Deep Learning

Shallow vs Deep

Understanding Machine Learning

Recap: Goal of ML

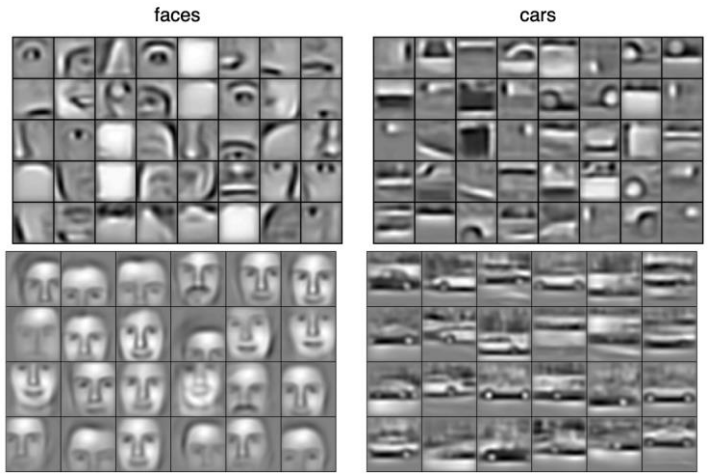
generalization from optimization on training data set (approximation of true data generating probability distribution by empirical risk minimization)

- fitting: complex function approximation
- for generalization: learning of good abstraction/representation of data/concepts

→ deep learning methods (MLP, CNN, …) optimal candidates

e.g., CNNs can learn hierarchical representation by means of many convolutional and pooling layers

the deeper the better (accuracy, hierarchical representation)



source

Deep Learning

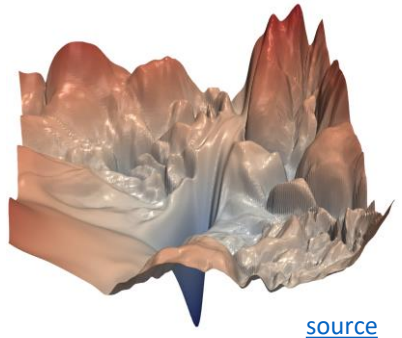
But … How to Train Deep Neural Networks?

optimization and regularization difficult

- non-convex optimization problem (e.g., local vs global minima), easily overfitting
- many hyperparameters to tune

many methods to get it working in practice (despite partly patchy theoretical understanding)

typical loss surface:



source

optimization

- activation and loss functions
- weight initialization
- stochastic gradient descent
- adaptive learning rate
- batch normalization

explicit regularization

- weight decay
- dropout
- data augmentation
- weight sharing

implicit regularization

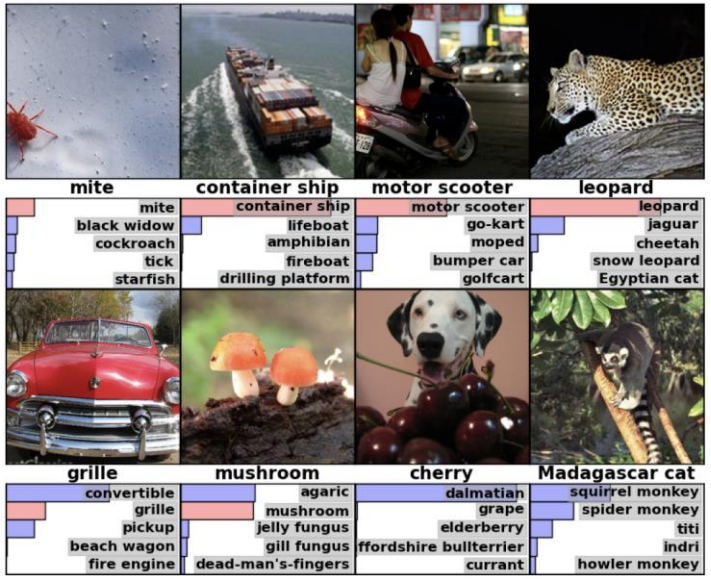
- early stopping
- batch normalization
- stochastic gradient descent

History: Rise of Deep Learning

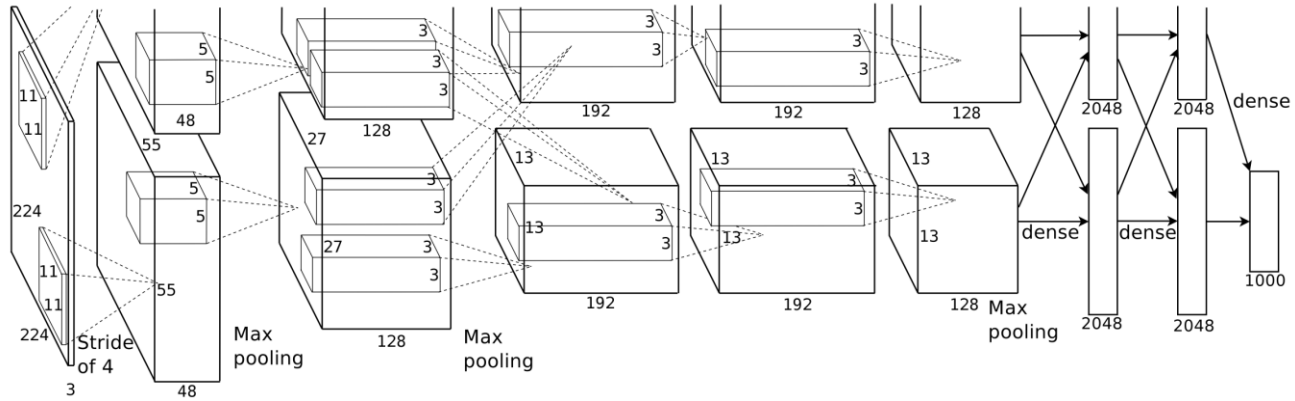
a little bit oversimplified:

deep learning = lots of training data + parallel computation + smart algorithms

AlexNet: ImageNet + GPUs + ReLU, dropout



source



Rectified Linear Unit (ReLU)

reminder: activation function non-linear transformation of summed weighted input of a node (linear), output to be used as input for nodes of subsequent layer

needs to be differentiable for back-propagation (ReLU at 0 no issue, just set to 0 or 1)

neural network model with ReLU activation can be interpreted as exponential number of linear models that share parameters

main advantages leading to enablement of deeper networks by better optimization:

- unlike sigmoid or $\tanh$ (predominantly used before) activation, no issue with vanishing gradients from saturation effects
- very efficient computation: constant gradients of 0 and 1 below and above input of zero
- sparse activation: many hidden nodes deactivated (output 0)

Weight Initialization

starting values for weights crucial for convergence of deep learning trainings

most important: need to break symmetry between different nodes in a hidden layer (same initial weights lead to identical weight updates) ← early issue in back-propagation

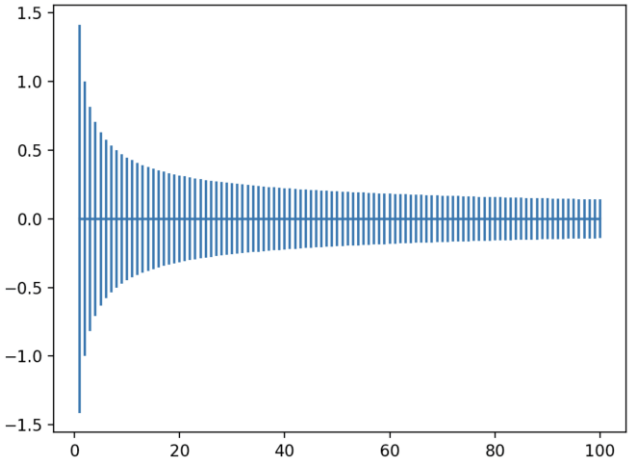
→ small random numbers (from Gaussian or uniform distribution) work

(only bias weights set to zero by default)

but specific heuristics using information on activation function and number of inputs to node can improve optimization

for ReLU, he initialization works well: randomly draw from zero-mean Gaussian with standard deviation $\sqrt{2/n}$

number of inputs



Plot of Range of He Weight Initialization With Inputs From One to One Hundred

source

(Stochastic) Gradient Descent

using gradient of cost (objective) function with respect to weights: $\nabla_{\widehat{\boldsymbol{w}}} J(\widehat{\boldsymbol{w}})$

updates $\widehat{\boldsymbol{w}} \leftarrow \widehat{\boldsymbol{w}} - \eta \nabla_{\widehat{\boldsymbol{w}}} J(\widehat{\boldsymbol{w}})$ can be done with whole training data set (n observations) or small random sample:

- $J(\widehat{\boldsymbol{w}}) = \frac{1}{n} \sum_{i=1}^{n} J_i(\widehat{\boldsymbol{w}})$ batch (or deterministic) gradient descent
- $J(\widehat{\boldsymbol{w}}) = J_i(\widehat{\boldsymbol{w}})$ stochastic gradient descent (single example)
- $J(\widehat{\boldsymbol{w}}) = \frac{1}{m} \sum_{i=1}^{m} J_i(\widehat{\boldsymbol{w}})$ mini-batch stochastic gradient descent (size m)

implicit regularization: (mini-batch) stochastic gradient descent follows gradient of true generalization error, as long as no examples are repeated (but usually many epochs in training)

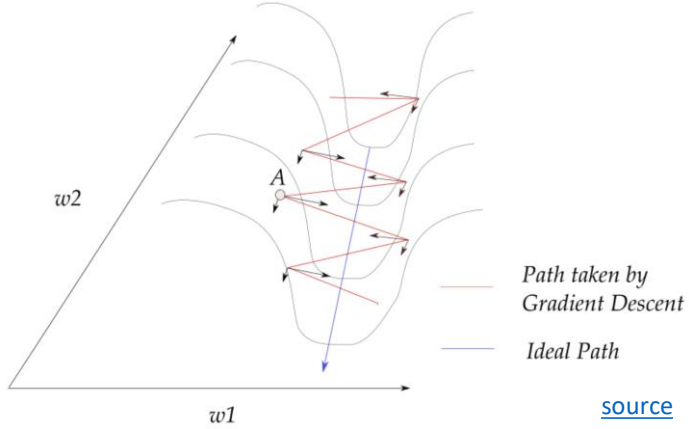
Mini-Batch Sizes

trade-off:

- larger batches give more accurate gradient estimates → allowing for higher learning rate
- smaller batches have (implicit) regularization effect and better convergence

in practice, also need to consider memory limitations and run times

Adaptive Learning Rate



source

strategies for gradient descent learning rate: constant, decaying, with momentum (escape from local minima and saddle points)

better convergence by adapting learning rate for each weight: lower/higher learning rates for weights with large/small updates (avoid direction of oscillations)

popular methods ($\hat{w}$ here denotes individual weight → $g_{\hat{w}}$ component of gradient):

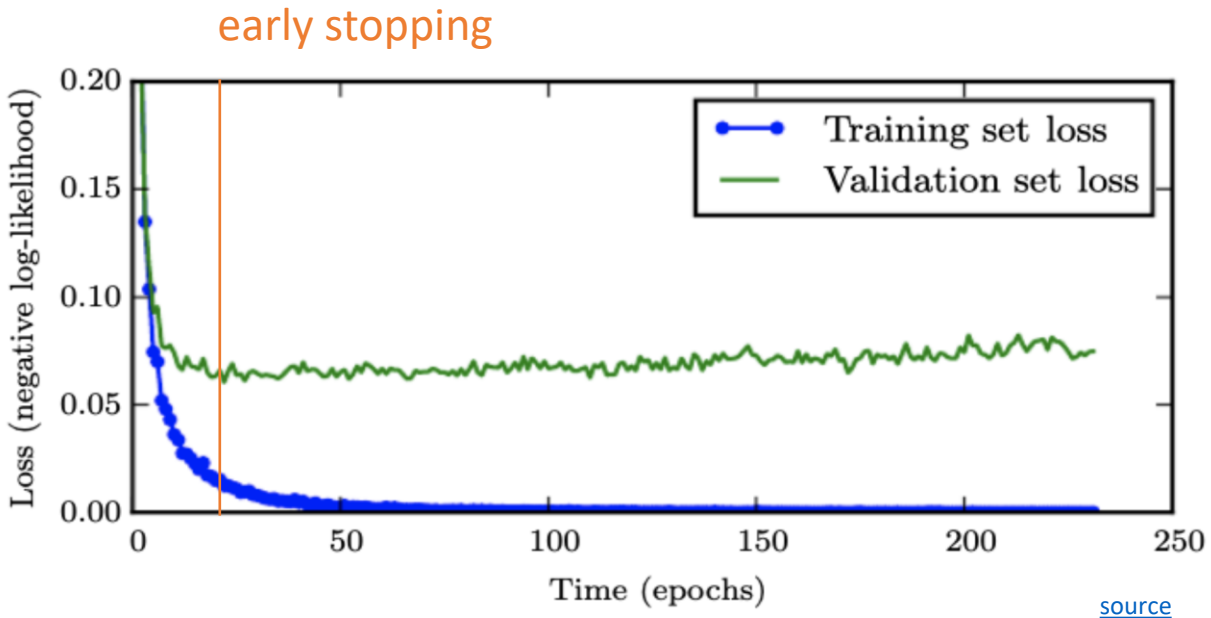
- Adagrad: $\hat{w} \leftarrow \hat{w} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_{\hat{w},\tau}^2}} g_{\hat{w}}$ with t, τ denoting current and past iterations

  (issue: sum in denominator grows with more iterations → danger of stucking)
- RMSProp: $\hat{w} \leftarrow \hat{w} - \frac{\eta}{\sqrt{v(\hat{w})}} g_{\hat{w}}$ with $v(\hat{w}) \leftarrow \gamma\, v(\hat{w}) + (1-\gamma) g_{\hat{w}}^2$
- Adam (Adaptive Moment Optimization): combines RMSProp with momentum

Early Stopping

loss independently measured on validation set

halting training when overfitting begins to occur



source

(suppresses double descent)

Skip Connections: ResNet

issue: degradation of training (and test) error when adding more and more layers (not due to overfitting)

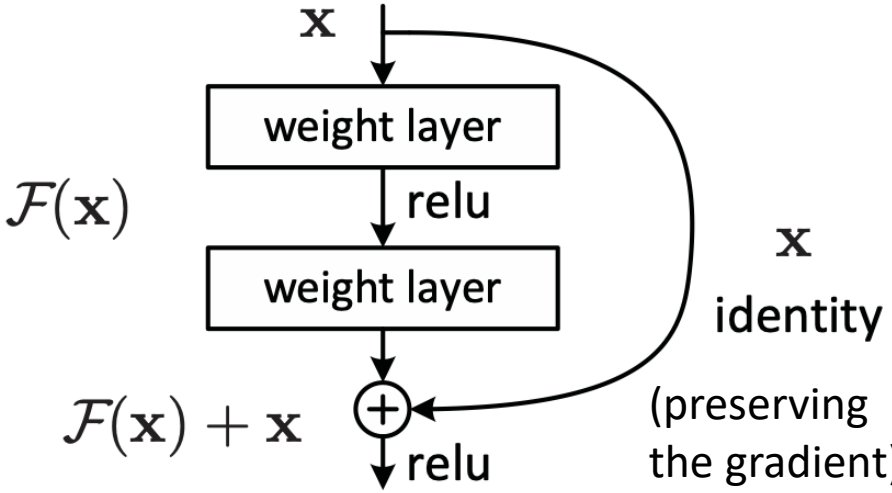
reason: optimization issues with near-identity mappings (deeper layers add only a bit of expressive power) → introducing variance

solution: learning of residuals by means of skip connections (zero weights easier to learn than identity mapping)

produces loss functions that train easier

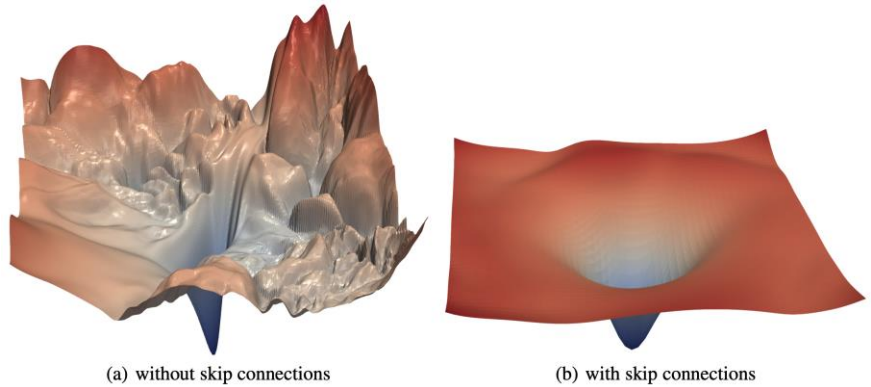
→ enabling extremely deep residual networks (several hundred layers) without degradation

residual mapping (skip/shortcut connections):



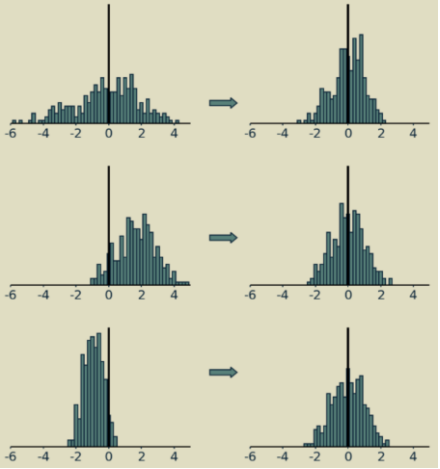
source

loss surface:



source

Batch Normalization



source

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

source

adaptive reparameterization of inputs to a network layer (before or after activation)

independently for each input/feature

(not to confuse with weight normalization: decoupling of length and direction of weight vectors)

to maintain expressive power (optional): γ, β learned together with weights via back-propagation

Benefits from Batch Normalization

- allows higher learning rates
- reduces importance of weight initialization
- alleviates vanishing/exploding gradients
- (implicit) regularization effect: introducing both additive and multiplicative noise, sometimes making dropout (multiplicative noise) unnecessary

reason why batch normalization improves optimization still controversial

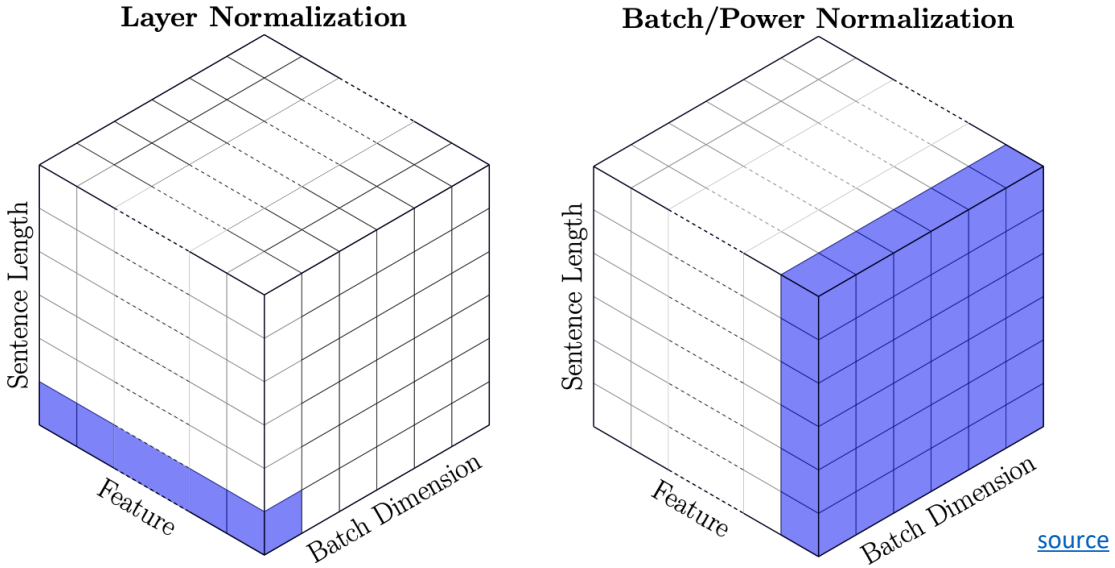
most plausible explanation: smoothening of loss landscape (similar to skip connections)

Layer Normalization

normalization over inputs/features, independently for each data sample

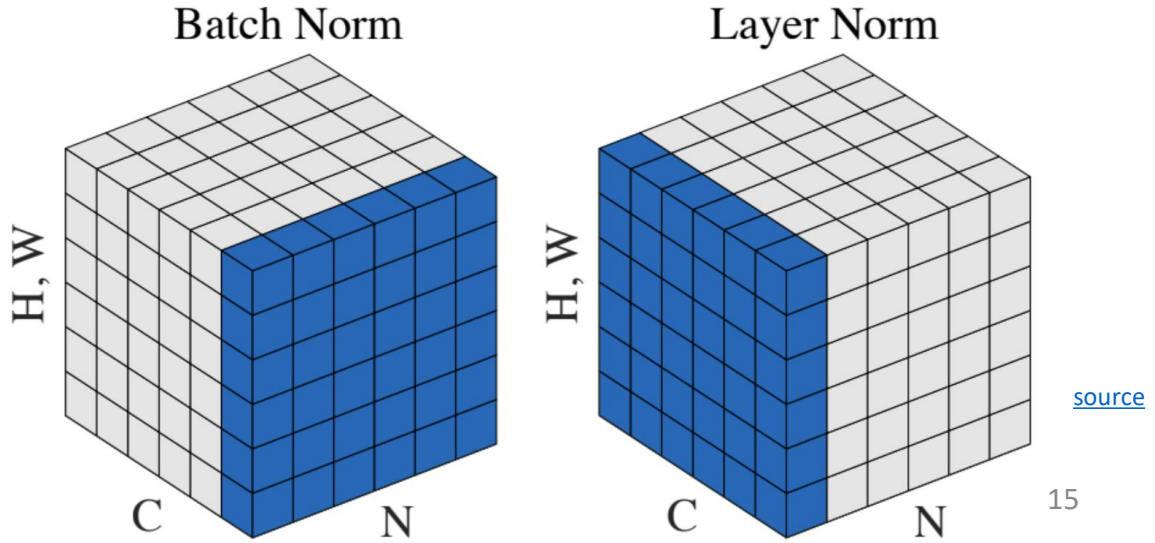
→ mean and variance shared over all hidden nodes of a network layer

batch norm often in computer vision (CNN), layer norm in NLP (variable-sized inputs)



source

transformer:



source

Comparison to Shallow Methods

Feature Engineering vs Feature Learning

shallow learning:

representation encoded in features

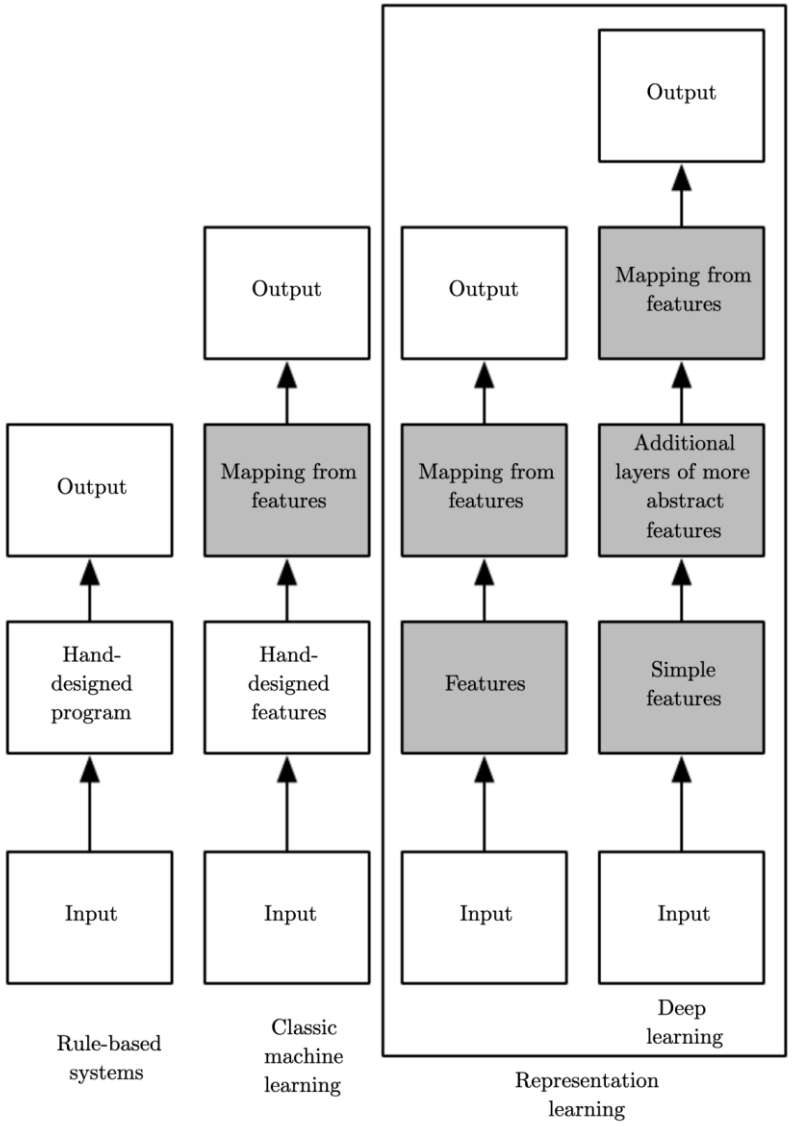
→ feature engineering

deep learning:

representation encoded in network

→ feature/representation learning

(hierarchy of concepts learned from raw data in deep graph with many layers)



source

Tabular vs Unstructured Data

deep learning methods dominate applications on unstructured data (like text or images), but not necessarily on tabular data

typical characteristics of tabular data difficult to handle for deep learning:

- irregular patterns in target function (neural networks require piecewise continuous targets)
- uninformative features
- non-rotationally invariant data (linear combinations of features misrepresent the information)

tree-based models (e.g., gradient boosting) can naturally deal with these situations

Categorical Variables

tabular data usually heterogenous, often with sparse categorical variables (like color of an object)

→ need for an encoding for categorical variables

different possibilities:

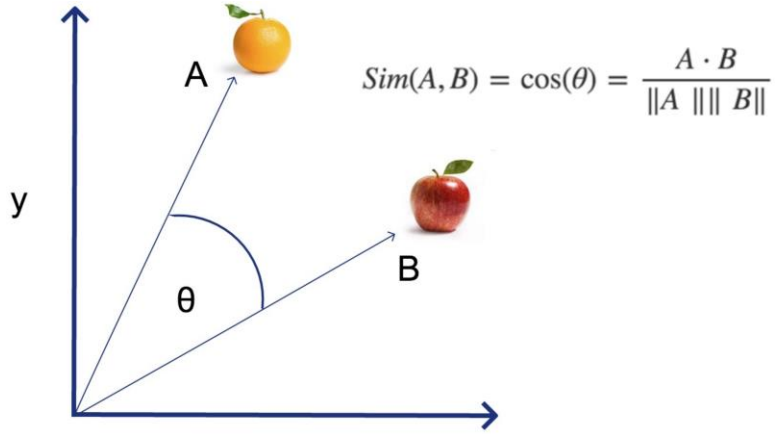
- ordinal encoding (introduces artificial order to unordered categories)
- leave-one-out encoding (use mean of target for given category excluding current row, used in CatBoost)
- one-hot encoding (can suffer from curse of dimensionality)
- embeddings (can also alleviate issue of non rotationally-invariant data)

Embeddings

Vector Representations

embeddings: representation of entities by vectors

similarity between embeddings by, e.g., cosine similarity → semantic similarity



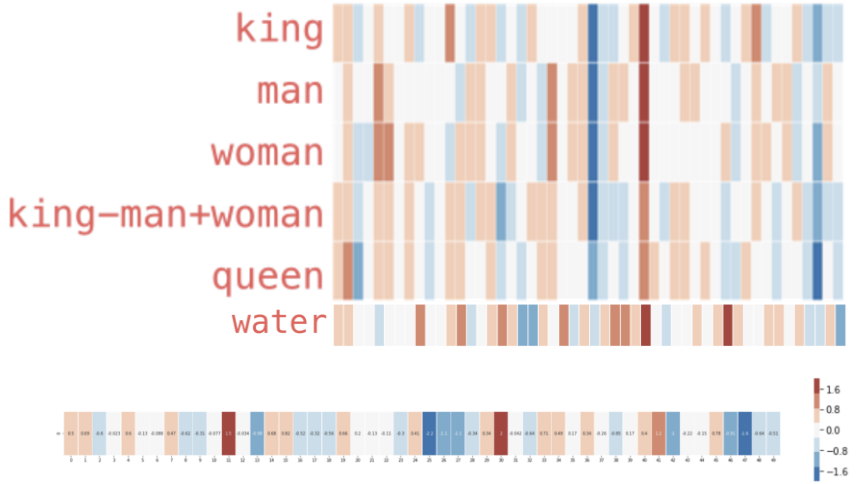
most famous application: word embeddings → associations (natural language processing)

but general concept: embeddings of (categorical) features (e.g., products in recommendation engines)

learned via co-occurrence (e.g., word2vec)

but also direction of difference vectors interesting (analogies):

king − man + woman ~= queen



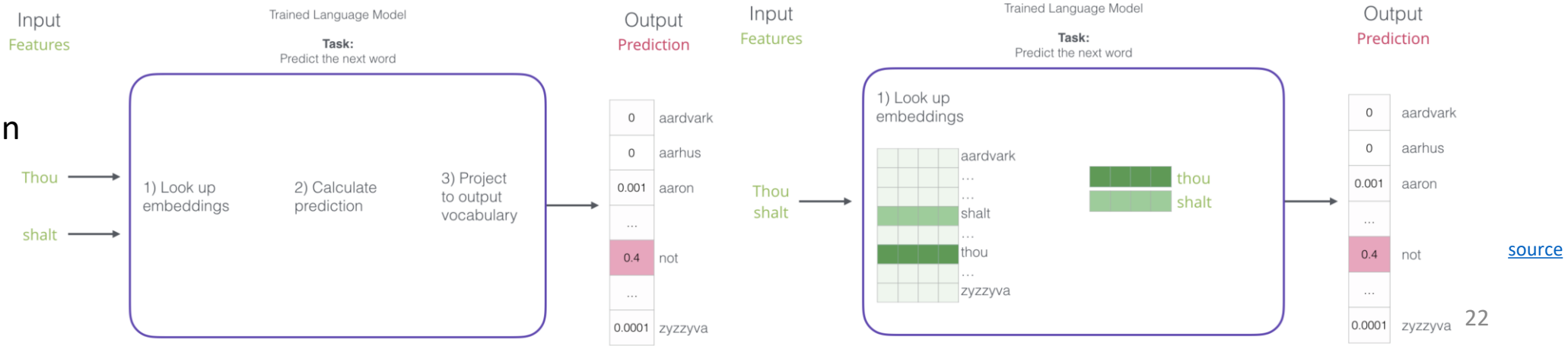
source

Word Embeddings as Part of Language Model

language models contain embedding matrix as part of learned parameters

- can be extracted and subsequently used as pre-trained embeddings for other task
- typically several hundred dimensions for word vectors
- trained on huge data sets (millions in vocabulary)

prominent task:
next-word prediction



source

Neural Language Models

using neural networks: learning distributed representation of words

self-supervised learning: sliding (with some sliding window) over text to generate training data set

learning of embeddings: kind of feature learning

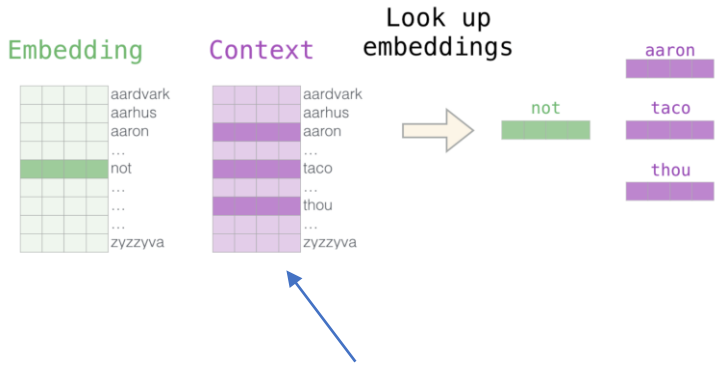
contextually-meaningful embeddings can be learned by means of sequence models (RNN/LSTM, transformer)

word2vec

focus on generating word embeddings, not entire language model → negative sampling:

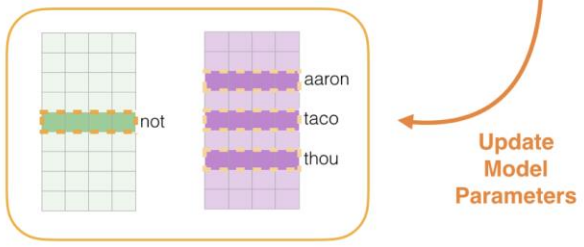
- use input and output words of language model as features, binary target if neighbors (dropping expensive projection to output vocabulary → much faster)
- include random negative samples (samples of words that are not neighbors)

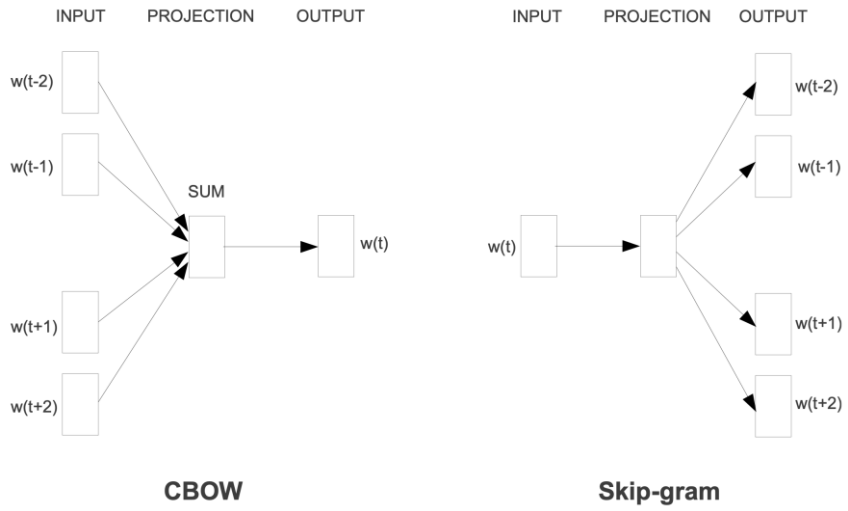
not a deep neural network (just single hidden layer)



"helper" matrix

input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68





Continuous Bag Of Words

source

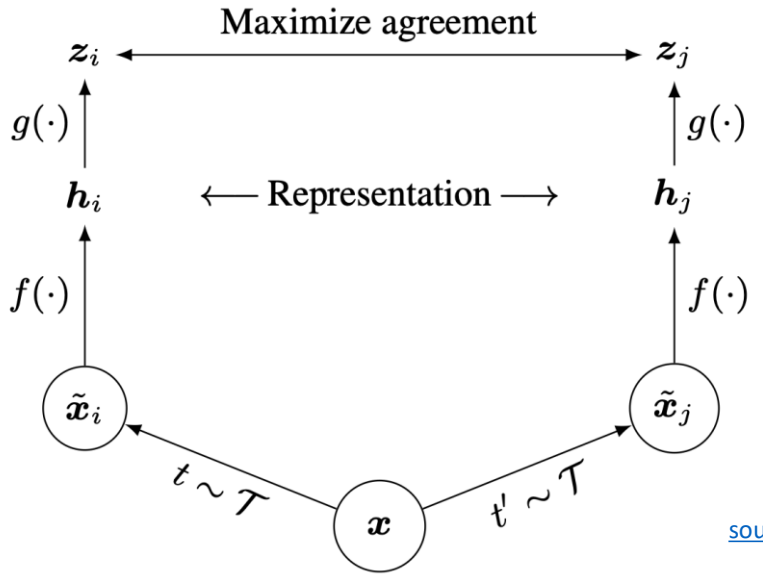
Contrastive Learning

goal: create embedding space in which similar samples are close to each other and dissimilar ones are far apart

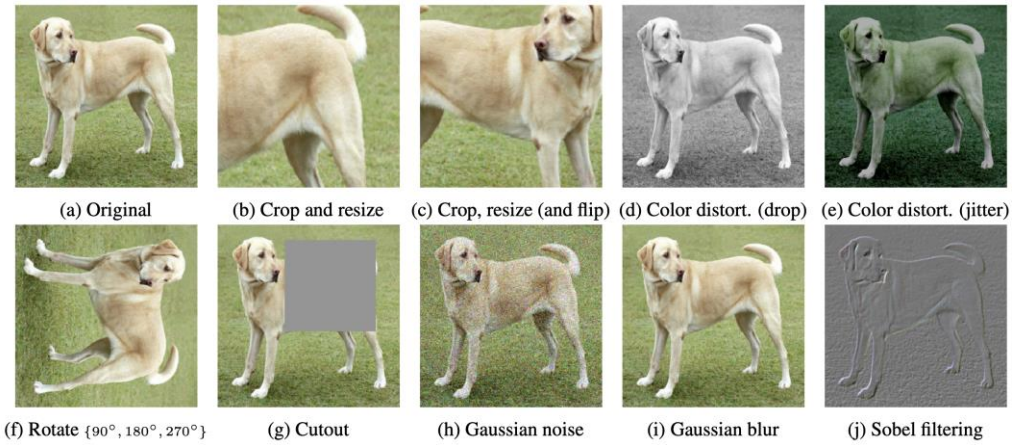
often learned in a self-supervised way

examples:

- natural language processing: word2vec
- computer vision: SimCLR, SimCLRv2 (learning of image representations)



source



Autoencoders

Representation Learning

autoencoders as prime example of representation learning

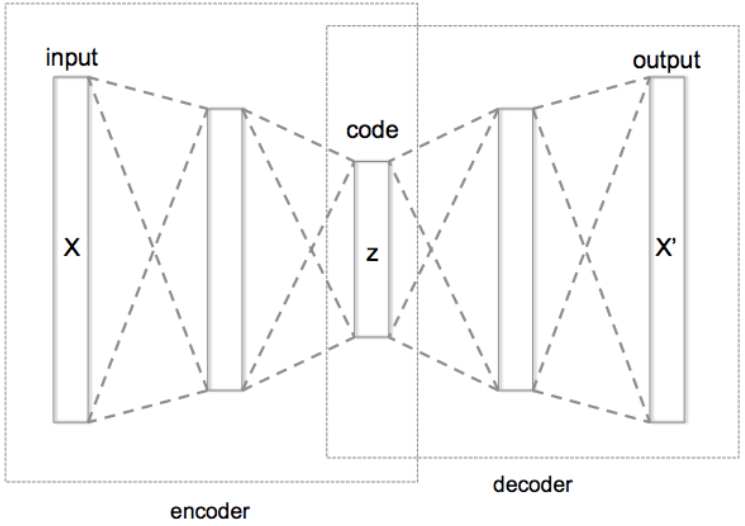
combination of

- encoder: converting input data into different representation (code)
- decoder: converting learned representation back into original format

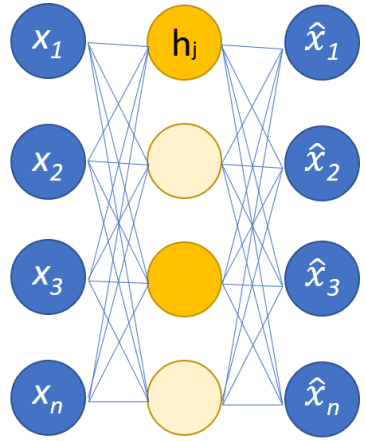
possibilities to avoid simple duplication:

- undercomplete autoencoders: code with smaller dimension (less nodes) than input (generalized PCA)
- sparse autoencoders: sparsity penalty to deactivate hidden nodes (e.g., with help from ReLU activation)

learned in the same way as feed-forward neural networks



from wikipedia



History: Unsupervised Pre-Training

breakthrough in effectiveness of deep learning training in 2006:

Deep Belief Networks introduced idea of greedily initializing each layer by unsupervised learning

(using an energy-based method called Restricted Boltzmann Machine)

→ commonly seen as actual starting point of deep learning wave

only later, ReLU activation functions (and other improvements) enabled effective deep learning without unsupervised pre-training

but unsupervised pre-training still beneficial in context of semi-supervised learning, using large amounts of unlabeled data

Stacked Autoencoders

besides dimensionality reduction, autoencoders can also be used (instead of Restricted Boltzmann Machines) for unsupervised pre-training

→ feature learning (internal distributed representations, high-level abstractions of input data), initializing weights in region near a good local minimum

stacking of autoencoders (or RBMs) → industry adoption:

- object recognition (cat paper)
- speech recognition in industry

stimuli for cat neuron:

source

Recurrent Neural Networks (RNN)

Sequential Structures

speech recognition, natural language processing, time series, …

problem: need to generalize across different points in time (or multiple positions of words within a sentence) and learn from context

idea: parameter sharing across different parts of model

CNNs apply parameter sharing (convolutions) on grid-like structures (including 1-D grids like time series or speech), but this is limited to neighboring inputs.

→ need for approach to learn sequential structures (process input as stream, e.g., speech, rather than one batch, e.g., image)

Back-Propagation through Time

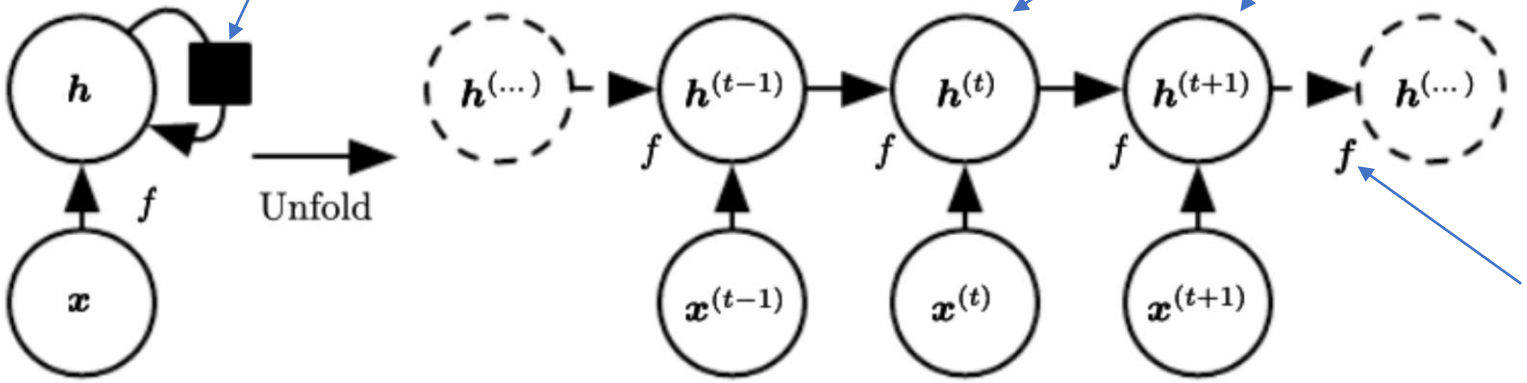
RNN: output of node directed back to input

activation of recurrent nodes at that point in time

interaction with delay of single time step

computational graph with repeated pieces:

hidden layer: several recurrent neurons



input layer: e.g., one-hot encoded words or embeddings

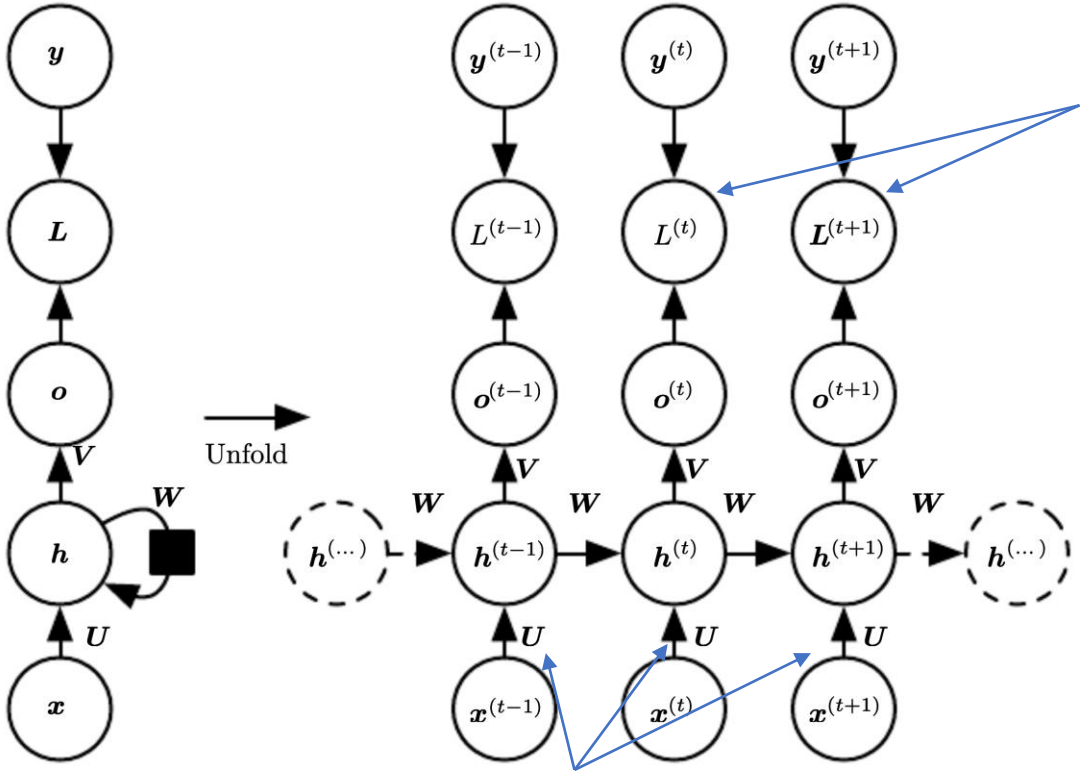
single model operating on all time steps

source

different kind of depth: one recurrence for each sequential step (e.g., word)

Weight Sharing across Time

example: input-output mapping at each time step with recurrent hidden nodes



parameter/weight sharing over time

total loss for sequence: sum of losses over time steps

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}), \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}, \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)}),
\end{aligned}
$$

more popular than ReLU here (exploding gradients)

discrete outputs

source

Further Examples

summarization of sequence:

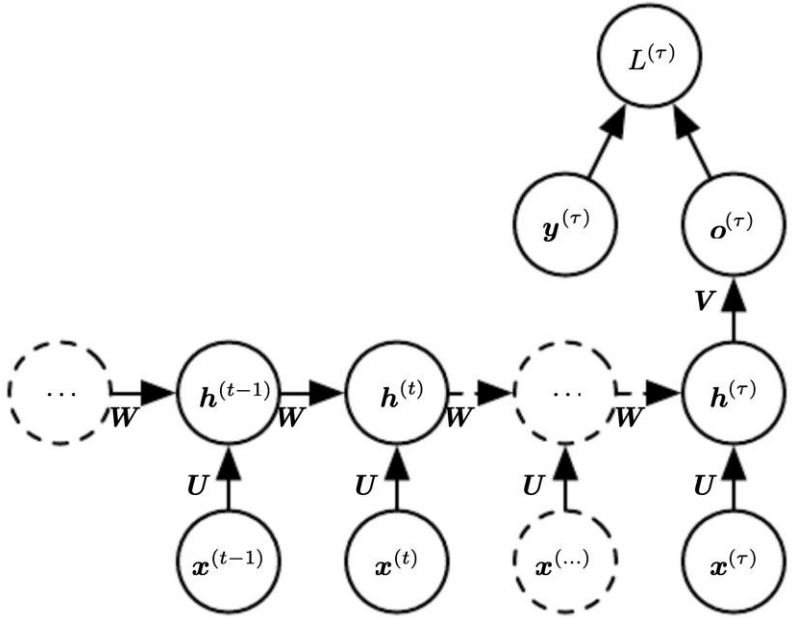
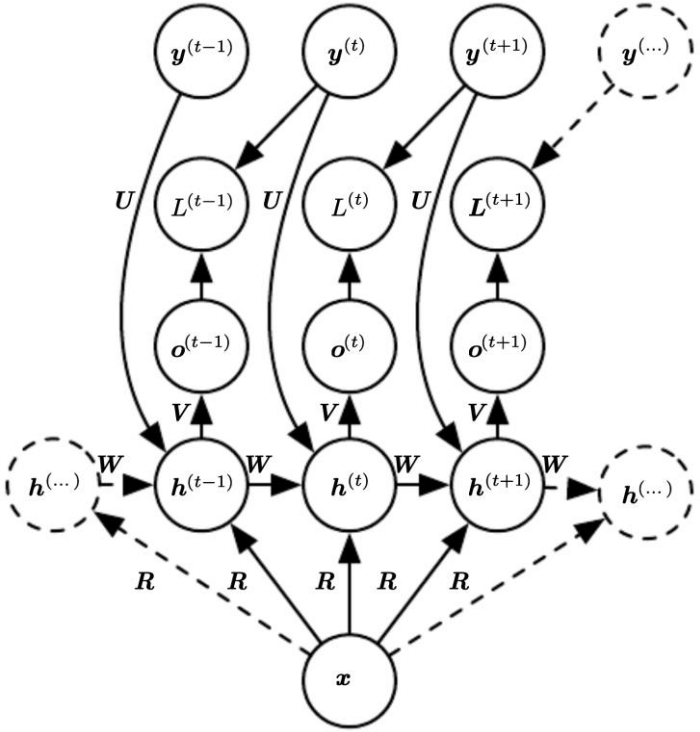
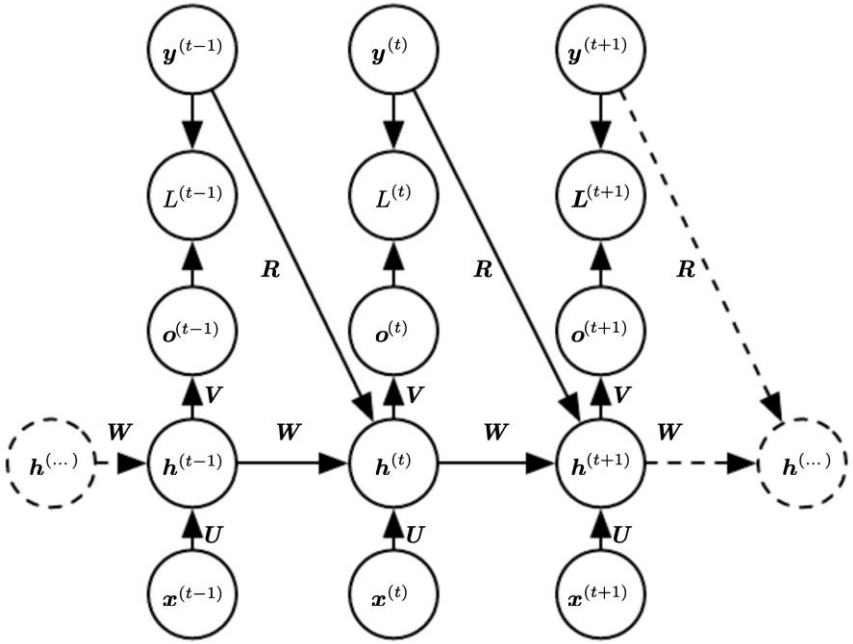


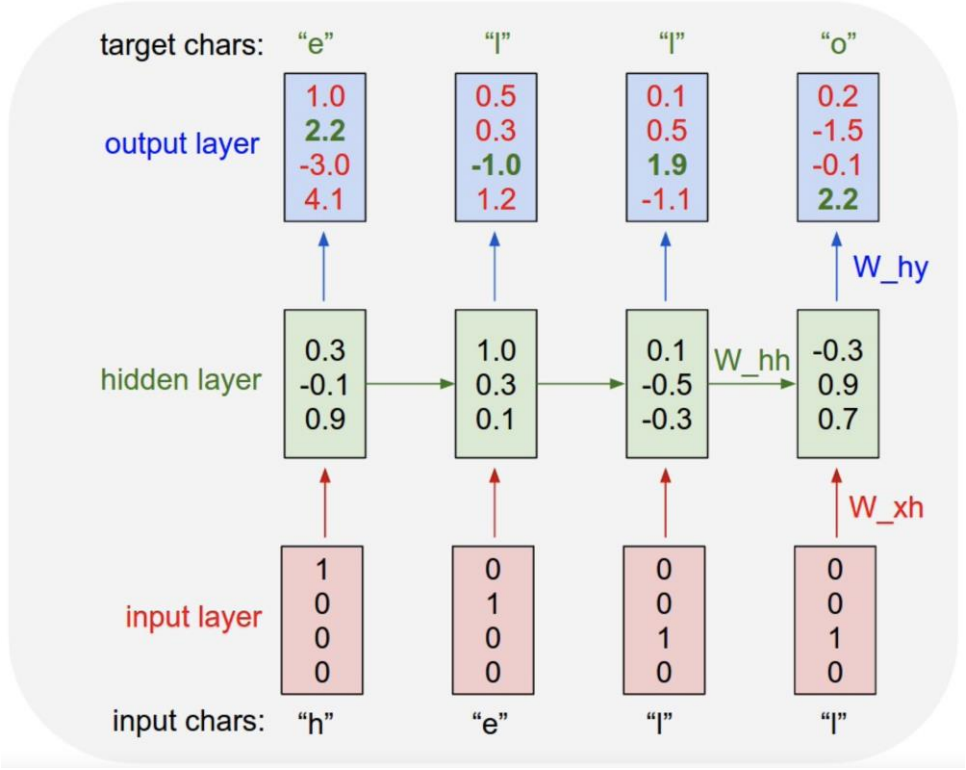
image captioning:



conditional sequence:



Visualization



source

neuron getting excited inside URLs (excited, not excited):

high/low activation:

Cell sensitive to position in line:

```
The sole importance of the crossing of the Berezina lies in the fact
that it plainly and indubitably proved the fallacy of all the plans for
cutting off the enemy's retreat and the soundness of the only possible
line of action--the one Kutuzov and the general mass of the army
demanded--namely, simply to follow the enemy up. The French crowd fled
at a continually increasing speed and all its energy was directed to
reaching its goal. It fled like a wounded animal and it was impossible
to block its path. This was shown not so much by the arrangements it
made for crossing as by what took place at the bridges. When the bridges
broke down, unarmed soldiers, people from Moscow and women with children
who were with the French transport, all--carried on by vis inertiae--
pressed forward into boats and into the ice-covered water and did not,
surrender.
```

Cell that turns on inside quotes:

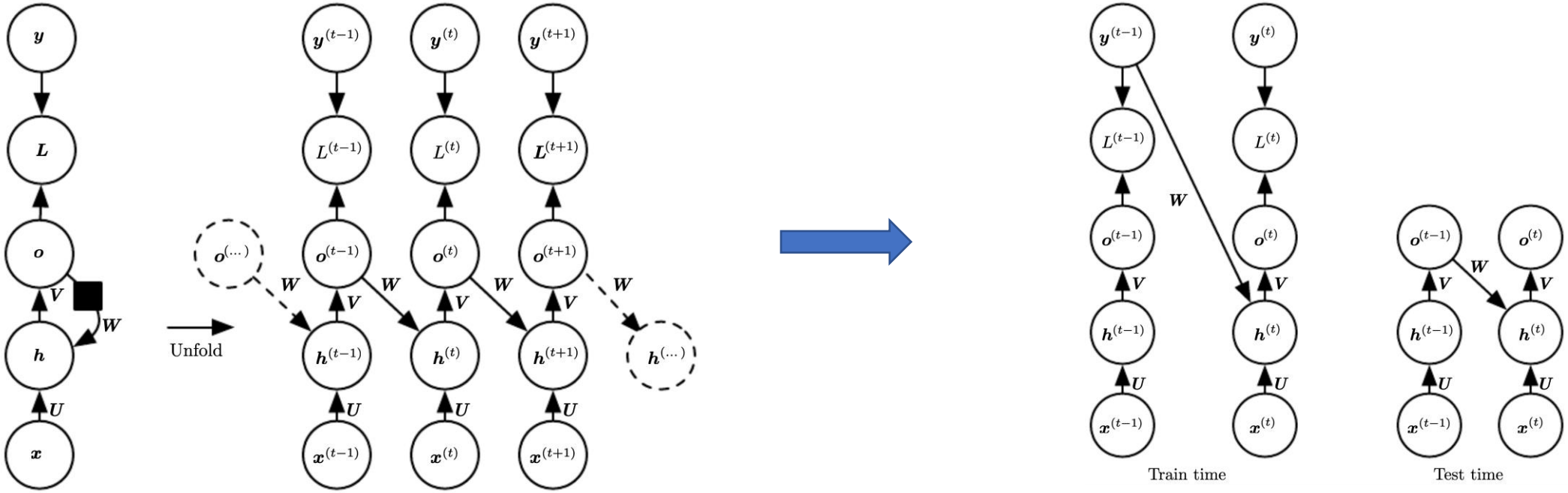
```
"You mean to imply that I have nothing to eat out of.... On the
contrary, I can supply you with everything even if you want to give
dinner parties," warmly replied Chichagov, who tried by every word he
spoke to prove his own rectitude and therefore imagined Kutuzov to be
animated by the same desire.

Kutuzov, shrugging his shoulders, replied with his subtle penetrating
smile: "I meant merely to say what I said."
```

Cell that robustly activates inside if statements:

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
        siginfo_t *info)
{
  int sig = next_signal(pending, mask);
  if (sig) {
    if (current->notifier) {
      if (sigismember(current->notifier_mask, sig)) {
        if (!(current->notifier)(current->notifier_data)) {
          clear_thread_flag(TIF_SIGPENDING);
          return 0;
        }
      }
    }
    collect_signal(sig, pending, info);
  }
  return sig;
}
```

Teacher Forcing

for feedback from output to hidden layer: instead of feeding model output back into itself, use target values directly



source

long-term memory: weights
short term memory: activation patterns

Gated RNNs

issue with long chain of gradients through recurrences: vanishing (mainly) and exploding gradients in back-propagation

→ need to focus on important sequence elements

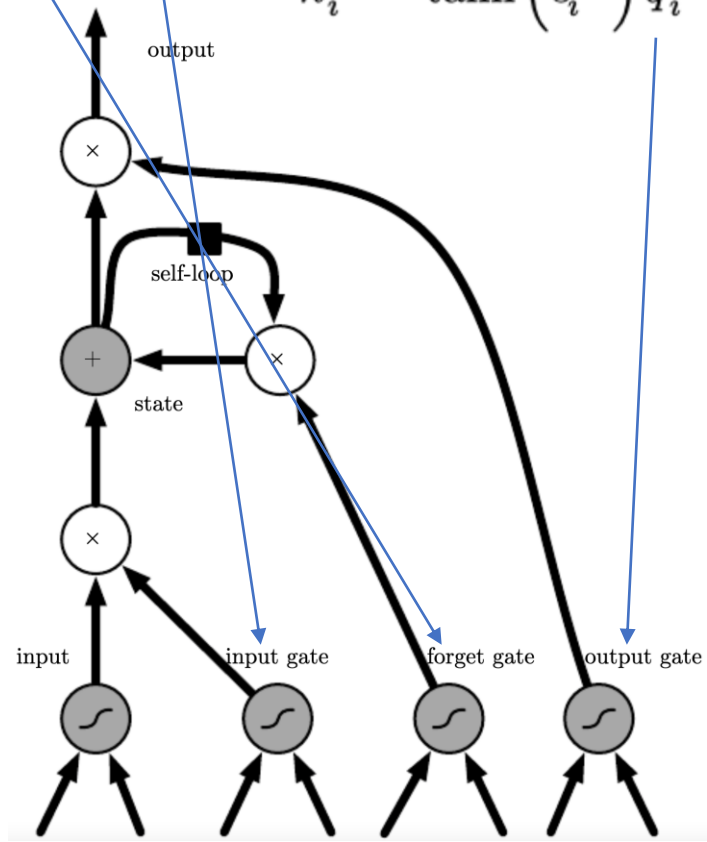
replace usual recurrent hidden nodes with long short-term memory (LSTM) cells with internal recurrence (self-loop)

- linear self-loop in addition to outer recurrence of RNN: error carousel preserving (→ long) short-term memory (i.e., activation patterns)
- sigmoid activations with independent weights for input, forget, and output gates

other prominent gated RNN: gated recurrent unit (GRU)

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right)$$

$$h_i^{(t)} = \tanh \left(s_i^{(t)} \right) q_i^{(t)}$$



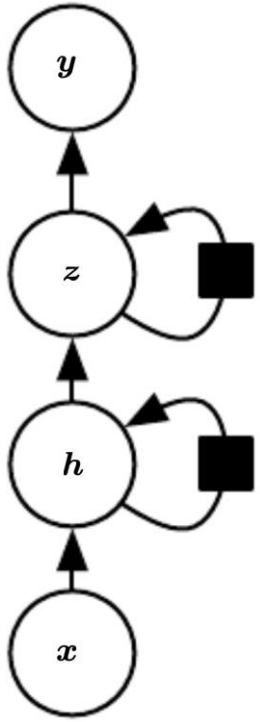
source

Issues with RNNs

- especially LSTMs very computationally expensive (with lots of parameters)
- transfer learning (using pre-trained network layers on new task, e.g., popular for CNNs) difficult

(hierarchical) representation learning: go deep by stacking several layers of recurrent nodes → worsening efficiency

(self-)attention and transformers to the rescue …



source

Literature

papers:

- ResNet
- A Neural Probabilistic Language Model
- word2vec
- training of Deep Belief Nets

blogs:

- The Illustrated Word2vec
- Karpathy on RNNs

Black-Box Models

To build trust in AI systems, individual predictions/actions need to be fully transparent, i.e., explainable.

Unfortunately, complex models like deep learning methods are difficult to interpret.

→ need for model-agnostic methods to explain black-box models

examples: local surrogates (LIME), Shapley values (SHAP)

overview