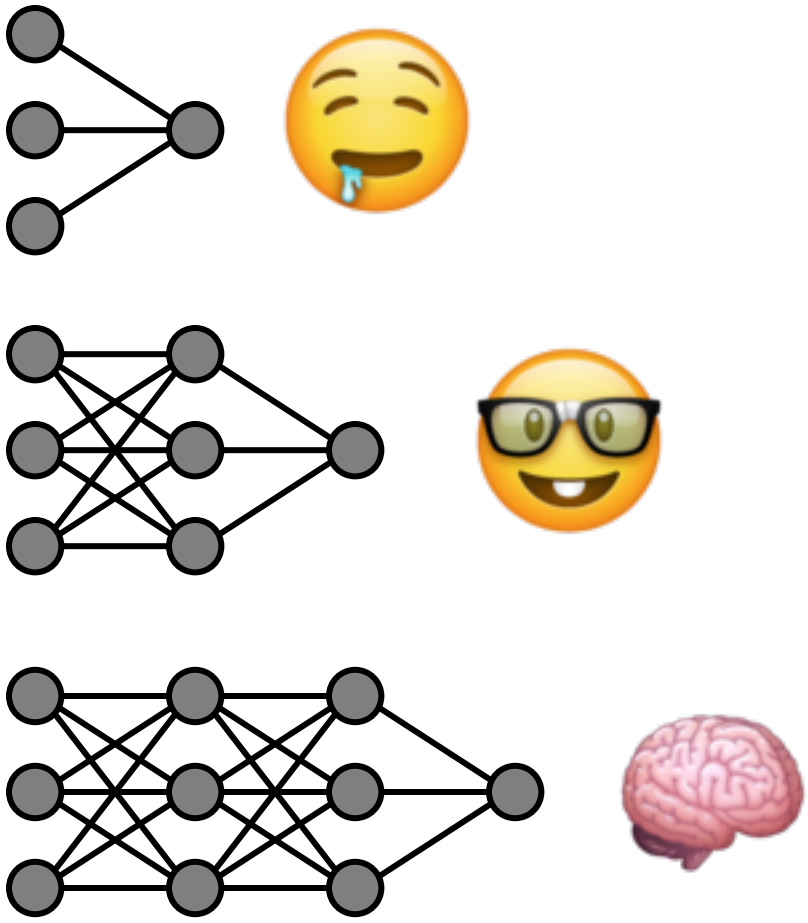
COMP 432 Machine Learning

Neural Networks

Computer Science & Software Engineering
Concordia University, Fall 2021

[Deep] Neural Networks



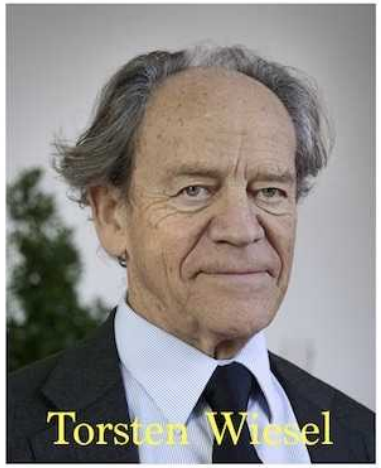
RECEPTIVE FIELDS, BINOCULAR INTERACTION AND FUNCTIONAL ARCHITECTURE IN THE CAT'S VISUAL CORTEX

By D. H. HUBEL and T. N. WIESEL

From the Neurophysiology Laboratory, Department of Pharmacology Harvard Medical School, Boston, Massachusetts, U.S.A.

(*Received* 31 *July* 1961)

What chiefly distinguishes cerebral cortex from other parts of the central nervous system is the great diversity of its cell types and interconnexions. It would be astonishing if such a structure did not profoundly modify the response patterns of fibres coming into it. In the cat's visual cortex, the receptive field arrangements of single cells suggest that there is indeed a degree of complexity far exceeding anything yet seen at lower levels in the visual system.



Received Nobel Prize in 1981 for their famous "Cat Experiment"

Demonstrated how complex cells recognize simple patterns falling on retina, such as oriented edges.

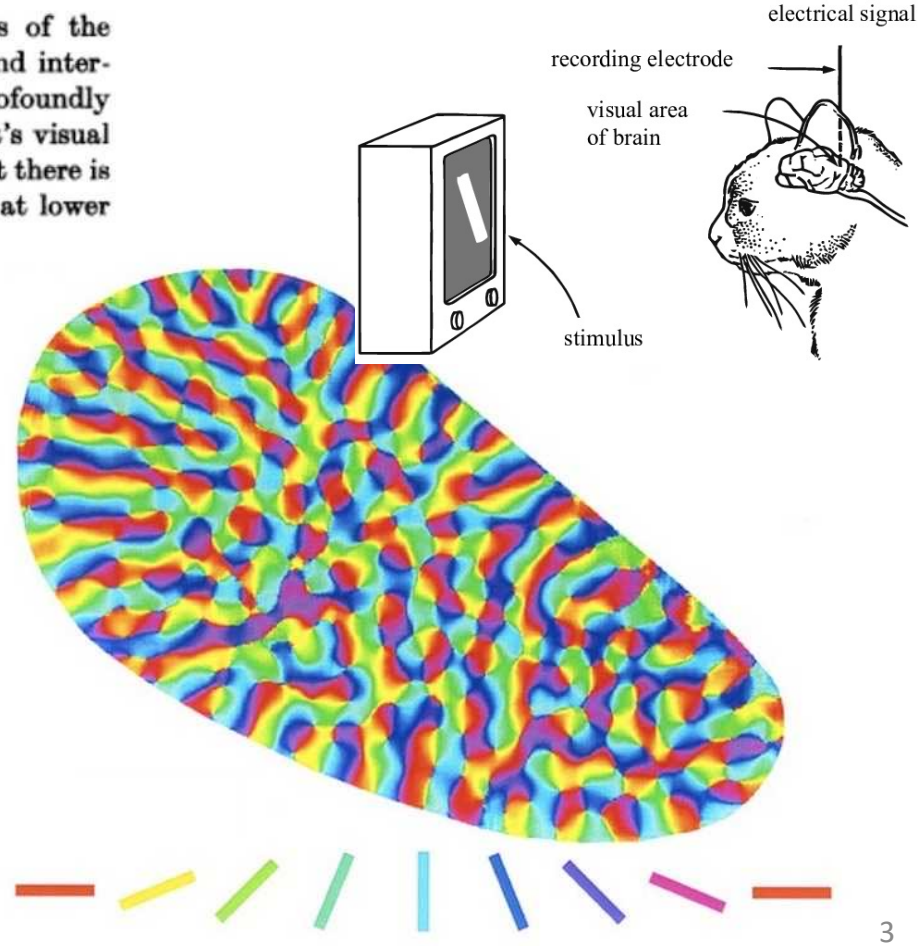
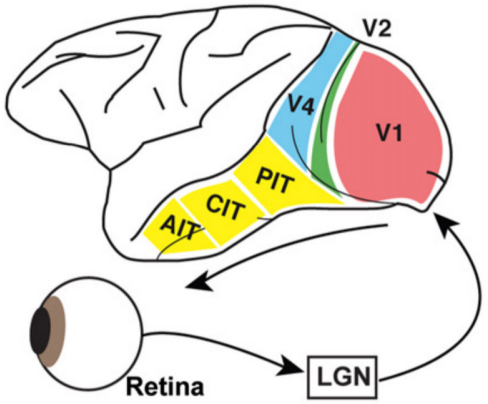
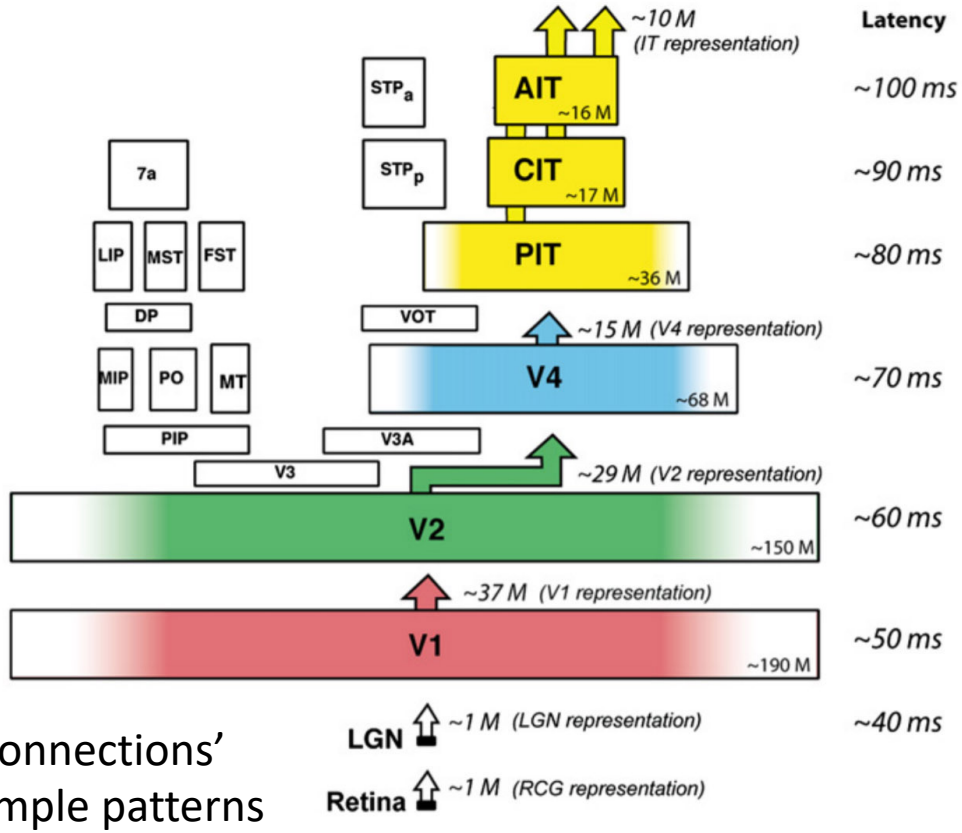


Image: Gabriel Peyré

Hubel and Wiesel Cat Experiment
https://www.youtube.com/watch?v=IOHayh06LJ4

Characteristics of visual pathway





- Organized into stages (layers) with local 'connections'
- Early stages (shallow layers) respond to simple patterns
- Later stages (deeper layers) respond to complex patterns
- Each stage is "wide" and does lots of parallel processing

Inspired what were called "connectionist" architectures

Mother nature gave us strong evidence that this style of 'architecture' is powerful at transforming stimuli!

Image: DiCarlo et al, *Neuron*, 2012

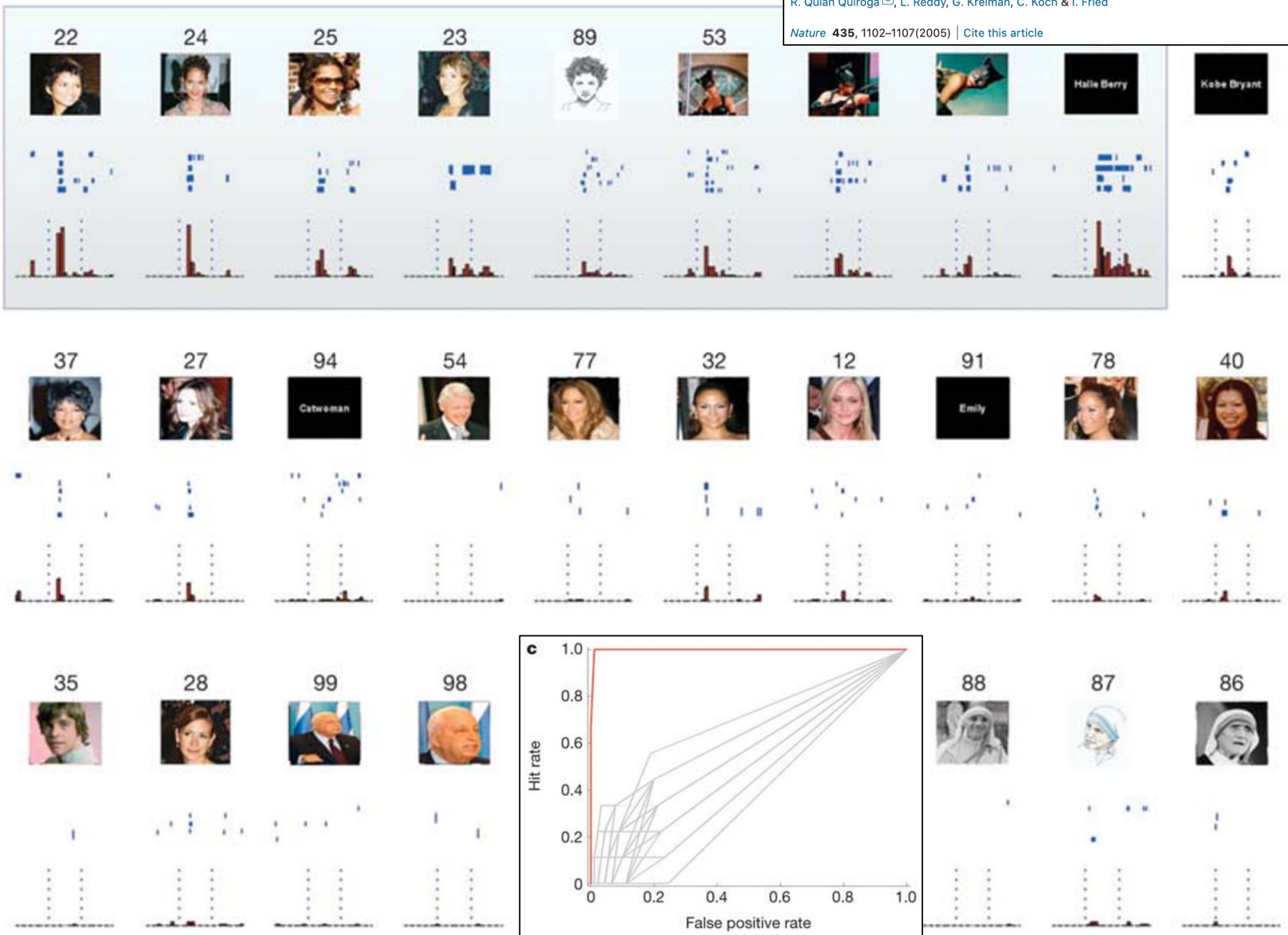
The “Halle Berry neuron” 🤯

Published: 23 June 2005

Invariant visual representation by single neurons in the human brain

R. Quian Quiroga ✉, L. Reddy, G. Kreiman, C. Koch & I. Fried

Nature **435**, 1102–1107(2005) | Cite this article



Neural nets were developed as a machine learning approach to A.I.

Gottfried Leibniz: "The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: Let us calculate! Without further ado, to see who is right."

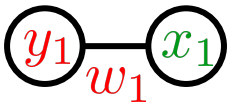
Geoffrey Hinton: "In the old days, the basic idea was that human reasoning is the core of intelligence, and so to understand human reasoning we'd better get something like logic into the computer. Then the computer would 'reason away' and maybe we could make it reason like people. But it's quite tricky to reason like people, mainly because **people don't do most of their thinking by reasoning.**

So the alternative view was that we should look at biology and we should try to make systems that work roughly like the brain, and the brain doesn't do most of its thinking by reasoning; it uses things like analogies. It's a great big neural network that has huge amounts of knowledge in the connections. It's got so much knowledge in it that **you couldn't possibly program it all in by hand.**"

Interview: https://www.youtube.com/watch?v=FTrQNmI3G0s

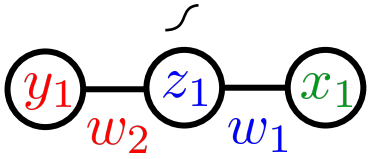
Some trivial "neural networks"

"1-1 network with linear output"



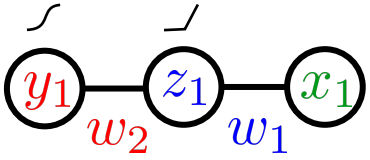
$$y_1 = w_1 x_1 + b_1$$

"1-1-1 sigmoid net with linear output"



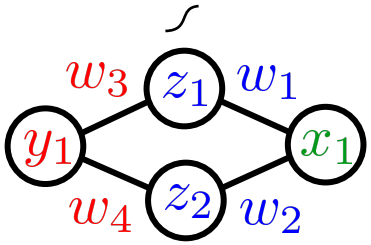
$$z_1 = \sigma(w_1 x_1 + b_1)$$
$$y_1 = w_2 z_1 + b_2$$

"1-1-1 ReLU network with logistic output"



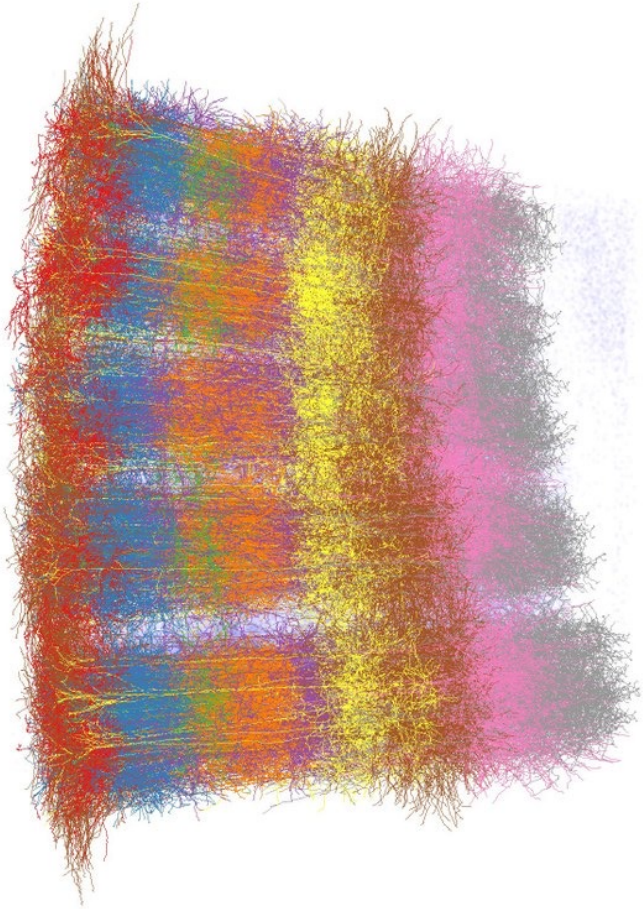
$$z_1 = \max(0, w_1 x_1 + b_1)$$
$$y_1 = \sigma(w_2 z_1 + b_2)$$

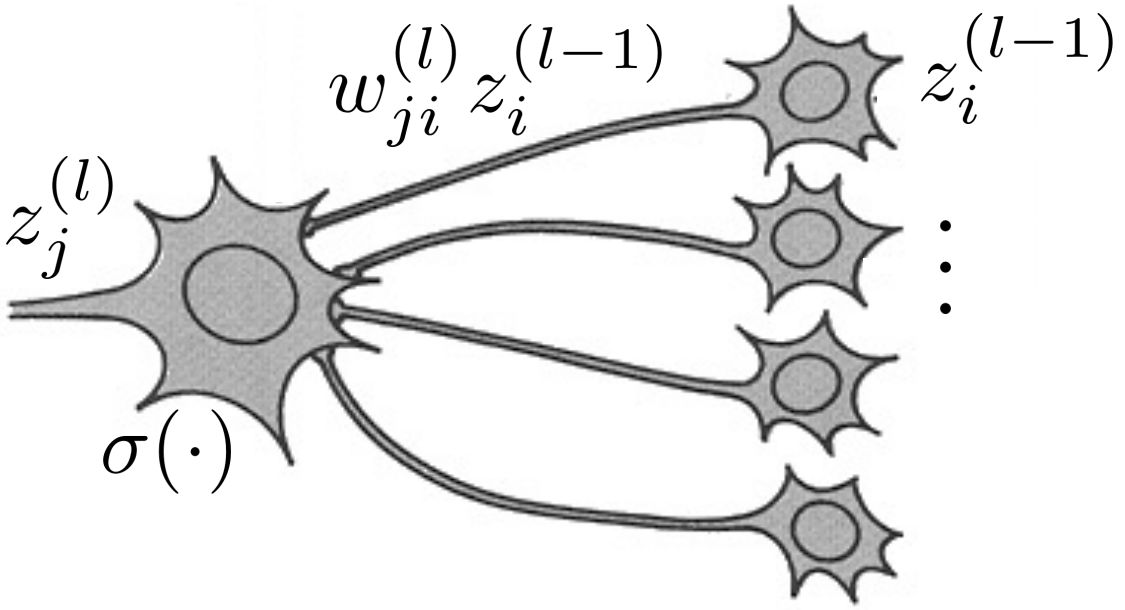
"1-2-1 sigmoid net with linear output"



$$z_1 = \sigma(w_1 x_1 + b_1)$$
$$z_2 = \sigma(w_2 x_1 + b_2)$$
$$y_1 = w_3 z_1 + w_4 z_2 + b_3$$

In what sense is each z_j a 'neuron'?





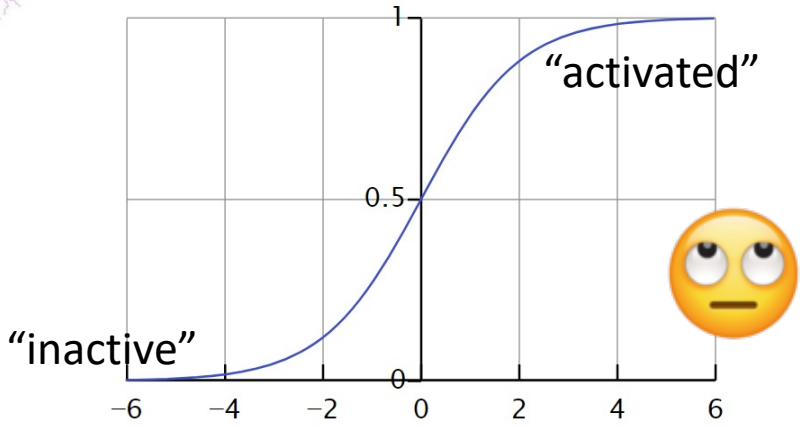
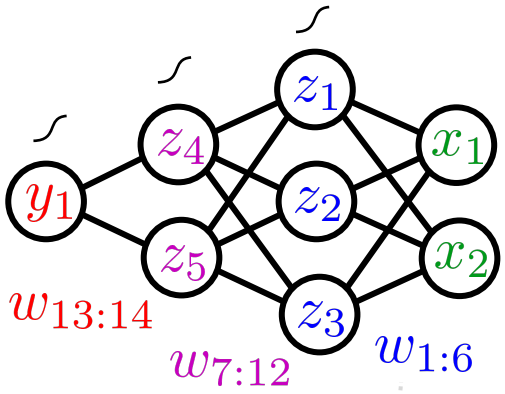


Image credit: Marcel Oberlaender
Image credit: Qef (talk), Public Domain

Anatomy of a neural net

"2-3-2-1 sigmoid net with logistic output"



"hidden layer activation *function*"

"input layer" x_1, x_2

"hidden layer activation"

"first layer weights"

"hidden layer 1"

$$z_1 = \sigma(w_1 x_1 + w_2 x_2 + b_1)$$
$$z_2 = \sigma(w_3 x_1 + w_4 x_2 + b_2)$$
$$z_3 = \sigma(w_5 x_1 + w_6 x_2 + b_3)$$

"hidden values" or "representation" or "latent codes"

"*weight* parameter"

"*bias* parameter"

"hidden layer 2"

$$z_4 = \sigma(w_7 z_1 + w_8 z_2 + w_9 z_3 + b_4)$$
$$z_5 = \sigma(w_{10} z_1 + w_{11} z_2 + w_{12} z_3 + b_5)$$

"output layer"

"prediction" or "output"

$$y_1 = \sigma(w_{13} z_4 + w_{14} z_5 + b_6)$$

"output activation function" (logistic/softmax for classification, linear for regression)

Vectorizing a neural network

$$z_1^{(1)} = \sigma(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2 + b_1^{(1)})$$
$$z_2^{(1)} = \sigma(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2 + b_2^{(1)})$$
$$z_3^{(1)} = \sigma(w_{3,1}^{(1)} x_1 + w_{3,2}^{(1)} x_2 + b_3^{(1)})$$

Step 1: introduce new indexing scheme for inputs, parameters, hidden values, and outputs.

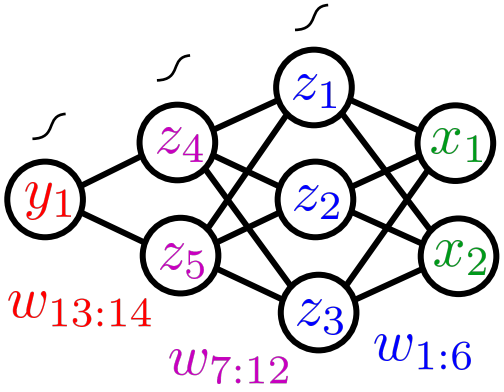
$$z_1^{(2)} = \sigma(w_{1,1}^{(2)} z_1^{(1)} + w_{1,2}^{(2)} z_2^{(1)} + w_{1,3}^{(2)} z_3^{(1)} + b_1^{(2)})$$
$$z_2^{(2)} = \sigma(w_{2,1}^{(2)} z_1^{(1)} + w_{2,2}^{(2)} z_2^{(1)} + w_{2,3}^{(2)} z_3^{(1)} + b_2^{(2)})$$

$$y_1 = \sigma(w_{1,1}^{(3)} z_1^{(2)} + w_{1,2}^{(3)} z_2^{(2)} + b_1^{(3)})$$

Step 2: organize the values into vectors and matrices

$$\boldsymbol{z}^{(1)} = \sigma(\boldsymbol{W}^{(1)} \boldsymbol{x} + \boldsymbol{b}^{(1)})$$
$$\boldsymbol{z}^{(2)} = \sigma(\boldsymbol{W}^{(2)} \boldsymbol{z}^{(1)} + \boldsymbol{b}^{(2)})$$
$$\boldsymbol{y} = \sigma(\boldsymbol{W}^{(3)} \boldsymbol{z}^{(2)} + \boldsymbol{b}^{(3)})$$

Python version of our net (not vectorized)



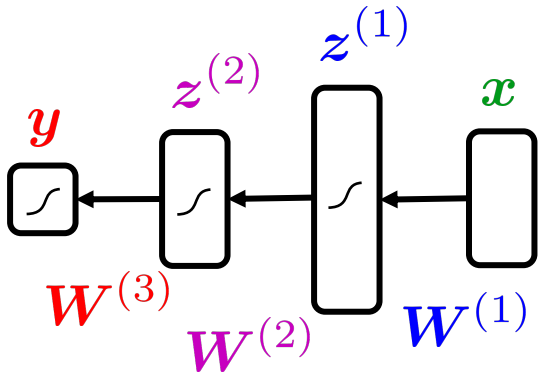

```
def sigmoid(a):
    return 1 / (1 + np.exp(-a))
```

```
def predict(x1, x2,
            w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14,
            b1, b2, b3, b4, b5, b6):

    z1 = sigmoid(w1*x1 + w2*x2 + b1)  # layer 1 hidden value 1
    z2 = sigmoid(w3*x1 + w4*x2 + b2)  # layer 1 hidden value 2
    z3 = sigmoid(w5*x1 + w6*x2 + b3)  # layer 1 hidden value 3

    z4 = sigmoid(w7*z1 + w8*z2 + w9*z3 + b4)     # layer 2 hidden value 1
    z5 = sigmoid(w10*z1 + w11*z2 + w12*z3 + b5)  # layer 2 hidden value 2

    y1 = sigmoid(w13*z4 + w14*z5 + b6)  # output layer value 1

    return y1
```

```
predict(1, 2,                                                    # inputs
        .1, .2, .3, .4, .5, .6, .7, .8, .9, .10, .11, .12, .13, .14,  # weights
        .01, .02, .03, .04, .05, .06)                            # bias
```

```
0.5628142981319841
```

Python version of our net (partially vectorized)




```
def predict(x,                  # shape (2,)
            W1, W2, W3,         # shape (3,2) (2,3) (1,2)
            b1, b2, b3):        # shape (3,) (2,) (1,)
    z1 = sigmoid(W1 @ x  + b1)  # shape (3,)
    z2 = sigmoid(W2 @ z1 + b2)  # shape (2,)
    y  = sigmoid(W3 @ z2 + b3)  # shape (1,)
    return y
```

The *architecture* did not change, we are just showing different "visual notation" of same network

```
x = np.array([1, 2])
W1 = np.array([[.1, .2], [.3, .4], [.5, .6]])   # shape (3,2)
W2 = np.array([[.7, .8, .9], [.10, .11, .12]])  # shape (2,3)
W3 = np.array([[.13, .14]])                     # shape (1,2)
b1 = np.array([.01, .02, .03])
b2 = np.array([.04, .05])
b3 = np.array([.06])

predict(x, W1, W2, W3, b1, b2, b3)
```

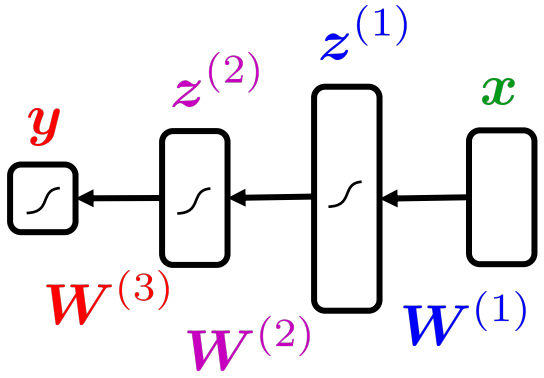
```
array([0.5628143])
```

Good, but still generates only one prediction at a time

```
X = np.random.rand(5, 2)
[predict(x, W1, W2, W3, b1, b2, b3) for x in X]
```

```
[array([0.56110293]),
 array([0.56092673]),
 array([0.560879]),
 array([0.56122544]),
 array([0.56107791])]
```

Python version of our net (fully vectorized)




```
def predict(X,                                  # shape (2,N)
            W1, W2, W3,                         # shape (3,2) (2,3) (1,2)
            b1, b2, b3):                        # shape (3,) (2,) (1,)
    Z1 = sigmoid(W1 @ X  + b1.reshape(-1, 1))   # shape (3,N)
    Z2 = sigmoid(W2 @ Z1 + b2.reshape(-1, 1))   # shape (2,N)
    Y  = sigmoid(W3 @ Z2 + b3.reshape(-1, 1))   # shape (1,N)
    return Y
```

"wrong" shape

annoying

```
X = np.random.rand(5, 2).T
predict(X, W1, W2, W3, b1, b2, b3)
```

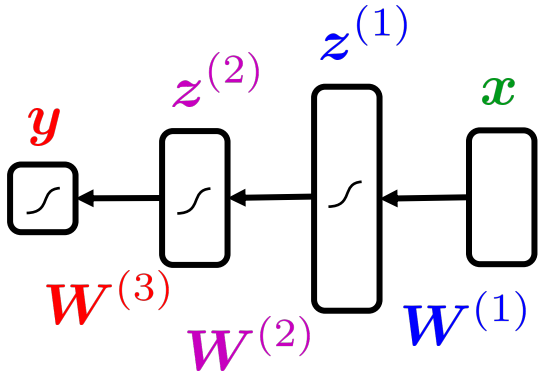
```
array([[0.56110293, 0.56092673, 0.560879  , 0.56122544, 0.56107791]])
```

Entire batch of inputs is vectorized in a single call, nice!

```
[array([0.56110293]),
 array([0.56092673]),
 array([0.560879]),
 array([0.56122544]),
 array([0.56107791])]
```

Same 5 values that, before, were generated one-at-a-time

Python version of our net (fully vectorized, final)




```
def predict(X,                      # shape (N,2)
            W1, W2, W3,             # shape (3,2) (2,3) (1,2)
            b1, b2, b3):            # shape (3,) (2,) (1,)
    Z1 = sigmoid(X  @ W1.T + b1)    # shape (N,3)
    Z2 = sigmoid(Z1 @ W2.T + b2)    # shape (N,2)
    Y  = sigmoid(Z2 @ W3.T + b3)    # shape (N,1)
    return Y
```

The *architecture* still did not change! We only changed *implementation,* to process many inputs at a time.

Numpy broadcasts 1-D vector as a *row* by default

Same matrix multiply computation as before

```
X = np.random.rand(5, 2)
predict(X, W1, W2, W3, b1, b2, b3)
```

```
array([[0.56110293],
       [0.56092673],
       [0.560879  ],
       [0.56122544],
       [0.56107791]])
```

Entire batch of inputs is vectorized, nice!

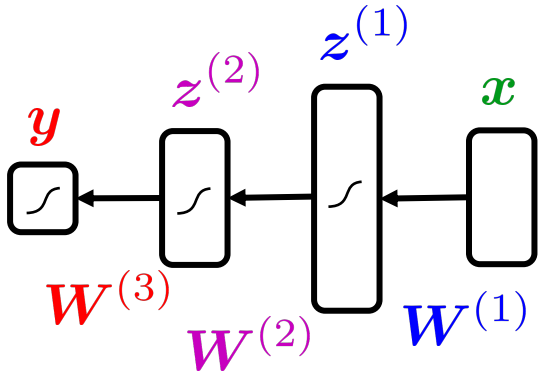
```
X = np.random.rand(10000, 2)
```

```
%time [predict(x, W1, W2, W3, b1, b2, b3) for x in X]
%time predict(X, W1, W2, W3, b1, b2, b3)
```

```
CPU times: user 139 ms, sys: 1.29 ms, total: 140 ms
Wall time: 139 ms
CPU times: user 873 µs, sys: 169 µs, total: 1.04 ms
Wall time: 909 µs
```

Vectorization is crucial for speed of both training and prediction

Python version of our net (vectorized, in a class)




```
class ExampleNet:
    def __init__(self, W1, W2, W3, b1, b2, b3):
        self.params = (W1, W2, W3, b1, b2, b3)

    def predict(self, X):
        W1, W2, W3, b1, b2, b3 = self.params
        Z1 = sigmoid(X  @ W1.T + b1)
        Z2 = sigmoid(Z1 @ W2.T + b2)
        Y  = sigmoid(Z2 @ W3.T + b3)
        return Y

net = ExampleNet(W1, W2, W3, b1, b2, b3)
```

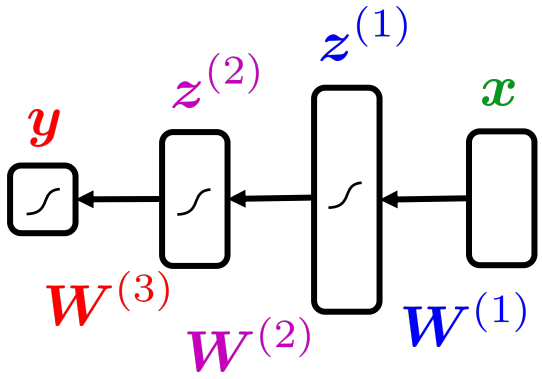
Need to give it some parameter settings

Now variable *net* refers to an *ExampleNet* object that remembers parameters for us, and is easier to call

```
X = np.random.rand(5, 2)
net.predict(X)
```

```
array([[0.56110293],
       [0.56092673],
       [0.560879  ],
       [0.56122544],
       [0.56107791]])
```

Python version of our net (vectorized, flexible)




```
class ExampleNet:
    def __init__(self, W1, W2, W3, b1, b2, b3, activation_func=sigmoid):
        self.params = (W1, W2, W3, b1, b2, b3)
        self.activation_func = activation_func

    def predict(self, X):
        W1, W2, W3, b1, b2, b3 = self.params
        actfunc = self.activation_func
        Z1 = actfunc(X  @ W1.T + b1)
        Z2 = actfunc(Z1 @ W2.T + b2)
        Y  = sigmoid(Z2 @ W3.T + b3)
        return Y

net = ExampleNet(W1, W2, W3, b1, b2, b3, relu)
```

allow other activation functions

override the default

```
def relu(a):
    return np.maximum(0, a)
```

```
X = np.random.rand(5, 2)
net.predict(X)
```

```
array([[0.56177784],
       [0.55702734],
       [0.55573794],
       [0.56515379],
       [0.56116858]])
```

different predictions for same parameters,
due to different activation function

Note: although this example shows sigmoid (logistic) applied at output, when training a classifier, the output activation is not explicitly computed

sklearn.neural_network.MLPClassifier

Implies logistic/softmax output activation function

class `sklearn.neural_network.` **`MLPClassifier`**(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*)

Lots of terminology and hyperparameters!
But some you already know from first lecture

Multi-layer Perceptron classifier.

negative log likelihood

This model optimizes the log-loss function using LBFGS or stochastic gradient descent.

Parameters:

? **hidden_layer_sizes** : ***tuple, length = n_layers - 2, default=(100,)***
The ith element represents the number of neurons in the ith hidden layer.

? **activation** : ***{'identity', 'logistic', 'tanh', 'relu'}, default='relu'***
Activation function for the hidden layer.

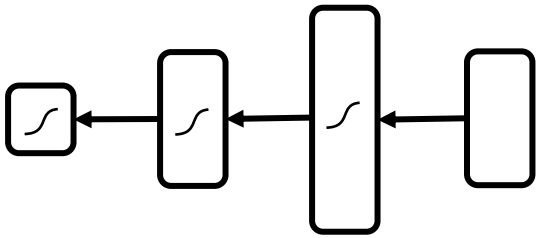
solver : ***{'lbfgs', 'sgd', 'adam'}, default='adam'***
The solver for weight optimization.

alpha : ***float, default=0.0001***
L2 penalty (regularization term) parameter.

batch_size : ***int, default='auto'***
Size of minibatches for stochastic optimizers. If the solver is 'lbfgs', the

learning_rate : ***{'constant', 'invscaling', 'adaptive'}, default='constant'***
Learning rate schedule for weight updates.

Note on the big picture



- Neural networks with ‘sigmoid’ (logistic) activations are just multiple-output logistic regressions, *stacked* on top of one another!
  - Regression output activation function is *linear* (do nothing)
  - Classification output activation function is *logistic* (for binary) or *softmax* (for multiclass)

😎 All parameters can be trained *jointly*; still by MLE (maximum likelihood) or MAP (maximum a posterior)

- Neural networks with are *universal approximators*, meaning that for any function (no matter how complex) there exists a neural network that can approximate it.

🤯

What do activation functions *do*?

- The non-linearity at each hidden layer is essential for the power of neural networks.
- Without them, you have no more "expressive power" than you did with linear regression

Linear activation functions (i.e., do nothing)

```
z1 = W1 @ x  + b1
z2 = W2 @ z1 + b2
y  = W3 @ z2 + b3
```

Substitute z1 and z2

```
y = W3 @ (W2 @ (W1 @ x  + b1) + b2) + b3
```

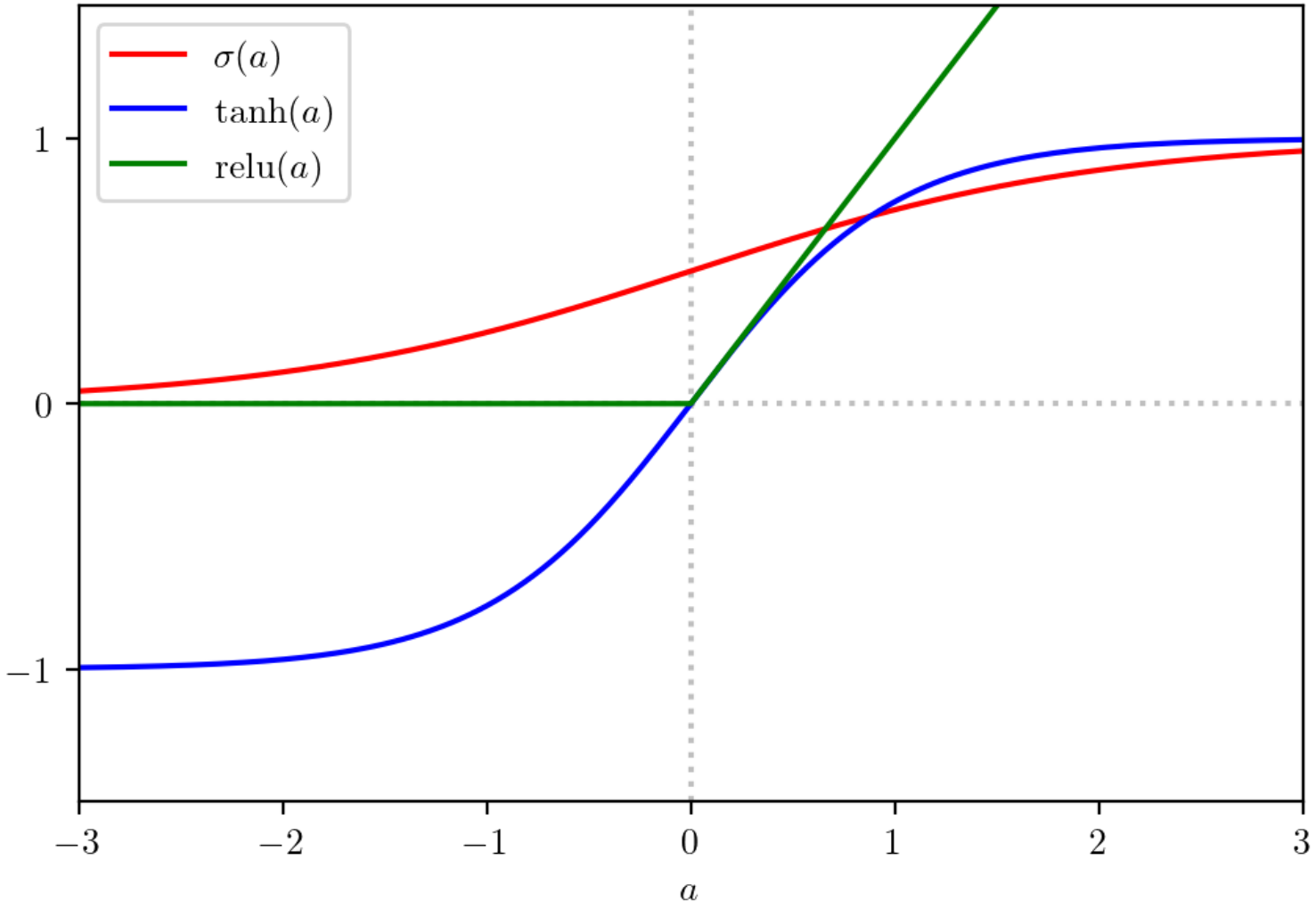
Since no non-linearity, the matrix multiplications are distributive and we can regroup terms

```
y = (W3 @ W2 @ W1) @ x + (W3 @ W2) @ b1 + (W3 @ b2) + b3
```

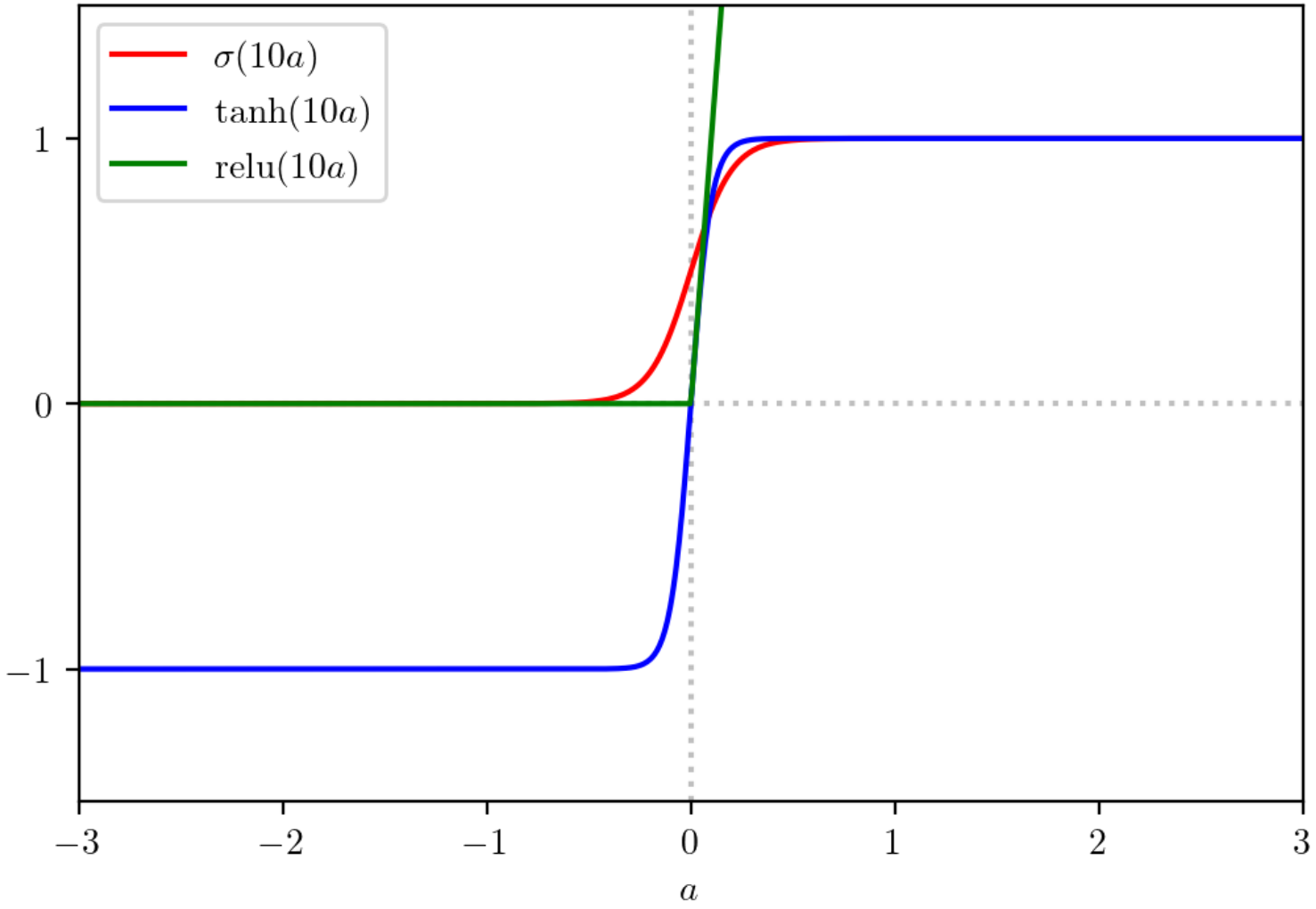
```
y = W @ x + b # where W = (W3@W2@W1) and b = (W3@W2)@b1 + (W3@b2) + b3
```

Our 2-3-2-1 linear neural network was just over-parametrized 2-1 linear regression!!

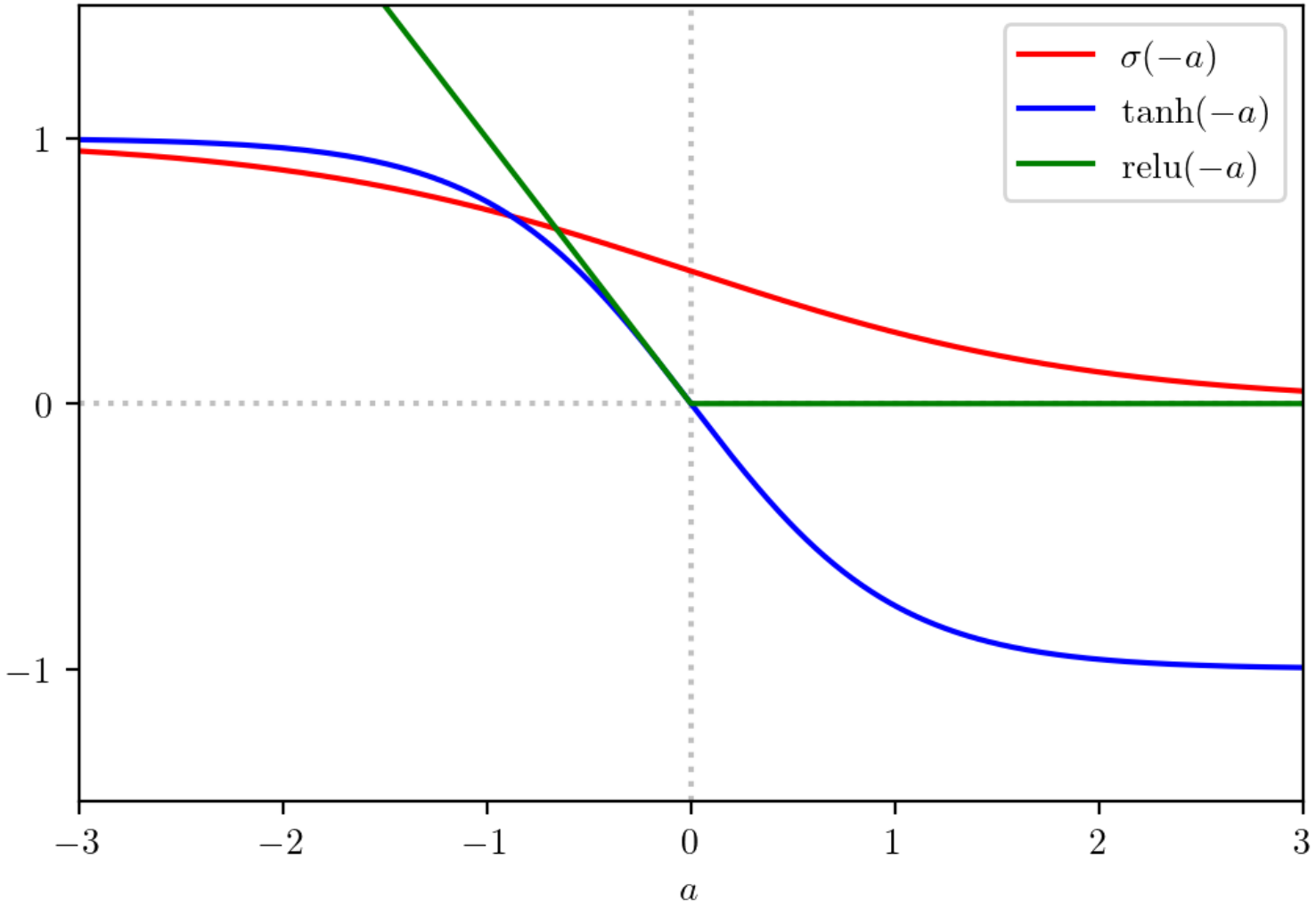
Common “activation functions”



Scaling an activation function



Flipping an activation function



Shifting an activation function

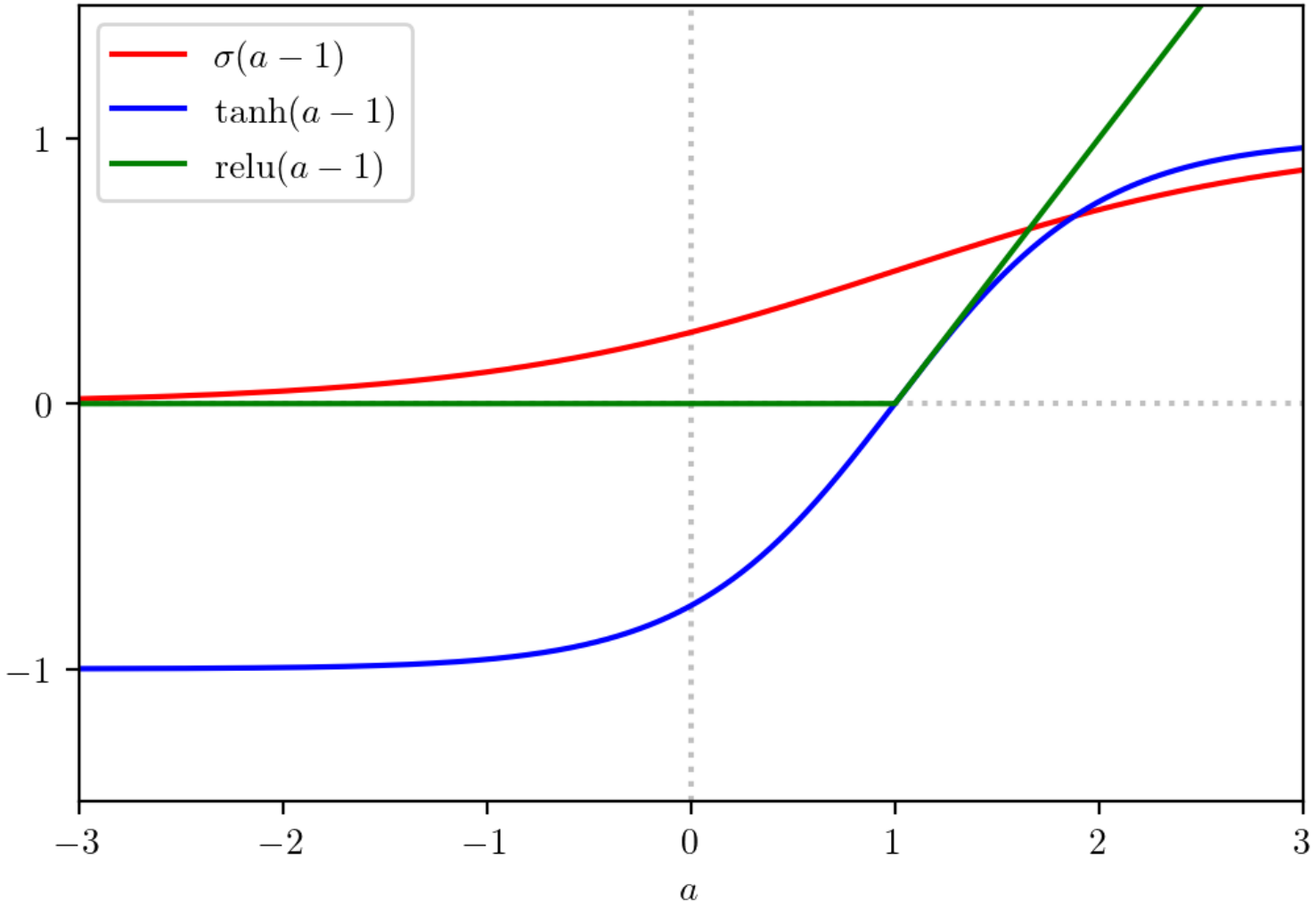
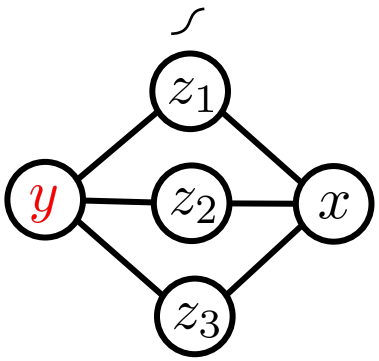
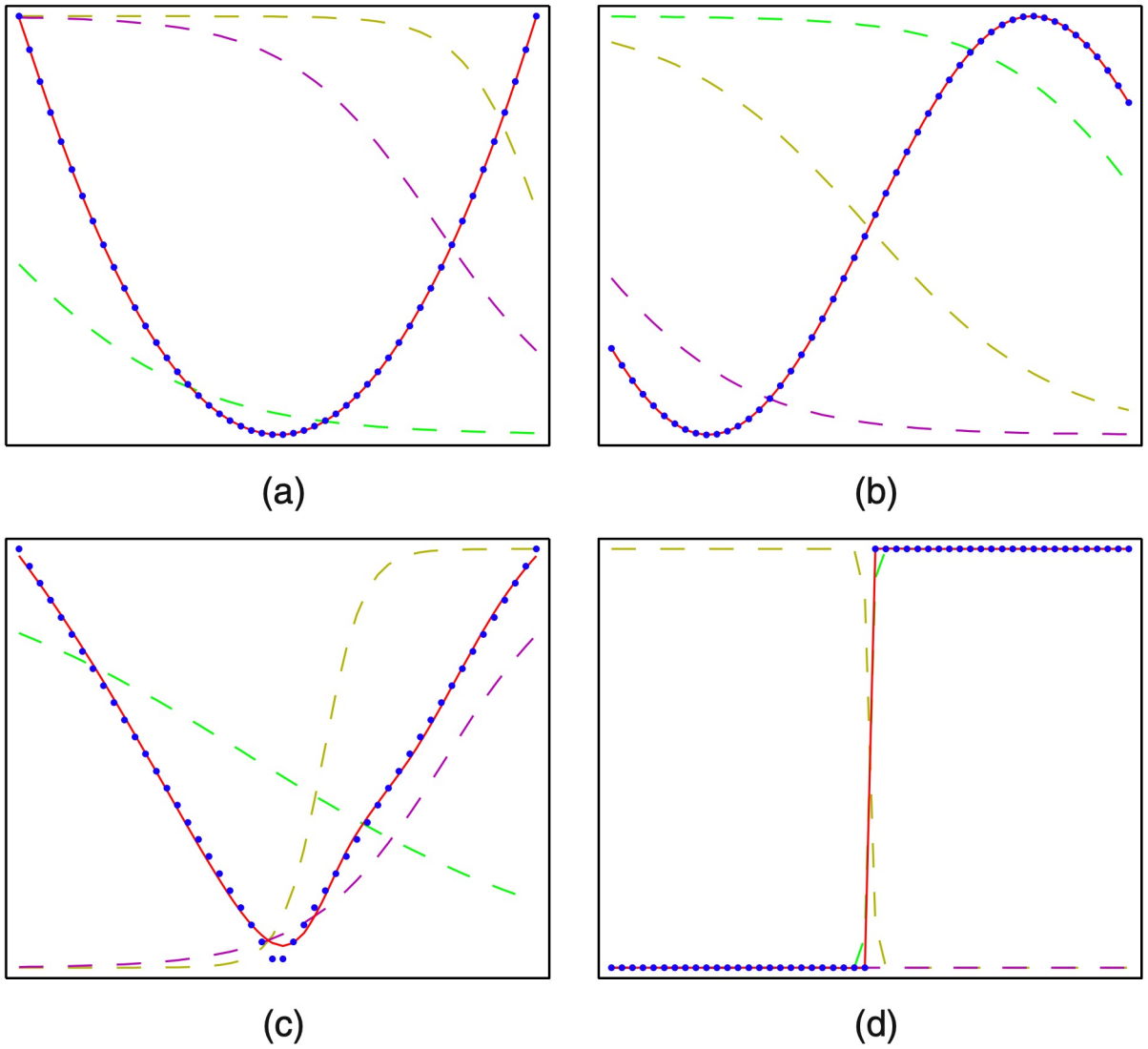


Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with 'tanh' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

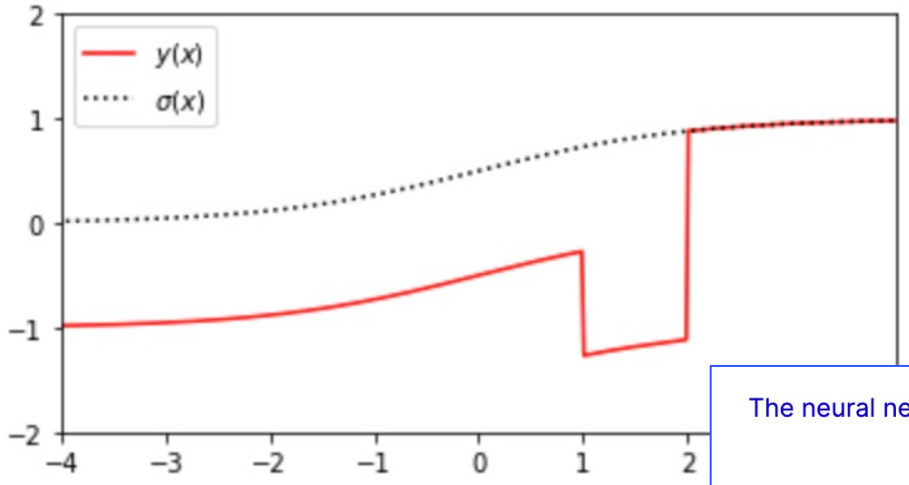


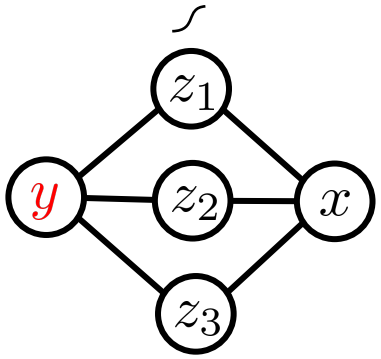


$$z_1 = \tanh(w_1 x + b_1)$$
$$z_2 = \tanh(w_2 x + b_2)$$
$$z_3 = \tanh(w_3 x + b_3)$$
$$y = w_4 z_1 + w_5 z_2 + w_6 z_3 + b_4$$

Practice question: reverse engineering a neural network

[3 marks] Give a sigmoidal neural network, including coefficients, that would closely approximation the function $y(x)$ shown below. Use as few hidden units as possible. The sigmoid $\sigma(x)$ is shown for reference.





The neural network would require three sigmoidal hidden units, taking the form

$$z_1 = \sigma(w_1 x + b_1)$$
$$z_2 = \sigma(w_2 x + b_2)$$
$$z_3 = \sigma(w_3 x + b_3)$$
$$y = w_4 z_1 + w_5 z_2 + w_6 z_3 + b_4$$

By using a large first layer weight, such as $w = 1000$, we can achieve a step function with a sigmoid. A good approximation of the red curve is:

$$w_1 = 1 \qquad b_1 = 0$$
$$w_2 = 1000 \qquad b_2 = -1000$$
$$w_3 = 1000 \qquad b_3 = -2000$$
$$w_4 = 1 \qquad w_5 = -1 \qquad w_6 = 2 \qquad b_4 = -1$$

Training a neural network

- Neural networks are trained by gradient descent
- Within the gradient descent, gradients are computed by a special algorithm called "backpropagation"
- Backpropagation is an *efficient algorithm* to *compute parameter gradients*.
  - Equivalent to "reverse mode automatic differentiation"
  - Used within many gradient-based training algorithms (SGD, RMSProp, Adam); it is not itself a training algorithm.
- Read supplementary note in Moodle, then Bishop

Backpropagation in pure Numpy

```python
class ExampleNet:
    def __init__(self, W1, W2, W3, b1, b2, b3):
        self.params = (W1, W2, W3, b1, b2, b3)

    def predict(self, X):
        W1, W2, W3, b1, b2, b3 = self.params
        Z1 = sigmoid(X  @ W1.T + b1)
        Z2 = sigmoid(Z1 @ W2.T + b2)
        Z3 = sigmoid(Z2 @ W3.T + b3)
        self.hidden = (Z1, Z2, Z3)    # Keep these for backprop
        return Z3

    def backprop(self, Y):
        W1, W2, W3, b1, b2, b3 = self.params
        Z1, Z2, Z3 = self.hidden

        D3 = Z3 - Y                          # Deltas for layer 3 (output)
        D2 = sigmoid_grad(Z2) * (D3 @ W3)    # Deltas for layer 2
        D1 = sigmoid_grad(Z1) * (D2 @ W2)    # Deltas for layer 1

        W3_grad = D3.T @ Z2        # Gradient for W3
        W2_grad = D2.T @ Z1        # Gradient for W2
        W1_grad = D1.T @ X         # Gradient for W1

        b3_grad = D3.sum(axis=0)   # Gradient for b3
        b2_grad = D2.sum(axis=0)   # Gradient for b2
        b1_grad = D1.sum(axis=0)   # Gradient for b1

        return W1_grad, W2_grad, W3_grad, \
               b1_grad, b2_grad, b3_grad

net = ExampleNet(W1, W2, W3, b1, b2, b3)
```

"3-layer sigmoid network, with sigmoid output activations"

Efficient algorithm for computing gradient

```python
X = np.random.rand(5, 2)
Y = np.random.randint(2, size=(5, 1))
print(Y)
```

```
[[0]
 [1]
 [1]
 [0]
 [0]]
```

```python
net.predict(X)
grads = net.backprop(Y)
```

```python
for name, grad in zip(["W1", "W2", "W3", "b1", "b2", "b3"], grads):
    print("%s_grad:" % name); print(grad)
```

```
W1_grad:
[[0.00263115 0.00239826]
 [0.00282629 0.00256289]
 [0.0028683  0.0025841 ]]
W2_grad:
[[0.0083168  0.00946165 0.01052009]
 [0.01549284 0.01761128 0.01957102]]
W3_grad:
[[0.66728894 0.45476535]]
b1_grad:
[0.00324869 0.00346679 0.00348111]
b2_grad:
[0.01486643 0.02770982]
b3_grad:
[0.80521201]
```

Given the computation you specify, autograd libraries (TensorFlow, PyTorch, etc.) automatically implement the backpropagation for you, correctly – very powerful!

These days you don't need to write the red code yourself.

But always try simpler baselines too!!

Uh oh, you do not want to see a "matters arising" about your own publication....

MATTERS ARISING

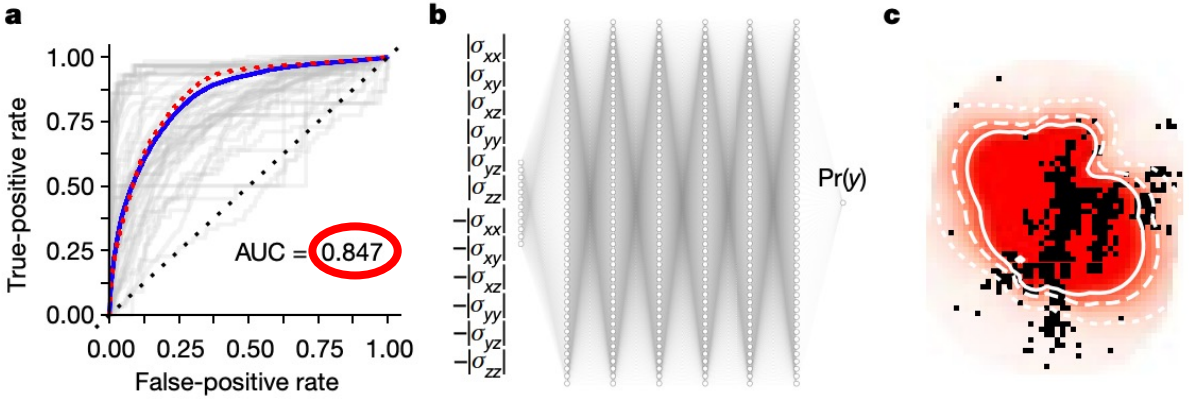
https://doi.org/10.1038/s41586-019-1582-8

(logistic regression ☺)

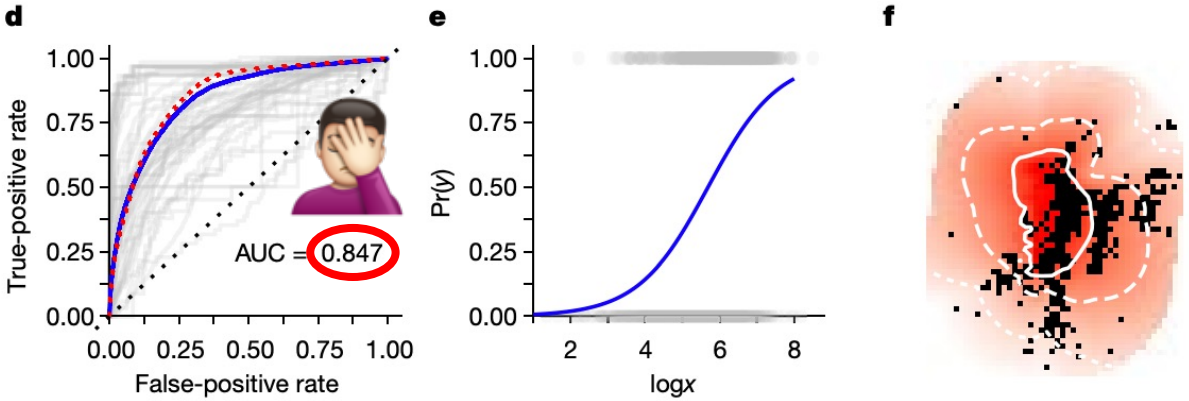
One neuron versus deep learning in aftershock prediction

Arnaud Mignan[1,2,3]* & Marco Broccardo[2,4]*

Deep neural network:
(from a paper in *Nature*!)



Logistic regression:
(applied to the same data)



Deep learning didn't actually do any better here!

PRML Readings

You will not have to memorize any formulas, but you should understand the notation and arguments made

§5.0.0 Neural Networks

§5.1.0 Feed-forward Network Functions

§5.2.0 Network Training

§5.2.1 Parameter optimization

§5.2.4 Gradient descent optimization

§5.3.0 Error Backpropagation

§5.3.1 Evaluation of error-function derivatives

§5.3.2 A simple example

…

PRML Readings

…

§5.3.3 Efficiency of backpropagation

§5.3.4 The Jacobian Matrix

§5.5.0 Regularization in Neural Networks

§5.5.2 Early stopping