fdag: A C++ Toolkit for Nonlinear Function Learning

Kevin Gimpel
Toyota Technological Institute at Chicago

Abstract

We describe fdag, a C++ toolkit for nonlinear function learning.[1] It is simple and lightweight, built for rapid prototyping and ease of extension. fdag implements automatic differentiation for functions representable by a formalism we call **formula directed acyclic graphs** (FDAGs). An FDAG is a directed acyclic edge-ordered multigraph in which nodes, called **formulas**, are operators like $+$ or tanh, parameters like θ, or constants like 7. Directed edges point from formulas toward their **child** formulas. Each formula has a **value**. To compute the value of an operator formula, the operator is applied to the values of all child formulas.

FDAGs are related to other low-level computational frameworks like sum-product networks (Poon and Domingos, 2011), but permit arbitrary operators with arbitrary arity and allow undirected cycles in the graph structure, increasing computation reuse. Classic algorithms for automatic differentiation are applicable, though they differ slightly from backpropagation due to the cycles. fdag can be used to compute gradients for training deep neural networks, but can also be used for a broader class of nonlinear functions that are not necessarily composed of "neural" layers. The philosophy of fdag is similar to that of the much more powerful Theano (Bergstra et al., 2010) and Dyna (Eisner and Filardo, 2011), but we instead focus on simplicity and expressivity. When using fdag, one writes programs that directly generate FDAGs, rather than specifying symbolic expressions which are then compiled into computation graphs. As a result, fdag requires more code to do simple computations than other toolkits, but complex computations may be easier in fdag as one is not limited by the capabilities of the symbolic expression language. This can be especially useful with richly-structured recursive functions such as recursive/recurrent neural networks.

In this document, we introduce fdag and describe the FDAG formalism. We give an overview of how to implement simple function learning in fdag, then give several examples, including logistic regression and a multilayer perceptron. We give details of supported operators and our automatic differentiation algorithm in the appendices.

[1]The work described here is based on efforts of several members of Noah's ARK in the Language Technologies Institute at Carnegie Mellon University, especially Shay Cohen, Dipanjan Das, Chris Dyer, Michael Heilman, Noah A. Smith, and Mengqiu Wang, and was originally inspired by elements in the Dyna language (Eisner and Filardo, 2011). The released version of the toolkit was written primarily by me, but is based on algorithms and code that have been circulating among the aforementioned (and others) for many years. No novelty is claimed for any of the technical material herein; I have merely concretized some ideas that have been in the air for many decades in an attempt to make them more accessible and useful.

Contents

1 Introduction

We describe `fdag`, a C++ toolkit for nonlinear function learning. `fdag` is inspired by deep learning but supports a broader class of functions than that generally assumed by neural networks. In particular, `fdag` supports functions representable as **formula directed acyclic graphs** (FDAGs), using automatic differentiation to compute gradients with respect to parameters. This relieves the programmer from working out backpropagation algorithms each time the function architecture is changed, enabling rapid prototyping.

Several generic toolkits already exist for function learning, many of which also include implementations of automatic differentiation algorithms. Examples include Theano (Bergstra et al., 2010) and Dyna (Eisner and Filardo, 2011). Why create another one? The design of `fdag` was driven by the following goals:

- **Simplicity**: `fdag` is readable and concise, with the core toolkit containing approximately 1200 lines of C++ code. It does not use any external libraries or C++11 features, making it easy to compile and run on a wide range of systems. The intent is for researchers to rapidly learn the capabilities of `fdag` and begin implementing models immediately. To this end, we provide concise implementations of several example models that can be used or extended. We include a variety of operators used in deep networks, but it is straightforward to implement new, arbitrary-arity operators, even those that require arbitrary amounts of additional state. All that is required is to implement code for evaluating the node based on its children and differentiating the node's value with respect to each (numbered) child. See Table 1 in Appendix A for currently-supported operators. Simple finite difference checks are provided for debugging gradient computation when experimenting with new function architectures or operators.

- **Flexibility**: `fdag` is deliberately lower-level than Theano and Dyna. The latter compile symbolic expressions into computation graphs upon which operations are performed. `fdag` requires the programmer to write code that generates computation graphs (i.e., FDAGs) directly. Rather than using the static graph transformations of Theano or Dyna, the programmer is responsible for doing his own FDAG substructure reuse. This is often reminiscent of memoization in dynamic programming. The advantage of this trade-off is flexibility. The programmer has direct access to the underlying machinery that performs computations, which only makes sense here because it is simple enough to understand easily. Also, the programmer need not worry about how to represent complex expressions in the symbolic language supported by the toolkit, but can instead focus on the structure of the graph that performs the desired computations. This can be especially useful for functions with rich dependencies that resist concise symbolic expressions, such as those found in recursive and recurrent neural networks. The underlying representation is always at least as flexible as any symbolic expression-based interface to it.

- **Transparency**: the two aims above have the side effect of promoting transparency while developing with `fdag`. The code is short and simple enough to manually inspect. FDAGs

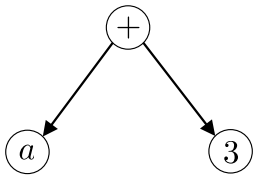

Figure 1: Example FDAG. The value computed at the root node is $a + 3$.

can be printed and checked during debugging. It is hoped that `fdag` can be used as a teaching tool in courses and by graduate students in machine learning. It is also hoped that it can be easily adapted and/or translated to other programming languages of interest.

In the remainder of this document, we detail the formal elements of `fdag`, give details of implementing models, and discuss examples.

2 Formula Directed Acyclic Graphs

We now present our formula directed acyclic graph (FDAG) formalism. An example is shown in Figure 1. Each circle in the figure is a **node** in the FDAG. There is an **operator** node $+$, a **parameter** node a, and a **constant** node 3. Each node has directed edges leading to its **child** nodes. A node may have zero or more children. The graph must be connected and may not have directed cycles. There must be a single node with no incoming edges which is designated as the **root** of the FDAG. Each node has a **value** and a **derivative**. Each operator node **type** (e.g., "plus" or "times") has an evaluation function we denote EVAL and a derivation function we denote DERIVE. To denote the output of the EVAL function called on a node x, we use the notation EVAL(x). We define the EVAL and DERIVE functions for several node types in Table 1 in Appendix A. Intuitively, parameters and constants evaluate to their values, and operators perform their operation on their children.

We show a larger example in Figure 2. This FDAG represents "$(a + 3)^2 + 4a^2$". We note that operators need not be binary. We permit arbitrary arity for operators, including unary operators. We also note that the FDAG in Figure 2 contains repeated substructure, and that it is a tree (no undirected or directed cycles). Using trees leads to very simple algorithms for automatic differentiation via backpropagation. However, if we relax the tree constraint to permit undirected cycles, we can obtain smaller graphs that permit more reuse of computation. In Figure 3, we show an equivalent FDAG that has fewer nodes. We created directed edges to the single a parameter node each time a is used, and did the same for the "$a + 3$" subtree in the "$(a + 3)^2$" term. This strategy does not change the result of calling EVAL on the root. It makes its computation faster by using fewer nodes and edges.

Formally, an FDAG is a tuple $(V, E, r, \boldsymbol{\theta}, \Phi)$, where V is a set of **nodes**, $E \subseteq V \times V$ is the set of directed edges connecting nodes in V, $r \in V$ is the unique root node which has no incoming

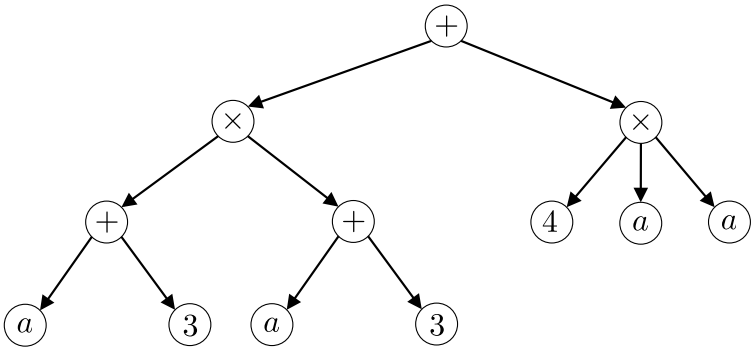

Figure 2: Larger example. Computes $(a + 3)^2 + 4a^2$.

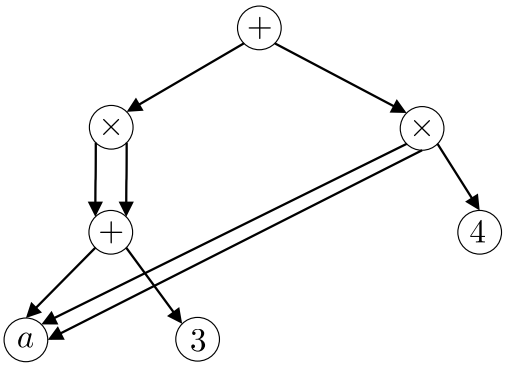

Figure 3: Smaller FDAG to compute the same expression as Figure 2. Note the use of (undirected) cycles to reuse computation.

edges, $\boldsymbol{\theta}$ is the parameter set for the FDAG, and $\Phi : V \to \boldsymbol{\theta}$ is a **parameter map**, a function that links parameter nodes to actual parameters in $\boldsymbol{\theta}$.

We define a **node** x as a tuple $\langle t, \boldsymbol{c}, u \rangle$, where t is the node's **type**, $\boldsymbol{c} = \langle v_1, v_2, ..., v_{|\boldsymbol{c}|} \rangle$ is a vector of references to child nodes in V, possibly with repeated entries, and u is the node's **value**. The type may be an operator like $+$ or $\times$ or any other (arbitrary-arity) operator, or may indicate that the node is a constant (in which case $\oplus = \gamma$) or parameter ($\oplus = \pi$). If the node is a constant, we assume u contains the constant value that is filled upon creation of the node and is never changed. If node x is a parameter, then whenever the node's value u is requested, $\Phi(x)$ is returned. Since the child vector $\boldsymbol{c}$ is ordered and can contain duplicates, the formalism is actually closer to a **directed acyclic edge-ordered multigraph** rather than a DAG, though we retain the term DAG here for its familiarity and since it evokes the essential idea of the formalism: (undirected) cycles can increase reuse of computation and thereby speed up learning.

3 Implementing Models in `fdag`

We now discuss some details of implementing models using `fdag`. The expected use of `fdag` is to learn models by minimizing a loss function that sums over examples in the training data. For each term in the sum, we build an FDAG f, compute its value, and perform automatic differentiation to compute the gradient of the value with respect to the parameters in f. Functions are provided for online or mini-batch parameter updating via stochastic gradient descent or AdaGrad (Duchi et al., 2011), among others.

The `Model` class (`src/fdag/model.*`) owns parameter and gradient vectors and has several useful functions for learning. When creating the FDAG for a training example, The `Model` functions `getFormulaObject(operator)`, `getConstantFormulaObject(v)`, and `getCachedParameterFormula(id)` should be used to get `Formula` objects. The `getCachedParameterFormula(id)` function uses a cache to store pointers to parameter `Formula` nodes so that they can be reused whenever possible, and also so that automatic differentiation works correctly. The automatic differentiation algorithm requires calling a function on each parameter `Formula` node in the FDAG. All `Formula`'s corresponding to model parameters are cached upon creation by the `Model` class and reused if the `getCachedParameterFormula()` function is called whenever parameter `Formula`'s are needed in user code.[2]

After creating the FDAG for an example, the `evalAndComputeDerivatives()` function in the `Model` class is provided for evaluating the FDAG and computing derivatives of its root with respect to the parameters in the cache of parameter `Formula`'s. For debugging, this function can optionally perform simple finite difference checking, which is essential when debugging FDAG construction code. An error message is printed if the derivative for parameter j is such that

$$\left| \nabla_j^{\text{auto}} - \nabla_j^{\text{numerical}} \right| > \delta$$

where ∇_j^{auto} is the derivative of parameter j computed by automatic differentiation and $\nabla_j^{\text{numerical}}$ is the derivative estimated via the following finite difference method:

$$\nabla_j^{\text{numerical}} = \frac{f(\boldsymbol{\theta} + \delta \boldsymbol{j}) - f(\boldsymbol{\theta} - \delta \boldsymbol{j})}{2\delta}$$

where $\boldsymbol{\theta}$ is the parameter vector, $\delta > 0$ is some small value (we use 1×10^{-6} by default), and $\boldsymbol{j}$ is a "one-hot" vector with entry j set to 1 and all other entries set to zero.

Functions in the `Model` class are provided for parameter updating after the gradient vector has been computed: `doSGDUpdate()`, `doSGDMomentumUpdate()`, `doAdaGradUpdate()`, and `doAdaDeltaUpdate()`. After calling them, call `resetGradientVector()` to reset the gradient vector to all zeroes. Mini-batch optimization is easily supported: when computing the gradient on an FDAG, the new derivative is added to the old entry in the gradient

[2] If you instead wish to bypass the cache (e.g., if you are using your own parameter Formula cache or you want to keep separate parameter Formula nodes in your FDAGs each time a parameter if used), use `getParameterFormulaObject(id)` instead of `getCachedParameterFormula(id)`.

vector. Call `evalAndComputeDerivatives()` after creating the FDAG for a single example, then call `resetFormulaObjects()` as usual, but only call `do*Update()` (followed by `resetGradientVector()`) after computing gradients for the FDAGs for all examples in the mini-batch.

This process can involve frequent creation and deletion of `Formula` objects. To improve efficiency, the `Model` class uses pools of `Formula` instances (see `FormulaPool` in `src/fdag/formula_pool.*`). Rather than constantly creating new `Formula` instances, they are drawn from the pool when needed. After processing an example, the model's `resetFormulaObjects()` function must be called to reset the formula pool and reset cached parameter `Formula` nodes for the next example. The initial pool capacity is passed as an optional argument to the `Model` constructor, which can affect performance. Choosing an initial capacity equal to the high watermark of `Formula` node usage across examples will avoid allocating new `Formula` instances after initialization. When you are finished processing an example and no longer need any of the Formula objects you created, call `resetFormulaObjects()`. If you are using the parameter `Formula`'s cache, pass true as the argument.

4 Examples

We now give examples of model implementations that use `fdag`. Self-contained code is provided for each in `src/samples`. Each example uses the `Model` class in `src/fdag/model.h`. For larger examples it may be more appropriate to extend `Model` to have finer control over its members.

4.1 A Minimal Example

We start with a simple example of building an FDAG and computing its value and derivatives of its parameters. The program is contained in the file `src/samples/run_minimal_example.cpp`. It creates a model with a single parameter a with value 5, creates an FDAG for "$2a + 3$", prints it, computes derivatives, and prints it again. Running `./run_minimal_example` should produce the following output:

```
Printing FDAG before evaluation:
+   0   0
    *   0   0
        const   2   0
        param_0 0   0
    const   3   0

Computed value = 13

Printing FDAG after evaluation:
+   13  1
    *   10  1
```

```
      const   2   0
      param_0 5   2
  const   3   0
```

We print an FDAG by calling `printFormula()` on the root `Formula`. This function prints the `Formula` node's type (e.g., + or `param_0` or `const`) followed by its value and its derivative. Then it recurs on its children, printing a single `Formula` per line and adding tabs to indicate nesting of children. The derivative of a node is here defined as the derivative of the root of the FDAG with respect to the particular node. So the derivative of $2a + 3$ with respect to $2a + 3$ is 1, the derivative of $2a + 3$ with respect to $2a$ is 1, and the derivative of $2a + 3$ with respect to a is 2. The derivative of a constant node is defined to be 0. Before evaluation and automatic differentiation, all values are 0 (other than those for constant nodes) and all derivatives are 0. After evaluation, the values and derivatives for all nodes have been filled in.

4.2 Logistic Regression

We now describe the implementation of a multiclass logistic regression classifier in `fdag`, contained in the file `src/samples/run_logistic_regression.cpp`.

Given data with d features for each instance and ℓ possible labels, we use a $d \times \ell$ weight matrix W, where W_{ji} is the weight for feature j and label i. We use a length-ℓ bias vector $\boldsymbol{b}$ where b_i is the bias weight for label i. Then the logistic regression model assigns the following probability to label i for input feature vector $\boldsymbol{x} = \langle x_1, x_2, ..., x_d \rangle$:

$$\Pr(i|\boldsymbol{x}) = \frac{\exp\left\{b_i + \sum_{j=1}^{d} W_{ji} x_j\right\}}{\sum_{k=1}^{\ell} \exp\left\{b_k + \sum_{j=1}^{d} W_{jk} x_j\right\}}$$

During training, we minimize log-loss summed over all training examples; it is written below for a single training example $\langle i, \boldsymbol{x} \rangle$:

$$\text{loss}(i, \boldsymbol{x}) = -\log \Pr(i|\boldsymbol{x}) = -\left(b_i + \sum_{j=1}^{d} W_{ji} x_j\right) + \log \sum_{k=1}^{\ell} \exp\left\{b_k + \sum_{j=1}^{d} W_{jk} x_j\right\}$$

The function `createScoreFDAGForLabel`$(i, \boldsymbol{x})$ builds the FDAG for $\left(b_i + \sum_{j=1}^{d} W_{ji} x_j\right)$. We note that this FDAG appears twice for the correct label but we can save computation by only building it once. The function `createLossFDAGForDatum`$(i, \boldsymbol{x})$ builds the FDAG for $\text{loss}(i, \boldsymbol{x})$.

Figure 4 shows the FDAG for the loss function for a binary classification task where the given example has correct label "0". We only build the score FDAG for each label once, and use an undirected cycle to link twice to the score for the label that is correct for this example. This saves computation without changing the result.

We also include an L2 regularization term with coefficient C. The function `createL2RegularizationTerm` creates an FDAG for the following, where n is the num-

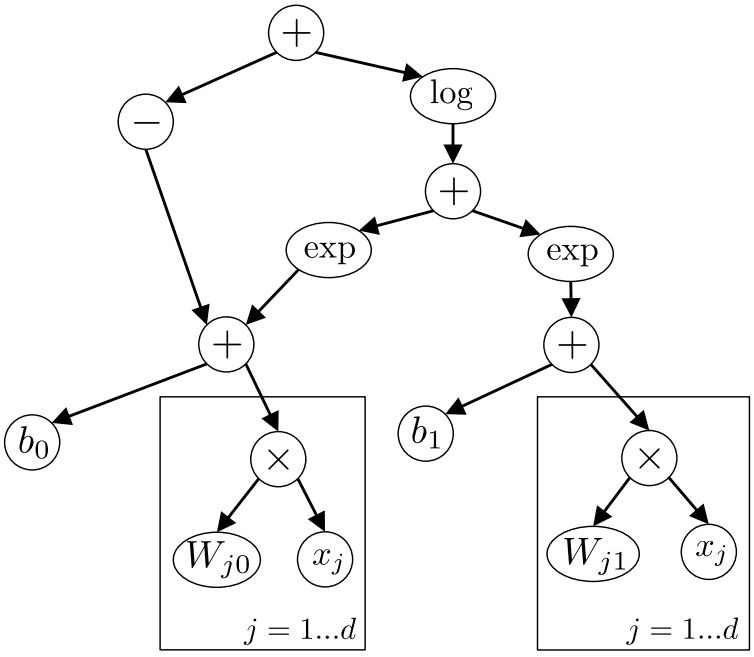

Figure 4: FDAG showing log-loss for a single example for training binary logistic regression. The two labels are 0 and 1. We assume the correct label is 0 for this training example. Since the score for label 0 appears twice, we reuse that part of the FDAG, introducing an undirected cycle in the process. The use of "plate" notation is inspired by analogous notation in graphical models: a plate (i.e., a rectangle in the graph) represents repeated graph structure specified by the bounds in the lower right corner of the plate. We note that we could have also reused the constant formula nodes corresponding to the x_j instead of creating them each twice.

ber of training examples:

$$\text{reg}(W, \boldsymbol{b}) = \frac{C}{2n} \left(\sum_{i=1}^{\ell} b_i^2 + \sum_{i=1}^{\ell} \sum_{j=1}^{d} W_{ji}^2 \right)$$

After building the FDAG f for a single training example, where $f = \text{loss}(i, \boldsymbol{x}) + \text{reg}(W, \boldsymbol{b})$, the function `evalAndComputeDerivatives()` in the `Model` class is called on f. This function evaluates f and computes derivatives with respect to all cached parameter `Formula`'s.
To classify, we return the highest-scoring label (see `classifyDatum`($\boldsymbol{x}$)):

$$\hat{k} = \underset{k}{\text{argmax}} \; b_k + \sum_{j=1}^{d} W_{jk} x_j$$

Experiments We use the ionosphere binary classification dataset from the UCI Machine Learning Repository Bache and Lichman (2013).[3] The code uses 5-fold cross validation by default. After compilation, the experiment can be run as follows: `./run_logistic_regression data_file=data/ionosphere.data`

Details will be printed to `stderr`, ending with the total time for the experiment and the mean held-out accuracy across folds. With $C = 0$, this accuracy should be 85.1429%. Reusing the same `Formula` node for the repeated score (as shown in Figure 4) reduces total training time by about 30%. This experimental comparison can be run by commenting and uncommenting particular lines in `createLossFDAGForDatum()`; see notes therein.

4.3 Deep Neural Networks

We also include an implementation of a feed-forward, fully-connected deep neural network, also known as a multilayer perceptron (MLP). See the file `src/samples/run_dnn.cpp`.

We assume data with d features for each instance and ℓ possible labels. We consider an MLP with h hidden layers and a "softmax" layer at the end to normalize the scores. We use $h+1$ weight matrices $W^{(q)}$, where $W^{(q)}_{ji}$ is the weight for input j and output i in layer q. We use a vector of **layer widths**, denoted $\boldsymbol{\lambda} = \langle \lambda_1, ..., \lambda_h \rangle$. The first weight matrix, $W^{(0)}$, uses the features directly as input and has dimension $d \times \lambda_1$. The final weight matrix, $W^{(h)}$, has dimension $\lambda_h \times \ell$.[4] We also use $h+1$ bias vectors $\boldsymbol{b}^{(q)}$ where $b^{(q)}_i$ is the bias weight for output i in layer q. For the first h bias vectors, the length of $\boldsymbol{b}^{(q)}$ is λ_q; the final bias vector has length ℓ.

The MLP assigns the following probability to label i for input feature vector $\boldsymbol{x} = \langle x_1, x_2, ..., x_d \rangle$:

$$\Pr(i|\boldsymbol{x}) = \frac{\exp\left\{b^{(h)}_i + \sum_{j=1}^{\lambda_h} W^{(h)}_{ji} z^{(h)}_j\right\}}{\sum_{k=1}^{\ell} \exp\left\{b^{(h)}_k + \sum_{j=1}^{\lambda_h} W^{(h)}_{jk} z^{(h)}_j\right\}}$$

where $z^{(q)}_i$ is defined recursively as follows:

$$z^{(q)}_i = g\left(b^{(q-1)}_i + \sum_{j=1}^{\lambda_{q-1}} W^{(q-1)}_{ji} z^{(q-1)}_j\right)$$

where g is a nonlinear function. We use rectified linear units (ReLU) by default. The base case is defined:

$$z^{(0)}_i = x_i$$

For learning, we again minimize log-loss. In lieu of regularization, we use dropout. For $q > 0$, we set $z^{(q)}_i = 0$ with probability α, where α is the **dropout rate**.

[3] Provided in the software release (in `data/ionosphere.data`) and also downloadable from `archive.ics.uci.edu/ml/machine-learning-databases/ionosphere/ionosphere.data`

[4] If $h = 0$, then $\boldsymbol{\lambda} = \langle\rangle$ and the single weight matrix $W^{(0)}$ has dimension $d \times \ell$.

Experiment We again use the ionosphere dataset with the same 5-fold cross validation setting as Section 4.2. After compilation, the experiment can be run as follows: `./run_dnn data_file=data/ionosphere.data layer_widths=10 dropout_rate=0.5`

The `layer_widths` parameter is a comma-delimited string of layer widths. If we set it to 0, we recover logistic regression. The `dropout_rate` parameter is α. Running with a single hidden layer of width 10 (`layer_widths=10`) and a dropout rate of 0.5 (`dropout_rate=0.5`) results in an average held-out accuracy of 90.5714%. We can switch from rectified linear units to tanh activations simply by changing the line

```
Formula* activation = model.getFormulaObject(RELU);
```

to

```
Formula* activation = model.getFormulaObject(TANH);
```

We can easily use more layers of varying widths. E.g., to train with two layers of width 20 we pass `layer_widths=20,20`.

4.4 Class-Separated Deep Neural Networks

We also include an implementation of a feed-forward, fully-connected deep neural network which permits the use of both shared hidden layers across all classes as well as separated hidden layers for each class label. This allows us to learn a separate non-linear scoring function for each class that builds upon a shared non-linear scoring function for all classes. We call the resulting model a class-separated deep neural network (CSDNN). See the file `src/samples/run_csdnn.cpp`.

We assume data with d features for each instance and ℓ possible class labels. We consider h_1 **shared hidden layers** and h_2 **separated hidden layers**. In the code, there can be different numbers and widths of separated hidden layers for each class, but for this exposition we assume that each class has the same sequence of separated hidden layers. We use a vector of **shared layer widths**, denoted $\boldsymbol{\lambda} = \langle \lambda_1, ..., \lambda_{h_1} \rangle$, and a vector of **separated layer widths**, denoted $\boldsymbol{\mu} = \langle \mu_1, ..., \mu_{h_2} \rangle$.

We use h_1 **shared weight** matrices $W^{(q)}$, $1 \leq q \leq h_1$, where $W^{(q)}_{ji}$ is the weight for input j and output i in shared layer q. Defining λ_0 to equal the input dimension d, each shared weight matrix $W^{(q)}$ has dimension $\lambda_{q-1} \times \lambda_q$. We also use h_1 **shared bias** vectors $\boldsymbol{b}^{(q)}$, $1 \leq q \leq h_1$, where $b^{(q)}_i$ is the bias weight for output i in shared layer q. The length of $\boldsymbol{b}^{(q)}$ is λ_q.

For each class label k, we use $h_2 + 1$ **separated weight** matrices $W^{k,(q)}$, $0 \leq q \leq h_2$, where $W^{k,(q)}_{ji}$ is the weight for input j and output i in separated layer q for class label k. For convenience, we define $\mu_0 = \lambda_{h_1}$ and $\mu_{h_2+1} = 1$. That is, the input to the first separated hidden layer for each class is the output of the final shared hidden layer ($\mu_0 = \lambda_{h_1}$). The output of the final separated hidden layer for each class is a single dimension ($\mu_{h_2+1} = 1$) since our goal is to produce a scalar score for each class label. Then, for a given label k, each of the $h_2 + 1$ separated weight matrices $W^{k,(q)}$ ($0 \leq q \leq h_2$) has dimension $\mu_q \times \mu_{q+1}$. We similarly define $h_2 + 1$ **separated bias** vectors $\boldsymbol{b}^{k,(q)}$, $0 \leq q \leq h_2$, where $b^{k,(q)}_i$ is the bias weight for output i in shared layer q for class k. The length of $\boldsymbol{b}^{k,(q)}$ is μ_{q+1}.

The CSDNN assigns the following probability to class label i for input feature vector $\boldsymbol{x} = \langle x_1, x_2, ..., x_d \rangle$:

$$\Pr(i|\boldsymbol{x}) = \frac{\exp\left\{z_1^{i,(h_2+1)}\right\}}{\sum_{k=1}^{\ell} \exp\left\{z_1^{k,(h_2+1)}\right\}}$$

where $z_i^{k,(q)}$ is a class-separated output unit for dimension i, class label k, and separated layer depth q. It is defined recursively as follows:

$$z_i^{k,(q)} = g\left(b_i^{k,(q-1)} + \sum_{j=1}^{\mu_{q-1}} W_{ji}^{k,(q-1)} z_j^{k,(q-1)}\right)$$

where g is a nonlinear function. Rather than defining the base case in terms of the input features as above in the DNN, we define the base case in terms of the shared hidden layer output units:

$$z_i^{k,(0)} = z_i^{(h_1)}$$

where $z_i^{(q)}$ is defined recursively as follows:

$$z_i^{(q)} = g\left(b_i^{(q-1)} + \sum_{j=1}^{\lambda_{q-1}} W_{ji}^{(q-1)} z_j^{(q-1)}\right)$$

The base case is defined:

$$z_i^{(0)} = x_i$$

For learning, we again minimize log-loss. In lieu of regularization, we use dropout. For $q > 0$, we set $z_i^{(q)} = 0$ with probability α, where α is the **dropout rate**.

4.5 Convolutional Neural Networks for Sequence Classification

We include an implementation of a simple convolutional neural network for sequence classification. See the files in `src/samples/cnn/`.

4.6 Recurrent Neural Networks

We include an implementation of a recurrent neural network (RNN). The RNN is used to convert an arbitrary-length sequence into a fixed-size vector, which is then used to predict the output class using a softmax layer. See the file `src/samples/run_recurrentnn.cpp`.

We also include implementations of recurrent neural networks that use long short-term memory units and gated recurrent units. See `src/samples/run_lstmrnn.cpp`.

A sequence classifier that uses both recurrent neural networks and convolutional filters is included in the file `src/samples/seqclassify/run_seqclassify.cpp`.

name	$\oplus$	$\textsc{eval}(x)$	$\textsc{derive}(x, j)$
constant	γ	u	N/A
parameter	π	$\Phi(x)$	N/A
plus	$+$	$\sum_i \textsc{eval}(c_i)$	1
times	$\times$	$\prod_i \textsc{eval}(c_i)$	$u/\textsc{eval}(c_j)$
minus	$-_2$	$\textsc{eval}(c_1) - \textsc{eval}(c_2)$	$\mathbb{I}[j=1] - \mathbb{I}[j=2]$
neg	$-_1$	$-\textsc{eval}(c_1)$	-1
exp	exp	$\exp\{\textsc{eval}(c_1)\}$	u
log	log	$\log(\textsc{eval}(c_1))$	$1/u$
tanh	tanh	$\tanh(\textsc{eval}(c_1))$	$1-u^2$
relu	ReLU	$\max(0, \textsc{eval}(c_1))$	$\mathbb{I}[u \leq 0]0 + \mathbb{I}[u > 0]$
logistic	logit	$1/(1+\exp\{-\textsc{eval}(c_1)\})$	$u(1-u)$
divide	$\div$	$\textsc{eval}(c_1)/\textsc{eval}(c_2)$	$\mathbb{I}[j=1]/\textsc{eval}(c_2)$ $+\mathbb{I}[j=2](-\textsc{eval}(c_1)/\textsc{eval}(c_2)^2)$
power	$\hat{}$	$\textsc{eval}(c_1)^{\textsc{eval}(c_2)}$	$\mathbb{I}[j=1](\textsc{eval}(c_2) * \textsc{eval}(c_1)^{\textsc{eval}(c_2)-1})$ $+\mathbb{I}[j=2](u\log(\textsc{eval}(c_1)))$
max	max	$u \leftarrow \max_i \textsc{eval}(c_i)$, $k \leftarrow \text{argmax}_i \textsc{eval}(c_i)$	$\mathbb{I}[j=k]$

Table 1: Evaluation and derivation function definitions for nodes of different types. The node of interest is denoted $x \in V$, the $\textsc{eval}(x)$ column shows the computed value of node x, and the $\textsc{derive}(x, j)$ column shows the value to propagate to the jth child node of x, i.e., the increment when deriving x with respect to child node c_j. We assume i ranges over child node indices of x. We also assume that $\textsc{derive}(x, j)$ is called after $\textsc{eval}(x)$, and therefore that u holds $\textsc{eval}(x)$. Computing DERIVE for a max node uses some additional bookkeeping, namely the index k of the child node with max value. The values in the "name" column match the identifiers used in the toolkit. The DERIVE function is left undefined for constants and parameters because they never have child nodes.

A Formula Node Types

In Table 1 we show the definitions of the evaluate and derivation functions for several node types supported in the `fdag` toolkit. To implement an additional node type, only the EVAL and DERIVE functions need to be implemented. We note that the max node uses some additional state to store the index of the maximum-value child for differentiation, but this is not essential; it is only done for runtime efficiency at an increase of memory use. We also note that DERIVE for a "times" node uses the fast rule shown here only when $u \neq 0$; when $u = 0$, it checks whether $\textsc{eval}(c_j) = 0$ and returns the product of the remaining children if so.

B Automatic Differentiation in FDAGs

We now describe how we perform automatic differentiation in an FDAG. [To be written]

References

K. Bache and M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer, 2011.

Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *UAI*, pages 337–346, 2011.