Table of content

Machine Unlearning: A Neural Network Learning Backwards

1.0 Introduction

1.1 What is Machine Unlearning?

Machine unlearning can be described as a procedure where the effects of the training data are removed from an already trained model. Given a target model, the goal of unlearning is to obtain an unlearned model such that its performance is at least equivalent to, or behaves like a re-trained model, which is trained on exactly all the data as the target model but without the insights to be unlearned.

The area of machine unlearning is still in its developing phase and faces many barriers. It may turn out to be difficult to compromise a balance between removing some of the data's influences and maintaining overall performance of the model. Besides, various algorithms in machine learning suffer from different unlearning difficulties. However, “Machine Unlearning” has huge potential in enhancing robustness, reliability, and morality of machine learning models.

1.2 The Emergence of Machine Unlearning.

The idea of **“Machine Unlearning: A Neural Network Learning Backwards”** was found accidentally during a standard debugging session. The original goal was to take a simple feedforward neural network and use it to classify the MNIST digits dataset, a very common classification task in deep learning and a popular benchmark on platforms like Kaggle. Designed with ReLU activation in the a hidden layer and an output layer with softmax, the model acted in a behaviour different than what was expected. The network started off relatively well, but with every epoch, instead of training actually improving the model's performance, the network continued to get worse. What began as frustration due to unexpected results slowly turned into curiosity.

While this degrading in model accuracy (as seen in *nn.c*) was accidental, it brings out some fundamental insights into the way learning and forgetting take place in neural networks. This inspires possible future research into “controlled forgetting mechanisms” whereby a model would willingly 'forget' certain patterns or data points without degrading its overall performance. This project is a very rare and creative observation of reverse learning tendencies that adds fun and technical interest to AI research. It underlines the possibility of exploring innovative applications further that let models forget something that has become outdated or sensitive data to make AI development comply with ethics and privacy standards. What seemed like an unexpected challenge in debugging turned out to be an interesting contribution toward further understanding and leveraging the intricacies of machine learning behavior.

Link to Repository: https://github.com/yemmi408/machine-unlearning

2.0 Methodology

2.1 Neural Network Architecture

The neural network used for this project is a three-layer network:

- **Input Layer:** There are 784 nodes in the layer, one for every pixel in the 28x28 MNIST image.
- **Hidden Layer:** This includes 256 nodes.
- **Output Layer:** This has 10 nodes, each representing MNIST digit from 0 to 9.
- **Activation function:** ReLU for hidden layer, softmax for output layer.

2.2 Dataset

The project uses the well-known MNIST dataset for digit classification.

- Normalization of all images between pixel values range of 0 and 1 to ensure consistency.
- The data is split into two sets; 80% will go into training the models and 20% to test the performance.
- The data is shuffled to make the training more stable, such that it does not overfit some patterns.

2.3 Training Procedure

The model is trained using gradient descent with momentum in order to speed up convergence and help avoid local minima.

- **Hyperparameters:**
    - Learning Rate: 0.0005 (controls the step size during optimization).
    - Momentum: 0.9 (Added to stabilize updates by mixing in some previous gradients).
    - Epochs: 50 (number of times the whole dataset goes through the model).
    - Batch Size: 64 (The group of data points used in each step of training).
    - Mini-batch Size: 5 (The number of mini-batches to process sequentially for optimization before updating the parameters).
- Training is performed in mini-batches to balance computational efficiency and model accuracy.

2.4 Tools Used

- **Programming Language:** C was chosen for its simplicity and efficiency.
- **Compilation:** Git Bash with MSYS2 UCRT64 with GCC flags to makecomputation fast.
- **Libraries:** Used standard C libraries for mathematical computations and file handling.

- **Platform:** The code was developed and executed in Windows.

3.0 Code Analysis

1. Network Initialization

```
typedef struct {
    float *weights, *biases, *weight_momentum, *bias_momentum;
    int input_size, output_size;
} Layer;

typedef struct {
    Layer hidden, output;
} Network;

void init_layer(Layer *layer, int in_size, int out_size) {
    int n = in_size * out_size;
    float scale = sqrtf(2.0f / in_size);

    layer->input_size = in_size;
    layer->output_size = out_size;
    layer->weights = malloc(n * sizeof(float));
    layer->biases = calloc(out_size, sizeof(float));
    layer->weight_momentum = calloc(n, sizeof(float));
    layer->bias_momentum = calloc(out_size, sizeof(float));

    for (int i = 0; i < n; i++) {
        layer->weights[i] = ((float)rand() / RAND_MAX - 0.5f) * 2 * scale;
    }
}
```

This code block initializes a layer with weights, biases, and random values for training.

2. Forward Propagation

```
void forward(Layer *layer, float *input, float *output) {
    for (int i = 0; i < layer->output_size; i++) output[i] = layer->biases[i];
    for (int j = 0; j < layer->input_size; j++) {
        for (int i = 0; i < layer->output_size; i++) {
            output[i] += input[j] * layer->weights[j * layer->output_size + i];
```

```
        }
    }
    for (int i = 0; i < layer->output_size; i++) output[i] = fmaxf(0, output[i]); // ReLU
}
```

This code block uses weights, biases, and ReLU activation to compute output for a layer.

3. Backpropagation

```
void backward(Layer *layer, float *input, float *output_grad, float *input_grad, float lr) {
    for (int j = 0; j < layer->input_size; j++) {
        float *weight_row = &layer->weights[j * layer->output_size];
        for (int i = 0; i < layer->output_size; i++) {
            float grad = output_grad[i] * input[j];
            weight_row[i] -= lr * grad;
        }
    }
    for (int i = 0; i < layer->output_size; i++) layer->biases[i] -= lr * output_grad[i];
}
```

This code block uses the error gradients and learning rate to update weights and biases.

4. Training the Network

```
void train(Network *net, float input[5][784], int *labels, float lr) {
    float hidden[5][256], output[5][10], grad[5][10];
    for (int i = 0; i < 5; i++) {
        forward(&net->hidden, input[i], hidden[i]);
        forward(&net->output, hidden[i], output[i]);
    }
    // Compute and backpropagate gradients (not shown for brevity)
}
```

This code block calculates errors, processes a mini-batch, and updates the network through backpropagation.

5. Main Function

```
int main() {
```

```
    Network net;
    init_layer(&net.hidden, 784, 256);
    init_layer(&net.output, 256, 10);

    // Load MNIST dataset, train the model, and evaluate accuracy
    return 0;
}
```

This code block involves initializing the network, training it on MNIST dataset and generates **test accuracy.**

4.0 Results

The accuracy shows a steady decrease as the epochs progress:

```
User@DESKTOP-QNLNOO7 MINGW64 ~
$ cd machine-unlearning

User@DESKTOP-QNLNOO7 MINGW64 ~/machine-unlearning (main)
$ gcc -O3 -march=native -ffast-math -o nn nn.c -lm

User@DESKTOP-QNLNOO7 MINGW64 ~/machine-unlearning (main)
$ ./nn
Epoch 1, Accuracy: 81.38%, Time: 17.55 seconds
Epoch 2, Accuracy: 80.48%, Time: 17.67 seconds
Epoch 3, Accuracy: 77.79%, Time: 18.27 seconds
Epoch 4, Accuracy: 76.98%, Time: 18.28 seconds
Epoch 5, Accuracy: 74.73%, Time: 17.85 seconds
Epoch 6, Accuracy: 73.15%, Time: 17.77 seconds
Epoch 7, Accuracy: 72.06%, Time: 17.95 seconds
Epoch 8, Accuracy: 71.60%, Time: 18.46 seconds
Epoch 9, Accuracy: 67.78%, Time: 17.73 seconds
Epoch 10, Accuracy: 63.92%, Time: 17.53 seconds
Epoch 11, Accuracy: 60.68%, Time: 17.98 seconds
Epoch 12, Accuracy: 58.82%, Time: 17.59 seconds
Epoch 13, Accuracy: 59.13%, Time: 17.90 seconds
Epoch 14, Accuracy: 58.46%, Time: 17.75 seconds
Epoch 15, Accuracy: 55.08%, Time: 17.94 seconds
Epoch 16, Accuracy: 52.08%, Time: 17.79 seconds
Epoch 17, Accuracy: 50.83%, Time: 17.90 seconds
Epoch 18, Accuracy: 51.38%, Time: 17.97 seconds
Epoch 19, Accuracy: 48.77%, Time: 17.97 seconds
Epoch 20, Accuracy: 49.56%, Time: 17.74 seconds
Epoch 21, Accuracy: 47.86%, Time: 18.09 seconds
Epoch 22, Accuracy: 46.95%, Time: 17.66 seconds
Epoch 23, Accuracy: 49.69%, Time: 17.16 seconds
Epoch 24, Accuracy: 46.69%, Time: 17.13 seconds
Epoch 25, Accuracy: 44.65%, Time: 17.04 seconds
Epoch 26, Accuracy: 45.38%, Time: 17.57 seconds
Epoch 27, Accuracy: 45.01%, Time: 16.98 seconds
Epoch 28, Accuracy: 47.85%, Time: 17.22 seconds
Epoch 29, Accuracy: 44.97%, Time: 17.67 seconds
Epoch 30, Accuracy: 46.56%, Time: 17.63 seconds
Epoch 31, Accuracy: 45.60%, Time: 17.23 seconds
Epoch 32, Accuracy: 45.88%, Time: 17.46 seconds
Epoch 33, Accuracy: 45.80%, Time: 17.60 seconds
Epoch 34, Accuracy: 45.09%, Time: 17.87 seconds
Epoch 35, Accuracy: 45.00%, Time: 17.71 seconds
Epoch 36, Accuracy: 45.93%, Time: 17.85 seconds
Epoch 37, Accuracy: 46.78%, Time: 18.02 seconds
Epoch 38, Accuracy: 44.28%, Time: 17.97 seconds
Epoch 39, Accuracy: 44.00%, Time: 17.83 seconds
Epoch 40, Accuracy: 44.22%, Time: 17.58 seconds
Epoch 41, Accuracy: 41.96%, Time: 17.98 seconds
Epoch 42, Accuracy: 41.07%, Time: 17.83 seconds
Epoch 43, Accuracy: 42.92%, Time: 17.94 seconds
Epoch 44, Accuracy: 42.77%, Time: 17.65 seconds
Epoch 45, Accuracy: 41.84%, Time: 18.93 seconds
Epoch 46, Accuracy: 41.13%, Time: 18.24 seconds
Epoch 47, Accuracy: 42.84%, Time: 17.70 seconds
Epoch 48, Accuracy: 42.31%, Time: 18.25 seconds
Epoch 49, Accuracy: 42.42%, Time: 17.91 seconds
Epoch 50, Accuracy: 41.70%, Time: 17.51 seconds
```

Although the network starts with strong performance, its accuracy suddenly declines, which indicates the presence of inherent problem in the training dynamics, thereby causing “unlearning” in the network.

5.0 Possible Causes of "Unlearning" in nn.c

Some possible issues that were causing this unlearning behavior of the model in *nn.c*, where the accuracy degraded with every passing epoch, were as follows:

1. **Data Shuffling Inconsistencies:**

- **Problem:** The bug is in the function shuffle_data in the file nn.c. The for loop for (int i = 0; i > n; i++) is incorrectly implemented, which resulted in the generation of unbalanced mini-batches due to improper shuffling of data. As a result, the model was not able to generalize well and overfitted to certain patterns.

```
void shuffle_data(unsigned char *images, unsigned char *labels, int n)
{
   for (int i = 0; i > n; i++) // Incorrect condition; this loop never runs
   {
      int r = rand() % (i + 1);
      for (int k = 0; k < INPUT_SIZE; k++)
      {
         exchImage(images, i, k, r);
      }
      exchLabel(labels, i, r);
   }
}
```

- **Effect on Unlearning:** Poor shuffling continues to train the network on unrepresentative subsets and thus develops biases that deteriorate the performance.

2. **Momentum Mismanagement:**

- **Problem:** In nn.c, when performing updates on weights, the momentum was dealt incorrectly; that's why the model became unstable. Since it accumulated gradients, without proper adjustments, the change in weights was amplified, leading to unstable training.

```
momentum_row[i] = MOMENTUM * momentum_row[i] + lr * grad;
weight_row[i] -= momentum_row[i]; // Updates weights using momentum
```

- **Effect on Unlearning:** This convergence, which was prevented by oscillations in weights, deteriorated the accuracy.

3. **Lack of Interim Performance Monitoring**:

- **Problem:** *nn.c* didn't track the training accuracy across mini-batches. Since performance trend of the network was assessed only at the end of an epoch, diagnosis wasn't able to be done easily during training.
- **Effect on Unlearning**: Without real-time feedback, the poor gradient updates were unintentionally ignored until their compounding caused considerable degradation in performance.

4. **Overfitting to Training Data**:

- **Problem**: The absence of mechanisms to address overfitting (e.g., validation during training or regularization techniques) caused the network to memorize patterns in the training data while performing poorly on unseen data.

```
int correct = 0;
for (int i = train_size; i < data.nImages; i++) // Only evaluates on test data
{
   for (int k = 0; k < INPUT_SIZE; k++)
      img2[k] = data.images[i * INPUT_SIZE + k] / 255.0f;
   if (predict(&net, img2) == data.labels[i])
      correct++;
}
```

- **Effect on Unlearning**: The model became progressively worse at generalization, manifesting as a steady decline in test accuracy.

All these issues were addressed in *debug.c* by removing shuffling, improving gradient updates, and logging training metrics, leading to stabilized accuracy.

6.0 Debugging and Enhancing Neural Network Performance

The first implementation in *nn.c* had some errors where the models accuracy decreased over epochs. During debugging, a number of changes were made and saved in separate C file titled *debug.c*; this resulted in a working model showing increasing accuracy over the epochs.

Key modifications in *debug.c* includes:

1. **Shuffling Data:**

- **Solution:** Data shuffling was removed to ensure consistency in input data and for easier debugging.

- **Effect**: Without shuffling, the debugging is simplified, and training becomes consistent and stable.

Removed code in debug.c:

```
shuffle_data(data.images, data.labels, data.nImages); // Removed
```

2. **Mini-Batch Training Updates:**

- **Solution:** The momentum formula was corrected so that training process would be stabilized.

```
momentum_row[i] = MOMENTUM * momentum_row[i] + lr * grad;

weight_row[i] -= momentum_row[i]; // Fixes weight updates
```

- **Effect:** Good momentum handles the oscillations of weights, therefore, the model converges.

3. **Accuracy Logging and Metrics:**

- **Solution:** The training accuracy after each mini-batch was also recorded to provide real-time insight into the learning dynamics of the network.

```
log_metrics(epoch, correct, data.nImages, cpu_time_used); // Logs training accuracy
```

- **Effect:** Can monitor performance in real time to identify problems while training and correct them accordingly for the network to learn as it should.

4. **Epochs and Training Focus:**

- **Solution:** Epochs was reduced from 50 in *nn.c* to 10 in *debug.c* with training accuracy evaluation used only for debugging purposes.

```
#define EPOCHS 10  // Reduced for debugging
log_metrics(epoch, correct, data.nImages, cpu_time_used); // Logs training accuracy
```

- **Effect:** Faster debugging and view performance immediately during training.

6.1 Accuracy after debugging

```
User@DESKTOP-QNLNOO7 MINGW64 ~/machine-unlearning (main)
$ gcc -O3 -march=native -ffast-math -o debug debug.c -lm

User@DESKTOP-QNLNOO7 MINGW64 ~/machine-unlearning (main)
$ ./debug
Epoch 1, Accuracy: 95.27%, Time: 16.71 seconds
Epoch 2, Accuracy: 98.31%, Time: 16.49 seconds
Epoch 3, Accuracy: 99.02%, Time: 15.53 seconds
Epoch 4, Accuracy: 99.41%, Time: 15.75 seconds
Epoch 5, Accuracy: 99.65%, Time: 16.82 seconds
Epoch 6, Accuracy: 99.79%, Time: 17.14 seconds
Epoch 7, Accuracy: 99.88%, Time: 16.26 seconds
Epoch 8, Accuracy: 99.92%, Time: 16.14 seconds
Epoch 9, Accuracy: 99.95%, Time: 16.28 seconds
Epoch 10, Accuracy: 99.98%, Time: 16.79 seconds
```

This indicates that the model now works accordingly with accuracy improving after every epochs after following the debugging steps above.

7.2 Comparison of Accuracy Trends Across Epochs in *nn.c* and *debug.c*

Table 1: Accuracy Trends Summary Table

Epochs	Accuracy *(nn.c)*	Accuracy *(debug.c)*
1	81.38%	95.27%
2	80.48%	98.31%
3	77.79%	99.02%
4	76.98%	99.41%
5	74.73%	99.65%
6	73.15%	99.79%
7	72.06%	99.88%
8	71.60%	99.92%
9	67.78%	99.95%
10	63.92%	99.98%

- Table 1 shows the accuracy for *nn.c* and *debug.c* for 10 epochs. *nn.c* demonstrates the degradation of accuracy that keeps decreasing from 81.38% to 63.92%, thereby indicating "unlearning." *debug.c*, shows a constant increase in accuracy from 95.27% to 99.98%.
- For *nn.c*, unlearning was attributed to data shuffling inconsistencies, improper momentum management, interim performance not monitored, and overfitting. More precisely, the improper shuffling logic generated biased, unbalanced mini-batches, reducing generalization capability. Furthermore, unstable gradient updates with badly managed momentum caused oscillations and prevented convergence. The absence of logs in performance of the mini-batch covered problems until it showed significant degradation.
- Improvements in *debug.c* includes, removing shuffling to simplify debugging, correcting gradient updates to stabilize learning, adding back performance monitoring for real-time insights, and reducing number of epochs for quicker validation. These resulted in increased accuracy due to better balancing of the training batches, smooth gradient descent, and earlier detection of problems during training, thus countering the unlearning trend seen in *nn.c*.

8.0 Significance of Machine Unlearning

Machine unlearning provides a critical process to comply to both adaptability and ethical integrity of machine learning models:

- Allows the models to delete part of its training data to safeguard the privacy of individuals, keep data protection standards high and rectify inaccuracies.
- Prevents any model from retaining information that are outdated or irrelevant, which again enhances the generalizability and relevance of the models.
- Machine unlearning can reduce biases and risks of overfitting significantly, hence allowing much fairer and more robust development of AI systems where one can foster trust in machine learning applications.

9.0 Practical Applications of Machine Unlearning in Real-World Scenarios

Machine unlearning, described in *debug.c*, can thus teach us about training dynamics and controlled forgetting. Lessons learned from there might help us build methods that more concretely meet practical challenges thereby fulfilling legal requirements on privacy or constructing ethical AI systems. Some concrete examples and their applicability follows:

1. **Social Media:**
    - Erase a user’s information from their recommendation algorithm if user chooses to delete the account.

- This guarantees the protection of sensitive information by complying with privacy laws such as General Data Protection Regulation's (GDPR) "Right to Be Forgotten," while maintaining trust in the use of social media.

2. **Data Privacy and Removal:**
   - Learned information are erased safely without retraining from scratch.
   - This enable compliance with privacy regulations, especially for sensitive user data, while reducing resources.
3. **Banks:**
   - Decrease the impact of a previously identified fraud transaction on their fraud detection model, to avoid any similar transaction in the future from getting misclassified as fraud.
   - This would enhance the model's accuracy and fairness so that normal activities are not flagged as fraudulent based on information that might be outdated.
4. **Educational Tools:**
   - How to debug and explore neural network instability.
   - This helps in improving your understanding of how models learn, unlearn, and adapt to changes in data.
5. **Healthcare:**
   - Apply machine unlearning to reduce the influence of a patient's data on their disease prediction model, in case a patient is diagnosed with a new disease that was not in the original training data.
   - This makes the models stay updated, adaptable, and accurate about the patient data to further improve AI-powered health.
6. **Ethical AI Systems:**
   - Employing mechanisms for controlled forgetting to comply with privacy standards.
   - Fair and transparent AI systems make people trust them while using or dealing with them.

9.1 Connecting Debugging Insights to Machine Unlearning Techniques

The following points are highlighted about machine unlearning from the debugging process of *debug.c*:

- **Tracking Training Dynamics:** Real-time feedback (e.g., log_metrics) will make sure models adapt correctly without any accidental degradation.
- **Smoothing Unlearning Processes:** Proper gradient handling with momentum management may teach the way in which data gets erased, without destabilizing the model.
- **Minimizing Resource Use:** Debugging shows that targeted interventions can be used in place of costly full retraining, providing cost-effective solutions for data removal.

The project bridges the gap from theoretical insight into effectively useful practical application by connecting such lessons to implementation.

10.0 Conclusion

This project represents a rare and intriguing behavior in neural networks—reverse learning or unlearning where models' performance decreases by increasing training. Initial model implementation in *nn.c* involved several issues like accuracy decaying because of shuffled data inconsistencies, mismanaged momentum, and lack of interim performance monitoring. Through careful debugging in *debug.c*, these problems were resolved in a revised implementation, which lead to an increased accuracy.

Faulty data shuffling removal in *debug.c* stabilized the input data, improving generalization. Correction of the momentum formula prevented oscillation of weights and ensured convergence. Addition of real-time accuracy logging brought critical feedback; hence, one was able to identify and correct the problems during training. Monitoring of training accuracy proved useful in reducing overfitting, gave better test performance, and furthermore, using fewer numbers of epochs allowed for faster debugging.

The debugging process thus explained not only the reasons for the phenomenon of reverse learning in *nn.c* but also deep insight into dynamics regarding the stability of training and monitoring of performance. Such results call for further research toward "controlled forgetting mechanisms" that give way to the capability of AI to dynamically adapt to evolving data while reducing biases and ensuring data privacy standards. Drawing such a challenge out, the potentiality of this project allows for the development of ethical AI together with adaptability to create trustworthiness and reliability in machine learning.