Machine Learning

Neural Networks

Fabio Vandin

December 5, 2017
Hypothesis Set of a NN

Architecture of a NN: \((V, E, \sigma)\)

Once we specify the architecture and \(w\), we obtain a function:

\[
h_{V,E,\sigma,w} : \mathbb{R}^{|V_0| - 1} \rightarrow \mathbb{R}^{|V_T|}
\]

The hypothesis class of a neural network is defined by fixing its architecture:

\[
\mathcal{H}_{V,E,\sigma} = \{ h_{V,E,\sigma,w} : w \text{ is a mapping from } E \text{ to } \mathbb{R} \}
\]

**Question**: what type of functions can be implemented using a neural network?
Expressiveness of NN

Proposition
For every $d$, there exists a graph $(V, E)$ of depth 2 such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{-1, 1\}^d$ to $\{-1, 1\}$.

NN can implement every boolean function!

Unfortunately the graph $(V, E)$ is very big...

Proposition
For every $d$, let $s(d)$ be the minimal integer such that there exists a graph $(V, E)$ with $|V| = s(d)$ such that $\mathcal{H}_{V,E,\text{sign}}$ contains all functions from $\{-1, 1\}^d$ to $\{-1, 1\}$. Then $s(d)$ is an exponential function of $d$.

Note: similar result for $\sigma = \text{sigmoid}$
Proposition

For every fixed $\varepsilon > 0$ and every Lipschitz function $f : [-1, 1]^d \rightarrow [-1, 1]$ it is possible to construct a neural network such that for every input $x \in [-1, 1]^d$ the output of the neural network is in $[f(x) - \varepsilon, f(x) + \varepsilon]$.

Note: first result proved by Cybenko (1989) for sigmoid activation function, requires only 1 hidden layer!

NNs are universal approximators!

But again...

Proposition

Fix some $\varepsilon \in (0, 1)$. For every $d$, let $s(d)$ be the minimal integer such that there exists a graph $(V, E)$ with $|V| = s(d)$ such that $\mathcal{H}_{V,E,\sigma}$, with $\sigma = \text{sigmoid}$, can approximate, with precision $\varepsilon$, every 1-Lipschitz function $f : [-1, 1]^d \rightarrow [-1, 1]$. Then $s(d)$ is exponential in $d$. 
2 layer neural networks (with $\sigma = \text{sign}$) can express intersection of halfspaces
3 layer neural networks (with $\sigma = \text{sign}$) can express unions of intersection of halfspaces
An Extremely Powerful Hypothesis Class...

- expert system
  - use prior knowledge to construct $\phi(x)$ and learn $\langle w, \phi(x) \rangle$
- deep networks
  - less prior knowledge
    - more data
  - No Free Lunch
Sample Complexity of NNs

How much data is needed to learn with NNs?

Proposition

The VC dimension of $\mathcal{H}_{V,E,\text{sign}} = O(|E| \log |E|)$

Different $\sigma$?

Proposition

Let $\sigma$ be the sigmoid function. The VC dimension of $\mathcal{H}_{V,E,\sigma}$ is:

- $\Omega(|E|^2)$
- $O(|V|^2|E|^2)$

$\Rightarrow$ large NNs require a lot of data!

**Question**: assume we have a lot of data, can we find the best hypothesis?
Runtime of Learning NNs

**Informally:** applying the ERM rule with respect to $\mathcal{H}_{V,E,\text{sign}}$ is *computationally difficult*, even for small NN...

**Proposition**

Let $k \geq 3$. For every $d$, let $(V, E)$ be a layered graph with $d$ input nodes, $k + 1$ nodes at the (only) hidden layer, where one of them is the constant neuron, and a single output node. Then, it is NP-hard to implement the ERM rule with respect to $\mathcal{H}_{V,E,\text{sign}}$.

Well maybe the above is only for very specific cases...

- instead of ERM rule, find $h$ close to ERM? *Computationally infeasible!* (probably)
- other activation functions (e.g., sigmoid)? *Computationally infeasible!* (probably)
- smart embedding in larger network? *Computationally infeasible!* (probably)
So? *Heuristic* for training NNs ➞ SGD algorithm is used: gives good results in practice!
Consider layer $t$, $0 < t < T$:

- let $d^{(t)} + 1$ the number of nodes:
  - constant node $1$
  - nodes for (hidden) variables: $v_{t,1}, \ldots, v_{t,d^{(t)}}$

- arc from $v_{t-1,i}$ to $v_{t,j}$ has weight $w^{(t)}_{ij}$

Let

$$v^{(t)} = \left(1, v_{t,1}, \ldots, v_{t,d^{(t)}} \right)^T$$

$$w^{(t)}_j = \left(w^{(t)}_{0j}, w^{(t)}_{1j}, \ldots, w^{(t)}_{d^{(t-1)}j} \right)^T$$

Then

$$v_{t,j} = \sigma \left( \langle w^{(t)}_j, v^{(t-1)} \rangle \right)$$
Note:

\[ \mathbf{v}(t) = \begin{bmatrix} 1 \\ v_{t,1} \\ \vdots \\ v_{t,d(t)} \end{bmatrix} = \begin{bmatrix} 1 \\ \sigma \left( \langle w_1(t), \mathbf{v}(t-1) \rangle \right) \\ \vdots \\ \sigma \left( \langle w_{d(t)}(t), \mathbf{v}(t-1) \rangle \right) \end{bmatrix} \]

Let

\[ a_{t,j} := \langle w_j(t), \mathbf{v}(t-1) \rangle \]

and

\[ a(t) = \begin{bmatrix} a_{t,1} \\ \vdots \\ a_{t,d(t)} \end{bmatrix}, \quad \sigma \left( a(t) \right) = \begin{bmatrix} \sigma \left( a_{t,1} \right) \\ \vdots \\ \sigma \left( a_{t,d(t)} \right) \end{bmatrix} \]

Then

\[ \mathbf{v}(t) = \begin{bmatrix} 1 \\ \sigma \left( a(t) \right) \end{bmatrix} \]
Let

\[
    w^{(t)} = \begin{bmatrix}
        w_0^{(t)} & w_0^{(t)} & \cdots & w_0^{(t)} \\
        w_1^{(t)} & w_1^{(t)} & \cdots & w_1^{(t)} \\
        \vdots & \vdots & \ddots & \vdots \\
        w_{d(t-1)}^{(t)} & w_{d(t-1)}^{(t)} & \cdots & w_{d(t-1)}^{(t)}
    \end{bmatrix}
\]

\(w^{(t)}\) describes the weights of edges from layer \(t - 1\) to layer \(t\)

Then

\[
    a^{(t)} = \left( w^{(t)} \right)^T v^{(t-1)}
\]
Warm-Up: Forward Propagation Algorithm

Input: $\mathbf{x} = (x_1, \ldots, x_d)^T$; NN with 1 output node
Output: prediction $y$ of NN;

$v^{(0)} \leftarrow (1, x_1, \ldots, x_d)^T$;
for $t \leftarrow 1$ to $T$ do
  $a(t) \leftarrow (w(t))^T v^{(t-1)}$;
  $v^{(t)} \leftarrow \left(1, \sigma (a(t))^T\right)^T$;
$y \leftarrow v^{(T)}$;
return $y$;
Fitting NN parameters

How do we compute the weights $w_{ij}^{(t)}$?

ERM: given training data $(x_1, y_1), \ldots, (x_m, y_m)$ pick $w_{ij}^{(t)}, \forall i, j, t$

(definition a specific model $h$) minimizing the training error:

$$L_S(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h, (x_i, y_i))$$

How?

Not easy!
Fitting NN parameters (2)

We use SGD seeing $L_S(h)$ as a function of $w(t), \forall 1 \leq t \leq T$:

SGD Update rule:

$$w^{(t)} \leftarrow w^{(t)} - \eta \nabla L_S(w^{(t)})$$

where $L_S(w^{(t)})$ is the gradient of $L_S$ (and $\eta$ is the learning parameter). To compute it we need $\forall t, 1 \leq t \leq T$:

$$\frac{\partial L_S}{\partial w^{(t)}} = \frac{\partial}{\partial w^{(t)}} \left( \frac{1}{m} \sum_{i=1}^{m} \ell(h, (x_i, y_i)) \right) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L(h, (x_i, y_i))}{\partial w^{(t)}}$$

$\Rightarrow$ need $\frac{\partial L}{\partial w^{(t)}}$
Fitting NN parameters (3)

Definition: Sensitivity vector for layer $t$

$$
\delta(t) = \frac{\partial L}{\partial a(t)} = \left[ \begin{array}{c} \frac{\partial L}{\partial a_{t,1}} \\ \vdots \\ \frac{\partial L}{\partial a_{t,d(t)}} \end{array} \right]
$$

$\delta(t)$ quantifies how the training error changes with $a(t)$ (the inputs to the $t$ layer - before the nonlinear transformation)
Consider a weight $w_{ij}^{(t)}$: if it changes it only changes $a_{t,j}$ therefore by chain rule we have

$$\frac{\partial L}{\partial w_{ij}^{(t)}} = \frac{\partial L}{\partial a_{t,j}} \cdot \frac{\partial a_{t,j}}{\partial w_{ij}^{(t)}}$$

$$= \delta(t) \cdot \frac{\partial}{\partial w_{ij}^{(t)}} \left( \sum_{k=0}^{d^{(t-1)}} w_{kj}^{(t)} v_{t-1,k} \right)$$

$$= \delta(t) \cdot v_{t-1,i}$$

Therefore to compute the gradient we only need $\delta(t) = \frac{\partial L}{\partial a^{(t)}} \forall t$. How can we compute it?
Fitting NN parameters (5)

Since $L$ depends from $a_{t,j}$ only through $v_{t,j}$, then from chain rule:

$$
\delta_{j}^{(t)} = \frac{\partial L}{\partial a_{t,j}} = \frac{\partial L}{\partial v_{t,j}} \cdot \frac{\partial v_{t,j}}{\partial a_{t,j}} = \frac{\partial L}{\partial v_{t,j}} \cdot \sigma'(a_{t,j})
$$

(the last equality derives from the definition of $v_{t,j}$)
Fitting NN parameters (6)

Consider \( \frac{\partial L}{\partial v_{t,j}} \): we need to understand how loss \( L \) changes due to changes in \( v_{t,j} \)

- change in \( v^{(t)} \) affects only \( a^{(t+1)} \) (and then \( L \))
- changes in \( v_{t,j} \) can affect every \( a_{t+1,k} \)

⇒ sum chain rule contributions

Then

\[
\frac{\partial L}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} \frac{\partial a_{t+1,k}}{\partial v_{t,j}} \cdot \frac{\partial L}{\partial a_{t+1,k}} = \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \cdot \delta_k^{(t+1)}
\]
Putting everything together:

\[ \delta_j^{(t)} = \sigma'(a_{t,j}) \cdot \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)} \]

Notes:

- \( \sigma'(a_{t,j}) \) depends on the function \( \sigma \) chosen
- To compute \( \delta_j^{(t)} \) need \( \delta_k^{(t+1)} \), \( 1 \leq k \leq d^{(t+1)} \)  
  \[ \Rightarrow \text{backpropagation algorithm} \]
- To start: need \( \delta^{(L)} = \frac{\partial L}{\partial a^{(L)}} \) (sensitivity of final layer): depends on the loss \( L \) used
Algorithm to compute sensitivities $\delta(t), \forall t$, for a given data point $(x_i, y_i)$.

**Input:** data point $(x_i, y_i)$, NN (with weights $w_{ij}^{(t)}$, for $1 \leq t \leq T$)

**Output:** $\delta(t)$ for $t = 1, \ldots, T$
compute $a^{(t)}$ and $v^{(t)}$ for $t = 1, \ldots, T$;

$\delta^{(L)} \leftarrow \frac{\partial L}{\partial a^{(L)}}$;

for $t = T - 1$ downto 1 do

$\delta^{(\ell)} \leftarrow \sigma'(a_{t,j}) \cdot \sum_{k=1}^{d^{(\ell+1)}} w_{jk}^{(t+1)} \delta^{(t+1)}_k$ for all $j = 1, \ldots, d^{(t)}$;

return $\delta^{(1)}, \ldots, \delta^{(T)}$;
This is the final backpropagation algorithm, based on SGD, to train a NN

**Input:** training data \((x_1, y_1), \ldots, (x_m, y_m)\), NN (no weights \(w^{(t)}_{ij}\))

**Output:** NN with weights \(w^{(t)}_{ij}\)

initialize \(w^{(t)}_{ij}\) for all \(i, j, t\);

**for** \(t \leftarrow 0, 1, 2, \ldots\) **do** /* until convergence */

- pick \((x_k, y_k)\) at random from training data;
- compute \(v^{(t)}_{t,j}\) for all \(j, t\); /* forward propagation */
- compute \(\delta^{(t)}_j\) for all \(j, t\); /* backward propagation */
- \(w^{(t)}_{ij} \leftarrow w^{(t)}_{ij} - \eta v^{(t-1)}_{i} \delta^{(t)}_j\) for all \(i, j, t\); /* update weights */

**if** converged **then** return \(w^{(t)}_{ij}\) for all \(i, j, t\);
Notes on Backpropagation Algorithm

- preprocessing: all inputs are normalized and centered
- initialization of $w_{ij}^{(t)}$?
  Random values around 0 - regime where model is $\approx$ linear
  - $w_{ij}^{(t)} \sim U(-0.7, 0.7)$ (uniform distribution)
  - $w_{ij}^{(t)} \sim N(0, \sigma^2)$ with small $\sigma^2$
- when to stop?
  Usually combination of:
    - “small” (training) error;
    - “small” marginal improvement in error;
    - upper bound on number of iterations
- $L_S(h)$ usually has multiple local minima
  $\Rightarrow$ run stochastic gradient descent for different (random) initial weights
Regularized NN

Instead of training a NN by minimizing $L_S(h)$, find $h$ that minimizes:

$$L_S(h) + \frac{\lambda}{m} \sum_{i,j,t} (w_{ij}^{(t)})^2$$

where $\lambda = \text{regularization parameter}$

How do we find $h$? SGD.

This is called \textit{squared weight decay regularizer}

Other regularizations are possible.