Hardware for Training

MICRO Tutorial (2016)
Website: http://eyeriss.mit.edu/tutorial.html
Joel Emer, Vivienne Sze, Yu-Hsin Chen
Cost function for Model Training

Model output: \( y = f(x) \)

Desired output: \( z \)

Error: \( e = (y - z) \)

Over all training inputs \( x \):

Minimize \( \sum(y - z)^2 \)

What do we vary to minimize the error?
Training Optimization Problem

- Model parameters: $\theta$ (include bias, weights, …)
- Model output: $y(\theta) = f(x, \theta)$
- Desired output: $z$
- Error: $e(\theta) = y(\theta) - z$
- Cost function*: $E(\theta) = \Sigma e(\theta)^2$
- Minimization: $dE(\theta)/d\theta = 0$ (but no closed form)

* Over all inputs in the training set
Steepest descent

Classical first order iterative optimization scheme: Gradient is steepest descent – follow it!

\[ \theta^{n+1} = \theta^n - \alpha \cdot \frac{dE(\theta^n)}{d\theta} \]

where \( \alpha \) is the step size along the gradient…
Calculating Steepest Descent

Also called error back-propagation

- **Steepest descent**
  \[ \theta^{n+1} = \theta^n - \alpha \cdot \frac{dE(\theta)}{d\theta} \]

- **E(\theta)**
  \[ = \sum e(\theta)^2 \]
  \[ = \sum (y(\theta) - z)^2 \]

- **dE(\theta)/d\theta**
  \[ = 2 \cdot \sum [(y(\theta) - z) \cdot \frac{dy(\theta)}{d\theta}] \]
Chain rule -> Back propagation

• The chain rule of calculus allows one to calculate the derivative of a layered network, i.e., a composition of functions, iteratively working backwards through the layers using the (feature map) values of the layer, i.e., function, and the derivative from the next layer.

• Back propagation is the process of doing this calculation numerically for a given input.
Per Layer Calculations

\[ y = f(x) \]

For layer \( k \):
- Inputs: \( x^k \)
- Weights: \( w^k \)
- Outputs: \( y^k \)

So
\[ y_i^k = f^k [\Sigma (w_{ij}^k x_j^k)] \]

Where
\[ x_j^k = y_j^{k-1} \]

or
\[ y^k = f^k(y^{k-1}, \theta) \]
Layer Operation Composition

- **Steepest descent**
  \[ \theta^{k+1} = \theta^k - \alpha \cdot \frac{dE}{d\theta} \]

- **Derivative (1)**
  \[ \frac{dE}{d\theta} = 2 \cdot \Sigma [(y(\theta) - z) \cdot \frac{dy(\theta)}{d\theta}] \]

- **Model output**
  \[ y = f(x) \]
  \[ y^n = f^n(y^{n-1}) = f^n(f^{n-1}(y^{n-2})) \]

- **Layer k**
  \[ y^k = f^k(y^{k-1}) = f^k(f^{k-1}(y^{k-2})) \]
Chain rule

- Chain rule for functions

\[ y = f(g(x)) \]

\[ y' = f'(g(x)) \cdot g'(x) \]

\[ y = f^n(y^{n-1}) = f^n(f^{n-1}(y^{n-2})) \]

\[ y' = f^n'(f^{n-1}(y^{n-2})) \cdot f^{n-1}'(y^{n-2}) \]

\[ = f^n'(y^{n-1}) \cdot f^{n-1}'(y^{n-2}) \]
Back propagation

- $y_{00} = (a \cdot x_0 + b \cdot x_1)$
- $y_{01} = \ldots$
- $y_{10} = y_{00} \cdot c + y_{01} \cdot d$
- $\frac{dy_{10}}{da} = \frac{dy_{10}}{dy_{00}} \cdot \frac{dy_{00}}{da} = c \cdot x_0$
Back Propagation for Addition

- $y_0 = a + b$
- $y_1 = f(y_0)$
- $\frac{dy_0}{da} = 1$
- $\frac{dy_0}{db} = 1$
- $\frac{dy_1}{dy_0} = f'(y_0)$
- $\frac{dy_1}{da} = \frac{dy_1}{dy_0} \times \frac{dy_0}{da} = \frac{dy_1}{dy_0} \times 1 = \frac{dy_1}{dy_0}$
- $\frac{dy_1}{db} = \frac{dy_1}{dy_0} \times \frac{dy_0}{db} = \frac{dy_1}{dy_0} \times 1 = \frac{dy_1}{dy_0}$
Back Propagation for Multiplication

- $y_0 = a \times b$
- $y_1 = f(y_0)$
- $dy_0/da = b$
- $dy_0/db = a$
- $dy_1/dy_0 = f'(y_0)$
- $dy_1/da = dy_1/dy_0 \times dy_0/da = dy_1/dy_0 \times b$
- $dy_1/db = dy_1/dy_0 \times dy_0/db = dy_1/dy_0 \times a$
Back propagation for Network

- \( y_{00} = (a \cdot x_0 + b \cdot x_1) \)
- \( y_{01} = \ldots \)
- \( y_{10} = y_{00} \cdot c + y_{01} \cdot d \)
- \( \frac{dy_{10}}{da} = \frac{dy_{10}}{dy_{00}} \cdot \frac{dy_{00}}{da} = c \cdot x_0 \)
Back Propagation Recipe

Start point
- Select a initial set of weights ($\theta$) and an input ($x$)

Forward pass
- For all layers
  - Compute layer outputs use as input for next layer (and save for later)

Backward pass
- For all layers (with output of previous layer and gradient of next layer)
  - Compute gradient, i.e., (partial) derivative, for layer
  - Back-propagate gradient to previous layer
  - Compute (partial) derivatives for (local) weights of layer

Calculate next set of weights
- $\theta^{k+1} = \theta^k - \alpha \cdot \frac{dE}{d\theta}$
Back Propagation

Forward Pass ->

Convolution (n)

Weights

Backward Pass

<- Backward Pass

Weight gradients
Learning rate may be very small (10^{-5} or less)

\[ \alpha \]

\[ \Delta w \text{ very small} \]

\[ \begin{align*}
    a_j & \xrightarrow{\times} x \\
g_j & \xrightarrow{\times} x \\
\end{align*} \]

\[ \begin{align*}
    \Delta w_{ij} & \xrightarrow{SR} \Delta w'_{ij} \\
E(\Delta w'_{ij}) & = \Delta w_{ij} \\
w_{ij} & \xrightarrow{+} w_{ij}
\end{align*} \]

- Beware truncating changes to zero
- Rounding can bias result -> use stochastic rounding

[Gupta et al., ICML 2015]
Back Propagation Batches

Issue:
• $N = 1$ is often too noisy, weights may oscillate around the minimum

Solution:
• Use batches of $N$ inputs…
• Max theoretical speed up: $N$
Parallel creation of gradient

- Steepest descent
  \[ \theta^{k+1} = \theta^k - \alpha \cdot \frac{dE}{d\theta} \]

- Derivative
  \[ \frac{dE}{d\theta} = 2 \cdot \sum [(y(\theta) - z) \cdot \frac{dy(\theta)}{d\theta}] \]

Split sum of pieces of \( \frac{dE}{d\theta} \) across different nodes!
Batch Parameter Update

Parameter Server \( p' = p + \Delta p \)

Model Workers

Data Shards

[Dean et al., NIPS 2012]
Training Uses a Lot of Memory

GPU memory usage proportional to network depth

[Rhu et al., vDNN, MICRO 2016]
How Much Memory Is It?

Up to Tens of Gigabytes

GPU memory

Deeper networks (VGG-like topology)

[Direct et al., vDNN, MICRO 2016]
Reuse Distance of Feature Maps

[Reused from Rhu et al., vDNN, MICRO 2016]
Problems with saturation

**Issue**

- A null gradient results in no learning, which happens if:
  - the sigmoid saturates, or
  - the ReLU saturates

**Solution**

- Initialize weights so the average value is zero, i.e., work in the interesting zone of the activation functions
- Normalize data (zero mean)
Non-differential operations

Issue

• Discrete activation function / weights
  – extreme case is binary net
• Derivative not well defined

Solution

• Use approximate derivative, or
• Discretize a-posteriori
Model Overfitting

Problem:
• Neural net learns too specifically from input set, rather than generalizing from input, called overfitting
• Overfitting can be a result of too many parameters in model

Solution:
• Dropout – turn off neurons at random; other neurons will take care of their job.
  – + Reliability
  – - Redundancy (-> pruning)
Architecture Challenges for Training

- Floating point accuracy
- Where to store the gradients
- Synchronization for parallel processing