§3.3 MLBD MRes practical From a single neuron to the multilayer perceptron

Jarvist Moore Frost.

Imperial College London
Email: jarvist.frost@ic.ac.uk
Twitter: @JarvistFrost https://jarvist.github.io

Intended Learning Outcomes §3.3

Demonstrate how multilayer perceptrons are built, by building a network and solving a task within PyTorch.

- ➔ Be able to summarise what we learn in the first session. (Taught from these slides)
- ➔ Reimplement a single neuron binary classifier in PyTorch. (Jupyter)
- ➔ Be aware of the mathematics of backpropagation, the multi-layer learning algorithm. (Jupyter)
- ➔ Use a multilayer perceptron to solve a non-linear task. (Jupyter)
- ➔ Run well defined machine-learning experiments to explore and document the parameter space of model construction and learning, communicating progress with your peers. (Jupyter)

Recommended reading

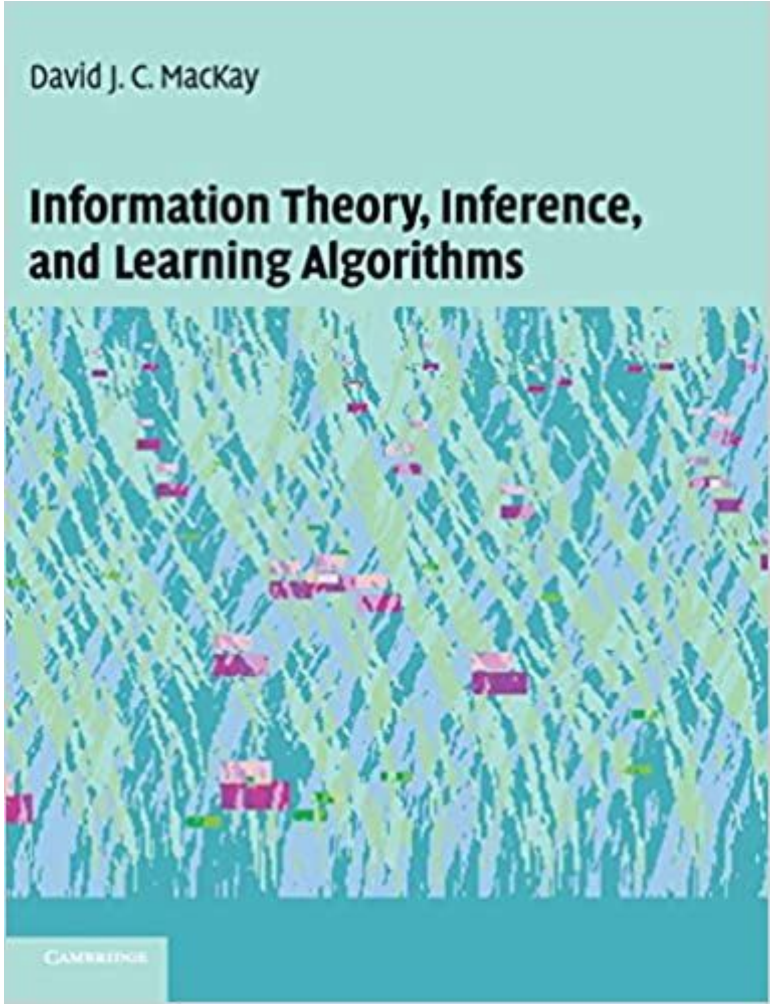
David MacKay, Information Theory, Inference and Learning Algorithms (ITILA), 2003, Chapters 38–42.

Freely available online!

A physicists perspective on ML.

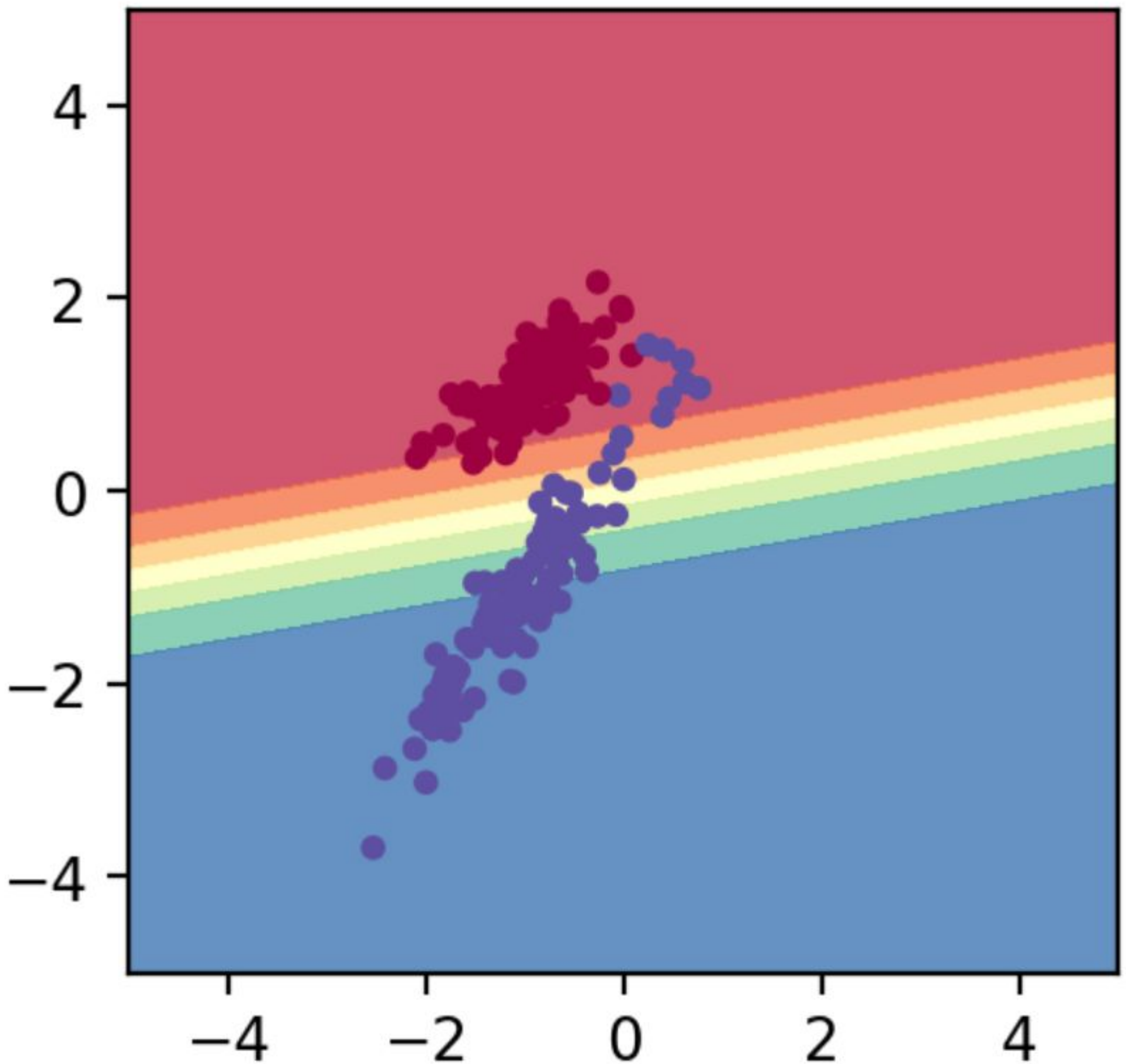
http://www.inference.org.uk/mackay/itila/book.html

These slides and classworks follow some of the structure of Chapter 38 & 39.

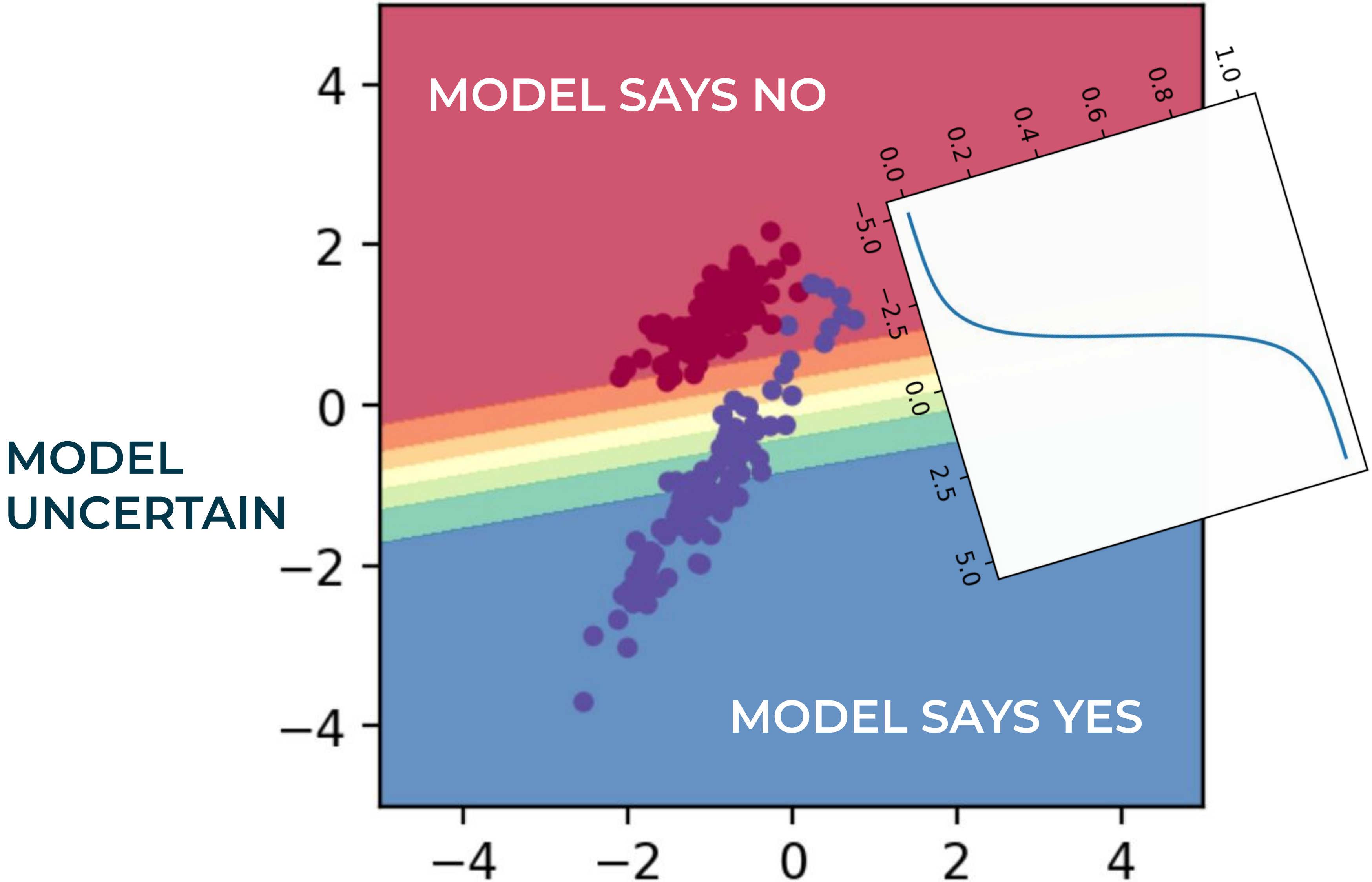


§3.2 Practical

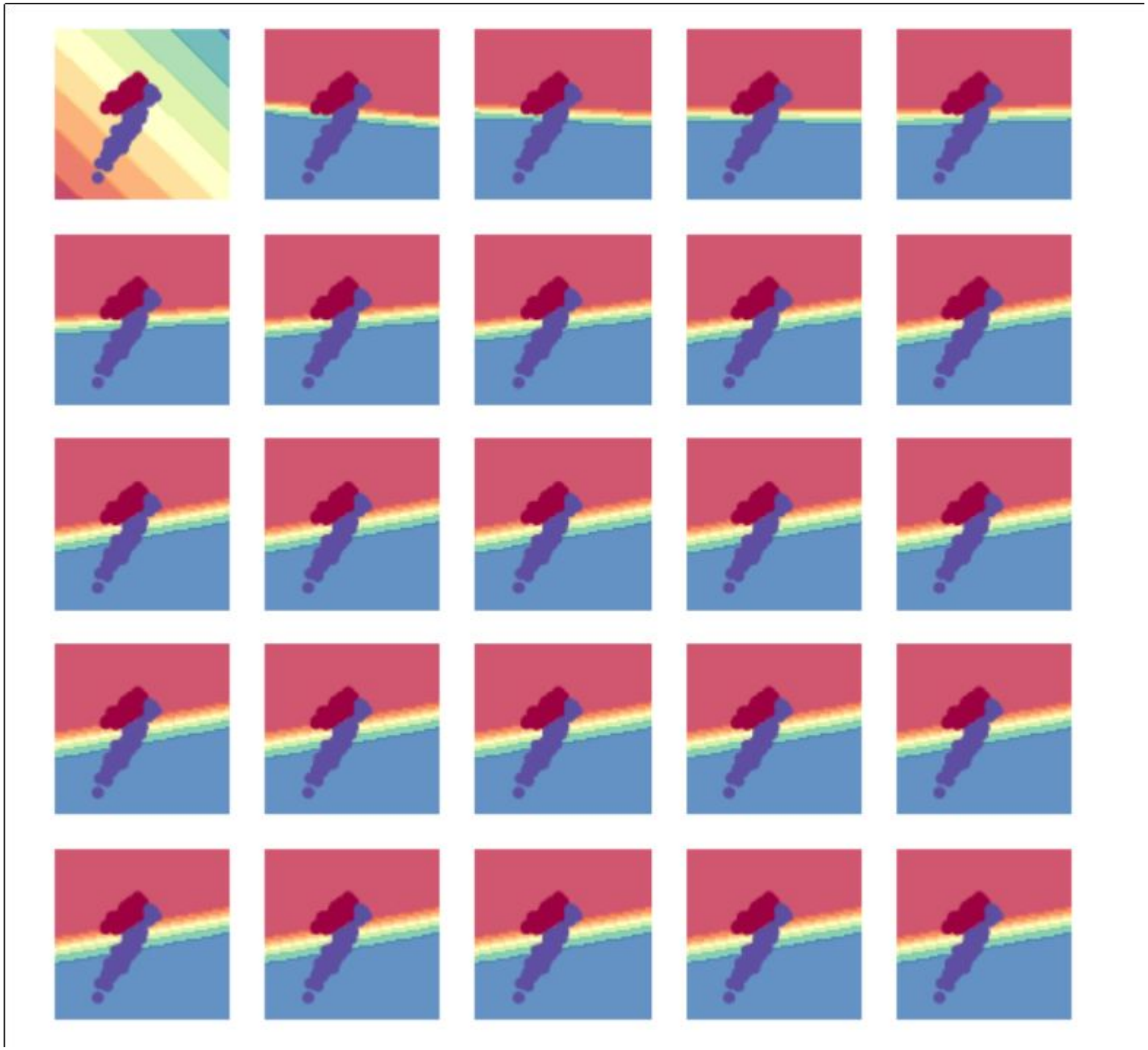
- Add missing code:
  - Sigmoid activation function
  - Neuron function
  - Training loop
- You can now train a model & visualise the decision boundary!
  - Document how the training performs, with different training rates ('eta') and weight decay ('alpha)
  - Compare training with a simple linear classifier
  - How does the neuron fair on the more difficult Gaussian dataset?
- Advanced concepts
  - Change the activation function. (Think about the gradients.)
  - Often neural networks have an additional 'bias' input. Add this to your code.
  - Batch training - currently all data is used to build the single gradient.
  - What happens if you try and use the method for regression (against a function)?
  - Can you improve the regression performance by adding extra neurons side-by-side, each fitting a different part of the function?
  - Compare regression to Gaussian processes: https://jarvist.github.io/2021-PhysicsMachineLearningPracticum/02_GaussianProcessPotentialEnergySurface.html
- Suggested homework / self-study
  - David MacKay, Information Theory, Inference and Learning Algorithms (ITILA), 2003, Chapters 38–42.
  - PyTorch '60 minute Blitz' https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
  - https://fluxml.ai/tutorials/2020/09/15/deep-learning-flux.html - Julia ML library, the tutorial is based on the Pytorch 60 minute Blitz

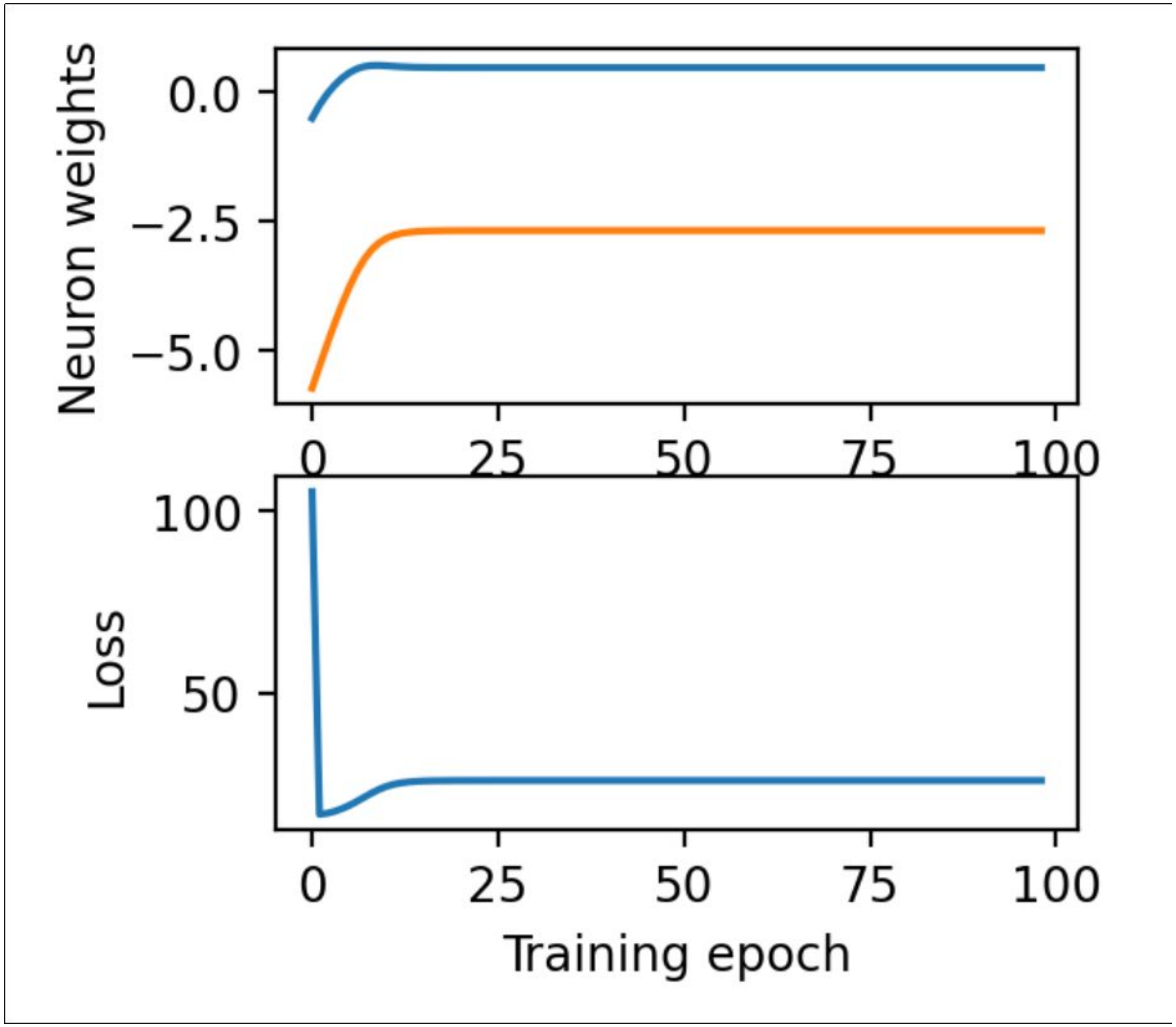


Decision boundary visualisation



Decision boundary visualisation





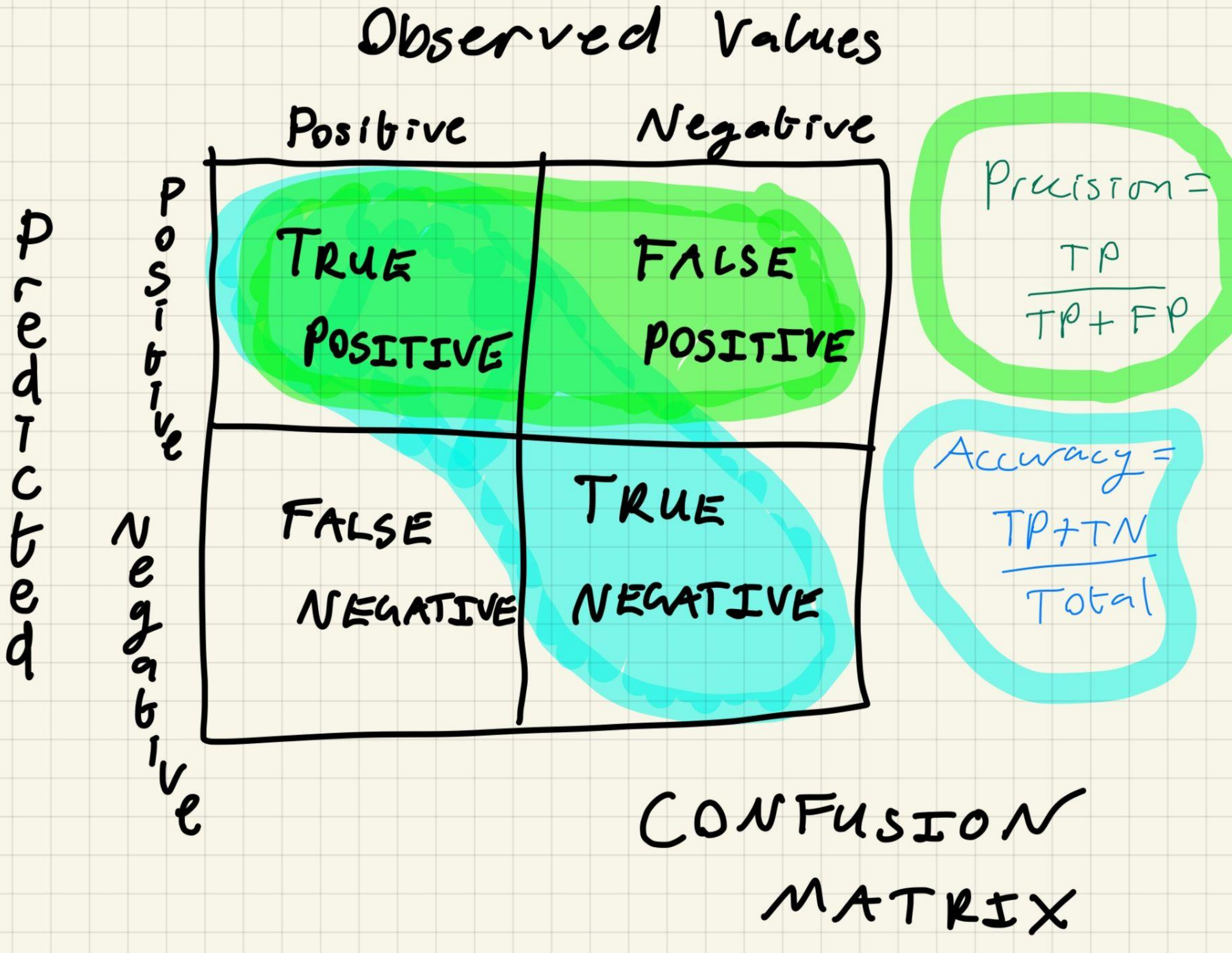
'Learning curves'

		Observed Values	
		Positive	Negative
Predicted	Positive	TRUE POSITIVE	FALSE POSITIVE
Predicted	Negative	FALSE NEGATIVE	TRUE NEGATIVE

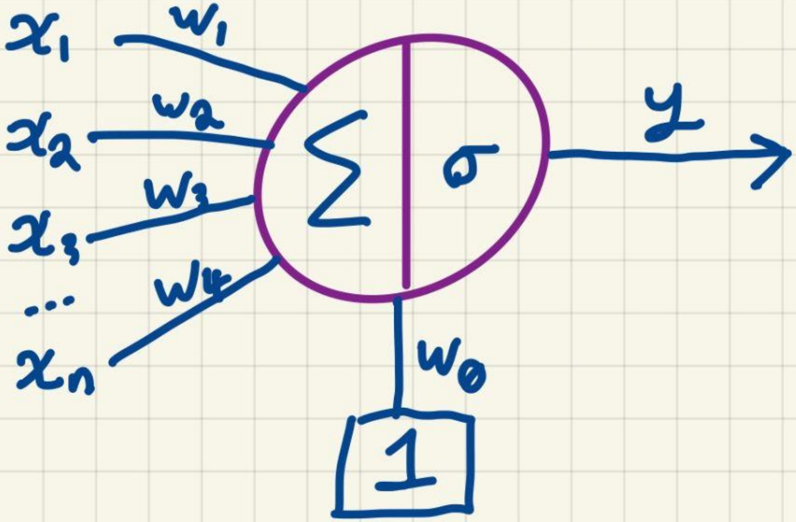
$$\text{Precision} = \frac{TP}{TP+FP}$$

$$\text{Accuracy} = \frac{TP+TN}{\text{Total}}$$

CONFUSION MATRIX



Observed Values
Positive
Negative
Predicted
Positive
Negative
TRUE POSITIVE
FALSE POSITIVE
FALSE NEGATIVE
TRUE NEGATIVE
Precision = $\frac{TP}{TP+FP}$
Accuracy = $\frac{TP+TN}{Total}$
CONFUSION MATRIX



Training: (supervised)

$$\boxed{\begin{aligned} e &= y - t \\ g_i &= -e x_i \\ \Delta w_i &= -\eta \sum_n g_i \end{aligned}}$$

Neuron:

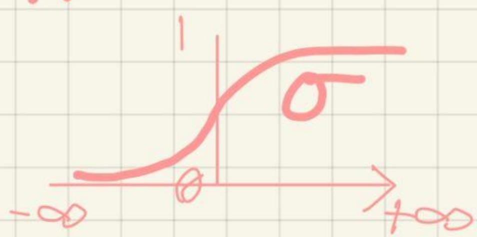
$$\boxed{\begin{aligned} a &= \sum_i w_i x_i \\ y &= \sigma(a) \end{aligned}}$$

Linear

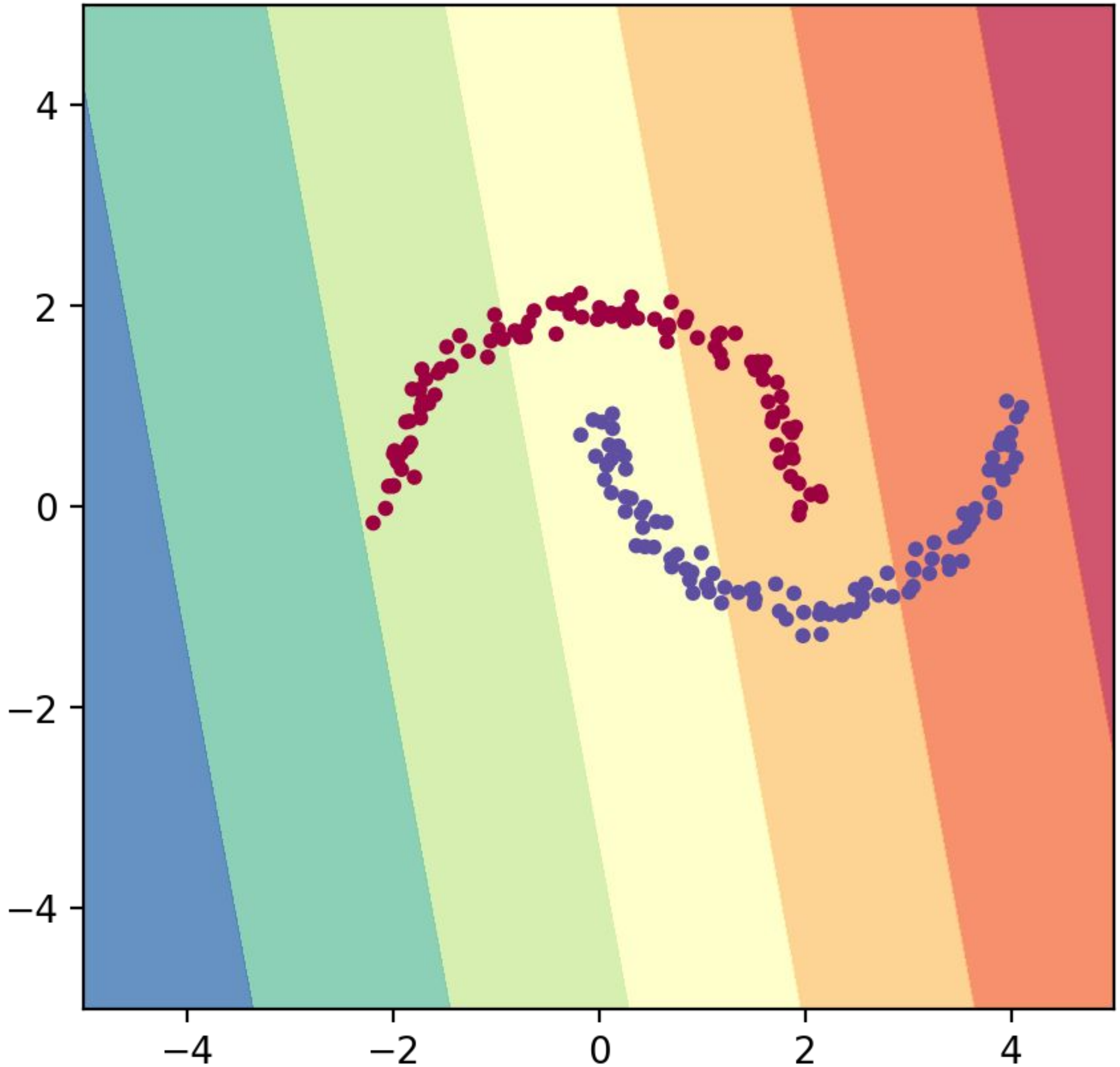
$$a = X * w$$

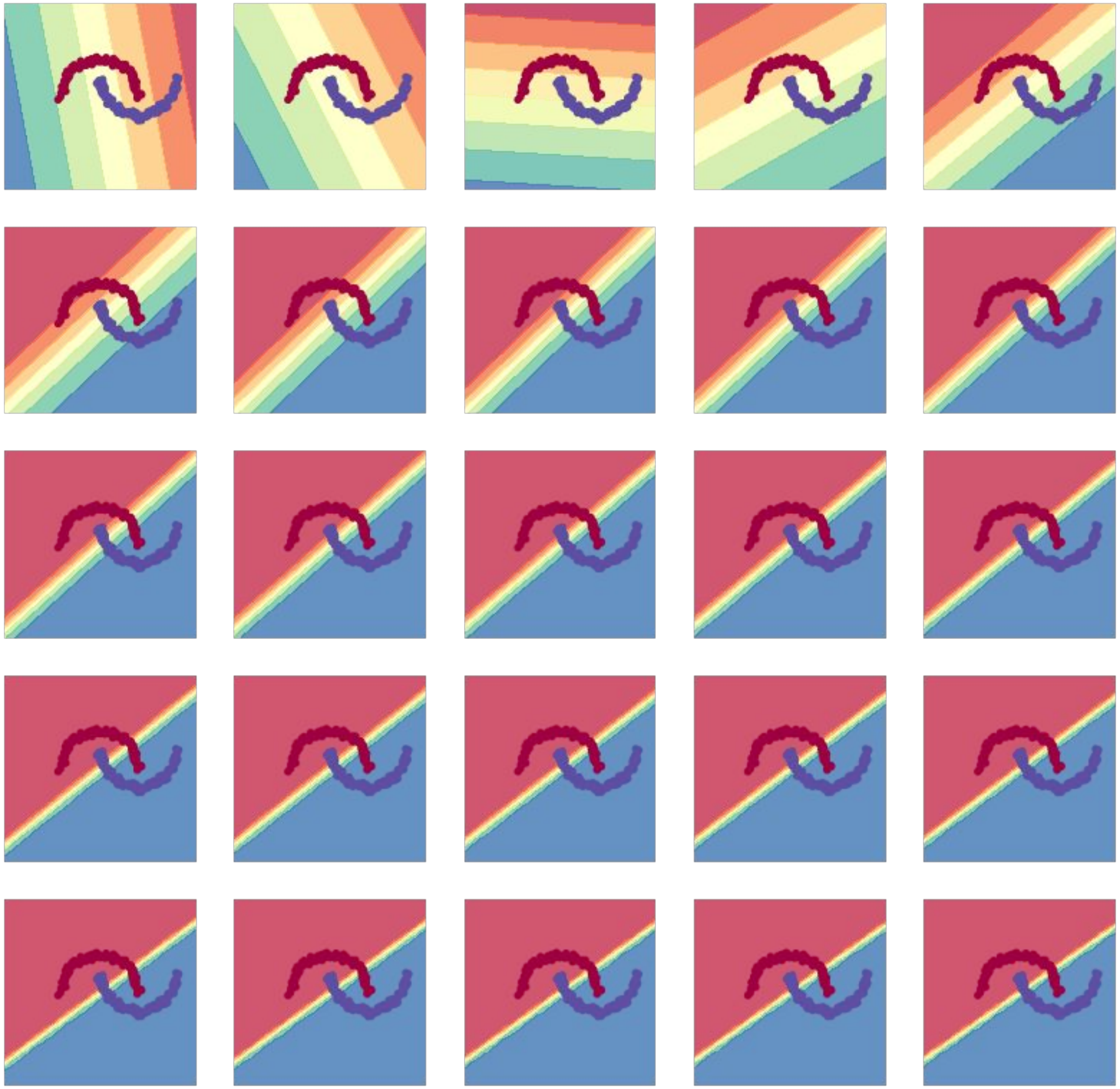
$$X = \begin{bmatrix} x_1 & x_2 \\ x_1 & x_2 \\ x_1 & x_2 \\ x_1 & x_2 \end{bmatrix} \qquad w = \begin{bmatrix} w_1 & w_2 \end{bmatrix}$$

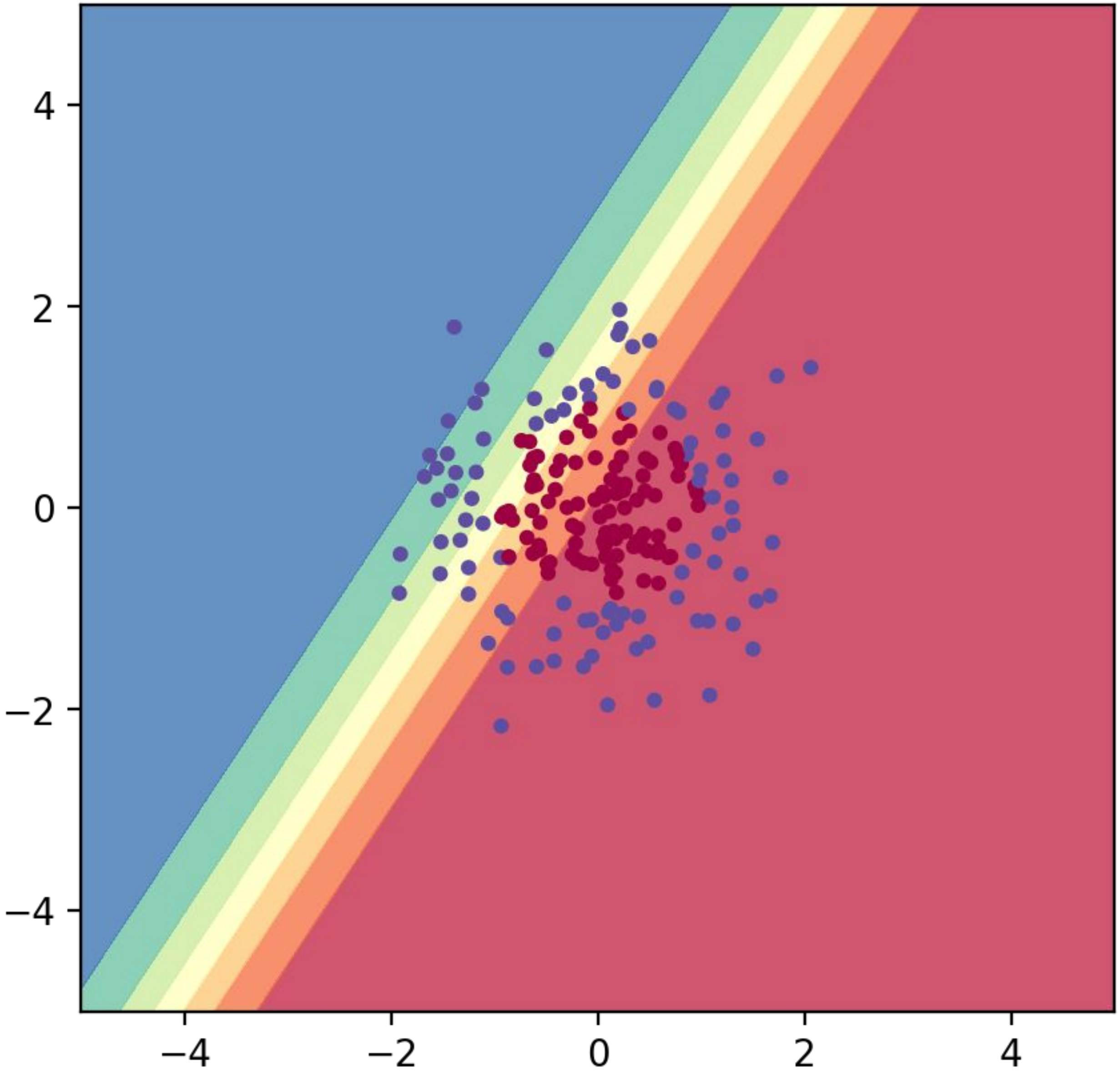
Non-linear



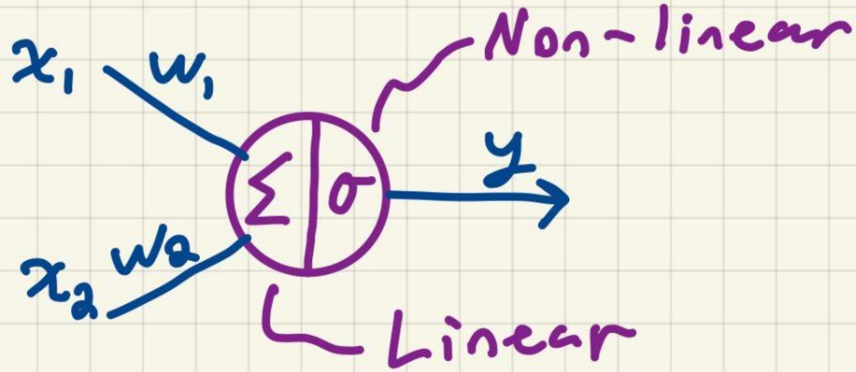
Single neuron by hand. No magic involved!


4
2
0
−2
−4
−4
−2
0
2
4

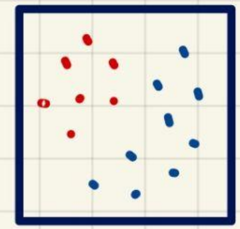


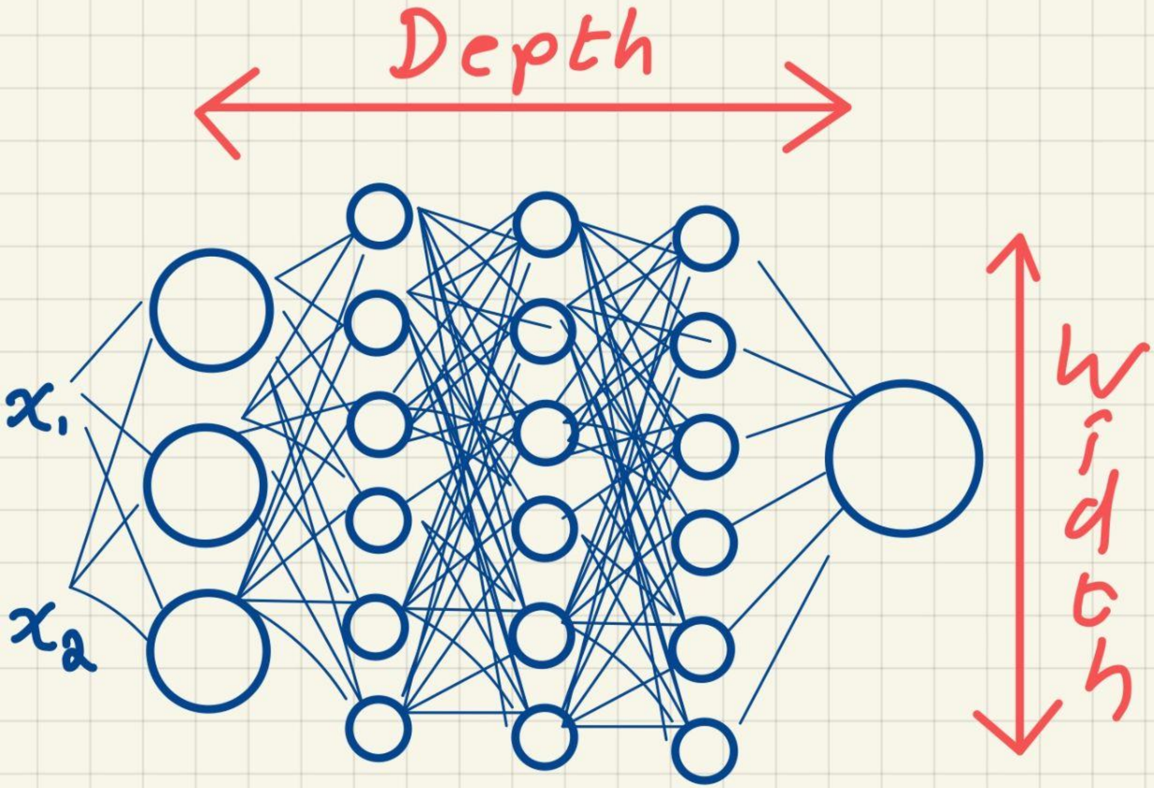


Similarly poor on overlapping Gaussians...



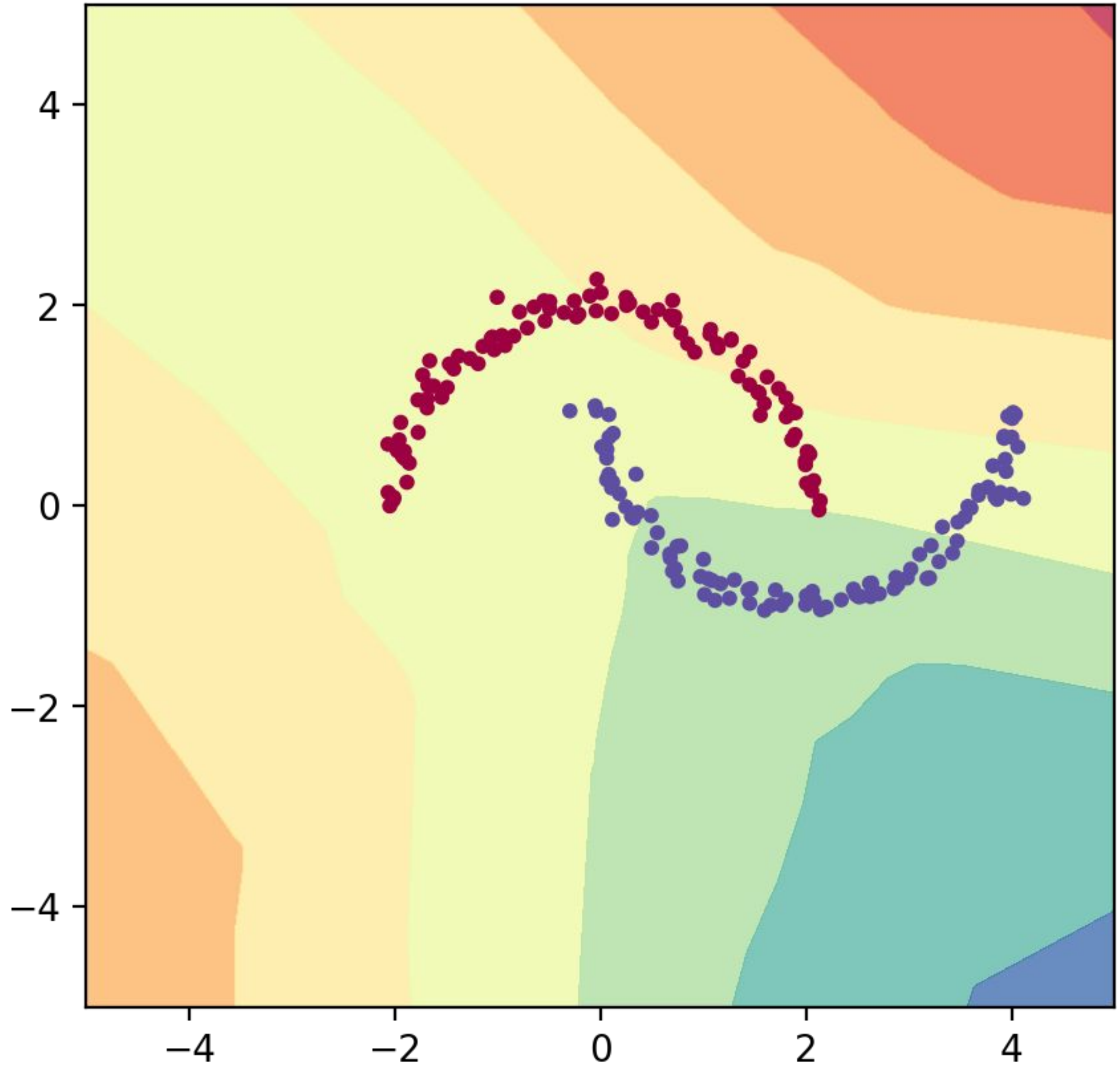
- learns from data
- only forms linear decision boundary





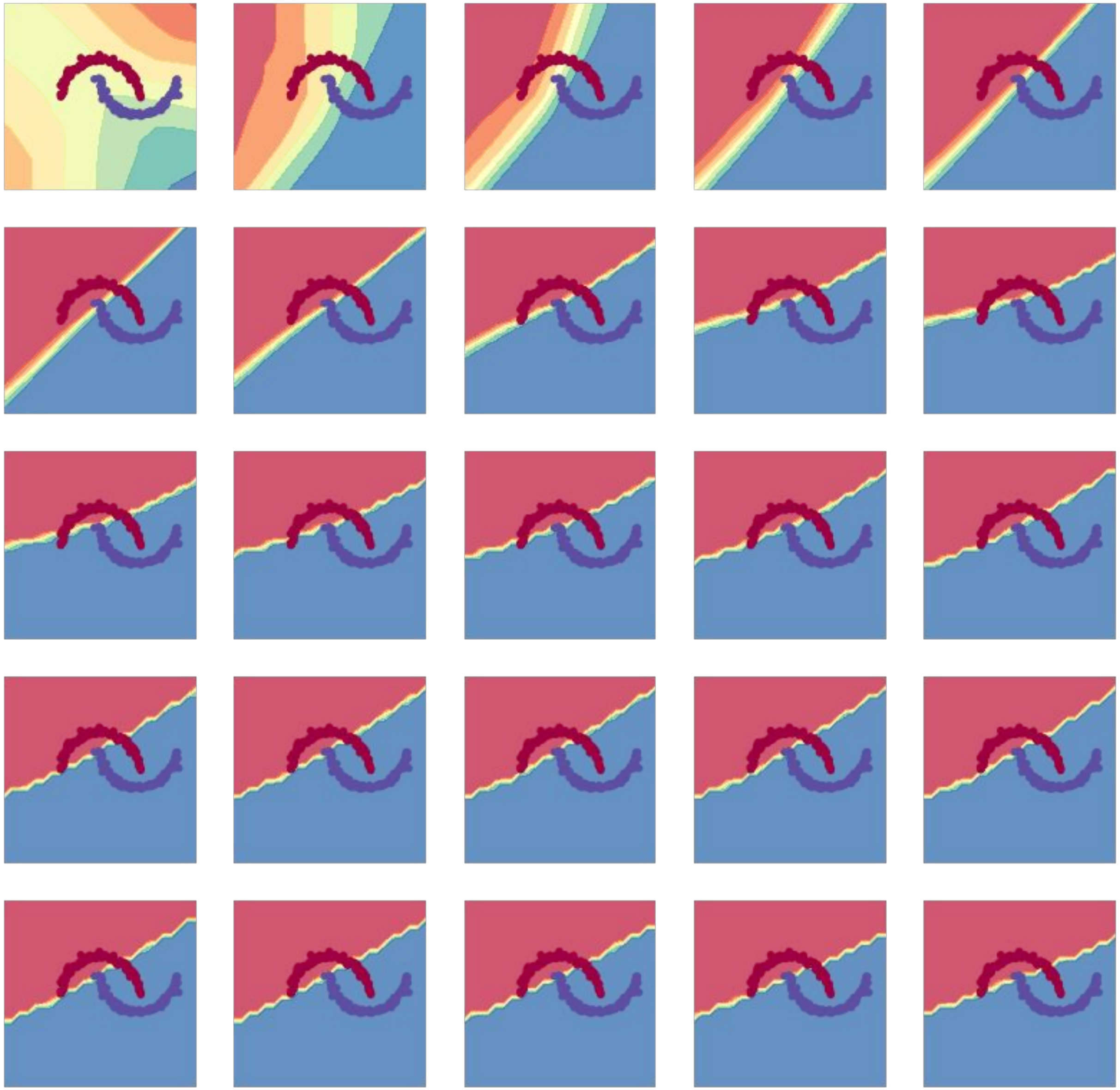
- universal function approximator
- millions of weights trivial
- billions of weights SOTA

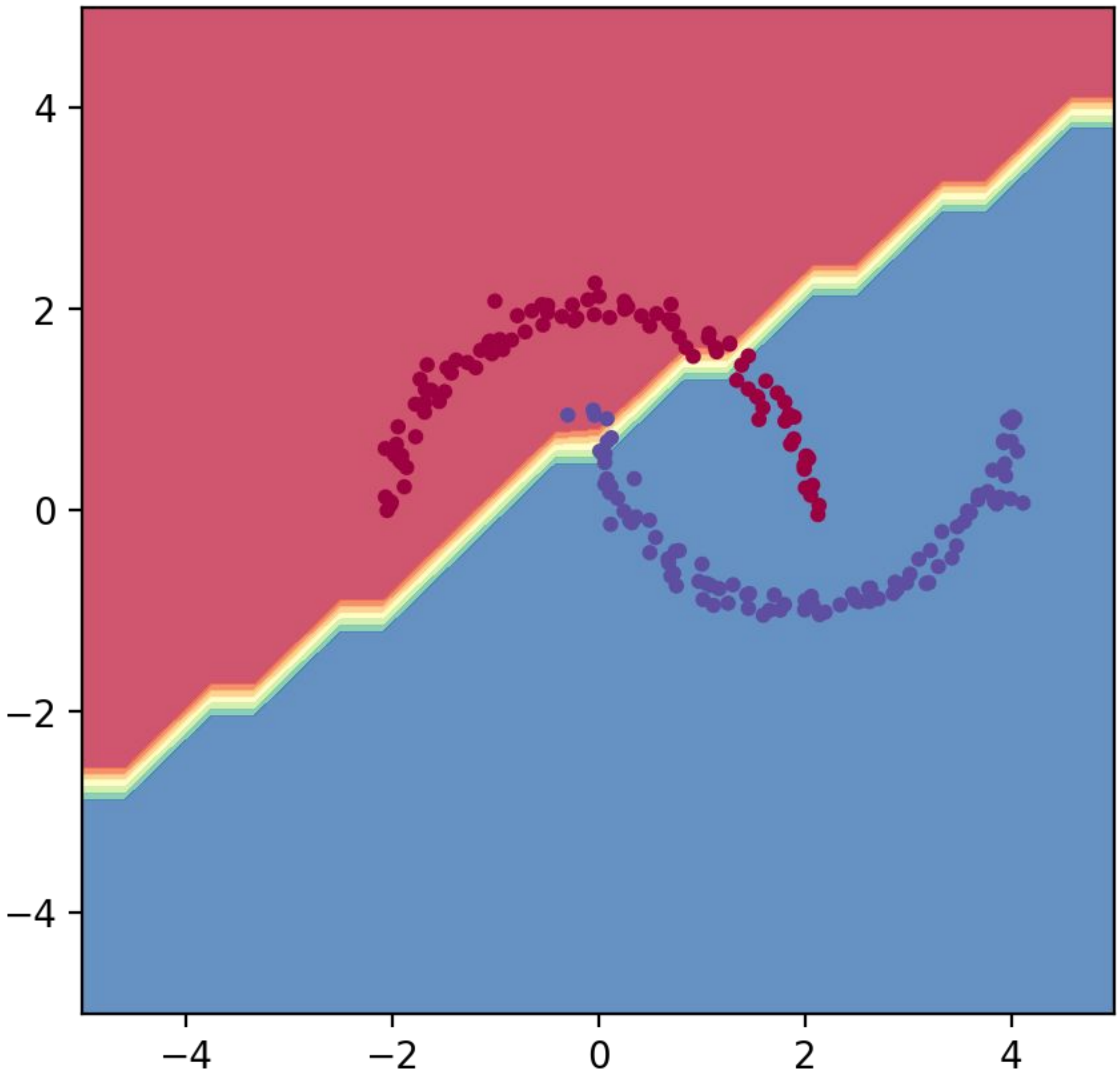
Magic comes from emergent behaviour...



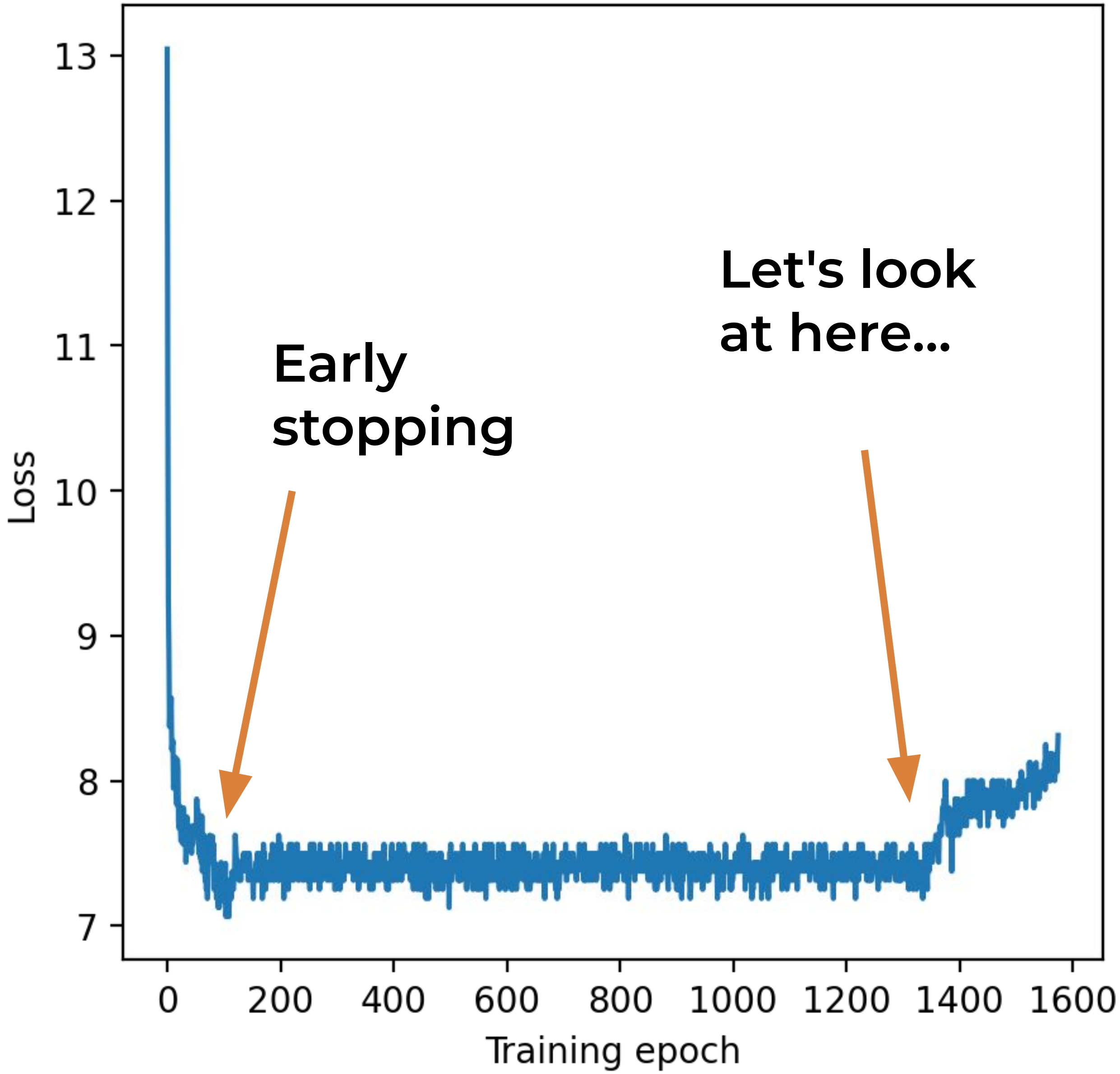
Multilayer perceptron: 2=>40, ReLU; 40=> 1, Sigmoid
Training by back propagation (G.Hinton 1986)

Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. Nature 323, 533–536 (1986). https://doi.org/10.1038/323533a0

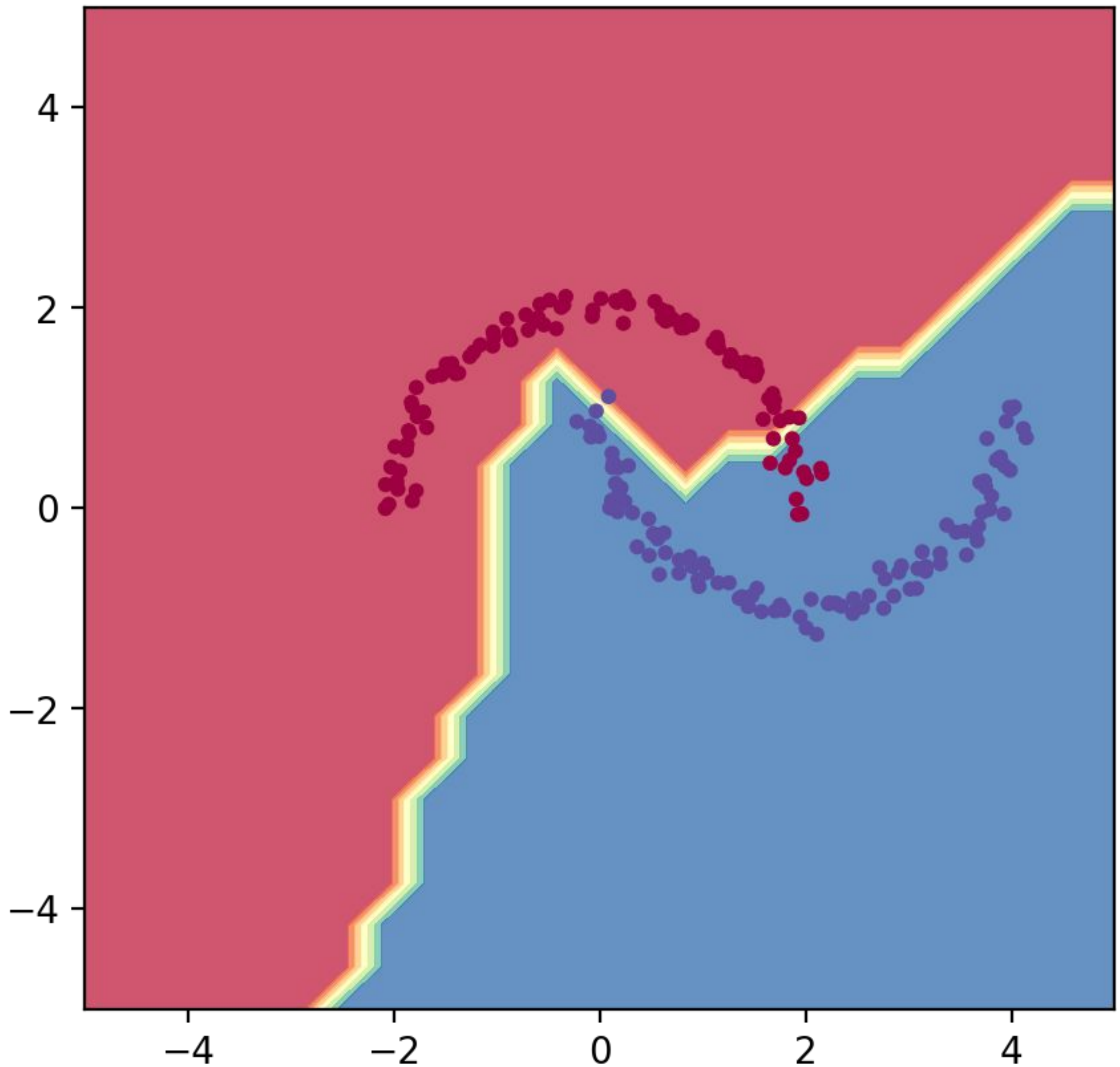




OK! Solution not so good (accuracy = 35%?) But non-linear, and interesting wiggles



Training is highly empirical; can take a long time.



4
2
0
−2
−4
−4
−2
0
2
4

§3.3 Practical

- Add missing (PyTorch!) code:
  - Rebuild a single neuron in Pytorch
    - Layers, loss function
    - accuracy function: true positives + true negatives / total
  - Multilayer Perceptron
    - Define architecture; define loss function
- Single neuron experiments - now with PyTorch
  - Change the activation function. (Think about the gradients.)
  - What effect does the bias have?
  - Set the weights by hand, and document the decision boundary.
- Multi-layer perceptron experiments - with PyTorch
  - Experiment with model construction, task, learning parameters, etc.
    - Document as you go! (Jupyter is really bad for this.)
    - Scoreboard for highest accuracy against each task:
      - Moons
      - Gaussians
      - Your own challenge here?
- Extend PyTorch code to Regression
  - What happens if you try and use the method for regression (against a function)?
    - Collect data points from a function, and fit over a range, i.e.
      - y=sin(x) ; y=x ; y=x^2 ; y=sinh(x)
  - Can you improve the regression performance by adding extra neurons side-by-side, each fitting a different part of the function?
  - Compare regression to Gaussian processes: https://jarvist.github.io/2021-PhysicsMachineLearningPracticum/02_GaussianProcessPotentialEnergySurface.html