Documentation Feedforward Neural Network

Data Fetching Module (fetch_data.py):

Purpose:

The data fetching module is responsible for acquiring historical financial data and transforming it into a numerical time series suitable for machine learning.

How the Module Works:

The module retrieves historical market data from Yahoo Finance using a specified financial instrument, time interval, and historical period. Once the raw price data is downloaded, missing observations are removed to ensure data consistency. Instead of using raw prices, the module computes "returns" from the closing price.

Preprocessing Module (preprocessing_data.py):

Purpose:

The preprocessing module prepares raw return data for supervised learning by transforming it into model-ready inputs and labels. It also ensures that the data is scaled appropriately for neural network training and split correctly for model evaluation.

Data Normalization:

Neural networks perform best with centered, scaled data. The preprocessing module standardizes returns using z-score normalization:

$$\tilde{x} = \frac{x - \mu}{\sigma}$$

$\boldsymbol{x}$**:** original value
$\boldsymbol{\tilde{x}}$**:** normalized value
$\boldsymbol{\mu}$**:** mean of the data
$\boldsymbol{\sigma}$**:** standard deviation of the data

Denormalization:

After predicting in normalized space, outputs are denormalized to the original return scale for meaningful interpretation and performance evaluation.

$$x = \tilde{x} \cdot \sigma + \mu$$

$\boldsymbol{x}$**:** original value
$\boldsymbol{\tilde{x}}$**:** normalized value
$\boldsymbol{\mu}$**:** mean of the data
$\boldsymbol{\sigma}$**:** standard deviation of the data

Train–Test Splitting:

The dataset is split into training and testing sets by a specified ratio, preserving temporal order. This prevents look-ahead bias and mirrors realistic forecasting conditions.

Sliding Window Construction:

Financial time series are sequential. To turn returns into a supervised learning problem, a sliding window is used: a fixed number of past returns predicts the next return.

Conceptually, this forms input–output pairs:

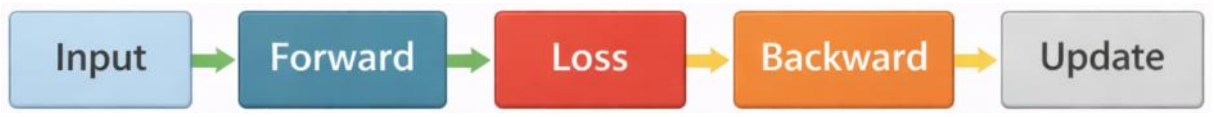
$$[r_t - n, \dots, r_t - 1] \rightarrow \; r_t$$

$\boldsymbol{r_t}$**:** value to predict (target)
n: window size (how many past steps the model sees)

Creation of Layers (Layers.py):

Purpose:

This file defines dense (fully connected) layers, where each neuron is connected to every neuron in the previous layer with its own weight and bias. The class provides the functionality to create layers, perform forward and backward propagation, and update parameters.



Created by ChatGPT

Initialization:

These attributes define the layer's structure, store forward-pass data, and hold gradients required to learn from the loss.

- **Input dimension:** Defines how many inputs features the layer receives per iteration; used to correctly size the weight matrix.
- **Output dimension:** Defines how many neurons the layer has and therefore how many outputs it produces per iteration.
- **weights:** Trainable matrix that scales and combines inputs; initialized with random weights from a normal distribution, that are scaled to stabilize training.
- **biases:** Trainable offset added to the linear output; initialized to zero so learning starts without biasing activations.
- **A previous:** Stores the input from the forward pass; required during backpropagation to compute gradients for the weights.
- **Z:** Stores the linear output (XW + b) before activation; needed to compute activation derivatives during backward pass.
- **A:** Stores the activated output of the layer; used when computing gradients for non-linear activations.

- **self.dB:** Gradient of the loss with respect to the biases; used to update the bias vector during optimization.
- **self.dW:** Gradient of the loss with respect to the weights; used to update the weight matrix during optimization.

Forward Function:

The forward function defines how input data flows through a neural network layer. It takes the input tensor from the previous layer, computes a linear transformation using weights and biases, and then applies an activation function to produce the layer's output.

Linear computation:

The linear computation allows the model to **learn how important each input feature is** by assigning weights to them. It is the mechanism that lets the model form combinations of inputs that are useful for prediction.

$$z = Aw + b$$

A: input to the layer (activations from previous layer or input data)
W: weights of the layer (one per connection)
B: biases of the layer (one per neuron)
Z: linear output (before activation)

Activation function:

Activation functions determine how a neuron's output is passed to the next layer by introducing nonlinearity into the network. Although neurons first compute a linear weighted sum of inputs, activation functions transform this output so the network can learn complex patterns. Without activation functions, a neural network would behave like a single linear model and fail to capture nonlinear relationships in data.

ReLU activation:

ReLU is an activation function type, which only allows positive signals passing through and blocking negative signals by setting them to zero.

$$\mathrm{ReLU}(x) = \max(0,1)$$

Linear activation:

The linear activation function returns the linear output directly without applying any nonlinearity. It is used in the output layer, where the model needs to predict continuous values without restricting the range of the output.

Loss function:

A loss function measures how well a neural network's predictions match the true target values. It provides a single value representing the model's error, which is used during training to adjust the weights and biases through gradient-based optimization.

Mean Squared error (MSE):

MSE calculates the average of the squared differences between predicted and actual values. Squaring the errors penalizes larger mistakes more, helping the network adjust weights to reduce overall error.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

$\boldsymbol{n}$**:** number of samples
$\boldsymbol{\hat{y}_i}$**:** predicted value
$\boldsymbol{y_i}$**:** true value

Backward function:

Backward propagation in a neural network computes the gradients of the loss with respect to all trainable parameters (weights and biases) and the outputs of previous layers. A **gradient** measures how a small change in a parameter affects the loss:

- **Large gradient:** parameter strongly influences output
- **Small gradient:** parameter has little effect (network closer to minimum)

Gradients are necessary to **update weights and biases** to reduce the loss. The process uses the **linear output of each neuron** from the forward pass, showing how each input, weight, and bias contributed to the neuron's output and the final loss.

Backward propagation starts at the **loss** (derivative of MSE) by comparing the network's predictions to the targets and then **propagates backward layer by layer**, applying the **chain rule** to account for both linear transformations and activations.

By systematically computing gradients for **weights, biases, and previous activations**, the network learns how each parameter influenced the loss, providing the information needed to update them and improve the model.

Loss Gradient (MSE):

Measures how far the predicted output is from the target and provides the initial error signal.

$$dA = \frac{2}{m}(A - Y)$$

dA: derivative of the loss with respect to the output
m: number of samples
A: predicted output of the current layer
Y: true target values

Activation Gradient:

Adjusts the error according to the activation function. Only active neurons pass error in ReLU; linear passes it unchanged.

$$dZ = dA \cdot f'(Z)$$

dA: incoming gradient from the next layer (or loss)
f'(Z): derivative of the activation function (e.g., ReLU or linear)
Z: linear output of the neuron before activation
dZ: gradient after applying the activation

Weight Gradient:

Shows how much each weight contributed to the loss and is used to update weights.

$$dW = A_{prev}^{T} \cdot dZ$$

$\boldsymbol{A_{prev}^{T}}$**:** input to the current layer (output of the previous layer)
dZ: gradient after activation
dW: gradient of the weights

Bias Gradient:

Represents how much each bias affected the loss and is updated along with weights.

$$dB = \sum_{i=1}^{m} dZ_i$$

$\boldsymbol{dZ_i}$**:** gradient of the linear output for sample i
dB: gradient of the biases
m: number of samples in the batch

Previous Layer Activation Gradient:

Computes the gradient to pass backward to the previous layer, enabling earlier layers to update correctly.

$$dA_{prev} = dZ \cdot w^{T}$$

dZ: gradient after activation
$\boldsymbol{w^{T}}$**:** weights of the current layer
$\boldsymbol{dA_{prev}}$**:** gradient of the previous layer's output

Update parameter's function:

The update parameter's function changes the weights and biases to decrease the loss.

Weight update:

$$\mathrm{W} \leftarrow \mathrm{W} - \eta\, \mathrm{dW}$$

b: biases of the layer
η: learning_rate
dW: gradient of the loss w.r.t. the weights

Bias update:

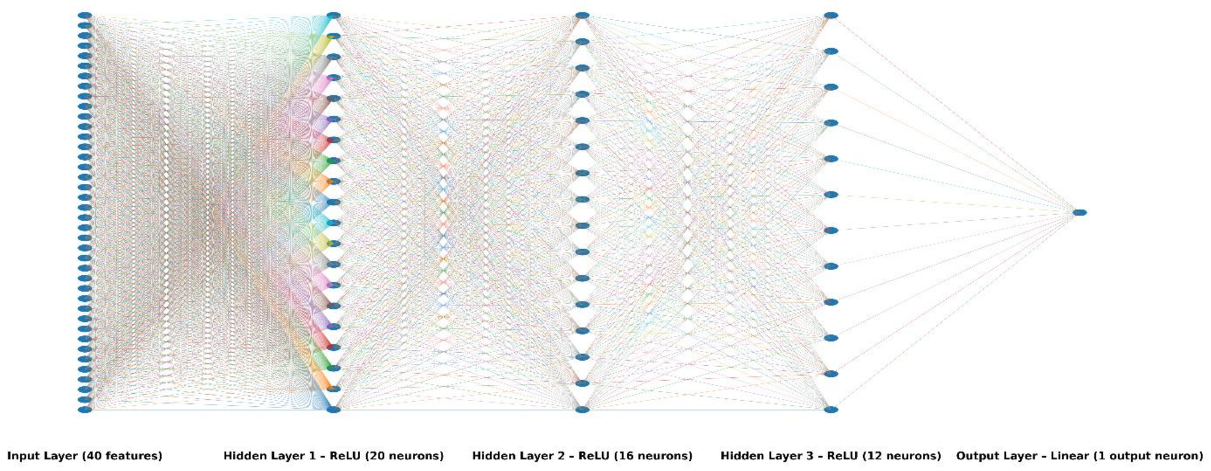
$$\mathrm{b} \leftarrow \mathrm{b} - \eta\, \mathrm{dB}$$

b: biases of the layer
η: learning_rate
dB: gradient of the loss w.r.t. the biases

Creation of the model (ffnn.py):

Here, the feedforward neural network is constructed. It consists of an input layer that receives all input data in the form of tensors X. The input data has 40 features which are defined as window size in main.py (sliding window). The network includes three hidden layers: the first with 20 neurons, the second with 16 neurons, and the third with 12 neurons. These hidden layers use the ReLU activation function to introduce non-linearity. The output layer has a single neuron, since the model predicts future returns based on past returns and uses a linear activation function.

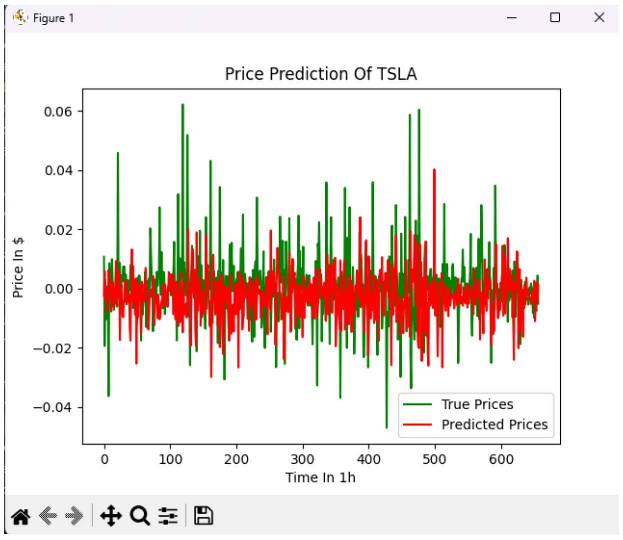
During training, a forward pass is performed in which data flows through all layers to produce an output prediction. After the forward pass, the loss is computed by comparing the predicted output with the true target values. Backpropagation is then applied to compute gradients. Only the output layer directly receives the true target values because the loss is defined at the network's output; the error signal is then propagated backward through the hidden layers. After gradients are computed, the weights and biases of all layers are updated using gradient descent. To monitor training progress, the loss is printed every 100 epochs. Finally, once training is complete, the network is evaluated on unseen test data to generate predictions.


Input Layer (40 features)
Hidden Layer 1 - ReLU (20 neurons)
Hidden Layer 2 - ReLU (16 neurons)
Hidden Layer 3 - ReLU (12 neurons)
Output Layer - Linear (1 output neuron)

Created by ChatGPT

Plotting test data (plot.py):

Here, the network's predictions are compared with the true values by plotting both over time. (example tesla, this shows on all test data, you can zoom in to see details)



Evaluation of the model (evaluation.py):

This file provides functions to quantitatively evaluate the performance of the neural network by measuring how closely its predictions match the true target values.

Mean Squared Error (MSE):

MSE measures the average squared difference between predicted and true values. It penalizes larger errors more heavily.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$\boldsymbol{y_i}$: true value for sample i
$\boldsymbol{\hat{y}_i}$: predicted value for sample i
$\boldsymbol{n}$: total number of samples

Root Mean Squared Error (RMSE):

RMSE is the square root of MSE, giving an error measure in the **same units as the target values**, which makes it easier to interpret than MSE.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} = \sqrt{\text{MSE}}$$

$\boldsymbol{y_i}$: true value for sample i
$\boldsymbol{\hat{y}_i}$: predicted value for sample i
$\boldsymbol{n}$: total number of samples

Mean Absolute Error (MAE):

MAE calculates the average absolute difference between predicted and true values. Unlike MSE, it treats all errors equally, so it is less sensitive to outliers.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} | y_i - \hat{y}_i |$$

$\boldsymbol{y_i}$**:** true value for sample i
$\boldsymbol{\hat{y}_i}$**:** predicted value for sample i
$\boldsymbol{n}$**:** total number of samples

R^2 Score (Coefficient of Determination):

R^2 measures how well the predictions explain the variance of the true data. A value of 1 means perfect prediction, 0 means the model predicts no better than the mean and negative values indicate worse-than-mean predictions.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

$\boldsymbol{y_i}$**:** true value
$\boldsymbol{\hat{y}_i}$**:** predicted value
$\boldsymbol{\bar{y}}$**:** mean of true values
$\boldsymbol{n}$**:** total number of samples
$\boldsymbol{SS_{\text{res}} = \sum(y_i - \hat{y}_i)^2}$**:** residual sum of squares
$\boldsymbol{SS_{\text{tot}} = \sum(y_i - \bar{y})^2}$**:** total sum of squares

Edge case:
If $SS_{\text{tot}} = 0$ (all true values are the same), R^2 is set to 0 to avoid division by zero.

Parameter setting & main file (main.py):

This file serves as the main execution pipeline of the project: it fetches and preprocesses financial time-series data, trains a feedforward neural network to predict future returns, evaluates the model and visualizes the predicted versus true values.

The parameters set in this file include the data-related constants, symbol = "TSLA" to specify the asset, interval = "1h" to define the time resolution, window_size = 40 to determine the number of past observations used as input features, and train_ratio = 0.8 to split the data into training and testing sets, as well as the training hyperparameters learning_rate = 0.1 and epochs = 1000, which control how fast and how long the neural network is trained.

Evaluation of the model:

Stats from output console (example):

```
Epoch 0, Loss: 4.7671967660010415
Epoch 100, Loss: 1.0766797423791212
Epoch 200, Loss: 1.0222011089701386
Epoch 300, Loss: 0.9419033147496666
Epoch 400, Loss: 0.8530885270095552
Epoch 500, Loss: 0.775939711220693
Epoch 600, Loss: 0.7153259653161739
Epoch 700, Loss: 0.6693282193415615
Epoch 800, Loss: 0.6192879728966904
Epoch 900, Loss: 0.5958274087598466
```

MSE: 0.00018939379079667412

RMSE: 0.013762041665271695

MAE: 0.009638527270234967

R^2: -0.5333275293959312

Evaluation of the stats (from example):

Loss: The loss decreases from about 4.8 at the beginning to around 0.6 after 900 epochs. This steady decrease shows that the network is learning from the data and improving its predictions over time by reducing the error.

MSE: The Mean Squared Error is very small because the model predicts log returns, which naturally have low values close to zero. Squaring these small errors results in a low MSE, meaning the predictions are numerically close to the true values, but this does not necessarily imply strong predictive performance in a financial sense.

RMSE: The Root Mean Squared Error is low and reflects the small scale of log returns. Since RMSE is expressed in the same units as the target variable, it mainly indicates that the average prediction error is small in magnitude rather than economically significant.

MAE: The Mean Absolute Error is small, which is expected when predicting log returns. This shows that the average absolute deviation between predictions and true values is limited, but it should be interpreted in the context of the low volatility of returns.

R^2: The R^2 score is negative, which typically indicates poor explanatory power. However, this is common when predicting financial log returns, as they are highly noisy and weakly predictable. Even models that reduce error can produce negative R^2 values in this setting.

General evaluation of the model:

Overall, the model is not suited for real-world financial prediction, as market data is sequential and highly noisy, while the implemented feedforward neural network does not explicitly model temporal dependencies. More advanced architectures such as LSTM networks are better suited for this type of data. Nevertheless, the decreasing loss during training shows that the network can learn and optimize its parameters. To further validate the implementation, the same network architecture was conceptually applied to a simpler task such as handwritten digit recognition using the MNIST dataset, where feedforward networks are known to perform well, indicating that the model structure and training procedure are technically correct.

Improvements that can be made:

Several improvements could enhance the model's performance. Using the Adam optimizer instead of standard gradient descent could lead to faster and more stable convergence. Introducing batch training instead of using the full dataset at once would improve training efficiency and generalization. Adding dropout layers could help reduce overfitting, while increasing the number of neurons or layers could allow the model to learn more complex patterns. Finally, using larger and more diverse datasets, as well as incorporating additional market features, would likely improve the model's predictive capability. Additionally, as mentioned before a feed forward neural network, isn't the most suited network to predict financial market data, so changing to an LSTM model could also improve the stats.

What I have learned from this project:

Through this project, I gained a deeper understanding of the mathematical foundations of artificial intelligence, particularly how neural networks learn through forward propagation, loss functions, backpropagation, and gradient descent. Building a deep learning model from scratch clarified how weights and biases are updated and how each component contributes to reducing the loss. I also learned how data preprocessing, normalization, and windowing strongly affect model performance. Additionally, this project highlighted the limitations of simple feedforward networks when applied to complex, noisy data such as financial markets, and showed the importance of choosing suitable architectures, evaluation metrics and realistic expectations when developing machine learning models.

AI disclosure:

AI tools were used for linguistic support in the documentation, to help understand concepts, assist with debugging, and support the writing of parts of the code. All final content, code, and conclusions were reviewed, verified, and fully understood by the author.

Rescources:

https://chatgpt.com

https://en.wikipedia.org/wiki/Feedforward_neural_network

https://www.youtube.com/watch?v=rEDzUT3ymw4

https://www.youtube.com/watch?v=Fu273ovPBmQ

https://www.youtube.com/shorts/20g0DnJWI3Q

https://www.youtube.com/watch?v=SftOqbMrGfE