Markov Chain Monte Carlo methods and neural networks

Statistics and Data Science

Spring 2025

Goals for today: you should be able to...

- **lecture 26/27 notebook:**
  - Use emcee to perform MCMC evaluation of a multidimensional posterior distribution
- Describe nested sampling and Hamiltonian Monte Carlo methods
- Understand the basics of neural networks and deep learning

  - **Reminder**: please see Slack/Canvas for presentation schedule and let me know ASAP about conflicts!
  - Notebooks are due soon!

Review: Markov Chain Monte Carlo methods

- Start at some point in parameter space; jump to another point with probability proportional to the ratio of the posterior probability at the new point to the posterior at the original point
- As the number of points approach infinity, the distribution of points in the resulting Markov Chain will follow the desired posterior probability distribution
- This is not true at small N: there is a 'burn-in' period while the chain approaches the region where the posterior is high

- Figure from wikimedia.org . Red points are the ones after burn-in.

Review: Markov Chain Monte Carlo methods

- Start at some point in parameter space; jump to another point with probability proportional to the ratio of the posterior probability at the new point to the posterior at the original point
- As the number of points approach infinity, the distribution of points in the resulting Markov Chain will follow the desired posterior probability distribution
- This is not true at small N: there is a 'burn-in' period while the chain approaches the region where the posterior is high

Source: https://twiecki.io/blog/2014/01/02/visualizing-mcmc/

Review: The affine-invariant ensemble sampler

- Relies on sending a number of 'walkers' on individual chains, rather than only following one Markov chain
- Test points are along the line between a pair of walkers (instead of in a neighborhood of the original value)
- Can explore parameter space more efficiently than Metropolis-Hastings algorithm

Algorithm 2 A single stretch move update step from GW10

for $k = 1, \ldots, K$ **do**
  Draw a walker X_j at random from the complementary ensemble $S_{[k]}(t)$
  $z \leftarrow Z \sim g(z)$, Equation (10)
  $Y \leftarrow X_j + z\,[X_k(t) - X_j]$
  $q \leftarrow z^{N-1}\,p(Y)/p(X_k(t))$ *// This line is generally expensive*
  $r \leftarrow R \sim [0, 1]$
  if $r \leq q$, Equation (9) **then**
    $X_k(t+1) \leftarrow Y$
  else
    $X_k(t+1) \leftarrow X_k(t)$
  end if
end for

$$g(z) \propto \begin{cases} \dfrac{1}{\sqrt{z}} & \text{if } z \in \left[\dfrac{1}{a}, a\right], \\ 0 & \text{otherwise} \end{cases}$$

where a is an adjustable scale parameter that GW10 set to 2.

Review: Applying MCMC methods to an astronomical dataset

- We are working with the Pantheon compilation of supernova distances from Scolnic et al. 2018
- For each SN in the Pantheon sample, we have measurements of distance modulus (μ) that we can compare to predictions for some cosmological model:

$$\mu = m - M = 5 \log_{10} (d_L / 10 \text{ pc})$$

where m is apparent magnitude, M is absolute magnitude, and d_L is luminosity distance.

- Last time we implemented our likelihood & priors, and set up parameters for `emcee`

Basic parameters for running emcee

- Next we initialize an Ensemble Sampler object:

```
import emcee # import the emcee package

print(f'emcee version: {emcee.__version__}')

# the data arrays used in calculating the posterior are passed
#  as 'arguments' in emcee:(the redshifts, the distance moduli,
#  and errors in our case).  We pass them as a tuple.
argslist = (redshift, mu, error)

# set up the sampler: we provide the number of walkers;
# the number of dimensions we are sampling over;
# the name of our posterior function;
# and then the arguments, which provide the data values.
sampler = emcee.EnsembleSampler(Nens, ndims, log_posterior,
                                 args=argslist)
```

Doing all the work (this may take a few minutes)

```
# we will use this to see how long things take
from time import time

# pass the initial samples and total number of samples required

t0 = time() # store the time at start
sampler.reset()

# running with progress=True will show a progress bar
sampler.run_mcmc(initial_samples, Nsamples + Nburnin,
                 progress=True,store=True);
t1 = time() # store the time at end

print(f"Time taken to run 'emcee' is {t1-t0:.2f} seconds")

# extract the samples (removing the burn-in).
# The first index in sampler.chain runs over the different walkers;
#  the second, over the steps each walker takes; and the third,
#  over the different parameters being fit.
samples_emcee = sampler.chain[:, Nburnin:, :].reshape((-1, ndims))

lnprob_emcee = sampler.lnprobability[:,Nburnin:]
```

Ways to check results from MCMC

- 1) **Look at the acceptance rate of steps for each walker.** If this is near zero, the walkers are having trouble finding good parts of parameter space; if one, they aren't exploring at all. 20-50% is generally seen as ideal

```
print(f'Mean of acceptance fraction across walkers: {np.mean(sampler.acceptance_fraction):.4f}')

print(f'Std. dev. of acceptance fraction across walkers: {np.std(sampler.acceptance_fraction):.4f}')
```

Ways to check results from MCMC

- 2) **Look at the trajectory the walkers take in each parameter**. You should find that after an initial period where they start close to their initial position, they bounce around a more limited range once they find a good regime in parameter space. The initial part of the chains (the 'burn-in') should be discarded and not used in analysis.

```
plt.figure(figsize=(10,5))
for i in arange(0,Nens,10):
    plt.plot(np.exp(sampler.chain[i,:,0]),alpha=0.37)
plt.ylabel(r'$\Omega_m$')
```

Ways to check results from MCMC

- 3) Look at the distributions of values in the chains in one dimension (which should be proportional to the posterior for each parameter) and in each pair of dimensions. Make sure that if there are multiple islands of nonnegligible probability they are well- explored, and that there are not islands of probability concentrated around your initial guess.
  - Dan Foreman-Mackey's `corner` Python package provides a good way of doing this
    - There is one **substantial** caveat: it draws contours that do NOT correspond to 68, 95%, etc. credible intervals, by default. See https://corner.readthedocs.io/en/latest/pages/sigmas/ for an explanation of what it does actually do.

Ways to check results from MCMC

```
import corner
plt.figure(figsize=(10,10))

# corner takes as input the values to be plotted against each other
# (as combined scatter/contour plots) and for which histograms will be shown,
# as well many optional inputs such  as labels for each parameter,
# true values of the parameters (or best-fit values) to overplot;
# whether to overplot e.g. 68% ranges on the histograms; and whether to add
# titles to each subplot with mean + uncertainty from the MCMC results.
fig = corner.corner(np.exp(samples_emcee), labels=["$\Omega_m$", "$\Omega_\Lambda$"],
                    truths=[0.284,0.716],bins=40, quantiles=[0.16, 0.5, 0.84],
                    show_titles=True,title_kwargs={"fontsize": 16},label_kwargs={"fontsize": 16})
```

Ways to check results from MCMC

- There are a couple of options to get the contour levels right:

  A. A Bayesian credible region will contain (68%, 95%, etc.) of the total probability, and hence that fraction of the number of points. The smallest possible credible region will be the one that contains the (68%, 95%, etc.) of points with highest probability.

     - Conveniently, in corner the contour levels are set as quantiles; if you want just a 68% contour specify `levels=(0.68,)` (with the comma!); for 68 + 95%, do `levels = (0.68,0.95)`

  B. If you are doing a frequentist (likelihood) analysis, the difference in -2 ln (likelihood) (compared to the maximum likelihood) evaluated at the true value of parameters should be distributed as a chi-squared distribution with (number of parameters) degrees of freedom. We can therefore set cutoff values of -2 ln (likelihood) corresponding to the 68, 95%, etc. confidence regions using the percentile point function for the chi-squared distribution. ln(likelihood) should be stored for all the points in your chain by the sampler.

Ways to check results from MCMC

- 4) Overlay your data with the best-fit model (or, better yet, an ensemble of models, each plotted with a small alpha) to make sure the data and model are a reasonable match.
  - We can use the `sampler.flatlnprobability` attribute to find the chain step (combining all walkers, and including burnin) with the highest log posterior.
  - The argmax attribute will be the index corresponding to that point.

```
# get the index in the flattened chain with the maximum probability

whmax=(sampler.flatlnprobability.argmax())

# determine omegam and omegal at that point.
omegam_opt,omegal_opt = np.exp(sampler.flatchain[whmax,:])
print(f'omegam: {omegam_opt:.4f}, omegal: {omegal_opt:.4f}')
```

Ways to check results from MCMC

- 5) Check the autocorrelation time of your chains. See https://emcee.readthedocs.io/en/latest/tutorials/autocorr/ .
  - To calculate the autocorrelation time (a measure of how fast the samples get uncorrelated), you can do: `sampler.get_autocorr_time()`
  - Unfortunately with Nsamples=200 emcee is unable to calculate this quantity accurately, longer chains are needed.
  - We want the autocorrelation time to be much less than the length of the chain.
  - Some analyses will use only every N'th sample from a chain, where N >~ the autocorrelation time, to avoid using correlated samples.

Calculating the Evidence (for Bayesian model comparison)

- As we saw previously, the classic Bayesian method of determining the best model for a dataset is calculating the ratio of the evidences for each model.
- It turns out a simple calculation of the evidence using the the MCMC chain and the likelihood at the chain points is unstable
- Instead a better solution turns out to be a trick from statistical mechanics; the method is known as thermodynamic integration
  - See https://eriqande.github.io/sisg_mcmc_course/thermodynamic-integration.nb.html for details
- Alternatively, a class of MCMC-like algorithms known as nested sampling algorithms can do the necessary integrals stably

An alternative to MCMC: Nested sampling

- Nested sampling algorithms produce a set of contours of constant likelihood at many different likelihood levels, as well as a chain of points that can be reweighted to match the posterior
- Iteratively throw out lowest-likelihood point and replace it with a higher-likelihood one to explore the posterior
- Can calculate the evidence integral from the volume within contours of different likelihood levels
- E.g.: Multinest: https://academic.oup.com/mnras/article/398/4/1601/981502 or dynesty: https://dynesty.readthedocs.io/en/latest/index.html

Cartoon illustrating (a) the posterior of a two-dimensional problem and (b) the transformed $\mathcal{L}(X)$ function where the prior volumes X_i are associated with each likelihood $\mathcal{L}_i$.

Another alternative method for MCMC: Hamiltonian Monte Carlo

- Hamiltonian Monte Carlo basically follows the Metropolis-Hastings algorithm, but uses a VERY different strategy to choose next step
- Instead of drawing random offset in parameters, draw a random momentum from multi-dimensional Gaussian
- Evolve from initial position with that momentum via Hamiltonian mechanics, with potential energy given by $V = -\ln \pi(\theta | x)$, where π is the posterior PDF, θ is the parameter vector, and **x** is the data we got.

Animation: Justinkunimune, wikipedia

Another alternative method for MCMC: Hamiltonian Monte Carlo

- Since the Hamiltonian $H = T + V$ does not change with time, V can't change too much along trajectory: so will tend to stay in regions where posterior is high
  - Need to choose how many time steps to evolve trajectory for (No-U-turn HMC methods do that dynamically)
- At the end of a trajectory, evaluate whether to make the move based on relative posterior values, as in Metropolis-Hastings

Animation: Justinkunimune, wikipedia

Another alternative method for MCMC: Hamiltonian Monte Carlo

- Trajectory calculation depends on derivatives of the Hamiltonian
- HMC can take advantage of fast GPU-based algorithms for automatic differentiation
  - ... which were key for developing deep learning methods
- For more, see A Conceptual Introduction to Hamiltonian Monte Carlo by Michael Betancourt
- Also see https://chi-feng.github.io/mcmc-demo/ for interactive demos of different MCMC methods

Animation: Justinkunimune, wikipedia

Some useful resources for MCMC

- *Bayesian inference and computation: a beginner's guide*: https://odysee.com/@BrendonBrewer:3/wsbook:f
  - This is a good general introduction to the ideas behind MCMC and nested samplers
- http://mattpitkin.github.io/samplers-demo/pages/samplers-samplers-everywhere/
  - This includes some examples for nested samplers, which are capable of calculating the Bayesian evidence correctly
- http://bjlkeng.github.io/posts/markov-chain-monte-carlo-mcmc-and-the-metropolis-hastings-algorithm/
- https://arxiv.org/abs/1411.5018 (Frequentism and Bayesianism: A Python-driven Primer by Vanderplas)
- https://ixkael.github.io/fitting-a-line-to-data-a-quick-tutorial/
  - includes examples of fitting lines with errors on X + Y via MCMC

A very brief introduction to neural networks

A note before we begin

- The slides that follow are substantially indebted to the book *Understanding Deep Learning* (available at https://udlbook.github.io/udlbook/) and course materials provided by the author, Simon Prince (Bath University)
- That site has lots of notebooks available if you want to play with basic deep learning methods
  - Simplified artificial datasets are now available that allow testing deep learning methods at small scale; e.g. MNIST-1D

Basic applications of deep learning methods

Neural network vs. linear regression

Simple linear regression

$$
\begin{aligned}
y &= \mathrm{f}[x, \boldsymbol{\phi}] \\
&= \phi_0 + \phi_1 x
\end{aligned}
$$

Example shallow neural network

$$
\begin{aligned}
y &= \mathrm{f}[x, \boldsymbol{\phi}] \\
&= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x]
\end{aligned}
$$

Example shallow network

Activation function

$$y = \mathrm{f}[x, \boldsymbol{\phi}]$$
$$= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x]$$

Rectified Linear Unit (ReLU):

most commonly-used activation function

$$\mathrm{a}[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}.$$

Other activation functions can be used, but ReLU is easy/quick to differentiate and is never too shallow

Example of shallow network outputs

$$y = \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x].$$

This constructs piecewise linear functions with three joints

Training the network

$$y = \mathrm{f}[x, \boldsymbol{\phi}]$$
$$= \phi_0 + \phi_1 \mathrm{a}[\theta_{10} + \theta_{11} x] + \phi_2 \mathrm{a}[\theta_{20} + \theta_{21} x] + \phi_3 \mathrm{a}[\theta_{30} + \theta_{31} x]$$

- This model has 10 parameters: $\boldsymbol{\phi} = \{\phi_0, \phi_1, \phi_2, \phi_3, \theta_{10}, \theta_{11}, \theta_{20}, \theta_{21}, \theta_{30}, \theta_{31}\}$
  - Represents a family of functions
- Given parameter values, it can make predictions
- Given a training dataset: $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{I}$ ...
  - and a loss function $L[\Phi]$ (e.g., least squares) ...
  - we can then change the parameters to minimize the loss function for the training set
- This is analogous to training any other machine learning model!

1. Compute three linear functions

2. Pass through ReLU functions (creates hidden units)

3. Weight the hidden units

4. Sum the weighted hidden units to create output

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

Depicting neural networks

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

Each parameter multiplies its source and adds to its target

Often the constants are left out of the graph

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x] \qquad y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$

With enough hidden units…

… we can describe any function to arbitrary accuracy, even for multiple dimensions (this is a provable theorem!)

Nomenclature

- Y-offsets = ***biases***
- Slopes = ***weights***
- Everything in one layer connected to everything in the next = ***fully-connected network***
- No loops = f***eedforward network***
- Values after ReLU (activation functions) = ***activations***
- Values before ReLU = ***pre-activations***
- One hidden layer = ***shallow neural network***
- More than one hidden layer = ***deep neural network***

Two-layer network

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$

$$h'_1 = \mathrm{a}[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3]$$
$$h'_2 = \mathrm{a}[\psi_{20} + \psi_{21}h_2 + \psi_{22}h_2 + \psi_{23}h_3]$$
$$h'_3 = \mathrm{a}[\psi_{30} + \psi_{31}h_2 + \psi_{32}h_2 + \psi_{33}h_3]$$

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$

- A network that is fully-connected, feed-forward, and has at least one hidden layer is sometimes called a ***Multi-layered perceptron***: basically the simplest deep architecture

Improvements in optimizers help minimize loss more efficiently

Stochastic gradient descent methods compute gradients of loss with different subsets of training data at each step

A "learning rate" parameter (here, α) controls how big the jumps taken will be

Adaptive moment estimation (Adam) algorithm is most commonly used

- *Adam* optimizer is current standard
- Normalizes gradients to avoid oscillatory behavior
- uses 'momentum' from averaging recent gradients to guide trajectory
- Often start with large learning rate and reduce over time
- https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c/ provides an interactive tool for trying different optimizers

Convolutional neural networks greatly improved performance for image-based tasks

- Example convolutions with kernel size 3
- Convolution kernel is the same for all input data, strides across input space
- Weight values are free parameters in the network

2D Convolution

2D Convolution

Often some strategy is used to downsample the outputs of the convolutional layer to smaller # of dimensions

Sample every other position (equivalent to stride two)

Max pooling (partial invariance to translation)

Mean pooling

ImageNet database is standard testing ground for image-based tasks

- 224 x 224 images
- 1,281,167 training images, 50,000 validation images, and 100,000 test images
- 1000 classes

CNN design has advanced rapidly. AlexNet (2012):

Almost all the 60 million parameters parameters are in fully connected layers

Resnet 200 (2016)

Performance has continued to improve with newer designs (e.g., ConvNext)

Deep learning models can continue to improve loss even when # of parameters >> amount of training data

Classical or under-parameterized regime

Critical regime

Modern or over-parameterized regime

Decoder model: e.g., GPT3

Decoder model: e.g., GPT3

- One job: predict the next word in a sequence
- More formally: builds an autoregressive probability model:

$$Pr(t_1, t_2, \ldots t_N) = Pr(t_1) \prod_{n=2}^{N} Pr(t_n | t_1 \ldots t_{n-1})$$

Decoder model: e.g., GPT3

- Example of autoregressive probability model: probability of "It takes great courage to let yourself appear weak" is:

$$
\begin{aligned}
&Pr(\text{It takes great personal courage to let yourself appear weak}) = \\
&Pr(\text{It}) \times Pr(\text{takes}|\text{It}) \times Pr(\text{great}|\text{It takes}) \times Pr(\text{courage}|\text{It takes great}) \times \\
&Pr(\text{to}|\text{It takes great courage}) \times Pr(\text{let}|\text{It takes great courage to}) \times \\
&Pr(\text{yourself}|\text{It takes great courage to let}) \times \\
&Pr(\text{appear}|\text{It takes great courage to let yourself}) \times \\
&Pr(\text{weak}|\text{It takes great courage to let yourself appear}).
\end{aligned}
$$

Loss is evaluated based on how well the network predicts the next word

Large Language Models are getting even larger...

Figures: Etienne Bernard

- Not much use for physics yet
- Incorrect 50% of the time in many applications, "hallucinate" fake references, etc.
- Write code importing libraries that didn't exist... that people then have made malware under the same names
- User beware!