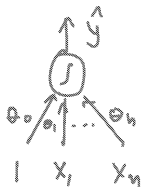
Artificial neuron

- **An artificial neuron is a linear classifier with activation**
- **A neuron fires if the weighted sum of inputs is above a threshold**
- **A neural network is a collection of artificial neurons**



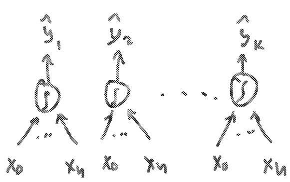
$$\hat{y} = \overset{\text{activation}}{f}(\theta_1 x_1 + \cdots + \theta_n x_n + \theta_0)$$

$$= f(\theta^T x + \underset{\text{bias}}{\theta_0})$$

$$= \begin{cases} 1 & \text{if } \theta^T x > \theta_0 \\ 0 & \text{otherwise} \end{cases}$$

Neural network layer

- **A neural network with a single layer is a set of K artificial neuron classifiers**



$$\hat{y}_1 = f(\underset{1\times n}{\theta_1^T}\ \underset{n\times 1}{x} + \underset{1\times 1}{\theta_{10}})$$

$$\vdots$$

$$\hat{y}_K = f(\theta_K^T x + \theta_{K0})$$

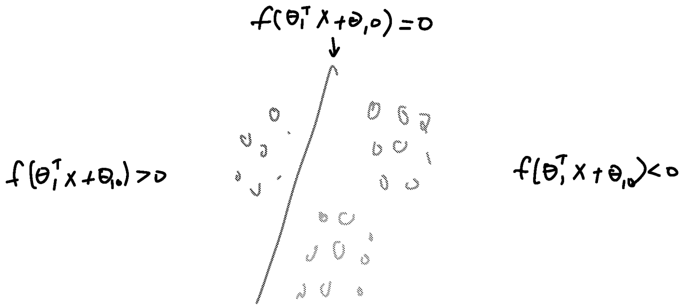
$$\underset{k\times 1}{\hat{y}} = f(x;\theta) = \underset{k\times n}{\theta^T}\ \underset{n\times 1}{x} + \underset{k\times 1}{\theta_0}$$

Neural network layer interpretation

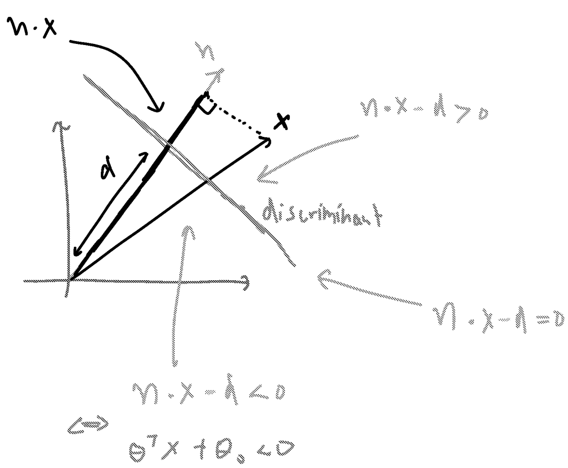
* perform linear discrimination
  => each unit defines a linear decision boundary
* perform template matching
  => each unit defines a template
* perform projection into a new basis
  => each unit defines a basis vector

Linear discrimination interpretation

- The rows of the weight matrix θ^T represent parameters of K linear discriminant functions
- Each linear discriminant function separates one class from all others
- The discriminant function measures distance from a linear decision boundary. The distance is positive for examples belonging to the class and negative for all other examples

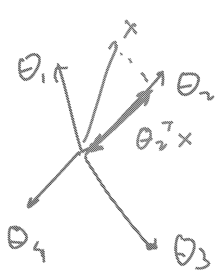


Linear discriminants



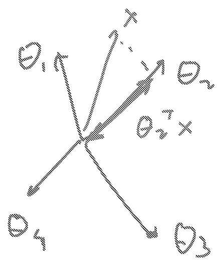
Template matching interpretation

- The rows of the weight matrix θ^T represent templates
- With K rows we have K templates (one template per class)
- The product $\theta^T x$ measures how well The input vector matches the template (dot product between vectors)
- High similarity to the template of a particular class indicates membership in this class



Projection into a new basis interpretation

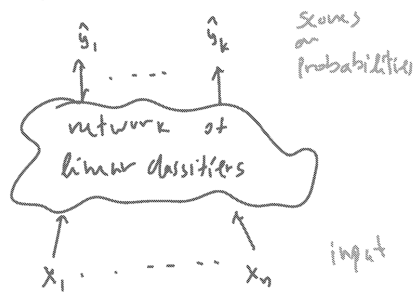
- **The rows of the weight matrix θ^T represent basis vectors**
- **By taking the dot product of X with each of the rows we move to a representation in a new basis and so transform the data**



Linear classifiers

- what if decision boundaries are non-linear?
- what if more than one template is needed per class?

⇒ Combine multiple linear classifiers

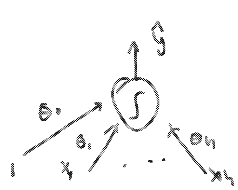


Activation functions

- **Combining linear classifiers in sequence can be reduced to a single layer (linear combination of linear combinations is just a different linear combination)**
- **Therefore we need a nonlinear activation function between linear units**

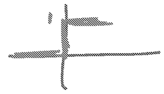
non-linear activation

$$\hat{y} = h(\underbrace{\theta_0 + \theta_1 x_1 + \dots + \theta_n x_n}_{\text{linear combination}})$$

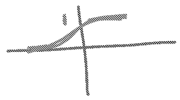


Activation functions

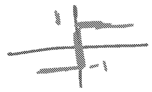
step: $h(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$



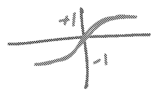
sigmoid: $h(a) = \dfrac{1}{1 + \exp(-a)}$



step*: $h(a) = \begin{cases} 1 & \text{if } a > 0 \\ -1 & \text{otherwise} \end{cases}$



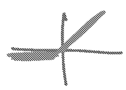
tanh: $h(a) = \tanh(a) = \dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$



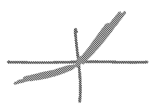
Activation functions

ReLU: $h(a) = \max(0, a)$

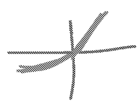
(Rectified Linear Units)



leaky ReLU: $h(a) = \max(\alpha a, a)$



ELU: $h(a) = \begin{cases} a & a \geq 0 \\ \alpha(e^{a} - 1) & a < 0 \end{cases}$



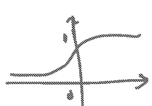
Converting similarity to probability at the output layer

- Given similarity score (or decision boundary distance s_j) from a linear classifier (higher better), compute:

$$\hat{y}_j^{(i)} \equiv P(y = j \mid x^{(i)})$$

- In a 2-class classification case:

$$\hat{y}_j^{(i)} \equiv P(y = j \mid x^{(i)}) = \text{sigmoid}(s_j^{(i)})$$



$\text{sigmoid}(x) = \dfrac{1}{1 + e^{-x}}$

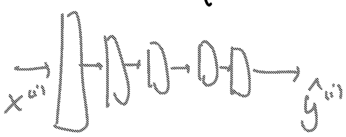
- In a k-class classification case

$$\hat{y}_j^{(i)} \equiv P(y = j \mid x^{(i)}) = \frac{\exp(s_j^{(i)})}{\sum_{l=1}^{K} \exp(s_l^{(i)})} \leftarrow \text{Softmax}$$

Training the network

- Training: find weights for all network layers
- Loss: difference between obtained expected output
- Optimization: find the network parameters that will minimize the loss
- Select:
  - Lost function
  - Optimization framework

Given Training examples $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$



$$L(\theta) = \sum_{i=1}^{n} d(y^{(i)}, \hat{y}^{(i)}) \qquad \theta^* = \underset{\theta}{\operatorname{argmin}}\, L(\theta)$$

Loss function

- The weights θ are set to minimize loss:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, L(\theta)$$

sample loss

$$= \underset{\theta}{\operatorname{argmin}}\, \frac{1}{m} \sum_{i=1}^{m} L_i(x^{(i)}, y^{(i)}; \theta)$$

$$= \underset{\theta}{\operatorname{argmin}}\, \frac{1}{m} \sum_{i=1}^{m} L_i(\underbrace{f(x^{(i)}; \theta)}_{\hat{y}^{(i)}}, y^{(i)})$$

L1 and L2 loss

- In regression problems minimize distance between known and predicted values:

$$L_1: \quad L_i(\theta) = \sum_{j=1}^{k} |\hat{y}_j^{(i)} - y_j^{(i)}|$$

$$L_2: \quad L_i(\theta) = \sum_{j=1}^{k} (\hat{y}_j^{(i)} - y_j^{(i)})^2$$

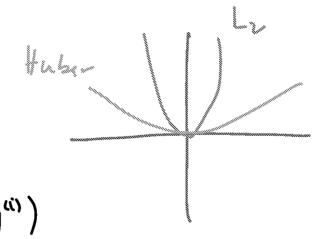
$y_j^{(i)}$ is predicted value: $\quad y_j^{(i)} \in \mathbb{R}$

Huber loss

- cap loss of outliers (robust regression)

$$\rho_\delta(d) = \begin{cases} \frac{1}{2} d^2 & \text{if } |d| \le \delta \\ \delta\left(d - \frac{1}{2}\delta\right) & \text{otherwise} \end{cases}$$

{ quadratic for small d
{ linear for large d



$$L_i(\theta) = \sum_{j=1}^{k} \rho_\delta\left(\hat{y}^{(i)} - y^{(i)}\right)$$

Cross entropy loss

* In classification problems assume that the true class label $y^{(i)}$ is one-hot-encoded

$$x^{(i)} \in C_k \Rightarrow y^{(i)} = \begin{array}{ccccccccccc} 1 & 2 & 3 & \cdots & & & & k & \cdots & & \\ \hline \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{0} \end{array}$$

* The outputs of the network are assumed to be the probabilities of belonging to each class

$$\hat{y}^{(i)} = \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & \cdots & & k & \cdots & & \\ \hline \boxed{0} & \boxed{.1} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{0} & \boxed{.9} & \boxed{0} & \boxed{0} & \boxed{0} \end{array}$$

$$\hat{y}_j^{(i)} = P\left(y = j \mid x^{(i)}\right)$$

Cross entropy loss

- Probability of the true class: $\prod_{j=1}^{k} \left(\hat{y}_j^{(i)}\right)^{y_j^{(i)}}$

- Likelihood: $L(\theta) = \prod_{i=1}^{m} \prod_{j=1}^{k} \left(\hat{y}_j^{(i)}\right)^{y_j^{(i)}}$

  ↑ maximize

- negative Log-likelihood: $\ell(\theta) = -\log L(\theta) = -\sum_{i=1}^{m} \underbrace{\sum_{j=1}^{k} y_j^{(i)} \log\left(\hat{y}_j^{(i)}\right)}_{\text{sample loss}}$

  ↑ minimize (negative LL)

Cross entropy loss

- possible cross-entropy loss values:

$L_i(\theta) \in [0, \infty]$ 0 when $p \to 1$, ∞ when $p \to 0$

- For random class assignment:

$\hat{y}_j^{(i)} = P(y=j \mid x^{(i)}) = \frac{1}{K}$

$\ell(\theta) = -\sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log\left(\frac{1}{K}\right) = -\sum_{i=1}^{m} -K = mK$

bad loss value

Regularization

- Occam's razor: simpler explanations are better
- A simpler solution is when the weights θ are lower (e.g. when $\theta_{ij} = 0$ we remove one coefficient).
- Smaller coefficients → more stable solution that will generalize better

$L(\theta) = \frac{1}{m} \sum_{i=1}^{m} L_i(f(x_i; w), y_i) + \lambda \underbrace{R(\theta)}_{\text{regularization term}}$

λ: weight of regularization (hyper parameter)

$R(\theta) = \sum_i \sum_j \theta_{ij}^2$

Regularization

- Alternative Regularization (L_1):

$R(\theta) = \sum_i \sum_j |\theta_{ij}|$

- L_2 Regularization minimizes weights while spreading them (e.g. $0.5^2 + 0.5^2 < 1^2 + 0^2$)
- L_1 Regularization does not have this property and may concentrate weights (e.g. $|0.5| + |0.5| = |1| + |0|$)

Optimization

- To minimize the loss (objective) solve:

$$\theta^* = \arg\min_{\theta} L(\theta)$$

$\Rightarrow \nabla L(\theta) = 0 \Rightarrow \theta^*$

- If $\nabla L(\theta)$ is not linear it is hard to find an explicit solution
  $\Rightarrow$ numerical iterative solution
  (e.g. Gradient descent)

Gradient descent

* Iterative optimization algorithm
* start with guess θ_0
* Repeat:

learning rate

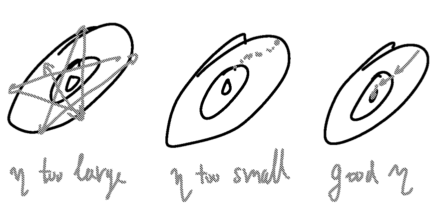
$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta \nabla L(\theta^{(i)})$$

* stop condition:

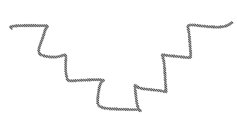
$$|L(\theta^{(i+1)}) - L(\theta^{(i)})| < \tau$$

Gradient descent

* The learning rate has to be selected correctly



* Does not work on piecewise constant functions
  (e.g. empirical loss)



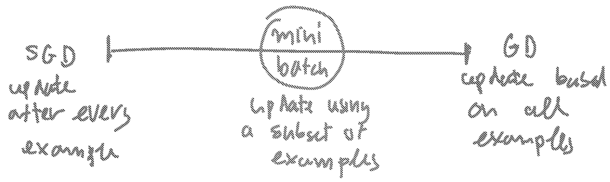
Stochastic gradient descent (SGD)

* Randomly order examples
* For $j = 1..m$

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta \nabla L_j(\theta^{(i)})$$

loss of j-th example

* More frequent but less accurate updates



Data normalization

- Features with high values have more influence and different features require a different learning rate.
(e.g. price ~ 10k vs. age ~ 10)

- Data normalization:

$$\bar{x}_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x_j^{(i)} - \bar{x}_j\right)^2$$

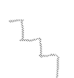
$$\tilde{x}_j^{(i)} = \frac{x_j^{(i)} - \bar{x}_j}{\sigma_j}$$

normalized features

Learning rate decay

- Learning rate need not be fixed
- Make learning smaller as iterations progress

* strategies:
1) step decay:
every k iterations $\eta \leftarrow \eta / 2$



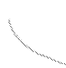
2) exponential decay

$$\eta = \eta_0 e^{-k/t}$$

decay rate
iteration index

3) fractional decay

$$\eta = \eta_0 / (1 + k t)$$



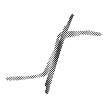
Additional optimization methods

- Newton's method:
  compute learning rate instead of specifying it
  ⇒ second order derivatives (Hessian matrix)
- AdaGrad:
  replace Hessian computation (expensive) with approximation
- RMSProp:
  add decay to AdaGrad
- ADAM:
  add gradient momentum to RMSProp
- Quasi-Newton methods (e.g. BFGS)
  approximate Hessian inverse (faster)

Weight initialization

Θ = constant → all outputs identical
→ no learning
→ random initialization

θ = large → saturated activations,
→ zero gradients
→ no learning
→ initialize close to zero



⇒ Initialize at random close to zero

Weight initialization

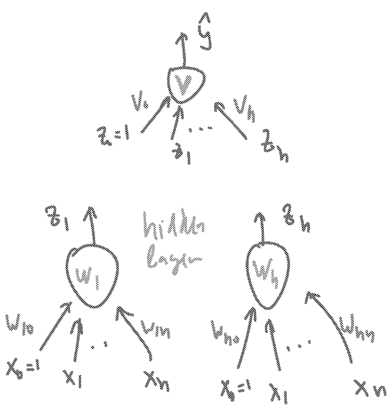
* Glorot (Xavier) normal initialization
- Draw weights from normal distribution with:

$$\mathrm{Var}[w_i] = \left(\frac{2}{\text{fan\_in} + \text{fan\_out}}\right)^{1/2} \qquad \mu = 0$$

* Glorot uniform initialization
- Draw weights from uniform distribution within $[-lim, lim]$

$$lim = \left(\frac{6}{\text{fan\_in} + \text{fan\_out}}\right)^{1/2}$$

Computing gradients using Backpropagation



chain rule:

$$\frac{\partial E}{\partial v} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial v} \rightarrow \text{find } v$$

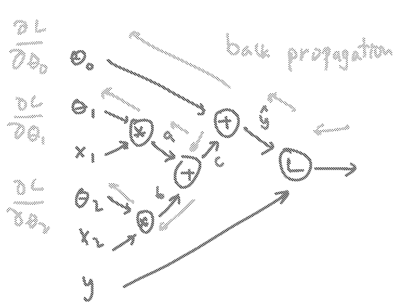
$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial \hat{y}} \underbrace{\frac{\partial \hat{y}}{\partial z_j} \frac{\partial z_j}{\partial w_j}}_{\frac{\partial \hat{y}}{\partial w_j}} \rightarrow \text{find } w_j$$

Backpropagation

- Start with guess for parameters: v, $\{w_j\}$ (at random close to zero)
- Use $x^{(i)}$ and current parameters to compute outputs: $z^{(i)}$, $\hat{y}^{(i)}$ — forward pass
- Use outputs $z^{(i)}$, $\hat{y}^{(i)}$ to update parameters: v, $\{w_j\}$ — backward pass (push back gradients)
- Continue while loss changes

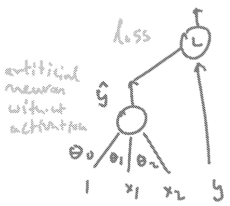
Backpropagation

- Represent the network using a computational graph
- Because each node is simple, it has a simple explicit expression for its derivative
- Forward pass: push input to compute all intermediate node values
- Backward pass: starting with end nodes, push gradients towards the beginning
- Multiply backpropagated gradients (front back) by current gradient and propagate this to the next node

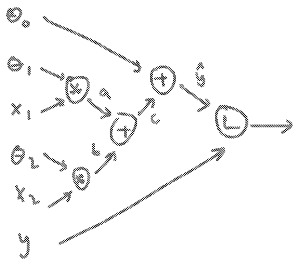


Backpropagation example

network



Computational graph



$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2$

$L = (y - \hat{y})^2$

nodes can be simple or complex

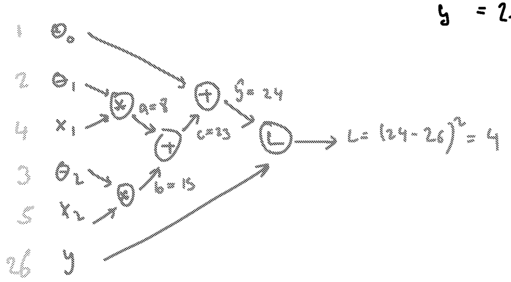
Backpropagation example

* Forward pass

$[\theta_0, \theta_1, \theta_2] = [1, 2, 3]$ ← current guess of parameters

$[x_1, x_2] = [4, 5]$ ← feature vector

$y = 26$ ← expected output

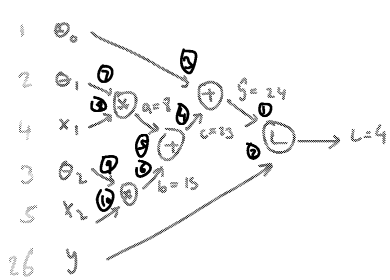


push numbers through network from start to end

Backpropagation example

* Backward pass (compute gradients)

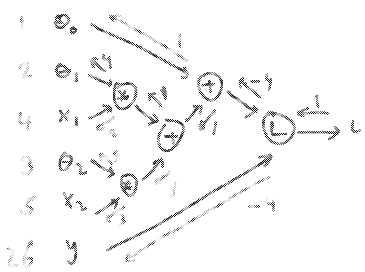
1. For each node: write derivative of output w.r.t. each input
2. Compute gradient at input of each node (using forward values)
3. multiply gradients along paths



1. $\partial L/\partial \hat{y} = 2(y-\hat{y})(-1)$ = -4
2. $\partial L/\partial y = 2(y-\hat{y})(1)$ = -4
3. $\partial \hat{y}/\partial \theta_0 = 1$ = 1
4. $\partial \hat{y}/\partial c = 1$ = 1
5. $\partial c/\partial a = 1$ = 1
6. $\partial c/\partial b = 1$ = 1
7. $\partial a/\partial \theta_1 = x_1$ = 4
8. $\partial a/\partial x_1 = \theta_1$ = 2
9. $\partial b/\partial \theta_2 = x_2$ = 5
10. $\partial b/\partial x_2 = \theta_2$ = 3

Backpropagation example

* Backward pass (contd.)



$$\frac{\partial L}{\partial \theta_0} = 1\cdot(-4)\cdot 1 = -4$$

$$\frac{\partial L}{\partial \theta_1} = 1\cdot(-4)\cdot 1\cdot 1\cdot 4 = -16$$

$$\frac{\partial L}{\partial \theta_2} = 1\cdot(-4)\cdot 1\cdot 1\cdot 5 = -20$$

1. $\partial L/\partial \hat{y} = 2(y-\hat{y})(-1) = -4$
2. $\partial L/\partial y = 2(y-\hat{y})(1) = -4$
3. $\partial \hat{y}/\partial \theta_0 = 1 = 1$
4. $\partial \hat{y}/\partial c = 1 = 1$
5. $\partial c/\partial a = 1 = 1$
6. $\partial c/\partial b = 1 = 1$
7. $\partial a/\partial \theta_1 = x_1 = 4$
8. $\partial a/\partial x_1 = \theta_1 = 2$
9. $\partial b/\partial \theta_2 = x_2 = 5$
10. $\partial b/\partial x_2 = \theta_2 = 3$

$\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \frac{\partial L}{\partial y}$ can be computed but not needed