Spatio-Temporal Modeling and Simulation

Marcel Gietzmann-Sanders

2024

Dr. Andrew Seitz Dr. Curry Cunningham

Michael Courtney, M.S.

College of Fisheries and Ocean Sciences
University of Alaska Fairbanks

Contents

1 Log Odds Modeling

Objective 1. *To provide tooling that allows for using machine learning methods to fit models of the form*

$$\psi_k = G(\eta_k)$$

that maximize the following objective:

$$\mathcal{L} = \prod_i P'(v_i|\eta_i)$$

where:

$$P'(v_i|\eta_i) = \frac{e^{\psi_i}}{\sum_k e^{\psi_k}}$$

These models will be known as log odds models as they predict the "log odds for" each outcome v_k given the information contained in η_k.

1.1 Fitting an Odds Model

Our objective in using machine learning (ML) is to reduce the time spent looking for the particular form of $G(\eta_k)$ by instead allowing that form to be fit given the data at hand. Deep Learning (DL) models are especially well suited to this problem as they are both parametric (see Section 3.1 for why non-parametric models are not well suited to our problem) and have been shown to be able to represent just about any function given a large enough network [Benoit Liquet and Nazarathy, 2024]. There is also a robust field around probabilistic deep learning that, as we shall see in a moment, we can take advantage of [Oliver Durr, 2020].

Taking a look at traditional probabilistic classifiers we can see that they have two things in common with our intended log odds modeling.

First, probabilistic classifiers use the categorical cross entropy cost function to optimize their weights. This cost function happens to be just another name for optimizing for the Log Likelihood of the data [Oliver Durr, 2020]. Therefore probabilistic DL shares the same objective function as we do.

Second, the final layer in a probabilistic DL network is the one that produces a probability for each class. This layer uses the softmax function as it's activation function which if if $a_i = W_i x + b_i$ is given by - $\frac{e^{a_i}}{\sum_k e^{a_k}}$. This means that if we can get our ψ_k to be the a_i that we'll be, in effect, training a log odds model.

We can do this by taking advantage of Keras' Functional API [Keras, 2023] which allows us to split our network into a branches and also share layers between branches. Specifically we can follow the following steps:

1. Define a series of layers that represent our underlying log odds model (these layers should end in a layer of output size 1 which will represent our ψ_k).

2. Split our training decisions into N choices.

3. Create a branch in our overall model for each of these N choices.

4. Converge the branches at a softmax layer where the weights are the identity matrix I and the biases all 0 (this ensures we directly pass through our ψ_k to the softmax function).

5. Use categorical cross entropy as our fitness function.

This architecture is illustrated by Fig. 1.

The main advantage of this architecture as opposed to a more classic approach to probabilistic DL is the fact that the weights are shared on each of the branches. This has the effect of drastically reducing the number of parameters in our network meaning that we can use far less data to train this model then we'd need to train a full bore, fully connected, probabilistic model. Each of our columns in Fig. 1 is in fact the same model and therefore updates coming from each of the columns goes to all of the columns.

This, then, solves one of our issues - reducing the variance in the model by switching to the log odds approach. But the other issue we're trying to solve is being able to predict on variable numbers of choices per decision. How does that work out here?

Well, in the case of training we know the maximum number of choices in any of our training decisions. Therefore we can set N to this maximum and for decisions where we have fewer than N choices we can simply provide some kind of default "do not choose" feature for the "missing" choices.

Later during inference on new data we can simply take our shared layers and use them to predict on each of the choices we're presented with. Just another demonstration that while we're using the architecture in Fig. 1 in order to *train* the log odds model, the actual log odds model is simply the shared layers in a single column behind the softmax layer.

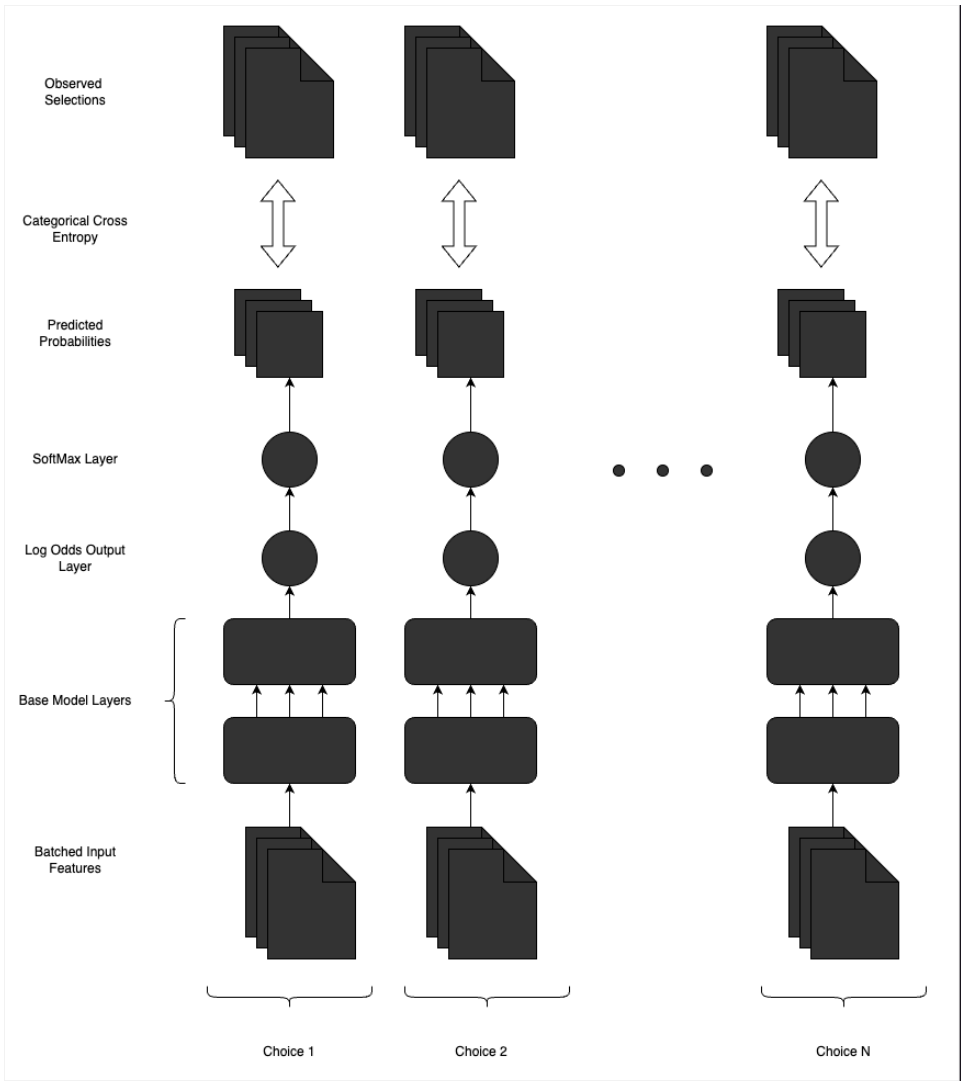

Figure 1: Overall Architecture

2 Simulation

2.1 Dealing with Scale

2.1.1 Dealing with Many-World

We're dealing with states in, states out, with associated relative abundances. Therefore a group by at the end keeps the many world constrained instead of having some D^n problem.

2.1.2 Parallelism

Because each state determines it's own outcomes we can parallelize things dramatically. As the number of states we're working with gets bigger (as we add more detail to the sim) this will become more and more important. Furthermore it means we can deal with our simulation as a single step that gets repeated on itself. We'll use pyspark to make sure that when the parallelization is needed it is close at hand.

2.1.3 Exploring Data

There's a ridiculous amount of data that comes out of this and only with a DB can we have any hope of exploring it. The data can be stored away, what we want can be queried, and then that can be uploaded into standard tools for viewing. This keeps the memory requirements of the exploration application to a minimum while still having all the data at one's fingertips.

3 Appendices

3.1 Convergence Issue with Non-Parametric Log Odds Models

All standard Machine Learning (ML) pipelines assume that you have at least two things - targets and features. In our case we certainly have the latter but our target ψ_k is both unknown to us and also unmeasureable. So how are we to fit ML models if we have no target? In short, through iteration. Let's see how this can be done.

First some notation to help us. Our data is composed of a series of decisions $D_j = \{v_{jk}\}$ where j indicates each of the specific decisions and k the options within each decision. For each iteration we will build a model $\hat{G}_i(\eta_{jk})$ using the pairs $\psi_{jk(i-1)}, \eta_{jk}$. We will designate the outcome of that model ϕ_{jki}:

$$\phi_{jki} = \hat{G}_i(\eta_{jk})$$

Our probability is therefore:

$$P'_i(v_{jk}|\eta_{jk}) = \frac{e^{\phi_{jki}}}{\sum_p e^{\phi_{jpi}}}$$

Now if the k selected per decision D_j is given by s_j we want to maximize:

$$\mathcal{L} = \prod_j P'_i(v_{js_j}|\eta_{js_j}) \to \sum_j \ln\left[P'_i(v_{js_j}|\eta_{js_j})\right]$$

Using this information we will then propose an update u_{jki} s.t. $\psi_{jki} = \phi_{jki} + u_{jki}$ and repeat our iteration loop.
With that notation cleared up we can begin our iteration procedure.

First, let's assume we already have a guess for $\psi_{jk(i-1)}$. We can therefore train our model off of the ψ_{jk0}, η_{jk} pairs in standard ML fashion. Specifically we will fit a model that optimizes Mean Squared Error (the most common objective across ML software packages):

$$\min\left[\sum_{jk}\left(\hat{G}_i(\eta_{jk}) - \psi_{jk(i-1)}\right)^2\right]$$

We now need to choose a set of updates u_{jki}. To get these we will turn to our overall objective function:

$$\sum_j \ln\left[P_i'(v_{js_j}|\eta_{js_j})\right] = \sum_j \ln\left[\frac{e^{\phi_{js_ji}}}{\sum_p e^{\phi_{jpi}}}\right]$$

Let's look at the gradient of this w.r.t the ϕ_{jki}. There are two cases:

$$\partial_{\phi_{js_ji}} \ln \mathcal{L} = \frac{1}{P_i'(v_{js_j}|\eta_{js_j})} \frac{\sum_p e^{\phi_{jpi}} - e^{\phi_{js_ji}}}{\left(\sum_p e^{\phi_{jpi}}\right)^2} e^{\phi_{js_ji}} = 1 - P_i'(v_{js_j}|\eta_{js_j})$$

$$\partial_{\phi_{j\not{s}_ji}} \ln \mathcal{L} = \frac{1}{P_i'(v_{js_j}|\eta_{js_j})} \frac{-e^{\phi_{js_ji}}}{\left(\sum_p e^{\phi_{jpi}}\right)^2} e^{\phi_{j\not{s}_ji}} = -P_i'(v_{j\not{s}_j}|\eta_{j\not{s}_j})$$

Next for point of illustration let's suppose that there are a set of ψ_{jki} which we'll designate as Z which share the same features η, i.e. our model has to give a single ϕ_{jki} for all such options. Our derivative then for that collection Z is given by:

$$\partial_Z \ln \mathcal{L} = \sum_{\phi_{js_ji} \in Z} \left(1 - P_i'(v_{js_j}|\eta_{js_j})\right) - \sum_{\phi_{j\not{s}_ji} \in Z} P_i'(v_{j\not{s}_j}|\eta_{j\not{s}_j})$$

Given classic optimization tactics we know that our function will be maximized when these sums are 0 (and technically we'd also want to know that the second derivative was negative but we'll assume that's the case given how our iterations will work).

With this in mind let's now propose that our updates are given by:

$$u_{jki} = \alpha_i \partial_{\phi_{jki}} \ln \mathcal{L}$$

where α_i is a constant we'll call our "learning rate". Note that by using this update we will increase our ψ_{jki} guess where it corresponds to a taken option (s_j) and will decrease it where it corresponds to an option not taken ($\not{s}_j$). This will therefore push us towards maximizing $\ln \mathcal{L}$ as opposed to minimizing it.

We are left with a final question - will our iteration sequence end when we've found the maximizing guesses of ψ_{jki}? To answer this we turn back to the term we are maximizing when fitting the $\hat{G}_i$:

$$\min \left[\sum_{jk} \left(\hat{G}_i(\eta_{jk}) - \psi_{jk(i-1)} \right)^2 \right]$$

Our new fit will look like:

$$\sum_{jk} \left(\hat{G}_{i+1}(\eta_{jk}) - (\hat{G}_i(\eta_{jk}) + u_{jki}) \right)^2 = \sum_{jk} \left(\delta\hat{G}_{jk} - u_{jki} \right)^2$$

Given our Z once again we can take the derivative w.r.t $\delta\hat{G}_{jk}$ where $\phi_{jki} \in Z$.

$$\partial_Z \left[\sum_{jk} \left(\delta\hat{G} - u_{jki} \right)^2 \right] = \sum_{\phi_{jki} \in Z} 2 \left(\delta\hat{G} - u_{jki} \right) = 2 \sum_{\phi_{jki} \in Z} \delta\hat{G} - 2 \sum_{\phi_{jki} \in Z} u_{jki}$$

But now remember that if we've maximized w.r.t Z that:

$$\sum_{\phi_{jki} \in Z} u_{jki} = 0$$

which means that in order for our partial derivative above to be zero (and therefore our error term be at a minimum) that $\delta\hat{G} = 0$. And this means our iteration will have stopped!

Procedure 1. *Fitting a Log Odds Model*

1. *Collect decisions D_j and corresponding features η_{jk}, options v_{jk}, and selection s_j.*
2. *Make an initial guess $\psi_{jk0} = 0$.*
3. *Fit $\hat{G}_i$ on the pairs of $\psi_{jk(i-1)}, \eta_{jk}$ using MSE to produce the ϕ_{jki}.*
4. *Generate the $u_{jki} = \alpha_i \partial_{\phi_{jki}} \ln \mathcal{L}$ and produce a new set of ψ_{jki}.*
5. *Repeat 3 and 4 until, varying α_i until convergence.*

$$\partial_{\phi_{js_ji}} \ln \mathcal{L} = 1 - P_i'(v_{js_j}|\eta_{js_j})$$

$$\partial_{\phi_{j\not{s}_ji}} \ln \mathcal{L} = -P_i'(v_{j\not{s}_j}|\eta_{j\not{s}_j})$$

There is however an issue with this approach. Recall that our gradient is:

$$\partial_Z \ln \mathcal{L} = \sum_{\phi_{js_ji} \in Z} \big(1 - P_i'(v_{js_j}|\eta_{js_j})\big) - \sum_{\phi_{j\not{s}_ji} \in Z} P_i'(v_{j\not{s}_j}|\eta_{j\not{s}_j})$$

Let's however look at the second derivative of $\ln \mathcal{L}$:

$$\partial_{\phi_{jk}} P_i'(v_{jk}|\eta_{jk}) = \partial_{\phi_{jk}} \frac{e^{\phi_{jk}}}{\sum_p e^{\phi_{jp}}} = (1 - P_i'(v_{jk}|\eta_{jk}))P_i'(v_{jk}|\eta_{jk})$$

$$\partial_Z^2 \ln \mathcal{L} = -\sum_{\phi_{js_ji} \in Z} \big(1 - P_i'(v_{js_j}|\eta_{js_j})\big) P_i'(v_{js_j}|\eta_{js_j}) - \sum_{\phi_{j\not{s}_ji} \in Z} \big(1 - P_i'(v_{j\not{s}_j}|\eta_{j\not{s}_j})\big) P_i'(v_{j\not{s}_j}|\eta_{j\not{s}_j})$$

What's important to note is that this function will be near 0 at $P \approx 1$ or $P \approx 0$ and will have its largest magnitude near $P \approx 0.5$. Why is this an issue? Well we know that as we get close to our maximum value for $\ln \mathcal{L}$ that our steps towards that maximum will get smaller and smaller. This is just because our steps are based on a derivative and we are seeking where the derivative is zero. Put another way getting to our max gets harder the closer we get to that maximum.

Now normally one deals with this by using the second derivative (the curvature) as a kind of correction. As you get closer to your maximum, you can use your curvature to guide how quickly you can move. If the curvature is very small in magnitude you can move more quickly because the odds of you overshooting your maximum are smaller. If your curvature is very large you'll slow things down because you know only small steps are required to make big changes to the derivative (the thing we are ultimately trying to set to a specific value here). I.e. you can modulate your step size by the inverse of the curvature.

However, in our case we don't actually know what the curvature is because we don't know what Z is! Therefore we have to plan for the most volatile case - the case where $P \approx 0.5$. And that means that for any of our Z's where P is approaching 1 or 0 we'll be moving at a snail's pace (convergence will take forever).

Now in case you're wondering whether we could solve the $P \approx 0.5$ case first and then move onto the others remember that anytime we change any of the ϕ all the other probabilities change. So we have to solve this problem all at once. Because of that and because we cannot depend on knowing the Z (different Z can give the same value ϕ) we're stuffed with having to take extraordinarily long convergence times. And this more or less means we've got no shot of using this in practice.

Log odds modeling doesn't work in practice for non-parametric models.

4 Bibliography

References

[Benoit Liquet and Nazarathy, 2024] Benoit Liquet, S. M. and Nazarathy, Y. (2024). *Mathematical Engineering of Deep Learning.* CRC Press.

[Keras, 2023] Keras (2023). Keras functional api.

[Oliver Durr, 2020] Oliver Durr, Beate Sick, E. M. (2020). *Probabilistic Deep Learning.* Manning Publications.