**CNN-Based Neural Language Model**

**Main Working**

1. Input is a sequence of tokens. Each token is embedded into a fixed-size vector, forming an input matrix of shape *n × d* (n = number of tokens, d = embedding size).
2. A fixed-size context window (e.g., 4 words) is applied to the input. Only this subset of the sentence is processed at a time.
3. The embeddings in the window are concatenated into a single vector of size *4d*.
4. A weight matrix of shape *4d × h* (where h = hidden dimension) is used to compute a hidden representation via a convolutional operation.
5. This hidden vector is passed through a linear layer of shape *h × V* (V = vocabulary size) to produce logits for all possible next words.
6. Softmax is applied to the logits to get a probability distribution. The highest-scoring word is predicted as the next word.
7. The model is trained to maximize the likelihood of the correct next word using cross-entropy loss.

**Advantages**

1. Uses dense word embeddings, allowing better generalization compared to sparse co-occurrence matrices in statistical models.
2. Supports parallel computation within each context window, leading to efficient training.
3. Learns contextual patterns within a fixed window, effectively capturing local word dependencies.
4. Trains well on large datasets using backpropagation and gradient descent.

**Disadvantages**

1. Fixed-size context window limits the model’s ability to capture long-range dependencies.
2. No interaction between word positions within the window — each token contributes independently to the output.
3. Increased window size increases the number of parameters and computation cost linearly.
4. Inefficient for tasks requiring understanding of sentence-wide or document-wide context.

**Recurrent Neural Networks (RNNs)**

**Main Working**

1. **Input Representation**:
   * Inputs are embeddings: X₁, X₂, X₃, X₄.
   * Each input position corresponds to a hidden state: h₁, h₂, h₃, h₄.
   * h₀ is a dummy initial hidden state.
2. **Hidden State Computation**:
   * At each time step t, hidden state hₜ is computed using:

ht=nonlinearity(ht−1Wh+xtWe+b)h\_t = \text{nonlinearity}(h\_{t-1}W\_h + x\_tW\_e + b)ht​=nonlinearity(ht−1​Wh​+xt​We​+b)

where:

* + - Wh​ = weight matrix for previous hidden state.
    - We​ = weight matrix for current input.
    - b = bias vector.
    - Nonlinearity = usually sigmoid or tanh.

1. **Weight Sharing**:
   * Wh​ and We​ are the same across all time steps (shared parameters).
   * Model is context-independent in terms of weight size.
2. **Dimensions**:
   * Dimensions depend only on hidden state size and embedding size, not input length.
3. **Embedding Lookup**:
   * A one-hot encoded word vector is multiplied with the pre-defined embedding matrix to retrieve the embedding.
4. **Prediction**:
   * Last hidden state (e.g., h₄) is passed through a linear layer + nonlinearity to generate a probability distribution over vocabulary.
   * Sampling is done from the distribution to generate the predicted word.

**Advantages**

1. **Handles Any Context Size**:
   * Not limited by a fixed context window like CNNs.
   * Can theoretically capture long-distance dependencies.
2. **Sequential Information Flow**:
   * Each hidden state incorporates information from all previous time steps.
   * In ideal cases, hₜ carries information from x₁ to xₜ.
3. **Parameter Efficiency**:
   * Model size does not grow with input sequence length.
   * Same set of parameters used across all time steps.
4. **Symmetry**:
   * All words are processed using the same projection (e.g., E₁, E₂ use same WeW\_eWe​).
   * Fair representation across words.

**Disadvantages**

1. **Sequential Computation = Slow**:
   * hₜ cannot be computed until hₜ₋₁ is ready.
   * Not parallelizable due to time-step dependency.
2. **Vanishing Gradient Problem**:
   * Despite theoretical capability, long-distance dependencies degrade over time.
   * As steps increase, influence from earlier words (e.g., x₁) on later outputs diminishes.

**Training RNNs**

1. **Objective**:
   * Predict next word at each time step using current hidden state.
2. **Loss Function**:
   * Use **Cross Entropy Loss** between predicted distribution and actual (ground-truth) word at each step.
   * For T steps:

Total Loss=1T∑t=1TJt(θ)\text{Total Loss} = \frac{1}{T} \sum\_{t=1}^{T} J\_t(\theta)Total Loss=T1​t=1∑T​Jt​(θ)

1. **Teacher Forcing**:
   * Two strategies during training:
     + **With Teacher Forcing**: Ground truth of previous time step is fed as input.
     + **Without Teacher Forcing**: Model’s own output is used for the next step.
   * Teacher forcing helps in stable and faster training.

**Cross Entropy in RNNs**

* **What is it for?**  
  Cross entropy measures how well the predicted output (softmax probabilities) aligns with the true label (one-hot encoded).
* **What does it do?**  
  It only focuses on the position where the actual class is 1. The loss is the negative log of the predicted probability for the correct word.

**🔹 Parameters in RNN**

* The RNN has **three main parameters**:
  + **W**: weight matrix for the hidden state transition (the tricky one to train)
  + **U** and **V** (or sometimes called W\_u and W\_v): relatively simpler to update

**🔹 Backpropagation Through Time (BPTT)**

* **Why is it needed?**  
  Regular backpropagation doesn't handle the time steps in RNNs. BPTT handles how errors flow back through all time steps.
* **How does it work?**  
  You apply the **chain rule** repeatedly to backpropagate the loss from the last time step back to earlier ones.
* **Idea:**  
  Each hidden state (e.g., h3h\_3h3​) is not just dependent on the weight matrix **W** directly but also through earlier hidden states like h2,h1,h0h\_2, h\_1, h\_0h2​,h1​,h0​. So, the derivative of loss with respect to **W** accumulates all paths through time.

**🔹 Multivariable Chain Rule in BPTT**

* Used to **track dependencies** from the loss all the way back to previous time steps.
* For example, to compute how h3h\_3h3​ affects the loss and how W affects h3h\_3h3​, we follow multiple indirect paths from h3h\_3h3​ to WWW via earlier hidden states.

**🔹 Summary of BPTT Gradient Update**

* The total derivative of the loss w.r.t **W** is a **summation** of how every earlier hidden state contributes to the final loss.
* These paths can be long and include multiple products of gradients at each time step.

**🔹 Matrix and Vector Derivatives**

* **Why it's tricky:**  
  You're working with vectors and matrices. So, the derivatives produce other matrices—not just scalars.
* Thankfully, modern libraries (like Hugging Face and PyTorch) handle these automatically.

**🔹 Vanishing and Exploding Gradients**

* **Vanishing Gradient Problem:**
  + When gradients are very small and keep multiplying, the contribution of earlier time steps (like h1h\_1h1​) becomes negligible.
  + This means the model *forgets* earlier inputs.
  + Happens when the **largest eigenvalue** of the gradient matrices is < 1.
* **Exploding Gradient Problem:**
  + Opposite scenario—when gradients grow too large.
  + Addressed by **gradient clipping** (restricting gradients to a max threshold).

**🔹 Truncated Backpropagation Through Time**

* **Solution to vanishing gradient:**  
  Instead of propagating loss all the way back to the beginning, **truncate** the number of time steps.
* This reduces complexity and controls vanishing gradients by not going back too far.