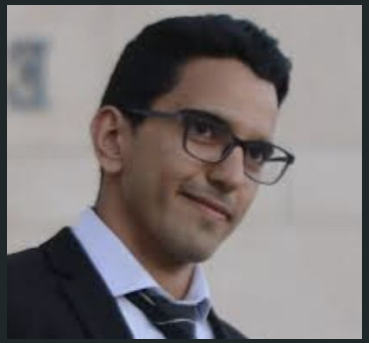
Multilayer perceptrons

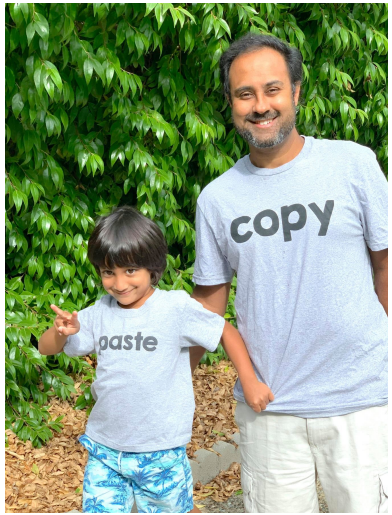
Surya Ganguli and Arash Ash



neuromatch academy

Who is Surya?

- Stanford Applied Physics & Facebook AI Research
- Interested in emergent computations in neuroscience, deep learning and physics
- College tennis player
- Bhangra dancer
- Father to multiple species
- Twitter: @SuryaGanguli



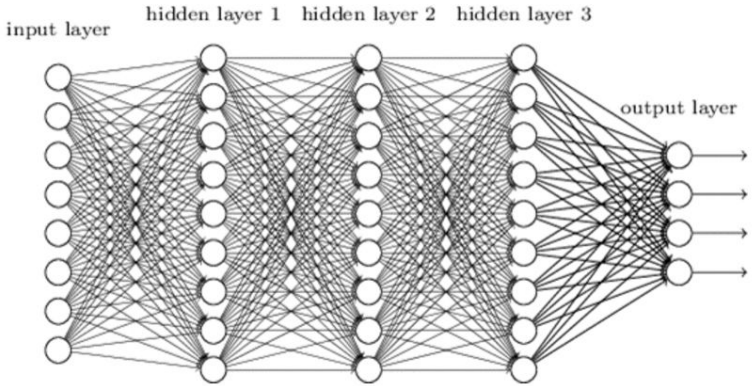
Who is Arash?

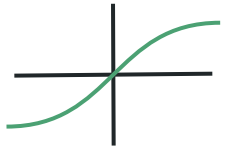
- Thank you for creating tutorials!

Major thanks to:

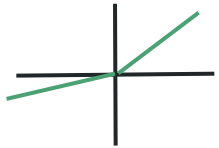
- Arash Ash for creating tutorial code
- Dylan Paiton for providing excellent references
- Konrad Koerding for a starting point for slides
- The volunteer army of Neuromatch!!

Our first computationally powerful deep net: a multilayer perceptron (MLP)





Sigmoid



Leaky ReLU

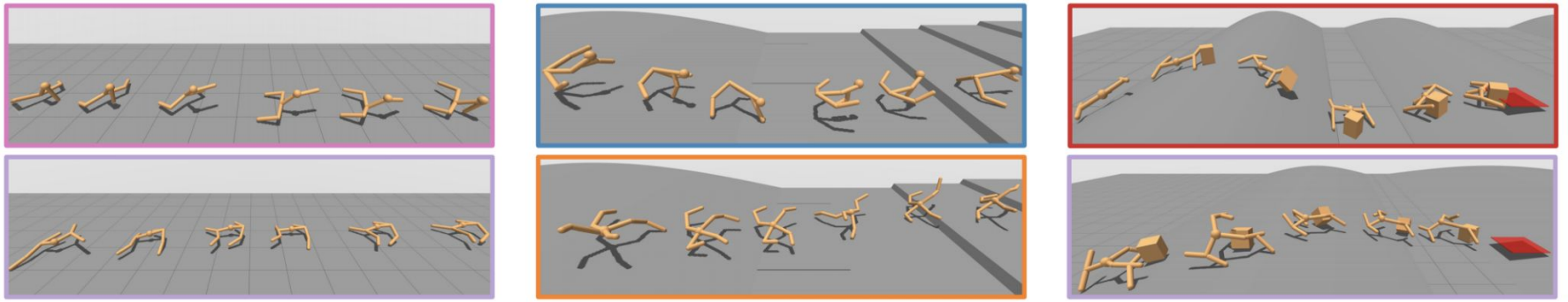
Motivation for MLPs

Embodied Intelligence via Learning and Evolution

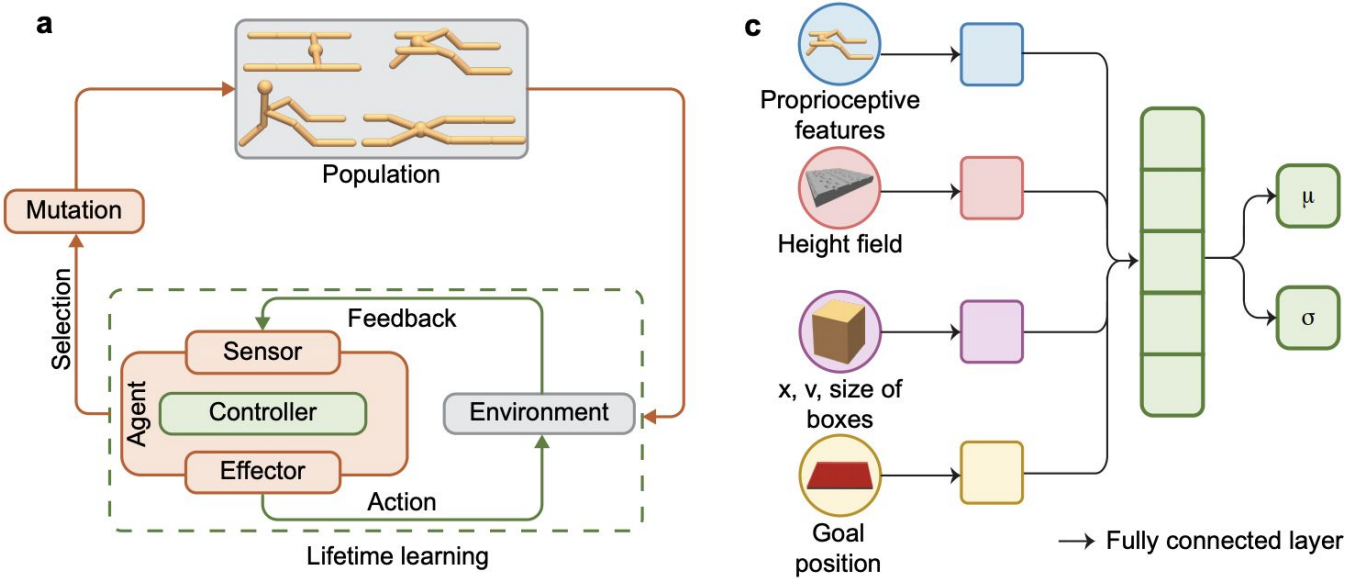
Agrim Gupta[1] **Silvio Savarese**[1] **Surya Ganguli**[2,3,4] **Li Fei-Fei**[1,4]

[1]Department of Computer Science, Stanford University
[2]Department of Applied Physics, Stanford University
[3]Wu-Tsai Neurosciences Institute, Stanford University, Stanford, CA, USA
[4]Stanford Institute for Human-Centered Artificial Intelligence, Stanford University, Stanford, CA, USA

Video



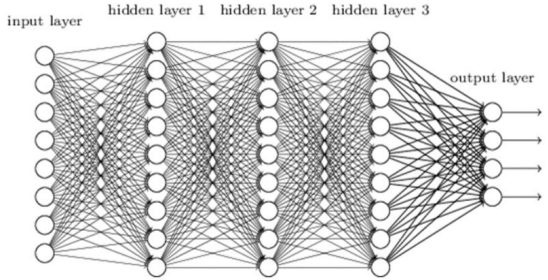
Motivation for MLPs



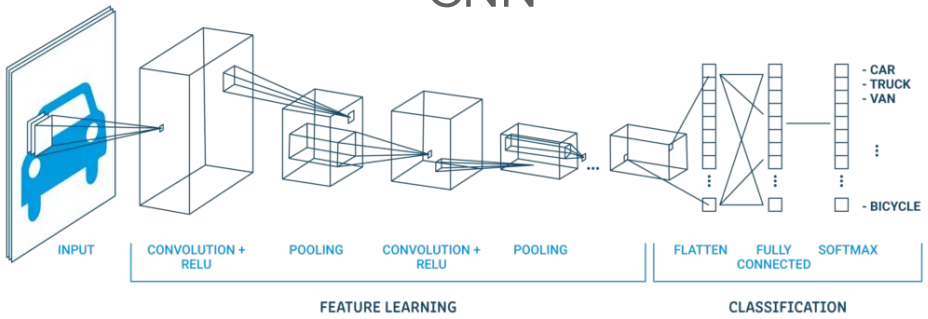
Video

MLPs are a basis for CNNs and RNNs

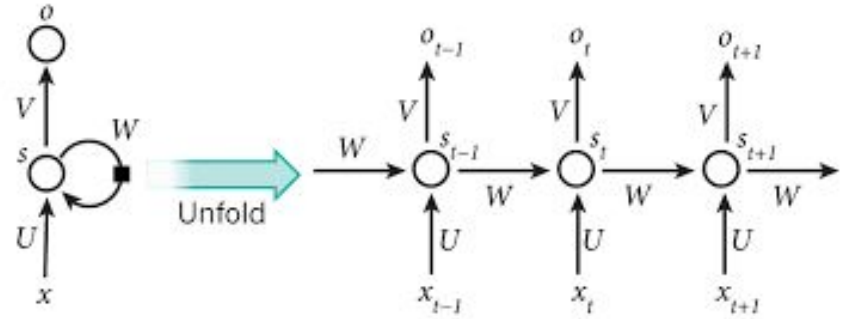
MLP



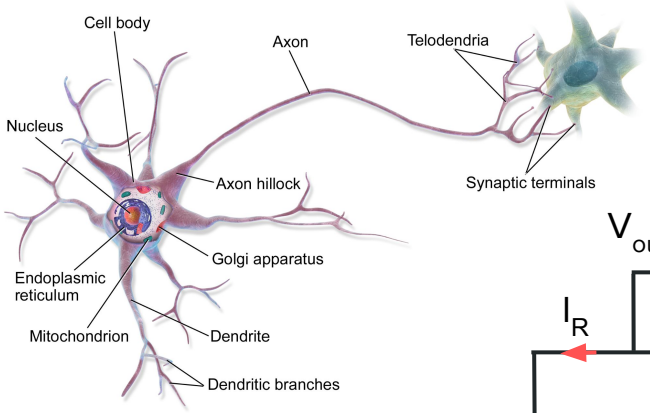
CNN

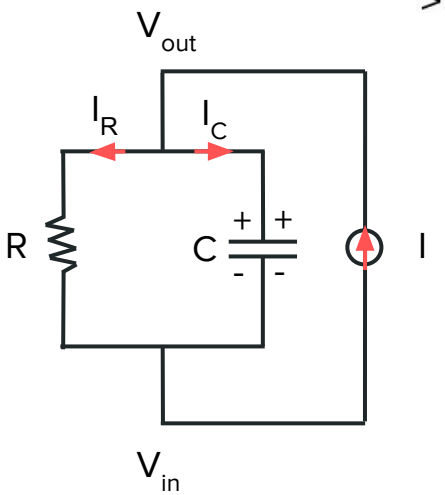


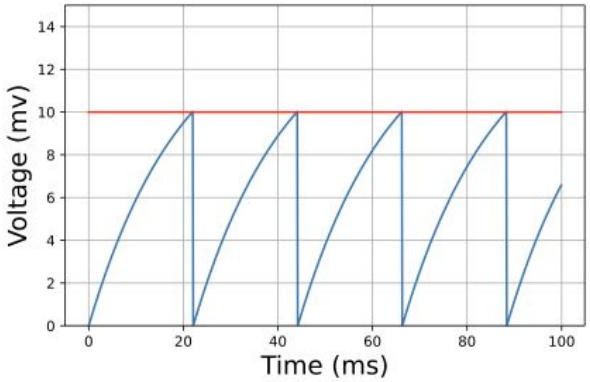
RNN

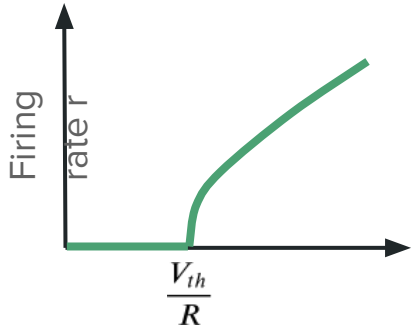


Tutorial 1: From biological to artificial neurons



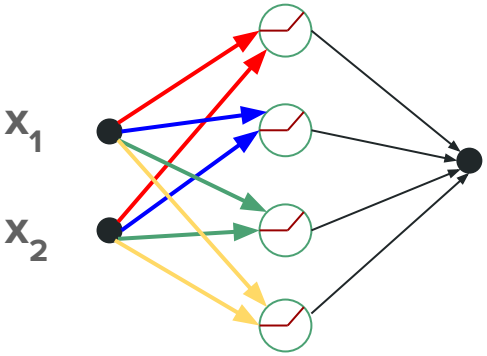






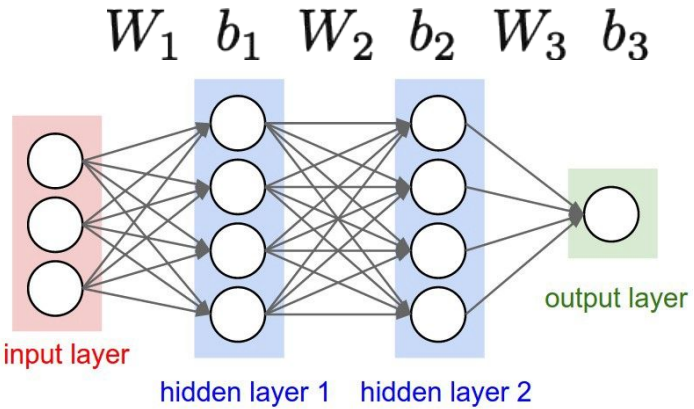
Tutorial 2: Computational benefits of nonlinearity

The universal approximation theorem: a 1 hidden layer neural network can compute any continuous function from input to output (though it may need lots of hidden neurons).



Tutorial 3: Build MLPs in PyTorch

$$y(x) \approx N(x) = W_k\sigma(\cdots\sigma(W_2\sigma(W_1x + b_1) + b_2)\cdots + b_{k-1}) + b_k$$



Turn this into Python code!

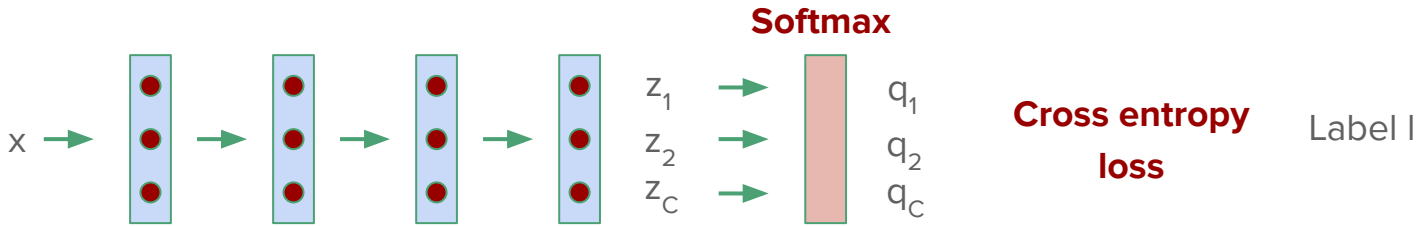
Tutorial 4: MLPs for classification: softmax and cross-entropy

Training data: $\{x_i\}$ A set of inputs (images, medical data, etc...)

$\{l_i\}$ A set of true labels (class, disease state, etc...)

$l_i \in \{1, 2, 3, \ldots, C\}$ C is the total number of classes

MLP:



Tutorial 4: Train and evaluate an MLP in PyTorch

Best practices for training and evaluating, as well as pitfalls:

What is the training set?
Does the split between train and test match your use case?
Is your metric for evaluation reasonable for real world deployment?
Will future validation data drift from train and test settings?
Are there biases due to problem selection, training data, algorithm design, evaluation metrics, or anywhere in ML pipeline?

Tutorial 5: Train and evaluate an MLP in PyTorch

Best practices for training and evaluating, as well as pitfalls:

What is the training set?
Does the split between train and test match your use case?
Is your metric for evaluation reasonable for real world deployment?
Will future validation data drift from train and test settings?
Are there biases due to problem selection, training data, algorithm design, evaluation metrics, or anywhere in ML pipeline?

Tutorial 6: Deep expressivity: if one hidden Layer can do anything, why do we need deeper nets?

There exist functions that can be computed using a small number of neurons using a deep net but require a large number of functions using a shallow net.

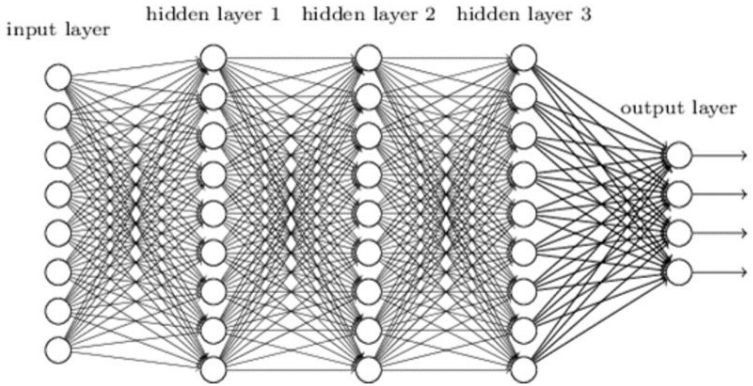
Even random deep nets can compute highly complex functions due to chaos that cannot be mimicked by shallow nets unless they have exponentially many more neurons.

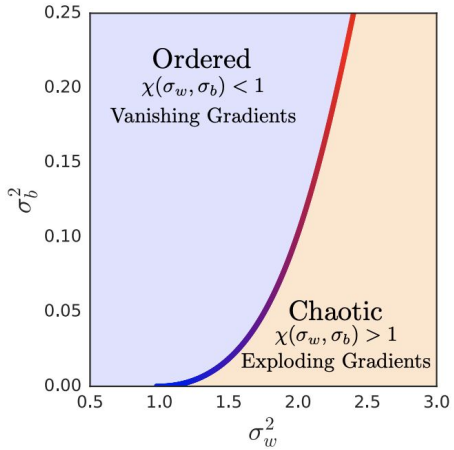
Tutorial 7: A case study on real data: classifying animal faces

Data augmentation

Look under the hood to see what the network learns.

Tutorial 8: The need for a good initialization: stick to the edge of chaos!





Forward propagation of activity / back propagation of errors

Weights too large: Could explode

Weights too small: Could vanish

Tutorial 9: Ethics: beware of hype in AI

Melanie Mitchell:Why AI is harder than we think

Michael Jordan: Artificial Intelligence— The Revolution Hasn't Happened Yet

Amanda Gefter: On Walter Pitts, The Man Who Tried to Redeem the World with Logic

Jeremy Bernstein: Marvin Minsky's vision of the future, New Yorker

Surya Ganguli: Intertwined quest for understanding biological intelligence and creating artificial intelligence

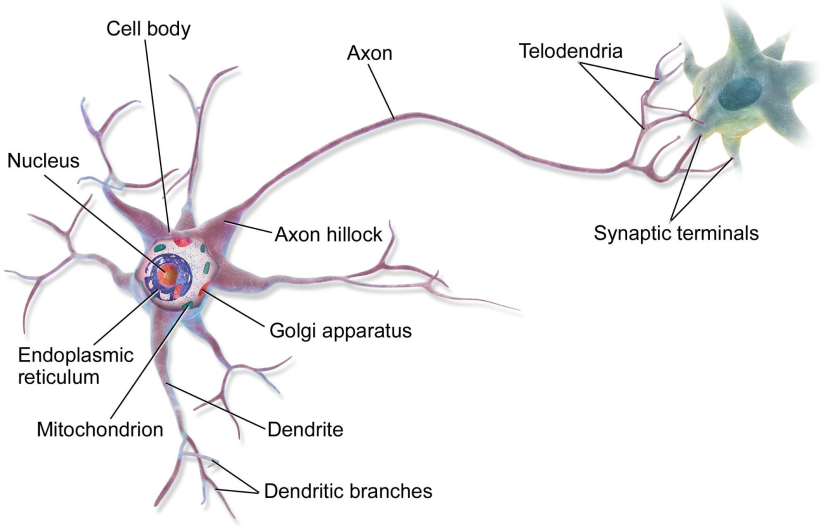
From biological to artificial neurons

Lecture 1

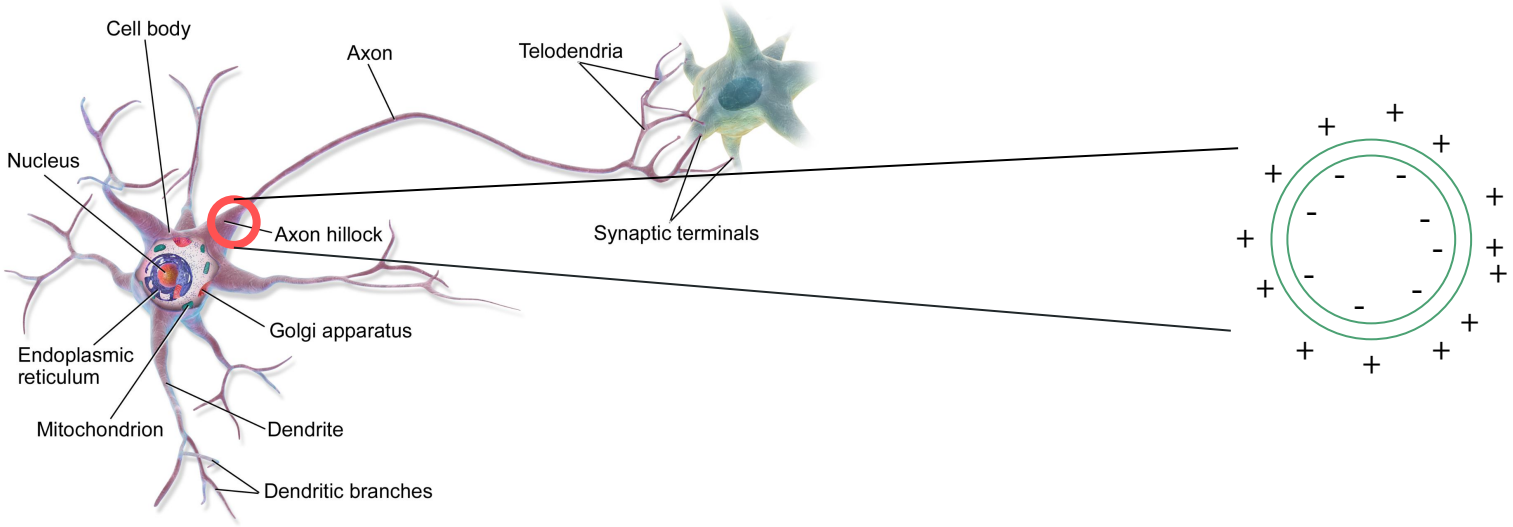
Understanding the biological origins of single neuron nonlinearities

From neurobiology to ReLUs

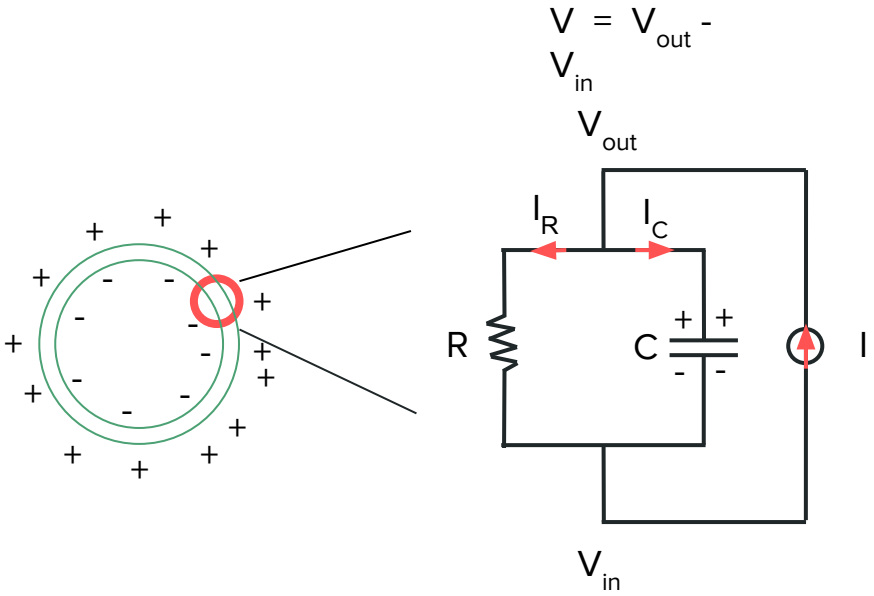
A biological neuron



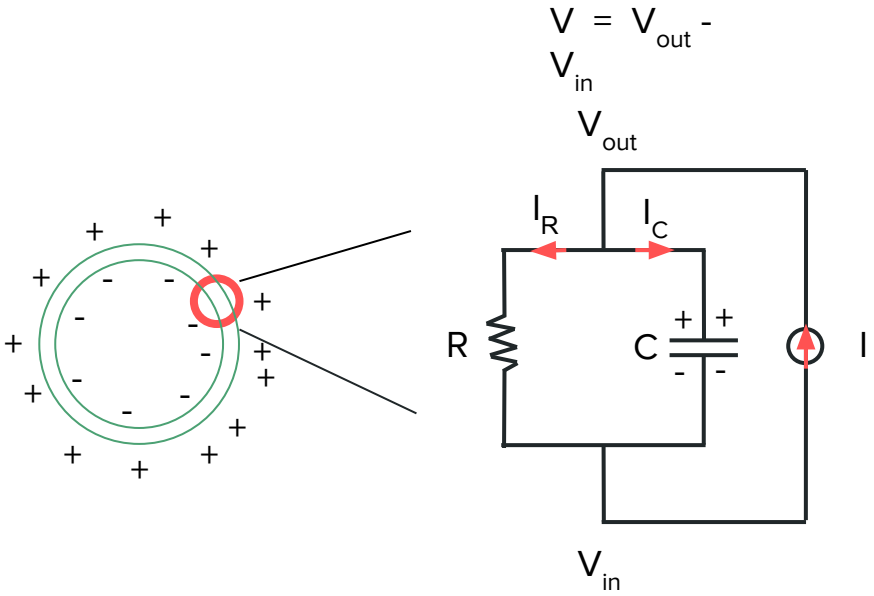
The membrane potential is the key variable



The passive membrane as an RC circuit

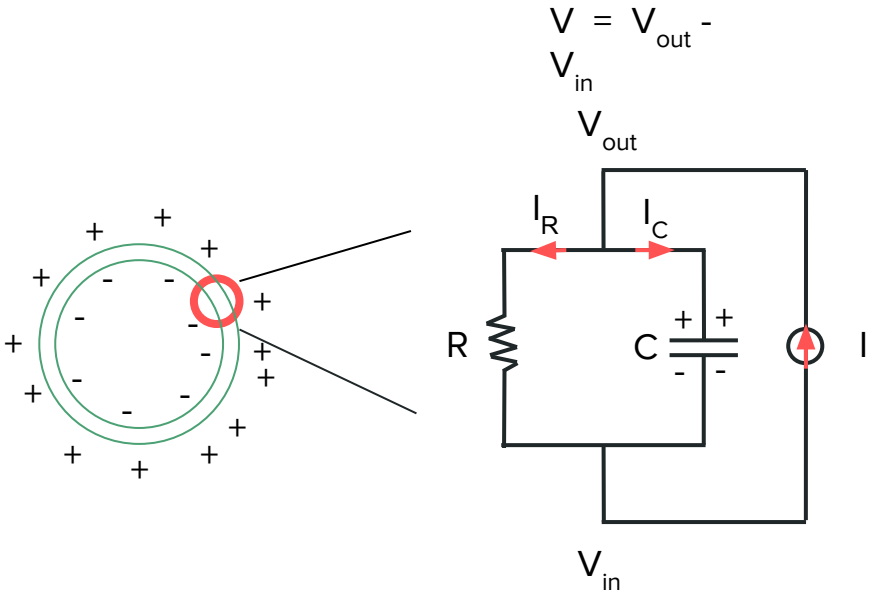


The passive membrane as an RC circuit



$$I_R + I_C = I$$

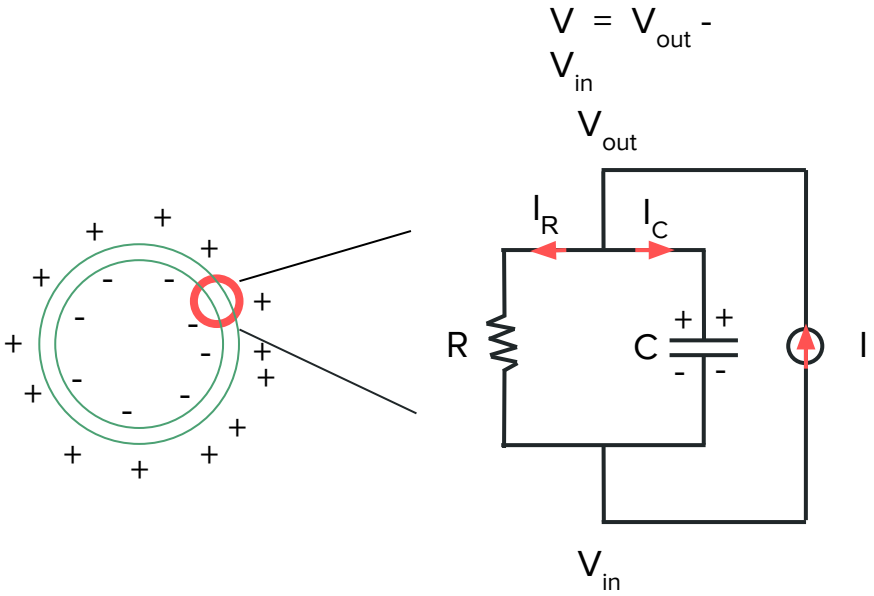
The passive membrane as an RC circuit



$$I_R + I_C = I$$

$$I_R = \frac{V}{R} \qquad I_C = C\frac{dV}{dt}$$

The passive membrane as an RC circuit

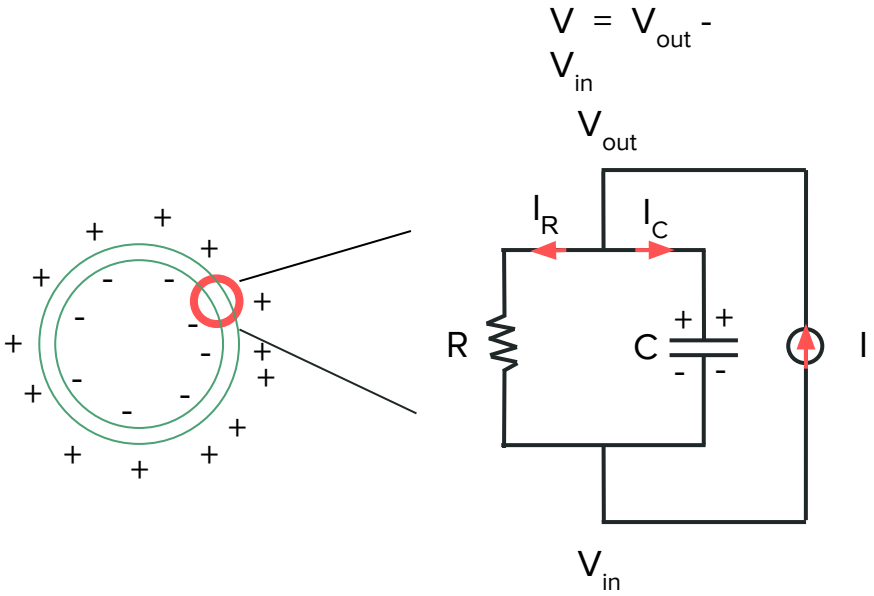


$$I_R + I_C = I$$

$$I_R = \frac{V}{R} \qquad I_C = C\frac{dV}{dt}$$

$$\frac{V}{R} + C\frac{dV}{dt} = I$$

The passive membrane as an RC circuit



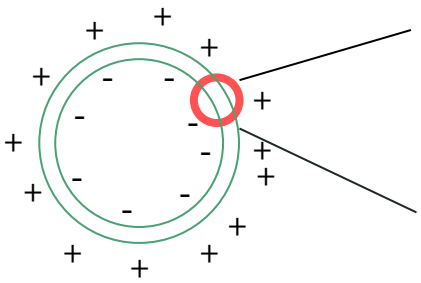
$$I_R + I_C = I$$

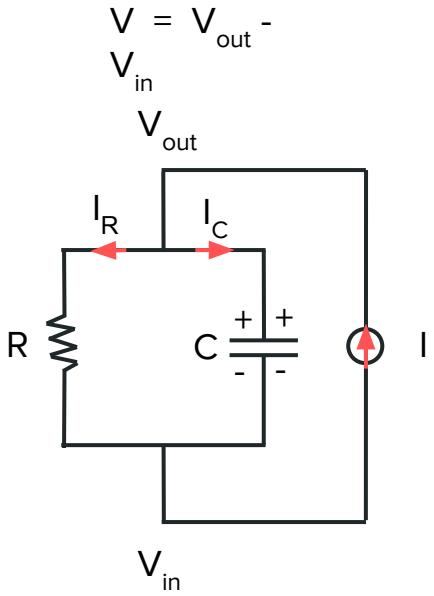
$$I_R = \frac{V}{R} \qquad I_C = C\frac{dV}{dt}$$

$$\frac{V}{R} + C\frac{dV}{dt} = I$$

$$\frac{dV}{dt} = -\frac{V}{RC} + \frac{I}{C}$$

The passive membrane as an RC circuit





$$I_R + I_C = I$$

$$I_R = \frac{V}{R} \qquad I_C = C\frac{dV}{dt}$$

$$\frac{V}{R} + C\frac{dV}{dt} = I$$

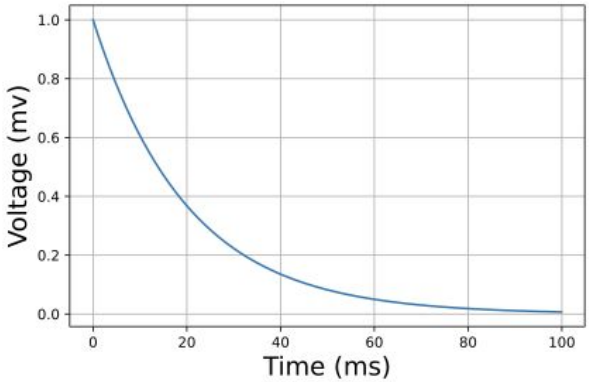
$$\frac{dV}{dt} = -\frac{V}{RC} + \frac{I}{C}$$

$$\tau = RC \approx 20ms$$

Passive membrane voltage dynamics

$$\tau = RC \approx 20ms$$

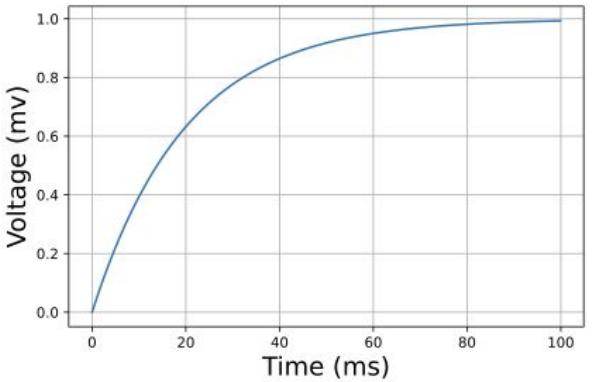
$$I = 0$$



$$V(t) = V(0)\, e^{-t/\tau}$$

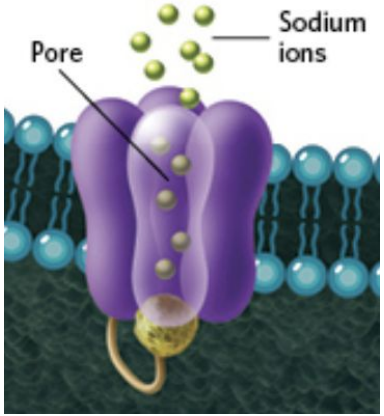
$$\frac{dV}{dt} = -\frac{V}{RC} + \frac{I}{C}$$

$$I > 0$$



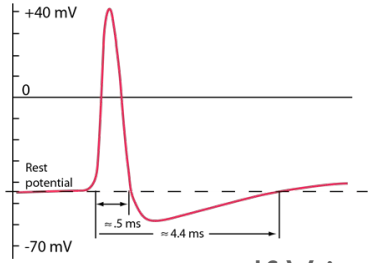
$$V(t) \to V(\infty) = IR$$

Thats not all: the membrane is active!



Sodium rushes into the cell

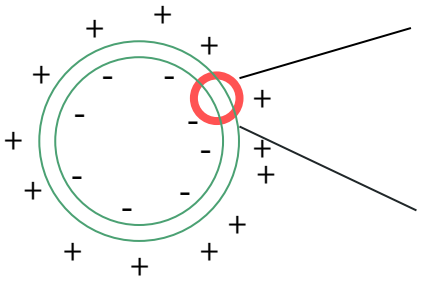
The sodium channel opens more

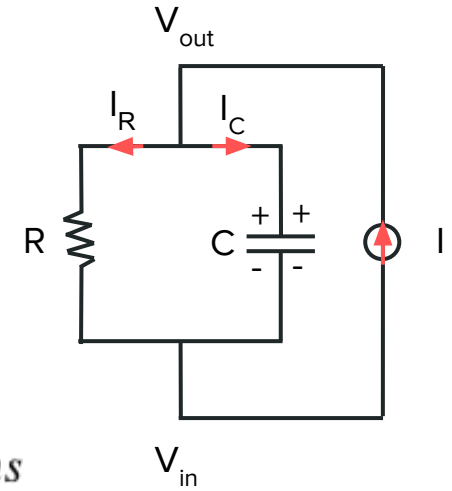


V increases more

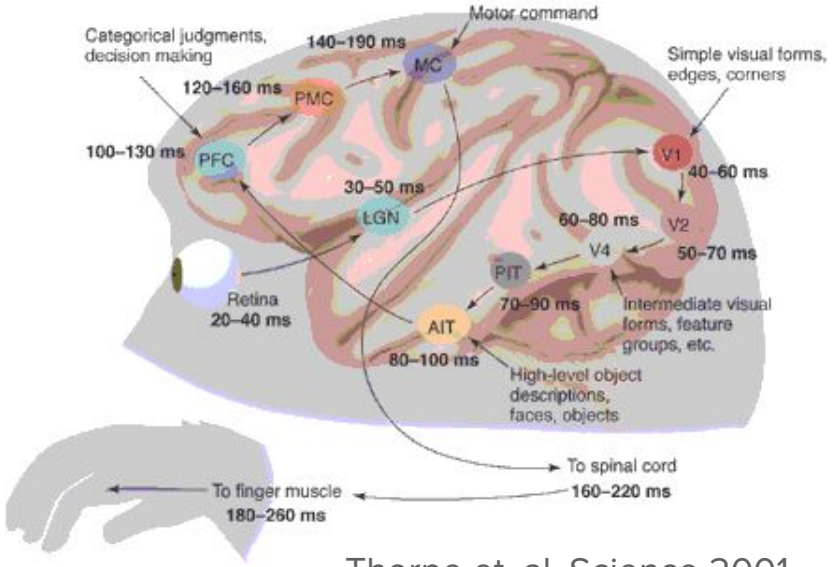
If V inc more above a threshold

From biophysics to psychology: the “clock” speed of the brain and mind





$$\tau = RC \approx 20ms$$



Thorpe et. al. Science 2001

A simple spiking neuron: integrate and fire

Dynamical rule:

If $V(t) < V_{th}$ then $\frac{dV}{dt} = -\frac{V}{RC} + \frac{I}{C}$

If $V(t) = V_{th}$ then emit a spike and set $V(t) = V_{reset}$

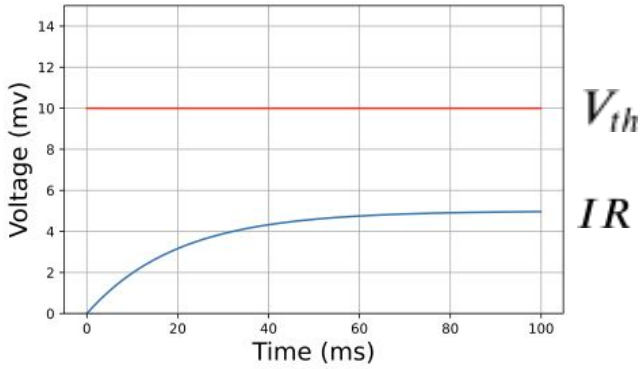
A simple spiking neuron: integrate and fire

Dynamical rule:

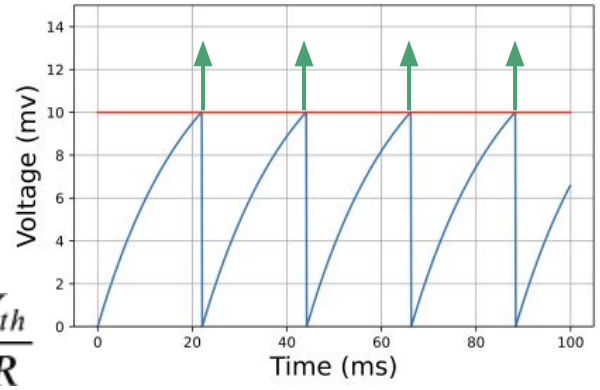
If $V(t) < V_{th}$ then $\frac{dV}{dt} = -\frac{V}{RC} + \frac{I}{C}$

If $V(t) = V_{th}$ then emit a spike and set $V(t) = V_{reset}$

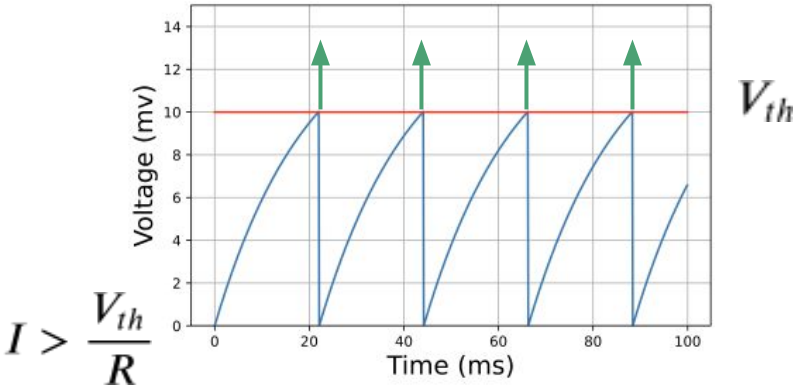
$I < \frac{V_{th}}{R}$

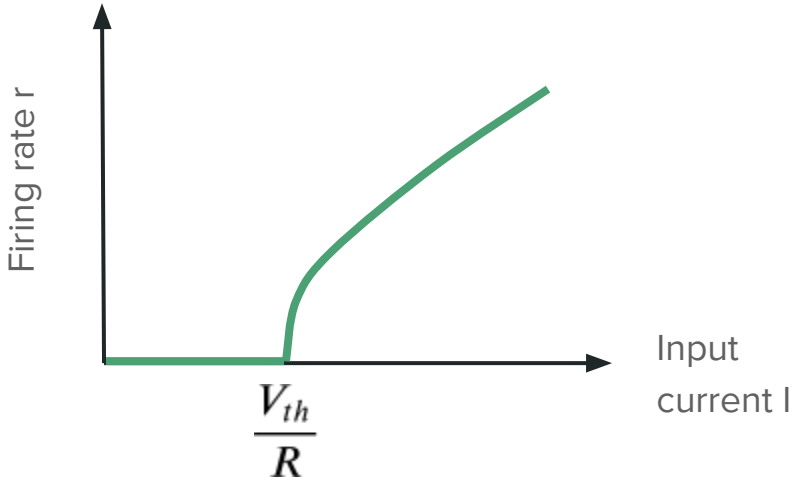


$I > \frac{V_{th}}{R}$

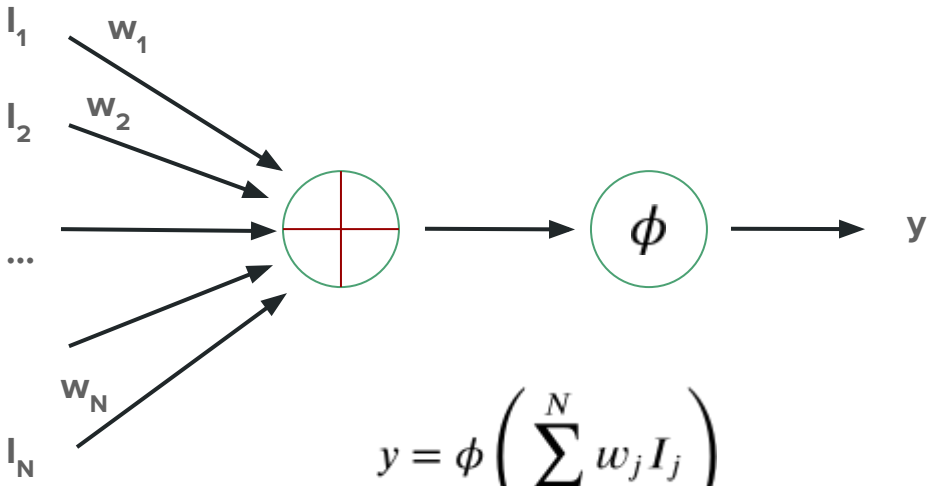


From integrate and fire spiking neuron to a rectified linear rate neuron

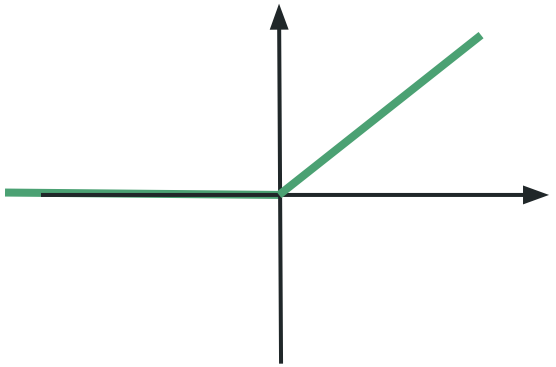




The rectified linear unit (ReLU) in AI



$$y = \phi\left(\sum_{j=1}^{N} w_j I_j\right)$$

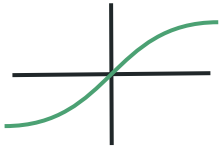


$$\phi(x) = \max(0, x)$$

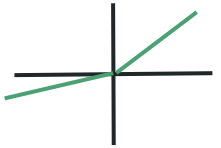
A history of activation functions

Table 3: Non-linearities tested.

Name	Formula	Year
none	$y = x$	-
sigmoid	$y = \frac{1}{1+e^{-x}}$	1986
tanh	$y = \frac{e^{2x}-1}{e^{2x}+1}$	1986
ReLU	$y = \max(x, 0)$	2010
(centered) SoftPlus	$y = \ln(e^x + 1) - \ln 2$	2011
LReLU	$y = \max(x, \alpha x), \alpha \approx 0.01$	2011
maxout	$y = \max(W_1 x + b_1, W_2 x + b_2)$	2013
APL	$y = \max(x,0) + \sum_{s=1}^{S} a_i^s \max(0, -x + b_i^s)$	2014
VLReLU	$y = \max(x, \alpha x), \alpha \in 0.1, 0.5$	2014
RReLU	$y = \max(x, \alpha x), \alpha = \text{random}(0.1, 0.5)$	2015
PReLU	$y = \max(x, \alpha x), \alpha$ is learnable	2015
ELU	$y = x$, if $x \geq 0$, else $\alpha(e^x - 1)$	2015



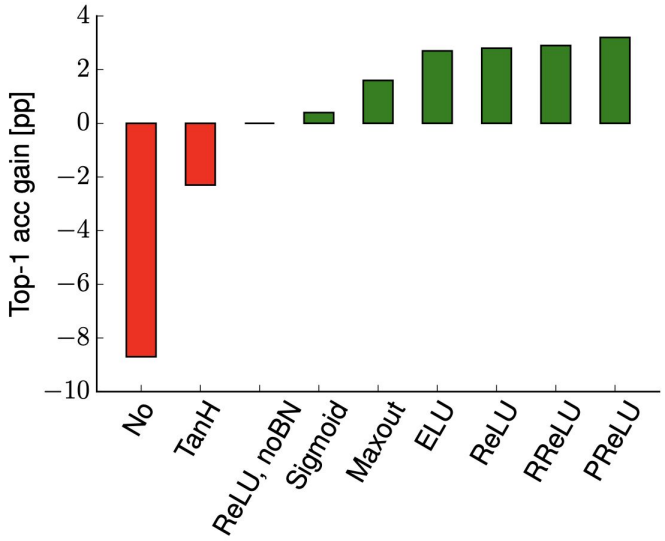
Sigmoid



Leaky ReLU

Systematic evaluation of CNN advances on the ImageNet, Mishkin et al

Performance for different nonlinearities



Systematic evaluation of CNN advances on the ImageNet, Mishkin et al

How to simulate an integrate and fire neuron

$$\frac{dV}{dt} = \frac{V(t+dt) - V(t)}{dt} = -\frac{V}{RC} + \frac{I}{C}$$

$$V(t+dt) = V(t) + dt\left(-\frac{V}{RC} + \frac{I}{C}\right) \qquad \text{works well when: } \frac{dt}{\tau} \ll 1$$

Now your turn! You get to play with code to simulate the integrate and fire neuron and compute firing rate curves.

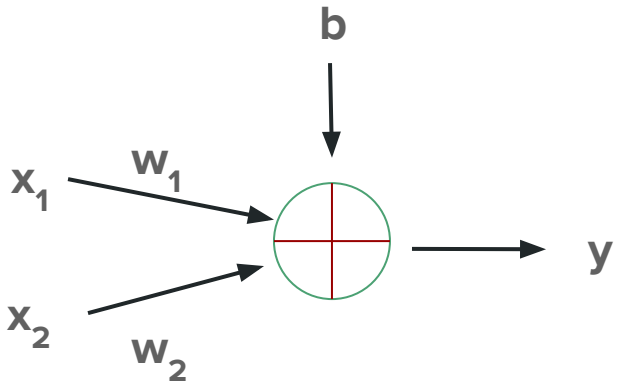
The computational benefits of nonlinearity

Lecture 2

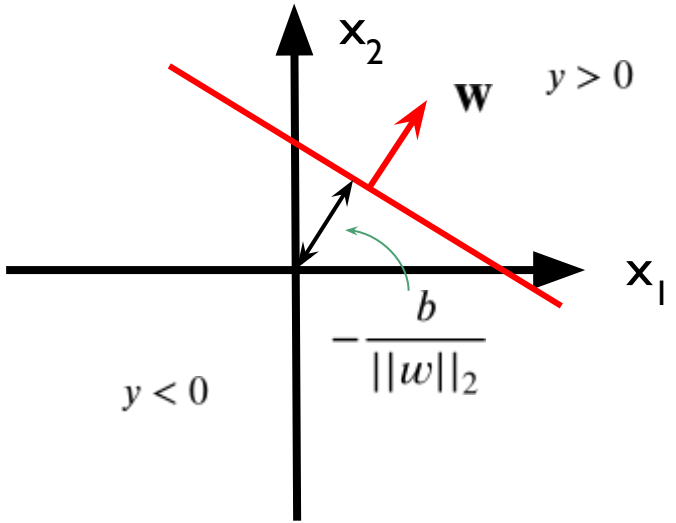
What can even a shallow nonlinear network with 1 hidden layer do that a linear network cannot?

(In Tutorial 6 we will cover what a deep network can do that a shallow network cannot)

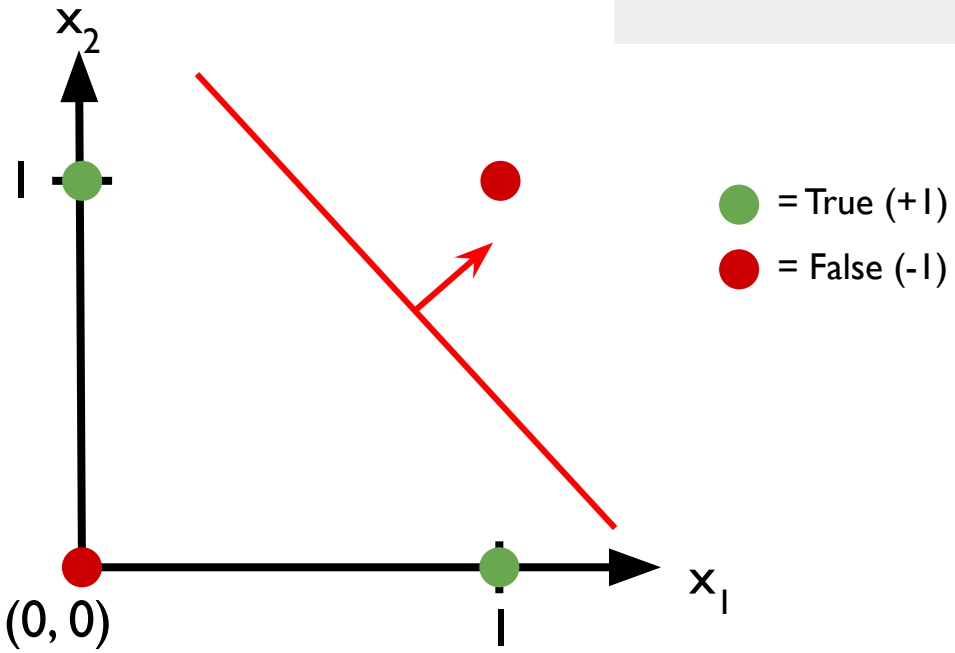
A single linear neuron can only construct linear functions and decision boundaries



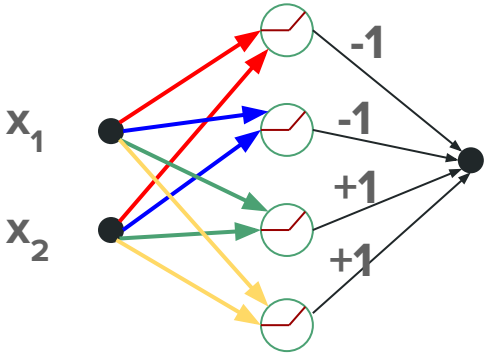
$$y = \mathbf{w} \cdot \mathbf{x} + b$$

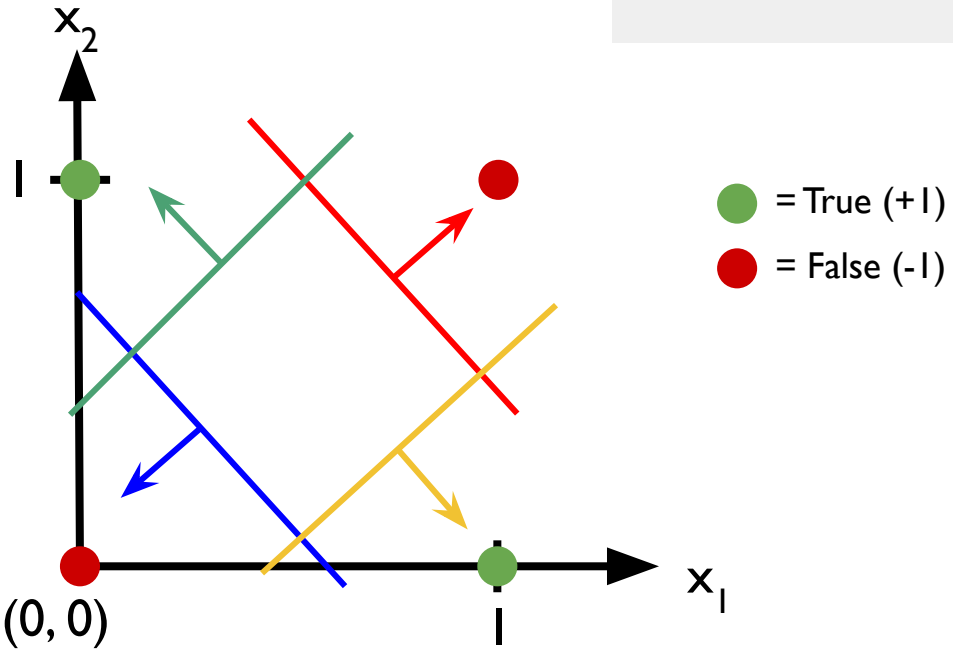


A single linear neuron can’ t solve XOR

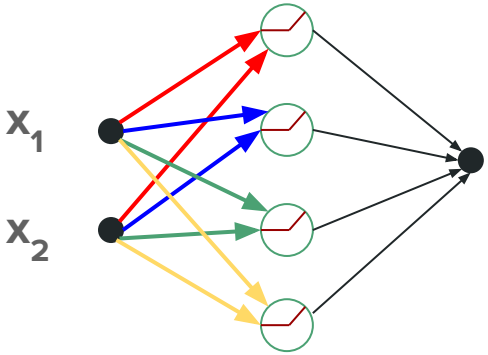


But a 1 hidden layer MLP can!





Ok... so what else can a 1 hidden layer MLP solve?



Answer: almost anything!*

(* If you give it enough hidden neurons)

Universal function approximation theorem

Let $\phi : \mathbb{R} \to \mathbb{R}$ be a continuous activation function (that is not a simple polynomial)

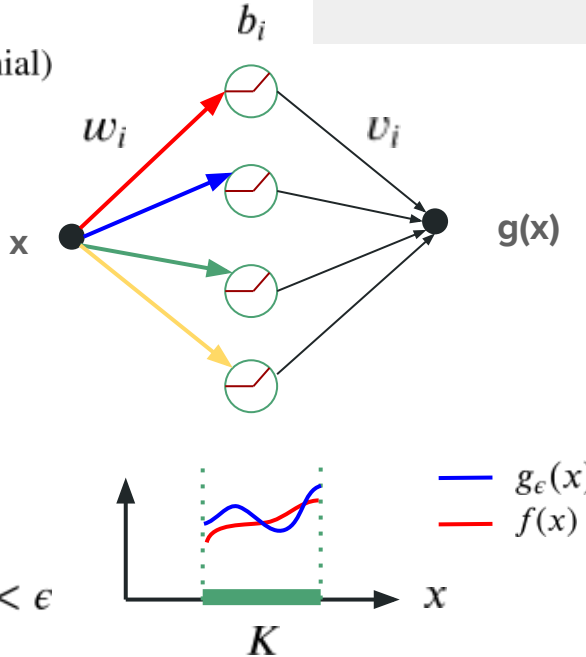
Let $f : \mathbb{R} \to \mathbb{R}$ be a continuous target function.

Let $g(x) = \sum_{i=1}^{N} v_i \phi(w_i x + b_i)$ be a family of neural network functions.

For every compact subset $K \in \mathbb{R}$ and for every error tolerance level ϵ

there exists an integer N and a set of parameters $\{v_i, w_i, b_i\}_{i=1}^{N}$

such that the corresponding function $g_\epsilon(x)$ obeys $\sup_{x \in K} |f(x) - g_\epsilon(x)| < \epsilon$

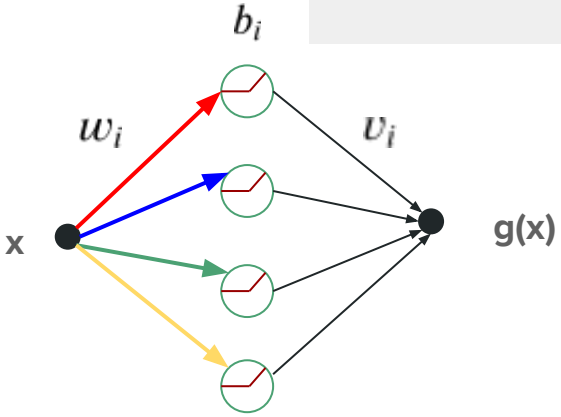


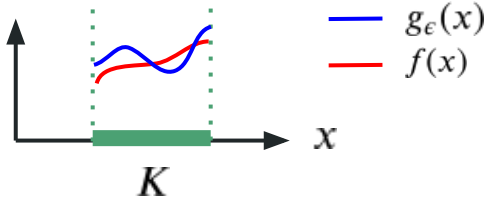
Okay... so why do we need deep nets with more than one hidden layer?

While the universal approximation theorem says we can approximate a function to some accuracy with a one hidden layer neural network,

It does not tell us how many hidden neurons we will need:

there exists an integer N and a set of parameters $\{v_i, w_i, b_i\}_{i=1}^{N}$



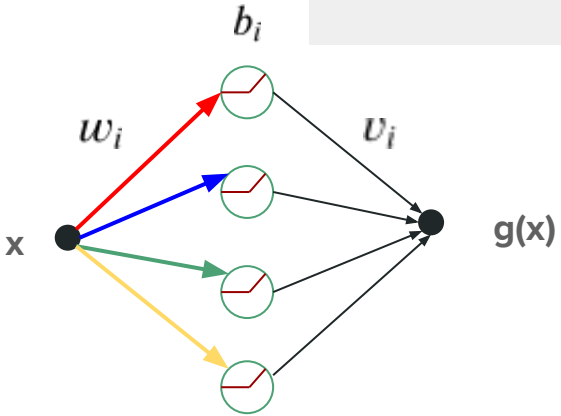


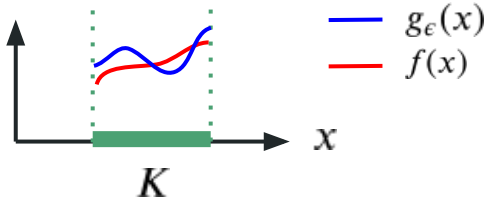
The phenomenon of deep expressivity

For example, there exist some functions which can be efficiently approximated by a deep network.

But to approximate these same functions with a 1 hidden layer network would require exponentially many more neurons.

We will explore this in a later tutorial.



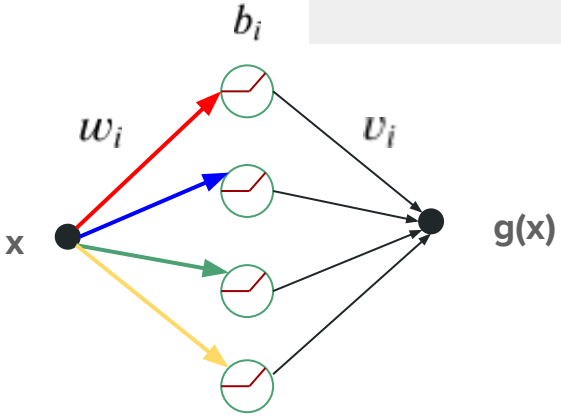


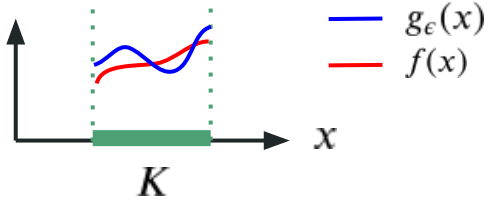
What else the universal approximation theorem not tell us: how to learn.

While there may exist a network that approximates our function,

There is no guarantee we can find this function given a finite set of example input output pairs.

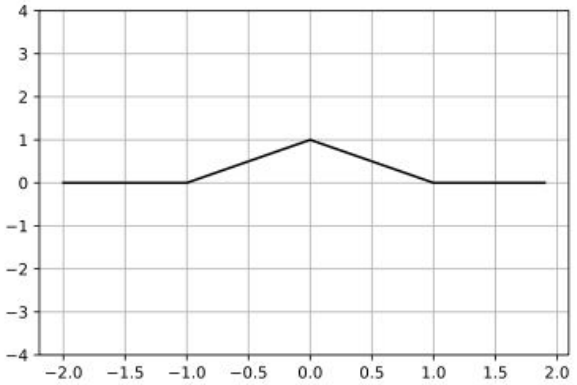
there exists an integer N and a set of parameters $\{v_i, w_i, b_i\}_{i=1}^{N}$





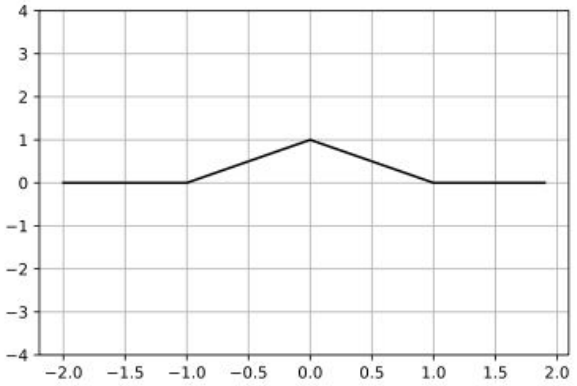
Intuition behind universal approximation

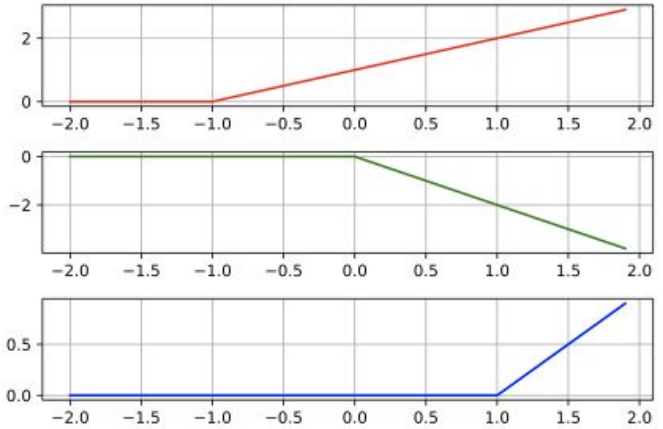
Example: construct a local bump function using ReLUs:



Intuition behind universal approximation

Example: construct a local bump function using ReLUs:





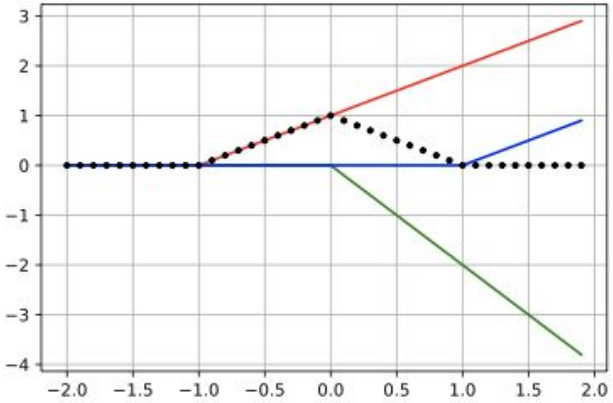
ReLU(x+1)

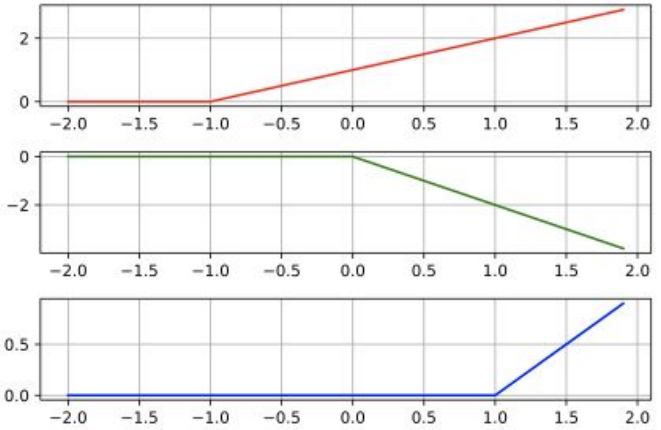
-2 * ReLU(x)

ReLU(x-1)

Intuition behind universal approximation

Example: construct a local bump function using ReLUs:



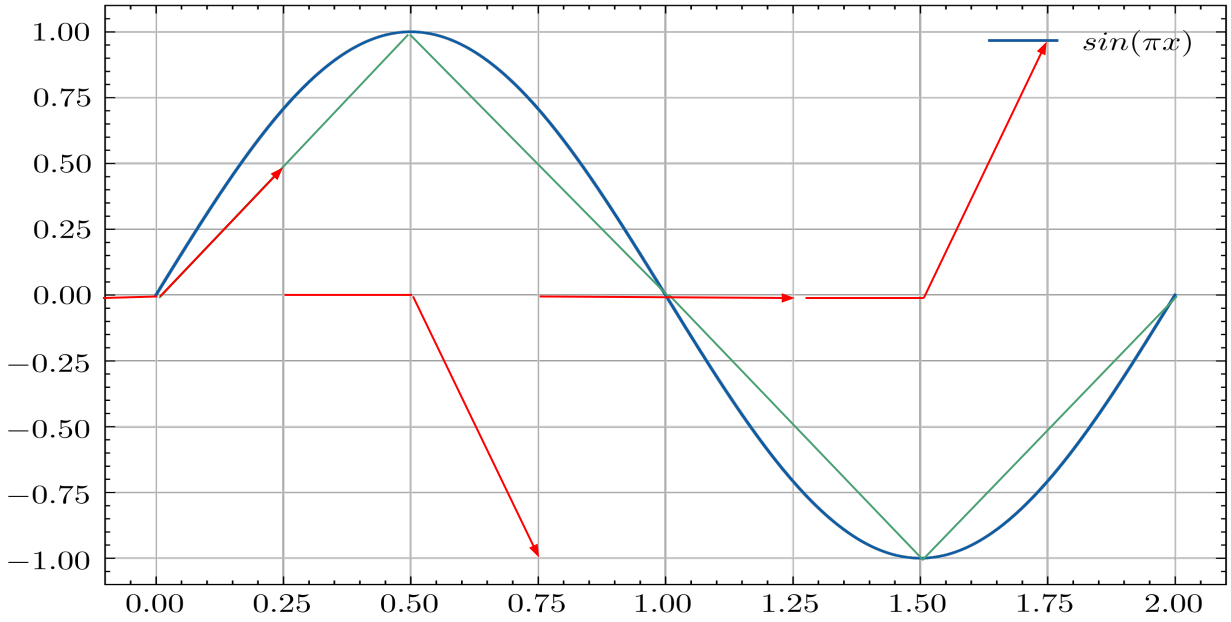


ReLU(x+1)

-2 * ReLU(x)

ReLU(x-1)

Another example: approximating sin function



Now your turn: play around with approximating the sin function using a 1 hidden layer ReLU MLP

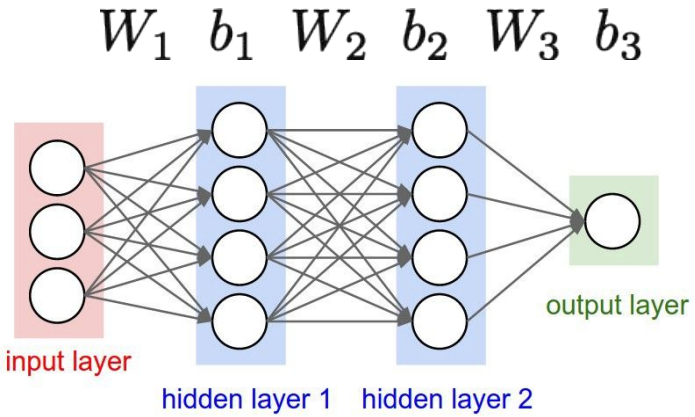
Building Multi Layer Perceptrons in PyTorch

Tutorial 3

Multi-layer perceptrons (MLPs)

$$y(x) \approx N(x) = W_k\sigma(\cdots\sigma(W_2\sigma(W_1x + b_1) + b_2)\cdots + b_{k-1}) + b_k$$

- ***Layer:*** one of the intermediate vectors
- ***Neuron:*** one of the entries of a layer vector
- ***Depth:*** the number of layers
- ***Width:*** the layer's dimension
- ***Weights:*** coefficients of the matrice W_k
- ***Biases*** coefficients of the vector b_k
- The ***activation function*** or ***non-linearity*** is the function σ.



So let us create a general MLP

Input/Output behaviour:

- We tell it sizes of **input**, each **hidden**, and **output** layers
- And which activation function to use for hidden layers
- Then it will construct an MLP with no output activation since it's general!

PyTorch Design

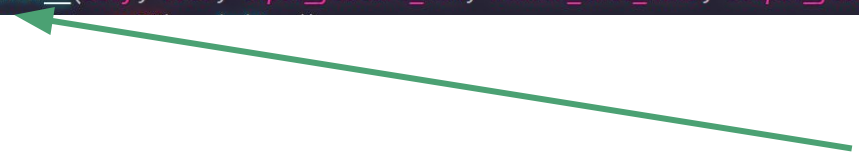
```
class Net(nn.Module):
```

Create a model class called “Net” which subclasses nn.Module, the base class for all neural network modules.

nn.Module takes care of backprop for you so you don’t need to define a backward() function!

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
```



Define the initialization inputs of the model class

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
```

nn.ELU, nn.Hardshrink, nn.Hardsigmoid,

nn.Hardtanh, nn.Hardswish, nn.LeakyReLU,

nn.LogSigmoid, nn.MultiheadAttention,

nn.PReLU, nn.ReLU, nn.ReLU6, nn.RReLU,

nn.SELU, nn.CELU, nn.GELU, nn.Sigmoid,

nn.SiLU, nn.Mish, nn.Softplus, nn.Softshrink,

nn.Softsign, nn.Tanh, nn.Tanhshrink,
nn.Threshold

"actv" is the string of the activation function with arguments which exists in torch.nn, e.g., "LeakyReLU(0.1)"

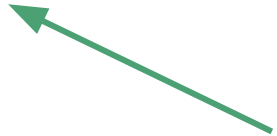
PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
```

Input layer size,
E.g., for an RGB image of
32x32 it is 32x32x3=3072

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
```



List of hidden layer sizes,
E.g., for 3 hidden layers,
[256, 128, 64]

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
```

Output layer size,
E.g., for a 3 way classification
it would be 3

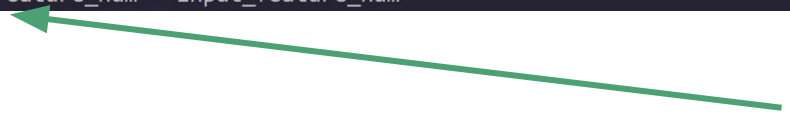
PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
```

Calls the init function of its base class (nn.Module)

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
```



Save the input size for later use in forward(), since we will reshape inputs using this

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()
```

Initialize another subclass of nn.Module with the functionality to run the given modules in sequence (we'll give the modules next)

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()

        in_num = input_feature_num
        for i in range(len(hidden_unit_nums)):
```

Initialize the variable that will determine the **input size** of each layer (nn.Linear module)

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()

        in_num = input_feature_num
        for i in range(len(hidden_unit_nums)):

          out_num = hidden_unit_nums[i]
          layer = ...
          in_num = out_num
          self.mlp.add_module('Linear_%d'%i, layer)
```

Initialize the variable that will determine the **output size** of each layer which is same as the hidden units in that layer

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()

        in_num = input_feature_num
        for i in range(len(hidden_unit_nums)):

          out_num = hidden_unit_nums[i]
          layer = ...
          in_num = out_num
          self.mlp.add_module('Linear_%d'%i, layer)
```

Determine the input size of next layer which is the output size of current layer

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()

        in_num = input_feature_num
        for i in range(len(hidden_unit_nums)):

          out_num = hidden_unit_nums[i]
          layer = ...
          in_num = out_num
          self.mlp.add_module('Linear_%d'%i, layer)
```

Now we can append the module you just constructed with a name to the Sequential module we initialized

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()

        in_num = input_feature_num
        for i in range(len(hidden_unit_nums)):

          out_num = hidden_unit_nums[i]
          layer = ...
          in_num = out_num
          self.mlp.add_module('Linear_%d'%i, layer)

          actv_layer = eval('nn.%s'%actv)
          self.mlp.add_module('Activation_%d'%i, actv_layer)
```

We initialize the activation module for that layer and append it similar to before

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()

        in_num = input_feature_num
        for i in range(len(hidden_unit_nums)):

          out_num = hidden_unit_nums[i]
          layer = ...
          in_num = out_num
          self.mlp.add_module('Linear_%d'%i, layer)

          actv_layer = eval('nn.%s'%actv)
          self.mlp.add_module('Activation_%d'%i, actv_layer)
```

Caution!
some activation modules have learnable parameters so it is important to initialize them separately for each layer

We initialize the activation module for that layer and append it similar to before

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()

        in_num = input_feature_num
        for i in range(len(hidden_unit_nums)):

          out_num = hidden_unit_nums[i]
          layer = ...
          in_num = out_num
          self.mlp.add_module('Linear_%d'%i, layer)

          actv_layer = eval('nn.%s'%actv)
          self.mlp.add_module('Activation_%d'%i, actv_layer)

        out_layer = nn.Linear(in_num, output_feature_num)
        self.mlp.add_module('Output_Linear', out_layer)
```

Finally, we define and append the output layer separately since it does not have an activation layer and its output size is not in the hidden unit list

PyTorch Design

```
class Net(nn.Module):
    def __init__(self, actv, input_feature_num, hidden_unit_nums, output_feature_num):
        super(Net, self).__init__()
        self.input_feature_num = input_feature_num
        self.mlp = nn.Sequential()

        in_num = input_feature_num
        for i in range(len(hidden_unit_nums)):

          out_num = hidden_unit_nums[i]
          layer = ...
          in_num = out_num
          self.mlp.add_module('Linear_%d'%i, layer)

          actv_layer = eval('nn.%s'%actv)
          self.mlp.add_module('Activation_%d'%i, actv_layer)

        out_layer = nn.Linear(in_num, output_feature_num)
        self.mlp.add_module('Output_Linear', out_layer)
```

Now only left is the forward() function which should be easy since we designed it right ;)

You complete it!

MLPs for classification: softmax and cross-entropy

Lecture 4

How to classify data by expressing and training probability distributions over a finite set of class labels.

Many DL problems involve classification

MNIST: Which one of 10 digits is an input image?

ImageNET: which one of 1000 classes is an input image?

Language models: which word out of a given vocabulary is the most likely next word?

Medical diagnosis: does certain patient medical data signify a diseased state or not?

Classification involves returning a probability distribution over possible label values

MNIST: Probability distribution over 10 digit labels given image.

ImageNET: Probability distribution over 1000 classes given image.

Language models: Probability distribution over next word given previous words.

Medical diagnosis: Probability of disease given medical data.

How to express and train a probability distribution?

Training data: $\{x_i\}$ A set of inputs (images, medical data, etc...)

$\{l_i\}$ A set of true labels (class, disease state, etc...)

$l_i \in \{1, 2, 3, \ldots, C\}$ C is the total number of classes

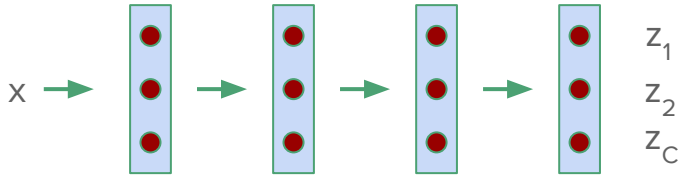
How to express and train a probability distribution?

Training data: $\{x_i\}$ A set of inputs (images, medical data, etc...)

$\{l_i\}$ A set of true labels (class, disease state, etc...)

$l_i \in \{1, 2, 3, \ldots, C\}$ C is the total number of classes

MLP:



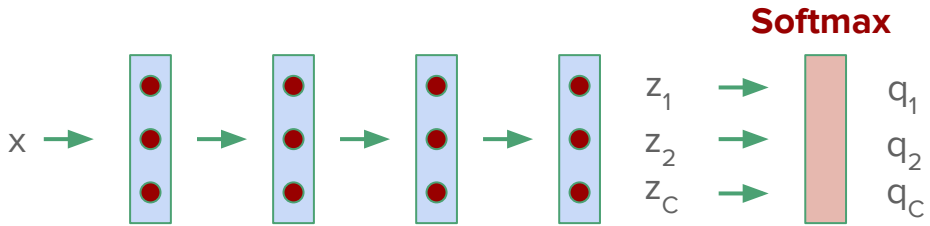
How to express and train a probability distribution?

Training data: $\{x_i\}$ A set of inputs (images, medical data, etc...)

$\{l_i\}$ A set of true labels (class, disease state, etc...)

$l_i \in \{1, 2, 3, \ldots, C\}$ C is the total number of classes

MLP:



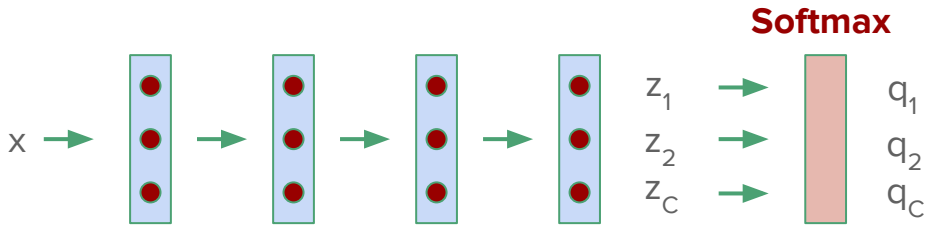
How to express and train a probability distribution?

Training data: $\{x_i\}$ A set of inputs (images, medical data, etc...)

$\{l_i\}$ A set of true labels (class, disease state, etc...)

$l_i \in \{1, 2, 3, \ldots, C\}$ C is the total number of classes

MLP:



Label l

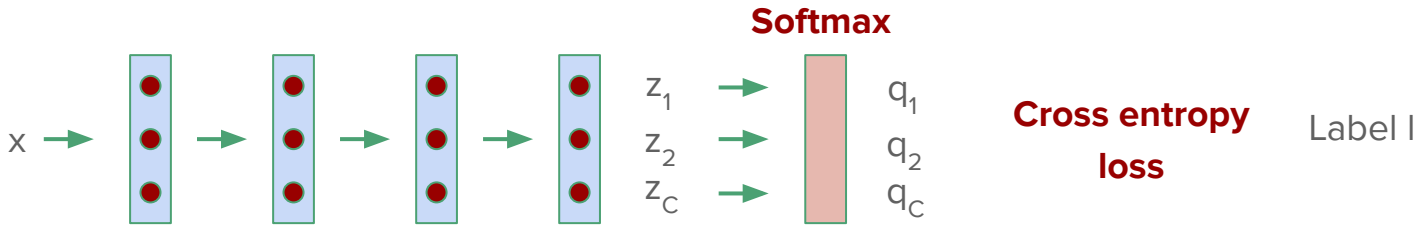
How to express and train a probability distribution?

Training data: $\{x_i\}$ A set of inputs (images, medical data, etc...)

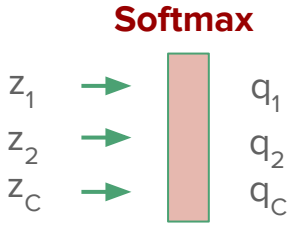
$\{l_i\}$ A set of true labels (class, disease state, etc...)

$l_i \in \{1, 2, 3, \ldots, C\}$ C is the total number of classes

MLP:

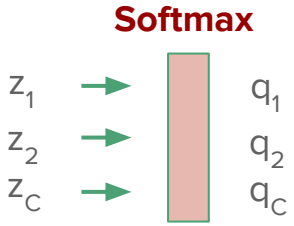


Converting logits (z) to probabilities (q)



$$q_i > 0 \qquad \sum_{i=1}^{C} q_i = 1$$

Converting logits (z) to probabilities (q)



$$q_i > 0 \qquad \sum_{i=1}^{C} q_i = 1$$

The softmax function solves these constraints:

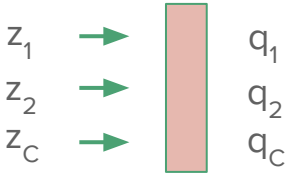
$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Training: inc probability of correct class
dec probability of incorrect classes

Goal: increase q_j if and only if $j = l_i$

Training: inc probability of correct class
dec probability of incorrect classes

Softmax



Cross entropy loss

Label l_i

Goal: increase q_j if and only if $j = l_i$

One hot encoding of labels

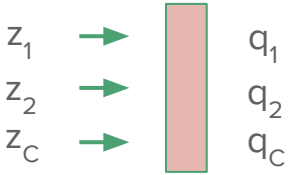
$$l_i \rightarrow \begin{array}{ll} y_{ij} = 1 & \text{if } j = l_i \\ y_{ij} = 0 & \text{if } j \neq l_i \end{array}$$

$l_1 = 1$ $l_1 = 2$ $l_1 = 3$

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Training: inc probability of correct class
dec probability of incorrect classes

Softmax



Cross entropy loss

Label l_i

Goal: increase q_j if and only if $j = l_i$

$$\mathcal{L}_i = \sum_j -Y_{ij} \log q_j = -\log q_{l_i}$$

Loss on training example i

One hot encoding of labels

$$l_i \rightarrow \begin{array}{ll} y_{ij} = 1 & \text{if } j = l_i \\ y_{ij} = 0 & \text{if } j \neq l_i \end{array}$$

$l_1 = 1$ $l_1 = 2$ $l_1 = 3$

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Cross entropy, entropy and KL divergence

Y_j correct distribution over C labels

q_j network's distribution over C labels

$$\mathcal{L} = -\sum_{j=1}^{C} Y_j \log q_j = -\sum_{j} Y_j \log Y_j + \sum_{j} Y_j \log \frac{Y_j}{q_j}$$

Cross entropy

Entropy of Y

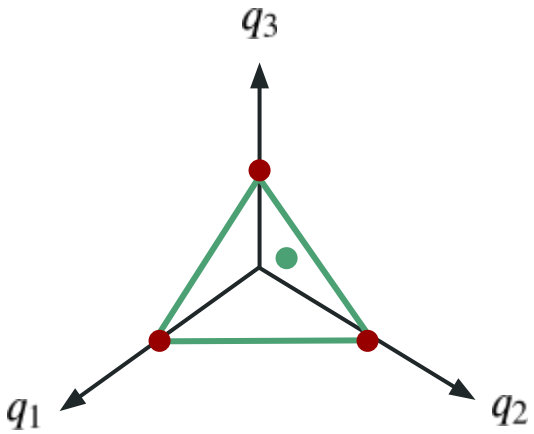
$H(Y)$

KL divergence

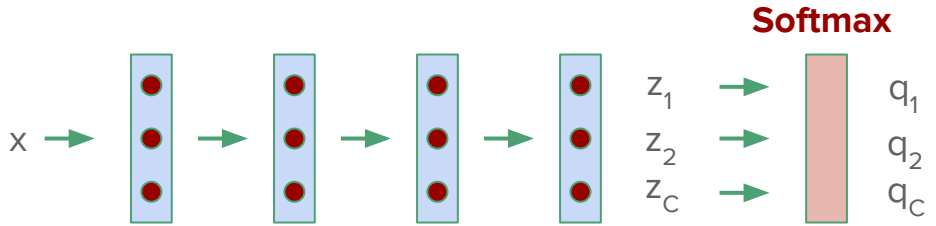
$D_{\mathrm{KL}}(Y||q)$

≥ 0

$= 0 \iff Y = q$



Training scheme for classification



$$\mathcal{L} = \sum_i \sum_j -Y_{ij} \log q_j(x_i, \mathbf{w})$$

Now your turn! You get to play with code to implement the softmax + cross entropy

Putting it together: training and evaluating an MLP in PyTorch

Lecture 5

Let’s train an MLP!

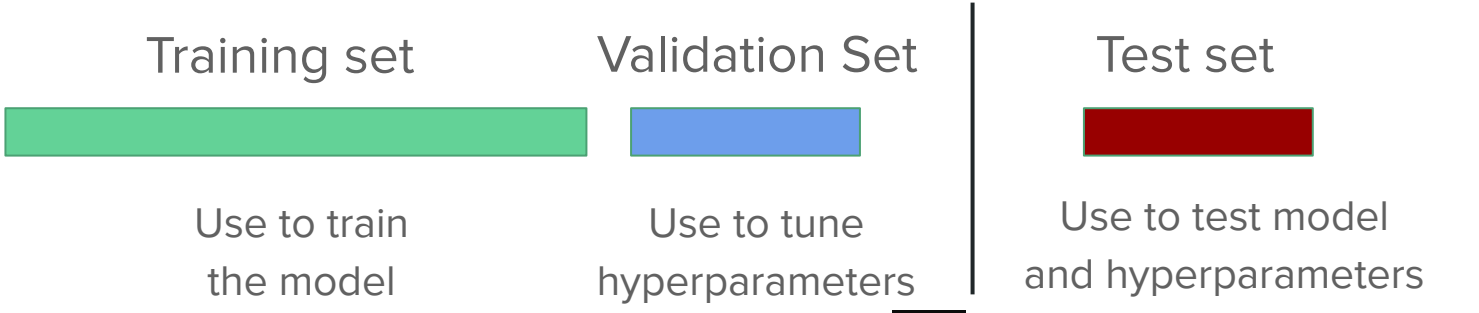
But wait, how do we know if it actually works after we train it?

Cross-validation to combat overfitting

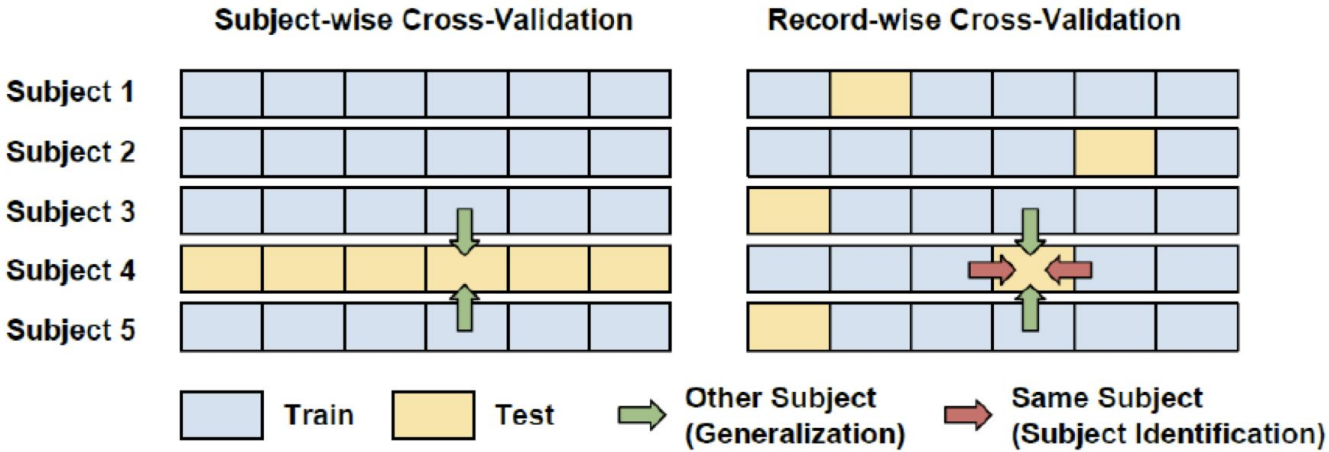
Training error < Test error (if much less then you are overfitting)

If you tune too many hyperparameters…

Training error < Test error (if much less then you are overfitting)



The cross-validation strategy must match the use case



Saeb ... Kording 2018

The evaluation metric must be meaningful

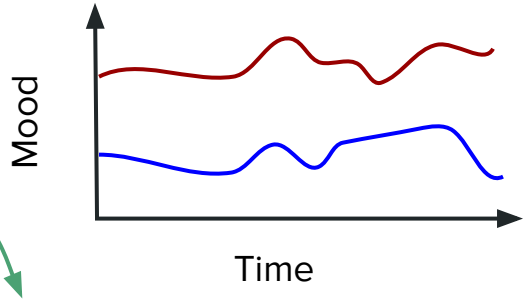
Example: Many people do mood estimates on mobile phones. $R^2 \sim .7$

$$R^2 = 1 - \frac{Var\ (\text{Model-Reality})}{Var\ (\text{Reality})}$$

Used complex subject specific models.

What is the denominator?

- Variance of reported mood across all subjects?
- Variance of reported mood within subject?



But trivial within subject mean can get average $R^2 \sim .7$

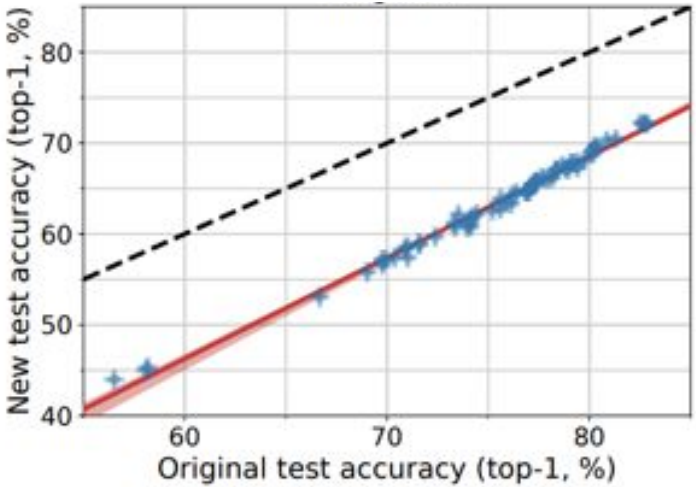
DeMasi, Kording, Recht

An entire field overfitting on a dataset?

Collect new images using same protocol as CIFAR10 / ImageNet

Get accuracy drops of ~ 10 percent

On ImageNet: corresponds to loss of 5 years of progress in performance



Recht et al 2019

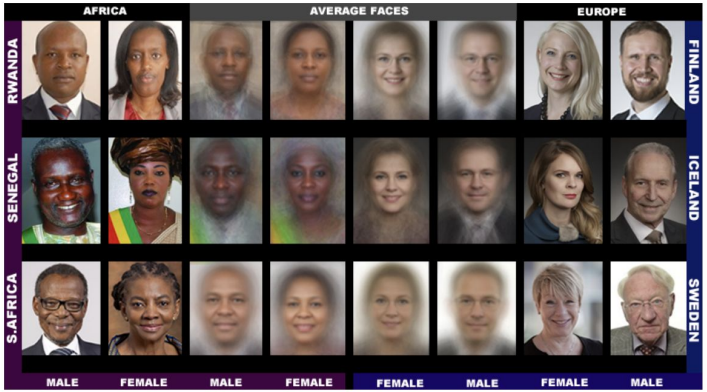
Overall accuracy on a test set is not enough: bias and fairness

3 commercial gender classification systems performed significantly worse on:

females compared to males

darker compared to lighter skinned faces

Need to make sure performance across important subpopulations is uniformly high



Buolamwini and Gebru, Conf on Fairness, Accountability and Transparency, 2018.

So be careful in training and evaluation!

Need to make important choices of:

What is the training set?

Does the split between train and test match your use case?

Is your metric for evaluation reasonable for real world deployment?

Will future validation data drift from train and test settings?

Are there biases due to problem selection, training data, algorithm design, evaluation metrics, or anywhere in ML pipeline?