Applied Deep Learning

Lecture 1: *Fundamentals of DL*

Ali Bereyhi
ECE Department, University of Toronto
Fall 2026

Lecture Overview

Lecture 1: *Preliminaries and Fundamentals of DL*

Today's lecture covers

1. Basics of machine learning and deep learning
2. Components of machine learning
3. Model training via risk minimization

Preliminaries

Machine Learning (ML)

Goodfellow et al. *informally* define ML as *"... a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions..."*

Though good, this definition is still unclear! Let's put it in simple words

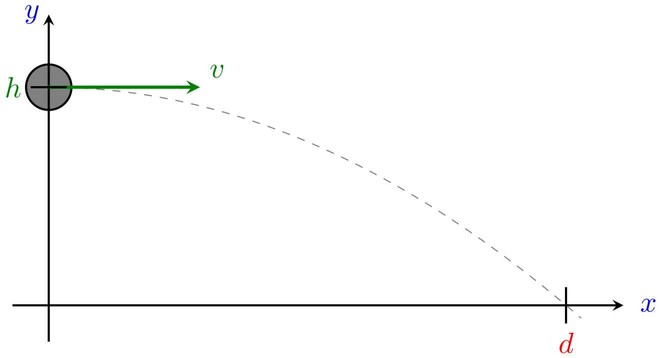
For most problems, there are two approaches to get to a solution

- through analytic derivation
- using data-driven algorithms

ML develops efficient data-driven algorithms for *complicated problems* whose analysis is either *infeasible* or *too complicated*

Motion of a Projectile

A projectile, located at height h, is initiated by horizontal velocity v. We are asked to determine horizontal distance d at which *it hits the ground*



Motion of a Projectile: *Analytic Solution*

At time t, the location of the projectile (x, y) is

$$x = vt \qquad\qquad y = -\frac{g}{2}t^2 + h$$

with $g \approx 9.8$ m/s^2 being the gravitational acceleration. When it hits the ground, we have $y = 0$; thus, we can find the hitting time as

$$t_0 = \sqrt{\frac{2h}{g}} = \gamma\sqrt{h}$$

At $t = t_0$, we have $x = d = vt_0$. Defining $\gamma = \sqrt{2/g} \approx 0.45$, we have

$$d = \gamma v \sqrt{h}$$

Motion of a Projectile: *Analytic Solution*

The projectile hits the ground at the horizontal distance

$$d = \gamma v \sqrt{h}$$

To derive this analytic result, we have used two facts

- We knew Newton's laws that describe the motion
- We could solve the equations for d analytically

Well! This is not the case in all problems! In fact, there might be

- no analytic law known that describes the relations, or
- a complicated law whose analysis is computationally infeasible

Motion of a Projectile: *ML Approach*

We conduct the experiment for I different times: in try $i = 1, \ldots, I$

- we initiate with different velocity v_i and height h_i
- we measure the horizontal distance d_i

We then assume that d_i and $[v_i, h_i]$ are related via a *pre-defined function*; for instance, we assume d_i and $[v_i, h_i]$ are related as

$$d_i = w_0 v_i + w_1 h_i$$

We then try to find the optimal values

$$w_0 = w_0^\star \text{ and } w_1 = w_1^\star$$

such that *pre-defined function closely matches our experimental results*

Motion of a Projectile: *ML Approach*

How can we find $w_0^\star$ *and* $w_1^\star$*?*

The function should match our experimental data, i.e., *for any* i

$$d_i \overset{!}{=} w_0^\star v_i + w_1^\star h_i \rightsquigarrow (d_i - w_0^\star v_i - w_1^\star h_i)^2 \overset{!}{=} 0$$

By $\overset{!}{=}$, we mean that we intend to have this identity holding

Having this identity *for any* i is equivalent to write

$$\sum_{i=1}^{I} (d_i - w_0^\star v_i - w_1^\star h_i)^2 \overset{!}{=} 0$$

But, such $w_0^\star$ and $w_1^\star$ do not necessarily exist if $I \geqslant 3$!

Motion of a Projectile: *ML Approach*

How can we find $w_0^\star$ *and* $w_1^\star$*?* We find $w_0^\star$ and $w_1^\star$ such that

$$\sum_{i=1}^{I} \left(d_i - w_0^\star v_i - w_1^\star h_i\right)^2$$

is as small as possible

Let us define the *loss* $\mathcal{L}\left(w_0, w_1\right)$ as

$$\mathcal{L}\left(w_0, w_1\right) = \sum_{i=1}^{I} \left(d_i - w_0 v_i - w_1 h_i\right)^2$$

We then find $w_0^\star$ *and* $w_1^\star$ *that minimize the loss*

$$\left(w_0^\star, w_1^\star\right) = \operatorname*{argmin}_{w_0, w_1} \mathcal{L}\left(w_0, w_1\right)$$

ML Components: *Dataset*

Dataset is the collection of data-points; in our example, the dataset was

$$\mathbb{D} = \{([v_i, h_i], d_i) : i = 1, \dots, I\}$$

which has I data-points $([v_i, h_i], d_i)$ for $i = 1, \dots, I$

> This is an example of a labeled dataset: a dataset whose data-points contain both the inputs and their corresponding labels (outputs)

More general, a labeled dataset with I data-points is

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \dots, I\}$$

- $\boldsymbol{x}_i \in \mathbb{R}^N$ is an input vector with N entries
- y_i is the label of $\boldsymbol{x}_i$ that is a scalar

ML Components: *Supervised Learning*

When the dataset is labeled, our *task* is clear:

- We observe I inputs to a function with their outputs (labels)
- We try to find out (learn) this function

Our task is a *learning task*, because we are trying to

learn the unknown relation between inputs and labels

It describes a *supervised learning problem*, since

a supervisor has filled our dataset with labels

In other words, for the sample inputs in our hand, we know the outputs

ML Components: *Unsupervised Learning*

+ *But can a dataset be unlabeled?*
– Yes! This is the case in *unsupervised learning* in which we are to learn features of a data-point $\boldsymbol{x}$ by investigating a set of its samples

In this case, the dataset is of the form

$$\mathbb{D} = \{\boldsymbol{x}_i : i = 1, \ldots, I\}$$

which is unlabeled. Our *learning task* is further *unsupervised*

Let's make it crystal clear via an example!

Example of Unsupervised Learning: *PCA*

An N-dimensional data-point $\boldsymbol{x}$ is coming from a natural process, e.g., it contains the pixel values of an image taken from body. It is known that $\boldsymbol{x}$ is a linear combination of $Q \ll N$ *principal vectors*: any $\boldsymbol{x}$ is

$$\boldsymbol{x} = \sum_{q=1}^{Q} a_q \boldsymbol{v}_q = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_Q] \begin{bmatrix} a_1 \\ \vdots \\ a_Q \end{bmatrix} = \mathbf{V}\mathbf{a}$$

with $\mathbf{a} \in \mathbb{R}^Q$ and $\mathbf{V} \in \mathbb{R}^{N \times Q}$. *Nevertheless, we do not know the principle vectors, i.e., matrix $\mathbf{V}$ is unknown to us.*

> *Our learning task is to*
>
> *find out what $\mathbf{V}$ is by investigating I samples of $\boldsymbol{x}$*

ML Components: *Model*

Now, we know what supervised learning and labeled dataset are

> Let's assume we are given by the labeled dataset
>
> $$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$
>
> and are to learn the function that relates an input $\boldsymbol{x}$ to its label y

For this task we need to consider a Model

> Model is a *parameterized* function that is used to describe the relation between the input $\boldsymbol{x}$ and its label y

In our dummy example, the model was linear

$$\text{label} = d = w_0 v + w_1 h = [w_0, w_1] \begin{bmatrix} v \\ h \end{bmatrix} = \mathbf{w}^{\mathsf{T}} \boldsymbol{x}$$

ML Components: *Model*

In our dummy example, we considered a linear model, but

- we could have considered an affine model

$$\text{label} = b + \mathbf{w}^{\mathsf{T}} \boldsymbol{x}$$

- or a polynomial model

$$\text{label} = b + \mathbf{w}_1^{\mathsf{T}} \boldsymbol{x} + \mathbf{w}_2^{\mathsf{T}} \boldsymbol{x}^2 + \ldots + \mathbf{w}_P^{\mathsf{T}} \boldsymbol{x}^P$$

Notation

By $f(\boldsymbol{x})$ we refer to entry-wise function operation

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \rightsquigarrow \boldsymbol{x}^p = \begin{bmatrix} {x_1}^p \\ \vdots \\ {x_N}^p \end{bmatrix} \text{ or } \sqrt{\boldsymbol{x}} = \begin{bmatrix} \sqrt{x_1} \\ \vdots \\ \sqrt{x_N} \end{bmatrix}$$

ML Components: *Model*

In our dummy example, we considered a linear model, but

- we could have considered even a wired model

$$\text{label} = \left(\mathbf{w}_1^{\mathsf{T}} \boldsymbol{x}\right) \left(\mathbf{w}_2^{\mathsf{T}} \sqrt{\boldsymbol{x}}\right)$$

Models are always parameterized meaning that

they contain some parameters that are to be tuned

Example: $\mathbf{w}$ *in linear model or* $\mathbf{w}_p$*'s and* b *in polynomial model*

Model parameters are of two types:

- *Hyperparameters*
- *Learnable parameters*

ML Components: *Model*

Hyperparameters

Parameters that are required to specify the model *explicitly*

Best example is the order P in the polynomial model: *we need to know P in order to write down the model explicitly*

If we know $P = 2$; then, we know that our model is

$$\textit{label} = b + \mathbf{w}_1^\mathsf{T} \boldsymbol{x} + \mathbf{w}_2^\mathsf{T} \boldsymbol{x}^2$$

Hyperparameters are specified prior to the start of learning process

ML Components: *Model*

Learnable Parameters

Once the hyperparameters are set, we have a model with some parameters that are to be learned, such that the model fits our dataset

In our polynomial example with $P = 2$: we learn $\mathbf{w}_1^\star$, $\mathbf{w}_2^\star$, and $b^\star$ such that

$$y_i \approx b^\star + \mathbf{w}_1^{\star\mathsf{T}} \boldsymbol{x}_i + \mathbf{w}_2^{\star\mathsf{T}} \boldsymbol{x}_i^{\,2}$$

+ *But what does this $\approx$ mean?*
– This approximation needs to be quantified!

This is what the loss function does for us

+ *How do we learn the learnable parameters?*
– We answer this after we understand the loss function

ML Components: *Loss*

Loss Function

Loss function quantifies *the difference between the output of the model and the true label*

Back to our polynomial example with dataset $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

- We set the hyperparameter P to $P = 2$
- We now set the learnable parameters

If we give the data-point $\boldsymbol{x}_i$ *to the model as input, it returns*

$$z_i = b + \mathbf{w}_1^{\mathsf{T}} \boldsymbol{x}_i + \mathbf{w}_2^{\mathsf{T}} \boldsymbol{x}_i^2$$

Loss determines the difference between z_i *and true label* y_i *in* $\mathbb{D}$

$$\mathcal{L}(z_i, y_i) = \ell_i \in \mathbb{R}$$

ML Components: *Loss*

Examples of loss function are

1. We can calculate the *squared error*

$$\mathcal{L}\left(z_i, y_i\right) = \left(z_i - y_i\right)^2$$

which intuitively determines the energy of the difference

2. We can calculate the *error indicator*

$$\mathcal{L}\left(z_i, y_i\right) = \mathbb{1}\left(z_i \neq y_i\right) = \begin{cases} 1 & z_i \neq y_i \\ 0 & z_i = y_i \end{cases}$$

which indicates the occurrence of error

ML Components: *A Quick Wrap-up*

Any ML problem has three components:

1. *Dataset* which is the collection of samples

$\boxed{\text{supervised learning}}$ $\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$

2. *Model* that describes the relation between the input and label

$\boxed{\mathbf{h}\text{: Hyperparameters}}$ $z = f_{\mathbf{h}}(\boldsymbol{x}|\mathbf{w})$ $\boxed{\mathbf{w}\text{: Learnable Parameters}}$

3. *Loss* quantifies the difference between the *model's output* and *true label*

$$\mathcal{L}(z_i, y_i)$$

What's Next?

Now, we know the components of an ML problem!

+ *How can we solve the problem? Or, speaking in the language of ML people, how can we address the learning task?*
– The learning task is to tune the learnable parameters

We approximate the lable of a new input $\boldsymbol{x}_{new}$ *as* $y_{new} = f_{\mathbf{h}}(\boldsymbol{x}_{new}|\mathbf{w})$

The process of tuning the learnable parameters is called training

+ *How do we do the training?*
– We see it shortly, but first we need a more serious example!

Classification

Classification is a supervised learning problem in which

labels belong to a discrete set: $y_i \in \{c_1, \ldots, c_J\}$

the label y_i represents the class to which the input $\boldsymbol{x}_i$ belongs

Best example is image classification:

- The dataset contains some images and their labels

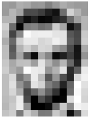
$$\mathbb{D} = \{(\text{image } 1, \text{dog}), (\text{image } 2, \text{cat}), \ldots, (\text{image } I, \text{cat})\}$$

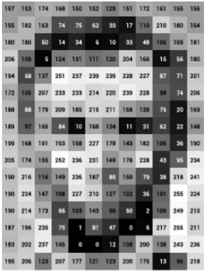
- The images are either *dog* or *cat*

We can simply convert $\mathbb{D}$ into a numerical dataset

Classification: *Image Classification*

Any image is nothing but *a collection of pixel values*





157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Thus, image i in $\mathbb{D}$ can be shown by a pixel vector $\boldsymbol{x}_i$

Classification: *Image Classification*

The labels can further be marked by integer numbers as classes

> In our example, we have *two classes of dog and cat*: so, we could say
>
> *label of dog images* $= 0$ and *label of cat images* $= 1$

So, a collection of I images like

$$\mathbb{D} = \left\{ \left(\text{image } 1, \text{dog}\right), \left(\text{image } 2, \text{cat}\right), \ldots, \left(\text{image } I, \text{cat}\right) \right\}$$

can be represented by the numerical dataset

$$\mathbb{D} = \left\{ \left(\boldsymbol{x}_1, 0\right), \left(\boldsymbol{x}_2, 1\right), \ldots, \left(\boldsymbol{x}_I, 1\right) \right\}$$

pixel vector of image 1 it's an image of a dog

Classification: *Binary Classification*

We now start with the basic case of binary classification

Binary Classification

A classification problem with only two classes, i.e., $y_i \in \{0, 1\}$

Our example was a binary classification with dog: 0 and cat: 1

Let's build the main components of this ML problem

Binary Classification: *Dataset*

Dataset is similar to what we had in our dog or cat example

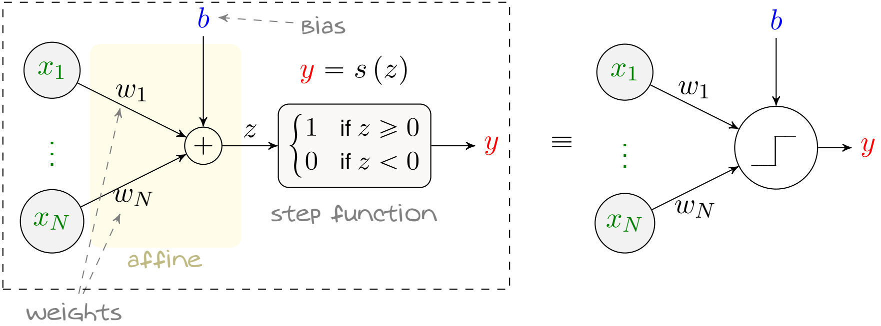
$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

- $\boldsymbol{x}_i \in \mathbb{R}^N$ is a real-valued N-dimensional vector
  - *For instance it contains the pixel values of an image with N pixels*
- $y_i \in \{0, 1\}$ is a binary label

Binary Classification: *Model*

+ *What should our model be?*
– The model gets N real numbers and returns a binary number

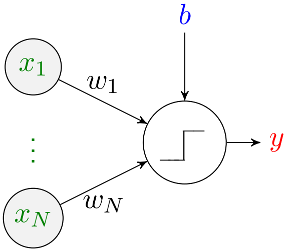
Let's start with perceptron who is the mother of neural networks



Binary Classification: *Perceptron*

Perceptron is a *linear classifier*

- *it determines an linear transform of $\boldsymbol{x}$*
- *it classifies using the sign of this transform*

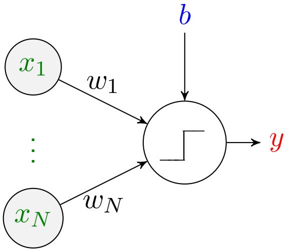


Mathematically, the perceptron is described by

$$y = s\Big(\sum_{n=1}^{N} w_n x_n + b\Big) = s\Big(\underbrace{[w_1, \ldots, w_N]}_{\mathbf{w}^\mathsf{T}} \underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}}_{\boldsymbol{x}} + b\Big) = s\big(\mathbf{w}^\mathsf{T} \boldsymbol{x} + b\big)$$

Binary Classification: *Perceptron*

$$y = s(\mathbf{w}^{\mathsf{T}}\boldsymbol{x} + b)$$

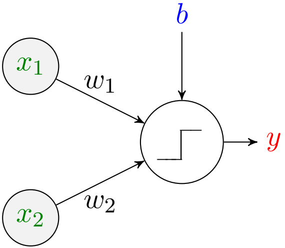


+ *Is there any hyperparameter in this model?*
– We could look at step function as a hyperparameter: *we could have chosen another function to map the affine transform to a binary label*
+ *Is there any learnable parameter in this model?*
– Of course! $\mathbf{w}$ and b are the learnable parameters

Perceptron: *Geometrical Interpretation*

Perceptron geometrically realizes a linear division of $\mathbb{R}^N$

To see this, let's focus on the two-dimensional case, i.e., $N = 2$



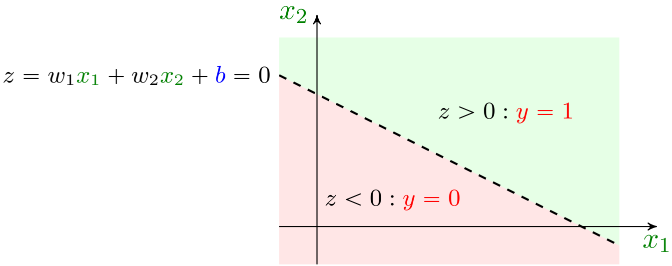
$$y = s(\mathbf{w}^\mathsf{T}\boldsymbol{x} + b) = s(w_1 x_1 + w_2 x_2 + b) = \begin{cases} 1 & \text{if } w_1 x_1 + w_2 x_2 + b \geqslant 0 \\ 0 & \text{if } w_1 x_1 + w_2 x_2 + b < 0 \end{cases}$$

Perceptron: *Geometrical Interpretation*

Let's go to the two-dimensional space with one dimension denoting x_1 and the other x_2 and plot the line

$$z = w_1 x_1 + w_2 x_2 + b = 0$$

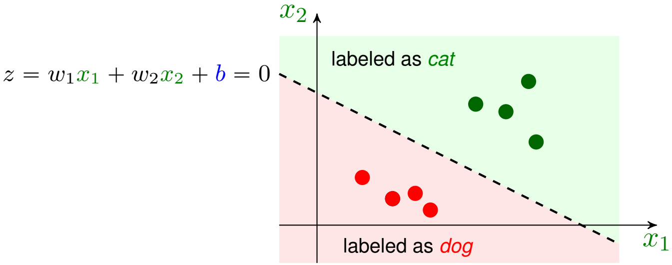
for some $w_1, w_2 > 0$ and $b < 0$



Perceptron draws a line, and then classifies every point above it with label 1 and every point below it with label 0

Perceptron: *Geometrical Interpretation*

So, if we have a perceptron with learnable parameters $w_1, w_2 > 0$*, and* $b < 0$



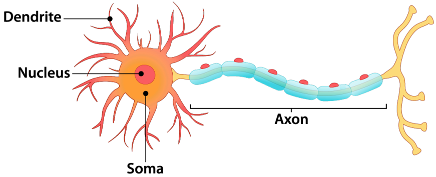
For instance in the example of image classification

- *Every • represents a data-point ≡ pixel vector: think of a two-pixel image* ☺
- *Perceptron gets the values of the two pixels and predicts whether it is an image of a dog or a cat*

In realistic problems, this viewpoint extends to large-dimensional space

Perceptron: *A Bit of History*

- Perceptron machine was implemented by Frank Rosenblatt in 1957
- He wrote a paper in 1958 illustrating the machine and its algorithm
- The original idea is however older than that
  - ↳ *It was proposed in 1943 by Warren McCulloch and Walter Pitts*
  - ↳ *They introduced it to abstractly describe biological neurons*
  - ↳ *This was why perceptron is also called McCulloch-Pitts neuron*



- Perceptron was a breakthrough in the long-time ongoing attempt to understand the functionality of brain

Components of Binary Classification

Back to binary classification: we have the first two components

1. A dataset with N-dimensional inputs $\boldsymbol{x}_i$ and binary labels $y_i \in \{0,1\}$

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

2. Perceptron as the model

$$\hat{y} = s(\mathbf{w}^{\mathsf{T}} \boldsymbol{x} + b)$$

3. For two binary variables y and $\hat{y}$, we can set the loss to

$$\mathcal{L}(\hat{y}, y) = \mathbb{1}(\hat{y} \neq y) = \begin{cases} 1 & \hat{y} \neq y \\ 0 & \hat{y} = y \end{cases}$$

Training a Model

Training a Model

Up to now, we have learned the main components of ML problems and seen a classical example, i.e., classification

---

We now want to understand how we can tune the learnable parameters

> As mentioned, tuning the learnable parameters of a model is called
>
> *training of the model*

Before starting with training, let's review some basics of probability theory

Recap: Probability Theory

Discrete random variable x is described by *probability mass function* $P(x)$

$$\Pr\{x = x_0\} = P(x_0)$$

Assuming that $x \in \mathbb{X}$, we have

$$\sum_{x \in \mathbb{X}} P(x) = 1$$

The *expectation* is also defined as

$$\mathbb{E}\{x\} = \sum_{x \in \mathbb{X}} x P(x)$$

Recap: Probability Theory

Continuous random variable x is described by *probability density function* $P(x)$

$$\Pr\{a < x \leqslant b\} = \int_a^b P(x)\mathrm{d}x$$

We have in this case

$$\int_{-\infty}^{+\infty} P(x)\mathrm{d}x = 1$$

The *expectation* is also defined as

$$\mathbb{E}\{x\} = \int x P(x)\,\mathrm{d}x$$

When we talk about a general random variable, we call $P(x)$ the distribution

Recap: Probability Theory

A random vector $\boldsymbol{x}$ is a vector of random variables and its distribution

$$P(\boldsymbol{x}) = P(x_1, \ldots, x_N)$$

is the *joint distribution of the entries*

Also, we can take expectation of any function of $\boldsymbol{x}$ *as*

$$\mathbb{E}\left\{f(\boldsymbol{x})\right\} = \int f(\boldsymbol{x}) P(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$$

Recap: Law of Large Numbers

Consider a random sequence $x_1, \ldots, x_I$

we call it independent and identically distributed (i.i.d.) with $x \sim P(x)$

if x_i's are generated *independently* all with the same distribution $P(x)$

Law of Large Numbers

Assume $x_1, \ldots, x_I$ is an i.i.d. sequence with $x \sim P(x)$; then, we have[1]

$$\frac{1}{I} \sum x_i \rightarrow \mathbb{E}\{x\}$$

as $I \rightarrow \infty$

Intuition. If we *arithmetically average* too many samples of a random process, we get a value very close to its *expectation*

[1] Of course under some conditions which we assume holding

Training a Model

Let us now use the probability theory to derive a meaningful approach of model training: for sake of simplicity, let's assume that we have dataset

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \ldots, I\}$$

with *scalar (one-dimensional) inputs*, i.e., x_i's are real numbers

---

Let us further denote our model as

$$y = f(x|\mathbf{w})$$

where in this notation

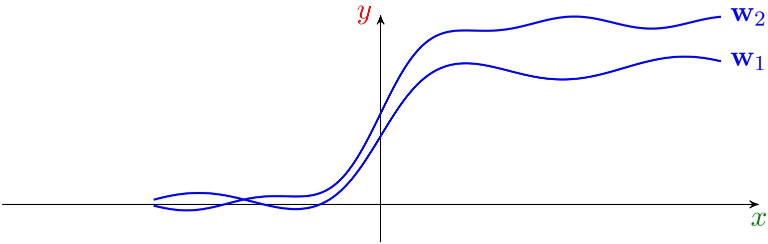
- $\mathbf{w}$ contains all learnable parameters that are to be learned in training
- we drop the hyperparameters, since *they are yet fixed by genie*

Training a Model

The *one-dimensional* assumption helps us visualize the model

$$y = f\left(x \middle| \mathbf{w}\right)$$

With scalar input, we can visualize the model as



As learnable parameters change, the model represents different functions

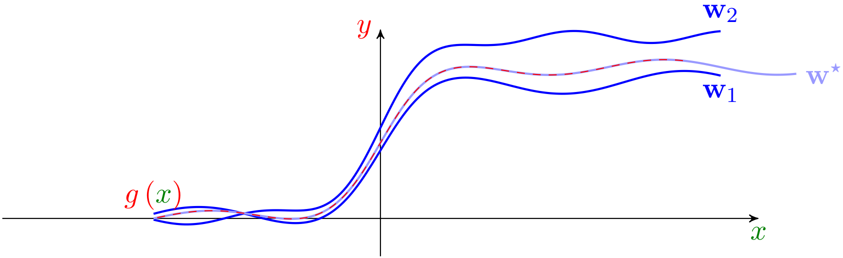
Training a Model

In reality, the labels and inputs are related through a function $g(\cdot)$

$$\boxed{y = g(x) \rightsquigarrow \text{function } g(\cdot) \text{ is however unknown to us}}$$

In fact, our whole learning task is to learn it from the dataset!

+ *What if we knew function* $g(\cdot)$*? How would have we trained our model?*
– Well! We would have tuned $\mathbf{w}$ until the model matches $g(\cdot)$



Training a Model

We could represent such $\mathbf{w}^{\star}$ *using the loss*

Point-wise Loss

The model with parameter $\mathbf{w}$ *gives us the label* $\hat{y}_0 = f(x_0|\mathbf{w})$ *for input* x_0

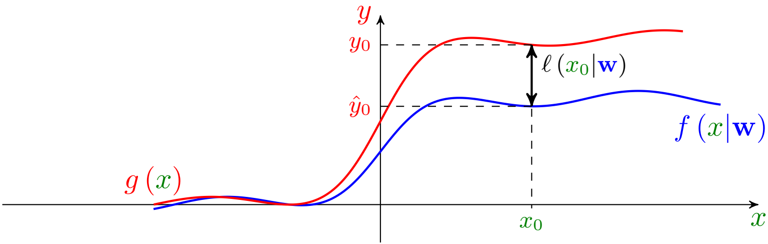
- *It may be different from the true label* $y_0 = g(x_0)$
- *The loss between the two is given by*

$$\ell(x_0|\mathbf{w}) = \mathcal{L}(\hat{y}_0, y_0) = \mathcal{L}(f(x_0|\mathbf{w}), g(x_0))$$

Training a Model: *Point-wise Loss*

$$\ell(x_0|\mathbf{w}) = \mathcal{L}(\hat{y}_0, y_0) = \mathcal{L}(f(x_0|\mathbf{w}), g(x_0))$$

Let's visualize this loss



Training a Model: *Risk*

We would like our model to recover the true label as closely as possible: so, the best option at point x_0 is to find the choice of $\mathbf{w}$ that minimizes $\ell\left(x_0|\mathbf{w}\right)$

- \+ *But x_0 is a single point! What about other inputs?!*
- – *Right! We should learn $\mathbf{w}$ for any x_0*
- \+ *How can we do it?*
- – *We treat x_0 as a random variable with some distribution $P\left(x_0\right)$, and minimize the expexted loss often called risk*

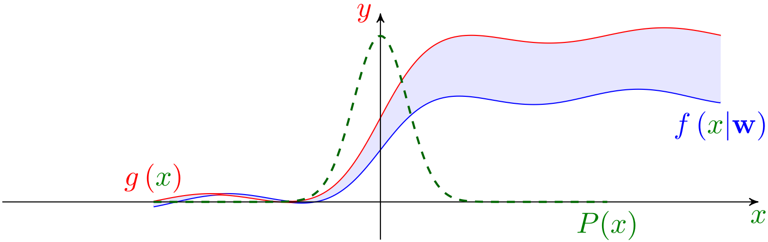
Risk

For given learnable parameter $\mathbf{w}$, the risk is defined as

$$R\left(\mathbf{w}\right) = \mathbb{E}\left\{\ell\left(x_0|\mathbf{w}\right)\right\} = \int \ell\left(x_0|\mathbf{w}\right) P\left(x_0\right) \mathrm{d}x_0$$

Training a Model: *Risk Minimization*

Let's visualize the risk



Risk Minimization

Ideally, the training is formulated as the minimization of risk

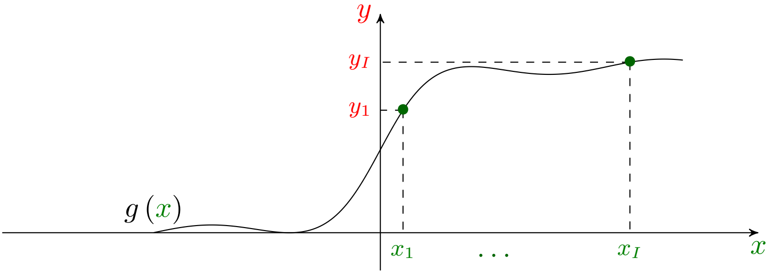
$$\mathbf{w}^\star = \operatorname*{argmin}_{\mathbf{w}} R(\mathbf{w})$$

Training a Model: *Empirical Risk*

+ *Bravo! But, the training seems infeasible, since we have neither the true function $g(\cdot)$, nor the distribution $P(x)$!*
– Right! But, we could handle this approximately using the LLN

Let's look at what we have: *the dataset which contains samples of* $g(\cdot)$

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \ldots, I\}$$



Training a Model: *Empirical Risk*

We can determine point-wise losses for points in the dataset

$$\ell\left(x_i|\mathbf{w}\right) = \mathcal{L}\left(f\left(x_i|\mathbf{w}\right), g\left(x_i\right)\right) = \mathcal{L}\left(f\left(x_i|\mathbf{w}\right), y_i\right)$$

Now, assume that data-points are *drawn from i.i.d. unknown* $P\left(x\right)$

LLN suggests that average of point-wise losses converges to the risk when we have a large enough number of data-points, i.e.

$$\frac{1}{I}\sum_{i=1}^{I} \ell\left(x_i|\mathbf{w}\right) \to \mathbb{E}\left\{\ell\left(x|\mathbf{w}\right)\right\} = R\left(\mathbf{w}\right)$$

We call this arithmetic average the empirical risk

Empirical risk is the best estimate of risk that we get from our dataset

Training a Model: *Empirical Risk Minimization*

Empirical Risk

Let $\mathbf{w}$ *includes all learnable parameters of the model, and the dataset be*

$$\mathbb{D} = \{(\boldsymbol{x}_i, y_i) : i = 1, \ldots, I\}$$

for loss function $\mathcal{L}$, *the empirical risk is defined as*

$$\hat{R}(\mathbf{w}) = \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}\left(f\left(\boldsymbol{x}_i | \mathbf{w}\right), y_i\right)$$

Model Training: *Empirical Risk Minimization*

The training is performed by minimizing the empirical risk

Empirical Risk Minimization

Training model on a *dataset* $\equiv$ *minimizing empirical risk* computed from the *dataset*

$$\mathbf{w}^\star = \operatorname*{argmin}_{\mathbf{w}} \hat{R}(\mathbf{w}) \qquad \text{(Training)}$$

$$= \operatorname*{argmin}_{\mathbf{w}} \frac{1}{I} \sum_{i=1}^{I} \mathcal{L}\left(f\left(\boldsymbol{x}_i | \mathbf{w}\right), y_i\right)$$

Wrap-Up: *Universal Scheme for ML*

So, we know pretty much all the theory for supervised learning:

1. We build the main three components:
   1.1 Dataset
   1.2 Model
   1.3 Loss Function
2. We determine the empirical risk of the model using loss function
3. We train the model by minimizing the empirical risk

We call this universal approach ML 1-2-3!

The theory is pretty short; however,

how to execute each step of ML 1-2-3 is a pretty long story

that we are going to learn in the remaining of this course!

Further Read

Further Read

Read More on These Topics

- Goodfellow, et al., *Deep Learning*
  - ↳ Chapters 2 and 3
- Bishop & Bishop, *Deep Learning: Foundations and Concepts*
  - ↳ Chapters 2 and 5

Nice to Check

1. *A logical calculus of the ideas immanent in nervous activity*
   - ↳ McCulloch and Pitts (1943)
2. *An overview of statistical learning theory*
   - ↳ Vapnik (1999)