Fall 2015 Deep Learning CMPSCI 697L

Deep Learning Lecture 2

Sridhar Mahadevan
Autonomous Learning Lab
UMass Amherst

UMASS CS COLLEGE OF COMPUTER SCIENCE 50 YEARS

Outline

Some topics to be covered:

1. Quick review of classic neural nets, single layer, multi layer.

2. Where does backprpagation run into difficulties?

3. Examples of new deep architectures: CNNs, max pooling units, etc.

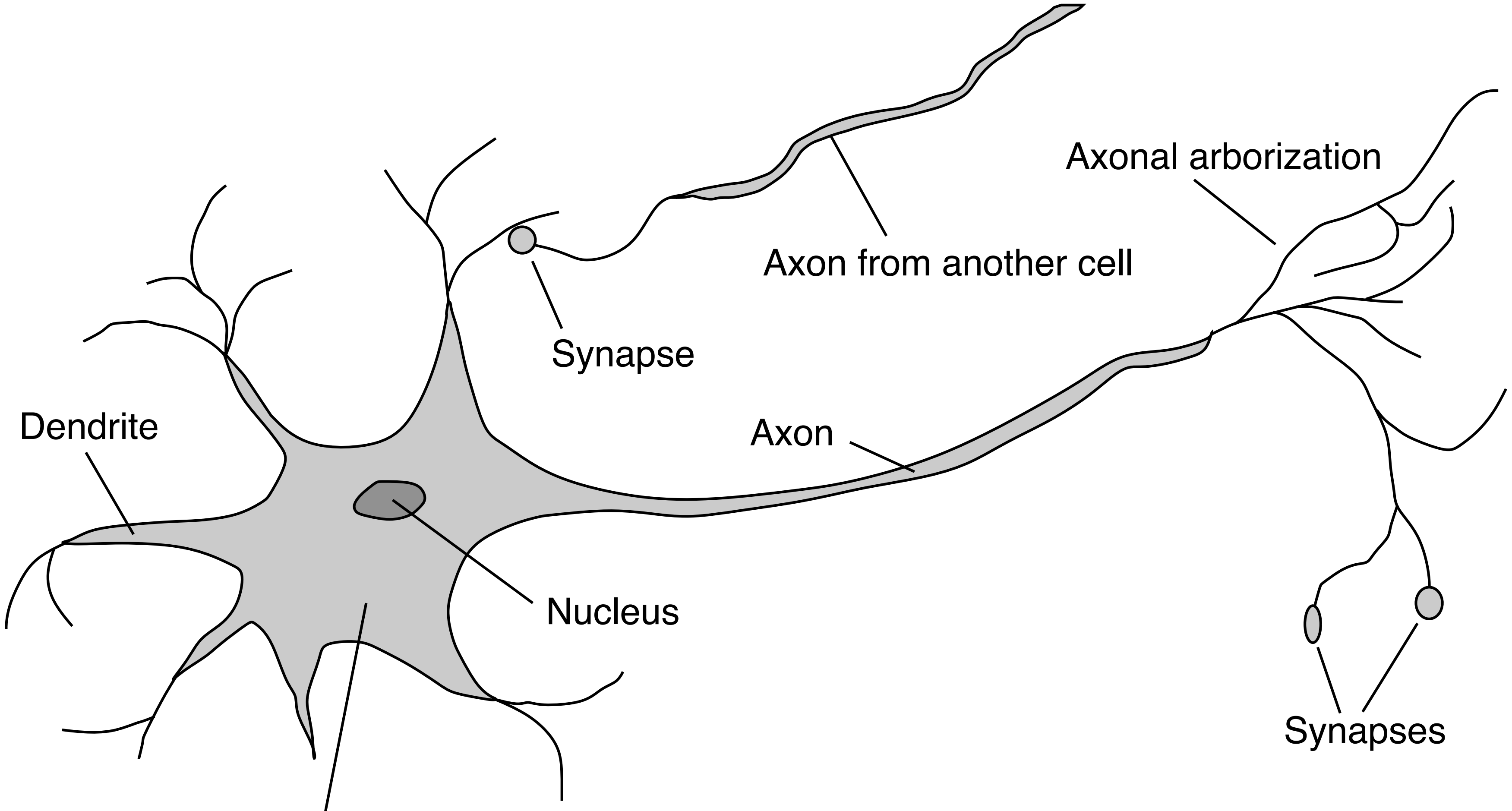
4. Software implementations in Theano, Caffe.

5. Forum discussions.

6. More details on common midterm group project.

Human Brain

10^{11} neurons of >20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential



What's new this time around?

- New ideas for preventing overfitting: dropout
- New types of units: RLUs, max pooling
- Lots more data and compute (GPU) power
- New stochastic gradient algorithms
- Renewed interest in convolutional neural networks

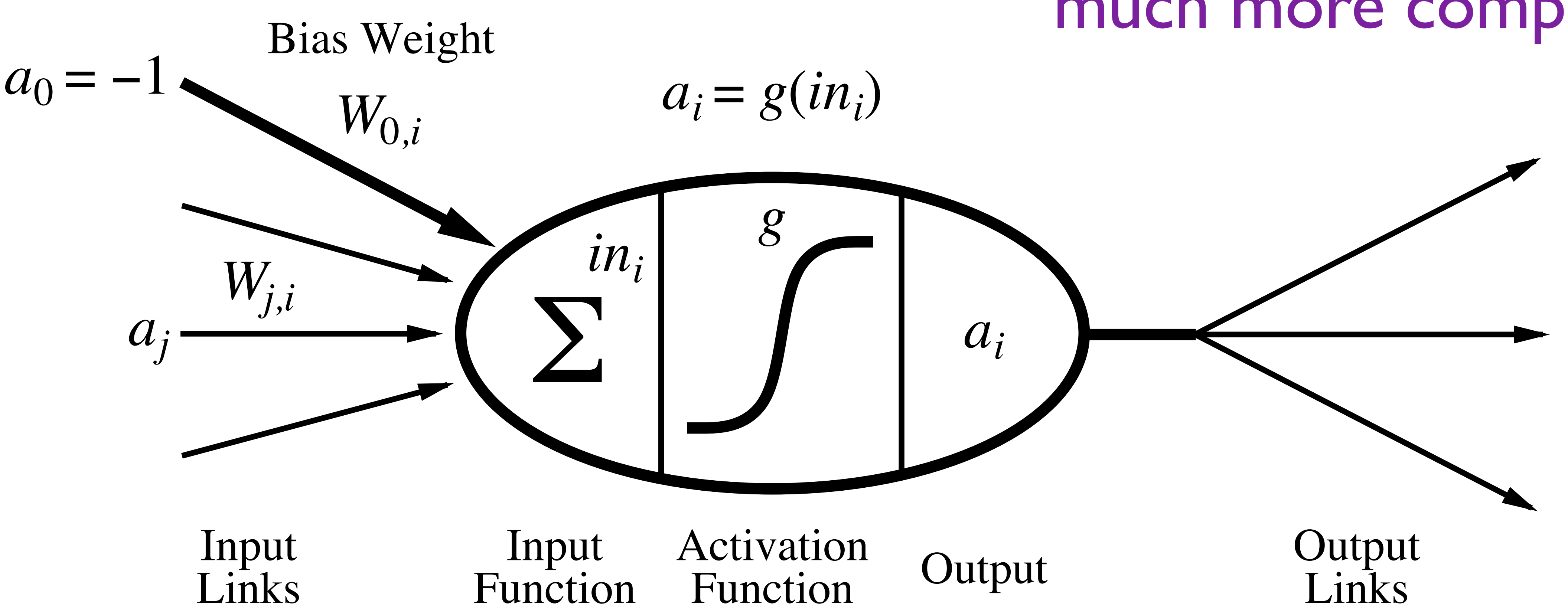
Quick Overview of Neural Networks

Simple Model of Neuron

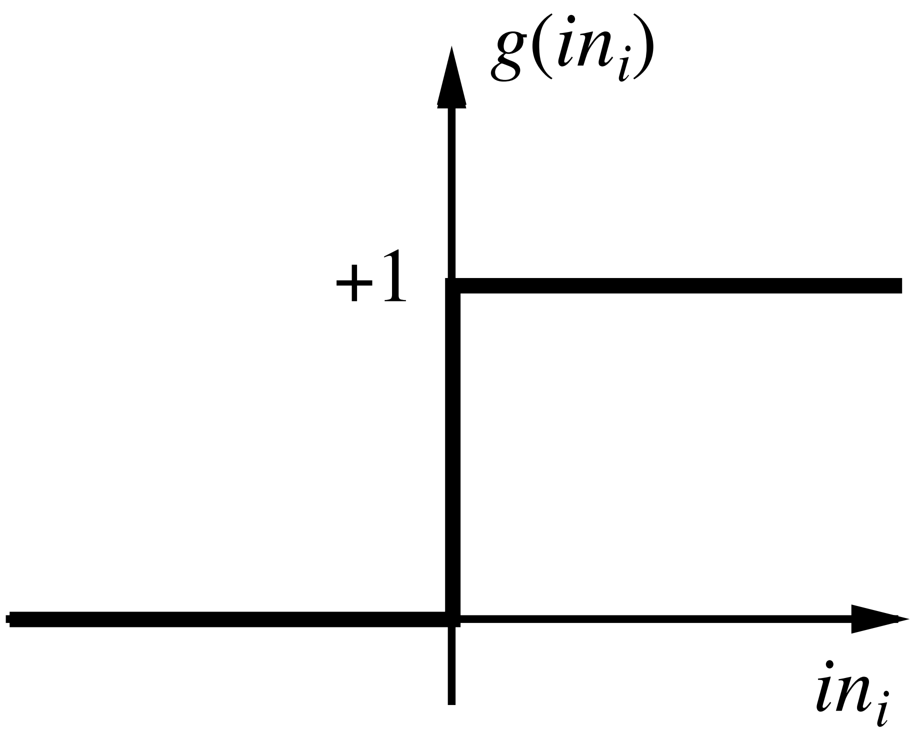
Output is a "squashed" linear function of the inputs:

$$a_i \leftarrow g(in_i) = g\left(\Sigma_j W_{j,i} a_j\right)$$

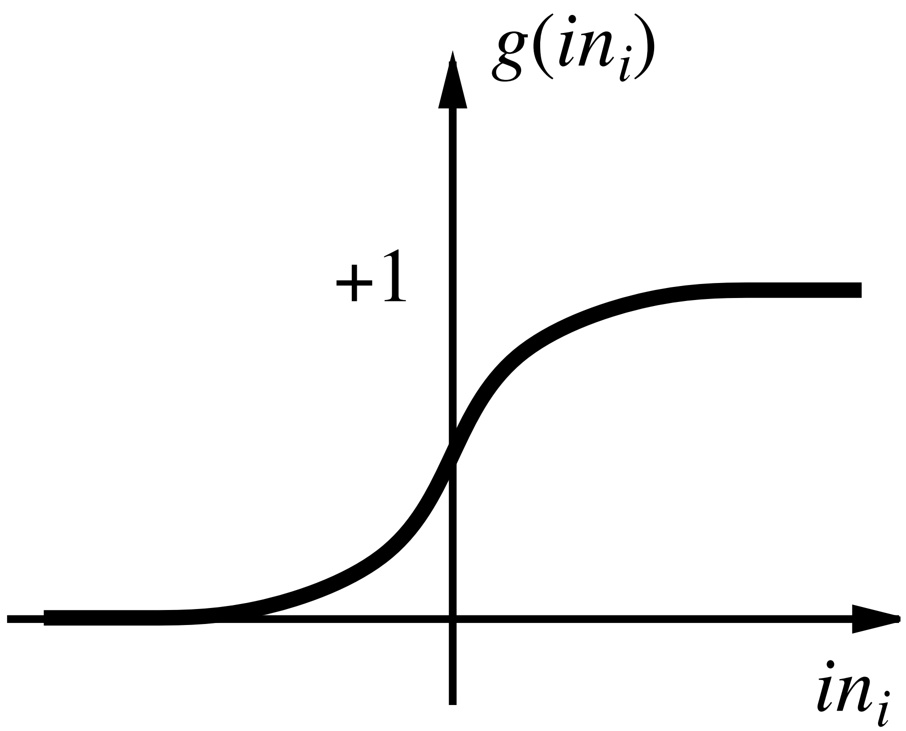
Real neurons are much more complex!



Activation Function



(a)



(b)

(a) is a step function or threshold function

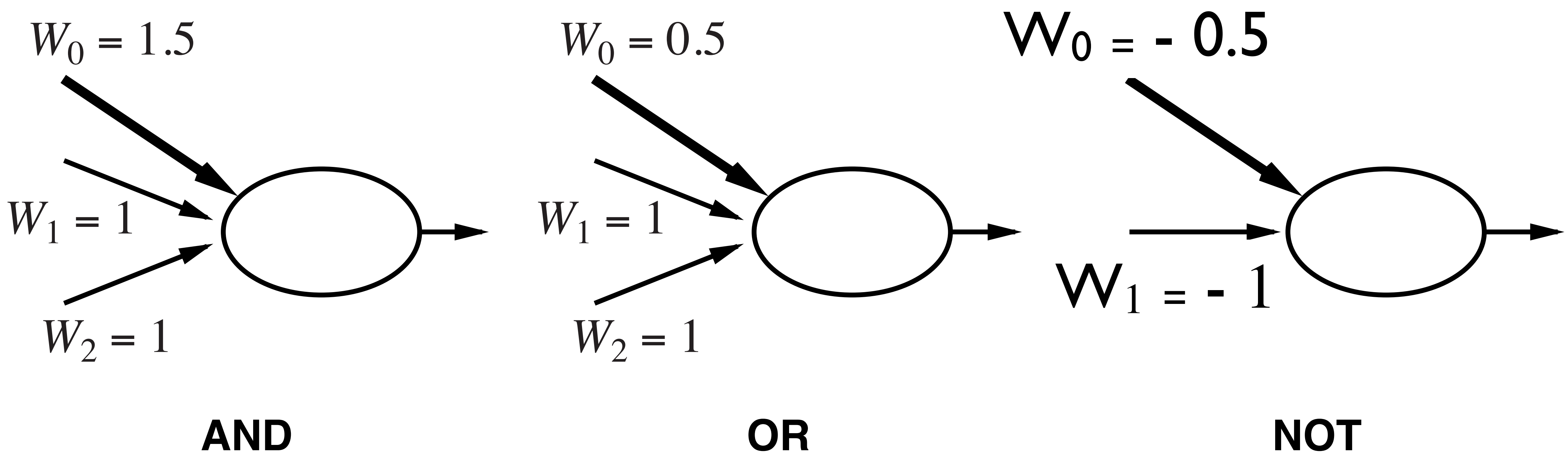
(b) is a sigmoid function $1/(1+e^{-x})$

Changing the bias weight $W_{0,i}$ moves the threshold location

Types of units

- Linear: compute weighted sum of inputs
- Perceptrons
- RLU: rectified linear units (negative -> 0)
- Sigmoid units: logistic regression function
- Hyperbolic tangent unit
- Convolutional neural nets filter units

Boolean Functions



McCulloch and Pitts: every Boolean function can be implemented

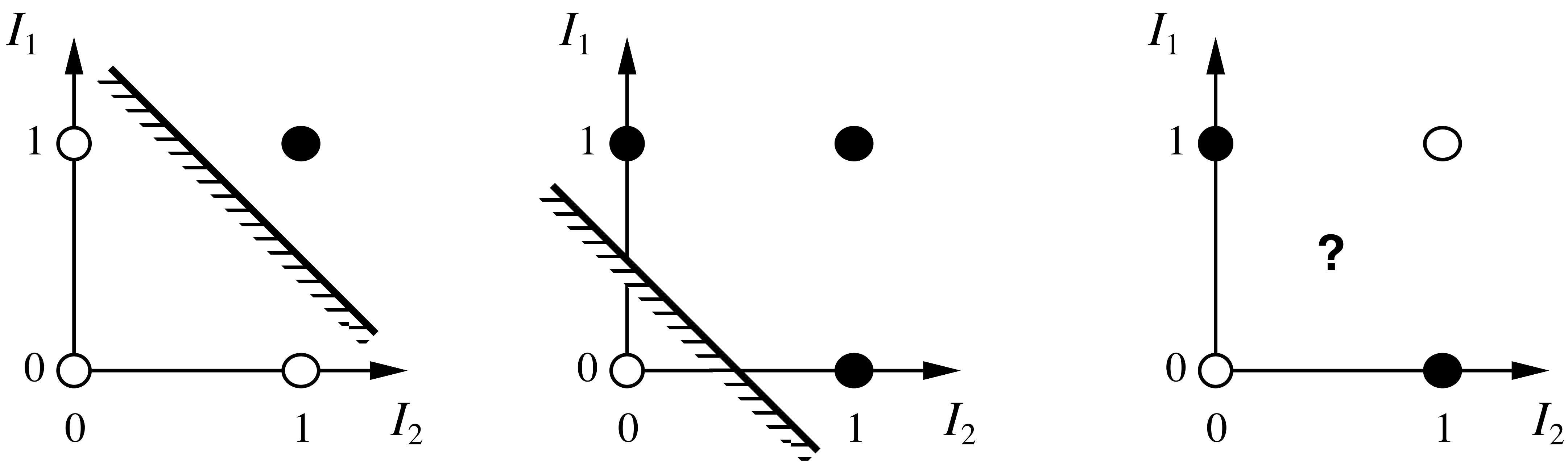
Perceptrons are limited

Consider a perceptron with $g =$ step function (Rosenblatt, 1957, 1960)

Can represent AND, OR, NOT, majority, etc.

Represents a linear separator in input space:

$\Sigma_j W_j x_j > 0$ or $\mathbf{W} \cdot \mathbf{x} > 0$



(a) I_1 **and** I_2 (b) I_1 **or** I_2 (c) I_1 **xor** I_2

Generalized Linear Models and Deep Learning

- Statistical models represent relationships between the covariates and response in terms of a *systematic* component, and a *random* component.
- Linear regression model: $Y = X\beta + \epsilon$
- Here, $\mu = E(Y) = X\beta$, $E(\epsilon) = 0$, and $cov(\epsilon) = \sigma^2 I$.
- Generalized linear models (GLMs) extend this framework to cases where the response variable is binary (e..g, classification) or discrete (e.g,. prediction of counts).

Link Functions

- In GLMs, the concept of a *link function* is fundamental
- The link function represents the relationship between a linear predictor and the response mean $E(Y)$.
- In linear regression, $\eta = X\beta$, and $E(Y) = \mu$, so the linl function is an identity (because $\eta = \mu$).
- If the response is a binary variable, or a probability, the link function has to be modified.
- Linear regression also assumes the variances are constant, but in many problems, variances may depend on the mean.

Logit Function and Logistic Regression

- If the response variable y is binary, we need to change the way the linear predictor is coupled to the response.
- One approach is to use the logistic function:

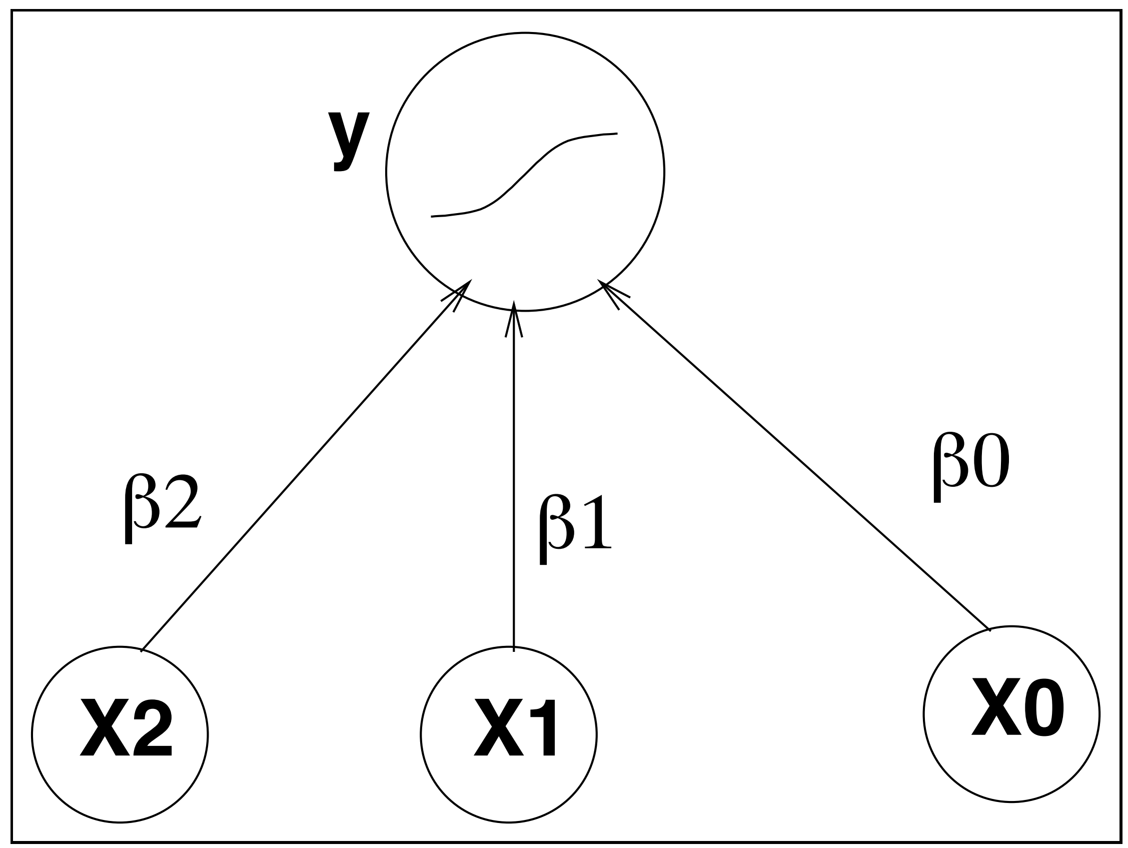
$$P(y=1|x,\beta) = \mu(x|\beta) = \frac{e^{\beta^T x}}{1+e^{\beta^T x}} = \frac{1}{1+e^{-\beta^T x}}$$

$$P(y=0|x,\beta) = 1-\mu(x|\beta) = \frac{1}{1+e^{\beta^T x}}$$

- Inverting the above transformation gives us the *logit* function

$$g(x|\beta) = \log\frac{\mu(x|\beta)}{1-\mu(x|\beta)} = \beta^T x$$

Logistic Regression



Maximum Likelihood Estimation

- Consider fitting a logistic regression model to a dataset of n observations $X = (x^1, y^1), \ldots, (x^n, y^n)$.
- The *conditional likelihood* of a single observation is

$$P(y^i | x^i, \beta) = \mu(x^i | \beta)^{y^i} (1 - \mu(x^i | \beta))^{1 - y^i}$$

- The conditional likelihood of the entire dataset is

$$P(Y | X, \beta) = \prod_{i=1}^{n} \mu(x^i | \beta)^{y^i} (1 - \mu(x^i | \beta))^{1 - y^i}$$

Newton Raphson Method

- Newton's method finds the roots of an equation $f(\theta) = 0$.

$$\theta_{t+1} = \theta_t - \frac{f(\theta_t)}{f'(\theta_t)}$$

- Newton's method finds the minimum of a function f.
- The maximum of a function $f(\theta)$ is exactly when its derivative $f'(\theta) = 0$.

$$\theta_{t+1} = \theta_t - \frac{f'(\theta_t)}{f''(\theta_t)}$$

Newton Raphson Method

- The gradient of the log likelihood can be written in matrix form as

$$\frac{\partial l(\beta|X,Y)}{\partial\beta} = \sum_{i=1}^{n} x^i(y^i - \mu(x^i|\beta)) = X^T(Y-P)$$

- The Hessian can be written as $\frac{\partial^2 l(\beta|X,Y)}{\partial\beta\partial\beta^T} = -X^TWX$
- The Newton-Raphson algorithm then becomes

$$\begin{aligned}
\beta^{new} &= \beta^{old} + (X^TWX)^{-1}X^T(Y-P) \\
&= (X^TWX)^{-1}X^TW\left(X\beta^{old} + W^{-1}(Y-P)\right) \\
&= (X^TWX)^{-1}X^TWZ \qquad \text{where } Z \equiv X\beta^{old} + W^{-1}(Y-P)
\end{aligned}$$

Stochastic Gradient Method

- Newton's method can be expensive since it involves computing and inverting the Hessian matrix.
- Stochastic gradient methods are slower, but computationally cheaper at each time step.

$$\frac{\partial l(\beta|x,y)}{\partial \beta_j} = x_j(y - \mu(x|\beta))$$

- The stochastic gradient ascent rule can be written as (for instance (x^i, y^i))

$$\beta_j \leftarrow \beta_j + \alpha(y^i - \mu(x^i|\beta))x^i_j$$

Logistic Regression in Theano

```
class LogisticRegression(object):
    """Multi-class Logistic Regression Class

    The logistic regression is fully described by a weight matrix :math:`W`
    and bias vector :math:`b`. Classification is done by projecting data
    points onto a set of hyperplanes, the distance to which is used to
    determine a class membership probability.
    """

    def __init__(self, input, n_in, n_out):
        """ Initialize the parameters of the logistic regression

        :type input: theano.tensor.TensorType
        :param input: symbolic variable that describes the input of the
                      architecture (one minibatch)

        :type n_in: int
        :param n_in: number of input units, the dimension of the space in
                     which the datapoints lie

        :type n_out: int
        :param n_out: number of output units, the dimension of the space in
                      which the labels lie
```

MNIST problem

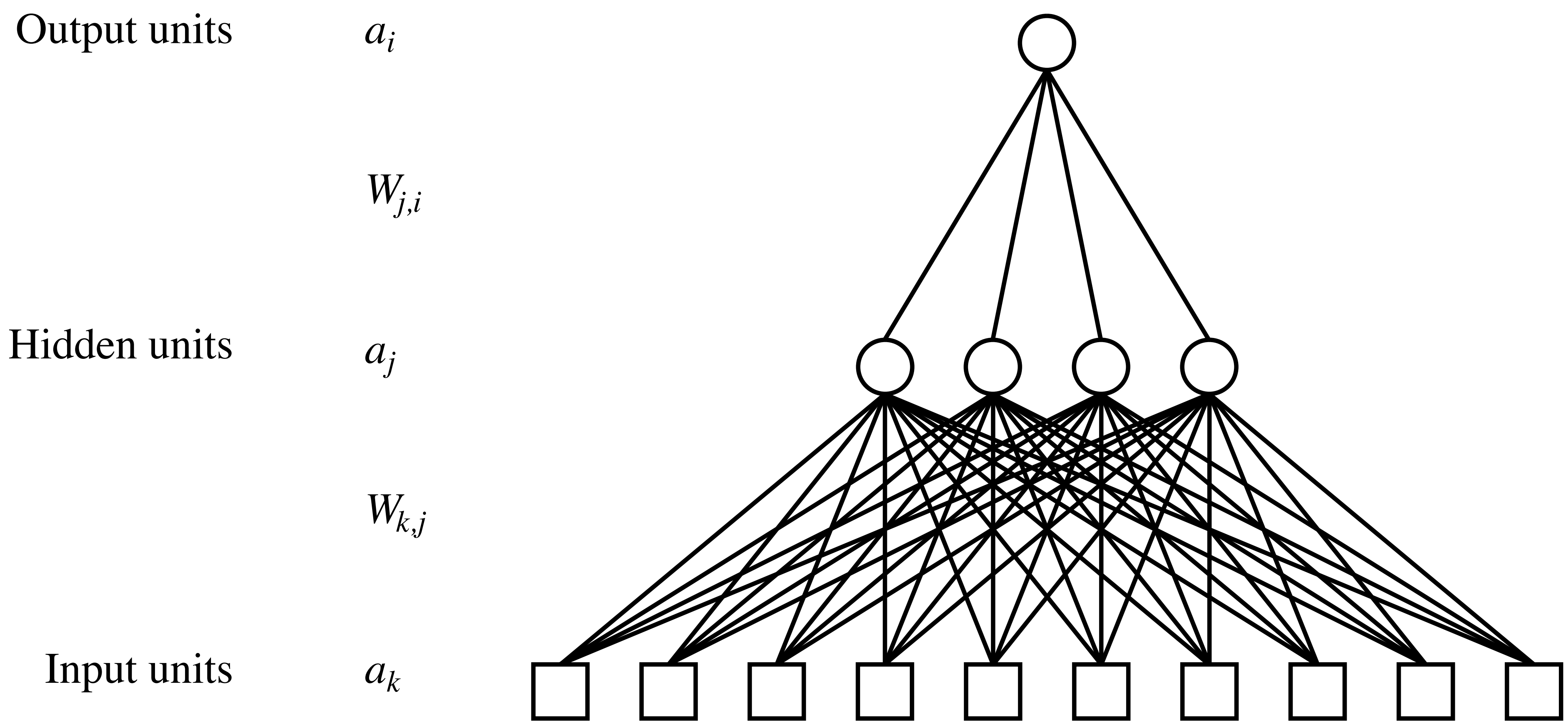
Logistic Regression:MNIST

http://deeplearning.net/tutorial/logreg.html#logreg

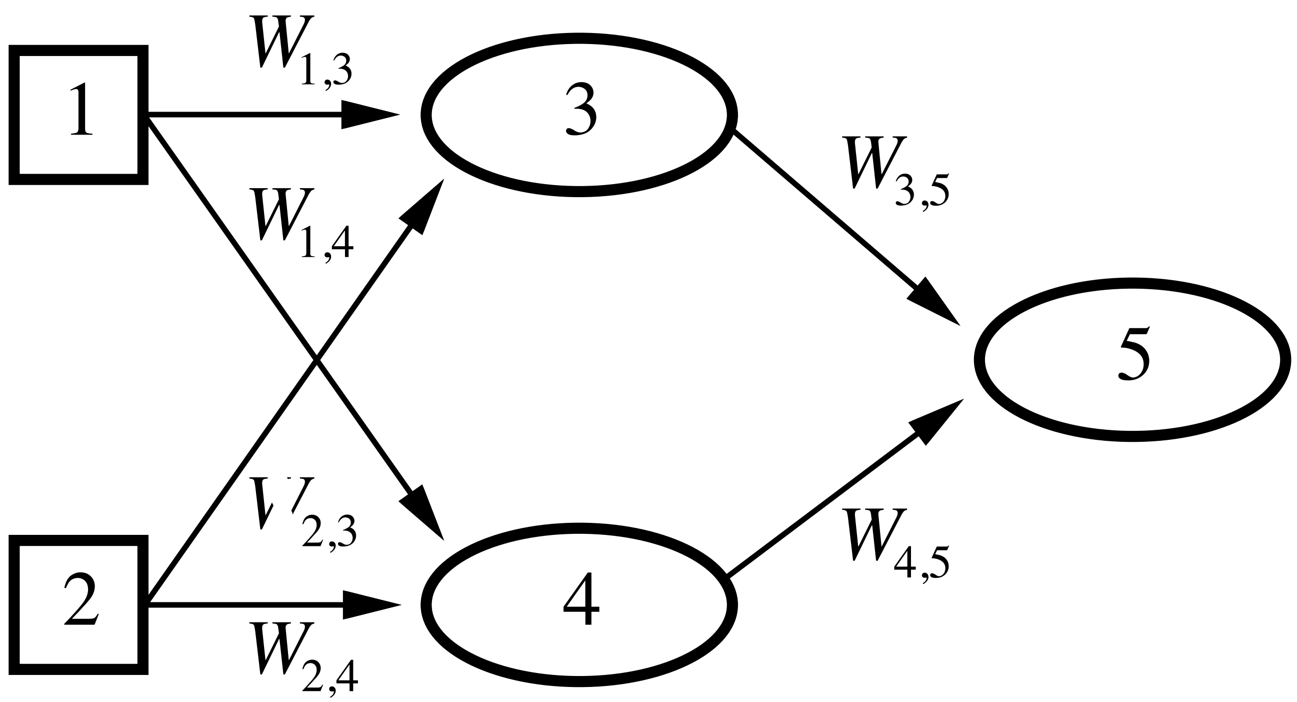
```
mahadeva@manifold:~/Documents/courses/Deep Learning Course UMass Fall 2015/code$ python logistic_
Using gpu device 0: Tesla K80
Downloading data from http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz
... loading data
... building the model
... training the model
epoch 1, minibatch 83/83, validation error 12.458333 %
    epoch 1, minibatch 83/83, test error of best model 12.375000 %
epoch 2, minibatch 83/83, validation error 11.010417 %
    epoch 2, minibatch 83/83, test error of best model 10.958333 %
epoch 3, minibatch 83/83, validation error 10.312500 %

epoch 73, minibatch 83/83, validation error 7.500000 %
    epoch 73, minibatch 83/83, test error of best model 7.489583 %
Optimization complete with best validation score of 7.500000 %,with test performance 7.489583 %
The code run for 74 epochs, with 24.234342 epochs/sec
```

Multilayer Perceptrons

Layers are usually fully connected;
numbers of hidden units typically chosen by hand

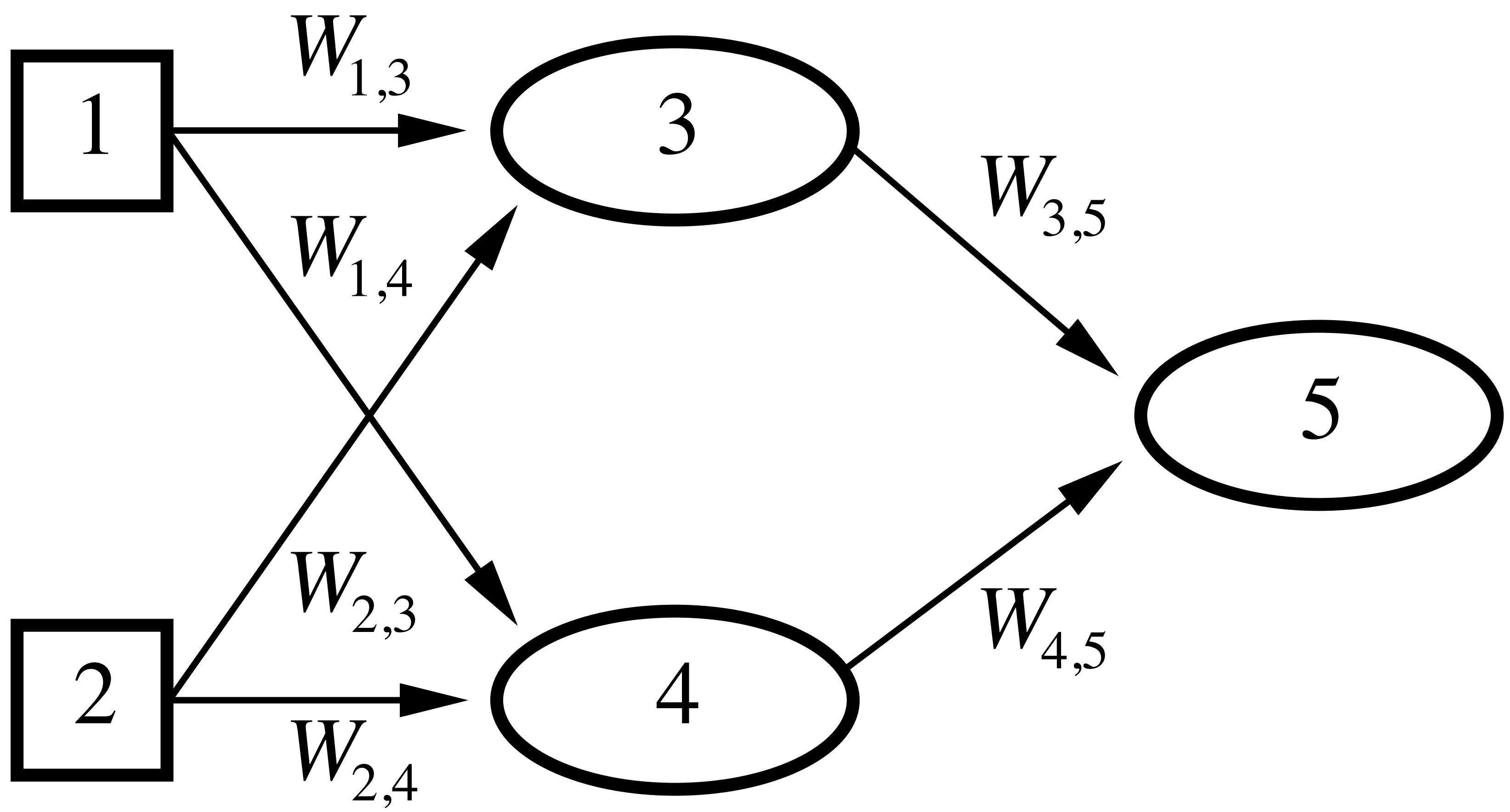


What's hard about training feedforward networks?



There are training signals for the output and input layers. But, what are the hidden nodes supposed to compute?

Feedforward Networks



Feed-forward network = a parameterized family of nonlinear functions:

$$
\begin{aligned}
a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\
&= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))
\end{aligned}
$$

Gradient Learning Rule

Learn by adjusting weights to reduce error on training set

The squared error for an example with input $\mathbf{x}$ and true output y is

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2\ ,$$

Perform optimization search by gradient descent:

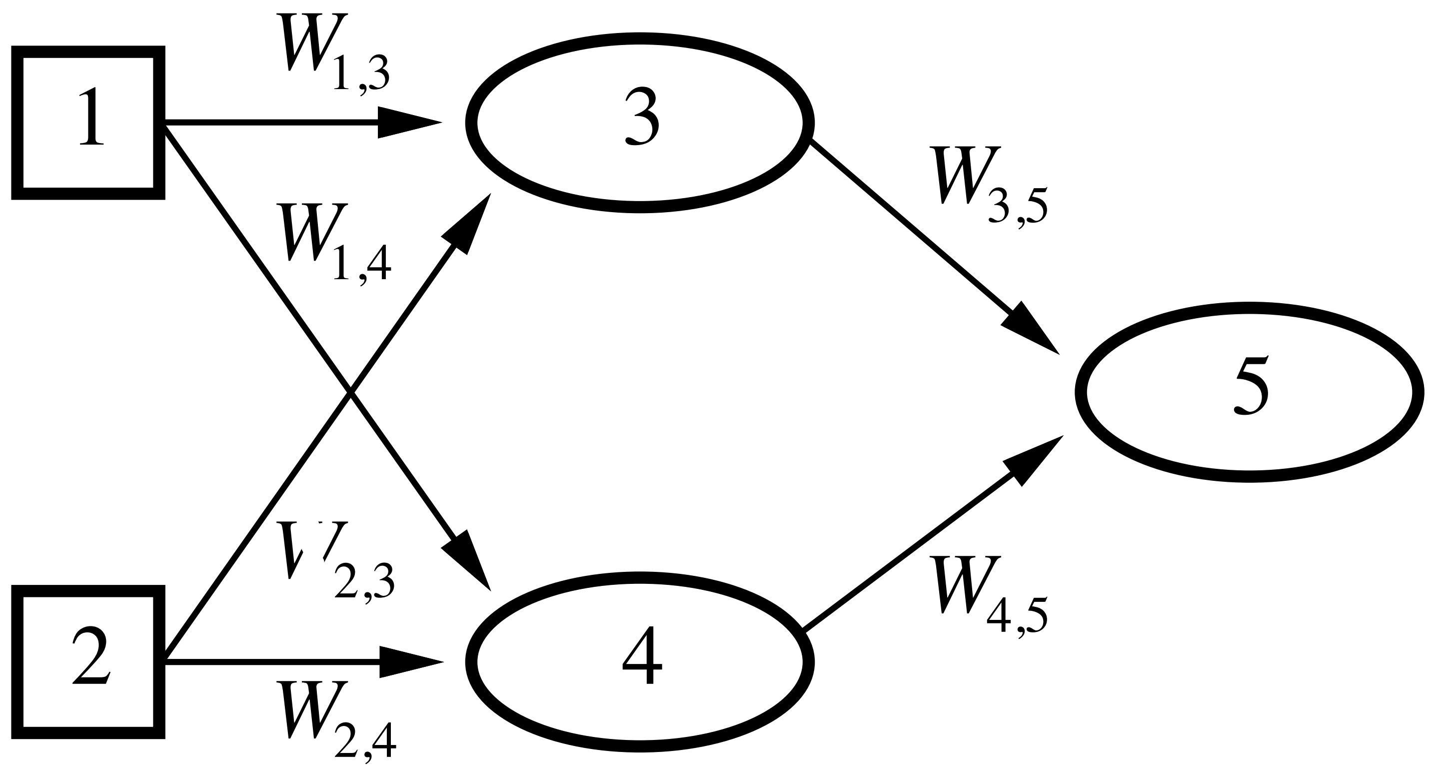
$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\textstyle\sum_{j=0}^{n} W_j x_j)\right) \\ &= -Err \times g'(in) \times x_j\end{aligned}$$

Simple weight update rule:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

E.g., $+$ve error $\Rightarrow$ increase network output
$\Rightarrow$ increase weights on $+$ve inputs, decrease on -ve inputs

Backpropagation



Forward propagation: compute activation levels of each unit on a particular input

Backpropagation: compute errors

Gradient Training Rule

The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 \ ,$$

where the sum is over the nodes in the output layer.

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

Hidden Units

$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
\end{aligned}
$$

Backpropagation Algorithm

- Given: training examples $\{(x_i, y_i)\}$, network
- Randomly set initial weights of network
- Repeat
  - For each training example
    - Compute error beginning with output units, and then for each hidden layer of units
    - Adjust weights in direction of lower error
- Until error is acceptable

Backpropagation Algorithm

- Initialize weights to small random values
- REPEAT
  - For each training example:
    - FORWARD PROPAGATION: Fix network inputs using training example and compute network outputs
    - BACKPROPAGATION:
- For output unit k, compute delta value $\Delta_k = a_k\ (1\text{-}a_k)(t_k - a_k)$
- Compute delta values of hidden units

$$\Delta_h = a_h\ (1 - a_h)\ \Sigma_k W_{hk}\ \Delta_k$$

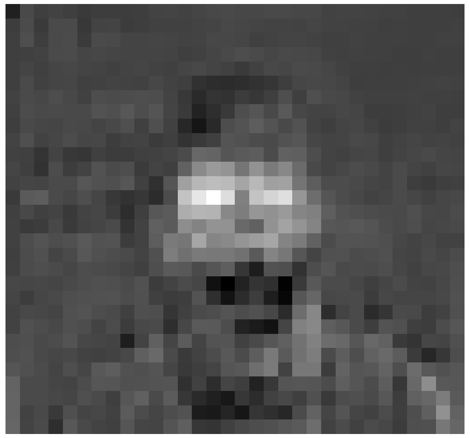
- Update each network weight

$$W_{ij} = W_{ij} + \eta\ a_i\ \Delta_j$$

Facial Pose Detection

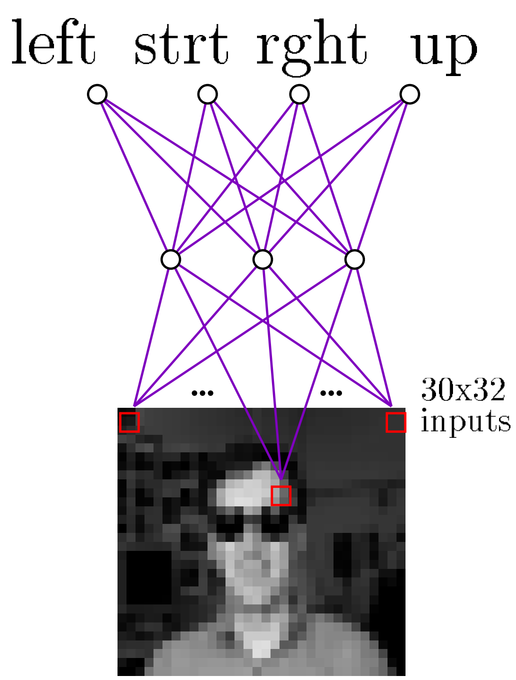
Tom Mitchell (CMU)

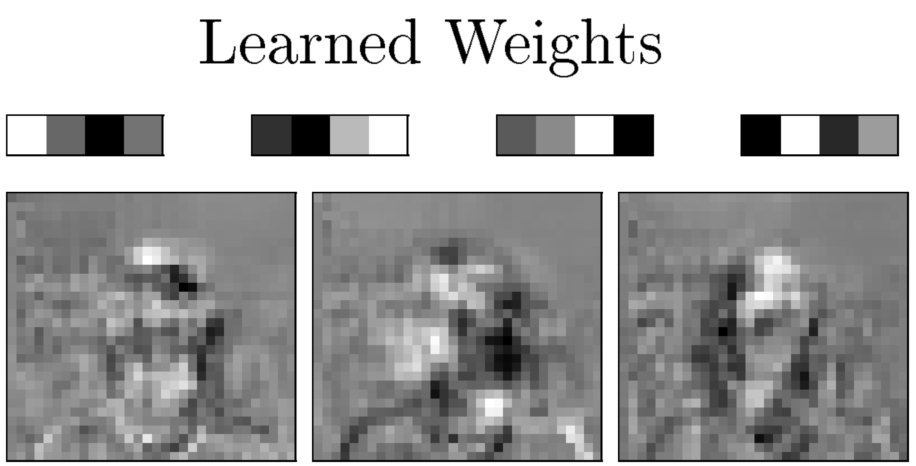
“Hinton” diagram (showing activation of hidden units)



“Sunglass detector”

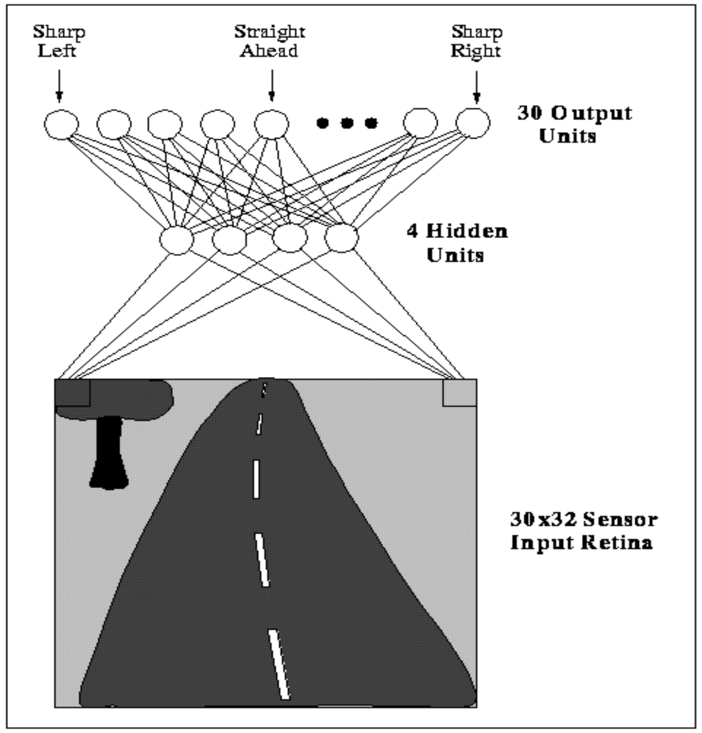
Hidden Unit Detectors





ALVINN

ALVINN learns from a human driver



Neural Network

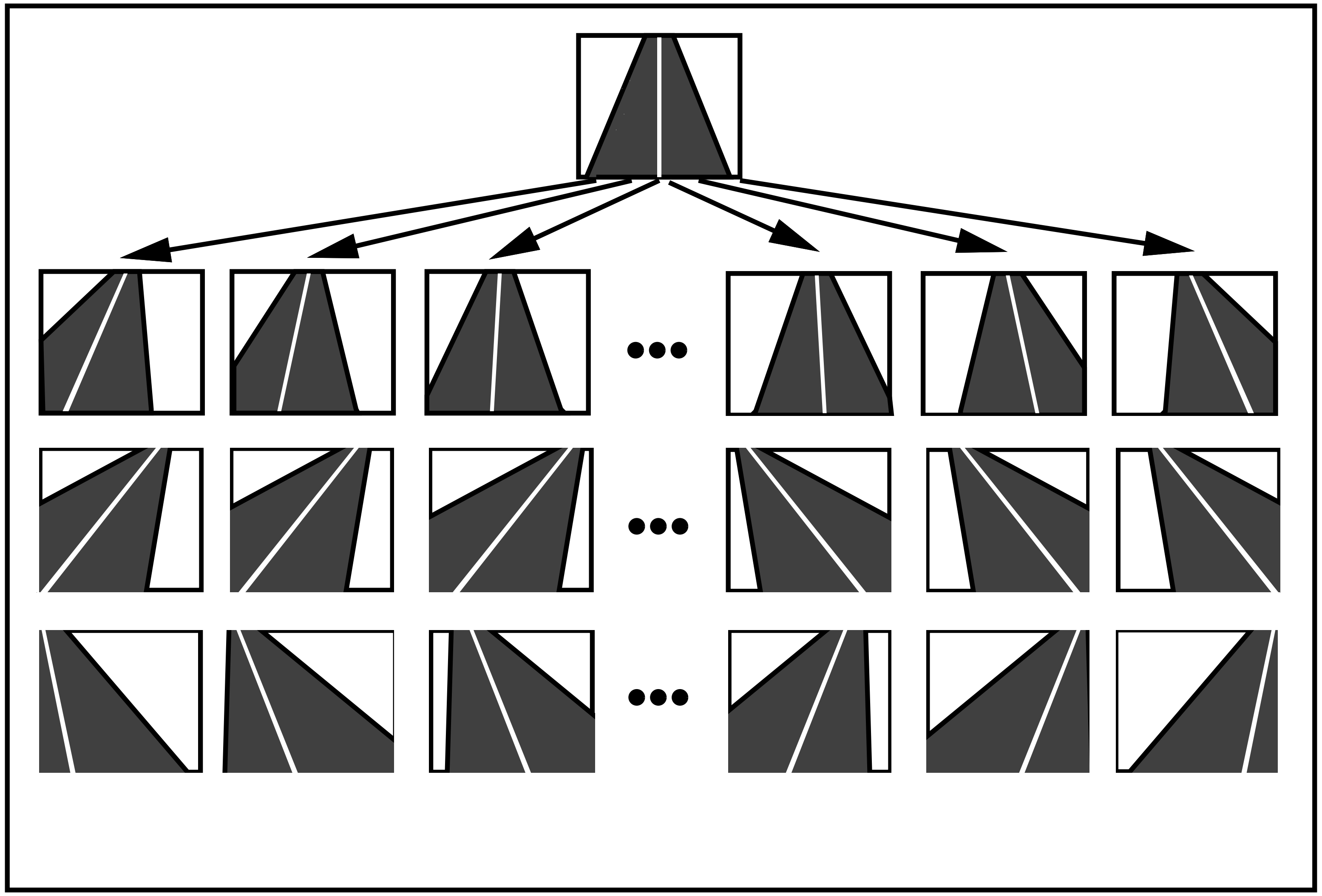
Can drive on actual highways at 70 miles per hour!

ALVINN training

Examples of roads traversed by ALVINN

ALVINN training

Synthetic training data created from actual data



MNIST problem

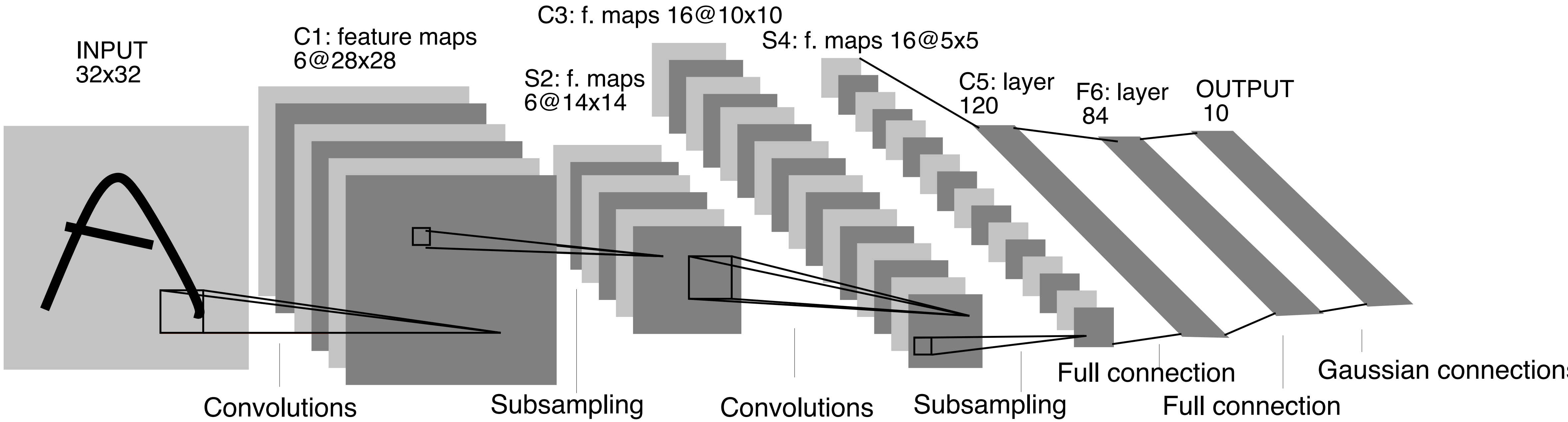
MNIST using feedforward networks in Theano

http://deeplearning.net/tutorial/code/mlp.py

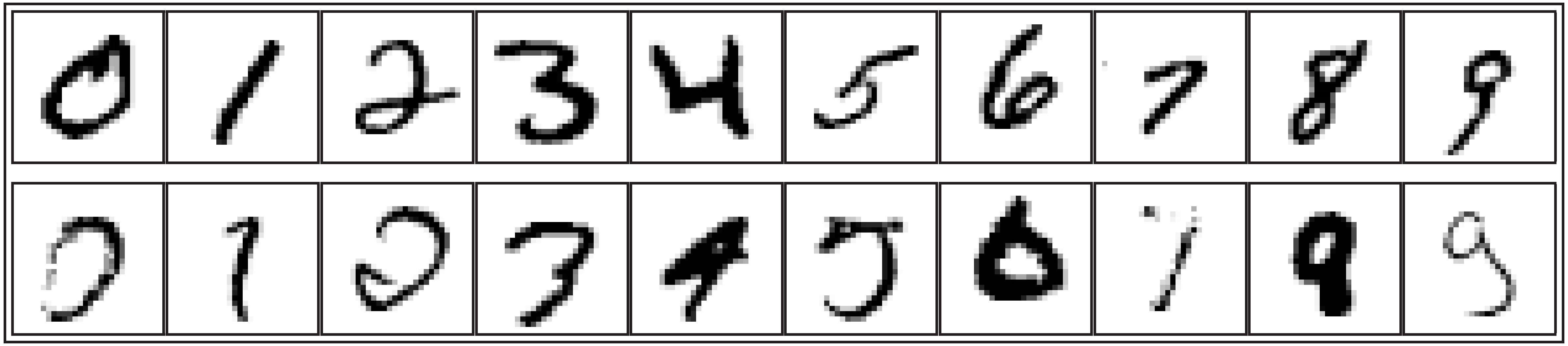
epoch 995, minibatch 2500/2500, validation error 1.700000 %
epoch 996, minibatch 2500/2500, validation error 1.700000 %
epoch 997, minibatch 2500/2500, validation error 1.700000 %
epoch 998, minibatch 2500/2500, validation error 1.700000 %
epoch 999, minibatch 2500/2500, validation error 1.700000 %
epoch 1000, minibatch 2500/2500, validation error 1.700000 %
Optimization complete. Best validation score of 1.690000 % obtained at iteration 2070000, with test performance 1.650000 %
The code for file mlp.py ran for 45.72m

LeNet Network

(Le Cun, 1998)



Digit Recognition



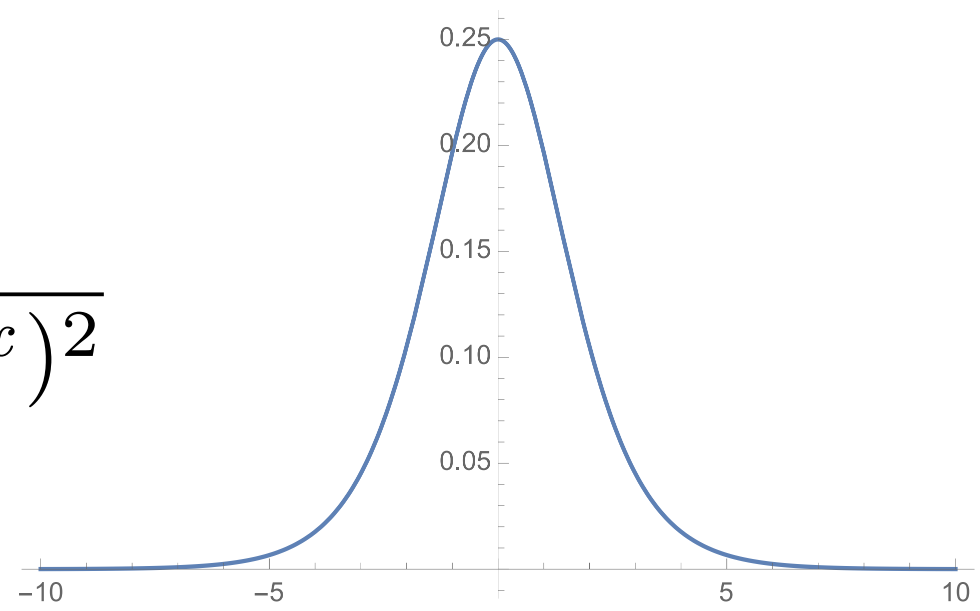
3-nearest-neighbor $=$ 2.4% error

400–300–10 unit MLP $=$ 1.6% error

LeNet: 768–192–30–10 unit MLP $=$ 0.9%

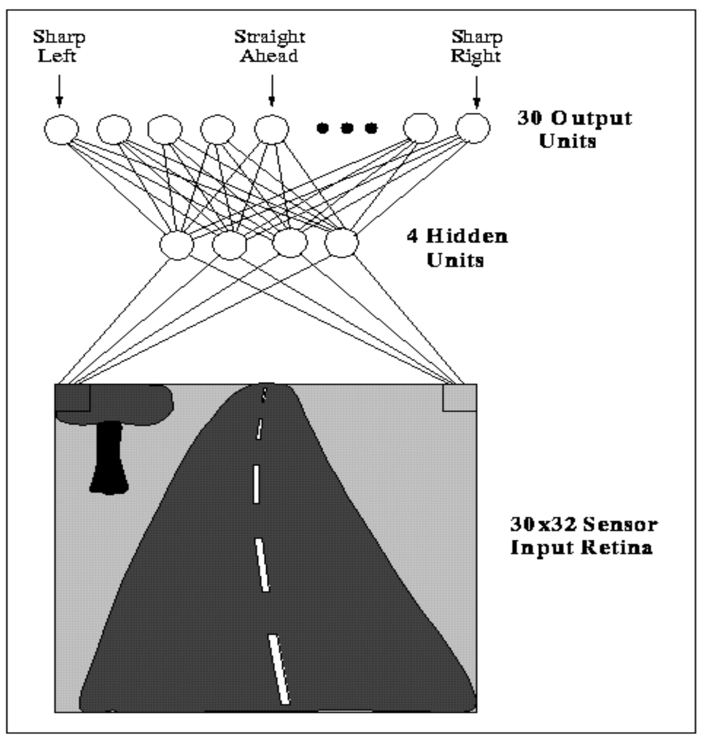
Gradient of Sigmoid

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$



Vanishing gradient problem!

1990 vs. 2015

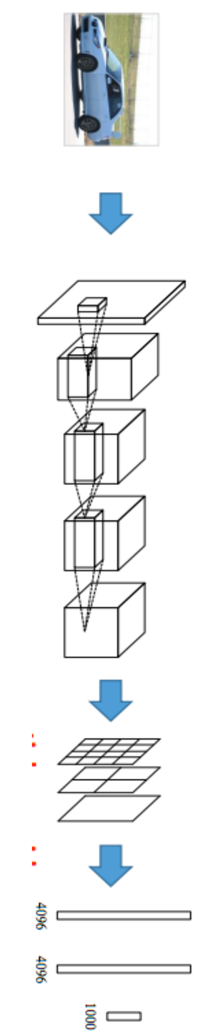


Learning to drive

Year 2014

GoogLeNet VGG MSRA

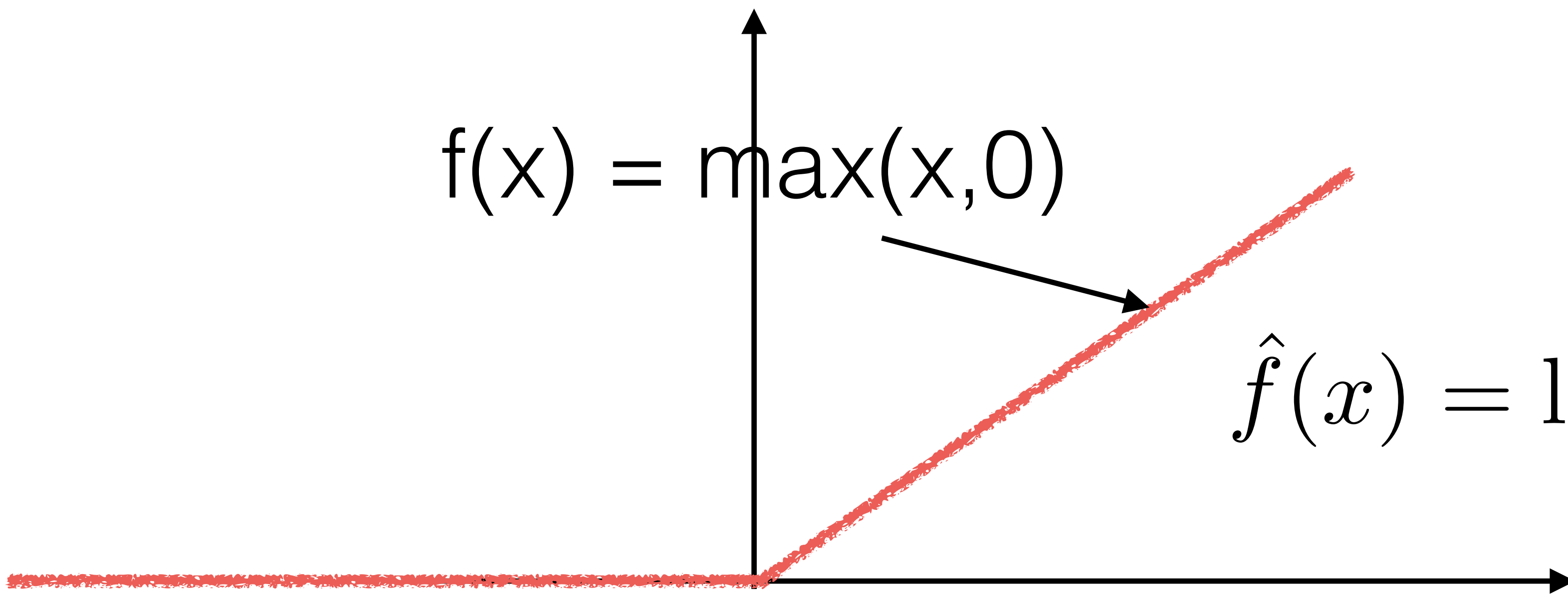
Convolution
Pooling
Softmax
Other



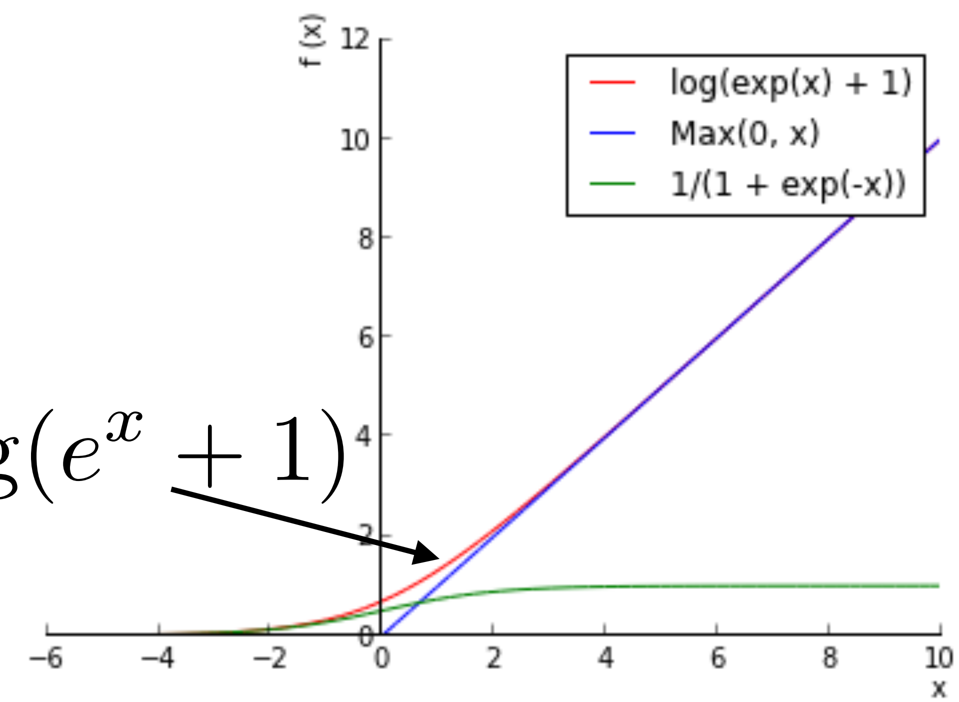
[Szegedy arxiv 2014] [Simonyan arxiv 2014] [He arxiv 2014]

Rectified Linear Units

f(x) = max(x,0)



$$\hat{f}(x) = \log(e^x + 1)$$



RLUs for speech recognition

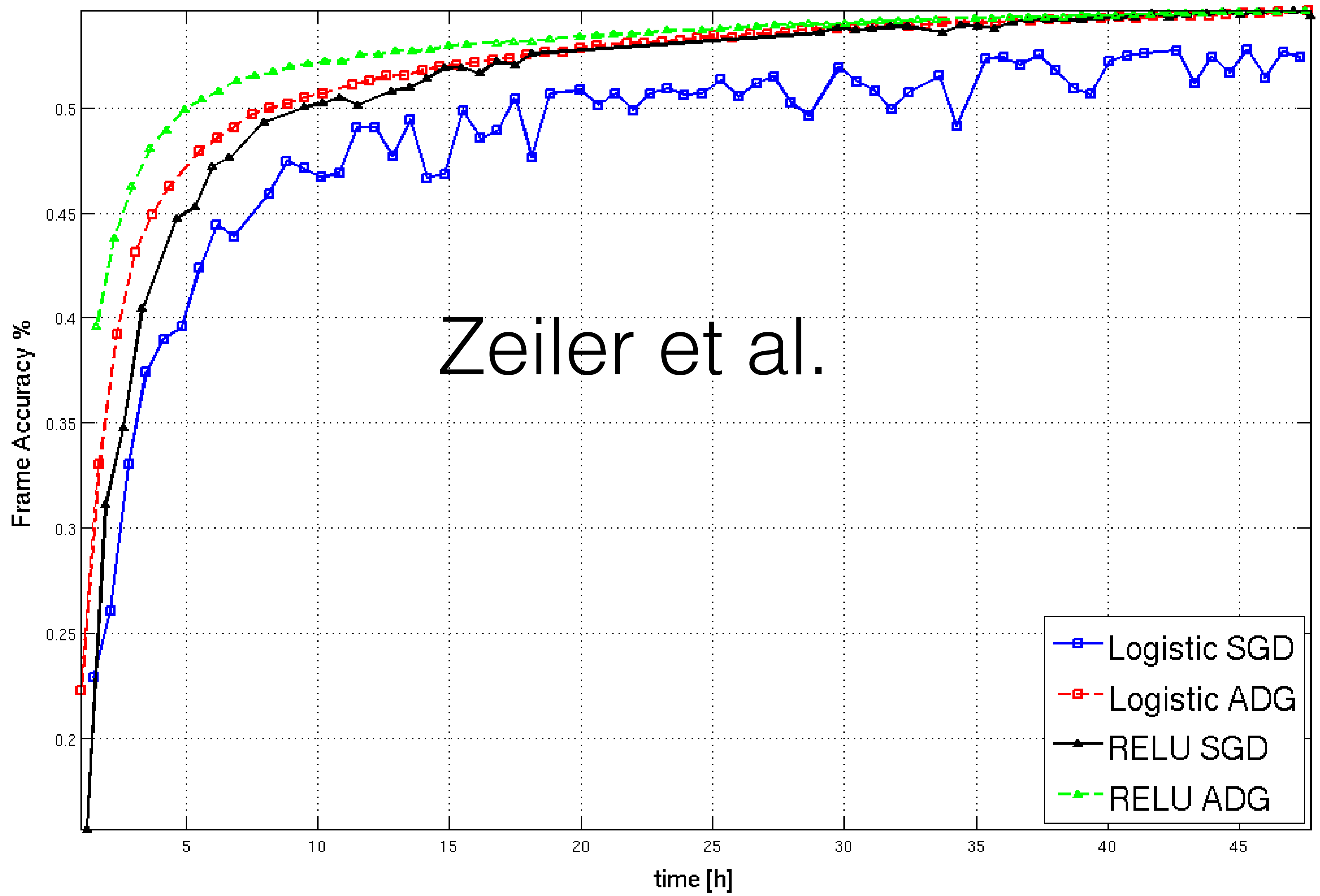
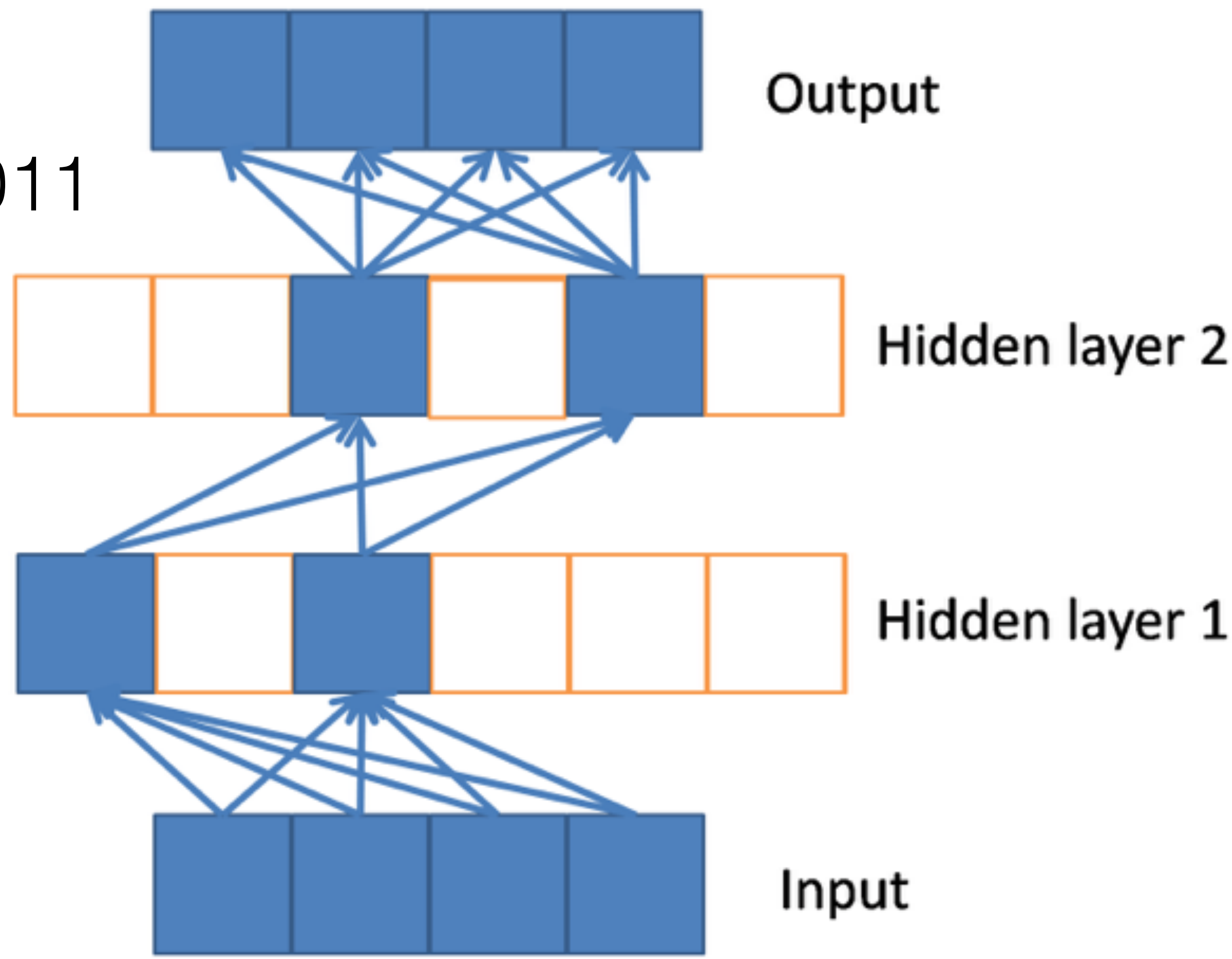


Fig. 3. Frame accuracy as a function of time for a 4 hidden layer neural net trained with either logistic or ReLUs and using as optimizer either SGD or SGD with Adagrad (ADG).

Sparse propagation

Glorot et al., 2011



Specifying LeNet in Caffe

https://developers.google.com/protocol-buffers/docs/overview

```
name: "LeNet"
```

Data layer

```
layer {
  name: "mnist"
  type: "Data"
  data_param {
    source: "mnist_train_lmdb"
    backend: LMDB
    batch_size: 64
    scale: 0.00390625
  }
  top: "data"
  top: "label"
}
```

Convolution layer

```
layer {
  name: "conv1"
  type: "Convolution"
  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"
    }
  }
  bottom: "data"
  top: "conv1"
}
```

Max Pooling and RLU Layer

```
layer {
  name: "pool1"
  type: "Pooling"
  pooling_param {
    kernel_size: 2
    stride: 2
    pool: MAX
  }
  bottom: "conv1"
  top: "pool1"
}
```

Convolution layer

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

RLU layer

Loss Layer

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
}
```

RLU layer

MNIST solver in Caffe

```
# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: GPU
```

Running LeNet on Caffe

```
I0917 19:20:26.375691 26575 layer_factory.hpp:75] Creating layer mnist
I0917 19:20:26.375877 26575 net.cpp:110] Creating Layer mnist
I0917 19:20:26.375903 26575 net.cpp:432] mnist -> data
I0917 19:20:26.375928 26575 net.cpp:432] mnist -> label
I0917 19:20:26.378226 26581 db_lmdb.cpp:22] Opened lmdb examples/mnist/mnist_test_lmdb
I0917 19:20:26.378762 26575 data_layer.cpp:44] output data size: 100,1,28,28
I0917 19:20:26.380553 26575 net.cpp:155] Setting up mnist
I0917 19:20:26.380594 26575 net.cpp:163] Top shape: 100 1 28 28 (78400)
I0917 19:20:26.380614 26575 net.cpp:163] Top shape: 100 (100)
I0917 19:20:26.380635 26575 layer_factory.hpp:75] Creating layer label_mnist_1_split
I0917 19:20:26.380668 26575 net.cpp:110] Creating Layer label_mnist_1_split
I0917 19:20:26.380686 26575 net.cpp:476] label_mnist_1_split <- label
I0917 19:20:26.380707 26575 net.cpp:432] label_mnist_1_split -> label_mnist_1_split_0
I0917 19:20:26.380738 26575 net.cpp:432] label_mnist_1_split -> label_mnist_1_split_1
```

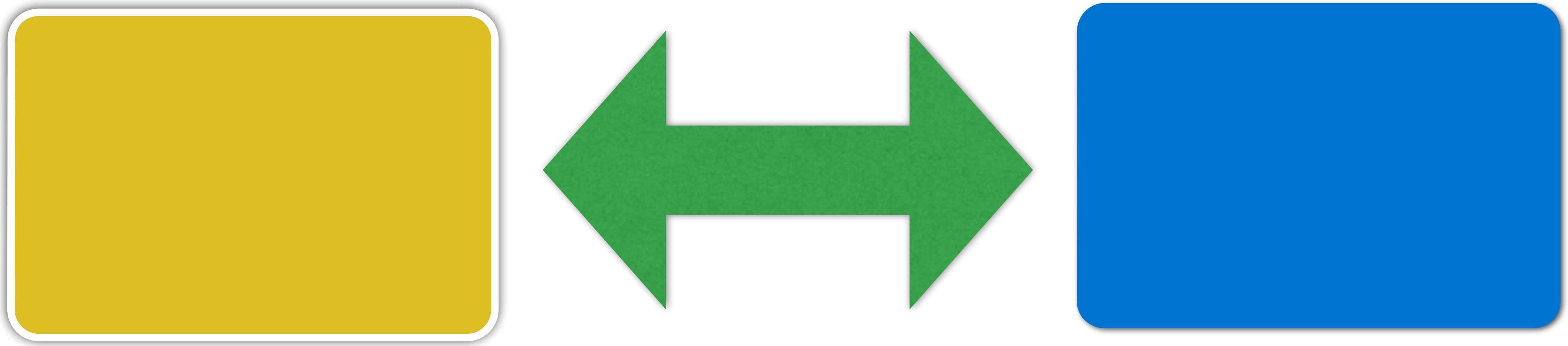
```
I0917 19:20:26.405414 26575 solver.cpp:266] Learning Rate Policy: inv
I0917 19:20:26.406183 26575 solver.cpp:310] Iteration 0, Testing net (#0)
I0917 19:20:26.601101 26575 solver.cpp:359]     Test net output #0: accuracy = 0.0777
I0917 19:20:26.601132 26575 solver.cpp:359]     Test net output #1: loss = 2.3651 (* 1 = 2.3651 loss)
I0917 19:20:26.604207 26575 solver.cpp:222] Iteration 0, loss = 2.34867
I0917 19:20:26.604233 26575 solver.cpp:238]     Train net output #0: loss = 2.34867 (* 1 = 2.34867 loss)
```

```
I0917 19:20:59.081962 26575 solver.cpp:291] Iteration 10000, loss = 0.00325083
I0917 19:20:59.081985 26575 solver.cpp:310] Iteration 10000, Testing net (#0)
I0917 19:20:59.215575 26575 solver.cpp:359]     Test net output #0: accuracy = 0.9904
I0917 19:20:59.215605 26575 solver.cpp:359]     Test net output #1: loss = 0.0291382 (* 1 = 0.0291382 loss)
I0917 19:20:59.215615 26575 solver.cpp:296] Optimization Done.
I0917 19:20:59.215622 26575 caffe.cpp:184] Optimization Done.
```

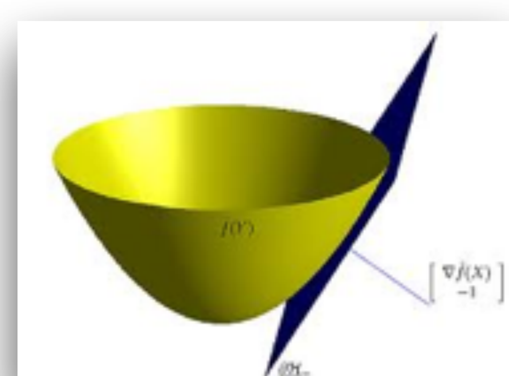
real **0m34.403s**
user 0m27.744s
sys 0m25.308s

New Stochastic Gradient Methods

Dual Primal

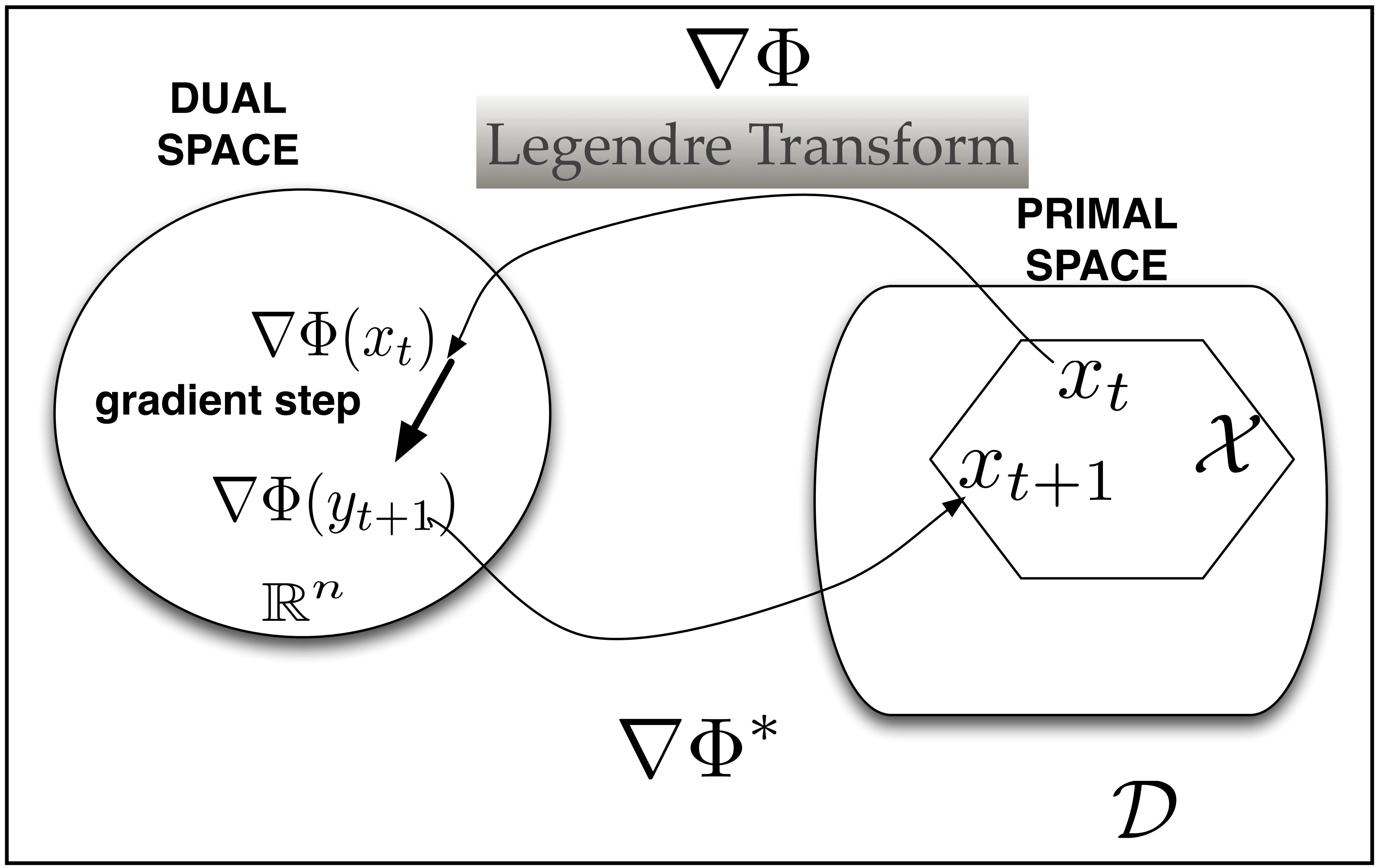


$$x_{k+1} = \nabla\psi^* \left(\nabla\psi(x_k) - t_k \partial f(x_k)\right)$$



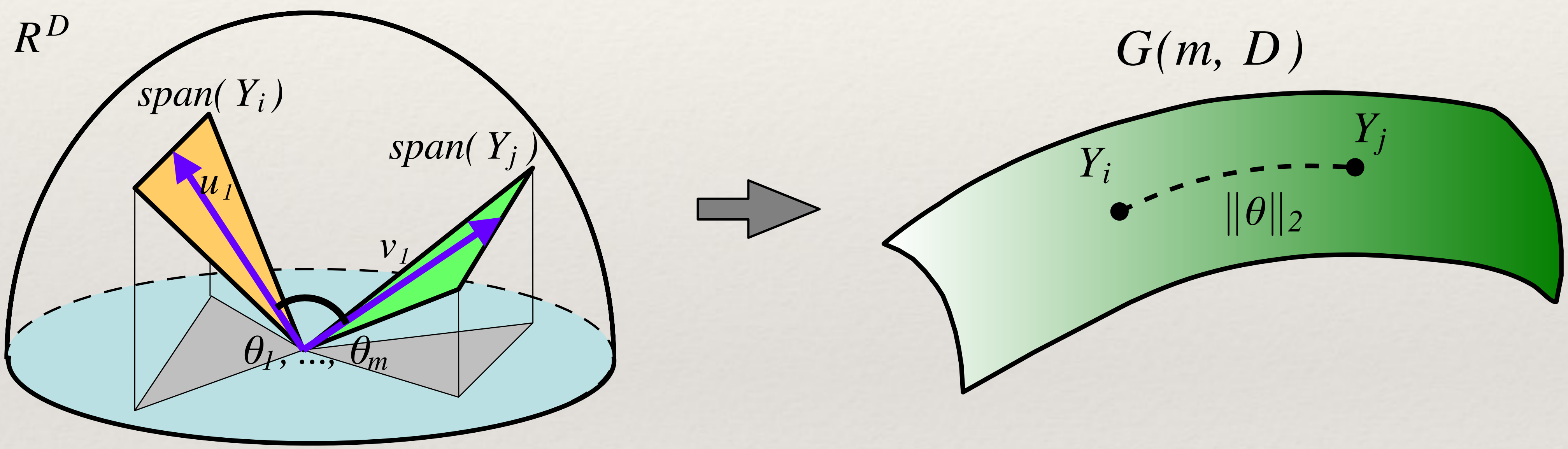
Mirror Maps

(Nemirovski and Yudin, 1980s; Bubeck, 2014)



“Natural” Gradients on Manifolds

In a manifold, gradients live in the tangent space, not in the original space



Riemannian gradient $\dfrac{\nabla F}{\nabla \Phi} = (I - \Phi\Phi^T)\dfrac{\nabla F}{\nabla \Phi}$

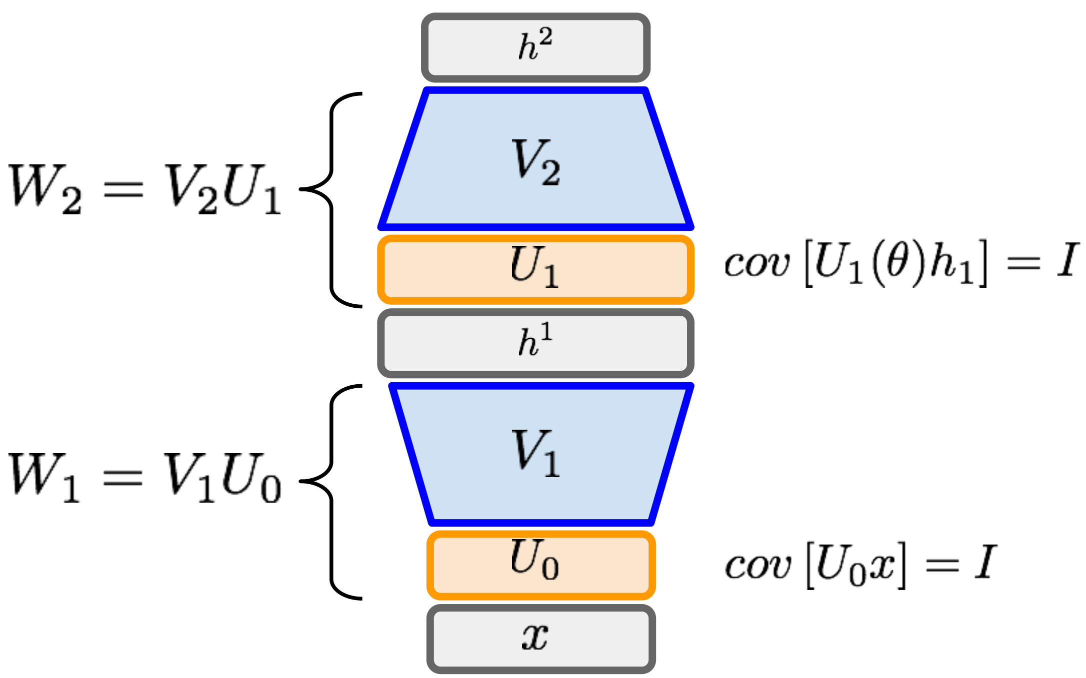
Mirror Descent => “Natural” Gradient

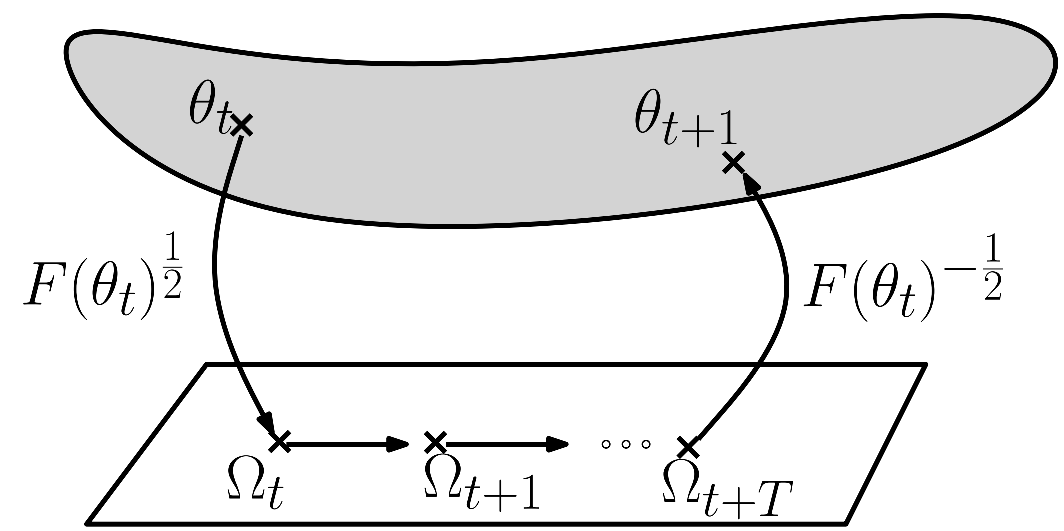
(Nemirovsky and Yudin; Amari, 1980s)

Mirror Descent $\xrightarrow{\text{Mirror Map}}$ Natural gradient

Thomas, Dabney, Mahadevan, Giguere, NIPS 2013

Natural Neural Network (DesJardin et al., Deep Mind, 2015)





Builds on our recent identification of mirror descent and natural gradient methods

Group Midterm Project

- Atari Game Deep Reinforcement Learning
- Each group will be tested on the same suite of Atari problems
- Groups will be given code to run the Atari games and the deep learning package(s)
- Groups are free to modify hyperparameters or introduce architectural innovations

Summary

- Training deep neural networks is an old idea
- The original back propagation idea goes back to the early 80s (or even before!)
- Sigmoid units have the problem of vanishing gradients
- New rectified linear units provide improved results
- Faster stochastic gradient methods are being used
- Start working more actively with Caffe, Theano etc.