

# Multilayer Perceptron & How to Train Deep Learning Models

Cédric Poutré, M. Sc, ASA

## Supervised Learning and Neural Networks: Overview

Given a sample  $\mathcal{S}$  of  $N$  independent observations  $\mathbf{x}$  and their associated label  $y$  identically generated by an unknown distribution  $\mathcal{D}$ :

$$\mathcal{S} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)\},$$

where

$$\mathbf{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R}, \text{ s.t. } (\mathbf{x}_i, y_i) \stackrel{iid}{\sim} \mathcal{D}, 1 \leq i \leq N,$$

we are interested in finding a function (a neural network)

$$f_{\theta} : \mathbb{R}^d \mapsto \mathbb{R}$$

that approximates the target  $y$  from the input  $\mathbf{x}$ , using the parameters  $\theta$ , i.e.  $f_{\theta}(\mathbf{x}_i) \approx y_i, \forall i$ .

## Perceptron

The perceptron is the basic unit responsible for bigger (more complex) neural networks. We define the pre-activation:

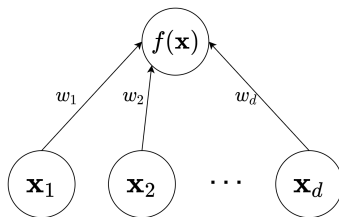
$$a(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b,$$

and the activation (the output in this case):

$$f(\mathbf{x}) = (g \circ a)(\mathbf{x}) = g(a(\mathbf{x})),$$

where

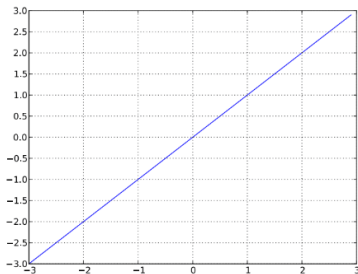
- $\mathbf{w}$  is weight vector,
- $b$  is the bias,
- $\theta = \{\mathbf{w}, b\}$
- $g : \mathbb{R} \rightarrow \mathbb{R}$  is the activation function.



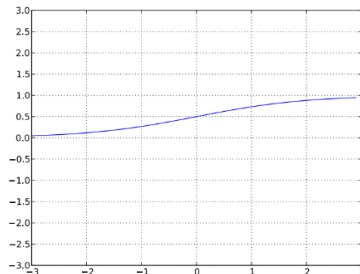
# Activation Functions

There exist multiple activation functions, each with its own properties and uses:

- Linear:  $g(a) = a$ ,
- Sigmoid:  $g(a) = \sigma(a) = \frac{1}{1 + e^{-a}}$



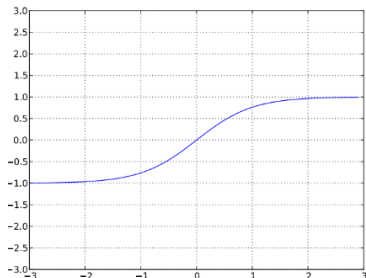
Linear



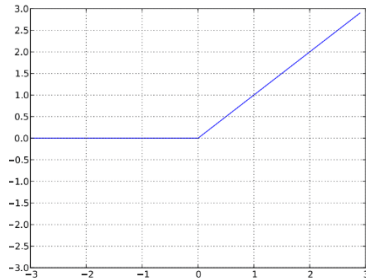
Sigmoid

## Activation Functions

- Tanh:  $g(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$ , mostly used in Recurrent Neural Networks,
- Rectified Linear Unit (ReLU):  $g(a) = \text{ReLU}(a) = \max(0, a)$ , heavily used everywhere else,
- Etc., etc., etc.



Tanh



ReLU

## Logistic Regression

Notice that a perceptron with a sigmoid activation function results in the logistic regression:

$$f(\mathbf{x}) = \sigma(a(\mathbf{x})) = \frac{1}{1 + e^{-a(\mathbf{x})}} = \frac{1}{1 + e^{-(\langle \mathbf{w}, \mathbf{x} \rangle + b)}}.$$

We see that  $f(\mathbf{x}) \in [0, 1]$ , and it is interpreted as an estimator of  $\mathbb{P}(y = 1 | \mathbf{x})$  in the binary classification task where  $y \in \{0, 1\}$ .

The perceptron with a sigmoid activation function is a probabilistic binary classifier!

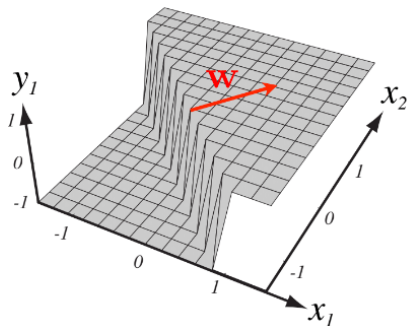
## Logistic Regression

The perceptron remains a linear classifier, even though the activation function isn't linear. Indeed, its decision function is defined as

$$h(\mathbf{x}) = \begin{cases} 1, & \text{if } f(\mathbf{x}) \geq 0.5 \\ 0, & \text{otherwise} \end{cases} \iff h(\mathbf{x}) = \begin{cases} 1, & \text{if } a(\mathbf{x}) \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

and  $a(\mathbf{x})$  is linear by definition. We are stuck with a hyperplane  
 $\implies$  only good for linearly separable  $\mathcal{S}$ .

# Logistic Regression



**Figure:** Decision Function for Logistic Regression (taken from Pascal Vincent's IFT6390 lectures at MILA)

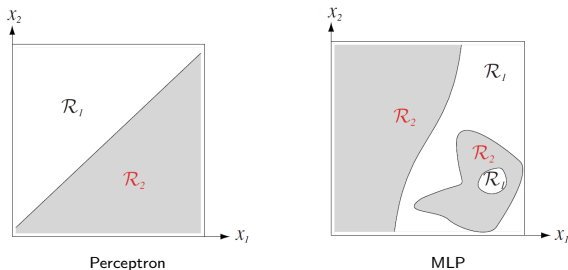



## Perceptron v.s. Multilayer Perceptron

To go beyond a linear decision function, we apply a second activation function to the inputs:

$$f(\mathbf{x}) = f_2(f_1(\mathbf{x})) = g_2(a_2(g_1(a_1(\mathbf{x}))))).$$

Why not go *deeper* with a third activation function, a fourth, etc.? This is the idea behind the Multilayer Perceptron (MLP).



**Figure:** Decision Regions for Perceptron and MLP (taken from Pascal  Fin ML Vincent's IFT6390 lectures at MILA)

# Multilayer Perceptron (MLP)

For  $L$  the number of hidden layers, here are the formulas at hidden layer  $k$  with  $d^{(k)}$  hidden units:

- Layer pre-activation ( $1 \leq k \leq L + 1$ ):

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x}) + \mathbf{b}^{(k)}$$

where  $\mathbf{h}^{(0)} = \mathbf{x}$  and  $d^{(0)} = d$ .

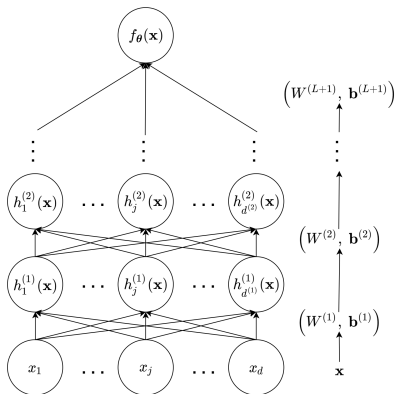
- Layer activation ( $1 \leq k \leq L$ ):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}^{(k)}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- Output layer ( $k = L + 1$ ):

$$f_{\theta}(\mathbf{x}) = o(\mathbf{a}^{(L+1)}(\mathbf{x})).$$

where  $o : \mathbb{R}^{d^{(L+1)}} \rightarrow \mathbb{R}^K$  is the output function.



# Multilayer Perceptron (MLP)

An easier way of seeing the MLP's formulas:

$$\mathbf{h}^{(0)} = \mathbf{x}$$

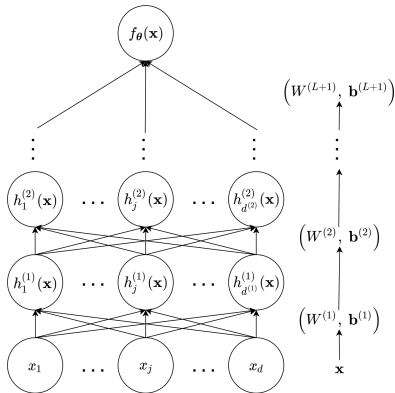
$$\mathbf{h}^{(1)} = g^{(1)}(\mathbf{W}^{(1)}\mathbf{h}^{(0)} + \mathbf{b}^{(1)})$$

$$\mathbf{h}^{(2)} = g^{(2)}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

$$\vdots$$

$$\mathbf{h}^{(L)} = g^{(L)}(\mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)})$$

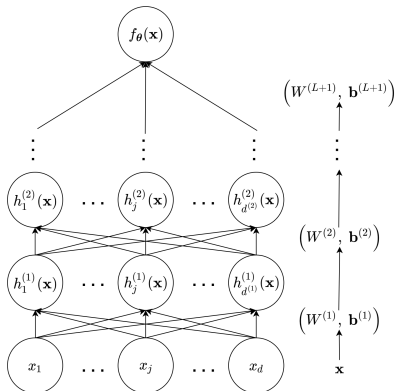
$$f_{\theta}(\mathbf{x}) = o(\mathbf{W}^{(L+1)}\mathbf{h}^{(L)} + \mathbf{b}^{(L+1)})$$



# Multilayer Perceptron (MLP)

The parameters  $\theta$  of a MLP with  $L$  hidden layers with respective number of hidden units  $d^{(k)}$ ,  $1 \leq k \leq L$ , are:

- The  $L + 1$  weight matrices:  $\left\{ \mathbf{W}^{(k)} \in \mathbb{R}^{d^{(k)} \times d^{(k-1)}} \right\}_{k=1}^L$  and  $\mathbf{W}^{(L+1)} \in \mathbb{R}^{K \times d^{(L)}}$ ,
- The  $L + 1$  bias vectors:  $\left\{ \mathbf{b}^k \in \mathbb{R}^{d^{(k)}} \right\}_{k=1}^L$  and  $\mathbf{b}^{(L+1)} \in \mathbb{R}^K$ .



Totaling  $\sum_{k=1}^L \left( d^{(k)} \times (d^{(k-1)} + 1) \right) + K(d^{(L)} + 1)$  parameters.

## Output Functions of MLPs

The output function  $o : \mathbb{R}^{d^{(L+1)}} \rightarrow \mathbb{R}^K$  strictly depends on the problem you're working on (characteristics of target  $y$ ). Here are usual examples:

- Regression ( $K = 1$ ,  $y \in \mathbb{R}$ ): use linear function

$$o(a) = a,$$

e.g. predict a stock's daily return.

- Binary classification ( $K = 1$ ,  $y \in \{0, 1\}$ ): use sigmoid function

$$o(a) = \sigma(a) = \frac{1}{1 + e^{-a}},$$

e.g. predict a stock's daily direction.

## Output Functions of MLPs

- Multi-class classification ( $K = C$ ,  $y \in \{1, 2, \dots, C\}$ ): use softmax function

$$o(\mathbf{a}) = \textit{softmax}(\mathbf{a}) = \left[ \frac{e^{a_1}}{\sum_c e^{a_c}} \quad \frac{e^{a_2}}{\sum_c e^{a_c}} \quad \cdots \quad \frac{e^{a_C}}{\sum_c e^{a_c}} \right]^T$$

Output vector sums to 1 and each of its component is strictly positive  $\implies$  components are the probability of the input  $\mathbf{x}$  to be of class  $c$ .

The class with the highest associated probability is the one predicted by the MLP.

## Output Functions of MLPs

Other examples applied to finance:

- Regression on real positives ( $K = 1, y \in \mathbb{R}_{>0}$ ): use ReLU function

$$o(a) = \text{ReLU}(a) = \max(0, a)$$

e.g. predict the price of a financial option.

- Regression on  $[-1, 1]$  ( $K = 1, y \in [-1, 1]$ ): use tanh function

$$o(a) = \tanh(a)$$

e.g. predict the buy/sell signal of a stock.

- There's basically no limits...

## Hyperparameters of a MLP

The parameters that describe the structure of a MLP and how it is trained are called hyperparameters. They are parameters that are set before training, and they are not trained. In a MLP, we have:

- The number of hidden layers  $L$ ,
- The number of hidden units in every layer  $\left\{d^{(k)}\right\}_{k=1}^L$ ,
- The activation functions of the hidden layers  $\left\{g^{(k)}\right\}_{k=1}^L$ ,
- The weights initialization,
- The parameters related to the regularization and training of the MLP (we will see that soon).



## MLP: A Universal Approximator

- Hornik (1991)<sup>1</sup>: "A single hidden layer neural network with a linear output can approximate any continuous function arbitrarily well, given enough hidden units."
- It is more computationally efficient to use deeper (and not wider) MLP and still achieve same level of accuracy/predictive power with a lesser amount of units.
- In practice, there's no guarantee to learn the true parameters (global optimum).
- The MLP is found almost everywhere (in Convolutional Neural Networks, Autoencoders, Variational Autoencoders, etc.).

---

<sup>1</sup>Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks*, Vol. 4, pp. 251-257

## Cost Functions

Let's define the predictions of a MLP (or any learning algorithm) as  $f_{\theta}(\mathbf{x}_i) = \hat{y}_i$ .

Given the true targets  $y = \{y_i\}_{i=1}^N$ , and the predictions  $\hat{y} = \{\hat{y}\}_{i=1}^N$ , we need to know how well the algorithm performs on our data in order to train (fit) it. This is the loss function's role.

Usually written as  $L(y, \hat{y})$ , this function depends on the task at hand, and on the architecture of the model that is trained.

## Cost Functions

Here are the usual cost functions used in machine learning.

- Classification;  $f_{\theta} : \mathbb{R}^d \rightarrow \{0, 1, \dots, C - 1\}$ ,

$$L(y, \hat{y}) = \mathbb{I}_{\{\hat{y} \neq y\}}$$

- Regression;  $f_{\theta} : \mathbb{R}^d \rightarrow \mathbb{R}$ ,

$$L(y, \hat{y}) = \|\hat{y} - y\|_2^2$$

- Density estimation;  $f_{\theta} : \mathbb{R}^d \rightarrow \mathbb{R}_{>0}$ ,

$$L(\hat{y}) = -\log(\hat{y})$$

## Cost Functions

Sometimes, it is hard to optimize directly the cost function that we really want (e.g. classification loss), so we use a *surrogate* loss function. Here are the surrogate loss functions used for classification tasks with neural networks:

- Binary classification task ( $C = 2$ ):

Binary cross-entropy:  $L(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$

- Multi-class classification ( $C > 2$ ):

$$\hat{\mathbf{y}} = [\mathbb{P}(y = 1|\mathbf{x}) \ \mathbb{P}(y = 2|\mathbf{x}) \ \dots \ \mathbb{P}(y = C|\mathbf{x})]^T$$

$$L(y, \hat{\mathbf{y}}) = -\log(\hat{\mathbf{y}}_y)$$

## Expected Risk and Empirical Risk

Remember that  $(\mathbf{x}_i, y_i) \stackrel{iid}{\sim} \mathcal{D}$ ,  $1 \leq i \leq N$ .

We want to maximize the effectiveness of the network, i.e. minimize the cost function on the data set  $\mathcal{S}$ .

- Generalization error (Expected risk):

$$R(f_\theta) = \mathbb{E}_{\mathcal{D}} [L(y, \hat{y})]$$

- Empirical risk:

$$\hat{R}(f_\theta, \mathcal{S}) = \frac{1}{N} \sum_{(\mathbf{x}, y) \in \mathcal{S}} L(y, \hat{y}).$$

## Empirical Risk Minimization

Since we do not know  $\mathcal{D}$ , we cannot compute the true generalization error of the model, but given a big enough  $\mathcal{S}$  we have

$$R(f_\theta) \approx \hat{R}(f_\theta, \mathcal{S}),$$

since for a large  $N$  we have

$$\mathbb{E}_{\mathcal{D}}[L(y, \hat{y})] \approx \frac{1}{N} \sum_{(\mathbf{x}, y) \in \mathcal{S}} L(y, \hat{y}).$$

This is the quantity that we want (and can) minimize in order to train the neural network, i.e.

$$\theta^* = \arg \min_{\theta} \hat{R}(f_\theta, \mathcal{S}).$$

## Stochastic Gradient Descent (SGD)

In order to find

$$\theta^* = \arg \min_{\theta} \hat{R}(f_{\theta}, \mathcal{S}) = \arg \min_{\theta} \frac{1}{N} \sum_{(\mathbf{x}, y) \in \mathcal{S}} L(y, f_{\theta}(\mathbf{x}))$$

we use gradient descent, since there is no analytic solution to

$$\frac{\delta \hat{R}(f_{\theta^*}, \mathcal{S})}{\delta \theta} = \mathbf{0}.$$

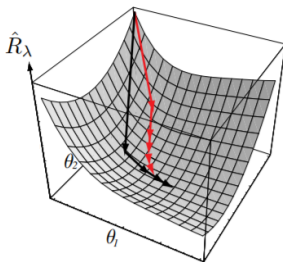


Figure: Gradient Descent on Total Loss Function (taken from Pascal Vincent's IFT6390 lectures at MILA)

## Stochastic Gradient Descent (SGD)

Set

$$J(\boldsymbol{\theta}) = \underbrace{\hat{R}_\lambda(f_\theta, \mathcal{S}) = \frac{1}{N} \sum_{(\mathbf{x}, y) \in \mathcal{S}} L(y, f_\theta(\mathbf{x}))}_{\text{Empirical Risk}} + \underbrace{\lambda \Omega(\boldsymbol{\theta})}_{\text{Regularization Term}},$$

the total cost function to optimize, where  $\lambda \in \mathbb{R}_{\geq 0}$  is the regularization parameter and  $\Omega(\boldsymbol{\theta})$  is the regularizer (more on that soon).

Gradient descent consists of iteratively going against

$$\nabla J(\boldsymbol{\theta}) \equiv \frac{\delta J(\boldsymbol{\theta})}{\delta \boldsymbol{\theta}}$$

in order to find (or approximate) a local minimum.



## Stochastic Gradient Descent (SGD)

Pseudo code of stochastic gradient descent, for a batch of size  $m$ :

1) Initialize  $\theta$

2) while (not stopping criterion) do:

    Select random batch of size  $m$  in  $\mathcal{S}$

    Update parameters:  $\theta \leftarrow \theta - \eta \nabla J(\theta)$

where  $\nabla J(\theta) = \frac{1}{m} \sum_{i=1}^m \nabla L(y_i, f_{\theta}(\mathbf{x}_i)) + \lambda \nabla \Omega(\theta)$ , and  $\eta \in \mathbb{R}_{>0}$  the learning rate.

The  $m$  data points are randomly selected in  $\mathcal{S}$ . When  $m = 1$ ; stochastic gradient descent, when  $m < N$ ; minibatch gradient descent (usually what is used), and  $m = N$ ; batch gradient descent.

Whenever we have used all  $N$  data points for training, we say that we went through an *epoch*. Training a neural network takes multiple epochs.

# Gradient Backpropagation

How do we compute  $\nabla J(\theta)$ ? With backward propagation of the gradient.

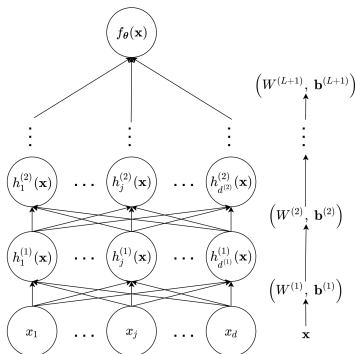


Figure: Forward Propagation

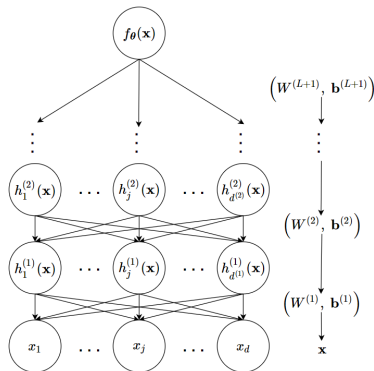


Figure: Backward Propagation (Backprop)

# Gradient Backpropagation

---

**Algorithm 6.4** Backward computation for the deep neural network of algorithm 6.3, which uses, in addition to the input  $\mathbf{x}$ , a target  $\mathbf{y}$ . This computation yields the gradients on the activations  $\mathbf{a}^{(k)}$  for each layer  $k$ , starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

---

After the forward computation, compute the gradient on the output layer:

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

**for**  $k = l, l-1, \dots, 1$  **do**

Convert the gradient on the layer's output into a gradient on the pre-nonlinearity activation (element-wise multiplication if  $f$  is element-wise):

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

Compute gradients on weights and biases (including the regularization term, where needed):

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

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

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

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

**end for**

---

# Gradient Backpropagation

Just remember that it is a direct application of the chain rule that starts at the output of the neural net, down to the input.

It gets extremely tedious and complicated very fast, especially with more advanced neural networks.

Luckily, every deep learning library computes  $\nabla J(\theta)$  for us; PyTorch, Tensorflow, Keras, etc., whatever the model's architecture.

## Neural Network Training Scheme

What really interests us is to have a neural network that can generalize well on unseen data (i.e. not "memorize" the data). Training and tuning the model on all of  $\mathcal{S}$  would directly lead to *overfitting*.

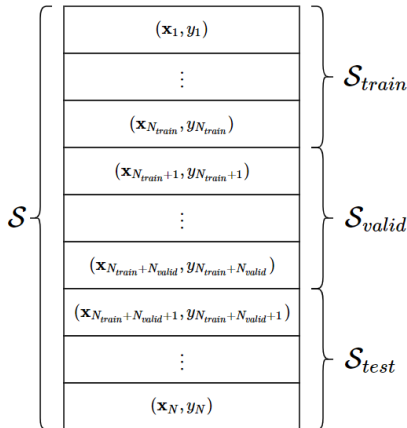
This is why we split randomly  $\mathcal{S}$  into three subsamples:

$$\mathcal{S} = \{\mathcal{S}_{train}, \mathcal{S}_{valid}, \mathcal{S}_{test}\}.$$

- $\mathcal{S}_{train}$ : used to find  $\theta^*$ ,
- $\mathcal{S}_{valid}$ : used to find the hyperparameters,
- $\mathcal{S}_{test}$ : used to compute the generalization error of the network.

General rule-of-thumb: 60% for  $\mathcal{S}_{train}$ , 20% for  $\mathcal{S}_{valid}$  and 20% for  $\mathcal{S}_{test}$ .

# General Meta-Algorithm for Training and Selecting a Neural Network



For each hyperparameter combination  $\mathcal{H}$  do:

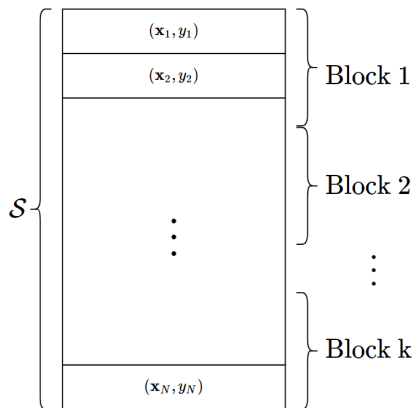
1) Train  $f_{\theta}$  on  $S_{train}$  with hyperparameters  $\mathcal{H}$  to find  $\theta^*$

2) Compute  $\hat{R}_{\lambda}(f_{\theta^*}, S_{valid})$

Return the neural network generated by the best combination  $\mathcal{H}^*$  with its optimal parameters, i.e.  $f_{\theta^*}^{\mathcal{H}^*}$ .

Compute the generalization error  $\hat{R}_{\lambda}(f_{\theta^*}^{\mathcal{H}^*}, S_{test})$

## k-Fold Cross-Validation



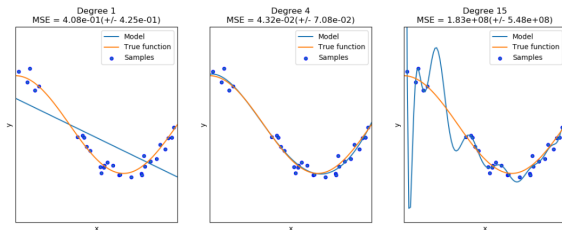
**Idea:** split  $\mathcal{S}$  into  $k$  even blocks, use  $k - 1$  blocks for  $\mathcal{S}_{\text{train}}$  and the other block for  $\mathcal{S}_{\text{valid}}$ . Repeat  $k$  times using a different block for  $\mathcal{S}_{\text{valid}}$ .

$\mathcal{S}_{\text{train}}$	$\mathcal{S}_{\text{valid}}$
$\mathcal{S} \setminus \text{Block 1}$	Block 1
$\mathcal{S} \setminus \text{Block 2}$	Block 2
$\vdots$	$\vdots$
$\mathcal{S} \setminus \text{Block } k$	Block k

Useful when there is not a lot of data. The generalization error is compiled on the  $k$   $\mathcal{S}_{\text{valid}}$ . When  $k = N$ , it is called *leave-one-out*.

# Underfitting and Overfitting

- Capacity: Flexibility of a model (e.g. number of parameters, complexity of model, etc.).
- Underfitting: the model cannot explain the data → needs more training or capacity.
- Overfitting: the model has poor generalization and overfits the training data → need to lower training time or capacity.



**Figure:** Example of Complexity on 1D Linear Regression Model



## Underfitting and Overfitting

In deep learning, *underfitting* or *overfitting* mainly refers to the number of iterations in the training of the network, which we visualize using the *learning curves* of the neural network.



**Figure:** Toy Example of Learning Curves (taken from Hugo Larochelle's IFT 725 Lectures)

A great way to increase generalization performance is to use regularization.

## Regularization

We want to ensure that the model will perform well on unseen data, i.e. have great generalization.

$$\text{Generalization Error} = \text{Bias}^2 + \text{Variance}$$

Regularization techniques add constraints into the machine learning model as to exploit prior knowledge on the model, or to force it to consider alternate hypothesis that explain the data.

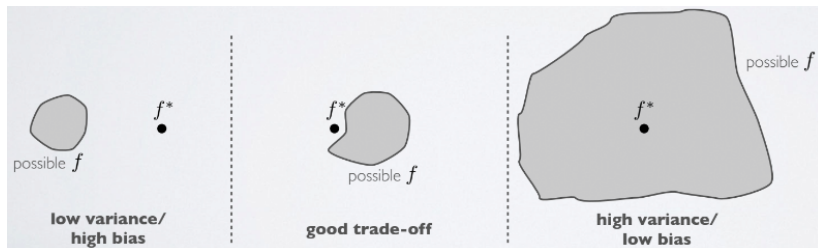
These techniques utilize the variance-bias trade-off, by reducing the variance of the model, and potentially increasing its bias. A successful regularization method will heavily decrease the variance while barely inducing any bias.

# Regularization

Structured risk minimization:

$$J(\theta) = \hat{R}_\lambda(f_\theta, \mathcal{S}) = \underbrace{\frac{1}{N} \sum_{(\mathbf{x}, y) \in \mathcal{S}} L(y, f_\theta(\mathbf{x}))}_{\text{Empirical Risk}} + \underbrace{\lambda \Omega(\theta)}_{\text{Regularization Term}}$$

$\lambda$  controls how much we regularize the network.



## $L^2$ Regularization

$$\begin{aligned}\Omega(\theta) &= \frac{1}{2} \sum_k \left\| \mathbf{w}^{(k)} \right\|_2^2 \\ &= \frac{1}{2} \sum_i \sum_j \sum_k W_{i,j}^{(k)2}\end{aligned}$$

- Is the most common version of *weight decay*.
- Used in ridge regression (a.k.a. Tikhonov regularization).
- Reduces variance and prevents over-fitting (higher complexity) by forcing weights towards 0.

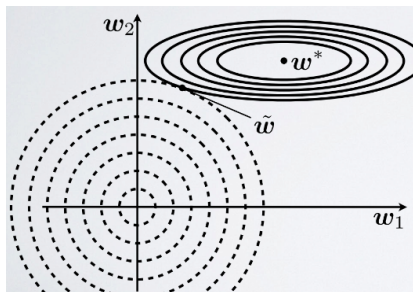
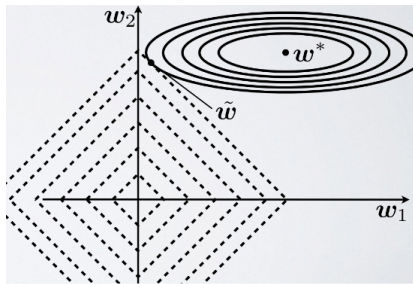


Figure: Empirical Risk Function (line) and  $L^2$  Regularizer (dotted)

## $L^1$ Regularization

$$\begin{aligned}\Omega(\theta) &= \sum_k \|\mathbf{w}^{(k)}\|_1 \\ &= \sum_i \sum_j \sum_k |w_{i,j}^{(k)}|\end{aligned}$$

- Is a version of *weight decay*.
- Used in LASSO regression.
- Penalizes larger weights and promotes sparsity (biggest difference from  $L^2$ ), which can be seen as a feature selection mechanism.



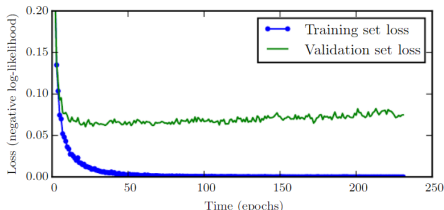
**Figure:** Empirical Risk Function (line) and  $L^1$  Regularizer (dotted)

Elastic-net uses a linear combination of  $L^1$  and  $L^2$  regularization

## Early Stopping

**Basic idea:** Stop training when the network has not increased its performance sufficiently on the validation set inside a predefined amount of iterations.

Limits the optimization procedure to a small volume of parameter space, it is shown to be equivalent to  $L^2$  regularization.



**Figure:** Example of Learning Curves with asymmetric U-shape Validation Loss Curve

# Early Stopping

---

**Algorithm 7.1** The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

---

Let  $n$  be the number of steps between evaluations.

Let  $p$  be the “patience,” the number of times to observe worsening validation set error before giving up.

Let  $\theta_o$  be the initial parameters.

$\theta \leftarrow \theta_o$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

**while**  $j < p$  **do**

    Update  $\theta$  by running the training algorithm for  $n$  steps.

$i \leftarrow i + n$

$v' \leftarrow \text{ValidationSetError}(\theta)$

**if**  $v' < v$  **then**

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

**else**

$j \leftarrow j + 1$

**end if**

**end while**

Best parameters are  $\theta^*$ , best number of training steps is  $i^*$ .

---

# Dropout

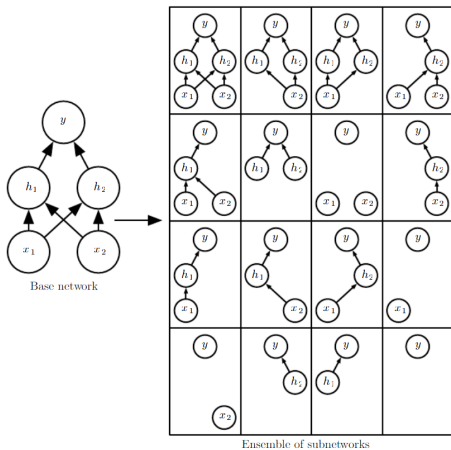
Computationally inexpensive, but powerful regularization technique that can be approximately thought of *bagging* many neural networks. It also controls for *co-adaptation* between units.

For each unit of the network, we randomly deactivate it with probability  $p$  independently from the other units, to generate a subnetwork that is then trained by SGD at each batch.

Once the training phase is done, we do the forward propagation on the network containing all units (i.e. the base network) by multiplying the weights by  $p$ .



# Dropout



**Figure:** Subnetworks Generated by Dropout

## Batch Normalization

**Basic idea:** Standardize dataflow throughout the neural network to smooth out loss surface and facilitate learning. Basically the idea of normalizing inputs, but applied at each hidden layer.

For  $\mathbf{a}^{(k)}$  the pre-activation at layer  $k$ , then the normalized pre-activation for batch  $\mathcal{B}$  is given by:

$$\hat{\mathbf{a}}^{(k)} = \frac{\mathbf{a}^{(k)} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}^{(k)}}{\sqrt{\left(\hat{\boldsymbol{\sigma}}_{\mathcal{B}}^{(k)}\right)^2 + \epsilon}}$$

where  $\hat{\boldsymbol{\mu}}_{\mathcal{B}}^{(k)} = \frac{1}{|\mathcal{B}|} \sum_{\mathcal{B}} \mathbf{a}^{(k)}$ ,  $\left(\hat{\boldsymbol{\sigma}}_{\mathcal{B}}^{(k)}\right)^2 = \frac{1}{|\mathcal{B}|} \sum_{\mathcal{B}} \left(\mathbf{a}^{(k)} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}^{(k)}\right)^2$  and  $\epsilon \sim 10^{-8}$ .

## Batch Normalization

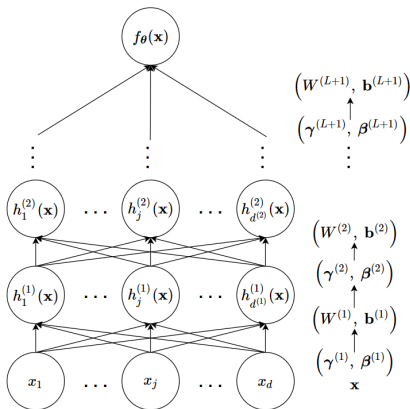
Then, the normalized pre-activation is scaled and a bias is also added, thus batch normalization gives us:

$$BN(\mathbf{a}^{(k)}) = \gamma^{(k)} \hat{\mathbf{a}}^{(k)} + \beta^{(k)}$$

where  $\gamma^{(k)}, \beta^{(k)} \in \mathbb{R}^{d^{(k)}}$  are new trainable parameters learned by backprop.

The layer activation becomes  $\mathbf{h}^{(k)} = g^{(k)}(BN(\mathbf{a}^{(k)}))$ , instead of  $\mathbf{h}^{(k)} = g^{(k)}(\mathbf{a}^{(k)})$ .

## Batch Normalization



During test time, use running average during training time for  $\gamma$ s and  $\beta$ s.

Makes deeper networks more stable during training and generally more robust, thus enabling the use of greater learning rates  $\eta$ .

It is extremely powerful, as it acts as a regularizer and it also speeds up training, but it is not yet fully understood.

## New Hyperparameters

Notice that since the first hyperparameters slide, we have added three new ones that are used in the training of a neural network:

- 1)  $\lambda$ : the regularization parameter,
- 2)  $\eta$ : the learning rate, and
- 3)  $m$ : the batch size.

We have also implicitly added other hyperparameters:

- 1) in the regularizer  $\Omega(\theta)$ , and
- 2) in the stopping criterion of gradient descent.

Thus, the number of possible hyperparameter combinations is astronomical.

# Hyperoptimization

We supposed that we knew the combinations  $\mathcal{H}$  to test, but it's not really the case. The hyperspace is extremely large, but we can only test a limited number of combinations. There exist multiple methods:

## 1) Grid search:

- Define a finite set of values for each hyperparameter (a grid).
- Test all combinations of that grid in conjunction with the meta-algorithm.

## 2) Random search:

- Define a range or set of values for each hyperparameter.
- Select a random combination.
- Test as many as possible within time and/or resources constraints in conjunction with the meta-algorithm.

# Hyperoptimization

The last method is more advanced, but can yield better results, more rapidly.

## 3) Bayesian search:

- Define a range or set of values for each hyperparameter.
- Based on previous trials, select a new combination with high potential (the one with the greatest expected improvement).
- Test as many as possible within time and/or resources constraints in conjunction with the meta-algorithm.

This method is implemented in the library Hyperopt.

Let's practice!