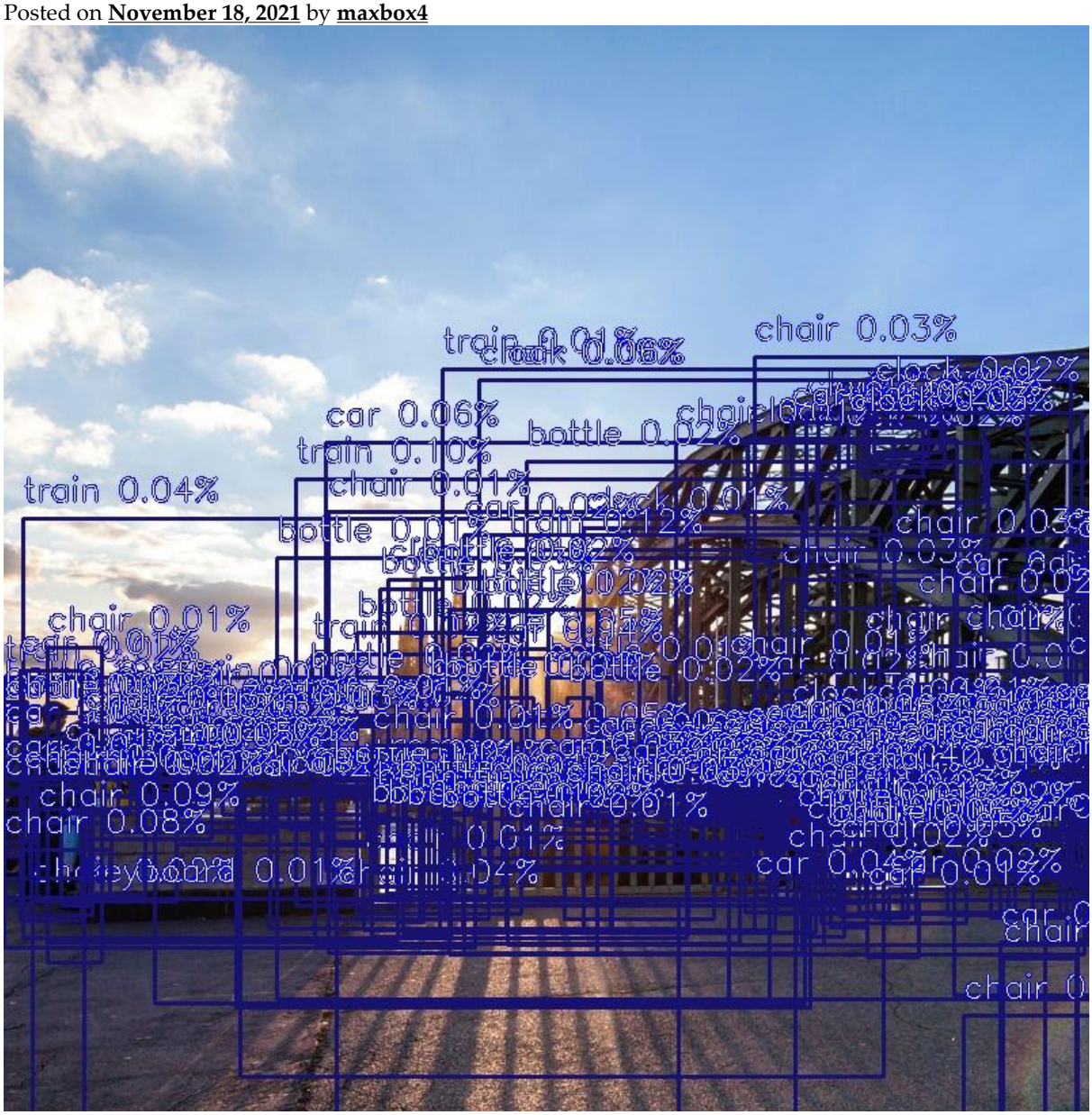
Welcome to maXbox Code (Blog)

maXbox

VisualLearn

Posted on **November 18, 2021** by **maxbox4**



Learning boolean functions XOR, AND and OR with the CAI NeuralNetwork Framework

This example has these main steps:

- Preparing training data
- Creating the neural network
- Fitting

- Printing a test result
- Visualize inference

These are the inputs and expected outputs:

```
const inputs : TBackInput =
  ( // x1,   x2
    ( 0.1,  0.1), // False, False
    ( 0.1,  0.9), // False, True
    ( 0.9,  0.1), // True,  False
    ( 0.9,  0.9)  // True,  True
  );

const reluoutputs : TBackOutput =
  (// XOR, AND,   OR
    ( 0.1, 0.1, 0.1),
    ( 0.8, 0.1, 0.8),
    ( 0.8, 0.1, 0.8),
    ( 0.1, 0.8, 0.8)
  );
```

The first row in `reluoutputs` has expected outputs for **XOR**, **AND** and **OR** boolean functions while the first row of `inputs` contains inputs for these 3 boolean functions. All 3 boolean functions will be trained together in this example.

This is how the training data is prepared with training pairs (input,output):

```
TrainingPairs := TNNetVolumePairList.Create();
...
    for Cnt := Low(inputs) to High(inputs) do
    begin
      TrainingPairs.Add(
        TNNetVolumePair.Create(
          TNNetVolume.Create(vInputs[Cnt]),
          TNNetVolume.Create(vOutput[Cnt])
        )
      );
    end;
```

In this example, values smaller than 0.5 mean **false** while values bigger than 0.5 mean **true**. This is called **monopolar** encoding.

CAI also supports bipolar encoding (-1, +1). Please have a look directly into the source code at these 2 methods:

- `EnableMonopolarHitComparison()`
- `EnableBipolarHitComparison()`

This is how the neural network is created:

```
NN := TNNet.Create();
...
NN.AddLayer( TNNetInput.Create(2) );
NN.AddLayer( TNNetFullConnectReLU.Create(3) );
NN.AddLayer( TNNetFullConnectReLU.Create(3) );
```

As you can see, there is one input layer followed by 2 fully connected layers with 3 neurons each.

This is how the fitting object is created and run:

```
NFit := TNeuralFit.Create();
...
    NFit.InitialLearningRate := 0.01;
    NFit.LearningRateDecay := 0;
    NFit.L2Decay := 0;
    NFit.Verbose := false;
    NFit.HideMessages();
    NFit.Fit(NN, TrainingPairs, nil, nil, {batchsize=}4, {epochs=}3000);
```

The neural network is then tested for each input with:

```
// tests the learning
for Cnt := Low(inputs) to High(inputs) do
begin
  NN.Compute(vInputs[Cnt]);
  NN.GetOutput(pOutPut);
  WriteLn
  (
    ' Output:',
    pOutPut.Raw[0]:5:2,' ',
    pOutPut.Raw[1]:5:2,' ',
    pOutPut.Raw[2]:5:2,
    ' - Training/Desired Output:',
    vOutput[cnt][0]:5:2,' ',
    vOutput[cnt][1]:5:2,' ' ,
    vOutput[cnt][2]:5:2,' '
  );
end;
```

Code in maXbox scripting

```
type TBackInput  = array[0..3] of array[0..1] of TNeuralFloat;
type TBackOutput = array[0..3] of array[0..2] of TNeuralFloat;

 {
const inputs : TBackInput =
  ( // x1,   x2
    ( 0.1,  0.1), // False, False
    ( 0.1,  0.9), // False, True
    ( 0.9,  0.1), // True,  False
    ( 0.9,  0.9)  // True,  True
  );

const reluoutputs : TBackOutput =
  (// XOR, AND,   OR
    ( 0.1, 0.1, 0.1),
    ( 0.8, 0.1, 0.8),
    ( 0.8, 0.1, 0.8),
    ( 0.1, 0.8, 0.8)
  );  }

  procedure DefinelogicalMatrix(var inputs:TBackInput; var routput:TBackOutput);
  begin
    Inputs[0][0]:= 0.1; Inputs[0][1]:= 0.1;
    Inputs[1][0]:= 0.1; Inputs[1][1]:= 0.9;
    Inputs[2][0]:= 0.9; Inputs[2][1]:= 0.1;
    Inputs[3][0]:= 0.9; Inputs[3][1]:= 0.9;

    routput[0][0]:= 0.1; routput[0][1]:= 0.1;  routput[0][2]:= 0.1;
    routput[1][0]:= 0.8; routput[1][1]:= 0.1;  routput[1][2]:= 0.8;
    routput[2][0]:= 0.8; routput[2][1]:= 0.1;  routput[2][2]:= 0.8;
    routput[3][0]:= 0.1; routput[3][1]:= 0.8;  routput[3][2]:= 0.8;
   end;

  procedure RunSimpleAlgo();
  var
    NN: TNNet;
    EpochCnt: integer;
    Cnt: integer;
    pOutPut: TNNetVolume;
    vInputs: TBackInput;
    vOutput: TBackOutput;
    inputs: TBackInput;  routput : TBackOutput;
    Rate, Loss, ErrorSum : TNeuralFloat;
  begin
   definelogicalMatrix(inputs, routput);
    NN := TNNet.Create();
    NN.AddLayer( TNNetInput.Create3(2) );
    NN.AddLayer( TNNetFullConnectReLU.Create30(3,0) );
    NN.AddLayer( TNNetFullConnectReLU.Create30(3,0) );
    NN.SetLearningRate(0.01, 0.9);

    vInputs := inputs;
    vOutput := routput;
  //constructor Create(pSizeX, pSizeY, pDepth: integer; c: T = 0); {$IFNDEF FPC} overload; {$ENDIF}
    pOutPut := TNNetVolume.Create0(3,1,1,1);
    for EpochCnt := 1 to 4000 do begin
      for Cnt := Low(inputs) to High(inputs) do begin
        NN.Compute68(vInputs[Cnt],0);
        NN.GetOutput(pOutPut);
        NN.Backpropagate70(vOutput[Cnt]);
        if EpochCnt mod 300 = 0 then
        WriteLn
        (
          itoa(EpochCnt)+' x '+itoa(Cnt)+
          ' Output:'+
          format(' %5.2f',[poutPut.Raw[0]])+' '+
          format(' %5.2f',[poutPut.Raw[1]])+' '+
          format(' %5.2f',[poutPut.Raw[2]])+' '+
         (* pOutPut.Raw[1]:5:2,' ',
          pOutPut.Raw[2]:5:2, *)
          ' - Training/Desired Output:'+
```

```
            format('%5.2f',[vOutput[cnt][0]])+' '+
            format('%5.2f',[vOutput[cnt][1]])+' '+
            format('%5.2f',[vOutput[cnt][2]])+' '
            {vOutput[cnt][0]:5:2,' ',
            vOutput[cnt][1]:5:2,' ' ,
            vOutput[cnt][2]:5:2,' ' }
          );
        end;
        if EpochCnt mod 300 = 0 then WriteLn('');
      end;
      // TestBatch( NN, pOutPut, 10000,  Rate, Loss, ErrorSum);

      //NN.DebugWeights();
      NN.DebugErrors();
      pOutPut.Free;
      NN.Free;
      Write('Press ENTER to exit.');
      //ReadLn;
    end;
```

Output:

Ref: simple logical learner
300 x 0 Output: 0.30 0.07 0.26 – Training/Desired Output: 0.10 0.10 0.10
300 x 1 Output: 0.40 0.29 0.62 – Training/Desired Output: 0.80 0.10 0.80
300 x 2 Output: 0.72 0.22 0.62 – Training/Desired Output: 0.80 0.10 0.80
300 x 3 Output: 0.37 0.56 1.00 – Training/Desired Output: 0.10 0.80 0.80

600 x 0 Output: 0.29 0.02 0.23 – Training/Desired Output: 0.10 0.10 0.10
600 x 1 Output: 0.52 0.28 0.62 – Training/Desired Output: 0.80 0.10 0.80
600 x 2 Output: 0.74 0.15 0.69 – Training/Desired Output: 0.80 0.10 0.80
600 x 3 Output: 0.27 0.66 0.96 – Training/Desired Output: 0.10 0.80 0.80

900 x 0 Output: 0.22 0.02 0.20 – Training/Desired Output: 0.10 0.10 0.10
900 x 1 Output: 0.65 0.23 0.66 – Training/Desired Output: 0.80 0.10 0.80
900 x 2 Output: 0.75 0.13 0.74 – Training/Desired Output: 0.80 0.10 0.80
900 x 3 Output: 0.18 0.72 0.90 – Training/Desired Output: 0.10 0.80 0.80

1200 x 0 Output: 0.15 0.04 0.16 – Training/Desired Output: 0.10 0.10 0.10
1200 x 1 Output: 0.73 0.18 0.72 – Training/Desired Output: 0.80 0.10 0.80
1200 x 2 Output: 0.78 0.13 0.77 – Training/Desired Output: 0.80 0.10 0.80
1200 x 3 Output: 0.13 0.75 0.85 – Training/Desired Output: 0.10 0.80 0.80

1500 x 0 Output: 0.11 0.06 0.12 – Training/Desired Output: 0.10 0.10 0.10
1500 x 1 Output: 0.79 0.15 0.77 – Training/Desired Output: 0.80 0.10 0.80
1500 x 2 Output: 0.80 0.12 0.79 – Training/Desired Output: 0.80 0.10 0.80
1500 x 3 Output: 0.10 0.77 0.82 – Training/Desired Output: 0.10 0.80 0.80

1800 x 0 Output: 0.10 0.07 0.11 – Training/Desired Output: 0.10 0.10 0.10
1800 x 1 Output: 0.80 0.13 0.79 – Training/Desired Output: 0.80 0.10 0.80
1800 x 2 Output: 0.80 0.11 0.80 – Training/Desired Output: 0.80 0.10 0.80
1800 x 3 Output: 0.10 0.78 0.81 – Training/Desired Output: 0.10 0.80 0.80

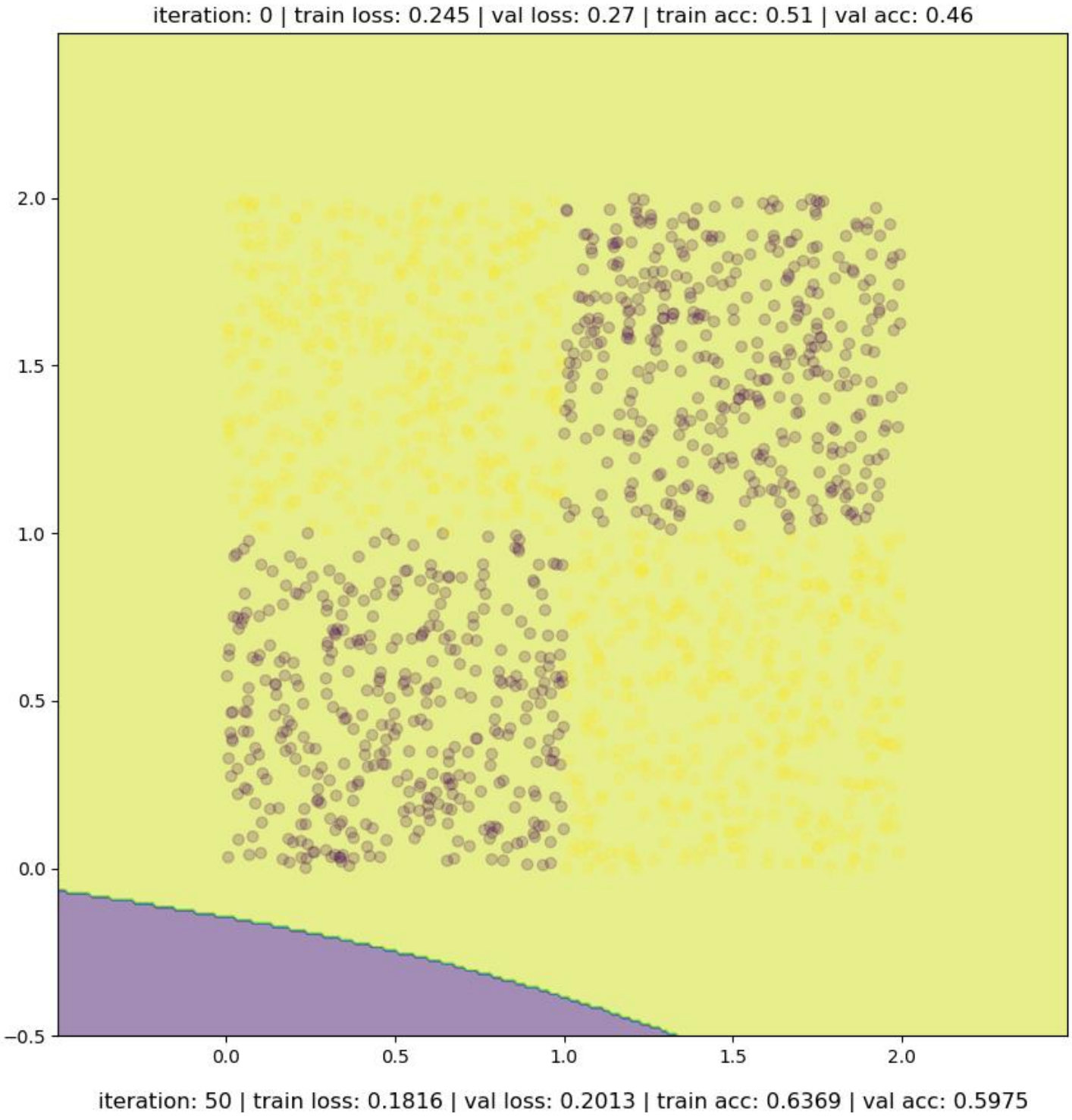
2100 x 0 Output: 0.10 0.09 0.11 – Training/Desired Output: 0.10 0.10 0.10
2100 x 1 Output: 0.80 0.12 0.79 – Training/Desired Output: 0.80 0.10 0.80
2100 x 2 Output: 0.80 0.11 0.80 – Training/Desired Output: 0.80 0.10 0.80
2100 x 3 Output: 0.10 0.79 0.80 – Training/Desired Output: 0.10 0.80 0.80

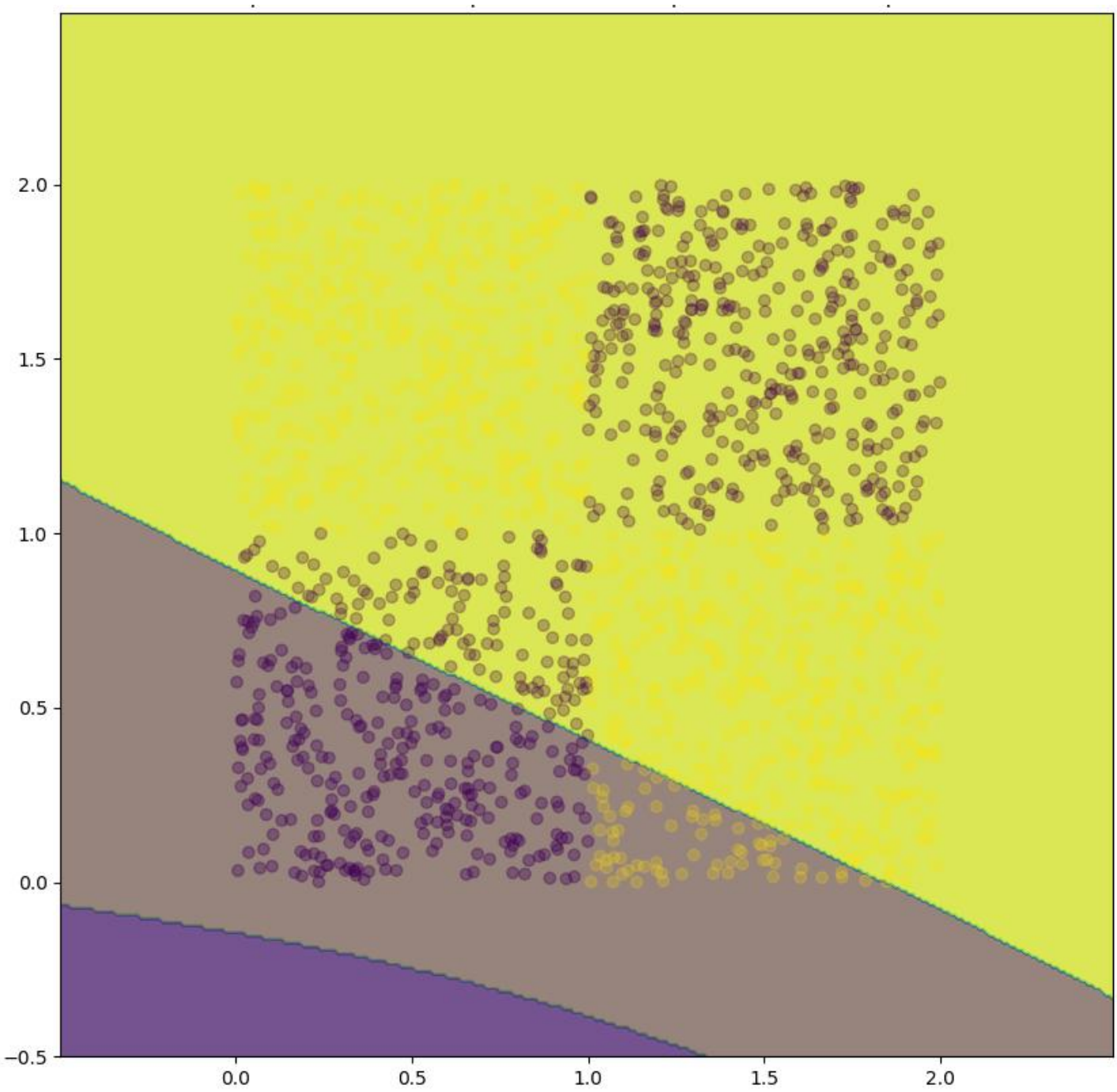
2400 x 0 Output: 0.10 0.09 0.10 – Training/Desired Output: 0.10 0.10 0.10
2400 x 1 Output: 0.80 0.11 0.80 – Training/Desired Output: 0.80 0.10 0.80
2400 x 2 Output: 0.80 0.10 0.80 – Training/Desired Output: 0.80 0.10 0.80
2400 x 3 Output: 0.10 0.79 0.80 – Training/Desired Output: 0.10 0.80 0.80

2700 x 0 Output: 0.10 0.09 0.10 – Training/Desired Output: 0.10 0.10 0.10
2700 x 1 Output: 0.80 0.11 0.80 – Training/Desired Output: 0.80 0.10 0.80
2700 x 2 Output: 0.80 0.10 0.80 – Training/Desired Output: 0.80 0.10 0.80
2700 x 3 Output: 0.10 0.80 0.80 – Training/Desired Output: 0.10 0.80 0.80

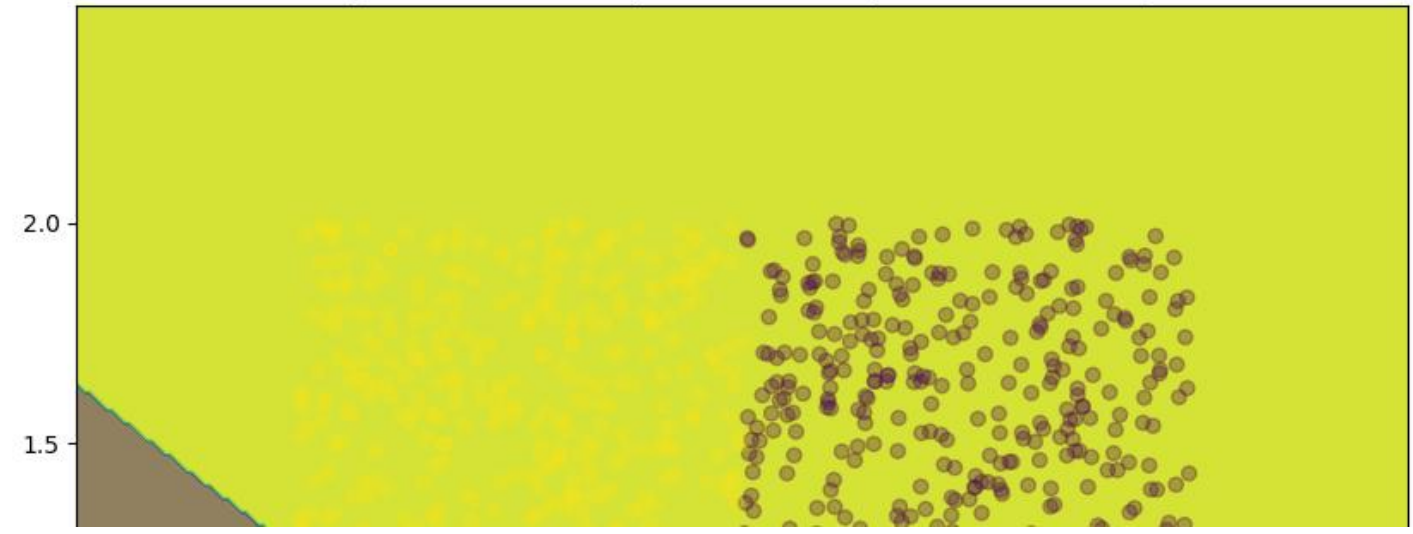
3000 x 0 Output: 0.10 0.10 0.10 – Training/Desired Output: 0.10 0.10 0.10
3000 x 1 Output: 0.80 0.10 0.80 – Training/Desired Output: 0.80 0.10 0.80
3000 x 2 Output: 0.80 0.10 0.80 – Training/Desired Output: 0.80 0.10 0.80
3000 x 3 Output: 0.10 0.80 0.80 – Training/Desired Output: 0.10 0.80 0.80

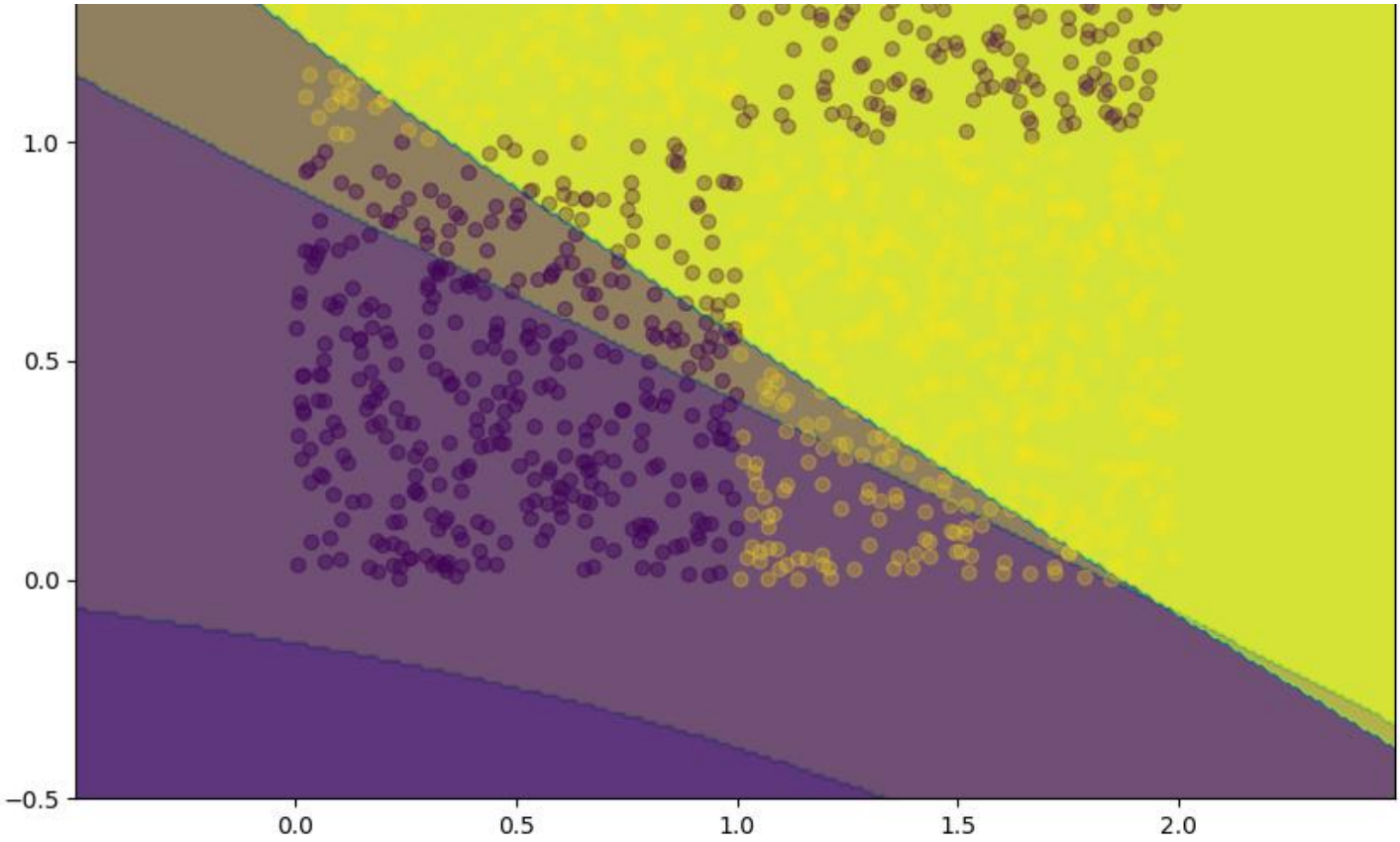
Visualize XOR



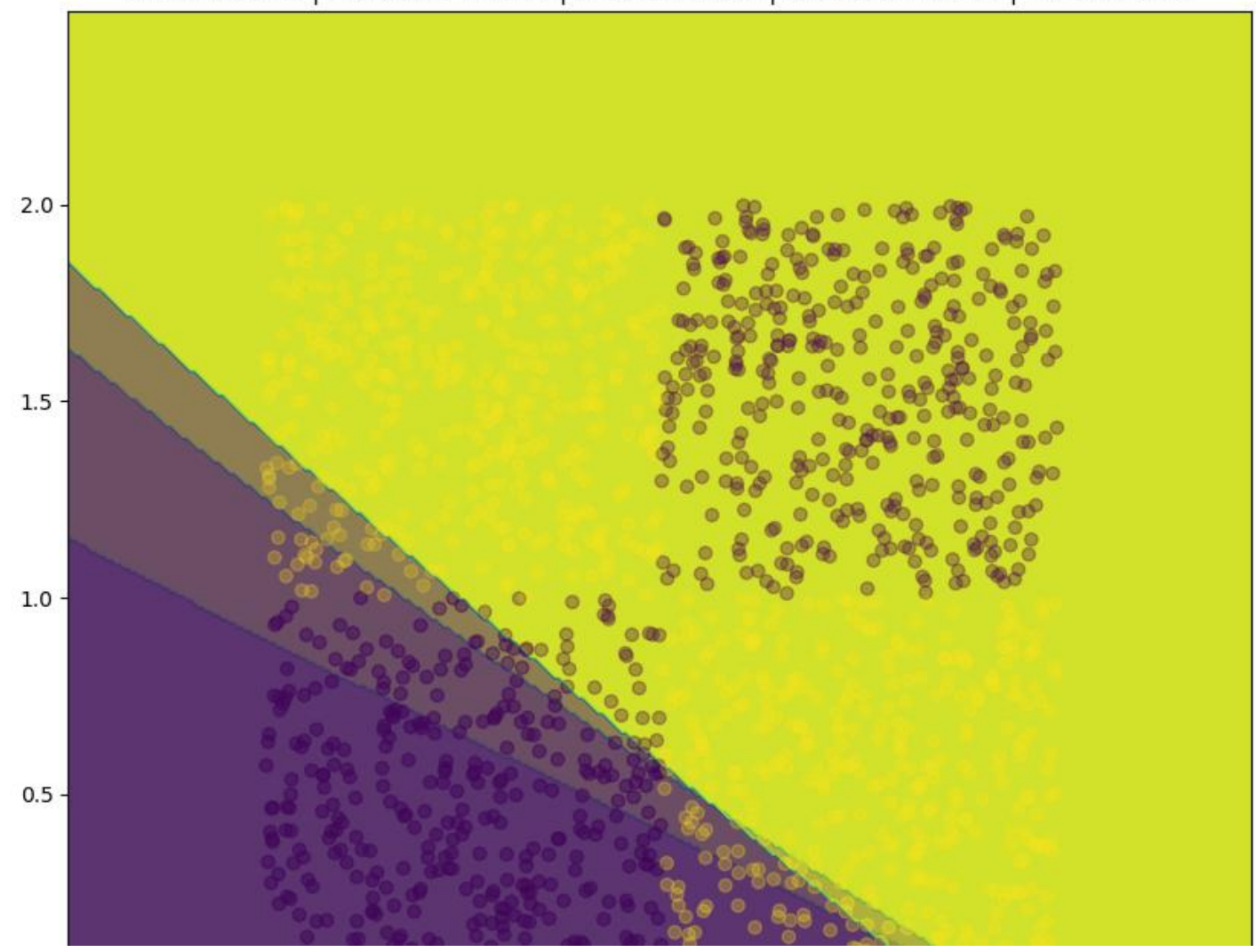


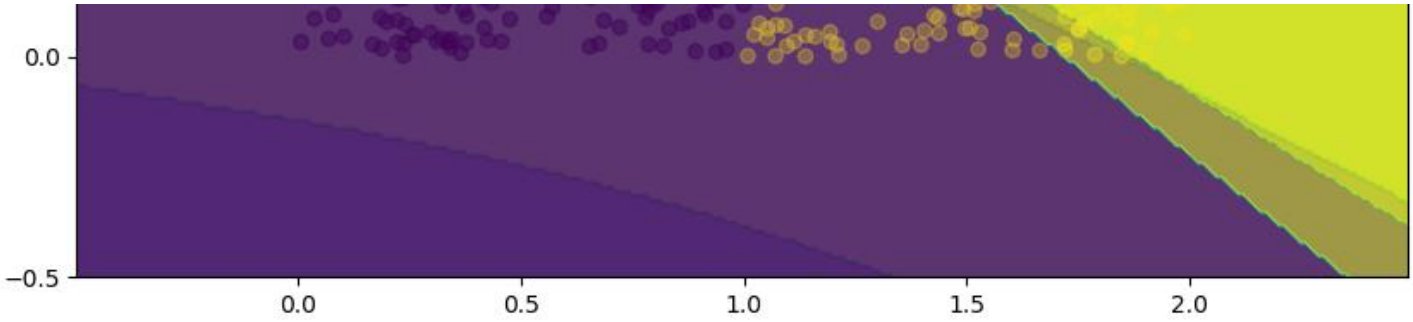
iteration: 100 | train loss: 0.1734 | val loss: 0.195 | train acc: 0.6531 | val acc: 0.61



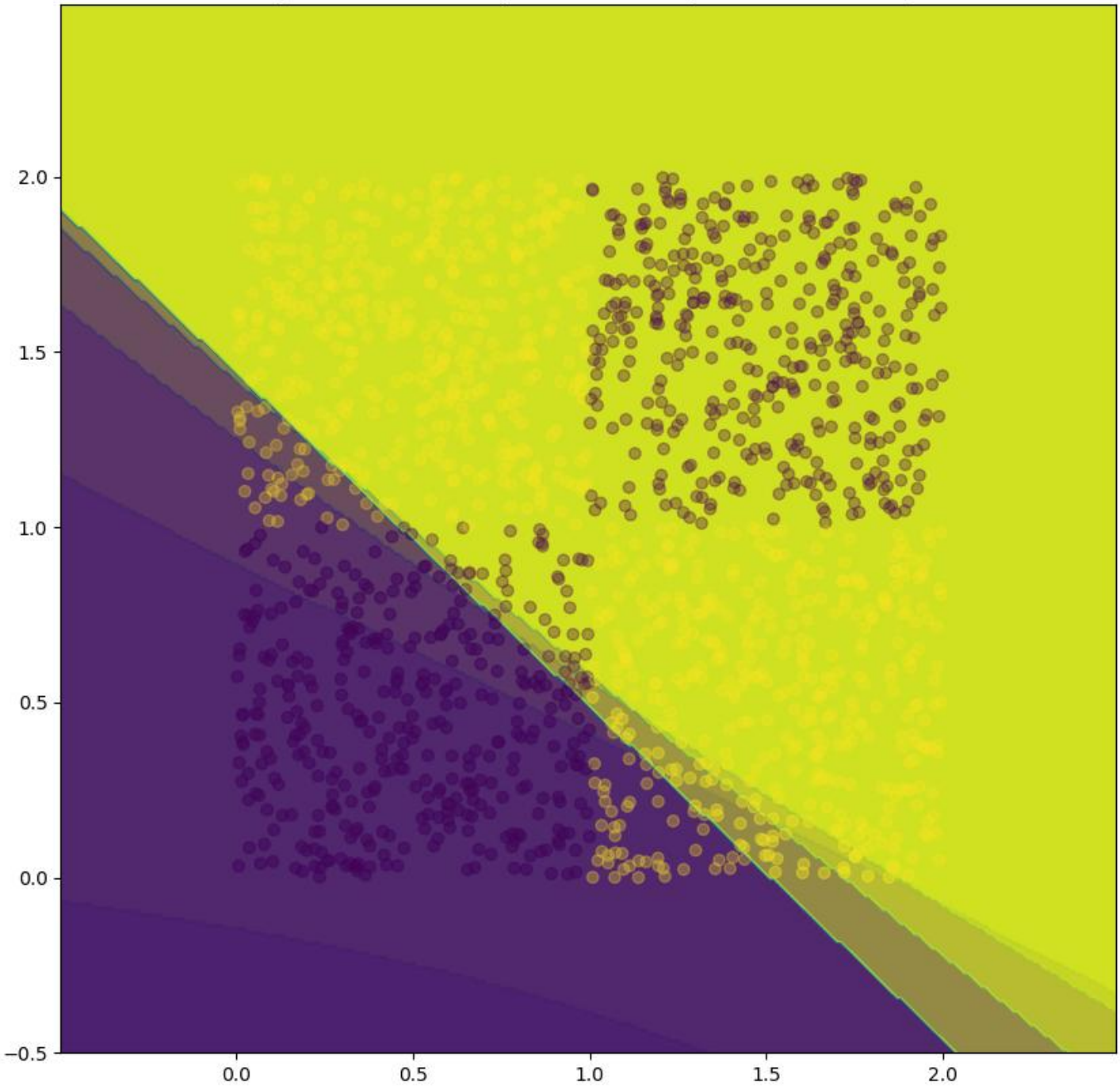


iteration: 150 | train loss: 0.1716 | val loss: 0.195 | train acc: 0.6569 | val acc: 0.61

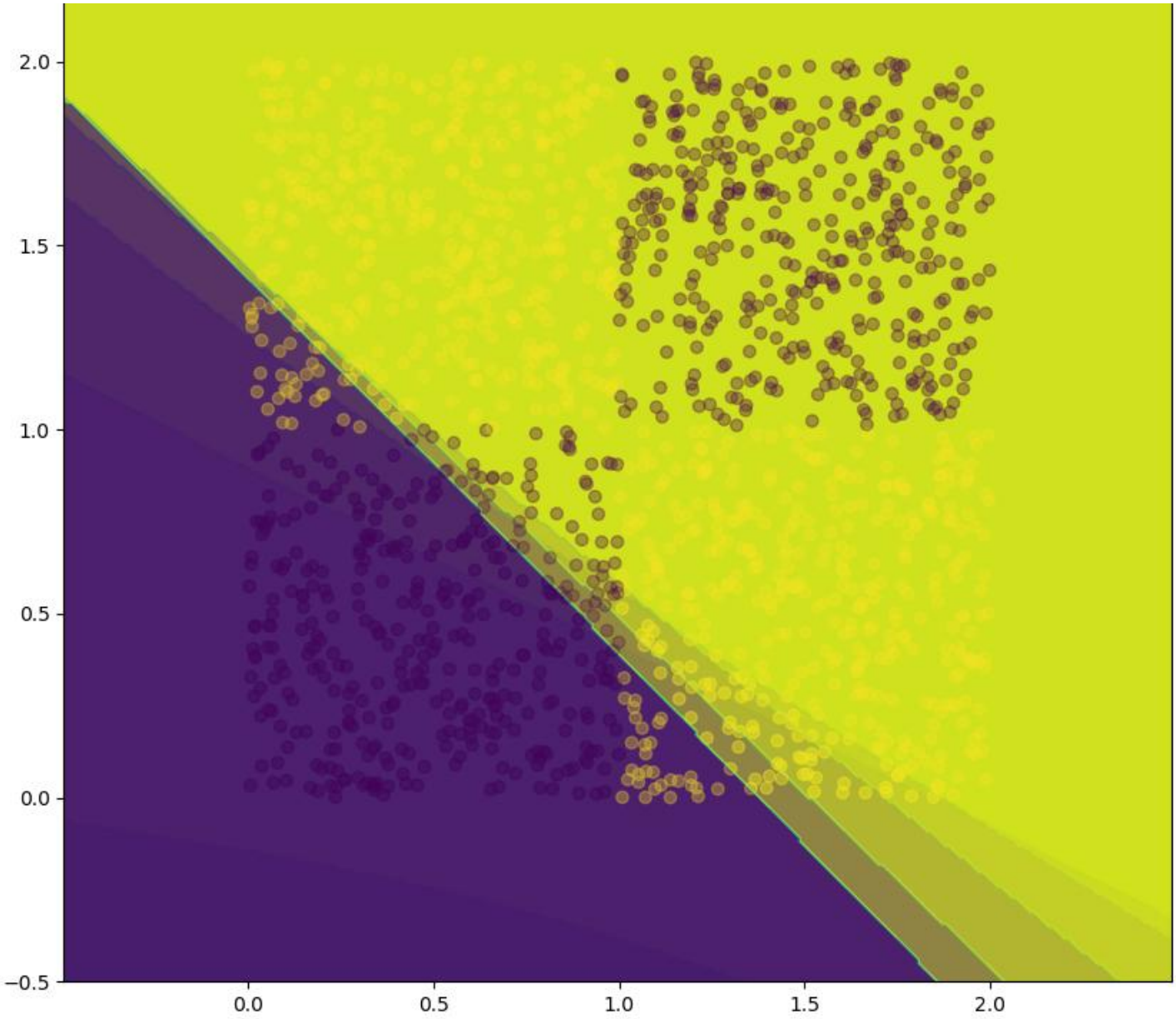




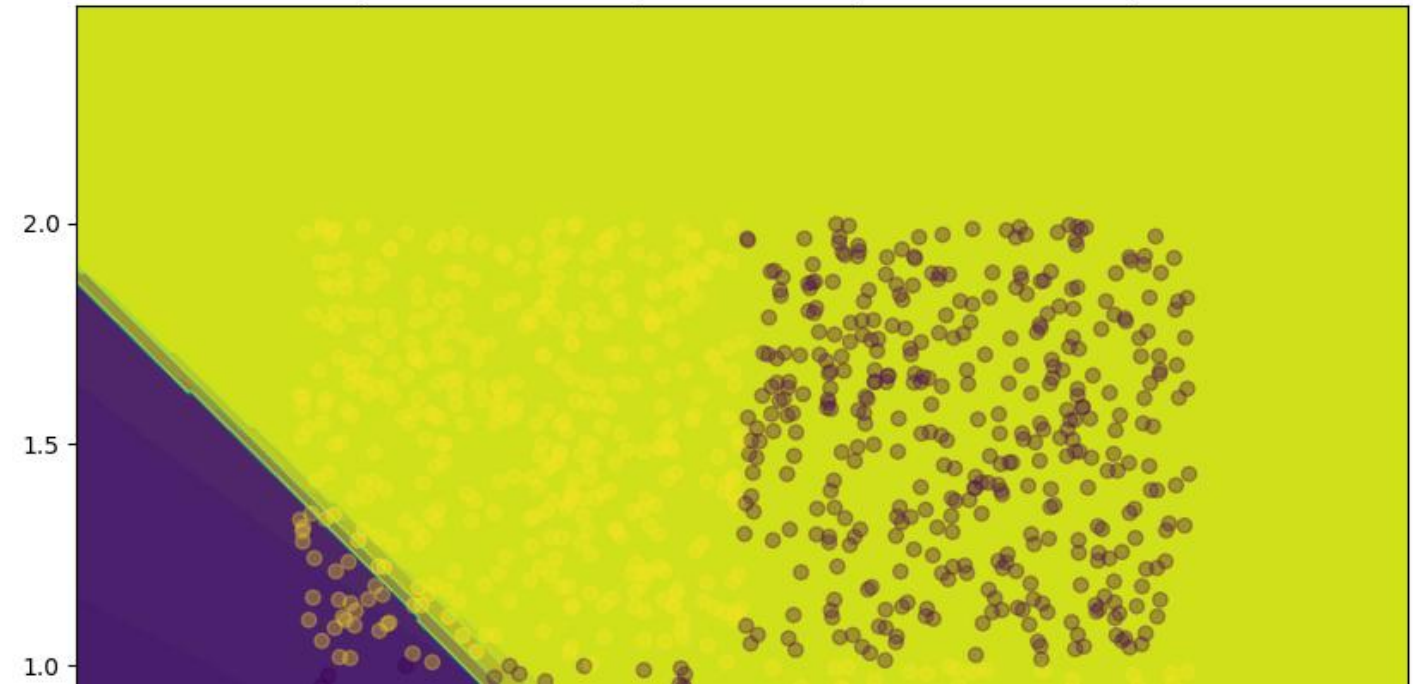
iteration: 200 | train loss: 0.1644 | val loss: 0.185 | train acc: 0.6712 | val acc: 0.63

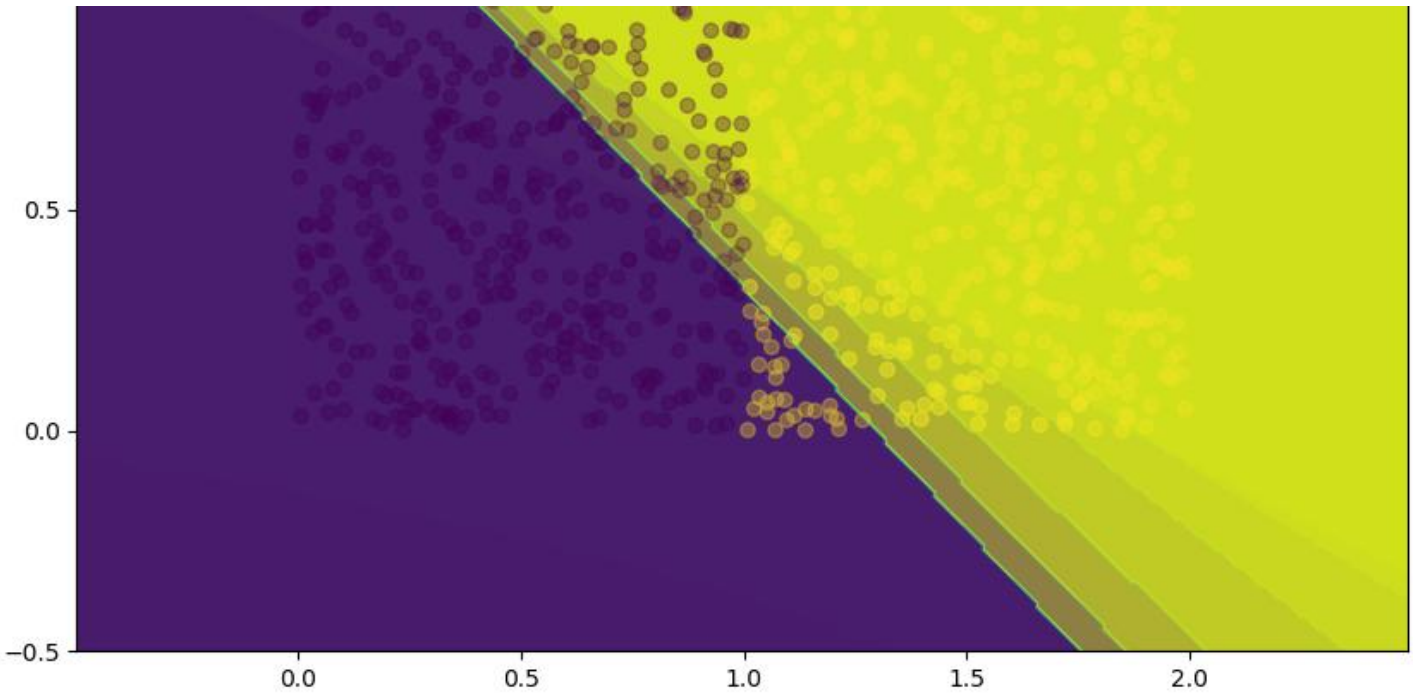


iteration: 250 | train loss: 0.1631 | val loss: 0.1825 | train acc: 0.6738 | val acc: 0.635

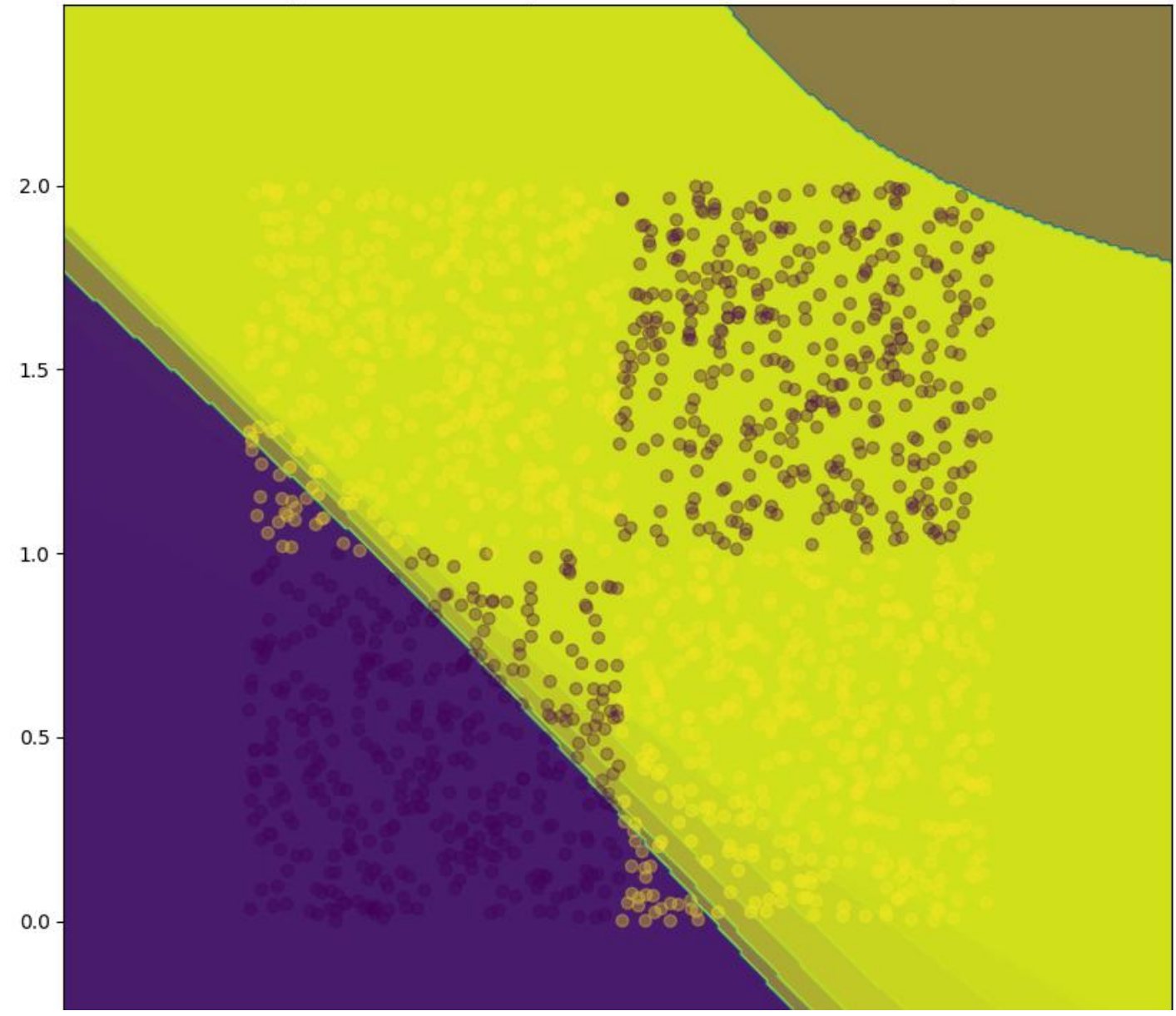


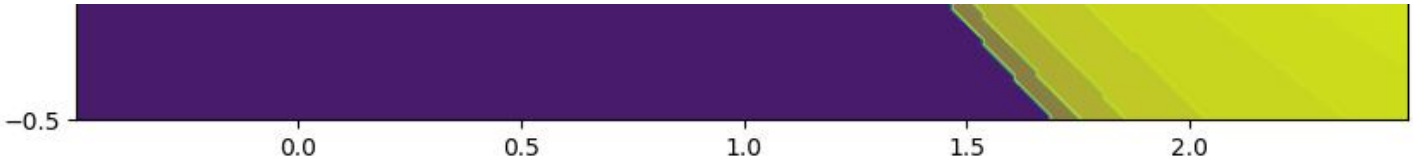
iteration: 300 | train loss: 0.1641 | val loss: 0.18 | train acc: 0.6719 | val acc: 0.64



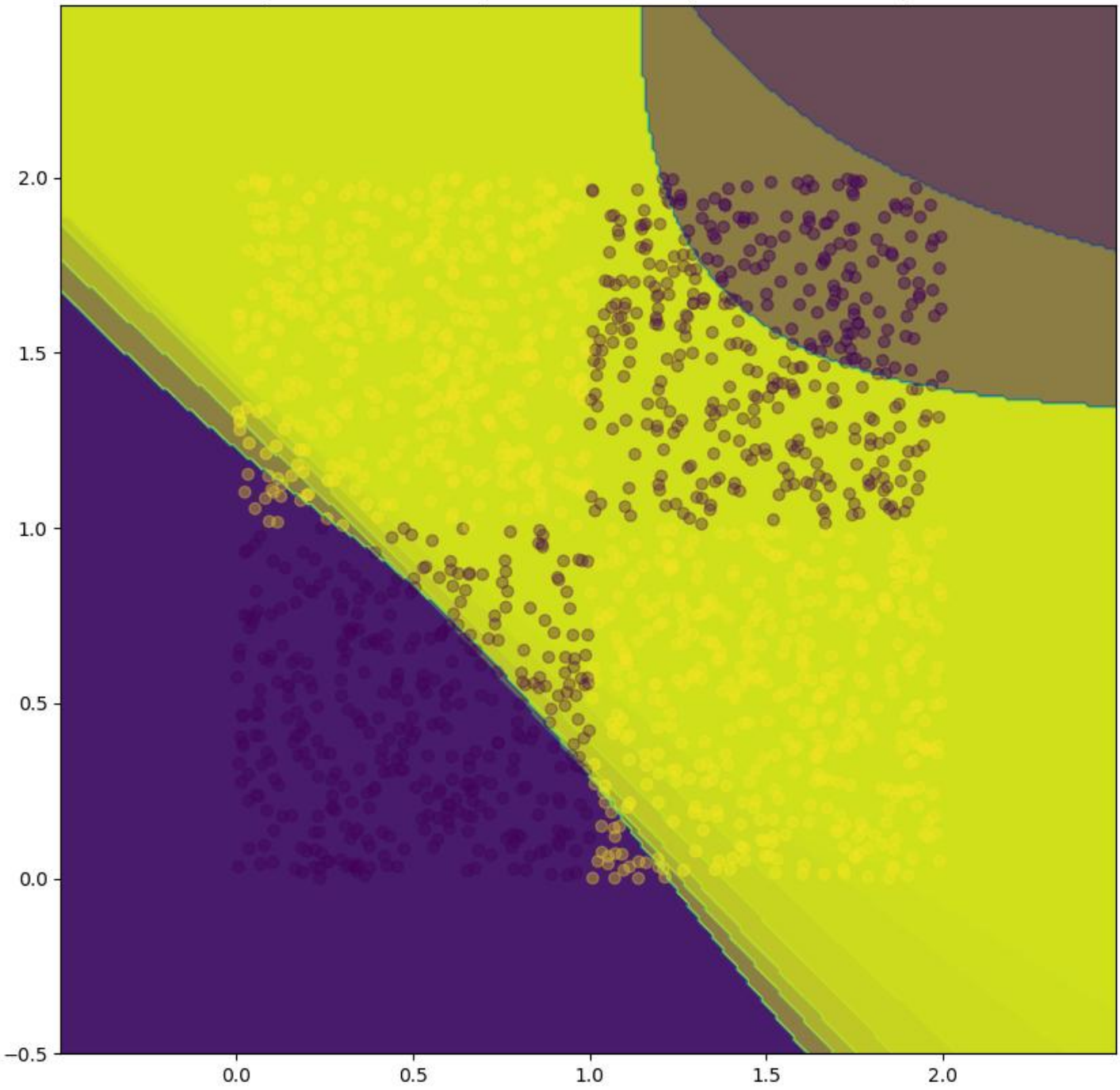


iteration: 350 | train loss: 0.1644 | val loss: 0.175 | train acc: 0.6712 | val acc: 0.65

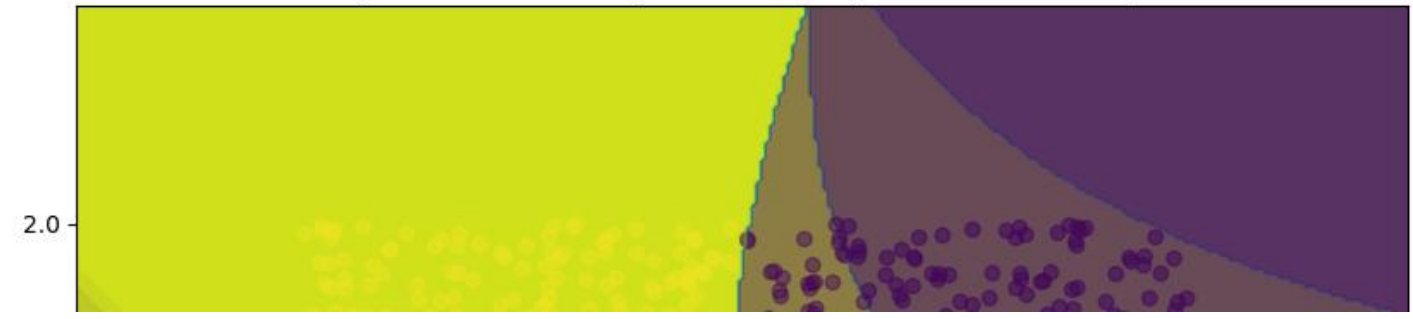


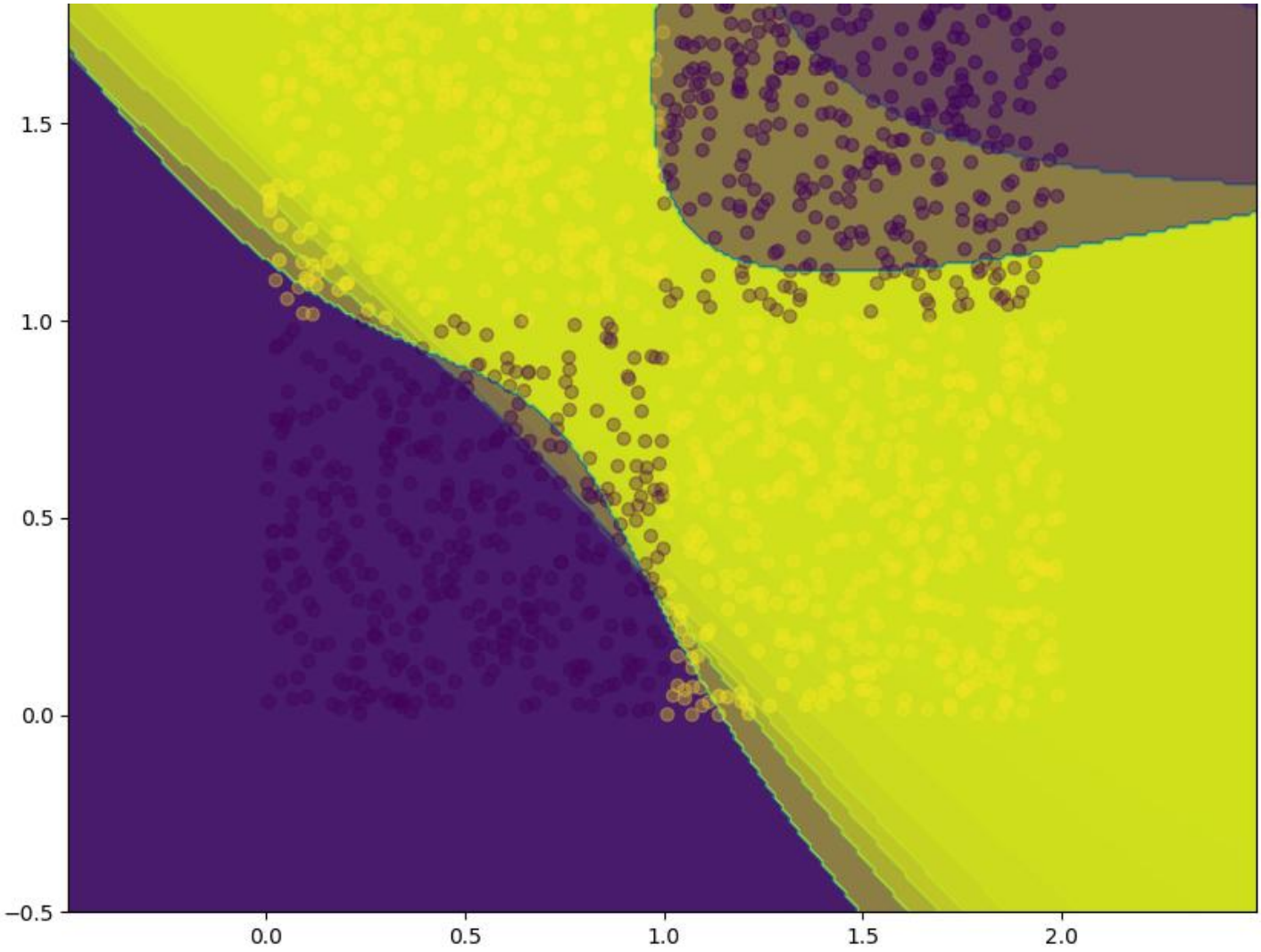


iteration: 400 | train loss: 0.1169 | val loss: 0.1275 | train acc: 0.7662 | val acc: 0.745

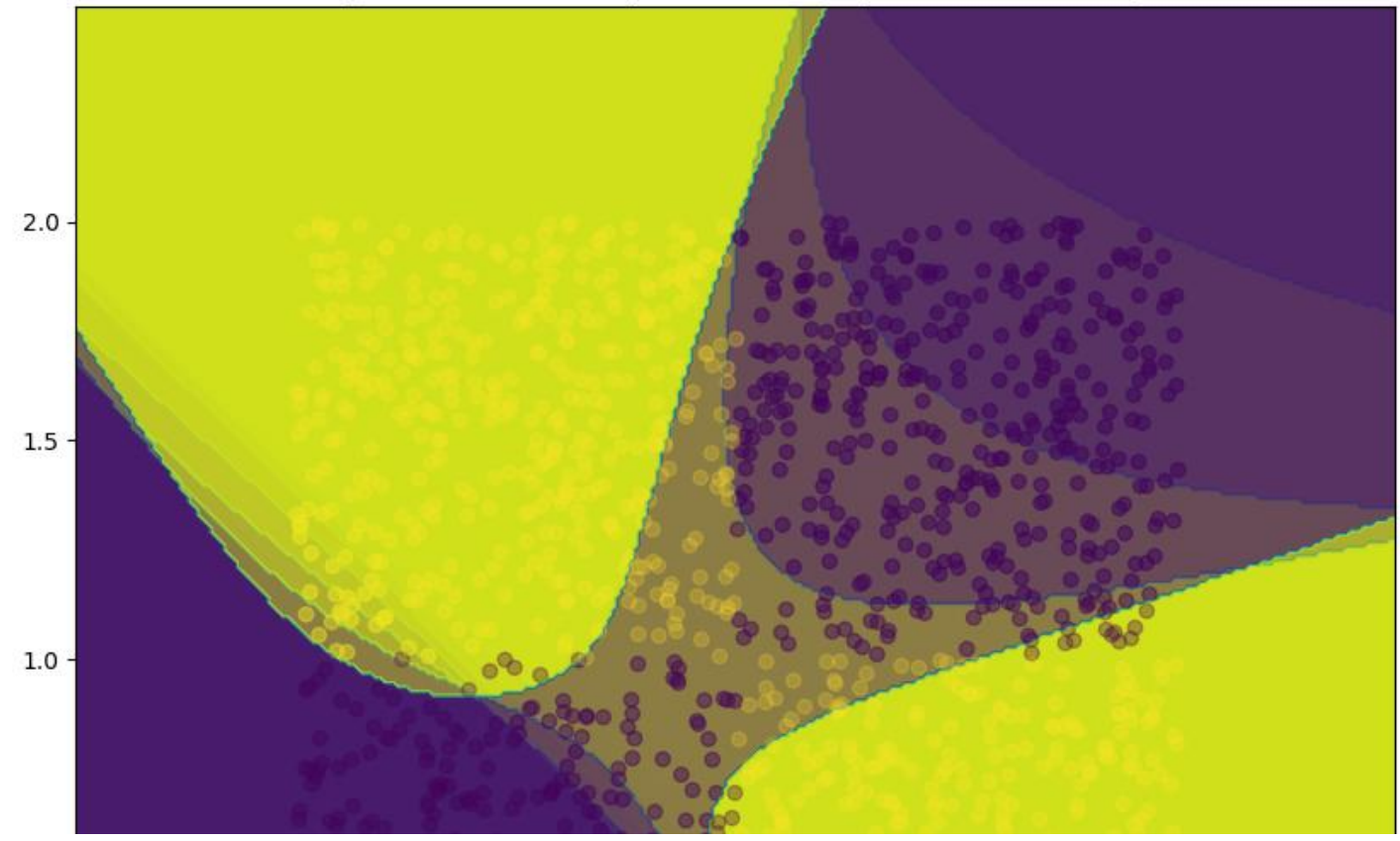


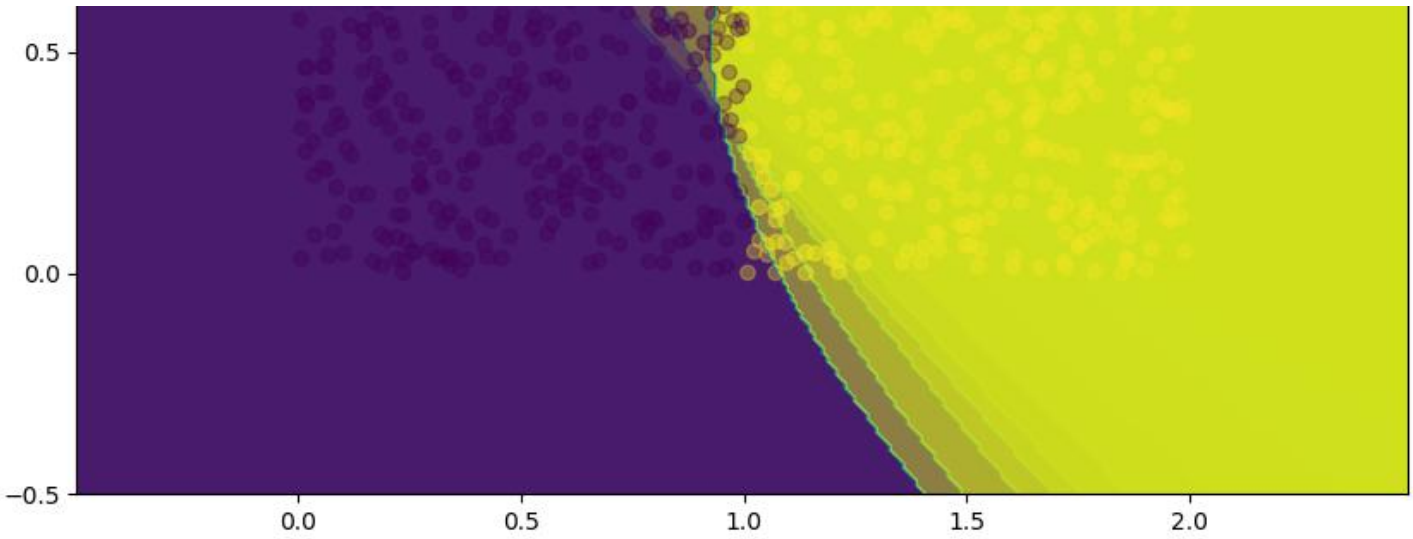
iteration: 450 | train loss: 0.0462 | val loss: 0.06 | train acc: 0.9075 | val acc: 0.88



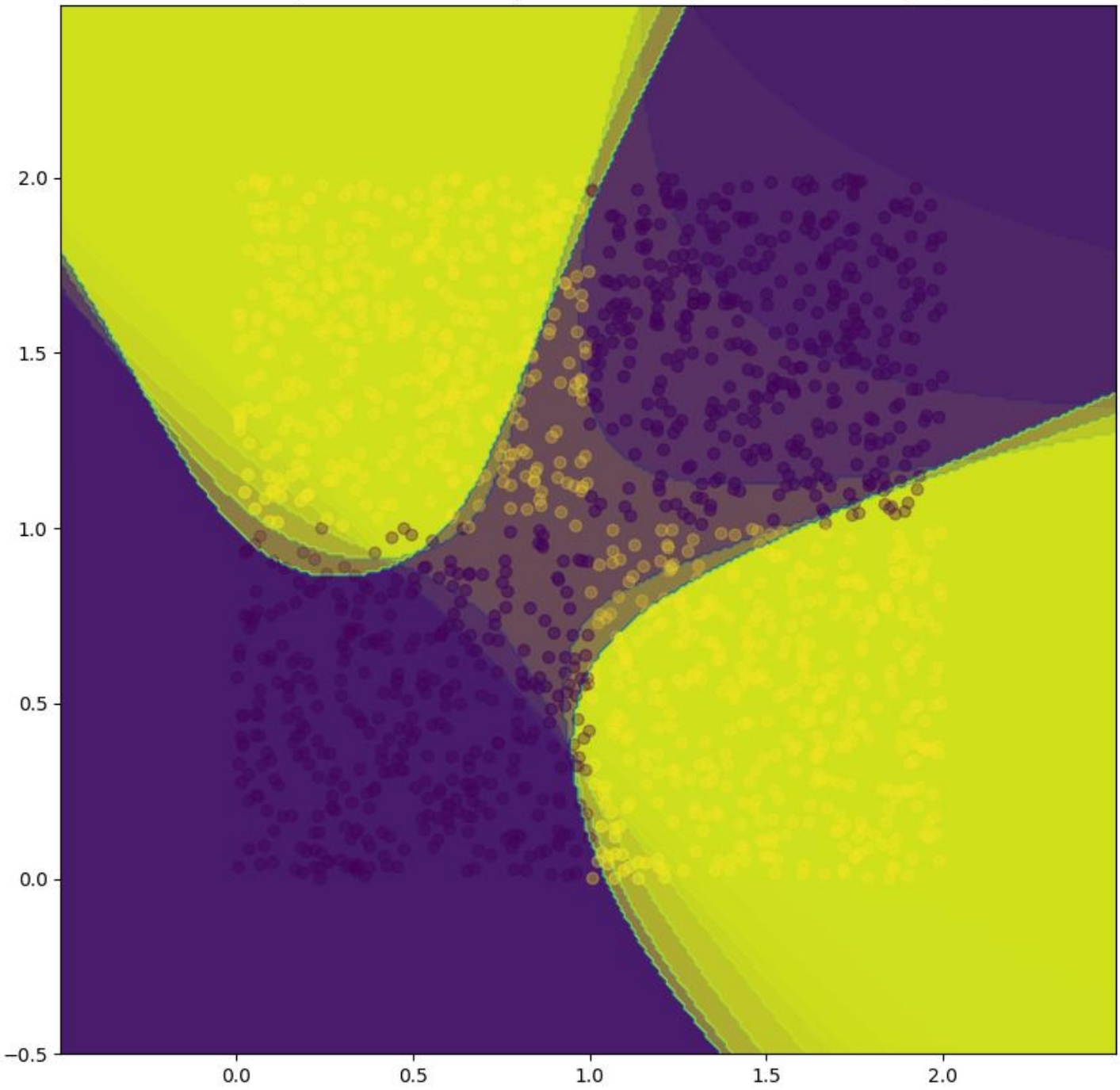


iteration: 500 | train loss: 0.0391 | val loss: 0.045 | train acc: 0.9219 | val acc: 0.91





iteration: 550 | train loss: 0.0434 | val loss: 0.05 | train acc: 0.9131 | val acc: 0.9



XOR Benchmark Classification Training
Its not always so clear as we do reasoning.

If you assume that:
0.1 = False.
0.8 = True.
(0.1+0.8) = 0.45 = Threshold.
y < 0.45 = False.
y > 0.45 = True.

With above assumptions, you'll find that your "others very different" results are precise results in boolean terms.

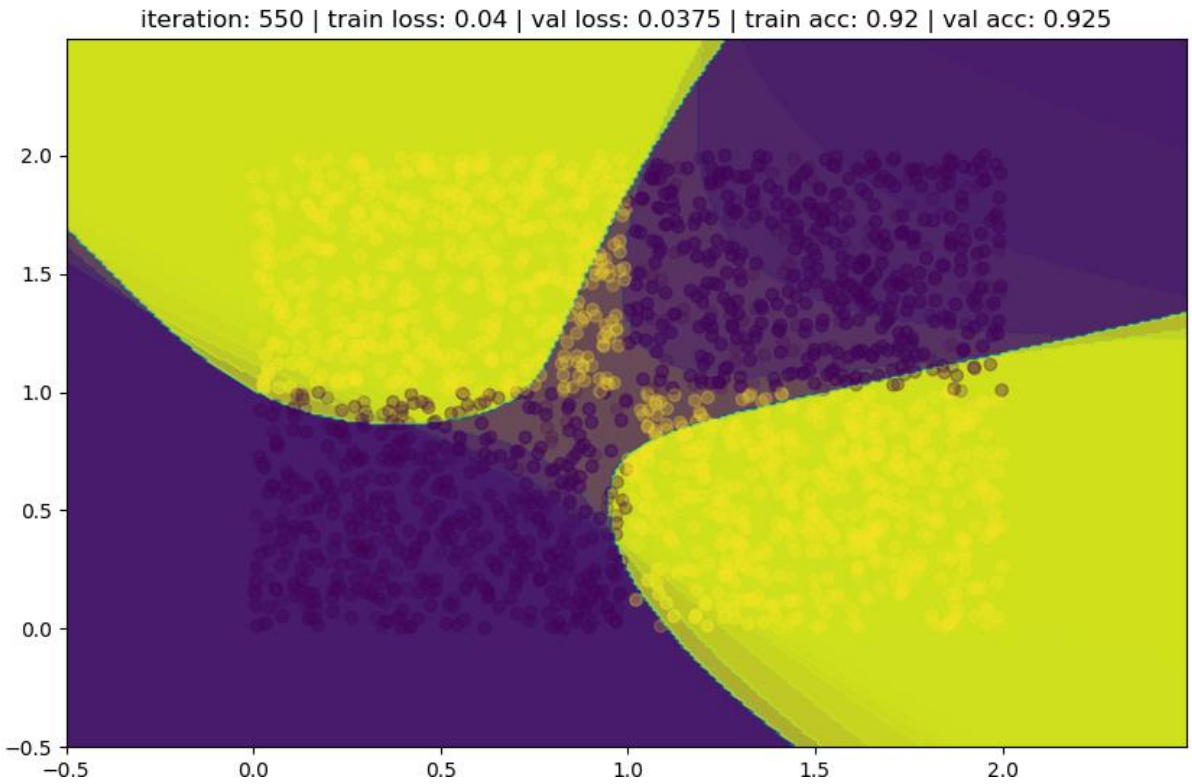
This API implements Stochastic Gradient Descent. As the name implies, it's not deterministic. Most probably, if you increment the number of epochs, results will look more stable.

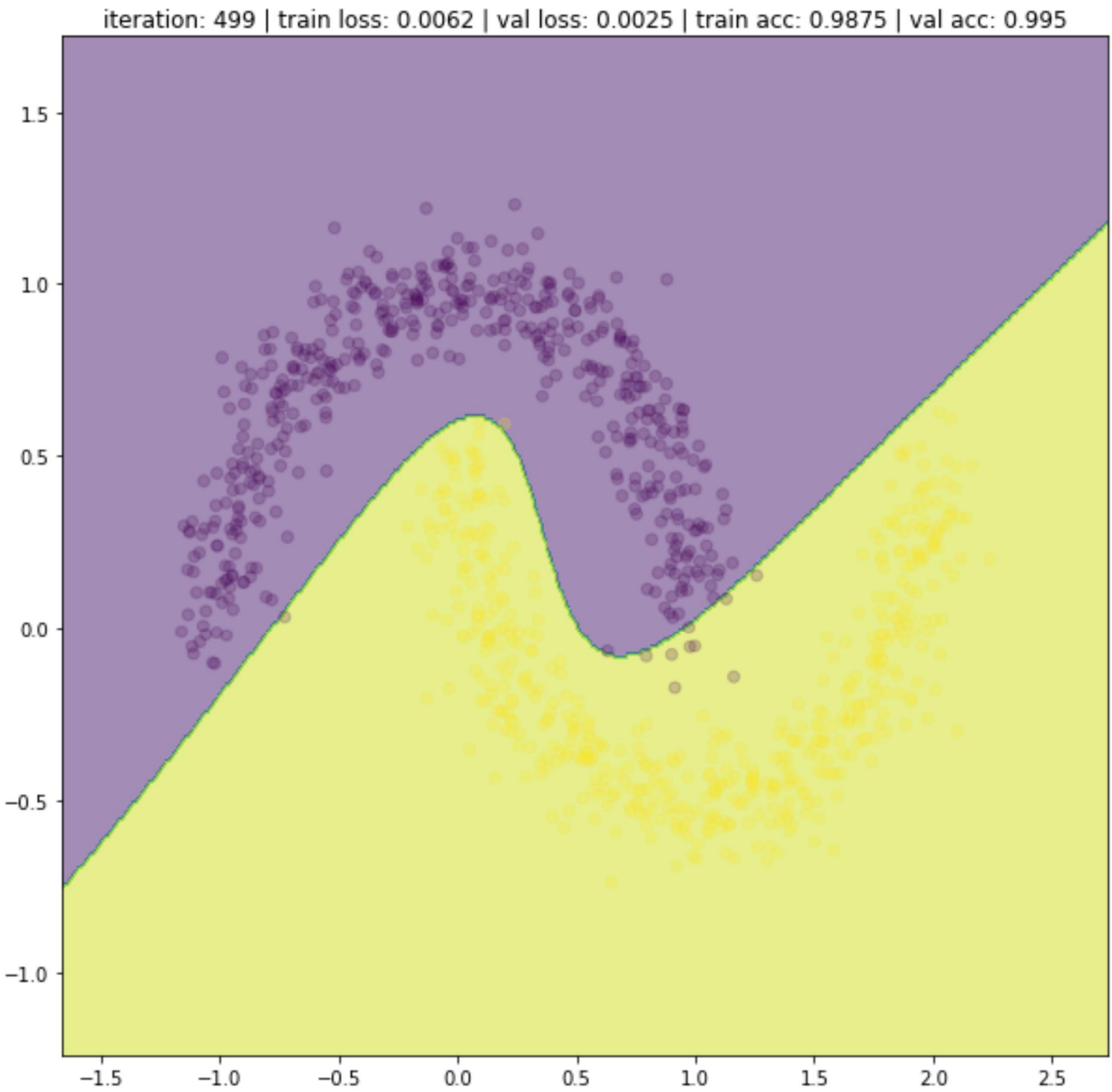
3000 x 0 Output: 0.11 0.00 0.11 – Training/Desired Output: 0.10 0.10 0.10
3000 x 1 Output: 0.79 0.10 0.79 – Training/Desired Output: 0.80 0.10 0.80
3000 x 2 Output: 0.79 0.10 0.79 – Training/Desired Output: 0.80 0.10 0.80
3000 x 3 Output: 0.10 0.80 0.81 – Training/Desired Output: 0.10 0.80 0.80

Layer 0 Max Error: 0 Min Error: 0 Max ErrorD: 0 Min ErrorD: 0 TNNetInput 2,1,1
debug errors else
Layer 1 Max Error: 0.000858666782733053 Min Error: -0.00092624151147902 Max ErrorD: 0 Min ErrorD: 0 TNNetFullConnectReLU 3,1,1
Parent:0
Layer 2 Max Error: 0.0012739896774292 Min Error: -0.00215935707092285 Max ErrorD: 0 Min ErrorD: 0 TNNetFullConnectReLU 3,1,1
Parent:1
Press ENTER to exit.

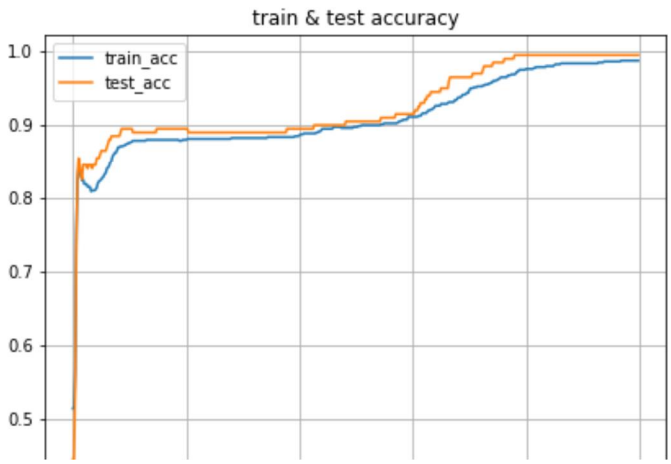
3000 x 0 Output: 0.37 0.16 0.56 – Training/Desired Output: 0.10 0.10 0.10
3000 x 1 Output: 0.77 0.10 0.82 – Training/Desired Output: 0.80 0.10 0.80
3000 x 2 Output: 0.59 0.00 0.55 – Training/Desired Output: 0.80 0.10 0.80
3000 x 3 Output: 0.11 0.72 0.57 – Training/Desired Output: 0.10 0.80 0.80

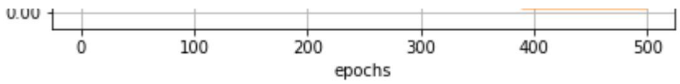
We can also visualize the decision boundary :

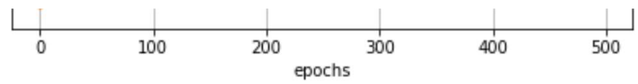


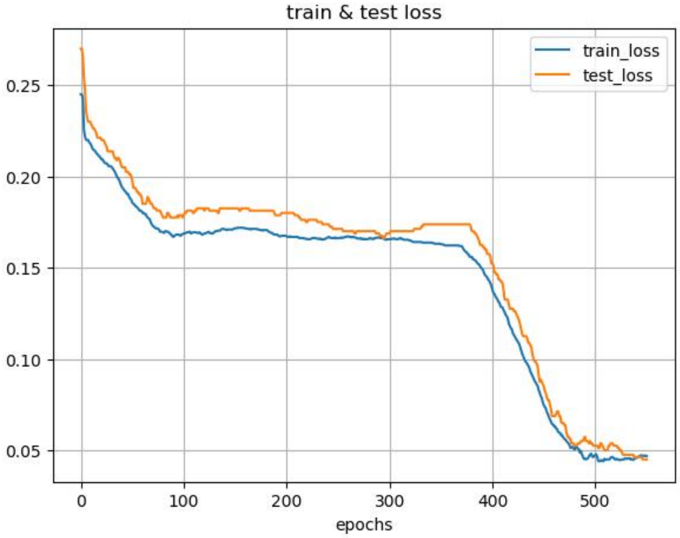

iteration: 499 | train loss: 0.0062 | val loss: 0.0025 | train acc: 0.9875 | val acc: 0.995
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5

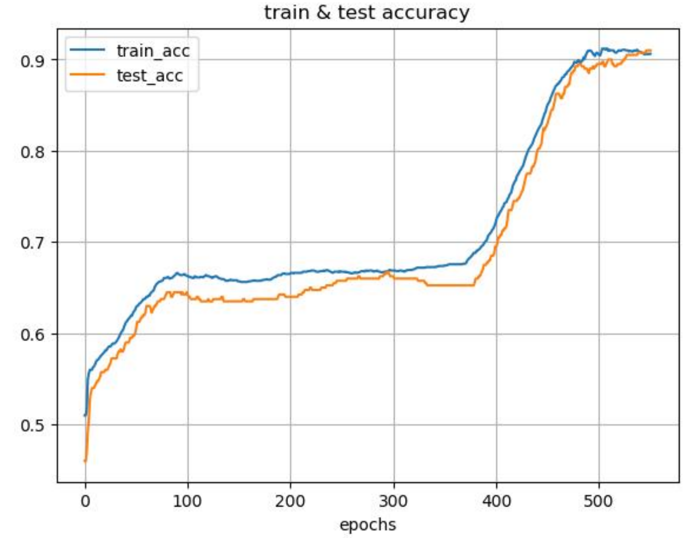

train & test loss
train_loss
test_loss
0.25
0.20
0.15
0.10
0.05

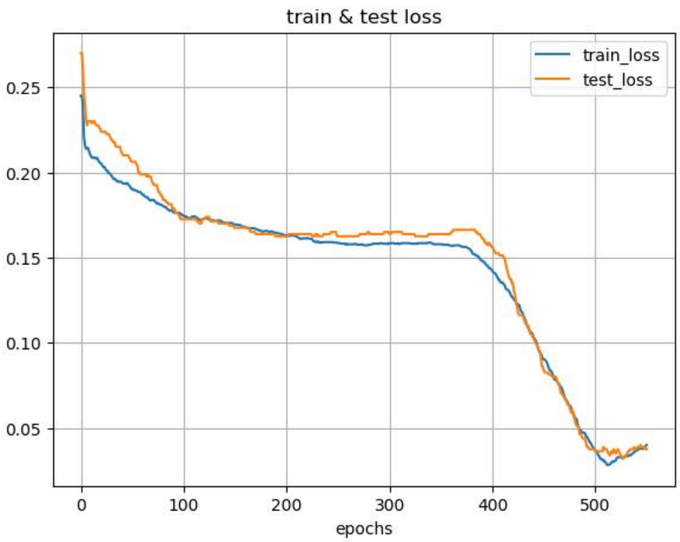

train & test accuracy
train_acc
test_acc
1.0
0.9
0.8
0.7
0.6
0.5

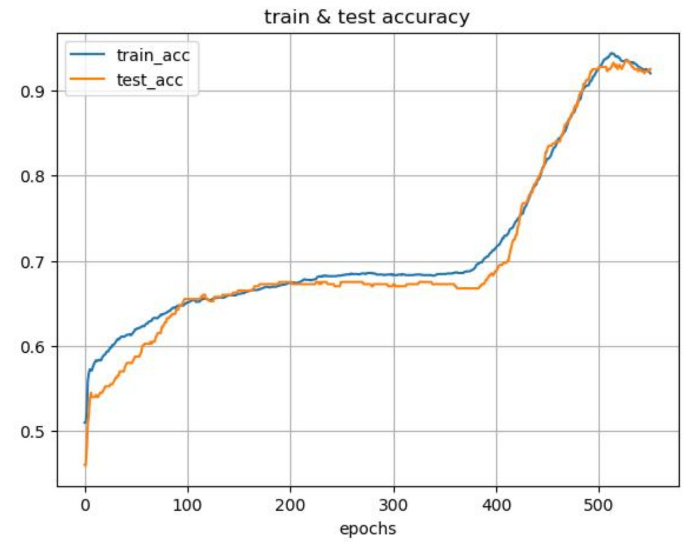


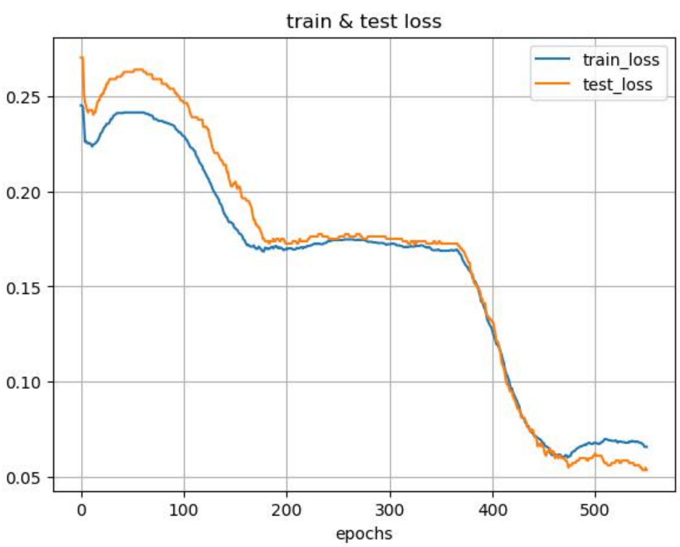


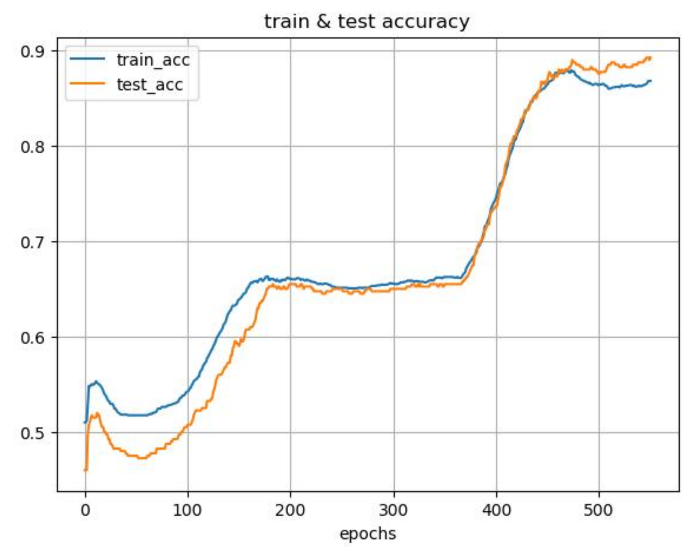










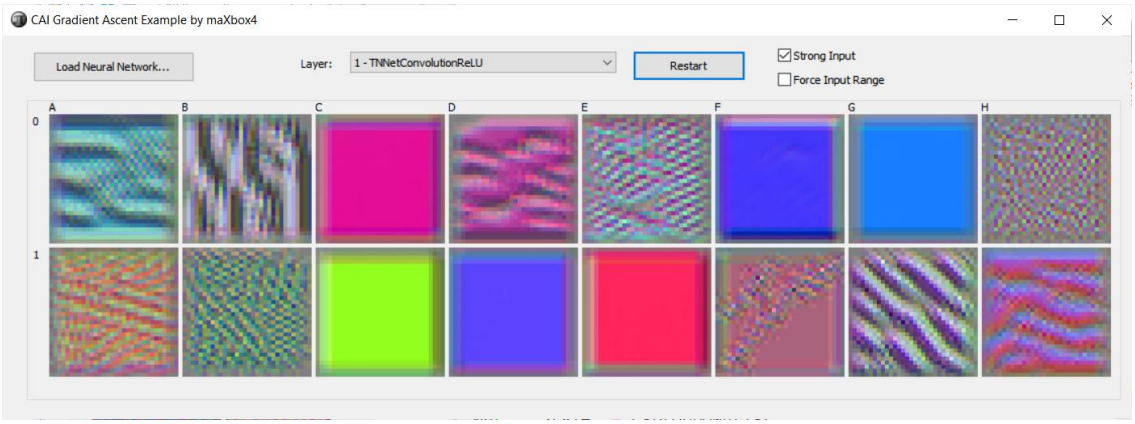


Visualise Neurons

It's usually very hard to understand neuron by neuron how a neural network dedicated to image classification internally works. One technique used to help with the understanding about what individual neurons represent is called Gradient Ascent.

In this technique, an arbitrary neuron is required to activate and then the same backpropagation method used for learning is applied to an input image producing an image that this neuron expects to see.

To be able to run this example, you'll need to load an already trained neural network file (for example **https://sourceforge.net/projects/maxbox/files/Examples/CAI/1076_SimpleImageClassifierEKON25_5000.nn/download (https://sourceforge.net/projects/maxbox/files/Examples/CAI/1076_SimpleImageClassifierEKON25_5000.nn/download)**) and then select the layer you intend to visualize.



Convolution ReLU

Deeper convolutional layers tend to produce more complex patterns. Above image was produced from a very first convolutional layer. The following image was produced from a third convolutional layer. Notice that patterns are more complex.

This is the API method used for an arbitrary neuron backpropagation (Gradient Ascent):

```
procedure TNNet.BackpropagateFromLayerAndNeuron(LayerIdx, NeuronIdx: integer; Error: TNeuralFloat);
```

Errors on the input image aren't enabled by default. In this example, errors regarding the input image are enabled with this:

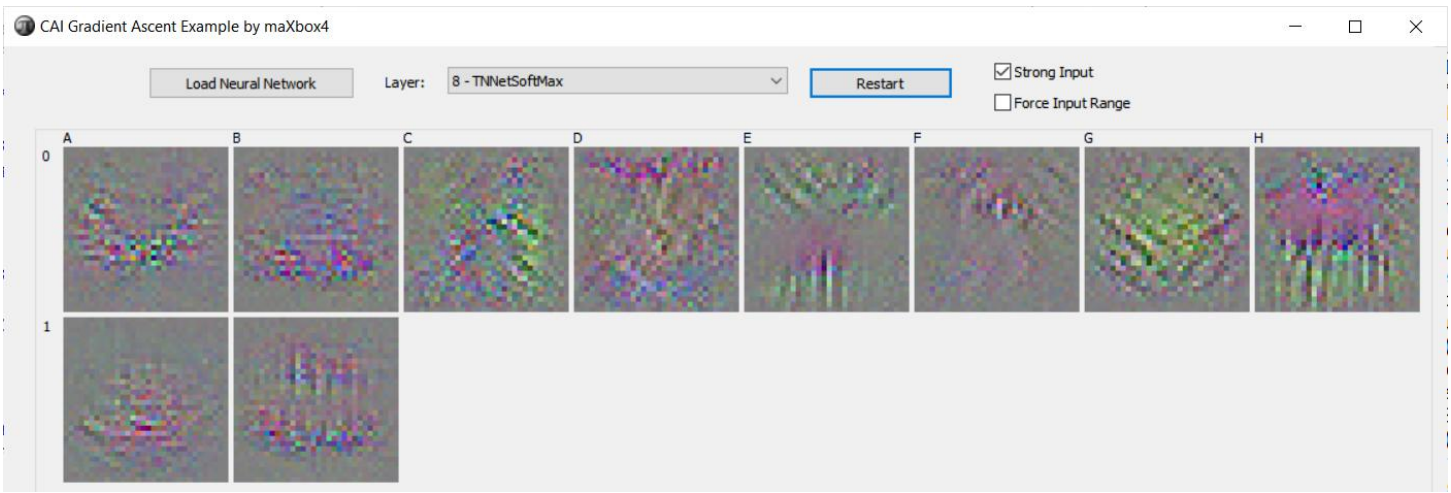
```
TNNetInput(FNN.Layers[0]).EnableErrorCollection();
```

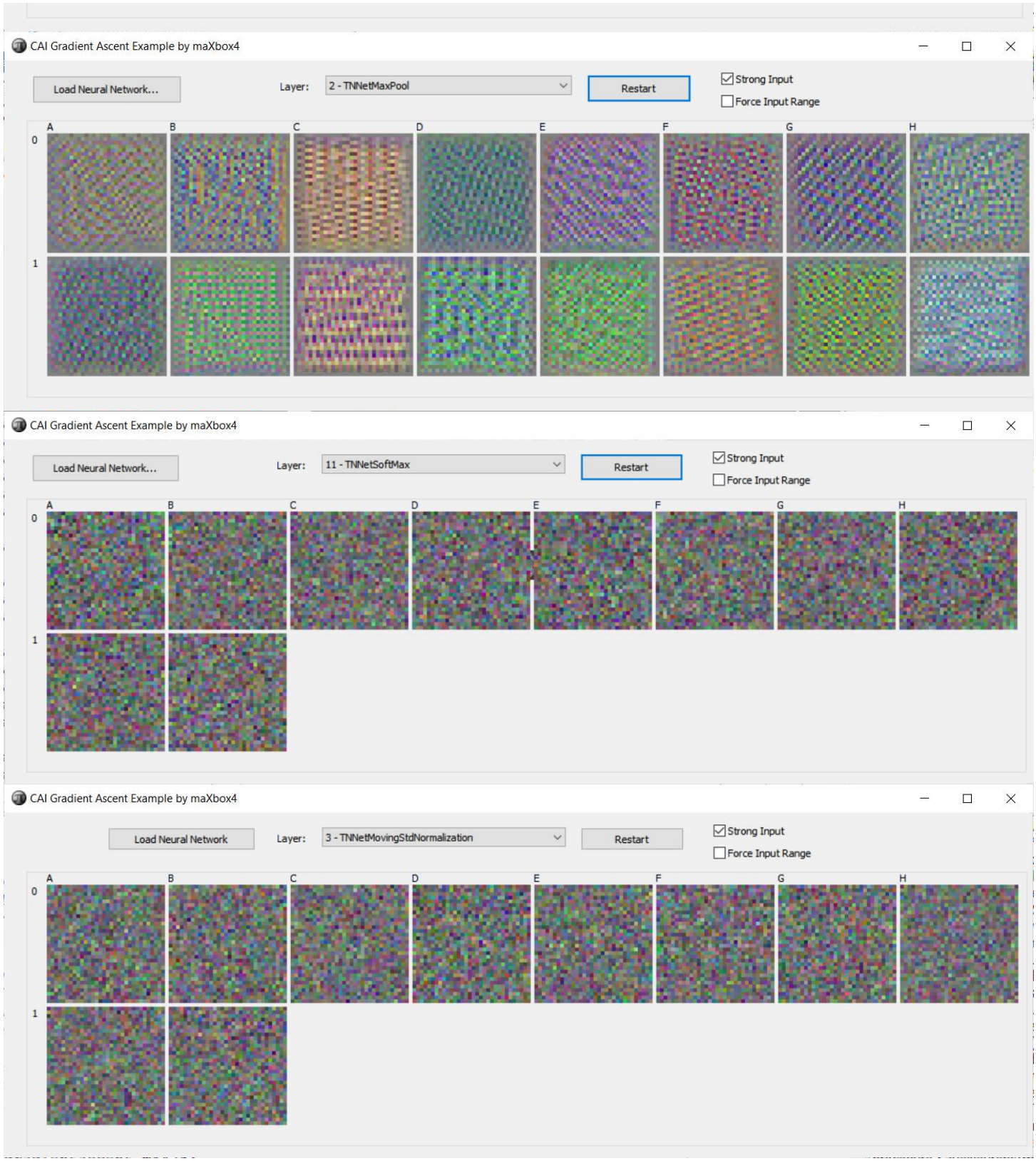
Then, errors are added to the input with this:

```
vInput.MulAdd(-1, FNN.Layers[0].OutputError);
FNN.ClearDeltas();
FNN.ClearInertia();
```

You can find more about Gradient Ascent at:

- **Lecture 12: Visualizing and Understanding – CS231n – Stanford (http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture12.pdf)**
- **Understanding Neural Networks Through Deep Visualization (http://yosinski.com/deepvis)**

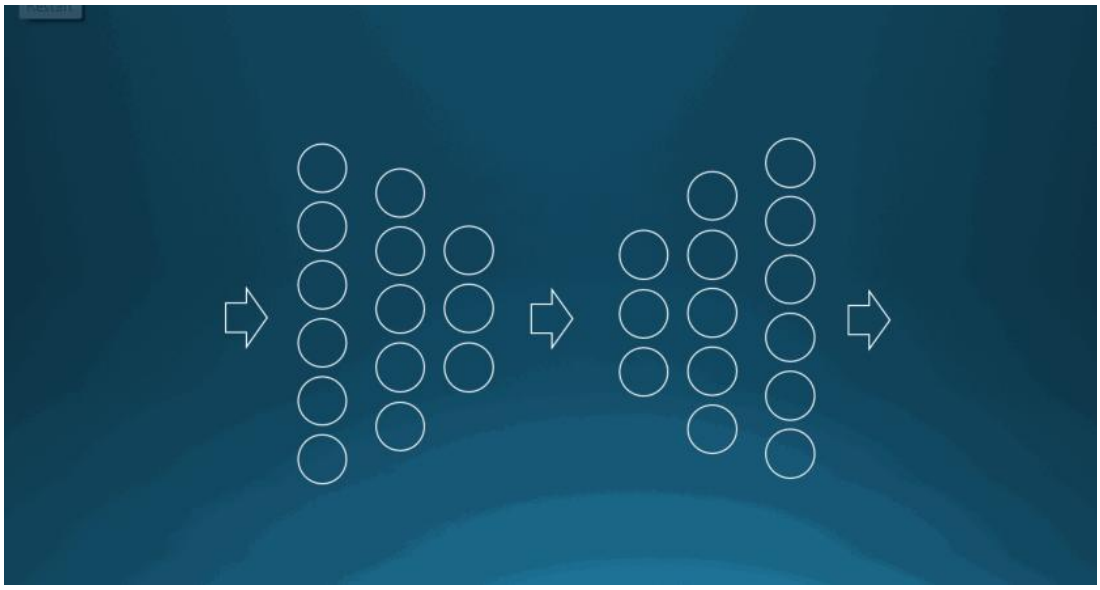




As we know about 50 epochs deliver useful results with the CIFAR10 image set:

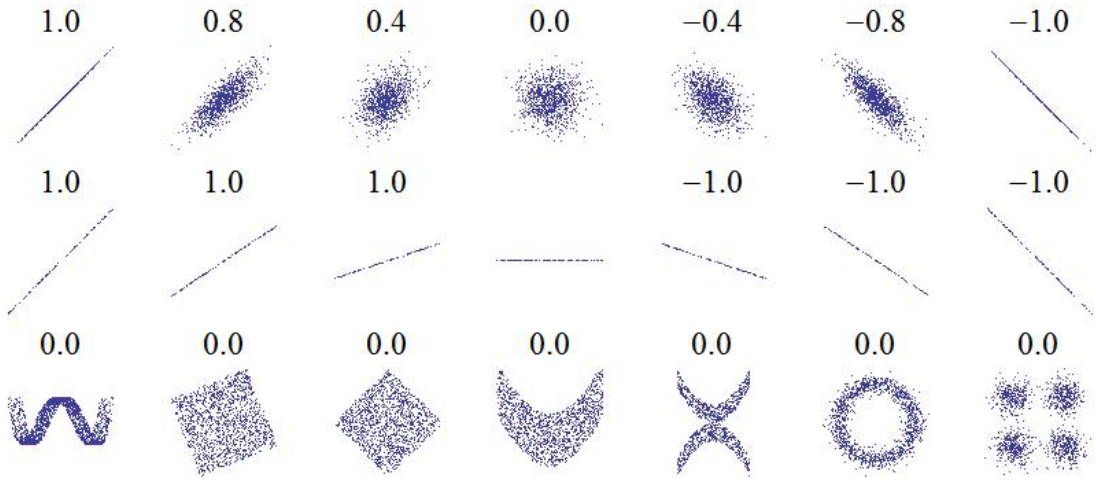
Starting Testing.
Epochs: 50 Examples seen:2400000 Test Accuracy: 0.8481 Test Error: 0.4238 Test Loss: 0.4620 Total time: 188.44min
Epoch time: 3.2887 minutes. 50 epochs: 2.7406 hours.
Epochs: 50. Working time: 3.14 hours.
Loading SimpleImageClassifier-60.nn for final test.
Starting Testing.
Epochs: 50 Examples seen:2400000 Test Accuracy: 0.8481 Test Error: 0.4238 Test Loss: 0.4620 Total time: 189.71min
Loading best performing results SimpleImageClassifier-60.nn.

Finished.



Backpropagation Demo

Last point concerning correlations: Correlations are useful because they can indicate a predictive relationship that can be exploited in practice. For example, an electrical utility may produce less power on a mild day based on the correlation between electricity demand and weather. In this example, there is a **causal relationship (https://en.wikipedia.org/wiki/Causality)**, because **extreme weather (https://en.wikipedia.org/wiki/Extreme_weather)** causes people to use more electricity for heating or cooling. However, in general, the presence of a correlation is not sufficient to infer the presence of a causal relationship (i.e., **correlation does not imply causation (https://en.wikipedia.org/wiki/Correlation_does_not_imply_causation)**).



https://en.wikipedia.org/wiki/Correlation (https://en.wikipedia.org/wiki/Correlation)

You can find the scripts at_:

https://sourceforge.net/projects/maxbox/files/Examples/CAI/ (https://sourceforge.net/projects/maxbox/files/Examples/CAI/)

Formally, random variables are *dependent* if they do not satisfy a mathematical property of **probabilistic independence (https://en.wikipedia.org/wiki/Independence_(probability_theory))**. In informal parlance, *correlation* is synonymous with *dependence*.

Posted in **EKON**, **Machine Learning**, **maXbox** **Leave a comment**