Basic and Recurrent Neural Language Models: Implementation and Comparison

February 2026

1 Biggest Takeaways

Neural language models learn to predict the next word in a text. They use word embeddings, which are vectors learned from data. Compared with classic n-gram models, they do not rely on large count tables. Instead, they learn dense representations and can generalize better to similar contexts.

This report compares two neural approaches. The first one uses a fixed context window. It looks at the last few tokens only, so it is simple and fast, but it cannot use information outside this window. The second one is recurrent (RNN). It keeps a hidden state that is updated at each step. This can help to use longer context, but training is slower because it processes tokens in order.

2 Models

2.1 Basic Neural Language Model (Fixed Context)

Given a context of C previous tokens $(w_{t-C}, \ldots, w_{t-1})$, the model looks up the embeddings of these tokens and concatenates them. Then it applies one hidden layer with a tanh activation. Finally, it uses a softmax layer to predict a probability distribution over the vocabulary for the next token.

2.2 RNN Neural Language Model

The recurrent model processes tokens sequentially. At each time step t, it updates a hidden state h_t from the current embedding and the previous state, then predicts the next token via a softmax. This experiment uses a simple tanh RNN with truncated backpropagation through time on fixed-length sequences.

3 Experimental Setup

3.1 Datasets

Two local datasets were used. The first dataset is News (HuffPost headlines and short descriptions) from `data/News_Category_Dataset_v2.json`. The second dataset is Amazon Reviews (review title and review body) from `data/AmazonReviews/train.csv`.

3.2 Preprocessing

Text was lowercased and tokenized at word and punctuation level. Documents were concatenated into a single stream with `<bos>` and `<eos>` markers. For each dataset, a capped word vocabulary of size 1200 was built (including `<pad>`, `<unk>`, `<bos>`, and `<eos>`), and tokens with frequency smaller than 2 were removed. Out-of-vocabulary tokens map to `<unk>`.

3.3 Training and Evaluation

Both models were implemented in PyTorch (`torch.nn.Module`) and trained with Adam and gradient clipping. The main hyperparameters are: embedding dimension 32, hidden dimension 64, learning rate 0.003, context size 4 for the Basic model, sequence length 20 for the RNN model, and 3 training epochs.

Perplexity (PPL) is reported on held-out validation samples as $\exp(cross - entropy)$.

4 Results

4.1 Validation Perplexity

Dataset	Basic NLM PPL	RNN NLM PPL
News	92.25	99.77
Amazon	133.21	161.92

Table 1: Validation perplexity (lower is better).

4.2 Figures

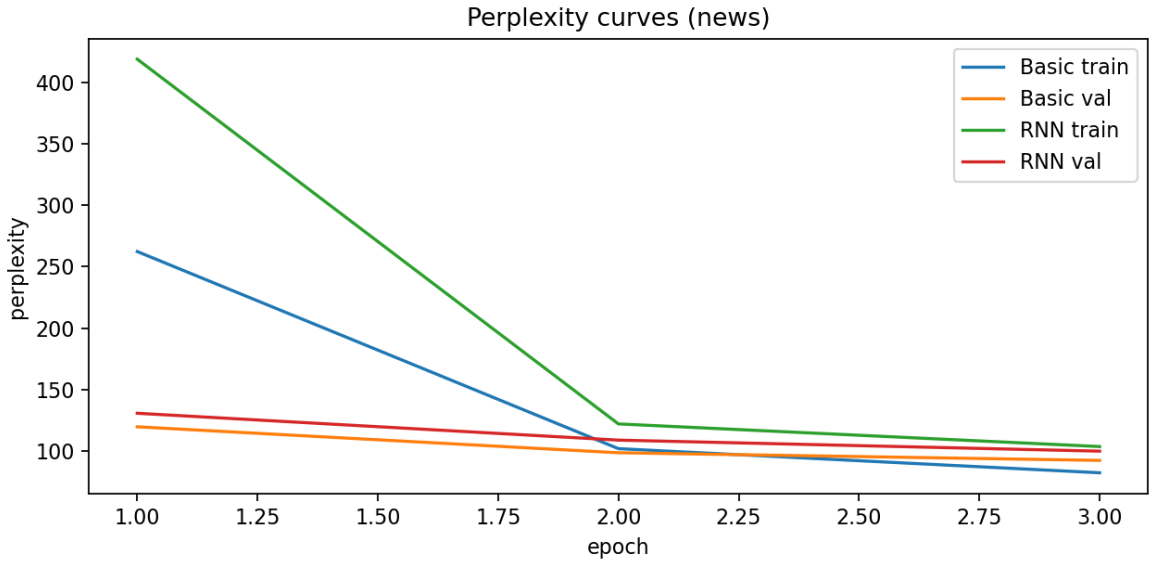

Figure 1: Perplexity curves on the News dataset.

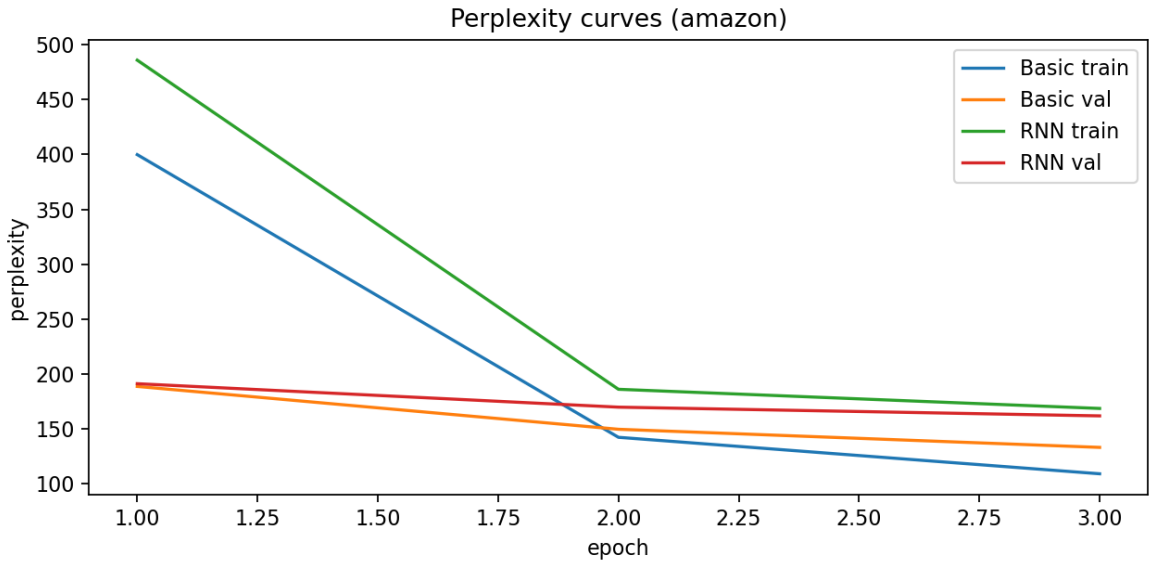

Figure 2: Perplexity curves on the Amazon dataset.

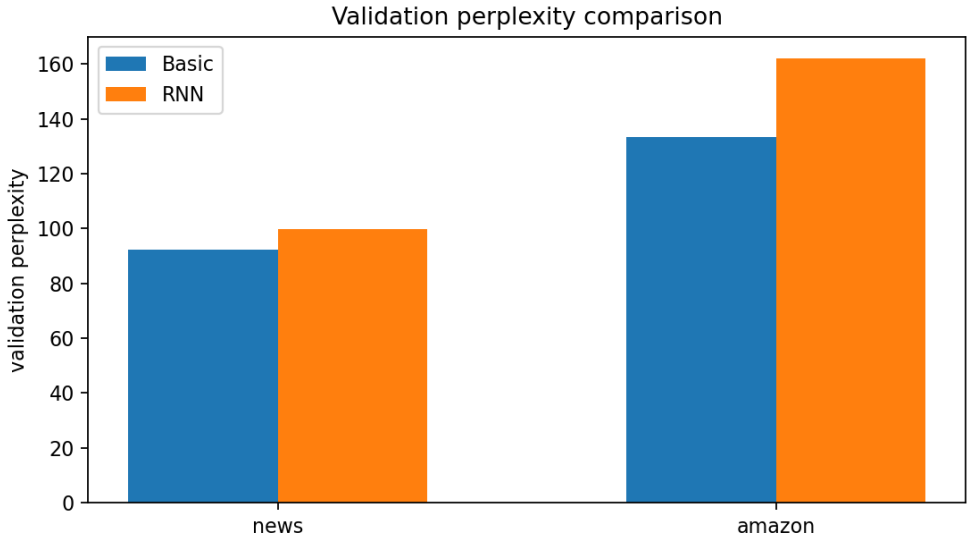

Figure 3: Final validation perplexity comparison across datasets.

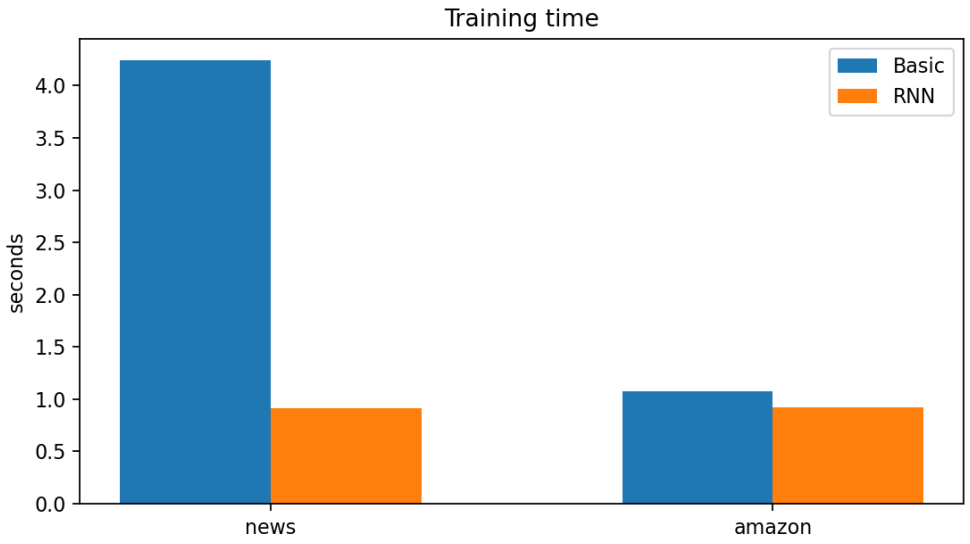

Figure 4: Training time comparison.

5 Discussion

The results highlight a practical tradeoff between the two formulations. The fixed-window Basic NLM is easy to optimize and can be competitive when the next token is mostly determined by a short local context. The recurrent model removes the hard context cutoff by carrying a hidden state forward, which can help when longer dependencies matter, but training is sequential and often benefits from careful tuning (hidden size, sequence length, number of epochs).