CSE-571

Robotics

Neural Networks

[Slides courtesy of Daniel Gordon]


An Introduction To
Deep
Learning

What is deep learning anyway?

Typical ML pipeline:

Extract features -> optimize model -> inference

Deep learning:

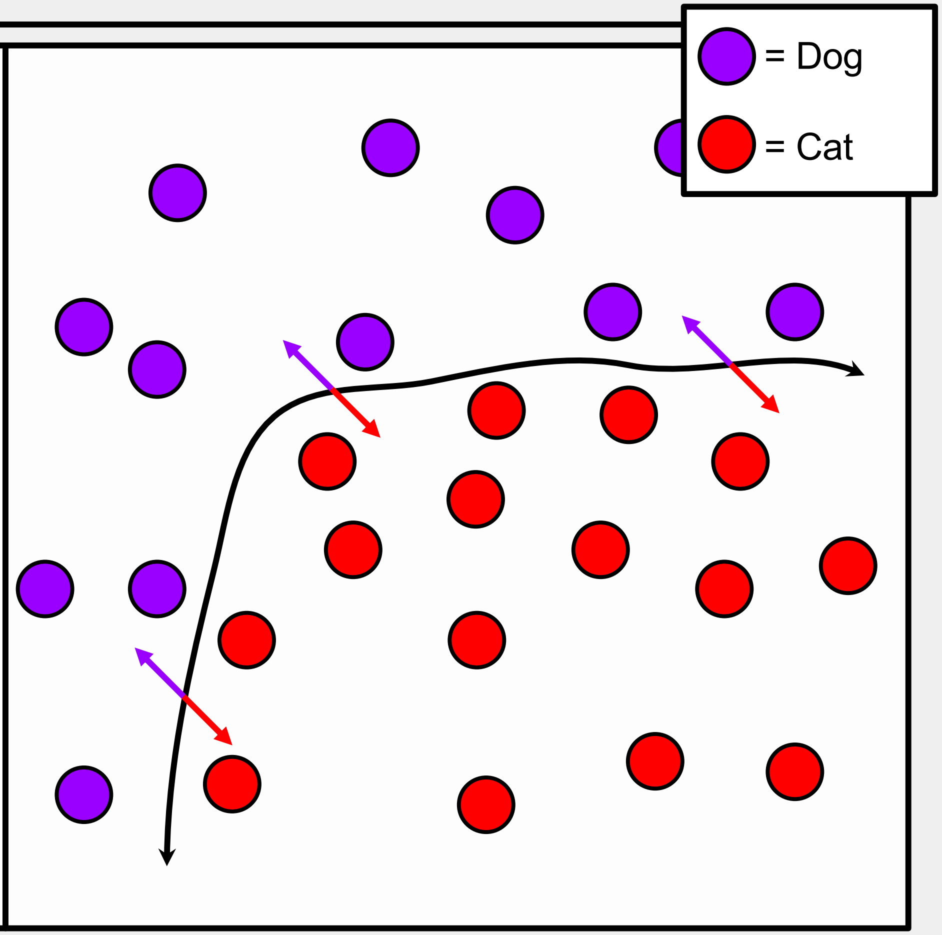
Optimize model -> inference

Classification: class labels

Each point belongs to one class

Goal: build a model that separates classes

E.g.: is this an image of a dog or a cat?

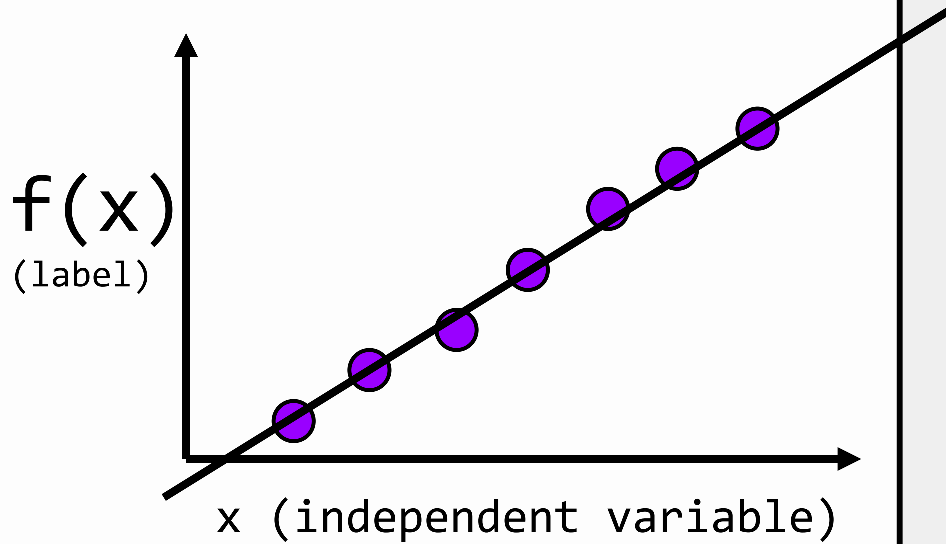


Regression: real-valued labels

Each point has a (or many) real-valued label

Goal: build model to predict real-values

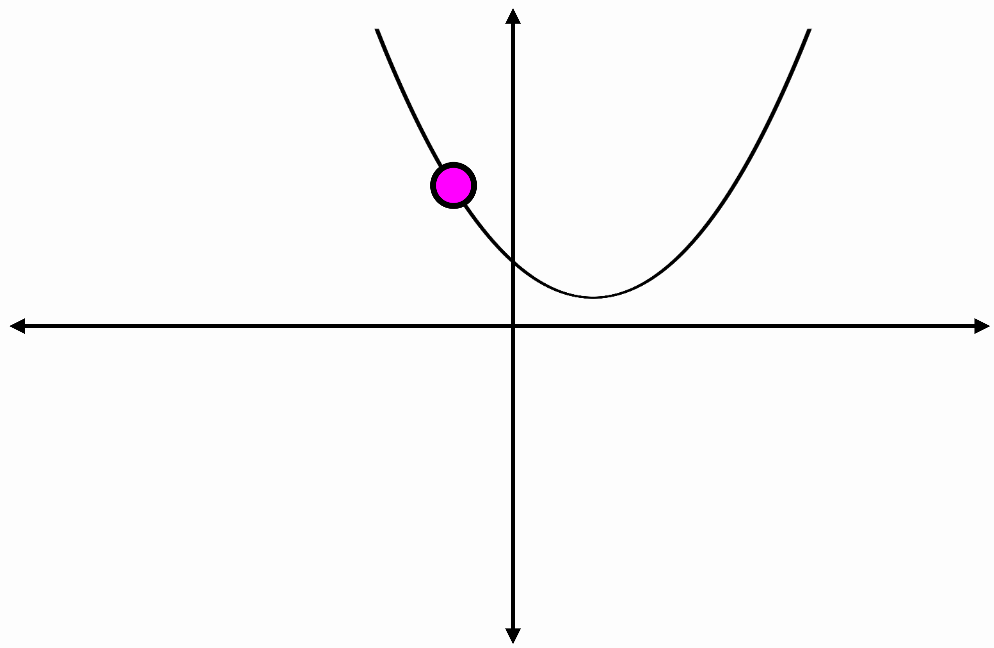
E.g.: How old is the person in this image?



Minimize some function

To minimize f(x)

$f(x) = e^{-x} + x^2$



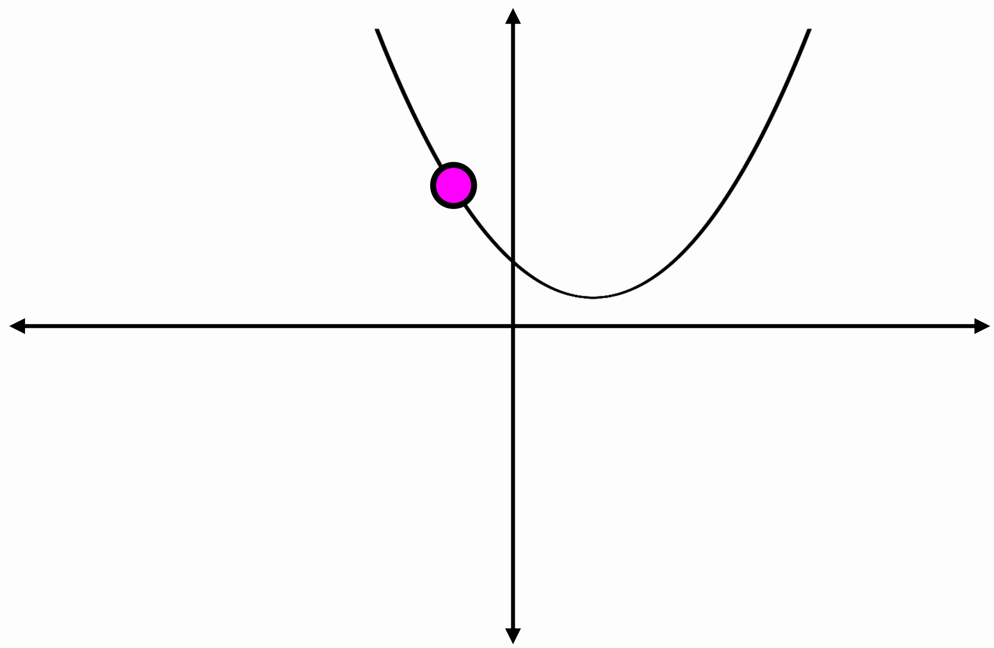
Guess and check

To minimize f(x), when tiny change makes f(x) smaller, do that!

```
f(x) = e^-x + x^2
```

$f(x) = e^{-x} + x^2$

```
f(-.50) = 1.8987
f(-.51) = 1.9254
f(-.49) = 1.8724
```

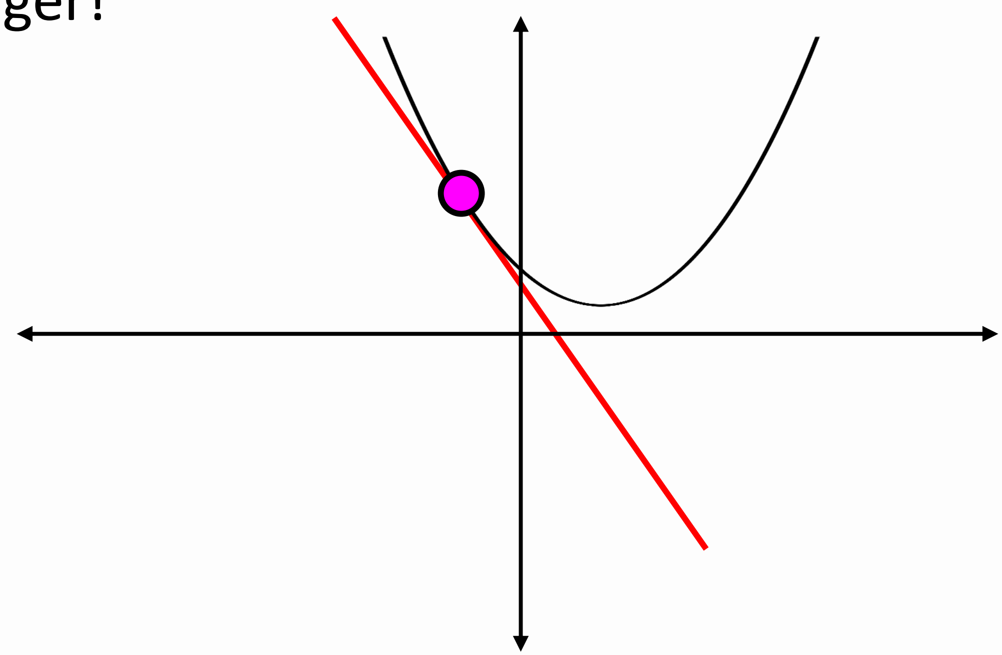


Gradient descent

To minimize f(x), when gradient is positive make x smaller, when negative make x larger!

```
f(x)  = e^-x + x^2

f'(x)= -e^-x + 2x
```

```
f (-.5) =  1.8987
f'(-.5) = -2.64872

Gradient is neg,
make x bigger!
```



Gradient descent

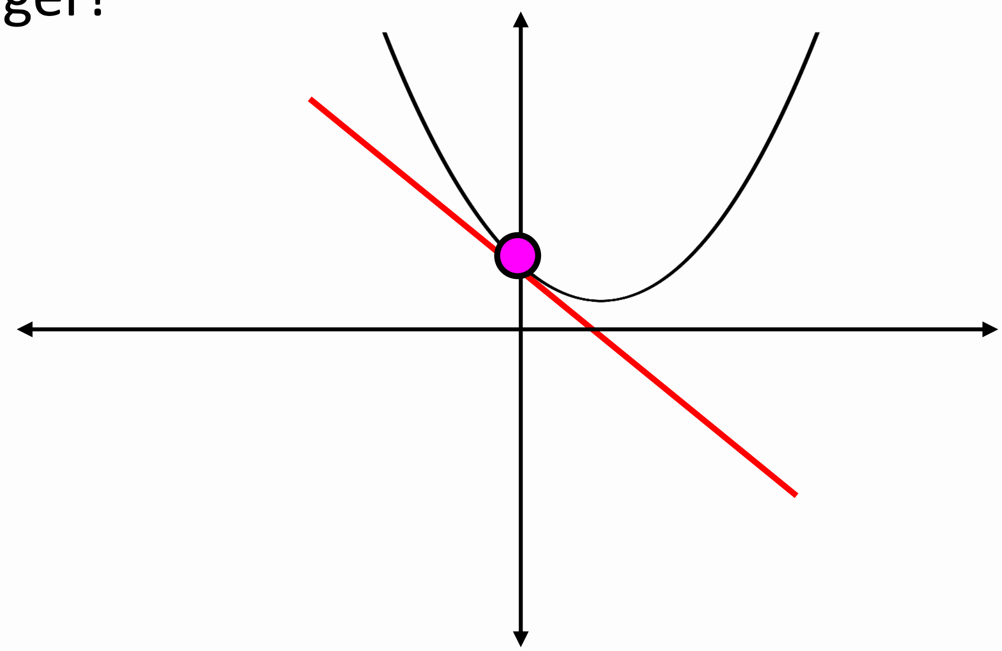
To minimize f(x), when gradient is positive make x smaller, when negative make x larger!

$f(x) = e^{-x} + x^2$

$f'(x) = -e^{-x} + 2x$

$f(0) = 1$
$f'(0) = -1$

Gradient is neg,
make x bigger!

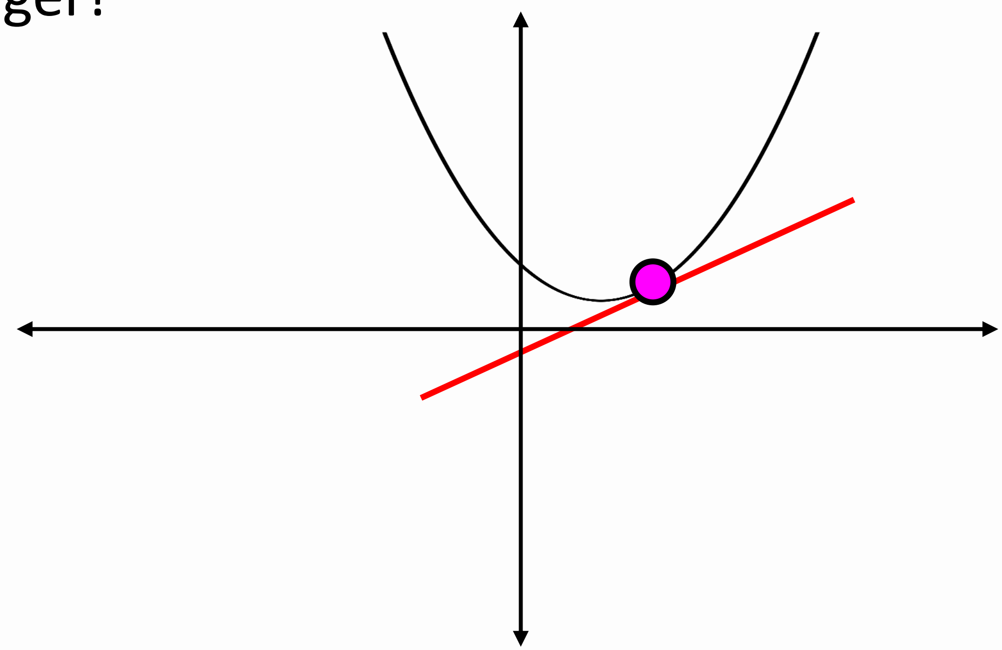


Gradient descent

To minimize f(x), when gradient is positive make x smaller, when negative make x larger!

$f(x) = e^{-x} + x^2$

$f'(x) = -e^{-x} + 2x$



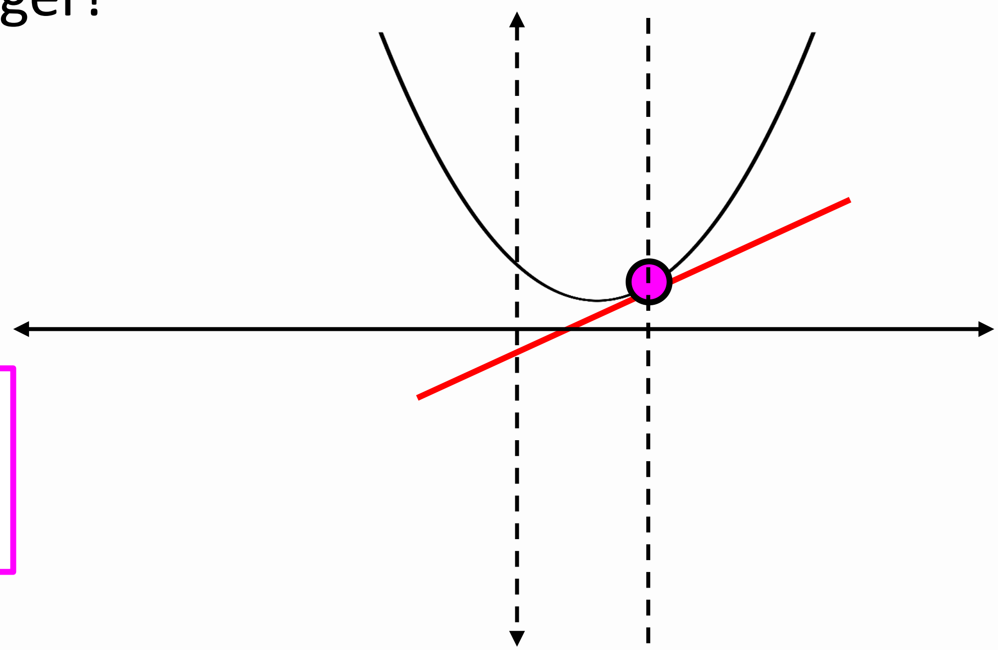
```
f (0.5) = 0.85653
f'(0.5) = 0.39347

Gradient is pos,
make x smaller!
```

Gradient descent

To minimize f(x), when gradient is positive make x smaller, when negative make x larger!

$f(x) = e^{-x} + x^2$

$f'(x) = -e^{-x} + 2x$



```
Know min is between
0 and 0.5, how can
we get more exact??
```

Gradient Descent Algorithm

To find $\mathrm{argmin}_x f(x)$

Initialize x somehow

Until converged:

    Compute gradient $\nabla f(x)$

    $x = x - \eta\nabla f$

η is *learning rate*

What does this have to do with ML?

Remember, we wanted to optimize our models to fit the data. First we need a measure of "goodness-of-fit":

Likelihood function - how likely our model thinks our data is

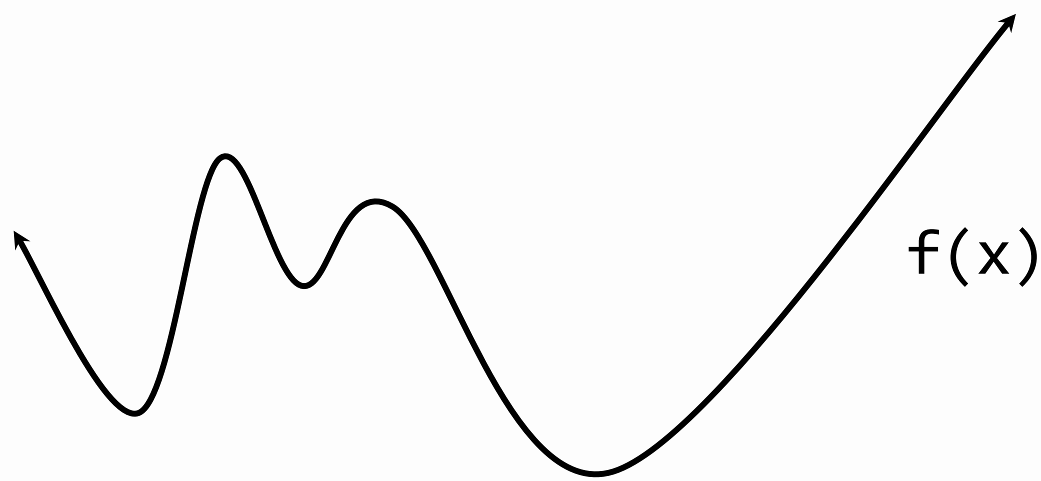
Loss function - how wrong is our model

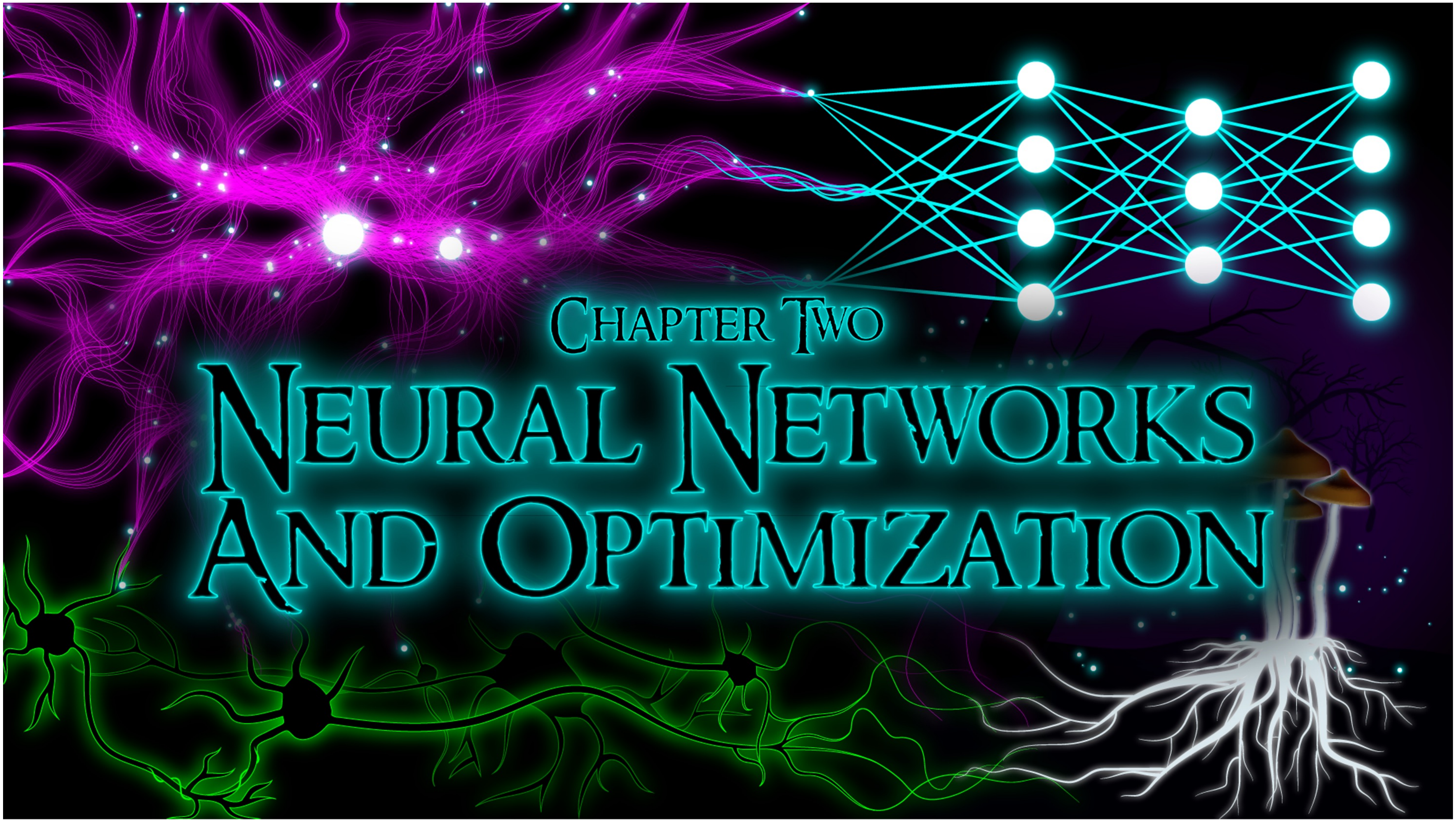
Want to find parameters that maximize likelihood or minimize loss!

Non-convex optimization

Extrema may be local or global, don't always know which you have!

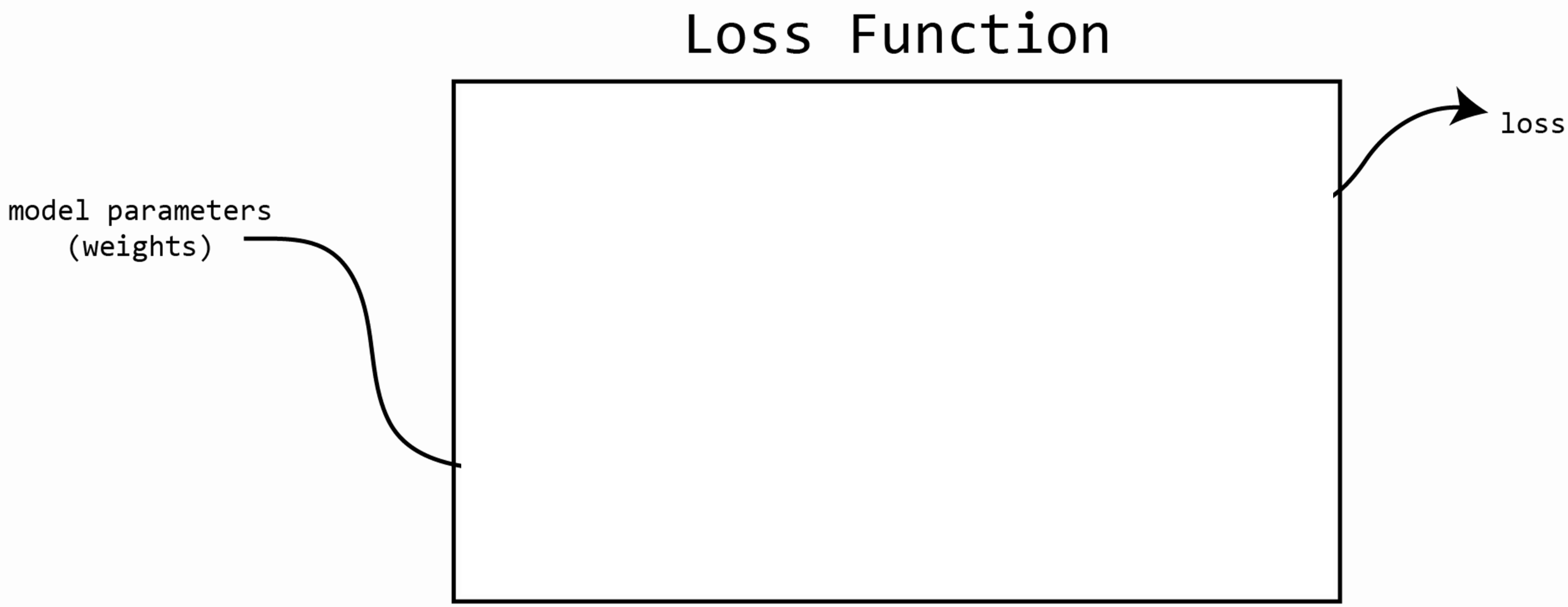
With neural networks we are performing non-convex optimization, we aren't guaranteed a globally optimal solution :-(




Chapter Two
Neural Networks
And Optimization

Loss functions are the key!

Just a function! Want to find $\text{argmin}_{\text{weights}}$(Loss Function)



A note on notation...

In 1d we talk about derivatives,

$f'(x) = d/dx\ f(x)$

We want to see how to change the input to modify the output, only one variable to worry about!

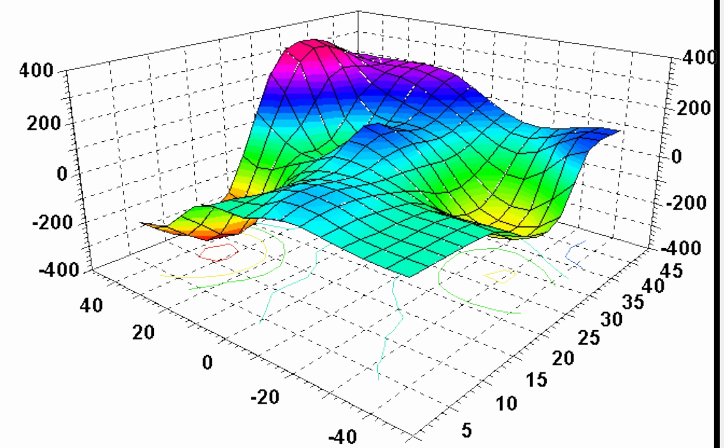
A note on notation...

In more dimensions, we want partial derivatives to see how each component in input affects output:

$\nabla f(\mathbf{x}) = [\partial f(x)/\partial x_1, \partial f(x)/\partial x_2 \ldots]$

$\nabla f(\mathbf{x})$ is a vector of partial derivatives of the function f

$\nabla L(\mathbf{w})$ is gradient of loss function wrt. **w**



Basic ML Models

Linear Regression: Best fit line

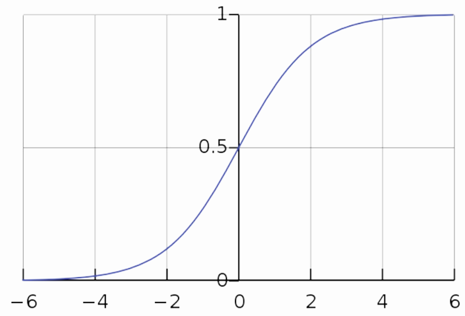
$$f(\mathbf{x}) = \Sigma_i \, w_i \cdot x_i = \mathbf{w} \cdot \mathbf{x}$$

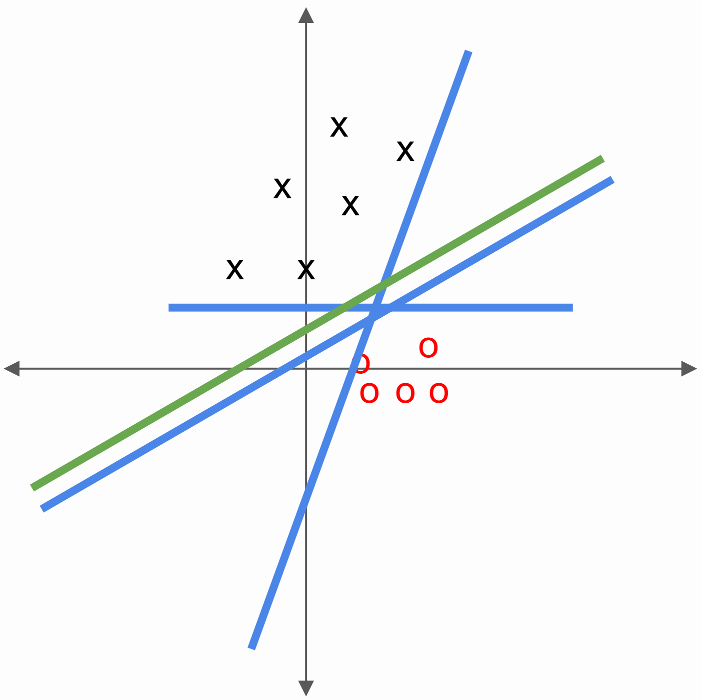
$$L(\mathbf{w}) = ||Y - f(x)||^2$$

$$\partial L(\mathbf{w})/\partial w_i = x_i[Y - \mathbf{w} \cdot \mathbf{x}]$$

Basic ML Models

- Logistic Regression
  - $\sigma(x) = 1/(1 + e^{-x}) = e^{x}/(1 + e^{x})$
  - Maps all reals -> [0,1], probabilities!





$f(\mathbf{x}) = \sigma(\Sigma_i\, w_i \cdot x_i) = \sigma(\mathbf{w} \cdot \mathbf{x})$

$L(\mathbf{w}) = ||Y - f(x)||^2$

$\partial L(\mathbf{w})/\partial w_i = x_i[Y - \mathbf{w} \cdot \mathbf{x}]$

Not actual logistic regression, but same principle

Gradient Descent Algorithm

To find $\text{argmin}_x f(x)$

Initialize x somehow

Until converged:

  Compute gradient $\nabla f(x)$

  $x = x - \eta \nabla f$

η is *learning rate*

Stochastic gradient descent (SGD)

Estimate $\nabla L(\mathbf{w})$ with only some of the data

Before:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \, \Sigma_i \nabla L_i(\mathbf{w}), \text{ for all i in |data|}$$

Now:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \, \Sigma_j \nabla L_j(\mathbf{w}), \text{ for some subset j}$$

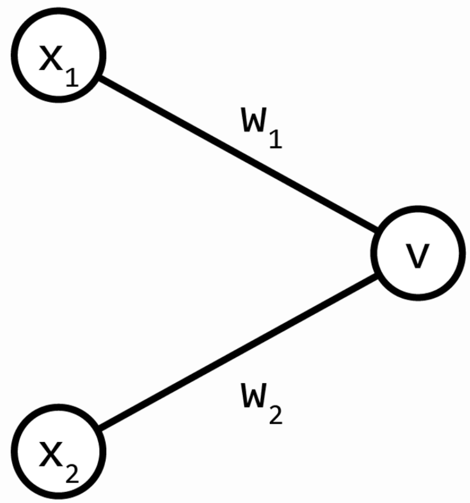
Maybe even:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \, \nabla L_k(\mathbf{w}), \text{ for some random k}$$

# of points used for update is called *batch size*

Basic ML Models

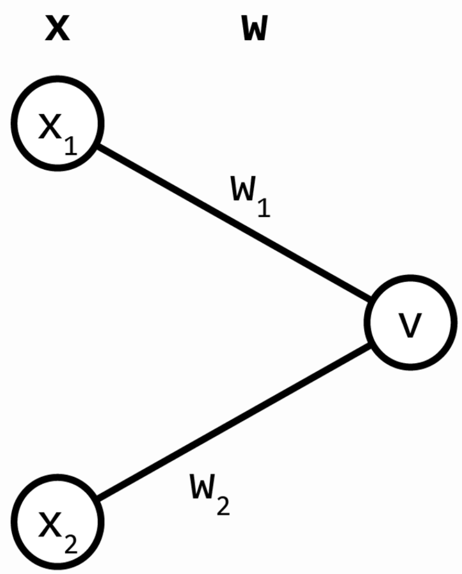
Single Layer Neural Network:



$$v = w_1x_1 + w_2x_2$$

Basic ML Models

Single Layer Neural Network:



$v = w_1x_1 + w_2x_2$

$v = \mathbf{w}\cdot\mathbf{x}$

$\nabla L(\mathbf{w}) = \mathbf{X}[Y-\mathbf{w}\cdot\mathbf{x})]$

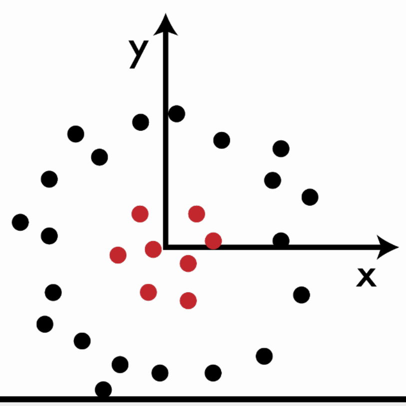
Weight update rule:

$\mathbf{w} = \mathbf{w} + \eta\ \mathbf{x}[Y-\mathbf{w}\cdot\mathbf{x}]$

Feature engineering

Arguably the **core** problem of machine learning (especially in practice)

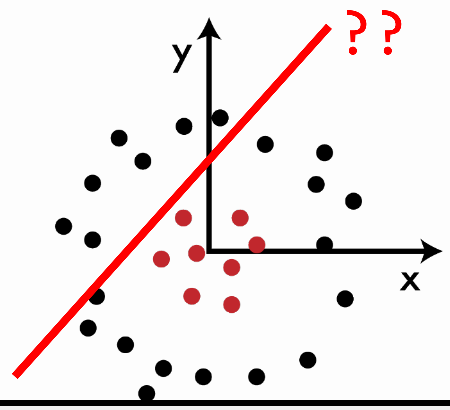
ML models work well if there is a clear relationship between the inputs and outputs of the function you are trying to model



Feature engineering

Arguably the **core** problem of machine learning (especially in practice)

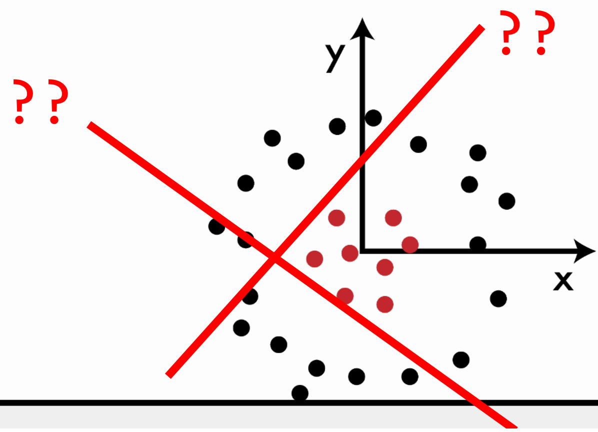
ML models work well if there is a clear relationship between the inputs and outputs of the function you are trying to model



Feature engineering

Arguably the **core** problem of machine learning (especially in practice)

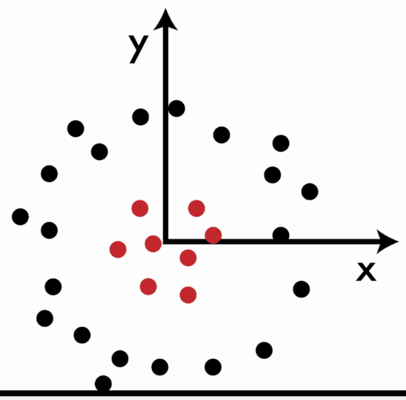
ML models work well if there is a clear relationship between the inputs and outputs of the function you are trying to model

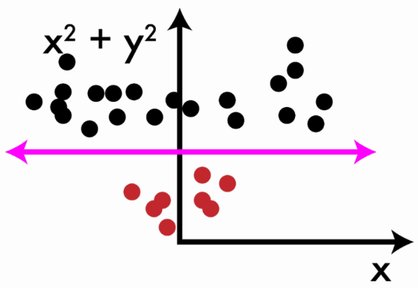


Feature engineering

Arguably the **core** problem of machine learning (especially in practice)

ML models work well if there is a clear relationship between the inputs and outputs of the function you are trying to model





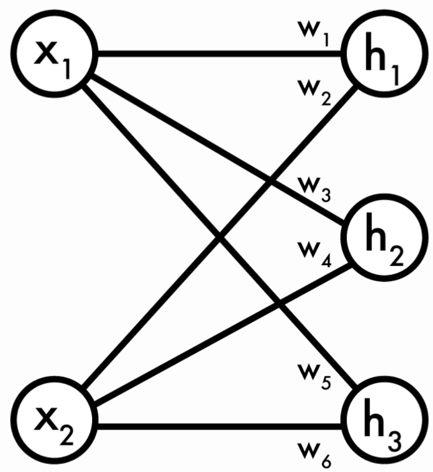
Quick Demo

https://playground.tensorflow.org/#activation=linear&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=&seed=0.79237&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

https://playground.tensorflow.org/#activation=linear&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=&seed=0.79237&showTestData=false&discretize=false&percTrainData=50&x=false&y=false&xTimesY=false&xSquared=true&ySquared=true&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

What if we added more processing?

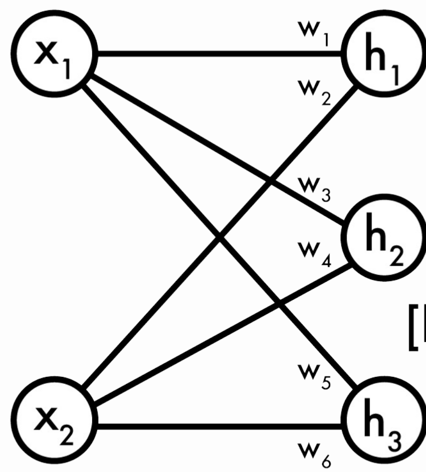
Create “new” features using old ones. We’ll call **H** our *hidden layer*



$$h_1 = \varphi(w_1 x_1 + w_2 x_2)$$
$$h_2 = \varphi(w_3 x_1 + w_4 x_2)$$
$$h_3 = \varphi(w_5 x_1 + w_6 x_2)$$

What if we added more processing?

H can be expressed in matrix operations



$h_1 = \varphi(w_1x_1 + w_2x_2)$

$h_2 = \varphi(w_3x_1 + w_4x_2)$

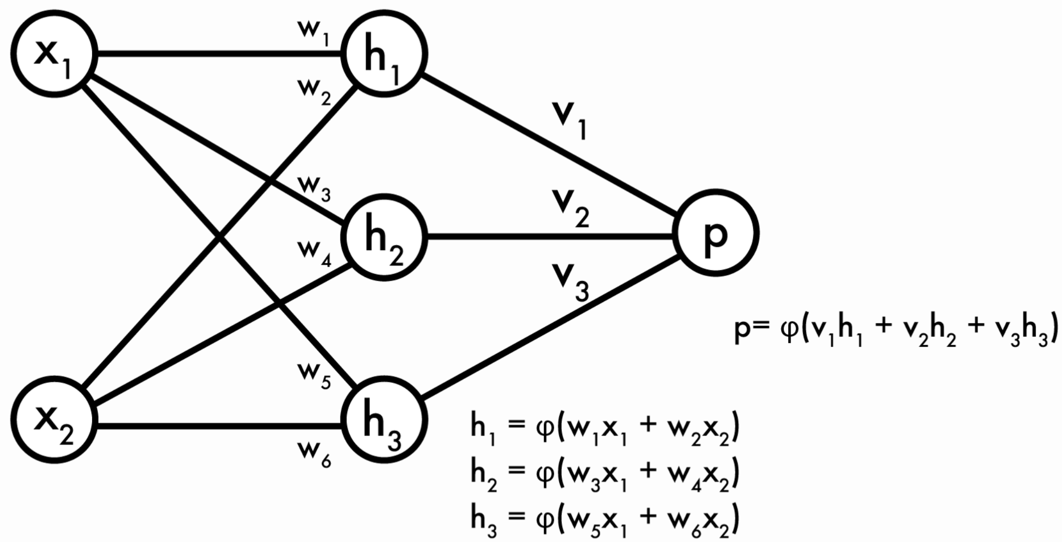
$h_3 = \varphi(w_5x_1 + w_6x_2)$

$$[h_1\ h_2\ h_3] = \varphi([x_1\ x_2]\begin{bmatrix} w_1 & w_3 & w_6 \\ w_2 & w_4 & w_5 \end{bmatrix})$$

$$\mathbf{H} = \varphi(\mathbf{Xw})$$

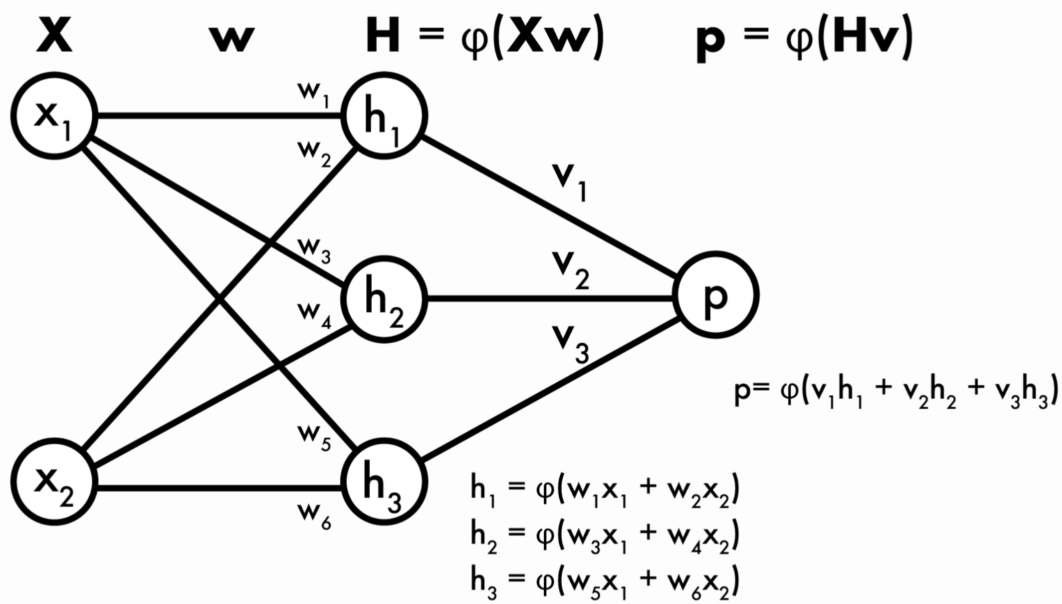
What if we added more processing?

Now our prediction p is a function of our hidden layer



What if we added more processing?

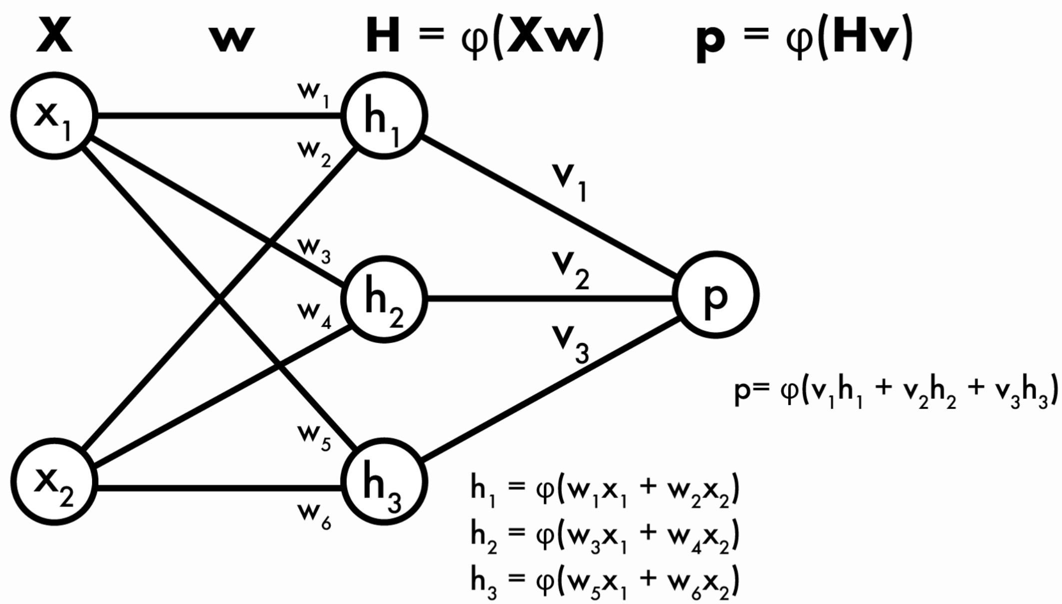
Can express the whole process in matrix notation! Nice because matrix ops are fast



This is a neural network!

This one has 1 hidden layer, but can have **way** more

Each layer is just some function φ applied to linear combination of the previous layer



Quick Demo

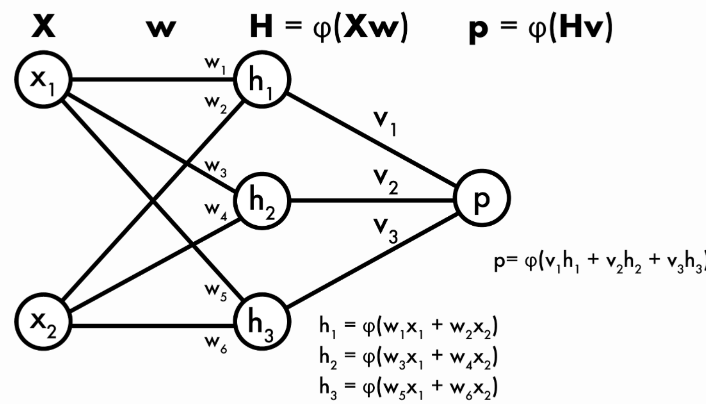
https://playground.tensorflow.org/#activation=linear&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=3&seed=0.79237&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

What went wrong?

φ is our *activation function*

Want to apply some extra processing at each layer. Why?

Imagine φ(x) = x, linear activation



φ is our *activation function*

Want to apply some extra processing at each layer. Why?

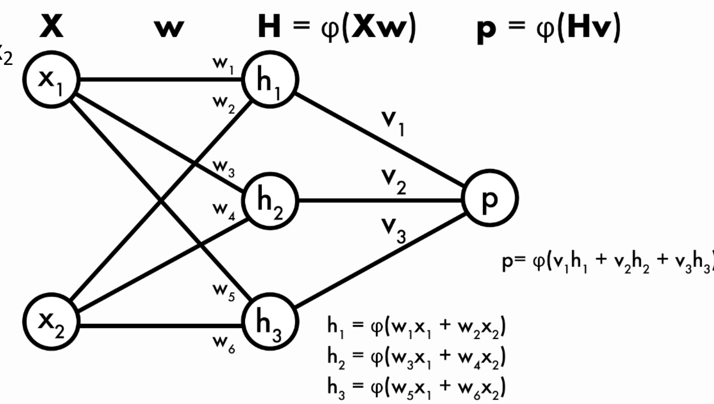
Imagine $\varphi(x) = x$, linear activation

$p = v_1h_1 + v_2h_2 + v_3h_3$

But $h_1 = x_1w_1 + x_2w_2$, $h_2 = \ldots$ etc

So

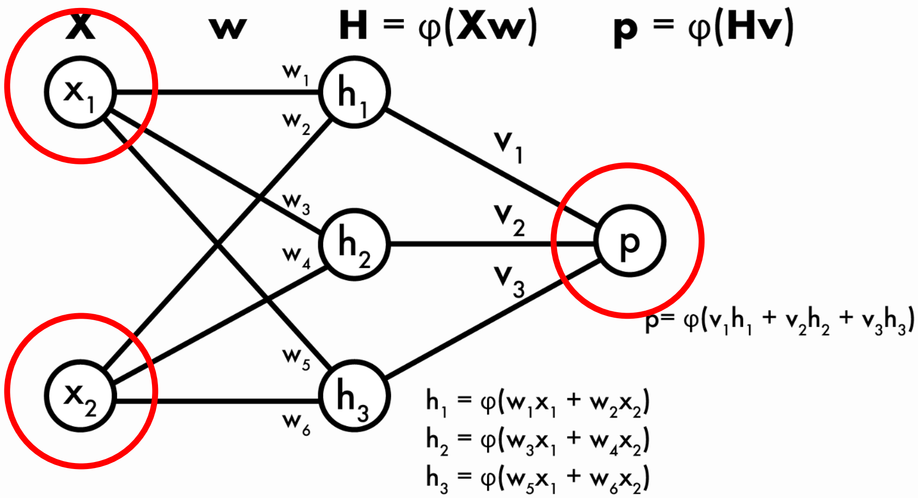
$$p = v_1w_1x_1 + v_1w_2x_2 + v_2w_3x_1 + v_2w_4x_2 + v_3w_5x_1 + v_3w_6x_2$$
$$= (v_1w_1+v_2w_3+v_3w_5)x_1 + (v_1w_2+v_2w_4+v_3w_6)x_2$$
$$= u_1x_1 + u_2x_2$$



Quick Demo

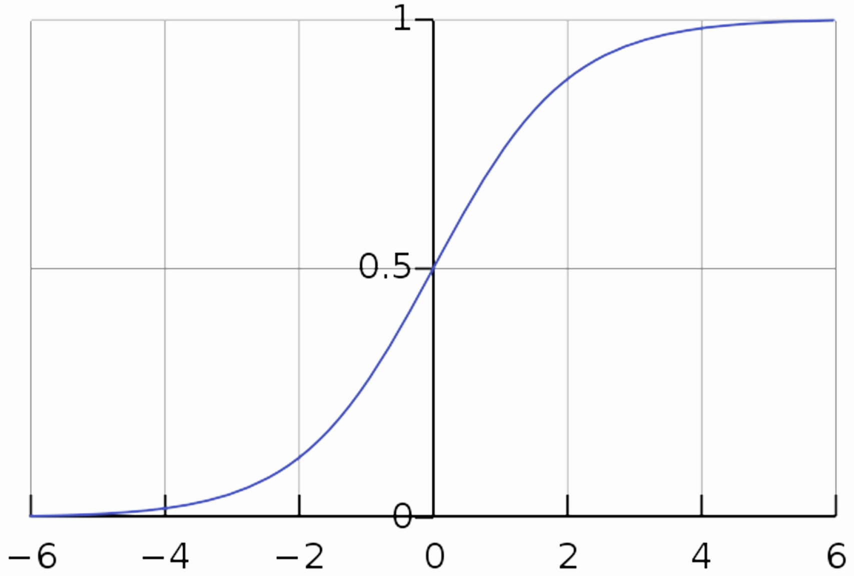
https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=3&seed=0.79237&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

Putting everything together



How do we learn it?: Logistic regression

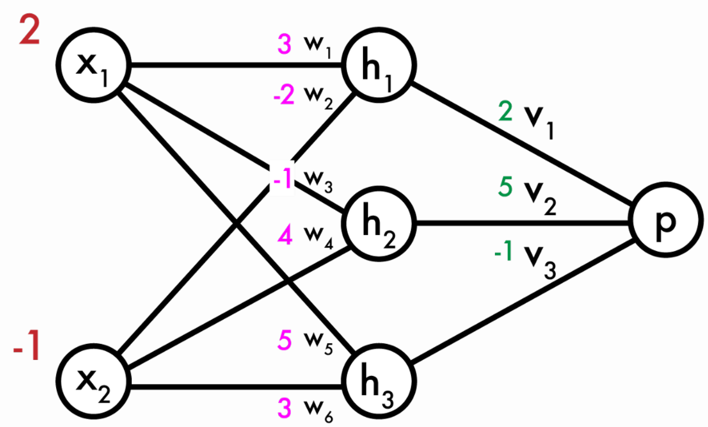
- Linear classifier, f is logistic function
  - $\sigma(x) = 1/(1 + e^{-x}) = e^x/(1 + e^x)$
  - Maps all reals -> [0,1], probabilities!



How do we learn it?

Now we have a “real” neural network (using linear activation for simplicity). How do we predict p?

P = **(2, -1)**
Label = +

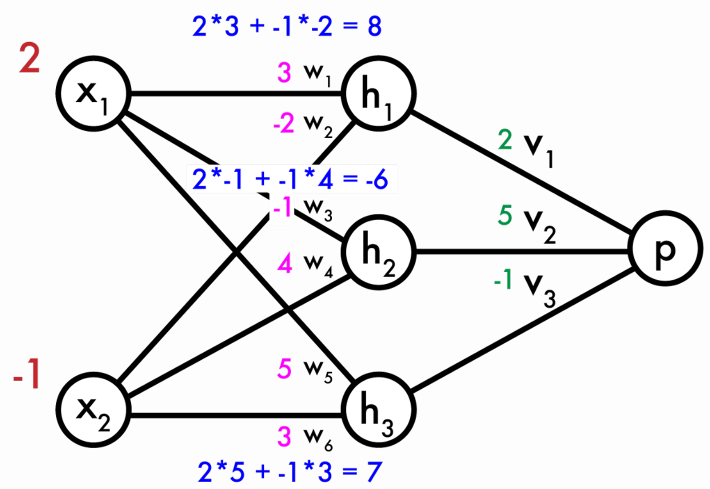


How do we learn it?

Now we have a "real" neural network (using linear activation for simplicity). How do we predict p?

Calculate hidden layer neurons

P = **(2, -1)**
Label = +



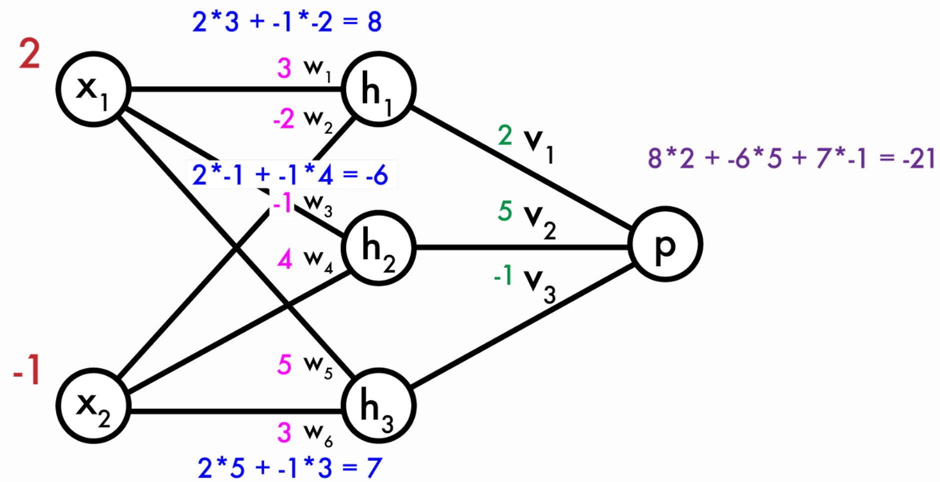
How do we learn it?

Now we have a “real” neural network (using linear activation for simplicity). How do we predict p?

Calculate hidden layer neurons

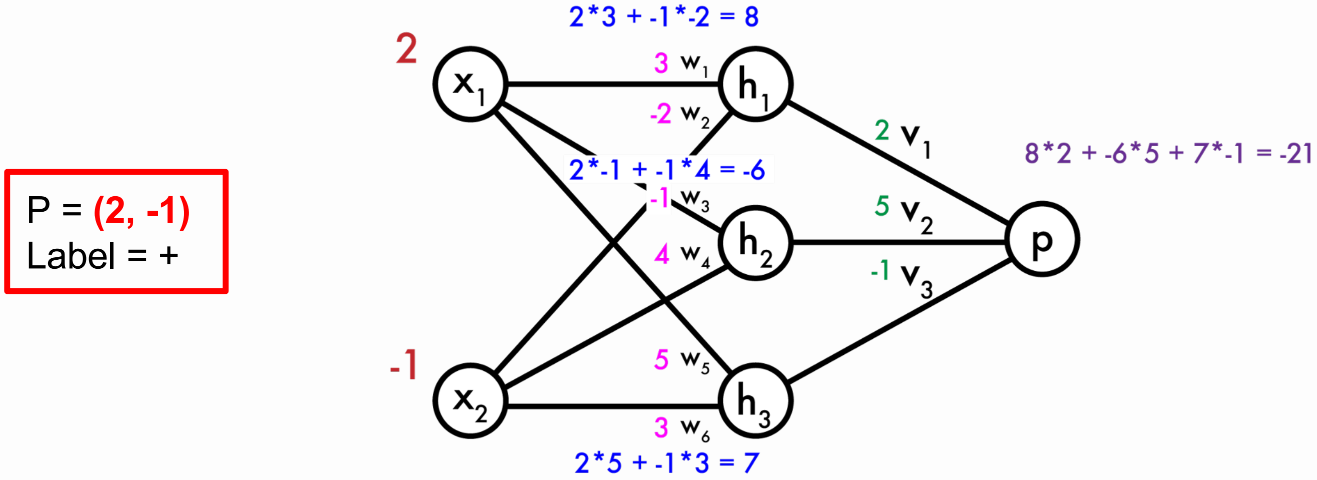
Calculate output p

P = **(2, -1)**
Label = +



How do we learn it?

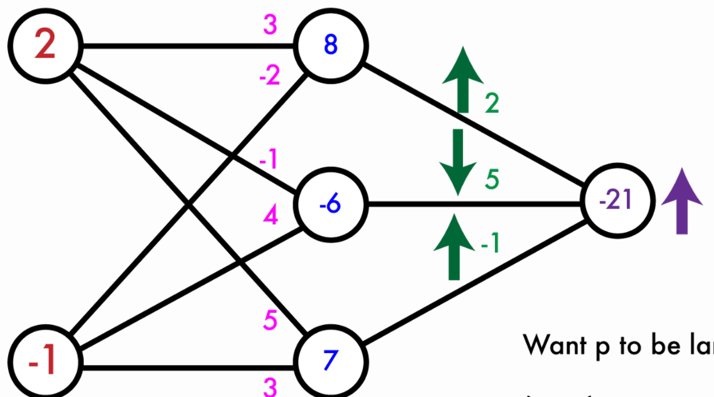
We want to make p larger. How do we modify the weights? The first layer is easy, same as normal linear model:



How do we learn it?

Say we want to make p larger. How do we modify the weights? The first layer is easy, same as normal linear model:

P = **(2, -1)**
Label = +



Want p to be larger...

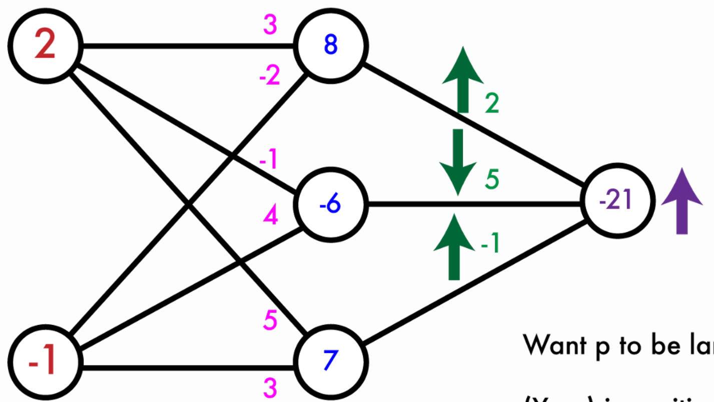
(Y - p) is positive

How do we change our weights?

How do we learn it?

Now what? Let's calculate the "error" that the hidden layer makes. We want p to be larger, given current weights how should we adjust the hidden layer output to do that?

P = **(2, -1)**
Label = +



Want p to be larger...

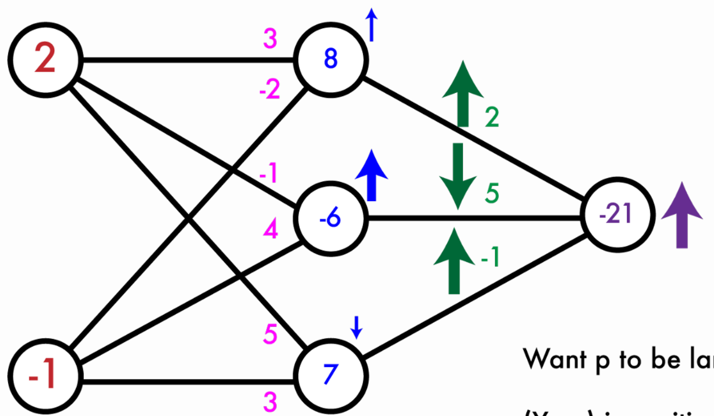
(Y - p) is positive

How do we change our weights?

How do we learn it?

Now what? Let’s calculate the “error” that the hidden layer makes. We want p to be larger, given current weights how should we adjust the hidden layer output to do that?

P = **(2, -1)**
Label = +



Want p to be larger...

(Y - p) is positive

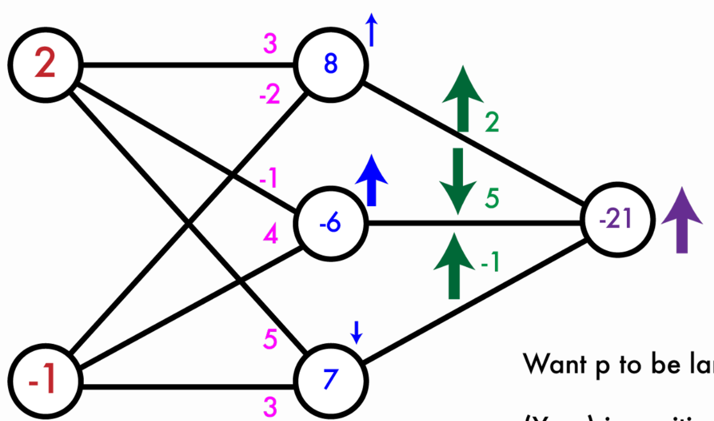
How do we change our weights?

How do we learn it?

Now that we have an "error" in our hidden layer, want to modify the previous weights.

Easy again, just like our linear model.

P = **(2, -1)**
Label = +



Want p to be larger...

(Y - p) is positive

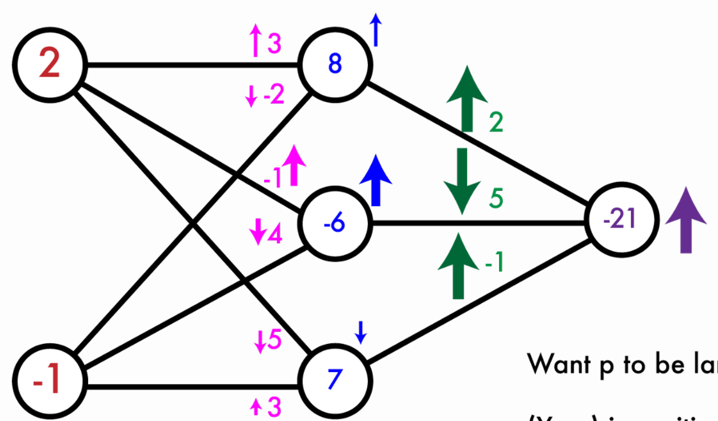
How do we change our weights?

How do we learn it?

Now that we have an “error” in our hidden layer, want to modify the previous weights.

Easy again, just like our linear model.

P = **(2, -1)**
Label = +



Want p to be larger...

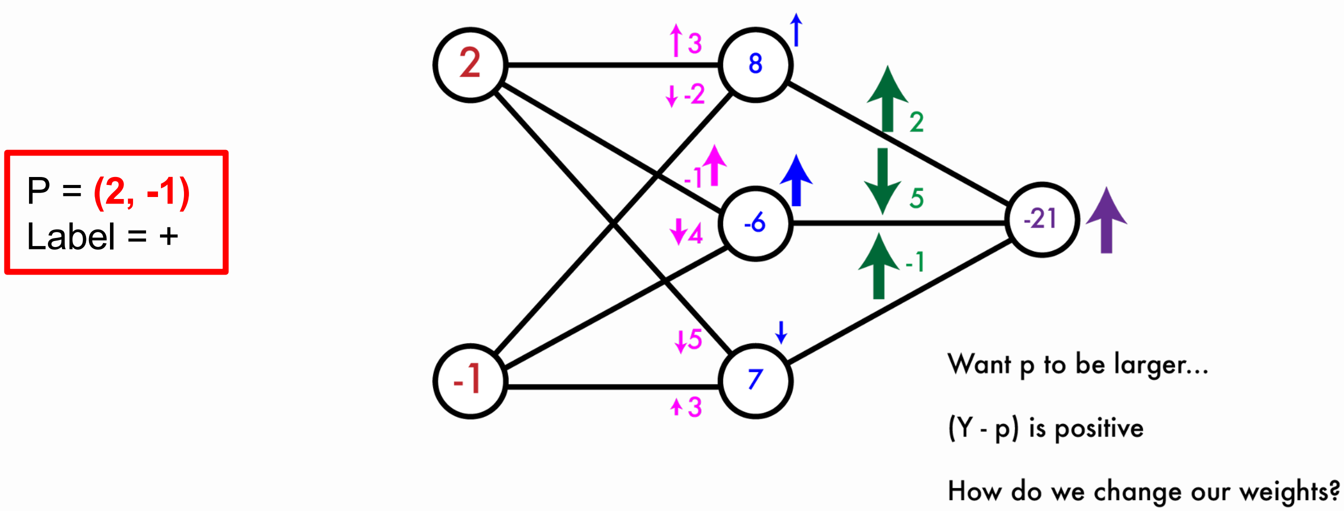
(Y - p) is positive

How do we change our weights?

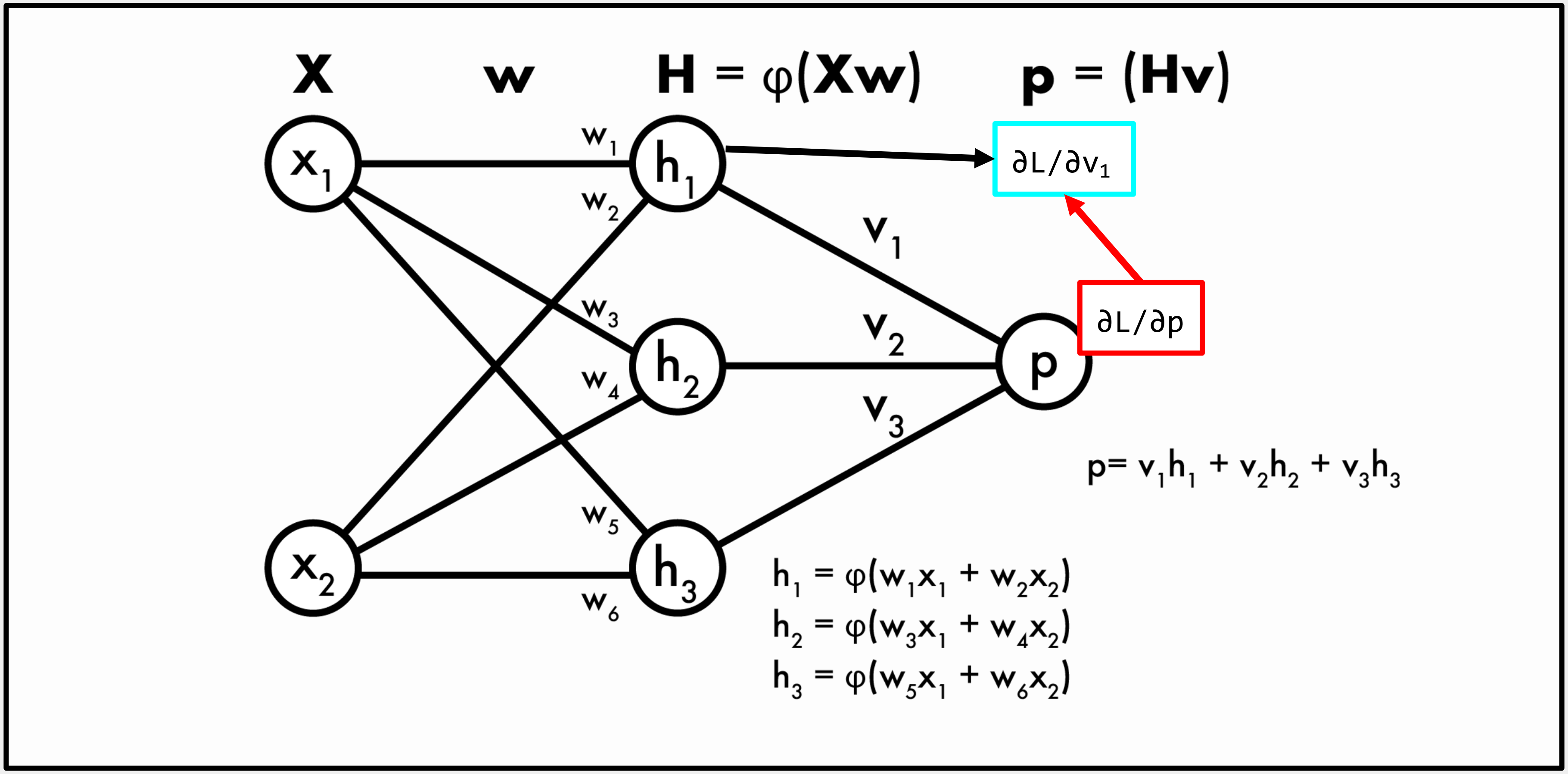
Backpropagation: just taking derivatives

Move in the (opposite) direction of the gradient proportional to the error.

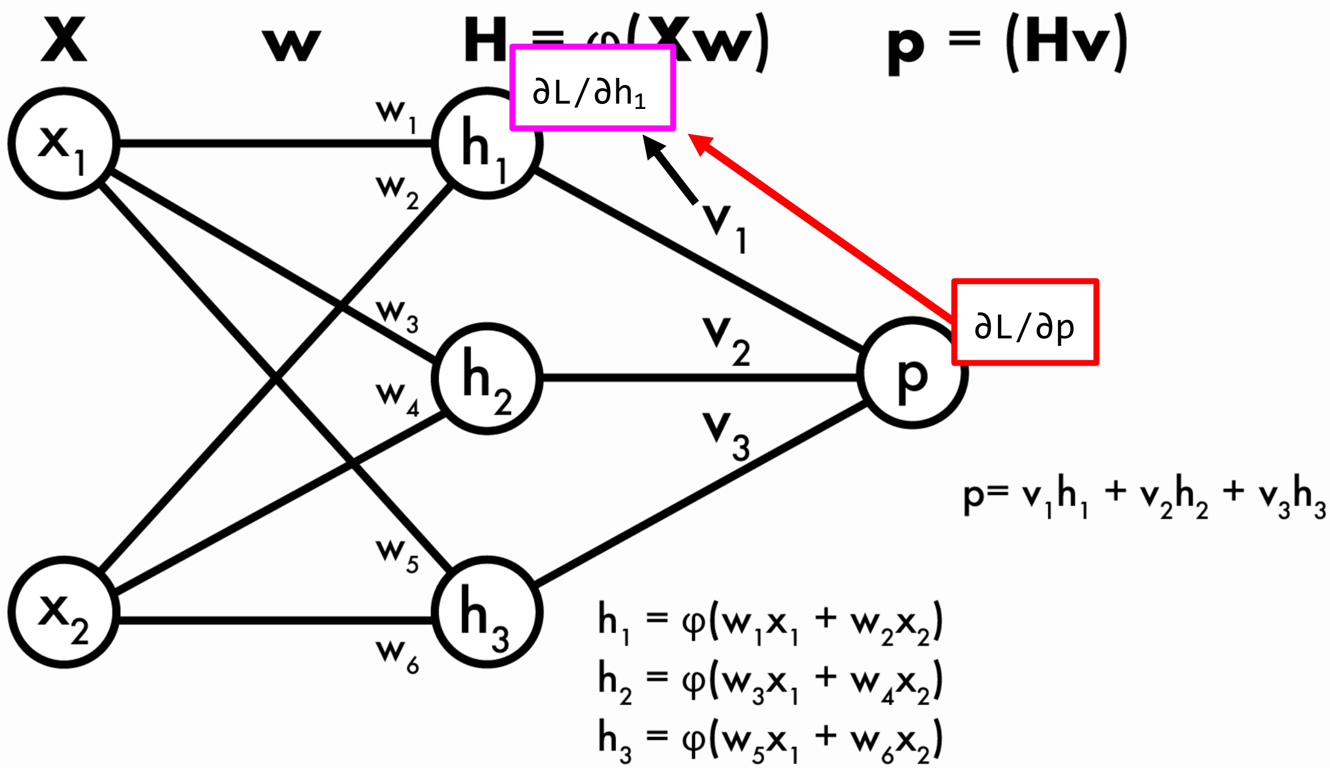
This was with linear activations but the process is the same for any φ, just have to calculate φ'(x) for that neuron as well.



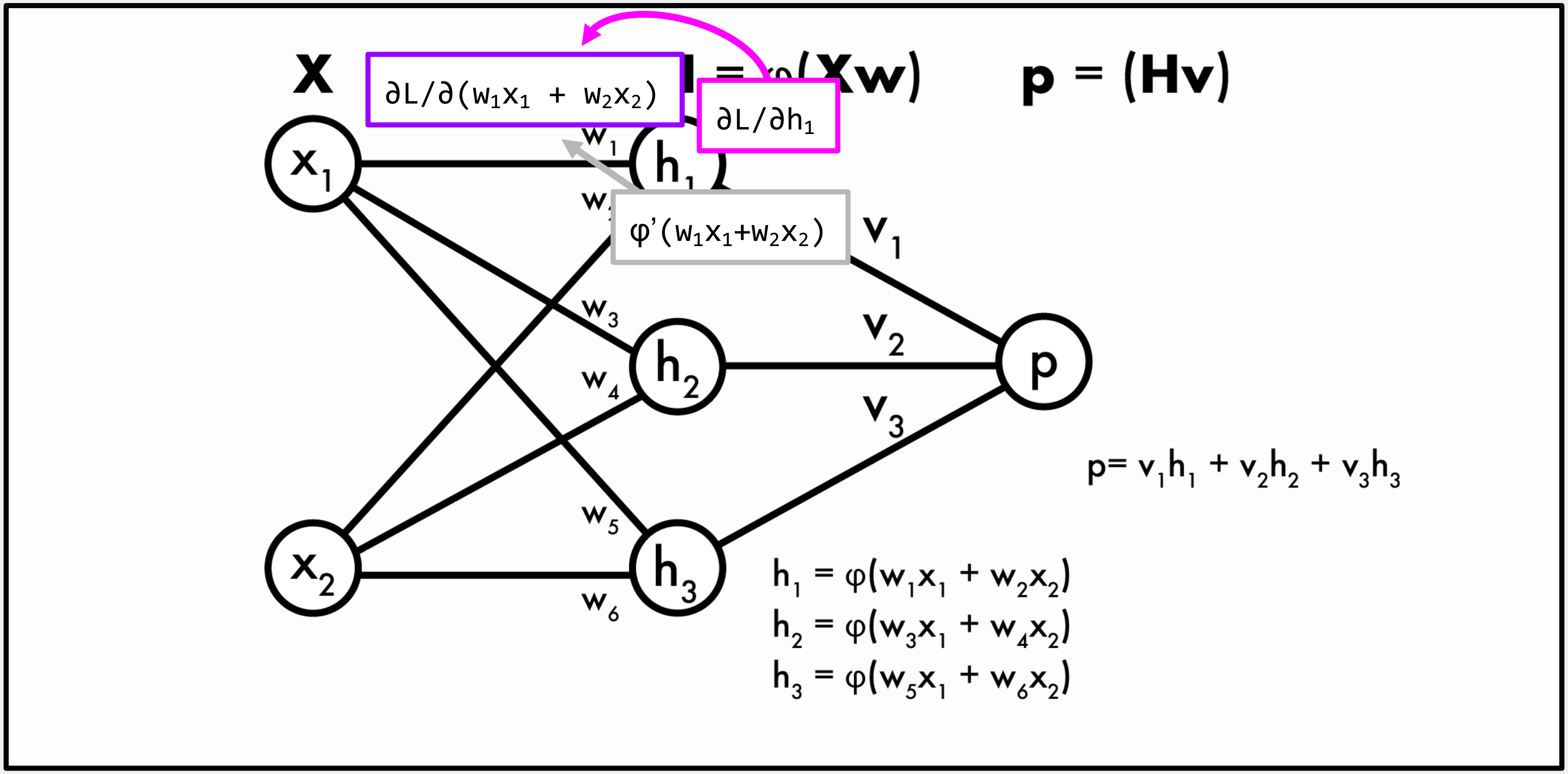
Backpropagation: the math



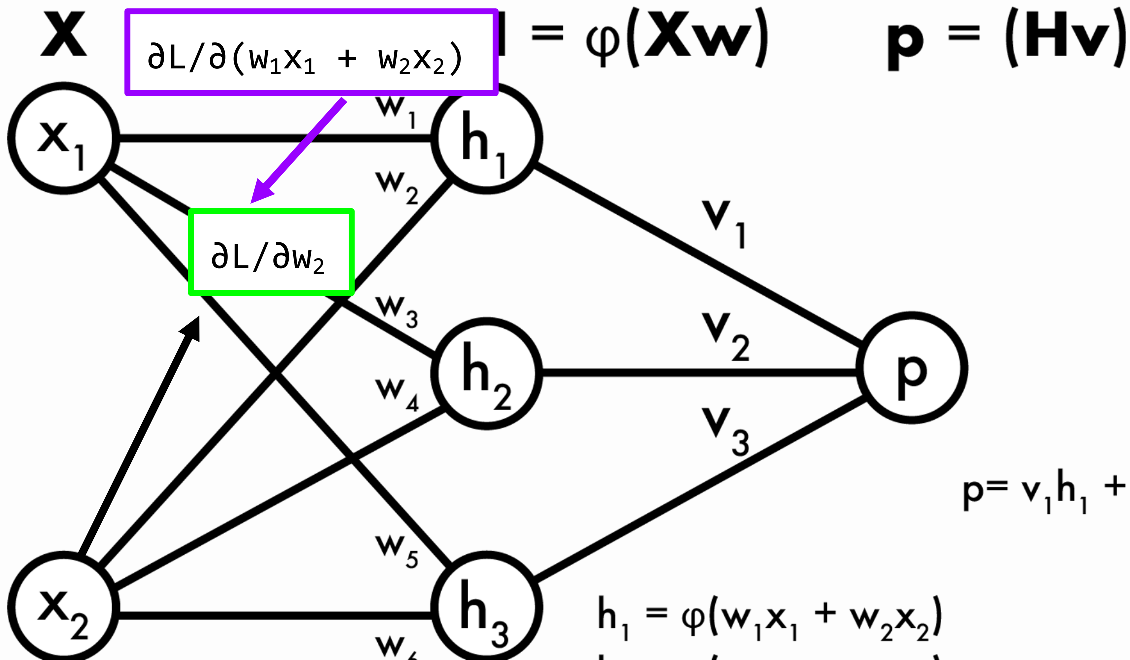
Backpropagation: the math



Backpropagation: the math



Backpropagation: the math



$p = v_1h_1 + v_2h_2 + v_3h_3$

$h_1 = \varphi(w_1x_1 + w_2x_2)$
$h_2 = \varphi(w_3x_1 + w_4x_2)$
$h_3 = \varphi(w_5x_1 + w_6x_2)$

What if we have multiple classes?

What if we normalized logistic regression across classes?

Softmax!

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

If we have 2 classes and we assume $z_0 = 0$, $z_1 = \mathbf{w} \cdot \mathbf{X}$ then this is normal logistic regression.

Softmax Classifiers

Are great!

Softmax function:

$\sigma(\mathbf{x})_j = e^{x\_j} / \Sigma_k e^{x\_k}$

"Loss" is negative log-likelihood:

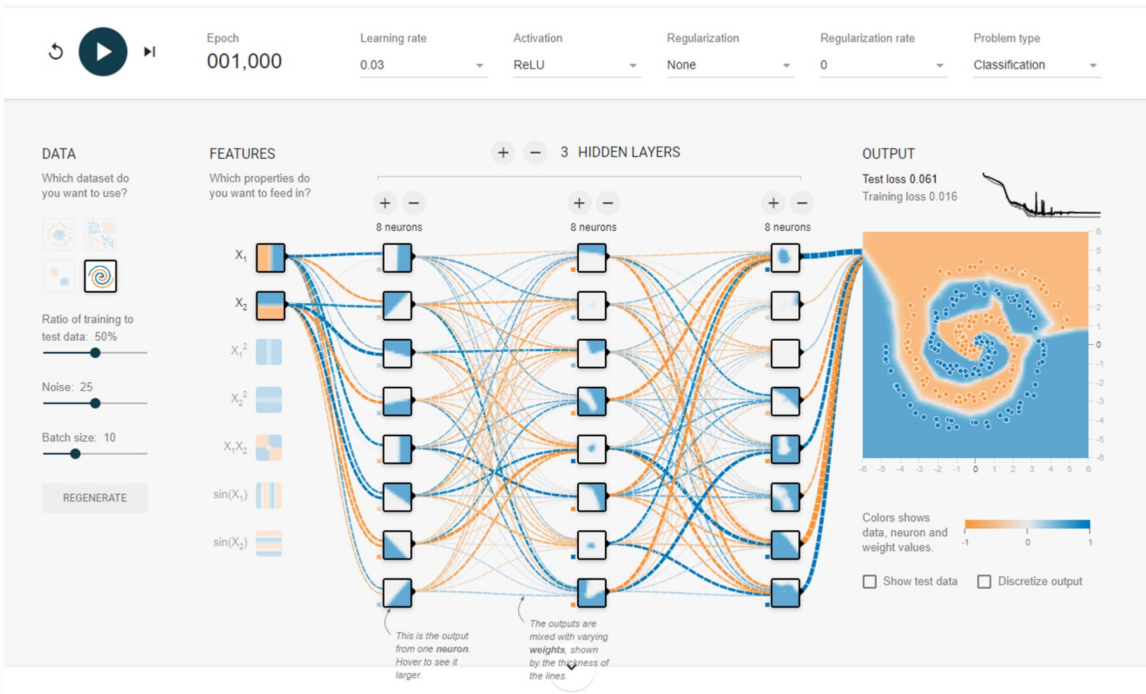
Data point has truth value y = [0,0,...,1,0,...,0]

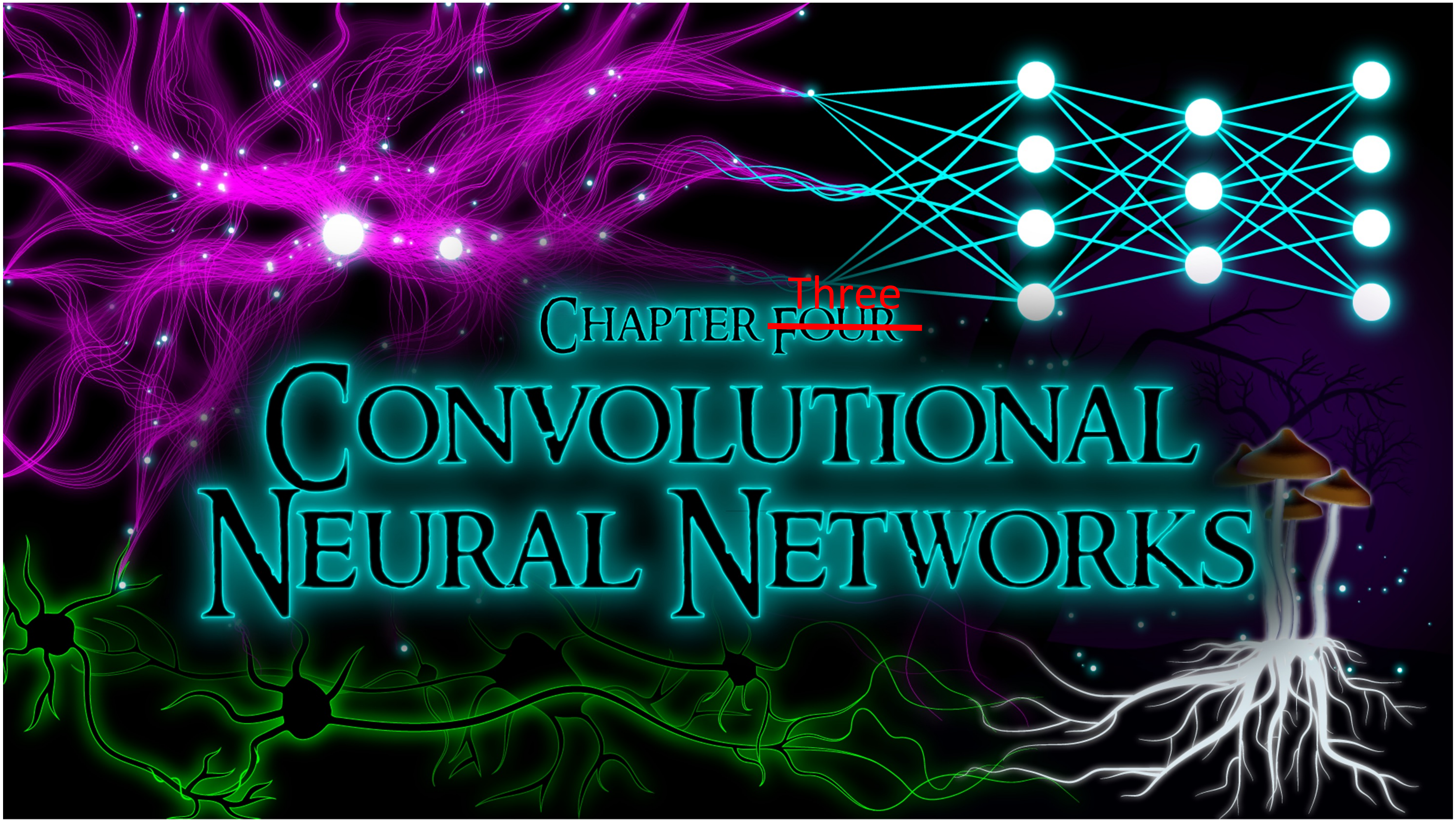
$L = -\Sigma_i y_i \log[\sigma(\mathbf{x})_i]$

And... $dL/d\mathbf{x} = y - \sigma(\mathbf{x})$ (just truth minus prediction)

Can read more here: https://knet.readthedocs.io/en/latest/softmax.html

Neural Network Playground




CHAPTER ~~FOUR~~ Three
CONVOLUTIONAL
NEURAL NETWORKS

Neural networks and images

Neural networks are densely connected

Each neuron in layer i connected to every neuron in layer i+1

Neural networks and images

Neural networks are densely connected

Each neuron in layer i connected to every neuron in layer i+1

Say we want to process images:

Input : 256 x 256 x 3 RGB image

Hidden : 32 x 32 x 36 feature map?

Output : 1000 classes

Neural networks and images

Neural networks are densely connected

Each neuron in layer i connected to every neuron in layer i+1

Say we want to process images:

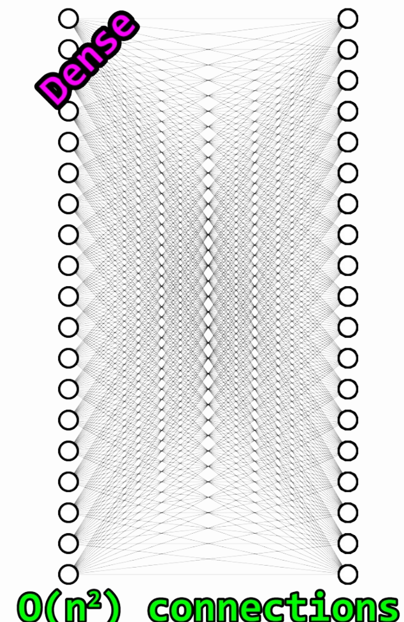
Input	: 256 x 256 x 3 RGB image
Hidden	: 32 x 32 x 36 feature map?
Output	: 1000 classes

Input -> hidden is 7.2 billion connections!

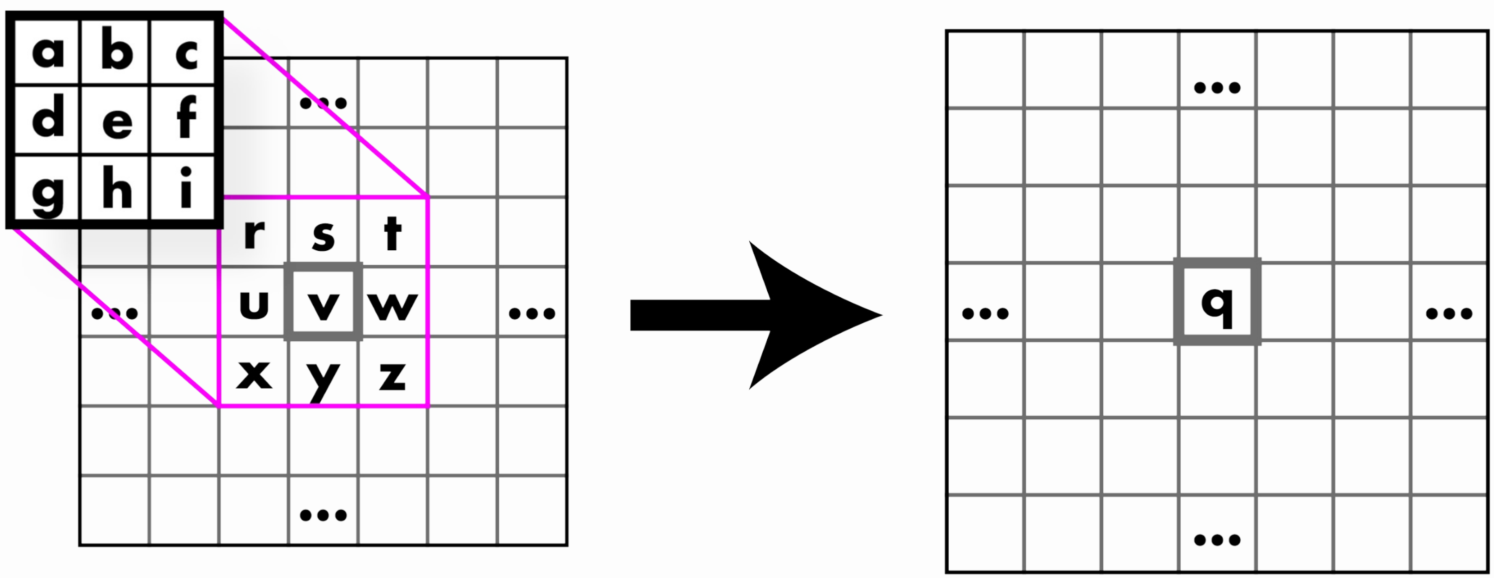
Too many weights!

Neural networks are densely connected

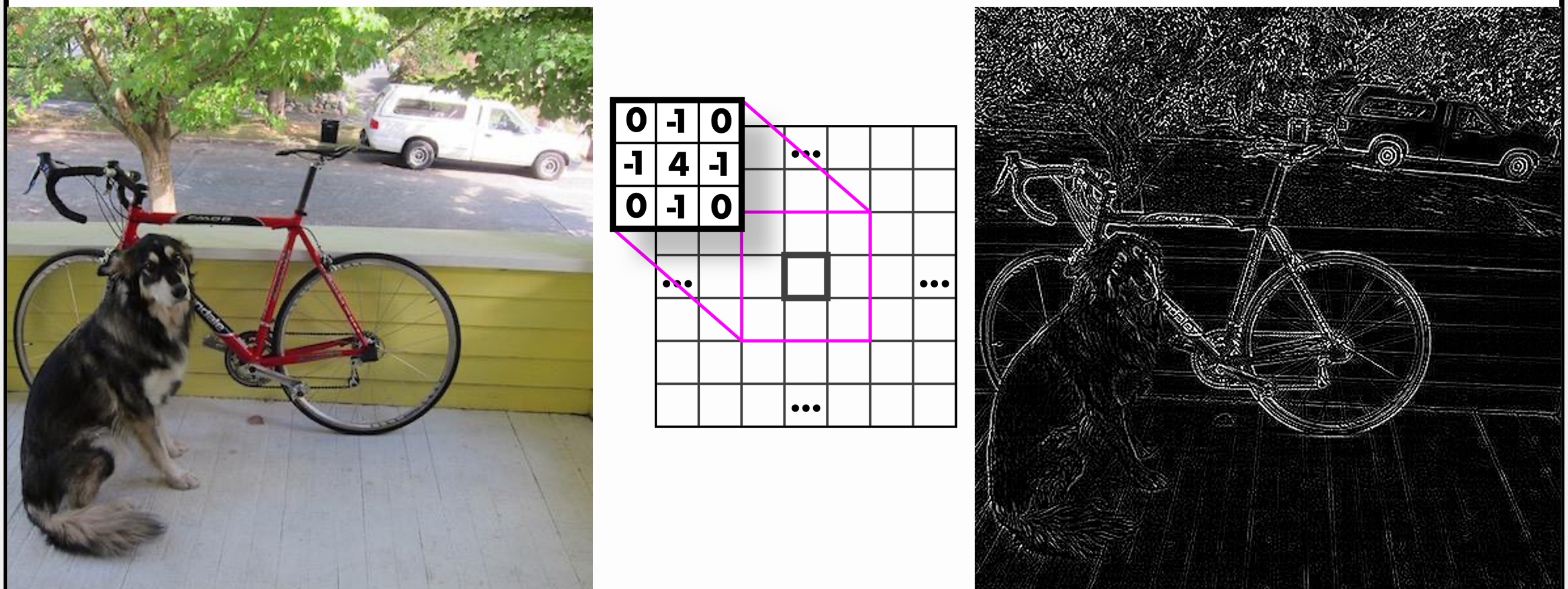
But is this really what we want when processing images?



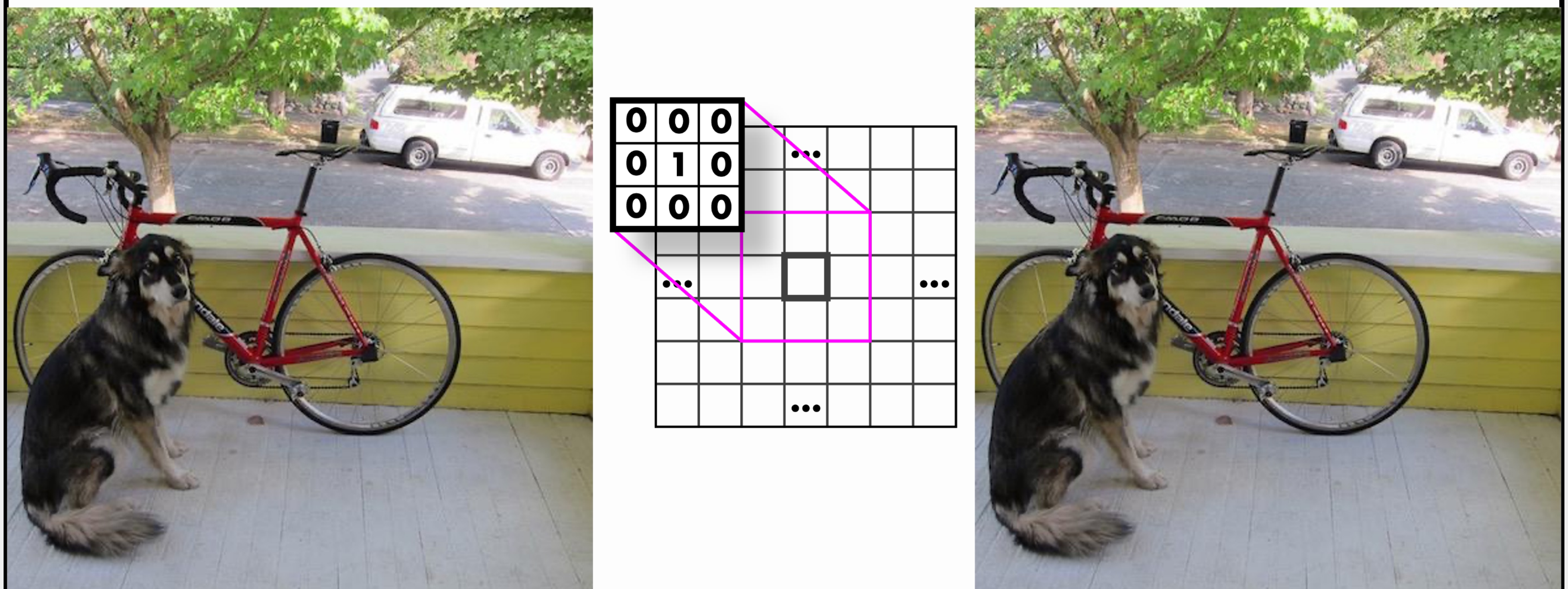
Convolutions?



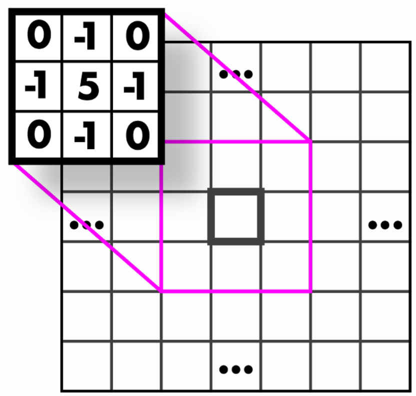
Highpass Kernel: finds edges



Identity Kernel: Does nothing!



Sharpen Kernel: sharpens!



Note: sharpen = highpass + identity!

So the situation is...

Neural Networks

- Can learn from d
- Learn to extract
  specific task
- Too many conne

Convolutions

eatures from images
d

Too many weights!

Would rather have sparse connections

Fewer weights

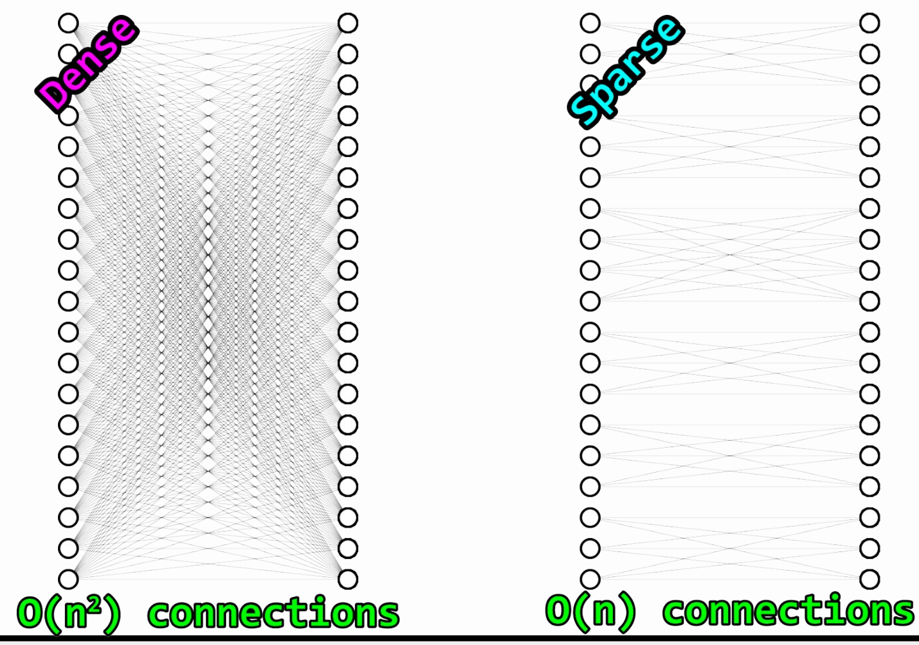
Nearby regions - related

Far apart - not related

Convolutions!

Just weighted sums of small areas in image

Weight sharing in different locations in image



Convolutional neural networks

Use convolutions instead of dense connections to process images

Takes advantage of structure in our data!

Imposes an assumption on our model:

- Nearby pixels are related, far apart ones are less related.
- Features in one part of the image are also useful in other parts.

Convolutional Layer

Input: an image

Processing: convolution with multiple filters

Output: an image, # channels = # filters

Output still weighted sum of input (with activation)

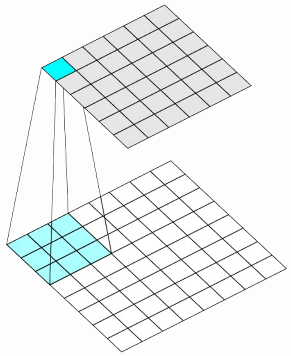
Kernel size

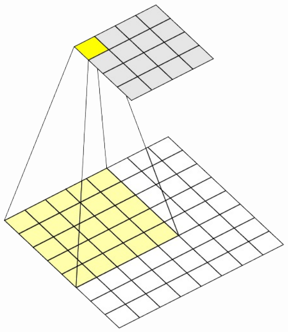
How big the filter for a layer is

Typically 1 x 1 <-> 11 x 11

1 x 1 is just linear combination of channels in previous image (no spatial processing)

Filters have same number of channels as input image.



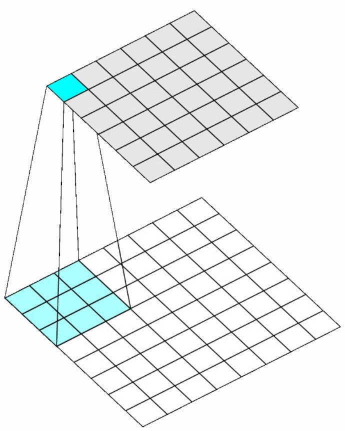


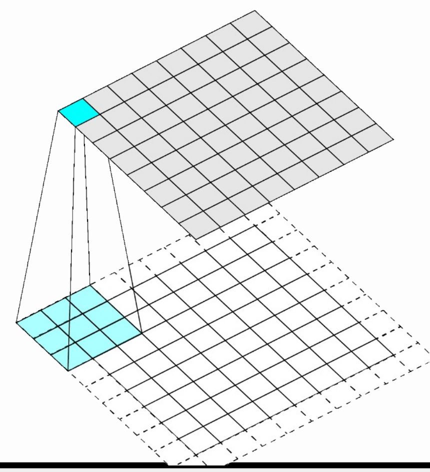
Padding

Convolutions have problems on edges

Do nothing: output a little smaller than input

Pad: add extra pixels on edge





Stride

How far to move filter between applications

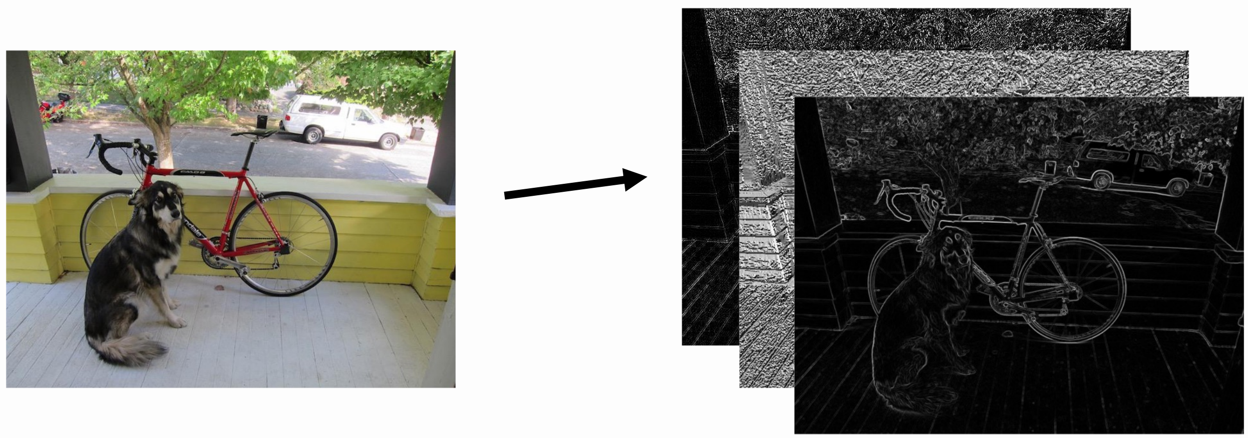
We’ve done stride 1 convolutions up until now, approximately preserves image size

Could move filter further, downsample image

Images are BIG

Even a 256 x 256 images has hundreds of thousands of pixels and that's considered a small image!

Convolution:



Aggregate information, maybe we don't need all of the image, can subsample without throwing away useful information

Pooling Layer

Input: an image

Processing: pool pixel values over region

Output: an image, shrunk by a factor of the stride

Hyperparameters:

- What kind of pooling? Average, mean, max, min
- How big of stride? Controls downsampling
- How big of region? Usually not much bigger than stride

Most common: 2x2 or 3x3 maxpooling, stride of 2

Maxpooling Layer, 2x2 stride 2

-7	6	-1	3	9	9	6	-9
3	-8	0	7	10	8	-3	10
-4	2	-6	4	-7	5	5	7
-3	-9	1	8	-8	9	-1	-5
-7	10	-9	-5	9	-8	-7	10
-5	5	9	4	10	-8	7	6
-3	8	0	2	2	-3	-2	5
4	-6	7	-3	1	4	10	0

Maxpooling Layer, 2x2 stride 2

-7	6	-1	3	9	9	6	-9
3	-8	0	7	10	8	-3	10
-4	2	-6	4	-7	5	5	7
-3	-9	1	8	-8	9	-1	-5
-7	10	-9	-5	9	-8	-7	10
-5	5	9	4	10	-8	7	6
-3	8	0	2	2	-3	-2	5
4	-6	7	-3	1	4	10	0

6			

Maxpooling Layer, 2x2 stride 2

-7	6	-1	3	9	9	6	-9
3	-8	0	7	10	8	-3	10
-4	2	-6	4	-7	5	5	7
-3	-9	1	8	-8	9	-1	-5
-7	10	-9	-5	9	-8	-7	10
-5	5	9	4	10	-8	7	6
-3	8	0	2	2	-3	-2	5
4	-6	7	-3	1	4	10	0

6			

Maxpooling Layer, 2x2 stride 2

-7	6	-1	3	9	9	6	-9
3	-8	0	7	10	8	-3	10
-4	2	-6	4	-7	5	5	7
-3	-9	1	8	-8	9	-1	-5
-7	10	-9	-5	9	-8	-7	10
-5	5	9	4	10	-8	7	6
-3	8	0	2	2	-3	-2	5
4	-6	7	-3	1	4	10	0

6	7		

Maxpooling Layer, 2x2 stride 2

-7	6	-1	3	9	9	6	-9
3	-8	0	7	10	8	-3	10
-4	2	-6	4	-7	5	5	7
-3	-9	1	8	-8	9	-1	-5
-7	10	-9	-5	9	-8	-7	10
-5	5	9	4	10	-8	7	6
-3	8	0	2	2	-3	-2	5
4	-6	7	-3	1	4	10	0

6	7		

Maxpooling Layer, 2x2 stride 2

-7	6	-1	3	9	9	6	-9
3	-8	0	7	10	8	-3	10
-4	2	-6	4	-7	5	5	7
-3	-9	1	8	-8	9	-1	-5
-7	10	-9	-5	9	-8	-7	10
-5	5	9	4	10	-8	7	6
-3	8	0	2	2	-3	-2	5
4	-6	7	-3	1	4	10	0

6	7	10	

Maxpooling Layer, 2x2 stride 2

-7	6	-1	3	9	9	6	-9
3	-8	0	7	10	8	-3	10
-4	2	-6	4	-7	5	5	7
-3	-9	1	8	-8	9	-1	-5
-7	10	-9	-5	9	-8	-7	10
-5	5	9	4	10	-8	7	6
-3	8	0	2	2	-3	-2	5
4	-6	7	-3	1	4	10	0

6	7	10	10

Maxpooling Layer, 2x2 stride 2

-7	6	-1	3	9	9	6	-9
3	-8	0	7	10	8	-3	10
-4	2	-6	4	-7	5	5	7
-3	-9	1	8	-8	9	-1	-5
-7	10	-9	-5	9	-8	-7	10
-5	5	9	4	10	-8	7	6
-3	8	0	2	2	-3	-2	5
4	-6	7	-3	1	4	10	0

6	7	10	10
2	8	9	7
10	9	10	10
8	7	4	10

Fully Connected Layer

The standard neural network layer where every input neuron connects to every output neuron

Often used to go from image feature map -> final output or map image features to a single vector

Eliminates spatial information

Convnet Building Blocks

Convolutional layers:

Connections are convolutions

Used to extract features

Pooling layers:

Used to downsample feature maps, make processing more efficient

Most common: maxpool, avgpool sometimes used at end

Fully Connected layers:

Often used as last layer, to map image features -> prediction

No spatial information

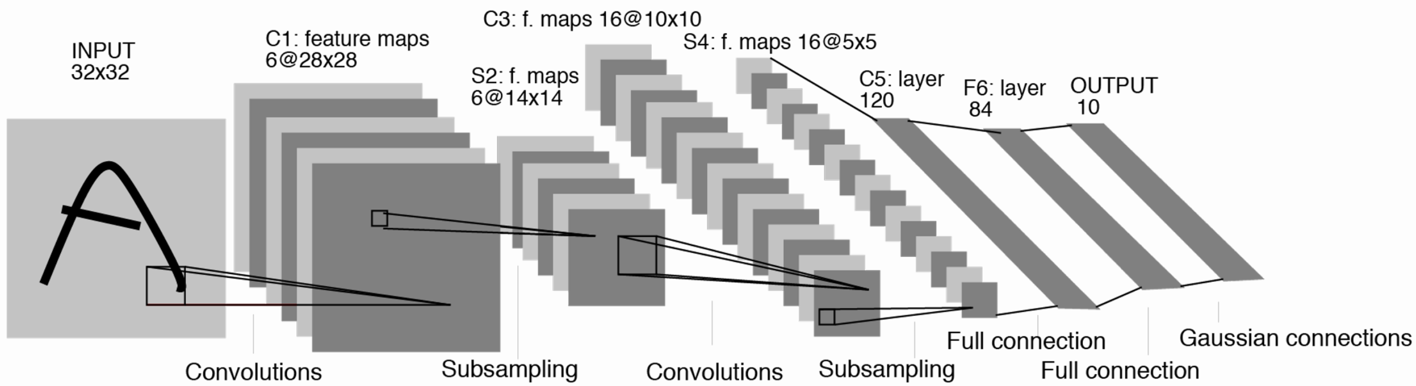
Inefficient: lots of weights, no weight sharing

LeNet: First Convnet for Images*

99% accuracy on MNIST (Yann LeCun 1998)

Has all elements of modern convnet

- Convolutions, maxpooling, fully connected layers
- Logistic activations after pooling layers (nowadays use RELU)
- Weight updates through backpropagation



*probably? Maybe neocognitron but not trained w/ backprop Fukushima, Kunihiko (1980). "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position" (PDF). Biological Cybernetics. 36 (4): 193–202.