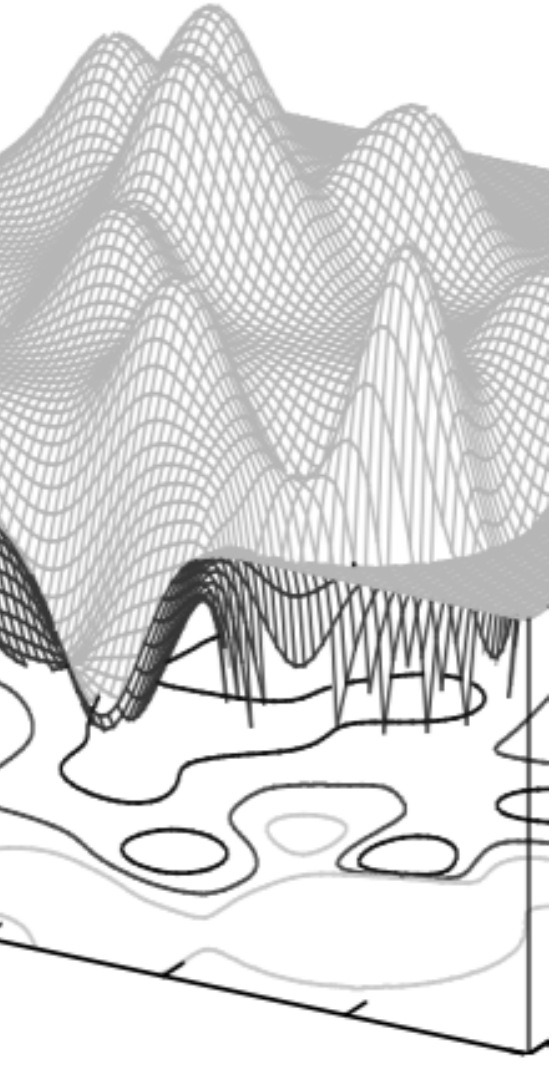


Machine Learning

Neural Networks

NN models, objectives & regularization, training, stochastic gradient descent, computation graphs, images & sequences, architectures

Marc Toussaint
University of Stuttgart
Summer 2019

Outline

- Model, Objective, Solver:
  - How do NNs represent a function $f(x)$, or discriminative function $f(y, x)$?
  - What are objectives? (standard objectives, different regularizations)
  - How are they trained? (Initialization, SGD)
- Computation Graphs & Chain Rules
- Images & Sequences
  - CNNs
  - LSTMs & GRUs
  - Complex architectures (e.g. Mask-RCNN, dense pose prediction, etc)

Neural Network models

- NNs are a parameterized function $f_\beta : \mathbb{R}^d \mapsto \mathbb{R}^M$
  - β are called weights
  - Given a data set $D = \{(x_i, y_i)\}_{i=1}^n$, we minimize some loss

$$\beta^* = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{n} \ell(f_\beta(x_i), y_i) + \text{ regularization}$$

- In that sense, they just replace our previous model assumption $f(x) = \phi(x)^\top \beta$, the reset is "in principle" the same

Neural Network models

- A (fwd-forward) NN $\mathbb{R}^{h_0} \mapsto \mathbb{R}^{h_L}$ with L layers, each h_l-dimensional, defines the function

  1-layer: $f_\beta(x) = W_1 x + b_1$, $\quad W_1 \in \mathbb{R}^{h_1 \times h_0}, b_1 \in \mathbb{R}^{h_1}$

  2-layer: $f_\beta(x) = W_2 \sigma(W_1 x + b_1) + b_2$, $\quad W_l \in \mathbb{R}^{h_l \times h_{l\text{-}1}}, b_l \in \mathbb{R}^{h_l}$

  L-layer: $f_\beta(x) = W_L \sigma(\cdots \sigma(W_1 x + b_1) \cdots) + b_L$

- The parameter $\beta = (W_{1:L}, b_{1:L})$ is the collection of all **weights** $W_l \in \mathbb{R}^{h_l \times h_{l\text{-}1}}$ and **biases** $b_l \in \mathbb{R}^{h_l}$
- To describe the mapping as an iteration, we introduce notation for the intermediate values:
  - the **input** to layer l is $z_l = W_l x_{l\text{-}1} + b_l \in \mathbb{R}^{h_l}$
  - the **activation** of layer l is $x_l = \sigma(z_l) \in \mathbb{R}^{h_l}$

  Then the L-layer NN model can be computed using the **forward propagation:**

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

  where $x_0 \equiv x$ is the input, and $f_\beta(x) \equiv z_L$ the output

Neural Network models

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

  rectified linear unit (ReLU) $\quad \sigma(z) = [z]_+ = \max\{0, z\} = z[z \geq 0]$

  leaky ReLU $\quad \sigma(z) = \max\{0.01z, z\} = \begin{cases} 0.01z & z < 0 \\ z & z \geq 0 \end{cases}$

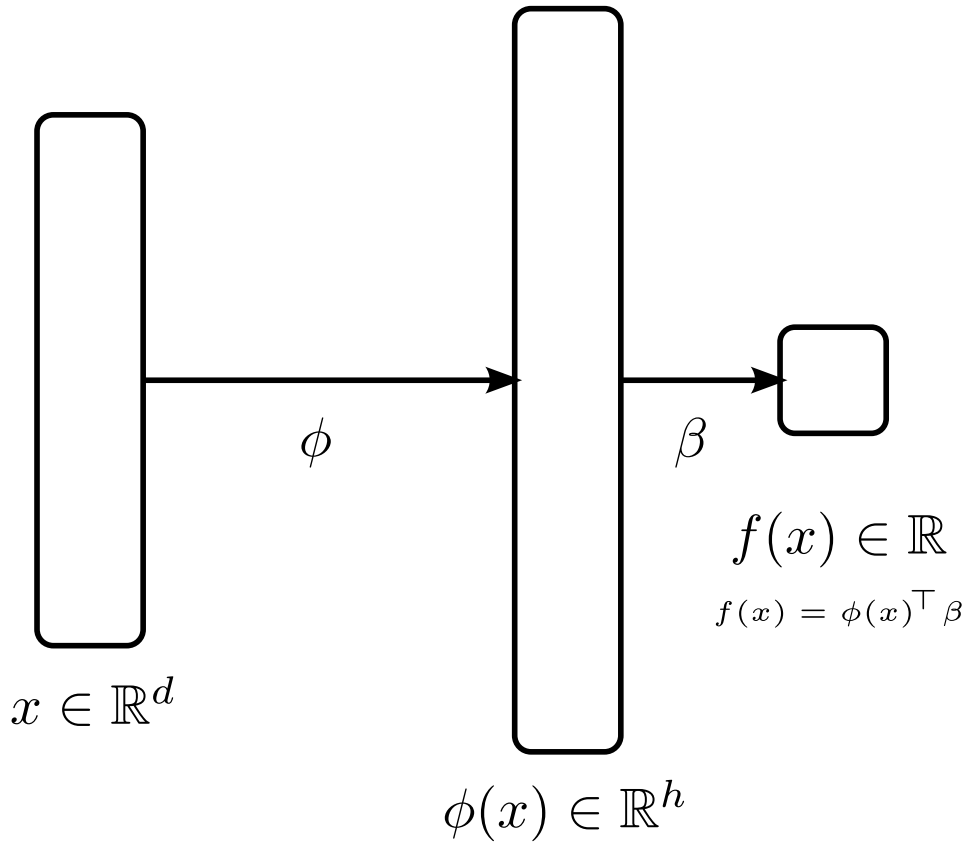
  sigmoid, logistic $\quad \sigma(z) = 1/(1 + e^{-z})$

  tanh $\quad \sigma(z) = \tanh(z)$

- L-layer means $L - 1$ hidden layers plus 1 output layer. (The input x_0 is not counted.)

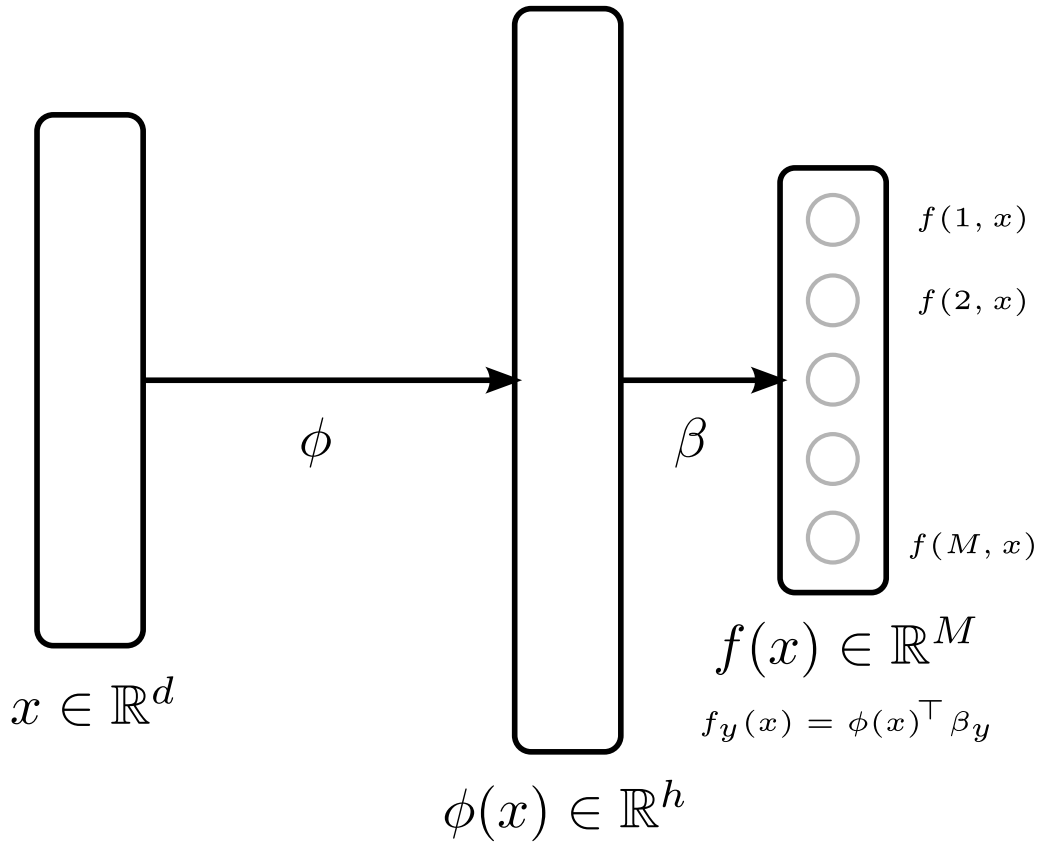
- The forward propagation therefore iterates applying
  - a linear transformation $x_{l\text{-}1} \mapsto z_l$, highly parameterized with W_l, b_l
  - a non-linear transformation $z_l \mapsto x_l$, element-wise and without parameters

feature-based regression



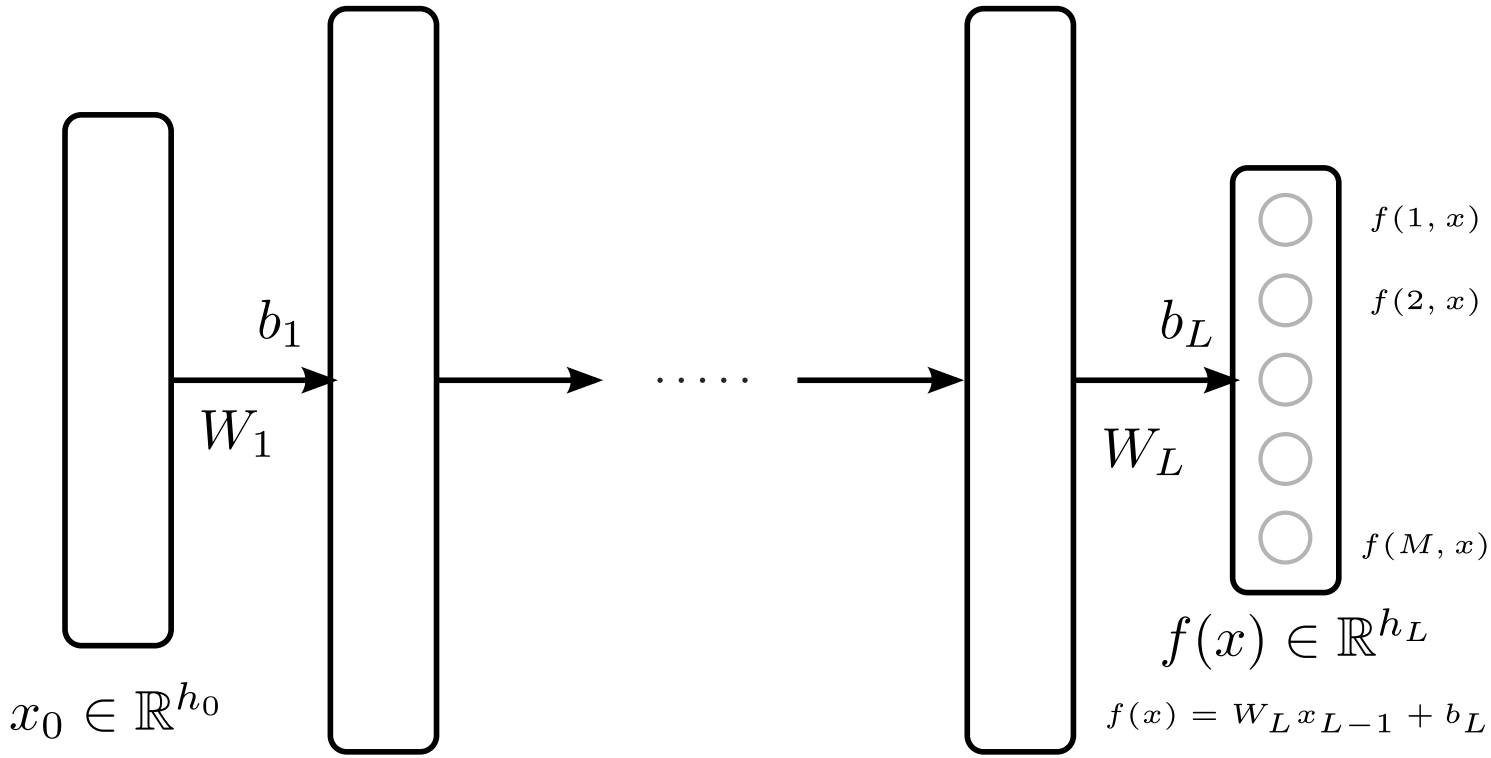
feature-based classification

(same features for all outputs)



neural network

(notation: $x_0 \equiv x$, $h_0 \equiv d$, $h_L \equiv M$, $x_{L-1} \equiv \phi(x)$)



Neural Network models

- We can think of the second-to-last layer $x_{L\text{-}1}$ as a feature vector

$$\phi_\beta(x) = x_{L\text{-}1}$$

- This aligns NNs models with what we discussed so far. But the crucial difference is:
  In NNs, the features $\phi_\beta(x)$ are also parameterized and trained!
  While in previous lectures, we had to fix $\phi(x)$ by hand, NNs allow us to learn features and **intermediate representations**

- Note: It is a common approach to train NNs as usual, but after training fix the trained features $\phi(x)$ ("remove the head (=output layer) and fix the remaining body of the NN") and use these trained features for similar problems or other kinds of ML on top.

NNs as unversal function approximators

- A 1-layer NN is linear in the input
- Already a 2-layer NN with $h_1 \to \infty$ hidden neurons is a universal function approximator
  - Corresponds to $k \to \infty$ features $\phi(x) \in \mathbb{R}^k$ that are well tuned

Objectives to train NNs

- loss functions
- regularization

Loss functions as usual

- Squared error regression, for 1-dim or h_L-dimensional:
  - for a single data point (x, y^*), $\ell(f(x), y^*) = (f(x) - y^*)^2$
  - the loss gradient is $\frac{\partial \ell}{\partial f} = 2(f - y^*)^\top$

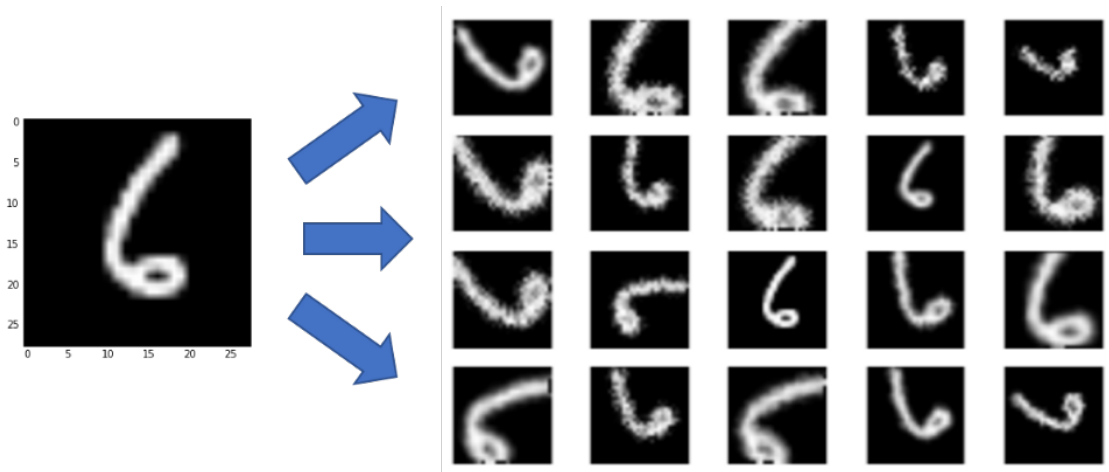
- For multi-class classification we have $h_L = M$ outputs, and $f_\beta(x) \in \mathbb{R}^M$ represents the discriminative function
- Neg-log-likelihood or cross entropy loss:
  - for a single data point (x, y^*), $\ell(f(x), y^*) = -\log p(y^*|x)$
  - the loss gradient at output y is $\frac{\partial \ell}{\partial f_y} = p(y|x) - [y = y^*]$
- One-vs-all hinge loss:
  - for a single data point (x, y^*), $\ell(f(x), y^*) = \sum_{y \neq y^*} [1 - (f_{y^*} - f_y)]_+$,
  - the loss gradient at non-target outputs $y \neq y^*$ is $\frac{\partial \ell}{\partial f_y} = [f_{y^*} < f_y + 1]$
  - the loss gradient at the target output y^* is $\frac{\partial \ell}{\partial f_{y^*}} = -\sum_{y \neq y^*} [f_{y^*} < f_y + 1]$

New types of regularization

- Conventional, add a L_2 or L_1 regularization.
  - adds a penalty $\lambda W_{l,ij}^2$ (Ridge) or $\lambda |W_{l,ij}|$ (Lasso) for every weight
  - In practise, compute the unregularized gradient as usual, 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 (executed after the unregularized weight update) just decays weights
- Dropout
  - *Srivastava et al: Dropout: a simple way to prevent neural networks from overfitting, JMLR 2014.*
  - "a way of approximately combining exponentially many different neural network architectures efficiently"
  - "p can simply be set at 0.5, which seems to be close to optimal for a wide range ofnetworks and tasks"
  - on test/prediction time, take true averages
- Others:
  - Data Augmentation
  - Training ensembles, bagging (averaging bootstrapped models)
  - Additional embedding objectives (e.g. semi-supervised embedding)
  - Early stopping

Data Augmentation

- A very interesting form of regularization is to modify the data!
- Generate more data by applying invariances to the given data. The model then learns to generalize as described by these invariances.
- This is a form of regularization that directly incorporates expert knowledge



Optimization

Computing the gradient

- Recall **forward propagation** in an L-layer NN:

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

- For a single data point (x, y^*), assume we have a loss $\ell(f(x), y^*)$
  We define $\delta_L \triangleq \frac{d\ell}{df} = \frac{d\ell}{dz_L} \in \mathbb{R}^{1\times M}$ as the gradient (as row vector) w.r.t. output values z_L.
- **Backpropagation:** We can recursivly compute the gradient $\frac{d\ell}{dz_l} \in \mathbb{R}^{1\times h_l}$ w.r.t. all other layers z_l as:

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

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

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

- This forward and backward computations are done for each data point (x_i, y_i).
- Since the total loss is the sum $L(\beta) = \sum_i \ell(f_\beta(x_i), y_i)$, the total gradient is the sum of gradients per data point.
- Efficient implementations send multiple data points (tensors) simultaneously through the network (fwd and bwd), which speeds up computations.

Optimization

- For small data size:
  We can compute the loss and its gradient $\sum_{i=1}^{n} \nabla_\beta \ell(f_\beta(x_i), y_i)$.
  - Use classical gradient-based optimization methods
  - default: L-BFGS, oldish but efficient: Rprop
  - Called **batch learning** (in contrast to online learning)

- For large data size: The $\sum_{i=1}^{n}$ is highly inefficient!
  - Adapt weights based on much smaller data subsets, **mini batches**

Stochastic Gradient Descent

- Compute the loss and gradient for a mini batch $\hat{D} \subset D$ of fixed size k.

$$L(\beta, \hat{D}) = \sum_{i \in \hat{D}} \ell(f_\beta(x_i), y_i)$$
$$\nabla_\beta L(\beta, \hat{D}) = \sum_{i \in \hat{D}} \nabla_\beta \ell(f_\beta(x_i), y_i)$$

- Naive Stochastic Gradient Descent, iterate

$$\beta \leftarrow \beta - \eta \nabla_\beta L(\beta, \hat{D})$$

  - Choice of learning rate η is crucial for convergence!
  - Exponential cooling: $\eta = \eta_0^t$

Stochastic Gradient Descent

- SGD with momentum:

$$\Delta\beta \leftarrow \alpha\Delta\beta - \eta\nabla_\beta L(\beta, \hat{D})$$
$$\beta \leftarrow \beta + \Delta\beta$$

- Nesterov Accelerated Gradient ("Nesterov Momentum"):

$$\Delta\beta \leftarrow \alpha\Delta\beta - \eta\nabla_\beta L(\beta + \Delta\beta, \hat{D})$$
$$\beta \leftarrow \beta + \Delta\beta$$

Yurii Nesterov (1983): *A method for solving the convex programming problm with convergence rate* $O(1/k^2)$

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)

Adam & Nadam

- Adam interpretations (everything element-wise!):
  - $m_t \approx \langle g \rangle$ the mean gradient in the recent iterations
  - $v_t \approx \langle g^2 \rangle$ the mean gradient-square in the recent iterations
  - $\hat{m}_t, \hat{v}_t$ are bias corrected (check: in first iteration, $t = 1$, we have $\hat{m}_t = g_t$, unbiased, as desired)
  - $\Delta\theta \approx -\alpha \frac{\langle g \rangle}{\langle g^2 \rangle}$ *would* be a Newton step if $\langle g^2 \rangle$ *were* the Hessian...

- Incorporate Nesterov into Adam: Replace parameter update by

$$\theta_t \leftarrow \theta_{t\text{-}1} - \alpha/(\sqrt{\hat{v}_t} + \epsilon) \cdot (\beta_1 \hat{m}_t + \frac{(1-\beta_1) g_t}{1-\beta_1^t})$$

Dozat: *Incorporating Nesterov Momentum into Adam*, ICLR'16

Initialization

- The Initialization of weights is important! Heuristics:

- Choose random weights that don't grow or vanish the gradient:
  - E.g., initialize weight vectors in $W_{l,i\cdot}$ with standard deviation 1, i.e., each entry with sdv $\frac{1}{\sqrt{h_{l\text{-}1}}}$
  - Roughly: If each element of z_l has standard deviation ϵ, the same should be true for z_{l+1}.

- Choose each weight vector $W_{l,i\cdot}$ to point in a uniform random direction $\rightarrow$ same as above

- Choose biases $b_{l,i}$ randomly so that the ReLU hinges cover the input well (think of distributing hinge features for continuous piece-wise linear regression)

Brief Discussion

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. What changed is the metrics by which they’re are evaluated:

- NNs did not become “better” than they were 20y ago. What changed is the metrics by which they’re are evaluated:
- Old:
  - Sample efficiency & generalization; get the most from little data
  - Guarantees (both, w.r.t. generalization and optimization)
  - **generalize** much better than nearest neighbor
- New:
  - Ability to cope with billions of samples

  $\rightarrow$ no batch processing, but **stochastic** optimization (Adam) without monotone convergence

  $\rightarrow$ nearest neighbor methods infeasible, **compress** data into high-capacity NNs

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. Sample efficiency and the precise generalization behavior beome less relevant.
- That's how the metrics changed from '90-'10 to nowadays

Computation Graphs

- A great collateral benefit of NN research!
- Perhaps a new paradigm to design large scale systems, beyond what software engineering teaches classically
- [see section 3.2 in “Maths” lecture]

Example

- Three real-valued quantities x, g and f which depend on each other:

$$f(x, g) = 3x + 2g \quad \text{and} \quad g(x) = 2x \ .$$

What is $\frac{\partial}{\partial x} f(x, g)$ and what is $\frac{d}{dx} f(x, g)$?

Example

- Three real-valued quantities x, g and f which depend on each other:

$$f(x, g) = 3x + 2g \quad \text{and} \quad g(x) = 2x \ .$$

  What is $\frac{\partial}{\partial x} f(x, g)$ and what is $\frac{d}{dx} f(x, g)$?

- The *partial* derivative only considers a single function $f(a, b, c, ..)$ and asks how the output of this single function varies with one of its arguments. (Not caring that the arguments might be functions of yet something else).
- The *total* derivative considers full networks of dependencies between quantities and asks how one quantity varies with some other.

Computation Graphs

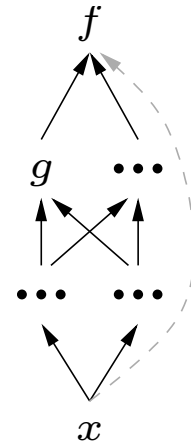
- A **function network** or **computation graph** is a DAG of n quantities x_i where each quantity is a deterministic function of a set of parents $\pi(i) \subset \{1, .., n\}$, that is

$$x_i = f_i(x_{\pi(i)})$$

  where $x_{\pi(i)} = (x_j)_{j \in \pi(i)}$ is the tuple of parent values
- (This could also be called *deterministic Bayes net*.)

- Total derivative: Given a variation dx of some quantity, how would all child quantities (down the DAG) vary?

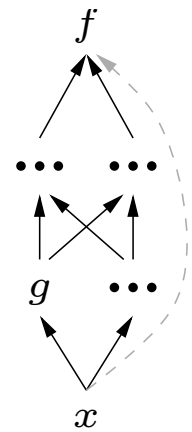
Chain rules



- Forward-version: (I use in robotics)

$$\frac{df}{dx} = \sum_{g \in \pi(f)} \frac{\partial f}{\partial g} \frac{dg}{dx}$$

Why "forward"? You've computed $\frac{dg}{dx}$ already, now you move forward to $\frac{df}{dx}$.
Note: If $x \in \pi(f)$ is also a direct argument to f, the sum includes the term $\frac{\partial f}{\partial x} \frac{dx}{dx} \equiv \frac{\partial f}{\partial x}$.
To emphasize this, one could also write $\frac{df}{dx} = \frac{\partial f}{\partial x} + \sum_{\substack{g \in \pi(f) \\ g \neq x}} \frac{\partial f}{\partial g} \frac{dg}{dx}$.



- Backward-version: (used in NNs!)

$$\frac{df}{dx} = \sum_{g: x \in \pi(g)} \frac{df}{dg} \frac{\partial g}{\partial x}$$

Why "backward"? You've computed $\frac{df}{dg}$ already, now you move backward to $\frac{df}{dx}$.
Note: If $f \in \pi(g)$, the sum includes $\frac{df}{df} \frac{\partial f}{\partial x} \equiv \frac{\partial f}{\partial x}$. We could also write
$\frac{df}{dx} = \frac{\partial f}{\partial x} + \sum_{\substack{g: x \in \pi(g) \\ g \neq f}} \frac{df}{dg} \frac{\partial g}{\partial x}$.

Images & Time Series

Images & Time Series

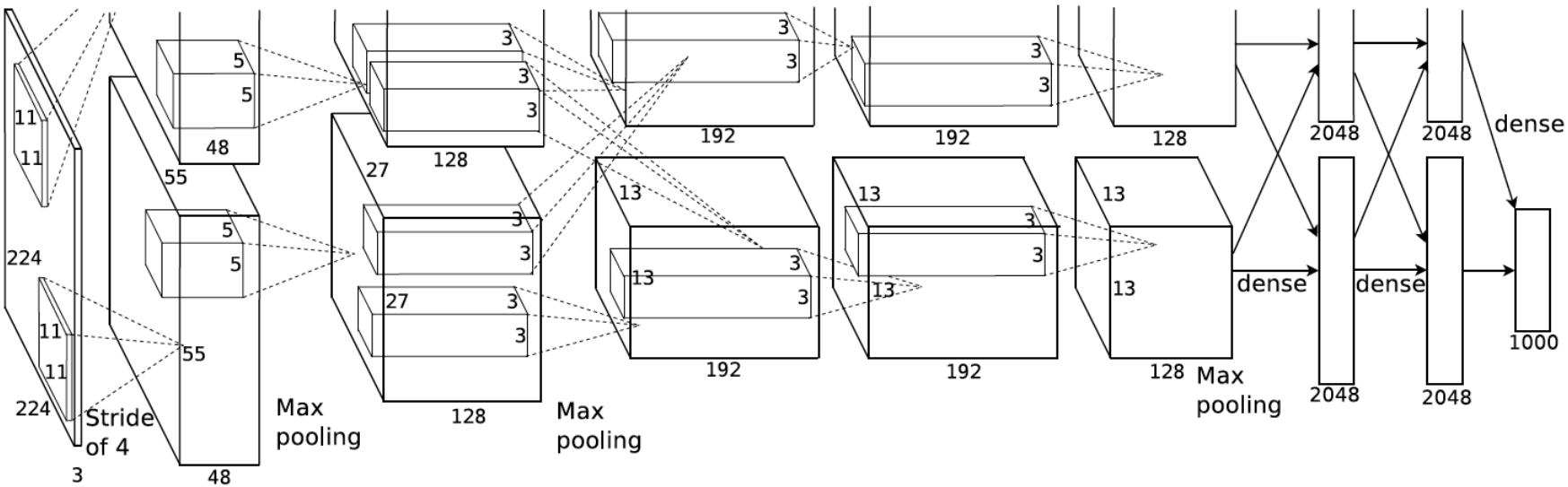
- My guess: 90% of the recent success of NNs is in the areas of images or time series
- For images, convolutional NNs (CNNs) impose a very sensible prior; the representations that emerge in CNNs are in fact similar to representations in the visual area of our brain.
- For time series, long-short term 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.

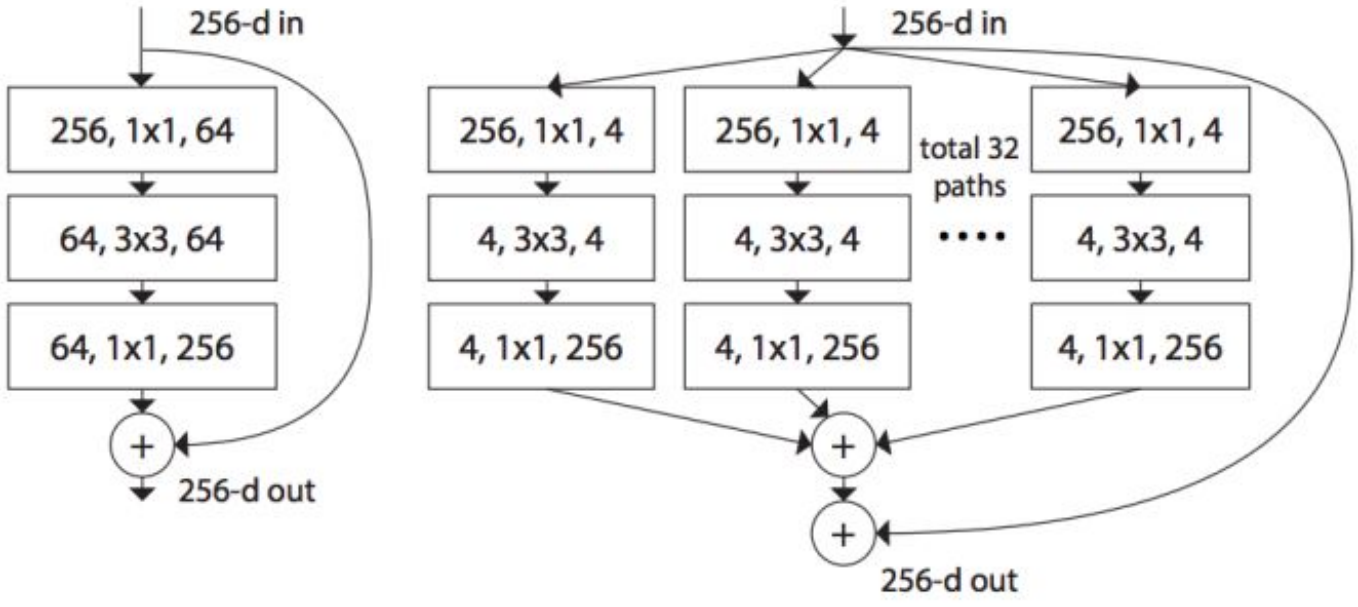
Learning to read these diagrams...



AlexNet

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

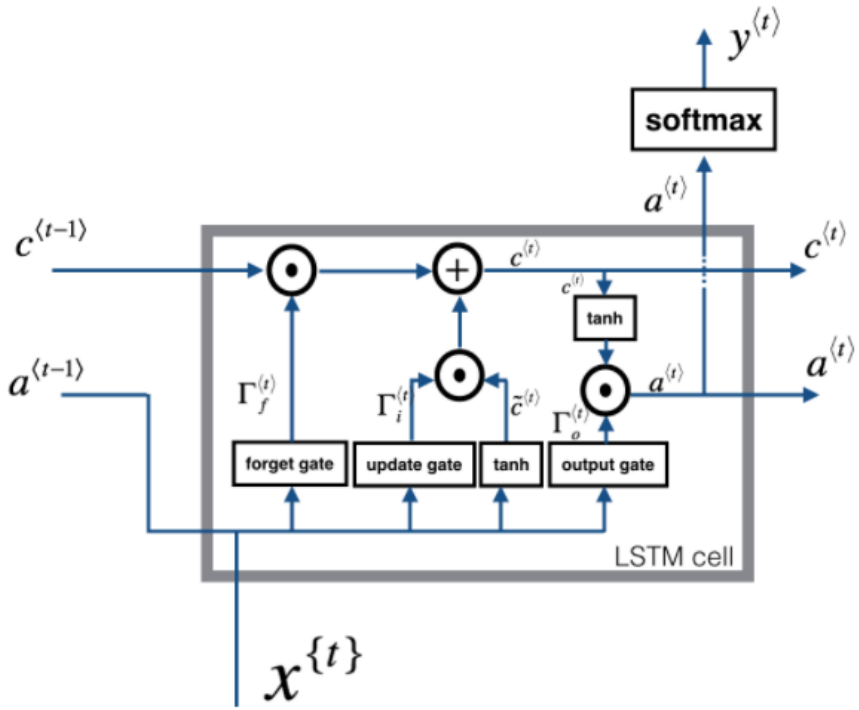
Pretrained networks

- 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

Gated Recurrent Units

- Cleaner and more modern: Gated Recurrent Units
  but perhaps just similar performance

- Gated Feedback RNNs

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.