Neural Networks

This project implements a neural network framework from first principles, beginning with linear regression and extending to more complex architectures.

Linear Regression

Neural networks begin with linear regression. Let $x \in \mathbb{R}^d$ be our input vector with dimension d, $W \in \mathbb{R}^{p \times d}$ a weight matrix mapping to p outputs, and $b \in \mathbb{R}^p$ a bias vector. The linear transformation computes:

$$\hat{y} = \boldsymbol{W}\boldsymbol{x} + b$$

This computation forms the core of our linear layer. Learning occurs through gradient descent, which requires computing how changes in weights affect the loss. Let $\mathcal{L}$ be our loss function, mean square error. The weight gradients are:

$$\nabla_{\boldsymbol{W}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{W}}$$

For a linear layer, $\frac{\partial \hat{y}}{\partial \boldsymbol{W}_{ij}} = x_j$ for output $\hat{y}_i$, which in matrix form is $\frac{\partial \hat{y}}{\partial \boldsymbol{W}} = x^T$.

For MSE, defined as:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n} (y_i - \hat{y}_i)^2$$

This gradient is:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{2}{n}(y - \hat{y})$$

Our implementation seperates these concerns into distinct modules:

- A `Linear` module handling forward computation and gradient calculations ;
- A `MSELoss` module computing the loss and its gradient ;
- The base `Module` class defining interfaces for all layers, given by the handout.

The training process iteratively applies gradient descent:

```
Gradient descent for linear regression
Input: $\boldsymbol{x}$ examples, $y$ labels, $\eta$, $\mathcal{E}$
Ouput: $\boldsymbol{W}, b$
$\boldsymbol{W} \overset{⚂⚃}{\longleftarrow} \mathbb{R}^{p \times d}, b \overset{⚂⚃}{\longleftarrow} \mathbb{R}^p$
For $e \in \{1, ..., \mathcal{E}\}$
    $\hat{y} \leftarrow \boldsymbol{W}\boldsymbol{x} + b$
    $\mathcal{L} \leftarrow \frac{1}{n} \sum (y_i - \hat{y}_i)^2$
    $\nabla_{\boldsymbol{W}}\mathcal{L} \leftarrow -\frac{2}{n}(y - \hat{y})\boldsymbol{x}^T$
    $\nabla_b\mathcal{L} \leftarrow \sum -\frac{2}{n}(y - \hat{y})$
    $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta\nabla_{\boldsymbol{W}}\mathcal{L}$
    $b \leftarrow b - \eta\nabla_b\mathcal{L}$
```

Results analysis.

Fig. 1 shows our linear regression results. The loss curve displays the expected pattern: rapid initial decrease followed by gradual convergence. Within 200 epochs, most improvement occurs, with minimal gains thereafter. The bottom plots contrast the median and worst models. Both capture linear relationships but with different slopes. The data points cluster tightly around both lines, showing both models learned useful patterns performance differences. This variability stems from random initialization and dataset generation.

These results indicate our implementation correctly applies the gradient descent algorithm. The smooth convergence indicates proper gradient computation and parameter updates.

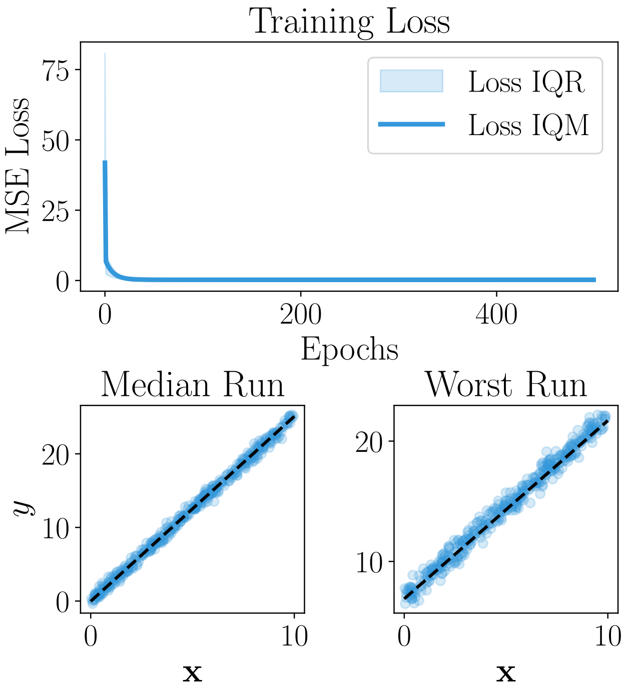

Fig. 1. – This figure displays training results from 100 linear regression trials ($n = 100$) with 200 samples per run ($\sigma = 200$), learning rate 0.01 ($\eta = 0.01$), and 1000 epochs ($\mathcal{E} = 1000$). The top plot shows the interquartile mean loss and Q1-Q3 range during training, while the bottom plots contrast the median and worst performing models from the ensemble. To assure statistical significance, we made 15 runs for each experiments.

Linear and non-linear classification

Neural networks can also be used for classification tasks. We can adapt our linear regression model to classify data points into k classes. The output layer is a softmax layer, which normalizes the output to a probability distribution over the classes. The softmax activation and cross-entropy loss are often used together in classification tasks because they form a mathematically convenient and numerically stable pair.

Softmax turns logits into probabilities Given raw outputs $z \in \mathbb{R}^k$, softmax maps them to a probability distribution:

$$\text{softmax}(z)_i = \frac{e^{\{z_i\}}}{\sum_{\{j=1\}}^{k} e^{\{z_j\}}}$$

This ensures the outputs are in $[0, 1]$ and sum to 1, thus interpretable as class probabilities.

Cross-entropy measures the distance between predicted and true distributions. For a one-hot encoded true label y, and predicted probability $\hat{y} = \text{softmax}(z)$, the loss is:

$$\mathcal{L}(y, \hat{y}) = -\sum_{\{i=1\}}^{k} y_i \log(\hat{y}_i)$$

Since y is one-hot, only the log probability of the correct class is penalized.

Also, the gradient simplifies nicely as the derivative of the loss with respect to the input logits z is:

$$\frac{\partial \mathcal{L}}{\partial z} = \hat{y} - y$$

This is much simpler than computing the gradient of softmax and cross-entropy separately. It reduces computation and avoids numerical instability.

The training process is similar to linear regression, but we use the softmax layer and categorical cross-entropy loss. The algorithm is as follows:

```
Gradient descent for classification
    Input: $\boldsymbol{x}$ examples, $y$ labels, $\eta$, $\mathcal{E}$
    Ouput: $\boldsymbol{W}, b$
    $\boldsymbol{W} \xleftarrow{⚂⚃} \mathbb{R}^{p \times d}, b \xleftarrow{⚂⚃} \mathbb{R}^p$
    For $e \in \{1, ..., \mathcal{E}\}$
        $\hat{y} \leftarrow \text{softmax}(\boldsymbol{W}\boldsymbol{x} + b)$
        $\mathcal{L} \leftarrow -\frac{1}{n} \sum (y_i \log(\hat{y}_i))$
        $\nabla_{\boldsymbol{W}} \mathcal{L} \leftarrow \nabla_{\hat{\boldsymbol{y}}} \mathcal{L} \boldsymbol{x}^T$
        $\nabla_b \mathcal{L} \leftarrow \sum \nabla_{\hat{\boldsymbol{y}}} \mathcal{L}$
        $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \nabla_{\boldsymbol{W}} \mathcal{L}$
        $b \leftarrow b - \eta \nabla_b \mathcal{L}$
```

Results analysis.

Fig. 2 shows the results of our linear classification task. The model is trained on a dataset where the classes are linearly separable. The top plot displays the evolution of the loss during training, which decreases rapidly in the first epochs and then plateaus, indicating fast convergence. The interquartile mean and Q1-Q3 range show that most runs achieve similar performance, with low variance. The bottom plots illustrate the decision boundaries learned by the median and worst models. Both models successfully separate the two classes. The worst model still captures the general structure, but with a less optimal orientation, highlighting the effect of initialization and data variability.

Fig. 3 presents the results of our non-linear classification task, specifically on the XOR problem, which is not linearly separable: no single line can divide the true and false outputs in input space. This necessitates the use of a non-linear model with hidden layers to capture the disjoint decision regions. Here, the model includes a non-linear activation function, enabling it to learn the complex decision boundary required to separate the classes. The loss curve in the top plot decreases more slowly than in the linear case, reflecting the increased difficulty of the task, but still shows fast improvement. The bottom plots show the decision boundaries for the median and worst models. The median model successfully learns the non-linear structure of the XOR problem, while the worst model demonstrates some misclassification and less understanding of the problem as some parts at the right of the plot are classified as a part of the red dots. Overall, these results confirm that our implementation can handle both linear and non-linear classification problems effectively.

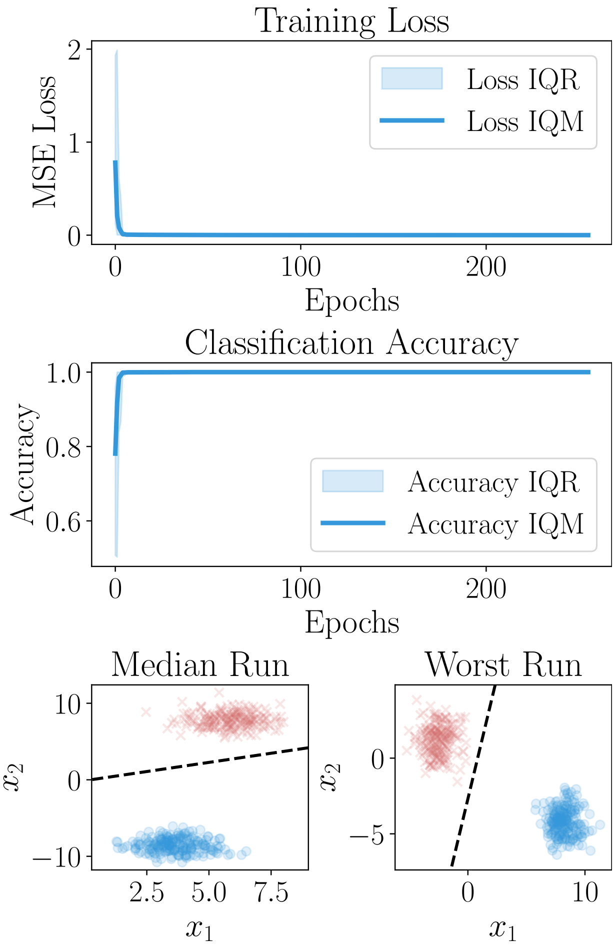

Fig. 2. – This figure displays training results from 100 linear classification trials ($n = 100$) with 200 samples per run ($\sigma = 200$), learning rate 0.01 ($\eta = 0.01$), and 1000 epochs ($\mathcal{E} = 1000$). The top plot shows the interquartile mean loss and Q1-Q3 range during training, while the bottom plots contrast the median and worst performing models from the ensemble.

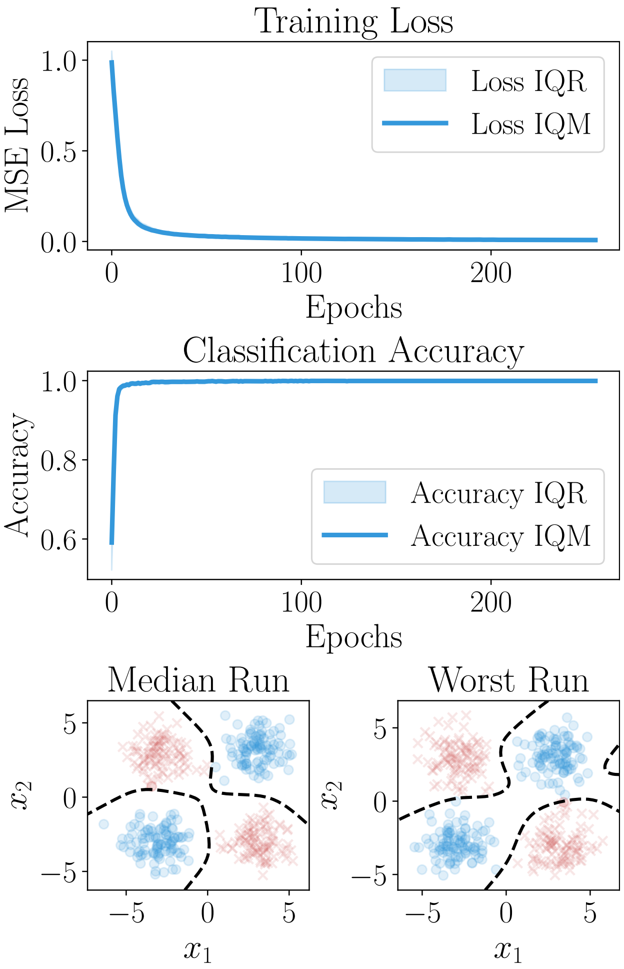

Fig. 3. – This figure displays training results from 100 non-linear classification trials ($n = 100$) with 200 samples per run ($\sigma = 200$), learning rate 0.01 ($\eta = 0.01$), and 1000 epochs ($\mathcal{E} = 1000$). The top plot shows the interquartile mean loss and Q1-Q3 range during training, while the bottom plots contrast the median and worst performing models from the ensemble.

MNIST classification

The MNIST dataset is a fundamental benchmark in the field of machine learning and computer vision. It consists of 60,000 training and 10,000 test grayscale images of handwritten digits, each of size 28×28 pixels, flattened into vectors of 784 dimensions. The classification task involves assigning each image to one of 10 digit classes (0 through 9).

To solve this problem, we designed a multi-layer perceptron (MLP). The network architecture consists of two fully connected hidden layers, each followed by a non-linear activation function, and a final softmax output layer. The softmax layer converts the raw outputs (logits) into a probability distribution over the 10 classes. The model is trained using categorical cross-entropy loss, which is appropriate for multi-class classification.

Fig. 4 illustrates the evolution of the training and validation loss as well as accuracy over time. As expected, the training loss decreases steadily, and validation accuracy increases over the course of training. The convergence pattern indicates the model generalizes well, with minimal overfitting. These results validate the effectiveness of our network design and optimization strategy on this task. The model stop to learn after five epochs and ~96% accuracy.

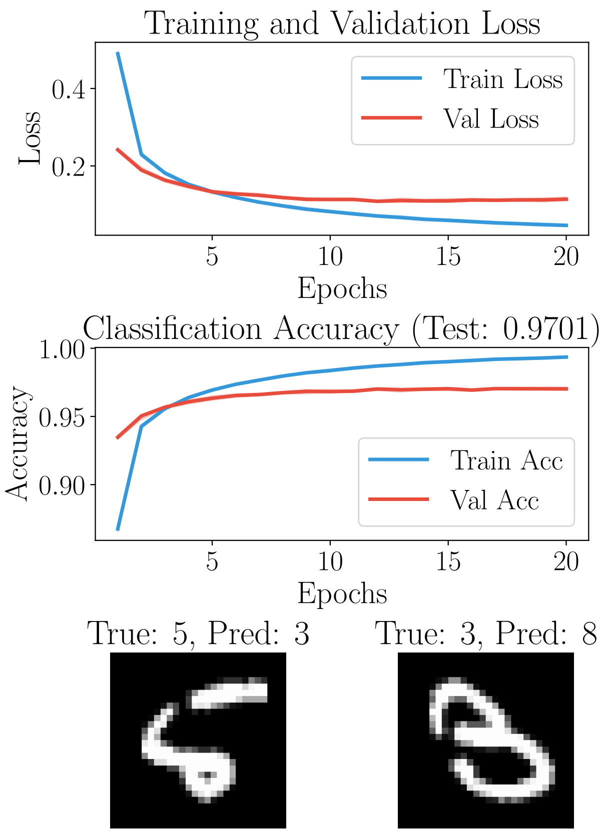

Fig. 4. – The figure displays training and validation loss (resp. accuracy) for the MNIST classification task. The model is a sequential model with two hidden layers of 64 then 32 neurons and a softmax output layer. The model is trained with the Adam optimizer, a learning rate of 0.001, a batch size of 128 and 50 epochs. The activation function of the first layer is ReLU. The model is trained on the MNIST dataset, which consists of 60,000 training images and 10,000 test images. The model is evaluated on the test set after training. The last figure shows two examples of misclassified images.

Activation functions experiments

Activation functions play a crucial role in the learning dynamics of neural networks. They introduce non-linearity into neural networks, allowing them to approximate complex, non-linear functions. Without non-linear activations, a network with multiple layers would be equivalent to a single linear transformation. Choosing an appropriate activation function is therefore crucial for enabling the network to model the structure of the data.

We compare the performance of three common non-linear activation functions: ReLU, Sigmoid, and TanH. Each function is used in the first hidden layer of an otherwise identical model, which includes two hidden layers of sizes 64 and 32, and a softmax output layer.

The activation functions are defined as follows:

ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$

Sigmoid: $f(x) = \frac{1}{1+e^{\{-x\}}}$

TanH (Hyperbolic Tangent): $f(x) = \frac{\{e^{x}-e^{\{-x\}}\}}{\{e^{x}+e^{\{-x\}}\}}$

As shown in Fig. 5, TanH and ReLU largely outperforms the Sigmoid activation function in terms of final test accuracy. TanH seems a little bit better than ReLU but it must be reminded that we trained the networks on a simple dataset and only have two hidden layers. In practice, in more advanced networks, ReLU attenuates the vanishing gradient problem. It must also be precised that it converge way faster and is way cheaper in calculation time because of its simplicity. These results confirm ReLU as a robust default choice for modern neural networks.

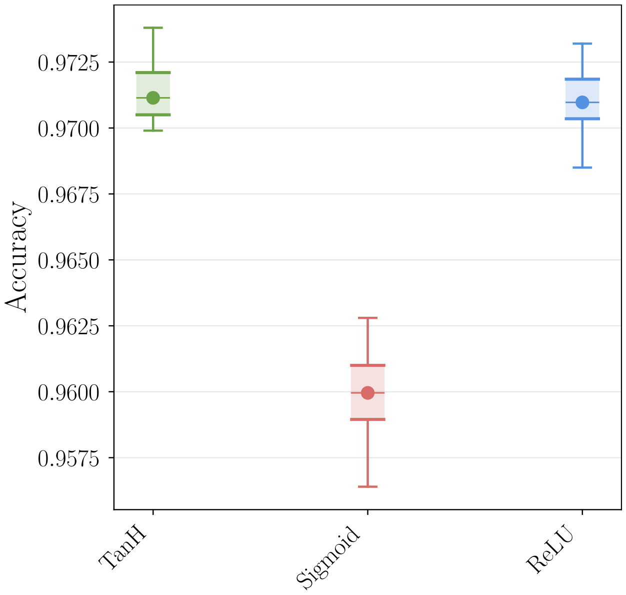

Fig. 5. – Accuracy of the model with different activation functions on MNIST dataset. The model is a sequential model with two hidden layers of 64 then 32 neurons and a softmax output layer. The model is trained with the Adam optimizer, a learning rate of 0.001, a batch size of 128 and 50 epochs. The activation function of the first layer is either TanH, Sigmoid or ReLU.

Batch experiments

Batch size refers to the number of training samples used in a single forward and backward pass during training. It impacts both training stability and computational efficiency. Larger batches make better use of parallel computation (e.g., on GPUs), but too-large batches can lead to poor generalization and slower convergence. They may cause the optimizer to get stuck in sharp minima or plateaus. Smaller batches introduce noise in the gradient estimates, which can help escape local minima but can also destabilize training and convergence stability. We evaluate several batch sizes to assess their impact on training dynamics and final accuracy.

In Fig. 6, we observe that medium-sized batches strike a good balance between convergence speed and final accuracy. These results suggest that batch size is a key hyperparameter for controlling the stability and efficiency of training. 64 samples per batch, even if it achieves the highest accuracy, has a large variance typical of a low number of samples per batch. On MNIST dataset, 128 samples per batch seem to be a good compromise between accuracy and variance.

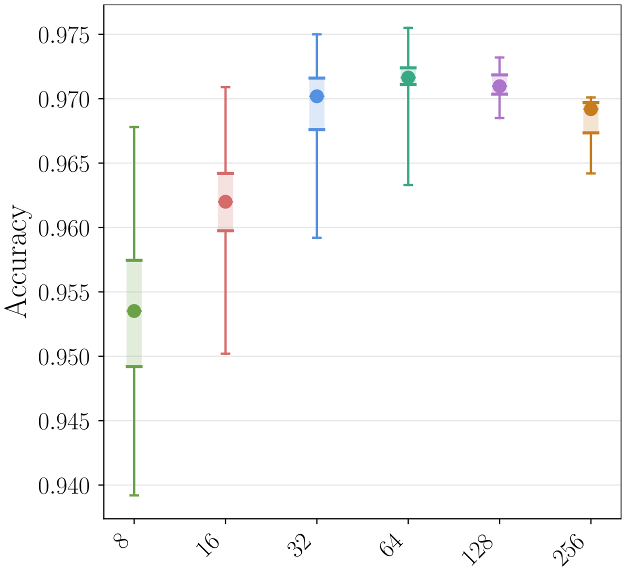

Fig. 6. – Accuracy of the model with different batch sizes on MNIST dataset. The model is a sequential model with two hidden layers of 64 then 32 neurons and a softmax output layer. The model is trained with the Adam optimizer, a learning rate of 0.001 and 50 epochs. The activation function of the first layer is ReLU.

Hidden layers size experiments

The size of each hidden layer determines the capacity of the network to model complex functions. A layer with too few neurons may underfit the data, failing to capture its patterns. Conversely, too many neurons can overfit or introduce unnecessary computation.

We investigate how the number of neurons in hidden layers affects classification performance. We experiment with models containing two hidden layers, each having 16, 32, 64, or 128 neurons.

As shown in Fig. 7, increasing the number of neurons improves accuracy up to a point. The model with [64, 64] hidden neurons achieves high accuracy, while [128, 128] offers only marginal improvement by a few tenths of a percent. Models with [16, 16] or [32, 32] neurons underperform due to insufficient capacity to model the data complexity. These findings highlight the importance of finding an appropriate trade-off between model capacity and overfitting risk.

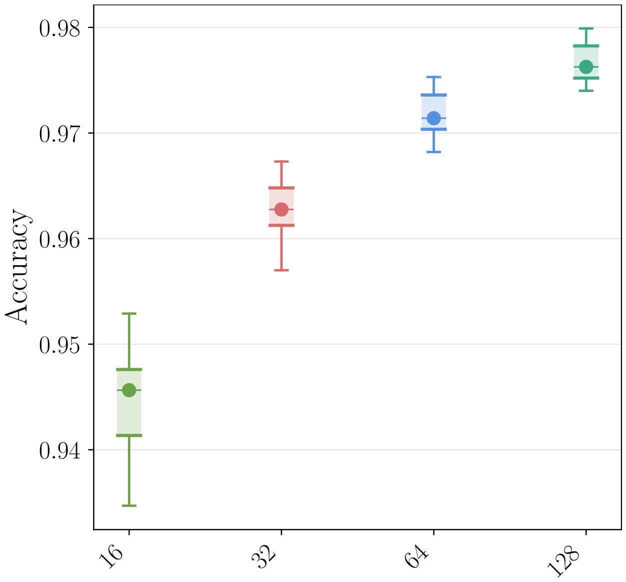

Fig. 7. – Accuracy of the model with different numbers of hidden layers on MNIST dataset. The model is a sequential model with hidden layers of sizes 16, 32, 64, and 128 neurons (e.g., [16, 16], [32, 32], [64, 64], [128, 128]) and a softmax output layer. The model is trained with the Adam optimizer, a learning rate of 0.001, a batch size of 128, and 50 epochs. The activation function of the first layer is ReLU.

Optimizers

The optimizer determines how the model's parameters are updated during training based on gradients. Different optimizers implement different update rules and tradeoffs between speed of convergence, stability, and generalization. We compare the standard SGD, SGD with momentum, and Adam, a popular adaptive gradient method.

SGD (Stochastic Gradient Descent) updates weights using noisy gradient estimates based on mini-batches. It is simple but is slow and can get stuck in local minima. SGD with Momentum incorporates a moving average of past gradients to accelerate convergence and smooth updates. Helps escape shallow minima and navigate ravines. Adam (Adaptive Moment Estimation) combines momentum and per-parameter adaptive learning rates. It maintains estimates of both first and second moments of gradients. Adam is known for fast convergence and robust performance across a wide range of tasks.

Fig. 8 shows that Adam consistently leads to better final accuracy. This reflects the benefit of adaptive learning rates and momentum for deep learning. It is making it well-suited for training deep models. While SGD with momentum improves over vanilla SGD, it still lags behind Adam in performance. These results support the widespread use of Adam as a reliable optimizer in neural network training.

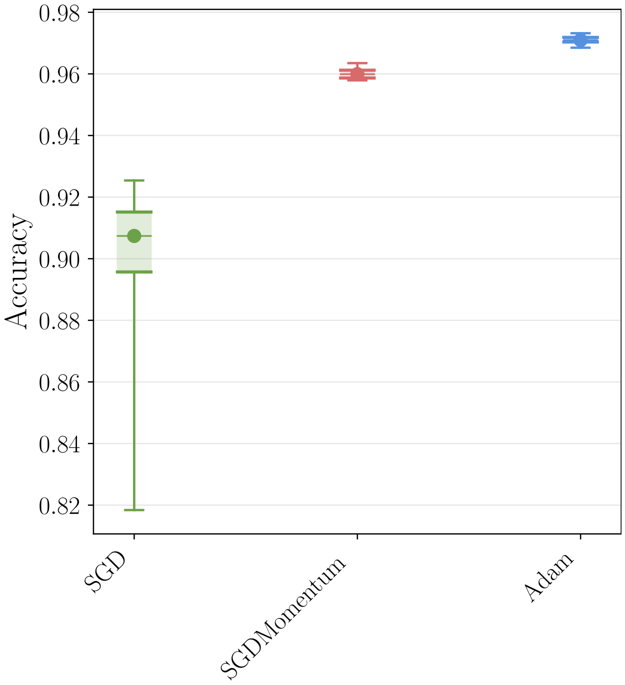

Fig. 8. – Accuracy for the classic SGD optimizer, SGD with Momentum and the Adam optimizer on MNIST dataset. The model is a sequential model with two hidden layers of 64 then 32 neurons and a softmax output layer. The model is trained with a learning rate of 0.001, a batch size of 128 and 50 epochs. The activation function of the first layer is ReLU.