## Contents

1 **Quick Start Guide**  
1.1 Installation ................................................................. 3  
1.2 Usage ........................................................................ 3  

2 **User Guide**  
2.1 Running an Experiment .................................................. 5  
2.2 Analyzing Results ............................................................ 7  

3 **Library Architecture**  
3.1 Low-Level Code ............................................................. 15  
3.2 Experiments ................................................................. 22  
3.3 GLAB API ................................................................. 22  

4 **API Reference**  
4.1 glimpse.backends .......................................................... 23  
4.2 glimpse.models ........................................................... 23  
4.3 glimpse.prototypes ........................................................ 24  
4.4 glimpse.pools ............................................................. 24  
4.5 glimpse.experiment ....................................................... 24  
4.6 glimpse.glab.api ......................................................... 24  
4.7 Command-Line Interface ............................................... 26  

5 **Indices and tables**  

**Bibliography** ................................................................. 31  

**Python Module Index** ...................................................... 33
Welcome to Glimpse, a General Layer-wise IMage ProceSsing Engine!

The Glimpse project [IJCNN2013] is a library for implementing hierarchical visual models in C++ and Python. The goal of this project is to allow a broad range of feed-forward, hierarchical models to be encoded in a high-level declarative manner, with low-level details of the implementation hidden from view. This project combines an efficient implementation with the ability to leverage parallel processing facilities and is designed to run on multiple operating systems using only common, freely-available components. A prototype of Glimpse has been used to encode an HMAX-like model, achieving results comparable with those found in the literature. The project has been supported by NSF Grant 1018967 (PIs: Melanie Mitchell and Garrett Kenyon).

Source code and documentation are available for the project, as well as an installable package.
1.1 Installation

Using `pip`:

```
$ pip install glimpse ipython matplotlib
```

To get the most current (but possibly unstable) version:

```
$ pip install -e git+https://github.com/mthomure/glimpse-project.git#egg=glimpse
```

**Note:** On Mac OSX, you may need to build for a 32-bit architecture. For example, this happens when using 32-bit Python on a 64-bit machine. To do this, download and unpack the project, and then use the modified install command:

```
$ ARCHFLAGS='--arch i386' pip install glimpse
```

1.2 Usage

To get started quickly with Glimpse, use the `glab` API from the ipython shell. In the example below, we perform object detection on a sample dataset using an HMAX-like model.

```
$ ipython --pylab

```

```python
>>> from glimpse.glab.api import *

>>> SetCorpusByName("easy")

>>> ImprintS2Prototypes(10)

>>> EvaluateClassifier()

>>> results = GetEvaluationResults()

>>> print "Classification accuracy:", results.score
0.75

>>> StoreExperiment("my-experiment.dat")
```

The same experiment can be run from the command-line using the `glab` script.
Note: If you have trouble getting access to the glab command, check the note about system paths.
The easiest way to conduct an experiment is to use the GLAB application programming interface (API), which provides a simplified MATLAB-like interface to specify the experimental protocol and drive the computation of results. This section walks through an example use of this API.

### 2.1 Running an Experiment

We start by launching IPython, which is the recommended way to use Glimpse.

```
$ ipython --pylab
```

We then set up the environment:

```
>>> from glimpse.glab.api import *

>>> Verbose(True)
```

Here, we first import GLAB, and then ask that future commands log their activity to the console.

**Note:** Note that the “$” prefix used here denotes a command that is entered at the shell. The >>> denotes text that is submitted to the Python interpreter, or written as commands in a script. Lines without either marker indicate logging output.

The next step is to specify the experimental protocol and configure the model. We first tell Glimpse where to find the image corpora. The simplest way to achieve this is the SetCorpus command, which assumes one sub-directory per object class. That is, each of these sub-directories should contain images for only that object class. In this example, we have a corpus of images in a directory named “cats_and_dogs” with the following structure:

```
cats_and_dogs/
cat/
    cat1.png
    cat2.png
cat3.png
cat4.png
dog/
    dog1.png
dog2.png
dog3.png
dog4.png
```
We tell Glimpse to use this corpus with the following command:

```python
>>> SetCorpus('cats_and_dogs')
INFO:root:Reading class sub-directories from: cats_and_dogs
INFO:root:Reading images from class directories:
    ['cats_and_dogs/cat', 'cats_and_dogs/dog']
```

Here, the system reports that it has read the images for object classes “cat” and “dog” from disk.

**Note:** If you would like to learn about Glimpse without using your own image corpus, try using the `SetCorpusByName()` command.

Next, the model is configured with a set of (in this case 10) imprinted prototypes:

```python
>>> ImprintS2Prototypes(10)
INFO:root:Using pool: MulticorePool
INFO:root:Learning 10 prototypes at 1 sizes from 4
    images by imprinting
Time: 0:00:01 |###############| Speed: 3.07 unit/s
INFO:root:Learning prototypes took 1.304s
```

The first line of log output shows that images are being evaluated in parallel. Since we did not specify a training and testing split, the system has automatically chosen four (i.e., half) of the images for training and the remaining four for testing. The next line of log output confirms that the system imprinted 10 S2 prototypes, using the four images in the training set. (Here, prototypes are of 1 size, because the model uses only 7x7 prototypes by default. If we had configured the model to use 7x7 and 11x11 prototypes, for example, then we would have imprinted 10 prototypes at 7x7, and another 10 at 11x11, or 20 prototypes total.)

Finally, the model is used to extract features for each image, and the classifier is tested in the resulting feature space.

```python
>>> EvaluateClassifier()
INFO:root:Computing C2 activation maps for 8 images
Time: 0:00:01 |###############| Speed: 5.66 unit/s
INFO:root:Computing activation maps took 1.414s
INFO:root:Evaluating classifier on fixed train/test
    split on 8 images using 10 features from layer(s): C2
INFO:root:Training on 4 images took 0.003s
INFO:root:Classifier is Pipeline(learner=LinearSVC [...OUTPUT REMOVED...])
INFO:root:Classifier accuracy on training set is
    1.000000
INFO:root:Scoring on training set (4 images) took
    0.001s
INFO:root:Scoring on testing set (4 images) took 0.001s
INFO:root:Classifier accuracy on test set is 0.500000
```

The log output shows that the system is computing model activity through the C2 layer for all eight images in the corpus. Feature vectors are then constructed from C2 layer activity only, which provides 10 features per image (since we used 10 S2 prototypes). The classifier—which we can see is a linear SVM—is adapted to the training set, and then scored on both the training and test sets. In this case, accuracy was 100% on the training set, and 50% on the test set. See the [GLAB API reference](#) for the full set of available commands.

### 2.1.1 Command-Line Experiments

The experiment described above can also be run from the command line using the `glab` command. This command exposes much of the functionality of the API, but does so through command-line arguments. To run the experiment
above, we could enter the following at the command line (rather than the Python interpreter).

```bash
$ glab -v -c cats_and_dogs -p imprint -n 10 -E
```

This results in the same calls that were used in above, and thus should produce the same log output. Here, the `-v` flag enables the generation of log output, the `-c` flag specifies the image corpus, the `-p` and `-n` flags choose the prototype learning method and number of prototypes, and the `-E` flag evaluates the resulting feature vectors with a linear SVM classifier. The `glab` command has many more possible arguments, which are documented in the `GLAB CLI reference`.

### 2.2 Analyzing Results

Results can be analyzed in much the same way as an experiment is run. Using the `GLAB` API, we can first retrieve the set of images and their class labels:

```python
>>> GetImagePaths()
array([
    'cats_and_dogs/cat/cat1.png',
    'cats_and_dogs/cat/cat2.png',
    'cats_and_dogs/cat/cat3.png',
    'cats_and_dogs/cat/cat4.png',
    'cats_and_dogs/dog/dog1.png',
    'cats_and_dog/dog/dog2.png',
    'cats_and_dog/dog/dog3.png',
    'cats_and_dog/dog/dog4.png'],
   dtype='|S18')
```

```python
>>> GetLabelNames()
array(['cat', 'cat', 'cat', 'cat',
       'dog', 'dog', 'dog', 'dog'],
      dtype='|S3')
```

This means that the filenames and labels are stored as arrays. The “`dtype`” line can be ignored. It simply means that the values in that array are 18-character strings.

#### 2.2.1 Model, Prototypes, and Activity Analysis

Information about the “extraction” phase of the experiment—i.e., how features were extracted from each image—is also available for analysis, which includes the model’s prototypes and parameters. The model parameters can be printed as follows:

```python
>>> params = GetParams()

>>> params
Params(
    cl_kwidth = 11,
    cl_sampling = 5,
    cl_whiten = False,
    [...]OUTPUT REMOVED...]
    s2_operation = 'Rbf',
    s2_sampling = 1,
    scale_factor = 1.189207115,
)
```
For example, this shows that an RBF activation function was used for the S2 layer (according to $s2\_operation$), and that each scale band is $2^{1/4}$ larger than the scale above it (according to $scale\_factor$). The full set of parameters is documented here.

Note: The experiment’s model parameters can be edited with a graphical interface using `SetParamsWithGui()`. However, this must be done before the call to `ImprintS2Prototypes()` above.

If a set of S2 prototypes was used, they are available from the `GetPrototype()` command:

```python
>>> prototype = GetPrototype(0)

>>> prototype.shape
(4, 7, 7)

>>> prototype
array([[[4.08234773e-03, 4.08234773e-03, ..., 4.65552323e-03, 5.46302684e-02
                    ]],
          dtype=float32])
```

Output from the second command shows that each prototype is a seven-by-seven patch with four bands—i.e., a prototype is a three-dimensional array. The width of the patch is determined by the model parameters, as seen by the following:

```python
>>> params.s2_kernel_widths
[7]
```

Remember that an S2 prototype is just a patch of C1 activity. Thus, the number of bands in a prototype is determined by the number of orientations used at the S1 and C1 layers. This can be confirmed as follows:

```python
>>> params.s1_num_orientations
4
```

The number of available prototypes depends on the number that were learned. In our example, 10 prototypes were imprinted, which can be verified as:

```python
>>> GetNumPrototypes()
10
```

As seen above, an S2 prototype is a somewhat complicated object to visualize, particularly in the form of text output. However, there are other ways to visualize a prototype. The simplest is to plot the activation at each band as a set of images, which we do here for the first prototype $^1$.

```python
>>> gray()

>>> ShowPrototype(0)
```

![Figure 2.1: Figure 1: S2 prototype bands, with plots corresponding to edge orientations.](image)

$^1$ Indexing in Python is zero-based, so the first prototype is at index zero.
Note: Plotting in Glimpse requires the `matplotlib` library.

This results in a plot similar to that shown in Figure 1. Here, active locations are shown in white, while inactive locations are shown in black. These four plots correspond to the four edge orientations detected at S1. For reference, the S1 edge detectors\(^2\) are shown as in Figure 2 using the following command:

```python
>>> ShowS1Kernels()
```

![Figure 2: S1 edge detectors.](image)

Unfortunately, the above visualization is not very intuitive. An alternative approach to visualizing an imprinted prototype is to plot the image patch from which the prototype was “imprinted”:

```python
>>> AnnotateImprintedPrototype(0)
```

Note: To add a colorbar to this plot, just type:

```python
>>> colorbar()
```

![Figure 3: Chosen location for the imprinted S2 prototype.](image)

\(^2\) We use the terms “prototype” and “kernel” somewhat interchangeably.

---

### 2.2. Analyzing Results
The results are shown in Figure 3. Taken together, it is much easier to interpret the behavior of this prototype. We see that it was imprinted from the cat’s front leg, which contains a large amount of vertical energy and very little horizontal energy. This is reflected in the S2 prototype matrix, which shows light gray and white for the components corresponding to the two near-vertical detectors, and shows black for much of the components corresponding to the two near-horizontal detectors.

Additionally, we can investigate the model activity that was computed for various layers. Here, we visualize the activity of the S2 layer when the prototype in Figure 1 is applied to the image in Figure 3. We first need to know where the first prototype was imprinted, which is given by:

```python
>>> image, scale, y, x = GetImprintLocation(0)
```

This returns the image and S2 location (including the scale band) for the imprinted prototype. Here, the image is specified by its index in the list of image paths, with other data organized in the same way. To plot the S2 activation for the first prototype on the image and scale from which it was imprinted, use:

```python
>>> AnnotateS2Activity(image, scale, 0)
```

![Figure 1](image1.png)

Fig. 2.4: Figure 4: S2 response for the prototype visualized in Figure 1. For reference, the S2 activity is plotted on top of the image data.

This is shown for our example in Figure 4, where active regions are shown in red and inactive regions are shown in blue. If the `scale` is larger than 2 or 3, the image data may be hard to recognize (as it has been down-sampled multiple times). In this case, try recreating the plot for smaller values of the `scale` argument.

A similar visualization as above is available for S1 and C1 activity as:

```python
>>> AnnotateS1Activity(image, scale)
```

```python
>>> AnnotateC1Activity(image, scale)
```

which should produce results similar to Figure 5 and Figure 6.
Fig. 2.5: Figure 5: S1 response maps (enhanced for illustration).

Fig. 2.6: Figure 6: C1 response maps.
2.2.2 Classifier Analysis

In the final stage of an experiment, a classifier is evaluated on the feature vectors extracted by the model. The information about this stage is available via two functions: `GetEvaluationLayers()` and `GetEvaluationResults()`. First, we can verify that the classifier used an image representation based on C2 activity:

```python
>>> GetEvaluationLayers()
['C2']
```

Information about the classifier’s performance can be given as:

```python
>>> results = GetEvaluationResults()

>>> results.score_func
'accuracy'

>>> results.score
0.5

>>> results.training_score
1.0
```

The first result indicates that the classifier is evaluated based on accuracy, and the second result gives that accuracy on the set of test images. The last result gives the accuracy on the set of training images. The induced classifier can be retrieved as well:

```python
>>> results.classifier
Pipeline(steps=[
    ('scaler', StandardScaler([...OUTPUT REMOVED...]))
    ('learner', LinearSVC([...OUTPUT REMOVED...]))
])
```

The per-image predictions made by the classifier are accessible by using:

```python
>>> GetPredictions()
[('cats_and_dog/cat/cat3.png', 'cat', 'dog'),
 ('cats_and_dog/cat/cat4.png', 'cat', 'dog'),
 ('cats_and_dog/dog/dog1.png', 'dog', 'dog'),
 ('cats_and_dog/dog/dog2.png', 'dog', 'dog')]

>>> GetPredictions(training=True)
[('cats_and_dog/cat/cat3.png', 'cat', 'cat'),
 ('cats_and_dog/cat/cat4.png', 'cat', 'cat'),
 ('cats_and_dog/dog/dog1.png', 'dog', 'dog'),
 ('cats_and_dog/dog/dog2.png', 'dog', 'dog')]
```

The first call returned information about images in the test set, while the second call used the training images. Each entry of the results gives the image’s filename, its true label, and the label predicted by the classifier, respectively. In our example, the classifier predicted “dog” for all test images—thus achieving 50% accuracy—while correctly classifying all training images.

Finally, it may be useful during analysis to compute feature vectors for images outside the experiment’s corpus, which can be done using:

```python
>>> features = GetImageFeatures('bird/bird1.png')
>>> features.shape
(1, 10)
```
This shows that our existing model with 10 S2 prototypes was used to extract 10 features from a single image. To extract features from several images at once, use:

```python
>>> features = GetImageFeatures(['bird/bird1.png', 'bird/bird2.png'])
>>> features.shape
(2, 10)
```
Library Architecture

Glimpse is broken into low-level components that provide fundamental objects (such as `glimpse.models` and `prototype learning`), mid-level code for managing experiments, and the high-level `glab` API.

3.1 Low-Level Code

In Glimpse, a `backend` is an implementation of a set of low-level filtering operations, such as dot products (convolution) and radial basis functions (RBFs). Each backend uses a different technique for implementing these operations, and thus provide different performance benefits. For example, the `base backend` implements most operations in compiled C++.

A `model` implements a full hierarchical network, with layer processing implemented using an arbitrary backend. Functionally, a model provides a transformation from an image to a feature vector given by the activity of the highest layer in the model. In the case of an HMAX-like model, the feature vector may be given by the activity of the C2 layer.

In general, however, this transformation is a mapping between arbitrary states of the network. Thus, given an input state and a description of the desired result, the transformation emits a corresponding output state. Notably, this deals with all (transitive) dependencies needed to compute the output state, provided that these dependencies are eventually satisfied. As an example, this means that a model with three layers (A, B, and C, in that order) can be used to compute layer C from any input state containing either A, B, or both A and B.

A `worker pool` implements a parallelization strategy for evaluating an arbitrary model on a set of images. Example strategies include multi-core processing on a single host, as well as multi-host processing on a compute cluster. More information can be found below.

3.1.1 Backends

A backend is an implementation of a consistent interface, which provides basic operations for filtering N-dimensional arrays. These include filtering operations that build selectivity, pooling operations that build invariance, and an operation providing local contrast enhancement of an image.

Filtering

Four filter operations are supported. The operation `DotProduct` compares the input neighborhood and the weight vector (i.e., prototype) using a dot product, where each output is given by

\[ y = X^T W \]
for input neighborhood $X$ (given as a vector) and weight vector $W$, where $X^T$ denotes the matrix transpose. The operation `NormDotProduct` is similar, but constrains each vector to have unit norm. Thus, the output is given by

$$y = \text{NDP}(X, W) = \frac{X^T W}{\|X\|\|W\|},$$

where $\|\cdot\|$ denotes the Euclidean norm.

Instead of a dot product, the operation `Rbf` compares the input and weight vectors using a radial basis function (RBF). Here, the output is given as

$$y = \exp\left\{-\beta \|X - W\|^2\right\},$$

where $\beta$ controls the sensitivity of the RBF. Constraining the vector norm of the arguments gives the final operation `NormRbf`, where the output is given as

$$y = \exp\left\{-2\beta \left(1 - \text{NDP}(X, W)\right)\right\},$$

Here, we have used the bilinearity of the inner product to write the distance as

$$\|V_a - V_b\|^2 = 2 - 2V_a^T V_b$$

for unit vectors $V_a$ and $V_b$.

### Pooling

Currently, the only operation that is supported is a maximum-value pooling function. For a local neighborhood of the input $X$, this computes an output value as

$$y = \max_{i,j} x_{ij}.$$

This has been argued to provide a good match to cortical response properties \(^1\), and has been shown in practice to lead to better performance \(^2\).

### Contrast Enhancement

Given a local input neighborhood $X$, the output is

$$y = \frac{x_c - \mu}{\max(\sigma, \epsilon)},$$

where $x_c$ is the center of the input neighborhood, $\mu$ and $\sigma$ are the mean and standard deviation of $X$, and $\epsilon$ is a bias term. Thus, we measure the local deviation from the mean, where large deviations are squashed if the window contains a large amount of variance.

The bias term is used to avoid the amplification of noise, and to ensure a non-zero divisor. Without the bias term, this method performs badly on homogeneous regions, where the variance approaches zero. In this case, very small local deviations (usually caused by image noise) become enhanced when the value of the denominator drops below unity. Because of this bias term, we will never enhance deviations, only “squash” them. This removes the noise in the background while retaining contrast enhancing effects of the foreground. This is illustrated below.

---


In Glimpse, the model object defines the network topology, including the number of layers and the operations used at each layer. When the object is constructed, it is given a backend implementation and a set of parameters that control its behavior.

The model can be viewed as a transformation between states, where a state encodes the activity of all computed model layers. To process an image, we first wrap the image in a new state object. The model is then applied to transform this state to a new state, which contains activation for a higher layer in the network. This is shown in the following example.

```python
>>> from glimpse.models.ml import BuildLayer, Model, Layer
>>> model = Model()
>>> istate = model.MakeState("example.jpg")
>>> ostate = BuildLayer(model, Layer.C1, istate)
>>> c1 = ostate[Layer.C1]
```

In this case, the `c1` variable will now contain activation for the C1 layer of the `model`. A feature vector can then be derived from the activation data as:

```python
>>> from glimpse.util.garray import FlattenArrays
>>> features = FlattenArrays(c1)
```

Oftentimes, it may be preferable to use the `glab` module. In this case, the above example could be written as:

```python
>>> SetLayer("C1")
>>> features = GetImageFeatures("example.jpg")
```

There is currently one hierarchical model included in the Glimpse project. It specifies an HMAX-like network, in which an alternating sequence of “simple” and “complex” layers gradually build up object specificity while also building invariance to certain transformations. Specifically, an image is first preprocessed, and then filtered with a
layer of S1 units to detect edges at various orientation and scale. The corresponding response maps are then blurred by replacing each local neighborhood with its maximum activation. This process is then repeated, with a layer of S2 units being applied to result of the C1 layer. Here, each S2 unit is characterized by the input template, or prototype, to which it responds. Given N different prototypes, therefore, the S2 layer will generate N different response maps per scale. Finally, the C2 layer summarizes the image by computing the maximum response for each S2 prototype for any location or scale.

To compute scale bands, the model uses a scale pyramid approach. Instead of using different S1 filters for each scale, the model uses different-sized versions of the input image. Thus, the course-scale response maps are computed by applying a battery of Gabors to the original input image. Response maps for the finest-grained scale use the same battery of Gabors, but apply them to a resized (shrunken) version of the image.

**Preprocessing**

An initial preprocessing stage, referred to as the *retinal* layer, is used to 1) remove color information, 2) scale the input image, and 3) enhance image contrast. Color information is removed according to the ITU-R 601-2 luma transform (see the `Image.convert` method in the Python Imaging Library). Optionally, the input image can also be scaled (via bicubic interpolation), such that its shorter side has a given length. Finally, image contrast optionally is enhanced by applying the `ContrastEnhance` backend function.

**Model Parameters**

Behavior for each model is controlled by a set of parameters, which are described below according to the layer they affect. To customize these parameters, the user should first create a `Params` object corresponding to the model class, and then set the desired values. An example is shown below:

```python
>>> from glimpse.models.ml import Params
>>> params = Params()
>>> params.num_scales = 8
>>> params.s1_num_orientations = 16
>>> m = Model(params)
```

Using the `glab` interface, this simplifies to:

```python
>>> params = GetParams()
>>> params.num_scales = 8
>>> params.s1_num_orientations = 16
```

**Preprocessing Options**

**Image Resizing Method** The method to use when resize the input image. One of “score short edge”, “scale long edge”, “scale width”, “scale height”, “scale and crop”, or “none”. When the method is “scale and crop”, use the length parameter to specify the output image width, and the aspect_ratio parameter to specify the (relative) output image height.

```python
>>> image_resize_method = 'scale short edge',
```

**Image Aspect Ratio** The aspect ratio to use when the resize method is “scale and crop”.

```python
>>> image_resize_aspect_ratio = 1.0,
```

**Image Length** The output image length.

```python
>>> image_resize_length = 220,
```

**Retina Enabled** Whether to use the retinal stage during preprocessing. (Note that color information will always be removed.)
>>> params.retina_enabled = False

**Retina Bias**  The bias term used in the contrast enhancement function to avoid noise amplification.
>>> params.retina_bias = 1.0

**Retina Kernel Width**  Size of the local neighborhood used by the preprocessing function.
>>> params.retina_kwidth = 15

**S1 and S2 Layer Options**

**Beta**  Tuning parameter of the activation function (for Rbf and NormRbf).
>>> params.s1_beta = 1.0
>>> params.s2_beta = 5.0

**Bias**  Bias term for normalization in the activation function (for NormDotProduct and NormRbf operations).
>>> params.s1_bias = 0.01
>>> params.s2_bias = 0.1

**Kernel Width**  Spatial extent of the local neighborhood.
>>> params.s1_kwidth = 11
>>> params.s2_kwidth = [7]

**Note:**  The S2 layer supports kernels (aka prototypes) with multiple different widths. Thus, the *s2_kwidth* parameter is a list.

**Operation**  The form of the activation function (one of DotProduct, NormDotProduct, Rbf, or NormRbf). See the set of filter operations supported by the backends.
>>> params.s1_operation = "NormDotProduct"
>>> params.s2_operation = "Rbf"

**Sampling**  The sub-sampling factor used when computing S-unit activation.
>>> params.s1_sampling = 1
>>> params.s2_sampling = 1

**S1 Gabor Filter Options**

**Number of Orientations**  Number of different Gabor orientations.
>>> params.s1_num_orientations = 4

**Shift Orientations**  Whether Gabors are shifted to avoid lining up with the axes.
>>> params.s1_shift_orientations = False

**Number of Phases**  Number of different phases for the S1 Gabor filters (two phases means detecting a black to white transition, and vice versa).
>>> params.s1_num_phases = 2

**Number of Scales**  Number of different scales with which to analyze the image.
>>> params.num_scales = 9
Scale Factor  (ml model only) The down-sampling factor used to create course representations of the input image.

>>> params.scale_factor = 2**(1/4)

C1 and C2 Layer Options

Kernel Width  Size of the local neighborhood used in the C-unit pooling function.

>>> params.c1_kwidth = 11

Sampling  The sub-sampling factor used when computing C-unit activation.

>>> params.c1_sampling = 5

C1 Whiten  Whether to whiten C1 data. See the Whiten function.

>>> params.c1_whiten = False

3.1.3 Worker Pools

A worker pool implements a strategy for parallelizing a map() operation. That is, given a set of elements and a function taking a single input, a worker pool returns the result of evaluating that function on each input element in turn. In Glimpse, the function is usually a model’s BuildLayer method, and the elements are generally the model’s input state for different images.

When not using a compute cluster, the best worker pool to use is generally the one returned by MakePool(). For example

>>> pool = glimpse.pools.MakePool()
>>> pool.map(hex, [1, 2, 3])
['0x1', '0x2', '0x3']

Single-Host Pools

The most common parallelization scheme is the MulticorePool, which spreads evaluation of elements across multiple cores of a single host. Additionally, a fall-back scheme is provided by the SinglecorePool, which uses the builtin map() function. This can be useful for debugging, and when the complexity of multicore communication is unwanted.

Caution:  Not all functions can be used in a parallel fashion. In case of mystifying errors, check the documentation for MulticorePool. Additionally, try using SinglecorePool to identify whether the error is due to parallelization.

Multi-Host Pools

Multi-host worker pools, or cluster pools, are more advanced than single-host pools, and require some additional configuration. These algorithms spread work across available cores on multiple machines connected over the network. The most stable cluster pool is the ipython cluster, which can be accessed as:

>>> pool = glimpse.pools.GetClusterPackage("ipython").MakePool(config_file)
>>> pool.map(hex, [1, 2, 3])
['0x1', '0x2', '0x3']

3.1. Low-Level Code
3.2 Experiments

A central data structure in Glimpse is the `ExperimentData` class, which records everything needed to identify how an experiment was conducted and what results were found. The experimental protocol includes the choice of hierarchical model and its parameters, the corpus of images, and possibly the training and testing splits (if chosen manually). The experimental results include the set of S2 prototypes, the features extracted by the model from the images, the training and testing splits (if chosen automatically), the classifier used, and its performance on the task.

The `glimpse.experiment` package implements the `ExperimentData` data structure, as well as functions that operate on this structure to conduct an experiment. The `glimpse.glab` package provides high-level, pseudo-declarative interfaces for specifying and running experiments with a minimum of programming. The rest of the Glimpse library is composed of the low-level components needed to run an experiment, such as hierarchical model implementations in `glimpse.models`, “worker pools” for parallel computation in `glimpse.pools`, and prototype learning algorithms in `glimpse.prototypes`.

3.3 GLAB API

The glab API is documented extensively in the user guide, and in the API and command-line references.

- Future Work
Low-level modules:

4.1 glimpse.backends

ACTIVATION_DTYPE
Element type for an array of Glimpse activation values.

DEFAULT_BACKEND_NAME
Backend name used by MakeBackend when no name is supplied.

4.1.1 base_backend Module

4.1.2 scipy_backend Module

4.2 glimpse.models

DEFAULT_MODEL_NAME
Model name used by GetModelClass(), MakeModel(), and MakeParams() when no name is supplied. This should not be "base".

4.2.1 Subpackages

glimpse.models.base

Subpackages

glimpse.models.base.layer
glimpse.models.ml

4.3 glimpse.prototypes

4.4 glimpse.pools

DEFAULT_POOL_TYPE
DEFAULT_CLUSTER_TYPE

4.4.1 Subpackages

glimpse.pools.ipythoncluster

Experiment Code:

4.5 glimpse.experiment

4.5.1 Data Structures

4.5.2 Running Experiments

4.5.3 Prototype Algorithms

4.5.4 Analyzing Results

4.5.5 Miscellaneous

4.5.6 Subpackages

glimpse.experiment.mf_wkmeans

glimpse.experiment.om_wkmeans

GLAB API:

4.6 glimpse.glab.api

DEFAULT_LAYER
Default model layer to use for evaluation.
Fig. 4.1: Figure 1: A screenshot of the editor for model parameters.
4.7 Command-Line Interface

The `glab` command provides a high-level interface to run experiments from the command line. The command-line interface to the GLAB API can be run as follows.

```
$ glab [ARGUMENTS]
```

**Note:** The location of the `glab` script is controlled by the `pip` installer. On Linux, this location is `$HOME/.local/bin`. If `glab` can not be found, try adding this location to your system path.

As an alternative, this script can always be run as:

```
$ python -m glimpse.glab.cli [ARGUMENTS]
```

4.7.1 Arguments

Arguments to this command are described below, where they are grouped depending on their functionality.

**Corpus arguments**

- `--corpus-subdir=DIR, -C DIR` Use DIR as a corpus sub-directory (use `-C` repeatedly to specify all sub-directories in corpus)
- `--corpus=DIR, -c DIR` Use DIR as corpus directory.
- `--balance, -b` Choose equal number of images per class
- `--corpus-name=NAME, --corpus=NAME` Choose the sample corpus `NAME`.

**Extraction arguments**

- `--save-all, -A` Save activity for all layers, rather than just the layers from which features are extracted.
- `--no-activity, -N` Do not compute activity model activity for each image (implies no classifier). This can be used to learn prototypes without immediately evaluating them.
- `--num-prototypes=NUM, -n NUM` Generate `NUM` S2 prototypes (default: 10)
- `--options=FILE, -O FILE` Read model options from `FILE` (overrides `-p`)

**Prototype learning arguments**

- `--base-weight=w` Add a weight of `w` for all training patches (default: 0.0)
- `--high=VALUE` Use `VALUE` as the high end of uniform distribution for random prototypes (default: 1.0)
- `--low=VALUE` Use `VALUE` as the low end of uniform distribution for random prototypes (default: 0.0)
- `--masks=DIR` Set mask directory for `object_mask_wkmeans` to `DIR` (default: '')
- `--prototype-algorithm=ALG, -P FILE` Use prototype learning algorithm `ALG` (one of: histogram, ica, imprint, kmeans, kmedoids, meta_feature_wkmeans, nearest_kmeans, nmf, normal, object_mask_wkmeans, pca, shuffle, sparse_pca, uniform)
--regr-samples=NUM  Sample NUM patches when training regression model for meta_feature_wkmeans (default: 0)
--samples=NUM  Sample NUM training patches for S2 prototype learning (default: 0)

Evaluation arguments
-E, --evaluate  Train and test a classifier

-f NUM, --num-folds=NUM  Use NUM folds for cross-validation (default: 10)

-H, --hist-features  Use histograms (accumulated over space and scale) for each feature band (requires spatial features, such as C1)

-l LAYER, --layer=LAYER  Extract image features from model layer LAYER (default: ‘C2’)

-L ALG, --learner=ALG  Use learning algorithm ALG to use for classification. ALG can be a Python expression, or one of ‘svm’ or ‘logreg’) (default: ‘svm’)
--predictions  Print the classifier’s per-image predictions.

-S FUNC, --score-function=FUNC  Use scoring function FUNC for classifier evaluation (one of: accuracy or auc) (default: ‘accuracy’)

-T SIZE, --train-size=SIZE  Set the size of the training set to SIZE (number of instances or fraction of total)
-x, --cross-validate  Compute test accuracy via (10x10-way)cross-validation instead of fixed training/testing split

Other arguments
--command=cmd  Execute cmd after running the experiment (but before results are saved)

-h, --help  Print this help and exit
-i FILE, --input=FILE  Read initial experiment data from FILE.
-o FILE, --output=FILE  Store results to FILE
-t TYPE, --pool-type=TYPE  Set the worker pool type to TYPE (one of: s, singlecore, m, multicore, c, cluster)
-v, --verbose  Enable verbose logging

4.7.2 Examples
Evaluate with a logistic regression classifier using 100 C2 features based on imprinted prototypes.

```$ glab -v -n 100 -p imprint --corpus-name easy --learner=logreg -E --predictions
INFO:root:Reading class sub-directories from: easy
INFO:root:Reading images from class directories: ['easy/circle', 'easy/cross']
INFO:root:Using pool: MulticorePool
INFO:root:Learning 2 prototypes at 1 sizes from 4 images by imprinting
Time: 0:00:01 |##################################################| Speed: 3.10 unit/s
INFO:root:Learning prototypes took 1.294s
INFO:root:Computing C2 activation maps for 8 images
Time: 0:00:02 |##################################################| Speed: 3.32 unit/s
INFO:root:Computing activation maps took 1.712s
INFO:root:Evaluating classifier on fixed train/test split on 10 images using 2 features from layer(s)
INFO:root:Training on 4 images took 0.003s
```
INFO:root:Classifier is Pipeline(learner=LogisticRegression([...OUTPUT REMOVED...]))
INFO:root:Classifier accuracy on training set is 1.000000
INFO:root:Scoring on training set (4 images) took 0.001s
INFO:root:Scoring on testing set (6 images) took 0.001s
INFO:root:Classifier accuracy on test set is 0.833333

Classifier Predictions
======================
Each line gives the true and predicted labels (in that order) for an image in the corpus.

Training Set Predictions
------------------------
easy/circle/4_circle_r-25s0.834y-14x-06_white.jpg circle circle
easy/circle/5_circle_r-05s0.744y011x-16_white.jpg circle circle
easy/cross/2_cross_r015s0.473y025x-12_white.jpg cross cross
easy/cross/3_cross_r000s0.496y022x-22_white.jpg cross cross

Test Set Predictions
---------------------
easy/circle/4_circle_r-25s0.834y-14x-06_white.jpg circle circle
easy/circle/5_circle_r-05s0.744y011x-16_white.jpg circle circle
easy/cross/2_cross_r015s0.473y025x-12_white.jpg cross circle
easy/cross/3_cross_r000s0.496y022x-22_white.jpg cross cross
Indices and tables

- genindex
- modindex
- search
g

- glimpse.backends, 23
- glimpse.backends.base_backend, 23
- glimpse.backends.scipy_backend, 23
- glimpse.experiment.mf_wkmeans, 24
- glimpse.experiment.om_wkmeans, 24
- glimpse.glab.api, 24
- glimpse.glab.cli, 26
- glimpse.models, 23
- glimpse.models.base, 23
- glimpse.models.base.layer, 23
- glimpse.models.ml, 24
- glimpse.pools, 24
- glimpse.pools.ipythoncluster, 24
- glimpse.prototypes, 24