Machine Learning

- ML is basically function approximation
- function: data (features) $\rightarrow$ labels
- function shape:
  - this is where different ML algorithms differ
  - neural nets: compound non-linear functions
- training/learning = adjusting function parameters to minimize error (see next week)
  - supervised learning = based on data + labels given in advance
  - reinforcement learning = based on exploration & rewards given online
Typical machine learning problems in NLP

- **regression**
  - many inputs, 1 float output

- **classification**
  - many inputs, 1 categorial output (k classes)

- **sequence labelling**
  - sequence of inputs, label each (~ repeated classification)
  - 1-to-1 input to output

- **ranking**
  - multiple inputs, choose best one (~ diff regression)

- **sequence prediction** (autoregressive generation)
  - some inputs (sequence/something else)
  - generate outputs, use previous output in predicting next one
Neural networks

• **Non-linear functions**, composed of basic building blocks
  • stacked into **layers**

• Layers are made of **activation functions**:
  • linear functions (~basic, default)
  • nonlinearities – sigmoid, tanh, ReLU
  • softmax – probability estimates:
    \[
    \text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}
    \]

• Fully differentiable – training by **gradient descent**
  • network output incurs loss/cost
  • gradients **backpropagated** from loss to all parameters
    (composite function differentiation)

https://medium.com/@shrutijadon104776/survey-on-activation-functions-for-deep-learning-9689331ba092
Layers visualization

- [https://playground.tensorflow.org/](https://playground.tensorflow.org/)
  - 2 numeric features (=2 input variables) → binary classification (=1 output, 2 classes)
    - easiest case, but you can see the internals
    - more complex input features →
  - **feed-forward = fully connected = multi-layer perceptron** here
    - easiest case: connect everything & let the network figure it out
    - nice but gets too large very quickly, not good for variable-sized inputs
  - added layers & power to distinguish different classes
    - fits the training data Y/N?
  - different activation functions
    - without them, it’s just linear – no matter how many layers!
  - best NN conceptualization – pipeline / flow (computational graph)
    - data flows through individual layers, gets changed
    - corresponds to a math formula, but flow graph can be easier to read
Feature representation

• technically can be anything, as long as it’s meaningful
  • the network will learn to assign meaning/values itself

• 1-hot/binary
  • words – numbered vocabulary
    • bigrams, n-grams, positional…
  • other features – especially handcrafted
    • word classes
    • various word combinations
    • outputs of other classifiers (sentiment, part-of-speech…)
    • is capitalized/is loud?

• numeric (floats)
  • best for continuous inputs: vision, audio
    • raw pixels, MFCCs…

• vectors (embeddings) →
Embeddings

- distributed (word) representation
  - each word = a vector of floats
  - basically an easy conversion of 1-hot → numeric
  - a dictionary of trainable features
- part of network parameters – trained
  a) random initialization
  b) pretraining
- the network learns which words are used similarly
  - they end up having close embedding values
  - embeddings end up different with different tasks & data & settings
- embedding size: ~100s-1000
- vocab size: ~50-100k

http://ruder.io/word-embeddings-2017/
Pretrained Word Embeddings

• **Word2Vec**
  
  • Continuous Bag-of-Words (CBOW) (~ “masked LM”)
    
    • predict a word, given ±k words window
    • disregarding word order within the window
  
  • Skip-gram: reverse
    
    • given a word, predict its ±k word window
    • closer words = higher weight in training

• **GloVe**
  
  • optimized directly from corpus co-occurrences (= \(w_1\) close to \(w_2\))
    
    • target: \(e_1 \cdot e_2 = \log(#\text{co-occurrences})\)
      
      • number weighted by distance, weighted down for low totals
    
    • trained by minimizing reconstruction loss on a co-occurrence matrix

**References**

- Mikolov et al., 2013
  
  http://arxiv.org/abs/1301.3781

- Pennington et al., 2014
  
  http://aclweb.org/anthology/D14-1162

- https://geekyisawesome.blogspot.com/2017/03/word-embeddings-how-word2vec-and-glove.html

Word Embeddings

- Vocabulary is unlimited, embedding matrix isn’t
  - + the bigger the embedding matrix, the slower your models
- Special **out-of-vocabulary token** `<unk>`
  - “default” / older option
  - all words not found in vocabulary are assigned this entry
  - can be trained using some rare words in the data
  - problem for generation – you don’t want these on the output
- Using limited sets
  - **characters** – very small set
    - works, but makes for very long sequences
      (20 words ~ 80-100 characters)
    - slower, might be less accurate
  - **subwords** – compromise

Subwords

- group of characters that:
  - make shorter sequences than using individual characters
  - cover everything

- **byte-pair encoding**
  - start from individual characters
  - iteratively merge most frequent bigram, until you get desired # of subwords
  - *sub@@ word* – the @@ marks “no space after”

- **SentencePiece** – don’t pre-tokenize
  - criterium: likelihood of joined vs. separate
  - *sub word*_ – the _ marks a space

- 20-50k subwords for 1 language
  - ~250k subwords to cover them all

(Sennrich et al., 2016)
https://www.aclweb.org/anthology/P16-1162/

https://github.com/google/sentencepiece
https://blog.floydhub.com/tokenization-nlp/
https://d2l.ai/chapter_natural-language-processing-pretraining/subword-embedding.html
Convolutional Networks

- Designed for computer vision – inspired by human vision
  - works for language in 1D, too!
- less parameters than fully connected – **filter/kernel**
- Apply (multiple) filter(s) repeatedly over the input
  - element-wise multiply window of input x filter
  - sum + apply non-linearity (ReLU) to result
  - => produce 1 element of output
  - can have more dimensions (~“set of filters”)
- **Stride** – how many steps to skip
  - less overlap, reducing output dimension
- **Pooling** – no filter, pre-set operation
  - **maximum**/average on each window
  - typical CNN architecture alternates convolution & pooling

https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2
Recurrent Neural Networks

- Identical layers with shared parameters (**cells**)
  - ~ the same layer is applied multiple times, taking its own outputs as input
    - ~ same number of layers as there are tokens
    - output = hidden state – fed to the next step
  - additional input – next token features
  
- **basic RNN**: linear + tanh
  - tanh: squashes everything to \([-1,1]\]
    - good for repeated application
  - very simple structure
  - numeric problem: vanishing gradients
    - training updates get too small
    - can’t hold long sequences well

---


![Diagram of Recurrent Neural Networks](https://medium.com/@saurabh.rathor092/simple-rnn-vs-gru-vs-lstm-difference-lies-in-more-flexible-control-5f33e07b1e57)
LSTMs & GRUs

- **GRU, LSTM**: more complex, to make training more stable
  - “gates” to keep old values
  - \( \sigma \sim [0,1] \) decisions:
    - forget stuff from previous?
    - take input into account?
    - put stuff onto output?
    - over individual dimensions (e.g. input has 100 dims, forget gate forgets dims 1-3 & 4-25)
  - all based on current input & state
- LSTM is older & more complex
- GRU almost as good but faster
- both slower than base RNN
- both handle long recurrences

Encoder-Decoder Networks (Sequence-to-sequence)

- Default RNN paradigm for sequences/structure prediction
  - **encoder** RNN: encodes the input token-by-token into hidden states $h_t$
    - next step: last hidden state + next token as input
  - **decoder** RNN: constructs the output token-by-token
    - initialized by last encoder hidden state
    - output: hidden state & softmax over output vocabulary + argmax
    - next step: last hidden state + last generated token as input
- LSTM/GRU cells over vectors of ~ embedding size
- used in MT, dialogue, parsing…
  - more complex structures linearized to sequences

$h_0 = 0$
$h_t = \text{cell}(x_t, h_{t-1})$

$s_0 = h_T$
$p(y_t | y_1, ... y_{t-1}, x) = \text{softmax}(s_t)$
$s_t = \text{cell}(y_{t-1}, s_{t-1})$

https://medium.com/syncedreview/a-brief-overview-of-attention-mechanism-13c578ba9129
Attention

- Encoder-decoder is too crude for complex sequences
  - the whole input is crammed into a fixed-size vector (last hidden state)
- **Attention** = “memory” of **all encoder** hidden states
  - weighted combination, re-weighted for every decoder step
  - can focus on currently important part of input
  - fed into decoder inputs + decoder softmax layer
- **Self-attention** – over **previous decoder steps**
  - increases consistency when generating long sequences

https://skymind.ai/wiki/attention-mechanism-memory-network
Seq2seq RNNs with Attention

Token representation: **embeddings** = vectors of ~100-1000 numbers

Source "word" embeddings

Vocabulary is numbered

Encoder outputs - “hidden states” (=again, vectors of numbers)

Attention = weighted combination (weights different for each step)

Probability distribution over the whole vocabulary

Target word embeddings

Cells: identical (compound) neural layers

Input: prev. output + token embedding

Bahdanau & Luong Attention

- different combination with decoder state
  - Bahdanau: use on input to decoder cell
  - Luong: modify final decoder state
- different weights computation
- both work well – exact formula not important

**attention value = context vector**

\[ c_t = \sum_{i=1}^{n} \alpha_{ti} h_i \]

**attention weights = alignment model**

Bahdanau:

\[ \alpha_{ti} = \text{softmax}(v_\alpha \cdot \tanh(W_\alpha \cdot s_{t-1} + U_\alpha \cdot h_i)) \]

Luong:

\[ \alpha_{ti} = \text{softmax}(h_i^T \cdot s_t) \]

---

(Bahdanau et al., 2015)
http://arxiv.org/abs/1409.0473
(Luong et al., 2015)
http://arxiv.org/abs/1508.04025

---

http://cnyah.com/2017/08/01/attention-variants/
Transformer

(Waswani et al., 2017)  
https://arxiv.org/abs/1706.03762

• getting rid of (encoder) recurrences
  • making it faster to train, allowing bigger nets
  • replace everything with attention + feed-forward networks
  • ⇒ needs more layers
  • ⇒ needs to encode positions

• positional encoding
  • adding position-dependent patterns to the input

• attention – dot-product (Luong style)
  • scaled by $\frac{1}{\sqrt{\#\text{dims}}}$ (so values don’t get too big)
  • **more heads** ( attentions in parallel)
    – focus on multiple inputs
Transformer

feed-forward (fully connected) network
- ReLU activations
- tricks for better training

attention over all of input

encoder

no recurrent connections

decoder

attention over all of input & output generated so far (self-attention)

(Vaswani et al., 2017) http://arxiv.org/abs/1706.03762
Pretrained Language Models

• Beyond pretrained word embeddings
  • reflects different word meanings in sentence context (~contextual embeddings)
  • used as input to added layers on top / base for model fine-tuning (next week)

• LSTM-based: ELMo (trained on language modelling)
  • weighted sum of static word embeddings & LSTM outputs

• Transformer encoders: BERT, RoBERTa…
  • for classification, sequence tagging
  • any Transformer layer used (typically the last one)

• Transformer decoders: GPT-2, GPT-3…
  • for generation, language modelling
  • input: force-decoding

• Transformer encoder-decoders: BART, T5…
  • same as ↑, explicit input

https://github.com/jessevig/bertviz
Summary

• ML as a function mapping in $\rightarrow$ out
  • input features – 1-hot, numeric, embeddings
    • pretrained embeddings
  • function: layers ~ pipeline, data flows through (= complicated function)
  • outputs: classification (category), regression (float)
    • structured prediction – sequence tagging, ranking, generation

• Neural networks (~function shapes)
  • feed-forward/fully connected
  • CNNs (filters, pooling)
  • RNNs (LSTMs, GRUs)
  • encoder-decoder (seq2seq)
  • attention, Transformer (positional encoding & feed-forward & attention)
    • pretrained models

• Next week: how to train this stuff
Thanks

Contact us:
https://ufaldsg.slack.com/
{odusek,hudecek,kasner}@ufal.mff.cuni.cz
Zoom/Skype/Troja

Get the slides here:
http://ufal.cz/npfl099

References/Further:

Neural nets tutorials:
• https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0
• https://minitorch.github.io/index.html
• https://objax.readthedocs.io/en/latest/

No lab today
Next week: lecture & lab
Monday 12:20

NPFL099 L3 2022