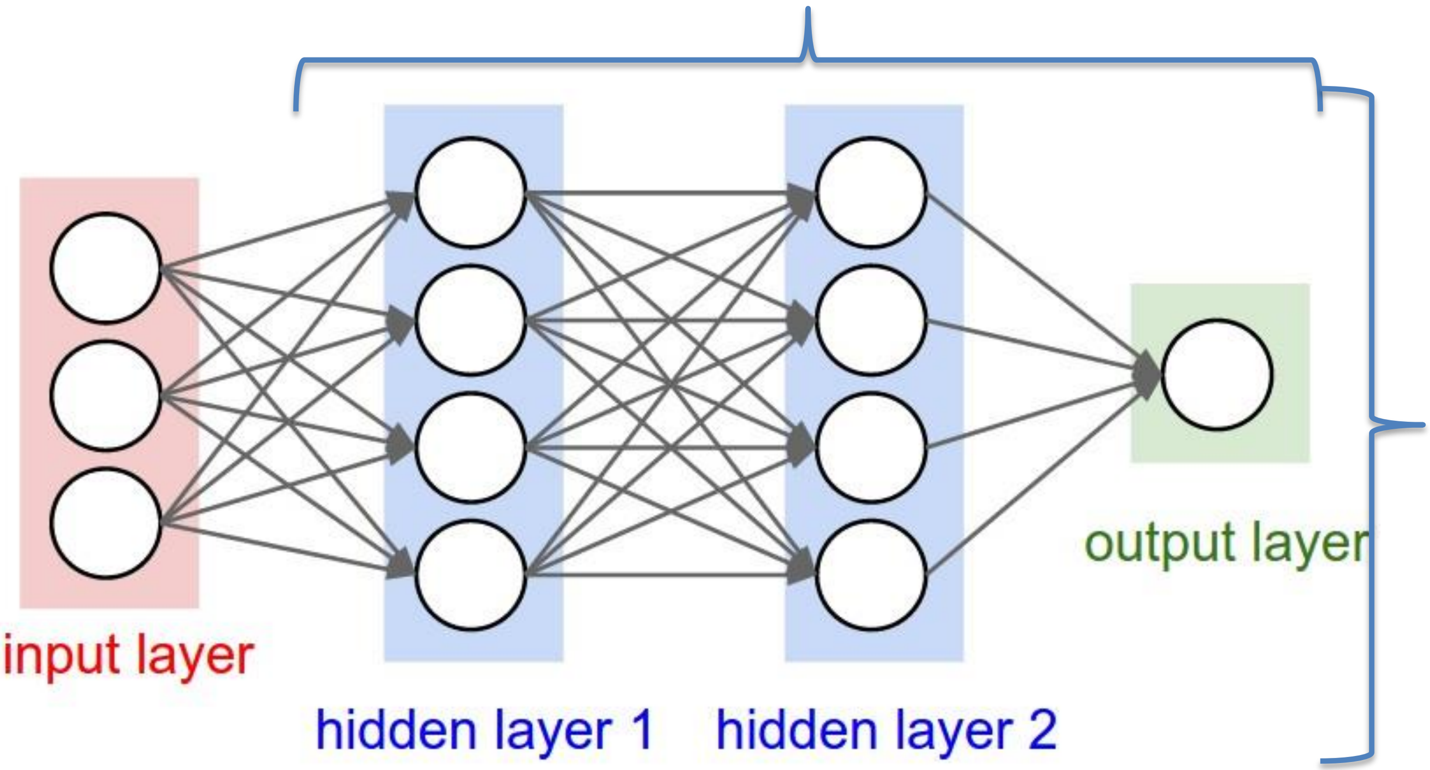
Neural Networks
Getting learning to work

Jan Chorowski
Instytut Informatyki
Wydział Matematyki i Informatyki Uniwersytet Wrocławski

2018

A basic neural network

Depth – how many layers?



Width – how many neurons per layer?

http://cs231n.github.io/neural-networks-1/

Motivation

Adding a hidden layer to a neural net makes a change:

No hidden units
(Lin & log regression)

- Linear relationships
- Easy (convex) optimization problem
- Can compute the gradient wrt. params in an easy way (backpropagation)

Hidden units
(Deep neural network)

- Universal approximation
- Hard (non-convex) optimization problem (local optimums, saddle points)
- Backpropagation works, but
  - Nonlinearities may make the grad nearly 0 or ∞!
  - Nets are sensitive to range of values that flow through it

Getting a neural net to train requires some care and design choices!

Where to get more information

- “**Efficient backprop**” by Y. Le Cun
- „Preactical Recommendation for Gradient-Based Training of Deep Architectures” Y. Bengio
- Slides by Geoff Hinton in his Coursera lectures:
  https://www.coursera.org/course/neuralnets
  and epsecially
  http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
- „Stochastic Gradient Descent Tricks” L. Bottou
- Slides from Stanford:
  http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf
  http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture7.pdf

Practical aspects of NN training

Neural nets read numbers and are scale sensitive.

Normalize inputs:

- Map each input to $[-1,1]$ range
- Or scale each input to have mean 0, variance 1
- Even better: use PCA to de-correlate the inputs
- For discrete inputs:
  - 1-of-N: $'a' \rightarrow [1,0,0], 'b' \rightarrow [0,1,0], 'c' \rightarrow [0,0,1]$
    please note: this turns the matrix multiplication into a look-up table. This is how you do word embeddings.
- Specific cases:
  - Thermometer : $1 \rightarrow [1,0,0], 2 \rightarrow [1,1,0], 3 \rightarrow [1,1,1]$
  - Transform angles using trig. functions, e.g.
    $\alpha \rightarrow [\sin(\alpha), \sin(\alpha + 120°), \sin(\alpha + 240°)]$
  - Similar approach possible for other periodic inputs!

Choose initial weights wisely

- Goal: the excitation of the neuron has 0 mean, st. dev 1.
- Inputs are OK - already normalized!
- Typical rules of thumb (for $\tanh()$ transfer):
  - $b_{init} = \bar{0}$
  - $W_{k_{init}} \sim Uniform\left[-\frac{1}{\sqrt{n_k}}, \frac{1}{\sqrt{n_k}}\right]$
- Or also account for fan-out:
  - $W_{k_{init}} \sim Uniform\left[-\frac{\sqrt{6}}{\sqrt{n_k+n_{k+1}}}, \frac{\sqrt{6}}{\sqrt{n_k+n_{k+1}}}\right]$

Glorot, Bengio, "Understanding the difficulty of training deep feedforward neural networks"

Batch normalization

http://arxiv.org/abs/1502.03167

- Normalize all activations in each minibatch!

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

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

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

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

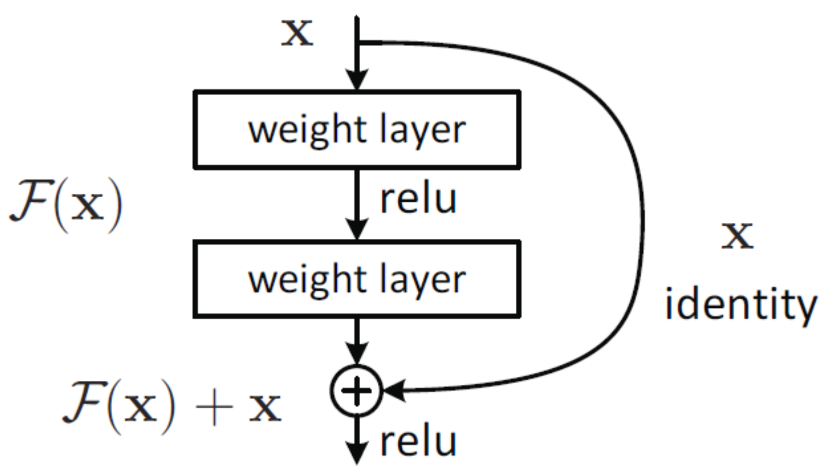
- BN was originally said to prevent covariate shift (during training the upper layer's inputs are constantly changing)
  - In the beginning the feature transforms are pretty much random
  - But get better over time: Upper layers shoot at moving targets!

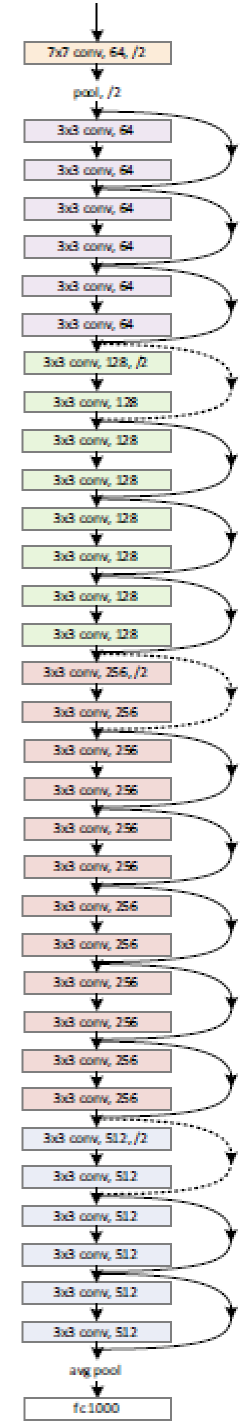
Avoid flat regions of the sigmoids

- For output layer and classification: use SoftMax and cross-entropy loss
- For inner layers:
  - Prefer tanh over log. sigmoid (tanh gives mean 0)
  - The contemporary default: ReLU http://machinelearning.wustl.edu/mlpapers/papers/icml2010_NairH10
  - Or Maxout http://arxiv.org/abs/1302.4389

Use gradient shortcuts

- Residual networks (SOTA Imagenet 2015): https://arxiv.org/abs/1512.03385
- Networks with more than 150 layers!





Learnable shortcuts: gating

Highway networks

https://papers.nips.cc/paper/5850-training-very-deep-networks.pdf

Sigmoid activation:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

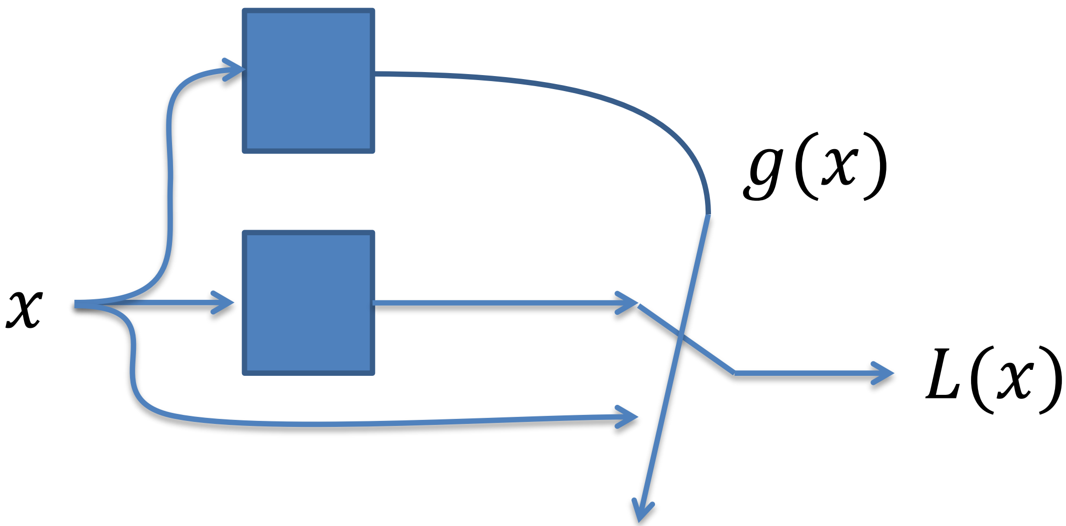
Gate:

$$g(x) = \sigma\left(W_g x + b_g\right)$$

Elemwise multiplication

Layer:

$$L(x) = g(x) \odot (Wx + b) + (1 - g) \odot x$$



Historical note:
Gating is used since
1997 in the LSTM cell!

Optimization objective

- Goal of training:

$$\text{find } \Theta^* = \arg\min_{\Theta} J(\Theta, X, Y)$$

- Batch Gradient Descent (BGD) algorithm

$$\Theta \leftarrow \Theta - \alpha \frac{\partial J}{\partial \Theta}$$

- The gradient has a structure

$$\frac{\partial J}{\partial \Theta} = \frac{1}{N} \sum_i \frac{\partial J\left(\Theta, X^{(i)}, Y^{(i)}\right)}{\partial \Theta} \approx \mathbb{E}_{x,y}\left[\frac{\partial J(\Theta, x, y)}{\partial \Theta}\right]$$

- Stochastic Gradient Descent (SGD) estimates $\frac{\partial J}{\partial \Theta}$ on mini-batches (1-100 samples)

Better Batch Algorithms

For smallish nets: The best: Newton method:

$$\theta := \theta - \alpha (H_\Theta(J))^{-1} \frac{\partial J}{\partial \Theta}$$

Where: $H_\Theta(J)$ is the Hessian (matrix of second derivatives:

$$H_{ij} = \frac{\partial^2 J}{\partial \Theta_i \partial \Theta_j}$$

Typical algorithm:

1. Choose step direction $(H_\Theta(J))^{-1} \frac{\partial J}{\partial \Theta}$
2. Choose the best step length α (line search)

:) Quick convergence (1 step for quadratic funs)
:(Hessian is large!! → $O(n^2)$ memory
:(Hessian inversion is costly → $O(n^3)$ time

Quasi Newton methods

- Many optimization methods approximate the Hessian.
- For NNets people have adapted:
  - Conjugate gradient
  - Levenberg Marquardt
- Usually a good generic quasi-Newton method works well
  - E.g. the L-BFGS method from scipy
- Quasi-Newton is good for smaller datasets without redundant samples.

Stochastic vs batch gradient

Batch Gradient Descent

- Few expensive and precise steps
- Use second order methods
- Sort of useful on small datasets
- Exploits parallel hardware

Stochastic Gradient Descent

- Many small steps
- Noisy
- Deals better with redundancies in data
- Scales well with large data sets

We typically use „mini-batches"

- Use about 100 examples for each step
- Better use of CPU/GPU (possibility to use parallel hardware)

Note:

batch size is driven by hardware, we may see larger minibatches

Tips for stochastic gradient descent

Learning rate selection (very important!!!):

- Think of golf clubs:
  first use a larger one, then a smaller one
- Constant – may not converge
- **Reduce after some iterations (staircase fun)**
  often combined with validation error monitoring :
  reduce LR when validation error stops to improve
- Some other annealing schedules
  - $\alpha_t = \alpha_0 \cdot c^t$ with $c \approx 0.995$
  - $\alpha_t = \frac{b}{c+t}$ or $\alpha_t = \frac{b}{t}$ or $\alpha_t = \alpha_0 \frac{\tau}{\max(t,\tau)}$
  - Adapt based on error on validation set (lower whenever validation loss didn't improve)

Speeding up SGD

- Momentum: https://distill.pub/2017/momentum/
  - Spread each update over many steps

$$V_t \leftarrow \mu_t V_{t-1} - \alpha_t \frac{\partial J}{\partial \Theta}$$
$$\Theta_t \leftarrow \Theta_{t-1} + V_t$$

  - Theoretical justification in the batch setting (Nesterov accelerated gradient)
  - Popular

Only use sign of the gradient

- Resilient Propagation (RProp, IRProp):
  - Use only the sign of the gradient
  - Rules for learning rate scaling
  - Only works in BGD
- RMSProp (Rprop for SGD)

$$r_t \leftarrow (1-\gamma)\left(\frac{\partial J}{\partial \Theta}\right)^2 + \gamma r_{t-1}$$

$$\Theta_t \leftarrow \Theta_t - \frac{\alpha_t}{\sqrt{r_t + \epsilon}}\frac{\partial J}{\partial \Theta}$$

- Keep a running average of the magnitude to get the sign

Adam – the recommended default

- RMSprop with momentum
- http://arxiv.org/abs/1412.6980

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)

Two more tricks for optimization

- Use many models:
  - Train many models, and average (ensemble)
  - Train 1 net, but once in a while use high LR to jump to another local optimum ("snapshot ensemble")
- Stabilize the weights by Polyak averaging
  - Keep a running average of parameters:

$$\Theta \leftarrow \text{normal train update}$$
$$\Theta^{ave} \leftarrow 0.995\Theta^{ave} + 0.005\Theta$$

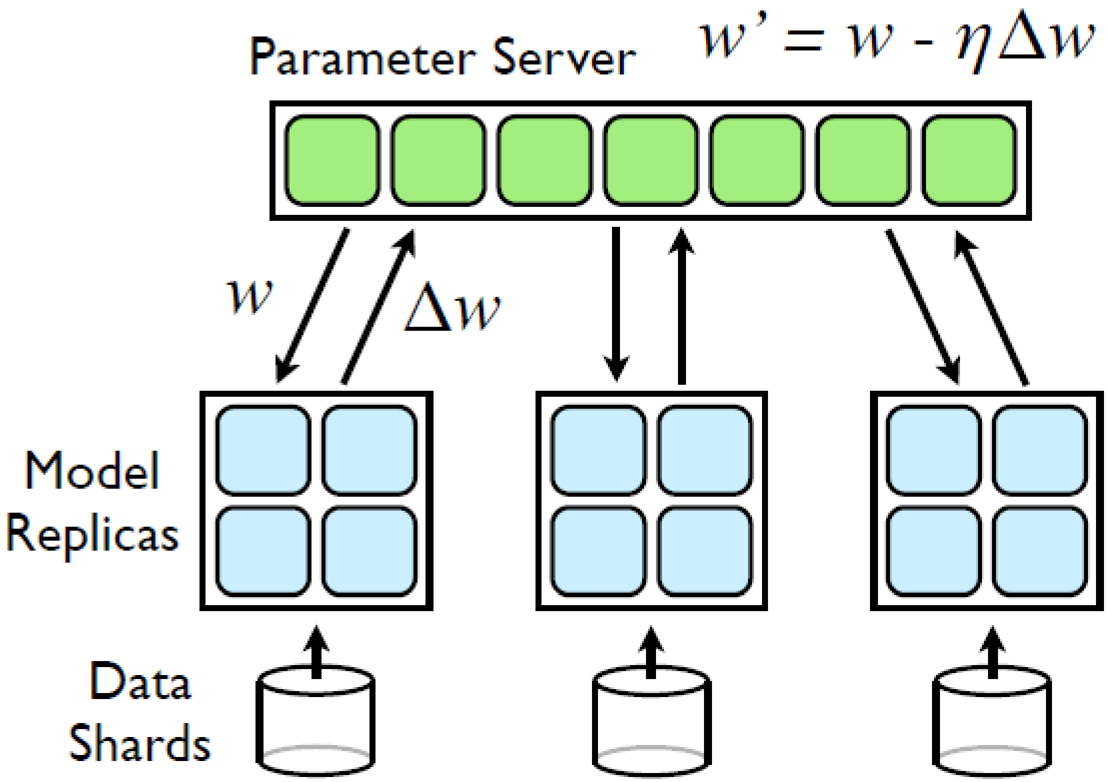
  Use Θ^{ave} for testing & profit!

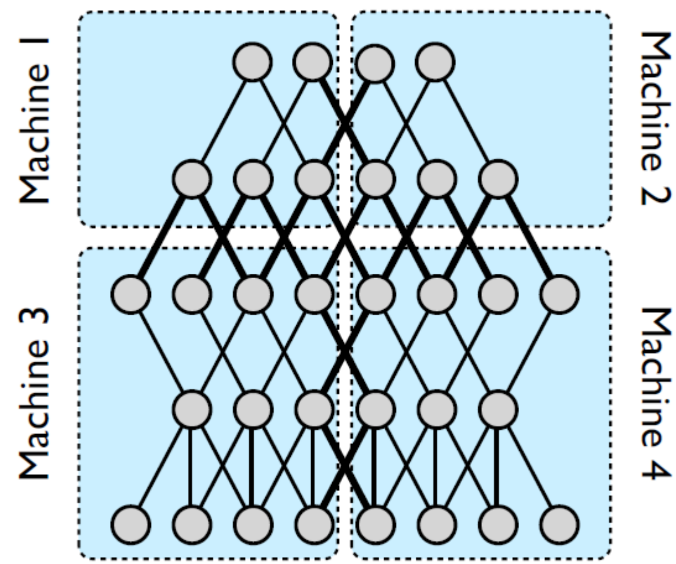
Special topic: how to use many cores

- Use large minibatches (whatever that means)
- Need proper learning rate and momentum scaling
- Hot research topic due to hardware trends
- When aggregating gradients across machines:
  - Start more workers than needed
  - Make step when 90% of workers have finished (don’t wait for stragglers)

Alterative: asynchronous distributed training

http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks





What to monitor during training

- Loss
- Magnitude (mean and st. dev) of gradient for (best for each layer)
- Activation values (mean, st. dev.) for each layer (monitors whether units are “stuck”)
- Good practices:
  - Save the model during training.
  - Make the analysis from the saved models -> less clutter in the optimization code.

When to stop training?

- Loss function doesn't improve sufficiently
- Typically the error will decrease on the training set, but increase on a test/validation set
  - Use an auxiliary (validation) dataset to monitor error and stop when validation error doesn't improve – early stopping prevents over-fitting to training data
- Small gradient
- But with SGD we expect some steps in the wrong direction !?
  - Patience algorithm:

```
Set patience to some initial value
While iters < patience
        if significant improvement of some criterion
                patience = max(patience, iter*2)
```