A Functional Embedding of Terms in a Spatial Hierarchy

Alex Grabanski[*]

May 10, 2021

Abstract

Motivated by the problem of program induction (learning of a computer program from data), we describe an incremental inference routine to derive embeddings of terms in a simply-typed combinatory calculus as probability distributions over finite-dimensional vector spaces. Our method assigns a space of embeddings to every type of the underlying language, and assigns an embedding to every term which has been evaluated by the interpreter. As the interpreter is issued commands from a down-stream task, these term embeddings are kept updated in such a way that they reflect the best-available knowledge about term behavior. To do so, we propose novel distributional inference routines for random-feature kernel regresssors situated in a hierarchy of typed functions. Finally, we discuss ways that supervised, cumulative, multi-task, and reinforcement learning could potentially benefit from the proposed framework.

1 Introduction

The problem of *inductive reasoning*, or reasoning about hypotheses from data, is the central problem of machine learning. While many modern machine learning techniques typically restrict the collection of hypotheses to well-defined and manageable classes, in the spirit of radicalism, we choose instead to jump back to the 1964 theory Ray Solomonoff advanced about induction [15], which was fundamentally a process of induction over *computer programs.* In some sense, this is the most general setting for the problem of induction in machine learning, since the hypothesis class of computer programs *is* the most general class of hypotheses expressible by machine. Unfortunately, induction over the space of all programs is not computable. In fact, this is incredibly unfortunate, since under the definition of "artificial general intelligence" ("AGI") adopted by AIXI [5], we only need optimal inductive inference to achieve provably-optimal AGI. While approximations to the conceptual process of Bayesian inference over all computer programs have been previously advanced, such as the one in AIXI-tl [5], they have exhibited very poor performance to date.

Motivated by this problem, and by the utility of embeddings for dealing with otherwise-unwieldy objects such as natural-language words in machine learning, we devise a method to embed the terms of a simply-typed combinatory calculus into separate spaces for each type. However, instead of simple vector embeddings, we embed functional terms as *probability distributions* over vectors in the embedding space. In this way, we hope to provide an immensely practical Bayesian inference method over program terms which is not only computable, but also has a very intuitive spatial structure which is readily consumed by down-stream tasks.

2 Prior Research

While what we will describe in this paper does not involve any kind of process for program search, since we leave that open to future developments, the closest area we can draw a reference point to is that of *program synthesis.* A good high-level survey of this area is given in ([4]). In particular, the sub-field of *neural program synthesis* has some relatively recent works [12] [10] which focus on the compositionality of programs and program embeddings using neural-network based encoders as part of their gradient-based optimization pipeline to derive programs from examples. However, unlike neural-network-based frameworks such as AlphaNPI [12], our framework for deriving embeddings is *not* based on interpretation in an imperative language, but instead a typed combinatory calculus,

[*]e-mail: ajg137@case.edu

which we would expect to have better compositional properties. In addition, unlike the aforementioned neural program synthesis approaches, we do not require any gradients in our derivation of embeddings, which means that we are free to use an actual interpreter instead of a differentiable surrogate.

Even closer, very recently, ([1]) proposed a neural-network-based framework for gradually growing and abstracting a library of functional programs in a multi-task learning environment. However, unlike the approach here, they do not at any point generate embeddings for program terms, but instead, a discriminative neural-network model on programs which encodes how likely they are to solve the task at hand.

Another research area which is tangentially related is that of *neural architecture search*, a survey of which is provided in ([16]). In neural architecture search, building-blocks of neural networks are composed to attempt to achieve optimal architecture, which is quite similar to the task of composing a program from combinators, with the caveat that neural architecture search is intended to run before a full optimization over all parameters in the network. While this research area is largely divergent from what we consider here, we do note that [7] did consider a Bayesian Optimization framework for neural architecture search which leveraged Gaussian Processes, which we also do here. However, the internal representation in their framework differs radically from the one we are about to present, as their representation is not compositional in nature.

3 Types, Terms, and Interpreter

We now describe the simple combinatory source language which we will derive term embeddings for.

3.1 Types

The combinatory language we consider only has two broad categories of types: vector types, and function types.

3.1.1 Vector Types

We suppose that there is some finite set $\mathcal{V}$ of *vector types*, and an associated mapping $dim : \mathcal{V} \to \mathbb{N}$ from vector types to their *dimensionality*. Every vector type $V \in \mathcal{V}$ with $dim(V) = n$ conceptually may be taken to correspond to some labeled, not-necessarily-unique copy of $\mathbb{R}^n$.

While the restriction of primitive types to finite-dimensional vector spaces rules out a large collection of sequence types (e.g: Strings, arbitrary-length Lists), it notably allows representation of many of the data formats which are found commonly in ML literature (e.g: spatial vectors, images, voxel bitmaps, word embeddings, etc.)

3.1.2 Function Types

In order to represent the types of mappings between certain types, we also suppose that there is some finite set $\mathcal{F}$ of *function types*, with an associated one-to-one mapping $sig : \mathcal{F} \to \mathcal{U} \times \mathcal{U}$, where $\mathcal{U} = \mathcal{F} \cup \mathcal{V}$ is the *universe* of types, such that the directed graph with nodes in $\mathcal{U}$ and edges for every $F \in \mathcal{F}$ from F to $\pi_1(sig(F))$ and from F to $\pi_2(sig(F))$ has no cycles. If $sig(F) = (A, B)$, we call A the *domain* of F, and B the *codomain*, and in light of the injectivity of sig, we will frequently write $F = A \to B$ for the function type F.

While the requirement that the set of function types is a finite set may seem restrictive at first, it's important to note that in actual practice, human programmers rarely use functions with high arity nor higher-order functions with overly-complex functional arguments. In practice, this means that in any given language, we could restrict the nesting of the function type constructor to a sufficiently large finite value and lose absolutely nothing of practical import. Since our language has only a finite number of primitive types, such a nesting restriction would be sufficient to guarantee that $\mathcal{F}$ is a finite set.

3.2 Terms

The terms of our simple combinatory language largely correspond to the intuitive descriptions of types which we have given previously. We employ the common notation $t : T$ for denoting that the term t belongs to the type T.

3.2.1 Vector Terms

For every vector type $V \in \mathcal{V}$, and every $v \in \mathbb{R}^n$, where $n = dim(V)$, $v_V : V$.

3.2.2 Primitive Function Terms

For every function type $F \in \mathcal{F}$ for which $F = A_1 \to (A_2 \to (...A_k \to B)...)$, we suppose that there is some finite set $\mathcal{P}_F$ of *primitive function terms* for which every $f \in \mathcal{P}_F$ is such that

$f : F$, and for each such f, there is a corresponding effective procedure which maps tuples of terms in $A_1 \times A_2 \times ...A_k$ to terms of B within finite time after each time the procedure is invoked. We do not demand that the corresponding procedure for f is a function in the mathematical sense, as we intend to explicitly allow mappings which incorporate external sources of randomness. In this situation, we say that f is a primitive with arity k. When considering particular instances of invocations of f on tuples of terms in $A_1 \times A_2 \times ...A_k$, we will write $f(a_1, a_2, ...a_k) \rightsquigarrow b$ for an invocation of the procedure for f on $a_1 : A_1$, $a_2 : A_2$... $a_k : A_k$ which yielded $b : B$.

3.2.3 Partially-Applied Function Terms

In addition to primitive function terms, for every primitive function term f as specified previously, every $0 < n < k$ and every collection of terms $a_1 : A_1$, $a_2 : A_2$... $a_n : A_n$, we allow for a *partially-applied* function term, denoted $\mathit{PartiallyApplied}(f, a_1, ...a_n) : A_{n+1} \to (A_{n+2} \to (...A_k \to B)...)$.

3.3 Evaluation Mapping

For every function term $g : G = X \to B$, and every $x : X$, we define the *evaluation mapping* $\mathit{eval}(g, x)$ by:

- Case: g is a primitive function term
  $\mapsto \mathit{PartiallyApplied}(g, x)$.
- Case: g is $\mathit{PartiallyApplied}(f, a_1, ...a_n)$
  for arity-$k > n + 1$ primitive function term f
  $\mapsto \mathit{PartiallyApplied}(g, a_1, ...a_n, x)$
- Case: g is $\mathit{PartiallyApplied}(f, a_1, ...a_n)$
  for arity-$k = n + 1$ primitive function term f
  $\mapsto b$ in some freshly-obtained $f(a_1, a_2, ...a_n, x) \rightsquigarrow b$.

3.4 Interpreter

With the definitions of the constructs of the language given, we can now describe the operation of the interpreter and its interface, as exposed to downstream tasks.

In our interpreter, the unique terms of functional types will be assigned their own *address* for long-term storage. Upon initialization, only the primitive function terms will be assigned addresses, but new addresses will be incrementally assigned as new partially-applied terms are derived from evaluations. Vector terms, in contrast, are never assigned addresses, and simply exist inline as arrays of floating-point values. To be able to refer to both situations under the same framework, we say that a *term reference* is either the address of a function term, or a bare array of floating-point values representing a vector term.

We only expose a single mechanism to interact with the interpreter in downstream tasks: Evaluating a function $f : X \to Y$ corresponding to a given address on an argument $x : X$ for a given term reference, and returning a resulting term reference to $f(x)$ to the downstream task. In addition, as a side-effect of doing this, if $f(x)$ is a function term which does not already have an address, we allocate space for it and assign it to a new address. During the operation of the interpreter, we also keep track of the result of every evaluation for later use in the embedding-derivation process.

Although this mechanism for interacting with the interpreter is extremely minimal, it still allows the evaluation of arbitrarily-nested applications of terms to terms, since the code invoking the interpreter could take an arbitrary expression tree and evaluate it using the strict evaluation order, with intermediate expressions always represented by their term references.

4 Term Embeddings

We are now ready to discuss the embeddings associated with each type of the language. Recall that an embedding is a mapping from objects of some class to a finite vector space, such that similar objects map to vectors which are close in norm. The notion of similarity we intend to adopt here is one of *operational similarity* – namely, supposing that all functions in the language are suitably continuous, we can declare two terms of the same type as being similar if most expressions involving the two terms evaluate to similar results. In the particular example of vector types, to satisfy this demand, we could simply choose any arbitrary continuous map from the dimensionality of the vector type to some target dimension to be our embedding map, including the possibility of simply choosing the identity mapping, as we do in the sequel.

However, a mechanism for deriving embeddings of function terms is far less obvious. We can naturally suppose that we want function embeddings to be derived from mappings from input embeddings to output embeddings, but function spaces on real vector spaces are inherently infinite-dimensional. We want the embeddings of functions to be finite-dimensional. To get around this issue, we *could* try to use the fashionable approach of neural networks, and pick the embeddings of functions to be the parameters of their corresponding neural networks, assuming that we fix some network topology. However, this approach will not result in true *embeddings*, since it's common in practice to see neural networks trained on the same data which nevertheless wind up with radically differing parameters. Ideally, the representation we adopt for functions should have a single unique best-fit embedding for a given collection of observed data-points.

We can achieve these desiredata straightforwardly by adopting linear regression on feature-mapped input, output pairs as the mechanism for deriving our function embeddings. By first non-linearly mapping inputs to a larger feature space before performing a linear regression, we can effectively approximate kernel regression using the well-known method of random features [11]. To this aim, our reference implementation supports sketched linear features, randomized quadratic features [9], and Random Fourier Features [11].

In spite of the elegance of this approach, we run into an immediate issue when considering embeddings of higher-order and curried multiple-argument function types. In the above proposal, if $f \in \mathbb{N}$ is the number of random features on the input embedding space, and $d \in \mathbb{N}$ is the dimensionality of the output embedding space, then the dimensionality of the associated function space will be $f \times d$. If the output space is in fact another function space, perhaps even a function space with a function-typed output, or if the number of input features is taken to be proportional to the dimensionality of the input embeddings, and we perform a similar nesting for the input space, we reach a regime where the dimensionality of the embedding space grows exponentially with the number of "$\rightarrow$" type-constructors in the type, which quickly leads to intractability.

To resolve the issue of exponentially-growing dimensionality of function embedding spaces with the number of "$\rightarrow$" type constructors, we first pose a hypothesis that the collection of functions of a given type that are of practical use do not span the full $f \times d$-dimensional space that they embed to, and in fact lie almost entirely within some smaller-dimensioned subspace. Our intuitive reasoning for why this could be expected to be the case is that typically, as the arity or the order of functions increase, the expressivity of the underlying operation also increases, meaning that far fewer such functions are needed in practical programming compared to their lower-order and lower-arity counterparts. If this hypothesis is true, then we should still be able to maintain a faithful representation of the embeddings of functions by passing them through a random projection of an appropriate target dimension, while substantially reducing the dimensionality of the derived embeddings.

In light of the previous observations, and the fact that real-world data-types in programming languages have substantial diversity in their representation and form, we proceed by establishing the following framework of associated spaces defined for each type of the language:

- *Base Space*: For a vector type V, the base space is simply $\mathbb{R}^d$, where $dim(V) = d$. For a function type $F = A \rightarrow B$, the base space is $\mathbb{R}^{f \times d}$, where $f = dimfeature(A)$ is the dimensionality of the feature space of A, and d is the dimensionality of the compressed space of B. The dimensionality of the base space of a type T is denoted $dim_{base}(T)$.

- *Compressed Space*: For a vector type V, the compressed space is identical to the base space. For a function type F, the compressed space is some space $\mathbb{R}^c$ where $c \leq dim_{base}(F) = d$. For any such function type F, we associate a (fixed, a priori) mapping called the *projection mapping* for the function type, which is some orthonormal linear map $\rho_F : \mathbb{R}^d \rightarrow \mathbb{R}^c$. The dimensionality of the compressed space of a type T is denoted $dim_{compressed}(T)$.

- *Feature Space*: For a type T, the feature space is a copy of $\mathbb{R}^f$, for $f = dim_{feature}(T)$ the number of features for T. For any such type T, we associate a (fixed, a priori) mapping on T's compressed space called the *feature mapping* for the type, which is an arbitrary, potentially non-linear mapping $\phi_T : \mathbb{R}^c \rightarrow \mathbb{R}^f$, where $c = dim_{compressed}(T)$.

Equipped with these definitions, we can feel more confident in the approach of performing featurize-mapped linear regression for deriving embeddings of function types. However, simply representing the embedding of a function by the maximum-likelihood ordinary regression model is insufficient for our purposes upon further reflection. In particular, the proposed model above takes on a recursive character in the case where $F = X \rightarrow Y$ and either X or Y are themselves function types. Since estimation of function embeddings is data-dependent, the accuracy of a fit at the level of F would implicitly depend on the accuracy of fits at the levels of X or Y. This motivates a consideration of the error in the best coefficients associated with the regression, and consequently, an extension to our previously-introduced notion of an embedding.

4.1 Schmears and Extended Embeddings

To approach the problem of representing uncertainty in our knowledge of function embeddings, we first introduce some definitions which will greatly simplify this task.

First, a *schmear* is our term for the representation of [limited information about] a probability distribution by its first two moments [mean and covariance]. In other words, an n-dimensional schmear is a pair (μ, Σ) where $\mu \in \mathbb{R}^n$ and Σ is a n-by-n positive semi-definite real matrix. We denote the set of all n-dimensional schmears by $\mathcal{S}^n$.

With the definition of "schmear" in hand, we can now extend our previous notion of an "embedding" to accurately model uncertainty. An *extended embedding* for the type X into $\mathbb{R}^n$ is a map:

$$\phi : X \to \mathcal{S}^n$$

which associates each term reference of that type to n-dimensional schmears. Moreover, while not a requirement of our definition, we would like our extended embeddings to send operationally-similar terms to schmears that are "close" to one another. In the case of vector types, just as before, we may very simply satisfy such a demand by simply mapping vectors v to schmears with zero covariance centered at v. Once again, for function terms, however, we will need to do substantially more work to represent the uncertainty in our knowledge of embeddings.

4.2 Extended Embeddings for Function Types

In line with our motivating sections above, we will use regression models to represent embeddings of function terms. However, in order to get the extended embeddings we desire, it is necessary for us to consider full-blown *Bayesian* multivariate regression, not just formulas for the maximum-likelihood regression coefficients.

Recall that for the multivariate linear regression model:

$$y = A * z + \epsilon$$

for y a t-dimensional vector and z a r-dimensional vector, where we assume that $\epsilon \sim_{iid} N(\mathbf{0}_{t\times t}, \mathbf{\Sigma}_\epsilon)$ for noise covariance matrix $\mathbf{\Sigma}_\epsilon$, we can put a conjugate *matrix-normal inverse-Wishart* prior over A and $\mathbf{\Sigma}_\epsilon$ [14], meaning that we suppose:

$$\mathbf{\Sigma}_\epsilon \sim \mathcal{IW}_t(V, v)$$

$$B|\mathbf{\Sigma}_\epsilon \sim \mathcal{MN}_{t,r}(B, \Lambda^+, \mathbf{\Sigma}_\epsilon)$$

Where $\mathcal{IW}_t(V, v)$ denotes the *inverse-Wishart* distribution with $t \times t$ scale matrix V and v degrees of freedom, and $\mathcal{MN}_{t,r}(B, \Lambda^+, \mathbf{\Sigma}_\epsilon)$ denotes the *matrix normal* distribution with $t \times r$ mean B, $r \times r$ input covariance Λ^+ (for precision Λ) [1] and $t \times t$ output covariance $\mathbf{\Sigma}_\epsilon$. We refer the reader to [2] or [13] for the probability density functions and some additional properties of these distributions. Here, we simply emphasize the fact that if

$$X \sim \mathcal{MN}_{t,r}(M, U, V)$$

then

$$vec(X) \sim \mathcal{N}_{t*r}(vec(M), V \otimes U)$$

where $vec(-)$ denotes vectorization of a matrix, $\mathcal{N}_n(-,-)$ denotes the multivariate normal distribution parameterized by its n-dimensional mean and $n \times n$ covariance, respectively, and $\otimes$ denotes the Kronecker product.

In what follows, we will use the shorthand $\mathcal{MNIW}_{t,r}(B, \Lambda, V, v)$ for the matrix-normal inverse-Wishart prior as described above. Since the mean of $\mathcal{IW}_t(V, v)$ is $\frac{V}{v-t-1}$ and the mean of the matrix-normal inverse-Wishart prior is independent of the noise covariance, by an application of the law of total covariance, we can note that if

$$X \sim \mathcal{MNIW}_{t,r}(B, \Lambda, V, v)$$

then

$$E[vec(X)] = B, \quad Cov[vec(X)] = \frac{V}{v-t-1} \otimes \Lambda^+$$

As a consequence, we can now define our extended embeddings for function terms. All we need to do is let $z = \psi_X(x)$ in the regression model above, and assume that our current knowledge of the regression parameters is matrix-normal

[1] Here and elsewhere in this paper, we opt to use the Moore-Penrose pseudoinverse instead of the matrix inverse, because we frequently translate between covariance and precision matrices which are sometimes of deficient rank since they often originate from sums of outer products in our inference routines.

inverse-Wishart distributed as above, which results in the extended embedding:

$$\phi(f) = (B_f, \frac{V_f}{v_f - t - 1} \otimes \Lambda_f^{+})$$

where $f : F = (X \to Y)$, and $(B_f, \Lambda_f, V_f, v_f)$ is the current parameter tuple for the $t \times r$ matrix-normal inverse-Wishart model placed on f.

5 Bayesian Updates

With our assumption of matrix-normal inverse-Wishart priors on our models of functions, our task is now to keep these models updated with the current best-available information as we evaluate new term applications in our interpreter.

5.1 Data Updates

As with any typical supervised regression model, we need to utilize information from the input/output pairings which are provided to us in order to determine the regression coefficients. However, unlike a typical regression model, the input/output pairings used in our case to update the function type $F = (X \to Y)$ are not in general input/output pairings of data points, but rather, of input/output pairings of *schmears*. As a result, we cannot directly employ the typical expressions for incremental updates to a multivariate linear regression.

However, we can *reduce* this problem of ours to a form which is expressible as weighted input/output data point updates. To accomplish this, we first greatly simplify matters by ignoring the input covariance entirely, so we are left with just a pairing of an observed input mean and the output schmear. Our reason for doing this is that in general, accounting for uncertainty in inputs in a Bayesian regression framework typically adds significant complexity [2]. On the other hand, uncertainty in outputs in a Bayesian regression framework is relatively simple to deal with.

After ignoring the spread of the input distribution, we may then proceed using a trick from the field of control theory to also eliminate the need to consider the output distribution in favor of instead considering a collection of points with the same mean and covariance. For a given schmear $(\mu, \Sigma) \in \mathcal{S}^n$, we can derive the *symmetric set of* $2n+1$ *sigma points* [6] as:

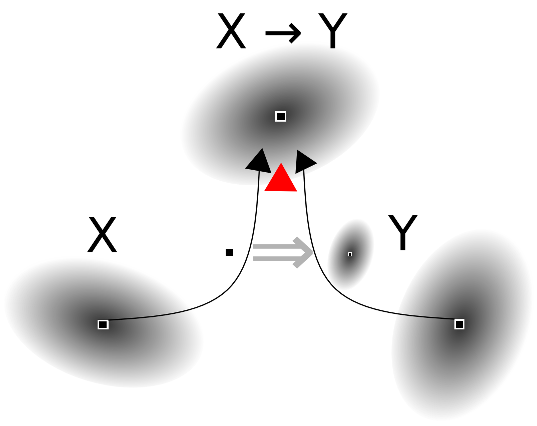


Figure 1: Schematic diagram for data updates, which utilize the mean of the input schmear and the entirety of the output schmear. In this and in the other diagrams of this paper, red triangles will be used to indicate which distribution the update(s) are being performed on, and black arrows will be used to indicate how information is propagated for the update(s).

$$\{\mu\} \cup \{\mu \pm [\sqrt{n\Sigma}]_i \quad | \quad i \in [0..n]\}$$

Where $[\sqrt{n\Sigma}]_i$ denotes the ith column of the unique symmetric matrix square root of $n\Sigma$. Straightforward calculation may be used to verify that this set of points has the same mean and covariance as the original schmear. Consequently, we can represent our data-point/schmear update as a collection of $2n+1$ data-point/data-point updates. In particular, if (μ_X, Σ_X) is our input schmear, and $s_Y = (\mu_Y, \Sigma_Y)$ is our output schmear, we perform $2n+1$ data-point updates, each with weight $\frac{1}{2n+1}$ using the input-output pairings $\{(\mu_X, \sigma_{Yi}) \quad | \quad i \in [0..2n+1]\}$, where σ_{Yi} is the ith sigma-point derived from s_Y.

Now that we have reduced the problem of keeping our models updated with respect to input/output schmears to the problem of keeping our models updated with respect to input/output data-points, we may simply use the usual expressions for the Bayesian update of the matrix-normal inverse-Wishart distribution $\mathcal{MNIW}(B, \Lambda, V, v)$ with respect to observation of a w-weighted input-output vector pairing (z, y).

$$\begin{aligned}
B &\Rightarrow (B\Lambda + w * yz^T)\Lambda_*^+ && = B_* \\
\Lambda &\Rightarrow \Lambda + w * zz^T && = \Lambda_* \\
v &\Rightarrow v + w \\
V &\Rightarrow V + (B - B_*)\Lambda(B - B_*)^T \\
&\quad + w * (\frac{yz^T}{z^T z} - B_*)zz^T(\frac{yz^T}{z^T z} - B_*)^T
\end{aligned}$$

Note that using the above expressions, we may also undo an update by simply replacing $w \Rightarrow -w$. In practice, we will not use the above formulas directly, but will instead opt to also track the value of Λ^+ across updates utilizing the Sherman-Morrison rank-1 inverse update formula for greater computational efficiency [8].

5.2 Prior Updates

While we are now relatively well-equipped, since the previously-described process is able to account for observed input/output pairings resulting from evaluation of a function application, we still have not exploited all information available to us. To see this, we will consider a worked example involving partial application of a binary function.

Suppose that we have a function term $f : X \to (Y \to Z)$ and two terms $x_1, x_2 : X$, and we have already directed the interpreter to evaluate $f(x_1) = f_1$ and $f(x_2) = f_2$. Furthermore, suppose that we have a large collection of terms $y_1, y_2, ...y_n : Y$ for which we have evaluated $f_1(y_i)$ for each i, but we have not yet evaluated f_2 on anything. In this situation, if we only use the above update process, the embedding of f_1 will be close to the maximum-likelihood regression model for the pairings $\{(y_i, f_1(y_i)|i \in [1, ...n]\}$. On the other hand, the embedding for f_2 will be entirely determined by the prior we adopt over regression coefficients. From these observations, we can immediately notice a conflict: If we take the limit as the embeddings of x_1 and x_2 approach each other, there's a sharp discontinuity in the behavior we purport to observe for f on embeddings. Fundamentally, this discontinuity originates from the fact that the estimates of the embeddings for f_1 and f_2 are not linked, despite the fact that both stem from partial evaluation of f.

To resolve this problem at a conceptual level, we can note that our observations of f_1 and f_2 will also result in a data-based update to the embedding of f. In the situation above, if we could only propagate the information we have gained about f to f_2 (ignoring the information stemming from the embedding of f_2 itself), we could encode the intuition that f_2 should be "close" to f_1.

We follow exactly this intuition in proposing another update routine which we call a *prior update*, since it in some sense updates the prior knowledge we apply to partially-evaluated functions. Moving beyond the particular example above, consider the general scenario where $f : X \to Y$ where $Y = (Z \to W)$ and $x : X$, $g = f(x) : (Z \to W)$. We want to use the embeddings of f and x to impute an update to the embedding of g. Conceptually, we break down this routine into a series of steps: First, we impute a schmear in the embedding space of $(Z \to W)$ which represents our best current estimate of the schmear for the application of f applied to x, where we modify our information about f to ignore any data update applied to it stemming from x. Then, once we have obtained our best-estimate schmear in the embedding space of $(Z \to W)$, we translate this schmear into a Bayesian update to perform on the matrix-normal inverse-Wishart model for g.

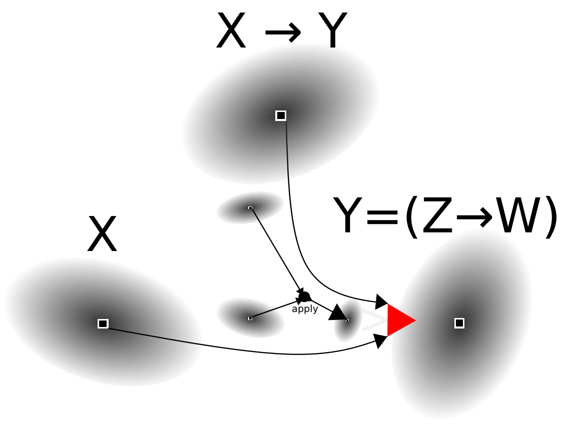


Figure 2: Schematic diagram for prior updates. The embedded schmear for the function and the embedded schmear for an argument are applied to one another to impute an output schmear, which is then translated into an update for the model of the output term.

5.2.1 Imputing the Output Schmear

For the moment, consider the straightforward linear model $\tilde{A}\tilde{z}+\epsilon = \tilde{y}$ where $\tilde{A}$ is $t \times r$ and further suppose that we employ a $\mathcal{MNIW}_{t,r}(\tilde{B}, \tilde{\Lambda}, \tilde{V}, \tilde{v})$ prior over the usual parameters and suppose that $E[\tilde{z}] = \mu$ and $Cov[\tilde{z}] = \Sigma$. Then,

$$E[\tilde{y}] = \tilde{B}\mu$$

$$Cov[\tilde{y}] = [\langle \Sigma, \tilde{\Lambda}^{+} \rangle_F + \mu^T \tilde{\Lambda}^{+} \mu] \frac{\tilde{V}}{\tilde{v} - t - 1} + \tilde{B}\Sigma\tilde{B}^T$$

where $\langle -, - \rangle_F$ denotes the Frobenius inner product of matrices. An elementary derivation of this fact is provided in Appendix A.

While the above formulas may be used to get an output schmear from the matrix-normal inverse-Wishart distribution and the input distribution for a *linear* model, we are interested in the case of a potentially *nonlinear* model where $\tilde{z} = \phi(\tilde{x})$ for some feature mapping $\phi : \mathbb{R}^s \to \mathbb{R}^r$. Once again, we borrow a trick from the field of control theory, since from a schmear over $\tilde{x}$, we may use an *unscented transform* [6] to obtain an estimate of the schmear over $\tilde{z}$ by using a collection of sigma-points over the input distribution, transforming them through ϕ, and then computing the empirical mean and covariance of the transformed points. [TODO: diagram of the whole process here]

By composing an unscented transform on the argument-space schmear to obtain a feature-space schmear with the above description of the output schmear obtained by applying a given linear model to a feature-space schmear, we are now able to straightforwardly impute output schmears from arbitrary term applications.

5.2.2 Updating the Output Model

For the second stage of a prior update, we need to utilize the obtained output-space schmear over the embedding space $(Z \to W)$ to update our information about the model $w = A * \phi(z) + \epsilon$, which we assume has a $\mathcal{MNIW}_{t,r}(B, \Lambda, V, v)$ prior over the relevant model parameters.

First, it is important to note that there is an inherent ambiguity in the interpretation of the covariance of a schmear in the embedding space for $(Z \to W)$ with respect to our usual model. Fundamentally, this is due to the fact that the covariance in the linear operator obtained from a matrix-normal inverse-Wishart distribution is only expressible as a *Kronecker product* of relevant input/output covariances, which is in general invariant with respect to alterations of the relative scaling of the input covariance and output covariance factors. [3]. Luckily, in our case, we may bypass this difficulty by assuming that the schmear for the prior update carries *no information* about the residual noise ϵ. This is a reasonable thing for us to do, since we only can get a sense of the "typical" size of the residual error from data updates, not prior updates.

After adopting the assumption that we gain no information about the noise covariance, we can note that the $\mathcal{MNIW}_{t,r}$ likelihood may be separated back into its constituent matrix-normal and inverse-Wishart components, of which we only want to update the matrix-normal component, since the inverse-Wishart component entirely pertains to the likelihood for the noise covariance. Recall that in our models, the matrix normal component of the likelihood is given by $\mathcal{MN}_{t,r}(B, \Lambda^{+}, \mathbf{\Sigma}_\epsilon)$, where $\mathbf{\Sigma}_\epsilon$ is the noise covariance. Ignoring scaling factors, the log-likelihood of this distribution is given by:

$$\mathcal{L}_{\mathcal{MN}}(X) = -\frac{1}{2}\langle \Sigma_\epsilon^{+}, (X - B)\Lambda(X - B)^T \rangle_F + C$$

for some normalizing constant $C \in \mathbb{R}$. Now, suppose that we also have a schmear $(\mu, \Sigma) \in \mathcal{S}^{t*r}$ over the space that $vec(X)$ belongs to. We would like to obtain a reasonable log-likelihood function from the schmear which is amenable to manipulation when combined with the above log-likelihood for our pre-existing matrix-normal model. To do so, since we are assuming that our update will yield no additional information about the error covariance, a reasonable functional form for such a log-likelihood is given by another matrix normal log-likelihood:

$$\mathcal{L}_{schmear}(X) = -\frac{1}{2}\langle \Sigma_\epsilon^{+}, (X - M)\Lambda_\sigma(X - M)^T \rangle_F + C$$

for some $r \times r$ matrix Λ_σ which we will derive from Σ, where $vec(M) = \mu$ and C is once again a normalizing constant.

To obtain a suitable Λ_σ, we simply find where $\frac{V}{v-t-1} \otimes \Lambda_\sigma^{+}$ is closest to Σ in the Frobenius norm. This yields the simple expression:

$$\Lambda_\sigma = \frac{V\Sigma_T^{+}}{v - t - 1}$$

where Σ_T is the result of re-interpreting Σ as 4-tensor laid out in matrix form as $(t \times r) \times (t \times r)$ and transposing dimensions to re-order as $(t \times t) \times (r \times r)$, which is once again laid out in matrix form.

Once we have done this, we can convert the likelihood of the update to a form which is not conditional on the error covariance by packaging it as a matrix-normal inverse-Wishart distribution whose inverse-Wishart component is completely non-informative. The result of doing this is the distribution $\mathcal{MNIW}_{t,r}(B, \Lambda, V, v)$ where:

$$B = M$$
$$\Lambda = \Lambda_\sigma$$
$$v = t$$
$$V = \mathbf{0}$$

Using the formulas expressed in [CITE own ref!], we note that if we have two matrix-normal inverse-Wishart priors $\mathcal{MNIW}_{t,r}(B_1, \Lambda_1, V_1, v_1)$ and $\mathcal{MNIW}_{t,r}(B_2, \Lambda_2, V_2, v_2)$, we may combine the information from these priors into a new matrix-normal inverse-Wishart prior $\mathcal{MNIW}_{t,r}(B, \Lambda, V, v)$ using the $\mathcal{MNIW}$-sum as follows:

$$B = (B_1\Lambda_1 + B_2\Lambda_2)\Lambda^+$$
$$\Lambda = \Lambda_1 + \Lambda_2$$
$$v = v_1 + v_2 + t$$
$$V = V_1 + V_2 + (B_1 - B)\Lambda_1(B_1 - B)^T + (B_2 - B)\Lambda_2(B_2 - B)^T$$

Consequently, we may use the above formulas to incorporate the information from the imputed $\mathcal{MNIW}$ distribution into the model for the output term. If we want to later remove the information added to the model in this way (e.g: because we have more accurate information to apply instead), we can simply take the $\mathcal{MNIW}$ distribution used in the update and add its *additive inverse* to the model parameters instead. In line with [CITE own reference], the additive inverse of a matrix-normal inverse-Wishart prior $\mathcal{MNIW}_{t,r}(B_+, \Lambda_+, V_+, v_+)$ is given by $\mathcal{MNIW}_{t,r}(B_-, \Lambda_-, V_-, v_-)$ where:

$$B_- = B_+$$
$$\Lambda_- = -\Lambda_+$$
$$v_- = 2t - v_+$$
$$V_- = -V_+$$

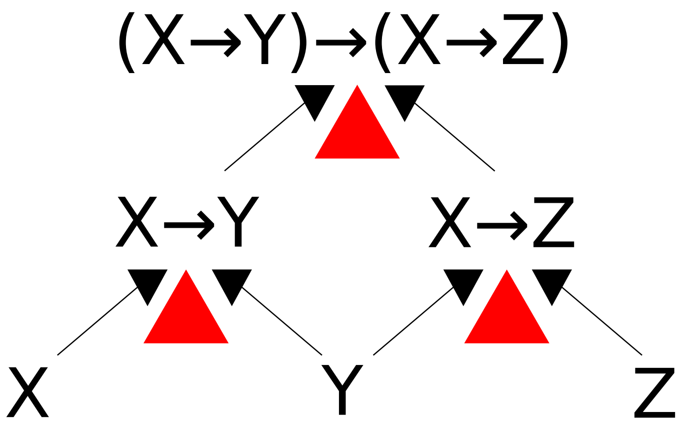
5.3 Update Ordering

With our previously-described mechanisms for updating the models for function terms with respect to data updates and prior updates, we are now faced with the task of scheduling these updates during the regular operation of the interpreter. Since, for a function type $X \to Y$ where $Y = Z \to W$, data updates will depend on paired terms in X and Y, and prior updates will be issued back to the terms in Y which were outputs of the function. Consequently, there is an inherent cyclicity in the update process. Ideally, we would deal with this by deriving explicit expressions for the fixed point of the update process. However, given the intricacies of the processes we have described, it is highly unlikely that there is such a closed-form expression for the fixed point. Instead, we will simply settle for an update process which will gradually approach a fixed point in the large-sample limit, as the number of interpreter-evaluated terms goes to infinity.

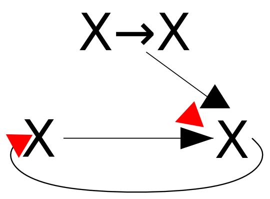
To do so, we first note that after each term application is evaluated by the interpreter, we would ideally like to only update those terms which need updating as a consequence of the just-performed evaluation. To this end, we define two kinds of passes over the structure defined by all term applications which have been evaluated in the past.

First, we define a *data update pass* as a pass where we take the newly-updated term applications from a previous pass, and update the models for the functions. Then, we repeatedly take the terms that were modified, and find all term applications involving them as either the argument or the result of an evaluation until there are no more terms to propagate data updates for. The following is a schematic for this pass:

For the prior updates, we also define a *prior update pass* as a pass where we take all function terms modified in a previous pass and compute prior updates on their output spaces, if their output type is a function type. We then iterate this until we run out of terms to propagate prior updates for. It's important to note that we only perform prior updates when

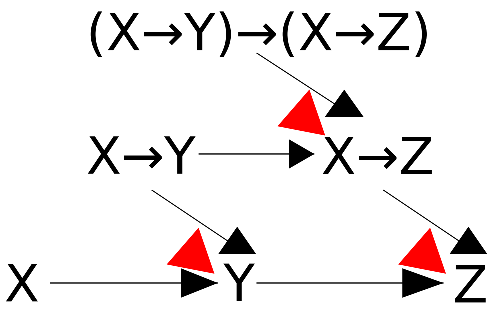


the *function* embedding changes. This is necessary because if we propagated updates whenever the argument embedding changed, we could easily run into an infinite loop, as in the following figure.



The following figure is a schematic of the prior update pass.

In our reference implementation, after each batch of interpreter evaluations, we perform one data update pass followed by one prior update pass.



6 Conclusions and Directions for Further Research

We have presented an incremental procedure to keep distributional embeddings of terms in a simply-typed combinatory calculus updated in a Bayesian fashion. The authors hope that this procedure yields the inspiration necessary for an entirely new class of machine-learning systems to take root.

But before discussing such potential applications, the authors think it prudent to suggest extensions to the basic framework presented in this paper. In particular, the framework we have discussed is far from unique – different choices could have been made at many steps along the way. For instance, instead of the matrix-normal inverse-Wishart conjugate Bayesian models we have adopted for the functional embeddings, a more expressive class of distributions may have been chosen instead. For another example, instead of using the unscented transform to propagate uncertainty through nonlinear functions, we could have instead used a local Taylor series approximation, drawn empirical samples from the distribution instead of using sigma points, or even regularized the estimates to a degree to reduce the risk of overconfidence in our estimates. However, we largely consider such things to be incremental improvements.

More intriguing to the authors is the potential for extending our framework to languages with type systems with more expansive capabilities. A very simple, yet impactful extension that the authors have in mind is to extend the framework to allow for types of finite, unbounded-length sequences, such as lists and strings. However, the authors also wonder if the framework may be extended to universal types, which could provide better generalization behavior over types, or if the framework could be extended even further in some fashion to allow some analogue of dependent typing.

On applications, the authors believe that the most immediate task at hand is to leverage the proposed framework to tackle supervised learning problems. In particular, the authors believe that some variant of Monte Carlo Tree Search which takes into account the probability distributions defined by the embeddings could be a fruitful avenue in this pursuit. Here, the benefits of the proposed framework really show, since if supervised learning is tackled under this framework, the nature of this framework virtually guarantees that multi-task learning will follow soon after.

Taking a longer view, the authors are particularly interested in applications of the proposed framework to general reinforcement learning problems. On the purely-theoretical side, the authors ask whether there is a version of the universal artificial general intelligence procedure AIXI to the setting we have described, of a combinatorial base language for which the distributional information maintained over programs is fundamentally spatial and kept in the form of embeddings through probability distributions. On the practical side, the authors are also curious as to whether there are any immediate and pragmatic fusions of the proposed framework with typical algorithms from the reinforcement learning setting. We leave this rich area for exploration open to other daring researchers who wish to investigate one of the most important topics of our time leveraging the framework which we have described.

Our reference implementation in Rust is available under the MIT License at `https://github.com/bubble-07/FETISH-RS`.

References

[1] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

[2] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis.* CRC press, 2013.

[3] Hunter Glanz and Luis Carvalho. An expectation-maximization algorithm for the matrix normal distribution. *arXiv preprint arXiv:1309.6609*, 2013.

[4] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[5] Marcus Hutter. A theory of universal artificial intelligence based on algorithmic complexity. *arXiv preprint cs/0004001*, 2000.

[6] Simon J Julier and Jeffrey K Uhlmann. New extension of the kalman filter to nonlinear systems. In *Signal processing, sensor fusion, and target recognition VI*, volume 3068, pages 182–193. International Society for Optics and Photonics, 1997.

[7] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in neural information processing systems*, pages 2016–2025, 2018.

[8] Kaare Brandt Petersen and Michael Syskind Pedersen. The matrix cookbook, nov 2012. *URL http://www2. imm. dtu. dk/pubdb/p. php*, 3274:14, 2012.

[9] Ninh Pham and Rasmus Pagh. Fast and scalable polynomial kernels via explicit feature maps. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 239–247, 2013.

[10] Thomas Pierrot, Guillaume Ligner, Scott E Reed, Olivier Sigaud, Nicolas Perrin, Alexandre Laterre, David Kas, Karim Beguir, and Nando de Freitas. Learning compositional neural programs with recursive tree search and planning. In *Advances in Neural Information Processing Systems*, pages 14673–14683, 2019.

[11] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2008.

[12] Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

[13] Peter E Rossi, Greg M Allenby, and Rob McCulloch. *Bayesian statistics and marketing.* John Wiley & Sons, 2012.

[14] Stanley Sawyer. Wishart distributions and inverse-wishart sampling. *URL: www. math. wustl. edu/sawyer/hmhandouts/Whishart. pdf*, 2007.

[15] Ray J Solomonoff. A formal theory of inductive inference. part i. *Information and control*, 7(1):1–22, 1964.

[16] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A survey on neural architecture search. *arXiv preprint arXiv:1905.01392*, 2019.

Appendix A Derivation for Output Schmear Imputation

Consider the linear model $Az + \epsilon = y$ where A is $t \times r$ and suppose we employ a $\mathcal{MNIW}_{t,r}(B, \Lambda, V, v)$ prior. Suppose further that we have an input schmear given by $E[z] = \mu$ and $Cov[z] = \Sigma$.

First, note that using an observation we previously made about the $\mathcal{MNIW}$ distribution, we can notice that:

$$E[vec(A)] = vec(B)$$

$$Cov[vec(A)] = \frac{V}{v-t-1} \otimes \Lambda^+$$

Through elementary manipulation [see, e.g: here], we can see that:

$$Cov[Az]_{kl} = \sum_i \sum_j Cov[A_{ki}, A_{jl}](\Sigma_{ij} + \mu_i \mu_j) + B_{ki} B_{lj} \Sigma_{ij}$$

Since $Cov[A_{ki}, A_{jl}] = \frac{1}{v-t-1} V_{kl} \Lambda^+_{ij}$, we can substitute this expression in and re-express the outer double-summation in matrix operations to arrive at the claimed result:

$$Cov[Az] = [\langle \Sigma, \Lambda^+ \rangle_F + \mu^T \Lambda^+ \mu] \frac{V}{v-t-1} + B \Sigma B^T$$