CAFFE FRAMEWORK OVERVIEW

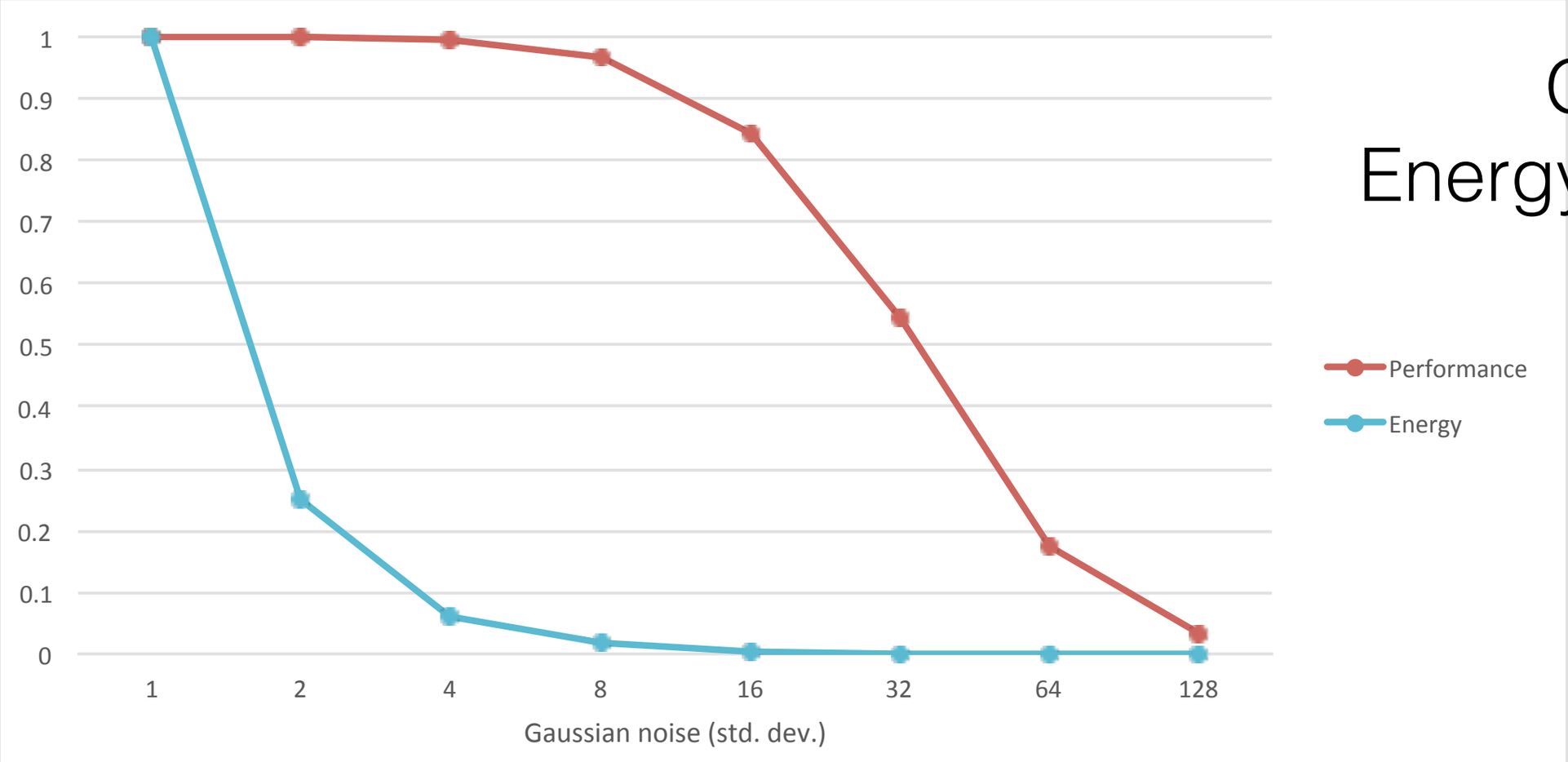
Image classification with Caffe
Performance & energy analysis

Julian Gao, Mia Polansky
Rice University RECG

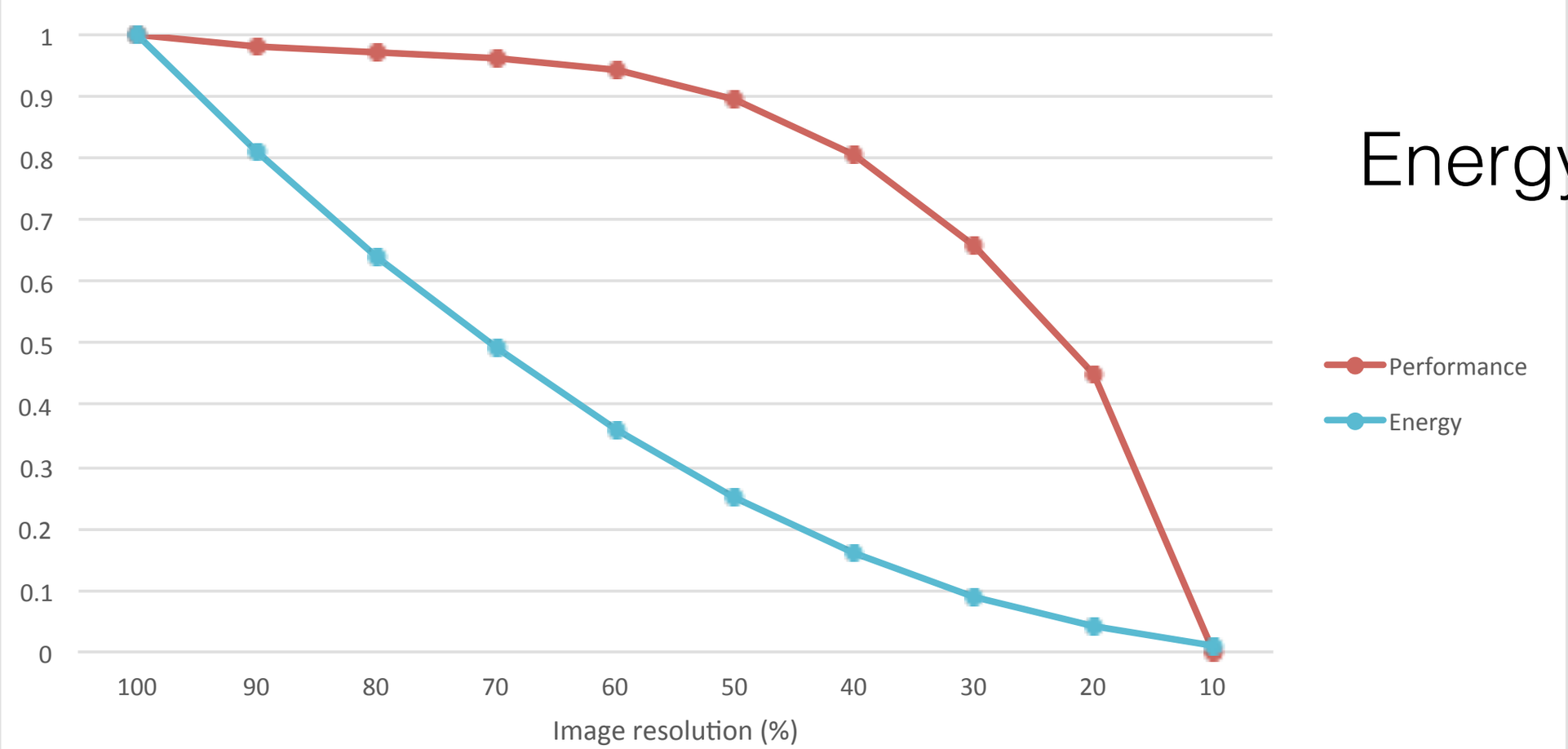
Motivations

- To implement Deep Convolutional Networks (ConvNets) on analog circuits
- To apply analog ConvNets for mobile vision tasks, object detection & classification
- To achieve optimal intersection of energy consumption & task performance

Gaussian Energy-Performance Chart



Resize Energy-Performance Chart



- Lower power input means less energy consumption, but also induces more noise to the circuit
- Signal degradation results in lower classification/detection accuracy
- Goal: to construct a ConvNet that has high noise tolerance, parameters pre-trained prevalent to noise

Introduction to Caffe

- What is Caffe?

A fast, modularized deep learning framework by UC Berkeley PhD student Yangqing Jia.

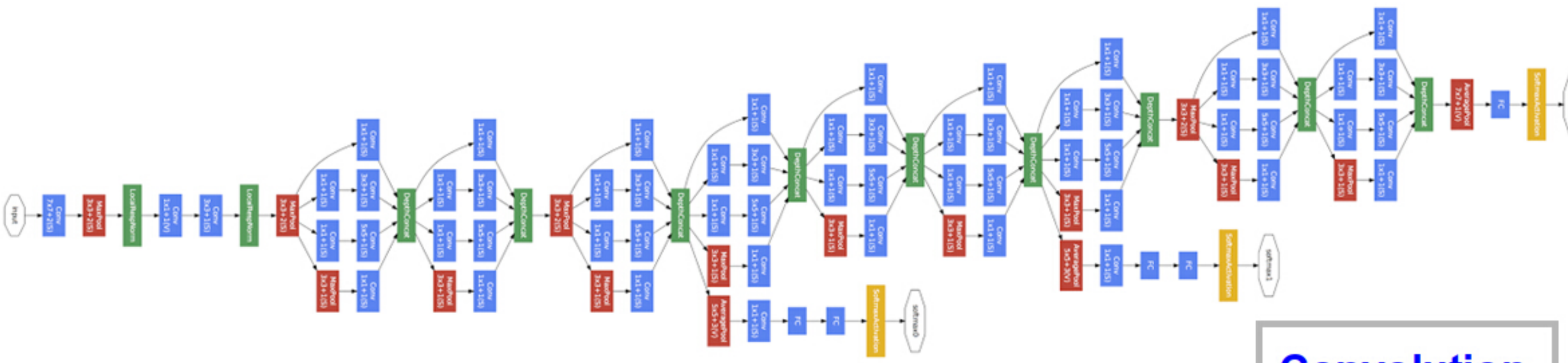
- Why are we using Caffe?

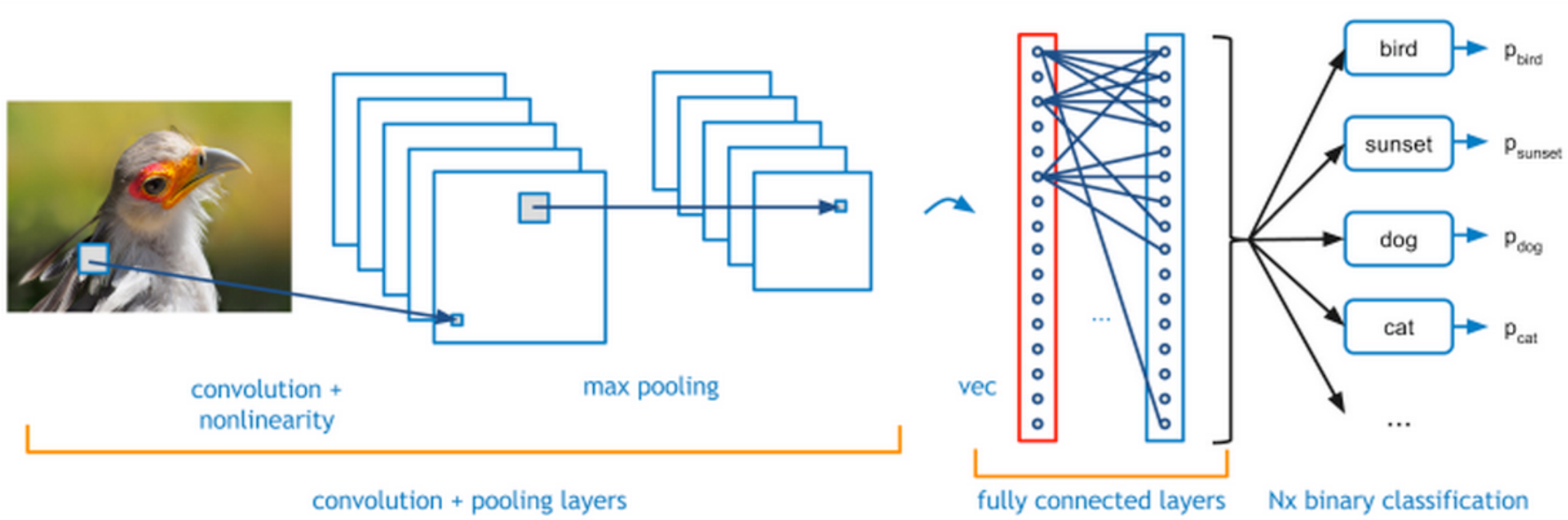
To simulate analog ConvNets for assessment & analysis.

- How to use Caffe?

Run tests, play with source files, construct new models, fine-tune parameters.

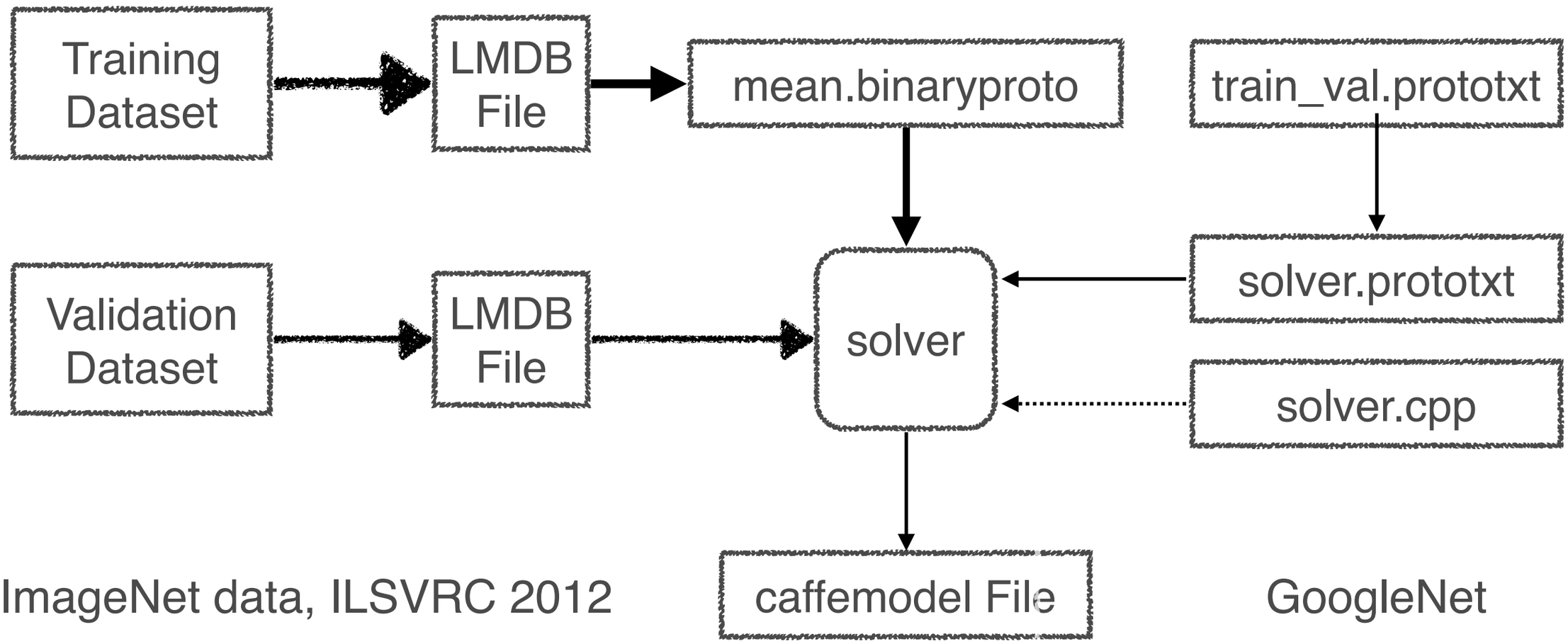
Current ConvNet Model: GoogleNet





How Caffe Works

Training phase:

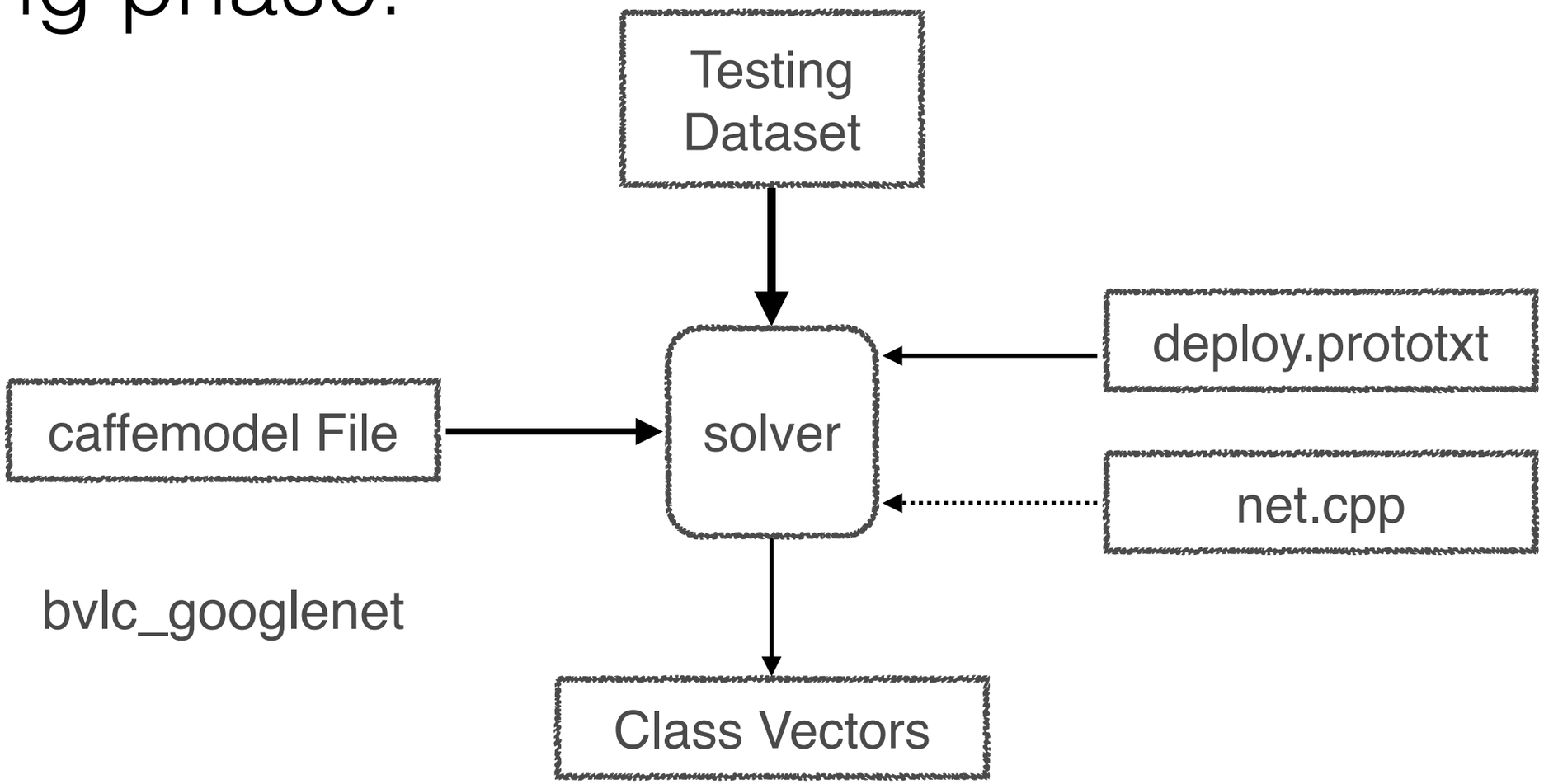


ImageNet data, ILSVRC 2012

GoogleNet

How Caffe Works

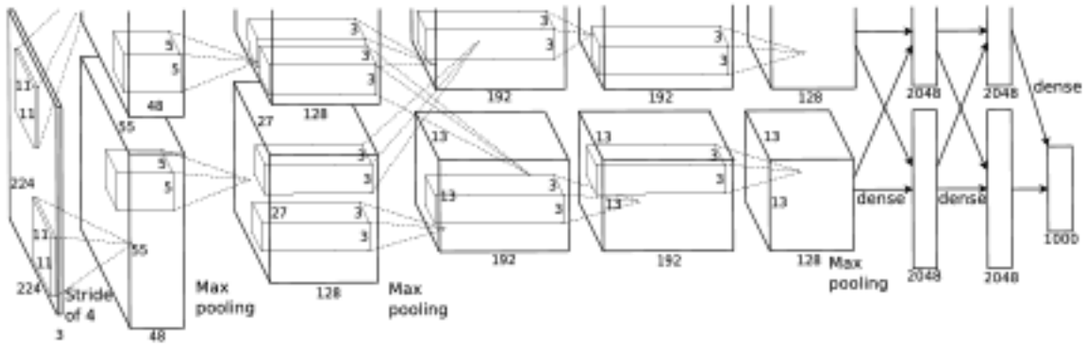
Testing phase:



Class 0-1000:
[[0.01, 0.02, …], [0.8, 0.05, …],
[0.06, 0.42, …], …]

Caffe Loss Function

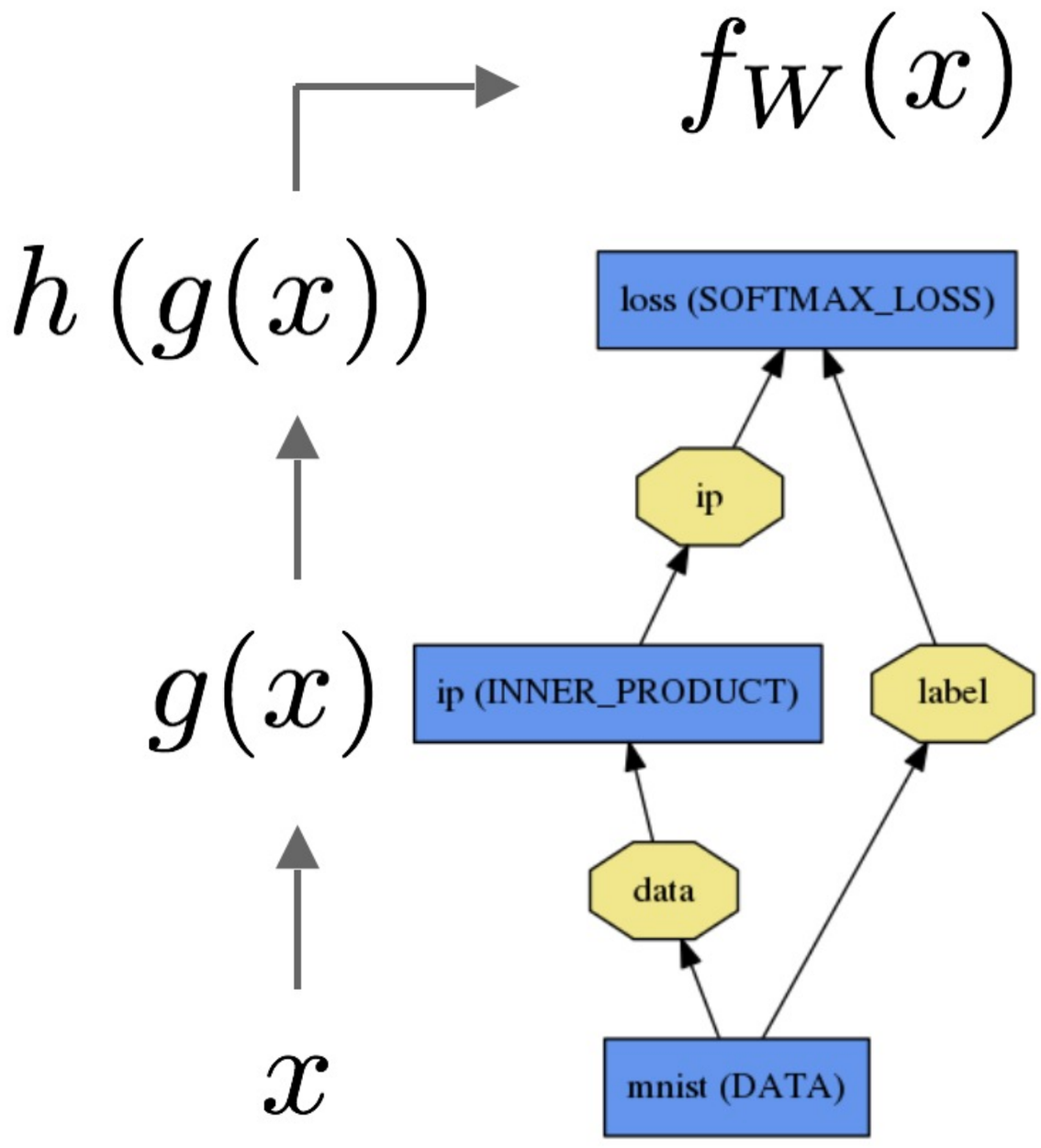
Forward: inference $f_W(x)$



"espresso" + loss

$\nabla f_W(x)$ Backward: learning

Forward and Backward

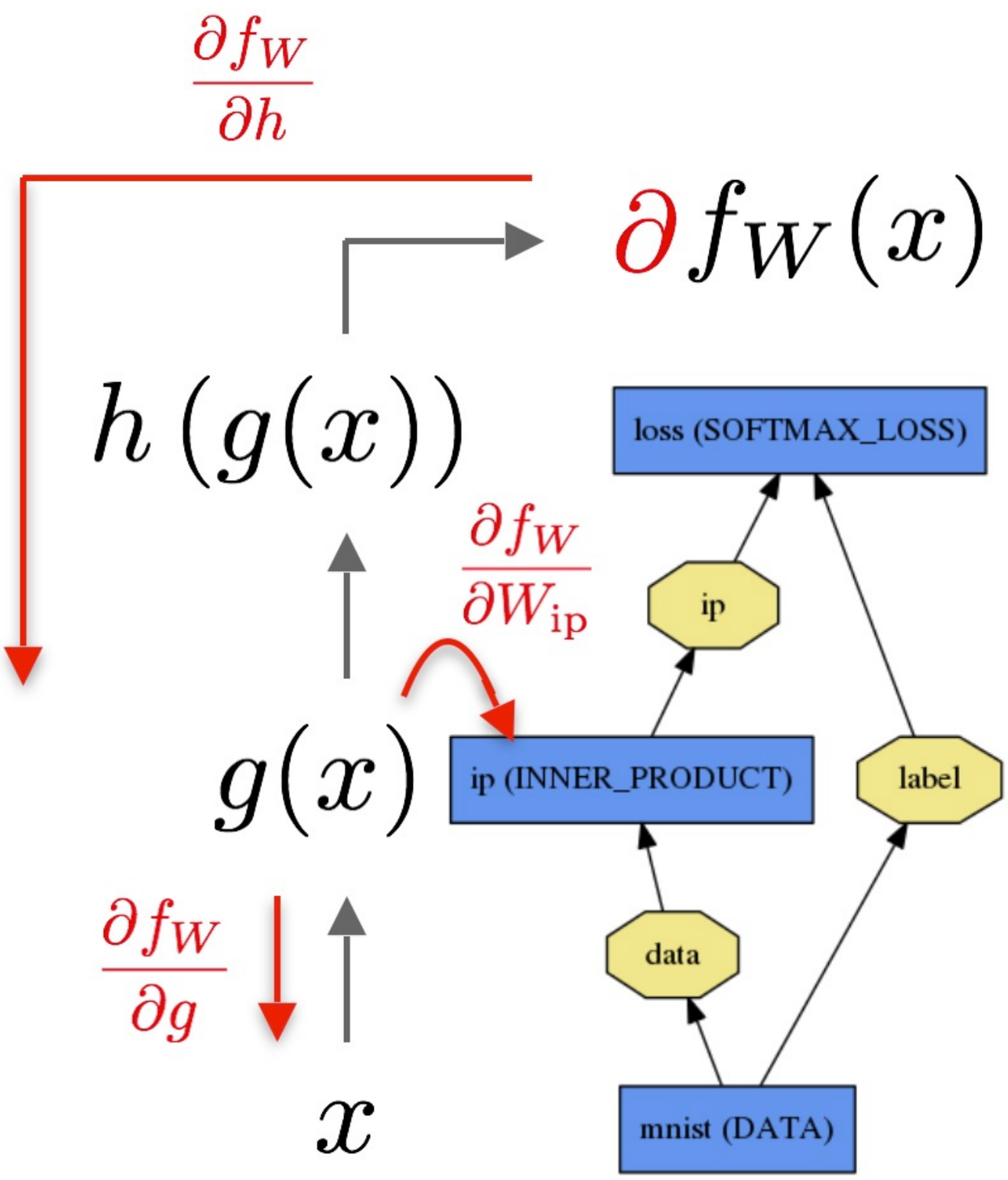


From bottom to top, Caffe computes and stores the output for each layer and generates a final loss value.

The ‘prob’ layer output is the result of prediction.

Forward and Backward

From top to bottom, Caffe back-propagation starts with loss value and computes the gradient layer-by-layer by chain-rule, based on layer output or parameters.



Loss = Performance

```
layer {
  name: "loss1/loss"
  type: "SoftmaxWithLoss"
  bottom: "loss1/classifier"
  bottom: "label"
  top: "loss1/loss1"
  loss_weight: 0.3
}
layer {
  name: "loss1/top-1"
  type: "Accuracy"
  bottom: "loss1/classifier"
  bottom: "label"
  top: "loss1/top-1"
  include {
    phase: TEST
  }
}
layer {
  name: "loss1/top-5"
  type: "Accuracy"
  bottom: "loss1/classifier"
  bottom: "label"
  top: "loss1/top-5"
  include {
    phase: TEST
  }
  accuracy_param {
    top_k: 5
  }
}
```

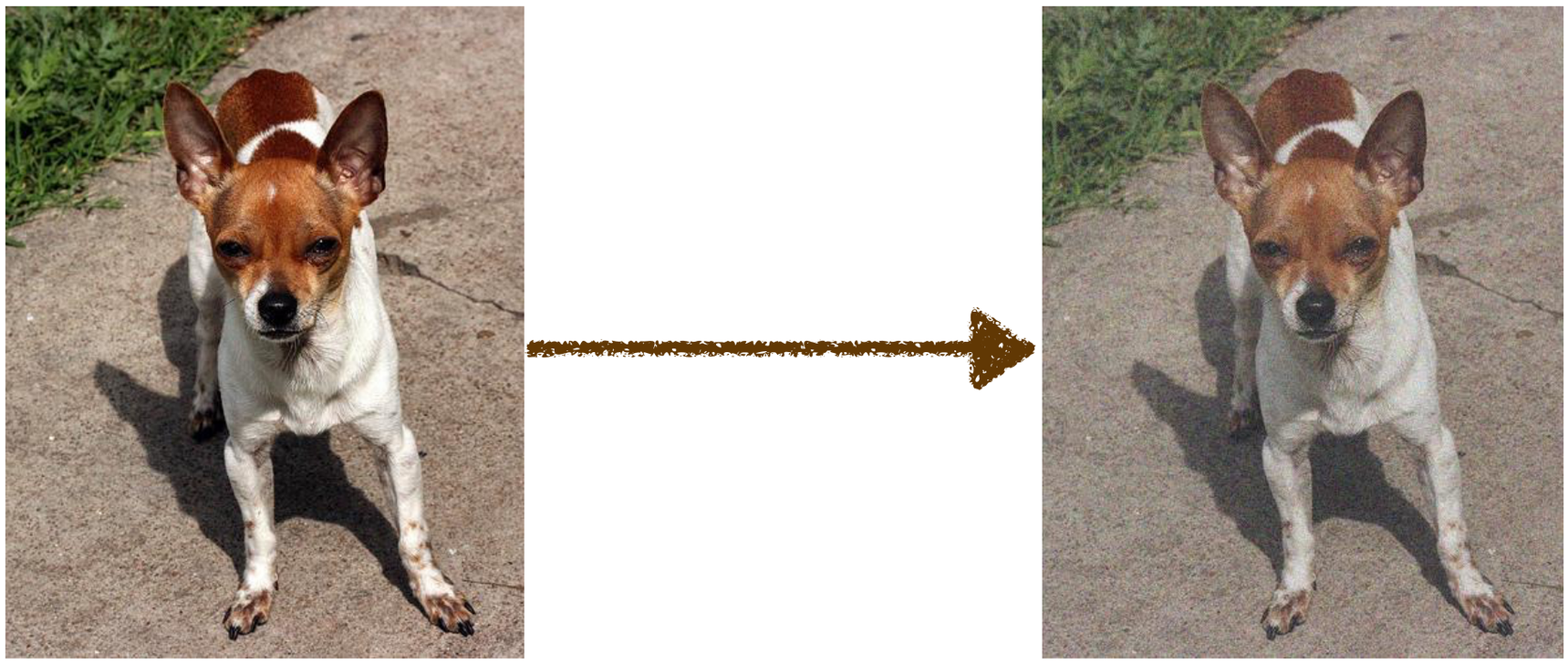
Loss = Performance

+ Energy

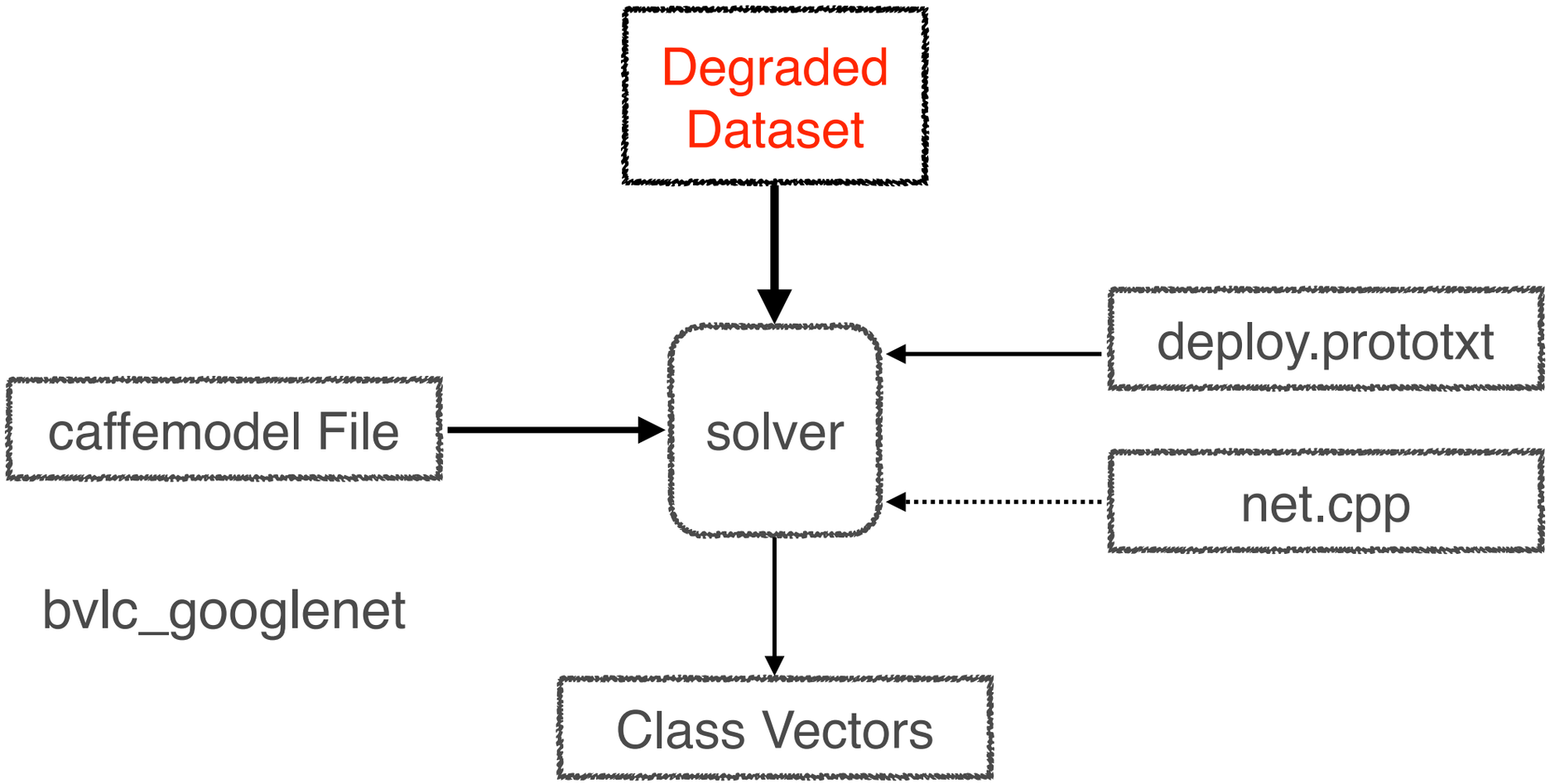
Intuitive Approaches

Before taking energy loss into account, what effect does noise produce on performance?

First of all, what if testing images have low qualities on input level (caffe model is clean)?



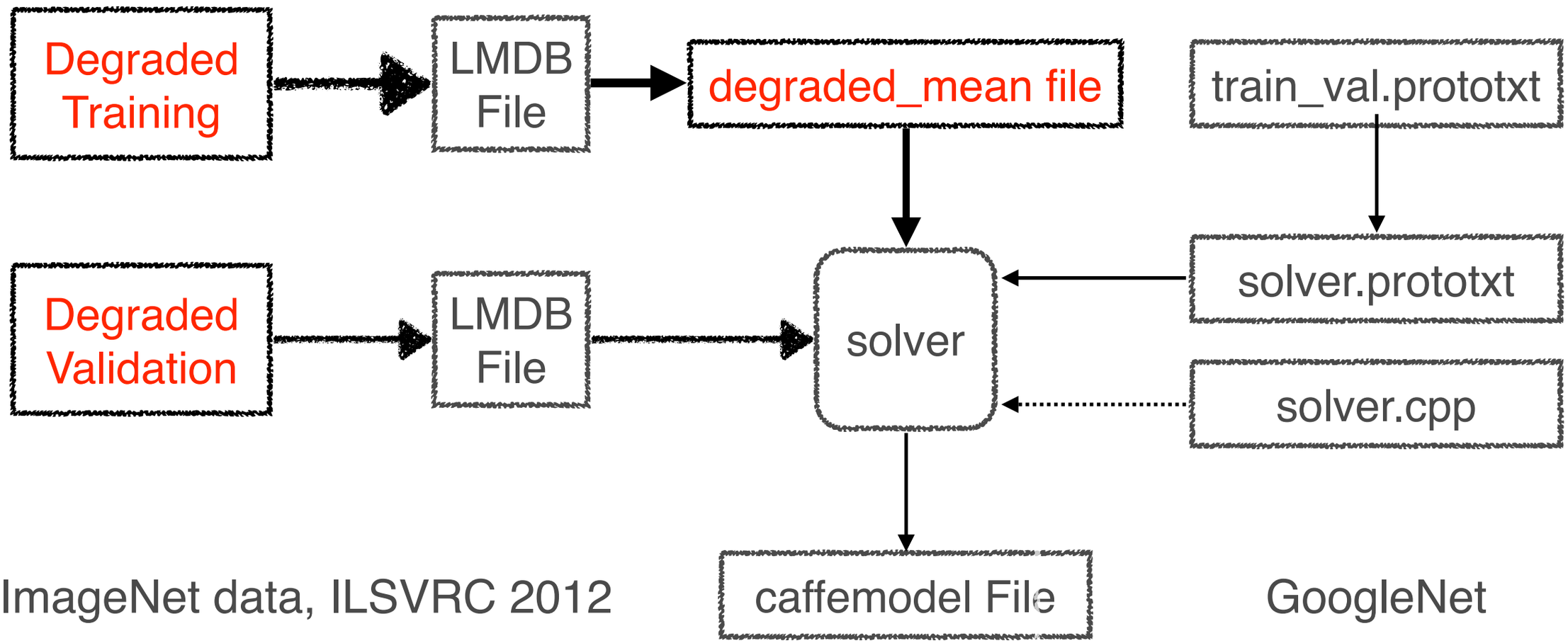
Intuitive Approaches



Class 0-1000:
[[0.01, 0.02, …], [0.8, 0.05, …],
[0.06, 0.42, …], …]

Intuitive Approaches

What would happen if we train a caffe model on noisy inputs?



Footnote: Noise In Circuit

- Gaussian

Source: capacitors, image sensors, wires, literally everywhere

Properties: mean = 0, standard deviation based on formula $v_n = \sqrt{k_B T / C}$

- Poisson (aka shot noise I photon noise)

Source: image sensors

Properties: inherent to the measurement of light, no parameters to be calibrated

- Quantization Error

Source: anywhere involving ADC

Properties: can be treated as additive white noise (uniform)

Intuitive Approaches

Keep incorporating situations. What about different noise types?

Standardize the noise:

- Two types are used, Gaussian (0,σ) for testing and Resize (p%) for comparison

- Degradation parameters: initial value α, number of steps N, step size δ

Intuitive Approaches

What would happen if we train a caffe model on noisy inputs and intermediate values?

Before showing the results, let's define the comparison metrics.

Comparison Metrics

Top1 Accuracy: the matching percentage of argmax from predicted class vectors of clean and degraded testing images. Consider clean image prediction A, and two degraded predictions B and C on a bird image.

Prediction A: [0.02, 0.55, 0.43] -> argmax = 1, bird!

Prediction B: [0.11, 0.47, 0.42] -> argmax = 1, bird!

Prediction C: [0.36, 0.29, 0.35] -> argmax = 2, dog!

B matches A, while C does not. Top1 compares all testing images and generates matching percentage.

Comparison Metrics

Top10 Accuracy: the intersection percentage of top 10 argmax from predicted class vectors of clean and degraded testing images. Case for top 3:

Prediction A: [0.02, 0.15, 0.03, 0.27, 0.53]
Top 3 classes: 1, 3, 4

Prediction B: [0.11, 0.17, 0.02, 0.00, 0.70]
Top 3 classes: 0, 1, 4

1 and 4 intersects. This produces accuracy of 66%.
Case for top 10 accuracy is similar.

Comparison Metrics

Test Condition Rates: the classic four rates of machine learning, performed on thresholded clean (A) and degraded (B) predicted class vectors.

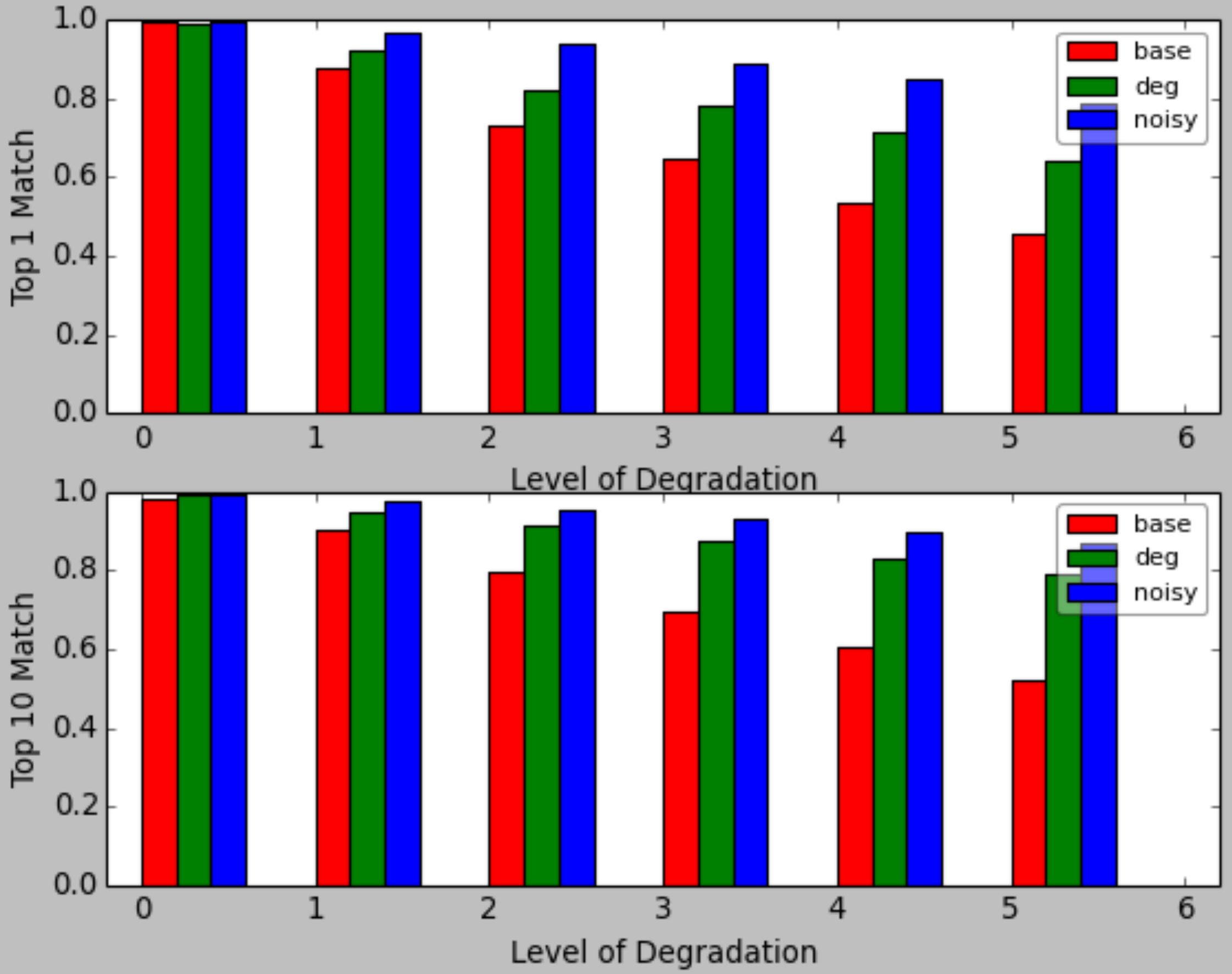
TRUE POSITIVE: (In A & In B) / len(A)

FALSE POSITIVE: (In !A & In B) / len(!A)

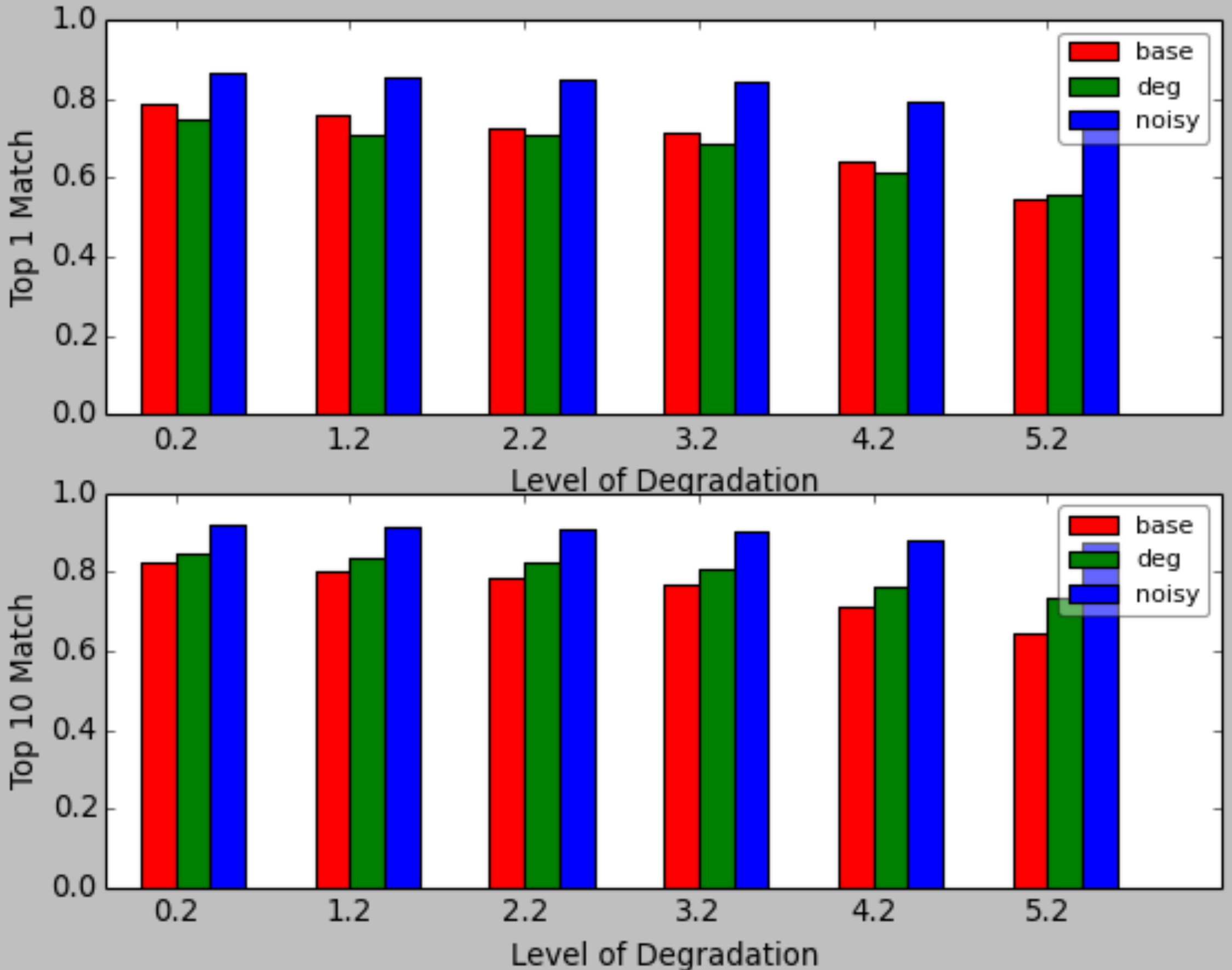
TRUE NEGATIVE: (In A & In !B) / len(A)

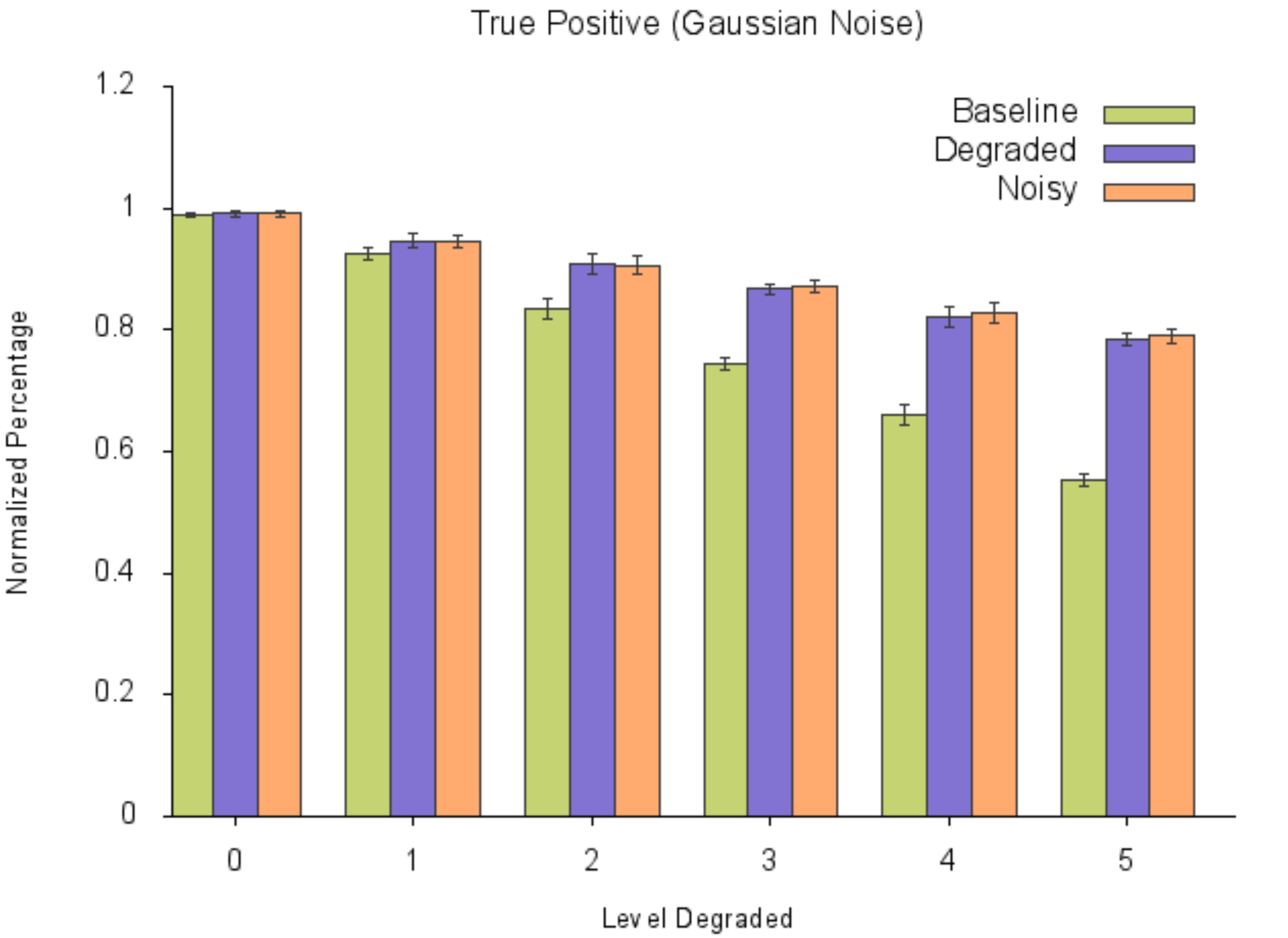
FALSE NEGATIVE: (In !A & In !B) / len(!A)

RESIZE


Top 1 Match
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
base
deg
noisy
Level of Degradation
Top 10 Match
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
base
deg
noisy
Level of Degradation

GAUSSIAN


1.0
0.8
0.6
0.4
0.2
0.0
Top 1 Match
base
deg
noisy
0.2
1.2
2.2
3.2
4.2
5.2
Level of Degradation
1.0
0.8
0.6
0.4
0.2
0.0
Top 10 Match
base
deg
noisy
0.2
1.2
2.2
3.2
4.2
5.2
Level of Degradation


True Positive (Gaussian Noise)
Baseline
Degraded
Noisy
Normalized Percentage
1.2
1
0.8
0.6
0.4
0.2
0
0
1
2
3
4
5
Level Degraded

False Negative (Gaussian Noise)

Baseline
Degraded
Noisy

Normalized Percentage

1.2
1
0.8
0.6
0.4
0.2
0

0
1
2
3
4
5

Level Degraded

True Positive (Resize Noise)

Baseline
Degraded
Noisy

Normalized Percentage

1.2
1
0.8
0.6
0.4
0.2
0

0
1
2
3
4
5

Level Degraded

False Negative (Resize Noise)

Baseline
Degraded
Noisy

Normalized Percentage

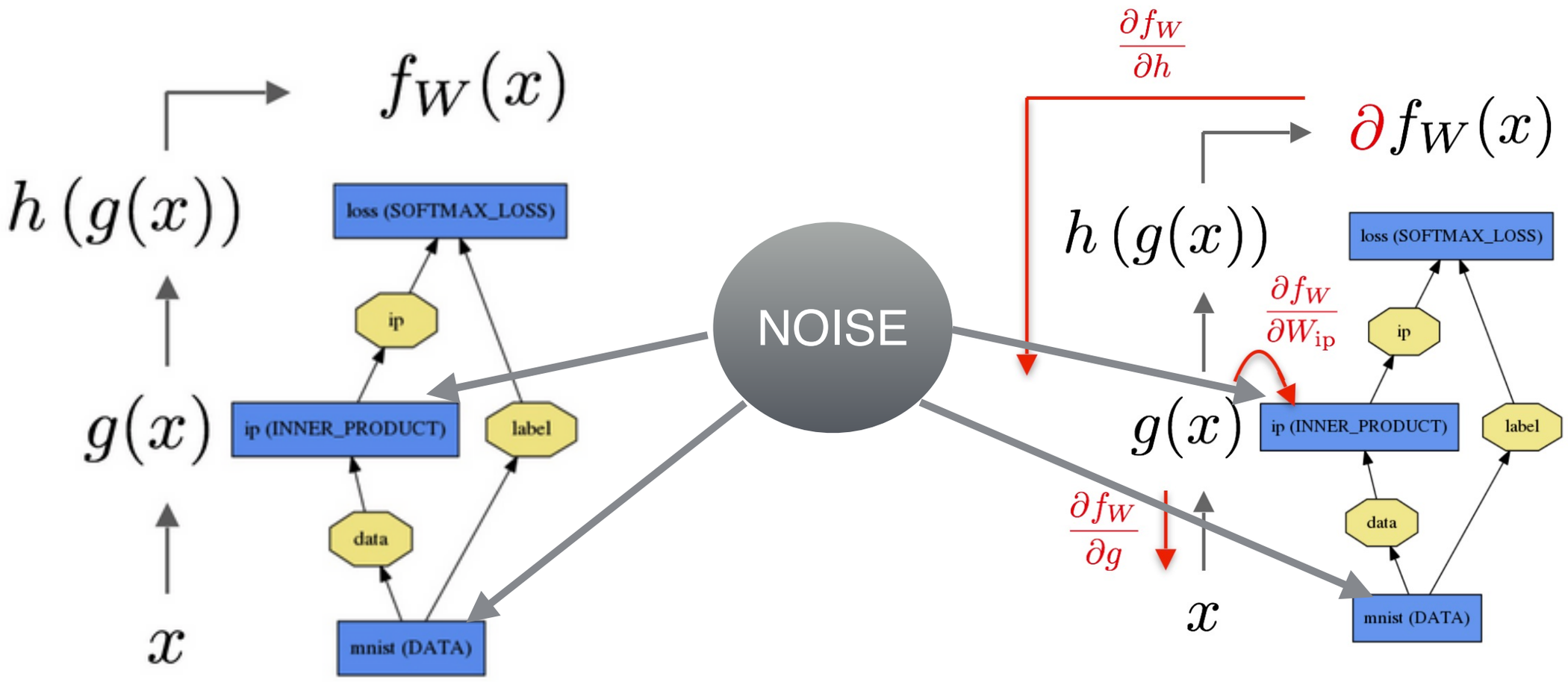
1.2
1
0.8
0.6
0.4
0.2
0

0
1
2
3
4
5

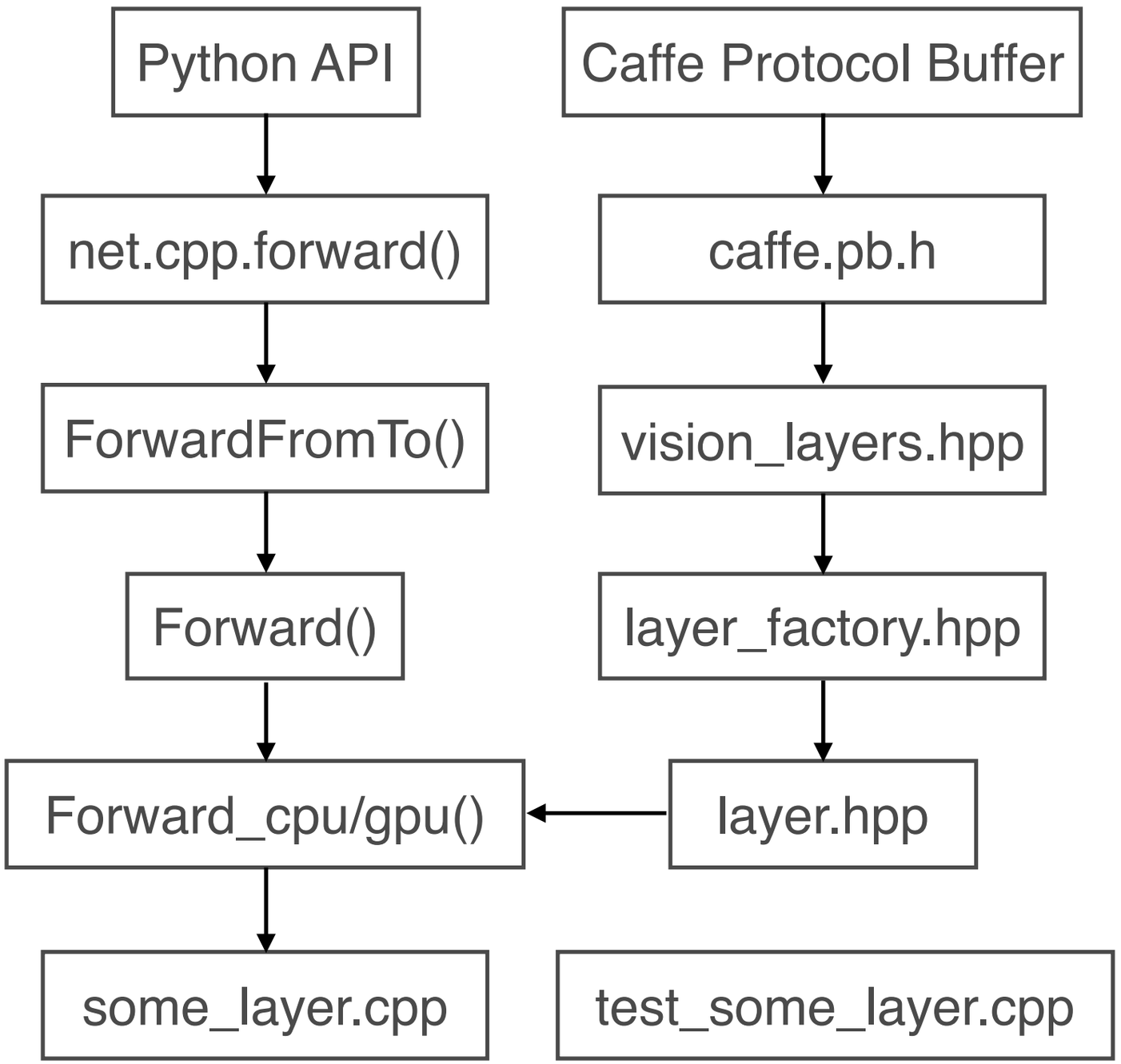
Level Degraded

Intuitive Approaches

Noise Injection technique here (deprecated, but useful to understand the framework architecture):



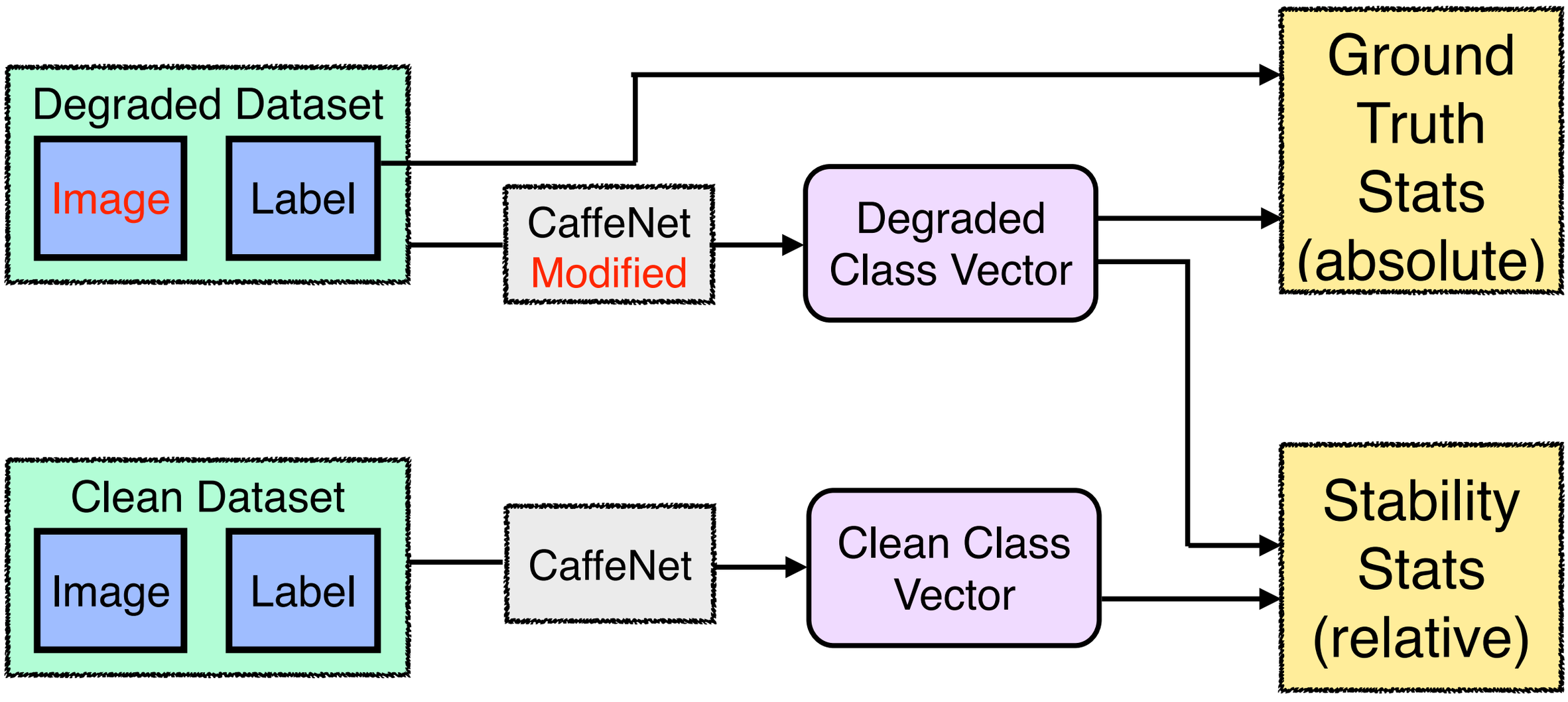
The Caffe Call Stack



Add Noise!

The Complete Workflow (Mia's task)

Now a fully constructed caffe-based testing framework:



The Complete Workflow

Now it's time to dig deeper into caffe. How about constructing our own CaffeNet?

- NoiseLayer: crucial part of the model to simulate the environment of analog circuit

- EnergyLossLayer: our own definition of loss function, and now we made it:

- Loss = Energy + Performance!

The NoiseLayer

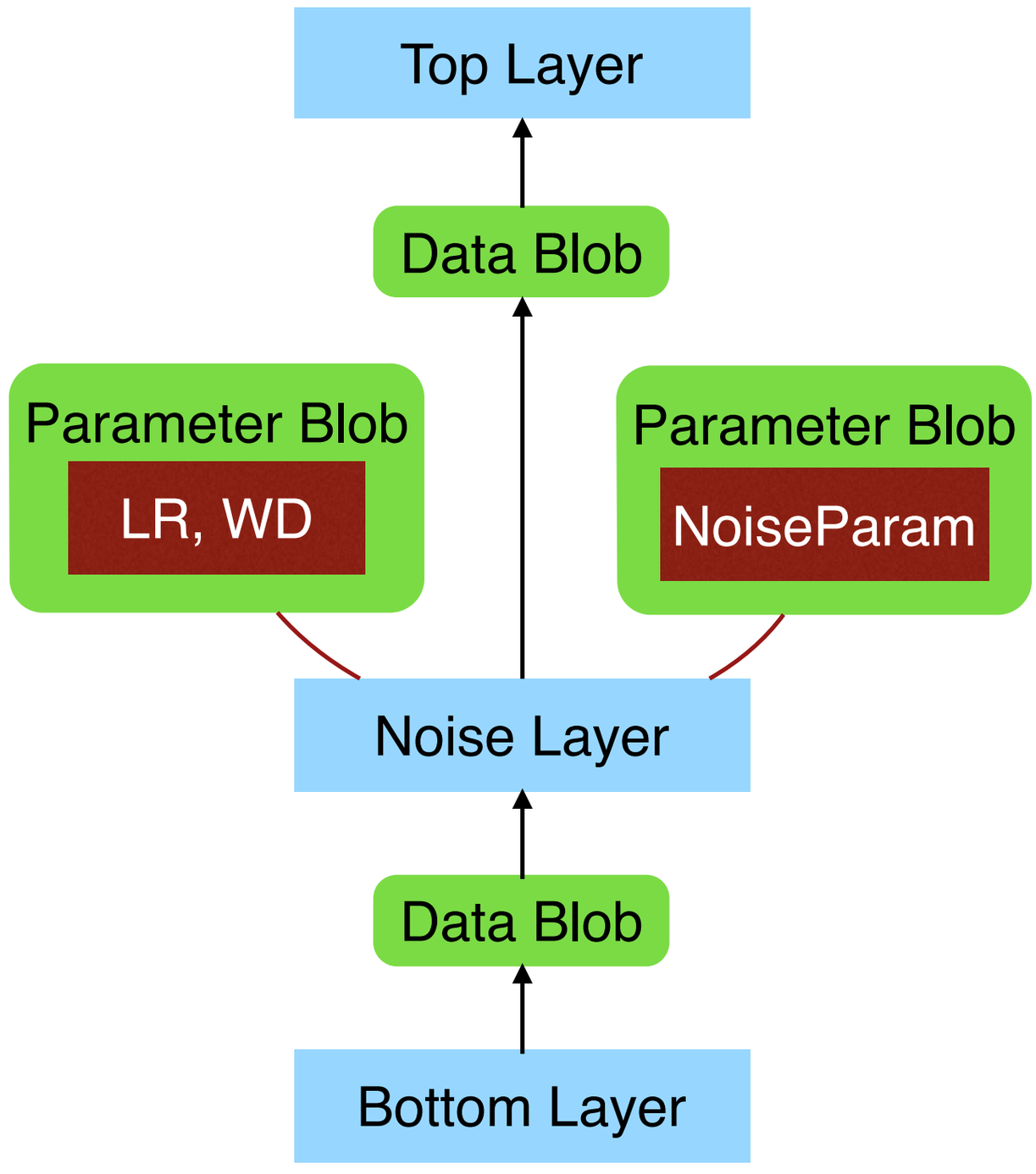
Blob: 4d-array to carry data
(num, channel, height, width)

Data Blob: Transfers data

Parameter Blob: Stores learnable parameters

The NoiseLayer add noise separately to each num and each channel of input data blob with different parameters. Updates parameters each training iteration.

Add to train_val.prototxt

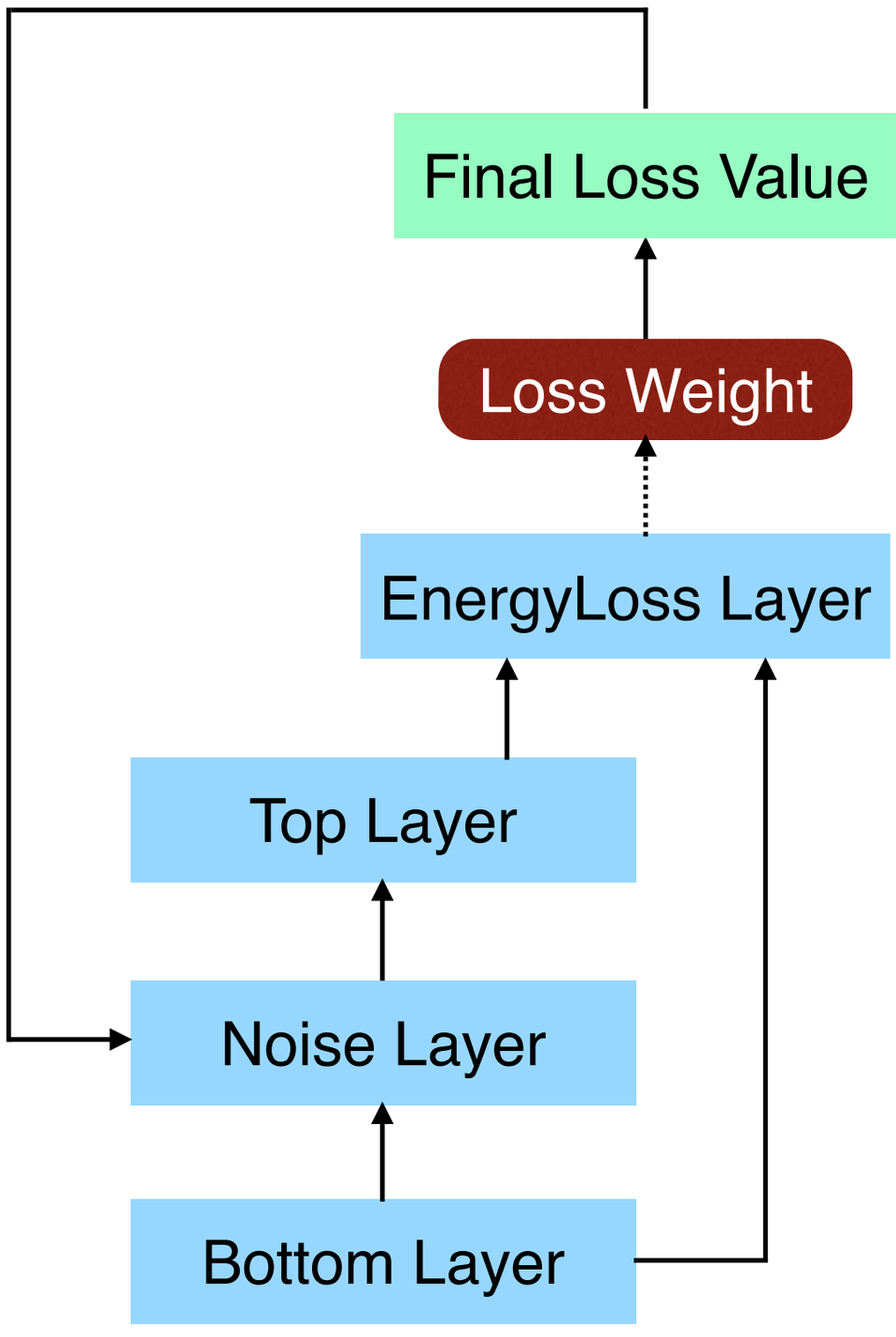


The NoiseLayer

```cpp
#include "caffe/layer.hpp"
#include "caffe/util/im2col.hpp"
#include "caffe/util/math_functions.hpp"
#include "caffe/vision_layers.hpp"

namespace caffe {

template<typename Dtype>
void NoiseLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {

    CHECK_EQ(bottom.size(), 1) << "Can only have one bottom blob input.";
    CHECK_EQ(top.size(), 1) << "Can only have one top blob output.";

    NoiseParameter noise_param = this->layer_param_.noise_param();

    // to store the noise parameters and layer parameters
    this->blobs_.resize(1);
    // different noise parameters for each input number and channel
    this->blobs_[0].reset(new Blob<Dtype>(bottom[0]->num(),
        bottom[0]->channels(), 1, 1));
    // compute the correct noise param size since shape is different
    const int noise_param_size = this->blobs_[0]->count();
    // get the pointer of noise parameter array
    Dtype* blob_ptr = this->blobs_[0].get()->mutable_cpu_data();

    _noise_type = noise_param.ntype();
    _scale = noise_param.scale();
    switch(_noise_type) {
        case NoiseParameter_NoiseType_GAUSSIAN: {
            _mean = noise_param.mean();
            _stddev = noise_param.stddev();
            caffe_set(noise_param_size, _stddev, blob_ptr);
        }   break;
        case NoiseParameter_NoiseType_POISSON: {
            lambda = noise_param.lambda();
```

Define forward(), backward(), and most importantly, update()

The EnergyLossLayer



Unlike NoiseLayer, the EnergyLossLayer takes two bottom layer blobs as input, computes noise amount and relates it with energy consumption, then integrates it into loss value/function.

```
name: "GoogleNet"
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 224
    mean_value: 104
    mean_value: 117
    mean_value: 123
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_train_lmdb"
    batch_size: 32
    backend: LMDB
  }
}
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    mirror: false
    crop_size: 224
    mean_value: 104
    mean_value: 117
    mean_value: 123
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_val_lmdb"
    batch_size: 50
    backend: LMDB
  }
}
layer {
  name: "conv1/7x7_s2"
  type: "Convolution"
```

```
layer {
  name: "data/noise"
  type: "Noise"
  bottom: "data"
  top: "data/noise"
  param {
    lr_mult: 2
    decay_mult: 1.5
  }
  noise_param {
    stddev: 0.0000001
    mean: 0.0
  }

#  loss_weight: 0.5
}

layer {
  name: "conv1/7x7_s2"
  type: "Convolution"
#  bottom: "data"
  bottom: "data/noise"
  top: "conv1/7x7_s2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 64
    pad: 3
    kernel_size: 7
    stride: 2
    weight_filler {
      type: "xavier"
      std: 0.1
    }
    bias_filler {
      type: "constant"
      value: 0.2
    }
  }
}
```

Simply add the specifications to train_val.prototxt

What's Next

- Make NoiseLayer and EnergyLoss Layer work in run time, and updates parameters that minimizes loss

- Implement SIFT in Caffe (on Mia), construct our own CaffeNet