Supplementary Protocol

Here we provide details on the implementation of five analyses in Thunder, explaining for each the goal of the analysis, how it can be run from the Python shell (e.g. in iPython) ("*shell*"), how it can be run as a standalone app from the terminal ("*standalone*"), and a walkthrough of the implementation with reference to some of the underlying code ("*walkthrough*"). Further materials available at http://freeman-lab.github.io/thunder/ and http://research.janelia.org/zebrafish.

Direction selectivity

Mass univariate tuning curve analyses describe individual responses as tuned to particular stimulus attributes. Estimating tuning requires two steps: a regression analysis to estimate the response to each orientation, and a calculation of preferred direction and tuning width.

shell

```
>> from thunder.io import load
>> from thunder.regression import RegressionModel
>> from thunder.regression import TuningModel
>> data = load(sc, 'data')
>> regressmodel = RegressionModel.load(modelfile='designmatrix', regressmode='bilinear')
>> tuningmodel = TuningModel.load(modelfile='stimparams', tuningmode='circular')
>> betas, stats, resid = regressmodel.fit(data)
>> params = tuningmodel.fit(betas)
```

standalone

```
$ spark-submit tuning.py <master> <data> <stimparams> <result> --regressmodelfile <designmatrix> --regressmode bilinear
```

walkthrough

First we create a regression model by specifying the design matrix, and the form of regression. The design matrix can be a numpy array (if you are already handling your stimulus parameters in Python) or it can be the location of a file (e.g. a MAT file if you are a Matlab user)

```
regressmodel = RegressionModel.load('designmatrix', 'bilinear')
```

Now fit the model to each voxel

```
betas, stats, resid = regressmodel.fit(data)
```

(***Note***: you'll realize that this operation took no time. That's because in Spark all RDD transformations, inclduing the map operation underlying this step, are lazy; nothing is executed until we try and return a result to the user, or write a result to disk. For now, think of what we're doing as constructing a list of operations we want to perform.)

Fitting the model applied the following function to each voxel in parallel (found in regression/util.py). A variety of other models could be fit simply by replacing this snippet with a different operation.

```
def get(self, y):
  b = dot(self.x_hat, y)
  predic = dot(b, self.x)
  resid = y - predic
  sse = sum((predic - y) ** 2)
  sst = sum((y - mean(y)) ** 2)
  r2 = 1 - sse / sst
  return b[1:], r2, resid
```

Finally, we use the output of the regression to estimate tuning curve parameters, by first constructing a tuning curve model, and then doing the fit, as for regression.

```
tuningmodel = TuningModel.load('stimparams', 'circular')
params = tuningmodel.fit(betas)
```

The result is an RDD with the tuning curve parameters for each voxel, which can be collected into a numpy array (for immediate inspection), or saved to disk.

Principal component analysis

Principal component analysis (PCA) aims to describe whole-brain spatio-temporal responses using a low-dimensional factorization, with spatial and temporal factors that best capture the joint dynamics. The temporal factors capture common profiles of temporal response, and the corresponding spatial factors show to what extent individual voxels exhibit those temporal profiles.

shell

```
>> from thunder.io import load
>> from thunder.factorization import PCA
>> data = load(sc, 'data')
>> pca = PCA(k=3)
>> pca.fit(data)
```

standalone

```
$ spark-submit pca.py <master> <data> <result> 3
```

walkthrough

We perform PCA using the singular value decomposition, after subtracting the mean of each response. To subtract the mean, we use a map operation

```
data = data.mapValues(lambda x: x - mean(x))
```

Our implementation of the SVD is a class that lets you specify the number of singular vectors to return, and whether to use a direct or iterative method, e.g.

```
svd = SVD(k=3, method="direct")
svd.calc(data)
```

svd now has as attributes v (the right singular vectors, as a numpy array), s (the singular values, as a numpy array), and u (the left singular vectors as an RDD).

Under the hood, the algorithm is using one of two strategies for estimating singular vectors. The first strategy is to compute the gramian matrix (the outer product of the matrix with itself), do a local eigendecomposition on the resulting small matrix, then project back into the raw data:

```
mat = RowMatrix(data)
c = mat.gramian() / mat.nrows
eigw, eigv = eig(c / n)
eigw = real(eigw)
eigv = real(eigv)
inds = argsort(eigw)[::-1]
s = sqrt(eigw[inds[0:k]]) * sqrt(n)
v = transpose(eigv[:, inds[0:k]])
u = mat.times(v.T / s)
```

The initial step creates a RowMatrix (a class from the util package). A RowMatrix is backed by an RDD of (key, numpy array) pairs, and a variety of common matrix operations can be performed on it. For example, the gramian method computes the outer product of a matrix with itself by computing and aggregating rank 1 outer products, and the times method uses a map (and broadcasting) to multiply the large matrix row-wise by a smaller one. An alternative strategy, which can be much faster when the dimensionality of the data is large but the number of desired singular vectors is small, is to initialize a set of components with random values, and then iteratively apply a sequence of matrix updates, reformulated as distributed operations:

```
iter = 0
c = random.rand(k, m)
while (iter < maxiter) & (error > tol):
  c_inv = dot(c.T, inv(dot(c, c.T)))
  xx = mat.times(c_inv).gramian()
  cx = dot(c_inv, xx_inv)
  c = mat.rows().map(lambda x: outer(x, dot(x, cx))).reduce(add)
  c = transpose(c)
```

Local post processing, followed by a final map, recovers the singular vectors.

Low-dimensional trajectories

Principal component analysis can be used to recover a dynamical portrait or trajectoryshowing how whole-brain neural activity evolves through a low-dimensional state space. The analysis uses regresion and an SVD to estimate a three-dimensional space on trial-averaged responses, and then projects individual trial data into that space.

shell

```
>> from thunder.io import load
>> from thunder.regression import RegressionModel
>> from thunder.factorization import PCA
>> data = load(sc, 'data')
>> model = RegressionModel.load(modelfile='designmatrix',
regressmode='mean')
>> betas, stats, resid = model.fit(data)
>> pca = PCA(k=3, svdmethod="direct")
>> pca.fit(betas)
>> traj = model.fit(data, pca.comps)
```

standalone

```
$ spark-submit regresswithpca.py <master> <data> <designmatrix>
<result> mean
```

walkthrough

Before doing dimensionality reduction, we want to average responses across repeated trials. This can be implemented using a regression, where the regressor matrix is constructed to compute the average. Although trial averaging is simple, this formulation would support more complex regressions.

```
model = RegressionModel.load('designmatrix', 'mean')
betas, stats, resid = model.fit(data)
```

betas is an RDD that contains the trial-averaged responses of each voxel. We perform dimensionality reduction on this matrix using principal components analysis, using the direct method for the underlying singular value decomposition, which is appropriate assuming the trial duration is relatively short

```
pca = PCA(k=3, svdmethod="direct")
pca.fit(betas)
```

pca now contains as attributes comps (a basis for a low dimensional space, as an array) and scores (the representation of the data in the low dimensional space, as an RDD). We keep the scores as an RDD because they are likely to be large, whereas the comps are small. Finally, to project the trial-by-trial data into the space spanned by these components, we can use the fit method from the regression, passing the components as an additional argument.

```
traj = model.fit(data, pca.comps)
```

Where traj is simply a numpy array containing the trial-by-trial low dimensional trajectories.

Low-dimensional trajectories

Cross correlation can be used to relate stimulus or behavioral variables to individual voxel responses. To capture the fact that the relationships may depend on the relative timing between predictor and response, the analysis examines multiple time lags (with the predictor preceding or following the response), and then uses dimensionality reduction to capture the most common temporal relationships.

shell

```
>> from thunder.io import load
>> from thunder.timeseries import CrossCorr
>> data = load(sc, 'data')
>> crc = CrossCorr(sigfile='behavior', lag=17)
>> corr = crc.calc(data)
```

standalone

```
$ spark-submit ica.py <master> <data> <result> 100 20
```

walkthrough

Create a cross-correlation model to correlate the voxel responses with the behavioral variable at time lags of up to 17 time points. The signal to correlate against can be a numpy array (if you are already processing your behavior in Python) or can be loaded from a file (e.g. a MAT file if you are using Matlab)

```
crc = CrossCorr(sigfile='behavior', lag=17)
```

Compute the cross-correlations

```
coeffs = crc.calc(data)
```

(***Note***: you'll realize that this operation took no time. That's because in Spark all RDD transformations are lazy; nothing is executed until we try and return a result to the user, or write a result to disk. Think of what we're doing here as constructing a list of the operations we want to perform.)

Computing the cross-correlations applied the following function to each voxel (found in timeseries/base.py). A variety of other functions could be used simply by extending or replacing this with a different operation.

```
def get(self, y):
  y = y - mean(y)
  n = norm(y)
  y /= norm(y)
  b = dot(self.x, y)
  return b
```

We now have an RDD where each record is a long vector of cross-correlations for that voxel. To reduce dimensionality, we can pass the result to PCA

```
pca = PCA(k=2).fit(data)
```

pca now as attributes comps (a small numpy array containing the principal components, the two patterns of cross-correlation that together explain the most variance in the data), and scores (an RDD containing the projection of each data point onto these components).

Independent component analysis

Independent component analysis is an unsupervised learning algorithm that aims to decompose a dataset into a set of underlying additive signals, assuming they are statistically independent. The signals are found by optimizing an objective function that measures their non-Gaussianity. The result is a set of spatial weight maps and corresponding temporal profiles. It is particularly useful for data lacking other well-defined stimulus or behavioral structure.

shell

```
>> from thunder.io import load
>> from thunder.factorization import ICA
>> data = load(sc, 'data').cache()
>> ica = ICA(k=100, c=20)
>> ica.fit(data)
```

standalone

```
$ spark-submit ica.py <master> <data> <result> 100 20
```

walkthrough

As is common, before performing ICA, we reduce the dimensionality of the data, and whiten (decorrelate) it, using the SVD. First, create a RowMatrix (because we will be doing repeated matrix operations), and compute the SVD

```
mat = RowMatrix(data)
svd = SVD(k).calc(mat)
```

We can compute the whitening transform locally (because the matrix is small), and then apply it using a map

```
whtmat = real(dot(inv(diag(latent/sqrt(n))), svd.v))
whtdata = mat.times(whtmat.T)
```

The core of ICA is a fixed-point algorithm that iteratively updates and orthgonolizes a matrix. The update depends on a gradient that can be computed through one map-reduce operation on the whitened data

```
b = orth(random.randn(k, c))
iter = 0
err = 0

while (iter < maxiter) & ((1 - err) > tol):
    b = whtdata.rows().map(lambda x: outer(x, dot(x, b) ** 3)).sum() / 
wht.nrows - 3 * b
    b = dot(b, real(sqrtm(inv(dot(transpose(b), b)))))
    err = min(abs(diag(dot(transpose(b), b_old))))
    iter += 1
```

We post-process the result by reapplying the whitening transform (locally, because the matrix is small), and then using a final map to apply the unmixing matrix and estimate the components.