INFO 7375 - Neural Networks & AI

Home Work to Chapter - 20

Submitted By:

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

How does language model (LM) work?

Language models fundamentally operate as probability distributions over sequences of words or tokens, learning to assign probabilities to sequences based on patterns observed in training data. At its core, a language model estimates $P(w_1, w_2, ..., w_n)$, the joint probability of a sequence of words, which is typically factorized using the chain rule into a product of conditional probabilities:
$P(w_1, w_2, ..., w_n) = P(w_1) \times P(w_2 | w_1) \times P(w_3 | w_1, w_2) \times ... \times P(w_n | w_1, ..., w_{n-1})$. This decomposition allows the model to generate text autoregressively by predicting one word at a time conditioned on previous context. Neural language models, particularly those based on RNNs, maintain a hidden state that summarizes the history of previously seen tokens, updating this representation as each new token is processed: $h_t = f(W_hh * h_{t-1} + W_xh * x_t)$, where x_t is typically an embedding vector representing the current token.

The architecture transforms discrete tokens into continuous representations through an embedding layer that maps each vocabulary item to a dense vector in a learned embedding space. These embeddings capture semantic and syntactic relationships between words, with similar words occupying nearby regions in the embedding space. As the sequence is processed, the hidden state accumulates information about the context, theoretically capturing long-range dependencies and complex linguistic patterns. At each position, the model projects the hidden state through an output layer (typically a linear transformation followed by softmax) to produce a probability distribution over the entire vocabulary:
$P(w_t | w_1, ..., w_{t-1}) = softmax(W_{out} * h_t + b_{out})$. This distribution represents the model's belief about which word should come next given the context it has observed. Modern language models have evolved from simple RNN architectures to more sophisticated designs like LSTMs, GRUs, and eventually Transformers, but the fundamental principle of modeling sequential dependencies through learned representations remains constant.

How does word prediction work?

Word prediction in neural language models involves transforming the abstract hidden state representation into concrete probability distributions over the vocabulary through a series of computational steps. After processing the context through recurrent layers, the final hidden state h_t passes through a projection layer that maps from the hidden dimension to the vocabulary size:$z_t = W_{vocab} * h_t + b_{vocab}$, where $W_{vocab} \in \mathbb{R}^{|V| \times d_{hidden}}$ and $| V |$ represents vocabulary size. This produces raw scores (logits) for each word in the vocabulary, which are then normalized using the softmax function:$P(w_i | context) = exp(z_{t[i]}) / \Sigma_j exp(z_{t[j]})$, ensuring the output forms a valid probability distribution summing to 1. The word with the highest probability can be selected greedily, though this often leads to repetitive or suboptimal text generation.

Practical word prediction employs various decoding strategies to balance between exploration and exploitation of the probability distribution. Greedy decoding simply selects argmax_w P(w| context) at each step but can get stuck in repetitive patterns. Beam search maintains k parallel hypotheses, expanding the k most promising continuations at each step and ultimately selecting the sequence with highest overall probability, though it tends to favor generic, safe predictions.

Sampling-based approaches introduce randomness by drawing from the probability distribution rather than always selecting the maximum, with temperature scaling controlling the randomness: P(w_i) ∝ exp(z_t[i]/T), where higher temperature T produces more diverse but potentially less coherent text. Top-k sampling restricts sampling to the k most likely words, while nucleus (top-p) sampling dynamically selects the smallest set of words whose cumulative probability exceeds threshold p, adapting to the uncertainty at each position. These techniques can be combined - for instance, applying temperature scaling before top-p sampling to achieve desired generation characteristics balancing fluency, diversity, and coherence.

How to train an LM?

Training a language model involves optimizing the model parameters to minimize the negative log-likelihood of observed sequences in the training corpus, equivalent to minimizing cross-entropy between predicted and actual next-word distributions. Given a training sequence $(w_1, w_2, ..., w_T)$, the loss function is $L = -\Sigma_{t=1}^{T} logP(w_t | w_1, ..., w_{t-1}; \theta)$, where θ represents all model parameters including embeddings, recurrent weights, and output projections. This objective encourages the model to assign high probability to the actual next word observed in the training data at each position. The training process uses teacher forcing, where the true previous words are provided as input during training rather than the model's own predictions, ensuring stable gradient flow and faster convergence, though this creates a train-test mismatch known as exposure bias. The optimization typically proceeds through minibatch stochastic gradient descent, where sequences are grouped into batches for computational efficiency. For RNN-based models, backpropagation through time computes gradients by unrolling the network for a fixed number of steps (truncated BPTT) to manage memory constraints and gradient stability. Key training considerations include gradient clipping to prevent exploding gradients (typically clipping the global norm of gradients to a threshold like 5.0), learning rate scheduling with warmup periods followed by decay, and dropout regularization applied to embeddings and hidden states to prevent overfitting. The perplexity metric, defined as PP = exp(L/T), serves as the standard evaluation measure, representing the average branching factor or uncertainty when predicting the next word. Modern language model training also incorporates techniques like curriculum learning (starting with shorter sequences), data augmentation through paraphrasing or back-translation, and mixed precision training to accelerate computation. Large-scale language models require distributed training across multiple GPUs or TPUs, with techniques like gradient accumulation to simulate larger batch sizes and gradient checkpointing to reduce memory consumption during backpropagation.

Describe the problem and the nature of vanishing and exploding gradients

The vanishing and exploding gradient problems represent fundamental challenges in training RNNs that arise from the repeated application of the same weight matrices across many time steps. During backpropagation through time, computing the gradient of the loss with respect to hidden states at early time steps requires multiplying many Jacobian matrices: $\partial h_t / \partial h_k = \Pi_{i=k+1}^{t} \partial h_i / \partial h_{i-1}$, where each term $\partial h_i / \partial h_{i-1} = W_h h^T * diag(f'(z_{i-1}))$for activation function f. When this product involves many terms (long sequences), the resulting gradient's magnitude depends critically on the eigenvalues of $W_{hh}$and the activation function derivatives. If the largest eigenvalue λ_max < 1, the gradient shrinks exponentially as$(\lambda_{max})^{t-k}$, while if λ_max > 1, it grows exponentially, with the number of time steps acting as an exponent that amplifies these effects dramatically. The vanishing gradient problem manifests when gradients become exponentially small, effectively preventing the network from learning long-range dependencies because errors from distant time steps contribute negligibly to parameter updates. Consider a sequence where understanding the ending requires information from the beginning with vanishing gradients, the network cannot propagate error signals far enough back to establish this connection. The gradient magnitude might decay from 1.0 to 10^-10 or smaller over just 10-20 time steps, making those early states essentially invisible to the learning process. Conversely, exploding gradients cause parameter updates of enormous magnitude that destabilize training, potentially causing overflow errors or sending parameters into regions where the loss function becomes nan. The problem is particularly severe for vanilla RNNs using tanh or sigmoid activations, whose derivatives are bounded by 1 and often much smaller in practice (sigmoid derivative peaks at 0.25), naturally promoting gradient vanishing. While gradient clipping provides a crude solution for explosion by thresholding gradient norms, and careful initialization (like orthogonal initialization of W_hh), these approaches don't fundamentally solve the issue of learning long-range dependencies, motivating the development of gated architectures like LSTMs and GRUs that maintain gradient highways through time.

What is LSTM and the main idea behind it?

Long Short-Term Memory networks revolutionize sequence modeling through an ingenious gating mechanism that creates controllable information highways through time, fundamentally addressing the vanishing gradient problem. The key innovation is the cell state c_t, a separate memory vector that flows through time with minimal transformations, protected from the repeated nonlinearities that plague vanilla RNN hidden states. The cell state update follows $c_t = f_t \odot c_{t-1} + i_t \odot g_t$, where the forget gate $f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)$ determines what information to discard from previous memory, the input gate $i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i)$ controls what new information to store, and the candidate values $g_t = tanh(W_g * [h_{t-1}, x_t] + b_g)$ represent potential new memories. The output gate $o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o)$ then controls what parts of the cell state to expose as the hidden state: h_t = o_t ⊙ tanh(c_t). This architecture allows gradients to flow through the cell state with only element-wise multiplications by the forget gate values, which can be close to 1 for important information, preserving gradient magnitude across many time steps.

The mathematical elegance of LSTMs lies in how they balance memory preservation with selective updating and output generation. Each gate serves a distinct purpose: forget gates learn to maintain information for exactly as long as needed (potentially hundreds of time steps for important features), input gates prevent irrelevant information from polluting the memory, and output gates ensure only currently relevant information affects predictions. During backpropagation, the gradient ∂L/∂c_t flows back through time via $\partial c_t / \partial c_{t-1} = f_t$, meaning gradients are multiplied by learnable forget gate values rather than fixed weight matrices. When the network learns that certain information is important, it sets f_t ≈ 1, creating an almost perfect gradient highway. This adaptive gradient flow is the crucial insight - the network learns not just what patterns to recognize but also how long to remember them. The additional parameters (roughly 4x compared to vanilla RNNs due to four sets of weights for three gates plus candidates) are a worthwhile trade-off for the ability to capture long-range dependencies that span hundreds or thousands of time steps, making LSTMs the architecture of choice for tasks like language modeling, speech recognition, and machine translation before the advent of Transformers.

What is GRU?

Gated Recurrent Units represent an elegant simplification of LSTMs that achieves comparable performance with fewer parameters by unifying the cell state and hidden state into a single state vector and reducing three gates to two. The GRU update equations center on the update gate $z_t = \sigma(W_z * [h_{t-1}, x_t] + b_z)$, which interpolates between the previous hidden state and a candidate new state, and the reset gate$r_t = \sigma(W_r * [h_{t-1}, x_t] + b_r)$, which controls how much past information influences the candidate state computation. The candidate hidden state is computed as $\tilde{h}_t = tanh(W_h * [r_t \odot h_{t-1}, x_t] + b_h)$, where the reset gate modulates the influence of the previous hidden state through element-wise multiplication. The final hidden state update combines the previous state and candidate through the update gate:$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$, effectively implementing a learnable interpolation between maintaining old information and incorporating new information. The philosophical difference between GRUs and LSTMs reflects different approaches to information flow control - while LSTMs maintain separate memory and hidden states with independent control over reading and writing, GRUs adopt a more streamlined approach where the update gate simultaneously controls forgetting and input. When z_t approaches 0, the unit maintains its previous state unchanged, creating gradient highways similar to LSTM cell states, while z_t approaching 1 allows the unit to completely replace its state with new information. The reset gate provides additional flexibility by allowing the network to drop previous information when computing candidates, effectively implementing a soft reset mechanism. This architectural simplification reduces parameters by approximately 25% compared to LSTMs (three weight matrix sets instead of four) while empirical studies show comparable or sometimes superior performance, particularly on smaller datasets where the reduced parameter count helps prevent overfitting. The GRU's simpler gating mechanism also translates to faster training and inference, making it an attractive choice when computational efficiency matters. The success of GRUs demonstrates that the core principle of adaptive gated information flow matters more than the specific gating architecture, and their widespread adoption alongside LSTMs shows that different sequence modeling tasks may favor different approaches to managing temporal information flow.