Advanced machine learning

Mattias Ermakov Thing

April 18, 2024

1 Introduction, GitHub, and neural networks

The first lecture is focused on establishing prerequisites and leveling out the playing field such that everyone is on the same page as we dive into the details in the following lectures. We take a look at the structure of the course and GitHub where the course material is located. Then we will review the basics of neural network which will be the focus of this course.

1.1 Course overview

This course aim to introduce the students to various of methods of machine learning, primarily focused on neural networks and ways of training them. By the end of the course students should know about various types of neural network architectures and when to use a given architecture. We also look at ways to improve neural networks with physics guidance. The course consists of five two hour lectures with the following content:

- Introduction, GitHub, and neural networks
- Machine learning and methods of training
- Neural network architectures
- Physics guidance in neural networks
- TBD

Each lecture will come with relevant code and exercises to prepare for the next lesson. Although the exercises are not mandatory, it is highly recommended to do as it will improve practical skills. After the first lesson we will start the lectures by discussing the exercise from the previous lecture. The course material is available at: `https://github.com/CP3-Origins/advanced-machine-learning`. Beyond the lecture notes Google is your friend.

1.2 Prerequisites

This course will use Python and basic programming skills in this language is expected. To develop machine learning algorithms we will use `Keras` and `TensorFlow 2` [2, 1], which means students must have Python 3.8 installed or a newer version. The university provides the paid version of PyCharm as an IDE for Python, but you can use whatever for writing code.

The required packages for the course can be found in the `requirement.txt` file in the `code` folder. From that directory you can install the packages with pip using: `pip install -r requirements.txt`

The course material is located on GitHub (see the link in the course overview), and students can find the lecture notes in the `lecture_notes` folder. The relevant code be found in the subfolders of the `code` folder.

Students are expected to know calculus, linear algebra, and the basics of neural networks. This course builds on concepts from FY555 (Introduction to Python, machine learning and data handling for the physical sciences), but it is not necessary to have taken that course to follow.

1.3 Introduction to Git

All course material for this course is on GitHub and it is recommended you know how to access this material. GitHub is a developer platform using Git to manage projects. A link to the homepage of Git can be found here: `https://git-scm.com/`.

Git is a free and open source distributed version control system. This means that it helps us manage versions of project and collaborate in an asynchronous fashion. Each contributor has a local version of the repository, and can develop locally on that version, and then share changes in a controlled and versioned manner. Git is a key tool for code development and if you ever plan to work in IT, it is something you need to know. It can also be very useful for doing scientific work in groups if code is involved. Also, it allows you to save code and other files in the cloud as a backup. Two of the most popular platforms for using Git is GitHub and GitLab. In this course we use GitHub. Students are encourages to study Git and version control.

There are five main commands you will need to use Git: `git clone`, `git pull`, `git add`, `git push`, and `git push`.

To get started you need to download and install Git. Having installed Git you can clone the course repository. This can be done several ways:

- Using a Git plugin in your IDE
- With a `git clone` command in your terminal
- By downloading the repository as a zip file (not recommended)

If you just download the repository as a zip file, you an not leveraging Git. This method works if you simply want to download the course content, but will not teach you how to use Git.

In the following we will consider how to work with Git using the terminal. Assume you want to download the course content from GitHub. Then you would go to the directory where you want to download the repository and execute the following command:

```
git clone https://github.com/CP3-Origins/advanced-machine-learning.git
```

This will download the repository on your device. Since this course will be updated and improved throughout the course, there might be changes to the course. To update to the latest version you can use the following command anywhere inside the Git project by executing the `git pull` command.

This will however not work, if you have locally modified a part of the repository, which you are trying to update. This is to avoid overwriting changes when merging the new file with your local files. In this case you need to resolve any merge issues, such that the file are merged in the desired way without having loosing any work due to file overwriting.

If you plan to participate in exercises and develop your own networks, then it is recommended that you start by creating a fork of the repository. This will create a copy of the repository which is on your own GitHub account. The only downside is that you need to remember update your own fork to get the updates from the main repository on CP3. Other than that you can clone your forked repository and use pull to update your repository.

Suppose you have made your own script with some neural network as part of an exercise, you can now upload that to your fork and save your changes to the cloud. In this case, if you loose your computer, you will still have everything from this course including any code you have created.

To do this you need to push your code from you local machine to your fork on GitHub. To get an overview of file changes in the Git project, you can use the command `git status`. Then you can add the files you want to save using the command `git add <path/to/file>`. You can specify a folder or file which you want to add. You can execute `git add .` if you want all changes to be added.

Next, you need to commit the code to Git. This is for version control. Together with this it is recommended that you add a short note on what you change does. And example would be `git commit -m "Adds a network to model rainfall"`, where `-m` is the message flag and the part in double quotation is the commit message.

Finally, you can use the push command to push the changes using the command `git push`. This will push your local changes to your fork and you should be able to see the changed if you open your repository on GitHub.

The reason for using a fork is that it creates a personal copy for each

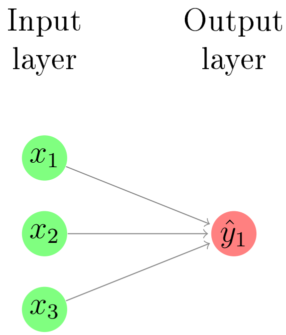

Figure 1: *A simple neural network with an input layer, and a single neuron in the output layer.*

student. We cannot have each student to save their work on the main course repository. This would also cause collisions if students used the same file names. You own your fork and you can invite colleagues to your participate if you want to work on the exercise together.

1.4 Neural networks

Before going into a more thorough overview of machine learning (ML) methods, let us recall the basics of neural networks and look a implementing neural networks. In the next lecture we will get back to a general discussion of neural networks.

1.4.1 Theory

As the name suggest neural networks are composed of neurons similar to our brain. Any neuron network uses the same basic building block, the single neuron. In some applications such as in robots less than 100 may be sufficient [6], whereas image classification networks may contain thousands of neurons [10]. Thus, to understand neural networks we should take our time to understand how they work. Neurons are fundamentally doing linear regression, and later we discuss how nonlinear behavior is obtained as this is key to the power of neural networks. As a simple example, one can consider a single neuron "network" shown in Figure 1. This "network" takes in three inputs, x_i, where i = 1,2,4, and computes a predicted value $\hat{y}$. The word network is in quotation as one would need multiple interconnected neurons to properly talk of a network.

In the case of this single neuron with three inputs shown in Figure 1 the equation becomes,

$$z(x_i) = w_1x_1 + w_2x_2 + w_3x_3 + b, \tag{1.1}$$

where each input is associated with the weight w_i which modulates the influence of the input, and then there is a bias term, b. This can be written in

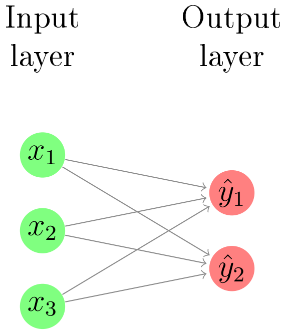

Figure 2: *A simple neural network with an input layer, and two neurons in the output layer.*

a compact matrix form as,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b \end{bmatrix} = w_{ij}x_i + b_j \cdot \vec{1}, \tag{1.2}$$

where an additional j index is added to indicate the number of neurons, but since there is only one neuron it is redundant. The use of this index becomes apparent when considering a network like in Figure 2, but with two fully connected output nodes, $j = 2$. In this case, the equation would be,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = w_{ij}x_i + b_j \cdot \vec{1}. \tag{1.3}$$

This highlights the benefit of the notation with the weight matrix as w_{ij}, the input vector as x_i, and the bias vector as b_j. A technical note would be that, in practice, it is more efficient to skip the process of adding the bias terms explicitly. One can exploit the definition of matrix multiplication and write it as,

$$z_j(x_i) = \begin{bmatrix} w_{11} & w_{21} & w_{31} & b_1 \\ w_{12} & w_{22} & w_{32} & b_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} = w_{ij}x_i + b_j \cdot \vec{1}. \tag{1.4}$$

From this we can then also consider more complex networks with hidden layers as in Figure 3. In this case we would first compute the first the values at the hidden layers $z_1(x_i) = a_i$ and then we can use that result to compute that result $z_2(a_i) = \hat{y}$.

A linear equation is not sufficient to produce complex nonlinear decision boundaries, therefore it is key to modify the result $z_j(x)$ such nonlinearity is achieved. This is very simply to prove, consider our simple network with one

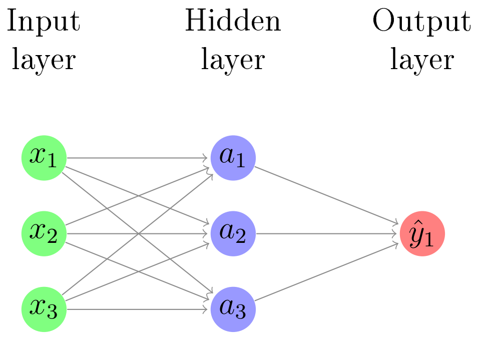

Figure 3: *A simple neural network with an input layer, a hidden layer, and a single neuron in the output layer.*

hidden layers. The math becomes,

$$\hat{y} = z_2(z_1(x_i)) = w_{i2}\left(w_{i1}x_i + b_1 \cdot \vec{1}\right) + b_2 \cdot \vec{1}. \tag{1.5}$$

The problem here is that while we have different weights and biases, due to the lack of nonlinearity, this hidden actually achieve nothing because we can redefine this equation to be just like a network without a hidden layer,

$$\hat{y} = z_2(z_1(x_i)) = \underbrace{w_{i2}w_{i1}x_i}_{w_{i1}x_i} + \underbrace{w_{i2}b_1 \cdot \vec{1} + b_2 \cdot \vec{1}b_1 \cdot \vec{1}}_{b_1\cdot\vec{1}}. \tag{1.6}$$

This is perfectly valid because the latter terms is nothing but a sum of trainable variable, which is nothing but a number, and then first part contains the input times two different weight, but again we can just redefine that as a single weight. Therefore, without any nonlinearity we can add an arbitrary number of hidden layers and achieve the equivalent of having no hidden layers.

This is fixed using nonlinear activation functions, $a(z)$, such as the sigmoid function, $\sigma(x)$, and the rectifier linear (ReLu) function, ReLu(x). There are many more options, and it is up to the designer of the network to determine the best activation function. The activation function, f, can be applied on a per-neuron basis, but it is commonly applied across the layers,

$$a(z_j(x_i)) = f(w_{ij}x_i + b_j \cdot \vec{1}). \tag{1.7}$$

Each layer of the neural network is generally computed in this way. One can then stack the layers, $a(z_j)^n$, where n is the number of layers. Deep neural networks then have three layers or more, $n \geq 3$. The output of the neural network is then computed by iteratively computing the layers from input to output, with the previous layer being the input to the next layer.

1.4.2 Tensorflow and keras

For this course we will be using `Keras` and `TensorFlow 2` [2, 1] to develop neural network. There are also other packages such as `PyTorch` [7] which you are free to use, but it will not be used in the example code of this course. At the end of the day it is all linear algebra, but there are some differences in the packages. In general `Keras` provides a more beginner-friendly interface, but we will introduce advanced features which can be done in both `TensorFlow` and `PyTorch`. Some of the exercises are will be suitable to do in `PyTorch` and you are welcome to do so.

This weeks exercise is about introducing the basics of coding with `Keras` and `TensorFlow 2`. This example has also been used in FY555, but in this course we will consider a different exercise.

For this first part we will use `Keras` and some `sklearn` functions to started. First we import dependencies:

```python
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical

```

Then we download the iris dataset from `sklearn` as our simple training set, and convert the text labels to numbers using encoding.

```python
# Data (as pandas dataframes)
X = load_iris().data
y = load_iris().target

# Convert string labels to numeric values using Label Encoding
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)

# Convert labels to one-hot encoding for categorical crossentropy
y_one_hot = to_categorical(y)

```

Next step is to split the dataset into training and test. The input data is then normalized before testing.

```python
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot, test_size=0.2, random_state=42)

# Standardize the features (optional but often recommended)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

```

In the following section we create the network using the `Sequential` method. The network consists of a hidden layer with 8 nodes and a final layer of three nodes for each possible classification category. For the hidden layer ReLu functions are used, and for the final layer the softmax activation function is used for classification.

```python
# Create a sequential model
model = Sequential()

# Add the input layer and the first hidden layer
model.add(Dense(units=8, input_dim=4, activation='relu'))

# Add the output layer with softmax activation for classification
model.add(Dense(units=3, activation='softmax'))

```

After defining the model we compile it with an optimizer, in this case the `adam` optimizer, and then we define a loss function and metrics. Then we train the model with fit and split the training dataset into the training and validation set. Then train with batch sizes of 8 across 200 epochs.

```python
# Compile the model with categorical crossentropy loss for multi-class classification
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=200, batch_size=8, validation_split=0.2)

```

Having trained the model, we can evaluate the performance using the test dataset and then print the result.

```python
# Evaluate the model on the test set
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")

# Plot training and validation loss over epochs
plt.figure(figsize=(12, 6))

# Plot training & validation loss values
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'], loc='upper right')

# Plot training & validation accuracy values
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='lower right')

plt.tight_layout()
plt.show()

```

This example will output how the loss and accuracy improves across the training. Note, that this problem is linear and a neural network is a bit overkill. This simple example is to show all the parts needed to have a functional setup with plotting of the training process. We will build on this in further lectures.

1.5 Exercises

The exercises this week aim to introduce the basic ways of creating neural networks. We will take a look at three methods, with the latter one being more advanced and what you will need to master to have maximal customization control of your network. Relevant code for this week is found in the folder `week 1`. The network you need to recreate is the network we discussed in the previous section with one hidden layer with 8 nodes, 3 nodes in the final layer, and 4 input parameters. A working version of this network can be found in Python script `main.py`. This network you will have to recreate in different ways in this weeks exercises. If you can replicate the result from this example code using the method specified in the exercise you can consider the exercise complete. If you need help, consult the official documentation.

Exercise 1: Sequential class
In this exercise you need to use the sequential class as in this weeks example code. However, instead of using the `model.add` method you should create a list of layers and parse them directly to the sequential class. You should thus have a model definition like `model = Sequential([your_list])`. For this exercise the script `sequential.py` has been prepared and you need to implement code where the `TODO` comment is located.

Exercise 2: Functional interface
In this exercise you need to use the functional interface. Here you need to define your input as a layer and then the layers with neurons. Then you can define the model as `model = Model(inputs=input_layer, outputs=x)`. For this exercise the script `functional.py` has been prepared and you need to implement code where the `TODO` comment is located.

Exercise 3: Subclassing the model class
In this exercise you need to use Subclassing, where you create your network as a classing inheriting from the model class. Here you need to define your class as a subclass for the model class, e.g. `class Network(Model)`. Then you

need to declare your layers in the `__init__` function and the then build the network by defining a function called `call` which takes `self`, and the input tensor as variables. Remember to return the result. Then you can define the model as `model = Network()`. For this exercise the script `subclassing.py` has been prepared and you need to implement code where the `TODO` comment is located.

2 Machine learning and methods of training

In this lecture we look at machine learning in a broader perspective to get a feeling for the different methods which one can use. Then we focus in on neural networks and deep learning, before looking at methods of training neural networks.

2.1 Machine learning

Before we go into advanced concepts of neural networks, we better take a step back to understand the bigger picture. What is the difference between machine learning (ML), artificial intelligence (AI), neural networks (NNs), and deep learning (DL)?

As it turns out, neural networks is an outlier in this list. Neural networks is a particular method of machine learning, just like linear regression and decision trees.

As shown in Figure 4, AI is the overarching name for algorithms that aim to imitate some kind of intelligence. A very simple AI that does not use any kind of fancy ML methods would be something like the following code:

```
if temperature > 20:
  return 0
else:
  return 1

```

This example code is very simple, but it shows a simple AI algorithm which for example could be used to control temperature based on some temperature reading. One can make a more complex logic, but this is how one can make basic AI algorithms, and such algorithms have often been used in the past for various of applications to control sensors or bots in computer games.

Machine learning is a more advanced subset of AI methods where the algorithm can learn from data. Such algorithms do not require developers to hard code logic into the algorithms, as the algorithm will create its own logic after been trained on data. For complex methods such as neural networks it leads to the black box issue, that the developer does not understanding how the algorithm makes its decisions, but can merely try to assert if the decision is correct or not.

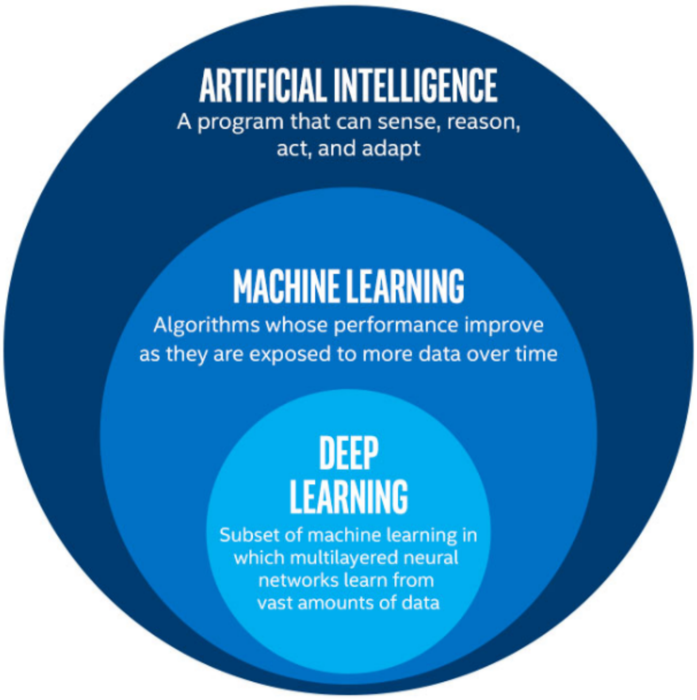
ARTIFICIAL INTELLIGENCE
A program that can sense, reason, act, and adapt
MACHINE LEARNING
Algorithms whose performance improve as they are exposed to more data over time
DEEP LEARNING
Subset of machine learning in which multilayered neural networks learn from vast amounts of data

Figure 4: *An overview of the connection between AI, ML, DL. Note that neural networks are not mentioned.*

To give you an idea of ML methods here is a list of some well-known methods:

- Artificial neural networks
- Decision trees
- Genetic algorithms
- Support-vector machines
- Regression analysis
- Bayesian networks
- Gaussian processes

These different methods have different areas of applications, and many of them a valid in the same application areas. The training requirements may also differ for each method, and many of these methods can often be designed to trained in different ways.

A short word on deep learning. Deep learning is when we talk of more complex ML, meaning not just to a simple densely connected neural network with a few layers, but rather models with thousands if not millions of parameters. Such models can often combine different ML methods and different training methods for the different components of the DL model. An example is a convolutional network that uses convolution and pooling layers to condense a lot of information into a densely connected network.

2.2 Training of ML methods

Since the point of ML methods is to learn from data, and not work by parameters set by humans, a key part to this field is the way you train your ML method. In this section we will look at three main ways of training ML methods.

2.2.1 Supervised learning

On of the most popular ways of training neural networks is supervised training. It is very simple to do, and with more data you can generally make an increasingly good ML model with supervised learning and a ML method like a neural network. In this section we consider neural networks. Some of the pros of this method is:

- For classification we can decide the number of categories
- You can mimic anything given enough labeled data
- A trained model is often very resource effective
- We can have a good idea of the accuracy of the model

One the other hand it can also have the following downside:

- Requires a lot of labeled data
- Poor performance outside of the scope of the training
- Cannot detect new categories
- Training is time consuming

Consider an arbitrary function $f(x)$ of input x. The result of this will some value y. We are thus considering a function that maps and input x to y,

$$f : x \to y. \tag{2.1}$$

Given enough data we can train a neural network to become any function $f(x)$ and thus have the neural network imitate whatever we want. The main requirement is that our neural network have enough degrees of freedom and that we have an appropriate amount of training data.

In the case of regression problems we can often use a neural network. In some cases we might be model something using a very complicated or compute intensive mathematical model, fx fluid dynamics. In this case it is possible to make an approximate model with a neural network. It can often be orders of magnitude faster to compute a bit of matrix multiplication than solving differential equations, and with enough data one can get very close with the approximation one can do with a neural network, keeping mind that the alternative model often also uses approximations to increase compute performance [3, 9, 8, 5].

Another case is classification and image recognition, and this is a huge field where supervised learning is showing potential. This has many practical applications. Consider making a self-driving car, one thing to do is to recognize the surroundings. There could be trees, people, other cars and many other obstacles.

Consider making to mathematical model of how a car or humans look. It's impossible to do with a simple description, because cars can be of different types such as SUV's, micro cars, hatchbacks, estates, vans etc. and the color could also be different together with other factors. Likewise, humans comes in different sizes ans shapes and with different skin color, clothes. There is no simple description for such objects.

Instead of trying to model this with some complex function, we can just use a neural network and train it with supervised learning if we have enough labeled training data. We don't have to know the underlying model, it's hidden inside the neural networks in the different weights and biases. As long as we have all types of cars, humans, etc. represented in the training data, it should be possible to get a pretty good classification from a neural network. This we can then use to figure out the environment around our self-driving car and take act. Should we brake because there is a pedestrian in front of us, or is that a car in front? Together with radar sensors to measure and attach velocities to the objects around us we can react to the environment. One can also use AI to recognize speed signs etc. to understand the speed limit or other signs.

Without ML methods it would be near impossible to make a self-driving car, but it seems like it can more or less be done with machine learning. But we also see problems to work on, such as handling things that the networks hasn't been trained to handle.

This is one of the limitations, there is no guarantee that the network will perform well outside the scope of the training. The logic could be very divergent outside of the training area causing very dangerous driving in the case of our self-driving car, and it cannot detect new categories and somehow evolve without retraining.

In Figure 5 it is illustrated how supervised learning aims a optimizing the boundary between the categories we have defined. Given we have activation function this boundary can be non-linear and generally it is non-linear, but

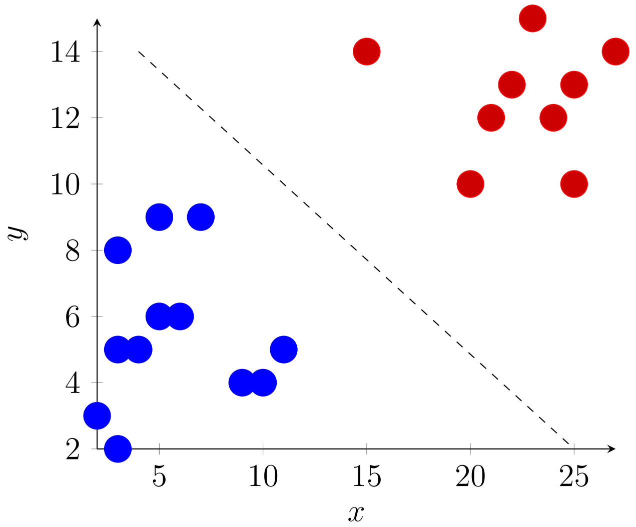

Figure 5: *Boundary division from supervised learning.*

is shown linear for simplicity.

2.2.2 Unsupervised learning

An alternative to supervised learning is unsupervised learning. There are many ML methods which can be used in this context including neural networks such as autoencoders. The main goal of unsupervised learning is to train an algorithms without having any labeled training data, in general you just have some result, y, but not the input x that led to the result.

Some of the cons of this method can be listed as:

- It can construct categories by itself
- New features and patterns can be discovered
- No need for labeled data

One the other hand it can also have the following downside:

- Interpretation is require to understand the detected patterns
- Accuracy is often worse than supervised learning
- No guarantee a useful result will be obtained

Since unsupervised learning doesn't require any knowledge input from the creator, it will try to find logic in any data given. It also means it is sensitive to anomalies and can thus be used to anomaly detection from data.

Consider some sinus curve for simplicity as seen in Figure 6. You might have some analogue signal of this shape, and you want to monitor if there is something unusual going on with signal. You can of course manually make

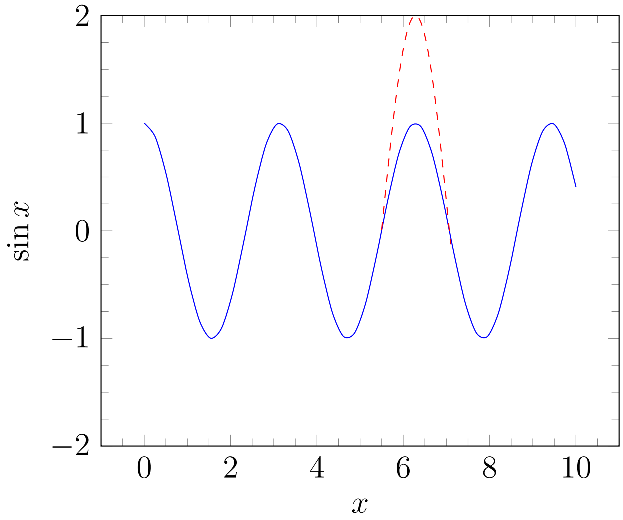

Figure 6: *Anomaly detection using unsupervised learning. The expected result is the blue line, but then there might be a peak like the dashed red line.*

checks if the signal has a certain amplitude, period etc. But in less simple cases like monitoring of internet packages or or complex flows, it might not be trivial with hand made definitions what is normal and what is an anomaly.

The point is we can use machine learning methods to learn and by itself define some expected patterns of a given input. Then if suddenly an input yields something unexpected the algorithm can flag it, and thus, detect the anomaly without any human definition of what an anomaly is. In this way we don't have any human bias in determining the parameters of the anomaly and we may detect patterns that a human could have missed.

Generally, we can again consider the case of two different populations like with supervised learning. In the former case we look at optimizing a boundary between the predefined categories. In the case of unsupervised learning the grouping is done based on finding common ground in the data and thus finding a cluster of related points. This clustering is illustrated in Figure 7, and one can also see some of the dots falling outside the clusters either suggesting some noise or some anomaly depending on the physical background of the data. If the errors are under control than we could argue that the red point outside the circle is an anomaly.

Another example of unsupervised learning that is a be different is that of autoencoder neural networks (and the similar variational autoencoder). This type of network have the same input as output, or some slightly perturbed input from the output.

The idea is that you take in an input and then reduce it to some latent vector representation of the input, that is the encoder part of the network. Then the decoder part of the network then recovers the input from the latent vector. You might ask why you would want to do this, and there are a few reasons.

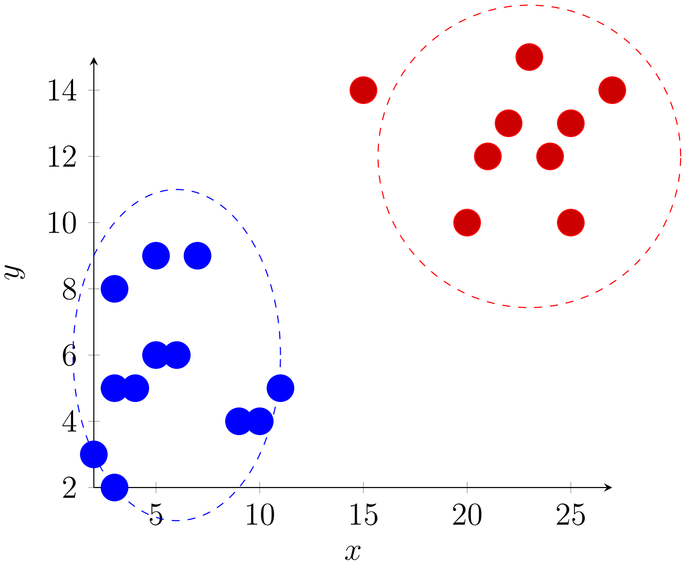

Figure 7: *Clustering of self learned groups from unsupervised learning.*

The latent vector in the middle is a compact representation of some input. You could think of simple example like that of irises. Let's say you have a stack of pictures of 4 different types of iris flowers. Now, the input is the picture and so is the output. But then in principle, you should be able to compress the input to a vector of 4 degrees of freedom for each type of irises. In the case you didn't know that, you might be able to obtain a minimal categorization of whatever input you consider, because the latent vector contains the minimal information to recover the input again.

Another example is that of denoising of images. You can easily take a lot of images, add a bit of Gaussian noise to the input and demand a clear and sharp output. In this case you are can train the autoencoder to denoise images or perform any other systematic transformation of an image.

2.2.3 Reinforcement learning

Somewhere between supervised and unsupervised learning we have the method called reinforcement learning. In this case we don not require a labeled dataset, but we are to define a reward function to maximize. It is incredible powerful and one of the methods also used in ChatGPT together with supervised learning.

This result in some of the following cons of this method:

- It can self correct errors
- it can learn from experience
- Very similar to human learning
- No need for labeled data

One the other hand it can also have the following downside:

- Not suitable for simple problems
- It is data and compute hungry
- It assumes a Markovian world
- Depends on the validity of the reward function

The fact reinforcement learning does not require labeled data is a huge advantage, but it does require a lot of data to train. This type of learning is similar to how humans think. We often have some values that we consider when we make decisions. How can I maximize profit? How can I make my life as good as possible. This is essentially the same as what we want to achieve with the reward function. Then the training measures different inputs and how such actions help achieving the goal encoded in the reward function.

As mentioned, reinforcement learning may face issue from the fact that the world is not Markovian which means that the next state depends only on the previous state, which is an oversimplification and in general we humans sometimes do things that make no sense, which again does not fit into this picture. Nevertheless, this approach is good enough, and using neural networks with memory one can future counter such issues.

A key concept of reinforcement learning is exploration. Let's say you want to make a robot that can walk, but you have no idea how to program that. You could define a reward function stating, the further you move, the bigger the reward. In principle this is simple and a valid loss function. There is however a big issue.

If you just put your stationary robot on the table and tell it to learn it will do nothing. Why? Well, it's not actually doing any changes that could lead to a better state. This leads to the importance of exploration. To make the robot learn we need to allow it to do something random to try out what could possible work, this is why we will modify the action to do with some perturbation ϵ called exploration. The probability of the robot doing what it believes is the best is thus $1 - \epsilon$. This, means that if the exploration is high, there is a high probability the robot will not do the action it considers to be best. This is good in the beginning, because the robot will not know what is good, but as it learns we will likely want to decrease the value of ϵ as we converge to the optimal action. With exploration on can make a robot with a neural network walk just from defining a reward function based on how far the robot walks.

Another key to remember, if you are training a neural network with reinforcement learning, then ensure that the reward has a dependency on weights, otherwise the weights will not be updated using gradient descent as the derivative of the update is zero.

2.3 Training of neural networks

Focusing on how to train neural networks it is necessary to update the trainable parameters of the network which is the weights and biases. Whatever training method you are using, there will be some input to the network. In the case of supervised learning there will also be some output from the training data. The result is that we compute some loss or reward function to determine how we should update the network. Note that if you have some reward function, you need to make sure you end up with a loss function that can be minimized if you want to train the network as described in this section.

One of the key concepts to train neural networks is the loss function. For this we consider the case of supervised learning. This is a function that measures the deviation of our prediction and thus how wrong the result from the network is. There are different functions one can use, but one simple example would be the mean squared error (MSE) loss,

$$\mathcal{L}(\hat{Y}, Y) = \frac{1}{n}\sum_{n=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{n}\left(\hat{Y} - Y\right)^2, \tag{2.2}$$

where we are summing over the n values of the output vector Y and predicted output vector $\hat{Y}$. Likewise one could also consider the mean absolute error (MAE),

$$\mathcal{L}(\hat{Y}, Y) = \frac{1}{n}\left|\hat{Y} - Y\right|, \tag{2.3}$$

where we simple consider the difference of the prediction and true value. You can define whatever value you want, but some loss functions might be more suitable than others. The MSE might not always be the best if the error is less than one, because the squared value yields a smaller loss than just taking the absolute loss, but often the MSE loss is quite effective.

This loss function is then a measure in our cost function, where we consider the total cost of our model. Note, that you can have more than just a single loss function as part of the cost function. We can write the cost function as a function of the underlying trainable weights $\Theta = \{w_{ij}, b_j\}$ which is composed of weights and biases,

$$J(\Theta) = \frac{1}{m}\sum_{m=1}^{k}\mathcal{L}(\hat{Y}, Y)_k, \tag{2.4}$$

where we sum over the batch size m. The cost function is then used for gradient descent updating of the weights and biases in the following ways,

$$w'_{ij} = w_{ij} - \alpha\frac{\partial J}{\partial w_{ij}}, \tag{2.5}$$

$$b'_j = b_j - \alpha\frac{\partial J}{\partial b_j}, \tag{2.6}$$

where α is the learning rate. Now I should note that this math describes the simple gradient descent optimization, but we often use a different optimizer such as the one called Adam [4]. This algorithm introduces concepts such as momentum, which helps to effectively adjust the learning rate along the way.

2.4 Exercises

In this week we will consider exercises related to training methods. We look at supervised and reinforcement learning and compare them. Here we make use of Python classes, though you do not have to worry about this. It is just to show more advanced use of Python, and for now you don't have to program more advanced customization, but your are encouraged to study the whole code. If you can understand all that is going on, you are doing well. The code for this week and be found in the folder `week 2`.

Exercise 1: Reinforcement learning
In this task we consider solving a few ordinary differential equations (ODEs). We consider the range $x \in$ [0,2]. In particular we want to solve the following equations,

$$\frac{\partial u}{\partial x} = 2x, \quad u(0) = 1, \tag{2.7}$$

$$\frac{\partial u}{\partial x} = x^2, \quad u(0) = 1, \tag{2.8}$$

$$\frac{\partial u}{\partial x} = x^2 - 2x, \quad u(0) = 1. \tag{2.9}$$

$$\tag{2.10}$$

The idea is that we can make a neural network $NN(x)$ be an approximate function of $u(x)$, thus $NN(x) \approx u(x)$. To train it we consider the definition of the derivative,

$$\frac{\partial NN}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}, \tag{2.11}$$

and note h as a infinitesimal exploration parameter. This allows us to write out loss function as an MSE loss,

$$\mathcal{L} = \left(\frac{\partial NN(x)}{\partial x} - \frac{\partial u}{\partial x} \right)^2, \tag{2.12}$$

now to incorporate the initial condition we can make the substitution,

$$g(t) = u(0) + xNN(x), \tag{2.13}$$

and obtain the loss function,

$$\mathcal{L} = \left(\frac{\partial g(x)}{\partial x} - \frac{\partial u}{\partial x} \right)^2 . \tag{2.14}$$

This loss function is already implemented together with a custom training method in the file `reinforcement.py`. What you need to do is to understand what is going on, tune the hyperparamters, and make the network where `TODO` comment is located such that it can we can obtain an accurate approximation. You can use the plot that is created to see if you are close.

Exercise 2: Supervised learning
Unfortunately, it is not possible to solves ODEs with supervised learning as it requires labeled y values to train. Fortunately, we know the analytical solutions. We consider the range $x \in [0,2]$ again. In this exercise you must create a network and compile is as `self.model` where `TODO` comment is located in the file `supervised.py`. You must train it and use the data created for you as the parameters x and y. You are free to build and train the model as you like. The goal is to make a supervised model that perfectly imitates the analytical solution.

Exercise 3: Out of your comfort zone?
As we discussed before, supervised learning in particular is not going to guarantee a good result outside of the training zone. In this task you need to compare the result of using your models from the reinforcement and supervised learning exercise and compare the result when trying to apply them to inputs of the range $x \in [0,5]$. Note that they should still be trained on the range $x \in [0,2]$. The code for this task can be found in `compare.py` and you need to create your data and plot it `TODO` comment is located.

Compare the two machine learned model and the analytical result for all three solutions in a nice plot. What does this tell you about the performance of supervised and reinforcement learning models outside the training scope?

References

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[3] Kai Fukami, Koji Fukagata, and Kunihiko Taira. Super-resolution reconstruction of turbulent flows with machine learning. *Journal of Fluid Mechanics*, 870:106–120, 2019.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[5] Julia Ling, Andrew Kurzawski, and Jeremy Templeton. Reynolds averaged turbulence modelling using deep neural networks with embedded invariance. *Journal of Fluid Mechanics*, 807:155–166, 2016.

[6] Jan Nagler, Anna Levina, and Marc Timme. Impact of single links in competitive percolation. *Nature Physics*, 7(3):265–270, January 2011.

[7] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[8] Thomas Vandal, Evan Kodra, Sangram Ganguly, Andrew Michaelis, Ramakrishna Nemani, and Auroop R Ganguly. Deepsd: Generating high resolution climate change projections through single image super-resolution, 2017.

[9] Rui Wang, Karthik Kashinath, Mustafa Mustafa, Adrian Albert, and Rose Yu. Towards physics-informed deep learning for turbulent flow prediction, 2019.

[10] Òscar Lorente, Ian Riera, and Aditya Rana. Image classification with classic and deep learning techniques, 2021.