Learning Error Vs. Generalization Error

["Train faster, generalize better", Hardt, 2016]

Consider a regression problem. We are given a sample

\[ S = \{(x_1, y_1), \ldots, (x_n, y_n)\} \]

\[ y_i = f(x_i), \quad x_1, \ldots, x_n \text{ i.i.d. } \sim p_x \]

\( f \) is the function that we want to learn.

Our estimate is the function \( \hat{f}_S \) and our performance metric is L2 Loss: Then, the expected value is

\[ R = \mathbb{E}_{x \sim p_x} \left[ \left| f(x) - \hat{f}_S(x) \right|^2 \right] \]

Since we do not know the true \( f \), we cannot compute the risk.

However, we can compute the empirical risk

\[ R_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} \left| f(x_i) - \hat{f}_S(x_i) \right|^2 \]

\[ R \leq |R - R_{\text{emp}}| + R_{\text{emp}} \]

(expected) generalization error (expected) learning error

The aim of the paper by Hardt is to provide upper bounds on the generalization error.
Formally, \( \text{Stability} \Rightarrow \text{Generalization} \) means the following.

**Definition [\( \varepsilon \)-uniform stability]** We say that the estimate \( \hat{f} \) is \( \varepsilon \)-uniformly stable if for all data sets \( S, S' \) s.t. \( S \) and \( S' \) differ in at most one example, we have

\[
\sup_x | \hat{f}_{S'}(x) - \hat{f}_S(x) | \leq \varepsilon.
\]

**Theorem ["Stability and generalization", Elisseeff, 2002]** Let \( \hat{f} \) be \( \varepsilon \)-uniformly stable. Then,

\[
| E(S) - \text{Risk}(\hat{f}) | \leq \varepsilon.
\]

If an algorithm is uniformly stable, then its generalization error is small.
Backpropagation Algorithm

Toy Example

- all remaining weights are 1
- all biases are 0
- activation function at all hidden nodes is \( \phi(\cdot) \)
- activation function at the output is \( \phi(x) = ax \)
- cost function at the output is \( C(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2 \)

The most natural thing to do is: "Forward propagation Algorithm".

It is a message-passing algorithm where messages flow on the edges of the network from input to output.

1) Propagate input and store derivatives at operating point.

Suppose we want to compute \( \frac{\partial C}{\partial w_1} \).

2) Initialisation: initialise the message on the edge corresponding to \( w \) with its input. All edges of the same layer are initialised to 0.

3) Message passing rule: The output of each node is the sum of the inputs times the value stored in the node. The output is propagated to all outgoing edges.

Stop when reach the output:

\[
\frac{\partial C}{\partial w_1} = (\hat{y} - y) \left[ \phi'(z_1) \phi'(z_2) + \phi'(z_3) \phi'(z_4) \right] x
\]

So far so good... Why not then backpropagation?

Double check

\[
\frac{\partial C}{\partial w} = \frac{1}{2} (\hat{y} - y)^2 \left[ \phi''(z_1) + \phi''(z_2) \right] x
\]
The problem is that if now you want to compute \( \frac{\partial C}{\partial w_i} \) you have to repeat (2) and (3) again.

We would like to obtain all derivatives in 1 shot.

**Back propagation**

Message passing algorithm where messages flow on the edges of the net from output to input:

\[
\begin{align*}
\tilde{y} &= y'_{(n)} \left( \psi'(h) + \psi'(l) \right) \\
\hat{y} &= y_{(n)} \left( \psi'(h) + \psi'(l) \right)
\end{align*}
\]

1) Same as before

2) Initialization: start from output and initialize edges to the value in the output node

3) Message passing rule: The input of each node is the sum of the outputs into the value stored in the node. The input is propagated to all incoming edges.

Stop when reach the input of the edge you are interested in.

**Results**

\[
\begin{align*}
\frac{\partial C}{\partial w_1} &= (\tilde{y} - y) \psi'(h) \left( \psi'(h) + \psi'(l) \right) \\
\frac{\partial C}{\partial w_2} &= (\tilde{y} - y) \psi'(l) \left( \psi'(h) + \psi'(l) \right)
\end{align*}
\]
Some Practical Tricks

1. Preprocess your data

2. 1) O-mean and 1-variance
2) Dimensionality reduction (e.g., PCA)

Principal Component Analysis

Compute empirical covariance matrix of the data

\[ S = \frac{1}{n} \sum_{i=1}^{n} x(i) x^T(i) \]

\[ \in \mathbb{R}^{d \times d}, \quad \in \mathbb{R}^{d \times d} \]

Compute eigenvalues and eigenvectors of S

1) Project data into d' directions (eigenvectors) associated to the largest eigenvalues. Usually we want to retain 75% - 90% of energy. i.e., d' is chosen so the sum of the d' largest eigenvalues is \( \approx 75\% - 90\% \) of the sum of all eigenvalues.

Different cost function (cross entropy instead of MSE)

Advantage: avoids learning slowdown.

\[ C(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} y_i(x) \log \hat{y}_i(x) + (1 - y_i(x)) \log (1 - \hat{y}_i(x)) \]

KL distance from \( y(x) \) to \( \hat{y}(x) \)
So far \( y_j := b_j = \sum_{\ell=1}^{L-1} W^\ell_{jk} h^\ell_{(\ell-1)} + b^\ell_j \) ...

\[ h^L_j = \phi(h^L_j) \]

Softmax layer means that
\[ h^L_j = \frac{e^{z^L_j}}{\sum_k e^{z^L_k}} \]

Now \( \sum_j h^L_j = 1 \) and they can be interpreted as probabilities.

Typically softmax is associated with log-likelihood cost function
\[ C = -\text{ln} \ h^L_j \]

Perfect classification means \( h^L_j = 1 \) and \( h^L_k = 0 \) for \( i \neq j \).

So \( C = 0 \) and it makes sense.

With log-likelihood cost function also no learning slowdown (as with cross-entropy cost function + sigmoid layer).

Split training set into actual training + validation set.

Use actual training to learn the weights and validation to test the results. This is useful to avoid overfitting and to choose the hyperparameters of the network (step size of gradient descent, # hidden nodes, # epochs for which one has to learn).
Regu Caractahan.

L2 regularized - cross entropy

\[ C = \frac{-1}{n} \sum_{i,x} (y_i(x) \log \hat{y}_i(x) + (1-y_i(x)) \log (1-\hat{y}_i(x))) \]

\[ + \frac{\lambda}{2n} \sum_{w} w^2 \]

regularization parameter

L1 regularization

\[ C = C_0 + \frac{\lambda}{n} \sum_{w} |w| \]

→ Artificial expansion of training data: Pick all MNIST images and

(i) rotate them by 5°, 10°, 15°, -5°, -10°, -15°

(ii) translate them by 1 pixel on tx, ex, up, down

→ Weight initialization

Initialize the weights that are given as input to some neuron with \( N(0, \frac{1}{\text{nin}}) \) where \( \text{nin} \) is the number of input weights of the neuron.

In this way, the neuron will have an input with variance 1 and will not immediately saturate with good probability.

\[ \sum_{k=1}^{\text{nin}} w_{xk} \sim N(0,1) \]

\[ \sim N(0, \frac{1}{\text{nin}}) \]