UO 631 Parallel Processing: Multi-layer Perceptron Classification

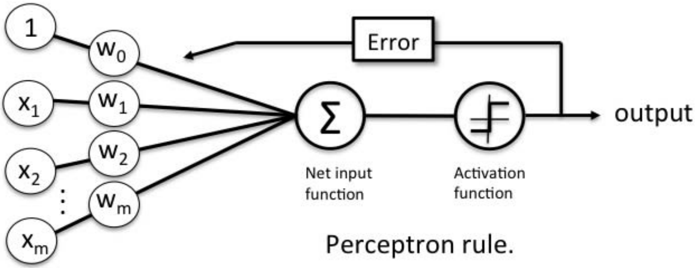
Luis Guzman & Steven Walton
University of Oregon

December 3, 2020

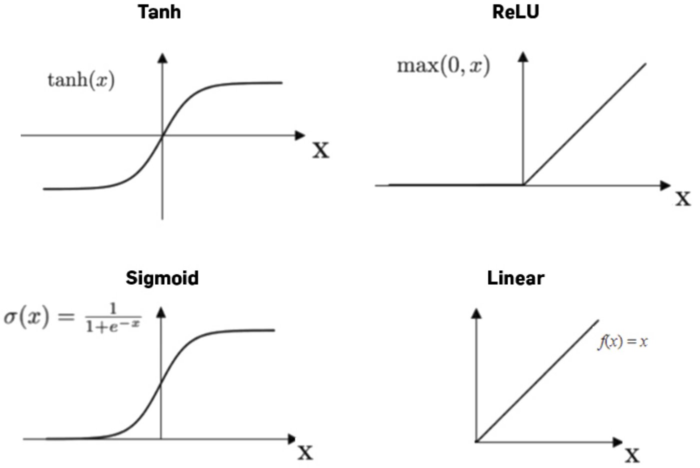
- **What are Multi-Layer Perceptrons**
- Gradient Descent
- Code Profiling
- CPU Parallelization
- GPU Parallelization
- Performance Results
- Next Steps

Perceptron

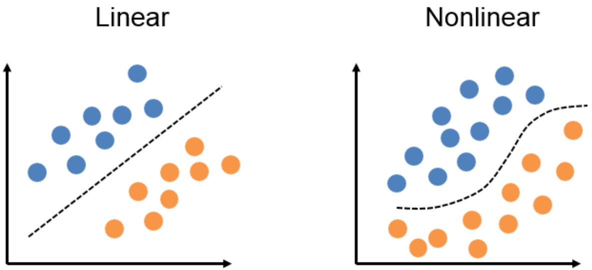
- $\hat{y} = g(\sum_{i=0}^{n} w_i x_i)$



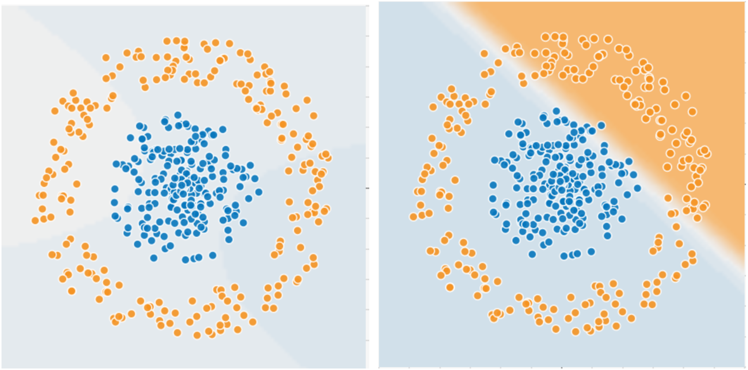
Activation Functions



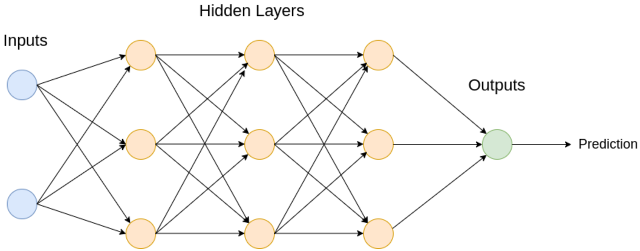
Decision Boundary

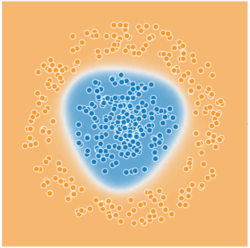


Limitations

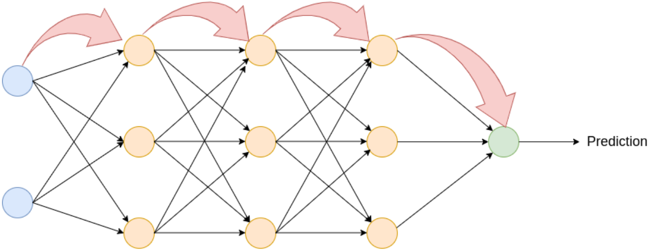


Multi-Layer Perceptron





Making predictions: Forward pass



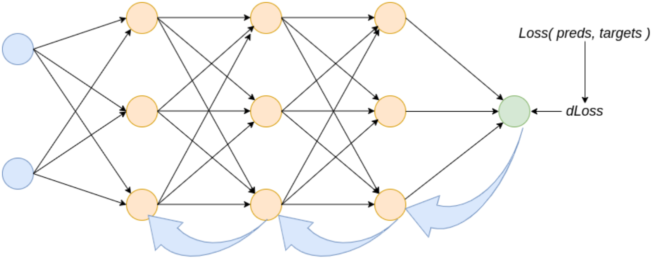
- $h_{1_{act}} = Inputs \cdot W_{h_1}$
- $h_{2_{act}} = h_{1_{act}} \cdot W_{h_2}$
- $h_{3_{act}} = h_{2_{act}} \cdot W_{h_3}$
- $\hat{y} = h_{3_{act}} \cdot W_O$
- How do we improve the predictions?

- What are Multi-Layer Perceptrons
- **Gradient Descent**
- Code Profiling
- CPU Parallelization
- GPU Parallelization
- Performance Results
- Next Steps

Gradient Descent

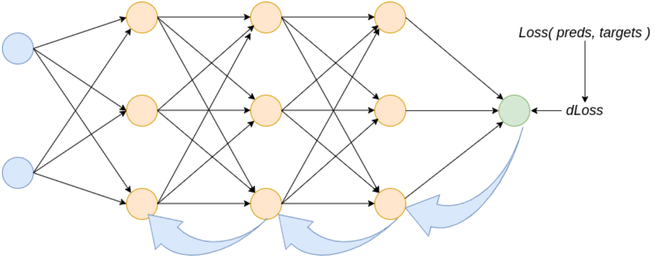
- Once a prediction has been made, find out *how wrong* its with a *Loss function.*
- Binary Cross Entropy Loss:
  - $L = y * log(\hat{y}) + (1 - y) * log(1 - \hat{y})$
- Gradient descent step:
  - $w = w - \gamma \frac{d}{d_w} L$
  - γ - learning rate parameter

Backward Pass: Back-propagating the error



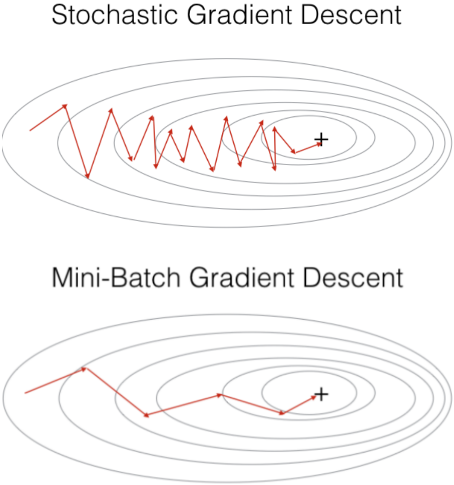
- $O_{error} = dLoss$
- $h3_{error} = O_{error} \cdot (W_O)^T$
- $h2_{error} = h3_{error} \cdot (W_{h_3})^T$
- $h1_{error} = h2_{error} \cdot (W_{h_2})^T$

Backward Pass: Updating the weights



- $W_{h_1} = W_{h_1} - \gamma((Inputs)^T \cdot h1_{error})$
- $W_{h_2} = W_{h_2} - \gamma((h1_{act})^T \cdot h2_{error})$
- $W_{h_3} = W_{h_3} - \gamma((h2_{act})^T \cdot h3_{error})$
- $W_O = W_O - \gamma((h3_{act})^T \cdot O_{error}\ )$

Stochastic GD vs (Mini) Batch GD



- What are Multi-Layer Perceptrons
- Gradient Descent
- **Code Profiling**
- CPU Parallelization
- GPU Parallelization
- Performance Results
- Next Steps

Code Profiling

- Used GProf to profile code.
- Found slowest function *we wrote* was our matrix multiply.
- Found that allocating data took more time.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 12.99      0.63     0.63 232252000     0.00     0.00  std::vector<std::vector<float, std::allocator<float> >, st
  6.80      0.96     0.33 19220148     0.00     0.00  std::move_iterator<float*> std::__make_move_if_noexcept_ite
  4.33      1.17     0.21 130682022     0.00     0.00  std::vector<float, std::allocator<float> >::size() const
  3.92      1.36     0.19 156813500     0.00     0.00  std::vector<float, std::allocator<float> >::operator[](uns
  ...
  2.68      1.49     0.13   170500     0.00     0.00  math_funcs::matrix_mult(std::vector<std::vector<float, std:
<std::vector<float, std::allocator<float> > > >, std::vector<std::vector<float, std::allocator<float> >, std::al
```

- What are Multi-Layer Perceptrons
- Gradient Descent
- Code Profiling
- **CPU Parallelization**
- GPU Parallelization
- Performance Results
- Next Steps

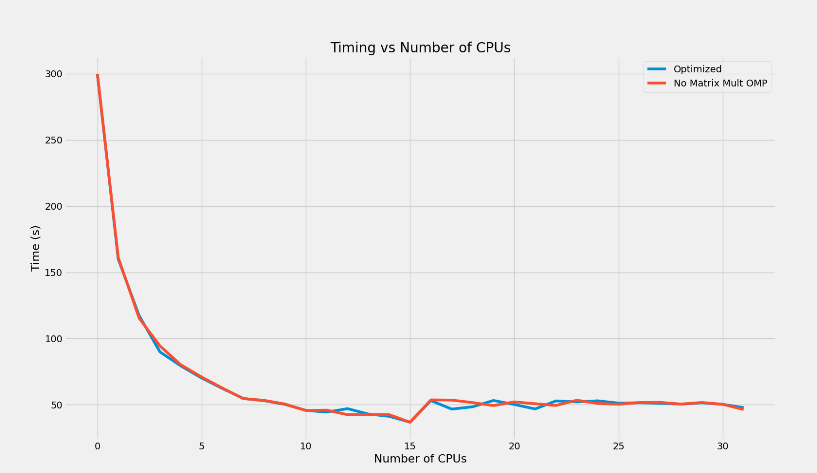
CPU Optimization/Parallelization

- Started with gpof to see what was slowing things down.
- Optimized serial version looking to make sure it was memory efficient (loops).
- Found the matrix multiplication and transpose were the most heavy computation.

Optimizing Matrix Multiplication

- Original implementation was not great for parallelization.
- Introduced batching.
- Found that if the problem was too small parallelization made things worse (actually need DNN).
- Tried multiple versions of matrix multiply (2 naive and blocking)

Performance



- What are Multi-Layer Perceptrons
- Gradient Descent
- Code Profiling
- CPU Parallelization
- **GPU Parallelization**
- Performance Results
- Next Steps

CUDA

- Intended to get entire network into GPU but did not have enough time.
- Implemented matrix multiplication.
- Struggled more with getting CUDA configured than writing actual code :(
- Problem being small still resulted in not enough work.

CUDA

OMP (μs)	CUDA (μs)	Kernel (μs)
1.213	121.734	4.98

- What are Multi-Layer Perceptrons
- Gradient Descent
- Code Profiling
- CPU Parallelization
- GPU Parallelization
- Performance Results
- **Next Steps**

What Next?

- Get **full** model into GPU.
- Test with cuda optimized functions and compare to ours.
- Introduce optimized CPU libraries.
- Flatten 2D arrays and vectors to decrease cache misses.