On Deep Learning

Alexander G. Ororbia II

Introduction to Machine Learning

CSCI-736

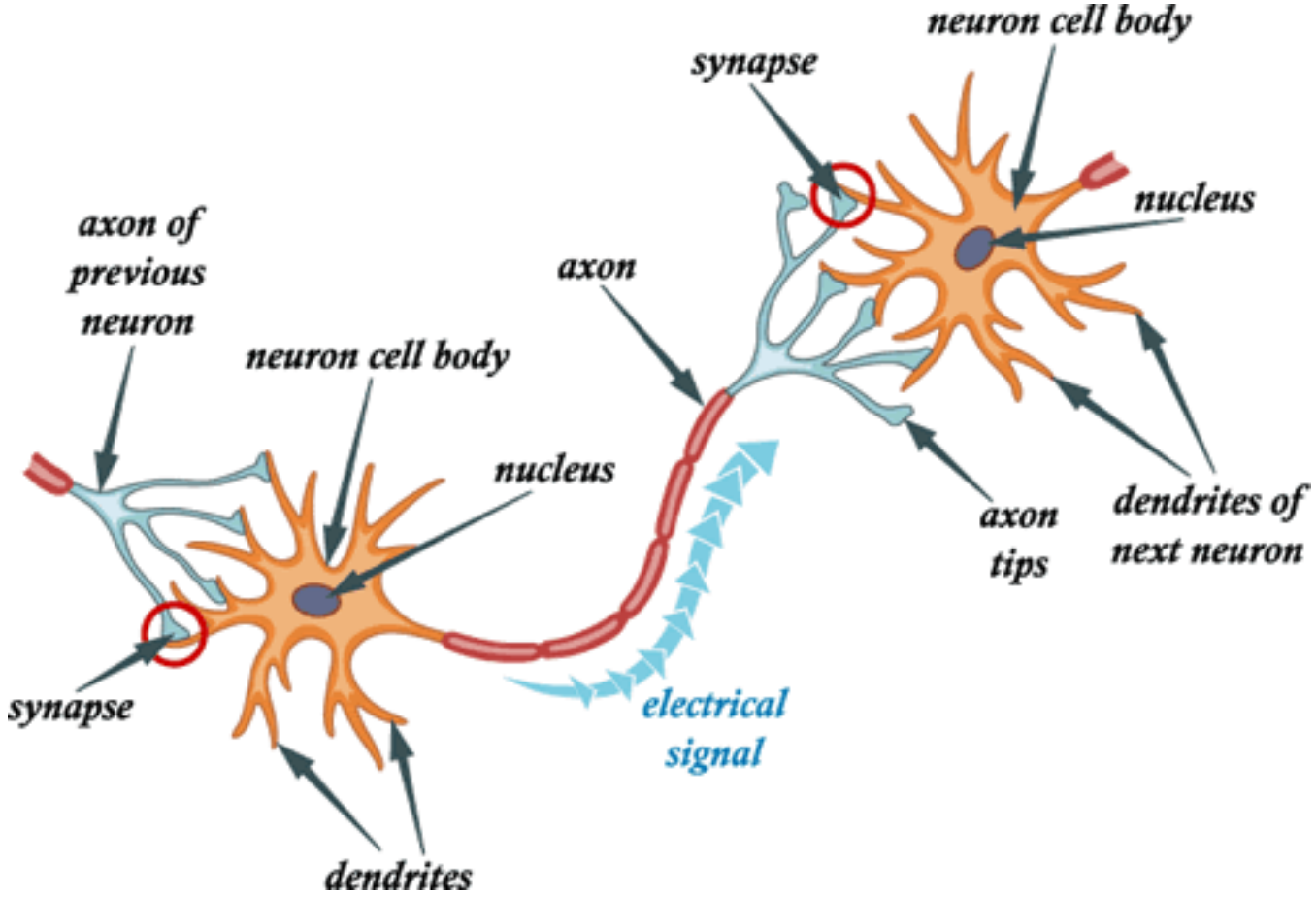
1/24/2023

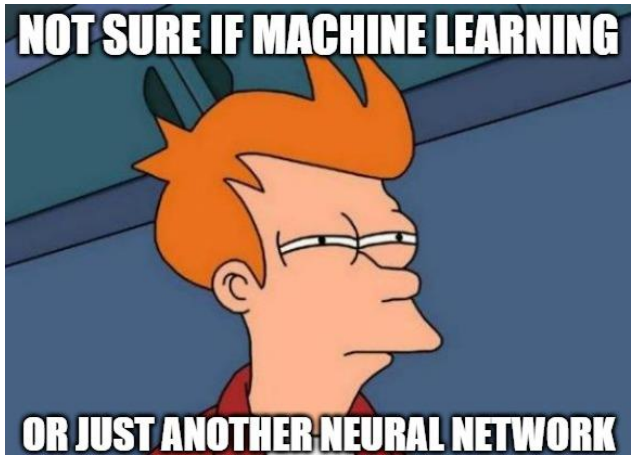
Companion reading:

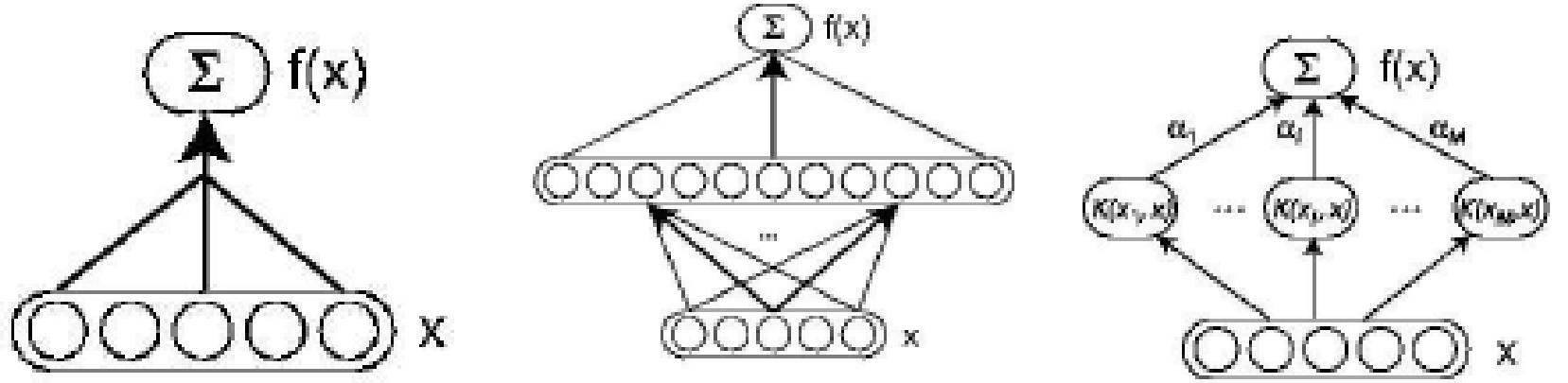
Chapter 6-8 of Deep Learning textbook

Artificial Neural Networks (ANNs): Neurobiological Motivations

- **Human brain** = a good candidate learning algorithm
  - Evidence of layered architectures in neuro-scientific research (i.e., cortical structures)
- Early success of specialized yet deep architectures
  - Convolutional Networks, NeoCognitron





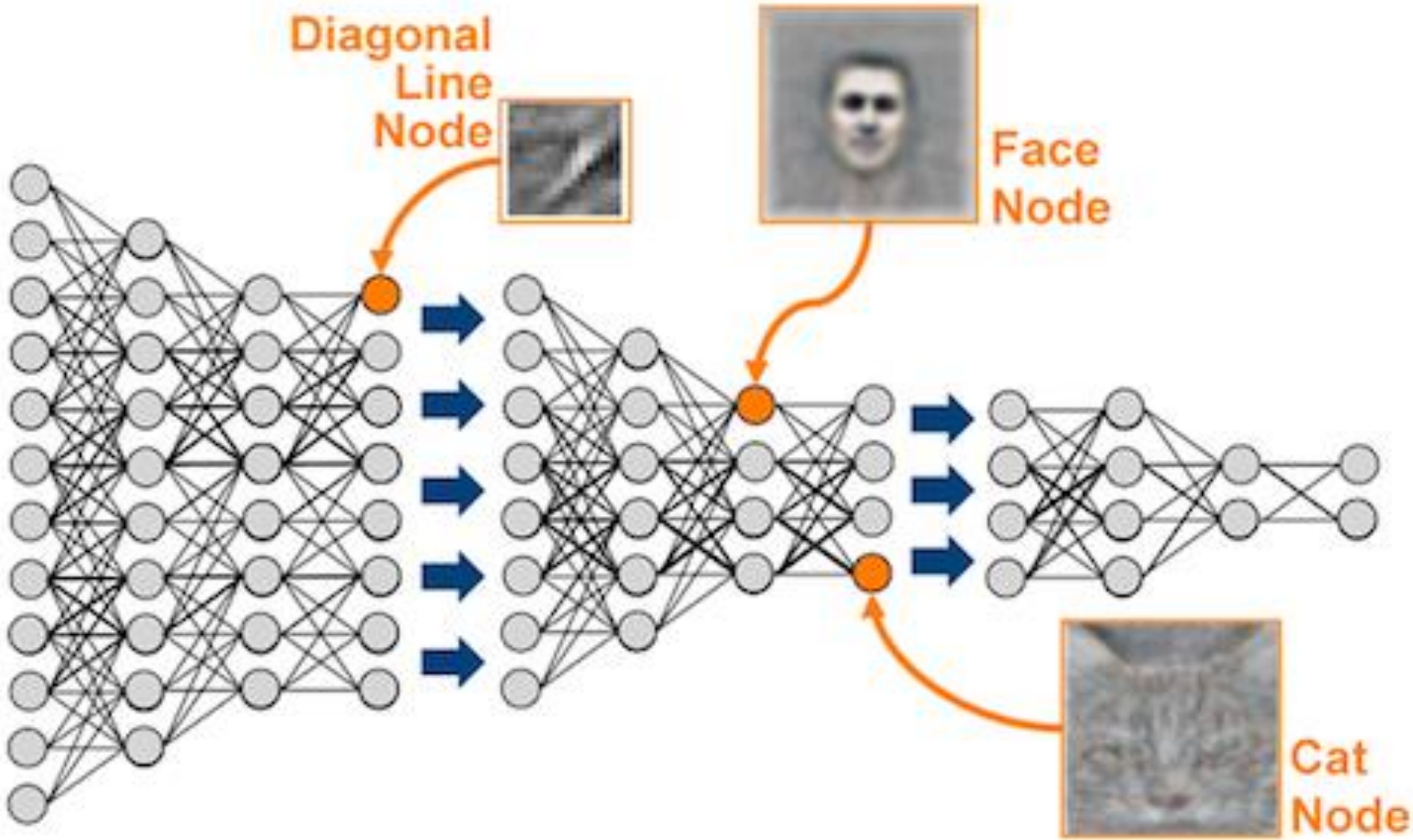


(a) Linear model architecture

(b) Single layer neural network architecture

(c) Kernel SVM architecture

Most of machine learning models can be viewed as a type of ANN…if you squint hard enough…


Diagonal
Line
Node
Face
Node
Cat
Node

Background

A Recipe for Machine Learning

1. Given training data:

$$\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$$

2. Choose each of these:

- Decision function

$$\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

- Loss function

$$\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R}$$

3. Define goal:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

4. Train with SGD:

(take small steps opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

Background

A Recipe for

1. Given training dat

$$\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$$

2. Choose each of t

– Decision functio

$$\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

– Loss function

$$\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R}$$

$y_i)$

opposite the gradient)

$$\theta^{(t} \quad {}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

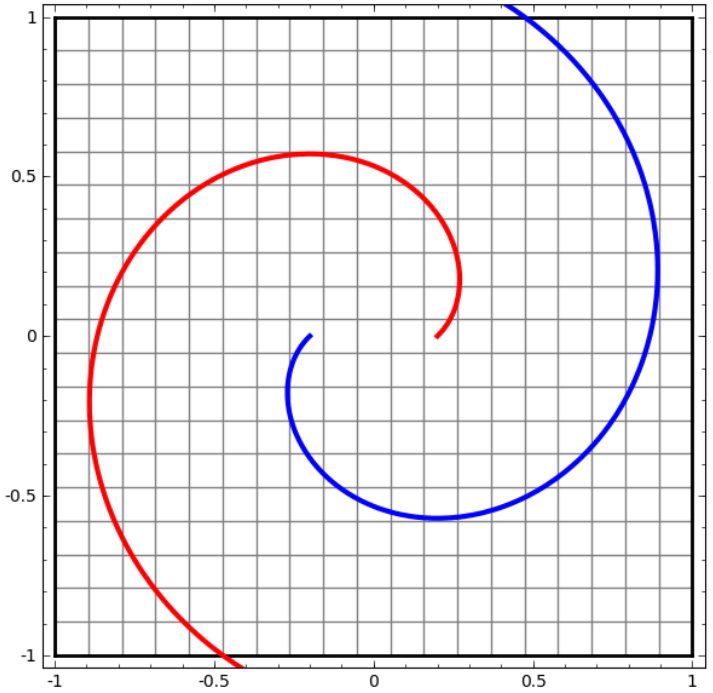
Gradients

Backpropagation can compute this gradient!

And it's a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!

Reverse Mode Differentiation

- Application of the chain-rule from (vector) calculus
- Can view ANNs at level of processing elements (PEs)—neuronal graph
  - Follow dot-arrow diagram to get partial derivative scalars
  - Limited flexibility, but simple to understand
- Can view this at lowest level—computation graph
  - Follow graph of operators & get partial derivatives using sub-rules (sum rule, product rule, etc.)
  - Highly flexible
  - Tools that do this:
    - Theano: http://deeplearning.net/software/theano/
    - TensorFlow: https://www.tensorflow.org/



Deep calculus!

Approaches to Differentiation

Given $f : \mathbb{R}^A \rightarrow \mathbb{R}^B$, $f(\mathbf{x})$

Compute $\frac{\partial f(\mathbf{x})_i}{\partial x_j} \forall i, j$

1. Finite Difference Method
   - Pro: Great for testing implementations of backpropagation
   - Con: Slow for high dimensional inputs / outputs
   - Required: Ability to call the function f(**x**) on any input **x**
2. Symbolic Differentiation
   - Note: The method you learned in high-school
   - Note: Used by Mathematica / Wolfram Alpha / Maple
   - Pro: Yields easily interpretable derivatives
   - Con: Leads to exponential computation time if not carefully implemented
   - Required: Mathematical expression that defines f(**x**)
3. Automatic Differentiation - Reverse Mode
   - Note: Called *Backpropagation* when applied to Neural Nets
   - Pro: Computes partial derivatives of one output $f(\mathbf{x})_i$ with respect to all inputs x_j in time proportional to computation of f(**x**)
   - Con: Slow for high dimensional outputs (e.g. vector-valued functions)
   - Required: Algorithm for computing f(**x**)
4. Automatic Differentiation - Forward Mode
   - Note: Easy to implement. Uses dual numbers.
   - Pro: Computes partial derivatives of all outputs $f(\mathbf{x})_i$ with respect to one input x_j in time proportional to computation of f(**x**)
   - Con: Slow for high dimensional inputs (e.g. vector-valued **x**)
   - Required: Algorithm for computing f(**x**)

The Finite Difference Method

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

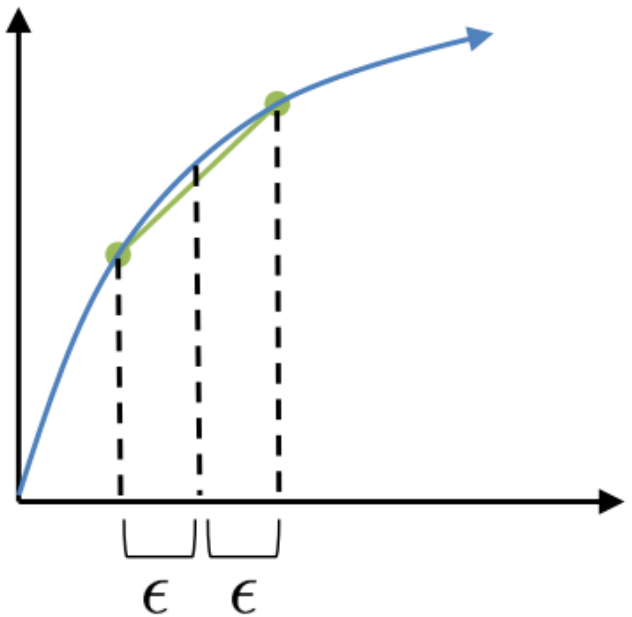
The *centered* finite difference approximation is:

$$\frac{\partial}{\partial \theta_i} J(\boldsymbol{\theta}) \approx \frac{(J(\boldsymbol{\theta} + \epsilon \cdot \boldsymbol{d}_i) - J(\boldsymbol{\theta} - \epsilon \cdot \boldsymbol{d}_i))}{2\epsilon}$$

where $\boldsymbol{d}_i$ is a 1-hot vector consisting of all zeros except for the ith entry of $\boldsymbol{d}_i$, which has value 1.

Notes:

- Suffers from issues of floating point precision, in practice
- Typically only appropriate to use on small examples with an appropriately chosen epsilon



Backpropagation of Errors

White board time!

The Vanishing Gradient Problem

- Solving credit assignment problem with back-propagation too difficult
  - Difficult to know how much importance to accord to remote inputs (Bengio et al., 1994)
  - Information passed through a chain of multiplications back through network
    - Any value slightly less than 1 in hadamard product, and derivative signal quickly shrinks to useless values (near zero)
  - Learning long-term dependencies in temporal sequences becomes near impossible
- Complementary problem: Exploding gradients
  - Any value greater than 1 in hadamard, derivative signal increases dramatically (numerical overflow)

Random Parameter Initializations

- Classical approaches
  - Sample from ~*U(-a, a)*, where is *a* small scalar
  - Sample from ~*N(0, a)*, where is *a* small standard deviation
- Fan-in-Fan-out (number inputs, number output)
  - Calibrate by variances of neuronal activities
- Simple distributional schemes
  - Fan-in/Fan-out Uniform
  - Fan-in/Fan-out Gaussian (good for ReLU activations)
- Orthogonal Initialization
  - Use Singular Value Decomposition (SVD) to find initial weights
- Identity Initialization / Constraint (for RNNs)
  - Does not always work unless constraint is enforced
- Or other intelligent methods?
  - Greedy layer-wise pre-training (we will go over this later in the course!)

Why Do We Care How Parameters Are Initialized?

- Initialization affects final performance
  - Will put closer to some spots in function space and farther from others
- Where we end up in function space will often correlate w/ our error performance

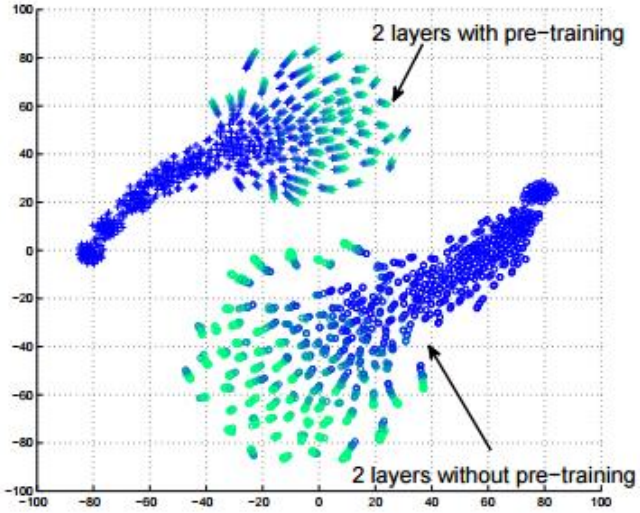


Figure 5: 2D visualizations with tSNE of the functions represented by 50 networks with and 50 networks without pre-training, as supervised training proceeds over MNIST. See Section 6.3 for an explanation. Color from dark blue to cyan and red indicates a progression in training iterations (training is longer without pre-training). The plot shows models with 2 hidden layers but results are similar with other depths.

"Why Does Unsupervised Pre-training Help Deep Learning?", Erhan et al. 2010 http://jmlr.org/papers/volume11/erhan10a/erhan10a.pdf

Or, Just Wait Longer…

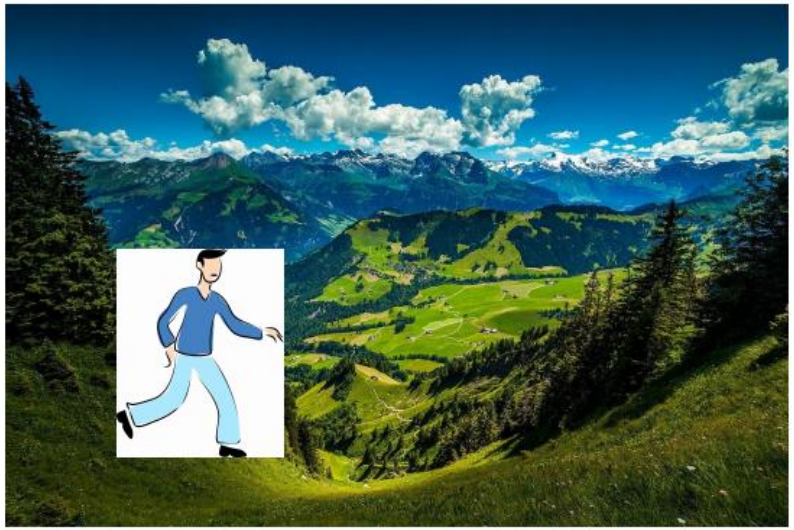
- Even with poor initialization, just wait a really long time….
- Patience + really good hardware
- So one answer = more hardware

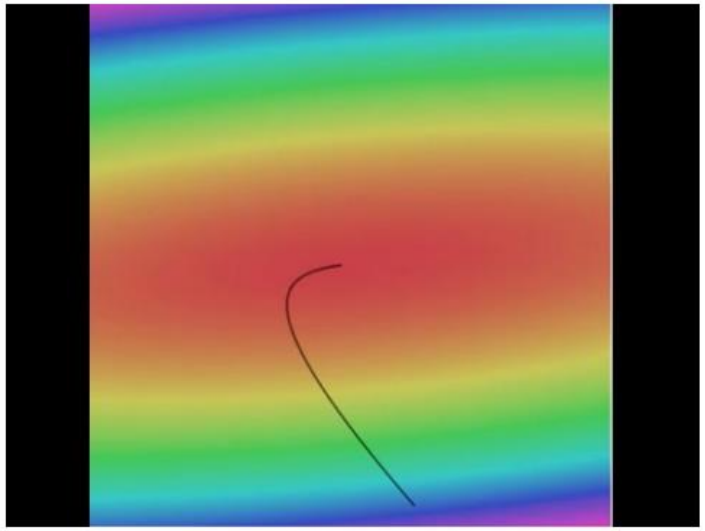
How to make those gradients work for you!

PARAMETER OPTIMIZATION

Optimization Schemes

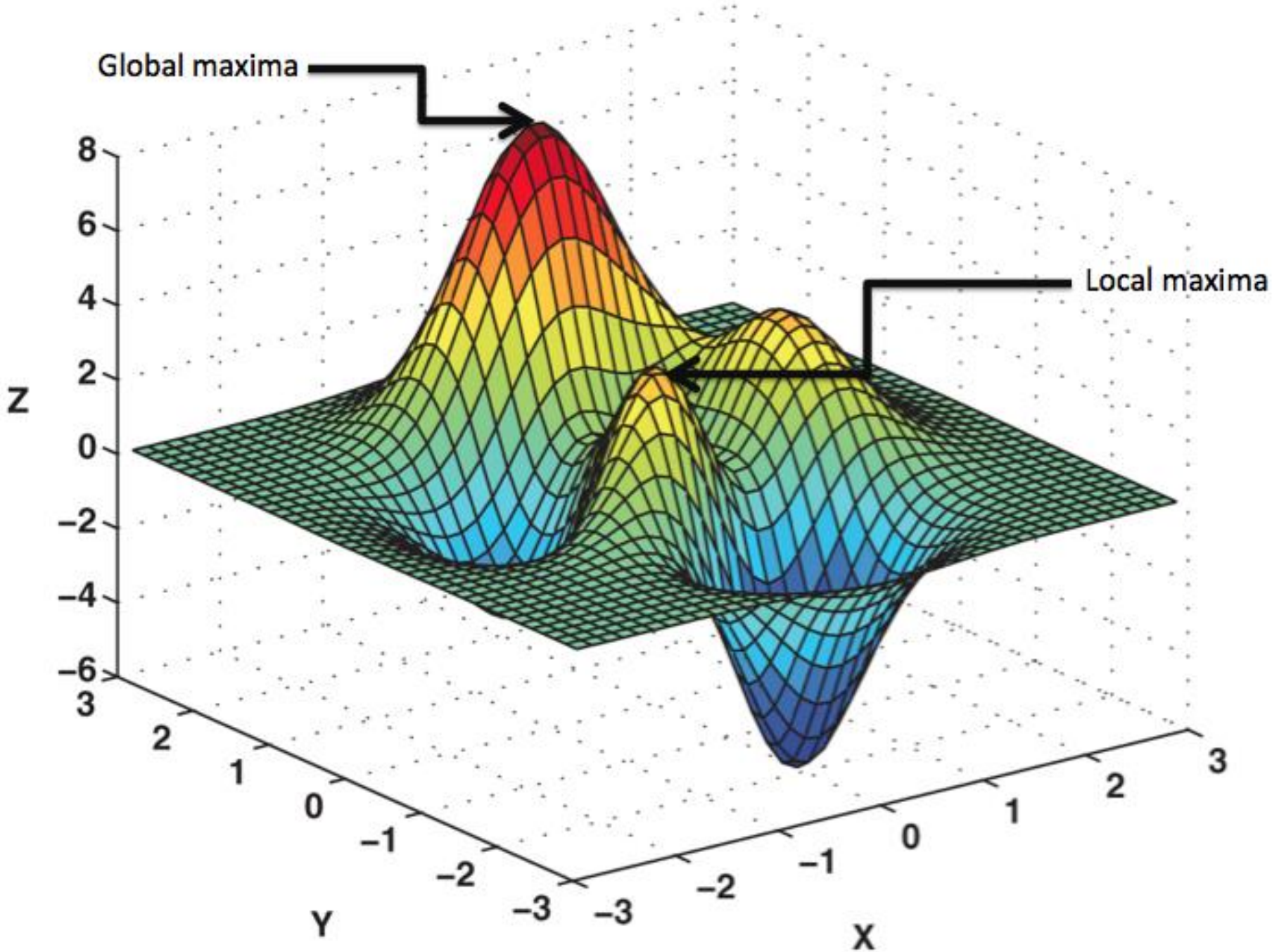
- Steepest (mini-batch) gradient descent
  - Use an estimator (i.e., backprop) to get gradient,
    then update parameters; online case = stochastic gradient descent (SGD)
- Alternative optimizers = shiny toys to make learning even faster






```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```


Global maxima
Local maxima
Z
8
6
4
2
0
−2
−4
−6
3
2
1
0
−1
−2
−3
Y
−3
−2
−1
0
1
2
3
X

Steepest Gradient Descent

- Simplest update rule
- Combine with early stopping
  - Early stopping = tracking loss/error on validation set
  - A simple form of regularization (weights will be smaller)

```
# Vanilla update
x += - learning_rate * dx
```

Simple Momentum

- Maintains rolling average of previous gradients
  - Smooths out descent of minimization algorithm
  - Prevent “bouncing around” on loss/error surface

- Many variants: momentum, Nesterov’s Accelerated Gradient (NAG), etc.

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

Adaptive Learning Rates

- Learning rate per parameter → empirically improves convergence
- **AdaGrad:**
  - Weights that receive high gradients → effective learning rate reduced
  - Weights that receive small/infrequent updates → effective learning rate increased
- **RMSprop:**
  - Reduces AdaGrad's aggressive, monotonically decreasing learning rate
  - Moving average of squared gradients
- **ADAM**: RMSprop + momentum (also corrects for bias towards zero at start of training)
  - Very common in modern optimization of deep architectures

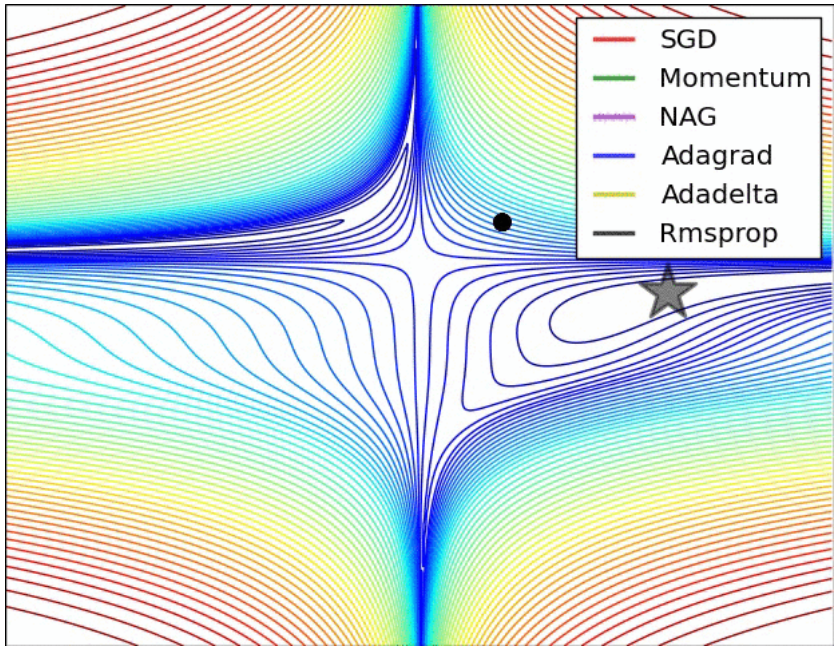
```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

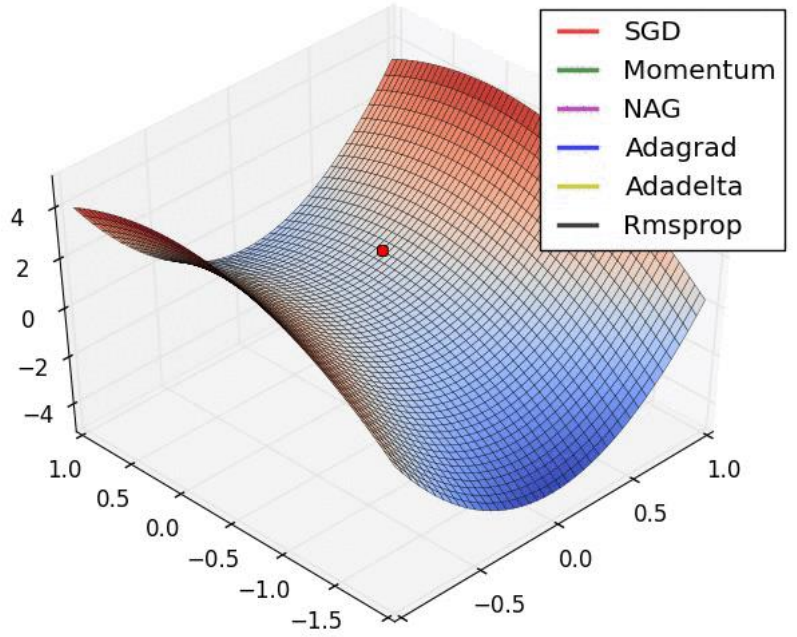
RMSProp

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

AdaGrad

Race of the Optimizers!





http://cs231n.github.io/neural-networks-3/#hyper

Every new idea is really yet another regularizer…

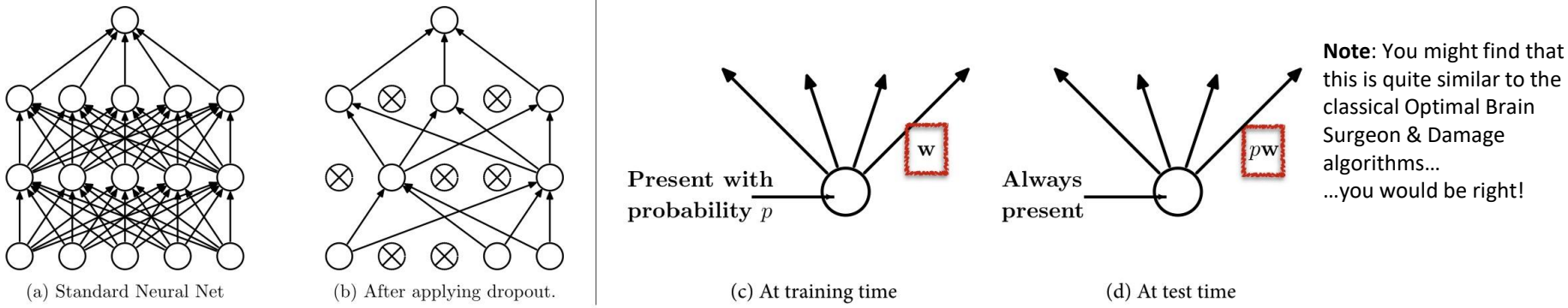
REGULARIZATION OF PARAMETERS

Drop-out & Co-Adaptation

Also some white board!!

- Feature coadaptation: during learning, weights settle into their w/in network
  - Neuronal weights tuned for specific features = some specialization ("neuronal context")
  - Neighboring neurons end up relying on this specialization → could result in a fragile model too specialized to the training data
- Each iteration, omit some units w/ given probability (binary masks)
  - At inference time, simply multiply activations by probability

- In single hidden layer model, equivalent to Bayesian model averaging
- A form of architectural regularization
  - Controls for overfitting
  - Could also drop edges (i.e., Drop-Connect)



(a) Standard Neural Net (b) After applying dropout. (c) At training time (d) At test time

Note: You might find that this is quite similar to the classical Optimal Brain Surgeon & Damage algorithms…
…you would be right!

Batch Normalization & Covariate Shift

Covariate Shift = change in the distribution of a function's domain

- When your inputs change on you, your algorithm can't deal with it
- This happens within layers of a deep network

Solution: standardize internal layers!

- Will need to learn how to scale & shift
- Done on a per-activation basis (mini-batch statistics = mean & variance)
- Test-time: Obtain unbiased estimate of mean & variance on entire training sample

Speeds up learning!

Layer normalization → for recurrent neural networks (RNNs)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

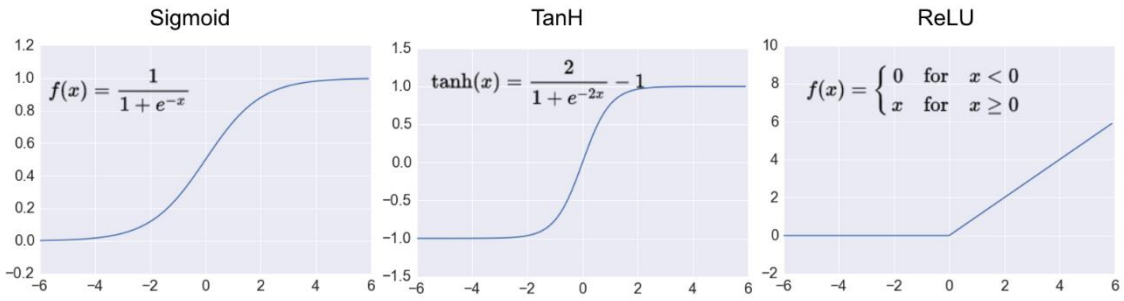
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

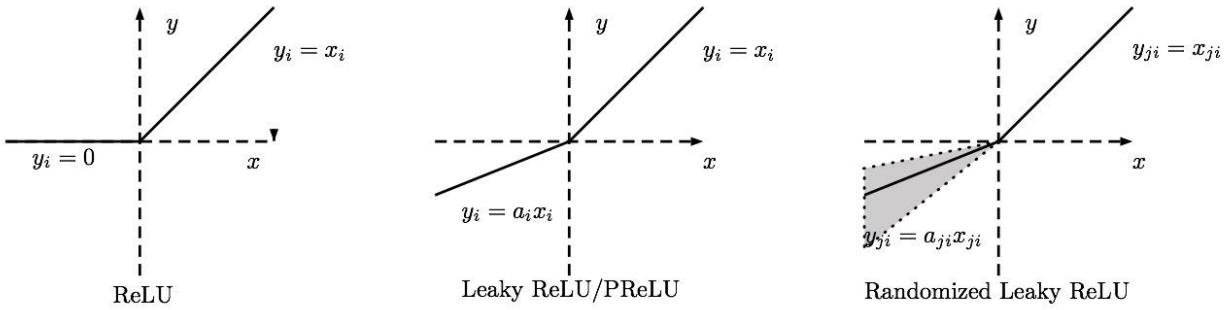
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Non-Standard Activations



Linear Rectified Unit (Relu)

Not smooth / not differentiable everywhere, Benefit: Hard sparsity
Issues: Dead units, explosive weight updates
Parametric Relu (PRelu) & Leaky Relu: Learn the slope of the activation function



Skip Connections

A classical idea

Add short-circuiting to architecture

Can improve gradient flow

Residual Networks:

The value of identity connections

Highway Networks

More complex gating (how much of input passes through, how much is transformed)

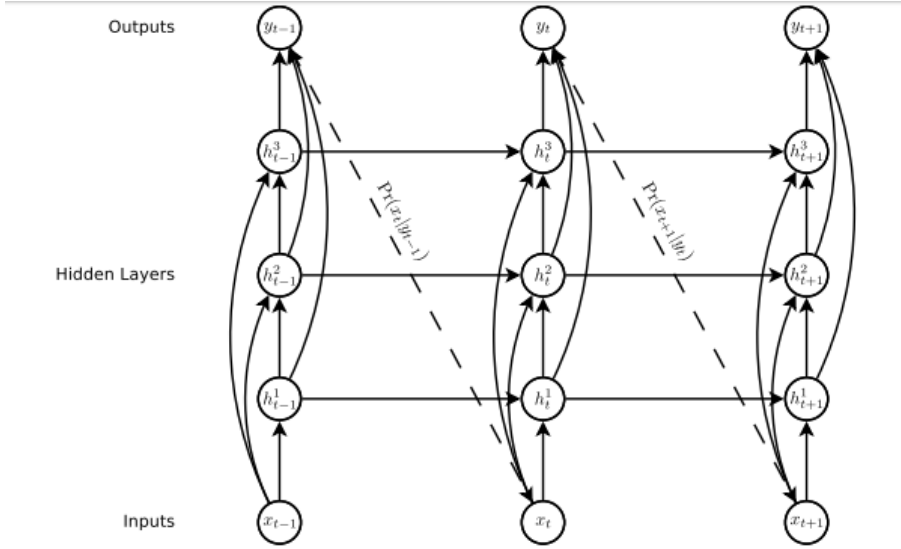


Figure 1: **Deep recurrent neural network prediction architecture.** The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions.

https://arxiv.org/pdf/1308.0850v5.pdf

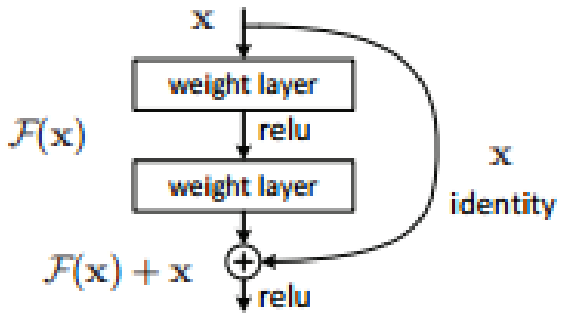


Figure 2. Residual learning: a building block.

https://arxiv.org/abs/1512.03385

On the human-in-the-loop…

TUNING A DEEP ARCHITECTURE

Manual & Exhaustive Search

Manual Search

Fast if you know what you are doing!
Explore a few configurations, based on literature/heuristics
Select lowest validation loss configuration

Deep tuning!

Grid Search

Compose an n-dimensional hypercube, where along each axis is a hyper-parameter
(length determined by max & min values to explore)
Exhaustively calculate loss/error for each configuration (or combination of meta-parameter values) in hypercube
- Choose lowest error/minimal loss configuration as optimal model
- Loss/error is calculated on a held-out validation/development set (or in held-out set in cross-fold validation schemes)

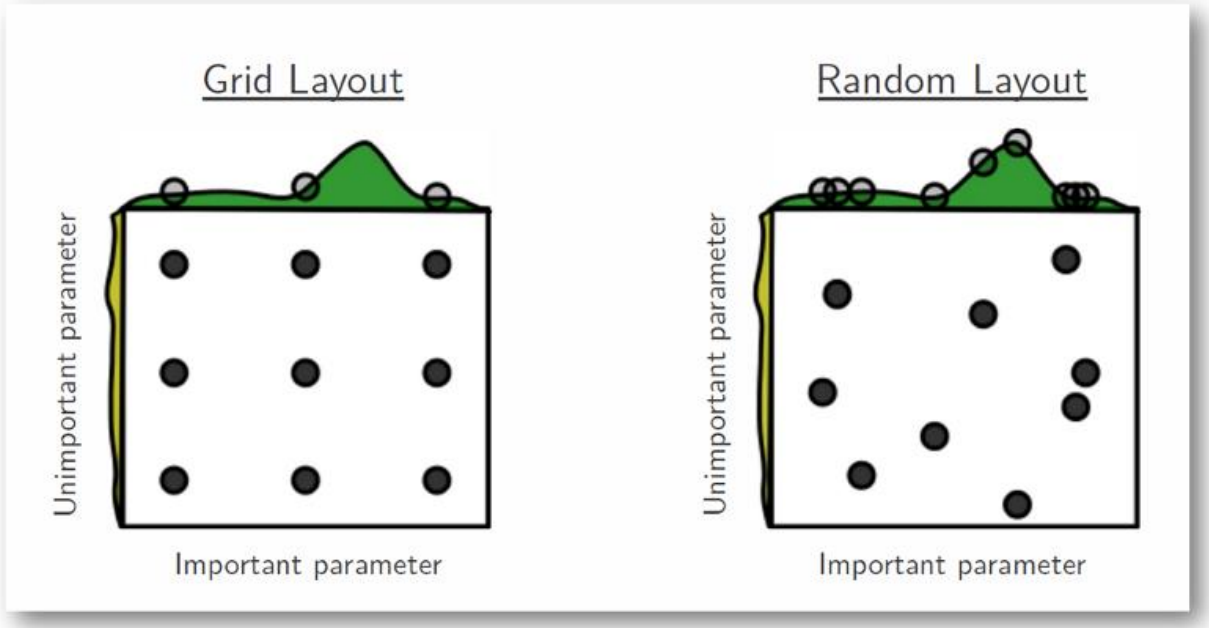
Will ultimately find optimal model (depends on coarseness of grid-search)
- Takes long time!

Random Search

Draw *k* sample configurations from hypercube & calculate validation loss for each (w/o replacement)

- Repeat *T* trials, can use optimal of each trial to inform subsequent trials
- Could “guide” or “target” next set of random samples based on best last found point (a guided stochastic search)

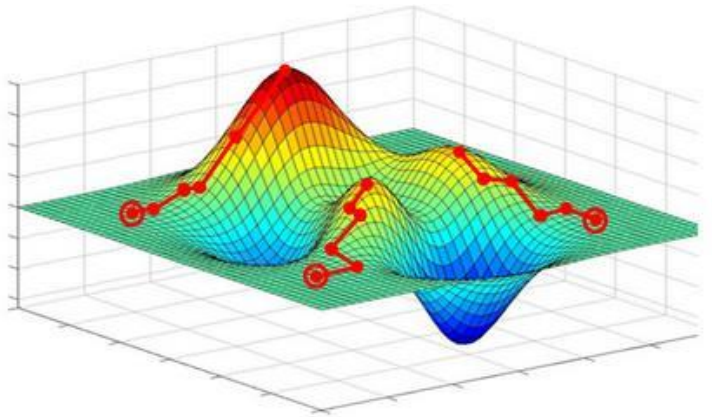
Surprisingly effective (over manual search) & faster than grid search



Bayesian Optimization: Meta Machine Learning

Use machine learning to do your research for you…

- Sequential Model Optimization (*SMO*)
- Gaussian Processes for surface-response modeling
- Gradient-based: Use another ANN
  - How do we tune this higher-level parametric model?
  - Meta-meta-meta-….-machine learning??



High-level idea:

- Build a meta-model (w/ some prior that encodes intuition about hyper-parameter space)
- Draw samples from space (i.e., run few model configurations)
- Update meta-model using these samples
- Meta-model selects next best point to evaluate
  - Balancing criterion, i.e., minimal error & minimal compute time

Bayesian Optimization: Meta Machine Learning

Use machine learning to do your re

- Sequential Model Optimizati
- Gaussian Processes f
- Gradient
  - How
  - paran
  - Meta-

High-level

- Build a me
- encodes int
- Draw sample
- Update meta
- Meta-model s
- g crit
- compute time

Now all we need is for our model to write our papers for us & submit them to the conferences on our behalf… …and then go to class for us too, and life will be good!

Deep Thinking!

It is a matter of posing the problem

What is the low-level representation of your sample?
(i.e., low-level features, inputs, or sensors)
Is there an output we are interested in?

Regression: a real-valued target
Categorization: a discrete target

How much data do you have?

More data is better! (MNIST is 60K)
Only a small sample?
Go Bayesian Neural Networks!

What kind of hardware do you have?

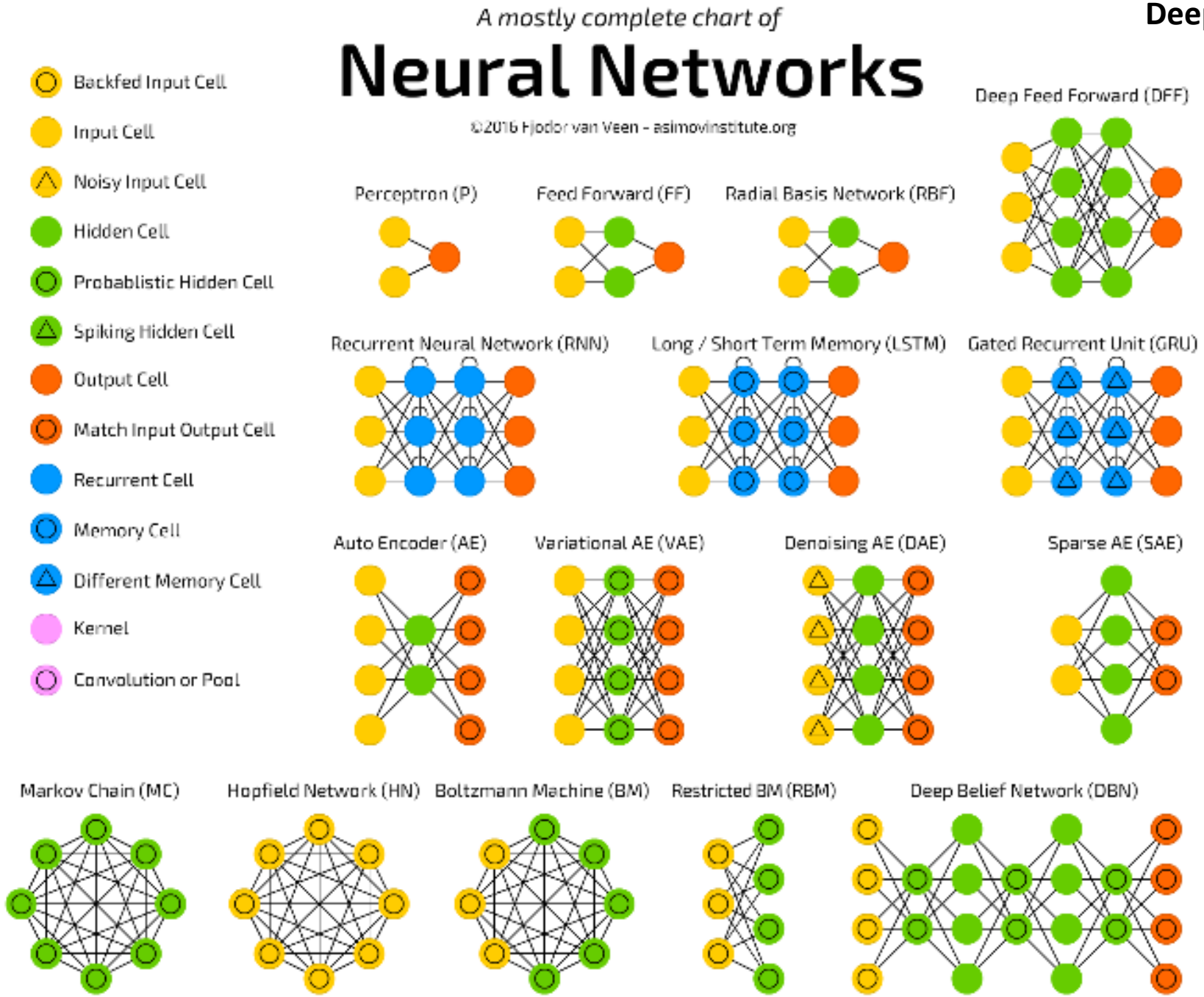
Multi-CPU settings
GPUs
Specialized hardware?
FPGAs, TPUs?

Deep digit recognition!

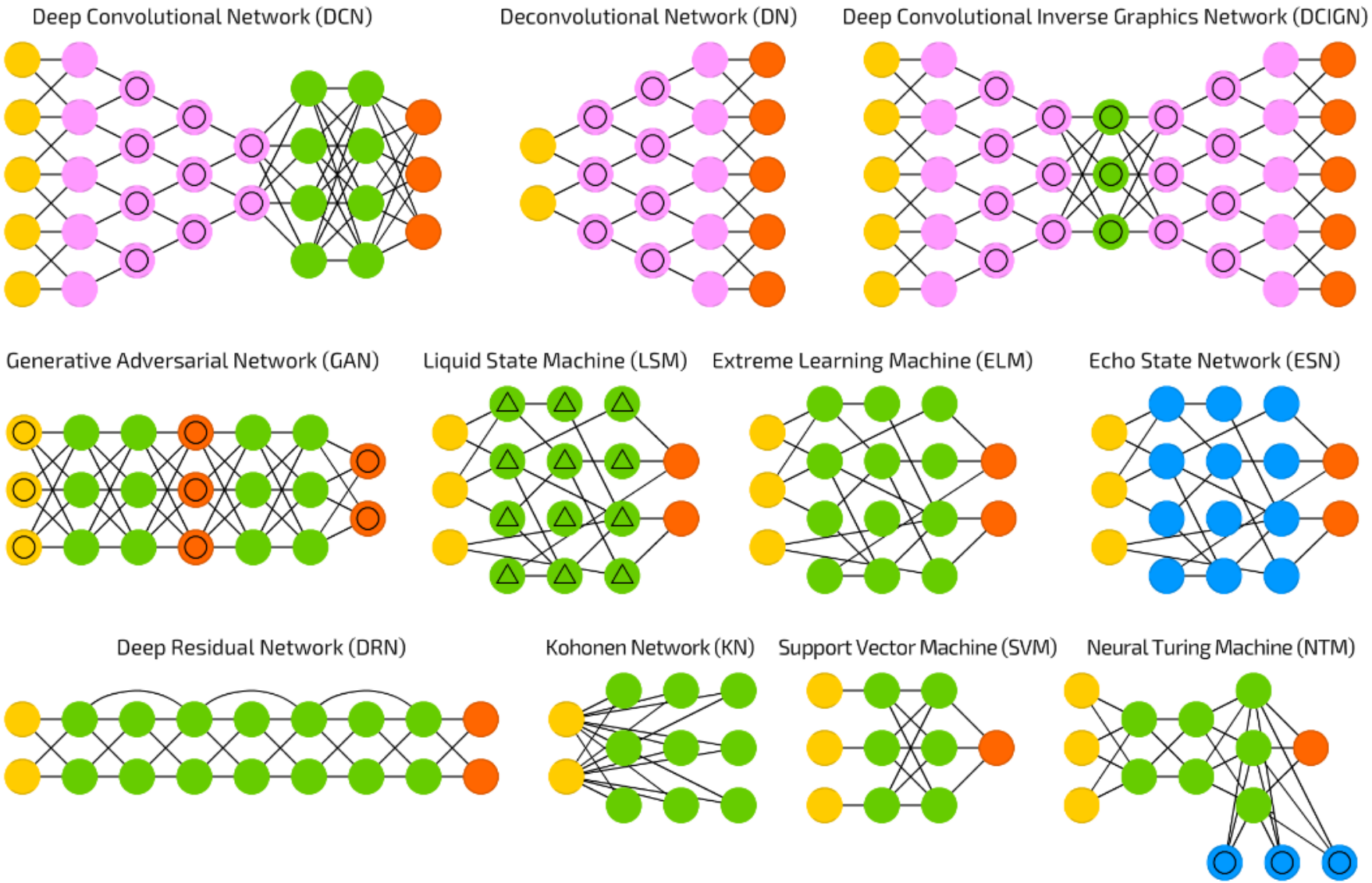
On the plethora of model structures…

THE SPACE OF NEURAL ARCHITECTURES

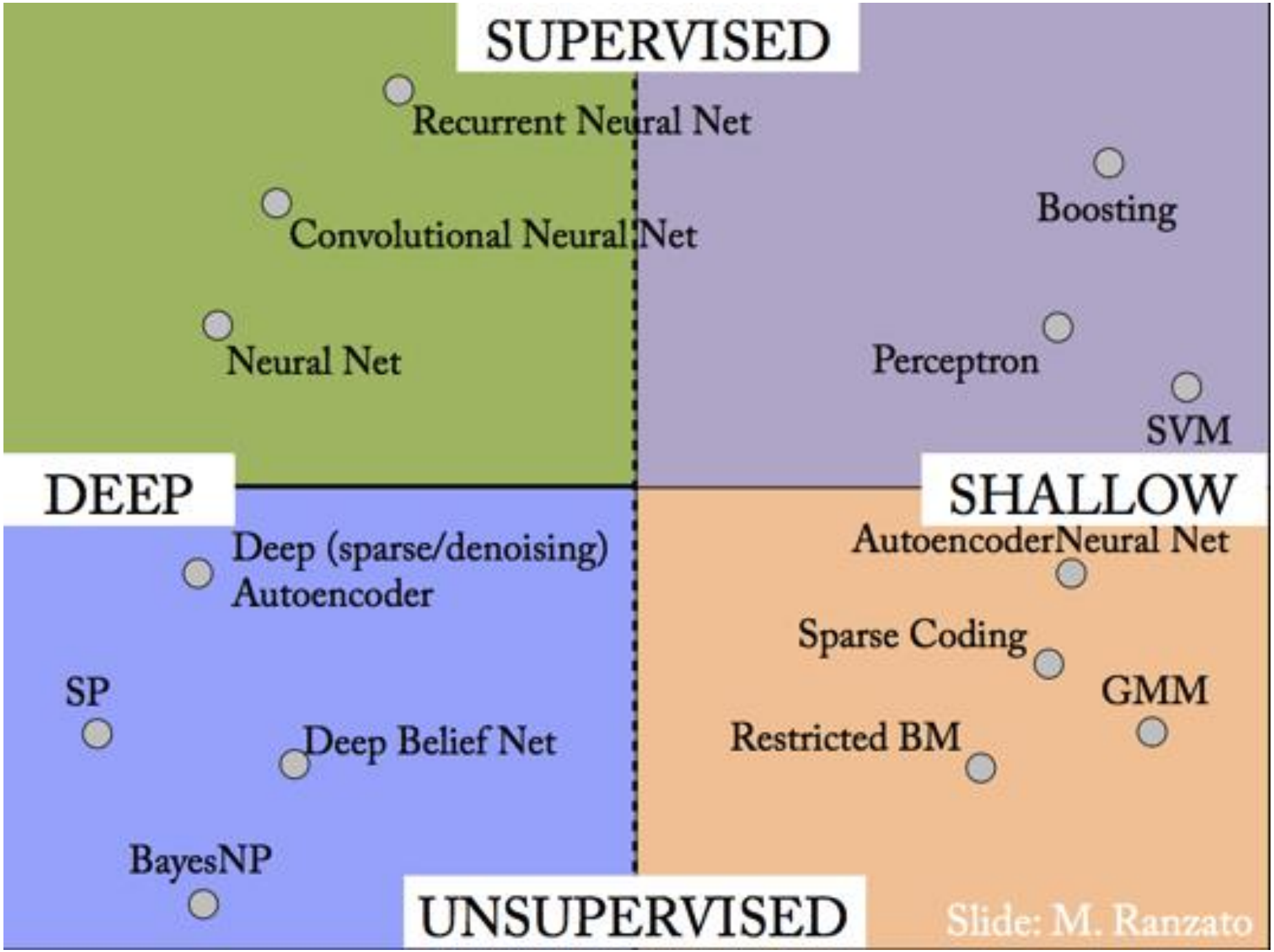
Deep zoo!



http://www.asimovinstitute.org/neural-network-zoo/



http://www.asimovinstitute.org/neural-network-zoo/


SUPERVISED
Recurrent Neural Net
Convolutional Neural Net
Neural Net
Boosting
Perceptron
SVM
DEEP
SHALLOW
Deep (sparse/denoising) Autoencoder
SP
Deep Belief Net
BayesNP
AutoencoderNeural Net
Sparse Coding
GMM
Restricted BM
UNSUPERVISED
Slide: M. Ranzato

Read for This Thursday

- Read Chapter 15 of Deep Learning textbook
  - https://www.deeplearningbook.org/

QUESTIONS?