



On Deep Learning

Alexander G. Ororbia II
Introduction to Machine Learning
CSCI-736
1/25/2025

Companion reading:
Chapter 6-8 of Deep Learning textbook

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 $\sim U(-a, a)$, where a is a small scalar
 - Sample from $\sim N(0, a)$, where a 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

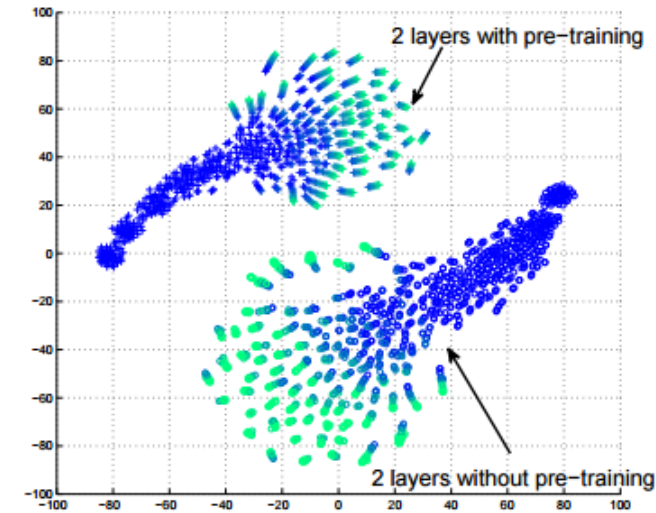


Figure 5: 2D visualizations with tSNE of the functions represented by 50 networks with and 50 networks without pre-training, as supervised training proceeds over MNIST. See Section 6.3 for an explanation. Color from dark blue to cyan and red indicates a progression in training iterations (training is 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, really long time....
- Patience + really good hardware is “all you need”
- So, one answer = more hardware

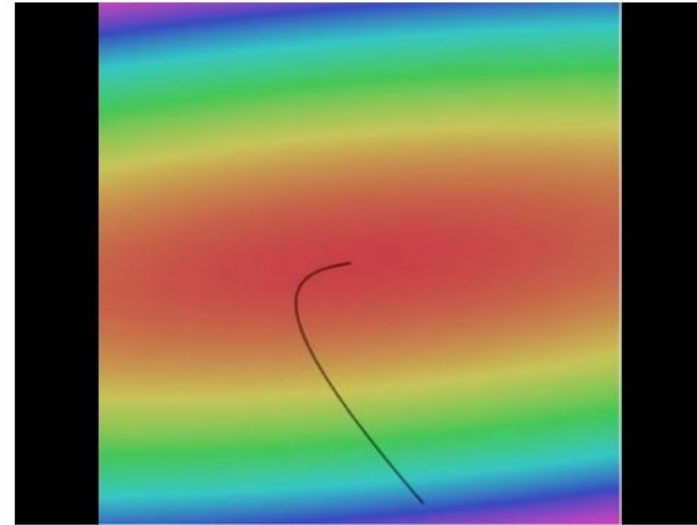
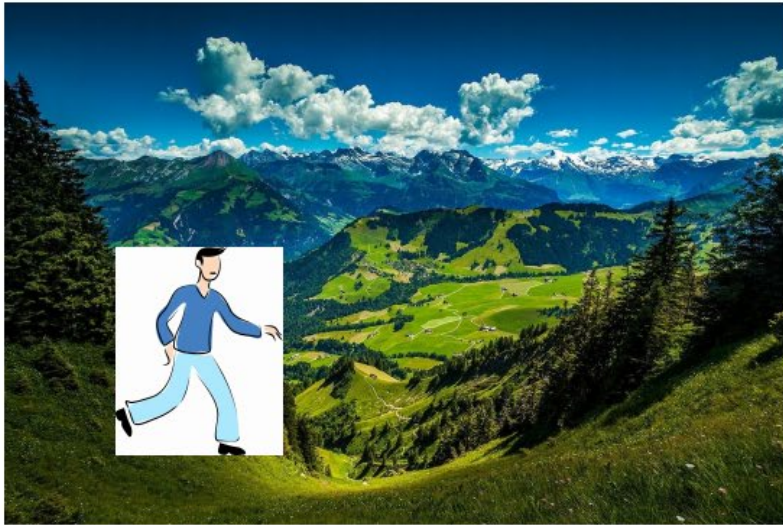


How to make those gradients work for you!

PARAMETER OPTIMIZATION

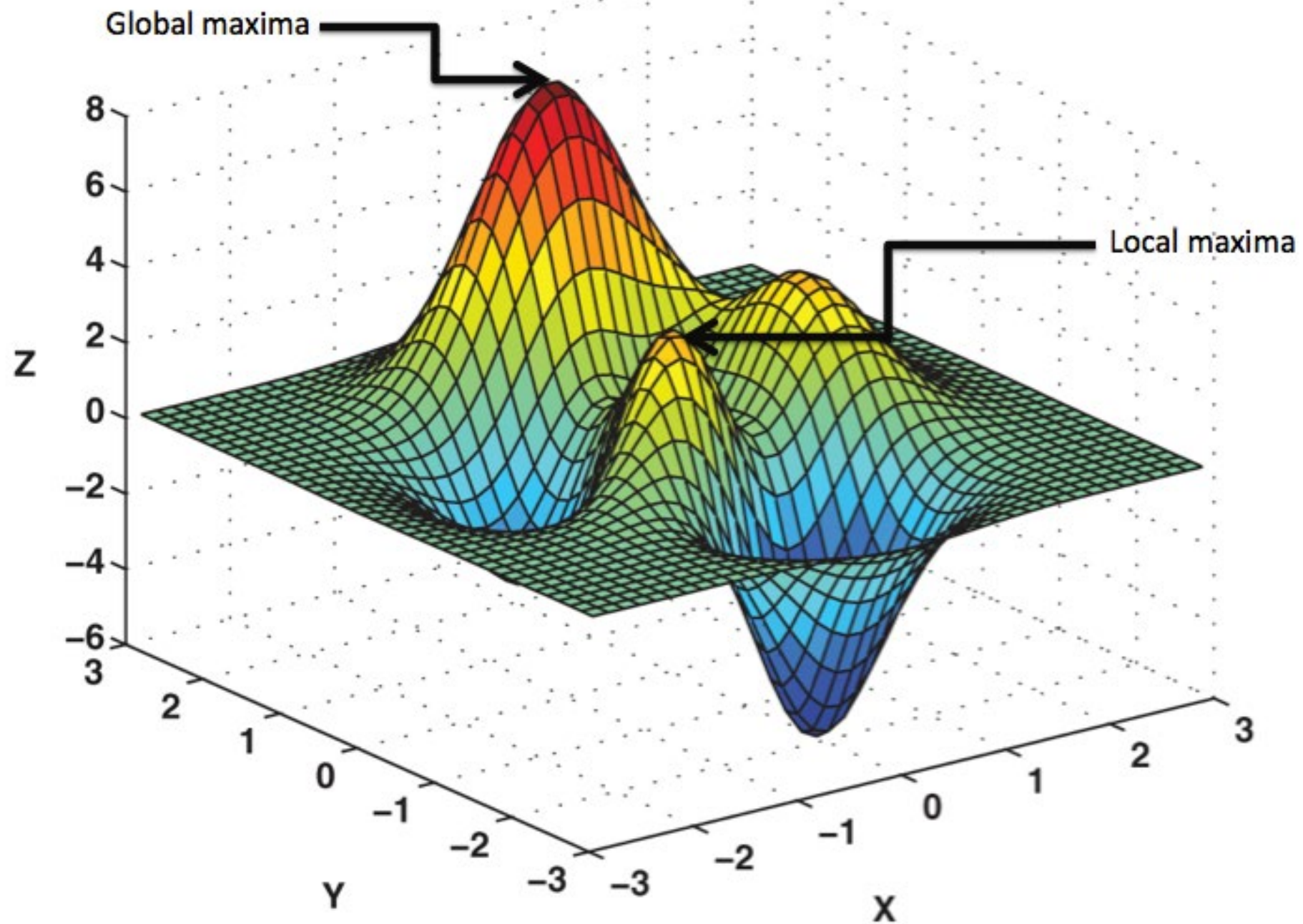
Optimization Schemes

- 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

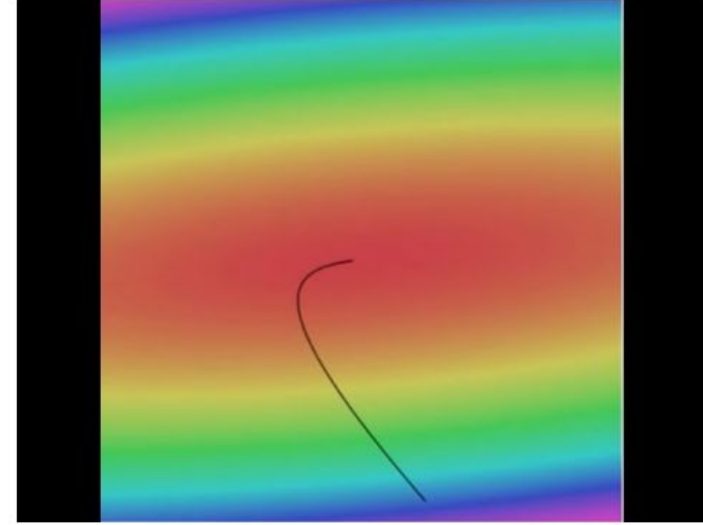


```
# Vanilla Gradient Descent

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

Optimization



[Landscape image](#) is [CC0 1.0](#) public domain
[Walking man image](#) is [CC0 1.0](#) public domain

```
# Vanilla Gradient Descent

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

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

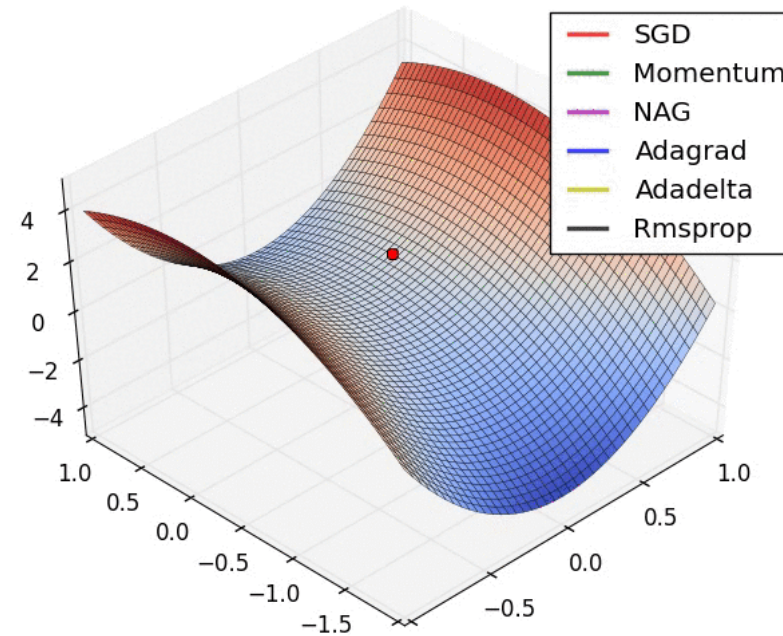
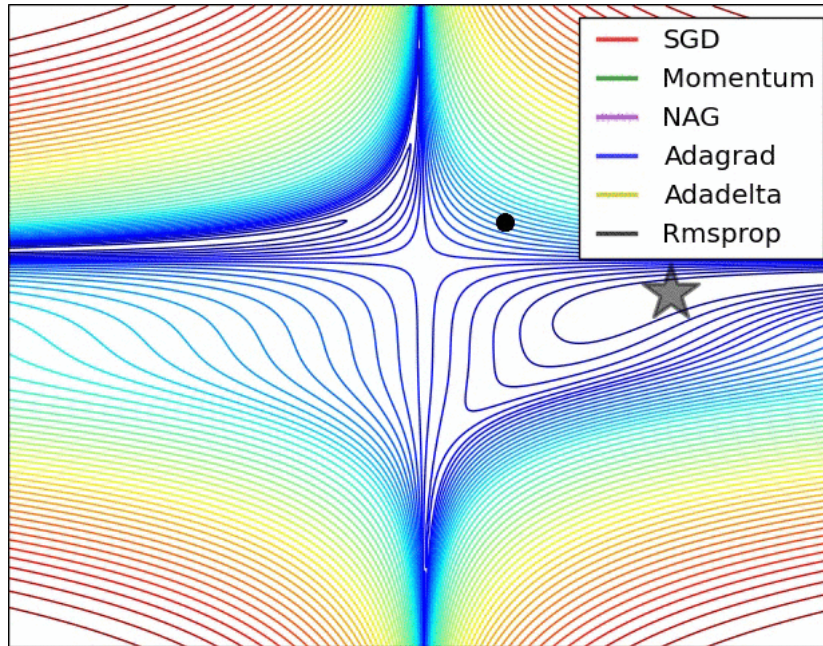
```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

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

Regularization: L2 Penalty

$$C = -\frac{1}{n} \sum_{x_j} \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right] + \frac{\lambda}{2n} \sum_w w^2$$

The first term is just the usual expression for the cross-entropy. But we've added a second term, namely the sum of the squares of all the weights in the network. This is scaled by a factor $\lambda/2n$, where $\lambda > 0$ is known as the *regularization parameter*, and n is, as usual, the size of our training set

$$\begin{aligned} w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \\ &= \left(1 - \frac{\eta \lambda}{n} \right) w - \eta \frac{\partial C_0}{\partial w} \end{aligned}$$

Regularization: L1 Penalty

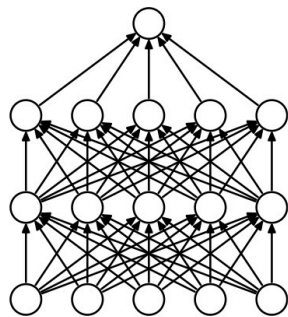
$$C = C_0 + \frac{\lambda}{n} \sum_w |w|.$$

Intuitively, this is similar to L2 regularization, penalizing large weights, and tending to make the network prefer small weights

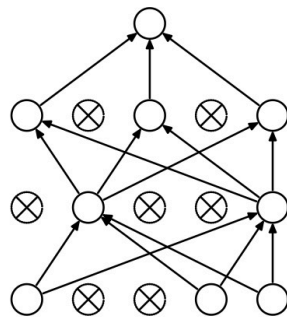
$$w \rightarrow w' = w - \frac{\eta\lambda}{n} \text{sgn}(w) - \eta \frac{\partial C_0}{\partial w},$$

Drop-Out and Co-Adaptation

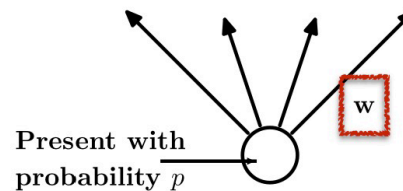
- Feature coadaptation: during learning, weights settle into their w/in network
 - Neuronal weights tuned for specific features = some specialization (“neuronal context”)
 - Neighboring neurons end up relying on this specialization → could result in a fragile model too specialized to the training data
- 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)



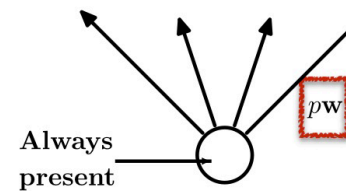
(a) Standard Neural Net



(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: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

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

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

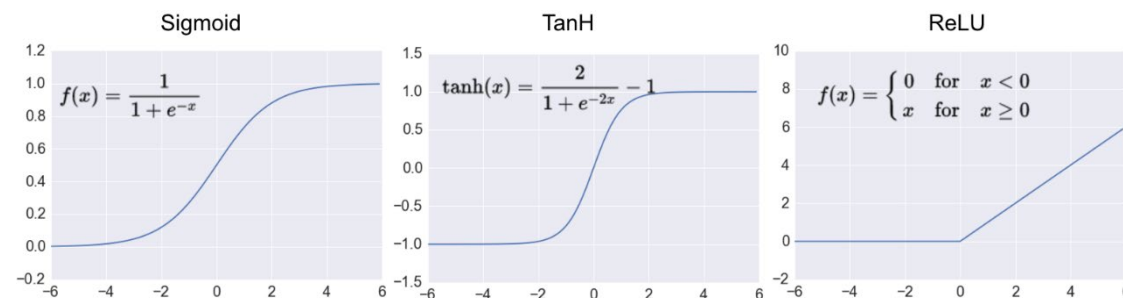
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

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

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

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

Non-Standard Activations

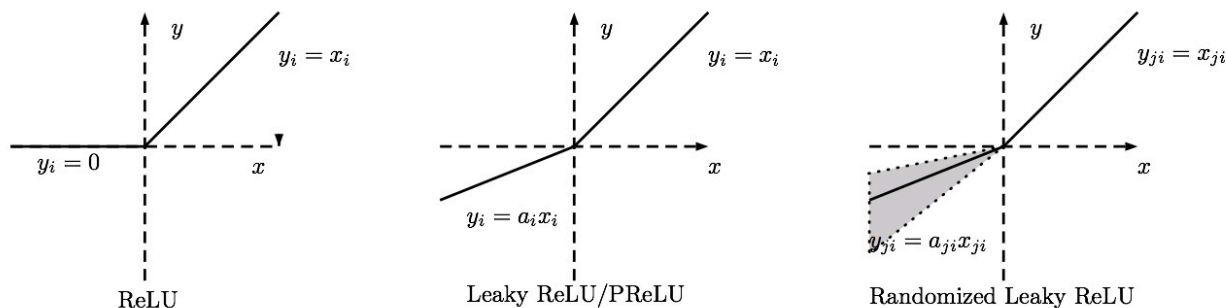


Linear Rectified Unit (Relu)

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)

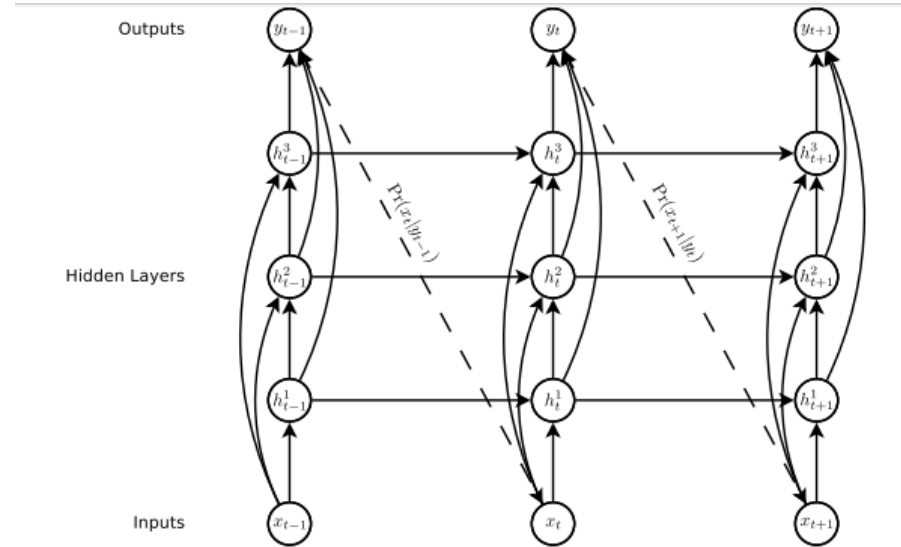


Figure 1: **Deep recurrent neural network prediction architecture.** The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions.

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

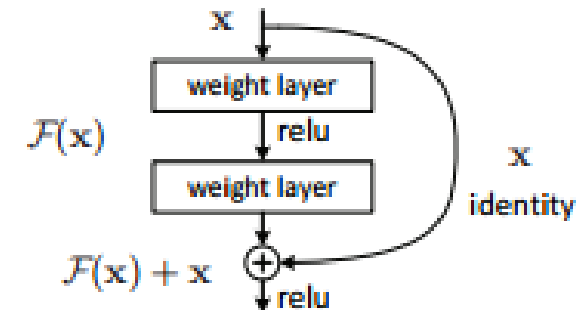


Figure 2. **Residual learning: a building block.**

<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

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!



Deep tuning!

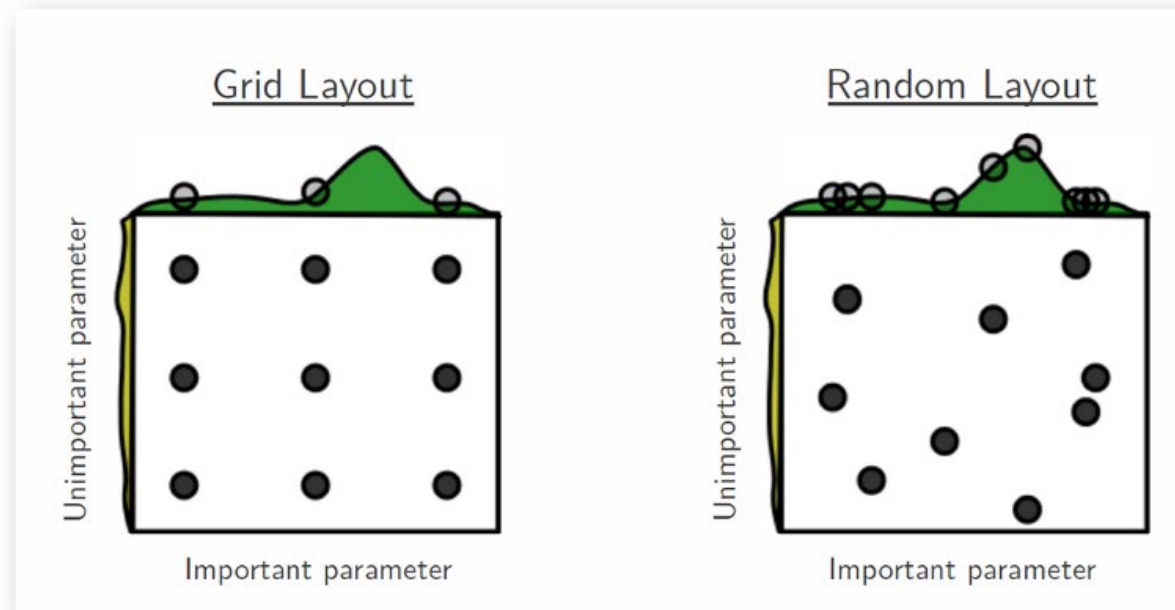
Random Search

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:

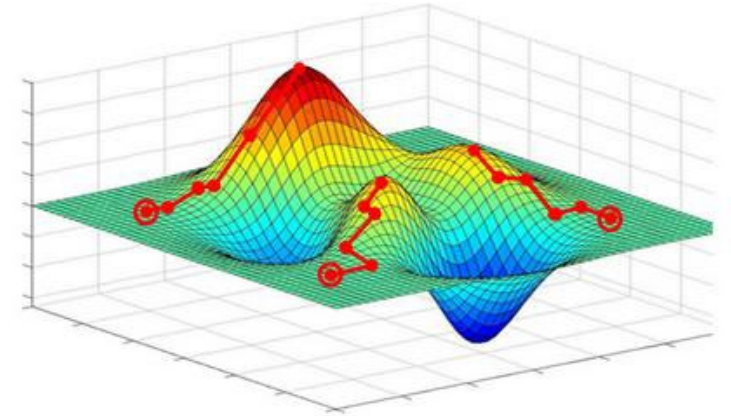
Build a meta-model (w/ some prior that
encodes intuition about hyper-parameter space)

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



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

A mostly complete chart of Neural Networks

Deep zoo!

©2016 Hjoor van Veen - asimovinstitute.org

-  Backfed Input Cell
-  Input Cell
-  Noisy Input Cell
-  Hidden Cell
-  Probabilistic Hidden Cell
-  Spiking Hidden Cell
-  Output Cell
-  Match Input Output Cell
-  Recurrent Cell
-  Memory Cell
-  Different Memory Cell
-  Kernel
-  Convolution or Pool

Perceptron (P)



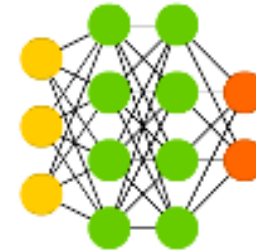
Feed Forward (FF)



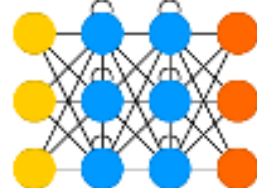
Radial Basis Network (RBF)



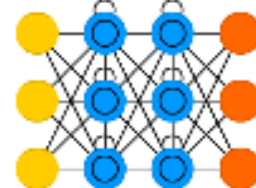
Deep Feed Forward (DFF)



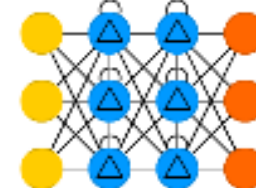
Recurrent Neural Network (RNN)



Long / Short Term Memory (LSTM)



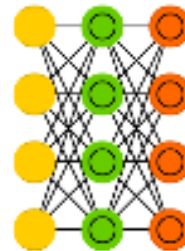
Gated Recurrent Unit (GRU)



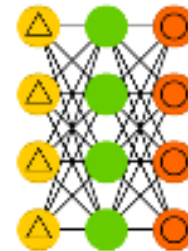
Auto Encoder (AE)



Variational AE (VAE)



Denoising AE (DAE)



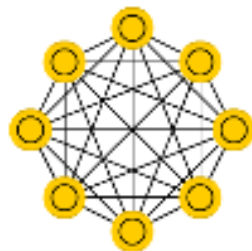
Sparse AE (SAE)



Markov Chain (MC)



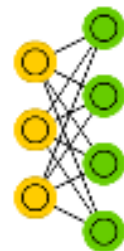
Hopfield Network (HN)



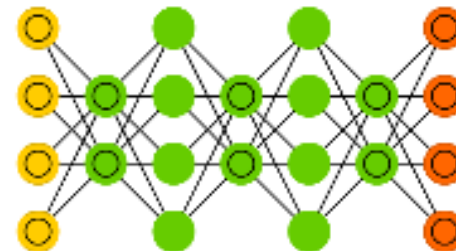
Boltzmann Machine (BM)



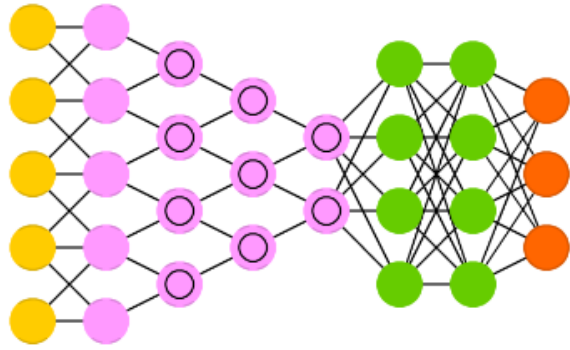
Restricted BM (RBM)



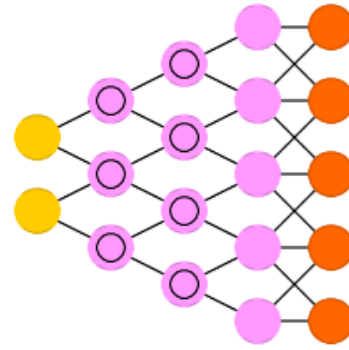
Deep Belief Network (DBN)



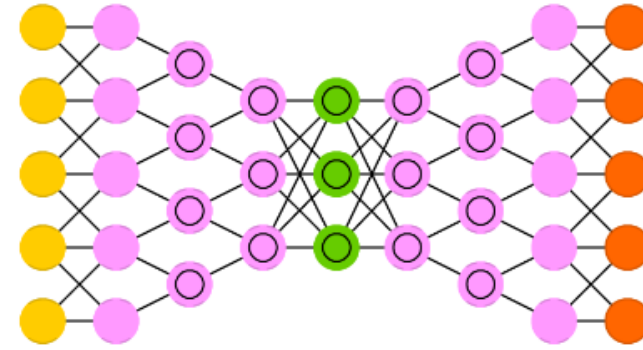
Deep Convolutional Network (DCN)



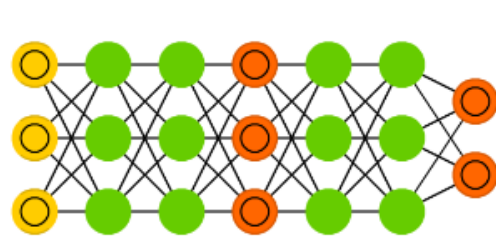
Deconvolutional Network (DN)



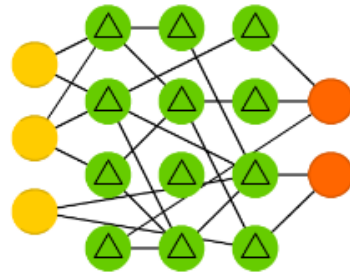
Deep Convolutional Inverse Graphics Network (DCIGN)



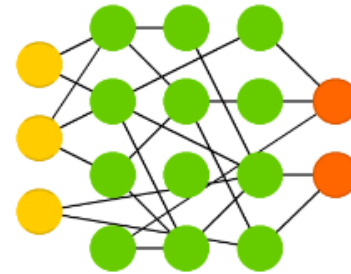
Generative Adversarial Network (GAN)



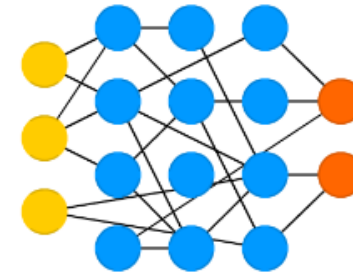
Liquid State Machine (LSM)



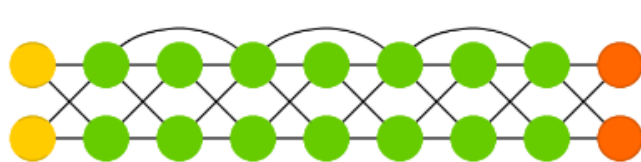
Extreme Learning Machine (ELM)



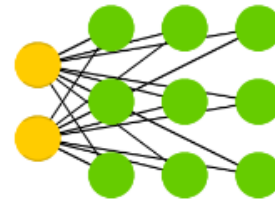
Echo State Network (ESN)



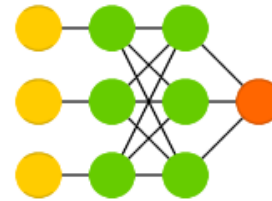
Deep Residual Network (DRN)



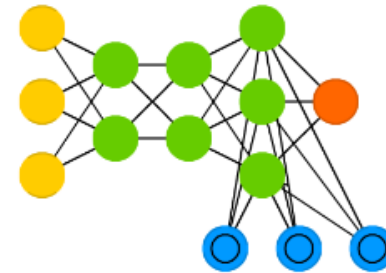
Kohonen Network (KN)

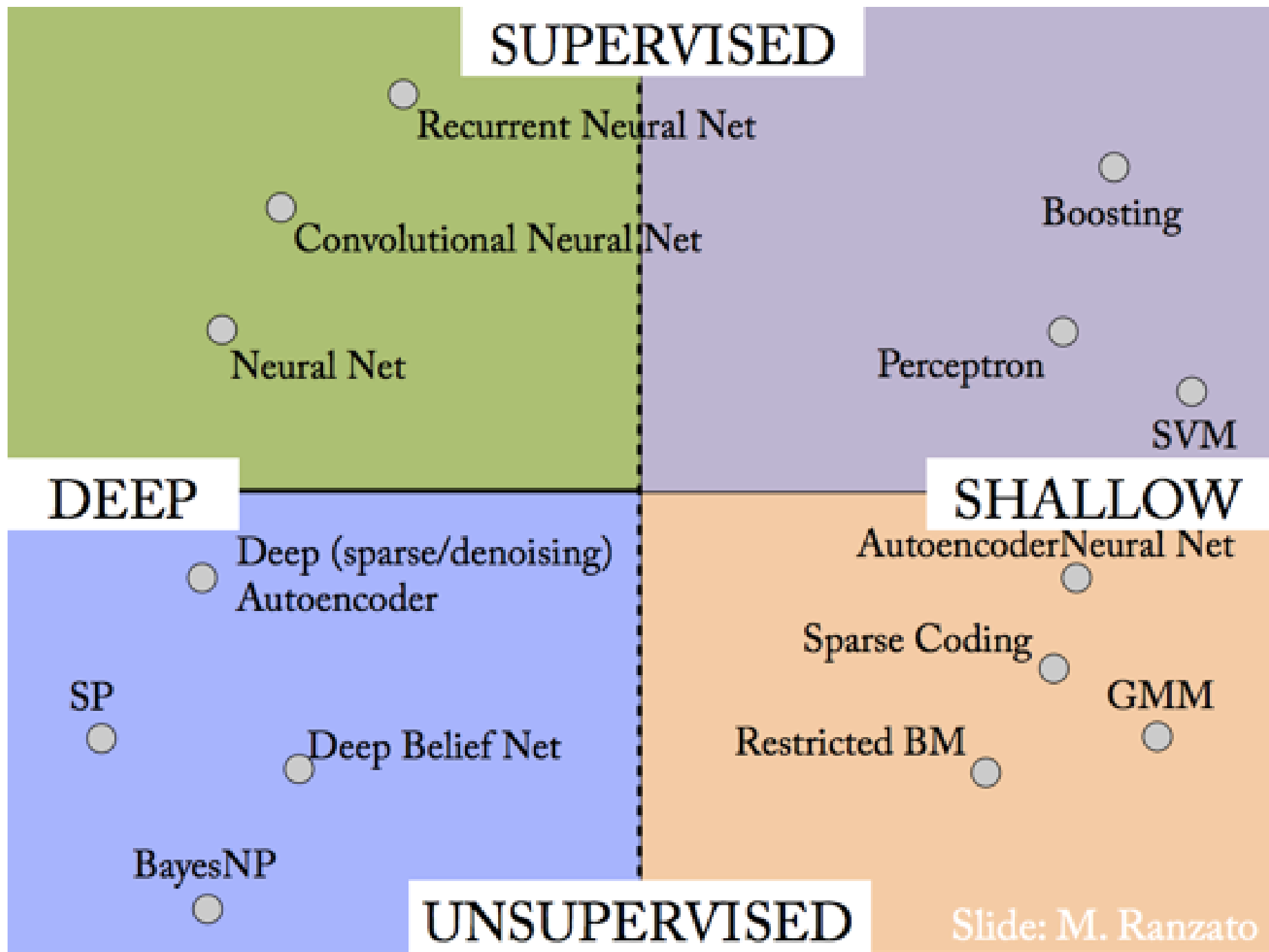


Support Vector Machine (SVM)



Neural Turing Machine (NTM)

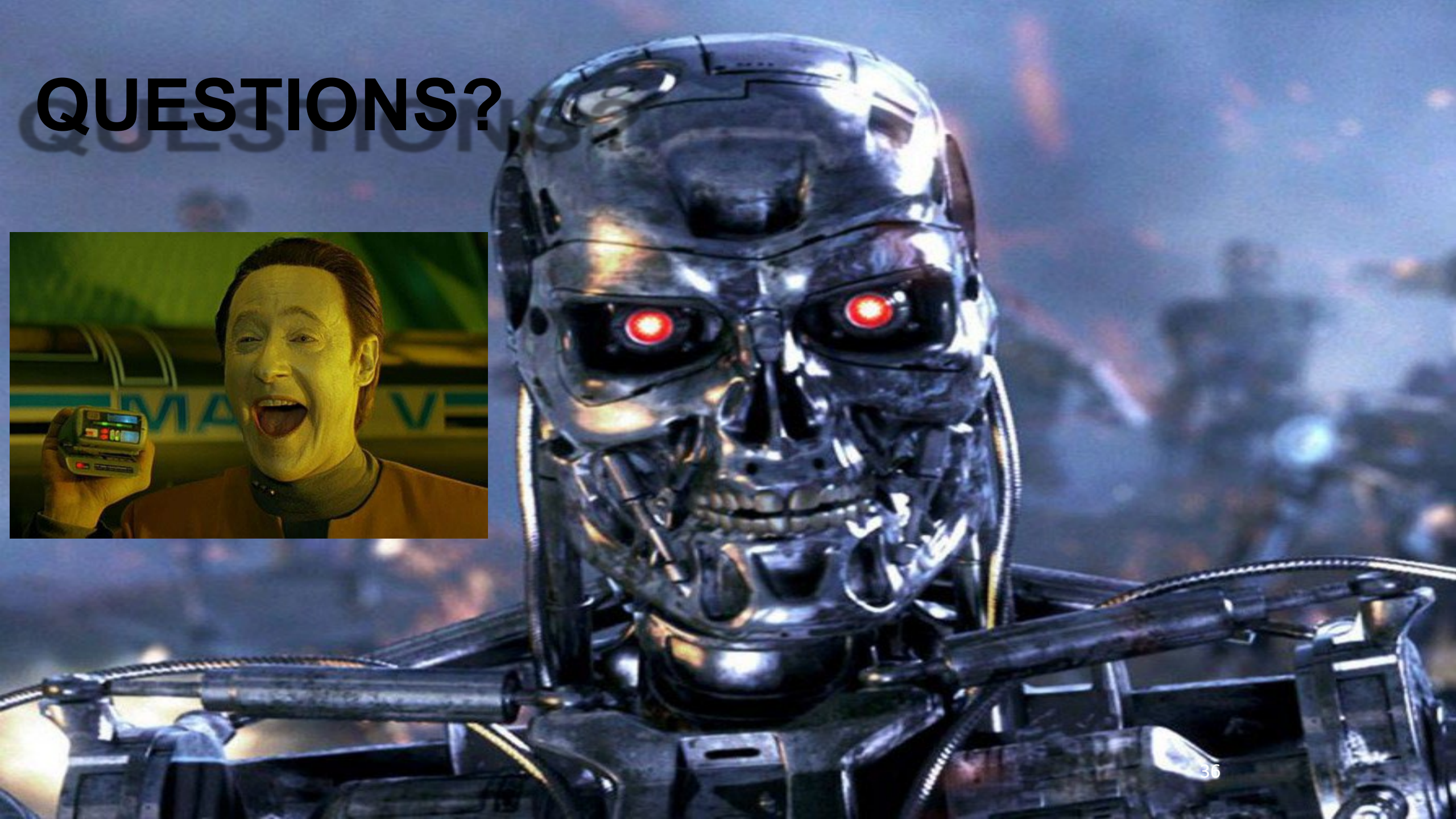




Read for This Thursday / Next Week

- **Read** Chapter 15 of Deep Learning textbook
 - <https://www.deeplearningbook.org/>
 - Mini homework might be assigned asking questions about this chapter
- Two papers will be assigned for next week (chosen by the class teams) for you to read
 - These will be posted on MyCourses once they have been determined by the teams
 - We will reach out to the teams selected for next week (please get us your team choices by tomorrow noon and no later!)

QUESTIONS?



Extra Content