Deep Learning book

I read it so you don't have to

Outline

- intro (ch1, 20p)
- part 1 (ch2-5, 160p): **Math & ML**
  - LA, prob, numerical
  - ML in general
- part 2 (ch6-12, 300p): **Deep Learning practical**
  - feedforward
  - regularisation, optimisation
  - CNN & RNN
  - practical & applications
- part 3 (ch13-20, 230p): **Deep Learning research**
  - did not read yet (autoencoders, monte carlo, prob models, generative)

In general

- Recent: 2016
- Very good book, clear and extensive, contains a lot
- Advanced: start with simpler introductions (NN&DL book) first
- Math-heavy, for big picture skip some sections

- Now quickly skipping over stuff from Nielson's book;
  hopefully it will feel like a summary of our 10 weeks

ch1: big picture

Enables of DL:

1. more data -> beter generalisation
2. more compute power -> bigger networks
3. algorithmic tweaks: (mostly due to) cross-entropy + RLUs

Learning hierarchies of concepts. Inspired by neuroscience, but simpler and focus on engineering.

ch2-4: math

Linear algebra:

...

Probability:

...

Numerical:
Practical problems when implementing math:
underflow (rounding to 0), overflow (rounded tot \inf), poor conditioning
→ reason for log() in many implementations

ch5: machine learning

Designing a ML algorithm:

1. Dataset specification
2. Cost function: estimator + regularisation terms
3. Optimisation procedure: directly, iteratively (gradient descent)
4. Model

Challenges motivating Deep learning:

1. Curse of dimensionality: interesting problems have huge D's, generalisation is hard with few data
2. Local constancy & smoothness: hierarchical + holistic
3. Manifold learning: find interesting low-D manifolds in huge D space

ch6: deep feedforward networks

1. Dataset specification: 0-1, known answer
2. Cost function: cross-entropy or LSE + regularisation (simple models)
3. Optimisation procedure: iteratively using minibatch SGD
4. Model
   - architecture
   - activation functions

Interesting description on algorithms used in DL libraries (computational graph based).

ch7: regularisation

Tricks for DL:

- **parameter norm penalties**: add parameter size penalty to cost function; usually on weights only; has effect of slowly decaying weights on every update
- **dataset augmentation**: simple transformations on or adding noise to data; highly problem-dependent but can cause huge improvements
- **early stopping**: use parameters of lowest validation error, stop when not improving (kind of treating # epochs as a hyperparameter to learn)
- **parameter sharing**: force sets of parameters to be equal (convnets); allows dramatic increase of network sizes for same amount of data (less parameters to learn)
- **sparse representations**: most neurons shouldn't fire; added to cost function; biologically inspired
- **ensemble**: averaging independently trained (possibly quite different = boosting) neural networks, possibly trained on different data subsets (bagging); very powerful and reliable
- **dropout**: disable half of input and hidden neurons for each mini-batch; makes representation more robust and holistic; kind of bagging for many networks at once but with shared parameters; kind of adding noise to hidden layers; very cheap and generic (FFD, RNNs, Boltzmann) on medium-sized datasets, but does need a bigger model and more training

ch8: optimisation

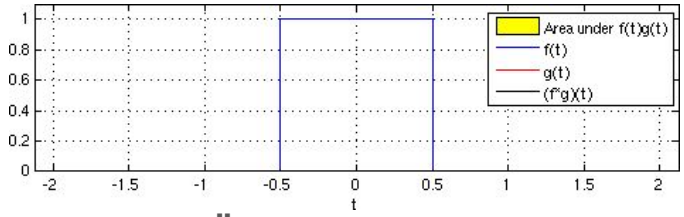
ML = optimisation on indirect targets (unseen test data)
= optimisation + generalisation tricks

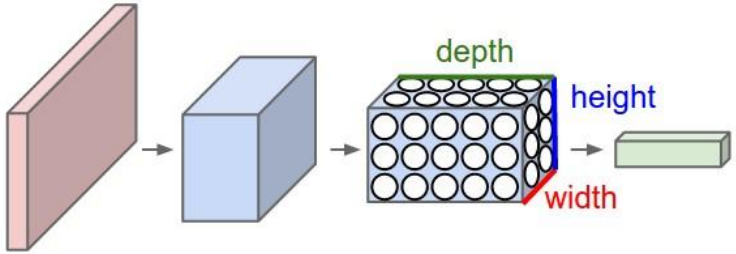
Loss function is surrogate for what we actually care about

Iteratively using SGD. Improvements:

- decreasing **learning rate**
- **adaptive learning rate** (often per dimension = parameter) SGD: AdaGrad, RMSProb, Adam
- **momentum-based**: accelerate learning at places of high curvature or consistent or noisy gradients; can be seen as numerical physical simulation
- **second-order gradient** approximators: Newton's Method (very expensive O(k^3), Hessian), conjugate gradients, BFGS (iteratively estimate Hessian^-1)
- other tricks: **batch normalisation** (solves not-so-independent-updates problem); **polyak averaging** (average visited locations to find valley); **supervised pretraining** (start on simpler problem or with simpler model; e.g., subset of all layers, or add layers in between); **continuation methods** (iteratively solve less blurred versions of cost function; similar to simulated annealing); just choose **simpler models** (RLUs, LSTMs, connections that skip layers).

ch9: CNNs





"Shared weights over space"

Handy for grid-like data: time-series, images, voxels, video, kinematics

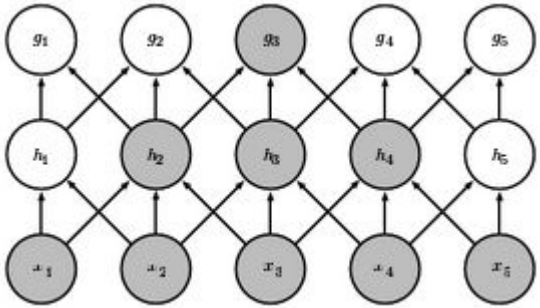
CNNs leverage three important ideas:

1. **sparse interactions**: exploit localised information; enables huge scaling (orders of magnitude): O(m.n) -> O(k.n) runtime; information can still reach all output neurons, but travels (much) slower outwards towards deeper layers.
2. **parameter sharing**: reuse kernel (window function) for all neurons; weights and usually biases too: O(m.n) -> O(k.n) parameters = storage and statistical efficiency
3. **equivariant representations**: translation invariant (but not scale or rotation)

ch9: CNNs

"Shared weights over space"

Information slowly flows sideways ("strong prior"):

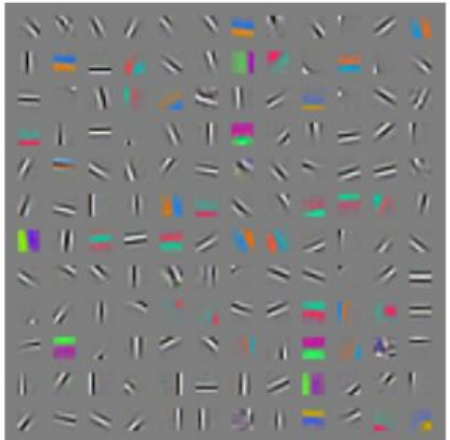


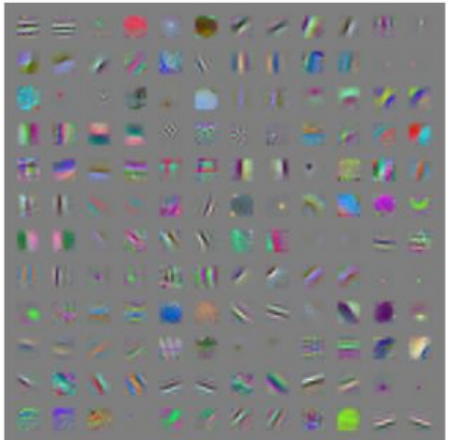
Can also work on varying size data (e.g., adaptive pooling window size).

Pooling:

1. over window: small translation-invariance
2. over feature maps: other learned transformation invariance (rotation, scale, ..)

Features learned (1st hidden layer). Often resemble neurological research. Can sometimes be reused for new models.

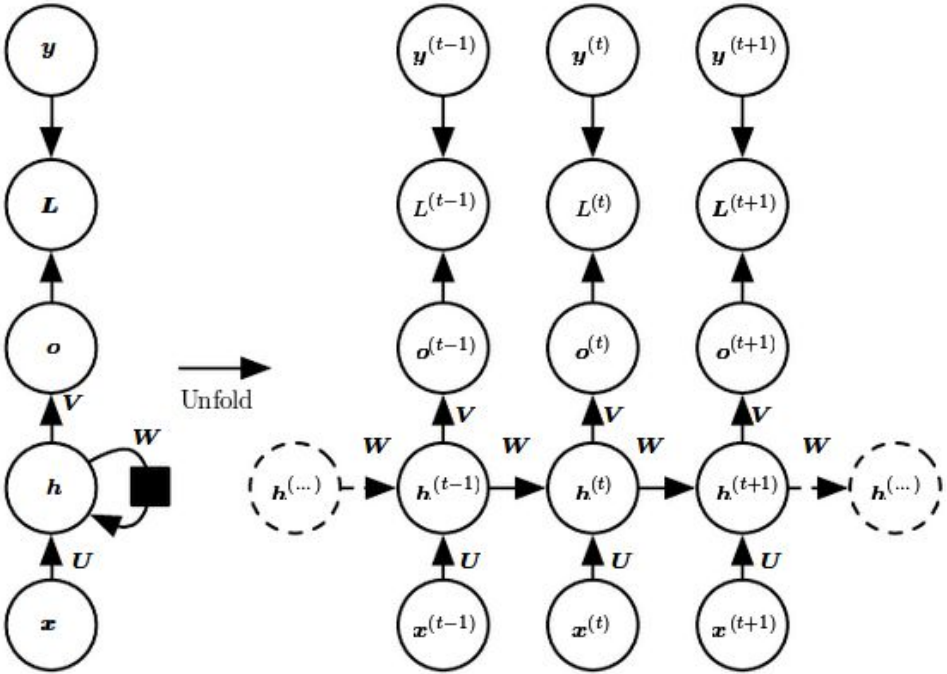




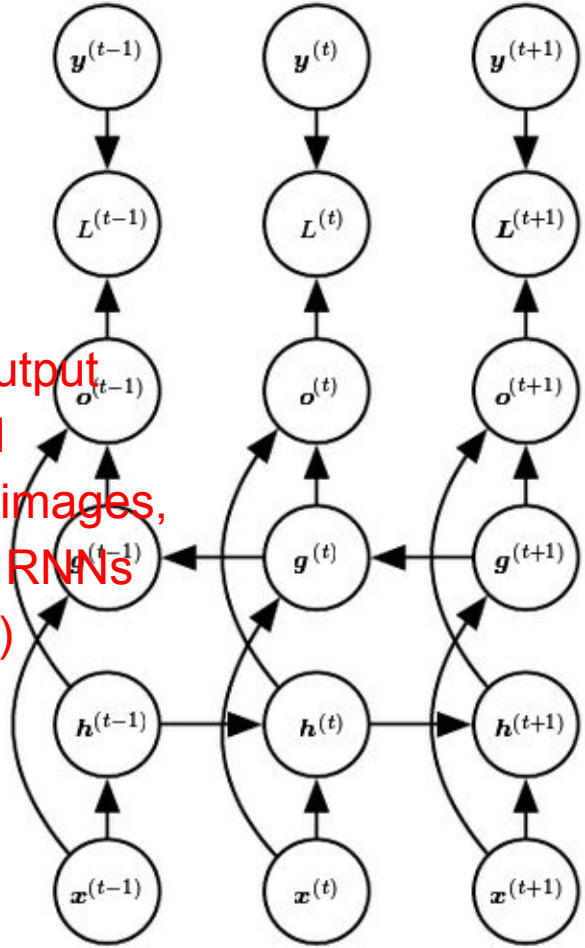
ch10: RNNs

"Shared weights over time"

neural net vs computational graph

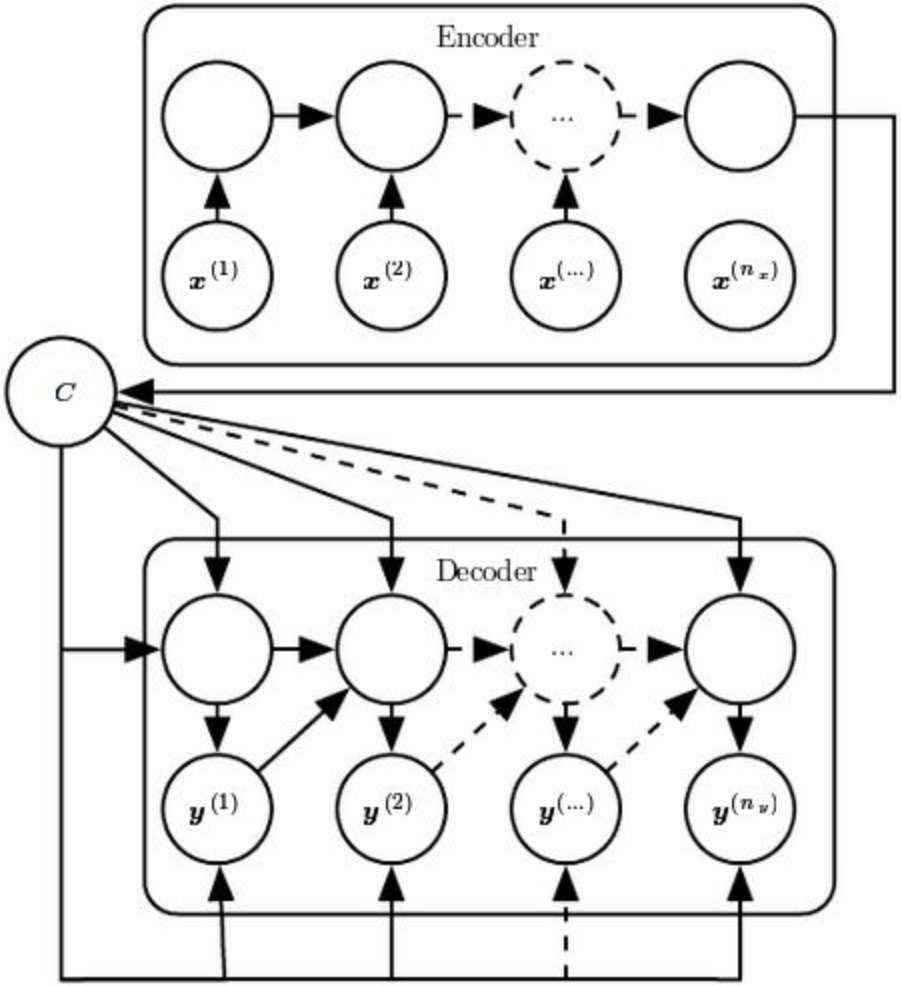


bidirectional (output depends on full sequence); for images, can combine 4 RNNs (up,dwn,lft,rght)



ch10: RNNs

variable-length
input and output:
use status vector C
as "meaning vector"

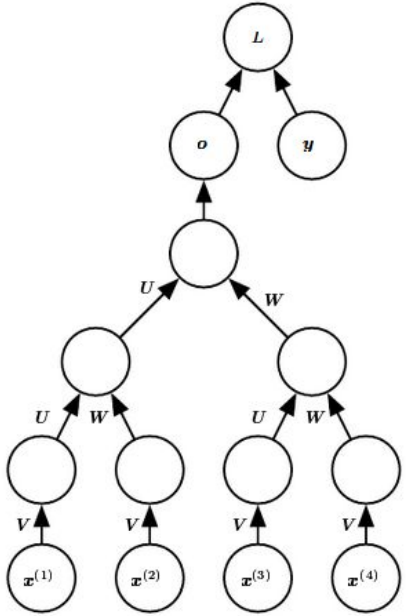


ch10: RNNs

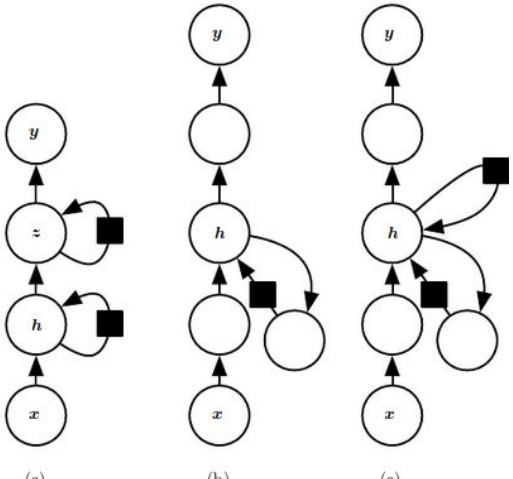
long-term dependencies

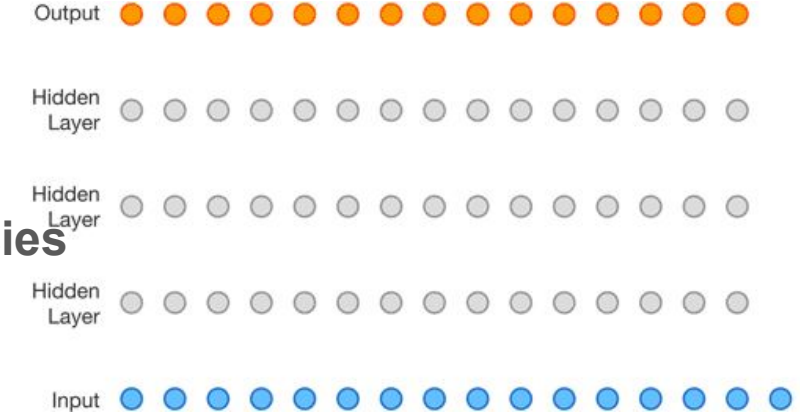
various architectures:

hierarchical

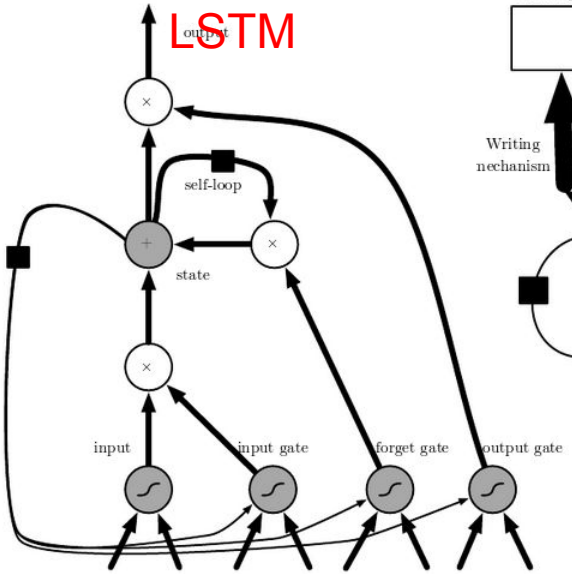


various deep RNNs

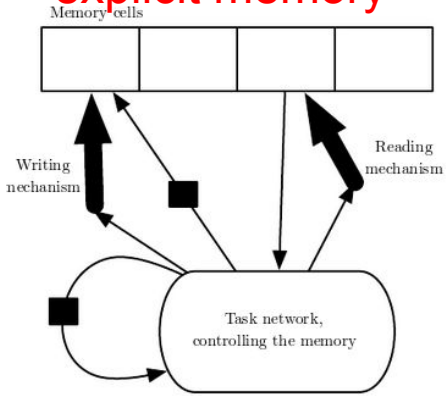




LSTM



explicit memory

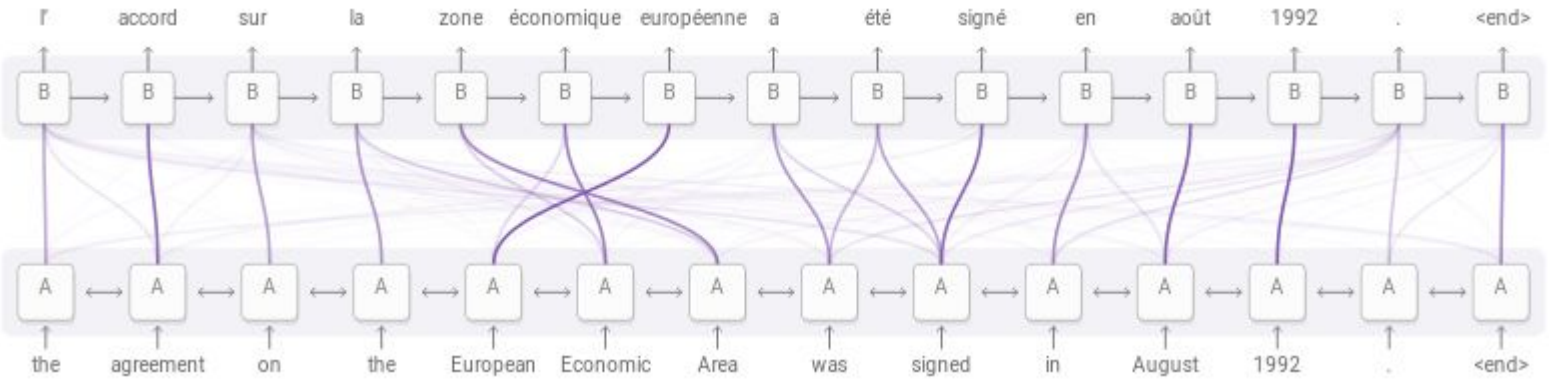


other:

- Echo state networks
- Multiple time-scales / "Leaky Units"
- optimisation tricks

ch10: [extra] attention-based

Upcoming: let the network decide what to look at.
Can be used for stacked networks, with explicit memory, etc.



ch11: Practical Methodology

The most important ML skills to learn are practical matters (experience + "Art").

1. **Determine your goals**: Error metric (accuracy, precision+recall or F-score, coverage, custom metrics like click-through rates) + Target value
2. **Get end-to-end pipeline working ASAP**: Start with baseline models (FNN, CNN or RNN). Use ReLUs, SGD with momentum and decaying learning rate, and early stopping. Try batch normalisation. Add regularisation. Try models trained on similar problems. Try unsupervised learning if relevant (NLP).
3. **Diagnose bottlenecks**: overfit, underfit, bugs, data problem?
4. **Repeatedly make incremental changes**:
   - *more data*: once results on the training data are OK but poor on the test set
   - *adjust hyperparameters*: manual (increase model capacity, then increase regularisation) and/or automatic (grid-search)
   - *change algorithms*

ch12: applications

Real-world applications are usually run on **large-scale distributed networks with** general purpose (2007+) **GPUs**, or even DL-specific hardware. They use highly optimised GPU code (DL libraries). SGD can partly be **parallelised**.

Computer vision: most popular applications in DL, mostly object recognition and some image generation.

Speech: used to be HMMs + GMMs, recently switched to DL + HMMs.

NLP: used to be n-grams, now mainly RNNs / encoder-decoder networks. Needs some tweaks for the one-shot input (word embeddings) and high-D outputs (importance sampling). Research into attention-based translators.

Other: **Recommender systems** (very sparse data, needs to generalise; for long-running systems, reinforcement learning), **knowledge-based systems** (learn facts and generalise; explicit memory)

Part III

Research part. Not read (yet).

Includes exciting frontiers (autoencoders, monte-carlo, reinforcement learning?) but will likely be outdated soon.