Artificial Neural Networks:
The Practice of Neural Smithing

Alexander G. Ororbia II

Introduction to Machine Learning

CSCI-635

11/29/2023

Could Pretrain…or…Use Smarter Random Initializations

- Classical simple 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…Just Wait Longer?!

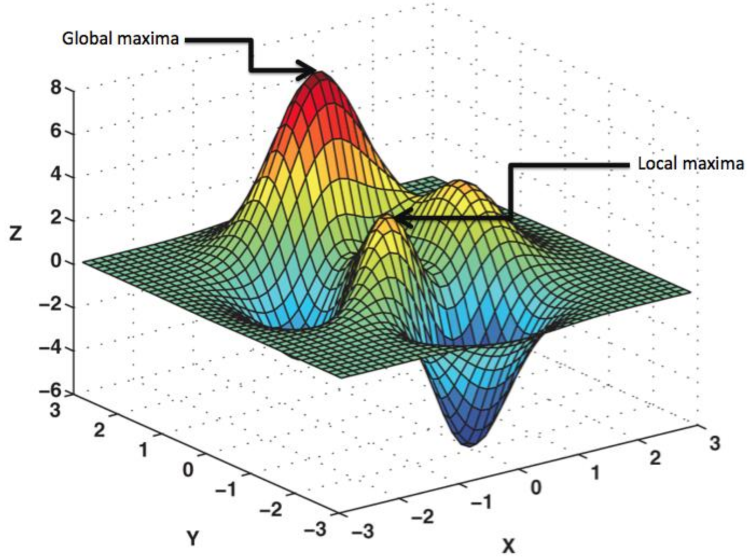
- Even with poor initialization, just wait a really long time….
- Patience + really good hardware
- So one answer = ***more hardware***

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
  - Stochastic gradient descent (SGD)
- Alternative optimizers = shiny toys to make learning even faster



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

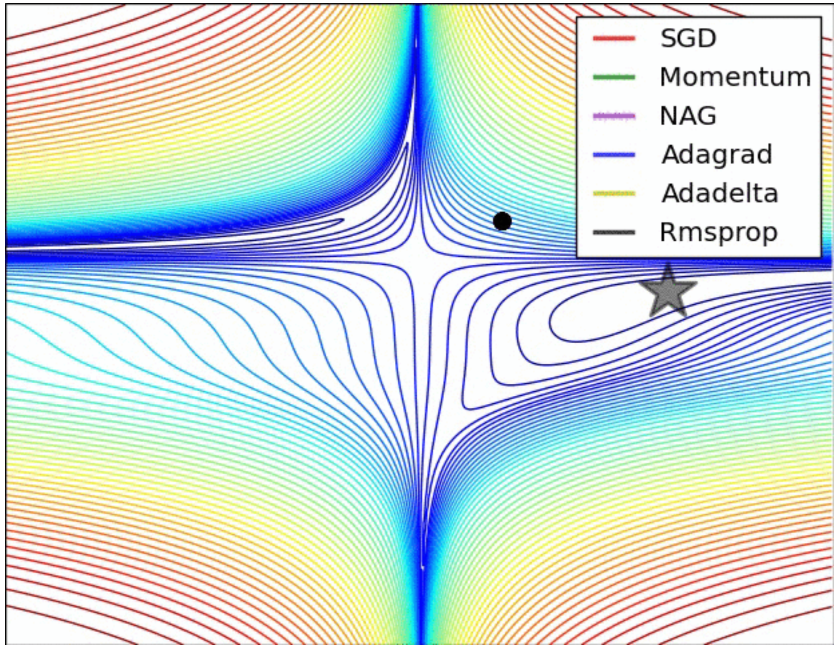
```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

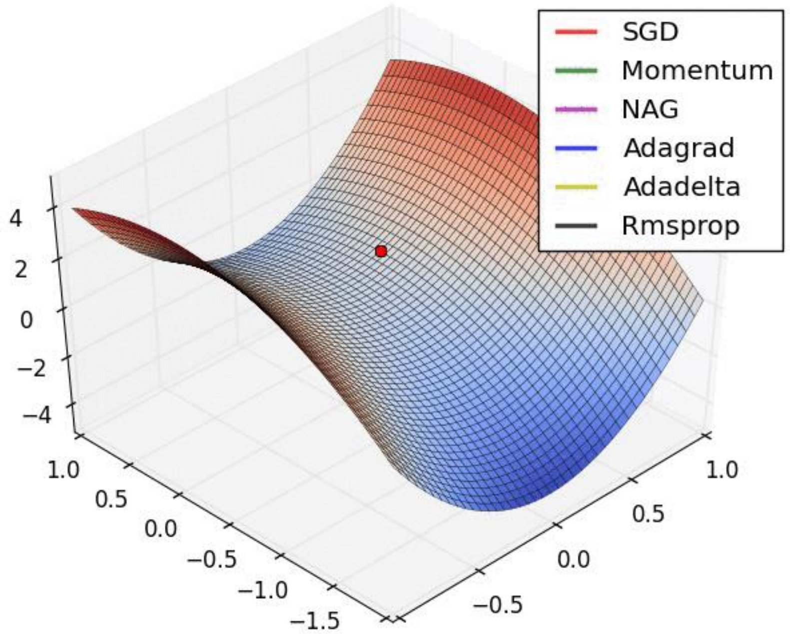
RMSProp

```
# 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

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

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

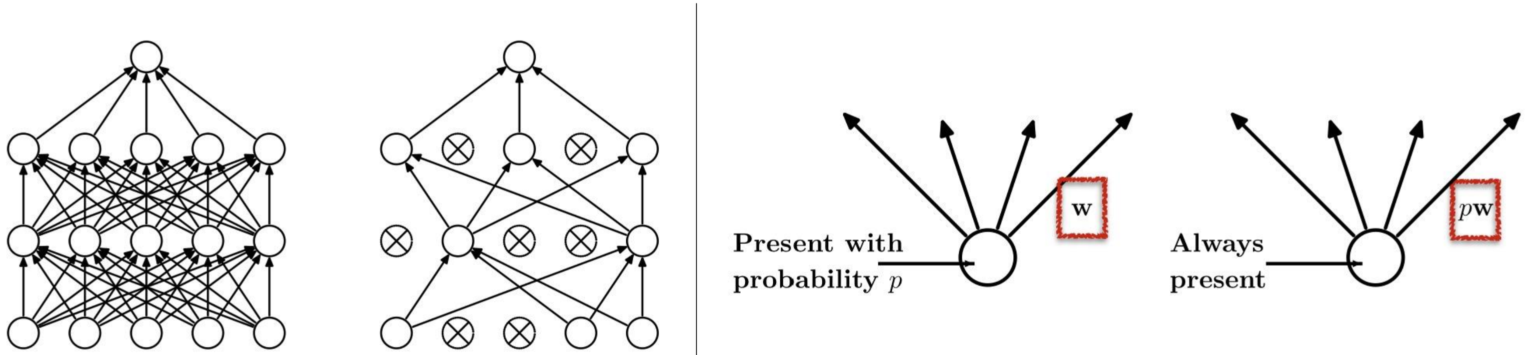
$$w \rightarrow w' = w - \frac{\eta\lambda}{n} \mathrm{sgn}(w) - \eta \frac{\partial C_0}{\partial w},$$

http://neuralnetworksanddeeplearning.com/chap3.html

Drop-out and "Coadaptation"

More white board time incoming!!

- 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!

White Board Time!

(Dropout)

QUESTIONS?