Machine Learning techniques for numerical solvers

Mini-course SUSTech

11.03 - 14.03

Maria Han Veiga

THE OHIO STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

What to expect

- 5h mini-course
- A brief introduction to machine learning and different ways ML can be useful for numerics
- Mix of slides and codes
- Materials in: https://github.com/hanveiga/sustech-ml-workshop

Schedule

- **Monday:** Introduction to Machine Learning + Computational Framework (1h) 17:00-18:00
- **Tuesday:** Supervised learning: integrating data-driven methods within a numerical solver + ML tricks of the trade (2h) 14:00-16:00
- **Wednesday:** Unsupervised learning: Physics informed neural networks (1h) 14:00-15:00
- **Thursday:** Reinforcement Learning ? (1h) 14:00-15:00

Introduction

- Originally from Portugal 
- PhD, Mathematics, University of Zurich, Switzerland
- Postdoc, University of Michigan
- Fall 2023: Assistant Professor, Dpt. of Mathematics, Ohio State University

It might not be obvious but this is what peak performance looks like

Research interests

Numerical analysis for hyperbolic partial differential equations:

- Development of high-order space-time discretisations
- Structure preserving properties
- Applications in astrophysics

Scientific machine learning:

Development of machine learning methods to enhance computational fluid dynamics codes

- Supervised learning strategies to learn surrogate models (e.g. sub-grid models, shock-capturing methods)
- Guarantees on performance (e.g. physics preserving, error estimates on prediction)

Reinforcement learning:

Establishing convergence of policy gradient methods (i.e. Why/when do they work?)

- Development of theory and novel methods
- Applications of RL in fluid dynamics (e.g. optimal control problems governed by PDEs)

Research interests

Numerical analysis for hyperbolic partial differential equations:

- Development of high-order space-time discretisations
- Structure preserving properties
- Applications in astrophysics

Workshop talk

Scientific machine learning:

Development of machine learning methods to enhance computational fluid dynamics codes

- Supervised learning strategies to learn surrogate models (e.g. sub-grid models, shock-capturing methods)
- Guarantees on performance (e.g. physics preserving, error estimates on prediction)

Reinforcement learning:

Establishing convergence of policy gradient methods (i.e. Why/when do they work?)

- Development of theory and novel methods
- Applications of RL in fluid dynamics (e.g. optimal control problems governed by PDEs)

Day 1: Introduction to ML

Outline

- Quick introduction to ML
  - Definitions
  - Hypothesis classes
- On the optimisation problem
- Neural networks
- Software concerns
  - Software stack
  - Reproducibility

Introduction to ML

- Machine learning aims at building algorithms that autonomously learn how to perform a task from examples

Introduction to ML

- Machine learning aims at building algorithms that autonomously learn how to perform a task from examples

- Formally, it's function approximation

$$f : \mathcal{X} \rightarrow \mathcal{Y}$$

  - $\mathcal{X}$ can be a vector, a representation of text, image, etc…
  - $\mathcal{Y}$ is the output, it can be a number, text, image, etc…

Three main paradigms

- Supervised learning: access to labelled examples (input and output data) and we want to learn the map between these two.
    - e.g. spam filter detection
- Unsupervised learning: access to examples with no labels.
    - e.g. examples are a set of paintings and we want to group them by guessing which come from the same artist / share the same style
- Reinforcement learning: examples are generated from interacting with an environment.
    - e.g. controlling a drone by navigating the world by trial and error

Supervised learning

Setup:

- Input space $\mathcal{X}$ and output space $\mathcal{Y}$
- An unknown mapping $f : \mathcal{X} \to \mathcal{Y}$ to approximate
- A probability distribution D on $\mathcal{X}$
- A dataset $\{(x_i, y_i) : i = 1,...,m\}$ such that $f(x_i) = y_i$ and x_i are independent and identically distribution with common law D
- A hypothesis class $\mathcal{H}$ that is a set of functions mapping $\mathcal{X}$ to $\mathcal{Y}$
- A loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$, such that for $h \in \mathcal{H}$, $\ell(h(x_i), y_i)$ measures the error of the prediction of h at x_i from its true label $f(x_i)$.

Supervised learning

Definition (Classification problem): A problem is a classification problem if the labels are categorical. Formally, if $\mathcal{Y}$ is discrete and the loss is $\ell(y, y') = 1_{y \neq y'}$.

Definition (Regression problem): A problem is a regression problem if the labels take continuous numerical values, i.e. $\mathcal{Y} \in \mathbb{R}$ and the loss is $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$.

Supervised learning

Definition (Classification problem): A problem is a classification problem if the labels are categorical. Formally, if $\mathcal{Y}$ is discrete and the loss is $\ell(y, y') = 1_{y \neq y'}$.

Definition (Regression problem): A problem is a regression problem if the labels take continuous numerical values, i.e. $\mathcal{Y} \in \mathbb{R}$ and the loss is $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$.

The main difference is that in regression the error also provides us with a magnitude of the error.

Hypothesis classes

- Hypothesis class is the space of functions we consider to approximate the unknown function
  - e.g. neural network, linear model, boosting trees, etc…

Hypothesis classes

- Hypothesis class is the space of functions we consider to approximate the unknown function
  - e.g. neural network, linear model, boosting trees, etc…
- Choosing a parametric model means choosing a hypothesis class, learning amounts to finding the right parameters given the class of functions where the approximating function lives
  - e.g. if $\mathscr{H} = \{f(x) = \overrightarrow{w}^T x + b, \overrightarrow{w} \in \mathbb{R}^n, b \in \mathbb{R}\}$ yields the class of n -dimensional linear planes and the parameters are $\overrightarrow{w}$ and b.

Hypothesis classes

- Hypothesis class is the space of functions we consider to approximate the unknown function
  - e.g. neural network, linear model, boosting trees, etc…
- Choosing a parametric model means choosing a hypothesis class, learning amounts to finding the right parameters given the class of functions where the approximating function lives
  - e.g. if $\mathscr{H} = \{f(x) = \overrightarrow{w}^T x + b, \overrightarrow{w} \in \mathbb{R}^n, b \in \mathbb{R}\}$ yields the class of n -dimensional linear planes and the parameters are $\overrightarrow{w}$ and b.
- How to choose a suitable hypothesis class is somewhat of an *art*

Measuring performance

Definition (Generalisation error): For a predictor $h \in \mathscr{H}$, the generalisation error is defined as

$$R_D(h) = \mathbb{E}_{x \sim D}\left[\ell(h(x), f(x))\right].$$

Goal of supervised learning, for a given f and D is:

$$\min_{h \in \mathscr{H}} R_D(h)$$

Measuring performance

Definition (Generalisation error): For a predictor $h \in \mathscr{H}$, the generalisation error is defined as

$$R_D(h) = \mathbb{E}_{x \sim D}\left[\ell(h(x), f(x))\right].$$

Goal of supervised learning, for a given f and D is:

$$\min_{h \in \mathscr{H}} R_D(h)$$

We are unable to evaluate this quantity, as f and D are unknown.

Measuring performance

We have access to a finite dataset $S = \{(x_i, y_i) : i = 1,...,m\}$.

Definition (Empirical error/risk): For a predictor $h \in \mathscr{H}$ and a dataset S, the empirical error is defined as

$$L(h) = \frac{1}{m} \sum_{i}^{m} \ell(h(x_i), y_i)$$

Measuring performance

We have access to a finite dataset $S = \{(x_i, y_i) : i = 1,...,m\}$.

Definition (Empirical error/risk): For a predictor $h \in \mathscr{H}$ and a dataset S, the empirical error is defined as

$$L(h) = \frac{1}{m} \sum_{i}^{m} \ell(h(x_i), y_i)$$

Note that if x_i are i.i.d and $m \to \infty$, by the law of large numbers this quantity converges to the generalisation error.

Measuring performance

We solve the following minimisation problem, called the *Empirical Risk Minimisation* problem:

Given a dataset S:

$$\min_{h \in \mathcal{H}} L(h)$$

This is the quantity we minimise with when trying to find a good predictor. Of course, you can already see that there might be some issues because of finiteness of S…

Measuring performance

We solve the following minimisation problem, called the *Empirical Risk Minimisation* problem:

Given a dataset S:

$$\min_{h \in \mathcal{H}} L(h)$$

This is the quantity we minimise with when trying to find a good predictor. Of course, you can already see that there might be some issues because of finiteness of S…

Remark: We defined L using labelled data, but this is not necessary. L can encode, for example, some minimisation between elements in $\mathcal{X}$, or some reward. Meaning, what we say here holds for unsupervised/reinforcement learning

Losses

- Typical losses for regression problems:
  - $\ell(y, y') = ||y - y'||_2^2$, squared loss function
  - $\ell(y, y') = |y - y'|$, L_1 loss function
- Typical losses for classification:
  - Instead of $1_{y \neq y'}$, which is discontinuous, surrogate losses or smooth losses are considered

Loss minimisation

Given a dataset S:

$$\min_{h \in \mathcal{H}} L(h)$$

Loss minimisation

Given a dataset S:

$$\min_{h \in \mathcal{H}} L(h)$$

For some losses and models in certain hypothesis classes, derive the minimum explicitly:

e.g. For loss ℓ the square loss and $h(x) = \vec{w}^T x + b$ a linear model, $L(h)$ is a convex function with respect to its parameters and the minimiser is:

$$\vec{w} = (X^T X)^{-1} X^T Y, \text{ where } X = [\vec{x}_1, \ldots, \vec{x}_m]^T,\ Y = [y_1, \ldots, y_m]^T$$

Loss minimisation

Given a dataset S:

$$\min_{h \in \mathcal{H}} L(h)$$

For some losses and models in certain hypothesis classes, derive the minimum explicitly:

e.g. For loss ℓ the square loss and $h(x) = \vec{w}^T x + b$ a linear model, $L(h)$ is a convex function with respect to its parameters and the minimiser is:

$$\vec{w} = (X^T X)^{-1} X^T Y, \text{ where } X = [\vec{x}_1, \ldots, \vec{x}_m]^T,\ Y = [y_1, \ldots, y_m]^T$$

Unfortunately, this is not the case for most models of interest.

Gradient descent

Gradient descent is a common way to solve with the empirical risk minimisation.

First order iterative method that finds a local minimum of a differentiable function.

Definition: Let $w_0 \in \mathbb{R}^p$ and fix $\eta > 0$. Define w_k recursively for $k \geq 0$,

$$w_{k+1} = w_k - \eta \nabla L(w_k)$$

Gradient descent

Theorem: If $L \in \mathscr{C}^2$ is L-smooth and convex, and $\eta < \frac{1}{L}$, then there exists a global minimum w^* s.t.

$$0 \leq L(w_k) - L(w^*) \leq \frac{||w_0 - w^*||}{2\eta K}.$$

Convergence rate $\mathcal{O}(1/K)$.

Gradient descent

Theorem: If $L \in \mathscr{C}^2$ is L-smooth and convex, and $\eta < \frac{1}{L}$, then there exists a global minimum w^* s.t.

$$0 \leq L(w_k) - L(w^*) \leq \frac{||w_0 - w^*||}{2\eta K}.$$

Convergence rate $\mathcal{O}(1/K)$.

Even if loss ℓ is convex in both arguments, it might not be convex with respect to the parameters of the model.

Gradient descent

Theorem: If $L \in \mathscr{C}^2$ is L-smooth, $\eta < \frac{2}{L}$, and $K \in \mathbb{N}$, then the cost function under gradient descent converges. That is $\lim_{K\to\infty} L(w_K)$ exists and

$$\lim_{K\to\infty} \nabla_w C(w_K) = 0.$$

Gradient descent

Theorem: If $L \in \mathscr{C}^2$ is L-smooth, $\eta < \frac{2}{L}$, and $K \in \mathbb{N}$, then the cost function under gradient descent converges. That is $\lim_{K\to\infty} L(w_K)$ exists and

$$\lim_{K\to\infty} \nabla_w C(w_K) = 0.$$

Theorem (Lee et al, 2016): Let $L \in \mathscr{C}^2$ and L-smooth, and w^* is a strict saddle point. Assume $0 < \eta < \frac{1}{L}$, then

$$Pr\left(\lim_{k\to\infty} w_k = w^* \right) = 0.$$

Gradient descent

Non-convex optimisation:

- Under mild assumptions ($L \in \mathscr{C}^2$, L-smooth and learning rate), GD converges to a critical point
- Asymptotically it does not get stuck on strict saddle points (Lee et al. 2016)
- But it might take exponential time to escape saddle points (Du et al. 2017)
- Noisy gradient estimates (stochastic gradient descent) or momentum can help escape saddle points more quickly

Gradient descent

In practice:

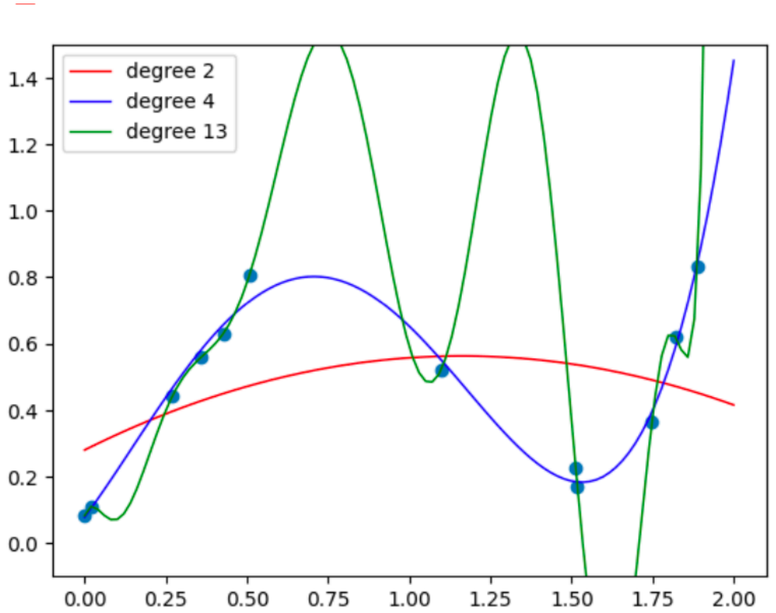
- Momentum based GD (e.g. ADAM, Nesterov, AdaGrad, etc…)
- Decaying learning rate η
- Noisy gradient estimates (batch gradient)

Over-fitting & cross-validation

Definition: Let $h \in \mathscr{H}$ be a predictor. In the Empirical Risk Minimisation framework, the difference between the generalisation loss of h and its empirical loss is the generalisation gap.

- Estimating the generalisation gap: train with dataset S', estimate the generalisation error with dataset S'', where $S' \cap S'' = \varnothing$.

- **Overfitting** occurs when h fits the data well but is too complex to generalise outside of the dataset, i.e. when $L(h) \approx 0$ but $R_D(h)$ is large.



ML pipeline

- Given dataset $S = \{(x_i, y_i) : i = 1,...,m\}$, split into training set S_T, validation set S_V and test set S_{Test}.
- For a set of hypotheses classes $\{\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_n\}$, solve empirical risk minimisation with S_T
- Use S_V to select the best performing hypothesis class, $\mathcal{H}^*$.
- Solve empirical risk minimisation with $S_T \cup S_V$ to find h^*.
- Evaluate h^* on S_{Test}. This gives an estimate of the generalisation error.

ML pipeline

Cross-validation: Repeat the procedure below k times, and evaluate average validation error.

- Given dataset $S = \{(x_i, y_i) : i = 1,...,m\}$, split into training set S_T and test set S_{Test}.

  - Split S_T into training set S_T and validation set S_V.
  - For a set of hypotheses classes $\{\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_n\}$, solve empirical risk minimisation with S_T
  - Use S_V to select the best performing hypothesis class, $\mathcal{H}^*$.

- Solve empirical risk minimisation with $S_T \cup S_V$ to find h^*.
- Evaluate h^* on S_{Test}. This gives an estimate of the generalisation error.

ML pipeline

Cross-validation: Repeat the procedure below k times, and evaluate average validation error.

- Given dataset S … st set S_{Test}.
  - Split S_T into t…
  - For a set of h… risk minimisation with S_T
  - Use S_V to sel…
- Solve empirical …
- Evaluate h^* on S_{Test}. This gives an estimate of the generalisation error.

In practice, you want to make sure that the test set and training set are different enough to evaluate generalisation error. But also not too *different*, to not be an out-of-distribution sample.

Common hypothesis classes

Neural networks

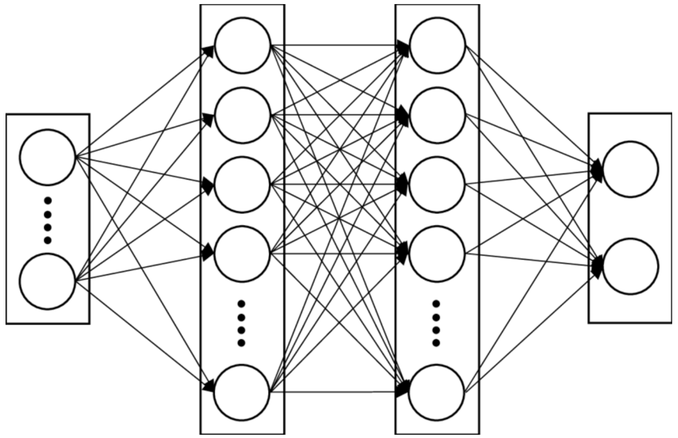
Definition (Fully connected feedforward neural network): Let

- $L \in \mathbb{N}$ denote the number of layers
- $\sigma_i : \mathbb{R} \to \mathbb{R}$ the activation functions $(i = 1,...,N)$
- $d_0, \ldots, d_L \in \mathbb{R}$ the number of neutrons on the $l-$th layer, $l = 0,...,N$

Then, the feedforward neural network can be defined as follows: let $\alpha^{(0)}(x) := x$ for $x \in \mathbb{R}^{d_0}$ and define the recursion for $l = 1,...,L$

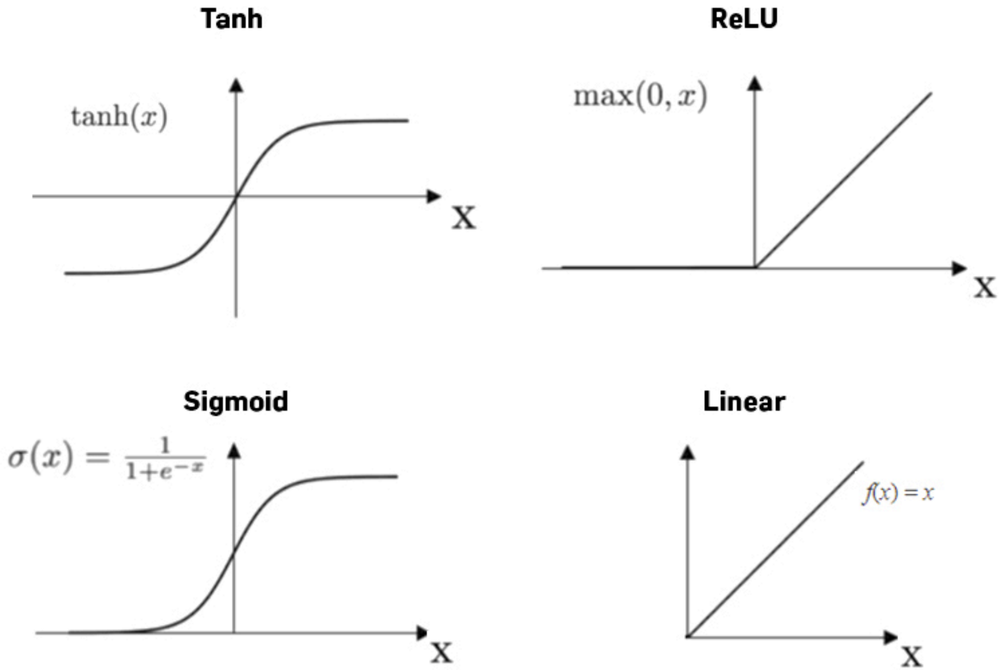
$$\tilde{\alpha}^{(l)}(x) = W_l \alpha^{(l-1)}(x) + b_l \qquad \textit{Pre-activation at layer l}$$

$$\alpha^{(l)}(x) = \sigma_l(\tilde{\alpha}^{l}(x)) \,. \qquad \textit{Post-activation at layer l}$$



Neural networks

- The number of parameters in a neural networks is $\sum_{l=1}^{L} d_l d_{l-1} + d_l$
- σ_i gives the nonlinearity to the neural network



Neural networks

- Neural networks are trained using gradient descent (type) of algorithms and **back-propagation**

Neural networks

- Neural networks are trained using gradient descent (type) of algorithms and **back-propagation**
- Recursively compute the gradients:
  - Suppose we have computed $\frac{\partial C}{\partial \alpha_n} = \left(\frac{\partial C}{\partial \alpha_{n,1}}, \ldots, \frac{\partial C}{\partial \alpha_{n,d_n}} \right)$, the gradients for parameters at layer n
  - Then, recursively, we can compute the update for parameters at layer $n-1$

$$\frac{\partial C}{\partial \tilde{\alpha}_n} = \sigma_n'(\tilde{\alpha}_n) \circ \frac{\partial C}{\partial \alpha_n}$$

$$\frac{\partial C}{\partial W_n} = \left(\frac{\partial C}{\partial \tilde{\alpha}} \right) \left(\alpha_{n-1} \right)^T$$

$$\frac{\partial C}{\partial \alpha_{n-1}} = W_n^T \frac{\partial C}{\partial \tilde{\alpha}_n},$$

Neural networks

- Neural networks are trained using gradient descent (type) of algorithms and **back-propagation**
- Recursively compute the gradients:
  - Suppose we have computed $\frac{\partial C}{\partial \alpha_n} = \left(\frac{\partial C}{\partial \alpha_{n,1}}, \ldots, \frac{\partial C}{\partial \alpha_{n,d_n}} \right)$, the gradients for parameters at layer n
  - Then, recursively, we can compute the update for parameters at layer $n-1$

$$\frac{\partial C}{\partial \tilde{\alpha}_n} = \sigma_n'(\tilde{\alpha}_n) \circ \frac{\partial C}{\partial \alpha_n}$$

$$\frac{\partial C}{\partial W_n} = \left(\frac{\partial C}{\partial \tilde{\alpha}} \right) \left(\alpha_{n-1} \right)^T$$

$$\frac{\partial C}{\partial \alpha_{n-1}} = W_n^T \frac{\partial C}{\partial \tilde{\alpha}_n},$$

  - Or simpler, using the δ notation:

$$\frac{\partial C}{\partial W_n} = \delta^n \alpha_{n-1}^T$$

$$\delta^L = \sigma_L'(\tilde{\alpha}_L) \circ \frac{\partial C}{\partial \alpha_L},$$

$$\delta^{n-1} = \sigma_{n-1}'(\tilde{\alpha}_{n-1}) \circ W_{n-1}^T \delta^n, \quad n = 1,...,L\,.$$

Neural networks

- Neural networks are trained using gradient descent (type) of algorithms and **back-propagation**
- Recursively compute the gradients:
  - Suppose we have computed $\frac{\partial C}{\partial \alpha_n} = \left(\frac{\partial C}{\partial \alpha_{n,1}}, \ldots, \frac{\partial C}{\partial \alpha_{n,d_n}} \right)$, the gradients for parameters at layer n
  - Then, recursively, we can compute the update for parameters at layer $n-1$

$$\frac{\partial C}{\partial \tilde{\alpha}_n} = \sigma_n'(\tilde{\alpha}_n) \circ \frac{\partial C}{\partial \alpha_n}$$

$$\frac{\partial C}{\partial W_n} = \left(\frac{\partial C}{\partial \tilde{\alpha}} \right) \left(\alpha_{n-1} \right)^T$$

$$\frac{\partial C}{\partial \alpha_{n-1}} = W_n^T \frac{\partial C}{\partial \tilde{\alpha}_n},$$

  - Or simpler, using the δ notation:

$$\frac{\partial C}{\partial W_n} = \delta^n \alpha_{n-1}^T$$

$$\delta^L = \sigma_L'(\tilde{\alpha}_L) \circ \frac{\partial C}{\partial \alpha_L},$$

$$\delta^{n-1} = \sigma_{n-1}'(\tilde{\alpha}_{n-1}) \circ W_{n-1}^T \delta^n, \quad n = 1,\ldots,L.$$

The update on the $n-1$-th layer depends on the gradients upstream, this can cause problems….

Neural networks

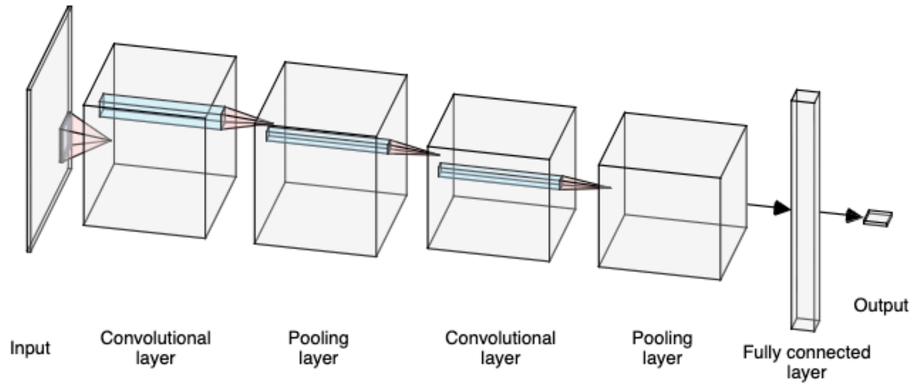
Approximation properties:

- Cybenko (1989) shows a 1-hidden layer neural network with sigmoidal activation function is dense in $C(I_n)$ with respect to the supremum norm.
- Extensions of this result exist for n-hidden layer, classification, different activation functions.

Neural networks: beyond fully connected

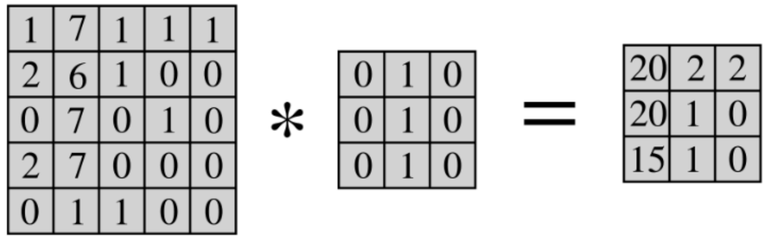
- Convolutional neural networks — inputs are not vectors but matrices or high order tensors
- Account for *spatial* structure of input

- Input space: $\mathcal{X} = (\mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d})^k$, where k denotes the number of *channels.*
  - e.g. a coloured image of size $n_1 \times n_2$ is represented by an element in $(\mathbb{R}^{n_1 \times n_2})^3$, because of the RGB channels.

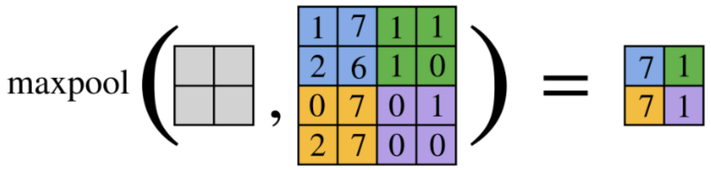


- **Main components:**
  - **Convolution layer:** a collection of filters $g_1, \ldots, g_l \in (\mathbb{R}^{n_{f_1} \times \ldots \times n_{f_d}})^k$
  - **Pooling layer:** downsizes the objects to reduce $n_1 \times \ldots \times n_d$

Neural networks: beyond fully connected



Example: Convolution of a 3x3 filter with a 5x5 matrix



Example: Pooling with a max operator using a 2x2 filter with stride = 2

Neural networks: beyond fully connected

- Others — transformers, encoders, recurrent, residual, generative adversarial, etc…
- Many common features
- We will not go in depth about these

Computational concerns

Outline

- Computational stack
- Reproducibility

Computational stack

Python:

- Scikit-learn: traditional machine learning tools
  - https://scikit-learn.org/
  - Cross-validation, datasets, feature transformation, many hypothesis classes (including neural networks)
- PyTorch: deep learning
  - https://pytorch.org/
  - Similar functionality as tensorflow, keras
  - Personal opinion: more explicit and easy to develop with, ML (theory) researchers seem to like it more

Reproducibility

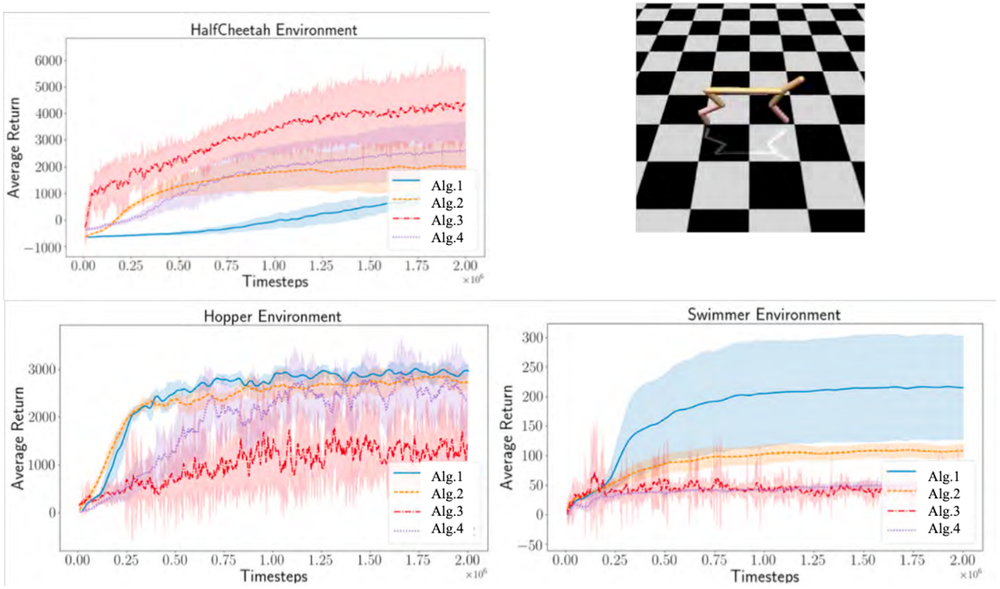
- *Experimental* Machine Learning can be thought of as an experimental science
- There are many sources of randomness:
  - sampling, optimisation, parameter initialisation, parallelism, etc…
- Other sources of *non-uniqueness*:
  - Development environment (e.g. different software versions, floating point accuracy, etc…)
- Experimental results are only as good as we can reproduce them

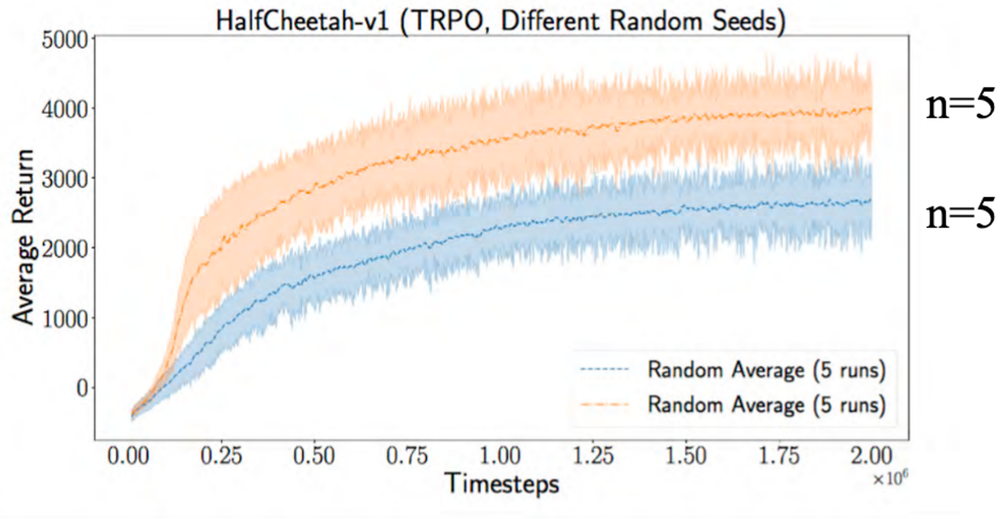
Reproducibility

A case study (Pineau et al. 2020)

- They studied the behaviour of common baseline algorithms that are used to compare to new algorithms

Consider Mujoco simulator:





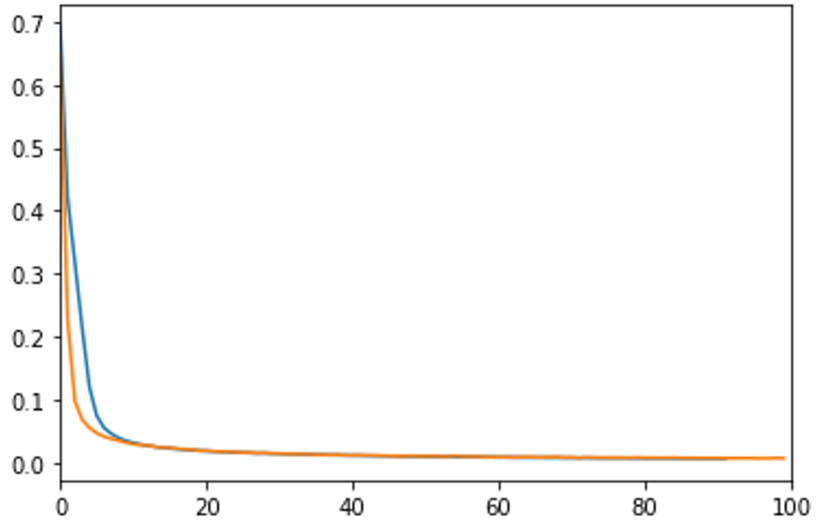
Reproducibility

```
# Define model (MLP) with sklearn
from sklearn.neural_network import MLPRegressor
modelSK = MLPRegressor(hidden_layer_sizes=[24,24,24])
# Fit model
modelSK.fit(scaled_x_train, scaled_y_train)
```

```
import keras
from keras.models import Sequential
from keras.layers import Input, Dense

inputs = Input(shape=(input_size,))
f = Dense(24,activation='relu')(inputs)
f = Dense(24,activation='relu')(f)
f = Dense(24,activation='relu')(f)
outputs = Dense(1)(f)
model = keras.Model(inputs, outputs)
model.compile('Adam', loss="mse", metrics="mse")
```

- Same training / test set
- Same optimizer
- Architecture



	SK-Learn	Keras
R2 score	0.9789	0.9844

Tools that can be helpful

- **Dataset:**
  - Dryad https://datadryad.org/stash/
  - Zenodo https://zenodo.org/
  - University-hosted repositories
- **Experiments tracking:**
  - MLFlow https://mlflow.org/
  - TensorBoard https://www.tensorflow.org/tensorboard
  - Weights and biases https://wandb.ai/site
  - Or your own way to store trained models and experimental set-up
- **Presenting results and sharing code:**
  - Google colab
  - Github
  - Development environment, e.g. environment.yml

The Machine Learning Reproducibility Checklist (v2.0, Apr.7 2020)

For all **models** and **algorithms** presented, check if you include:

- ❑ A clear description of the mathematical setting, algorithm, and/or model.
- ❑ A clear explanation of any assumptions.
- ❑ An analysis of the complexity (time, space, sample size) of any algorithm.

For any **theoretical claim**, check if you include:

- ❑ A clear statement of the claim.
- ❑ A complete proof of the claim.

For all **datasets** used, check if you include:

- ❑ The relevant statistics, such as number of examples.
- ❑ The details of train / validation / test splits.
- ❑ An explanation of any data that were excluded, and all pre-processing step.
- ❑ A link to a downloadable version of the dataset or simulation environment.
- ❑ For new data collected, a complete description of the data collection process, such as instructions to annotators and methods for quality control.

For all shared **code** related to this work, check if you include:

- ❑ Specification of dependencies.
- ❑ Training code.
- ❑ Evaluation code.
- ❑ (Pre-)trained model(s).
- ❑ README file includes table of results accompanied by precise command to run to produce those results.

For all reported **experimental results**, check if you include:

- ❑ The range of hyper-parameters considered, method to select the best hyper-parameter configuration, and specification of all hyper-parameters used to generate results.
- ❑ The exact number of training and evaluation runs.
- ❑ A clear definition of the specific measure or statistics used to report results.
- ❑ A description of results with central tendency (e.g. mean) & variation (e.g. error bars).
- ❑ The average runtime for each result, or estimated energy cost.
- ❑ A description of the computing infrastructure used.

Reproduced from: *www.cs.mcgill.ca/~jpineau/ReproducibilityChecklist-v2.0.pdf*

- Proposed by Dr. Pineau
- Adopted by NeurIPS

A shorter checklist

- *environment.yml* to recreate runtime environment
- Versioned dataset (if size allows it, otherwise use timestamps)
- Full code used to preprocess data
- Full code used to train and evaluate model(s)
- Hyperparameters used for training (model, optimiser)
- Final trained models in some universally readable format
- Strict version control of GitHub repo (for example)

Summary

- Quick intro to ML: problem statement, optimisation problem, hypothesis classes
- Computational stack: software and reproducibility
- **Homework:** what types of problems of your own interest can you frame in a supervised learning framework?