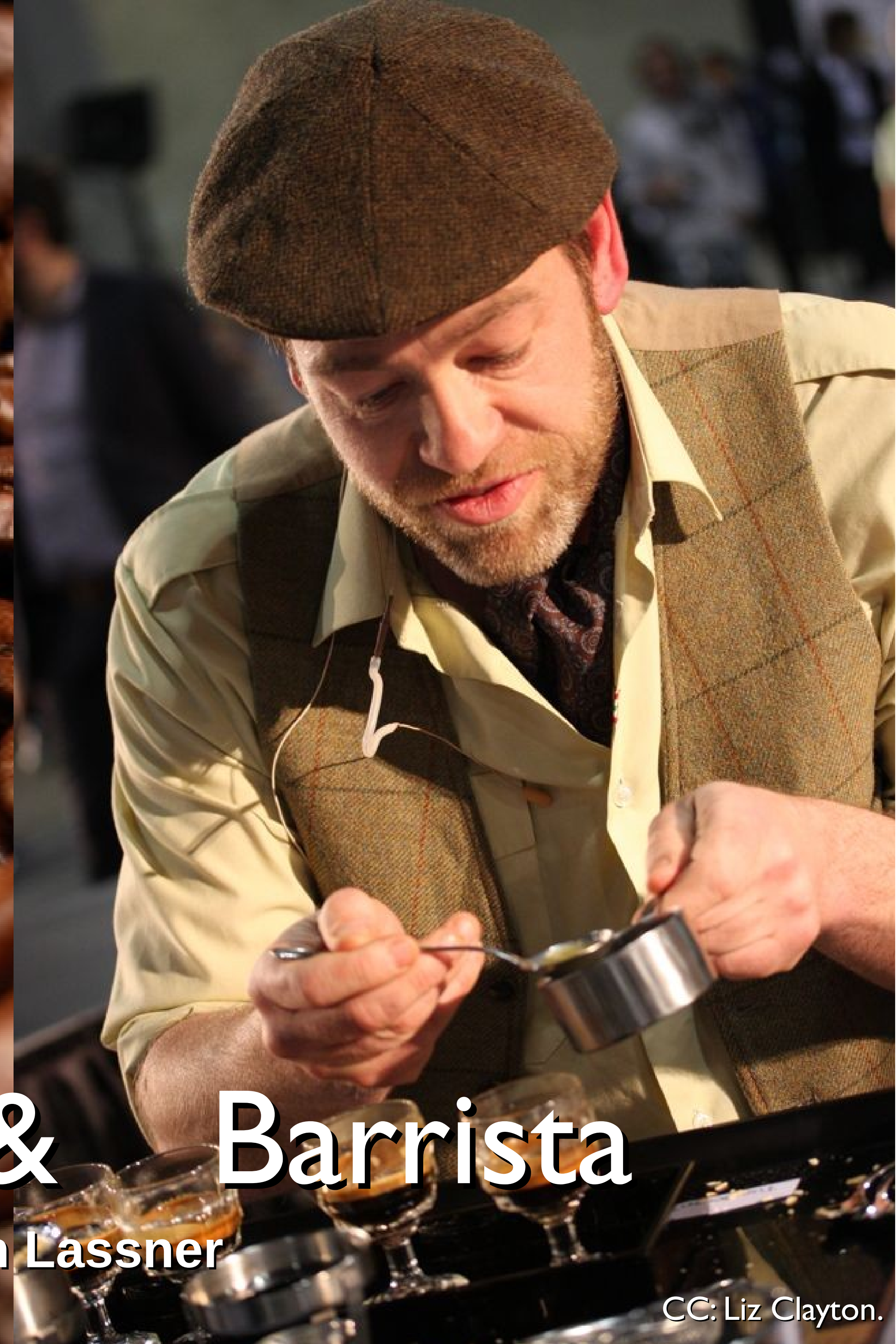

Caffe & Barrista

Christoph Lassner

Why caffe?

Competing frameworks:

- Neon,
- Torch,
- Chainer,
- Theano,
- TensorFlow,
- Caffe

Hard numbers on performance

Currently:

https://github.com/soumith/convnet-benchmarks

Starting June 16^{th} 2016
(according to website, still not available as of today):

Deepmark

Caffe - facts

- From U.C. Berkeley,
- Available at http://caffe.berkeleyvision.org,
- Written in C++ with basic Python and MATLAB bindings.

Pros:

- Especially easy to use for finetuning models,
- Many models available, including ‘exotic’ layers.

Cons:

- Tons of dependencies (though easy to get on Debian/Ubuntu),
- It’s not really DRY.

Caffe – ‘unique’ features

A full model including it’s training is stored in three parts:

- The model description (e.g., resnet-train.prototxt),
- The model parameters (e.g., resnet.caffemodel),
- The solver (i.e., optimizer) parameters (e.g., training-stage1.prototxt).

The `.prototxt` files are in a caffe-specific google protobuf format.

This has the advantages that:

- The **model and the parameters** are stored **separately**,
- The model specification is **human-readable** and adjustable, which means that models can be altered without programming, but just editing a textfile,
- Which allows **finetuning or training** a model can mostly be done on the **command line.** However this means that,
- **No procedural model generation** is possible and **consistency problems** arise.

Making the most out of caffe – the barrista

- Completely pythonic interface,
  as for *keras/lasagne*, but using standard *caffe*,
- Protobuf object introspection allows for automatic full compatibility to your caffe version, including custom layers,
- No more need to edit *.prototxt* files, though they are fully supported,
- Automatic setup of *fit* and *predict* network configurations,
- Transparent split of data preparation tasks, with built-in support for automatic resizing and padding of images and corresponding output extraction,
- Built-in support for sliding window prediction,
- Monitoring and plotting capabilities included.

CC: Liz Clayton.

Making the most out of caffe – the barrista

- Specify network architectures conveniently:

```
import barrista.design as ds
netspec = ds.NetSpecification([[10, 3, 51, 51], [10]],
                              # batchsize 10, 3 dim. of 51x51 signal, 10 labels
                              inputs=['data', 'annotations'])
netspec.layers.append(ds.ConvolutionLayer(Convolution_kernel_size=3,
                                          Convolution_pad=1,
                                          Convolution_num_output=1))
# The layers are wired together automatically, unless you specify something else:
netspec.layers.append(ds.InnerProductLayer(tops=['net_out'],
                                           InnerProduct_num_output=10))
netspec.layers.append(ds.SoftmaxWithLossLayer(bottoms=['net_out',
                                                       'annotations']))
net = netspec.instantiate()
```

- Train networks:

```
import barrista.solver
net.fit(1000,
        barrista.solver.SGDSolver(base_lr=0.01),
        X={'data': np.ones((21, 3, 51, 51)),  # Automatically batched.
           'annotations': np.zeros((21,))})
net.predict({'data': np.zeros((5, 3, 51, 51)), [...]})
```

Making the most out of caffe – the barrista

- Use monitors for fine training control:

```
import barrista.monitoring
net.fit(# ... as before
        train_callbacks=[
          # Write the network weights every 100 iterations to disk.
          barrista.monitoring.Checkpointer('/tmp', 100),
          # Get a progress bar with ETA.
          barrista.monitoring.ProgressIndicator()])
```

- Store and load models in a fully compatible way:

```
netspec.to_prototxt(output_filename='net.prototxt')
net.save('net.caffemodel')  # Save the weights.
new_netspec = ds.NetSpecification.from_prototxt(filename='net.prototxt')
new_network = new_netspec.instantiate()
new_network.load_blobs_from('net.caffemodel')  # Load the weights.
```

Making the most out of caffe – the barrista

- Use monitors for fine training control:

```
import barrista.monitoring
net.fit(# ... as before
        train_callbacks=[
          # Write the network weights every 100 iterations to disk.
          barrista.monitoring.Checkpointer('/tmp', 100),
          # Get a progress bar with ETA.
          barrista.monitoring.ProgressIndicator()])
```

- Store and load models in a fully compatible way:

```
netspec.to_prototxt(output_filename='net.prototxt')
net.save('net.caffemodel')  # Save the weights.
new_netspec = ds.NetSpecification.from_prototxt(filename='net.prototxt')
new_network = new_netspec.instantiate()
new_network.load_blobs_from('net.caffemodel')  # Load the weights.
```

Practical session overview

Ready-to-use VirtualBox images are available!

- Session I (caffe)
  - Get to **know your way around** the caffe source & compilation.
  - The core caffe **API concepts**, specifying networks.
    - Specifying a logistic regression **network for MNIST**.
  - **Data layers** and **solvers**.
    - Fitting a logistic **regression to MNIST**.
  - **Finetuning** an existing network.
    - Finetuning the fitted regression model with one additional layer.
  - Bonus: **Adding a new layer type** to a caffe installation.

- Session II (barrista)
  - Get to **know your way around** the barrista source.
  - The core **barrista concepts**, specifying networks.
    - Specifying a linear unit.
  - **Fitting and testing** a network.
    - Fitting and visualizing the linear unit.
    - Extending this to a two-layer non-linear network.
  - Moving towards **deeper networks**.
    - **Procedural generation of ResNets** and stacked linear/pool units,
    - Fitting on **CIFAR10**.
  - **Monitoring** the training.

Thank you for your attention!