Recurrent Neural Networks (RNN) and Natural Language Processing (NLP)

1. Introduction to Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks designed to process sequential data. Unlike feedforward networks, RNNs have cyclic connections that allow information to persist across time steps. This makes them well-suited for tasks where the order of inputs matters, such as natural language processing (NLP), time series forecasting, speech recognition, and more.

- **Key Property**: RNNs maintain a hidden state $\mathbf{h}_t$ that acts as a memory, capturing information about preceding elements in the sequence.
- **Typical Applications in NLP**:
  - Language Modeling
  - Text Classification
  - Sequence Tagging (e.g., POS tagging, Named Entity Recognition)
  - Machine Translation
  - Text Generation

2. Vanilla (Elman) RNN Architecture

2.1. Basic RNN Cell

At each time step t, the RNN cell takes the input vector $\mathbf{x}*t* \in *\mathbb{R}^{d_x}* $and the previous hidden state $*\mathbf{h}*$\mathbf{h}_{t-1} \in \mathbb{R}^{d_h}$ to produce a new hidden state $\mathbf{h}_t \in \mathbb{R}^{d_h}$. A simple formulation (Elman RNN) is:

$$ \mathbf{h}_t ;=; \phi\bigl(\mathbf{U},\mathbf{x}*t ;+; *\mathbf{W},\mathbf{h}*$\mathbf{h}_{t-1} ;+; \mathbf{b}\bigr) $$

- $\mathbf{U} \in \mathbb{R}^{d_h \times d_x}$: input-to-hidden weight matrix
- $\mathbf{W} \in \mathbb{R}^{d_h \times d_h}$: hidden-to-hidden (recurrent) weight matrix
- $\mathbf{b} \in \mathbb{R}^{d_h}$: bias vector
- $\phi(\cdot)$: element-wise nonlinearity (e.g., $\tanh$, ReLU, or sigmoid)

If we want to produce an output $\mathbf{y}_t \in \mathbb{R}^{d_y}$ at each step, we can add:

$$\mathbf{o}_t ;=; \mathbf{V},\mathbf{h}_t ;+; \mathbf{c}, \quad \hat{\mathbf{y}}_t ;=; \mathrm{softmax}(\mathbf{o}_t)$$

- $\mathbf{V} \in \mathbb{R}^{d_y \times d_h}$: hidden-to-output weight matrix
- $\mathbf{c} \in \mathbb{R}^{d_y}$: output bias
- $\hat{\mathbf{y}}_t$: predicted probability distribution (e.g., over vocabulary in language modeling)

2.2. Unfolding Through Time

An RNN over a sequence of length T can be "unfolded" into T copies of the same cell, each passing its hidden state to the next. Let the sequence of inputs be $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$ and hidden states $\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_T$ (with $\mathbf{h}_0$ often initialized as zeros). The recurrence:

For $t = 1, 2, \ldots, T$: $$ \begin{aligned} \mathbf{h}_t &= \phi\bigl(\mathbf{U},\mathbf{x}*t ;+; *mathbf{W},*mathbf{h}*{t-1} ;+; \mathbf{b}\bigr), \ \mathbf{o}_t &= \mathbf{V},\mathbf{h}_t ;+; \mathbf{c}, \ \hat{\mathbf{y}}_t &= \mathrm{softmax}(\mathbf{o}_t). \end{aligned} $$

2.3. Loss Function (Sequence-Level)

For a many-to-many task (e.g., language modeling), we compute the loss at each time step and sum (or average) over T:

- **Cross-Entropy Loss** for classification at each step:

  $$ \mathcal{L}*t ;=; - *sum*{i=1}^{d_y} y_{t,i} ,\log \bigl(\hat{y}_{t,i}\bigr) $$ where $\mathbf{y}_t$ is the one-hot (or target probability) vector.

- **Total Loss** over the sequence of length T:

  $$ \mathcal{L} ;=; \sum_{t=1}^{T} \mathcal{L}*t *quad *text{or} *quad *frac{1}{T}*sum*{t=1}^{T} \mathcal{L}_t. $$

Backpropagation Through Time (BPTT) is used to compute gradients of $\mathcal{L}$ with respect to all parameters $\mathbf{U}, \mathbf{W}, \mathbf{b}, \mathbf{V}, \mathbf{c}$ by unrolling the network over T time steps.

2.4. Vanishing and Exploding Gradients

Because the same weight matrix $\mathbf{W}$ is multiplied repeatedly during BPTT, gradients can:

- **Vanish**: $|\mathbf{W}^t| \to 0$ if $|\mathbf{W}| < 1$, making it hard to learn long-range dependencies.
- **Explode**: $|\mathbf{W}^t| \to \infty$ if $|\mathbf{W}| > 1$, causing instability.

Common solutions:

- **Gradient clipping**: Clip the norm of the gradient to a threshold.
- **Advanced RNN cells**: Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) mitigate vanishing gradients via gating mechanisms.

3. Long Short-Term Memory (LSTM)

LSTM cells introduce gates to control the flow of information and maintain long-range dependencies more effectively. An LSTM cell has:

- **Input Gate** $\mathbf{i}_t$
- **Forget Gate** $\mathbf{f}_t$
- **Output Gate** $\mathbf{o}_t$
- **Cell State** $\mathbf{c}_t$
- **Candidate Cell Update** $\tilde{\mathbf{c}}_t$

3.1. LSTM Equations

For each time step t:

1. **Forget Gate**: $$ \mathbf{f}_t ;=; \sigma\bigl(\mathbf{W}_f \mathbf{x}_t ;+; \mathbf{U}*f *mathbf{h}*{t-1} ;+; \mathbf{b}_f \bigr) $$

2. **Input Gate**: $$ \mathbf{i}_t ;=; \sigma\bigl(\mathbf{W}_i \mathbf{x}_t ;+; \mathbf{U}*i* *mathbf{h}*{t-1} ;+; \mathbf{b}_i \bigr) $$
3. **Candidate Cell State**: $$ \tilde{\mathbf{c}}_t ;=; \tanh\bigl(\mathbf{W}_c \mathbf{x}_t ;+; \mathbf{U}*c* *mathbf{h}*{t-1} ;+; \mathbf{b}_c \bigr) $$
4. **Cell State Update**: $$ \mathbf{c}_t ;=; \mathbf{f}*t* *odot* *mathbf{c}*{t-1} ;+; \mathbf{i}_t \odot \tilde{\mathbf{c}}_t $$
5. **Output Gate**: $$ \mathbf{o}_t ;=; \sigma\bigl(\mathbf{W}_o \mathbf{x}_t ;+; \mathbf{U}*o* *mathbf{h}*{t-1} ;+; \mathbf{b}_o \bigr) $$
6. **Hidden State**:

$$\mathbf{h}_t ;=; \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

- $\sigma(\cdot)$: sigmoid activation
- $\odot$: element-wise multiplication

3.2. LSTM Parameters

- $\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_c, \mathbf{W}_o \in \mathbb{R}^{d_h \times d_x}$
- $\mathbf{U}_f, \mathbf{U}_i, \mathbf{U}_c, \mathbf{U}_o \in \mathbb{R}^{d_h \times d_h}$
- $\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_c, \mathbf{b}_o \in \mathbb{R}^{d_h}$

The LSTM's gating mechanism allows it to "remember" or "forget" information in the cell state, thus mitigating the vanishing gradient problem and enabling learning of long-range dependencies.

4. Gated Recurrent Unit (GRU)

The Gated Recurrent Unit (Cho et al., 2014) is a streamlined variant of LSTM with fewer gates. It merges the forget and input gates into a single **update gate** $\mathbf{z}_t$ and uses a **reset gate** $\mathbf{r}_t$.

4.1. GRU Equations

For each time step t:

1. **Update Gate**: $$ \mathbf{z}_t ;=; \sigma\bigl(\mathbf{W}_z \mathbf{x}_t ;+; \mathbf{U}*z* *mathbf{h}*{t-1} ;+; \mathbf{b}_z\bigr) $$
2. **Reset Gate**: $$ \mathbf{r}_t ;=; \sigma\bigl(\mathbf{W}_r \mathbf{x}_t ;+; \mathbf{U}*r* *mathbf{h}*{t-1} ;+; \mathbf{b}_r\bigr) $$
3. **Candidate Activation**: $$ \tilde{\mathbf{h}}_t ;=; \tanh\bigl(\mathbf{W}_h \mathbf{x}_t ;+; \mathbf{U}_h(\mathbf{r}*t* *odot* *mathbf{h}*{t-1}) ;+; \mathbf{b}_h\bigr) $$
4. **Hidden State Update**: $$ \mathbf{h}_t ;=; (1 - \mathbf{z}*t)* *odot* *mathbf{h}*{t-1} ;+; \mathbf{z}_t \odot \tilde{\mathbf{h}}_t $$

- $\mathbf{z}_t$ decides how much of the previous hidden state to keep (like a combined forget/input gate).
- $\mathbf{r}_t$ controls how much past information to forget when computing the candidate $\tilde{\mathbf{h}}_t$.

Parameters:

- $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}_h \in \mathbb{R}^{d_h \times d_x}$
- $\mathbf{U}_z, \mathbf{U}_r, \mathbf{U}_h \in \mathbb{R}^{d_h \times d_h}$
- $\mathbf{b}_z, \mathbf{b}_r, \mathbf{b}_h \in \mathbb{R}^{d_h}$

GRUs often perform similarly to LSTMs but with fewer parameters and slightly faster training.

5. RNN Variants and Enhancements

1. **Bidirectional RNN (BiRNN)**: Processes the sequence in both forward and backward directions and concatenates hidden states.
    - Forward hidden state: $\overrightarrow{\mathbf{h}}_t = \phi(\mathbf{U}_f x_t + \mathbf{W}*f* *overrightarrow{\mathbf{h}}*{t-1} + \mathbf{b}_f)$
    - Backward hidden state: $\overleftarrow{\mathbf{h}}_t = \phi(\mathbf{U}_b x_t + \mathbf{W}*b* *overleftarrow{\mathbf{h}}*{t+1} + \mathbf{b}_b)$
    - Combined: $\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t; \overleftarrow{\mathbf{h}}_t]$
2. **Deep RNN**: Stacking multiple RNN layers. Hidden state of layer ℓ becomes input to layer $\ell + 1$.
    - Layer 1: $\mathbf{h}*t^{(1)} = *phi(*mathbf{U}*^{(1)} x_t + *mathbf{W}*^{(1)} *mathbf{h}*{t-1}^{(1)} + \mathbf{b}^{(1)})$
    - Layer 2: $\mathbf{h}_t^{(2)} = \phi(\mathbf{U}^{(2)} \mathbf{h}*t^{(1)} + *mathbf{W}*^{(2)} *mathbf{h}*{t-1}^{(2)} + \mathbf{b}^{(2)})$
    - And so on...
3. **Attention Mechanisms**: Allows the network to focus on different parts of the input sequence when producing each output. Widely used in seq2seq models for machine translation, summarization, etc.

6. Applications in Natural Language Processing

6.1. Language Modeling

Language modeling is the task of assigning a probability to a sequence of words $w_1, w_2, \ldots, w_T$. Using a recurrent architecture:

$$P(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_1, \ldots, w_{t-1}).$$

An RNN-based language model predicts: $$ \hat{\mathbf{y}}*t = P(w_t *mid w*{<t}) = \mathrm{softmax}\bigl(\mathbf{V},\mathbf{h}_t + \mathbf{c}\bigr), $$ where $\mathbf{h}*t $encodeshistory{w_1, \dots, w}*{t-1}}$. The training objective is to minimize negative log-likelihood (equivalent to cross-entropy):

$$ \mathcal{L}*{*text{LM}} = - *sum*{t=1}^{T} \log P(w_t \mid w_{<t}) = - \sum_{t=1}^{T} \log \hat{y}_{t,,\text{idx}(w_t)} $$

- **Perplexity** is a common evaluation metric:

$$\text{Perplexity} = \exp\Big(\frac{1}{T}\sum_{t=1}^{T} -\log P(w_t \mid w_{<t})\Big).$$

6.2. Text Classification

Given a sentence (or document) of length T, represented as words $w_1, \ldots, w_T$, we map each word to an embedding $\mathbf{x}_t$. A vanilla RNN, LSTM, or GRU processes the sequence to produce hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_T$. We then derive a fixed-size representation (e.g., the final hidden state $\mathbf{h}_T$ or an average over $\mathbf{h}_t$) and feed it into a classifier:

1. **Sequence Processing**: $$ \mathbf{h}_t = \text{RNNCell}(\mathbf{x}*t, *mathbf{h}*{t-1}) $$
2. **Representation**: $$ \mathbf{h}_{\text{final}} = \mathbf{h}*T \quad *text{(or)}* \quad *frac{1}{T}**sum*{t=1}^T \mathbf{h}_t $$
3. **Classification Layer**: $$ \mathbf{o} = \mathbf{W}*o,*mathbf{h}*{\text{final}} + \mathbf{b}_o, \quad \hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) $$
4. **Loss** (Cross-Entropy for C classes): $$ \mathcal{L}*{*text{CLS}} ;=; - *sum*{i=1}^{C} y_i ,\log(\hat{y}_i) $$

Variants:

- **Bidirectional RNN**: Take both forward and backward hidden states at the final time step:

$$\mathbf{h}_{\text{bi}} = [, \overrightarrow{\mathbf{h}}_T; , \overleftarrow{\mathbf{h}}_1,] \in \mathbb{R}^{2d_h}.$$

6.3. Sequence Tagging (e.g., POS Tagging, NER)

For token-level tagging, each time step t has an output $\mathbf{y}_t$ (e.g., POS tag or entity label). We use an RNN/LSTM/GRU to compute hidden states $\mathbf{h}_t$ for each token, then apply a classification layer at each t:

1. **Hidden State**: $$ \mathbf{h}_t = \text{RNNCell}(\mathbf{x}*t, *mathbf{h}*{t-1}) $$
2. **Tag Prediction**: $$ \mathbf{o}*t = *mathbf{W}*{\text{tag}},\mathbf{h}*t + *mathbf{b}*{\text{tag}}, \quad \hat{\mathbf{y}}_t = \mathrm{softmax}(\mathbf{o}_t) $$
3. **Loss**: $$ \mathcal{L}*{*text{Tag}} ;=; - *sum*{t=1}^{T} \sum_{i=1}^{C} y_{t,i} ,\log(\hat{y}_{t,i}) $$

Optionally, add a **Conditional Random Field (CRF)** layer on top of RNN outputs to model label dependencies. In that case, scores from RNN are fed into a CRF which computes sequence-level loss.

6.4. Encoder–Decoder (Seq2Seq) for Machine Translation

An encoder processes the source sentence $x_1, \ldots, x_{T_x}$ and compresses it into a context vector. A decoder generates the target sentence $y_1, \ldots, y_{T_y}$ one token at a time.

- **Encoder** (e.g., Bi-LSTM): $$ \overrightarrow{\mathbf{h}}*t^{*text{enc}} = *mathrm{LSTM}*{\text{fwd}}(\mathbf{x}*t, *overrightarrow{*mathbf{h}*}{t-1}^{\text{enc}}), \quad \overleftarrow{\mathbf{h}}*t^{*text{enc}} = *mathrm{LSTM}*{\text{bwd}}(\mathbf{x}*t, *overleftarrow{*mathbf{h}*}{t+1}^{\text{enc}}) $$ Concatenate: $\mathbf{h}_t^{\text{enc}} = [,\overrightarrow{\mathbf{h}}_t^{\text{enc}};,\overleftarrow{\mathbf{h}}_t^{\text{enc}},]$.
- **Decoder** (e.g., LSTM):
  - Initial hidden state $\mathbf{s}_0$ often set from the final encoder states (or a transformation of them).
  - At step t', input previous target embedding $\mathbf{y}*{t'-1}^{*text{embed}} $and previous decoder state $*mathbf{s}*{t'-1}:$ $\mathbf{s}*{t'} = *mathrm{LSTM}*(\mathbf{y}*{t'-1}^{\text{embed}}, \mathbf{s}_{t'-1}). $$
  - **Attention**: Compute attention weights over encoder outputs ${\mathbf{h}*1^{*text{enc}}, *dots, *mathbf{h}*{T_x}^{\text{enc}}}:$ e_{t', j} = \mathbf{v}*a^*top *tanh*bigl(*mathbf{W}*a *mathbf{s}*{t'-1}

+ \mathbf{U}a \mathbf{h}j^{\text{enc}}\bigr), \quad \alpha{t', j} = \frac{\exp(e{t', j})} {\sum{k=1}^{T_x} \exp(e_{t', k})}

$$Contextvector:$$

\mathbf{c}{t'} = \sum{j=1}^{T_x} \alpha_{t', j},\mathbf{h}_j^{\text{enc}}. $$

- **Output Prediction**: $$ \mathbf{o}{t'} = \mathbf{W}o ,[,\mathbf{s}{t'}; \mathbf{c}{t'},] ;+; \mathbf{b}o, \quad \hat{\mathbf{y}}{t'} = \mathrm{softmax}(\mathbf{o}_{t'}). $$

- **Loss**: Sum of cross-entropies over all target time steps: $$ \mathcal{L}{\text{MT}} = -\sum{t'=1}^{T_y} \log P\bigl(y_{t'} \mid y_{<t'},, \mathbf{x}_{1:T_x}\bigr). $$

6.5. Text Generation

Using an RNN-based language model or a seq2seq model, we generate text by sampling or beam search. At test time:

1. Initialize hidden state $\mathbf{h}_0 = \mathbf{0}$ (or learned start state).
2. Feed a special start token $\langle \mathrm{SOS} \rangle$ as $\mathbf{x}_1$.
3. For $t = 1, 2, \ldots$:
   - Compute $\mathbf{h}_t = \phi(\mathbf{U}\mathbf{x}t + \mathbf{W}\mathbf{h}{t-1} + \mathbf{b})$.
   - Compute $\hat{\mathbf{y}}_t = \mathrm{softmax}(\mathbf{V}\mathbf{h}_t + \mathbf{c})$.
   - Sample or choose $\tilde{w}t = \arg\max_i \hat{y}{t,i}$ (or perform beam search).
   - Embed $\tilde{w}ttoform\mathbf{x}{t+1}$.
   - Stop if $\tilde{w}_t$ is the end-of-sequence token $\langle \mathrm{EOS} \rangle$ or a maximum length is reached.

7. Word Embeddings

Before feeding words into RNNs, we map discrete tokens to continuous embeddings.

- **One-Hot Encoding**: $\mathbf{x}_t \in 0,1^{|V|}$, sparse; multiplies with embedding matrix $\mathbf{E} \in \mathbb{R}^{d_x \times |V|}$ to produce dense vector:

$$\mathbf{e}_t = \mathbf{E}, \mathbf{x}_t, \quad \mathbf{e}_t \in \mathbb{R}^{d_x}.$$

- **Pretrained Embeddings**: GloVe, Word2Vec, FastText, or contextual embeddings (ELMo, BERT). RNN input $\mathbf{x}_t = \mathbf{e}_t$.

- **Learned Embeddings**: Train embedding matrix $\mathbf{E}$ jointly with RNN parameters. During backpropagation, update both RNN weights and $\mathbf{E}$.

8. Training Considerations

8.1. Batching and Padding

- Sentences have variable lengths. Pad shorter sequences with a special token (e.g., $\langle \mathrm{PAD} \rangle$) to match the length of the longest sequence in a batch.
- Use a **masking vector** $m_t \in 0,1$ per time step to ignore padded positions in loss computation:

$$\mathcal{L} = -\frac{1}{\sum_{t=1}^{T} m_t} \sum_{t=1}^{T} m_t \sum_{i=1}^{d_y} y_{t,i} \log(\hat{y}_{t,i}).$$

8.2. Sequence Length Truncation

- For very long sequences, perform **truncated BPTT**: backpropagate over fixed windows of length L. Example: every L time steps, detach the hidden state from the graph to save memory.

8.3. Learning Rate and Optimization

- Use optimizers such as **Adam**, **RMSProp**, or **SGD with momentum**.
- Learning rate scheduling (e.g., decay, warm-up) can help stabilize training.

8.4. Regularization

- **Dropout**: Apply dropout on inputs or between RNN layers (e.g., between stacked LSTM layers).
- **Weight Decay (L2 regularization)**: Penalize large weights.
- **Gradient Clipping**: Clip gradient norm to a threshold τ: $$ \text{if } |\nabla_\theta \mathcal{L}|*2 > *tau, \quad *nabla*\theta \mathcal{L} \leftarrow \tau ,\frac{\nabla_\theta \mathcal{L}}{|\nabla_\theta \mathcal{L}|_2}. $$

9. Advanced Topics

9.1. Attention Mechanisms

Attention allows the model to focus on different parts of the input sequence when generating each output token.

- **Bahdanau (Additive) Attention**: $$ e_{t', j} = \mathbf{v}*a^\top *tanh*\bigl(*mathbf{W}*a *mathbf{s}*{t'-1} + *mathbf{U}*a *mathbf{h}*j^{*text{enc}}*\bigr), \quad *alpha*{t', j} = *frac{*exp(e*{t', j})}{*sum{k=1}^{T_x} \exp(e_{t', k})}. $$
- **Luong (Multiplicative) Attention**: $$ e_{t', j} = \mathbf{s}_{t'-1}^\top \mathbf{W}_a ,\mathbf{h}*j^{*text{enc}}, \quad *alpha*{t', j} = \mathrm{softmax}*j(e*{t', j}). $$
- **Context Vector**: $\mathbf{c}*{t'} = *sum{j=1}^{T_x} \alpha_{t', j},\mathbf{h}_j^{\text{enc}}$.
- **Final Decoder Hidden**: Combine $\mathbf{c}*{t'}with decoder state*mathbf{s}*{t'}$ to produce output distribution.

9.2. Sequence-to-Sequence with RNNs

A basic encoder–decoder RNN without attention might encode the entire source sentence into a single vector (last hidden state), which is often a bottleneck for long sequences. Adding attention alleviates this.

9.3. Bidirectional Encoder Representations

Rather than using a unidirectional encoder, one often uses a bidirectional RNN for the encoder side. This captures both past and future context for each source token:

$$\mathbf{h}_j^{\text{enc}} = [, \overrightarrow{\mathbf{h}}_j^{\text{enc}} ; , \overleftarrow{\mathbf{h}}_j^{\text{enc}} ,].$$

9.4. Sequence-to-Sequence with LSTM/GRU

Replace vanilla RNN cells with LSTM or GRU units in the encoder and decoder for better performance on long sequences.

10. Common NLP Tasks and RNN Formulations

Task	Input	RNN Usage	Output
Language Modeling	Sequence of words $w_1, \ldots, w_{T-1}$	RNN predicts next word at each time step	$\hat{w}_t$ for $t = 2, \ldots, T$
Text Classification	Sequence of words $w_1, \ldots, w_T$	RNN final hidden state (or pooled states)	Class probability vector
Sequence Tagging (POS, NER)	Sequence of words $w_1, \ldots, w_T$	RNN hidden state at each time step	Tag distribution at each t
Machine Translation (Seq2Seq)	Source sequence $x_1, \ldots, x_{T_x}$	Encoder RNN → context vectors; Decoder RNN w/ attention	Target sequence $y_1, \ldots, y_{T_y}$
Text Generation	(Optionally prompt / prefix)	RNN generates token-by-token	Generated text sequence

11. Example: One-Layer LSTM for Sentiment Classification

1. **Embedding Layer**: Map each word w_t to $\mathbf{x}_t = \mathbf{E} w_t$.
2. **LSTM Layer**: $$ \mathbf{h}_t,,\mathbf{c}*t* = *mathrm{LSTM}(\mathbf{x}t, (\mathbf{h}{t-1}, *mathbf{c}*{t-1})) $$
3. **Final Representation**: Use $\mathbf{h}_T$ (last hidden state) or apply attention/pooling over $\mathbf{h}_1, \ldots, \mathbf{h}_T$.
4. **Classification Layer**: $$ \mathbf{o} = \mathbf{W}_{\text{cls}},\mathbf{h}*T* + *mathbf{b}*{\text{cls}}, \quad \hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}). $$
5. **Loss**: $$ \mathcal{L}*{*text{sentiment}} = - *sum*{i=1}^C y_i \log(\hat{y}_i). $$

12. Practical Tips and Tricks

- **Initialize Hidden States**: Typically $\mathbf{h}_0 = \mathbf{0}$ (or learn a trainable parameter vector). For LSTM, also initialize $\mathbf{c}_0 = \mathbf{0}$.
- **Gradient Clipping**: Critical for training stability; clip gradients to norm τ.
- **Batching**: Group sequences of similar lengths together to reduce padding waste.
- **Bidirectional and Stacking**: Use BiLSTM or stacked LSTM layers for richer representations.
- **Pretrained Embeddings**: Initialize embedding matrix with GloVe/Word2Vec for faster convergence.
- **Dropout**: Apply dropout between layers or on the inputs to prevent overfitting (e.g., on the embedding layer).
- **Learning Rate Scheduling**: Warm-up + cosine decay or step decay often helps.
- **Regularization**: Consider L2 weight decay on RNN weights.

13. Summary of Key Formulas

1. **Vanilla RNN Update**: $$ \mathbf{h}_t = \phi\bigl(\mathbf{U},\mathbf{x}*t* + *mathbf{W},*mathbf{h}*{t-1} + \mathbf{b}\bigr). $$
2. **LSTM**: $$ \begin{aligned} \mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}*f* *mathbf{h}* {t-1} + \mathbf{b}_f), \ \mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}*i* *mathbf{h}*{t-1} + \mathbf{b}_i), \ \tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}*c* *mathbf{h}*{t-1} + \mathbf{b}_c), \ \mathbf{c}_t &= \mathbf{f}*t* *odot* *mathbf{c}*{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \ \mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}*o* *mathbf{h}*{t-1} + \mathbf{b}_o), \ \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \end{aligned} $$
3. **GRU**: $$ \begin{aligned} \mathbf{z}_t &= \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}*z* *mathbf{h}* {t-1} + \mathbf{b}_z), \ \mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}*r* *mathbf{h}*{t-1} + \mathbf{b}_r), \ \tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h (\mathbf{r}*t* *odot* *mathbf{h}*{t-1}) + \mathbf{b}_h), \ \mathbf{h}_t &= (1 - \mathbf{z}*t) \odot* *mathbf{h}*{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t. \end{aligned} $$
4. **Cross-Entropy Loss (Sequence)**:

$$\mathcal{L} = -\sum_{t=1}^{T}\sum_{i=1}^{d_y} y_{t,i}\log(\hat{y}_{t,i}).$$

5. **Attention Score (Bahdanau)**: $$ e_{t', j} = \mathbf{v}*a*^*top* *tanh(*mathbf{W}a* *mathbf{s}*{t'-1} + *mathbf{U}a* *mathbf{h}j^{*text{enc}}}), *quad* *alpha{t', j} = *frac{*exp(e{t', j})}{*sum*{k=1}^{T_x} \exp(e_{t', k})}. $$

> **Note**: These notes cover the fundamentals of RNN architectures (vanilla, LSTM, GRU), training considerations, and how RNNs are applied in various NLP tasks with accompanying formulas. Feel free to further expand with examples, diagrams, or code snippets as needed!