Deep learning as a building block in probabilistic models

Pierre-Alexandre Mattei

http://pamattei.github.io/

@pamattei

Inria, Maasai team
Université Côte d'Azur

But actually, what is deep learning?

Deep learning is a **general framework for function approximation.**

But actually, what is deep learning?

Deep learning is a **general framework for function approximation.**

It uses parametric approximators called **neural networks**, which are compositions of some tunable **affine functions** $f_1, \ldots, f_L$ with a simple fixed **nonlinear function** σ**:**

$$F(\mathbf{x}) = f_1 \circ \sigma \circ f_2 \circ \ldots \circ \sigma \circ f_L(\mathbf{x})$$

These functions are called **layers.** The nonlinearity σ is usually called the **activation function.**

But actually, what is deep learning?

Deep learning is a **general framework for function approximation.**

It uses parametric approximators called **neural networks**, which are compositions of some tunable **affine functions** $f_1, \ldots, f_L$ with a simple fixed **nonlinear function** σ**:**

$$F(\mathbf{x}) = f_1 \circ \sigma \circ f_2 \circ \ldots \circ \sigma \circ f_L(\mathbf{x})$$

These functions are called **layers.** The nonlinearity σ is usually called the **activation function.**

The derivatives of F with respect to the tunable parameters can be computed using the chain rule via the **backpropagation algorithm.**

A glimpse at the zoology of layers

The simplest kind of affine layer is called a **fully connected layer:**

$$f_l(\mathbf{x}) = \mathbf{W}_l\mathbf{x} + \mathbf{b}_l,$$

where $\mathbf{W}_l$ and $\mathbf{b}_l$ are tunable parameters.

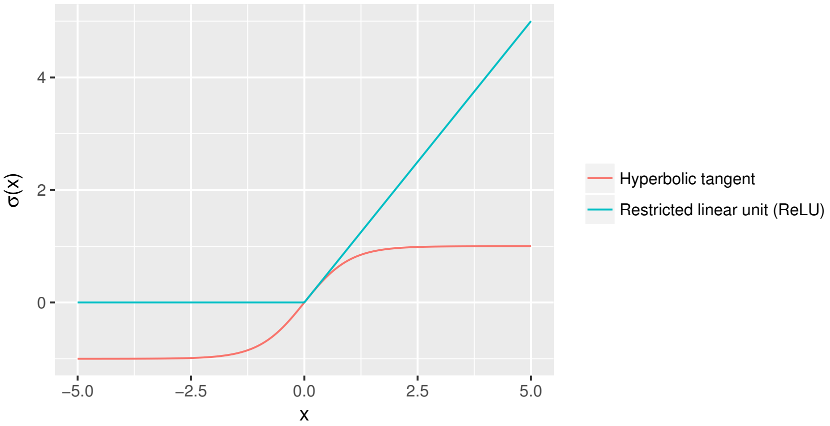
A glimpse at the zoology of layers

The simplest kind of affine layer is called a **fully connected layer:**

$$f_l(\mathbf{x}) = \mathbf{W}_l\mathbf{x} + \mathbf{b}_l,$$

where $\mathbf{W}_l$ and $\mathbf{b}_l$ are tunable parameters.

The activation function σ is usually a **univariate fixed function** applied elementwise. Here are two popular choices:



Why is it convenient to compose affine functions?

- Neural nets are powerful approximators: **any continuous function can be arbitrarily well approximated** on a compact using a three-layer fully connected network $F = f_1 \circ \sigma \circ f_2$ (**universal approximation theorem**, Leshno, Lin, Pinkus, and Schocke, Neural Netw., 1993). The conditions are that σ is not a polynomial and that **the network can be arbitrarily wide.** A good review of similar results is the one of Pinkus (Acta Numer., 1998).

Why is it convenient to compose affine functions?

- Neural nets are powerful approximators: **any continuous function can be arbitrarily well approximated** on a compact using a three-layer fully connected network $F = f_1 \circ \sigma \circ f_2$ (**universal approximation theorem**, Leshno, Lin, Pinkus, and Schocke, Neural Netw., 1993). The conditions are that σ is not a polynomial and that **the network can be arbitrarily wide.** A good review of similar results is the one of Pinkus (Acta Numer., 1998).

- There are similar results for **very thin but arbitrarily deep networks** (Lin & Jegelka, NeurIPS 2018).

Why is it convenient to compose affine functions?

- Neural nets are powerful approximators: **any continuous function can be arbitrarily well approximated** on a compact using a three-layer fully connected network $F = f_1 \circ \sigma \circ f_2$ (**universal approximation theorem**, Leshno, Lin, Pinkus, and Schocke, Neural Netw., 1993). The conditions are that σ is not a polynomial and that **the network can be arbitrarily wide.** A good review of similar results is the one of Pinkus (Acta Numer., 1998).

- There are similar results for **very thin but arbitrarily deep networks** (Lin & Jegelka, NeurIPS 2018).

- Some **prior knowledge** can be distilled into the **architecture** (i.e. the type of affine functions/activations) of the network. For example, **convolutional neural networks** (CNNs, LeCun et al., NeuIPS 1990) leverage the fact that local information plays an important role in images/sound/sequence data. In that case, the affine functions are convolution operators with some learnt filters.

Why is it convenient to compose affine functions?

- Often, this prior knowledge can be based on **known symmetries**, leading to deep architectures that are **equivariant or invariant to the action of some group** (see e.g. the work of Taco Cohen or Stéphane Mallat). This is useful when dealing with images, sound, molecules...

Why is it convenient to compose affine functions?

- Often, this prior knowledge can be based on **known symmetries**, leading to deep architectures that are **equivariant or invariant to the action of some group** (see e.g. the work of Taco Cohen or Stéphane Mallat). This is useful when dealing with images, sound, molecules...

- The layers can capture **hierarchical representations** of the data, sometimes (almost) explicitely (e.g. the capsules of Hinton et al., ICLR 2018).

Why is it convenient to compose affine functions?

- Often, this prior knowledge can be based on **known symmetries**, leading to deep architectures that are **equivariant or invariant to the action of some group** (see e.g. the work of Taco Cohen or Stéphane Mallat). This is useful when dealing with images, sound, molecules...
- The layers can capture **hierarchical representations** of the data, sometimes (almost) explicitely (e.g. the capsules of Hinton et al., ICLR 2018).
- When the neural network parametrises a regression function, empirical evidence shows that **adding more layers leads to better out-of-sample behaviour.** Roughly, this means that adding more layers is a way of increasing the complexity of statistical models without paying a large overfitting price: there is a **regularisation-by-depth effect.**

A simple example: nonlinear regression with a multilayer perceptron (MLP)

We want to perform regression on a data set

$$(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n) \in \mathbb{R}^p \times \mathbb{R}.$$

A simple example: nonlinear regression with a multilayer perceptron (MLP)

We want to perform regression on a data set

$$(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \in \mathbb{R}^p \times \mathbb{R}.$$

We can model the regression function using a **multilayer perceptron (MLP):** two connected layers with an hyperbolic tangent in-between:

$$y \approx F_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_1 \text{tanh}(\mathbf{W}_0 \mathbf{x} + \mathbf{b}_0) + \mathbf{b}_1.$$

We call the complete set of parameters $\boldsymbol{\theta} = (\mathbf{W}_1, \mathbf{W}_0, \mathbf{b}_1, \mathbf{b}_0)$. The coordinates of the intermediate representation $\mathbf{W}_0 \mathbf{x} + \mathbf{b}_0$ are called **hidden units.** The more hidden units, the more flexible (and difficult to train) the model.

A simple example: nonlinear regression with a multilayer perceptron (MLP)

We want to perform regression on a data set

$$(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \in \mathbb{R}^p \times \mathbb{R}.$$

We can model the regression function using a **multilayer perceptron (MLP):** two connected layers with an hyperbolic tangent in-between:

$$y \approx F_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_1 \text{tanh}(\mathbf{W}_0\mathbf{x} + \mathbf{b}_0) + \mathbf{b}_1.$$

We call the complete set of parameters $\boldsymbol{\theta} = (\mathbf{W}_1, \mathbf{W}_0, \mathbf{b}_1, \mathbf{b}_0)$. The coordinates of the intermediate representation $\mathbf{W}_0\mathbf{x} + \mathbf{b}_0$ are called **hidden units.** The more hidden units, the more flexible (and difficult to train) the model.

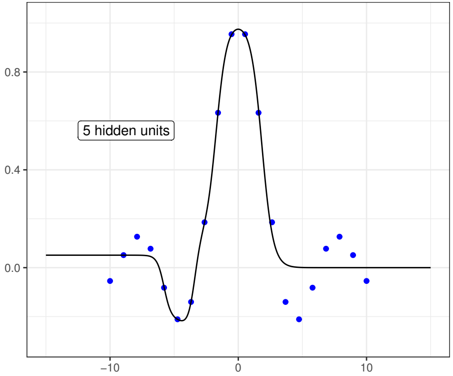
A natural way to find the parameters $\boldsymbol{\theta}$ (*a.k.a.* the weights) of the MLP $F_{\boldsymbol{\theta}}$, is to minimise the mean squared loss:

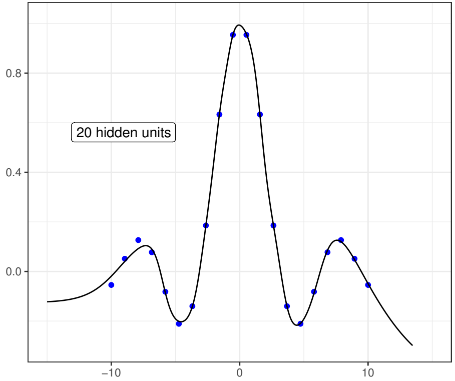
$$\text{MSE}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - F_{\boldsymbol{\theta}}(\mathbf{x}_i))^2$$

A simple example: nonlinear regression with a multilayer perceptron (MLP)

$$\forall i \leq n,\ y_i \approx F_{\boldsymbol{\theta}}(\mathbf{x}_i) = \mathbf{W}_1 \text{tanh}(\mathbf{W}_0 \mathbf{x}_i + \mathbf{b}_0) + \mathbf{b}_1,$$

Let's try to recover the function $\sin(x)/x$ using 20 samples:





What’s "kind of wrong" with the approach we just saw?

This was of just minimising the MSE allows us to do predictions, but **we cannot assess the uncertainty of our predictions.**

What’s "kind of wrong" with the approach we just saw?

This was of just minimising the MSE allows us to do predictions, but **we cannot assess the uncertainty of our predictions.**

Being able to model uncertainty is critical in machine learning, even in purely predictive tasks.

What’s "kind of wrong" with the approach we just saw?

This was of just minimising the MSE allows us to do predictions, but **we cannot assess the uncertainty of our predictions.**

Being able to model uncertainty is critical in machine learning, even in purely predictive tasks.

So what do we want?

What’s "kind of wrong" with the approach we just saw?

This was of just minimising the MSE allows us to do predictions, but **we cannot assess the uncertainty of our predictions.**

Being able to model uncertainty is critical in machine learning, even in purely predictive tasks.

So what do we want?

We want to be able to make probabilistic preditions, like

- "the probability that the temperature in Nice tomorrow is between 20 and 25 degrees in 17%",
- "the probability that this patient has this kind of cancer is 56%".

What’s "kind of wrong" with the approach we just saw?

This was of just minimising the MSE allows us to do predictions, but **we cannot assess the uncertainty of our predictions.**

Being able to model uncertainty is critical in machine learning, even in purely predictive tasks.

So what do we want?

We want to be able to make probabilistic preditions, like

- "the probability that the temperature in Nice tomorrow is between 20 and 25 degrees in 17%",
- "the probability that this patient has this kind of cancer is 56%".

To do that, we need to have a **probabilistic model of our data**, hence the need for **generative models**.

What's a generative model?

Let's start with some data $\mathcal{D}$. For example, in the **regression** case with p-dimensional continuous features,

$$\mathcal{D} = ((\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)) \in (\mathbb{R}^p \times \mathbb{R})^n.$$

In the **binary classification** case,

$$\mathcal{D} = ((\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)) \in (\mathbb{R}^p \times \{0, 1\})^n.$$

What's a generative model?

Let's start with some data $\mathcal{D}$. For example, in the **regression** case with p-dimensional continuous features,

$$\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)) \in (\mathbb{R}^p \times \mathbb{R})^n.$$

In the **binary classification** case,

$$\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)) \in (\mathbb{R}^p \times \{0, 1\})^n.$$

In the **unsupervised** case, the data usually looks like $\mathcal{D} = (\mathbf{x}_1, ..., \mathbf{x}_n) \in (\mathbb{R}^p)^n$.

What's a generative model?

Let's start with some data $\mathcal{D}$. For example, in the **regression** case with p-dimensional continuous features,

$$\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)) \in (\mathbb{R}^p \times \mathbb{R})^n.$$

In the **binary classification** case,

$$\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)) \in (\mathbb{R}^p \times \{0, 1\})^n.$$

In the **unsupervised** case, the data usually looks like $\mathcal{D} = (\mathbf{x}_1, ..., \mathbf{x}_n) \in (\mathbb{R}^p)^n$.

We call $(\mathbf{x}_1, ..., \mathbf{x}_n)$ the **features** and $(y_1, ..., y_n)$ the **labels.** The features are usually stored in a $n \times p$ **matrix called the design matrix.**

What's a generative model?

Let's start with some data $\mathcal{D}$. For example, in the **regression** case with p-dimensional continuous features,

$$\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)) \in (\mathbb{R}^p \times \mathbb{R})^n.$$

In the **binary classification** case,

$$\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)) \in (\mathbb{R}^p \times \{0, 1\})^n.$$

In the **unsupervised** case, the data usually looks like $\mathcal{D} = (\mathbf{x}_1, ..., \mathbf{x}_n) \in (\mathbb{R}^p)^n$.

We call $(\mathbf{x}_1, ..., \mathbf{x}_n)$ the **features** and $(y_1, ..., y_n)$ the **labels.** The features are usually stored in a $n \times p$ **matrix called the design matrix.**

A generative model "describes a process that is assumed to give rise to some data"

David MacKay, in his book *Information Theory, Inference, and Learning Algorithms* (2003).

Formally, **a generative model will just be a probability density $p(\mathcal{D})$.**

Generative models for supervised learning: *General assumptions*

Although **we'll mostly focus on the unsupervised case my lectures**, let us begin with the (arguably simpler) supervised case $\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n))$. It could be either a regression or a classification task, for example.

Generative models for supervised learning: *General assumptions*

Although **we'll mostly focus on the unsupervised case my lectures**, let us begin with the (arguably simpler) supervised case $\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n))$. It could be either a regression or a classification task, for example.

Most of the time, it makes sense to build generative models that assume that **the observations are independent.** This leads to

$$p(\mathcal{D}) = p((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)) = \prod_{i=1}^{n} p(\mathbf{x}_i, y_i).$$

Usually, we also further assume that the data are **identically distributed.** This means that all the $(\mathbf{x}_i, y_i)$ will follow the same distribution that we may denote $p(\mathbf{x}, y)$

Generative models for supervised learning: *General assumptions*

Although **we'll mostly focus on the unsupervised case my lectures**, let us begin with the (arguably simpler) supervised case $\mathcal{D} = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n))$. It could be either a regression or a classification task, for example.

Most of the time, it makes sense to build generative models that assume that **the observations are independent.** This leads to

$$p(\mathcal{D}) = p((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)) = \prod_{i=1}^{n} p(\mathbf{x}_i, y_i).$$

Usually, we also further assume that the data are **identically distributed.** This means that all the $(\mathbf{x}_i, y_i)$ will follow the same distribution that we may denote $p(\mathbf{x}, y)$

When these two assumptions are met, we say that the data are **independent and identically distributed (i.i.d.).** This is **super useful** is practice because, rather than having to find a distribution $p((\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n))$ over a very large space (whose dimension grows linearly with n), **we'll just have to find a much lower dimensional distribution $p(\mathbf{x}, y)$.**

Generative models for supervised learning: *Do we really have to be fully generative?*

Using the product rule, we may rewrite our $p(\mathbf{x}, y)$ as

$$p(\mathbf{x}, y) = p(\mathbf{x})p(y|\mathbf{x}) = p(y)p(\mathbf{x}|y).$$

Generative models for supervised learning: *Do we really have to be fully generative?*

Using the product rule, we may rewrite our $p(\mathbf{x}, y)$ as

$$p(\mathbf{x}, y) = p(\mathbf{x})p(y|\mathbf{x}) = p(y)p(\mathbf{x}|y).$$

But if we mainly want to do (probailisitic) preditions, knowing $p(y|\mathbf{x})$ is enough. It's exactly this **conditional distribution** that will give us statements like "the probability that this patient $\mathbf{x}$ has this kind of cancer is 56%".

Generative models for supervised learning: *Do we really have to be fully generative?*

Using the product rule, we may rewrite our $p(\mathbf{x}, y)$ as

$$p(\mathbf{x}, y) = p(\mathbf{x})p(y|\mathbf{x}) = p(y)p(\mathbf{x}|y).$$

But if we mainly want to do (probailisitic) preditions, knowing $p(y|\mathbf{x})$ is enough. It's exactly this **conditional distribution** that will give us statements like "the probability that this patient $\mathbf{x}$ has this kind of cancer is 56%".

Based on these insights, there are **two main approaches** for building $p(\mathbf{x}, y)$:

- The **fully generative (or model-based) approach posits a joint distribution** $p(\mathbf{x}, y)$ (often by specifying both $p(y)$ and $p(\mathbf{x}|y)$).
- The **discriminative (or conditional) approach** just specifies $p(y|\mathbf{x})$ and completely ignores $p(\mathbf{x})$.

Generative models for supervised learning: *Do we really have to be fully generative?*

Using the product rule, we may rewrite our $p(\mathbf{x}, y)$ as

$$p(\mathbf{x}, y) = p(\mathbf{x})p(y|\mathbf{x}) = p(y)p(\mathbf{x}|y).$$

But if we mainly want to do (probailisitic) preditions, knowing $p(y|\mathbf{x})$ is enough. It's exactly this **conditional distribution** that will give us statements like "the probability that this patient $\mathbf{x}$ has this kind of cancer is 56%".

Based on these insights, there are **two main approaches** for building $p(\mathbf{x}, y)$:

- The **fully generative (or model-based) approach posits a joint distribution** $p(\mathbf{x}, y)$ (often by specifying both $p(y)$ and $p(\mathbf{x}|y)$).
- The **discriminative (or conditional) approach** just specifies $p(y|\mathbf{x})$ and completely ignores $p(\mathbf{x})$.

What do you think are the benefits of the two approaches?

Generative models for supervised learning: *Discriminative vs fully generative*

A few examples of the two approaches:

- **Discriminative:** linear and logistic regression, Neural nets for regression/classification, Gaussian process regression/classification
- **Generative:** Linear/quadratic discriminant analysis, Mixture discriminant analysis, Supervised variational autoencoders, most of the models Charles Bouveyron will talk about in his course[1]

[1] cf. his book with G. Celeux, B. Murphy et A. Raftery.

Generative models for supervised learning: *Discriminative vs fully generative*

A few examples of the two approaches:

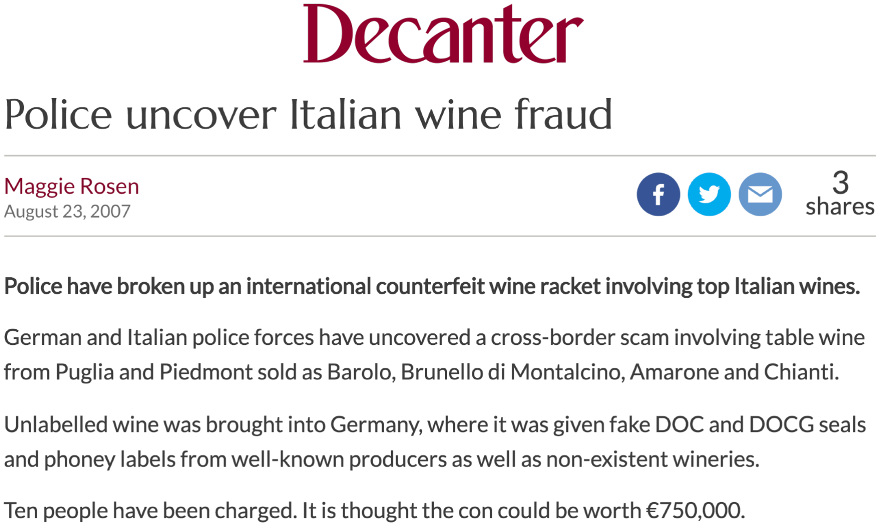
- **Discriminative:** linear and logistic regression, Neural nets for regression/classification, Gaussian process regression/classification
- **Generative:** Linear/quadratic discriminant analysis, Mixture discriminant analysis, Supervised variational autoencoders, most of the models Charles Bouveyron will talk about in his course[1]

Some of the advantages/drawbacks:

- **Discriminative:** much easier to design (and usually train) because we don't have to model $p(\mathbf{x})$. Usually more accurate where we have a lot of data. **Cannot acommodate to missing features or do semi-supervised learning (missing labels) easily.**
- **Generative:** Can deal with missing features/labels. Usually more accurate when we do not have a lot of data. Usually more robust to adversarial examples. **Requires to specify $p(\mathbf{x})$ which is often hard because $\mathbf{x}$ may be high-dimensional/complex.**

[1] cf. his book with G. Celeux, B. Murphy et A. Raftery.

Generative vs Discriminative: a concrete example



Decanter

Police uncover Italian wine fraud

Maggie Rosen
August 23, 2007

3 shares

Police have broken up an international counterfeit wine racket involving top Italian wines.

German and Italian police forces have uncovered a cross-border scam involving table wine from Puglia and Piedmont sold as Barolo, Brunello di Montalcino, Amarone and Chianti.

Unlabelled wine was brought into Germany, where it was given fake DOC and DOCG seals and phoney labels from well-known producers as well as non-existent wineries.

Ten people have been charged. It is thought the con could be worth €750,000.

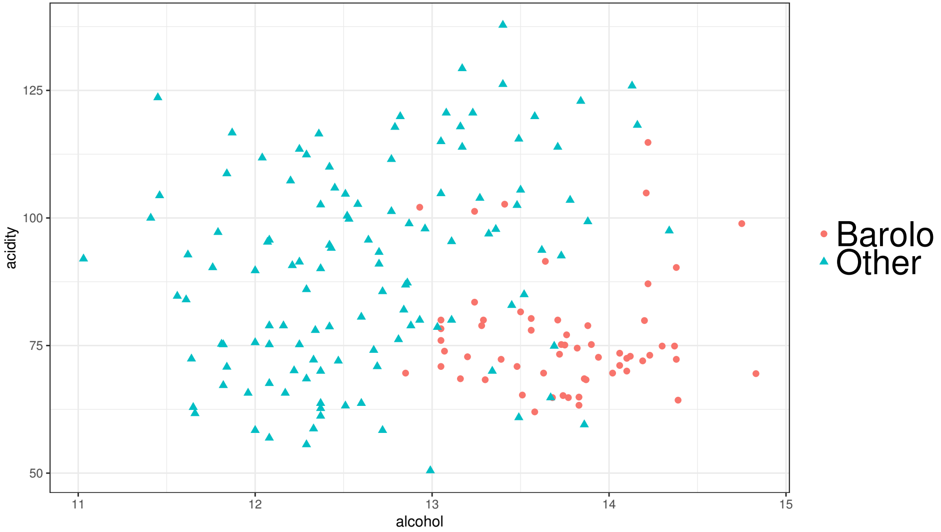
Article from `decanter.com/wine-news/police-uncover-italian-wine-fraud-88060/`

Generative vs Discriminative: a concrete example

One of the wines the bad guys counterfeited was from the Barolo region. According to Wikipedia, those wines have "pronounced tannins and acidity", and "moderate to high alcohol levels (Minimum 13%)". This would help a trained human recognise them, but **could we train an algorithm to learn those characteristics?**

Picture from Wikipedia

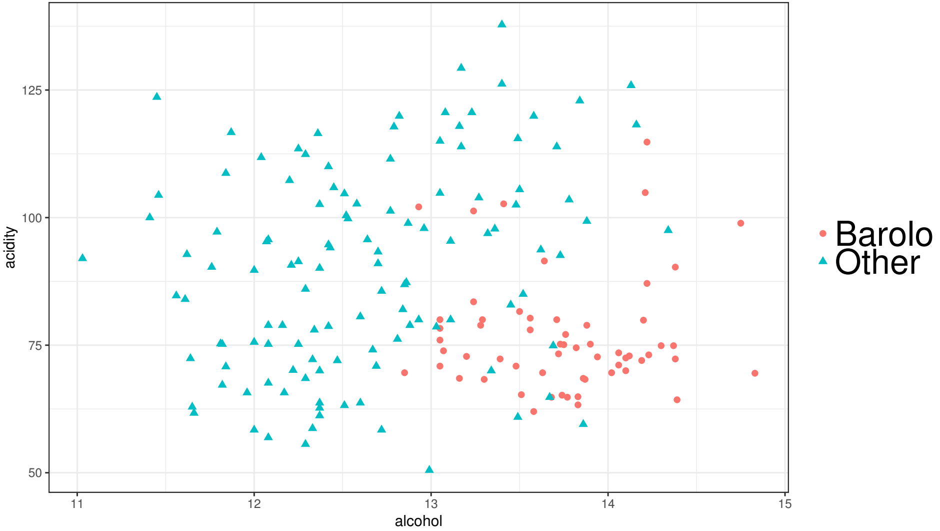
Generative vs Discriminative: a concrete example



Data from Forina, Armanino, Castino, and Ubigli, (Vitis, 1986).

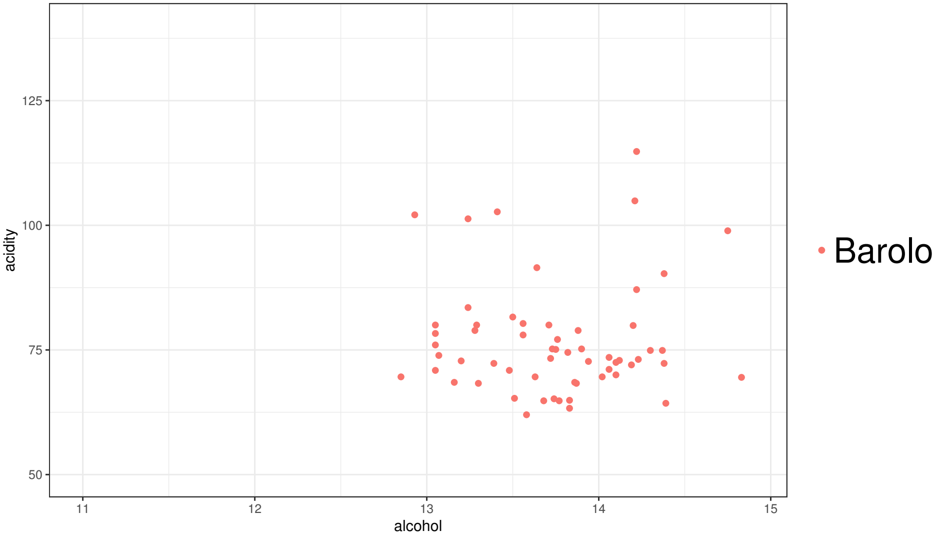
Generative vs Discriminative: a concrete example

The **generative** way would use the formula $p(\mathbf{x}, y) = p(y)p(\mathbf{x}|y)$ and model the class conditional distributions $p(\mathbf{x}|y)$ using a continuous bivariate ditribution (e.g. 2D Gaussians).



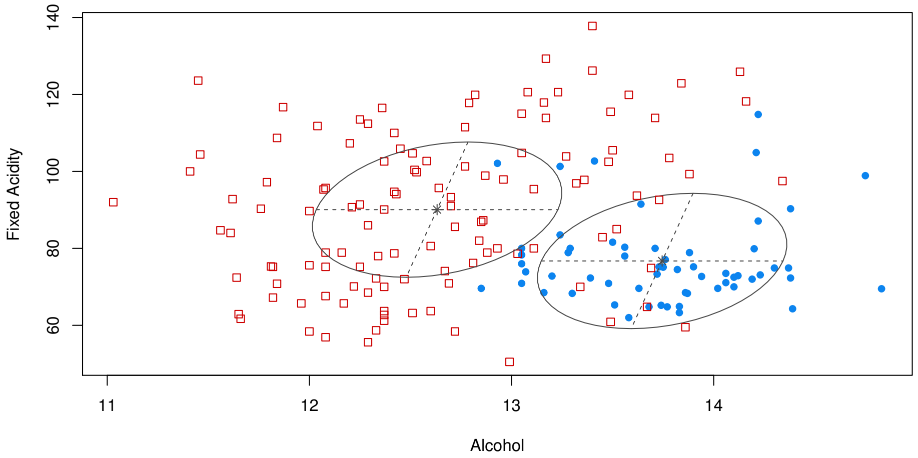
Generative vs Discriminative: a concrete example

The **generative** way would use the formula $p(\mathbf{x}, y) = p(y)p(\mathbf{x}|y)$ and model the class conditional distributions $p(\mathbf{x}|y)$ using a continuous bivariate ditribution (e.g. 2D Gaussians).



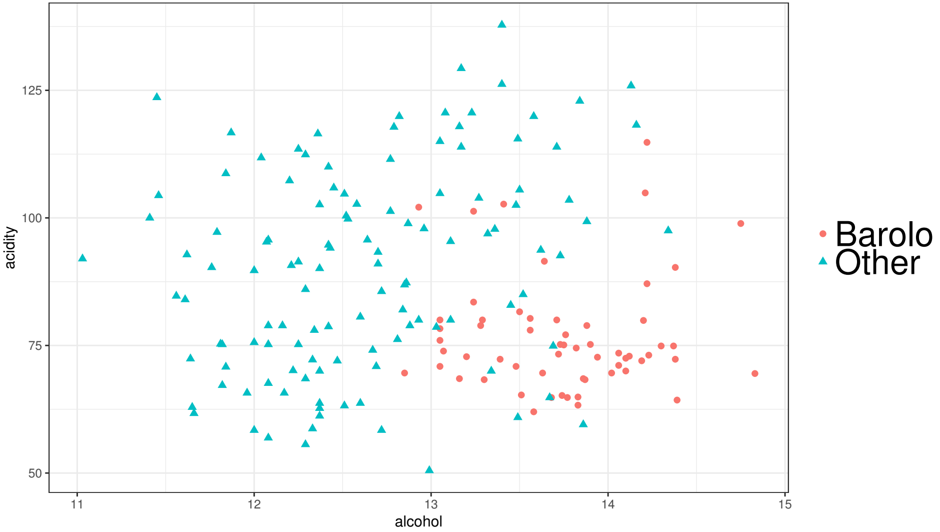
Generative vs Discriminative: a concrete example

The **generative** way would use the formula $p(\mathbf{x}, y) = p(y)p(\mathbf{x}|y)$ and model the class conditional distributions $p(\mathbf{x}|y)$ using a continuous bivariate ditribution (e.g. 2D Gaussians). Here is what we obtain using the R package `Mclust` (Scrucca, Fop, Murphy, and Raftery, R Journal, 2016).



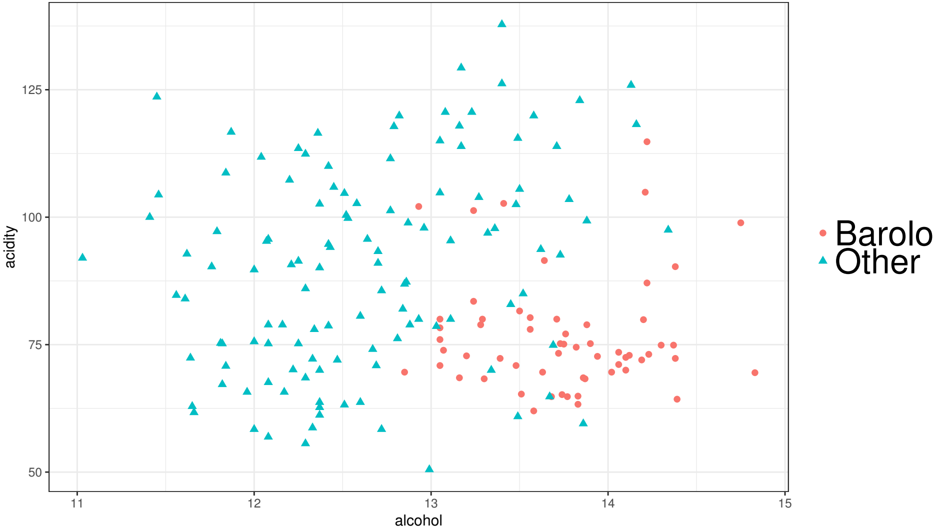
Generative vs Discriminative: a concrete example

The **discriminative** way would only model $p(y|\mathbf{x})$. Since there are only 2 classes, this means that $p(y|\mathbf{x})$ will be a **Bernoulli random variable whose parameter $\pi(\mathbf{x}) \in [0,1]$ is a function of the features.**



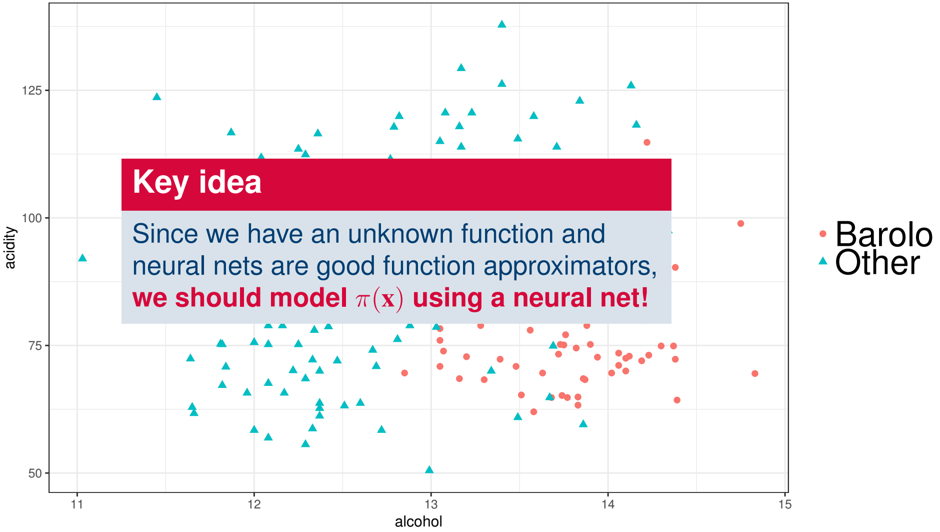
Generative vs Discriminative: a concrete example

The **discriminative** way would only model $p(y|\mathbf{x})$. Since there are only 2 classes, this means that $p(y|\mathbf{x})$ will be a **Bernoulli random variable whose parameter $\pi(\mathbf{x}) \in [0, 1]$ is a function of the features.**



Generative vs Discriminative: a concrete example

The **discriminative** way would only model $p(y|\mathbf{x})$. Since there are only 2 classes, this means that $p(y|\mathbf{x})$ will be a **Bernoulli random variable whose parameter $\pi(\mathbf{x}) \in [0, 1]$ is a function of the features.**



Key idea

Since we have an unknown function and neural nets are good function approximators, **we should model $\pi(\mathbf{x})$ using a neural net!**

Generative vs Discriminative: last words

We'll **focus now on the discriminative approach using neural nets**, because it is simpler. For more on the differences and links between the generative and discriminative schools, a wonderful reference is Tom Minka's short note on the subject: *Discriminative models, not discriminative training* [2].

[2] https://tminka.github.io/papers/minka-discriminative.pdf

Generative vs Discriminative: last words

We'll **focus now on the discriminative approach using neural nets**, because it is simpler. For more on the differences and links between the generative and discriminative schools, a wonderful reference is Tom Minka's short note on the subject: *Discriminative models, not discriminative training* [2].

Let us go back to our discriminative model: we can write it

$$p(y|\mathbf{x}) = \mathcal{B}(y|\pi(\mathbf{x})) = \pi(\mathbf{x})^y (1-\pi(\mathbf{x}))^{1-y},$$

where $\mathcal{B}(\cdot|\theta)$ denotes the density of a Bernoulli distribution with parameter $\theta \in [0,1]$. The key idea is then to **model the function $\mathbf{x} \mapsto \pi(\mathbf{x})$ using a neural net.**

[2] https://tminka.github.io/papers/minka-discriminative.pdf

Generative vs Discriminative: last words

We'll **focus now on the discriminative approach using neural nets**, because it is simpler. For more on the differences and links between the generative and discriminative schools, a wonderful reference is Tom Minka's short note on the subject: *Discriminative models, not discriminative training* [2].

Let us go back to our discriminative model: we can write it

$$p(y|\mathbf{x}) = \mathcal{B}(y|\pi(\mathbf{x})) = \pi(\mathbf{x})^y(1-\pi(\mathbf{x}))^{1-y},$$

where $\mathcal{B}(\cdot|\theta)$ denotes the density of a Bernoulli distribution with parameter $\theta \in [0,1]$. The key idea is then to **model the function $\mathbf{x} \mapsto \pi(\mathbf{x})$ using a neural net.**

This key idea goes way beyond the discriminative context

This general strategy of **using outputs of neural nets as parameters of simple probability distributions is the main recipe for building deep generative models.** It has been used extensively, for example in deep latent variable models such as variational autoencoders (VAEs) or generative adversarial networks (GANs).

[2] https://tminka.github.io/papers/minka-discriminative.pdf

How to model π

Our discriminative model for binary classification is

$$p(y|\mathbf{x}) = \mathcal{B}(y|\pi(\mathbf{x})) = \pi(\mathbf{x})^y(1 - \pi(\mathbf{x}))^{1-y},$$

and we wish to model π using a neural net. **But what kind of neural net?**

How to model π

Our discriminative model for binary classification is

$$p(y|\mathbf{x}) = \mathcal{B}(y|\pi(\mathbf{x})) = \pi(\mathbf{x})^y(1-\pi(\mathbf{x}))^{1-y},$$

and we wish to model π using a neural net. **But what kind of neural net?**

The only really important constraint of the problem is that we need to have

$$\forall \mathbf{x} \in \mathbb{R}^p,\ \pi(\mathbf{x}) \in [0,1].$$

Is it possible to enforce that easily in a neural net?

How to model π

Our discriminative model for binary classification is

$$p(y|\mathbf{x}) = \mathcal{B}(y|\pi(\mathbf{x})) = \pi(\mathbf{x})^y(1-\pi(\mathbf{x}))^{1-y},$$

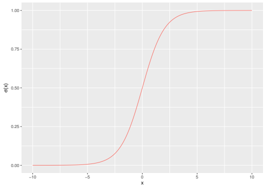
and we wish to model π using a neural net. **But what kind of neural net?**

The only really important constraint of the problem is that we need to have

$$\forall \mathbf{x} \in \mathbb{R}^p,\ \pi(\mathbf{x}) \in [0,1].$$

Is it possible to enforce that easily in a neural net?

Yes! By using a function that only output stuff in $[0,1]$ as the output layer. For example the **logistic sigmoid function** $\sigma : a \mapsto \frac{1}{1+\exp(-a)}$.



How to model π

So at the end, we'll model π using the formula

$$\pi(\mathbf{x}) = \sigma(f_{\boldsymbol{\theta}}(\mathbf{x})),$$

where σ is the sigmoid function and $f_{\boldsymbol{\theta}} : \mathbb{R}^p \longrightarrow \mathbb{R}$ **is any neural network** (whose weights are stored in a vector $\boldsymbol{\theta}$) that takes the features as input and returns an unconstrained real number.

How to model π

So at the end, we'll model π using the formula

$$\pi(\mathbf{x}) = \sigma(f_{\boldsymbol{\theta}}(\mathbf{x})),$$

where σ is the sigmoid function and $f_{\boldsymbol{\theta}} : \mathbb{R}^p \longrightarrow \mathbb{R}$ **is any neural network** (whose weights are stored in a vector $\boldsymbol{\theta}$) that takes the features as input and returns an unconstrained real number.

We have a lot of flexibility to choose f_θ. In particular, if the features $\mathbf{x}_1, ..., \mathbf{x}_n$ are images, we could use a CNN. In the case of time-series, we could use a recurrent neural net. In the case of sets, we could use a deepsets architecture (Zaheer et al., NeurIPS 2017).

How to model π

So at the end, we'll model π using the formula

$$\pi(\mathbf{x}) = \sigma(f_{\boldsymbol{\theta}}(\mathbf{x})),$$

where σ is the sigmoid function and $f_{\boldsymbol{\theta}} : \mathbb{R}^p \longrightarrow \mathbb{R}$ **is any neural network** (whose weights are stored in a vector $\boldsymbol{\theta}$) that takes the features as input and returns an unconstrained real number.

We have a lot of flexibility to choose f_θ. In particular, if the features $\mathbf{x}_1, ..., \mathbf{x}_n$ are images, we could use a CNN. In the case of time-series, we could use a recurrent neural net. In the case of sets, we could use a deepsets architecture (Zaheer et al., NeurIPS 2017).

For the wine example, we could just take a small MLP

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_1 \text{tanh}(\mathbf{W}_0 \mathbf{x}_i + \mathbf{b}_0) + b_1.$$

How to model π

So at the end, we'll model π using the formula

$$\pi(\mathbf{x}) = \sigma(f_{\boldsymbol{\theta}}(\mathbf{x})),$$

where σ is the sigmoid function and $f_{\boldsymbol{\theta}} : \mathbb{R}^p \longrightarrow \mathbb{R}$ **is any neural network** (whose weights are stored in a vector $\boldsymbol{\theta}$) that takes the features as input and returns an unconstrained real number.

We have a lot of flexibility to choose f_θ. In particular, if the features $\mathbf{x}_1, ..., \mathbf{x}_n$ are images, we could use a CNN. In the case of time-series, we could use a recurrent neural net. In the case of sets, we could use a deepsets architecture (Zaheer et al., NeurIPS 2017).

For the wine example, we could just take a small MLP

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_1 \text{tanh}(\mathbf{W}_0 \mathbf{x}_i + \mathbf{b}_0) + b_1.$$

Since the function π and the model $p(y|\mathbf{x})$ now depend on some parameters $\boldsymbol{\theta}$, **we'll denote them by $\pi_{\boldsymbol{\theta}}$ and $p_{\boldsymbol{\theta}}(y|\mathbf{x})$ from now on.**

How to find θ

There are many ways to find good parameter values for a generative model. One could use Bayesian inference, score matching, the method of moments, adversarial training... Let us focus on one of the most traditional ways: **maximum likelihood.** The idea is to find a $\hat{\boldsymbol{\theta}}$ that maximises the **log-likelihood function** $\log p_{\boldsymbol{\theta}}(\mathcal{D})$**.**

How to find θ

There are many ways to find good parameter values for a generative model. One could use Bayesian inference, score matching, the method of moments, adversarial training... Let us focus on one of the most traditional ways: **maximum likelihood.** The idea is to find a $\hat{\theta}$ that maximises the **log-likelihood function** $\log p_{\boldsymbol{\theta}}(\mathcal{D})$**.**

In the discriminative case, the likelihood is:

$$\log p_{\boldsymbol{\theta}}(\mathcal{D}) = \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(y_i, \mathbf{x}_i) = \sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(y_i|\mathbf{x}_i) + \sum_{i=1}^{N} \log p(\mathbf{x}_i),$$

but, since we don't model $p(\mathbf{x})$, $\sum_{i=1}^{n} \log p(\mathbf{x}_i)$ is constant, and maximising $\ell(\boldsymbol{\theta})$ is equivalent to maximising

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(y_i|\mathbf{x}_i).$$

We'll also call $\ell(\boldsymbol{\theta})$ the likelihood (in fact, we'll call any function that is equal to $\log p_{\boldsymbol{\theta}}(\mathcal{D})$ up to a constant the likelihood).

How to find θ: from ML to XENT

We have

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(y_i|\mathbf{x}_i) = \sum_{i=1}^{N} \log \left(\pi(\mathbf{x})^{y_i}(1-\pi(\mathbf{x}))^{1-y_i}\right),$$

which leads to

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left[y_i \ln \pi(\mathbf{x}_i) + (1-y_i)\ln(1-\pi(\mathbf{x}_i))\right].$$

How to find θ: from ML to XENT

We have

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(y_i|\mathbf{x}_i) = \sum_{i=1}^{N} \log \left(\pi(\mathbf{x})^{y_i}(1-\pi(\mathbf{x}))^{1-y_i}\right),$$

which leads to

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left[y_i \ln \pi(\mathbf{x}_i) + (1-y_i)\ln(1-\pi(\mathbf{x}_i))\right].$$

We will want to maximise this function, which is equivalent to minimising its opposite, which is called the **cross-entropy loss**.
The cross-entropy loss is the most commonly used loss for neural networks, and is **a way of doing maximum likelihood without necessarily saying it.**