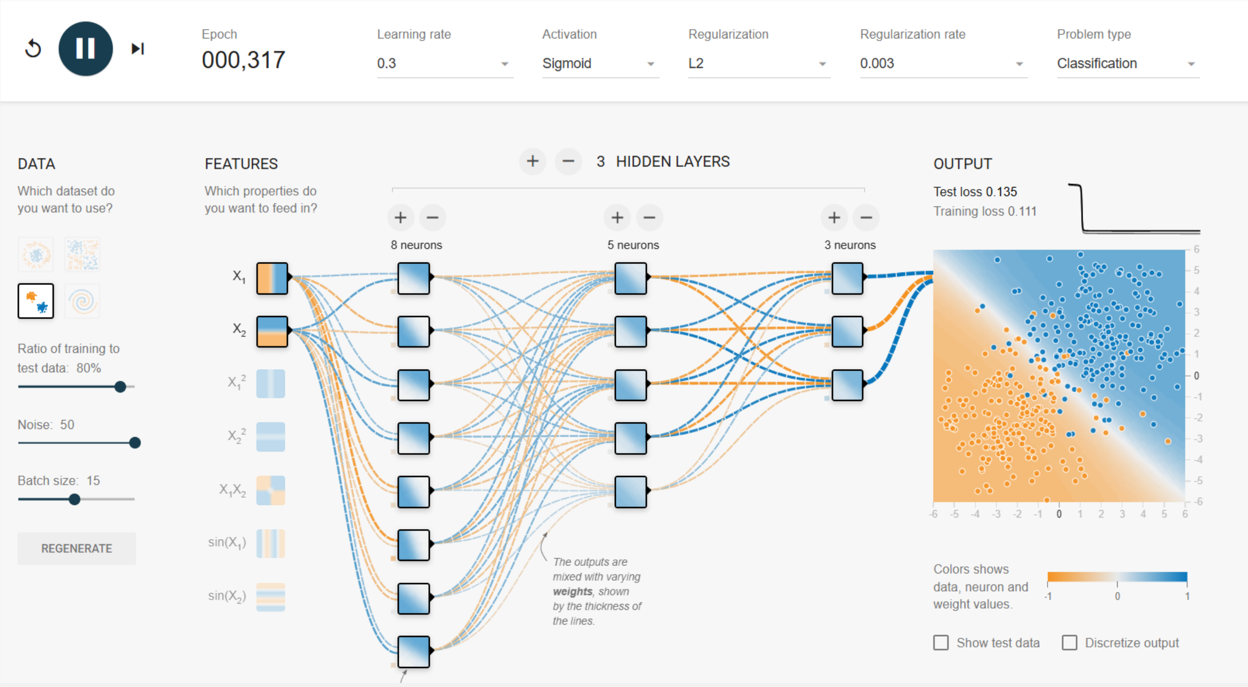
Having a large learning rate, such as 3 does not let the XOR, circle, and Gaussian dataset to reach a good point of accuracy. On Spiral, it takes a lot of epochs to decrease the loss, with lots of fluctuations. Here is the output of the model:



On the other hand, if we set a very low learning rate, the training process take a long time with no major change on the accuracy. In Fact, it seems that the model freezes while using a very low learning rate at the value of 0.001. Here is an image of how model behaves while using a pretty low learning rate:

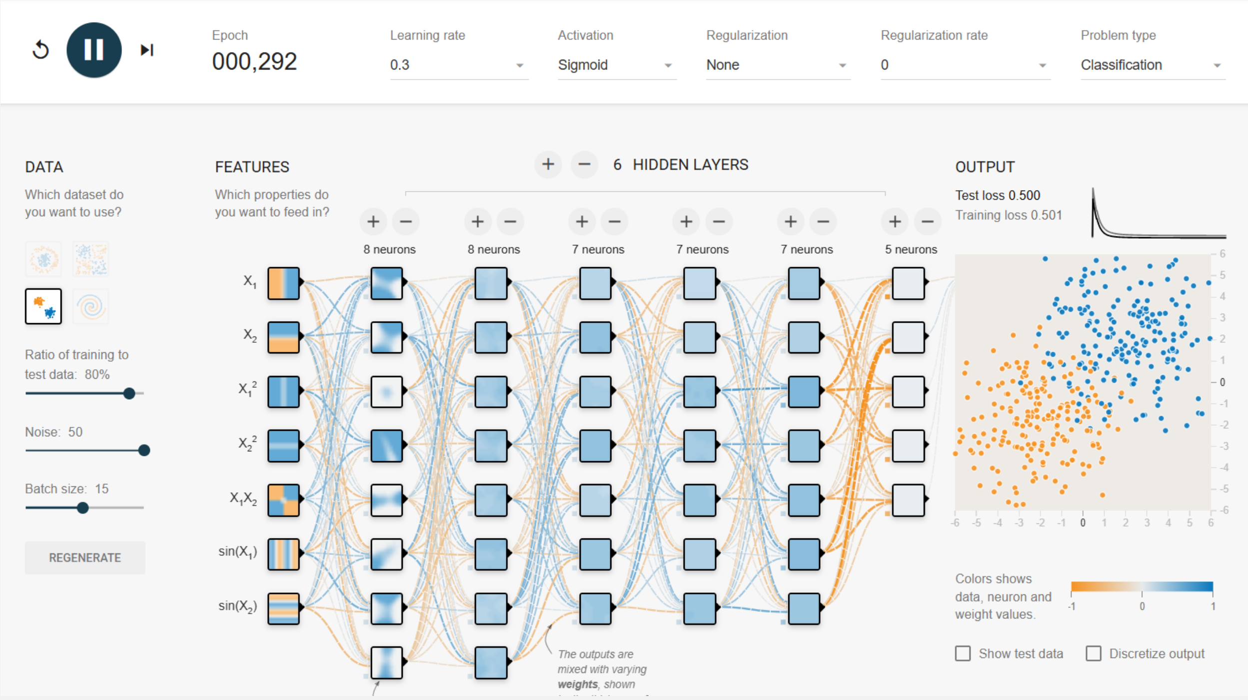


If we use L2 regularization with the rate of rate 0.3, we have to wait a little for the model to find the patterns. After that time, the model suddenly shows pretty good loss on both train and test data and not any sign of overfitting or underfitting.

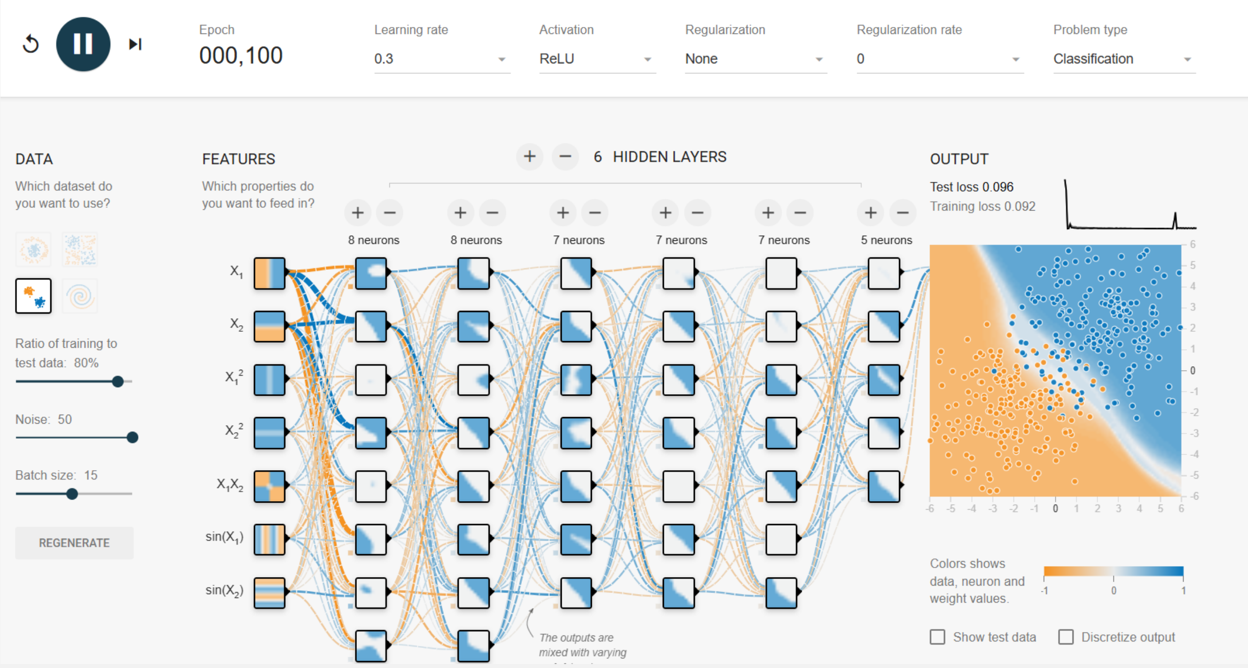


However, on some complex datasets, the model cannot find any pattern and seem to stuck on the model.

Please note that having more layers and neurons than what we already have, seems to have negative impacts on the network and does not let the model proceed, especially for more complex datasets such as spiral. Here is an image of the model with too much layers and neurons:

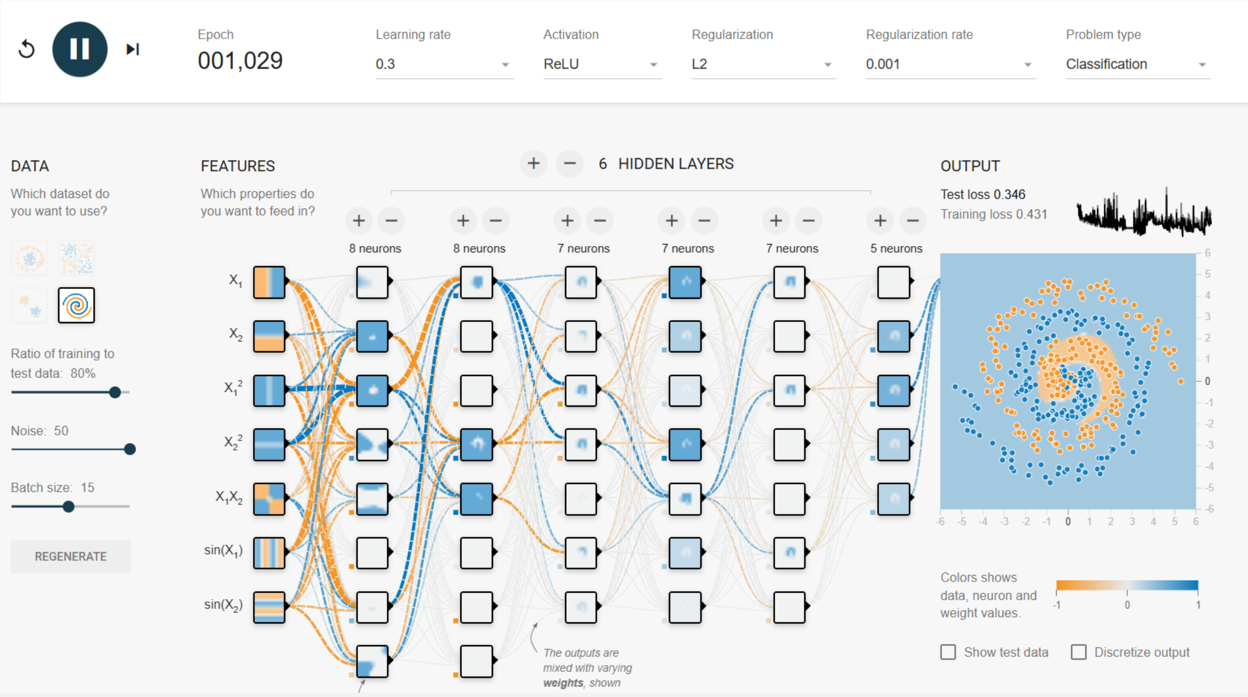


Changing the activation function from sigmoid to Relu will fix the problem and the model behaves like that with no overfitting or underfitting:



Training this model on this dataset for a ling time will result in a come back on the loss and accuracy, see the train and test graph:

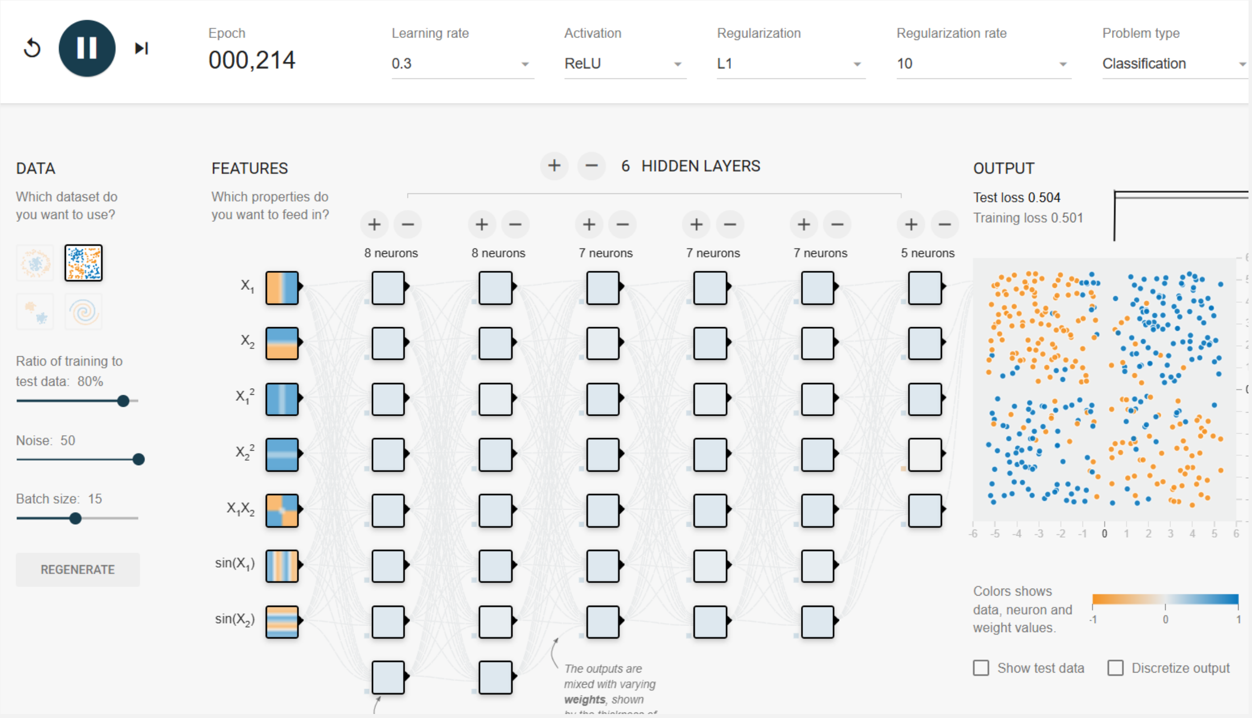
L2 regularization needs observation as to some level model behaves pretty improving and shows gradual improvements, but at a certain level, it starts fluctuating; like this:



This dataset looks like this, with a high amount of overfitting:

Testing with very large regularization will break the website 😊

But if we change it to L1, it will not do anything at all and results in underfitting:

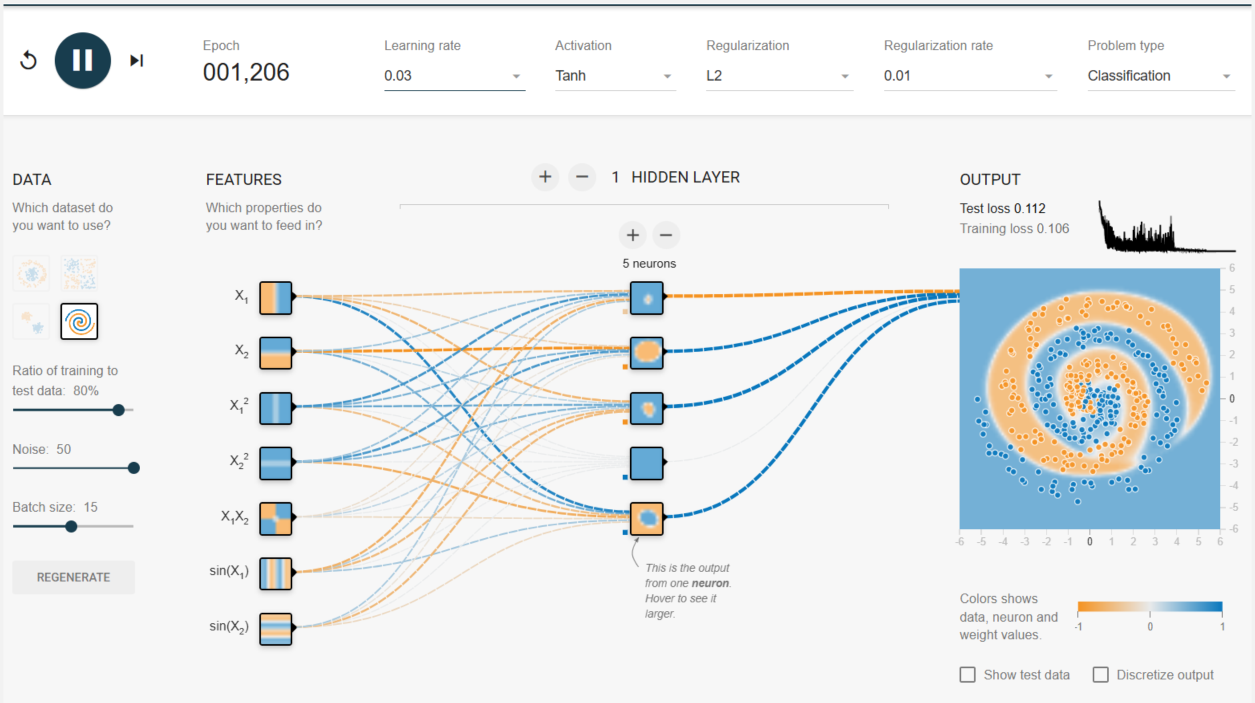


On the spiral dataset, it is amazing that we have better accuracy on test data:

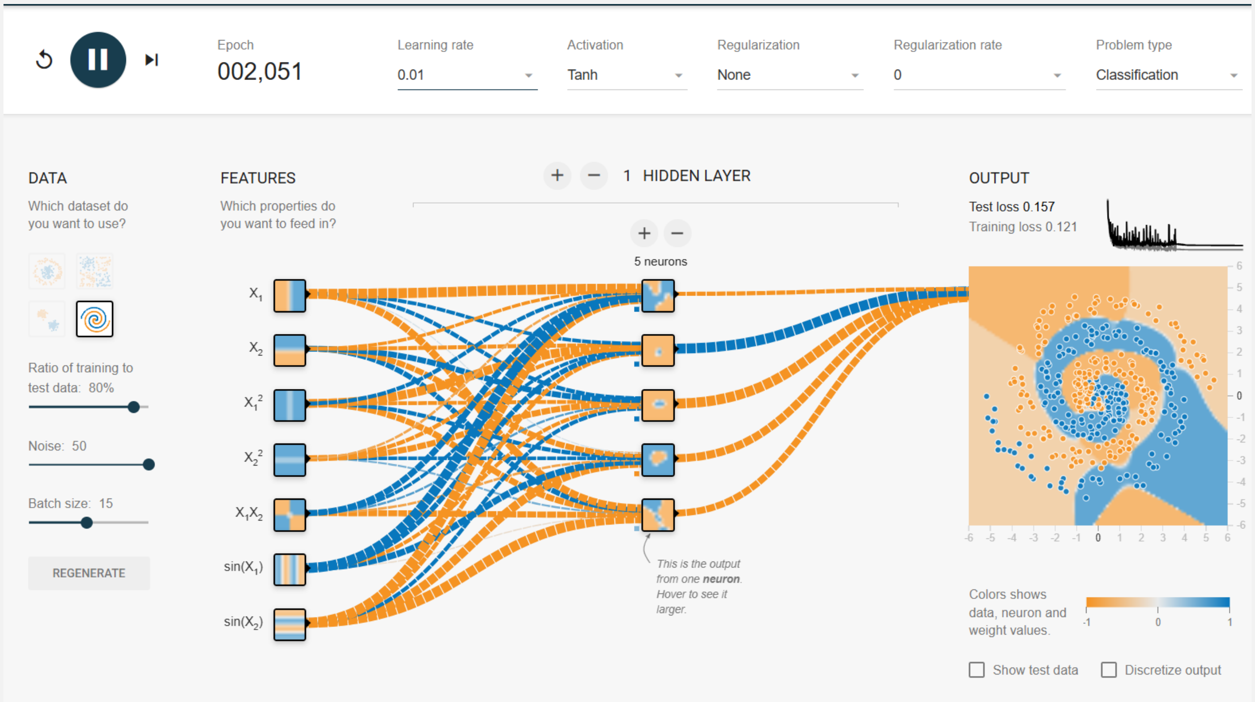
Weight Shrinking is happening when using L2 regularization as it penalizes large weights, shrinking them towards zero. This reduces the model's complexity and helps prevent it from fitting the noise in the training data.

It is not requested in the question, but to consider underfitting with very low layers and neurons, we have this. With a very low number of neurons and no hidden layers, the model cannot learn anything at all and this is called underfitting:

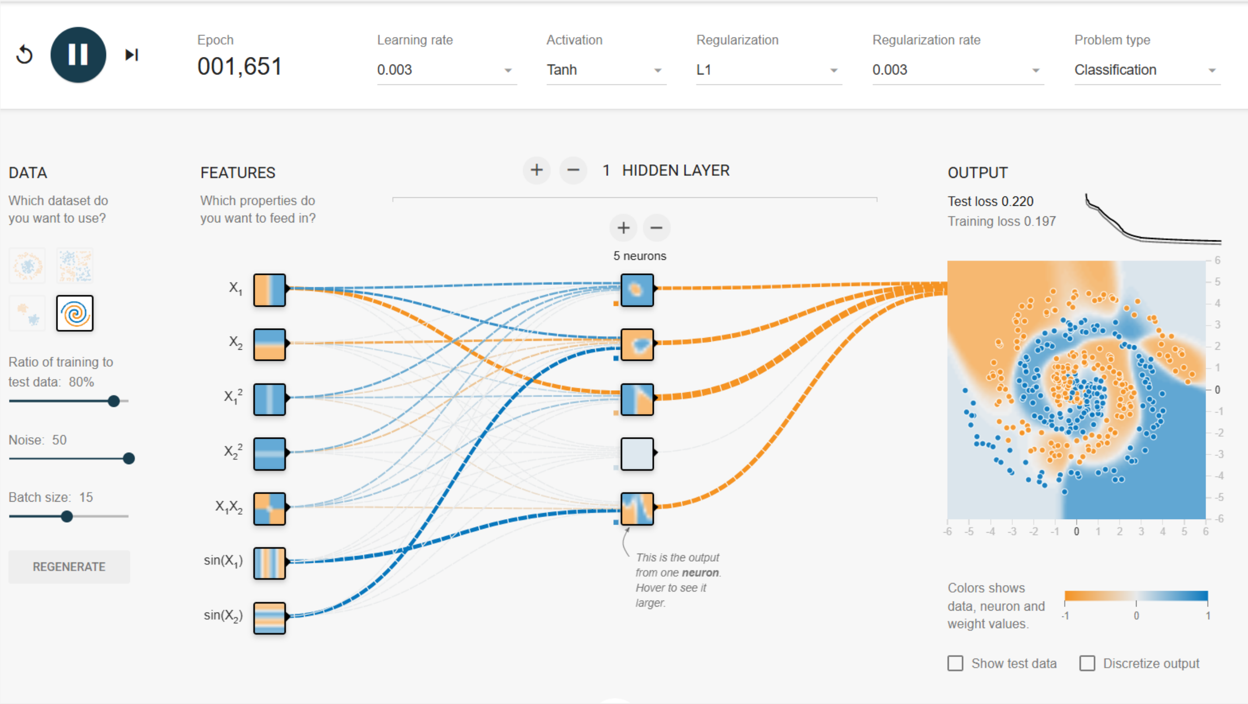
This is one of the best balances to capture the features of the spiral dataset. Please note that during this training, we degrade the learning rate as the training process proceeds:



If we do the same but with no regularization, we reach this:



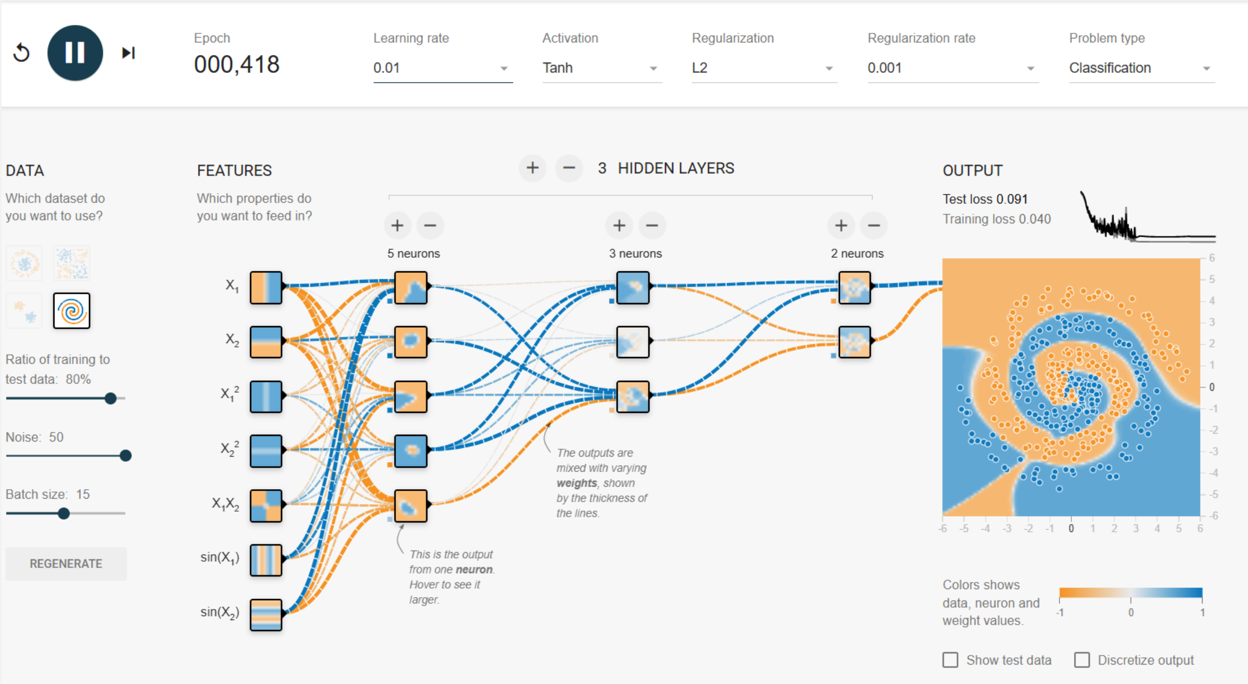
Having the L1 regularization will result in this:



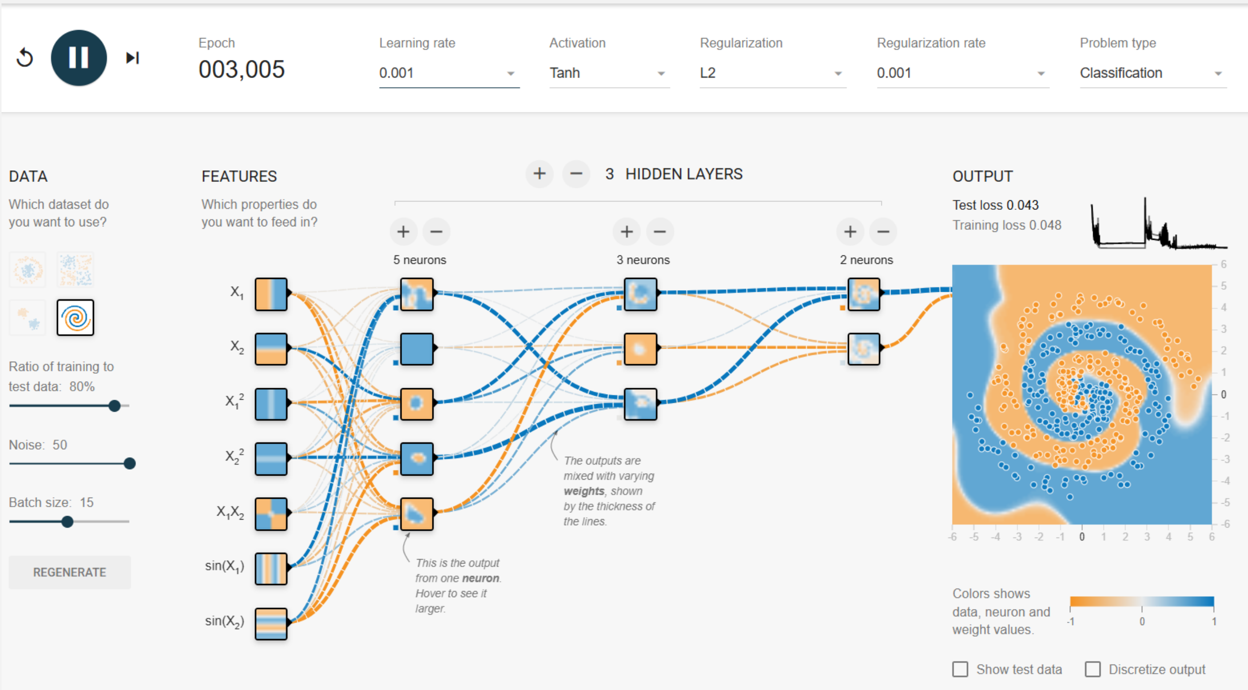
Conclusion

Finnaly, to have a good model without underfitting or overfitting, we have to set a appropriate number of layers as well as neurons. In addition to that, for some datasets, to have a good train and test loss, it is necessary to set a good numher for L1 or L2 regularization. Having an activation function such as

Tanh or RelU helps a lot. And a high number of epochs is the foundation of a good model along the correct orientation of the parameters. It means that if we do not set parameters appropriately, it is not matter how long we let the model train, we never reach a good test and train loss. The final point is that, if our data is noisy, we cannot reach to very high accuracies. The only thing that we can do is to wisely set the hyperparameters and let the model continue enough. The best model that I could make is this:



After suddenly incresing the learning rate to give a shock to the network, we reach this:



This model has a gradual decrease of the number of neurons as the layers goes on. L2 regularization with the value of 0.001. Tanh activation function and more that 3,000 epochs. I also change the learning as the training goes on. Having enough inputs, an appropriate number of layers and neurons in each, and have only two neurons at the final layer, with a perfect regularization, and large number of epochs are the keys of this perfect loss.

Problem 3

In this question we worked on fashion mnist dataset, that contains image of clothes and things related to fashion, at the size of 28*28 pixels. This dataset has 10 classes. This dataset is pre-splitted into 60K images for training and 10K for test. All of the images are grayscale. The dataset contains images from these categories:

T-shirt/top	Trouser	Pullover	Dress	Coat
Sandal	Shirt	Sneaker	Bag	Ankle boot

Data Preparation

Showing some images

Here is a series of images available at the dataset.

Preprocessing

For preprocessing, we divide the pixel of each image by 255 to make sure that they are at the range of 0 to 1. This, by experience, shows that gives us a better result in comparison to the time that we feed the images with the pixel values of 255.

To do so, we use this simple line of code:

```
# Normalize images
train_images, test_images = train_images / 255.0, test_images / 255.0
```

We could also apply more preprocessing such as dilation, erosion, threshholding, and such stuff that commonly are applied to image on image-processing tasks. But having these is just fine for our simple model.

Model Implementation

layers

As requested by the question, we implement a simple linear neurla network. This model has three layers: first is input layer with size of 724 neurons (28*28 pixels for each image); the second layer, which is our first hidden layer, has 128 neurons and the ReLU activation function; the last layer, which is called output layer, has 10 neurons, each for a single class of the fashion mnist dataset.

The shape and number of parameters of the model that I have created is shown in this image:

Layer (type)	Output Shape	Param #
flatten_2 (Flatten)	(None, 784)	0
dense_8 (Dense)	(None, 128)	100,480
dense_9 (Dense)	(None, 10)	1,290

Total params: 101,770 (397.54 KB)
Trainable params: 101,770 (397.54 KB)
Non-trainable params: 0 (0.00 B)

Activation Function

For the second layer (the first hidden layyer), we use ReLU activation functio. For the output layer, the activation function is softmax. The reason that we chose ReLU for activation function of the second layer is that it is normally works fair well on neural networks and also it does not have computational overhead i.e. it is inexpensive to run and facilitate the learning process, in both speed and accuracy. *Unlike sigmoid or tanh functions, which squash inputs to a small range, ReLU has an unbounded positive range. This prevents gradients from becoming extremely small during backpropagation,* *allowing networks to learn faster and with more stability in deeper layers.* In addition to that, *ReLU introduces sparsity in the network by setting all negative values to zero. This "sparsity" leads to more efficient computations and a form of regularization, as it reduces interdependencies between neurons and decreases overfitting.*[1]

The final layer has the softmax activation function becase in classification tasks that we need to output one class as the result of the problem, it is the best option. The other option is hardmax, which is a little more different. While using hardmax, we cannot see the probabilities of each class, hence, being aware of the sureness of the model for its predictions. Moreover, we cannot calculate the derivation of the output of the hardmax, thus, we cannot use gradient descent for backpropagation.

Optimizer & Loss Function

We use adam optimizer during the training and the 'sparse_categorical_crossentropy' for loss calculation. The reason behind the choice of Adam is that it is perfect for large dataset with high dimensions. Moreover, it is effective for noisy datasets, and has few parameters for tuning, which makes it a preferable default choice that does not have a lot of parameters to be set manually.

The reason behind the loss function is that we have a multi-class dataset and a multi-class classification; for these kinds of problems, that is one of the best options. The difference between "sparse_categorical_crossentropy" and "categorical_crossentropy" is that the first one is optimized and programmed to be able to accept integers as inputs, rather than one hot encoding (which is in the second one). Therefore, it is more memory-friendly for large datasets with lots of classes, altough we don't have that much class in this problem.

Training and Evaluation

We train the model in 35 epochs. Starting from:

```
accuracy: 0.8103 - loss: 0.5584 - val_accuracy: 0.8165 - val_loss: 0.5214
```

at the first epoch and reaching to this at the 35th epoch:

```
accuracy: 0.8840 - loss: 0.3228 - val_accuracy: 0.8573 - val_loss: 0.5595
```

The accuracy for test data (unseen samples) is like this:

```
accuracy: 0.8450 - loss: 0.5461
```

Moreover, we have these classification report from the library:

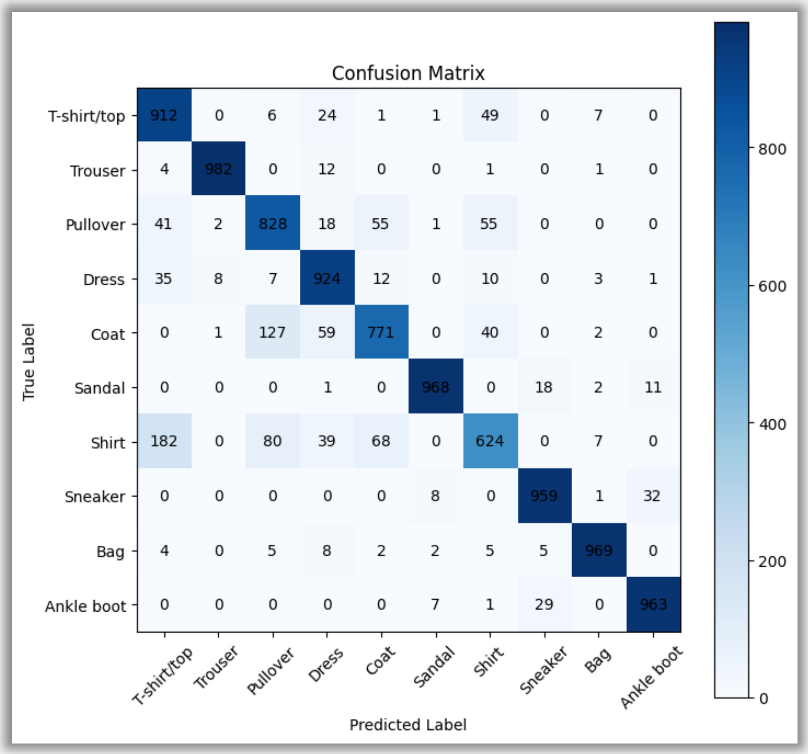
[1] The *italic* text is generated by AI.

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8450 - loss: 0.5461
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
              precision    recall  f1-score   support

 T-shirt/top       0.82      0.81      0.81      1000
     Trouser       0.97      0.96      0.97      1000
    Pullover       0.80      0.67      0.73      1000
       Dress       0.84      0.84      0.84      1000
        Coat       0.64      0.87      0.74      1000
      Sandal       0.96      0.94      0.95      1000
       Shirt       0.59      0.53      0.56      1000
     Sneaker       0.94      0.94      0.94      1000
         Bag       0.98      0.92      0.95      1000
  Ankle boot       0.95      0.95      0.95      1000

    accuracy                           0.84     10000
   macro avg       0.85      0.84      0.84     10000
weighted avg       0.85      0.84      0.84     10000
```

As we see, the accuracy for the model is about 84.50%, which is a relatively good percentage for such a simple model.

Confusion Matrix

The confustion matrix is calculated and shown at this image. We can clearly see that the model struggles with predicting the images of Shirt and T-Shirt, since they are pretty the same in the shape, size, and probably, orientation. In some other classes, it is still the same, like Coat and Pullover. With adding more neurons and layers, the model can learn more patterns and features and probably be able to distinguish between these classes.



Accuracy

The first metric is accuracy with the formula below:

We have to be careful while using and depending on this metric as it may be misleading on datasets that are imbalanced. It means that if we have the majority class with 90% of samples in the dataset, the metric can deceive us and give considerably good results, while the model is poorly performing.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

Precision

Precision is the ratio of true positive predictions to the total predicted positives. It measures how many of the predicted positives were actual positives.

The formula is like this:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Precision is very important in cases that the cost of false positives is high. Such as cancer detection.

Recall (Sensitivity)

Recall, or sensitivity, is the ratio of true positive predictions to all actual positives. It measures how many of the actual positives the model correctly identified.

The formula is like this:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

F1-Score

The F1-score is a combination of precision and recall, providing a single metric that balances both of them. It is useful when we need a metric that considers both precision and recall.

The formula is like this:

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

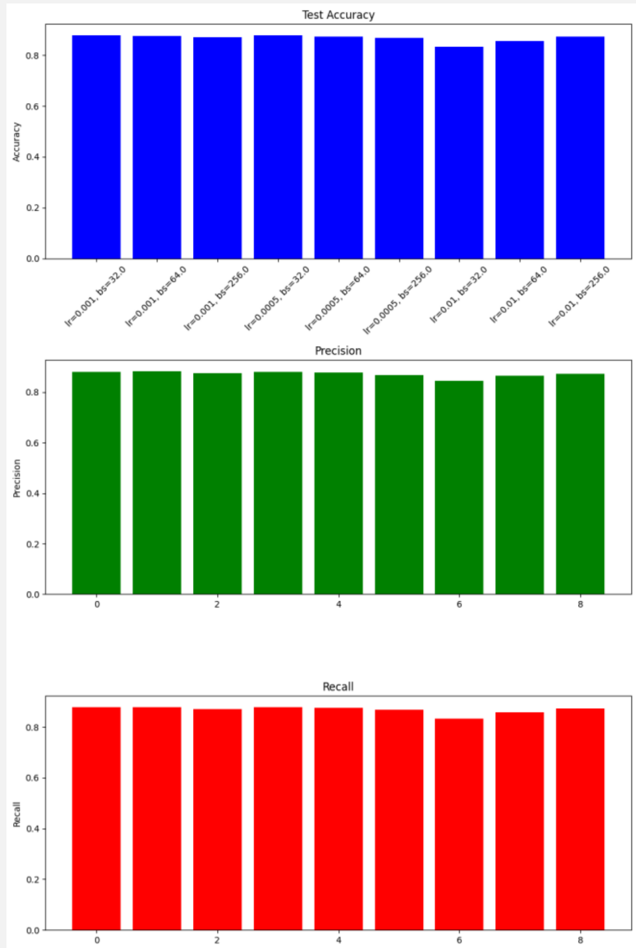
Hyperparameter Tuning

Here, we wrote a code that systematically trains and evaluates the model with different hyperparameters, such as learning rate and batch size. Each learning rates of [0.001, 0.0005, 0.01] will go with each of batch size of [32, 64, 256].

Here are the results:

Learning Rate	Batch size	Epochs	Accuracy	Precision	Recall	F1-Score
0.001	32	10	87.91	88	88	88
0.001	64	10	87.69	88	87	87
0.001	256	10	87.02	87	87	86
0.0005	32	10	87.80	88	87	87
0.0005	64	10	87.45	87	87	87
0.0005	256	10	86.79	86	86	86
0.01	32	10	83	84	83	83
0.01	64	10	85.72	86	85	85
0.01	256	10	87	87	87	87

Here are some graphs, comparing these results:



Reflection

This question highlights the importance of hyperparameter tuning in model performance.

Learning Rate

Affects convergence speed and stability. An optimal learning rate is very important to ensure that the model reaches a good performance.

Batch Size

Influences the model's ability to learn fine details in the data. Smaller batch sizes often lead to better performance, although at a computational cost. That's probably because the model has more chances to fix the weights.

Epochs

Directly impact how long the model has to learn from the data. This experiment suggests that certain configurations might benefit from additional training, i.e. if we set poor hyperparameters it is not important how long we let the model to be trained on data, we never reach a good point. On the other hand, if we use meticulously chosen hyperparameters, we can be sure that if we let the training continues, we can achieve high performance and accuracies.

Problem 4

After implementing the question and adding data to the code, we achieve this:

```
import numpy as np

input_colors = np.array([
    [255, 0, 0],     # Red
    [0, 255, 0],     # Green
    [0, 0, 255],     # Blue
    [255, 255, 0]    # Yellow
])

# initial weights for the 2x2 sofn nodes
weights = np.array([
    [100, 100, 100],
    [150, 150, 150],
    [50, 50, 50],
    [200, 200, 200]
])

learning_rate = 0.1
epochs = 100
map_size = (2, 2)
radius = 1
```

Find BMU

BMU or Best Matching Unit is the node that is nearest to the input color. In here, we are talking about weights.

Get neighbors

This function determines which nodes in the grid are considered neighbors of the BMU based on the defined neighborhood radius.

Training

During training, first we find the BMU or best matching unit to the input color. This is done with the aid of find_bmu function. Here we feed in the four colors—red, green, blue, yellow—that we have to the network. For each color, first we find the nearest node and then find the neighbors based on the radius that we have defined which uses Euclidean Distance. After finding them, we start to update the weights of the node and its neighbors.

The formula for updating the weights is:

$$w_i(t+1) = w_i(t) + \eta.(x - w_i(t)$$

The difference and the previous wieghts are added because we need to move toward the new input.

Final weights

After all, we have these weights:

```
Trained weights for each node after competition, cooperation, and adaptation:
Node 1 weight: [72      81      90]
Node 2 weight: [ 90     165     81]
Node 3 weight: [165     90      81]
Node 4 weight: [165     174     0]
```

Stability

The weights of the nodes have become relatively constant after several iterations of training.

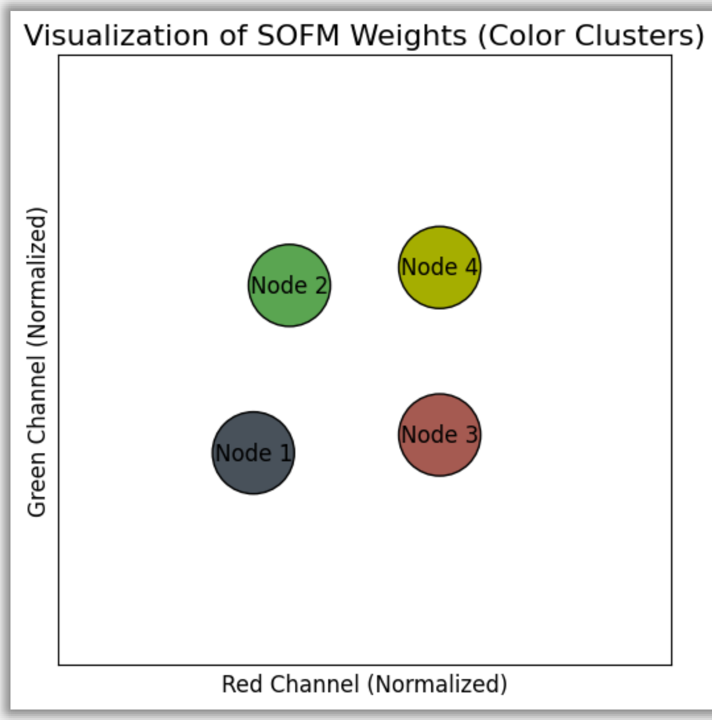
```
Node 1 weight: [72 81 90]
Node 2 weight: [ 90 165  81]
Node 3 weight: [165  90  81]
Node 4 weight: [165 174   0]
```

1. Red (255, 0, 0)
2. Green (0, 255, 0)
3. Blue (0, 0, 255)
4. Yellow (255, 255, 0)

Considering the weights of the colors and weights of the nodes, we can make guesses that which node is related to which color. Here are the guesses:

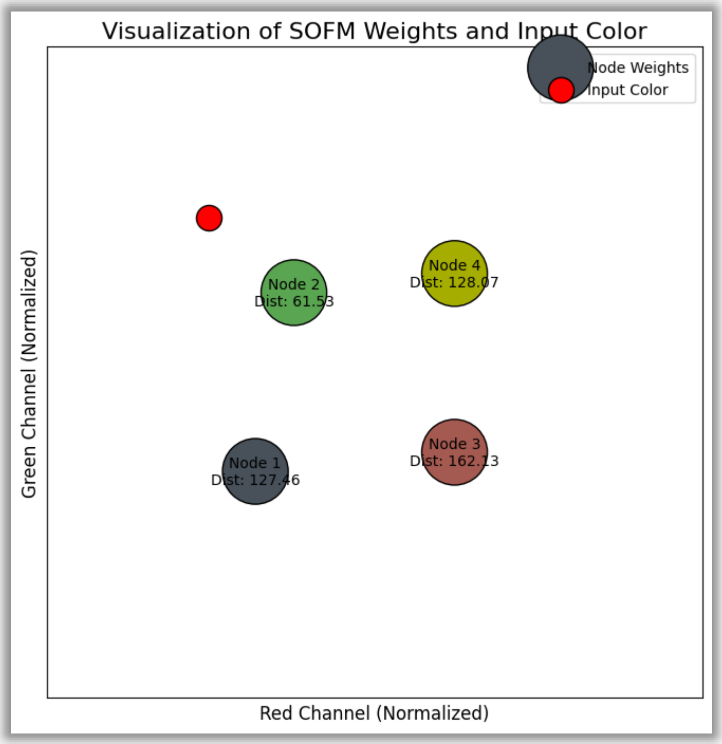
- *Node 1* is probably related to the Blue, since the third value is the highest among other values for this node
- *Node 2* is most likely related to the Green, as its middle value is very high compared to the other two
- *Node 3* seems to be for the Red, because its first value is larger than two others
- *Node 4* is for sure related to Yellow, seeing the first and second values are high and the last one is 0

Each node acts as a centroid for a group of similar colors. Say, when new color data is input into the system, the SOFM can determine which node is the closest (Best Matching Unit or BMU) based on the Euclidean distance that we already saw the function between the input color and the weights of each node. By calculating that, we can classify each new color to one of the four colors. However, we may encounter some errors since the number of our colors is very limited and other colors may have overlap with other colors. The true color of the nodes is shown at the graph below.[2]

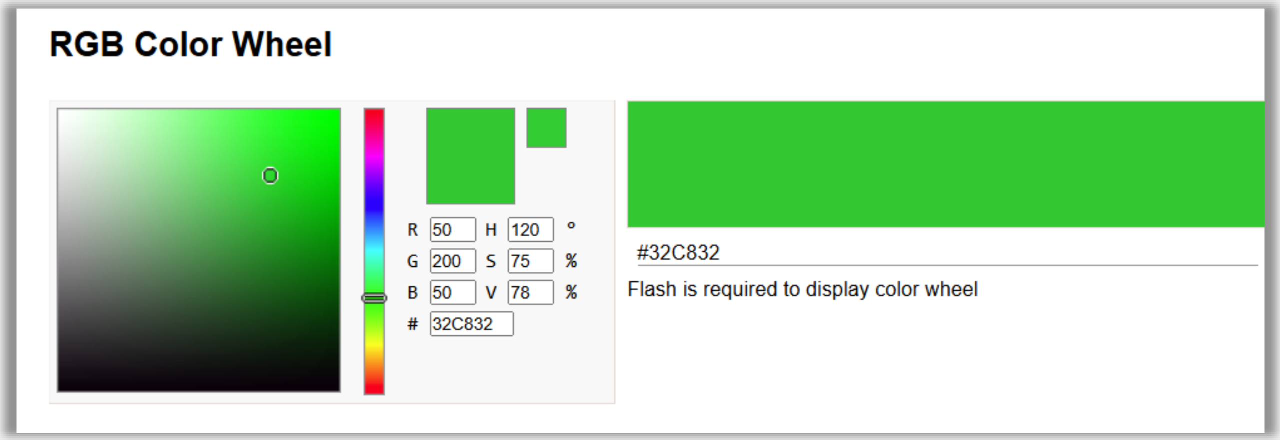


[2] Code is generated with the aid of A.I.

If we test a custom input, it will result in this:



The color that was fed into the system (red dot shown on the image) was this:



For any other color clustering, we can feed the colors into the system and check whether it is related (nearer) to which node? Then we cluster that color based on the nearest distance to each node.

Problem 5

In this problem we will delve into the programming a SOM that can organize colors based on the values of RGB.

Dataset creation

A simple NumPy code will do that for us in one line. We print out some samples to make sure it is working fine.

```
[16]:  import numpy as np

       random_rgb_colors = np.random.rand(256, 3)
       random_rgb_colors[:5]

[16]:  array([[0.70191319, 0.64374983, 0.98927134],
              [0.05158297, 0.69808881, 0.23500443],
              [0.40098102, 0.88149189, 0.04738057],
              [0.28342378, 0.32786944, 0.67564838],
              [0.58476278, 0.46543854, 0.55999761]])
```

SOM Initialization

With this class and code, we can initialize the data we need and further access them easily.

Please note that as we further call the constructor of the class and pass the value of 10 and 10 to width and height respectively, we do not define the number of neurons explicitly here.

```python
class SelfOrganizingMap:
    def __init__(self, width, height, input_len,
                 lr=0.1, radius=5, epochs=500):
        self.width = width
        self.height = height
        self.lr = lr
        self.radius = radius
        self.epochs = epochs
        self.input_len = input_len

        # init weights
        self.weights = np.random.rand(width * height,
                                      input_len)
```

Training Parameters

Train

This function will do the work for us. First, it loops for 500 times, the number of epochs. Then, it loops over the data—which is the 256 random colors that we have made already randomly—.

```
def train(self, data):
    for _ in range(self.epochs):
        for in_vec in data:
            bmu_idx = self._find_bmu(in_vec)
            self._update_weights(in_vec,
                                 bmu_idx)
```

After thet, it calculates the Euclidean distance between the input vector and each neuron's weight, which identifies the neuron with the smallest distance.

Update: I used *tqdm* library to visualize the training process, but I didn't update the screenshot here.

_find_bmu

The norm functino computes the Euclidean distance between the input vector and each weight vector along the first axis, returning a 1D array where each entry represents the distance to a specific neuron.

```
def _find_bmu(self, input_vector):
    distances = np.linalg.norm(
        self.weights - input_vector,
        axis=1) #this calculates EUCLIDIAN
    return np.argmin(distances) # nearest
```

_update_weights

```
def _update_weights(self, input_vector, bmu_idx):
    bmu_x, bmu_y = divmod(bmu_idx, self.width)

    for i in range(self.weights.shape[0]):
        x, y = divmod(i, self.width)
        distance_to_bmu = np.sqrt( ((x - bmu_x) ** 2) + ((y - bmu_y) ** 2) )#this calculates EUCLIDIAN


        # Only neurons within this radius have their weights updated
        if distance_to_bmu <= self.radius:
            t = np.exp(-distance_to_bmu ** 2 / (2 * (self.radius ** 2))) # Gaussian neighborhood function
            # Neurons closer to the BMU have a higher influence (closer to 1),
            # while neurons farther away within the radius have a lower influence.
            self.weights[i] += t * self.lr * (input_vector - self.weights[i])
```

This part of code has comments and they are enough to make it clear what it does.

Run

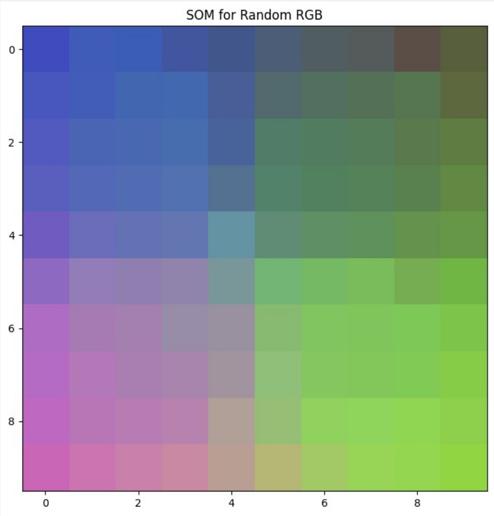
After doing all of these, we make an instance from the class and initialize it, then we run the code and wait until it gets done.

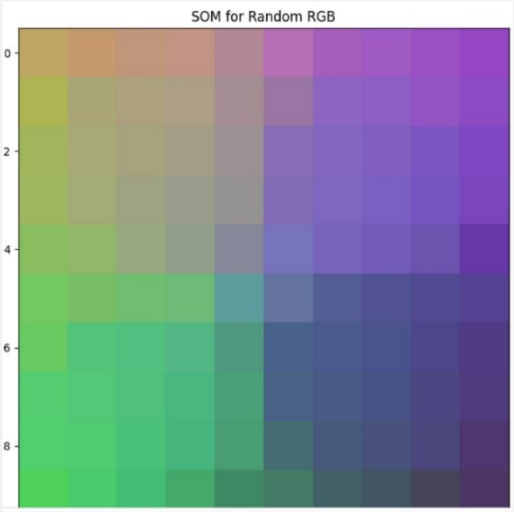
Visualization[3]

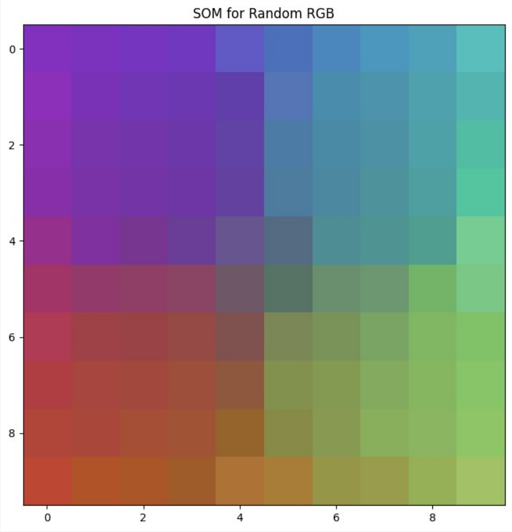
This small code will visulize the result for us.

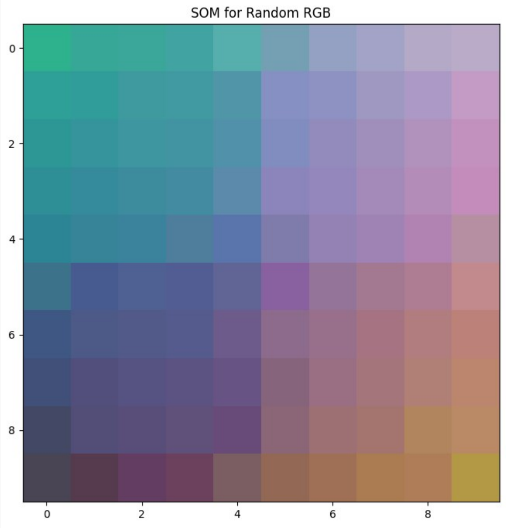
```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 8))
plt.imshow(trained_weights_fixed)
plt.title("SOM for Random RGB")
plt.show()
```

After running the code several times, we have these images. As they are initiated randomly, they are different.







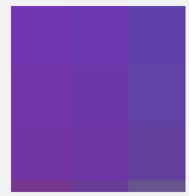


[3] Some of the codes for visualization created using AI.

Analysis

The colors are organized and clustered in a way that same colors are near to each other and they move from ine shade to another shade smoothly, making a beautiful color map that worth training several times to see other kind of arrangements and colors. By looking more thoroughly, we can see that in most cases, the colors of purple and blue are grouped together, and shades of green and yellow and light blue are grouped together. That's because they are very similar in colors and their codes i.e. RGB valuse are very close to each other. Thus, they will be assigned to neurons with the same value while mapping them to the SOM. If we add more neurons to the map, we will have a smoother map and the colors will be changing more gradually, making even more beautiful map.

Talking about effectiveness, I guess the map could be better in some ways. It means that there are some neurons at the map that their colors are almost identical, but the map assigned different neurons for clustering those colors. The image above shows a sample of what I mean. Of course they are a little different but they are very close. However, one of the reasons for this is because we have limited amount of training data, only 256 samples. Therefore, if we have lots of purple colors, they will occupy large part of the map and cause this. Adding more samples to the training data or distributing them more widely will probably fix this problem, as we can see in other maps that provided above, with a even distribution, the colors may not be this close, making a well-separated map.



Total Words in this document: 4892

Total editing time: 538 Minutes (about 9 Hours)

With thanks and respect,

Hesam Koucheki

403 7239 87