# Parallelized Neural Network training with the MNIST dataset

The RISCtakers

# Background

# What is a neural network?

- A type of an AI model inspired by the workings of the human brain
- Designed to recognize patterns in order to problem solve
- Consists of a network of interconnected nodes that process data
- Consists of an input layer, one or more hidden layers, and an output layer
- Learns through training, getting smarter with the goal of minimizing the error
    - Through forward propagation, backpropagation, and  gradient descent and loss functions

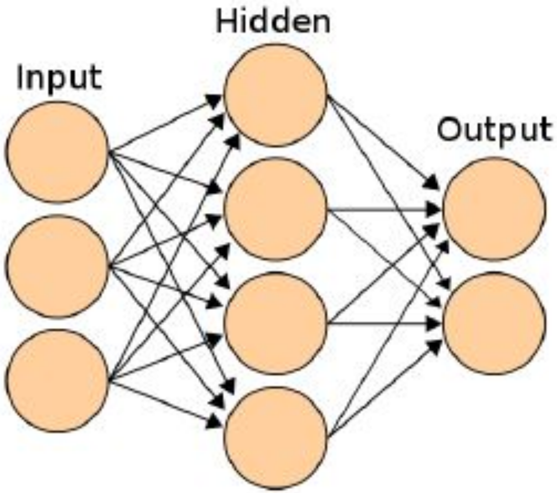
# MNIST dataset

- Contains 70,000 images of handwritten digits (0 through 9)
- Each image is a bitmap consisting of 28x28 pixels, representing a single digit
- Training set is 60,000 images, with a test set of 10,000 images
- The problem's simplicity creates a good environment to evaluate machine learning techniques and algorithms

# How does a neural network work?

# Layers

- A neural network is composed of three layers



# Input layer

- The starting point of the neural network
- Receives the input data, each neuron represents one feature of the input data
- Each neuron connects to every other neuron
- Each neuron represents one feature in the image
- In the context of the MNIST data set, each neuron represents the intensity of each grayscale pixel in the image.

# Hidden layers

- Where the majority of the computation is done
- “Hidden” means that they aren’t directly in contact with the input/output of the neural network
- Each neuron performs a weighted sum of inputs from the previous layer, applies a bias, then sends to an activation function
    - Non-linearity helps the network learn more complex relationships
- Different hidden layers can have different functions
    - Early layers might detect simple color features and shapes
    - Deeper layers may identify more complex patterns or shapes
- The number of hidden layers depends on how complex the task is
    - MNIST can be processed with just 3, may be improved with N layers.

# Output layer

- Final layer in the network
  - Presents the results of all of the network's computations
- These neurons represent the final output of the network
  - Output depends on the task
  - Could be a classification table, a single value, or in our case a 10 element vector with the probability that the input image is a certain digit
- [0.1, 0.05, **0.6**, 0.1, 0.02, 0.04, 0.01, 0.04, 0.1, 0.2]
  - 60% chance that the image is the digit '2'
- The output is a probability distribution showing the network's confidence level for each image

# Layer Initialization

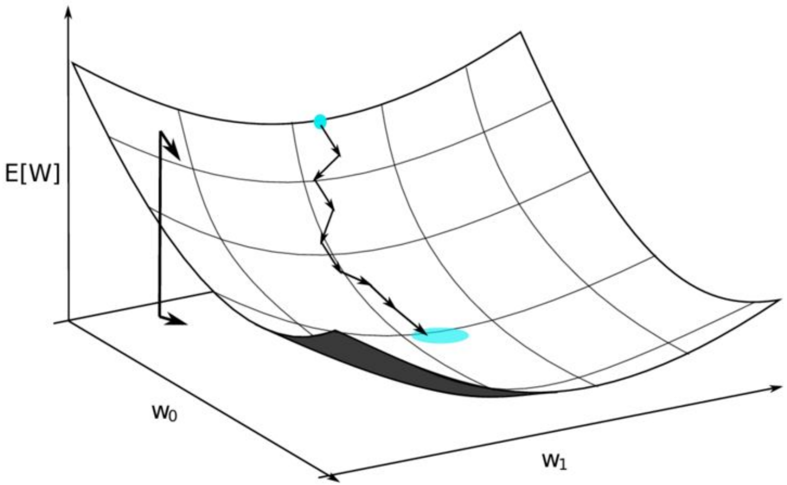
```
NeuralNetwork::NeuralNetwork(int inputSize, int hiddenSize, int outputSize, double learningRate, int numLayers)
{
    // add input layer to network
    network_layers.emplace_back(Layer(inputSize, hiddenSize));

    // add hidden layers
    for (int i = 0; i < numLayers; i++) {
        network_layers.emplace_back(hiddenSize, hiddenSize);
    }

    // add output layer
    network_layers.emplace_back(hiddenSize, outputSize);
}
```

# Approach

# Optimization method: Stochastic gradient descent

- This is an optimization algorithm used to minimize the loss function, updating the model's weights and biases accordingly
- Uses small batch of samples to compute the gradient
- In the context of MNIST, SGD works by dividing the process into small batches
  - It makes predictions based on the current weights and biases
  - The loss is calculated using these predictions and the correct labels of the batch
  - The gradient is computed with respect to the loss of each parameter
  - The weights and biases are updated in the opposite direction of the gradient
- More efficient than gradient descent, which looks at the dataset as a whole.

# SGD visualization



# Forward propagation

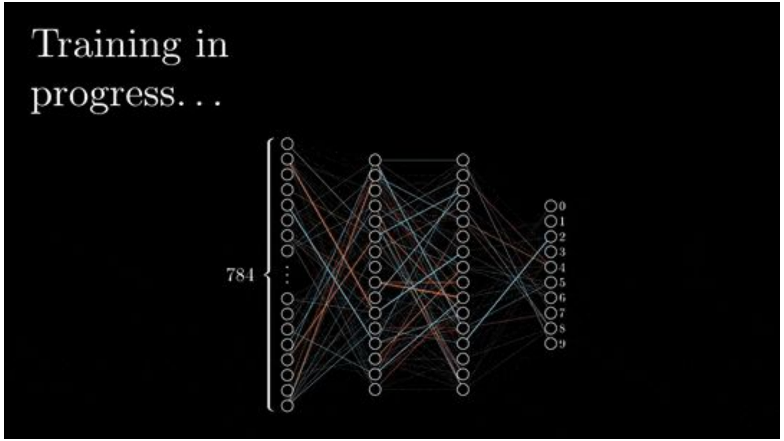
For each layer in the neural network, call the forward method and return the newly formulated vector.

Foreshadowing! ->

```
vector<double> NeuralNetwork::forward(vector<double> input) {
    // copy data
    vector<double> output = input;

    for (auto &layer: network_layers) {
        output = layer.forward_propagation_serial(output);
        // output = layer.forward_propagation_parallelized(output);
        // output = layer.forward_propagation_mkl(output);
    }

    return output;
```

# Backpropagation

- The process of computing gradients of the loss in respect to the weights and biases
- Propagates this gradient backward through the layers of the network, layer by layer
- Training stops when a certain number of epochs is reached, or improvement is no longer substantial
- Key component to neural network training




*Credit: 3Blue1Brown*

# Loss function

Crucial for gradient descent optimization. Compares the the actual to the target to determine how much a feature must be changed.

```
double NeuralNetwork::computeLoss(const vector<double> output, const vector<double> target)
{
    double loss = 0.0;

    // square loss function
    for (int i = 0; i < outputSize; ++i)
    {
        loss += pow(target.at(i) - output.at(i), 2);
    }
    return loss / output.size(); // Returns the average loss
}
```

# Why can this be parallelized?

# What makes this a parallelization problem?

1. Forward and backward propagations rely on vector multiplication, with layer of neurons being updated in an embarrassingly parallel manner.
2. Weights and biases updates
    a. After these are computed, they do need to be synchronized, but can often be done in parallel for different parts of the network
3. MNIST is a reasonably large dataset, and processing the data in parallel can lend a significantly faster training time

# What makes this a parallelization problem? (2)

- Vector Multiplication
    - Vector multiplication in forward propagation
    - Each element computed independently, inherently parallelizable
    - Vectors are multiplied by the weight matrix to produce another vector, then passed into an activation function. This process continues through multiple layers.
- Hardware optimization
    - Running this neural network model on a GPU would take advantage of its parallel nature. Optimized for fast vector and matrix operations
    - Hardware acceleration is one of the reasons why neural networks have become feasible for deep learning models.

# Vector Multiply Approaches

# Serial Multiply

```
std::vector<double> Layer::forward_propagation_serial(const std::vector<double> &input_vector) {
    std::vector<double> output_vector;
    double value;

    for (int i = 0; i < outputs; i++) {
        value = 0;
        for (int j = 0; j < input_vector.size(); i++) {
            value += input_vector.at(j) * weight_matrix.at(i).at(j);
        }
        output_vector.at(i) = sigmoid(value + bias);
    }
    return output_vector;
}
```

# Parallel For

```
std::vector<double> Layer::forward_propagation_parallelized(const std::vector<double> &input_vector) {
    std::vector<double> output_vector;
    double value;

    #pragma omp parallel for
    for (int i = 0; i < outputs; i++) {
        value = 0;
        for (int j = 0; j < inputs; i++) {
            value += input_vector.at(j) * weight_matrix.at(i).at(j);
        }
        output_vector.at(i) = sigmoid(value + bias);
    }
    return output_vector;
}
```

# Intel MKL

```
std::vector<double> Layer::forward_propagation_mkl(const std::vector<double> &input_vector) {
    std::vector<double> output_vector;

    // do vector multiplication
    cblas_dgemv(CblasRowMajor, CblasNoTrans,
        inputs, outputs, 1.0, weight_matrix.data(), inputs, input_vector.data(), 1, 1.0, output_vector.data(), 1);
    return output_vector;
}
```

# Challenges to look out for

- Hyperparameter tuning
  - Being sure to choose the correct learning rate, number of layers, number of neurons in each layer
  - Inappropriate choices may lead to a poor learning rate or slow convergence
- Model explainability
  - Due to the complex nature of a neural network, it can be challenging to understand why a certain prediction is happening
  - Many parameters and different possible outputs
  - Ensuring that a model is consistent across tests and the data can be repeated is important for creating a strong neural network

# Thanks!