Assignment 4 — Solutions

Sequence Models: RNN, LSTM, GRU

Part A — Foundations · 22 questions with answers

Q1. What is the difference between an RNN and an ANN? ★

Answer

- **ANN (feed-forward)** – input flows in one direction; each input is processed **independently**; fixed input size; no memory.
- **RNN** – has a **loop**: the hidden state from the previous time step is fed back with the current input, h_t = f(W_xh·x_t + W_hh·h_(t−1) + b). It has **memory**, processes **sequences** of any length, and **shares the same weights** across time steps.

Use ANN for tabular data; RNN for sequential data (text, speech, time series).

Q2. How does backpropagation in an RNN differ from an ANN? What is Backpropagation Through Time (BPTT)?

Answer

In an ANN, backprop flows backward **through layers** only. In an RNN, the network is **unrolled through time** – each time step is like a layer that shares the same weights – and the gradients flow backward **through layers and through time**. This is called **Backpropagation Through Time (BPTT)**.

BPTT steps:

1. Unroll the RNN for T time steps and do the forward pass.
2. Compute the loss (at each step or at the end).
3. Backpropagate from step T to step 1; the gradient of a shared weight is the **sum of its gradients at all time steps**.
4. Update the shared weights.

Because gradients are multiplied across many steps, BPTT suffers from vanishing/exploding gradients. **Truncated BPTT** limits backprop to the last k steps to save compute.

Q3. How does an RNN handle variable-length inputs?

Answer

- The same cell (same weights) is applied step by step, so in principle it can process **any number of time steps**.
- For **batch training**, sequences in a batch must have the same length, so we use **padding** (add zeros) to the longest/max length and **truncation** for long sequences.
- **Masking** (`mask_zero=True` in Embedding, or a Masking layer) tells the RNN to ignore padded steps.
- In PyTorch, `pack_padded_sequence` skips padding efficiently.
- Bucketing (grouping sequences of similar length) reduces wasted padding.

Q4. What are the three weight matrices used in an RNN? Explain parameter sharing.

Answer

Three weight matrices:

- **W_xh** (input → hidden): transforms the current input x_t.
- **W_hh** (hidden → hidden): transforms the previous hidden state h_(t−1) – the "memory" connection.
- **W_hy** (hidden → output): produces the output y_t from the hidden state.

Parameter sharing: the **same** W_xh, W_hh and W_hy are used at **every time step**. Benefits: the number of parameters does not grow with sequence length, the model can handle variable-length sequences, and a pattern learned at one position (e.g. "not good") is recognized at any position. (CNNs share parameters across space in the same way.)

Q5. What is a bidirectional RNN? When is it useful?

Answer

A **bidirectional RNN** has two RNNs: one reads the sequence **forward** (left → right) and the other **backward** (right → left). Their hidden states are concatenated at each time step, so each output has context from both past and future words.

Useful when the whole sequence is available: NER, POS tagging, sentiment classification, speech recognition, machine-translation encoders.

Not usable for real-time/autoregressive generation (predicting the next word), because the future is not known.

Q6. Why do RNNs suffer from vanishing gradients and how can it be solved? ★ · Asked: NLP interview rounds

Answer

During BPTT, the gradient for early time steps is a product of many terms involving W_hh and the derivative of tanh (≤ 1). Over long sequences this product shrinks exponentially, so the gradient from words far back becomes ~0. Result: the RNN **cannot learn long-term dependencies** ("The **man** who lived in the house that ... **was** tall").

Solutions:

- **LSTM / GRU** – gated architectures with an additive memory path.
- ReLU activations + identity/orthogonal initialization.
- Gradient clipping (for exploding gradients).
- Skip connections, layer normalization.
- Ultimately **attention / Transformers**, which connect every position directly.

Q7. Explain the LSTM cell: cell state, hidden state and the three gates (forget, input, output). ★ · Asked: NLP interview rounds

Answer

An **LSTM cell** has two states and three gates:

- **Cell state (C_t)** – the long-term memory "conveyor belt"; information flows along it with only small, controlled changes.
- **Hidden state (h_t)** – short-term memory / output at the current step.
- **Forget gate f_t** – decides what to **remove** from the cell state (sigmoid: 0 = forget, 1 = keep).

- **Input gate i_t** – decides what **new information** to add, together with the candidate values C̃_t (tanh).
- **Output gate o_t** – decides what part of the cell state is **output** as the hidden state.

Gates use sigmoid (values 0–1) to act like soft switches.

Q8. How is the cell state updated in an LSTM? Write the equations.

Answer

f_t = σ(W_f·[h_(t−1), x_t] + b_f) (forget gate)

i_t = σ(W_i·[h_(t−1), x_t] + b_i) (input gate)

C̃_t = tanh(W_C·[h_(t−1), x_t] + b_C) (candidate memory)

C_t = f_t ⊙ C_(t−1) + i_t ⊙ C̃_t (cell state update)

o_t = σ(W_o·[h_(t−1), x_t] + b_o) (output gate)

h_t = o_t ⊙ tanh(C_t) (hidden state)

(⊙ = element-wise multiplication.) Old memory is partially forgotten, new candidate memory is partially added.

Q9. How does LSTM address vanishing gradients and long-term dependencies? ★

Answer

The key is the **additive** cell-state update: C_t = f_t ⊙ C_(t−1) + i_t ⊙ C̃_t.

- The gradient flowing back through the cell state is multiplied by the forget gate f_t, not by repeated weight matrices and tanh derivatives.
- When f_t ≈ 1, the gradient passes almost unchanged across many time steps (the "constant error carousel"), so it does not vanish.
- The gates **learn** what to keep for a long time (e.g. the subject of a sentence) and what to forget, so important information can be carried across long sequences.

LSTMs reduce, but do not completely eliminate, the problem for very long sequences (hundreds+ of steps).

Q10. What is the difference between the hidden state and the cell state?

Answer

- **Cell state C_t** – internal **long-term memory**; passed only to the next time step; not directly exposed as output; updated additively by forget/input gates.
- **Hidden state h_t** – **short-term/working memory** and the **output** at each time step; passed to the next time step **and** to the next layer/output layer; it is a filtered version of the cell state: h_t = o_t ⊙ tanh(C_t).

Q11. What is a GRU? Explain its gates (update, reset). How does it differ from LSTM? ★

Answer

A **GRU (Gated Recurrent Unit)** is a simplified LSTM with **two gates and only one state** (hidden state, no separate cell state).

- **Update gate z_t** – how much of the past to keep vs how much new information to add (combines LSTM's forget + input gates).
- **Reset gate r_t** – how much of the previous hidden state to use when computing the new candidate.

Equations: $z_t = \sigma(W_z \cdot [h_{(t-1)}, x_t])$; $r_t = \sigma(W_r \cdot [h_{(t-1)}, x_t])$; $\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{(t-1)}, x_t])$; $h_t = (1 - z_t) \odot h_{(t-1)} + z_t \odot \tilde{h}_t$

Differences from LSTM: 2 gates vs 3, no cell state, ~25% fewer parameters, faster to train, often similar accuracy.

Q12. How would you decide between LSTM and GRU?

Answer

- **Choose GRU** when data is smaller, you need faster training/inference, or lower memory (mobile/edge), and sequences are moderate length.
- **Choose LSTM** when sequences are long, the dataset is large, and you need more expressive memory control (separate cell state + output gate).
- In practice the performance is often similar – **try both** and pick by validation score. For most modern NLP tasks, Transformers are preferred over both.

Q13. Advantages and disadvantages of RNN, LSTM and GRU.

Answer

RNN – Pros: simple, few parameters, handles sequences. Cons: vanishing/exploding gradients, short memory, sequential (slow) processing.

LSTM – Pros: learns long-term dependencies, solves most of the vanishing gradient problem, robust. Cons: more parameters (4× RNN), slower, still sequential.

GRU – Pros: fewer parameters than LSTM (3× RNN), faster, good on small datasets. Cons: slightly less expressive than LSTM on very long/complex sequences; still sequential.

All three: cannot be parallelized across time steps and struggle with very long contexts compared with Transformers.

Q14. Why did Transformers replace RNNs/LSTMs for most NLP tasks? ★

Answer

- **Parallelization:** RNNs process tokens one by one; Transformers process the whole sequence at once → much faster training on GPUs.
- **Long-range dependencies:** self-attention connects any two tokens directly (path length 1), whereas RNNs must pass information through many steps.
- **No vanishing gradient through time.**
- **Scalability:** Transformers scale to billions of parameters and huge datasets; this enabled pre-training (BERT, GPT) and transfer learning.
- **Better results** on almost every NLP benchmark.

(RNNs are still used in small/edge or streaming settings, and new recurrent-style models like Mamba/state-space models are being explored for long contexts.)

Q15. What does return_sequences=True do in an LSTM layer and when do you need it?

Answer

- `return_sequences=False` (default) – the LSTM returns **only the last hidden state**, shape (batch, units).
- `return_sequences=True` – returns the **hidden state at every time step**, shape (batch, timesteps, units).

Set it to True when:

- **Stacking** LSTM layers – every LSTM except the last must return sequences to feed the next LSTM.
- **Sequence labelling** (NER, POS tagging) – one output per token (use with TimeDistributed(Dense)).
- Feeding an **attention layer** or seq2seq encoder outputs.

(`return_state=True` additionally returns the final hidden and cell states.)

Q16. How do you calculate the number of trainable parameters in RNN, LSTM and GRU layers?

Answer

Let n = number of units, m = input dimension.

- **SimpleRNN:** $n \cdot (n + m) + n = n^2 + nm + n$
- **LSTM:** $4 \times [n \cdot (n + m) + n]$ (4 sets: forget, input, candidate, output)
- **GRU:** $3 \times [n \cdot (n + m) + n]$ in the original version. **Keras default (reset_after=True) uses $3 \times [n \cdot (n + m) + 2n]$** (two bias vectors).

Example (m = 100 embedding dim, n = 64 units):

- SimpleRNN = $64 \cdot (64+100) + 64 = 10{,}560$
- LSTM = $4 \times 10{,}560 = 42{,}240$
- GRU (Keras) = $3 \times (64 \cdot 164 + 128) = 31{,}872$

Q17. What is padding? What is truncating? Explain pre vs post padding and the related hyperparameters (maxlen, padding, truncating).

Answer

- **Padding** – adding a special value (usually 0) to shorter sequences so that all sequences in a batch have equal length.
- **Truncating** – cutting sequences that are longer than the maximum length.
- **Pre vs post:** `padding="pre"` adds zeros at the **start**, `"post"` at the **end**. Same for `truncating` – remove tokens from the start or the end. Pre-padding is the Keras default and often works better for RNNs (the real tokens are closest to the final hidden state).

Hyperparameters: `maxlen` (choose e.g. the 90–95th percentile of lengths), `padding`, `truncating`, `value` (pad value), `dtype`. Too small maxlen loses information; too large wastes compute.

Q18. What is the Embedding layer in Keras? How many parameters does it have?

Answer

The **Embedding layer** is a trainable lookup table that converts integer token IDs into dense vectors. Input shape (batch, seq_len) → output (batch, seq_len, output_dim). Weights are learned during training, or can be initialized with pre-trained vectors (Word2Vec/GloVe) and frozen.

Parameters = input_dim (vocabulary size) × output_dim (embedding size). Example: vocab 10,000 × 128 dims = **1,280,000** parameters (no bias).

```
layers.Embedding(input_dim=10000, output_dim=128, mask_zero=True)
```

Q19. Explain dropout and recurrent dropout.

Answer

- **Dropout** – during training, randomly sets a fraction p of a layer's inputs/outputs to 0 (and scales the rest by 1/(1−p)), so the network cannot rely on specific neurons → reduces overfitting. Turned off during inference.
- **Recurrent dropout** – applies dropout to the **recurrent connections** (h_(t−1) → h_t) in RNN/LSTM/GRU, using the **same mask at every time step** (variational dropout) so memory is not destroyed randomly over time.

```
layers.LSTM(64, dropout=0.2, recurrent_dropout=0.2)
```

(Note: recurrent_dropout > 0 disables the fast cuDNN kernel.)

Q20. Write code for padding sequences. [CODE]

Answer

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

texts = ["I love NLP", "Deep learning is amazing and powerful", "Hello"]
tok = Tokenizer(num_words=1000, oov_token="<OOV>")
tok.fit_on_texts(texts)
seqs = tok.texts_to_sequences(texts)
print(seqs)          # [[2, 3, 4], [5, 6, 7, 8, 9, 10], [11]]   (index 1 = <OOV>)

padded = pad_sequences(seqs, maxlen=5, padding="post", truncating="post", value=0)
print(padded)
# [[ 2  3  4  0  0]
#  [ 5  6  7  8  9]
#  [11  0  0  0  0]]       (2nd sequence truncated at the end)

# PyTorch equivalent:
# torch.nn.utils.rnn.pad_sequence(list_of_tensors, batch_first=True, padding_value=0)
```

Q21. Build networks with (a) SimpleRNN, (b) LSTM, (c) GRU and (d) Bidirectional layer for sentiment classification. [CODE]

Answer

```
import tensorflow as tf
from tensorflow.keras import layers, models

VOCAB, EMB, MAXLEN = 10000, 128, 200
(x_tr, y_tr), (x_te, y_te) = tf.keras.datasets.imdb.load_data(num_words=VOCAB)
x_tr = tf.keras.preprocessing.sequence.pad_sequences(x_tr, maxlen=MAXLEN)
x_te = tf.keras.preprocessing.sequence.pad_sequences(x_te, maxlen=MAXLEN)

def build(rnn_layer):
    m = models.Sequential([
        layers.Input(shape=(MAXLEN,)),
        layers.Embedding(VOCAB, EMB, mask_zero=True),
        rnn_layer,
        layers.Dense(32, activation="relu"),
        layers.Dense(1, activation="sigmoid")])
    m.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])
    return m

rnn   = build(layers.SimpleRNN(64))                          # (a)
lstm  = build(layers.LSTM(64, dropout=0.2))                  # (b)
gru   = build(layers.GRU(64))                                # (c)
bilstm = build(layers.Bidirectional(layers.LSTM(64)))        # (d)

# Stacked version: layers.LSTM(64, return_sequences=True), layers.LSTM(32)
for name, m in [("RNN", rnn), ("LSTM", lstm), ("GRU", gru), ("BiLSTM", bilstm)]:
    m.fit(x_tr, y_tr, epochs=2, batch_size=64, validation_split=0.2, verbose=0)
    print(name, m.evaluate(x_te, y_te, verbose=0))
```

Expected: LSTM/GRU/BiLSTM ≈ 85–88% accuracy on IMDB; SimpleRNN lower.

Q22. How can you reduce RNN training problems without moving directly to LSTM/GRU? · Asked: NLP interview rounds

Answer

- **Gradient clipping** – cap the gradient norm (e.g. 1.0) to stop exploding gradients.
- **Better activations and initialization** – ReLU with identity/orthogonal initialization of W_hh (IRNN) keeps gradients stable.
- **Truncated BPTT** – backpropagate only through the last k steps.
- **Shorter sequences** – truncate/split long texts, or summarize.
- **Normalization** – layer normalization inside the recurrent cell.
- **Skip/residual connections** across time or layers.
- **Lower learning rate**, adaptive optimizers (Adam/RMSProp).
- Regularization: dropout / recurrent dropout.

If long-range dependencies still matter, then move to LSTM/GRU or Transformers.