Artificial Intelligence

AI & Machine Learning & Neural Nets

Marc Toussaint
University of Stuttgart
Winter 2018/19

Motivation:

Neural networks became a central topic for Machine Learning and AI. But in principle, they're just parameterized functions that can be fit to data. They lack many appealing aspects that were the focus of ML research in the '90-'10. So why are they so successful now? This lecture introduces the basics and tries to discuss the success of NNs.

What is AI?

- AI is a research field
- AI research is about systems that take decisions
  - AI formalizes decision processes: interactive (decisions change world state), or passive
  - AI distinguishes between agent(s) and the external world: decision variables
- ... systems that take optimal/desirable decisions
  - AI formalizes decision objectives
  - AI aims for systems to exhibit functionally *desirable behavior*
- ... systems that take optimal decisions on the basis of all available information
  - AI is about inference
  - AI is about learning

What is Machine Learning?

- In large parts, ML is: (let's call this ML^0)

  Fitting a function $f : x \mapsto y$ to given data $D = \{(x_i, y_i)\}_{i=1}^n$

What is Machine Learning?

- In large parts, ML is: (let's call this ML^0)

  Fitting a function $f : x \mapsto y$ *to given data* $D = \{(x_i, y_i)\}_{i=1}^n$

- And what does that have to do with AI?
- Literally, not much:
  - The decision made by a ML^0 method is only a single decision: Decide on the function $f \in \mathcal{H}$ in the hypothesis space $\mathcal{H}$
  - This "single-decision process" is not interactive; ML^0 does not formalize/model/consider how the choice of f changes the world
  - The objective $\mathcal{L}(f, D)$ in ML^0 depends only on the given static data D and the decision f, not how f might change the world
  - "Learning" in ML^0 is not an interactive process, but some method to pick f on the basis of the static D. The typically iterative optimization process is not the decision process that ML^0 focusses on.

But, function approximation can be used to help solving AI (interactive decision process) problems:

- Building a trained/fixed f into an interacting system based on human expert knowledge:
  An engineer trains a NN f to recognize street signs based on a large fixed data set D. S/he builds f into a car to drive autonomously. That car certainly solves an AI problem; f continuously makes decisions that change the state of the world. But f was never optimized literally for the task of interacting with the world; it was only optimized to minimze a loss on the static data D. It is not a priori clear that minimizing the loss of f on D is related to maximizing rewards in car driving. That fact is only the expertise of the engineer; it is not some AI algorithm that discovered that fact. At no place there was an AI method that "learns to drive a car". There was only an optimization method to minimize the loss of f on D.
- Using ML in interactive decision processes:
  To approximate a Q-function, state evaluation function, reward function, the system dynamics, etc. Then, other AI methods can use these approximate models to actually take decisions in the interactive context.

What is Machine Learning? (beyond ML^0)

- In large parts, ML is: (let's call this ML^0)

  Fitting a function $f : x \mapsto y$ *to given data* $D = \{(x_i, y_i)\}_{i=1}^n$

  Beyond ML^0:

- Fitting more structured models to data, which includes
  - Time series, recurrent processes
  - Graphical Models
  - Unsupervised learning (semi-supervised learning)

  ...but in all these cases, the scenario is still not interactive, the data D is static, the decision is about picking a single model f from a hypothesis space, and the objective is a loss based on f and D only.

- Active Learning, where the "ML agent" makes decisions about what data label to query next
- Bandits, Reinforcement Learning

ML^0 objective: Empirical Risk Minimization

- We have a hypothesis space $\mathcal{H}$ of functions $f : x \mapsto y$
  In a standard parameteric case $\mathcal{H} = \{f_\theta \,|\, \theta \in \mathbb{R}^n\}$ are functions $f_\theta : x \mapsto y$ that are described by n parameters $\theta \in \mathbb{R}^n$

- Given data $D = \{(x_i, y_i)\}_{i=1}^n$, the standard objective is to minimize the "error" on the data

$$f^* \operatorname*{argmin}_{f \in \mathcal{H}} \sum_{i=1}^{n} \ell(f(x_i), y_i) \ ,$$

  where $\ell(\hat{y}, y) > 0$ penalizes a discrepancy between a model output $\hat{y}$ and the data y.
  - Squared error $\ell(\hat{y}, y) = (\hat{y} - y)^2$
  - Classification error $\ell(\hat{y}, y) = [\hat{y} \neq y]$
  - neg-log likelihood $\ell(\hat{y}, y) = -\log p(y \,|\, \hat{y})$
  - etc

What is a Neural Network?

- A parameterized function $f_\theta : x \mapsto y$

What is a Neural Network?

- A parameterized function $f_\theta : x \mapsto y$
  - θ are called weights
  - $\min_\theta \sum_{i=1}^{n} \ell(f_\theta(x_i), y_i)$ is called training

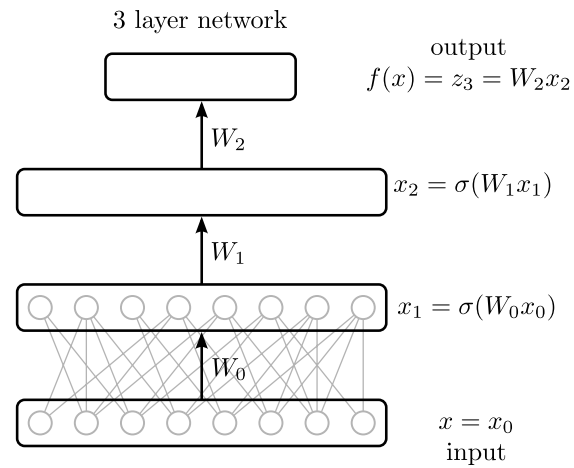
What is a Neural Network?

- Standard fwd-forward NN $\mathbb{R}^{h_0} \mapsto \mathbb{R}^{h_L}$ with L layers:

1-layer	$f_\theta(x) = W_0 x$	$\theta = (W_0), W_0 \in \mathbb{R}^{h_1 \times h_0}$
2-layer	$f_\theta(x) = W_1 \sigma(W_0 x)$	$\theta = (W_1, W_0), W_i \in \mathbb{R}^{h_{i+1} \times h_i}$
3-layer	$f_\theta(x) = W_2 \sigma(W_1 \sigma(W_0 x))$	$\theta = (W_3, W_2, W_0)$

- The *activation function* $\sigma(z)$ is applied *element-wise*

rectified linear unit (ReLU)	$\sigma(z) = z[z \geq 0]$
leaky ReLU	$\sigma(z) = \begin{cases} 0.01z & z < 0 \\ z & z \geq 0 \end{cases}$
sigmoid, logistic	$\sigma(z) = 1/(1 + e^{-z})$
tanh	$\sigma(z) = \tanh(z)$



Neural Networks: Basic Equations

- Consider L layers (hidden plus output), each h_l-dimensional
  - let $z_l = W_{l\text{-}1} x_{l\text{-}1} \in \mathbb{R}^{h_l}$ be the inputs to all neurons in layer l
  - let $x_l = \sigma(z_l) \in \mathbb{R}^{h_l}$ be the **activation** of all neurons in layer l
  - redundantly, we denote by $x_0 \equiv x$
- **Forward propagation:** An L-layer NN recursively computes,

$$\forall_{l=1,..,L\text{-}1} : \; z_l = W_{l\text{-}1} x_{l\text{-}1} \;, \quad x_l = \sigma(z_l)$$

  and then computes the output $f \equiv z_L = W_L x_L$
- **Backpropagation:** Given some loss $\ell(f)$, let $\delta_L \triangleq \frac{\partial \ell}{\partial f} = \frac{\partial \ell}{\partial z_L}$. We can recursivly compute the loss-gradient w.r.t. the *inputs* of layer l:

$$\forall_{l=L\text{-}1,..,1} : \; \delta_l \triangleq \frac{d\ell}{dz_l} = \frac{d\ell}{dz_{l+1}} \frac{\partial z_{l+1}}{\partial x_l} \frac{\partial x_l}{\partial z_l} = [\delta_{l+1} \, W_l] \circ [\sigma'(z_l)]^\top$$

  where $\circ$ is an *element-wise product*. The gradient w.r.t. weights is:

$$\frac{d\ell}{dW_{l,ij}} = \frac{d\ell}{dz_{l+1,i}} \frac{\partial z_{l+1,i}}{\partial W_{l,ij}} = \delta_{l+1,i} \, x_{l,j} \quad \text{or} \quad \frac{d\ell}{dW_l} = \delta_{l+1}^\top x_l^\top$$

Behavior of Gradient Propagation

- Propagating δ_l back through many layers can lead to problems
- For the classical sigmoid $\sigma(z)$, $\sigma(z)'$ is always $< 1 \quad \Rightarrow$ **vanishing gradient**
  Modern activations functions (ReLU) reduce this problem

- The Initialization of weights is super important!
  E.g., initialize weights in W_l with standard deviation $\frac{1}{\sqrt{h_l}}$. Roughly: If each element of z_l has standard deviation ϵ, the same should be true for z_{l+1}.

NN regression & regularization

- In the standard regression case, $h_L = 1$, we typically assume a squared error loss $\ell(f) = \sum_i (f_\theta(x_i) - y_i)^2$. We have

$$\delta_L = \sum_i 2(f_\theta(x_i) - y_i)^\top$$

- Regularization:
  - Old: Add a L_2 or L_1 regularization. First compute all gradients as before, then add $\lambda W_{l,ij}$ (for L_2), or $\lambda \operatorname{sign} W_{l,ij}$ (for L_1) to the gradient. Historically, this is called **weight decay**, as the additional gradient leads to a step decaying the weighs.
  - Modern: Dropout
- The optimal output weights are as for standard regression

$$W^*_{L\text{-}1} = (X^\top X + \lambda I)^{\text{-}1} X^\top y$$

where X is the data matrix of activations $x_{L\text{-}1} \equiv \phi(x)$

NN classification

- In the multi-class case we have $h_L = M$ output neurons, one for each class. The function $f(x) \in \mathbb{R}^m$ is the *discriminative function*, which means that the predicted class is the $\text{argmax}_{y \in \{1,..,M\}} [f_\theta(x)]_y$.
- Choosing neg-log-likelihood objective $\leftrightarrow$ logistic regression
- Choosing hinge loss objective $\leftrightarrow$ "NN + SVM"..
  - Let y^* be the correct class and let's use the short notation $f_y = [f_\theta(x)]_y$ for the discriminative value for class y
  - The **one-vs-all** hinge loss is $\sum_{y \neq y^*} [1 - (f_{y^*} - f_y)]_+$
  - For output neuron $y \neq y^*$ this implies a gradient $\delta_y = [f_{y^*} < f_y + 1]$
  - For output neuron y^* this implies a gradient $\delta_{y^*} = -\sum_{y \neq y^*} [f_{y^*} < f_y + 1]$ Only data points inside the margin induce an error (and gradient).
  - This is also called **Perceptron Algorithm**

Discussion: Why are NNs so successful now?

Historical Perspective

(This is completely subjective.)

- Early (from 40ies):
  - McCulloch Pitts, Hebbian learning, Rosenblatt, Werbos (backpropagation)
- 80ies:
  - Start of connectionism, NIPS
  - ML wants to distinguish itself from pure statistics (“machines”, “agents”)
- ’90-’10:
  - More theory, better grounded, Statistical Learning theory
  - Good ML is pure statistics (again) (Frequentists, SVM)
  - ...or pure Bayesian (Graphical Models, Bayesian X)
  - sample-efficiency, great generalization, guarantees, theory
  - Great successes, in applications across disciplines; supervised, unsupervised, structured
- ’10-:
  - Big Data. NNs. Size matters. GPUs.
  - Disproportionate focus on images
  - Software engineering becomes central

- NNs did not become “better” than they were 20y ago. By the standards of ’90-’10, they would still be horrible. What changed is the standards by which they’re are evaluated:

- NNs did not become “better” than they were 20y ago. By the standards of ’90-’10, they would still be horrible. What changed is the standards by which they’re are evaluated:
  Old:
  - Sample efficiency & generalization; get the most from little data
  - Guarantees (both, w.r.t. generalization and optimization)
  - Being only as good as a nearest neighbor methods is embarrasing

  New:
  - Ability to cope with billions of samples $\rightarrow$ no batch processing, but stochastic optimization
  - Happy to end up in some local optimum. (Theory on “every local optimum of a large deep net is good”.)
  - Stochastic optimization methods (ADAM) without monotone convergence
  - Nobody compares to nearest neighbor methods – nearest neighbor on 1B data points is too expensive anyway. I guess that it’d perform very well (for a descent kernel) and a NN could be glad to perform equally well

NNs vs. nearest neighbor

- Imagine an autonomous car. Instead of carrying a neural net, it carries 1 Petabyte of data (500 hard drives, several billion pictures). In every split second it records an image from a camera and wants to query the database to returen the 100 most similar pictures. Perhaps with a non-trivial similarity metric. That's not reasonable!
- In that sense, NNs are much better than nearest neighbor. They store/compress/memorize huge amounts of data. Whether they actually generalize better than a good nearest neighbor methods is not so relevant.
- That's how the standards changed from '90-'10 to nowadays

Images & Time Series

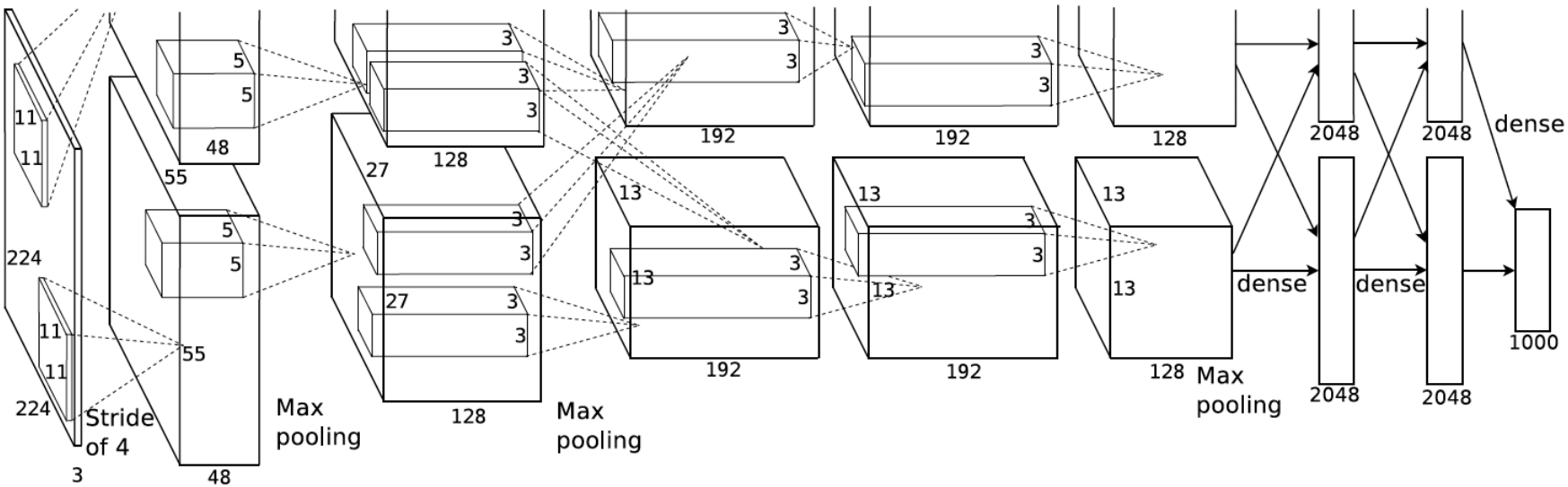
- I'd guess, 90% of the recent success of NNs is in the areas of images or time series
- For images, convolutional NNs (CNNs) impose a very sensible prior; the representations that emerge in CNNs are in fact similar to representations in the visual area of our brain.
- For time series, long-short term memory (LSTM) networks represent long-term dependencies in a way that is well trainable – something that is hard to do with other model structures.
- Both these structural priors, combined with huge data and capacity, make these methods very strong.

Convolutional NNs

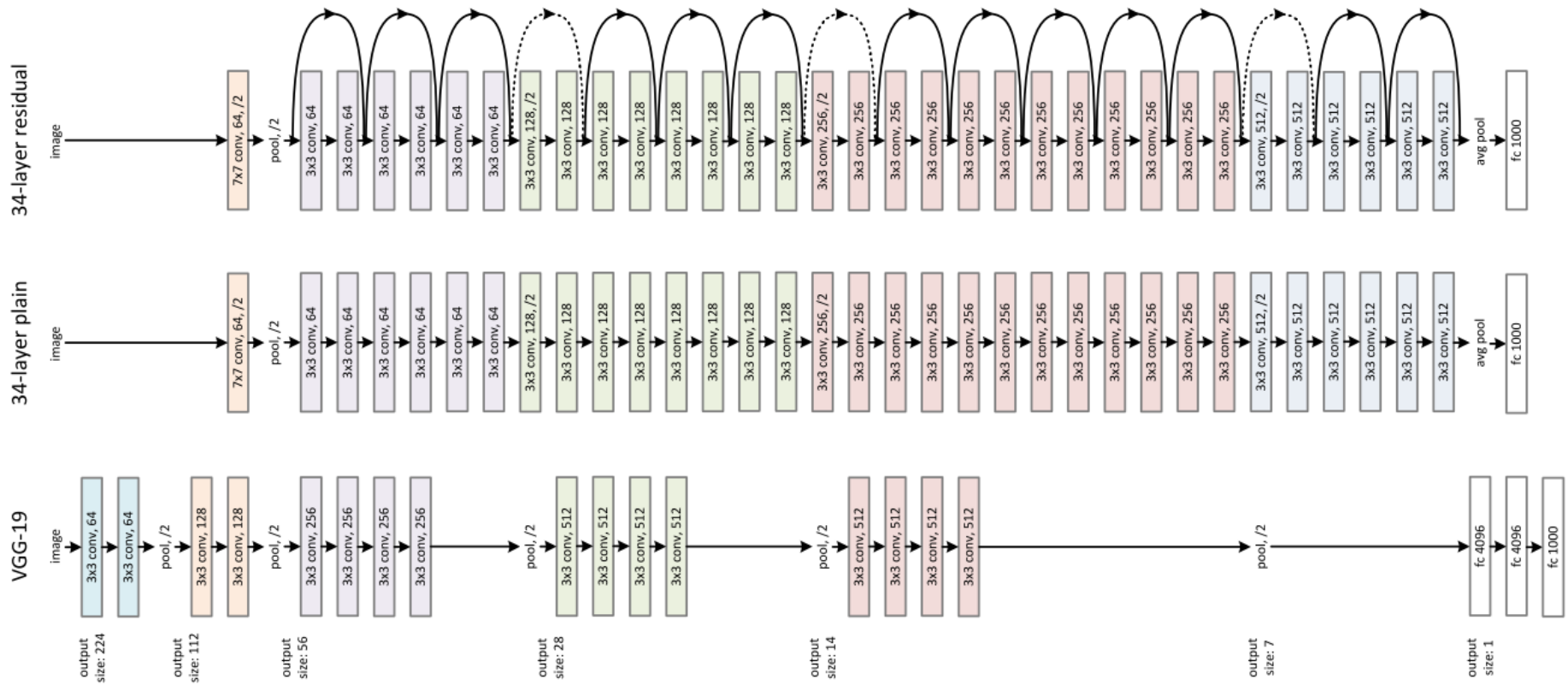
- Standard fully connected layer: full matrix W_i has $h_i h_{i+1}$ parameters
- Convolutional: Each neuron (entry of z_{i+1}) receives input from a square receptive field, with $k \times k$ parameters. All neurons *share* these parameters $\rightarrow$ *translation invariance*. The whole layer only has k^2 parameters.
- There are often multiple neurons with the same receitive field ("depth" of the layer), to represent different "filters". Stride leads to downsampling. Padding at borders.

- Pooling applies a predefined operation on the receptive field (no parameters): max or average. Typically for downsampling.

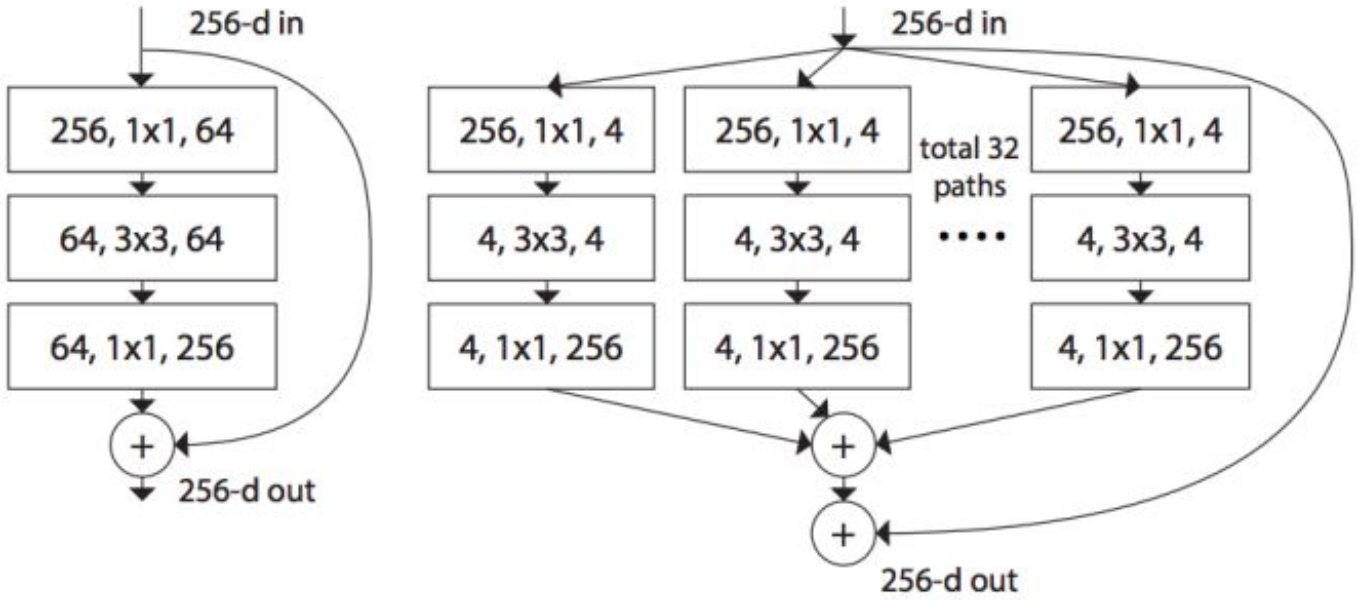
Learning to read these diagrams...



AlexNet


VGG-19
34-layer plain
34-layer residual
image
3x3 conv, 64
pool, /2
3x3 conv, 128
3x3 conv, 256
3x3 conv, 512
output size: 224
output size: 112
output size: 56
output size: 28
output size: 14
output size: 7
output size: 1
fc 4096
fc 1000
7x7 conv, 64, /2
3x3 conv, 128, /2
3x3 conv, 256, /2
3x3 conv, 512, /2
avg pool

ResNet


256-d in
256, 1x1, 64
64, 3x3, 64
64, 1x1, 256
+
256-d out
256-d in
256, 1x1, 4
256, 1x1, 4
total 32 paths
256, 1x1, 4
4, 3x3, 4
4, 3x3, 4
4, 3x3, 4
4, 1x1, 256
4, 1x1, 256
4, 1x1, 256
+
+
256-d out

ResNeXt

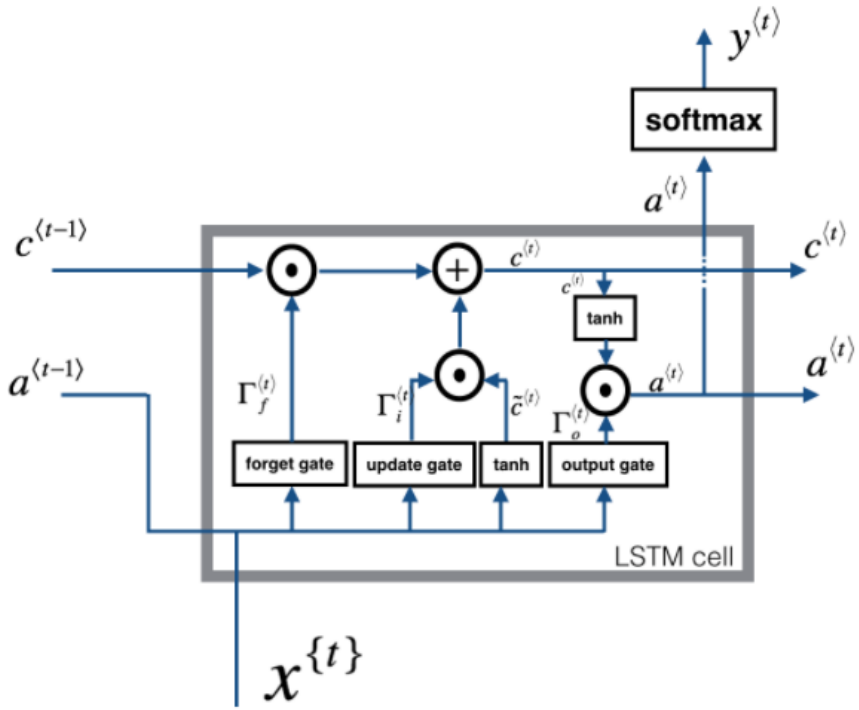
Pretrained networks

- ImageNet5k, AlexNet, VGG, ResNet, ResNeXt

LSTMs

2 - Long Short-Term Memory (LSTM) network

This following figure shows the operations of an LSTM-cell.



$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$
$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\tilde{c}^{\langle t \rangle} = \tanh(W_C[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_C)$$
$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}$$
$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$
$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

Figure 4: LSTM-cell. This tracks and updates a "cell state" or memory variable $c^{\langle t \rangle}$ at every time-step, which can be different from $a^{\langle t \rangle}$.

LSTM

- c is a memory signal, that is multiplied with a sigmoid signal Γ_f. If that is saturated ($\Gamma_f \approx 1$), the memory is preserved; and backpropagation copies gradients back
- If Γ_i is close to 1, a new signal $\tilde{c}$ is written into memory
- If Γ_o is close to 1, the memory contributes to the normal neural activations a

Collateral Benefits of NNs

- The *differentiable computation graph paradigm*
  - Perhaps a new paradigm to design large scale systems, beyond what software engineering teaches classically
- NN Diagrams as a specification language of models
  - “Click your Network Together”
  - High expressiveness to be creative in formulating really novel methods (e.g., Autoencoders, Embed2Control, GANs)

Optimization: Stochastic Gradient Descent

- Standard optimization methods (gradient descent backtracking line search, L-BFGS, other (quasi) Newton methods) have strong guarantees, but require exact gradients.
  - But computing exact gradients of the loss $\mathcal{L}(f, D)$ would require to go through the full data set D – for *every* gradient evaluation. That does not scale to big data.

- Instead, use stochastic gradient descent, where the gradient is computed only for a batch $\hat{D} \subsetneq D$ of fixed size k, subsampled uniformly from the whole D.

- Core reference:
  Yurii Nesterov (1983): *A method for solving the convex programming problm with convergence rate* $O(1/k^2)$
  Y Nesterov (2013): *Introductory lectures on convex optimization: A basic course* Springer

- See also:
  Mahsereci & Hennig (NIPS'15): *Probabilistic line searches for stochastic optimization*

ADAM

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

arXiv:1412.6980

(all operations interpreted element-wise)

Deep RL

- Value Network
- Advantage Network
- Action Network
- Experience Replay (prioritized)
- Fixed Q-targets
- etc, etc

Conclusions

- Conventional feed-forward neural networks are by no means magic. They're a parameterized function, which is fit to data.
- Convolutional NNs do make strong and good assumptions about how information processing on images should be structured. The results are great and related to some degree to human visual representations. A large part of the success of deep learning is on images.
  Also LSTMs make good assumptions about how memory signals help represent time series.
  The flexibility of "clicking together" network structures and general differentiable computation graphs is great.
  All these are innovations w.r.t. *formulating structured models* for ML
- The major strength of NNs is in their capacity and that, using massive parallelized computation, they can be trained on tons of data. Maybe they don't even need to be better than nearest neighbor lookup, but they can be queried much faster.