Artificial Intelligence and Deep Machine Learning for Fin-Techs

Cristian TOMA & Marius POPA

Cyber | ICT Security Master Program – ISM
Applied Computer Science and Cybersecurity R&D Team – ACS
Department of Economic Informatics & Cybernetics – DICE | DEIC
www.ism.ase.ro | www.acs.ase.ro | www.dice.ase.ro

Make Artificial Intelligence and Machine Learning Work for

Fin-Techs

Visit us at www.fintech-ho2020.eu

or contact as at info@fintech-ho2020.eu

or follow us on LinkedIn

The development of the materials has also been supported by the project **A FINancial supervision and TECHnology compliance training programme.**

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825215 (Topic: ICT-35-2018 Type of action: CSA)

Agenda for the Presentation

1. Fundamentals of Machine Learning
2. AI – Neural Networks & Deep Learning
3. Exchange Ideas

www.dice.ase.ro

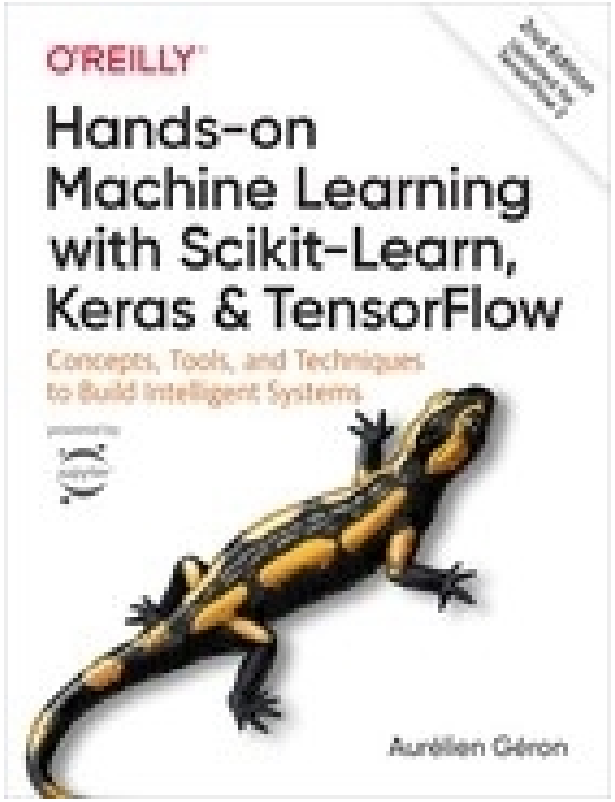
www.ism.ase.ro

Machine Learning Technology Overview: Classification, Training models, SVM, Decision Trees …

Machine Learning Fundamentals

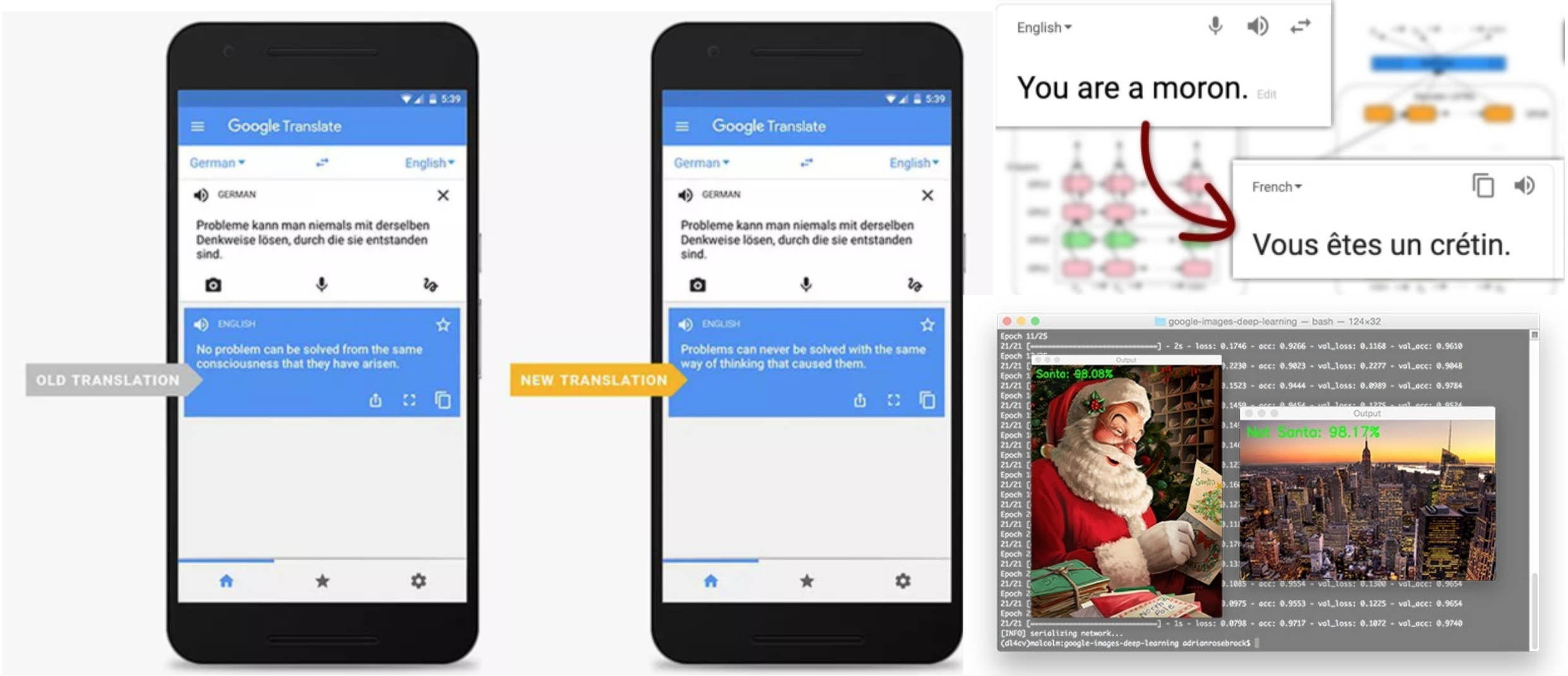
Artificial Intelligence & Machine Learning Overview

When most people hear "Machine Learning", they picture:



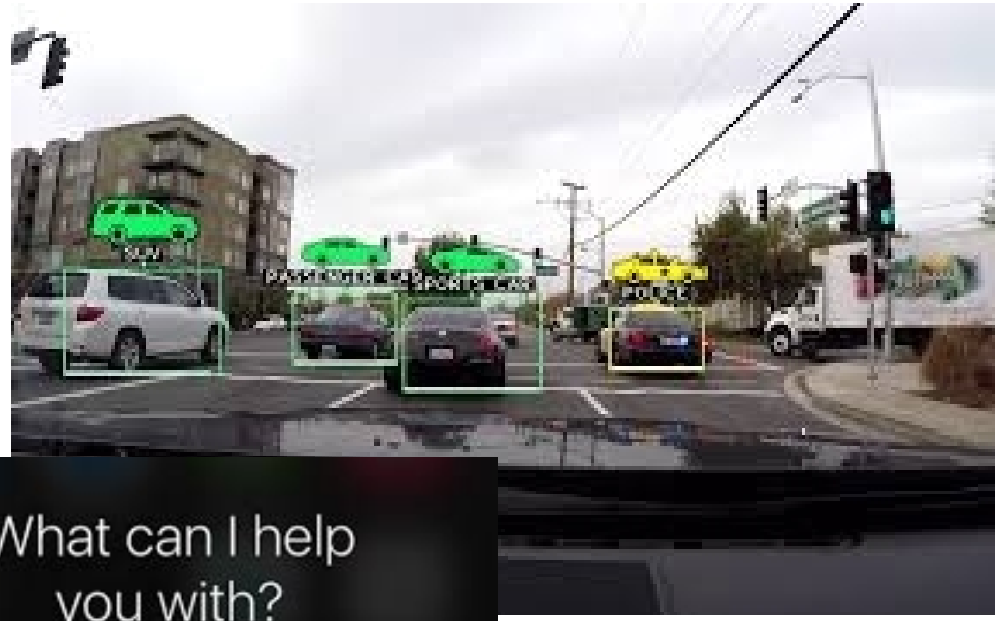
Artificial Intelligence & Machine Learning Overview

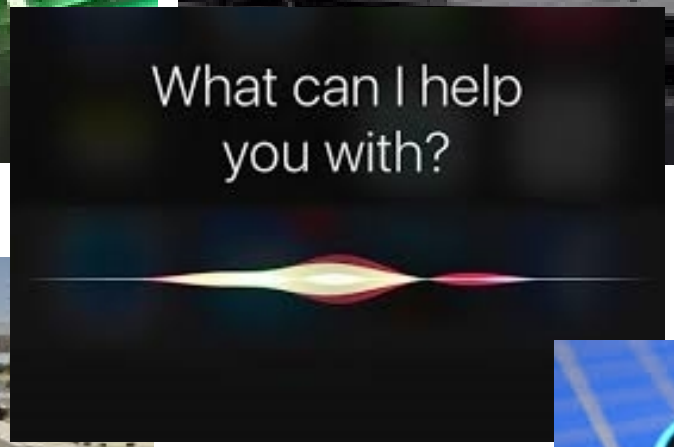
When most people hear "Machine Learning" / "Învățare automată", they picture:



Artificial Intelligence & Machine Learning Overview

When most people hear "Machine Learning," they picture?:





Artificial Intelligence & Machine Learning Overview

"But Machine Learning is not just a futuristic fantasy; it's already here. In fact, it has been around for decades in some specialized applications, such as Optical Character Recognition (OCR). But the first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: the *spam filter*. It's not exactly a self-aware Skynet, but it does technically qualify as Machine Learning (it has actually learned so well that you seldom need to flag an email as spam anymore). It was followed by hundreds of ML applications that now quietly power hundreds of products and features that you use regularly, from better recommendations to voice search."

Machine Learning is the science (and art) of programming computers so they can *learn from data*.

Here is a slightly more general definition:

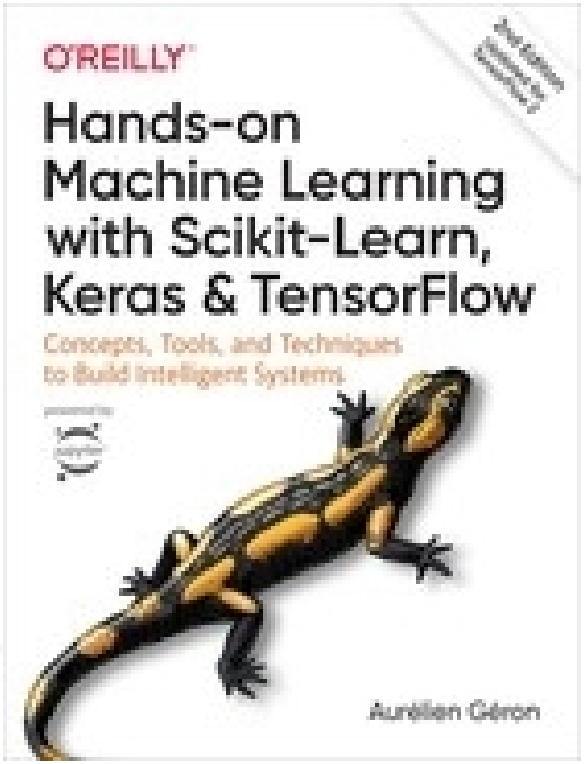
[Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.

Arthur Samuel, 1959

And a more engineering-oriented one:

A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

Tom Mitchell, 1997



Artificial Intelligence & Machine Learning Overview

"Where does Machine Learning start and where does it end?

What exactly does it mean for a machine to *learn* something?

If I download a copy of Wikipedia, has my computer really learned something?

Is it suddenly smarter?"

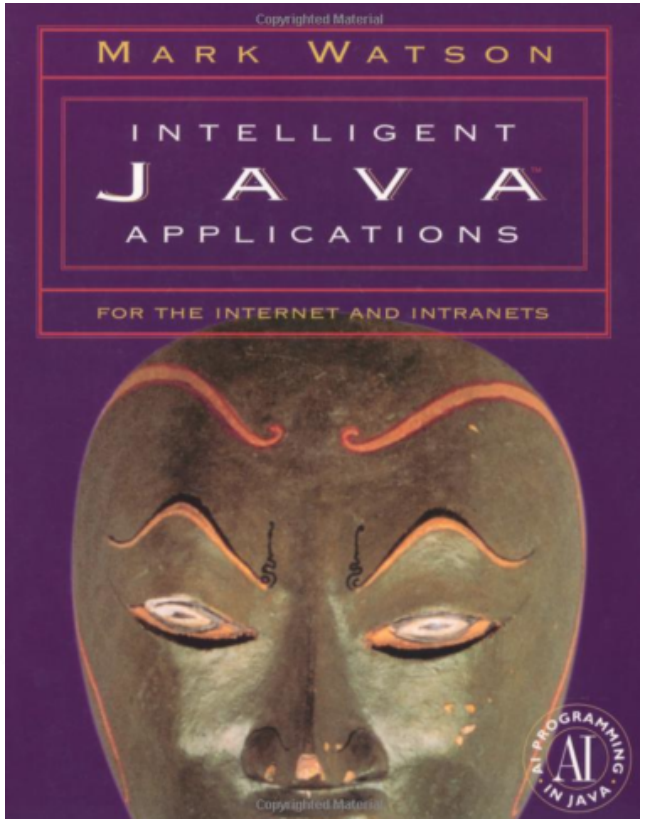
If you just download a copy of Wikipedia, your computer has a lot more data, but it is not suddenly better at any task. Thus, downloading a copy of Wikipedia is not Machine Learning.

Machine Learning is not just a futuristic fantasy; it's already here. In fact, it has been around for decades in some specialized applications, such as Optical Character Recognition (OCR). But the first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: the *spam filter*.

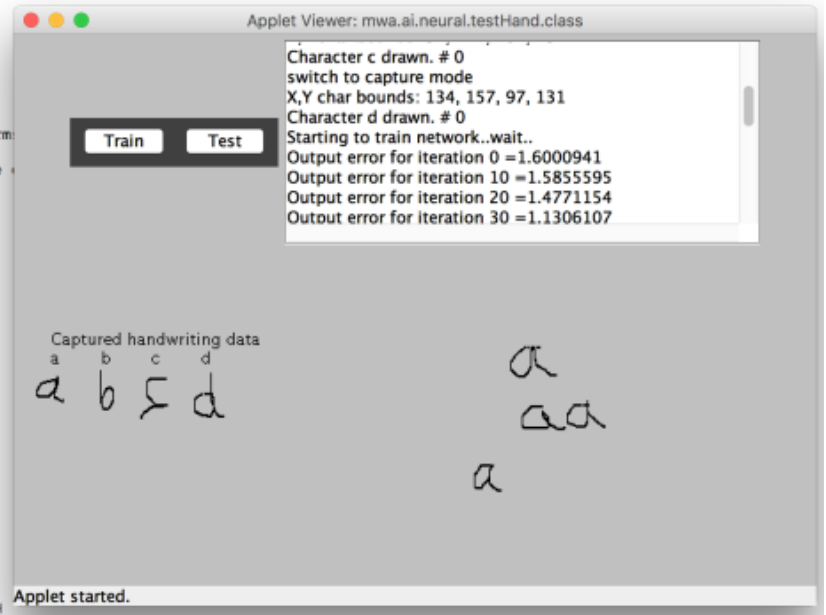
Your spam filter is a Machine Learning program that, given examples of spam emails (e.g., flagged by users) and examples of regular (nonspam, also called "ham") emails, can learn to flag spam. The examples that the system uses to learn are called the *training set*. Each training example is called a *training instance* (or *sample*). In this case, the task *T* is to flag spam for new emails, the experience *E* is the *training data*, and the performance measure *P* needs to be defined; for example, you can use the ratio of correctly classified emails. This particular performance measure is called *accuracy*, and it is often used in classification tasks.

DEMO 0 in Java

```
$ export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home
$ export PATH=.:$JAVA_HOME/bin:$PATH
$ appletviewer testHand_2.html
```



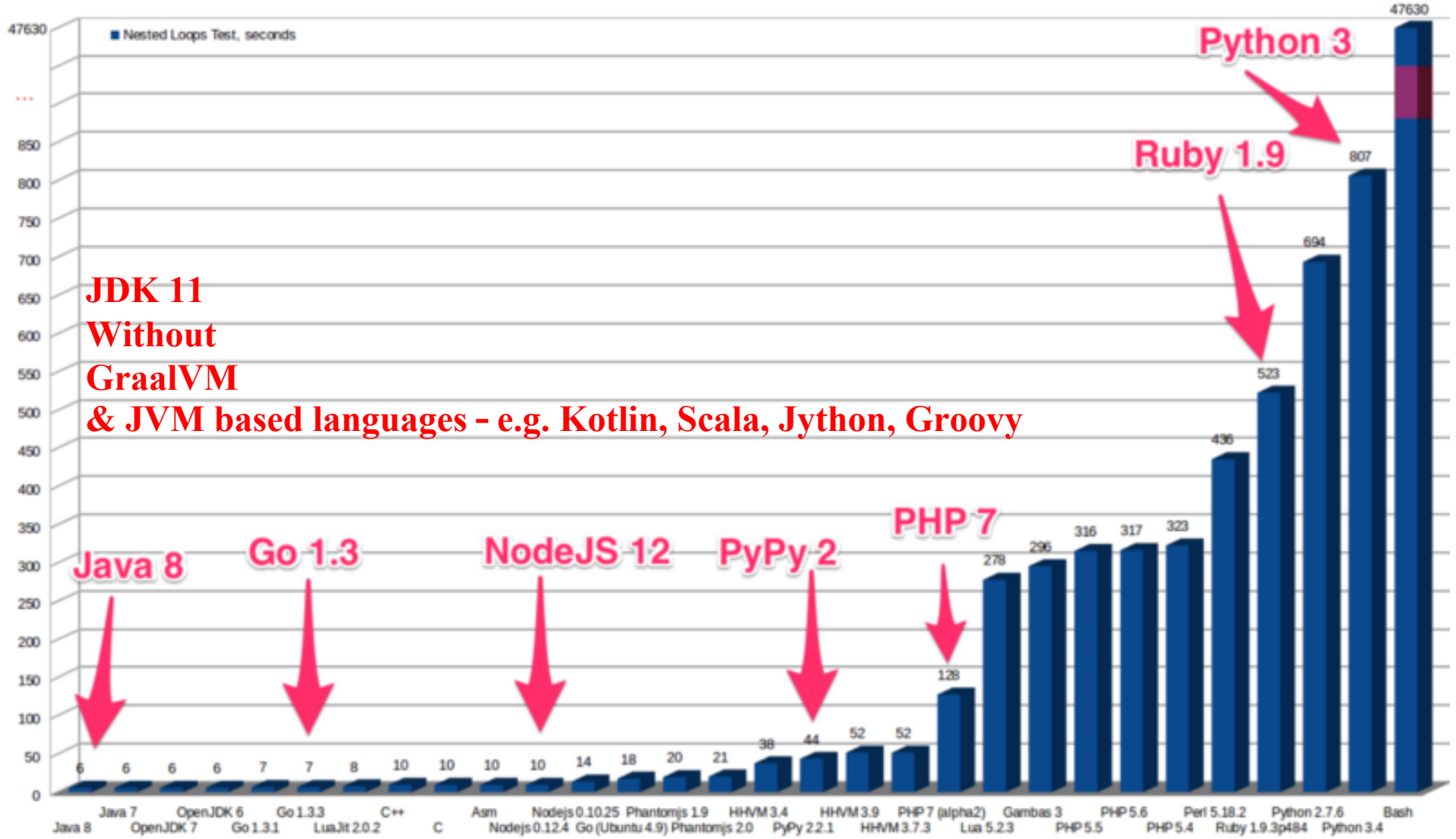

```
  {
    "cell_type": "markdown",
    "metadata": {
      "colab_type": "text",
      "id": "eTZpu6-mNG9j"
    },
    "source": [
      "## Hiding code\n",
      "[Forms example](https://colab.research.google.com/notebooks/form
      "\n",
      "The cell below will automatically load with code hidden. Double
    ]
  },
  {
    "cell_type": "code",
    "metadata": {
      "colab_type": "code",
      "id": "aXJaZqFpNK-q",
      "colab": {}
    },
    "source": [
      "#@title Give me a name {display-mode: \"form\"}\n",
      "\n",
      "# This code will be hidden when the notebook is loaded.\n"
    ],
    "execution_count": 0,
    "outputs": []
  }
]
Cristians-MacBook-Pro-2:JavaAI ctoma$ pwd
/Users/ctoma/Data/School/F0100_PhD/F0100_Doctorat/JavaAI
Cristians-MacBook-Pro-2:JavaAI ctoma$ appletviewer testHand_2.html
In BackProp constructor
Cristians-MacBook-Pro-2:JavaAI ctoma$ javac -version
javac 1.8.0_74
Cristians-MacBook-Pro-2:JavaAI ctoma$ export PATH=.:$JAVA_HOME/bin:$PATH
Cristians-MacBook-Pro-2:JavaAI ctoma$ echo $PATH
.:/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin:.:/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin:/usr/local/bin:/usr/bin:/b
Cristians-MacBook-Pro-2:JavaAI ctoma$ export PATH=.:/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
Cristians-MacBook-Pro-2:JavaAI ctoma$ echo $PATH
.:/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
Cristians-MacBook-Pro-2:JavaAI ctoma$ appletviewer testHand_2.html
In BackProp constructor
1001
Run button pressed

1001
Reset button pressed
```



ISBN-10: 1558604200, ISBN-13: 978-1558604209 | https://www.amazon.com/Intelligent-Java-Applications-Internet-Intranets/dp/1558604200

Why Python for ML learning (students) and Java/Kotlin/Scala or C/C++ for Back-end Production?

Speed Benchmarking



Are AI/ML/NN algorithms DETERMINISTIC or STOCHASTIC?

Google
&
https://www.quora.com/Are-neural-networks-stochastic-or-deterministic

"Thus once the weights and the structure of a Neural Network are fixed (let's say once it has been trained), it becomes a deterministic function." …

"it's just that the training procedure is stochastic."

Is AI deterministic?

Deterministic AI environments are those on which the outcome can be determined base on a specific state. In other words, **deterministic** environments ignore uncertainty. Most real world **AI** environments are not **deterministic**. Instead, they can be classified as stochastic. Jan 12, 2017

medium.com › 6-types-of-artificial-intelligence-environments-825e3c47...

6 Types of Artificial Intelligence Environments - Jesus Rodriguez ...

Search for: Is AI deterministic?

Is machine learning non deterministic?

Machine learning is stochastic, not **deterministic**. Jul 29, 2019

towardsdatascience.com › the-limitations-of-machine-learning-a00e0c30...

The Limitations of Machine Learning - Towards Data Science

Search for: Is machine learning non deterministic?

Can artificial intelligence have free will?

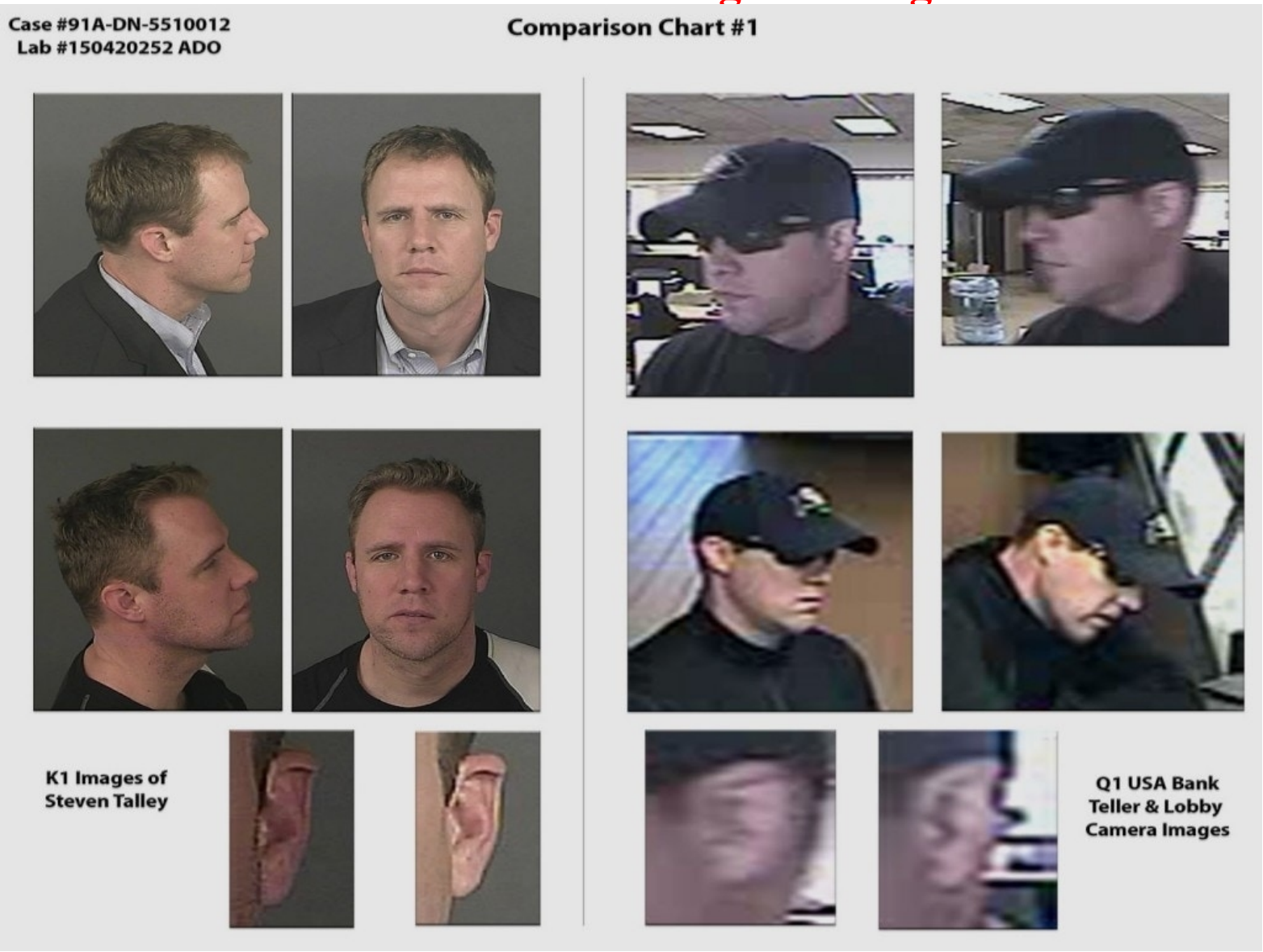
In order for robots to rule human beings, they **will need** to possess the autonomy to take decisions by themselves. They should be able to make their own choice consciously and take initiative. And for all these, they should **have** "**free will**". Dec 28, 2017

beytulhikme.org › Makaleler › 1657575238_05_Cevik_(75-87)

Will It Be Possible for Artificial Intelligence Robots to Acquire Free ...

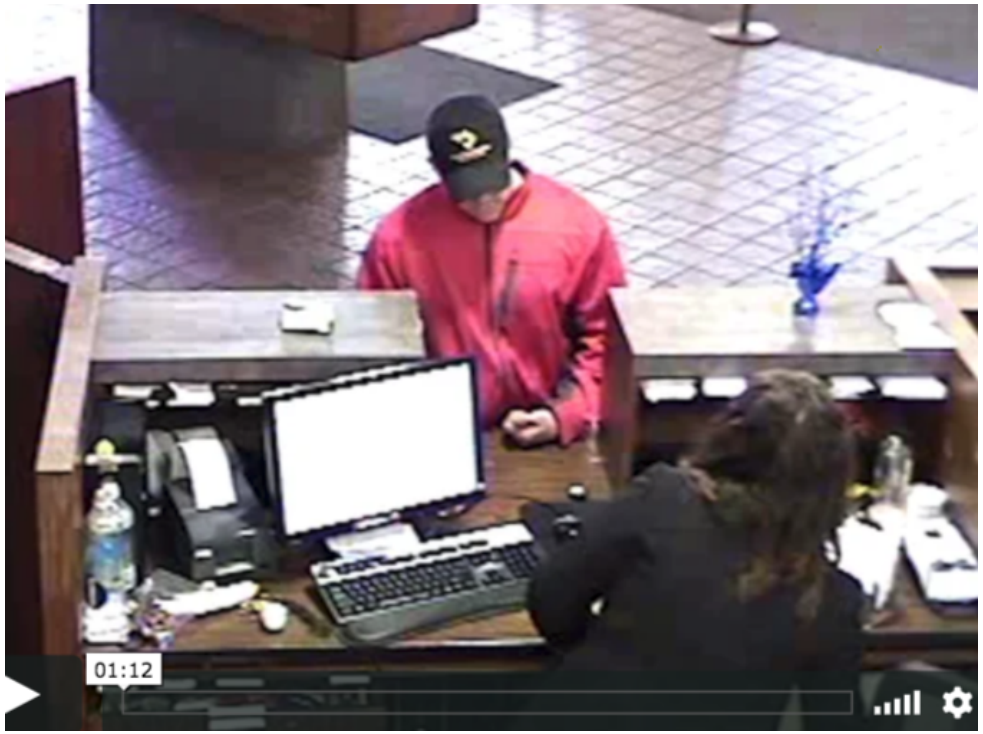
Artificial Intelligence / Neural Networks & Machine Learning

Are AI/ML/NN/Face Recognition algorithms DETERMINISTIC or STOCHASTIC?



https://theintercept.com/2016/10/13/how-a-facial-recognition-mismatch-can-ruin-your-life/

TEVE TALLEY IS hardly the first person to be arrested for the errors of a forensic evaluation.



What is a Fin-Tech? FORBES: Examples of Fin-Techs

What are examples of Fintech companies?

Examples of Fintech-related companies or products include:

- Payment infrastructure, processing and issuance, such as services provided by Square, Ant Financial, Revolut, and Stripe.
- Stock trading apps from Robinhood, TD Ameritrade, and Schwab.
- Alternative lending marketplaces, such as Prosper, LendingClub, and OnDeck.

More items... • Oct 12, 2019

www.forbes.com › sites › allbusiness › 2019/10/12 › fintech-startup-co...

10 Key Issues For Fintech Startup Companies - Forbes

What is a Fin-Tech? Examples of Fin-Techs

According to investopedia.com

View 2+ more

Ant Financial | Adyen | Qudian | Xero | SoFi | Lufax | Avant

The World's Top 10 FinTech Companies

- Ant Financial.
- Adyen.
- Qudian.
- Xero.
- SoFi.
- Lufax.
- Avant.
- ZhongAn.

More items... • Oct 13, 2019

www.investopedia.com › Investing › Stocks › Tech Stocks

The World's Top 10 FinTech Companies - Investopedia

This is the latest accepted revision, reviewed on 21 January 2020. **Revolut** Ltd is a UK financial technology company and neobank. Its services include a prepaid debit card (MasterCard or Visa), fee free currency exchange, commission free stock trading, cryptocurrency exchange and peer-to-peer payments.

Services: Peer-to-peer payments, Currency Ex...

Products: Current Accounts, Debit Cards, Insur...

Headquarters: London, England, UK

en.wikipedia.org › wiki › Revolut

Revolut - Wikipedia

What is a Fin-Tech? Examples of Fin-Techs

Fintech companies encompass a broad landscape of businesses, generally around financial-oriented services and products. ***Examples of Fintech-related companies*** or products include:

- **Payment infrastructure**, processing and issuance, such as services provided by Square, Ant Financial, Revolut, and Stripe
- Stock trading apps from Robinhood, TD Ameritrade, and Schwab
- Alternative lending marketplaces, such as Prosper, LendingClub, and OnDeck
- **Cryptocurrencies and digital cash**, a prime example of which is Bitcoin
- **Blockchain technology**, such as Ethereum
- **Insurtech**, which seeks to modernize and simplify the insurance industry, with companies such as Lemonade, Oscar, and Fabric
- **Money transfer and remittances**, including services from TransferWise, **PayPal**, and Venmo
- **Mortgage lending**, such as through LendingHome and Better Mortgage
- **Robo investment advisors**, such as Betterment and Wealthfront
- **Neobanks**, including Chime, N26, and Monzo
- Credit reporting, such as Credit Karma
- Online business loan providers such as Lendio and Kabbage
- Small business credit cards, payments, and financing, such as through Brex and Fundbox
- **Financial cybersecurity companies** seeking to protect institutions from money laundering, chargeback risk. and cybercrimes, such as Forter, EverCompliant, and CrowdStrike.
- **Infrastructure and software to power financial applications**, such as from **Plaid**

Top 10 AI / ML Applications for Finance & Fin-Techs

1. Portfolio Management – Robo-Advisors

2. Algorithmic Trading

3. High-Frequency Trading (HFT)

4. Fraud Detection

5. Loan/ Insurance Underwriting

6. Risk Management

7. Chatbots

8. Document Analysis
9. Trade Settlements
10. Money-Laundering Prevention

… Future Applications of Artificial Intelligence in Finance

Top 10 AI / ML Applications for Fin-Techs

#1. Digital Financial Coach/Advisor

Transactional bots are one of the most popular use cases in AI, probably because the range of applications is so broad — across all industries, at several levels.

#2. Transaction search & visualization

Chat-bots can also be used in banking to focus on search tasks.
Managers give access to the bot to the users' transactional data (banking transactions), and it uses NLP to detect the meaning of the request sent by the user (a search query). Requests could be related to balance inquiries, spending habits, general account information and more. The bot then processes the requests and displays the results.

#3. Client Risk Profile

A critical part of banks and insurance companies' job is the profiling of clients based on their risk score.
AI is an excellent tool for this as it can automate the categorization of clients depending on their risk profile, from low to high.

Top 10 AI / ML Applications for Fin-Techs

#4. Underwriting, Pricing & Credit Risk Assessment

Insurance companies offer underwriting services, mainly for loans and investments.
An AI-powered model can provide an instantaneous assessment of a client's credit risk, which then allows advisors to craft the most adapted offer.

#5. Automated Claims Processes

The insurance industry as we know it functions on a standard process: clients subscribe insurance, for which they pay. If the customer has a problem (sickness for health insurance, a car accident for automobile insurance, water damage for a housing insurance), she needs to activate her coverage by filing a claim. This process is often lengthy and complicated.
Transactional bots can transform the user experience into a more pleasant process.

#6. Contract Analyzer

Contract analysis is a repetitive internal task in the finance industry. Managers and advisors can delegate this routine task to a machine learning model.
Optical Character Recognition (OCR) can be used to digitize hard copy documents. An NLP model with layered business logic can then interpret, record, and correct contracts at high speed.

Top 10 AI / ML Applications for Fin-Techs

#7. Churn Prediction

Churn (or attrition) rate is a key KPI across all industries and businesses. Companies need to retain clients, and to do so, predicting coming churn can be extremely helpful to take preventive actions.
AI can support managers in this mission by providing a prioritized list of clients who show signs of considering to cancel their policy. The manager can then address this list accordingly: give a higher degree of service or improved offering.

#8. Algorithmic Trading — the most advanced ML you will never see.

Most applications of algorithmic trading happen behind the closed doors of investment banks or hedge funds. **Trading, very often, comes to analyzing data and making decisions, fast. Machine learning algorithm excels in analyzing data, whatever its size and density.**

#9. Valuation Models

Valuation models are usually applications for investment and banking in general.
The model can quickly calculate the valuation of an asset using data points around the asset and historical examples. These data points are what a human would use to value the asset (ex: the creator of a painting), but the model learns which weights to assign to each data point by using historical data.

Top 10 AI / ML Applications for Fin-Techs

#10. Augmented research tools

In investment finance, a large portion of time is spent doing research. New machine learning models increase the available data around given trade ideas.

Sentiment analysis can be used for due diligence about companies and managers. It allows an analyst to view at a glance the tone/mood of large sets of text data such as news or financial reviews. It can also provide insight into how a manager reflects their company performance.

Satellite Image Recognition can give a researcher insight into many real-time data points. Examples of such are parking lot traffic in specific locations (retailer shops, for example) or freighter traffic in the ocean. From this data, the model and the analyst can derive business insights such as the frequency of shopping at specific stores of the retailers mentioned above, the flow of shipments, routes, and so on. Advanced **NLP** techniques can help a researcher analyze a company financial reports quickly. Pulling out key topics that are of most interest to the firm.

Other data science techniques can also format and standardize financial statements.

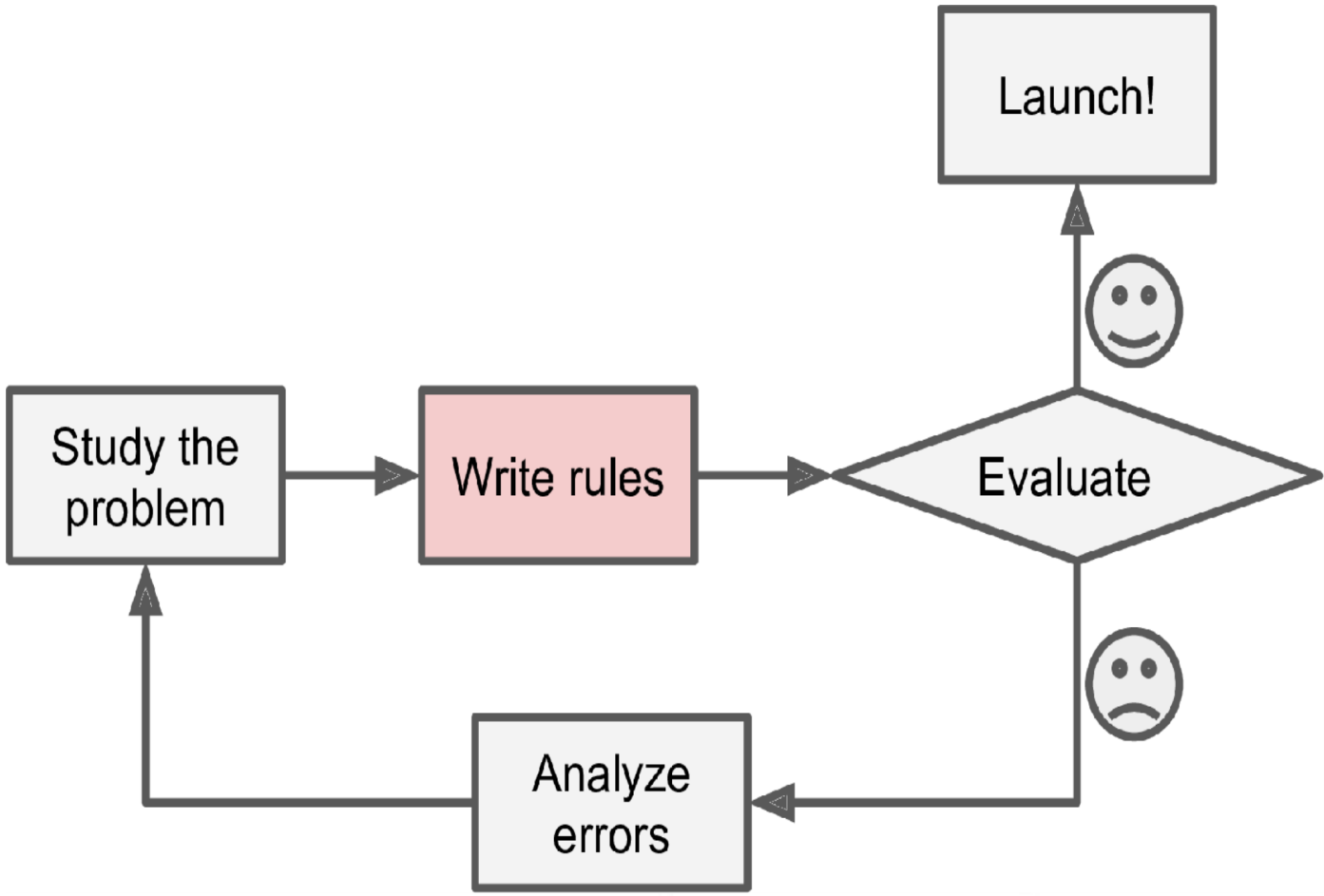
Artificial Intelligence & Machine Learning Overview

Why Use Machine Learning?

Consider how you would write a spam filter using traditional programming techniques:

First you would consider what spam typically looks like. You might notice that some words or phrases (such as “4U,” “credit card,” “free,” and “amazing”) tend to come up a lot in the subject line. Perhaps you would also notice a few other patterns in the sender’s name, the email’s body, and other parts of the email.

You would write a detection algorithm for each of the patterns that you noticed, and your program would flag emails as spam if a number of these patterns were detected. You would test your program and repeat steps 1 and 2 until it was good enough to launch. Since the problem is difficult, your program will likely become a long list of complex rules—pretty hard to maintain.



The traditional approach

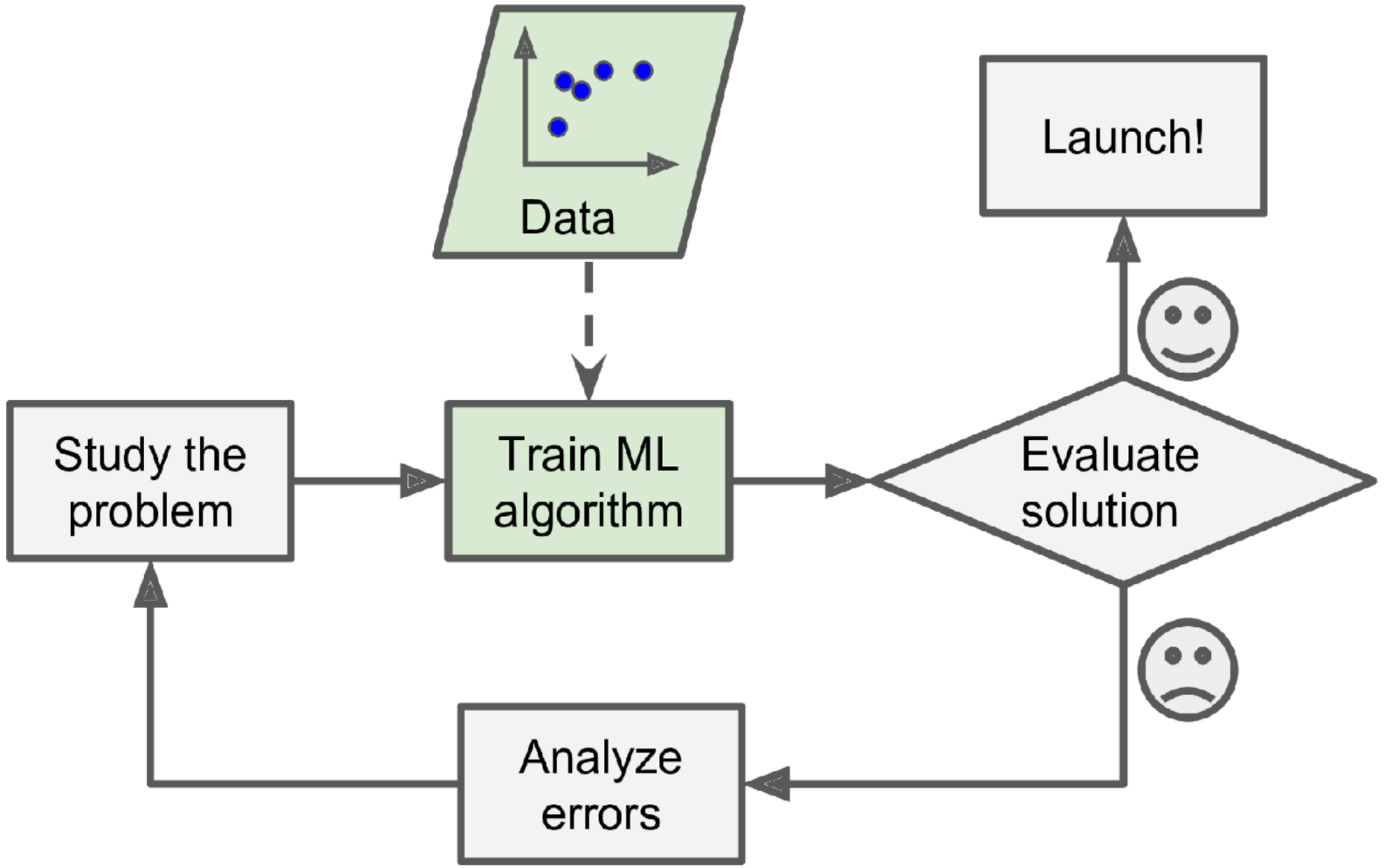
Artificial Intelligence & Machine Learning Overview

In contrast, a spam filter based on Machine Learning techniques automatically learns which words and phrases are good predictors of spam by detecting unusually frequent patterns of words in the spam examples compared to the ham examples – fig. The program is much shorter, easier to maintain, and most likely more accurate.

What if spammers notice that all their emails containing "4U" are blocked? They might start writing "For U" instead.

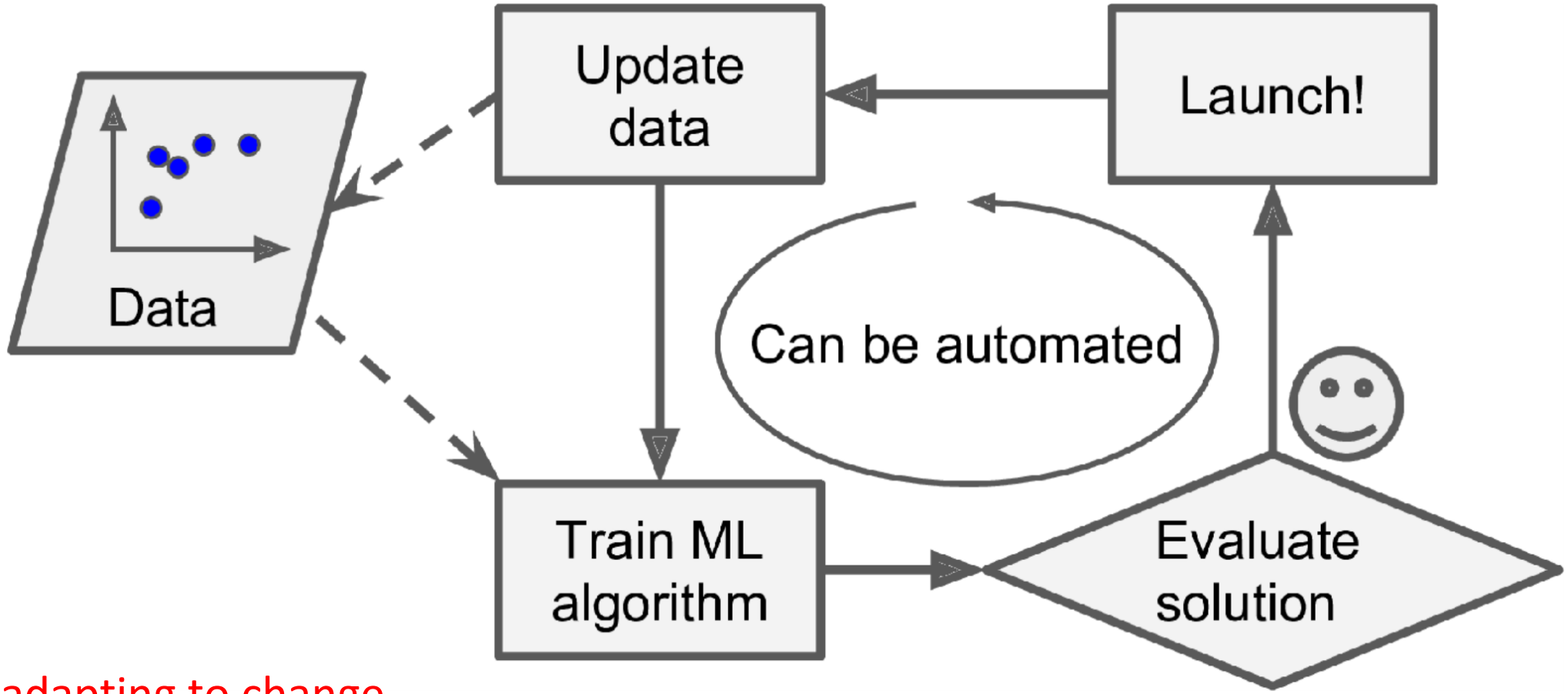
A spam filter using traditional programming techniques would need to be updated to flag "For U" emails. If spammers keep working around your spam filter, you will need to keep writing new rules forever.

The Machine Learning approach



Artificial Intelligence & Machine Learning Overview

In contrast, a spam filter based on Machine Learning techniques automatically notices that “For U” has become unusually frequent in spam flagged by users, and it starts flagging them without your intervention:



ML Automatically adapting to change

Artificial Intelligence & Machine Learning Overview

Another area where Machine Learning shines is for problems that either are too complex for traditional approaches or have no known algorithm.

For example, consider speech recognition / face recognition.

Say you want to start simple and write a program capable of distinguishing the words “one” and “two.” You might notice that the word “two” starts with a high-pitch sound (“T”), so you could hardcode an algorithm that measures high-pitch sound intensity and use that to distinguish ones and twos—but obviously this technique will not scale to thousands of words spoken by millions of very different people in noisy environments and in dozens of languages.

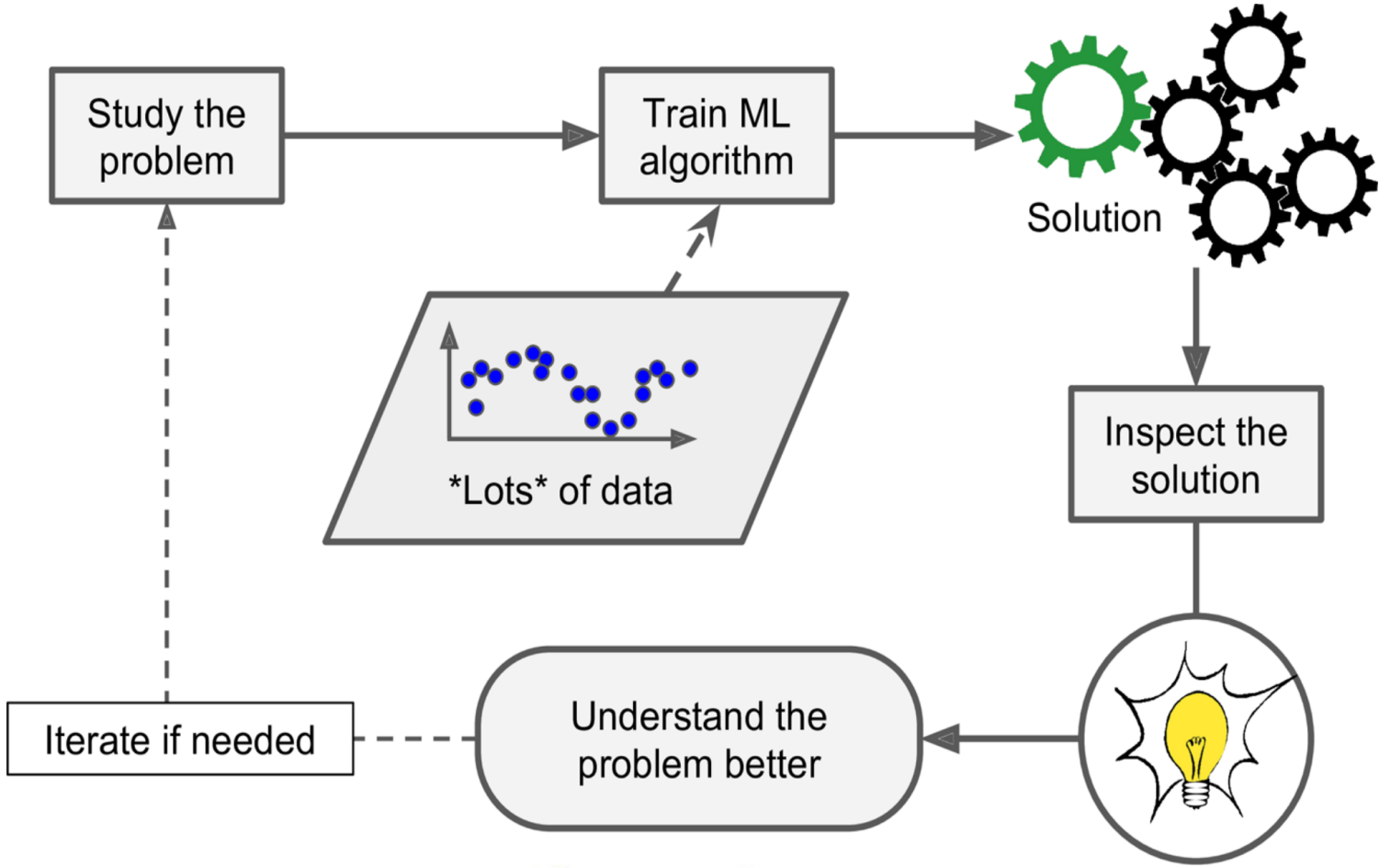
The best solution (at least today) is to write an algorithm that learns by itself, given many example recordings for each word.

Artificial Intelligence & Machine Learning Overview

Finally, Machine Learning can help humans learn – figure.

ML algorithms can be inspected to see what they have learned (although for some algorithms this can be tricky). For instance, once a spam filter has been trained on enough spam, it can easily be inspected to reveal the list of words and combinations of words that it believes are the best predictors of spam.

Sometimes, this will reveal unsuspected correlations or new trends, and thereby lead to a better understanding of the problem. Applying ML techniques to dig into large amounts of data can help discover patterns that were not immediately apparent. This is called ***data mining***.



Artificial Intelligence & Machine Learning Overview

To summarize, Machine Learning is great for:

- ***Problems for which existing solutions require a lot of fine-tuning or long lists of rules*: one Machine Learning algorithm can often simplify code and perform better than the traditional approach.**
- ***Complex problems for which using a traditional approach yields no good solution*: the best Machine Learning techniques can perhaps find a solution.**
- ***Fluctuating environments*: a Machine Learning system can adapt to new data.**
- *Getting insights about complex problems and large amounts of data.*

Artificial Intelligence & Machine Learning Systems Types

There are so many different types of Machine Learning systems that it is useful to classify them in broad categories, based on the following criteria:

- A. Whether or not they are trained with human supervision (Supervised, Unsupervised, Semi-supervised, and Reinforcement Learning)

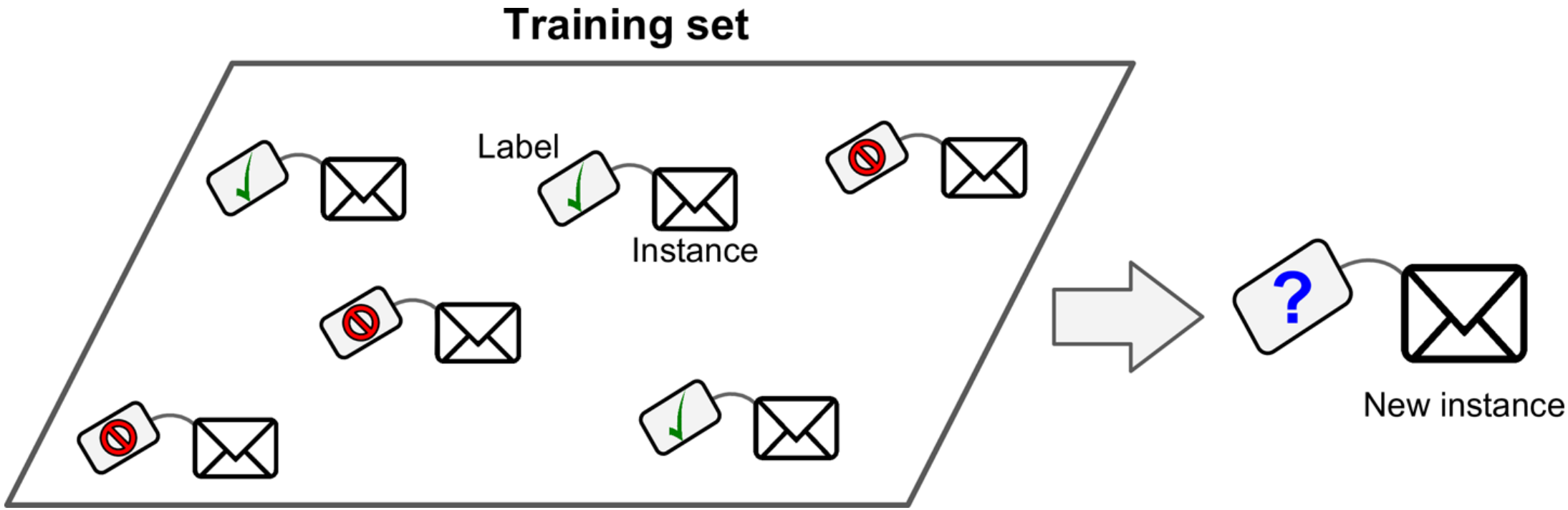
- B. Whether or not they can learn incrementally on the fly (Online versus Batch learning)

- C. Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model, much like scientists do (Instance-based versus Model-based learning)

For example, a state-of-the-art spam filter may learn on the fly using a deep neural network model trained using examples of spam and ham; this makes it an online, model-based, supervised learning system.

A.1 Supervised Learning

In supervised learning, the training set you feed to the algorithm includes the desired solutions, called labels

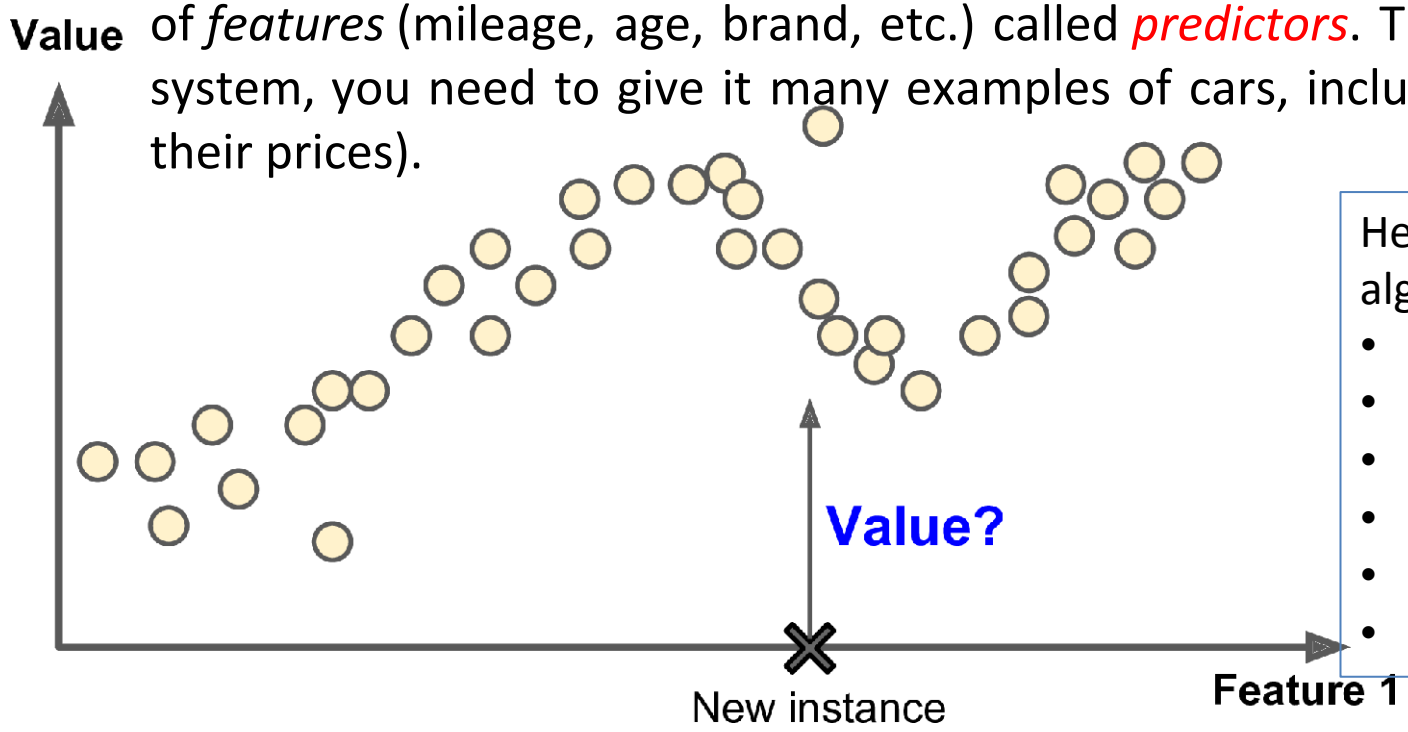


A labeled training set for spam classification (an example of supervised learning)

A.1 Supervised Learning

A typical supervised learning task is *classification*. The spam filter is a good example of this: it is trained with many example emails along with their *class* (spam or ham), and it must learn how to classify new emails.

Another typical task is to predict a *target* numeric value, such as the price of a car, given a set of *features* (mileage, age, brand, etc.) called *predictors*. This sort of task is called *regression*. To train the system, you need to give it many examples of cars, including both their predictors and their labels (i.e., their prices).



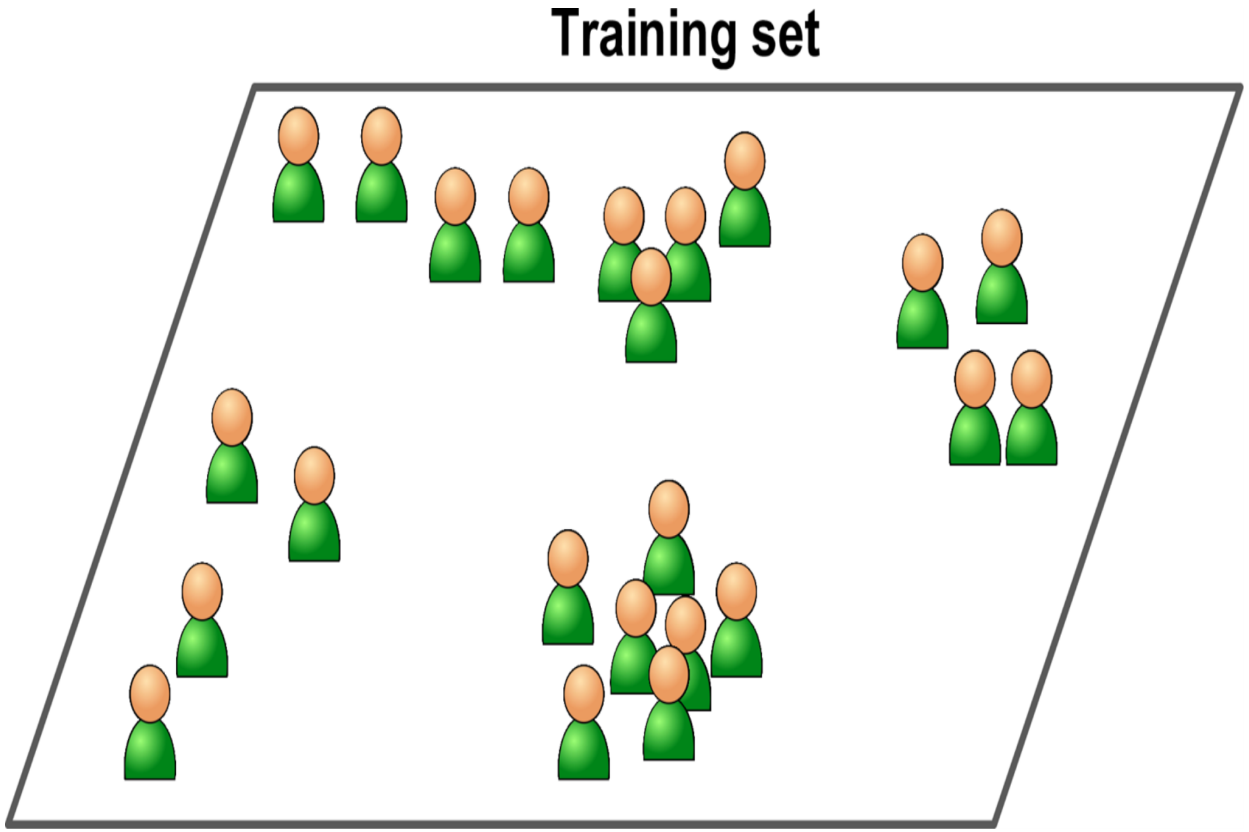
Here are some of the most important supervised learning algorithms:

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural networks

A regression problem: predict a value, given an input feature (there are usually multiple input features, and sometimes multiple output values)

A.2 Unsupervised Learning

In *unsupervised learning*, as you might guess, the training data is unlabeled. The system tries to learn without a teacher.



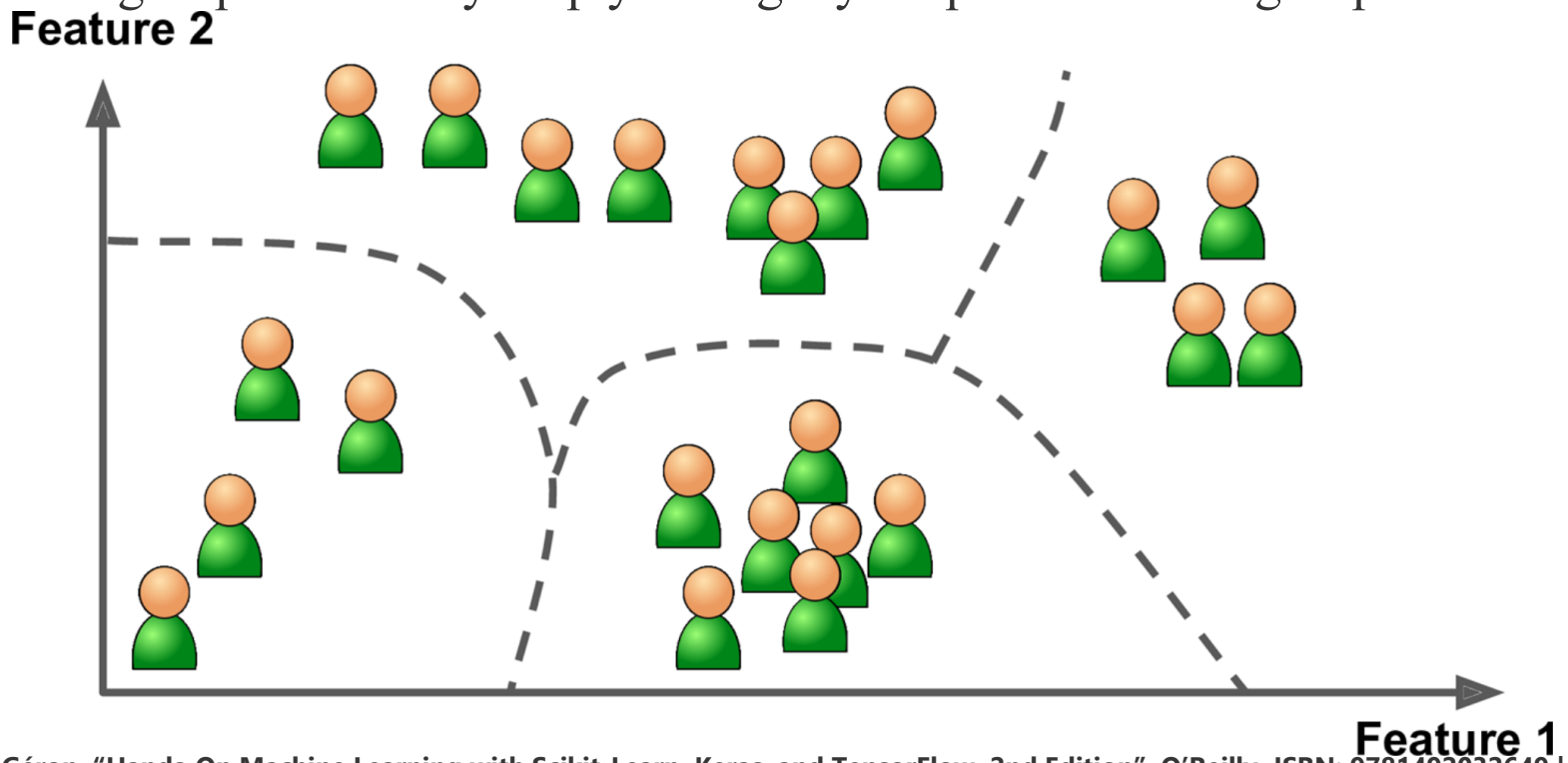
An unlabeled training set for unsupervised learning

Here are some of the most important unsupervised learning algorithms (most of these are covered in Chapters 8 / 9):

- **Clustering**
  - K-Means
  - DBSCAN
  - Hierarchical Cluster Analysis (HCA)
- **Anomaly detection and novelty detection**
  - One-class SVM
  - Isolation Forest
- **Visualization and dimensionality reduction**
  - Principal Component Analysis (PCA)
  - Kernel PCA
  - Locally Linear Embedding (LLE)
  - t-Distributed Stochastic Neighbor Embedding (t-SNE)
- **Association rule learning**
  - Apriori
  - Eclat

A.2 Unsupervised Learning

For example, say you have a lot of data about your blog's visitors. You may want to run a *clustering* algorithm to try to detect groups of similar visitors. At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it might notice that 40% of your visitors are males who love comic books and generally read your blog in the evening, while 20% are young sci-fi lovers who visit during the weekends. If you use a *hierarchical clustering* algorithm, it may also subdivide each group into smaller groups. This may help you target your posts for each group.



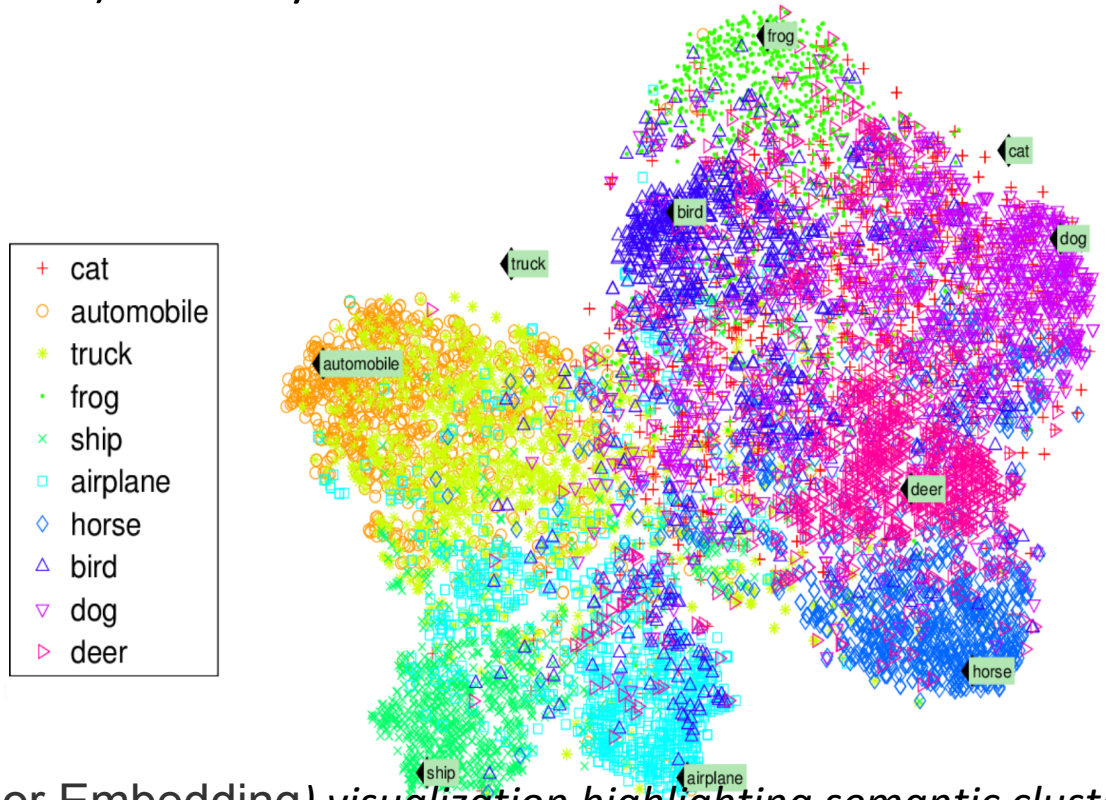
Artificial Intelligence & Machine Learning Terminology

A.2 Unsupervised Learning

Visualization algorithms are also good examples of unsupervised learning algorithms: you feed them a lot of complex and unlabeled data, and they output a 2D or 3D representation of your data that can easily be plotted. These algorithms try to preserve as much structure as they can (e.g., trying to keep separate clusters in the input space from overlapping in the visualization) so that you can understand how the data is organized and perhaps identify unsuspected patterns.

A related task is *dimensionality reduction*, in which the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one. For example, a car's mileage may be strongly correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called *feature extraction*.

In Machine Learning an *attribute* is a data type (e.g., "mileage"), while a *feature* has several meanings, depending on the context, but generally means an attribute plus its value (e.g., "mileage = 15,000"). Many people use the words *attribute* and *feature* interchangeably.

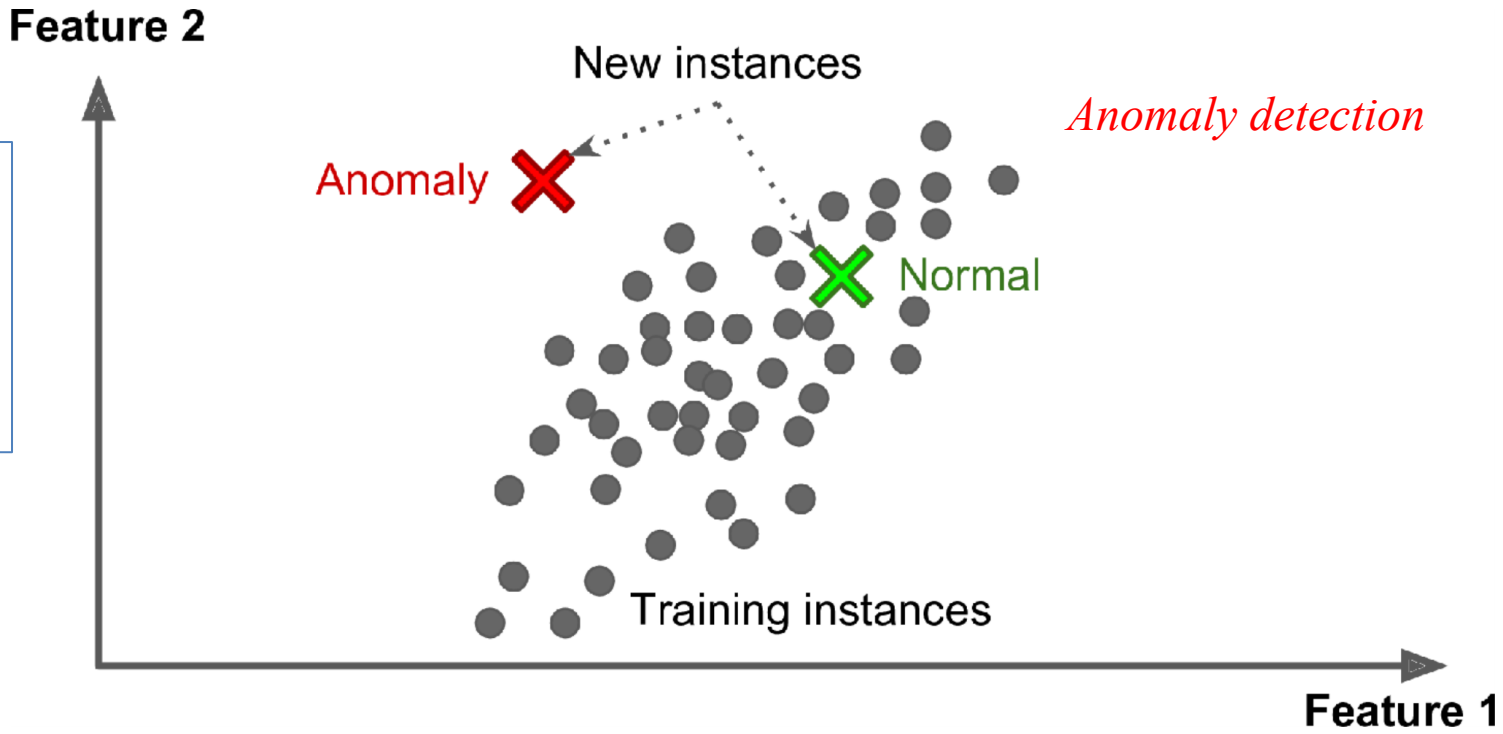


*Example of a t-SNE (*t-Distributed Stochastic Neighbor Embedding*) visualization highlighting semantic cluste*

A.2 Unsupervised Learning

Yet another important unsupervised task is anomaly detection: for example, detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is shown mostly normal instances during training, so it learns to recognize them; then, when it sees a new instance, it can tell whether it looks like a normal one or whether it is likely an anomaly. A very similar task is novelty detection: it aims to detect new instances that look different from all instances in the training set. This requires having a very "clean" training set, devoid of any instance that you would like the algorithm to detect.

Finally, another common unsupervised task is **association rule learning**, in which the goal is to dig into large amounts of data and discover interesting relations between attributes.

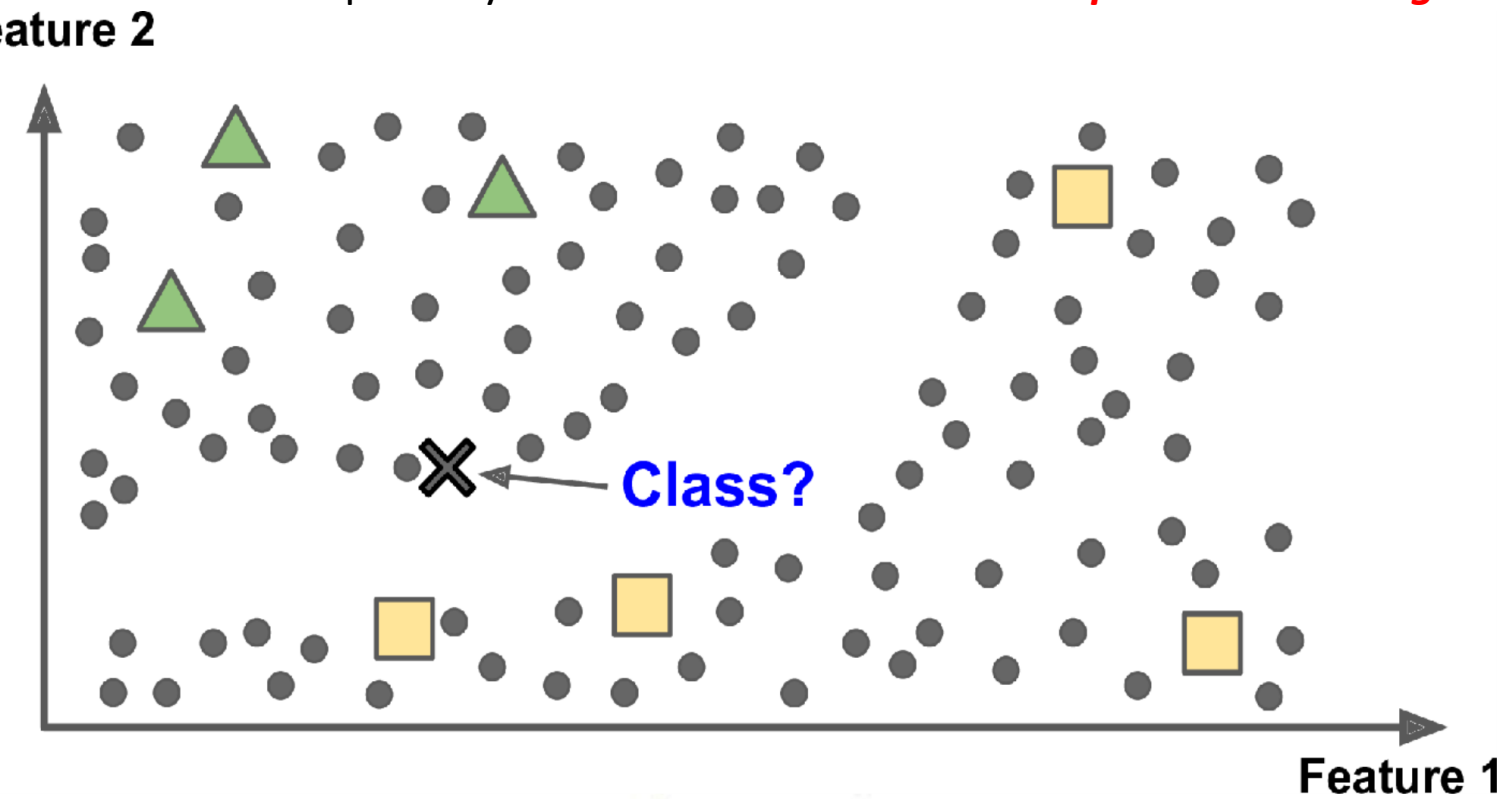


A.3 Semi-supervised Learning

Since labeling data is usually time-consuming and costly, you will often have plenty of unlabeled instances, and few labeled instances. Some algorithms can deal with data that's partially labeled. This is called ***semi-supervised learning***.

Some photo-hosting services, such as Google Photos, are good examples of this. Once you upload all your family photos to the service, it automatically recognizes that the same person A shows up in photos 1, 5, and 11, while another person B shows up in photos 2, 5, and 7. This is the unsupervised part of the algorithm (clustering). Now all the system needs is for you to tell it who these people are. Just add one label per person and it is able to name everyone in every photo, which is useful for searching photos.

Most semi-supervised learning algorithms are combinations of unsupervised and supervised algorithms. For example, ***deep belief networks*** **(DBNs)** are based on unsupervised components called *restricted Boltzmann machines* (RBMs) stacked on top of one another. RBMs are trained sequentially in an unsupervised manner, and then the whole system is fine-tuned using supervised learning techniques.



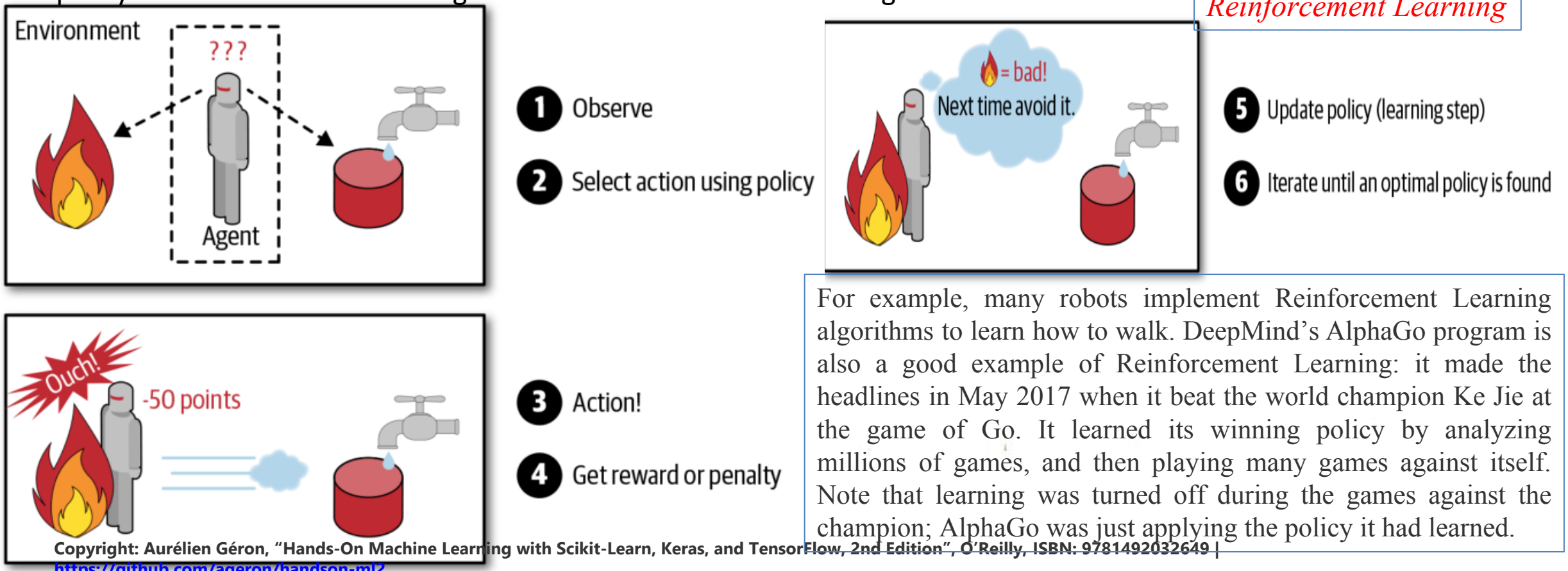
Semi-supervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

Artificial Intelligence & Machine Learning Terminology

A.4 Reinforcement Learning

Reinforcement Learning is a very different beast. The learning system, called an agent in this context, can observe the environment, select and perform actions, and get ***rewards*** in return (or ***penalties*** in the form of negative rewards, as shown in the figure). It must then learn by itself what is the best strategy, called a ***policy***, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

Reinforcement Learning



For example, many robots implement Reinforcement Learning algorithms to learn how to walk. DeepMind's AlphaGo program is also a good example of Reinforcement Learning: it made the headlines in May 2017 when it beat the world champion Ke Jie at the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself. Note that learning was turned off during the games against the champion; AlphaGo was just applying the policy it had learned.

B.1 Batch Learning

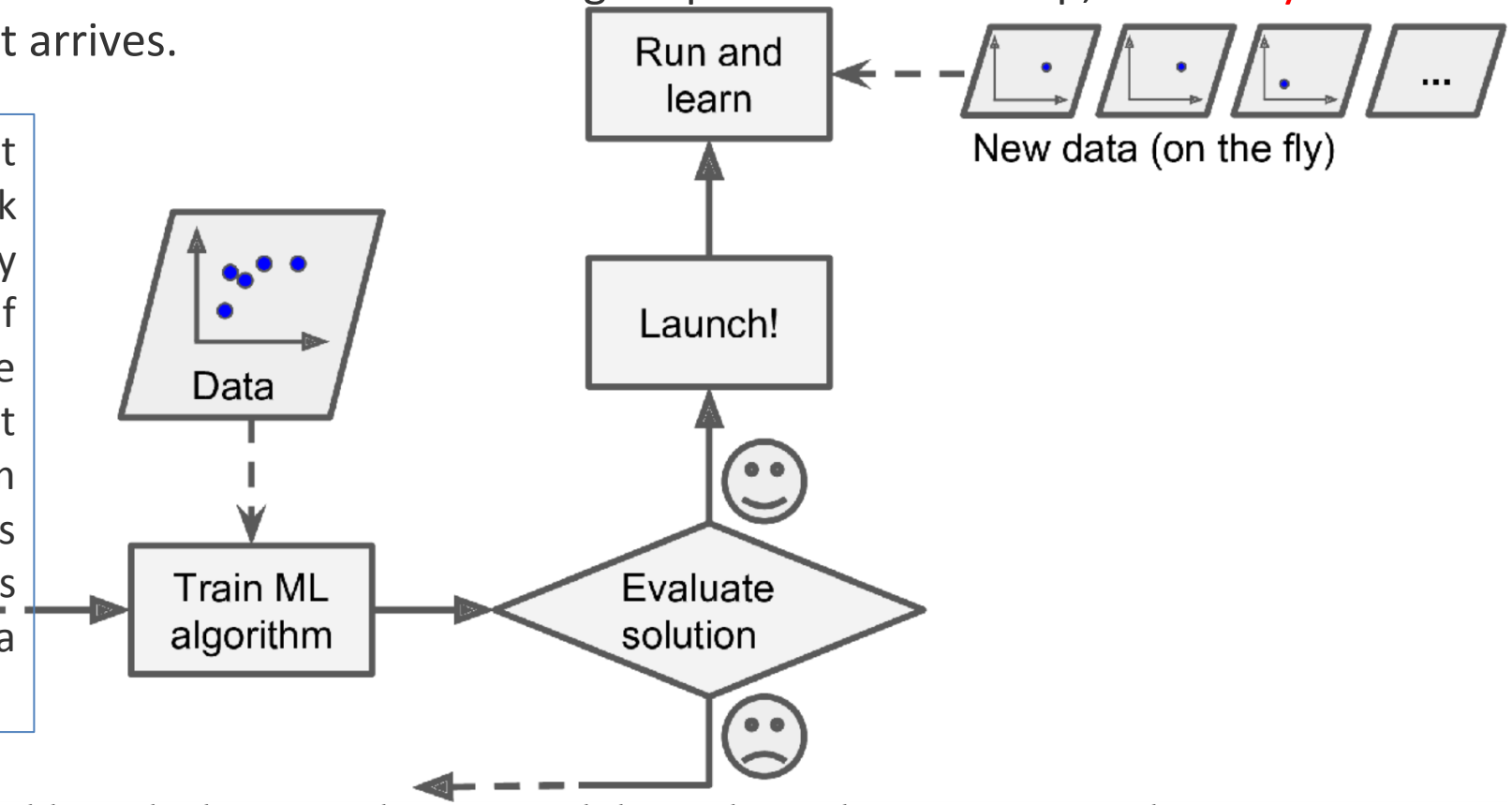
In ***batch learning***, the system is incapable of learning incrementally: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called ***offline learning***.
If you want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then stop the old system and replace it with the new one.
Fortunately, the whole process of training, evaluating, and launching ***a Machine Learning system can be automated fairly easily*** (as shown in previous slide – figure: **ML Automatically adapting to change**), so ***even a batch learning system can adapt to change***. Simply update the data and train a new version of the system from scratch as often as needed.
This solution is simple and often works fine, but training using the full set of data can take many hours, so you would typically train a new system only every 24 hours or even just weekly. If your system needs to adapt to rapidly changing data (e.g., to predict stock prices), then you need a more reactive solution.

Also, training on the full set of data requires a lot of computing resources (CPU, memory space, disk space, disk I/O, network I/O, etc.). If you have a lot of data and you automate your system to train from scratch every day, it will end up costing you a lot of money. If the amount of data is huge, it may even be impossible to use a batch learning algorithm.
Finally, if your system needs to be able to learn autonomously and it has limited resources (e.g., a smartphone application or a rover on Mars), then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper.

B.2 Online Learning

In *online learning*, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called *mini-batches*. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives.

Online learning is great for systems that receive data as a continuous flow (e.g., stock prices) and need to adapt to change rapidly or autonomously. It is also a good option if you have limited computing resources: once an online learning system has learned about new data instances, it does not need them anymore, so you can discard them (unless you want to be able to roll back to a previous state and "replay" the data). This can save a huge amount of space.



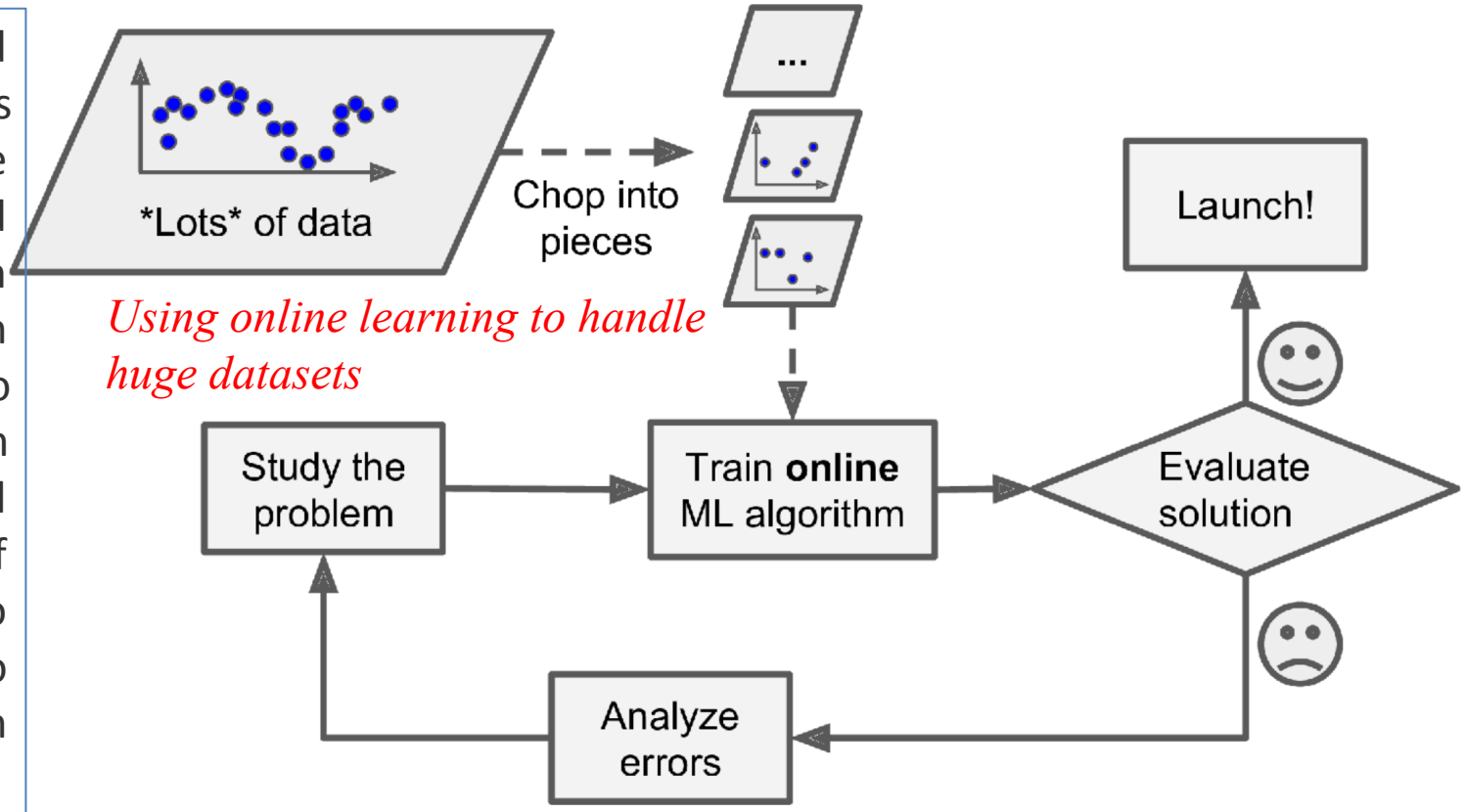
In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in.

Artificial Intelligence & Machine Learning Terminology

B.2 Online Learning

Online learning algorithms can also be used to train systems on huge datasets that cannot fit in one machine's main memory (this is called ***out-of-core* learning**). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data. Warning: *Out-of-core learning is usually done offline (i.e., not on the live system), so online learning can be a confusing name. Think of it as incremental learning.*

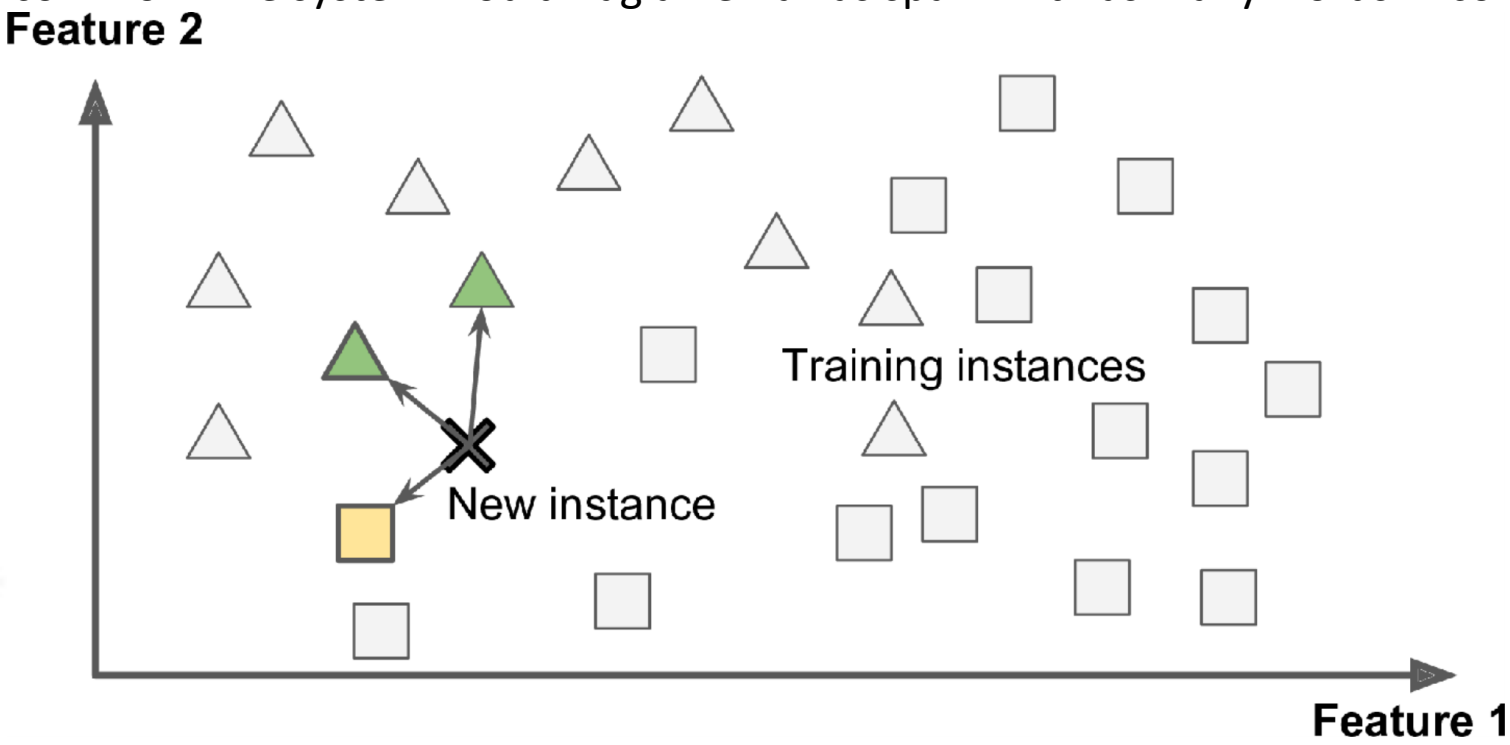
A big challenge with online learning is that if bad data is fed to the system, the system's performance will gradually decline. If it's a live system, your clients will notice. For example, bad data could come from a malfunctioning sensor on a robot, or from someone spamming a search engine to try to rank high in search results. To reduce this risk, you need to monitor your system closely and promptly switch learning off (and possibly revert to a previously working state) if you detect a drop in performance. You may also want to monitor the input data and react to abnormal data (e.g., using an anomaly detection algorithm).



Using online learning to handle huge datasets

C.1 Instance-Based Learning

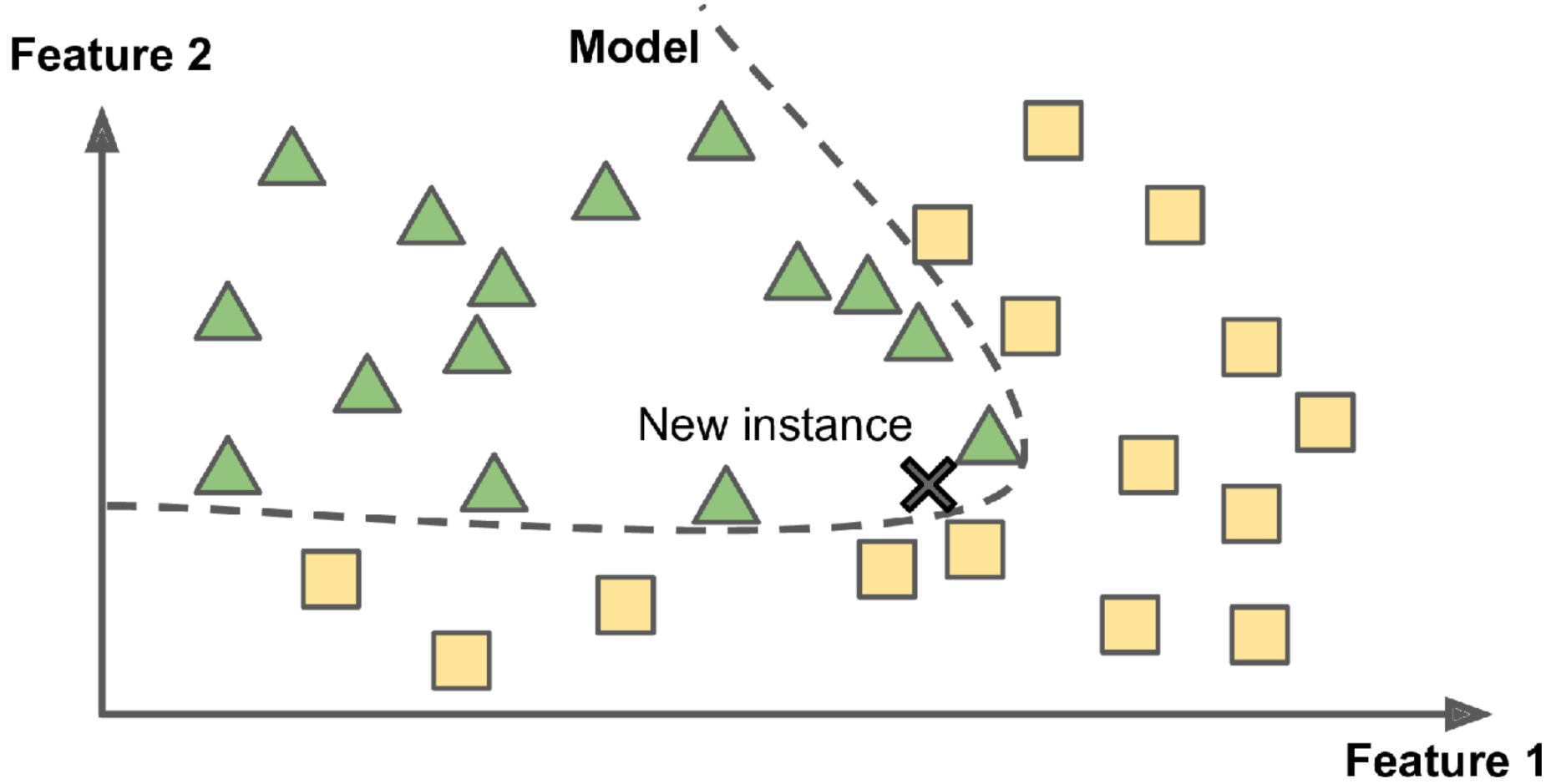
Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are identical to emails that have already been flagged by users—not the worst solution, but certainly not the best. Instead of just flagging emails that are identical to known spam emails, your spam filter could be programmed to also flag emails that are very similar to known spam emails. This requires a ***measure of similarity*** between two emails. A (very basic) similarity measure between two emails could be to count the number of words they have in common. The system would flag an email as spam if it has many words in common with a known spam email.



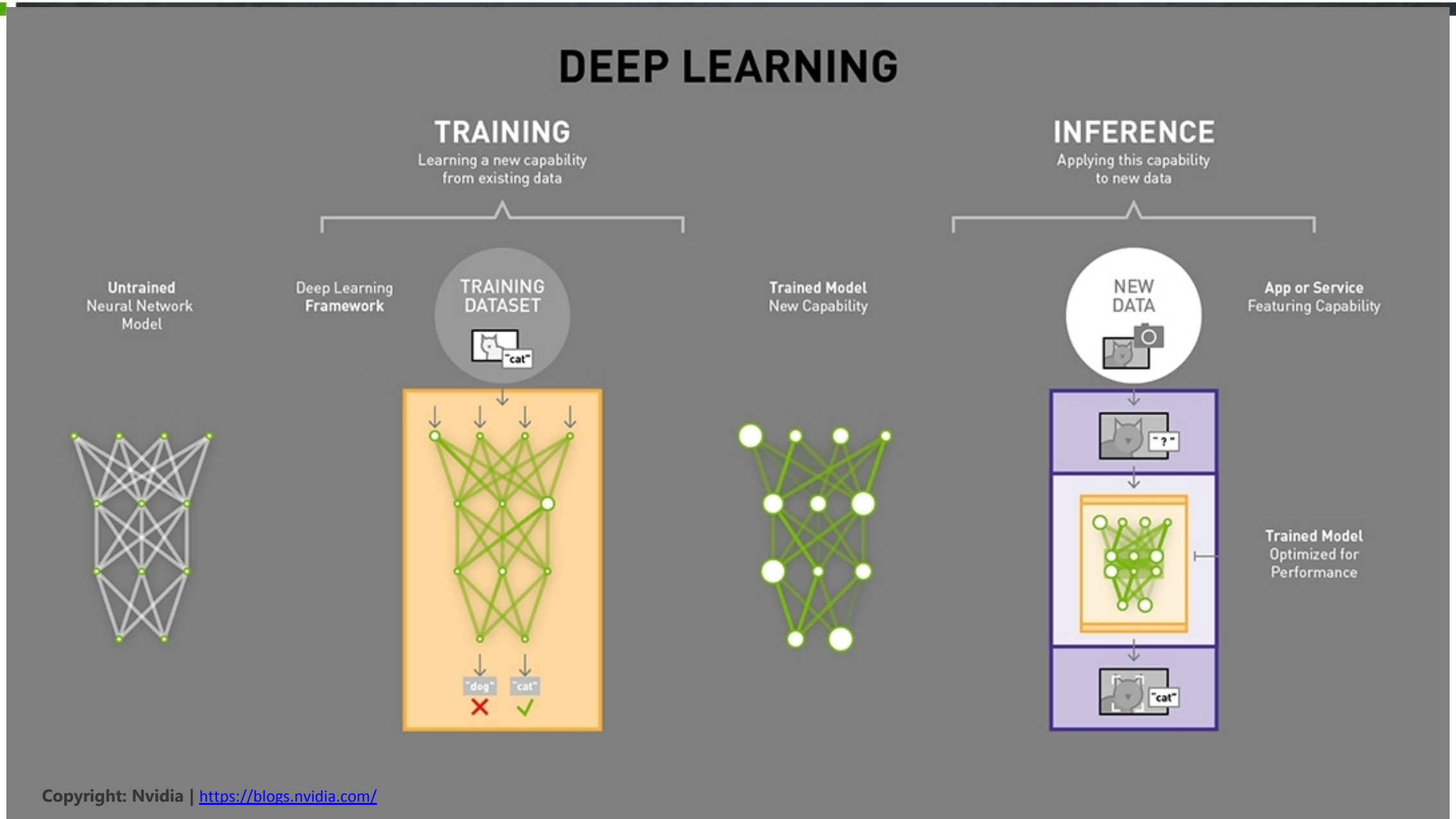
This is called ***instance-based learning***: the system learns the examples "by heart", then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them). For example, in figure the new instance would be classified as a triangle because the majority of the most similar instances belong to that class.

C.2 Model-Based Learning

Another way to generalize from a set of examples is to build a model of these examples and then use that model to make ***predictions***. This is called ***model-based learning***.

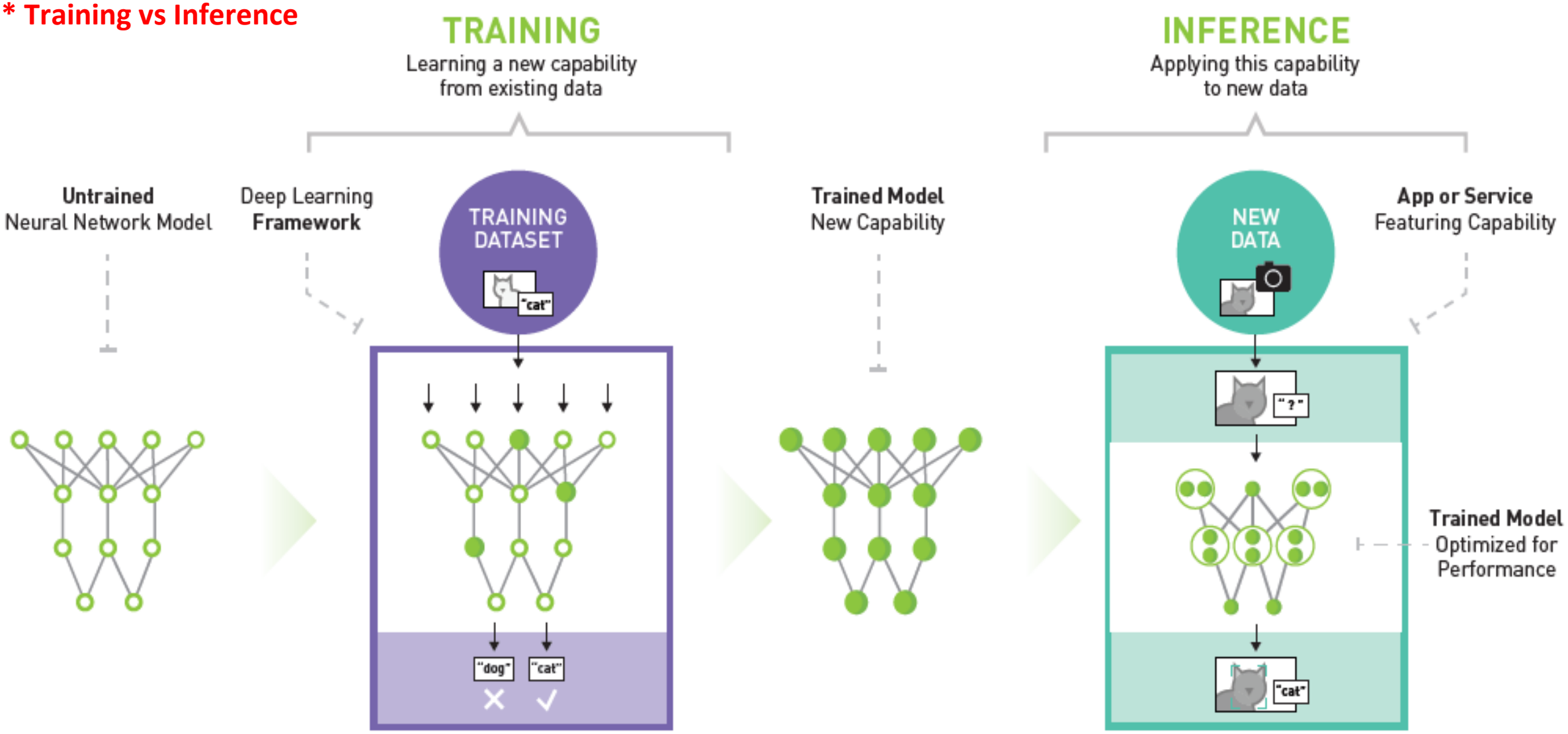


Artificial Intelligence & Machine Learning Terminology



Artificial Intelligence & Machine Learning Terminology

*** Training vs Inference**



DEMO 1 in Python/Jupiter for Model-Based Learning

For example, suppose you want to know if money makes people happy, so you download the Better Life Index data from the OECD's website (Organisation for Economic Co-operation and Development: https://homl.info/4) and stats about gross domestic product (GDP) per capita from the IMF's website (International Monetary Fund: https://homl.info/5). Then you join the tables and sort by GDP per capita. Table shows an excerpt of what you get.

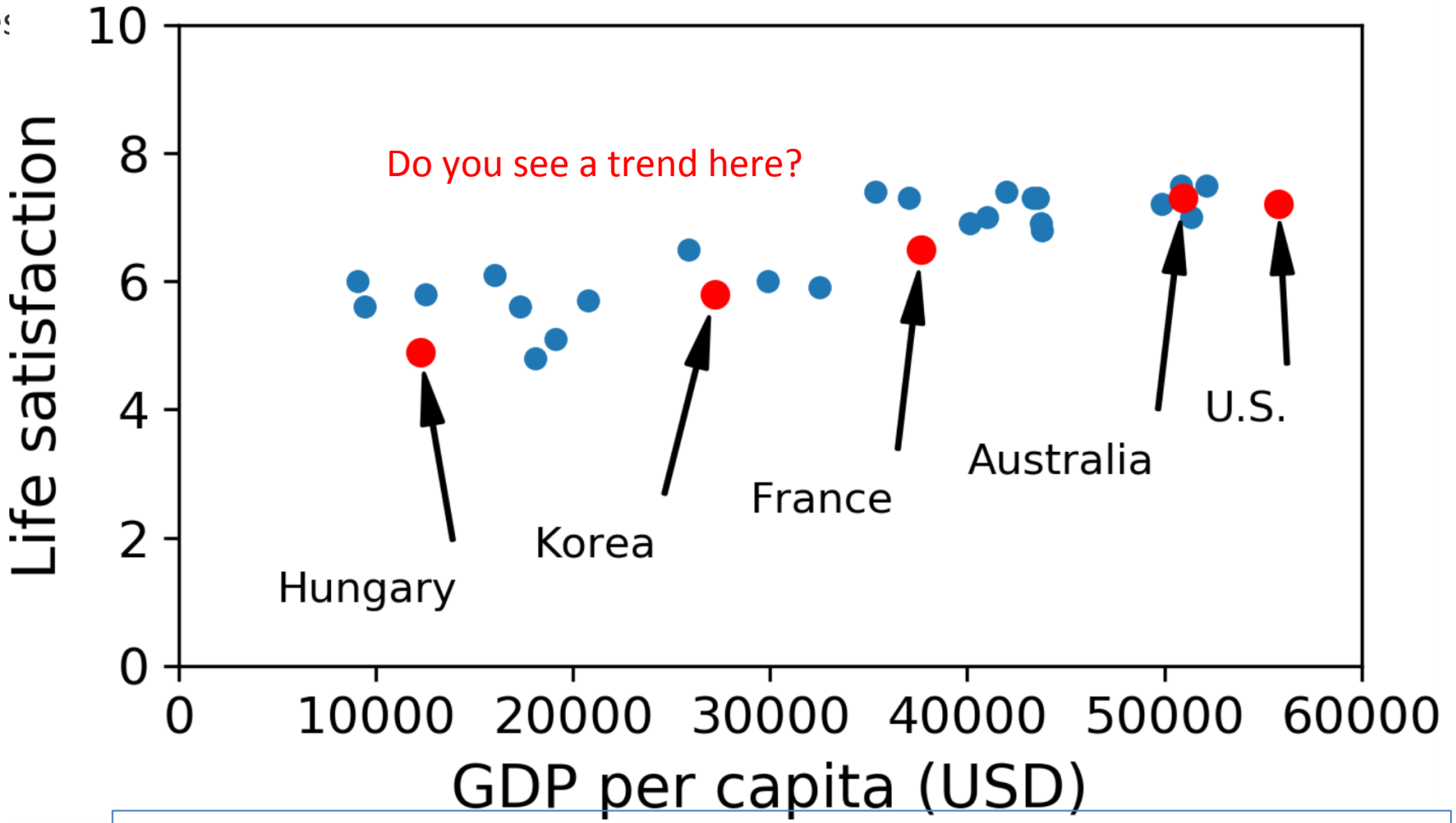
Table (Year 2015):
Does money make people happier?

Country	GDP per capita (USD)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2

DEMO 1 in Python/Jupiter for Model-Based Learning

Let's plot the data for these countries

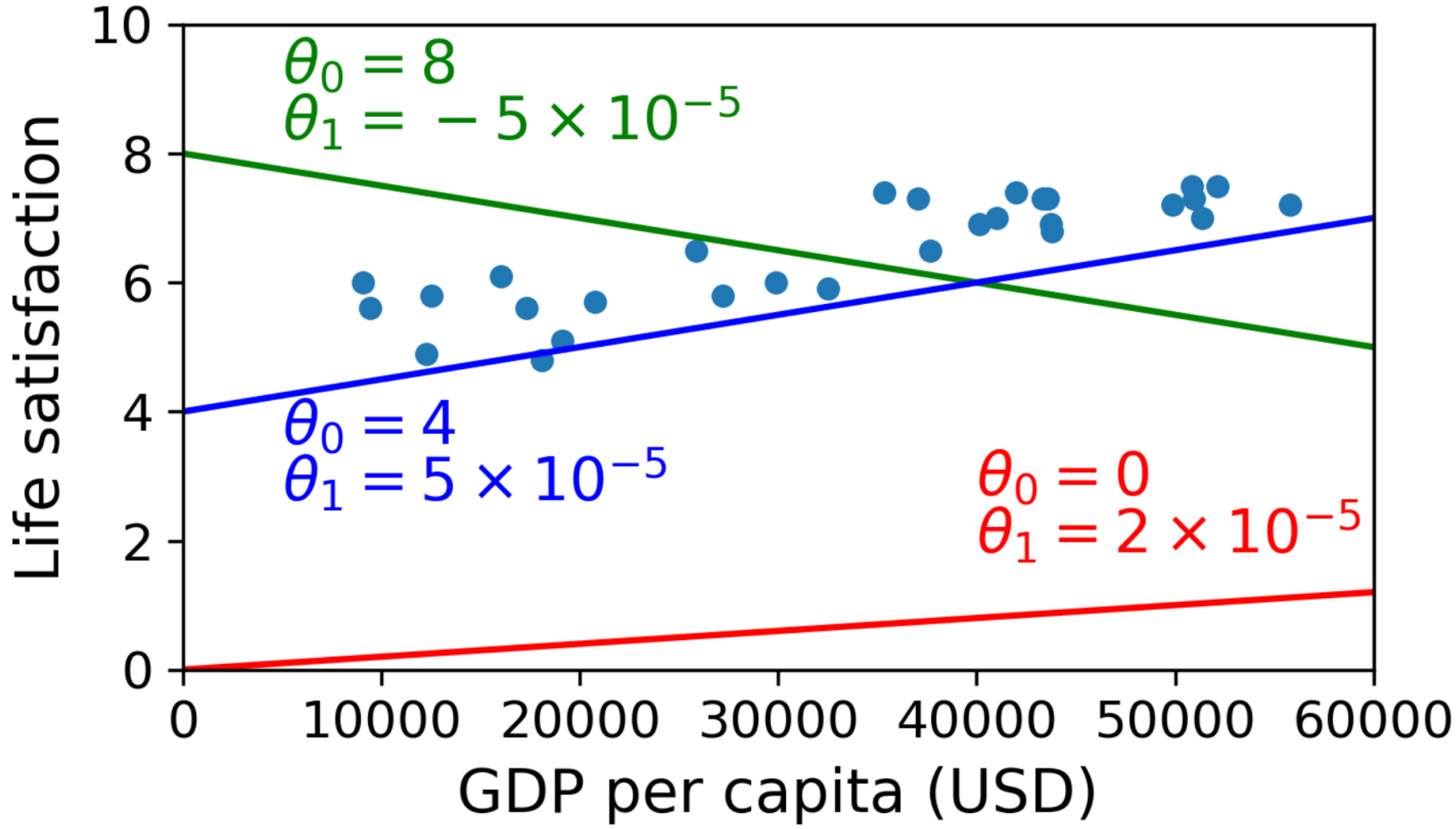
There does seem to be a trend here! Although the data is noisy (i.e., partly random), it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So you decide to model life satisfaction as a linear function of GDP per capita. This step is called model selection: you selected a linear model of life satisfaction with just one attribute, GDP per capita (Equation 1-1: A simple linear model).



Equation 1-1: $life\_satisfaction = \theta_0 + \theta 1 \times GDP\_per\_capita$

DEMO 1 in Python/Jupiter for Model-Based Learning

This model has two model parameters, θ_0 and θ_1. By tweaking these parameters, you can make your model represent any linear function, as shown in figure: ***a few possible linear models***.



DEMO 1 in Python/Jupiter for Model-Based Learning

Before you can use your model, you need to define the parameter values θ_0 and θ_1. How can you know which values will make your model perform best? To answer this question, you need to specify a performance measure. You can either define a ***utility function*** (or ***fitness function***) that measures how *good* your model is, or you can define a ***cost function*** that measures **how *bad* it is**. For **Linear Regression problems**, people typically use ***a cost function that measures the distance between the linear model's predictions and the training examples; the objective is to minimize this distance.***

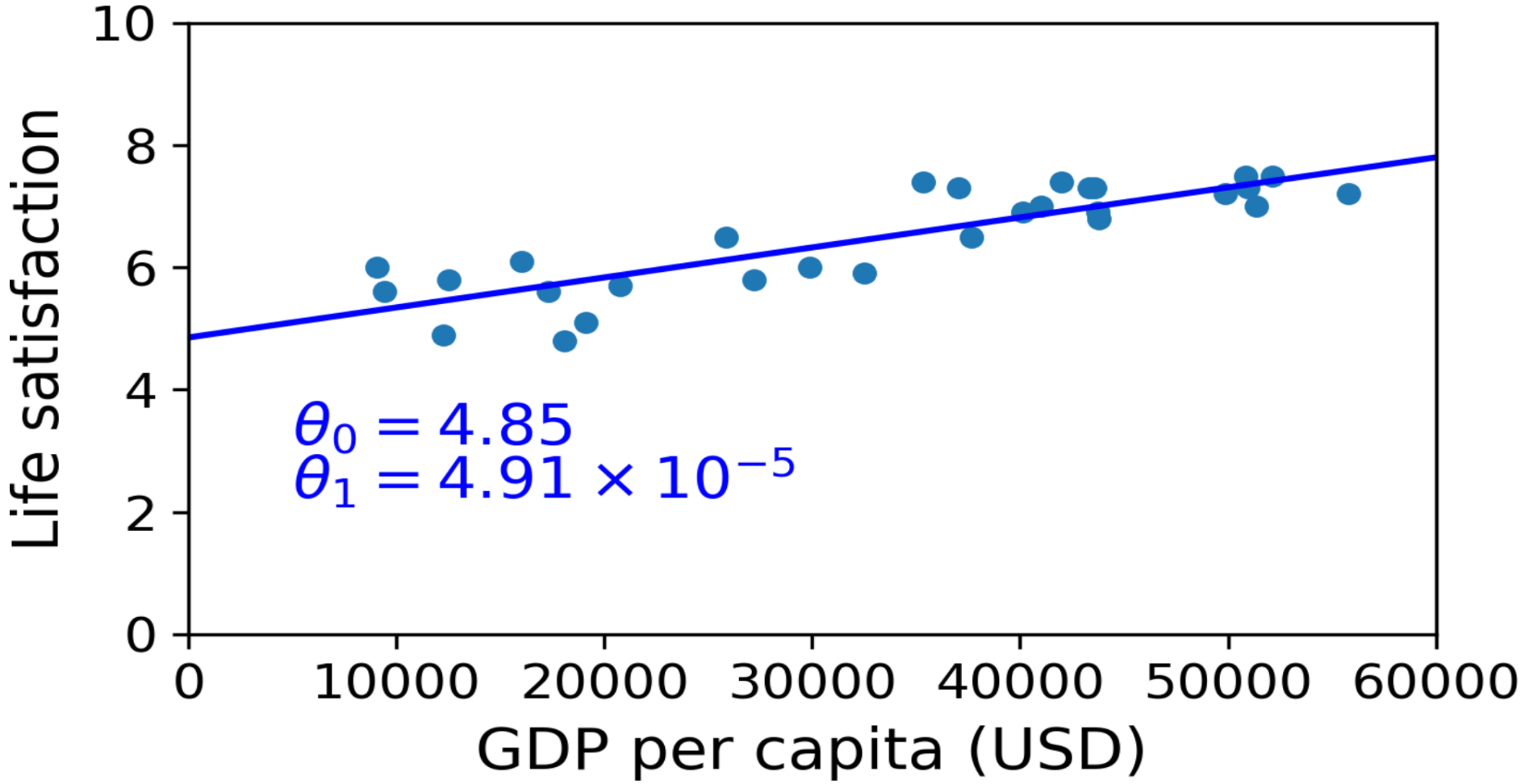
This is where t***he Linear Regression algorithm*** comes in: you feed it your ***training examples***, and it finds the parameters that make the linear model fit best to your data. This is called ***training* the model**.

In our case, the algorithm finds that the optimal parameter values are: $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$

WARNING: Confusingly, the same word "**model**" can refer to a ***type of model*** (e.g., Linear Regression), to a ***fully specified model architecture*** (e.g., ***Linear Regression with one input and one output***), or to the ***final trained model*** ready to be used for predictions (e.g., **Linear Regression with one input and one output, using $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$**). ***Model selection consists in choosing the type of model and fully specifying its architecture***. **Training a model** means **running an algorithm to find the model parameters** that will make it b***est fit the training data (and hopefully make good predictions on new data).***

DEMO 1 in Python/Jupiter for Model-Based Learning

Now the model fits the training data as closely as possible (for a linear model), as you can see ***[The linear model that fits the training data best]***



Artificial Intelligence & Machine Learning Overview

DEMO 1 in Python/Jupiter for Model-Based Learning

https://colab.research.google.com/github/ageron/handson-ml2/blob/master/01_the_machine_learning_landscape.ipynb#scrollTo=xYoqxdHtY9sA

You are finally ready to run the model to make predictions. For example, say you want to know how happy Cypriots are, and the OECD data does not have the answer. Fortunately, you can use your model to make a good prediction: you look up Cyprus's GDP per capita, find $22,587, and then apply your model and find that life satisfaction is likely to be somewhere around:

$4.85 + 22{,}587 \times 4.91 \times 10^{-5} = 5.96$

Training and running a linear model using Python 3.5+ and Scikit-Learn:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model
# Load the data
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
                             encoding='latin1', na_values="n/a")


# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# Select a linear model
model = sklearn.linear_model.LinearRegression()

# Train the model
model.fit(X, y)

# Make a prediction for Cyprus
X_new = [[22587]]  # Cyprus's GDP per capita
print(model.predict(X_new)) # outputs [[ 5.96242338]]
```

DEMO 1 in Python/Jupiter for Model-Based Learning

NOTE

If ***you had used an instance-based learning algorithm instead***, you would have found that Slovenia has the closest GDP per capita to that of Cyprus ($20,732), and since the OECD data tells us that Slovenians' life satisfaction is 5.7, you would have predicted a life satisfaction of 5.7 for Cyprus.

If you zoom out a bit and look at the two next-closest countries, you will find Portugal and Spain with life satisfactions of 5.1 and 6.5, respectively. Averaging these three values, you get 5.77, which is pretty close to your model-based prediction.
This simple algorithm is called ***k-Nearest Neighbors*** regression (in this example, ***k*** = 3).

Replacing the **Linear Regression model** with **k-Nearest Neighbors regression** in the previous code is as simple as **replacing these two lines:**

```
import sklearn.linear_model
model = sklearn.linear_model.LinearRegression()
```

With these two lines:

```
import sklearn.neighbors
model = sklearn.neighbors.KNeighborsRegressor(
    n_neighbors=3)
```

DEMO 1 in Python/Jupiter for Model-Based Learning

If all went well, your model will make good predictions. If not, you may need to use more attributes (employment rate, health, air pollution, etc.), get more or better-quality training data, or perhaps select a more powerful model (e.g., a Polynomial Regression model)

In summary:

- ***Studied the data***.
- ***Selected a model***.
- ***Trained it on the training data*** (i.e., the learning algorithm searched for the model parameter values that minimize a cost function).
- Finally, you ***applied the model to make predictions on new cases*** (this is called ***inference***), hoping that this model will generalize well.

This is what a typical Machine Learning project looks like. In the next sections (Chapter 2 from the book) you will experience this firsthand by going through ***a project end to end***. We have covered a lot of ground so far: you now know what Machine Learning is really about, why it is useful, what some of the most common categories of ML systems are, and what a typical project workflow looks like.

DEMO 1 in Python/Jupiter for Model-Based Learning

Main Challenges of Machine Learning

In short, since your main task is to select a learning algorithm and train it on some data, the ***two things that can go wrong*** are D. "bad data" and/or E. "bad algorithm.":

- **D.1 Insufficient Quantity of Training Data:** For a toddler to learn what an apple is, all it takes is for you to point to an apple and say "apple" (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius.
- **D.2 Non-representative Training Data:** In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.
- **D.3 Poor-Quality Data:** Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well.
- **D.4 Irrelevant Features:** As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones.

DEMO 1 in Python/Jupiter for Model-Based Learning

Main Challenges of Machine Learning

In short, since your main task is to select a learning algorithm and train it on some data, the ***two things that can go wrong*** are D. "bad data" and/or E. "bad algorithm.":

- **E.1 Over-fitting the Training Data:** Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that *all* taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In Machine Learning this is called *over-fitting*: it means that the model performs well on the training data, but it does not generalize well.

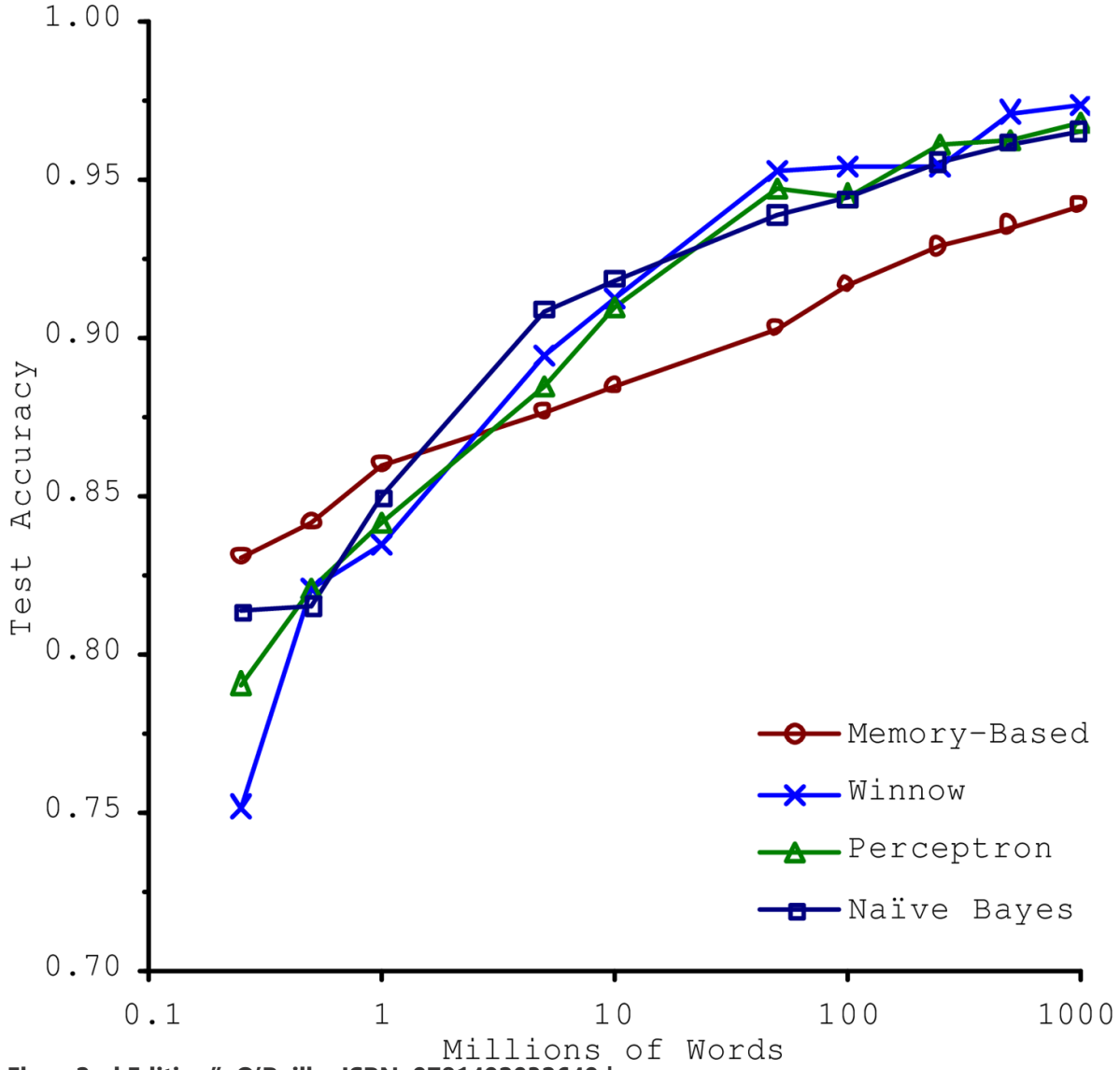
- **E.2 Under-fitting the Training Data:** As you might guess, *under-fitting* is the opposite of over-fitting: it occurs when your model is too simple to learn the underlying structure of the data.

Artificial Intelligence & Machine Learning Overview

D.1 Insufficient Quantity of Training Data: DEMO 1 in Python/Jupiter for Model-Based Learning

For a toddler to learn what an apple is, all it takes is for you to point to an apple and say "apple" (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius. ***Machine Learning is not quite there yet***; it takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

In a famous paper published in 2001, Microsoft researchers Michele Banko and Eric Brill showed that very different Machine Learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation, once they were given enough data.

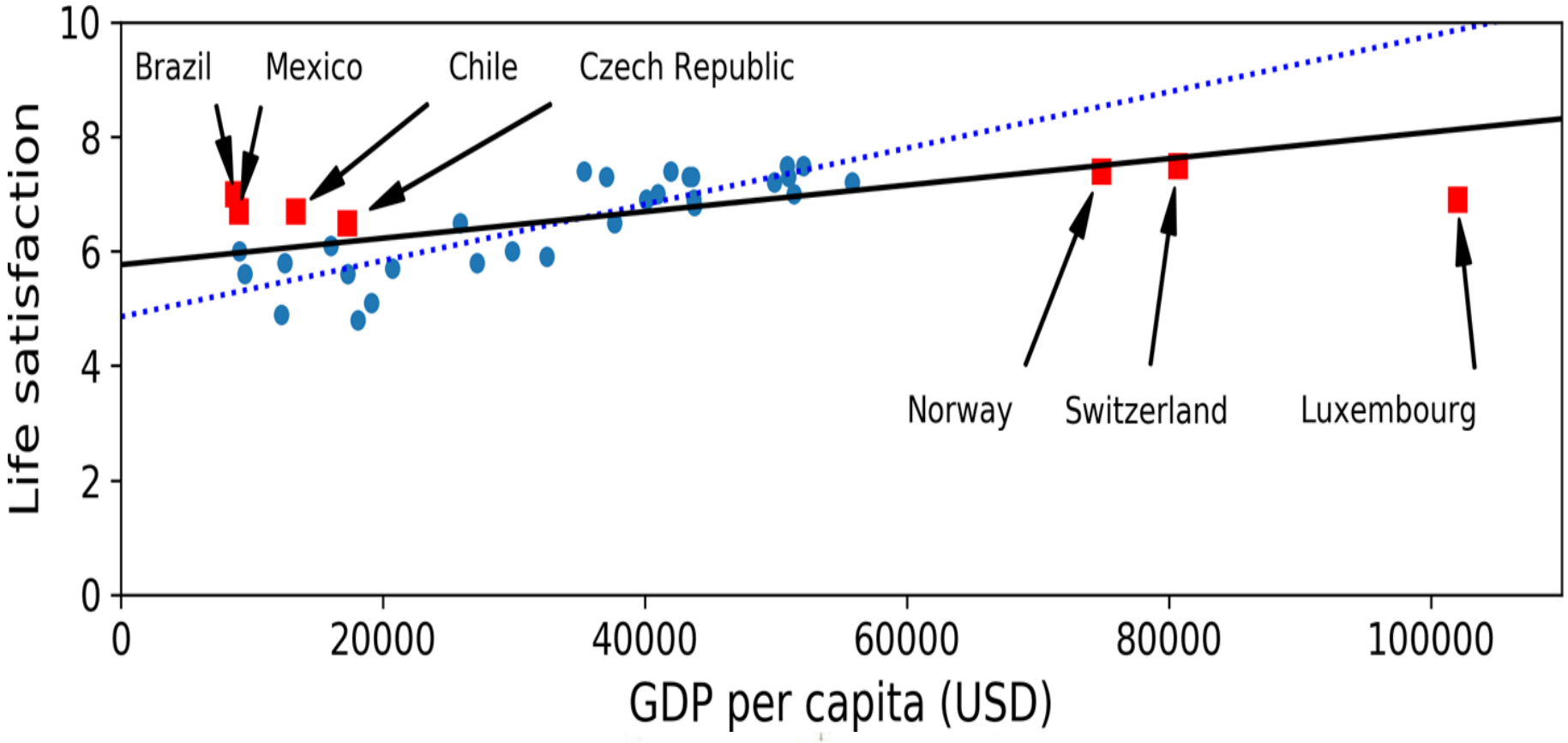


DEMO 1 in Python/Jupiter for Model-Based Learning

D.2 Non-representative Training Data:

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.

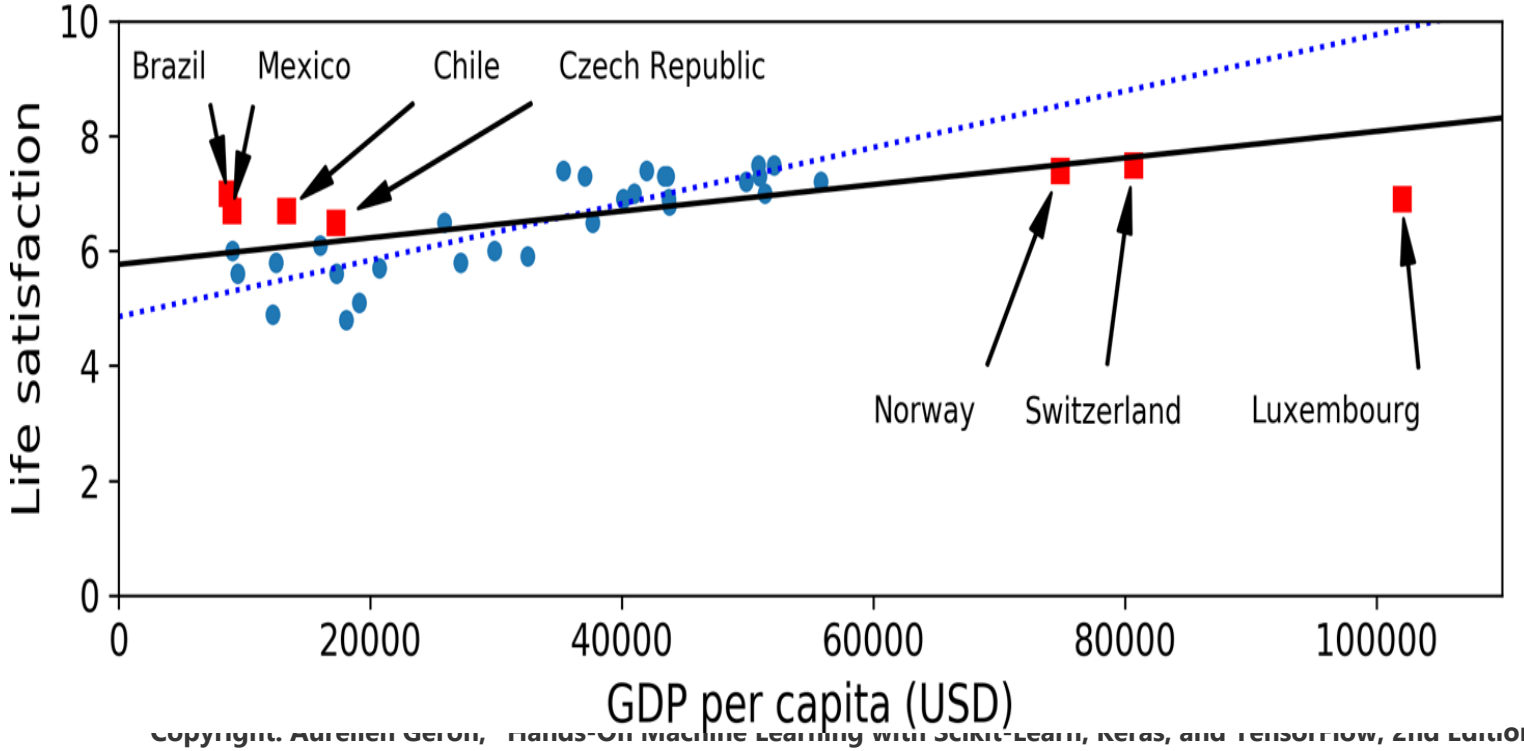
For example, the set of countries we used earlier for training the linear model was not perfectly representative; a few countries were missing. Figure shows what the data looks like when you add the missing countries.



DEMO 1 in Python/Jupiter for Model-Based Learning

D.2 Non-representative Training Data:

If you train a linear model on this data, you get the solid line, while the old model is represented by the dotted line. As you can see, not only does adding a few missing countries significantly alter the model, but it makes it clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact, they seem unhappier), and conversely some poor countries seem happier than many rich countries.
By using a ***non-representative training set***, we ***trained a model that is unlikely to make accurate predictions***, especially for very poor and very rich countries.



It is crucial to ***use a training set*** that ***is representative*** of the cases you want to generalize to. This is often harder than it sounds: ***if the sample is too small***, you will have **sampling noise** (i.e., ***non-representative data*** as a result of chance), but even very ***large samples can be non-representative if the sampling method is flawed***. This is called **sampling bias**.

DEMO 1 in Python/Jupiter for Model-Based Learning

D.3. Poor-Quality Data

Obviously, ***if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements),*** it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple of examples of when you'd want to clean up training data:

- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (e.g., 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it.

D.4. Irrelevant Features

As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a Machine Learning project is coming up with a good set of features to train on. This process, called ***feature engineering***, involves the following steps:

- ***Feature selection*** (selecting the most useful features to train on among existing features)
- ***Feature extraction*** (combining existing features to produce a more useful one—as we saw earlier, dimensionality reduction algorithms can help)
- ***Creating new features by gathering new data***

DEMO 1 in Python/Jupiter for Model-Based Learning

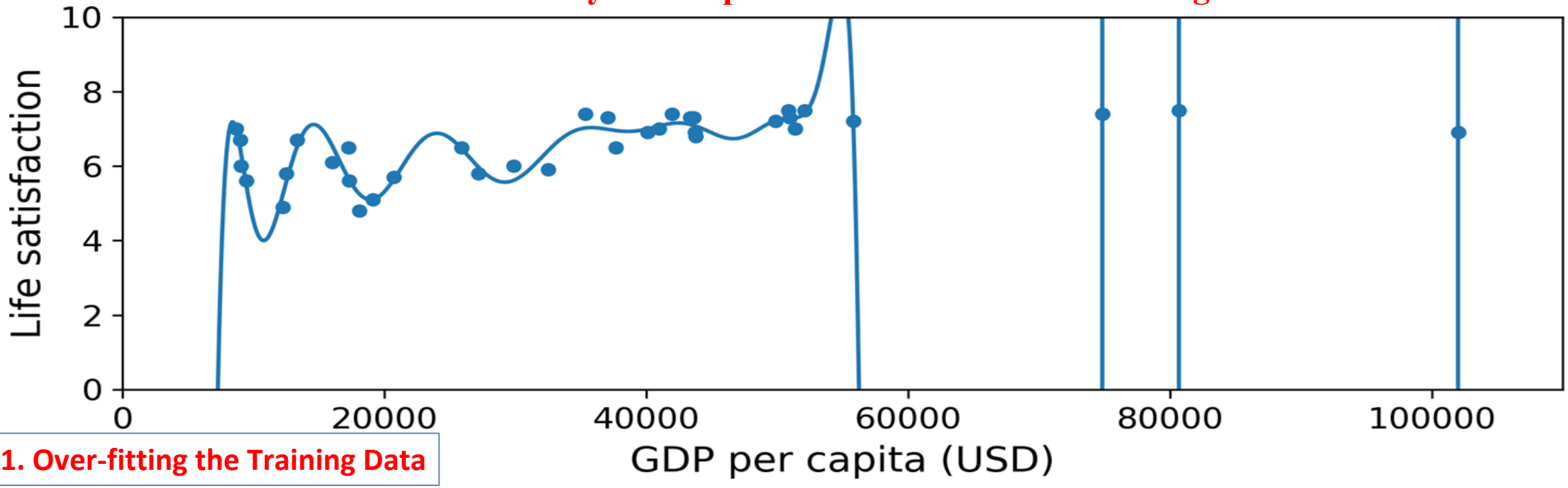
E.1. Over-fitting the Training Data

Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that all taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful.

In Machine Learning this is called over-fitting: it means that the model performs well on the training data, but it does not generalize well.

The figure shows an example of a high-degree polynomial life satisfaction model that strongly over-fits the training data. Even though it performs much better on the training data than the simple linear model, would you really trust its predictions?

DEMO 1 in Python/Jupiter for Model-Based Learning



E.1. Over-fitting the Training Data

Complex models such as deep neural networks can detect subtle patterns in the data, but if the training set is noisy, or if it is too small (which introduces sampling noise), then the model is likely to detect patterns in the noise itself. Obviously these patterns will not generalize to new instances. For example, say you feed your life satisfaction model many more attributes, including uninformative ones such as the country's name. In that case, a complex model may detect patterns like the fact that all countries in the training data with a *w* in their name have a life satisfaction greater than 7: New Zealand (7.3), Norway (7.4), Sweden (7.2), and Switzerland (7.5). How confident are you that the *w*-satisfaction rule generalizes to Rwanda or Zimbabwe? Obviously this pattern occurred in the training data by pure chance, but the model has no way to tell whether a pattern is real or simply the result of noise in the data.

DEMO 1 in Python/Jupiter for Model-Based Learning

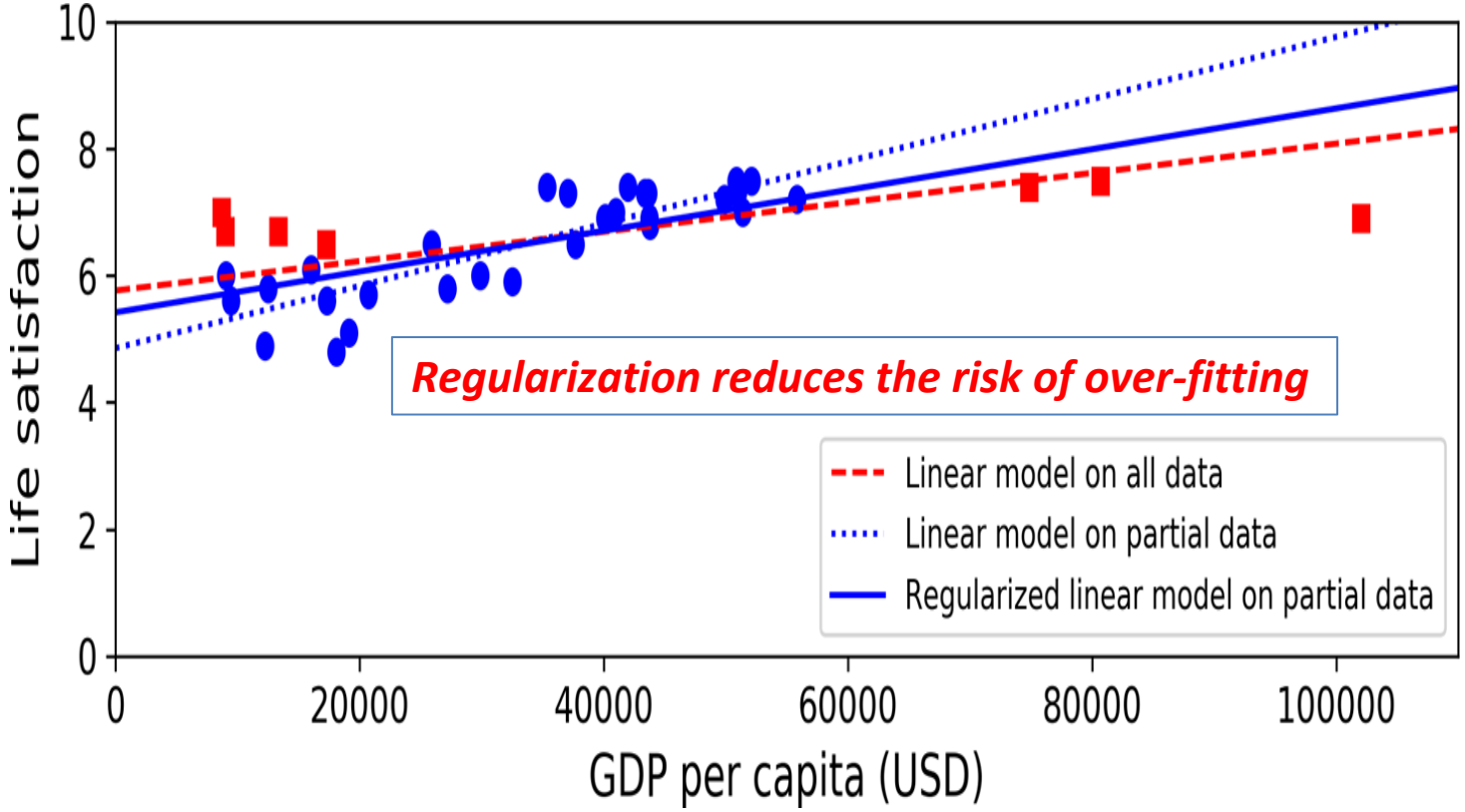
E.1. Over-fitting the Training Data

Constraining a model to make it simpler and reduce the risk of over-fitting is called ***regularization***. For example, the linear model we defined earlier has two parameters, θ_0 and θ_1. This gives the learning algorithm two ***degrees of freedom*** to adapt the model to the training data: ***it can tweak both the height (θ_0) and the slope (θ_1) of the line. If we forced $\theta_1 = 0$***, the algorithm would have only ***one degree of freedom*** and would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the training instances, so it would end up around the mean. A very simple model indeed! If we allow the algorithm to modify θ_1 but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a model that's simpler than one with two degrees of freedom, but more complex than one with just one. You want to find the right balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalize well.

Artificial Intelligence & Machine Learning Overview

DEMO 1 in Python/Jupiter for Model-Based Learning

The following figure shows three models. The dotted line represents the original model that was trained on the countries represented as circles (without the countries represented as squares), the dashed line is our second model trained with all countries (circles and squares), and the solid line is a model trained with the same data as the first model but with a regularization constraint. You can see that regularization forced the model to have a smaller slope: this model does not fit the training data (circles) as well as the first model, but it actually generalizes better to new examples that it did not see during training (squares).



The amount of regularization to apply during learning can be controlled by a ***hyper-parameter***. A hyper-parameter is a parameter of a learning algorithm (not of the model). As such, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training. If you set the regularization **hyper-parameter** to a very large value, you will get an almost flat model (a slope close to zero); the learning algorithm will almost certainly not **over-fit** the training data, but it will be less likely to find a good solution. Tuning **hyper-parameters** is an important part of building a Machine Learning system.

DEMO 1 in Python/Jupiter for Model-Based Learning

E.2. Under-fitting the Training Data

As you might guess, ***under-fitting*** is the opposite of ***over-fitting***: it occurs when your model is too simple to learn the underlying structure of the data.

For example, a linear model of life satisfaction is ***prone to under-fit***; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples.

Here are the main options for fixing this problem:

- Select a more powerful model, with more parameters.
- Feed better features to the learning algorithm (feature engineering).
- Reduce the constraints on the model (e.g., reduce the regularization hyper-parameter).

Stepping Back **DEMO 1 in Python/Jupiter for Model-Based Learning**

By now, there are a lot info about the Machine Learning. However, we went through so many concepts that you may be feeling a little lost, so let's step back and look at the big picture:

- ***Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules***.
- ***There are many different types of ML systems***: ***supervised or not, batch or online, instance-based or model-based.***
- In an ***ML project you gather data in a training set***, and you ***feed the training set to a learning algorithm***. If the algorithm is model-based, it tunes some parameters to fit the model to the training set (i.e., to make good predictions on the training set itself), and then hopefully it will be able to make good predictions on new cases as well. If the algorithm is instance-based, it just learns the examples by heart and generalizes to new instances by using a similarity measure to compare them to the learned instances.
- The system will not perform well if your training set is too small, or ***if the data is not representative***, is ***noisy***, or is ***polluted with irrelevant features*** (garbage in, garbage out). Lastly, your model needs to be neither too simple (in which case it will ***under-fit***) nor too complex (in which case it will ***over-fit***).

There's just one last important topic to cover: once you have trained a model, you don't want to just "hope" it generalizes to new cases.

You want to evaluate it and fine-tune it if necessary. Let's see how to do that.

Testing and Validating

DEMO 1 in Python/Jupiter for Model-Based Learning

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. One way to do that is to put your model in production and monitor how well it performs. This works well, but if your model is horribly bad, your users will complain—not the best idea.

A better option is to split your data into two sets: the *training set* and the *test set*. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the *generalization error* (or *out-of-sample error*), and by evaluating your model on the test set, you get an estimate of this error. This value tells you how well your model will perform on instances it has never seen before.

***If the training error is low* (i.e., your model makes few mistakes on the training set) *but the generalization error is high*, it means that *your model is over-fitting the training data*.**

* It is common to use 80% of the data for training and hold out 20% for testing. However, this depends on the size of the dataset: if it contains 10 million instances, then holding out 1% means your test set will contain 100,000 instances, probably more than enough to get a good estimate of the generalization error.
* For simple tasks you may evaluate linear models with various levels of regularization, and for a complex problem you may evaluate various neural networks.

DEMO 2 in Python/Jupiter - End2End Project

End-to-End Machine Learning Project

In this chapter you will work through an example project end to end, pretending to be a recently hired ML Solutions Developer / Data Scientist at a real estate company.

Here are the main steps you will go through:

1. Look at the big picture
2. Get the data
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune the model.
7. Present the solution => Launch, monitor, and maintain your system.

DEMO 2 in Python/Jupiter - End2End Project

Working with Real Data

When you are learning about Machine Learning, it is best to experiment with real-world data, not artificial datasets. Fortunately, there are thousands of open datasets to choose from, ranging across all sorts of domains. Here are a few places you can look to get data:

- Popular open data repositories
  - UC Irvine Machine Learning Repository
  - Kaggle datasets
  - Amazon's AWS datasets
- Meta portals (they list open data repositories)
  - Data Portals
  - OpenDataMonitor
  - Quandl
- Other pages listing many popular open data repositories
  - Wikipedia's list of Machine Learning datasets
  - Quora.com
  - The datasets subreddit

Perceptron, MLP, ANN FFN with Back-propagation, CNNs, etc.

AI NEURAL NETWORKS: Deep Learning

2. AI - NN

Artificial Neural Networks with Keras

Birds inspired us to fly, Human Eye inspired Cameras with Wider Field of View, and nature has inspired countless more inventions.

It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the logic that sparked ***artificial neural networks (ANNs): an ANN is a Machine Learning model inspired by the networks of biological neurons found in our brains***.
However, although planes were inspired by birds, they don't have to flap their wings. Similarly, ANNs have gradually become quite different from their biological cousins.

ANNs are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks such as classifying

- *billions of images (e.g., Google Images),*
- *powering speech recognition services (e.g., Apple's Siri),*
- *recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or*
- *learning to beat the world champion at the game of Go (DeepMind's AlphaGo), etc.*

2. AI – NN | Artificial Neural Networks with Keras

From Biological to Artificial Neurons

Surprisingly, ANNs have been around for quite a while: they were first introduced back in **1943** by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their landmark paper: "***A Logical Calculus of Ideas Immanent in Nervous Activity***", *McCulloch and Pitts* presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using *propositional logic*. ***This was the first artificial neural network architecture***. Since then many other architectures have been invented, as we will see.

The early successes of ANNs led to the widespread belief that we would soon be conversing with ***truly intelligent machines***. When it became clear in the **1960s** that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. In the early **1980s**, ***new architectures were invented and better training techniques were developed***, ***sparking*** a revival of interest in ***connectionism*** (the study of neural networks). But progress was slow, and by the **1990s** other powerful **Machine Learning** techniques were invented, such as **Support Vector Machine - SVM**. ***These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.***

2. AI – NN | Artificial Neural Networks with Keras

From Biological to Artificial Neurons

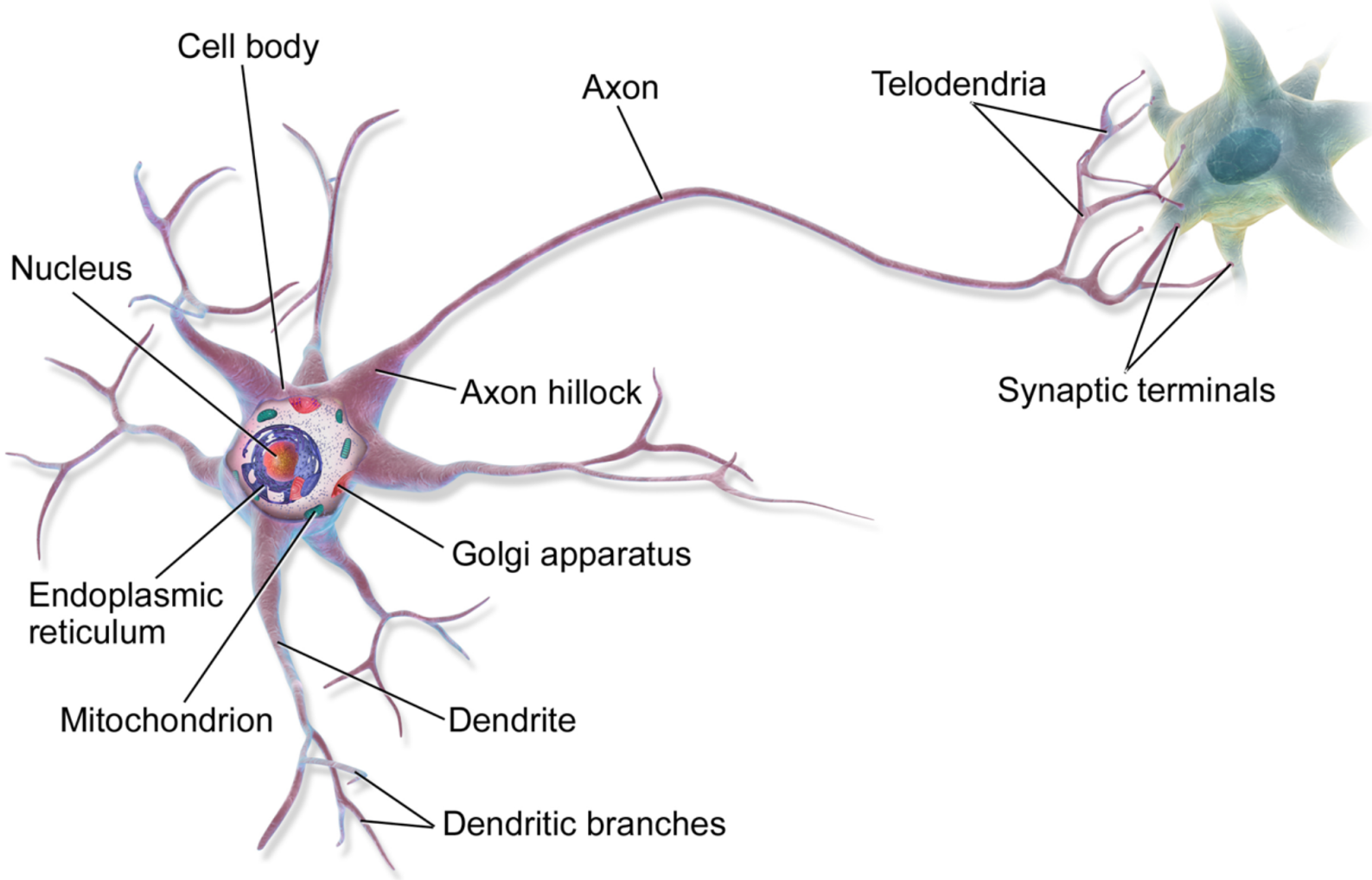
We are now (**2018**) witnessing yet another wave of interest in **ANNs**. ***Will this wave die out like the previous ones did?*** Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

- There is now a ***huge quantity of data (and BigData products)*** available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The ***tremendous increase in computing power since the 1990s*** now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's law (the number of components in integrated circuits has doubled about every 2 years over the last 50 years), but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions. Moreover, ***cloud platforms*** have made this power accessible to everyone.
- ***The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s***, but these relatively small tweaks have had a huge positive impact.
- ***Some theoretical limitations of ANNs have turned out to be benign in practice***. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice (and when it is the case, they are usually fairly close to the global optimum).
- ***ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news***, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products.

2. AI – NN | Artificial Neural Networks with Keras

Biological Neurons

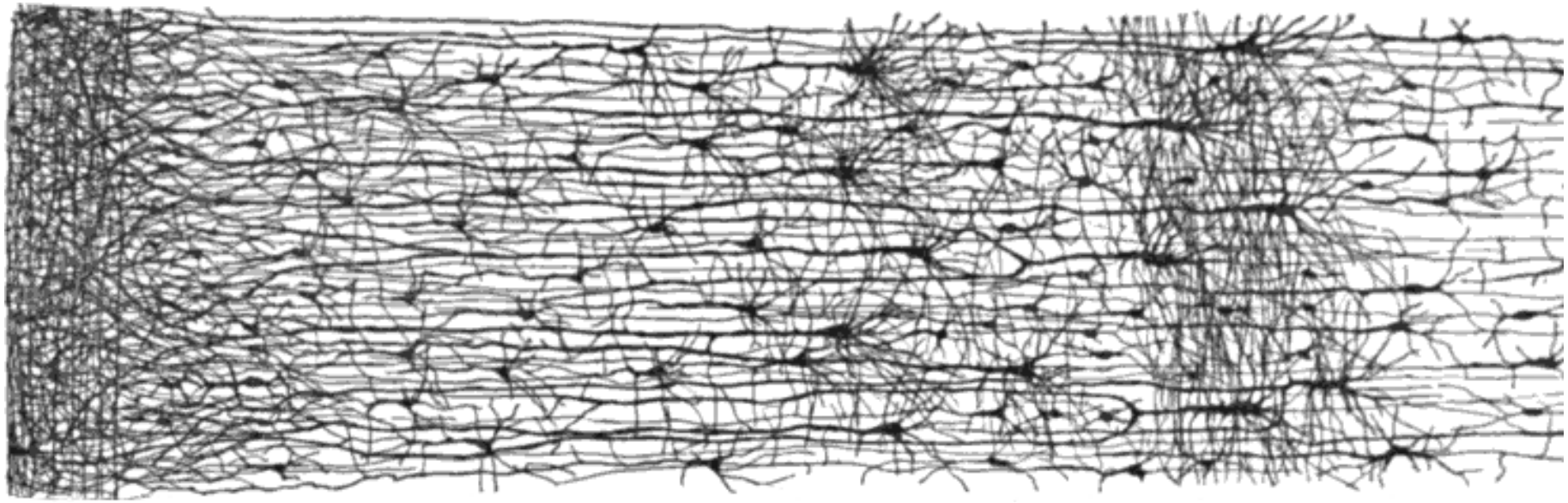
Before we discuss artificial neurons, let's take a quick look at a biological neuron. It is an unusual-looking cell mostly found in animal brains. It's composed of a cell body containing the nucleus and most of the cell's complex components, many branching extensions called dendrites, plus one very long extension called the axon. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called telodendria, and at the tip of these branches are minuscule structures called synaptic terminals (or simply synapses), which are ***connected to the dendrites or cell bodies of other neurons.*** Biological neurons produce short electrical impulses called action potentials (APs, or just signals) which travel along the axons and make the synapses release chemical signals called neurotransmitters. ***When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).***



Biological neuron

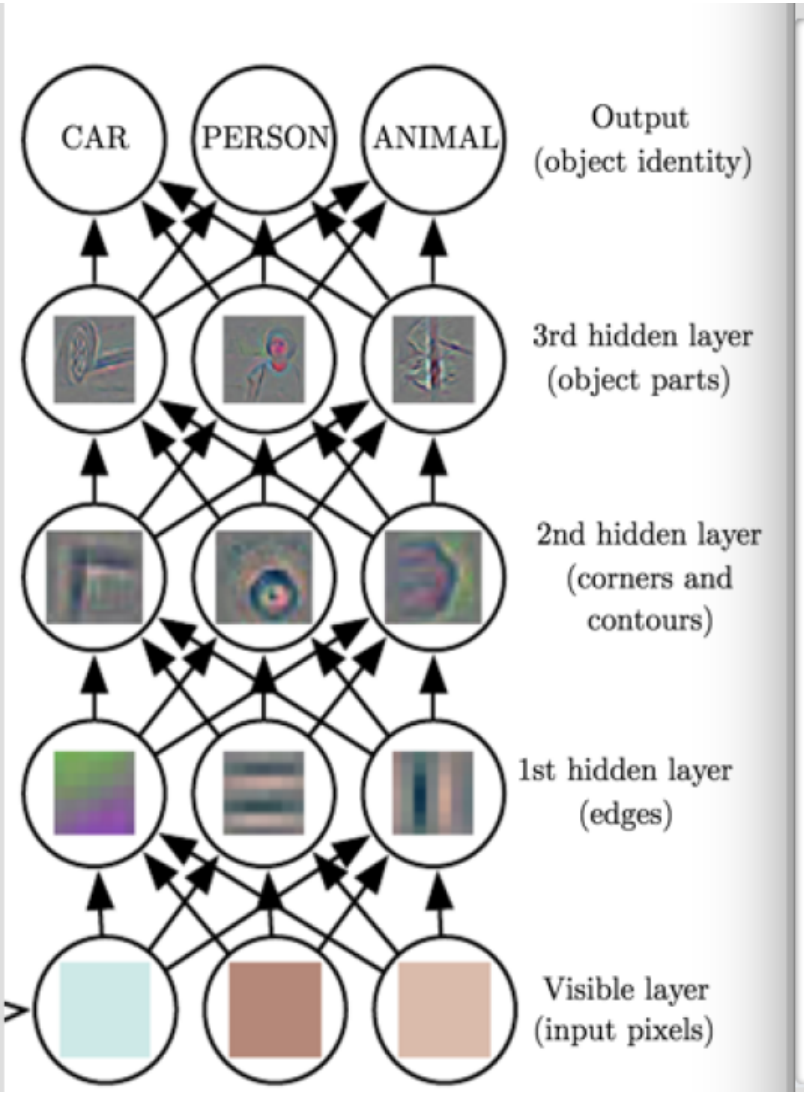
Artificial Neural Networks with Keras

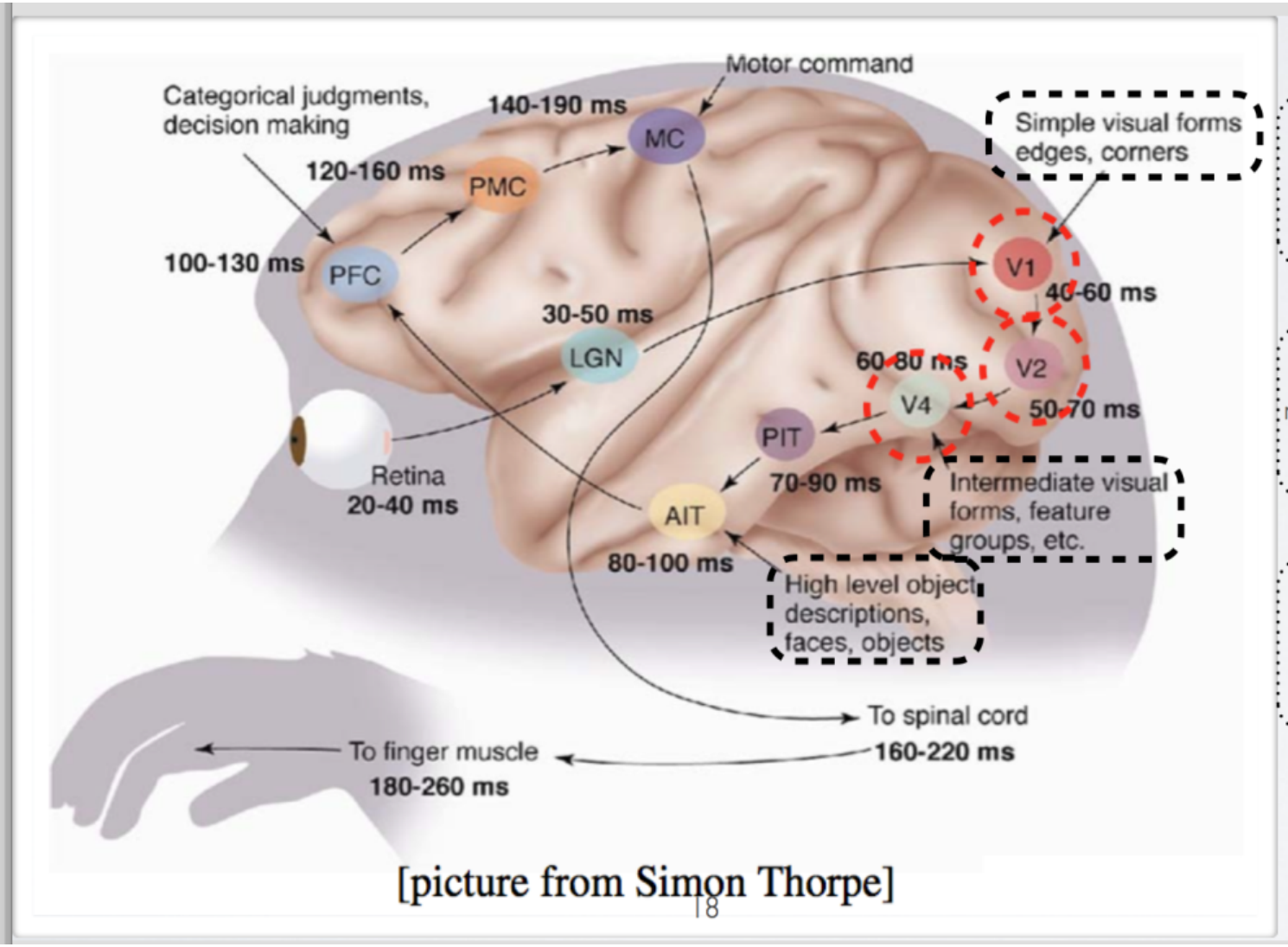
Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of ***biological neural networks (BNNs)*** is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, especially in the cerebral cortex (i.e., the outer layer of your brain):

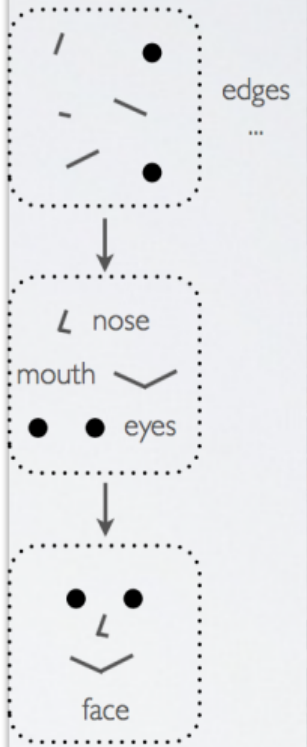


Multiple layers in a biological neural network (human cortex)

2. AI – NN | Artificial Neural Networks



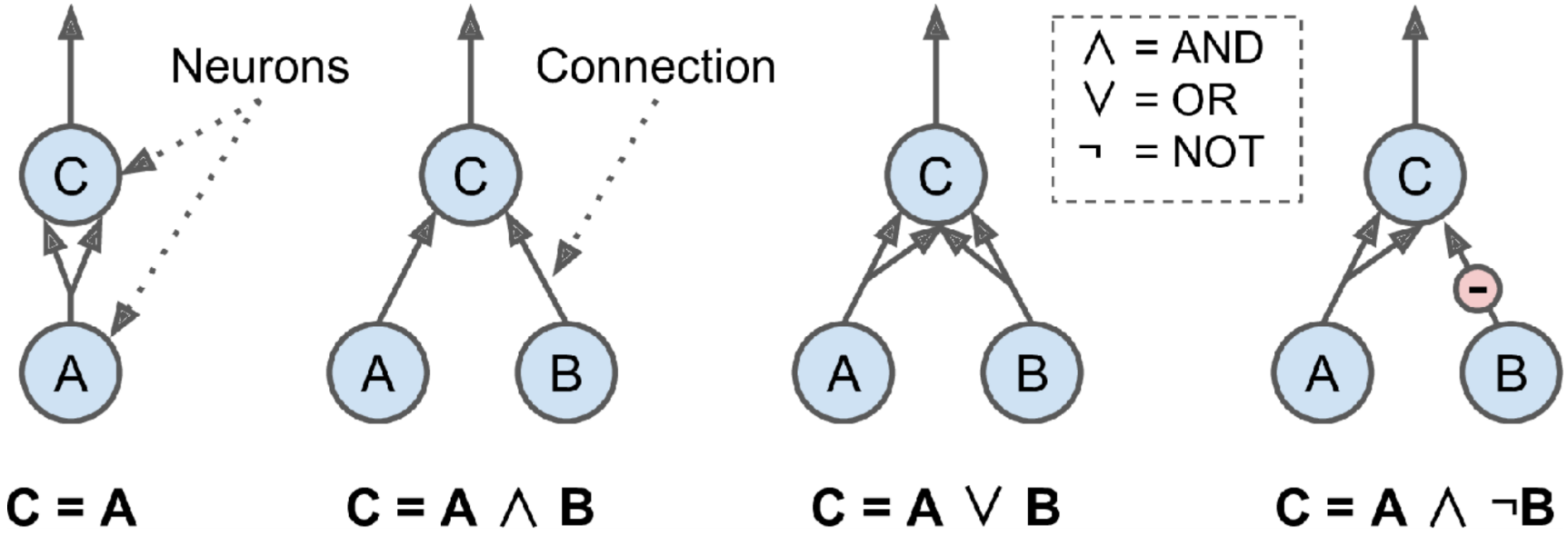




2. AI – NN | Artificial Neural Networks with Keras

Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an ***artificial neuron***: ***it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active***. In their paper, they showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want. ***To see how such a network works, let's build a few ANNs that perform various logical computations, assuming that a neuron is activated when at least two of its inputs are active.***

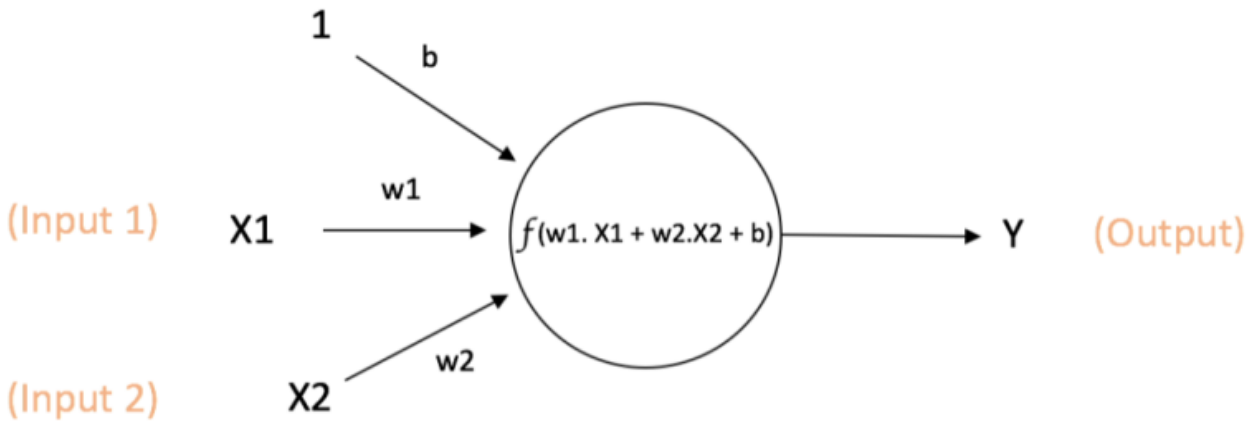


2. AI – NN | Artificial Neural Networks with Keras

The Neuron

A Single Neuron

The basic unit of computation in a neural network is the **neuron**, often called a **node** or **unit**. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated **weight** (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function f (defined below) to the weighted sum of its inputs as shown in Figure 1 below:



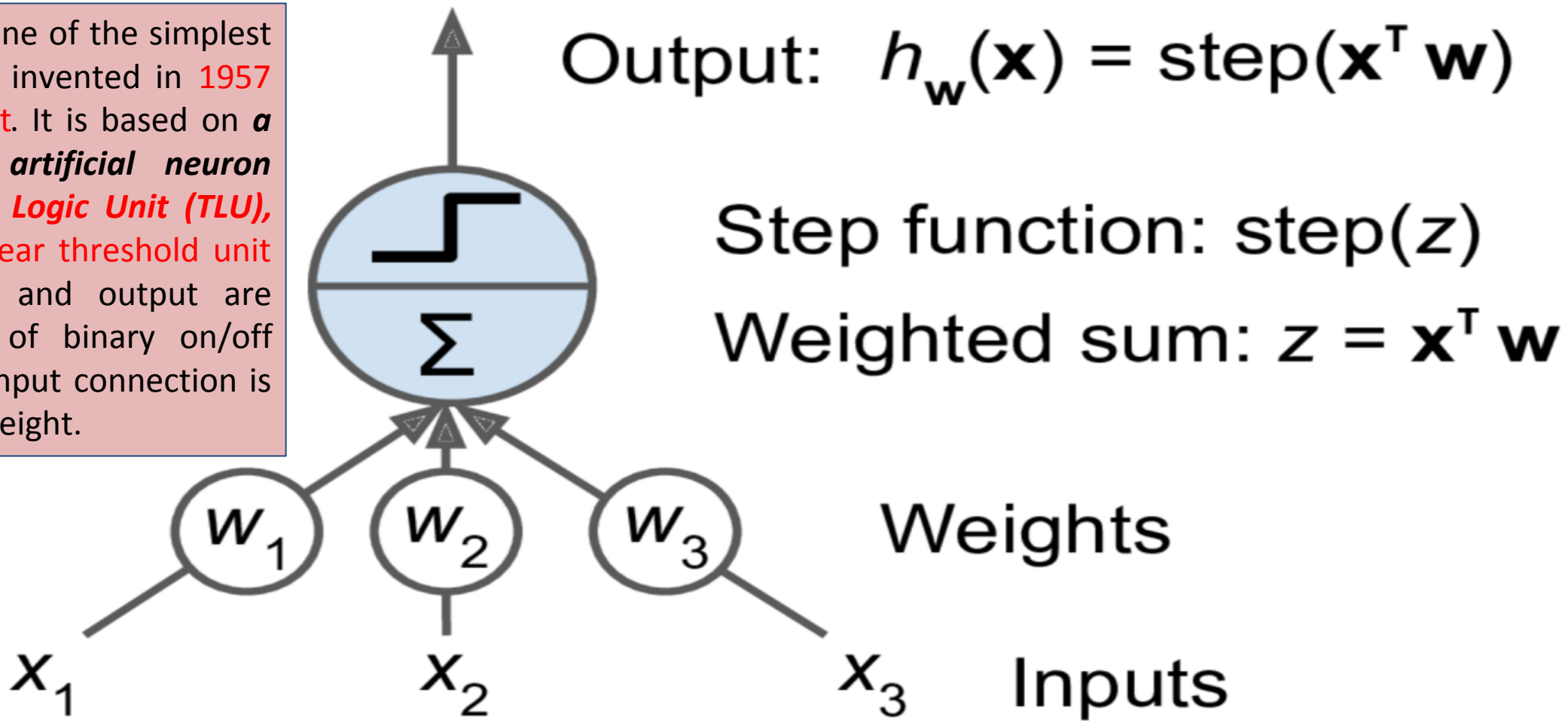
Output of neuron = Y= $f(w1.\ X1 + w2.X2 + b)$

2. AI – NN | Artificial Neural Networks with Keras

The Perceptron

The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs ($z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{x}^\intercal \mathbf{w}$), then applies a *step function* to that sum and outputs the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$, where $z = \mathbf{x}^\intercal \mathbf{w}$.

The Perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on ***a slightly different artificial neuron*** called ***a Threshold Logic Unit (TLU),*** or sometimes a linear threshold unit (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight.



Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a step function

2. AI – NN | Artificial Neural Networks with Keras

The Perceptron step functions

The most common step function used in Perceptrons is the *Heaviside step function* (see Equation 10-1). Sometimes the sign function is used instead.

Equation 10-1. Common step functions used in Perceptrons (assuming threshold = 0)

$$\text{heaviside}\ (z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}\ (z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs, and if the result exceeds a threshold, it outputs the positive class. ***Otherwise it outputs the negative class (just like a Logistic Regression or linear SVM classifier).*** *You could, for example, use a single TLU to classify iris flowers based on petal length and width (also adding an extra bias feature $x_0 = 1$, just like we did in previous sections). Training a TLU in this case means finding the right values for w_0, w_1, and w_2 (the training algorithm is discussed shortly).*

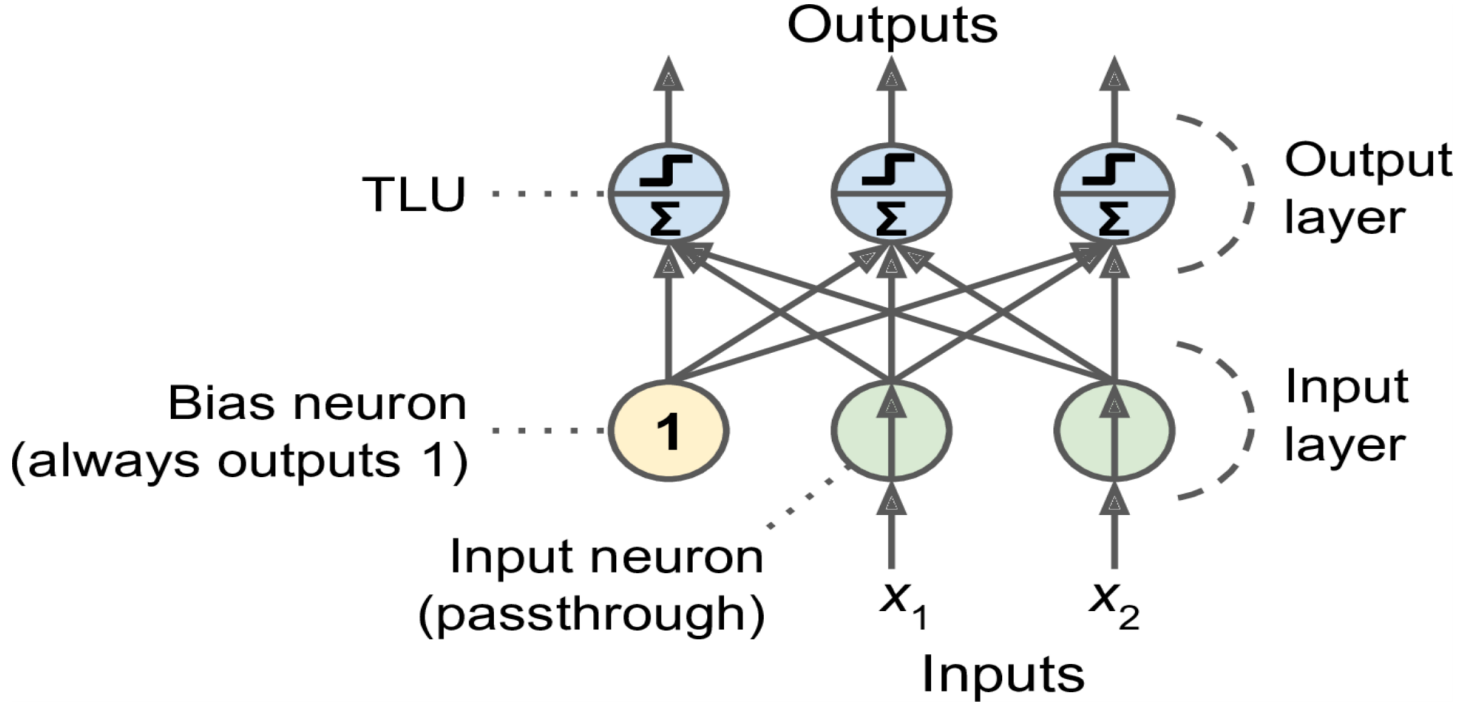
The Perceptron

A Perceptron is simply composed of a single layer of TLUs, ***with each TLU connected to all the inputs***. When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), the layer is called a ***fully connected layer***, or a ***dense layer***. ***The inputs of the Perceptron are fed to special pass-through neurons called input neurons: they output whatever input they are fed***. All the input neurons form the ***input layer.***

Moreover, an extra bias feature is generally added ($x_0 = 1$): it is typically represented using a special type of neuron called a ***bias neuron***, which outputs 1 all the time. ***A Perceptron with two inputs and three outputs (and 1 bias)*** is represented in figure and ***this Perceptron can classify instances simultaneously into three different binary classes, which makes it a multi-output classifier***.

Thanks to the magic of linear algebra, the equation (*Computing the outputs of a fully connected layer*) makes it possible to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

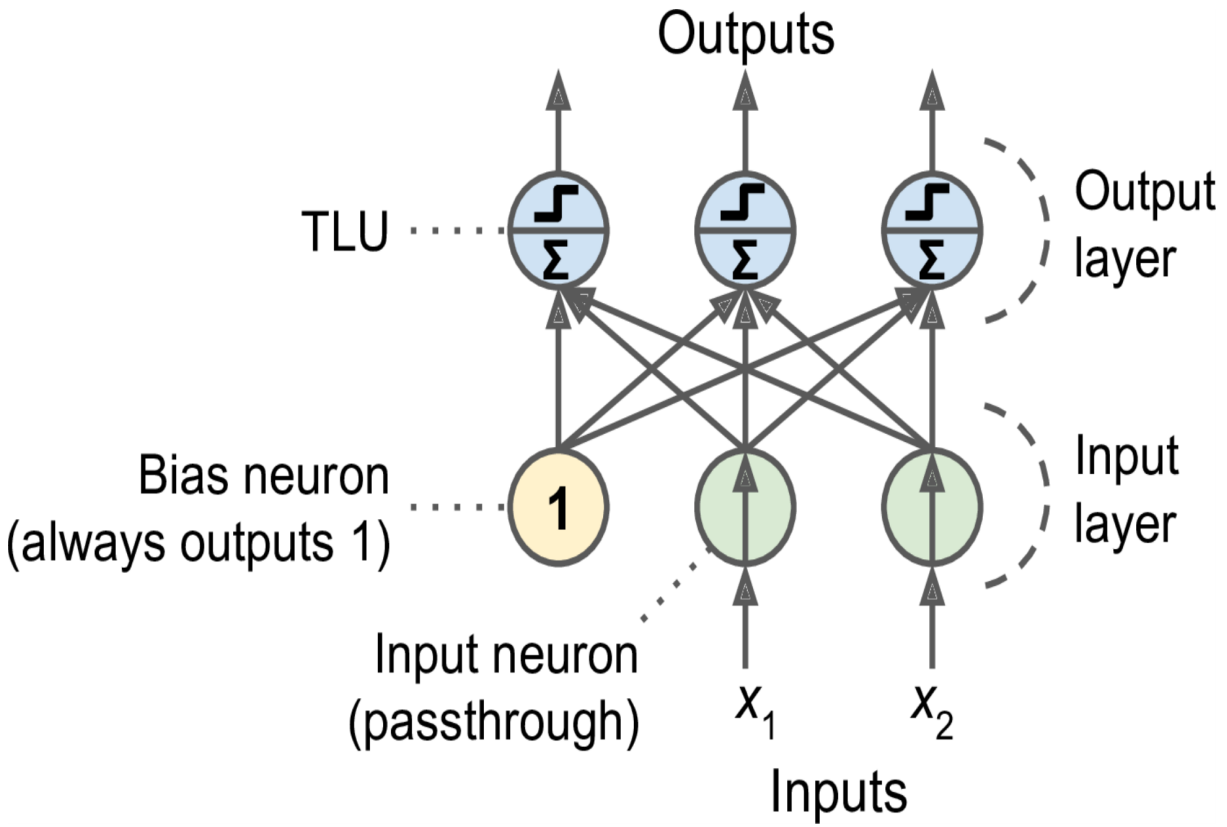


The Perceptron

In this equation:

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

- As always, **X** represents the matrix of input features. It has one row per instance and one column per feature.
- The weight matrix **W** contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector **b** contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function **ϕ** is called the *activation function*: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).



The Perceptron

So, *how is a Perceptron trained?* The Perceptron training algorithm proposed by Rosenblatt was largely inspired by ***Hebb's rule***. In his *1949 book* ***The Organization of Behavior*** *(Wiley), Donald Hebb* suggested that ***when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger***. Siegrid Löwel later summarized Hebb's idea in the catchy phrase, ***"Cells that fire together, wire together"***; that is, the connection weight between two neurons tends to increase when they fire simultaneously. ***This rule later became known as Hebb's rule (or Hebbian learning).*** Perceptrons ***are trained using a variant of this rule*** that takes into account the error made by the network when it makes a prediction; ***the Perceptron learning rule reinforces connections that help reduce the error.*** More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta \left(y_j - \hat{y}_j\right) x_i$$

In this equation:

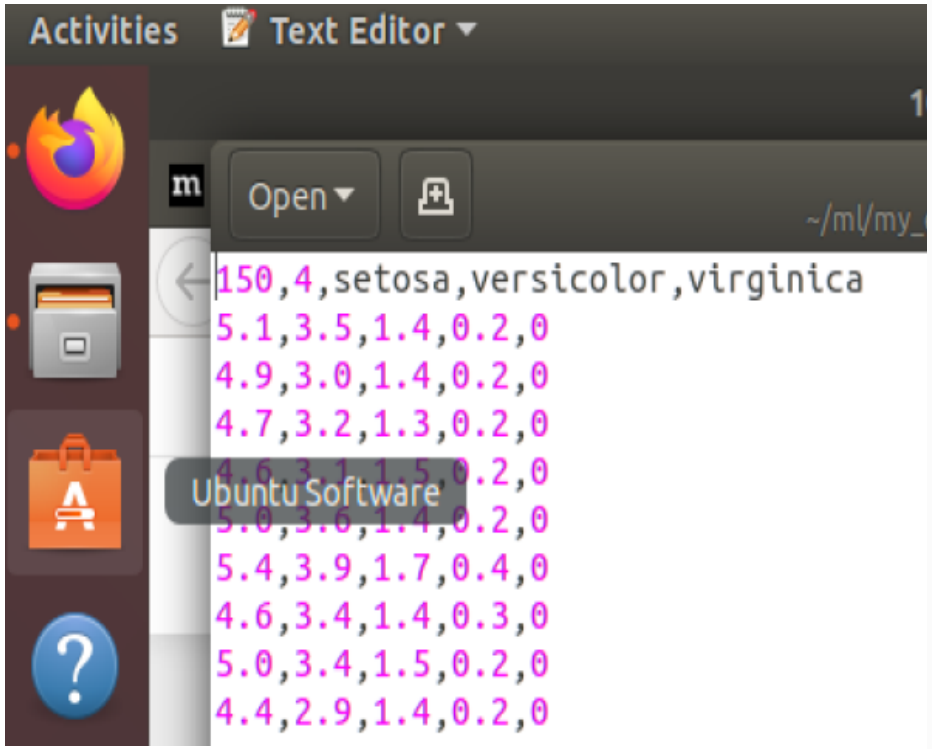
Perceptron learning rule (weight update)

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution. This is called the ***Perceptron convergence theorem***.

The Perceptron

Scikit-Learn provides a Perceptron class that implements ***a single-TLU network***. It can be used pretty much as you would expect: for example, on the iris dataset (introduced in previous sections as **MNIST** - Modified National Institute of Standards and Technology database)



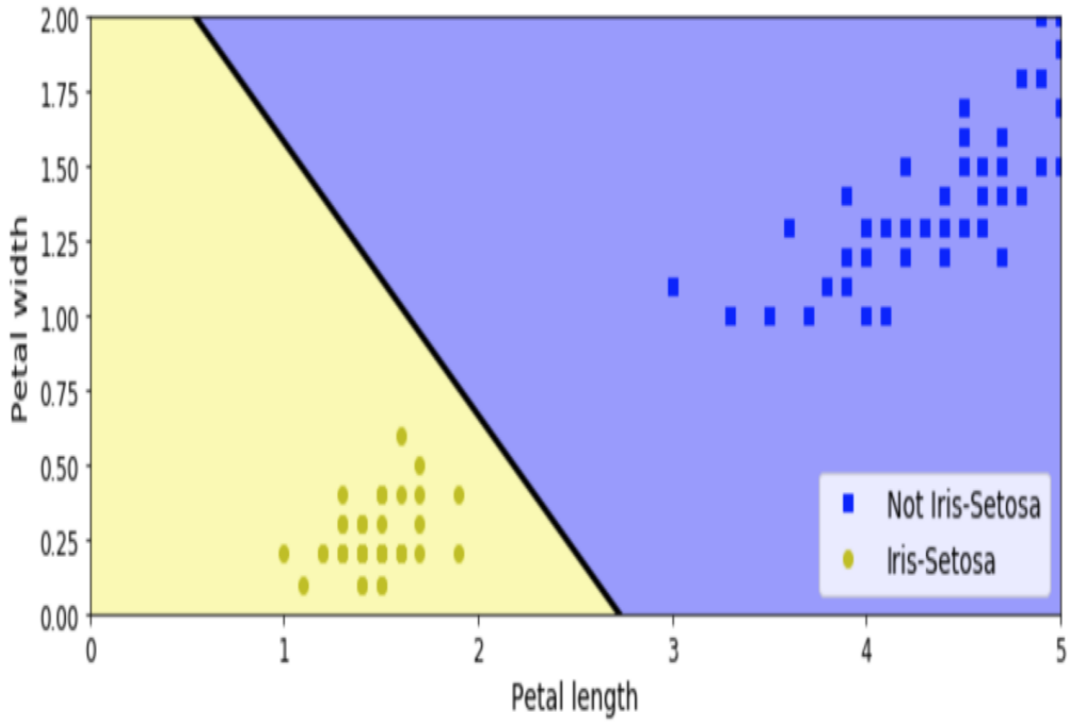

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

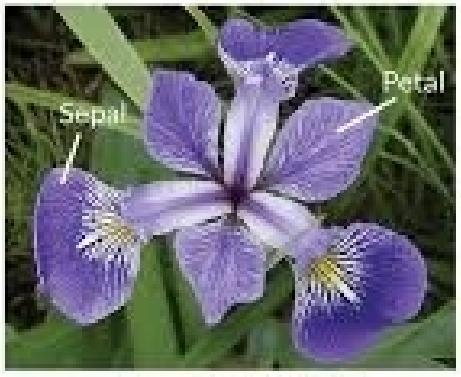
2. AI - NN | Artificial Neural Networ

The Perceptron

```
a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
        np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
        np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]
y_predict = per_clf.predict(X_new)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

plt.plot([axes[0], axes[1]], [a * axes[0] + b, a * axes[1] + b], "k-", linewidth=3)
from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)

save_fig("perceptron_iris_plot")
plt.show()
```

Saving figure perceptron_iris_plot





Iris Versicolor

Iris Setosa

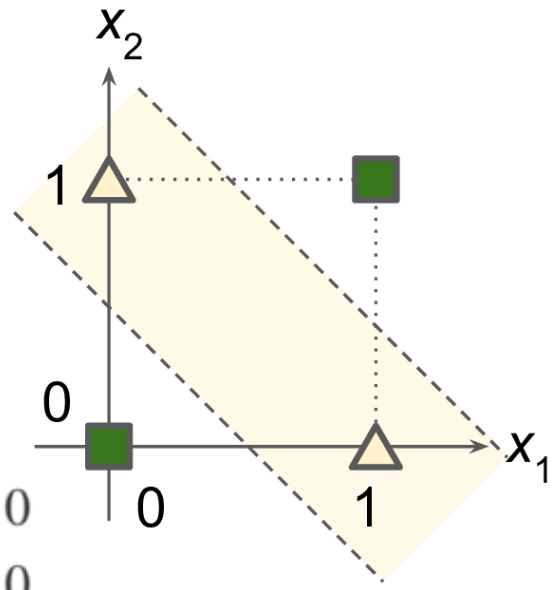
Iris Virginica

MLP - Multi-Layer Perceptron

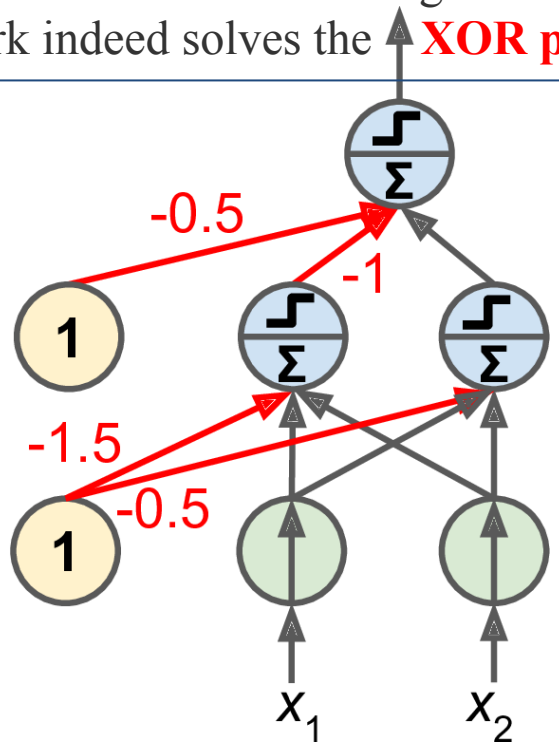
In their **1969** monograph Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons—in particular, the fact that ***they are incapable of solving some trivial problems (e.g., the Exclusive OR (XOR) classification problem***; see the left side of figure). ***This is true of any other linear classification model (such as Logistic Regression classifiers)***.

It turns out that some of the limitations of Perceptrons can be eliminated by **stacking multiple Perceptrons**. ***The resulting ANN is called a Multilayer Perceptron (MLP)***. An MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the right side of figure: with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. All connections have a weight equal to 1, except the four connections where the weight is shown. Try verifying that this network indeed solves the **XOR problem**!

Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0



$$\text{heaviside}\ (z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



(-0.5 => **0**)

(-0.5*1 + 1*(-1) + 1)

(0.5 =>1 , 1.5 => 1)

(-1.5+1*1+1,-0.5+1*1+1)

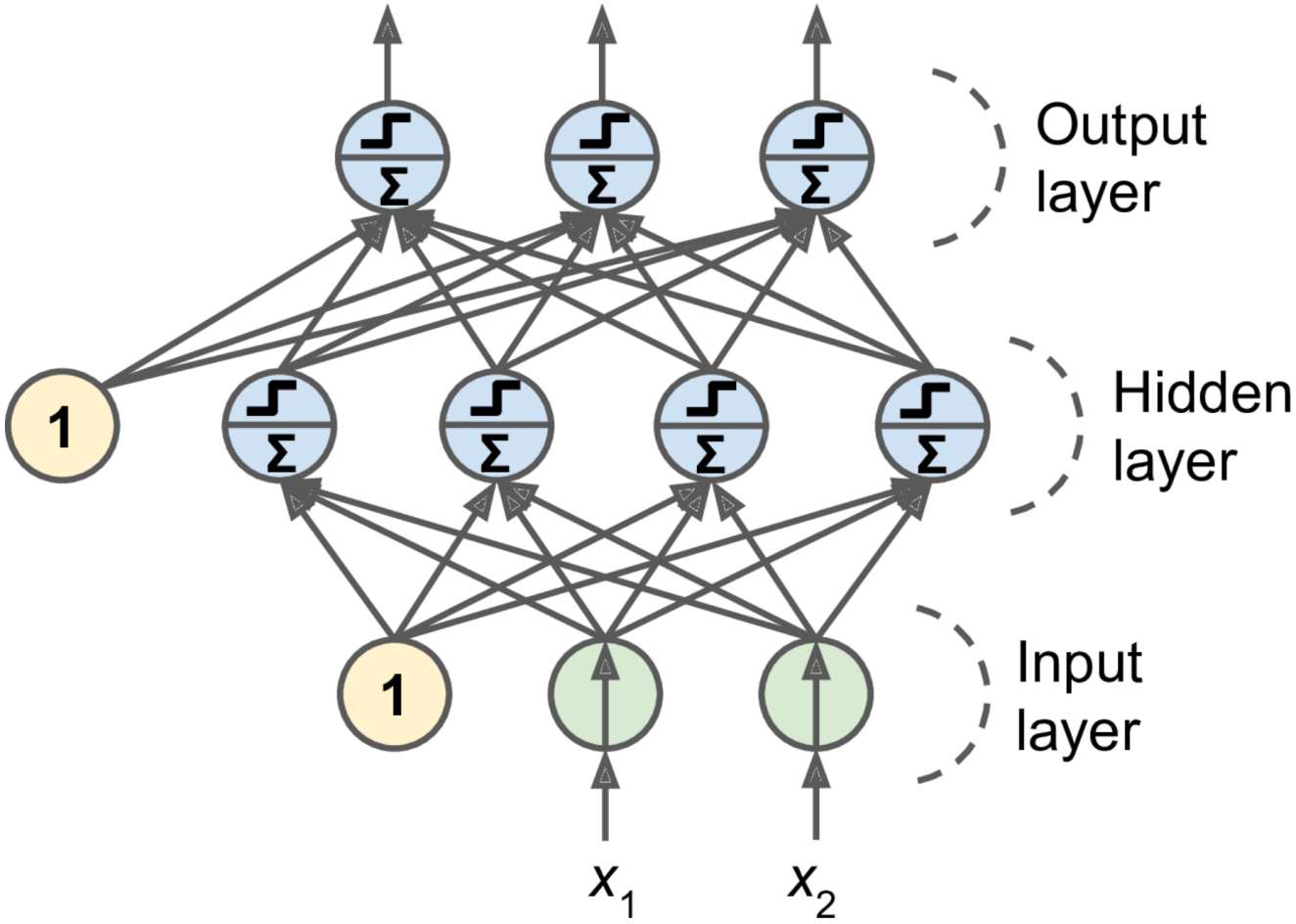
(**1** , **1**)

MLP -Multilayer Perceptron & Back-propagation

An MLP is composed of:

- one (passthrough) ***input layer***,
- one or more layers of TLUs, called ***hidden layers***, and
- one final layer of TLUs called the ***output layer***.

The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a ***feed-forward neural network (FNN).***



Architecture of a Multilayer Perceptron with two inputs, one hidden layer of four neurons, and three output neurons (the bias neurons are shown here, but usually they are implicit)

MLP -Multilayer Perceptron & Back-propagation

When an **ANN** contains a ***deep stack of hidden layers***, it is called a **deep neural network (DNN).** The field of Deep Learning studies DNNs, and more generally models containing deep stacks of computations.

Even so, many people talk about Deep Learning whenever neural networks are involved (even shallow ones).

For many years researchers struggled to find a way to train MLPs, without success. ***But in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper that introduced the back-propagation training algorithm, which is still used today.*** In short, it is **Gradient Descent** (introduced in previous sections) using an efficient technique for computing the gradients automatically:

- in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regard to every single model parameter.

*In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error. Once it has these gradients, it just performs a regular **Gradient Descent** step, and the whole process is repeated until the network converges to the solution.*

MLP -Multilayer Perceptron & Back-propagation

In order for this algorithm (Back-propagation) to work properly, its authors made a key change to the MLP's architecture: they ***replaced the step function*** with the ***logistic (sigmoid) function***, $\sigma(z) = 1 / (1 + \exp(-z))$.

In fact, the **back-propagation algorithm** works well with many other activation functions, not just the logistic function. Here are two other popular choices:

The hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$

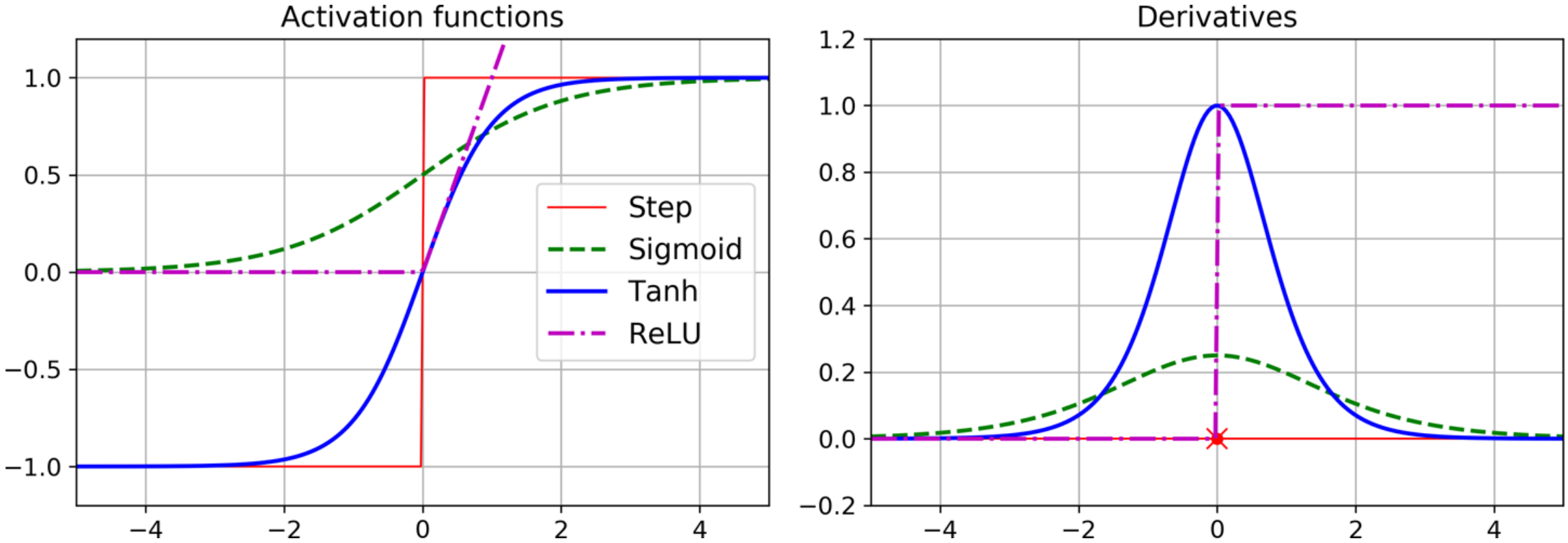
Just like the logistic function, this activation function is *S*-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function). That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

The Rectified Linear Unit function: $ReLU(z) = \max(0, z)$

The ReLU function is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default.[12] Most importantly, the fact that it does not have a maximum output value helps reduce some issues during Gradient Descent (we will come back to this in Chapter 11).

MLP - Multilayer Perceptron & Back-propagation

These **popular activation functions** and their derivatives are represented in figure:



Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

OK! You know where neural nets came from, what their architecture is, and how to compute their outputs.

You've also learned about the back-propagation algorithm. But what exactly can you do with them?

Regression & Classification!

Regression MLP - Multilayer Perceptron

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with four output neurons.

In general, when building an MLP for regression, you do not want to use any activation function for the output neurons, so they are free to output any range of values. If you want to guarantee that the output will always be positive, then you can use the ReLU activation function in the output layer.

Alternatively, you can use the ***softplus** activation function*, which is a smooth variant of ReLU: $\text{softplus}(z) = \log(1 + \exp(z))$. It is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and then scale the labels to the appropriate range: 0 to 1 for the logistic function and –1 to 1 for the hyperbolic tangent.

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both.

2. AI – NN | Artificial Neural Networks & Deep ML

Regression MLP – **Multilayer Perceptron**

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLP - Multilayer Perceptron

MLPs can also be used for classification tasks. For a ***binary classification problem***, you just need a single output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

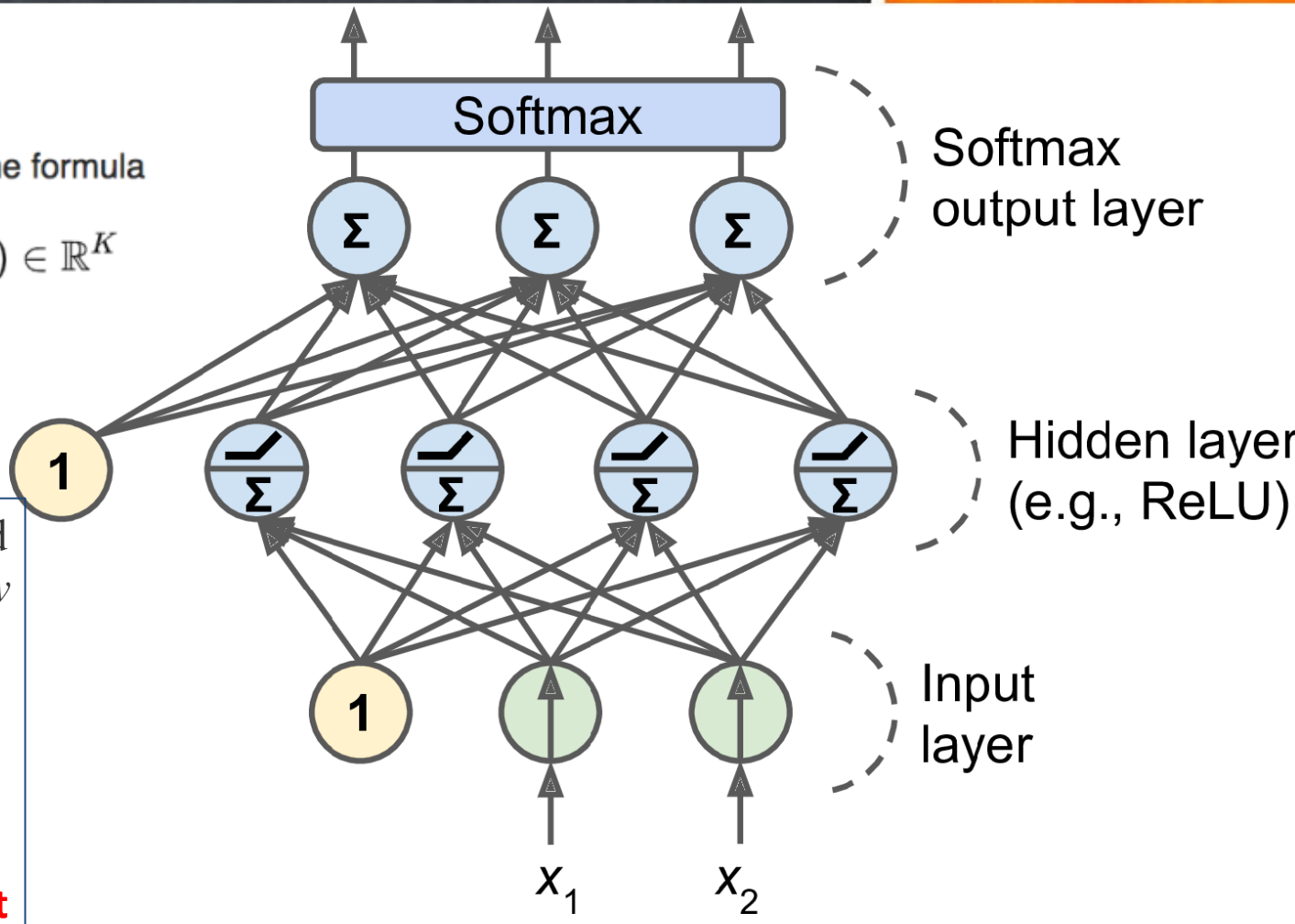
MLPs can also easily handle mu***lti-label binary classification tasks*** (see **Classification**). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or non-urgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you can have non-urgent ham, urgent ham, non-urgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see figure). The **softmax** function (introduced in Training Models) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (which is required if the classes are exclusive). This is called **multiclass classification**.

Classification MLP - Multilayer Perceptron

The standard (unit) softmax function $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ is defined by the formula

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$$



A modern MLP (including ReLU and softmax) for classification

Play with various neural network architectures and visualize their outputs using the *TensorFlow Playground:*

https://playground.tensorflow.org/
https://github.com/tensorflow/playground

Now you have all the concepts you need to start implementing MLPs with Keras! BUT after going through full numeric example and returning to the first JAVA example from the previous slides.

Classification MLP - Multilayer Perceptron

Table 10-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

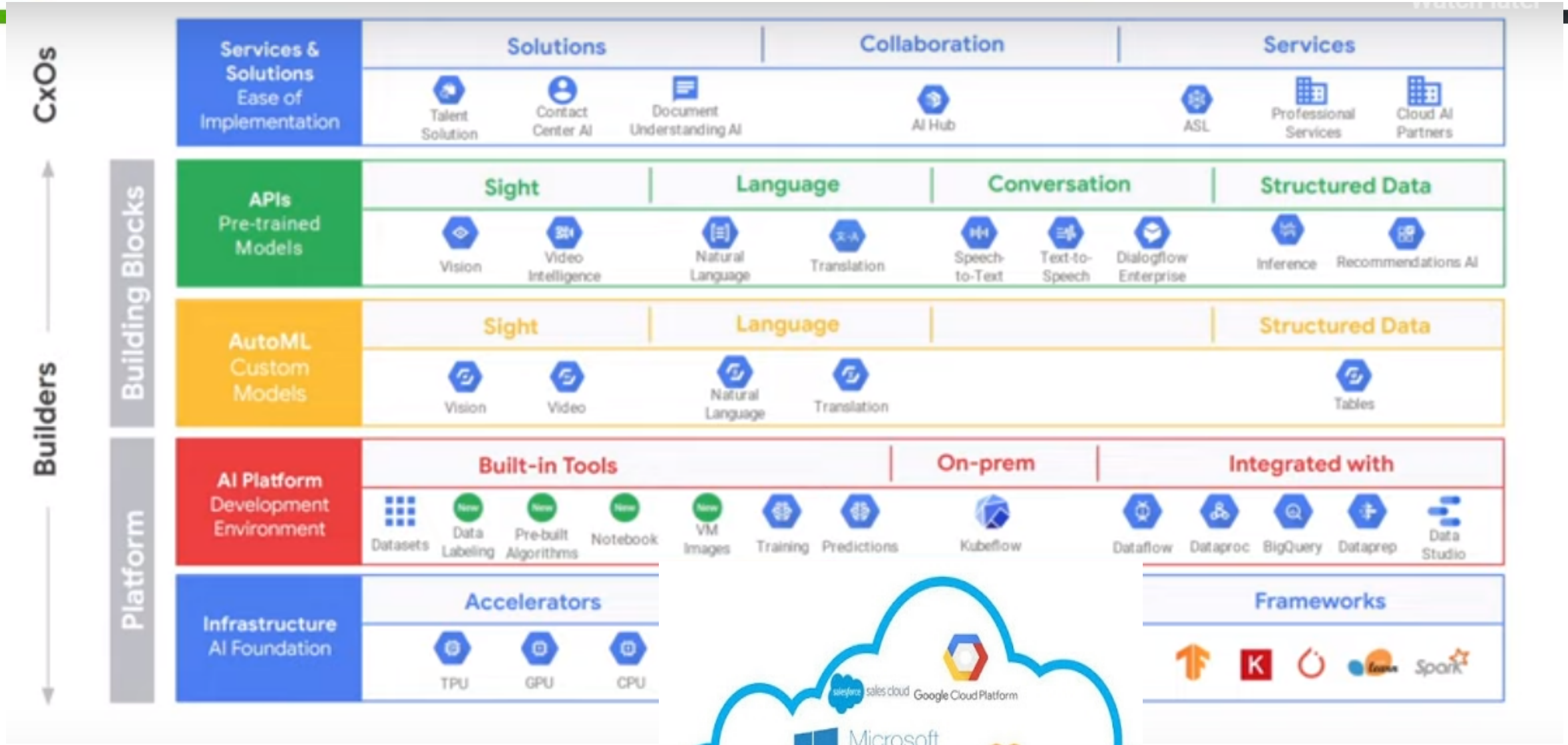
How would be the future on the A.I / M.L. & Deep Learning?

Communicate & Exchange Ideas

Tools & Install – Python, PIP, Jupyter, SciKit, TenserFlow, Keras

```
# sudo rm /var/cache/apt/archives/lock # python --version
# sudo rm /var/lib/dpkg/lock # install python if necessary
sudo apt install python-pip
sudo apt-get install python3-pip
sudo pip install --upgrade pip
sudo pip3 install --upgrade pip

python3 -m pip --version
python3 -m pip install --user -U virtualenv
$ export ML_PATH="$HOME/ml" # You can change the path if you prefer
$ mkdir -p $ML_PATH
$ cd $ML_PATH
$ python3 -m virtualenv my_env

$ source my_env/bin/activate # on Linux or macOS
python3 -m pip install -U jupyter matplotlib numpy pandas scipy scikit-learn
python3 -m ipykernel install --user --name=python3

python3 -m pip install -U tensorflow & python3 -m pip install -U pydot & python3 -m pip install -U graphviz
sudo apt-get install graphviz
$ jupyter notebook
```

Artificial Intelligence – A.I. & ML Cloud Providers



Q & A


thank you
danke
謝謝
рахмат
ngiyabonga
teşekkür ederim
gracias
merci
obrigado
спасибо
Баярлалаа
dziękuję
bedankt
sukriya
kop khun krap
go raibh maith agat
mochchakkeram
arigatô
grazie
terima kasih
감사합니다
ありがとう
dank je
tapadh leat
dakujem
мерси
takk
sagolun
taiku
hvala
mauruuru
faafetai lava
kiitos
dankie
dhanyavad
nanni
nandri
enkosi
bayarlalaa
gràcie
köszönöm
sobodi
dekuji
mesi
didi madloba
kam sah hamnida
najis tuke
rahmat
chnorakaloutioun
gratias ago
gràcies
tanemirt
rahmet
xièxie
ευχαριστώ
diolch
dhanyavadagalu
shukriya
merce
trugarez
mamnun
дякую
djiere dieuf
sulpáy
akun dankon aciu
tau
mersi
kia ora
barka
welalin
tack
spas
vinaka
спасибі
blagodaram
misaotra
matondo
paldies
grazzi
mahalo
хвала
asante
manana
obrigada
murakoze
tenki
chokrane

www: ism.ase.ro | acs.ase.ro | dice.ase.ro

Scan the Tag
to get the web
Mobile Address