Introduction to Machine Learning and Deep Neural Networks for Scattering Science

Bill Triggs

Laboratoire Jean Kuntzmann, Grenoble, France

Bill.Triggs@imag.fr

Workshop on AI Applied to Photon and Neutron Science
Institut Laue-Langevin, Grenoble, France
12 Nov 2019

Machine Learning

Creating empirical models that capture the typical behaviour of complex data in some limited domain

- essentially flexible data modelling + statistical parameter estimation

Artificial Intelligence

Engineering artificial systems that have near-human-level problem solving capabilities in some limited domain

- essentially a systems integration problem not a data modelling one
- typically uses ML + other techniques as components

This talk

- Only discusses Machine Learning, not Artificial Intelligence
- Only covers a few topics within the huge field of ML
  - hopefully ones relevant to neutron / X-ray scattering applications...
- Emphasizes basic principles and intuition, not technical details.

Common ML Tasks 1 – Predictive Modelling

- Given input samples, predict corresponding output values.
- Goes with **supervised learning:** training samples come annotated with desired output(s). Training minimizes a "**loss**" function on the samples.

Examples

- **Discrimination / classification:** outputs are categorical, *e.g.* "class" of input sample.
- **Regression**: outputs are continuous, *e.g.* predicting intensity or response strength.
- **Pattern detection / segmentation:** outputs include locations / geometry
  - *e.g.* find / localize / segment occurrences of given object(s) / pattern(s) in images by scanning "is there pattern here?" classifier over image and postprocessing its outputs
- **Metric learning:** given pairs of input samples, return continuous measure of "distance" or "dissimilarity" between them.
- **Comparison / ranking:** given several input samples, output relative ordering / preference scores for them.

Common ML Tasks 2 – Descriptive Modelling

- Learning to *empirically characterize* input samples in some feature space, not predict specific outputs for them.
- Goes with **unsupervised learning** – training samples are not annotated.
- Useful for:
  - producing intermediate “feature representations” for later tasks
  - data visualization, guidance for improved modelling, . . .
  - data cleaning / anomaly detection

Examples

- **Clustering:** finds proximity-based groupings of samples in feature space (*e.g.* k-means)
- **Dimensionality reduction:** finds low-dimensional embeddings that characterize the samples and/or display their range of variation in feature space.
- **Distribution / support learning:** finds empirical probability distributions for the samples, or geometric models for the regions that they occupy in feature space.

Common ML Tasks 3 – Modelling Behaviours

Reinforcement Learning learns behavioural strategies for agents (robots, game players, . . .) searching for rewards / avoiding penalties in evolving environments

- The agent takes "actions" that produce (stochastic) state evolutions $\mathrm{action}_t \to \mathrm{state}_{t+1}$
- It tries to minimize its **regret** (expected total future loss) $\mathsf{E}[\sum_{t'=t}^{\infty} \mathrm{loss}_{t'}]$, by learning:
  - an **action policy** – a direct mapping $(\mathrm{state}_t, \mathrm{inputs}_t, t) \to \mathrm{action}_t$
  - -or- a **valuation function** estimating the expected future regret contribution for each possible state at some horizon $t+k$. This is used to explicitly search for the best action at time t.
  - -or- **both**: a policy to guide the search from t to $t+k$ and a valuation function to estimate the regret from $t+k$ onwards.

Examples

- Recommender systems & targeted advertising – a $100 billion industry
- Game-playing agents – *e.g.* the AlphaGo program used deep neural net policy & valuation functions to defeat the world's best Go players
- Robotics, planning systems, . . .

Common ML Tasks 4 – Modelling Competition

The above tasks traditionally assume passive, time-invariant environments:

- the statistical properties of samples and outputs remain constant over time
- nothing in the environment is actively trying to select samples / actions / strategies that will defeat the method's best efforts.

Adversarial Learning provides strategies for handling "combative" environments that actively try to defeat the method

- Central to areas like game playing, but useful elsewhere
  - results are more robust to unmodelled effects, but less 'fine tuned' to any given distribution
- Training is *much* harder because training distributions are not fixed
  - The solution is a saddle point (Nash equilibrium) of a two-agent objective function not a local minimum of a single-agent one. Finding saddle points is delicate.

Examples

- The AlphaZero Go program used adversarial training to defeat AlphaGo.
- Adversarial deep nets are good at learning to generate realistic images or signals.

Common ML Tasks 5 – Reducing / Re-using Supervision

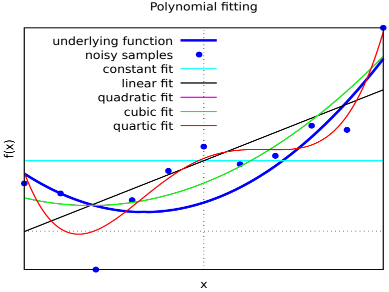
Annotating large training sets is time-consuming! – Can we do more with less?

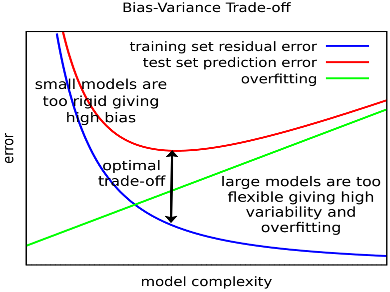
- **Semi-supervised learning:** learning an output predictor when only some of the training samples have "desired output" annotations
  - the unannotated samples help to characterize the underlying distribution in feature space
- **Weakly supervised learning:** learning a full output predictor from training samples with incomplete annotations
  - *e.g.* learn to locate dogs in images from training images annotated with tags like "dog" but not with their image locations
  - missing annotation information corresponds to *latent variables* in statistics.
- **Transfer learning:** adjusting an existing model to cope with new output classes/tasks or changed feature space distributions
  - the existing model (trained on many samples) provides regularity so comparatively few new training samples are needed
- **Multi-task learning:** learning a joint representation for several different tasks
  - *e.g.* a common "feature space" + distinct "output layers"

Bias-Variance Trade-Off – The Central Dilemma of ML

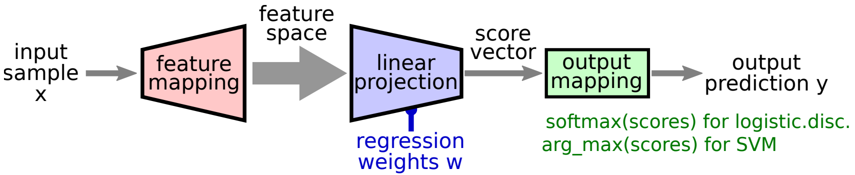
The more flexible a model is, the better it can fit the training data but the more it will *overfit* to noise in the data

- Overly rigid models have high *bias*, giving systematically poor predictions
- Overly flexible models have high *variance* / unstable parameter estimates
  - **overfitting** to training set produces low training residuals but poor performance on test set
  - usually controlled by explicit **weight regularization** or by **adding noise** to the fitting process





Basic Discriminants – Logistic Regression and SVM

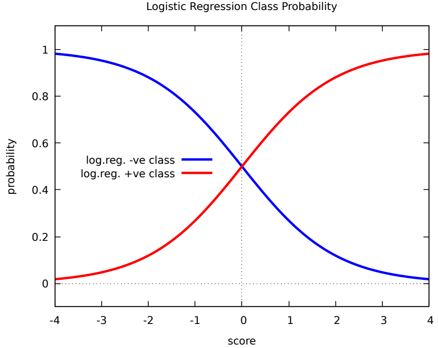


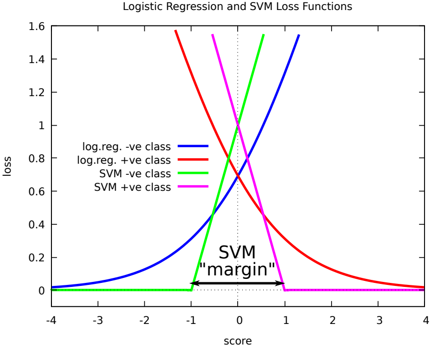
- Input samples are mapped into a high-dimensional **feature space** by some user-specified feature mapping
- A **score vector s** is found by linear projection: $\mathbf{s} = \mathbf{weights} \cdot \mathbf{features} + \mathbf{bias}$
  - the weights are the scorer's learnable parameters
- The scores are translated into output values
  - Logistic regression outputs class probabilities: $p(\text{class } c) = \text{softmax}_c(\mathbf{s}) \equiv \frac{\exp(s_c)}{\sum_c \exp(s_c)}$
  - SVM outputs class ID's: $\text{class} = \arg\max(\mathbf{s})$
  - Logistic regression provides calibrated output probabilities for overlapping classes but does not naturally handle non-overlapping classes
  - SVM handles both overlapping and non-overlapping classes, but doesn't give probabilities

Logistic Regression and SVM Loss Functions

The corresponding loss functions for weight learning are

- logistic regression: $\mathrm{loss}_c = \mathrm{lmax}(\mathbf{s}) - s_c$ where $\mathrm{lmax}(\mathbf{s}) = \log\left(\sum_c \exp(s_c)\right)$
- SVM: $\mathrm{loss}_c = \max(s_1, \ldots, s_c - 1, \ldots, s_n) - (s_c - 1)$ where “1” is the SVM **margin**





Both methods are often used as output layers, *e.g.* for deep neural nets

Feature Mappings

Many kinds of feature mappings are available to convert diverse forms of learning inputs to vectors of real features

- text, images, *etc*., but also graphs, molecular structures, ...

Simple examples are

- **one hot** encodings $(0, ..., 0, 1, 0, ..., 0)$ for categorical variables
- **bag of word** (vocabulary histogram) encodings for text

Kernels

- Kernels are functions $k(\mathbf{x}_1, \mathbf{x}_2)$ giving a notional "inner product" or "similarity score" between pairs of samples
- They are widely used in ML and stock kernels exist for many kinds of inputs
- Given fixed reference samples $\mathbf{x}_1, ..., \mathbf{x}_n$ (*e.g*. the training set), input samples $\mathbf{x}$ can be represented as vectors using the feature mapping $\mathbf{x} \to (k(\mathbf{x}_1, \mathbf{x}), ..., k(\mathbf{x}_n, \mathbf{x}))$

Reproducing Kernel Hilbert Spaces are function spaces built over kernels

Ensemble Methods 1 – Mixture of Experts

Ensemble Methods

- These combine a set of weak prediction models to produce a stronger one
  - this often works surprisingly well, even if the individual models are hopelessly weak
- Conditions:
  - the weak models must be *complementary* (the space of possibilities is covered)
  - they must be *diverse* (having different weaknesses, not similar ones)
  - the combination method / averaging weights must be well chosen

Mixture of Experts

- MoE methods learn spatial weightings / an output-combining layer for a predefined set of predictors
- Even very strong methods like Deep Nets often benefit – *e.g.* combining focused DN “domain experts” with shallow but diverse “breadth coverage” ones

Ensemble Methods 2 – Random Forests and Boosting

Boosting

Greedy sequential selection of new weak model(s) to include to improve the ensemble

- works over a space of weak models, often by randomly sampling new ones to try
- **Gradient boosting** chooses new models to approximate gradient descent w.r.t. ensemble performance

Boosted Regression Forest

Regression Tree: send multi-variable input samples through a short hierarchy (tree) of single-variable splits, output a real value attached to the leaf node reached

Regression Forest: apply a set of regression trees to input sample, sum outputs

Boosted Regression Forest: choose trees and weights by boosting

- A good approach for predictors over messy tabular / data-mined data: samples with many discrete (or binnable real) attributes, frequent missing/erroneous values, . . .
- Usually not competitive for predictors over regular signals or images

Bayesian Methods

Bayesian methods represent all phases of reasoning in terms of probabilities

- the training data **T**, model choices/parameters **w** and outputs **y** are all characterized in terms of assumed probability distributions – typically highly structured ones
- learning = using Bayes' rule to evaluate the posterior distribution over models **w**

$$p\left(\mathbf{w}\,|\,\mathbf{T}, \text{model\_prior}\right) \;=\; \frac{p\left(\mathbf{T}\,|\,\mathbf{w}\right)\, p\left(\mathbf{w}\,|\,\text{model\_prior}\right)}{p\left(\mathbf{T}\,|\,\text{model\_prior}\right)}$$

- prediction = using the model posterior to evaluate the posterior distribution over **y**

$$p\left(\mathbf{y}\,|\,\mathbf{T}, \text{model prior}\right) \;=\; \int_{\mathbf{w}} p\left(\mathbf{y}\,|\,\mathbf{w}\right)\, p\left(\mathbf{w}\,|\,\mathbf{T}, \text{model\_prior}\right)\, \mathbf{dw}$$

  – to use this we can average over **y** values or find the mean/mode/variance of their posterior

Bayesian Inference

Bayesian computations can seldom be completed in closed form, so approximate inference methods are needed

Monte-Carlo Methods

- Inference by generating and averaging over random samples
  - gold-standard results, but often slow
- Various flavours
  - complex models typically use Markov Chain MC
  - simple MC and quasi-random methods tend to get forgotten but they are often more efficient
  - for MCMC on easily-differentiable distributions, Hamiltonian MCMC is popular

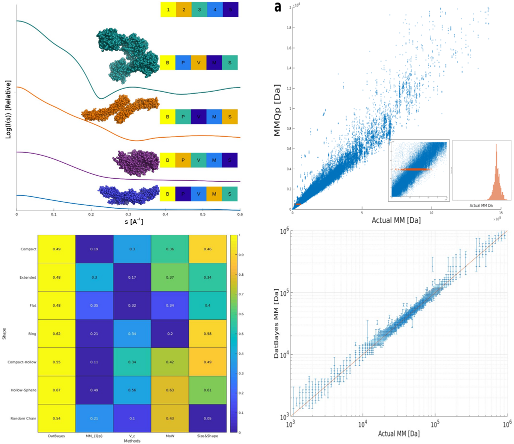
Variational Inference

- Approximates complex joint distributions with factored analytic marginals
- Simple and fast, but typically under-estimates effects of variable coupling
  - variational approximation ‘sits under the peak’ of the true joint distribution
  - **Expectation Propagation** is a variant that ‘sits over the peak’ and thus tends to over-estimate the effects of coupling

Example 1 – Bayesian Ensemble for SAXS Protein Mass

This paper[1] merged four simple protein-mass estimators in a Bayesian ensemble

- the estimators are based on integrals of SAXS profiles
- for each one, the probability distribution $p(\text{estimated mass} \mid \text{true mass})$ is obtained empirically from simulated profiles of known proteins
- the ensemble outperforms each estimator and provides calibrated uncertainty estimates

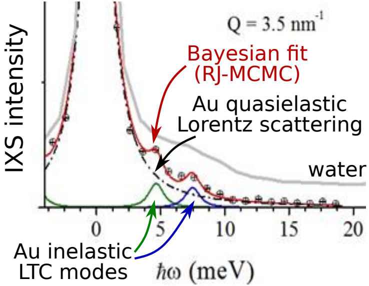


[1] Consensus Bayesian assessment of protein molecular mass from solution X-ray scattering data N. Hajizadeh *et al.* Nature Scientific Reports 8, 2018

Example 2 – Bayesian Fitting of Inelastic XS Models

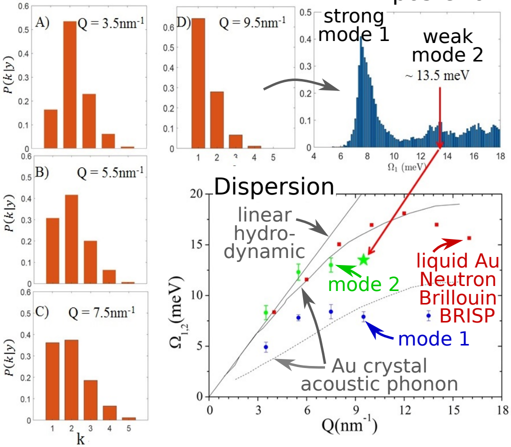
Bayesian fitting method[2] for Inelastic X-ray Scattering from gold nanoparticles in water

- models # active longitudinal-transverse modes *vs.* momentum transfer Q
- reversible jump MCMC inference



Bayesian posterior on number of active modes

Excitation freq. posterior



[2]Interpreting the Terahertz Spectrum of Complex Materials: The Unique Contribution of the Bayesian Analysis. A. De Francesco *et al.* Materials 12, 2019

Gaussian Processes / Kriging

Gaussian Processes (GP's) are simply Gaussian distributions defined over infinite-dimensional function spaces

- Gaussians are defined by their mean and pairwise covariances, so a scalar GP over variables $\mathbf{x}$ is defined by its **mean function** $\mu(\mathbf{x}) = E[\, f(\mathbf{x}) \mid f \sim \mathrm{GP}\,]$ and its **kernel** (covariance function) $k(\mathbf{x}, \mathbf{x}') = \mathrm{Cov}[\, f(\mathbf{x}), f(\mathbf{x}') \mid f \sim \mathrm{GP}\,]$
  - the kernel encodes our assumptions about how function values correlate across space
- Reasoning about functions sampled from GP's can always be reduced to finite-dimensional Gaussian calculations
- *E.g.* if a GP has mean zero and we observe $f \sim \mathrm{GP}$ at points $\mathbf{x}_1, \ldots, \mathbf{x}_k$, its expected value at a new point $\mathbf{x}$ is

$$E[\, f(\mathbf{x}) \mid f \sim \mathrm{GP}\,] = \begin{pmatrix} k(\mathbf{x},\mathbf{x}_1) & \ldots & k(\mathbf{x},\mathbf{x}_k) \end{pmatrix} \begin{pmatrix} k(\mathbf{x}_1,\mathbf{x}_1) & \ldots & k(\mathbf{x}_1,\mathbf{x}_k) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_k,\mathbf{x}_k) & \ldots & k(\mathbf{x}_1,\mathbf{x}_k) \end{pmatrix}^{-1} \begin{pmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_k) \end{pmatrix}$$

Bayesian / Gaussian Process Optimization

These properties make GP's an excellent choice for extrapolating the values of unknown functions from observations taken at a few locations.

Bayesian Optimization uses this to provide efficient methods for optimizing very expensive functions sample-by-sample

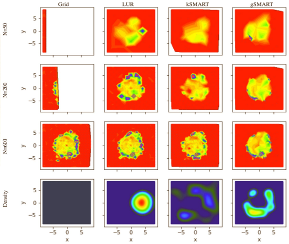
- A GP model is assumed: this encodes our critical prior assumptions about the underlying function's behaviour
- Function values are observed one-by-one:
  - after each sample, the GP model is updated
  - numerical optimization is run on the updated model to find the location $\mathbf{x}$ that maximizes {expected improvement in best-known function value} from a sample at that point
  - the underlying function is sampled at this point

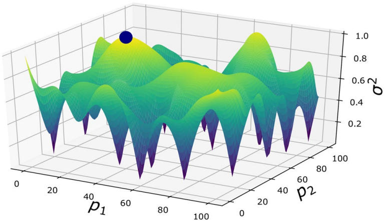
This is useful for economizing observations when each new one requires (*e.g.*) an expensive physical simulation or a real experimental run

- its main limitation is the inevitable mis-match between the assumed GP kernel and the (unknown) correlation behaviour of the underlying function.

Example – Kriging-based Automatic Sampling for SAXS

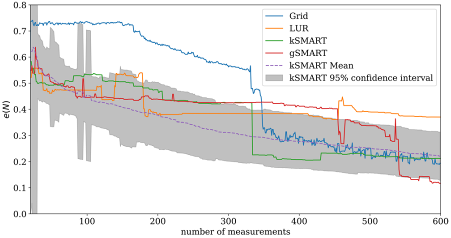
(a)





This paper[3] used a variant of GP Optimization to automatically choose sampling positions for SAXS samples

- the new sample is placed at the point with the greatest GP uncertainty
- see Jamie Sethian's talk later today

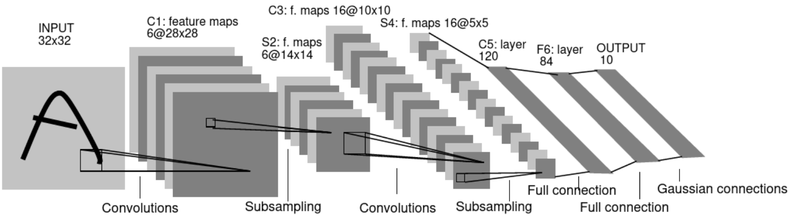


[3] A Kriging-Based Approach to Autonomous Experimentation with Applications to X-Ray Scattering. M. Noack *et al.* Nature Scientific Reports 9, 2019

Neural Network Methods

- The model is built recursively from heuristic nonlinear layers
  - think of this as deeply-nested feature extraction + output layer(s) for given task(s)
  - each layer is composed of linear projections + scalar nonlinearities (“activation functions”)
  - the linearities can be simple combinations, convolutions, . . .
  - the model is parametrized by the weights of the linear projections
- This gives a powerful representation that makes few assumptions, but it is even more heuristic than other ML methods
  - especially effective for continuous signals, images
  - also good for complex structured inputs like text, ranking, . . .
  - less good for discrete “data mining” problems
- Neural nets have huge numbers of parameters & highly non-convex learning problems
  - training requires *a lot* of training data and careful regularization
  - usually trained by variants of Stochastic Gradient Descent (SGD)

1990's Neural Nets – LeNet-5 Convolutional Net



LeNet-5 Architecture[4]

- 32×32 window scans cheques, reading handwritten digits one-by-one
- Arguably the first successful neural network product
- The network is tiny by todays standards!

[4] Handwritten digit recognition with a back-propagation network. Y. LeCun *et al.* NIPS 1989, Gradient-based learning applied to document recognition. Y. LeCun *et al.* Proc. IEEE 1998

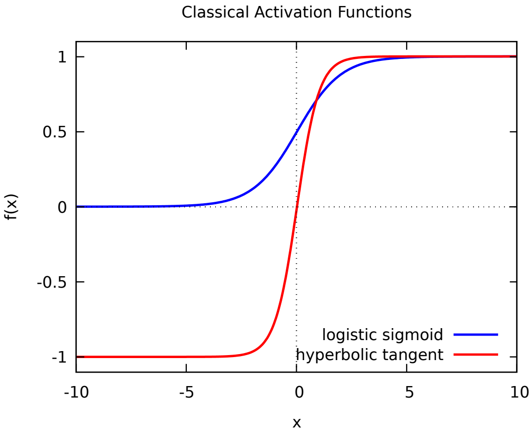
1990's Neural Nets

- Each node is linear projection + sigmoid nonlinearity with learnable weights and bias

$$\text{node\_output} = \text{activation\_function}\left(\sum \text{weights} \cdot \text{node\_inputs} + \text{bias}\right)$$

- Nodes may be densely connected or convolutional
- Output layer is logistic ("softmax") classifier
- Usually limited to $\sim$4-6 layers if fully connected, 8-10 if convolutional
- Seldom more than 10^4-10^5 training samples
- Training is by SGD with momentum over backprop'ed scalar classification loss

1990's Activation Functions



- Responses are linear at small $|u|$ but saturate for large $|u|$
- The curves have the same basic shape but $\text{logistic\_sigmoid}(u) \equiv 1/(1+e^{-u})$ is positive whereas $\tanh$ is vertically centred
- In practice $\tanh$ learns more quickly
  - owing to positivity, each weight update for logistic_sigmoid requires a substantial compensatory bias adjustment
  - *i.e.* its Hessian is less well conditioned

Vanishing Gradient Problem

- Individual sigmoids saturate easily giving very small gradient contributions
- Saturated units learn slowly owing to their small gradients
- The problem compounds backwards over the layers
  - the gradient contribution of a path is small if any sigmoid along it saturates
  - early layers get tiny gradients through many competing paths
  - this confuses the feedback signal and gives very slow learning
- The problem gets worse as training progresses
  - gradients never vanish, pushing weights and saturation to ever-more-extreme values
  - inter-path cancellations increase during training
- The problem gets worse for small training sets, impoverished outputs (*e.g*. single class labels) and under-regularized models – all of these make it easier to overfit
- The problem gets *much* worse with suboptimal weight initializations
  - all units in all layers need to be initialized in their ‘sweet spots’ – non-saturated yet still significantly nonlinear

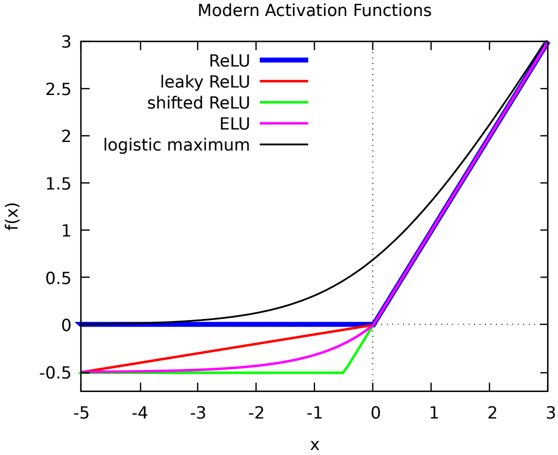
These issues limited the depth of network that could be trained for 20 years!

Making Deep Neural Nets Work

Deep Neural Networks took many years to develop because all of the following elements are critical for success

- Very large training sets (and the computational power and memory to use them)
- Well-chosen activation functions / “neuron” nonlinearities
- “Unit cells” that are carefully structured blocks of layers, not individual nodes
  - block normalization, skip connections, factored convolution, . . .
  - specialist custom layers – rendering, spatial transforms, optimization, . . .
- Rich ‘multi-loss’ feedback signals for training
- Carefully chosen weight initializations
- Richer regularization techniques
- More efficient optimization algorithms

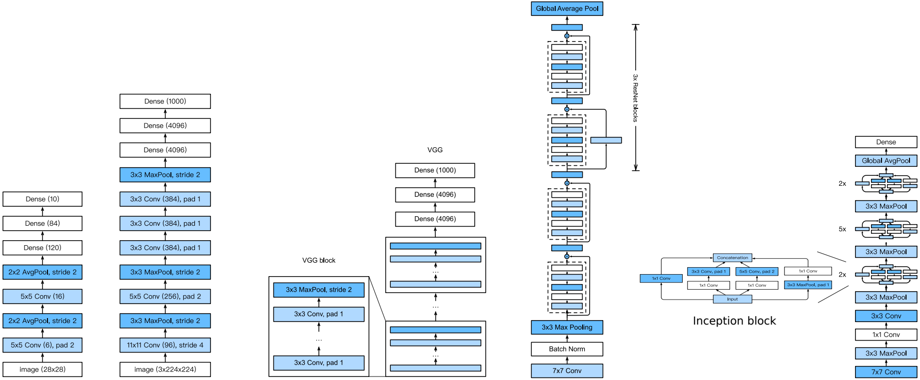
Modern Activation Functions



- "Rectified Linear Unit" $\mathrm{ReLU}(u) = \max(u, 0)$
  - Simple and fast, good for all non-output layers
  - Non-negative – bias-sensitivity may slow learning
  - Scale invariant so needs layerwise data normalization to prevent blow-up
- "Dead ReLU" problem
  - Gradient vanishes for $u < 0$, so nodes that become negative on all samples during training often remain dead from then on
  - Fixes: Leaky / Shifted ReLU add linear tilt / constant offset at $u < 0$
  - Exponential LU (ELU) adds decaying exponential at $u < 0$

Output layer (for classifiers) is still sigmoidal, giving logistic maximum ("cross-entropy") loss contributions – this activation function looks like a smoothed ReLU

Evolution of Block-based Architectures



LeNet-5 AlexNet VGG Net ResNet Inception V1

The “unit cells” of traditional neural nets are simple feed-forward nodes.
Modern deep nets use carefully structured compound blocks.

Images from: Dive into Deep Learning, Zhang *et al.*, 2019

Normalization Layers 1 – (Mini-)Batch Normalization

- Sigmoid nonlinearities are sensitive to scales and offsets of their input activations
- ReLU is scale-invariant (nothing controls activation scales!) but offset-sensitive
- Both are sensitive to differing scales of different inputs

Batch Normalization[5] is scalar affine normalization of a node's activation a_i across a single mini-batch, designed to keep the activations balanced during learning

- For node i, $a_i \to \gamma_i \frac{a_i - \mu_i}{\sigma_i + \epsilon} + \beta_i$ where $\mu_i = \mathrm{mean}_{\mathrm{MiniBatch}}(a_i)$ and $\sigma_i = \mathrm{stddev}_{\mathrm{MiniBatch}}(a_i)$
  - for convolutional nets, pool μ_i, σ_i across image but not across convolution filters
  - γ_i, β_i are learnable scaling parameters
- Usually applied immediately before each nonlinear unit
- Each mini-batch produces *different* ("inconsistent") normalizations
  - this "normalization noise" seems to help learning a lot! – it is controversial why ...
  - moderately-sized mini-batches are usually best – say $\mathcal{O}(10^2)$ samples
- After training, use whole training set to find final μ_i, σ_i values

[5] Batch normalization: accelerating deep network training by reducing internal covariate shift. Ioffe & Szegedy, arXiv:1502.03167

Normalization Layers 2 – Layer Normalization

Layer Normalization[6] is similar to Batch Normalization except that it pools the normalization signals

- over all neurons of a layer for a single training sample
- not over all samples in a mini-batch for a single neuron

It is especially useful for Recurrent Neural Nets, but also works well with CNN's.

[6] Layer Normalization. J.L. Ba *et al.* arxiv:1607.06450, 2016

Skip Connections and Residual Networks 1

In practical applications one often has mappings that can be represented well as a linear core with nonlinear corrections:

- the nonlinearities of conventional neural nets make it hard to learn the linear parts!

Residual Blocks

These convert standard neural net block(s) to nonlinear correctors by adding direct **skip connections** from their inputs to their outputs

- the skip connections provide clean paths for backprop signals, allowing very deep networks (100's of layers) to be trained

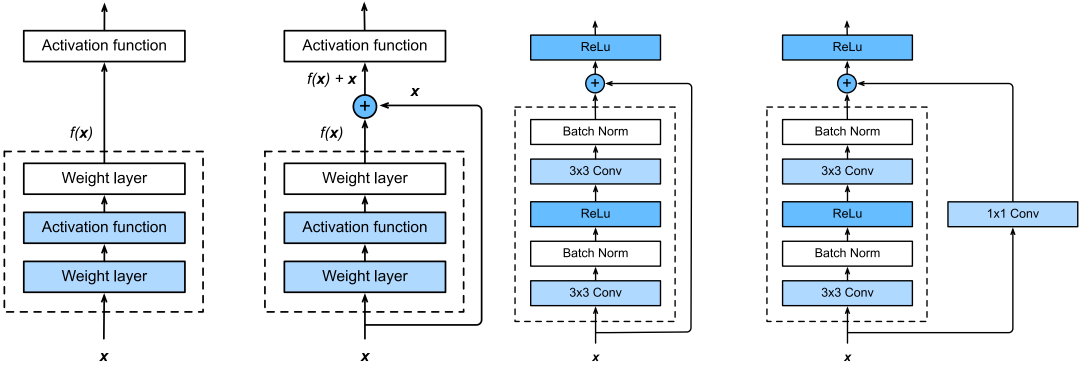
Residual Networks (ResNets)[7]

These are deep nets built from residual blocks

- useful especially when the input and output represent similar things, *e.g.* image-to-image tasks

[7] Deep residual learning for image recognition. He *et al*. CVPR 2016

Skip Connections and Residual Networks 2



Simple NN block | Residual block | Basic ResNet block | ResNet with 1×1 conv.

Image from: Dive into Deep Learning. Zhang, Lipton, Li, Smola, 2019.

Optimization Methods for Deep Nets 1

Deep Nets are usually trained by variants of **Stochastic Gradient Descent (SGD)**

- stochastic = optimizer sees samples individually in random order, not all together
- mini-batch = small set of samples presented together, usually chosen to fit GPU cache
- back-propagation = gradients are found by running the chain rule backwards through the net

Momentum Based Methods

SGD with Momentum (SGDM): the most basic approach – the parameter updates are proportional to a simple moving average of the current and past gradients

- the averaging reduces the influence of sampling noise and (weakly) compensates for the poor conditioning of typical loss Hessians

Nesterov Accelerated Gradient (NAG): a predictor-corrector variant of SGDM

- step along momentum, evaluate gradient, step along gradient
  - there are two network evaluations per update
- excellent theoretical properties & very robust, even for non-smooth objective functions

Optimization Methods for Deep Nets 2

Adaptive Gradient Methods

in addition to moving averages of gradient values, these use moving averages of gradient variances to provide componentwise rescalings of the step directions

- each variable gets its own adaptive learing rate, reducing the effects of poor scaling
- especially useful for applications with large numbers of rarely-active features
  - text, recommender systems ...

- **ADAGrad** uses a simple cumulative sum for the squared gradients
- **RMSProp** uses a moving average for the squared gradients
- **ADADelta** includes an additional moving average to set the learning rates
- **ADAM** corrects for a start-of-run estimation bias in RMSProp
  - it is currently the most popular approach for training deep nets

Warning: Adaptive Gradient methods are popular mainly owing to rapid convergence. Their aggressive variable rescaling often leads to *worse solutions* than SGDM or NAG[8].

[8] The Marginal Value of Adaptive Gradient Methods in Machine Learning. A. Wilson *et al.* arxiv:1705.08292, 2018

Weight Initialization

Network weights are usually initialized randomly with scalings chosen such that input activations of $\mathcal{O}(1)$ will produce output activations of $\mathcal{O}(1)$

Some Common Heuristics

- Initialize bias terms to zero
- Sample the weights from a Gaussian[9] with mean 0 and std.dev. $\sqrt{\frac{k}{\text{\# inputs per node}}}$
  - typically $k \sim 2$ for ReLU, $k \sim 1$ for sigmoid units
- For $\mathrm{softmax}$ output stages with many outputs, use std.dev. $\sqrt{\frac{1}{(\text{\# inputs per node})+(\text{\# outputs})}}$
- For ResNets, initialize output weights of each residual leg to small values
  - or use ‘Fixup’ normalization[10]

[9] Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. K. He *et al.* arXiv:1502.01852
[10] Fixup Initialization: Residual Learning without Normalization. H. Zhang *et al.* ICLR 2019.

Regularization for Deep Nets

Regularizing a network during training improves its test-set performance by reducing overfitting, and also helps training to converge. Adding noise has the same effect.

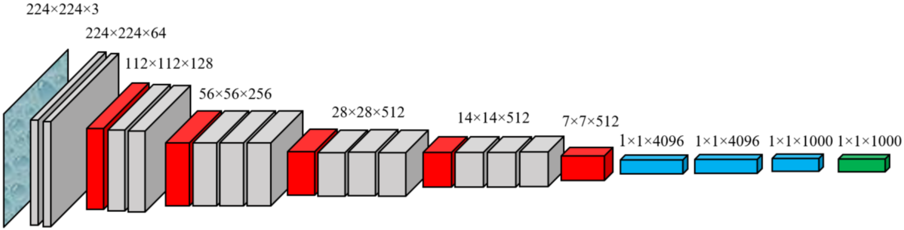
- **Weight Decay:** classical Tikhonov regularization of the weights
  - a quadratic penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ gives “weight decaying” update contributions $\delta\mathbf{w} \propto -\lambda\mathbf{w}$
  - other penalties can be used, *e.g.*, L1 penalties $\|\mathbf{w}\|_1$ lead to sparse weight vectors
- **Stochastic Gradient Optimization:** this itself is a noisy process with strong regularizing effects
- **Jittering:** apply small random perturbations to the training samples
  - *e.g.* for images, small translations & rotations, color changes, pixel noise, local damage ...
- **Drop-Out:** perturb layers of the network by randomly suppressing some fraction p of their node activations
  - to correct for bias, rescale the remaining activations by $\frac{1}{1-p}$
  - used mainly for densely connected layers not convolutional ones
- **Drop-Connect:** generalize Drop-Out by randomly suppressing some fraction of the $\mathrm{activation} * \mathrm{weight}$ products, not the activations themselves

Convolutional Deep Nets 1 – Overall Architecture

Convolutional nets are useful for applications based on complex continuous signals that are dominated by local interactions – audio, images, volumetric measurements, . . .

- They usually have several stages, each built from a series of **convolutional layers** followed by a single **pooling / dimension reduction** layer
  - normalization layers, residual connections, *etc.*, may also be included
  - the number of feature channels is usually increased as the map dimensions diminish
- The final convolutional feature layer can drive
  - a convolutional output layer, *e.g.* for image segmentation / object detection
  - a densely connected layer or an image-wide pooling one, *e.g.* for image classification

VGG image classification network



Convolutional Deep Nets 2 – Convolutional Layers

Convolutional layers run centred, finite-sized convolution filters (cross-correlation masks) with learned coefficients across their input feature maps

- **Parameter sharing:** the layer parameters (filter weights) apply to all output positions
- **Locality:** the output-map values depend only on *nearby* input-map values
- **Translation & size invariance:** of the input-map to output-map transformation

Besides their spatial dimensions, the i/o maps typically have multiple **feature channels**

- output channels usually take inputs from every input one, so for $m \times n$ rectangular filters mapping p input channels to q output ones, we need to sum over $m \times n \times p$ input values for each of q output channels
- this computation costs $m.n.p.q$ flops per output position
    - for deep nets over volumetric images, convolutional layers cost even more
- the inputs, outputs and weights form multi-dimensional arrays
    - the “tensors” of software like TensorFlow
- the input maps need to be padded (extended at their borders by the mask half-width), otherwise each layer reduces the map dimension

Convolutional Deep Nets 3 – Dimension-Reducing Layers

Pooling Layers

These merge 2×2 (or whatever) local blocks of activations to reduce the map dimension

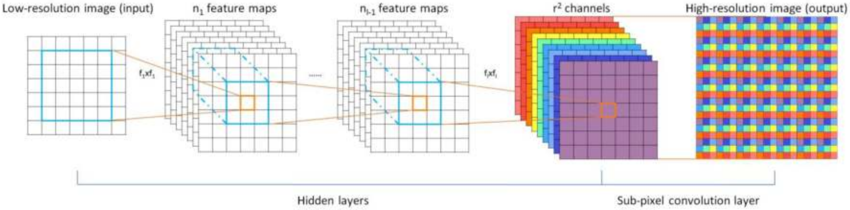
- **Max pooling** takes the maximum input activation from the given pooling region
- **Average pooling** takes the average instead (this is usually less effective)
- **Stochastic pooling** copies a random input from the pooling region – a form of Drop-Out adapted to pooling

Strided Convolution

This reduces the output map dimension by applying (dense) convolutional filters only at even input positions (or at each k-th one)

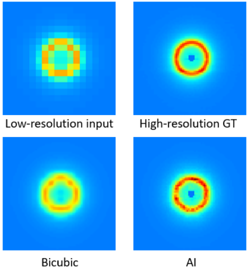
- *e.g.* this is often used to smoothly subsample feature maps

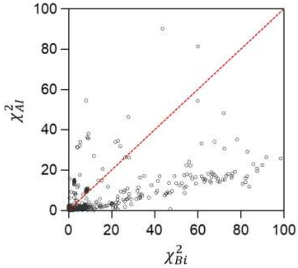
Example 1 – CNN-based Super-resolution for SANS



This paper[11] trained a short CNN to up-sample (increase the resolution) of Small Angle Neutron Scattering images – EQ-SANS at SNS

- the results were significantly more accurate than bicubic image interpolation



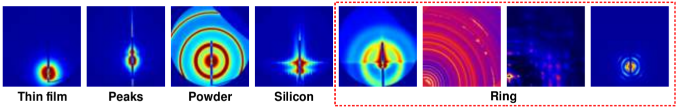


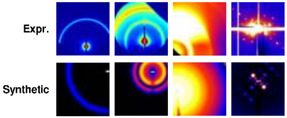
[11] Accelerating Neutron Scattering Data Collection and Experiments Using AI Deep Super-Resolution Learning. Ming-Ching Chan *et al.* arxiv:1904.08450, 2019

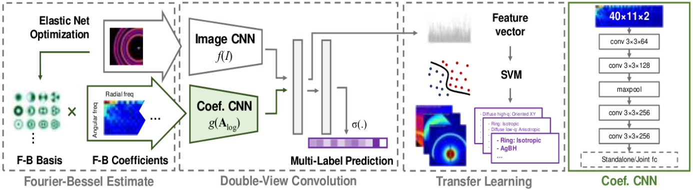
Example 2 – CNN-based Content Tagging for SAXS Images

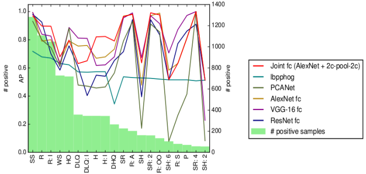
This paper[12] used an SVM over CNN image features to tag Small Angle X-Ray Scattering images with their content types (Rings, Halo, . . .)

- AlexNet on image + CNN on elastic-net-aligned Fourier-Bessel coefficients
- trained on synthetic images, converted to real ones with transfer learning









[12]Automatic X-ray Scattering Image Annotation via Double-View Fourier-Bessel Convolutional Networks. Ziqiao Guan *et al.* BMVC 2018

Speeding Up Convolution 1

Convolution is conceptually simple but computationally expensive, with many parameters

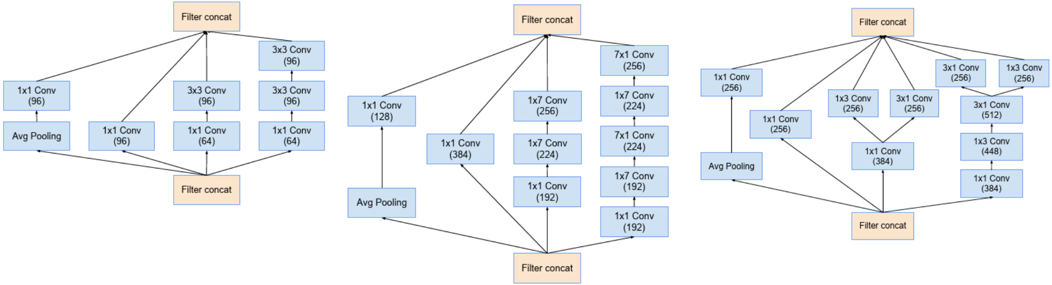
In practice both the cost and the number of parameters can often be reduced with little loss of accuracy:

- **Deepening:** replace convolution layers that require large filters with several layers using smaller ones
- **Dilated convolution:** use sparse convolution filters that only access, *e.g.*, even offsets
  - can also 'cut off the corners' of the convolution mask, *e.g.* octahedral masks
- **Depthwise separation:** replace single 'fat' convolutions with networks of 'thinner' (lower dimensional) ones
  - *e.g.* replace a $m \times n \times p \to q$ convolution block with separate $m \times n$ convolutions over each of the p input maps followed by 1×1 'convolutions' (p-D to q-D linear projections) to get the q output maps

Inception modules exploit these ideas.

Speeding Up Convolution 2 – Inception Modules

Inception V1–V4 is a series of module designs from Google for speeding up convolution.



The basic ‘A’, ‘B’ and ‘C’ modules from Inception-V4[13]

Inception-V4 also defines modules for spatial pooling and ResNet / skip-connection stages.

[13] Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, C. Szegedy *et al.*, arxiv:1602.07261

U-Net and Multiscale CNN

U-Net[14]

A CNN architecture for multi-scale image analysis & resynthesis

- down-going leg of ‘U’ recursively extracts feature images at multiple scales
- up-going leg of ‘U’ recursively recombines each level’s feature image (via direct skip connections) with information from coarser levels
- output is simple or multiscale image

Popular for image segmentation / labeling, image-to-image synthesis, ...

Multiscale CNN[15]

A U-Net, trained with multi-scale loss contributions instead of single scale ones

- applied to multiscale inverse problems: phase recovery, denoising
- see Christoph Koch’s talk tomorrow ...

---

[14] U-Net: Convolutional Networks for Biomedical Image Segmentation, O. Ronneberger *et al.*, MICCAI 2015

[15] Multi-scale Convolutional Neural Networks for Inverse Problems, Feng Wang *et al.*, arXiv:1810.12183

U-Net and Multiscale CNN – 2

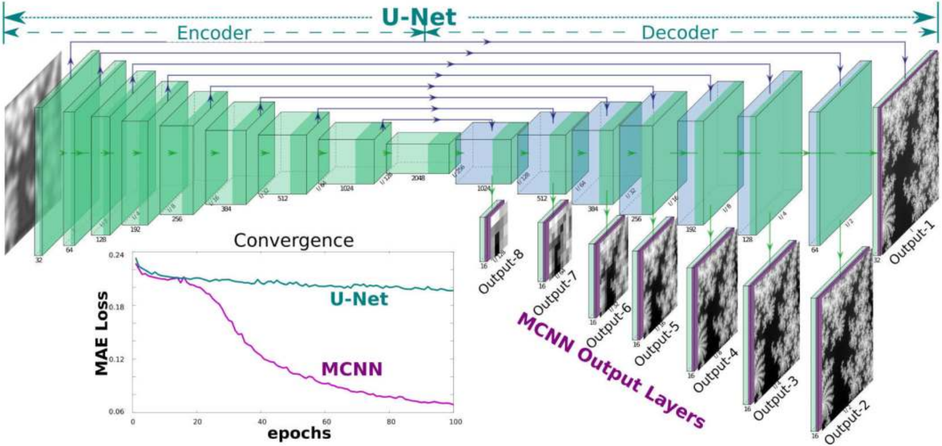


Image from: Multi-scale Convolutional Neural Networks for Inverse Problems, Feng Wang *et al.*, arXiv:1810.12183

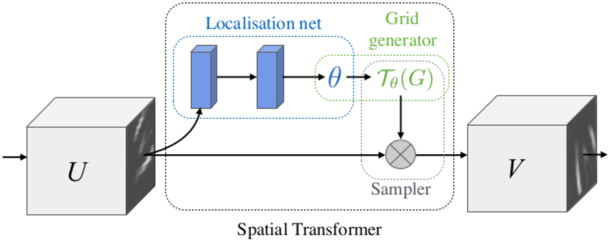
Spatial Transformer Layer

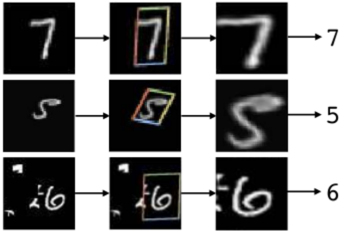
Image-based networks often need to spatially transform parts of their feature maps

- *e.g.* to register them to a content-classifying output network

Spatial Transformers[16] are custom deep net layers designed for this

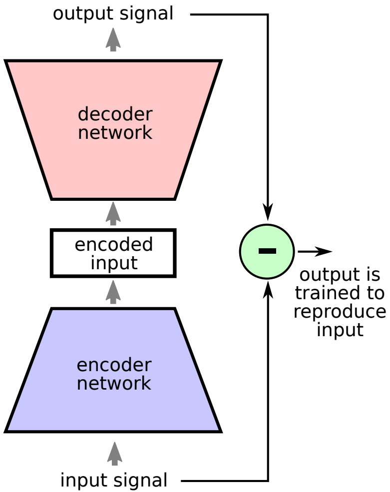
- a transform estimation sub-network takes the input feature map and returns the corresponding transform parameters
- a conventional image resampler applies the transform to the input map





[16] Spatial Transformer Networks. M. Jaderberg *et al.* arxiv:1506.02025, 2016

Generative Deep Nets – Auto-Encoder



Auto-Encoder = encoder-decoder pair, trained to reproduce its input

- lossy encoding maps high-dimensional input to a lower-dimensional latent space
- decoder tries to reconstruct input from encoding
- both encoder and decoder are typically deep neural networks

Variational Auto-Encoder (VAE)

Standard auto-encoders are trained *only* to reproduce their inputs

- their latent encodings are seldom very coherent owing to overfitting
- to get smoother, more useable encodings we need to regularize

Variational Auto-Encoders[17] are one popular way of doing this

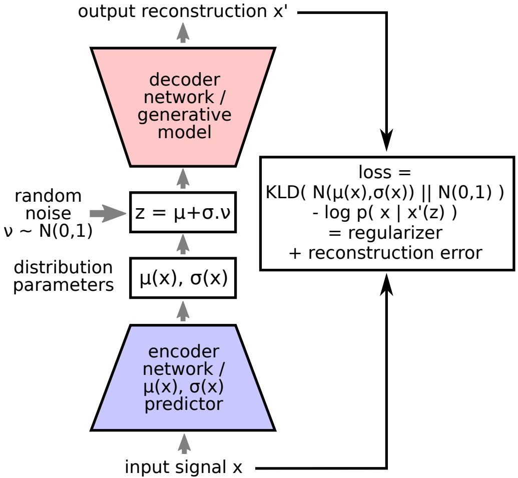
- during training, Gaussian noise is added to the encoded inputs before reconstruction
  - this regularizes the model: it learns to map nearby codes to nearby outputs to limit the impact of the noise
- a loss penalty controls the relative scaling of the noise and the incoming codes
  - traditionally Kullback-Leibler divergence is used – this is motivated in terms of Approximate Variational Bayes inference in a quasi-Gaussian model

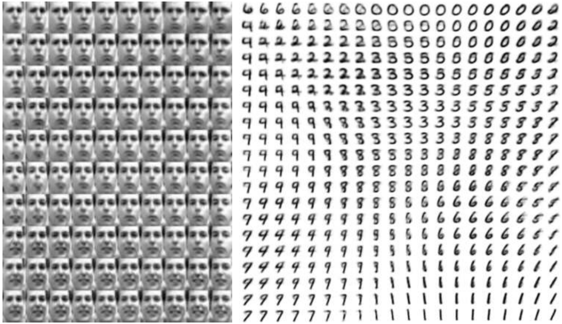
VAE's turn out to be an excellent way to synthesize complex signals and images

- synthetic outputs are generated by feeding pure Gaussian noise into the decoder

[17] Auto-Encoding Variational Bayes. D. Kingma, M. Welling. arxiv:1312.6114, 2014

Variational Auto-Encoder 2

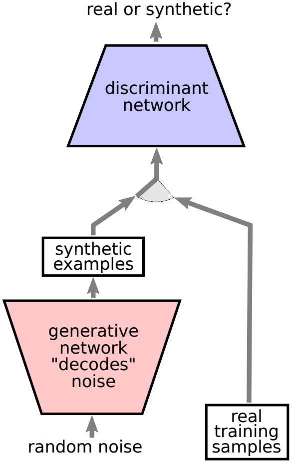




Some generated face and digit images, illustrating that VAE encodings vary smoothly across their latent spaces.

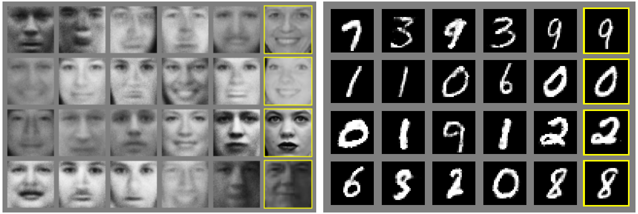
Images from: Auto-Encoding Variational Bayes. D. Kingma, M. Welling. arxiv:1312.6114, 2014

Generative Adversarial Networks (GANs)



Generative Adversarial Networks are another good way to synthesize realistic signals and images

- an example-generating network is fed with random noise
- a discriminator network is trained to distinguish between synthetic and real examples
- training is adversarial – generator & discriminator compete
  - delicate to train – optimal solution is a saddle point

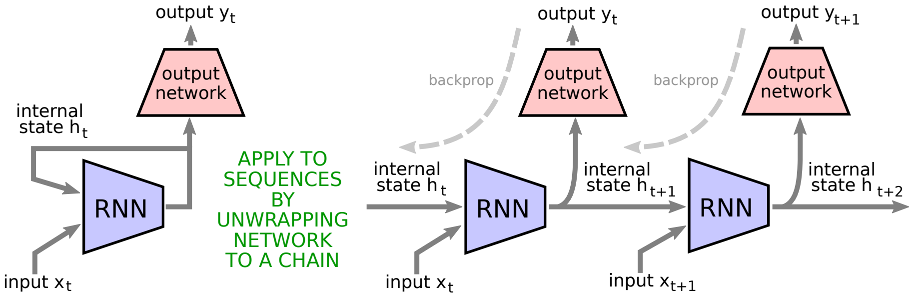


Images from: Generative Adversarial Nets. I. Goodfellow *et al.* NIPS 2014

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks are useful for modelling variable-length sequences with unpredictable long-term dependencies

- *e.g.* text, speech, action sequences, DNA / protein backbones,. . .
- the model has internal state, propagated from one time-step to the next
- training unwraps the net, applying it forwards through sequence, then back-propagating backwards through the chain



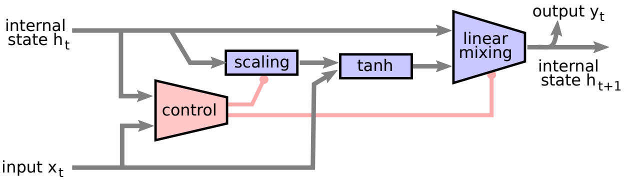
Gated / Long-Term Recurrent Networks

The state vectors of simple RNN's tend to increase or decrease exponentially over time

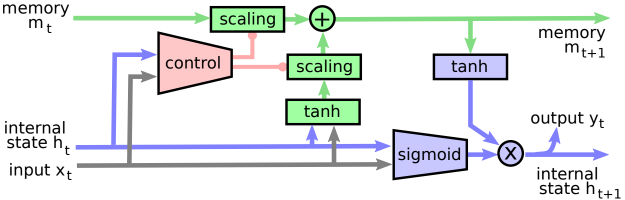
- numerical blow-up and/or quick forgetting of state information
- careful initialization sometimes helps to counter this[18]

Practical RNN's typically include **active gating** to choose what to remember

- usually GRU or LSTM
- most people use GRU these days as it is simpler and good enough



Gated Recurrent Unit (GRU)



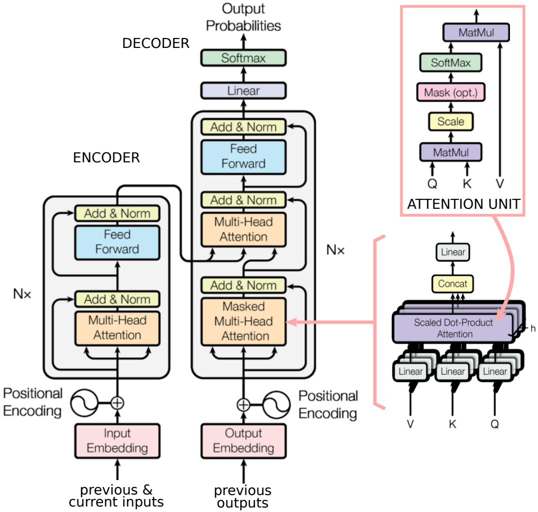
Long Short-Term Memory Unit (LSTM)

[18] A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. Q.V. Le *et al.* arxiv:1504.00941, 2015

Attention Based Networks / Transformers

Attention based networks / Transformers[19] are displacing RNN's in many sequence-to-sequence applications

- they are auto-regressive like RNN's but without any memory or hidden state
- at each time-step the network sees *all* of its previous (in the sequence) inputs and outputs
  - the inputs (with positional encodings) are mapped into a common embedding space and summed, and similarly for the outputs
- a "neural attention" mechanism then selects relevant features to propagate
  - weighted pooling of values gated by dot-product units

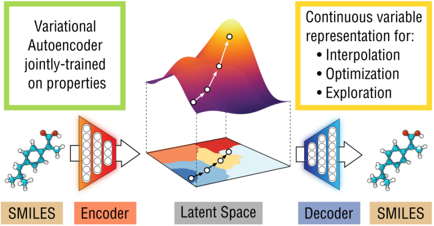


[19] Attention Is All You Need. A. Vaswani *et al.* arxiv:1706.03762

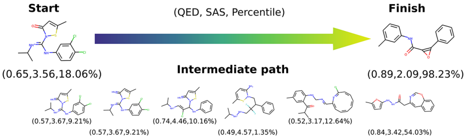
Example 1 – VAE/RNN Representation for Molecule Design

This paper[20] produced a Variational Auto-Encoder based latent space for molecule design

- molecules are represented as SMILE strings
  - *e.g.* C1=CC=CC=C1
- the VAE encoder is a GRU RNN or a CNN
  - a GAN is also possible
- the VAE decoder is a GRU RNN with random sampling from output-letter distribution
- the latent space dimension is 150-200
- multilayer perceptrons on the latent space are trained to predict molecular properties
- optimization uses Gaussian Process regression over the properties for smoothing



general approach



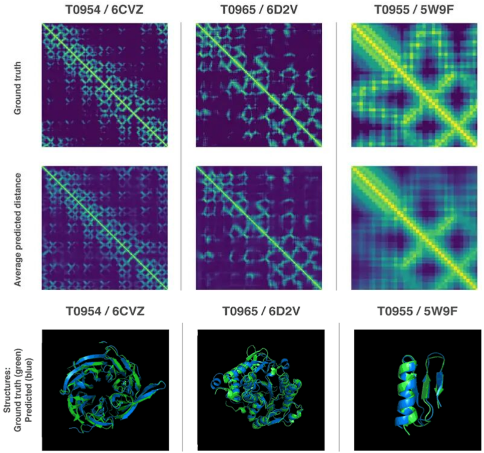
molecular property optimization

[20] Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules. R. Gomez-Bombarelli *et al.*, ACS Cent. Sci. 4, 2018

Example 2 – AlphaFold De Novo Protein Shape Prediction

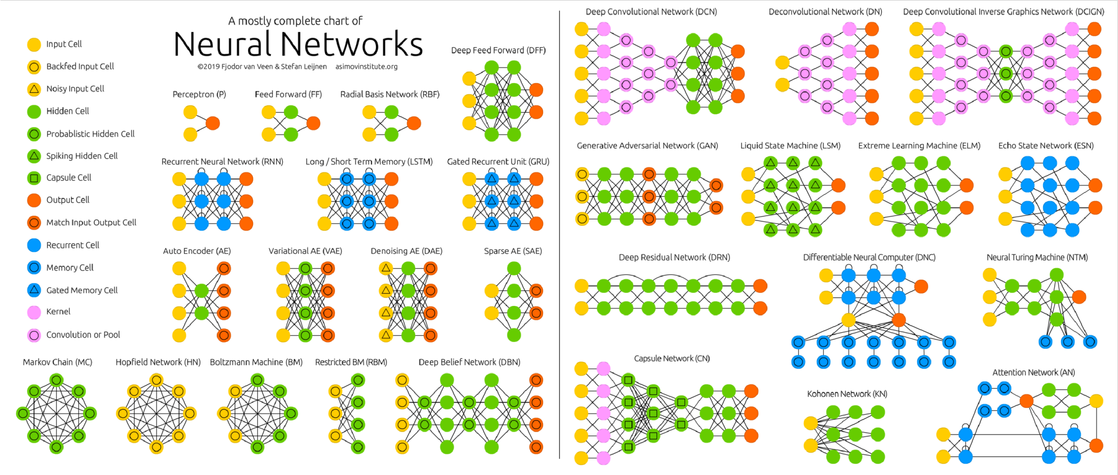
Deep Net[21] approach to predicting protein folding

- single domain, backbone not side chains, residue-level not atom-level
- evolutionary co-variation of residues gives hints regarding spatial proximity
- deep net predicts pairwise distance maps between residues using co-variation features and backbone sequence
- second net predicts protein-specific folding potential from distance maps and torsion angle priors
- gradient descent or simulated annealing adjusts torsion angles to optimize protein shape



[21] Unpublished. See https://deepmind.com/blog/article/alphafold and AlphaFold at CASP13, M. AlQuraishi, Bioinformatics, 2019

There are Many Other Kinds of Neural Nets . . .



From: F. van Veen & S. Leijnen, www.asimovinstitute.org/neural-network-zoo

Deep Net Software 1

TensorFlow

Google's main library for large-scale dataflow programming, not just deep learning

- many users but reputation for being hard to use (especially for beginners / debugging)
- efficient for huge industrial problems, resource hungry for small ones
- TensorFlow-2.0 just released – said to be much easier to use
- **Keras** is a simplified python front end for TensorFlow, popular with beginners

PyTorch

Facebook's python reboot of Torch ML+DL library

- recent but very popular, especially in academia
- simpler to use than TensorFlow but still powerful
- **FastAI** is a simplified interface

Deep Net Software 2

MXNet

Apache open-source Deep Net framework, used by Amazon & others

- efficient & scalable industrial solution, fewer users than TF but popular with them
- MXNet Gluon is a simplified interface, or use Keras over MXNet
- integrates well with other Apache infrastructure: Spark / MLlib, Singha, Mahout

Others

- CUDA – low-level NVIDIA drivers for numerics, DL, *etc*. on GPUs, used by everyone
- Theano, Caffe, Torch. . . – older DL libraries, now largely obsolete
- CNTK / Microsoft Cognitive Toolkit – Microsoft DL library, few users
- Chainer – deep learning in python, mainly used in Japan
- DL4J – deep learning in Java, few users

Deep Learning Resources

Books

- "Deep Learning", I. Goodfellow, Y. Bengio, A. Courville, 2016
  - the standard academic DL text, but it already looks dated (www.deeplearningbook.org)
- "Deep Learning with Python", F. Chollet, 2018
  - a simple hands-on introduction for ML engineers, using Keras
- "Dive into Deep Learning", A. Zhang, Z. Lipton, M. Li, A. Smola, 2019
  - an interactive web textbook (d2l.ai) using MXNet + Jupyter
  - still evolving so coverage is somewhat patchy, but useful for basic models
- there are many others, of variable quality

Other Resources

- Kaggle.com – popular ML challenge site, useful for seeing what works and/or finding people to solve your problem for you
- many good blogs and on-line courses – course.fast.ai, deeplearning.ai, edx.org, . . .

Other ML Software

SciKit-Learn

Popular Python / SciPy / NumPy library for general ML

Other ML Systems

- **Torch:** older but broader variant of PyTorch, in Lua but good interface to other languages
- **Shogun:** C++ ML library, mainly kernel methods for large-scale problems
- **LibSVM:** popular low-level C library for SVM
- **XGBoost:** low-level C++ library for gradient boosting
- **R:** popular system for statistical analysis, contains some basic ML but not very scalable
- **Weka:** popular but dated system for data mining, some ML
- **Julia:** a recent language for scientific computing – highly recommended but still a work-in-progress