Hochschule Hof
University of Applied Sciences

Applied Deep Learning

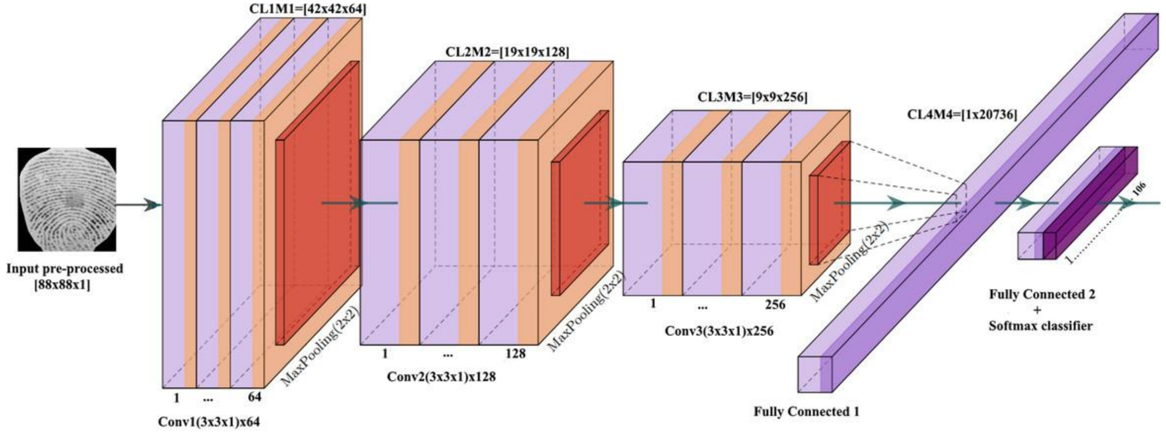
Transfer Learning and Memory

Targets

- Learn the purpose of transfer learning
- Gain an overview of the techniques used
- Learn different concepts of memory for neural nets

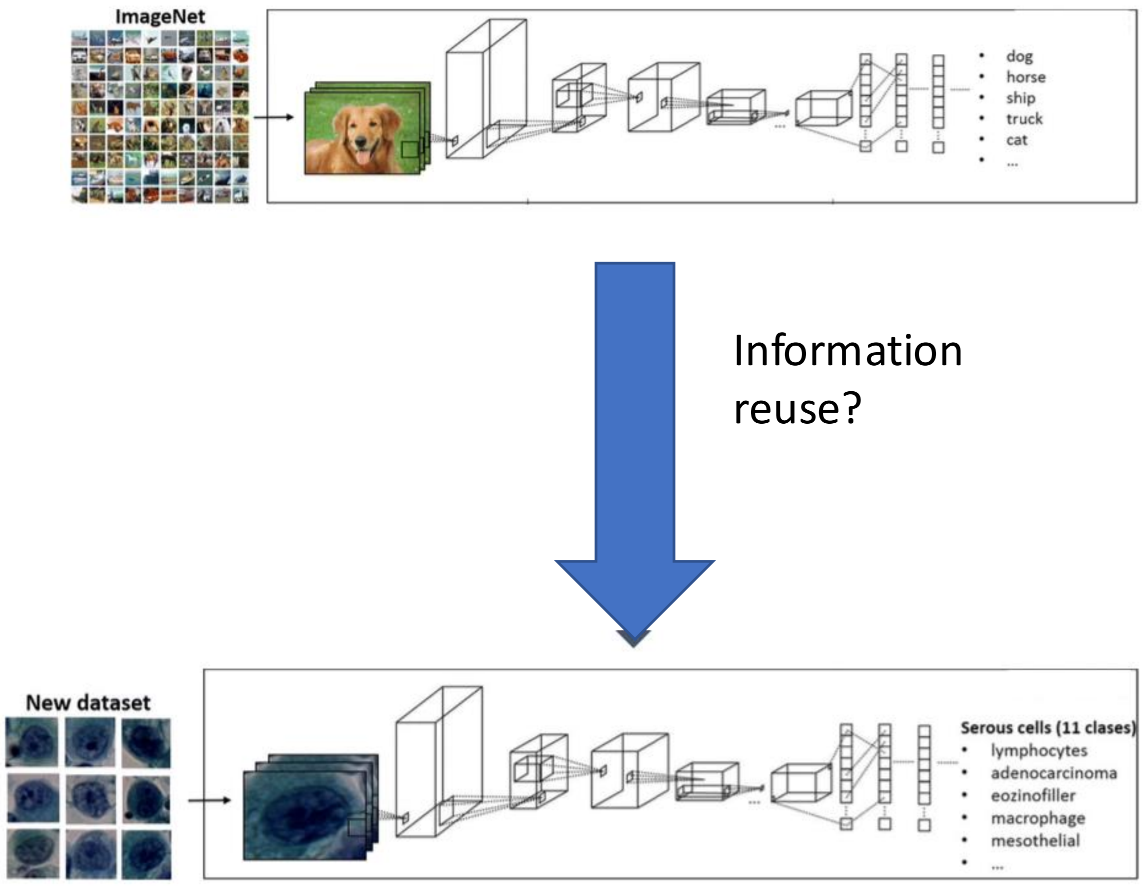
Short repetition

- Convolutional Neural Network
  - One input layer, several hidden layers (CONV, POOL, FC), one output layer
  - Process input volume
  - Typically automatic feature learning
  - Typically usable for image data but not for sequential data



Motivation

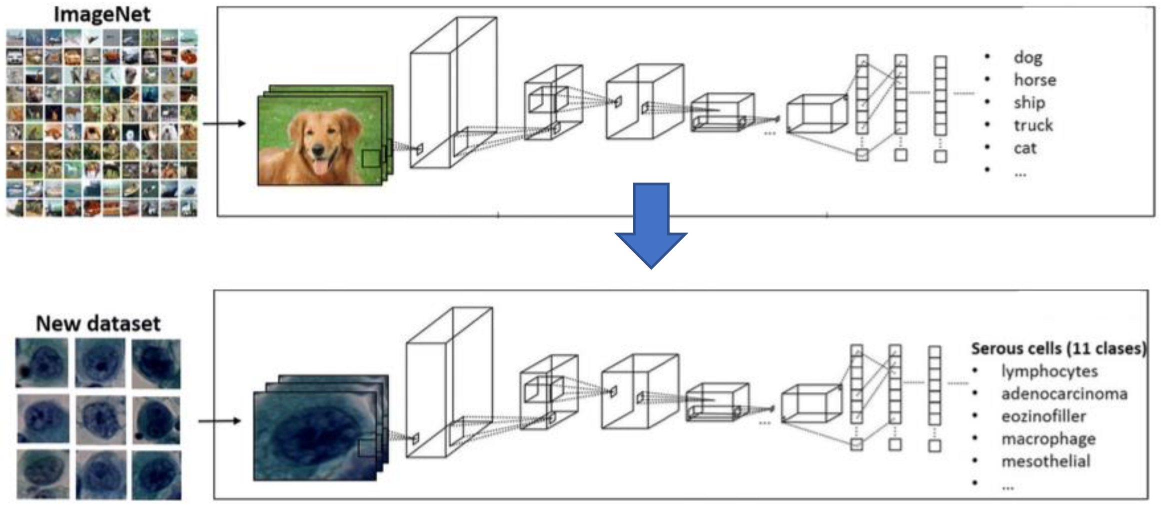
- For image-based tasks, CNNs are mostly used
- High accuracy is usually achieved with deep NN with very many parameters
- These are trained on large generic datasets, such as ImageNet (14 million images)
- Often not possible for our tasks

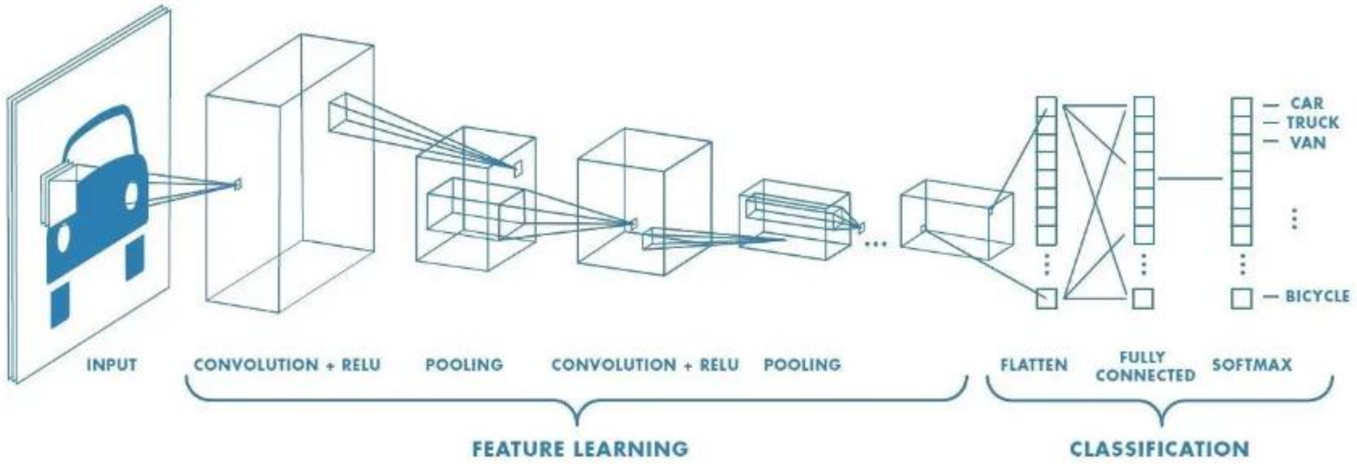


Information reuse?

Motivation

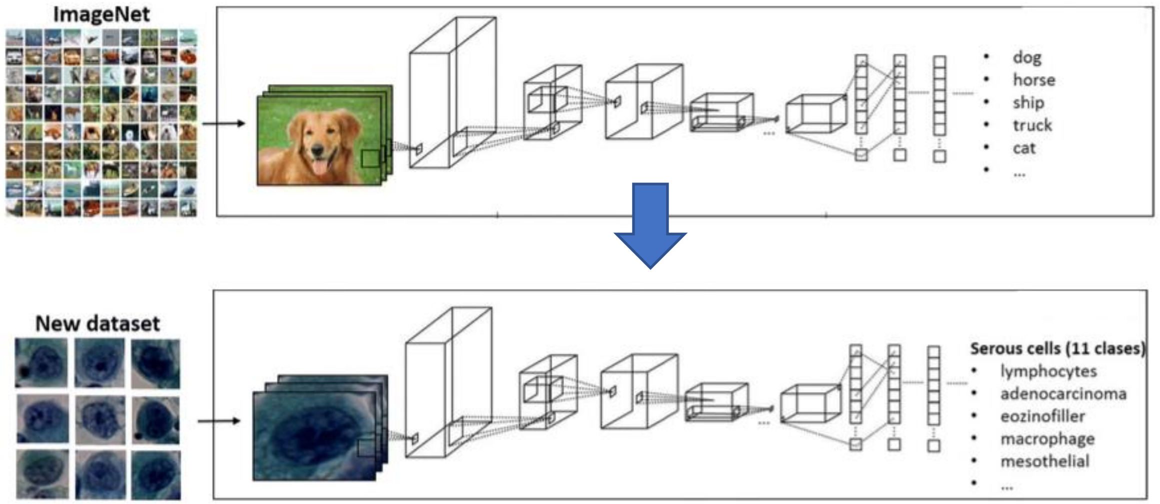
- Instead of training a new NN completely, only the problem-specific parts can be trained
- Which parts of an NN are problem-specific?



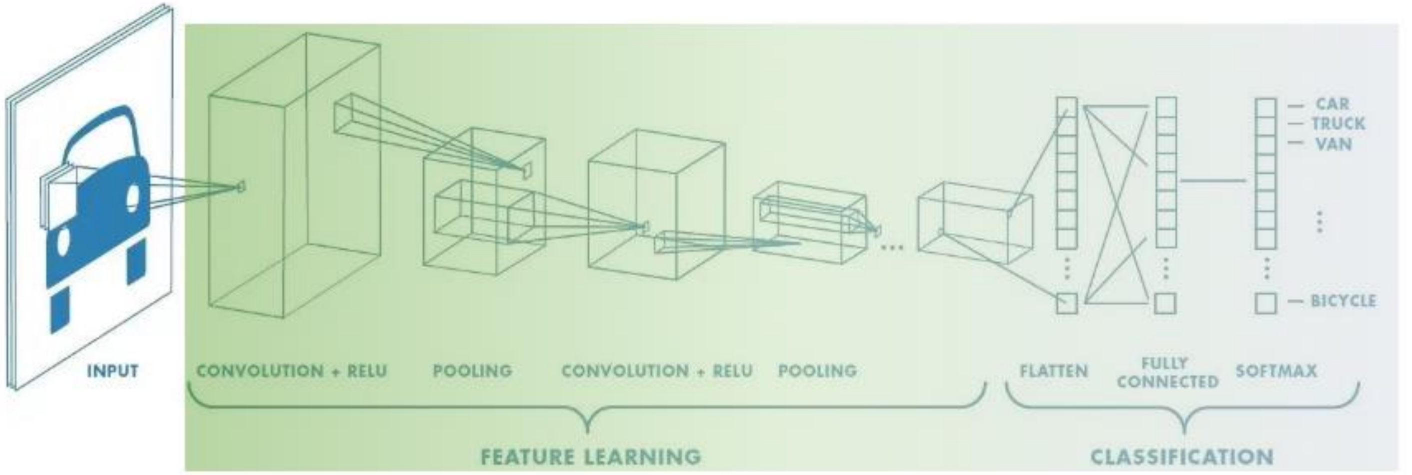


Motivation

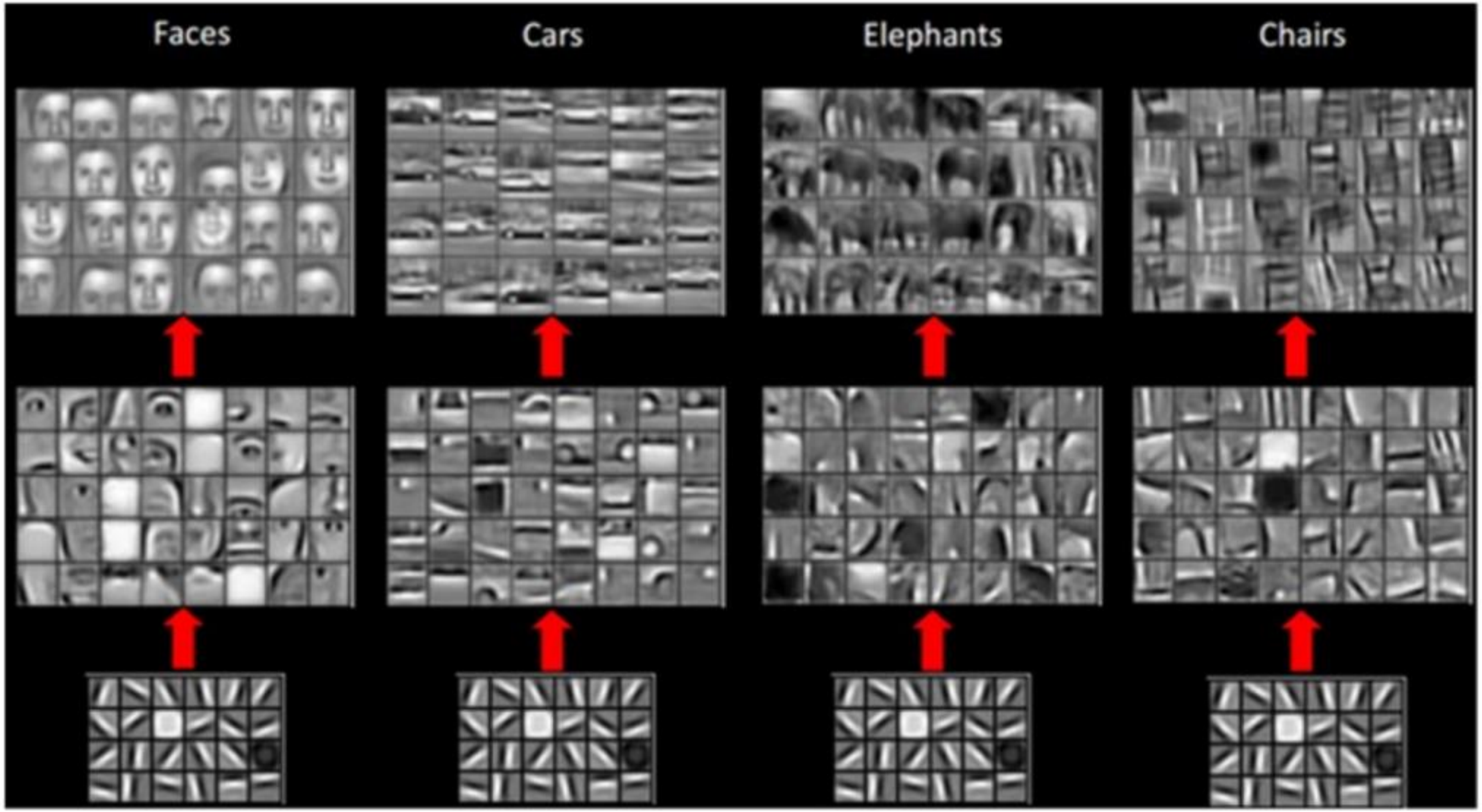
- Instead of training a new NN completely, only the problem-specific parts can be trained
- Which parts of an NN are problem-specific?



🟩 problem-independend
⬜ problem-dependend

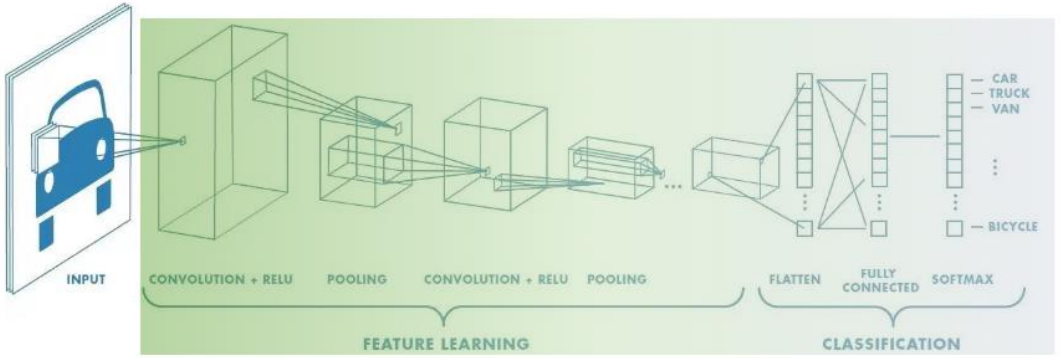


Finetuning



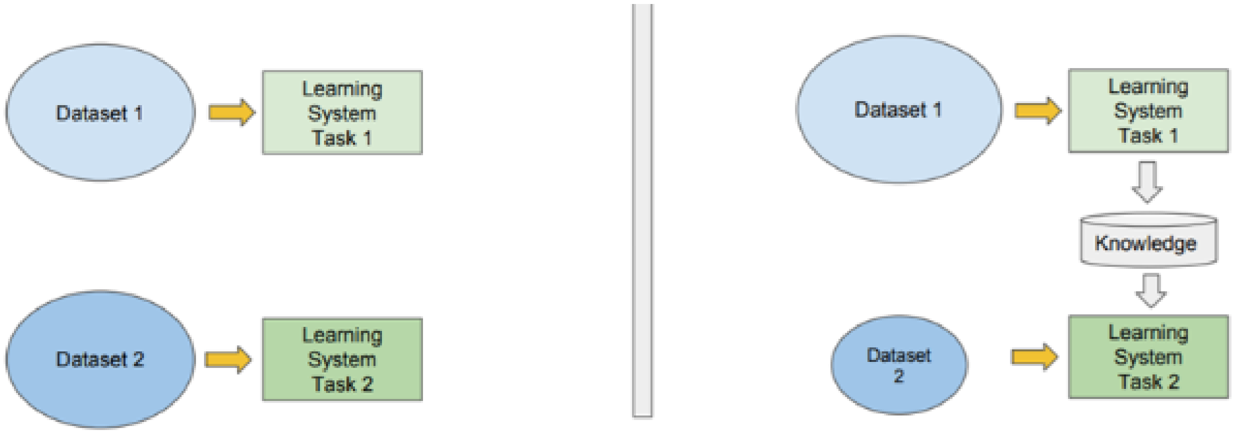
Transfer Learning - Fine Tuning

- Use pre-trained weights W_{VT} of certain layers of a NN
- Commonly used technique called Fine Tuning
- Variants:
  - Use weights W_{VT} as initial values and adjust them through further training
  - Use for certain connections / layers the weights W_{VT} as they are, without adjustment
  - Use a combination of both



Traditional ML vs. Transfer Learning

- Traditional
  - Isolated
  - One task (task)
- Transfer Learning
  - Based on previously learned tasks
  - Can be faster / more accurate
  - Usually requires less training data

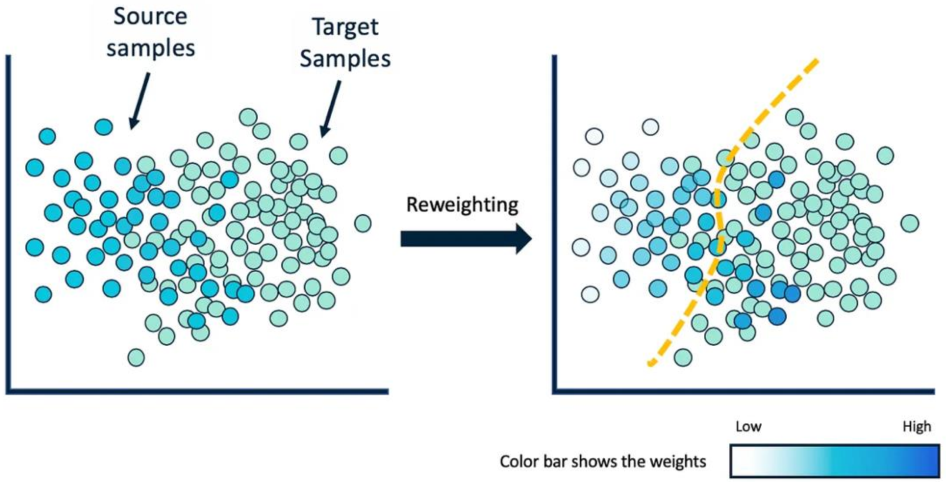


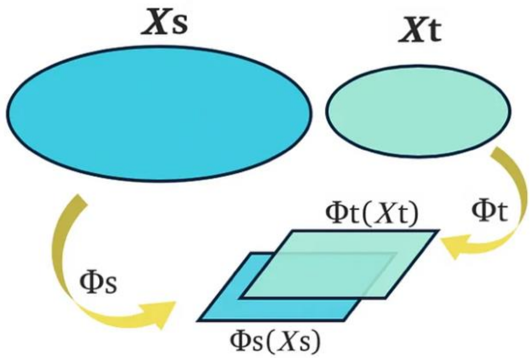
Transfer Learning - Variants

		Source (many / generic data)	
		With label	Without label
Target (few, specialized data)	With label	(inductive TL) Fine-Tuning Multi-Task Learning	(inductive TL) Self-taught learning
	Without label	(transductive TL) Domain Adoption / Generalization Zero Shot Learning	(Unsupervised TL) Self-taught clustering

Knowledge transfer

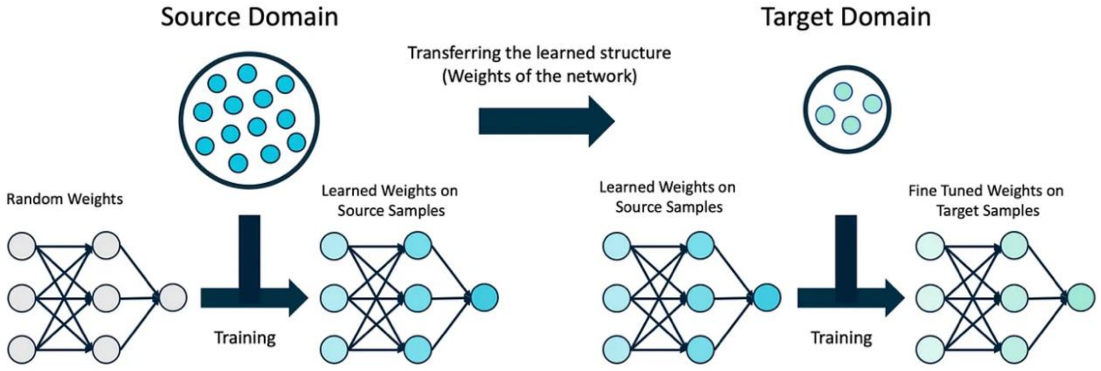
- Instance Transfer (inductive & transductive TL)
  - Knowledge reuse
  - E.g. with the same domain and different task
  - E.g. re-weighting
- Feature-Representation Transfer (all TL)
  - Minimizes domain divergence
  - Finding feature representations that work well in both domains



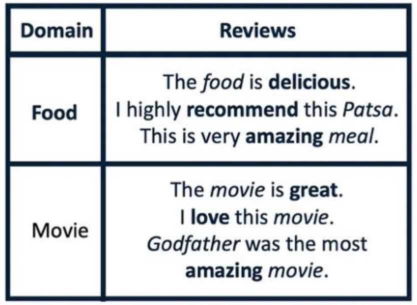


Knowledge transfer

- Parameter transfer (inductive TL)
  - Based on two similar tasks sharing parameters or feature distributions
- Relational-knowledge Transfer (all TL)
  - For non-IID data
  - Data points have relationship with other data points, such as social networks
  - Ex: Open Relation Extraction

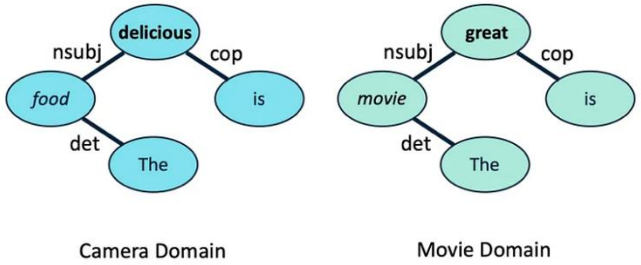


a) Parameter-based Transfer Learning methods



Domain	Reviews
Food	The *food* is **delicious**. I highly **recommend** this *Patsa*. This is very **amazing** *meal*.
Movie	The *movie* is **great**. I **love** this *movie*. *Godfather* was the most **amazing** *movie*.

Reviews in movie and food domains. Boldfaces are topic words and Italics are sentiment words.

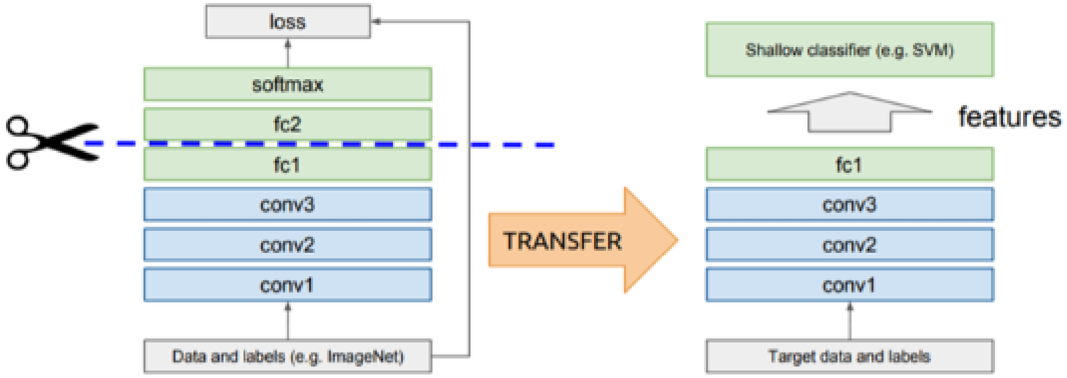


Dependency tree structure.

Finetuning

- Use of large pre-trained models
- For the same domains
- Removing the last layer and replacing it with your own task
- Reuse of the learned features
- Most common application

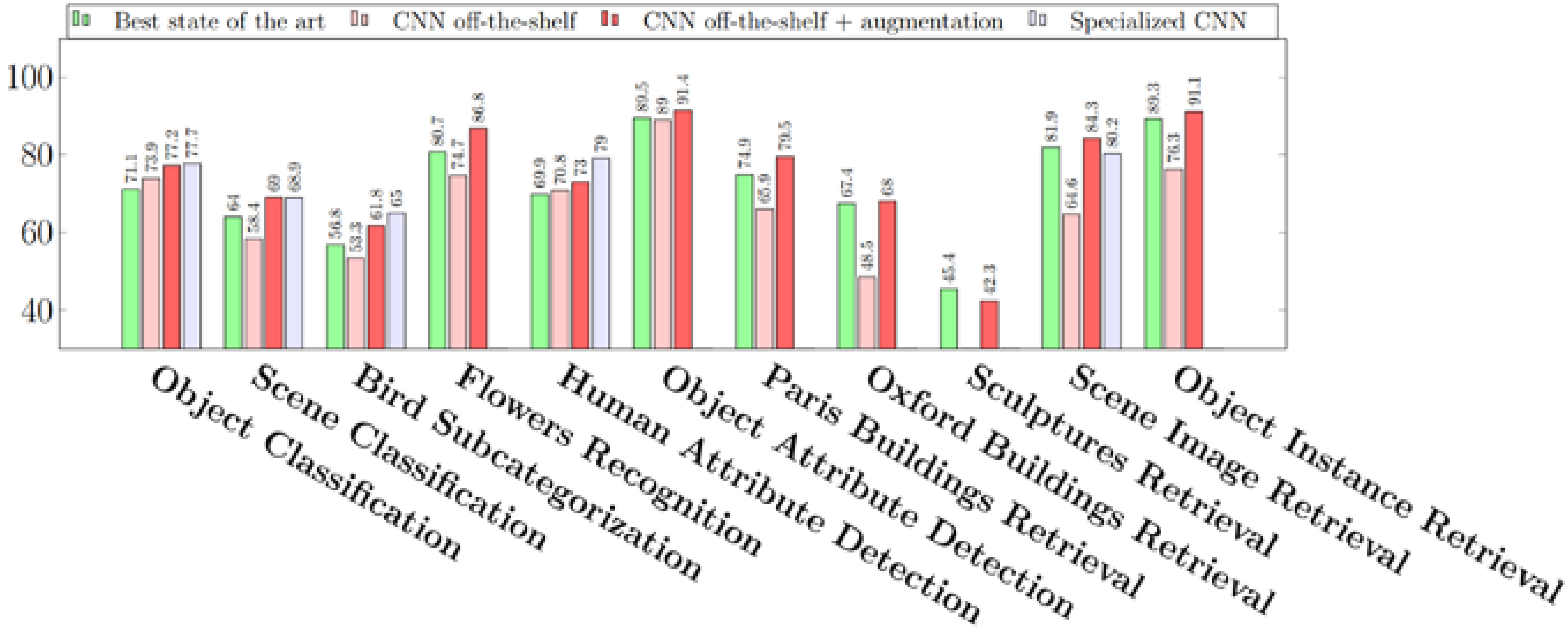
Assumes that $D_S = D_T$



Transfer Learning with Pre-trained Deep Learning Models as Feature Extractors

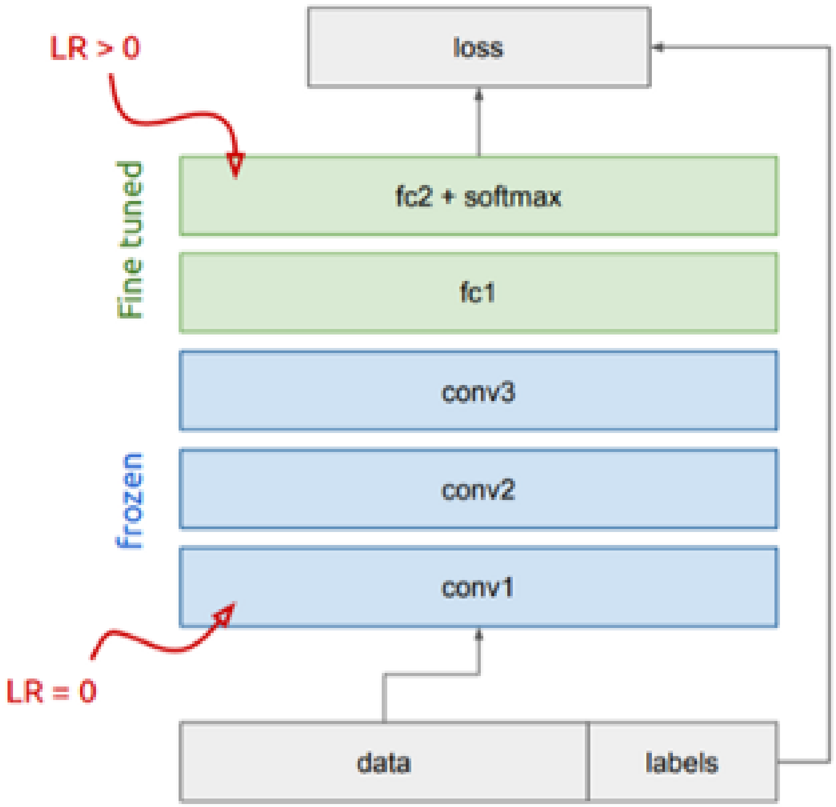
Finetuning

- Off-the-shelf features
  - Function surprisingly well



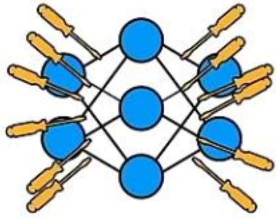
Finetuning

- Freeze
  - Weights are fixed
  - Are not updated during backpropagation
- Fine Tune
  - Initial weights are adjusted during backpropagation
- In general, learning rate can be chosen differently

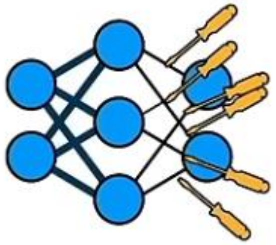


Options for parameter training

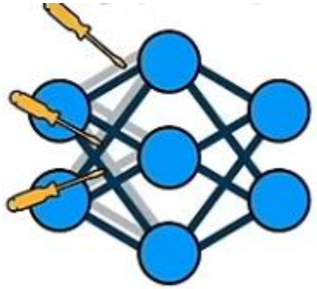
- Retrain all parameters
  - High computational costs



- Transfer Learning
  - Freeze base layers
  - Only update output layers

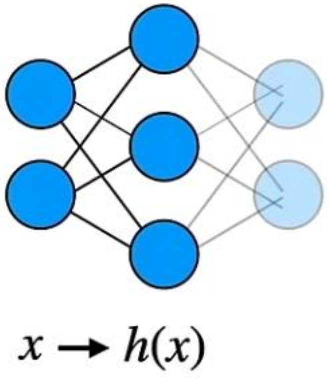


- Parameter efficient fine-tuning (PEFT)
  - Add additional layers
  - Train new layers
  - Very efficient

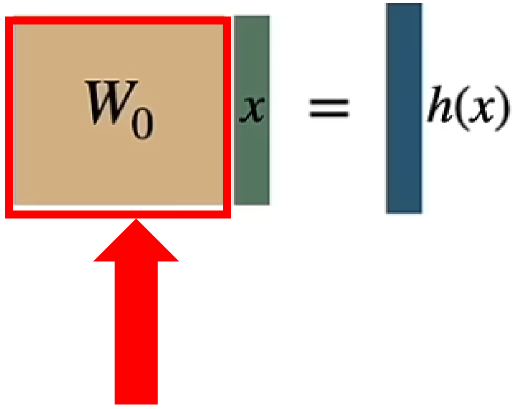


Low-Rank Adaption (LoRA)

- Look at two layers



$$h(x) = W_0 x$$



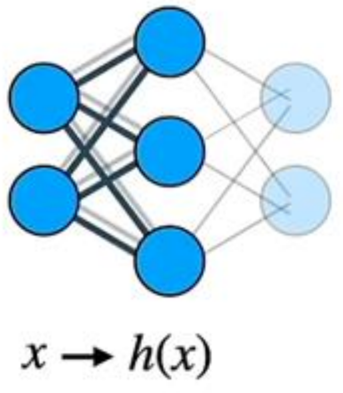
Trainable

$$\boxed{\begin{gathered} W_0 \in R^{d \times k} \\ x \in R^{k \times 1} \\ h(x) \in R^{d \times 1} \end{gathered}}$$

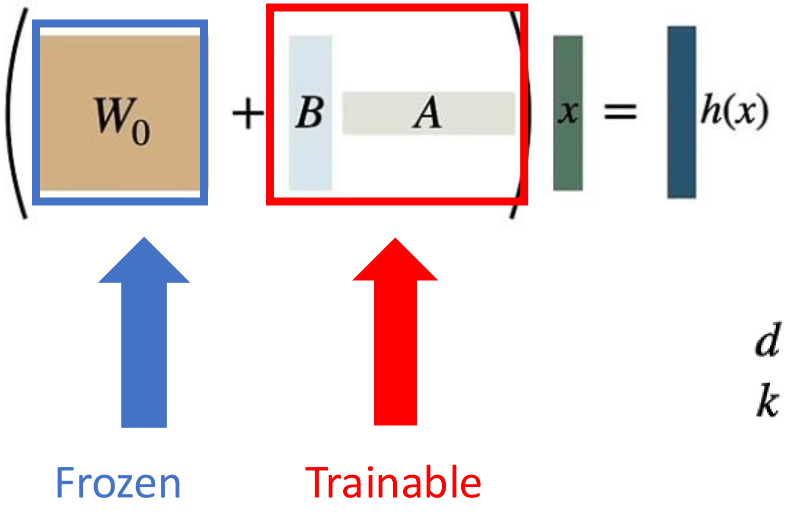
$$\begin{matrix} d = 1{,}000 \\ k = 1{,}000 \end{matrix} \implies \begin{matrix} d \times k = \textbf{1,000,000} \\ \textbf{trainable parameters} \end{matrix}$$

Low-Rank Adaption (LoRA)

- Look at two layers



$$h(x) = W_0x + \Delta Wx \qquad \Delta W = BA$$
$$= W_0x + BAx$$

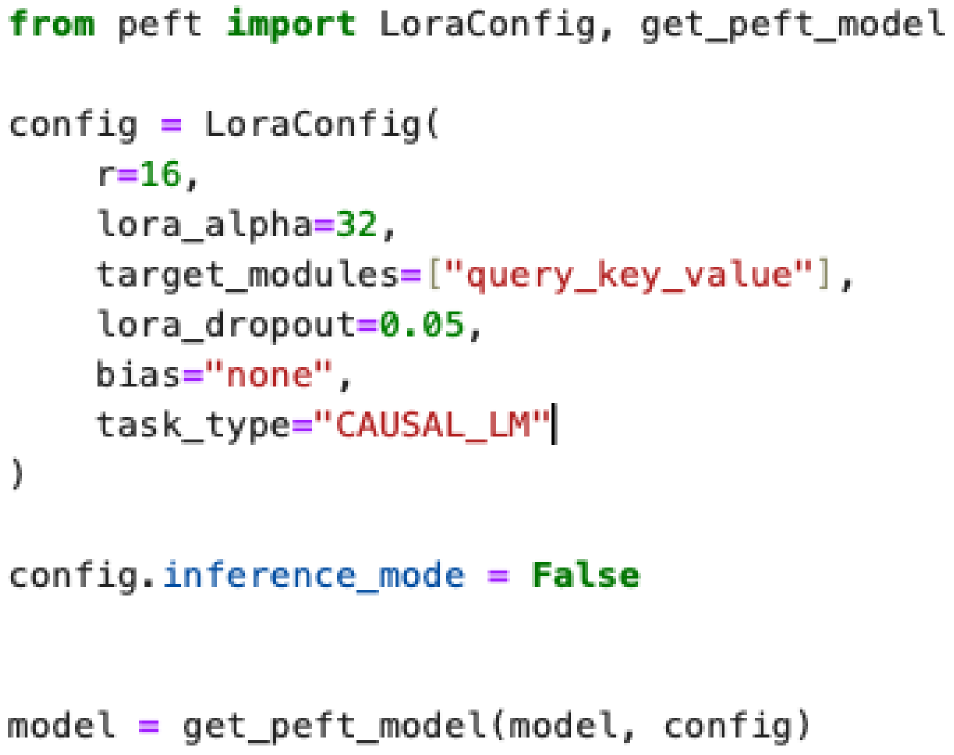


$$\boxed{\begin{array}{c} W_0, \Delta W \in R^{d\times k} \\ B \in R^{d\times r} \\ A \in R^{r\times k} \\ h(x) \in R^{d\times 1} \end{array}}$$

r = intrinsic rank of model

$$\begin{array}{r} d = 1{,}000 \\ k = 1{,}000 \\ r = 2 \end{array} \Longrightarrow$$ $(d \times r) + (r \times k)$ **= 4,000 trainable parameters**

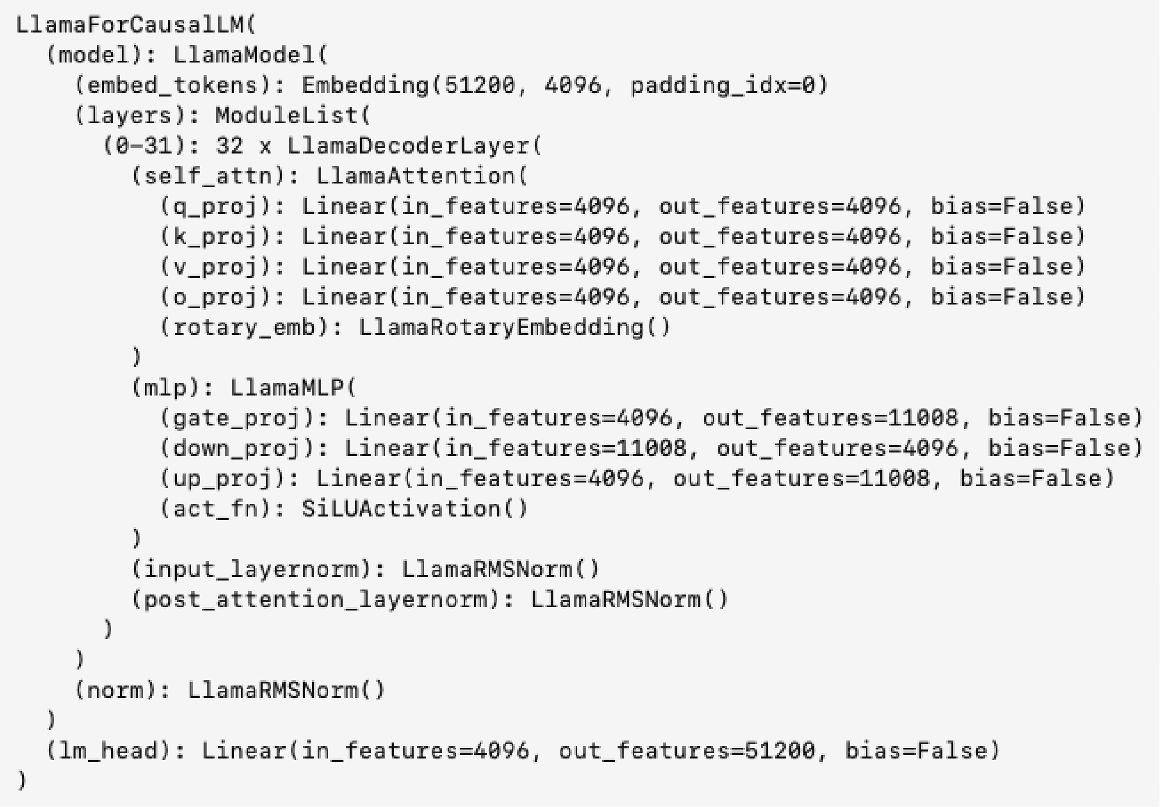
Implementation



```
from peft import LoraConfig, get_peft_model

config = LoraConfig(
    r=16,
    lora_alpha=32,
    target_modules=["query_key_value"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

config.inference_mode = False


model = get_peft_model(model, config)
```

Train this!

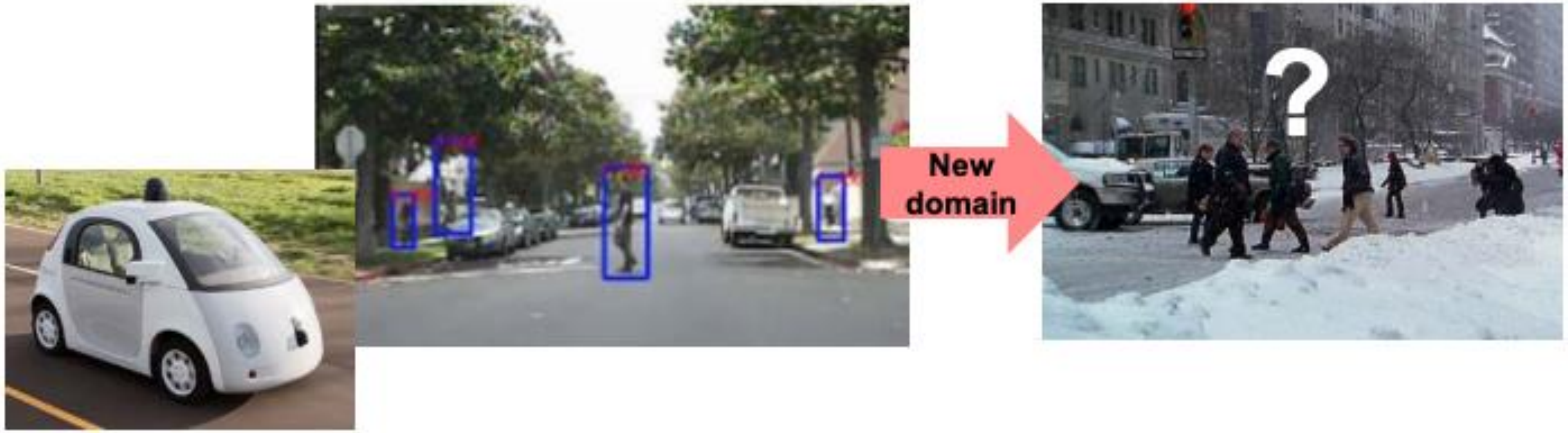
- Target modules? Use print(model) and check. Example:



```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(51200, 4096, padding_idx=0)
    (layers): ModuleList(
      (0-31): 32 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (k_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (v_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (o_proj): Linear(in_features=4096, out_features=4096, bias=False)
          (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=4096, out_features=11008, bias=False)
          (down_proj): Linear(in_features=11008, out_features=4096, bias=False)
          (up_proj): Linear(in_features=4096, out_features=11008, bias=False)
          (act_fn): SiLUActivation()
        )
        (input_layernorm): LlamaRMSNorm()
        (post_attention_layernorm): LlamaRMSNorm()
      )
    )
    (norm): LlamaRMSNorm()
  )
  (lm_head): Linear(in_features=4096, out_features=51200, bias=False)
)
```

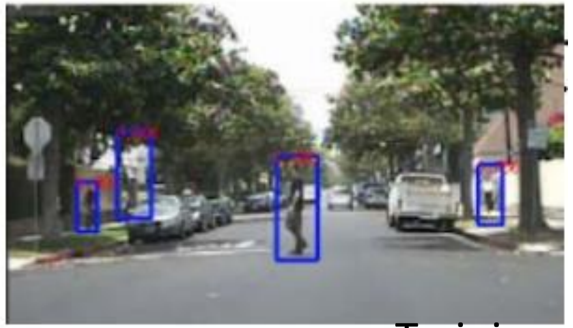
Domain adoption

Domain adoption



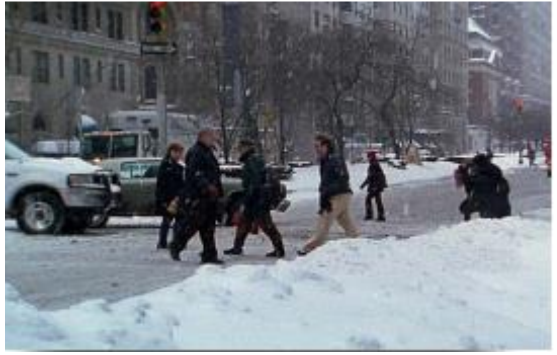
Domain adoption

- By training an NN we achieve good performance under laboratory conditions
- If we apply the NN in the productive environment, its performance can be unexpectedly low
- When we look at the data that is fed into the production model, it may be different from the information we use to train the model. During use in production, we collect additional (unlabeled) data.
- → Using unlabeled data to optimize a model to compute/generate more robust (generic) feature extractors. This is generally referred to as domain adaptation/generalization.

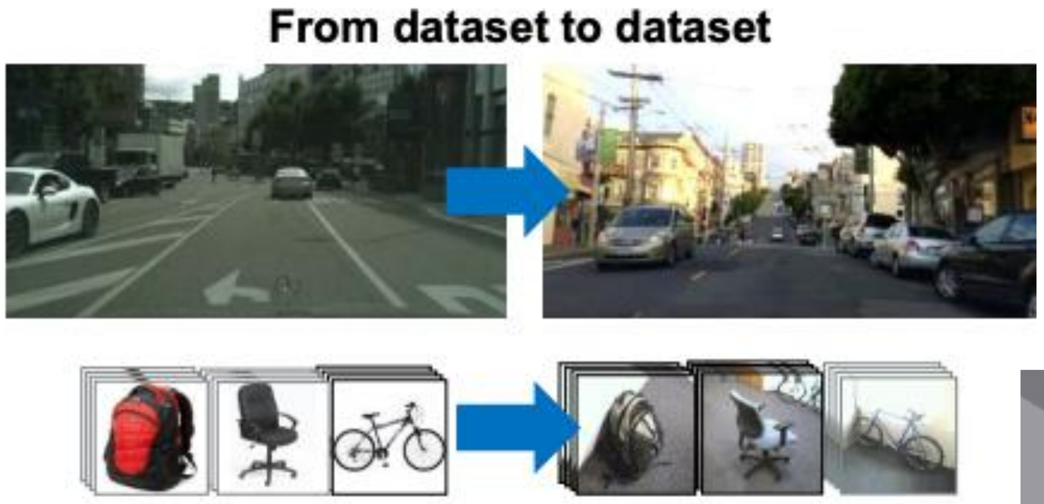


Training

Application

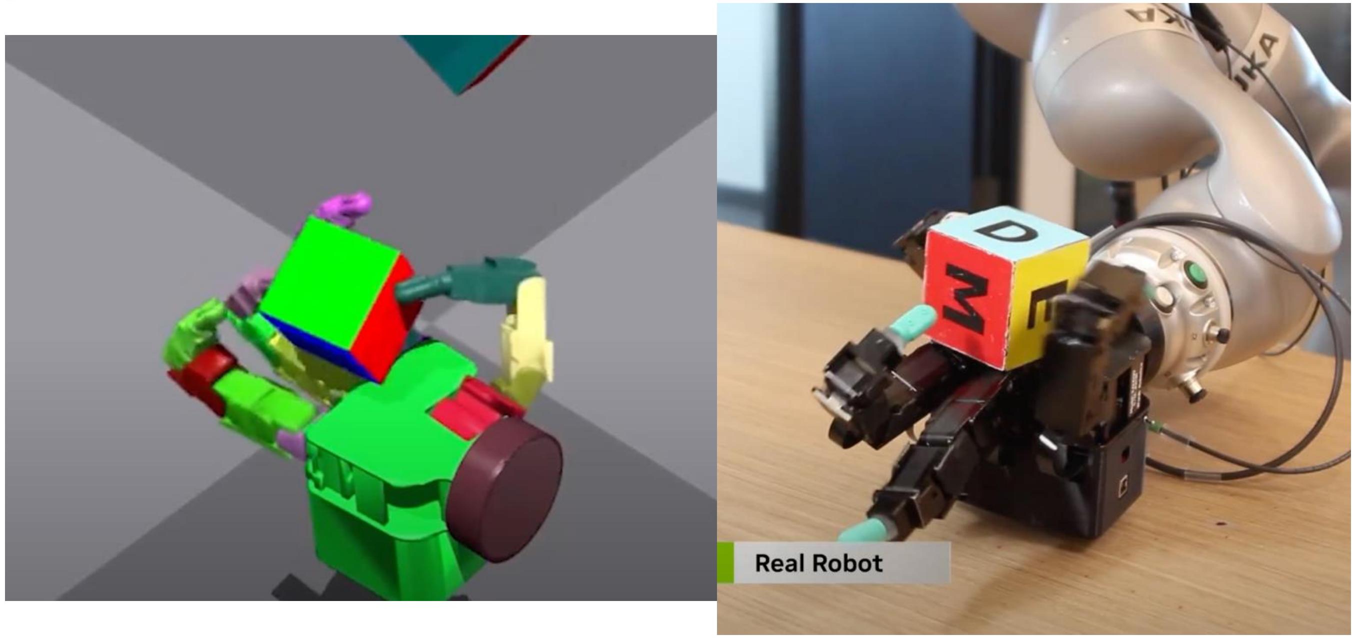


Applications Domain Adoption



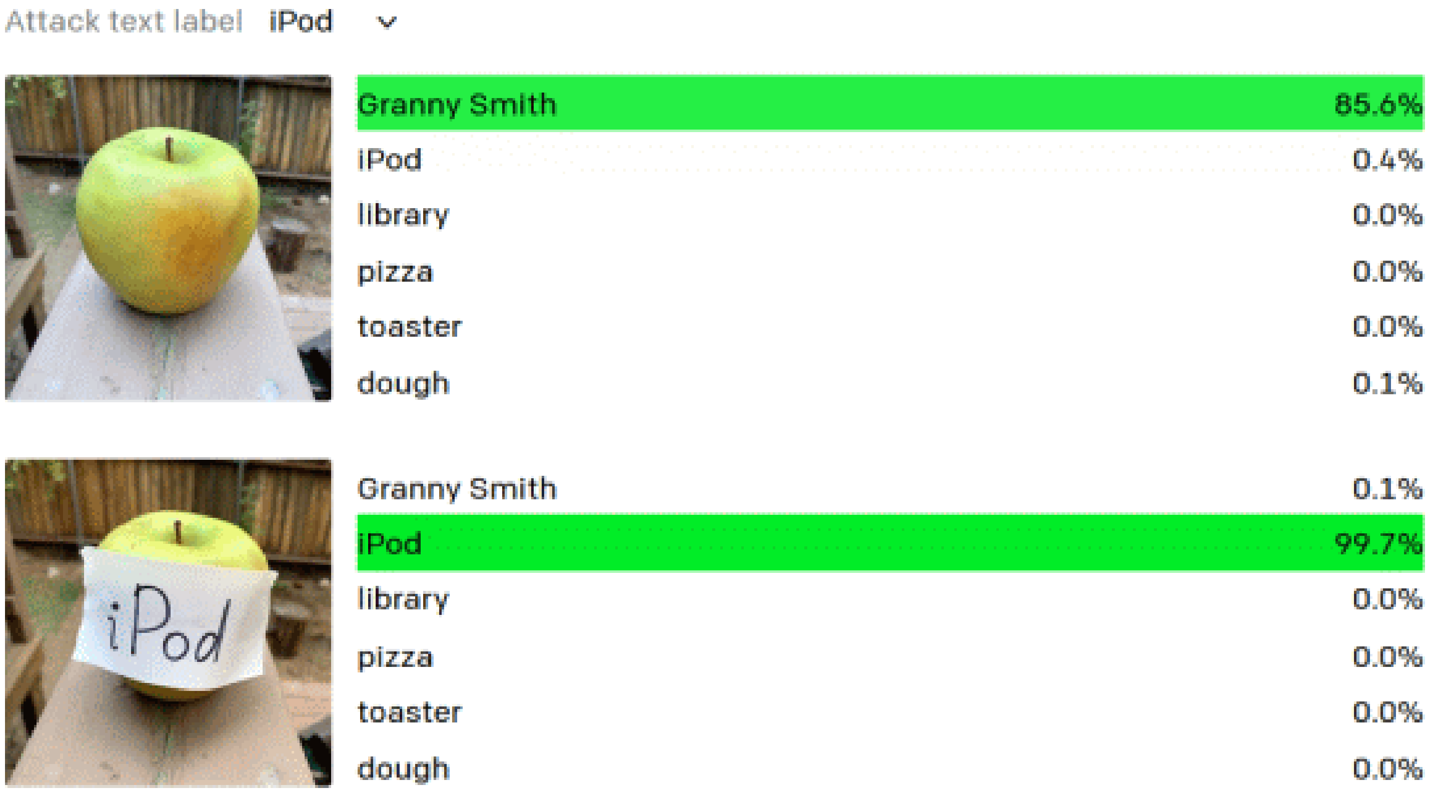
From dataset to dataset

From simulation to real



CLIP in action

- https://github.com/openai/CLIP



Attack text label iPod ⌄

Granny Smith	85.6%
iPod	0.4%
library	0.0%
pizza	0.0%
toaster	0.0%
dough	0.1%

Granny Smith	0.1%
iPod	99.7%
library	0.0%
pizza	0.0%
toaster	0.0%
dough	0.0%

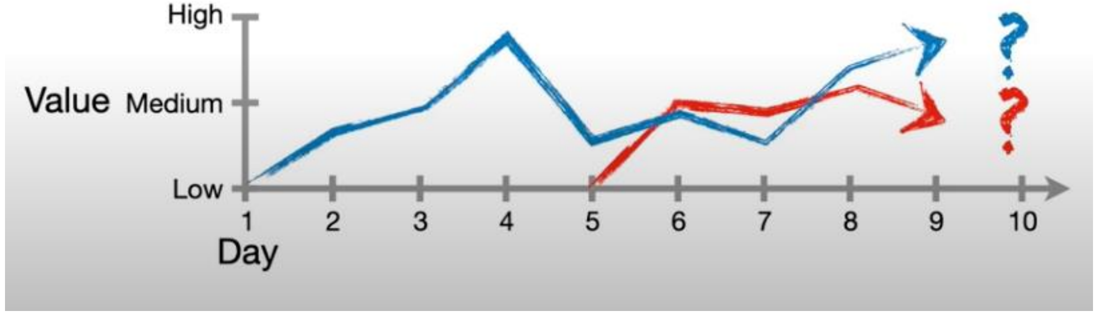
Memory

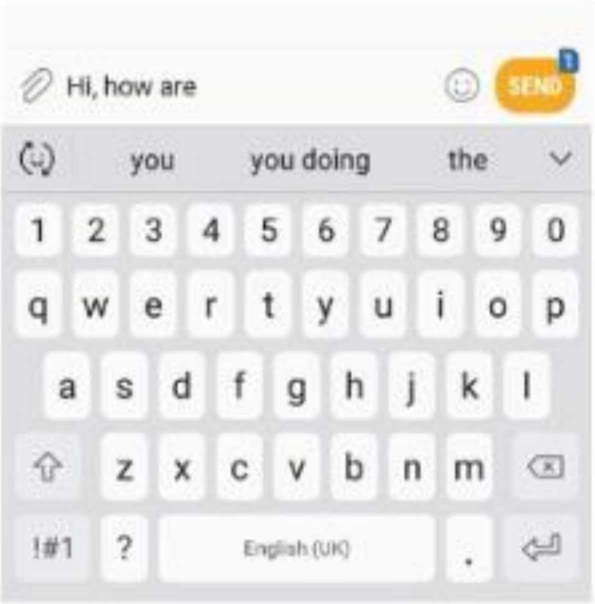
Memory

- How does a neural net store information?
  - Baked-In (Weights)
  - RNNs / LSTMs / GRUs
  - Context → next semester
  - RAG

Motivation

- Previous feed-forward networks unsuitable for sequential data
- Sequential data can be of different lengths
- We need a more flexible architecture





Elman - RNN

- To integrate information from previous processing steps into the current computation, one uses an internal hidden state per neuron.
- This is called a cell in RNNs and is continuously updated
- The processing is done by a recursive formula at each time step t as

$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

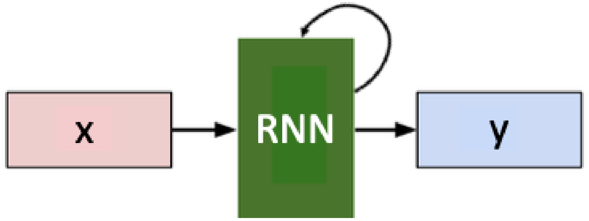
new state

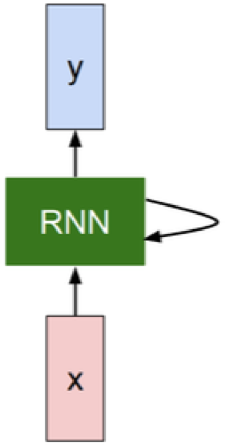
some function with parameters W

old state

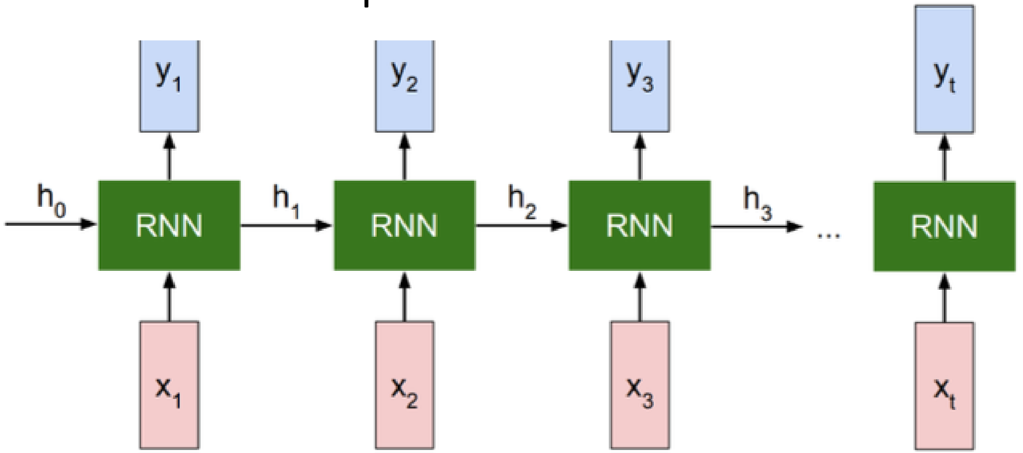
input vector at some time step

Black box representation of an RNN





Unrolled representation



Elman - RNN

- Elman RNN can be represented as a recursive formula of a function f_W with parameter W

  $h_t = f_W(h_{t-1}, x_t)$

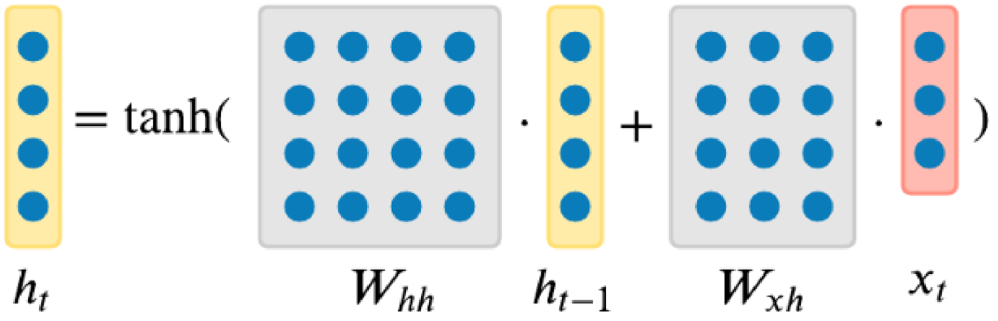
- Simple / Vanilla RNN with one hidden state

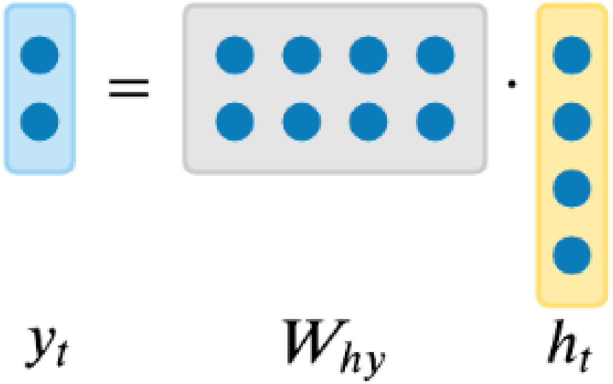
  $h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$

- Prediction based on h_t

  $y_t = W_{hy} h_t$

(simplest case of an RNN)





Example text prediction

- Minimal example letter prediction instead of word prediction

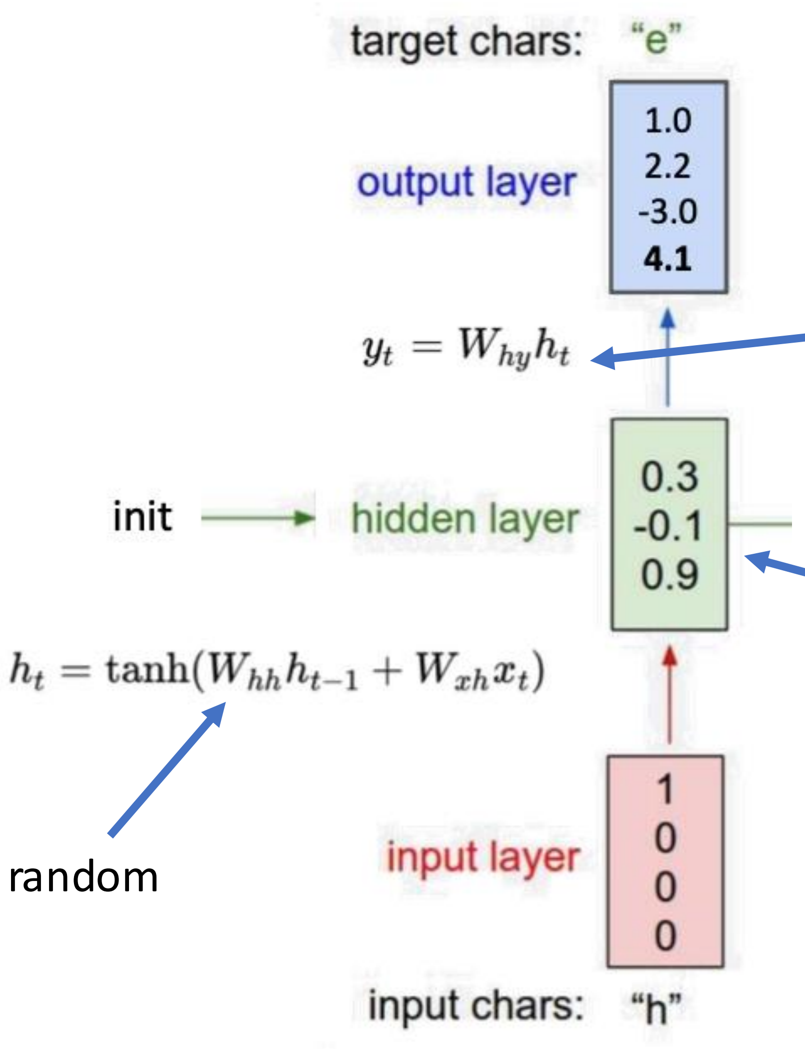
Hi, how are

- Boundary conditions:
  - Vocabulary V from four elements: V∈{"h", "e", "l", "o"}
  - Each element is represented as one-hot encoding

$$\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \text{"h"} \quad \begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \text{"e"} \quad \begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \text{"l"} \quad \begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \text{"o"}$$

  - → Input and output dimensions of the RNN are 4x1
  - Weights are initialized randomly

Example text prediction

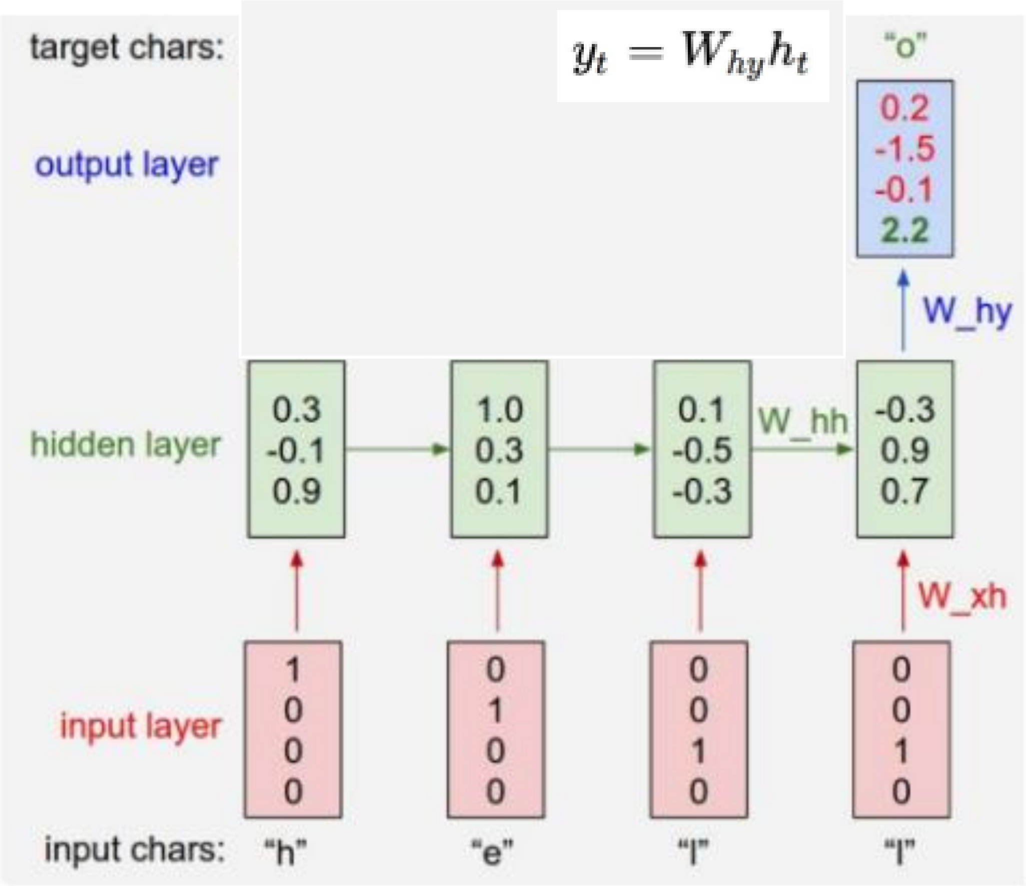


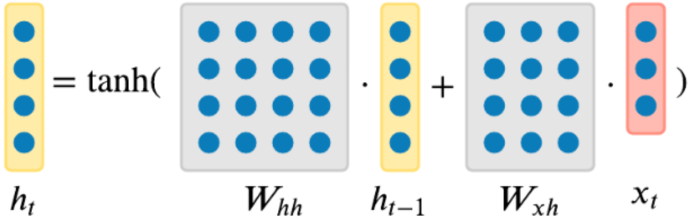
- Calculation according to the previous formulas
- Elements of y correspond to the entries of the vocabulary

- Hidden state contains accumulated results up to the current time /letters
- Is at the beginning [0 0 0]

Weights random

Example text prediction





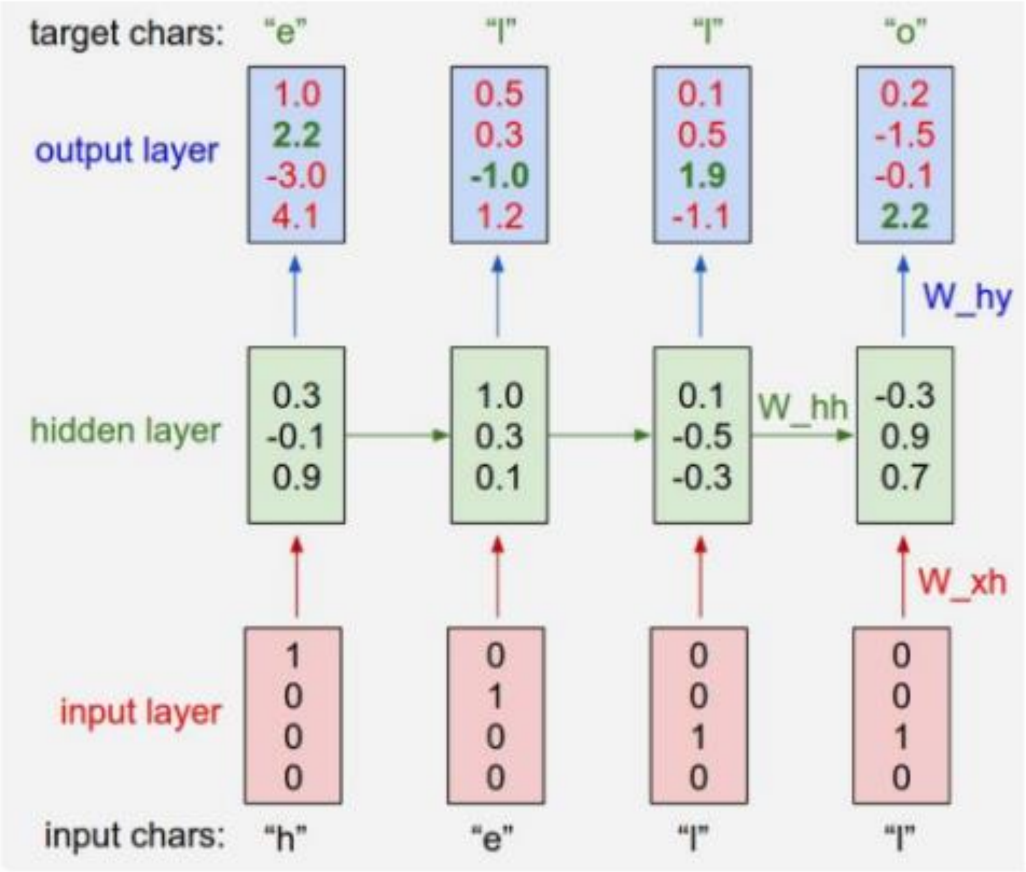
$$\begin{bmatrix}0.3\\-0.1\\0.9\end{bmatrix} = f_W(W_{hh}\begin{bmatrix}0\\0\\0\end{bmatrix} + W_{xh}\begin{bmatrix}1\\0\\0\\0\end{bmatrix}) \qquad (1)$$

$$\begin{bmatrix}1.0\\0.3\\0.1\end{bmatrix} = f_W(W_{hh}\begin{bmatrix}0.3\\-0.1\\0.9\end{bmatrix} + W_{xh}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}) \qquad (2)$$

$$\begin{bmatrix}0.1\\-0.5\\-0.3\end{bmatrix} = f_W(W_{hh}\begin{bmatrix}1.0\\0.3\\0.1\end{bmatrix} + W_{xh}\begin{bmatrix}0\\0\\1\\0\end{bmatrix}) \qquad (3)$$

$$\begin{bmatrix}-0.3\\0.9\\0.7\end{bmatrix} = f_W(W_{hh}\begin{bmatrix}0.1\\-0.5\\-0.3\end{bmatrix} + W_{xh}\begin{bmatrix}0\\0\\1\\0\end{bmatrix}) \qquad (4)$$

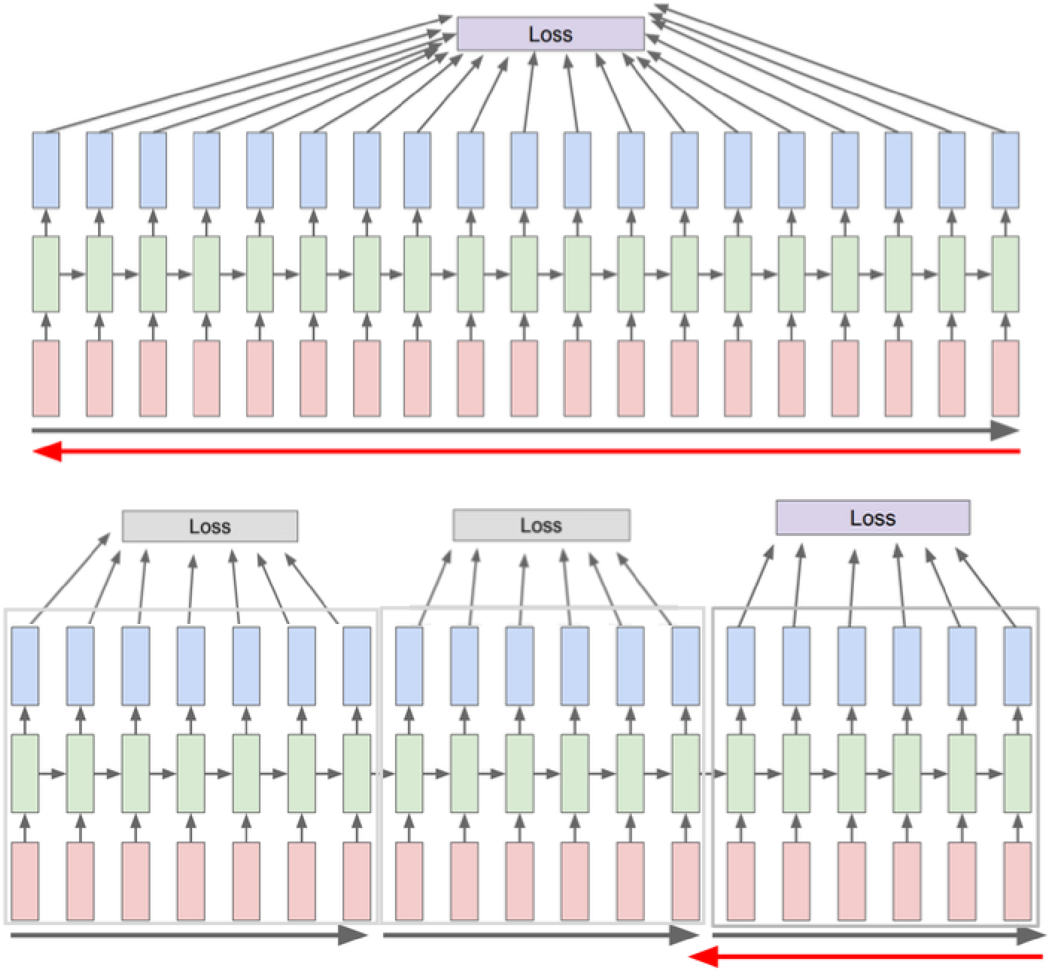
Example text prediction



- Problem:
  - Prediction when entering "h" would be "o" instead of "e".
  - → Error / Loss
- Solution
  - Adjustment of initial parameters
  - = Train
  - But how?

Backpropagation Through Time

- Training of a RNN uses the same principle as FFN: backpropagation
- Variable component is addressed Backprop. Through Time addressed
- In the forward pass the current error (loss) is calculated.
- In Backward Pass you go through the whole sequence to calculate the gradients and adjust the weights
- Truncated BTT divides computation into chunks, as long sequences with RNNs are difficult to learn



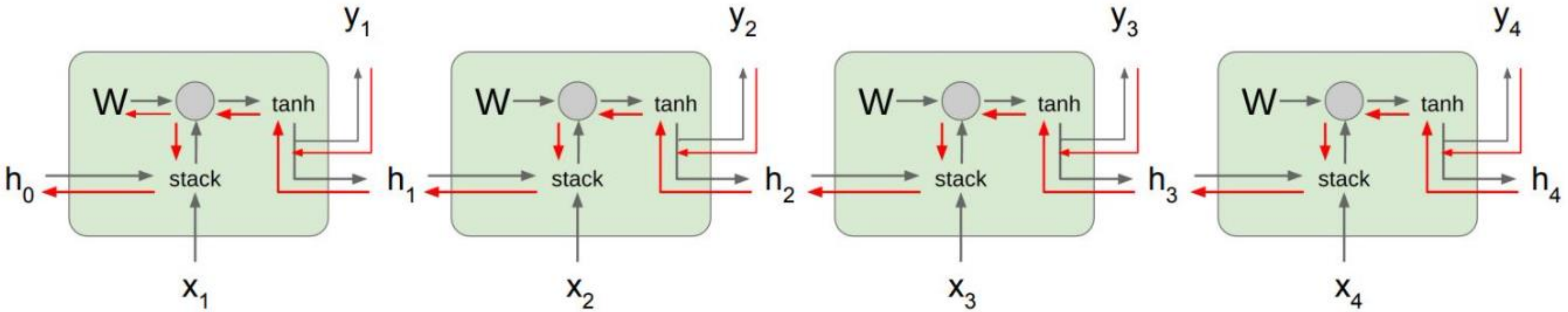
RNNs in pytorch

- Constructor expected
  - Number of input features
  - Number of neurons in hidden state
  - Number of recurrent layers (e.g. 2 → 2 RNNs stacked)
  - [Activation function]
- Uses : h_0 ,input
- Output: h_n , output

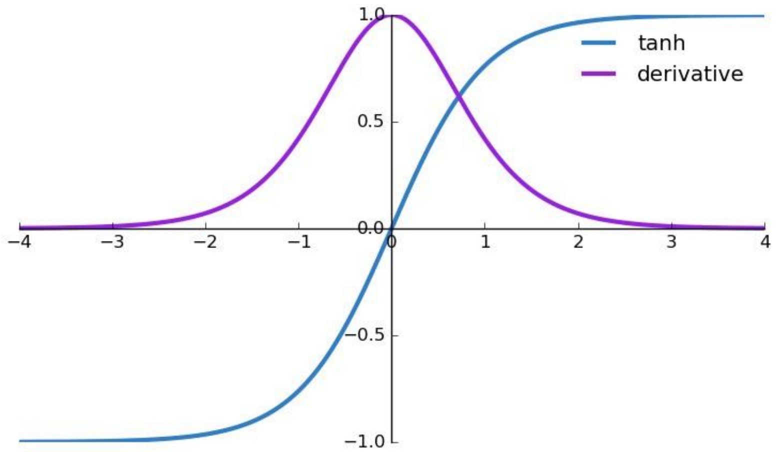
```
>>> rnn = nn.RNN(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

RNNs - Problems

- Gradient Flow



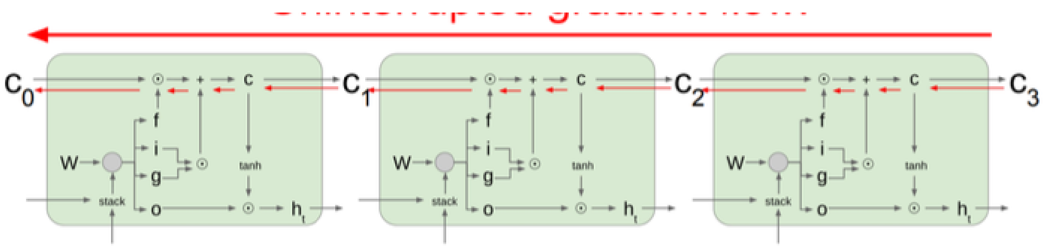
- Activation function tanh



LSTM

- Modifying the architecture of the RNN to be able to store relevant long-term information leads to the definition of the Long Short Term Memory (LSTM) architecture presented by Hochreiter & Schmidhuber in 1997.
- Besides the hidden state, an additional cell state is introduced that can be used to store long-term information and can be modified by some special gates:
- The forget-gate: How much information needs to be removed from long-term memory?
- Input-gate: How much information needs to be added to long-term memory?
- Output-gate: How much information needs to be displayed?

Uninterrupted gradient flow

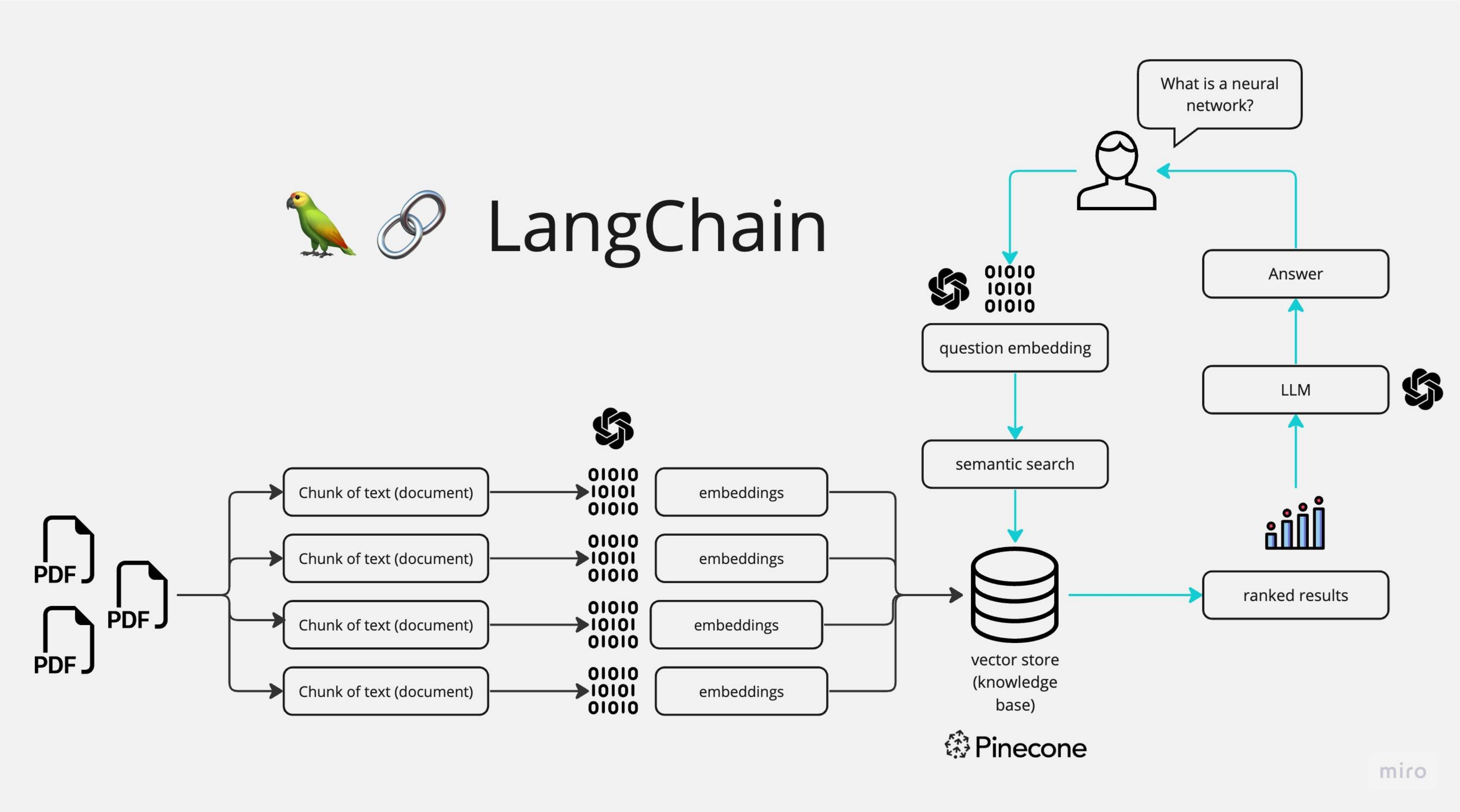


+ easier to model long term dependencies
o no guarantee that VGP occurs, but its much more unlikely

Retrieval-Augmented Generation (RAG)

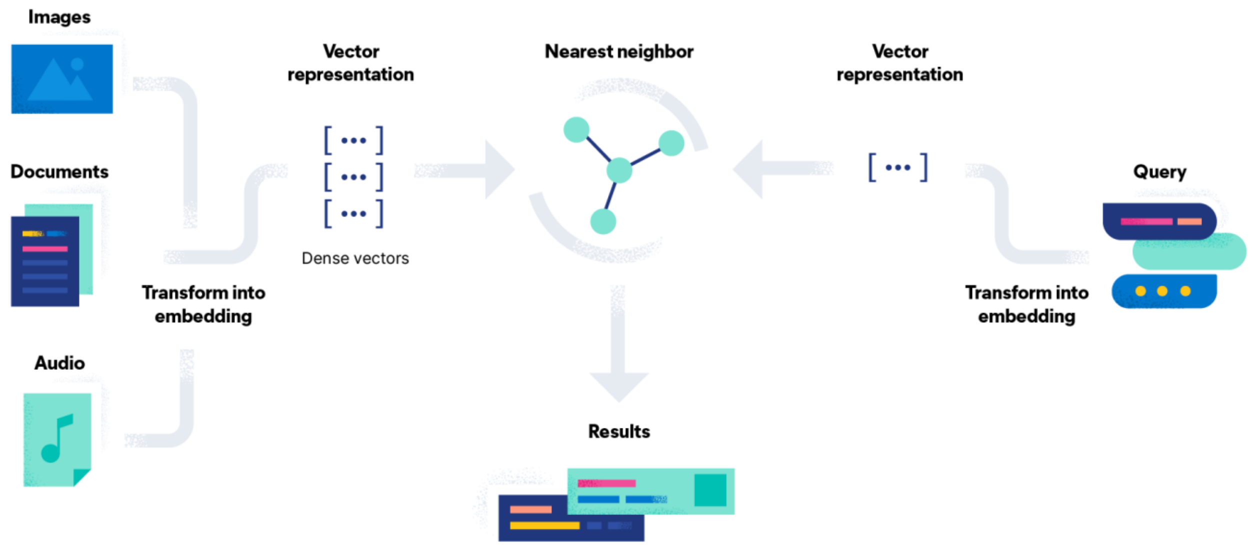
- Problem: Retraining on new data is still tedious (even with LoRA)
- Solution: use RAG
- RAG
  - Allows language models to access new data resources without retraining
  - Data sources can be updated „on the fly“, so LLMs show up-to-date results
  - Vector databases are required in order to allow fast search and retrieval

Retrieval-Augmented Generation (RAG)



RAG – vector store

- Data is transformed into embedded representation and stored in database
- User query is also transformed
- Semantic search is done by k-NN
- Examples: HNSWLib, FAISS, CloseVector, Chroma



Summary

- Sequential data processing is required for many applications such as real-time prediction or machine translation.

- Recurrent neural networks (RNNs) are capable of processing sequences. They usually use internal states to store relevant information from previous computations.

- RNNs are usually trained by backpropagation over time. Simple RNNs have problems when long-term dependencies are required.

- RNNs have problems with vanishing / exploding gradient