INFO 7375 - Neural Networks & AI

Home Work to Chapter - 19

Submitted By:

Abdul Haseeb Khan
NUID: 002844724
khan.abdulh@northeastern.edu

What are recurrent neural networks (RNN) and why are they needed?

Recurrent Neural Networks (RNNs) are a class of neural networks designed specifically to process sequential data by maintaining an internal state or "memory" that captures information from previous time steps. Unlike feedforward networks that process each input independently, RNNs contain loops that allow information to persist across sequence elements. The fundamental innovation is the introduction of recurrent connections where the network's hidden state at time t depends not only on the current input x_t but also on the previous hidden state h_{t-1}, formally expressed as $h_t = f(W_h h * h_{t-1} + W_x h * x_t + b_h)$, where W_{hh} represents the recurrent weight matrix, W_{xh} the input weight matrix, and b_h the bias term.

The necessity for RNNs arises from the inherent limitations of traditional feedforward networks when dealing with sequential data. Standard neural networks assume fixed-size inputs and outputs with no temporal dependencies between data points, making them unsuitable for tasks where context from previous inputs influences current predictions. Natural language processing exemplifies this need perfectly understanding a word's meaning often requires knowing the words that came before it. Similarly, time series prediction, speech recognition, video analysis, and music generation all involve patterns that unfold over time where the order and temporal relationships between elements are crucial. RNNs address this by sharing parameters across time steps and maintaining a hidden state that theoretically can capture arbitrary long-range dependencies, though in practice they face challenges with very long sequences.

What do time steps play in recurrent neural networks?

Time steps serve as the fundamental organizing principle in RNNs, representing discrete points in the sequence where the network processes input and updates its internal state. Each time step t corresponds to one element in the input sequence, whether that's a word in a sentence, a frame in a video, or a measurement in a time series. The network unrolls through time by repeatedly applying the same transformation at each step, creating what's essentially a very deep network when viewed across the temporal dimension. At each time step, the network performs three key operations: it receives the current input x_t, combines it with the previous hidden state h_{t-1} using learned weight matrices, and produces both a new hidden state h_t and potentially an output y_t.

The significance of time steps extends beyond mere sequencing, they define the computational graph's structure during both forward propagation and backpropagation. During training, the network is "unrolled" for a specific number of time steps, creating distinct nodes for each temporal position while sharing weights across all steps. This parameter sharing is crucial for generalization, allowing the network to process sequences of varying lengths with the same set of weights. The number of time steps

processed can be fixed (as in truncated backpropagation through time) or variable (processing entire sequences), with implications for both computational efficiency and the network's ability to capture long-range dependencies.

What are the types of recurrent neural networks?

RNN architectures can be categorized based on their input-output relationships and structural complexity. The most basic distinction involves the mapping between sequence inputs and outputs. One-to-many architectures take a single input and generate a sequence output, commonly used in image captioning where a single image produces a sentence description. Many-to-one architectures process an entire sequence to produce a single output, typical in sentiment analysis where an entire text sequence maps to a sentiment score. Many-to-many architectures come in two variants: synchronized, where input and output sequences have equal length (like part-of-speech tagging), and unsynchronized, where lengths differ (like machine translation). The encoder-decoder architecture represents a sophisticated many-to-many approach where an encoder RNN processes the input sequence into a fixed-size context vector, which a decoder RNN then uses to generate the output sequence. Beyond these input-output patterns, RNNs differ in their internal mechanisms for managing information flow. Vanilla RNNs use simple tanh or ReLU activations but suffer from vanishing and exploding gradients during backpropagation through long sequences. Long Short-Term Memory (LSTM) networks address this through a complex gating mechanism involving forget gates ($f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)$), input gates ($i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$), and output gates ($o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$), along with a cell state that provides a gradient highway through time. Gated Recurrent Units (GRUs) simplify LSTMs by combining the forget and input gates into a single update gate ($z_t = \sigma(W_z * [h_{t-1}, x_t])$) and merging the cell state with the hidden state, reducing parameters while maintaining comparable performance. Each architecture represents different trade-offs between computational complexity, parameter efficiency, and the ability to capture temporal dependencies.

What is the loss function for RNN defined?

The loss function for RNNs depends on the task type but generally involves aggregating losses across all relevant time steps. For sequence classification tasks using many-to-one architectures, the loss is typically computed only at the final time step T, such as L = -log(p(y|h_T)) for cross-entropy loss in classification. For sequence-to-sequence tasks, the loss accumulates across all output time steps: $L = \sum_{t=1}^{T} L_t(y_t, \hat{y}_t)$, where L_t represents the loss at time step t between the true output y_t and predicted output $\hat{y}_t$. In language modeling, this often takes the form of negative log-likelihood: $L = -\sum_{t=1}^{T} logp(x_t | x_1, \ldots, x_{t-1})$, measuring how well the model predicts the next token given previous context.

The choice of loss function must account for the sequential nature of RNN outputs and potential imbalances in sequence lengths. For variable-length sequences, the loss is often normalized by sequence length to prevent longer sequences from dominating gradient updates. In sequence generation tasks, teacher forcing is commonly employed during training, where the true previous outputs are fed as inputs rather than the model's own predictions, though this can create exposure bias. Some applications use specialized losses like the Connectionist Temporal Classification (CTC) loss for tasks where alignment between input and output sequences is unknown, such as speech recognition. The loss function ultimately drives the backpropagation through time algorithm, determining how gradients flow backward through the unrolled network to update all shared parameters.

How do forward and backpropagation of RNN work?

Forward propagation in RNNs proceeds sequentially through time, starting with an initial hidden state h_0 (often initialized to zeros) and processing each input in order. At time step t, the network computes the hidden state as $h_t = tanh(W_h h * h_{t-1} + W_x h * x_t + b_h)$, where the tanh activation introduces non-linearity. If the architecture produces outputs at each time step, these are computed as $y_t = W_h y * h_t + b_y$, potentially followed by a softmax for classification tasks. This process continues for all T time steps in the sequence, with each hidden state depending on the entire history of inputs up to that point through the recurrent connections. The computational graph maintains all intermediate activations needed for the subsequent backward pass.

Backpropagation through time (BPTT) unrolls the recurrent network and applies standard backpropagation to this unrolled graph, treating it as a very deep feedforward network with shared weights. The gradient of the loss with respect to the hidden state at time t receives contributions from both the loss at time t (if there's an output) and the gradient backpropagated from time. The gradient with respect to the recurrent weights accumulates contributions from all time steps: $\partial L / \partial W_h h = \Sigma_{t=1}^{T} \partial L / \partial h_t * \partial h_t / \partial W_h h$. This accumulation across many time steps leads to the notorious vanishing/exploding gradient problem, as gradients must flow through many matrix multiplications. The gradient $\partial h_t / \partial h_k$ for t > k involves the product $\Pi_{i=k+1}^{t} \partial h_i / \partial h_{i-1}$, and if the eigenvalues of $W_h h$ have magnitude less than 1, this product vanishes exponentially; if greater than 1, it explodes. Practical implementations often use gradient clipping to prevent explosion and architectural innovations like LSTMs to combat vanishing.

What are the most common activation functions for RNN?

The hyperbolic tangent (tanh) function is the most common and the sigmoid activation is exclusively used for gates (forget, input, and output gates) because its (0, 1) output range naturally represents gating decisions. GRUs similarly use sigmoid for gates and tanh for candidate hidden states.

Describe bidirectional recurrent neural networks (BRRN) and explain why they are needed.

Bidirectional Recurrent Neural Networks (BRNNs) represent a fundamental architectural innovation that addresses the inherent limitation of standard RNNs' unidirectional information flow by processing sequences simultaneously in both temporal directions. The architecture maintains two independent recurrent layers that traverse the input sequence in opposite directions, with the forward layer processing inputs from x_1 to x_T and the backward layer processing from x_T to x_1. Mathematically, the forward pass computes $h_t^{\rightarrow} = f(W_{xh}^{\rightarrow} * x_t + W_{hh}^{\rightarrow} * h_{t-1}^{\rightarrow} + b_h^{\rightarrow})$ while the backward pass computes $h_t^{\leftarrow} = f(W_{xh}^{\leftarrow} * x_t + W_{hh}^{\leftarrow} * h_{t+1}^{\leftarrow} + b_h^{\leftarrow})$, where the superscript arrows indicate directionality and each direction maintains completely separate weight matrices and biases. The crucial insight is that these two hidden states are computed independently during their respective passes, avoiding any circular dependencies that would make training impossible.

The output at each time step combines information from both directional states, typically through concatenation $y_t = g(W_y^{\rightarrow} * h_t^{\rightarrow} + W_y^{\leftarrow} * h_t^{\leftarrow} + b_y)$or sometimes through summation or more complex combination mechanisms. This dual representation means that at every position t, the network has encoded the entire past context (x_1, ..., x_t) in $h_t^{\rightarrow}$ and the entire future context (x_t, ..., x_T) in $h_t^{\leftarrow}$, providing a complete sequential picture. The gradient flow during backpropagation also benefits from this bidirectional structure, as gradients can flow through two independent paths, potentially alleviating some vanishing gradient issues that plague deep temporal dependencies in unidirectional architectures.

Describe Deep recurrent neural networks (DRRN) and explain why they are needed.

Deep RNNs (DRNNs) introduce multiple layers of recurrent processing, stacking RNN layers vertically to create hierarchical representations of sequential data. In a DRNN with L layers, each layer l maintains its own hidden state h_t^l that depends on both the previous time step at the same layer and the current time step from the layer below: $h_t^l = f(W_{hh}^l * h_{t-1}^l + W_{hh}^{l-1,l} * h_t^{l-1} + b_h^l)$, where $h_t^0 = x_t$ represents the input. This architecture allows different layers to capture patterns at different levels of abstraction - lower layers might capture local sequential patterns while higher layers model longer-range dependencies and more abstract sequential structures. The depth can be added in multiple ways: stacking multiple recurrent layers, adding deep transitions between time steps, or incorporating deep output layers. The motivation for deep RNNs stems from the proven success of depth in feedforward networks and the observation that single-layer RNNs may have insufficient representational capacity for complex sequential patterns. While RNNs are already deep when unrolled through time, they apply the same transformation repeatedly, limiting the complexity of features they can learn at

each time step. Deep RNNs allow for more complex per-time-step transformations, enabling the network to learn hierarchical temporal representations analogous to how CNNs learn hierarchical spatial features. In machine translation, lower layers might capture syntactic patterns while higher layers model semantic relationships. Speech recognition systems commonly use deep RNNs where initial layers process acoustic features, middle layers capture phonetic patterns, and upper layers model linguistic structure. However, training deep RNNs presents additional challenges beyond standard RNNs - gradient flow becomes even more problematic as gradients must traverse both depth and time dimensions, often requiring careful initialization strategies, residual connections between layers, or layer normalization to maintain stable training dynamics. Modern architectures like stacked LSTMs or deep GRUs have become standard in applications requiring sophisticated sequential modeling, with depths typically ranging from 2-8 layers depending on the task complexity and available training data.