Deep Learning (BEV033DLE)
Lecture 13 Recurrent Neural Networks & Transformer Networks

Czech Technical University in Prague

- Recurrent models
- Gated recurrent units, GRU and LSTM networks
- Transformer networks & GPT language models

Recurrent models in a nutshell

- input sequence $x = (x_1, \dots, x_t, \dots, x_T)$, $x_t \in \mathbb{R}^n$, output sequence $y = (y_1, \dots, y_T)$, $y_t \in \mathcal{Y}$ and sequence of hidden states $h = (h_1, \dots, h_T)$, $h_t \in \mathbb{R}^d$.
- recurrent (dynamic) system with outputs

$$\begin{aligned} h_t &= f(x_t, h_{t-1}, w) \\ y_t &= g(h_t, v) \end{aligned}$$

  where w and v are parameters. The model defines sequence-to-sequence mappings $h = F_w(x)$ and $y = G_v(h)$.
- loss function $\ell(y, y')$, often locally additive $\ell(y, y') = \sum_t \ell_t(y_t, y'_t)$

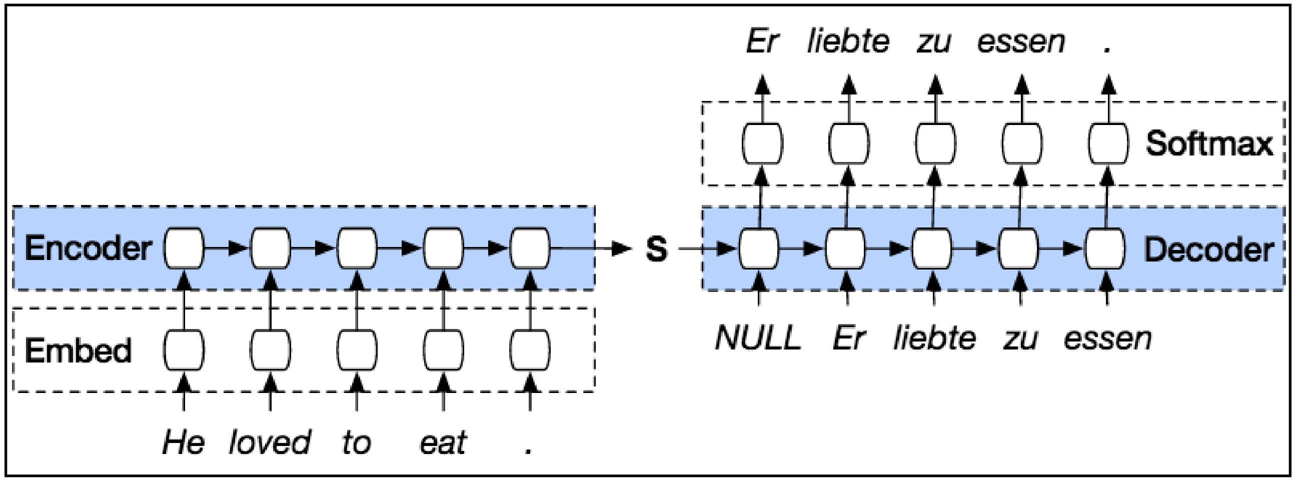
Training goal: given training data $\mathcal{T} = \{(x^j, y^j) \mid j = 1, \dots, m\}$, learn the model parameters w, v by solving

$$\frac{1}{m} \sum_{(x,y) \in \mathcal{T}} \ell\big(y, (G_v \circ F_w)(x)\big) \to \min_{w,v}$$

Recurrent networks

Incarnations of recurrent models and related tasks

- Deep neural network for classification with additional feedback connections: x_t - constant input, y_t - output of the network, h_t -states of all hidden layers. The loss function depends on the last output y_T only.
- "infinite state automata": the output space is sufficient for keeping the history, thus h and y can be identified, i.e. $y_t = f(x_t, y_{t-1}, w)$.
  Example: Earth observation, land-cover type monitoring x_t - sequence of spectral satellite measurements, y_t - sequence of states (e.g. coniferous forest, broadleaf forest, clearcut, bark beetle degradation etc.)
- general sequence-to-sequence segmentation: hidden states h_t are needed for keeping track of longer past and are latent.
  Example: NLP translation:



Learning RNNs special case: infinite state automata

Learning RNNs is particularly simple in the case that

- h and y can be identified, i.e. $y_t = f(x_t, y_{t-1}, w)$ and
- the loss is locally additive $\sum_t \ell(y_t, y'_t)$

Split each sequence $(x, y) \in \mathcal{T}^m$ into triplets (y_{t-1}, x_t, y_t) and train f from

$$\frac{1}{m} \sum_{(x,y)\in\mathcal{T}} \sum_t \ell\big(y_t, f(x_t, y_{t-1}, w)\big) \to \min_w$$

Neither forward nor backward propagation through the sequence are needed.

Learning RNNs general case: backpropagation through time

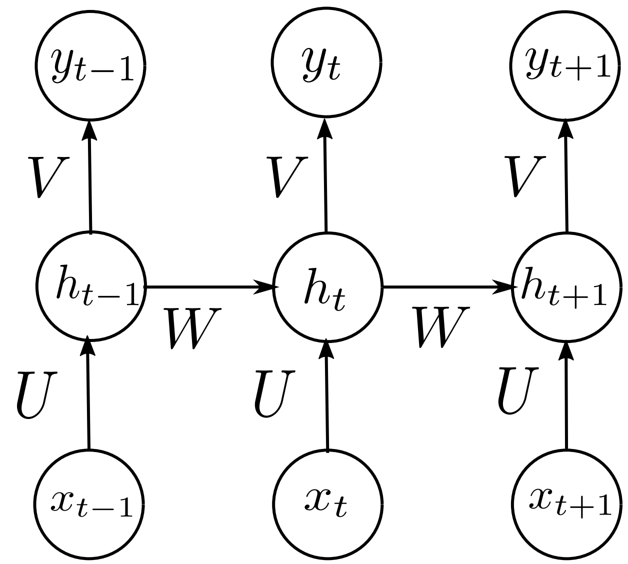
Assumptions:

$$h_t = f(x_t, h_{t-1}, w)$$
$$y_t = g(h_t, v)$$

The mappings f and g are implemented by neural networks and are differentiable w.r.t. their inputs and parameters. The loss function $\ell(y, y')$ is differentiable.

Example 1. Both mappings f and g are implemented by one layer networks

$$a_t = W h_{t-1} + U x_t + b \qquad h_t = \tanh(a_t)$$
$$o_t = V h_t + c \qquad y_t = \mathrm{softmax}(o_t)$$



Learning RNNs general case: backpropagation through time

Computing the gradients: Unroll the network in time and apply backpropagation

Let us consider the loss for a single example (x, y^*) from the training data.

Computing the gradient w.r.t. v is easy (see Slide 4.). Let us consider the gradient w.r.t. w

$$\partial_w \ell(y, y^*) = \sum_{t=1}^{T} \partial_w \ell(y_t, y_t^*) = \sum_{t=1}^{T} \partial_{y_t} \ell(y_t, y_t^*) \, \partial_{h_t} g(h_t, v) \, \partial_w h_t$$

The first two derivatives are simple. For the last one we have the recurrent expression

$$\partial_w h_t = \partial_w f(x_t, h_{t-1}, w) + \partial_{h_{t-1}} f(x_t, h_{t-1}, w) \, \partial_w h_{t-1}$$

This gives

$$\partial_w h_t = \partial_w f(x_t, h_{t-1}, w) + \sum_{i=1}^{t-1} \Big[\prod_{j=i+1}^{t} \partial_{h_{j-1}} f(x_j, h_{j-1}, w) \Big] \partial_w f(x_i, h_{i-1}, w)$$

Learning RNNs general case: backpropagation through time

Problems:

- backpropagation through time is computationally expensive
- Exploding/vanishing gradients: consider for simplicity the linear recurrence $h_t = W h_{t-1}$. For τ steps we get $h_\tau = W^\tau h_0$. Suppose that we can write $W = U^{-1} \Lambda U$, where Λ is diagonal. We get
$$h_\tau = U^{-1} \Lambda^\tau U h_0 .$$
Eigenvalues with magnitude less than one will decay and eigenvalues with magnitude greater than one will explode.
- We can not apply batch normalisation as simple remedy.
- We want the following model ability: events long in the past can trigger changes in conjunction with current measurements.

Possible solutions: skip connections? designate special nodes in h_t for keeping record of events long in the past?

RNNs with gated recurrent units

- Long short term memory, Schmidhuber, 1997
- Gated recurrent unit, Cho et al., 2014

Gated recurrent unit (simplified):

A cell consisting of a recurrent unit h_t and a gate unit $u_t \in [0, 1]$

$$
\begin{aligned}
h_t &= u_{t-1} h_{t-1} + [1 - u_{t-1}] f(x_t, h_{t-1}, w) \\
u_t &= \mathrm{S}(x_t, h_t, v)
\end{aligned}
$$

The gate unit u_t has sigmoid nonlinearity and “decides” whether to copy h_t from h_{t-1} or to apply the recurrence with f.

RNNs with gated recurrent units

Gated recurrent unit (general):

- h is a state vector
- u is a vector of “update” gates
- r is a vector of “reset” gates

The update equations are

$$h_t = u_{t-1} \odot h_{t-1} + [1 - u_{t-1}] \odot \mathrm{S}\Big(U x_{t-1} + W r_{t-1} \odot h_{t-1}\Big)$$

where $\odot$ denotes the element-wise product of vectors. The gate unit outputs are given by

$$u_t = \mathrm{S}\big(U^u x_t + W^u h_t\big)$$
$$r_t = \mathrm{S}\big(U^r x_t + W^r h_t\big)$$

LSTM cells are more complicated – they have separate “forget” and “update” gates.

Main weakness of LSTM & GRU: No explicit modelling of long and short range dependencies

Transformer Networks

Let us consider the task of **next token prediction** for NLP

Task: Given a corpus of tokens $\mathcal{X} = (x_1, x_2, \ldots, x_n)$, train a network for predicting the next token x_i given the context window $\mathcal{X}_i = (x_{i-k}, \ldots, x_{i-1})$.

$$L(\mathcal{X}, \theta) = \sum_i \log p(x_i \,|\, x_{i-k}, \ldots, x_{i-1}\,; \theta) \to \max_\theta$$

Language Model: Generative Pre-trained Transformer (GPT)

1. Vector embedding of tokens with position information and trainable parameter W: $y_i = \Gamma(x_i, i, W) \in \mathbb{R}^m$
2. For each i: $h_0 = \mathcal{Y}_i$
3. Apply transformer blocks: $h_l = \textit{transformer\_block}(h_{l-1})$
4. Predict x_i by: $p(x \,|\, \mathcal{X}_i) = \textit{softmax}(V h_L)$ with trainable parameter V.

Transformer Networks

Transformer (decoder):

1. **Self-Attention** with learnable parameters W^k, W^q, W^v
   - Key: $\phi(y_i, W^k) \in \mathbb{R}^m$
   - Query: $\psi(y_i, W^q) \in \mathbb{R}^m$
   - Value: $\chi(y_i, W^v) \in \mathbb{R}^m$

Output: weighted sum of value vectors + layer normalisation (not shown)

$$z_i = \sum_{j=i-k}^{i} \mathit{softmax}\Big(\psi^T(y_i)\phi(y_j)\Big)\chi(y_i)$$

2. **Feed forward network:** $h_i = F(z_i, W)$ + layer normalisation (not shown)
   - The attention sub-layer usually consists of several parallel attention heads
   - Both sub-layers have residual skip connections.
   - Transformer outputs are differentiable in all parameters

Transformer Networks

A GPT model can be used for various downstream tasks like

- natural language inference
- question answering
- semantic similarity

This can be achieved by adding a linear layer and fine tuning or, even simpler, with zero-shot or few-shot inference.

The downstream task performance of the model improves with the size of the training corpus and with the number of epochs in pre-training.