Scientific Computing Project 2a

Neural Networks

Matthew Crean

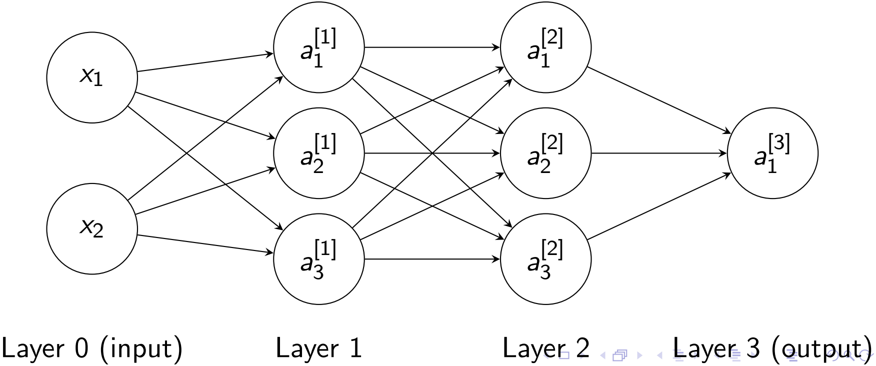
February 8, 2026

Background

- A neuron is a simple scalar functional unit, given by $\sigma : \mathbb{R} \to \mathbb{R}$
- In a feed-forward neural network, neurons are arranged in layers
  - Each neuron in a layer is connected to every neuron in the next layer
  - Each connection has an associated weight $\mathbf{W}$
  - Each neuron has an associated bias $\mathbf{b}$
- A network is trained by specifying 'training data' given a set of inputs and desired outputs, which changes the weights and biases to minimise the cost function
  - The cost function quantifies the error of the network's output compared to the desired output from the training data
- New inputs can then be classified by the trained network to retrieve the expected output

Neural Network Example: (2,3,3,1)

- Activation vector at each layer is $\mathbf{a}^{[l]} = \sigma_l \left(\mathbf{W}^l \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \right)$
  - $l = 1, \ldots, L$ corresponds to the layer
  - σ_l is the activation function for layer l
  - $\mathbf{W}^{[l]}$ and $\mathbf{b}^{[l]}$ are the weight matrix and bias vector of size $n_l \times n_{l-1}$ and length n_l respectively
  - $\mathbf{a}^{[l]} = \left(a_1^{[l]}, \ldots, a_{n_l}^{[l]} \right)^\top$, n_l is the number of neurons in layer l
  - $\mathbf{a}^{[0]} = (x_1, x_2)^\top$, the input to the network



Neural Network Implementation

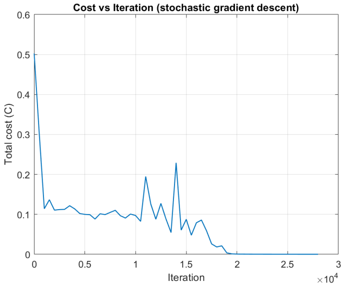
- Task: implement a feed-forward neural network for binary classification
- NeuralNetwork class:
  - Represents the entire neural network and the flow of data through it
  - Stores:
    - number of input variables to network
    - vector of all non-input layers ('NeuralNetworkLayer' objects)
  - Uses '*feed_forward*' to compute network output
    - does not modify model parameters
- NeuralNetworkLayer class:
  - Represents a single non-input layer of the network
  - For each layer, stores:
    - weight matrix
    - bias vector
    - activation function
  - Uses '*forward*' function which takes input activation vector and computes next layer's activation vector (computes $\mathbf{a}^{[l]}$)

Optimising Weights and Biases

- For the training algorithm, we aim to minimise the cost to find our weight matrices and bias vectors
- Derivatives of this function tell us how to change the network parameters to reduce the cost function
  - Derivatives applied practically using stochastic gradient descent (SGD), where parameters are updated using individual training examples chosen at random rather than the full dataset
- These derivatives must be calculated via either:
  - back-propagation
    - propoagates gradients 'backward' through the network
  - finite differencing
    - approximates derivatives by perturbing each parameter individually and recomputing cost
- Implemented within '*train*' function, given training rate η
  - η acts as a step-size controling how strongly gradients influence the update of weights and biases at each iteration
  - training stopped once either target cost τ reached or maximum number of iterations reached

Simple Test Case

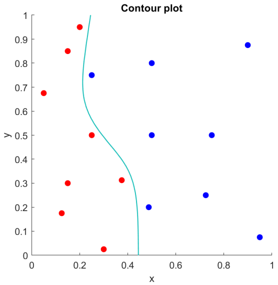
- Build neural network with 4 layers containing (2,3,3,1) neurons respectively
  - Activation function: $\tanh(x)$
  - Learning rate: $\eta = 0.1$
  - Target cost: $\tau = 10^{-4}$
  - Max iterations: 10^5
  - Initialise weights and biases from $N(0, 0.1^2)$



Plot shows total cost decreases to below target cost within maximum number of iterations.

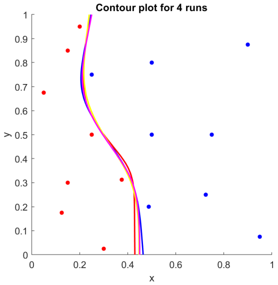
- Initial rapid decay
- Oscillations due to stochastic nature of SGD (point randomly selected from training set each iteration)
- Network trains successfully

Simple Test Case: Contour Plots



Contour plot of network output after training once

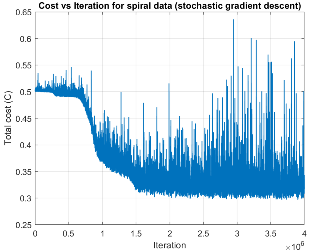
- Network has learned a decision boundary that separates the two classes accurately

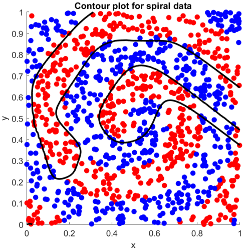


Contour plot after repeating training four times

- Slight variations in boundary due to different random initialisations and stochastic training process
- Training process is very reliable

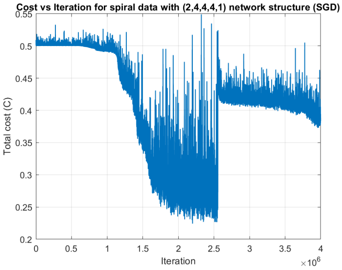
Spiral Test Case

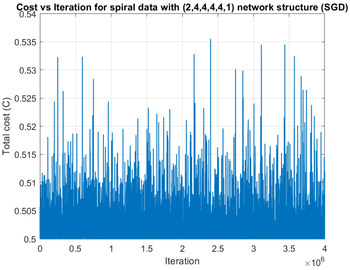


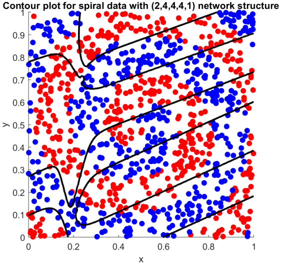


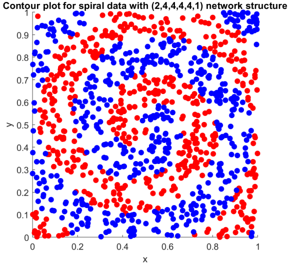
- Build network on the spiral dataset, with a (2,4,4,1) structure
  - Iteration plot shows cost initially decreases, but then starts to oscillate around cost 0.35
  - Network fails to converge to target cost
  - Contour plot shows the network captures some of the spiral shape, but not very accurately
- The (2,4,4,1) network is not expressive enough to capture the sprial structure; a wider (increasing neurons) or deeper (increasing layers) network should be tested

Spiral: Deeper Network Plots

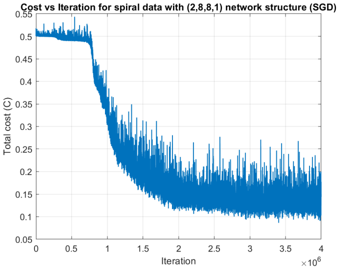


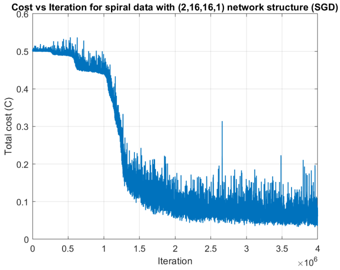


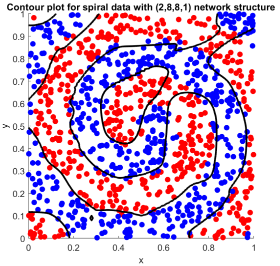


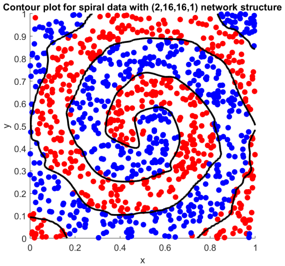


Spiral: Wider Network Plots









Spiral: Varying Network Structure Analysis

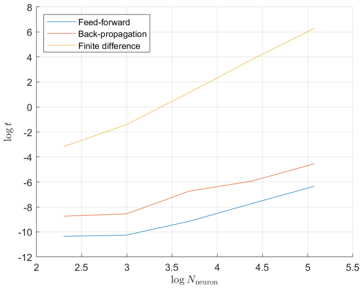
- Deeper networks:
  - Increasing the number of layers shows decrease in performance of the neural network
  - (2,4,4,4,1) structure provides worse approximation of boundary and more unpredictable oscillations
  - (2,4,4,4,4,1) structure fails completely; does not classify the points at all and total cost does not go below 0.5
  - For the spiral data, deeper networks are worse as noise from each update in SGD passes through more layers, so it has a more drastic impact on resulting network
- Wider networks:
  - Much more effective at classifying spiral data
  - Neither networks fully converge but still approximate boundary for classification better than initial structure
  - Contour plot shows that (2,16,16,1) structure approximates classification boundary very well, despite some outliers
  - Increasing neurons is more effective as network has more parameters per layer to be optimised, improving the approximation

Theoretical Computational Cost

- Aim: Assess cost of training algorithm
- Suppose a network consists of N_{layer} layers, each with N_{neuron} neurons (both large to neglect small order operations)
- Single feed-forward operation:
  - $\mathbf{a}^{[l]} = \sigma_l \left(\mathbf{W}^{[l]} \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \right)$
  - $\mathbf{W}^{[l]}$ $N_{\text{neuron}} \times N_{\text{neuron}}$ matrix and $\mathbf{a}^{[l-1]}$ length N_{neuron} vector
  - $\mathbf{W}^{[l]} \mathbf{a}^{[l-1]}$ requires $O\left(N_{\text{neuron}}^2\right)$ operations
  - Repeated over all layers, so feed-forward operation costs $O\left(N_{\text{layer}} N_{\text{neuron}}^2\right)$
- Derivatives by finite-differencing:
  - To compute derivatives by finite-differencing, we must complete a full feed-forward for each parameter
  - There are $O\left(N_{\text{layer}} N_{\text{neuron}}^2\right)$ parameters
  - One feed-forward $\times$ number of parameters $= O\left(N_{\text{layer}}^2 N_{\text{neuron}}^4\right)$
- Derivatives by back-propagation:
  - We pass forward once and pass backwards once
  - $2 \times O\left(N_{\text{layer}} N_{\text{neuron}}^2\right) = O\left(N_{\text{layer}} N_{\text{neuron}}^2\right)$

Computational Cost: Varying Neurons

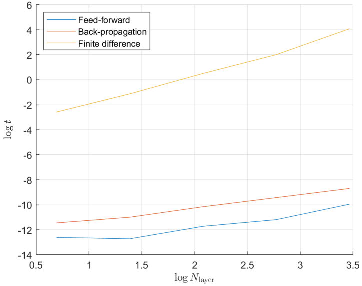
- Investigate costs of training by fixing $N_{\text{layer}} = 10$ and varying $N_{\text{neuron}} \in \{10, 20, 40, 80, 160\}$
  - Simulate random input and output data, initialise weights and biases randomly, record time taken for a single operation
  - Log-log plot should show gradient of 2 for feed-forward and back-propagation, and 4 for finite differencing



- Produces approximate straight lines
- Taking start and end points, find gradient:
  - Feed-forward ≈ 1.44
  - Finite differencing ≈ 3.41
  - Back-propagation ≈ 1.51
- Measured gradients are lower than the asymptotic predictions (2 and 4) due to limited range of N_{neuron} to prevent long run-times
- Results align with theory

Computational Cost: Varying Layers

- Further investigate costs of training by fixing $N_{\text{neuron}} = 40$ and varying $N_{\text{layer}} \in \{2, 4, 8, 16, 32\}$
  - Same set-up as varying neurons
  - Log-log plot should show gradient of 1 for feed-forward and back-propagation, and 2 for finite differencing



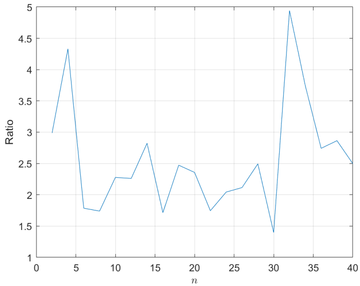
- Similarly, taking start and endpoints, find gradient of:
  - Feed-forward ≈ 0.96
  - Finite differencing ≈ 2.40
  - Back-propagation ≈ 0.99
- Measured gradients for feed-forward and back-propagation align with theory
- Finite differencing may have larger than predicted gradient due to increased depth resulting in more storage required, so slower run time

Interlacing Computation

- Currently in the implemented training algorithm, we:
  - Store each partial differential of each component of weight matrix and bias vector
  - Use stored values to update weight and bias
- This is inefficent and could be improved
  - Requires storing full gradient matrices and revisiting them later; more storage used than necessary, affecting run times
  - Several loops means longer computational times
- We seek to interlace these steps to make the algorithm more efficient
  - For testing purposes, use data from the simple test case
  - Train with identical initial parameters and time how long each algorithm takes
  - Use networks of varying sizes

Interlacing Computation Analysis

- Use $(2,n,n,1)$ network for varying n up to $n = 40$, since we know this reliably converges for the simple test case data by previous analysis
- Report ratio of time for the original training method against the time for the interlaced training method (implemented via new '*train_interlaced*' function)



- All ratios reliably larger than 1
- Interlaced algorithm is significantly and consistently faster than initial implementation of the training algorithm
- Aligns with prediction