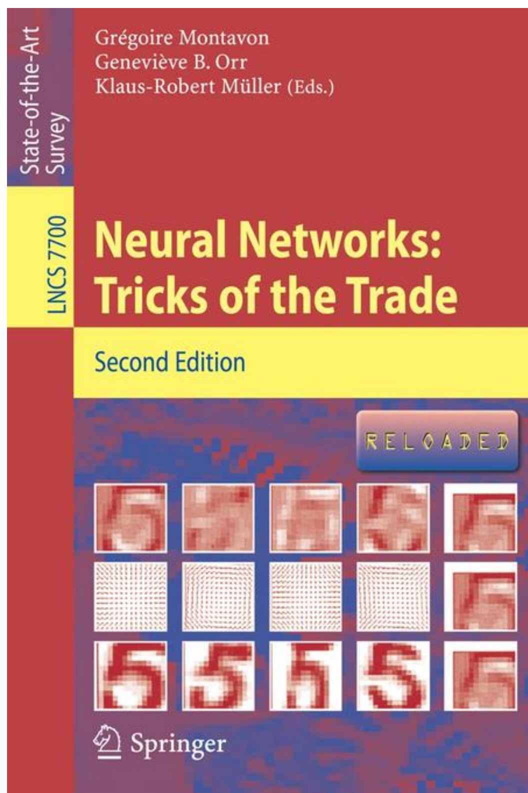
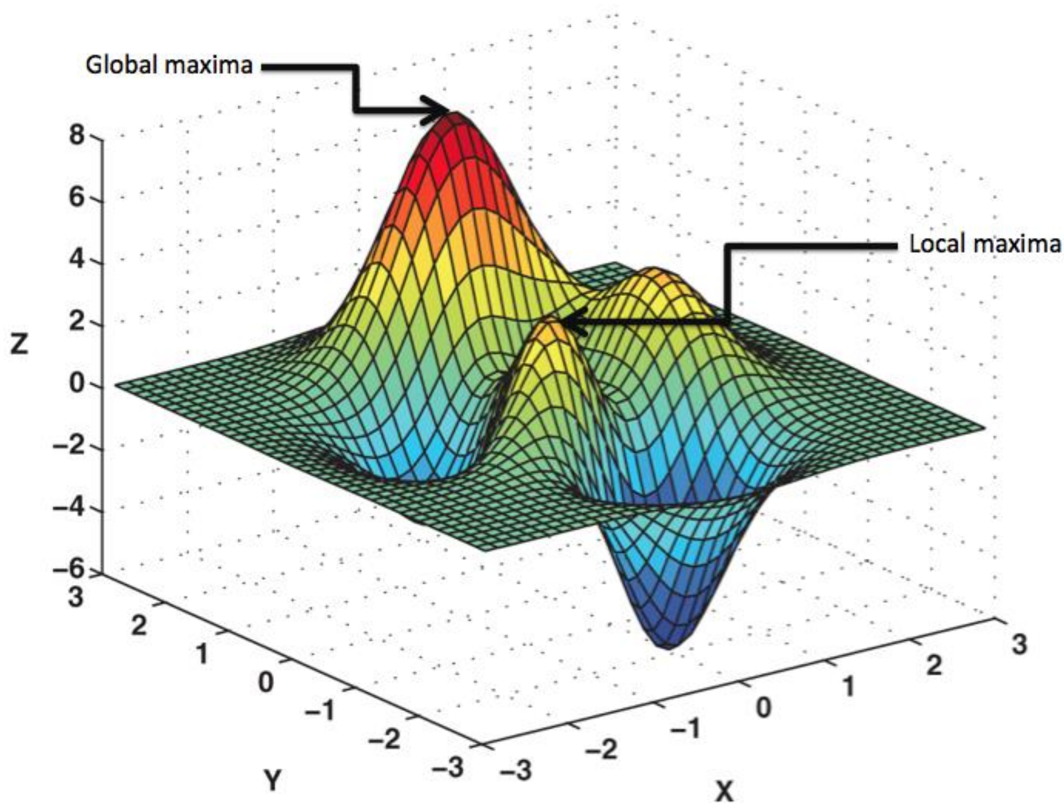
Artificial Neural Networks: Some More "Tricks of the Trade"

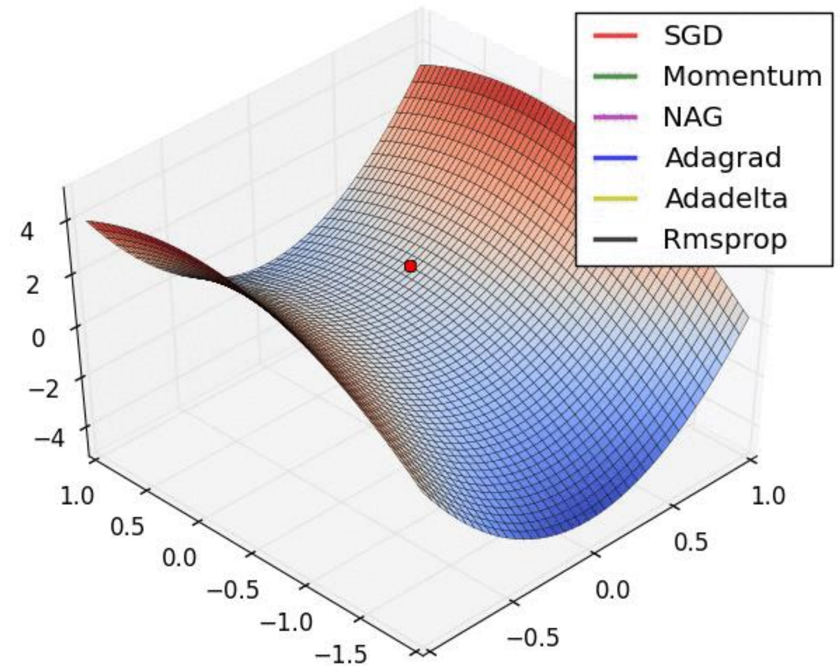


Alexander G. Ororbia II
Introduction to Machine Learning
CSCI-335
4/24/2026

Optimization Schemes

- *Alternative optimizers* = shiny toys to make learning even faster, often play off idea we can dynamically set learning rate





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:** **(*funny note*: a set of lecture slides are cited for this one in research)**
  - 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 popular in modern-day 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

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:** **(*funny note*: a set of lecture slides are ci... e in research)**
  - Reduces AdaGrad's aggressive, mon... arning rate
  - Moving average of squared...
- **ADAM:** RMSprop ... rds zero at start of training
  - Very popular in ... eep architectures

There are still yet new ones coming out these days, e.g., AdamW, etc.

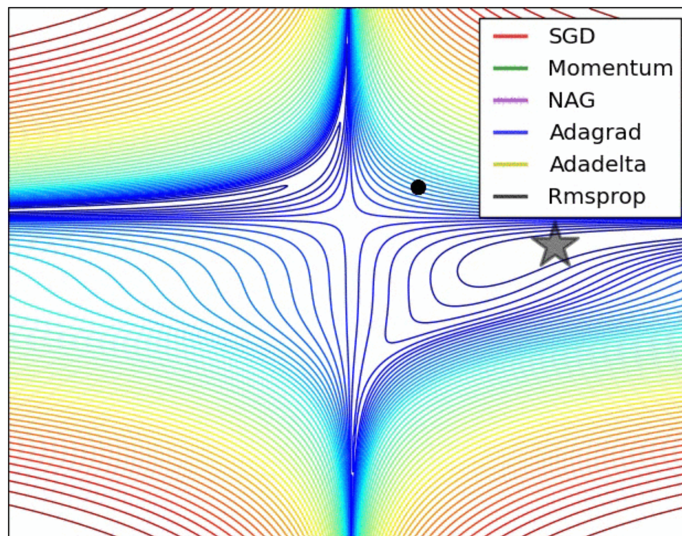
```
cache = decay_rat...        decay_rate) * dx**2
x += - learning_ra...     / (np.sqrt(cache) + eps)
```

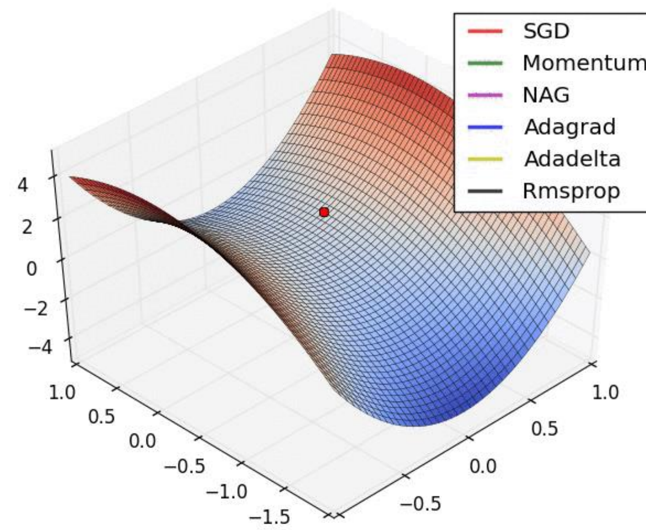
RMSProp

```
# 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

Classical Regularization Still Works!

L2 Regularizer ("weight decay"):

$$C = \underbrace{-\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right]}_{C_0} + \frac{\lambda}{2n}\sum_w w^2$$

The learning rule for the weights becomes:

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

L1 Regularizer:

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

The learning rule becomes:

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

Regularization: L2 Penalty

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

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

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

http://neuralnetworksanddeeplearning.com/chap3.html

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

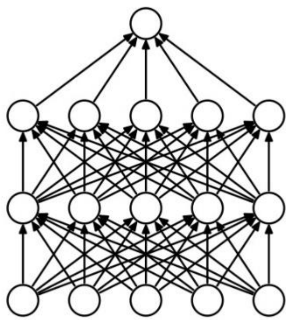
$$w \rightarrow w' = w - \frac{\eta\lambda}{n} \mathrm{sgn}(w) - \eta \frac{\partial C_0}{\partial w},$$

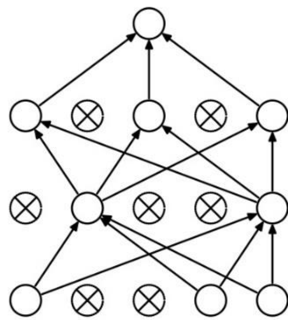
http://neuralnetworksanddeeplearning.com/chap3.html

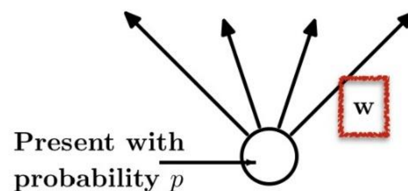
Drop-out/"Coadaptation": An NN Regularizer

More white board time incoming!!

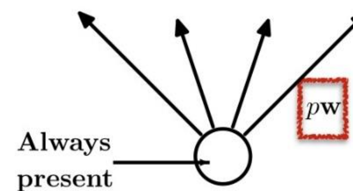
- Feature coadaptation: during learning, weights settle into their w/in network
  - Neuronal weights tuned for specific features = some specialization
  - Neighboring neurons end up relying on this specialization → could result in a fragile model too specialized to the training data
- Each iteration,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

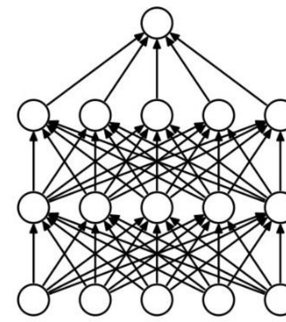


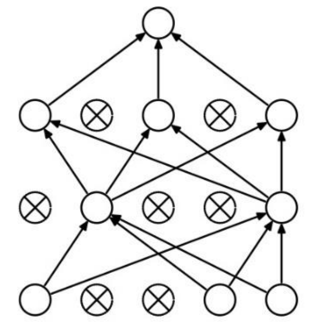
(d) At test time

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

White Board Time!

(Dropout)


(a) Standard Neural Net


(b) After applying dropout.

Another Nice Trick: 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)

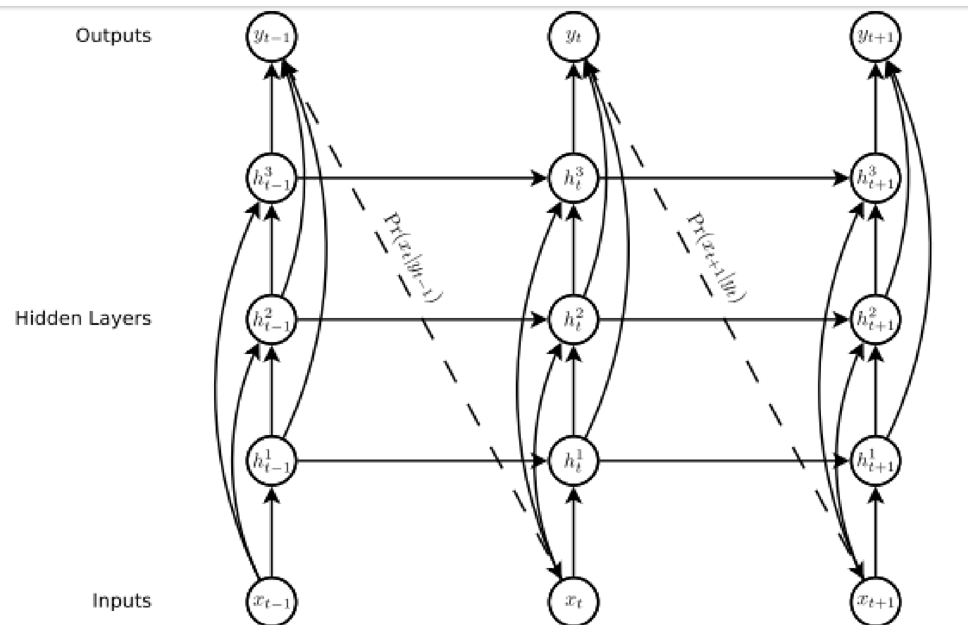


Figure 1: **Deep recurrent neural network prediction architecture.** The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions.

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

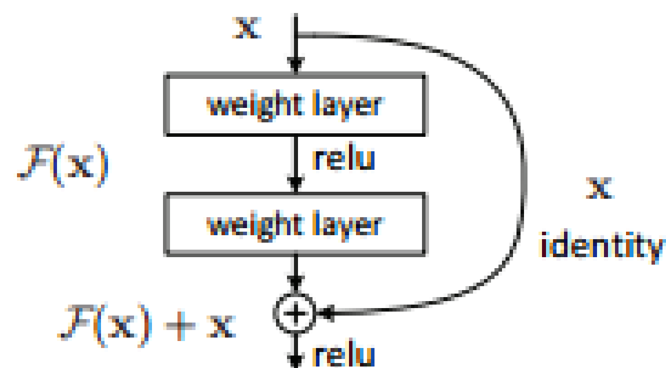


Figure 2. Residual learning: a building block.

https://arxiv.org/abs/1512.03385

QUESTIONS?