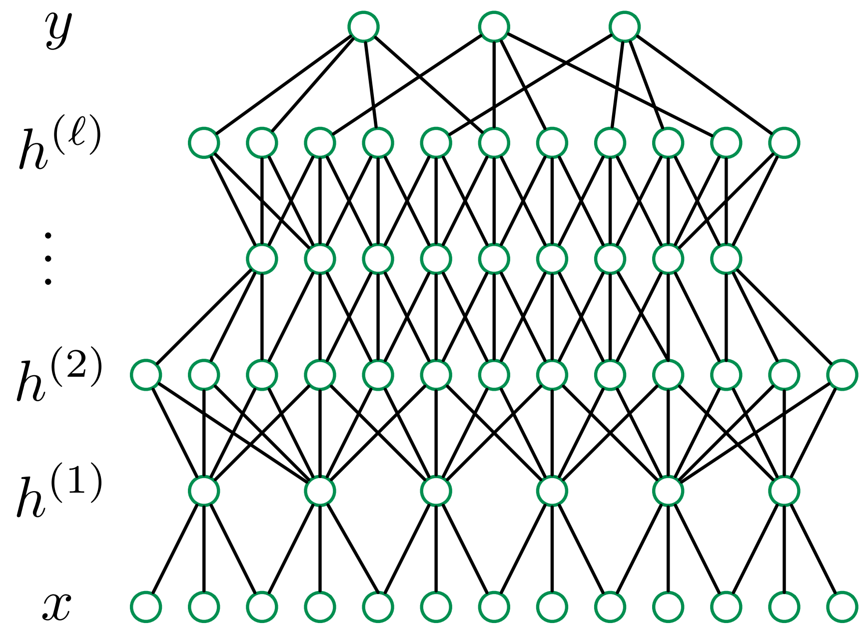
Feedforward neural nets

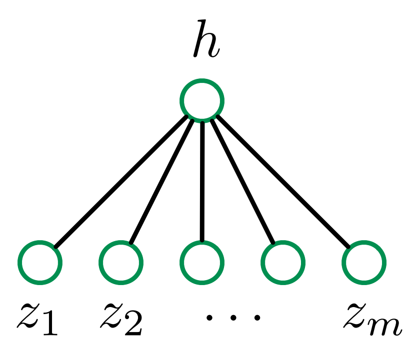
Outline

1. Architecture
2. Expressivity
3. Learning

The architecture



The value at a hidden unit



How is h computed from $z_1, \ldots, z_m$?

- $h = \sigma(w_1 z_1 + w_2 z_2 + \cdots + w_m z_m + b)$
- $\sigma(\cdot)$ is a nonlinear **activation function**, e.g. "rectified linear"

$$\sigma(u) = \begin{cases} u & \text{if } u \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Common activation functions

- Threshold function or Heaviside step function

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- Sigmoid

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

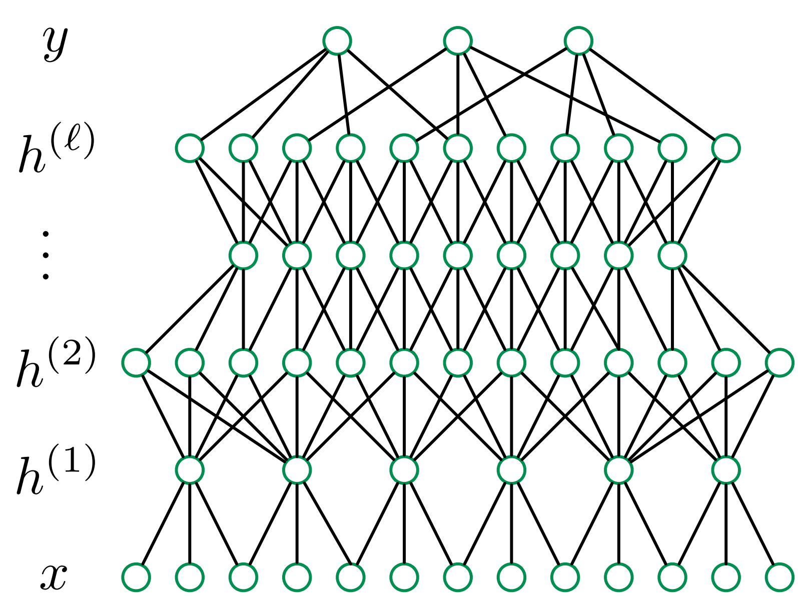
- Hyperbolic tangent

$$\sigma(z) = \tanh(z)$$

- ReLU (rectified linear unit)

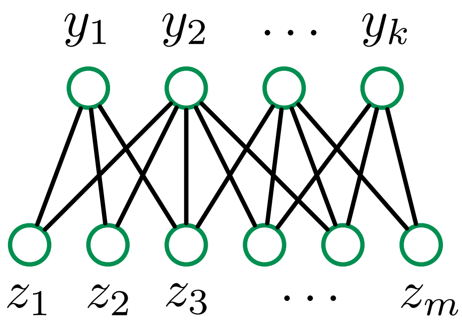
$$\sigma(z) = \max(0, z)$$

Why do we need nonlinear activation functions?



The output layer

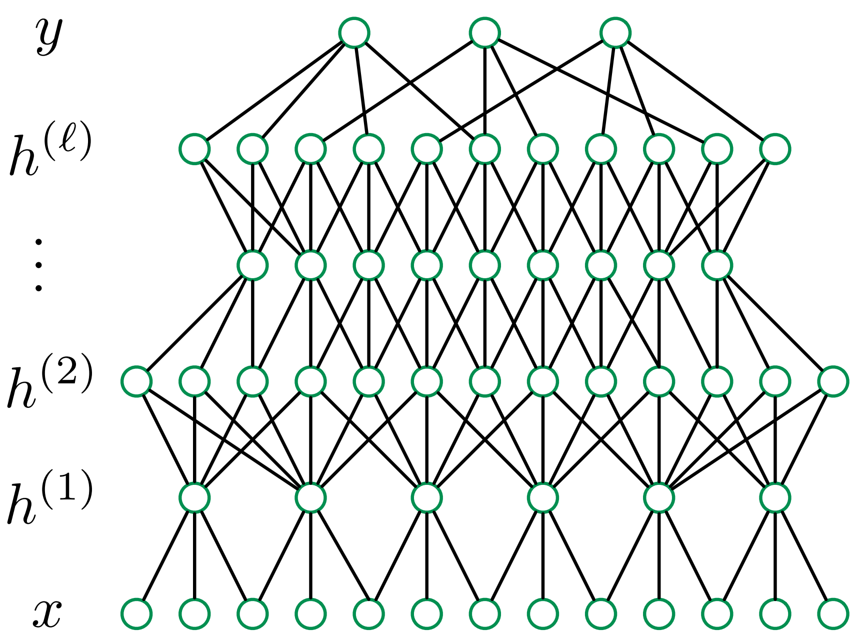
Classification with k labels: want k probabilities summing to 1.



- $y_1, \ldots, y_k$ are linear functions of the parent nodes z_i.
- Get probabilities using **softmax**:

$$\Pr(\text{label } j) = \frac{e^{y_j}}{e^{y_1} + \cdots + e^{y_k}}.$$

The complexity



Approximation capability

Let $f : \mathbb{R}^d \to \mathbb{R}$ be any continuous function.
There is a neural net with a single hidden layer that approximates f arbitrarily well.

- The hidden layer may need a lot of nodes.
- For certain classes of functions:
  - Either: one hidden layer of enormous size
  - Or: multiple hidden layers of moderate size

Learning a net: the loss function

Classification problem with k labels.

- Parameters of entire net: W
- For any input x, net computes probabilities of labels:

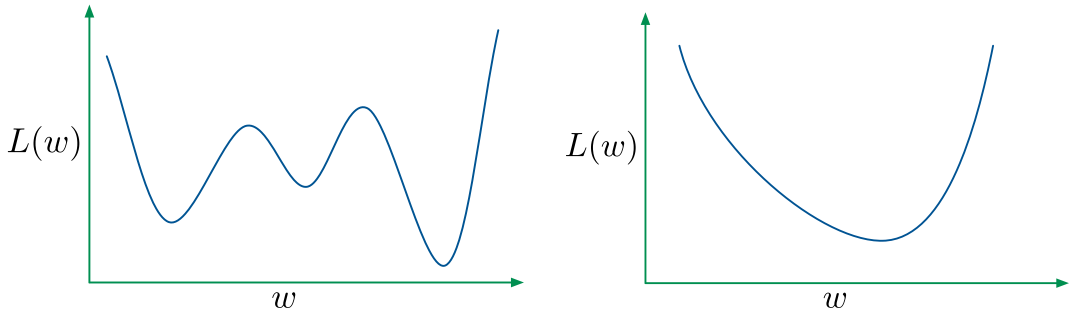
$$\Pr_W(\text{label} = j | x)$$

- Given data set $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})$, loss function:

$$L(W) = -\sum_{i=1}^{n} \ln \Pr_W(y^{(i)} | x^{(i)})$$

(also called **cross-entropy**).

Nature of the loss function



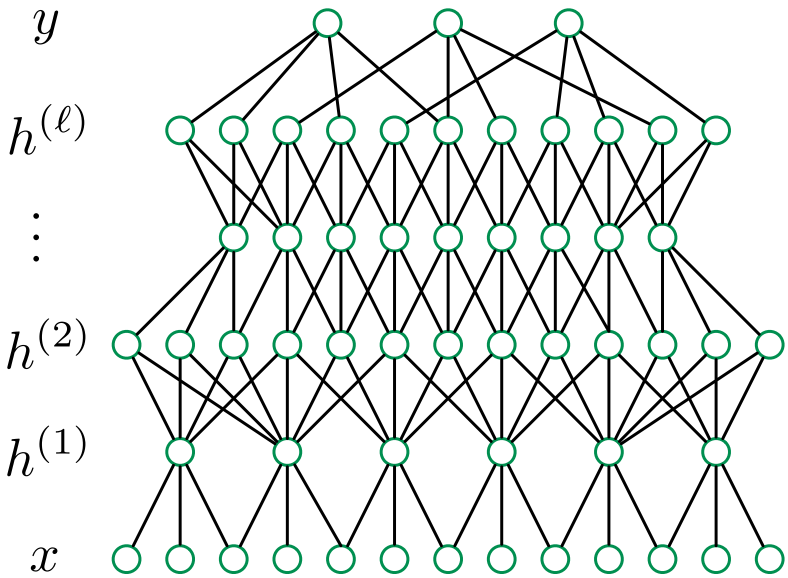
Variants of gradient descent

Initialize W and then repeatedly update.

1. Gradient descent
   Each update involves the entire training set.
2. Stochastic gradient descent
   Each update involves a single data point.
3. Mini-batch stochastic gradient descent
   Each update involves a modest, fixed number of data points.

Derivative of the loss function

Update for a specific parameter: derivative of loss function wrt that parameter.

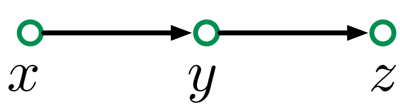


Chain rule

1. Suppose $h(x) = g(f(x))$, where $x \in \mathbb{R}$ and $f, g : \mathbb{R} \to \mathbb{R}$.

   Then: $h'(x) = g'(f(x))\, f'(x)$

2. Suppose z is a function of y, which is a function of x.



Then:

$$\frac{dz}{dx} = \frac{dz}{dy}\,\frac{dy}{dx}$$

A single chain of nodes

A neural net with one node per hidden layer:

$$x = h_0 \longrightarrow h_1 \longrightarrow h_2 \longrightarrow h_3 \longrightarrow \cdots \longrightarrow h_\ell$$

For a specific input x,

- $h_i = \sigma(w_i h_{i-1} + b_i)$
- The loss L can be gleaned from h_ℓ

To compute dL/dw_i we just need dL/dh_i:

$$\frac{dL}{dw_i} = \frac{dL}{dh_i}\frac{dh_i}{dw_i} = \frac{dL}{dh_i}\,\sigma'(w_i h_{i-1} + b_i)\,h_{i-1}$$

Backpropagation

- On a single forward pass, compute all the h_i.
- On a single backward pass, compute $dL/dh_\ell, \ldots, dL/dh_1$

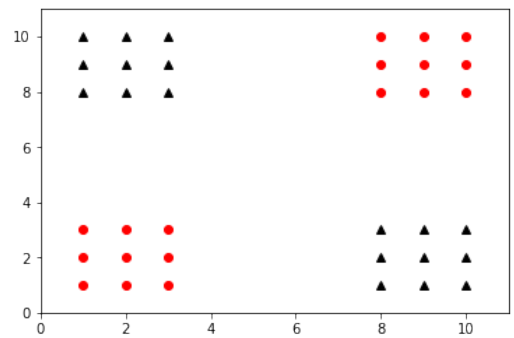
$$x = h_0 \longrightarrow h_1 \longrightarrow h_2 \longrightarrow h_3 \longrightarrow \cdots \longrightarrow h_\ell$$

From $h_{i+1} = \sigma(w_{i+1} h_i + b_{i+1})$, we have

$$\frac{dL}{dh_i} = \frac{dL}{dh_{i+1}}\frac{dh_{i+1}}{dh_i} = \frac{dL}{dh_{i+1}}\,\sigma'(w_{i+1} h_i + b_{i+1})\,w_{i+1}$$

Two-dimensional examples

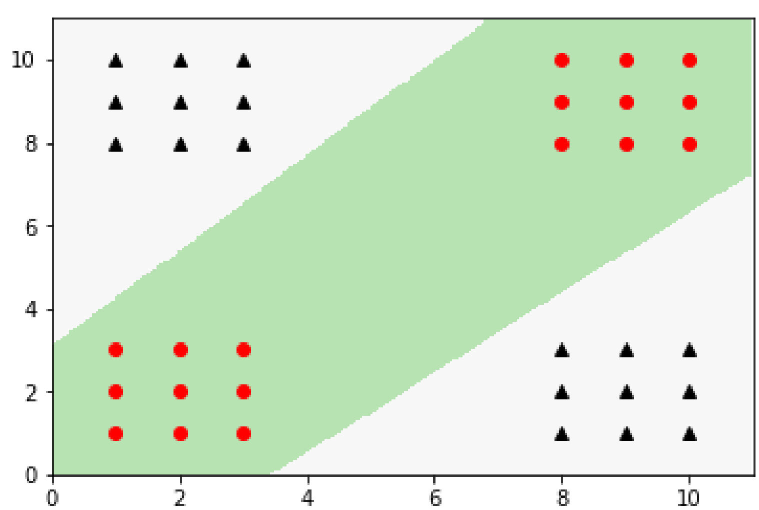
What kind of net to use for this data?



- Input layer: 2 nodes
- One hidden layer: H nodes
- Output layer: 1 node
- Input $\rightarrow$ hidden: linear functions, ReLU activation
- Hidden $\rightarrow$ output: linear function, sigmoid activation

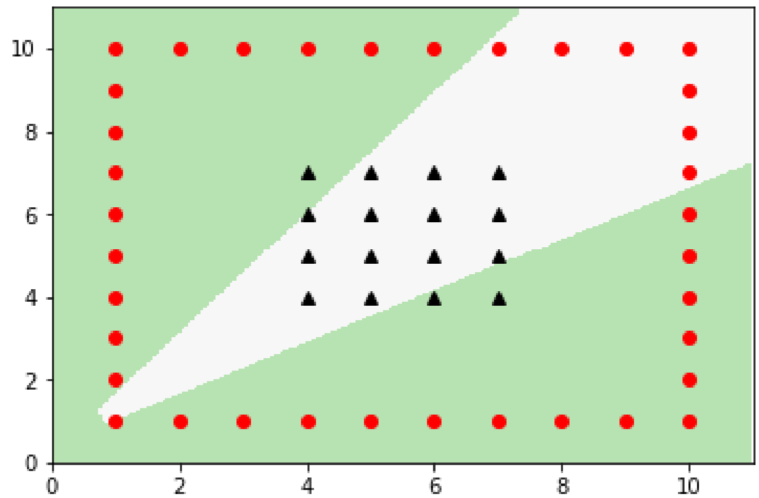
Example 1

$H = 2$



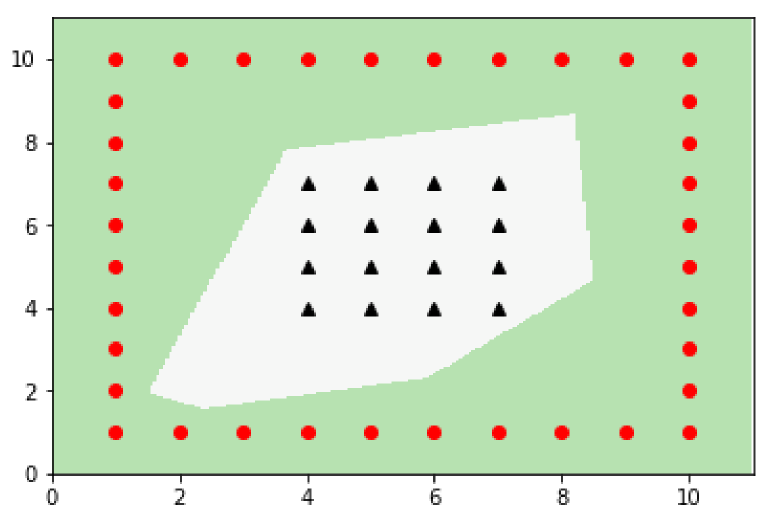
Example 2

$H = 4$



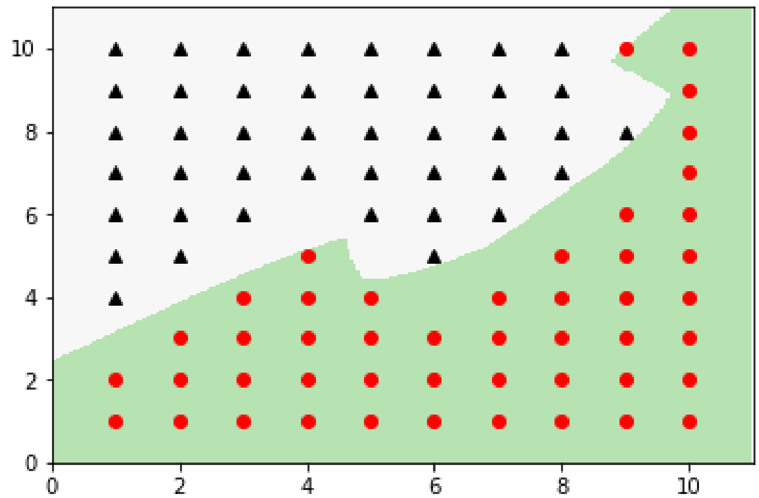
Example 2

$H = 8$: overparametrized



Example 3

$H = 64$



PyTorch snippet

Declaring and initializing the network:

```
d, H = 2, 8
model = torch.nn.Sequential(
    torch.nn.Linear(d, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, 1),
    torch.nn.Sigmoid())
lossfn = torch.nn.BCELoss()
```

A gradient step:

```
ypred = model(x)
loss = lossfn(ypred, y)
model.zero_grad()
loss.backward()
with torch.no_grad():
    for param in model.parameters():
        param -= eta * param.grad
```