Advanced machine learning

Mattias Ermakov Thing

May 16, 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` [6, 2], 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

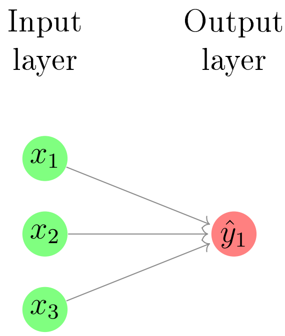


Figure 1: *A simple neural network with an input layer, and a single neuron in the output layer.*

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 [15], whereas image classification networks may contain thousands of neurons [26]. 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

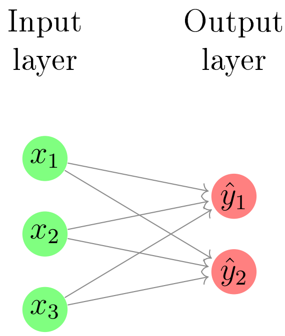


Figure 2: *A simple neural network with an input layer, and two neurons in the output layer.*

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

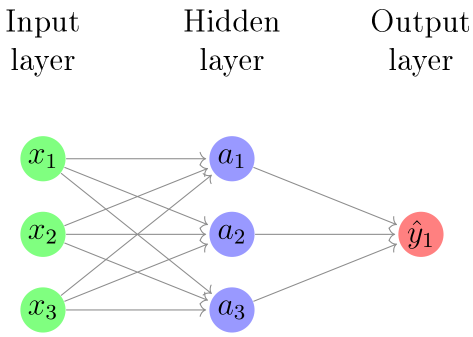


Figure 3: *A simple neural network with an input layer, a hidden layer, and a single neuron in the output layer.*

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,

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

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` [6, 2] to develop neural network. There are also other packages such as `PyTorch` [16] 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:

```
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.

```
# 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.

```
# 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.

```
# 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.

```
# 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.

```
# 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.

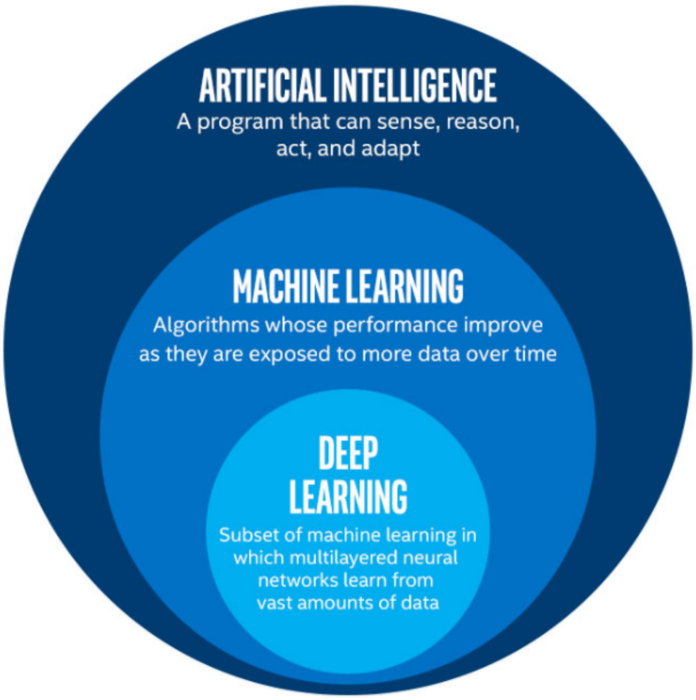


Figure 4: *An overview of the connection between AI, ML, DL. Note that neural networks are not mentioned.*

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 [9, 21, 20, 13].

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

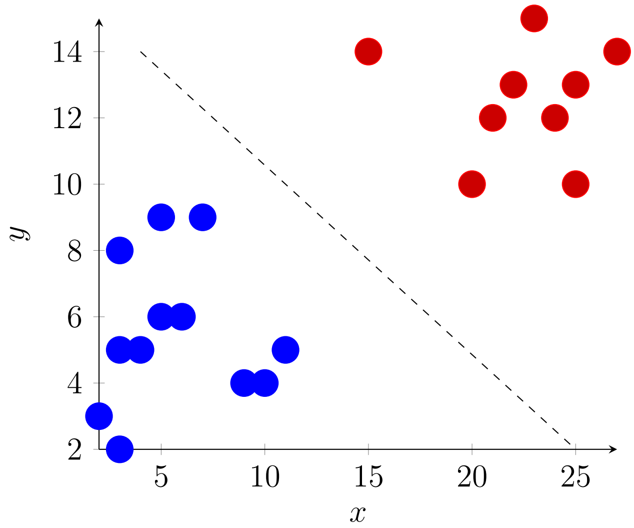


Figure 5: *Boundary division from supervised learning.*

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

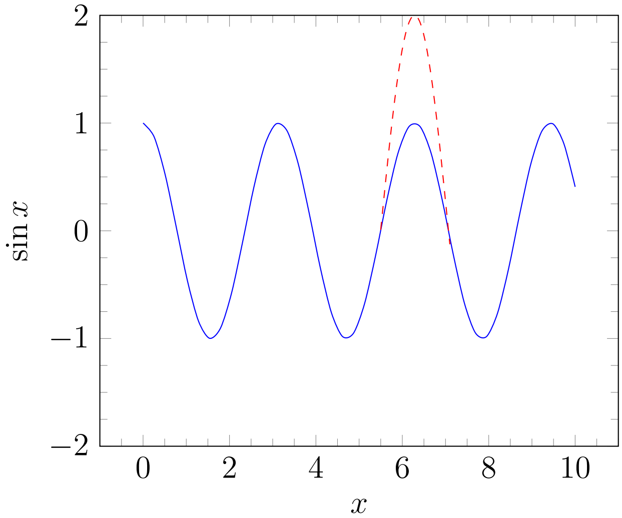


Figure 6: *Anomaly detection using unsupervised learning. The expected result is the blue line, but then there might be a peak like the dashed red line.*

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.

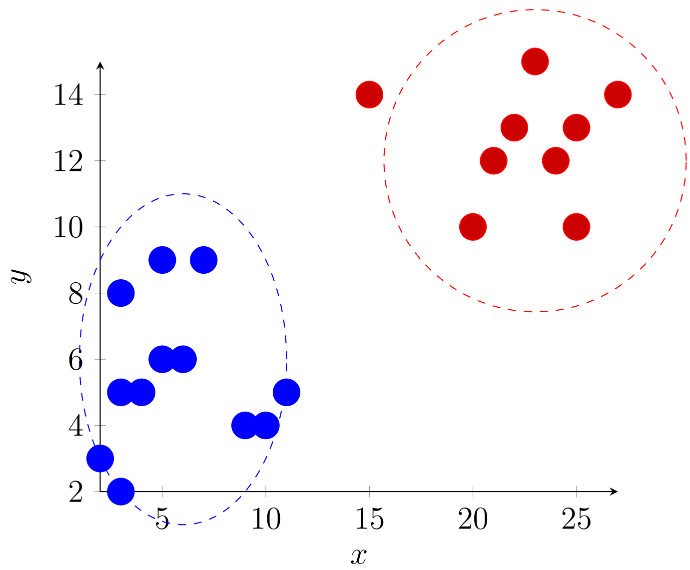


Figure 7: *Clustering of self learned groups from unsupervised learning.*

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 [11]. 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.

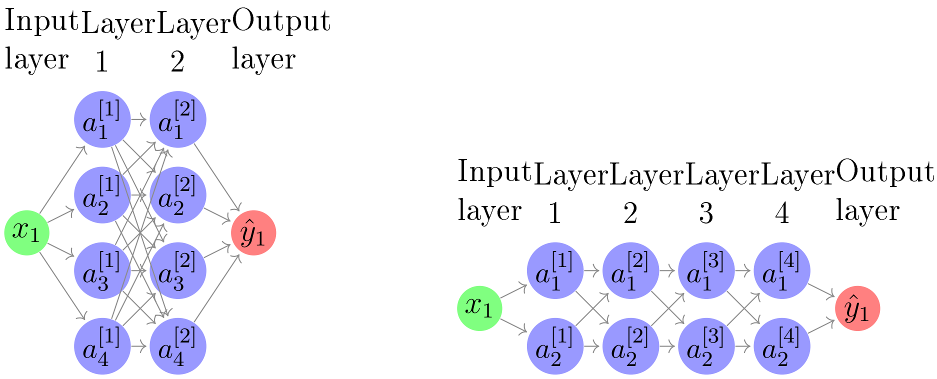
Exercise 3: Out of your comfort zone?
As we discussed before, supervised learning in particular is not going to guarantee a good result outside of the training zone. In this task you need to compare the result of using your models from the reinforcement and supervised learning exercise and compare the 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?

3 Neural network architectures

In this lecture we consider architectures of neural network and look at alternatives to densely connected neurons. In particular we take a look at convolutional neural networks (CNNs) and recurrent neural networks (RNNs) together with other methods of building networks with memory. The goal is to get a feel for the different types of layers and components we can use to build a network.

To this end, it is important that we carefully consider the problem at hand as this will help us understand how to go about designing a network. If we have a large input or output then maybe we should consider element of CNNs. Alternatively, if we consider applications requiring memory, often



(a) *A simple neural network with wide layers. This network 33 trainable parameters.*

(b) *A simple neural network with many narrow layers. This network 25 trainable parameters.*

Figure 8: *Two architectures with the same number of nodes, but with a different number of parameters.*

in cases with time evolution, then RNN or other methods that can store memory will be good to consider. Often it can be meaningful to combine the different types of layers and methods.

3.1 Artificial neural networks

Traditional artificial neural networks (ANNs) are very effective in many applications providing a function that can approximate effectively and cheaply [12].

When an ANN has three or more layers one can consider it as a deep neural network (DNN). This, arguably, is not enough to qualify the network as a DL method, but it is a step in that direction. DNNs can contain tens of layers if not even more in order to have enough parameters to model complex behavior. Having a very wide network with many neurons in each layer is one way to add degrees of freedom, but one may reduce the number of neurons per layer and have a longer network with many smaller layers. Since the layer parameters scale as the product of the number of nodes in the previous layer times the number in the current layer, one can quickly end up with a lot of parameters to train for wide networks. This fact is illustrated in Figure 8, where the wide network in Figure 8a as 33 parameters compared to the narrow network in Figure 8b with 25 parameters. This includes the weight parameters, each weight parameter corresponds to a connection, plus one bias parameter from each node. Both networks have the same number of neurons, but not the same number of parameters. This is something the architect of the network has to consider in terms of what effects the width has on the output and the trainability of the network.

This example demonstrates the concept that wide layers cost more to train, as you have more parameters to train. For many applications, a DNN with a sufficient number of layers can prove very successful, but the hidden layers are nothing but a black box, a common problem with neural networks. The larger the network, the harder it becomes to interpret the logic of the network, and in practice, it is generally not possible to understand the logic the network has learned.

Regarding the issue of training parameters, one can consider a problem in which the input is very large. In a DNN each input is densely connected to the first layer, thus for each neuron in the first layer, one can multiply the number of weights required to process the input. Furthermore, large inputs usually require a rather large first layer to retain the information from the input layer, thus the larger the input the larger the first layer should be leading to the problem of too many parameters. This is one of the limiting factors of DNNs, together with the fact that the neurons have no memory to store information about the previous value it processed.

3.2 Convolutional Neural Networks

A solution to the large input problem is the convolutional neural network (CNN). The problem is that dense connections can result in an exploding number of parameters, which become computationally unfeasible to train. Luckily, it isn't necessary to always use dense layers and a way to reduce the number of parameters is using CNN which uses convolution and pooling to reduce the problem, while retaining key information, before handing it over to a DNN network to produce the desired output. In the case of networks such as autoencoders, one might upscale the dimension of the output to match that of the input by doing upsampling instead of pooling.

In this section, the focus is the convolution and pooling used in CNNs. These networks are often used in image processing, thus, it makes sense to consider the case of a 2D input. Instead of connecting every input to every single neuron in the first layer, one can use a kernel. This means each neuron in the layer is only connected to a subspace of the full input. The size of the kernel is thus the size of the weights of each neuron, and one can still have a bias term. By having for example a three by three kernel, as shown in Figure 9, one would end up with only 9 weights per node plus the bias instead of e.g. 49 weights if the full input is a 7x7 two dimensional input. The reduction here is limited because the input as shown in Figure 9 is not very large due to practical reasons, but consider a modern 43.2 MP picture with dimensions of 6000x7200. In this case, the convolution neuron would still have 9 weights and not 43.2 million weights as the case would be for a fully connected neuron.

The values of the kernel acting on the input are then summed and processed with some activation function yielding the final value of the convolution neuron. Mathematically we can write this convolution G as a function

$$\underset{I}{\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}} \times \underset{K}{\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}} = \underset{I \times K}{\begin{pmatrix} 1 & 4 & 3 & \boxed{4} & 1 \\ 1 & 2 & 4 & 3 & 3 \\ 1 & 2 & 3 & 4 & 1 \\ 1 & 3 & 3 & 1 & 1 \\ 3 & 3 & 1 & 1 & 0 \end{pmatrix}}$$

Figure 9: *Example of convolution of input, I, with the kernel, K. The kernel is the trainable weights.*

of the input I and kernel K as,

$$G(I,K) = \sum_{j=1}^{3} \sum_{k=1}^{3} I(m-j, n-k)k(j,k)$$

$$= 1 \cdot 1 + 0 \cdot 0 + 0 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 = 4 \tag{3.1}$$

where j and k are summing over the indices of the kernel, in this case ae 3x3 matrices, and m and n are the row and column indices of the input according to the notation of Figure 9. Then one can apply an activation function to this result.

Note that the kernel requires input from surrounding values of the input, one can not simply apply the kernel to edge values. Depending on the size of the kernel one would thus naturally get a size reduction in the convolution layer. It is however possible to apply some padding like zero-padding to pad around the edge such that the kernel can be applied around the edge such that the convolution layer retains the same size as the input.

Another option in case one seeks to reduce the size of the convolution layer is to change the stride. The striding regulates the stepping which the kernel should move across the input. One can thus apply a stride of two to only compute a value for every second step. This will then reduce the size of the convolution layer by a factor of two.

The final important thing to consider with convolution layers is the number of filters. Since you can generally reduce the number of parameters by orders of magnitudes with convolution layers, one might face an issue with information being lost as the convolution might condense the problem with information loss. To compensate one can have multiple filters in convolution layers, in this case, one essentially repeat the convolution process as described in Figure 9. In this way, each point of the input is compiled multiple times with different weights that can be trained to capture different information about each point. For, example the filters might each capture some color value.

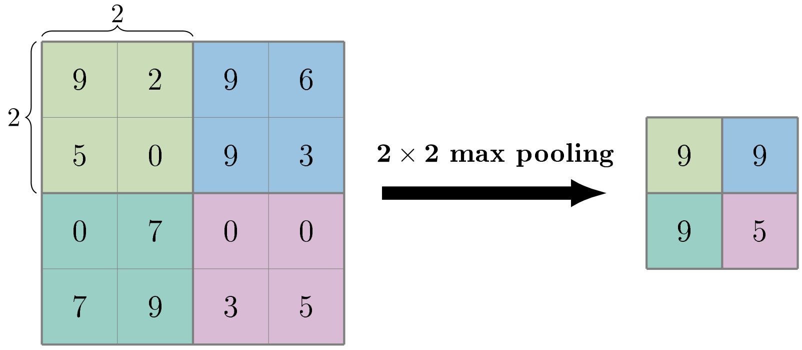


Figure 10: *Example of using max pooling with a pool size of two and a stride value of two.*

Overall, the convolution layer helps to massively reduce the number of parameters, but one should consider additional hyperparameters such as filter number, kernel size, padding and stride.

Another operation relevant to CNNs is that of pooling. In practice, this is a layer, but it doesn't add any new trainable parameters. It provides a systematical method to merge output values of the previous layer to decrease the number of parameters passed forward in the network as shown in Figure 10 with max pooling.

There are a few different ways to do the pooling such as min, max and average pooling. In the case of max pooling one reduces the pool size to a single value by selecting the maximum value in the given pool as shown in Figure 10. In this example the pool size is two, thus a 2×2 pool is considered and the highest value is selected. Since the stride is of value two, then one skips one ahead before creating another pool. Had the stride been of value one, then there would be overlap between the pools, but it might be desirable to choose such a value. As with the convolution layer one can also add padding to a pooling layer. The pooling layers generally have pool size, stride and padding as hyperparameters.

There is another ingredient you often need to consider when working with CNNs. Recall that you use convolution and pooling layer to handle a larger number of parameters and to condense them into an object with less parameters. In imagine processing you will often use these layer types to reduce the input until it is feasible to use dense layers. In case you have a multi dimensional object, you cannot just parse that to a dense layer as it expects a vector input. For this you can use a flatten function to map the multi dimension object layer to a vector which can then be feed to an ANN.

As a result a classical CNN will look like Figure 11, with layers of a convolutional layer followed by a pooling layer. Then the there is a flattening step before the input is feed into densely connected layers. In this case we can imagine a network with a softmax function in the output layer to classify the input into a fixed amount of categories. The combination of many types

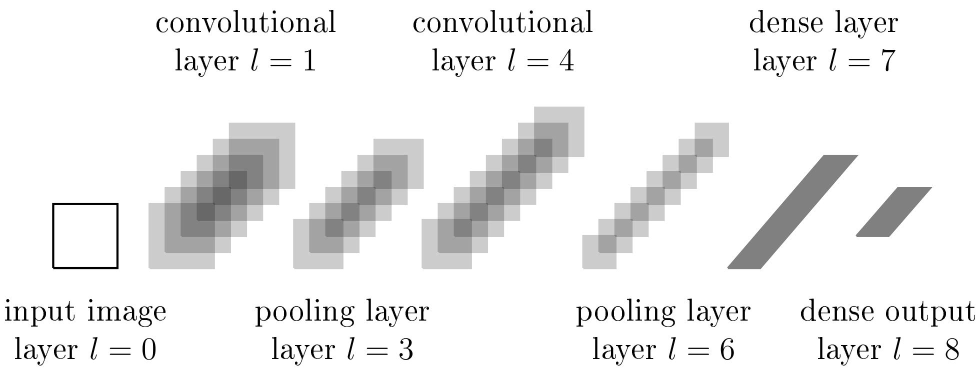


Figure 11: *The architecture of the original convolutional neural network, as introduced by LeCun et al. (1989), alternates between convolutional layers and pooling layers. The feature maps of the final polling layer are then fed into the actual classifier consisting of an arbitrary number of fully connected layers. The output layer usually uses softmax activation functions.*

of layers suggests that we are now entering the realm of DP methods.

To understand a CNN a bit better, let us discuss the effect of the convolution layer. For each submatrix of the full input we are scanning for features. If we think about animals or humans, this could be features like eyes. Now there could be multiple features in one submatrix, and therefore you usually have multiple feature maps per submatrix. In other words the procedure described in Equation 3.2 is done multiple times and the associated with different weights, one for each feature map. In the case of RGB pictures it is also common practice to have a layer for each color, thus in this case three. So lets say you have and RGB picture and you want to extract 4 features, you would then need 12 feature maps or filters as they are also called.

As you are generally reducing the input seize with convolution and poolings layers, you are naturally increasing the scale of the feature detection of the convolution layer, because you kernel is the same size and your layer input is containing information about a larger area. That creates a hierarchy of how the learning is done. In the first layers the network captures small features and then the subsequent layers capture larger features.

Regarding compute, there one should consider using GPU resources when using of networks with CNN features. The computations related to convolution and pooling layers are very simple, but involve a lot of computations. This allows for effective parallelization with modern GPUs, as they have an abundance of cores at the order of 10^3 compared to CPUs with core numbers typically at the order of 10^1. In particular modern GPUs tensor cores are highly efficient at these types of workloads.

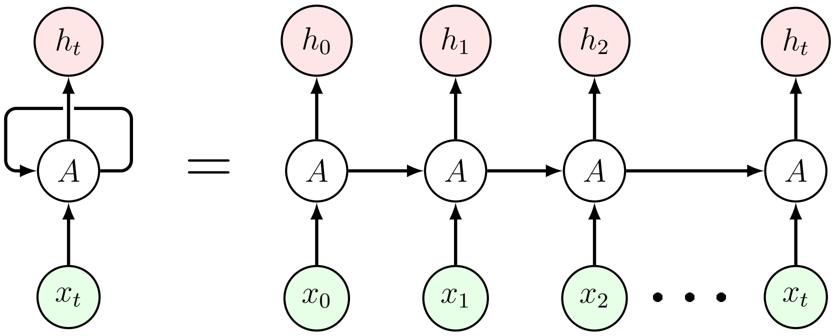


Figure 12: *Illustration of recurrent neurons using the previous output as input for the current node computation.*

3.3 Recurrent neural networks

In certain applications such as speech recognition, and other cases with time dependence, the previous input value may be correlated with the subsequent value. This is something a usual DNN or CNN cannot take into account at the neuron level. This requires the neuron to have some memory. To achieve this one can make use of recurrent neurons to make recurrent neural networks (RNN) [18].

The simplest way to incorporate memory is to modify the neuron to store the previous output and include it in the current input. In this way, the historical output is used for the current evaluation in the neuron. This concept of recurrent neurons is illustrated in Figure 12. Mathematically the math in the neuron changes in the following way,

$$a(z_j(x_i)) = f(w_{ij,1}x_i + w_{ij,2}h_{i-1} + b_j \cdot \vec{1}), \tag{3.2}$$

where h_{i-1} is the hidden state associated with the previous output of the neuron. The hidden state generally captures an encoding of the most recent neuron computation. This hidden state is a new object in recurrent neurons and it allows for short term memory in the neurons. Since both x_1 and h_i are inputs to the function, they both get a weight matrix associated to them.

One of the limitations of the simple recurrent neuron is that it only takes into account the previous output. This means we are limited to short-term memory. Using more complex logic it is possible to achieve long-term memory, and for this one can use long short-term memory (LSTM) layers. These layers a significantly more complex and compute-intensive, but the long term cells in these layers are capable of storing information long-term. Such models are useful in the case where there are long-term memory is needed to obtain the correct prediction.

A LSTM layer is a layer of LSTM cells, with each cell containing the more complex logic as shown in Figure 13. To understand this cell we need to consider the different inputs it takes and the outputs it yields:

- x_i: The current input to the cell

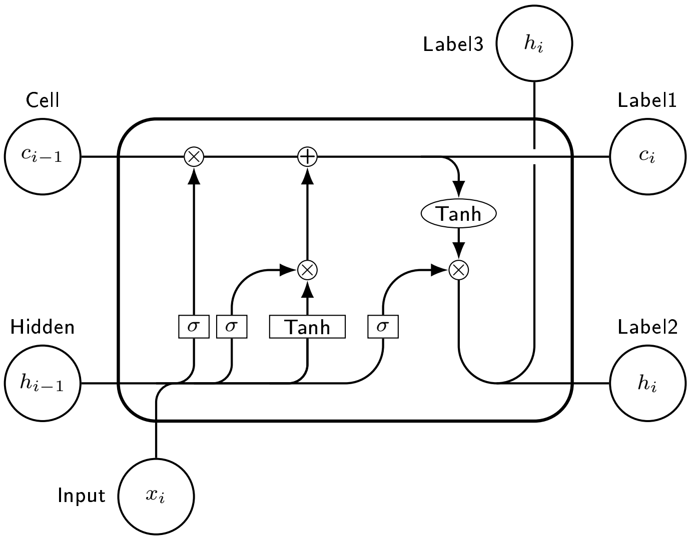


Figure 13: *A diagram of the LSTM cell with an input, a hidden state, and a cell state with internal logic to update the states.*

- h_{i-1}: The hidden state input to the cell
- c_{i-1}: The cell state input to the cell
- h_i: The hidden state output of the cell
- c_i: The cell state output of the cell

Compared to the simple recurrent neuron, the LSTM cell contains a new object which is the cell state. This is the object that holds long term memory that forms across many inputs unlike the hidden state which contains information related to the previous input. It is the updating and handling of the cell state that makes the LSTM cell much more complex due to the logic behind updating and using it.

There are three main logical gates computed in the LSTM cell. The first to consider it he forget gate f_{forget}. This gate considers the input x_i and the hidden state h_{i-1}. Here the computation is as follows,

$$f_{\text{forget}}(x_i, h_{i-1}) = \sigma\left(w_{ij,1}x_i + w_{ij,2}h_{i-1} + b_j \cdot \vec{1}\right), \tag{3.3}$$

which is then multiplied to the cell state c_{i-1} to get a scaled cell state c'_{i-1},

$$c'_{i-1} = f_{\text{forget}}(x_i, h_{i-1})c_{i-1}. \tag{3.4}$$

Since $0 \leq f_{\text{forget}} \leq 1$ the result of the forget gate is to decide how much of the past should be forgotten, with $f_{\text{forget}} \sim 1$ yielding a minimal change of the cell state $c'_{i-1} \sim c_{i-1}$. On the other hand $f_{\text{forget}} \ll 1$ yields $c'_{i-1} \sim 0$, thus wiping the long term memory clean.

The next gate is the input gate f_{input} which computes a quantity like f_{forget} and multiplies it with a similar function where the tanh function is used,

$$f_{\text{input}} = \sigma\left(w_{ij,3}x_i + w_{ij,4}h_{i-1} + b_j \cdot \vec{1}\right) \cdot \tanh\left(w_{ij,5}x_i + w_{ij,6}h_{i-1} + b_j \cdot \vec{1}\right), \tag{3.5}$$

where one can either do a point or piece wise multiplication of the two objects. The part with the sigmoid function can be considered as a scaling term like in the forget gate, and the part with the tanh function is the normalized encoding of the input plus the hidden state. The cell state is then updated,

$$c_i = c'_{i-1} + f_{\text{input}}, \tag{3.6}$$

and that cell state is then used in the next input step and to compute h_i in the last gate, the output gate f_{output}. This gate has the same components as the previous gate,

$$f_{\text{output}} = \sigma\left(w_{ij,7}x_i + w_{ij,8}h_{i-1} + b_j \cdot \vec{1}\right) \cdot \tanh\left(w_{ij,9}c_i + b_j \cdot \vec{1}\right) = h_i, \tag{3.7}$$

and the result is the new hidden state which is also the output of the LSTM cell. Note that the output gate contains both the input, hidden state and the updated cell state.

In the LSTM cell we have 9 weight matrices and a lot of biases to train. Likewise we have a lot of logic to process. This means that these cells are computationally much more intense compared to a simple neuron. Due to the extra complexity, it is often favorable to use CPU cores to deal with LSTM networks as their more powerful cores outclass the parallelization which CPUs offer.

3.4 Reservoir computing

If you have no clue what a suitable architecture is and you need memory, you might want to try out reservoir computing. This is an actively evolving field where we take even more inspiration from biology, in particular the human brain.

The concept is as follows, you take a bunch of neurons, which could be recurrent neurons with memory, and then connect them sparse. The result is hundreds or thousands of neurons connected randomly to each other, not densely and not in a particular way. This is similar to how neurons in our brains are connected. Human neurons only connect to a handful of neurons in the vicinity. The same is idea of reservoir computing, this construction of randomly connected neurons is our reservoir. On its own, this cannot achieve anything, you just have a N neurons.

Next consider some input signal $u(t) \in \mathbb{R}^{N_u}$ which we connect to N_u neurons in the reservoir with weights W^{in}. The reservoir's reservoir's high-dimensional are given as $X(t) \in \mathbb{R}^N$ and randomly initialized weight W, and in this reservoir there are states accessible for read out $x(t) \in \mathbb{R}^{N_x}$ where $N_u + N_x \ll N$ with wights W^{out}.

The prediction $\hat{y}$ can then be defined as,

$$\hat{y} = g\left(W^{\text{out}} x(t)\right), \tag{3.8}$$

$$x(t) = f\left(W x(t) + W^{\text{in}} u(t)\right), \tag{3.9}$$

with f and g being two activation functions.

It is possible to then train the network in a supervised way, or in other ways. The key to this method is that one does not train the reservoir, thus W is fixed, but on can train W^{in} and W^{out} for inputs u and true outputs y. We are thus optimizing how we write and read to the reservoir. It is also possible to have ANN's connected before and after the reservoir, and that ANN may be fully trainable too. The main idea is simply that we do not care about optimizing the reservoir itself, but rather our interaction with it. It is possible to try to train the reservoir itself, with on of the more straight forward methods would be using genetic learning.

3.5 Exercises

In this week the exercises allow you get practical experience with different types of networks using examples from `tensorflow`. We look doing image classification of using ANNs and CNNs to compare the practicality of the different methods. The first two exercises use the CIFAR10 dataset containing 60,000 color images in 10 classes, with 6,000 images in each class. The dataset is divided into 50,000 training images and 10,000 testing images. The classes are mutually exclusive and there is no overlap between them. Then we consider text classification with memory. Be careful with over-fitting! These example should give you a practical feel for the practicality of different methods in different situations. The code for this week and be found in the folder `week 3`.

Exercise 1: ANN image classification
In this exercise you need to construct a ANN that can correctly classify the image specified in the script `ann.py`. Note you need to flatten the input first before you can use dense layers and build your network where the `TODO` comment is. How many parameters to you need to obtain an accuracy of 30%, 40% or 50%?

Exercise 2: CNN image classification
In this exercise you need to construct a CNN that can correctly classify the image specified in the script `cnn.py` where the `TODO` comment is. How many parameters to you need to obtain an accuracy of 60%, 70% or 80%? Compare the results to the ANN network. How is the performance and how do the

parameter number differ?

Exercise 3: Text classification
In this exercise you need to classify text. This example used IMDB data and the goal is to determine if a text represent a negative or positive review. This is called sentiment analysis. Because we are working with text, we again need to consider encoding and also tokenization. The latter concept is beyond this lesson and you may study it if you are interested. The code for this exercise is in `text.py` and you need to construct a network with memory where the `TODO` comment is. To get started you are presented with the initial encoding and embedding layer and then it is up to you to implement layers such as bidirectional, recurrent neuron, and LSTM layers. To make RNN layers you can use subclassing, check the `keras` or `tensorflow` documentation on how to do it. Note LSTM layers are very compute heavy and you generally don't need a lot of them. The code may produce errors that have no functional meaning, they relate to the import method of the dataset. Can yo reach 90% accuracy?

4 Physics guidance in neural networks

In this lecture we look at ways of mitigating problems related to neural networks. In particular, we look at how physics can help us and how we can solve physics problems with neural networks. To do this we need to look closer at the cost function, simulation and architectures that leverage this physical constraints of the problem.

Before moving on with solutions, let us remind ourselves with some of the problems we have encountered so far:

- Lack of training data
- Over-fitting
- Failure to learn the underlying function
- Slow learning
- Poor results

Before looking and solutions from physics, let us consider a few general methods to consider.

4.1 Non-physics optimizations

When trying to tackle any problem wit a neural network, we should always start by considering what architecture is optimal. Should you use an ANN, CNN, RNN, or some other network architecture?

If your input is tabular data, like what you find in a spreadsheet and the problem at hand is regression or classification, then you might be fine with an ANN. This includes text date, image data (though generally not images themselves), time series data, and other types of data where the input size is not more than $< \mathcal{O}(10^3)$.

In the case you want to handle images and videos and consider regression or classification problems, then you would use CNN elements. You can also use a CNN for text data, time series data and sequence input data. This method is advantageous in the sense that it can handle large input sizes.

If you face mapping problems such as one-to-many, many-to-one or many-to-many, then RNN or LSTM methods can be useful. In particular this is useful for dataset of text data, speech data, classification and regression problems, and generative models. Do not use it for tabular or image data.

Reservoir computing can also be used generally on classification and regression problems, and it has the advantage of being more effective in terms of training as you do not train the reservoir itself. This result in lower costs and more efficient computing.

4.1.1 The cost function

Before moving on let us take another look at the cost function. Recall, the cost function is object that is used for training and contains at least one loss function.

As discussed earlier, the choice of a loss function can have an impact on training time. For example the MSE is generally good at penalizing large deviations above one, but for small deviations the MAE is more effective. Other functions also exist with the own pros and cons, and it is always important to select an appropriate loss function to speed up the training. Note that the choice also depend on whether you are doing regression or classification.

In principle you can have multiple loss functions, though it rarely improves the situation to just simply chain two loss function together as one of them will generally just dominate. Also, there can be stability issues if the two loss functions you optimize push the network in different directions.

What is often more meaningful is to have different loss function acting on different outputs, or you can have loss functions that checks intermediate results on top of the final results. We will talk more about that in the context of applying physics to our training methods.

The last thing to mention is regularization. This is a method of penalizing the general use of weights and trainable parameters. It is used to combat over-fitting as regularization pushes the network to set trainable parameters to zero, effectively removing them from the equation. With less parameters to train, the complexity the model can take is reduced.

There are two very common methods of regularization which one can use.

One is lasso regression also called L1 regularization,

$$L_1 = \lambda \sum_{j=1}^{n} |\Theta_j|, \tag{4.1}$$

where we sum over the absolute value of the trainable parameters Θ_j of the model and then we have a scale factor λ. Note that this generally mean that you will have a non-zero cost function even for a perfect training result as you will have non-zero trainable parameters.

Another common regularization method is ridge regression also called L2 regularization. In this method the math writes,

$$L_2 = \lambda \sum_{j=1}^{n} \Theta_j^2, \tag{4.2}$$

where we are now squaring the trainable parameters Θ_j, and there is still the scale factor λ. One thing to note here is that the L1 method is better for pushing weights to zero, as the L2 method doesn't penalize values below one very hard. On the other hand the L2 method penalizes large parameter values harder.

4.2 Physics optimizations

Physics is based on mathematics, which is full of possible combinations of functions and possibilities. We have however seen, that nature of tends to fit well with reasonably simple mathematical equations. As an example we can consider the Lagrangian which is just an equation of energy,

$$\mathcal{L} = \frac{1}{2} m \dot{x}^2 - \frac{1}{2} k x^2. \tag{4.3}$$

You might recognize this as equation of motion of a spring. The mathematics here is quite limited, just two terms with some derivative of the coordinate, constants and some power laws. Considering what mathematics exists, it could have been much worse.

The point is, the real world has a lot of physics constraints, and this can often make the situation much simpler. Having this domain knowledge from understanding the underlying physics can often help us. Add to that that we might often know good models for issues we want to apply machine learning to, and we can leverage that to speed up training and make the resulting models more precise.

4.2.1 Physical cost functions

One of the simpler ways we can apply this domain knowledge is via the cost function. Instead of just using standard loss functions like MSE or techniques like L2 regularization, we can implement terms that specifically leverage physical knowledge of the system such as conservation of energy. If you can penalize your prediction by violation of conservation, you have way of excluding nonphysical behavior and thus get a more realistic and robust model.

In general, we can consider some physically inspired cost function, in some general form,

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}_{loss}\left(\hat{Y}_i, Y_i\right) + \lambda R(\theta) + \lambda_{phy}\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}_{phy}(\hat{Y}_i), \tag{4.4}$$

where m is the batch number, $\mathcal{L}_{loss}$ is our loss function of choice, λ and λ_{phy} are hyperparameters for the two physics terms, $R(\theta)$ is a function of the model parameters evaluating parameter constraints, such as complexity, and $\mathcal{L}_{phy}$ is a physical reality check of the predicted value [7].

In Equation 4.4 there is the usual loss term, but also two additional terms where one can generally incorporate physical checks on the model parameters and the predicted value. The function $R(\Theta)$ of the model parameters is regulated in strength by λ. This may be a function of certain weights which shouldn't be positive or negative or there sum should be some certain value, or simply we might just want to penalize weight parameter similarly to a usual regularization term. We can consider this a physics inspired regularization term, which we set because we have some domain knowledge of what the parameters should yield.

In the last term, we can consider constraints on the result, maybe even taking into account the inputs, X, to compute some conservation of energy constraint on the result. Depending on the problem at hand, then the input may put certain limits on what the output is allowed to be, and this can be used to enhance the learning [19]. In many cases there could be more physics-inspired loss terms than in the example of Equation 4.4 [9].

A concrete example of using a physics-inspired loss function is in the case of modeling lake water temperatures at different depths. In this case, one can consider a neural network with LSTM layers, as we need memory to incorporate the effect of the previous step regarding the depth [10, 17]. For example, the water pressure only increases at lower depths, and depending on the season, the water temperature may decrease at lower depths in warmer months.

As we are interested in modeling temperature over time at different depths, one can consider a physics inspired MSE style loss function of the form,

$$\mathcal{L}_{loss}(\hat{Y}, Y) = \frac{1}{N_d T}\sum_{t=0}^{T}\sum_{d=0}^{N_d}\left(\hat{Y}_{t,d} - Y_{t,d}\right), \tag{4.5}$$

where N_d is the number of different depths, and T is the number of temperature steps. In this case, we have a physical system with electromagnetic radiation adding heat to the lake and some heat will be lost to the surroundings. In general, there are different physical aspects to take care of, but they can be incorporated in the total flux, ΔU_t, and the sum of heat flux, $\mathcal{F}$, to track the heat [10].

In this case we use domain knowledge relating to radiation and heat dissipation, and make a physical model of what the total flux of energy is to the lake. Thus, we consider how much energy comes in and how much goes out of the lake to have a physical value of what ΔU_t should be. Then, the sum of heat flux, $\mathcal{F}$ is monitoring the predicted heat flux from our neural network. With this we can track that the ML model is not braking energy conservation by demanding that $|\Delta U_t - \mathcal{F}| \approx 0$. Thus whatever energies that the lake has in the different depths etc. it has to be related to the energy coming in and going out based on our physically established model.

We are leveraging the fact that we can reliable model what energy goes in and out of our system, the lake, but we do not know how the energy is stored inside the lake. What is the lake temperature at different depths in different places?

In this case, we can define a loss function for this energy condition encoded in a constraint that looks something like,

$$\mathcal{L}_{EC}(\Delta U_t, \mathcal{F}) \sim \mathrm{ReLu}\left(|\Delta \mathrm{U_t} - \mathcal{F}| - \tau_{\mathrm{EC}}\right), \tag{4.6}$$

where τ_{EC} is the threshold value to allow for uncertainties in our model. Since we consider the absolute value, we consider any deviation from the physically expected total flux. The physical model will have uncertainty so we can take care of that by subtracting the uncertainty. If the value in the ReLu function is below zero we just just zero, and thus a zero loss. In this way we get penalties when the deviation is beyond the threshold only

Since the water pressure, $\Delta\rho$, in a physical system only increases with depth, one can also create a loss term that enforces this constraint. We can simply consider the difference,

$$\Delta\rho_{d,t} = \rho_{d,t} - \rho_{d+1,t}. \tag{4.7}$$

This results in a cost function of the form,

$$J(\theta) = \mathcal{L}_{loss}(\hat{Y}, Y) + \lambda_{EC}\mathcal{L}_{EC}(\Delta U_t, \mathcal{F}) + \lambda_{DC}\mathcal{L}_{DC}(\Delta\rho), \tag{4.8}$$

where the first term is the loss terms in terms of depth and temperature, and the second term contains the energy condition encoded in the loss $\mathcal{L}_{EC}$, modulated by the hyperparameter λ_{EC}, and $\mathcal{L}_{DC}$ incorporates the depth constraint, modulated by λ_{DC}.

There are various of different other use cases relating to human needs such as electricity and fresh water. Modeling of these resources is key to efficient use and exploitation of them. For this one can also use ML methods with learning functions that leverage domain knowledge [22, 23, 8]. In the latter case one uses statistical learning instead o af neural network highlighting the universality of physics-guided ML methods.

The physics guiding of ML methods is not limited to supervised learning, and it can also be applied to other learning methods such as reinforcement learning. Considering the problem of optimal traffic light control, one is faced with the problem of multiple states and the best way of switching between them. In this case, real-world knowledge can help with solving the problem, and a domain-informed training procedure can yield solid results [22].

4.2.2 Hybrid architecture

Another approach we can do beyond just making physics guided cost functions is to consider it in the architecture of the network or model. Let's say you are considering a physical system which be described with some model ρ. Unfortunately, this model does not take into account all aspects of the system. In this case one can consider a method like Knowledge-based Residual Learning [25]. This idea is that the ideal model, F, can be composed of the incomplete physical model and a residual neural network G,

$$F = \rho + G(\Theta), \tag{4.9}$$

with some trainable parameter, Θ, that needs to be trained. It has been shown in many cases, that complimenting the neural network with domain knowledge yields a more robust result and usually also a faster training procedure. This makes sense, because we have done some of the training but introducing knowledge to the system.

We, thus, have some driver parameters, x_i, that we give our model ρ and the neural network G. Then we concatenate the result together to yield the prediction $\hat{y}$. And then we train the parameters Θ as usual, for example with supervised learning.

If we just use a black-box network, then we will just have a function that will mimic the residue. But in principle one can pair this with a symbolic regression ML method and try to even recover a symbolic equation. In this way, we can use ML methods to learn corrections to our physical model.

4.2.3 Physics architecture

One can include domain knowledge in the architecture of the network itself. This often requires detailed knowledge of how the problem should be computed, and an understanding of different aspects of the physics problem at hand. It is in many cases possible to make a network architecture that internally applies physical constraints and domain knowledge. In this way, the network is internally constrained to obey physical concepts.

A case study of such an application can be seen considering drag prediction [14]. As this problem deals with fluid dynamics, it is known that we need to be working with quantities such as pressure and velocity. By design, one can create a sublayer with the degrees of freedom corresponding to what the pressure and velocity vector should have based on the underlying physics. These pressure and velocity layers are then used to compute the intermediate result of the pressure force, F_i^P, and shear force, F_i^V. From the dimensionality, of the problem it is known that the output should be three dimensional. The final result is the addition of these two quantities, thus, the final drag is known to be of the general form,

$$F_i = F_i^P + F_i^V, \tag{4.10}$$

leading to a network architecture where the two penultimate layers should be added to create the output layer. One may also consider rotational and Galilean invariance to improve the results, as such physical invariances enhance the choice of physically meaningful outputs [13].

With a physics guided architecture we can implement constraints directly into the network. We can also go back to our example of the spring and we want to predict the next position. We can consider the case where we know the velocity, $\dot{x}$ and position, x but not the spring constant, k, and mass, m. We know the solution to the problem is,

$$m\ddot{x} = -kx \tag{4.11}$$

Architecturally we will make two subnets; one that is the kinetic energy, and a function of the velocity, and one that is the potential energy, and a function of the position. In this case we could make a network to model these value as a trainable parameter Θ_k and Θ_m. But since we know the law of conservation must hold we can add a loss term like,

$$\mathcal{L}_{\text{Conservation}} = \left|\frac{1}{2}\Theta_m v^2 - \frac{1}{2}\Theta_k x^2\right|, \tag{4.12}$$

as optimizing this equation would also ensure conservation of energy.

4.2.4 Simulation

The growth in data points has been a major driver for enabling the success of neural networks. There are, however, many cases where data is lacking, at least there might not be a complete data set with both X and Y values, inputs and outputs. To bridge the gap when data points are few, one can again make use of physics models.

One approach that can work, if you have a strong physical model is two make synthetic data, but if you don't have that you can use a Generative Adversarial Network (GAN), which can contain physics guidance. The idea is that you have true images which your generator network tries to imitate, this network can itself be physics-guided. The evaluation is done by a discriminator network, which tries to identify the synthetic data from the real data,

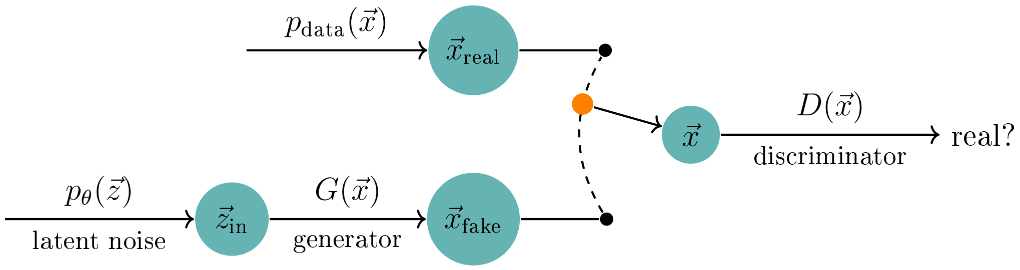


Figure 14: *A diagram of a GAN network. This is a method of making simulated data with a neural network.*

In Figure 14 we see the general layout. We have some true data, p_{data} yielding the true output $\vec{x}_{\text{real}}$. Then we have some latent noise, p_Θ, generated in some controlled way. Then we can make a generator network, G, that takes the latent noise as input and creates a simulated output $\vec{x}_{\text{fake}}$. Then a discriminator network, D, attempts to determine which is the real output.

In the ideal case, you would end up with the discriminator being incapable of telling the synthetic data and the real data apart. In this way, the fully trained generator can then be used to generate synthetic data, which one can then use for training. This can for example be used to train robots [4], here a simulated picture based on a physical model is the input to the generator that makes the simulation look more real. One can then do pre-training with synthetic data one needs less real data to achieve good results.

Pre-training is a step where one generally trains the model on some other data first, before training on the main data. Sometimes it might make sense to freeze some trainable parameters after pre-training. The goal of this process is to teach the network some basics before using the limited real data to learn the details.

As discussed previously, the inverse problem is a common problem with applications in different fields, particularly in imaging. For medical purposes, we often seek to make certain images, but phase issues with reconstruction, image quality, and noise [4].

Typically, you would use CNN networks to handle the large imagine inputs, but for certain applications, such as super-resolution, GAN architectures may yield further improvements [4]. A common and practical use for such GAN models is in CT-scan imaging [24]. It is also possible to use neural networks to reduce noise and improve resolution, which leads the way for making low-dose CT scans [5].

On the other hand, for reconstruction the problem is often that traditional algorithms are too slow. As a neural network can generally be trained to mimic any function, the point is to speed up the reconstruction by up to orders of magnitude by using neural networks. The problem is training the networks as you often have a limited data set.

In some applications, it is however easy to get physical output data, Y, but then the input data might be missing. In this case, physics knowledge can be used to convert an output data point to an input data point using the equation,

$$f = Hg, \tag{4.13}$$

where f is the input, H is the system matrix and g is the output. In this case, it is possible to generate simulated input data from real output data, Y, to complete the dataset by using domain knowledge and then train or pre-train the model with the help of partially synthetic data [3].

Finally, it is important to mention that one approach of implementing physics guidance doesn't exclude another. Many of the discussed models use more than one method, for example, physics-guided loss functions and architecture. It is in principle also possible to have use cases where all three approaches can be useful. One case study of that is that of the multi-fidelity physics-constrained neural networks (MF-PCNN), which is used for different applications such as heat transfer, phase transitions, and Dendrite Growth [1].

4.3 Exercises

In this week the exercises are about improving the learning and how you can avoid over-fitting. In the first exercise we take a look at regularizers to reduce over-fitting. Then you have to use physics knowledge to achive a similar result with a custom loss function. Note the results will not be significantly better, but they will be a little better if you implement the solution correctly, most notably you will not see over-fitting. The code for this week and be found in the folder `week 4`.

Exercise 1: Regularization

In this exercise you are given a network which is suffering from over-fitting, the code can be found in the script `regularizer.py`.

To help you chose the regularization method correctly here is some background on the data. The input is an array of numbers from 0 to 2. Using supervised training, the network must learn to produce the norm of the following wavefunction,

$$\Psi = \sqrt{\frac{2}{L}} \sin\left(\frac{\pi n x}{L}\right), \tag{4.14}$$

which is the solution to the infinite well potential where L is the width of the well potential and n is the energy level. Consider the what the values of the trainable parameters will be in this case where we have a normalized wavefunction.

In this task you need to add an appropriate regularization method to improve the result and training. You should **NOT** modify the number of nodes or layers. The place you have to consider is where the network layers are defined near the `TODO` comment is. How many percent can you improve the result?.

Exercise 2: Physics guided loss

In this exercise you are given a network which is suffering from over-fitting, similar to the previous exercise we consider wavefunction data. The code can be found in the script `physics_loss.py`. You should not touch the network or add any regularization methods. You are to improve the result and fix the issue of over-fitting.

To achieve this you are to develop you own custom loss function near the `TODO` comment. You are given the standard MSE function and a custom loss function takes two parameters; the true result Y and the predicted result $\hat{Y}$. You are to add a physics guided loss term to improve the result and get rid of over-fitting. Inspired by Equation 4.6, consider the fact that quantum mechanics is unitary thus,

$$\int_0^L \|\Psi\|^2 \mathrm{d}x = 1. \tag{4.15}$$

How many percent can you improve the result with this knowledge?
Hint: You might need to implement a threshold value.

Exercise 3: Physics network

In this exercise you are to design a simple two component network, which can help it act as the solution to the motion of a projectile motion in 1D.

The Lagrangian is as follows,

$$\mathcal{L} = \frac{1}{2}m\dot{x} - mhx, \tag{4.16}$$

where h is the height. Note that we have the x-axis as the up and downwards direction. Assuming $x_0 = 0$, the solution is,

$$x(t) = v_0 t - \frac{1}{2}gt^2. \tag{4.17}$$

In the the script `physics_loss.py` you will find this to be implemented in the `call` function of the network class. The equation depend on `self.g` and `self.g`, which are the two parameters your network needs to imitate to get the right result. You are therefore to create two networks that yields two predicted parameters Θ_g and Θ_{v_0}, which are what needs to be trained for the output to be correct. In the network class you find the `TODO` comment where you need to define your layers and then you needs to construct the network to train two parameters that such that we get the right output. We are thus using the network to determine the value for v_0 and g. You don't need to create a custom loss function, but you are free to do so if you want.

Hint: You might need initialize the trainable parameters Θ_g and Θ_{v_0} to appropriate values for the training to work.

References

[1] *Multi-Fidelity Physics-Constrained Neural Network and its Application in Materials Modeling*, volume Volume 2A: 45th Design Automation Conference of *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 08 2019. V02AT03A007.

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

[3] Mads J. Ahlebæk, Mads S. Peters, Wei-Chih Huang, Mads T. Frandsen, René L. Eriksen, and Bjarke Jørgensen. The hybrid approach – convolutional neural networks and expectation maximization algorithm – for tomographic reconstruction of hyperspectral images, 2022.

[4] Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke. Using simulation and domain adaptation to improve efficiency of deep robotic grasping, 2017.

[5] Hu Chen, Yi Zhang, Mannudeep Kalra, Feng Lin, Yang Chen, Peixi Liao, Jiliu Zhou, and Ge Wang. Low-dose ct with a residual encoder-decoder convolutional neural network. *IEEE Transactions on Medical Imaging*, 36:2524–2535, 06 2017.

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

[7] Arka Daw, Anuj Karpatne, William Watkins, Jordan Read, and Vipin Kumar. Physics-guided neural networks (pgnn): An application in lake temperature modeling, 2017.

[8] Arka Daw, R. Quinn Thomas, Cayelan C. Carey, Jordan S. Read, Alison P. Appling, and Anuj Karpatne. Physics-guided architecture (pga) of neural networks for quantifying uncertainty in lake temperature modeling, 2019.

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

[10] Xiaowei Jia, Jared Willard, Anuj Karpatne, Jordan Read, Jacob Zwart, Michael Steinbach, and Vipin Kumar. Physics guided rnns for modeling dynamical systems: A case study in simulating lake temperature profiles, 2018.

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

[12] Henry W. Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, July 2017.

[13] 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.

[14] Nikhil Muralidhar, Jie Bu, Ze Cao, Long He, Naren Ramakrishnan, Danesh Tafti, and Anuj Karpatne. Physics-guided design and learning of neural networks for predicting drag force on particle suspensions in moving fluids, 2019.

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

[16] 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.

[17] Jordan S. Read, Xiaowei Jia, Jared Willard, Alison P. Appling, Jacob A. Zwart, Samantha K. Oliver, Anuj Karpatne, Gretchen J. A. Hansen, Paul C. Hanson, William Watkins, Michael Steinbach, and Vipin Kumar. Process-guided deep learning predictions of lake water temperature. *Water Resources Research*, 55(11):9173–9190, 2019.

[18] Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview, 2019.

[19] Russell Stewart and Stefano Ermon. Label-free supervision of neural networks with physics and domain knowledge, 2016.

[20] 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.

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

[22] Yuanhao Xiong, Guanjie Zheng, Kai Xu, and Zhenhui Li. Learning traffic signal control from demonstrations. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 2289–2292, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Tianfang Xu and Albert J. Valocchi. Data-driven methods to improve baseflow prediction of a regional groundwater model. *Computers & Geosciences*, 85:124–136, 2015. Statistical learning in geoscience modelling: Novel algorithms and challenging case studies.

[24] Zeng Yang, Jin-Long Wu, and Heng Xiao. Enforcing deterministic constraints on generative adversarial networks for emulating physical systems, 2020.

[25] Guanjie Zheng, Chang Liu, Hua Wei, P. Jenkins, Chacha Chen, Tao Wen, and Zhenhui Jessie Li. Knowledge-based residual learning. In *International Joint Conference on Artificial Intelligence*, 2021.

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