An example of how we learn XOR using FFNN with hidden layer

⇒ Hidden layer mapping our data to a separatable space.

Gradient - Based Learning

- cost functions
  - Learning conditional distribution with Maximum Likelihood

    Relationship of likelihood $P(X|\theta)$, with the cost function $-\ln P(X|\theta)$
  - Learning Conditional Statistics

    which means we often learn a statistics instead of a distribution of $P(y|X,\theta)$
- Output Units

  Discussion of different type of output units
  - Linear output for Gaussian Output Distribution
  - Sigmoid units for Bernoulli Distribution

    Discussion of some interesting property of sigmoid function when we combine it with logarithm.
  - Softmax units for Multinoulli Distribution
  - other types of output units

    Here discussion focus on we can also model mixture Gaussian with FFNN or predict/inference variance using FFNN - by modifying the output of the neural network.

— Hidden Units

Talking about how we design Hidden Units (Considering Gradients and non-linearity)

- Rectified Units and their generation

  keep the linearity of positive values, some generation (maxout) also keep linearity.

- Logistic Sigmoid and Hyberbolic Tagent (Tanh)

For FFNN networks, sigmoid function may be not a good choice, some times tanh is more preferable cause it has more steep shape.

- Other Hidden Units

- Architecture design

- Universal Approximation properties and depth

Universal Approximation Theorem: MLP can theoretically simulate all functions but deeper neural network saves the parameters.

- Other considerations

- Back-propagation and other differentiation algorithms

- Computational Graphs

Fundemental data structure of backward and forward caculation.

- Chain Rule of Calculus

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \quad \text{for scalar}$$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \quad \text{for vector}$$

$$\nabla_X z = \sum_j (\nabla_X Y_j) \frac{\partial z}{\partial Y_j} \quad \text{for tensor}$$

- Recursively applying chain rule to get Back prop.

Defining computational process $\underbrace{u^1, u^2, \dots, u^i, \dots u^{n_i}}_{\text{Input node}}$ $\underset{\text{output}}{\underline{u^{(n)}}}$

Forward: $A^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$

$u^{(i)} \leftarrow f^{(i)}(A^{(i)})$

Backward: grad_table$[u^{(n)}] = 1$

using grad_table$[u^{(j)}]$

For $j = n-1$, down to 1 do

$$\frac{\partial u^{(n)}}{\partial u^{(i)}} = \sum_{j: j \in Pa(u^{i})} \boxed{\frac{\partial u^{(n)}}{\partial u^{(j)}}} \cdot \frac{\partial u^{(j)}}{\partial u^{(i)}}$$

– Back-Propagation Computation in fully connected MLP

Forward: $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$

$h^{(k)} = f(a^{(k)})$, until we get $h^{(l)}$, $J = L(\hat{y}, y) + \lambda\Omega$

Backward: $g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$

for $k = l, \dots, 1$ do:

$g \leftarrow \nabla_{a^{(k)}} J = g \odot f'(a^{(k)})$

compute activation gradient

Compute gradient for parameters

$\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$

$\nabla_{W^{(k)}} J = g h^{(k-1)} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$

$g \leftarrow \nabla_{h^{(k-1)}} J = W^{(k)T} \cdot g$

– Symbol-to-Symbol Derivatives

Two different computing approach (Symbol-to-number / Symbol)

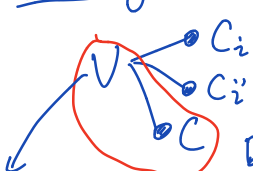
– General Back-Propagation

General framework: updating $\boxed{\text{grad\_table}}$

grad_table[z] ← 1

for V in T do: $\boxed{\text{build\_grad}}$(V, G, G', grad_table)

zoom into $\boxed{\text{build\_grad}}$:



$G \leftarrow \sum_i G^{(i)}$

grad_table[V] = G

recursive process for the child C

D: build_grad(C, G, G', grad_table)

$G^{(i)}$: op.bprop(get_inputs(C, G'), V, D)

calculate the gradient of this operation

– An example of back-prop MLP

– Complications

- Differentiate outside deep learning community.
  - Back-Prop is a kind of automatic differentiation
  - Consideration of order of vector-matrix multiplication
  - Back-prop cannot utilize (generally) math tricks due to its generality.