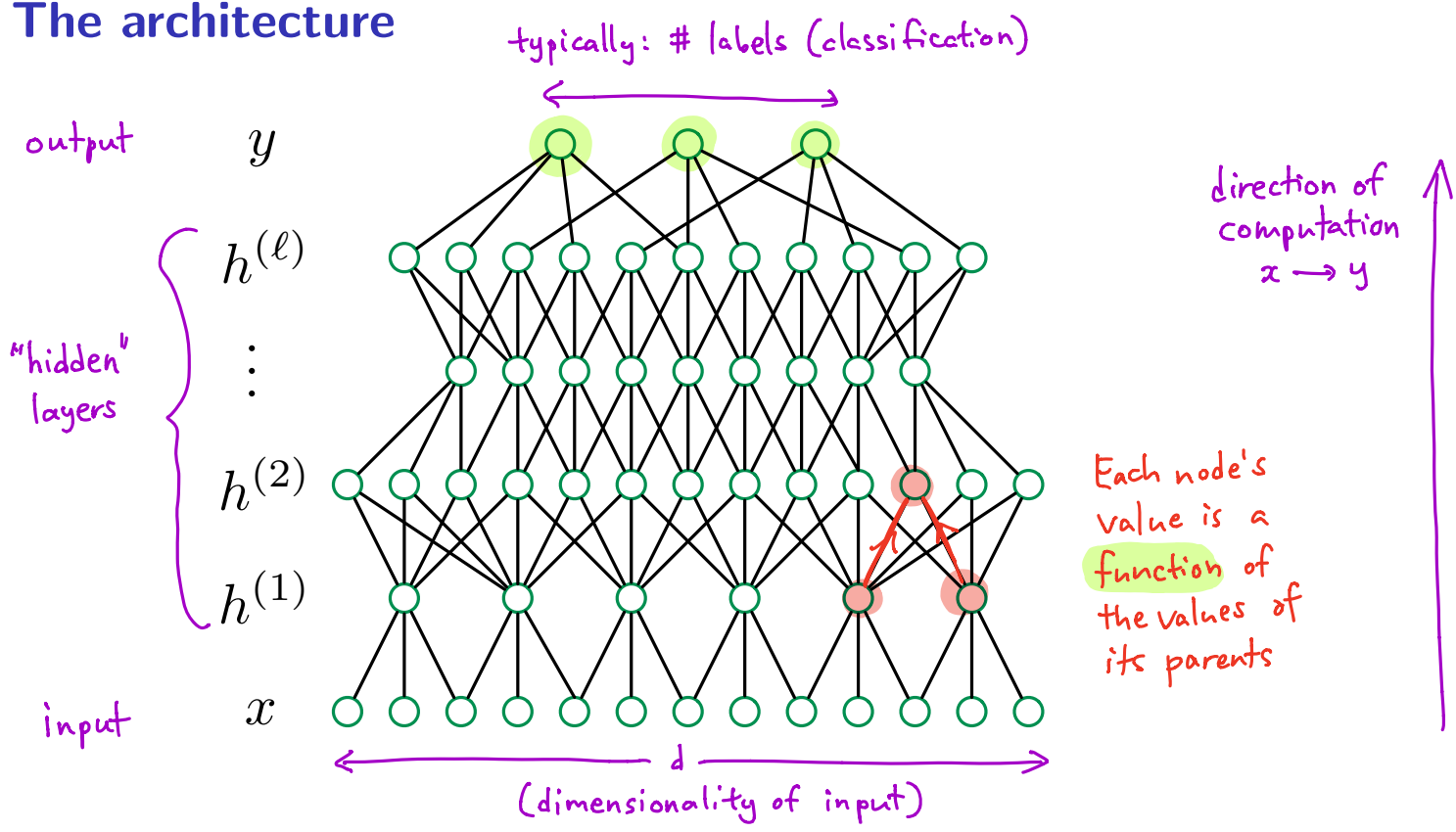
Feedforward neural nets

DSE 220

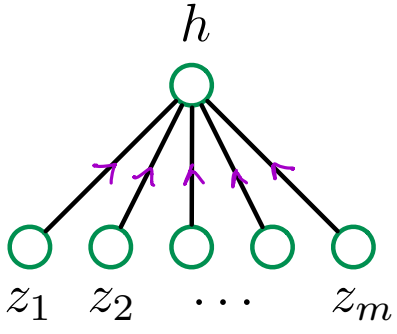
Outline

1. Architecture
2. Learning
3. Bells and whistles

The architecture

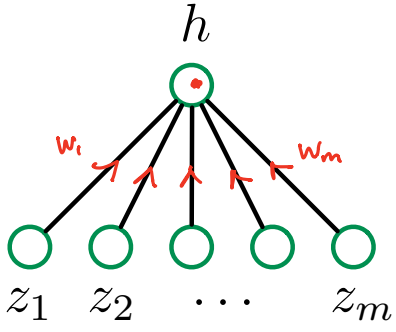


The value at a hidden unit

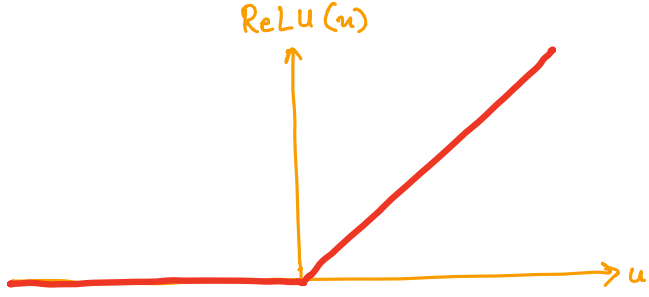


How is h computed from $z_1, \ldots, z_m$?

The value at a hidden unit



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



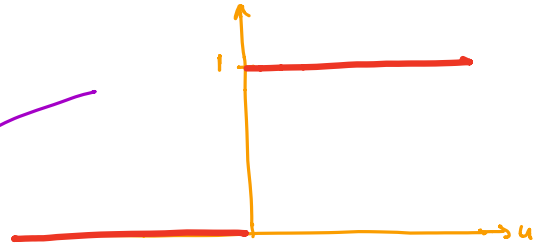
- $h = \sigma(w_1 z_1 + w_2 z_2 + \cdots + w_m z_m + b)$
- $\sigma(\cdot)$ is a nonlinear **activation function**, e.g. "rectified linear"

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

Common activation functions

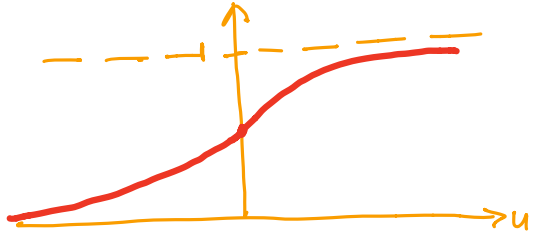
- Threshold function or Heaviside step function

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



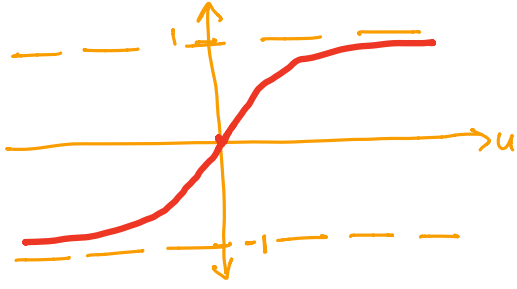
- Sigmoid

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



- Hyperbolic tangent

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

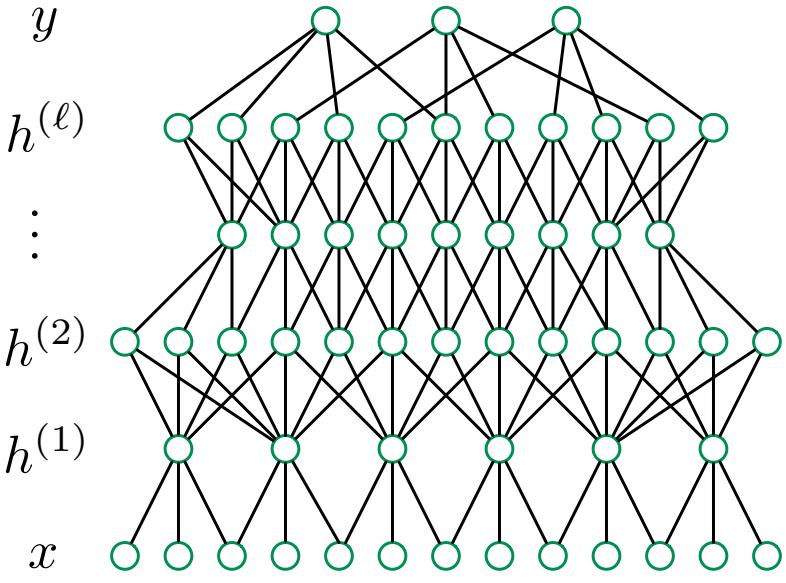


- ReLU (rectified linear unit)

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

Why do we need nonlinear activation functions?

Suppose all the mappings were linear



$y = w^T h^{(\ell)}$

$h^{(3)} = W_3 h^{(2)}$

$h^{(2)} = W_2 h^{(1)}$

$h^{(1)} = W_1 x$

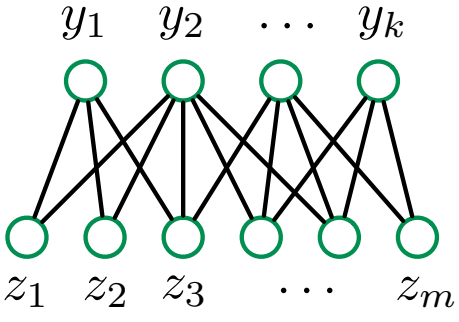
Then $y = (w^T W_\ell W_{\ell-1} \cdots W_1) x$, ie. y is a linear function of x.

In this case, one layer would suffice.

The output layer

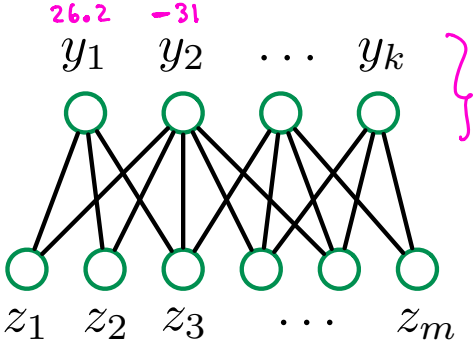
Output layer depends on whether we're doing classification or regression. Let's do classification.

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



The output layer

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



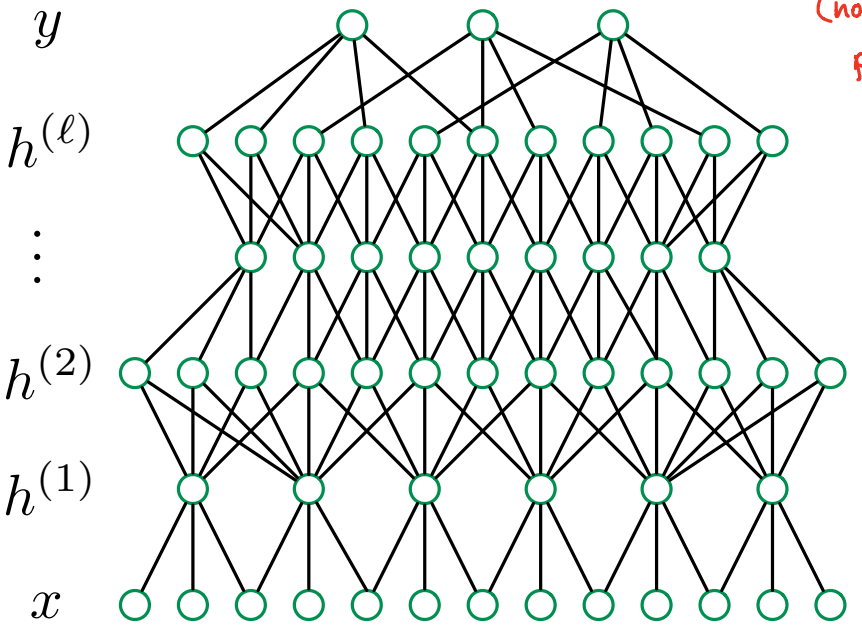
- $y_1, \ldots, y_k$ are linear functions of the parent nodes z_i.
- Get probabilities using **softmax**:

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

This last layer is exactly multiclass logistic regression.

The complexity

Two implications: ① Very expressive
② Lots of training data needed (not to mention computational power)



weights in a single layer $\sim 10^6$ (fully connected)

say hidden layers also have 1000 units

Vast # of parameters!

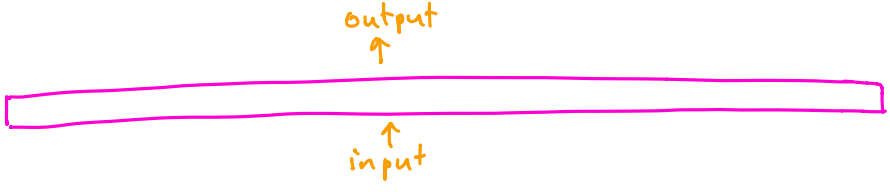
Approximation capability

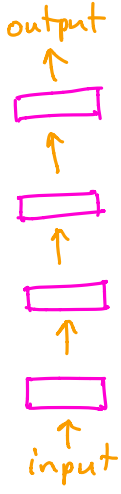
Let $f : \mathbb{R}^d \to \mathbb{R}$ be any continuous function.
There is a neural net with one hidden layer that approximates f arbitrarily well.

Approximation capability

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

- The hidden layer may need a lot of nodes.
- The benefit of depth: for certain classes of functions, you need:
  - Either: one hidden layer of enormous size
  - Or: multiple hidden layers of moderate size





Outline

1. Architecture
2. Learning
3. Bells and whistles

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:

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

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:

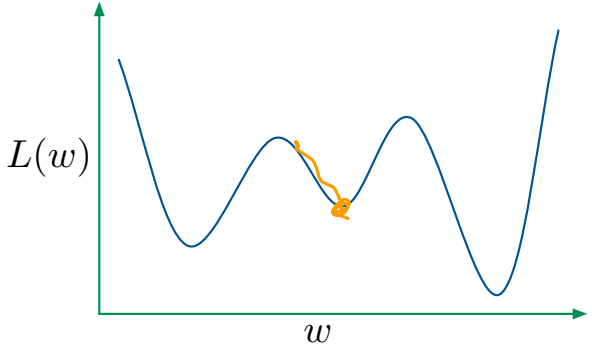
(maximum likelihood)

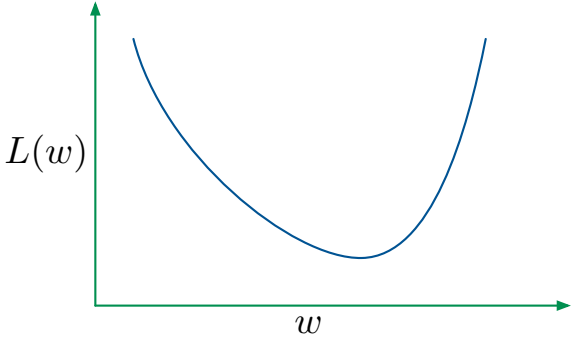
$$L(W) = -\sum_{i=1}^{n} \ln \Pr_W(y^{(i)}|x^{(i)})$$

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

If there are no hidden layers, this is logistic regression.

Nature of the loss function





This is the situation we're in (actually much worse!)

Variants of gradient descent

Initialize W and then repeatedly update.

1. Gradient descent
   Each update involves the entire training set.

   wrt W
   derivative of the (loss on the entire data set)

2. Stochastic gradient descent
   Each update involves a single data point.

   wrt W
   derivative of (loss on a single point)

3. Mini-batch stochastic gradient descent
   Each update involves a modest, fixed number of data points.

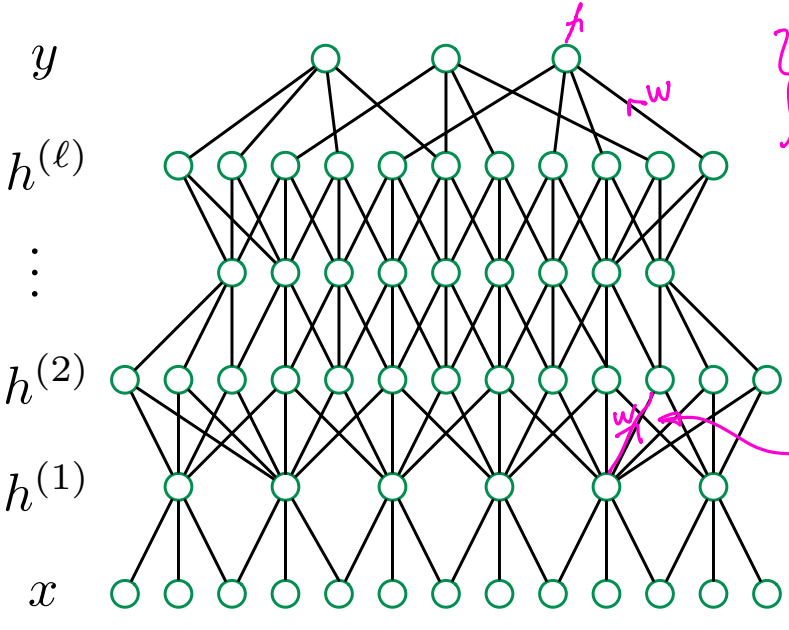
   derivative of (loss on a batch of points)

Derivative of the loss function

For a given x, we'll predict $\hat{y}$ and incur loss $\ell(\hat{y}, y)$ [y = true label]

To update any param w: we need $\frac{d\ell}{dw}$.

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



Not hard to compute $\frac{d\ell}{dw}$ when w is in the last layer... essentially logistic regression

But how do we compute $\frac{d\ell}{dw}$ for params w buried inside the net?

Backpropagation does this.

Chain rule

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

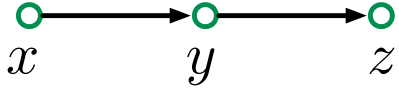
   Then: $h'(x) = g'(f(x))\, f'(x)$

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.

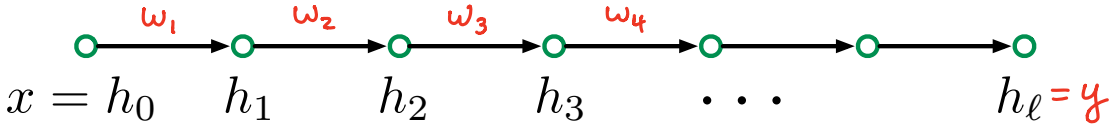


Then:

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

A single chain of nodes

A neural net with one node per hidden layer:



$h_3 = \sigma(w_3 h_2 + b_3)$

For a specific input x,

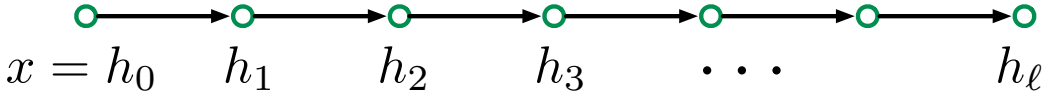
- $h_i = \sigma(w_i h_{i-1} + b_i)$
- The loss L can be gleaned from h_ℓ

Want $\frac{dL}{dw_i}$

$w_i \leftarrow w_i - \eta \frac{dL}{dw_i}$ (Gradient descent)

A single chain of nodes

A neural net with one node per hidden layer:



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:

enough to get this

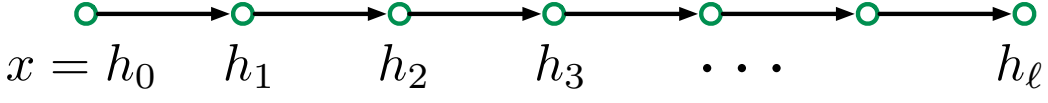
$$\frac{dL}{dw_i} = \frac{dL}{dh_i}\frac{dh_i}{dw_i} = \frac{dL}{dh_i}\sigma'(w_i h_{i-1} + b_i)\, h_{i-1}$$

we need this

we have these

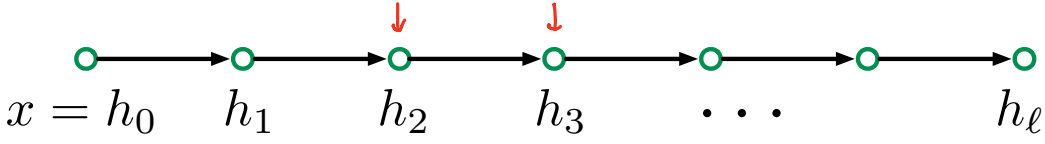
Backpropagation

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



Backpropagation

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



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

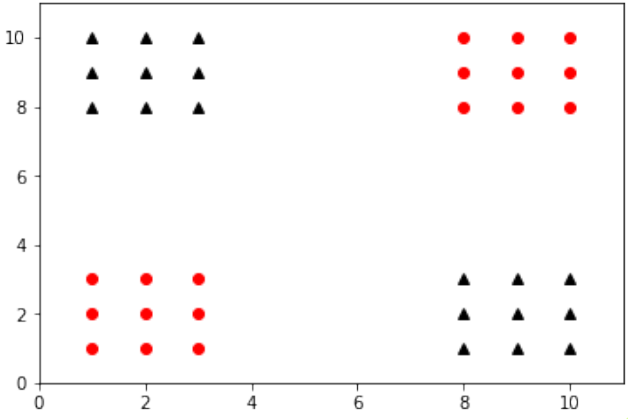
need this

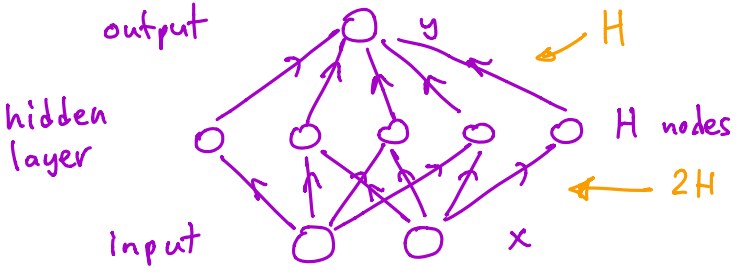
$$\frac{dL}{dh_i} = \frac{dL}{dh_{i+1}}\frac{dh_{i+1}}{dh_i} = \frac{dL}{dh_{i+1}}\sigma'(w_{i+1}h_i + b_{i+1})\,w_{i+1}$$

we know these

Given $\frac{dL}{dh_{i+1}}$, we can easily figure out $\frac{dL}{dh_i}$ ← backward pass

Two-dimensional examples





What kind of net to use for this data?

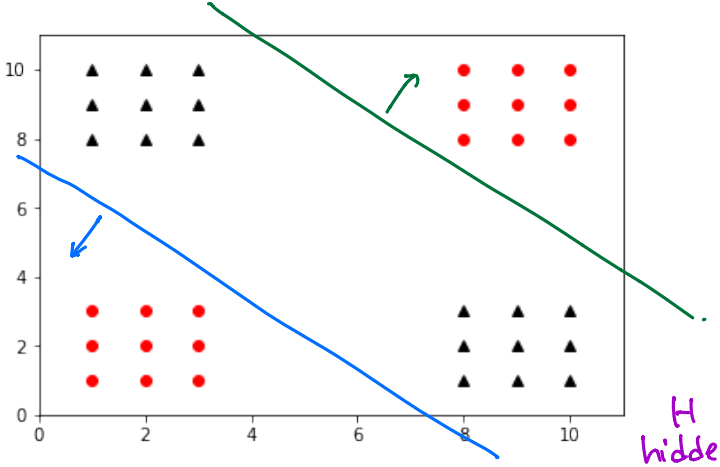
① Do we need a hidden layer?

Yes. Because the data is not linearly separable.

② Is one hidden layer enough?

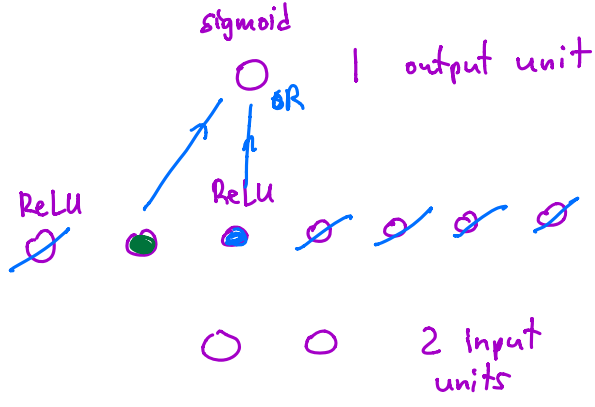
Yes. One hidden layer is always enough. But training might be quicker/easier if we have more layers.

Two-dimensional examples



H hidden units

What kind of net to use for this data?

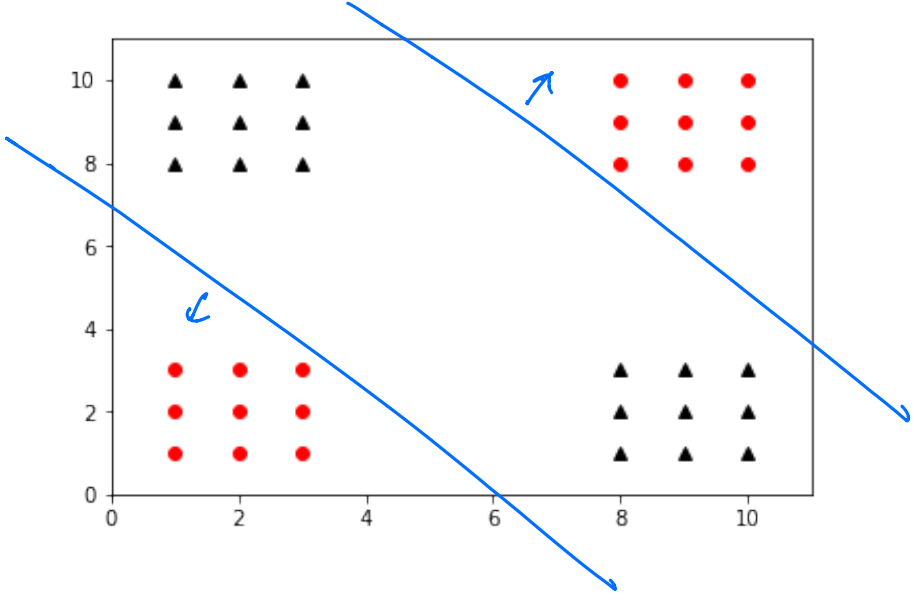


- Input layer: 2 nodes
- One hidden layer: H nodes
- Output layer: 1 node
- Input $\rightarrow$ hidden: linear functions, ReLU activation
- Hidden $\rightarrow$ output: linear function, sigmoid activation

What H to choose?

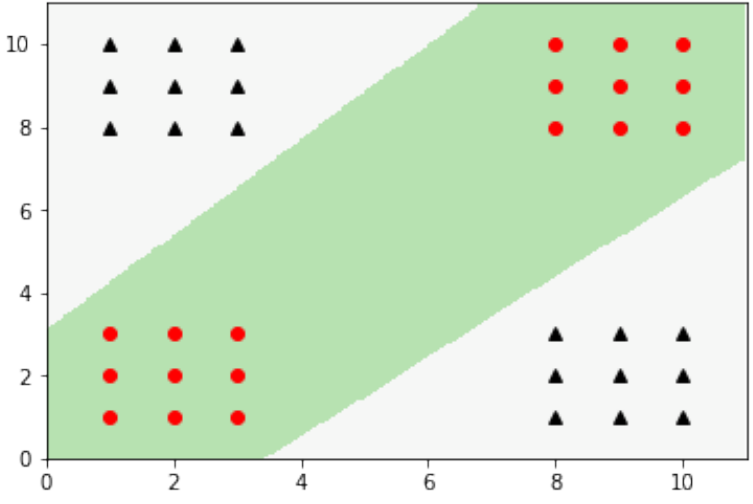
Example 1

How many hidden units should we use?



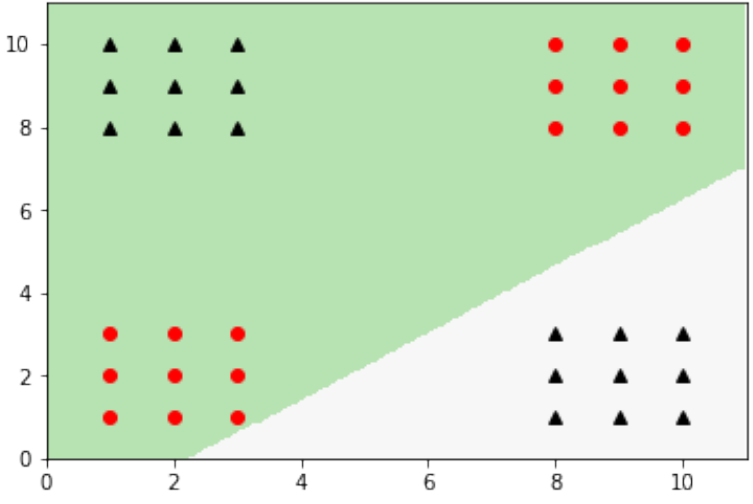
Example 1

$H = 2$



Example 1

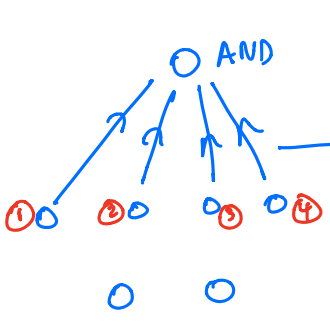
$H = 2$

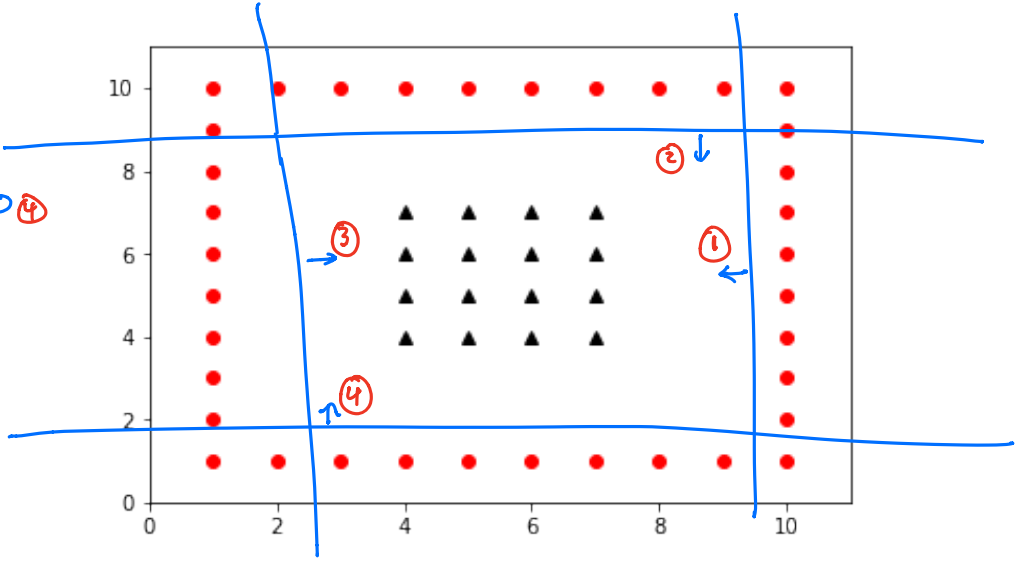


Example 2

How many hidden units should we use?

4 should be enough.

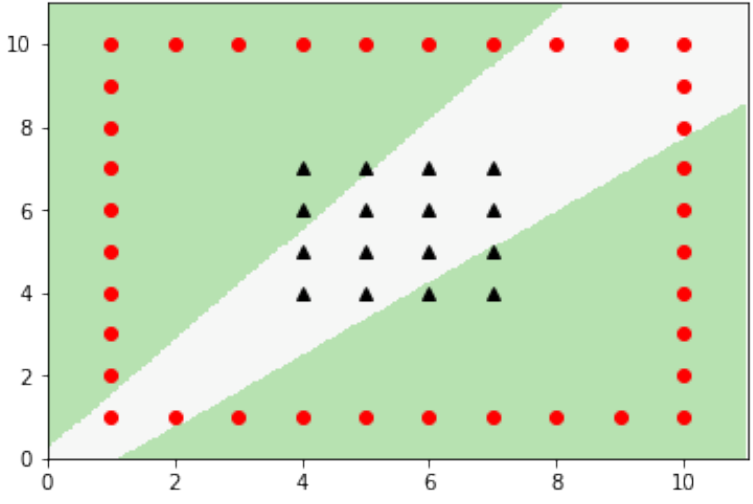




Take the AND of these.

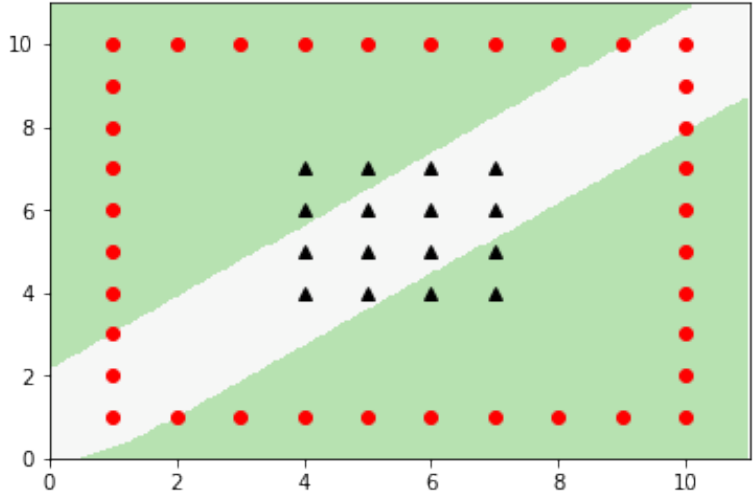
Example 2

$H = 4$



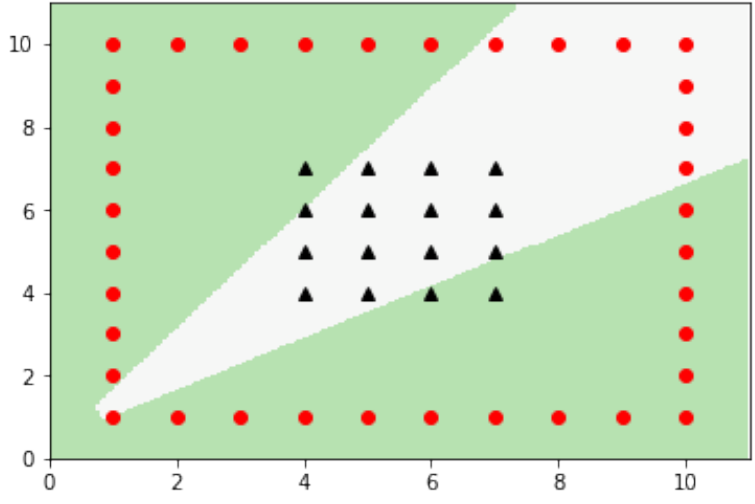
Example 2

$H = 4$



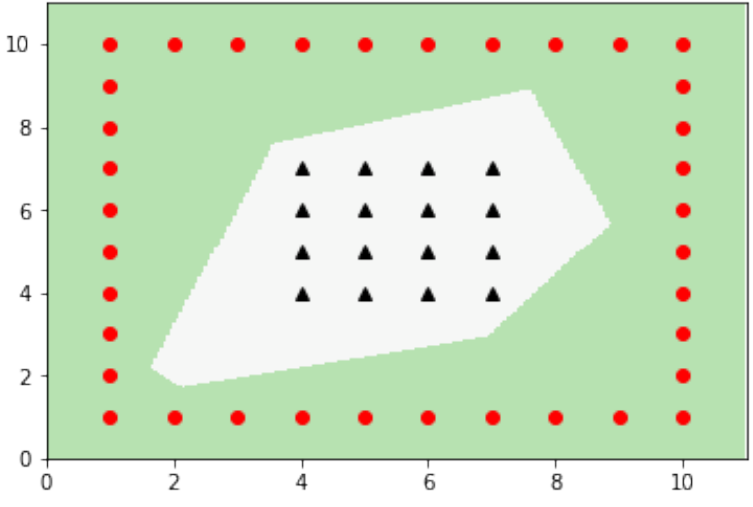
Example 2

$H = 4$



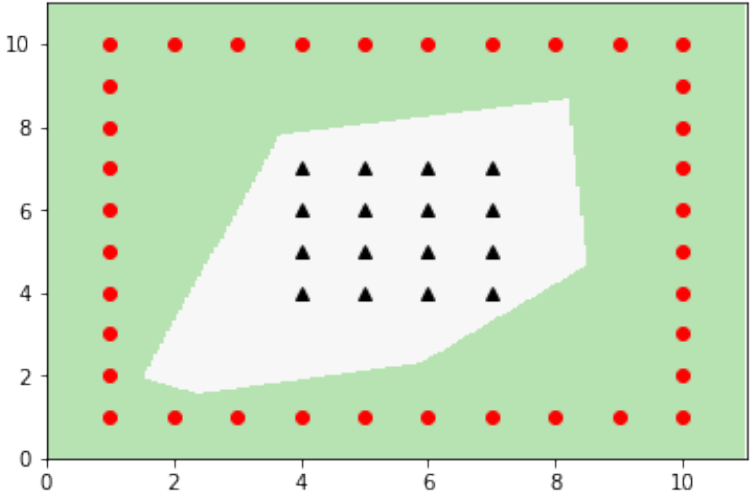
Example 2

$H = 4$



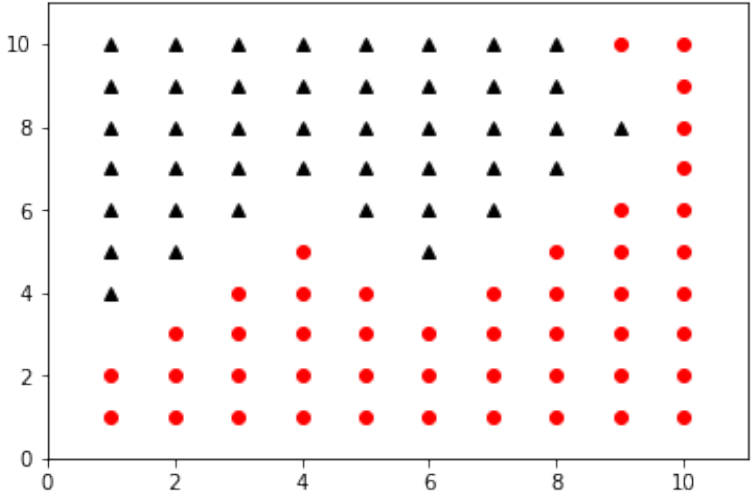
Example 2

$H = 8$: overparametrized



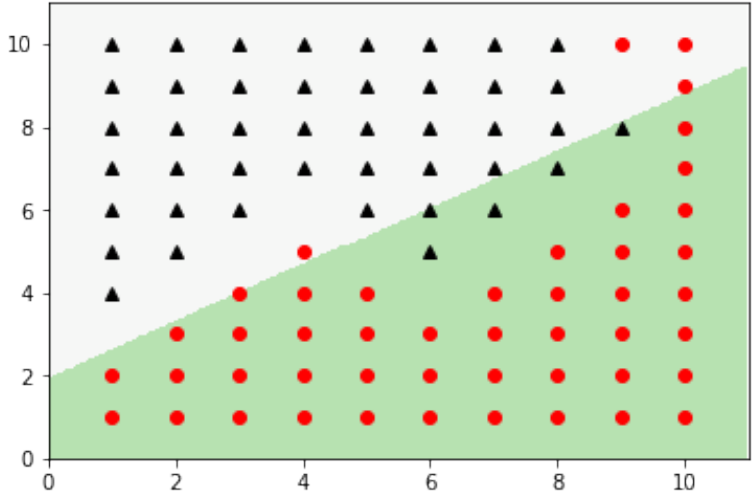
Example 3

How many hidden units should we use?



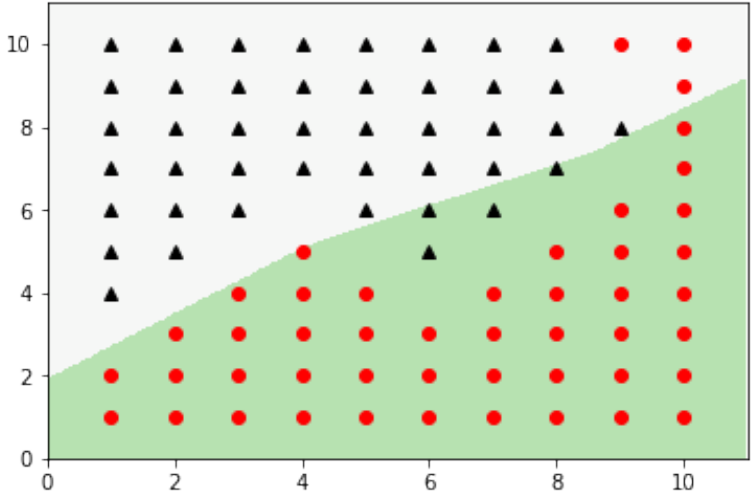
Example 3

$H = 4$



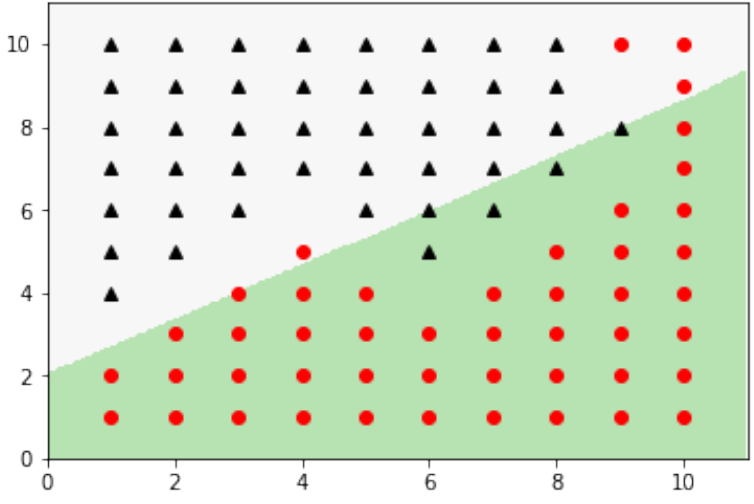
Example 3

$H = 8$



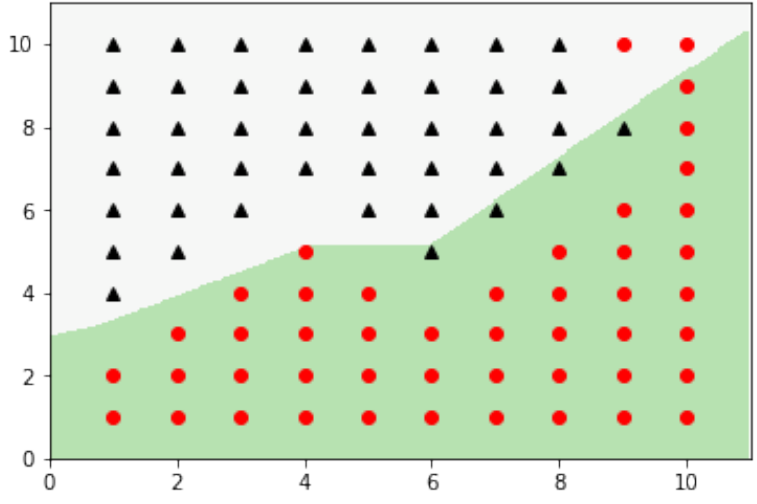
Example 3

$H = 16$



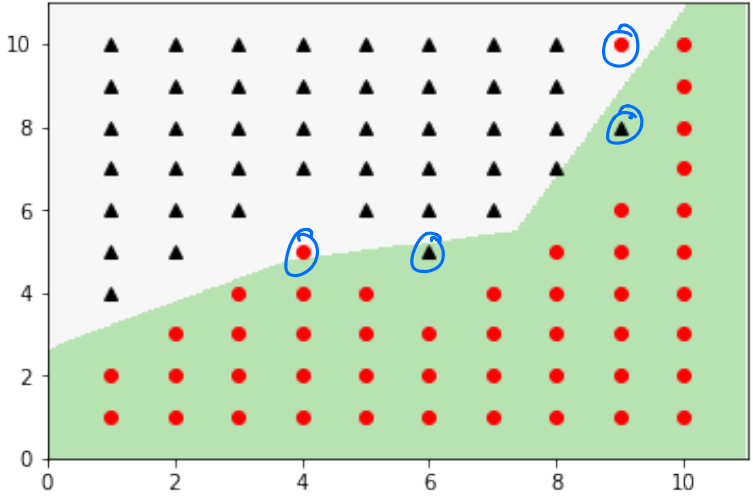
Example 3

$H = 16$



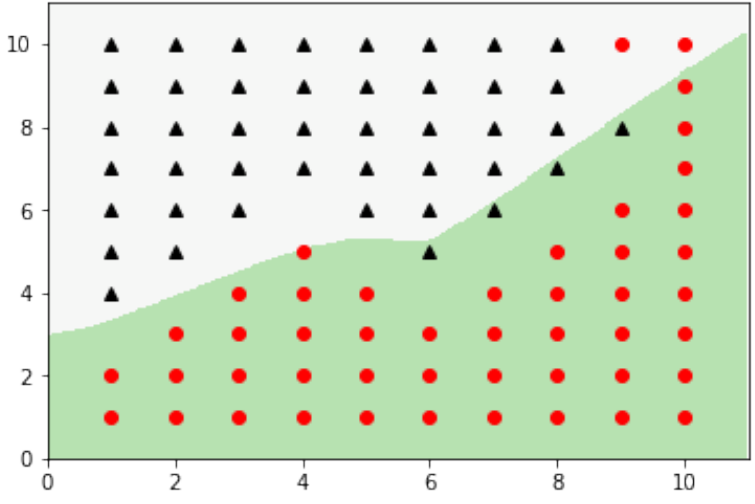
Example 3

$H = 16$



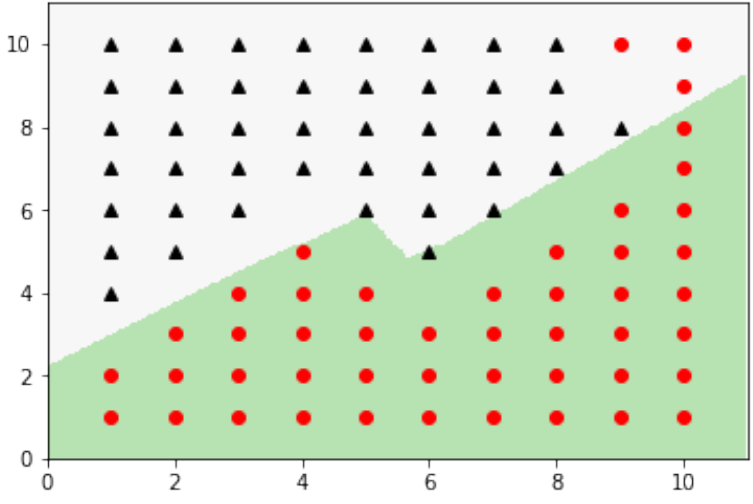
Example 3

$H = 32$



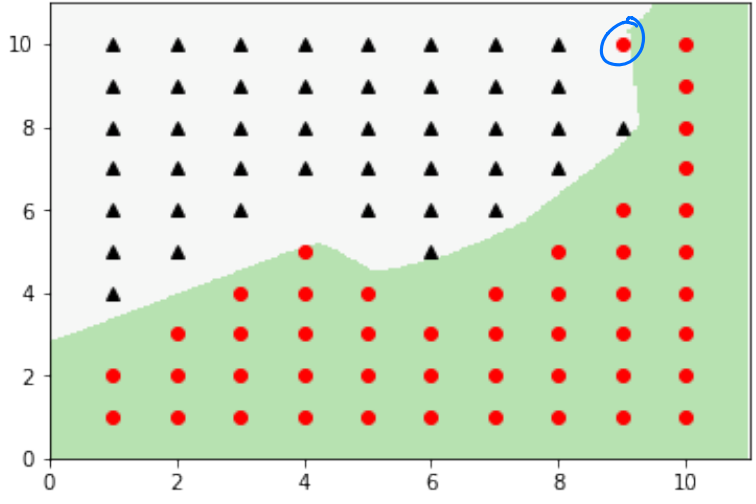
Example 3

$H = 32$



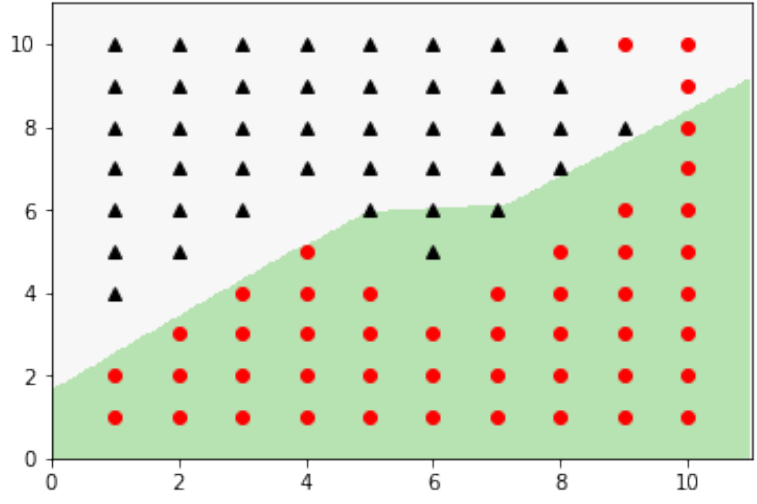
Example 3

$H = 32$



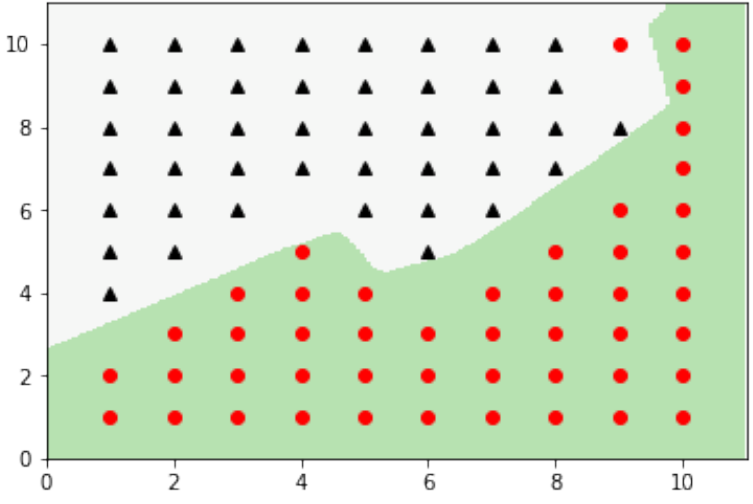
Example 3

$H = 64$



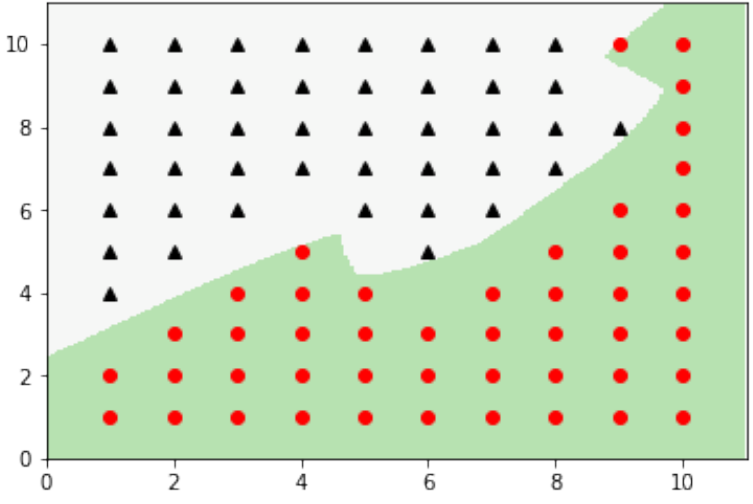
Example 3

$H = 64$



Example 3

$H = 64$



PyTorch snippet

Declaring and initializing the network:

input hidden

```
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
```

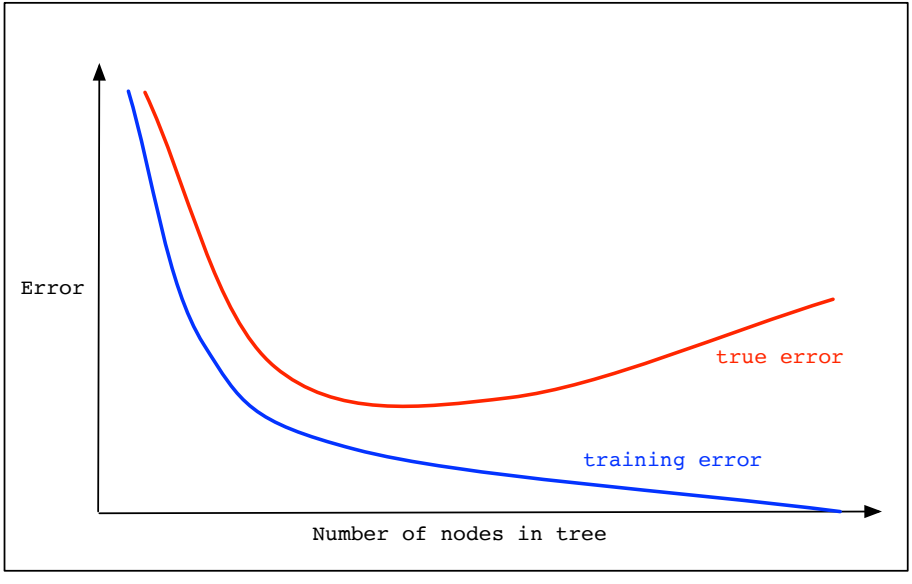
learning rate

Outline

1. Architecture
2. Learning
3. Bells and whistles

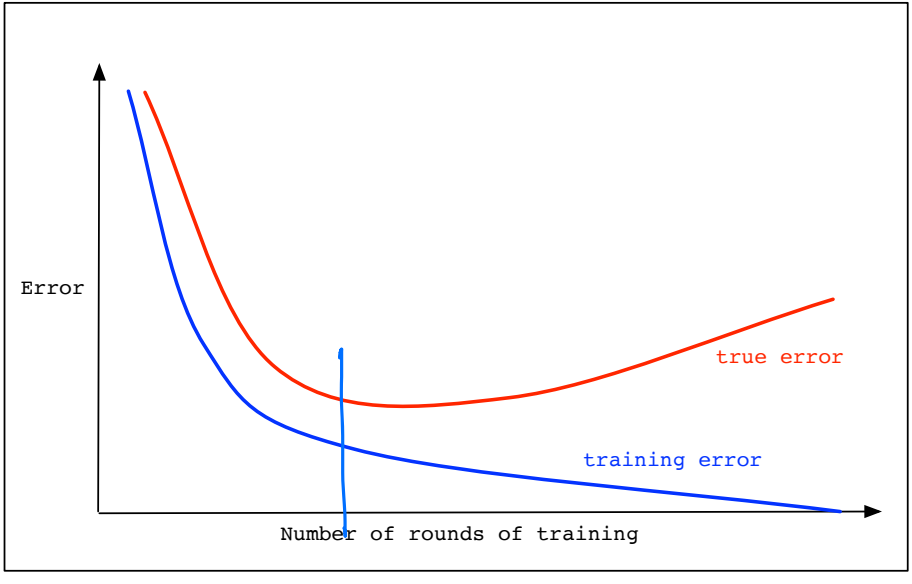
Improving generalization 1: Early stopping

- Validation set to better track error rate
- Revert to earlier model when recent training hasn't improved error



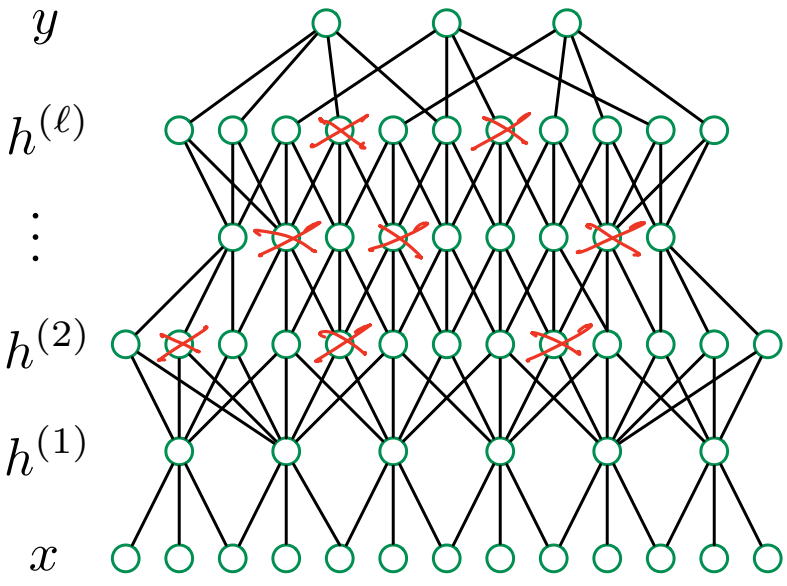
Improving generalization 1: Early stopping

- Validation set to better track error rate
- Revert to earlier model when recent training hasn't improved error



Improving generalization 2: Dropout

During training, delete each hidden unit with probability 1/2, independently.



Forces there to be multiple alternative paths to the answer... robustness.

What does this remind you of? Random forests

Facilitating optimization: Batch normalization

The distribution of inputs to a particular **layer** of the net can change dramatically during training: **internal covariate shift**.

Mitigate this with an additional normalization step.

For each layer $x_1, \ldots, x_p$ in the net, and *each mini-batch* B,

- Compute the mean $m_i^{(B)}$ and variance $v_i^{(B)}$ of each x_i in the mini-batch.
- Replace x_i by

$$x_i' = \frac{x_i - m_i^{(B)}}{\sqrt{v_i^{(B)} + \epsilon}}$$

before feeding to the next layer. This x_i' has mean 0 and variance ≈ 1.

Variants of SGD

Suppose we have parameters θ and loss $\ell(x, y; \theta)$. Usual SGD update:

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t g^{(t)}$$

where $g^{(t)} = \nabla \ell(x_t, y_t; \theta^{(t)})$ is the gradient at time t.

- **Momentum**: Accumulate gradients. For $g^{(t)}$ as above, and $v^{(0)} = 0$,

$$v^{(t)} = \mu v^{(t-1)} + \eta_t g^{(t)}$$
$$\theta^{(t+1)} = \theta^{(t)} - v^{(t)}$$

- **AdaGrad**: Different learning rate for each parameter, automatically tuned.

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \frac{\eta}{\sqrt{\sum_{t'<t} (g_j^{(t')})^2 + \epsilon}} g_j^{(t)}$$

Many others: **Adam**, **RMSProp**, etc.