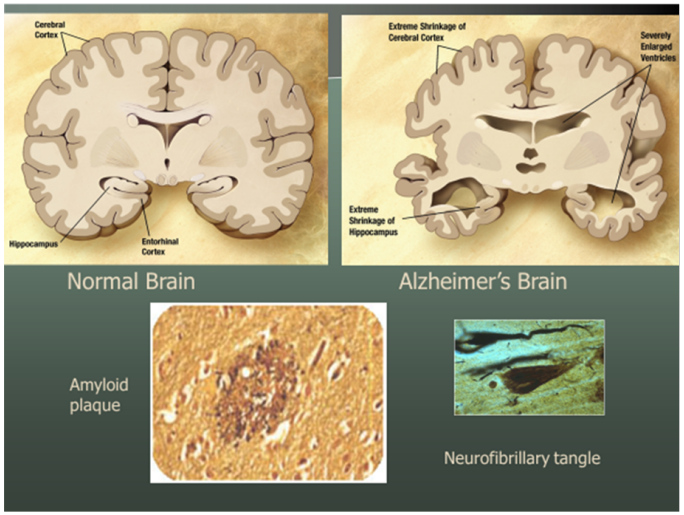
Using Longitudinal Clinical Features to Predict AD Brain Pathologies with LSTM

Qile Dai

2025-01-21

Motivation: Predict Brain Pathology for Timely Intervention



- Brain pathology in Alzheimer's disease (AD) begins early, often before symptoms.
- Early detection could enable intervention of AD, potentially slowing cognitive decline.
- This motivates the need to predict brain pathology, aiming for timely intervention to slow AD progression.

Project: Predict Brain Pathologies using Clinical Features

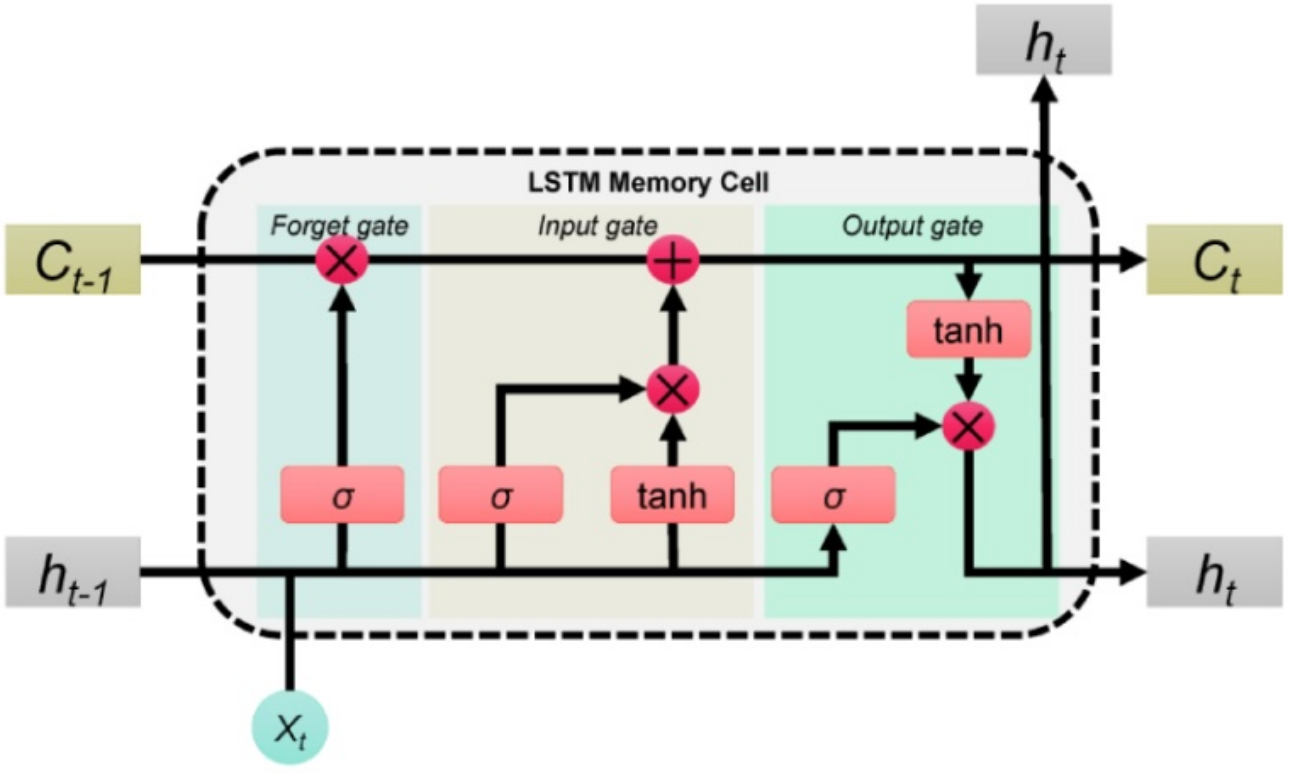
- Data: ROS/MAP longitudinal study; ~3700 Participants with no dementia were enrolled at study baseline and followed-up annually
- Goal: use longitudinal clinical features to predict brain pathologies
  - Predictors: 57 clinical features including, cognition test scores, underlying health condition, demographics variables, etc.
  - Outcomes: Deceased participants were profiled for brain pathologies (amyloid, tangles, gpath, NIA-Reagan)
  - Sample size: total 11,905 visits from 1,214 subjects
    - Training data: 9,490 visits from 971 (80%) subjects
    - Testing data: 2,415 visits from 243 (20%) subjects
- Challenges: number of visits are different across subjects

Model: Long Short-Term Memory (LSTM) Model

Long Short-Term Memory (LSTM) is a type of neural network designed to model sequential data. It has

- a cell state c_t represents the long-term memory
- a hidden state h_t represents the short-term memory at time t
- three gates to regulate how information flows through time:
  - Forget Gate – how much of the old memory to keep
  - Input Gate – how much new information to add to the memory
  - Output Gate – how much to pass the memory to next time point

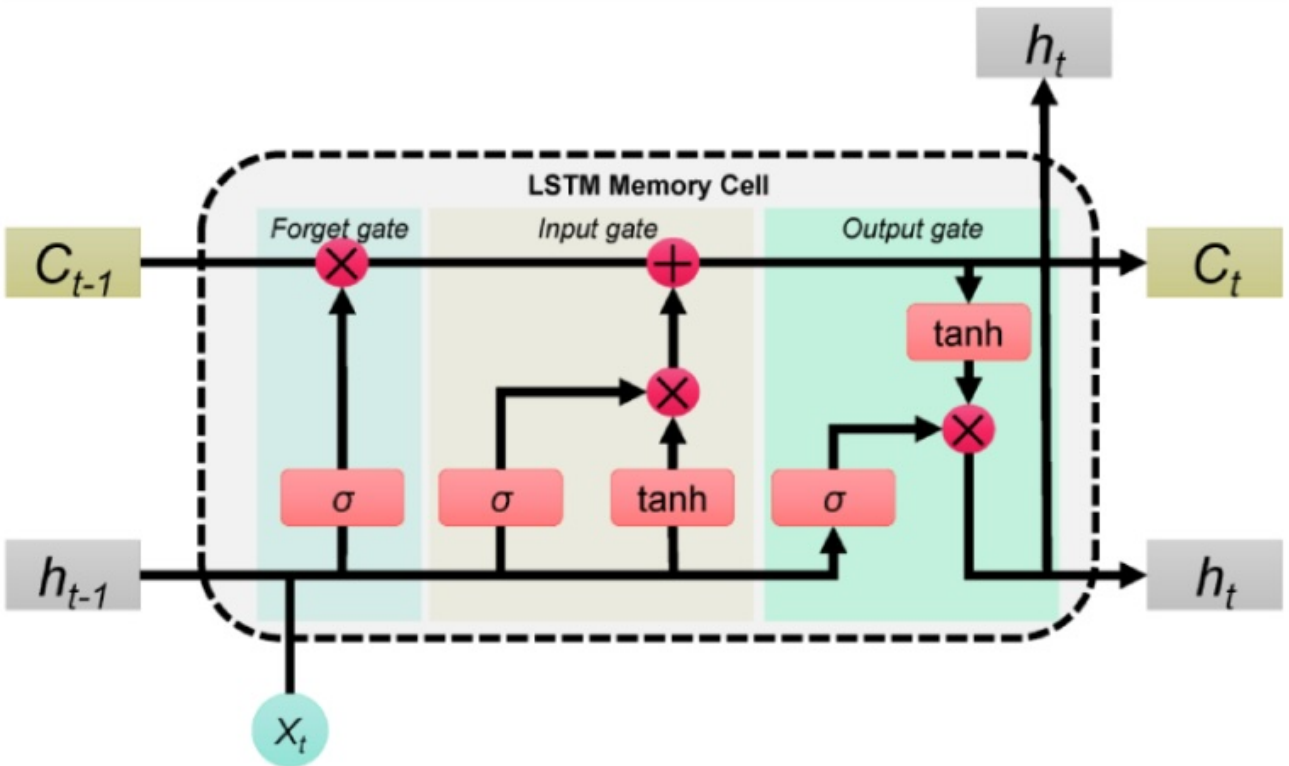
Inputs and outputs of a LSTM cell



At time step t:

- the input of a LSTM cell include: current predictors x_t, previous hidden state h_{t-1}, and previous cell state c_{t-1}

Inputs and outputs of a LSTM cell



At time step t:

- the input of a LSTM cell include: current predictors x_t, previous hidden state h_{t-1}, and previous cell state c_{t-1}
- the output include current hidden state h_t, and current cell state c_t

How the Cell State c_t is Computed

1. Forget Gate (f_t)

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

- $x_t \in \mathbb{R}^n$: Current predictors, where n is the dimensionality of the input.
- $h_{t-1} \in \mathbb{R}^m$: Previous hidden state, where m is the dimensionality of the hidden state.
- $W_f \in \mathbb{R}^{m \times n}$: Weight matrix for the input.
- $U_f \in \mathbb{R}^{m \times m}$: Weight matrix for the hidden state.
- $b_f \in \mathbb{R}^m$: Bias term.

It uses x_t and h_{t-1} to computes $f_t \in \mathbb{R}^m$ with element-wise values in $[0, 1]$

How the Cell State c_t is Computed

2. Input Gate (i_t)

$$i_t = \sigma\big(W_i x_t + U_i h_{t-1} + b_i\big)$$

It uses x_t and h_{t-1} to computes $i_t \in \mathbb{R}^m$ with element-wise values in $[0, 1]$

3. **Candidate cell state:**

$$\tilde{c}_t = \tanh\big(W_c x_t + U_c h_{t-1} + b_c\big)$$

It uses x_t and h_{t-1} to computes $\tilde{c}_t \in \mathbb{R}^m$ with element-wise values in $[-1, 1]$

How the new Cell State c_t is Computed

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

- f_t: forget gate ([0,1])
- c_{t-1}: previous cell state
- i_t: input gate ([0,1])
- $\tilde{c}_t$: candidate cell state
- $\odot$ denotes element-wise multiplication.

new cell state c_t = retaining part of the old state c_{t-1} + selected part of new information $\tilde{c}_t$.

- f_t thus can be interpreted as how much information from the old state is kept
- i_t thus can be interpreted as how much new information from the candidate cell state $\tilde{c}_t$ is added.

How the new Hidden State (h_t) is Computed

1. Output Gate (o_t)

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

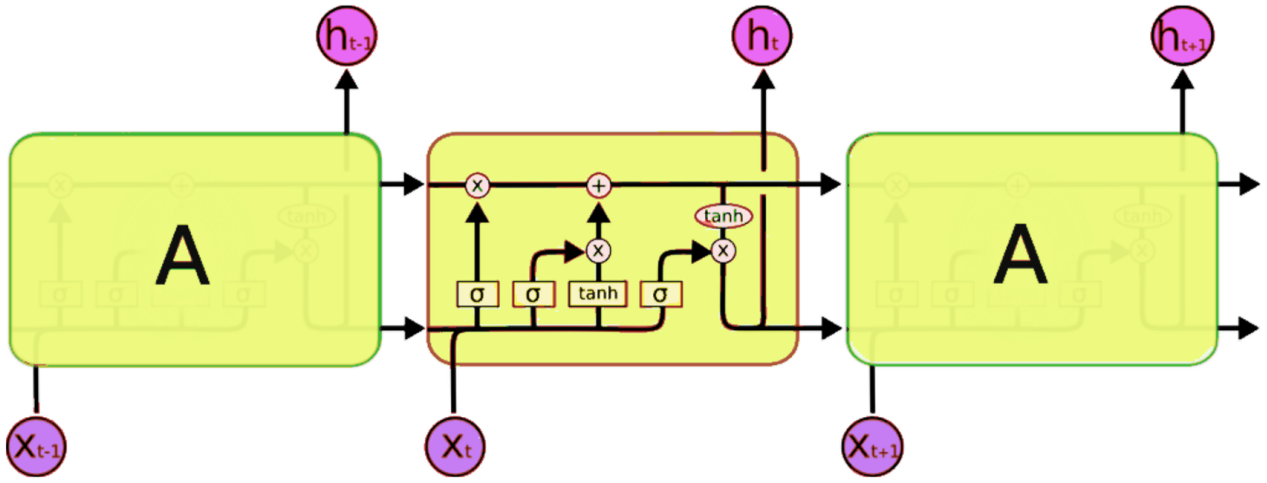
2. Compute Hidden State (h_t)

$$h_t = o_t \odot \tanh(c_t)$$

new hidden state h_t = selective information from new cell state c_t

- *tanh*() scaling the cell state values to [-1, 1]
- selection is controlled by the output gate (o_t)

connect LSTM cells at different t



- We previously introduced an LSTM cell time point t
- When we build a full LSTM model for a time series, these cells are connected across time points, creating a recurrent structure. The updated h_t and c_t are used as the input of the next LSTM Cell at $t+1$
- The weight matrices (e.g. W_f, U_f) and biases (e.g. b_f) are shared across time points. Thus, LSTM model can handle samples with variable lengths of time points.

Full model: LSTM layer + fully connected layer

1. **LSTM Layer**:
   - An LSTM processes take the input x_t, producing a hidden state h_t and a cell state c_t at each time step t:

$$h_t, c_t = \text{LSTM}(x_t, h_{t-1}, c_{t-1})$$

   - We use the hidden state from the **last time step** as the sequence's representation:

$$\mathbf{h}_{\text{final}} = \mathbf{h}_T$$

Full model: LSTM layer + fully connected layer

1. **LSTM Layer**:
   - An LSTM processes take the input x_t, producing a hidden state h_t and a cell state c_t at each time step t:

$$h_t, c_t = \text{LSTM}(x_t, h_{t-1}, c_{t-1})$$

   - We use the hidden state from the **last time step** as the sequence's representation:

$$\mathbf{h}_{\text{final}} = \mathbf{h}_T$$

2. **Fully Connected Layer**:
   - The hidden state from the LSTM is passed through a fully connected neural network layer to transform it into the desired output dimension:

$$\mathbf{y} = \mathbf{W}\mathbf{h}_{\text{final}} + \mathbf{b}$$

   - Here:
     - $\mathbf{W}$: Weight matrix of the fully connected layer.
     - $\mathbf{b}$: Bias vector.
     - $\mathbf{y}$: Final output (e.g., predictions of brain pathology at time T).

Loss function

- For continous brain pathology scores (gpath and tangles), we use Mean Squared Error (MSE) as loss function

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

- N: The number of subjects.
- y_i: The true brain pathology value at the last time point T for the i-th subject.
- $\hat{y}_i$: The predicted brain pathology value at the last time point T for the i-th sequence in the batch, produced by the model.

Use loss function to learn parameters: weights and biases

1. **Initialization**:
   - The model starts with random weights and biases for the LSTM and fully connected layers.
2. **Forward Pass**:
   - Inputs pass through the LSTM and fully connected layer to generate predictions ($\hat{y}_i$).
3. **Loss Calculation**: Compute the MSE
4. **Backward Pass**:
   - Gradients of the MSE loss with respect to each weight and bias ($\frac{\partial \text{MSE}}{\partial \theta}$) are computed.
   - Parameters are updated in the direction of the negative gradient:

$$\theta \leftarrow \theta - \eta \frac{\partial \text{MSE}}{\partial \theta}$$

     - θ: Weight or bias.
     - η: Learning rate.
5. **Iterative Learning**:
   - Repeat steps 2–5 over multiple times to minimize the MSE.

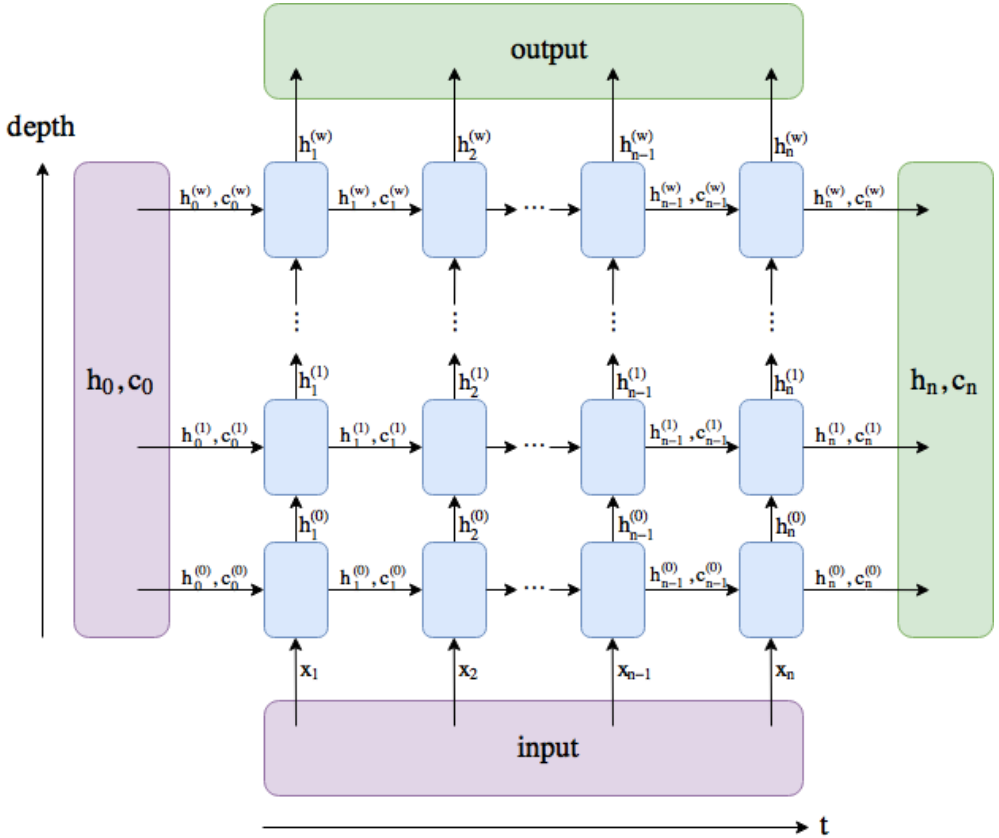
Select hyperparamters

- **Weights and Biases**: These are the parameters that the model learns during training to minimize the loss function.
- **Hyperparameters**: These are parameters **set before training** that control the model's structure (e.g., number of layers, hidden units) or the training process (e.g., learning rate, batch size).

Select hyperparamters

- **Weights and Biases**: These are the parameters that the model learns during training to minimize the loss function.
- **Hyperparameters**: These are parameters **set before training** that control the model's structure (e.g., number of layers, hidden units) or the training process (e.g., learning rate, batch size).
- For example, $h_t \in \mathbb{R}^m$. m is a hyperparameter (hidden units) that can be tuned. More units allow the model to capture complex patterns but increase computation and risk overfitting.

Hyperparamter: Number of Layers



- Number of Layers k: number of blue rectangles in the figure
  - At the time point t, we have k LSTM layers stacked on top of each other. The output of the first LSTM layer becomes the input to the second layer...

Hyperparamter: Batch Size

- Batch Size: the number of training samples processed together in a single forward pass and backpropagation step during the training of a neural network.

Hyperparamter: Learning Rate

- Learning Rate: controls the step size at which the model updates its parameters.
- For example, in gradient-based optimization methods (e.g., Stochastic Gradient Descent, Adam), the parameters (θ) of the model are updated using the gradient of the loss function (∇L):

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \nabla L(\theta_{\text{old}})$$

- η: Learning rate determines how far the weights move in the direction of the gradient during each update.
  - Large steps may overshoot the optimal point
  - Small steps may lead to slow convergence, requiring many iterations to reach the minimum.

Hyperparamter: Dropout Rate

- Dropout Rate p: a value between 0 and 1
  - During training, for each forward pass, hidden units are randomly deactivated (set $= 0$) with probability p. Only the remaining hidden units contribute to training.
  - Purpose: avoids relying too heavily on specific hidden units and let the network learns more robust features

Example of Dropout in LSTM

- Suppose a layer has the following hidden units before dropout: [0.8, 0.4, 0.6, 0.9]
- dropout rate (p) is 50%: randomly deactivate half of the hidden units [0.8, 0, 0, 0.9]

Tune hyperparameters using Cross-validation

Steps:

1. Split dataset into 5 equal parts (called folds).
2. Use 4 folds for training and 1 fold for testing. Repeat this process 5 times, using a different fold for testing each time.
3. For each parameter setting, calculate the performance (e.g., testing R^2) on the test fold.
4. Compute the average performance across the 5 folds for each parameter setting.
5. Choose the parameter setting with the highest average performance.

Results: best hyperparameters in Cross-validation

Table 1: Top 5 highest average R^2 values for gpath prediction across 5 folds

hidden_size	num_layers	dropout_rate	batch_size	learning_rate	gpath
8	2	0.5	8	0.005	0.259
16	4	0.2	16	0.010	0.245
16	2	0.6	8	0.005	0.241
8	2	0.2	8	0.005	0.240
8	2	0.4	8	0.001	0.239

Selected hyperparameters for gpath:

- hidden units: 8
- number of layers: 2
- drop out rate: 0.5
- batch size: 8
- learning rate: 0.005

Results for gpath

- Training LSTM with selected hyperparameters
  - the testing R^2 by LSTM for gpath is 0.308

Results for gpath

- Training LSTM with selected hyperparameters
  - the testing R^2 by LSTM for gpath is 0.308
- Compared to a baseline model that cannot fully utilize the longitudinal nature of the data
  - Elastic-net regression, using data from the last recorded visit before death
  - the testing R^2 by Elastic-net regression for gpath is: 0.262

Results for gpath

- Training LSTM with selected hyperparameters
  - the testing R^2 by LSTM for gpath is 0.308
- Compared to a baseline model that cannot fully utilize the longitudinal nature of the data
  - Elastic-net regression, using data from the last recorded visit before death
  - the testing R^2 by Elastic-net regression for gpath is: 0.262
- Improvement with LSTM
  - By fully utilizing longitudinal data, LSTM improved the prediction R^2 by $(0.308 - 0.262)/0.262 = 17.5\%$.

However, LSTM does not always work better...

Table 2: Top 5 highest average R^2 values for gpath prediction across 5 folds

hidden_size	num_layers	dropout_rate	batch_size	learning_rate	gpath
8	3	0.2	8	0.010	0.235
16	2	0.6	8	0.005	0.241
8	2	0.6	16	0.010	0.211
8	2	0.5	8	0.005	0.259
8	2	0.4	8	0.010	0.229

Selected hyperparameters for tangles:

- hidden units: 8
- number of layers: 3
- drop out rate: 0.2
- batch size: 8
- learning rate: 0.01

Results for tangles

- the testing R^2 by LSTM for tangles is 0.347
- the testing R^2 by Elastic-net regression for tangles is: 0.385
- help! thoughts? overfitting?

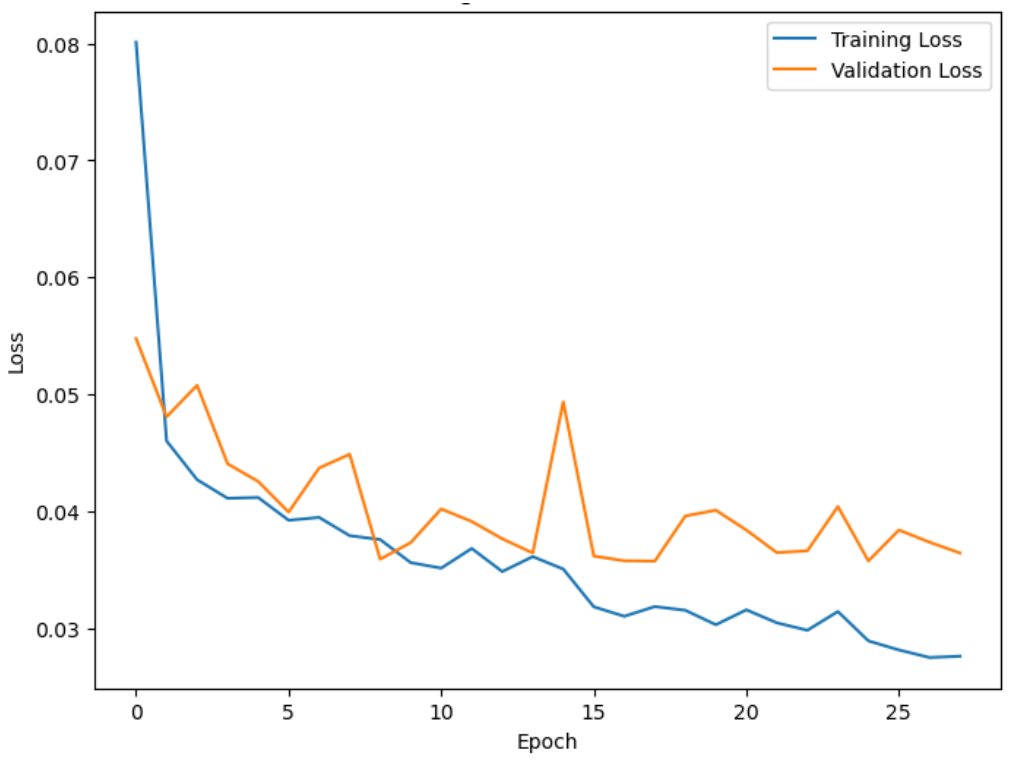


Figure 1: Training loss vs. validation loss by LSTM

Acknowledgment

- Dr. Jingjing Yang
- Conglin Bao
- Yingte Liu
- More details on github: https://github.com/daiqile96/AD_Pathology_Prediction

Notes on Batch Normalization

Batch normalization (BN) is a technique that normalizes the inputs of each layer in a neural network to have zero mean and unit variance, improving training stability and speed.

- During training, it uses mini-batch statistics for normalization
- During validation, it relies on running averages of these statistics computed over the training process.