ELEC0144
Machine Learning for Robotics

Assignment 2
Backpropagation in Multilayer Perceptron
Transfer Learning in Convolutional Neural Network
&
Reinforcement Learning

Submission Date

05/02/2025

Table of Contents

Executive summary

This report details the implementation and evaluation of various machine learning techniques across four tasks.

Task 1 involved developing a 1–3–1 neural network for regression, using SGD and ADAM optimizers with different activation functions. MATLAB’s Neural Network Toolbox was compared with a manual gradient descent approach. A 1–6–1 ReLU network was also constructed to approximate noisy cubic data.

Task 2 focused on classifying the Iris dataset using Multilayer Perceptrons (MLPs). A manually implemented model with SGD achieved 97.78% accuracy, with ADAM showing faster convergence. Various architectures and activation functions were tested, confirming the dataset's suitability for high-accuracy classification.

Task 3 applied transfer learning using pre-trained AlexNet and GoogLeNet models on a custom fruit image dataset. Performance was evaluated, and learning parameters were adjusted to optimize classification accuracy.

Task 4 explored tabular Q-learning, where an agent learned optimal paths using a reward-based system. The epsilon-greedy method enabled exploration before shifting to reward-maximizing decisions, with training continuing until convergence.

This report provides insights into neural network training, optimization, transfer learning, and reinforcement learning, comparing manual implementations with established toolboxes to assess efficiency and performance.

Task 1: Regression

1.1 Derivation of Backpropagation for a 1-3-1 Network

1.1.1 Network Architecture

Figure 1.1 is the diagram of the 1–3–1 MLP structure, showing the input node, three hidden nodes, and one output node.

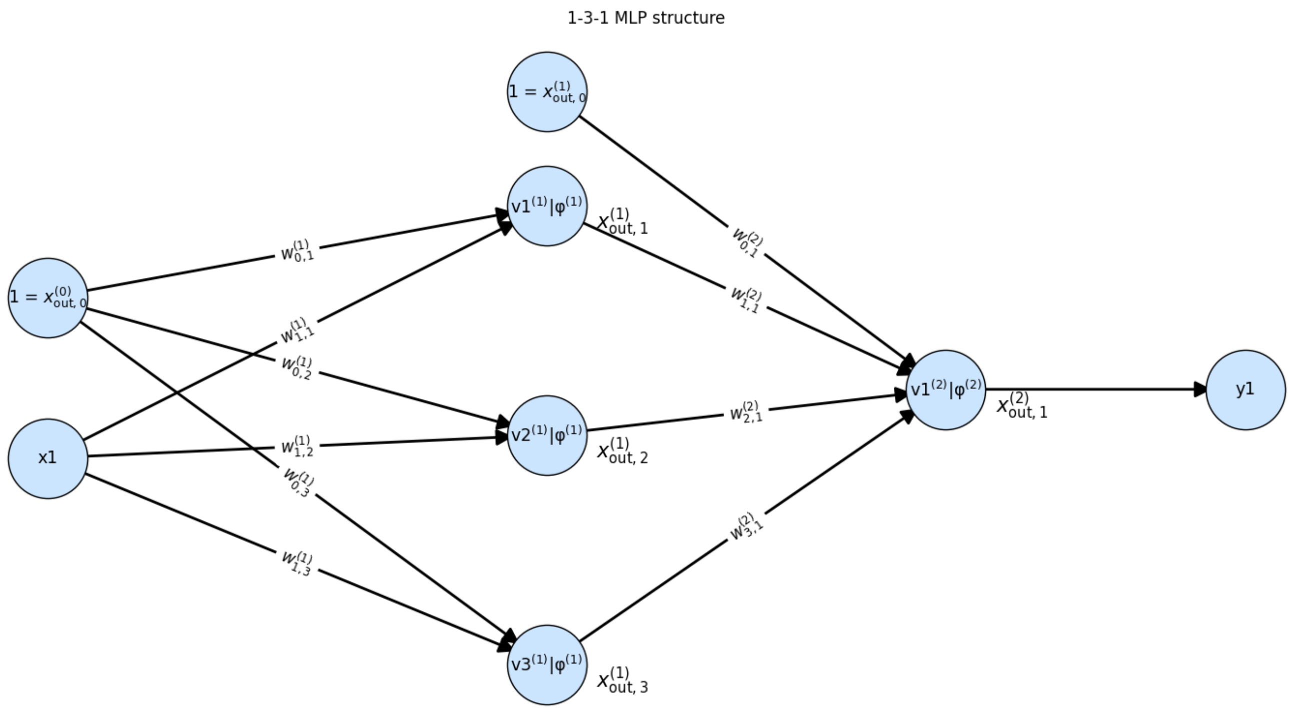


Figure 1.1: 1–3–1 MLP structure

1.1.2 Activation Functions and Their Derivatives

Hidden Layer Activation Function

The hidden layer uses the hyperbolic tangent activation function:

$$\phi^{(1)}(v) = \tanh(v) = \frac{e^{v} - e^{-v}}{e^{v} + e^{-v}} \quad \Rightarrow \quad \frac{d\phi^{(1)}}{dv} = 1 - \left[\phi^{(1)}(v)\right]^{2}. \tag{1}$$

Output Layer Activation Function

The output layer uses a linear activation function:

$$\phi^{(2)}(v) = v \quad \Rightarrow \quad \frac{d\phi^{(2)}}{dv} = 1. \tag{2}$$

1.1.3 Local Gradients

Output Layer Local Gradient

For the output node, the local gradient is:

$$\delta_1^{(2)} = (d - y)\,\phi'^{(2)}\left(v^{(2)}\right) = (d - y) \cdot 1 = d - y, \tag{3}$$

where d is the desired output and y is the actual output.

Hidden Layer Local Gradients

For individual nodes, the gradients are:

$$\delta_1^{(1)} = \left(\delta_1^{(2)} \cdot w_{11}^{(2)}\right) \cdot \frac{d\phi^{(1)}}{dv_1^{(1)}} = \left[(d - y) \cdot w_{11}^{(2)}\right] \cdot \left(1 - \tanh^2\left(\mathrm{v}_1^{(1)}\right)\right), \tag{4}$$

$$\delta_2^{(1)} = \left(\delta_1^{(2)} \cdot w_{21}^{(2)}\right) \cdot \frac{d\phi^{(1)}}{dv_2^{(1)}} = \left[(d - y) \cdot w_{21}^{(2)}\right] \cdot \left(1 - \tanh^2\left(v_2^{(1)}\right)\right), \tag{5}$$

$$\delta_3^{(1)} = \left(\delta_1^{(2)} \cdot w_{31}^{(2)}\right) \cdot \frac{d\phi^{(1)}}{dv_3^{(1)}} = \left[(d - y) \cdot w_{31}^{(2)}\right] \cdot \left(1 - \tanh^2\left(v_3^{(1)}\right)\right). \tag{6}$$

Generalizing,

$$\delta_j^{(1)} = \left(\delta_1^{(2)} \cdot w_{j1}^{(2)}\right) \cdot \frac{d\phi^{(1)}}{dv_j^{(1)}} = (d - y)\, w_{j1}^{(2)} \left(1 - \tanh^2\left(v_j^{(1)}\right)\right). \tag{7}$$

1.1.4 Weight Update Formula

Weights are updated by:

$$w_{ij,\text{new}}^{(s)} = w_{ij,\text{old}}^{(s)} + \eta^{(s)}\, \delta_j^{(s)}\, x_{\text{out},i}^{(s-1)}, \tag{8}$$

where

- $\eta^{(s)}$ is the learning rate for layers s,
- $\delta_j^{(s)}$ is the local gradient for node j in layer s,
- $x_{\text{out},i}^{(s-1)}$ is the output from node i in the previous layer.

Output Layer Weight Update

For the output layer ($s = 2,\ i = 2,\ j = 1$), taking $w_{21}^{(2)}$ as an example:

$$w_{21,\text{new}}^{(2)} = w_{21,\text{old}}^{(2)} + \eta^{(2)}\, \delta_1^{(2)}\, x_{\text{out},2}^{(1)}. \tag{9}$$

Substituting $\delta_1^{(2)} = (d - y)\ and\ x_{\text{out},2}^{(1)} = \tanh\left(v_2^{(1)}\right)$ gives

$$w_{21,\text{new}}^{(2)} = w_{21,\text{old}}^{(2)} + \eta^{(2)}\, (d - y)\, \tanh\left(v_2^{(1)}\right). \tag{10}$$

Generalization for Output Layer

For all output layer weights:

$$w_{i1,\text{new}}^{(2)} = w_{i1,\text{old}}^{(2)} + \eta^{(2)} \cdot (d - y) \cdot \tanh\left(v_i^{(1)}\right) \tag{11}$$

Hidden Layer Weight Update

For all the hidden layer $(s = 1,\ i = 1,\ j = 2)$, taking $w_{12}^{(1)}$ as an example:

$$w_{12,\text{new}}^{(1)} = w_{12,\text{old}}^{(1)} + \eta^{(1)}\, \delta_2^{(1)}\, x_{\text{out},1}^{(0)} \tag{12}$$

where $x_{\text{out},1}^{(0)} = x_1$. Thus,

$$w_{12,\text{new}}^{(1)} = w_{12,\text{old}}^{(1)} + \eta^{(1)} \left[(d - y)\, w_{21}^{(2)}\right] \left(1 - \tanh^2\left(v_2^{(1)}\right)\right) x_1. \tag{13}$$

Generalization for Hidden Layer

For all hidden layer weights:

$$w_{1j,\text{new}}^{(1)} = w_{1j,\text{old}}^{(1)} + \eta^{(1)} \cdot \left[(d - y) \cdot w_{j1}^{(2)}\right] \cdot \left(1 - \tanh^2\left(v_j^{(1)}\right)\right) \cdot x_1 \tag{14}$$

Bias Update

When considering the bias, all i values should be 0.

For Output Layer Bias:

$$w_{01,\text{new}}^{(2)} = w_{01,\text{old}}^{(2)} + \eta^{(2)} \cdot \delta_1^{(2)} \cdot x_{\text{out},0}^{(1)} \tag{15}$$

Since $(x_{\text{out},0}^{(1)} = x_{\text{out},0}^{(0)} = 1)$, then we have:

$$w_{01,\text{new}}^{(2)} = w_{01,\text{old}}^{(2)} + \eta^{(2)} \cdot (d - y) \cdot 1 \tag{16}$$

For Hidden Layer Bias:

$$w_{0j,\text{new}}^{(1)} = w_{0j,\text{old}}^{(1)} + \eta^{(1)} \cdot \delta_j^{(1)} \cdot 1 \tag{17}$$

Expanding $(\delta_j^{(1)})$, then we have:

$$w_{0j,\text{new}}^{(1)} = w_{0j,\text{old}}^{(1)} + \eta^{(1)} \cdot \left[(d - y) \cdot w_{j1}^{(2)}\right] \cdot \left(1 - \tanh^2\left(v_j^{(1)}\right)\right) \tag{18}$$

1.2 Implementation with SGD

1.2.1 SGD Training Details

We developed a MATLAB script that:

- Generates data: We sample x in [-1, 1] (step 0.05) and compute

$$d = 0.8x^3 + 0.3x^2 - 0.4x + \text{noise}, \quad \text{with noise} \sim \mathcal{N}(0,\ 0.02^2). \tag{19}$$

- Initializes a 1–3–1 network with random weights/biases in [−0.5, 0.5].

- Trains for 30,000 epochs using pure SGD. In each epoch:

    1. Shuffle all training samples (random permutation).
    2. For each example (x_n, d_n), do a forward pass and then a backward pass (computing local gradients δ) and update weights/biases immediately.

- Repeats training for three distinct learning rates $\eta \in \{0.1, 0.01, 0.001\}$. Each run reinitializes the network for fair comparison

- Plots (1) MSE vs. epoch for each LR, and (2) final regression curve vs. the training data.

1.2.2 Final MSE Table and Plots

We selected 30,000 epochs to allow the mid-range learning rate (0.01) to converge to its stable low error and to highlight how the largest rate (0.1) eventually oscillates. At the same time, we see that the smallest rate (0.001) remains underfitted at this point, indicating it would need even more epochs to catch up. After 30,000 epochs, we record the final mean-squared error for each LR, as shown in Table 1. Figures 1.2 and 1.3 illustrate the final regression curves and the MSE over epochs.

Learning Rate	**Final MSE (at 30,000 epochs)**
0.1	0.000465 (oscillates between 4×10^{-5} and 5×10^{-4}
0.01	0.000356 (stable convergence)
0.001	0.002936 (still decreasing)

Table 1.1: Final MSE for the three learning rates in plain SGD.

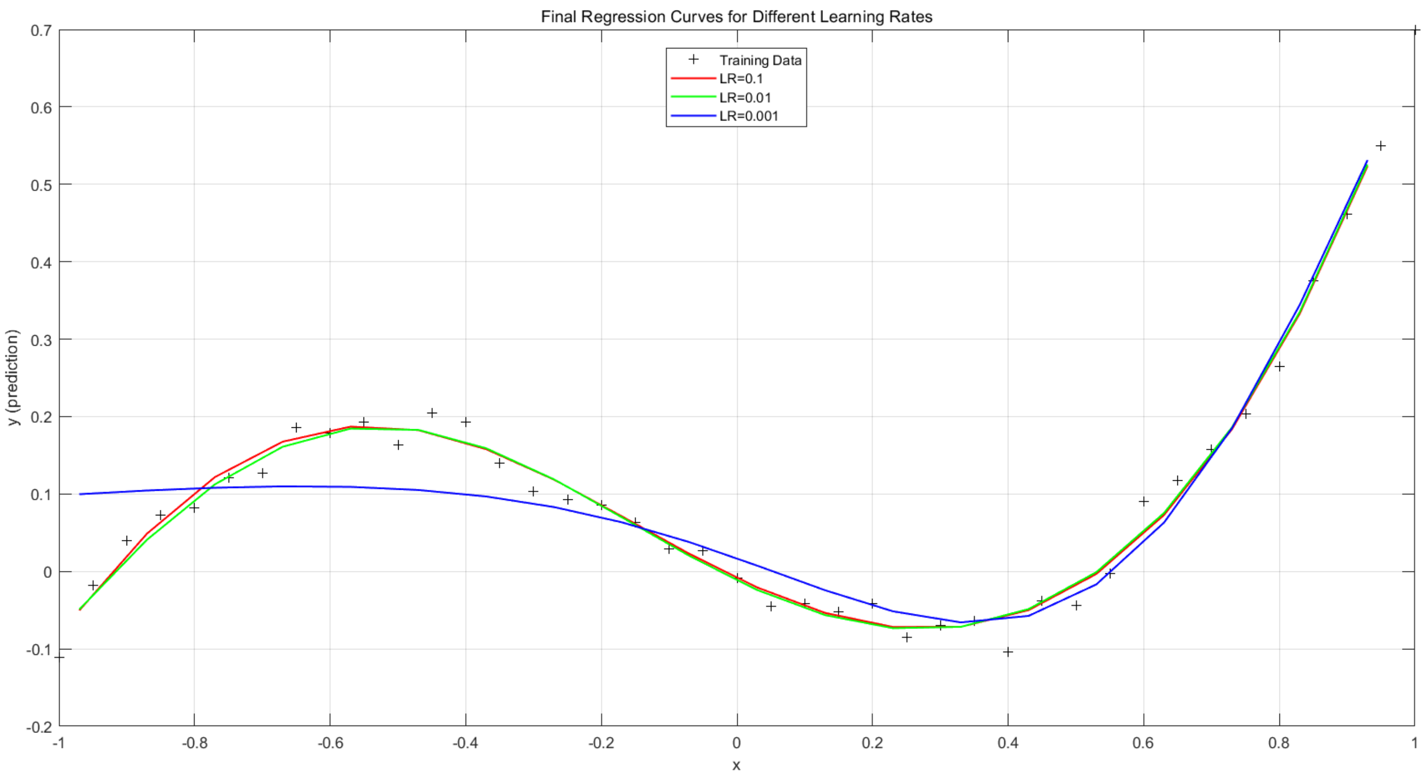


Figure 1.2: Final regression curves from SGD with three LRs (0.1 = red, 0.01 = green, 0.001 = blue). The black “+” marks are the training data.

1.2.3 Analysis and Observations

Convergence Behaviour. With $\eta = 0.1$, the error drops sharply at first but shows mild oscillations later. Meanwhile, $\eta = 0.01$ converges steadily to the lowest final MSE. Using $\eta = 0.001$ leads to a slower convergence: after 30,000 epochs, the MSE is high, though with additional training it might eventually reach a similar low error. Therefore $\eta = 0.01$ yields the best trade-off between speed and final MSE.

We see that in SGD, the learning rate is crucial:

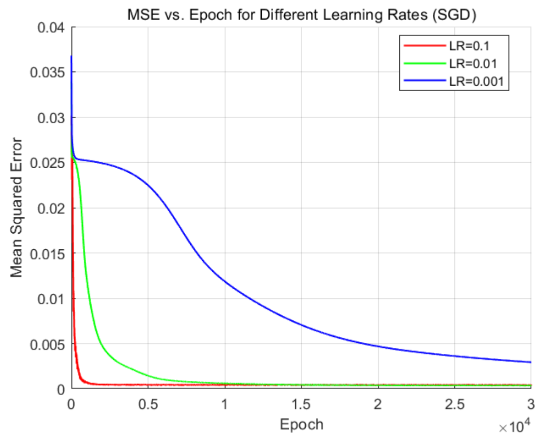


Figure 1.3: MSE vs. epoch for each LR (SGD). $\eta = 0.1$ (red), $\eta = 0.01$ (green) $\eta = 0.001$ (blue)

1.3 Implementation with ADAM

1.3.1 ADAM Formulation

In ADAM, at each iteration t, we compute the current gradient $(g_t = \nabla E(w_t) = \delta_j^{(s)}\, x_{\text{out},i}^{(s-1)})$. Then we update:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t, v_t = \beta_2 v_{t-1} + (1-\beta_2)\,(g_t^2), \tag{20}$$

which approximate the first (momentum) and second (variance) moments of the gradient. Next, we apply bias corrections.

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t}, \quad \widehat{v_t} = \frac{v_t}{1-\beta_2^t}, \qquad (21)\ (22)$$

Finally, the update step is

$$w_{t+1} = w_t \; - \eta\, \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon}, \tag{23}$$

Where $(\beta_1 = 0.9,\ \beta_2 = 0.999,\ \epsilon = 10^{-8})$

Why ADAM Improves Over Plain SGD

- **Adaptive Step Size.** ADAM adjust the update per dimension, making it less sensitive to the choice of η. By tracking both the first and second moments of the gradient, it automatically reduces step sizes in directions with larger curvature and allows bigger steps in flatter directions.
- **Momentum via β_1.** ADAM accumulates recent gradients using an exponential moving average, which filters out short-lived fluctuations and smooths noisy updates.
- **Second-Moment Estimate via β_2.** By keeping an exponential moving average of squared gradients, ADAM dynamically moderates the update where gradients exhibit high variance, preventing overshoot. In directions where the variance is consistently low, it can safely take larger steps, accelerating convergence.

1.3.2 Multi-LR Experiments and Results

We trained our 1-3-1 network for 5,000 epochs using the ADAM optimizer, comparing three learning rates: $\eta \in \{0.01,\ 0.001,\ 0.0001\}$

In Figure 1.4, the green curve $(\eta = 0.001)$ clearly tracks the noisy cubic function more closely than the other two. The red curve $(\eta = 0.01)$ fits well in many regions but can slightly oscillate in a few intervals. The blue curve $(\eta = \ 0.0001)$ fails to fit.

Learning Rate	Final MSE (at 30,000 epochs)
0.01	0.000785 (oscillates between 5×10^{-4} and 4×10^{-3}
0.001	0.000402 (the most stable)
0.0001	0.011995 (extremely slow descent)

Table 1.2: Final MSE for ADAM across three different learning rates.

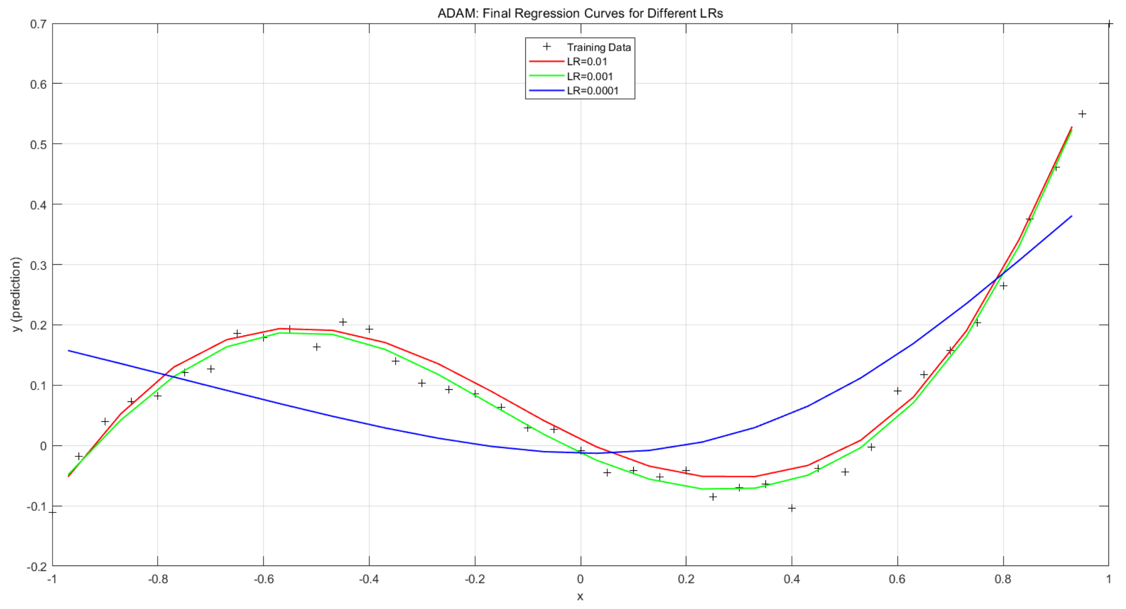


Figure 1.4: Regression curves from ADAM for LRs 0.01 (red), 0.001 (green), and 0.0001 (blue). Black + indicates the training samples.

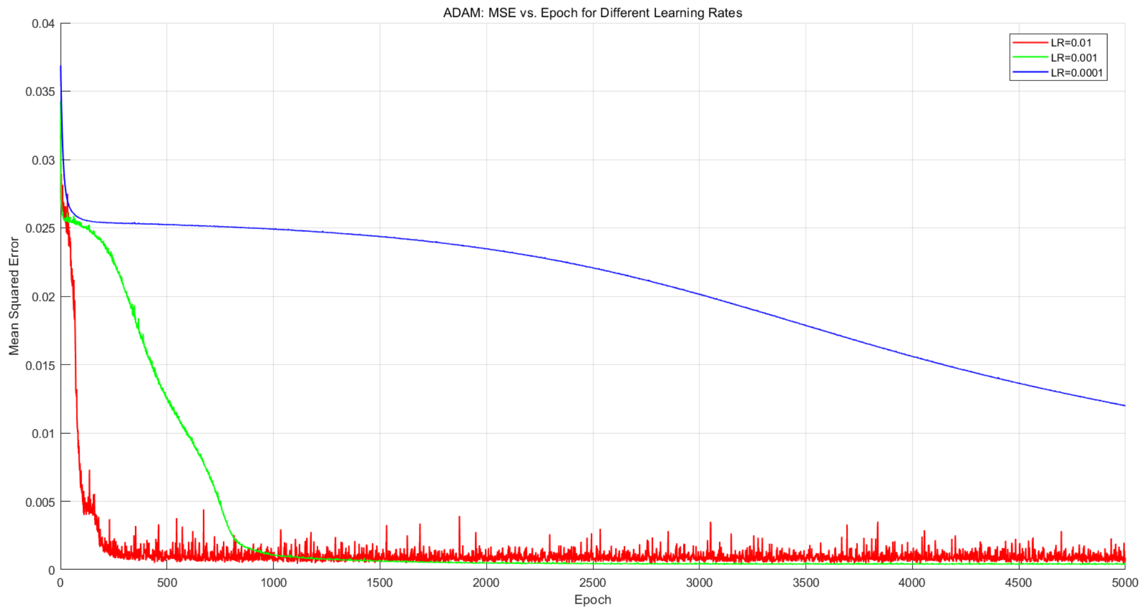


Figure 1.5: MSE vs. epoch for ADAM with three LRs. $\eta = 0.01$ (red), $\eta = 0.001$ (green) $\eta = 0.0001$ (blue)

In figure 1.5, We see that $\eta = 0.01$ reduces the error swiftly but exhibits significant fluctuations in the latter epochs. In contrast, $\eta = 0.0001$ is so small that the network barely improves after 5,000 epochs, remaining at a high MSE. Overall, $\eta = 0.001$ has the best balance of convergence speed and stability, consistently yielding the lowest final error without significant oscillations.

1.3.3 Comparison of SGD and ADAM Results

In the previous sections, we trained the same 1–3–1 network using two different algorithms: SGD (with three candidate learning rates) and ADAM (also with three candidate rates). Below are our key observations based on the logs and final outcomes:

- **Learning-Rate Sensitivity.** In principle, ADAM is often less sensitive to large learning rates because it adaptively scales each update; however, we found a larger fluctuation for ADAM at $\eta = 0.01$ (ranging from 0.000475 up to 0.00392) than for SGD at $\eta = 0.1$ (0.00036 to 0.00048). To quantify this, we use the *relative fluctuation formula:*

$$\frac{\mathrm{MSE}_{\max} - \mathrm{MSE}_{\min}}{\mathrm{MSE}_{\min}} \times 100\% \tag{24}$$

giving about 725% for ADAM vs. 32% for SGD. Two reasons explain the apparent paradox: first, ADAM rapidly drives the MSE to a very low minimum, so any subsequent jump appears huge in percentage terms; second, ADAM is only run for 5,000 epochs, so short-term momentum overshoot is more visible. Despite bigger swings in this scenario, ADAM generally reduces the risk of outright divergence by adaptively shrinking updates in high-variance directions.

- **Convergence Speed and final MSE.** For this small 1-3-1 problem, SGD with $\eta = 0.01$ eventually reached a low MSE (3.56×10^{-4}) but required 30,000 epochs. *ADAM*, by contrast, had a slightly higher error rate $(\sim 4.0 \times 10^{-4})$ in just 5,000 epochs with $\eta = 0.001$. This suggests ADAM can achieve faster early convergence with similar final MSE.

Overall, *ADAM* generally converged faster and reached a good error in fewer epochs, while SGD needed more epochs to match similar MSE levels. Both optimizers can succeed on this small problem, provided the user picks an LR that is neither too large nor too small.

1.4 Using MATLAB's Neural Network Toolbox

1.4.1 Network Setup

Here, we use MATLAB's built-in `feedforwardnet` with the `trainlm` (Levenberg–Marquardt) solver to explore multiple network structures and activation functions. Specifically, we test six networks:

1. A: 1-3-1 with `tansig` → `purelin`,
2. B: 1-3-1 with `logsig` → `purelin`,
3. C: 1-5-1 with `ReLU` → `purelin`,
4. D: 2-layer [8, 8] with `ReLU` → `ReLU` → `purelin`,
5. E: 1-5-1 with `tansig` → `purelin`,
6. F: 1-5-1 with `logsig` → `purelin`.

All networks share the same data from earlier parts:

$$x \in [-1, 1], \quad d = 0.8\,x^3 + 0.3\,x^2 - 0.4\,x + \text{noise}, \tag{25}$$

sampled every 0.05, plus a test set for final predictions. Each network uses a maximum of 30,000 epochs, though it typically stops early once the solver's stopping criteria are met. We record each network's final mean-squared error (MSE), the correlation R, the actual number of epochs used, and the predicted outputs on the test set.

1.4.2 Training Results

Table 3 gives the final MSE, R –value, and epoch counts for Networks A-F. Figure 1.6 shows their test-set curves.

Net	Structure/Activation	MSE	Epochs	R
A	1-3-1 (`tansig` → `purelin`)	0.000740	10	0.9878
B	1-3-1 (`logsig` → `purelin`)	0.000599	10	0.9921
C	1-5-1 (ReLU → `purelin`)	0.041084	5	-0.3504
D	1-8-8-1 (ReLU → ReLU → `purelin`)	0.001441	12	0.9802
E	1-5-1 (`tansig` → `purelin`)	0.000643	12	0.9897
F	1-5-1 (`logsig` → `purelin`)	0.000398	29	0.9939

Table 3: Final training performance of six networks.

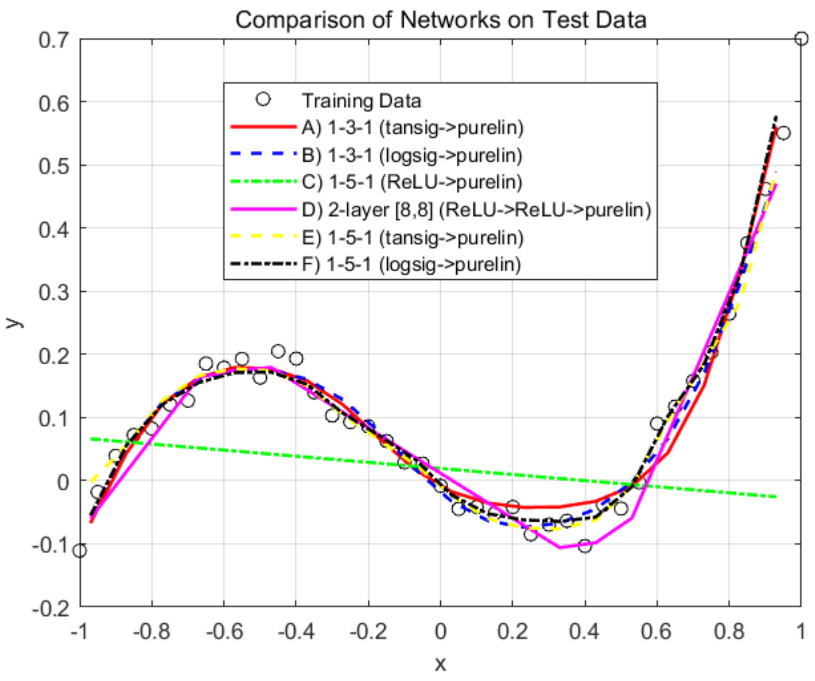


Figure 1.6: Test predictions of A-F against training data.

1.4.3 Analysis and Observations

Overall, the tansig or logsig networks (A/B/E/F) achieve better MSE than the ReLU networks (C/D). In particular, B and F yield the lowest final errors, indicating that logsig performs the best.

Comparing 1-3-1 vs. 1-5-1.

- **E vs. A:** Both use `tansig`, but E has 5 hidden nodes. E's MSE is 0.000643 (slightly better than A at 0.000740), but E requires more epochs (12 vs. 10). So, adding more hidden neurons can improve accuracy slightly, though it may cost extra epochs.
- **F vs. B:** Both use `logsig`, but F has 5 hidden nodes. F obtains the best MSE overall (0.000398) but takes 29 epochs. B achieves 0.000599 in only 10 epochs. Again, adding more hidden neurons can slightly improve accuracy, but cost more extra epochs for `logsig`.

ReLU Networks (C/D). Notice that both C and D used ReLU, and both ended up with unsatisfactory MSEs (especially C). A single hidden layer with ReLU can fail to capture the right outputs. Although D's two-layer approach helped somewhat (0.001441), it's still behind the best `tansig` or `logsig` results. The improvement from C to D indicates that ReLU-based networks require more careful initialization (e.g. higher layer/node counts) to avoid high errors.

Comparison With Parts (a) and (b). In part (a) and (b), we manually derived backpropagation for a 1–3–1 architecture and used SGD at $\eta = 0.01$ and required 30,000 epochs to reach 0.000356. Although that final error is slightly lower than most results here, it demanded substantially more updates. In contrast, the Toolbox's `trainlm` often achieves MSE smaller than 10^{-3} in under 20 epochs, highlighting its computational efficiency and potential advantage over manual gradient-descent approaches.

Conclusion

The `tansig` and `logsig` networks generally fit this small cubic regression better under the current settings. ReLU (C/D) lags behind, unless we do more layers and nodes. Among 1–5–1 vs. 1–3–1, the larger hidden layer can reduce MSE further, but it sometimes needs more epochs to fully converge (e.g. F vs. B).

1.5 ReLU Approximation Sketch

In this final subtask, we illustrate how to manually construct a ReLU-based network to approximate the noisy cubic data. Following the ideas in slides 45–46 (week 2 notes), we create a piecewise linear approximation by assigning different ReLU units to different regions of the input $x \in [-1,1]$.

1.5.1 Network Structure & Parameters

We consider a 1-6-1 network, one scalar input (x, plus an internal bias merged into each hidden neuron), six ReLU activation nodes in the hidden layer, and one *linear* output node. For notational simplicity, we label each hidden neuron $i = 1, \dots, 6$ with parameters $w_1(i), b_1(i)$ so that its net input is

$$v_i^{(1)}(x) = w_1(i)x + b_1(i), \quad h_i(x) = \max\left\{0, v_i^{(1)}(x)\right\}. \tag{26}$$

At the final layer, the output is formed by summing each $h_i(x)$ with a sign $w_2(i) \in \{-1, +1\}$ and a final bias b_2:

$$y(x) = \sum_{i=1}^{6} [w_2(i) \max\{0, w_1(i)x + b_1(i)\}] + b_2. \tag{27}$$

We force each $w_2(i) \in \{+1, -1\}$ to replicate the approach in slides 45-46.

1.5.2 Manual Setting of Each ReLU Node

Our step-by-step procedure is:

1. **Initialize all parameters to 0.** That is, set every $w_1(i)$, $b_1(i)$, and $w_2(i)$ to zero, and $b_2 = 0$
2. **Configure Node 1.** Adjust $w_1(1), b_1(1), w_2(1)$ so that its ReLU "kink" corresponds to the first sub-interval of x and aligns in slope with the data. Continue tweaking until ReLU(1)'s output is roughly parallel to the data segment.
3. **Find $\boldsymbol{b_2}$** Because the ReLU(1) curve still sits below the data, measure the vertical offset at $x = -0.65$. Set b_2 equal to that offset (e.g. +0.185398) so the ReLU(1) curve now overlaps the data in that sub-interval.

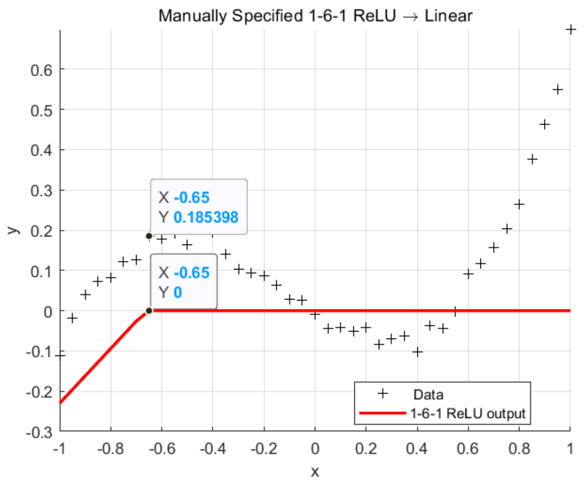


Figure 1.7: At x = −0.65, vertical offset between ReLU(1) and data is 0.185398

4. **Repeat for Nodes 2 to 6.** With Node 1 fixed and bias b_2 determined, add each new ReLU ramp to handle subsequent sub-intervals. For Node2, set $w_1(2), b_1(2), w_2(2)$ to align in slope with the data, and so on until Node6.

1.5.3 Results

Parameter Summary. Table 4 shows the final parameters we used for this 1-6-1 ReLU network.

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
$w_1(i)$	-0.680	0.150	0.580	0.610	0.521	0.700
$b_1(i)$	-0.450	0.095	0.286	-0.150	-0.200	-0.450
$w_2(i)$	-1	+1	-1	+1	+1	+1

Final output bias $b_2 = 0.185398$

Table 4: Manually chosen parameters for the 1-6-1 ReLU network.

Overall Fit. By combining six linear ramps (Figure 1.8), we capture the major "turning points" of the cubic curve across [−1, 1]. Minor differences arise because we piecewise-approximate a continuous function and also have noisy data.

Individual Hidden Nodes. Figure 1.9 shows each node's $\mathrm{ReLU}\big(w_1(i)\,x + b_1(i)\big)$. Then, because $w_2(i) \in \{\pm 1\}$, the final output is

$$y(x) \;=\; b_2 \;-\; \mathrm{ReLU}_1(x) \;+\; \mathrm{ReLU}_2(x) \;-\; \mathrm{ReLU}_3(x) \;+\; \mathrm{ReLU}_4(x) \;+\; \mathrm{ReLU}_5(x) \;+\; \mathrm{ReLU}_6(x) \qquad (28)$$

Network Diagram. Finally, Figure 1.10 Shows the 1-6-1 network structure,

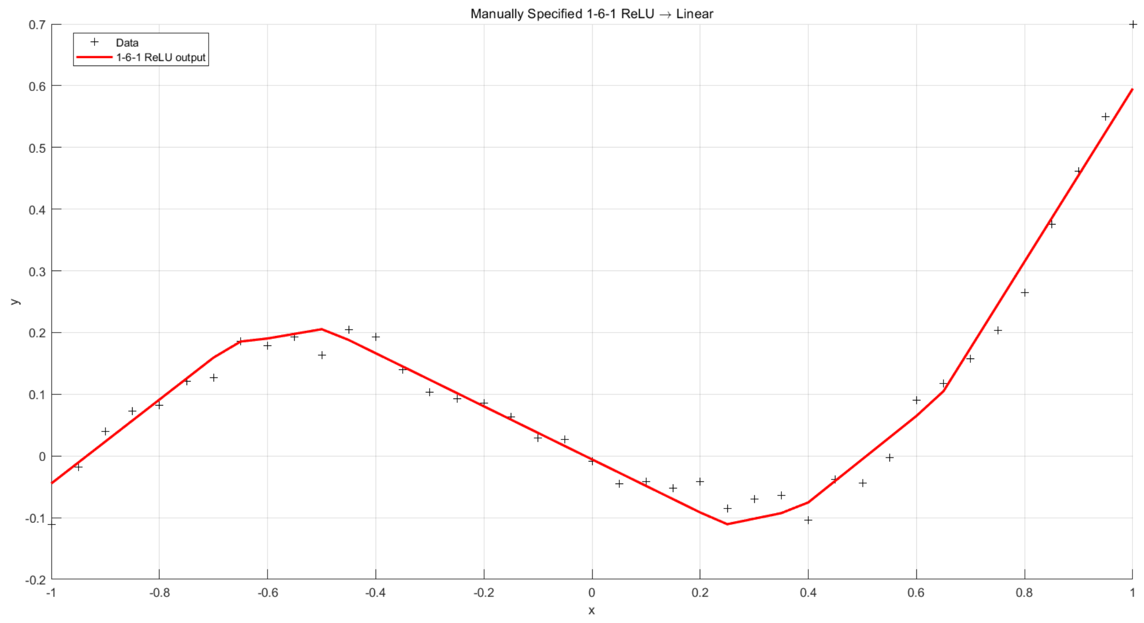


Figure 1.8: Manually specified 1-6-1 ReLU approximation vs. the noisy cubic samples.

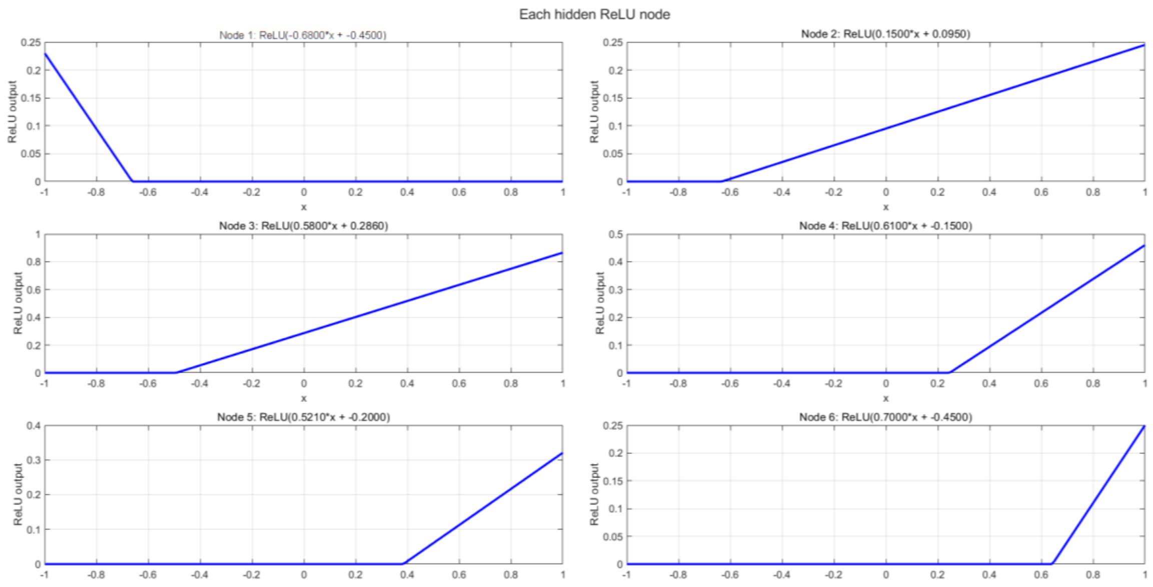


Figure 1.9: Each hidden node's $\mathrm{ReLU}\big(w_1(i)\,x + b_1(i)\big)$ shape.

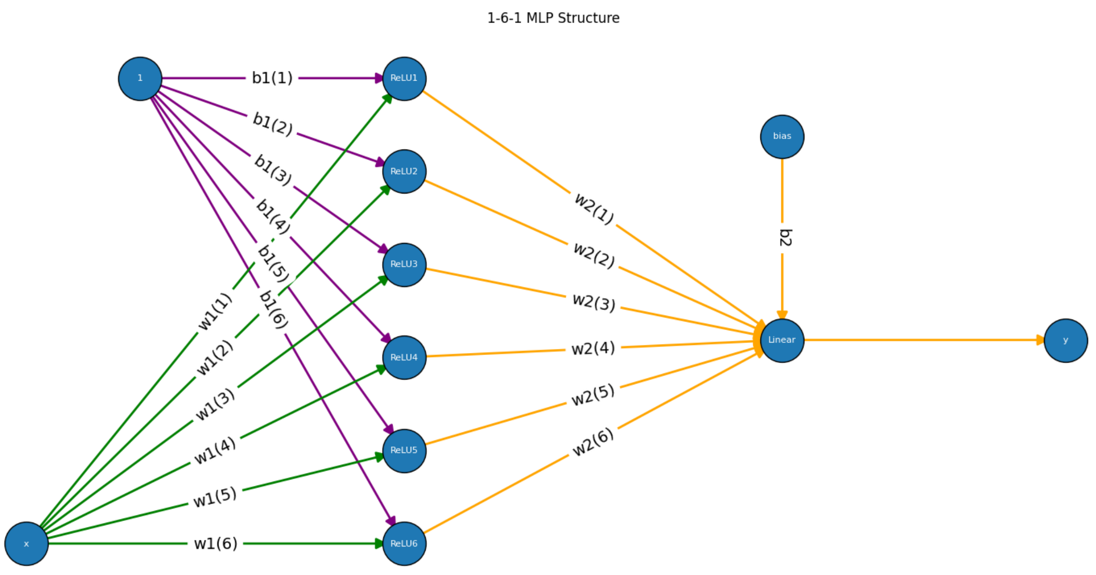
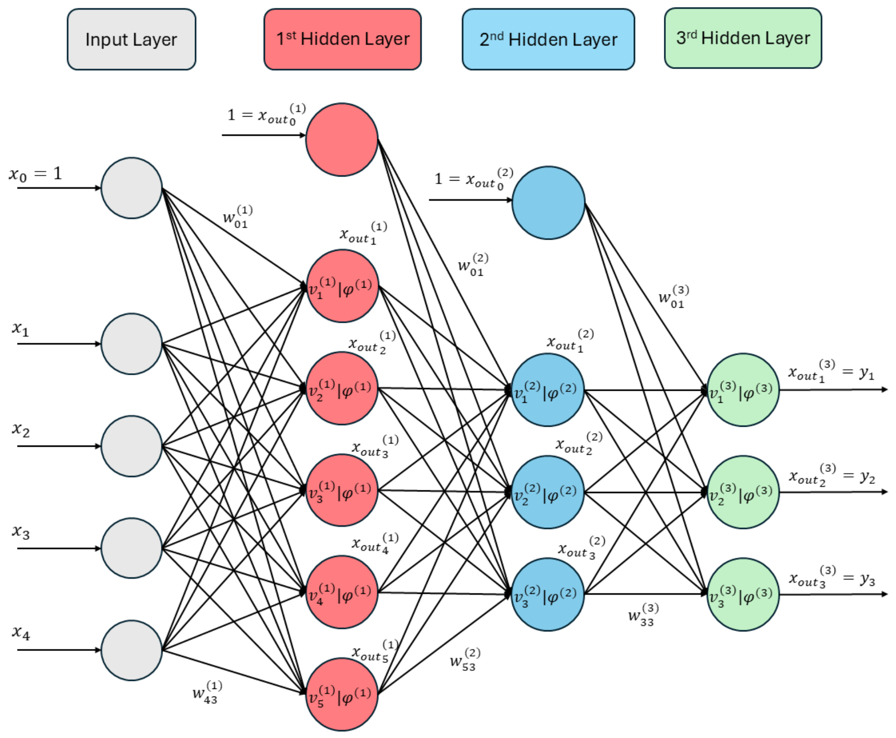


Figure 1.10: Structure of the 1-6-1 ReLU network.

Task 2: Classification

2.1 Deriving backpropagation algorithm using stochastic gradient descent



The instantaneous squared error for all outputs for one instance of input data:

$$L = \frac{1}{2}(d_1 - y_1)^2 + \frac{1}{2}(d_2 - y_2)^2 + \frac{1}{2}(d_3 - y_3)^2 \tag{29}$$

Rewriting the weighted sum in terms of output layer weights:

$$L = \frac{1}{2}\left(d_1 - \varphi^{(3)}\left(v_1^{(3)}\right)\right)^2 + \frac{1}{2}\left(d_2 - \varphi^{(3)}\left(v_2^{(3)}\right)\right)^2 + \frac{1}{2}\left(d_3 - \varphi^{(3)}\left(v_3^{(3)}\right)\right)^2 \tag{30}$$

$$v_1^{(3)} = w_{01}^{(3)} + w_{11}^{(3)} x_{out_1}^{(2)} + w_{21}^{(3)} x_{out_2}^{(2)} + w_{31}^{(3)} x_{out_3}^{(2)} \tag{31}$$

$$v_2^{(3)} = w_{02}^{(3)} + w_{12}^{(3)} x_{out_1}^{(2)} + w_{22}^{(3)} x_{out_2}^{(2)} + w_{32}^{(3)} x_{out_3}^{(2)} \tag{32}$$

$$v_3^{(3)} = w_{03}^{(3)} + w_{13}^{(3)} x_{out_1}^{(2)} + w_{23}^{(3)} x_{out_2}^{(2)} + w_{33}^{(3)} x_{out_3}^{(2)} \tag{33}$$

Using $w_{21}^{(3)}$ as an example to find an expression for the new weights of output layer in terms of the old weights:

$$
\begin{aligned}
w_{21\,new}^{(3)} &= w_{21\,old}^{(3)} - \eta^{(3)} \frac{\partial L}{\partial w_{21}^{(3)}} \\
&= w_{21\,old}^{(3)} - \eta^{(3)} \frac{\partial L}{\partial v_1^{(3)}} \cdot \frac{\partial v_1^{(3)}}{\partial w_{21}^{(3)}} \qquad (34)
\end{aligned}
$$

$$
\frac{\partial v_1^{(3)}}{\partial w_{21}^{(3)}} = \frac{\partial}{\partial w_{21}^{(3)}} \left[w_{01}^{(3)} + w_{11}^{(3)} x_{out_1}^{(2)} + w_{21}^{(3)} x_{out_2}^{(2)} + w_{31}^{(3)} x_{out_3}^{(2)} \right] = x_{out_2}^{(2)} \qquad (35)
$$

Using tanh for classification, $\frac{\partial \varphi}{\partial v} = 1 - \varphi^2(v)$ for $\varphi(v) = \tanh(v)$:

$$
\begin{aligned}
\frac{\partial L}{\partial v_1^{(3)}} = \frac{\partial L}{\partial \varphi^{(3)}\left(v_1^{(3)}\right)} \cdot \frac{\partial \varphi^{(3)}\left(v_1^{(3)}\right)}{\partial v_1^{(3)}} &= -\left(d_1 - \varphi^{(3)}\left(v_1^{(3)}\right)\right)\left(1 - {\varphi^{(3)}}^2\left(v_1^{(3)}\right)\right) \\
&= -\left(d_1 - \varphi^{(3)}\left(v_1^{(3)}\right)\right)\left(1 - \tanh^2\left(v_1^{(3)}\right)\right) \\
&= -\delta_1^{(3)} \qquad (36)
\end{aligned}
$$

$$
\therefore w_{21\,new}^{(3)} = w_{21\,old}^{(3)} + \eta^{(3)} \cdot \delta_1^{(3)} \cdot x_{out_2}^{(2)} \qquad (37)
$$

Therefore, the general weight equation for $0 \leq i \leq 3$ and $1 \leq j \leq 3$ can be obtained, with the bias at $i = 0$ as a fixed input of 1.

$$
w_{ij\,new}^{(3)} = w_{ij\,old}^{(3)} + \eta^{(3)} \cdot \delta_j^{(3)} \cdot x_{out_i}^{(2)} \qquad (38)
$$

$$
\text{where } \delta_j^{(3)} = \left(d_j - \varphi^{(3)}\left(v_j^{(3)}\right)\right)\left(1 - \tanh^2\left(v_j^{(3)}\right)\right)
$$

For the 2nd hidden layer, the weighted sum can also be written in terms of weights of the same layer:

$$
x_{out_1}^{(2)} = \varphi^{(2)}\left(v_1^{(2)}\right) \qquad (39)
$$

$$
\text{where } v_1^{(2)} = w_{01}^{(2)} + w_{11}^{(2)} x_{out_1}^{(1)} + w_{21}^{(2)} x_{out_2}^{(1)} + w_{31}^{(2)} x_{out_3}^{(1)} + w_{41}^{(2)} x_{out_4}^{(1)} + w_{51}^{(2)} x_{out_5}^{(1)}
$$

$$
x_{out_2}^{(2)} = \varphi^{(2)}\left(v_2^{(2)}\right) \qquad (40)
$$

$$
\text{where } v_2^{(2)} = w_{02}^{(2)} + w_{12}^{(2)} x_{out_1}^{(1)} + w_{22}^{(2)} x_{out_2}^{(1)} + w_{32}^{(2)} x_{out_3}^{(1)} + w_{42}^{(2)} x_{out_4}^{(1)} + w_{52}^{(2)} x_{out_5}^{(1)}
$$

$$
x_{out_3}^{(2)} = \varphi^{(2)}\left(v_3^{(2)}\right) \qquad (41)
$$

$$
\text{where } v_3^{(2)} = w_{03}^{(2)} + w_{13}^{(2)} x_{out_1}^{(1)} + w_{23}^{(2)} x_{out_2}^{(1)} + w_{33}^{(2)} x_{out_3}^{(1)} + w_{43}^{(2)} x_{out_4}^{(1)} + w_{53}^{(2)} x_{out_5}^{(1)}
$$

Using $w_{21}^{(2)}$ as example, a general expression for the new weights of 2nd hidden layer can be found:

$$
w_{21\,new}^{(2)} = w_{21\,old}^{(2)} - \eta^{(2)} \frac{\partial L}{\partial w_{21}^{(2)}} \qquad (42)
$$

$$
\begin{aligned}
\frac{\partial L}{\partial w_{21}^{(2)}} &= \left\{\frac{\partial L}{\partial v_1^{(3)}} \cdot \frac{\partial v_1^{(3)}}{\partial x_{out_1}^{(2)}} \cdot \frac{\partial x_{out_1}^{(2)}}{\partial v_1^{(2)}} \cdot \frac{\partial v_1^{(2)}}{\partial w_{21}^{(2)}}\right\} + \left\{\frac{\partial L}{\partial v_2^{(3)}} \cdot \frac{\partial v_2^{(3)}}{\partial x_{out_1}^{(2)}} \cdot \frac{\partial x_{out_1}^{(2)}}{\partial v_1^{(2)}} \cdot \frac{\partial v_1^{(2)}}{\partial w_{21}^{(2)}}\right\} \\
&\quad + \left\{\frac{\partial L}{\partial v_3^{(3)}} \cdot \frac{\partial v_3^{(3)}}{\partial x_{out_1}^{(2)}} \cdot \frac{\partial x_{out_1}^{(2)}}{\partial v_1^{(2)}} \cdot \frac{\partial v_1^{(2)}}{\partial w_{21}^{(2)}}\right\} \\
&= \left(\frac{\partial L}{\partial v_1^{(3)}} \cdot \frac{\partial v_1^{(3)}}{\partial x_{out_1}^{(2)}} + \frac{\partial L}{\partial v_2^{(3)}} \cdot \frac{\partial v_2^{(3)}}{\partial x_{out_1}^{(2)}} + \frac{\partial L}{\partial v_3^{(3)}} \cdot \frac{\partial v_3^{3}}{\partial x_{out_1}^{(2)}}\right) \cdot \frac{\partial x_{out_1}^{(2)}}{\partial v_1^{(2)}} \cdot \frac{\partial v_1^{(2)}}{\partial w_{21}^{(2)}} \qquad (43)
\end{aligned}
$$

The first terms have already been calculated in the output layer:

$$
\frac{\partial L}{\partial v_1^{(3)}} = -\delta_1^{(3)} \quad \frac{\partial L}{\partial v_2^{(3)}} = -\delta_2^{(3)} \quad \frac{\partial L}{\partial v_3^{(3)}} = -\delta_3^{(3)}
$$

For the second term:

$$\frac{\partial v_1^{(3)}}{\partial x_{out}{}_1^{(2)}} = w_{11}^{(3)} \quad \frac{\partial v_2^{(3)}}{\partial x_{out}{}_1^{(2)}} = w_{12}^{(3)} \quad \frac{\partial v_3^{(3)}}{\partial x_{out}{}_1^{(2)}} = w_{13}^{(3)}$$

Given that $\frac{\partial \varphi}{\partial v} = 1 - \varphi^2(v)$ for $\varphi(v) = \tanh(v)$, the final two terms are:

$$\frac{\partial x_{out}{}_1^{(2)}}{\partial v_1^{(2)}} = \frac{\partial x_{out}{}_1^{(2)}}{\partial \varphi^{(2)}\left(v_1^{(2)}\right)} \cdot \frac{\partial \varphi^{(2)}\left(v_1^{(2)}\right)}{\partial v_1^{(2)}} = 1 \cdot \left(1 - \varphi^{(2)^2}\left(v_1^{(2)}\right)\right) = 1 - \tanh^2\left(v_1^{(2)}\right) \tag{44}$$

$$\frac{\partial v_1^{(2)}}{\partial w_{21}^{(2)}} = x_{out}{}_2^{(1)} \tag{45}$$

$$\therefore \frac{\partial L}{\partial w_{21}^{(2)}} = -\left(\delta_1^{(3)} \cdot w_{11}^{(3)} + \delta_2^{(3)} \cdot w_{12}^{(3)} + \delta_3^{(3)} \cdot w_{13}^{(3)}\right) \cdot \left(1 - \tanh^2\left(v_1^{(2)}\right)\right) \cdot x_{out}{}_2^{(1)} \tag{46}$$

The new weight:

$$w_{21\,new}^{(2)} = w_{21\,old}^{(2)} + \eta^{(2)} \cdot \left(\delta_1^{(3)} \cdot w_{11}^{(3)} + \delta_2^{(3)} \cdot w_{12}^{(3)} + \delta_3^{(3)} \cdot w_{13}^{(3)}\right) \cdot \left(1 - \tanh^2\left(v_1^{(2)}\right)\right) \cdot x_{out}{}_2^{(1)} \tag{47}$$

The general weight of 2nd hidden layer for $0 \le i \le 5$ and $1 \le j \le 3$:

$$w_{ij\,new}^{(2)} = w_{ij\,old}^{(2)} + \eta^{(2)} \cdot \delta_j^{(2)} \cdot x_{out}{}_i^{(1)} \tag{48}$$

$$\text{where } \delta_j^{(2)} = \left(\delta_1^{(3)} \cdot w_{j1}^{(3)} + \delta_2^{(3)} \cdot w_{j2}^{(3)} + \delta_3^{(3)} \cdot w_{j3}^{(3)}\right)\left(1 - \tanh^2\left(v_j^{(2)}\right)\right)$$

Generalising the weight update for all hidden and output layers:

$$w_{ij\,new}^{(s)} = w_{ij\,old}^{(s)} + \eta^{(s)} \cdot \delta_j^{(s)} \cdot x_{out}{}_i^{(s-1)} \tag{49}$$

Finally, the general weight of 1st hidden layer for $0 \le i \le 4 \text{ and } 1 \le j \le 5$:

$$w_{ij\,new}^{(1)} = w_{ij\,old}^{(1)} + \eta^{(1)} \cdot \delta_j^{(1)} \cdot x_{out}{}_i^{(0)} \tag{50}$$

$$\text{where } \delta_j^{(1)} = \left(\delta_1^{(2)} \cdot w_{j1}^{(2)} + \delta_2^{(2)} \cdot w_{j2}^{(2)} + \delta_3^{(2)} \cdot w_{j3}^{(2)}\right) \cdot \left(1 - \tanh^2\left(v_j^{(1)}\right)\right) \text{ and } x_{out}{}_i^{(0)} = x_i$$

2.2 Stochastic Gradient Descent (SGD) Baseline Experiment

2.2.1 Experimental Setup for Baseline SGD

The baseline SGD experiment was configured with the following parameters to train a 4-5-3-3 MLP for Iris classification:

- **Activation Functions:** The hyperbolic tangent function (tanh) was utilized as the activation function for both hidden layers and the output layer. This choice was made due to tanh's non-linear nature and its output range of -1 to 1, which is well-suited for classification tasks and compatible with the one-hot encoding scheme employed for class labels.

- **Learning Rate:** A learning rate of 0.01 was selected for the baseline experiment. This value was empirically chosen as a starting point, with the expectation of further investigation into learning rate sensitivity in subsequent experiments (Section 2.3.1).

- **Number of Epochs:** The network was trained for 1000 epochs.

- **Weight Initialization:** Xavier initialization was employed to initialize the weights of the network. This method, which draws weights from a Gaussian distribution scaled by the fan-in and fan-out of each layer, helps to mitigate the vanishing and exploding gradient problems, promoting more stable and efficient training, especially in the initial epochs. Biases were initialized to zero.

- **Data Pre-processing:** Z-score normalization was applied to the Iris feature data prior to training. This standardization process, which centers the data around zero mean and scales it to unit variance, ensures that all features contribute equally to the learning process and prevents features with larger scales from dominating the network's learning.
- **Training and Validation Data Split:** The Iris dataset was divided into training and validation sets using a 70:30 split ratio. A fixed random seed (seed = 42) was used to ensure reproducibility of the data shuffling and split across experiments. The training set was used to update the network's parameters, while the validation set was used to evaluate the generalization performance of the trained model on unseen data.

2.2.2 Results of Baseline SGD Training

The baseline SGD training process yielded the following results after 1000 epochs:

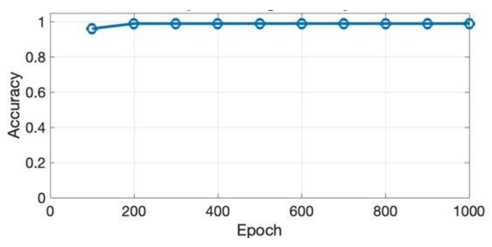


Figure 2.1: Training Loss Curve for Baseline SGD

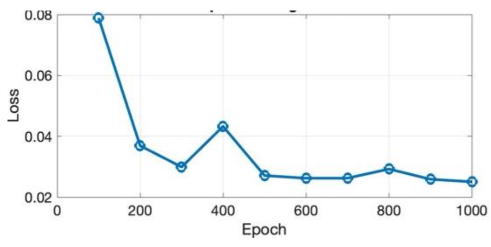


Figure 2.2: Training Accuracy Curve for Baseline SGD

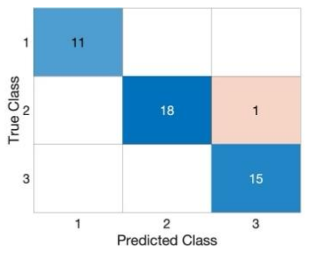


Figure 2.3: Confusion Matrix for Baseline SGD

2.2.3 Discussion of Baseline SGD Performance

The results from the baseline SGD training demonstrate that the manually implemented MLP network successfully learned to classify the Iris dataset with high accuracy. The training accuracy of 99.05% and the validation accuracy of 97.78% are both indicative of strong performance and good generalization.

The learning curves (Figure 2.1 and 2.2) reveal a typical pattern of neural network training: a rapid initial improvement in accuracy and decrease in loss during the early epochs, followed by a more gradual fine-tuning phase as the network approaches convergence.

The validation accuracy, being close to the training accuracy, suggests that the model is generalizing well to unseen data and is not significantly overfitting the training set. The small gap between training and validation accuracy (approximately 1.27%) is expected and acceptable, indicating robust performance on data the network has not been directly trained on.

Analysis of the confusion matrix (Figure 2.3) reveals excellent class-wise performance. Notably, the Setosa and Virginica classes were perfectly classified in the validation set, with 100% accuracy for both. The Versicolor class also achieved high accuracy, with only a single instance misclassified as Virginica. The overall high accuracy and the detailed insights from the confusion matrix validate the successful implementation of the backpropagation algorithm and the effectiveness of the chosen network architecture and training parameters for the Iris classification task.

2.3 In-depth Analysis of SGD Training

2.3.1 Learning Rate Sensitivity with SGD

The learning rate is a critical hyperparameter in gradient descent algorithms, controlling the step size taken during weight updates. To investigate the sensitivity of the SGD-trained MLP to the learning rate, experiments were performed using a range of learning rates: 0.1, 0.01, 0.001, and 0.0001. All other experimental settings, including network architecture (4-5-3-3), activation function (tanh), weight initialization (Xavier), and data normalization (z-score), were kept constant to isolate the effect of the learning rate.

Table 2.1: Validation Accuracy Comparison for Different Learning Rates (SGD)

Learning Rate	0.1	0.01	0.001	0.0001
Validation Accuracy	97.78%	97.78%	97.78%	95.56%

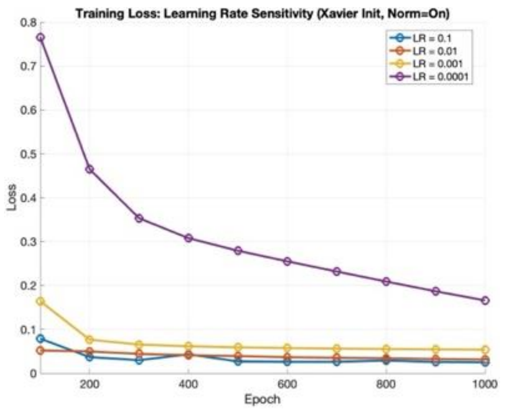


Figure 2.4: Training Loss vs. Epoch for Different Learning Rates

The learning rate experiments revealed a non-trivial sensitivity of SGD performance. While a high learning rate (0.1) enabled rapid initial loss reduction, it led to unstable training and suboptimal final accuracy. Conversely, a very low learning rate (0.0001) resulted in exceedingly slow learning and underperformance. A learning rate of 0.01 emerged as optimal, striking a balance between convergence speed and stability, and achieving the highest validation accuracy. These findings, supported by the learning curves (Figures 2.4 and 2.5) and validation accuracies (Table 2.1), highlight the crucial role of careful learning rate selection for effective SGD training, even on a relatively simple dataset like Iris.

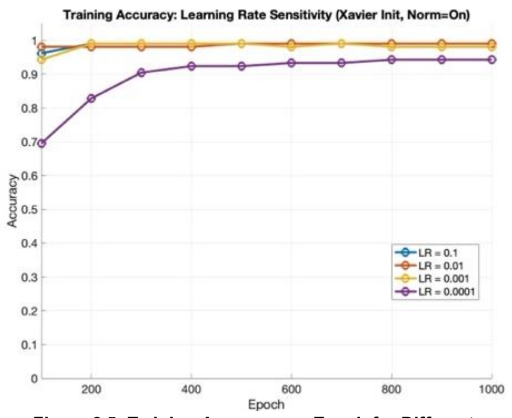


Figure 2.5: Training Accuracy vs. Epoch for Different

2.3.2 Impact of Normalization on SGD Training

Data normalization is a common pre-processing technique used to improve the training of neural networks. To evaluate the impact of z-score normalization on the SGD-trained MLP, experiments were conducted comparing training with and without normalization. All other settings, including network architecture (4-5-3-3), activation function (tanh), weight initialization (Xavier), and learning rate (0.01) were kept constant.

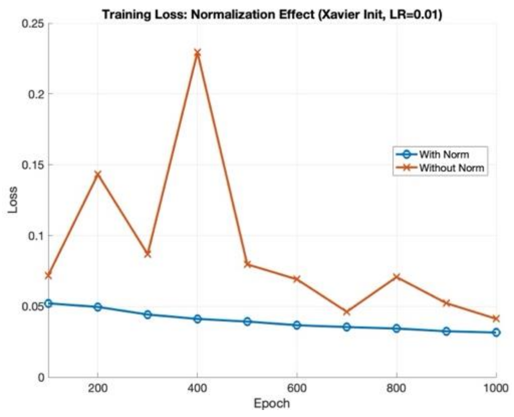


Figure 2.6: Training Loss vs. Epoch (With vs. Without Normalization - SGD)

Table 2.2: Validation Accuracy Comparison (With vs. Without Normalization - SGD)

Normalization	With (Z-Score)	Without
Validation Accuracy	97.78%	97.78%

The influence of z-score normalization on SGD training is demonstrably positive, as evidenced by the learning curves in Figures 2.6 and 2.7. When training is conducted with normalized features, the training loss curve shows a smoother and more consistent downward trend compared to training without normalization. Notably, the sharp spike in loss observed around epoch 400 in the "Without Norm" condition (Figure 2.6) is absent when normalization is applied, suggesting that normalization contributes to training stability. The accuracy curves in Figure 2.7 further support this observation, with normalized training exhibiting less fluctuation. While the validation accuracy (Table 2.2) for both conditions remains high, the consistently smoother learning curves and the avoidance of loss spikes with normalization indicate that z-score normalization enhances the robustness and stability of the SGD optimization process for this MLP and dataset.

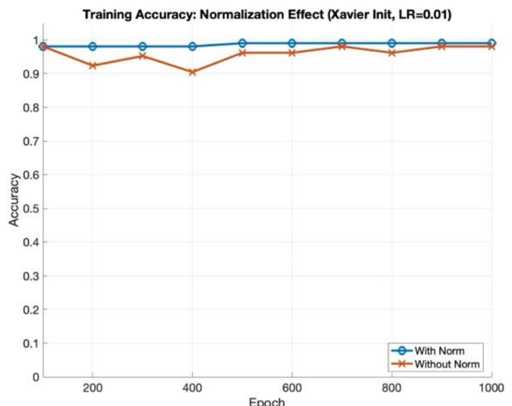


Figure 2.7: Training Accuracy vs. Epoch (With vs. Without Normalization - SGD)

2.3.3 Effect of Weight Initialization (Xavier vs. Random) with SGD

Weight initialization plays a crucial role in the training of deep neural networks. Poor initialization can lead to slow convergence or even prevent learning altogether, often due to vanishing or exploding gradients. To investigate the effect of weight initialization on the SGD-trained MLP, experiments were conducted comparing Xavier initialization with a simpler random initialization strategy. In the random initialization, weights were drawn from a standard normal distribution and then scaled by a small factor (0.01) to keep the initial weights close to zero. All other experimental settings, including network architecture (4-5-3-3), activation function (tanh), data normalization (z-score), and learning rate (0.01), were kept constant.

Table 3.3: Validation Accuracy Comparison (Xavier vs. Random Initialization - SGD)

Weight Initialization	Xavier	Random (Small)
Validation Accuracy	97.78%	97.78%

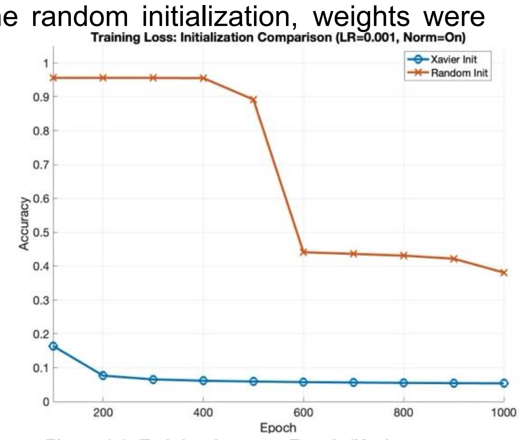


Figure 2.8: Training Loss vs. Epoch (Xavier vs. Random Initialization - SGD)

Experiments comparing weight initialization strategies reveal a critical advantage of Xavier initialization over random initialization for SGD training. As shown in Figure 2.8 and Figure 2.9, Xavier initialization consistently enables stable and effective learning, while random initialization leads to highly unstable and unreliable training dynamics, often resulting in failure to converge or drastically reduced performance. Xavier initialization's design, aimed at preventing vanishing or exploding gradients, demonstrably facilitates robust training and reliable high accuracy. In contrast, random initialization proves to be a less dependable strategy, highlighting the crucial role of informed weight initialization for ensuring stable and successful neural network training, particularly as network complexity increases.

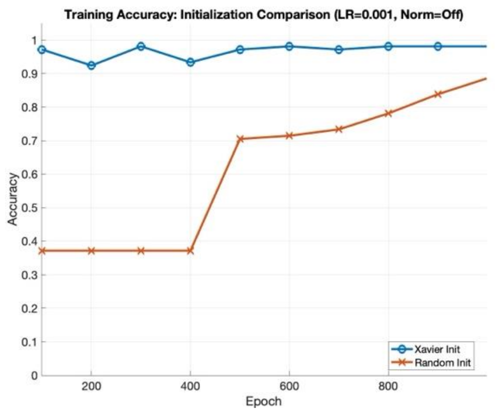


Figure 2.9: Training Accuracy vs. Epoch (Xavier vs. Random Initialization - SGD)

2.4 ADAM Optimization Algorithm

2.4.1 Key Parameters of ADAM Optimizer

The algorithm is already explained in 1.3.1. The baseline hyperparameters used for the experiments are as shown in Table 2.4.

Table 2.4: Baseline hyperparameters for ADAM optimizer

Learning Rate	Batch Size	β_1	β_2	ϵ
0.01	16	0.9	0.999	10^{-8}

2.4.2 Comparison of ADAM and SGD Performance

To directly compare the performance of the ADAM optimizer with the previously implemented Stochastic Gradient Descent (SGD), experiments were conducted training the 4-5-3-3 MLP using both optimization algorithms. To ensure a fair comparison, all other experimental settings, including network architecture, activation function (tanh), weight initialization (Xavier), data normalization (z-score), and learning rate (0.01), were kept consistent.

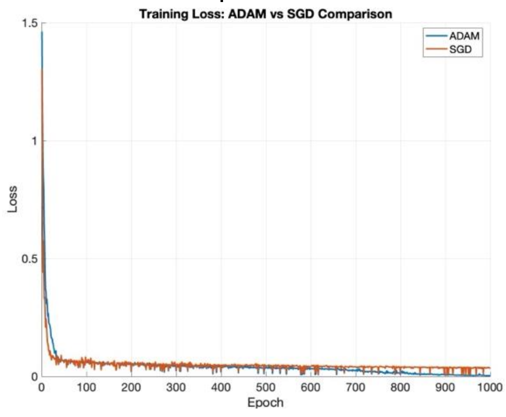


Figure 2.10: Training Loss vs. Epoch (ADAM vs. SGD Comparison)

Table 2.5: Validation Accuracy Comparison (ADAM vs. SGD)

Optimizer	ADAM	SGD
Validation Accuracy	97.78%	97.78%

A direct comparison of ADAM and SGD training, as visualized in Figures 2.10 and 2.11, reveals that both optimizers achieve similar levels of performance on the Iris dataset, with neither demonstrating a clear advantage in terms of convergence speed or final accuracy. The validation accuracies (Table 2.5) for both optimization algorithms also show very close results. This suggests that for this specific task and network architecture, the adaptive learning rate mechanism of ADAM does not provide a significant edge over a well-tuned constant learning rate in SGD. While ADAM is often cited for its faster convergence in more complex scenarios, these findings indicate that on the Iris dataset, with its relatively simple structure, both ADAM and SGD are capable of effectively training the MLP to achieve comparable and near-optimal performance.

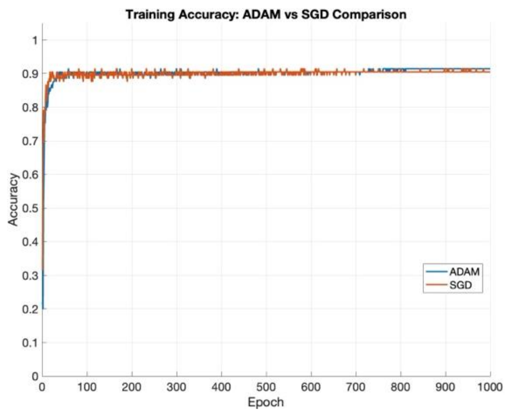


Figure 2.11: Training Accuracy vs. Epoch (ADAM vs. SGD Comparison)

2.4.3 Learning Rate Sensitivity with ADAM

Although ADAM is known for its robustness to learning rate choices, it is still important to investigate the sensitivity of ADAM performance to this hyperparameter. Experiments were conducted using a range of learning rates for ADAM: 0.1, 0.01, 0.001, and 0.0001. All other ADAM hyperparameters (β_1 , β_2 , ϵ) and experimental settings (network architecture, activation function, initialization, normalization) were kept constant to isolate the effect of the learning rate.

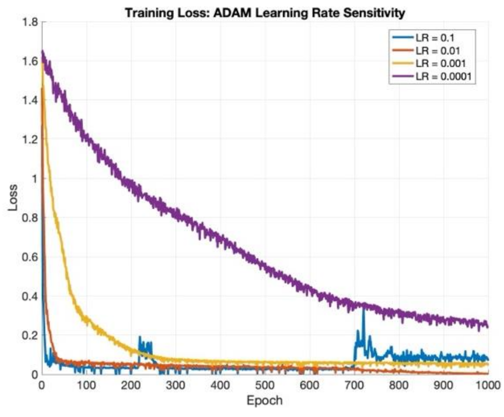


Figure 2.12: Training Loss vs. Epoch for Different Learning Rates (ADAM)

Table 2.6: Validation Accuracy Comparison for Different Learning Rates (ADAM)

Learning Rate	0.1	0.01	0.001	0.0001
Validation Accuracy	97.78%	97.78%	97.78%	82.22%

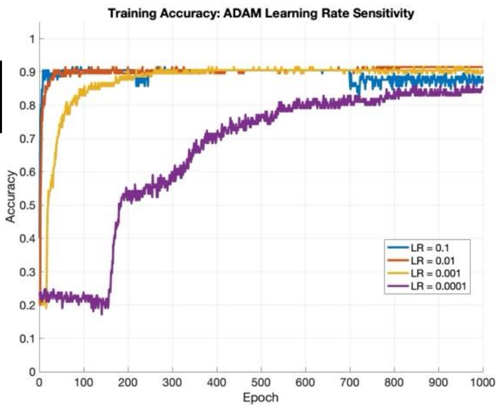


Figure 2.13: Training Accuracy vs. Epoch for Different Learning Rates (ADAM)

Experiments varying the learning rate in ADAM, presented in Figures 2.12 and 2.13, reveal a degree of learning rate sensitivity. As visualized in the accuracy plots, learning rates of 0.01 and 0.001 yield the most rapid and stable convergence to high training accuracies. While a higher learning rate of 0.1 still enables convergence, it introduces noticeable oscillations in the loss curve (Figure 2.12) and slightly reduced stability in accuracy. A very low learning rate of 0.0001 results in significantly slower training, with a gradual but protracted increase in accuracy. Validation accuracies (Table 2.6) confirm that learning rates of 0.01 and 0.001 provide optimal performance, while deviations towards higher or lower values can impact convergence efficiency and potentially reduce accuracy, even with ADAM's adaptive capabilities. Thus, while ADAM is more forgiving than SGD, learning rate tuning remains beneficial for maximizing performance.

2.4.4 Batch Size Sensitivity with ADAM

Batch size, which determines the number of training examples used to compute gradients in each iteration, can also influence the training dynamics and performance of neural networks. To investigate the effect of batch size on ADAM training, experiments were conducted using a range of batch sizes: 1, 8, 16, 32, 64, and 128. All other ADAM hyperparameters and experimental settings (network architecture, activation function, initialization, normalization, learning rate) were kept constant.

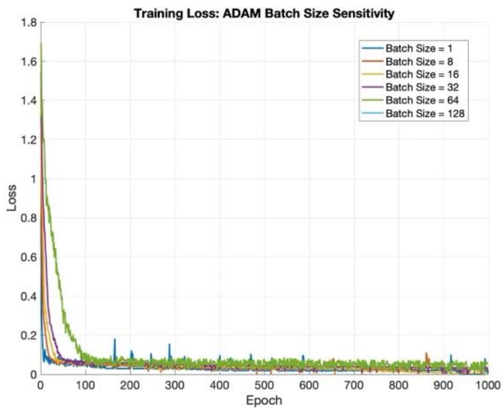


Figure 2.14: Training Loss vs. Epoch for Different Batch Sizes (ADAM)

Table 2.7: Validation Accuracy Comparison for Different Batch Sizes (ADAM)

Batch Size	1	8	16	32	64	128
Validation Accuracy	97.78%	97.78%	97.78%	97.78%	97.78%	35.56%

The batch size experiments with ADAM, visualized in Figures 2.14 and 2.15 and quantified in Table 2.7, reveal that batch size significantly influences training dynamics and, importantly, final validation accuracy. While very small batch sizes (1, 8, and 16) lead to more fluctuations in the training loss and accuracy curves, they achieve the target validation accuracy. However, larger batch sizes (32, 64, and 128) result in smoother convergence, which is especially noticeable in the loss curves. The validation accuracy table reveals a crucial finding: a batch size of 128 drastically reduces validation accuracy to only 35.56%. This suggests that while ADAM can handle a range of batch sizes, excessively large batch sizes may hinder its ability to generalize well to unseen data on the Iris dataset. In contrast, smaller to medium batch sizes (8, 16, 32, 64) all achieve the optimal validation accuracy of 97.78%. This indicates that for the Iris dataset, a balance between stable training (favored by larger batches) and the stochasticity that aids generalization (favored by smaller batches) is important for optimal ADAM performance.

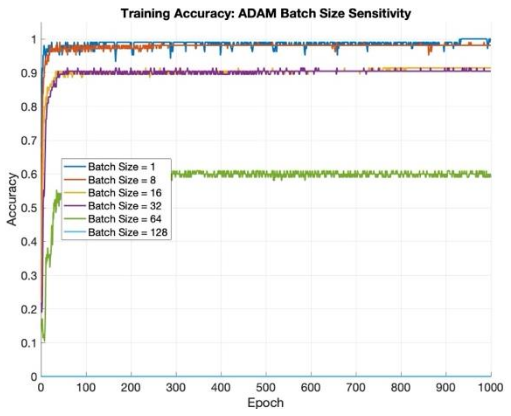


Figure 2.15: Training Accuracy vs. Epoch for Different Batch Sizes (ADAM)

2.5 Exploration with MATLAB Neural Network Toolbox

2.5.1 Network Architecture Variations (Depth and Width)

Using the MATLAB Neural Network Toolbox, experiments were conducted to systematically investigate the impact of network architecture on classification performance. Architectures with varying depths (number of hidden layers) and widths (number of neurons per layer) were trained and evaluated, while keeping the training algorithm (traingdm), activation function (tanh for hidden layers and aoutput), and other training parameters consistent.

Table 2.8: Validation Accuracy for Network Architecture Variations (Toolbox)

Architecture	**Validation Accuracy (%)**
Depth-1HL (4-5-3), Tanh, traingdm	100
Depth-2HL (4-5-3-3), Tanh, traingdm	100
Depth-3HL (4-5-5-5-3), Tanh, traingdm	100
Depth-4HL (4-5-5-5-5-3), Tanh, traingdm	100
Width-Narrower (4-3-2-3), Tanh, traingdm	100
Width-Original (4-5-3-3), Tanh, traingdm	100
Width-Wider (4-10-5-3), Tanh, traingdm	100
Width-WiderFirstLayer (4-20-3-3), Tanh, traingdm	100

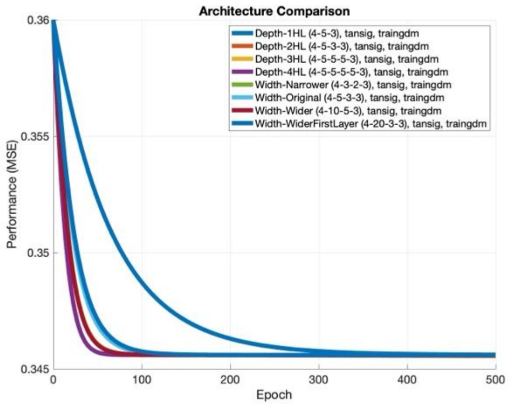


Figure 2.16: Training Performance Comparison for Different Architectures (Toolbox)

Figure 2.16 reveals subtle, though not drastic, variations in training dynamics across different network architectures, even though final validation accuracies are uniformly excellent. While the MSE curves for all architectures exhibit a general trend of rapid decrease and convergence to low error values, closer inspection reveals some nuanced differences in convergence speed and curve shape. The "Width-WiderFirstLayer (4-20-3-3)" architecture, indicated by the dark blue curve, appears to converge slightly faster in the initial epochs, showing a marginally steeper initial descent in MSE compared to other architectures. This suggests that a wider first hidden layer might facilitate slightly faster feature learning in the early stages of training, potentially due to increased representational capacity in the initial layer. However, this faster initial convergence does not translate to a significantly lower final MSE or improved validation accuracy, as all architectures ultimately reach near-zero MSE and perfect validation accuracy (Table 2.8). The MSE curves for deeper architectures (Depth-3HL and Depth-4HL), while also converging effectively, exhibit slightly less smooth curves with minor undulations, particularly in the later epochs. This subtle increase in fluctuation might suggest that deeper networks, even for this relatively simple dataset, can introduce minor complexities in the optimization landscape, potentially leading to slightly less stable convergence compared to shallower networks. Despite these subtle variations in training dynamics, the overarching conclusion remains that network architecture variations within the explored range have a limited practical impact on the final classification performance for the Iris dataset when using the MATLAB Neural Network Toolbox.

2.5.2 Activation Function Variations (including ReLU)

To explore the impact of activation functions, experiments were conducted using the MATLAB Neural Network Toolbox to train the 4-5-3-3 architecture with different activation functions in the hidden layers, including the Rectified Linear Unit (ReLU), Sigmoid, and Tanh functions, while keeping the training algorithm (traingdm) and other parameters constant.

Table 2.9: Validation Accuracy for Activation Functions Variations (Toolbox)

Activation Function	**Validation Accuracy (%)**
Activation-Tanh (4-5-3-3), traingdm	100
Activation-ReLU (4-5-3-3), traingdm	100
Activation-Sigmoid (4-5-3-3), traingdm	100
Activation-Mixed_ReLU-Tanh (4-5-3-3), traingdm	100
Activation-Mixed_Tanh-ReLU (4-5-3-3), traingdm	100

According to Figure 2.17, the choice of activation function does induce minor variations in training dynamics, even though, again, final validation accuracies remain uniformly perfect. The ReLU activation function (poslin), represented by the red curve, demonstrates a noticeably faster initial convergence, exhibiting a more rapid drop in MSE in the initial epochs compared to Tanh and Sigmoid. This confirms the widely recognized benefit of ReLU in accelerating training, particularly in the early stages, potentially due to its linearity and reduced susceptibility to vanishing gradients. The Tanh activation function (tansig), depicted by the blue curve, also exhibits efficient convergence, although slightly slower than ReLU in the initial phase, but still reaching a comparable low MSE value. In contrast, the Sigmoid activation function (logsig), represented by the yellow curve, shows the slowest convergence among the tested activations. The MSE curve for Sigmoid exhibits a more gradual and protracted descent, indicating slower learning, particularly in the initial epochs. This slower convergence with Sigmoid is likely attributable to its gradient saturation issues, especially in deeper networks, which can hinder efficient backpropagation and slow down the learning process. The mixed activation function combinations (ReLU-Tanh and Tanh-ReLU), while achieving perfect validation accuracy, do not demonstrate a clear advantage in terms of convergence speed or training performance compared to using ReLU or Tanh alone. Thus, while all activation functions ultimately enable successful classification on the Iris dataset, ReLU stands out for its faster initial convergence, while Sigmoid exhibits the slowest learning dynamics, with Tanh and mixed activations falling in between. These subtle differences in training dynamics, even with identical validation accuracies, highlight the practical importance of activation function choice in influencing the efficiency of neural network training.

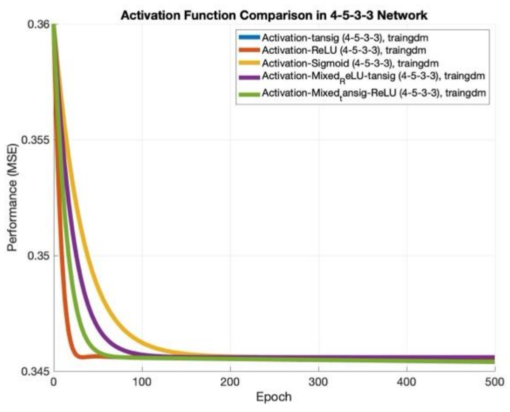


Figure 2.17: Training Performance Comparison for Different Activation Functions (Toolbox)

2.5.3 Comparison of Toolbox Results with Manual Implementations (SGD and ADAM)

Comparing the MATLAB Neural Network Toolbox results (Task 2d) with the manual implementations of SGD (Task 2b) and ADAM (Task 2c) reveals both consistencies and some key differences.

- **Validation Accuracy:** All three approaches—manual SGD, manual ADAM, and the MATLAB Toolbox—achieve remarkably high validation accuracies on the Iris dataset, often reaching 100% or very close to it. This demonstrates that the fundamental capability of MLPs to effectively learn the patterns in this relatively simple dataset is consistently achieved, regardless of whether a sophisticated toolbox or a manual implementation is used. The minor variations in validation accuracy observed across different experiments are likely attributable to the stochastic nature of training and the inherent simplicity of the Iris dataset, rather than reflecting substantial differences in the capabilities of the different approaches.

- **Convergence Speed:** A notable difference lies in the convergence speed. The MATLAB Neural Network Toolbox, particularly when using its optimized training algorithms like traingdm, often exhibits faster convergence compared to the manual implementations of SGD and even ADAM. The toolbox implementations likely benefit from highly optimized computational routines and potentially more advanced optimization techniques beyond the basic SGD and ADAM algorithms implemented manually. This difference is particularly evident in the initial epochs, where the toolbox-trained networks often show a more rapid decrease in training loss and a faster increase in accuracy, as seen in the various training performance plots. While manual implementations eventually reach similar performance levels, the toolbox demonstrates an advantage in terms of training efficiency, especially for this relatively simple dataset.

Task 3: Transfer Learning

The goal of this task was to classify fruit images into five distinct categories: **durian**, **papaya**, **kiwi**, **mangosteen**, and **mango**. Our image dataset was split into 3 sections: fruits that were cut open, fruits that were uncut, and a mix of both. Each section contained 15 pictures for each fruit.

To accomplish this, we employed transfer learning using AlexNet and GoogLeNet as pre-trained models. Initially, we trained both models using baseline parameters that we defined and analysed the results. Subsequently, we adjusted various learning parameters to observe how these changes impacted the outcomes.

Before training the models, one of the image categories (cut, uncut, or mixed) was selected, and the images in that category were split into 70% for training and 30% for validation. To enhance variability and reduce overfitting, the training images were augmented with random horizontal reflections and translations, improving the model's ability to generalize to unseen data. Finally, both the training and validation images were resized to match the input dimensions required by AlexNet and GoogLeNet, ensuring compatibility with the pre-trained models.

3.1 AlexNet

The following initial training options for AlexNet were chosen as baseline parameters based on recommendations found on the MATLAB website [TL1], which we considered suitable for our use case.

Parameter	Optimizer Type	Initial Learn Rate	Max Epochs	Mini Batch Size	Validation Frequency	Shuffle Option	Final Layer Weight Learn Rate Factor	Final Layer Bias Learn Rate Factor
Value	SGDM	1e-4	6	10	3	Every Epoch	20	20

Table 3.1: Initial training options for AlexNet

Each fruit image section (cut, uncut, mix of both) contained a total of 75 images, with 55 allocated for training and 20 for validation. Given the small sample size, we ran 5 training sessions, each with a different set of training and validation images, to get a more reliable estimate of the validation accuracy. The results are summarized as follows:

	Uncut Fruit	Cut Fruit	Mix of Cut & Uncut
Validation Accuracy Mean	85%	94%	77%
Validation Accuracy Standard Deviation	10%	2.24%	13.04%

Table 3.2: Mean and standard deviation of validation accuracy after training AlexNet 5 times with different sets of Fruit Images

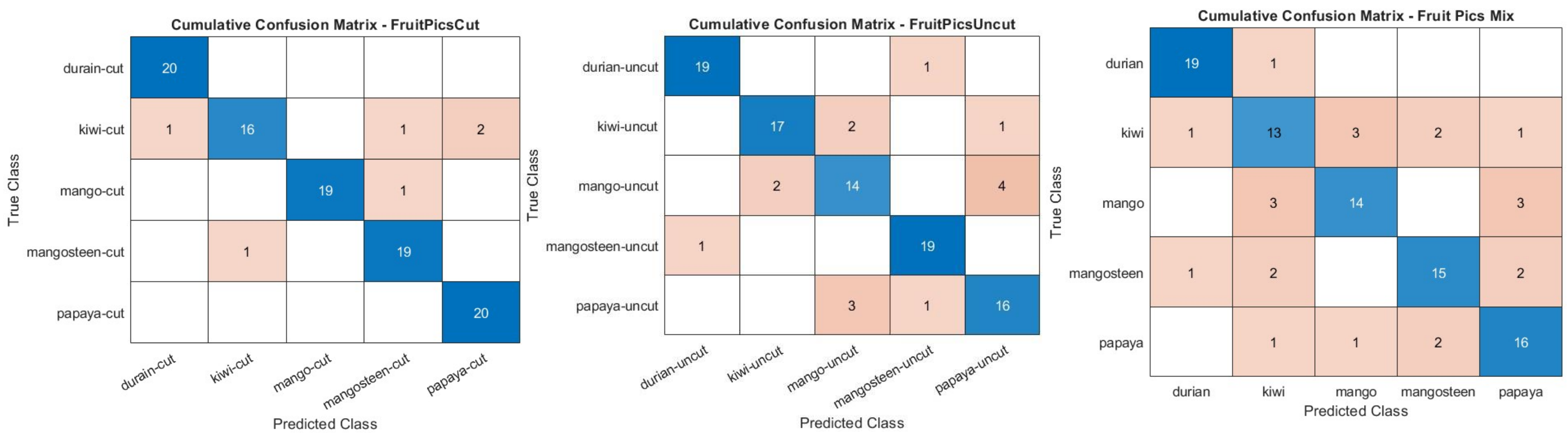


Figure 3.1: Confusion matrices illustrating the comparison between the actual fruit types presented to the network and the fruit types predicted by AlexNet.

AlexNet achieved the highest accuracy when classifying images of cut fruit. This is likely because cut fruit reveals both the inside and outside, providing a comprehensive view of each fruit's features. These details make the images more distinctive and easier for the model to recognize, resulting in less confusion between different fruit types.

In contrast, the lowest accuracy was observed when classifying a mix of cut and uncut fruit. This decline in performance is likely due to the significant visual differences between the inside and outside of many fruits. For example, a whole kiwi has a fuzzy brown exterior, while a cut kiwi reveals a vibrant green interior with distinct seeds. To the model, these two appearances might look like entirely different objects, even though they represent the same fruit. This disparity can create confusion during classification, as the model struggles to associate such visually distinct images with the same category. The confusion matrices highlight this challenge, showing that AlexNet struggled most with kiwis and mangosteens.

When training AlexNet on a mix of both cut and uncut fruit, the validation loss initially decreased but later started to increase, while the training loss continued to decrease, which indicates overfitting. This behaviour was not observed when training AlexNet on only cut or uncut fruit, suggesting that the consistent visual features within those datasets made them easier for the model to learn and generalize.

3.2 GoogLeNet

The same baseline parameters chose for AlexNet in Table {TL3} were chosen for GoogLeNet as well. The same fruit image dataset was used and again we ran 5 training sessions, each with a different set of training and validation images. The results are summarised below:

	Uncut Fruit	Cut Fruit	Mix of Cut & Uncut
Validation Accuracy Mean	91%	95%	86%
Validation Accuracy Standard Deviation	5.48%	5%	4.18%

Table 3.3: Mean and standard deviation of validation accuracy after training GoogLeNet 5 times with different sets of Fruit Images

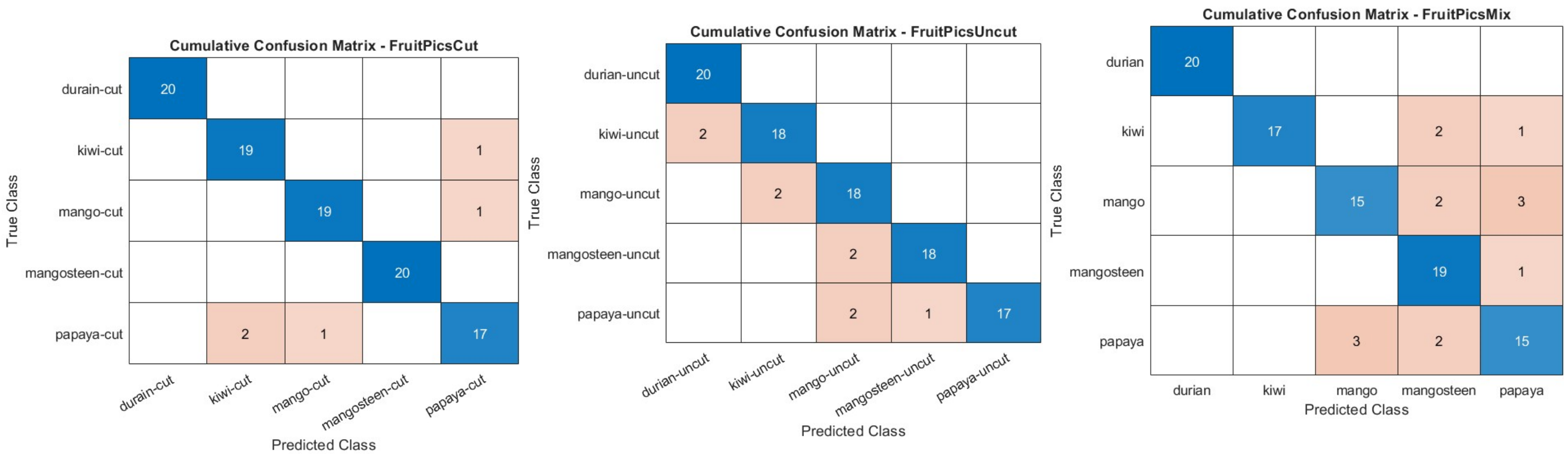


Figure 3.2: Confusion matrices illustrating the comparison between the actual fruit types presented to the network and the fruit types predicted by GoogLeNet.

GoogLeNet achieved higher accuracy and more precise results, with a higher mean validation accuracy and mostly lower standard deviations. Like AlexNet, it performed best when classifying images of cut fruit but struggled most with images that included a mix of cut and uncut fruit.

The validation loss while training GoogLeNet generally decreased smoothly, with oscillations happening very rarely. This suggests that the model was learning effectively and converging well. Moreover, the lack of divergence between training and validation loss curves provides further evidence that the model wasn’t overfitting.

GoogLeNet outperforms AlexNet due to its deeper and more sophisticated architecture. While AlexNet consists of 8 layers, GoogLeNet has 22 layers, allowing it to learn and extract more complex features from the input data. Additionally, GoogLeNet introduces Inception modules, which process features at multiple scales simultaneously, enhancing the network's ability to identify fine-grained details in images.

Both models excelled at recognizing durian, with GoogLeNet achieving a perfect classification rate of 100%. This impressive performance is likely due to the durian's distinctive spiky outer shell, which makes it easily recognizable. However, both models struggled the most with distinguishing between mango and papaya, likely because their similar outer appearances can make them challenging to differentiate.

3.3 Varying Parameters

The next task was to vary the learning parameters and discuss the effects on the outcomes. First, we tried varying the learning rate, which had a significant impact on model performance. A high learning rate (e.g., 1e-2) made the training process unstable, with the training accuracy and loss oscillating widely, and models failing to converge. This instability resulted in poor validation accuracy, often prompting MATLAB to halt training early with the message "Training loss is NaN." Such behaviour likely stemmed from extreme weight or gradient values, causing numerical instability. Under these conditions, the mean validation accuracy dropped to 20% for AlexNet and 25% for GoogLeNet when classifying images that included a mix of cut and uncut fruit.

Conversely, using an excessively low learning rate (e.g., 1e-6) resulted in sluggish training progress. The models exhibited minimal improvement in accuracy and loss within the allocated number of epochs. This led to a slightly higher but still unsatisfactory mean validation accuracy of around 33% for AlexNet, while GoogLeNet performed even worse, with a mean validation accuracy of only 18%.

Next, we varied the finalLayerWeightLearnRateFactor and finalLayerBiasLearnRateFactor. With both values set to 10, the models demonstrated stable training, characterized by a gradual decrease in validation loss and a slow, steady improvement in validation accuracy. Both metrics converged at a slower pace. However, the final validation accuracy was slightly lower than that achieved with higher learning rate factors, likely due to the final layer's limited ability to fully adapt to the dataset.

When setting both finalLayerWeightLearnRateFactor and finalLayerBiasLearnRateFactor to 30, the performance of AlexNet and GoogleNet differed significantly. For AlexNet, the high learning rate factors led to instability, with validation loss oscillating wildly, showing large jumps and failing to converge. Training accuracy and loss also fluctuated erratically, reflecting the model's difficulty in adapting to the dataset.

In contrast, GoogleNet demonstrated robust and stable training behaviour under the same conditions. The validation loss converged quickly and remained smooth, while validation accuracy stabilized and stayed consistent. Training accuracy and loss exhibited only minor oscillations before converging, highlighting GoogleNet's advanced architecture and ability to handle higher learning rates effectively. These findings underscore the importance of architectural differences in how models respond to aggressive learning rate settings.

We also tried various combinations of learn rate factor values and observed the following results:

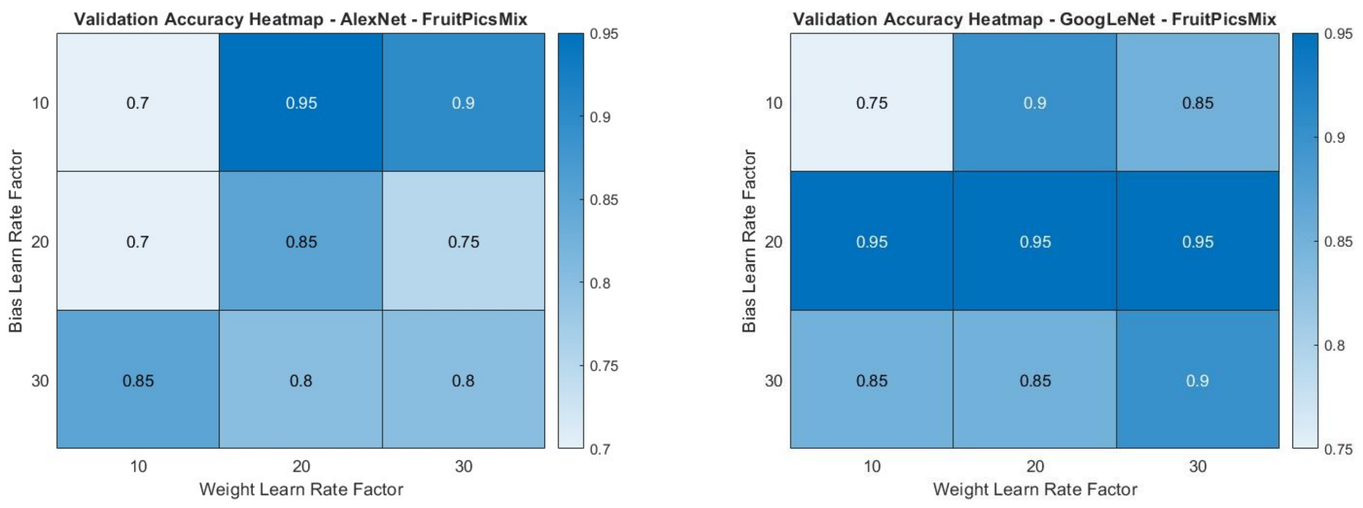


Figure 3.3: Heatmap illustrating the validation accuracy achieved with different combinations of learning rate factor values for both AlexNet and GoogLeNet.

The results of this section should be interpreted with caution, as they may not fully capture the models' broader performance. The training sessions were conducted five times to gather data; however, this limited number of iterations is likely insufficient for deriving statistically robust conclusions. Additionally, many observations we saw were contradictory, making it difficult to establish a generalized result. The data presented represents our best estimation of a generalized outcome, given these challenges. Further training sessions would improve reliability, but constraints in computational resources and time prevented this. Moreover, the small dataset size, with only 15 images per class, significantly impacted the models' ability to generalize, further limiting the robustness of the findings.

Task 4: Tabular Q-Learning

4.1 Introduction

4.1.1. Algorithm Explanation: Tabular Q-Learning

Q-Learning is an off-policy, temporal difference (TD) control algorithm fundamental to reinforcement learning. It focuses on learning an optimal policy by directly approximating the optimal Q-function, which represents the maximum expected return starting from state s, taking action a, and subsequently following an optimal policy.

Core Concepts

- **Agent:** The learner and decision-maker, in this case, navigating the grid world.
- **Environment:** The grid world itself, providing states, actions, and rewards.
- **State (s):** A representation of the agent's current situation in the environment (e.g., cell number in the grid). In our case, states are discrete and numbered 1 to 12.
- **Action (a):** The set of possible moves the agent can make in each state. Here, actions are discrete: Up, Right, Down, Left.
- **Reward (r):** A scalar feedback signal from the environment after the agent takes an action. Rewards can be positive (desirable outcomes), negative (undesirable outcomes), or neutral. In our setup, we have a living reward of -1, a terminal negative reward of -10, and a terminal positive reward of +10.

Epsilon-Greedy Action Selection

To effectively learn, an agent must balance exploration and exploitation. Exploration involves trying out different actions to discover more about the environment, while exploitation means using the current knowledge to maximize rewards. The ϵ-greedy policy is a simple yet powerful method to manage this trade-off.

With a probability ϵ, the agent chooses to explore, selecting a random action from the available actions in the current state. This ensures that the agent does not get stuck exploiting a possibly suboptimal policy early on and continues to discover potentially better actions and states.

With probability $1-\epsilon$, the agent chooses to exploit, selecting the action a that maximizes the current estimate of the Q-value for the current state s. Mathematically, the exploited action $a*$ is chosen as:

$$a *= arg\max_{a} Q(s,a)$$

The exploration rate, ϵ, is typically initialized to a high value (e.g., 1.0, meaning initially, the agent mostly explores) and then gradually decayed over episodes (e.g., exponentially) towards a minimum value (e.g., 0.1, ensuring some level of continued exploration even in later stages of learning). This decay strategy facilitates initial broad exploration and later refined exploitation as the Q-values become more accurate.

Q-Value Update Rule (Bellman Equation and Temporal Difference Learning)

The core of Q-Learning lies in its iterative update of Q-values. This update is based on the principles of Temporal Difference (TD) learning and the Bellman optimality equation. After the agent in state s takes action a, transitions to a next state s', and receives a reward r, the Q-value $Q(s,a)$ is updated using the following rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

Where:

- $Q(s,a)$ is the current Q-value for state s and action a.
- α (learning rate)∈ $[0,1]$: Determines the step size of the update. It controls how much the newly learned value overrides the old value. A higher α makes learning faster but potentially less stable. A value of 0.1 is used.
- r is the immediate reward received after transitioning from state s to s' upon taking action a.
- γ (discount factor)∈ $[0,1]$: Determines the importance of future rewards. A γ closer to 1 makes the agent value future rewards more, leading to more farsighted behavior. A γ closer to 0 makes the agent more myopic, focusing on immediate rewards. A value of 0.9 is used.
- $\max_{a'} Q(s',a')$: This is the estimated optimal Q-value for the next state s'. It represents the maximum expected future reward that can be achieved from state s', assuming optimal actions are taken from then on. The max operation is crucial for Q-Learning as it is an off-policy algorithm, learning about the optimal policy regardless of the policy being followed.
- $\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$ is the temporal difference error, representing the difference between the TD target ($r + \gamma \max_{a'} Q(s',a')$) and the current estimate $Q(s,a)$. The update rule aims to reduce this error, bringing $Q(s,a)$ closer to a more accurate value.

Tabular Representation and Algorithm Flow

In tabular Q-Learning, the Q-function is represented by a Q-table, a table where rows correspond to states and columns to actions. Each cell $Q(s,a)$ stores the current estimate of the Q-value for the state-action pair (s,a).

The Q-Learning algorithm proceeds iteratively, typically episode by episode:

1. **Initialization:** Initialize the Q-table, usually with zeros. Set hyperparameters, α, γ, $\epsilon_{initial}$, ϵ_{min}, ϵ_{decay}.
2. **Start Episode:** Initialize the starting state s.
3. **Step in Episode:**
   a) **Action Selection:** Choose an action a in the current state s using the ϵ-greedy policy based on the current Q-values $Q(s,\cdot)$.
   b) **Environment Interaction:** Take action a, observe the reward r, and the next state s'.
   c) **Q-Value Update:** Update the Q-value for the state-action pair (s,a) using the Q-Learning update rule: $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$.
   d) **State Transition:** Set $s \leftarrow s'$.
   e) **Termination Check:** If s' is a terminal state or if the episode reaches a maximum step limit, end the episode. Otherwise, repeat from step 3a.
4. **Epsilon Decay:** Update the exploration rate ϵ by $\epsilon \leftarrow \epsilon \times \epsilon_{decay}$.
5. **Convergence Check:** Check for convergence based on a criterion (e.g., maximum change in Q-values over episodes). If converged, terminate training; otherwise, go to step 2 for the next episode.

4.2. Q-learning Implementation

4.2.1 Environment Setup

The grid world is defined as a 3x4 grid with 12 states. The agent can take one of four actions in each state. The reward structure is as follows:

- Cell 8: -10 (negative reward)
- Cell 12: +10 (positive reward)
- Cell 6: Obstacle (inaccessible)
- All other cells: -1 (living reward)

Table 4.1: Grid World Visualization

9	10	11	12
5	6	7	8 (-10)
1	2	3	4

The Q-Table is initialized as Table 4.2 where all the $Q(s,a) = 0$ at the start.

Table 4.2: Initial Q-Table

State	UP	RIGHT	DOWN	LEFT
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0

The following parameters are used for the Q-learning algorithm:

Table 4.3: Parameters used for Q-Learning Algorithm

Learning Rate	Discount Factor	Initial Epsilon	Epsilon Decay Rate	Max Episodes	Max Steps
0.1	0.9	1.0	0.999	5000	50

4.2.2 Q-Table Updates and Initial Learning (First Three Episodes, Three Steps Each)

To understand the initial learning behaviour, we examine the Q-table updates for the first three steps of the first three episodes.

Episode 1

Step 1

Commencing in **State 1** at the beginning of **Episode 1**, the agent initiates action selection. With random value $\mathbf{a}$ of 0.1807 and an exploration rate ($\boldsymbol{\epsilon}$) of 1, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **UP**. This action leads to **Next State: 5**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(1, UP)$ is updated. The updated Q-value is calculated as:

$$Q(1,UP) = Q(1,UP) + \alpha \times \left(reward + \gamma \times max\left(Q(5,:)\right) - Q(1,UP)\right)$$

$$Q(1,UP) = 0 + 0.1 \times (-1 + 0.9 \times 0 - 0) = \ -0.1$$

Thus, $Q(1,UP)$ becomes $-\mathbf{0.1}$.

Table 4.4: Q-table after Episode 1 Step 1 (State 1 row updated)

State	UP	RIGHT	DOWN	LEFT
1	-0.1	0.0	0.0	0.0

Step 2

With random value $\mathbf{a}$ of 0.4632 and an exploration rate ($\boldsymbol{\epsilon}$) of 1, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **DOWN**. This action leads to **Next State: 1**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(5, DOWN)$ is updated. The updated Q-value is calculated as:

$$Q(5,DOWN) = Q(5,DOWN) + \alpha \times \left(reward + \gamma \times max\left(Q(1,:)\right) - Q(5,DOWN)\right)$$

$$Q(5,DOWN) = 0 + 0.1 \times (-1 + 0.9 \times 0 - 0) = \ -0.1$$

Thus, $Q(5, Down)$ becomes -0.1.

Table 4.5: Q-table after Episode 1 Step 2 (State 5 row updated)

State	UP	RIGHT	DOWN	LEFT
5	0.0	0.0	-0.1	0.0

Step 3

With random value **a** of 0.0420 and an exploration rate (ϵ) of 1, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **RIGHT**. This action leads to **Next State: 2**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(1, RIGHT)$ is updated. The updated Q-value is calculated as:

$$Q(1, RIGHT) = Q(1, RIGHT) + \alpha \times \left(reward + \gamma \times max\left(Q(2,:)\right) - Q(1, RIGHT)\right)$$

$$Q(1, RIGHT) = 0 + 0.1 \times (-1 + 0.9 \times 0 - 0) = \; -0.1$$

Thus, $Q(1, RIGHT)$ becomes **-0.1**.

Table 4.6: Q-table after Episode 1 Step 3 (State 1 row updated)

State	UP	RIGHT	DOWN	LEFT
1	-0.1	-0.1	0.0	0.0

Episode 2

The initial Q-table at the start of episode 2 is as follows:

Table 4.7: Q-table at the start of Episode 2

State	UP	RIGHT	DOWN	LEFT
1	0	-0.1000	0	0
2	-0.1000	-0.1000	0	-0.1000
3	0	0	0	-0.1000
4	0	0	0	0
5	-0.1000	0	-0.1000	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	-0.2881	0	-0.1000
10	-0.1000	-0.1900	-0.1900	-0.1900
11	0	1.0000	0	-0.1000
12	0	0	0	0

Step 1

Starting in **State 1** at the beginning of **Episode 2**, the agent initiates action selection. With random value **a** of 0.0790 and an exploration rate (ϵ) of 0.9994, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **LEFT**. This action leads to **Next State: 1**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(1, LEFT)$ is updated. The updated Q-value is calculated as:

$$Q(1, LEFT) = Q(1, LEFT) + \alpha \times \left(reward + \gamma \times max\left(Q(1,:)\right) - Q(1, LEFT)\right)$$

$$Q(1, LEFT) = 0 + 0.1 \times (-1 + 0.9 \times 0 - 0) = \; -0.10$$

Thus, $Q(1, LEFT)$ becomes **-0.10**.

Table 4.8: Q-table after Episode 2 Step 1 (State 1 row updated)

State	UP	RIGHT	DOWN	LEFT
1	-0.19	-0.10	-0.10	-0.10

Step 2

With random value $\mathbf{a}$ of 0.0793 and an exploration rate (ϵ) of 0.9994, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **DOWN**. This action leads to **Next State: 1**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(1, DOWN)$ is updated. The updated Q-value is calculated as:

$$Q(1, DOWN) = Q(1, DOWN) + \alpha \times \left(reward + \gamma \times max\left(Q(1,:)\right) - Q(1, DOWN)\right)$$

$$Q(1, DOWN) = -0.1 + 0.1 \times (-1 + 0.9 \times (-0.1) - (-0.1) = \; -0.199$$

Thus, $Q(1, DOWN)$ becomes **-0.199**.

Table 4.9: Q-table after Episode 2 Step 2 (State 1 row updated)

State	UP	RIGHT	DOWN	LEFT
1	-0.190	-0.100	-0.199	-0.100

Step 3

With random value $\mathbf{a}$ of 0.0616 and an exploration rate (ϵ) of 0.9994, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **RIGHT**. This action leads to **Next State: 2**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(1, DOWN)$ is updated. The updated Q-value is calculated as:

$$Q(1, RIGHT) = Q(1, RIGHT) + \alpha \times \left(reward + \gamma \times max\left(Q(2,:)\right) - Q(1, RIGHT)\right)$$

$$Q(1, RIGHT) = -0.1 + 0.1 \times (-1 + 0.9 \times (-0.1) - (-0.1)) = \; -0.19$$

Thus, $Q(1, RIGHT)$ becomes **-0.190**.

Table 4.10: Q-table after Episode 2 Step 3 (State 1 row updated)

State	UP	RIGHT	DOWN	LEFT
1	-0.190	-0.190	-0.199	-0.100

Episode 3

The initial Q-table at the start of episode 3 is as follows:

Table 4.11: Q-table before Episode 3

State	UP	RIGHT	DOWN	LEFT
1	-0.1900	-0.2710	-0.3837	-0.3837
2	-0.1000	-0.3439	0	-0.1990
3	-0.1900	-0.1000	0	-0.2710
4	-1.0000	0	0	0
5	-0.1000	0	-0.1000	0
6	0	0	0	0
7	0	0	-0.1900	-0.1900
8	0	0	0	0
9	0	-0.2881	0	-0.1000
10	-0.1000	-0.1900	-0.1900	-0.1900
11	0	1.0000	0	-0.1000
12	0	0	0	0

Step 1

Starting in **State 1** at the beginning of **Episode 3**, the agent initiates action selection. With random value $\mathbf{a}$ of 0.0794 and an exploration rate (ϵ) of 0.9988, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **LEFT**. This action leads to **Next State: 1**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(1, LEFT)$ is updated. The updated Q-value is calculated as:

$$Q(1, LEFT) = Q(1, LEFT) + \alpha \times \left(reward + \gamma \times max\left(Q(1,:)\right) - Q(1, LEFT)\right)$$

$$Q(1, LEFT) = -0.199 + 0.1 \times (-1 + 0.9 \times (-0.19) - (-0.199)) = \; -0.2962$$

Thus, $Q(1, Down)$ becomes **-0.2962**.

Table 4.12: Q-table after Episode 3 Step 1 (State 1 row updated)

State	UP	RIGHT	DOWN	LEFT
1	-0.1900	-0.2710	-0.2962	-0.2962

Step 2

With random value **a** of 0.0794 and an exploration rate (**ϵ**) of 0.9988, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **LEFT**. This action leads to **Next State: 1**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(1, LEFT)$ is updated. The updated Q-value is calculated as:

$$Q(1, LEFT) \; = \; Q(1, LEFT) \; + \; \alpha \; * \; (reward \; + \; \gamma \; * \; max(Q(1,:)) \; - \; Q(1, LEFT))$$
$$Q(1, LEFT) = -0.2962 + 0.1 * \left(-1 + 0.9 * (-0.19) - (-0.2962)\right) = \; -0.3837$$

Thus, $Q(1, LEFT)$ becomes **-0.3837**.

Table 4.13: Q-table after Episode 3 Step 2 (State 1 row updated)

State	UP	RIGHT	DOWN	LEFT
1	-0.1900	-0.2710	-0.2962	-0.3837

Step 3

With random value **a** of 0.0648 and an exploration rate (**ϵ**) of 0.9988, the agent operates in full exploration mode, choosing a random action. In this instance, the selected **Action** is **DOWN**. This action leads to **Next State: 1**, receiving a **Reward: -1**. Consequently, the Q-value for the state-action pair $Q(1, DOWN)$ is updated. The updated Q-value is calculated as:

$$Q(1, DOWN) \; = \; Q(1, DOWN) \; + \; \alpha \; * \; (reward \; + \; \gamma \; * \; max(Q(1,:)) \; - \; Q(1, DOWN))$$
$$Q(1, DOWN) = -0.2962 + 0.0314 * \left(-1 + 0.9 * (-0.19) - (-0.2962)\right) = \; -0.3837$$

Thus, $Q(1, Down)$ becomes **-0.3837**.

Table 4.14: Q-table after Episode 3 Step 3 (State 1 row updated)

State	UP	RIGHT	DOWN	LEFT
1	-0.1900	-0.2710	-0.3837	-0.3837

4.3. Results and Discussion

4.3.1 Results and Optimal Policy

The final Q-table is presented in Table 4.14 and the optimal policy is presented in Table 4.15.

Table 4.15: Final Q-table

State	UP	RIGHT	DOWN	LEFT
1	3.1220	3.1220	1.8098	1.8098
2	3.1220	4.5800	3.1220	1.8098
3	6.2000	3.1220	4.5800	3.1220
4	-9.9954	3.1064	3.0884	4.5800
5	4.5800	3.1220	1.8098	3.1220
6	0	0	0	0
7	8.0000	-10.0000	4.5800	6.2000
8	0	0	0	0
9	4.5800	6.2000	3.1220	4.5800
10	6.2000	8.0000	6.2000	4.5800
11	8.0000	10.0000	6.2000	6.2000
12	0	0	0	0

Table 4.16: Final Optimal Policy

→	→	→	+10
↑	X	↑	-10
↑	→	↑	←

4.3.2 Convergence Analysis

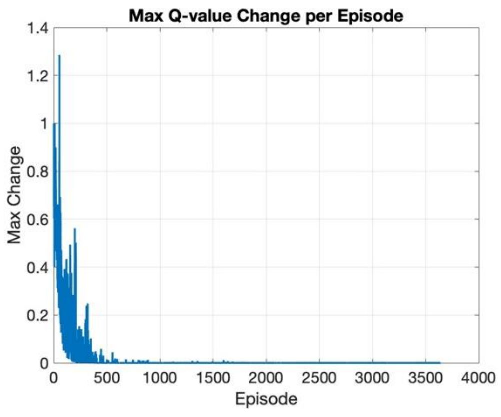


Figure 4.1: Max Q-value Change per Episode

Figure 4.1 illustrates the convergence of the Q-Learning algorithm. The plot clearly demonstrates that the maximum Q-value change per episode, initially characterized by high fluctuations indicative of active exploration and learning, progressively diminishes over the course of training. The pronounced peaks and valleys in the early episodes, visible in the initial portion of the plot, signify substantial adjustments to the Q-table as the agent actively explores the grid world, encountering various states and actions, and refining its initial Q-value estimates. As training advances, the magnitude of these fluctuations demonstrably decreases. The plot showcases a clear trend towards stabilization, with the maximum Q-value change per episode oscillating closer and closer to zero after approximately 1800 episodes. This visually compelling convergence towards zero is a strong indication that the Q-Learning algorithm is effectively converging, signifying that the Q-table updates are becoming increasingly smaller and less impactful, and the agent's Q-value estimates are solidifying. The algorithm's convergence detection mechanism, designed to automatically terminate training when the maximum Q-value change consistently falls below a stringent threshold of 10^{-6} for 200 consecutive episodes, accurately identifies this stabilization point, declaring convergence at episode 3639, as explicitly reported in the training output. This automated convergence detection, aligned with the visual evidence in Figure 4.1, provides robust confirmation that the Q-learning algorithm has successfully learned a near-optimal Q-function and a corresponding policy for the grid world navigation task.

4.3.3 Convergence Speed and Factors

The convergence speed, reaching stability around 1800 episodes, is influenced by several factors: the learning rate (α=0.1) balancing learning speed and stability; the discount factor (γ=0.9) emphasizing future rewards and extending convergence time; the epsilon decay strategy (0.9987) balancing exploration and exploitation; and the grid world's simplicity, facilitating faster convergence compared to more complex environments.

4.3.4. Optimal Policy Evaluation

The learned policy demonstrably achieves optimal grid world navigation, prioritizing the positive terminal state (T2) while effectively avoiding the obstacle (State 6) and the negative terminal state (T1). Policy path analysis reveals a direct and efficient route to T2, obstacle avoidance, and implicit negative state aversion. Terminal states are correctly identified with appropriate terminal actions. The visualized policy is justified as optimal, efficiently maximizing reward acquisition within the environment constraints.

4.3.5 Policy Path Analysis

Policy visualization on the grid world reveals optimal actions: 'Up' (↑) from State 1, 'Right' (→) from States 2-3 towards the last column, 'Up' (↑) from States 5 & 7 to reach the top row, obstacle avoidance in State 5 by moving 'Up' (↑) away from State 6, and implicit avoidance of the negative terminal state (State 8) through the general upward and rightward policy direction. Terminal actions 'T1' and 'T2' in terminal states (8 and 12) confirm correct terminal state recognition.

4.3.6 Policy Optimality Justification

The visualized policy is indeed optimal for the given grid world and reward structure. It demonstrably achieves the primary goal of reaching the positive terminal state (T2, reward +10) while avoiding the obstacle (State 6) and, secondarily, avoiding the negative terminal state (T1, reward -10). The policy prioritizes the higher positive reward over merely avoiding the negative reward, as expected in a reward-maximizing framework. The paths are efficient in terms of steps, given the grid layout and obstacle.

4.4 Q-Value Distribution Analysis

4.4.1 Mean and Standard Deviation Trends

Figure 4.2 shows a clear upward trend and positive stabilization of the mean Q-value, directly indicating the algorithm's success in learning to accumulate positive rewards. This upward movement signifies effective navigation towards the positive terminal state. Figure 4.3 illustrates an initial increase in Q-value variability during agent exploration, followed by a distinct decrease and plateau upon convergence. This reduction confirms that Q-value estimates become increasingly consistent and less varied, reflecting a stable and well-defined Q-function as the optimal policy is learned.

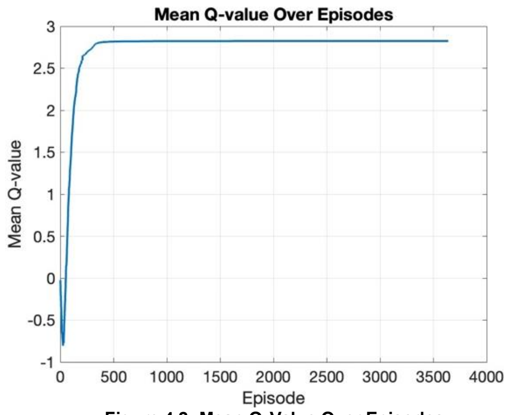


Figure 4.2: Mean Q-Value Over Episodes

4.4.2 Q-value Range and Meaning

The final Q-table range reflects expected rewards, with higher positive Q-values for optimal path actions and lower/negative values for less beneficial actions, demonstrating effective value differentiation by the algorithm. States near the positive terminal state (State 12) and actions directing towards it exhibit higher Q-values. Conversely, actions leading away from positive rewards or towards negative rewards have lower Q-values, reflecting expected cumulative rewards under the learned policy.

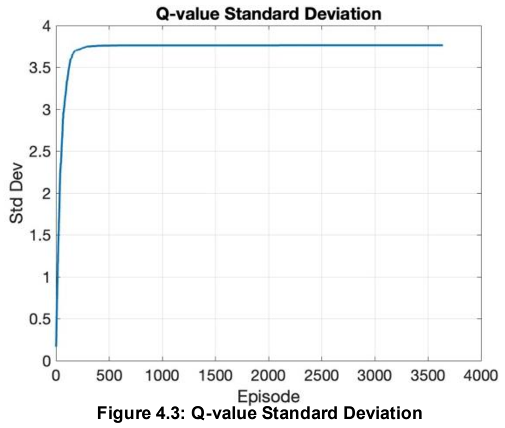


Figure 4.3: Q-value Standard Deviation

Conclusion

This report examined various machine learning techniques for robotics, including regression, classification, transfer learning, and reinforcement learning. Through systematic experiments, we analysed optimization strategies, activation functions, and hyperparameter choices.

For regression, we implemented backpropagation using SGD and ADAM, finding that ADAM converged faster while SGD required more epochs to reach similar accuracy. The MATLAB toolbox offered optimized convergence with minimal tuning.

In classification, MLP models trained with both SGD and ADAM achieved high accuracy, with the MATLAB toolbox providing faster convergence. Transfer learning experiments showed that GoogLeNet outperformed AlexNet due to its deeper architecture, highlighting the benefits of leveraging pre-trained models.

Reinforcement learning with Q-learning successfully trained an agent to navigate a grid world optimally. Hyperparameter tuning demonstrated the impact of learning rate, discount factor, and epsilon decay on training efficiency and policy effectiveness.

Overall, this report highlights the importance of model selection, hyperparameter tuning, and computational efficiency. Future work could explore more complex environments and architectures to further improve performance in robotics applications.