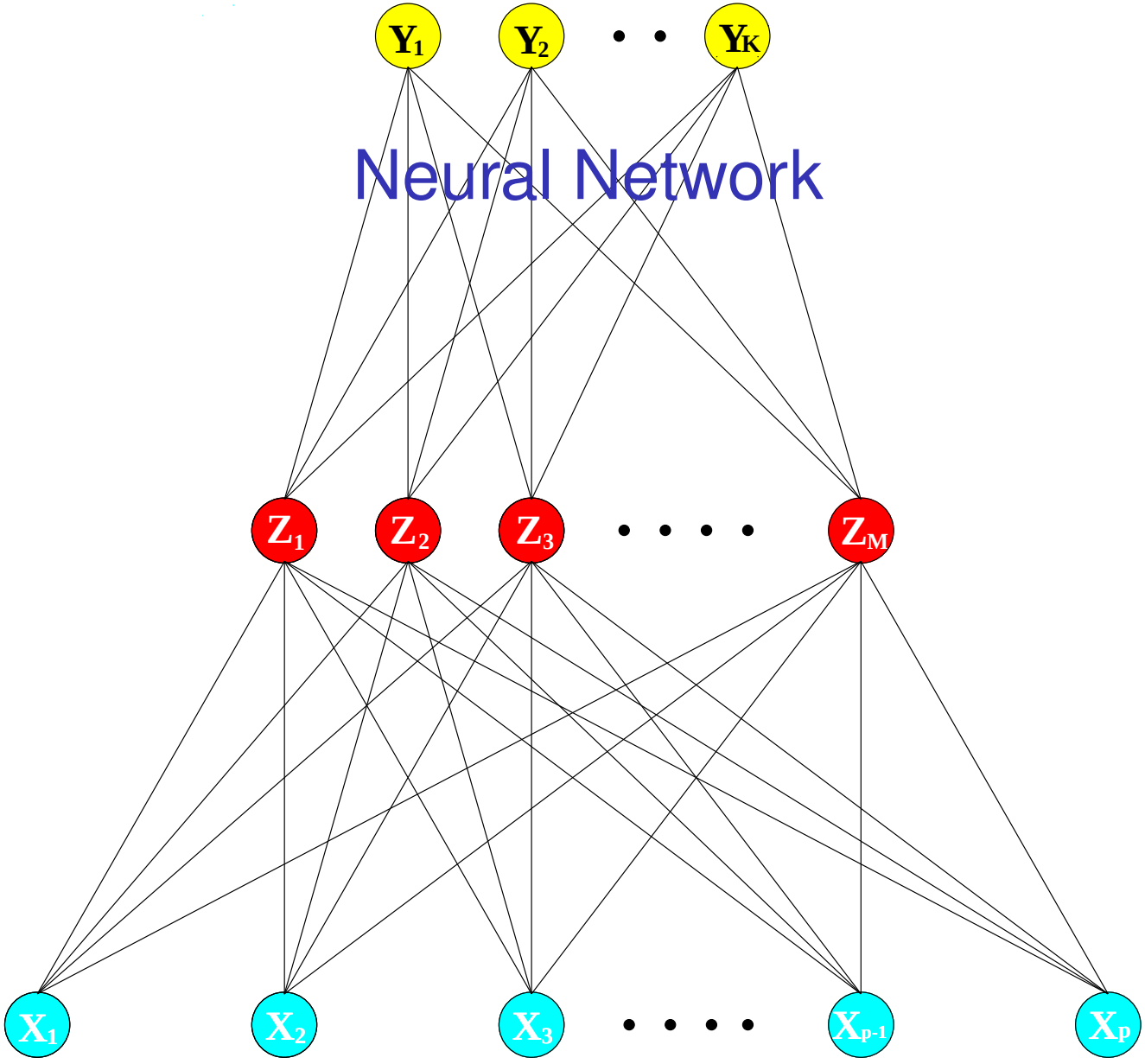
Neural networks and Deep Learning[1]

Germán G. Creamer

[1] Sources: Introduction to Statistical Learning with R, Elements of Statistical Learning, Goodfellow - Deep Learning, and Thomas Lonon's notes



Single hidden layer, feed-forward neural network

History of neural network and deep learning

- 1940's theoretical birth of neural networks McCulloch & Pitts (1943), Hebb (1949): Perceptron (linear model)
- 1950's & 1960's optimistic development using computer models Minsky (50's), Rosenblatt (60's)
- 1970's DEAD. Minsky & Papert showed serious limitations: linear separability problem

History of neural network and deep learning

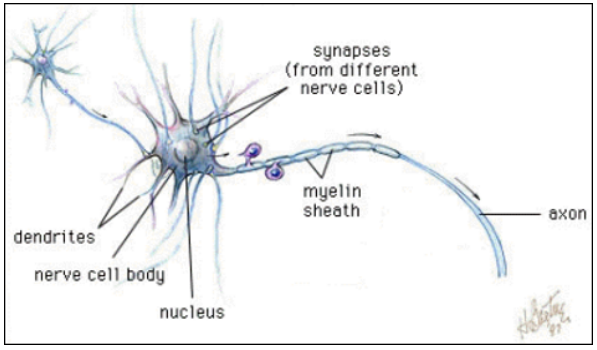
- 1980s: Multi-layer perceptron: Do not have significant difference from DNN today
- 1986: Backpropagation: Usually more than 3 hidden layers is not helpful
- 1989: 1 hidden layer is "good enough", why deep learning?
- 2006: Restricted Boltzmann machine initialization
- 2009: GPU
- 2011: Deep learning (DL): Start to be popular in speech recognition
- 2012: DL win ILSVRC image competition
- 2015.2: Image recognition surpassing human-level performance
- 2016.3: Alpha GO beats Lee Sedol (Google DeepMind Challenge Match)
- 2016.10: Speech recognition system as good as humans
- 2017 - 2021: DL is everywhere

Neural Networks

- Humans lack the speed & memory of computers; yet humans are capable of complex reasoning/action.
- Maybe our brain architecture is well-suited for certain tasks.

General brain architecture:

- Many (relatively) slow neurons, interconnected
- Dendrites serve as input devices (receive electrical impulses from other neurons)
- Cell body "sums" inputs from the dendrites (possibly inhibiting or exciting)
- If sum exceeds some threshold, the neuron fires an output impulse along axon



Neural Networks

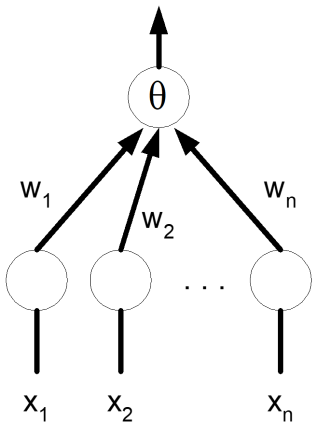
Connectionist models are based on the brain metaphor vs. emergent models (evolution):

- Large number of simple, neuron-like processing elements
- Large number of weighted connections between neurons. Note: the weights encode information, not symbols!
- Parallel, distributed control
- Emphasis on learning

Perceptron

McCulloch & Pitts (1943) described an artificial neuron

1. Inputs are either electrical impulse (1) or not (0)
2. Each input has a weight associated with it
3. The activation function multiplies each input value by its weight: if the sum of the weighted inputs $\geq \theta$, then the neuron fires (returns 1), else doesn't fire (returns 0)



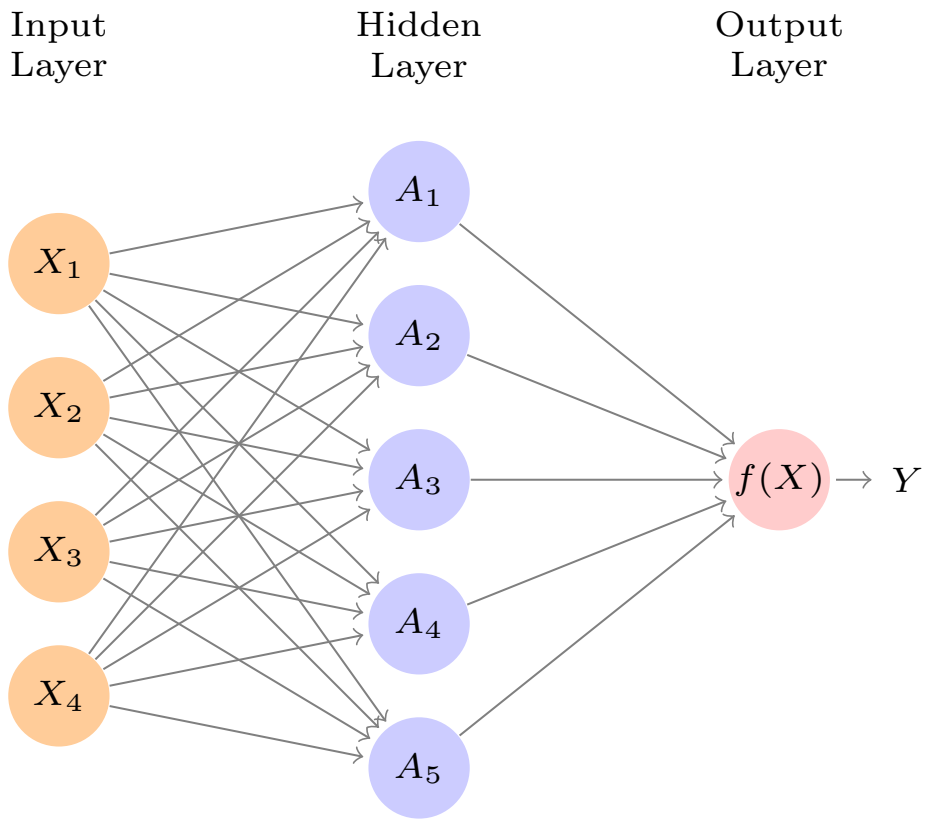
Neural Network with a Single Hidden Layer

Limitation of perceptron: linear model
Solution: build multilayer networks with hidden layers.
A neural network is a two-stage regression or classification model, typically represented by a network diagram.

For regression, typically $K = 1$ and there is only one output at the top (Y_1)

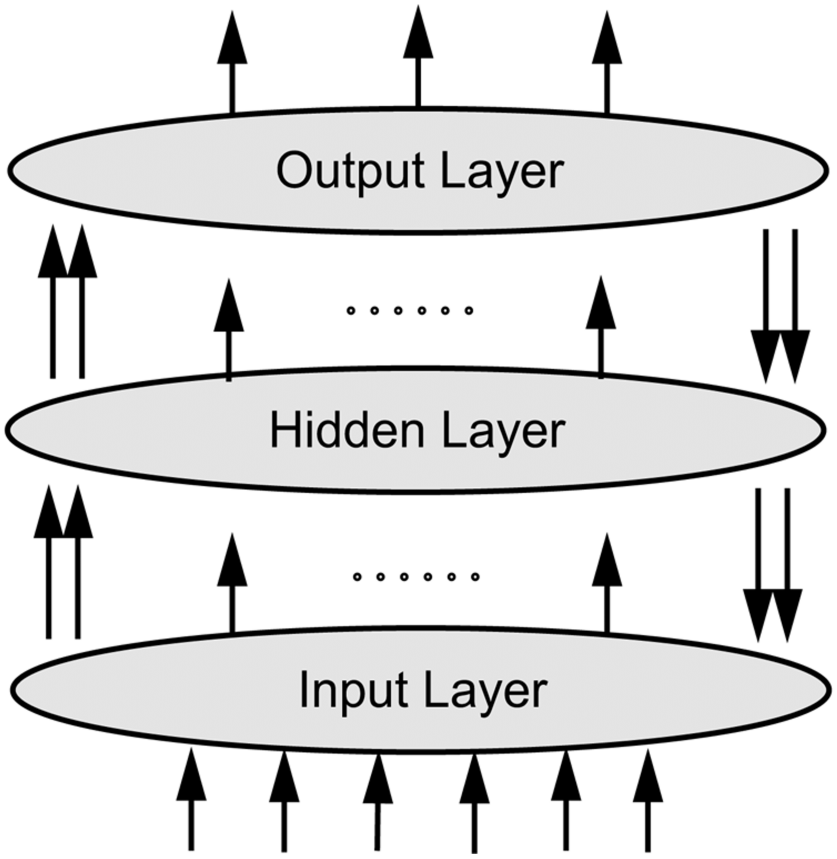
For K-class classification, there are K units at the top, with the k^{th} unit modeling the probability of class k.

Neural Network with a Single Hidden Layer



Backpropagation

Forward Network Activation



Backwards Error Propagation

Backpropagation

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji} \tag{T4.5}$$

Backpropagation

$x_1 = 1$

$x_2 = 0$

$x_{01} = 1$

$\eta = 3$ (learning rate)

Forward:

O_k = output: $f(net) = 0.5386$

$w_{0d} = 0.1$, $net = 0.549 * 0.1 + 1 * 0.1 = 0.154$

$w_{1d} = 0.1$

$net = \sum w_i x_i = 0.2$

$w_{01} = 0.1$, $w_{11} = 0.1$, $w_{21} = 0.1$

$f(net) = 0.549 = O_h$

1 1 0

t_k: target output = 1

Backward:

Error term of output unit k:

$$\delta_k = O_k (1 - O_k)(t_k - O_k) = 0.538(1 - 0.538)(1 - 0.538) = 0.114643$$

Error term of hidden unit h:

$$\delta_h = O_h (1 - O_h) \sum_{k \in outputs} w_{kl} \delta_k = 0.549(1 - 0.549) * 0.1 * 0.114643 = 0.002838$$

Update weights: $w_{ji} = w_{ji} + \Delta w_{ji}$ where $\Delta w_{ji} = \eta \delta_j x_{ji}$

$w_{11} = 0.1 + 3 * 0.002838 * 1 = 0.1085$

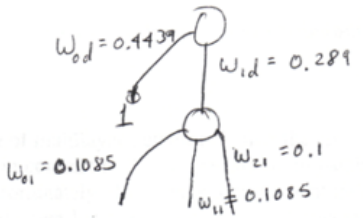
$w_{11} + \eta * \delta_h * x_1 =$

$w_{21} = 0.1 + 3 * 0.002838 * 0 = 0.1$

$w_{01} = 0.1 + 3 * 0.002838 * 1 = 0.1085$

$w_{1d} = w_{1d} + \eta * \delta_k * x_{1d} = 0.1 + 3 * 0.114643 * 0.5498 = 0.289$

$w_{0d} = w_{0d} + \eta * \delta_k * x_{0d} = 0.1 + 3 * 0.114643 * 1 = 0.4439$



Features Z_m are created from linear combinations of the inputs and the target Y_k is modeled as a function of linear combinations of the Z_m.

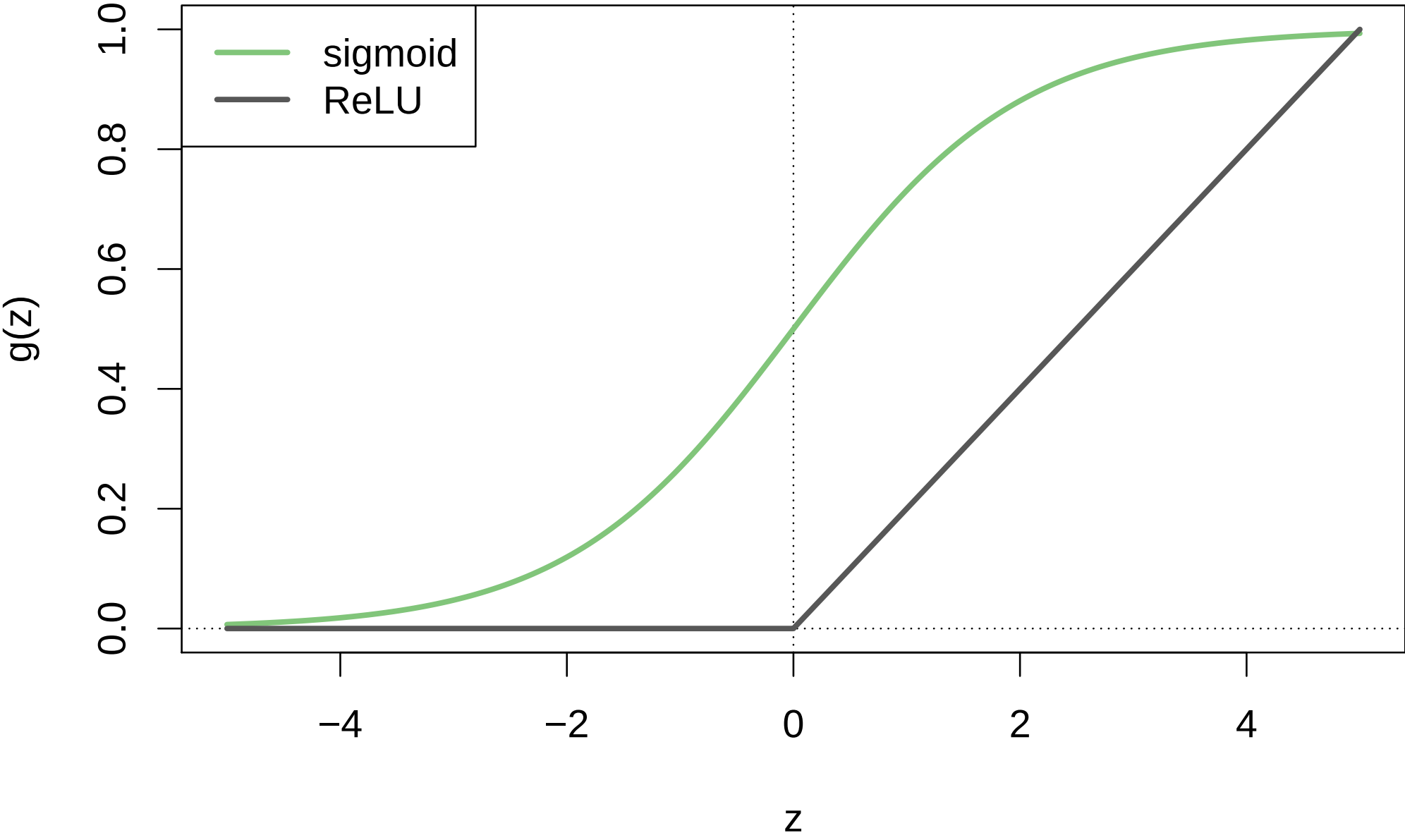
$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, \ldots, M$$
$$T_k = \beta_{0k} + \beta_k^T Z, k = 1, \ldots, K$$
$$f_k(X) = g_k(T), k = 1, \ldots, K$$

where $Z = (Z_1, Z_2, \ldots, Z_M)$ and $T = (T_1, T_2, \ldots, T_K)$
sigmoid was the most popular activation function $\sigma(\nu)$:

$$\sigma(\nu) = \frac{1}{1 + e^{-\nu}}$$

Current choice in neural networks is ReLU (rectified linear ReLU unit) activation function which can be computed and stored more efficiently:

$$\sigma(\nu) = (\nu)_+ = \begin{cases} 0 : \nu < 0 \\ \nu : \textit{otherwise} \end{cases}$$


sigmoid
ReLU
g(z)
0.0
0.2
0.4
0.6
0.8
1.0
−4
−2
0
2
4
z

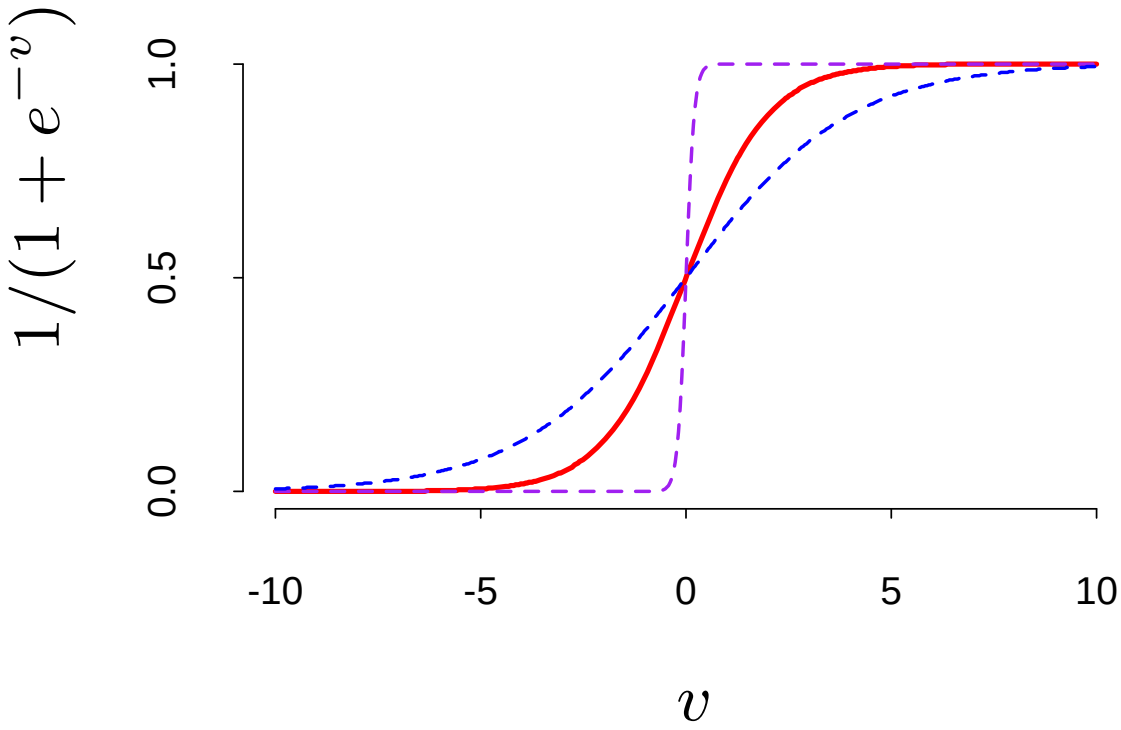


FIGURE 11.3. *Plot of the sigmoid function* $\sigma(v) = 1/(1+\exp(-v))$ *(red curve), commonly used in the hidden layer of a neural network. Included are* $\sigma(sv)$ *for* $s = \frac{1}{2}$ *(blue curve) and* $s = 10$ *(purple curve). The scale parameter* s *controls the activation rate, and we can see that large* s *amounts to a hard activation at* $v = 0$*. Note that* $\sigma(s(v - v_0))$ *shifts the activation threshold from* 0 *to* v_0*.*

The output function $g_k(T)$ does a final transformation of the vector T. For regression this is typically $g_k(T) = T_k$.

For K-classification we use the *softmax* function

$$g_k(T) = P(k|X) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$$

These Z_m are hidden units that are expressed as a basis expansion of the original inputs X.
$g_k(T)$ for each class k behaves like probabilities (non-negative and sum to one). Model estimates a probability for each of the k classes. The classifier selects the class with the highest probability.

For a loss function L we can define:

$$L(f) = \sum_{i=1}^{N} L(y_i, f(x_i))$$

The goal is then to determine:

$$\hat{\mathbf{f}} = \arg\min_{\mathbf{f}} L(\mathbf{f})$$

where $\mathbf{f} = \{f(x_1), \ldots, f(x_N)\}^T$

We can iterate this expression by expressing it as a sum of component vectors

$$\mathbf{f}_M = \sum_{m=1}^{M} \mathbf{h}_m, \mathbf{h}_m \in \mathbb{R}^N$$

where $\mathbf{f}_0 = \mathbf{h}_0$

Each successive $\mathbf{f}_m$ is induced based on the previous vector $\mathbf{f}_{m-1}$. These methods differ based on the computations for the increment vector $\mathbf{h}_m$.

Gradient Descent

If we let $\mathbf{h}_m$ be defined as:

$$\mathbf{h}_m = -\gamma_m \mathbf{g}_m$$

where

$$g_{im} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x_i)=f_{m-1}(x_i)}$$

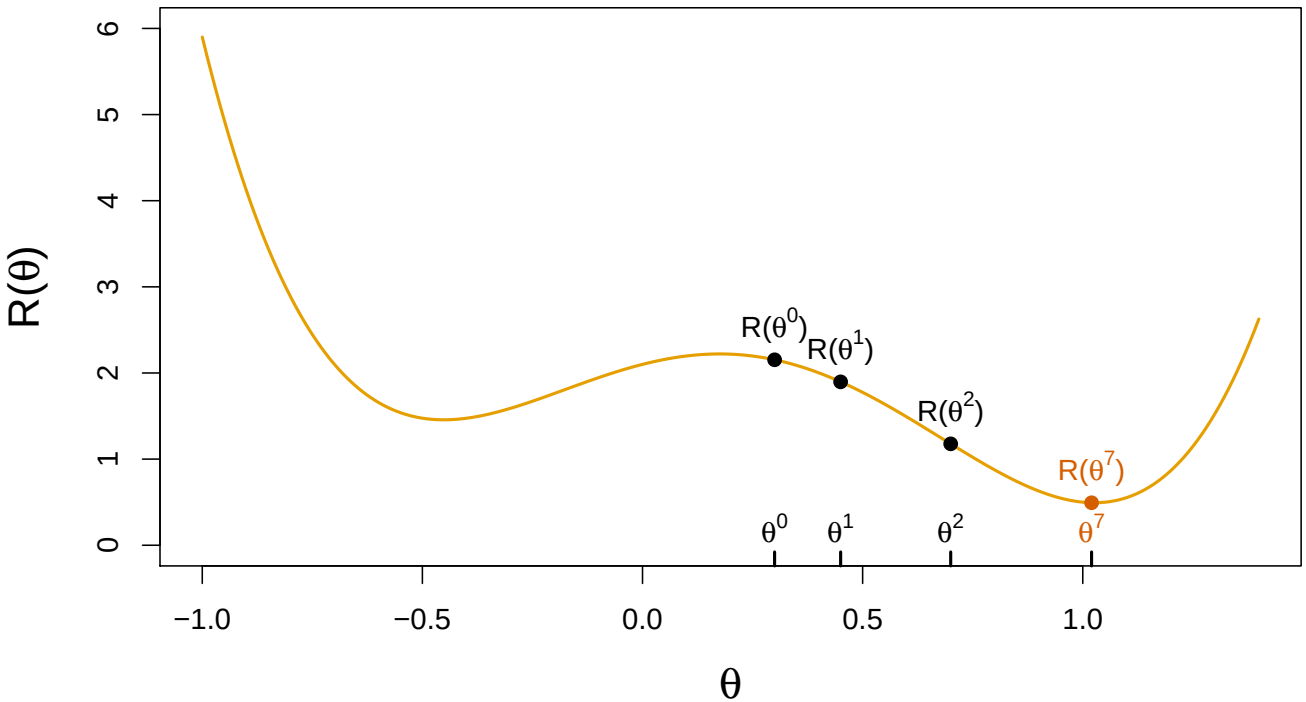
Then this is the gradient descent and we have:

$$\mathbf{f}_m = \mathbf{f}_{m-1} - \gamma_m \mathbf{g}_m$$

This γ_m is the step length and can be set up in a variety of ways.

Non Convex Functions and Gradient Descent

Let $R(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$ with $\theta = (\{w_k\}_1^K,\ \beta)$.



1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$.
2. Iterate until the objective $R(\theta)$ fails to decrease:
   (a) Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ *reduces* the objective; i.e. $R(\theta^{t+1}) < R(\theta^t)$.
   (b) Set $t \leftarrow t + 1$.

Fitting Neural Networks

In the neural network, we have unknown parameters which we denote *weights*. We label the set of these weights θ which consist of:

$\{\alpha_{0m}, \alpha_m; m = 1, 2, \ldots, M\}$	$M(p+1)$weights
$\{\beta_{0k}, \beta_k; k = 1, 2, \ldots, K\}$	$K(M+1)$weights

For regression, we use sum-of-squared errors as our measure of fit

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2$$

For classification we use either squared error or cross-entropy

$$R(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(x_i)$$

and the corresponding classifier is $G(x) = \arg\max_k f_k(x)$

We will minimize this $R(\theta)$ through gradient descent, called *back-propagation*. The steps involved for the squared error loss are as follows:

Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$ with $z_i = (z_{1i}, z_{21}, \ldots, z_{Mi})$. Then:

$$\begin{aligned} R(\theta) &= \sum_{i=1}^{N} R_i \\ &= \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2 \end{aligned}$$

This has derivatives:

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi}$$

$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{i\ell}$$

so the gradient descent update at $r+1$ has the form:

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}}$$

$$\alpha_{m\ell}^{(r+1)} = \alpha_{m\ell}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{m\ell}^{(r)}}$$

where γ_r is the *learning rate*

We can now write the partials as:

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi}$$
$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = s_{mi} x_{i\ell}$$

These quantities δ_{ki} and s_{mi} are "errors" from the forward pass and are used as inputs of the *back-propagation pass*:

$$\delta_{ki} = -2(y_{ik} - \hat{f}_k(x_i)) g_k'(\beta_k^{(r)T} z_i)$$

$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}$$

where $\hat{f}_k(x_i)$ is the output of the forward pass

Forward Pass

Use current weights (the α and β terms) in the equations:

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, \ldots, M$$
$$T_k = \beta_{0k} + \beta_k^T Z, k = 1, \ldots, K$$
$$f_k(X) = g_k(T), k = 1, \ldots, K$$

To determine the new values for $\hat{f}_k(x_i)$

Backward Pass

Using the $\hat{f}_k(x_i)$ terms from the forward pass and the weights for step r we can determine the new weights by using

$$\delta_{ki} = -2(y_{ik} - \hat{f}_k(x_i))g'_k(\beta_k^{(r)T} z_i)$$

$$s_{mi} = \sigma'(\alpha_m^{(r)T} x_i) \sum_{k=1}^{K} \beta_{km}^{(r)} \delta_{ki}$$

Our updated weights are now:

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \delta_{ki} z_{mi}$$

$$\alpha_{m\ell}^{(r+1)} = \alpha_{m\ell}^{(r)} - \gamma_r \sum_{i=1}^{N} s_{mi} x_{i\ell}$$

Repeat until termination condition is met: output convergence.

Classification Modification

For classification, we use a different error than squared error. For the deviance error measure, we have for the k^{th} component

$$\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} = \mathbb{I}_{\{y_i = \mathcal{G}_k\}} - p_k(x_i)$$

where

$$p_k(x_i) = \frac{e^{f_k(x_i)}}{\sum_{j=1}^{K} e^{f_j(x_i)}}$$

Issues in Training Neural Networks

Starting Values

Overfitting a method to avoid this is *weight decay* which is analogous to ridge regression for linear models. We add a penalty to the error function $R(\theta) + \lambda J(\theta)$ where

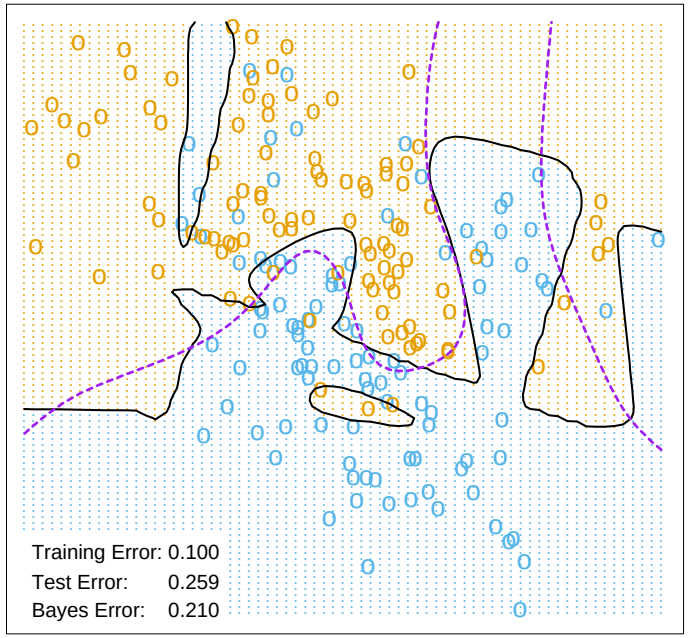
$$J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{m\ell} \alpha_{m\ell}^2$$

and $\lambda \geq 0$ is a tuning parameter. Or we could express the penalty as

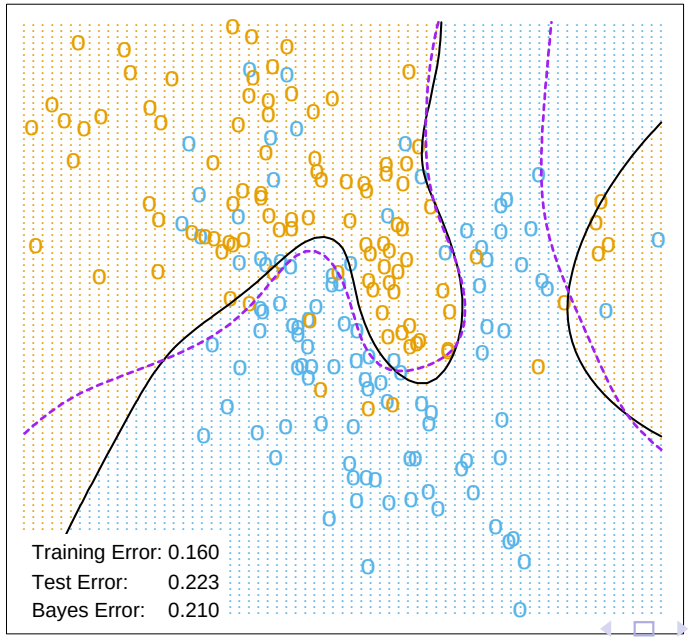
$$J(\theta) = \sum_{km} \frac{\beta_{km}^2}{1 + \beta_{km}^2} + \sum_{m\ell} \frac{\alpha_{m\ell}^2}{1 + \alpha_{m\ell}^2}$$

known as *weight elimination* penalty

Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02



Issues in Training Neural Networks

Scaling of the Inputs: mean zero and standard deviation one.

Number of Hidden Units and Layers: better to have too many hidden units than too few to have more flexibility. Control with regularization. Number of hidden layers is based on expert knowledge and experimentation.

Multiple Minima: error function R(θ) is nonconvex, possessing many local minima:

1. Try a number of random starting configurations, and choose the solution giving lowest (penalized) error.
2. Average predictions over the collection of networks as the final prediction.
3. Bagging: averages predictions of networks training from randomly perturbed versions of the training data.

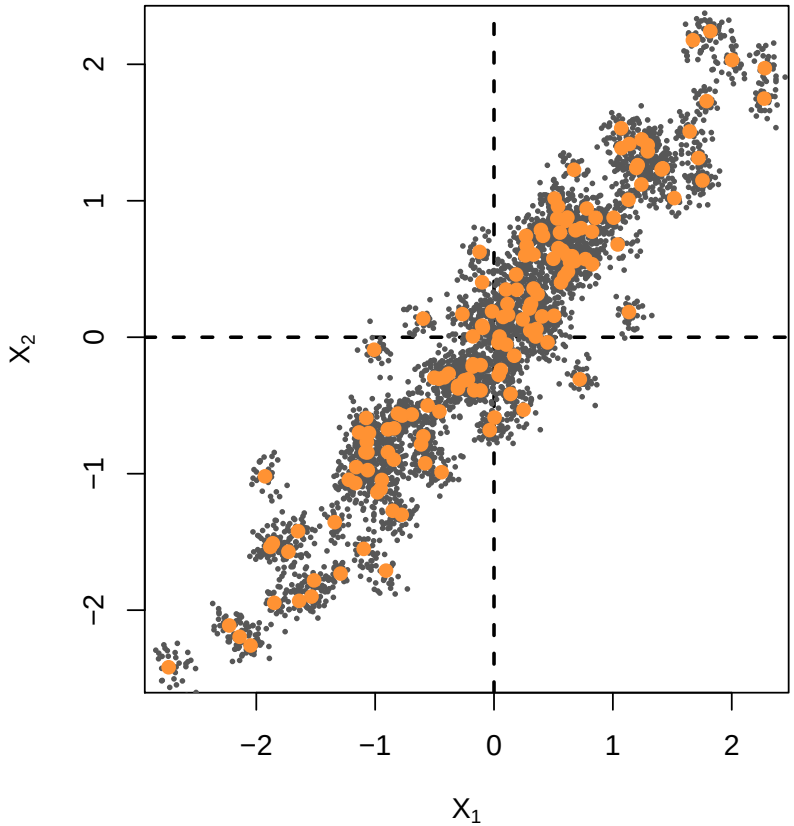
Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.
- *Stochastic gradient descent*. Rather than compute the gradient using *all* the data, use a small *minibatch* drawn at random at each step. E.g. for `MNIST` data, with $n = 60K$, we use minibatches of 128 observations.
- An *epoch* is a count of iterations and amounts to the number of minibatch updates such that n samples in total have been processed; i.e. $60K/128 \approx 469$ for `MNIST`.
- *Regularization*. Ridge and lasso regularization can be used to shrink the weights at each layer. Two other popular forms of regularization are *dropout* and *augmentation*, discussed next.

Dropout Learning

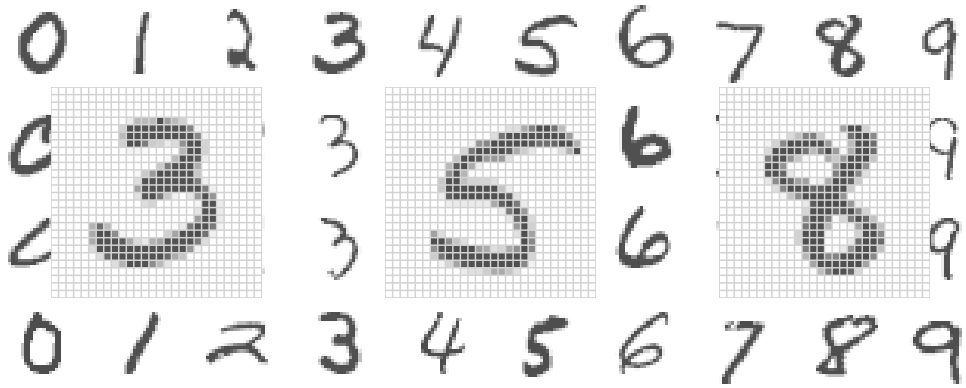
- At each SGD update, randomly remove units with probability ϕ, and scale up the weights of those retained by $1/(1-\phi)$ to compensate.
- In simple scenarios like linear regression, a version of this process can be shown to be equivalent to ridge regularization.
- As in ridge, the other units *stand in* for those temporarily removed, and their weights are drawn closer together.
- Similar to randomly omitting variables when growing trees in random forests (Chapter 8).

Ridge and Data Augmentation



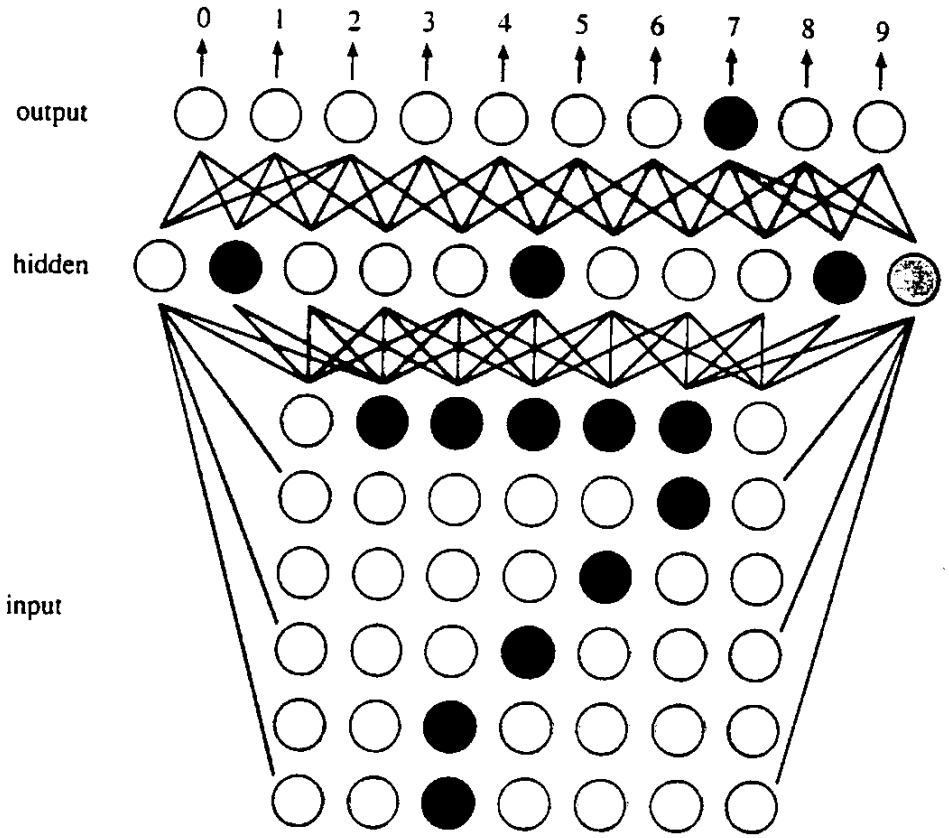
- Make many copies of each (x_i, y_i) and add a small amount of Gaussian noise to the x_i — a little cloud around each observation — but *leave the copies of y_i alone!*
- This makes the fit robust to small perturbations in x_i, and is equivalent to ridge regularization in an OLS setting.

Multilayer Neural Network

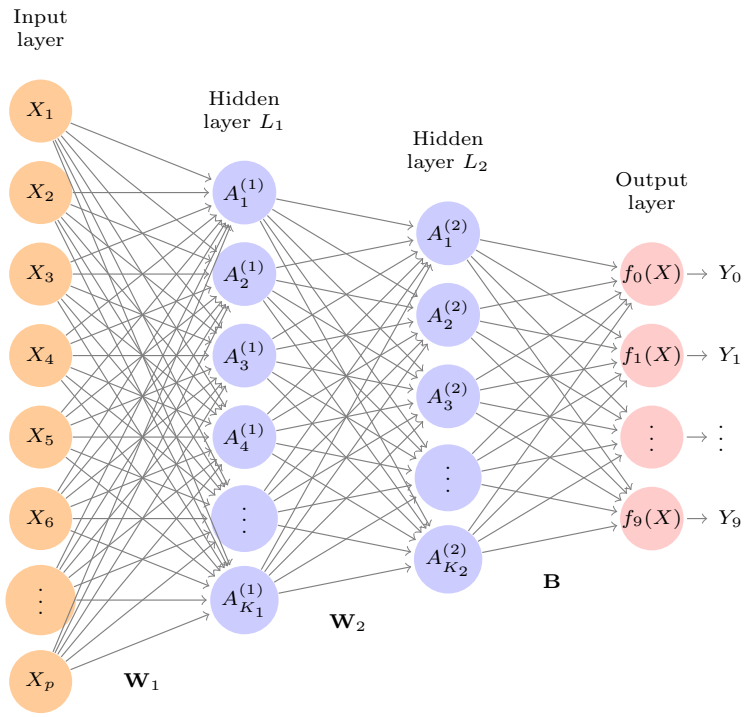


Handwritten digits from the MNIST corpus. Each grayscale image has 28 x 28 pixels, each of which is an eight bit number $(0-255)$ which represents how dark that pixel is. The first 3, 5, and 8 are enlarged to show their 784 individual pixel values.

Multilayer Neural Network

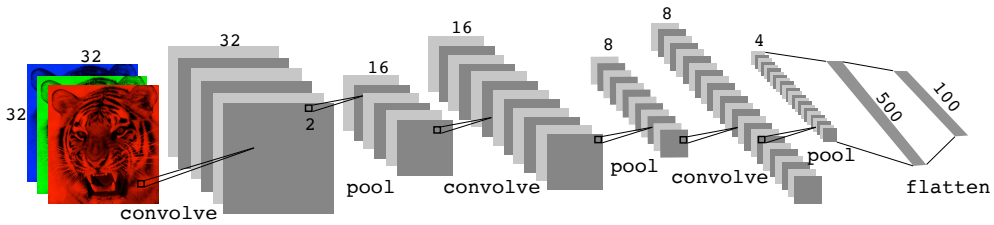


Neural Network with 2 Hidden Layers for MNIST



Input layer: p = 784 units; 2 hidden layers K1 = 256 and K2 = 128 units; output: 10 units. Total: 235,146 parameters

Deep Learning: Convolutional Neural Network



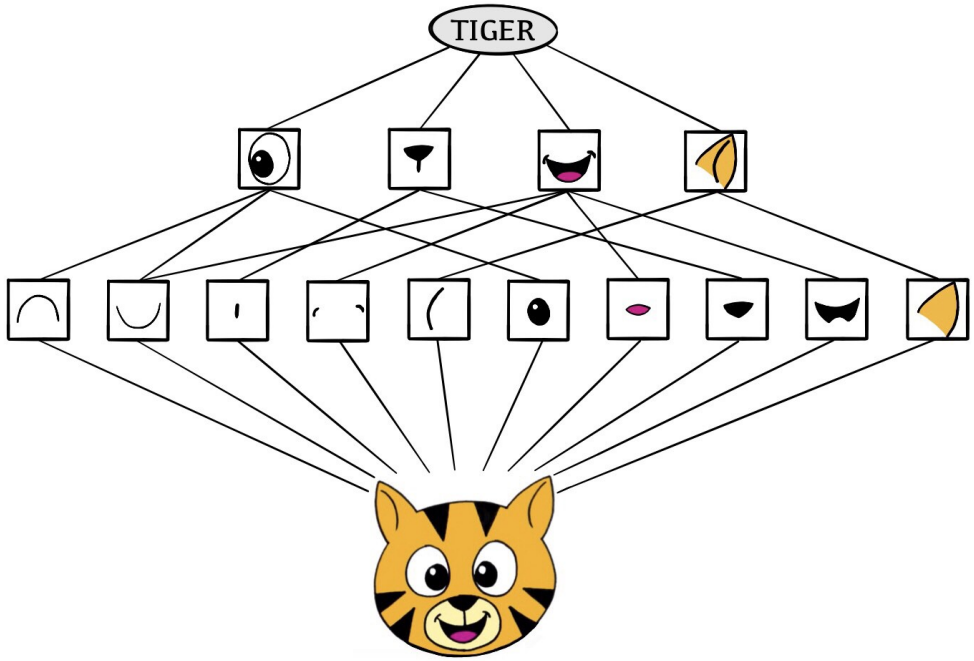
▸ Vision - Dance

Deep Learning: Convolutional Neural Network

Scale up neural networks to process very large images / video sequences:
- Sparse connections
- Parameter sharing
- Automatically generalize across spatial translations of inputs
- Applicable to any input that is laid out on a grid (1-D, 2-D, 3-D, ...)

Replace matrix multiplication in neural nets with convolution:
- Everything else stays the same
- Maximum likelihood
- Back-propagation
- etc.

How CNNs Work



- The CNN builds up an image in a hierarchical fashion.
- Edges and shapes are recognized and pieced together to form more complex shapes, eventually assembling the target image.
- This hierarchical construction is achieved using *convolution* and *pooling* layers.

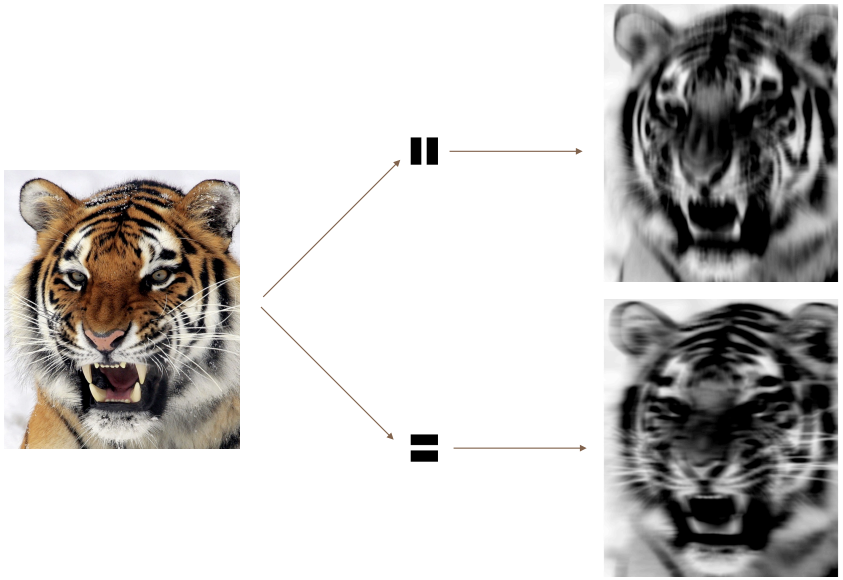
Convolution Filter

$$\text{Input Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix} \quad \text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$

- The filter is itself an image, and represents a small shape, edge etc.
- We slide it around the input image, scoring for matches.
- The scoring is done via *dot-products*, illustrated above.
- If the subimage of the input image is similar to the filter, the score is high, otherwise low.
- The filters are *learned* during training.

Convolution Example



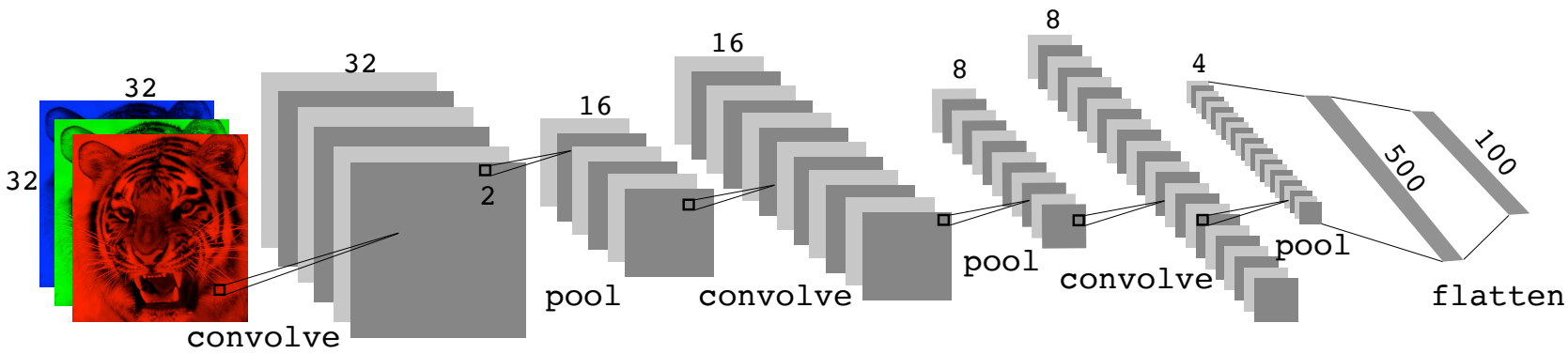
- The idea of convolution with a filter is to find common patterns that occur in different parts of the image.
- The two filters shown here highlight vertical and horizontal stripes.
- The result of the convolution is a new feature map.
- Since images have three colors channels, the filter does as well: one filter per channel, and dot-products are summed.
- The weights in the filters are *learned* by the network.

Pooling

$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

- Each non-overlapping 2×2 block is replaced by its maximum.
- This sharpens the feature identification.
- Allows for locational invariance.
- Reduces the dimension by a factor of 4 — i.e. factor of 2 in each dimension.

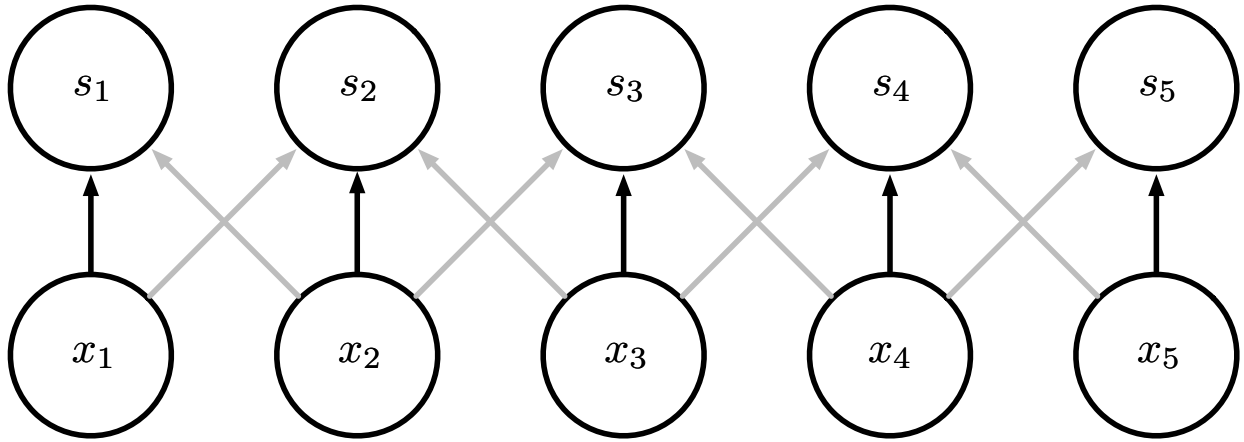
Architecture of a CNN



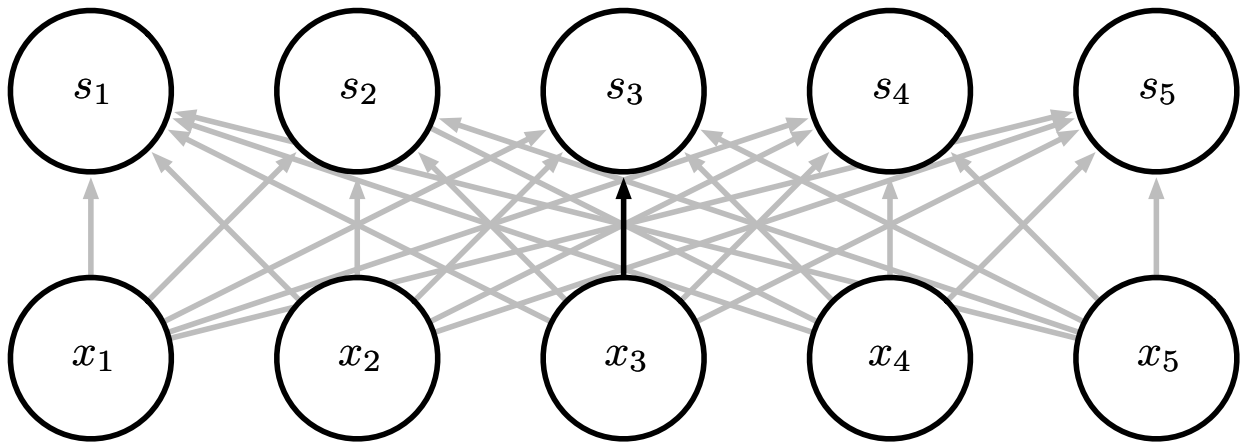
- Many convolve + pool layers.
- Filters are typically small, e.g. each channel 3×3.
- Each filter creates a new channel in convolution layer.
- As pooling reduces size, the number of filters/channels is typically increased.
- Number of layers can be very large. E.g. `resnet50` trained on `imagenet` 1000-class image data base has 50 layers!

Parameter Sharing

Convolution shares the same parameters across all spatial locations

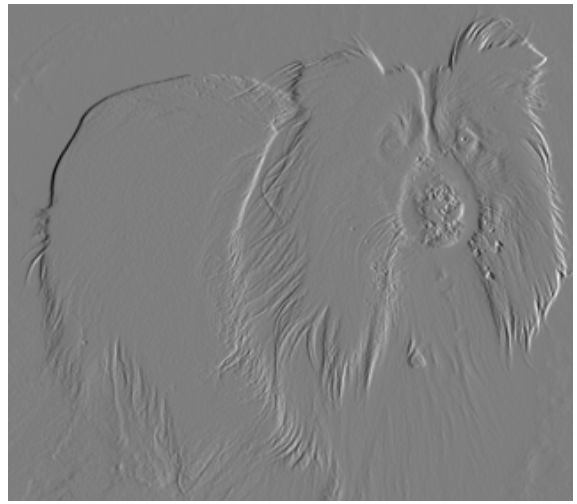


Traditional matrix multiplication does not share any parameters



Edge Detection by Convolution

Input



Output

1	-1

Kernel

Efficiency of Convolution

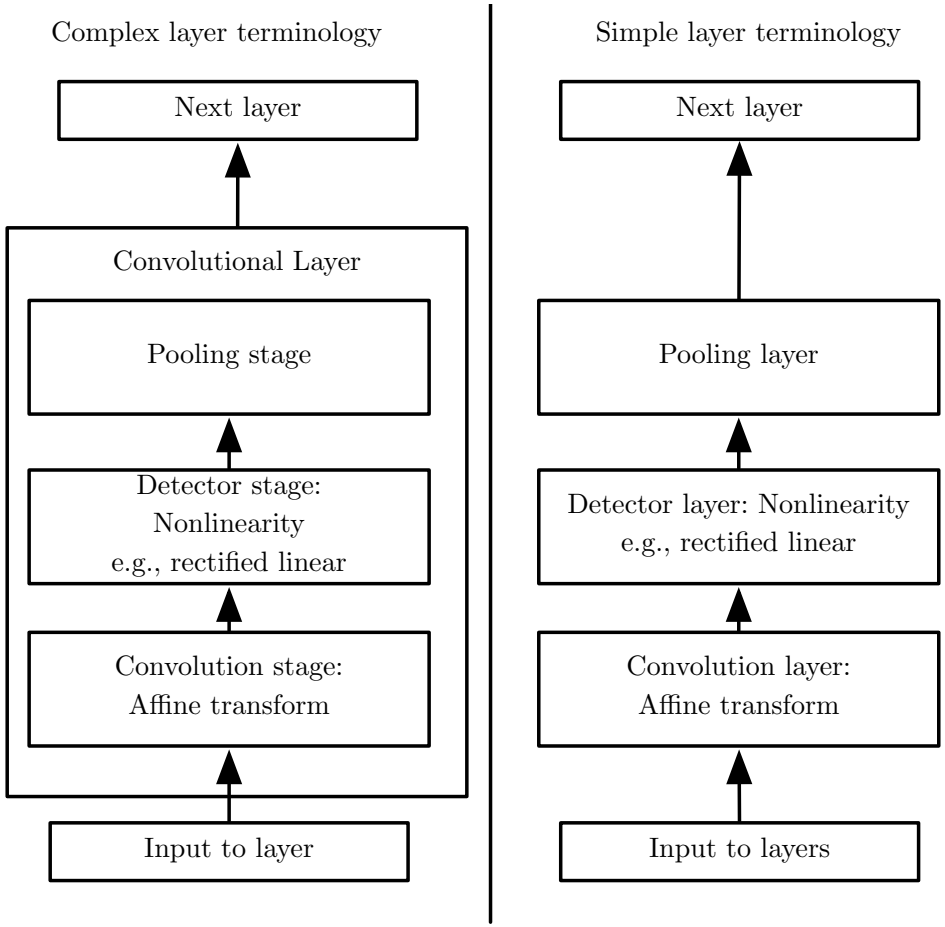
Input size: 320 by 280

Kernel size: 2 by 1

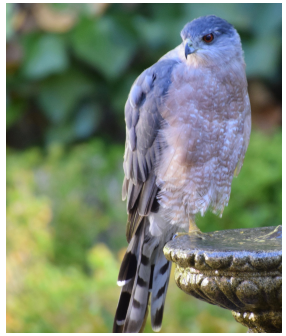
Output size: 319 by 280

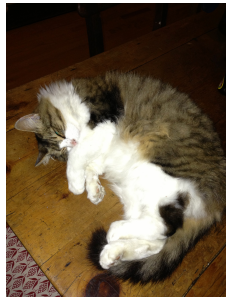
	Convolution	Dense matrix	Sparse matrix
Stored floats	2	319*280*320*280 $>$ 8e9	2*319*280 = 178,640
Float muls or adds	319*280*3 = 267,960	$>$ 16e9	Same as convolution (267,960)

Convolutional Network Components



Using Pretrained Networks to Classify Images

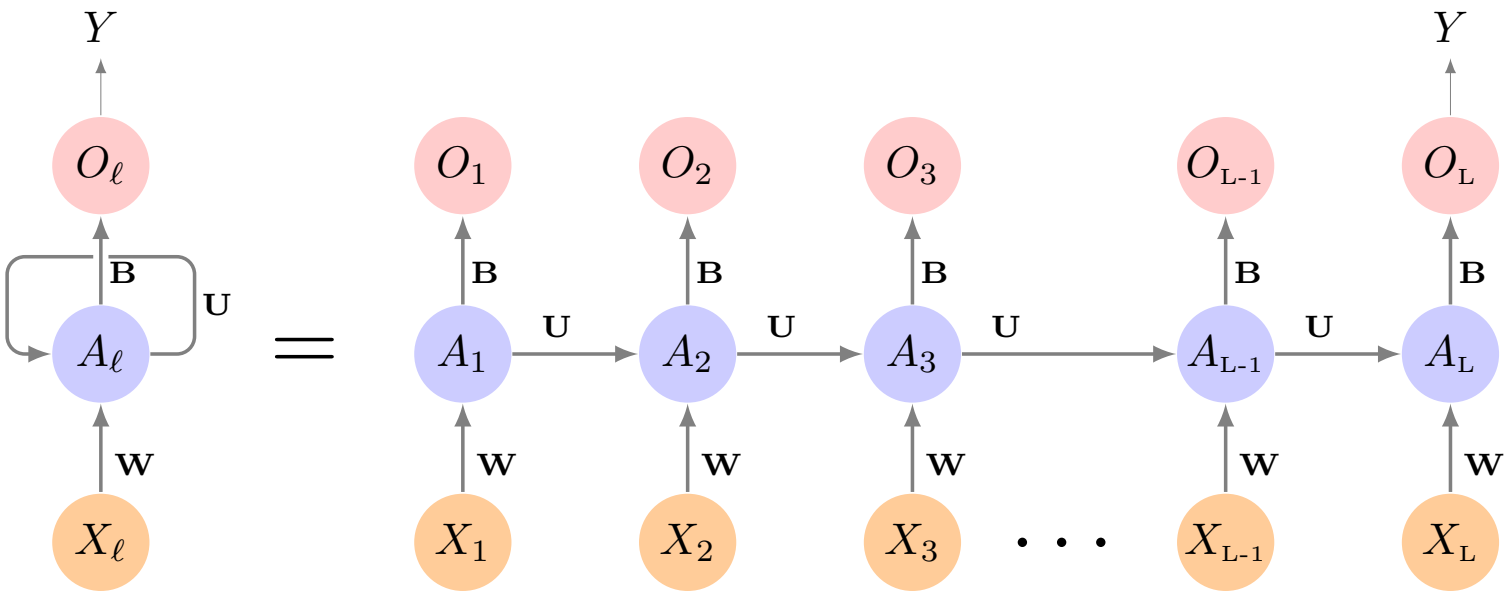




flamingo		Cooper's hawk		Cooper's hawk	
flamingo	0.83	kite (raptor)	0.60	fountain	0.35
spoonbill	0.17	great grey owl	0.09	nail	0.12
white stork	0.00	robin	0.06	hook	0.07
Lhasa Apso		**cat**		**Cape weaver**	
Tibetan terrier	0.56	Old English sheepdog	0.82	jacamar	0.28
Lhasa	0.32	Shih-Tzu	0.04	macaw	0.12
cocker spaniel	0.03	Persian cat	0.04	robin	0.12

Here we use the 50-layer `resnet50` network trained on the 1000-class `imagenet` corpus to classify some photographs.

Simple Recurrent Neural Network Architecture



- The hidden layer is a sequence of vectors A_ℓ, receiving as input X_ℓ as well as $A_{\ell-1}$. A_ℓ produces an output O_ℓ.
- The *same* weights $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{B}$ are used at each step in the sequence — hence the term *recurrent*.
- The A_ℓ sequence represents an evolving model for the response that is updated as each element X_ℓ is processed.

RNN in Detail

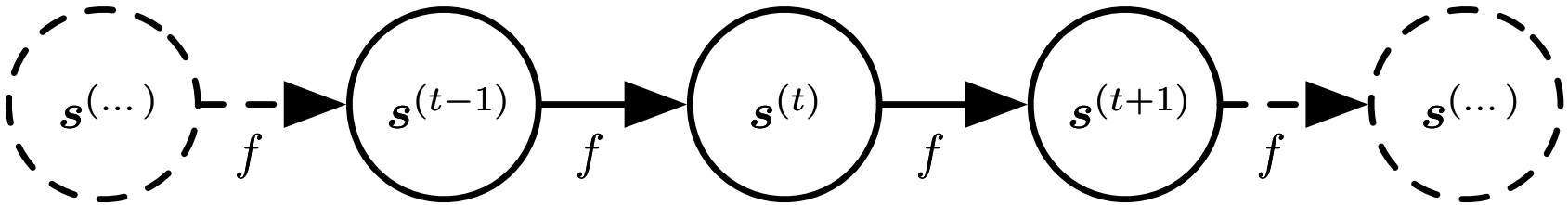
Suppose $X_\ell = (X_{\ell 1}, X_{\ell 2}, \ldots, X_{\ell p})$ has p components, and $A_\ell = (A_{\ell 1}, A_{\ell 2}, \ldots, A_{\ell K})$ has K components. Then the computation at the kth components of hidden unit A_ℓ is

$$\begin{aligned} A_{\ell k} &= g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\Big) \\ O_\ell &= \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k} \end{aligned}$$

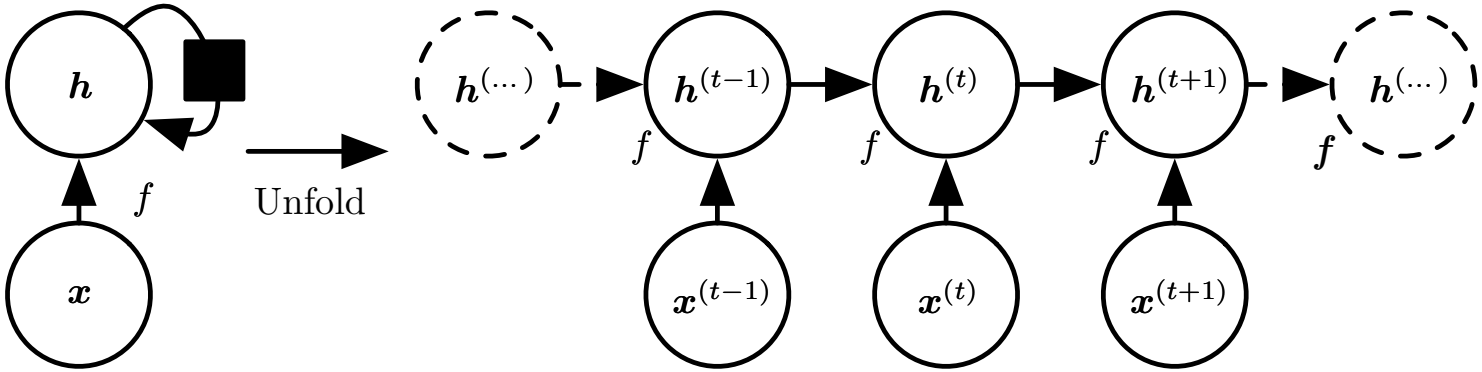
Often we are concerned only with the prediction O_L at the last unit. For squared error loss, and n sequence/response pairs, we would minimize

$$\sum_{i=1}^{n} (y_i - o_{iL})^2 = \sum_{i=1}^{n} \Big(y_i - \big(\beta_0 + \sum_{k=1}^{K} \beta_k g\big(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{iLj} + \sum_{s=1}^{K} u_{ks} a_{i,L-1,s}\big)\big)\Big)^2.$$

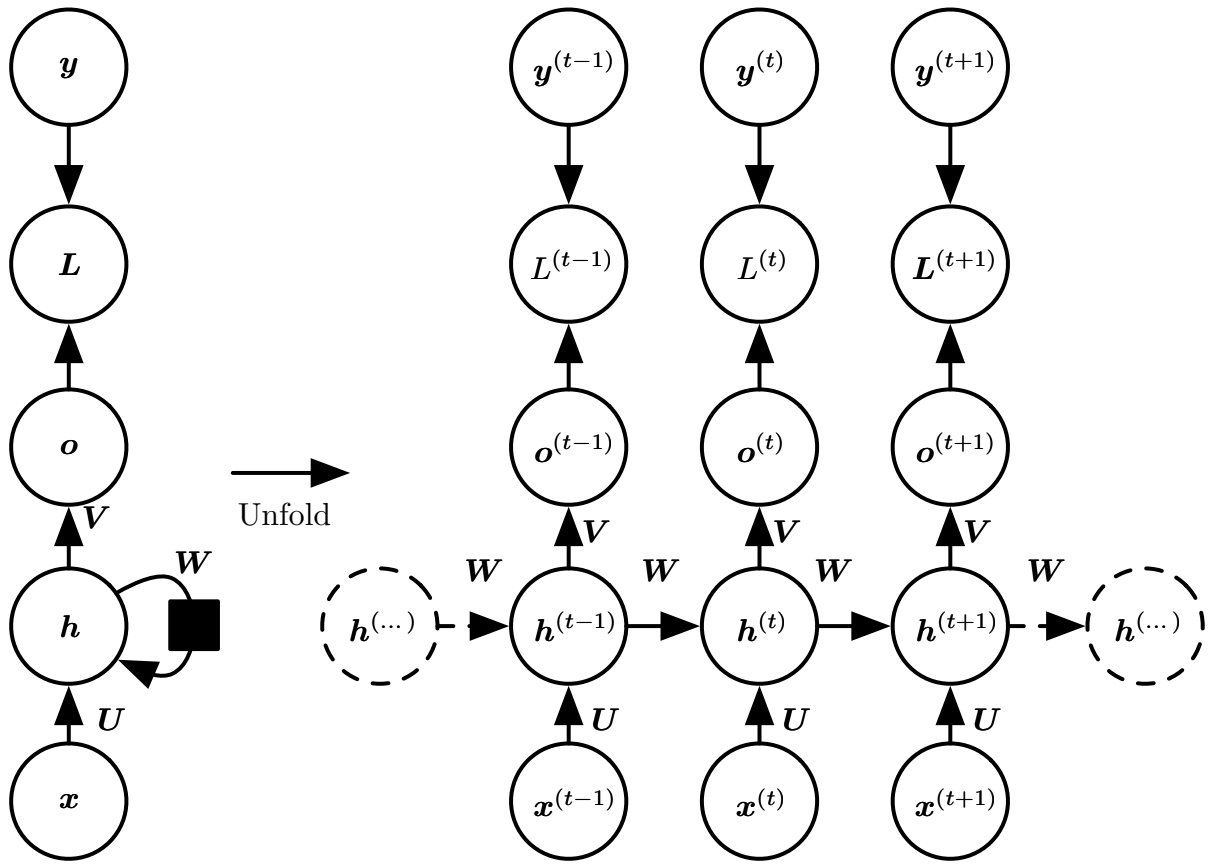
Classical Dynamical Systems



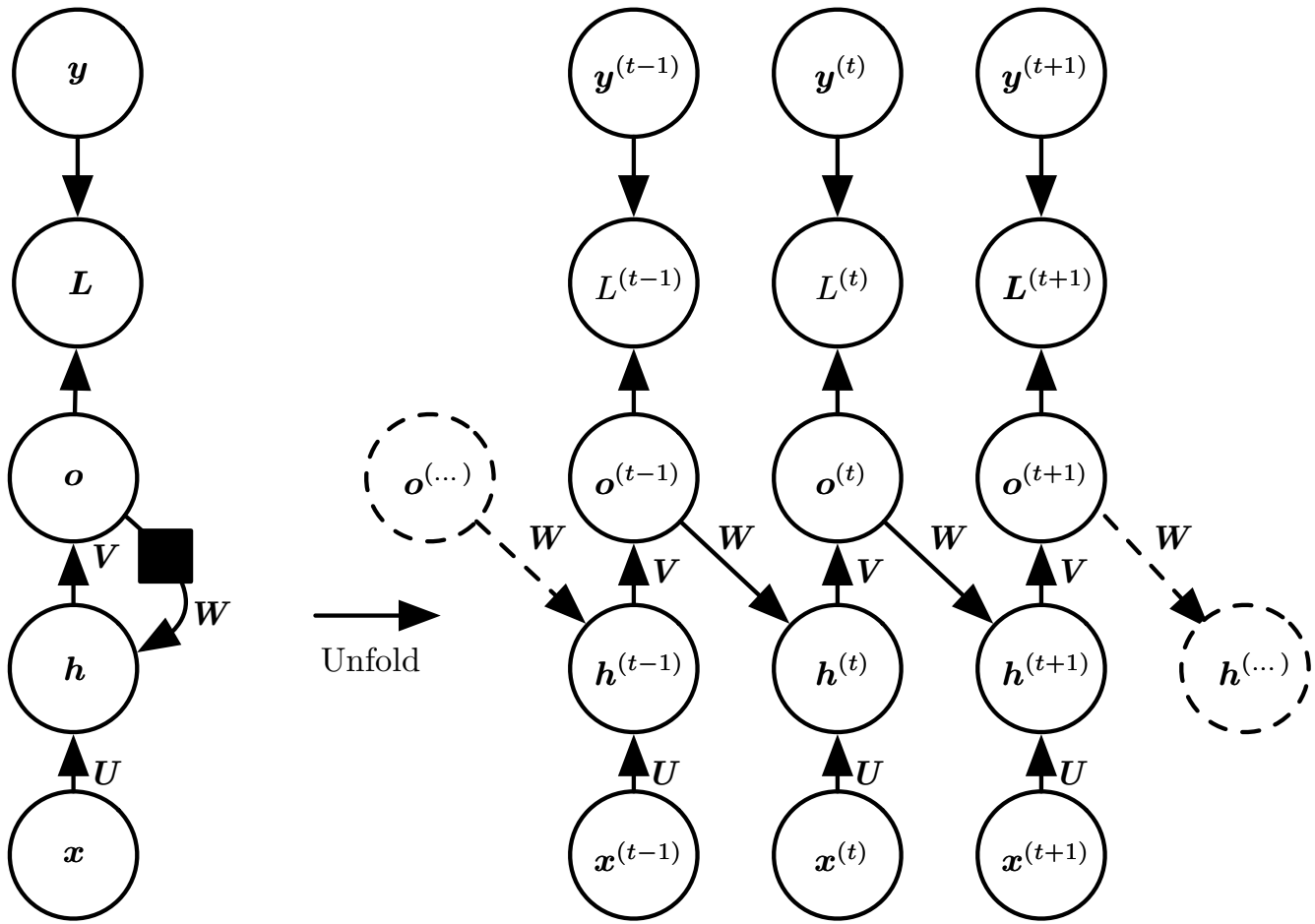
Unfolding Computation Graphs



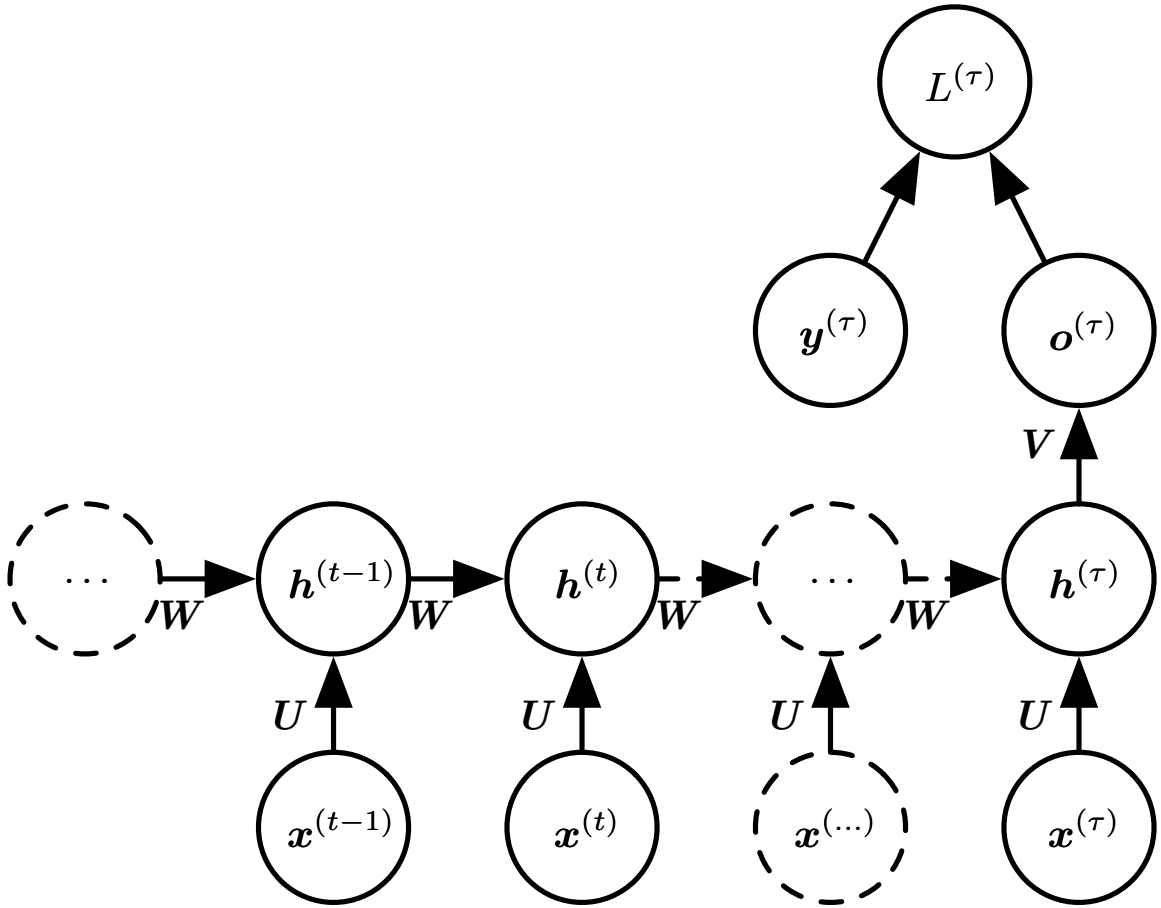
Recurrent Hidden Units



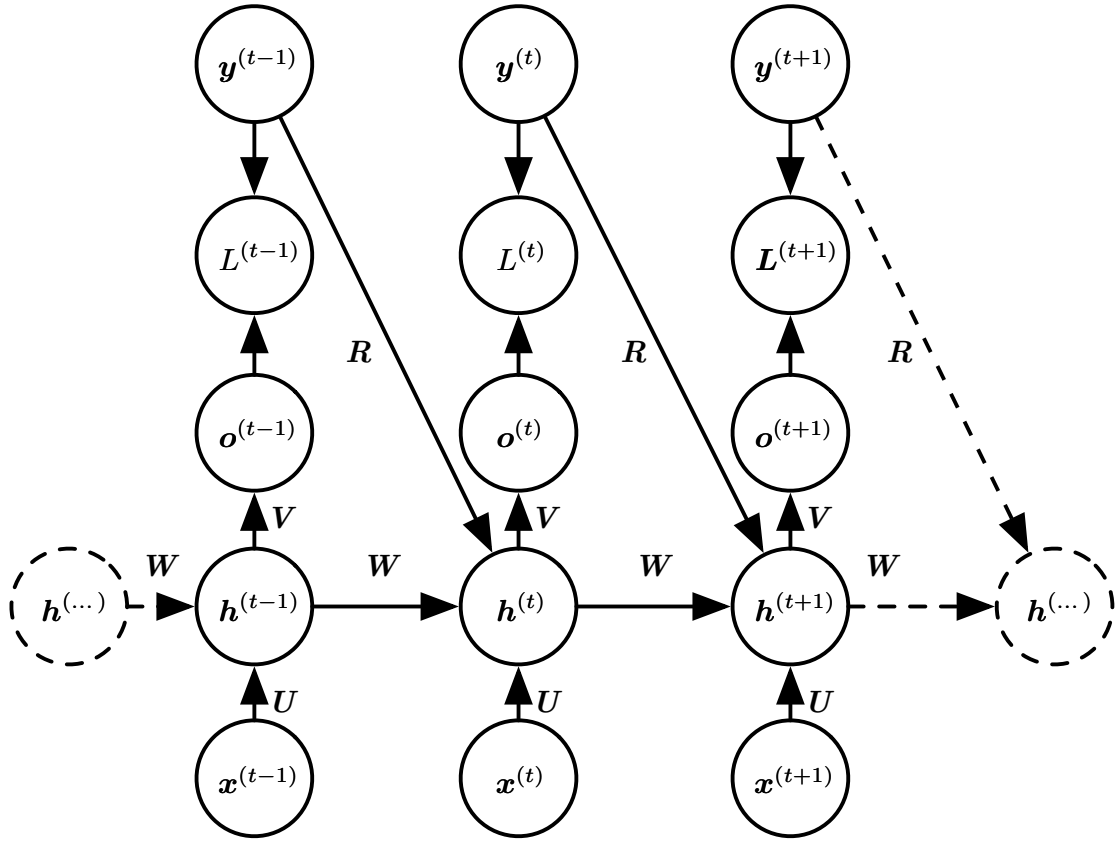
Recurrence through only the Output



Sequence Input, Single Output

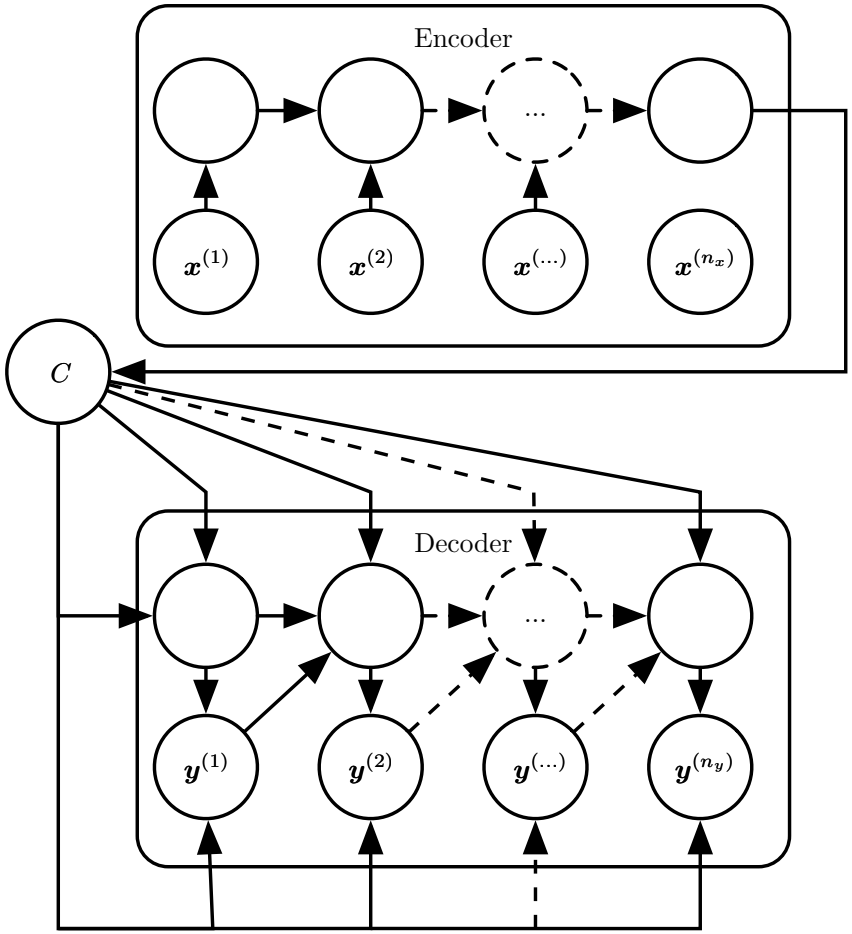


Hidden and Output Recurrence

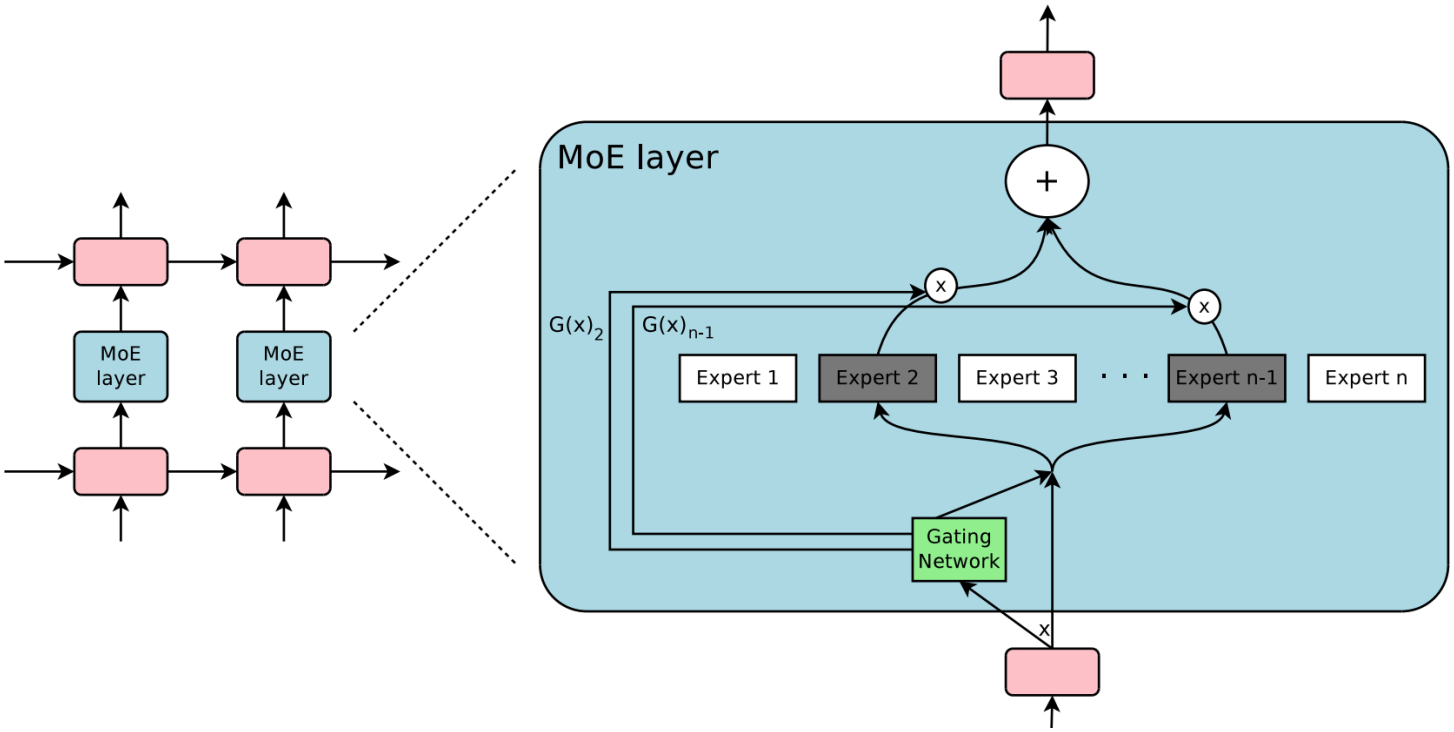


The output values are not forced to be conditionally independent in this model.

Sequence to Sequence Architecture

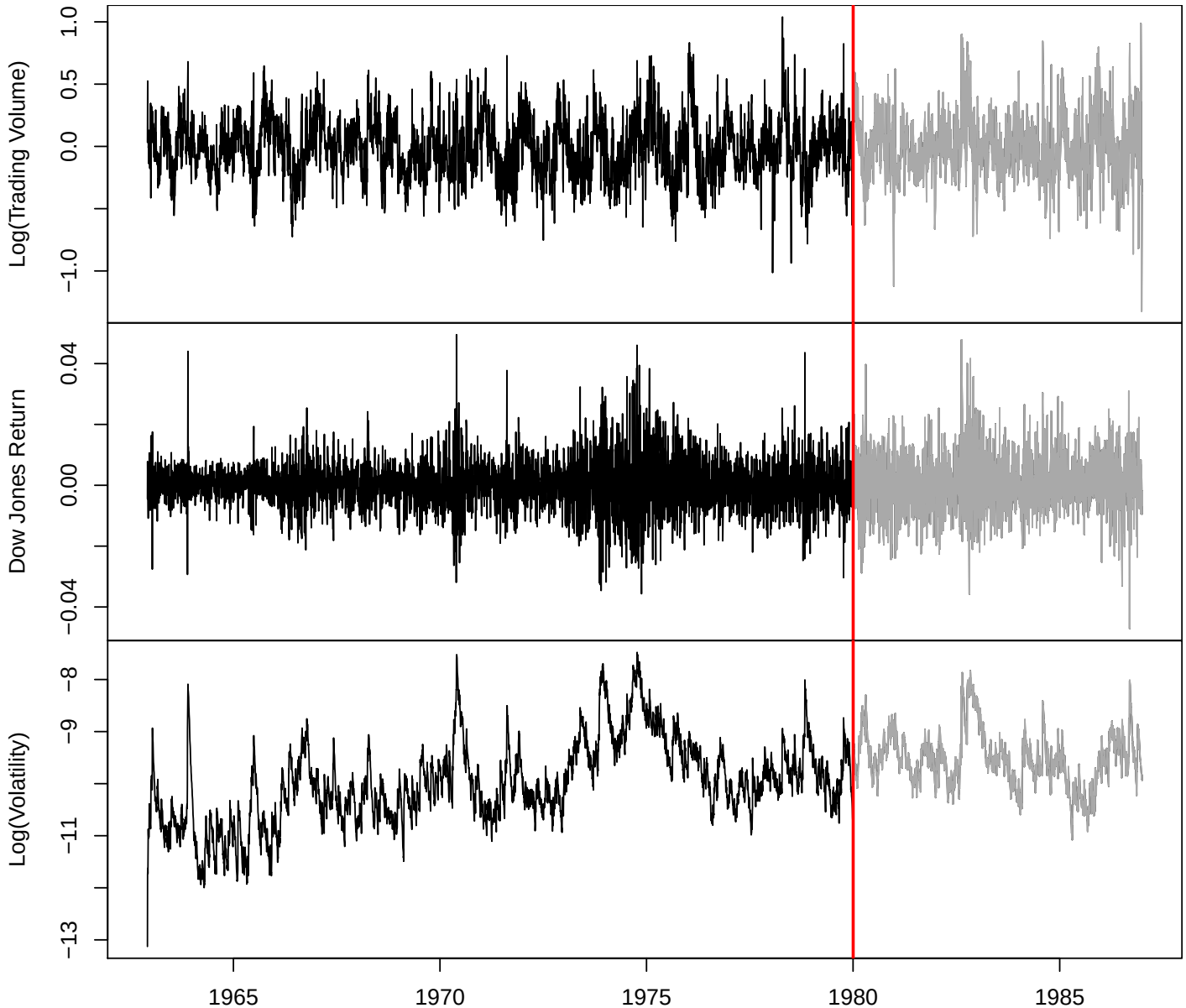


Dynamic Structure



Outrageously Large Neural Networks

Time Series Forecasting



RNN Forecaster

We only have one series of data! How do we set up for an RNN?

We extract many short mini-series of input sequences $X = \{X_1, X_2, \ldots, X_L\}$ with a predefined length L known as the *lag*:

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, \; X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, \cdots, X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}, \quad \text{and } Y = v_t.$$

Since $T = 6,051$, with $L = 5$ we can create $6,046$ such (X, Y) pairs.
We use the first $4,281$ as training data, and the following $1,770$ as test data. We fit an RNN with 12 hidden units per lag step (i.e. per A_ℓ.)

RNN Results for NYSE Data

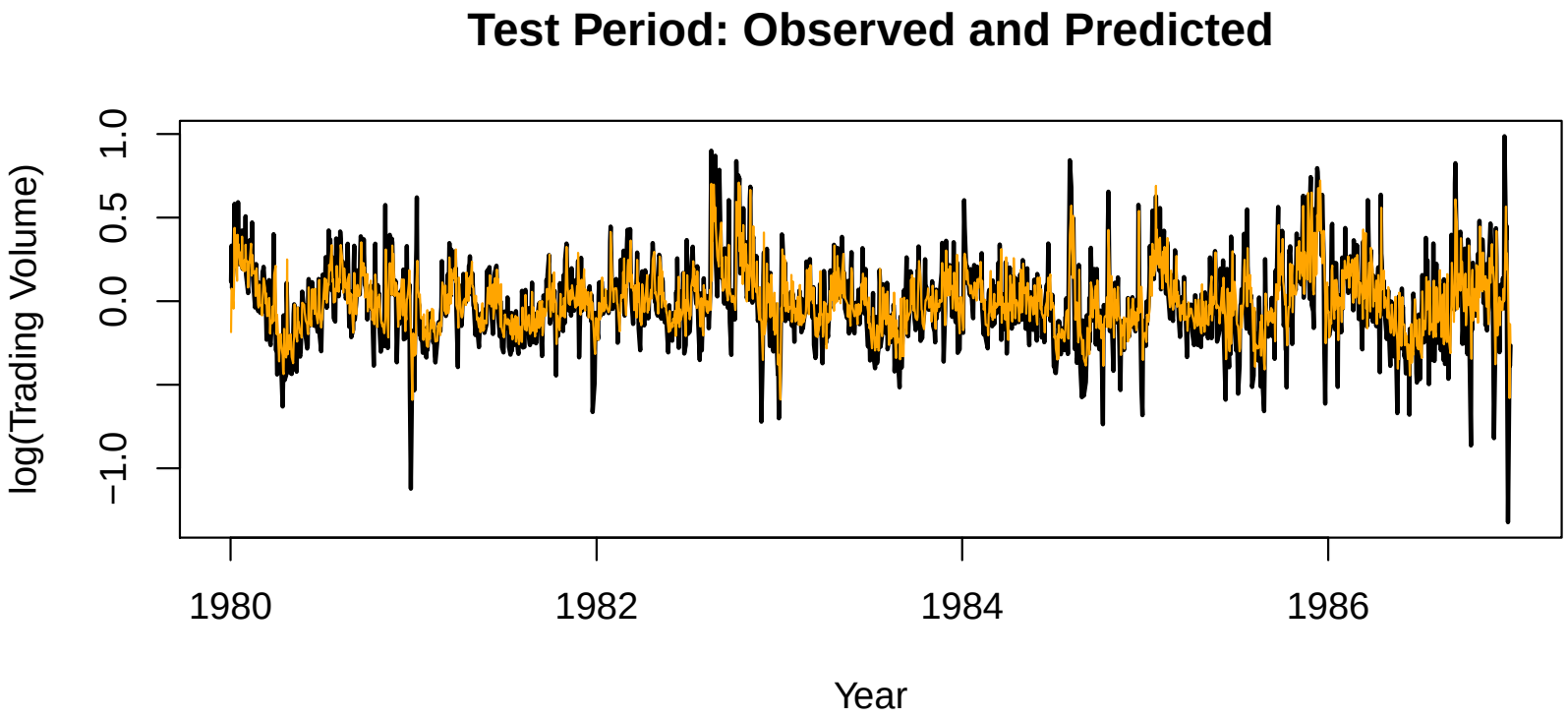


Figure shows predictions and truth for test period.

$R^2 = 0.42$ for RNN
$R^2 = 0.18$ for *straw man* — use yesterday's value of `Log trading volume` to predict that of today.

Autoregression Forecaster

The RNN forecaster is similar in structure to a traditional *autoregression* procedure.

$$\mathbf{y} = \begin{bmatrix} v_{L+1} \\ v_{L+2} \\ v_{L+3} \\ \vdots \\ v_T \end{bmatrix} \qquad \mathbf{M} = \begin{bmatrix} 1 & v_L & v_{L-1} & \cdots & v_1 \\ 1 & v_{L+1} & v_L & \cdots & v_2 \\ 1 & v_{L+2} & v_{L+1} & \cdots & v_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_{T-1} & v_{T-2} & \cdots & v_{T-L} \end{bmatrix}.$$

Fit an OLS regression of $\mathbf{y}$ on $\mathbf{M}$, giving

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \hat{\beta}_2 v_{t-2} + \cdots + \hat{\beta}_L v_{t-L}.$$

Known as an *order-L autoregression* model or AR(L).
For the `NYSE` data we can include lagged versions of `DJ_return` and `log_volatility` in matrix $\mathbf{M}$, resulting in $3L + 1$ columns.

Autoregression Results for NYSE Data

$R^2 = 0.41$ for AR(5) model (16 parameters)

$R^2 = 0.42$ for RNN model (205 parameters)

$R^2 = 0.42$ for AR(5) model fit by neural network.

$R^2 = 0.46$ for all models if we include `day_of_week` of day being predicted.

Summary of RNNs

- We have presented the simplest of RNNs. Many more complex variations exist.
- One variation treats the sequence as a one-dimensional image, and uses CNNs for fitting. For example, a sequence of words using an embedding representation can be viewed as an image, and the CNN convolves by sliding a convolutional filter along the sequence.
- Can have additional hidden layers, where each hidden layer is a sequence, and treats the previous hidden layer as an input sequence.
- Can have output also be a sequence, and input and output share the hidden units. So called `seq2seq` learning are used for language translation.

When to Use Deep Learning

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis. Examples include digital mammography, ophthalmology, MRI scans, and digital X-rays.
- RNNs have had big wins in speech modeling, language translation, and forecasting.

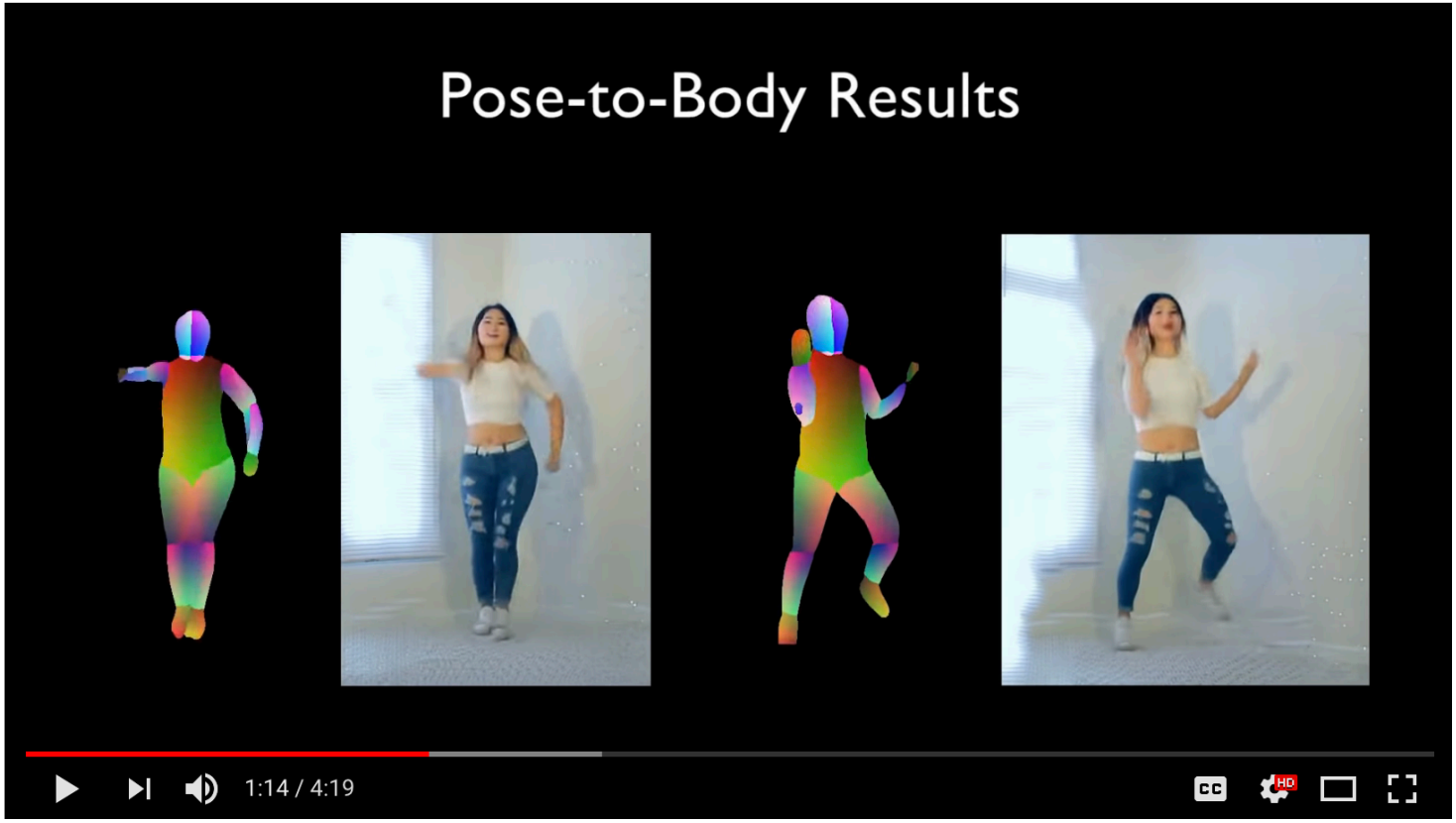
Should we always use deep learning models?

- Often the big successes occur when the *signal to noise ratio* is high — e.g. image recognition and language translation. Datasets are large, and overfitting is not a big problem.
- For noisier data, simpler models can often work better.
  - On the `NYSE` data, the AR(5) model is much simpler than a RNN, and performed as well.
  - On the `IMDB` review data, the linear model fit by `glmnet` did as well as the neural network, and better than the RNN.
- We endorse the *Occam's razor* principal — we prefer simpler models if they work as well. More interpretable!

Deep Learning Applications

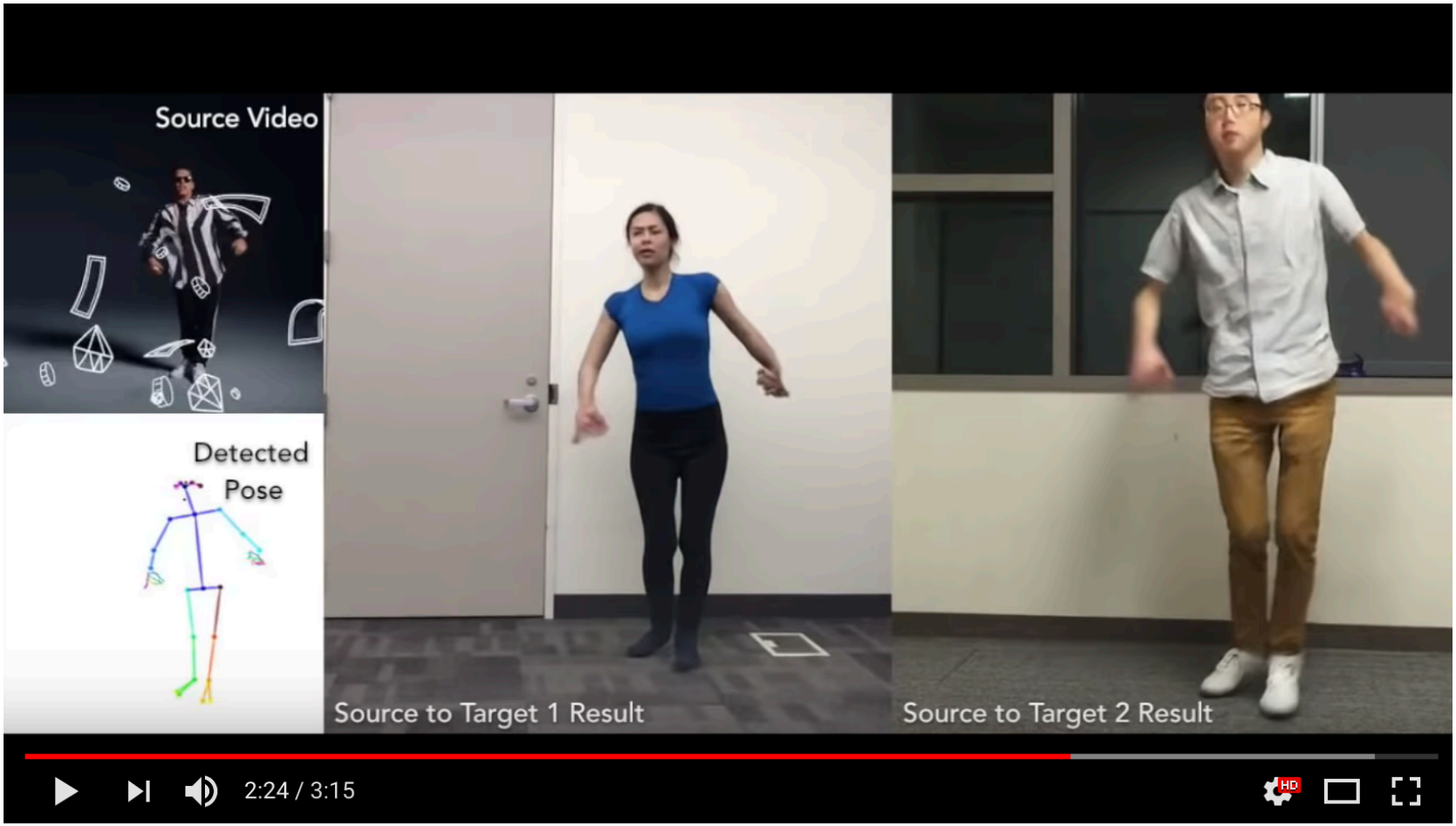
- Vision
- Vision - Dance
- Autonomous vehicles
- Google brain

Video Generation



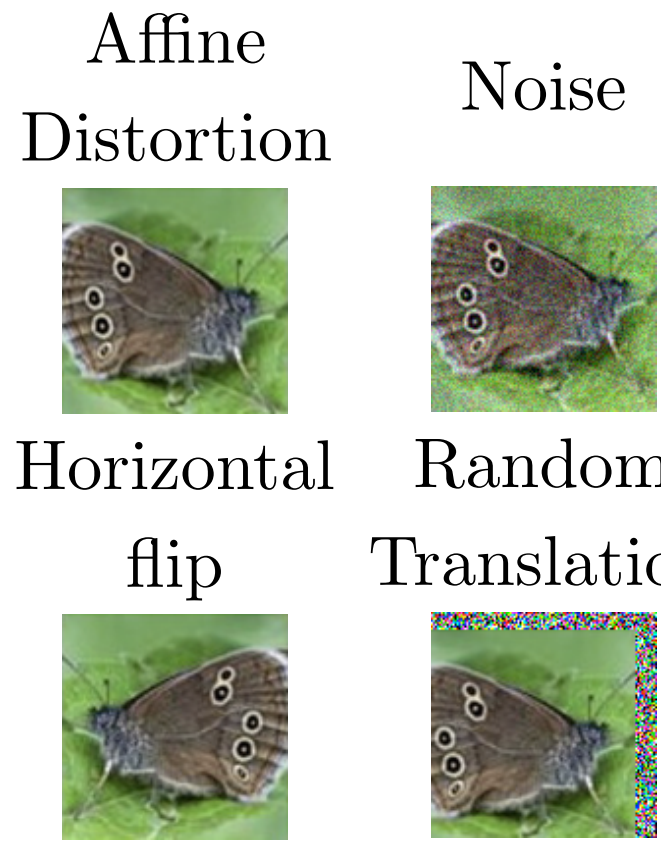
(Wang et al, 2018)

Everybody Dance Now!



(Chan et al 2018)

Dataset Augmentation for Computer Vision



Natural Language Processing

- An important predecessor to deep NLP is the family of models based on n-grams:

$$P(x_1, \ldots, x_\tau) = P(x_1, \ldots, x_{n-1}) \prod_{t=n}^{\tau} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}). \tag{12.5}$$

$$P(\texttt{THE DOG RAN AWAY}) = P_3(\texttt{THE DOG RAN}) P_3(\texttt{DOG RAN AWAY}) / P_2(\texttt{DOG RAN}). \tag{12.7}$$

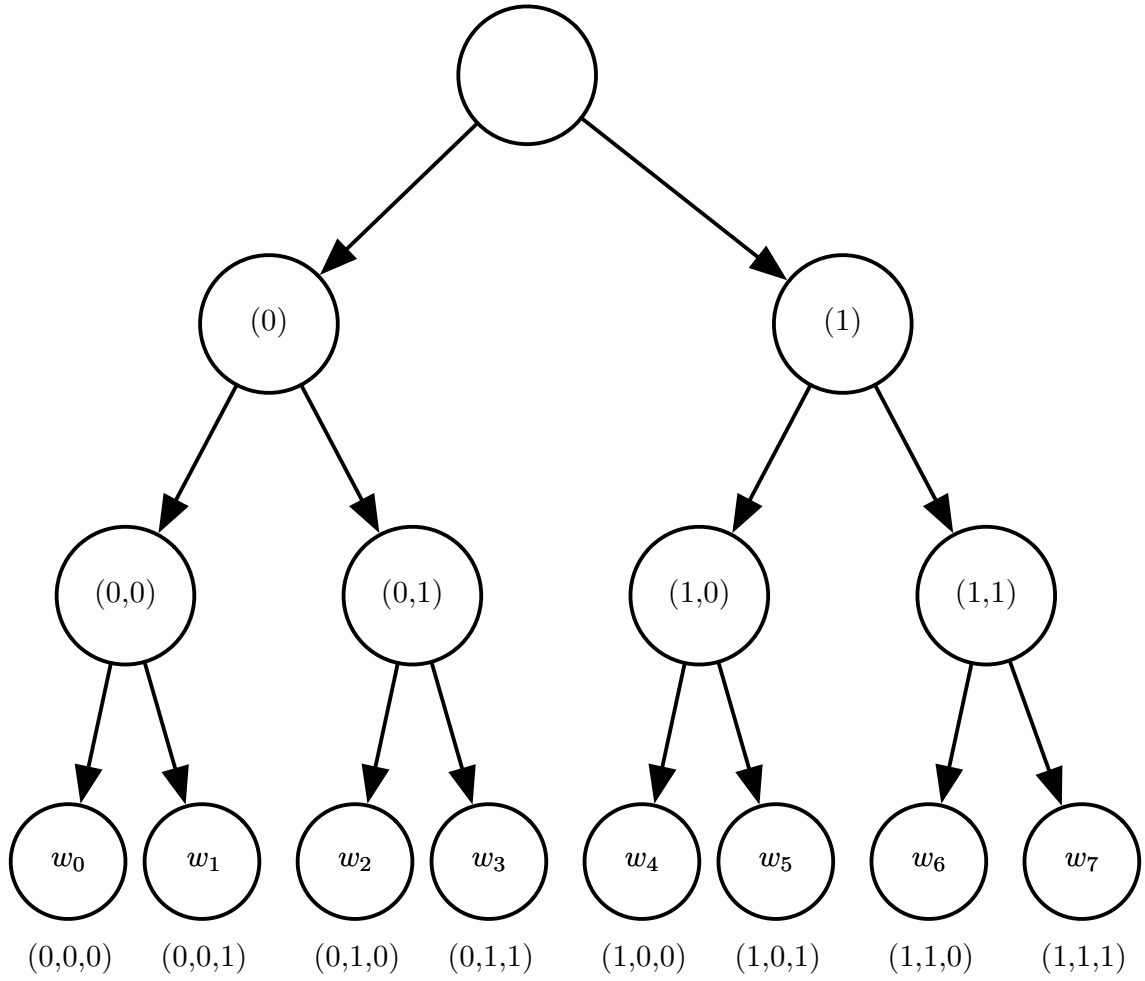
Improve with:

-Smoothing

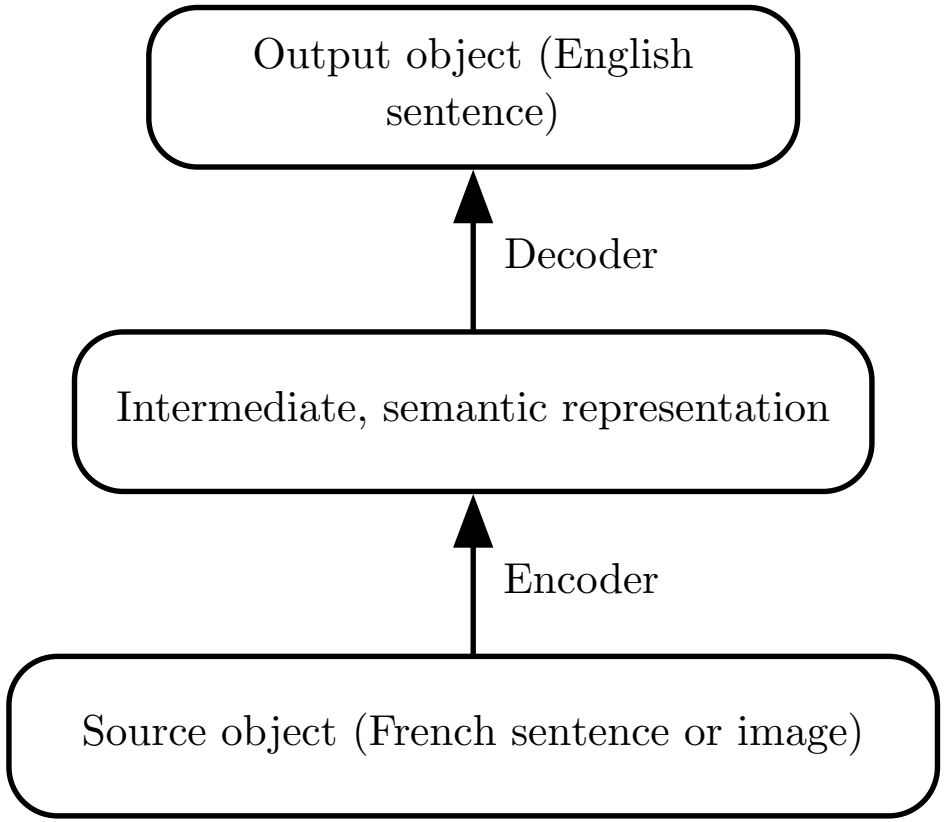
-Backoff

-Word categories

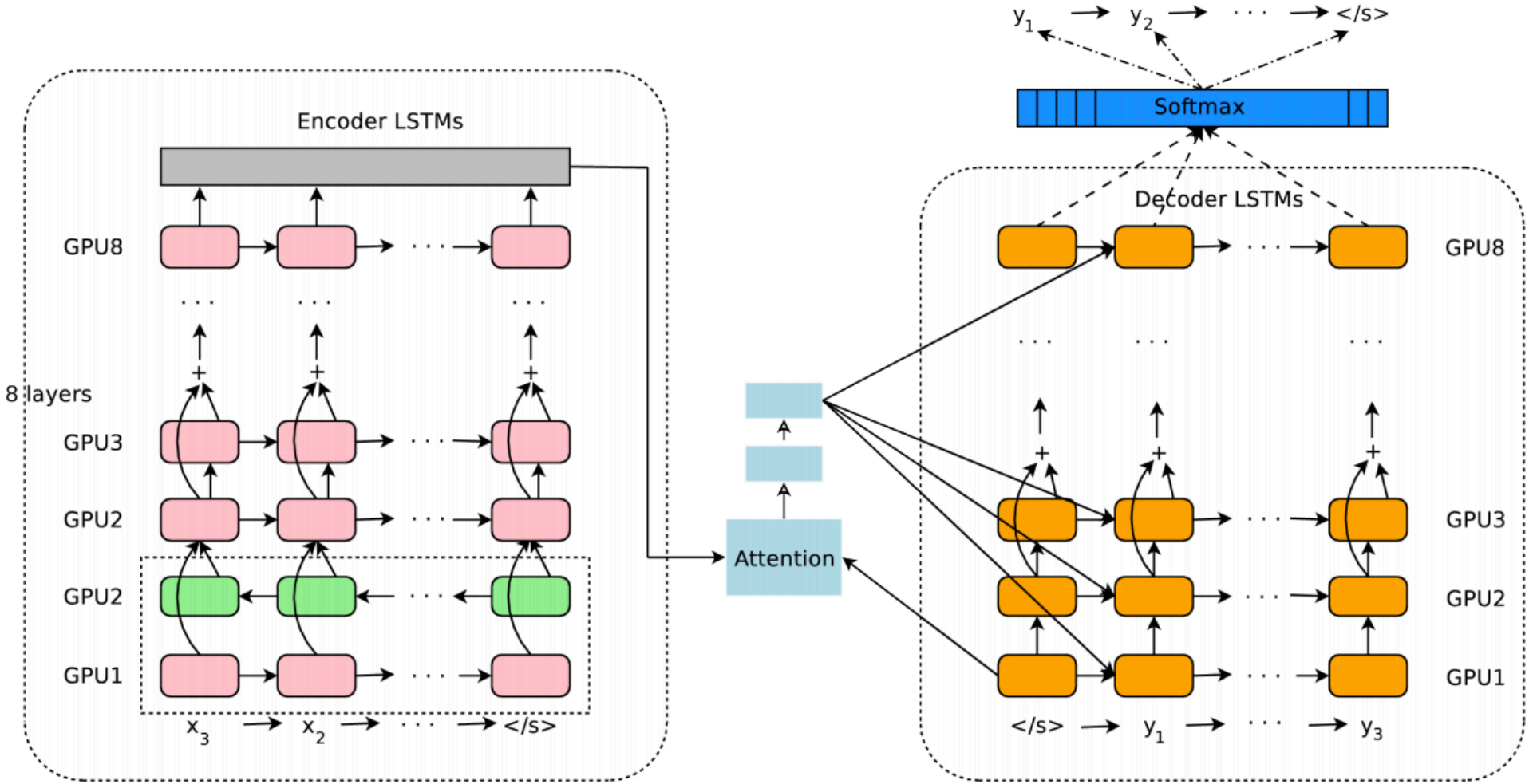
A Hierarchy of Words and Word Categories



Neural Machine Translation



Google Neural Machine Translation



Wu et al 2016

Speech Recognition

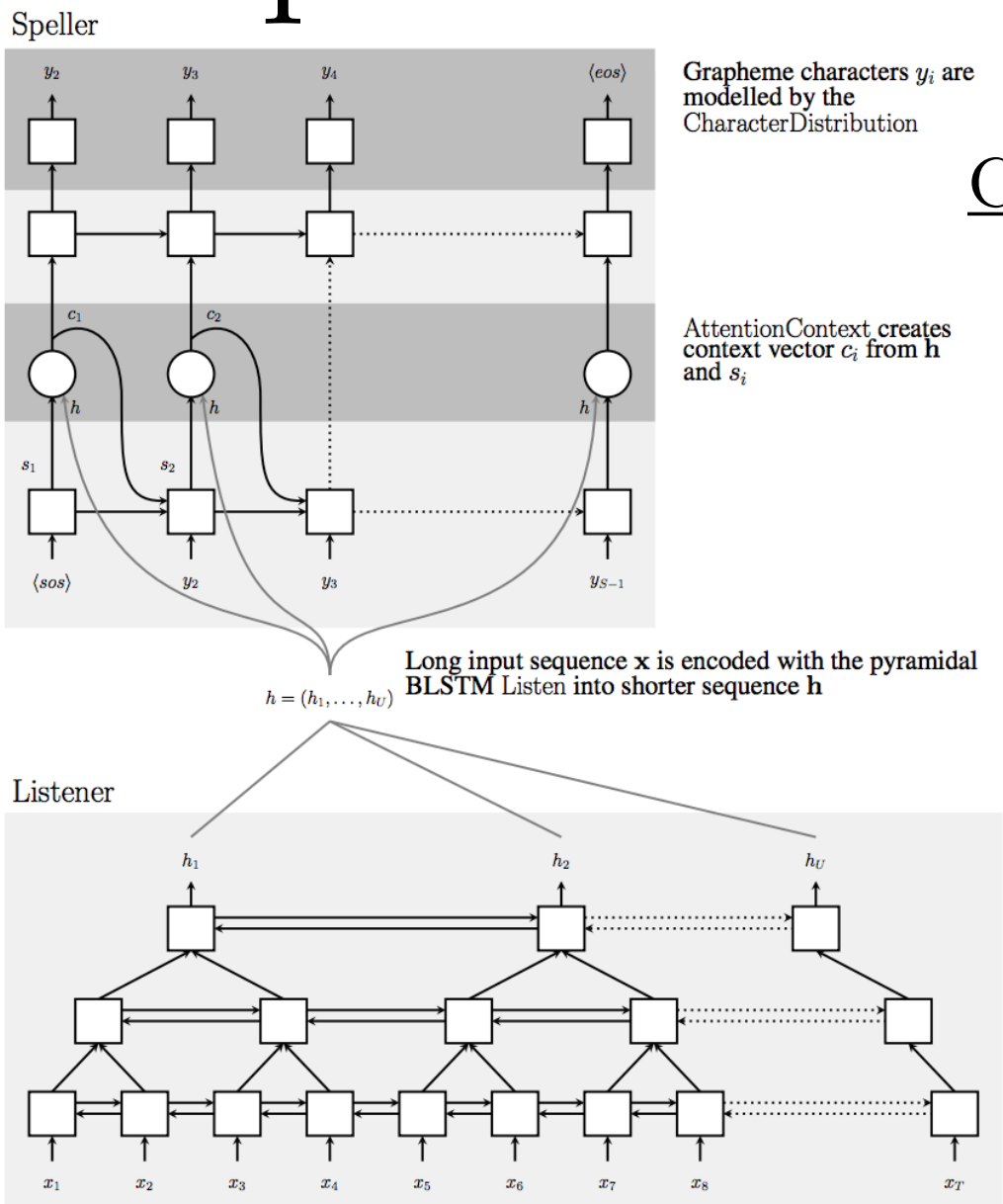
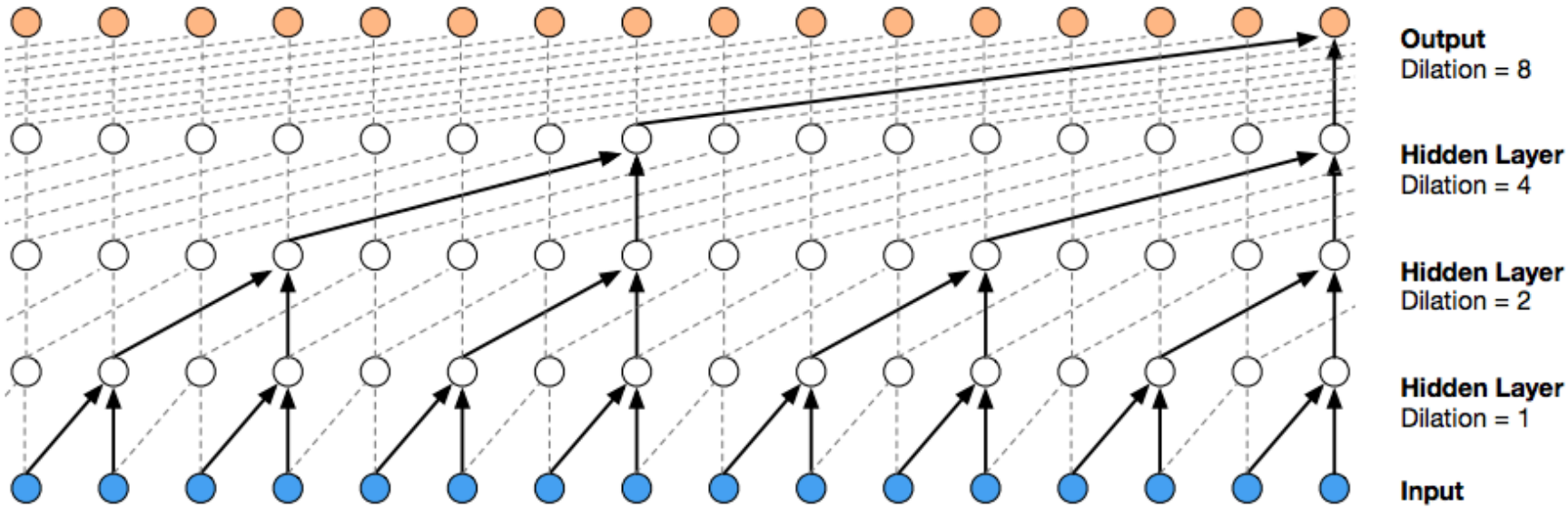


Figure 1: Listen, Attend and Spell (LAS) model: the listener is a pyramidal BLSTM encoding our input sequence $\mathbf{x}$ into high level features $\mathbf{h}$, the speller is an attention-based decoder generating the $\mathbf{y}$ characters from $\mathbf{h}$.

Current speech recognition is based on seq2seq with attention

Graphic from "Listen, Attend, and Spell" Chan et al 2015

Speech Synthesis



WaveNet

(van den Oord et al, 2016)

Deep RL for Atari game playing

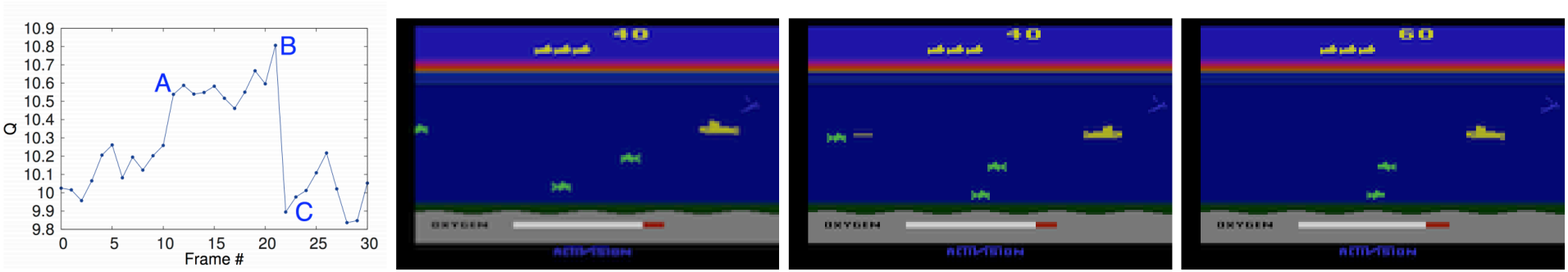


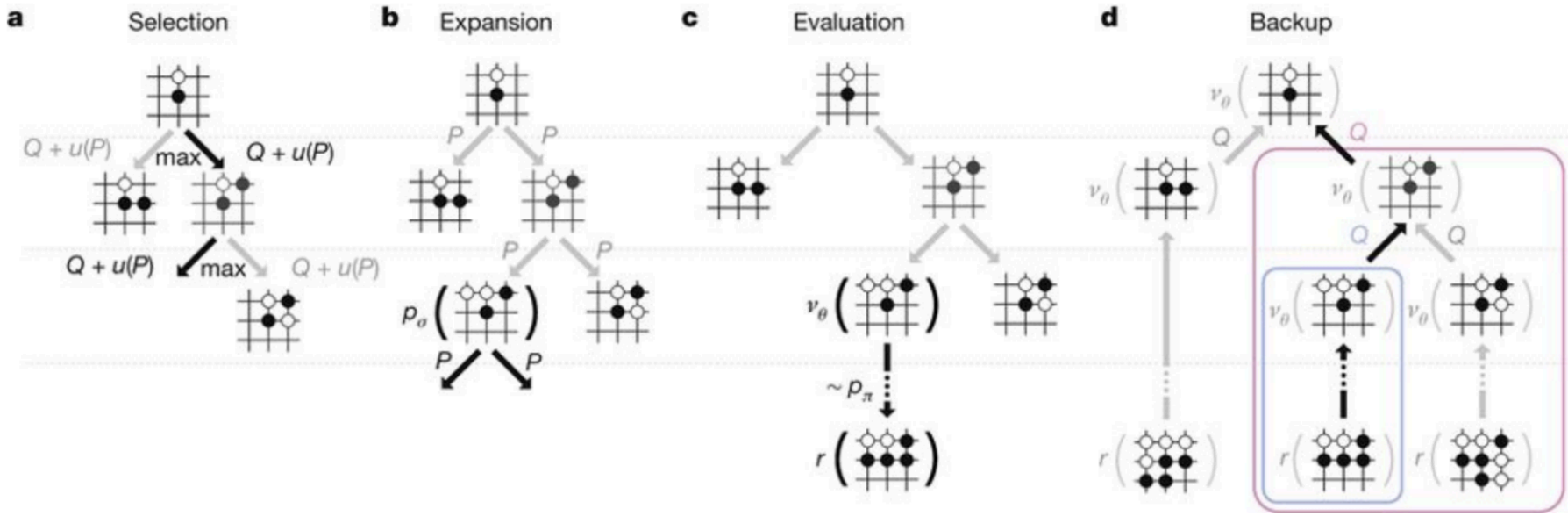
Figure 3: The leftmost plot shows the predicted value function for a 30 frame segment of the game Seaquest. The three screenshots correspond to the frames labeled by A, B, and C respectively.

(Mnih et al 2013)

Convolutional network estimates the value function (future rewards) used to guide the game-playing agent.

Superhuman Go Performance

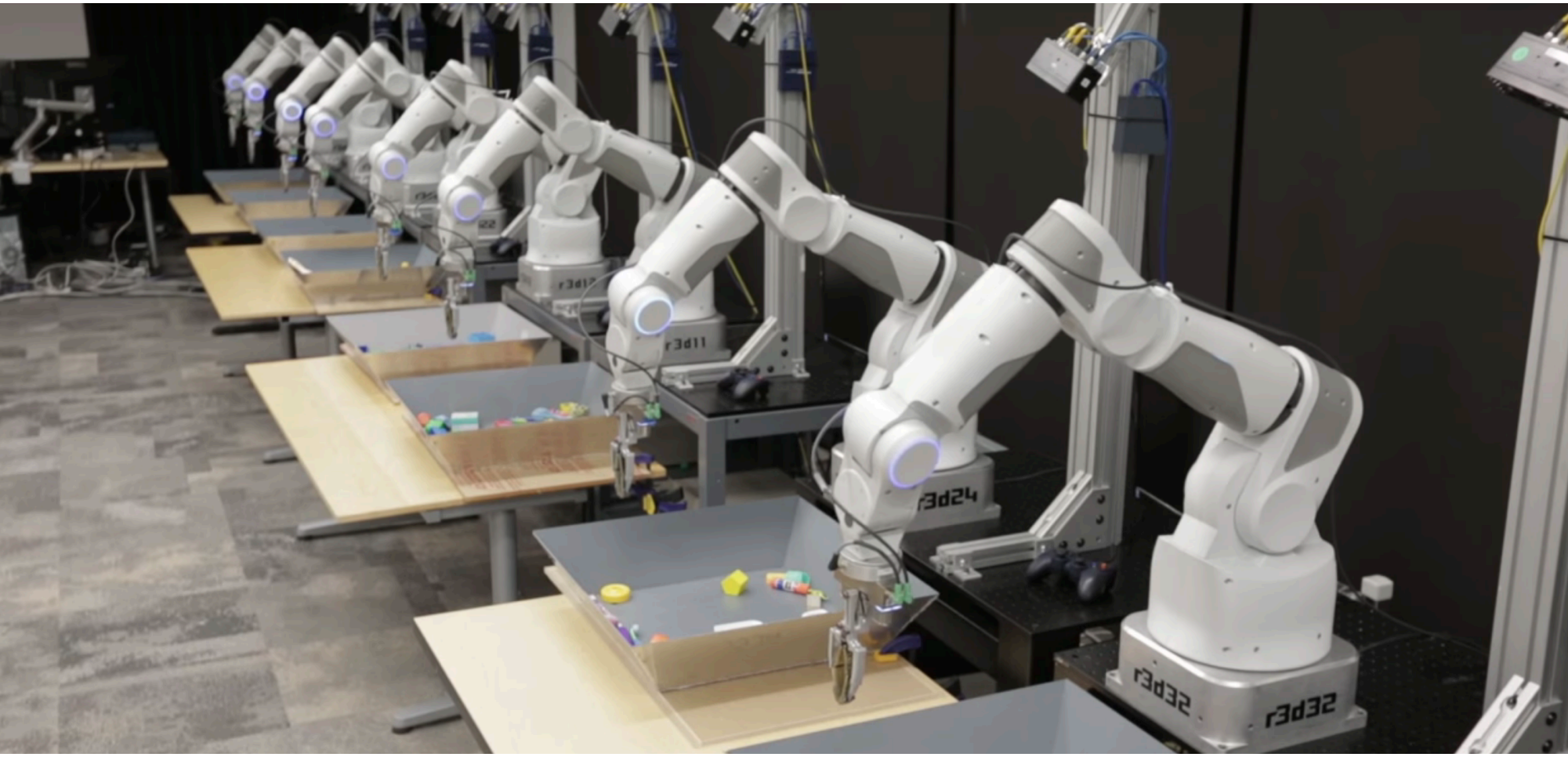
Monte Carlo tree search, with convolutional networks for value function and policy



a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

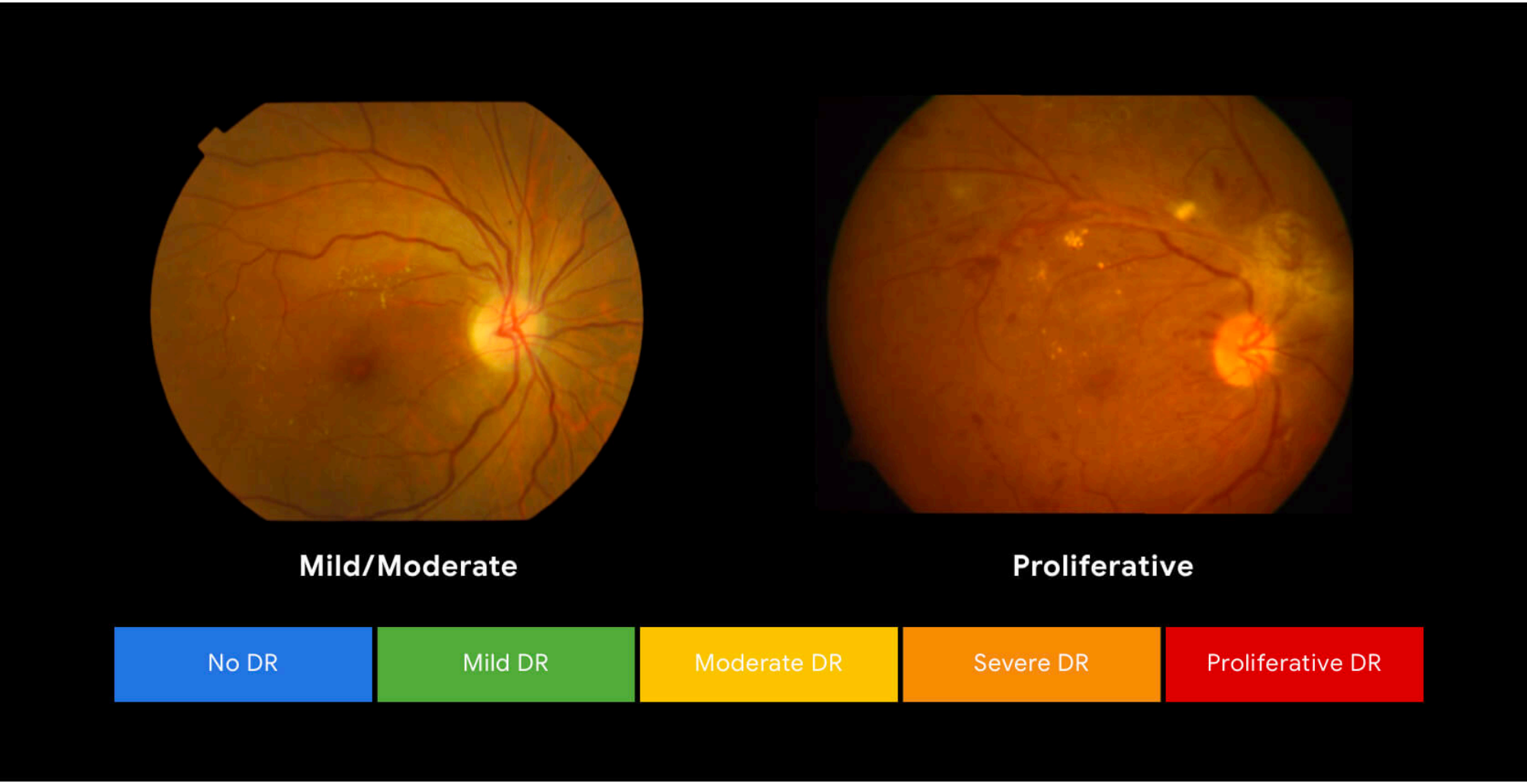
(Silver et al, 2016)

Robotics



(Google Brain)

Healthcare and Biosciences



(Google Brain)

Autonomous Vehicles

(WayMo)