Unit 4 : RNNs

Unfolding Computational graphs

1. Mapping inputs and parameters to outputs and losses is formalised using a computation graph.

state → previous input

Unfolding computational graphs in recurrent neural networks (RNNs) refers to the process of expanding the computation over multiple time steps. RNNs are designed to operate on sequences of data, such as time series or sequences of words in natural language processing tasks. Unfolding the computational graph allows us to visualize and understand how information flows through the network over time.

1. **Basic RNN Model:**
    - In a basic RNN model, the network consists of recurrent connections that allow information to persist over time. At each time step *t*, the network takes an input *xt* and produces an output *ht*.
    - The hidden state *ht* at time *t* is computed based on the input *xt* and the previous hidden state *ht*−1 using a set of trainable parameters (weights and biases) and an activation function.
2. **Unfolding Over Time:**
    - To unfold the computational graph, we repeat the computation for multiple time steps, typically equal to the length of the input sequence.
    - At each time step *t*, we apply the same set of parameters to compute the hidden state *ht* based on the input *xt* and the previous hidden state *ht*−1.
    - The computation is repeated recursively for each time step until the entire sequence is processed.

Advantages of Unfolding process

1. Regardless of sequence length, model has same input size. its specified in terms of transition from one state to another state than in terms of a variable length history of states.
2. can use the same function f with same parameters.

Recurrent Neural networks

RNNs are a type of neural network in which results of one step are fed into another steps computations

Traditional neural networks have inputs and outputs independent of one another. However, there is a need to remember the previous words in situations necessary to anticipate the next word in a sentence.

RNNs have a hidden state that retains some information about a sequence, and is the most significant characteristic of RNNs.

RNNs have “memory” that retains data, and it executes same action on all inputs and hidden layers to produce the output.

Working

Imagine you have a deep neural network with three hidden layers, one output layer, and one input layer. Each hidden layer has its own set of weights and biases, represented as $(w1,b1)$(w1,b1) for the first hidden layer, $(w2,b2)$(w2,b2) for the second hidden layer, and $(w3,b3)$(w3,b3) for the third hidden layer. In a traditional feedforward neural network, each hidden layer processes input independently, without considering the output of previous layers.

Now, let's introduce an RNN into the mix:

1. **Transforming Independent Activations into Dependent Activations:**
    - In an RNN, we give all layers the same weights and biases. This means that the weights and biases for each hidden layer are identical.
    - By doing this, the RNN transforms independent activations (output of one layer) into dependent activations, meaning each layer now takes into account the output of the previous layer.
2. **Combining Layers into a Recurrent Layer:**
    - Instead of having separate weights and biases for each hidden layer, we can combine all three layers into a single recurrent layer.
    - This recurrent layer uses the same weights and biases for all time steps, allowing it to remember information from previous time steps and incorporate it into the current computation.

💡 In simpler terms, think of it like this:

- In a regular neural network, each layer works independently, like individual workers in an assembly line. They don't communicate with each other directly.
- In an RNN, each layer is like a worker who not only performs their task but also talks to the previous worker to know what they've done. This allows the network to have memory and learn from previous inputs, making it more effective at tasks involving sequences, like language processing or time series prediction.

Training

1. **Input and Initial State**: At each time step, the network receives an input. This input, along with the previous state, is used to calculate the present state of the system. Initially, the initial state might be set to zeros or some other initialization strategy.
2. **Time Step Iteration**: The process described above is repeated for each time step. At each time step, the current input and the previous state are used to calculate the current state. Then, the current state becomes the previous state for the next time step.
3. **Memory Over Time**: One of the key advantages of recurrent neural networks (RNNs) is their ability to retain memory over time. This means that the network can capture temporal dependencies in the data by considering information from previous time steps.
4. **Error Calculation**: Once all the time steps have been processed and the final state is obtained, it is used to determine the output. This output is then compared to the desired or goal output (the actual output) to calculate the error.
5. **Backpropagation**: The error is then backpropagated through the network, starting from the output layer and moving backward through the network. This process updates the weights of the network in order to minimize the error, typically using optimization techniques like gradient descent.
6. **Training**: The network is trained by repeatedly presenting input sequences, calculating errors, and updating weights through backpropagation. This process continues until the network achieves satisfactory performance on the training data.

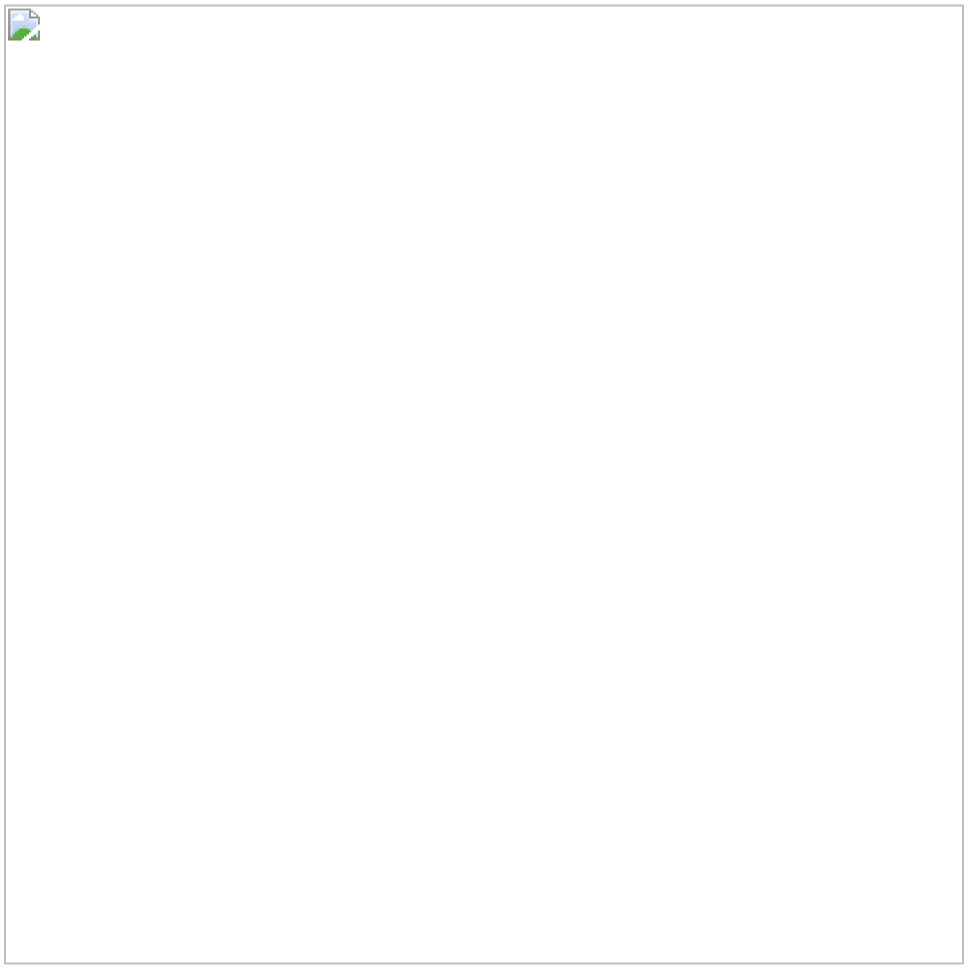
Advantages of RNNs

1. Can deal with consecutive data
2. Takes into account the current input

3. Can remember earlier inputs
4. RNN can handle sequential data, accepts both the input data being used and the inputs from the past.

Working of each recurrent unit

1. At each time step, the network takes in two vectors: the current input vector and the previous hidden state vector. Each vector is represented as a sequence of numbers, where each number corresponds to a specific feature or dimension of the input or hidden state.
2. The hidden state vector is multiplied element-wise by a set of weights called the hidden state weights. Similarly, the current input vector is multiplied element-wise by another set of weights called the current input weights. This process produces two new vectors: the parameterized hidden state vector and the parameterized current input vector. The purpose of this multiplication is to introduce flexibility and learnable parameters into the model.
3. The next step involves adding the two parameterized vectors together. This results in a combined vector that captures the influence of both the current input and the previous hidden state, weighted by the respective weights. This combined vector represents the updated hidden state.
4. Once the combined vector is obtained, an element-wise activation function, typically the hyperbolic tangent (tanh) function, is applied to each element of the vector. This ensures that the values of the hidden state remain within a certain range and introduces non-linearity into the model. The hyperbolic tangent function squashes the values of the vector to the range [-1, 1], which can help stabilize the learning process and prevent exploding gradients.



Backpropogation can result in a few issues

1. Vanishing gradients → gradients being so small lead to 0
2. Exploding gradients → Gradients grow excessively large

To adderss these issues, Gated RNNs and LSTMs were created to address these issues.

Disadvantages of RNNs

1. Tough to train
2. Gradients vanishing and exploding
3. TanH and ReLU cannot process very long sequences

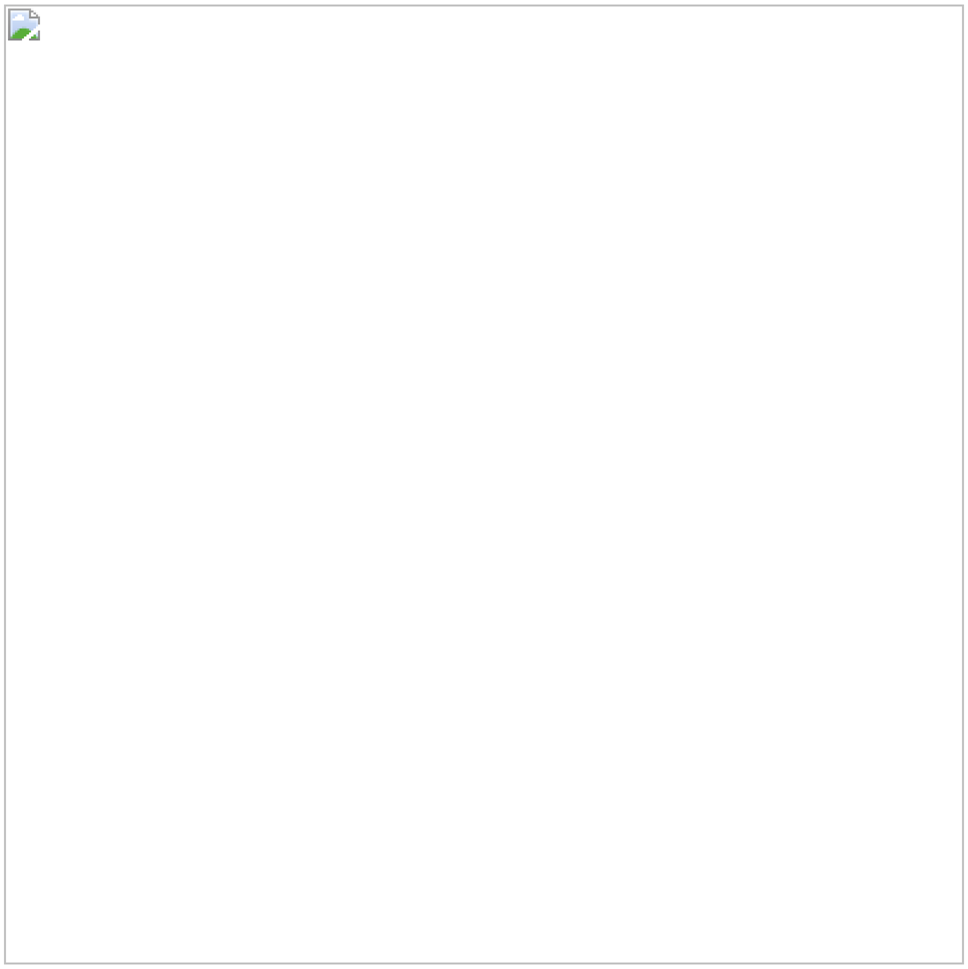
Bidirectional RNNs

1. Its basically 2 separate RNNs combined together.
2. For 1 input sequence, its fed in regular time order, and for a different network, its fed in reverse time order.
3. The outputs are typically concatenated.
4. This makes the RNNs access both forward and backward information about the sequence at each time step.

Encoder Decoder sequence to sequence architectures

1. A typical neural machine translation technique that competes with and occasionally exceeds traditional machine translation techniques is called encoder decoder architecture for rnns.
2. Encoder decoder model is composed of 3 blocks:

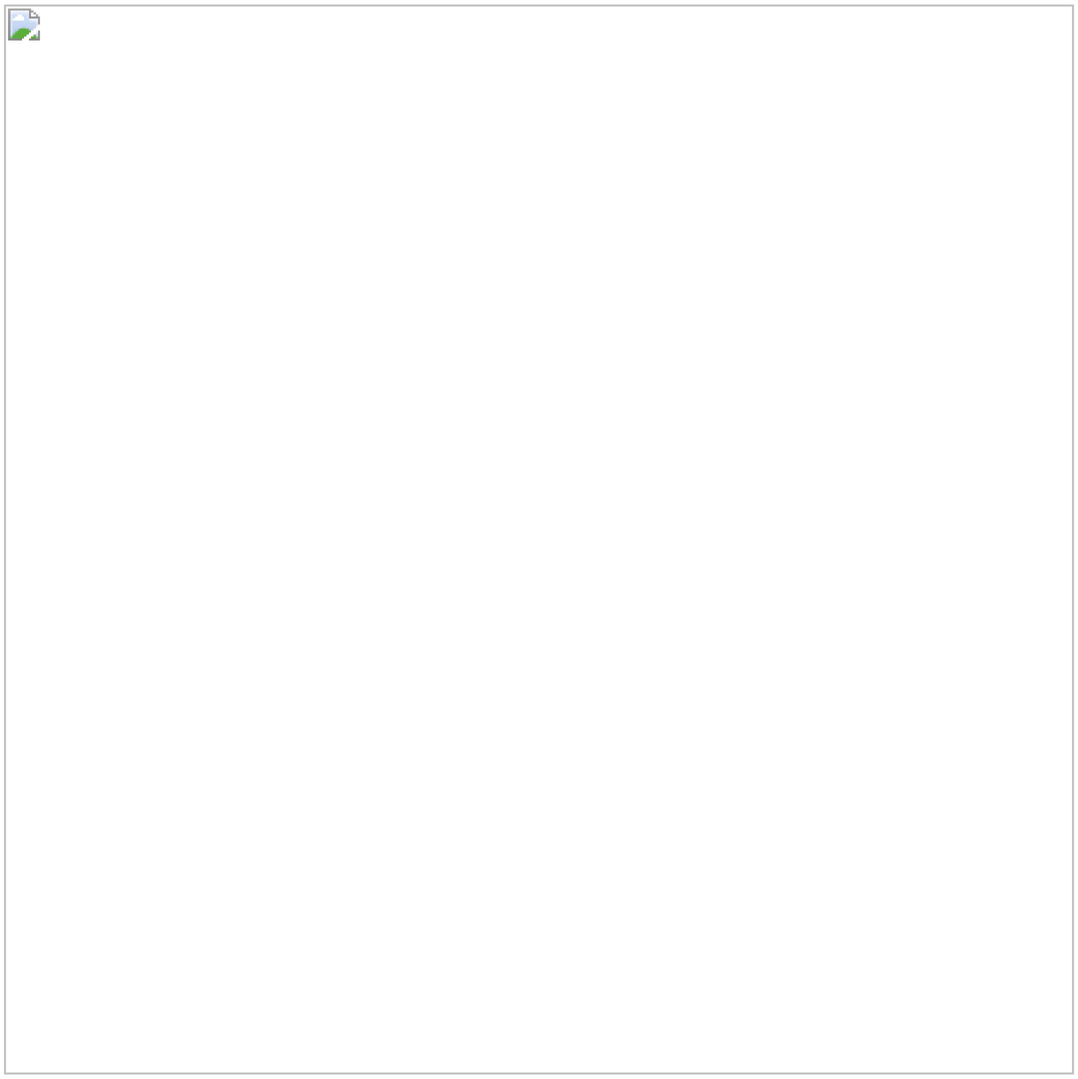
   a. Encoder
   b. Hidden vector
   c. Decoder
3. Encoder → creates a 1 D vector from the input sequence. The hidden cector will be transformed into a output sequence by the decoder.
4. They are jointly trained.



Encoder

1. Encoder is created by stacking many RNN cells. RNN scans every input.
2. Each hidden state is update in accordance with the input at that timestep.

3. The final hidden state of the model represents the summary of the whole input sequence after the encoder has read all the inputs.
4. Preceding hidden state h0 → 0
5. Using initial input and h0, RNN cell updates the current hidden state, and the updated hidden state and the output for each level are the 2 outputs each layer produces.



6. A group of recurrent units are stacked on top of the other, each recieves a single input sequence and gathers data for that element.

Encoder Vector

1. Models final hidden state is created by the encoder, and in order to aid the decode in producing precise predictions, vector seeks to include the data for all inputs.

Decoder

1. **Decoder Role**: In the context of a sequence-to-sequence model, the decoder is responsible for generating an output sequence based on the information encoded by the encoder. It takes the final hidden state obtained from the encoder as its initial input.
2. **Output Sequence Generation**: The decoder anticipates the subsequent output Y_t (symbolized as y_t) given the hidden state h_t. It iteratively generates the output sequence, typically one token (word or symbol) at a time, based on the current hidden state and previous outputs.
3. **Layer Inputs and Outputs**: Each layer of the decoder takes three inputs:
    - The original hidden vector from the encoder h,
    -
    - The hidden vector from the preceding layer (h_{t-1}),
    - The previous output (y_{t-1}).
      These inputs are used to compute the updated hidden state (h_t) and the output (y_t) for the current time step.
4. **Initialization**: At the beginning of decoding, the first layer's inputs consist of the encoder's output vector and a special symbol (often denoted as START), indicating the start of the output sequence. The outputs of this layer are the updated hidden state (h_1) and an empty hidden state (h_0), which represents the absence of any meaningful information from the previous time step.
5. **Layer-by-Layer Generation**: Subsequent layers of the decoder generate the hidden state (h_t) and output (y_t) for each time step based on the updated hidden state from the previous layer, the previous output, and the original hidden vector from the encoder.

6. **Output Forecasting**: Each recurrent unit within the decoder forecasts an output ($ytyt$) at each time step. The output sequence is constructed by concatenating these individual outputs together until an end-of-sequence symbol (often denoted as END) is encountered, indicating the completion of the output sequence.

At the output layer, softmax activation function is added. And the probability distribution from a vector of values is generated.

Applications of Encoder decoder RNN models

1. Googles machine translation
2. Chatbots
3. Speech recognition
4. Time series application