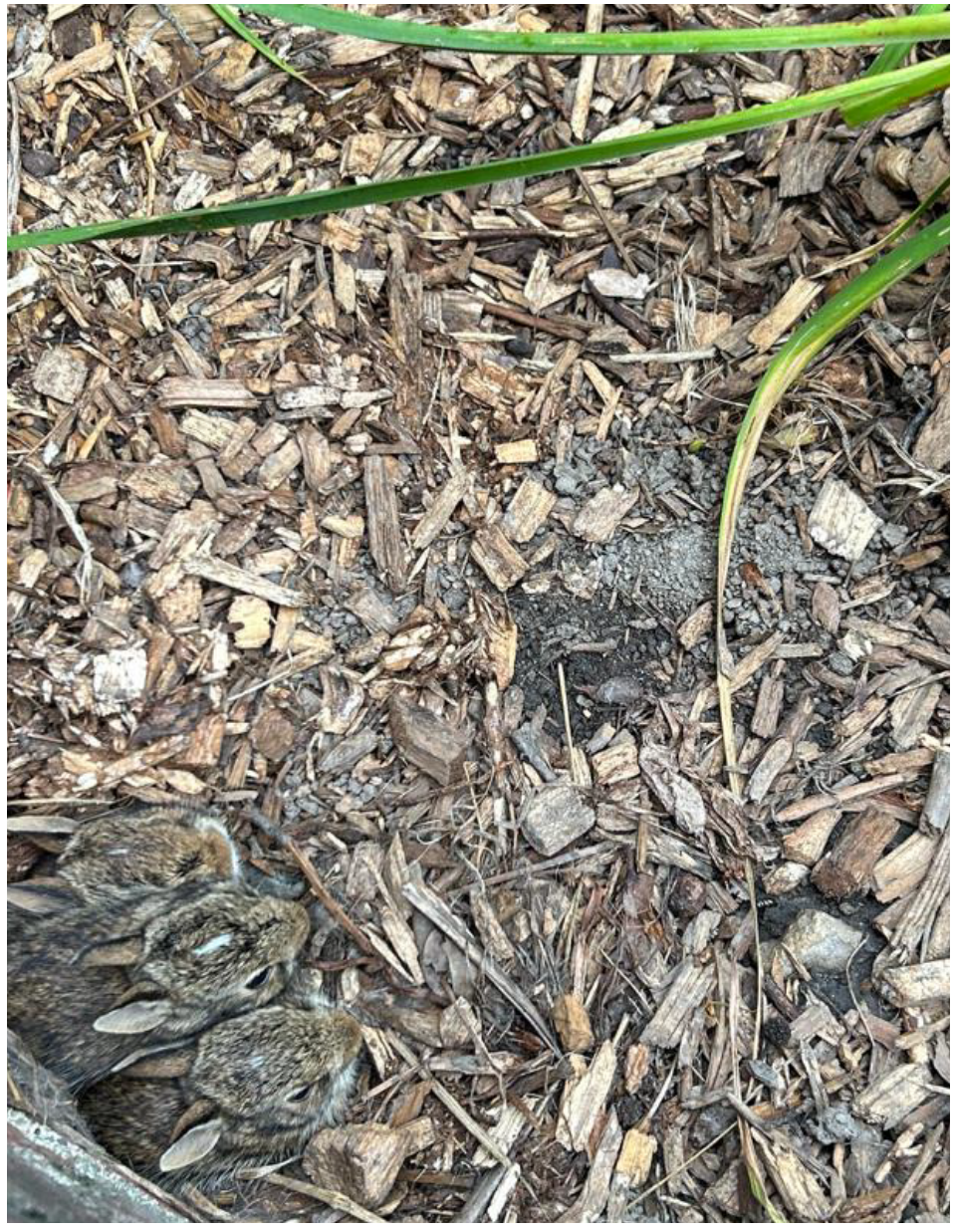
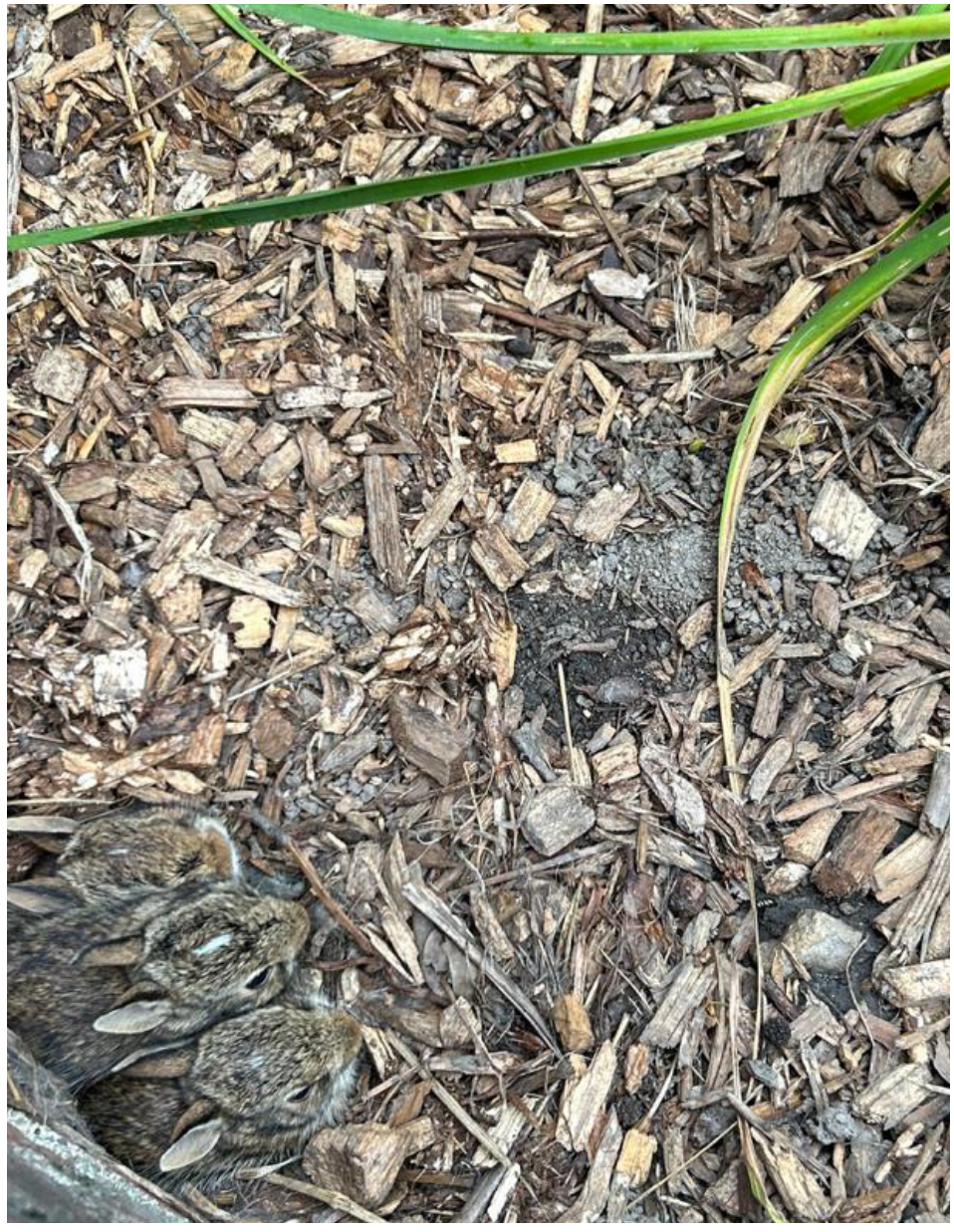




supervised/unsupervised learning

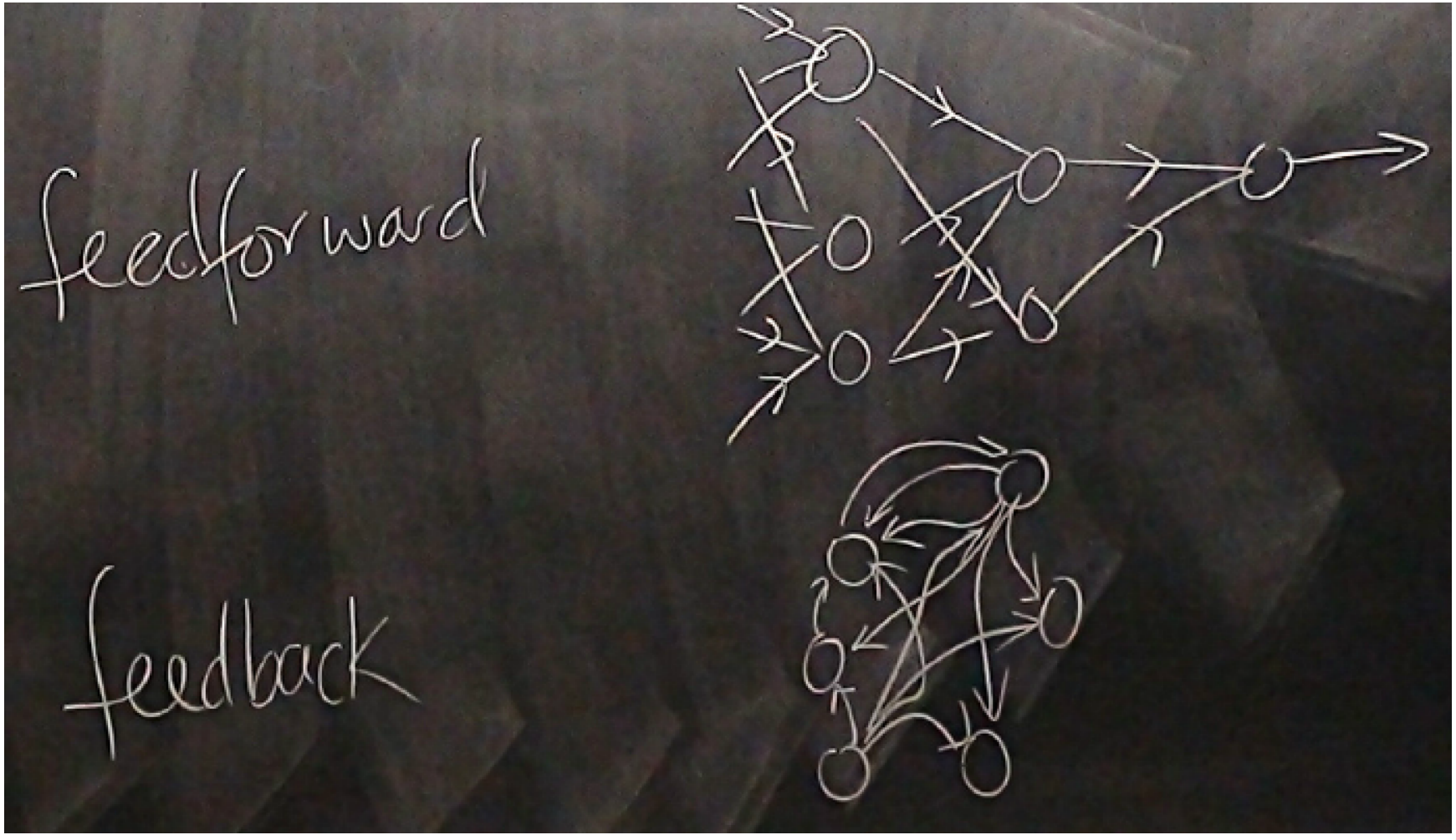
block b1

Feed Forward Networks

Protein 2D Structure

Feed Forward Networks (FFN)
Multilayer Perceptron (MLP)
Fully Connected Networks (FCN)
Fully Connected Layer (FC)

Feed Forward vs FeedBack Networks



Hopfield Networks are Feedback
memory adquisition/storage/retrieval

[Submitted on 16 Jul 2020 (v1), last revised 28 Apr 2021 (this version, v3)]

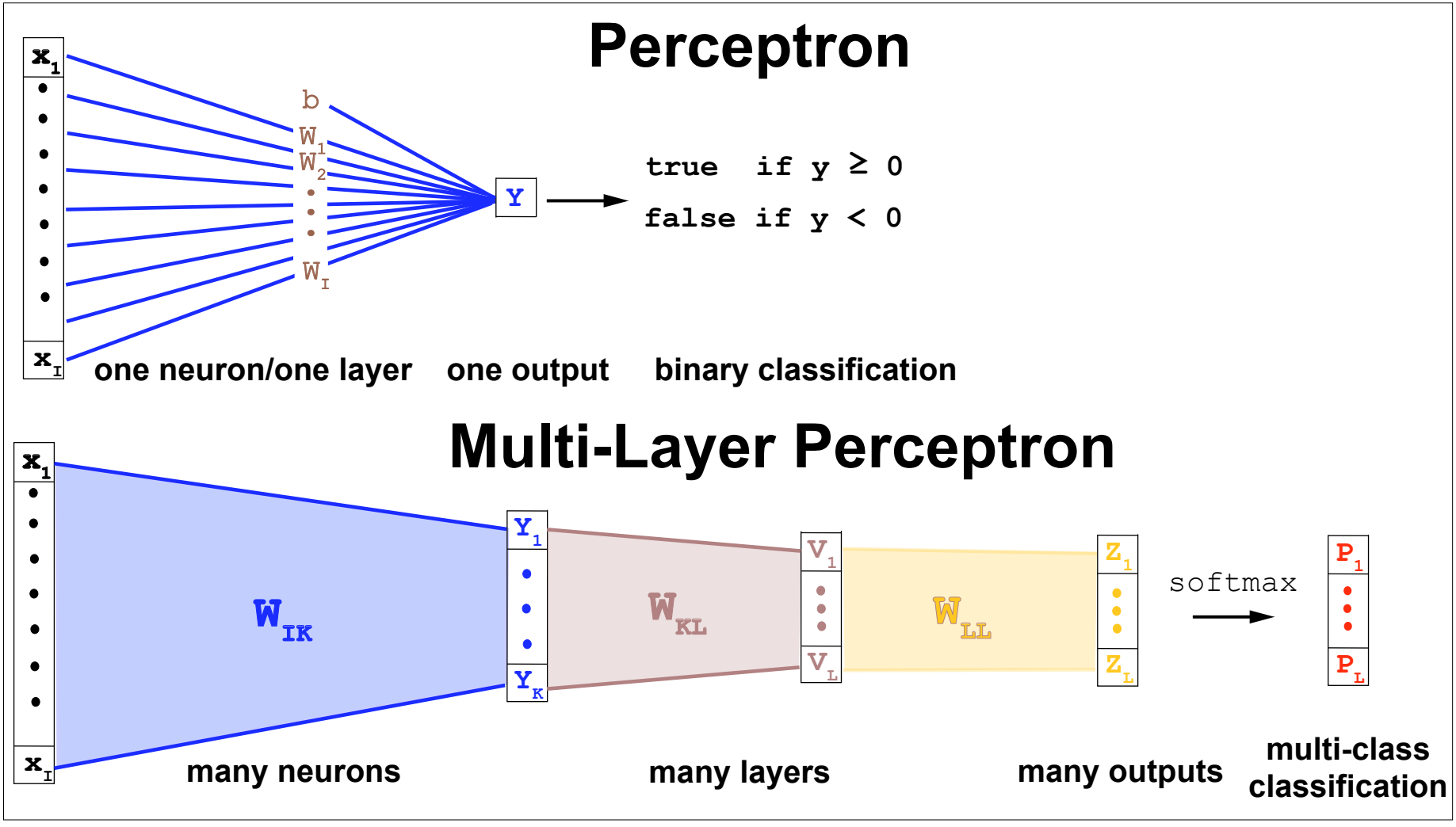
Hopfield Networks is All You Need

Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, Victor Greiff, David Kreil, Michael Kopp, Günter Klambauer, Johannes Brandstetter, Sepp Hochreiter

We introduce a modern Hopfield network with continuous states and a corresponding update rule. The new Hopfield network can store exponentially (with the dimension of the associative space) many patterns, retrieves the pattern with one update, and has exponentially small retrieval errors. It has three types of energy minima (fixed points of the update): (1) global fixed point averaging over all patterns, (2) metastable states averaging over a subset of patterns, and (3) fixed points which store a single pattern. The new update rule is equivalent to the attention mechanism used in transformers. This equivalence enables a characterization of the heads of transformer models. These heads perform in the first layers preferably global averaging and in higher layers partial averaging via metastable states. The new modern Hopfield network can be integrated into deep learning architectures as layers to allow the storage of and access to raw input data, intermediate results, or learned prototypes. These Hopfield layers enable new ways of deep learning, beyond fully-connected, convolutional, or recurrent networks, and provide pooling, memory, association, and attention mechanisms. We demonstrate the broad applicability of the Hopfield layers across various domains. Hopfield layers improved state-of-the-art on three out of four considered multiple instance learning problems as well as on immune repertoire classification with several hundreds of thousands of instances. On the UCI benchmark collections of small classification tasks, where deep learning methods typically struggle, Hopfield layers yielded a new state-of-the-art when compared to different machine learning methods. Finally, Hopfield layers achieved state-of-the-art on two drug design datasets. The implementation is available at: this https URL

Comments: 10 pages (+ appendix); 12 figures; Blog: this https URL GitHub: this https URL

From single-neuron perceptron to MLP



categorical variable = takes a fix number of values

The state of a single neuron (active/inactive) (apple/orange) is a **binary categorical variable**

Can be re-written using a **one-hot** representation

output/activity	one-hot
$t = \{0, 1\}$	$\bar{t} = [t_1, t_2]$
$t = 1$ (Apple)	$\bar{t}(A) = [1, 0]$
$t = 0$ (Orange)	$\bar{t}(O) = [0, 1]$

Easy generalization to more than two categories

For a categorical variable to distinguish a neuron in one of three states:
Active / Inactive / Refractory

we can use the one-hot encoding

$$t = [t_1, t_2, t_3]$$

$$t(\text{Active}) = [1, 0, 0]$$

$$t(\text{Inactive}) = [0, 1, 0]$$

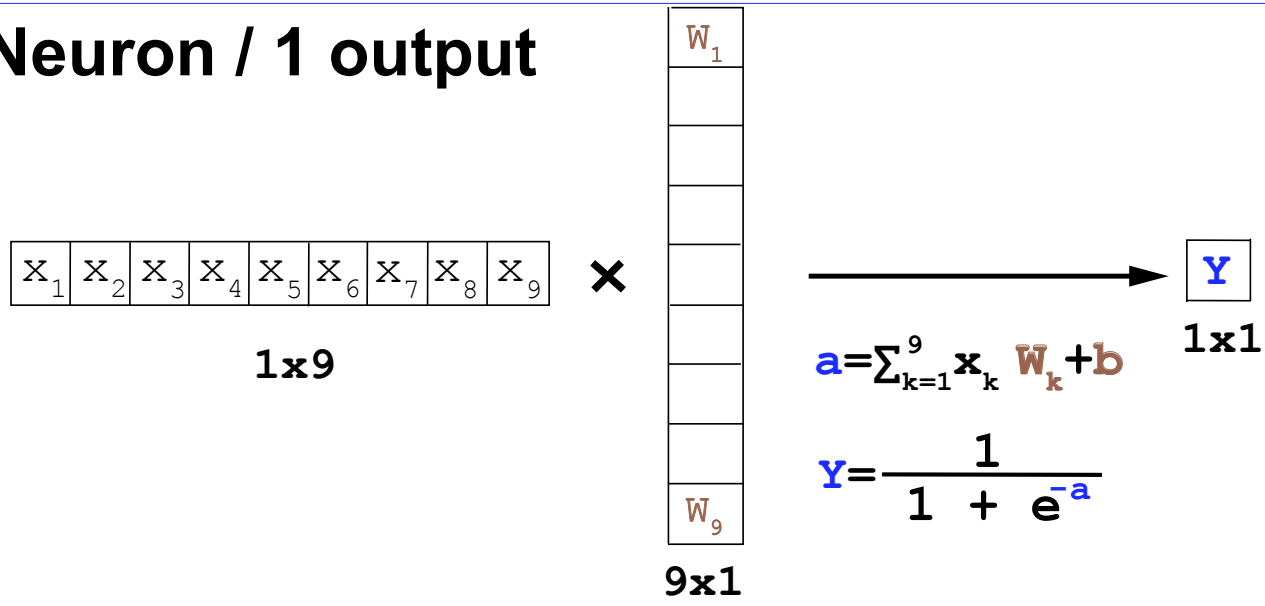
$$t(\text{Refractory}) = [0, 0, 1]$$

One-hot encoding for categorical variables

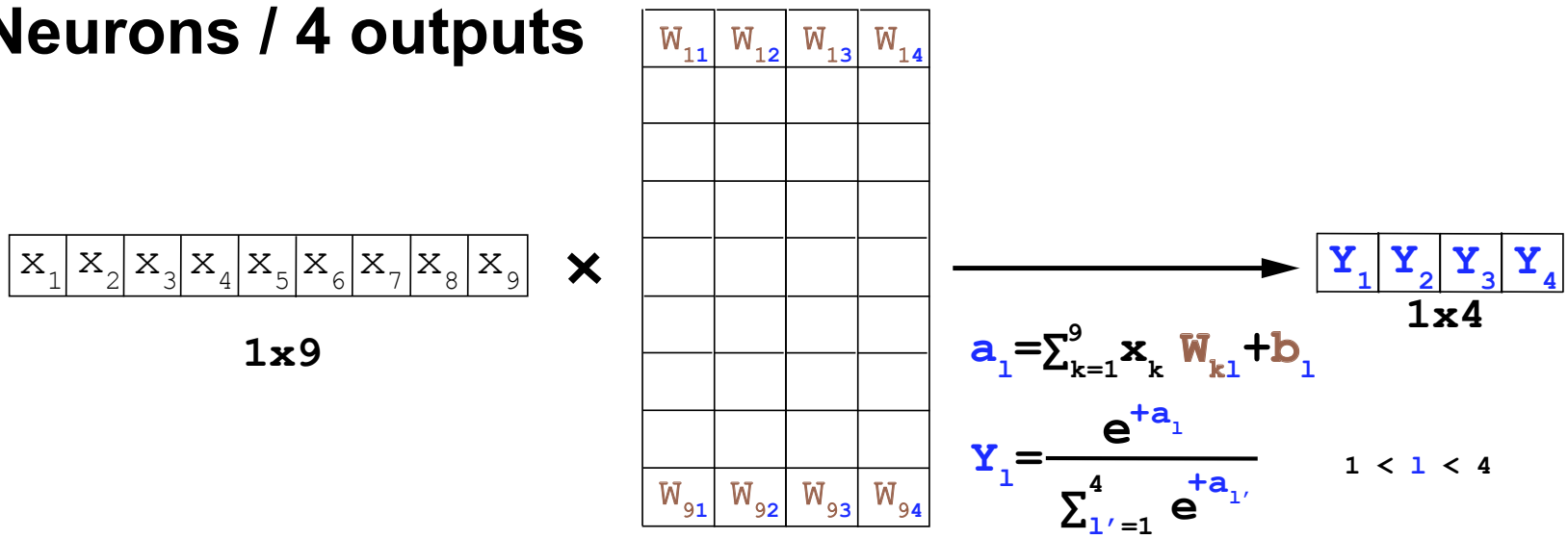
MLP

1 neuron / many outputs

1 Neuron / 1 output



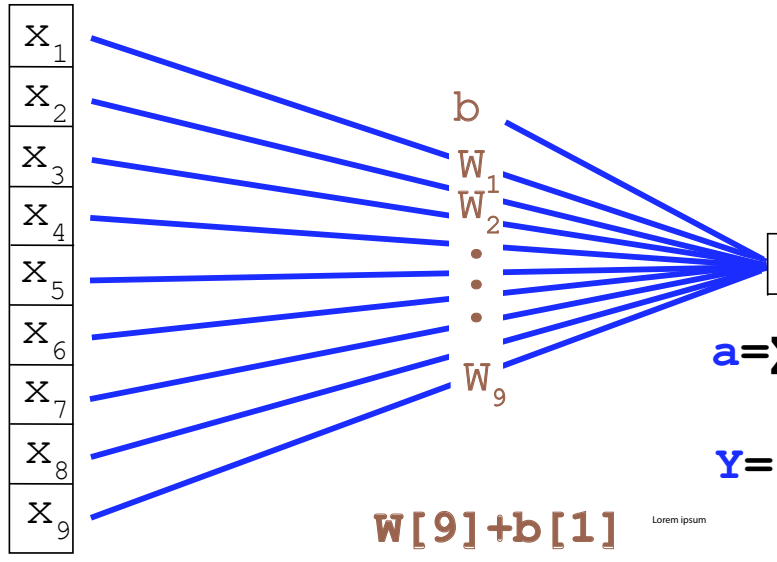
4 Neurons / 4 outputs



MLP

many neurons / many outputs

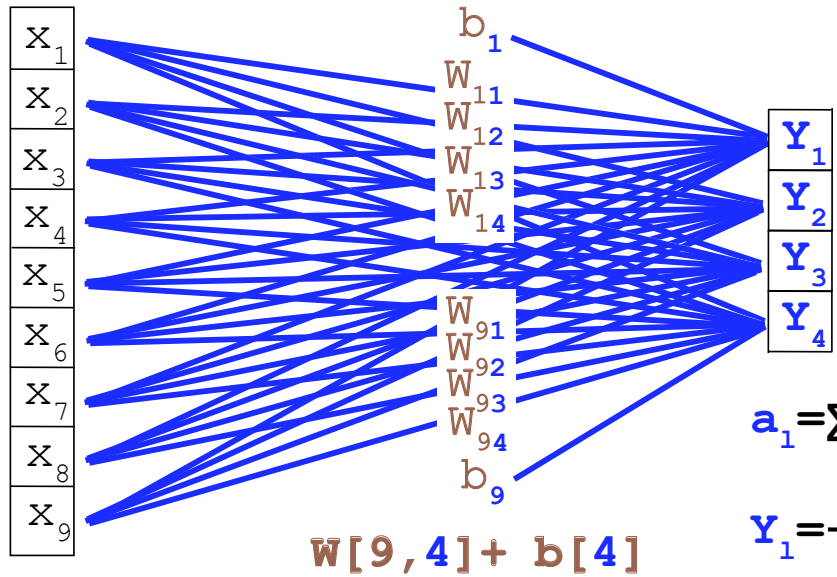
single-layer Perceptron one output



$$a=\sum_{k=1}^{9} x_k\, W_k + b$$

$$Y=\frac{1}{1+e^{-a}}$$

single-layer Perceptron many outputs



$$a_1=\sum_{k=1}^{9} x_k\, W_{k1} + b_1$$

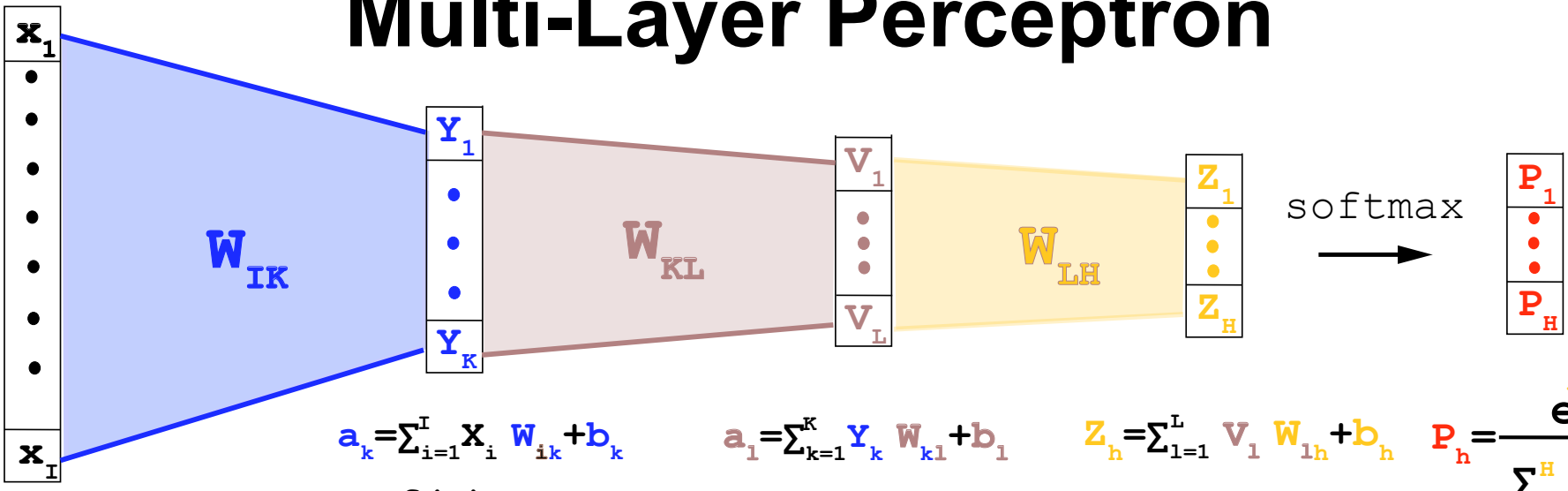
$$Y_1=\frac{e^{+a_1}}{\sum_{1'=1}^{4} e^{+a_{1'}}}$$

$1 < 1 < 4$

MLP

many layers

Multi-Layer Perceptron



$a_k = \sum_{i=1}^{I} x_i \, W_{ik} + b_k$

$Y_k = f(a)_k$

many neurons

$a_l = \sum_{k=1}^{K} Y_k \, W_{kl} + b_l$

$V_l = f_2(a)_l$

many layers

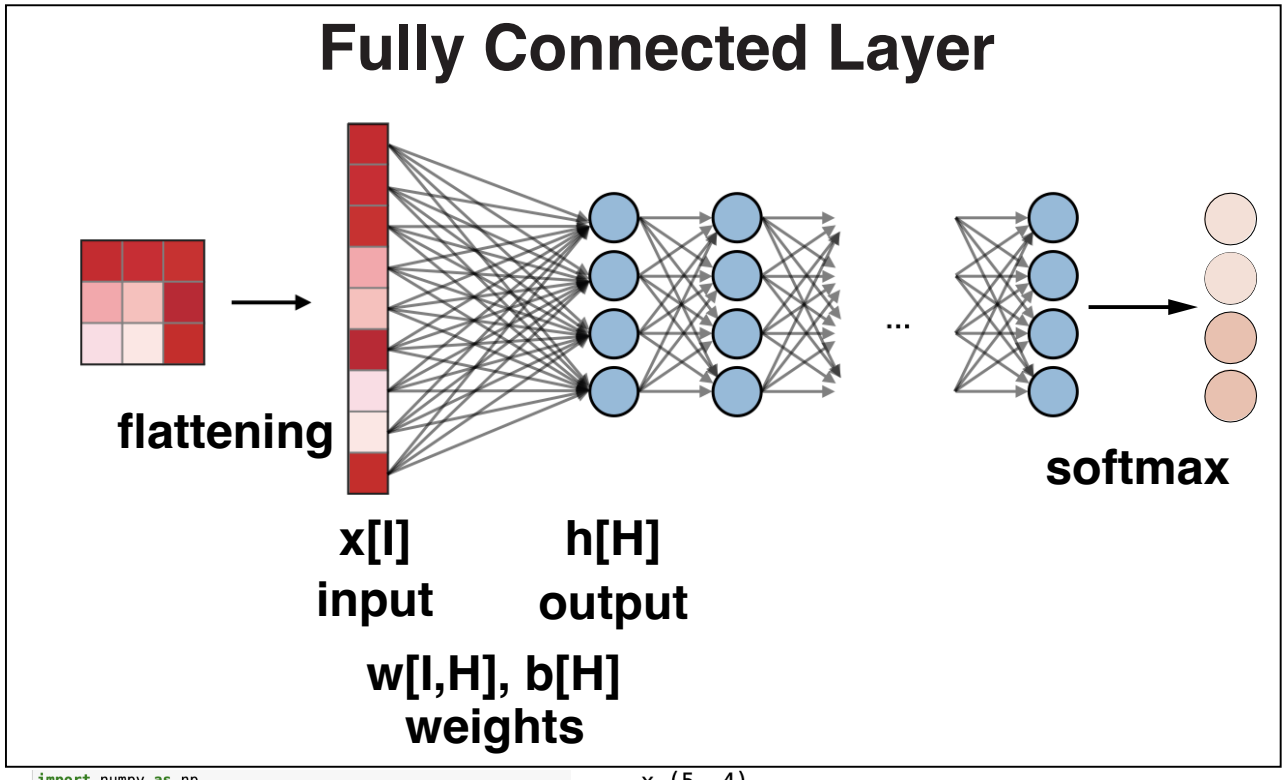
$Z_h = \sum_{l=1}^{L} V_l \, W_{lh} + b_h$

many outputs

$$P_h = \frac{e^{+Z_h}}{\sum_{h'=1}^{H} e^{+Z_{h'}}}$$

multi-class classification

MLP: input flattening




```python
import numpy as np

# DNA sequence seq="ACCTG"

# x[L,4]
x_2d = np.array([[1,0,0,0],
                 [0,1,0,0],
                 [0,1,0,0],
                 [0,0,0,1],
                 [0,0,1,0]])

# Flatten: x[L,4] -> x_flat[L*4]
x_2d_flat = x_2d.flatten()
print("x", x_2d.shape, "\n", x_2d)
print("x flatten", x_2d_flat.shape, "\n", x_2d_flat)

# x_T [4,L]
x_2d_T = x_2d.T
print("\nx_T", x_2d_T.shape, "\n", x_2d_T)

# Flatten: x_T[4,L] -> x_flat[4*L]
x_2d_T_flat = x_2d_T.flatten()
print("x_T flatten", x_2d_T_flat.shape, "\n", x_2d_T_flat)
```

```
x (5, 4)
 [[1 0 0 0]
 [0 1 0 0]
 [0 1 0 0]
 [0 0 0 1]
 [0 0 1 0]]
x flatten (20,)
 [1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0]

x_T (4, 5)
 [[1 0 0 0 0]
 [0 1 1 0 0]
 [0 0 0 0 1]
 [0 0 0 1 0]]
x_T flatten (20,)
 [1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0]
```

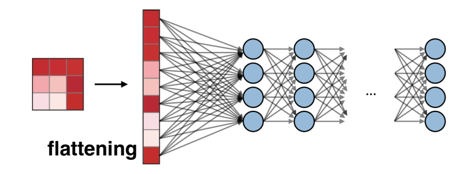
MLP: input flattening

```python
import numpy as np

# DNA sequence seq="ACCTG"

# x[L,4]
x_2d = np.array([[1,0,0,0],
                 [0,1,0,0],
                 [0,1,0,0],
                 [0,0,0,1],
                 [0,0,1,0]])

# Flatten: x[L,4] -> x_flat[L*4]
x_2d_flat = x_2d.flatten()
print("x", x_2d.shape, "\n", x_2d)
print("x flatten", x_2d_flat.shape, "\n", x_2d_flat)

# x_T [4,L]
x_2d_T = x_2d.T
print("\nx_T", x_2d_T.shape, "\n", x_2d_T)

# Flatten: x_T[4,L] -> x_flat[4*L]
x_2d_T_flat = x_2d_T.flatten()
print("x_T flatten", x_2d_T_flat.shape, "\n", x_2d_T_flat)
```

```
x (5, 4)
 [[1 0 0 0]
 [0 1 0 0]
 [0 1 0 0]
 [0 0 0 1]
 [0 0 1 0]]
x flatten (20,)
 [1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 (

x_T (4, 5)
 [[1 0 0 0 0]
 [0 1 1 0 0]
 [0 0 0 0 1]
 [0 0 0 1 0]]
x_T flatten (20,)
 [1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 (
```

MLP: weights

```python
# Weights and activation
#
# inputs:  x[I]
# weights: W[I,H]
#
# a[H] = x[I] @ W[I,H]
#
# a_h = sum_i=1^I x(i) w(i,h)
#
I = 20
H = 15
print("I", I, "H", H)

# seq = "ACCTG"
# x[I] = one_hot(seq)
x = np.array([1,0,0,0,
              0,1,0,0,
              0,1,0,0,
              0,0,0,1,
              0,0,1,0])
print("x[I] = ", x.shape, "\n", x)

# initialize weights to 1
# w[I,H]
w = np.full((I, H), 1)
print("w[I,H] = ", w.shape, "\n", w)

# activation using dot product
#
# sum product over:
# the last axis of the first array: x[I]
# and the second-to-last axis of the second array: w[I,H]
#
# a_h = sum_i=1^I x(i) w(i,h)
a = np.dot(x,w)
print("a[H] =", a.shape, "\n", a)

# activation using @ operator (shorthand for np.matmult)
# a[H] = x[I] @ W[I,H]
#
# it performs matrix multiplication in the last two indices of both tensors
#
# (different from * element-wise operator)
#
a = x@w
print("a[H] =", a.shape, "\n", a)
```



x[I]
input

h[H]
output

w[I,H], b[H]
weights

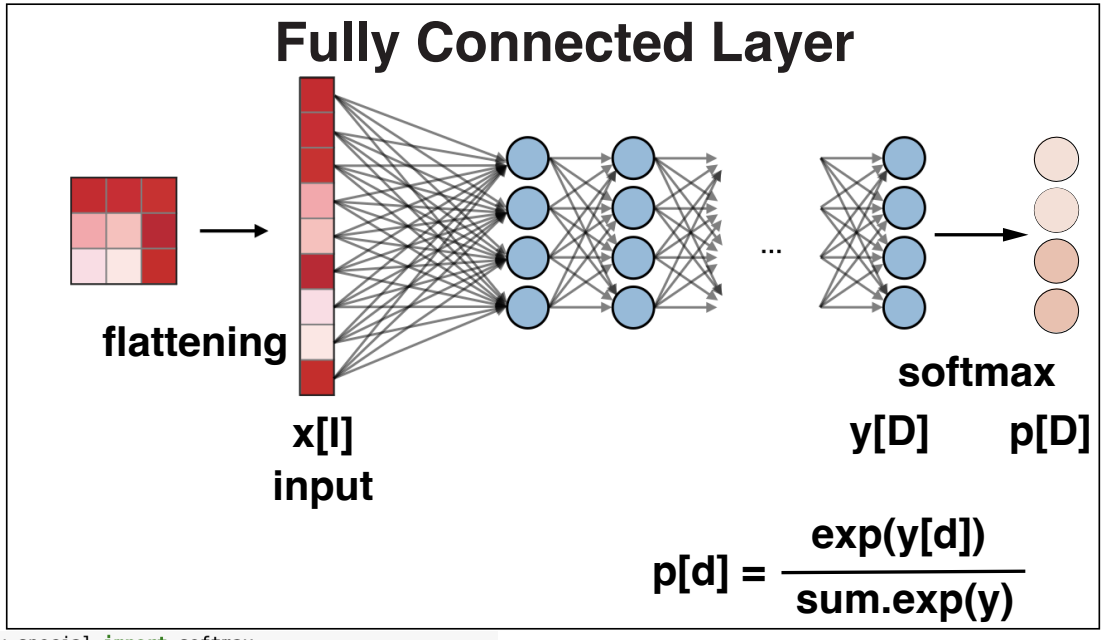
```
I 20 H 15
x[I] =  (20,)
 [1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0]
w[I,H] =  (20, 15)
 [[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]]
a[H] = (15,)
 [5 5 5 5 5 5 5 5 5 5 5 5 5 5 5]
a[H] = (15,)
 [5 5 5 5 5 5 5 5 5 5 5 5 5 5 5]
```

MLP: weights

```
# Weights and activation
#
# inputs:  x[I]
# weights: W[I,H]
#
# a[H] = x[I] @ W[I,H]
#
# a_h = sum_i=1^I x(i) w(i,h)
#
I = 20
H = 15
print("I", I, "H", H)

# seq = "ACCTG"
# x[I] = one_hot(seq)
x = np.array([1,0,0,0,
              0,1,0,0,
              0,1,0,0,
              0,0,0,1,
              0,0,1,0])
print("x[I] = ", x.shape, "\n", x)

# initialize weights to 1
# w[I,H]
w = np.full((I, H), 1)
print("w[I,H] = ", w.shape, "\n", w)

# activation using dot product
#
# sum product over:
# the last axis of the first array: x[I]
# and the second-to-last axis of the second array: w[I,H]
#
# a_h = sum_i=1^I x(i) w(i,h)
a = np.dot(x,w)
print("a[H] =", a.shape, "\n", a)

# activation using @ operator (shorthand for np.matmult)
# a[H] = x[I] @ W[I,H]
#
# it performs matrix multiplication in the last two indices of both tensors
#
# (different from * element-wise operator)
#
a = x@w
print("a[H] =", a.shape, "\n", a)
```

```
I 20 H 15
x[I] =  (20,)
 [1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0]
w[I,H] =  (20, 15)
 [[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]]
a[H] = (15,)
 [5 5 5 5 5 5 5 5 5 5 5 5 5 5 5]
a[H] = (15,)
 [5 5 5 5 5 5 5 5 5 5 5 5 5 5 5]
```

MLP: output softmax




```python
from scipy.special import softmax

# y[D]
#
#
D=20
mu = 3
sigma = 4
y = np.random.normal(loc=mu, scale=sigma, size=D)
print("\nraw scores = logits\n", y)

p = np.exp(y)/np.sum(np.exp(y))
print("\nsoftmax directly calculated\n", p)

p = softmax(y)
print("\nsoftmax using scipy.special\n", p)

imax = np.argmax(p)
print("argmax (y)", imax, "y(armax) = ",  y[imax])

imax = np.argmax(y)
print("argmax (p)", imax, "p(argmax) = ",  p[imax])

# why use softmax for classification?
```

```
raw scores = logits
 [ 8.44582642  2.81670835  3.71839622 11.15438217 10.40262056  2.63662707
 -1.86241886  4.25830586  2.83273711  3.34797951  1.68466004  0.25169234
 -2.12365387  5.65477395  1.41959583 -4.41036154 -0.81778331  1.56173462
 -6.84739061 14.47486963]

softmax directly calculated
 [2.28046583e-03 8.19084399e-06 2.01802582e-05 3.42242852e-02
 1.61379536e-02 6.84101191e-06 7.60693181e-08 3.46263326e-05
 8.32319089e-06 1.39333898e-05 2.64050088e-06 6.30022923e-07
 5.85809759e-08 1.39921638e-04 2.02567791e-06 5.95185139e-09
 2.16216573e-07 2.33507313e-06 5.20311954e-10 9.47117290e-01]

softmax using scipy.special
 [2.28046583e-03 8.19084399e-06 2.01802582e-05 3.42242852e-02
 1.61379536e-02 6.84101191e-06 7.60693181e-08 3.46263326e-05
 8.32319089e-06 1.39333898e-05 2.64050088e-06 6.30022923e-07
 5.85809759e-08 1.39921638e-04 2.02567791e-06 5.95185139e-09
 2.16216573e-07 2.33507313e-06 5.20311954e-10 9.47117290e-01]
argmax (y) 19 p =  14.474869627913282
argmax (p) 19 p =  0.9471172900724417
```

MLP: output softmax

```
from scipy.special import softmax

# y[D]
#
#
D=20
mu = 3
sigma = 4
y = np.random.normal(loc=mu, scale=sigma, size=D)
print("\nraw scores = logits\n", y)

p = np.exp(y)/np.sum(np.exp(y))
print("\nsoftmax directly calculated\n", p)

p = softmax(y)
print("\nsoftmax using scipy.special\n", p)

imax = np.argmax(p)
print("argmax (y)", imax, "y(armax) = ",  y[imax])

imax = np.argmax(y)
print("argmax (p)", imax, "p(argmax) = ",  p[imax])

# why use softmax for classification?
```

```
raw scores = logits
 [ 8.44582642  2.81670835  3.71839622 11.15438217 10.40262056  2.63662707
 -1.86241886  4.25830586  2.83273711  3.34797951  1.68466004  0.25169234
 -2.12365387  5.65477395  1.41959583 -4.41036154 -0.81778331  1.56173462
 -6.84739061 14.47486963]

softmax directly calculated
 [2.28046583e-03 8.19084399e-06 2.01802582e-05 3.42242852e-02
 1.61379536e-02 6.84101191e-06 7.60693181e-08 3.46263326e-05
 8.32319089e-06 1.39333898e-05 2.64050088e-06 6.30022923e-07
 5.85809759e-08 1.39921638e-04 2.02567791e-06 5.95185139e-09
 2.16216573e-07 2.33507313e-06 5.20311954e-10 9.47117290e-01]

softmax using scipy.special
 [2.28046583e-03 8.19084399e-06 2.01802582e-05 3.42242852e-02
 1.61379536e-02 6.84101191e-06 7.60693181e-08 3.46263326e-05
 8.32319089e-06 1.39333898e-05 2.64050088e-06 6.30022923e-07
 5.85809759e-08 1.39921638e-04 2.02567791e-06 5.95185139e-09
 2.16216573e-07 2.33507313e-06 5.20311954e-10 9.47117290e-01]
argmax (y) 19 p =  14.474869627913282
argmax (p) 19 p =  0.9471172900724417
```

Why use a softmax output?

- It is a probability distribution! what gives you more information?
  - $sc(imax) = 14.475$
  - $p(imax) = 0.95$

Why use a softmax output?

- It is a probability distribution! what gives you more information?
  - $sc(imax) = 14.475$
  - $p(imax) = 0.95$
- Can provide significance intervals

Why use a softmax output?

- It is a probability distribution! what gives you more information?
  - $sc(imax) = 14.475$
  - $p(imax) = 0.95$
- Can provide significance intervals
- Naturally works with the cross-entropy loss (next)

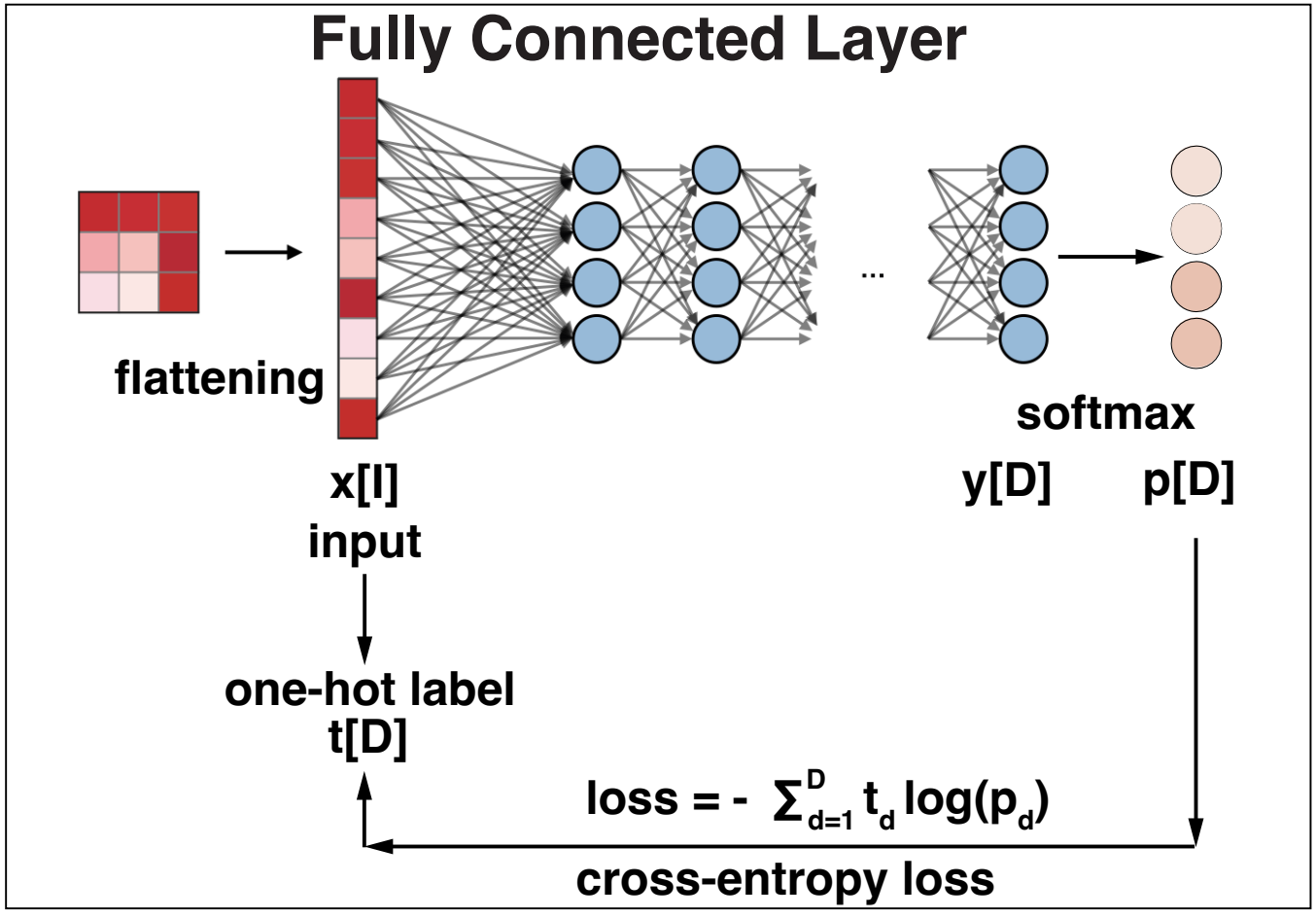
Why use a softmax output?

- It is a probability distribution! what gives you more information?
  - $sc(imax) = 14.475$
  - $p(imax) = 0.95$
- Can provide significance intervals
- Naturally works with the cross-entropy loss (next)
- It helps keep scores in range (not too large, not too small)

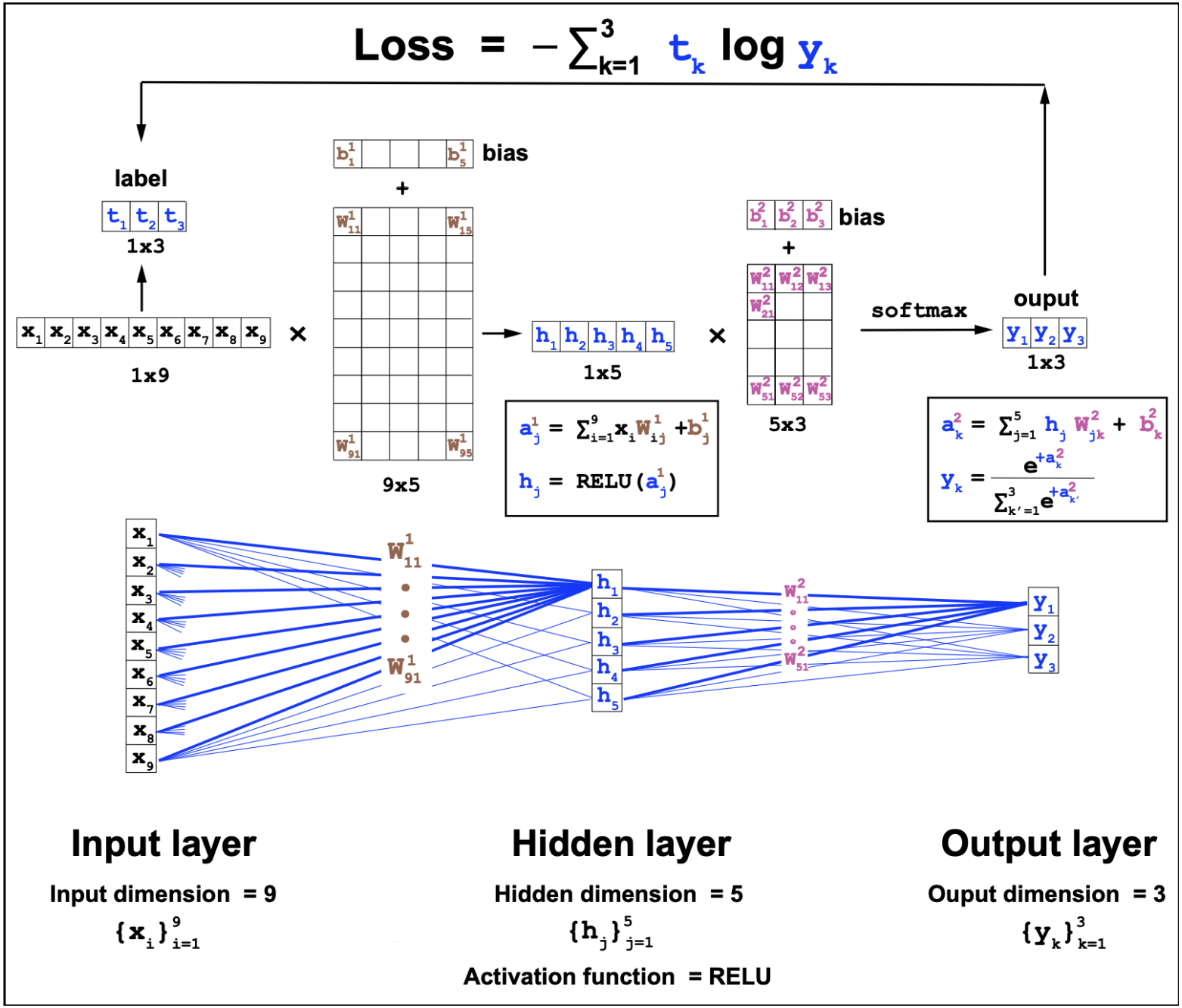
Why use a softmax output?

- It is a probability distribution! what gives you more information?
  - $sc(imax) = 14.475$
  - $p(imax) = 0.95$
- Can provide significance intervals
- Naturally works with the cross-entropy loss (next)
- It helps keep scores in range (not too large, not too small)
- It facilitates optimization by gradient descent

MLP: Loss


Fully Connected Layer
flattening
x[I]
input
...
softmax
y[D]
p[D]
one-hot label
t[D]
loss = - $\sum_{d=1}^{D} t_d \log(p_d)$
cross-entropy loss

MLP with one hidden layer



Input layer

Input dimension = 9

$\{x_i\}_{i=1}^{9}$

Hidden layer

Hidden dimension = 5

$\{h_j\}_{j=1}^{5}$

Activation function = RELU

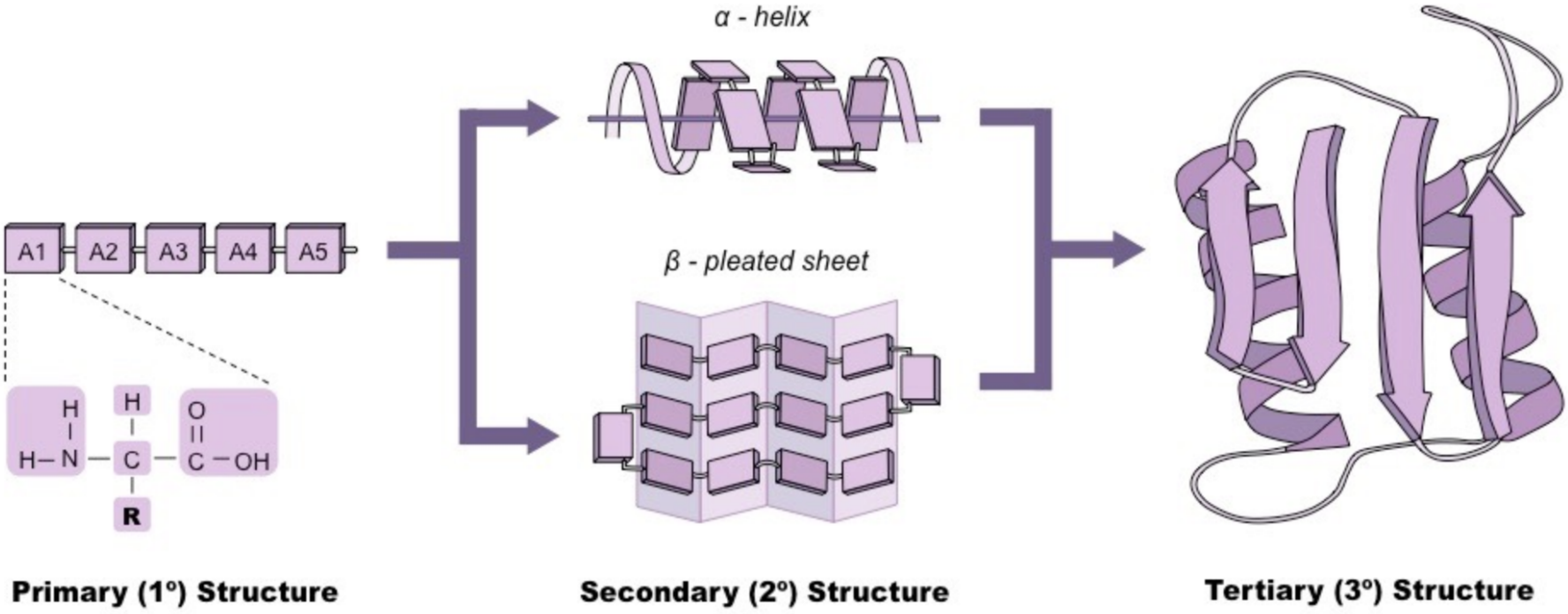
Output layer

Ouput dimension = 3

$\{y_k\}_{k=1}^{3}$

Protein 2D structure

The Qian-Sejnowski Model



Protein 2D structure

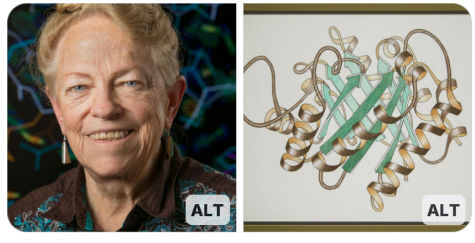
The ribbon representation

06:32

Jane Richardson was born #OTD in 1941

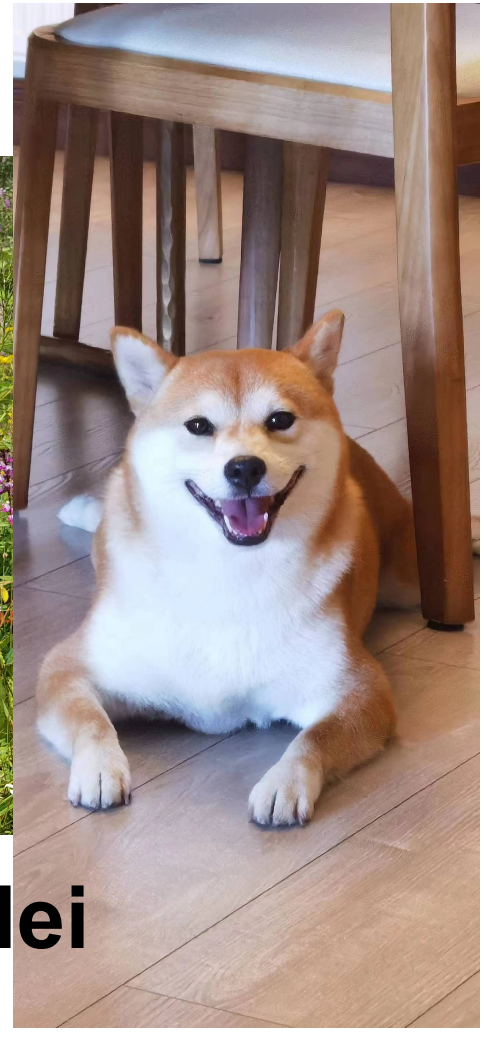
+ Developed the Richardson (ribbon) diagram to represent proteins' 3D structure (becoming a standard representation for protein structures)
+ MacArthur Fellow, 1985
+ Elected, Nat'l Academy of Sciences, 1991
+ President, Biophysical Society, 2012

#WomenInSTEM



3 18

George



MeiMei

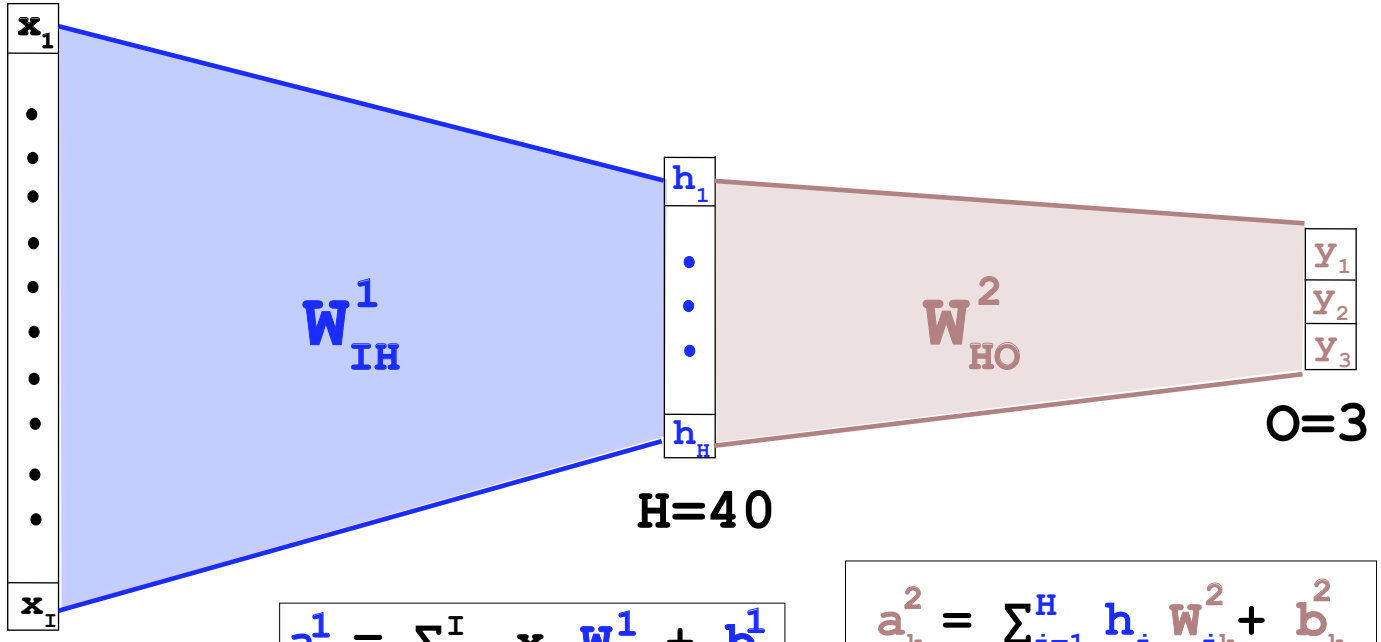
b1 - MLP Wed 3 Feb 2026

b0 homework due Fri 2/6 11:59pm

Qian-Sejnowski MLP

Protein 2D structure

Labels[3]
alpha helix = [1,0,0]
beta sheet = [0,1,0]
random coil = [0,0,1]



$$a^1_j = \sum_{i=1}^{I} x_i \, W^1_{ij} + b^1_j$$

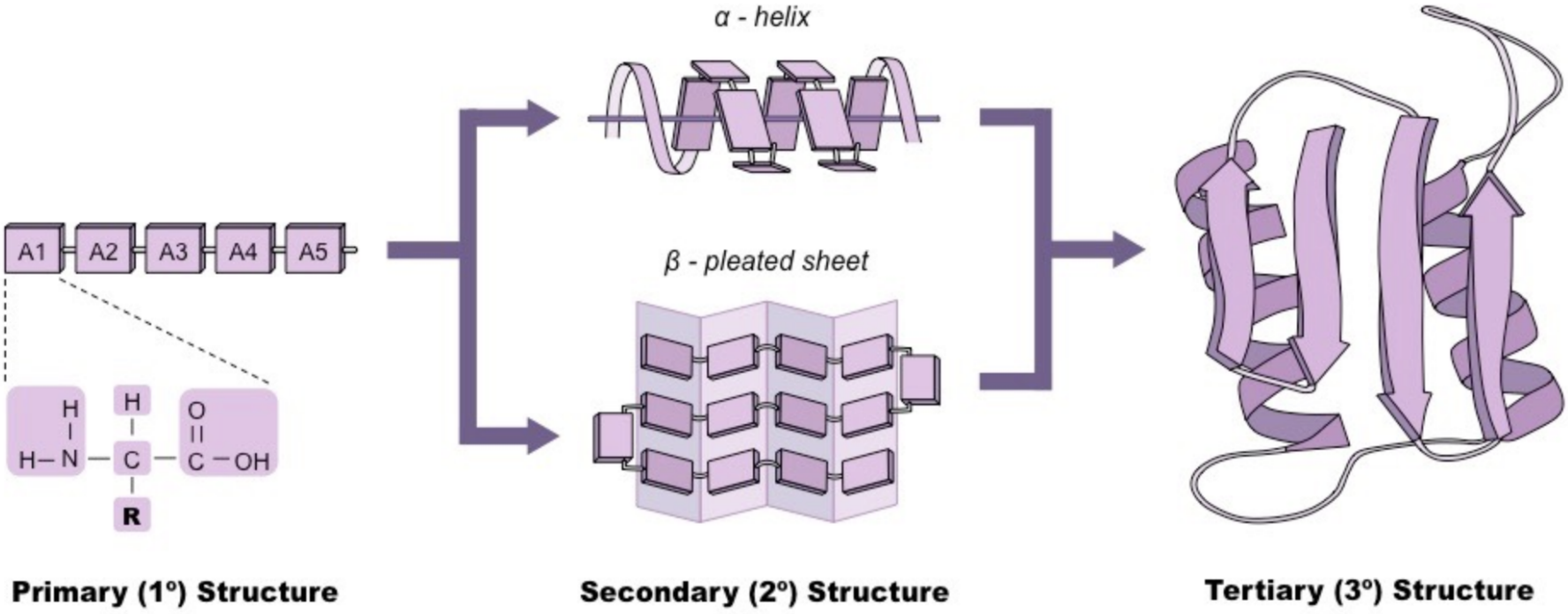
$$h_j = \frac{1}{1 + e^{-a^1_j}} \qquad 1 \le j \le H$$

$$a^2_k = \sum_{j=1}^{H} h_j \, W^2_{jk} + b^2_k$$

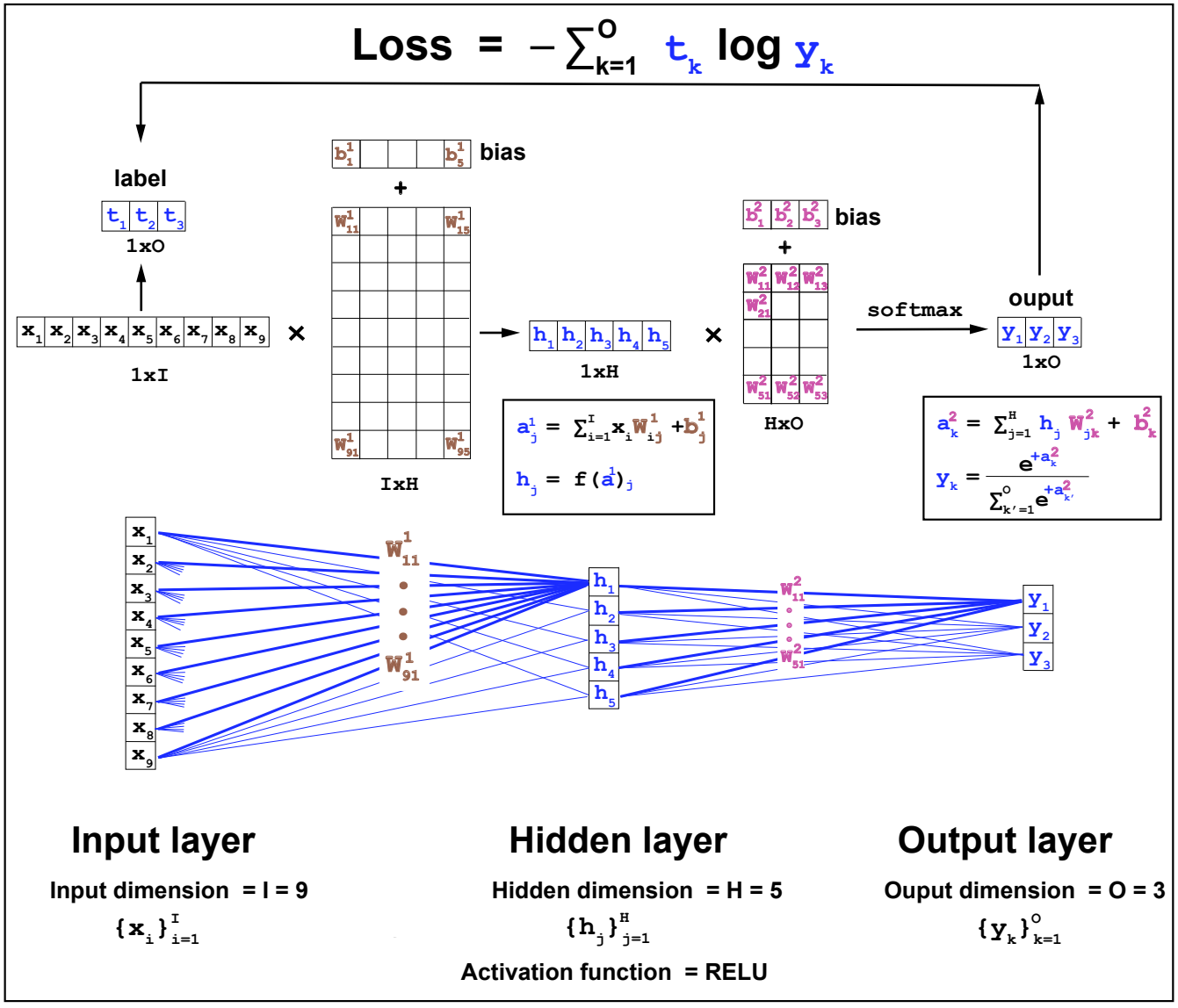
$$y_k = \frac{e^{a^2_k}}{\sum_{k=1}^{O} e^{a^2_k}} \qquad 1 \le k \le O$$

Protein 2D structure

The Qian-Sejnowski Model

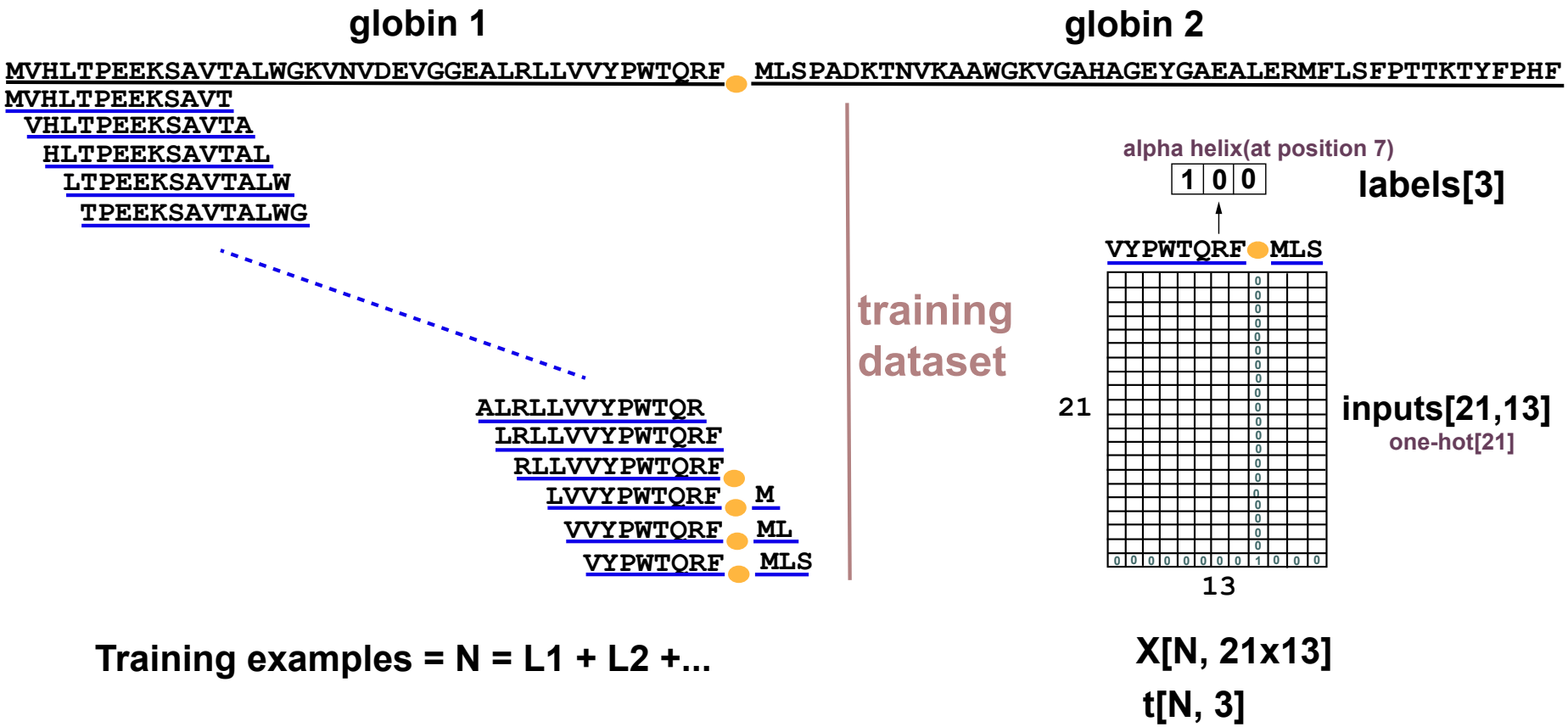


MLP with one hidden layer



Qian-Sejnowski MLP

inputs: sliding one-hot windows



Training examples = N = L1 + L2 +...

X[N, 21x13]

t[N, 3]

Training - The loss function

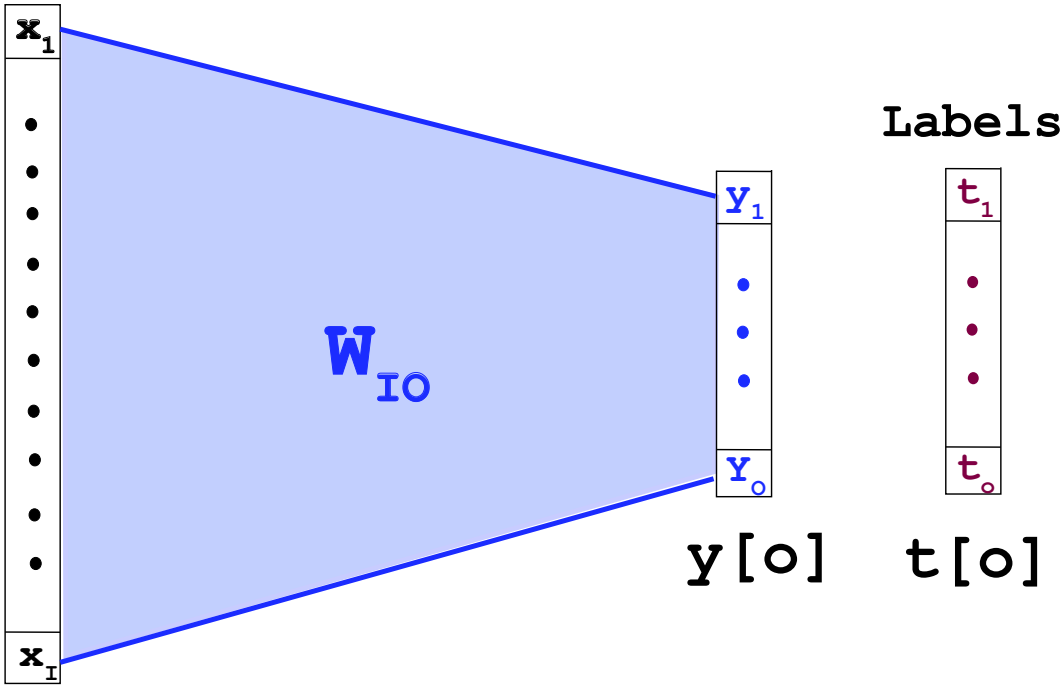
Cross Entropy

$$Loss = \frac{1}{N} \sum_{n=1}^{N} L^{(n)}(W^1, b^1, W^2, b^2).$$

$$L^{(n)}(W^1, b^1, W^2, b^2) = -\sum_{k=1}^{O} t_k^{(n)} \log y_k^{(n)}.$$

(I will drop the super index for the training example (n) after this)

Perceptron for multiclass classification

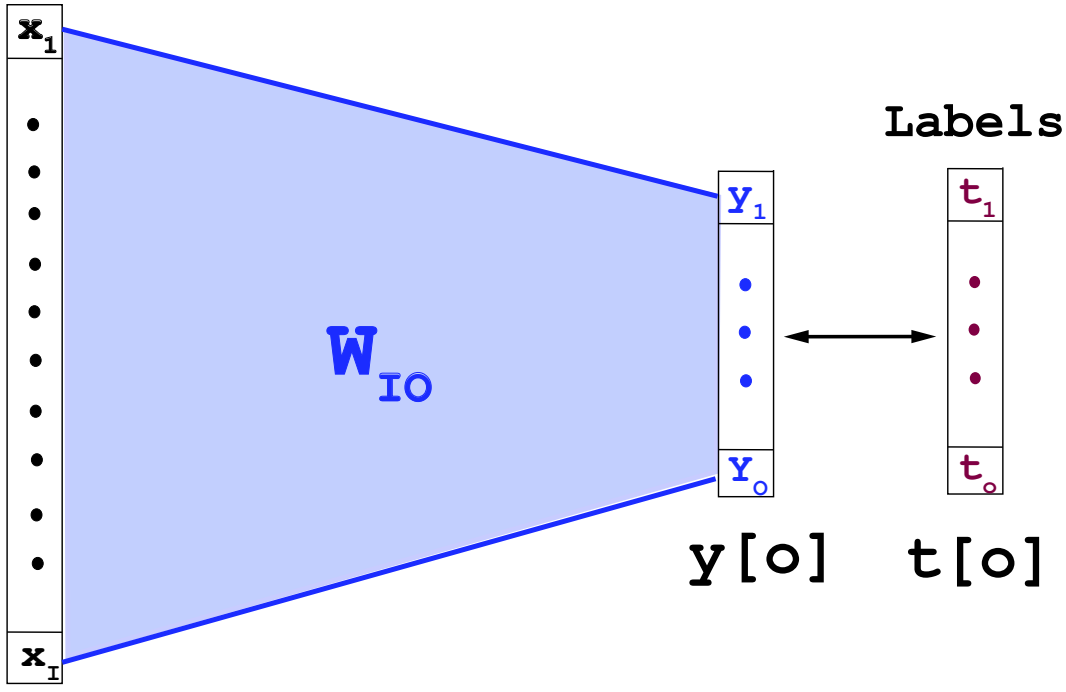


$$a_j = \sum_{i=1}^{I} x_i W_{ij} + b_j$$

$$y_j = \text{softmax}(a)_j = \frac{e^{a_j}}{\sum_{j'=1}^{O} e^{a_{j'}}}$$

$1 \le i \le I$
$1 \le j \le O$

Perceptron for multiclass classification



forward pass

$$a_j = \sum_{i=1}^{I} x_i W_{ij} + b_j$$

$$y_j = \text{softmax}(a)_j$$

$$L = -\sum_{j=1}^{O} t_j \log(y_j) \quad \begin{matrix} 1 \le i \le I \\ 1 \le j \le O \end{matrix}$$

Training a multiclass perceptron

forward pass

$$a_j = \sum_{i=1}^{I} x_i W_{ij} + b_j$$

$$y_j = softmax(a)_j = \frac{e^{a_j}}{\sum_{j'} e^{a_{j'}}}$$

$$L = -\sum_{j=1}^{O} t_j \log y_j$$

Training a multiclass perceptron

stochastic gradient descent (SGD)

$$W_{ij} \longleftarrow W_{ij} - \alpha \frac{\partial L}{\partial W_{ij}} \quad 1 \leq i \leq I$$

$$b_j \longleftarrow b_j - \alpha \frac{\partial L}{\partial b_j} \quad 1 \leq j \leq O$$

$$W[I, O] \quad b[O]$$

Training a multiclass perceptron

backpropagation

$$\frac{\partial L}{\partial W_{ij}}$$

$$\frac{\partial L}{\partial b_j}$$

$$L \rightarrow y \rightarrow a \rightarrow W, b$$

$$a_j = \sum_{i=1}^{I} x_i W_{ij} + b_j$$

$$y_j = softmax(a)_j = \frac{e^{a_j}}{\sum_{j'} e^{a_{j'}}}$$

$$L = -\sum^{O} t_j \log y_j$$

Training a multiclass perceptron

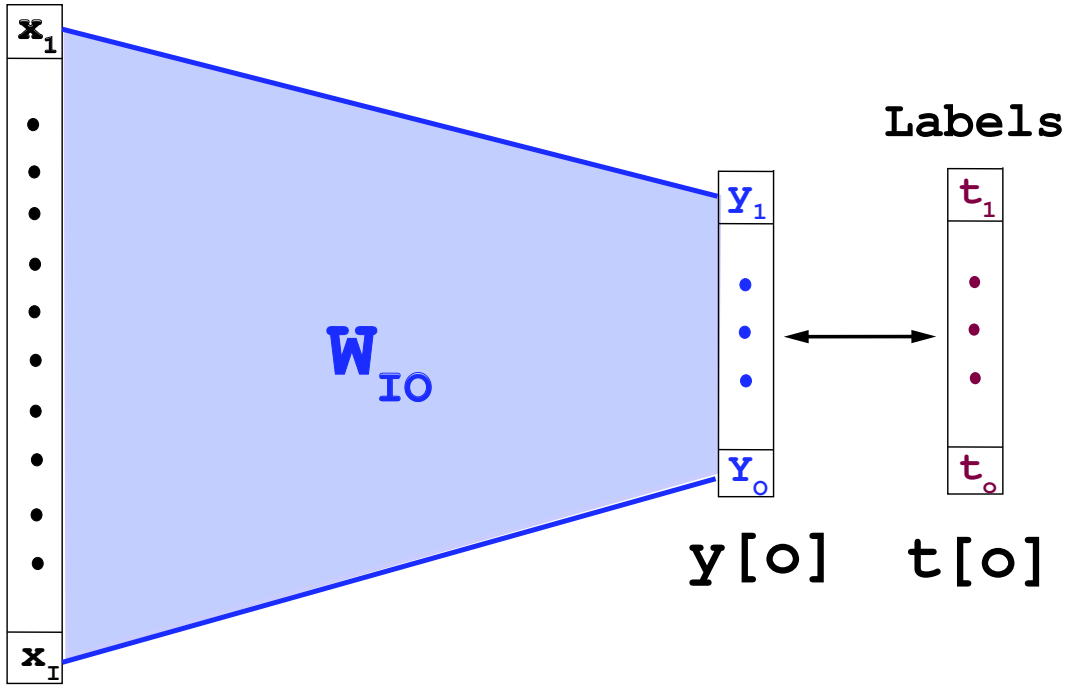
backpropagation

$$a_j = \sum_{i=1}^{I} x_i W_{ij} + b_j \qquad \delta W_{ij} =: \frac{\delta L}{\delta W_{ij}} = \delta a_j \frac{\delta a_j}{\delta W_{ij}} = \delta a_j \, x_i$$

$$y_j = \frac{e^{a_j}}{\sum_{j'} e^{a_{j'}}} \qquad \delta a_j =: \frac{\delta L}{\delta a_j} = \sum_{j'} \delta y_{j'} \frac{\delta y_{j'}}{\delta a_j} = -(t_j - y_j)$$

$$L = -\sum_{j=1}^{O} t_j \log y_j \qquad \delta y_j =: \frac{\delta L}{\delta y_j} = -\frac{t_j}{y_j}$$

Perceptron for multiclass classification



forward pass

$$a_j = \sum_{i=1}^{I} x_i W_{ij} + b_j$$

$$y_j = \text{softmax}(a)_j$$

$$L = -\sum_{j=1}^{O} t_j \log(y_j) \quad 1 \le i \le I,\ 1 \le j \le O$$

backwards pass

$$dW_{ij} = da_j x_i$$

$$da_j = -(t_j - y_j)$$

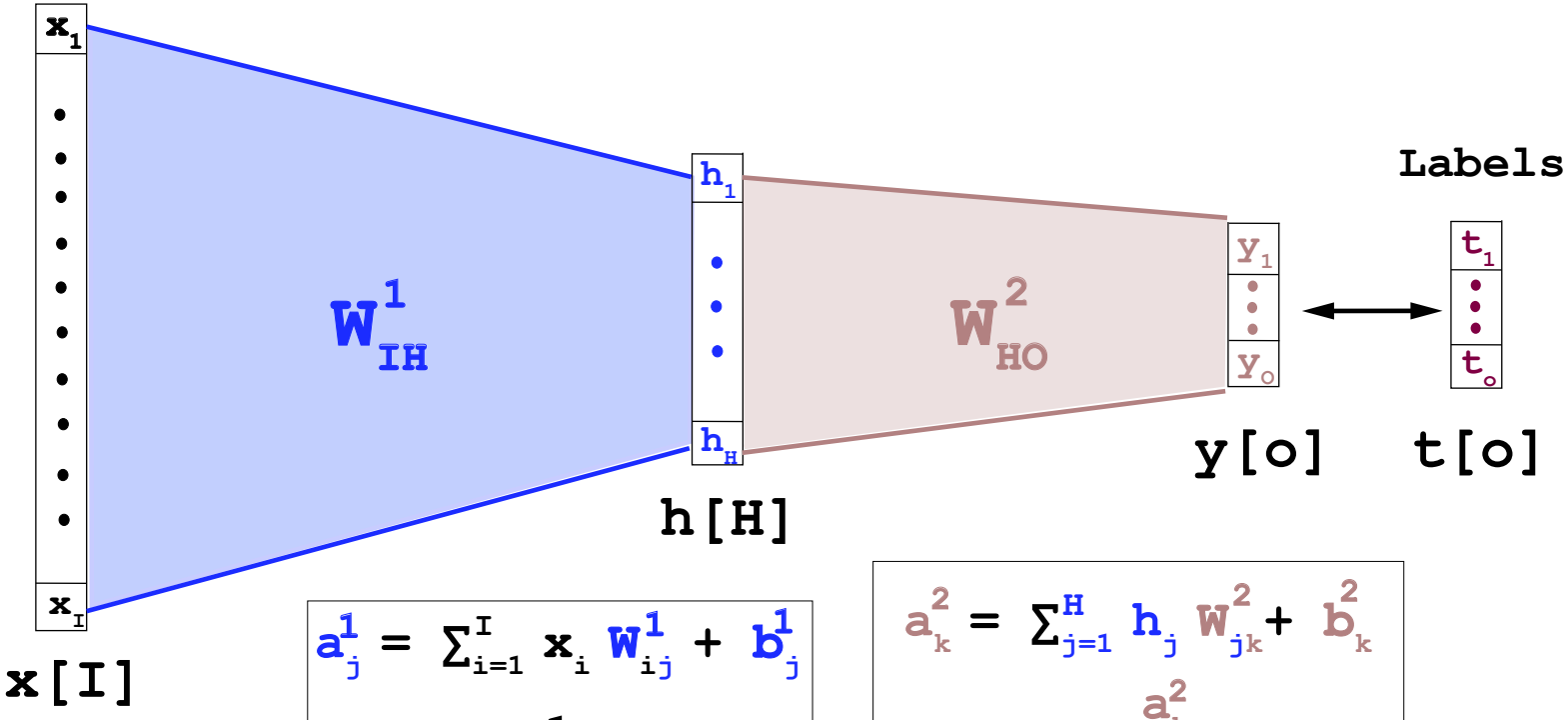
$$dy_j = -t_j / y_j \quad 1 \le i \le I,\ 1 \le j \le O$$

dv =: dL/dv

Perceptron for multiclass classification

```python
class PER:
    def __init__(self, I, O):
        # the parameters
        self.W = np.random.randn(I, O) / np.sqrt(I)
        self.b = np.zeros(O)

    def softmax(self, x):
        exps = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exps / exps.sum(axis=1, keepdims=True)

    def forward(self, X):
        # X [N,I]
        # W[I,O]
        # X @ W is [N,O]
        a = X @ self.W + self.b
        y = self.softmax(a)

        return a, y

    # X[N,I]
    # t[N,O] one-hot labels
    def train(self, X, t, epochs=20, lr=0.2):
        for epoch in range(epochs):
            # Forward
            a, y = self.forward(X)

            # cross entropy loss
            # t[N,O]
            # y[N,O]
            # L[N]
            L    = np.sum(t * np.log(y + 1e-7), axis=1)
            loss = -np.mean(L)

            # Backward
            # da = dL/da[N,O] = y-t
            # dW = dL/dW = da * da/dW = da * x
            da = (y - t) / X.shape[0]
            dW = X.T @ da
            db = da.sum(axis=0)

            # Update
            self.W -= lr * dW
            self.b -= lr * db

            # Accuracy
            if (epoch + 1) % 50 == 0 or epoch == 0:
                print(f"Epoch {epoch+1}: Loss={loss:.4f}")
```

Multilayer Perceptron (MLP) 1 hidden layer



$$a^1_j = \sum_{i=1}^{I} x_i \, W^1_{ij} + b^1_j$$

$$h_j = \frac{1}{1 + e^{-a^1_j}} \qquad 1 \le j \le H$$

$$a^2_k = \sum_{j=1}^{H} h_j \, W^2_{jk} + b^2_k$$

$$y_k = \frac{e^{a^2_k}}{\sum_{k=1}^{o} e^{a^2_k}} \qquad 1 \le k \le o$$

$$L = -\sum_{j=1}^{o} t_j \log(y_j)$$

Training 1-hidden layer MLP

stochastic gradient descent (SGD)

$$W^1_{ij} \longleftarrow W^1_{ij} - \alpha \frac{\partial L}{\partial W^1_{ij}} \quad 1 \leq i \leq I$$

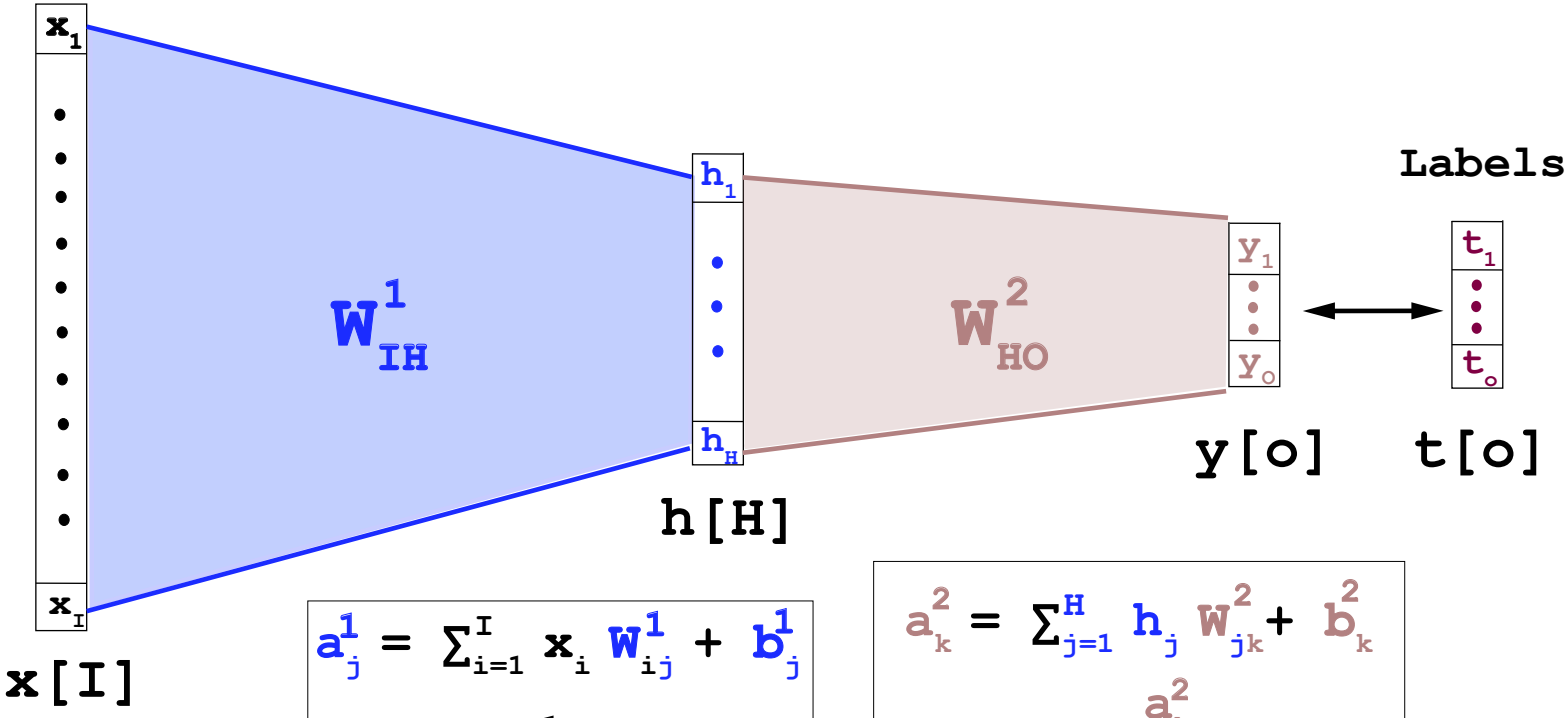
$$b^1_j \longleftarrow b^1_j - \alpha \frac{\partial L}{\partial b^1_j} \quad 1 \leq j \leq H$$

$$W^2_{jk} \longleftarrow W^2_{jk} - \alpha \frac{\partial L}{\partial W^2_{jk}} \quad 1 \leq k \leq O$$

$$b^2_k \longleftarrow b^2_k - \alpha \frac{\partial L}{\partial b^2_k}.$$

$$\begin{array}{cc} W^1[I,H] & b^1[H] \\ W^2[H,O] & b^2[O] \end{array}$$

Multilayer Perceptron (MLP) 1 hidden layer



$$a^1_j = \sum_{i=1}^{I} x_i \, W^1_{ij} + b^1_j$$

$$h_j = \frac{1}{1 + e^{-a^1_j}} \qquad 1 \le j \le H$$

$$a^2_k = \sum_{j=1}^{H} h_j \, W^2_{jk} + b^2_k$$

$$y_k = \frac{e^{a^2_k}}{\sum_{k=1}^{O} e^{a^2_k}} \qquad 1 \le k \le O$$

$$L = -\sum_{j=1}^{O} t_j \log(y_j)$$

$$L \longrightarrow y_k \longrightarrow a^2_k \longrightarrow W^2_{jk}, b^2_k$$

$$L \longrightarrow y_k \longrightarrow a^2_k \longrightarrow h_j \longrightarrow a^1_j \longrightarrow W^1_{ij}, b^1_j$$

Training 1-hidden layer MLP

backpropagation

$$a_j^1 = \sum_{i=1}^{I} x_i W_{ij}^1 + b_j^1 \qquad \delta W_{ij}^1 = \delta a_j^1 \frac{\delta a_j^1}{\delta W_{ij}^1} = \delta a_j^1 \, x_i$$

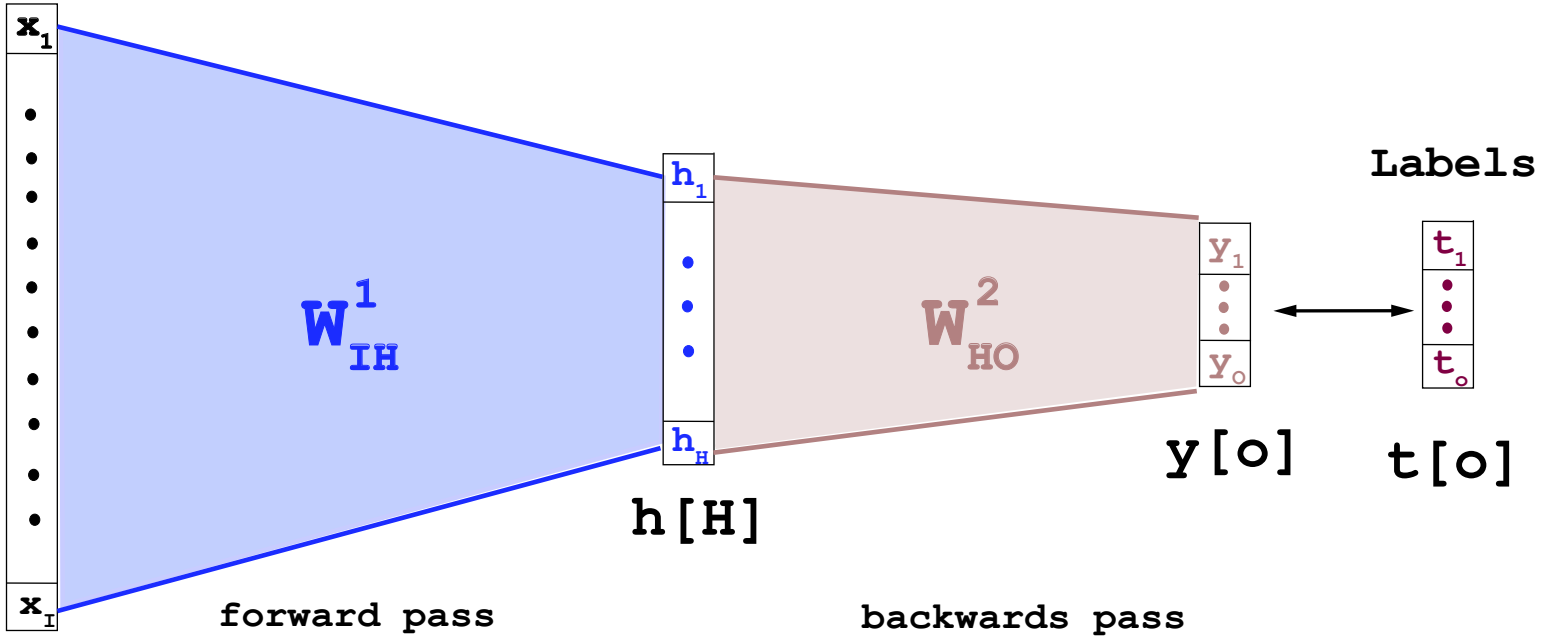
$$h_j = f(a^1)_j \qquad \delta a_j^1 = \sum_{j'} \delta h_{j'} \frac{df_{j'}}{da_j^1} \qquad \delta h_j = \sum_{k=1}^{0} \delta a_k^2 W_{jk}^2$$

$$a_k^2 = \sum_{j=1}^{H} h_j W_{jk}^2 + b_k^2 \qquad \delta W_{jk}^2 = \delta a_j^2 \frac{\delta a_k^2}{\delta W_{jk}^2} = \delta a_k^2 \, h_j$$

$$y_k = softmax(a^2)_k \qquad \delta a_k^2 = \sum_{k'} \delta y_{k'} \frac{\delta y_{k'}}{\delta a_k} = y_k - t_k$$

$$L = -\sum_{k=1}^{O} t_k \log y_k \qquad \delta y_j = -\frac{t_j}{y_j}$$

Multilayer Perceptron (MLP) 1 hidden layer



forward pass

$$a^1_j = \sum_{i=1}^{I} x_i \, W^1_{ij} + b^1_j$$

$$h_j = f(a^1)_j$$

$$a^2_k = \sum_{j=1}^{H} h_j \, W^2_{jk} + b^2_k$$

$$y_k = \text{softmax}(a^2_k)$$

$$L = -\sum_{j=1}^{O} t_j \log(y_j)$$

$1 \le i \le I$, $1 \le j \le H$, $1 \le k \le O$

backwards pass

$$dW^1_{ij} = da^1_j \, x_i$$

$$da^1_j = dh_j \, f'(a^1)_j$$

$$dh_j = \sum_{k=1}^{O} da^2_k \, W^2_{jk}$$

$$dW^2_{jk} = da^2_k \, h_j$$

$$da^2_k = y_k - t_k$$

$$dy_k = -t_k / y_k$$

$1 \le i \le I$, $1 \le j \le H$, $1 \le k \le O$

dv =: dL/dv

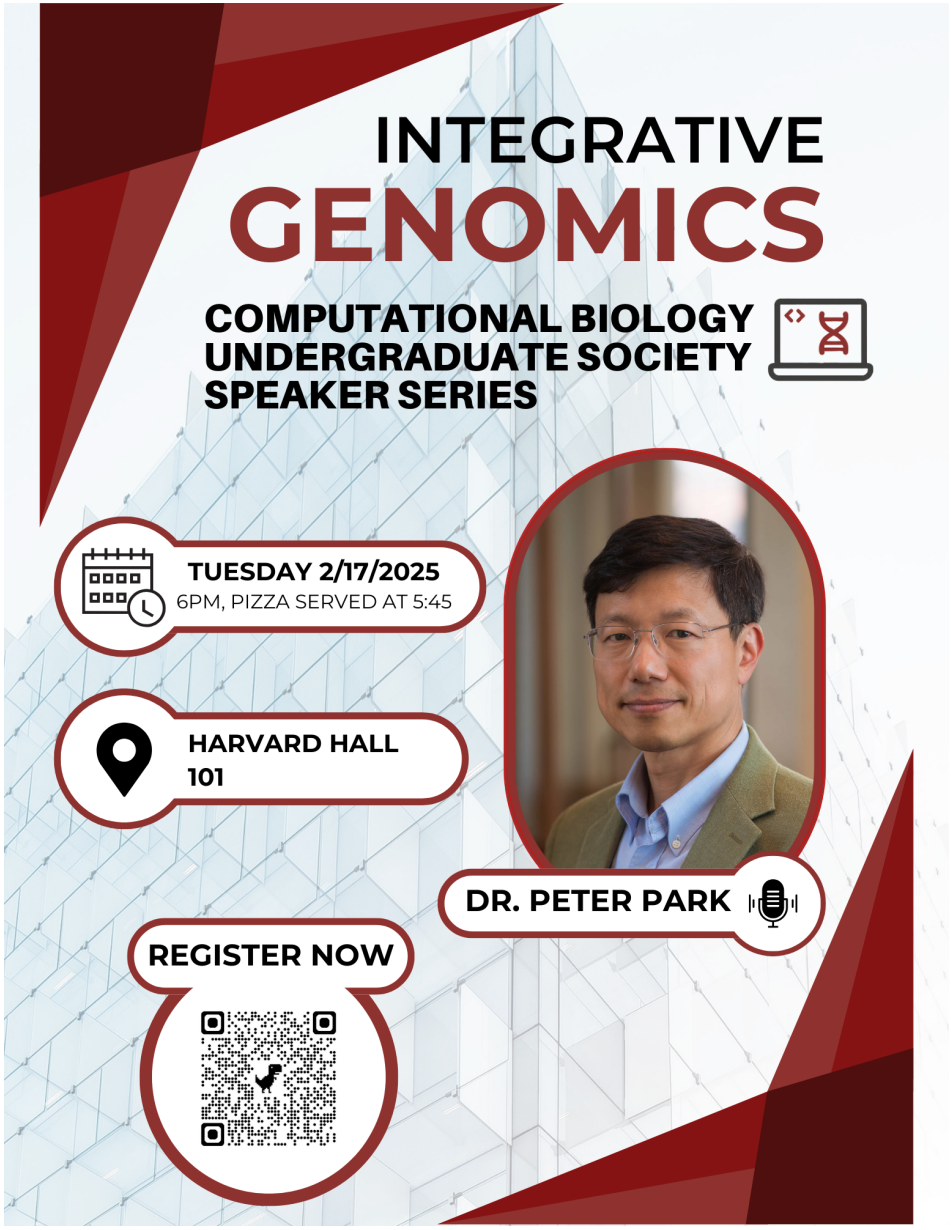
backpropagation by hand

Multilayer Perceptron (MLP) 1 hidden layer

```
# cross entropy loss
loss = -np.mean(np.sum(Y_onehot * np.log(probs + 1e-7), axis=1))

# Backward
#
# (h1[N,H])^T @ da2[N,O] = dW2[H, 0]
#
da2 = (probs - Y_onehot) / X.shape[0]
dW2 = h1.T @ da2
db2 = da2.sum(axis=0)

# dW1 = dL/dW1 = dL/da1 * da1/dW1 = da1 * X
#
# da2[N,O] @ (W2[H,O])^T = dh1[N, H]
#
dh1 = da2 @ self.W2.T
da1 = dh1 * self.relu_deriv(a1)

#
# (x[N,I])^T @ da1[N,H] = dW1[I, H]
#
dW1 = X.T @ da1
db1 = da1.sum(axis=0)

# Update
self.W2 -= lr * dW2
self.b2 -= lr * db2
self.W1 -= lr * dW1
self.b1 -= lr * db1
```

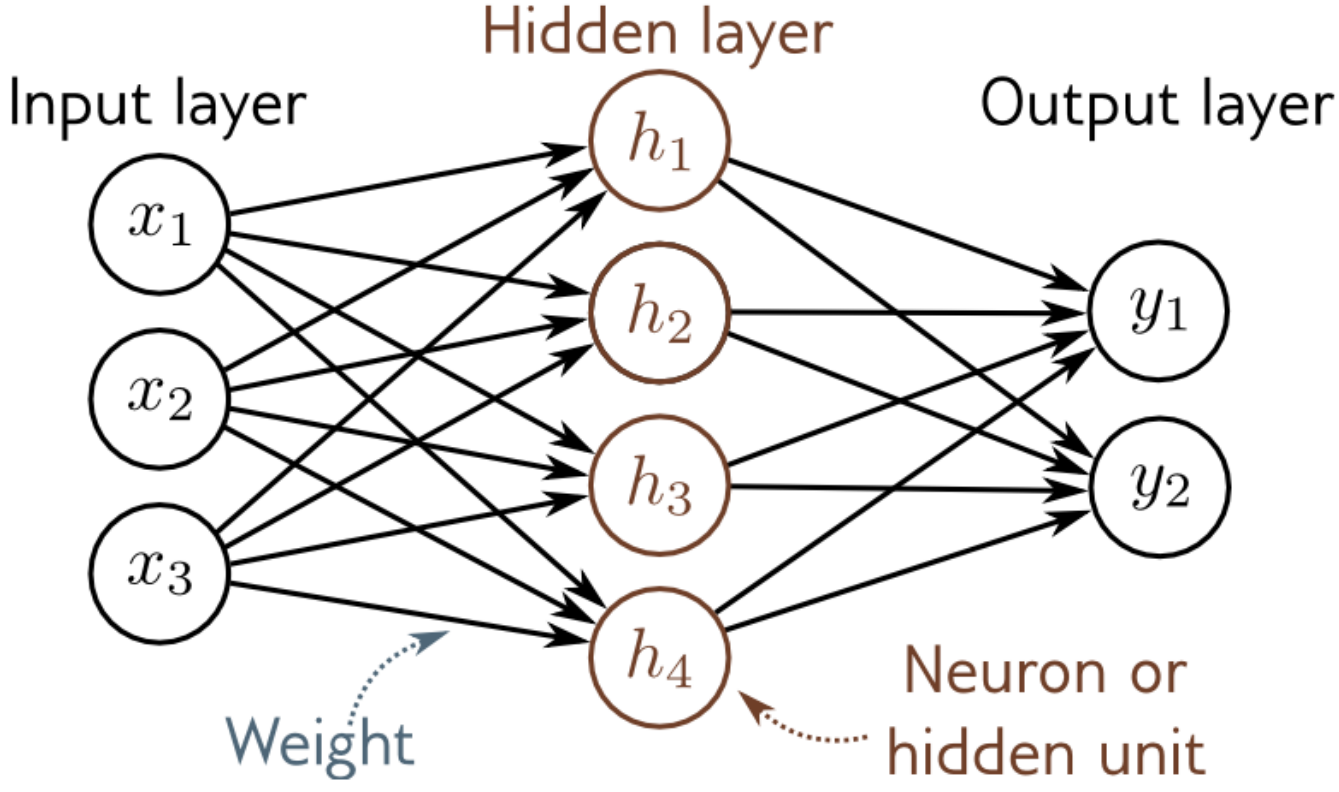
CBUGS talks

Joy Xu joyxu@college.harvard.edu
Adi Madduri adithyamadduri@college.harvard.edu



Multilayer Perceptron (MLP) 1 hidden layer
Feed Forward Network (FFN)

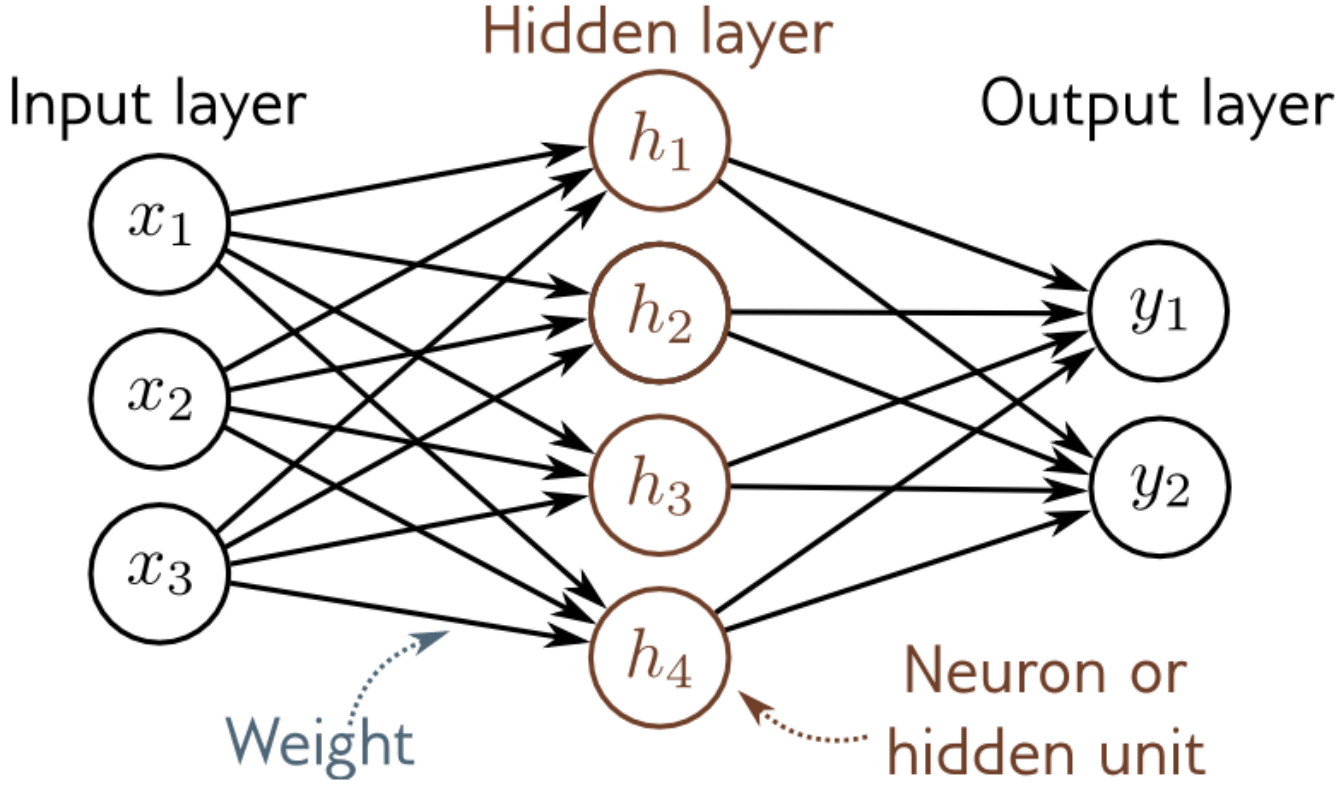
Fully connected layer (FC)



adapted from UDL, Prince Fig 3.12

Multilayer Perceptron (MLP) 1 hidden layer
Feed Forward Network (FFN)

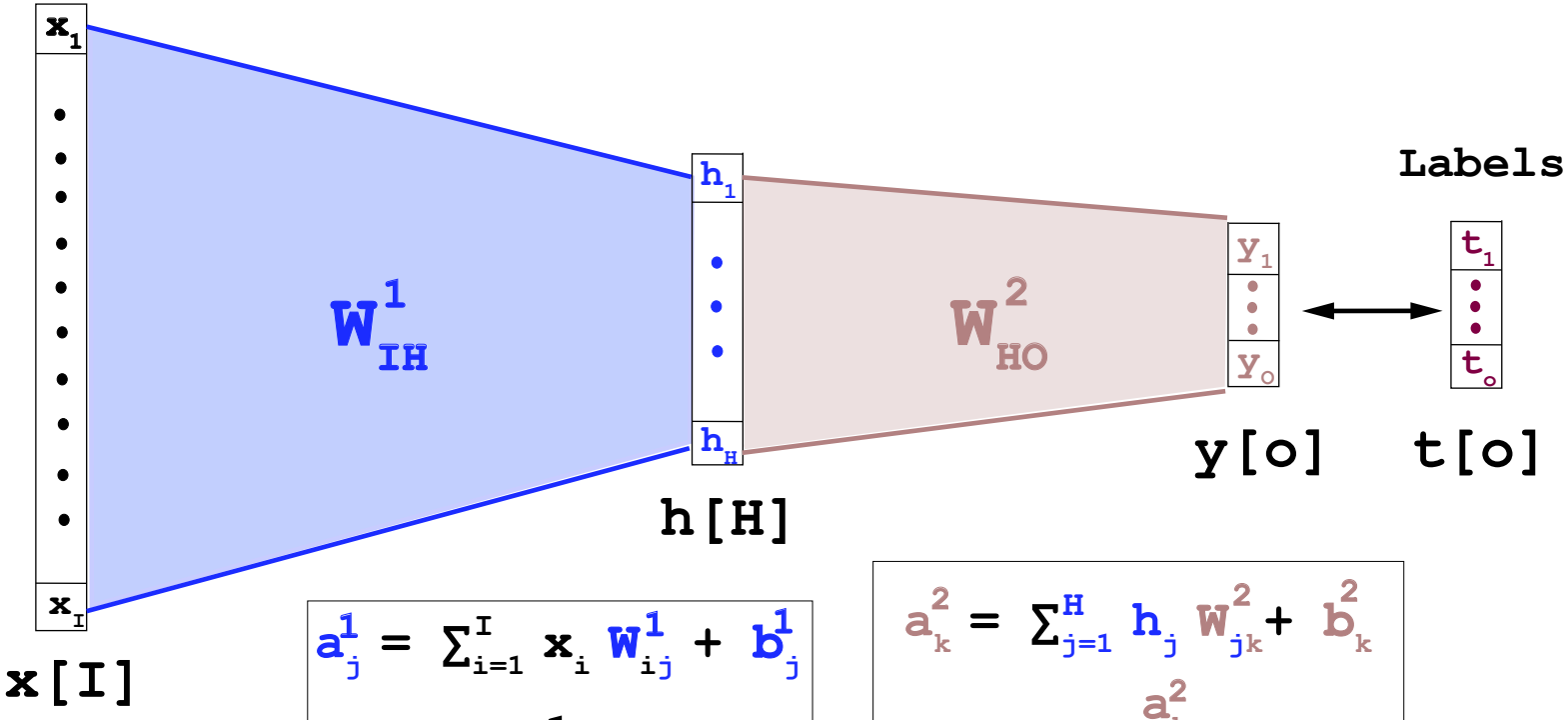
Fully connected layer (FC)



number of parameter:	Input-hidden:	3 x 4 + 4 = 16
	hidden-output:	4 x 2 + 2 = 10
	total	26

adapted from UDL, Prince Fig 3.12

Multilayer Perceptron (MLP) 1 hidden layer



$$a^1_j = \sum_{i=1}^{I} x_i \, W^1_{ij} + b^1_j$$

$$h_j = \frac{1}{1 + e^{-a^1_j}} \qquad 1 \le j \le H$$

$$a^2_k = \sum_{j=1}^{H} h_j \, W^2_{jk} + b^2_k$$

$$y_k = \frac{e^{a^2_k}}{\sum_{k=1}^{o} e^{a^2_k}} \qquad 1 \le k \le o$$

$$L = -\sum_{j=1}^{o} t_j \log(y_j)$$

PyTorch is all you need for backpropagation

Multilayer Perceptron (MLP) 1 hidden layer

pytorch code

```
class ProteinMLP(nn.Module):
    # MPL definition
    def __init__(self, I, H, O):
        super().__init__()
        self.fc1 = nn.Linear(I, H)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(H, O)

    # Definition of the forward pass
    # inputs are X outputs are Y in logits (not normalized)
    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

#  Qian & Sejnowski parameters
w = 13
I = w*20
H = 40
O = 3

# Training set converted to PyTorch tensors
# X are the inputs [N,I]
# Y are the labels [N]
X, Y = encode_dataset(seqs, sst3, w)
X_torch = torch.tensor(X, dtype=torch.float32)
Y_torch = torch.tensor(Y, dtype=torch.long)

# Create the model
model = ProteinMLP(I, H, O)
# Create the Loss
criterion = nn.CrossEntropyLoss()
# The optimization: Adam
optimizer = optim.Adam(model.parameters(), lr=0.1)

# The training loop
n_epochs = 1000
for epoch in range(n_epochs):
    optimizer.zero_grad()             # clear the gradients of all parameters
    logits = model(X_torch)           # Forward pass: get logits (scores for each class).
    loss = criterion(logits, Y_torch) # calculate cross-entropy loss
    loss.backward()                   # backward pass: calculate gradients dW = dL/dW
    optimizer.step()                  # update the parameters: W <- W - lr * dW
    pred = logits.argmax(dim=1)       # predict the class with higher probability
```

- Define the MLP model: 1 hidden layer, RELU activation
- Sets dimensions: Input = I, Outputs = O, Hidden = H
- Assign Loss: CrossEntropyLoss
- Assign optimizer for parameter updates: Adam.

For 1000 epochs:

- Initialize optimizer
- Forward pass: get logits
- Compute cross-entropy loss
- Backpropagate gradients
- Update parameters

The Cross-Entropy Loss

K-class classification

input $\mathbf{x}$
label $t \in \{1, \ldots K\}$
Outputs (logits) $\mathbf{y}(\theta) = (y_1, \ldots, y_K)$

covert to categorical probability distributions
$\mathbf{q} = (0, 1, 0, \ldots, 0)$ if $t = 2$
$\mathbf{p}(\theta) = softmax(y)$

objective: optimize weights θ so that $p(\theta) \approx q$

minimize KL divergence $KL(q||p) = \sum_k q_k \log \frac{q_k}{p_k}$

The Cross-Entropy Loss

$$\theta^* = argmin_\theta \sum_k q_k \log \frac{q_k}{p_k(\theta)}$$

$$= argmin_\theta \left[\sum_k q_k \log q_k - \sum_k q_k \log p_k(\theta) \right]$$

$$= argmin_\theta - \left[\sum_k q_k \log p_k(\theta) \right]$$

The cross-entropy

Minimizing the cross entropy is equivalent to minimizing the KL divergence

The Cross-Entropy Loss

why entropy?

The **Shannon information content** of an event with probability p is

$$\log \tfrac{1}{p} = -\log p$$

it quantifies:

"How much information I get by seen this event to happen?"

The Cross-Entropy Loss

The average information content in a probability distribution is called the **entropy** of the distribution

$$H(p) = -\sum_k p_k \log p_k$$

The **maximal information** of a categorical distribution, corresponds to the uniform distribution ($p_k = 1/K$)

$$H(p) = log(K)$$

The **cross-entropy**

$$H(p||q) = -\sum_k p_k \log q_k$$

is the amount of uncertainty in one distribution after taking into account what we know from the other.

The Cross-Entropy Loss

What values can it take?

$$0 \leq H(p||q) \leq +\infty$$

What values can it take for a perfect prediction?

For one-hot labels $t \in \{1, \ldots K\}$ and softmax logit outputs y,

$$H(t||y) = -\sum_k q_k \log y_k = 1 \log(y_t)$$

perfect classification means: $y_{k=t} = 1$, and

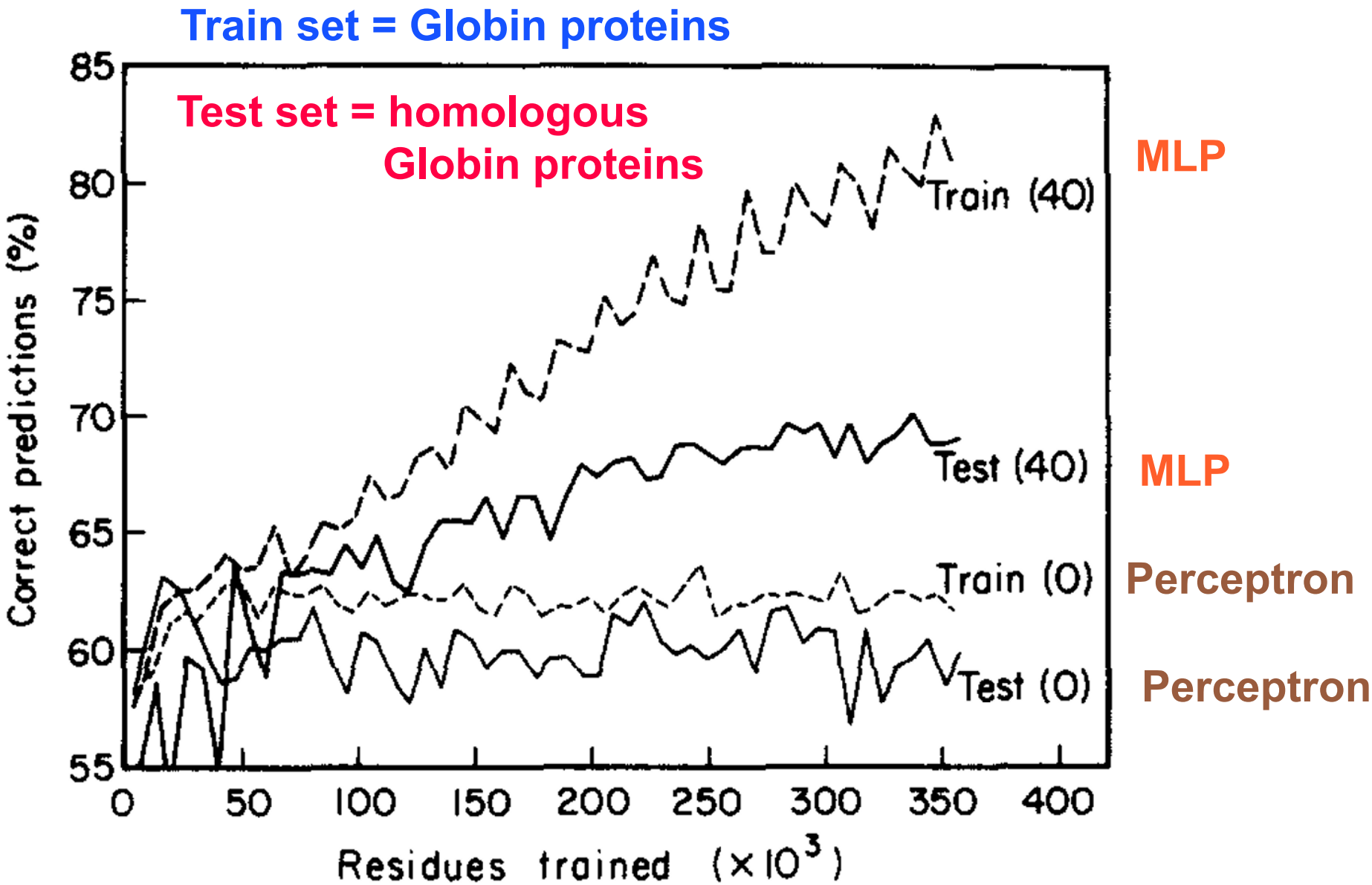
$$H(t||y) = -\sum_k q_k \log(y_k) = 1 \log(y_t) = \mathbf{0}$$

Testing the performance of the model

Large # parameters $\longrightarrow$ overfit to training data

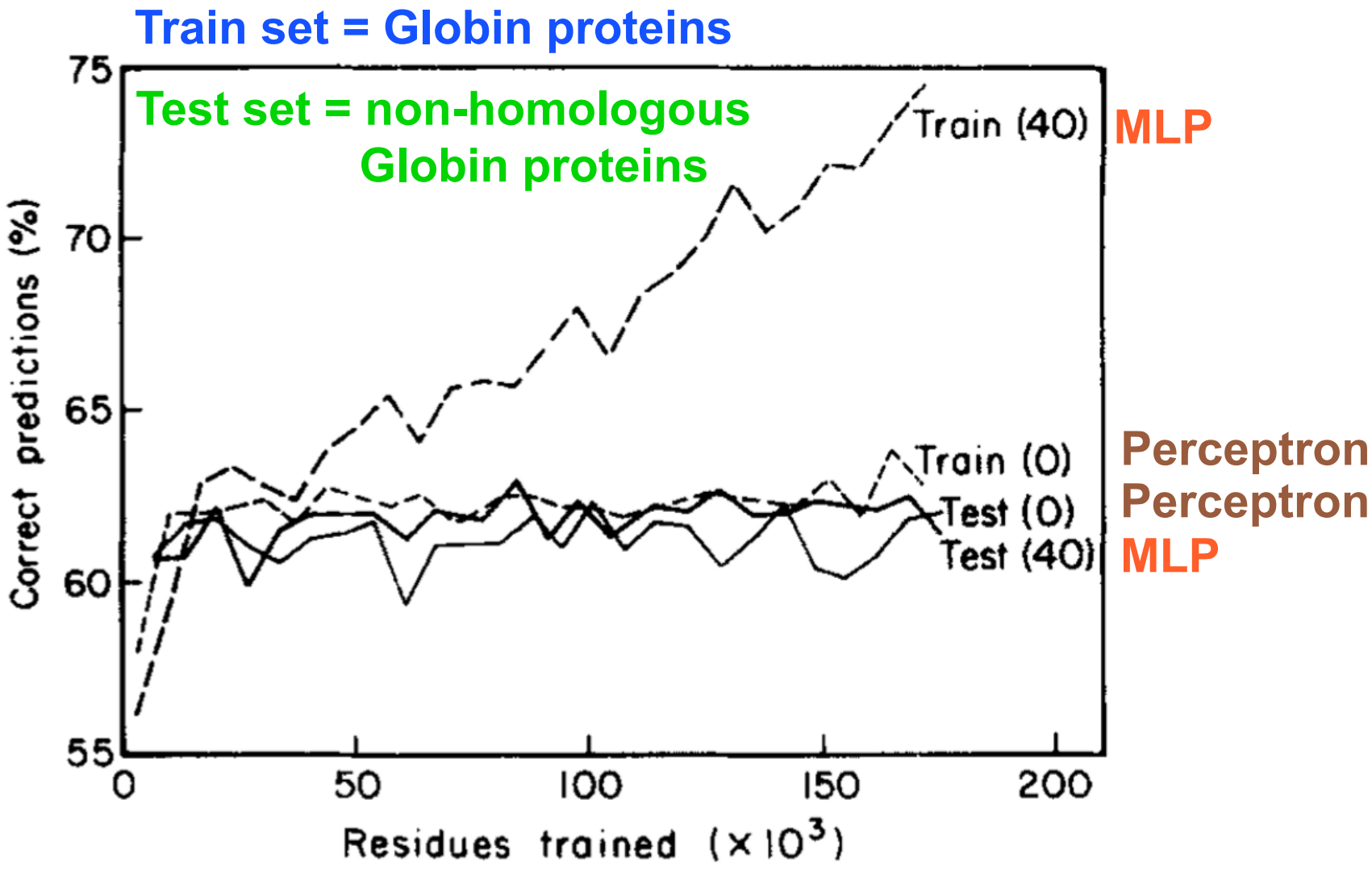
Testing Overfitting

Train-set/Test-set leakage



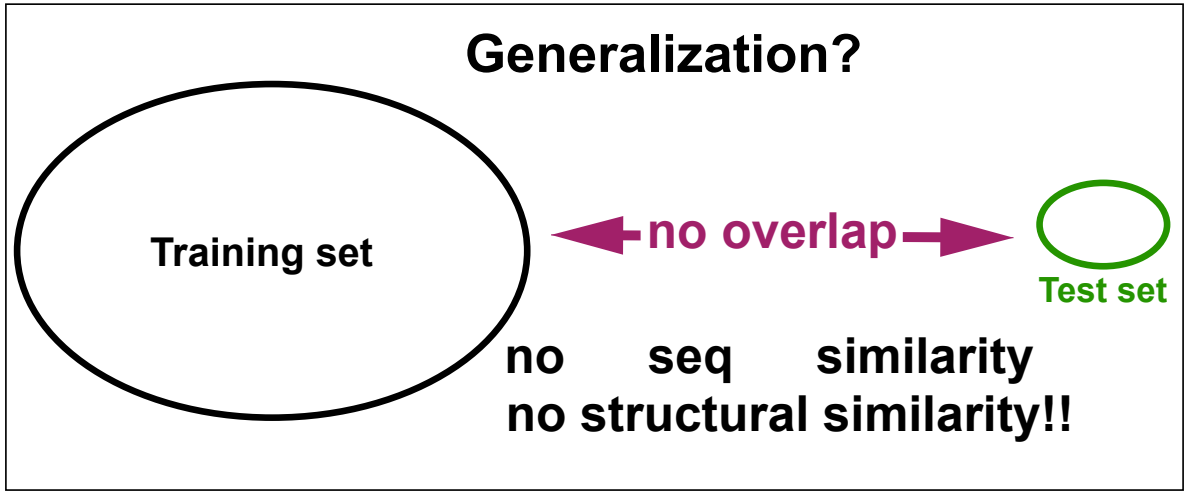
Testing overfitting

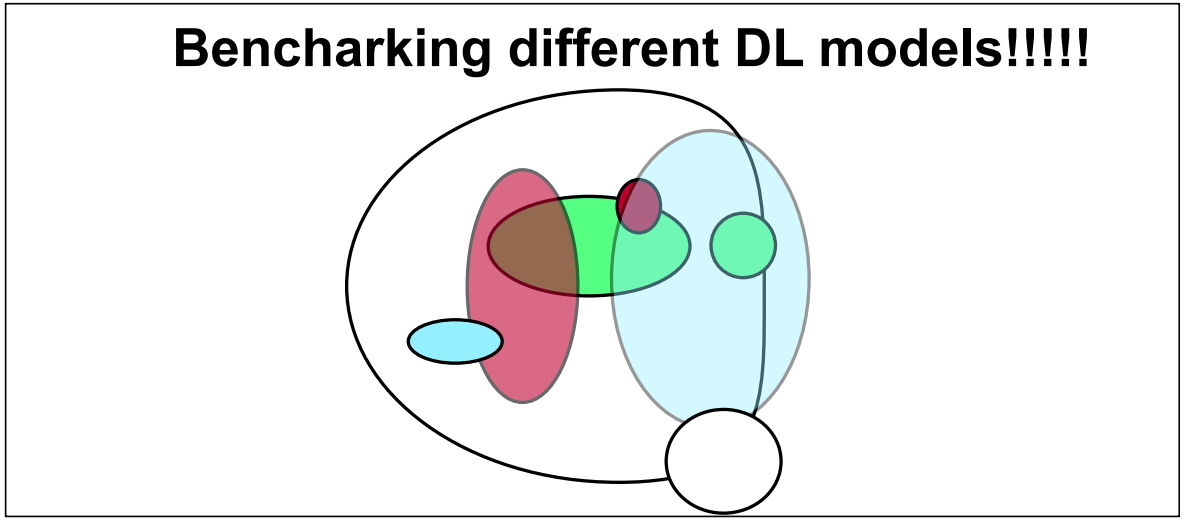
Train-set/Test-set NO leakage



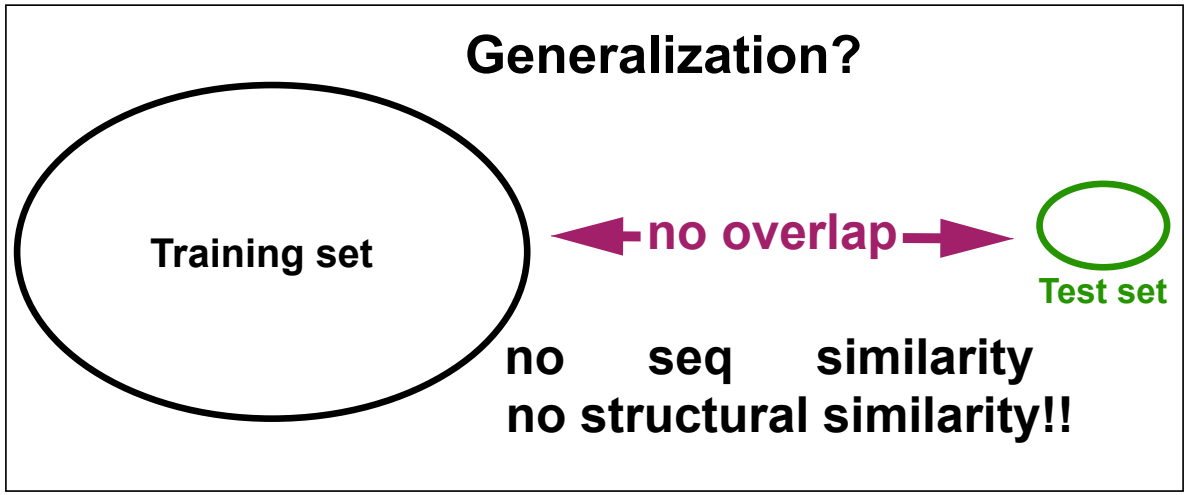
Qian-Sejnowski FIgure 8

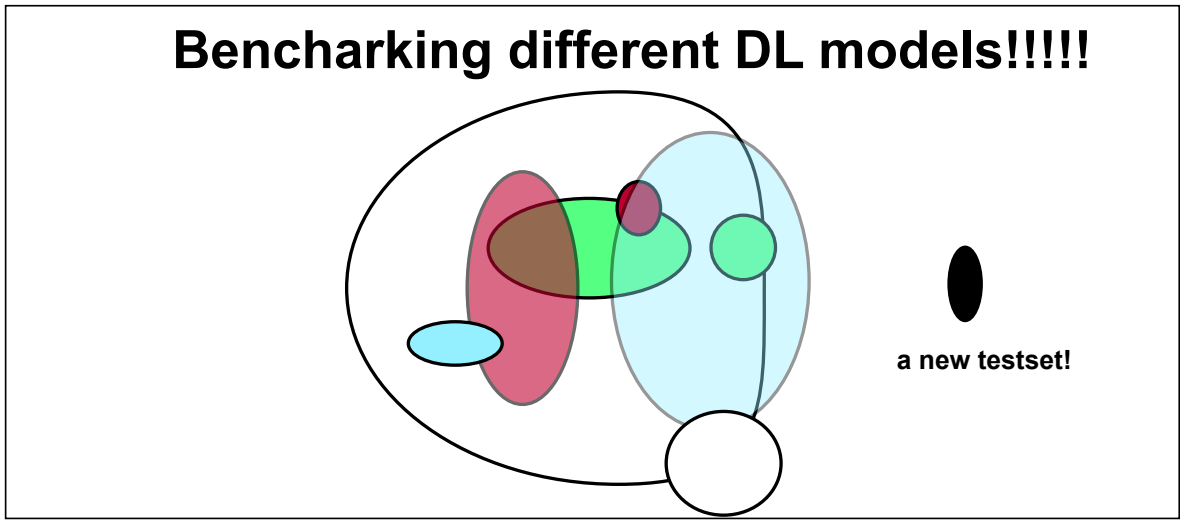
Generalization is not trivial



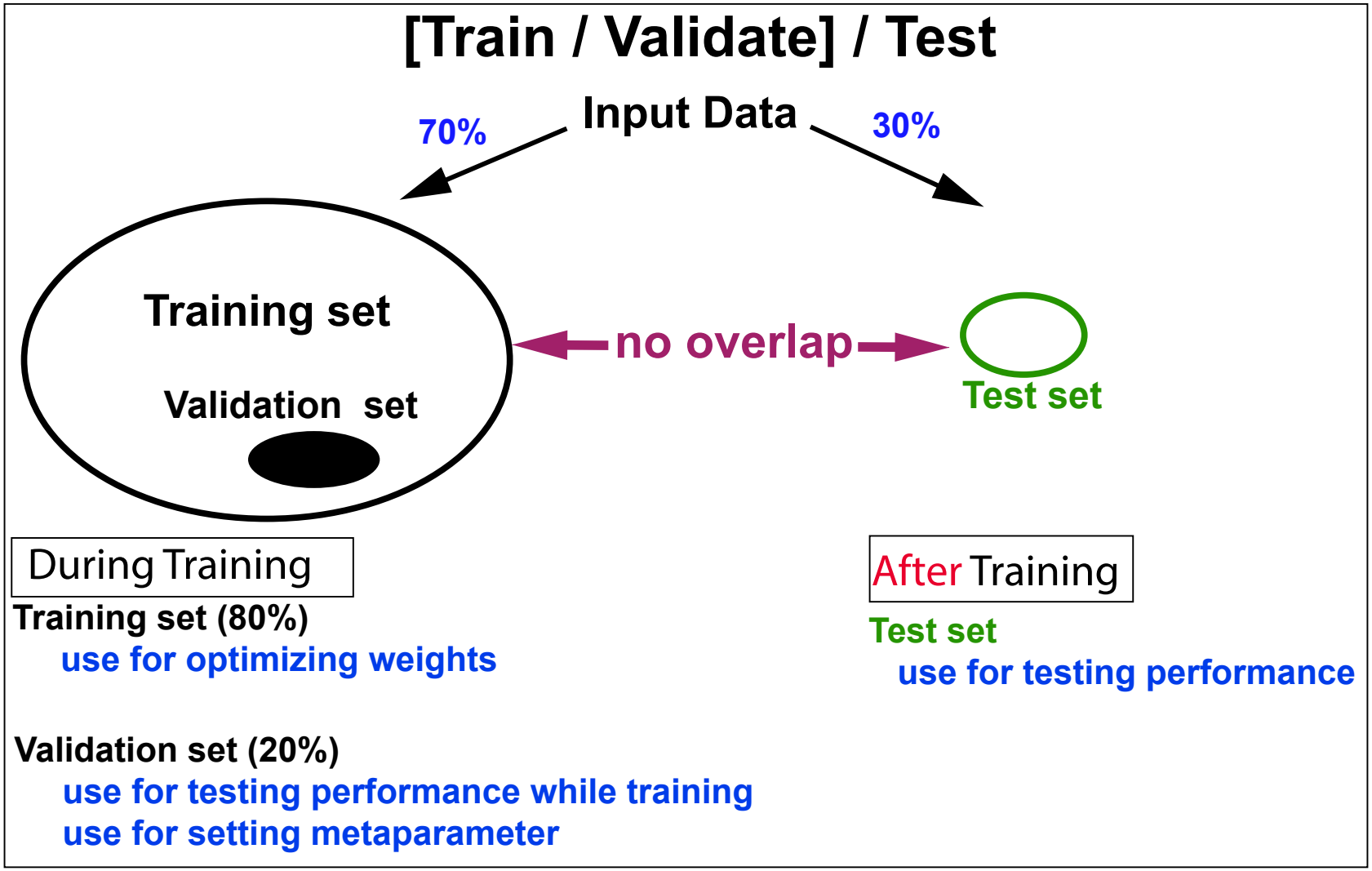


Generalization is not trivial



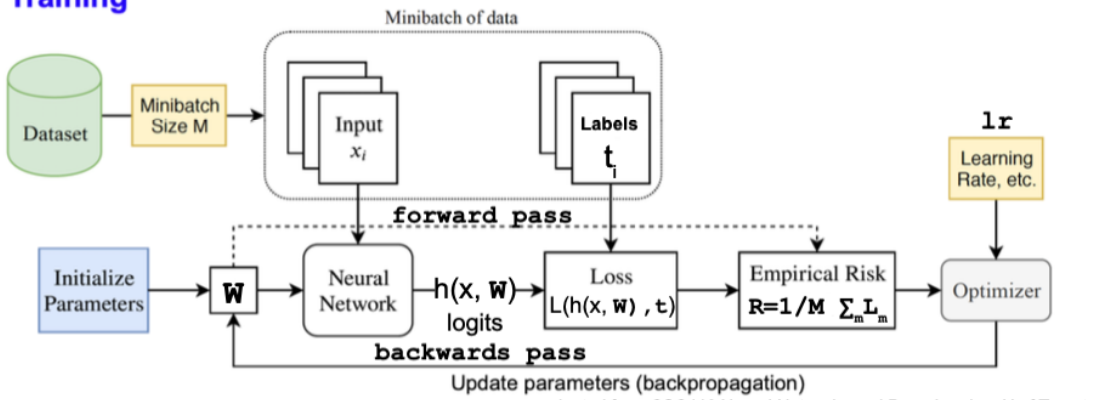


Training set / Validation set / Test set



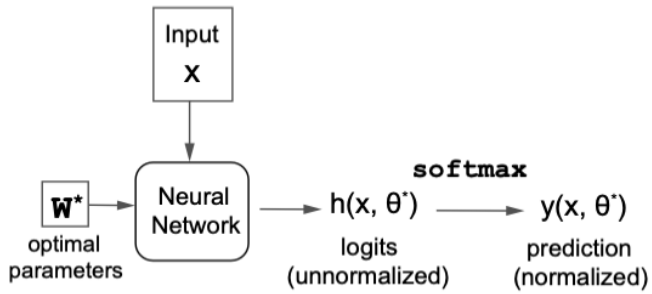
Basics of Neural Networks

Training

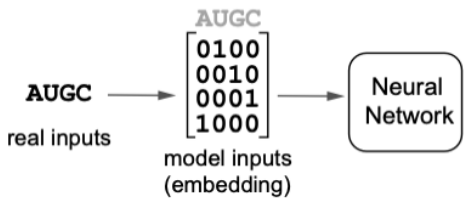


adapted from CSC413 Neural Networks and Deep Learing, U of Toronto

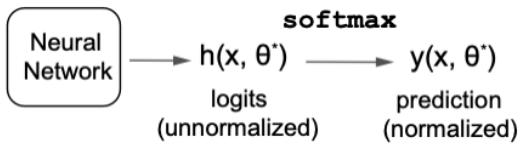
Inference



Inputs



Outputs



Optimization

Gradient descent

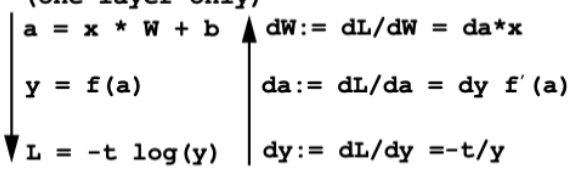
```
W <- W - lr * dR/dW
```

Stochastic Gradient descent

```
W <- W - lr * dLm/dW
```

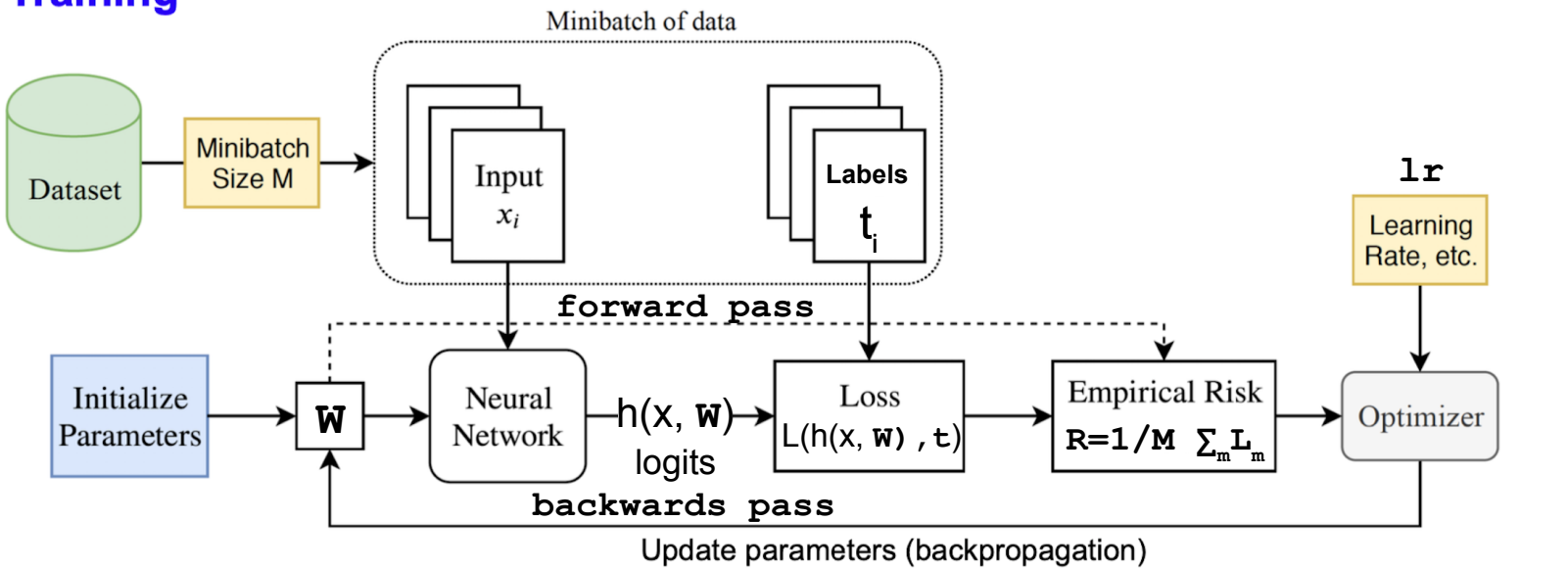
backpropagation/ automatic differentiation

(one layer only)



Basics of Neural Networks

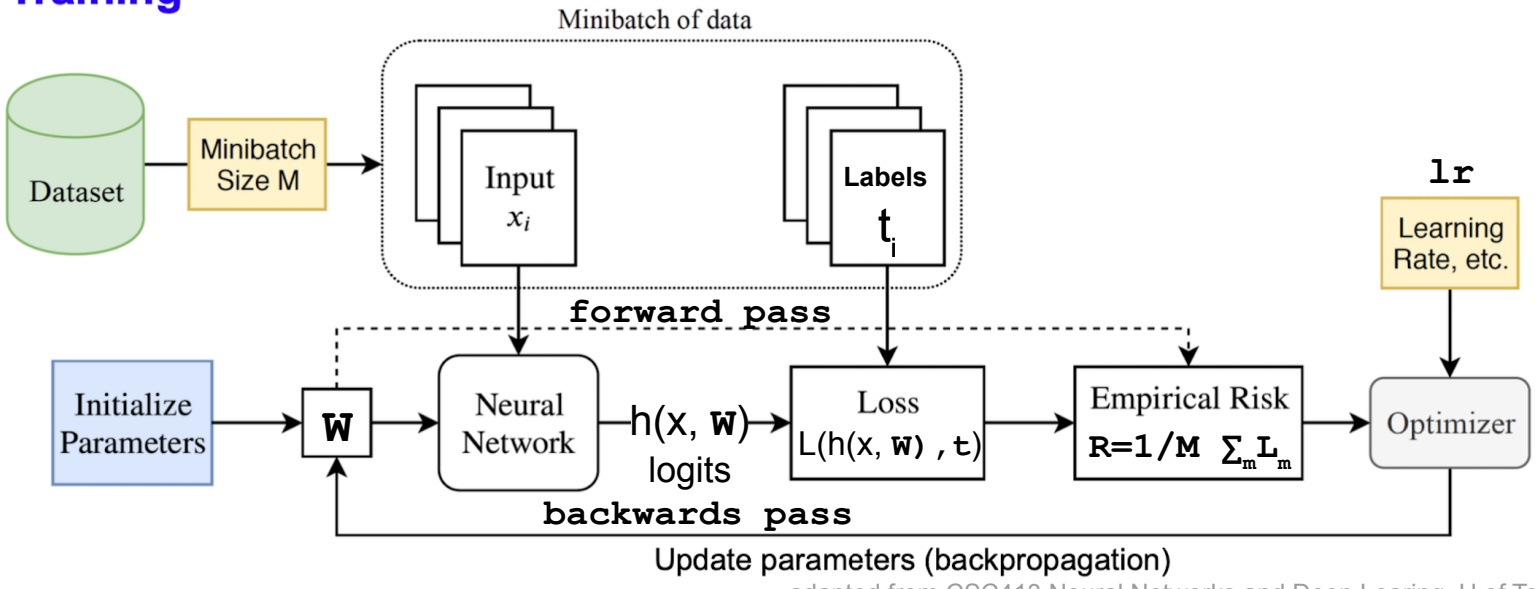
Training



adapted from CSC413 Neural Networks and Deep Learing, U of Toronto

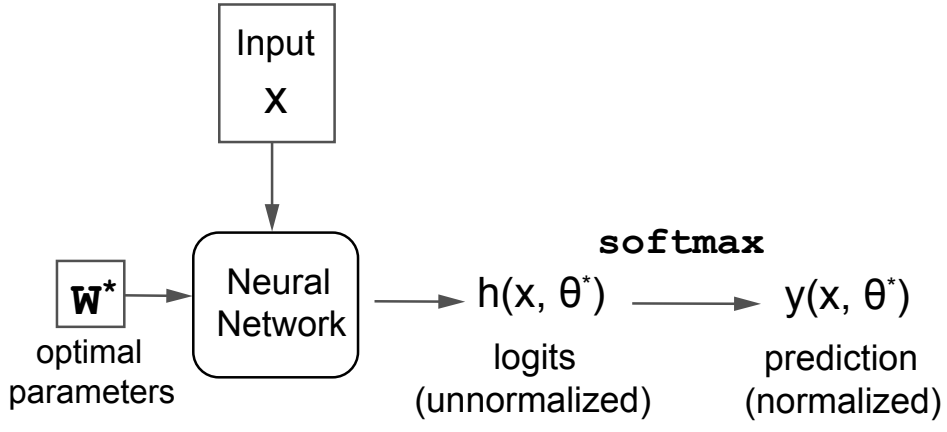
Basics of Neural Networks

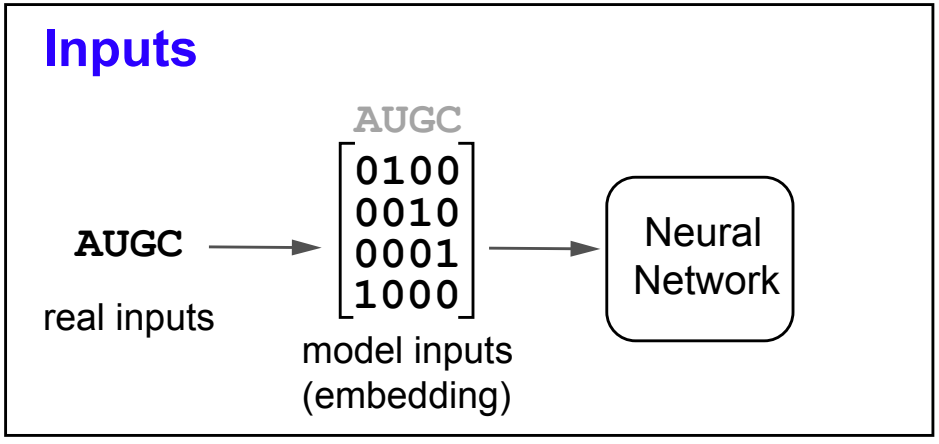
Training

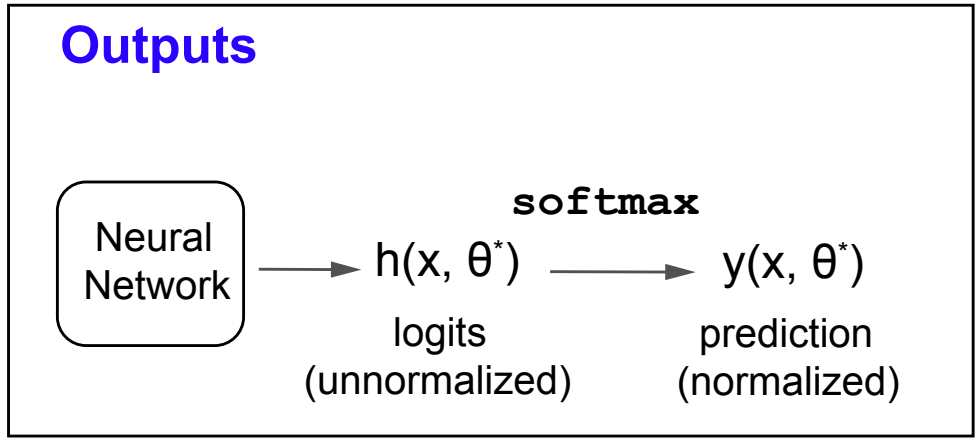


adapted from CSC413 Neural Networks and Deep Learing, U of Toronto

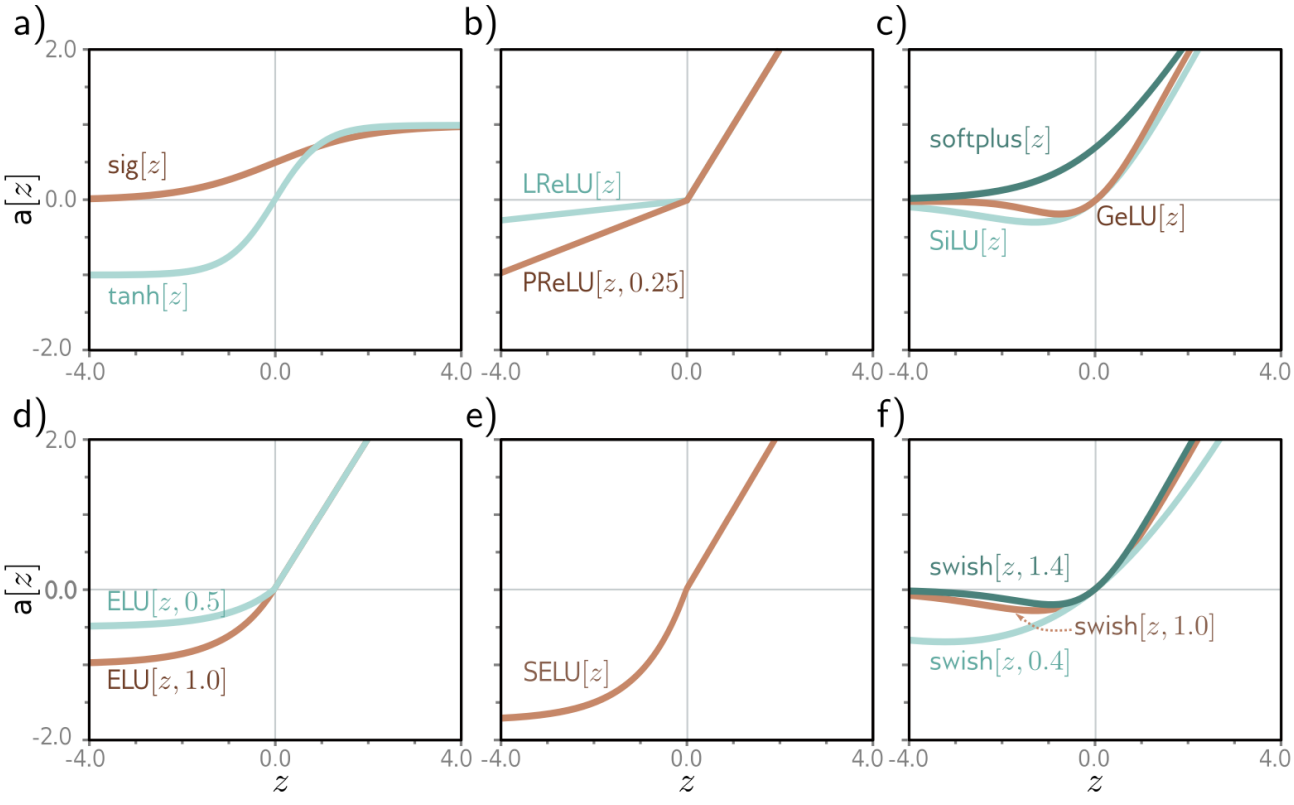
Inference




Inputs
AUGC
AUGC
0100
0010
0001
1000
Neural Network
real inputs
model inputs
(embedding)


Outputs
Neural Network
softmax
h(x, θ*)
y(x, θ*)
logits
(unnormalized)
prediction
(normalized)

Activation functions introduce non-linearity



Fitting the models

- Gradient descent
- Stochastic gradient descent
- Batch and epoch
- Adam (adaptive momentum estimation)

Optimization

Gradient descent

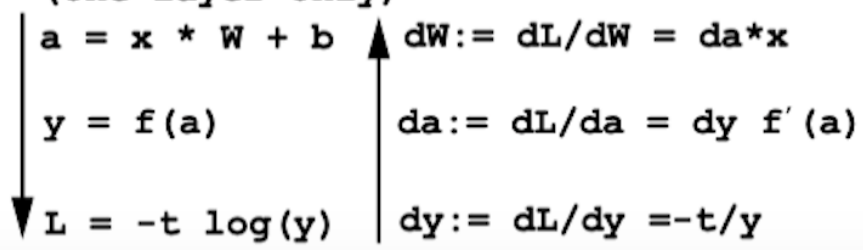
$$W \leftarrow W - lr * dR/dW$$

Stochastic Gradient descent

$$W \leftarrow W - lr * dL_m/dW$$

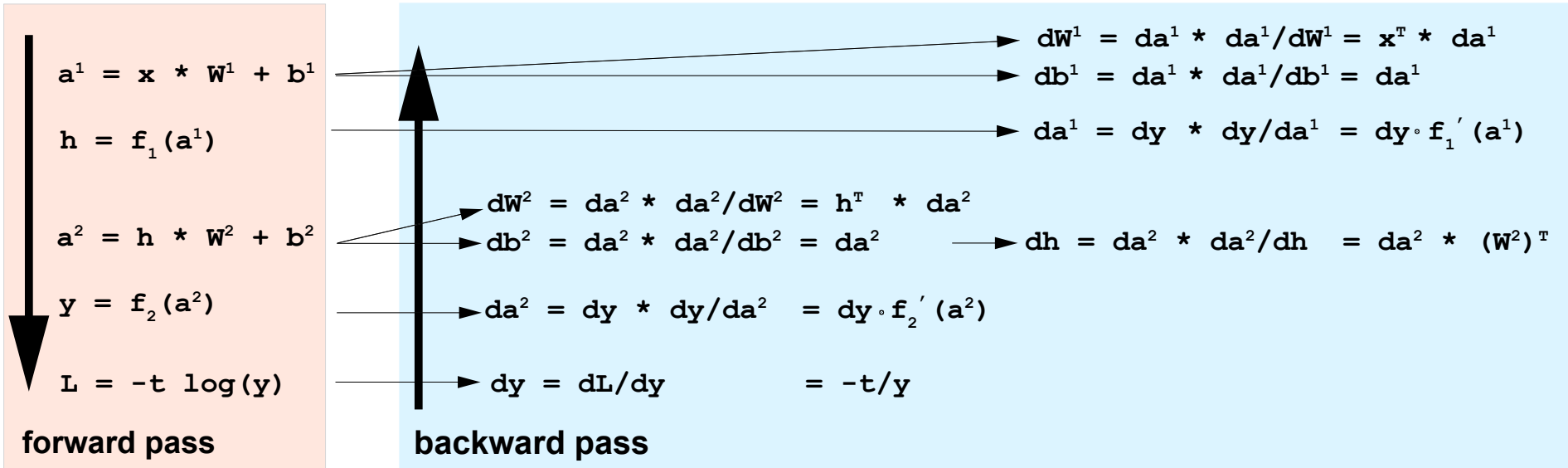
backpropagation/ automatic differentiation

(one layer only)



backpropagation/automatic differentiation

MLP with 1 hidden layer



notation:
dv := dL/dv, 1 <= n <= N (batch dim)
1 <= i <= I (input dim)
1 <= j <= H (hidden dim)
1 <= k <= O (output dim)

$x^T[I,N] * da^1[N,H] = \sum_n x(n,i) da^1(n,j) = dW^1[I,H]$
$h^T[H,N] * da^2[N,O] = \sum_n h(n,j) da^2(n,k) = dW^2[H,O]$
$da^2[N,O] * (W^2)^T[O,H] = \sum_k da^2(n,k) W^2(j,k) = dh[N,H]$

Minibatch and Epoch

$$\theta_t \leftarrow \theta_{t-1} - \alpha \sum_{m \in batch} \frac{\delta L_m}{\delta \theta_{t-1}}$$

$$L_m = L(\theta_{t-1}, y^{(m)}, t^{(m)})$$

Minibatch and Epoch

DataLoader

```python
# Epoch/baches with DataLoader
import torch
from torch.utils.data import TensorDataset, DataLoader

# Toy data: 300 samples, 20 features, 3-class labels
N = 300 # number of examples
K = 3   # number of classes
X = torch.randn(N, 20)
y = torch.randint(0, K, (N,))

# Create a Dataset
dataset = TensorDataset(X, y)

# Create a DataLoader that yields minibatches
batch_size = 38
train_loader = DataLoader(
    dataset,
    batch_size=batch_size,
    shuffle=True,      # shuffle each epoch
    drop_last=False    # keep last smaller batch
)

# Iterate over minibatches
n_samples = 0
for batch_idx, (inputs, targets) in enumerate(train_loader):
    print(f"Batch {batch_idx}: inputs.shape = {inputs.shape}, targets.shape = {targets.shape}")
    n_samples += inputs.shape[0]
    # here you'd do: outputs = model(inputs), etc.
print(f"{n_samples} samples per epoch")
```

```
Batch 0: inputs.shape = torch.Size([38, 20]), targets.shape = torch.Size([38])
Batch 1: inputs.shape = torch.Size([38, 20]), targets.shape = torch.Size([38])
Batch 2: inputs.shape = torch.Size([38, 20]), targets.shape = torch.Size([38])
Batch 3: inputs.shape = torch.Size([38, 20]), targets.shape = torch.Size([38])
Batch 4: inputs.shape = torch.Size([38, 20]), targets.shape = torch.Size([38])
Batch 5: inputs.shape = torch.Size([38, 20]), targets.shape = torch.Size([38])
Batch 6: inputs.shape = torch.Size([38, 20]), targets.shape = torch.Size([38])
Batch 7: inputs.shape = torch.Size([34, 20]), targets.shape = torch.Size([34])
300 samples per epoch
```

Adam (Adaptive momentum estimation)

$$\theta_{t-1} \rightarrow \theta_t$$

- Get gradients at time t

  $g_t \leftarrow \frac{\delta L}{\delta \theta_{t-1}}$
- Update first momentum estimate

  $m_t \leftarrow \beta_1 \, m_{t-1} + (1 - \beta_1) \, g_t$
- Update second momentum estimate

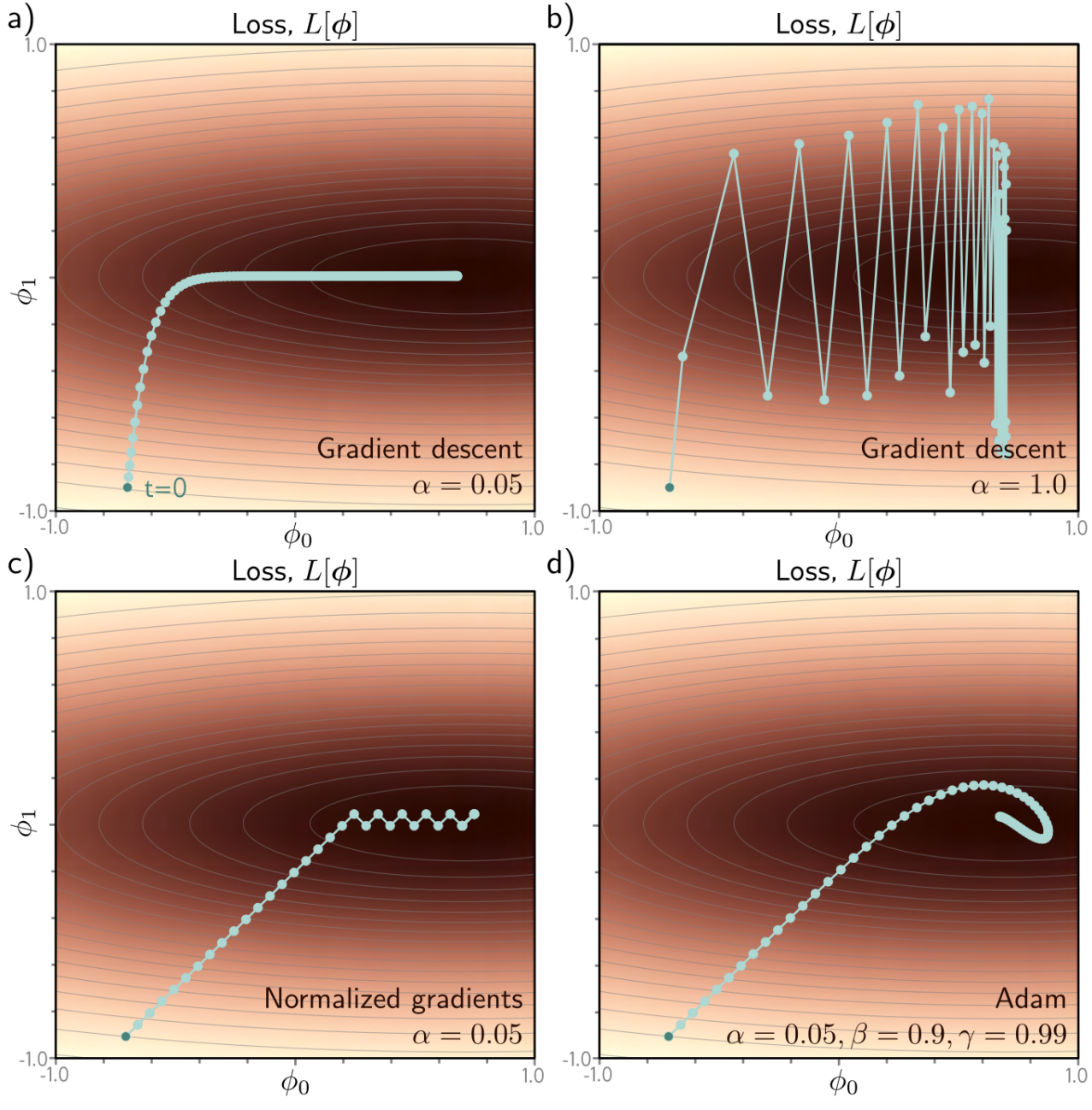
  $v_t \leftarrow \beta_2 \, v_{t-1} + (1 - \beta_2) \, (g_t)^2$
- Bias-correct first momentum

  $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
- Bias-corrected second momentum

  $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
- Update parameters

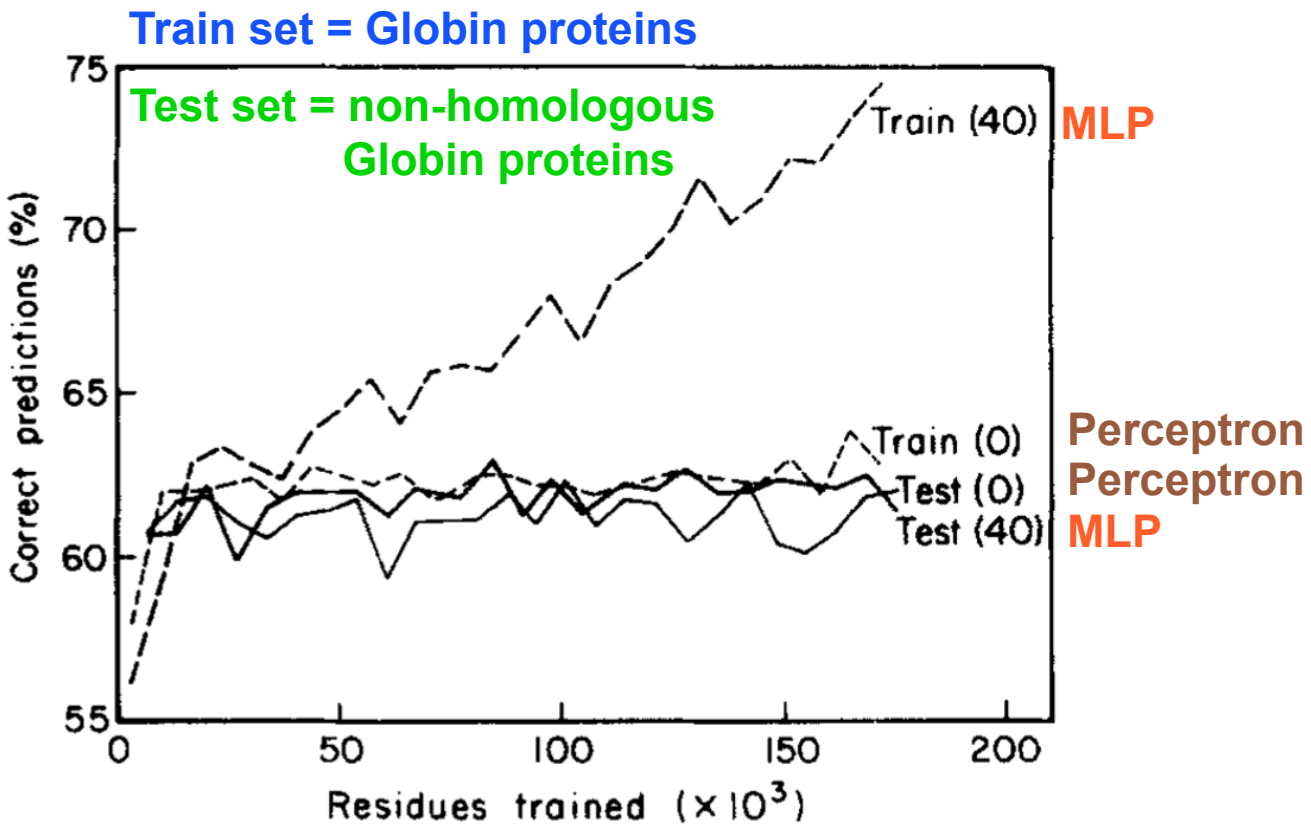
  $\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon} \qquad \alpha > 0 \, \beta_1, \beta_2 \in [0, 1)$

Fitting the model



Generalization

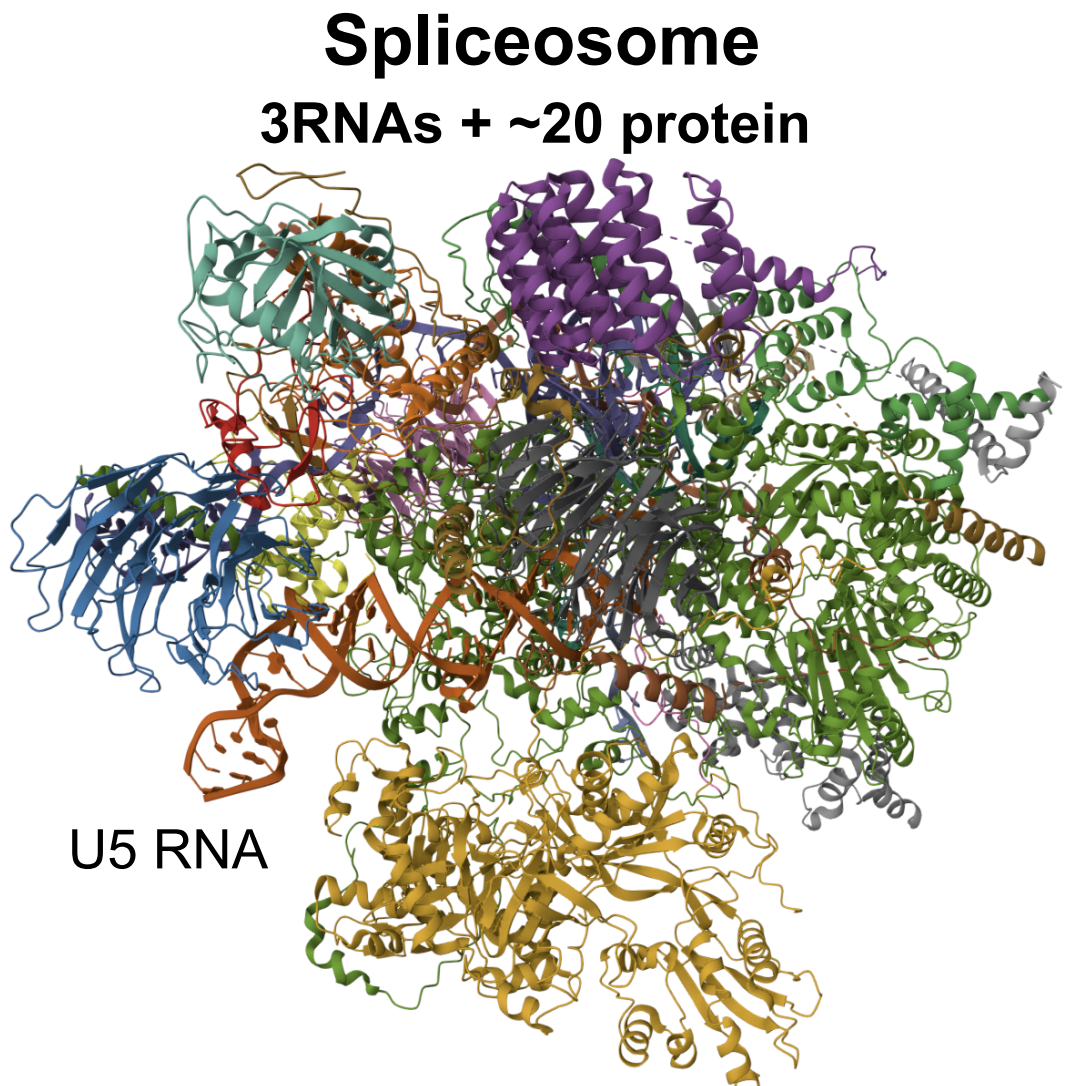
Performance on train set $\longrightarrow$ Performance on test set?



Qian-Sejnowski FIgure 8

no Generalization $\longrightarrow$

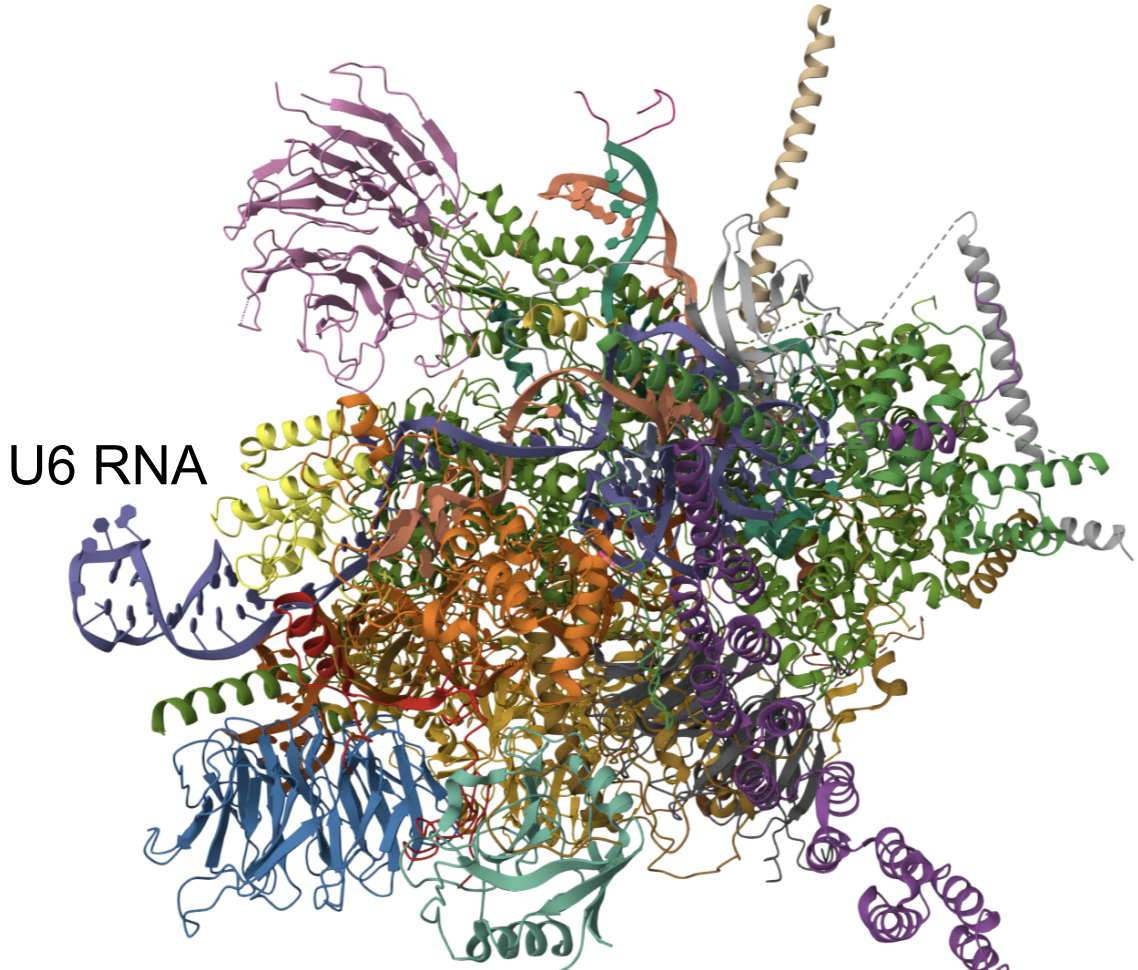
Memorization $=$ overfitting



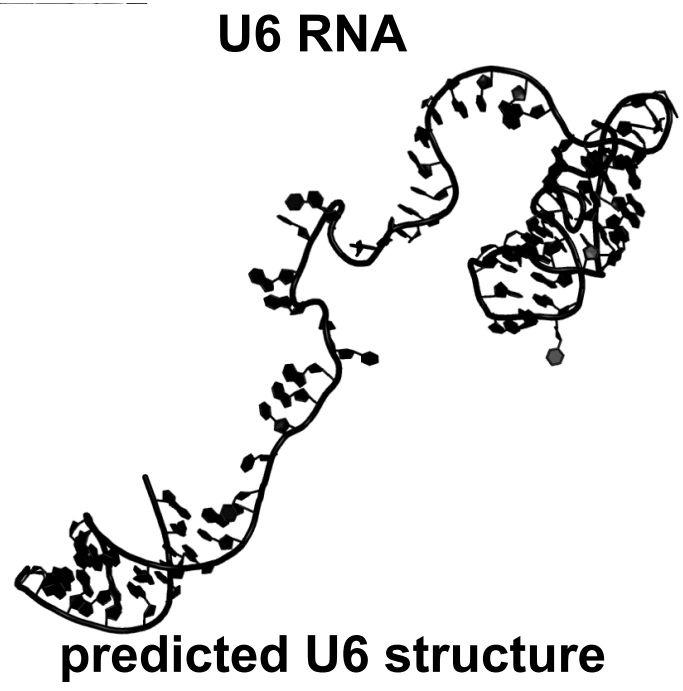
no Generalization $\longrightarrow$
Memorization $=$ overfitting

Spliceosome

3RNAs + ~20 protein

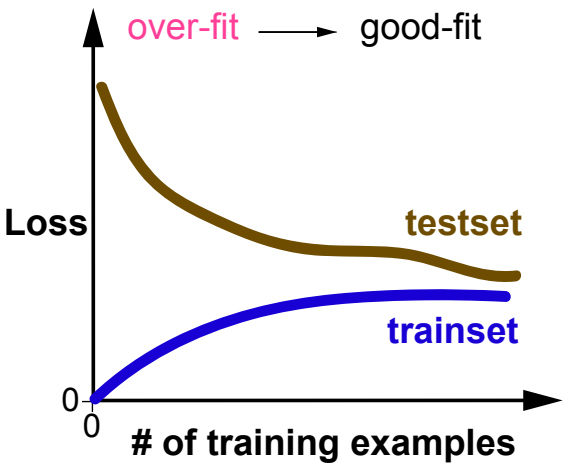


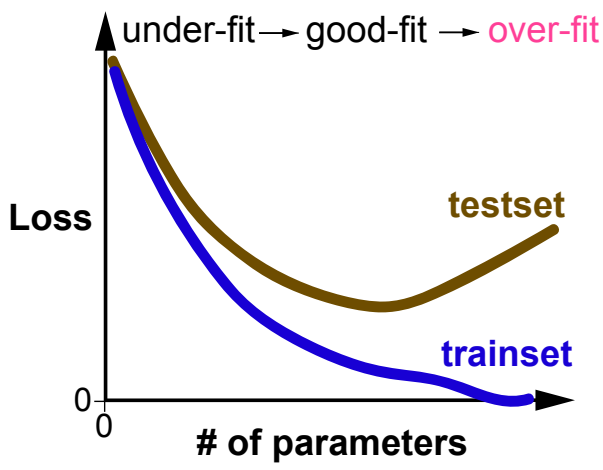
no Generalization $\longrightarrow$ Memorization $=$ overfitting

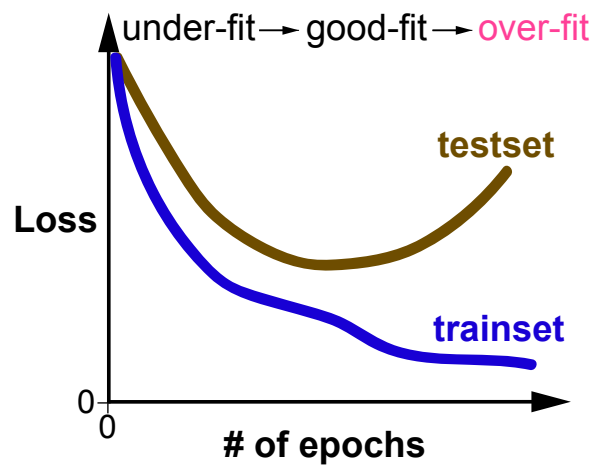


Generalization

Performance on train set $\longrightarrow$ Performance on test set?

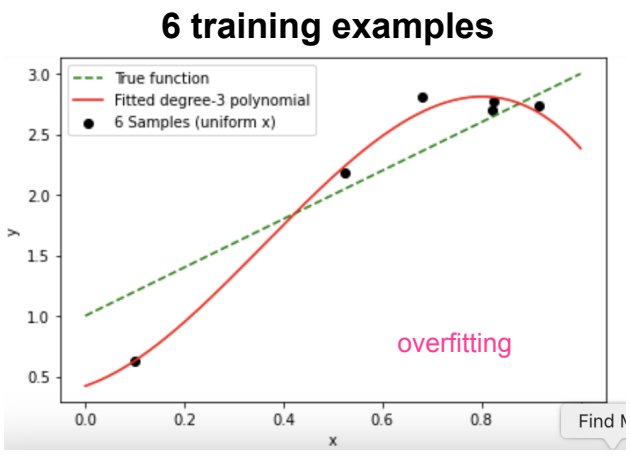


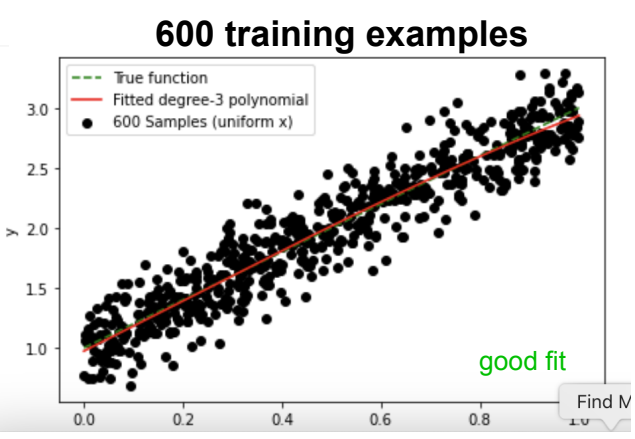




A bag of (useful) tricks!

Effect of # training examples





Larger training sets improve generalization

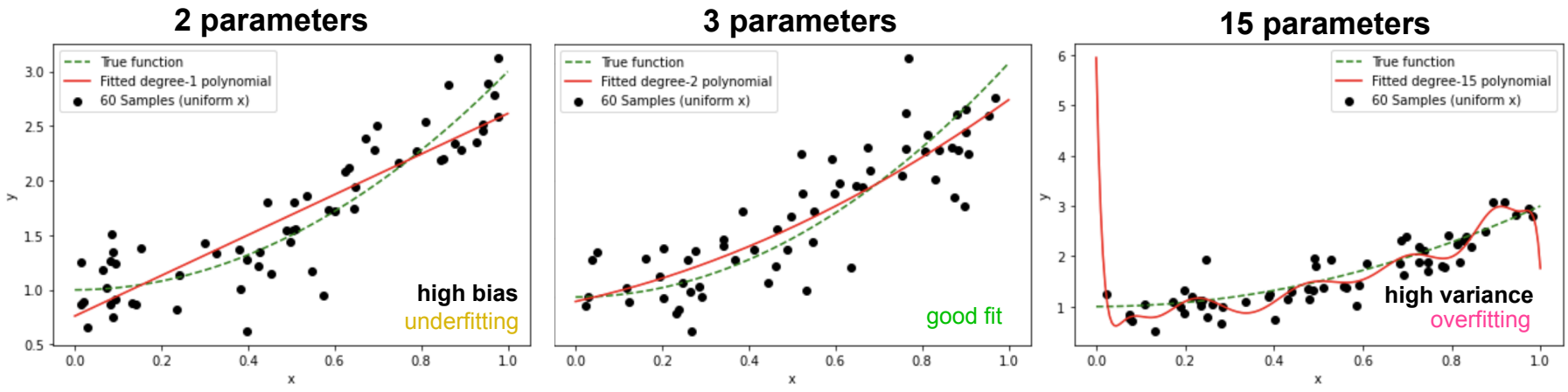
Generalization

- Adding more training data
  - Increase the size of the training set only
  - Be smart adding the data (find more distant homologs, or with not yet explored base compositions,...)

Generalization

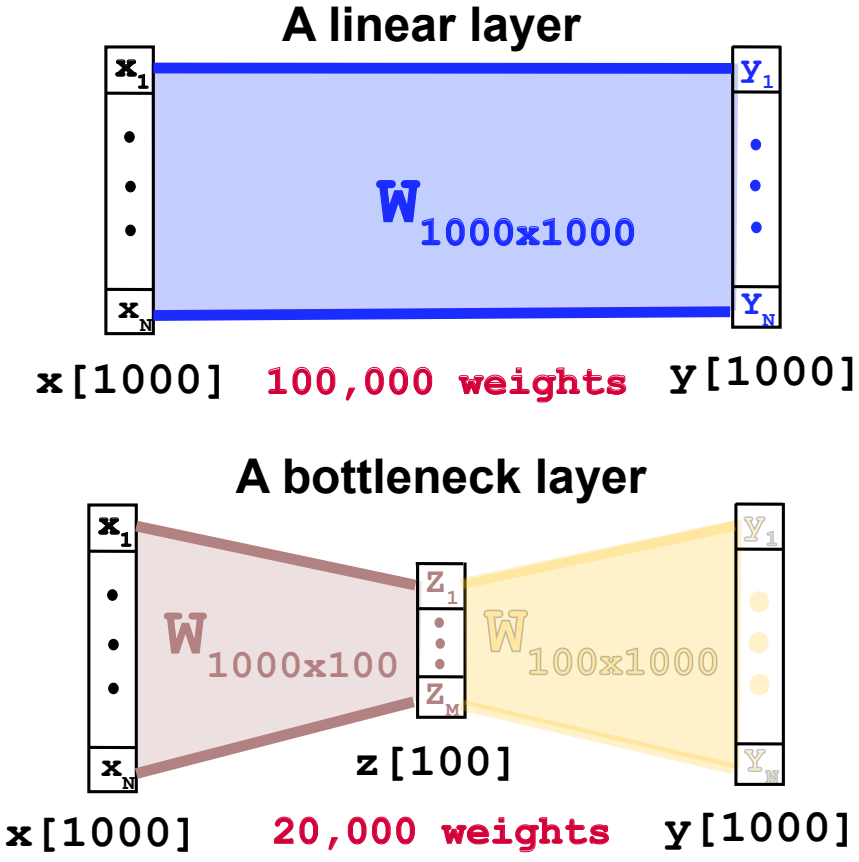
- Adding more training data
  - Increase the size of the training set only
  - Be smart adding the data (find more distant homologs, or with not yet explored base compositions,...)
  - do to train on "too many" artifacts

Effect of # of parameters



More parameters increase the chance of overfitting to the training data

Reducing the number of parameters with a bottleneck layer



forces to learn a compact representation (Autoencoders coming soon...)

Trade offs: less expressive power but also less overfitting

Generalization

- Adding more data
- Reduction of parameters adding a bottleneck layer

Balance data / parameters

- **Little data + Many params** $\rightarrow$ overfitting
- **Lots of data + Few params** $\rightarrow$ underfitting
- **Lots of data + Many params** $\rightarrow$ balance
- **lots of data** $<$ **Many params** $\rightarrow$ overparameterization

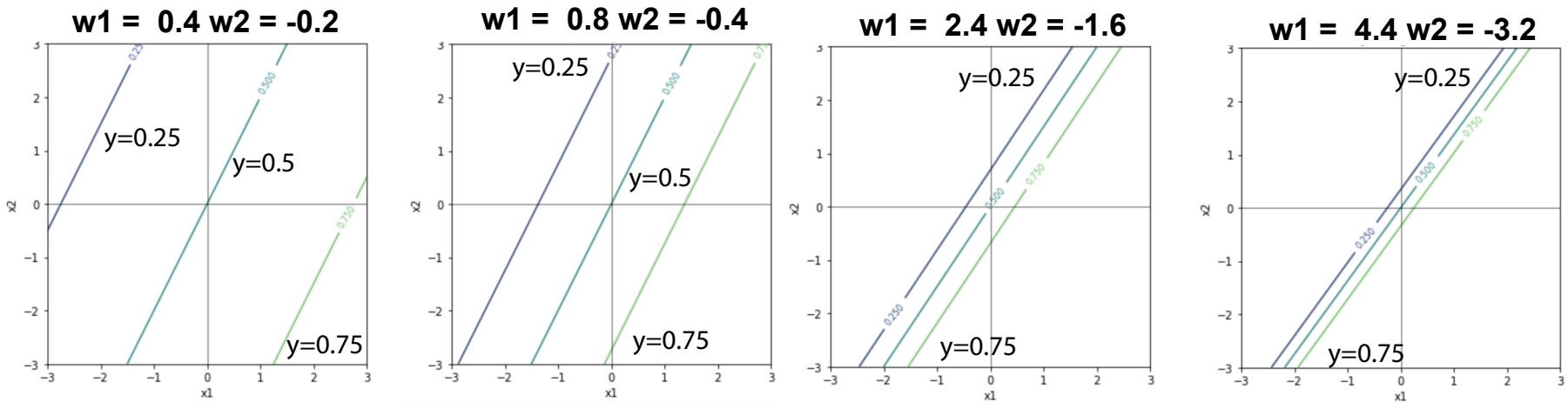
Generalization

- Adding more data
- Reduction of parameters adding a bottleneck layer
- Balance data/parameters

Large weights are a problem

- They induce overfitting to small details in data
- They amplifies small changes in the data
- The decision boundary can flip due to small non relevant variations
- They produce narrow margins. Generalization requires larger margins.

y =logistic(w1*x1+w2*x2)



Generalization

- Adding more data
- Reduction of parameters adding a bottleneck layer
- Balance data/parameters
- Large weights induce overfitting

Explicit regularization

$$\theta^* = \text{argmin}_\theta \left[L(y, x, \theta) + \eta R(\theta) \right]$$

The regularization function $R(\theta)$ returns an scalar

$R(\theta)$ is larger for parameters that are less preferred

L^2 regularization $R(\theta) = \frac{1}{2}\theta^2$

L^1 regularization $R(\theta) = ||\theta||$

L_2 **vs** L_1

$$\frac{\delta R_{L2}}{\delta \theta} = \eta\theta$$
$$\frac{\delta R_{L1}}{\delta \theta} = \eta sign(\theta)$$

L2: $\theta_t \longleftarrow \theta_{t-1} - \alpha\eta\theta_{t-1}$ **pull proportional to weights**

L1: $\theta_t \longleftarrow \theta_{t-1} - \alpha\eta$ **constant pull**

Explicit regularization

Probabilistic interpretation

$$P(\theta \mid x, y) = \frac{P(y|x,\theta)P(\theta)}{P(y|x)}$$

$$\begin{aligned}\theta^* &= \text{argmax}_\theta P(\theta \mid x, y) \\ &= \text{argmax}_\theta \log P(\theta \mid x, y) \\ &= \text{argmax}_\theta \left[\log P(y \mid x, \theta) + \log P(\theta)\right]\end{aligned}$$

Compare to

$$\theta^* = \text{argmin}_\theta \left[L(y, x, \theta) + \eta R(\theta)\right]$$

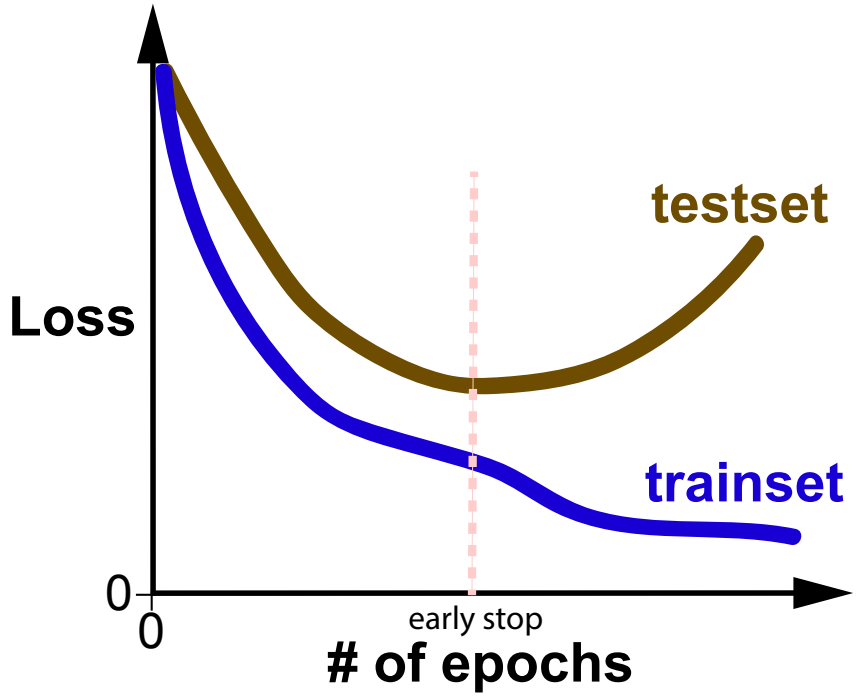
thus

$$\eta R(\theta) = -\log P(\theta)$$

Generalization

- Adding more data
- Reduction of parameters adding a bottleneck layer
- Balance data/parameters
- Large weights induce overfitting
- Regularization of large weights with the L^2 norm

Early stopping



Generalization

- Adding more data
- Reduction of parameters adding a bottleneck layer
- Balance data/parameters
- Large weights induce overfitting
- Regularization of large weights with the L^2 norm
- Early stopping

dropout

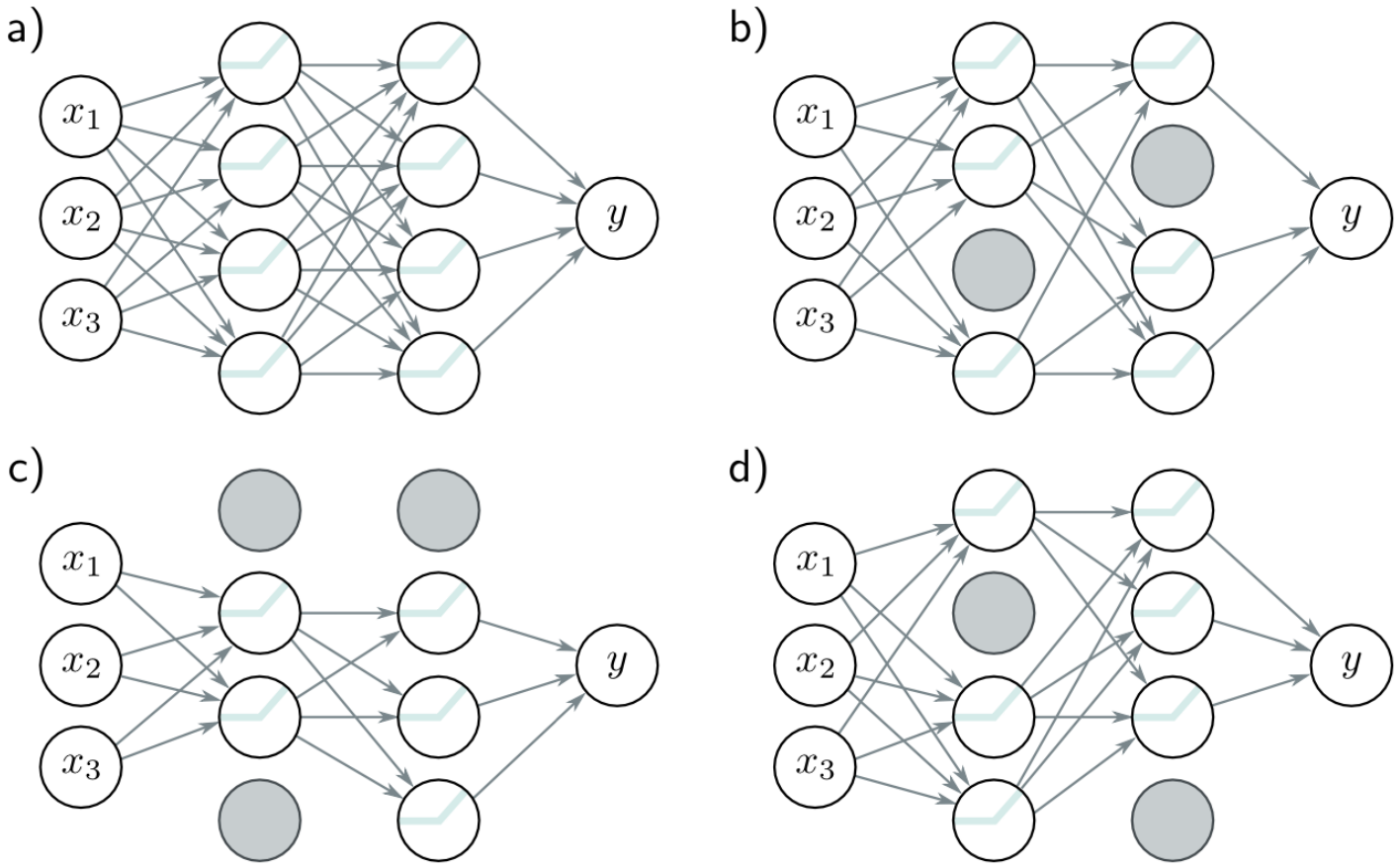


Figure 9.8 Dropout. a) Original network. b–d) At each training iteration, a random subset of hidden units is clamped to zero (gray nodes). The result is that the incoming and outgoing weights from these units have no effect, so we are training with a slightly different network each time.

Understanding Deep Learning, Prince

Generalization

- Adding more data
- Reduction of parameters adding a bottleneck layer
- Balance data/parameters
- Large weights induce overfitting
- Regularization of large weights with the L^2 norm
- Early stopping
- Stochastic regularizations
  - Dropout

Batch normalization

For a layer of dimension D, $(x_1, \ldots, x_d, \ldots, x_D)$

Mini-bach with m examples
inputs: $(x_1^{(i)}, \ldots, x_D^{(i)})$ for $i = 1, \ldots, m$

For each $d \in [1, D]$
mean and variance over batch
$\mu_d^B = \frac{1}{m} \sum_{i=1}^m x_d^{(i)} \quad and \quad (\sigma_d^B)^2 = \frac{1}{m} \sum_{i=1}^m (x_d^{(i)} - \mu_d^B)^2$

scale:
$\hat{x_d}^{(i)} = \frac{x_d^{(i)} - \mu_d^B}{\sqrt{(\sigma_d^B)^2 + \epsilon}}$

Batch Norm layer
outputs:
$y_d^{(i)} = \gamma \hat{x_d}^{(i)} + \beta$
γ and β are BatchNorm parameters, learned during the training loop.

Generalization

- Adding more data
- Reduction of parameters adding a bottleneck layer
- Balance data/parameters
- Large weights induce overfitting
- Regularization of large weights with the L^2 norm
- Early stopping
- Stochastic regularizations
  - Dropout
  - Batch normalization

Label smoothing

Replace

$$p(\text{true label}) = 1$$
$$p(\text{others}) = 0$$

with

$$p(\text{true label}) = 1 - \rho$$
$$p(\text{others}) = \rho/(K-1)$$

for some value $0 < \rho < 1$.

Generalization

- Adding more data
- Reduction of parameters adding a bottleneck layer
- Balance data/parameters
- Large weights induce overfitting
- Regularization of large weights with the L^2 norm
- Early stopping
- Stochastic regularizations
  - Dropout
  - Applying noise (to input data or weights)
  - Batch normalization
  - Applying noise (to input data or weights)
  - Label smoothing