The College Of WILLIAM & MARY

Navigating and Editing Prototxts

Alexander Radovic
College of William and Mary

What are prototxts?

A file format a little like an xml file:
https://developers.google.com/protocol-buffers/docs/overview

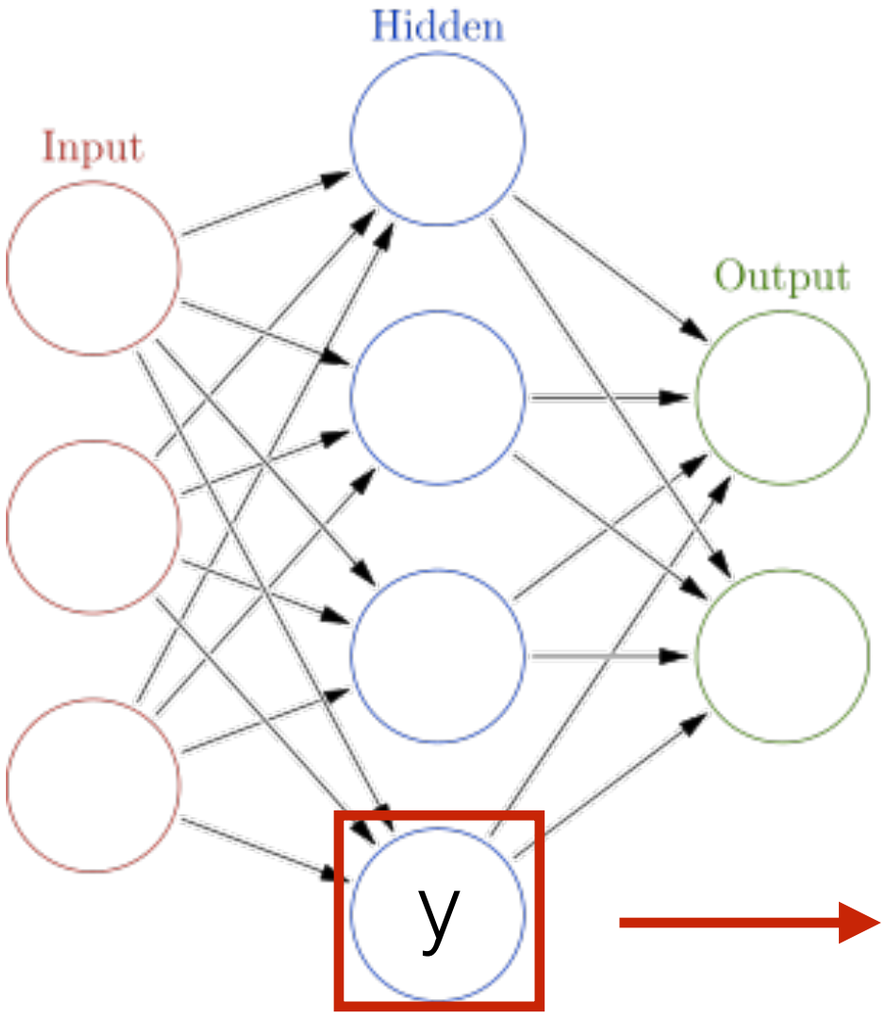
Caffe uses them to define the network architecture, and your training strategy.

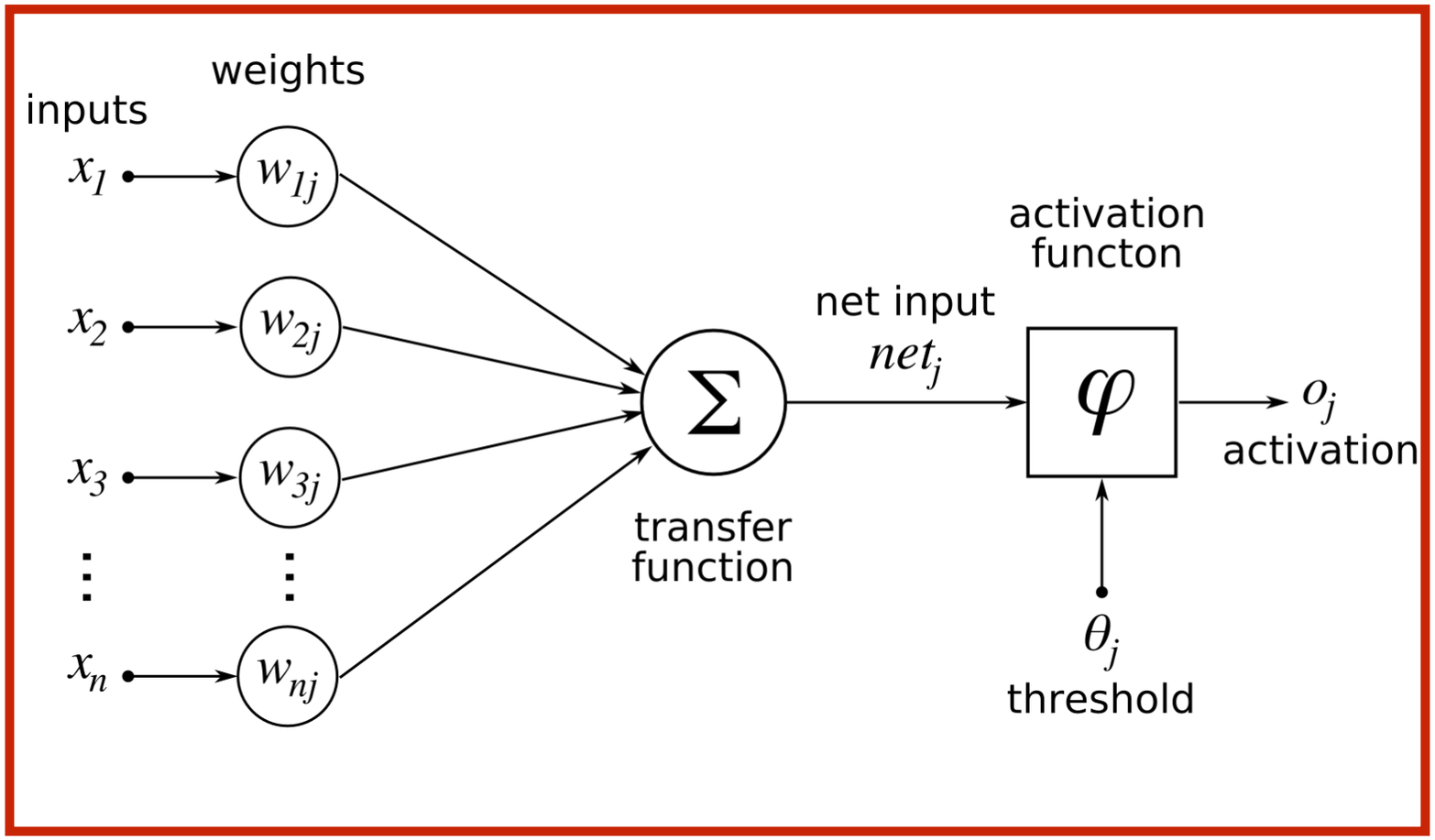
Individual pieces are quite simple, but can become unwieldy/ daunting when you have a large or complex network.

Finding good examples and checking draft networks with visualization tools (http://ethereon.github.io/netscope/#/editor) is the best way not to get stuck.

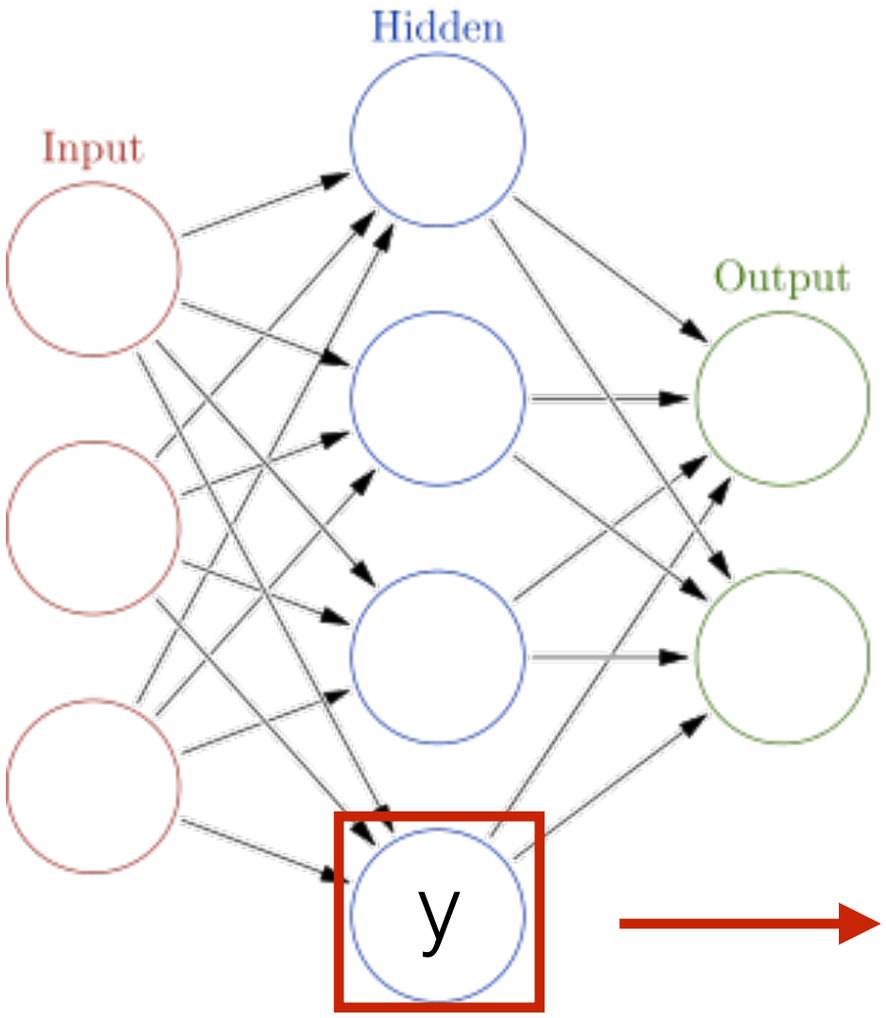
We'll connect a few example snippets to concepts you saw earlier here. then we'll walk through editing some prototxts together.

Neural Networks





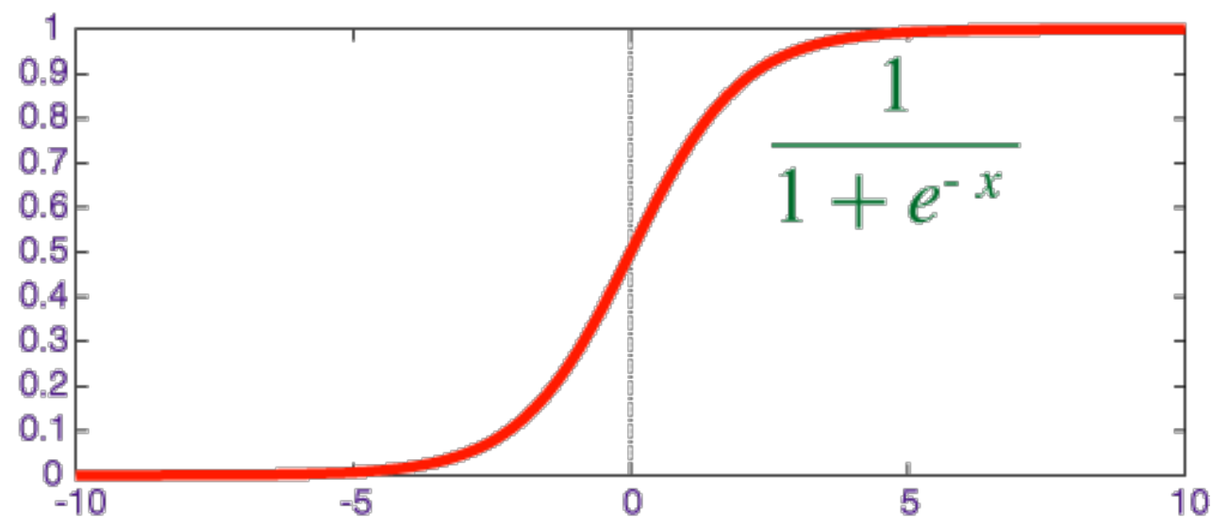
Neural Networks



$$x = input\ vector$$

$$y = \sigma\left(Wx + b\right)$$

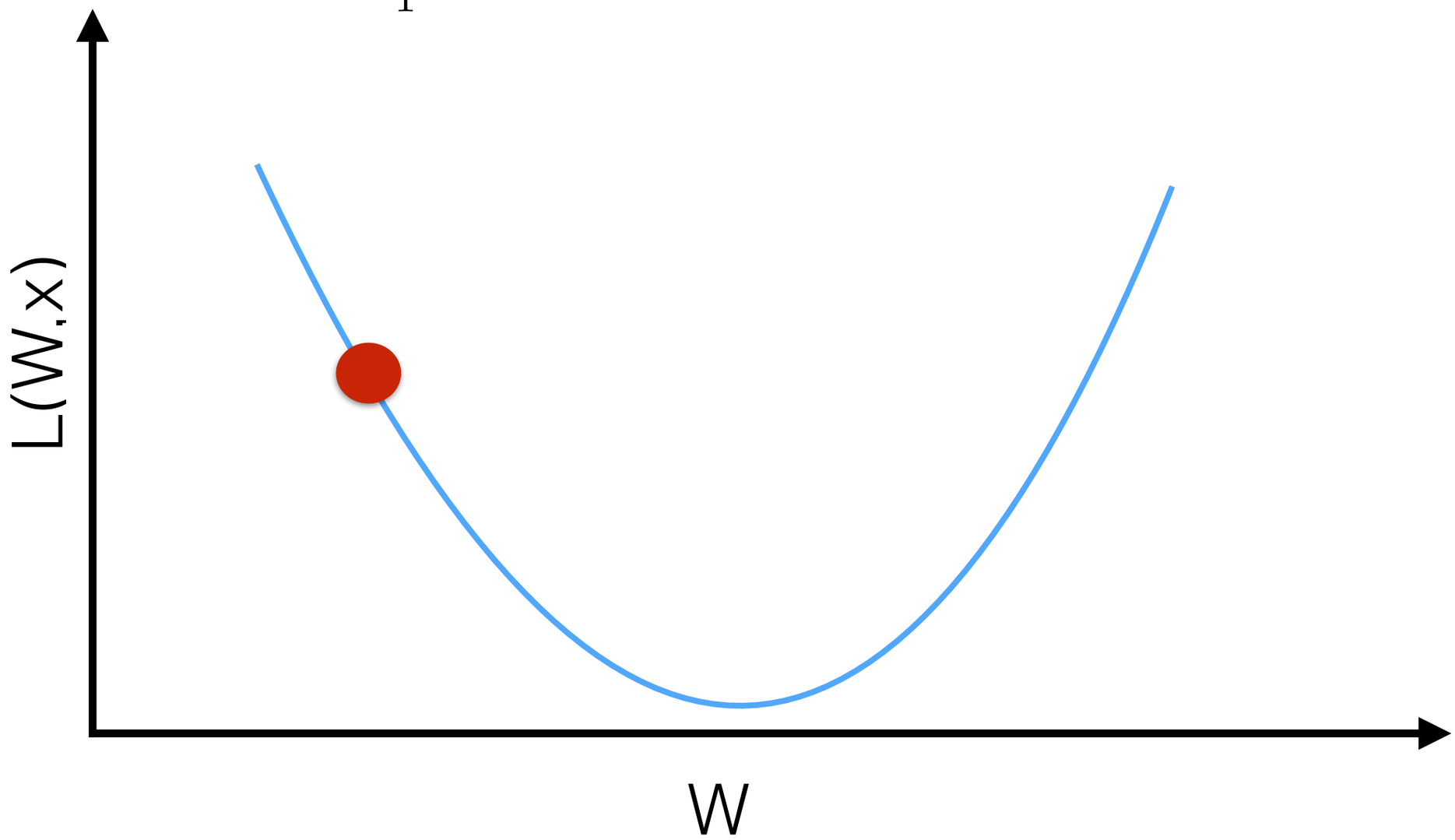
$$\sigma =$$



Training A Neural Network

Start with a “Loss” function which characterizes the performance of the network. For supervised learning:

$$L(W,X) = \frac{1}{N} \sum_{1}^{N_{examples}} -y_i \log\left(f(x_i)\right) - (1-y_i)\log\left(1-f(x_i)\right)$$



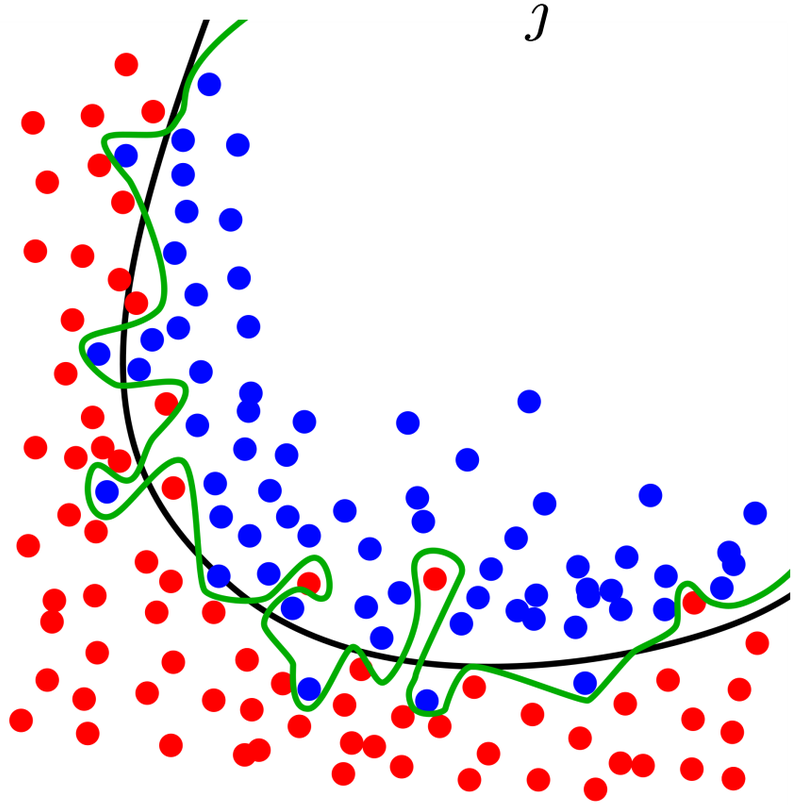
Training A Neural Network

Start with a “Loss” function which characterizes the performance of the network. For supervised learning:

$$L(W,X) = \frac{1}{N} \sum_{1}^{N_{examples}} -y_i \log\left(f(x_i)\right) - (1-y_i)\log\left(1-f(x_i)\right)$$

Add in a regularization term to avoid overfitting:

$$L' = L + \frac{1}{2}\sum_j w_j^2$$



Training A Neural Network

Start with a “Loss” function which characterizes the performance of the network. For supervised learning:

$$L(W,X) = \frac{1}{N} \sum_{1}^{N_{examples}} -y_i \log\left(f(x_i)\right) - (1-y_i)\log\left(1-f(x_i)\right)$$

Add in a regularization term to avoid overfitting:

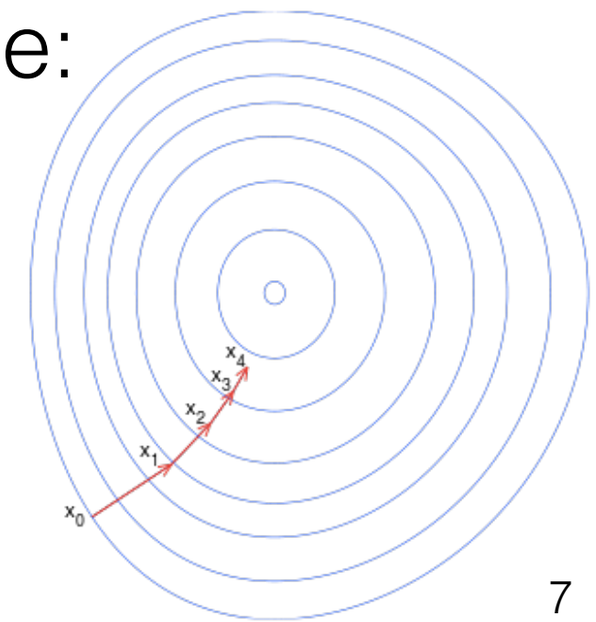
$$L' = L + \frac{1}{2}\sum_j w_j^2$$

Propagate the gradient of the network back to specific nodes using back propagation. AKA apply the chain rule:

$$\nabla_{w_j} L = \frac{\delta L}{\delta f}\frac{\delta f}{\delta g_n}\frac{\delta g_n}{\delta g_{n-1}}\cdots\frac{\delta g_{k+1}}{\delta g_k}\frac{\delta g_k}{\delta w_j}$$

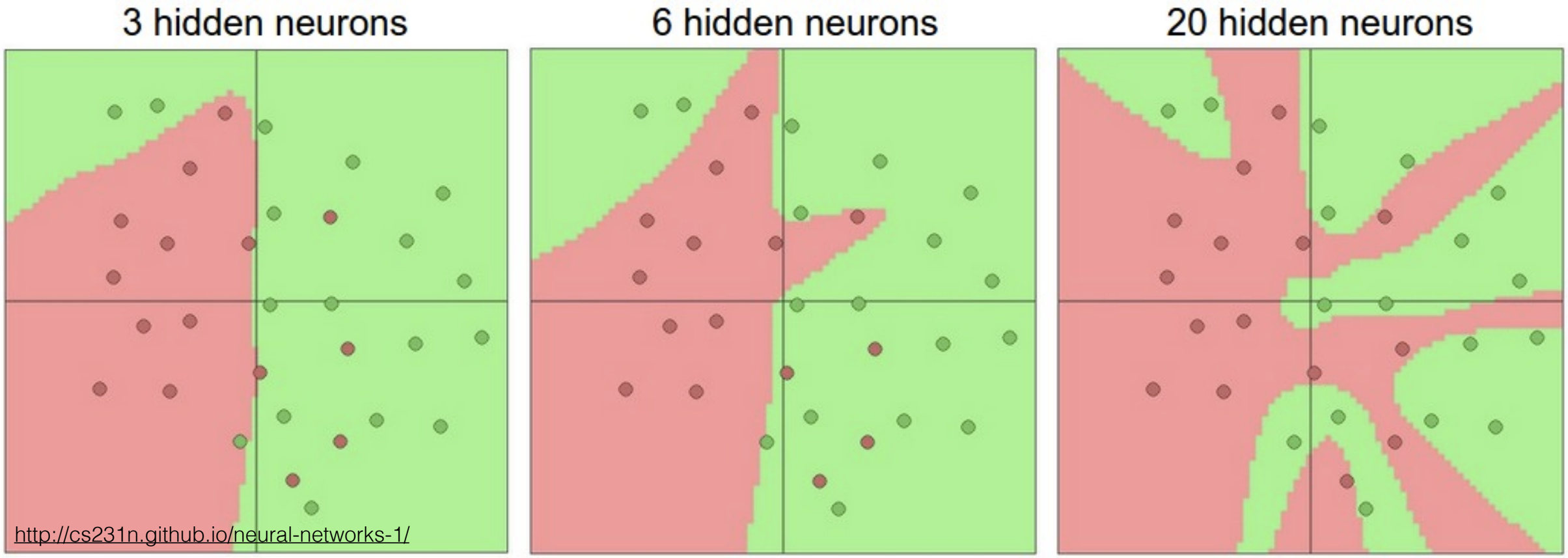
Update weights using gradient descent:

$$w_j^{'} = w_j - \alpha\nabla_{w_j} L$$



Deep Neural Networks

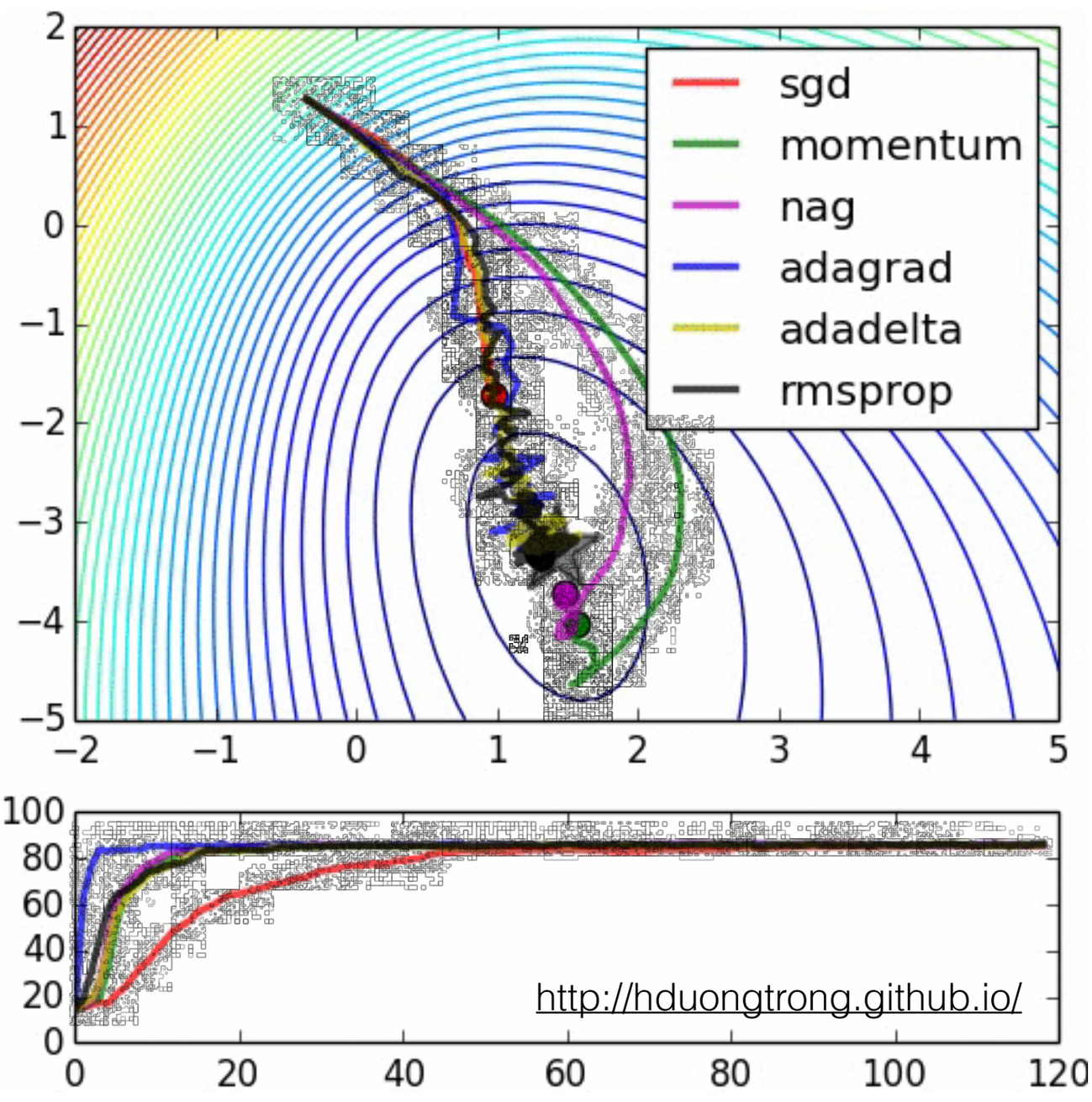
What if we try to keep all the input data? Why not rely on a wide, extremely Deep Neural Network (DNN) to learn the features it needs? Sufficiently deep networks make excellent function approximators:



However, until recently they proved almost impossible to train.

Smarter Training

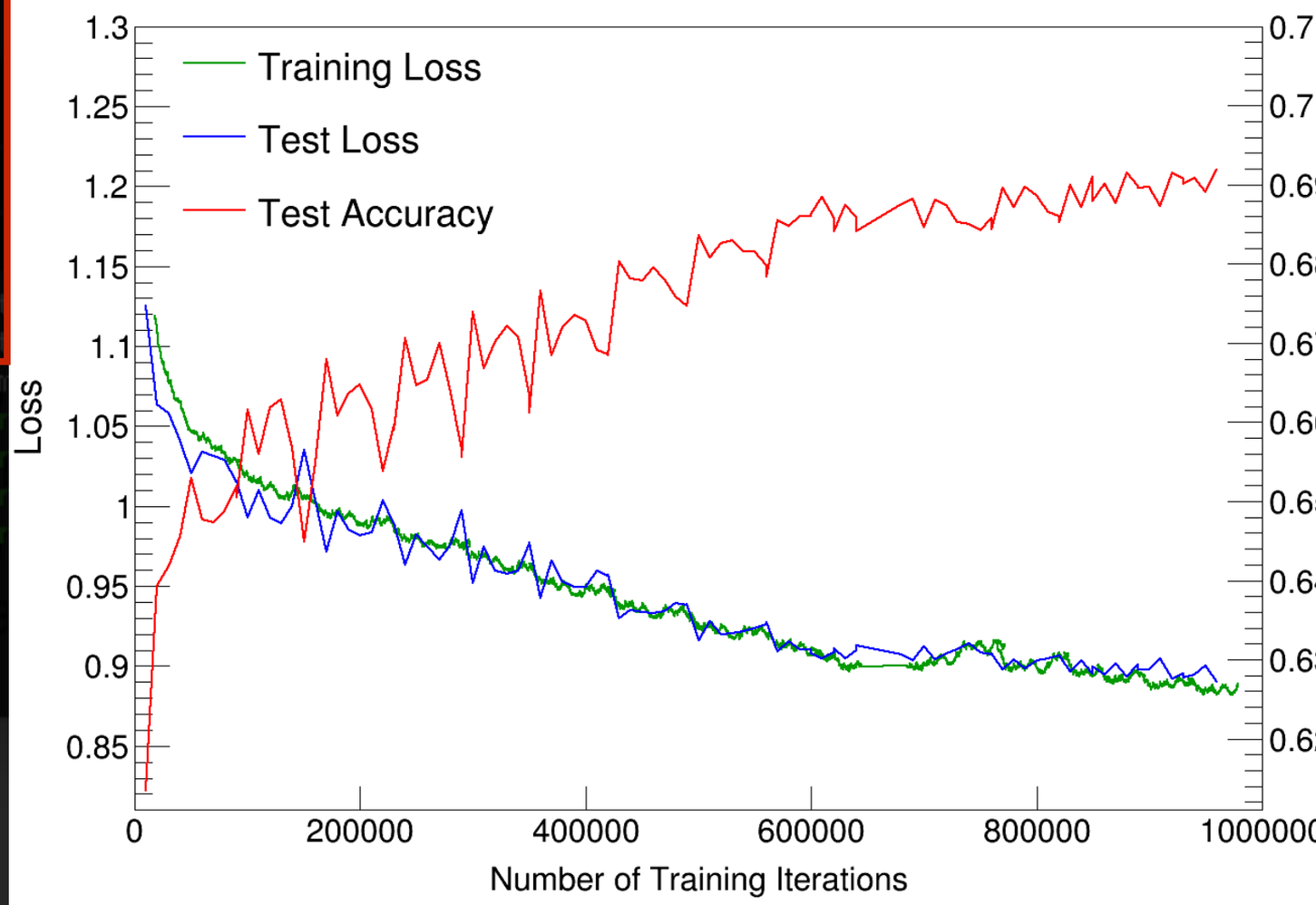
Another is stochastic gradient descent (SGD). In SGD we avoid some of the cost of gradient descent by evaluating as few as one event at a time. The performance of conventional gradient descent is approximated as the various noisy sub estimates even out, with the stochastic behavior even allowing for jumping out of local minima.



“Solver Prototxt”

Here you will define the basics of how you want the training to run. For example how often to run tests on the network, or how many events to evaluate in a given test phase.

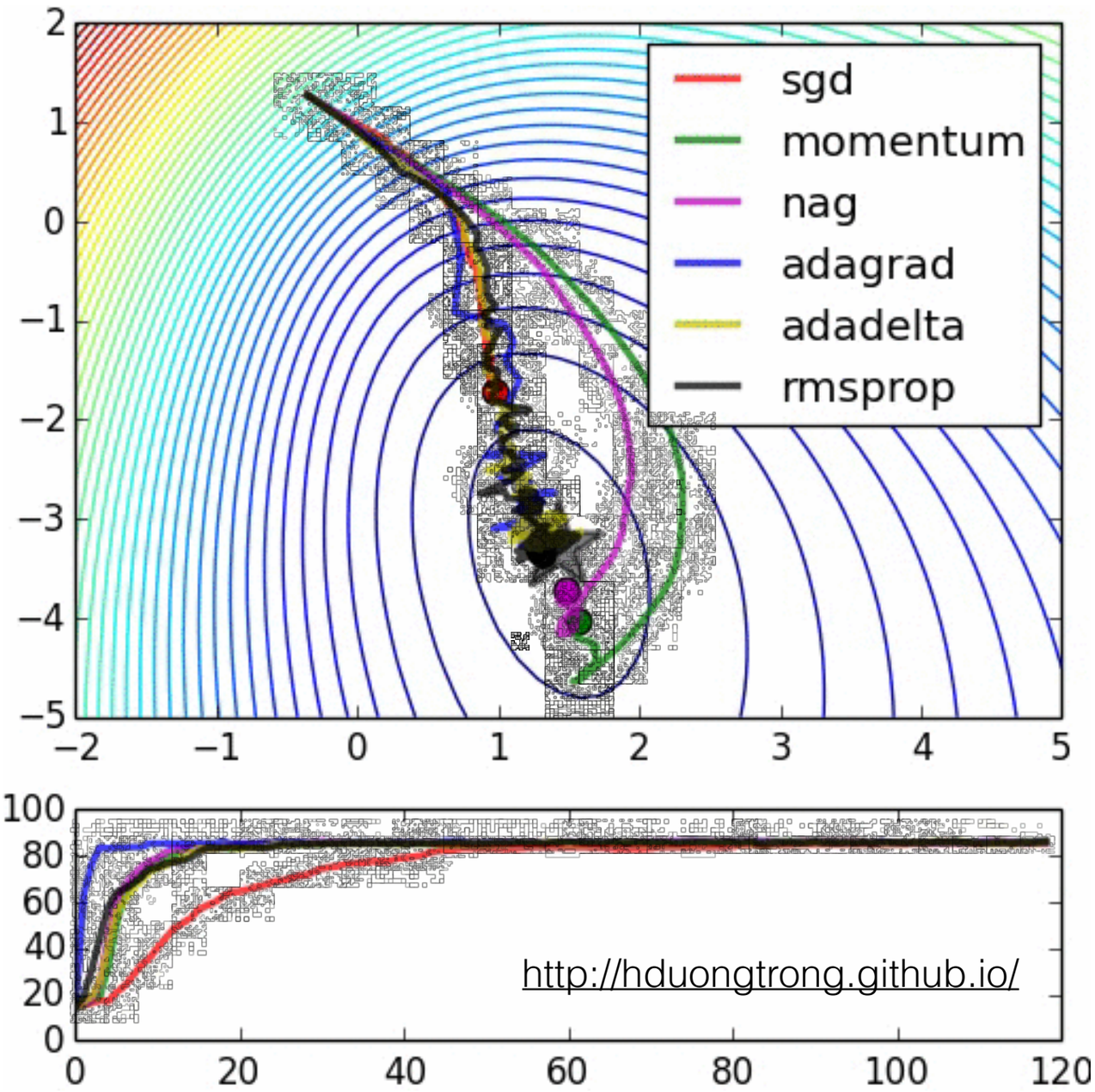
```
net: "lenet_nova.txt"
test_initialization: false
test_iter: 15000
test_interval: 50000
max_iter: 10000000
display: 40
average_loss: 40
snapshot_prefix: "/data/radovic/tutorial/lenet_nova"
snapshot: 25000
# solver mode: CPU or GPU
solver_mode: GPU
base_lr: 0.001
lr_policy: "step"
gamma: 0.1
momentum: 0.9
weight_decay: 0.0002
stepsize: 500000
# lr_policy: "inv"
# gamma: 0.0001
# power: 0.75
# momentum: 0.9
# weight_decay: 0.0005

# momentum: 0.9
# momentum2: 0.999
# lr_policy: "fixed"
# type: "Adam"
```



“Solver Prototxt”

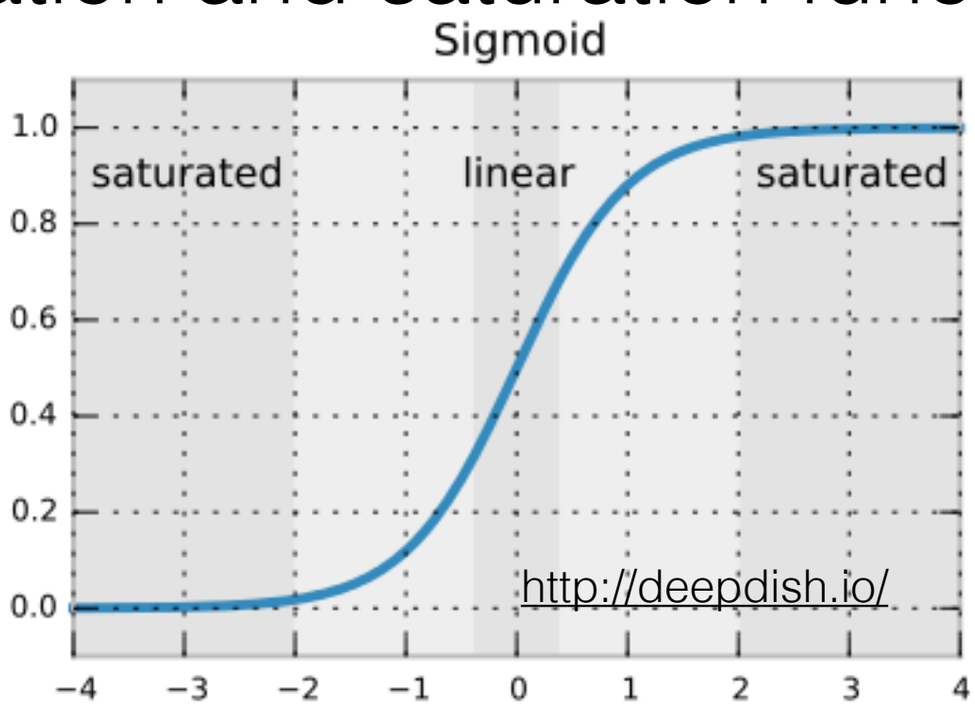
You’ll also set hyper parameters here, choosing your favorite variation on SGD and related terms like learning rate or momentum.

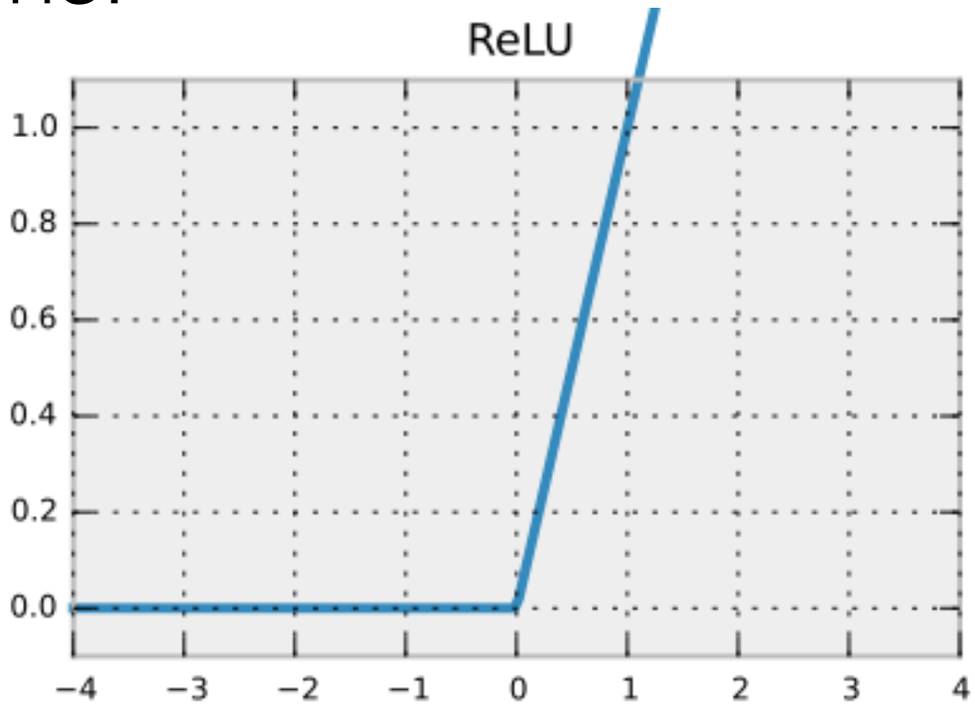
```
net: "lenet_nova.txt"
test_initialization: false
test_iter: 15000
test_interval: 50000
max_iter: 10000000
display: 40
average_loss: 40
snapshot_prefix: "/data/radovic/tutorial/lenet_nova"
snapshot: 25000
# solver mode: CPU or GPU
solver_mode: GPU
base_lr: 0.001
lr_policy: "step"
gamma: 0.1
momentum: 0.9
weight_decay: 0.0002
stepsize: 500000
# lr_policy: "inv"
# gamma: 0.0001
# power: 0.75
# momentum: 0.9
# weight_decay: 0.0005

# momentum: 0.9
# momentum2: 0.999
# lr_policy: "fixed"
# type: "Adam"
```



Better Activation Functions

But there were also some major technical breakthroughs. One being more effective back propagation due to better weight initialization and saturation functions:





The problem with sigmoids:

$$\frac{\delta\sigma(x)}{\delta x} = \sigma(x)(1-\sigma(x))$$

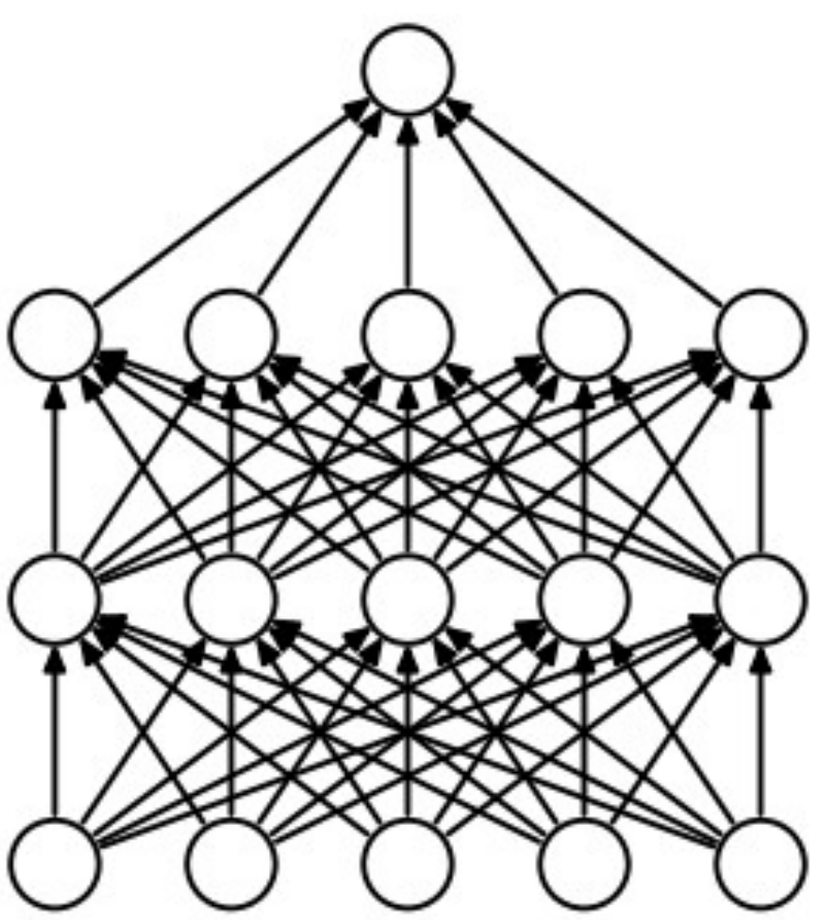
ReLU:

$$\frac{ReLU(x)}{\delta x} = \begin{cases} 1 & \text{when } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

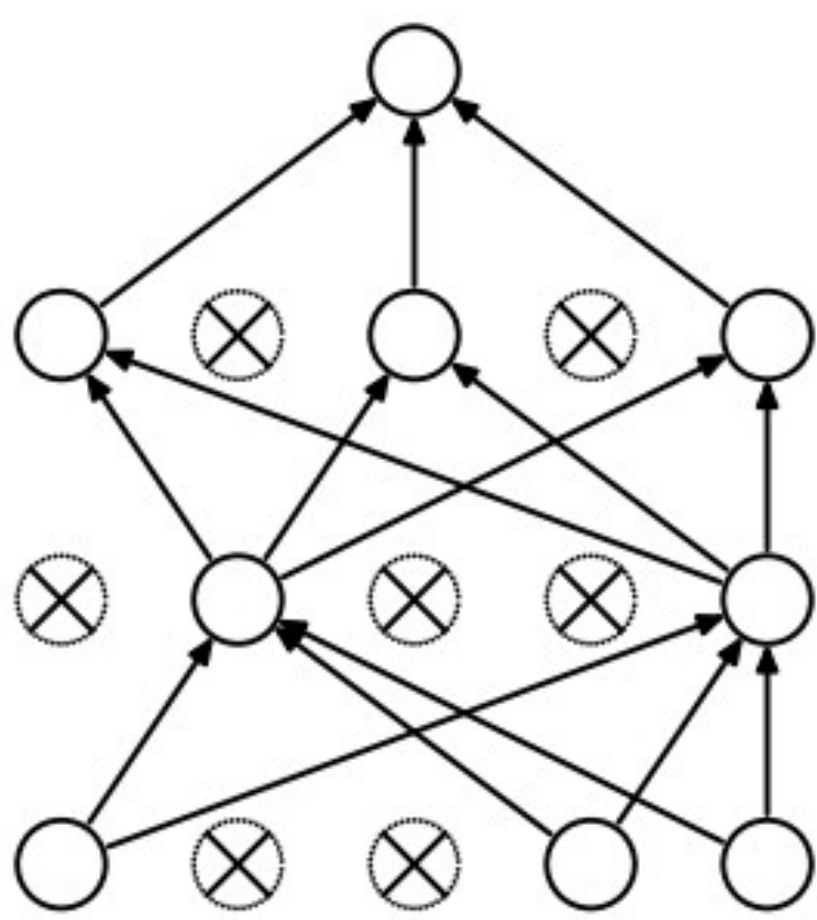
Sigmoid gradient goes to 0 when x is far from 1. Makes back propagation impossible! Use ReLU to avoid saturation.

Dropout

- Same goal as conventional regularization- prevent overtraining.
- Works by randomly removing whole nodes during training iterations. At each iteration, randomly set XX% of weights to zero and scale the rest up by 1/(1 – 0.XX).
- Forces the network not to build complex interdependencies in the extracted features.



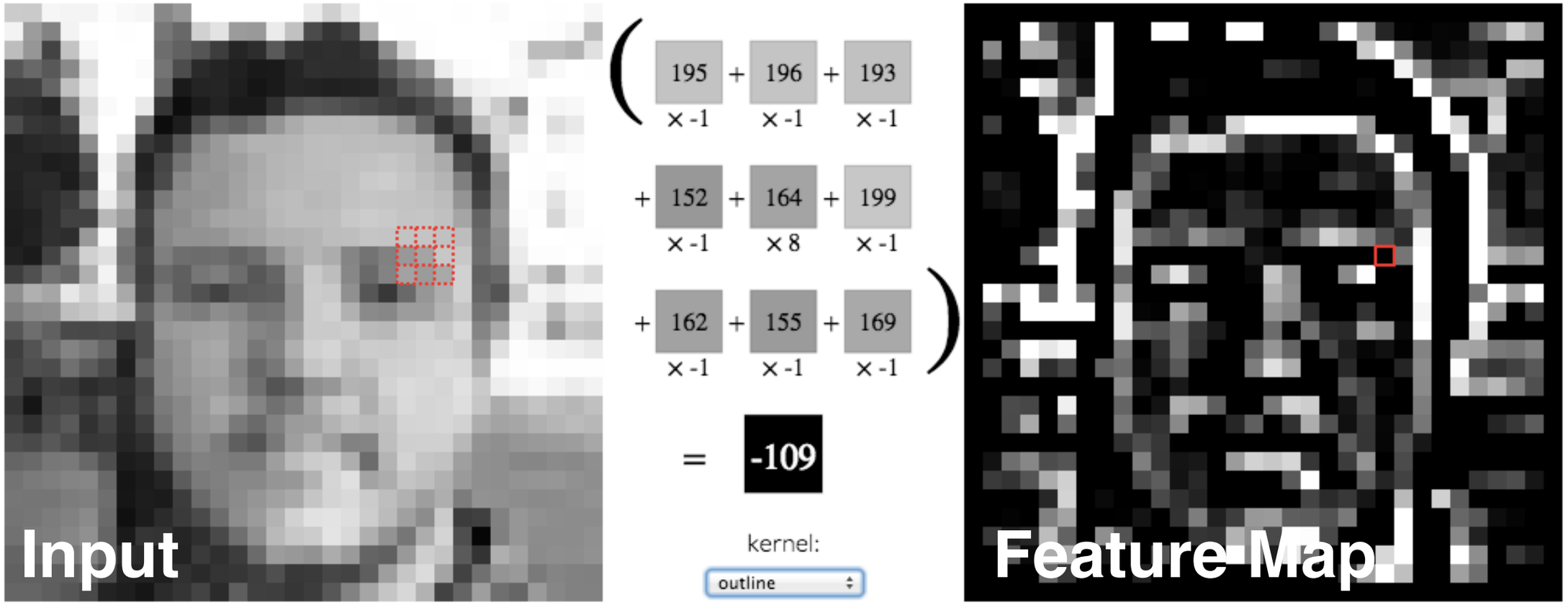
(a) Standard Neural Net



(b) After applying dropout.

Convolutional Neural Networks

Instead of training a weight for every input pixel, try learning weights that describe kernel operations, convolving that kernel across the entire image to exaggerate useful features.
Inspired by research showing that cells in the visual cortex are only responsive to small portions of the visual field.

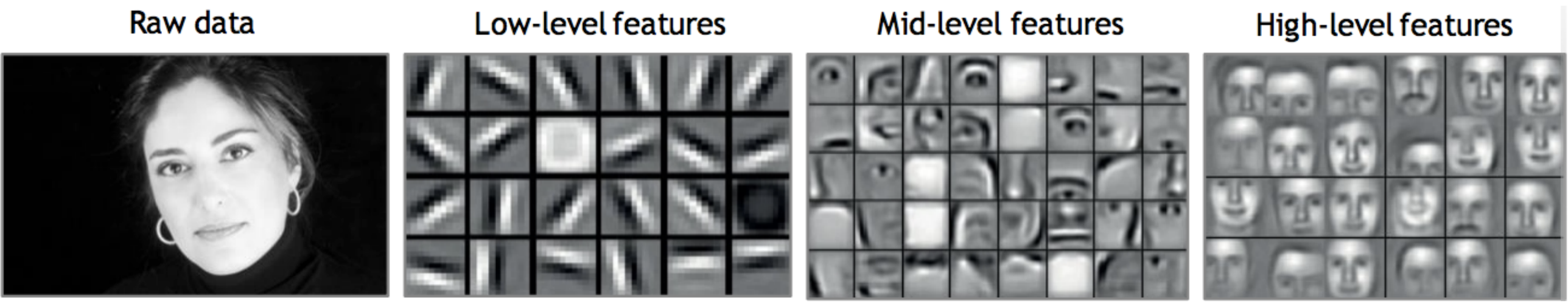


Kernel

http://setosa.io/ev/image-kernels/

Convolutional Neural Networks

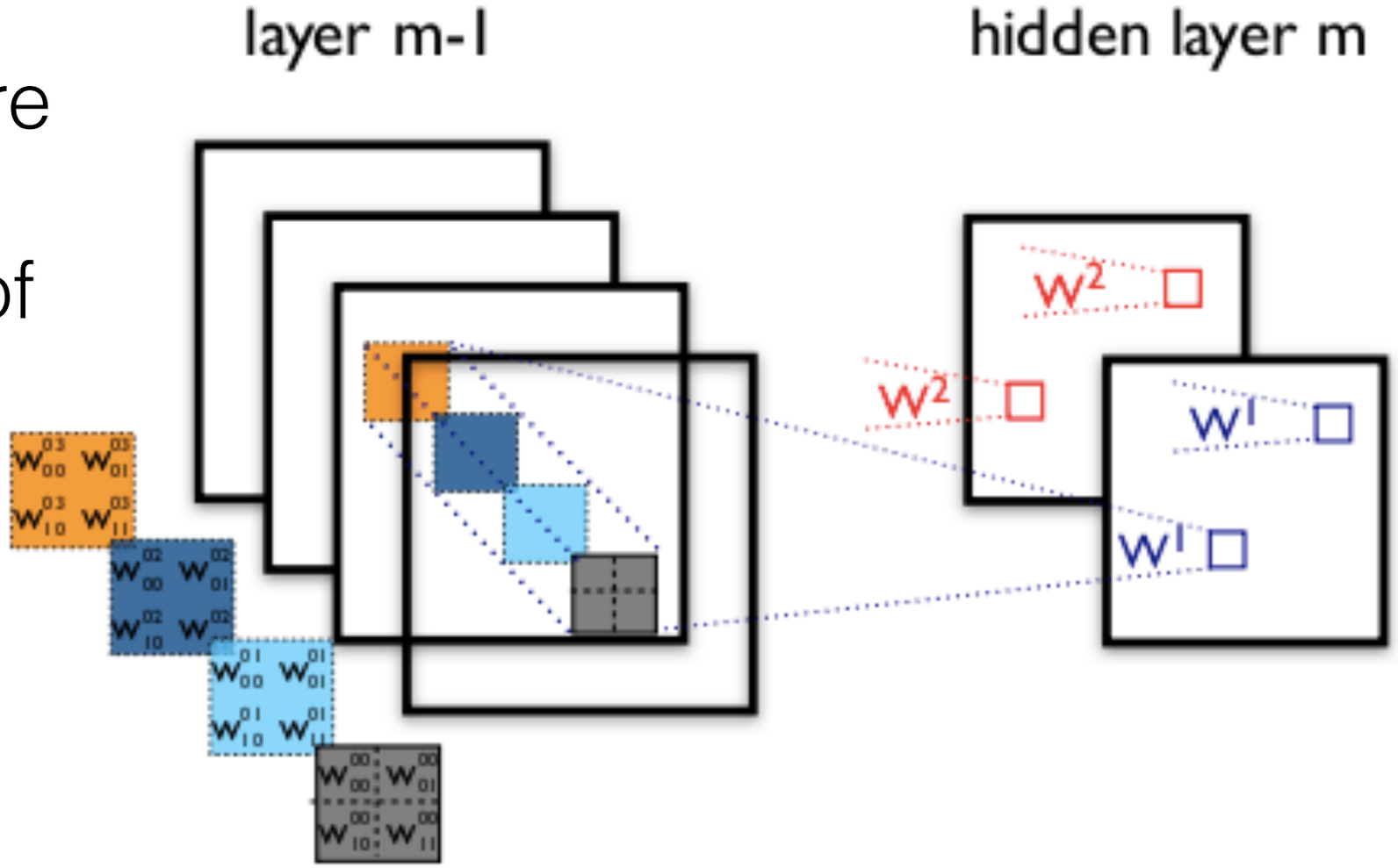
Instead of training a weight for every input pixel, try learning weights that describe kernel operations, convolving that kernel across the entire image to exaggerate useful features.
Inspired by research showing that cells in the visual cortex are only responsive to small portions of the visual field.



https://developer.nvidia.com/deep-learning-courses

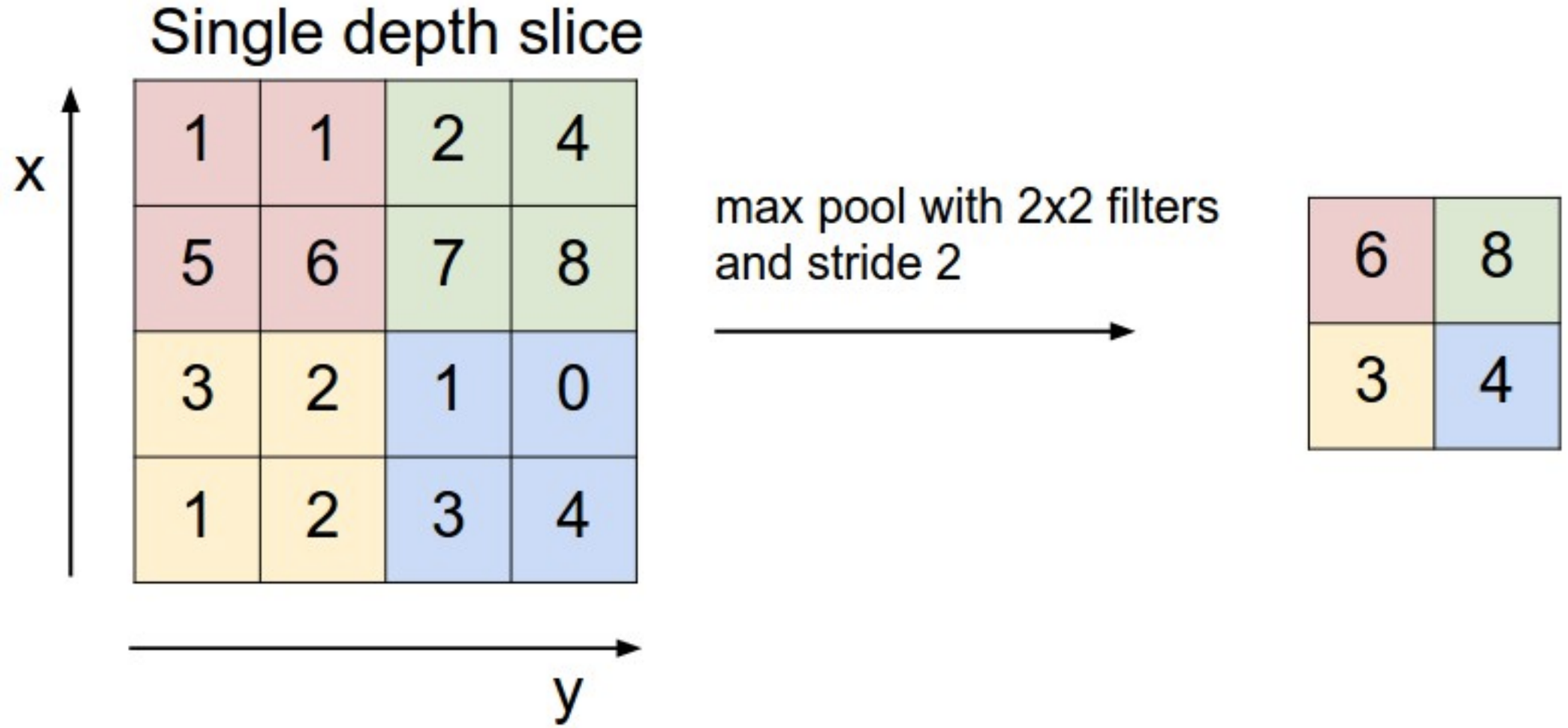
Convolutional Layers

- Every trained kernel operation is the same across an entire input image or feature map.
- Each convolutional layer trains an array of kernels to produce output feature maps.
- Weights for a given convolutional layer are a 4D tensor of NxMxHxW (number of incoming features, number of outgoing features, height, and width)

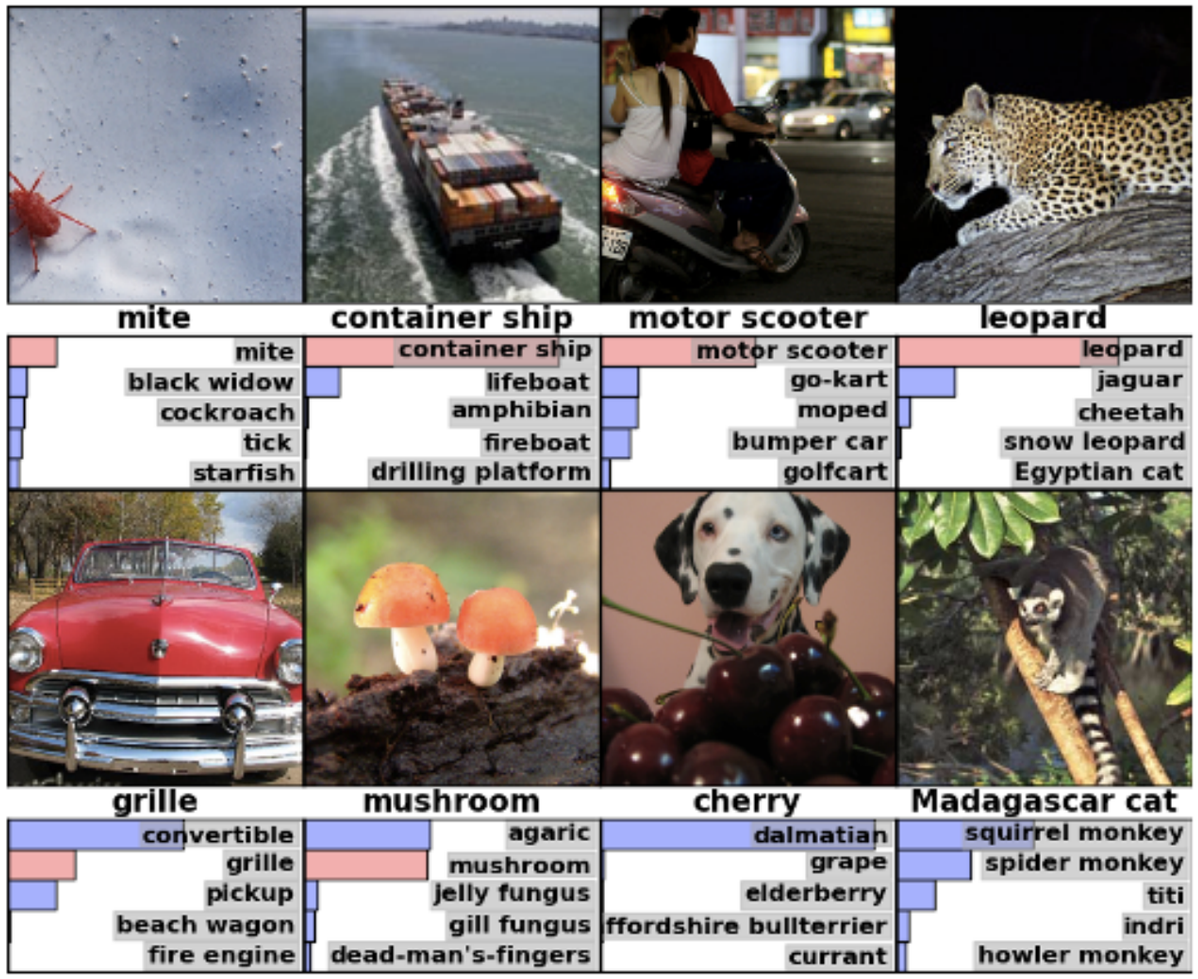


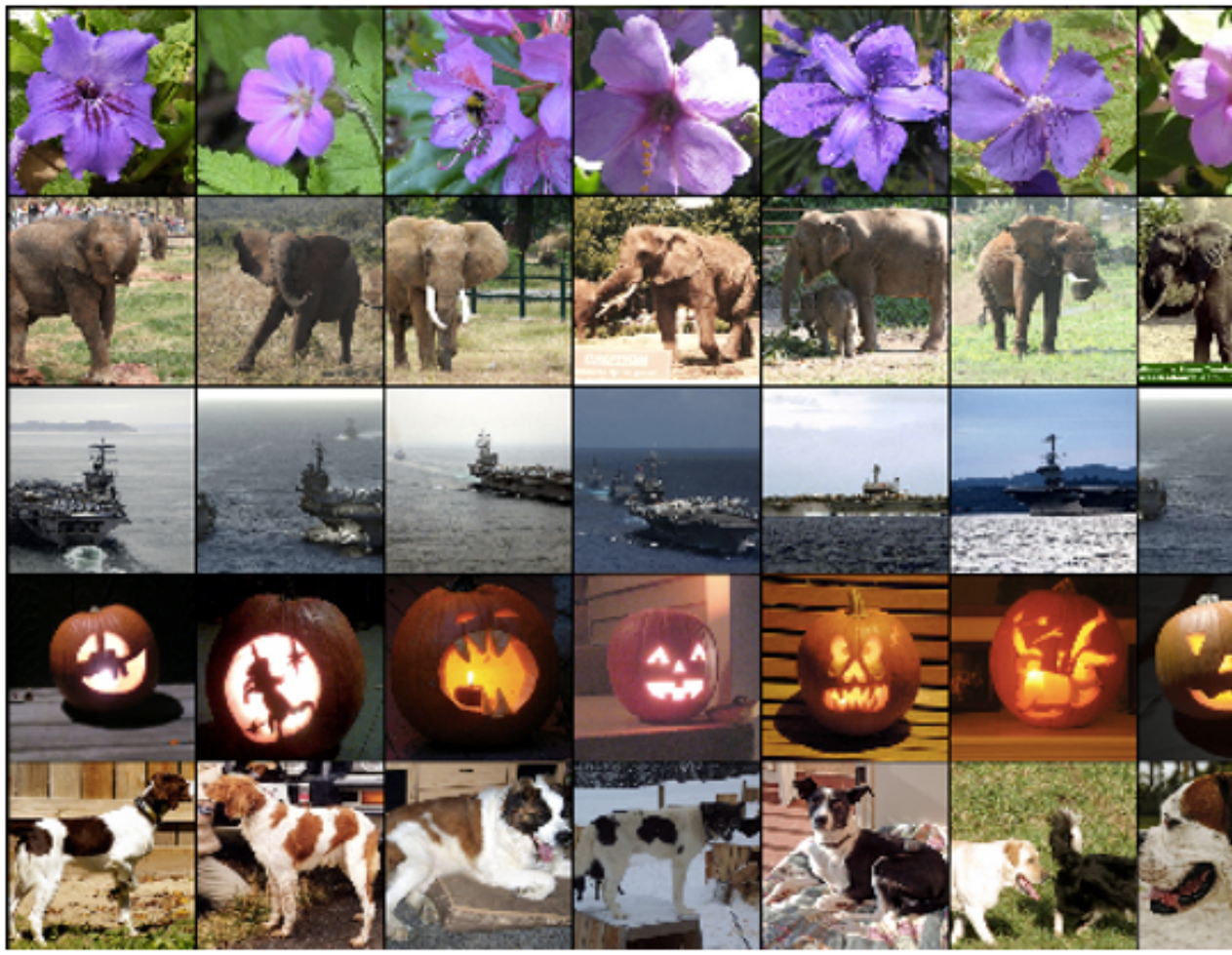
Pooling Layers

- Intelligent downscaling of input feature maps.
- Stride across images taking either the maximum or average value in a patch.
- Same number of feature maps, with each individual feature map shrunk by an amount dependent on the stride of the pooling layers.



Superhuman Performance



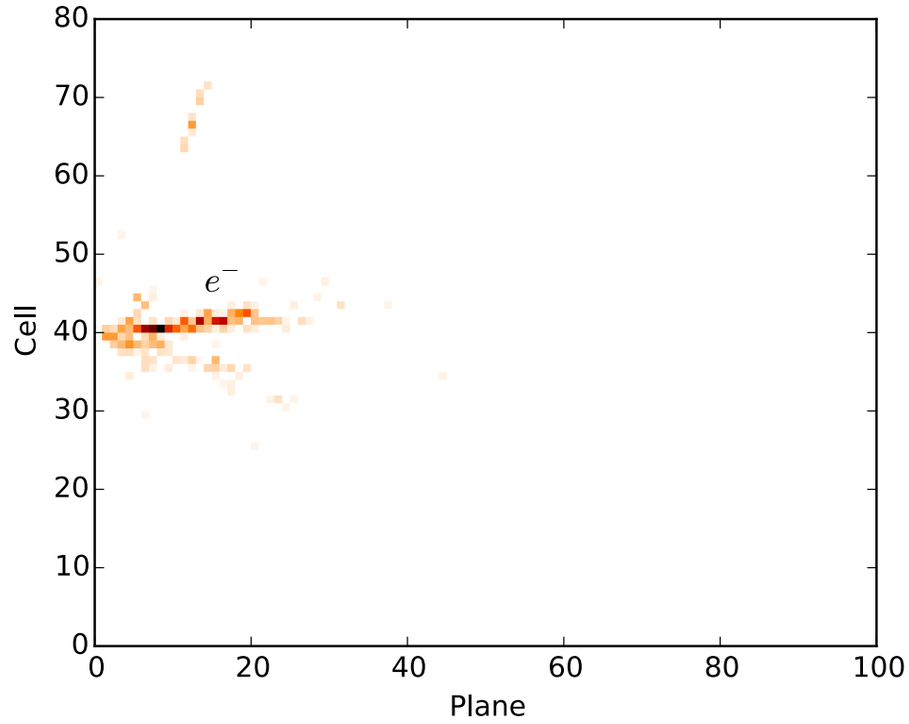


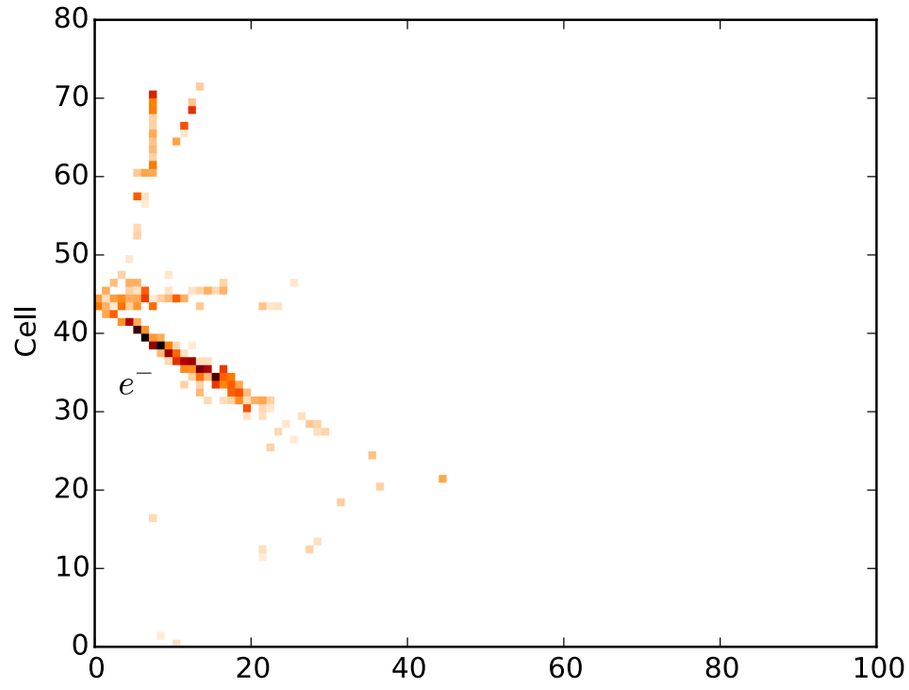
Some examples from one of the early breakout CNNs. Googles latest “Inception-v4” net achieves 3.46% top 5 error rate on the image net dataset. Human performance is at ~5%.

“Train/Test Prototxt”

This is where you’ll define your architecture, and your input datasets.

```
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    #scale: 0.00390625
  }
  data_param {
    source: "/data/atsaris/beamall_cosmic_devJun22_datasets/output/TrainLevelDB"
    batch_size: 16
    prefetch: 10
    backend: LEVELDB
  }
}
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  #transform_param {
  #mirror: true
  #scale: 0.00390625
  #}
  data_param {
    source: "/data/atsaris/beamall_cosmic_devJun22_datasets/output/TestLevelDB"
    batch_size: 64
    prefetch: 10
    backend: LEVELDB
  }
}
```

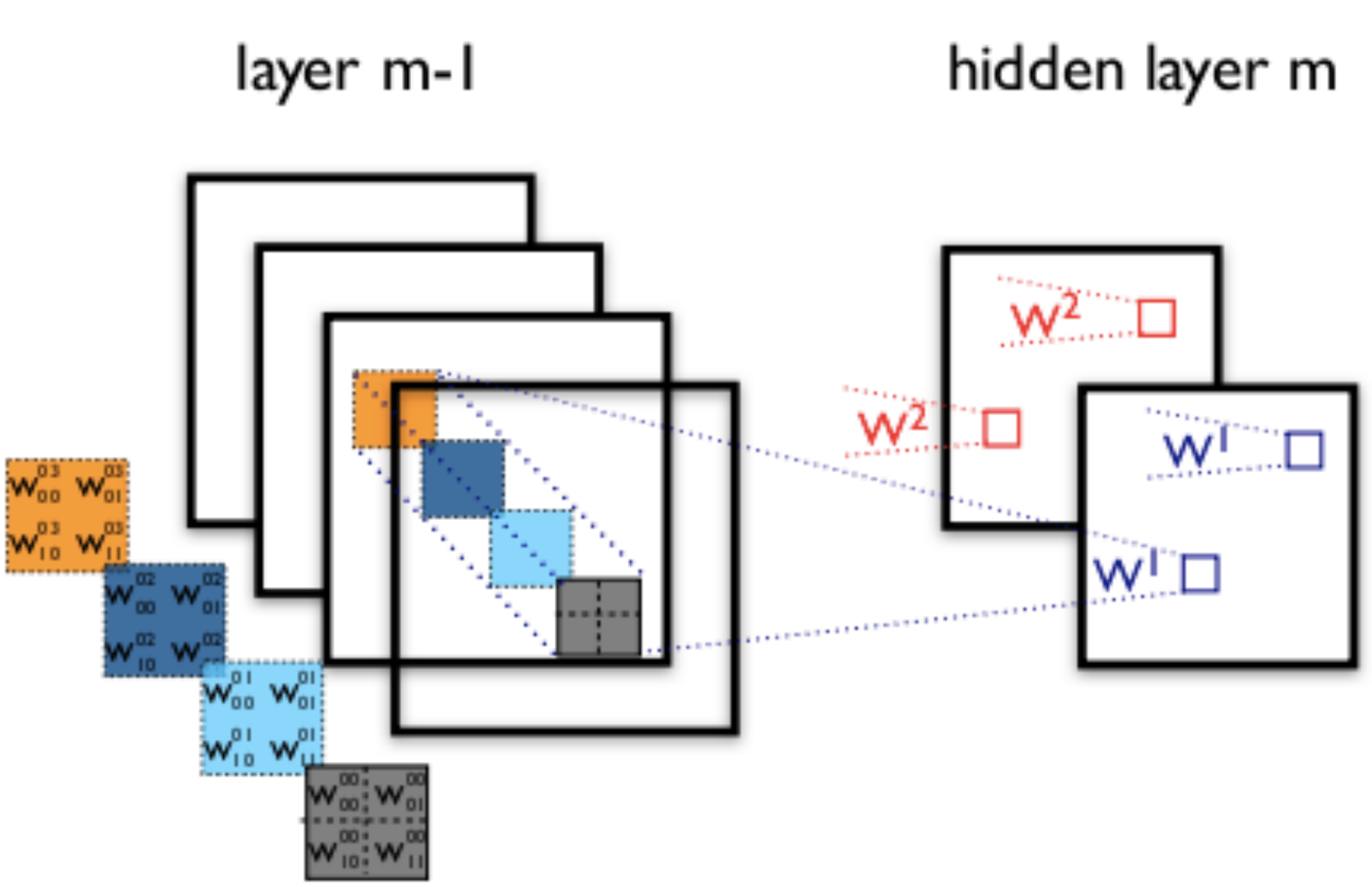




“Train/Test Prototxt”

The architecture itself is in a series of layers. You’ll need to describe those layers, and make sure they fit into the wider ensemble correctly. Some layers like this one defining a set of convolutional operations take a previous layers as input and output a new one.

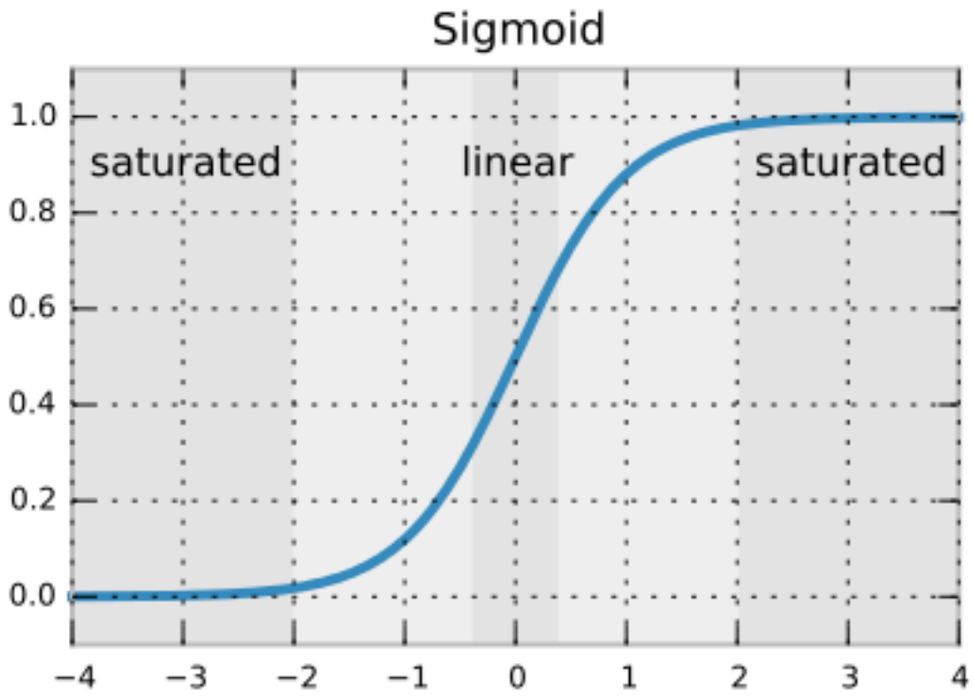
```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

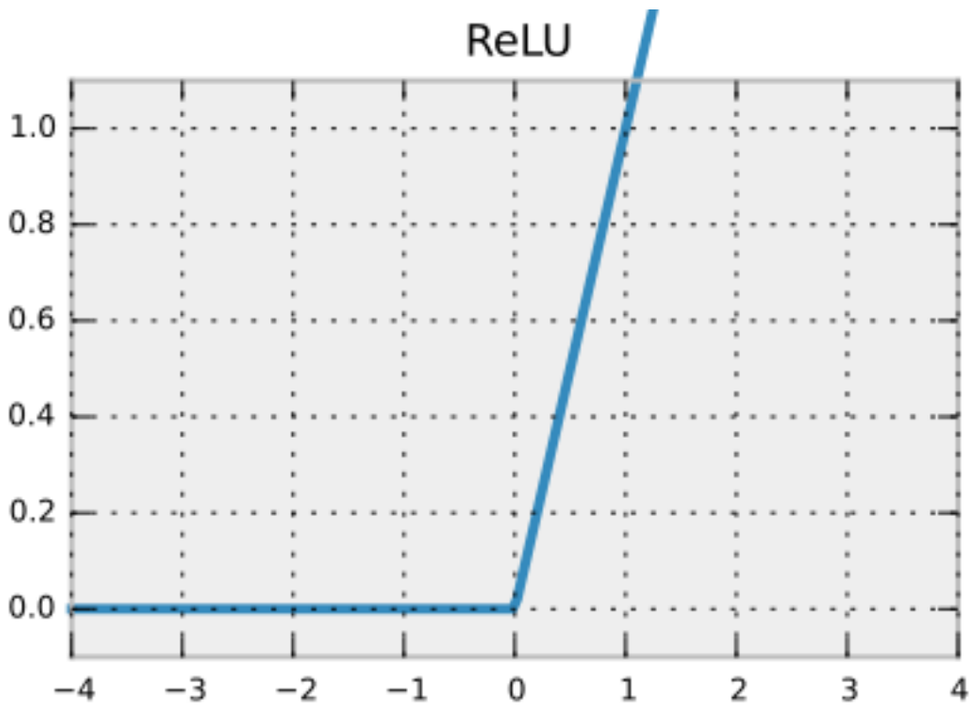


“Train/Test Prototxt”

Others modify a layer, defining for example which activation function to use.

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

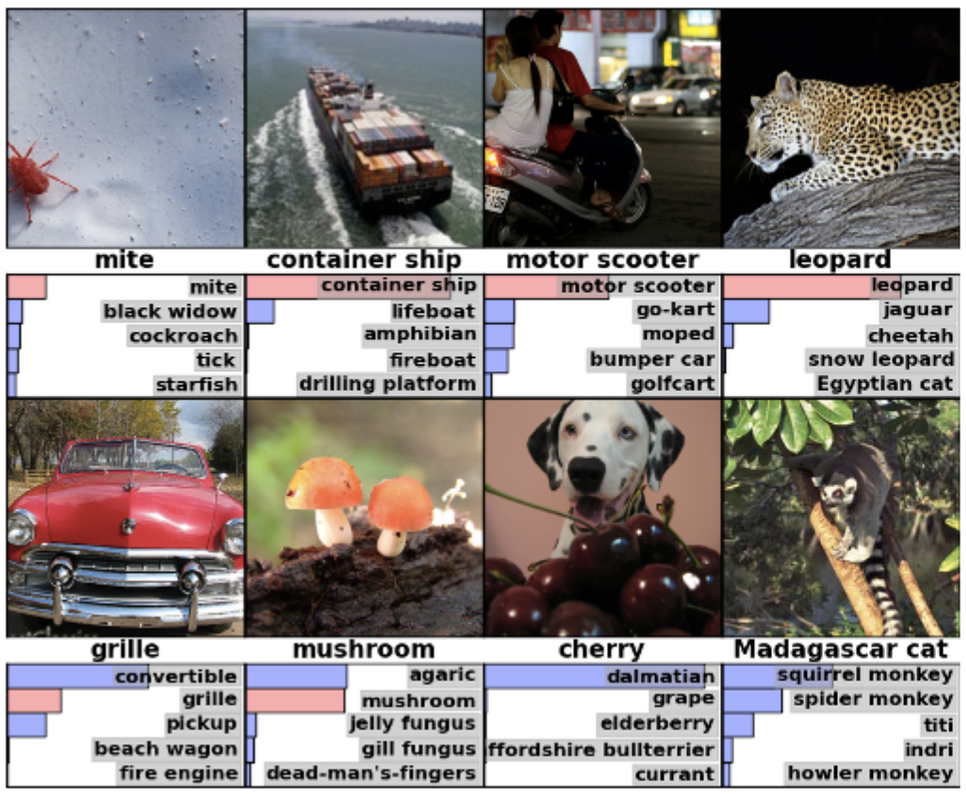


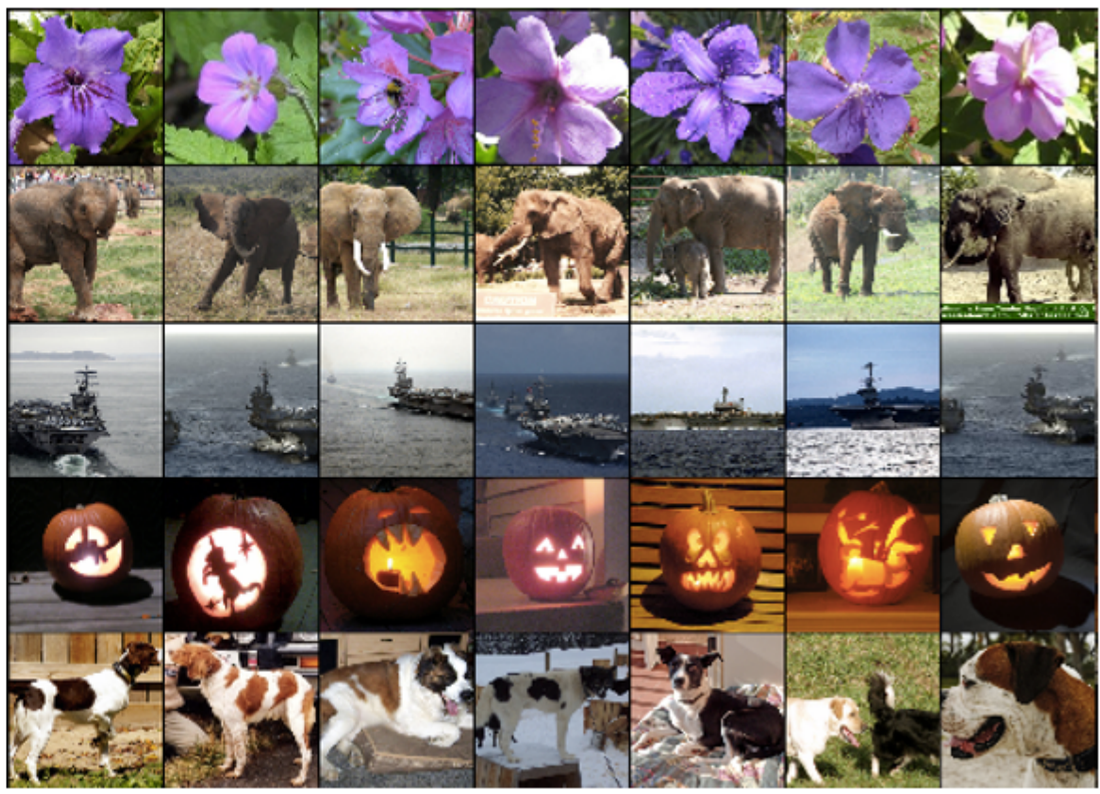


“Train/Test Prototxt”

```
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 392
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

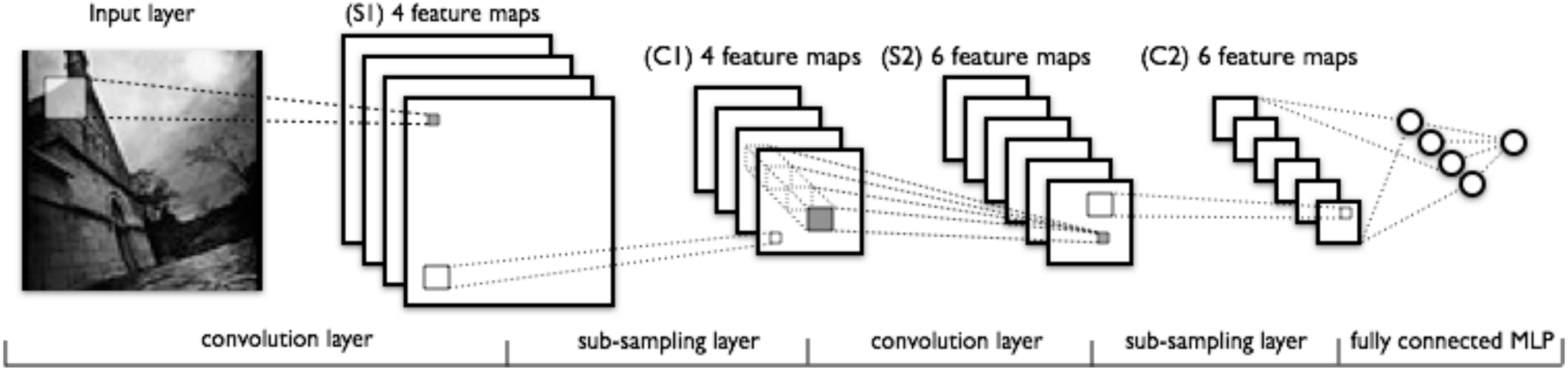
At the end of your network architecture you’ll need to pick a loss calculation and other metrics to output in test phases, like the top-1 or top-n accuracy.





The LeNet

Now let's take a look at the LeNet. A convolutional neural network in perhaps its simplest form, a series of convolutional, max pooling, and MLP layers:



The "LeNet" circa 1989

http://deeplearning.net/tutorial/lenet.html http://yann.lecun.com/exdb/lenet/

Some Toy Examples

In this directory (on the Wilson Cluster):

/home/radovic/exampleNetwork/forAris/tutorial/

You'll find an LeNet implementation designed for use on handwritten characters, an example network that comes with caffe (lenet_train_test.txt).

You'll also see an example of how that network has been edited to work with NOvA inputs (lenet_nova.txt), and some examples of how you might edit that (lenet_nova_extralayer.txt,lenet_solver_nova_branched.prototxt) to explore perturbations on that central design.

They come with solver files with commented out alternative solvers, please feel free to try them out! Also remember to try visualizing them using http://ethereon.github.io/netscope/#/editor.

http://deeplearning.net/tutorial/lenet.html http://yann.lecun.com/exdb/lenet/