Solving a binary classification problem with an artificial neural network (MLP)

Håkon Strand, Ole Gilje Gunnarshaug and Herman Ellingsen

INF205

Contents

- Code architecture
- Data structure
- Iteration walkthrough
- Performance
- Concurrency
- Alternative approaches

Code Architecture

mlp

Functions

Class MLP
- void insert sample
- void feed_forward
- void backpropagate
- void set_learning_rate
- void print_results
- void print_output_values

Variables

- **Public:**
  - map<int, Layers> map_of_layers
- **Private:**
  - static double learning_rate

layers

Functions

Class Layers
- **Public:**
- void activate_nodes
- void hidden_gradients
- void update_weights
- void set_layer_idx
- void set_total_nodes
- void set_output_weight_dim
- int get_layer_idx
- int get_total_nodes
- int get_output_weight_dim

Variables

- **Public:**
  - map<int, Node> map_of_nodes
- **Private:**
  - int layer_idx
  - int total_nodes
  - int output_weight_dim

node

Functions

Class Node
- **Public:**
- double sigmoid
- double sigmoid_derivative
- void output_gradient
- void set_output_value
- void set_output_value_bias
- void set_gradient
- double get_output_value
- int get_curr_node_idx
- const double get_gradient
- double get_loss

Variables

- **Public:**
  - vector<double> output_weights
- **Private:**
  - double output_value
  - int curr_node_idx
  - double gradient
  - double loss

read_data

Functions

Class ReadData
- **Public:**
  - void create_dataset
  - vector<vector<double>> get_my_data
  - vector<double> get_y_true

Variables

- **Private:**
  - vector<vector<double>> my_data
  - vector<double> y_true

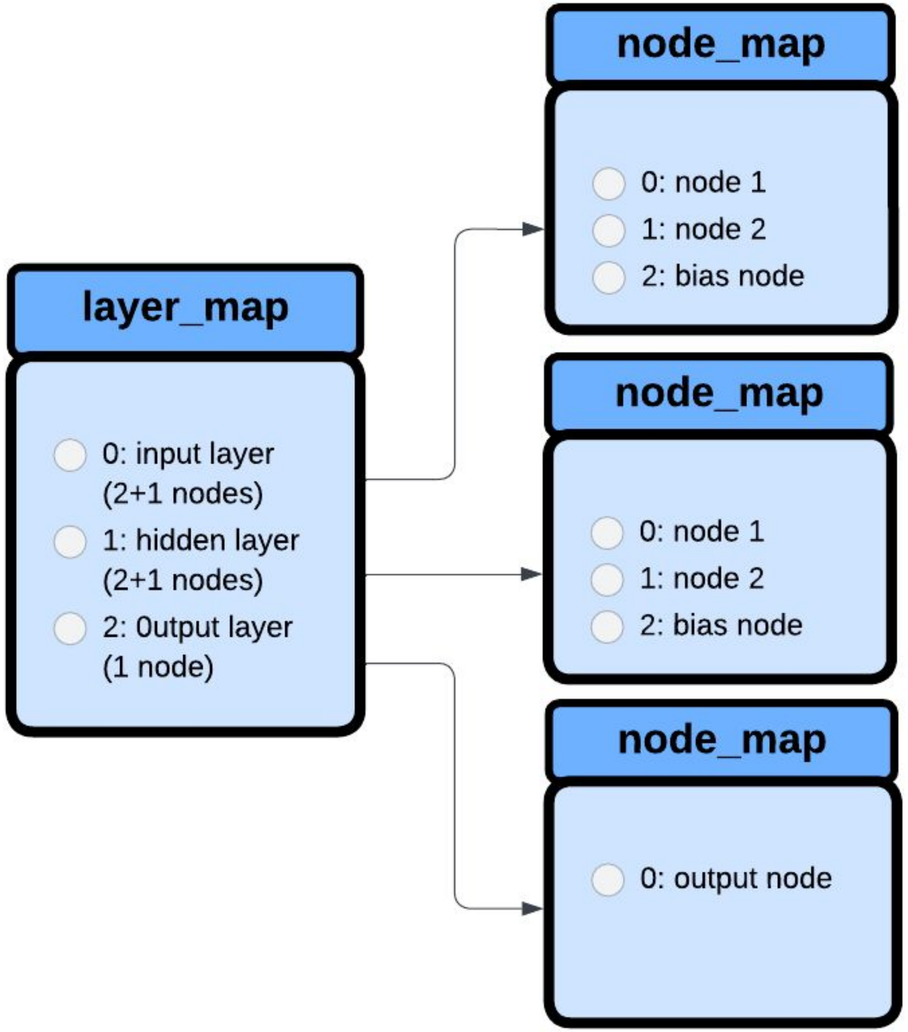
Data structure

- Sample data:
  - Samples in matrix (X)
  - True values in vector (y)
- Classes:
  - Initialize network in mlp constructor
  - Create map containing layers
    <layerId, layerObject>
  - Inside layerObject, create map containing nodes
    <nodeId, nodeObject> (weights as vectors)

Maps

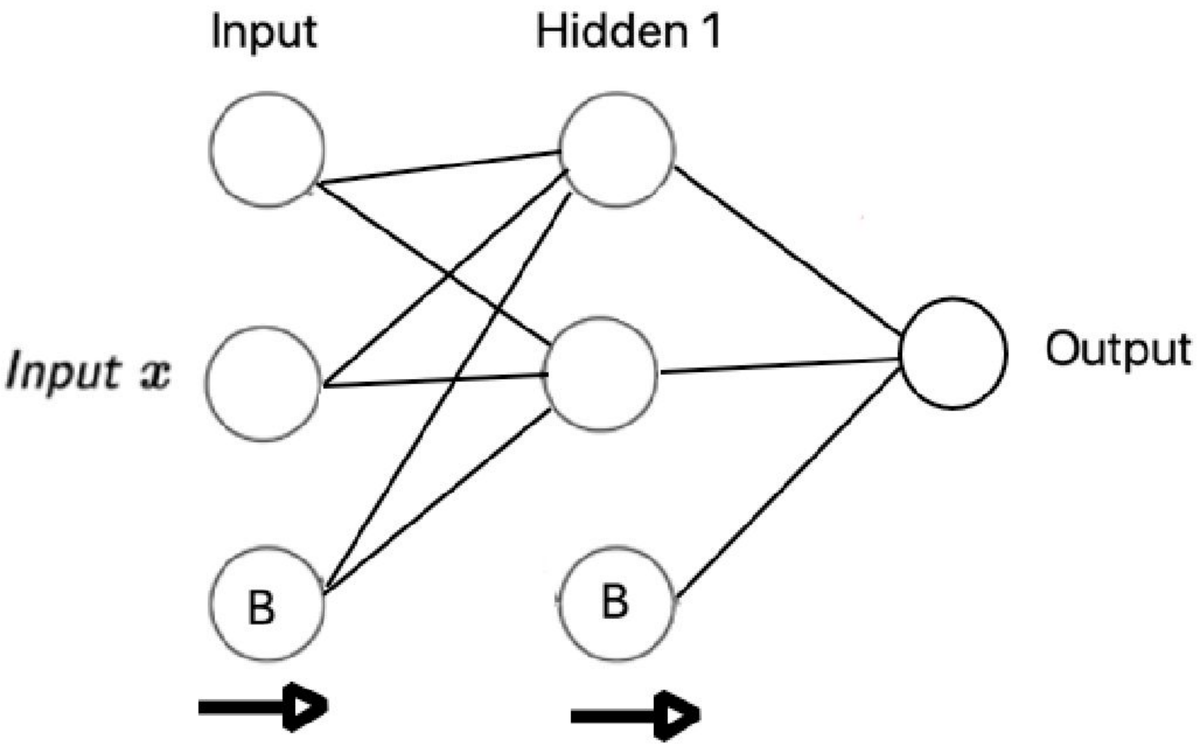
Example:

- Layer map <layerId, layerObject>:
    - 1 input layer (idx: 0)
    - 1 hidden layer (idx: 1)
    - 1 output layer (idx 2)
- Node map <nodeId, nodeObject>
    - First: 2 nodes (2 features), 1 bias
    - Second: 2 nodes, 1 bias
    - Third: 1 node (1 bias)



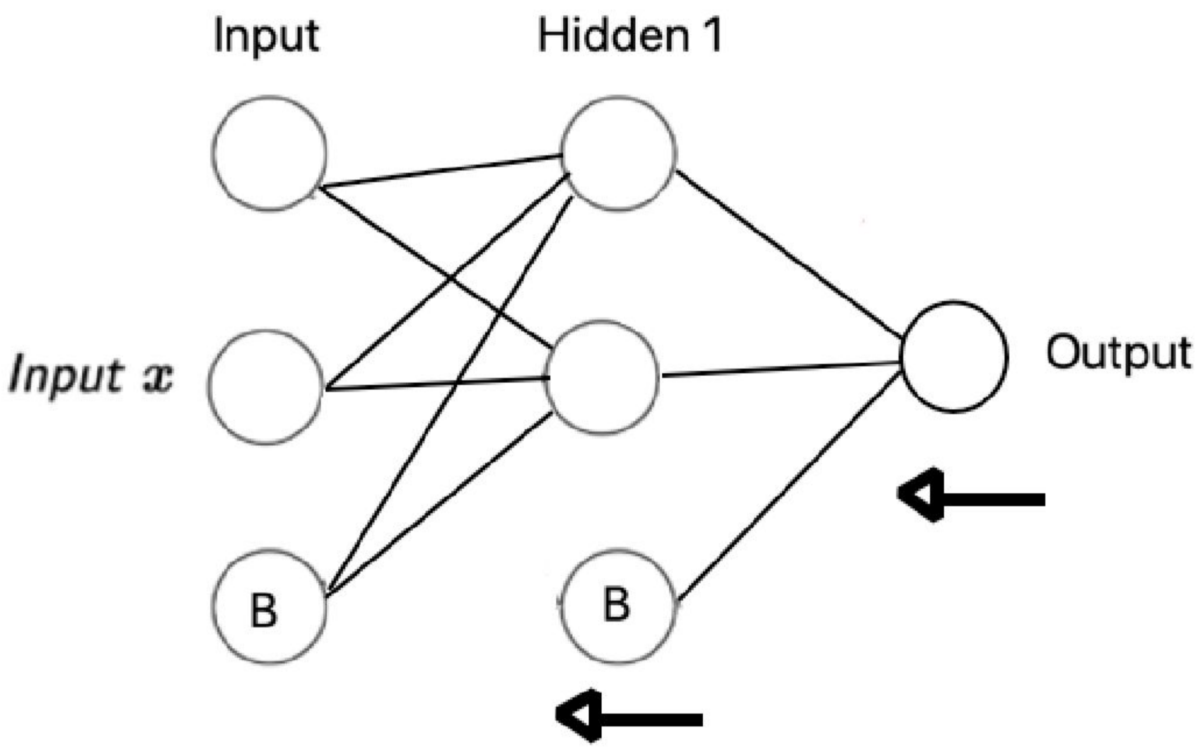
Iteration walkthrough: Feed forward

- Outgoing weights stored in layer to the left
  - i.e. input layer nodes has weights going to Hidden 1
- Weight vector matching id of node in the right layer
- Loop through layers
- Activate nodes
  - Sigmoid function
- *Note: Left and right layer/node*



Iteration walkthrough: Backpropagation

1. Calculate output gradient
2. Calculate hidden gradients
3. Update weights



Iteration walkthrough - Step 1: insert sample values

- Loop through nodes in input layer
- Skip bias (defined as 1 during node initialization)
- Set output value = feature value

```
/**...
void MLP::insert_sample(std::vector<double> const &input_data)
{
    int total_input_nodes = map_of_layers[0].get_total_nodes();
    for (int i = 0; i < total_input_nodes; i++){
        if (i == total_input_nodes-1){ // Bias node
            continue; // Already sat output value when initializing nodes
        }
        else{
            map_of_layers[0].map_of_nodes[i].set_output_value(input_data[i]);
        }
    }
}
```

Iteration walkthrough - Step 2: feed forward

- Loop through layer map
- Activate nodes (starts in first hidden layer)

```
/** ...
void MLP::feed_forward()
{
    // Loop through 1st hidden layer to (including) ouput layer
    for (int layer = 1; layer < map_of_layers.size(); layer++){
        Layers &left_layer = map_of_layers[layer-1];
        Layers &current_layer = map_of_layers[layer];

        current_layer.activate_nodes(left_layer);
    }
}
```

Iteration walkthrough - Step 2: feed forward

Activate nodes:

- For every node in current layer:

1. Loop through all nodes in the layer to the left, then sum:

left_node_weight[curr_node_id]

*

left_node_outputvalue

2. Apply sigmoid
3. Current node++ (excl. bias)

```
/**...
void Layers::activate_nodes(Layers &left_layer)
{
    // Inside current layer, has access to prev layer
    int current_node_id;
    double out_value;

    // Loop through nodes in current layer (excl. bias)
    for (int curr_n = 0; curr_n < get_total_nodes()-1; curr_n++){
        double z = 0.0;

        // Loop through nodes (incl bias) in previous layer and add sum of z (pre activation sum)
        for (int left_n = 0; left_n < left_layer.get_total_nodes(); left_n++){
            Node &left_node = left_layer.map_of_nodes[left_n];
            z += left_node.output_weights[curr_n] * left_node.get_output_value();
        }
        // Activate the current node
        out_value = map_of_nodes[curr_n].sigmoid(z);
        map_of_nodes[curr_n].set_output_value(out_value);
    }
}
```

Iteration walkthrough - Step 2: feed forward

- Output layer
- Output value

```
/** ...
void MLP::feed_forward()
{
    // Loop through 1st hidden layer to (including) ouput layer
    for (int layer = 1; layer < map_of_layers.size(); layer++){
        Layers &left_layer = map_of_layers[layer-1];
        Layers &current_layer = map_of_layers[layer];

        current_layer.activate_nodes(left_layer);
    }
}
```

Iteration walkthrough - Step 3: backpropagate

1. Calculate output layer gradient

```
void MLP::backpropagate(const int y_true)
{
    // Step 1: Output layer gradient
    Layers &output_layer = map_of_layers[map_of_layers.rbegin()->first];
    output_layer.map_of_nodes[0].output_gradient(y_true);

    // Step 2: Hidden layer gradients
    // Calculate hidden layer gradient (from last hidden layer down to (including) 1st hidden layer)
    for (int layer = map_of_layers.size()-2; layer > 0; layer--){
        Layers &right_layer = map_of_layers[layer + 1];

        map_of_layers[layer].hidden_gradients(right_layer);
    }

    // Step 3: Update weights
    // From output layer to first hidden layer, update weights
    for (int layer = map_of_layers.size() - 1; layer > 0; layer--){
        Layers &left_layer = map_of_layers[layer - 1];

        map_of_layers[layer].update_weights(left_layer, learning_rate);
    }
}
```

Iteration walkthrough - Step 3: backpropagate

1. Calculate output layer gradient

```
/**...
void Node::output_gradient(const int y_true)
{
    error = output_value - y_true;
    gradient = error * sigmoid_derivative();
}
```

Iteration walkthrough - Step 3: backpropagate

1. Calculate output layer gradient
2. Calculate hidden layer gradients

```
void MLP::backpropagate(const int y_true)
{
    // Step 1: Output layer gradient
    Layers &output_layer = map_of_layers[map_of_layers.rbegin()->first];
    output_layer.map_of_nodes[0].output_gradient(y_true);

    // Step 2: Hidden layer gradients
    // Calculate hidden layer gradient (from last hidden layer down to (including) 1st hidden layer)
    for (int layer = map_of_layers.size()-2; layer > 0; layer--){
        Layers &right_layer = map_of_layers[layer + 1];

        map_of_layers[layer].hidden_gradients(right_layer);
    }

    // Step 3: Update weights
    // From output layer to first hidden layer, update weights
    for (int layer = map_of_layers.size() - 1; layer > 0; layer--){
        Layers &left_layer = map_of_layers[layer - 1];

        map_of_layers[layer].update_weights(left_layer, learning_rate);
    }
}
```

Iteration walkthrough - Step 3: backpropagate

Hidden layer gradients:

- For each node in current layer:

1. Loop through every node in layer to the right
2. Sum of:

   curr_node_weight[right_n]

   *

   gradient (right node)

3. Set gradient (curr node)
4. Current node++ (incl. bias)

```
void Layers::hidden_gradients(Layers &right_layer)
{
    double sum_delta_hidden;
    double gradient_value;

    // Loop through every node in current layer (incl. bias)
    for (int curr_n = 0; curr_n < get_total_nodes(); curr_n++){

        // Loop through every node in layer to the right (excl. bias)
        for (int right_n = 0; right_n < right_layer.get_total_nodes()-1; right_n++){
            sum_delta_hidden +=
            map_of_nodes[curr_n].output_weights[right_n] *
            right_layer.map_of_nodes[right_n].get_gradient();
        }

        // Set gradient of current node
        gradient_value = sum_delta_hidden * map_of_nodes[curr_n].sigmoid_derivative();
        map_of_nodes[curr_n].set_gradient(gradient_value);
    }
}
```

Iteration walkthrough - Step 3: backpropagate

1. Calculate output layer gradient
2. Calculate hidden layer gradients
3. Update weights

```
void MLP::backpropagate(const int y_true)
{
    // Step 1: Output layer gradient
    Layers &output_layer = map_of_layers[map_of_layers.rbegin()->first];
    output_layer.map_of_nodes[0].output_gradient(y_true);

    // Step 2: Hidden layer gradients
    // Calculate hidden layer gradient (from last hidden layer down to (including) 1st hidden layer)
    for (int layer = map_of_layers.size()-2; layer > 0; layer--){
        Layers &right_layer = map_of_layers[layer + 1];

        map_of_layers[layer].hidden_gradients(right_layer);
    }

    // Step 3: Update weights
    // From output layer to first hidden layer, update weights
    for (int layer = map_of_layers.size() - 1; layer > 0; layer--){
        Layers &left_layer = map_of_layers[layer - 1];

        map_of_layers[layer].update_weights(left_layer, learning_rate);
    }
}
```

Iteration walkthrough - Step 3: backpropagate

Update weights:

- For every node in current layer:

1. Loop through every node in layer to the left
2. Add delta_weight to left node where weight id matching current node
3. Current node++ (excl. bias)

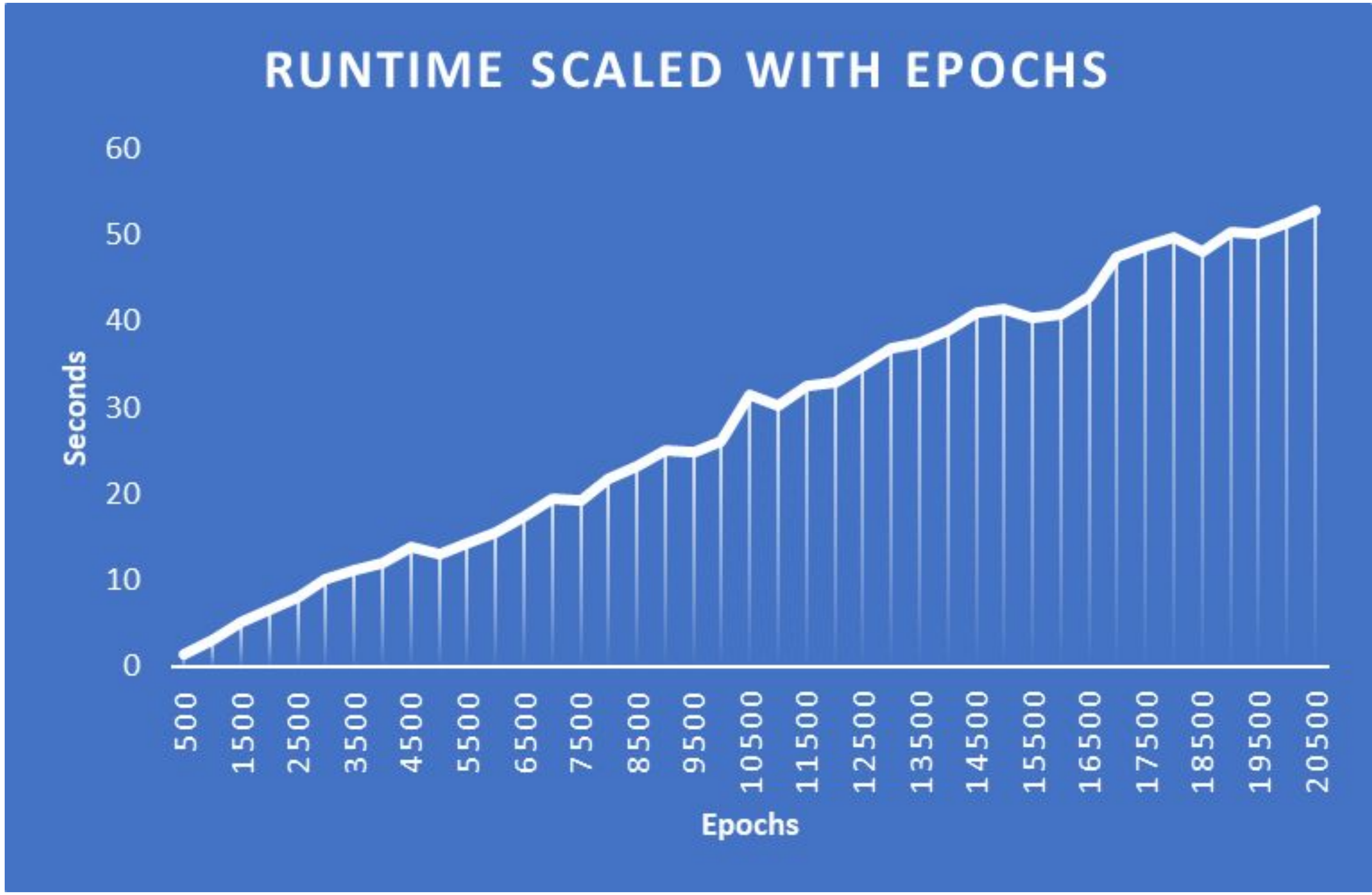
```
void Layers::update_weights(Layers &left_layer, double learning_rate)
{
    double delta_weight = 0.0;
    // Loop through every node in current layer (excl. bias)
    for (int curr_n = 0; curr_n < get_total_nodes()-1; curr_n++){
        Node &curr_node = map_of_nodes[curr_n];

        // Loop through every node in left layer (incl. bias)
        for (int left_n = 0; left_n < left_layer.get_total_nodes(); left_n++){
            Node &left_node = left_layer.map_of_nodes[left_n];

            double delta_weight =
            -(learning_rate * left_node.get_output_value() * curr_node.get_gradient());

            left_node.output_weights[curr_n] += delta_weight;
        }
    }
}
```

Performance overview

- Test how the runtime scales:
  - Increasing number of epochs
  - Increasing number of hidden layers
  - Increasing number of samples
- All cases looks to be scaling linearly

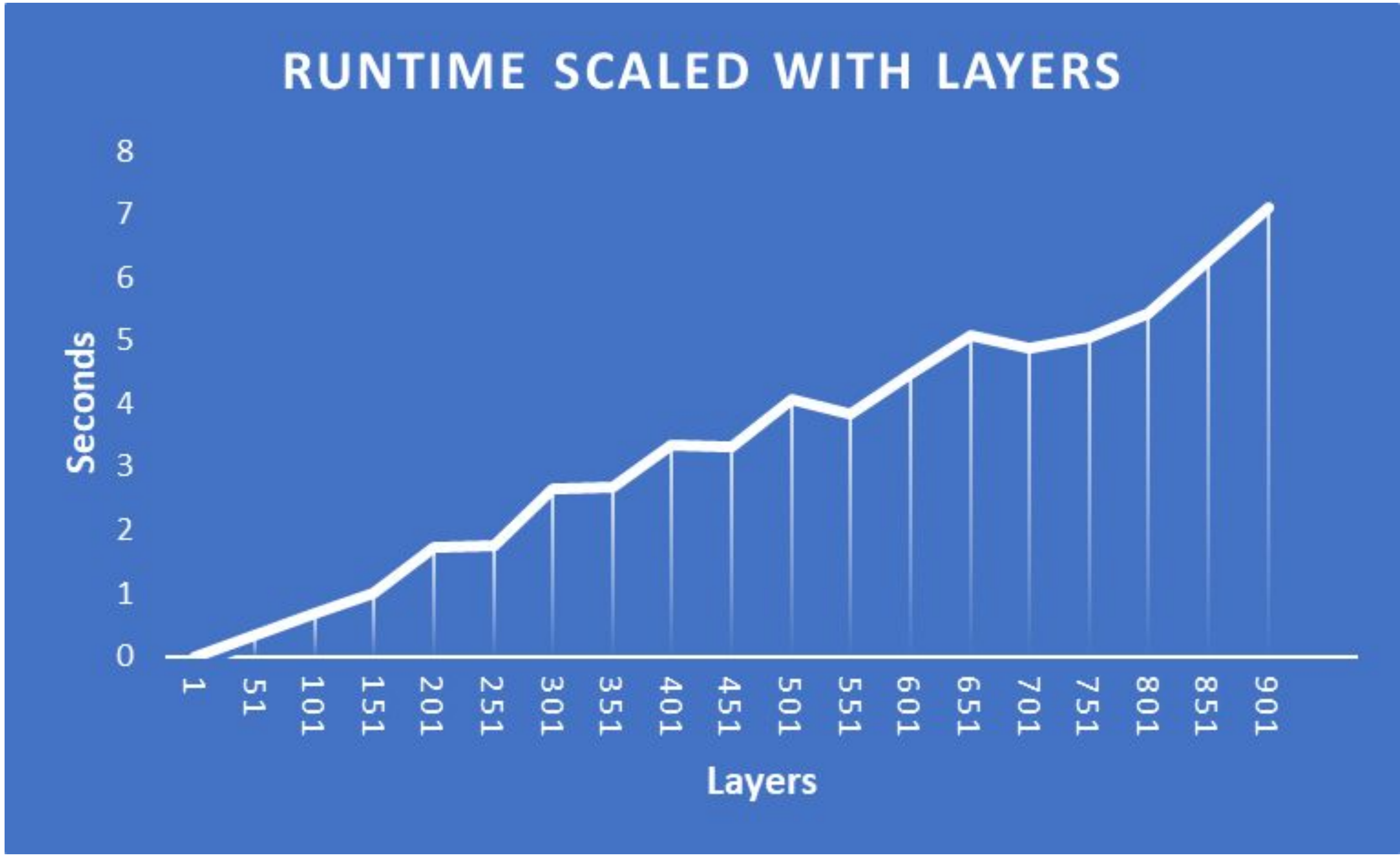
Performance (epochs)

- Data: Iris
- Hidden layers: 1
- Nodes: 4
- Increase number of epochs
- Linear runtime scaling



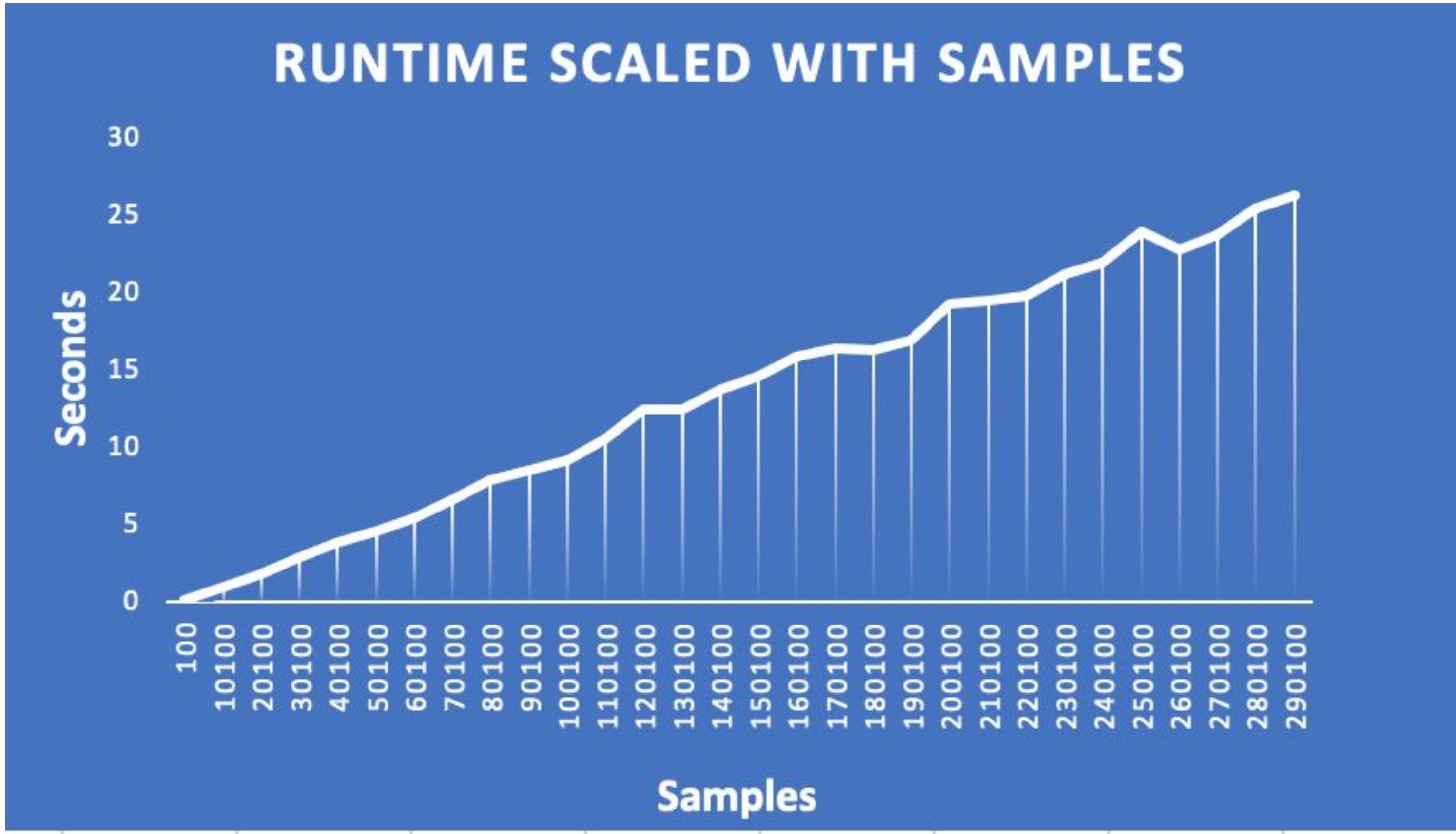
Performance (Layers)

- Data: Iris
- Nodes: 10
- Increase number of hidden layers
- Linear runtime scaling



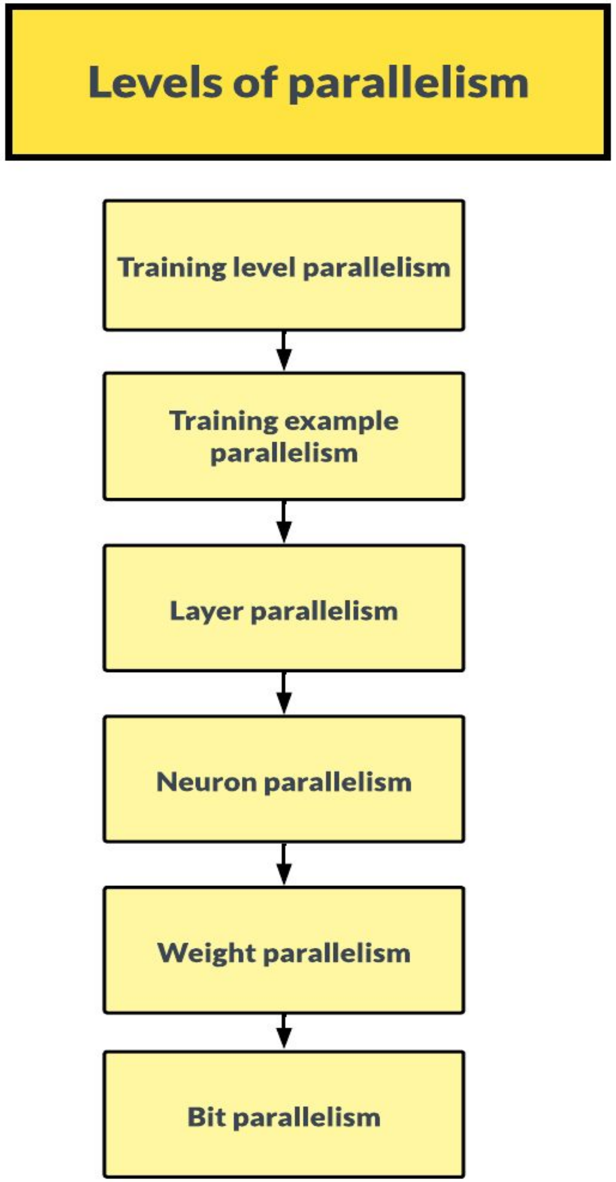
Performance (samples)

- Data: randomly generated data
- Hidden layers: 1
- Nodes: 4
- Cols: 20
- Increase number of samples
- Linear runtime scale



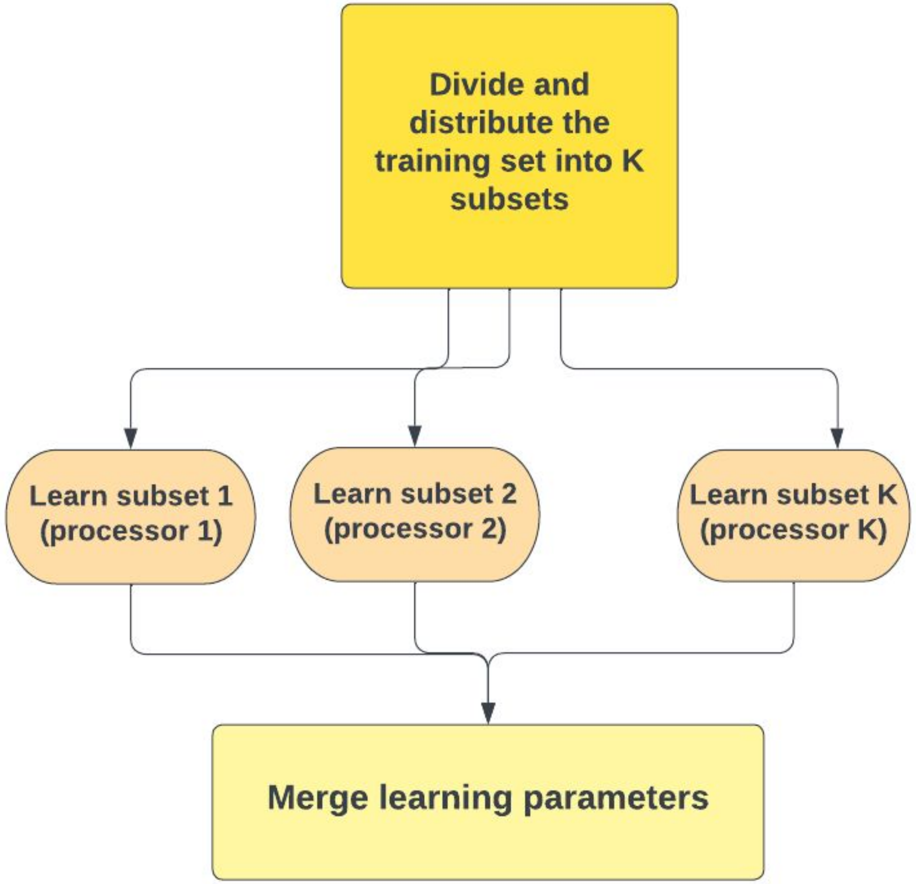
Concurrency

- The training and evaluation process of the nodes in a large network can take a long time
- Neural networks can be parallel on different levels
  - Computiations of each node are independent in the feed forward
  - However, backpropagation limits the speedup for parallel neuron computations



Concurrency (cont)

- Parallel computing with MPI
- Training set level
    a. Divide input in k subsets
    b. Train k models on k processors
    c. Merge learning parameters
- Compare performance for paralleled and non-paralleled network



Alternative approaches and improvements

- Store all data in vectors and matrices
  - Vector-matrix multiplication
    - Take advantage of more efficient linear algebra packages
  - A bit less intuitive
- Use Orion to speed up