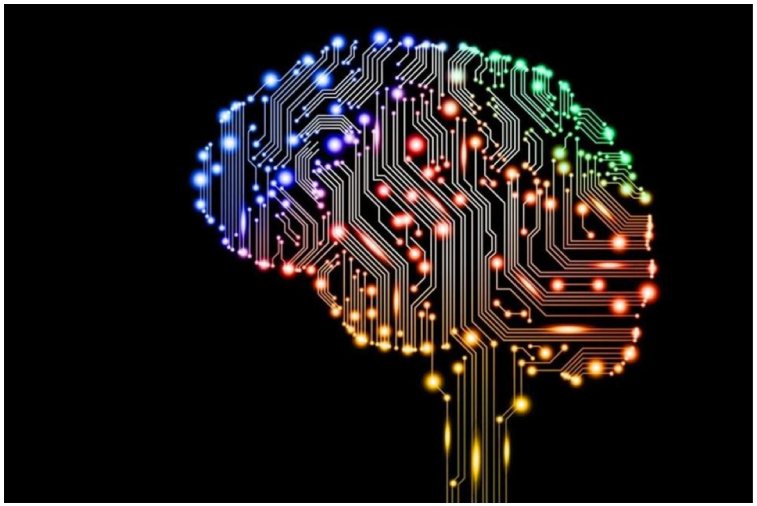
# Deep Learning



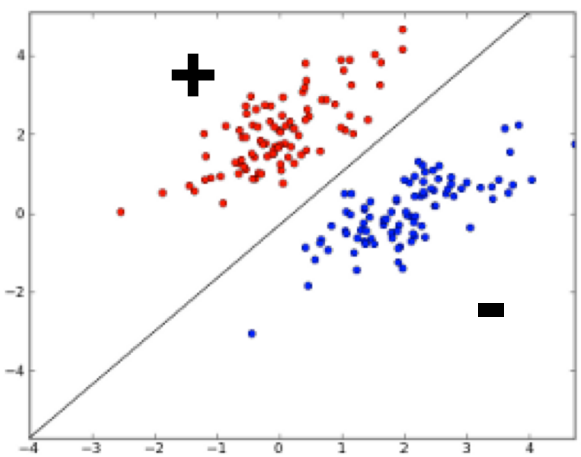
**Jianlin Cheng**
**Department of EECS**
**University of Missouri – Columbia**
**2019**

# Acknowledgements

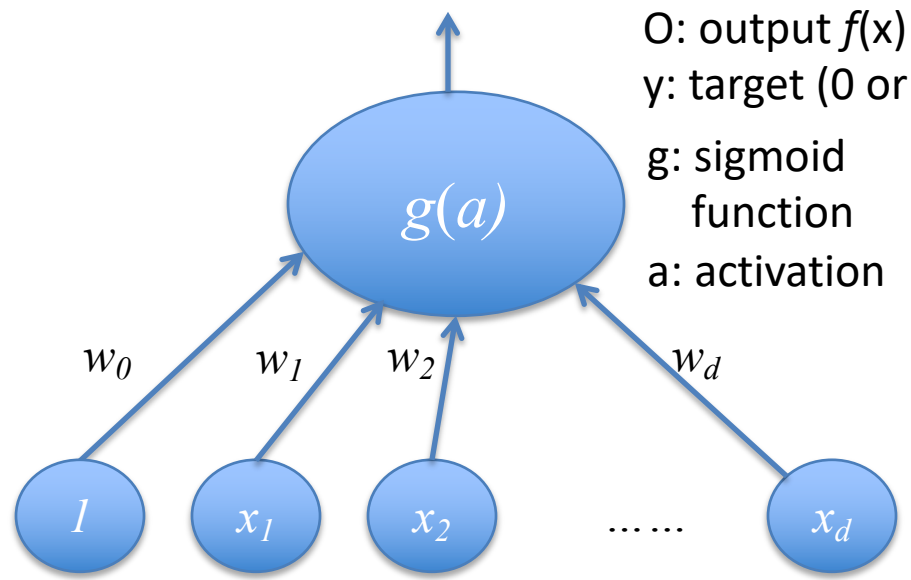
Some content of this teaching presentation was drawn from many sources created by great scientists in the field of deep learning (Hinton, LeCun, Bengio, Ng, et al.).

# Simplest Neural Network for Classification – Logistic Regression

- **Binary Classification**




**Loss/cross-entropy:**
**-(ylog(*o*) + (1-y)log(1- o))**




O: output *f*(x)
y: target (0 or 1)
g: sigmoid function
a: activation

$$activation = w_0 + w_1x_1 + w_2x_2 + \ldots, w_dx_d$$

$$P(y = 1) = f(x) = \frac{1}{1 + e^{-w_0 - \sum w_i x_i}}$$

$$w^{new} = w^{cur} - \frac{\partial Error}{\partial w} = w^{cur} - (f(x) - y)x$$

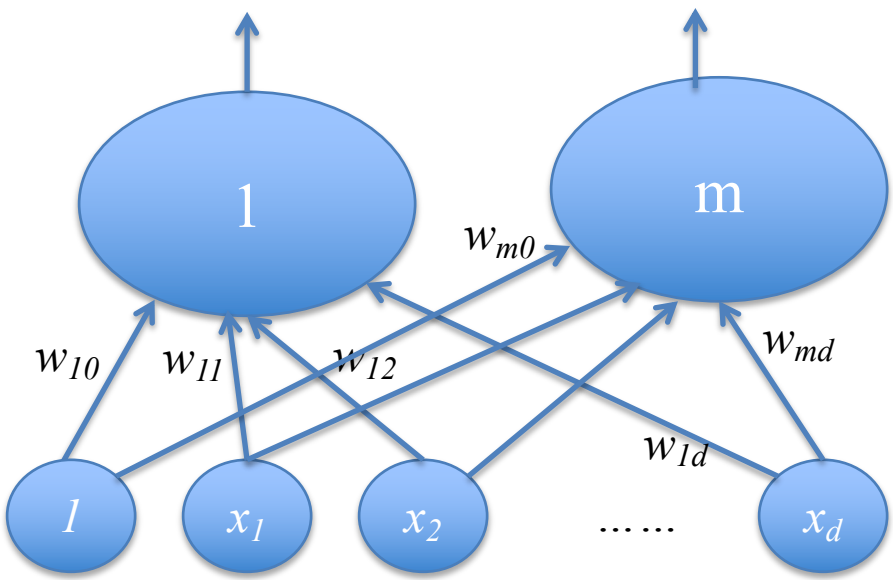
# Simplest Neural Network for Classification – Logistic Regression

- Multi Classification

**Loss/cross-entropy:** $-\sum_{i=1}^{m} y_i \log o_i$

O: output $f$(x)
y: target (0 or 1)

g: softmax
a: activation



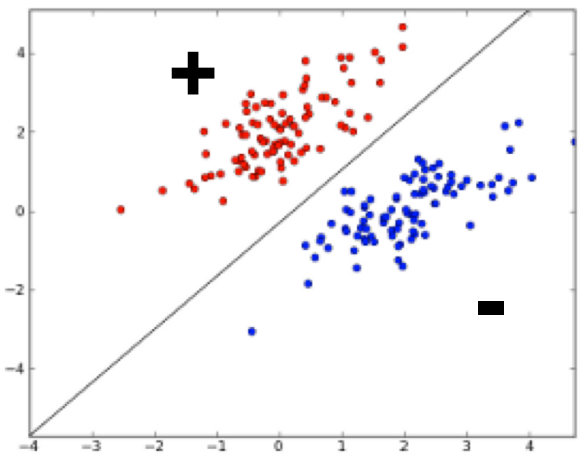

***activation*** $= w_0 + w_1x_1 + w_2x_2 + \ldots, w_dx_d$

$$P(y_i = 1) = f(x) = \frac{e^{w_{i0} + \sum w_{ij}x_j}}{\sum e^{w_{i0} + \sum w_{ij}x_j}}$$

# Perceptron

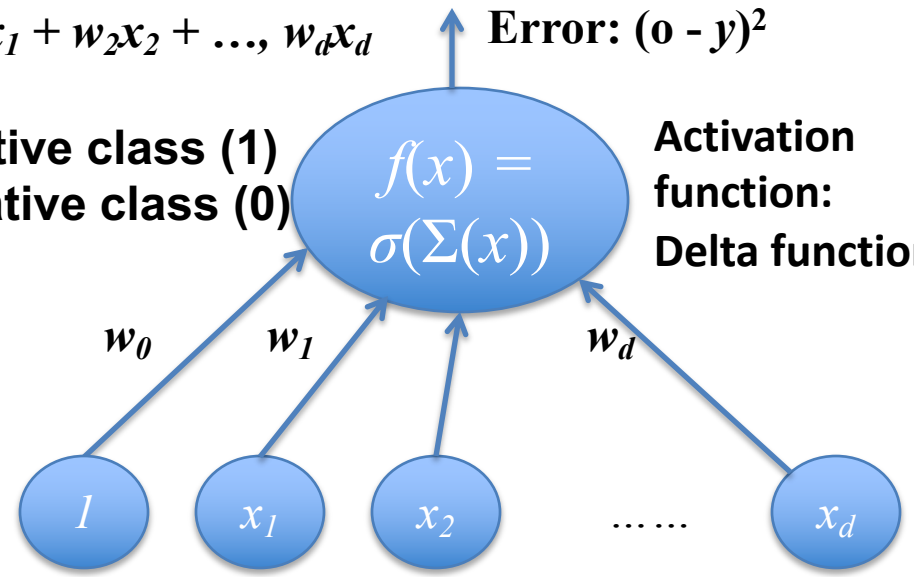
**Learning to map input to output (label) and is guided by output.**

**Training**



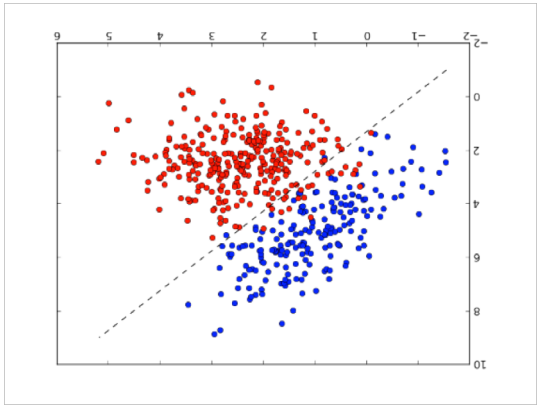

$f(x) = w_0 + w_1x_1 + w_2x_2 + \ldots, w_dx_d$

$f(x) > 0$: positive class (1)
$f(x) < 0$: negative class (0)




**Learning is to adjust w to minimize the squared error between $f(x)$ and true $y$.**

**Testing**



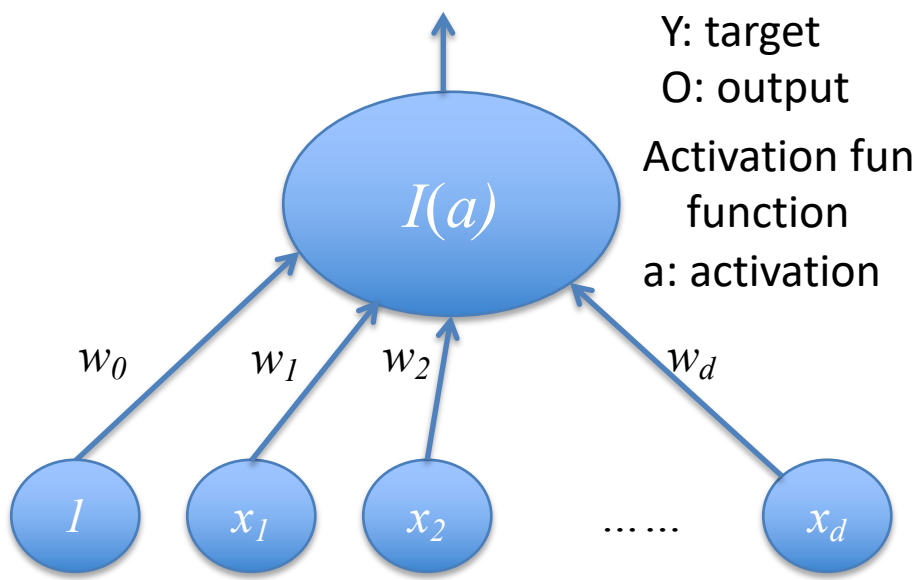
Label information

$$w^{new} = w^{cur} - \frac{\partial Error}{\partial w} = w^{cur} - (o - y)x$$

# Perceptron – 1960s

# Simplest Neural Network for Regression – Linear Regression

**Error: $(o\text{-}y)^2$**




Y: target
O: output
Activation function: identity function
a: activation

Output = $w_0 + w_1x_1 + w_2x_2 + \ldots, w_dx_d$

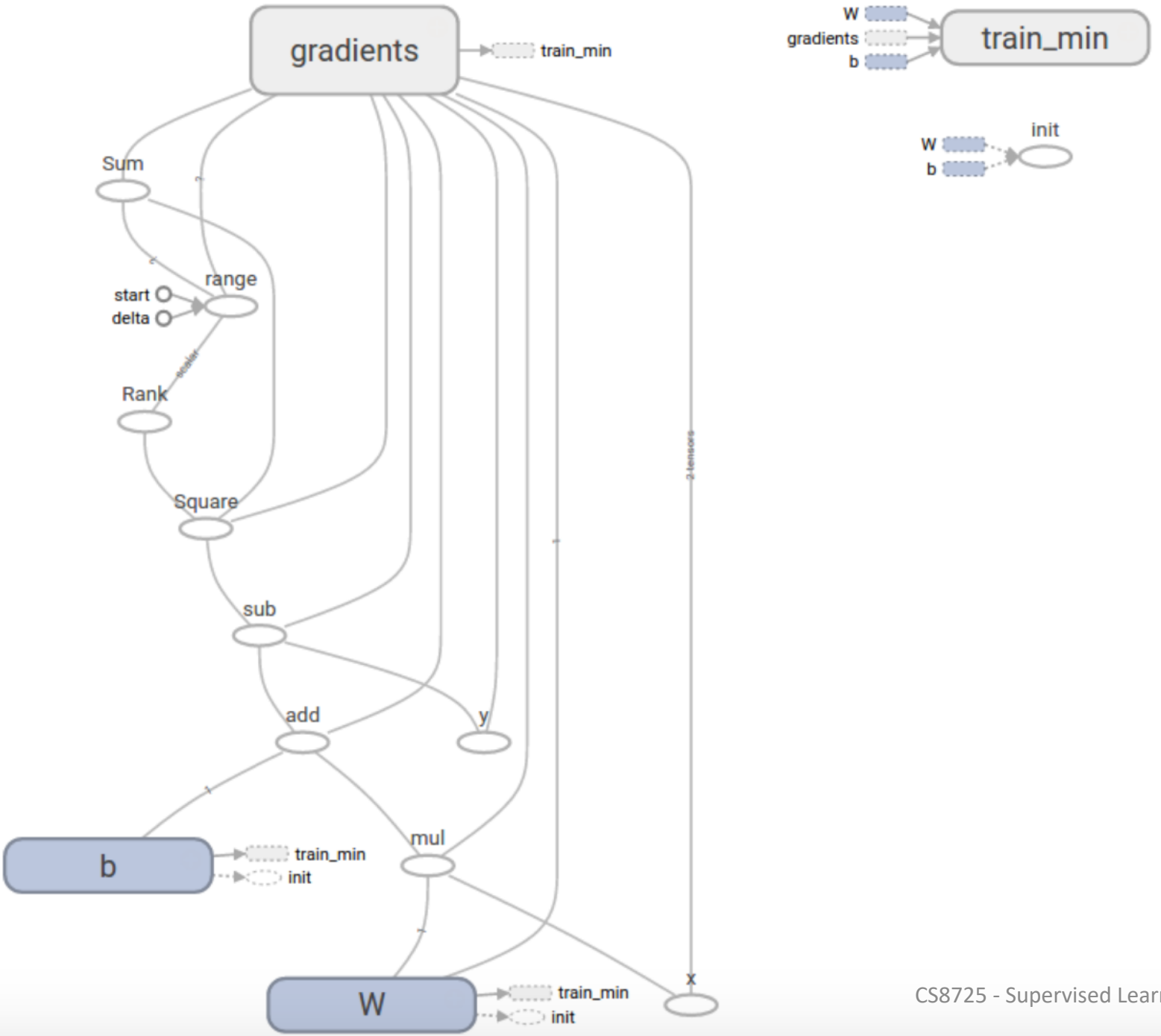
$$w^{new} = w^{cur} - \frac{\partial Error}{\partial w} = w^{cur} - (o - y)x$$

# TensorFlow Demo of One-Node Network

- Data Sets: MNIST digit recognition data or Iris flower classification data
- Google's TensorFlow installation: https://www.tensorflow.org/
- Install it on mac: https://www.tensorflow.org/install/install_mac
- Activate tensorflow: $source ~/tensorflow/bin/activate

# Tensor Flow Model for Linear Regression I

```python
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([-.3], dtype=tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)

# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)

# training data
x_train = [1, 2, 3, 4]
y_train = [0, -1, -2, -3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x: x_train, y: y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss = sess.run([W, b, loss], {x: x_train, y: y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

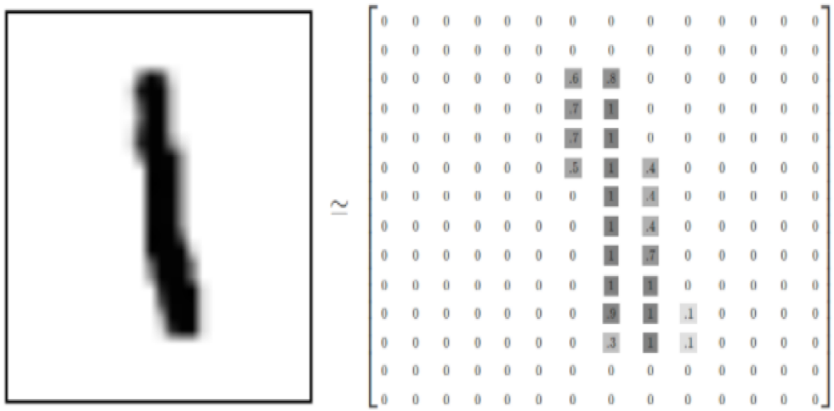

gradients
train_min
W
gradients
b
train_min
init
W
b
Sum
range
start
delta
Rank
Square
sub
add
y
b
train_min
init
mul
W
train_min
init
x

# Tensor Flow Linear Model II

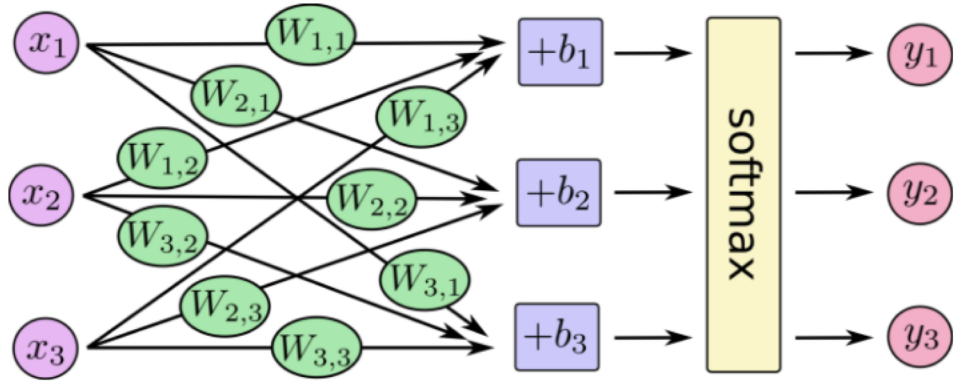
```python
import tensorflow as tf
# NumPy is often used to load, manipulate and preprocess data.
import numpy as np

# Declare list of features. We only have one numeric feature. There are many
# other types of columns that are more complicated and useful.
feature_columns = [tf.feature_column.numeric_column("x", shape=[1])]

# An estimator is the front end to invoke training (fitting) and evaluation
# (inference). There are many predefined types like linear regression,
# linear classification, and many neural network classifiers and regressors.
# The following code provides an estimator that does linear regression.
estimator = tf.estimator.LinearRegressor(feature_columns=feature_columns)

# TensorFlow provides many helper methods to read and set up data sets.
# Here we use two data sets: one for training and one for evaluation
# We have to tell the function how many batches
# of data (num_epochs) we want and how big each batch should be.
x_train = np.array([1., 2., 3., 4.])
y_train = np.array([0., -1., -2., -3.])
x_eval = np.array([2., 5., 8., 1.])
y_eval = np.array([-1.01, -4.1, -7, 0.])
input_fn = tf.estimator.inputs.numpy_input_fn(
    {"x": x_train}, y_train, batch_size=4, num_epochs=None, shuffle=True)
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    {"x": x_train}, y_train, batch_size=4, num_epochs=1000, shuffle=False)
eval_input_fn = tf.estimator.inputs.numpy_input_fn(
    {"x": x_eval}, y_eval, batch_size=4, num_epochs=1000, shuffle=False)

# We can invoke 1000 training steps by invoking the  method and passing the
# training data set.
estimator.train(input_fn=input_fn, steps=1000)

# Here we evaluate how well our model did.
train_metrics = estimator.evaluate(input_fn=train_input_fn)
eval_metrics = estimator.evaluate(input_fn=eval_input_fn)
print("train metrics: %r"% train_metrics)
print("eval metrics: %r"% eval_metrics)
```

# Logistic Regression for Hand Writing Classification

Each image is 28 pixels by 28 pixels. We can interpret this as a big array of numbers:



We can flatten this array into a vector of 28x28 = 784 numbers. It doesn't matter how we flatten the array, as long as we're consistent between images. From this perspective, the MNIST images are just a bunch of points in a 784-dimensional vector space, with a very rich structure (warning: computationally intensive visualizations).




If we write that out as equations, we get:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{matrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{matrix}\right)$$

We can "vectorize" this procedure, turning it into a matrix multiplication and vector addition. This is helpful for computational efficiency. (It's also a useful way to think.)

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right)$$

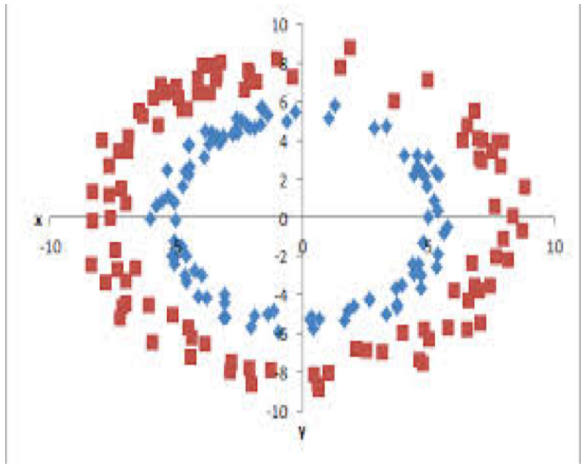
More compactly, we can just write:

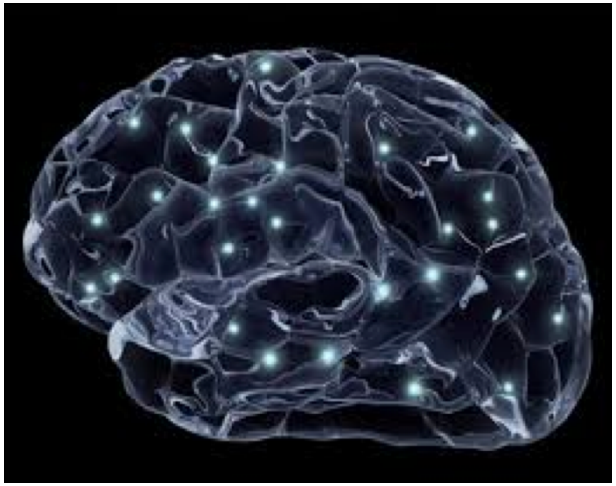
$$y = \text{softmax}(Wx + b)$$

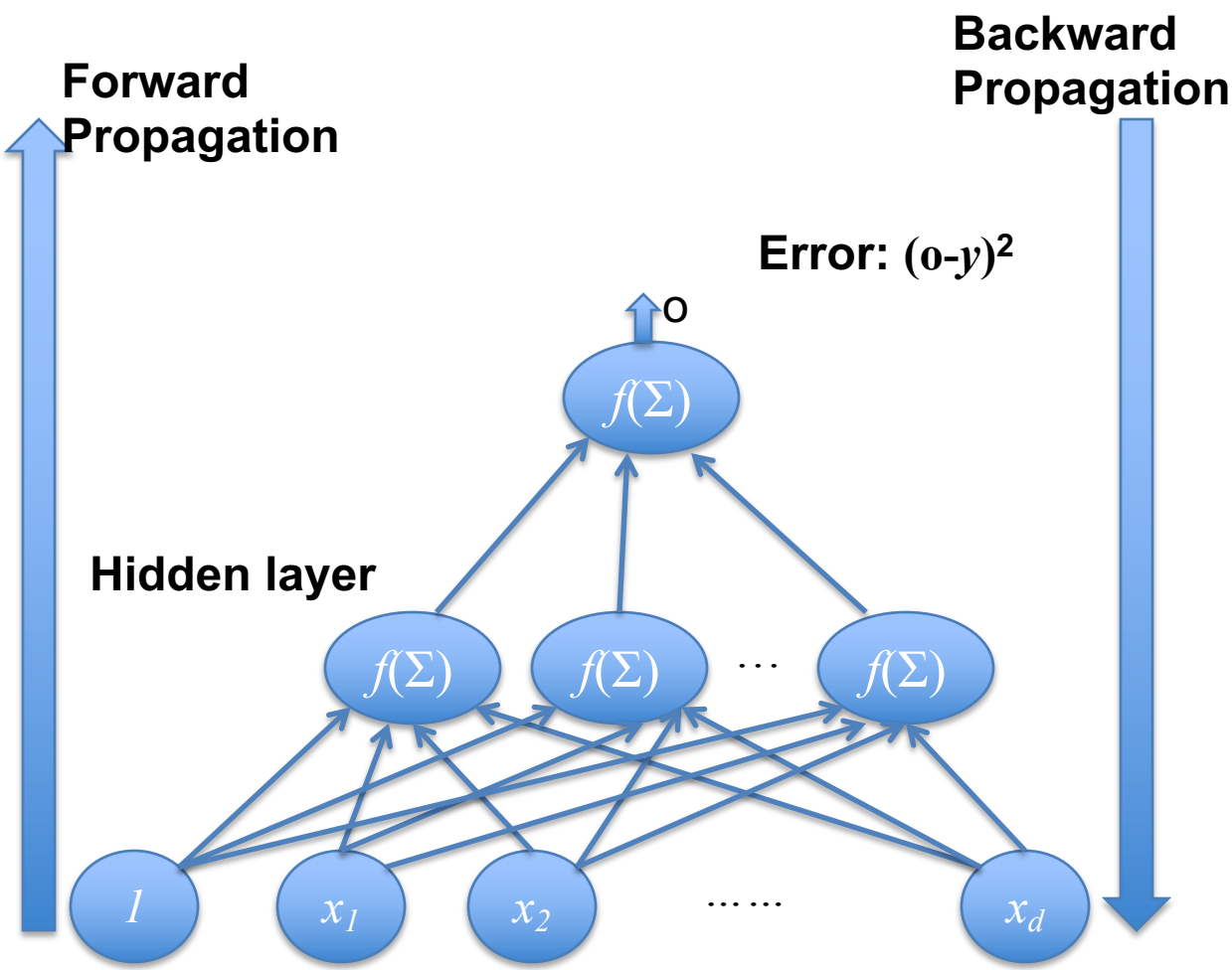
```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys

from tensorflow.examples.tutorials.mnist import input_data

import tensorflow as tf

FLAGS = None


def main(_):
  # Import data
  mnist = input_data.read_data_sets(FLAGS.data_dir, one_hot=True)

  # Create the model
  x = tf.placeholder(tf.float32, [None, 784])
  W = tf.Variable(tf.zeros([784, 10]))
  b = tf.Variable(tf.zeros([10]))
  y = tf.matmul(x, W) + b

  # Define loss and optimizer
  y_ = tf.placeholder(tf.float32, [None, 10])

  # The raw formulation of cross-entropy,
  #
```

```
  #   tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(tf.nn.softmax(y)),
  #                                 reduction_indices=[1]))
  #
  # can be numerically unstable.
  #
  # So here we use tf.nn.softmax_cross_entropy_with_logits on the raw
  # outputs of 'y', and then average across the batch.
  cross_entropy = tf.reduce_mean(
      tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
  train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

  sess = tf.InteractiveSession()
  tf.global_variables_initializer().run()
  # Train
  for _ in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

  # Test trained model
  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
  print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                      y_: mnist.test.labels}))

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('--data_dir', type=str, default='/tmp/tensorflow/mnist/input_data',
                      help='Directory for storing input data')
  FLAGS, unparsed = parser.parse_known_args()
  tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

# Neural Network



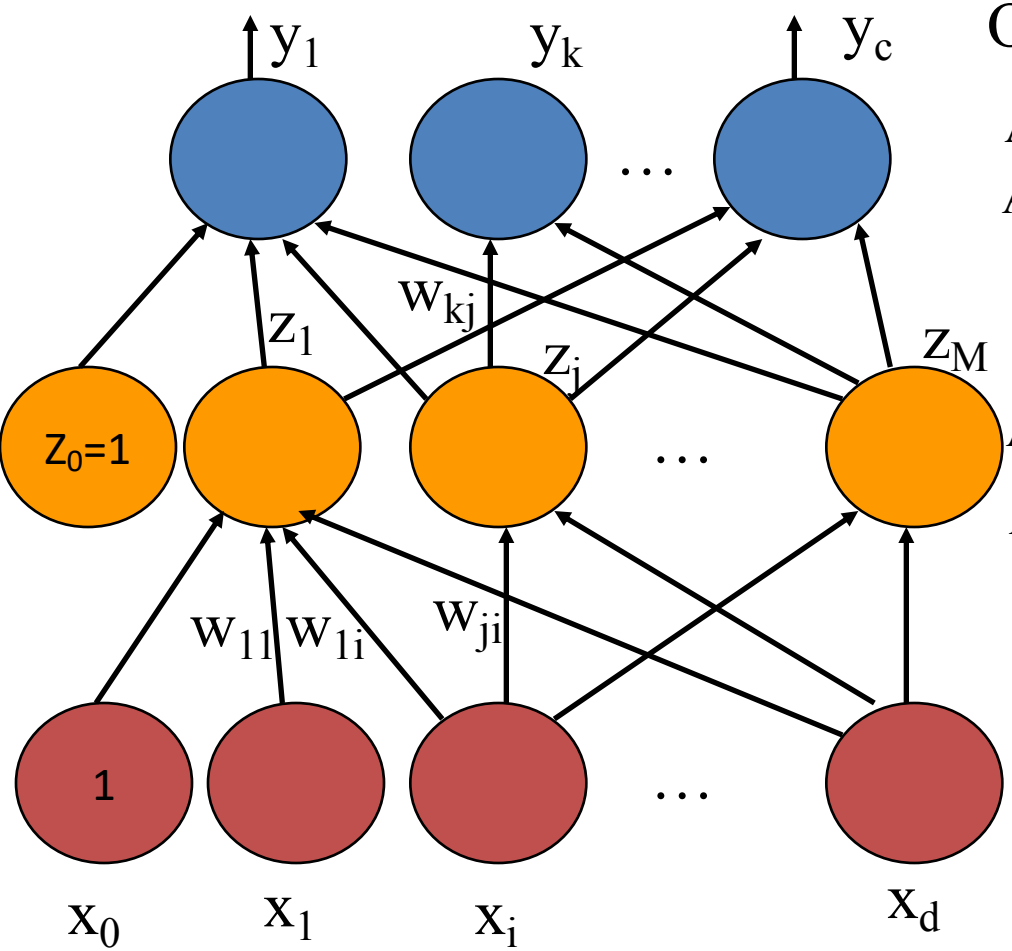








1980s – Neural Network Revolution

# Two-Layer Neural Network




Output

Activation function: $f$ (linear,sigmoid, softmax)

Activation of unit $a_k$:

$$\sum_{j=0}^{M} w_{kj} z_j$$

Activation function: $g$ (linear, tanh, sigmoid)

Activation of unit $a_j$:

$$\sum_{i=0}^{d} w_{ji} x_i$$

$$y_k = f\left(\sum_{j=0}^{M} w_{kj} \times g\left(\sum_{i=0}^{d} w_{ji} x_i\right)\right)$$

# Adjust Weights by Training

- How to adjust weights?
- Adjust weights using known examples (training data) $(x_1, x_2, x_3, \ldots, x_d, \mathrm{t})$.
- Try to adjust weights so that the difference between the output of the neural network $y$ and t (target) becomes smaller and smaller.
- Goal is to minimize Error (difference) as we did for one layer neural network

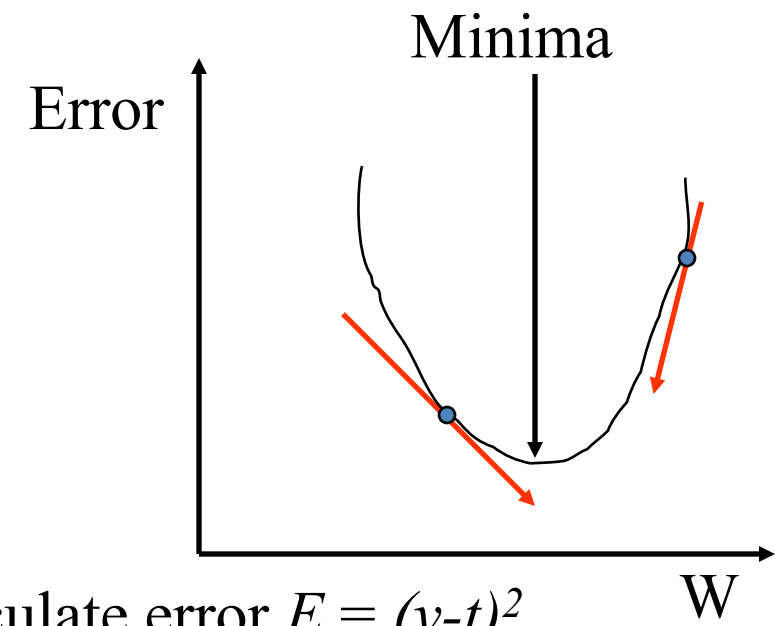
# Adjust Weights using Gradient Descent

Known:

Data: $(x_1, x_2, x_3, \ldots, x_n)$ target $t$.

Unknown weights $w$:

$w_{11}, w_{12}, \ldots\ldots$




Randomly initialize weights

Repeat

for each example, compute output $y$ calculate error $E = (y-t)^2$

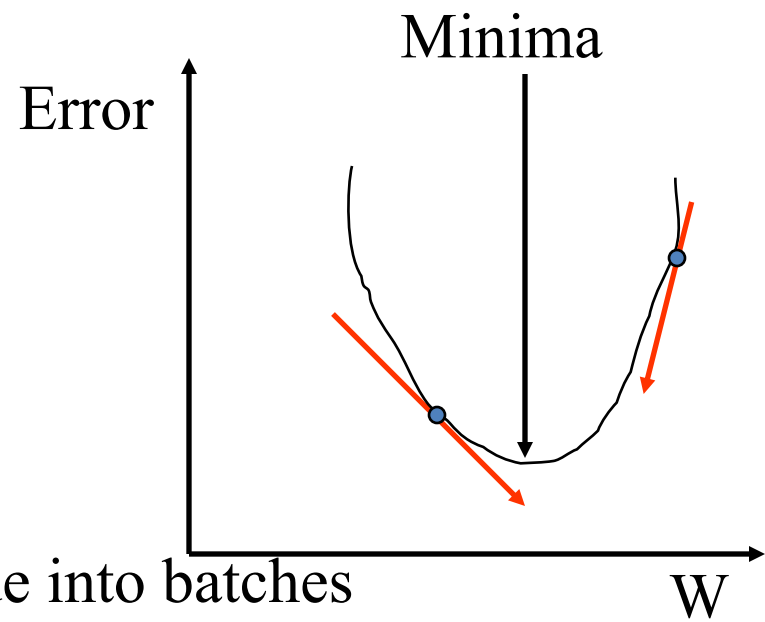
compute the derivative of $E$ over $w$: $dw = \frac{\partial E}{\partial w}$

$w_{\text{new}} = w_{\text{prev}} - \eta * dw$

Until error doesn't decrease or max num of iterations (epochs)

Note: $\eta$ is learning rate or step size.

# Stochastic Gradient Descent




Known:

Data: $(x_1, x_2, x_3, \ldots, x_n)$ target $t$.

Unknown weights $w$:

$w_{11}, w_{12}, \ldots\ldots$

Randomly initialize weights

Repeat

Randomize the order of examples, divide into batches

for each example in a batch, compute output $y$ and error $E = (y-t)^2$

compute the derivative of $E$ over $w$: $dw = \frac{\partial E}{\partial w}$

Add the derivatives of a batch together

$w_{\text{new}} = w_{\text{prev}} - \eta * dw$

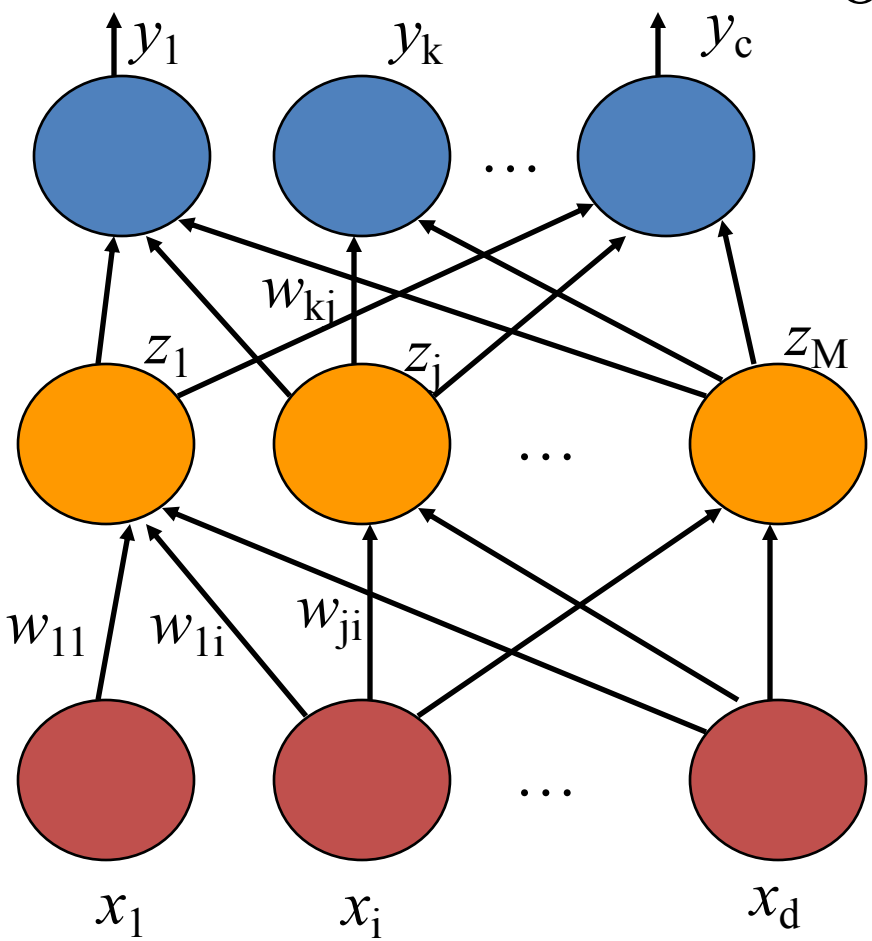
Until error doesn't decrease or max num of iterations (epochs)

Note: $\eta$ is learning rate or step size.

# Neural Network Learning: Two Processes

- Forward propagation: present an example (data) into neural network. Compute activation into units and output from units.
- Backward propagation: propagate error back from output layer to the input layer and compute derivatives (or gradients).

# Forward Propagation




Output $y_k$

Activation function: $f$ (linear,sigmoid, softmax)

Activation of unit $a_k$: $\sum_{j=1}^{M} w_{kj} z_j$

$z_j$

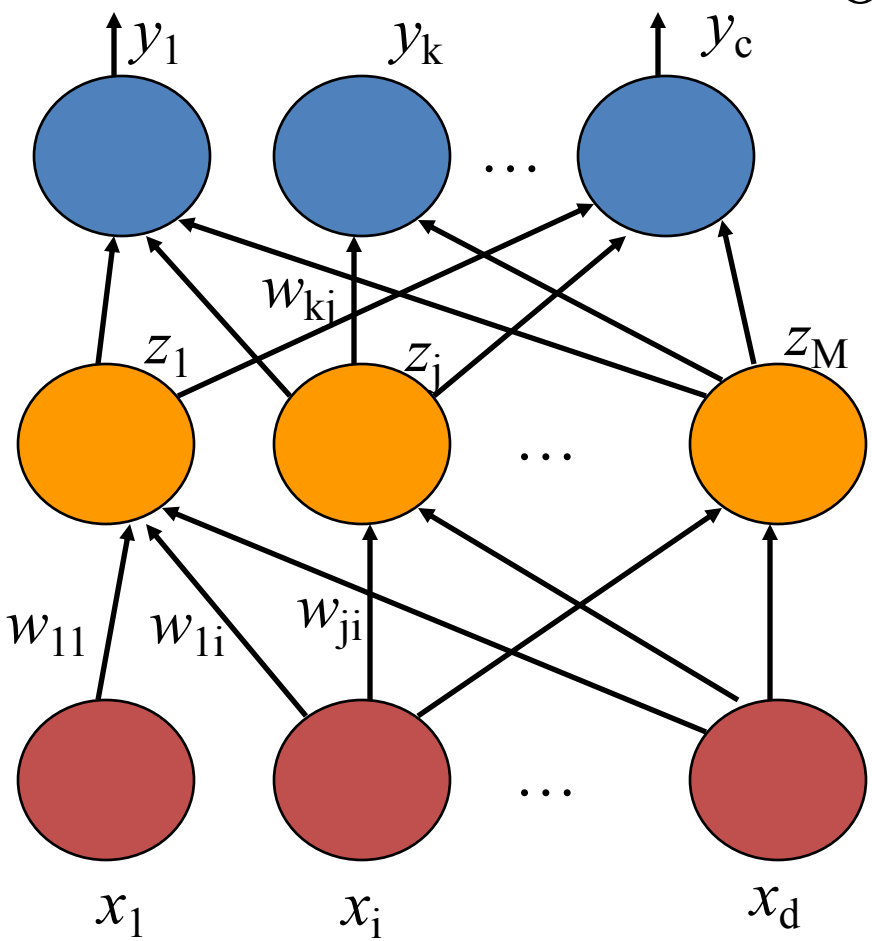
Activation function: $g$ (linear, tanh, sigmoid)

Activation of unit $a_j$:

$$\sum_{i=1}^{d} w_{ji} x_i$$

**Time complexity?**

# Forward Propagation




Output $y_k$

Activation function: $f$ (linear,sigmoid, softmax)

Activation of unit $a_k$: $\sum_{j=1}^{M} w_{kj} z_j$

$z_j$

Activation function: $g$ (linear, tanh, sigmoid)

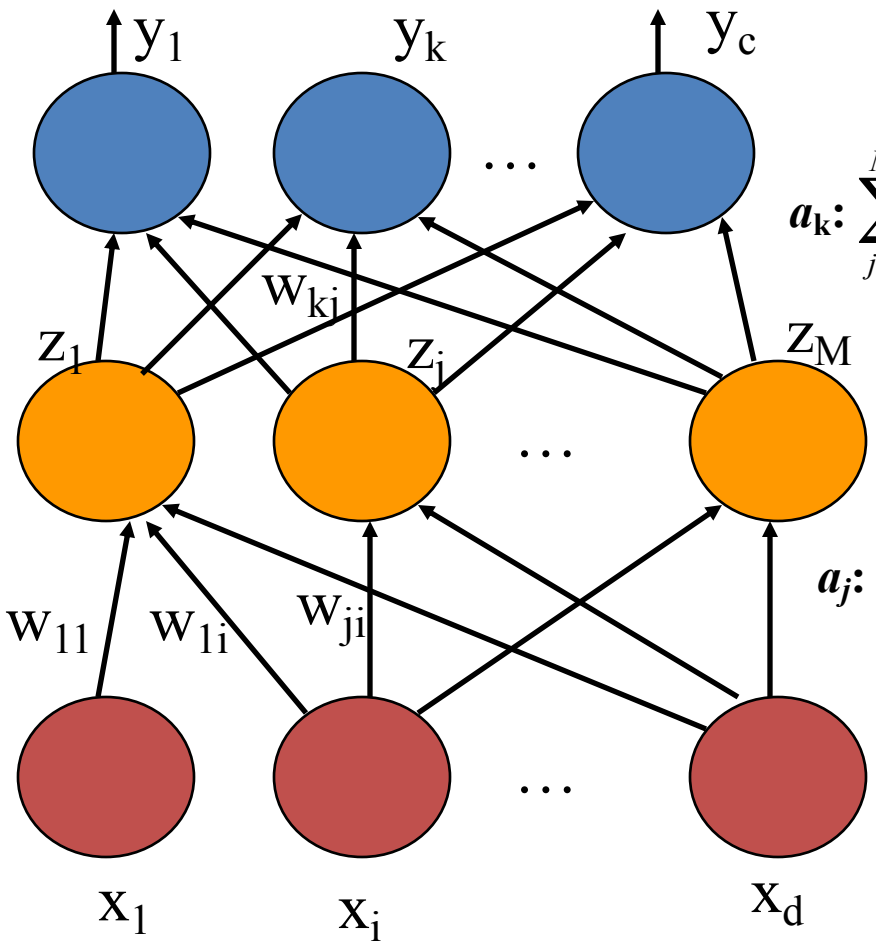
Activation of unit $a_j$:

$$\sum_{i=1}^{d} w_{ji} x_i$$

**Time complexity?**

**O(dM + MC) = O(W)**

# Backward Propagation




$$E = \frac{1}{2}\sum_{k=1}^{C}(y_k - t_k)^2$$

$$\frac{\partial E}{\partial y_k} = y_k - t_k$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = (y_k - t_k) f'(a_k) = \delta_k$$

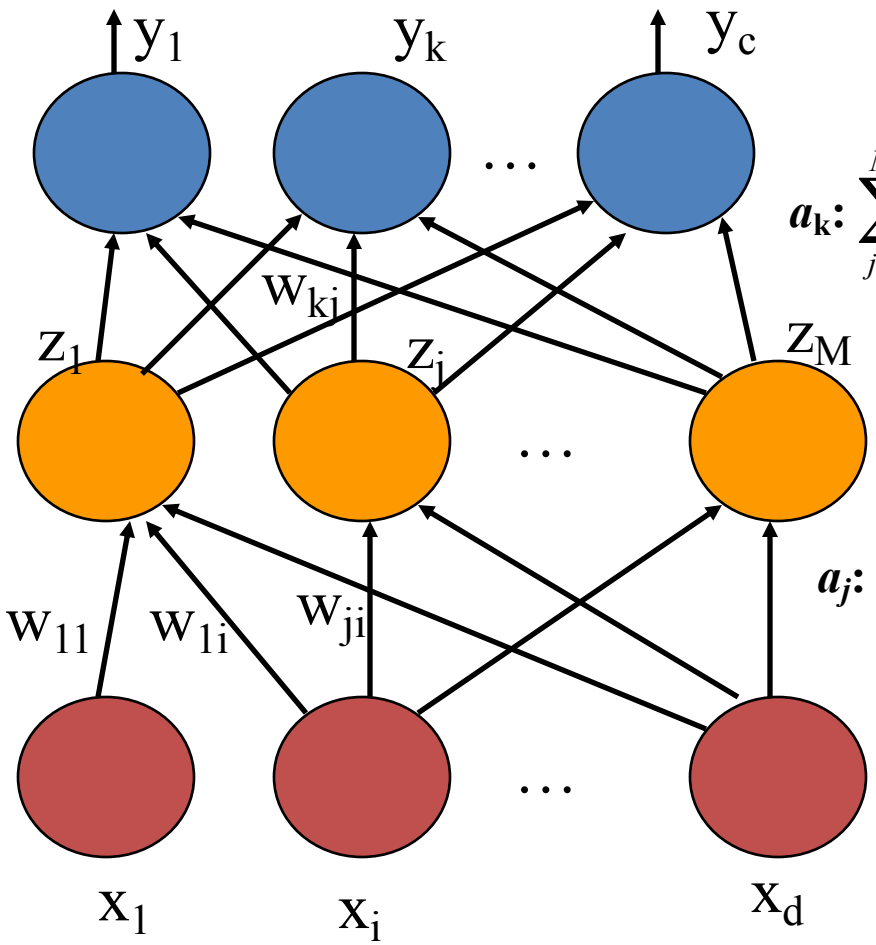
$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

$$\frac{\partial E}{\partial a_j} = \sum_{k=1}^{c}\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k}\frac{\partial a_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} = \sum_{k=1}^{C}\delta_k w_{kj} g'(a_j) = \delta_j$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial wji} = \delta_j x_i$$

**Time complexity?**

# Backward Propagation




$$E = \frac{1}{2}\sum_{k=1}^{C}(y_k - t_k)^2$$

$$\frac{\partial E}{\partial y_k} = y_k - t_k$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = (y_k - t_k) f'(a_k) = \delta_k$$

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

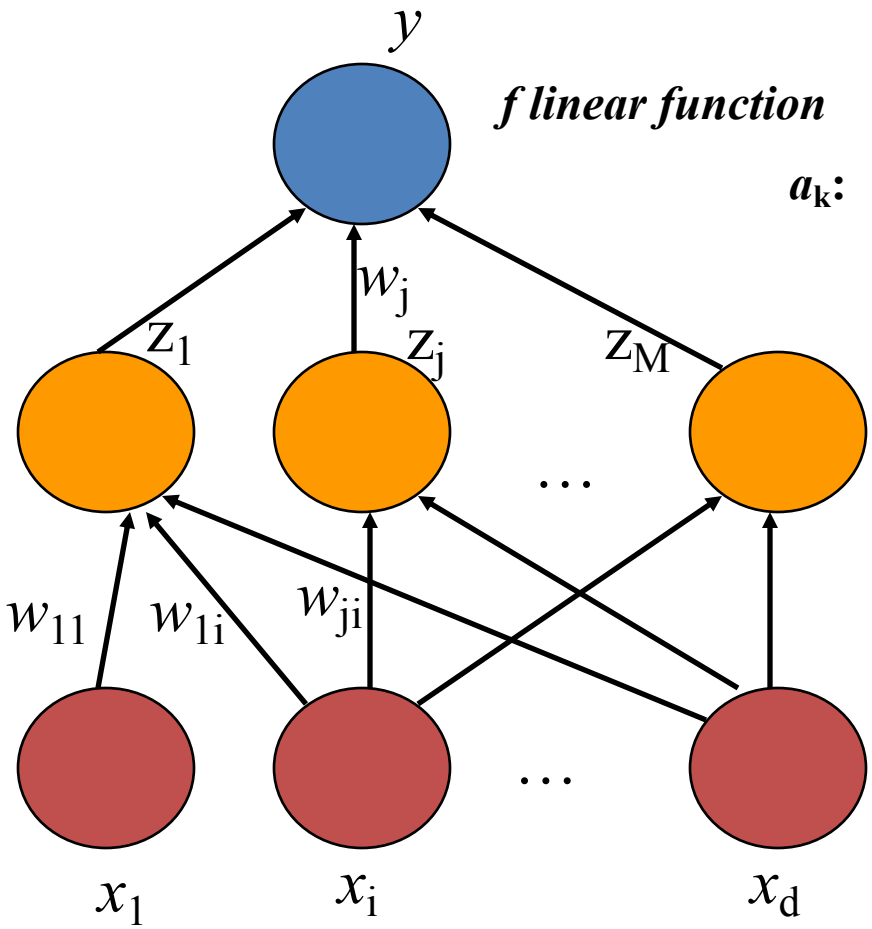
$$\frac{\partial E}{\partial a_j} = \sum_{k=1}^{c}\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k}\frac{\partial a_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} = \sum_{k=1}^{C}\delta_k w_{kj} g'(a_j) = \delta_j$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial wji} = \delta_j x_i$$

**Time complexity?**
**O(CM+Md) = O(W)**

If no back-propagation, time complexity is: (MdC+CM)

# Example




***f linear function***

$a_k$:

***g is sigmoid:***

$a_j$: $\sum_{i=1}^{M} w_{ji} x_i$

$$E = \frac{1}{2}(y-t)^2$$

$$\delta = \frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial a_k} = (y-t)$$

$$\frac{\partial E}{\partial w_j} = \delta z_j$$

$$\delta_j = \delta w_j g'(a_j) = (y-t) w_j z_j (1-z_j)$$

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i = (y-t) w_j z_j (1-z_j) x_i$$

# TensorFlow Demo of Simple Neural Networks

- Data Sets: Iris flower classification data
- Google's TensorFlow installation: [https://www.tensorflow.org/](https://www.tensorflow.org/)



From left to right, *Iris setosa* (by Radomil, CC BY-SA 3.0), *Iris versicolor* (by Dlanglois, CC BY-SA 3.0), and *Iris virginica* (by Frank Mayfield, CC BY-SA 2.0).

| Sepal Length | Sepal Width | Petal Length | Petal Width | Species |
|---|---|---|---|---|
| 5.1 | 3.5 | 1.4 | 0.2 | 0 |
| 4.9 | 3.0 | 1.4 | 0.2 | 0 |
| 4.7 | 3.2 | 1.3 | 0.2 | 0 |
| ... | ... | ... | ... | ... |
| 7.0 | 3.2 | 4.7 | 1.4 | 1 |
| 6.4 | 3.2 | 4.5 | 1.5 | 1 |
| 6.9 | 3.1 | 4.9 | 1.5 | 1 |
| ... | ... | ... | ... | ... |
| 6.5 | 3.0 | 5.2 | 2.0 | 2 |
| 6.2 | 3.4 | 5.4 | 2.3 | 2 |
| 5.9 | 3.0 | 5.1 | 1.8 | 2 |

For this tutorial, the Iris data has been randomized and split into two separate CSVs:

- A training set of 120 samples (iris_training.csv)
- A test set of 30 samples (iris_test.csv).

```python
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import urllib

import numpy as np
import tensorflow as tf

# Data sets
IRIS_TRAINING = "iris_training.csv"
IRIS_TRAINING_URL = "http://download.tensorflow.org/data/iris_training.csv"

IRIS_TEST = "iris_test.csv"
IRIS_TEST_URL = "http://download.tensorflow.org/data/iris_test.csv"

def main():
  # If the training and test sets aren't stored locally, download them.
  if not os.path.exists(IRIS_TRAINING):
    raw = urllib.urlopen(IRIS_TRAINING_URL).read()
    with open(IRIS_TRAINING, "w") as f:
      f.write(raw)

  if not os.path.exists(IRIS_TEST):
    raw = urllib.urlopen(IRIS_TEST_URL).read()
    with open(IRIS_TEST, "w") as f:
      f.write(raw)

  # Load datasets.
  training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
      filename=IRIS_TRAINING,
      target_dtype=np.int,
      features_dtype=np.float32)
  test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
      filename=IRIS_TEST,
      target_dtype=np.int,
      features_dtype=np.float32)
```

```python
# Specify that all features have real-value data
feature_columns = [tf.feature_column.numeric_column("x", shape=[4])]

# Build 3 layer DNN with 10, 20, 10 units respectively.
classifier = tf.estimator.DNNClassifier(feature_columns=feature_columns,
                                        hidden_units=[10, 20, 10],
                                        n_classes=3,
                                        model_dir="/tmp/iris_model")
# Define the training inputs
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": np.array(training_set.data)},
    y=np.array(training_set.target),
    num_epochs=None,
    shuffle=True)

# Train model.
classifier.train(input_fn=train_input_fn, steps=2000)

# Define the test inputs
test_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": np.array(test_set.data)},
    y=np.array(test_set.target),
    num_epochs=1,
    shuffle=False)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(input_fn=test_input_fn)["accuracy"]

print("\nTest Accuracy: {0:f}\n".format(accuracy_score))

# Classify two new flower samples.
new_samples = np.array(
    [[6.4, 3.2, 4.5, 1.5],
     [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)
predict_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": new_samples},
    num_epochs=1,
    shuffle=False)
```

```
  predictions = list(classifier.predict(input_fn=predict_input_fn))
  predicted_classes = [p["classes"] for p in predictions]

  print(
      "New Samples, Class Predictions:    {}\n"
      .format(predicted_classes))

if __name__ == "__main__":
    main()
```

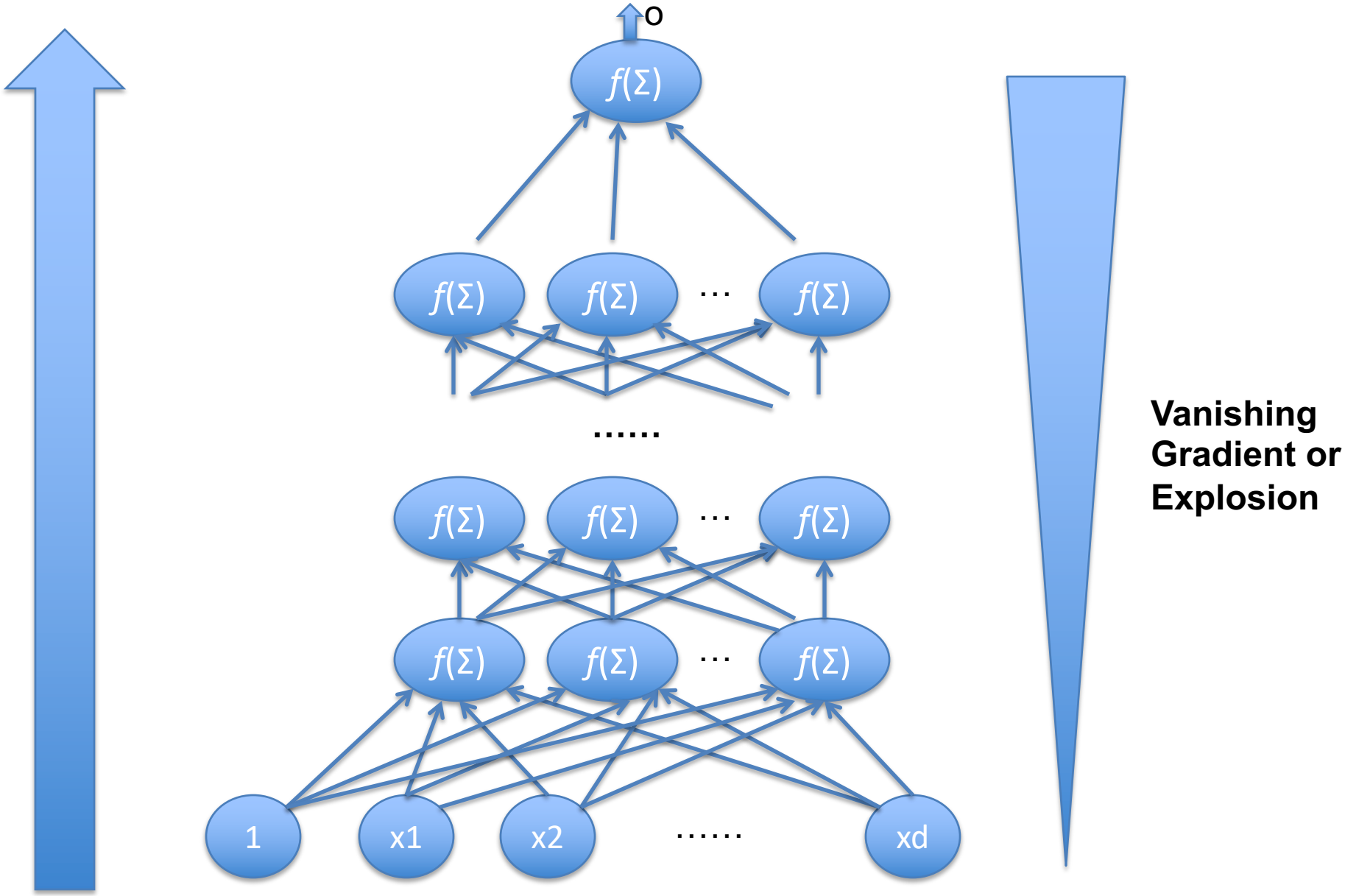
# Keras

- **Keras** is an open-source neural-network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, Theano, or PlaidML.[1][2] Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. It was developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System),[3] and its primary author and maintainer is François Chollet, a Google engineer. Chollet also is the author of the XCeption deep neural network model[4].
- In 2017, Google's TensorFlow team decided to support Keras in TensorFlow's core library.[5]Chollet explained that Keras was conceived to be an interface rather than a standalone machine learning framework. It offers a higher-level, more intuitive set of abstractions that make it easy to develop deep learning models regardless of the computational backend used.[6] Microsoft added a CNTK backend to Keras as well, available as of CNTK v2.0.[7][8]

# An Example with TensorFlow in Keras for MNIST Classification

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

*https://www.tensorflow.org/overview/*


o
f(Σ)
f(Σ)
f(Σ)
…
f(Σ)
……
f(Σ)
f(Σ)
…
f(Σ)
f(Σ)
f(Σ)
…
f(Σ)
1
x1
x2
……
xd
Vanishing Gradient or Explosion

# Neural Network's Winter in 1990s

- A standard three-layer neural network is a universal approximator
- Hard to train multi-layer neural networks
- Get different models from different training (local minimal)

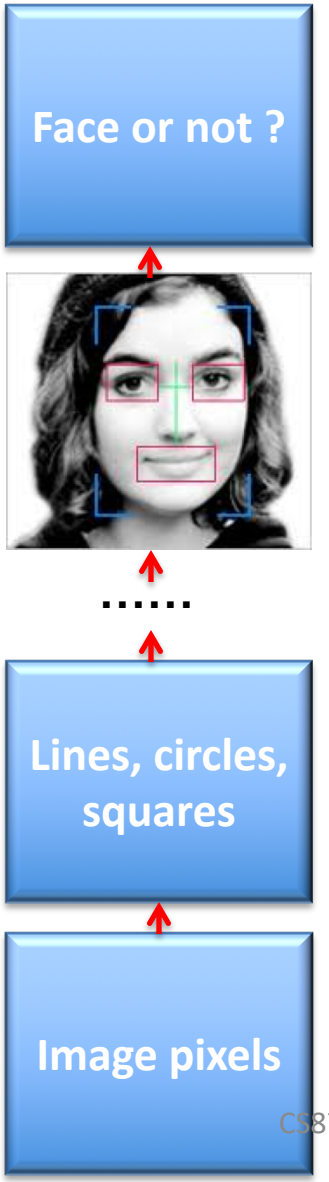
$$y_k = f(\sum_{j=0}^{M} w_{kj} \times g(\sum_{i=0}^{d} w_{ji} x_i))$$

# How to Construct Deep Networks?

**G. Hinton**

**2000s**

# Learning by Composition? – A Face Recognition Analogy




**Brain Learning**

# Breakthrough

***Deep Learning: machine learning algorithms based on learning multiple levels of representation / abstraction***

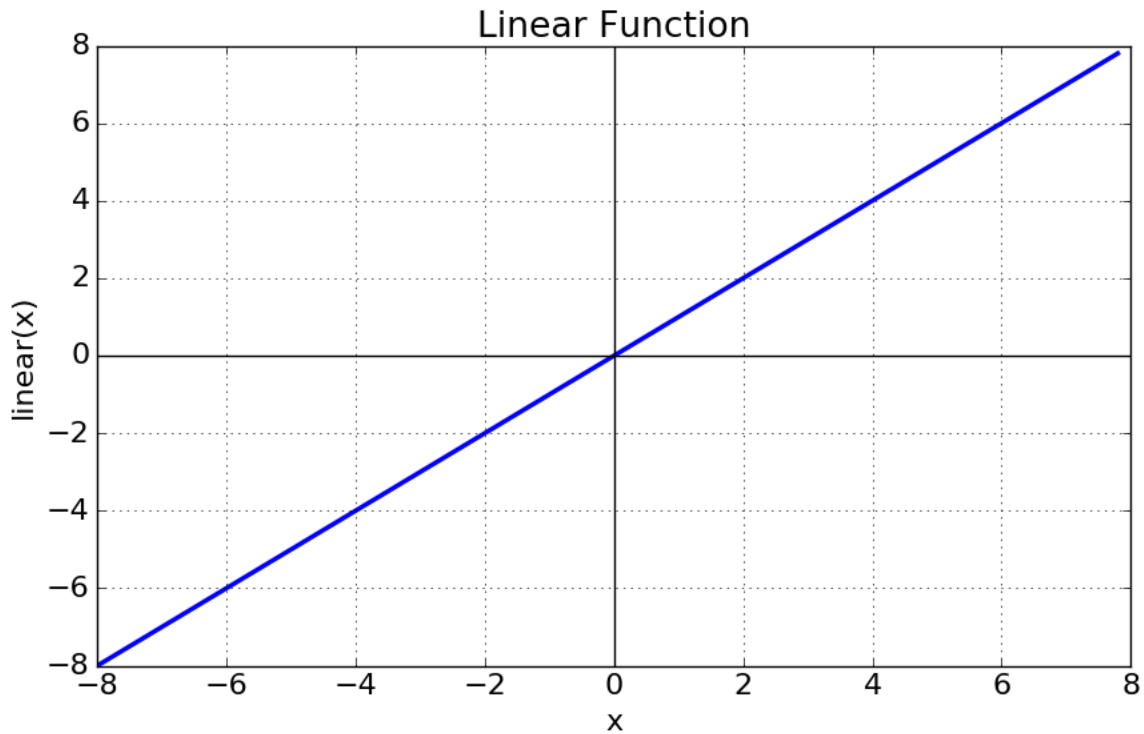
Amazing improvements in error rate in object recognition, object detection, speech recognition, and more recently, in natural language processing / understanding

# Machine Learning for Artificial Intelligence

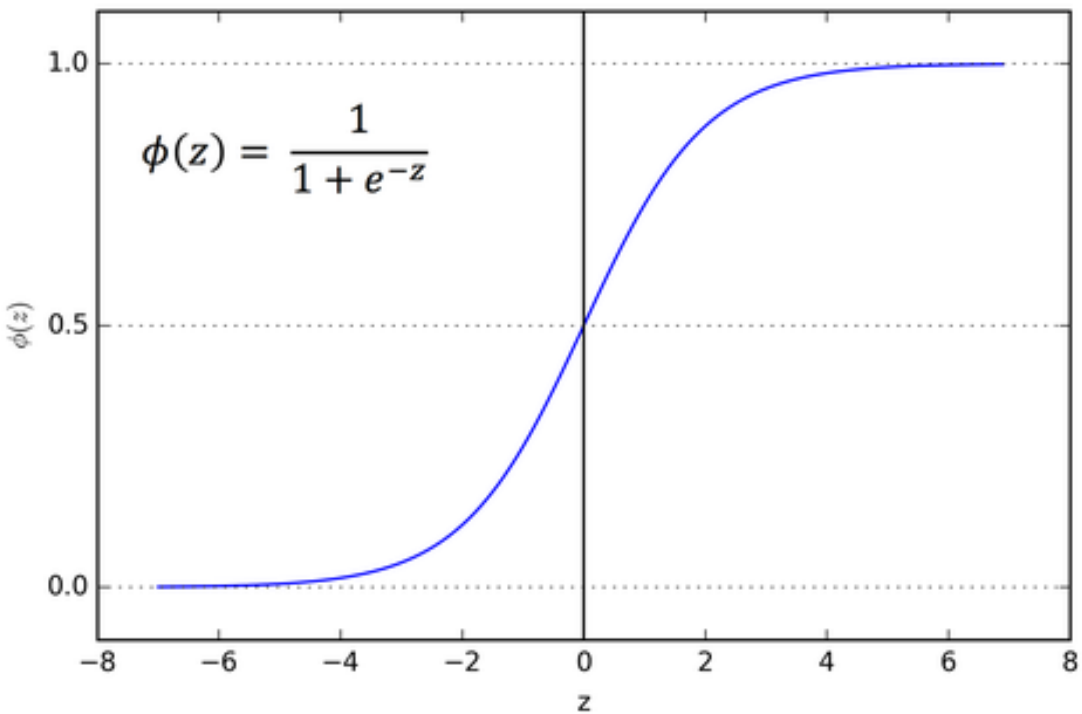
**Four key ingredients for ML towards AI**

- **Lots & lots of data**
- **Very flexible models**
- **Enough computing power**
- **Technical improvement (ReLu function, ResNet, semi-supervised learning)**
- **Powerful priors that can defeat the curse of dimensionality**

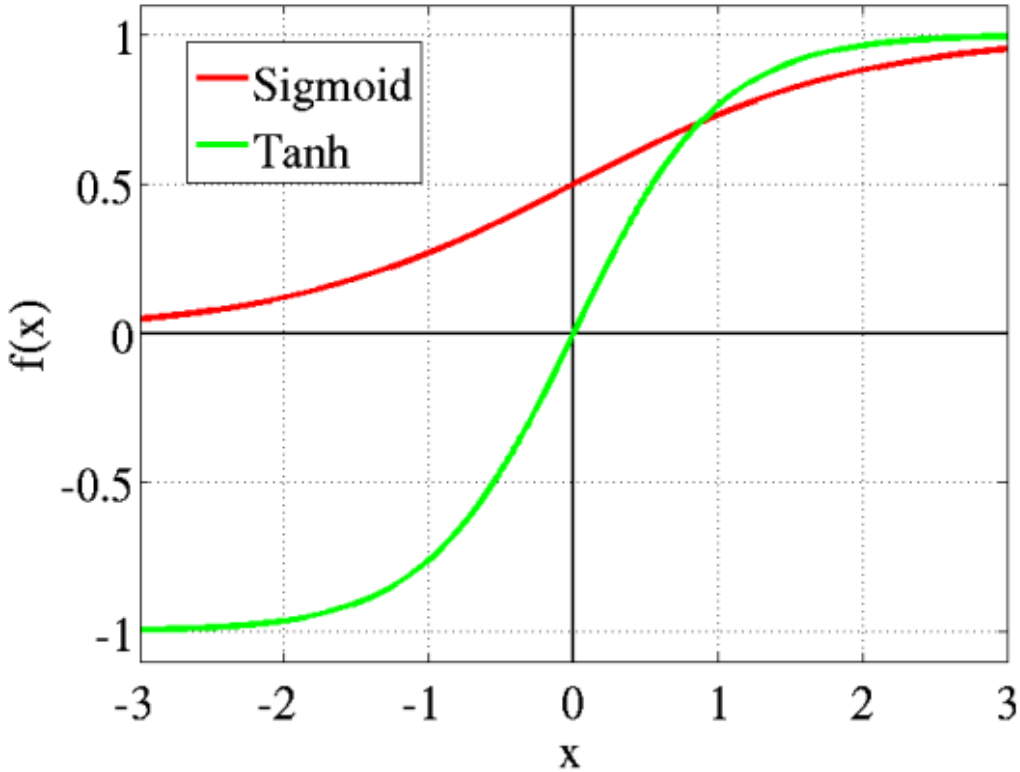
# Linear Activation Function



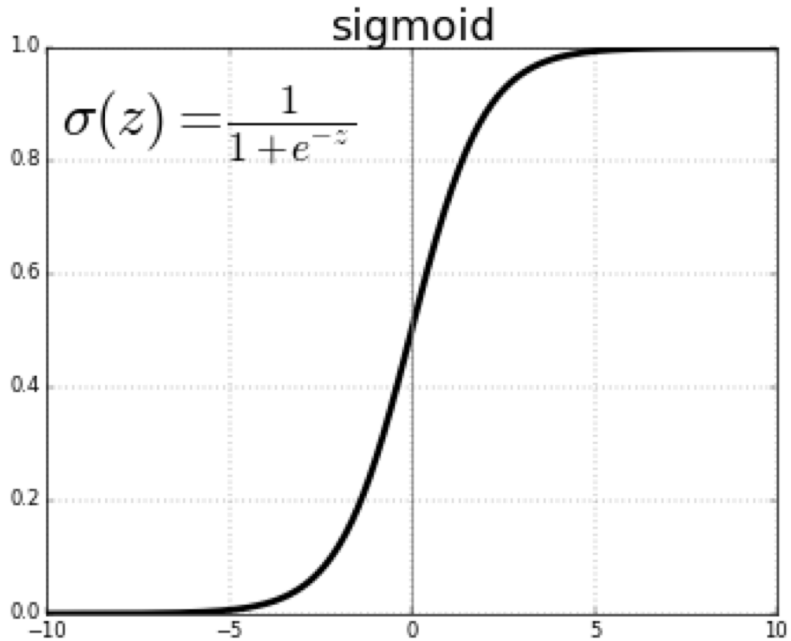
# Sigmoid Function

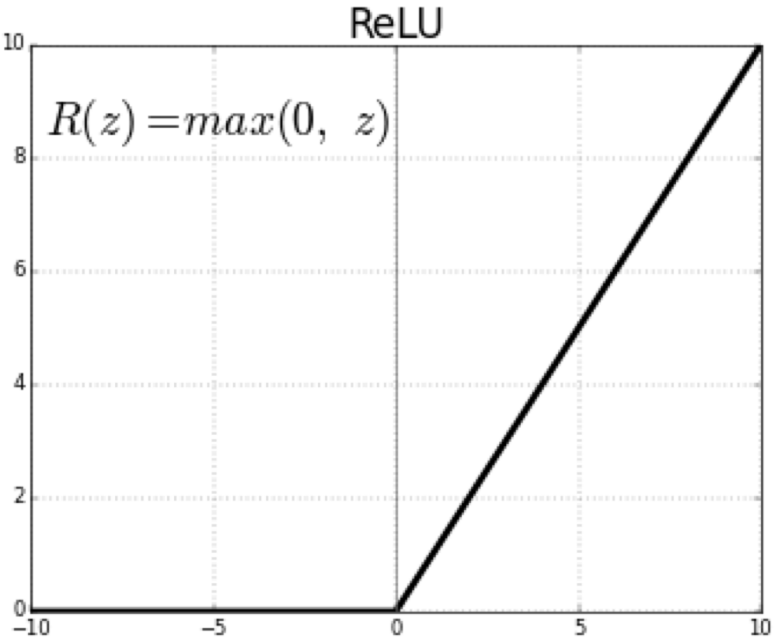


# Hyperbolic Tangent Function (TanH)

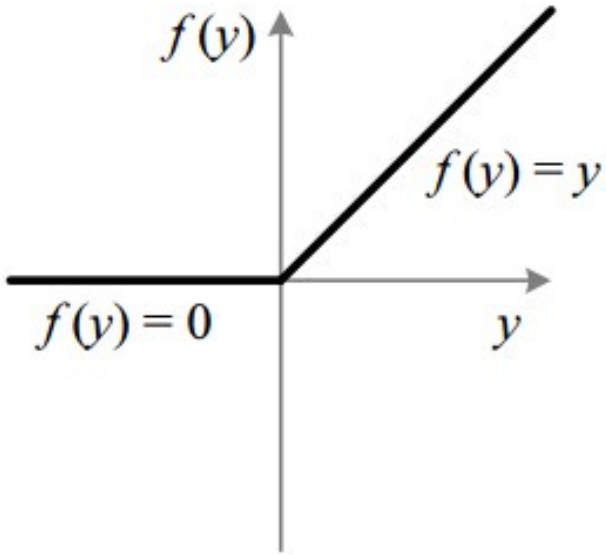


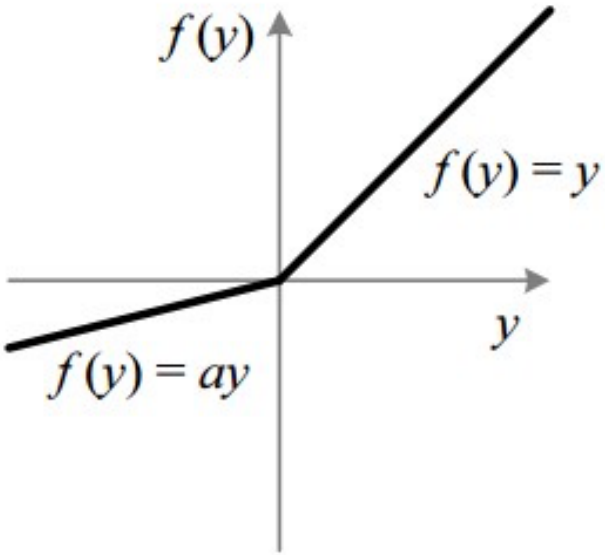
# Rectified Linear Unit (ReLU)



# Leaky ReLU



# Bypassing the curse of dimensionality

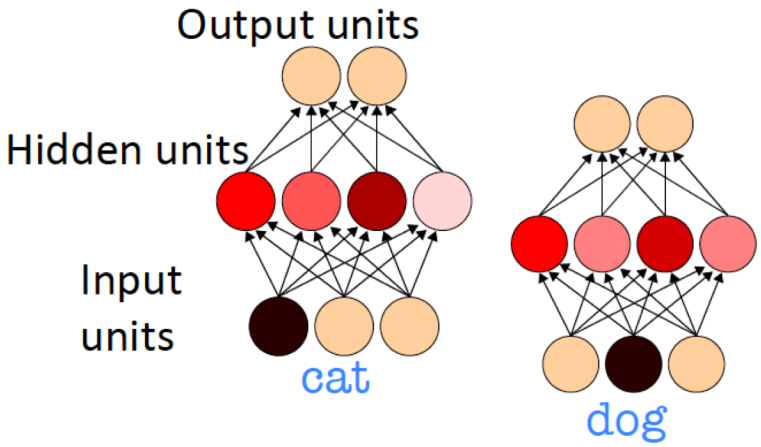
- We need to build compositionality into our ML models just as human languages exploit compositionality
- Exploiting compositionality gives an exponential gain in representational power: (1) distributed representations/embeddings (feature learning); (2) deep architecture ( multi-levels of feature learning)
- Additional prior: compositionality is useful to describe the world around us efficiently

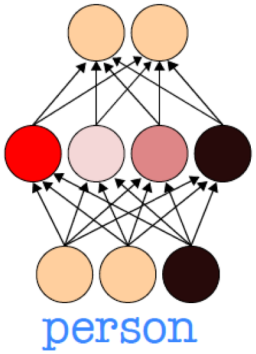
# Classical Symbolic AI vs Learning Distributed Representations

- Two symbols are equally far from each other
- Concepts are not represented by symbols in our brain, but by patterns of activation

  *(Connectionism, 1980's)*

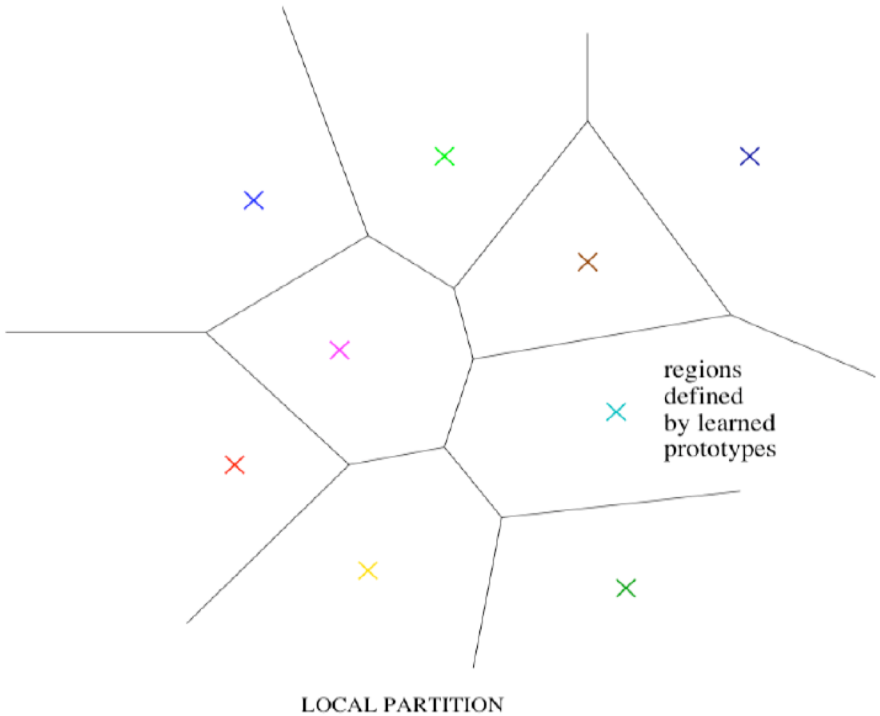
Geoffrey Hinton

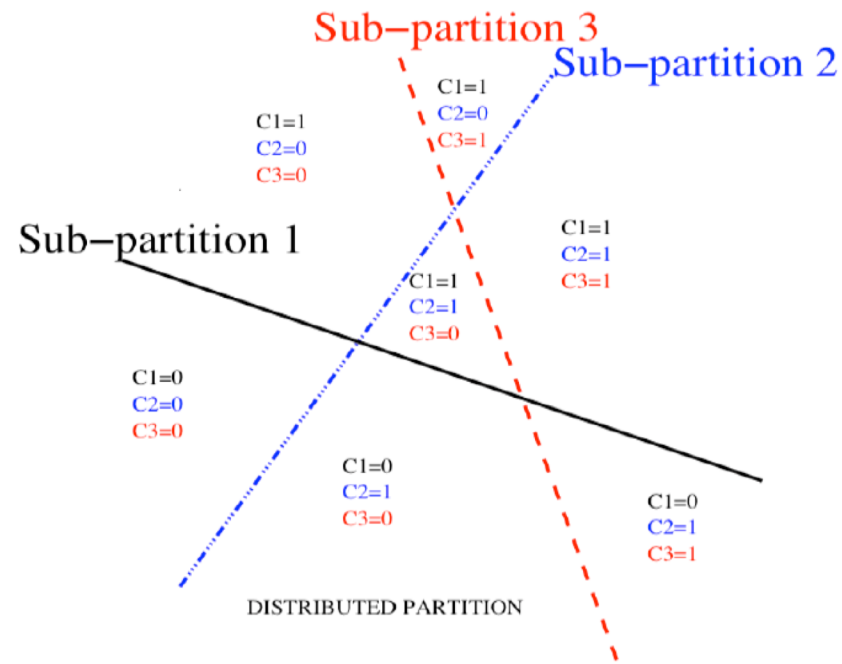



David Rumelhart

# Exponential advantage of distributed representations




Learning a **set of parametric features** that are not mutually exclusive can be exponentially more statistically efficient than having nearest-neighbor-like or clustering-like models

# Each feature can be discovered without the need for seeing the exponentially large number of configurations of the other features

- Consider a network whose hidden units discover the following features:
  - Person wears glasses
  - Person is female
  - Person is a child
  - Etc.

If each of $n$ feature requires $O(k)$ parameters, need $O(nk)$ examples

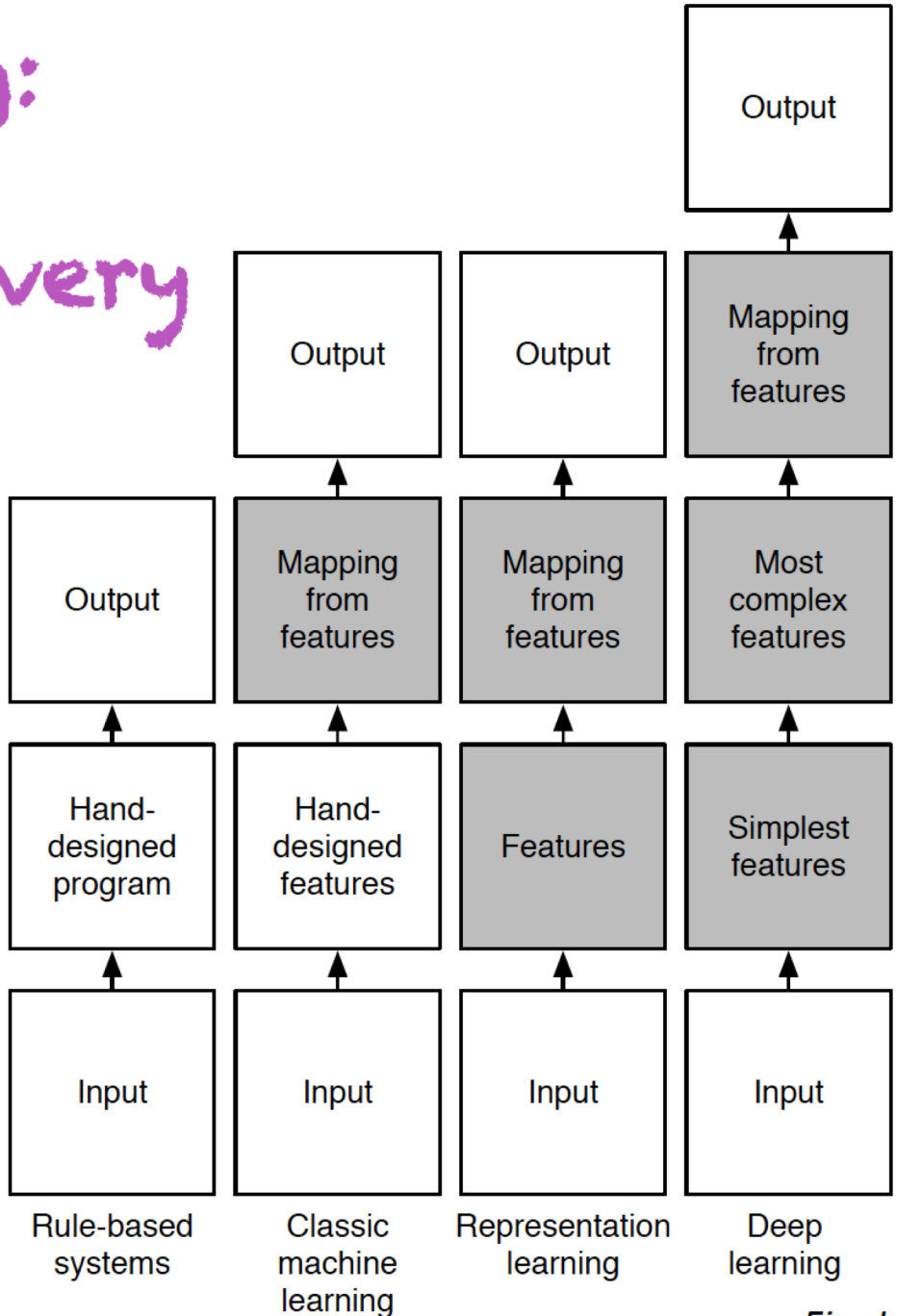
Non-parametric methods would require $O(n^d)$ examples

# Exponential advantage of distributed representations

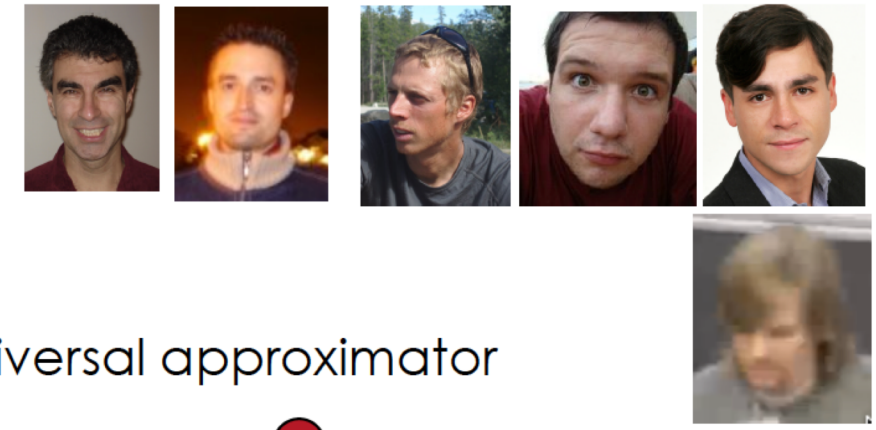
Prop. 2 of ***Pascanu, Montufar & Bengio ICLR'2014***: number of pieces distinguished by 1-hidden-layer rectifier net with *n* units and *d* inputs (i.e. *O(nd)* parameters) is

$$\sum_{j=0}^{d} \binom{n}{j} = O(n^d)$$

# Deep Learning: Automating Feature Discovery



# Exponential advantage of depth



Theoretical arguments:

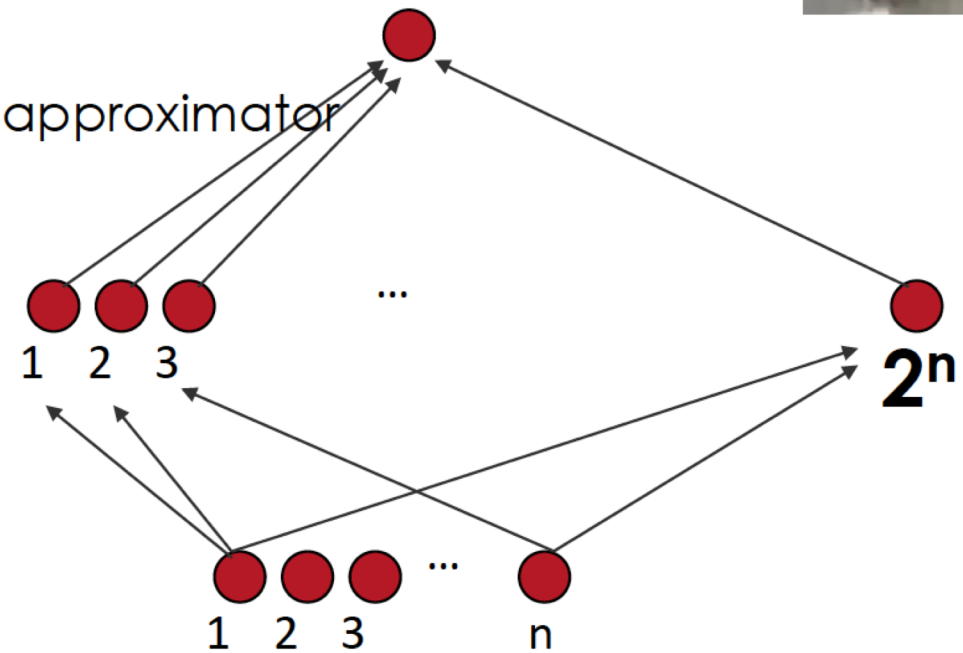
2 layers of
- Logic gates
- Formal neurons
- RBF units

= universal approximator

RBMs & auto-encoders = universal approximator

**Theorems on advantage of depth:**
(Hastad et al 86 & 91, Bengio et al 2007, Bengio & Delalleau 2011, Martens et al 2013, Pascanu et al 2014, Montufar et al **NIPS 2014**)

Some functions compactly represented with k layers may require exponential size with 2 layers



# Why does it work? No Free Lunch

- It only works because we are making some assumptions about the data generating distribution
- Worse-case distributions still require exponential data
- But the world has structure and we can get an exponential gain by exploiting some of it

# Exponential advantage of depth

- Expressiveness of deep networks with piecewise linear activation functions: exponential advantage for depth *(Montufar et al, NIPS 2014)*
- Number of pieces distinguished for a network with depth $L$ and $n_i$ units per layer is at least

$$\left(\prod_{i=1}^{L-1}\left\lfloor\frac{n_i}{n_0}\right\rfloor^{n_0}\right)\sum_{j=0}^{n_0}\binom{n_L}{j}$$

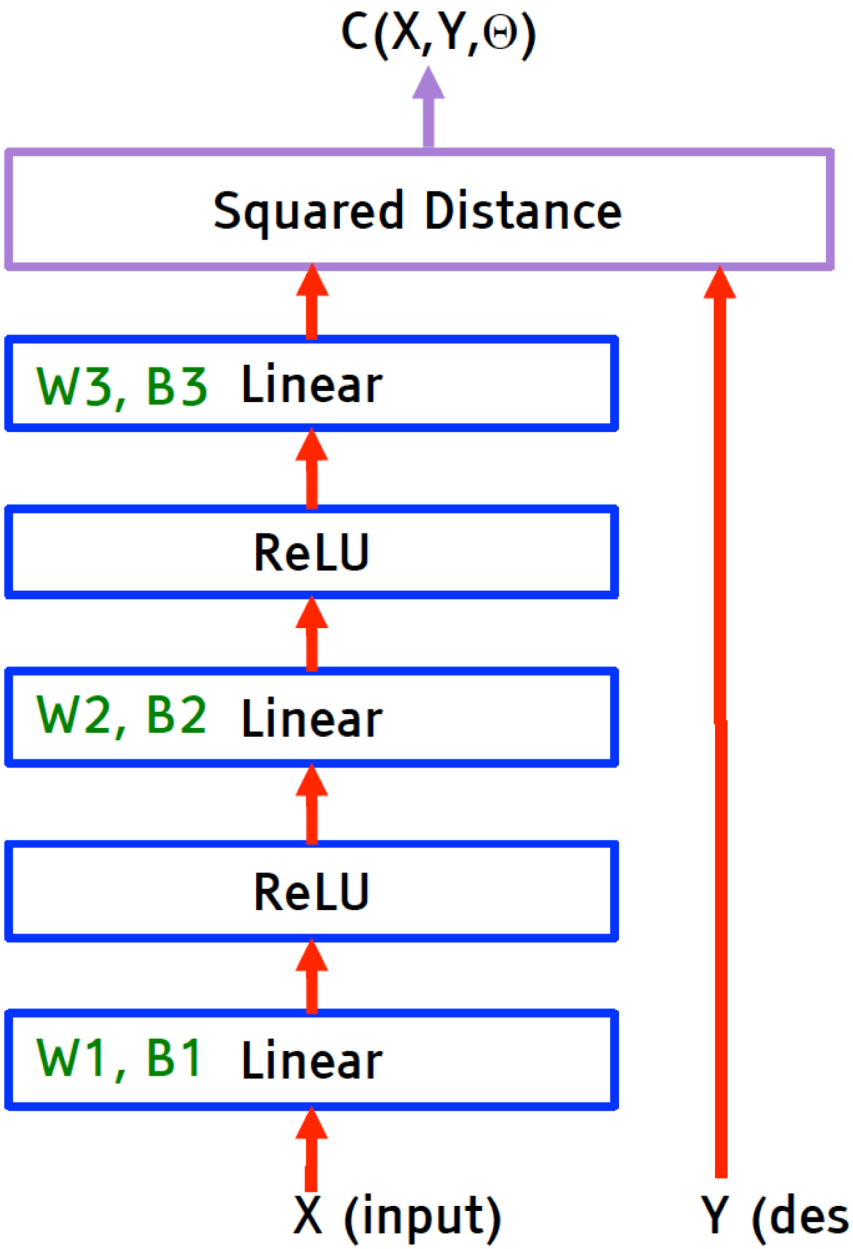
or, if hidden layers have width $n$ and input has size $n_0$

$$\Omega\left(\left(n/n_0\right)^{(L-1)n_0}\, n^{n_0}\right)$$

# Construct Deep Networks

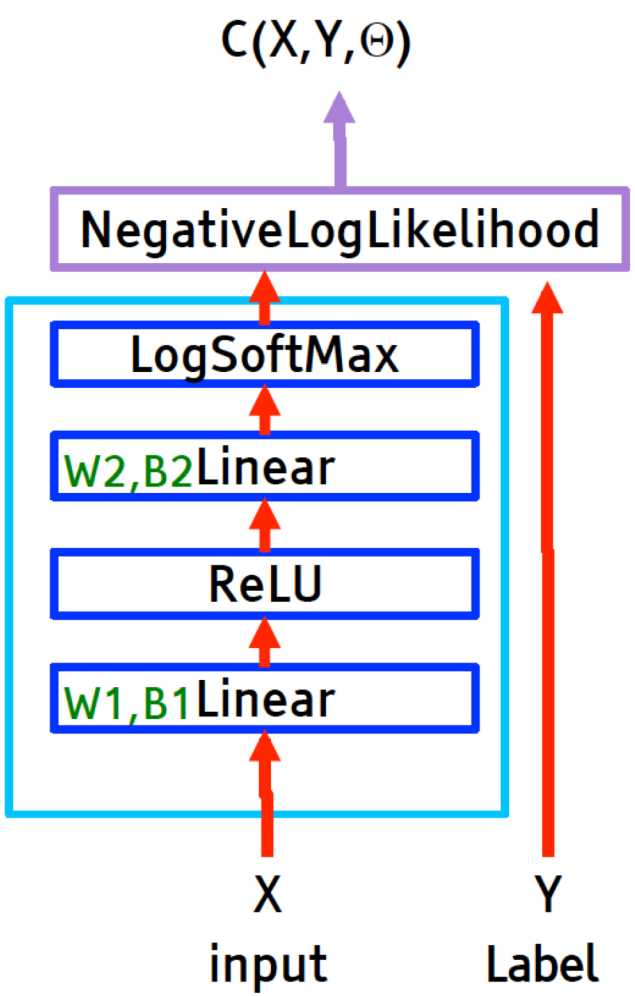
Backprop
(modular approach)

# Typical Deep Multilayer Neural Net



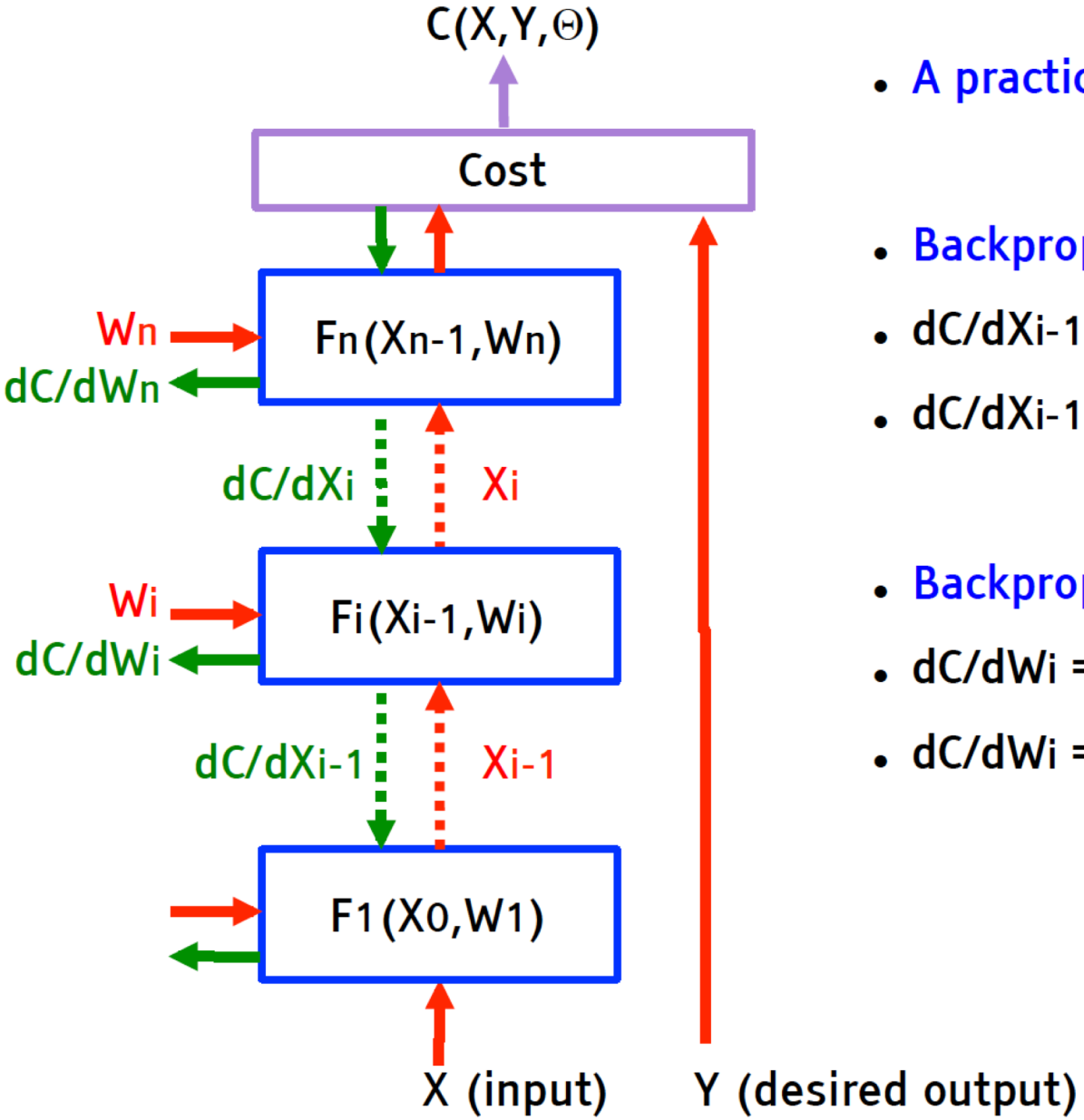

- Complex learning machines can be built by assembling modules into networks
- Linear Module
  - Out = W.In+B
- ReLU Module (Rectified Linear Unit)
  - $Out_i = 0$ if $In_i<0$
  - $Out_i = In_i$ otherwise
- Cost Module: Squared Distance
  - $C = ||In1 - In2||^2$
- Objective Function
  - $L(\Theta)=1/p \sum_k C(X^k,Y^k,\Theta)$
  - $\Theta = (W1,B1,W2,B2,W3,B3)$

- All major deep learning frameworks use modules (inspired by SN/Lush, 1991)
  - Torch7, Theano, TensorFlow....




```
-- sizes
ninput = 28*28  -- e.g. for MNIST
nhidden1 = 1000
noutput = 10

-- network module
net = nn.Sequential()
net:add(nn.Linear(ninput, nhidden))
net:add(nn.Threshold())
net:add(nn.Linear(nhidden, noutput))
net:add(nn.LogSoftMax()))

-- cost module
cost = nn.ClassNLLCriterion()

-- get a training sample
input = trainingset.data[k]
target = trainingset.labels[k]

-- run through the model
output = net:forward(input)
c = cost:forward(output, target)
```

# Computing Gradients by Back-Propagation



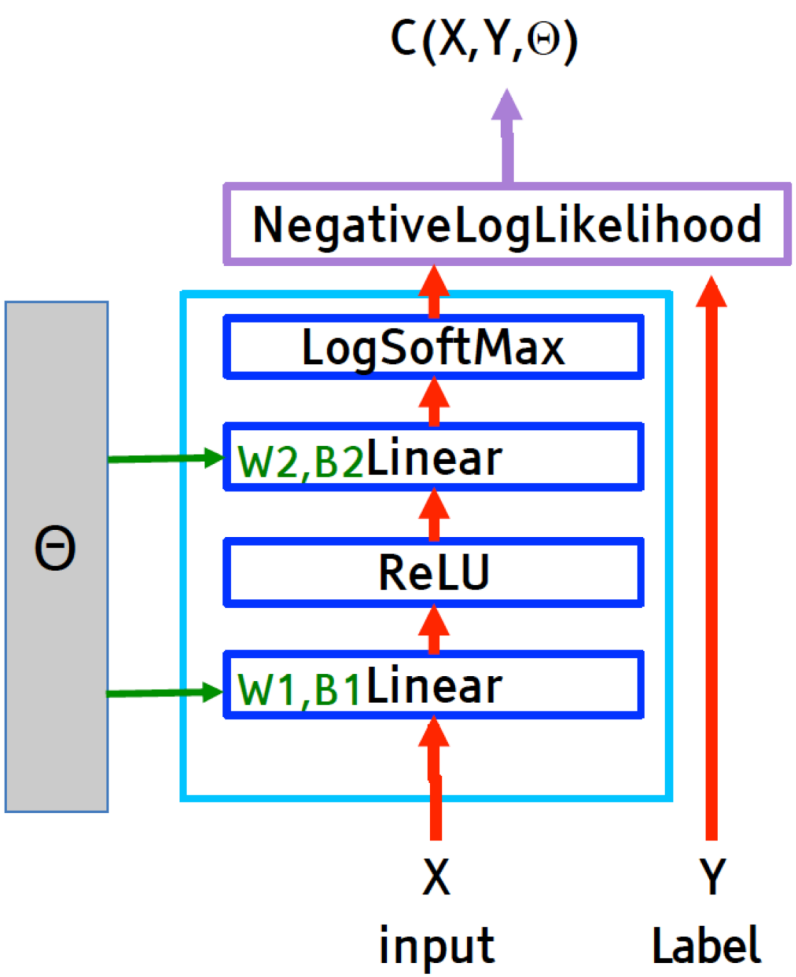

- A practical Application of Chain Rule

- Backprop for the state gradients:
- dC/dXi-1 = dC/dXi . dXi/dXi-1
- dC/dXi-1 = dC/dXi . dFi(Xi-1,Wi)/dXi-1

- Backprop for the weight gradients:
- dC/dWi = dC/dXi . dXi/dWi
- dC/dWi = dC/dXi . dFi(Xi-1,Wi)/dWi

# Running Backprop

- Torch7 example
- Gradtheta contains the gradient




```
-- network module
net = nn.Sequential()
net:add(nn.Linear(ninput, nhidden))
net:add(nn.Threshold())
net:add(nn.Linear(nhidden, noutput))
net:add(nn.LogSoftMax()))

-- cost module
cost = nn.ClassNLLCriterion()

-- gather the parameters in a vector
theta, gradtheta = net:getParameters()

-- get a training sample
input = trainingset.data[k]
target = trainingset.labels[k]

-- run through the model
output = net:forward(input)
c = cost:forward(output, target)

-- run backprop
gradtheta:zero()
gradoutput = cost:backward(output, target)
net:backward(input, gradoutput)
```

# Modular Classes

**Linear**
- $Y = W.X$ ; $dC/dX = W^T . dC/dY$ ; $dC/dW = dC/dY . dY/dx$

**ReLU**
- $y = ReLU(x)$ ; if $(x<0)$ $dC/dx = 0$ else $dC/dx = dC/dy$

**Duplicate**
- $Y1 = X, Y2 = X$ ; $dC/dX = dC/dY1 + dC/dY2$

**Add**
- $Y = X1 + X2$ ; $dC/dX1 = dC/dY$ ; $dC/dX2 = dC/dY$

**Max**
- $y = max(x1,x2)$ ; if $(x1>x2)$ $dC/dx1 = dC/dy$ else $dC/dx1=0$

**LogSoftMax**
- $Yi = Xi - \log\left[\sum_j \exp(Xj)\right]$ ; .....

# Modular Classes

- Many more basic module classes
- Cost functions:
  - Squared error
  - Hinge loss
  - Ranking loss
- Non-linearities and operators
  - ReLU, “leaky” ReLU, abs,....
  - Tanh, logistic
  - Just about any simple function (log, exp, add, mul,....)
- Specialized modules
  - Multiple convolutions (1D, 2D, 3D)
  - Pooling/subsampling: max, average, Lp, log(sum(exp())), maxout
  - Long Short-Term Memory, attention, 3-way multiplicative interactions.
  - Switches
  - Normalizations: batch norm, contrast norm, feature norm...
  - inception **(replace linear filter with non-linear filter in convolutional neural network)**

# Hinge loss

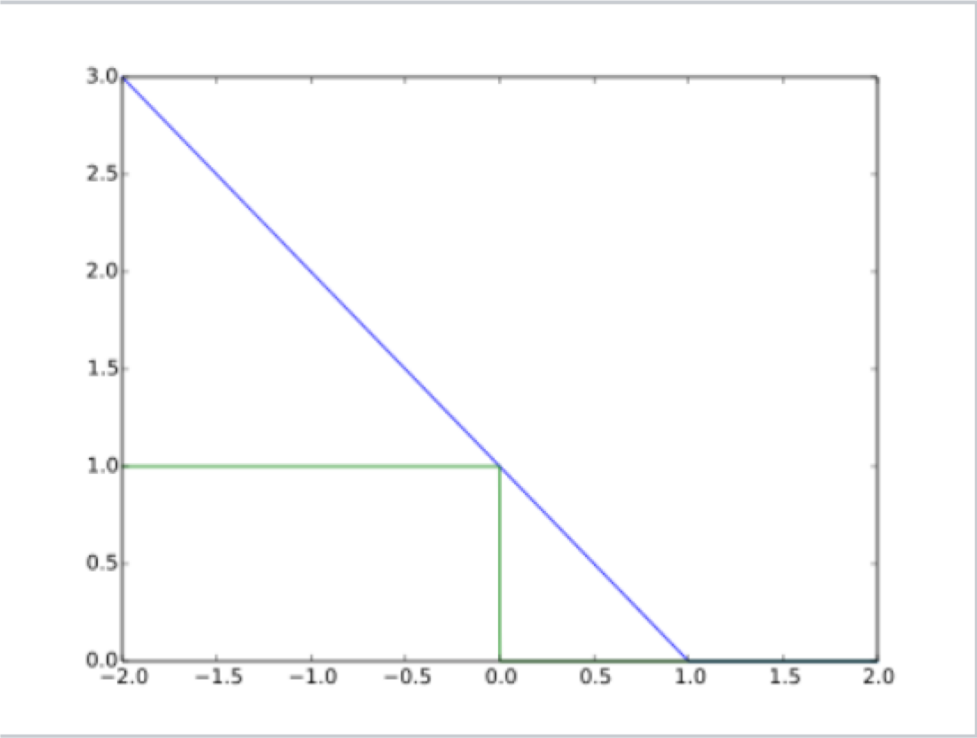
From Wikipedia, the free encyclopedia

In machine learning, the **hinge loss** is a loss function used for training classifiers. The hinge loss is used for "maximum-margin" classification, most notably for support vector machines (SVMs).[1] For an intended output $t = \pm 1$ and a classifier score $y$, the hinge loss of the prediction $y$ is defined as

$$\ell(y) = \max(0, 1 - t \cdot y)$$

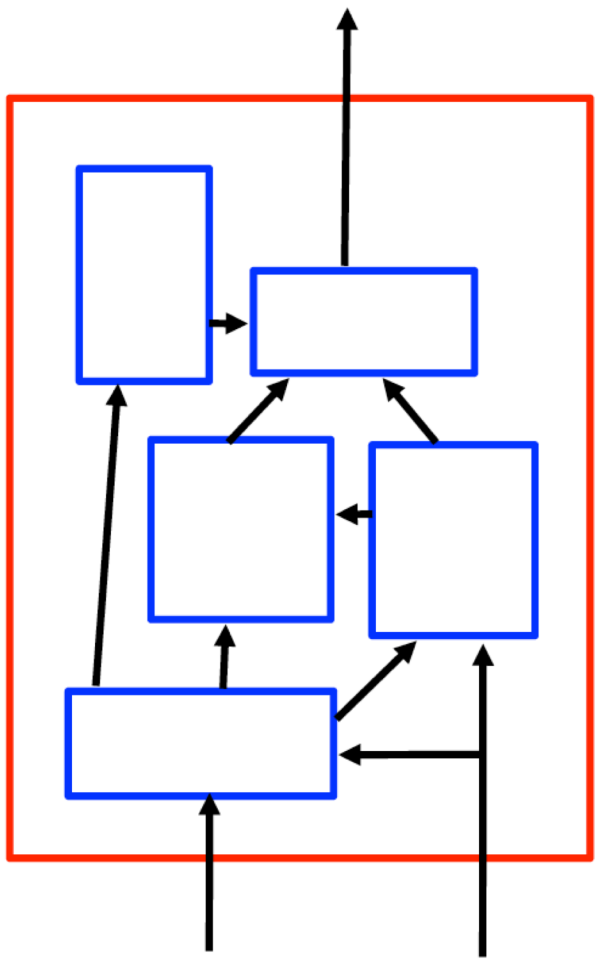
Note that $y$ should be the "raw" output of the classifier's decision function, not the predicted class label. For instance, in linear SVMs, $y = \mathbf{w} \cdot \mathbf{x} + b$, where $(\mathbf{w}, b)$ are the parameters of the hyperplane and $\mathbf{x}$ is the point to classify.

It can be seen that when $t$ and $y$ have the same sign (meaning $y$ predicts the right class) and $|y| \geq 1$, the hinge loss $\ell(y) = 0$, but when they have opposite sign, $\ell(y)$ increases linearly with $y$ (one-sided error).




Plot of hinge loss (blue, measured vertically) vs. zero-one loss (measured vertically; misclassification, green: $y < 0$) for $t = 1$ and variable $y$ (measured horizontally). Note that the hinge loss penalizes predictions $y < 1$, corresponding to the notion of a margin in a support vector machine.

# Any architecture works



- **Any connection graph is permissible**
  - Directed acyclic graphs (DAG)
  - Networks with loops must be "unfolded in time".
- **Any module is permissible**
  - As long as it is continuous and differentiable almost everywhere with respect to the parameters, and with respect to non-terminal inputs.
- **Most frameworks provide automatic differentiation**
  - Theano, Torch7+autograd,…
  - Programs are turned into computation DAGs and automatically differentiated.

# Backprop in Pratice

- Use ReLU non-linearities
- Use cross-entropy loss for classification
- Use Stochastic Gradient Descent on minibatches
- Shuffle the training samples (← very important)
- Normalize the input variables (zero mean, unit variance)
- Schedule to decrease the learning rate
- Use a bit of L1 or L2 regularization on the weights (or a combination)
  - But it's best to turn it on after a couple of epochs
- Use "dropout" for regularization
- Lots more in [LeCun et al. "Efficient Backprop" 1998]
- Lots, lots more in "Neural Networks, Tricks of the Trade" (2012 edition) edited by G. Montavon, G. B. Orr, and K-R Müller (Springer)
- More recent: Deep Learning (MIT Press book in preparation)

# Free Deep Learning Book 2016

**Ian Goodfellow and Yoshua Bengio and Aaron Courville**

## Deep Learning



URL: http://www.deeplearningbook.org

# Convolutional Networks

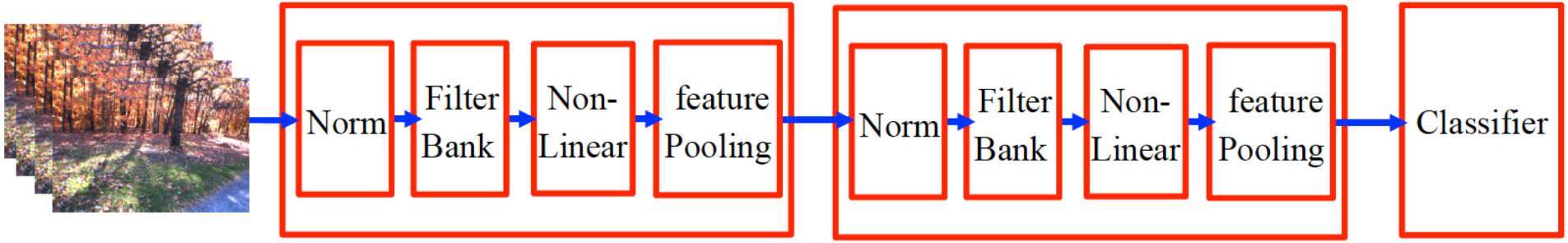
# Deep Learning = Training Multistage Machines

- Traditional Pattern Recognition: Fixed/Handcrafted Feature Extractor

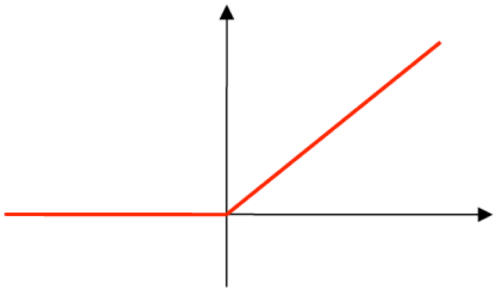

- Mainstream Pattern Recognition 9until recently)


- Deep Learning: Multiple stages/layers trained end to end

# Overall Architecture: multiple stages of Normalization → Filter Bank → Non-Linearity → Pooling

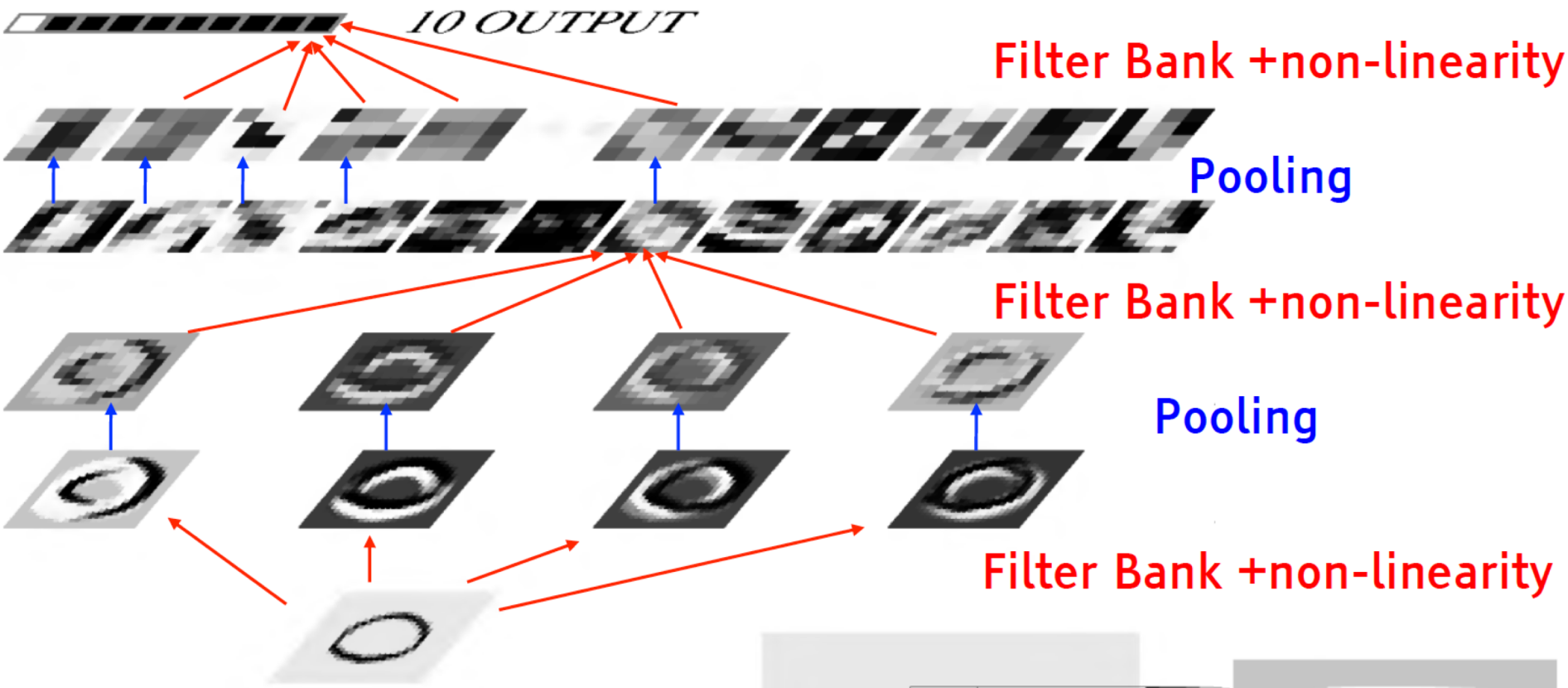


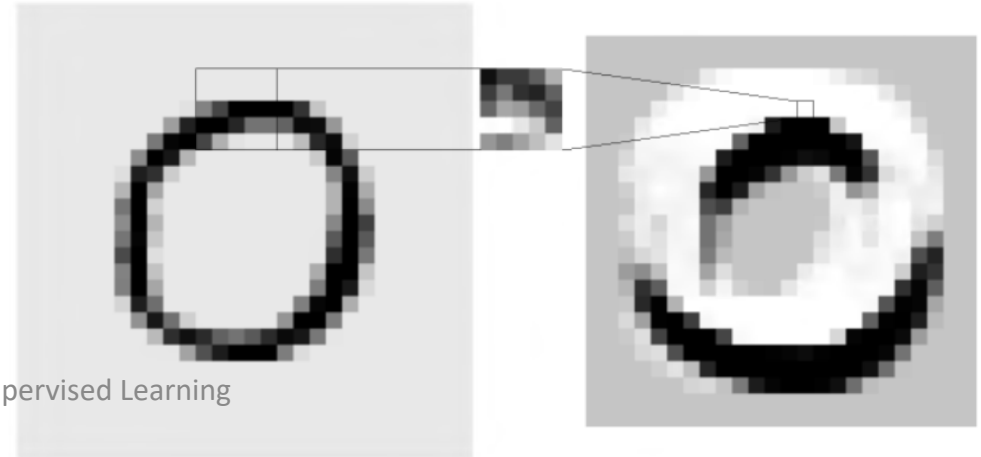

- **Normalization:** variation on whitening (optional)
  - Subtractive: average removal, high pass filtering
  - Divisive: local contrast normalization, variance normalization
- **Filter Bank:** dimension expansion, projection on overcomplete basis
- **Non-Linearity:** sparsification, saturation, lateral inhibition....
  - Rectification (ReLU), tanh, ....



- **Pooling:** aggregation over space or feature type
  - Max, log prob.

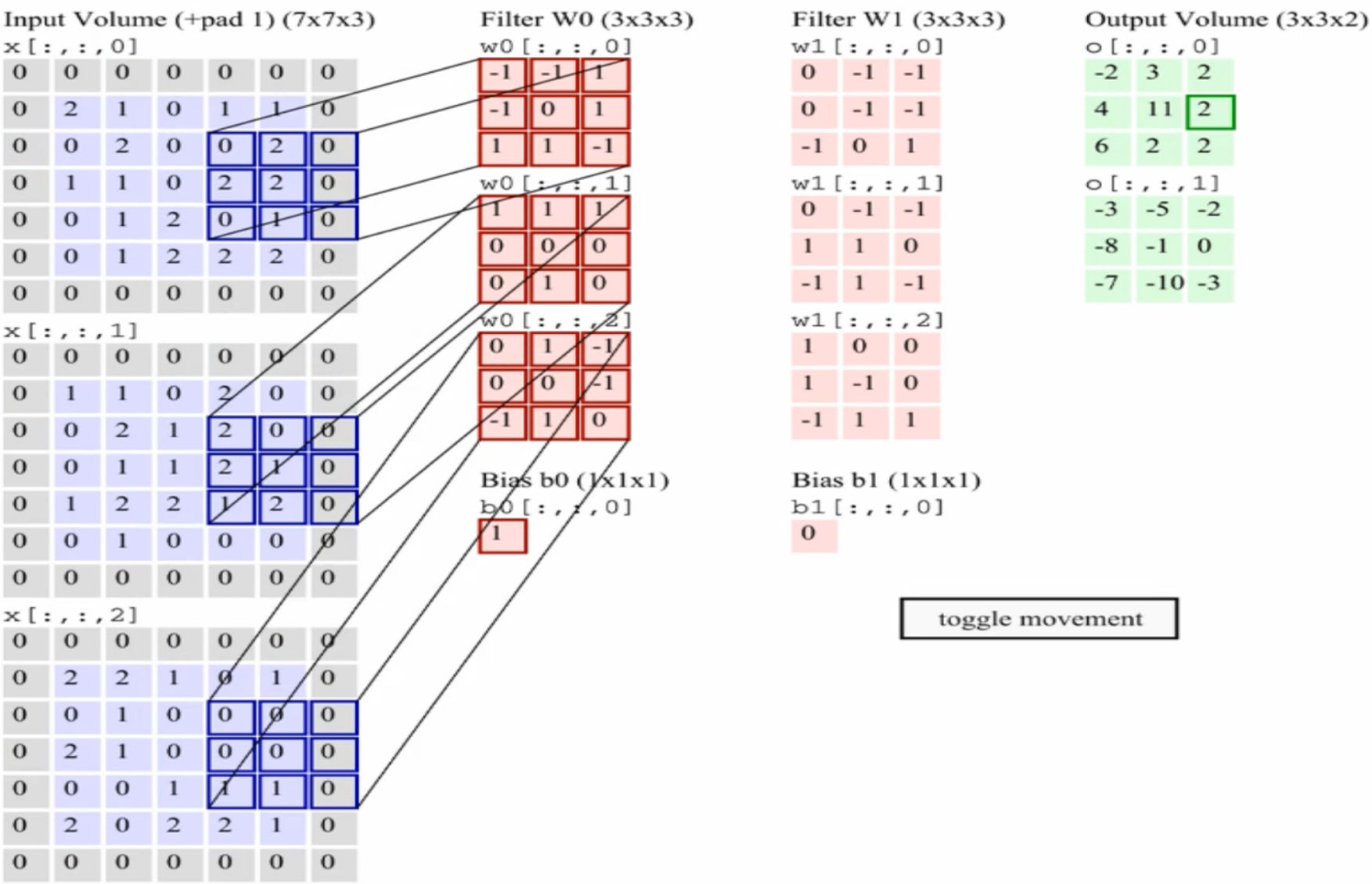
# Convolutional Architecture



- LeNet1 [LeCun et al. NIPS 1989]

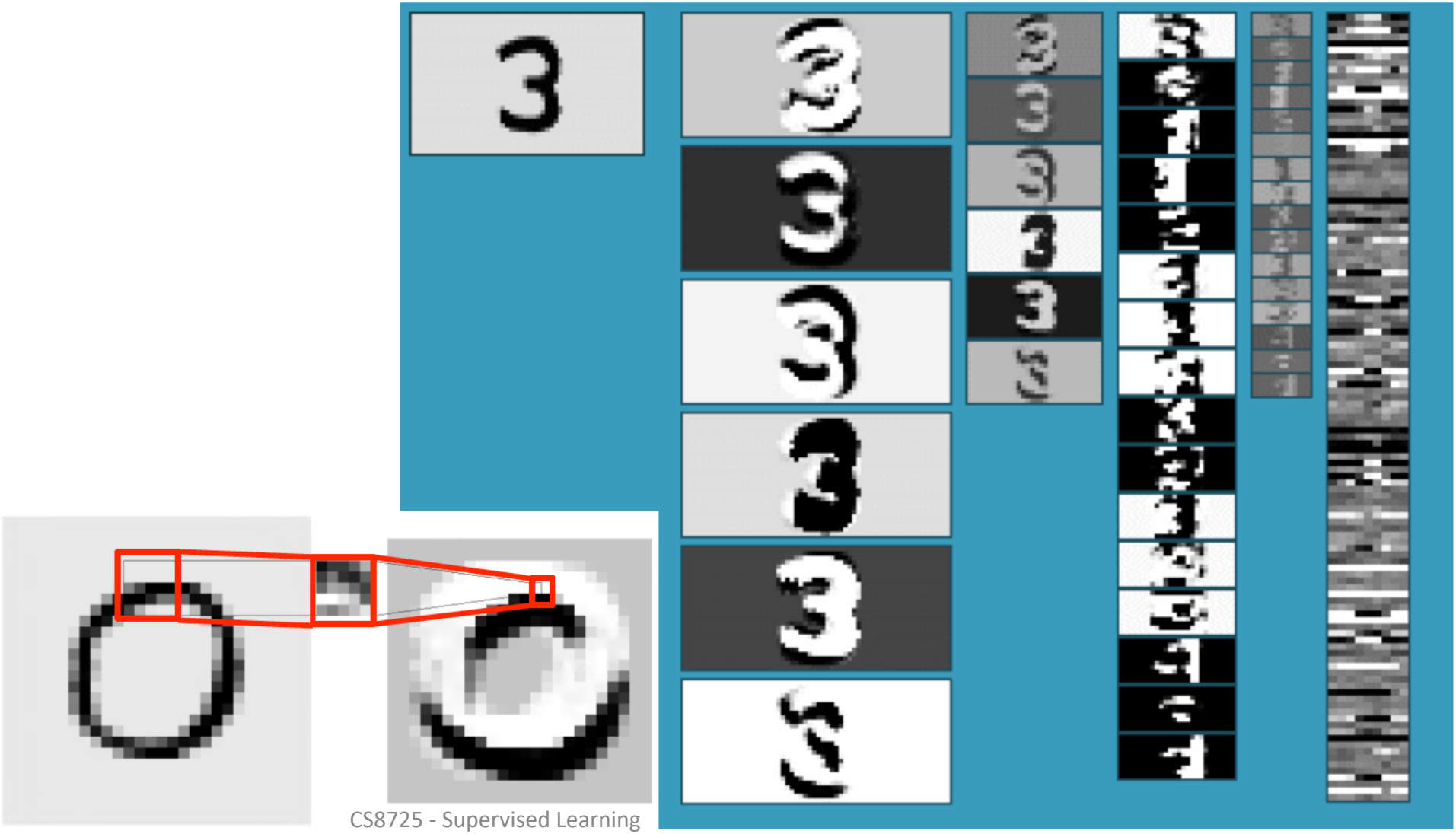
# Multiple Convolutions




Animation: Andrej Karpathy http://cs231n.github.io/convolutional-networks/

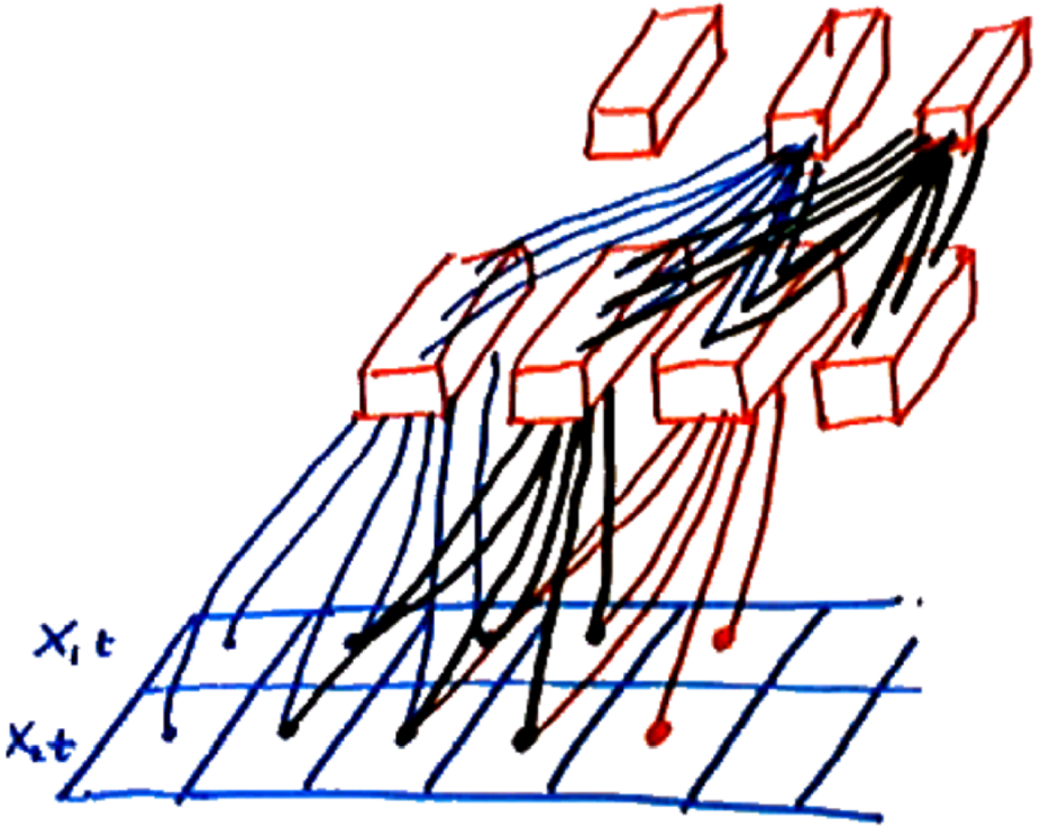
# Convolutional Networks (vintage 1990)

- filters → tanh → average-tanh → filters → tanh → average-tanh → filters → tanh

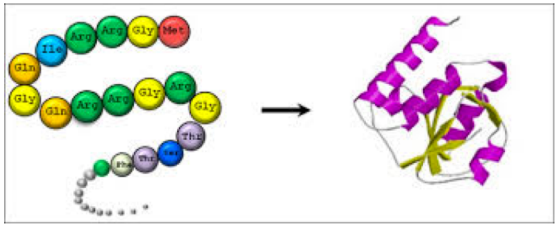


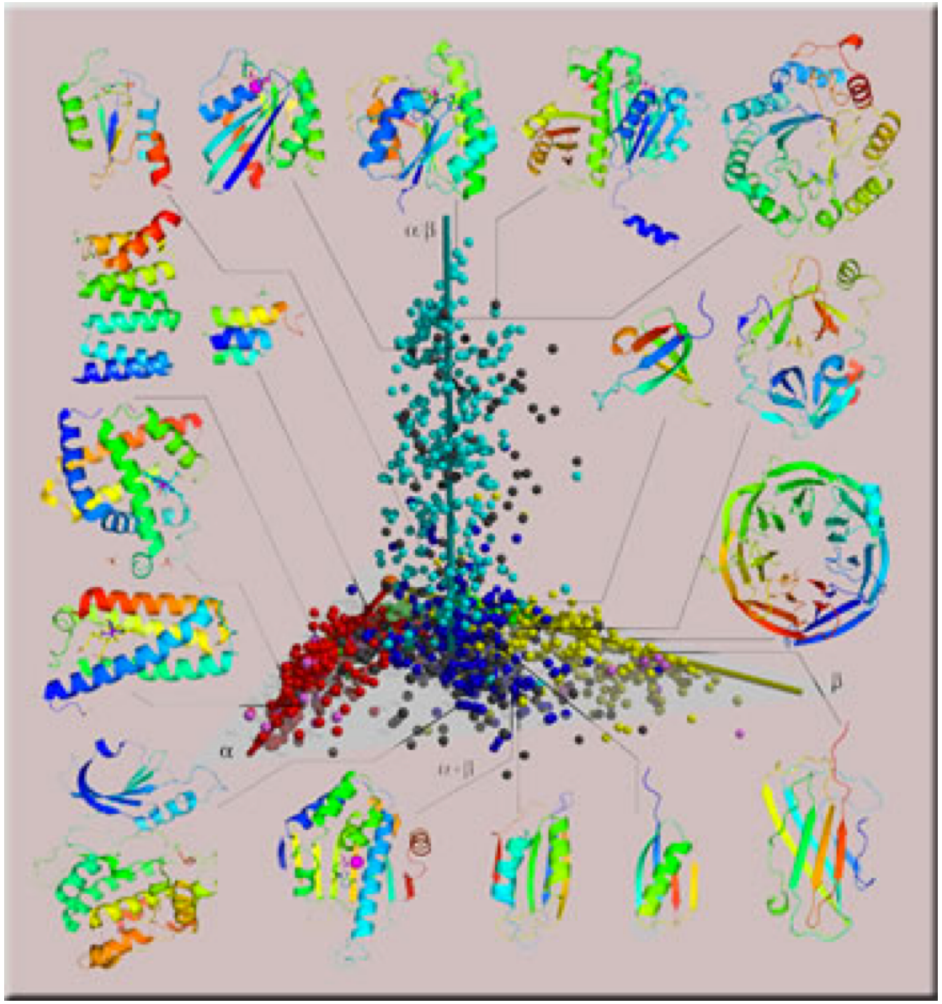
# 1D (Temporal) convolutional net

- 1D (Temporal) ConvNet, aka Timed-Delay Neural Nets
- Groups of units are replicated at each time step.
- Replicas have identical (shared) weights.

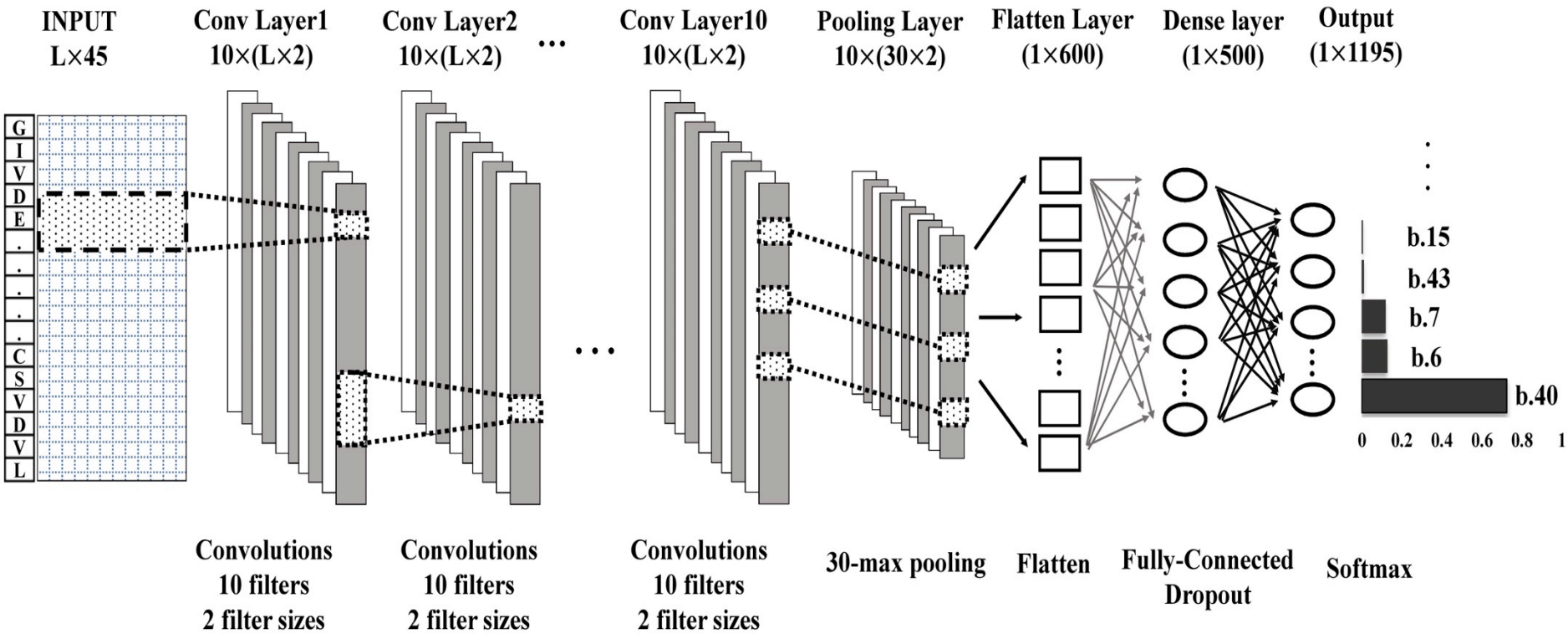


# 1D CNN for Protein Fold Classification



# Deep 1D-Convolution Neural Network

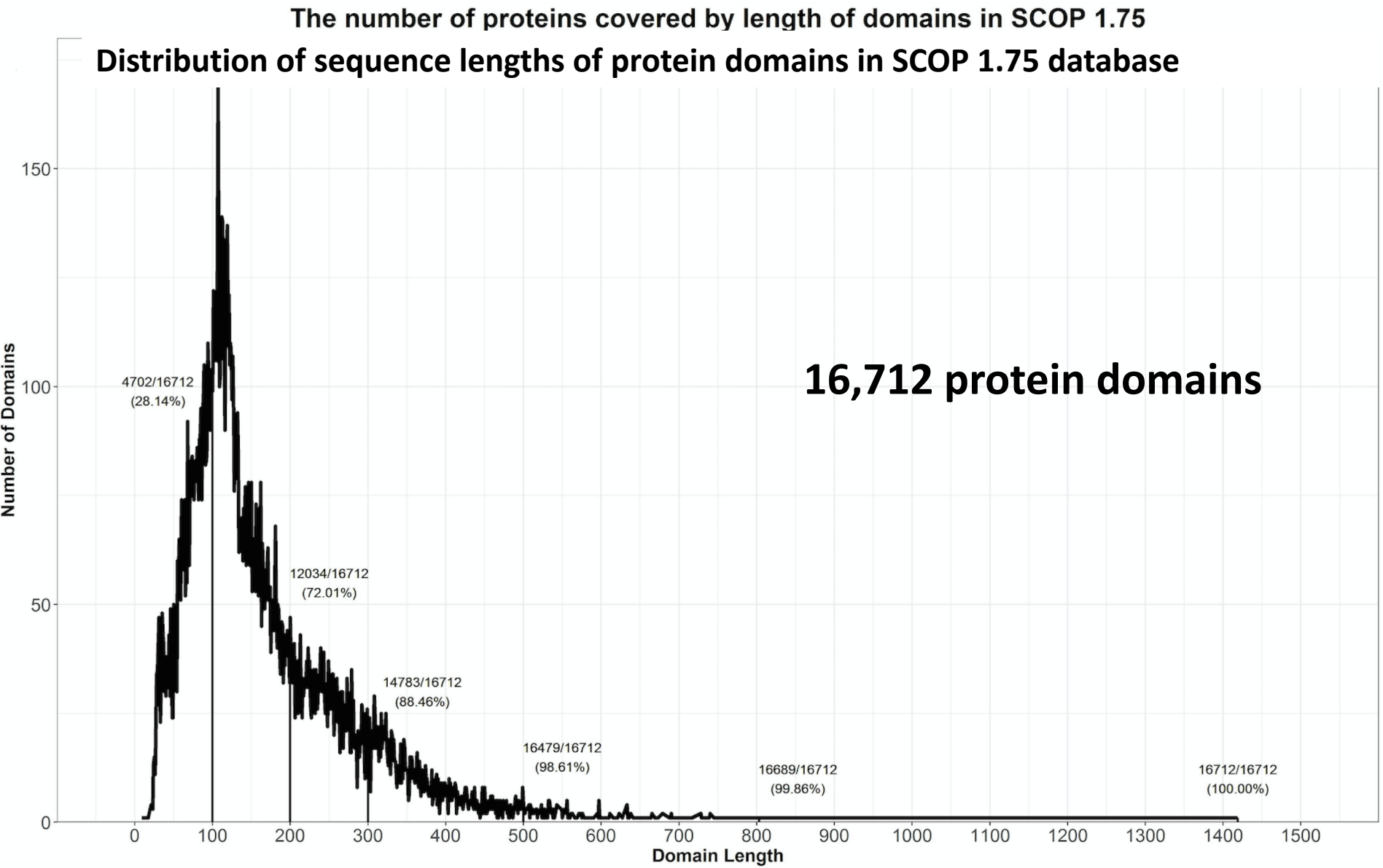



**Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$**
**Output layer: 1,195 nodes with sigmoid function**

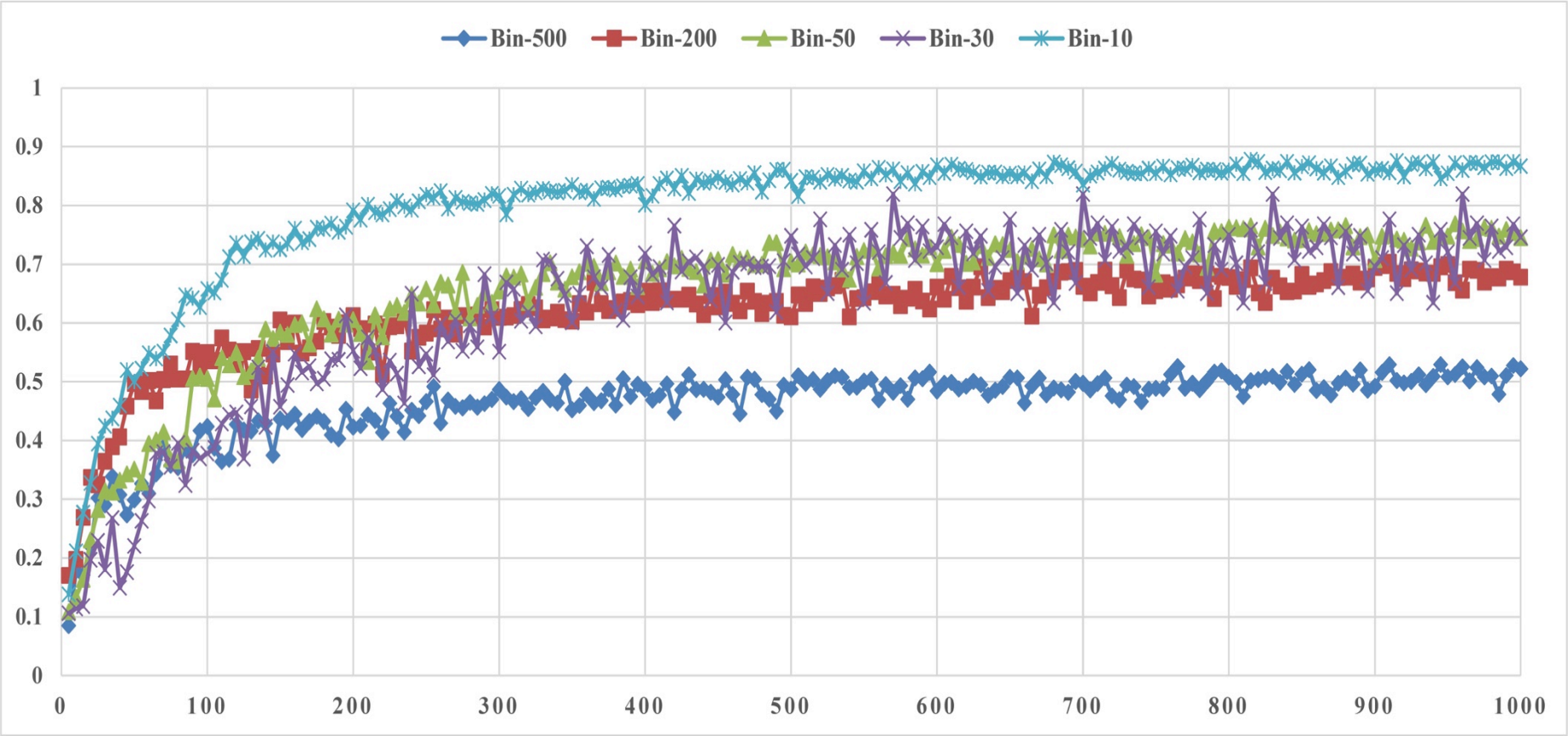


Hou et al., 2017

# Training Data



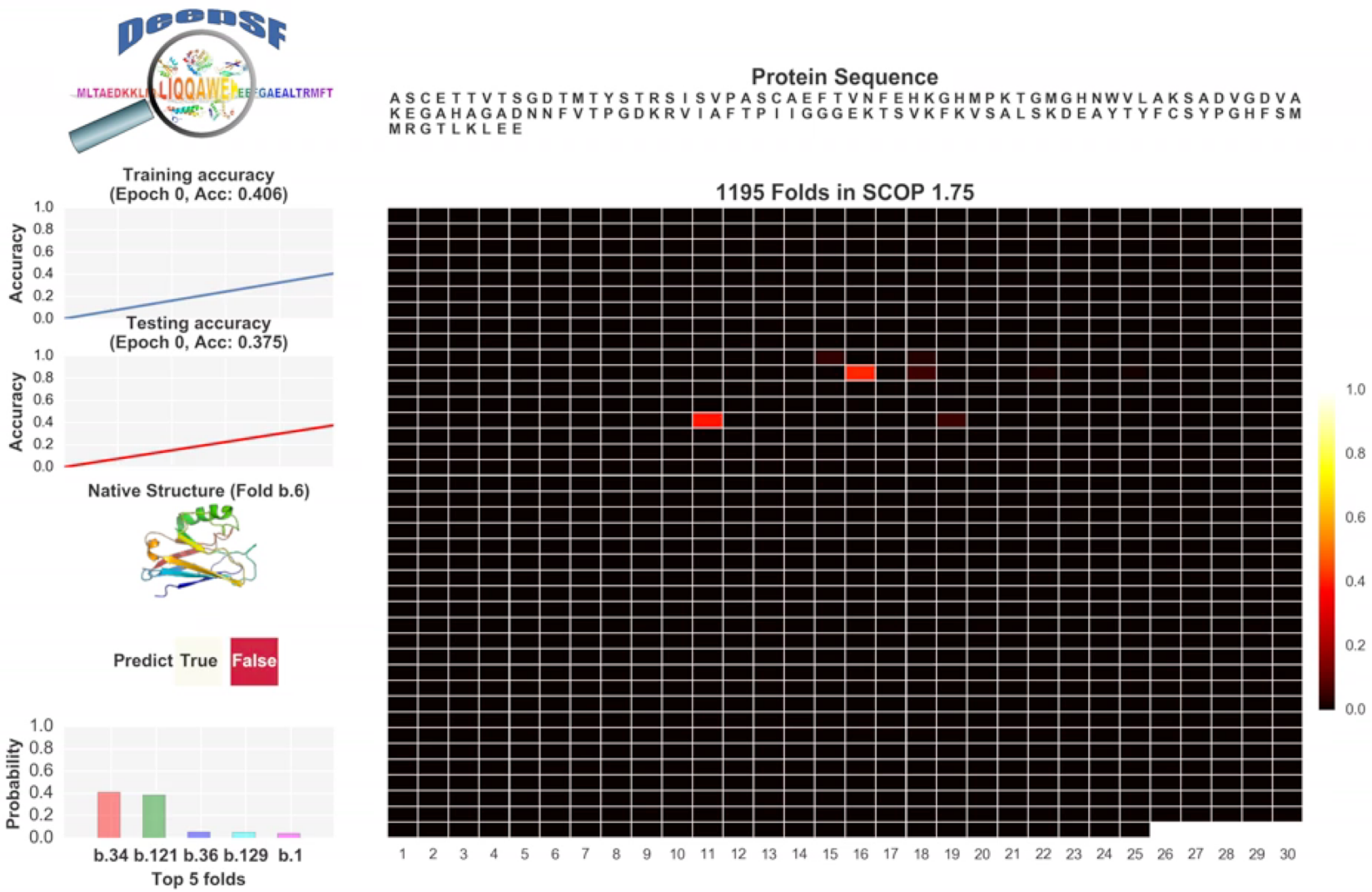
# Batch Training Using Binning and Padding according to Sequence Length






Hou et al., 2017

# Demo of Training DCNN



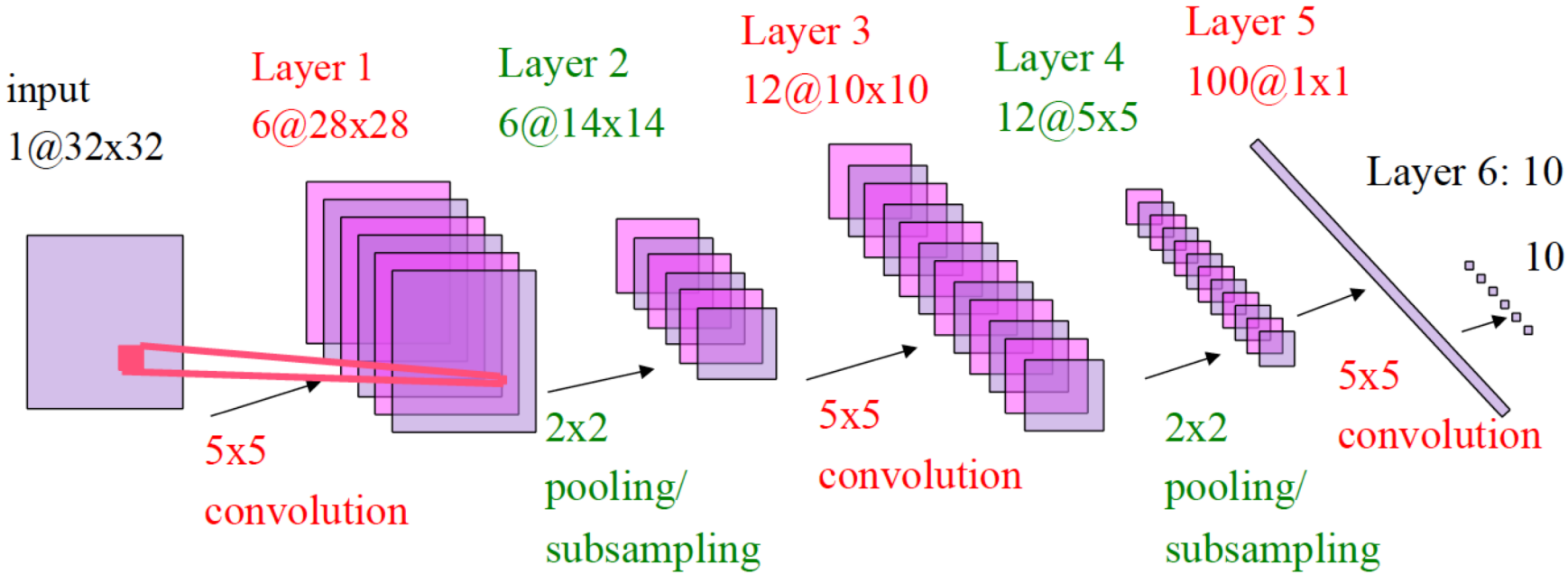
# Accuracy on Validation Datasets of SCOP1.75

| Num of Predictions | Set 1 (Sim < 95%) | Set 2 (Sim < 70%) | Set 3 (Sim < 40%) | Set 4 (Sim < 25%) | Average |
|---|---|---|---|---|---|
| Top 1 | 80.4% | 78.2% | 75.8% | 67.0% | 75.3% |
| Top 5 | 93.7% | 92.4% | 90.0% | 87.6% | 91.0% |

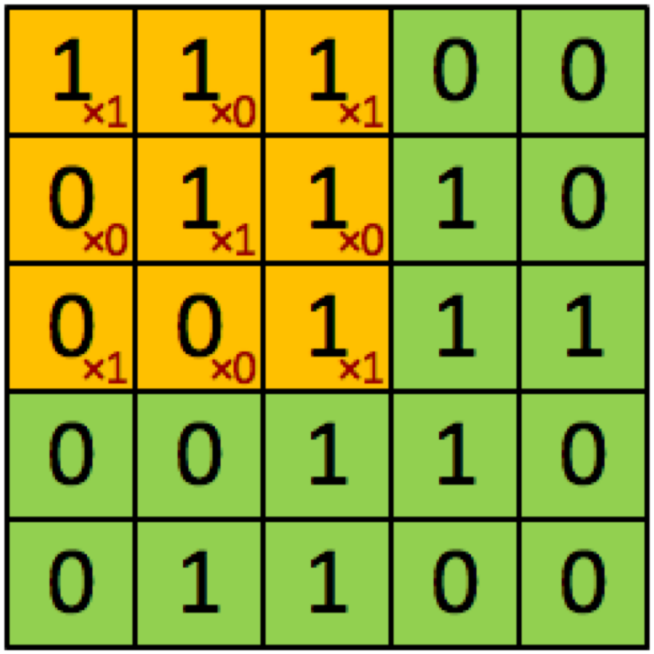
# Accuracy on Independent Dataset of SCOP 2.06 (4,418 proteins, Sim <= 40%)

| Method | Top 1 | Top 5 |
|---|---|---|
| DeepSF | 77% | 92% |
| MajorityAssignment | 4% | 16% |

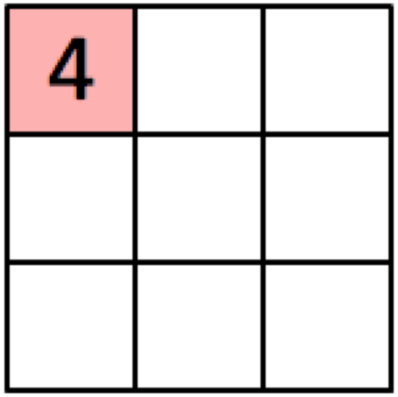
# Simple ConvNet for MNIST [LeCun 1998]



# Convolution Example without Padding



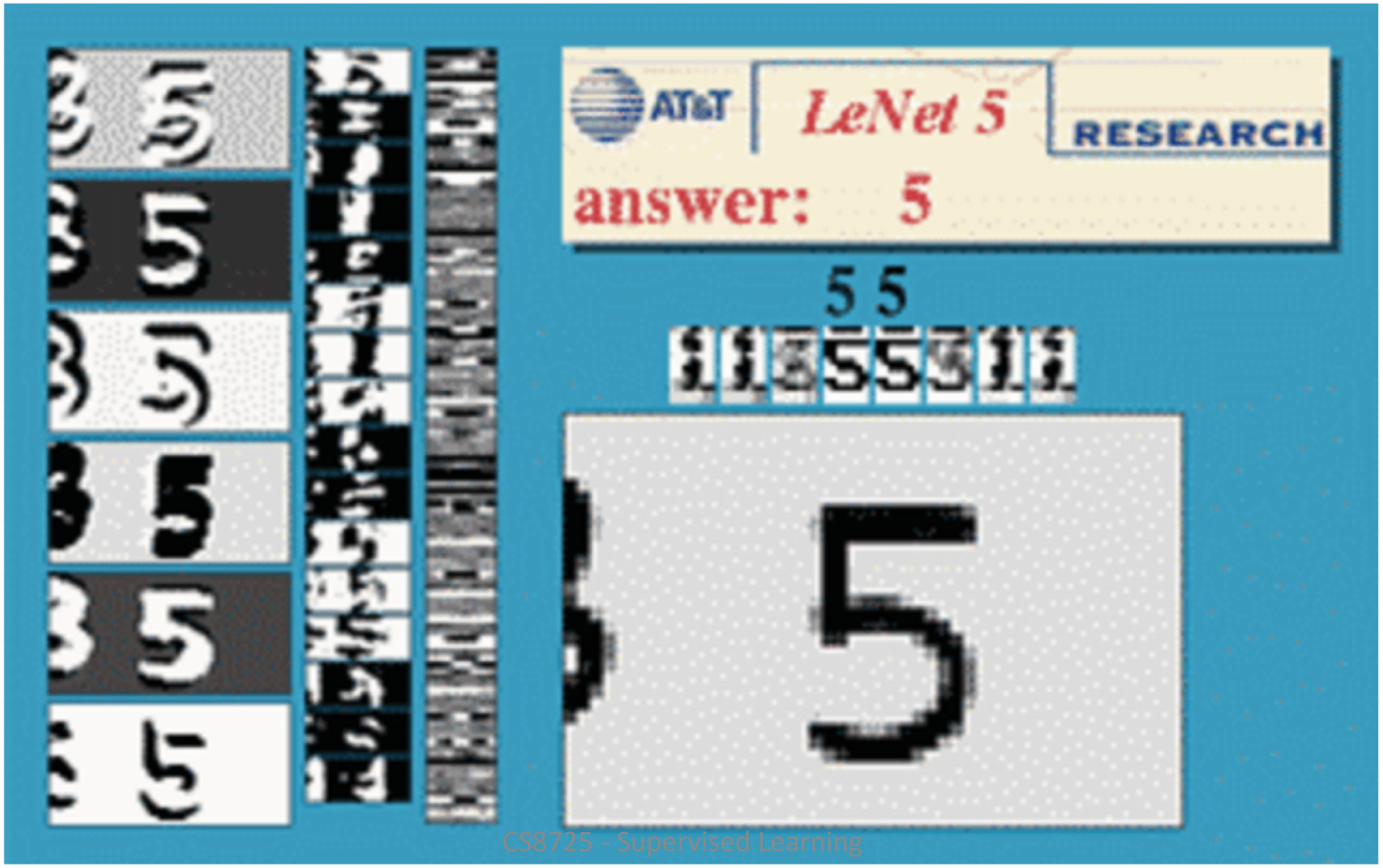

Image




Convolved Feature

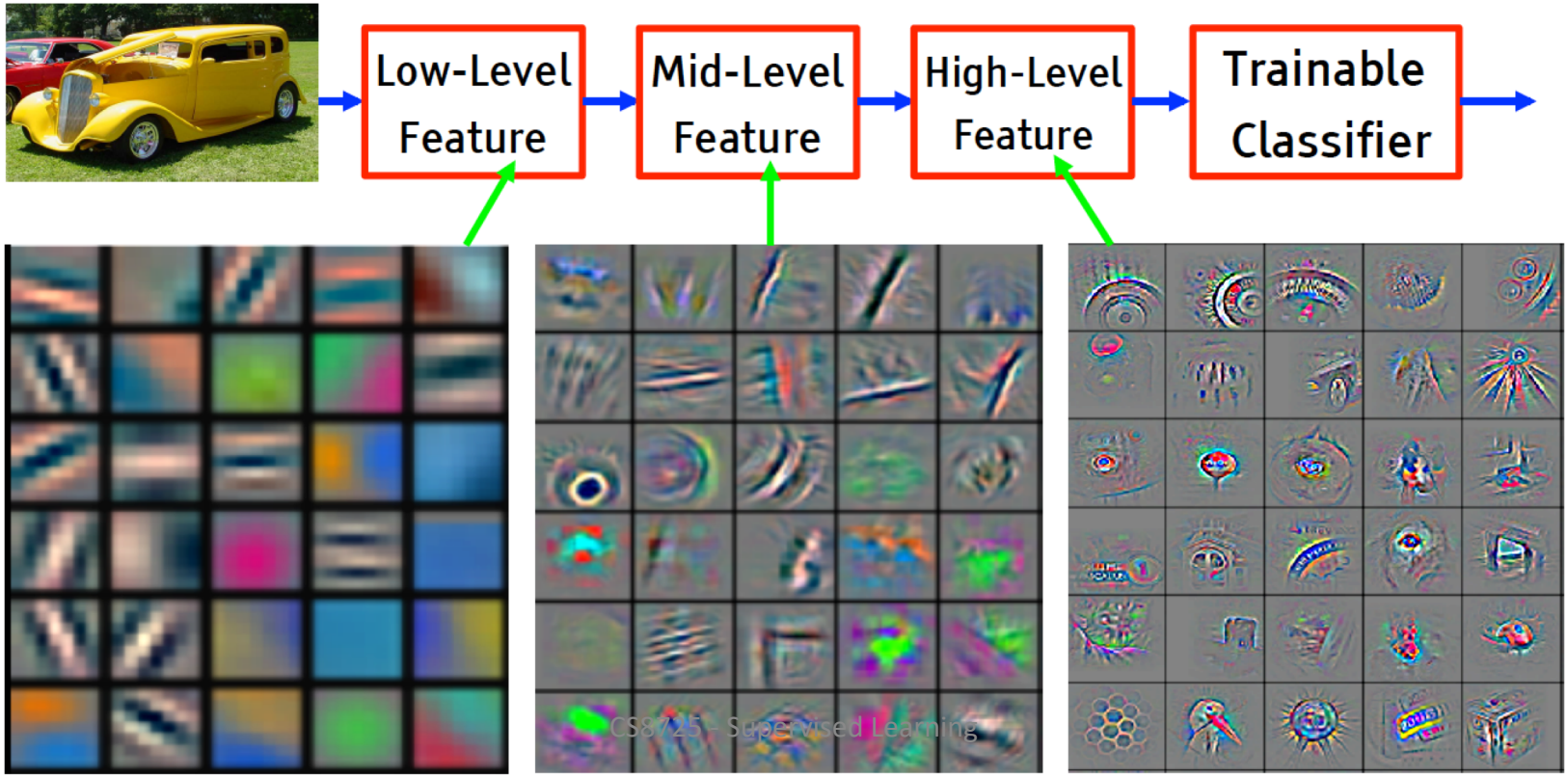
# Sliding Window ConvNet + Weighted FSM (Fixed Post-Proc)

[Matan, Burges, LeCun, Denker NIPS 1991] [LeCun, Bottou, Bengio, Haffner, Proc IEEE 1998]



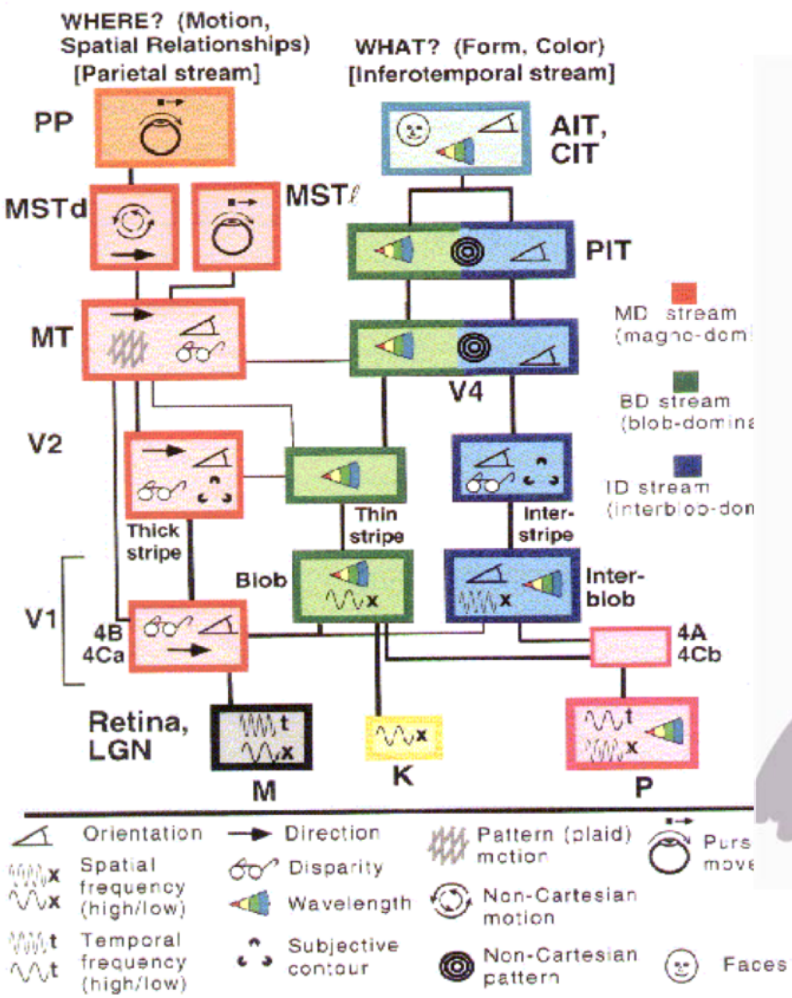
# Why Multiple Layers? The World is Compositional

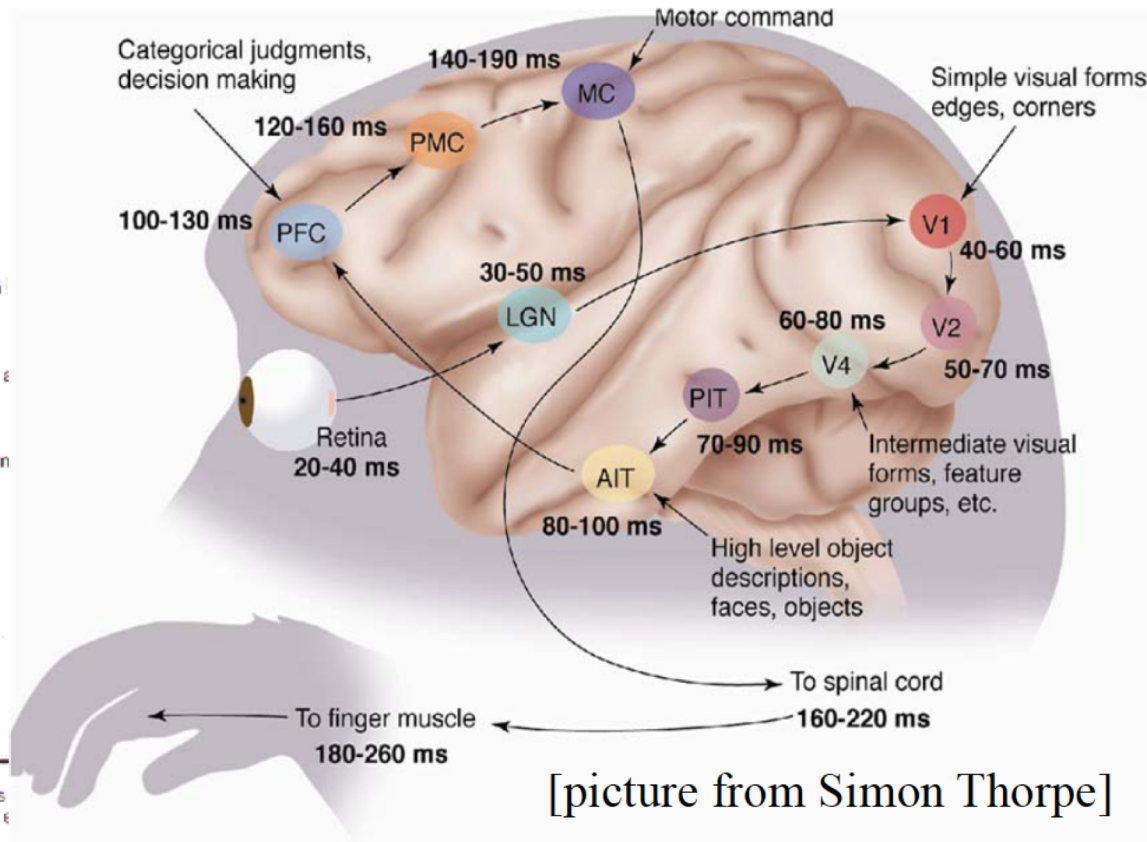
- Hierarchy of representations with increasing level of abstraction
- Each stage is a kind of trainable feature transform
- Image recognition: Pixel → edge → texton → motif → part → object
- Text: Character → word → word group → clause → sentence → story
- Speech: Sample → spectral band → sound → … → phone → phoneme → word



# ConvNets are somewhat inspired by the Visual Cortex

- The ventral (recognition) pathway in the visual cortex has multiple stages
- Retina - LGN - V1 - V2 - V4 - PIT - AIT ....




[picture from Simon Thorpe]

[Gallant & Van Essen]

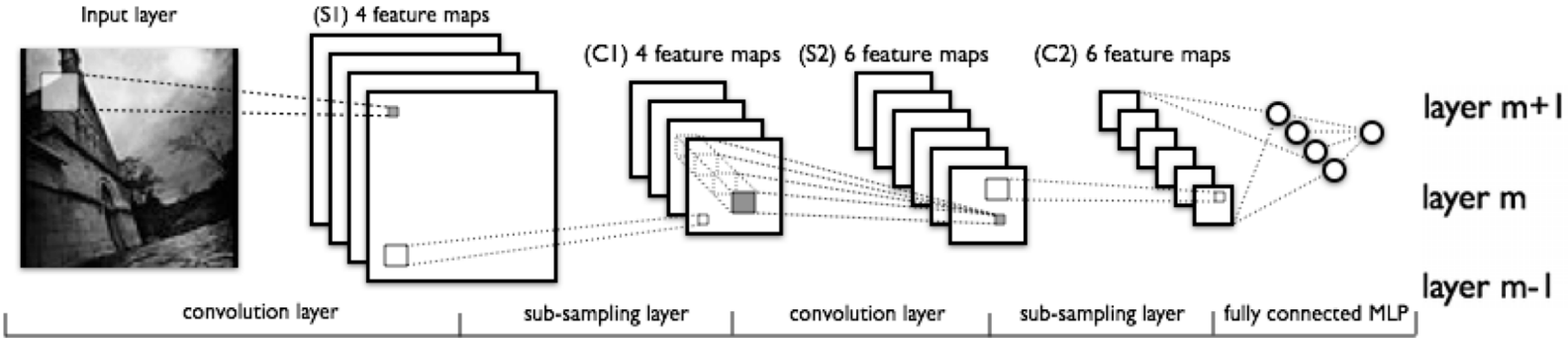
# What are ConvNets Good For

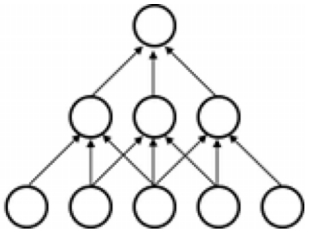
- Signals that comes to you in the form of (multidimensional) arrays.
- Signals that have strong local correlations
- Signals where features can appear anywhere
- Signals in which objects are invariant to translations and distortions.

- 1D ConvNets: sequential signals, text
  - Text Classification
  - Musical Genre Recognition
  - Acoustic Modeling for Speech Recognition
  - Time-Series Prediction
- 2D ConvNets: images, time-frequency representations (speech and audio)
  - Object detection, localization, recognition
- 3D ConvNets: video, volumetric images, tomography images
  - Video recognition / understanding
  - Biomedical image analysis
  - Hyperspectral image analysis

# CNN – Translation Invariance

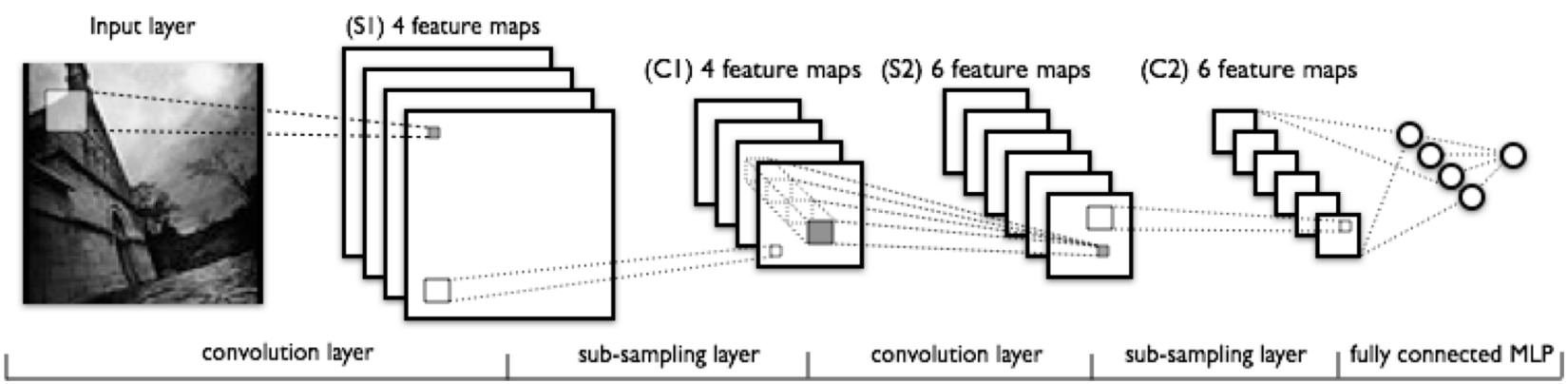
- The 2-*d* planes of nodes (or their outputs) at subsequent layers in a CNN are called *feature maps*
- To deal with translation invariance, each node in a feature map has the *same weights* (based on the feature it is looking for), and each node connects to a different overlapping receptive field of the previous layer
- Thus each *feature map* searches the full previous layer to see if, where, and how often its feature occurs (precise position less critical)
  - The output will be high at each node in the map corresponding to a receptive field where the feature occurs
  - Later layers could concern themselves with higher order combinations of features and rough relative positions
  - Each calculation of a node's net value, $\Sigma xw+b$ in the feature map, is called a convolution, based on the similarity to standard convolutions



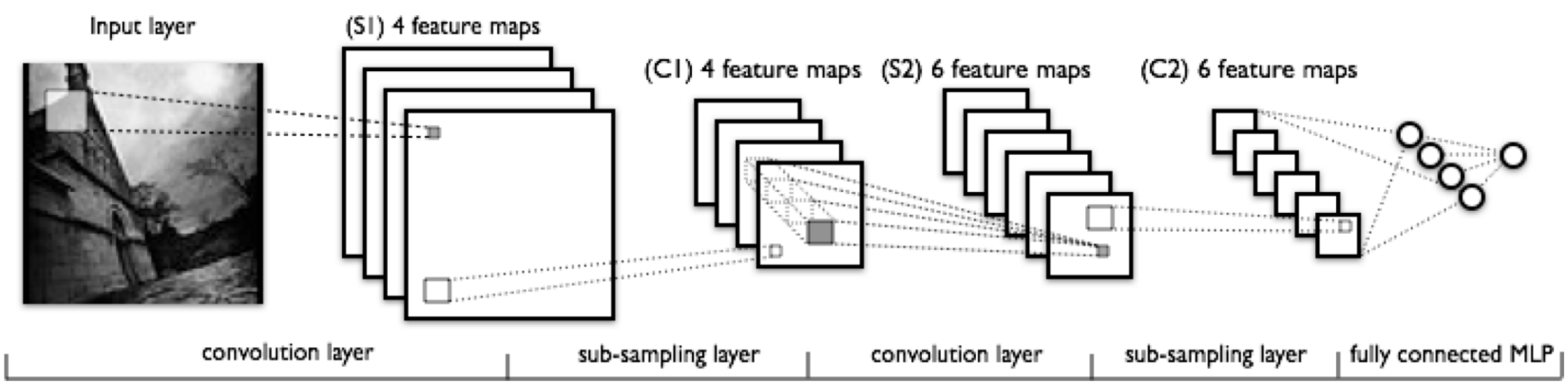
# CNN Structure

- Each node (e.g. convolution) is calculated for each receptive field in the previous layer
  - During training the corresponding weights are always tied to be the same (ala BPTT)
  - Thus a relatively small number of unique weight parameters to learn, although they are replicated many times in the feature map
  - Each node output in CNN is $f(\Sigma xw + b)$ (ReLU, tanh etc.)
  - Multiple feature maps in each layer
  - Each feature map should learn a different translation invariant feature
  - Since after first layer, there are always multiple feature maps to connect to the next layer, it is a pre-made human decision as to which previous maps the current convolution map receives inputs from, could connect to all or a subset
- Convolution layer causes total number of features to increase

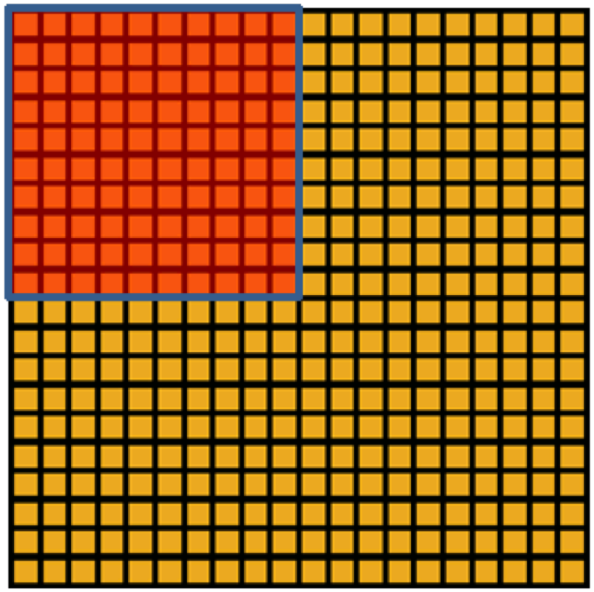


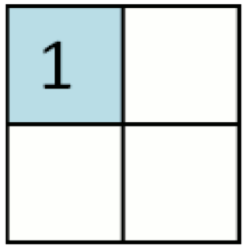
# Sub-Sampling (Pooling)

- Convolution and sub-sampling layers are interleaved
- Sub-sampling (Pooling) allows number of features to be diminished, and to pool information
  - Pooling replaces the network output at a certain point with a summary statistic of nearby outputs
  - Max-Pooling common (Just as long as the feature is there, take the max, as exact position is not that critical), also averaging, etc.
  - Pooling smooths the data  and reduces spatial resolution and thus naturally decreases importance of exactly where a feature was found, just keeping the rough location – translation invariance
  - 2x2 pooling would do 4:1 compression, 3x3 9:1, etc.
  - Convolution usually increases number of feature maps, pooling keeps same number of reduced maps (one-to-one correspondence of convolution map to pooled map) as the previous layer



# Pooling Example (Summing or averaging)



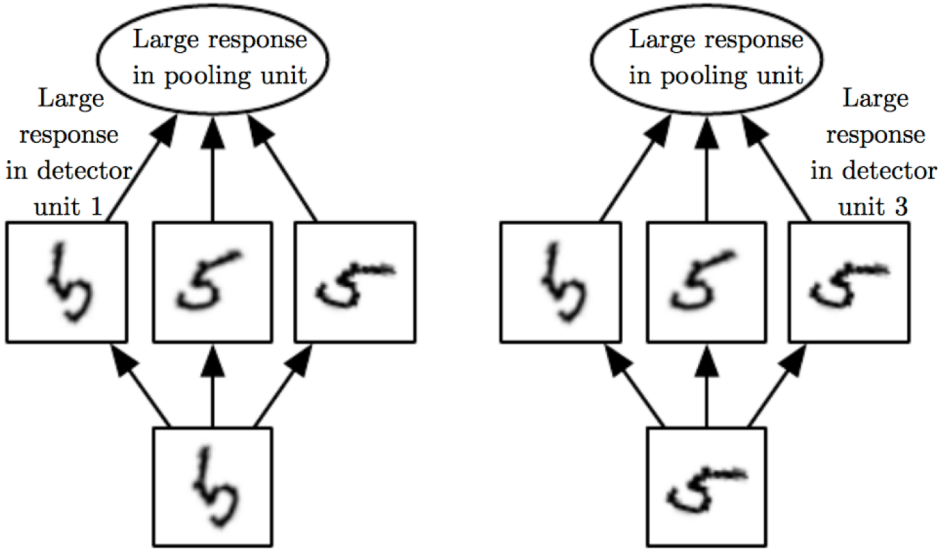



Convolved feature

Pooled feature

# Pooling (cont.)

- Common layers are convolution, non-linearity, then pool (repeat)
- Note that pooling always decreases map volumes (unless pool stride = 1, highly overlapped), making real deep nets more difficult. Pooling is sometimes used only after multiple convolved layers and sometimes not at all.
- At later layers pooling can make network invariant to more than just translation – *learned invariances*



# CNN Training

- **Trained with BP but with weight tying in each feature map**
  - **Randomized initial weights** through entire network
  - Just **average the weight updates** over the tied weights in feature map layers
- **Convolution layer**
  - **Each feature map has one weight matrix for each input and one bias**
  - Thus a feature map with a 5x5 receptive field (filter) would have a total of 26 weights, which are the same coming into each node of the feature map
  - If a convolution layer had 10 feature maps, then only a total of 260 unique weights to be trained in that layer (much less than an arbitrary deep net layer without sharing)
- **Sub-Sampling (Pooling) Layer**
  - <u>All elements of receptive field max'd, averaged, summed, etc. Result multiplied by one trainable weight and a bias added, then passed through non-linear function (detector, e.g. ReLU) for each pooling node</u>
  - If a layer had 10 pooling feature maps, then 20 unique weights to be trained

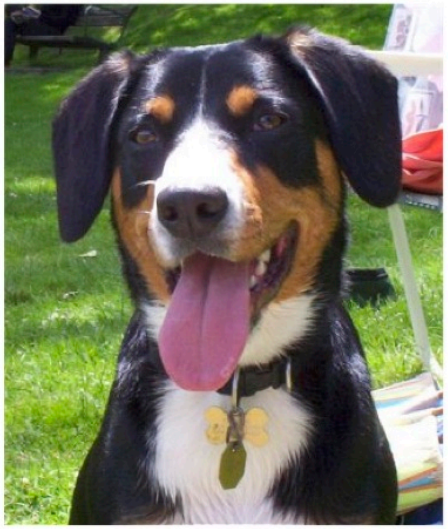
# CNN Hyperparameters

- **Structure** itself, **number of layers**, **size of filters**, **number of feature maps** in convolution layers, **connectivity** between layers, **activation functions**, **final supervised layers, Pooling parameters**, etc.
- **Drop-out often used in final fully connected layers for overfit avoidance** – less critical in convolution/pooling layers which already regularize due to weight sharing
- ***Stride*** – if don't have to test every location for the feature (i.e. stride = 1), could sample more coarsely
  - Another option for **down-sampling**
- As is, the feature map would always decrease in volume which is not always desirable - ***Zero-padding*** **avoids this and lets us maintain up to the same volume**
  - Would shrink fast for large kernel/filter sizes and would limit the depth (number of layers) in the network
  - Also **allows the different filter sizes to fit arbitrary map widths**

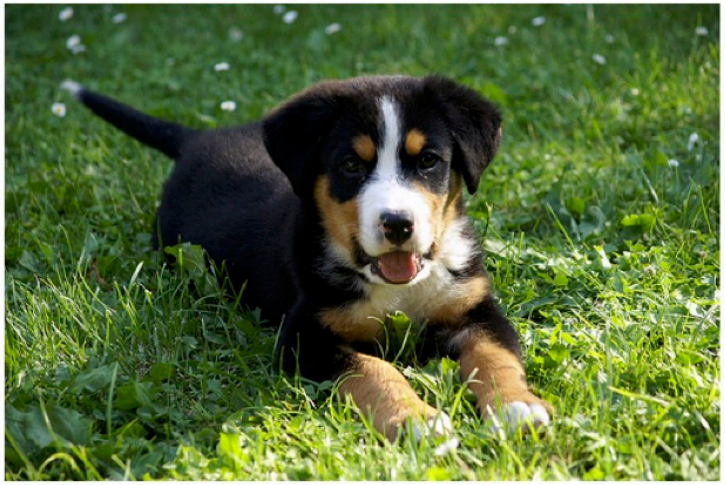
# ILSVRC Image net Large Scale Vision Recognition Competition

RGB: 224 x 224 x 3 = 150,528 raw real valued features

- Annual competition of image classification at large scale
- 1.2M images in 1K categories
- Classification: make 5 guesses about the image label

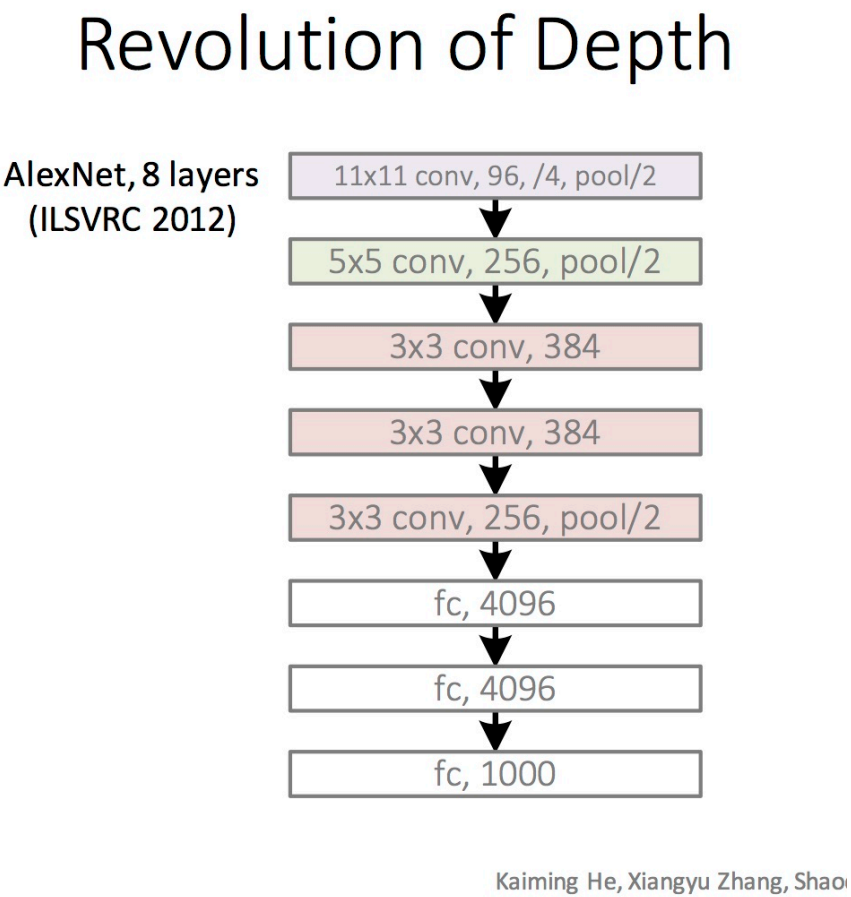


EntleBucher



Appenzeller

# Example CNNs Structures ILSVRC winners

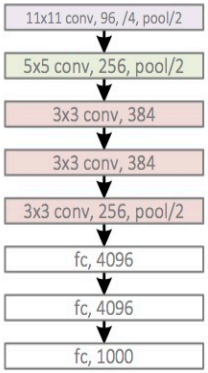



- Note Pooling considered part of the layer
- 96 convolution kernels, then 256, then 384
- Stride of 4 for first convolution kernel, 1 for the rest
- Pooling layers with 3x3 receptive fields and stride of 2 throughout
- Finishes with fully connected (fc) MLP with 2 hidden layers and 1000 output nodes for classes

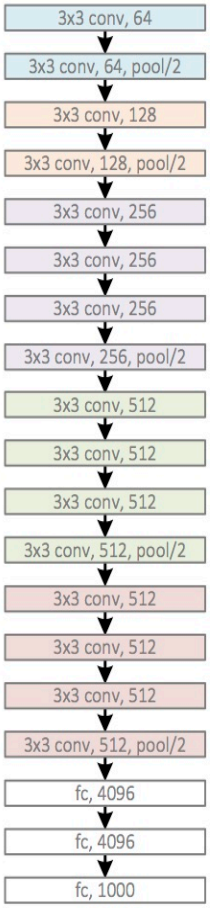
# Example CNNs Structures ILSVRC winners

## Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)




VGG, 19 layers
(ILSVRC 2014)

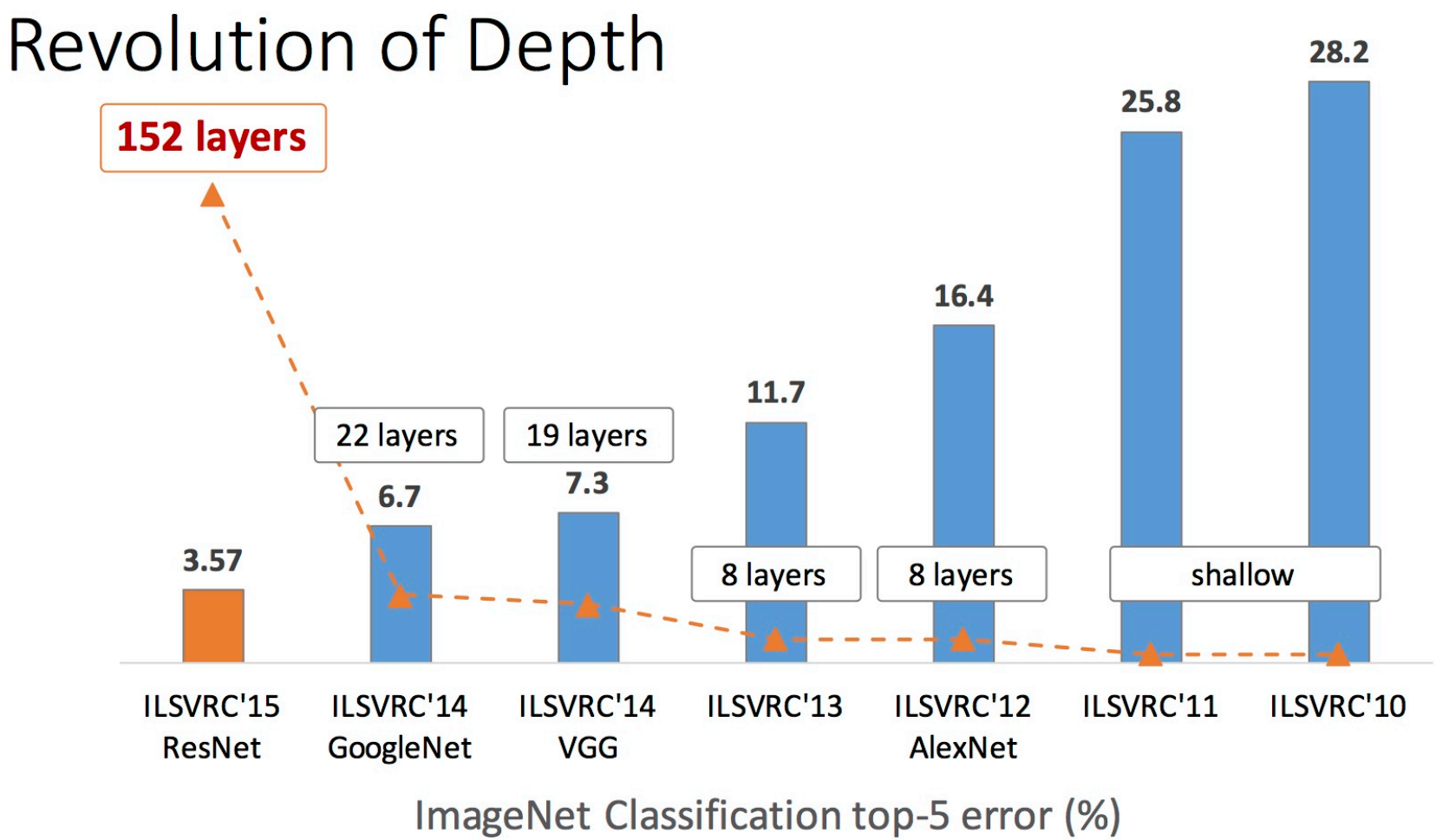



GoogleNet, 22 layers
(ILSVRC 2014)



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. CVPR 2016.

# Increasing Depth






Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR

# Go Deep with Residual Network

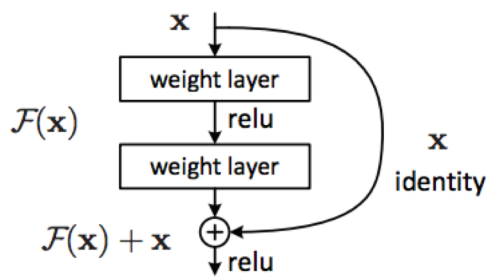
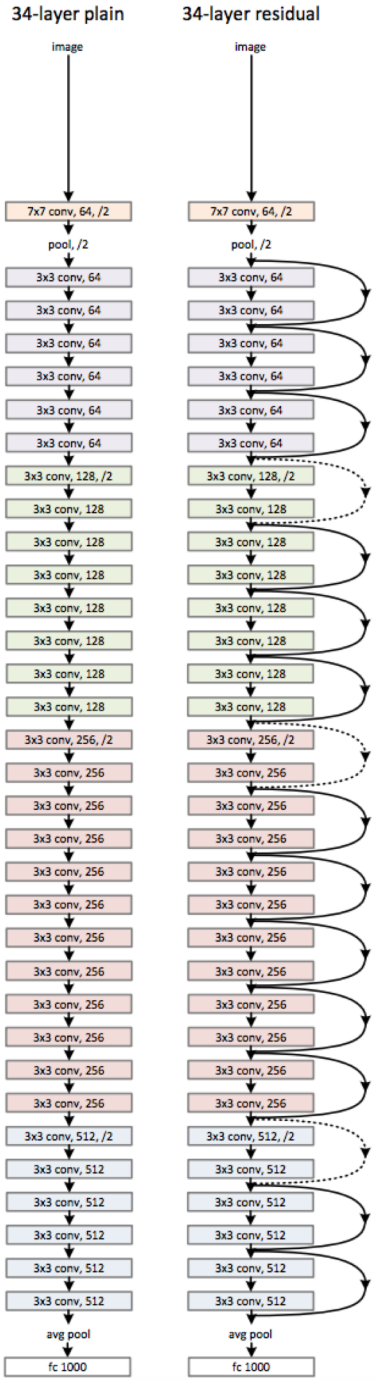



Figure 2. Residual learning: a building block.




He et al., 2015

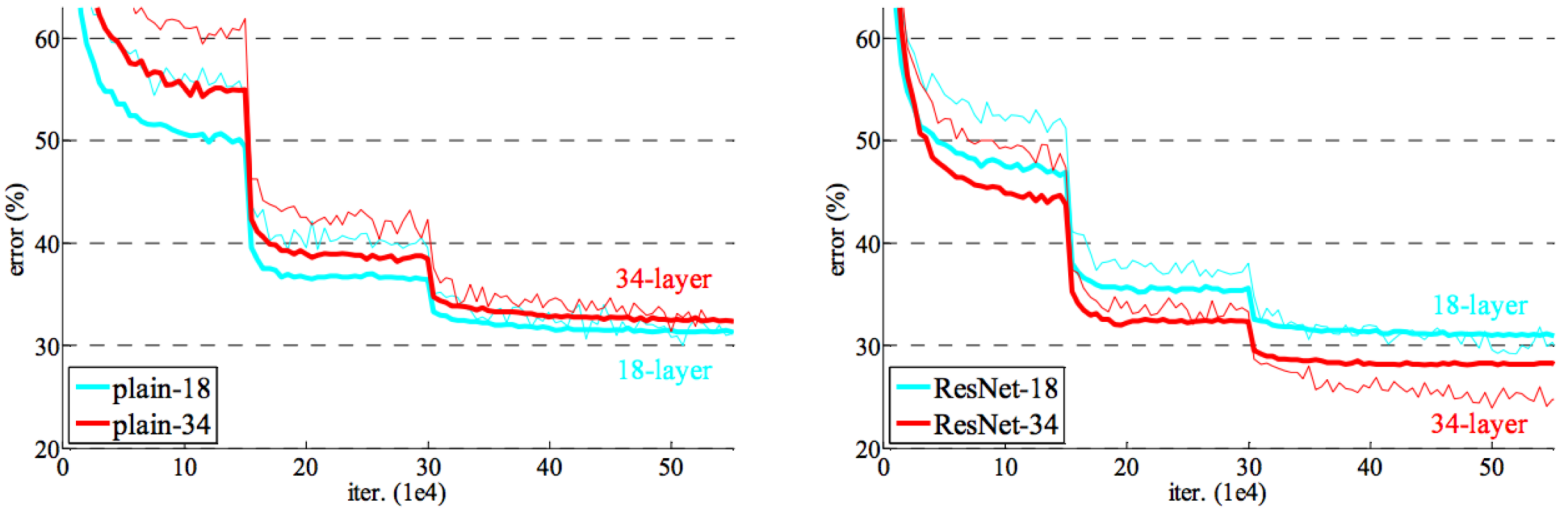


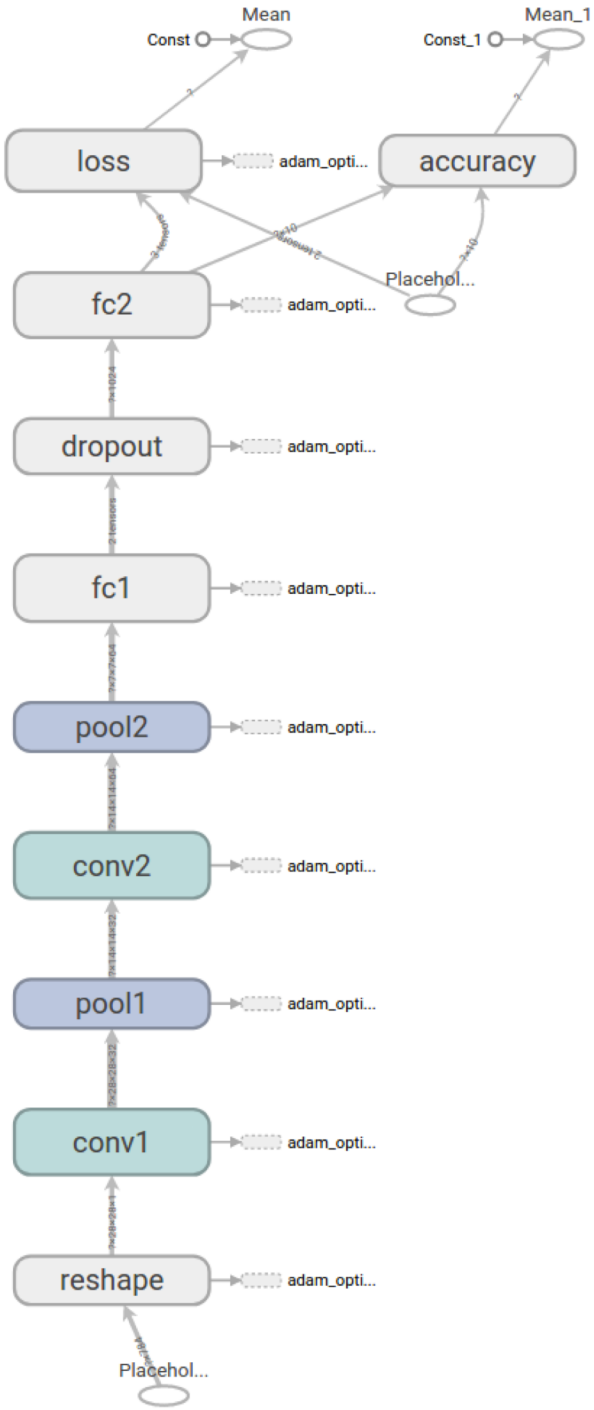

Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

# CNN Summary

- **High accuracy for image applications** – Breaking all records and doing it using just **using just raw pixel features**!
- **Special purpose net** – good for images or problems with strong grid-like or sequential local spatial/temporal correlation
- Once trained on one problem (e.g. vision) could use same net (often tuned) for a new similar problem – general creator of vision features (**Transfer learning)**
- Unlike traditional nets, **handles variable sized inputs**
  - Same filters and weights, just convolve across different sized image and dynamically scale size of pooling regions (not # of nodes), to normalize
  - Different sized images, different length speech segments, etc.
- **Lots of hand crafting and CV tuning to find the right recipe of receptive fields, layer interconnections**, etc.
  - **Lots more Hyperparameters than standard nets**, and even than other deep networks, since the structures of CNNs are more handcrafted
  - **CNNs getting wider and deeper with speed-up techniques (e.g. GPU, ReLU**, etc.) and lots of current research, excitement, and success

# Demo of DCNN with TensorFlow

**Build a DCNN to classify digital images on MNIST dataset**



# Key code

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

TensorFlow also gives us a lot of flexibility in convolution and pooling operations. How do we handle the boundaries? What is our stride size? In this example, we're always going to choose the vanilla version. Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input. Our pooling is plain old max pooling over 2x2 blocks. To keep our code cleaner, let's also abstract those operations into functions.

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
```

## First Convolutional Layer

We can now implement our first layer. It will consist of convolution, followed by max pooling. The convolution will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of `[5, 5, 1, 32]`. The first two dimensions are the patch size, the next is the number of input channels, and the last is the number of output channels. We will also have a bias vector with a component for each output channel.

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

To apply the layer, we first reshape `x` to a 4d tensor, with the second and third dimensions corresponding to image width and height, and the final dimension corresponding to the number of color channels.

```
x_image = tf.reshape(x, [-1, 28, 28, 1])
```

We then convolve `x_image` with the weight tensor, add the bias, apply the ReLU function, and finally max pool. The `max_pool_2x2` method will reduce the image size to 14x14.

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

## Second Convolutional Layer

In order to build a deep network, we stack several layers of this type. The second layer will have 64 features for each 5x5 patch.

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

## Densely Connected Layer

Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons to allow processing on the entire image. We reshape the tensor from the pooling layer into a batch of vectors, multiply by a weight matrix, add a bias, and apply a ReLU.

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

## Dropout

To reduce overfitting, we will apply dropout before the readout layer. We create a `placeholder` for the probability that a neuron's output is kept during dropout. This allows us to turn dropout on during training, and turn it off during testing. TensorFlow's `tf.nn.dropout` op automatically handles scaling neuron outputs in addition to masking them, so dropout just works without any additional scaling.[1]

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

## Readout Layer

Finally, we add a layer, just like for the one layer softmax regression above.

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

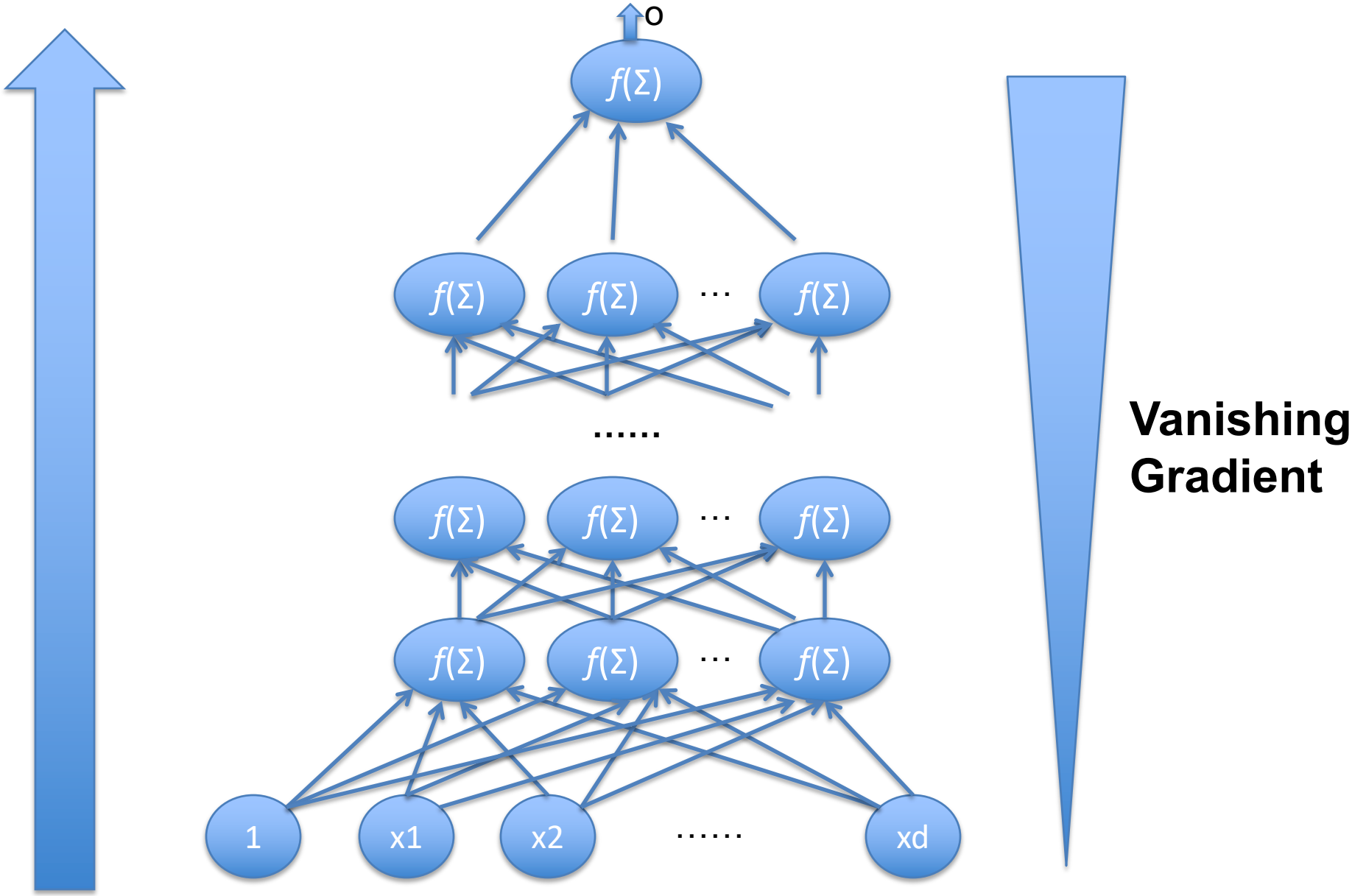
# Train and Evaluate

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i % 100 == 0:
      train_accuracy = accuracy.eval(feed_dict={
          x: batch[0], y_: batch[1], keep_prob: 1.0})
      print('step %d, training accuracy %g' % (i, train_accuracy))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

  print('test accuracy %g' % accuracy.eval(feed_dict={
      x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

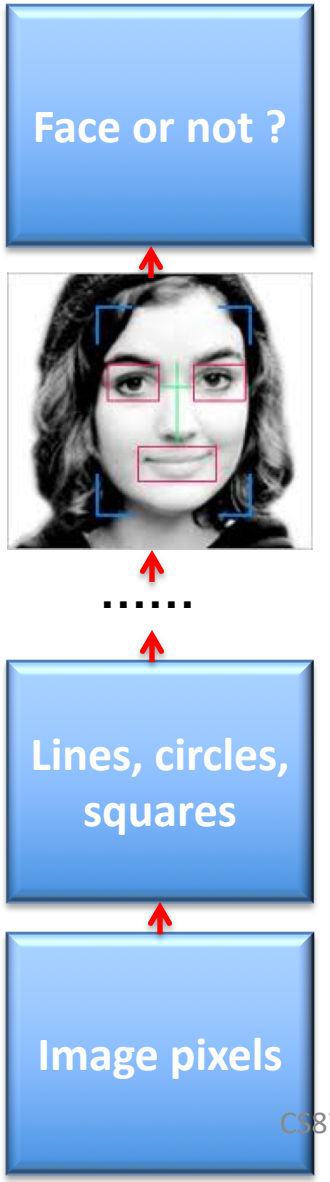
The final test set accuracy after running this code should be approximately 99.2%.

# Dilated Convolution

# Deep Belief Network – Learning Representation of Data First

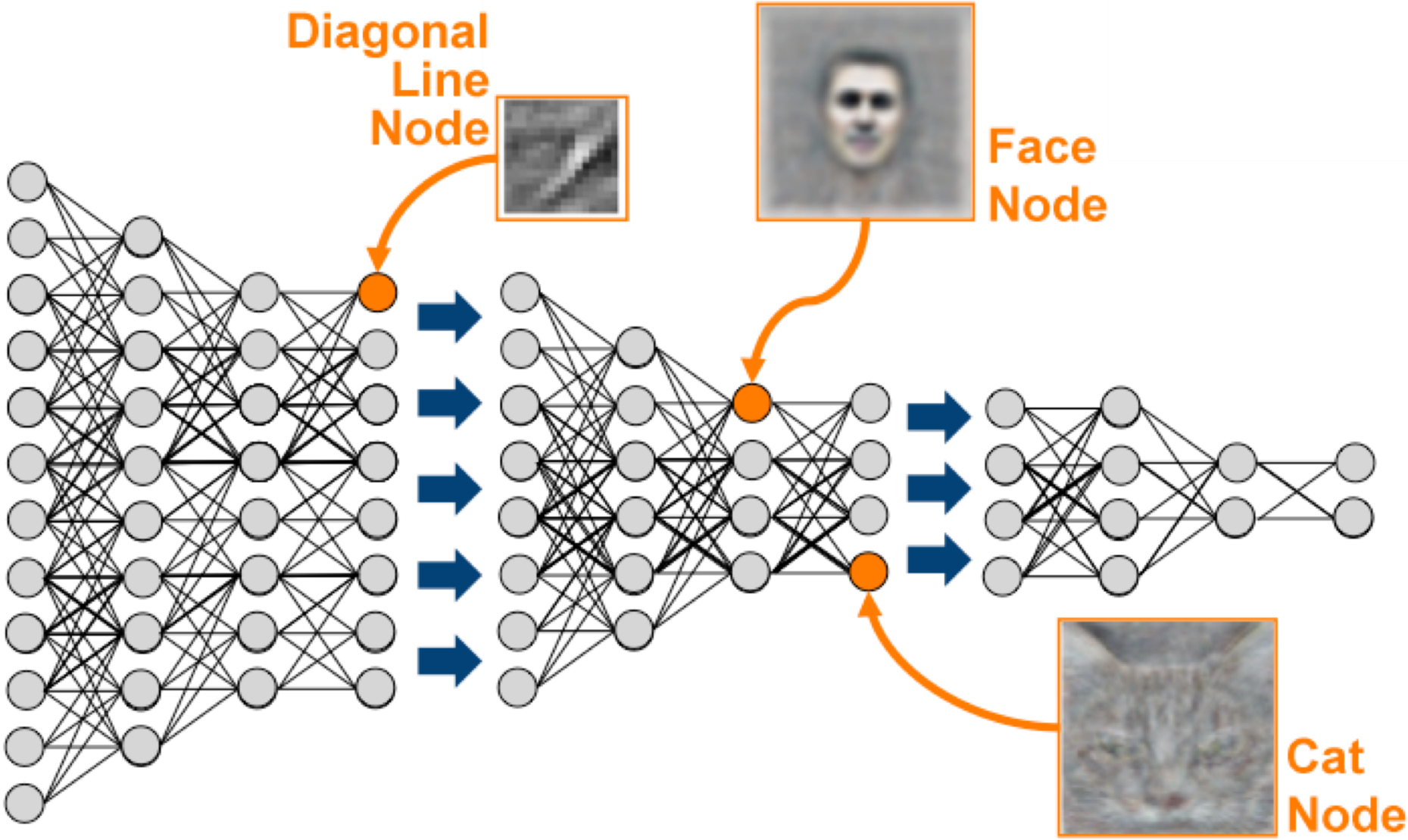

o
f(Σ)
f(Σ)
f(Σ)
…
f(Σ)
……
f(Σ)
f(Σ)
…
f(Σ)
f(Σ)
f(Σ)
…
f(Σ)
1
x1
x2
……
xd
Vanishing
Gradient

# Why Deep Learning? – A Face Recognition Analogy




**Brain Learning**

# Learning Representation First






http://theanalyticsstore.com/deep-learning/

# A Deep Learning Success

ROBERT MCMILLAN BUSINESS 03.13.13 6:30 AM

## GOOGLE HIRES BRAINS THAT HELPED SUPERCHARGE MACHINE LEARNING

Geoffrey Hinton (right) Alex Krizhevsky, and Ilya Sutskever (left) will do machine learning work at Google. *Photo: U of T*

# Energy Based Models

p(x) – probability of our data; data is represented by feature vector **x**.

$$p(x) = \frac{e^{-E(x)}}{Z}.$$

and

$$Z = \sum_x e^{-E(x)}$$

Attach an energy function (ie, *E*(x)) to score a configuration (ie, each possible input x).

We want desirable data to have low energy. Thus, tweak the parameters of *E*(x) accordingly.

*Restricted Boltzann Machines (RBM)*

# EBMs with Hidden Units

To increase power of EBMs, add hidden variables.

$$P(x) = \sum_h P(x,h) = \sum_h \frac{e^{-E(x,h)}}{Z}.$$

By using the notation,

Free energy

$$\mathcal{F}(x) = -\log \sum_h e^{-E(x,h)}$$

We can rewrite p(x) in a form similar to the standard EBM,

$$P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \text{ with } Z = \sum_x e^{-\mathcal{F}(x)}.$$

**log(P(x)) = -F(x) – log(Z)**

*Restricted Boltzmann Machines (RBM)*

# Tweakin' Parameters

Now we need to adjust the model so it reflects our data, do ML

- Likelihood fn

$$\mathrm{L}(\theta) = \Pi_{i=1}^{n} p(x_i; \theta)$$

- Avg. Log-likelihood fn

$$\ell(\theta) = \frac{1}{n}\log(\Pi_i p(x_i;\theta)) = \frac{1}{n}\sum_i \log(p(x_i;\theta))$$

$$= \frac{1}{n}\sum_i log\frac{e^{-F(x_i)}}{Z} = \frac{1}{n}\sum_i(-F(x_i) - \log(Z))$$

# Tweakin' Parameters

- Take the derivative

$$\frac{\partial \ell(\theta)}{\partial \theta_j} = \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j} - \frac{\partial \log Z}{\partial \theta_j}\right) = \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j} - \frac{1}{Z}\frac{\partial Z}{\partial \theta_j}\right)$$

$$= \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j} - \frac{1}{Z}\sum_{\hat{x}} e^{-F(\hat{x})}\frac{-\partial F(\hat{x})}{\partial \theta_j}\right)$$

$$= \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j}\right) - \sum_{\hat{x}} p(\hat{x})\frac{-\partial F(\hat{x})}{\partial \theta_j}$$

$$= \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j}\right) - E_p\left[\frac{-\partial F(x)}{\partial \theta_j}\right]$$

# Tweakin' Parameters

- Take the derivative

$$\frac{\partial \ell(\theta)}{\partial \theta_j} = \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j} - \frac{\partial \log Z}{\partial \theta_j}\right) = \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j} + \frac{1}{Z}\frac{\partial Z}{\partial \theta_j}\right)$$

$$= \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j} + \frac{1}{Z}\sum_{\hat{x}} e^{-F(\hat{x})}\frac{\partial F(\hat{x})}{\partial \theta_j}\right)$$

$$= \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j}\right) + \sum_{\hat{x}} p(\hat{x})\frac{\partial F(\hat{x})}{\partial \theta_j}$$

$$= \frac{1}{n}\sum_i \left(\frac{-\partial F(x_i)}{\partial \theta_j}\right) + E_p\left[\frac{\partial F(x)}{\partial \theta_j}\right]$$

Can think of as an expectation over dataset.

This is an expectation over all possible configurations of input x.

*Restricted Boltzann Machines (RBM)*

# Transition to RBM

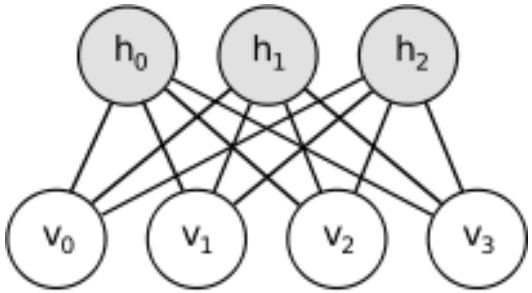
Looks like training a EBM is, in general, a tall task. But after much

Jump to an end result…

Restricted Boltzmann Machines (RBM)

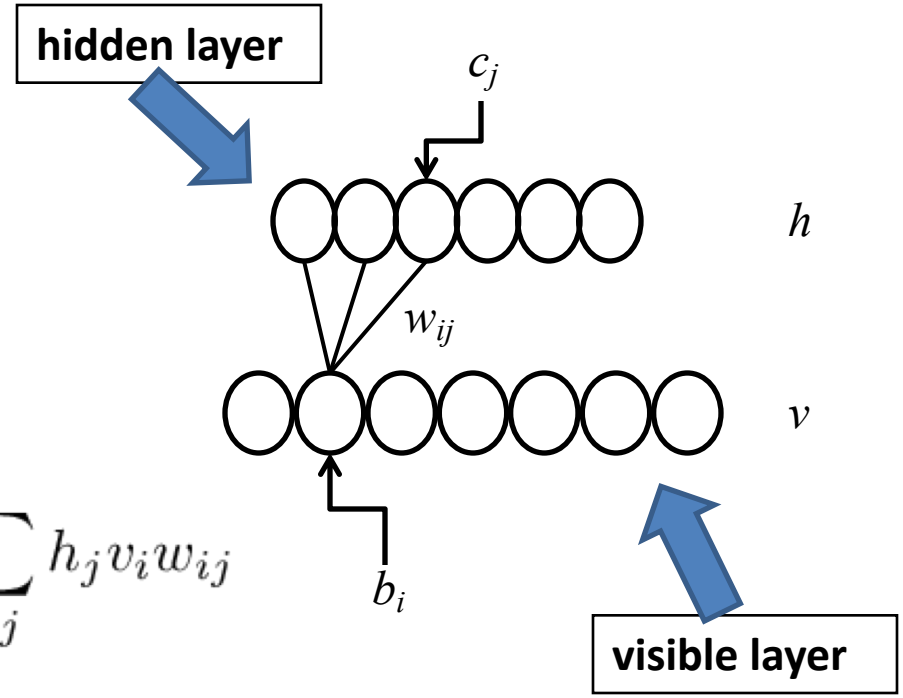
# RBMs

- Represented by a bipartite graph, with symmetric, weighted connections
- One layer has visible nodes and the other hidden (ie, latent) variables.
- Notes are often binary , <u>stochastic</u> units (ie, assume 0 or 1 based on probability)



# Unsupervised Restricted Boltzmann Machine (RBM)

- A model for a distribution over two layers of binary nodes
- Probability is defined via an "energy"




$$E(v,h) = -\sum_i b_i v_i - \sum_j c_j h_j - \sum_{i,j} h_j v_i w_{ij}$$

$$p(v,h) = \frac{e^{-E(v,h)}}{Z} \qquad Z = \sum_v \sum_h e^{-E(v,h)}$$

$$p(v) = \sum_h \frac{e^{-E(v,h)}}{Z}$$

# What's gained by "Restricted"

1) Conditional probabilities factor nicely

$P(h|v) = \Pi_i P(h_i|v)$ and $P(v|h) = \Pi_i P(v_i|h)$

2) Using binary units, we also can get

$$P(v_j = 1|h) = \sigma(b_j + W_j^{'} h)$$
$$P(h_i = 1|v) = \sigma(c_i + W_j v)$$

So we can get a sample of the visible or hidden nodes easily...

*Restricted Boltzann Machines (RBM)*

# Training a RBM – Maximum Likelihood

$$\boxed{l(\theta) = \frac{1}{n}\sum_i \log\left(\sum_h e^{-E(v_i,h)}\right) - \log Z}$$

$$p(v) = \sum_h \frac{e^{-E(v,h)}}{Z}$$

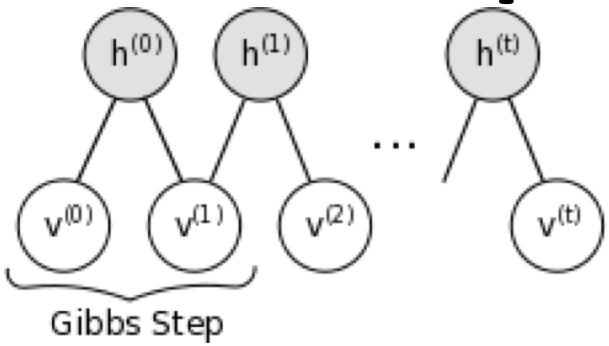
$$\boxed{\frac{\partial l(\theta)}{\partial \theta_j}} = \frac{1}{n}\sum_i \frac{\sum_h e^{-E(v_i,h)}}{\sum_h e^{-E(v_i,h)}} \frac{-\partial E}{\partial \theta_j} - \frac{1}{Z}\sum_v \sum_h e^{-E(v,h)} \frac{-\partial E}{\partial \theta_j}$$

$$= \frac{1}{n}\sum_i \sum_h \frac{p(v_i,h)}{p(v_i)} \left(\frac{-\partial E}{\partial \theta_j}\right) - E\left[\frac{-\partial E}{\partial \theta_j}\right]_{p^\infty}$$

$$= \frac{1}{n}\sum_i \sum_h p(h|v_i) \left(\frac{-\partial E}{\partial \theta_j}\right) - E\left[\frac{-\partial E}{\partial \theta_j}\right]_{p^\infty}$$

$$\boxed{= E\left[\frac{-\partial E}{\partial \theta_j}\right]_{p^0} - E\left[\frac{-\partial E}{\partial \theta_j}\right]_{p^\infty}}$$

# Gibbs Sampling




Can sample from p(v,h) by repeatedly sampling from v and h using the eqns. for p(v|h) and p(h|v).

As $t \rightarrow \infty$, $(v^{(t)}, h^{(t)})$ converge to samples of p(v,h).

But… hard to know when equilibrium has been reach, can be computationaly expensive

# Training a RBM - Contrastive Divergence based on Gibbs Sampling

Instead of attempting to sample from joint distribution p(v,h) (i.e. p∞), sample from $p^1$(v,h).

$$\Delta\theta_j \propto E\left[\frac{-\partial E}{\partial \theta_j}\right]_{p^0} - E\left[\frac{-\partial E}{\partial \theta_j}\right]_{p^\infty}$$

$$\Delta\theta_j \propto E\left[\frac{-\partial E}{\partial \theta_j}\right]_{p^0} - E\left[\frac{-\partial E}{\partial \theta_j}\right]_{p^1}$$

Hinton, *Neural Computation*(2002)

# Learning Rule

Recall energy function

$$E(v,h) = -\sum_i b_i v_i - \sum_j c_j h_j - \sum_{i,j} v_i h_j w_{i,j}$$

Calculating derivatives…

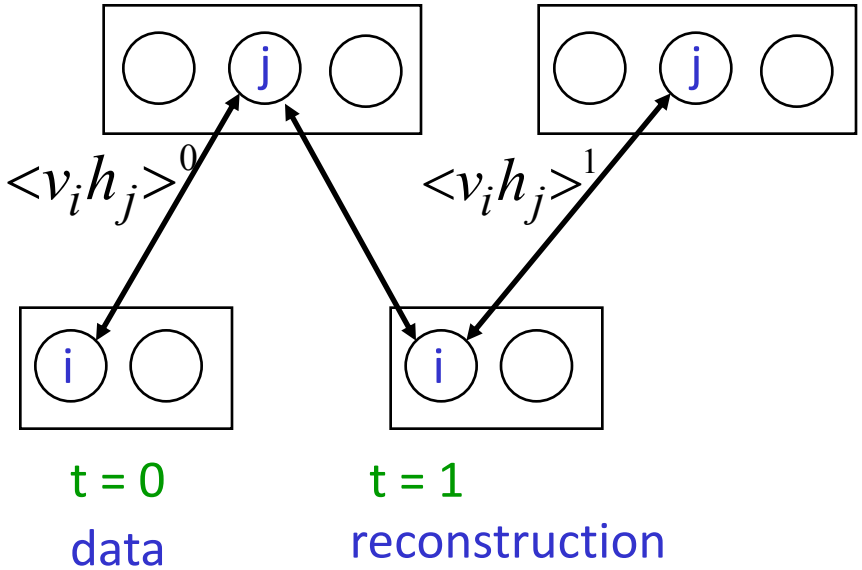
$$\frac{\partial E(v,h)}{\partial w_{i,j}} = v_i h_j \qquad \frac{\partial E(v,h)}{\partial b_i} = v_i$$

$$\frac{\partial E(v,h)}{\partial c_j} = h_j$$

So,

$$\Delta w_{i,j} \propto \epsilon(< v_i h_j >^0 - < v_i h_j >^\infty)$$

# A quick way to learn an RBM




Start with a training vector on the visible units.

Update all the hidden units in parallel

Update the all the visible units in parallel to get a "reconstruction".

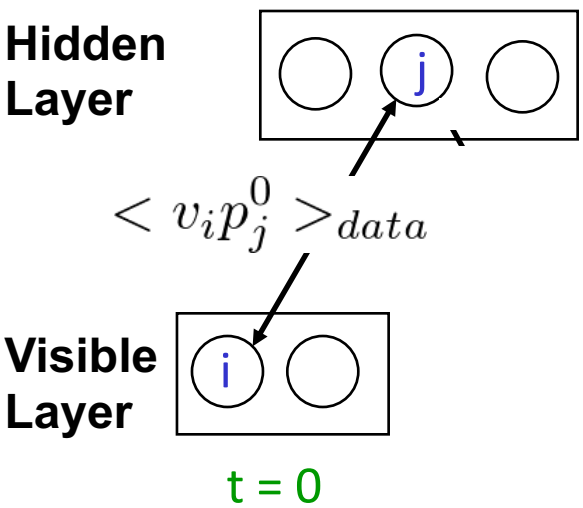
Update the hidden units again.

$$\Delta w_{ij} = \varepsilon \left( <v_i h_j>^0 - <v_i h_j>^1 \right)$$

**This is not following the gradient of the log likelihood**. But it works well. It is approximately following the gradient of another objective function (Carreira-Perpinan & Hinton, 2005).

# Training a RBM via Contrastive Divergence

**Gradient of the likelihood with respect to $w_{ij}$ ≈ the difference between interaction of $v_i$ and $h_j$ at time 0 and at time 1.**

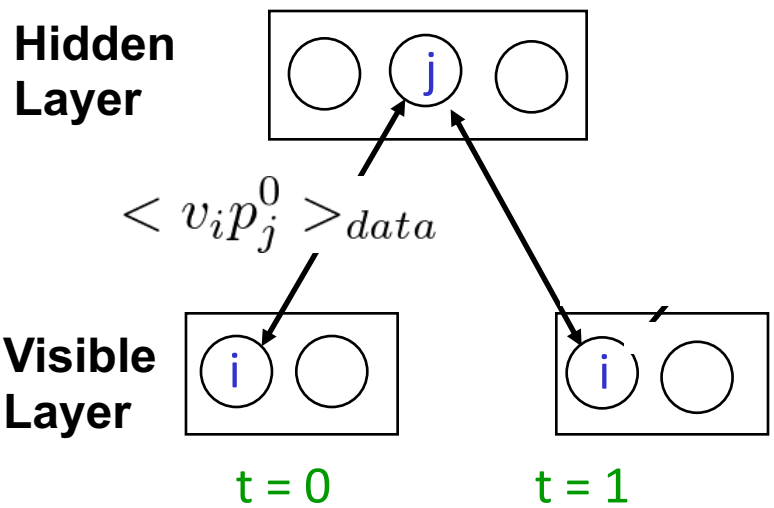



$$p_j^{(0)} = \sigma(\sum_i v_i w_{ij} + c_j)$$

Hinton, *Neural Computation*(2002)

# Training a RBM via Contrastive Divergence

**Gradient of the likelihood with respect to $w_{ij}$ ≈ the difference between interaction of $v_i$ and $h_j$ at time 0 and at time 1.**



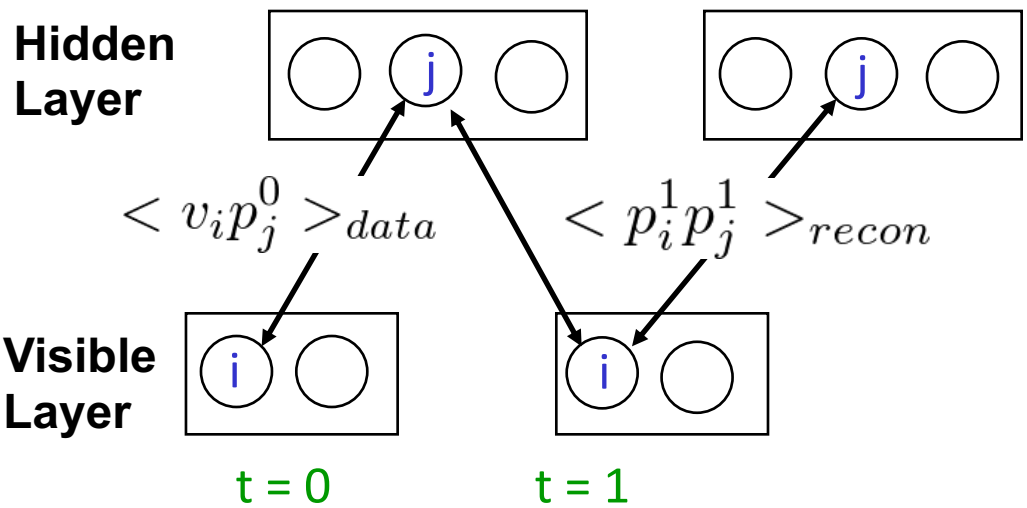

$$p_j^{(0)} = \sigma(\sum_i v_i w_{ij} + c_j)$$

$$p_i^{(1)} = \sigma(\sum_j h_j w_{ij} + b_i)$$

Hinton, *Neural Computation*(2002)

# Training a RBM via Contrastive Divergence

**Gradient of the likelihood with respect to $w_{ij}$ ≈ the difference between interaction of $v_i$ and $h_j$ at time 0 and at time 1.**




$$p_j^{(0)} = \sigma(\sum_i v_i w_{ij} + c_j)$$

$$p_i^{(1)} = \sigma(\sum_j h_j w_{ij} + b_i)$$

$$p_j^{(1)} = \sigma(\sum_i p_i^1 w_{ij} + c_j)$$

**σ: sigmoid function**

$$\Delta w_{i,j} = \langle v_i p_j^0 \rangle - \langle p_i^1 p_j^1 \rangle$$

Hinton, *Neural Computation*(2002)

# Challenges with RBMs

A number of choices to be made

- Types of nodes, learning weight, initial values, batch sizes, etc.
- Care should be taken to avoid over-fitting

A RBM "manual" is available on line…

http://www.cs.utoronto.ca/~hinton/absps/guideTR.pdf

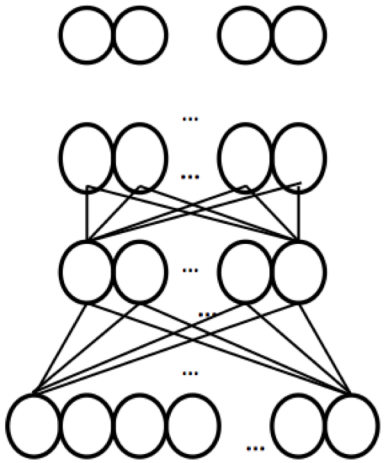
**Software package: Pylearn2:**
**http://deeplearning.net/software/pylearn2/**

**On both GPU and CPU**

# GPU Implementation

Calculations need for training and classification made use of CUDAMat and GPUs

Train with over one million parameters in about an hour



# Why ???

Okay, we can model p(x).

But how to…

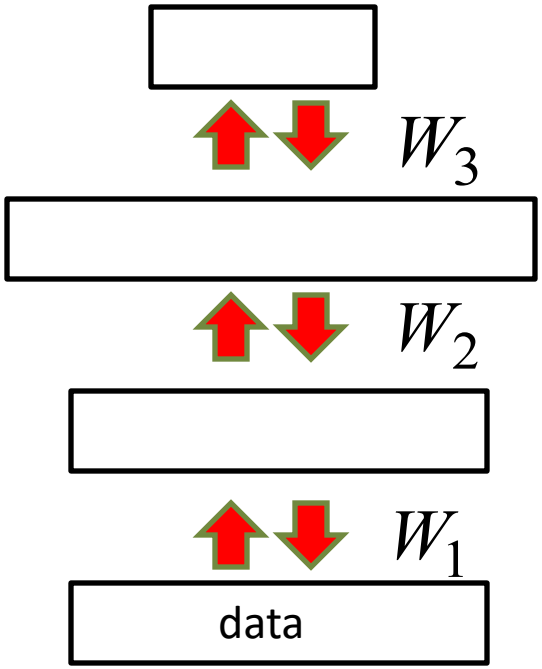
1. **Find p(label|x). We want a classifier!**
2. Improve the model for p(x).

# Deep Belief Nets

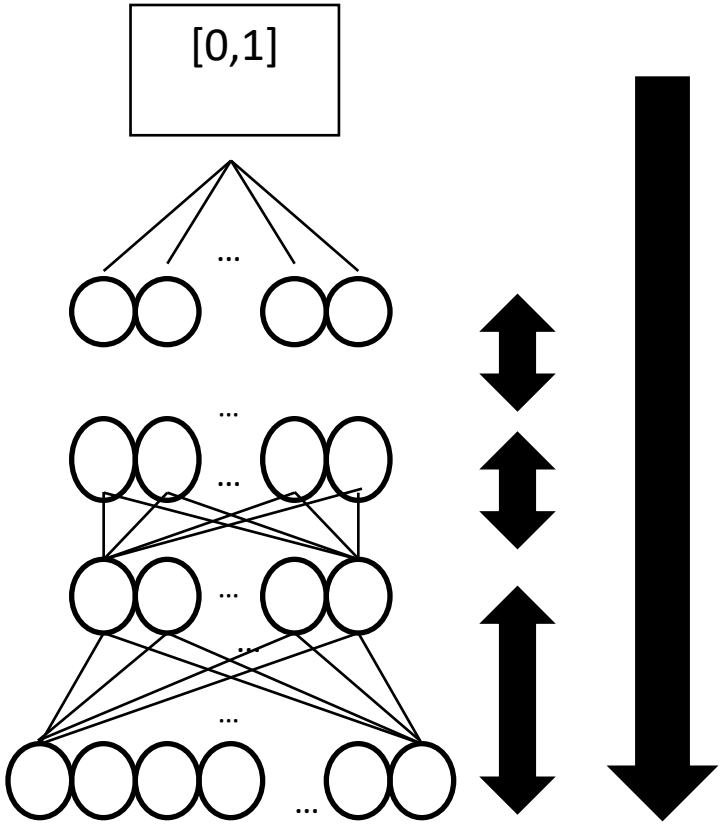
RBMs are typically used in stack

- Train them up one layer at a time
- Hidden units become visible units to the next layer up

**If your goal is a discriminator, you train a classifier on the top level representation of your input.**



# Training a Deep Network




1. **Weights are learned layer by layer via <u>unsupervised learning</u>.**
2. **Final layer is learned as a <u>supervised neural network</u>.**
3. **All weights are fine-tuned using <u>supervised back propagation</u>.**

Hinton and Salakhutdinov, *Science,* 2006

# Why stack them up? Why does this work?

This is a good question, with a long complicated answer.

Basically, doing so can improve a lower variation bound on the probability of training data under the model.

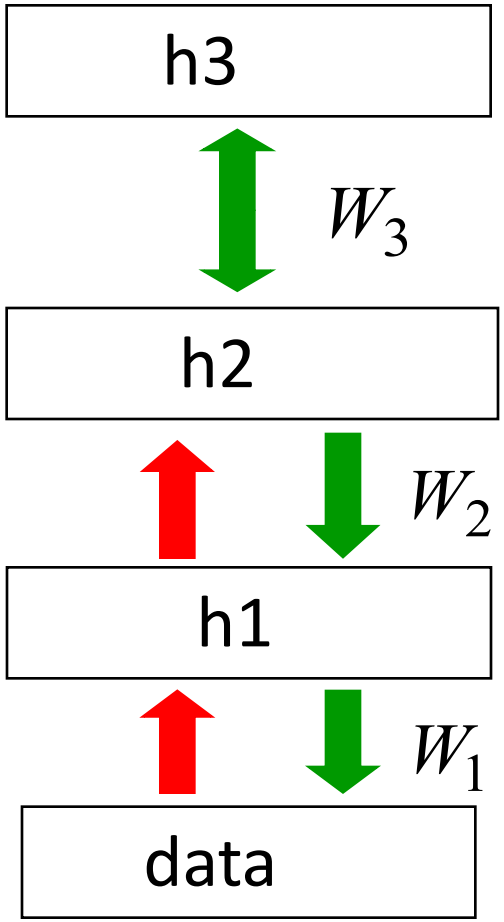
Hinton, Osindero, & The, 2006

# How to generate from the model

Bonus when modeling p(x), we can see what the model believes in

- To generate data:
  - Get an equilibrium sample from the top-level RBM by performing alternating Gibbs sampling for a long time.
  - Perform a top-down pass to get states for all the other layers.

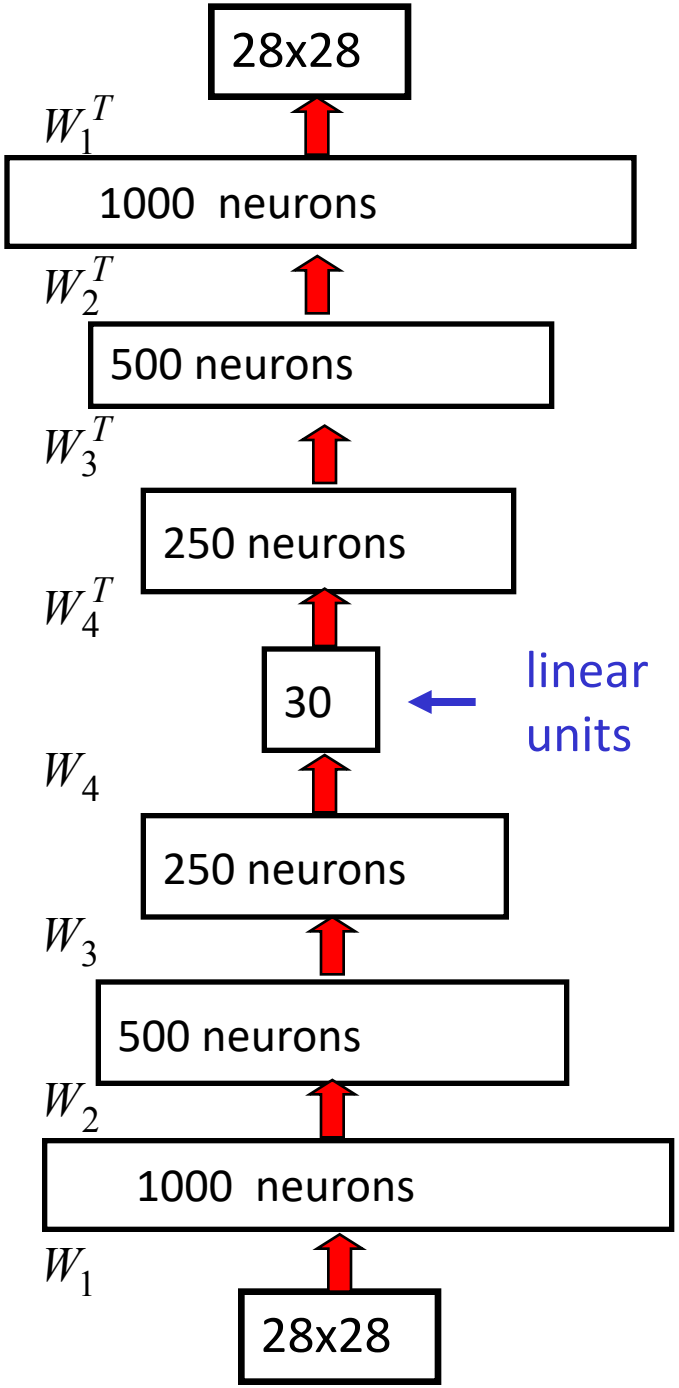
So the lower level bottom-up connections are not part of the generative model. They are just used for inference.




Slide modified from Hinton, 2007

# Deep Autoencoders

- They always looked like a really nice way to do non-linear dimensionality reduction:
  - But it is very difficult to optimize deep autoencoders using backpropagation.
- We now have a much better way to optimize them:
  - First train a stack of 4 RBM's
  - Then "unroll" them.
  - Then fine-tune with backprop.




Hinton & Salakhutdinov, 2006; slide form Hinton UCL tutorial

# Some Applications

We will look at two applications done by Hinton's Lab

- A model for digit recognition
- Cluster/search documents

# Applications: A model of digit recognition

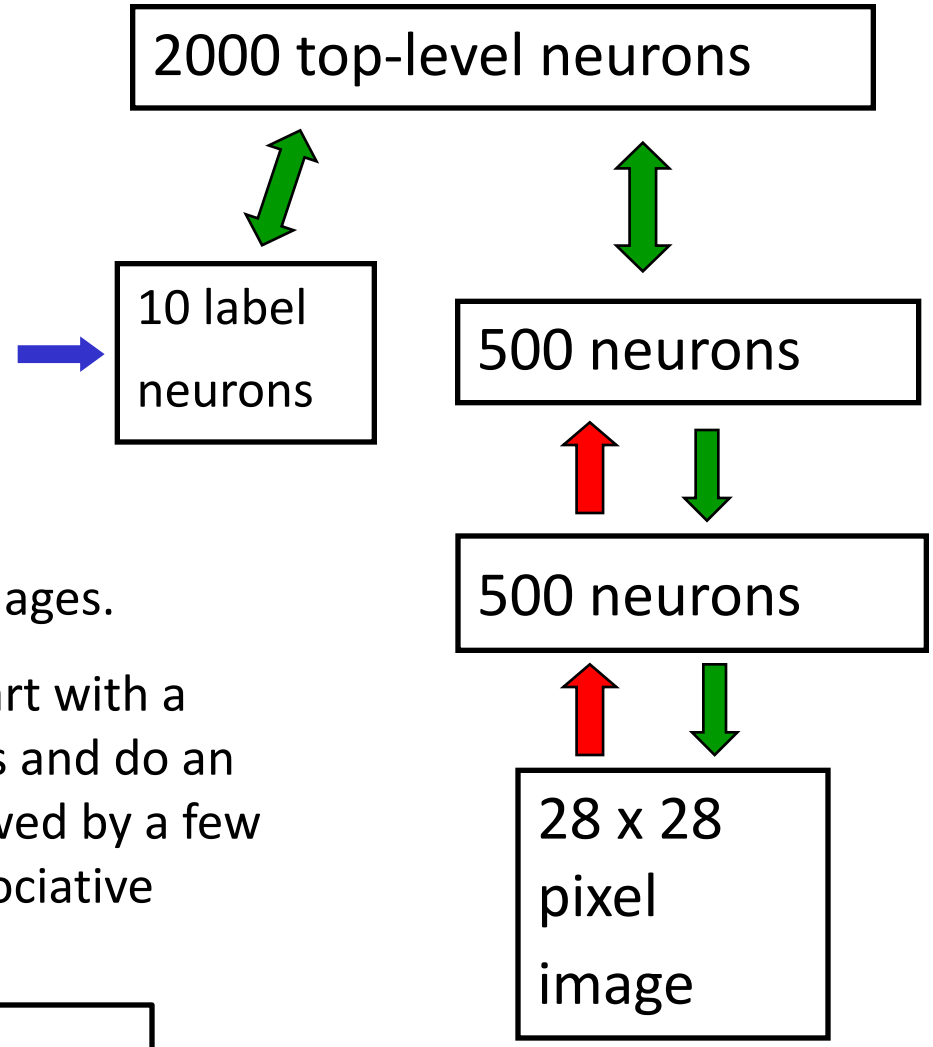
- Classify digits (0 – 9)
- Input is a 28x28 image from MNIST (training 60k, test 10k examples)

# Applications: A model of digit recognition

This is work from Hinton et al., 2006

The top two layers form an associative memory whose energy landscape models the low dimensional manifolds of the digits.

The energy valleys have names




The model learns to generate combinations of labels and images.

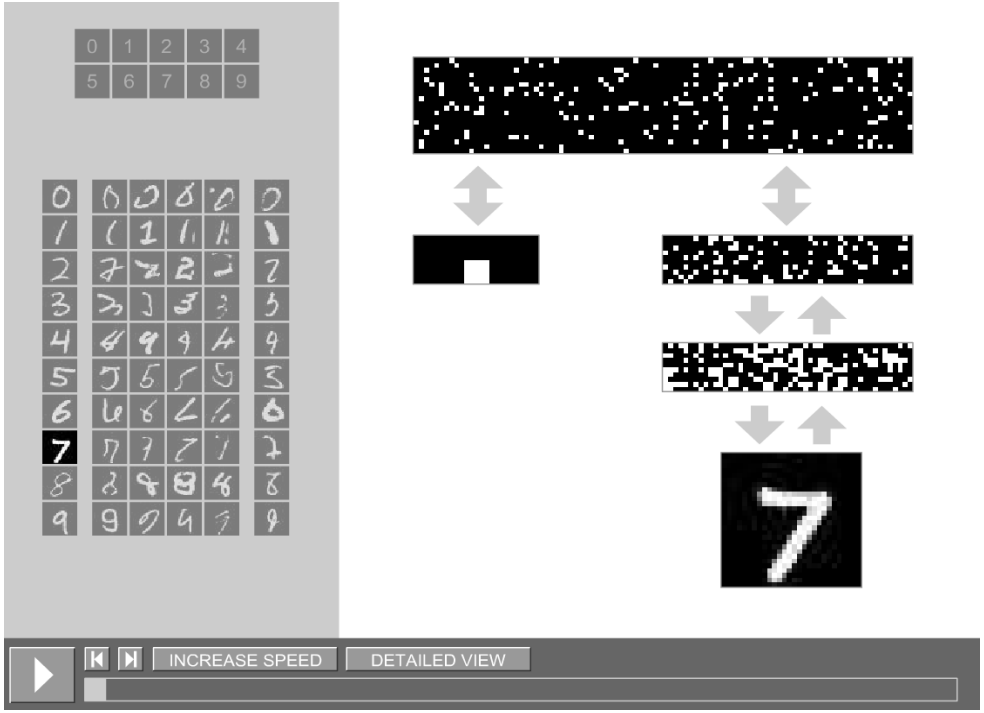
To perform recognition we start with a neutral state of the label units and do an up-pass from the image followed by a few iterations of the top-level associative memory.

Matlab/Octave code available at http://www.cs.utoronto.ca/~hinton/

Slide modified from Hinton, 2007

# Model in action

Hinton has provided an excellent way to view the model in action…




http://www.cs.toronto.edu/~hinton/digits.html

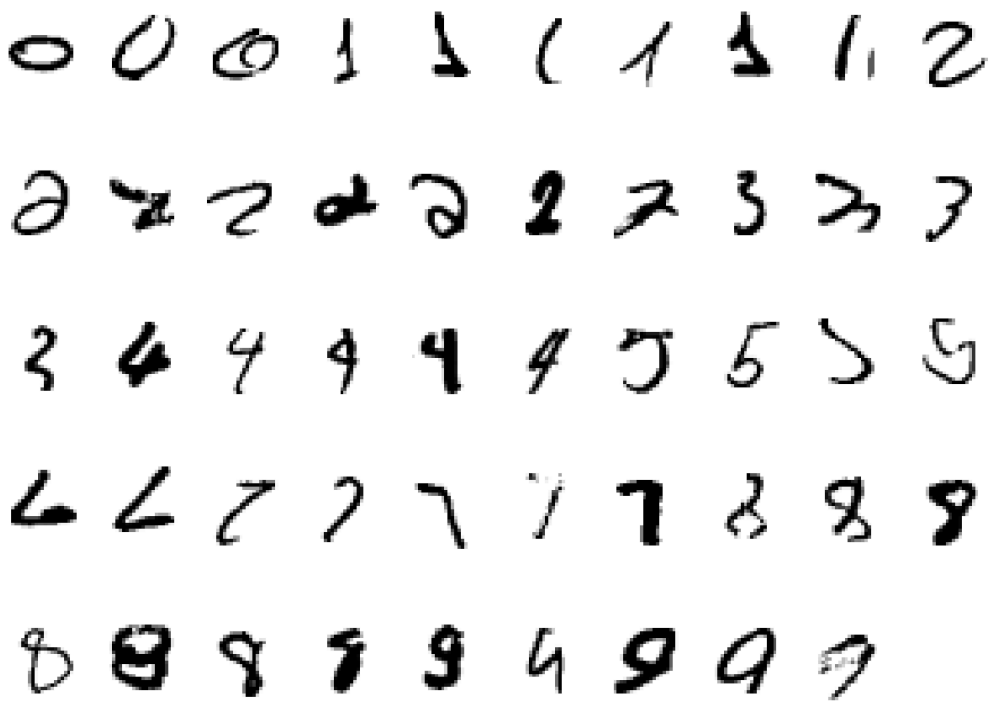
# More Digits

Samples generated by letting the associative memory run with one label clamped. There are 1000 iterations of alternating Gibbs sampling between samples.

Slide from Hinton, 2007

# Even More Digits

Examples of correctly recognized handwritten digits that the neural network had never seen before



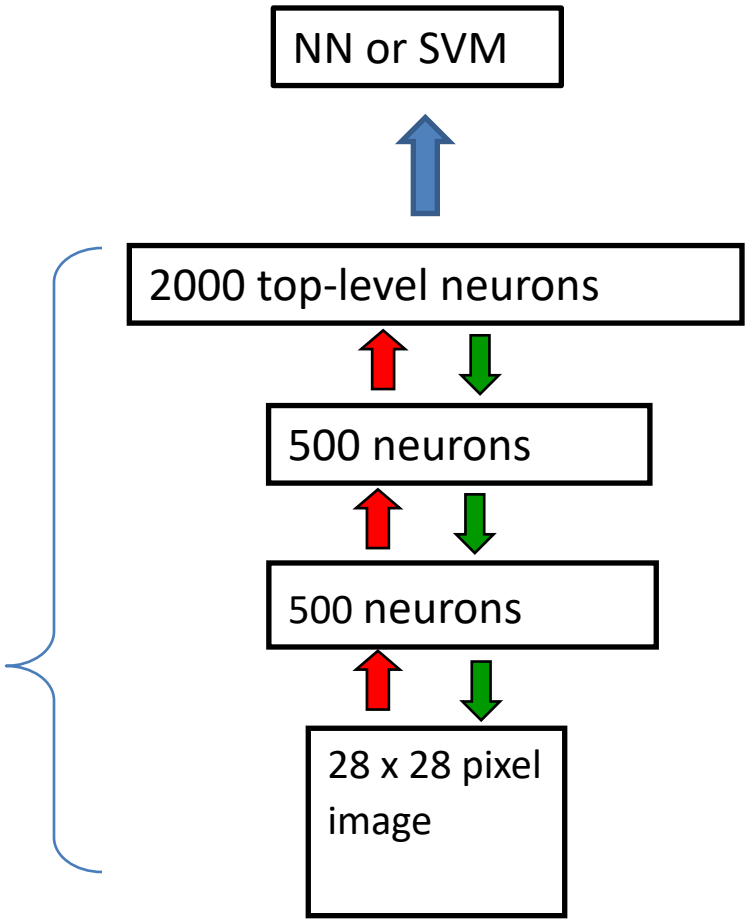
Slide from Hinton, 2007

# Extensions

Do classification.

One way (probably no the best), train generative model with labeled/unlabeled data

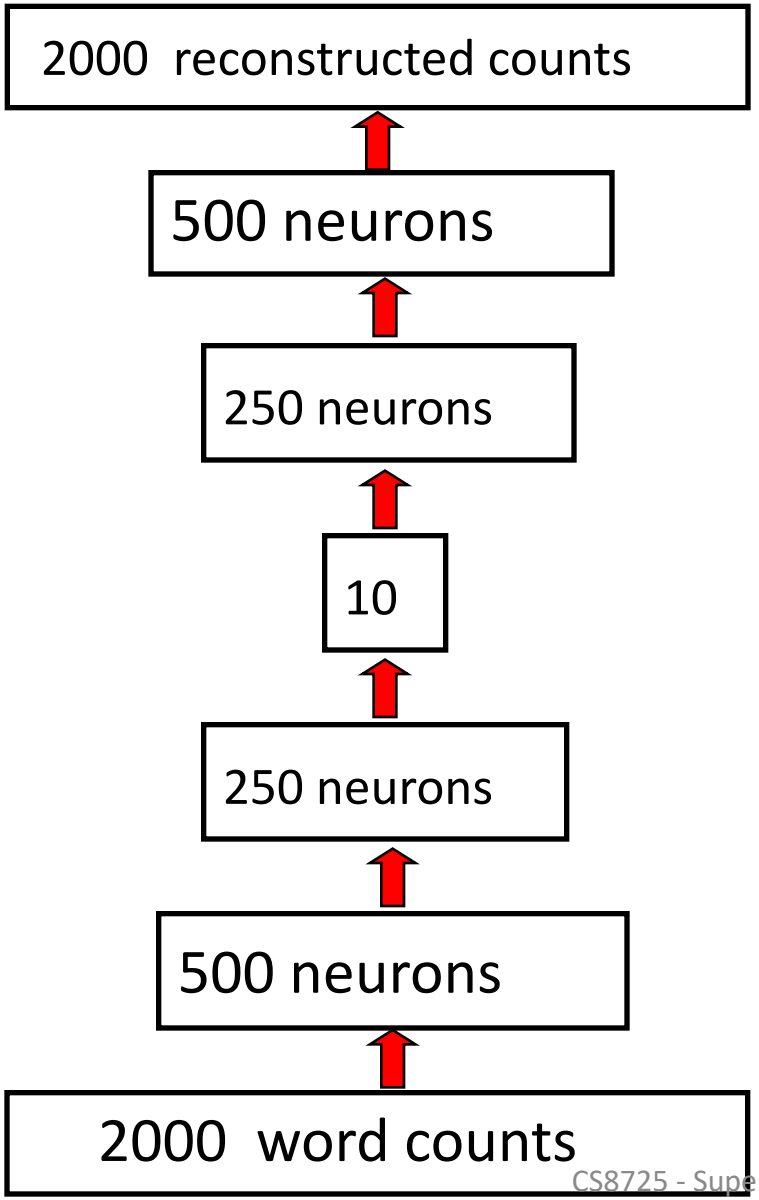
Then train a NN on higher dimensional representation.



# Applications: Classifying text documents

- A document can be characterized by the frequency of words that appear (ie, word counts for some dictionary become feature vector)
- Goals…
  1. Group/cluster similar documents
  2. Find similar documents

# How to compress the count vector

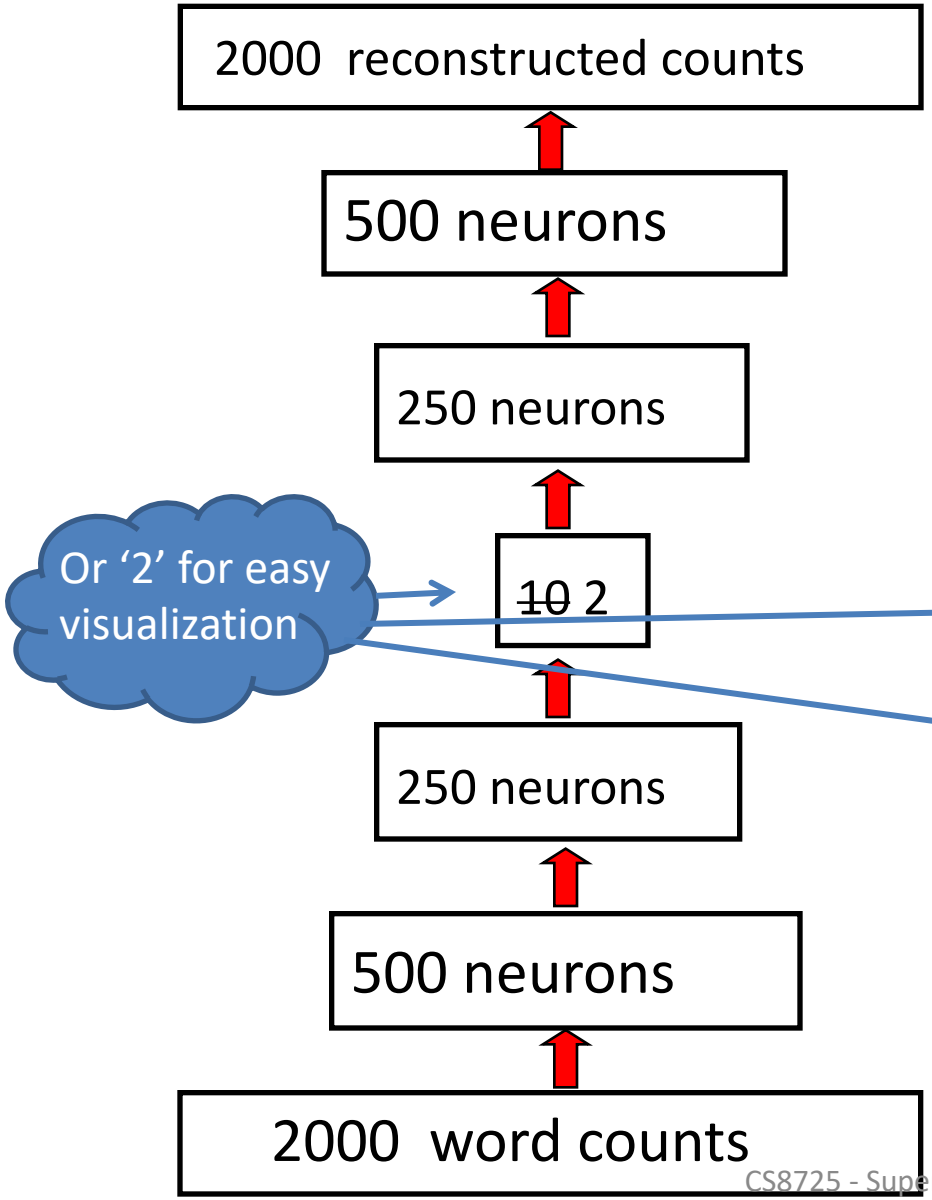



## Multi-layer auto-encoder

- Train a model to reproduce its input vector as its output
- This setup forces as much information as possible be compressed and passed thru the 10 numbers in the central bottleneck.
- These 10 numbers are then a good way to compare documents.

Slide modified from Hinton, 2007

# How to compress the count vector




## Multi-layer auto-encoder

- Train a model to reproduce its input vector as its output
- This setup forces as much information as possible be compressed and passed thru the ~~10~~ 2 numbers in the central bottleneck.
- These ~~10~~ 2 numbers are then a good way to compare documents.

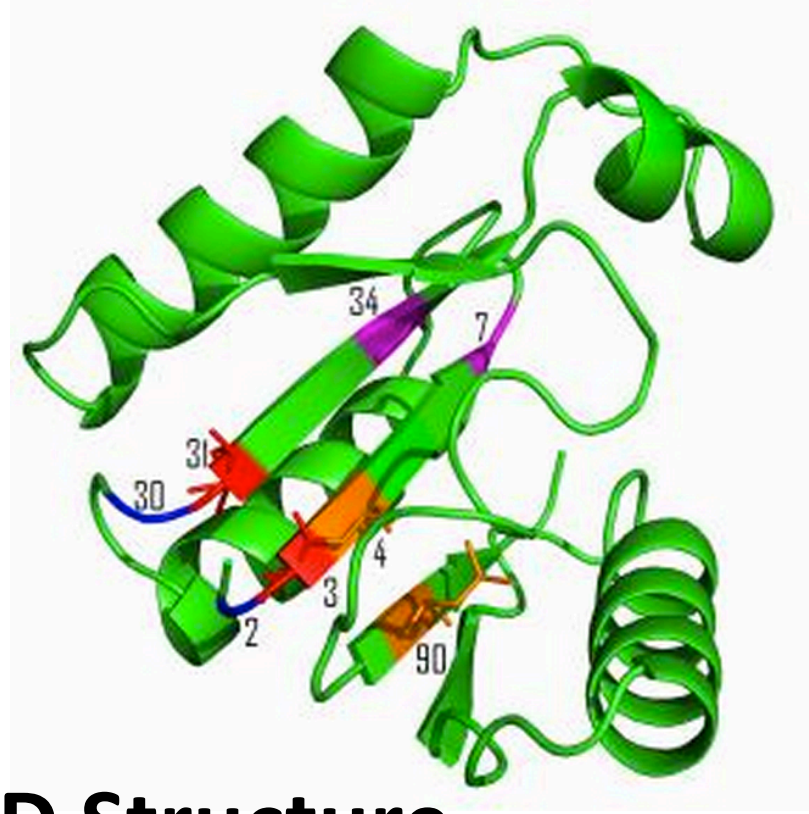
Slide modified from Hinton, 2007

# Residue-Residue Contact Prediction

## 1D Sequence

*i* *j*

**SDDEVYQYIVSQVKQYGIEPAELLSRKYGDKAKYHLSQRW**



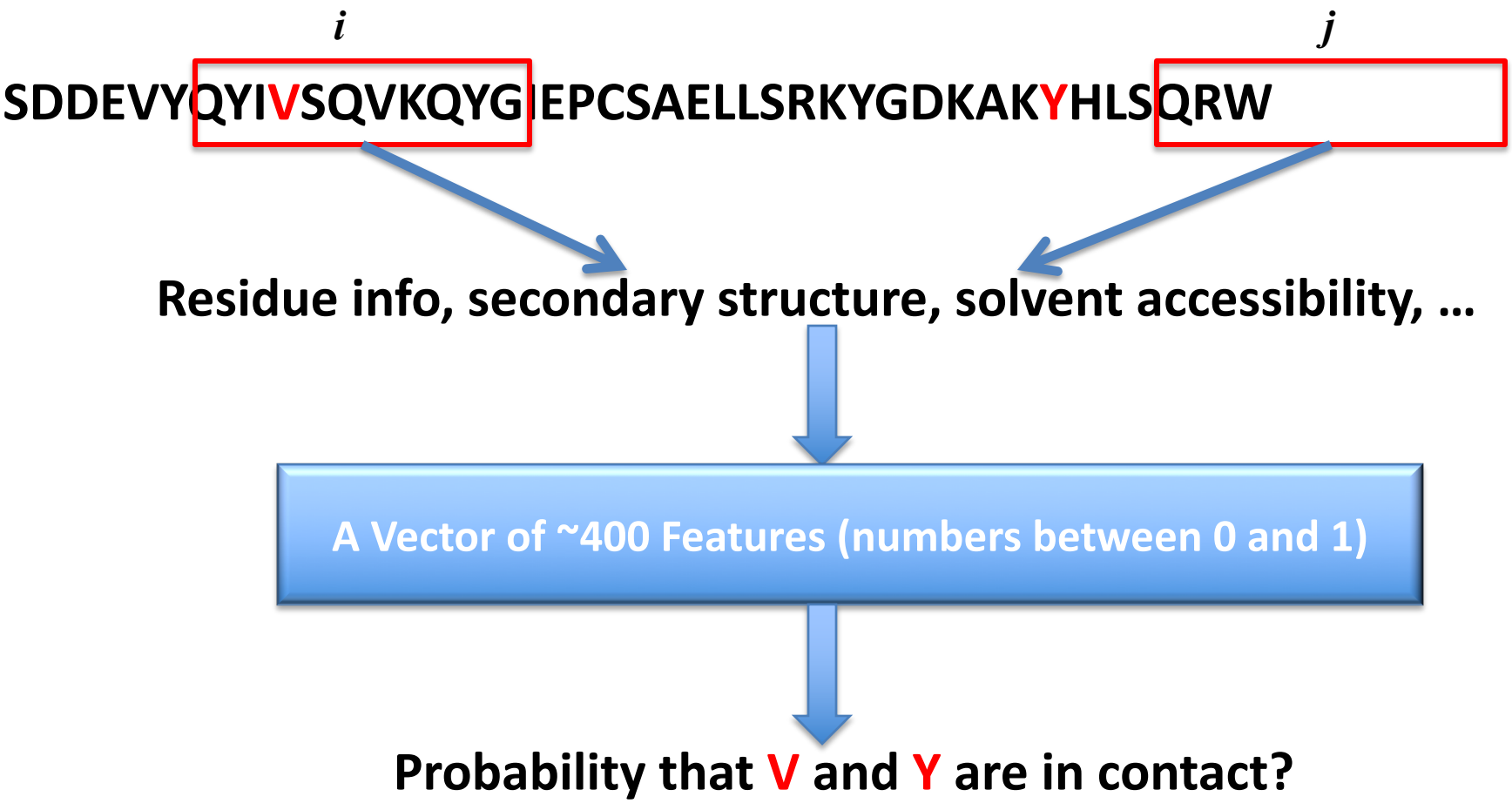

## 3D Structure

## Objective:

**Predict if two residues (*i*, *j*) are in contact (spatially close), i.e. distance(i, j) < 8 Angstrom**

Eickholt & Cheng, 2012

# A Binary Classification Problem



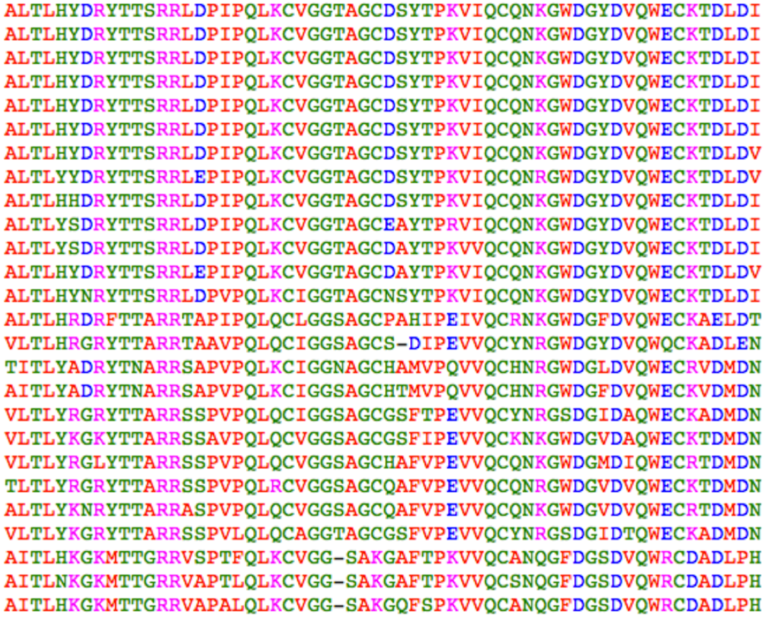

Cheng & Baldi, 2007; Tegge et al., 2009; Eickholt & Cheng, 2012

# Input Features

*i* *j*

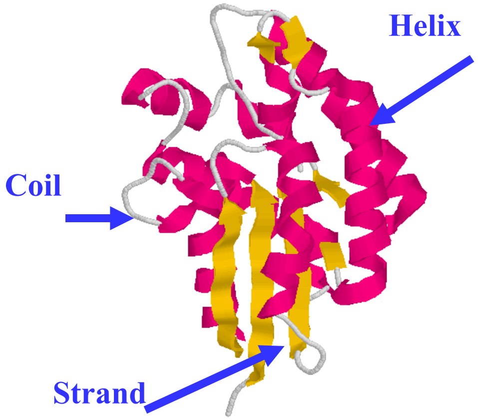
**ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQRCQN**

**20 numbers for residue profile**



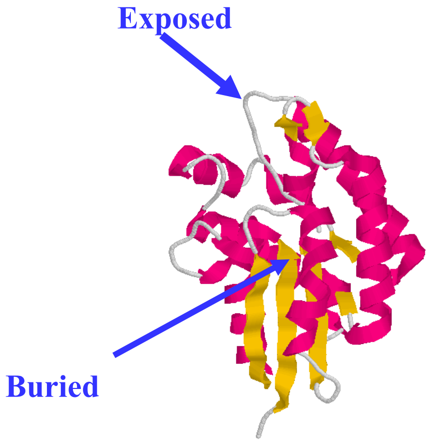
```
ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI
ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI
ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI
ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI
ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI
ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI
ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDV
ALTLYYDRYTTSRRLEPIPQLKCVGGTAGCDSYTPKVIQCQNRGWDGYDVQWECKTDLDV
ALTLHHDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI
ALTLYSDRYTTSRRLDPIPQLKCVGGTAGCEAYTPRVIQCQNKGWDGYDVQWECKTDLDI
ALTLYSDRYTTSRRLDPIPQLKCVGGTAGCDAYTPKVVQCQNKGWDGYDVQWECKTDLDI
ALTLHYDRYTTSRRLEPIPQLKCVGGTAGCDAYTPKVIQCQNKGWDGYDVQWECKTDLDV
ALTLHYNRYTTSRRLDPVPQLKCIGGTAGCNSYTPKVIQCQNKGWDGYDVQWECKTDLDI
ALTLHRDRFTTARRTAPIPQLQCLGGSAGCPAHIPEIVQCRNKGWDGFDVQWECKAELDT
VLTLHRGRYTTARRTAAVPQLQCIGGSAGCS-DIPEVVQCYNRGWDGYDVQWQCKADLEN
TITLYADRYTNARRSAPVPQLKCIGGNAGCHAMVPQVVQCHNRGWDGLDVQWECRVDMDN
AITLYADRYTNARRSAPVPQLKCIGGSAGCHTMVPQVVQCHNRGWDGFDVQWECKVDMDN
VLTLYRGRYTTARRSSPVPQLQCIGGSAGCGSFTPEVVQCYNRGSDGIDAQWECKADMDN
VLTLYKGKYTTARRSSAVPQLQCVGGSAGCGSFIPEVVQCKNKGWDGVDAQWECKTDMDN
VLTLYRGLYTTARRSSPVPQLQCVGGSAGCHAFVPEVVQCQNKGWDGMDIQWECRTDMDN
TLTLYRGRYTTARRSSPVPQLRCVGGSAGCQAFVPEVVQCQNRGWDGVDVQWECKTDMDN
ALTLYKNRYTTARRASPVPQLQCVGGSAGCQAFVPEVVQCQNKGWDGVDVQWECRTDMDN
VLTLYKGRYTTARRSSPVLQLQCAGGTAGCGSFVPEVVQCYNRGSDGIDTQWECKADMDN
AITLHKGKMTTGRRVSPTFQLKCVGG-SAKGAFTPKVVQCANQGFDGSDVQWRCDADLPH
AITLNKGKMTTGRRVAPTLQLKCVGG-SAKGAFTPKVVQCSNQGFDGSDVQWRCDADLPH
AITLHKGKMTTGRRVAPALQLKCVGG-SAKGQFSPKVVQCANQGFDGSDVQWRCDADLPH
```

**3 numbers**



| | |
|---|---|
| **Helix** | **100** |
| **Strand** | **010** |
| **Coil** | **001** |

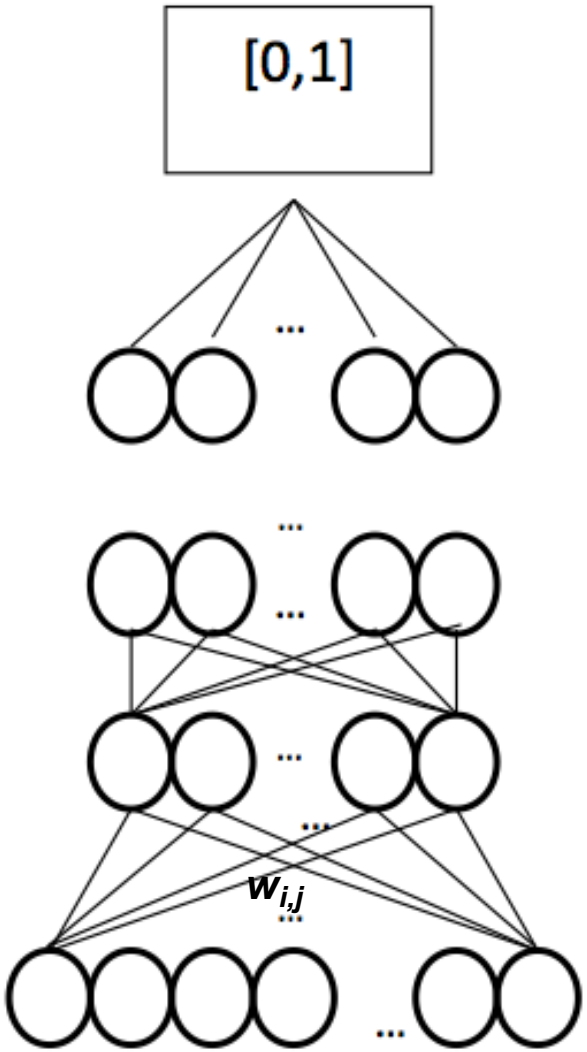
**2 numbers**



| | |
|---|---|
| **Exposed** | **10** |
| **Buried** | **01** |

**25 * 18 = 400 features for a pair (*i*, *j*)**

# Deep Belief Network Architecture




~350 nodes

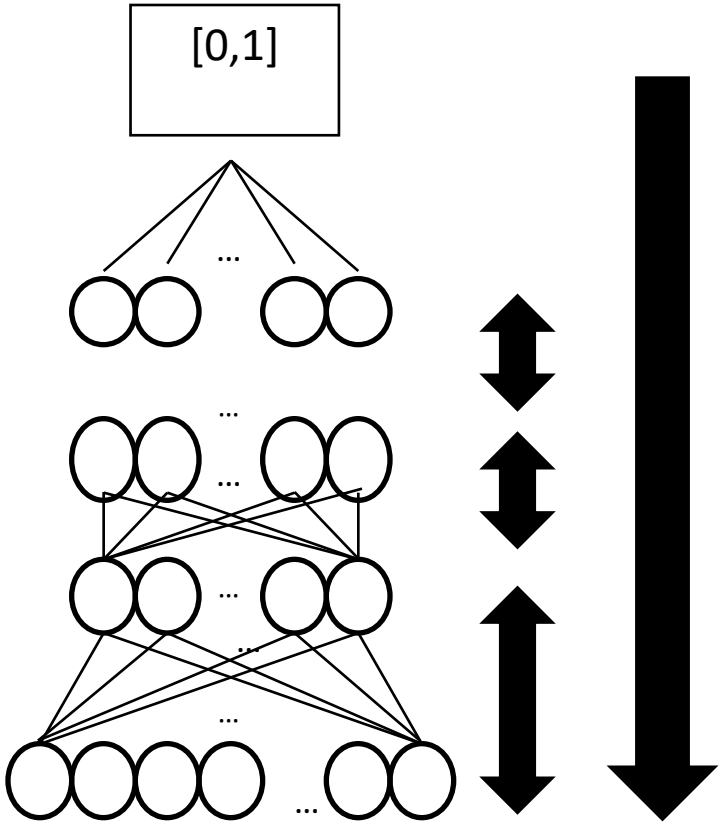
~500 nodes

~500 nodes

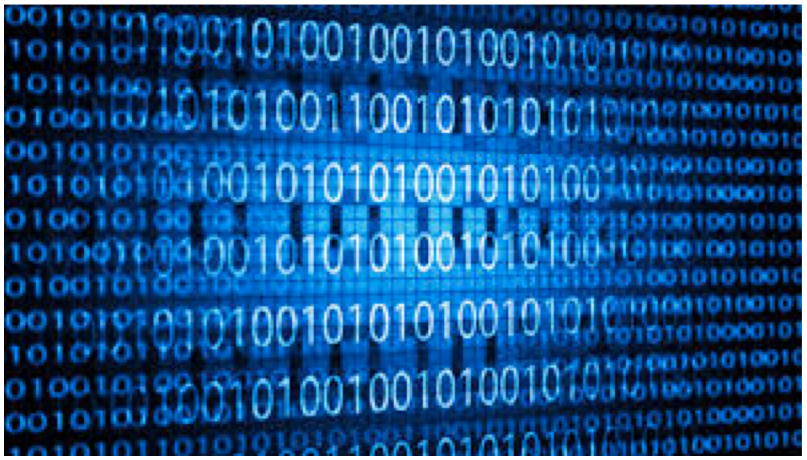
~400 input nodes

A Vector of ~400 Features (numbers between 0 and 1)

# Training a Deep Network



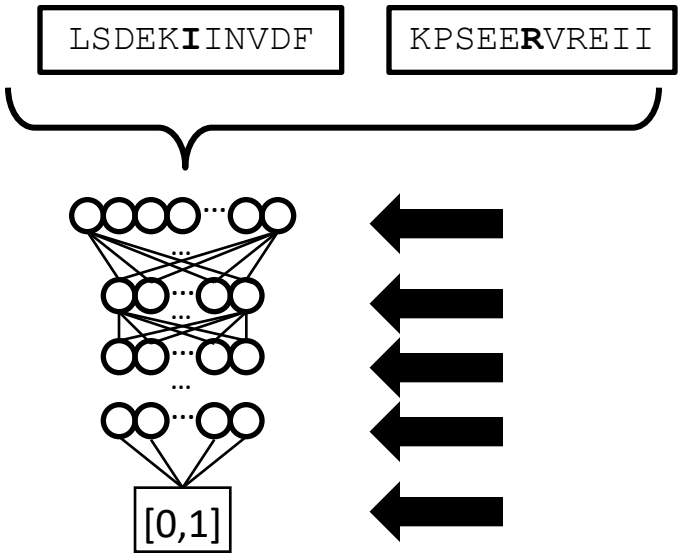

**1239 Proteins for Training Residue Pairs; Millions of Residue Pairs**



# GPU Implementation

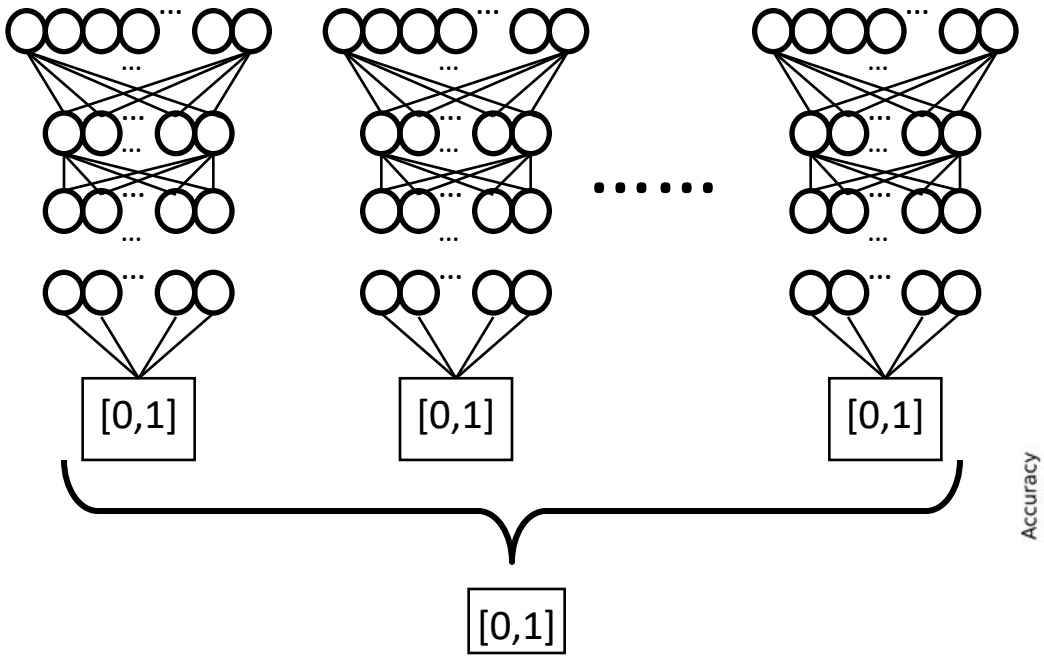
**Parallelize training of deep learning network with GPUs and CUDAMat**

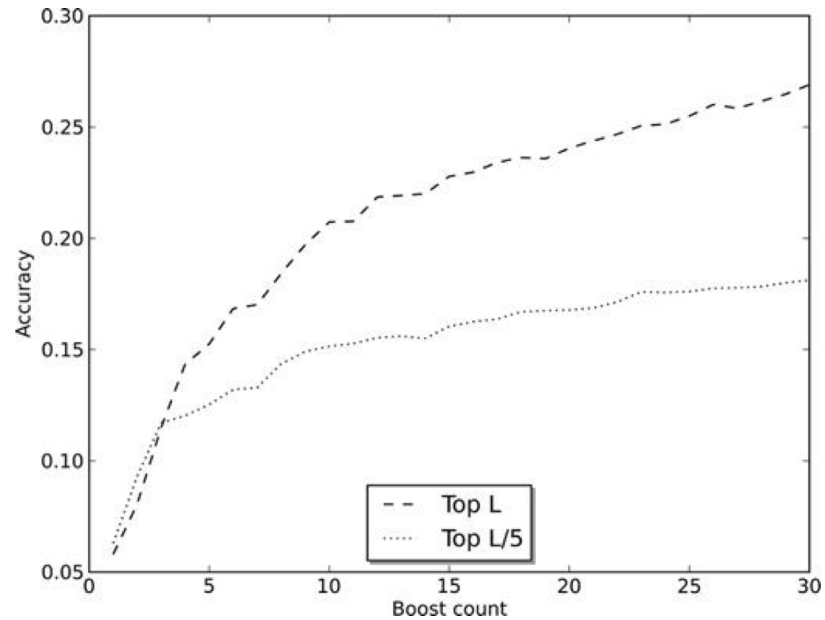
**Train DNs with over 1M parameters in about an hour**




Eickholt & Cheng, 2012

# Boosted Ensembles for Contact Prediction




**Final output of ensemble is the weighted sum of individual DN outputs.**

Eickholt and Cheng, *Bioinformatics* (2012)

# Results on Test Data Set (196 Proteins) and CASP

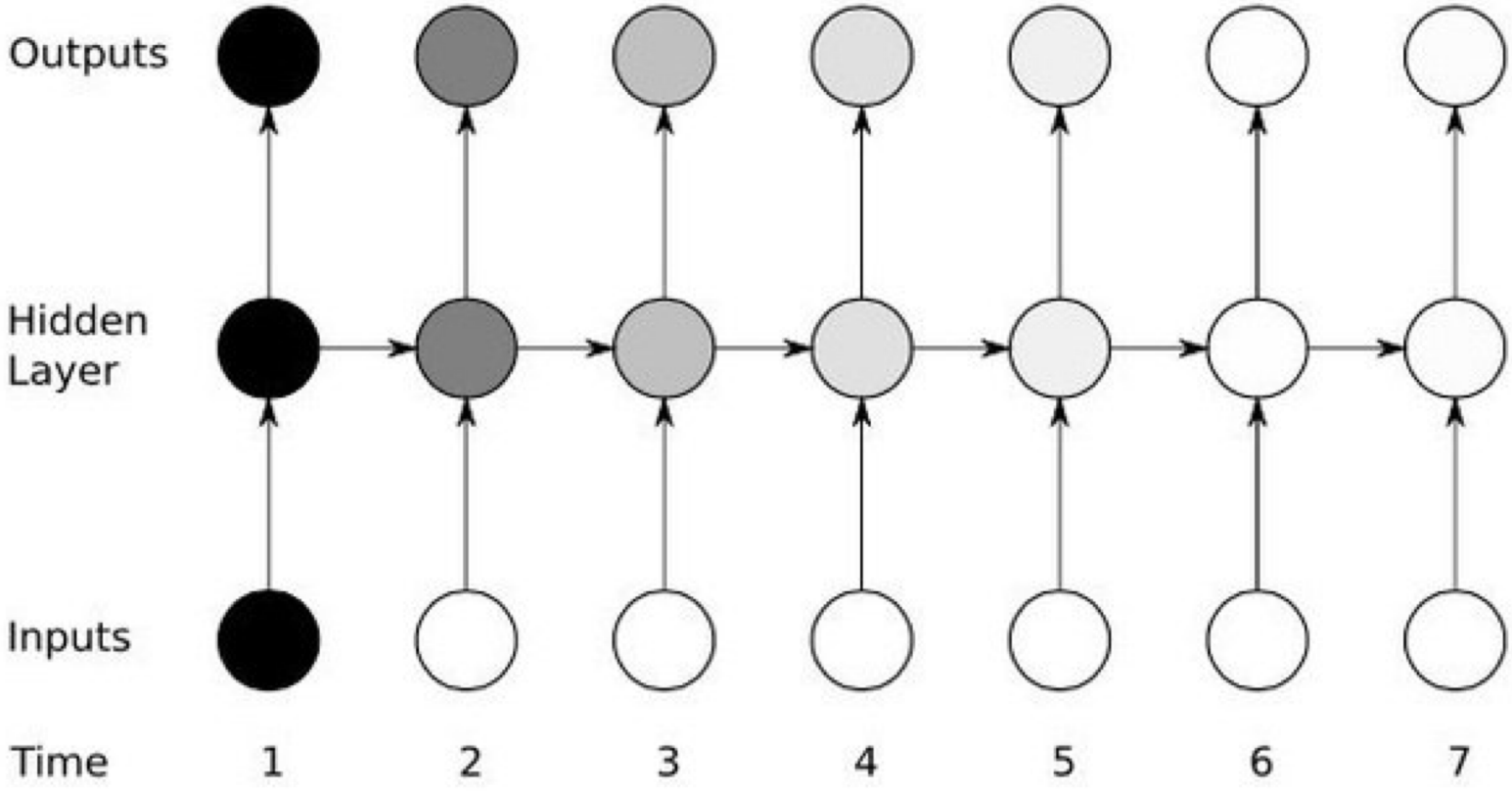
| Metric | Acc. L/5 | Acc. L/5 (one shift) |
|---|---|---|
| Short Range (6 <= \|i-j\| <12) | 0.51 | 0.79 |
| Medium Range (12 <= \|i-j\| <24) | 0.38 | 0.65 |
| Long Range (\|i-j\| >= 24) | 0.34 | 0.55 |

2012 2014 2016

2017

# Deep Recurrent Neural Network

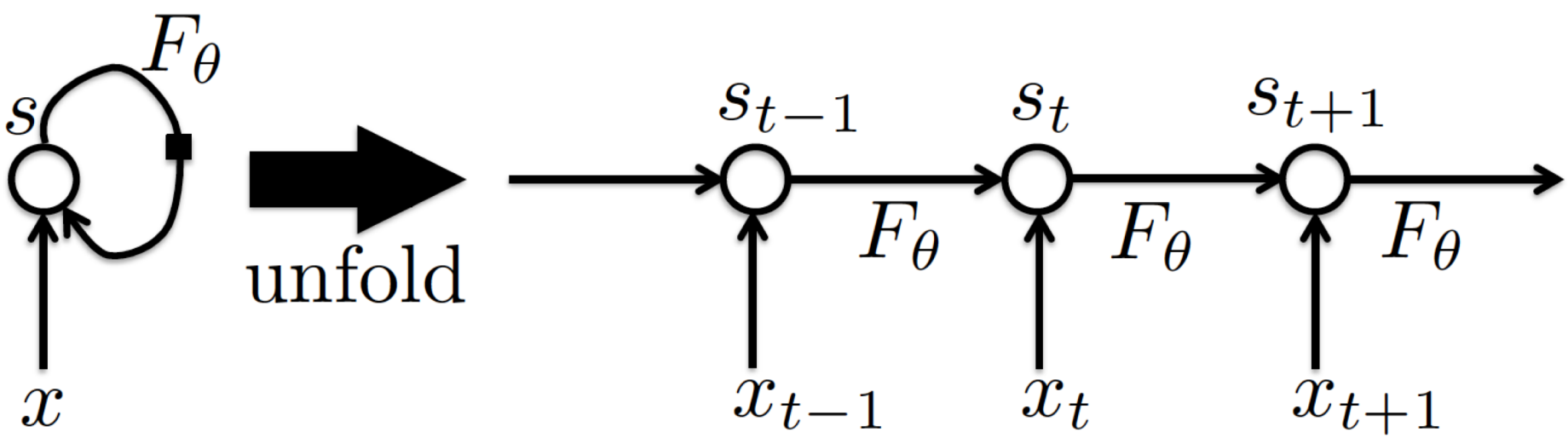
# Temporal and Spatial Series Problem



# Recurrent Neural Network

- Selectively summarize an input sequence in a fixed-size state vector via a recursive update

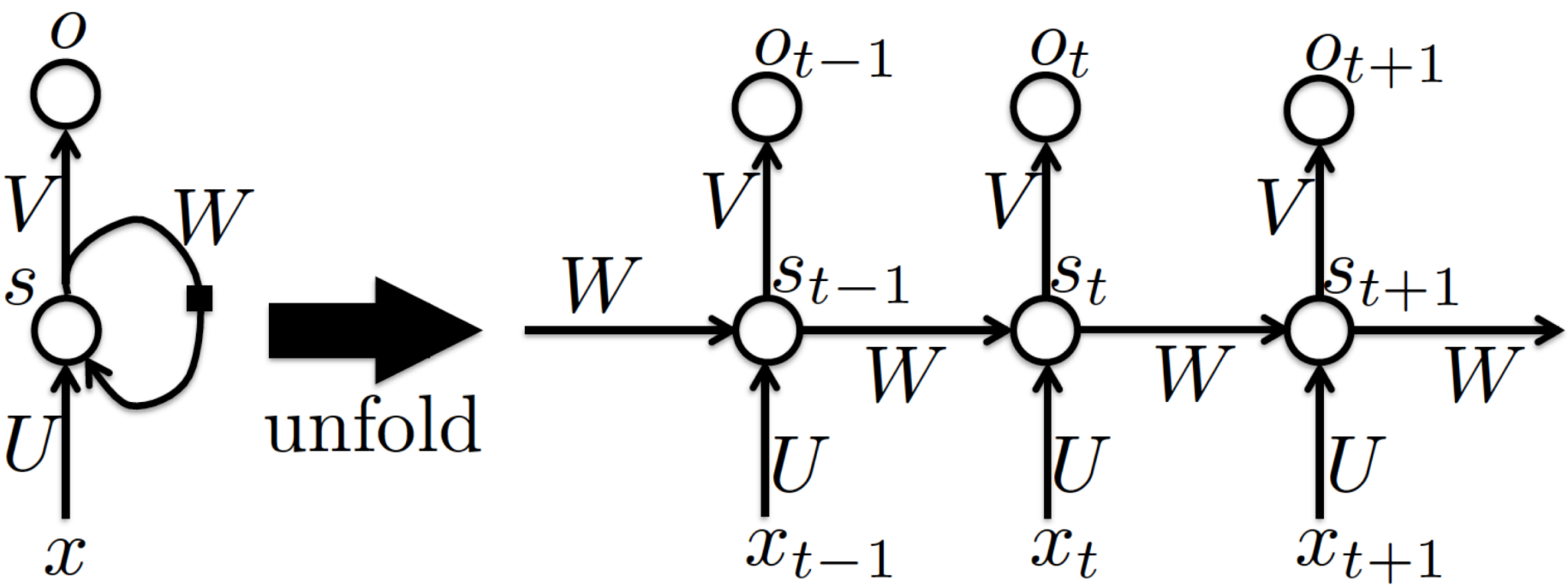
$$s_t = F_\theta(s_{t-1}, x_t)$$




$$s_t = G_t(x_t, x_{t-1}, x_{t-2}, \ldots, x_2, x_1)$$

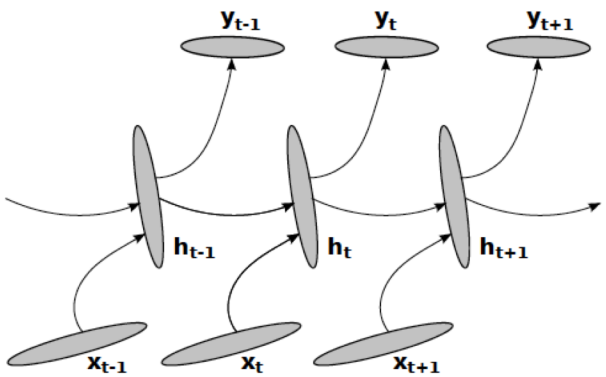
# Recurrent Neural Network

- Can produce an output at each time step: unfolding the graph tells us how to back-prop through time.



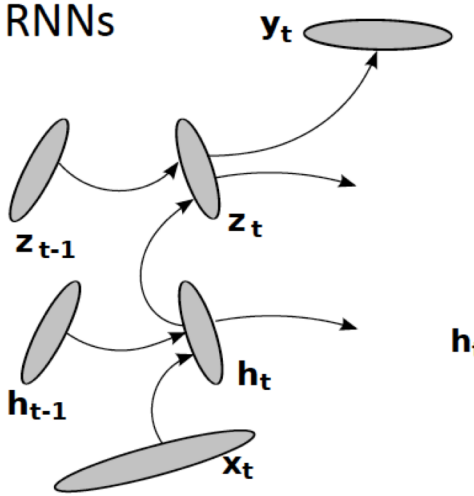
# Increase the expressive power of RNN with more depth

- ICLR 2014, *How to construct deep recurrent neural networks*

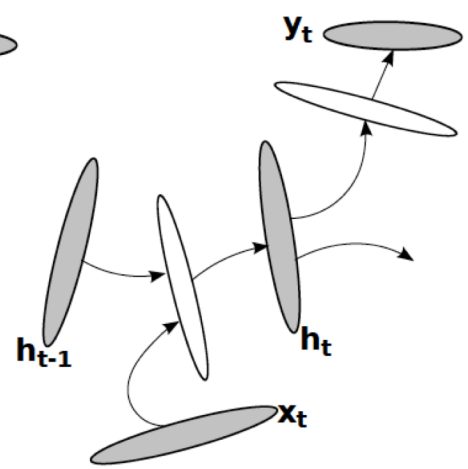


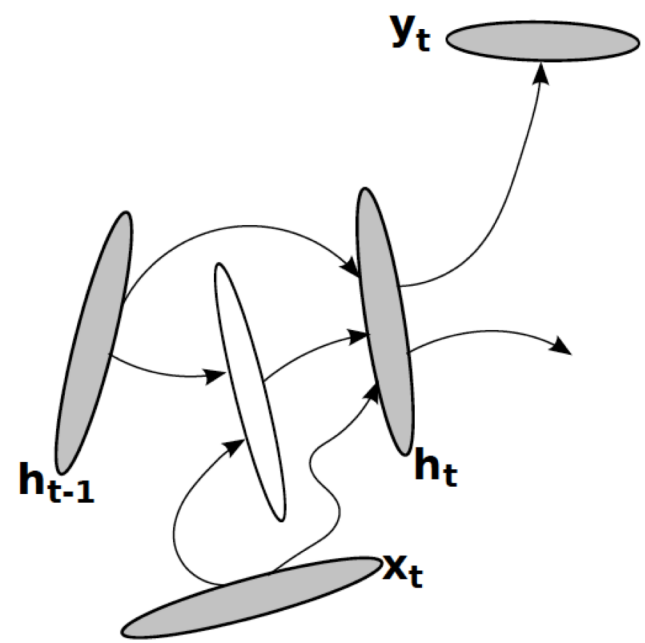

Ordinary RNNs

+ deep hid-to-out
+ deep hid-to-hid
+deep in-to-hid




+ stacking




+ skip connections for creating shorter paths

# Long-term dependencies

- The RNN gradient is a product of Jacobian matrices, each associated with a step in the forward computation. To store information robustly in a finite-dimensional state, the dynamics must be contractive [Bengio et al 1994].

$$L = L(s_T(s_{T-1}(\ldots s_{t+1}(s_t, \ldots))))$$

$$\frac{\partial L}{\partial s_t} = \frac{\partial L}{\partial s_T} \frac{\partial s_T}{\partial s_{T-1}} \cdots \frac{\partial s_{t+1}}{\partial s_t}$$

Storing bits robustly requires sing. values<1

- Problems:
  - sing. values of Jacobians > 1 → *gradients explode* → **Gradient clipping**
  - or sing. values < 1 → *gradients shrink & vanish* *(Hochreiter 1991)*
  - or random → variance grows exponentially

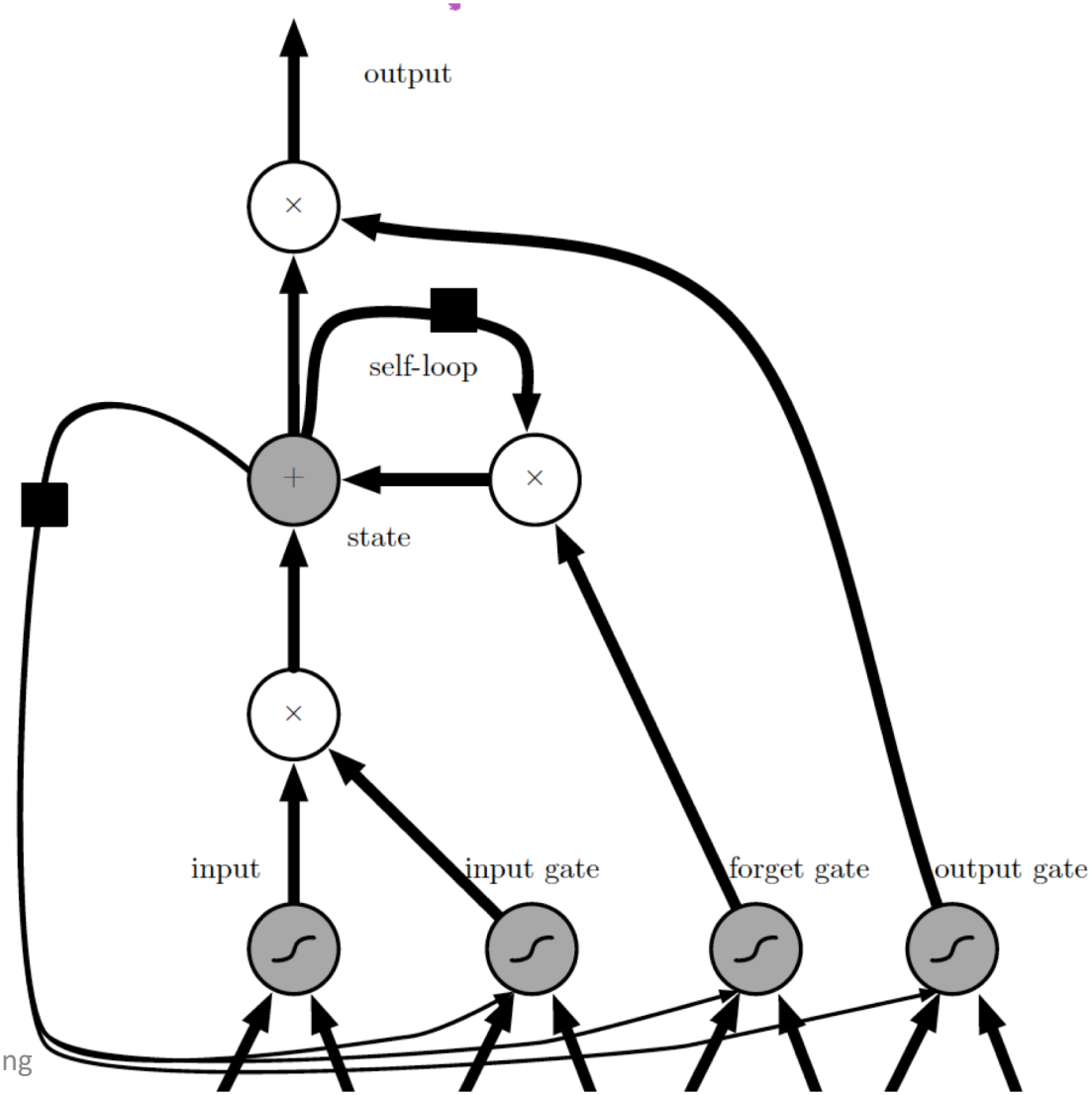
# RNN Tricks

(Pascanu, Mikolov, Bengio, ICML 2013; Bengio, Boulanger & Pascanu, ICASSP 2013)

- Clipping gradients (avoid exploding gradients)
- Momentum
- Initialization (start in right ballpark avoids exploding/vanishing)
- LSTM self-loops (avoid vanishing gradient)

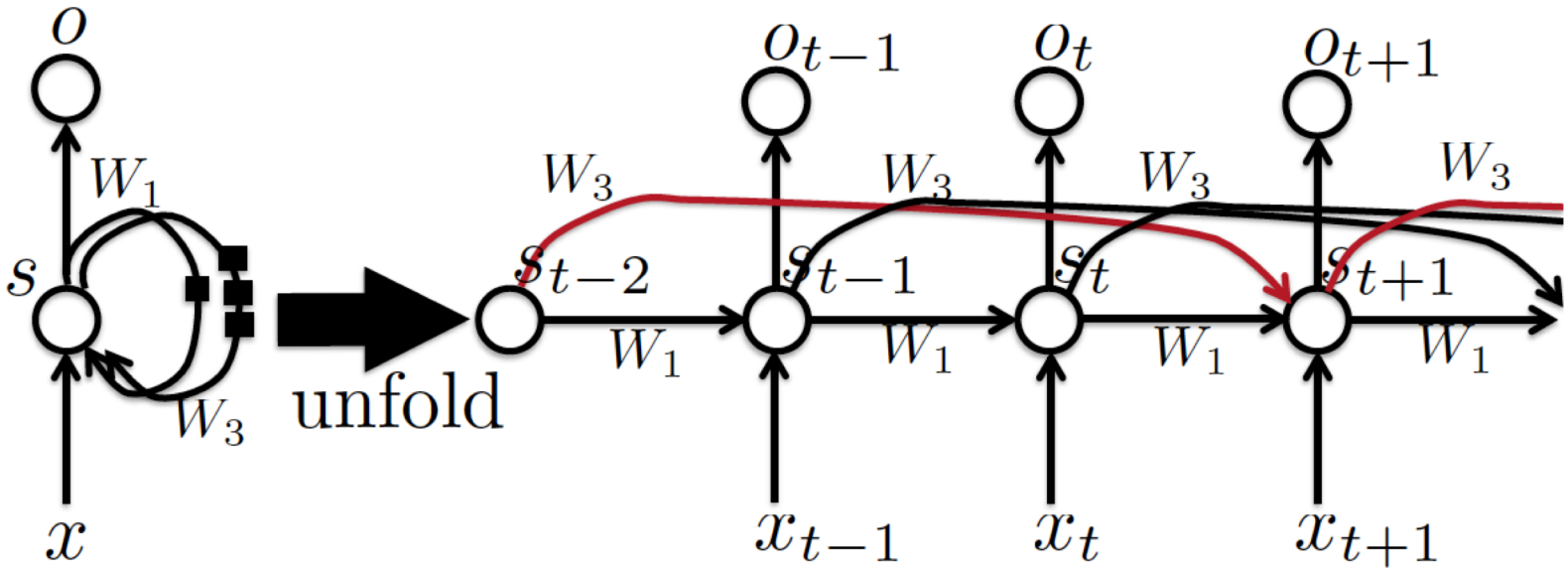
# Gated Recurrent Units and Long- and Short-Term Memory (LSTM)

- Create a path where gradients can flow for longer with self-loop
- Corresponds to an eigenvalue of Jacobian slightly less than 1
- LSTM is **heavily used** *(Hochreiter & Schmidhuber 1997)*
- GRU light-weight version *(Cho et al 2014)*

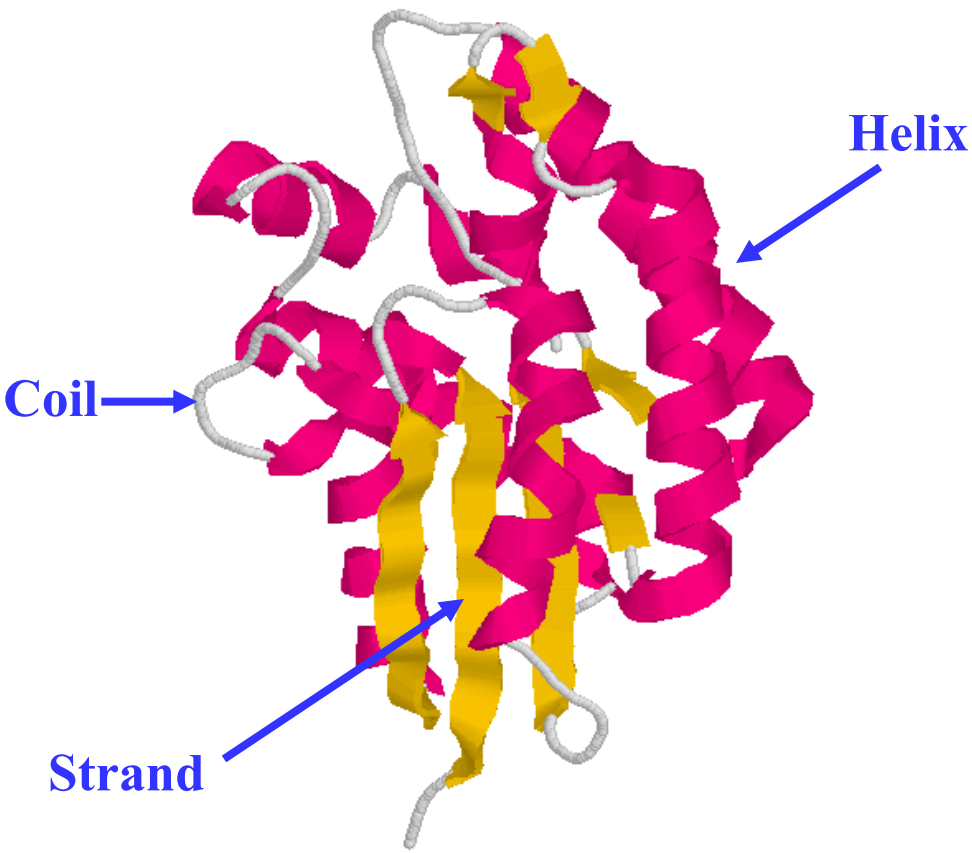


# RNN Tricks

- Delays and multiple time scales, Elhihi & Bengio NIPS 1996



# 1D: Secondary Structure Prediction




MWLKKFGINLLIGQSVOR

↓

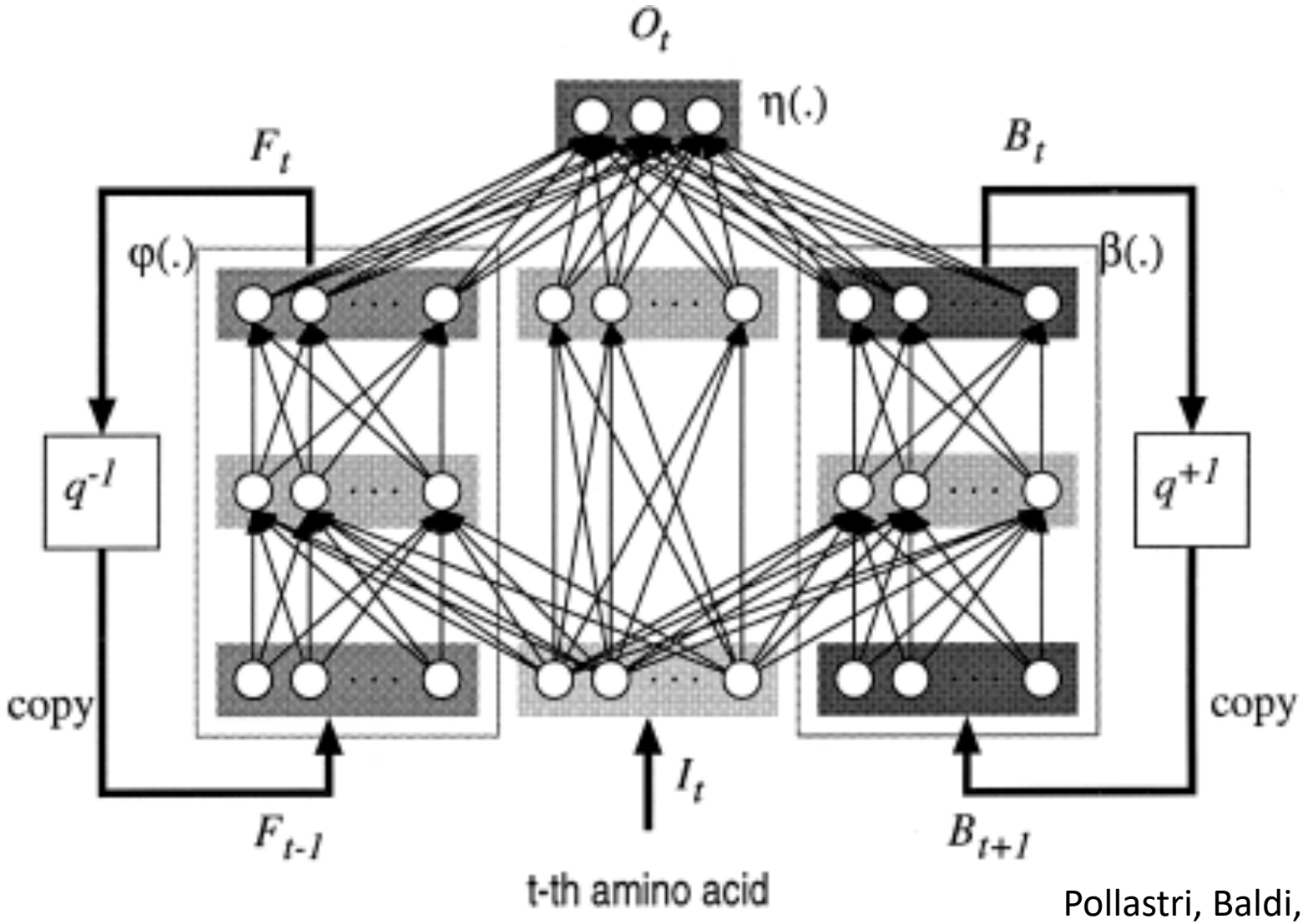
**Neural Networks**

↓

CCCCHHHHHCCCSSSSSS

**Cheng, Randall, Sweredoski, Baldi. *Nucleic Acid Research*, 2005**

# Bidirectional Recurrent Neural Network for Protein Secondary Structure Prediction




Pollastri, Baldi, 2002
Cheng et al., 2006

# The Convergence of Gradient Descent

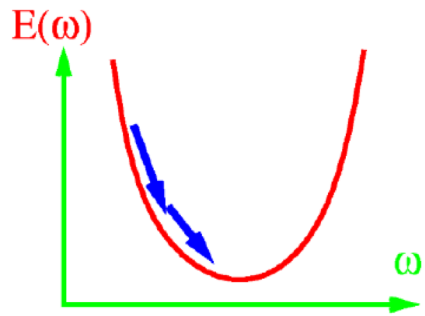
$$\omega \leftarrow \omega - \eta \frac{\partial E}{\partial \omega}$$

gradient of objective function

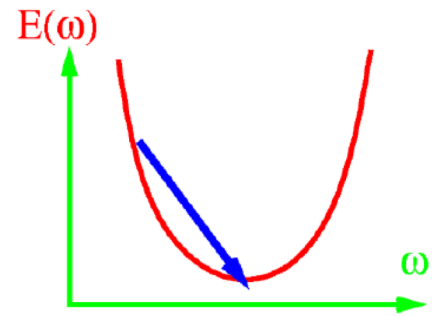
weight vector

learning rate

- Batch Gradient
- There is an optimal learning rate
- Equal to inverse 2nd derivative

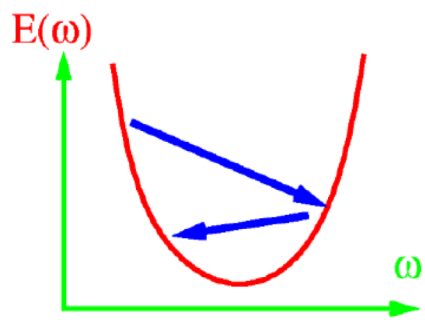



$\eta < \eta_{opt}$

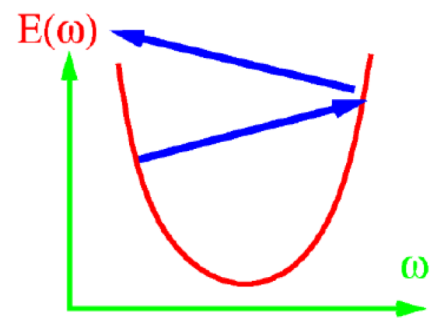



$\eta = \eta_{opt}$

$$\eta_{opt} = \left( \frac{\partial^2 E}{\partial \omega^2} \right)^{-1}$$



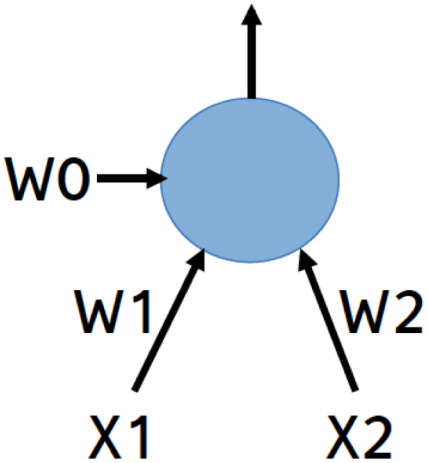

$\eta > \eta_{opt}$

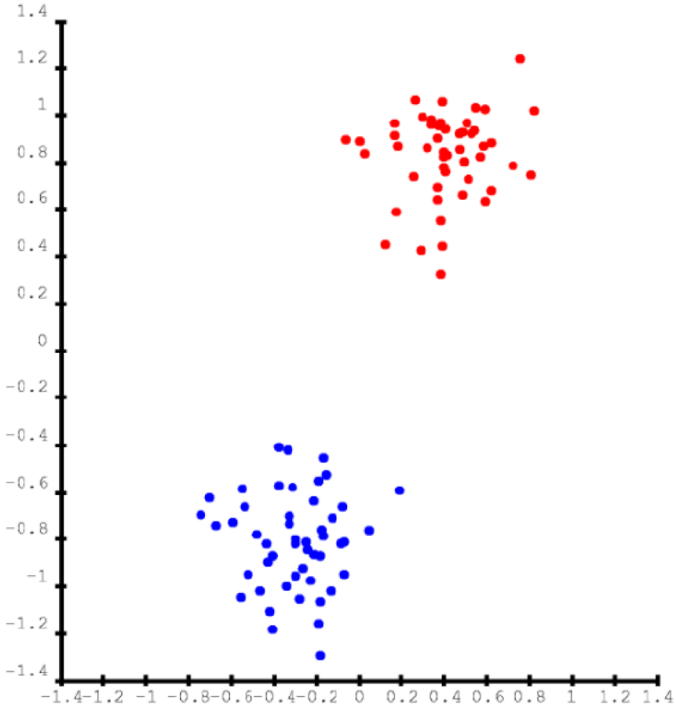



$\eta > 2\,\eta_{opt}$

# Let's Look at a single linear unit

- Single unit, 2 inputs
- Quadratic loss
  - $E(W) = 1/p \sum_p (Y - W \bullet X_p)^2$
- Dataset: classification: Y=-1 for blue, +1 for red.
- Hessian is covariance matrix of input vectors
  - $H = 1/p \sum X_p X_p^T$
- To avoid ill conditioning: normalize the inputs
  - Zero mean
  - Unit variance for all variable



## Batch Gradient, small learning rate

**Learning rate:**

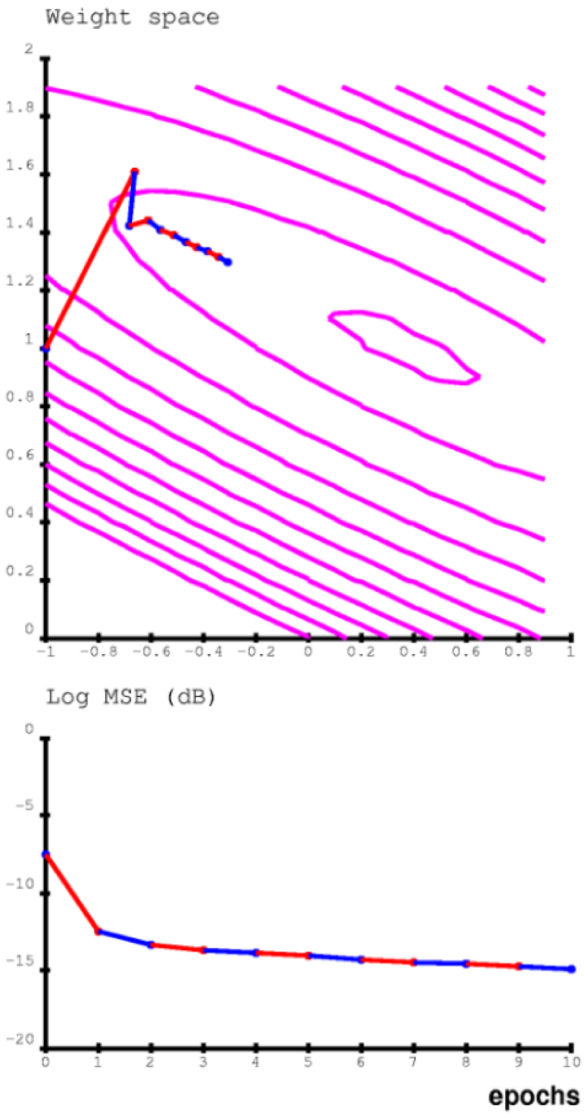
$\eta = 1.5$

**Hessian largest eigenvalue:**

$\lambda_{max} = 0.84$

**Maximum admissible Learning rate:**

$\eta_{max} = 2.38$




## Batch Gradient, large learning rate

**Learning rate:**

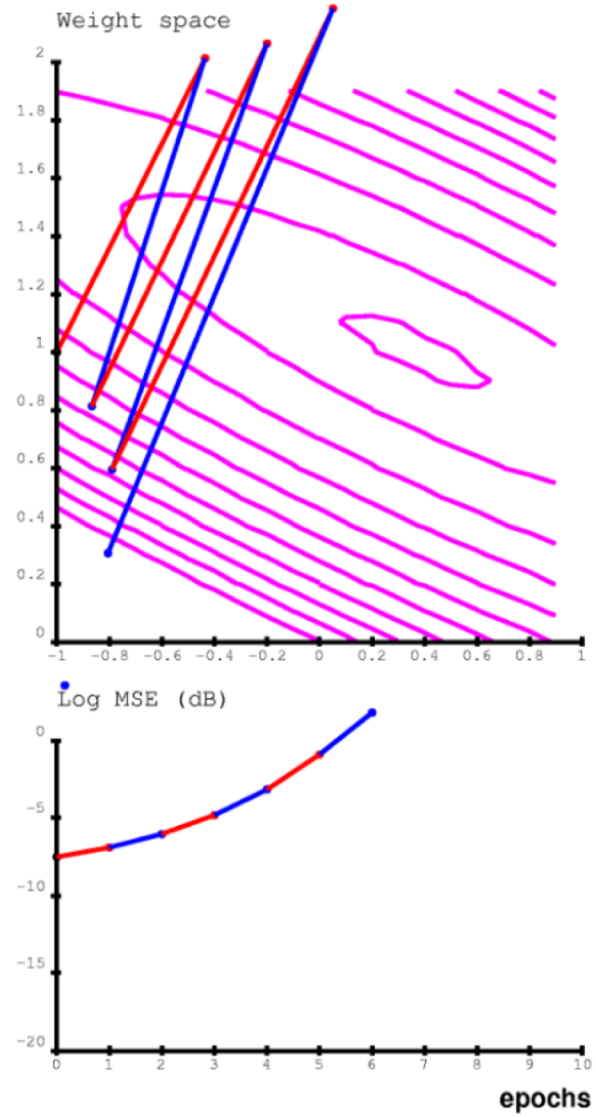
$\eta = 2.5$

**Hessian largest eigenvalue:**

$\lambda_{max} = 0.84$

**Maximum admissible Learning rate:**

$\eta_{max} = 2.38$



- Batch Gradient, small learning rate

Learning rate:

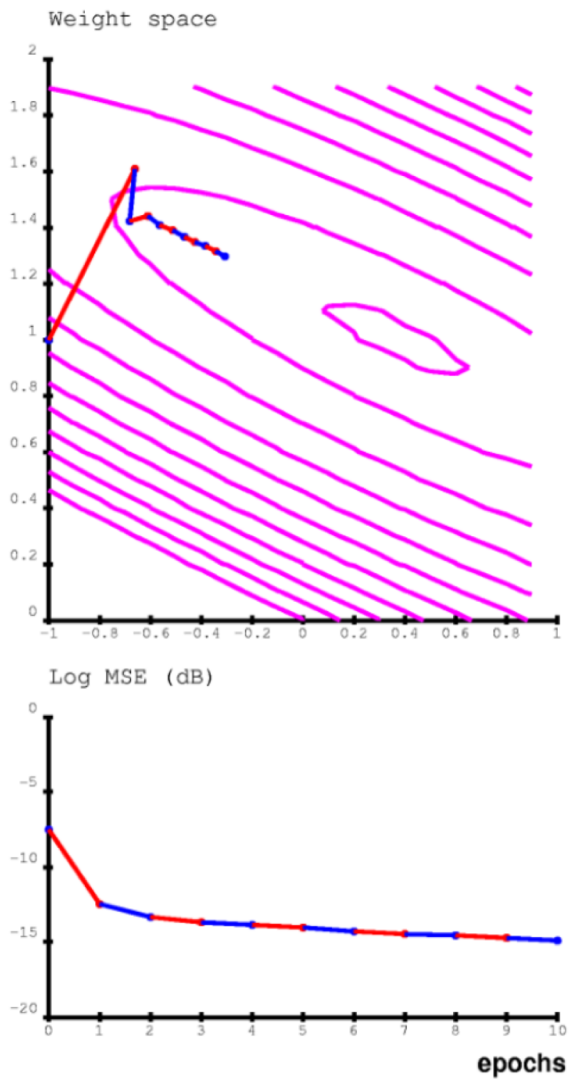
$\eta = 1.5$

Hessian largest eigenvalue:

$\lambda_{max} = 0.84$

Maximum admissible Learning rate:

$\eta_{max} = 2.38$




- Stochastic Gradient: Much Faster
- But fluctuates near the minimum

Learning rate:

$\eta = 0.2$

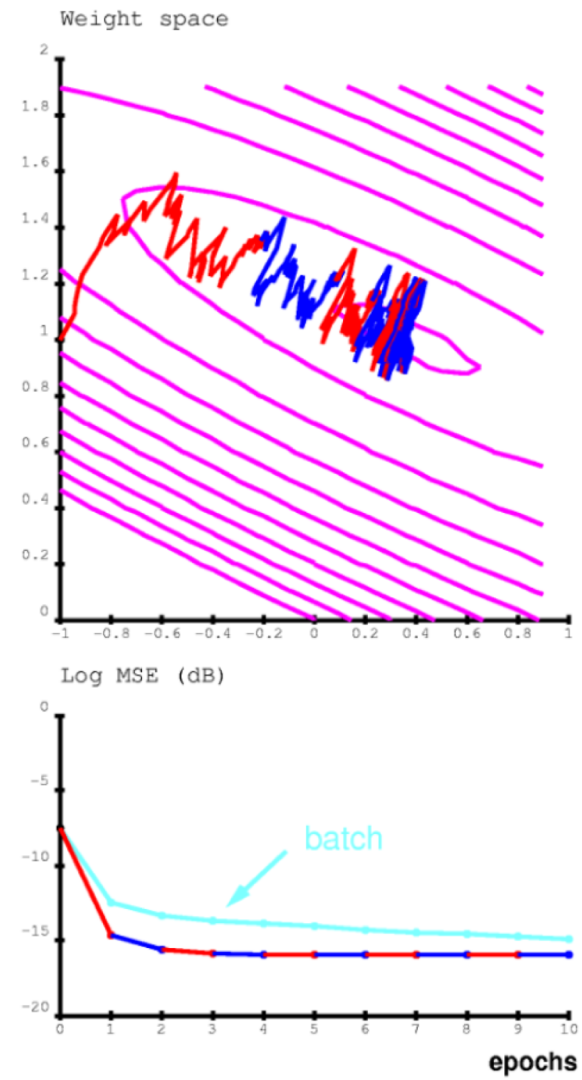
(equivalent to a batch learning rate of 20)

Hessian largest eigenvalue:

$\lambda_{max} = 0.84$

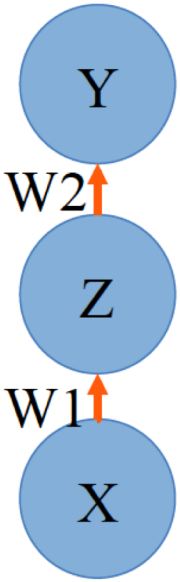
Maximum admissible Learning rate (for batch):

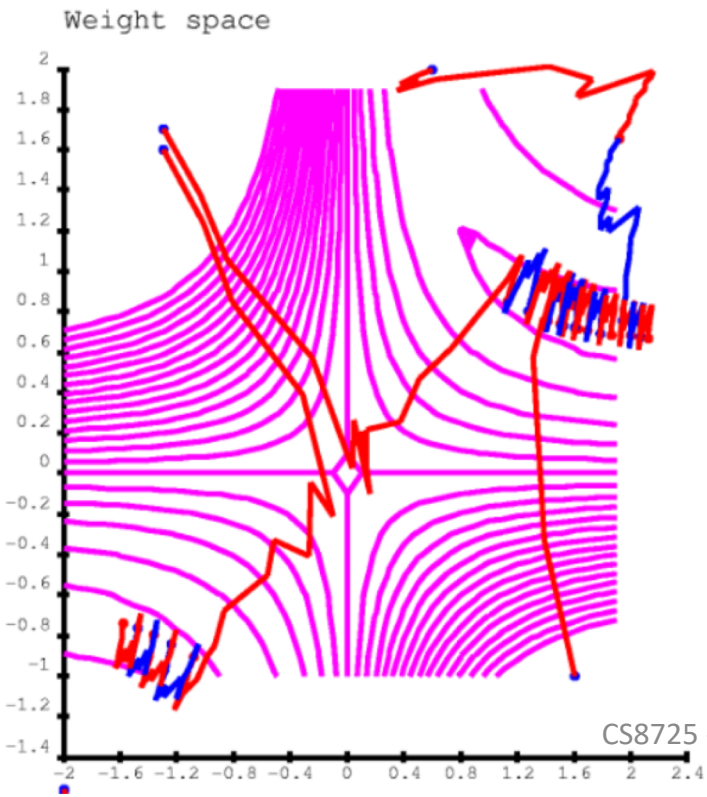
$\eta_{max} = 2.38$

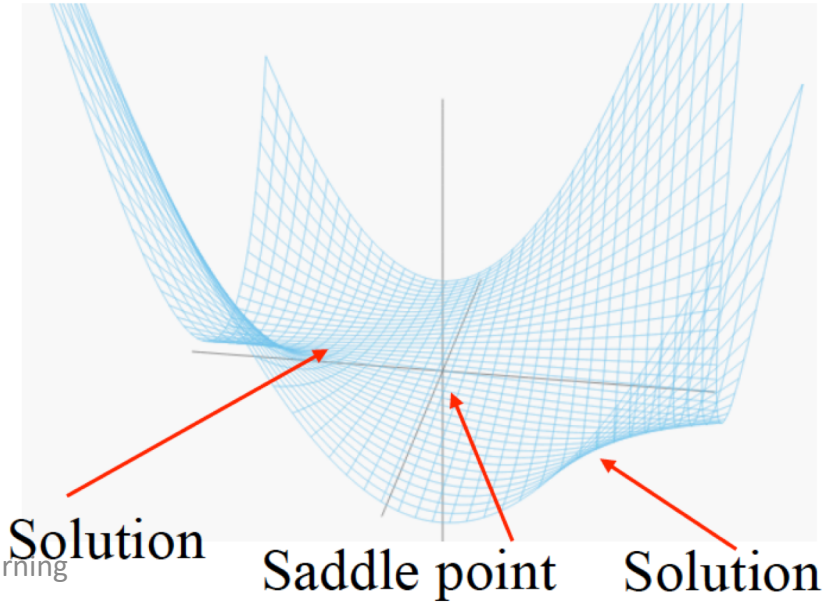


# Multilayer Nets Have Non-Convex Objective Functions

- 1-1-1 network
  - Y = W1*W2*X
- trained to compute the identity function with quadratic loss
  - Single sample X=1, Y=1  L(W) = (1-W1*W2)^2
- Solution: W2 = 1/W2  hyperbola.

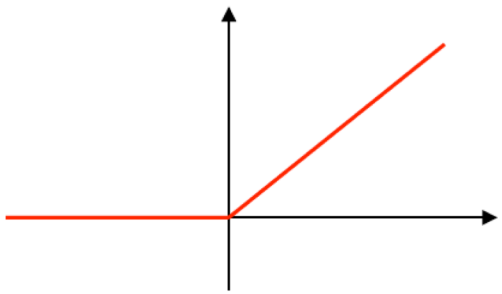


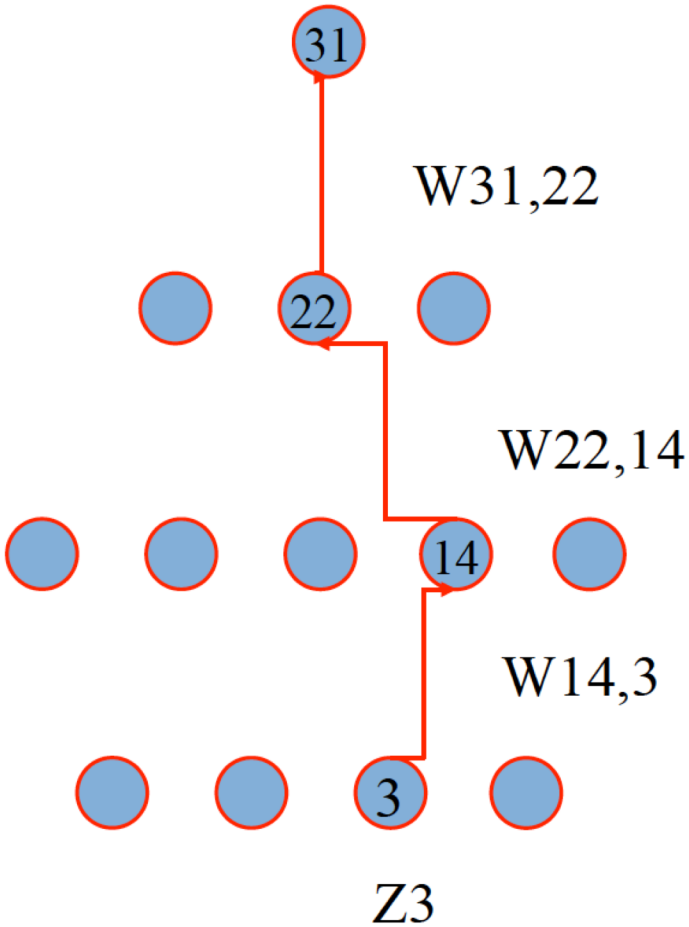



# Deep Nets with ReLUs and Max Pooling

- Stack of linear transforms interspersed with Max operators
- Point-wise ReLUs:





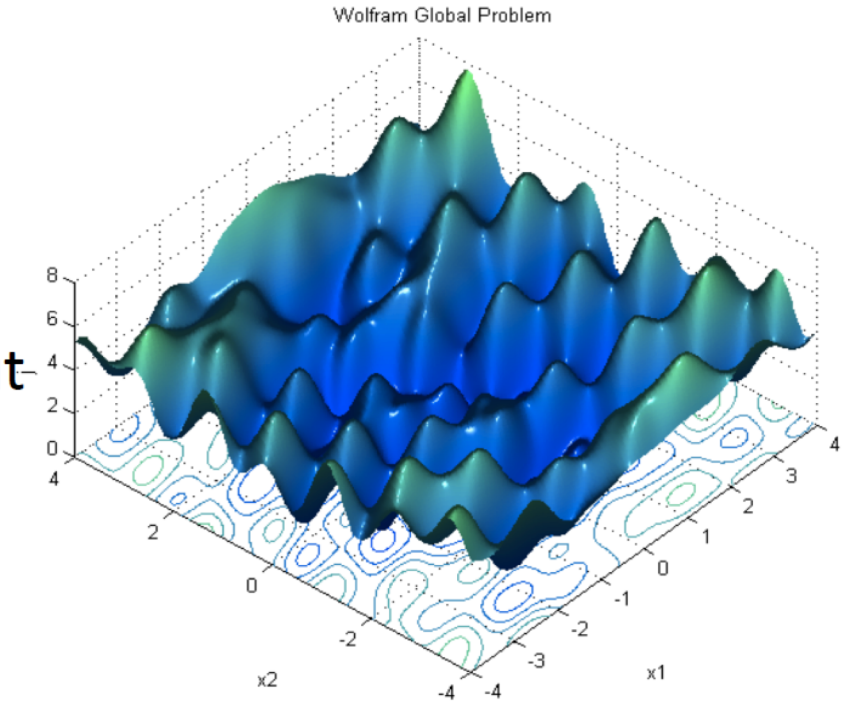

- Max Pooling
  - "switches" from one layer to the next
- Input-output function
  - Sum over active paths
  - Product of all weights along the path
  - Solutions are hyperbolas
- Objective function is full of saddle points

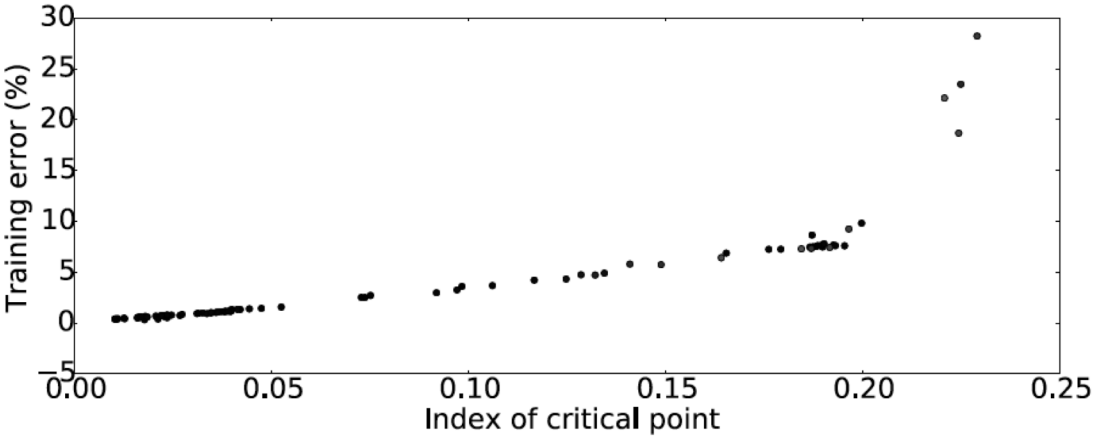
# A Myth Has Been Debunked: Local Minima in Neural Nets → Convexity is not needed

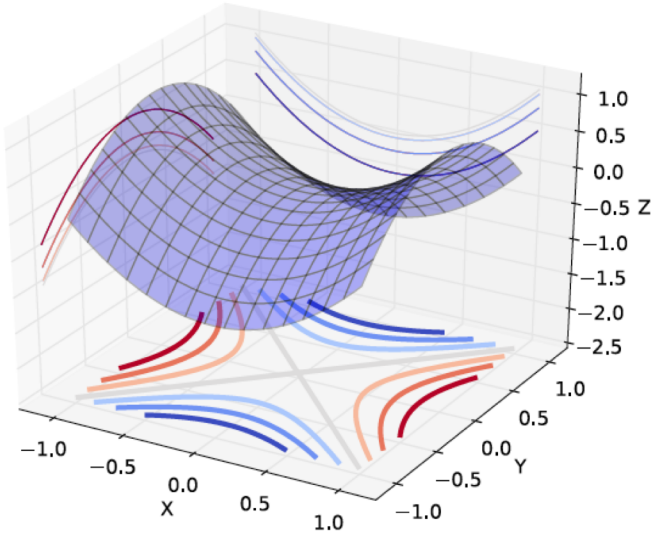
- (Pascanu, Dauphin, Ganguli, Bengio, arXiv May 2014): *On the saddle point problem for non-convex optimization*
- (Dauphin, Pascanu, Gulcehre, Cho, Ganguli, Bengio, NIPS' 2014): *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*
- (Choromanska, Henaff, Mathieu, Ben Arous & LeCun, AISTATS'2015): *The Loss Surface of Multilayer Nets*

# Saddle Points

- Local minima dominate in low-D, but saddle points dominate in high-D
- Most local minima are close to the bottom (global minimum error)

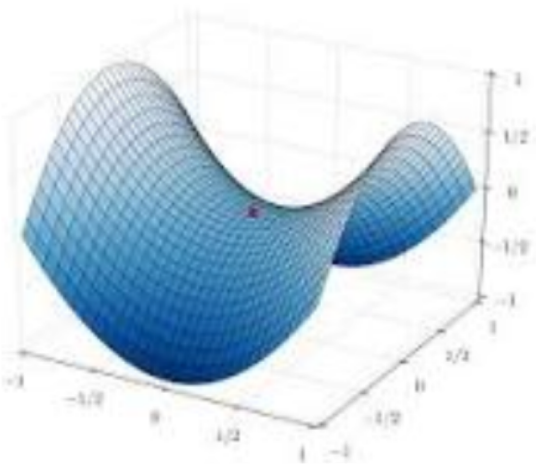


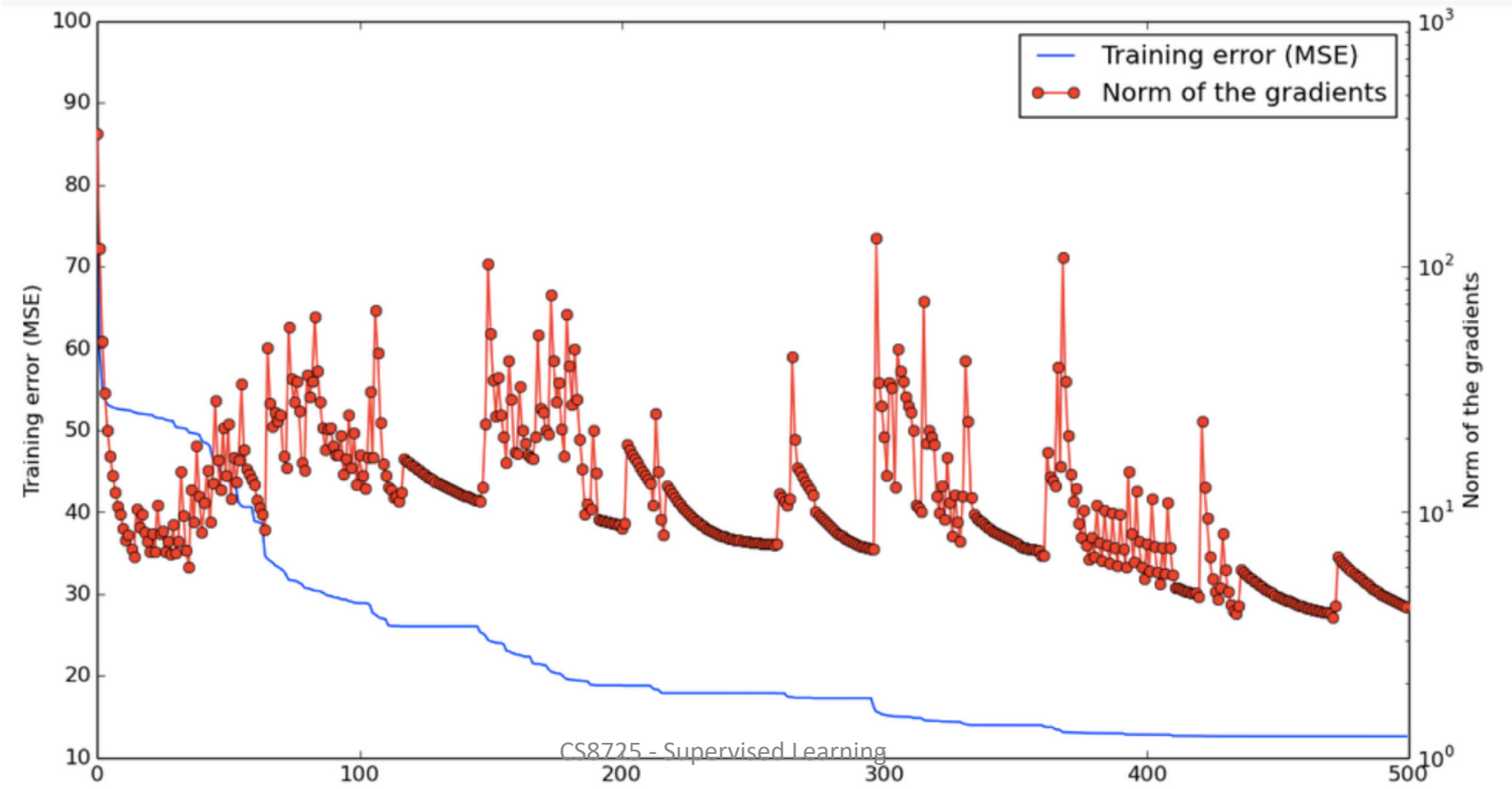
# Saddle Point

In mathematics, a **saddle point** or minimax **point** is a **point** on the surface of the graph of a function where the slopes (derivatives) of orthogonal function components defining the surface become zero (a stationary **point**) but are not a local extremum on both axes.

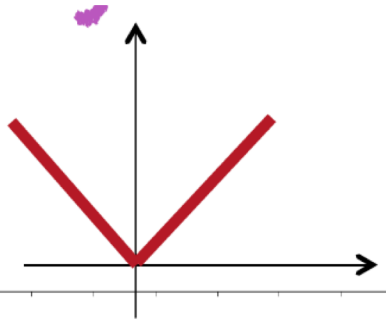


# Saddle Points During Training

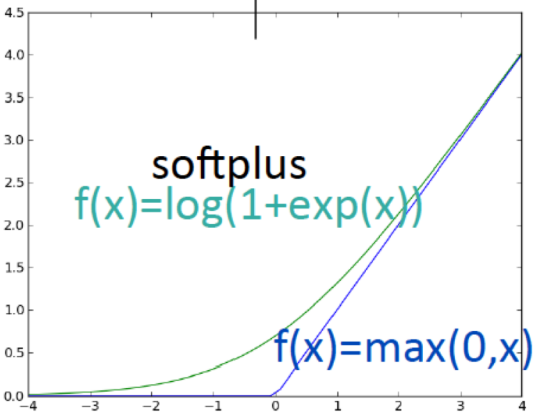
- Oscillating between two behaviors:
  - Slowly approaching a saddle point
  - Escaping it



# Piecewise Linear Nonlinearity



- *Jarreth, Kavukcuoglu, Ranzato & LeCun ICCV 2009*: absolute value rectification works better than tanh in lower layers of convnet




- *Nair & Hinton ICML 2010*: Duplicating sigmoid units with same weights but different bias in an RBM approximates a rectified linear unit (ReLU)

- *Glorot, Bordes and Bengio AISTATS 2011*: Using a rectifier non-linearity (ReLU) instead of tanh of softplus allows for the first time to **train very deep supervised networks** without the need for unsupervised pre-training; was **biologically motivated**

**Neuroscience motivations**

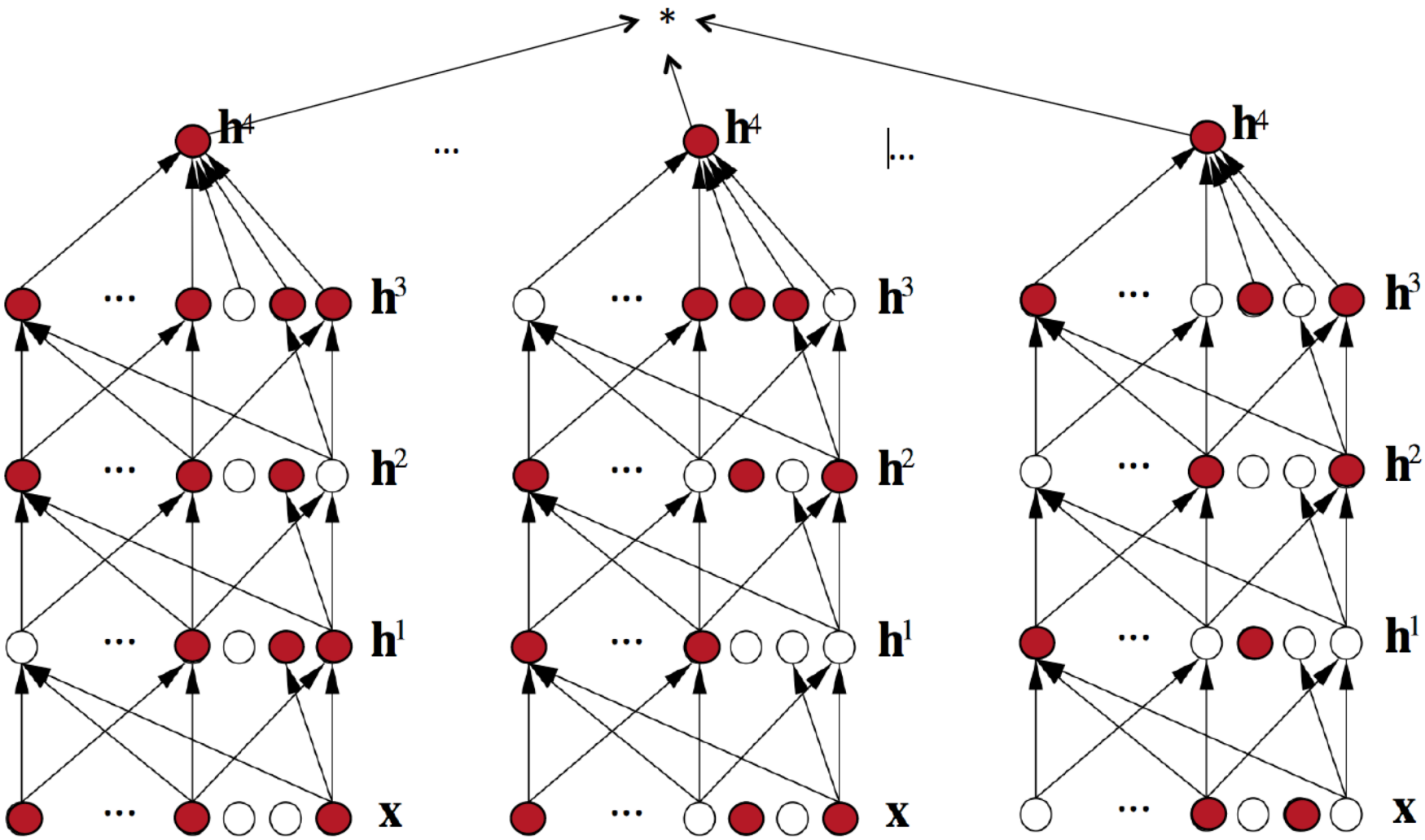
Leaky integrate-and-fire model

- *Krizhevsky, Sutskever & Hinton NIPS 2012*: rectifiers one of the crucial ingredients in ImageNet breakthrough

# Stochastic Neurons as Regularizer: Improving neural networks by prevenHng co-adaptaHon of feature detectors (Hinton et al 2012, arXiv)

- **Dropouts** trick: during training multiply neuron output by random bit (p=0.5), during test by 0.5
- Used in deep supervised networks
- Similar to denoising auto-encoder, but corrupting every layer
- Works better with some non-linearities (rectifiers, maxout) (Goodfellow et al. ICML 2013)
- Equivalent to averaging over exponentially many architectures
  - Used by Krizhevsky et al to break through ImageNet SOTA
  - Also improves SOTA on CIFAR-10 (18→16% err)
  - Knowledge-free MNIST with DBMs (.95→.79% err)
  - TIMIT phoneme classification (22.7→19.7% err)

# Dropout Regularizer: Super-Efficient Bagging



# Batch Normalization

*(Ioffe & Szegedy ICML 2015)*

- Standardize activations (before nonlinearity) across **minibatch**
- **Backprop through this operation**
- Regularizes & helps to train

$$\bar{\mathbf{x}}_k = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_{i,k},$$

$$\sigma_k^2 = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_{i,k} - \bar{\mathbf{x}}_k)^2$$

$$\hat{\mathbf{x}}_k = \frac{\mathbf{x}_k - \bar{\mathbf{x}}_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

$$BN(\mathbf{x}_k) = \gamma_k \hat{\mathbf{x}}_k + \beta_k$$

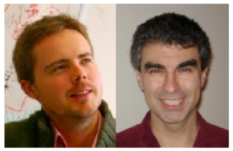
$$\mathbf{y} = \phi(BN(\mathbf{W}\mathbf{x}))$$

# Early Stopping

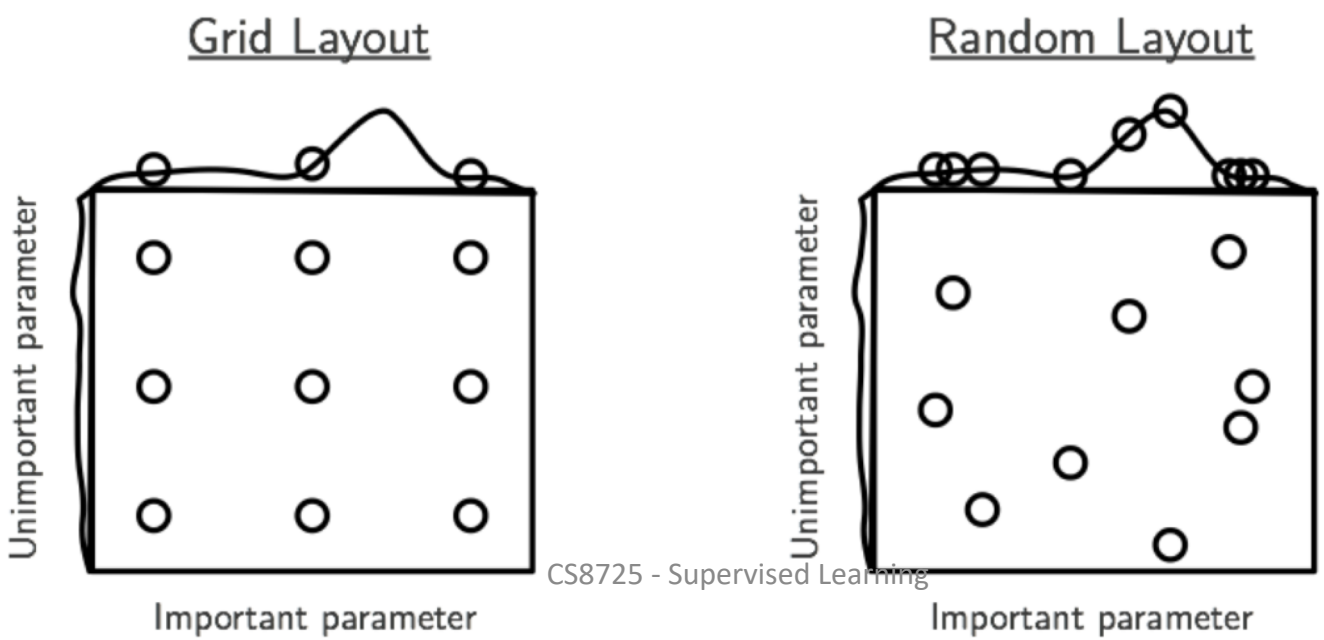
- Beautiful **FREE LUNCH** (no need to launch many different training runs for each value of hyper-parameter for #iterations)
- Monitor validation error during training (after visiting # of training examples = a multiple of validation set size)
- Keep track of parameters with best validation error and report them at the end
- If error does not improve enough (with some patience), stop.

# Random Sampling of Hyperparameters

(Bergstra & Bengio 2012)

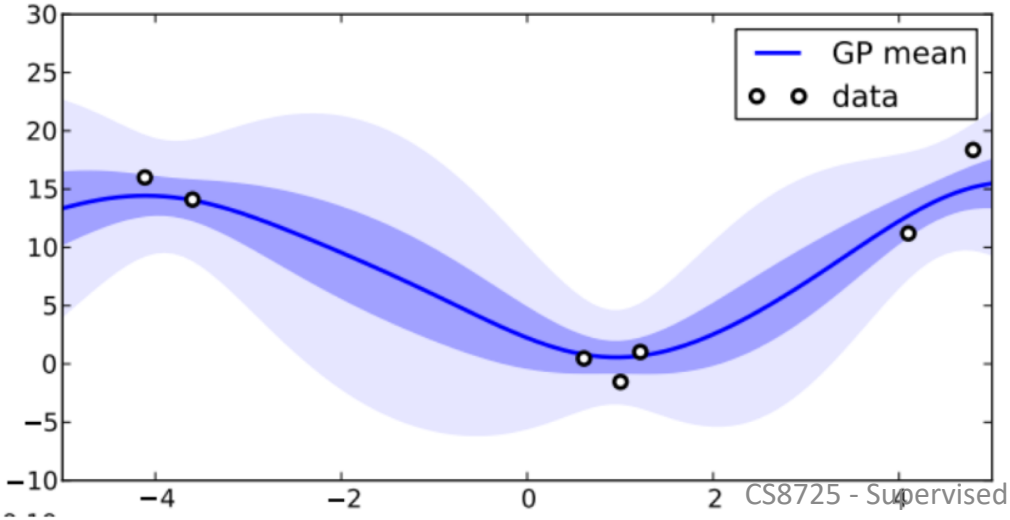


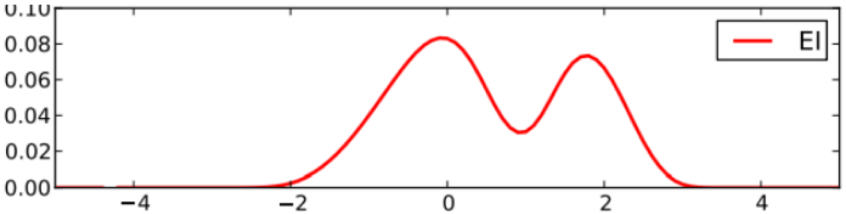
- Common approach: manual + grid search
- Grid search over hyperparameters: simple & wasteful
- Random search: simple & efficient
  - Independently sample each HP, e.g. l.rate~exp(U[log(.1),log(.0001)])
  - Each training trial is iid
  - If a HP is irrelevant grid search is wasteful
  - More convenient: ok to early-stop, continue further, etc.



# Sequential Model-Based Optimization of Hyper-Parameters

- (Hutter et al JAIR 2009; Bergstra et al NIPS 2011; Thornton et al arXiv 2012; Snoek et al NIPS 2012)
- Iterate
- Estimate P(valid. err | hyper-params config x, D)
- choose optimistic x, e.g. $\max_x$ P(valid. err < current min. err | x)
- train with config x, observe valid. err. v, D ← D U {(x,v)}



# Distributed Training

- Minibatches
- Large minibatches + 2nd order & natural gradient methods
- Asynchronous SGD (Bengio et al 2003, Le et al ICML 2012, Dean et al NIPS 2012)
  - Data parallelism vs model parallelism
  - Bottleneck: sharing weights/updates among nodes, to avoid node-models to move too far from each other
- EASGD *(Zhang et al NIPS 2015)* works well in practice
- Efficiently exploiting more than a few GPUs remains a challenge

# Deep AutoEncoder

# Generative Adversarial Network (GAN)

# Deep Reinforcement Learning

# Stochastic Optimization Algorithms