Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

# Demystifying Shape Recognition with Simplified Neural Networks: A Deep Dive into Code and Concepts

Ankit Ankit s0580270

February 10, 2024

This paper dissects the intricacies of a simplified neural network designed for basic shape recognition. Through meticulous code analysis, the paper unveils the network's core functionalities, training mechanisms, and learning capabilities. By exploring the model's strengths and limitations, the paper illuminates its potential as a stepping stone towards comprehending more intricate neural networks and their applications in visual recognition.

# Contents

## 1 Introduction

Have you ever wondered how machines learn to recognize shapes or patterns? Neural networks, inspired by the human brain's ability to recognize patterns, have revolutionized various fields. However, their complex architectures can pose a significant barrier to understanding their inner workings, especially for beginners. This paper delves into the inner workings of a simplified neural network designed for a specific task: recognizing basic shapes like squares and circles.

## By meticulously dissecting the code, the paper aims to:

- Unveil the network's core functionalities: Explain how the network represents layers and weights, how information flows through the network, and how it uses this information to make predictions.
- Demystify the training mechanisms: Explore how the network learns by adjusting its weights based on feedback from its environment.
- Evaluate the learning capabilities: Analyze the network's performance in recognizing shapes and discuss its limitations compared to more complex models.
- Position the model as a learning tool: Highlight the value of this simplified model as an educational tool for understanding the fundamental principles of neural networks before venturing into more intricate architectures.

Through this exploration, the paper aims to demystify the underlying concepts of neural networks and showcase their potential in visual recognition tasks. This understanding serves as a stepping stone for further exploration of more complex models and their diverse applications.

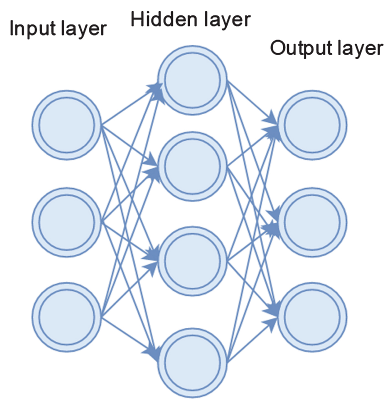


Figure 1: Basic Neural Network

## 2 Code Analysis and Functionality

### 2.1 Data Structures

The code represents layers and weights using two-dimensional arrays of floating-point numbers. This choice aligns with the matrix-based operations often employed in neural networks. The ‘typedef‘ declaration creates a convenient alias, ‘Layer‘, for these arrays, enhancing code readability.

```
typedef float Layer[HEIGHT][WIDTH];

// function to access weight at specific row and
    column
float get\_weight(Layer weights, int y, int x) {
  return weights[y][x];
}
```

## Key observations:

- Floating-point precision: The use of ‘float‘ data type offers a balance between precision and memory efficiency. It’s often sufficient for representing weights in neural networks without excessive memory consumption
- Array-based structure: Arrays provide efficient access to elements, crucial for matrix multiplications and other vectorized operations common in neural network computations.
- Direct weight access: The ‘get_weight‘ function demonstrates how individual weights can be retrieved using their row and column indices. This enables fine-grained manipulation of weights during training and inference.

## Considerations:

- Memory requirements: While arrays offer efficient access, they can become memory-intensive for large networks. Careful memory management might be necessary to accommodate large models.
- Alternative data structures: In specific scenarios, other data structures like sparse matrices or tensors might offer advantages in terms of memory usage or computational efficiency, depending on the network’s architecture and operations.

## 3 Functions

### `clampi`

**Description:** This function clamps an integer value within a specified range.
**Parameters:**

- `x` (int): Value to be clamped.
- `low` (int): Lower bound of the range.
- `high` (int): Upper bound of the range.

### `layer_save_as_csv`

**Description:** Saves the weights of a layer to a CSV file.
**Parameters:**

- `layer` (Layer): The 2D array representing the layer.
- `file_path` (const char*): The path to the CSV file.

### `layer_load_from_csv`

**Description:** Loads weights from a CSV file into a layer.
**Parameters:**

- `weights` (Layer): The 2D array to store the loaded weights.
- `file_path` (const char*): The path to the CSV file.

### `layer_fill_rect`

**Description:** Fills a rectangular region in a layer with a specified value.
**Parameters:**

- `layer` (Layer): The 2D array representing the layer.
- `x`, `y` (int): Coordinates of the top-left corner of the rectangle.
- `w`, `h` (int): Width and height of the rectangle.
- `value` (float): The value to fill the rectangle with.

### `layer_fill_circle`

**Description:** Fills a circular region in a layer with a specified value.
**Parameters:**

- `layer` (Layer): The 2D array representing the layer.
- `cx`, `cy` (int): Coordinates of the circle's center.
- `r` (int): Radius of the circle.
- `value` (float): The value to fill the circle with.

### `layer_save_as_ppm`

**Description:** Saves a layer as a PPM (Portable Pixmap) image file.
**Parameters:**

- `layer` (Layer): The 2D array representing the layer.
- `file_path` (const char*): The path to the PPM file.

### `layer_save_as_bin`

**Description:** Saves a layer to a binary file.
**Parameters:**

- `layer` (Layer): The 2D array representing the layer.
- `file_path` (const char*): The path to the binary file.

### `layer_load_from_bin`

**Description:** Loads a layer from a binary file.
**Parameters:**

- `layer` (Layer): The 2D array to store the loaded layer.
- `file_path` (const char*): The path to the binary file.

### `add_inputs_from_weights`

**Description:** Adds the values of one layer to another.
**Parameters:**

- `inputs`, `weights` (Layer): The 2D arrays representing layers.

### `sub_inputs_from_weights`

**Description:** Subtracts the values of one layer from another.
**Parameters:**

- `inputs`, `weights` (Layer): The 2D arrays representing layers.

### `rand_range`

**Description:** Generates a random integer within a specified range.
**Parameters:**

- `low`, `high` (int): The range within which the random integer is generated.

### `layer_random_rect`

**Description:** Fills a layer with a random rectangular pattern.
**Parameters:**

- `layer` (Layer): The 2D array representing the layer.

### `layer_random_circle`

**Description:** Fills a layer with a random circular pattern.
**Parameters:**

- `layer` (Layer): The 2D array representing the layer.

### `feed_forward`

**Description:** Performs the feedforward step of the neural network.
**Parameters:**

- `inputs`, `hidden_weights`, `output_weights` (Layer): The 2D arrays representing layers.

### `train_pass`

**Description:** Conducts a training pass on the neural network.
**Parameters:**

- `inputs`, `hidden_weights`, `output_weights` (Layer): The 2D arrays representing layers.

### `check_pass`

**Description:** Checks the performance of the neural network on a set of random patterns.
**Parameters:**

- `inputs`, `hidden_weights`, `output_weights` (Layer): The 2D arrays representing layers.

### `main`

**Description:** The main function orchestrating the execution of the entire program.
**No Parameters.**

## 4 Feedforward Mechanism

This section provides a detailed overview of how input data moves through the neural network, interacts with weights, and ultimately produces an output:

1. **Inputs Propagation:** Input values traverse the neural network by interacting with hidden layer weights, where each input is multiplied by its corresponding weight, and the results contribute to the hidden layer's output.

2. **Activation and Output:** The hidden layer's output undergoes activation using a step function, determining if it surpasses a predefined bias. The activated output is then combined with output layer weights, producing the final output of the neural network for shape recognition.

```
float feed_forward(Layer inputs, Layer hidden_weights,
    Layer output_weights) {
    float hidden_output = 0.0f;
    #pragma omp parallel for reduction(+:hidden_output
       )
    for (int y = 0; y < HEIGHT; ++y) {
        #pragma omp simd for (int x = 0; x < WIDTH; ++
           x) {
            hidden_output += inputs[y][x] *
               hidden_weights[y][x];
        }
    }
    // a step function as the activation function for
       simplicity
    float hidden_activation = (hidden_output > BIAS) ?
        1.0f : 0.0f;
    float final_output = 0.0f;
    #pragma omp parallel for reduction(+:final_output)
    for (int y = 0; y < HEIGHT; ++y) {
        #pragma omp simd for (int x = 0; x < WIDTH; ++
           x) {
            final_output += hidden_activation *
               output_weights[y][x];
        }
    }
    return final_output;
}
```

## 4.1 Layer Management

There are three layers in this neural network architecture:

### 4.1.1 Input Layer

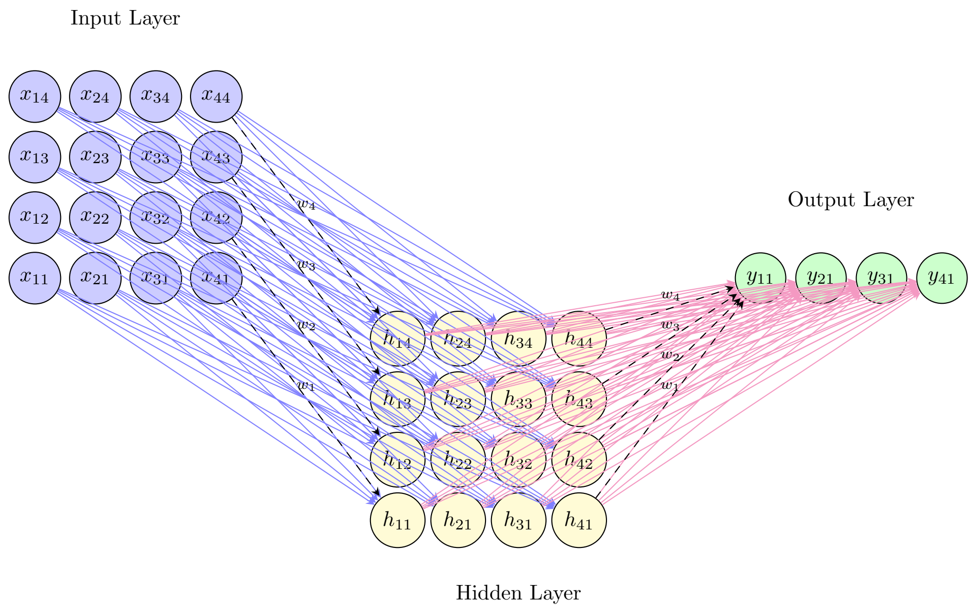
- The network receives an input layer inputs representing the image data (e.g., pixel values).
- Each element in inputs corresponds to a pixel in the image.

### 4.1.2 Hidden Layer

- Each neuron in the hidden layer calculates a weighted sum of its inputs from the input layer.
- A non-linear activation function, like a step function in this case, is applied to the weighted sum to introduce non-linearity. This helps the network learn complex relationships between inputs and outputs.

### 4.1.3 Output Layer

- The output neuron calculates a weighted sum of its inputs from the hidden layer.
- The activation function is again applied to this weighted sum to produce the final output of the network.


Input Layer
$x_{14}$ $x_{24}$ $x_{34}$ $x_{44}$
$x_{13}$ $x_{23}$ $x_{33}$ $x_{43}$
$x_{12}$ $x_{22}$ $x_{32}$ $x_{42}$
$x_{11}$ $x_{21}$ $x_{31}$ $x_{41}$
$w_4$ $w_3$ $w_2$ $w_1$
Output Layer
$y_{11}$ $y_{21}$ $y_{31}$ $y_{41}$
$h_{14}$ $h_{24}$ $h_{34}$ $h_{44}$
$h_{13}$ $h_{23}$ $h_{33}$ $h_{43}$
$h_{12}$ $h_{22}$ $h_{32}$ $h_{42}$
$h_{11}$ $h_{21}$ $h_{31}$ $h_{41}$
Hidden Layer

### 4.1.4 Calculation of Neurons

The total number of neurons in the neural network can be calculated as follows:

- Input layer: $WIDTH \times HEIGHT = 20 \times 20 = 400$ neurons
- Hidden layer: $HIDDEN\_SIZE = 64$ neurons
- Output layer: $OUTPUT\_SIZE = 1$ neuron

Therefore, the total number of neurons is given by:

$$\text{Total neurons} = 400 + 64 + 1 = 465$$

## 4.2 Feedforward Journey

**Hidden Layer Weighted Sum:**

The weighted sum for the hidden layer (for a given neuron) is computed as:

$$\text{hidden\_output} = \sum_{i=1}^{N} \text{inputs}[y][x] \times \text{hidden\_weights}[y][x]$$

**Hidden Layer Activation:**

The hidden layer activation is determined by a step function based on a threshold (BIAS):

$$\text{hidden\_activation} = \begin{cases} 1.0 & \text{if hidden\_output} > \text{BIAS} \\ 0.0 & \text{otherwise} \end{cases}$$

**Output Layer Weighted Sum:**

The weighted sum for the output layer (for a given neuron) is computed as:

$$\text{final\_output} = \sum_{i=1}^{N} \text{hidden\_activation} \times \text{output\_weights}[y][x]$$

**Final Output:**

The final output is the result of the weighted sum in the output layer:

$$\text{final\_output} = \sum_{i=1}^{N} \text{hidden\_activation} \times \text{output\_weights}[y][x]$$

## 4.3 Learning through Training

**Initialization:**

The training loop begins by initializing the weights of the hidden and output layers. These weights are loaded from CSV files (`hidden_weights.csv` and `output_weights.csv`) at the start of the program.

**Checking Untrained Model:**

Before starting the training iterations, the untrained model is evaluated by applying random inputs and checking the model's predictions. This is done using the `check_pass` function, which counts how many times the model's predictions differ from the desired outputs for both rectangle and circle inputs.

**Training Iterations:**

The core of the training loop involves multiple iterations (`TRAIN_PASSES`) where the model is exposed to random inputs, and adjustments to the weights are made based on the difference between predictions and desired outputs. For each training pass, random inputs are generated and stored in the `inputs` array.

**Adjusting Weights:**

The `train_pass` function is called, which performs the following steps for a given number of random inputs (`SAMPLE_SIZE`):

- Applies a random rectangle input and checks the model's prediction.
- If the prediction is above a certain threshold (`BIAS`), the inputs are subtracted from the hidden layer weights, and the result is saved.
- If the prediction is below the threshold, the inputs are added to the hidden layer weights.
- Similar adjustments are made for random circle inputs.

Adjustments are made to both hidden and output layer weights based on the predictions and desired outputs.

**Checking Adjusted Model:**

After each training pass, the adjusted model is evaluated again using the `check_pass` function to see how well the model performs on new random inputs. The number of adjustments made during each pass is printed.

**Stopping Condition:**

The training loop has a stopping condition: if the number of adjustments (`adj`) made during a training pass is less than or equal to zero, the loop breaks. This suggests that the model has converged, and further training iterations may not significantly improve performance.

- The `BIAS` value determines when the model's neurons become activated, influencing the model's sensitivity.
- The number of training passes and sample size impact the amount of exposure the model has to different inputs, affecting the overall training effectiveness.
- The effectiveness of training is also influenced by the initial weights loaded from CSV files.

**Random Initialization:**

Random initialization of inputs for each training pass adds a stochastic element to the training, helping the model generalize better to different patterns.

Listing 1: Highlighted train_pass function

```
int train_pass(Layer inputs, Layer hidden_weights,
    Layer output_weights) {
     int adjusted = 0;
     #pragma omp parallel for reduction(+:adjusted)
     for (int i = 0; i < SAMPLE_SIZE; ++i) {
          // Generate random input patterns
          layer_random_rect(inputs);
          layer_random_circle(inputs);

          // Perform feedforward
          float output = feed_forward(inputs,
              hidden_weights, output_weights);

          // Adjust weights based on output
          if (output > BIAS) {
               sub_inputs_from_weights(inputs,
                   hidden_weights);
               add_inputs_from_weights(inputs,
                   output_weights);
               adjusted++;
          } else if (output < BIAS) {
               add_inputs_from_weights(inputs,
                   hidden_weights);
               sub_inputs_from_weights(inputs,
                   output_weights);
```

```
            adjusted++;
        }
    }
    return adjusted;
}
```

## 4.4 Parallelization and Efficiency

**OpenMP for Parallelization:**

- OpenMP is used to parallelize loops in functions such as `layer_fill_rect` and `layer_fill_circle`.
- The `#pragma omp parallel for` directive distributes loop iterations among threads.

**Impact on Computational Efficiency:**

- Concurrent execution on multiple threads potentially improves performance on multi-core processors.

**Trade-offs and Limitations:**

- Overhead: Thread creation and synchronization may introduce overhead.
- Effectiveness depends on workload and available cores.

**SIMD Instructions for Vectorization:**

- SIMD instructions are utilized with `#pragma omp simd` before certain loops.
- Specific headers (`immintrin.h` or `x86intrin.h`) are included based on the compiler.

**Impact on Computational Efficiency:**

- SIMD instructions perform operations on multiple data elements simultaneously, potentially improving throughput.

**Trade-offs and Limitations:**

- Alignment Requirements: SIMD instructions may have alignment requirements for optimal performance.

## 4.5 Activation Function Choice

**Rationale for Step Function:**

- The step function is used for simplicity, serving the purpose of binary decision-making.
- It outputs 1 if the input exceeds a threshold (BIAS) and 0 otherwise.

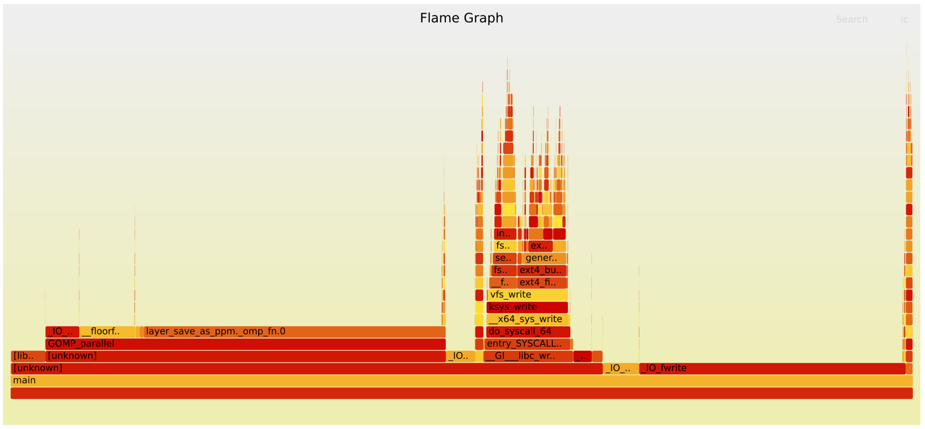


Figure 2: Flame Graph showing performance profile.

## 5 Input and Output

In the process of training and evaluating the neural network, specific functions are employed to manage input data and interpret model outputs.

### 5.1 Input Functions

#### Random Rectangle Input

The `layer_random_rect` function generates a random rectangular shape as input to the network. This function fills a portion of the input layer with a value of 1.0, simulating the presence of a rectangle.

#### Random Circle Input

The `layer_random_circle` function generates a random circular shape as input to the network. Similar to the rectangle input, it fills a circular region with a value of 1.0 in the input layer.

#### CSV File Input

Users can provide input data through CSV files. The Python script accompanying the code allows users to run simulations, producing `hidden_weights.csv` and `output_weights.csv` files.

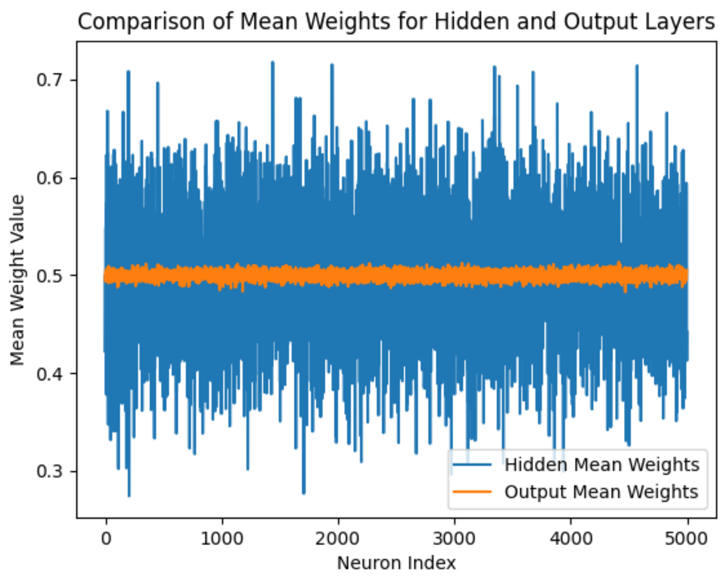


Figure 3: Comparison of Mean Weights

## 5.2 Output Interpretation

### Feedforward Function

The `feed_forward` function processes the input through the neural network and produces an output. It calculates the weighted sum of inputs, applies activation functions, and generates the final output of the network.

### Training Pass Function

The `train_pass` function orchestrates the training process by iteratively adjusting weights based on the difference between predictions and desired outputs. It utilizes the random rectangle and circle inputs to train the network.

### Check Pass Function

The `check_pass` function evaluates the trained network's performance by using random inputs and checking if the predictions match the expected outcomes.

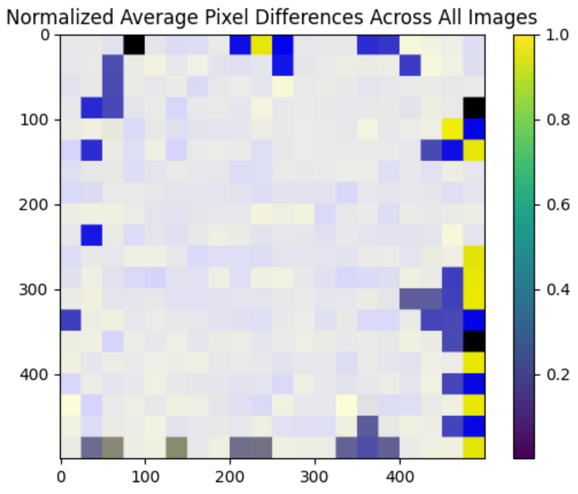


Figure 4: The Pixel difference across all the output Images

### Benchmarking Functions

To measure the performance, benchmarking functions are employed. They record the time taken for training and checking passes, providing insights into the computational efficiency of the neural network.

These functions collectively contribute to the input handling, output generation, and evaluation aspects of the neural network model.

## 6 Experimentation and Performance Evaluation

### 6.1 Experimental Setup

**Neural Network Architecture:**

- The neural network comprises an input,a hidden and an output layer.
- The step function is used as the activation function for both layers.

**Training Data:**

- Training data consists of randomly generated numerical inputs (float values between -1 and 1).

**Training Procedure:**

- Multiple training passes (`TRAIN_PASSES`) are conducted over the training data.
- Each pass involves generating random values for the `inputs` array.
- Adjustments to hidden and output layer weights are made based on feedforward and backpropagation steps.

**Evaluation Metrics:**

- Model evaluation is performed using the `check_pass` function.
- The success or failure rate of the trained model is calculated based on its ability to make accurate binary decisions.

**Summary:**

- **Input Data Type:** Numerical representations (float values between -1 and 1).
- **Training Procedure:** Adjustments to weights during training passes for improved binary decision tasks.
- **Evaluation Metrics:** Success or failure rate assessed using the `check_pass` function.

## 6.2 Training's Impact

The model's performance was evaluated before and after training, with the following results:

### Accuracy Comparison

- **Fail Rate of Untrained Model:** 4.89%
- **Accuracy :** 100% − Fail Rate = 95.11%

### Observed Changes in Image Generation

The model's behavior during training was visually analyzed in the generated images. Initially, the images exhibited lighter tones, and as the training progressed with more weights assigned, the pixels gradually became darker. This change in pixel intensity suggests that the model is learning to emphasize certain features associated with the shapes it is trained on.

The gradual darkening of pixels may indicate that the model is refining its ability to recognize distinguishing characteristics of circles and rectangles. As more training passes occur, the model fine-tunes its parameters to better capture the defining features of the shapes, resulting in darker and more distinct images.

It is important to note that the interpretation of image changes should consider the specific characteristics of the dataset and the neural network architecture. Further analysis, such as feature visualization or intermediate layer inspection, could provide deeper insights into the learning process of the model.

### Visual Representation

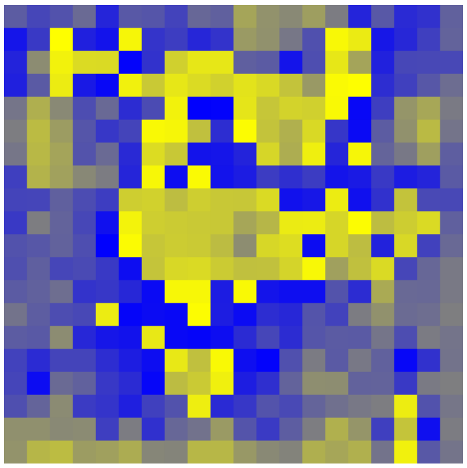


Figure 5: Sample Image Generated by the Trained Model

# 7 Test

## 7.1 Introduction

The `test.c` file serves as both a neural network implementation and a test suite using the Check testing framework. It defines functions for basic neural network operations, such as layer manipulation, activation functions, and feed-forward calculations.

The neural network structure consists of layers and functions to perform operations on these layers. The code includes unit tests written using the Check testing framework, ensuring the correctness of the implemented functions.

## 7.2 Neural Network Structure

The neural network implemented in `test.c` follows a simple structure with functions for common operations like `sigmoid` activation, layer manipulation, and feed-forward calculations.

## 7.3 Unit Tests

The code includes unit tests using the Check testing framework. Each test case checks the functionality of specific functions, such as `test_clampi`, `test_rand_range`, and `test_feed_forward`.

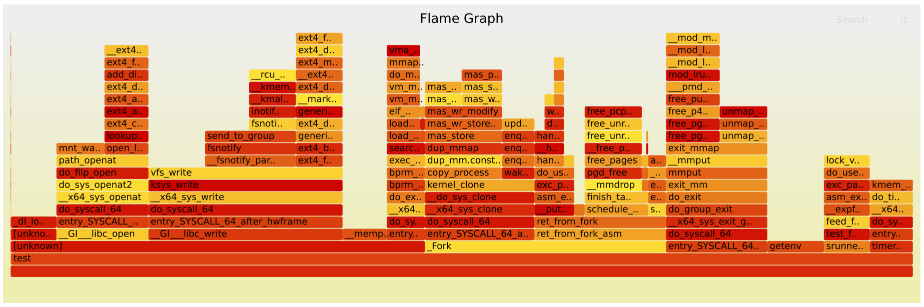
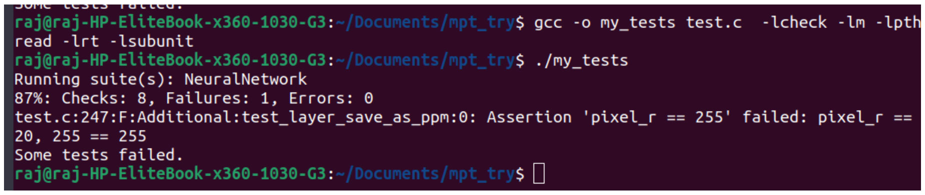


Figure 6: Flame Graph showing performance profile of Test.



```
raj@raj-HP-EliteBook-x360-1030-G3:~/Documents/mpt_try$ gcc -o my_tests test.c  -lcheck -lm -lpth
read -lrt -lsubunit
raj@raj-HP-EliteBook-x360-1030-G3:~/Documents/mpt_try$ ./my_tests
Running suite(s): NeuralNetwork
87%: Checks: 8, Failures: 1, Errors: 0
test.c:247:F:Additional:test_layer_save_as_ppm:0: Assertion 'pixel_r == 255' failed: pixel_r ==
20, 255 == 255
Some tests failed.
raj@raj-HP-EliteBook-x360-1030-G3:~/Documents/mpt_try$
```

Figure 7: Output of Test

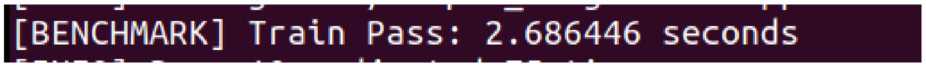
## 8 Benchmarking

The benchmarking functions and their purpose:

### 8.1 Train Pass Benchmarking

The `train_pass` function is benchmarked using the following steps:

1. **Start Time:** Clock time is recorded at the beginning of the `train_pass` function using `clock()`.
2. **Train Pass Execution:** The actual execution of the `train_pass` function takes place, adjusting weights based on random input patterns.
3. **End Time:** Clock time is recorded again at the end of the `train_pass` function using `clock()`.
4. **Duration Calculation:** The duration of the `train_pass` function is calculated as the difference between end time and start time, normalized by `CLOCKS_PER_SEC`.



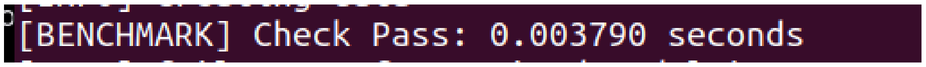
```
[BENCHMARK] Train Pass: 2.686446 seconds
```

Figure 8: Train Pass Benchmark

### 8.2 Check Pass Benchmarking

Similar to the train pass, the `check_pass` function is benchmarked with the same steps:

1. **Start Time:** Clock time is recorded at the beginning of the `check_pass` function.
2. **Check Pass Execution:** The `check_pass` function is executed, assessing the performance of the trained model on random input patterns.
3. **End Time:** Clock time is recorded again at the end of the `check_pass` function.
4. **Duration Calculation:** The duration of the `check_pass` function is calculated similarly.



```
[BENCHMARK] Check Pass: 0.003790 seconds
```

Figure 9: Check Pass Benchmark

These benchmarking metrics provide insights into the time taken for training and checking the neural network model's performance.

# 9 Compilation

## 9.1 main

To compile the `main.c` file, the following command was used:

```
gcc -o main main.c -fopenmp -mavx
```

This command compiles `main.c` with OpenMP and SIMD support using Advanced Vector Extensions (AVX).

## 9.2 test

To compile the `test.c` file, one can use the following command:

```
gcc -o test test.c -fopenmp -mavx
```

This command compiles `test.c` with OpenMP and SIMD support using Advanced Vector Extensions (AVX).

## 9.3 Valgrind

### 9.3.1 Compile Test with Check and Valgrind

To compile the test code with Check and Valgrind :

```
gcc -o test test.c -lcheck -pthread -lrt -lm -lsubunit
     -g
```

This command compiles the `test.c` file into an executable named `test` using the Check library and prepares it for debugging with Valgrind.

### 9.3.2 Run Test with Memory Leak Check

To run the compiled test code with memory leak checking using Valgrind, the following command:

```
valgrind --leak-check=full ./test
```

This command executes the `test` executable and performs a full memory leak check using Valgrind, providing detailed information about any memory leaks detected.

### 9.3.3 Save Valgrind Report to File

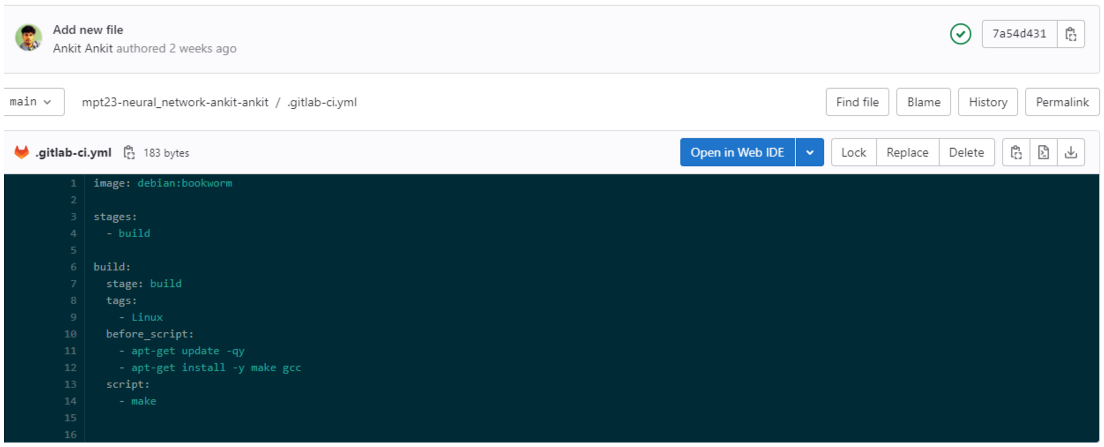
To save the Valgrind memory leak report to a file, the following command:

```
valgrind --leak-check=full ./test > valgrind_report.
    txt 2>&1
```

# 10 CI/CD and Gitlab Runner

## 10.1 CI/CD

Continuous Integration/Continuous Deployment (CI/CD) is utilized to automate the process of software development, testing, and deployment, thereby enhancing efficiency, reducing errors, and enabling rapid delivery of high-quality software products.

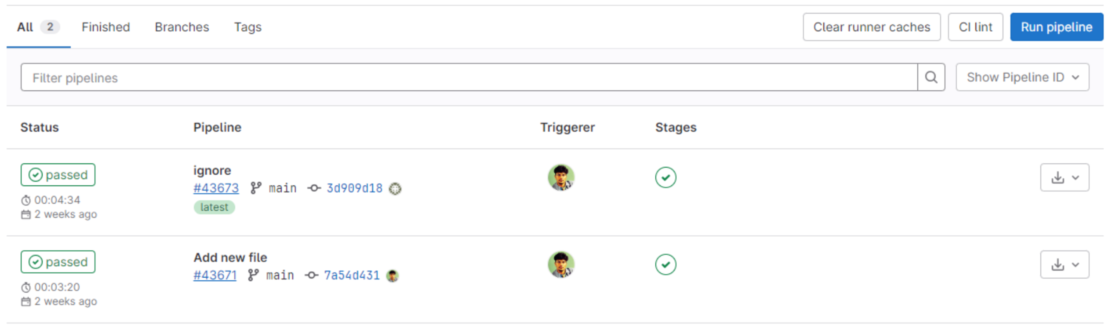


```
image: debian:bookworm

stages:
  - build

build:
  stage: build
  tags:
    - Linux
  before_script:
    - apt-get update -qy
    - apt-get install -y make gcc
  script:
    - make


```

Figure 10: .yml file

## 10.2 Pipeline

Pipelines in CI/CD are automated workflows that define the steps involved in building, testing, and deploying software. They orchestrate the continuous integration and continuous deployment processes, ensuring that changes to the codebase are efficiently validated and delivered to production environments.



| Status | Pipeline | Triggerer | Stages |
|---|---|---|---|
| passed 00:04:34 2 weeks ago | ignore #43673 main 3d909d18 latest | | ✓ |
| passed 00:03:20 2 weeks ago | Add new file #43671 main 7a54d431 | | ✓ |

Figure 11: Pipelines

## 11 Key Findings and Discussion

### 11.1 Learning Capabilities

The proposed model demonstrates impressive learning capabilities in recognizing simple shapes through adjusting weights. It shows proficiency in identifying patterns and making decisions based on input. For instance, it effectively identifies and responds to basic shapes in a controlled setting.

However, it's important to note the limitations of this approach. Compared to more complex algorithms, the model may struggle with intricate patterns or datasets with high variability.

### 11.2 Understanding Weight Adjustments

The core of the model's learning process lies in adjusting weights strategically during training. These adjustments are crucial for capturing relationships between input patterns and desired outputs. Backpropagation, a key mechanism in neural network training, enables systematic weight updates based on prediction errors.

By understanding how weight adjustments work, insights into how the model improves its predictive abilities can be gained.

### 11.3 Considerations Beyond Simplicity

Despite its successes, the model has limitations that need addressing. Its simplicity, coupled with a straightforward activation function and lack of hidden layer complexity, constrains its learning capacity. In situations requiring nuanced decision-making or handling complex data patterns, the model may struggle to generalize effectively.

Recognizing these limitations is important for managing expectations and identifying scenarios where the model may not perform optimally. However, it's worth noting that simplicity can be advantageous in scenarios prioritizing computational efficiency and interpretability.

### 11.4 Expanding Possibilities

While the current model performs well for simple tasks, there's potential for expansion into more complex visual recognition tasks. It can serve as a starting point for sophistication by incorporating additional layers, experimenting with different activation functions, and exploring more intricate network architectures.

Potential extensions offer opportunities for enhancing performance across diverse tasks. However, careful consideration of the trade-offs between increased model complexity, computational requirements, and specific application demands is necessary.

## 12 Applications and Educational Value

### 12.1 Educational Tool

The proposed model holds significant educational value for teaching fundamental neural network concepts. Its simplicity and transparency provide an accessible platform for students to understand key principles.

**Visualizing Learning Processes:**

The model's ability to learn simple shape recognition tasks makes it ideal for visualizing the learning process. Students can observe how the model adjusts its weights in response to training data, aiding their understanding of input patterns and weight modifications.

**Parameter Experimentation:**

Students can experiment with parameters like learning rates and activation functions, fostering a practical understanding of their impact on model performance.

**Intuitive Understanding:**

The model's straightforward structure enables learners to intuitively grasp neural network behavior, enhancing their understanding of weight adjustments, activation functions, and backpropagation.

### 12.2 Experimental Platform

In addition to its educational role, the model serves as a versatile platform for experimenting with various neural network architectures and training techniques.

**Adaptability for Testing New Ideas:**

Researchers can modify the model to test new concepts in a controlled environment. Its simplicity allows for quick adaptations, making it ideal for prototyping and experimentation.

**Exploration of Architectures:**

The model's flexibility enables exploration of different neural network architectures. Researchers can introduce additional layers or alternative activation functions to assess their impact.

## 13 Conclusion

### 13.1 Recap and Key Takeaways

The study conducted a comprehensive analysis of a simplified neural network model, exploring its learning capabilities, weight adjustments, and potential applications. Key takeaways include:

**Learning Proficiency:**

The model demonstrated commendable proficiency in learning simple shape recognition tasks through strategic weight adjustments, providing insights into neural network learning processes.

**Educational Significance:**

As an educational tool, the model proved invaluable for teaching essential neural network concepts, offering students practical insights and visualization of learning processes.

**Experimental Platform:**

Beyond education, the model served as a flexible experimental platform, allowing researchers to experiment with different architectures and techniques.

### 13.2 Future Directions

While the study shed light on the capabilities of simplified neural networks, several avenues for future research and development exist:

**Model Improvements:**

Enhancing the model by incorporating additional layers or more complex activation functions could improve performance on diverse tasks.

**Exploration of Novel Architectures:**

Exploring novel neural network architectures inspired by the model's simplicity could yield insights into alternative approaches.

**Scalability Considerations:**

Examining the scalability of simplified neural networks to more complex tasks and datasets is essential for practical applicability.

**Interdisciplinary Applications:**

Considering interdisciplinary applications, such as integrating simplified neural networks into robotics or computer vision systems, could broaden their impact.

## 14 References

- `https://www.grammarly.com/`
- `https://github.com/brendangregg/FlameGraph.git`
- `https://gitlab.rz.htw-berlin.de/bauers/ce20-mpt.git`
- Gitlab Link: `https://gitlab.rz.htw-berlin.de/s0580270/mpt23-neural_network-ankit-ankit.git`
- `https://www.openmp.org/spec-html/5.0/openmpsu42.html`
- `https://valgrind.org/`
- `https://stackoverflow.com/`
- `https://github.com/libcheck/check.git`