Recurrent Neural Network

Dr. Avinash Kumar Singh

AI Consultant and Coach, Robaita

Discussion Points

- Loss Functions
  - Binary Cross Entropy, Categorical Cross Entropy, Sparse Categorical Cross Entropy, Focal Loss, Triplet Loss
- Normalization Techniques
  - Batch and Layer Normalization
- Recurrent Neural Network
  - RNN, Long Short-Term Memory(LSTM), Bi-LSTM
- USE Cases
  - Class Imbalance
  - Batch and Layer Normalization
  - Text Generation

Loss Functions

- Binary Cross Entropy

Used for: Binary classification (2 classes — e.g., Apple vs. Orange)

True Label (y)	Predicted Probability (ŷ)
1	0.9
0	0.2

$$\mathrm{BCE} = -\left[y \cdot \log(\hat{y}) + (1-y) \cdot \log(1-\hat{y})\right]$$

✅ **Case 1: y = 1, ŷ = 0.9**

$$\mathrm{BCE} = -[1 \cdot \log(0.9) + 0 \cdot \log(0.1)] = -\log(0.9) \approx 0.105$$

❌ **Case 2: y = 0, ŷ = 0.2**

$$\mathrm{BCE} = -[0 \cdot \log(0.2) + 1 \cdot \log(1-0.2)] = -\log(0.8) \approx 0.223$$

```
model.compile(loss='binary_crossentropy', optimizer='adam')
```

Loss Functions

- Categorical Cross Entropy

Used for: Multi-class classification (labels are **one-hot encoded**)

🧠 Scenario: MNIST digit recognition (10 classes: 0–9)

Suppose the true label is digit `3` and it's one-hot encoded as:

Y	0	0	0	0	1	0	0	0	0	0
ŷ	0.01	0.05	0.02	0.80	0.03	0.02	0.01	0.03	0.02	0.01

$$\text{CCE} = -\sum_{i=1}^{C} y_i \cdot \log(\hat{y}_i)$$

Only the log probability of the true class (index 3) is used because the rest are multiplied by 0:

$$\text{CCE} = -\log(0.80) \approx 0.223$$

```
# Categorical crossentropy: true labels are one-hot encoded
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Loss Functions

- Sparse Categorical Cross Entropy

Used for: Multi-class classification, but true labels are **not one-hot encoded** — just a single integer (e.g., `3`)

🧠 **Scenario: MNIST digit recognition (10 classes: 0–9)**

Here the classes are represented in actual number

Y	3

ŷ	0.01	0.05	0.02	0.80	0.03	0.02	0.01	0.03	0.02	0.01

$$\text{SCCE} = -\log(\hat{y}_{\text{true class index}})$$

$$\text{SCCE} = -\log(0.80) \approx 0.223$$

```
# Sparse categorical crossentropy: true labels are integers
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
```

Loss Functions

- Focal Loss

💡 **Purpose:**

- Designed to **focus training on hard examples** and **down-weight easy ones**
- Useful in **imbalanced classification tasks** (e.g., rare disease detection, fraud detection)

$$\mathrm{FL}(p_t) = -\alpha_t \cdot (1 - p_t)^\gamma \cdot \boxed{\log(p_t)}$$

Where:

- $p_t = \hat{y}$ if label = 1, else $1 - \hat{y}$
- γ = focusing parameter (e.g., 2.0)
- α_t = class balancing factor

Parameter	Purpose
γ (gamma)	Focuses on **hard** misclassified cases
α (alpha)	Balances **minority/majority** classes

$$\boxed{\mathrm{BCE} = -\left[y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})\right]}$$

if y=1

if y=0

$$y \cdot \log(\hat{y}) \qquad (1 - y) \cdot \log(1 - \hat{y})$$

Let's Generalize the BCE

$$p_t = \begin{cases} \hat{y} & \text{if } y = 1 \\ 1 - \hat{y} & \text{if } y = 0 \end{cases}$$

$$\mathrm{BCE}(p_t) = -\log(p_t)$$

Loss Functions

- Focal Loss

1

Recap of the Focal Loss Formula (Binary Case):

$$\mathrm{FL}(p_t) = -\alpha_t \cdot (1 - p_t)^\gamma \cdot \log(p_t)$$

Where:

- $p_t = \hat{y}$ if $y = 1$, else $p_t = 1 - \hat{y}$
- $\alpha_t = \alpha$ if $y = 1$, else $1 - \alpha$
- $\gamma \geq 0$ is the **focusing parameter**

For Multiclass

$$\mathrm{FL}_i = -\alpha_i (1 - \hat{y}_i)^\gamma \cdot y_i \cdot \log(\hat{y}_i)$$

2

Scenario	True Label (y)	Prediction ($\hat{y}$)	p_t
Easy Example	1	0.95	0.95
Hard Example	1	0.2	0.2

3

Case 1: γ = 0 (Focal Loss behaves like BCE)

$$\mathrm{FL}(p_t) = -\log(p_t)$$

Example	p_t	FL (γ=0)
Easy (0.95)	0.95	0.051
Hard (0.2)	0.2	1.609

4

Case 2: γ = 2 (Focal Loss focuses on hard examples)

$$\mathrm{FL}(p_t) = -(1 - p_t)^2 \cdot \log(p_t)$$

Example	p_t	$(1 - p_t)^2$	FL (γ=2)
Easy (0.95)	0.95	0.0025	0.00013 ✅ Suppressed
Hard (0.2)	0.2	0.64	1.030 ✅ Still significant

5

- Minority class (fraud = 1): $\alpha = 0.25$
- Majority class (legit = 0): $1 - \alpha = 0.75$

Example 1: Fraud case (y = 1, α = 0.25), $\hat{y} = 0.2, p_t = 0.2, \gamma = 2$

$$\mathrm{FL} = -0.25 \cdot (1 - 0.2)^2 \cdot \log(0.2) = -0.25 \cdot 0.64 \cdot (-1.609) = 0.257$$

Example 2: Legit case (y = 0, α = 0.75), $\hat{y} = 0.2, p_t = 0.8$

$$\mathrm{FL} = -0.75 \cdot (1 - 0.8)^2 \cdot \log(0.8) = -0.75 \cdot 0.04 \cdot (-0.223) \approx 0.0067$$

```
tf.keras.losses.BinaryFocalCrossentropy
tf.keras.losses.CategoricalFocalCrossentropy
```

Loss Functions

- Triplet Loss

Triplet Loss helps a model learn embeddings such that:

  - Similar items are closer together in embedding space
  - Dissimilar items are farther apart

We want to map images of the same person to nearby points in embedding space and different persons to distant points.

$$\mathcal{L} = \max\left(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0\right)$$

- $f(x)$: the embedding (output) of sample x
- $\|\cdot\|$: L2 norm (Euclidean distance)
- α: margin (a small buffer, e.g., 0.2) that forces a **minimum distance gap**

Term	Description
Anchor (A)	A sample image (e.g., Person A)
Positive (P)	Same identity as anchor (Person A again)
Negative (N)	Different identity (Person B)

- **Anchor**: $f(A) = [1, 1, 1]$
- **Positive**: $f(P) = [1.2, 1.1, 0.9]$
- **Negative**: $f(N) = [3, 3, 3]$
- Margin $\alpha = 0.5$

Step 1: Compute distances

- $\|A - P\|^2 = (0.2^2 + 0.1^2 + 0.1^2) = 0.06$
- $\|A - N\|^2 = (2^2 + 2^2 + 2^2) = 12$

Step 2: Plug into formula

$$\mathcal{L} = \max(0.06 - 12 + 0.5, 0) = \max(-11.44, 0) = 0$$

Optimizers

1. Gradient Descent

$$\theta := \theta - \eta \cdot \nabla_\theta J(\theta)$$

- **Batch-based**: Uses full dataset to compute gradient.
- **Pros**: Stable convergence.
- **Cons**: Computationally expensive for large datasets.

➡️ **Motivation for SGD**: Reduce computation per step.

2. Stochastic Gradient Descent

$$\theta := \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$

- **Mini-batch variant** used in practice.
- **Pros**: Fast, memory efficient.
- **Cons**: Noisy updates, slower convergence, may get stuck in local minima.

➡️ **Motivation for Momentum**: Smooth out noisy updates.

3. Momentum

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta := \theta - v_t$$

- γ: momentum term (typically 0.9).
- **Pros**: Accelerates in right direction, reduces oscillations.
- **Cons**: May overshoot.

4. Adagrad

- g_t: Gradient at time step t
- G_t: Accumulated sum of squares of past gradients (per parameter)

$$g_t = \nabla_\theta J(\theta)$$
$$G_t = G_{t-1} + g_t^2$$
$$\theta := \theta - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

- **Per-parameter adaptive learning rates.**
- **Pros**: Good for sparse data (e.g., NLP).
- **Cons**: Learning rate shrinks too much over time.

➡️ **Motivation for RMSProp**: Fix Adagrad's decaying learning rate.

5. RMSProp

- $E[g^2]_t$: Exponential moving average of squared gradients
- β: Decay rate (typically 0.9)

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta) g_t^2$$
$$\theta := \theta - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$$

- **Exponential moving average** of squared gradients.
- **Pros**: Works well in RNNs, balances updates.
- **Cons**: No momentum term.

➡️ **Motivation for Adam**: Combine Momentum + RMSProp.

6. Adam (Adaptive Moment Estimation)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta := \theta - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

- m_t: First moment estimate (mean of gradients)
- v_t: Second moment estimate (variance of gradients)
- β_1: Decay rate for mean (typically 0.9)
- β_2: Decay rate for variance (typically 0.999)
- $\hat{m}_t, \hat{v}_t$: Bias-corrected moment estimates
- g_t: Current gradient

- Combines first moment (mean) and second moment (variance).
- **Pros**: Works well out of the box, fast convergence.
- **Cons**: Sometimes generalizes worse than SGD; more hyperparameters.

Optimizers

Optimizer	Key Features	Best For	TensorFlow API Name
Gradient Descent (GD)	Full batch update, simple but slow	Small datasets	"GradientDescent"
SGD	Per-sample or mini-batch, fast	General DL tasks	"SGD"
Momentum	Adds velocity term to SGD	ConvNets, image classification	"SGD(momentum=0.9)"
NAG	Lookahead version of momentum	Faster convergence in deep nets	"SGD(nesterov=True)"
Adagrad	Adaptive learning rate, good for sparse data	NLP, sparse inputs	"Adagrad"
RMSProp	Fixes Adagrad's decay issue	RNNs, time-series data	"RMSprop"
Adam	Combines momentum + adaptive LR	Most deep learning tasks	"Adam"
AdamW	Adam with decoupled weight decay	Transformers, pretraining tasks	"AdamW" (from tf.keras.optimizers.experimental)

Normalization

Covariate Shift

Occurs when the input data distribution changes between training and testing (or between different layers during training), but the function that maps inputs to outputs remains the same.

- External Covariate Shift: When training data and test data come from different distributions.
- Internal Covariate Shift: When the distribution of inputs to each layer changes during training, due to updates in the previous layers' weights.

Problem	Explanation
🔄 Slower Convergence	Each layer has to adapt continuously to the changing distribution of inputs.
📉 Gradient Instability	The gradients can become too small (vanishing) or too large (exploding), especially in deep networks.
❌ Poor Generalization	If test data has a different distribution than training data, the model may perform poorly.
🧠 Unstable Learning	The model may oscillate, diverge, or fail to learn altogether.

Normalization

▪ Batch Normalization

Normalization is a technique to standardize or scale the inputs to a neural network so that they have similar distributions.

To normalize the inputs of each layer in a neural network across the mini-batch. This helps reduce internal covariate shift (i.e., changes in the distribution of inputs to layers during training).

Example Neural Network

- **Batch size** = 32
- **Input** = 784
- **Hidden Layer 1** = 64 neurons
- **Hidden Layer 2** = 32 neurons
- **Output Layer** = 10 neurons

🔄 Step-by-Step Batch Normalization at Hidden Layer 1

🔹 Step 1: Compute Pre-activation Output

After linear transformation (dot product):

$$Z^{(1)} = XW^{(1)} + b^{(1)}$$

- X: shape = **(32, 784)**
- $W^{(1)}$: shape = **(784, 64)**
- $Z^{(1)}$: shape = **(32, 64)** → Each row is one sample, each column is a neuron's output before activation

Normalization

- Batch Normalization

Step-by-Step Batch Normalization at Hidden Layer 1

- **Step 2: Compute Mean per Feature (Neuron)**

$$\mu = \frac{1}{32}\sum_{i=1}^{32} Z^{(1)}[i,:]$$

- Mean μ: shape = **(64,)**
  (1 mean value per neuron, across the 32 samples)

- **Step 3: Compute Variance per Feature**

$$\sigma^2 = \frac{1}{32}\sum_{i=1}^{32}(Z^{(1)}[i,:]-\mu)^2$$

- Variance σ^2: shape = **(64,)**
  (1 variance value per neuron)

- **Step 4: Normalize Each Element**

$$\hat{Z}^{(1)}[i,j] = \frac{Z^{(1)}[i,j]-\mu[j]}{\sqrt{\sigma^2[j]+\epsilon}}$$

- Normalized Output $\hat{Z}^{(1)}$: shape = **(32, 64)**
  (same shape as original $Z^{(1)}$)

- **Step 5: Scale and Shift (Learnable Parameters)**

$$Y^{(1)}[i,j] = \gamma[j]\cdot\hat{Z}^{(1)}[i,j]+\beta[j]$$

- γ: shape = **(64,)**
- β: shape = **(64,)**
- Output after scale/shift $Y^{(1)}$: shape = **(32, 64)**

- **Step 6: Apply Activation Function (e.g., ReLU)**

$$A^{(1)} = \mathrm{ReLU}(Y^{(1)})$$

- Activated Output $A^{(1)}$: shape = **(32, 64)**

After batch normalization, each feature is forced to have:

- Mean ≈ 0
- Variance ≈ 1

This is good for training stability but can **limit the model's expressiveness**. Some features might naturally need higher variance or a shifted mean.

✅ Role of γ and β:

- **γ (gamma)**: scales the normalized output
- **β (beta)**: shifts the normalized output

Normalization

- Layer Normalization

Layer Normalization normalizes across the features of a single sample, instead of across the batch. So, instead of computing the mean and variance column-wise (per neuron across samples like in BatchNorm), it computes them row-wise (across all features in a layer for a single sample).

Step-by-Step: Layer Normalization at Hidden Layer 1

Step 1: Pre-activation Output

After linear transformation:

$$Z^{(1)} = XW^{(1)} + b^{(1)} \Rightarrow \text{shape: } (32, 64)$$

- Each row corresponds to 1 sample
- Each row has 64 neuron outputs (features)

Step 2: Compute Mean for Each Sample

For each row $i \in \{1, \ldots, 32\}$:

$$\mu^{(i)} = \frac{1}{64} \sum_{j=1}^{64} Z^{(1)}[i, j]$$

Mean μ: shape = **(32, 1)**

(1 mean per sample, shared across all 64 features)

Step 3: Compute Variance for Each Sample

$$\sigma^{2(i)} = \frac{1}{64} \sum_{j=1}^{64} (Z^{(1)}[i, j] - \mu^{(i)})^2$$

- Variance σ^2: shape = **(32, 1)**

Step 4: Normalize Features per Sample

$$\hat{Z}^{(1)}[i, j] = \frac{Z^{(1)}[i, j] - \mu^{(i)}}{\sqrt{\sigma^{2(i)} + \epsilon}}$$

Normalized Output $\hat{Z}^{(1)}$: shape = **(32, 64)**

(same shape as original pre-activation)

Step 5: Apply Scale and Shift (Per Feature)

$$Y^{(1)}[i, j] = \gamma_j \cdot \hat{Z}^{(1)}[i, j] + \beta_j$$

- γ: shape = **(64,)**
- β: shape = **(64,)**
- Output after scale/shift: shape = **(32, 64)**

Step 6: Apply Activation (e.g., ReLU)

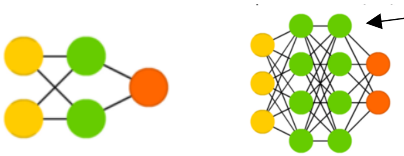
$$A^{(1)} = \mathrm{ReLU}(Y^{(1)})$$

- Activated Output: shape = **(32, 64)**

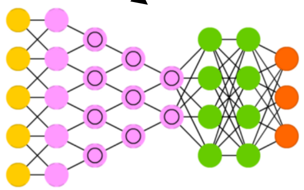
Recurrent Neural Network

Why Do We Need RNNs?

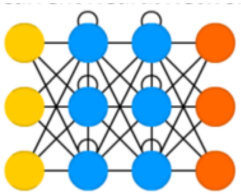
Neural Network



Multi Layer Perceptron
(Dense Network)



Convolutional Neural Network



Recurrent Neural Network

Every Sample is independent

Mostly One to One Association, for every input you need to provide an output

Helps to exploit the dependency between the samples

Good for sequence data for example sentence classification, sentence generation, music generation, action classification, time series forecasting, etc.

Recurrent Neural Network

Why Do We Need RNNs?

- Cannot capture temporal dependencies (e.g., sentence structure, time-series).
- Struggles with sequential patterns like speech, music, or stock prices.
- Requires fixed-size input/output — no flexibility for variable-length sequences.

The Need for RNNs:

- We need a model that can remember past inputs and learn from context.
- Ideal for tasks where sequence and timing matter — enter Recurrent Neural Networks (RNNs).

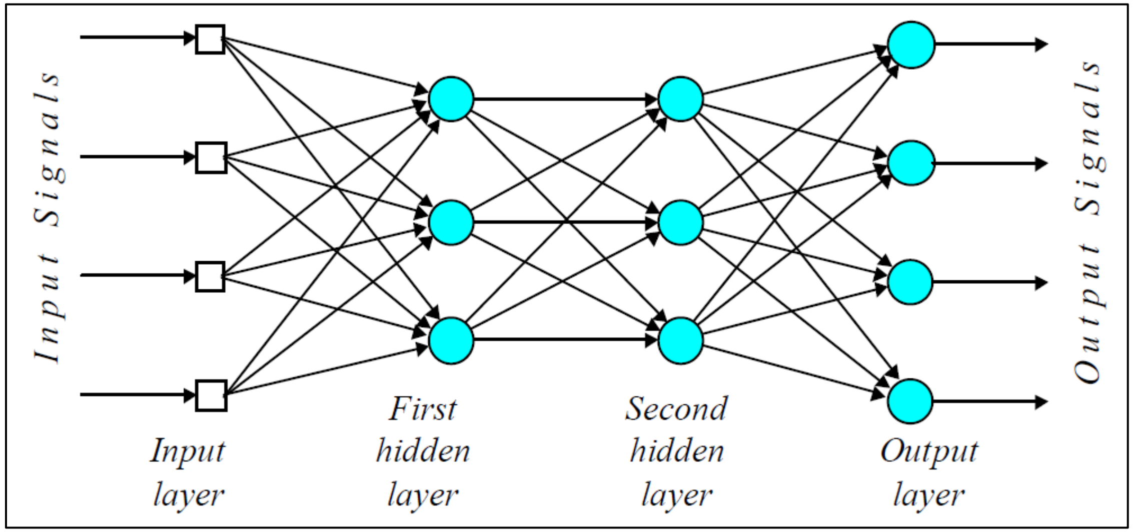
Recurrent Neural Network

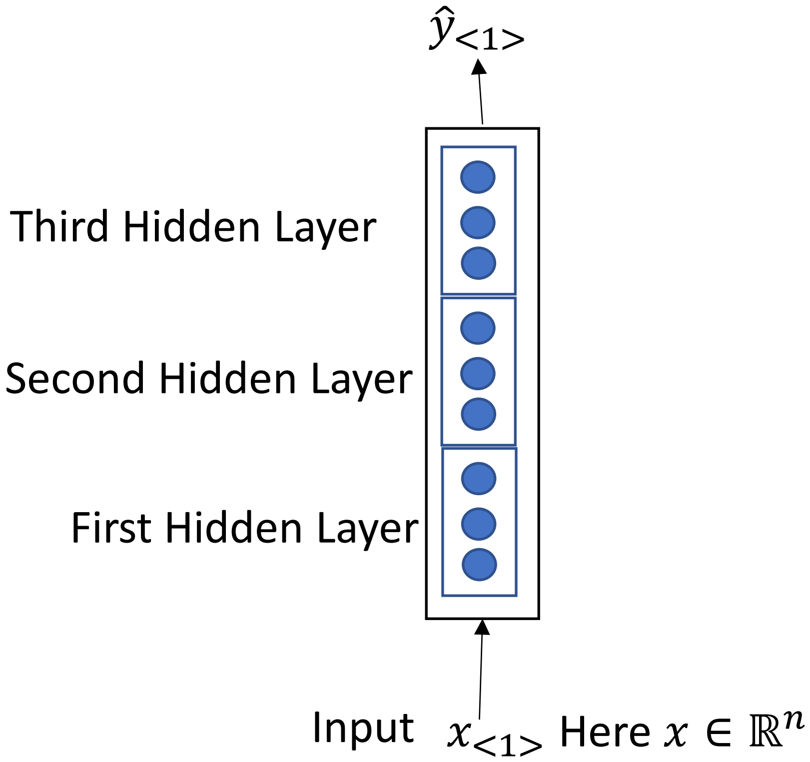
A Recurrent Neural Network (RNN) is a type of neural network where connections form a cycle, allowing information to persist across time steps.

- Takes input at each time step, maintains a hidden state (memory).
- Learns dependencies in ordered data.
- Shares parameters across time → efficient for sequence modeling.

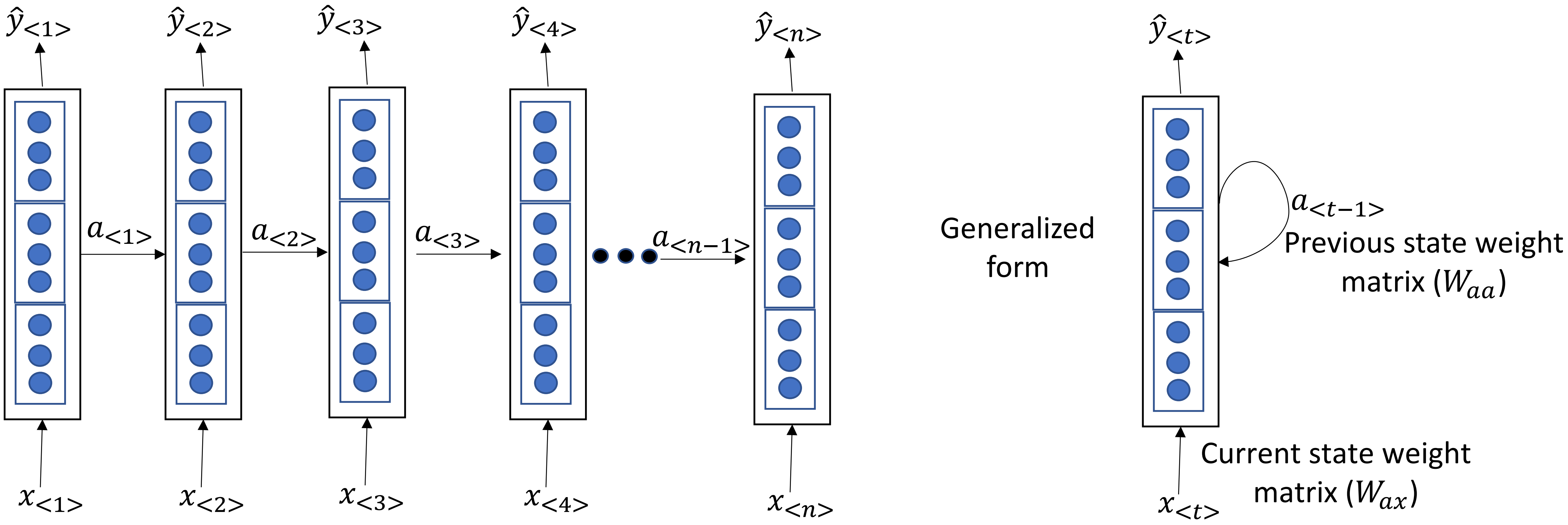
🔤 Language Modeling & Text Generation (e.g., GPT, LSTMs) 🗣️ Speech Recognition (e.g., Siri, Google Assistant) 📈 Time Series Forecasting (e.g., stock prediction, IoT sensors) 👁️ Video Frame Analysis (e.g., activity recognition) 💬 Chatbots & Machine Translation (sequence-to-sequence models)

Recurrent Neural Network

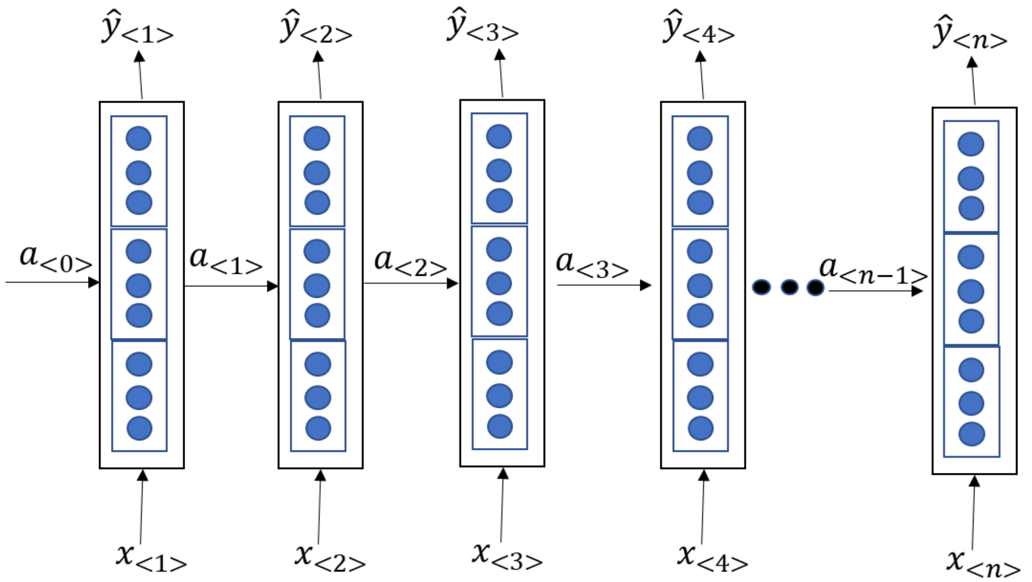




Recurrent Neural Network



Recurrent Neural Network



Initialization $a_{<0>} = 0$

$$a_{<1>} = f(W_{aa} * a_{<0>} + \boxed{W_{ax} * x_{<1>} + b_a})$$

Previous layer weight

Previous layer activation

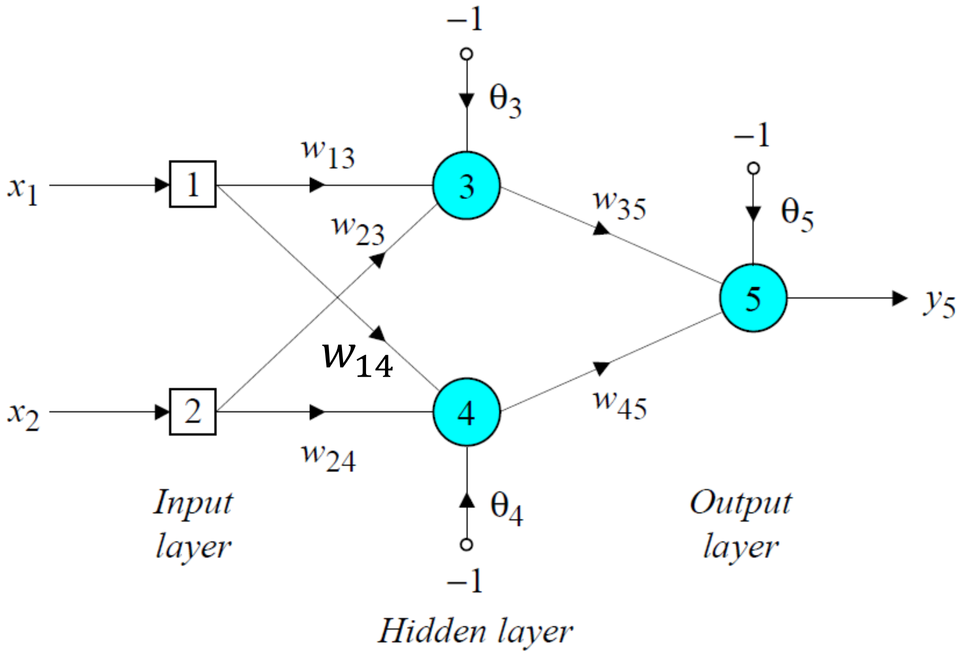
$$Y = f\{\sum_{i=1}^{n}(x_i * w_i) + b\}$$

$$\hat{y}_{<1>} = f(W_{ya} * a_{<1>} + b_y)$$

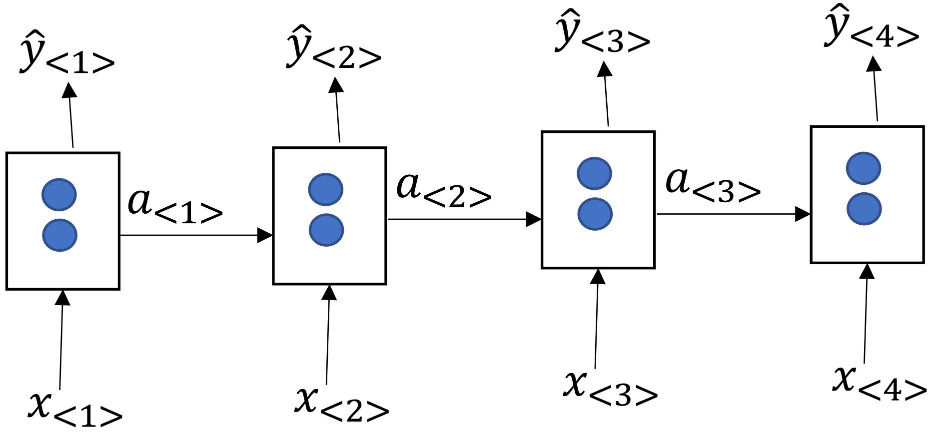
Robaita

Recurrent Neural Network: Example

x_1	x_2	w_{13}	w_{23}	θ_3	y_3	w_{14}	w_{24}	θ_4	y_4	w_{35}	w_{45}	θ_5	y_d	y_5	E	w_{13}'	w_{23}'	θ_4'	w_{14}'	w_{24}'	θ_4'	w_{35}'	w_{45}'	θ_5'
1	1	0.5	0.4	0.8		0.9	1.0	-0.1		-1.2	1.1	0.3	0											
0	1												1											
1	0												1											
0	0												0											

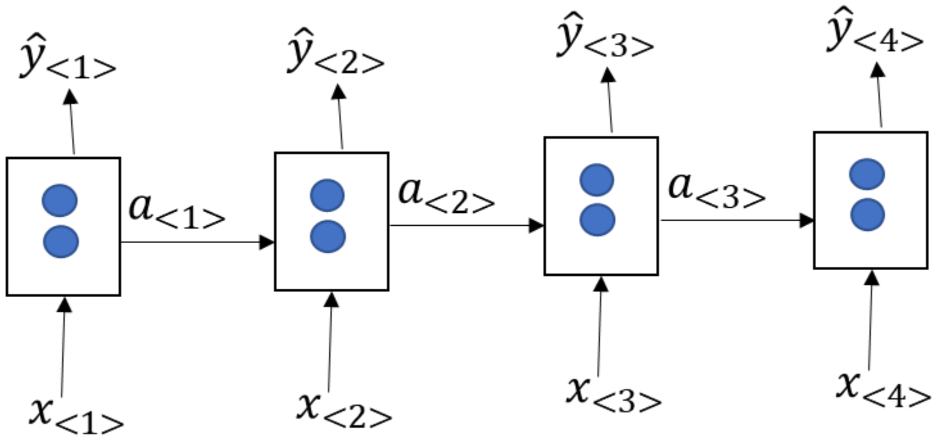


XOR problem, multi layer perceptron



XOR problem, with RNN

Recurrent Neural Network: Example



Here $x_{<1>} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

● represents y_3 and y_4

$\hat{y}_{<1>}$ represents y_5

$a_{<1>}$ represents $\begin{bmatrix} y_3 \\ y_4 \end{bmatrix}$

Let's keep the weights in a weight matrix,

$W_{ax} = \begin{bmatrix} w_{13} & w_{14} \\ w_{23} & w_{24} \end{bmatrix}$

So, the full equation become like

$$a_{<1>} = f(W_{aa} * a_{<0>} + W_{ax} * x_{<1>} + b_a)$$

Here, f is $sigmoid$ activation, $a_{<0>} = 0$

$$a_{<1>} = sigmoid(\begin{bmatrix} w_{13} & w_{14} \\ w_{23} & w_{24} \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \end{bmatrix})$$

Similarly, at the output layer

$$\hat{y}_{<1>} = f(W_{ya} * a_{<1>} + b_y)$$

$$\hat{y}_{<1>} = sigmoid(\begin{bmatrix} w_{35} \\ w_{45} \end{bmatrix} * a_{<1>} + (-1)$$

For the next time stamp

$$a_{<2>} = f(W_{aa} * a_{<1>} + W_{ax} * x_{<2>} + b_a)$$

$$a_{<2>} = sigmoid(W_{aa} * a_{<1>} + \begin{bmatrix} w_{13} & w_{14} \\ w_{23} & w_{24} \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \end{bmatrix})$$

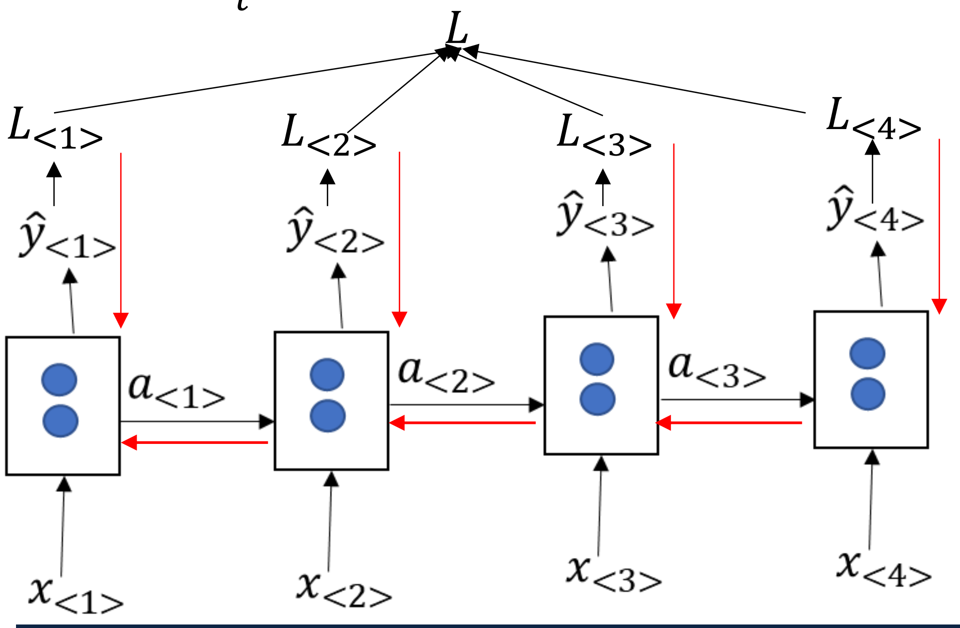
Backpropagation Through Time

The loss at the output layer is calculated by

$$L_{<t>}(\hat{y}_{<t>}, y_{<t>}) = -y_{<t>} log\hat{y}_{<t>} - (1 - y_{<t>})\log(1 - \hat{y}_{<t>})$$

Loss through the time stamp

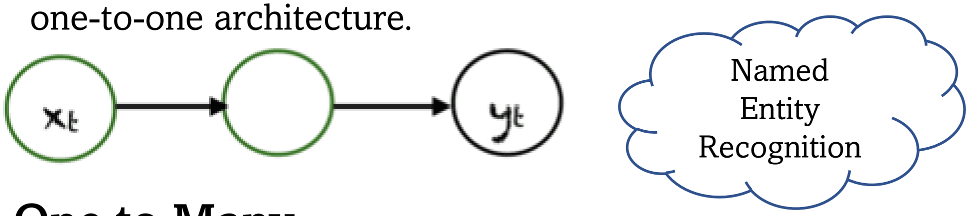
$$L(\hat{y}, y) = \sum_{t} L_{<t>}(\hat{y}_{<t>}, y_{<t>})$$



Types of RNN

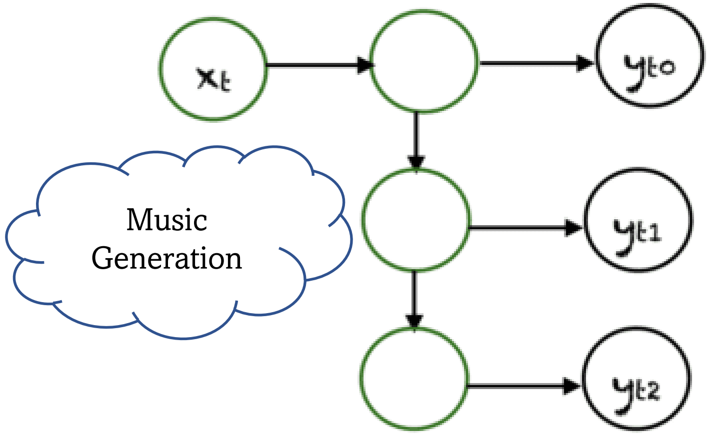
One to One

Here there is a single (x_t, y_t) pair. Traditional neural networks employ a one-to-one architecture.



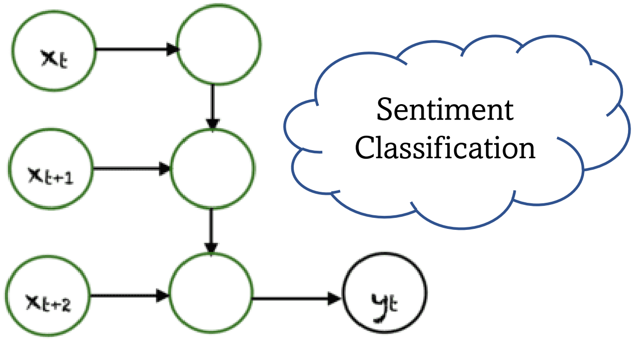
One to Many

In one-to-many networks, a single input at x_t can produce multiple outputs



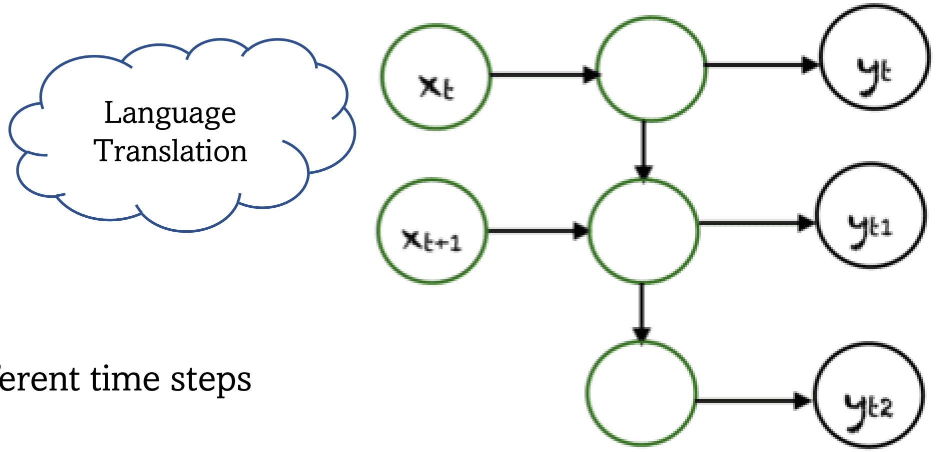
Many to One

In this case many inputs from different time steps produce a single output.



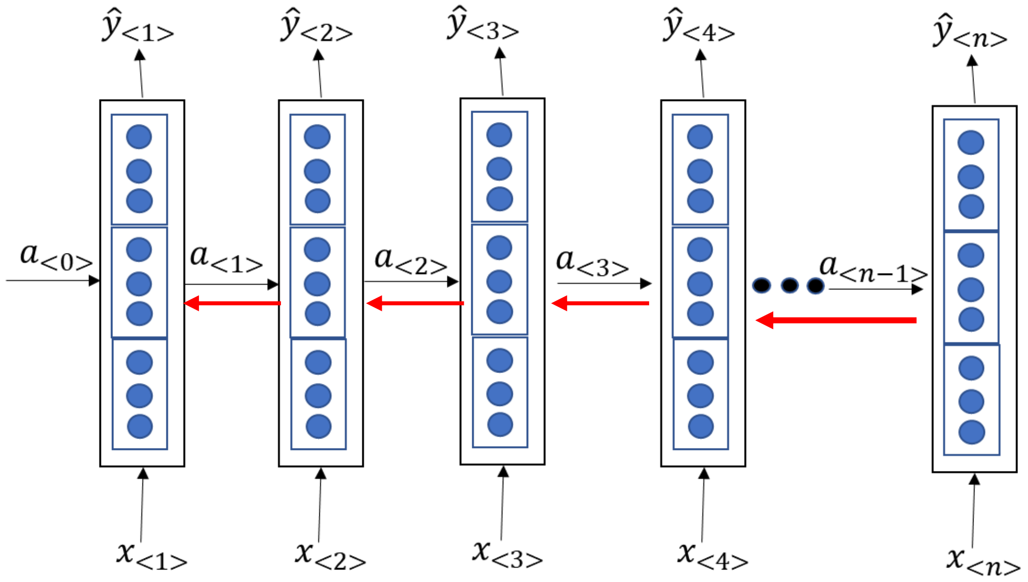
Many to Many

There are many possibilities for many to many. An example is shown above, where two inputs produce three outputs.



Vanishing Gradient: A problem in RNN

Due to the long-term dependencies the gradient at the initial layers become so small and it doesn't contribute to the significant weight change



$$a_{<1>} = f(W_{aa} * a_{<0>} + W_{ax} * x_{<1>} + b_a)$$

$$a_{<2>} = f(W_{aa} * a_{<1>} + W_{ax} * x_{<2>} + b_a)$$

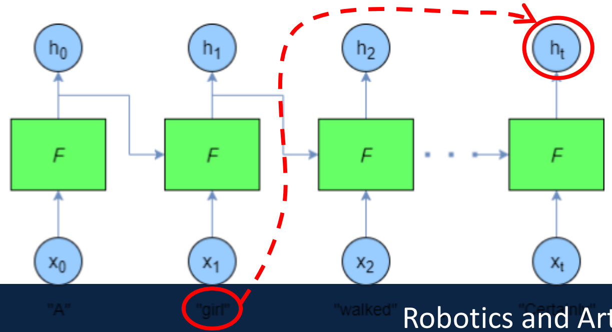
$$a_{<3>} = f(W_{aa} * a_{<2>} + W_{ax} * x_{<3>} + b_a)$$

Expanding the $a_{<3>}$

$$a_{<3>} = f(W_{aa} * (f(W_{aa} * a_{<1>} + W_{ax} * x_{<2>} + b_a)) + W_{ax} * x_{<3>} + b_a)$$

Reason: Due to long term dependencies and the deep networks

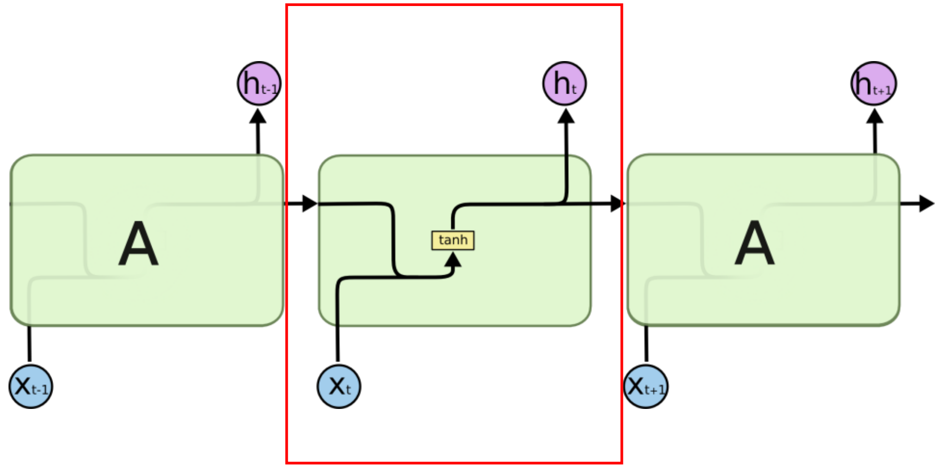
Sometimes, its difficult to memorize long dependencies

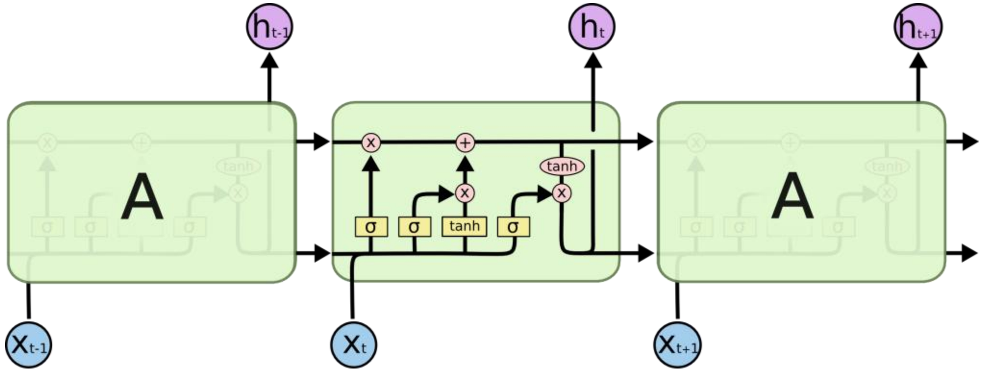


Wish could have some **"memory"** to memorize the dependencies

And some **control on information flowing**

Long Short-Term Memory (LSTM)





$$a_{<1>} = tanh(W_{aa} * a_{<0>} + W_{ax} * x_{<1>} + b_a)$$

- ❑Memory Cell
- ❑ Update gates
- ❑ Forget gates
- ❑ Output gates

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735-1780.

LSTM Components: Forget Gate

$$a_{<1>} = tanh(W_{aa} * a_{<0>} + W_{ax} * x_{<1>} + b_a)$$

Can be further simplified

$$a_{<1>} = tanh(W_a[a_{<0>}, x_{<1>}] + b_a)$$

$where$

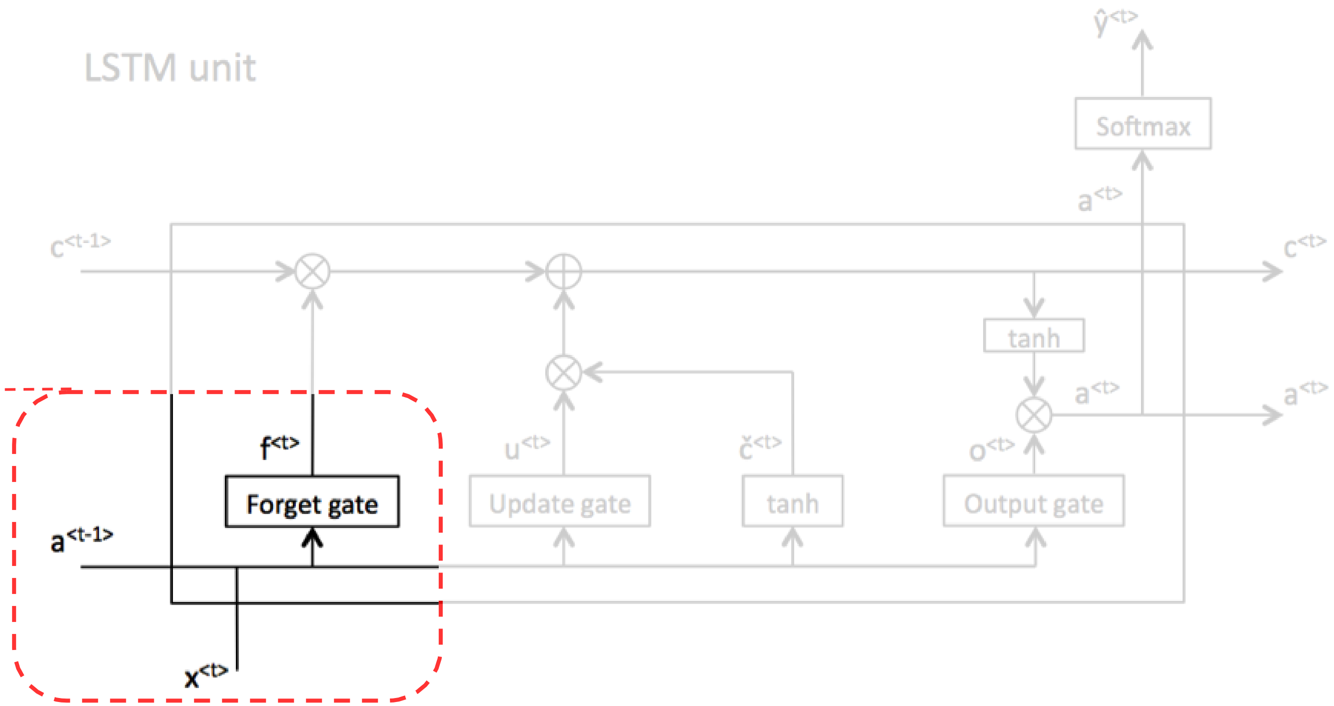
$$W_a = [W_{aa} | W_{ax}]$$

$Generalized\ form$

$$a_{<t>} = f(W_a[a_{<t-1>}, x_{<t>}] + b_a)$$

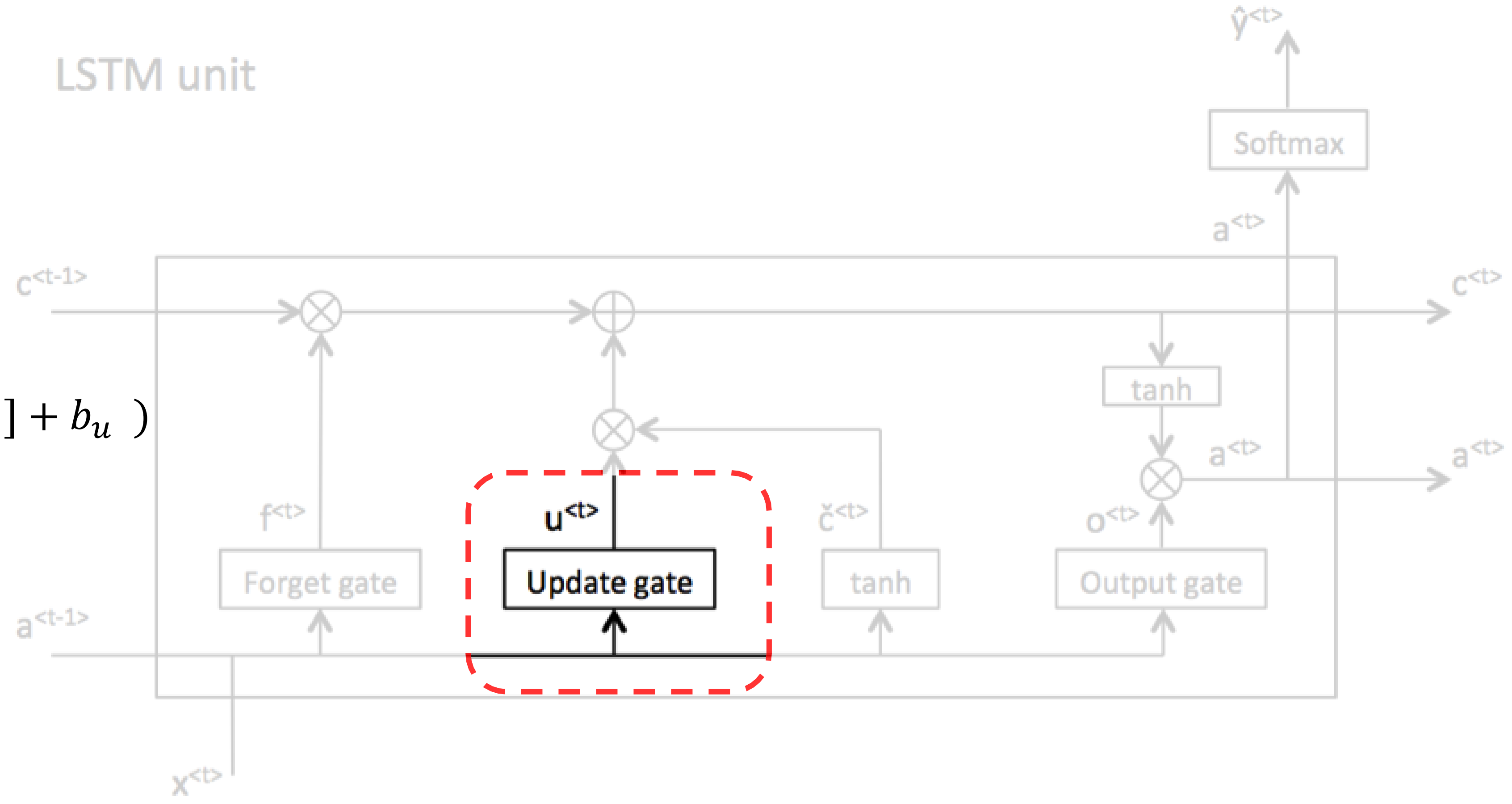
$Forget\ Gate$

$$f_{<t>} = \sigma(W_f[a_{<t-1>} + x_{<t>}] + b_f)$$



This will help the network learn which data can be forgotten and which data is important to keep

LSTM Components: Update Gate

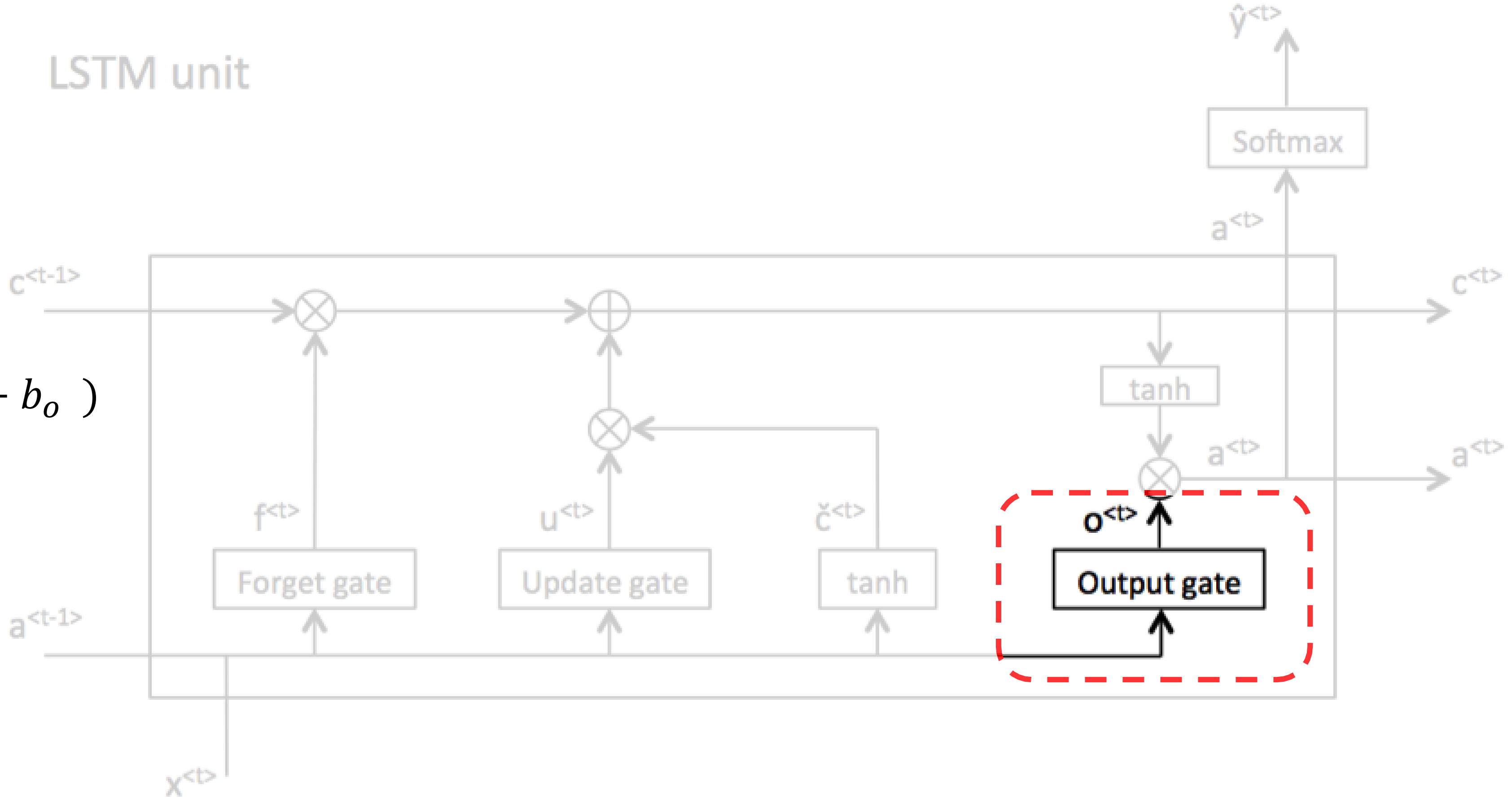


Update Gate

$$u_{<t>} = \sigma(W_u[a_{<t-1>} + x_{<t>}] + b_u)$$

Controls the updation of the memory cell. If output is 1 it would be update if 0 then not.

LSTM Components: Output Gate



Output Gate

$$o_{<t>} = \sigma(W_o[a_{<t-1>} + x_{<t>}] + b_o)$$

The output gate determines the value of the next hidden state. This state contains information on previous inputs.

LSTM Components: Other Operations

$Cell\ State/MemoryCell$

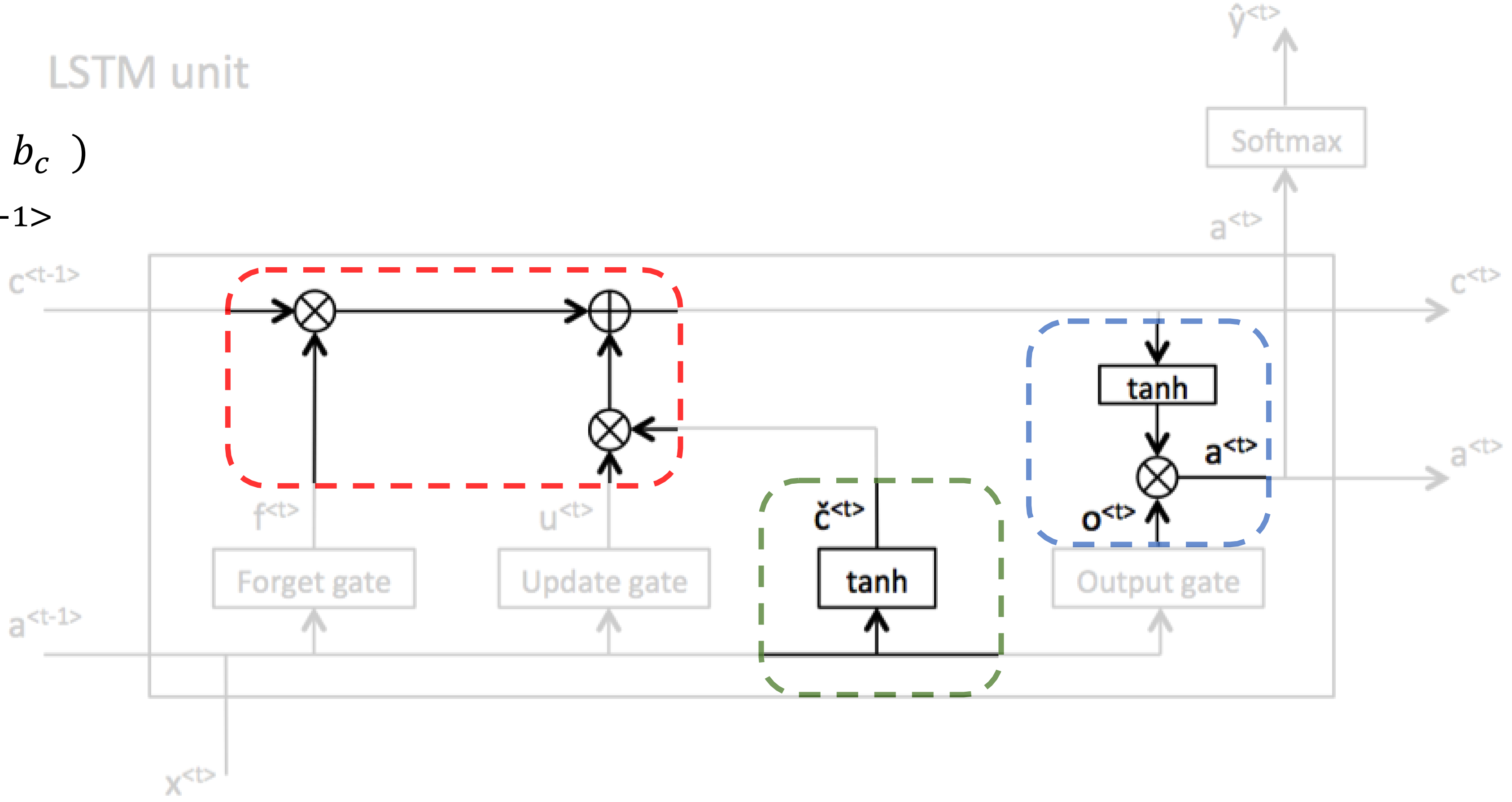
$$\breve{c}_{<t>} = tanh(W_c[a_{<t-1>} + \ x_{<t>}] + b_c \ )$$

$$c_{<t>} = u_{<t>} \ * \breve{c}_{<t>} + f_{<t>} \ * \breve{c}_{<t-1>}$$

The network has enough information from the **forget gate** and **update gate**. The next step is to decide and store the information from the new state in the cell state.

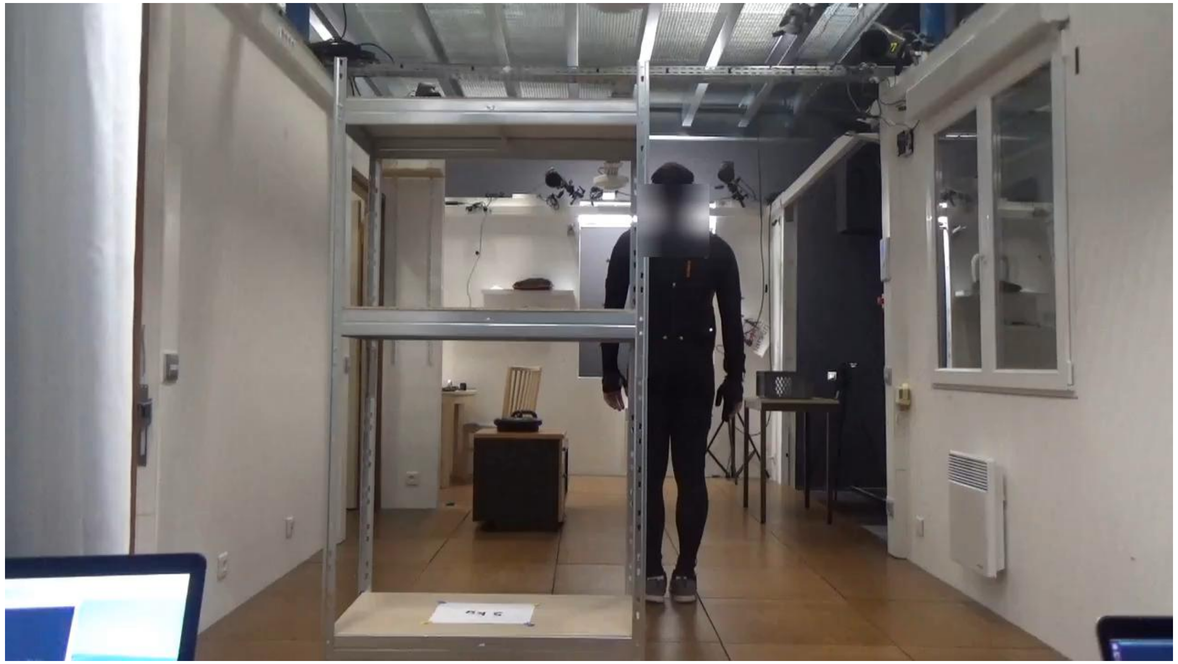
$activation\ value$

$$a_{<t>} = o_{<t>} \ * c_{<t-1>}$$



Tanh: To avoid information fading, a function is needed whose second derivative can survive for longer

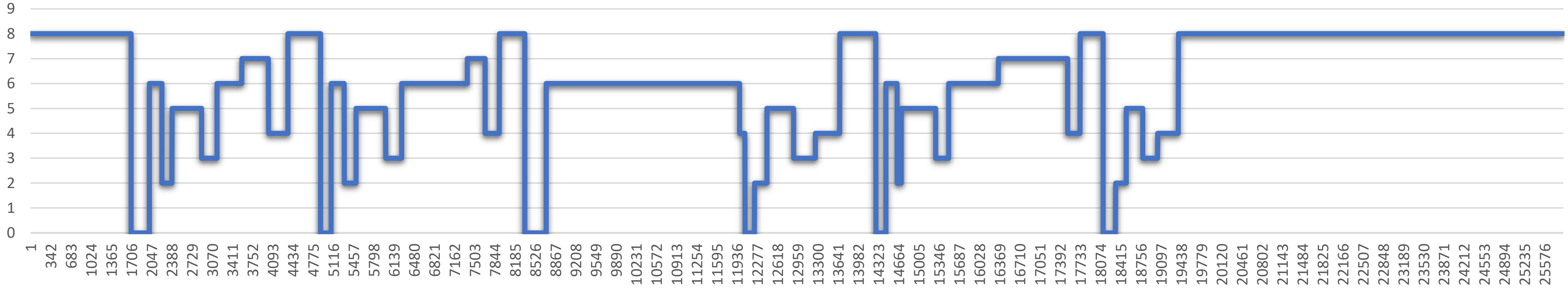
Case Study: Human Action Recognition



8 Action Classes

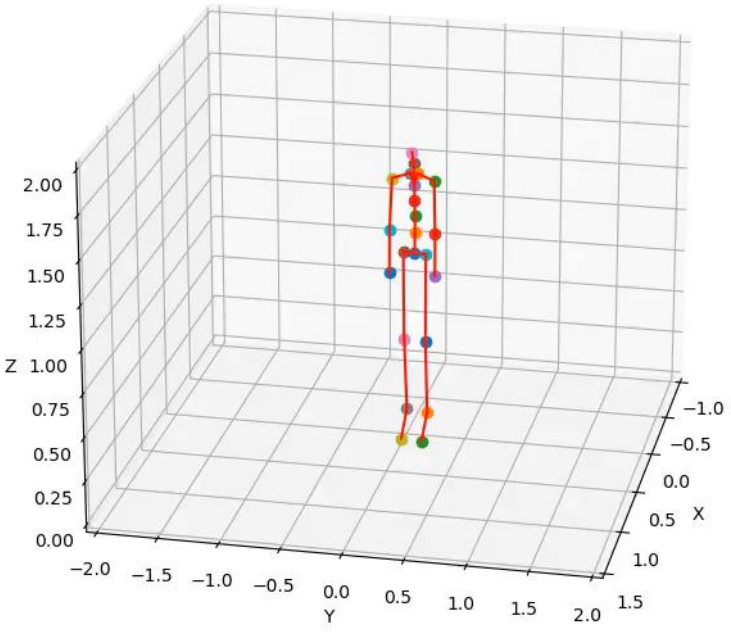
- Reach (1)
- Pick (2)
- Place (3)
- Release (4)
- Carry (5)
- Fine Manipulation (6)
- Screw (7)
- Idle (8)

Dataset is recorded with Xsens Sensor that has 23 marker position. Dataset is record at 240 FPS

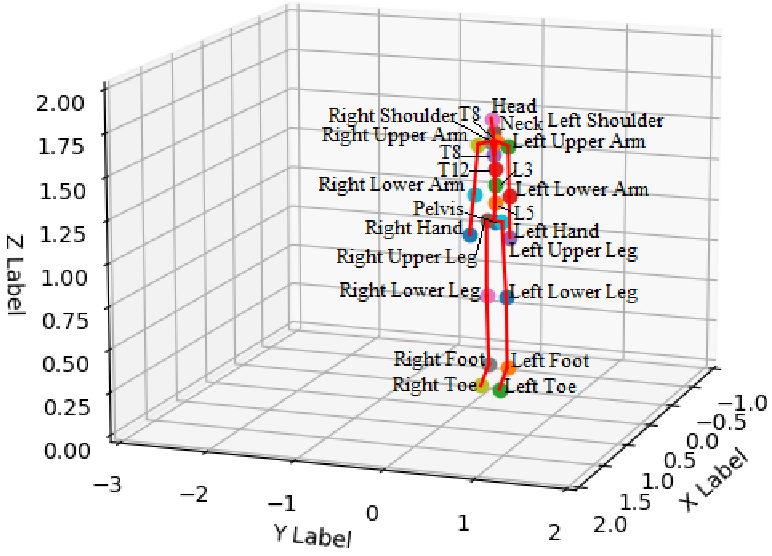


Case Study: Human Action Recognition

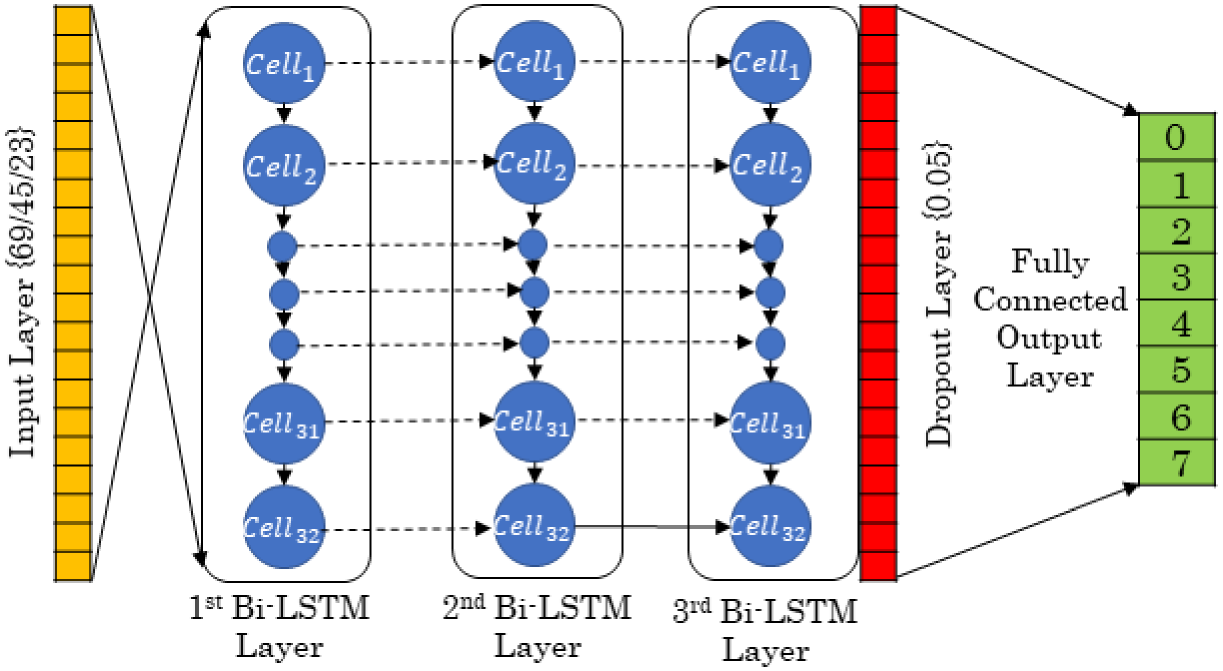
Joint Position Representation



Case Study: Human Action Recognition



Joint Representation

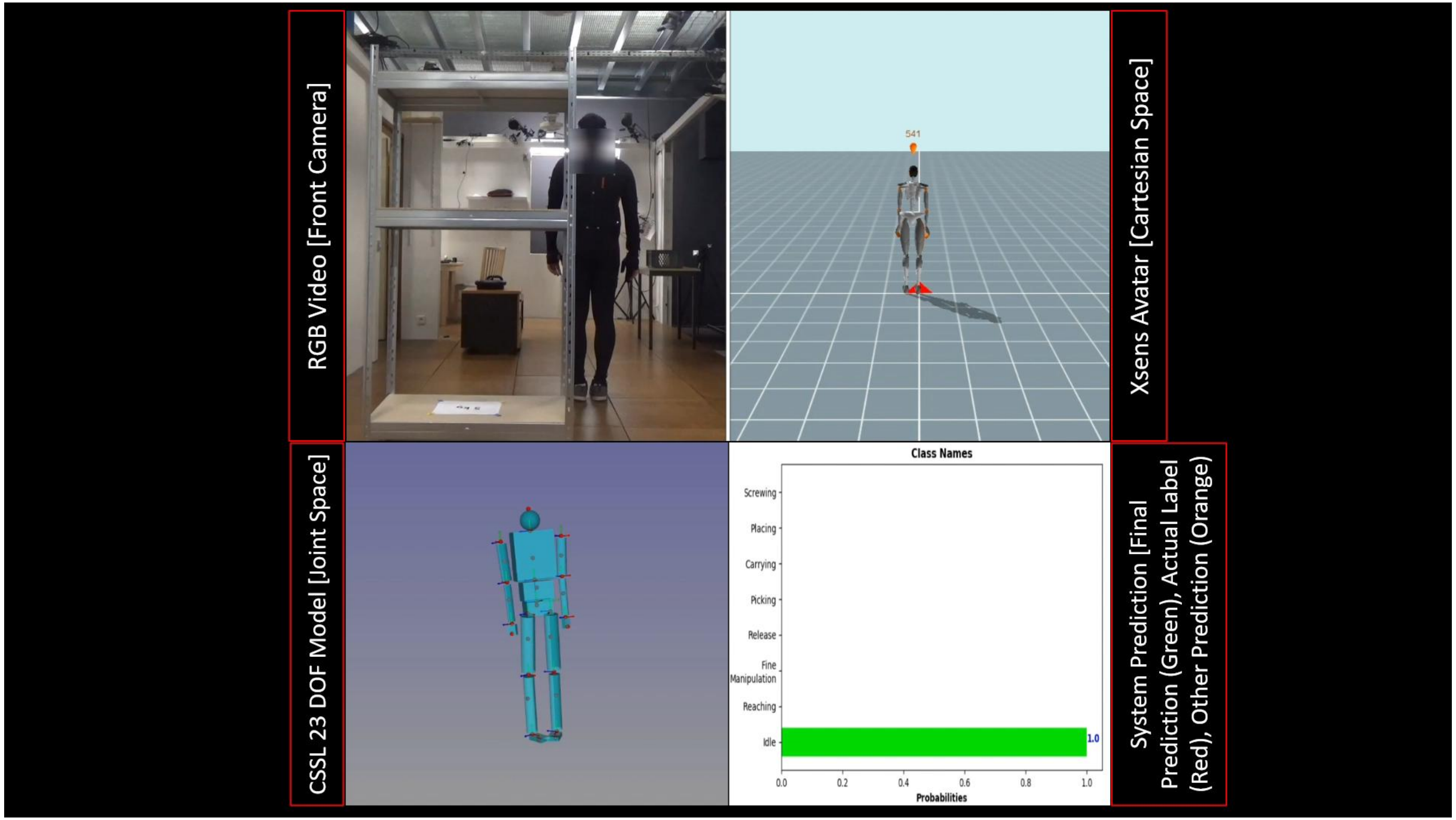


Neural Network Design

Time Window size: 240

```
n_timesteps, n_features, n_outputs = X_train.shape[1], X_train.shape[2], y_train.shape[1]
model = keras.Sequential()
model.add(layers.Bidirectional(layers.LSTM(32, return_sequences=True),
                                             input_shape=(n_timesteps,n_features)))
model.add(layers.Bidirectional(layers.LSTM(32, return_sequences=True)))
model.add(layers.Bidirectional(layers.LSTM(32)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(n_outputs, activation='softmax’))
model.(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Case Study: Human Action Recognition



Thanks for your time