

# NEURAL NETWORKS AND DEEP LEARNING 3

## -INTRODUCTION TO DATA SCIENCE-

Lecturer: Darren Homrighausen, PhD

# ADDITIONAL TOPICS

- Dropout
- Activation function
- Types of layers  
(Fully connected, pooling, normalization, and convolutional)
- Representation learning

# Dropout

# DROPOUT

The key idea is to randomly drop hidden units (along with their connections) from the neural network during training

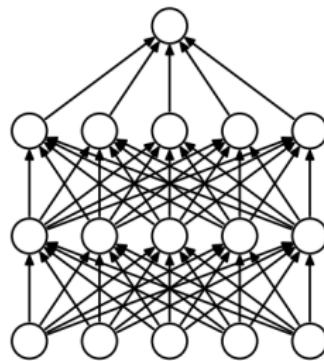
Effectively, this allows for the sampling from an exponential number of different “thinned” networks

When making predictions, we use the single “unthinned” network, but rescale all the weights

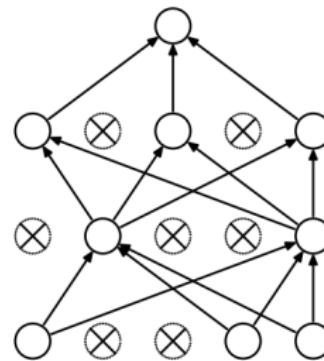
This approach has been shown to significantly reduce overfitting and gives major improvements over other regularization methods

(Srivastava et al. (2014))

# DROPOUT



(a) Standard Neural Net



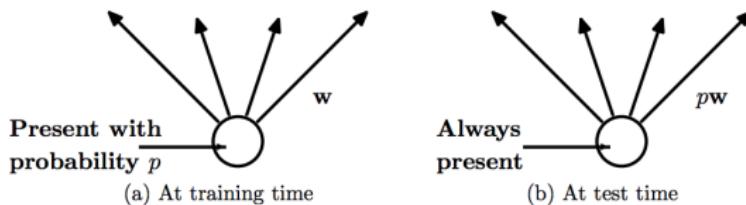
(b) After applying dropout.

Each unit is kept *i.i.d* with probability  $p$  according to..

- $p = 0.5$  for upper layers
- $p \approx 1$  for first layer

There are exponentially many such networks  $\rightarrow$  sampling from many “thinned” networks

# DROPOUT



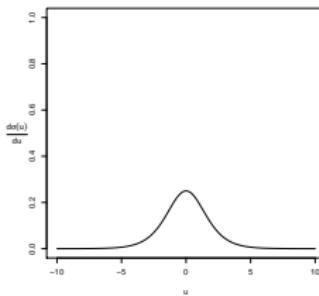
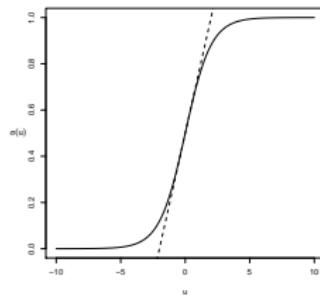
At test time, to mimic the averaging over all of the “thinned” networks, we...

- retain all the original units
- but rescale all the parameters  $w$  (known as **weights**) by the probability  $p$

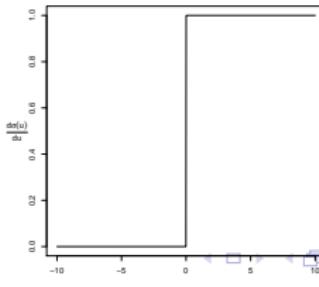
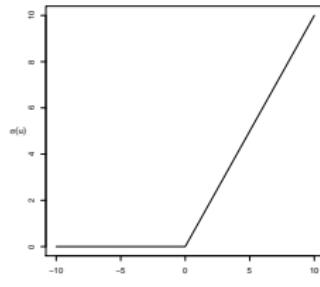
# Activation function

# ACTIVATION FUNCTION

The classical activation function is the **sigmoid**:



Modern neural networks use the **rectilinear activation function**  
(Also known as the “Rectified Linear Unit” or (ReLU))



# RECTILINEAR ACTIVATION FUNCTION

Relative to the sigmoid function, **ReLU**...

- tends to converge faster
- requires “lower level” computations  
(i.e. no exponentials/logarithms)
- Doesn’t have the derivative problem in gradient descent  
(Derivative of  $\sigma(u)$ :  $\sigma(u)(1 - \sigma(u))$ )
- can result in hidden units that are identically zero
- Scale invariant  
 $(\max\{0, au\} = a \max\{0, u\})$

There are variations on ReLU

- “leaky” ReLU  
(Adds a shallow slope to  $u < 0$ :  $\sigma(u) = bu\mathbf{1}(u < 0) + u\mathbf{1}(u \geq 0)$ ,  $b$  very small)
- maxout ReLU  
(This one doesn’t have the form  $\sigma(\mathbb{Z}\alpha)$  but does generalize the various ReLU’s)

# Types of layers

# TYPES OF LAYERS

We have generally discussed **fully connected** layers

Fully connected layers do not scale to large  $p$

(For example: a  $200 \times 200 \times 3$  image  $\rightarrow 200 * 200 * 3 + 1 = 120,001$  parameters per hidden unit)

Moreover, we would want to have several such hidden units/layers

This much connectivity would quickly lead to overfitting and computational difficulties

# TYPES OF LAYERS

If the features have a natural **distance**, then they can be **locally smoothed**

To be concrete, we will use **image data** as the running example/terminology

There are a few, commonly used smoothing steps

- **POOLING**
- **LOCALIZATION**
- **NORMALIZATION**
- **CONVOLUTION**

# POOLING

Perform a **down-sampling** step by taking a local {min, max, average, median,  $\ell^2$  norm}

This is can be interpreted as fixing the parameters  $\alpha$  for a hidden unit at particular and **not** applying the activation function

(Though, technically, only the average can be expressed this way)

# LOCALIZATION

Assign hidden units to regions or channels in an image

Instead of allowing  $\alpha \in \mathbb{R}^P$  to have all nonzero entries, only a **patch** of  $\alpha_j$ 's are nonzero

This dramatically reduces the number of parameters

Numerous choices still need to be made

- **BOUNDARIES:** What do we do with patches that exit the image?
- **STRIDE:** How do we move the patches around?
- **PATCH SIZE:** What are the patch's dimensions?

# NORMALIZATION

As previously noted, neural networks are **very** scale dependent  
(e.g. learning rate)

Hence, it has been suggested that each layer should be renormalized to mean zero and variance 1  
(Not just the original features)

Hence, we can **scale** each hidden unit over the training data:

$$Z_{ik} \leftarrow \frac{Z_{ik} - \bar{Z}_k}{sd(Z_k) + \epsilon}$$

(If using minibatch for the gradient updates, this is done over only the minibatch observations)

The  $\epsilon$  is a fixed, tiny number that prevents division by zero

# NORMALIZATION

In practice, this idea is usually pushed one step further: let the neural network learn the shift/scale as additional parameters

$$Z_{ik} \leftarrow \text{scale}_k \left( \frac{Z_{ik} - \bar{Z}_k}{sd(Z_k) + \epsilon} \right) + \text{shift}_k$$

(This allows the original parameterization as a special case (Ioffe, Szegedy (2015))

# CONVOLUTIONAL

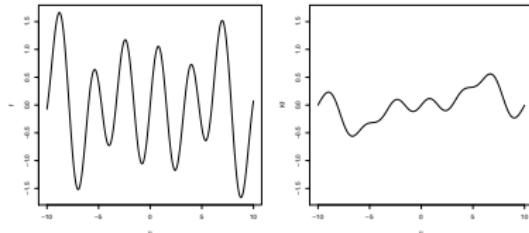
This looks a lot like the **localization** idea, with one change:  
the same **patch** is shared across the image

This even more dramatically reduces the number of parameters

The name comes from the forward pass now being a convolution

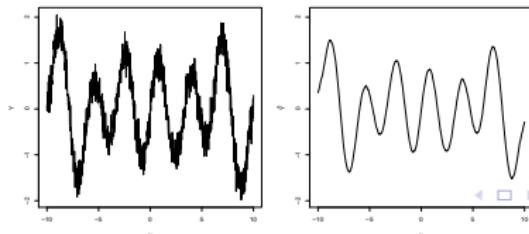
# CONVOLUTIONAL

General convolution:  $Kf(X) = (f * k)(X) = \int f(\tau)k(X - \tau)d\tau$

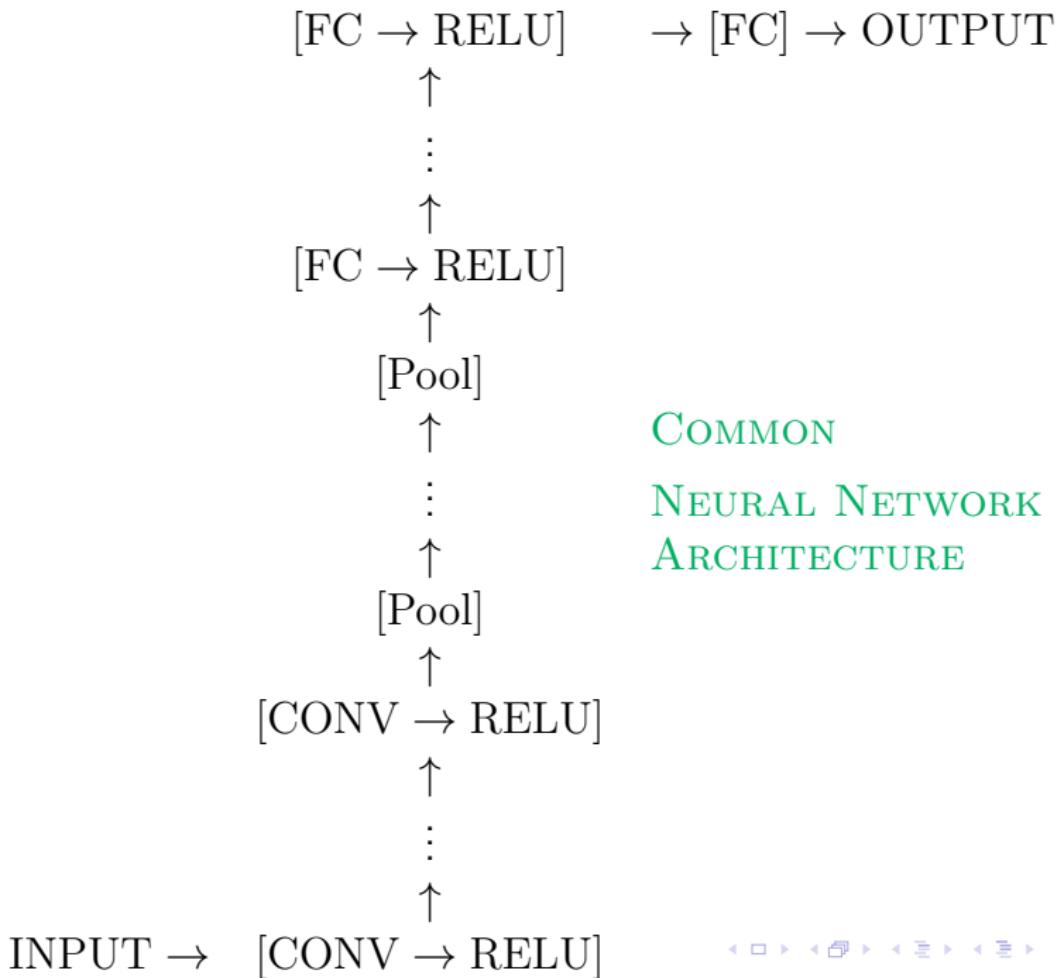


$$\text{Example: } \hat{f}(X) = \frac{\sum_{i=1}^n Y_i k_h(X - X_i)}{\sum_{i=1}^n k_h(X - X_i)}$$

(This is a kernel regression procedure (not to be confused with kernel ridge regression))



# Putting it all together: A neural network architecture



# Representation learning

# OVERVIEW

Representation learning is the idea that performance of ML methods is highly dependent on the choice of data representation

For this reason, much of ML is geared towards transforming the data into the relevant features and then using these as inputs

This idea is as old as statistics itself, really,  
(E.g. Pearson (1901), where PCA was first introduced)

However, the idea is constantly revisited in a variety of fields and contexts

# OVERVIEW

Commonly, these learned representations capture ‘low level’ information like overall shape types

Other sharp features, such as images, aren’t captured

It is possible to quantify this intuition for PCA at least

# PCA

# PCA

Principal components analysis (PCA) is an (unsupervised) dimension reduction technique

It solves various equivalent optimization problems

(Maximize variance, minimize  $L_2$  distortions, find closest subspace of a given rank,...)

At its core, we are finding linear combinations of the original (centered) features

$$Z_{ij} = \alpha_j^\top X_i$$

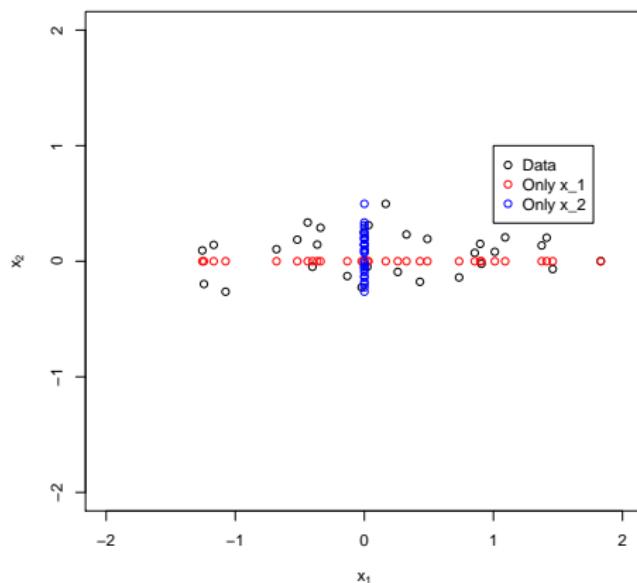
This is expressed via the SVD  $\mathbb{X} - \bar{\mathbb{X}} = UDV^\top$  as

$$Z = \mathbb{X}V = UD$$

# LOWER DIMENSIONAL EMBEDDINGS

Suppose we have predictors  $x_1$  and  $x_2$

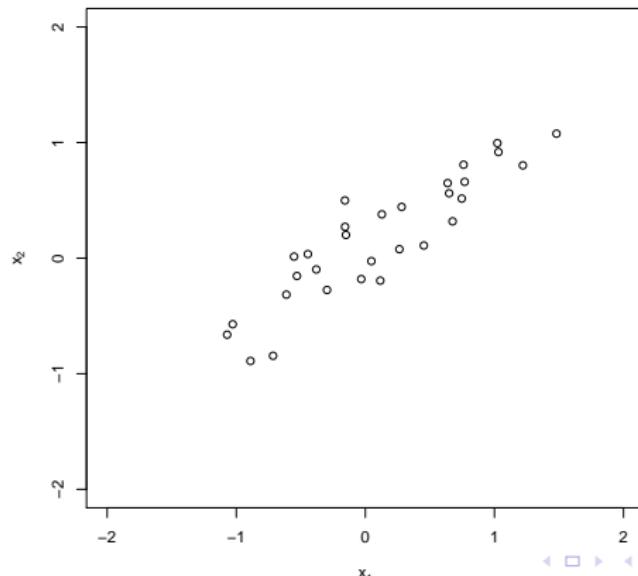
- We more faithfully preserve the structure of the data by keeping  $x_1$  and setting  $x_2$  to zero than the opposite



# LOWER DIMENSIONAL EMBEDDINGS

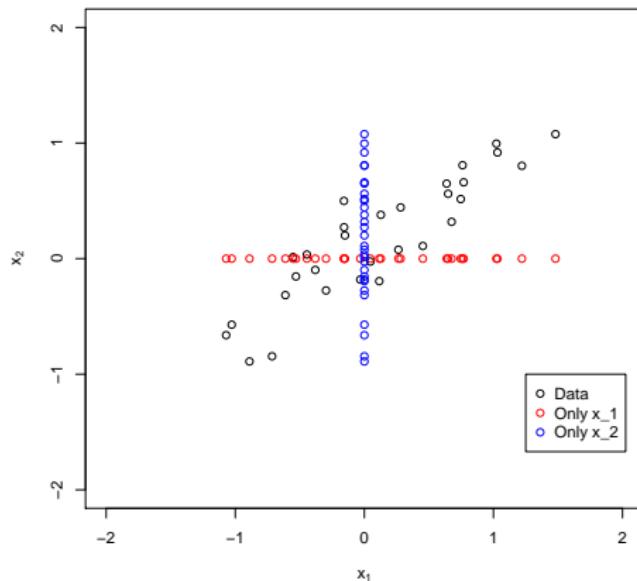
An important feature of the previous example that  $x_1$  and  $x_2$  aren't correlated

What if they are?



# LOWER DIMENSIONAL EMBEDDINGS

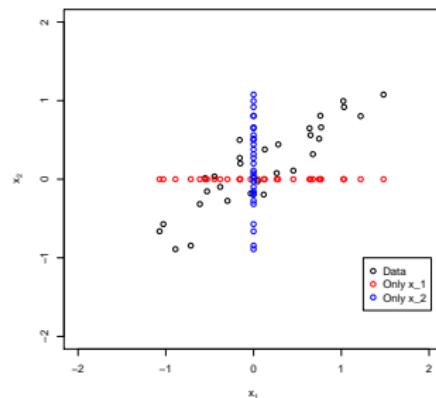
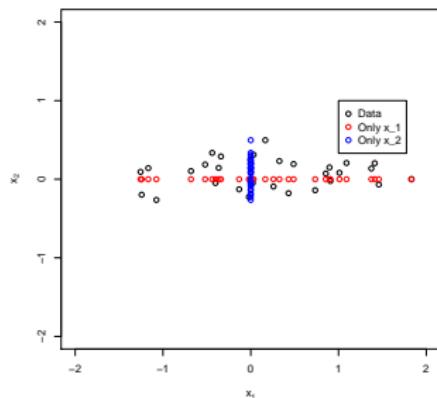
We **do** lose a lot of structure by setting either  $x_1$  or  $x_2$  to zero



# LOWER DIMENSIONAL EMBEDDINGS

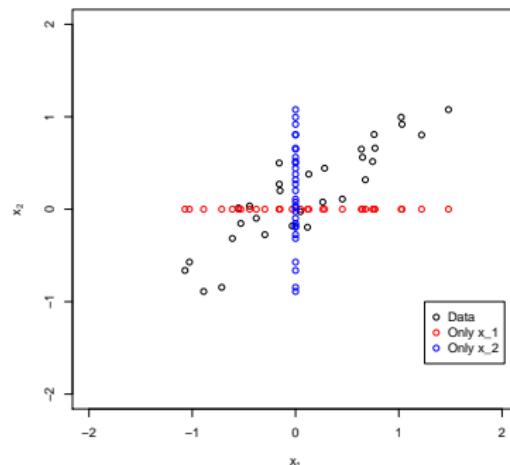
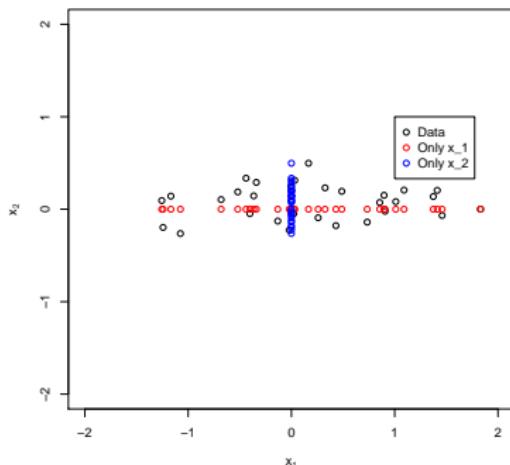
There isn't that much structurally different between the examples

One is just a **rotation** of the other



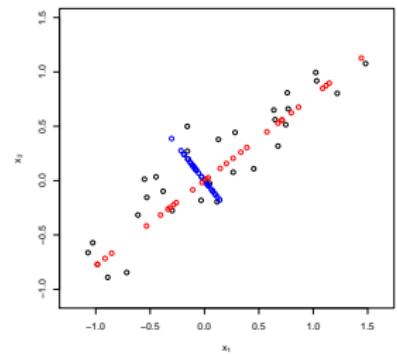
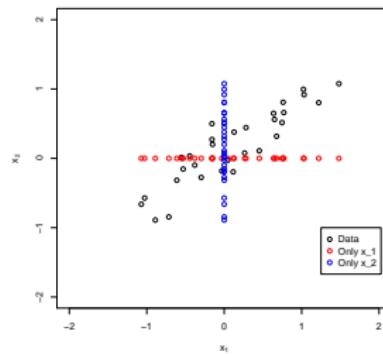
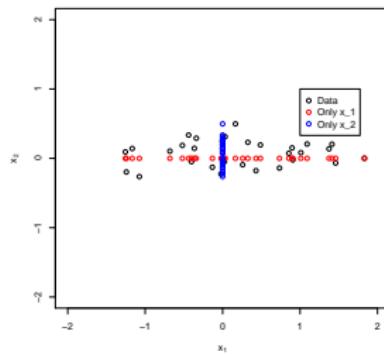
# LOWER DIMENSIONAL EMBEDDINGS

If we knew how to rotate our data, then we would be able to preserve more structure



# LOWER DIMENSIONAL EMBEDDINGS

It turns out that **PCA** gives us exactly this rotation.



# Digits example

# PCA



Source: <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>

The data: 658 handwritten 3s each drawn by a different person

Each image is 16x16 pixels, each taking grayscale values between -1 and 1.

## PCA

Think about each pixel location as a measurement

Consider these simple drawings of 3's. We convert this to an observation in a matrix by **unraveling** it along rows

$$X_1 = \begin{matrix} & \text{[Binary Image of a handwritten digit '1']} \\ X_1 = & [1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1]^\top \end{matrix}$$

$$X_2 = [1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1]^\top$$

(Here, let **black** be 1 and **white** be 0)

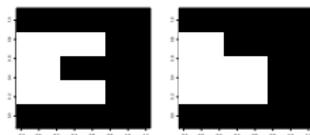
# PCA

We will consider digits with...

- more pixels ( $p = 256$ )
- a continuum of intensities



Vs.



# PCA

```
threesCenter = scale(threes, scale=FALSE)

svd.out = svd(threesCenter)

pcs      = svd.out$v
scores   = svd.out$u%*%diag(svd.out$d)
```

Or, using prcomp:

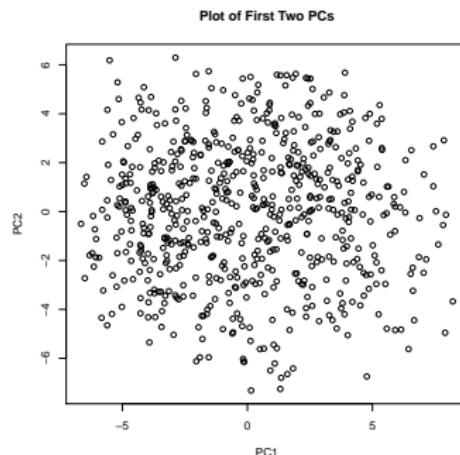
```
out = prcomp(threes, scale=F)
pcs = out$rot
scores = out$x
```

(Note that here we aren't scaling: the measurements are already on a consistent scale)

# PCA

We can plot the scores of the first two principal components versus each other:

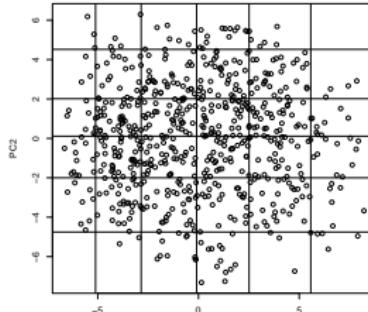
```
plot(scores[,1],scores[,2],xlab = 'PC1',ylab='PC2',  
     main='Plot of First Two PCs')
```



Note: Each circle in this plot represents a hand written '3'.

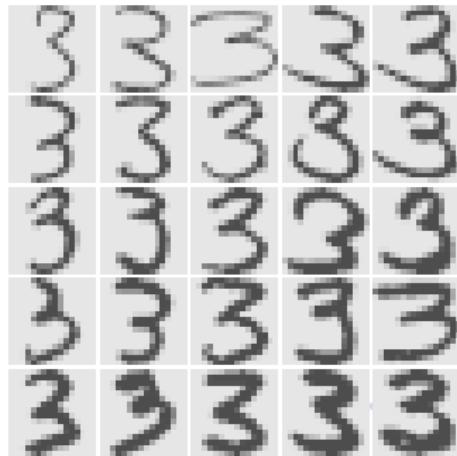
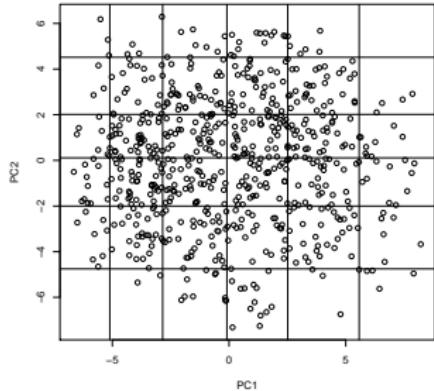
# PCA

```
quantile.vec = c(0.05,0.25,0.5,0.75,0.95)
quant.score1 = quantile(scores[,1],quantile.vec)
quant.score2 = quantile(scores[,2],quantile.vec)
plot(scores[,1],scores[,2],xlab = 'PC1',ylab='PC2')
for(i in 1:5){
  abline(h = quant.score2[i])
  abline(v = quant.score1[i])
}
identify(scores[,1],scores[,2],n=25) #to find points
```



# PCA

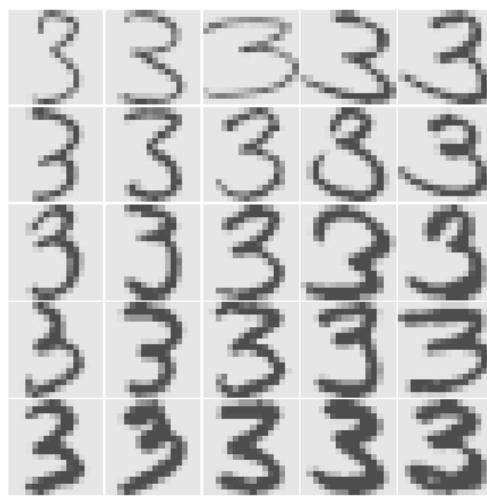
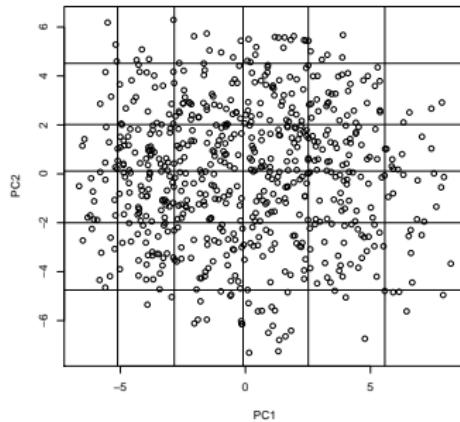
```
pcs.order = c(73,238,550,82,640,284,84,133,4,322,392,241,  
      554,220,500,247,344,142,405,649,184,149,234,375,176)  
par(mfrow=c(5,5))  
par(mar=c(.2,.2,.2,.2))  
for(i in pcs.order){  
  plot.digit(threes[i,])  
}
```



# PCA

The 3's get **lighter** as the location on PC2 increases.

The 3's get more **elongated** as the location along PC1 increases



# PCA

Each number represents a vector in  $\mathbb{R}^{256}$

(as each square is 16x16 pixels)

However, hopefully we can **reduce** this number by  
re-expressing the digits in PC-land

(For instance, the top-right pixel is always 0 and hence that feature is uninteresting)



# PCA

Lastly, we can also look at the loadings as well:



1ST PC: Takes a compact 3 and smears it out



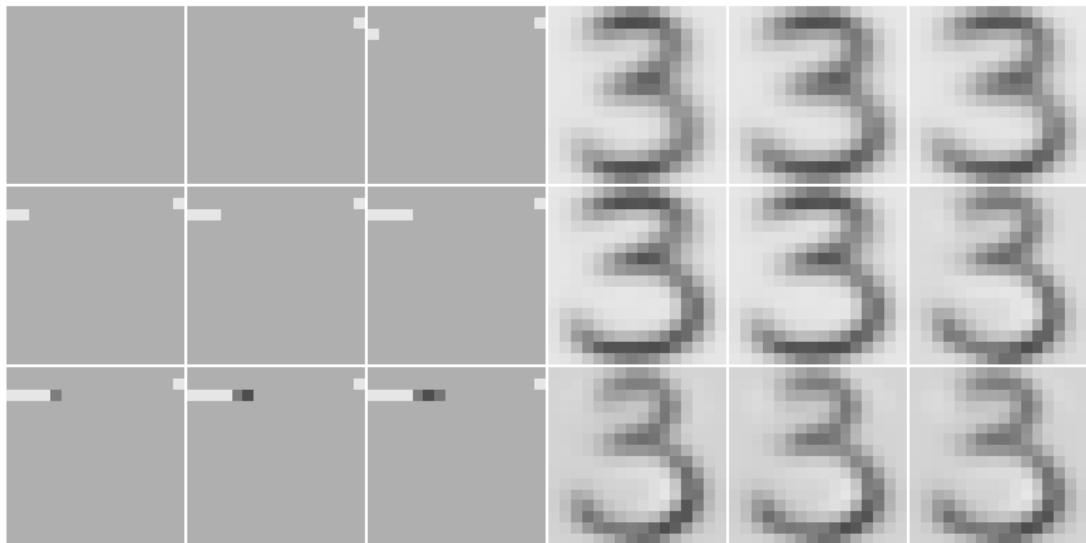
2ND PC: Deletes a portion of the inner part of a 3 and augments the outer (right) part



3RD PC: Moves a 3 down and tips it to the right

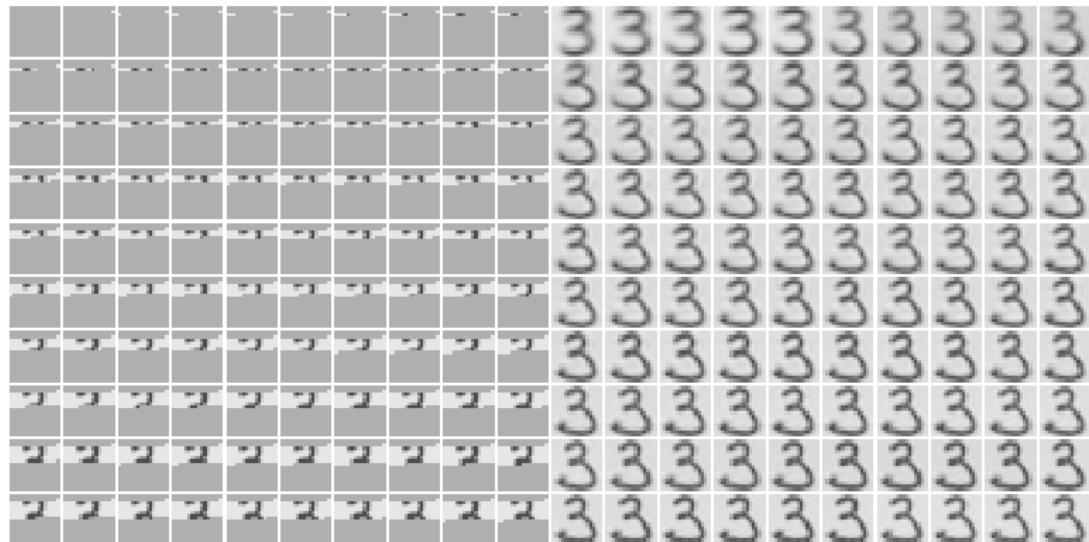
# RECONSTRUCTION

Using 9 axis dimensions



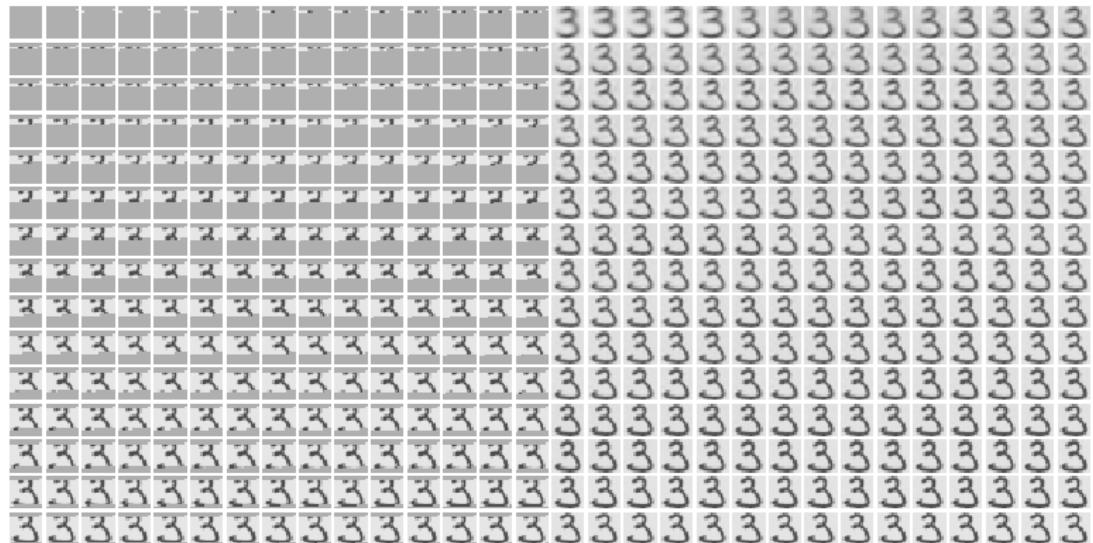
# RECONSTRUCTION

Using 100 axis dimensions



# RECONSTRUCTION

Using 225 axis dimensions



# RECONSTRUCTION



This is the **mean** (From centering  $\mathbb{X}$  :  $(\mathbb{X} - \bar{\mathbb{X}}) = UDV^\top$ )  
(that is, the origin of the PCA axis, or  $\bar{\mathbb{X}}$ )<sup>1</sup>

```
plot.digit(attributes(digitsCenter)$'scaled:center')
```

<sup>1</sup>Technically,  $\bar{X}_i$  for any  $i$

# Back to deep learning

# PCA

If we want to find the first  $K$  principal components, the relevant optimization program is:

$$\min_{\mu, (\lambda_i), V_K} \sum_{i=1}^n \|X_i - \mu - V_K \lambda_i\|^2$$

This representation is important

It shows that we are trying to reconstruct lower dimensional **representations** of the features

# PCA

$$\min_{\mu, (\lambda_i), V_K} \sum_{i=1}^n \|X_i - \mu - V_K \lambda_i\|^2$$

We can partially optimize for  $\mu$  and  $(\lambda_i)$  to find

- $\hat{\mu} = \bar{X}$
- $\hat{\lambda}_i = V_K^\top (X_i - \hat{\mu})$

We can find

$$\min_V \sum_{i=1}^n \|(X_i - \hat{\mu}) - VV^\top (X_i - \hat{\mu})\|^2$$

where  $V$  is constrained to be **orthogonal**

(This is the so called **Steifel manifold** of rank- $K$  orthogonal matrices)

The solution is given by the singular vectors  $V$

# Example: Facial recognition

# IMAGES

- There are 575 total images
- Each image is  $92 \times 112$  pixels and grey scale
- These images come from the Sheffield face database  
(See <http://www.face-rec.org/databases/> for this and other databases. See my rcode for how to read the images into R)

# FACES



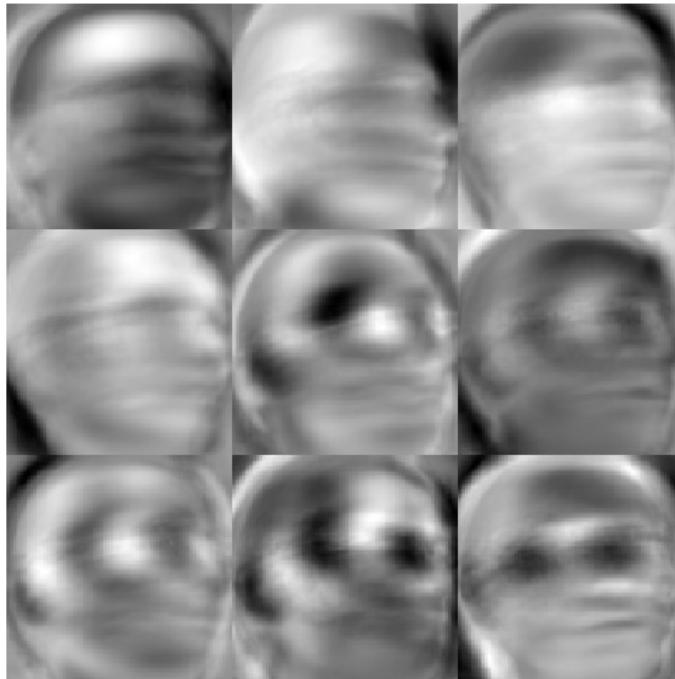
# FACES

Regardless of how you formulate the optimization problem for PCA, it can be done in R by:

```
svd.out = svd(scale(X, scale=F))  
pc.basis = svd.out$v  
pc.scores = X %*% pc.basis
```

Let's apply this to the faces

# FACES: PC BASIS

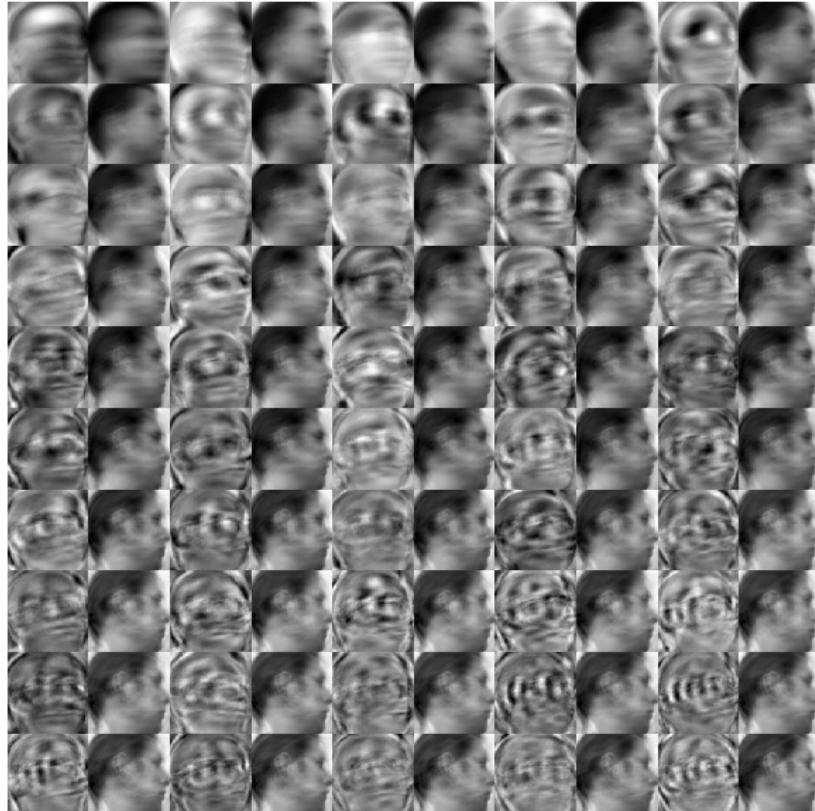


# FACES: PC PROJECTIONS

Varying levels of  $K$ :  $\tilde{\mathbb{X}} = \sum_{k=1}^K d_k u_k v_k^\top + \overline{\mathbb{X}}$



# FACES: PC PROJECTIONS AND BASIS



# Deep learning

# DEEP LEARNING: OVERVIEW

Neural networks are models for supervised learning

Linear combinations of features are fed through nonlinear functions repeatedly

At the top layer, the resulting latent factor is fed into a linear/logistic regression

# DEEP LEARNING: OVERVIEW

Deep learning is a new idea that has generated renewed interest in neural networks

Here, we wish to learn a hierarchy of features one level at a time, using

1. unsupervised feature learning to learn a new transformation at each level
2. which gets composed with the previously learned transformations

The top layer (which would be the output) is used to initialize a (supervised) neural network

# DEEP LEARNING: OVERVIEW

Traditionally, a neural net is fit to all **labelled** data in one operation, with weights randomly chosen near zero

Due to the nonconvexity of the objective function, the final solution can get 'caught' in poor local minima

**Deep learning** seeks to find a good starting value, while allowing for:

- ...modeling the joint distribution of the features separately
- ...use of unlabeled data (including the test features)

## EXAMPLES OF UNLABELED DATA

- **EMAILS:** For labelling **spam** or **not spam**, we might have a large number of emails where the label is known that we'd like to use somehow for classification
- **IMAGES:** For labelling **face** or **not face**, we could have a huge number of images which we don't know the content  
(For instance, all the frames of all the videos on youtube)

If we are trying to estimate the **Bayes' rule**, it tends to rely on a **conditional distribution**

$$\mathbb{P}(Y|X) = \frac{\mathbb{P}(X, Y)}{\mathbb{P}(X)}$$

We can use **unlabeled** data to get a better estimate of  $\mathbb{P}(X)$   
(And hence the Bayes' rule)

# Auto-encoders

# AUTO-ENCODERS

An **auto-encoder** generalizes PCA by specifying

- **FEATURE-EXTRACTING FUNCTION:** This function  $h : \mathbb{R}^P \rightarrow \mathbb{R}^K$  maps the features to a new representation and is also known as the **encoder**
- **RECONSTRUCTION FUNCTION:** This function<sup>2</sup>  $h^{-1} : \mathbb{R}^K \rightarrow \mathbb{R}^P$  is also known as the **decoder** and it maps the representation back into the original space

**GOAL:** Optimize any free parameters in the encoder/decoder pair that minimizes reconstruction error

---

<sup>2</sup>I've labeled this function  $h^{-1}$  to be suggestive, but I don't mean that  $h^{-1}(h(x)) = x$

# AUTO-ENCODER

Let  $W \in \mathbb{R}^{p \times K}$  (with  $K < p$ ) be a matrix of weights

Linear combinations of  $X$  are fed through a function  $\sigma$

$$h(X) = \sigma(W^\top X) \in \mathbb{R}^K$$

The **output layer** is then modeled as a linear combination of these inputs<sup>3</sup>

$$h^{-1}(h(X)) = Wh(X) = W\sigma(W^\top X) \in \mathbb{R}^p$$

---

<sup>3</sup>There is no restriction that the same matrix to be used in  $h$  and  $h^{-1}$ . Keeping them the same is known as **weight-tying**

# DEEP LEARNING

**REMINDER** Given inputs  $X_1, \dots, X_n$ , the PCA problem

$$\min_{(\lambda_i), V_K} \sum_{i=1}^n \|X_i - V_K \lambda_i\|^2$$

(Note I've implicitly subtracted of the mean)

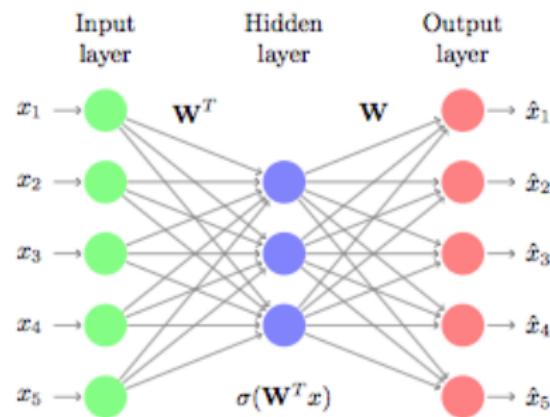
More general autoencoder: weight matrix  $W$  is estimated by solving the (non convex) optimization problem:

$$\min_{W \in \mathbb{R}^{p \times K}} \sum_{i=1}^n \|X_i - Wh(X_i)\|^2 = \min_{W \in \mathbb{R}^{p \times K}} \sum_{i=1}^n \|X_i - W\sigma(W^\top X_i)\|^2$$

(If  $\sigma(X) \equiv X$ , then we've recovered the PCA program)

# DEEP LEARNING SCHEMATIC

An autoencoder might look like:



(Image courtesy of Rob Tibshirani)

# NEURAL NETWORKS: REPRESENTATIONS

**IMPORTANT:** Neural networks themselves create (supervised) representations

Compare:

$$\alpha^\top X \Leftrightarrow W^\top X$$

# NEURAL NETWORKS: REPRESENTATIONS

Return to the US crime data

Run a single layer, two hidden unit neural network  
(with sigmoid activation function)

```
Y = subset(UScrime,select=y,drop=T)
X = scale(subset(UScrime,select=-y))
X = as.data.frame(X)
names(X) = names(subset(UScrime,select=-y))
model.out = as.formula(paste("Y ~ ",paste(names(X),
                                              collapse= '+'))))
nn.out = neuralnet(model.out,data=UScrime, hidden=2)

W = nn.out$weights[[1]][[1]]
plot(W[-1,],type='n',xlab='W_1',ylab='W_2')
text(W[-1,],names(X),cex=.75)
```

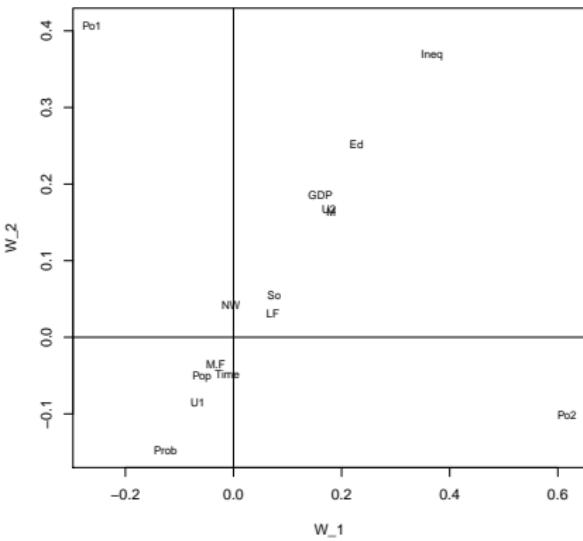
# NEURAL NETWORKS: REPRESENTATIONS

The interpretation is that each latent variable  $Z_k = \sigma(\alpha_k^\top X)$  is a **neuron** that is tuned to detect a particular type of structure

features that “positive” signs in the representations indicate the neuron is “tuned” to that signal-type

Note that this isn’t a derivative or importance-based interpretation

# NEURAL NETWORKS: REPRESENTATIONS



M: % males aged 14-24.

So: indicator for a Southern state.

Ed: mean years of schooling.

Po1: police 1960.

Po2: police 1959.

LF: labor force participation rate.

M.F.: # of males per 1000 females.

Pop: state population.

NW: # of non-whites per 1000 people.

U1: unemployment rate of urban males.

U2: unemployment rate of urban females.

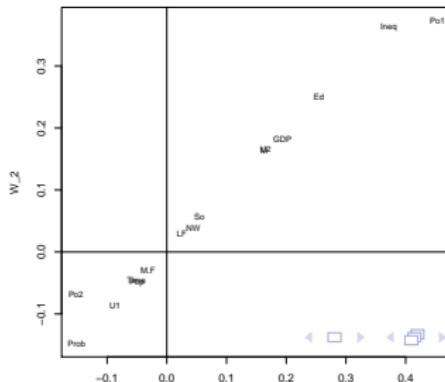
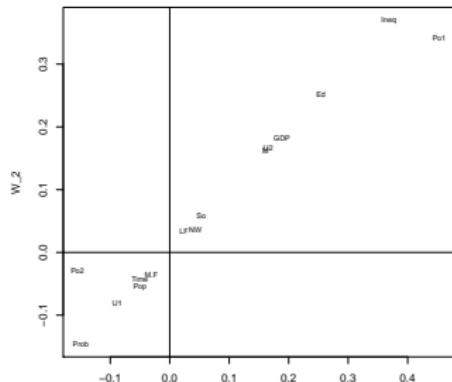
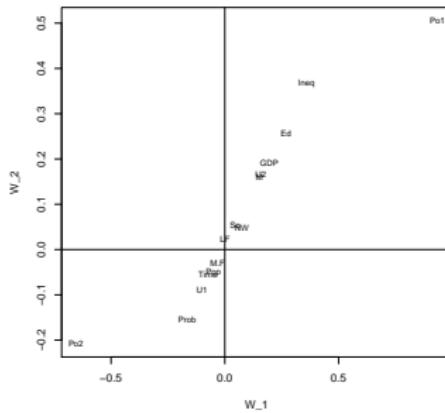
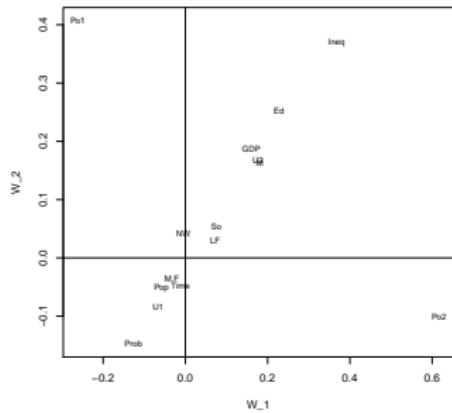
GDP: gross domestic product per capita.

Ineq: income inequality.

Prob: probability of imprisonment.

Time: average time served in state prison.

# NEURAL NETWORKS: REPRESENTATIONS



# DEEP LEARNING

The following is a brief overview of NNs from some researchers at Google

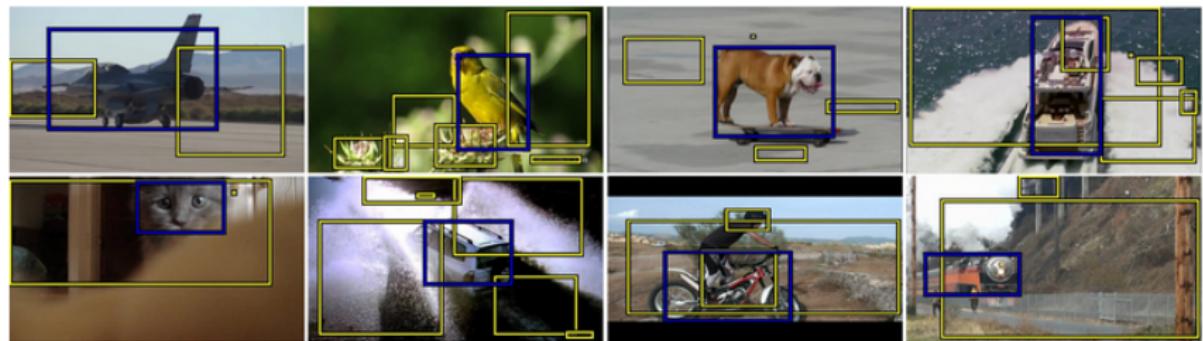
(Le, Ranzato, Monga, Devin, Chen, Dean, Ng (2012))

It has about 1 billion trainable parameters and uses advanced parallelism to make computation feasible

It also uses a decoupled encoder-decoder pair, plus regularization and a linear activation

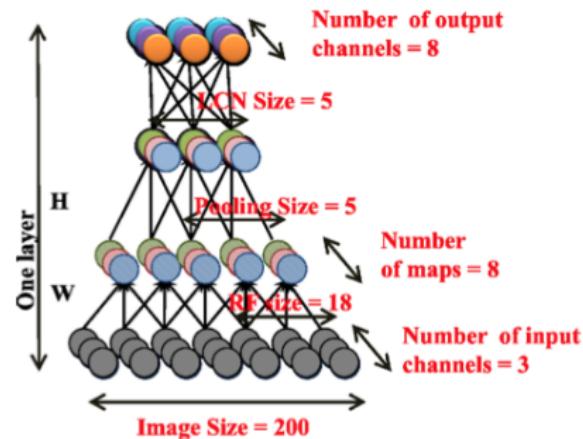
(This means that we write the representation as  $W_O W_I^\top X$ , where  $W_O \neq W_I$ )

# DEEP LEARNING: DATA



# DEEP LEARNING SCHEMATIC

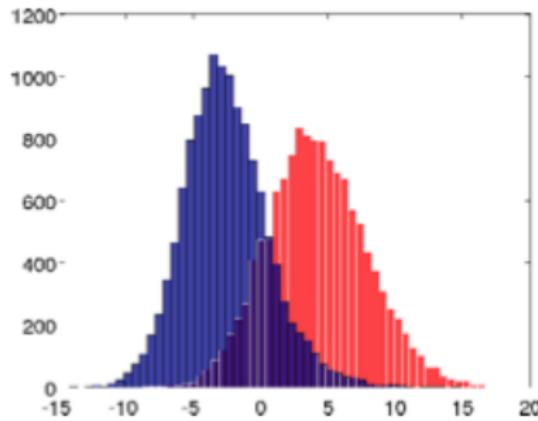
A representation of their implementation



## DEEP LEARNING RESULTS

If we look at every neuron (that is, hidden unit) in the network and take the output for a given body of test images

Maximize the classification rate of taking the  $\text{sign}(\cdot)$ , they find:



# DEEP LEARNING RESULTS

The test images with maximum activation of that optimal neuron



# DEEP LEARNING RESULTS

Finding the pixel wise maximizing input:

