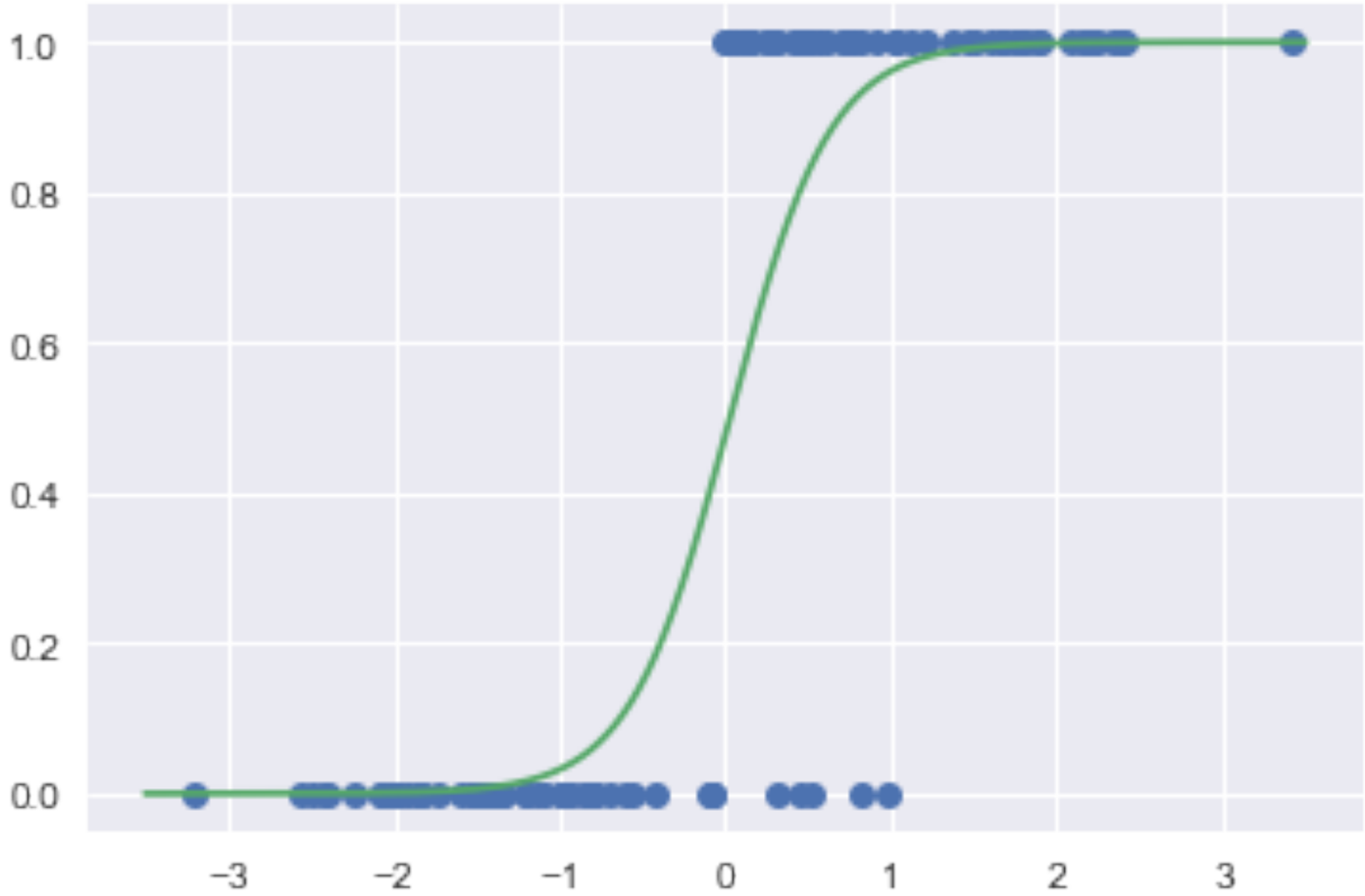
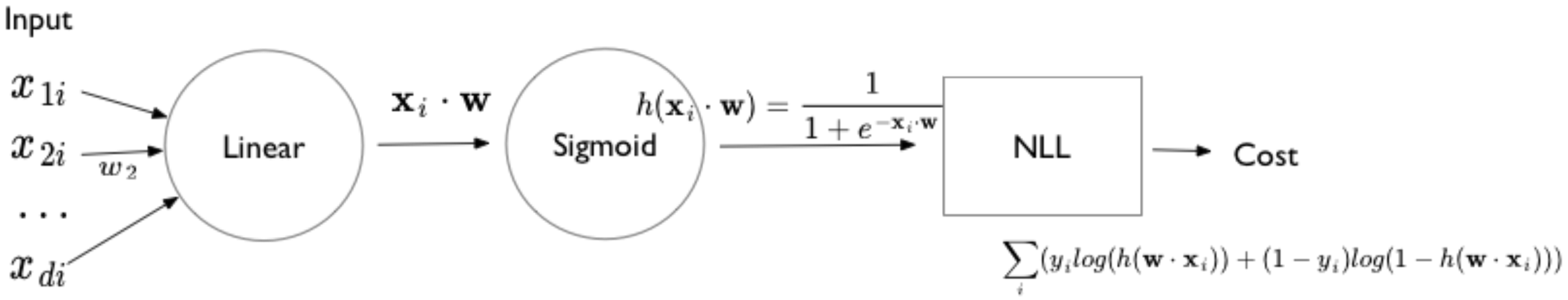
Day 2 Session 2

Fully Connected Networks

Logistic Regression



Standard Layers



Softmax formulation

- Identify p_i and $1 - p_i$ as two separate probabilities constrained to add to 1. That is $p_{1i} = p_i; p_{2i} = 1 - p_i$.
- $p_{1i} = \frac{e^{\mathbf{w}_1 \cdot \mathbf{x}}}{e^{\mathbf{w}_1 \cdot \mathbf{x}} + e^{\mathbf{w}_2 \cdot \mathbf{x}}}$
- $p_{2i} = \frac{e^{\mathbf{w}_2 \cdot \mathbf{x}}}{e^{\mathbf{w}_1 \cdot \mathbf{x}} + e^{\mathbf{w}_2 \cdot \mathbf{x}}}$
- Can translate coefficients by fixed amount ψ without any change

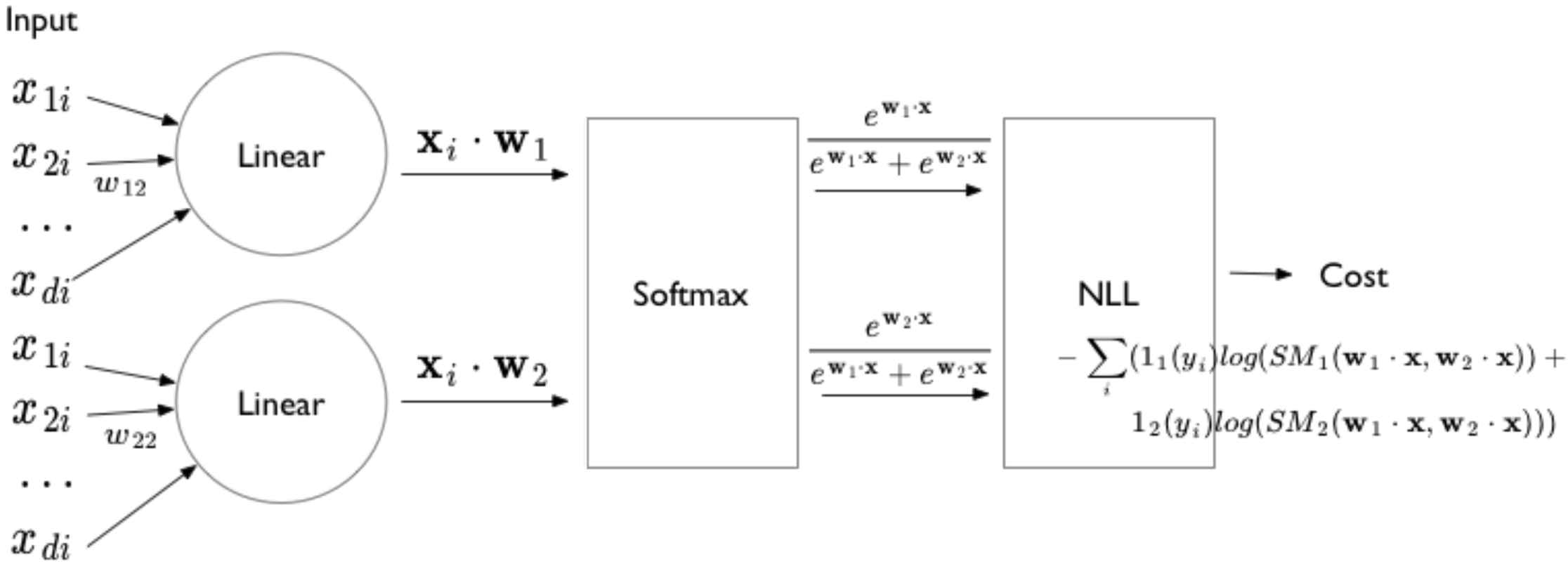
NLL and gradients for Softmax

$$\mathcal{L} = \prod_i p_{1i}^{1_1(y_i)} p_{2i}^{1_2(y_i)}$$

$$NLL = -\sum_i (1_1(y_i) log(p_{1i}) + 1_2(y_i) log(p_{2i}))$$

$$\frac{\partial NLL}{\partial \mathbf{w}_1} = -\sum_i \mathbf{x}_i (y_i - p_{1i}), \frac{\partial NLL}{\partial \mathbf{w}_2} = -\sum_i \mathbf{x}_i (y_i - p_{2i})$$

Units diagram for Softmax

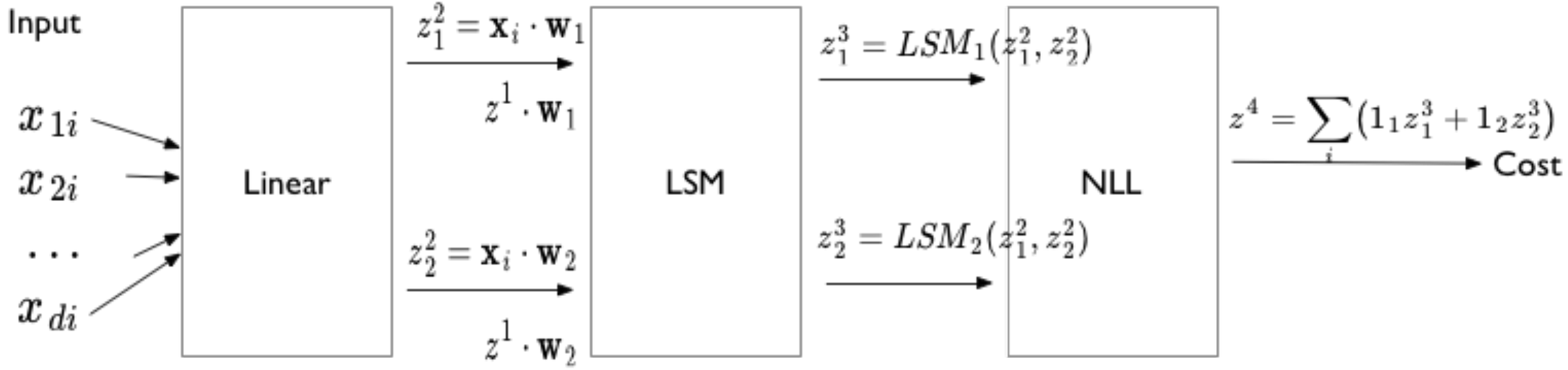


Rewrite NLL

$$NLL = -\sum_i (1_1(y_i) LSM_1(\mathbf{w}_1 \cdot \mathbf{x}, \mathbf{w}_2 \cdot \mathbf{x}) + 1_2(y_i) LSM_2(\mathbf{w}_1 \cdot \mathbf{x}, \mathbf{w}_2 \cdot \mathbf{x}))$$

where $SM_1 = \frac{e^{\mathbf{w}_1 \cdot \mathbf{x}}}{e^{\mathbf{w}_1 \cdot \mathbf{x}} + e^{\mathbf{w}_2 \cdot \mathbf{x}}}$ puts the first argument in the numerator. Ditto for LSM_1 which is simply $log(SM_1)$.

MLE for Logistic Regression



$z^1 = \mathbf{x}_i$

Equations, layer by layer

$$\mathbf{z}^1 = \mathbf{x}_i$$

$$\mathbf{z}^2 = (z_1^2, z_2^2) = (\mathbf{w}_1 \cdot \mathbf{x}_i, \mathbf{w}_2 \cdot \mathbf{x}_i) = (\mathbf{w}_1 \cdot \mathbf{z}_i^1, \mathbf{w}_2 \cdot \mathbf{z}_i^1)$$

$$\mathbf{z}^3 = (z_1^3, z_2^3) = \big(LSM_1(z_1^2, z_2^2), LSM_2(z_1^2, z_2^2)\big)$$

$$z^4 = NLL(\mathbf{z}^3) = NLL(z_1^3, z_2^3) = -\sum_i \big(1_1(y_i) z_1^3(i) + 1_2(y_i) z_1^3(i)\big)$$

Reverse Mode Differentiation

$$Cost = f^{Loss}(\mathbf{f}^3(\mathbf{f}^2(\mathbf{f}^1(\mathbf{x}))))$$

$$\nabla_{\mathbf{x}} Cost = \frac{\partial f^{Loss}}{\partial \mathbf{f}^3} \frac{\partial \mathbf{f}^3}{\partial \mathbf{f}^2} \frac{\partial \mathbf{f}^2}{\partial \mathbf{f}^1} \frac{\partial \mathbf{f}^1}{\partial \mathbf{x}}$$

Write as:

$$\nabla_{\mathbf{x}} Cost = (((\frac{\partial f^{Loss}}{\partial \mathbf{f}^3} \frac{\partial \mathbf{f}^3}{\partial \mathbf{f}^2}) \frac{\partial \mathbf{f}^2}{\partial \mathbf{f}^1}) \frac{\partial \mathbf{f}^1}{\partial \mathbf{x}})$$

From Reverse Mode to Back Propagation

- Recursive Structure
- Always a vector times a Jacobian
- We add a "cost layer" to z^4. The derivative of this layer with respect to z^4 will always be 1.
- We then propagate this derivative back.

Backpropagation

RULE1: FORWARD (`.forward` in pytorch) $\mathbf{z}^{l+1} = \mathbf{f}^l(\mathbf{z}^l)$

RULE2: BACKWARD (`.backward` in pytorch)

$\delta^l = \dfrac{\partial C}{\partial \mathbf{z}^l}$ or $\delta^l_u = \dfrac{\partial C}{\partial z^l_u}$.

$$\delta^l_u = \frac{\partial C}{\partial z^l_u} = \sum_v \frac{\partial C}{\partial z^{l+1}_v} \frac{\partial z^{l+1}_v}{\partial z^l_u} = \sum_v \delta^{l+1}_v \frac{\partial z^{l+1}_v}{\partial z^l_u}$$

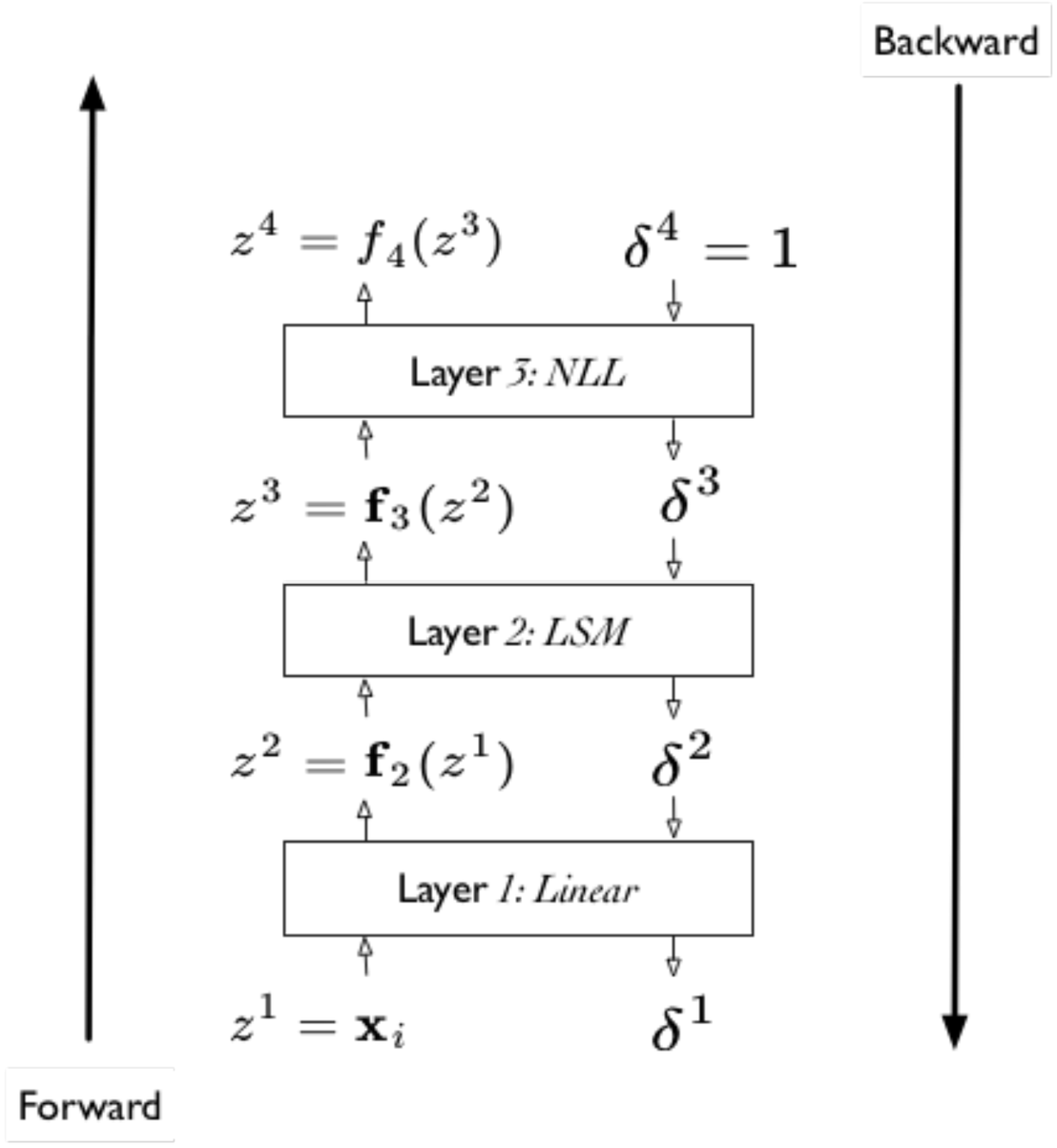
In particular:

$$\delta_u^3 = \frac{\partial z^4}{\partial z_u^3} = \frac{\partial C}{\partial z_u^3}$$

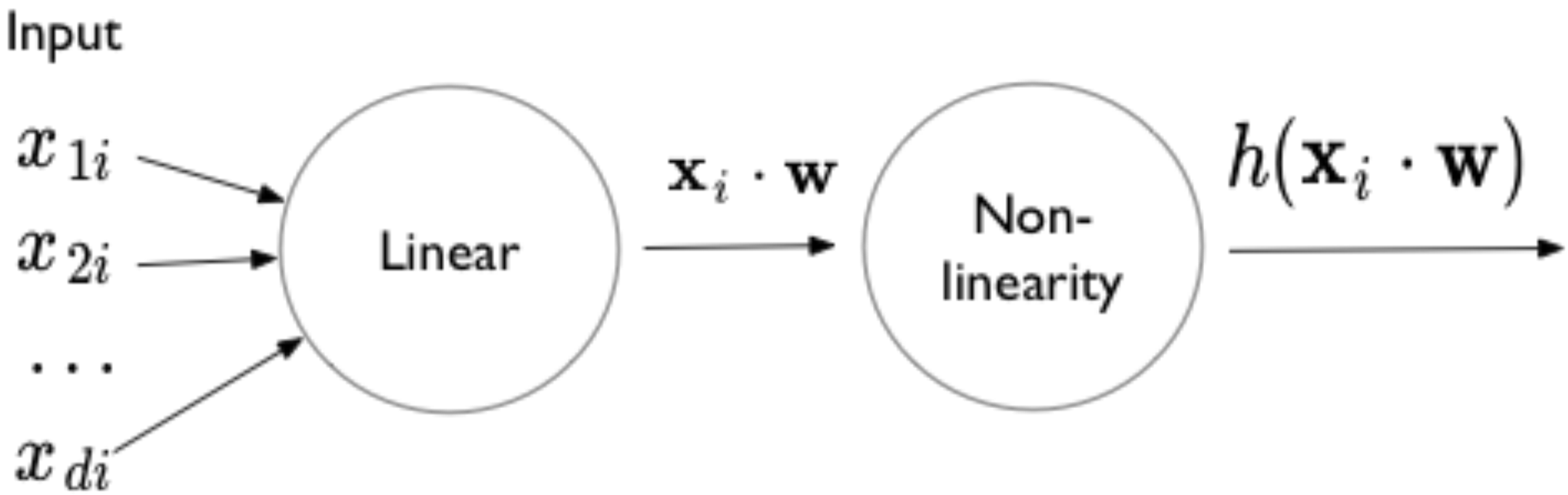
RULE 3: PARAMETERS

$$\frac{\partial C}{\partial \theta^l} = \sum_u \frac{\partial C}{\partial z_u^{l+1}} \frac{\partial z_u^{l+1}}{\partial \theta^l} = \sum_u \delta_u^{l+1} \frac{\partial z_u^{l+1}}{\partial \theta^l}$$

(backward pass is thus also used to fill the `variable.grad` parts of parameters in pytorch)


Backward
$z^4 = f_4(z^3)$
$\delta^4 = 1$
Layer 3: NLL
$z^3 = \mathbf{f}_3(z^2)$
δ^3
Layer 2: LSM
$z^2 = \mathbf{f}_2(z^1)$
δ^2
Layer 1: Linear
$z^1 = \mathbf{x}_i$
δ^1
Forward

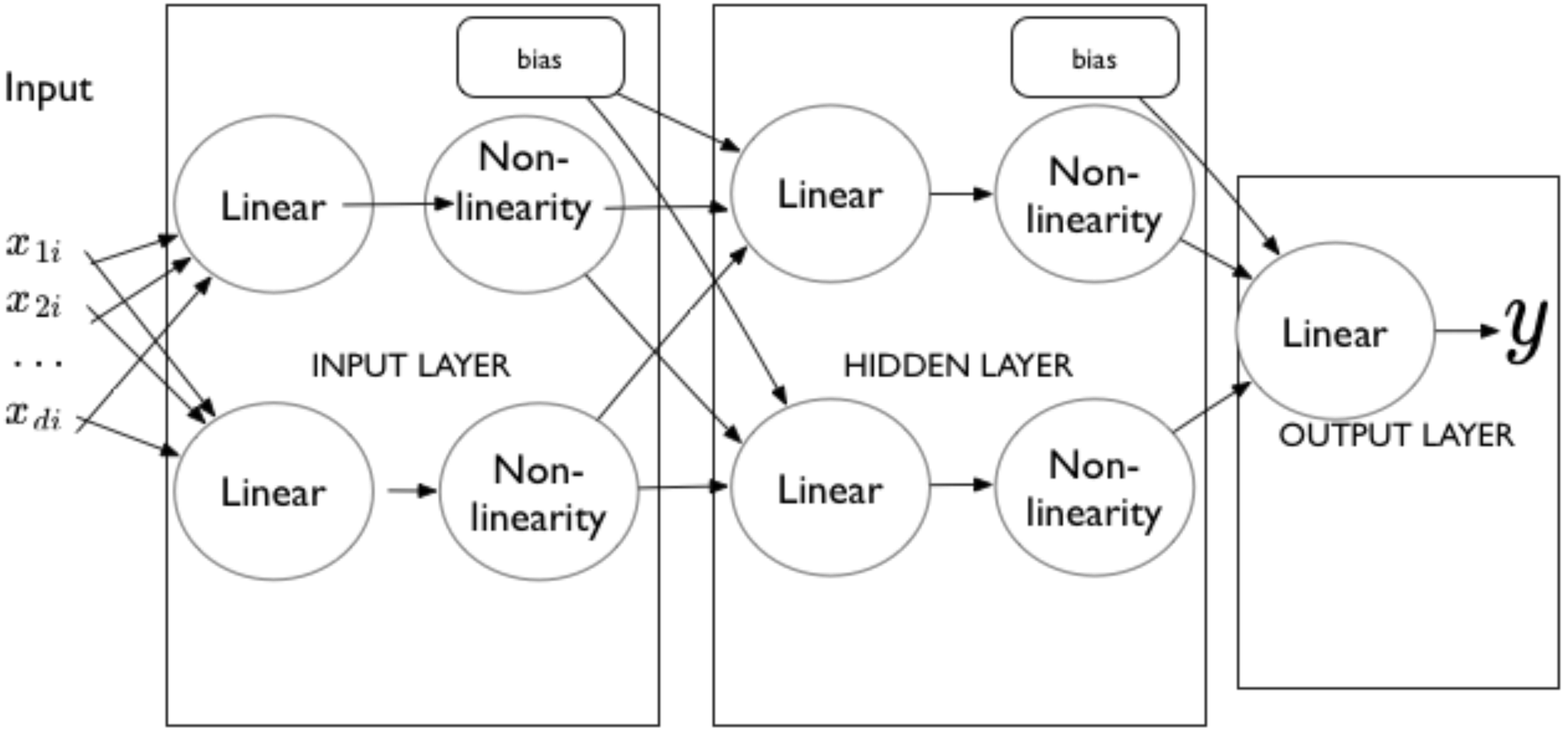
Feed Forward Neural Nets: The perceptron



Just combine perceptrons

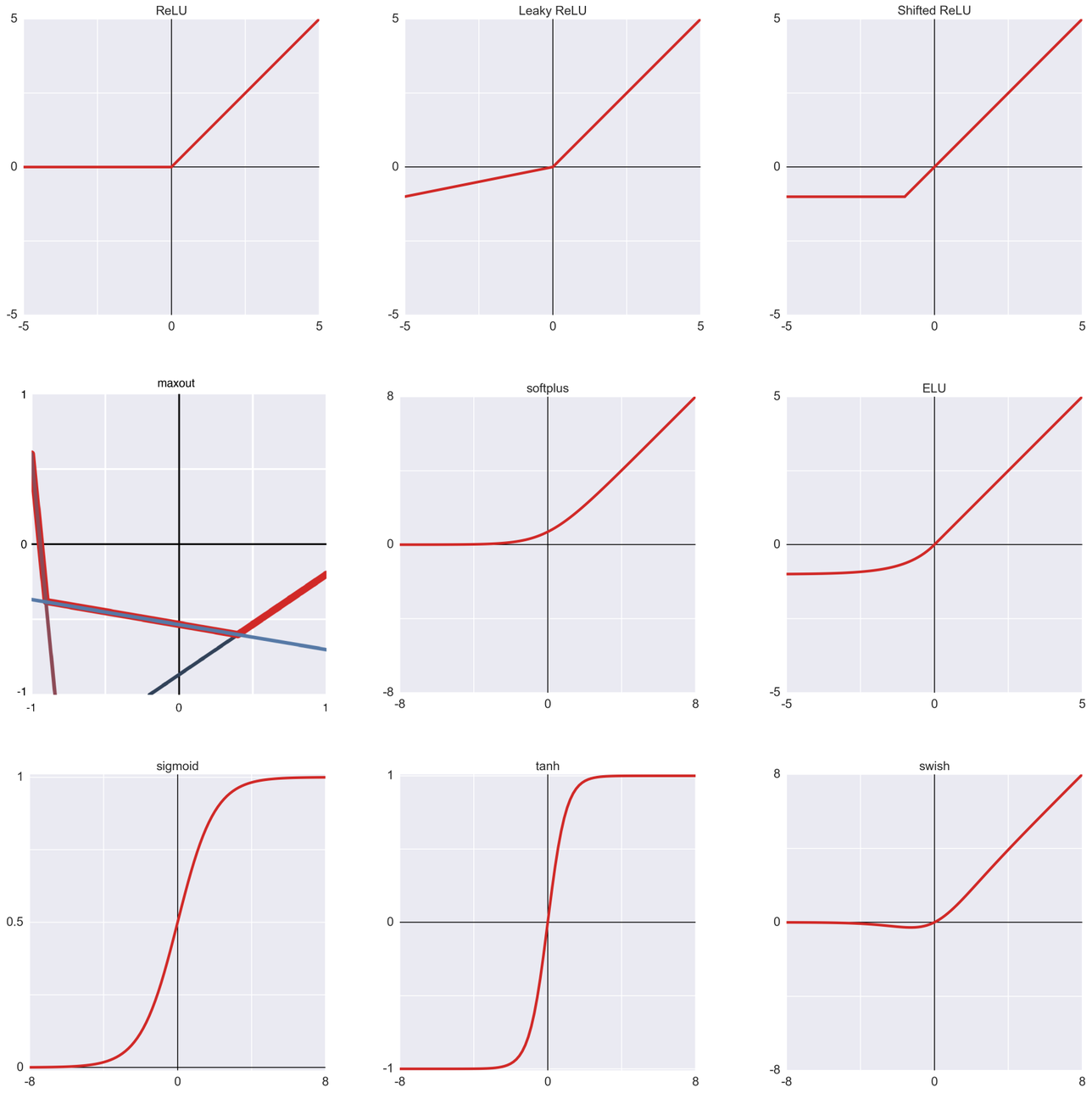
- both deep and wide
- this buys us complex nonlinearity
- both for regression and classification
- key technical advance: BackPropagation with
- autodiff
- key technical advance: gpu

Combine Perceptrons

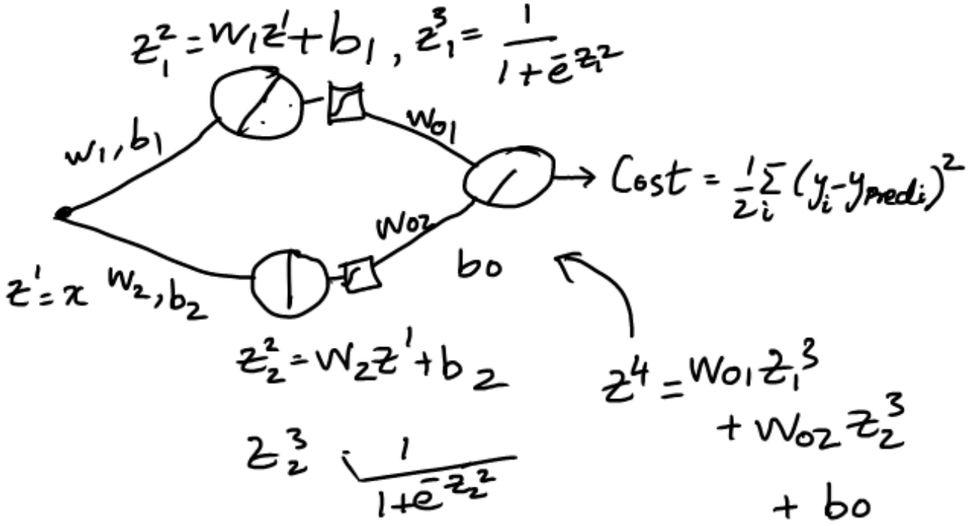


Non-Linearity

- we want a non-linearity as othersie combining linear regressions just gives a big honking linear regression
- the relu style is popular even with the kinks creating issues for differentiability because of the lack of saturations
- when need continuous derivatives we use sigmoid and tanh style functions which suffer from saturation issues (regularization helps)



Simple MLP



$$z^5 = \frac{1}{2}(z^4 - y)^2 \qquad z^6 = \text{Cost} = \sum_i z^{5(i)}$$

$$\frac{\partial z^6}{\partial z^5} = 1, \frac{\partial z^5}{\partial z^4} = z^4 - y, \frac{\partial z^4}{\partial z_1^3} = w_{01}$$

$$\frac{\partial z^4}{\partial z_2^3} = w_{02}, \frac{\partial z^4}{\partial w_{01}} = z_1^3, \frac{\partial z^4}{\partial w_{02}} = z_2^3$$

$$\frac{\partial z^4}{\partial b_0} = 1, \frac{\partial z_1^3}{\partial z_1^2} = z_1^3(1 - z_1^3),$$

$$\frac{\partial z_2^3}{\partial z_2^2} = z_2^3(1 - z_2^3), \frac{\partial z_1^2}{\partial z^1} = w_1,$$

$$\frac{\partial z_2^2}{\partial z^1} = w_2, \frac{\partial z_1^2}{\partial w_1} = z^1, \frac{\partial z_2^2}{\partial w_2} = z^1,$$

$$\frac{\partial z_1^2}{\partial b_1} = 1, \frac{\partial z_2^2}{\partial b_2} = 1$$

Forward Pass

We want to obtain gradients. For example: $\frac{\partial Cost}{\partial param} = \frac{\partial z^6}{\partial w_1}$

First we do the **Forward Pass**. Say we have 1 sample: (x=0.1, y=5). Initialize $b_1, w_1, b_2, w_2, w_{o1}, w_{o2}, b_o$. Then, plugging in the numbers will give us some Cost (z^5, z^6).

$z^5 = \frac{1}{2}(z^4 - y)^2$ $z^4 = w_{o1} z^3_1 + w_{o2} z^3_2 + b_o$ $z_1^3 = 1/(1+e^{-z_1^2})$, $z_2^3 = 1/(1+e^{-z_2^2})$ $z_1^2 = w_1 z^1 + b_1$, $z_2^2 = w_2 z^1 + b^2$, $z_1 = x = 0.1$

Backward Pass

Now it is time to find the gradients, for eg, $\frac{\partial z^6}{\partial w_1}$

The basic idea is to gather all parts that go to w_1, and so on and so forth. Now we perform GD (SGD) with some learning rate.

The parameters get updated. Now we repeat the forward pass.

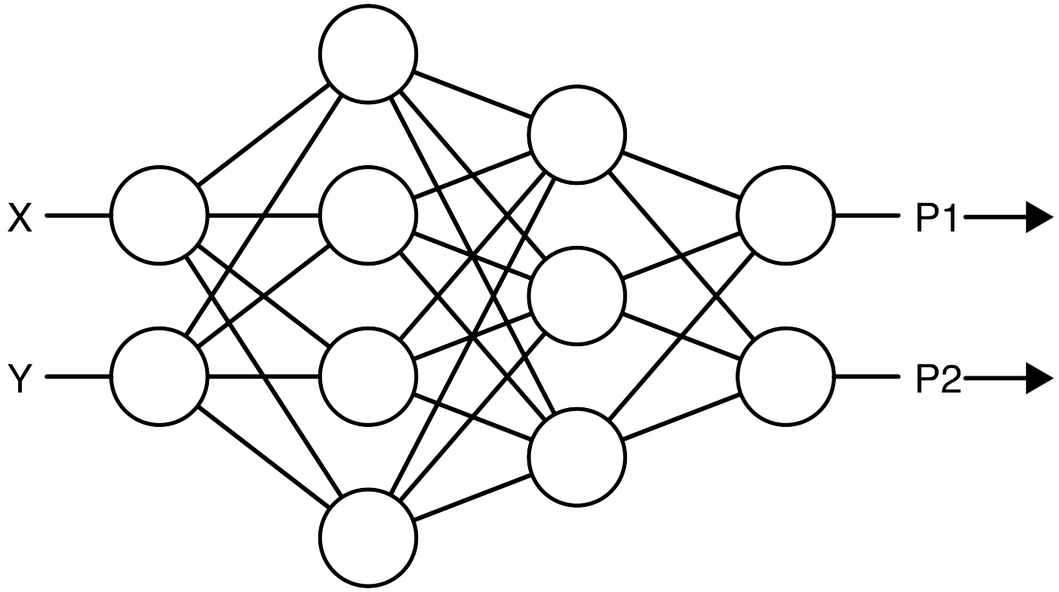
Thats it! Wait for convergence.

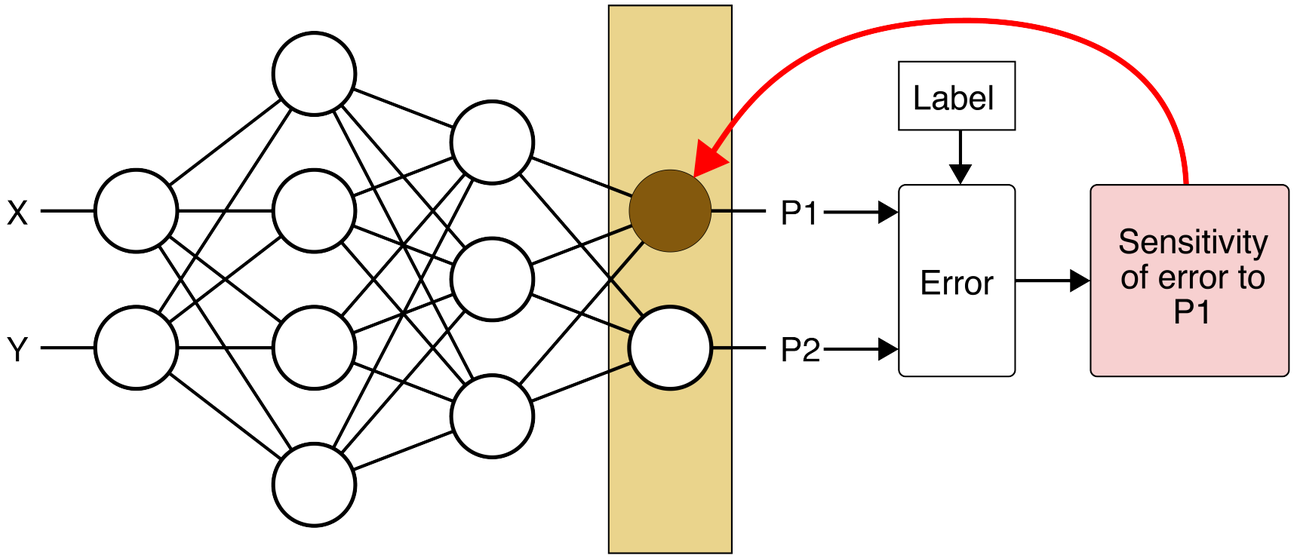
$$\frac{\partial z^5}{\partial w^1} = \frac{\partial z^5}{\partial z^4}\frac{\partial z^4}{\partial z_1^3}\frac{\partial z_1^3}{\partial z_1^2}\frac{\partial z_1^2}{\partial w_1}$$

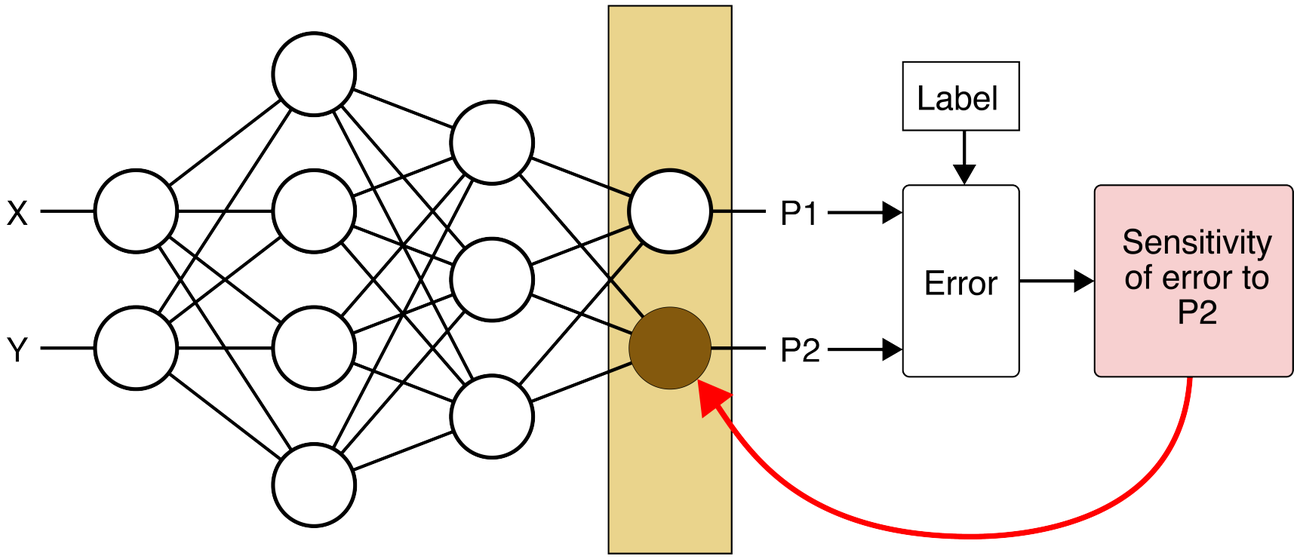
$$= (z^4 - y) \times w_{01} \times z_1^3(1 - z_1^3)\, z^1$$

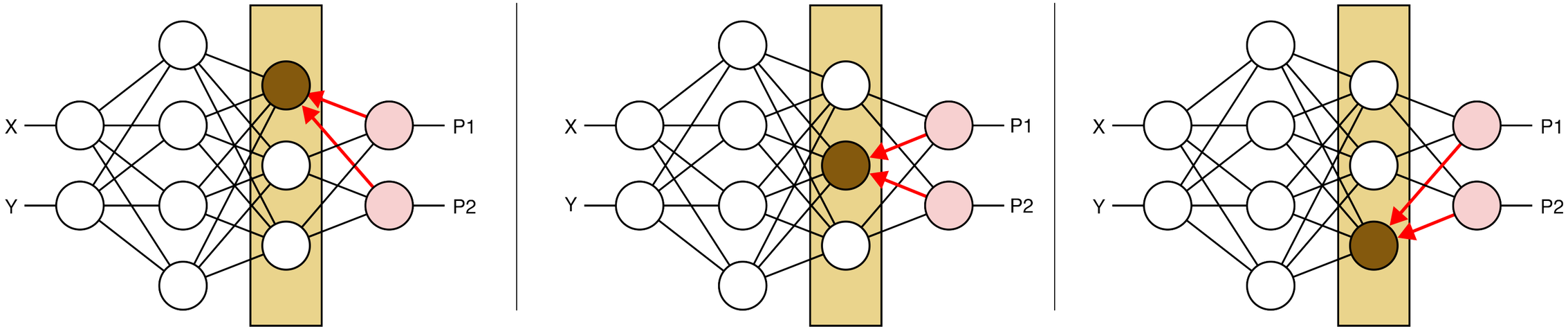
More Complex MLP:

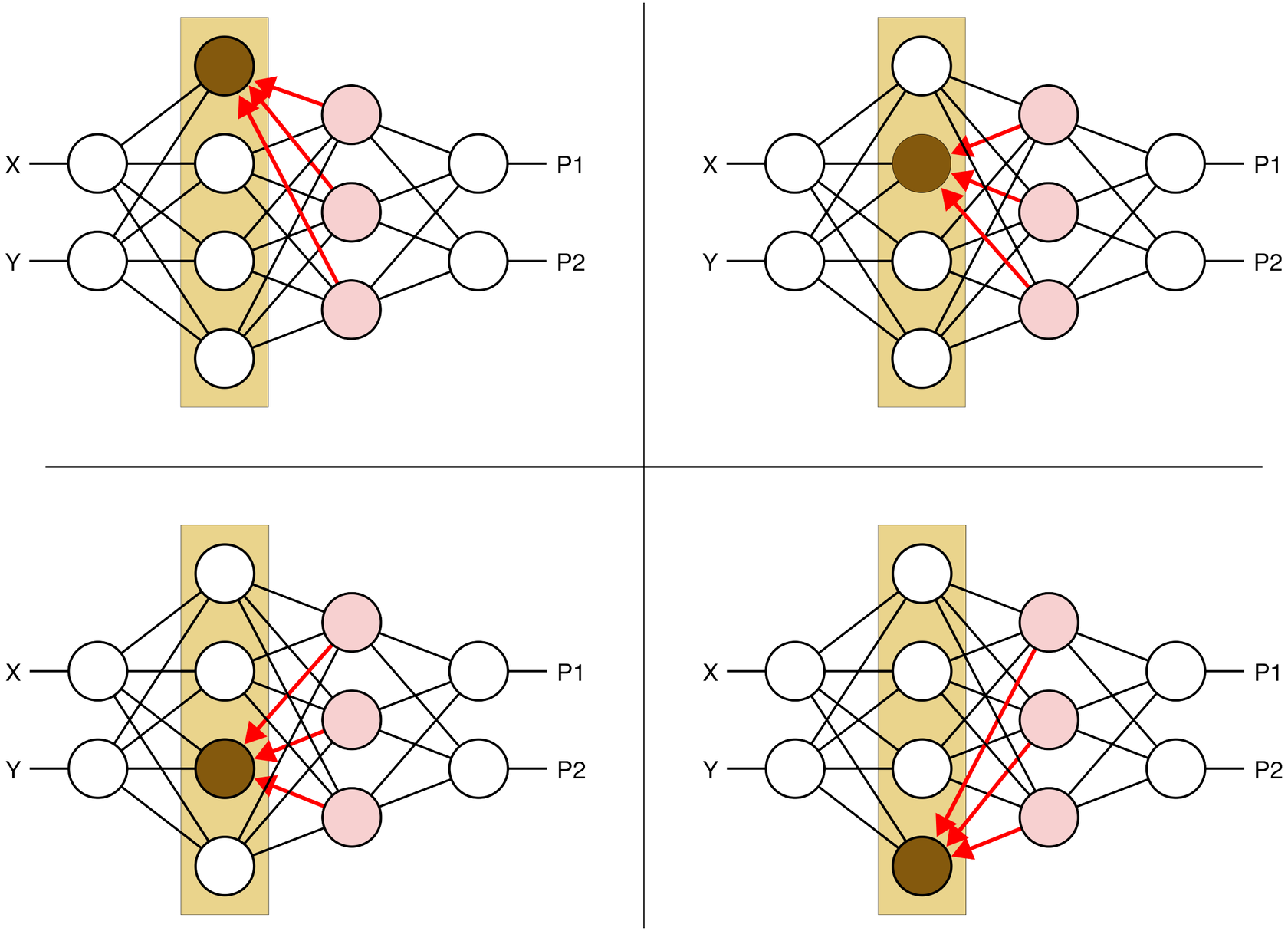
3 hidden layers

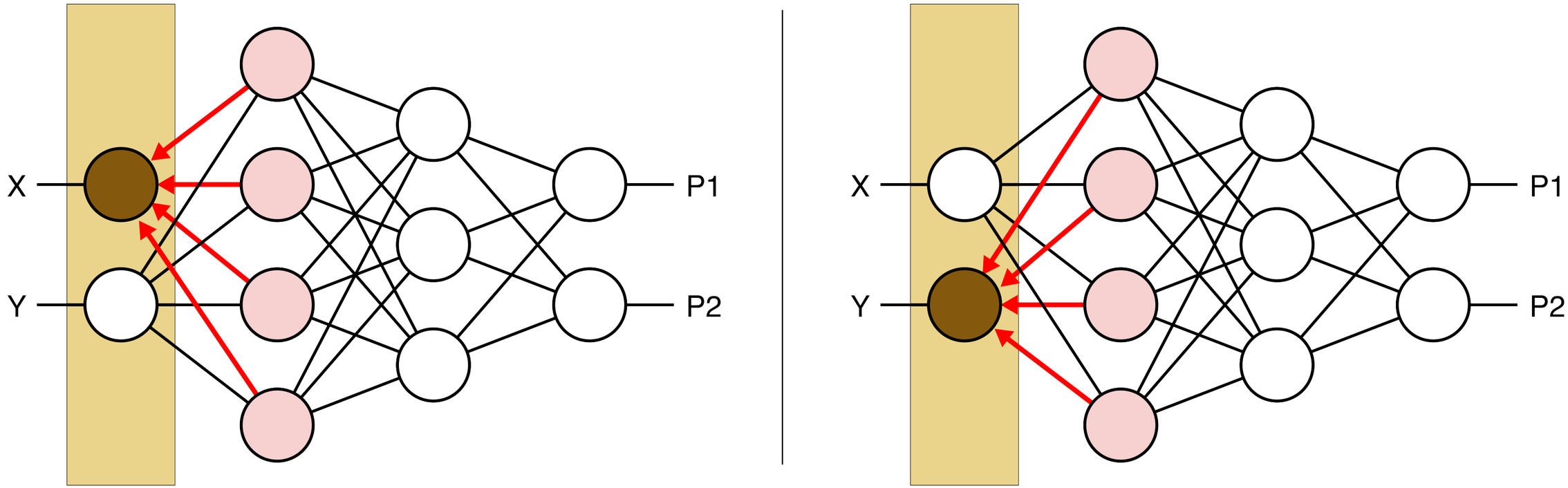







X
Y
P1
P2
X
Y
P1
P2
X
Y
P1
P2

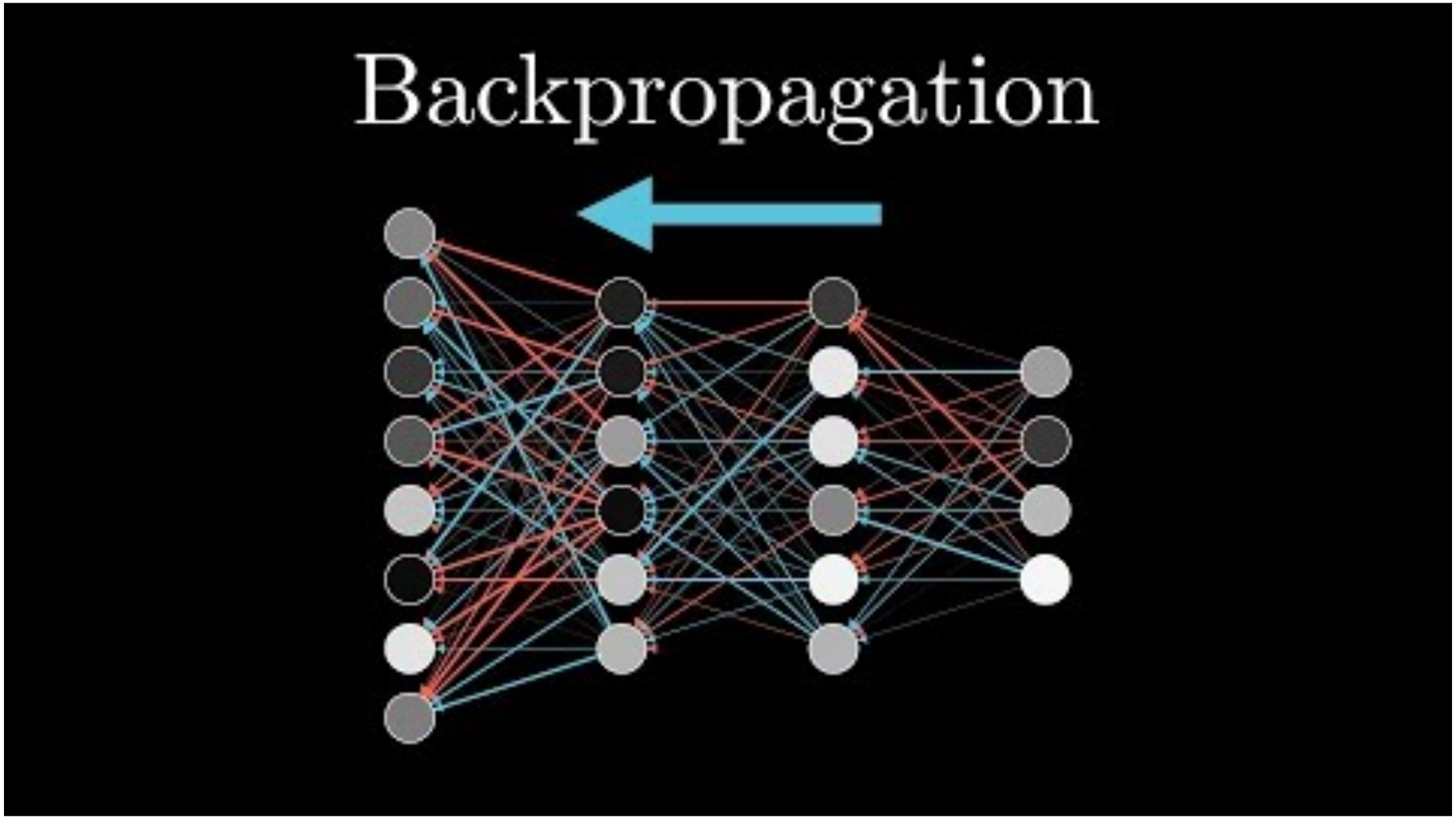

X
Y
P1
P2
X
Y
P1
P2
X
Y
P1
P2
X
Y
P1
P2


X
Y
P1
P2
X
Y
P1
P2

Basic code outline

```
dataset = torch.utils.data.TensorDataset(torch.from_numpy(xgrid.reshape(-1,1)), torch.from_numpy(ygrid))
loader = torch.utils.data.DataLoader(dataset, batch_size=64,shuffle=True)
def run_model(model, epochs):
    criterion = nn.MSELoss()
    lr, epochs, batch_size = 1e-1 , epochs , 64
    optimizer = torch.optim.SGD(model.parameters(), lr = lr )
    accum=[]
    for k in range(epochs):
        localaccum = []
        for localx, localy in iter(loader):
            localx = Variable(localx.float())
            localy = Variable(localy.float())
            output, _, _ = model.forward(localx)
            loss = criterion(output, localy)
            model.zero_grad()
            loss.backward()
            optimizer.step()
            localaccum.append(loss.data[0])
        accum.append((np.mean(localaccum), np.std(localaccum)))
    return accum
```

Backprop (Intuition)



Backpropagation

RULE1: FORWARD (`.forward` in pytorch) $\mathbf{z}^{l+1} = \mathbf{f}^l(\mathbf{z}^l)$

RULE2: BACKWARD (`.backward` in pytorch)

$\delta^l = \dfrac{\partial C}{\partial \mathbf{z}^l}$ or $\delta^l_u = \dfrac{\partial C}{\partial z^l_u}$.

$$\delta^l_u = \frac{\partial C}{\partial z^l_u} = \sum_v \frac{\partial C}{\partial z^{l+1}_v} \frac{\partial z^{l+1}_v}{\partial z^l_u} = \sum_v \delta^{l+1}_v \frac{\partial z^{l+1}_v}{\partial z^l_u}$$

In particular:

$$\delta_u^3 = \frac{\partial z^4}{\partial z_u^3} = \frac{\partial C}{\partial z_u^3}$$

RULE 3: PARAMETERS

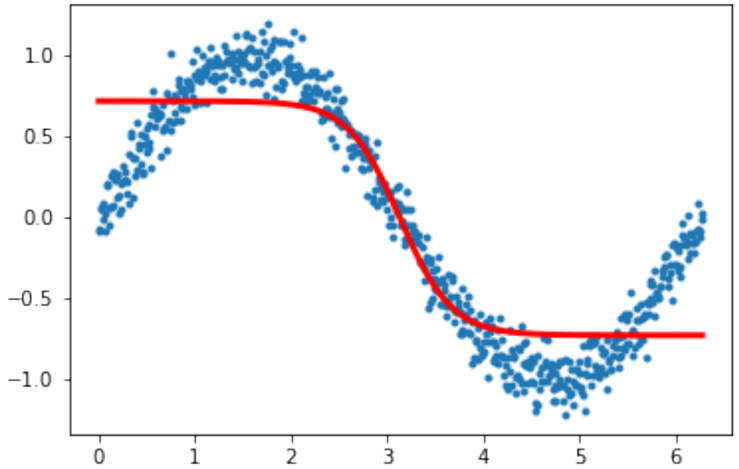
$$\frac{\partial C}{\partial \theta^l} = \sum_u \frac{\partial C}{\partial z_u^{l+1}} \frac{\partial z_u^{l+1}}{\partial \theta^l} = \sum_u \delta_u^{l+1} \frac{\partial z_u^{l+1}}{\partial \theta^l}$$

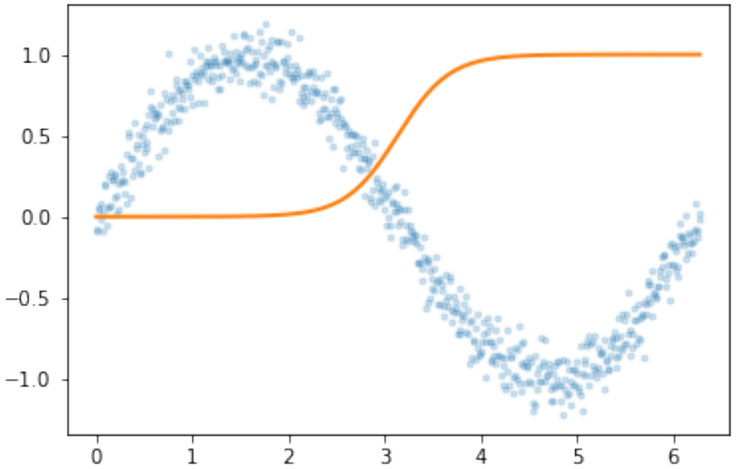
(backward pass is thus also used to fill the `variable.grad` parts of parameters in pytorch)

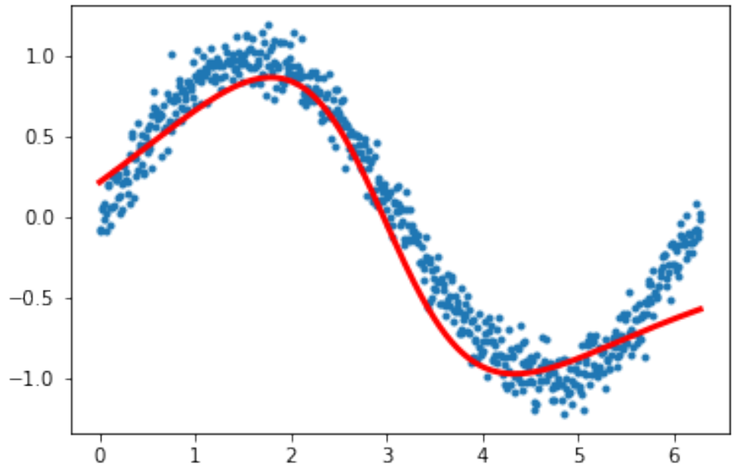
Universal Approximation

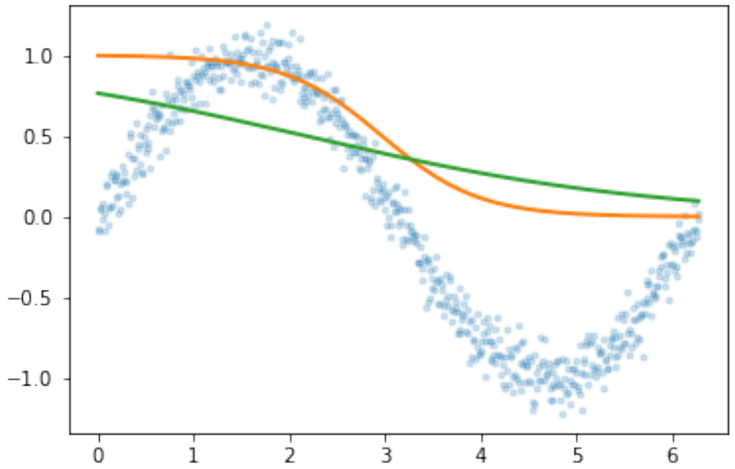
- any one hidden layer net can approximate any continuous function with finite support, with appropriate choice of nonlinearity
- under appropriate conditions, all of sigmoid, tanh, RELU can work
- but may need lots of units
- and will learn the function it thinks the data has, not what you think

One hidden, 1 vs 2 neurons

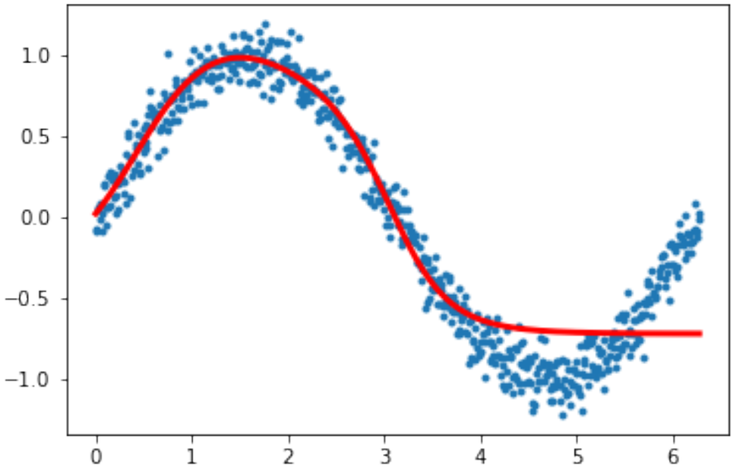


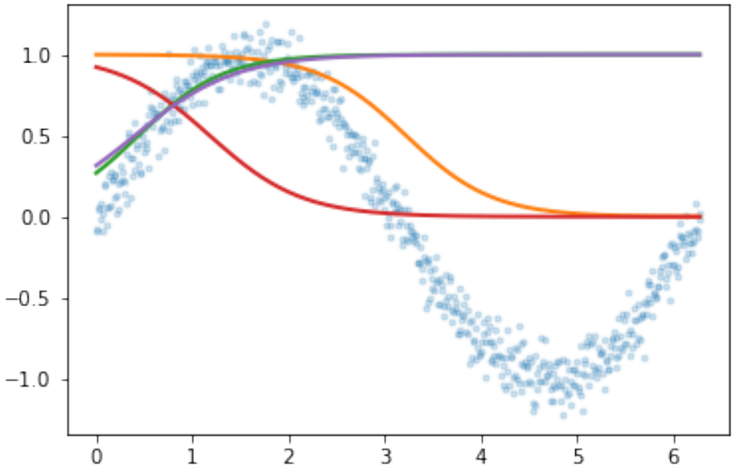


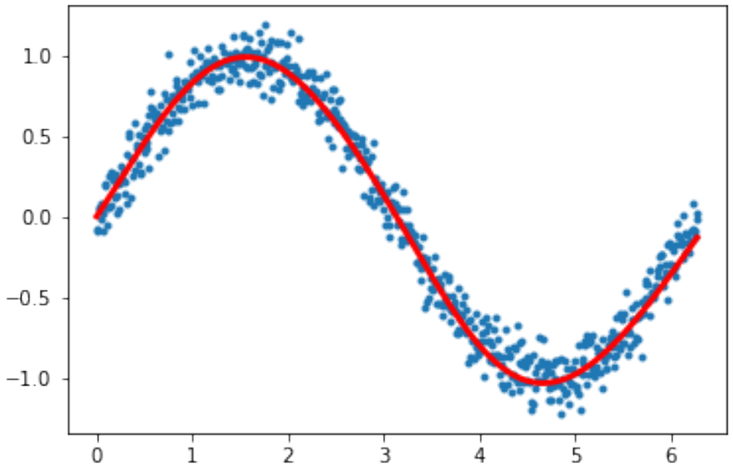


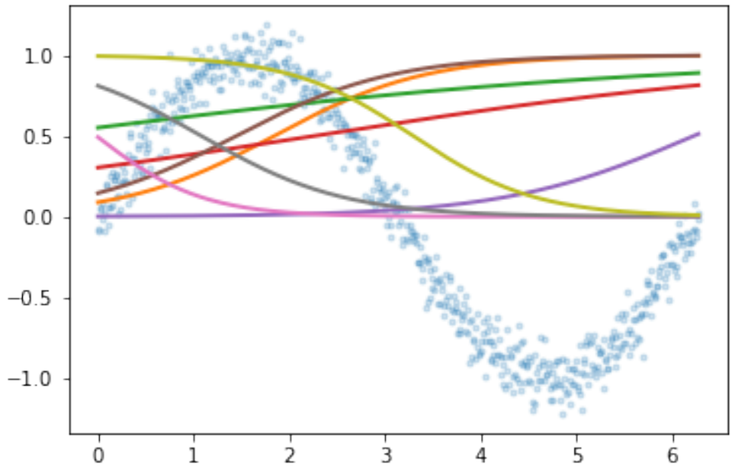


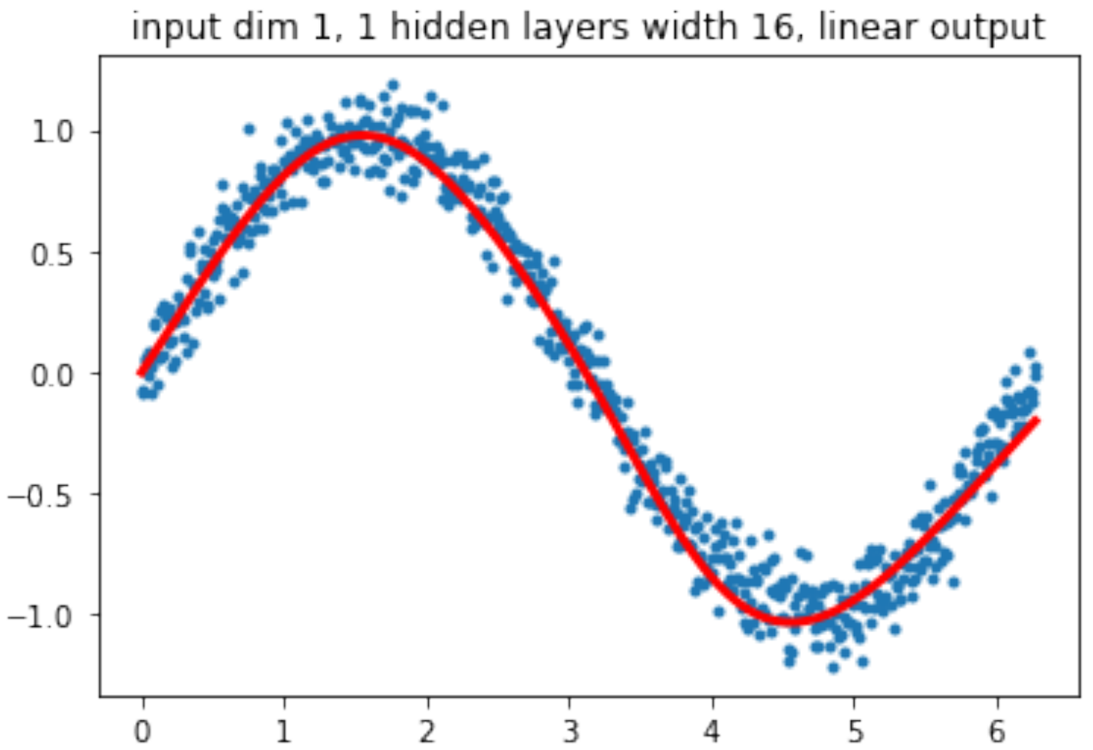
Two hidden, 4 vs 8 neurons

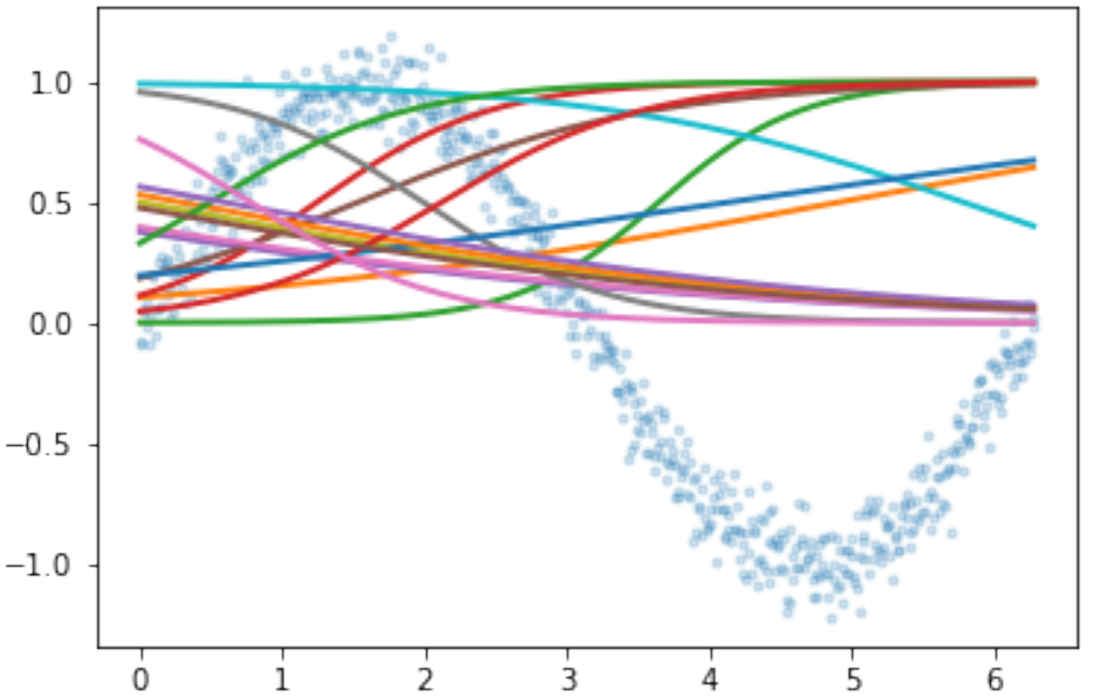




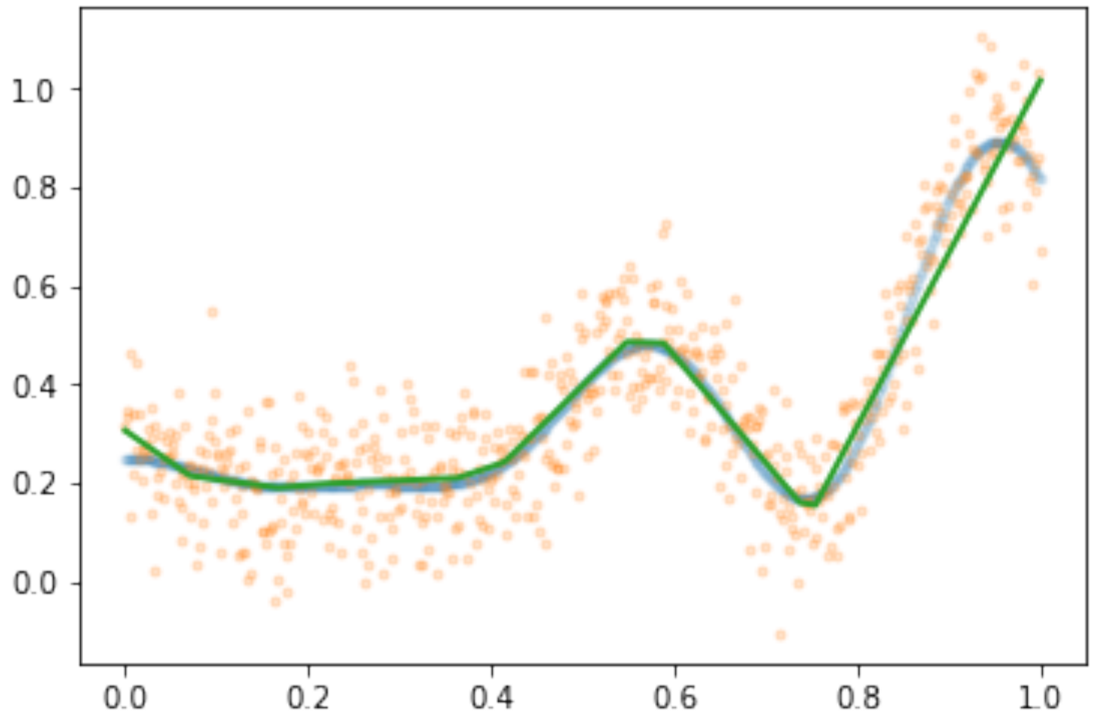


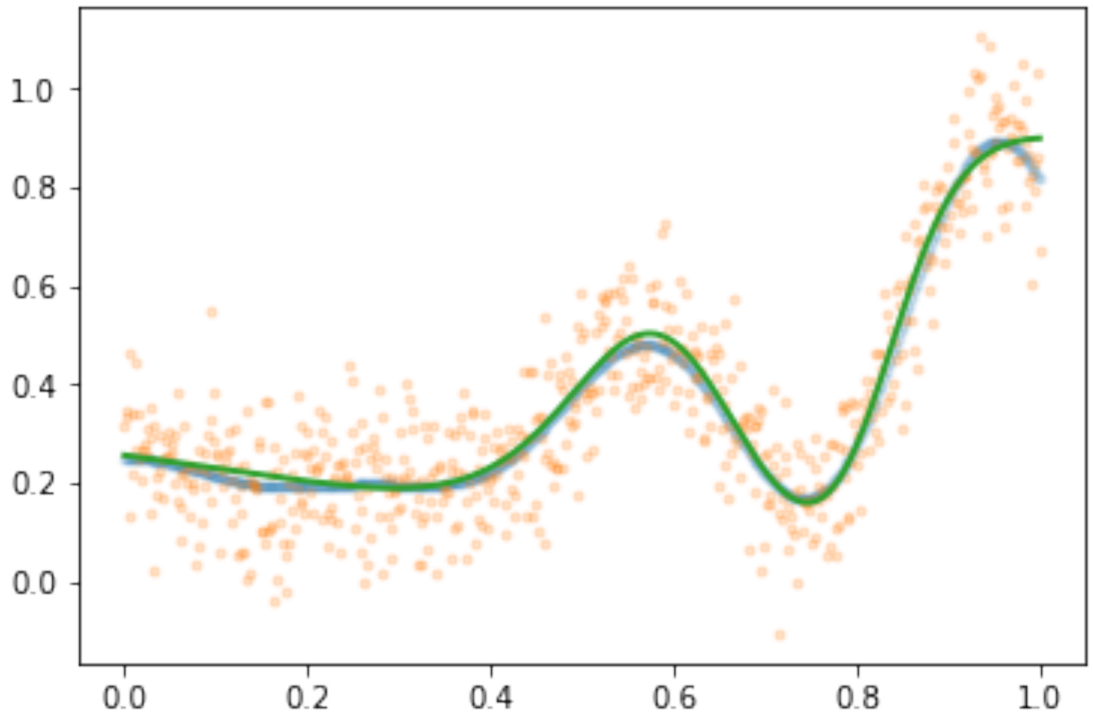



input dim 1, 1 hidden layers width 16, linear output
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6


1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

Relu (80, 1 layer) and tanh(40, 2 layer)

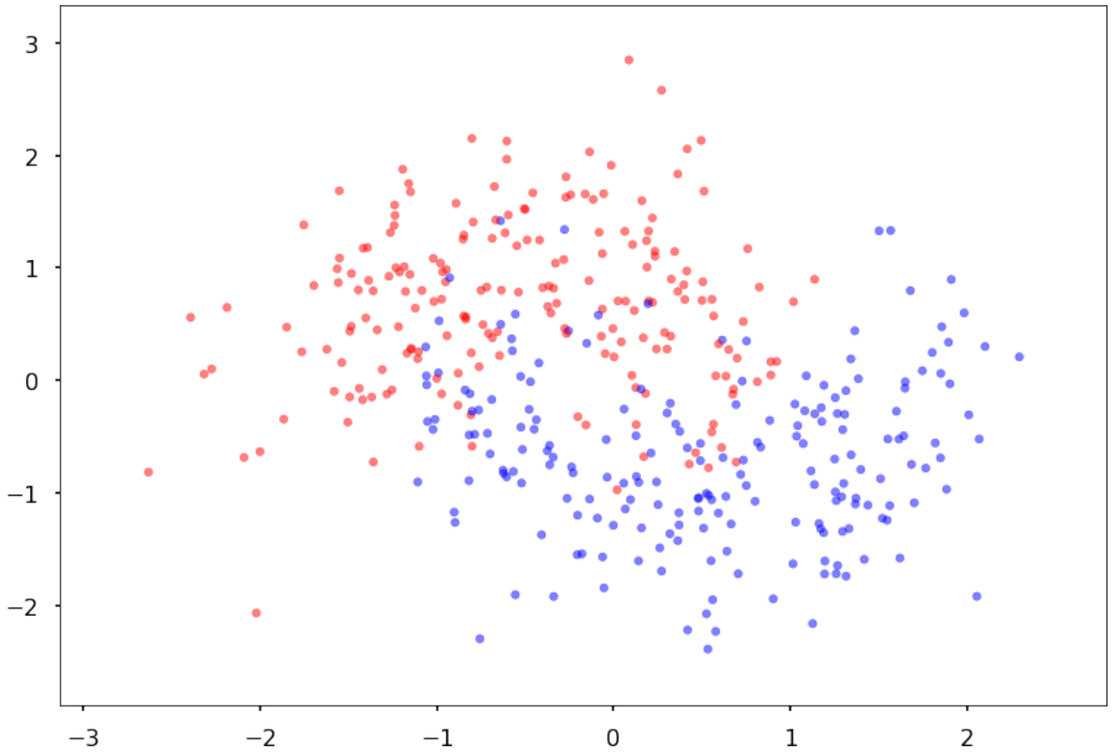


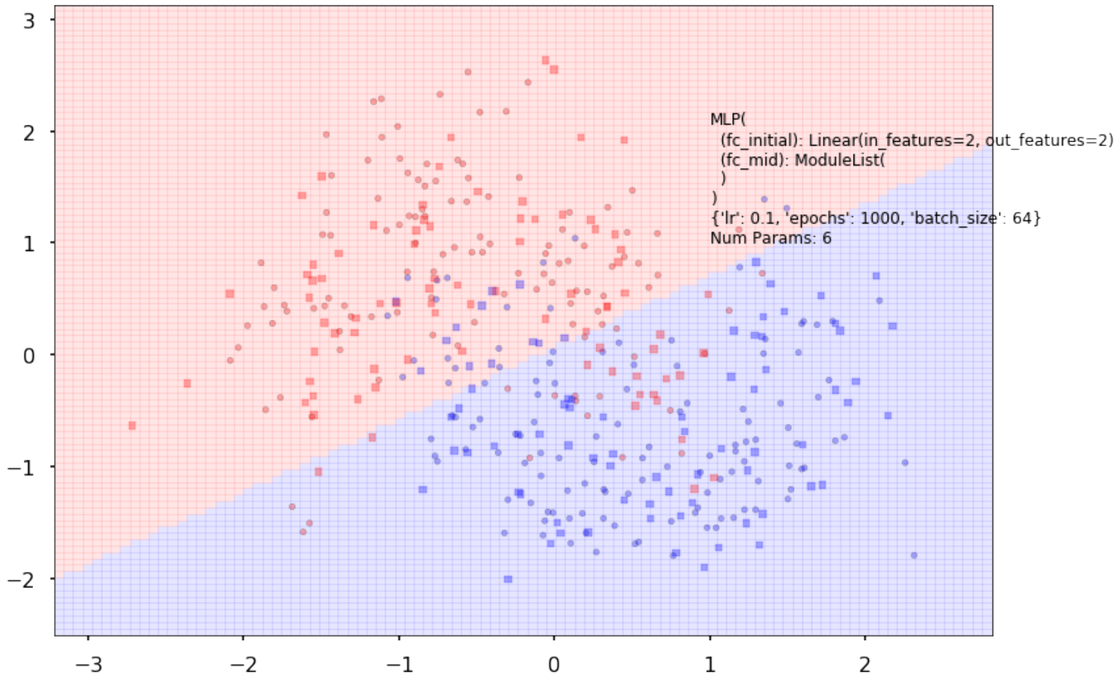


Some rules of thumb

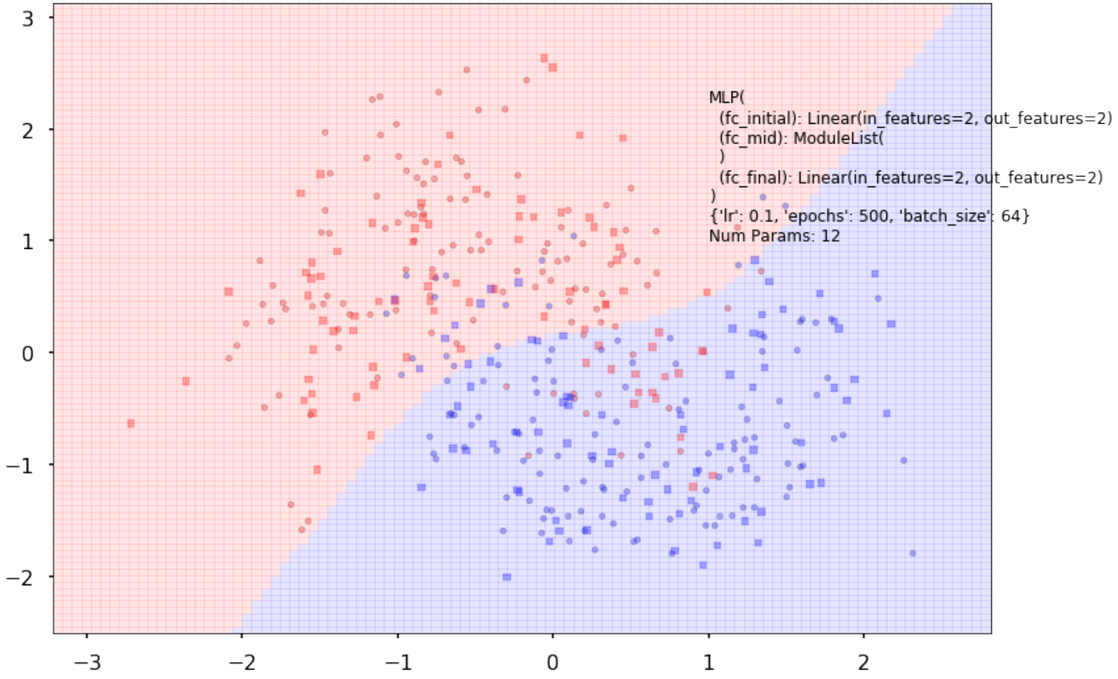
- relu and tanh are better non-linearities in hidden layers
- normalize your data by squashing to unit interval or standardizing so that no feature gets more important than the other
- outputs from non-linearity at any intermediate layer may need normalizing

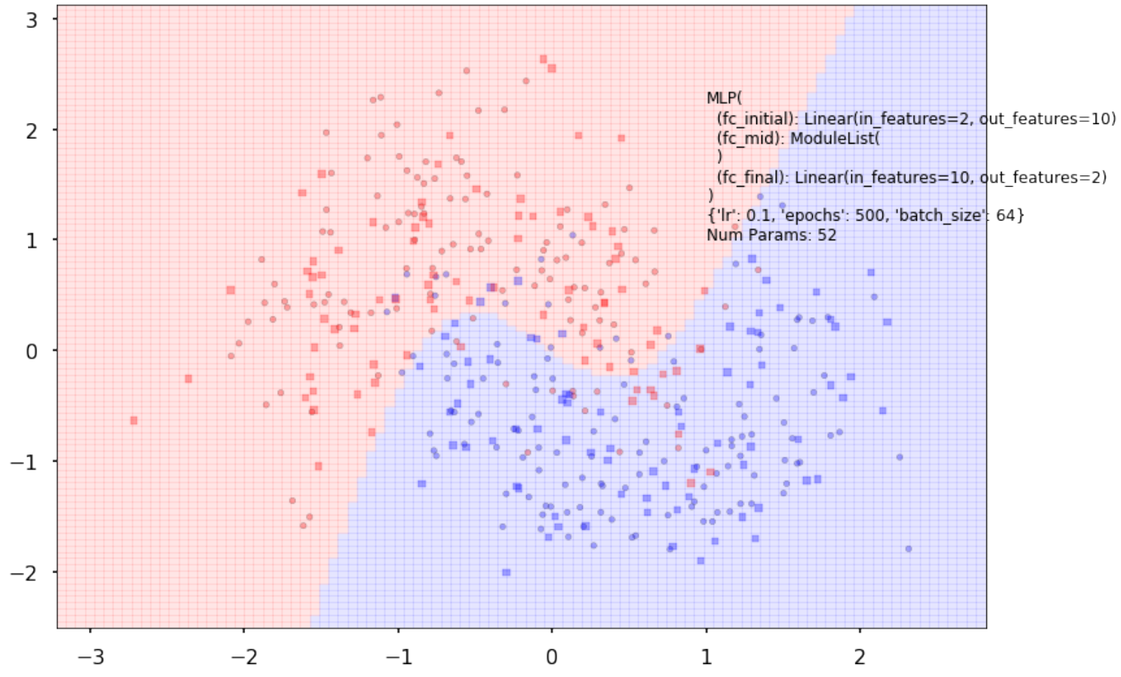
Half moon dataset (artificially GENERATED)



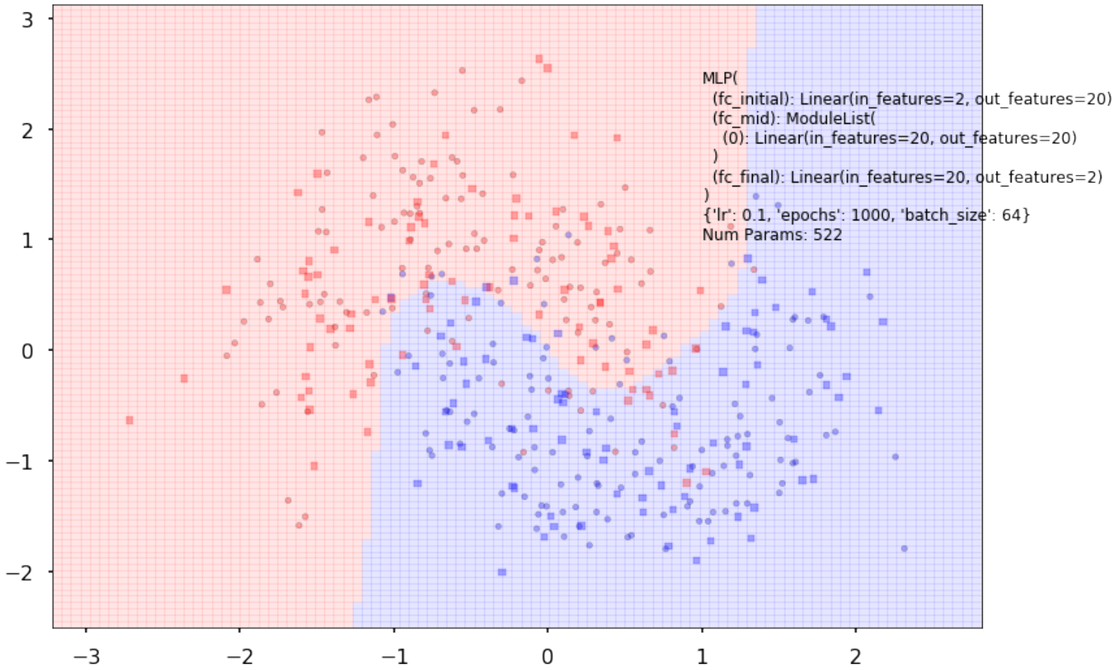


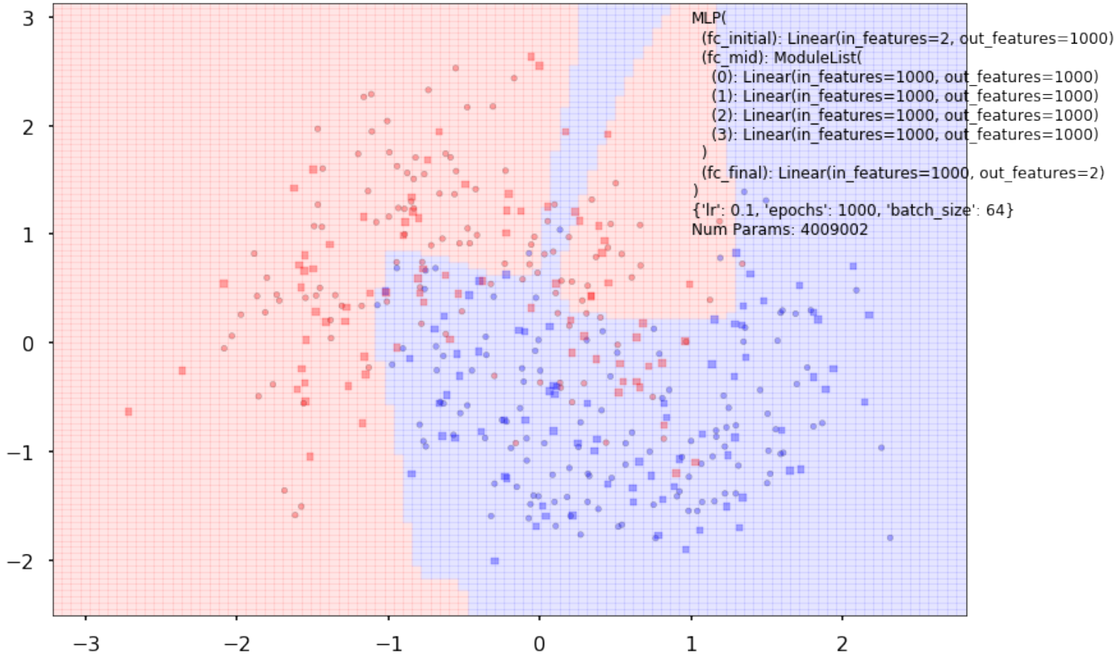
1 layer, 2 vs 10 neurons





2 layers, 20 neurons vs 5 layers, 1000 neurons





Why does deep learning work?

1. Automatic differentiation

2. GPU

3. Learning Representations

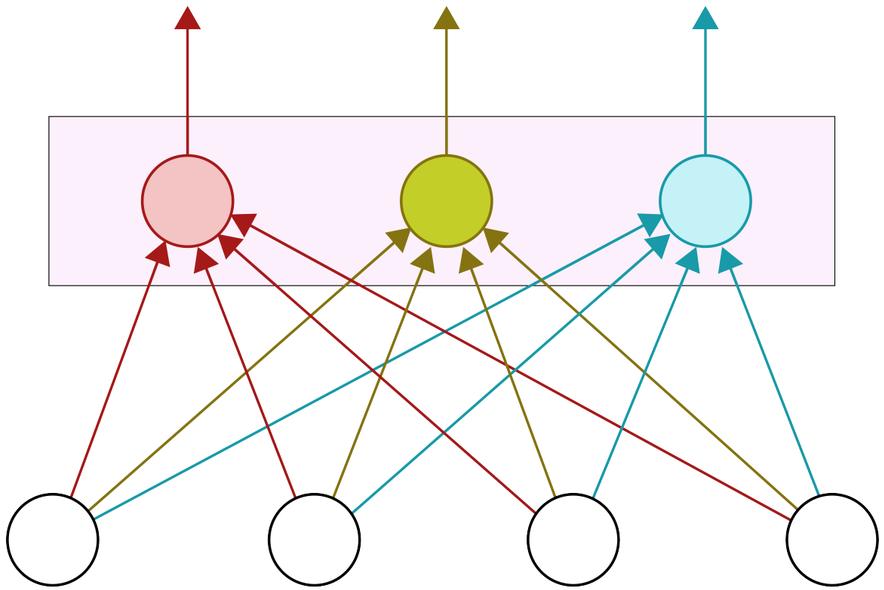
MLP's constructed "recursive functions":

$s(w_n . z_n + b_n)$ where $z_n = s(w_{n-1} . z_{n-1} + b_{n-1})$ and so on.

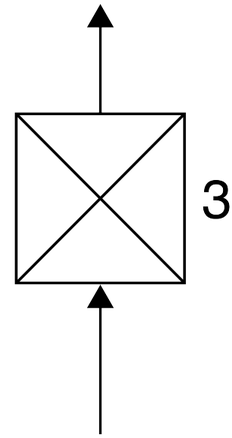
Deep Learning Components

Fully Connected layers

- also called dense layers
- these are the layers in MLPs
- we typically use them after convolutional layers to feed to a softmax to get probabilities



(a)

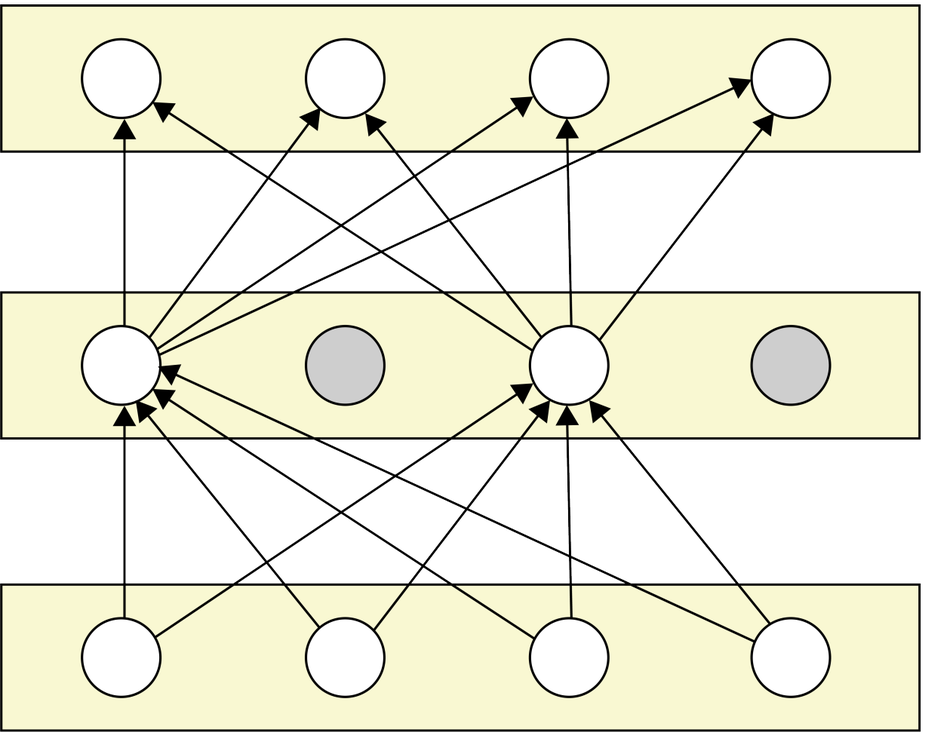


(b)

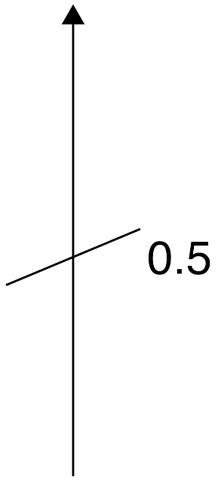
1-1 layers

- nonlinearities and softmax
- we dont really consider them layers ..you can think of them bolted to previous nodes in each layer.
- they are nor fully-connected..indeed theay are 1-1

Dropout



(a)



(b)

- not really a layer, but a technique
- randomly sever connections to and from some fraction of nodes in "previous" layer
- a regularization method, it prevents overfitting by not allowing specific nodes to specialize to say, for example "cat eye detection" at the expense of other things. By severing, other neurons are forced to step in
- only during training, at prediction time the connections are restored