ASTR 206 Lecture Notes

Simeon Bird

December 1, 2023

1 Machine Learning

Machine Learning can mean a variety of things to a variety of people.

An important division is between the unsupervised machine learning algorithms from SciKit Learn (`https://scikit-learn.org`) and the deep learning techniques from pytorch (`https://pytorch.org/`).

Line or polynomial fitting could also be classed as machine learning, although we have been doing it forever, and many of the algorithms for ML are similar to those from line fitting, but used on more data and higher dimensional problems.

An example of a scikit learn technique is the support vector machine, for classification. These models are useful for fitting complex data when there is limited theory.

An example of a pytorch technique is a neural network photometric redshift predictor. These models are useful for fitting complex data when you have a very large training set.

These techniques differ mostly by their number of **free parameters** and thus their training requirements. Line fitting has $O(2)$ free parameters. An SVM has $O(10)$ and a neural network may have very many. $O(10^6)$ is not uncommon. A characteristic of ML is that the models are over-parametrised: there may be more free parameters than training data. If done carefully these over-parametrised models can still be predictive.

1.1 Training Sets

All models have to be **trained**. The size and quality of the training set is extremely important, possibly more so than the actual algorithms used for the model (a side effect of being over-parametrised is that several models will often do about as well as each other).

Larger, more diverse, training sets drive much progress in machine learning. ChatGPT could not exist without stack overflow. The larger the training set, the easier it is to identify the patterns in it and to fix the parameters of the models.

The trend over the last ten years has been to increase training set size as much as possible. However, the limits in this direction have been reached and the trend over the next ten years is likely to be to focus on increasing the quality of the dataset. For example, a model trained on a dataset of quasars that actually contains some stars will be less reliable than one trained only on quasars.

1.2 Uses of Machine Learning in Astronomy

There are two common uses of machine learning in astronomy:

1. Interpolation or Surrogate modelling
2. Type Classification

In surrogate modelling the goal is to train a model on a set of expensive simulations. The model can make predictions for the output it would see for another simulation with slightly different parameters. Evaluating the model is computationally cheaper than running a new simulation. It is important when doing interpolation to stay within your training set when evaluating. Because the training set is being evaluated in the higher-dimensional space of neural network free parameters, it may not be completely obvious where the boundaries of your training set are.

In type classification, you need to determine, from possibly noisy data, which class of objects something falls into. The classic example is handwriting recognition. It is important that you do not give your ML model, highly trained on English text, non-latin script. Another example are image classifiers along the lines of "Do these images contain cats"? In astronomy the corresponding questions are things like "Does this image of the sky contain a galaxy?"

Most places you do not need the full power of deep learning. If you can get away with fitting a power law then you should. It is advisable to try the simplest methods first and work your way up to the complex methods. Almost all my group's work uses Gaussian Processes, which are easy to train, and have clear applicability boundaries. However, they require a matrix inversion step for training, which is $O(N^3)$ and this limits them to relatively small datasets. See PHYS 203 for more on GPs.

Note that we can always in principle solve surrogate problems by just throwing computer time at the problem and running more simulations. This also has the advantage that it will always give reliable answers, and is preferred if possible. However, it is not always possible.

1.3 Examples From Our N-body Simulation Code

Let us imagine that we have run an N-body simulation code for a long period of time, and found mergers. The properties of the merger are written to disc, including such things as the time of the merger, the position of the merger, and the IDs of the objects merging. We can do several such simulations with different numbers of N-body particles, and with (for example) different initial density profiles. This is our **Training Set**. There are a few questions we can ask of this dataset using machine learning.

1.3.1 Classification

One classification problem could be: "Is this a black hole that will undergo a merger before some time?" However, this is better framed as a regression problem, to estimate a merger probability for each black hole.

A better example would be to create a mock dataset of observations of merging stars / black holes, and non-merging stars / black holes. Since the true underlying properties of the merger are known from the simulation, one could train a model and then apply it to real data. The observations could be X-ray emission or gravitational waves.

There are fewer papers on this kind of thing because creating a high fidelity mock dataset requires people to have domain knowledge. It is very important that the properties of the mock dataset be exactly the same as the properties of the real dataset. It is, however, the more powerful application.

1.3.2 Interpolation

Possible surrogate questions:

- The total number of mergers in the simulation as a function of time, or input parameters.

For this you do not need DL. You could try fitting a power law, or you could try a Gaussian process.

- At what location and time will the next black hole merger take place?
- What are the positions of the black holes at a future time?
- How do the locations of the first merger depend on the number of N-body particles?

For these types of questions you need a deep learning model: the number of objects to consider is much larger than the number that can fit into a Gaussian process.

Let's start thinking about how we would build a surrogate model able to address these questions, using pytorch. We need to start by assembling a training set, and thinking about what data we need. Then we need to start thinking about the type of model we would want.

1.3.3 Things To Do to Make an ML Model

1. Design and Assemble the Training Set
2. Design the ML architecture you want: let's use a simple feed-forward CNN.
3. Train the model on a GPU
4. Check that the trained model works using a validation set.

2 How Does All This Work

2.1 Overfitting, Back-Propagation and Cross-Validation

Because the models are over-parametrised, it is easy to make one that models its training set very very accurately, but has poor predictive power. Think of a model trained on a biased dice that always gives six. It will be extremely predictive but work poorly if you take it to a casino.

Cross-validation is a way to avoid this. The idea is that you train repeatedly on a fraction of your training set, leaving out a subset, and then try to validate that you can predice the remaining fraction, which you did not train on. You may repeat the process, leaving out different sections at different times.

2.2 Training and Defining a Loss Function

Most ML models are looking for a global minimum in a loss function. The minimum is found using Stochastic Gradient Descent (SGD) or its descendant, the ADAM optimiser.

Stochastic Gradient Descent finds the gradient of the loss function at a particular set of parameters θ. The gradients are computed only for a single datapoint in the training set, which makes it very parallelizable:

$$\theta_{i+1} = \theta_i - \eta \cdot \nabla_\theta L(\theta, x_i) \tag{1}$$

When the parameters are close to a minimum, the gradients will be small and the parameters will not change. Many optimisers have a problem that they get stuck in local, rather than global, minima. SGD avoids this because the minimum for one part of the training data need not be the minimum for another, so in the early phases it will jump around. The step size (or learning rate) η should be set to gradually decrease as the training progresses.

ADAM is (more or less) a generalisation of SGD to use estimates of higher derivative gradients, and to include a term that models the uncertainty in the current gradient estimate. It seems to be better at finding the minimum in high dimensions and has become almost universal.

2.3 Types of Models (by increasing training difficulty)

Be advised that the ML field is full of words designed with the idea that the computer is 'thinking'. Most of these words have a 1-1 mapping onto linear algebra concepts. For example, 'neuron' means 'small matrix with estimated weights'.

CNN: Simply minimises the loss function.

GAN: Trains two models against each other, a discriminator and a generator. The generator makes a fake model and the discriminator tries to tell it apart from the real model. Generally does not converge, except to a saddle point.

Transformers: ChatGPT. For word and language models: `https://blog.research.google/2017/08/transformer-novel-neural-network.html`.

Diffusion Models: Maps a complex image onto a Gaussian random field by iterative transforms.

2.3.1 Normalizing Flows

Essentially they learn the coordinate transforms for an integral, mapping them onto a Gaussian. Likely to be the most useful ML technique for inference, but currently quite new.