Next Word Prediction : Week 1

Team Members :

Nimitt

Pratham Sharda

Siddharth Joshi

Anshul Mantri

Problem Statement

To develop an efficient LSTM based model for next word prediction on Nexys FPGA board.

Weekly Plan

Week 1] Implement the UART communication protocol to send the weights and word vectors.

Week 2] Implement the LSTM with hidden size of 25.

Week 3] The input size of the network can be based on character size and needs to be variable.

Week 4] You can apply the activation function in Verilog and after applying all the gates, bring back the class indexes of the predicted words.

Week 1 Update

We implemented UART communication module for reading and writing data into the FPGA boards. We will use this module to take input and give output back from the FPGA. The followings sections show the corresponding Verilog and Python Codes :

UART Communication

1. **Debouncing Module**

```
Code for debouncing
`timescale 1ns / 1ps

module debounce_explicit(
    input clk_100MHz,
    input reset,
    input btn,              // button input
    output reg db_level,    // for switches
    output reg db_tick      // for buttons
    );

    // state declarations
    parameter [1:0] zero  = 2'b00,
                    wait0 = 2'b01,
                    one   = 2'b10,
                    wait1 = 2'b11;

    // 100MHz clk with a period of 10ns
    // number of counter bits (2^N * 10ns = ~40ms)
    parameter N = 22;

    // signal declaration
    reg [1:0] state_reg, next_state;
    reg [N-1:0] q_reg;
    wire [N-1:0] q_next;
    wire q_zero;
    reg q_load, q_dec;

    // body
    // FSMD state and data registers
    always @(posedge clk_100MHz or posedge reset)
        if(reset) begin
            state_reg <= zero;
            q_reg <= 0;
        end
        else begin
            state_reg <= next_state;
            q_reg <= q_next;
        end

    // FSMD data path (counter) next state logic
    assign q_next = (q_load) ? {N{1'b1}} :       // load all 1s
                    (q_dec) ? q_reg - 1  :       // decrement
                     q_reg;                      // no change in

    // status signal
    assign q_zero = (q_next == 0);

    // FSMD control path next state logic
    always @* begin
        next_state = state_reg;
        q_load = 1'b0;
        q_dec = 1'b0;
        db_tick = 1'b0;

        case(state_reg)
            zero    : begin
                        db_level = 1'b0;
                        if(btn) begin
                            next_state = wait1;
                            q_load = 1'b1;
                        end
            end

            wait1   : begin
                        db_level = 1'b0;
                        if(btn) begin
                            q_dec = 1'b1;
                            if(q_zero) begin
                                next_state = one;
                                db_tick = 1'b1;
                            end
                        end
                        else
                            next_state = zero;
            end

            one     : begin
                        db_level = 1'b1;
                        if(~btn) begin
                            q_dec = 1'b1;
                            if(q_zero)
                                next_state = zero;
                        end
                        else
                            next_state = one;
            end

            default : next_state = zero;
        endcase
    end

endmodule
```

2. **FIFO :** Module to keep a stack for storing data

```
Code for fifo:

`timescale 1ns /1ps

module fifo
    #(
       parameter    DATA_SIZE      = 8,        // number of bits
                    ADDR_SPACE_EXP = 4         // number of addr
    )
    (
       input clk,                                    // FPGA clock
       input reset,                                  // reset button
       input write_to_fifo,                          // signal start w
       input read_from_fifo,                         // signal start r
       input [DATA_SIZE-1:0] write_data_in,          // data word into
       output [DATA_SIZE-1:0] read_data_out,         // data word out
       output empty,                                 // FIFO is empty
       output full                                   // FIFO is full (
);

    // signal declaration
    reg [DATA_SIZE-1:0] memory [2**ADDR_SPACE_EXP-1:0];      // r
    reg [ADDR_SPACE_EXP-1:0] current_write_addr, current_write_a
    reg [ADDR_SPACE_EXP-1:0] current_read_addr, current_read_add
    reg fifo_full, fifo_empty, full_buff, empty_buff;
    wire write_enabled;

    // register file (memory) write operation
    always @(posedge clk)
        if(write_enabled)
            memory[current_write_addr] <= write_data_in;

    // register file (memory) read operation
    assign read_data_out = memory[current_read_addr];

    // only allow write operation when FIFO is NOT full
    assign write_enabled = write_to_fifo & ~fifo_full;

    // FIFO control logic
    // register logic
    always @(posedge clk or posedge reset)
        if(reset) begin
            current_write_addr  <= 0;
            current_read_addr   <= 0;
            fifo_full           <= 1'b0;
            fifo_empty          <= 1'b1;       // FIFO is empty
        end
        else begin
            current_write_addr  <= current_write_addr_buff;
            current_read_addr   <= current_read_addr_buff;
            fifo_full           <= full_buff;
            fifo_empty          <= empty_buff;
        end

    // next state logic for read and write address pointers
    always @* begin
        // successive pointer values
        next_write_addr = current_write_addr + 1;
        next_read_addr  = current_read_addr + 1;

        // default: keep old values
        current_write_addr_buff = current_write_addr;
        current_read_addr_buff  = current_read_addr;
        full_buff  = fifo_full;
        empty_buff = fifo_empty;

        // Button press logic
        case({write_to_fifo, read_from_fifo})     // check both
            // 2'b00: neither buttons pressed, do nothing

            2'b01:  // read button pressed?
                if(~fifo_empty) begin   // FIFO not empty
                    current_read_addr_buff = next_read_addr;
                    full_buff = 1'b0;   // after read, FIFO not
                    if(next_read_addr == current_write_addr)
                        empty_buff = 1'b1;
                end

            2'b10:  // write button pressed?
                if(~fifo_full) begin    // FIFO not full
```

```
                    current_write_addr_buff = next_write_addr;
                    empty_buff = 1'b0;  // after write, FIFO no
                    if(next_write_addr == current_read_addr)
                        full_buff = 1'b1;
                end

            2'b11:  begin   // write and read
                current_write_addr_buff = next_write_addr;
                current_read_addr_buff  = next_read_addr;
                end
        endcase
    end

    // output
    assign full = fifo_full;
    assign empty = fifo_empty;

endmodule
```

3. **Transmitter:** Module to send data to external device

```
Code for transmitter:
`timescale 1ns / 1ps

module uart_transmitter
    #(
        parameter   DBITS = 8,          // number of data bits
                    SB_TICK = 16        // number of stop bit /
    )
    (
        input clk_100MHz,               // basys 3 FPGA
        input reset,                        // reset
        input tx_start,                     // begin data transmissi
        input sample_tick,                  // from baud rate genera
        input [DBITS-1:0] data_in,          // data word from FIFO
        output reg tx_done,                 // end of transmission
        output tx                           // transmitter data line
    );

    // State Machine States
    localparam [1:0]    idle  = 2'b00,
                        start = 2'b01,
                        data  = 2'b10,
                        stop  = 2'b11;

    // Registers
    reg [1:0] state, next_state;            // state registers
    reg [3:0] tick_reg, tick_next;          // number of ticks r
    reg [2:0] nbits_reg, nbits_next;        // number of bits tr
    reg [DBITS-1:0] data_reg, data_next;    // assembled data wo
    reg tx_reg, tx_next;                    // data filter for p

    // Register Logic
    always @(posedge clk_100MHz, posedge reset)
        if(reset) begin
            state <= idle;
            tick_reg <= 0;
            nbits_reg <= 0;
            data_reg <= 0;
            tx_reg <= 1'b1;
        end
        else begin
            state <= next_state;
            tick_reg <= tick_next;
            nbits_reg <= nbits_next;
            data_reg <= data_next;
            tx_reg <= tx_next;
        end

    // State Machine Logic
    always @* begin
        next_state = state;
        tx_done = 1'b0;
        tick_next = tick_reg;
        nbits_next = nbits_reg;
        data_next = data_reg;
        tx_next = tx_reg;

        case(state)
            idle: begin                            // no data in FIFO
                tx_next = 1'b1;                    // transmit idle
                if(tx_start) begin                 // when FIFO is NOT
                    next_state = start;
                    tick_next = 0;
                    data_next = data_in;
                end
            end

            start: begin
                tx_next = 1'b0;                    // start bit
                if(sample_tick)
                    if(tick_reg == 15) begin
                        next_state = data;
                        tick_next = 0;
                        nbits_next = 0;
                    end
                    else
                        tick_next = tick_reg + 1;
            end

            data: begin
                tx_next = data_reg[0];
                if(sample_tick)
```

```
                    if(tick_reg == 15) begin
                        tick_next = 0;
                        data_next = data_reg >> 1;
                        if(nbits_reg == (DBITS-1))
                            next_state = stop;
                        else
                            nbits_next = nbits_reg + 1;
                    end
                    else
                        tick_next = tick_reg + 1;
            end

            stop: begin
                tx_next = 1'b1;         // back to idle
                if(sample_tick)
                    if(tick_reg == (SB_TICK-1)) begin
                        next_state = idle;
                        tx_done = 1'b1;
                    end
                    else
                        tick_next = tick_reg + 1;
            end
        endcase
    end

    // Output Logic
    assign tx = tx_reg;

endmodule
```

4. **Receiver** : Module to receive data from the external device

```
module uart_receiver
    #(
        parameter   DBITS = 8,          // number of data bits i
                    SB_TICK = 16        // number of stop bit /
    )
    (
        input clk_100MHz,               // basys 3 FPGA
        input reset,                    // reset
        input rx,                       // receiver data line
        input sample_tick,              // sample tick from baud
        output reg data_ready,          // signal when new data
        output [DBITS-1:0] data_out     // data to FIFO
    );

    // State Machine States
    localparam [1:0] idle  = 2'b00,
                     start = 2'b01,
                     data  = 2'b10,
                     stop  = 2'b11;

    // Registers
    reg [1:0] state, next_state;        // state registers
    reg [3:0] tick_reg, tick_next;      // number of ticks recei
    reg [2:0] nbits_reg, nbits_next;    // number of bits receiv
    reg [7:0] data_reg, data_next;      // reassembled data word

    // Register Logic
    always @(posedge clk_100MHz, posedge reset)
        if(reset) begin
            state <= idle;
            tick_reg <= 0;
            nbits_reg <= 0;
            data_reg <= 0;
        end
        else begin
            state <= next_state;
            tick_reg <= tick_next;
            nbits_reg <= nbits_next;
            data_reg <= data_next;
        end

    // State Machine Logic
    always @* begin
        next_state = state;
        data_ready = 1'b0;
        tick_next = tick_reg;
        nbits_next = nbits_reg;
        data_next = data_reg;

        case(state)
            idle:
                if(~rx) begin              // when data line g
                    next_state = start;
                    tick_next = 0;
                end
            start:
                if(sample_tick)
                    if(tick_reg == 7) begin
                        next_state = data;
                        tick_next = 0;
                        nbits_next = 0;
                    end
                    else
                        tick_next = tick_reg + 1;
            data:
                if(sample_tick)
                    if(tick_reg == 15) begin
                        tick_next = 0;
                        data_next = {rx, data_reg[7:1]};
                        if(nbits_reg == (DBITS-1))
                            next_state = stop;
                        else
                            nbits_next = nbits_reg + 1;
                    end
                    else
                        tick_next = tick_reg + 1;
            stop:
                if(sample_tick)
                    if(tick_reg == (SB_TICK-1)) begin
                        next_state = idle;
                        data_ready = 1'b1;
                    end
                    else
                        tick_next = tick_reg + 1;
        endcase
    end

    // Output Logic
    assign data_out = data_reg;

endmodule
```

5. **UART Top Module** : combining the above modules

```
Code for UART top module:
`timescale 1ns / 1ps

// For 115,200 baud with 100MHz FPGA clock:
// 115,200 * 16 = 1,843,200
// 100 * 10^6 / 1,843,200 = ~55     (counter limit M)
// log2(52) = 6                     (counter bits N)

module uart_top
    #(
        parameter   DBITS = 8,          // number of data bits i
                    SB_TICK = 16,       // number of stop bit /
                    BR_LIMIT = 52,     // baud rate generator co
                    BR_BITS = 6,       // number of baud rate ge
                    FIFO_EXP = 2        // exponent for number o
    )
    (
        input clk_100MHz,                   // FPGA clock
        input reset,                        // reset
        input read_uart,                    // button
        input write_uart,                   // button
        input rx,                           // serial data in
        input [DBITS-1:0] write_data,       // data from Tx FIFO
        output rx_full,                     // do not write data to
        output rx_empty,                    // no data to read from
        output tx,                          // serial data out
        output [DBITS-1:0] read_data        // data to Rx FIFO
    );

    // Connection Signals
    wire tick;                              // sample tick from baud
    wire rx_done_tick;                      // data word received
    wire tx_done_tick;                      // data transmission com
    wire tx_empty;                          // Tx FIFO has no data t
    wire tx_fifo_not_empty;                 // Tx FIFO contains data
    wire [DBITS-1:0] tx_fifo_out;           // from Tx FIFO to UART
    wire [DBITS-1:0] rx_data_out;           // from UART receiver to

    // Instantiate Modules for UART Core
    baud_rate_generator
        #(
            .M(BR_LIMIT),
            .N(BR_BITS)
         )
        BAUD_RATE_GEN
        (
            .clk_100MHz(clk_100MHz),
            .reset(reset),
            .tick(tick)
         );
```

```
uart_receiver
    #(
        .DBITS(DBITS),
        .SB_TICK(SB_TICK)
     )
     UART_RX_UNIT
     (
        .clk_100MHz(clk_100MHz),
        .reset(reset),
        .rx(rx),
        .sample_tick(tick),
        .data_ready(rx_done_tick),
        .data_out(rx_data_out)
     );

uart_transmitter
    #(
        .DBITS(DBITS),
        .SB_TICK(SB_TICK)
     )
     UART_TX_UNIT
     (
        .clk_100MHz(clk_100MHz),
        .reset(reset),
        .tx_start(tx_fifo_not_empty),
        .sample_tick(tick),
        .data_in(tx_fifo_out),
        .tx_done(tx_done_tick),
        .tx(tx)
     );

fifo
    #(
        .DATA_SIZE(DBITS),
        .ADDR_SPACE_EXP(FIFO_EXP)
```

```
          )
          FIFO_RX_UNIT
          (
             .clk(clk_100MHz),
             .reset(reset),
             .write_to_fifo(rx_done_tick),
             .read_from_fifo(read_uart),
             .write_data_in(rx_data_out),
             .read_data_out(read_data),
             .empty(rx_empty),
             .full(rx_full)
           );

     fifo
         #(
             .DATA_SIZE(DBITS),
             .ADDR_SPACE_EXP(FIFO_EXP)
          )
          FIFO_TX_UNIT
          (
             .clk(clk_100MHz),
             .reset(reset),
             .write_to_fifo(write_uart),
             .read_from_fifo(tx_done_tick),
             .write_data_in(write_data),
             .read_data_out(tx_fifo_out),
             .empty(tx_empty),
             .full()                // intentionally disconnecte
           );

     // Signal Logic
     assign tx_fifo_not_empty = ~tx_empty;

endmodule
```

Then, we also implemented an LSTM model. We trained it over Microsoft Stock Data and then made predictions using the model. The model could appropriately fit the Stock data and capture the trends.

LSTM for stock Prediction

Python Code of the model :

```
# LSTM

class LSTM:
    def __init__(self, input_size, hidden_size, output_size):
        # Hyperparameters
        self.hidden_size = hidden_size

        # Forget Gate
        self.wf = initWeights(input_size, hidden_size)
        self.bf = np.zeros((hidden_size, 1))

        # Input Gate
        self.wi = initWeights(input_size, hidden_size)
        self.bi = np.zeros((hidden_size, 1))

        # Candidate Gate
        self.wc = initWeights(input_size, hidden_size)
        self.bc = np.zeros((hidden_size, 1))

        # Output Gate
        self.wo = initWeights(input_size, hidden_size)
        self.bo = np.zeros((hidden_size, 1))

        # Final Gate
        self.wy = initWeights(hidden_size, output_size)
        self.by = np.zeros((output_size, 1))

    # Reset Network Memory
    def reset(self):
        self.concat_inputs = {}

        self.hidden_states = {-1:np.zeros((self.hidden_size, 1)
        self.cell_states = {-1:np.zeros((self.hidden_size, 1))}

        self.activation_outputs = {}
        self.candidate_gates = {}
        self.output_gates = {}
        self.forget_gates = {}
        self.input_gates = {}
        self.outputs = {}

    # Forward Propogation
    def forward(self, X_):
        outputs = []
        self.reset()
        for q in range(len(X_)):

            self.concat_inputs[q] = np.concatenate((self.hidden_

            self.forget_gates[q] = sigmoid(np.matmul(self.wf, se
            self.input_gates[q] = sigmoid(np.matmul(self.wi, sel
            self.candidate_gates[q] = tanh(np.matmul(self.wc, se
            self.output_gates[q] = sigmoid(np.matmul(self.wo, se

            self.cell_states[q] = self.forget_gates[q] * self.ce
            self.hidden_states[q] = self.output_gates[q] * tanh

            outputs += [np.matmul(self.wy, self.hidden_states[q

        return np.array(outputs)

    # Backward Propogation
    def backward(self, errors, X, lr):
        d_wf, d_bf = 0, 0
        d_wi, d_bi = 0, 0
        d_wc, d_bc = 0, 0
        d_wo, d_bo = 0, 0
        d_wy, d_by = 0, 0


        dh_next, dc_next = np.zeros_like(self.hidden_states[0]),
        for q in reversed(range(len(X))):

            error = errors[q]

            # Final Gate Weights and Biases Errors
            d_wy += np.matmul(error, self.hidden_states[q].T)
            d_by += error

            # Hidden State Error
            d_hs = np.matmul(self.wy.T, error) + dh_next

            # Output Gate Weights and Biases Errors
            d_o = tanh(self.cell_states[q]) * d_hs * sigmoid(se
            d_wo += np.matmul(d_o, X[q].T)
            d_bo += d_o

            # Cell State Error
            d_cs = tanh(tanh(self.cell_states[q]), derivative =

            # Forget Gate Weights and Biases Errors
            d_f = d_cs * self.cell_states[q - 1] * sigmoid(self
            d_wf += np.matmul(d_f, X[q].T)
            d_bf += d_f

            # Input Gate Weights and Biases Errors
            d_i = d_cs * self.candidate_gates[q] * sigmoid(self
            d_wi += np.matmul(d_i, X[q].T)
            d_bi += d_i
            # Candidate Gate Weights and Biases Errors
            d_c = d_cs * self.input_gates[q] * tanh(self.candida
            d_wc += np.matmul(d_c, X[q].T)
            d_bc += d_c

            # Concatenated Input Error (Sum of Error at Each Ga
            d_z = np.matmul(self.wf.T, d_f) + np.matmul(self.wi

            # Error of Hidden State and Cell State at Next Time
            dh_next = d_z[:self.hidden_size, :]
            dc_next = self.forget_gates[q] * d_cs

        for d_ in (d_wf, d_bf, d_wi, d_bi, d_wc, d_bc, d_wo, d_
            np.clip(d_, -1, 1, out = d_)

        self.wf += d_wf * lr
        self.bf += d_bf * lr

        self.wi += d_wi * lr
        self.bi += d_bi * lr

        self.wc += d_wc * lr
        self.bc += d_bc * lr

        self.wo += d_wo * lr
        self.bo += d_bo * lr

        self.wy += d_wy * lr
        self.by += d_by * lr

    # predict
    def predict(self,X):
        out = []
        for i in range(len(X)):
            out.append(self.forward(X[i]))
        return np.array(out)
    # Train
    def train(self, X, y, epochs,lr):

        y_new = np.hstack((X,y))
        y_new = y_new[:,1:]


        for _ in range(epochs):
            for i in range(len(X)):
                prediction = self.forward(X[i])
                errors = []
                for q in range(len(prediction)):
                    errors += [y_new[i][q]- prediction[q]]
                errors = np.array(errors)
                self.backward(errors, self.concat_inputs,lr)
            current_loss = np.mean(abs(errors))
            print(current_loss)
```

We trained the model on Microsoft Stocks Data and then tested its performance.

```
# Creating Dataset

df = pd.read_csv('Datasets/MSFT.csv')
df = df[['Date', 'Close']]
# X = df.loc['2024-01-02':'2024-03-14']['Close'].to_numpy()
X = df['Close'].to_numpy()
X = X.reshape(1,len(X),1)
y = X[:,-1].reshape(-1,1,1)
X = X[:,:-1,:]
```

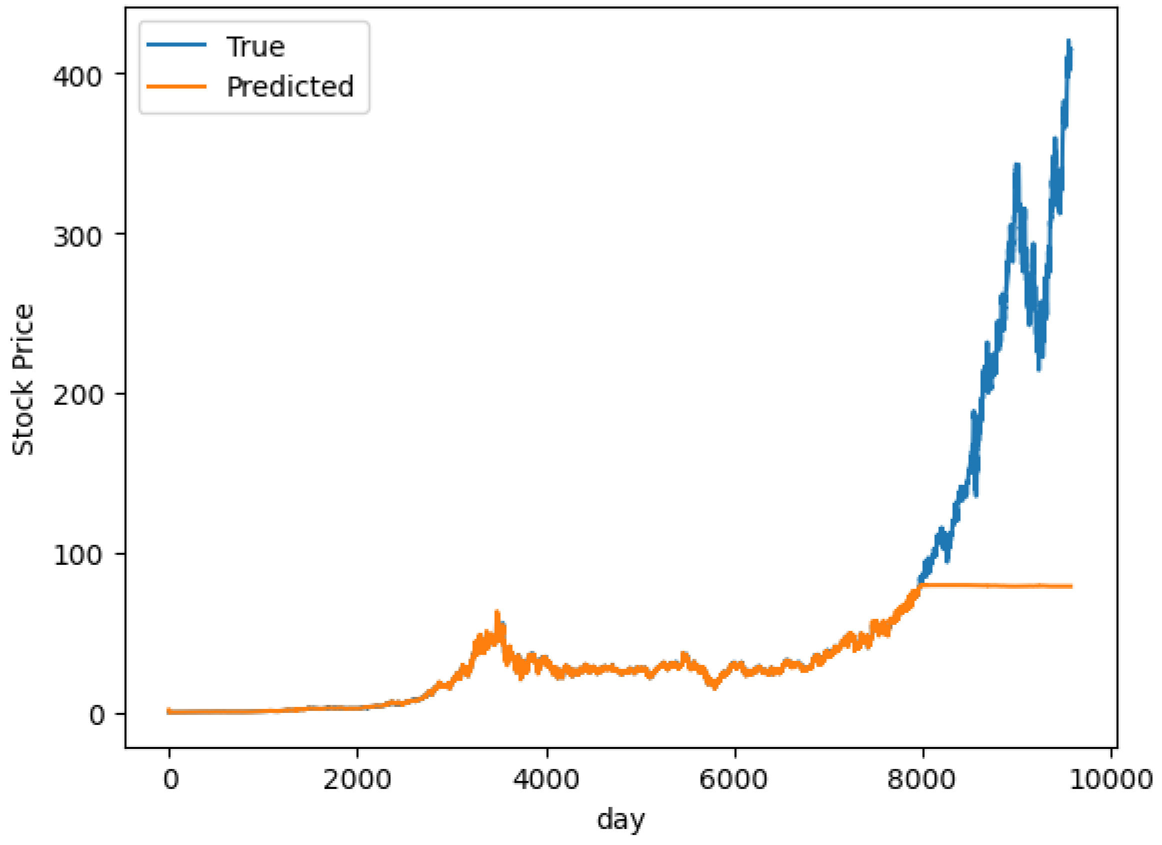
```
# Instantiating the model

hidden_size = 25
lstm = LSTM(len(X[0][0])+hidden_size,hidden_size,1)
```

```
# Training the model

lstm.train(X,y,1000,0.1)
```

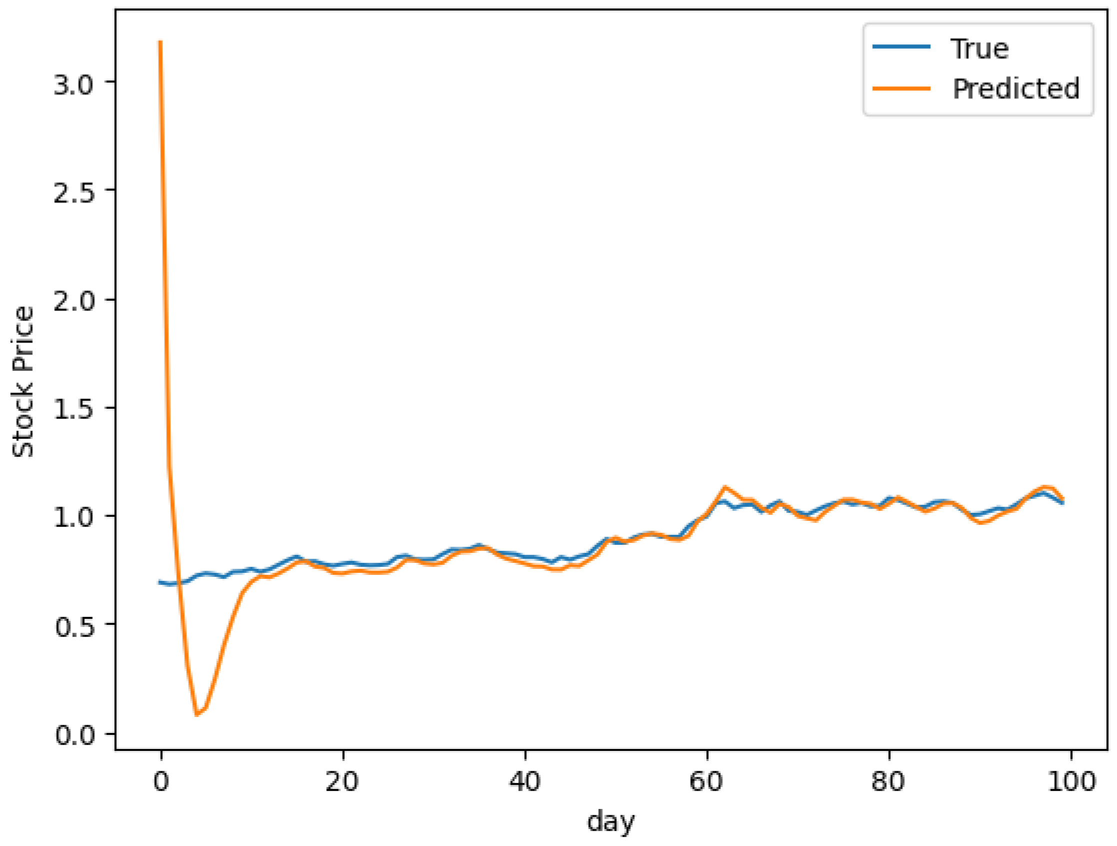
```
# Predicting
y_hat = lstm.predict(X)
```

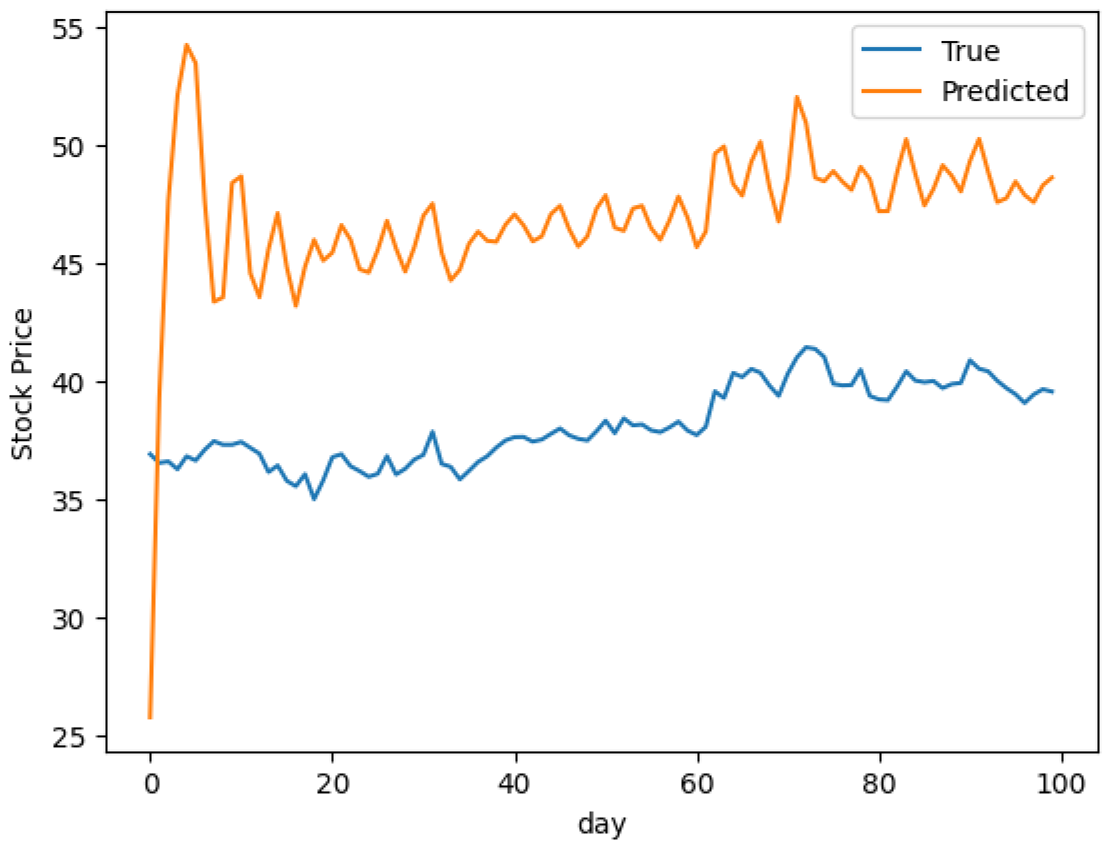
```
# Plotting

sr = np.arange(len(X[0][:]))
plt.plot(sr, X[0][:])
plt.plot(sr, y_hat[0][:,0,0])
```

Performance on Training Data



Performance on Test Data





Link to PPT:

https://docs.google.com/presentation/d/1FPQGWc3mQelemDOM8e9sxPmUa-1ezmO0W3lN4wPpaMQ/edit?usp=sharing