Neuro 140/240 Tutorial 2

Morgan Talbot

Outline

1. Intro to Pytorch Dataset and Dataloader classes
2. Code walkthrough for Assignment 2
3. Building intuition for ML with neural networks
4. Survey of some common neural network types (CNNs, RNNs, autoencoders)

Part 1: Datasets and DataLoaders

For reference - a good introduction to these important Pytorch classes can be found here: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

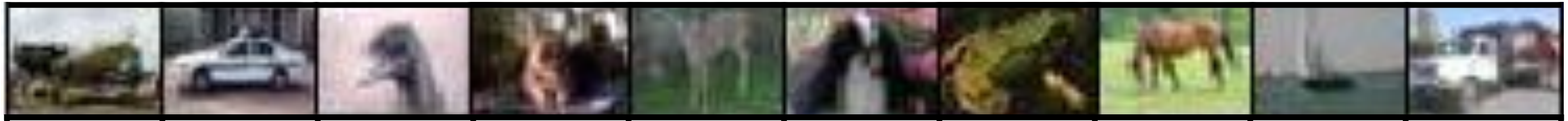
Pytorch Data**set** class

Any Data**set** class only needs three functions: “__init__”, “__len__”, and “__getitem__”.
__init__ is a constructor that sets up the mapping between images and labels, including how to access the images (e.g., storing directory paths to each image file)
__len__ returns the number of images in the Dataset
__getitem__ returns one image and one label when given an index (e.g. __getitem__(4) gets image number 4 in the Dataset, along with its label. This function may also apply transforms.

__getitem__ function

```
>>> my_dataset[2]
(<PIL Image>, 8)
```

Image:



Label:	7	4	8	2	5	2	9	5	1	3

__len__ function

```
>>> len(my_dataset)
10
```

__init__ function

sets up image files, labels, transforms, etc

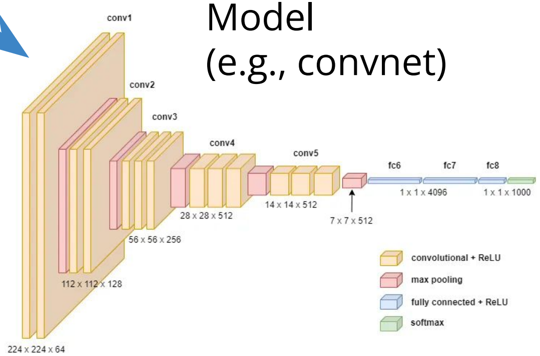
Pytorch Data**Loader** class (the Data**Loader** has a Dataset)

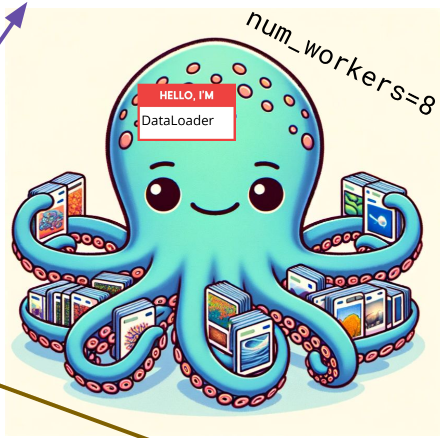
Transforms, such as:

- Format conversion
- Resizing
- Normalization
- Vertical/horizontal flips
- Random cropping

Feed into model for training

Model (e.g., convnet)





Batch
`batch_size=3`
`shuffle=True`

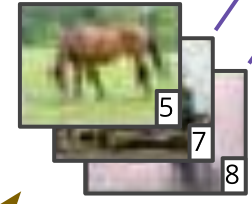
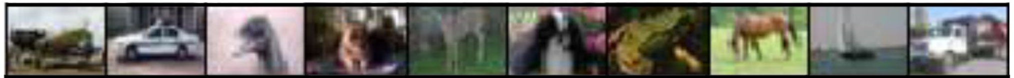


Image:



Label:	7	4	8	2	5	2	9	5	1	3

Data**set** instance

Pytorch Data**Loader** class (the Data**Loader** has a Data**set**)

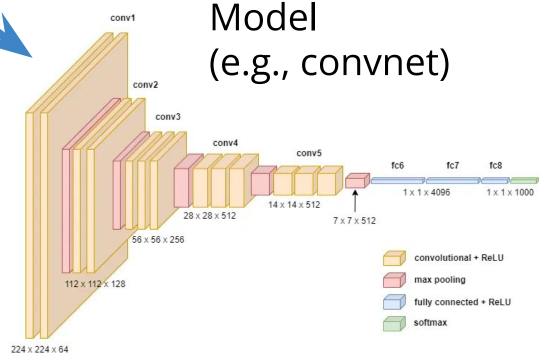
Transforms, such as:

- Format conversion
- Resizing
- Normalization
- Vertical/horizontal flips
- Random cropping

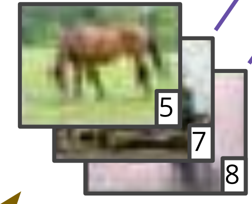
Note: transforms are typically applied within the Data**set**'s __getitem__ function, which the Data**Loader** calls to retrieve each image. It is also possible for the Data**Loader** to apply its own transforms.

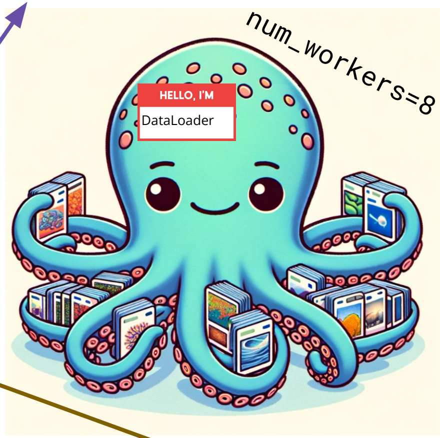
Feed into model for training

Model (e.g., convnet)



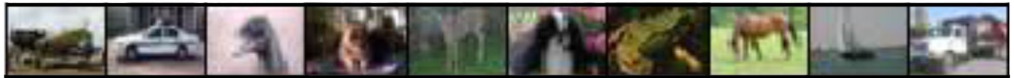
Batch
`batch_size=3`
`shuffle=True`





The Data**Loader** loads images in batches from the Data**set** which is passed to it upon initialization. It does so in parallel using many CPU thread "workers", which is important because the speed of training is often limited by loading the data rather than all the computations for training the network itself. num_workers=8 is a decent starting point.

Image:



Label:	7	4	8	2	5	2	9	5	1	3

Data**set** instance

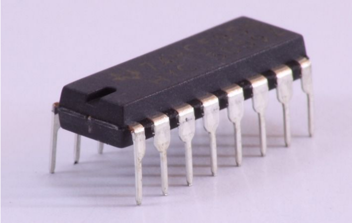
Part 2: Code walkthrough for Assignment 2

Part 3: Building intuition for ML with neural networks

List of key concepts

- Linear vs non-linear models
- Neural networks as function approximators
- Engineered vs learned features
- Basics of neural networks
- Backpropagation and gradient descent
- Overfitting
- CNNs
- RNNs
- Generative models

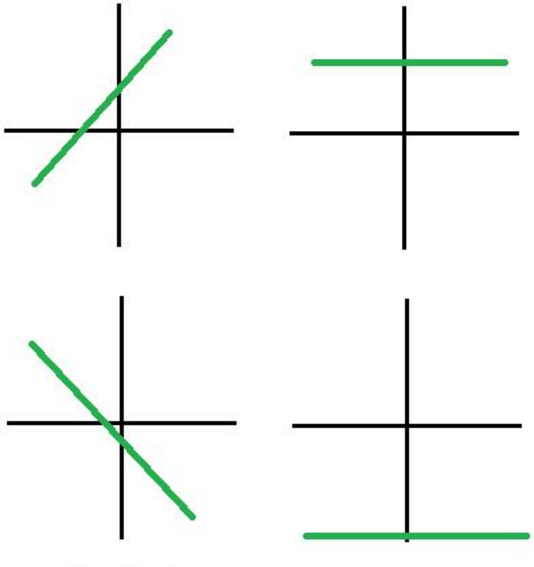
Features – aspects of input useful for the task

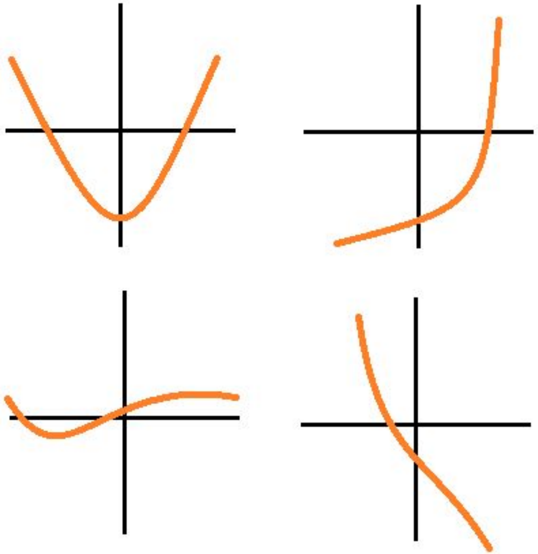



4 legs?

But then, how do you find legs in the image?

Pixabay, Flybar





Linear functions

$y = mx + b$

$y = c_1x_1 + c_2x_2 + \ldots + c_nx_n + b$

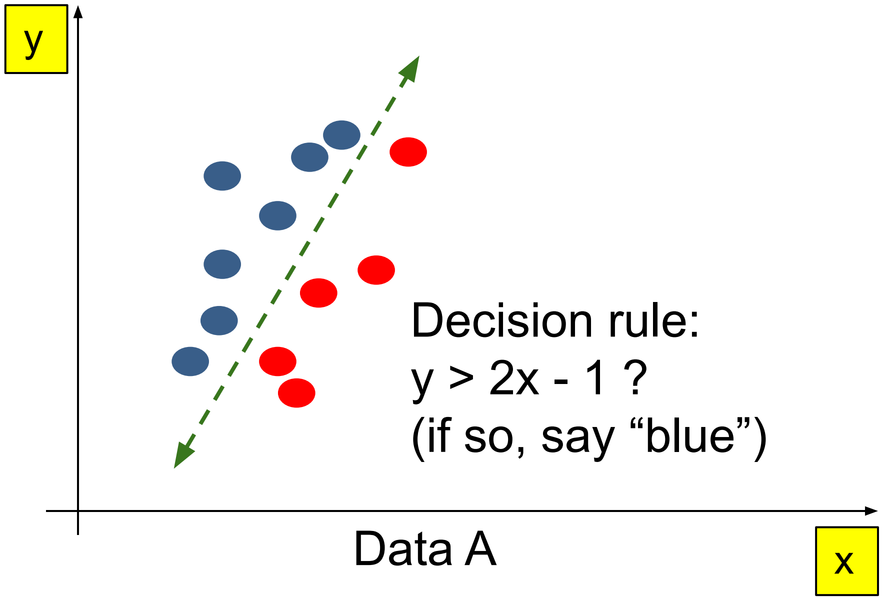
Non-linear functions

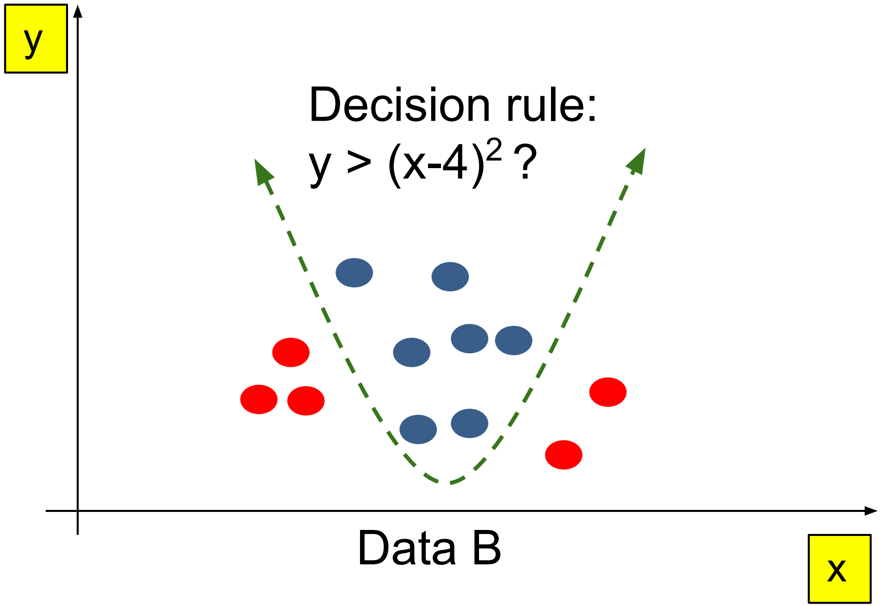
$y = x^2$

$y = c_1x_1^{\ 2} + \sin(x_2) + \ldots$

Linear and non-linear **classifiers**

How would you draw a line to separate the classes (red and blue) in these datasets?



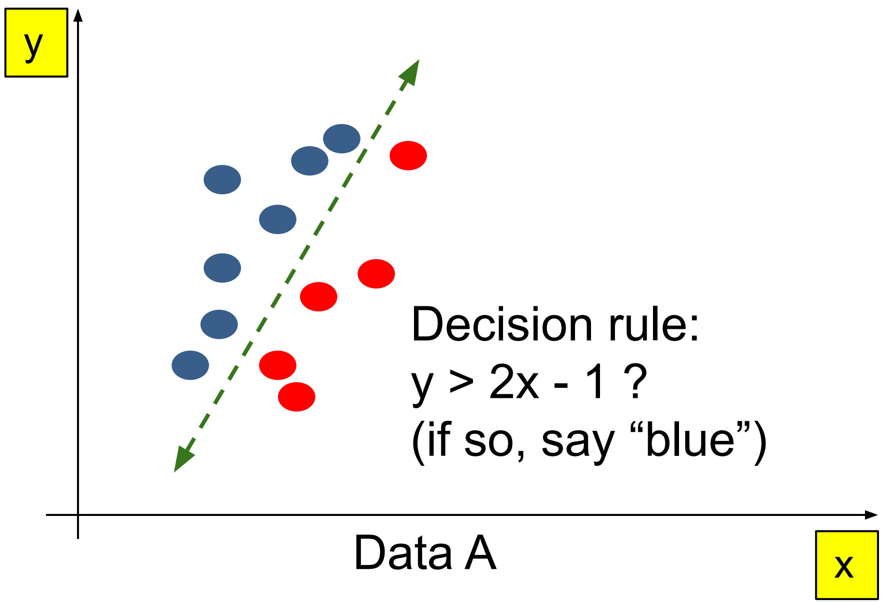


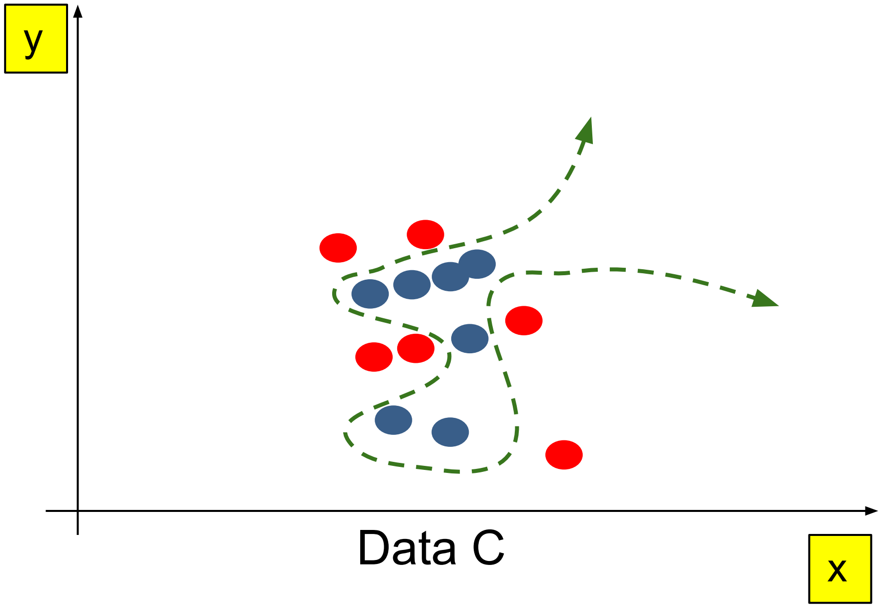
Recall: A <u>feature</u> is some aspect of the input that is useful for a task.

- For example, in the plots above, x and y are features of each dot!
- Given only x and y, we can classify the dots as “red” or “blue” using a decision rule

Linear and non-linear **classifiers**

How would you draw a line to separate the classes (red and blue) in these datasets?

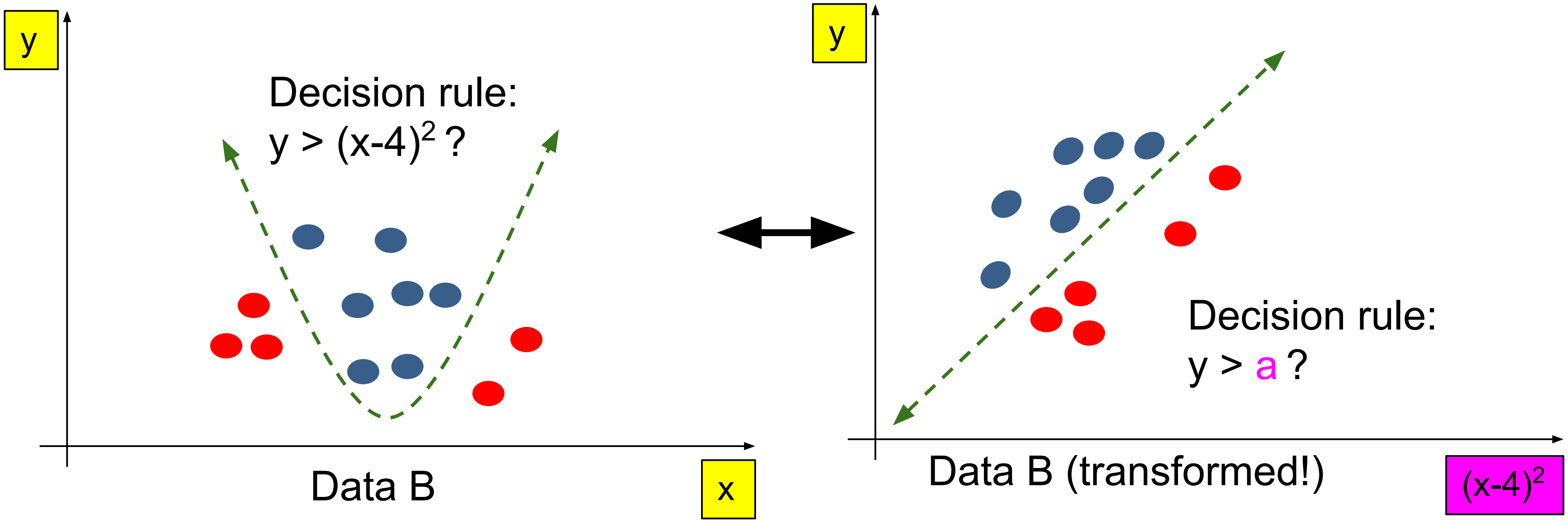




Recall: A <u>feature</u> is some aspect of the input that is useful for a task.

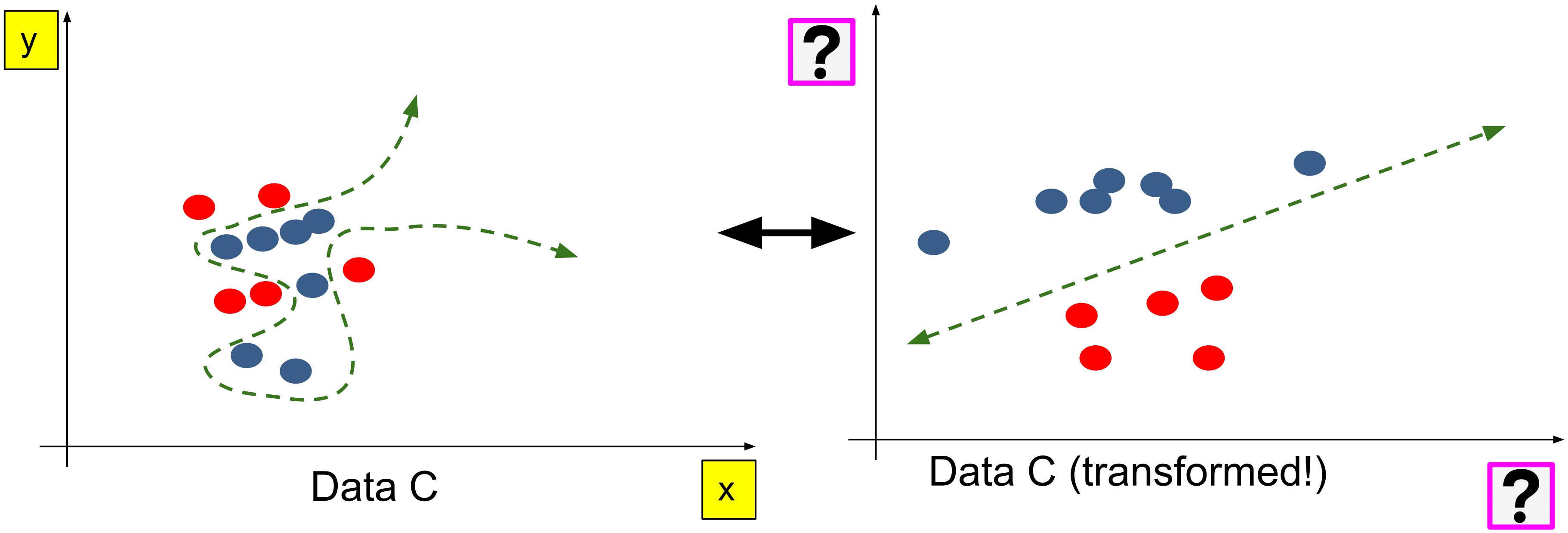
- For example, in the plots above, x and y are features of each dot!
- Given only x and y, we can classify the dots as “red” or “blue” using a decision rule

Linear and non-linear **classifiers**



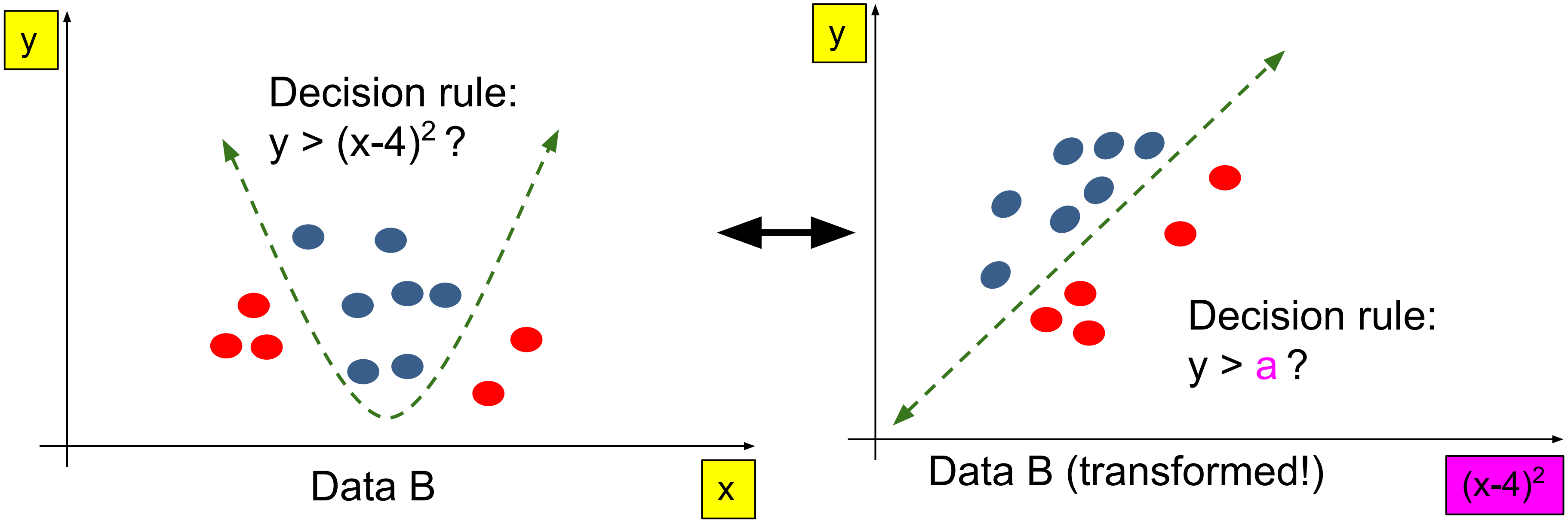
- **Idea:** transform the data so it becomes easy to classify with a straight line
- When we use $a = (x-4)^2$ as a <u>feature</u> (in addition to y), our decision rule is linear!

Linear and non-linear **classifiers**



- **Idea:** Can we find some non-linear features for data C, such that our final decision rule is linear?
- That is precisely what deep learning does!

Linear and non-linear **classifiers**



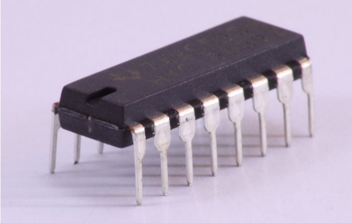
Making a linear function composed of linear functions results in… a linear function. For example:

$a = 2x + 3$, $b = 5x + 4$, $c = x + 1$

$3 \times a + 2 \times b + c + 1 = 16x + 19$

**You can think of this as a “linear combination” of functions

Features – aspects of input useful for the task



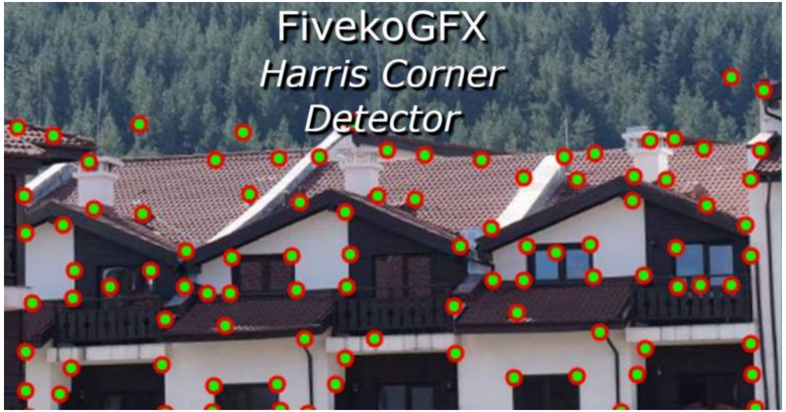
4 legs?

But then, how do you find legs in the image?

Pixabay, Flybar

Researchers have spent their entire careers finding useful features

Analytics Vidhya



Deep Learning: Let's **learn** the right features

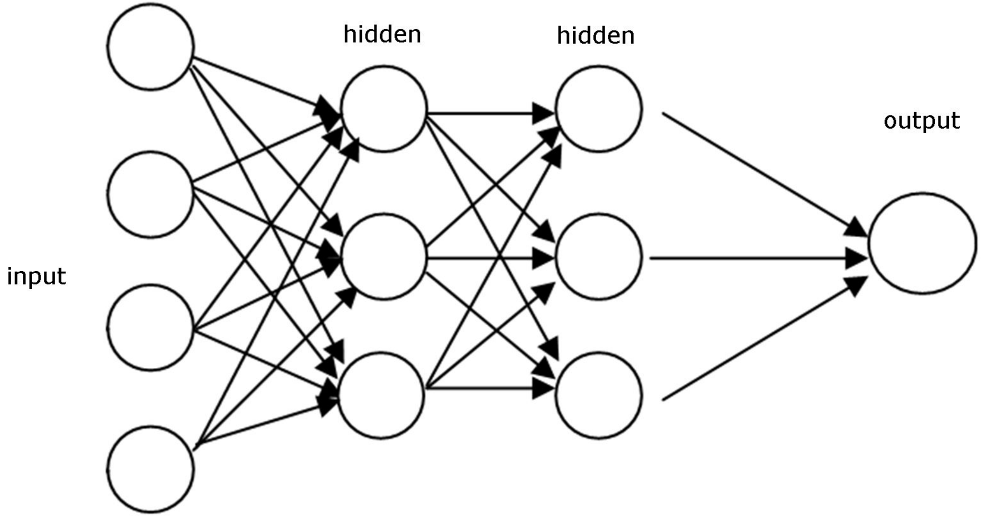
Learned features

Machine Learning System

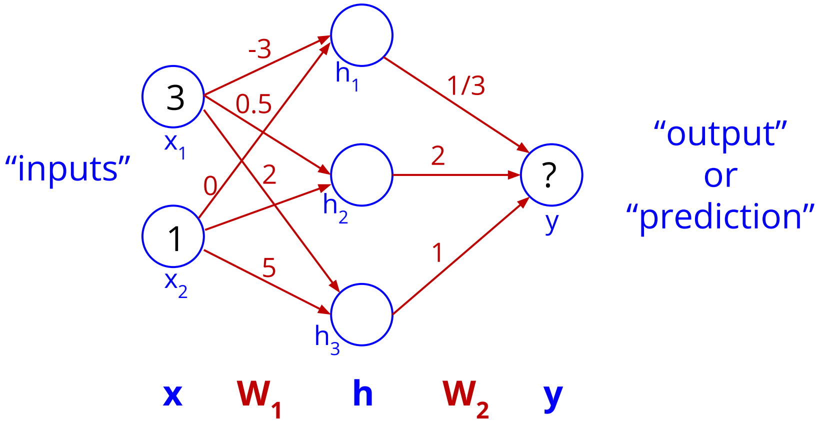
chair

Deep Learning System

Feed-forward neural networks: a foundation



Feed-forward neural networks: a foundation



Feed-forward neural networks: a foundation

The equation view:

$x_1 = 3$

$x_2 = 1$

$h_1 = -3x_1$

$h_2 = 0.5x_1 + x_2$

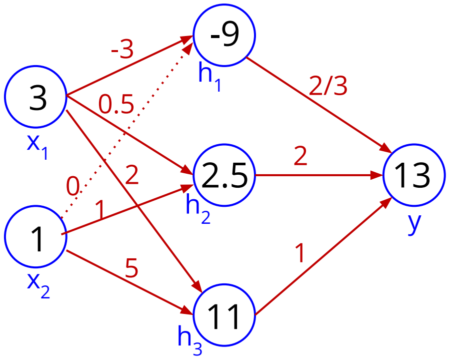
$h_3 = 2x_1 + 5x_2$

$y = 1/3h_1 + 2h_2 + 1h_3$

$= 1/3(-3x_1) + 2(0.5x_1 + x_2) + 1(2x_1 + 5x_2)$

The network view:

"inputs"



"output"
or
"prediction"

$\mathbf{x}$ $\mathbf{W_1}$ $\mathbf{h}$ $\mathbf{W_2}$ $\mathbf{y}$

Problem 1: we need y to be a probability between 0 and 1

Problem 2: y is still just a linear function of x!

The matrix-vector view:

$h = W_1x$

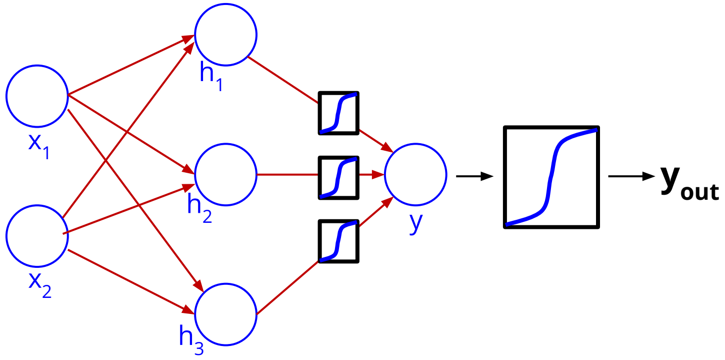
$y = W_2h$

$y = W_2(W_1x)$

Feed-forward neural networks: a foundation

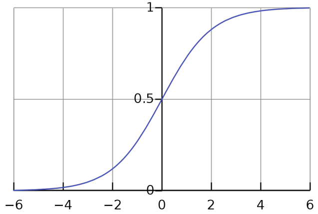
Problem 1: we need y to be a probability between 0 and 1

Problem 2: y is still just a linear function of x!



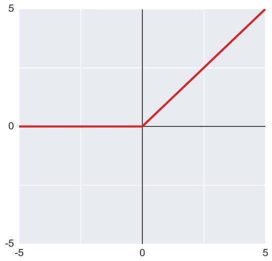
Solution 1: a sigmoid activation function

y_{out} = sigmoid(y) = $e^y/(1+e^y)$

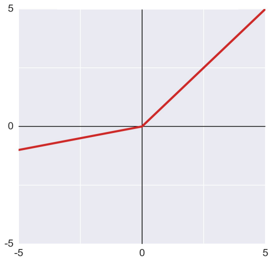


Solution 2: Let's put non-linear activation functions on h_1, h_2, and h_3 as well! (e.g. sigmoid, ReLU)

ReLU



Leaky ReLU



Universal Approximation Theorem:

A feed-forward network with 1 or more hidden layers, enough neurons, and a non-linear activation function for each neuron can approximate ANY continuous function with ANY degree of accuracy.

For example: probability of being a cat = f(pixels)

We have our network. How does it learn?

Gradient descent:

1. Define a "loss function", such as: $(y_{out} - y_{label})^2$

Intuition: the loss measures the network's "badness"

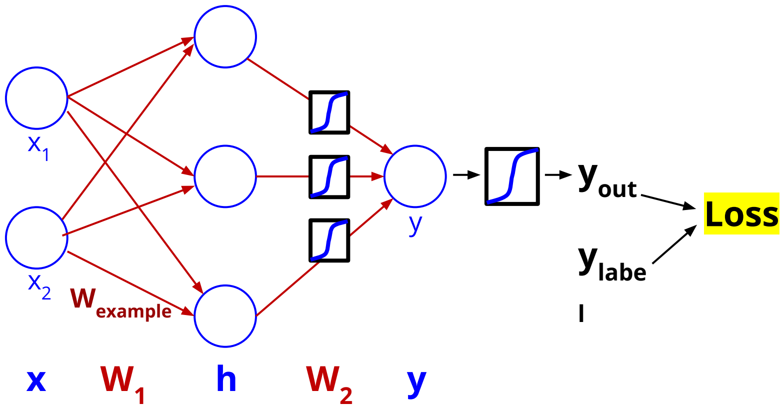
2. Calculate the partial derivative of the loss with respect to each parameter. These partial derivatives are called *gradients*

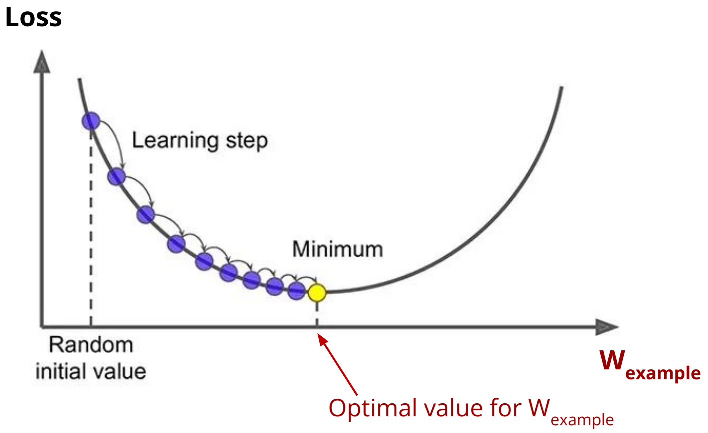
Intuition: if I increase parameter $\mathbf{W_{example}}$ by a tiny bit, does the loss go up or down? (by how much?)

3. Do this for every parameter: subtract the parameter's *gradient* times the *learning rate* (a small value, e.g. 0.001) from the parameter's original value, and set this as the parameter's new value.

Intuition: if increasing $\mathbf{W_{1:(2,3)}}$ improves loss, then increase it!
If it worsens the loss, decrease it! If no effect, do nothing.

4. Repeat 2 and 3 many times, using many different data points, to reduce the loss!





Original loss image: Rishi at GoPenAI

We have our network. How does it learn?

Gradient descent:

1. Define a "loss function", such as: $(y_{out} - y_{label})^2$

Intuition: the loss measures the network's "badness"

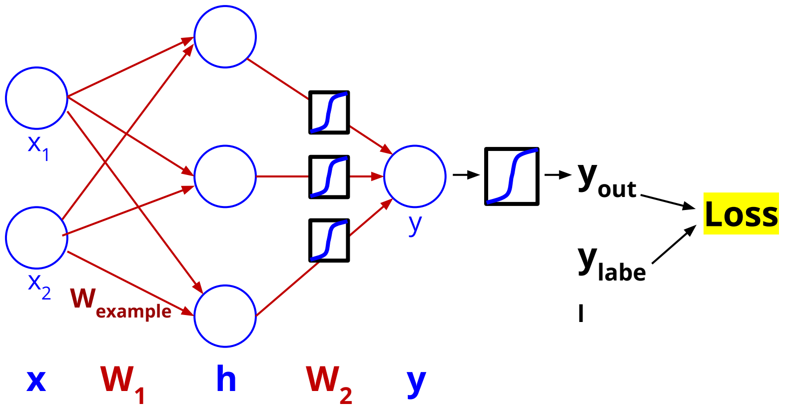
2. Calculate the partial derivative of the loss with respect to each parameter. These partial derivatives are called *gradients*

Intuition: if I increase parameter $\mathbf{W}_{\mathbf{example}}$ by a tiny bit, does the loss go up or down? (by how much?)

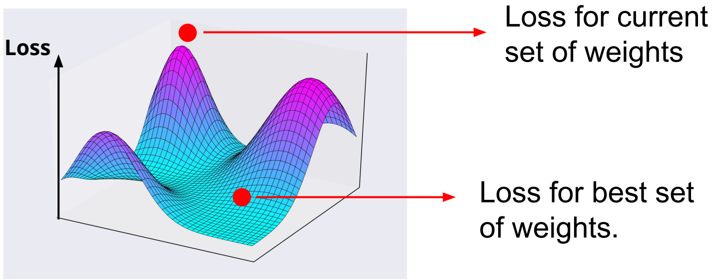
3. Do this for every parameter: subtract the parameter's *gradient* times the *learning rate* (a small value, e.g. 0.001) from the parameter's original value, and set this as the parameter's new value.

Intuition: if increasing $\mathbf{W}_{\mathbf{1:(2,3)}}$ improves loss, then increase it! If it worsens the loss, decrease it! If no effect, do nothing.

4. Repeat 2 and 3 many times, using many different data points, to reduce the loss!



We are actually doing this for all of the weights simultaneously, taking a series of "steps" in a high-dimensional "loss landscape" to find the lowest point. We process a randomly-chosen "batch" of inputs and labels to determine gradients at each step (stochastic gradient descent)



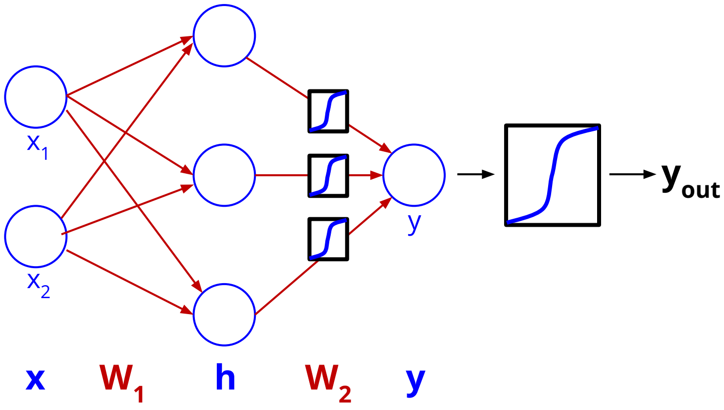
We have our network. How does it learn?

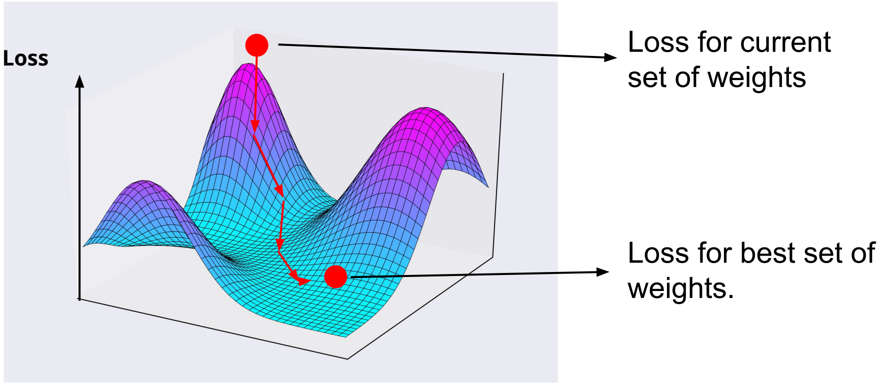
Backpropagation:

Just a method for finding the gradients, using the chain rule from calculus.

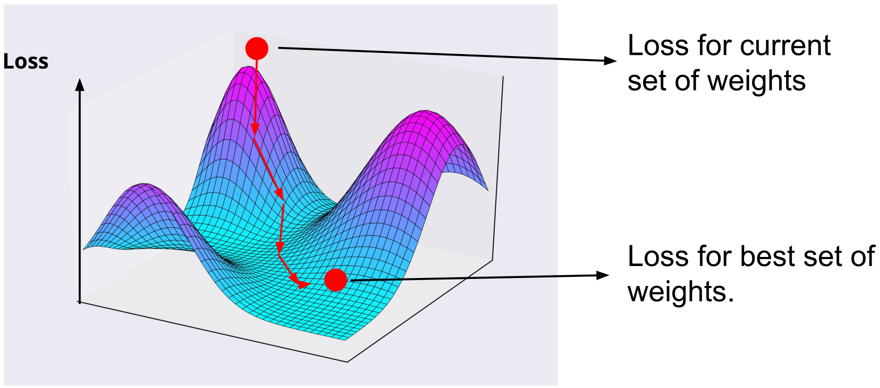
<u>Intuition:</u> the gradients for weights near the output end (where loss is calculated) are easy to find. Once you have these gradients, you can use them to find the gradients of the next earliest layer - continue "propagating" the gradients backwards until you arrive at the earliest set of weights.

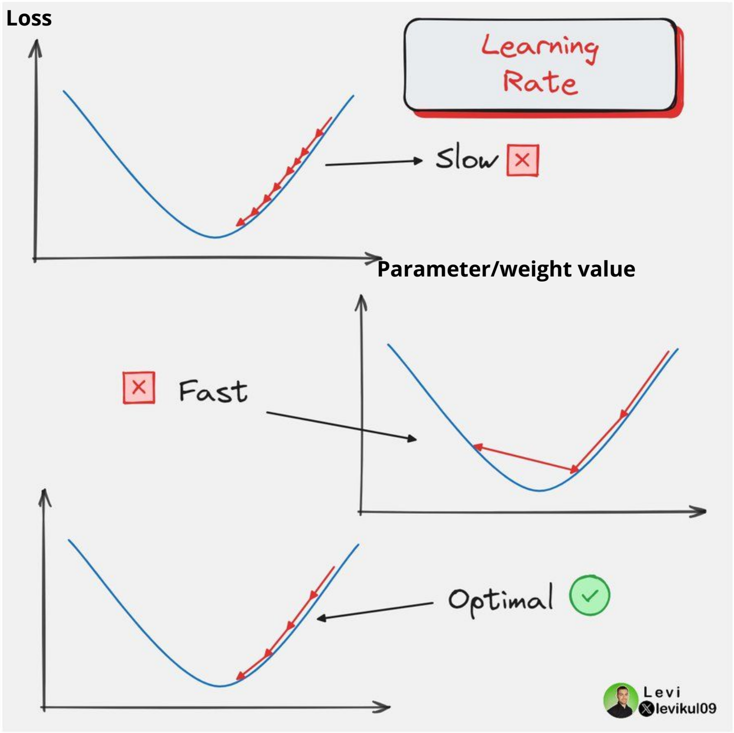
Here, you would first find the gradients for each value in w_2, then use those to find the gradients for w_1.





Learning rate: a very important "hyperparameter"



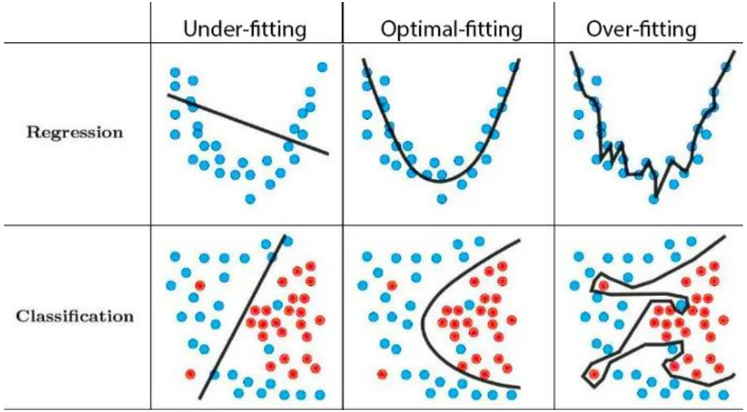


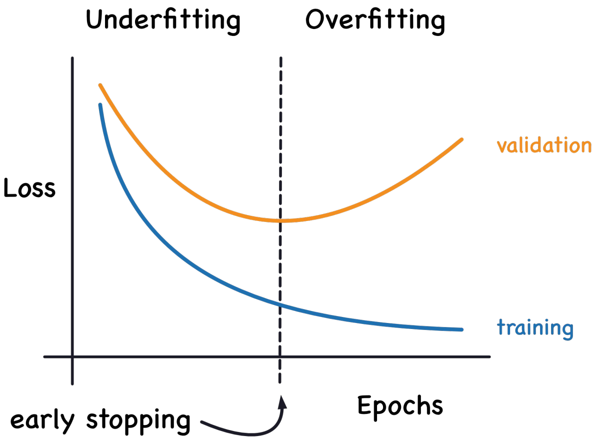
The learning rate, together with the gradients, determines the overall size of our steps in the loss landscape.

Small learning rate: **slow**, (+ tend to get stuck in local minima)

Large learning rate: **unstable**

A neural network can (in theory) approximate any **function**, which it learns from (almost never enough) **data**.





How to fix **under**fitting

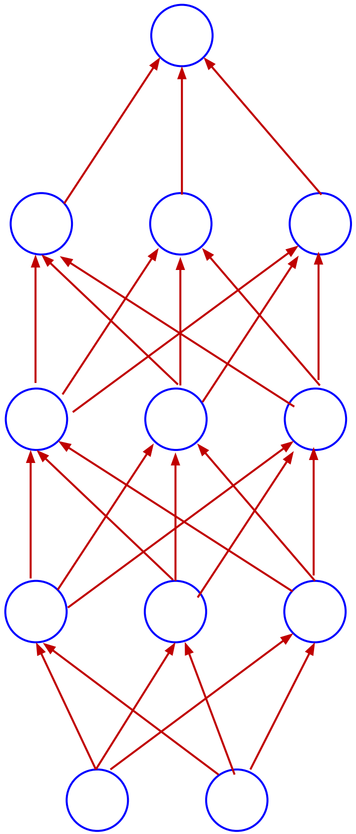
- Bigger model - more neurons, more layers, etc (in practice, this makes it easier to fit complex functions)
- Train for more epochs

How to fix **over**fitting:

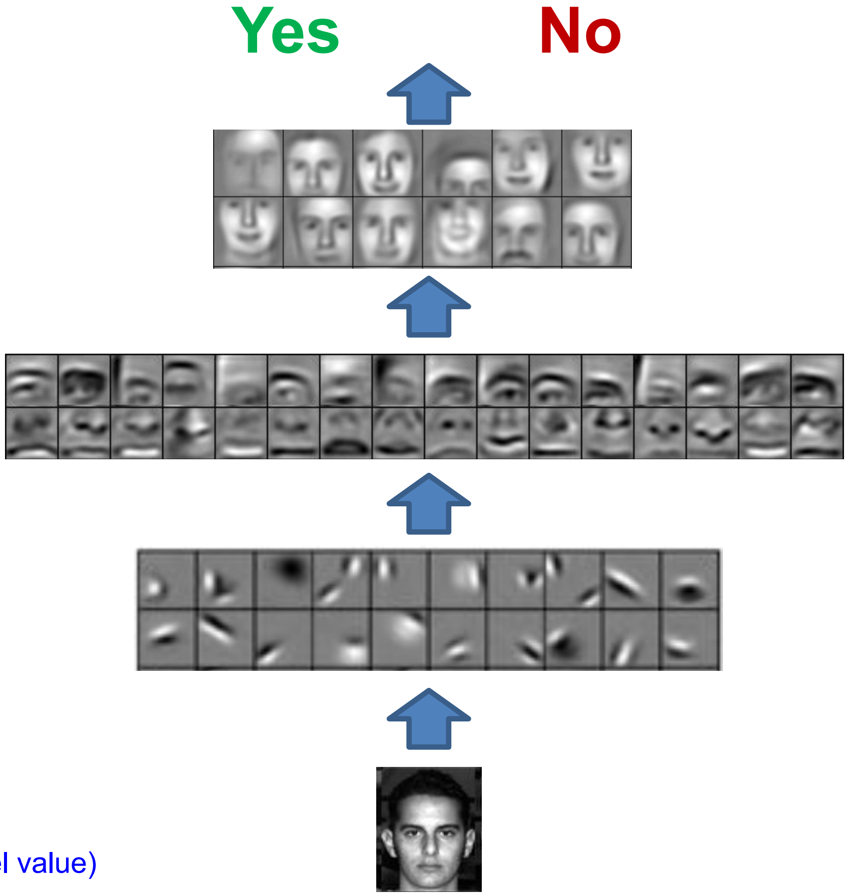
- Simpler model
- Reduce training epochs ("early stopping")
- Regularize (e.g., dropout, adding an L_2 regularizer term to the loss)
- **Add more data**

Fast.ai, Ryan Holbrook (L to R)

Multiple layers can learn increasingly complex features



input (1 neuron for each pixel value)



Classification (is it a face?)

high level features

medium level features

low level features

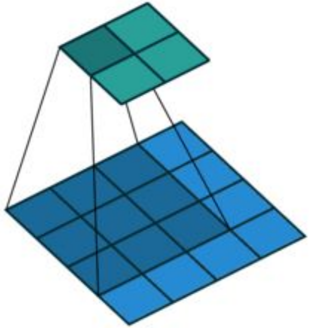
input image

[Honglak Lee]

End of tutorial recording here

Convolutional Neural Networks (CNNS)

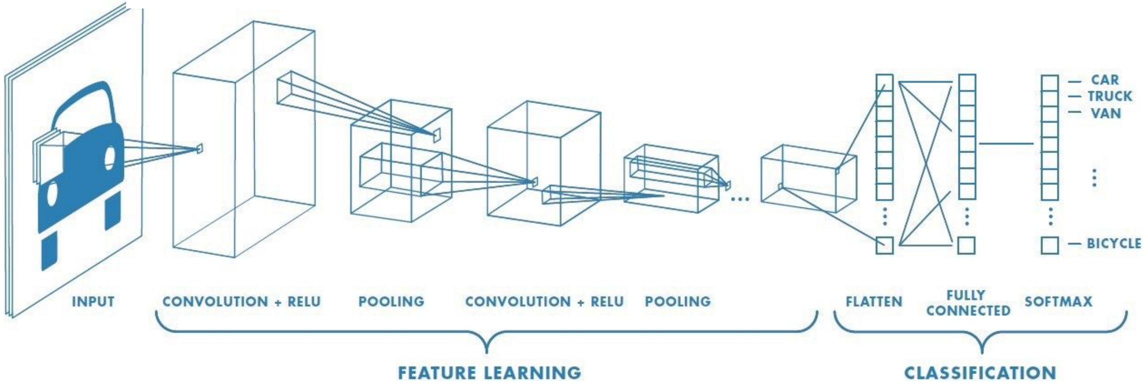
Before, we turned an image matrix/tensor into one long vector. But images have 2D structure - let's exploit that!



Visualization of a convolution operation

Legend:
Blue grid = input image
Dark blue shadow = "kernel"
Green grid = activation values of second layer

Example kernels. Each kernel is a feature detector!

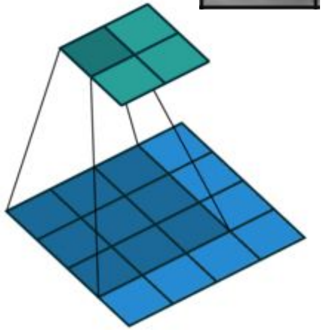


One convolution produces a new "image" that encodes *features* instead of *pixels*. We can apply another convolution to this!

Convolutional Neural Networks (CNNS) (slide version 2, more verbose)

Before, we turned an image matrix/tensor into one long vector. But images have 2D structure - let's exploit that!

In the visualization (right), the blue grid is an input image. The darker blue shadow is a "kernel" - we apply the same neural network function to each patch of the image. The green grid represents neurons in the second layer of the network.

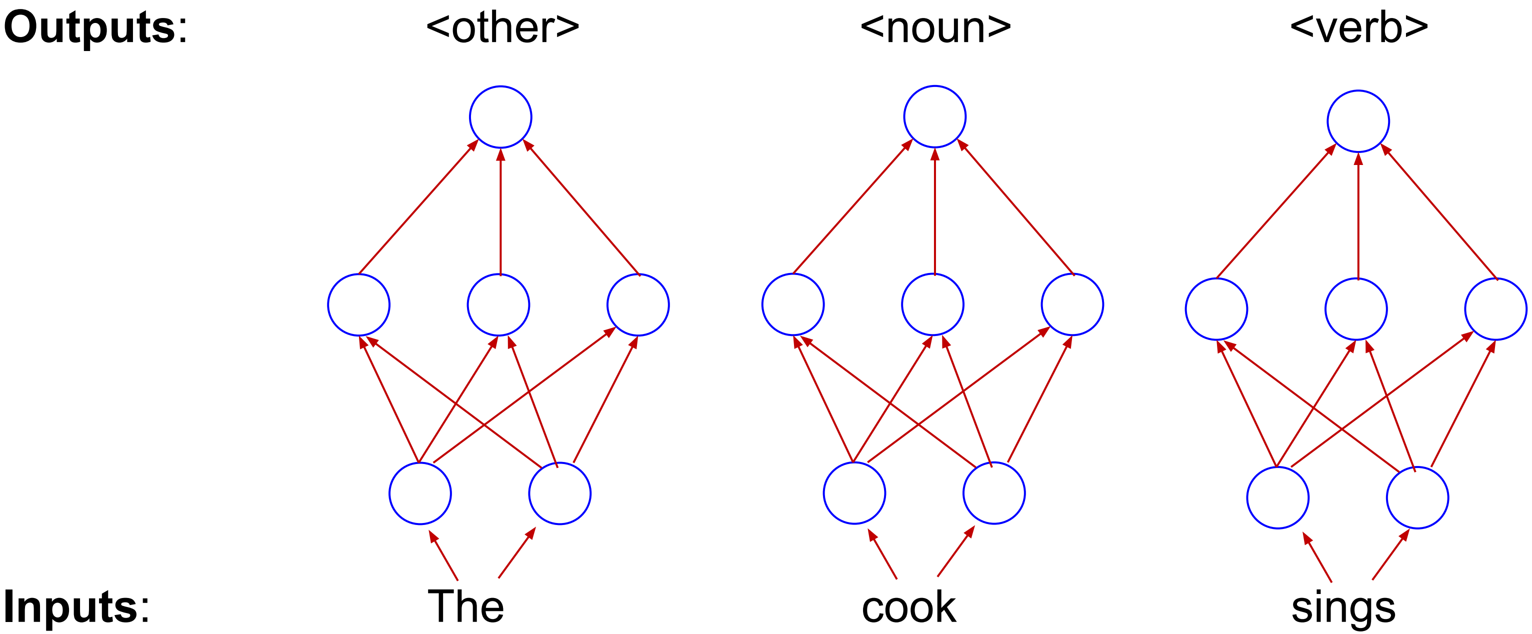


Visualization of a convolution operation

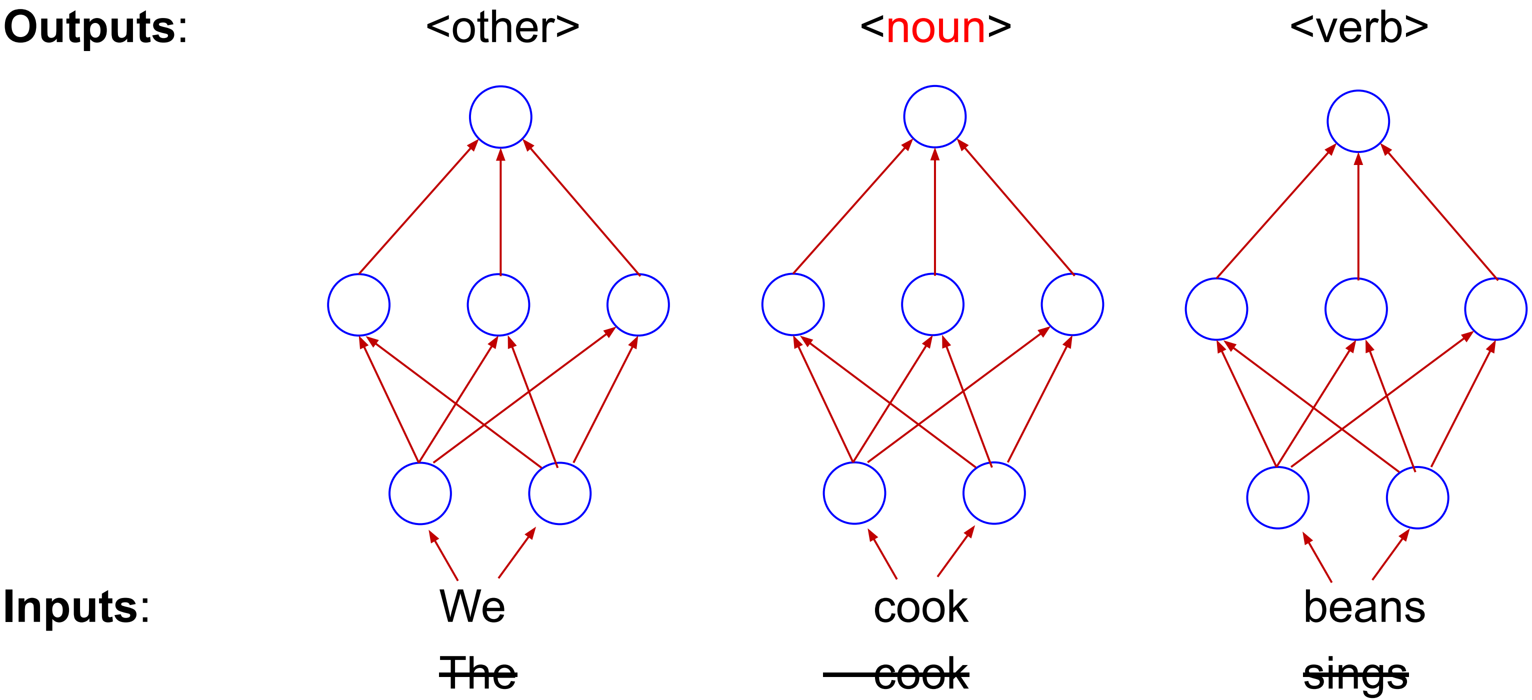
Intuition: let's say we have a neuron in our second layer that recognizes eyes. Convolution allows this neuron to find eyes anywhere in the image!

We apply many different "kernels**" (each of which may detect a specific **feature**), to every patch of the image. Eye detector, ear detector, banana detector, etc…)

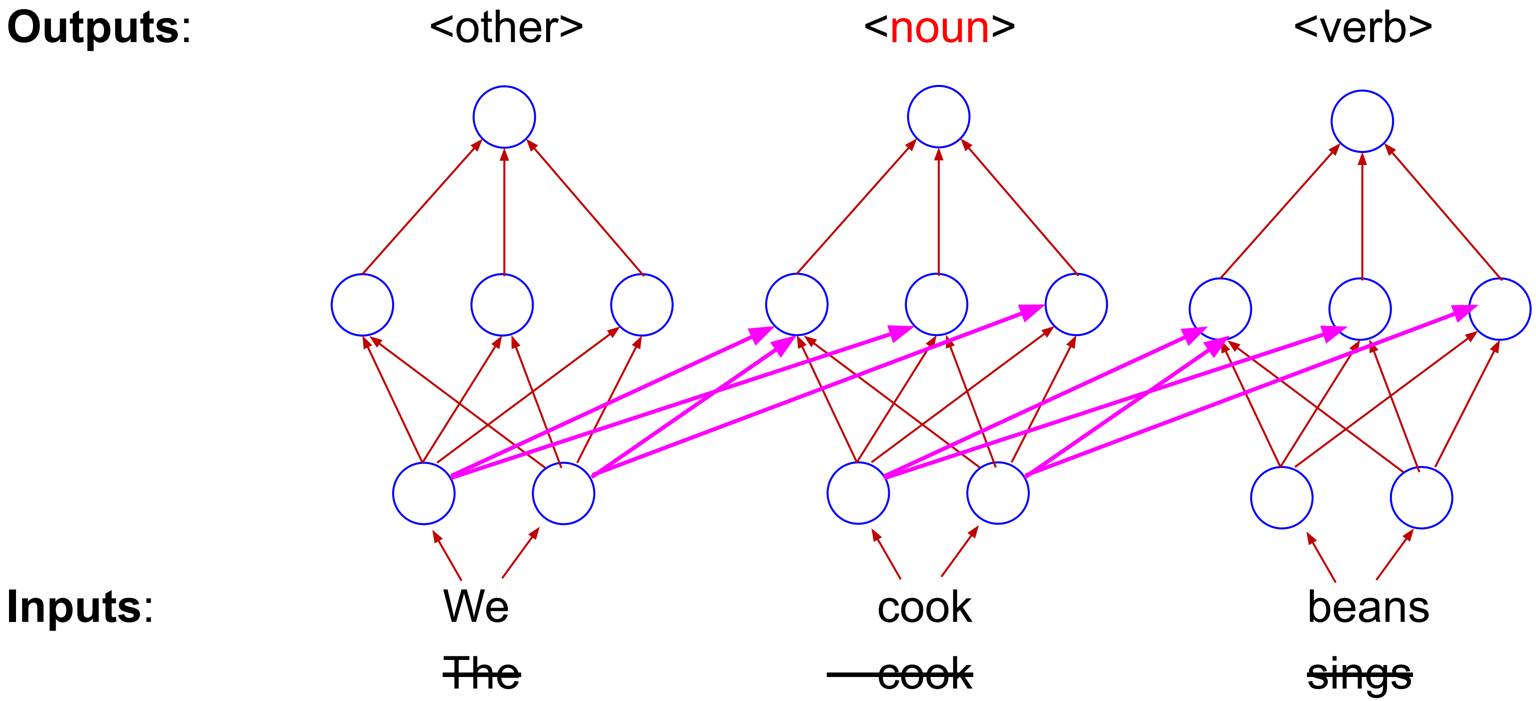
Feed-forward nets are not ideal for data with a sequential structure



Feed-forward nets are not ideal for data with a sequential structure



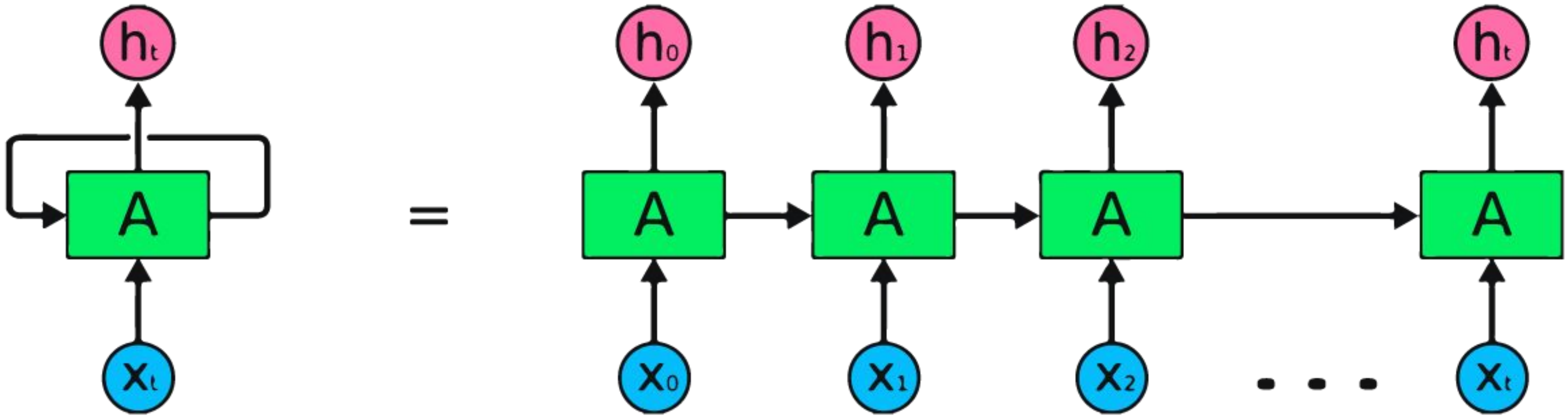
Idea behind recurrent neural networks (RNNs): save some activations and feed them back into the network at the next time step



Recurrent Neural Networks (RNN) intuition

1. RNNs maintain a state (here, “A”)
2. The current state is combined with the next input in the sequence to produce the next state.
3. Each state is used to make some kind of prediction at time t

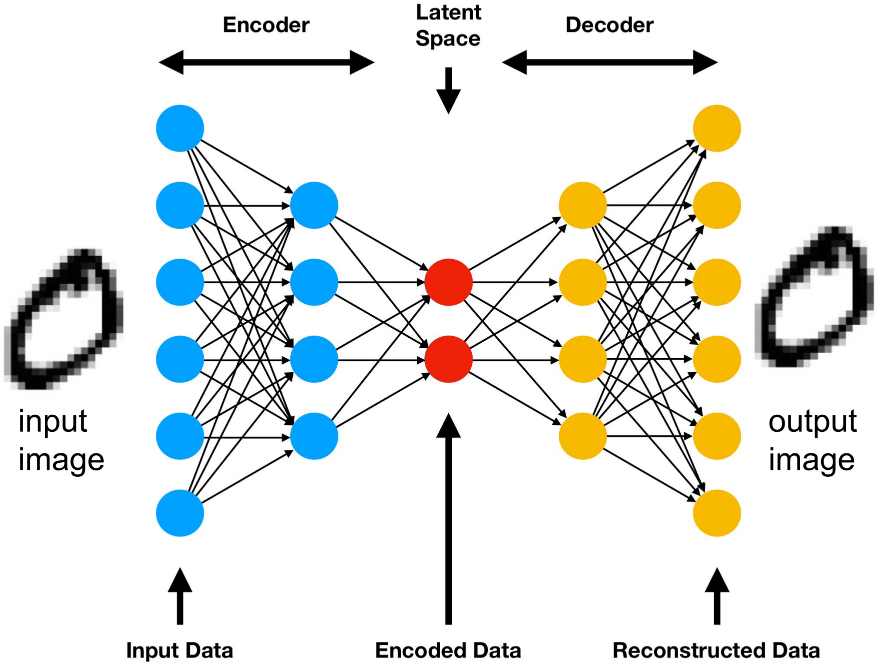
Note: each black arrow here represents an entire feed-forward network! (at least the vertical ones)



Generative models

https://thispersondoesnotexist.com

Generative models: autoencoders



Input: A picture in our dataset

Output: The same picture, reconstructed (we use the original image as our “label”)

Loss:
mean squared error of pixel values
<u>Intuition:</u> the more different the images are from each other, the higher the loss

We have our inputs, outputs, network model, and loss. *Everything we need for gradient descent and backpropagation!*

Our model learns to reconstruct images using a low-dimensional vector (in the latent space)

We can generate a new image by feeding a randomly-generated low-dim vector to the “decoder”