

ML

Fundamentals



Instituto
Balseiro

Instituto Balseiro
19/09/2022

Luis G. Moyano - Fundamentos de ML
Instituto Balseiro





Programa de Becas Marie Skłodowska-Curie

hasta el 30 de septiembre de 2022

[Press centre](#) [Employment](#) [Contact](#)

TOPICS ▾

SERVICES ▾

RESOURCES ▾

NEWS & EVENTS ▾

ABOUT US ▾

Search



Information for applicants

IAEA Marie Skłodowska-Curie Fellowship Programme

Information for applicants

Introduction

The IAEA Marie Skłodowska-Curie Fellowship Programme (MSCFP) seeks to inspire and support young women to pursue a career in the nuclear field. To that end, the MSCFP provides scholarships to selected students studying towards a Master's degree in nuclear related subjects as well as internship opportunities facilitated by IAEA.

Who should apply? Eligibility criteria

The IAEA MSCFP is open to students from IAEA Member States who meet the following entry requirements:

- Female candidates;
- Accepted by or enrolled in an accredited university in Master's programme in a nuclear related field.

Preference will be given to applicants with above average academic credentials (75% or above or GPA > 3.0 out of 4.0).

Related resources

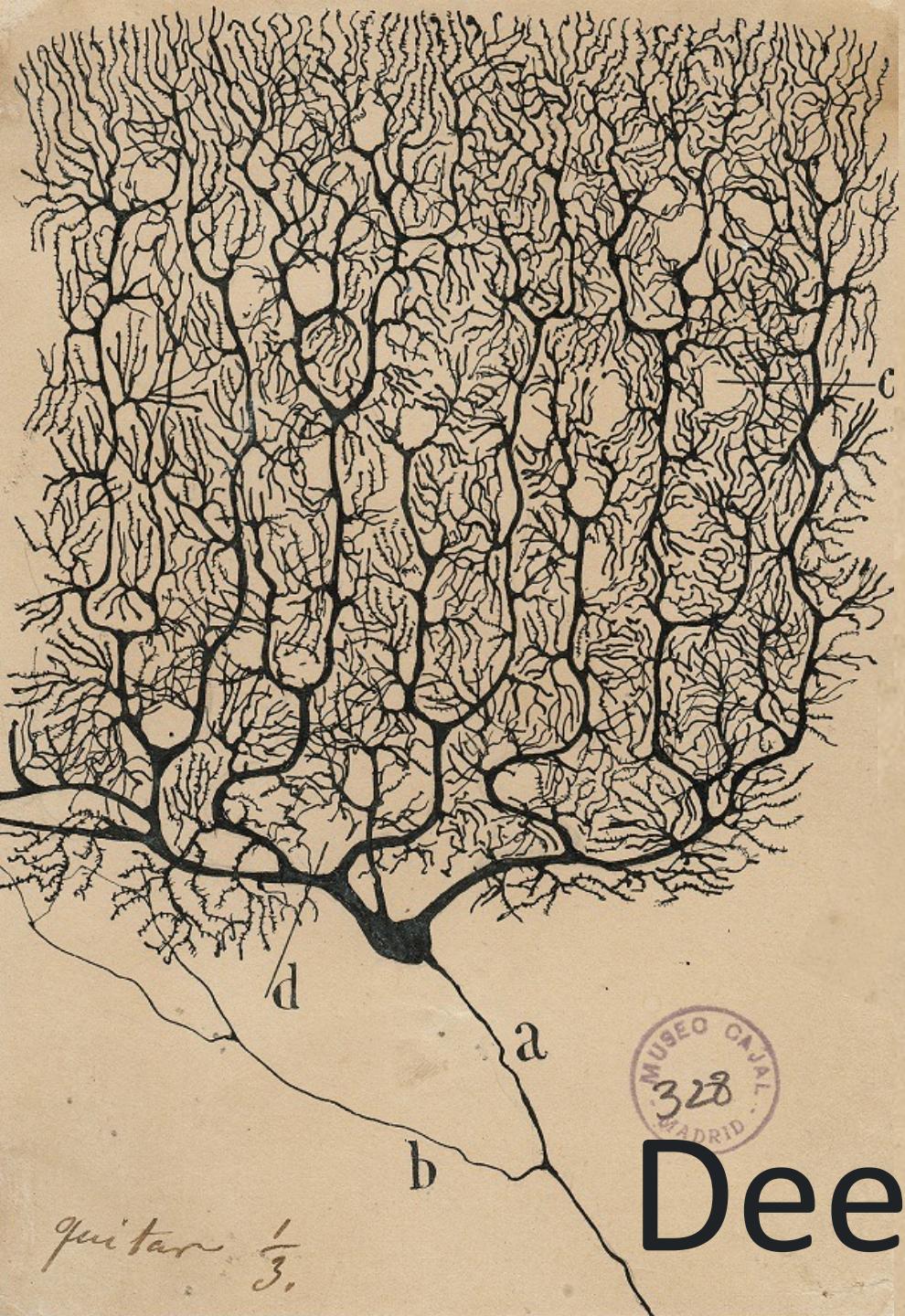
- [Applicant Reference Form](#)
- [Meet the Marie Skłodowska-Curie Fellowship Programme students](#)
- [IAEA Marie Skłodowska-Curie Fellowship Programme](#)

Frequently asked questions

- [Frequently asked questions](#)

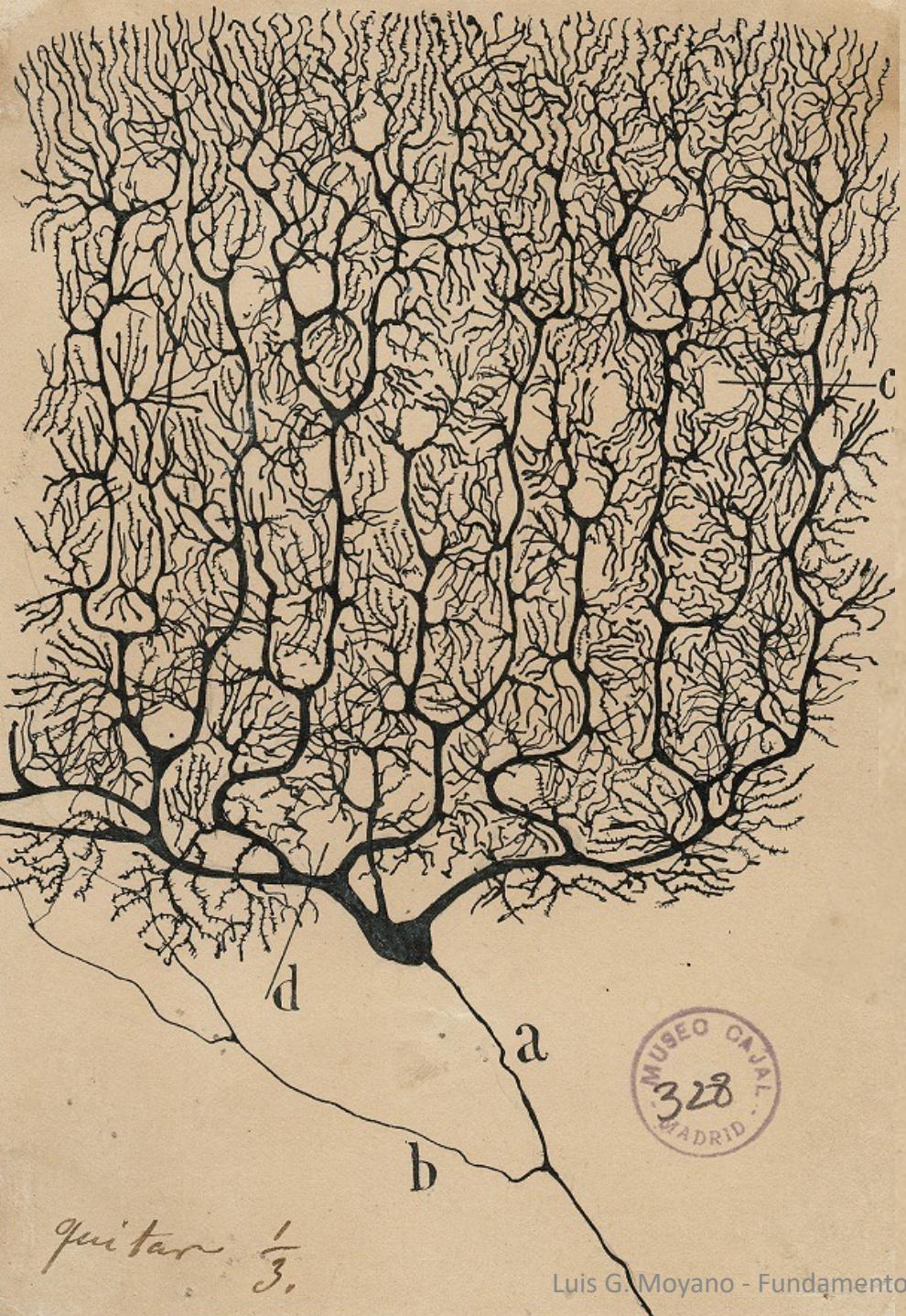
Contact

- ✉ [Send an email](#)



Lecture 9

Deep learning I



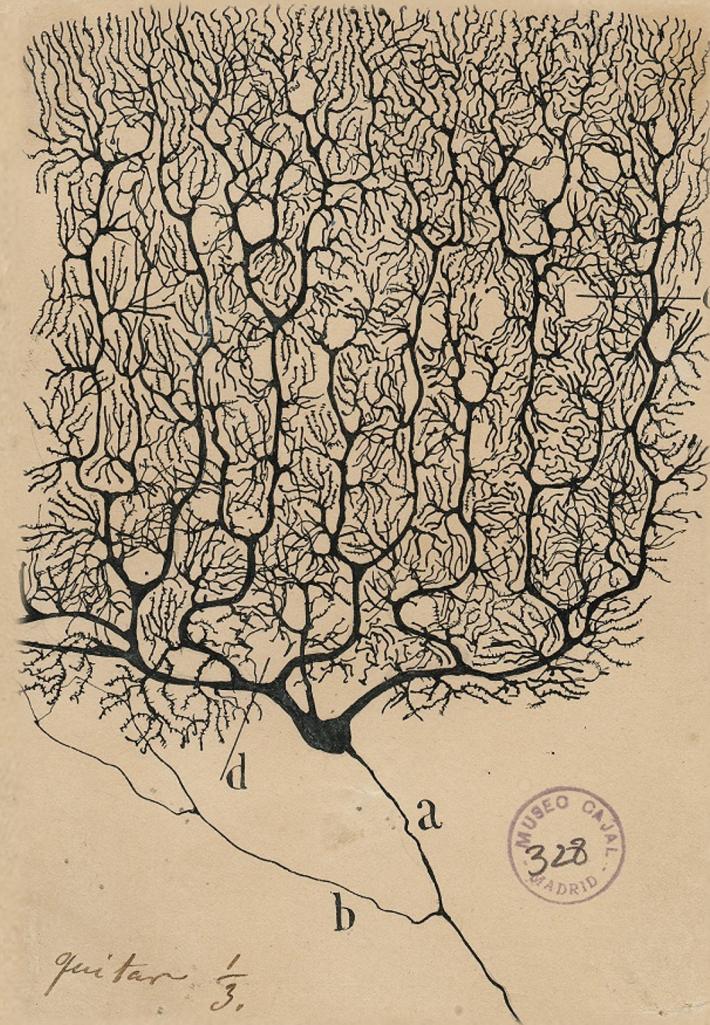
Guillermo 3.

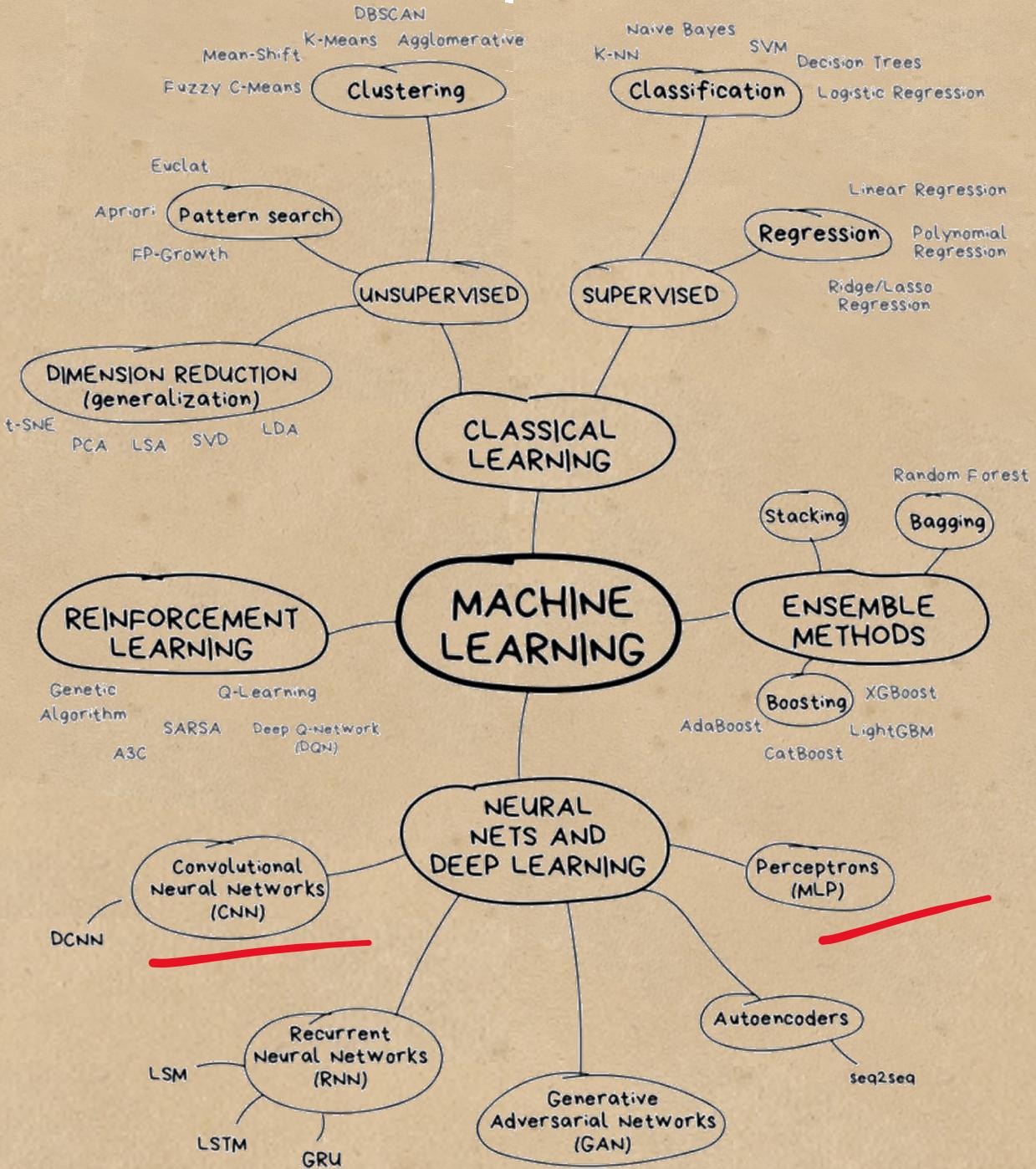
Luis G Moyano - Fundamentos de ML - Instituto Bakseiro



ML Fundamentals – Lecture 9

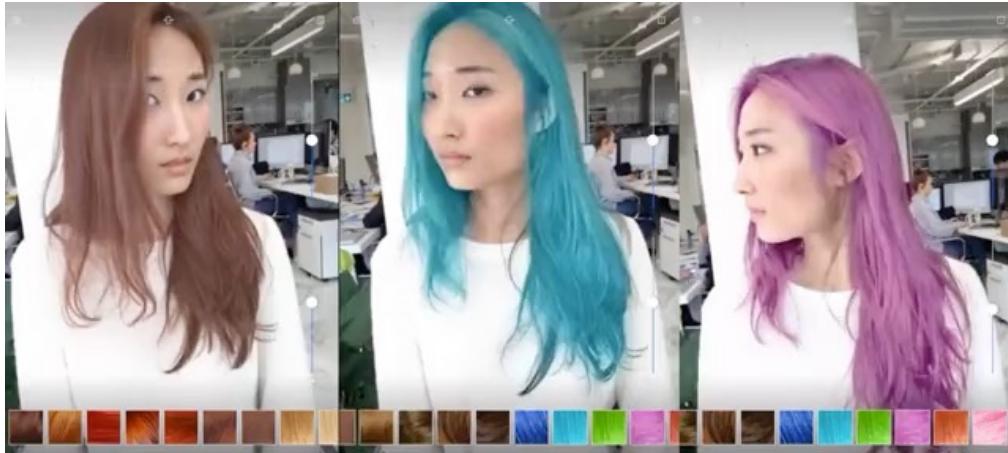
- Perceptron
- Feedforward nets
- Tensors & Keras 
- Architectures
- Training
 - Optimizers
 - Regularization







Why Deep Learning?



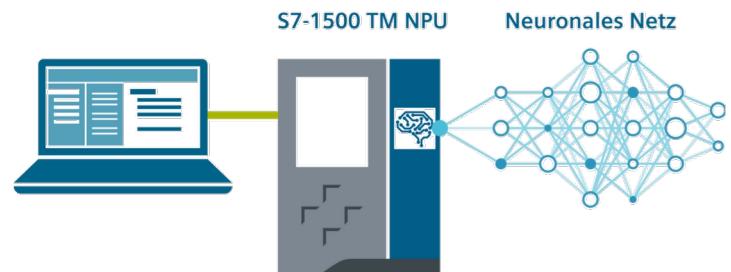
<https://news.developer.nvidia.com/3d-real-time-video-hair-coloration/>



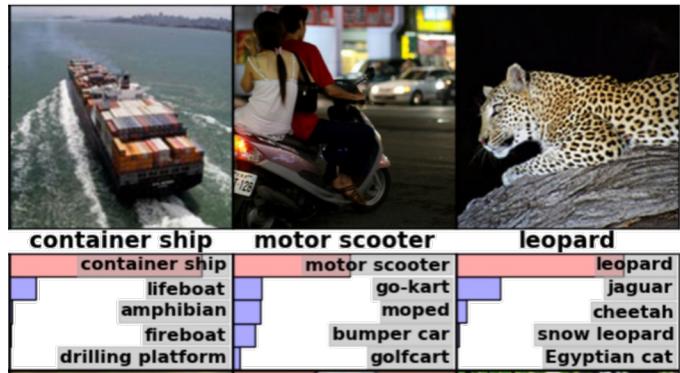
<https://news.developer.nvidia.com/new-app-uses-ai-to-enable-users-to-explore-sneakers-in-ar/>



Luis G. Moyano - Fundamentos de ML
Instituto Bakseiro



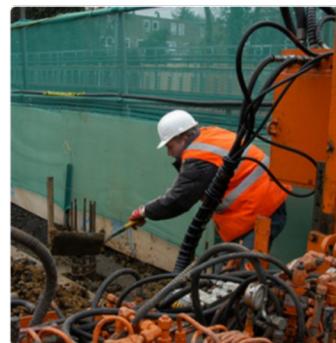
A few Deep Learning milestones



ions a leading CNN architecture made on images from the ImageNet dataset.



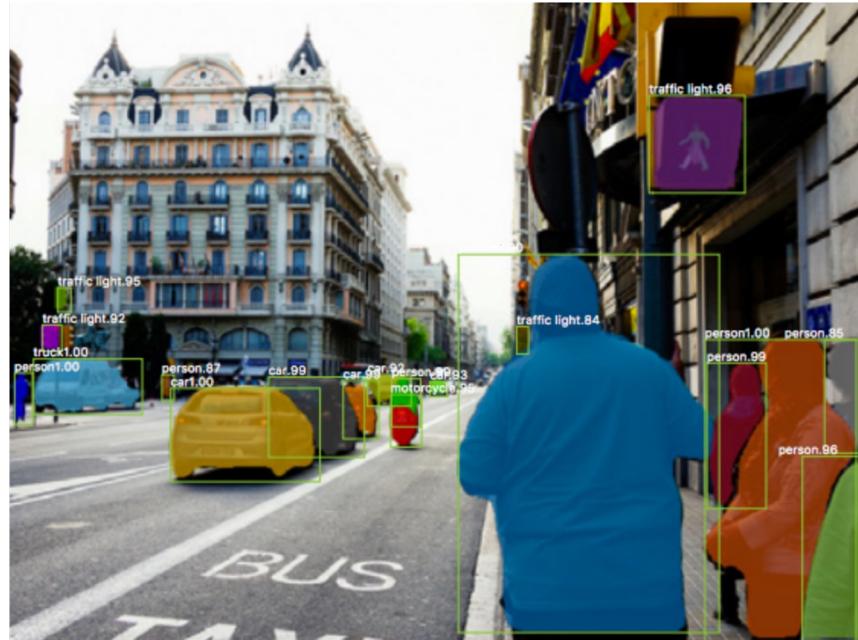
"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



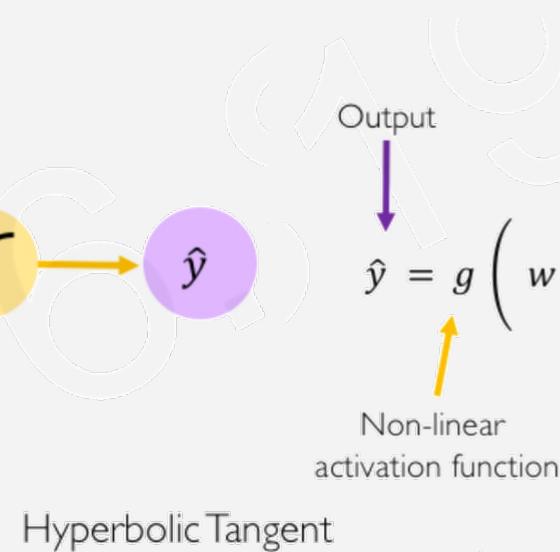
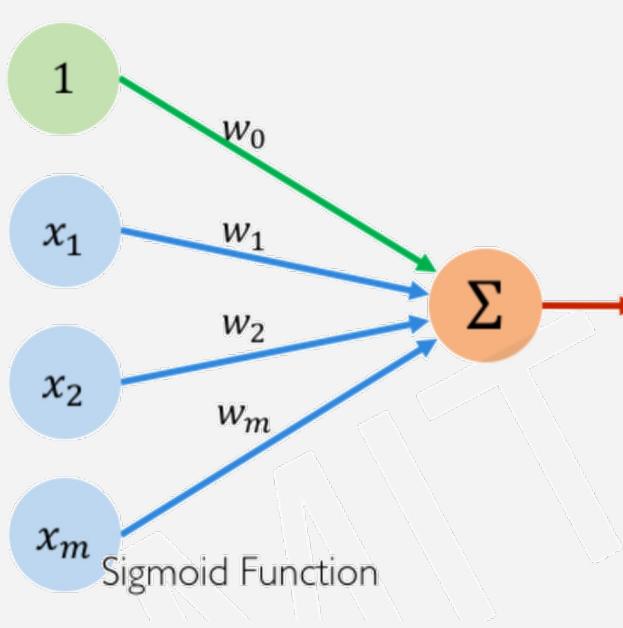
"two young girls are playing with lego toy."



- Near-human-level image classification
- Near-human-level speech transcription
- Near-human-level handwriting transcription
- Dramatically improved machine translation
- Dramatically improved text-to-speech conversion
- Digital assistants such as Google Now and Amazon Alexa
- Near-human-level autonomous driving
- Improved ad targeting, as used by Google, Baidu, or Bing
- Improved search results on the web
- Ability to answer natural-language questions
- Superhuman Go playing

The (single layer) perceptron

McCulloch-Pitts neuron, 1943
Rosenblatt, 1958



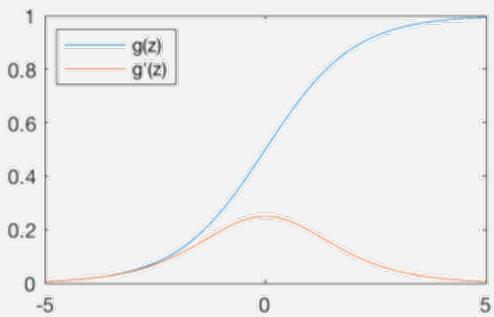
Linear combination of inputs

$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

Non-linear activation function

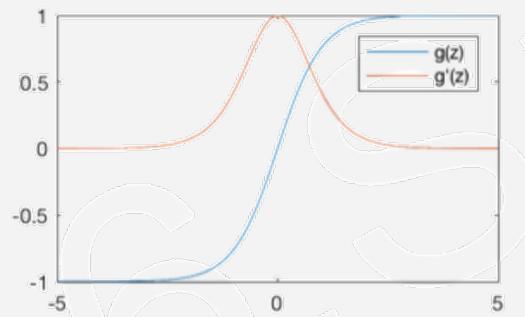
Bias

Rectified Linear Unit (ReLU)



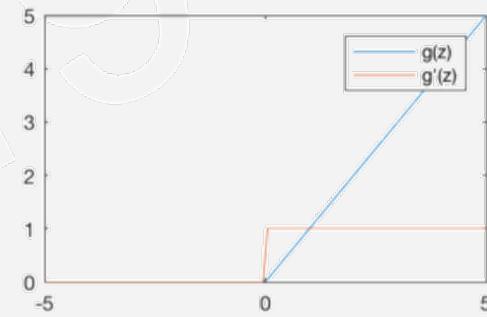
$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$



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

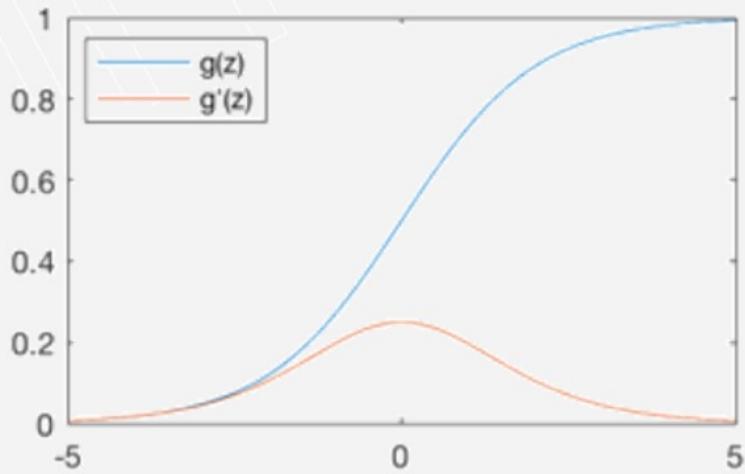
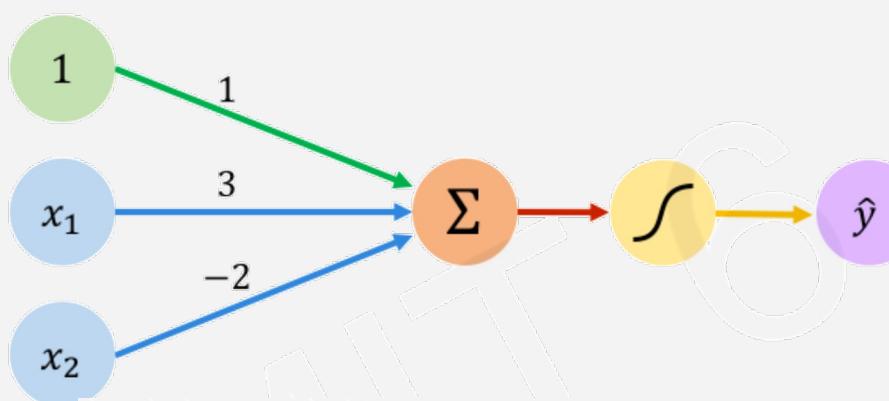
$$g'(z) = 1 - g(z)^2$$



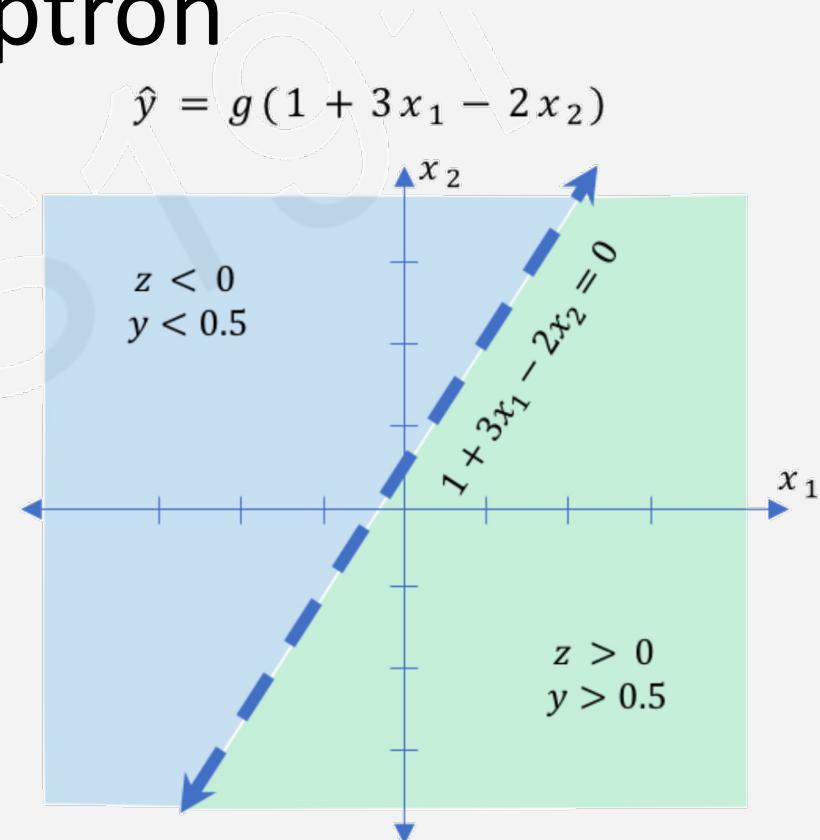
$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

The (single layer) perceptron

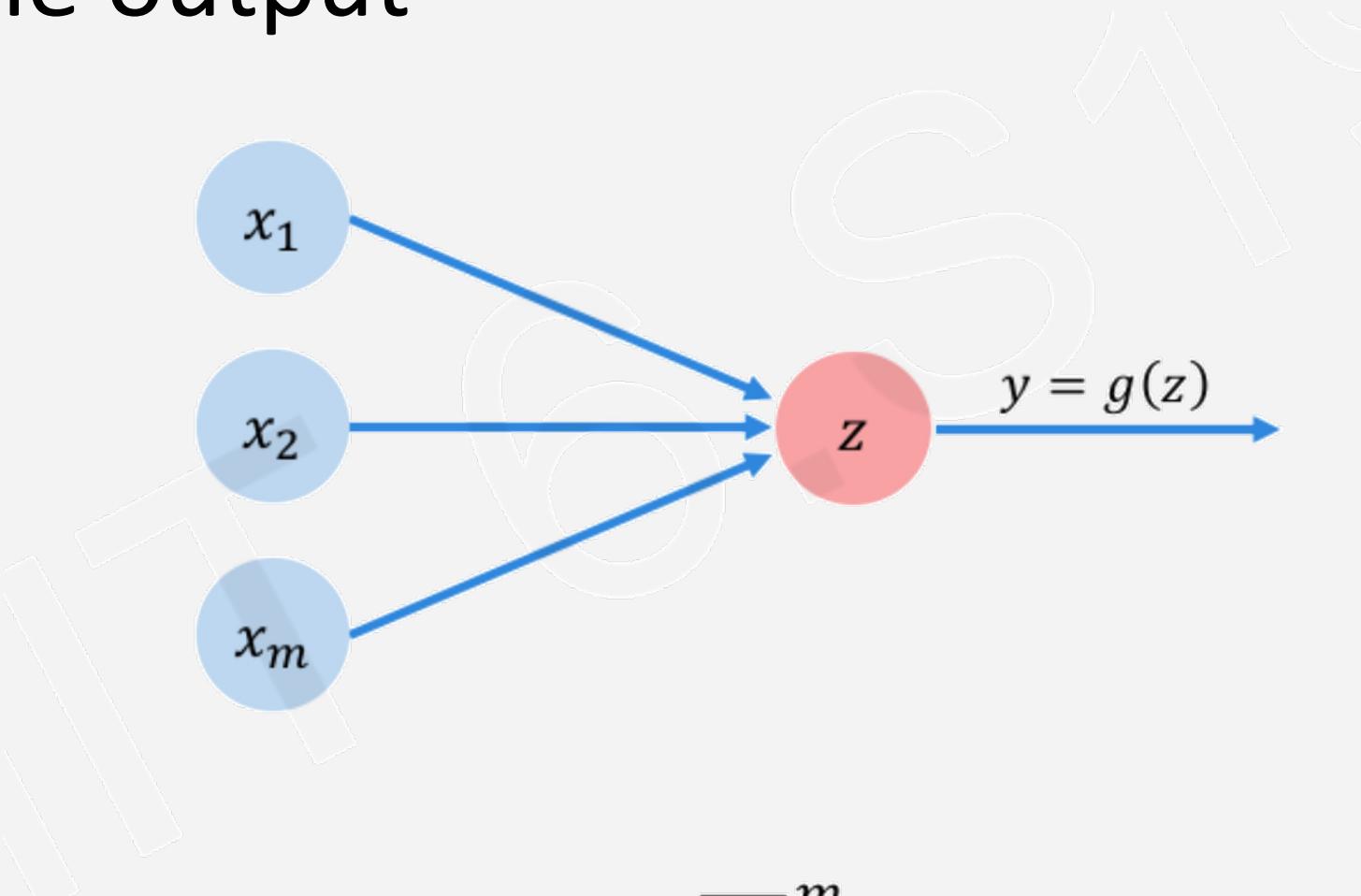


$$g(z) = \frac{1}{1 + e^{-z}}$$



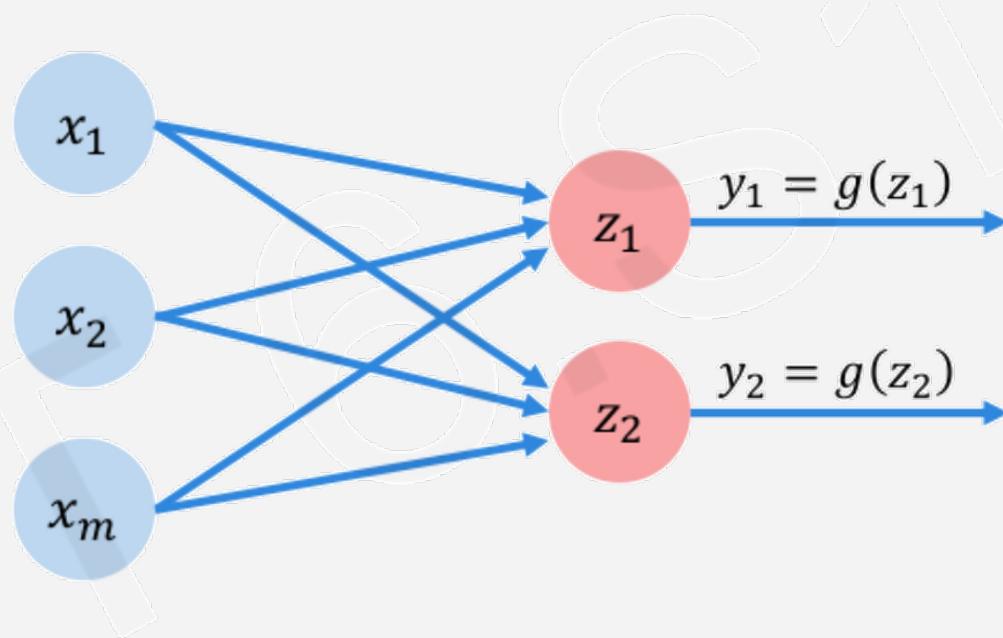
- The preceptron represents a single neuron
- Weights initialized randomly
- Then, updated through optimizer, (e.g. Gradient Descent)

One output



$$z = w_0 + \sum_{j=1}^m x_j w_j$$

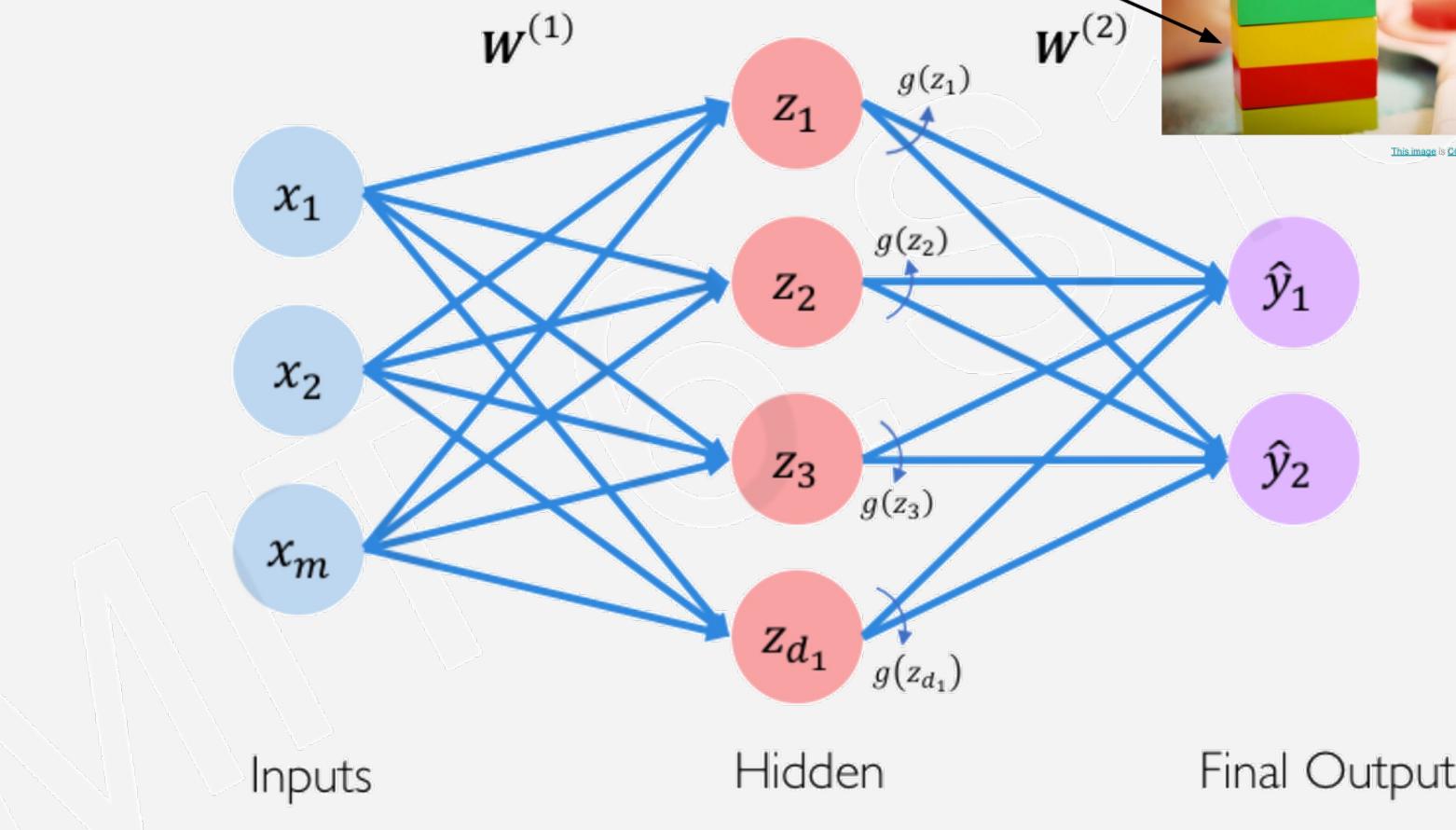
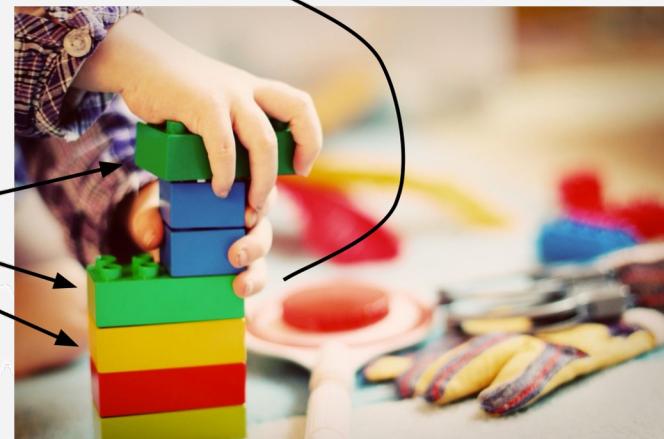
More than one output



$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

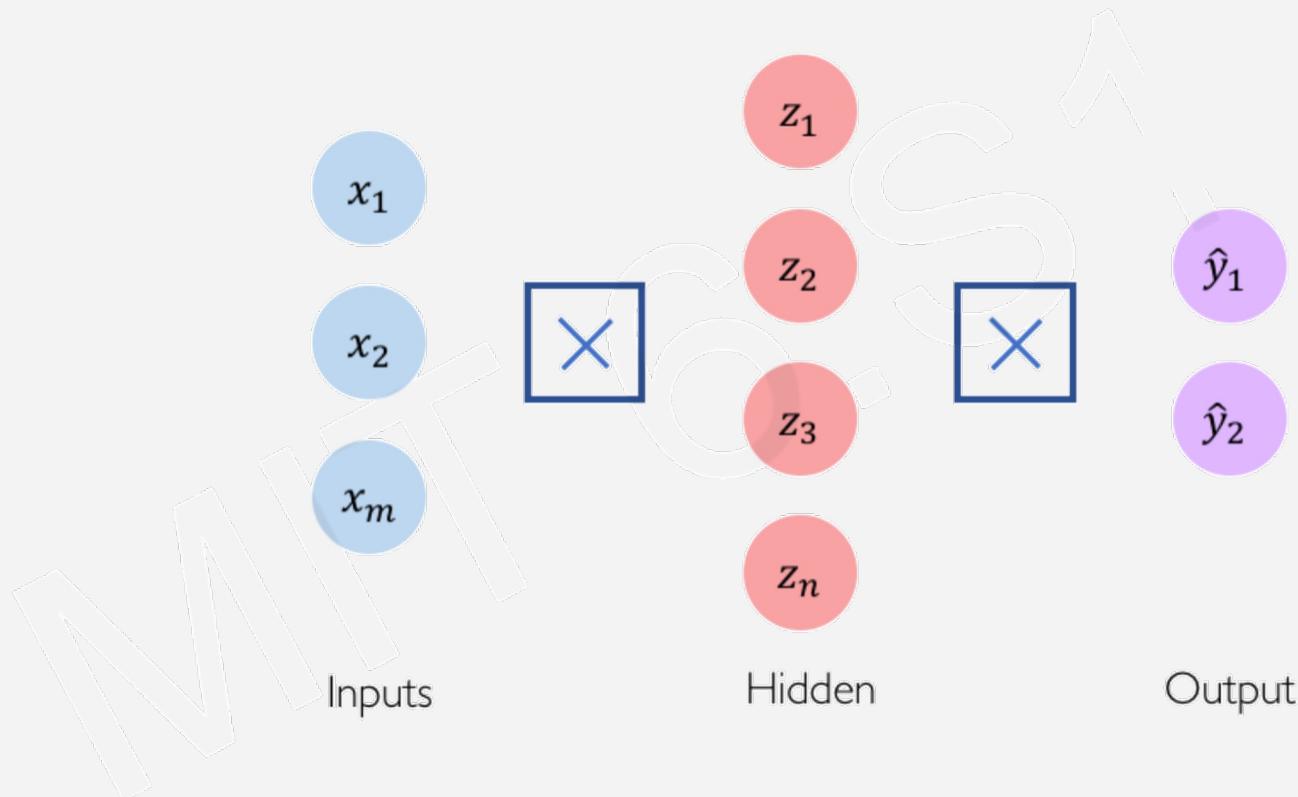
One hidden layer

Neural Network



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j w_{j,i}^{(1)} \quad \hat{y}_i = g \left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,i}^{(2)} \right)$$

Same, but simpler

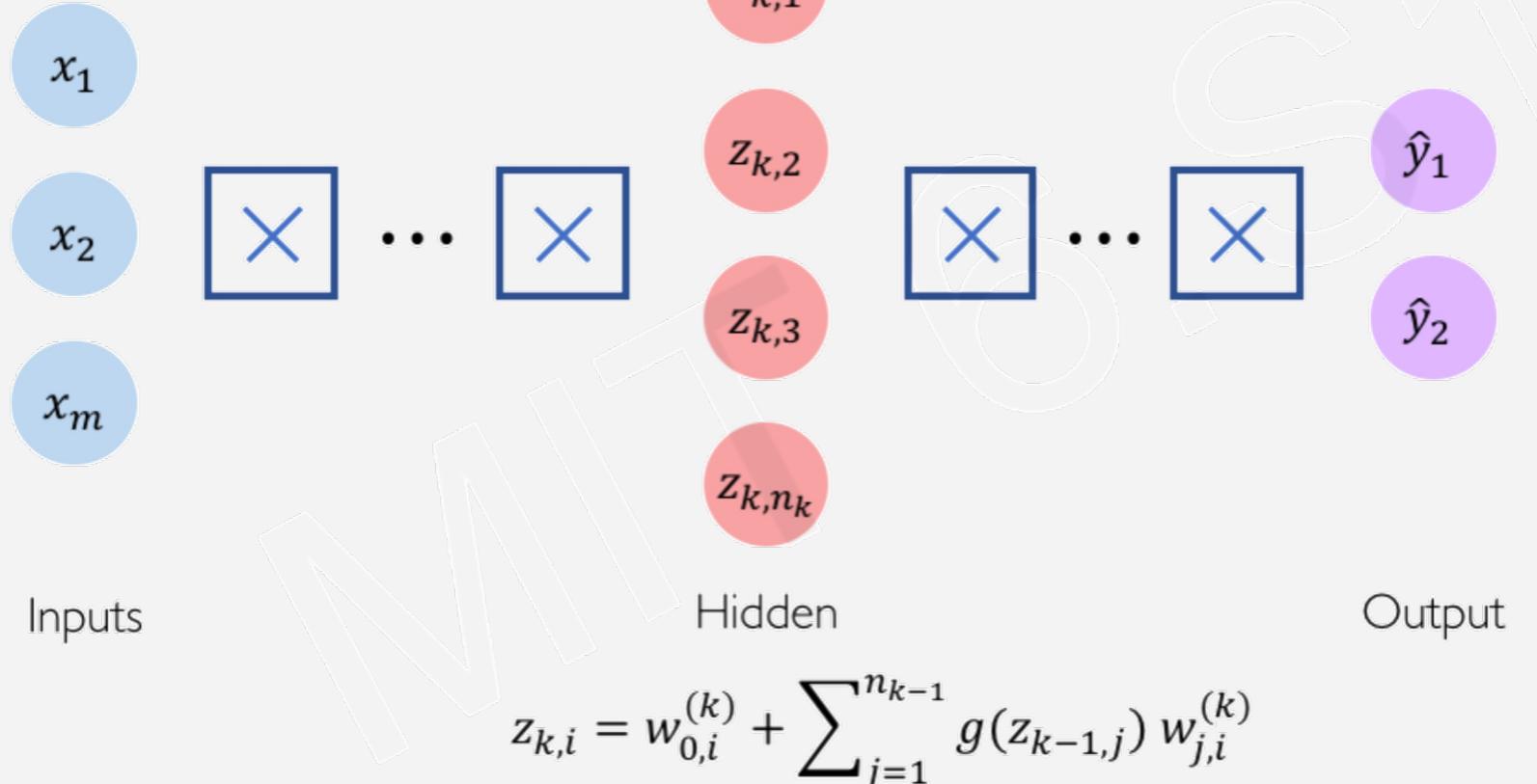


Feedforward Neural Network

- no cycles or loops
- information only moves forward

$$y = f(\mathbf{x}) = f^{(L)} \left(f^{(L-1)} \left(\dots f^{(1)}(\mathbf{x}) \dots \right) \right),$$

where $f^{(k)}: \mathbb{R}^{N_k} \rightarrow \mathbb{R}^{N_{k+1}}$.

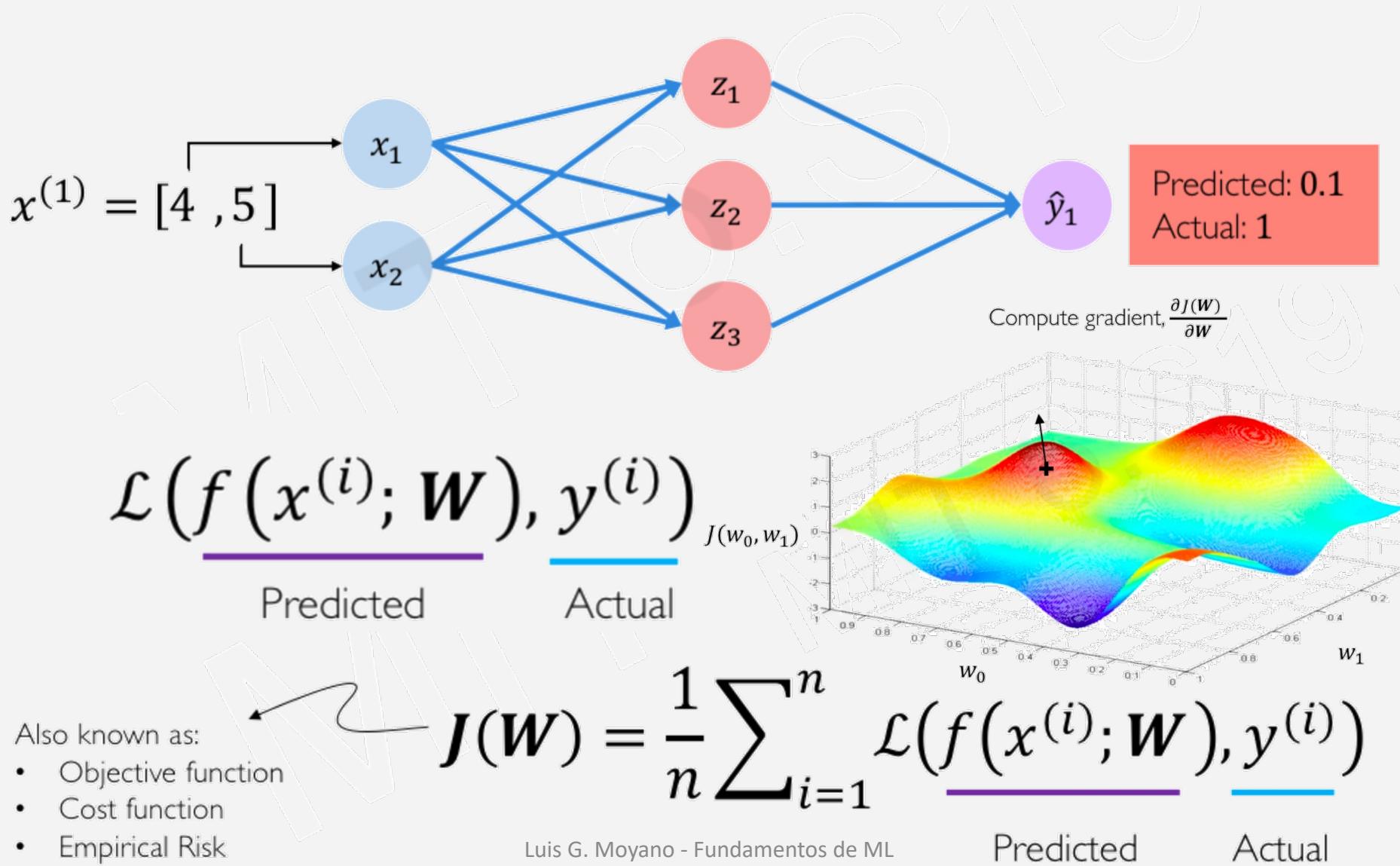


from keras import models
from keras import layers

```
model = models.Sequential()  
model.add(layers.Dense(32, input_shape=(784,)))  
model.add(layers.Dense(32))
```

K

Quantifying the error: the loss function



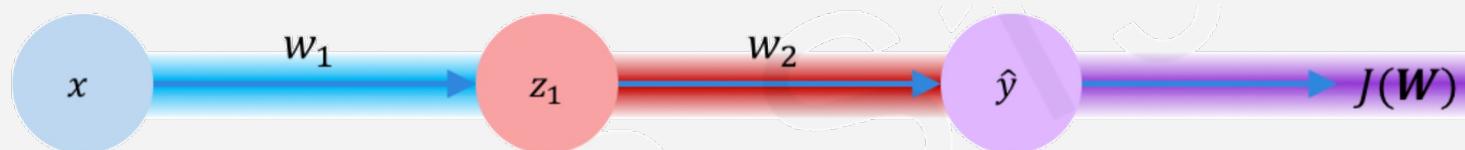
Backpropagation

Learning representations
by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California,
San Diego, La Jolla, California 92093, USA

† Department of Computer Science, Carnegie-Mellon University,
Pittsburgh, Philadelphia 15213, USA



$$\frac{\partial J(\mathbf{W})}{\partial w_1} = \frac{\partial J(\mathbf{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

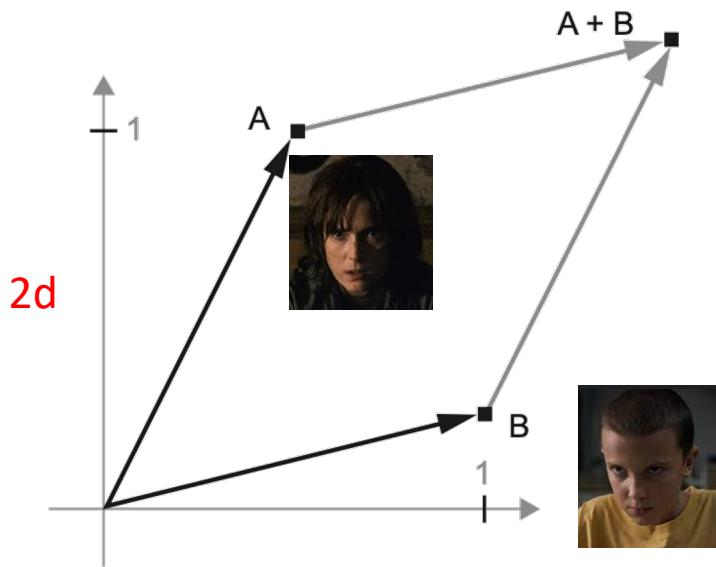
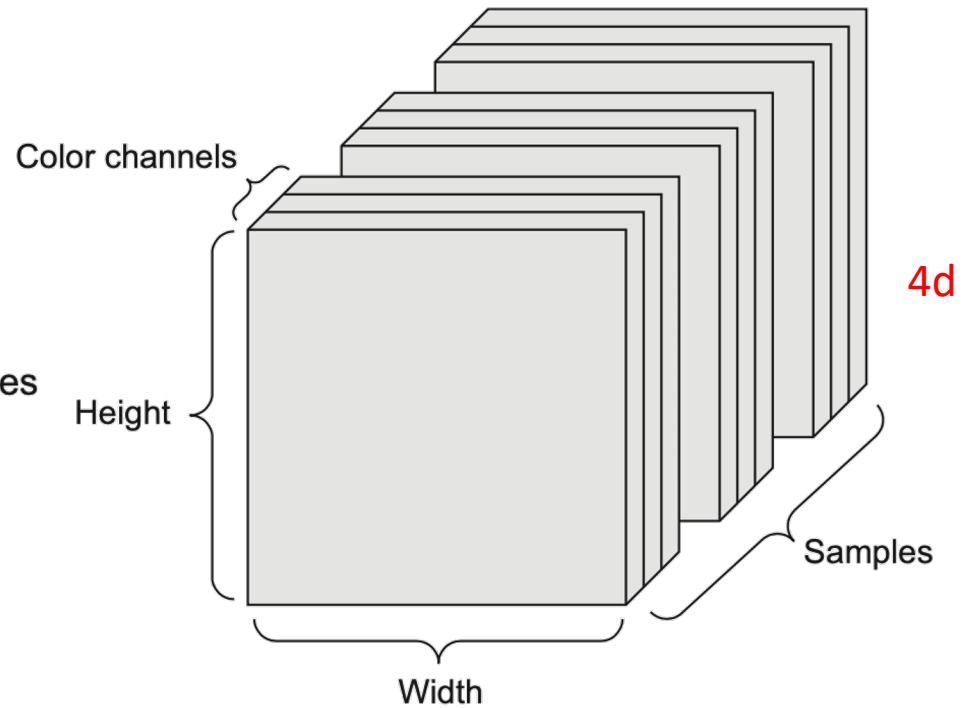
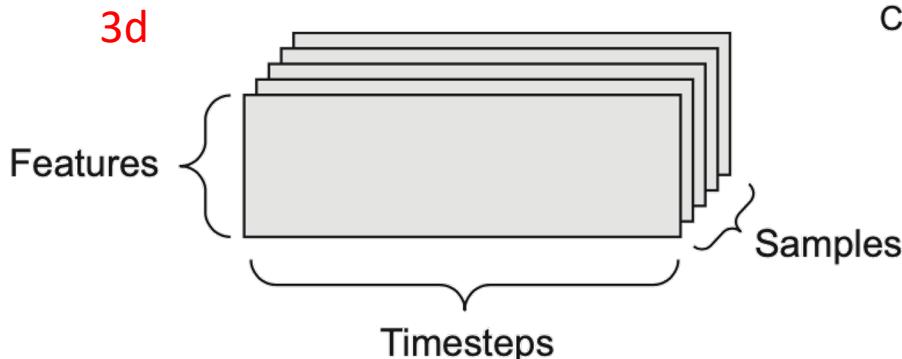
Repeat this for **every weight in the network** using gradients from later layers

Winona is a dot in feature space



$$\text{[Portrait of Winona]} \approx x_0 * \text{[Heatmap 1]} + x_1 * \text{[Heatmap 2]} + x_2 * \text{[Heatmap 3]} + x_3 * \text{[Heatmap 4]}.$$

Everything is a tensor



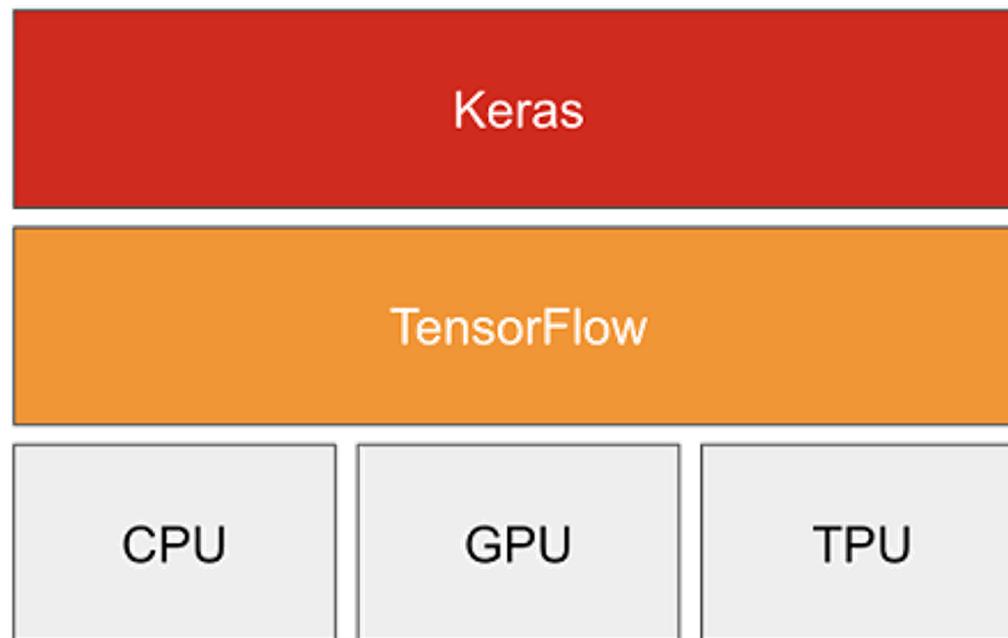
Batch dimension
↓

channels-last convention: (samples, height, width, color_depth)

K

Keras & TensorFlow

- **TensorFlow** is an Python ML platform by Google, free and open source.
- **Keras** is a Deep Learning API in Python, working as a wrapper over TensorFlow (and others), giving a convenient way to define and train models. **Keras models will transparently run on a single GPU**, with no code changes required.



Deep learning development:
layers, models, optimizers, losses,
metrics...

Tensor manipulation infrastructure:
tensors, variables, automatic
differentiation, distribution...

Hardware: execution



Luis G. Moyano - Fundamentos de ML - Instituto Bakseiro



Keras & TensorFlow

```
import tensorflow as tf
```

```
mnist = tf.keras.datasets.mnist
```

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

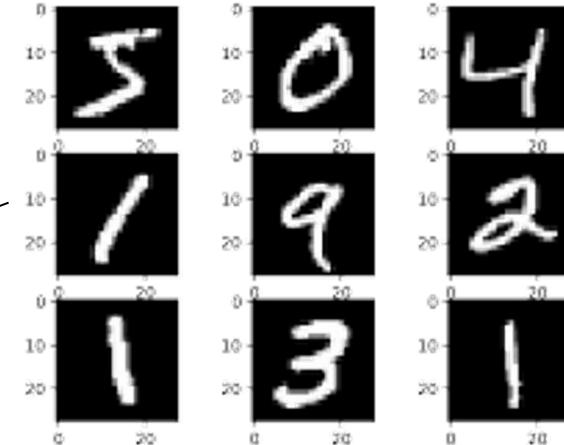
```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```
#setup simple shallow MLP
```

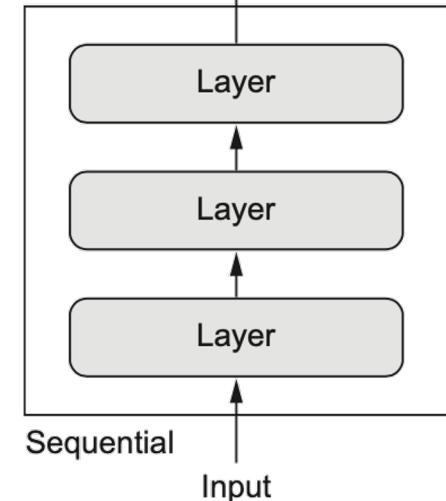
```
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(input_shape=(28, 28)),
tf.keras.layers.Dense(128, activation='relu'),
tf.keras.layers.Dropout(0.2),
tf.keras.layers.Dense(10)
])
```

```
model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossEntropy(from_logits=True), metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=10)
```

```
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
```



Output



Vectorization

- Use vectors not loops
- For keras/numpy arrays, we can use vector math instead of a loop:

$$c = a + b$$

- Gives an opportunity to execute vector addition in parallel
- Same with plenty of operations



```
for i in range(len(a)):  
    c[i] = a[i] + b[i]
```


$$\begin{array}{r} 3 \ 2 \ 5 \ 1 \\ + \ 1 \ 7 \ 4 \ 2 \\ \hline 4 \ 9 \ 9 \ 3 \end{array} \begin{matrix} a \\ b \\ c \end{matrix}$$

Vectorization in training loop

- Running one instance through network is how we think about it
- In practice, we send a subset or all X instances through the network in one go and compare all \hat{y} predictions to all y
 - Instead of looping through instances, we pass X through to use matrix-matrix multiplies instead of matrix-vector multiplies

```
for epoch in range(nepochs):  
    for i in range(n):  
        x = X[i]  
        y[i] = model(x)  
        ...
```

$$\frac{x}{3} = \frac{X}{3}[i] \quad y[i] = \frac{W}{3} @ \frac{x.T}{3}$$

```
for epoch in range(nepochs):  
    Y = model(X)  
    ...
```

$$Y = \frac{W}{3} @ \frac{X.T}{100}$$

Get 100 answers (mini batch)

Assume n=100, m=3, n_neurons=1 in 1x3 weight matrix W

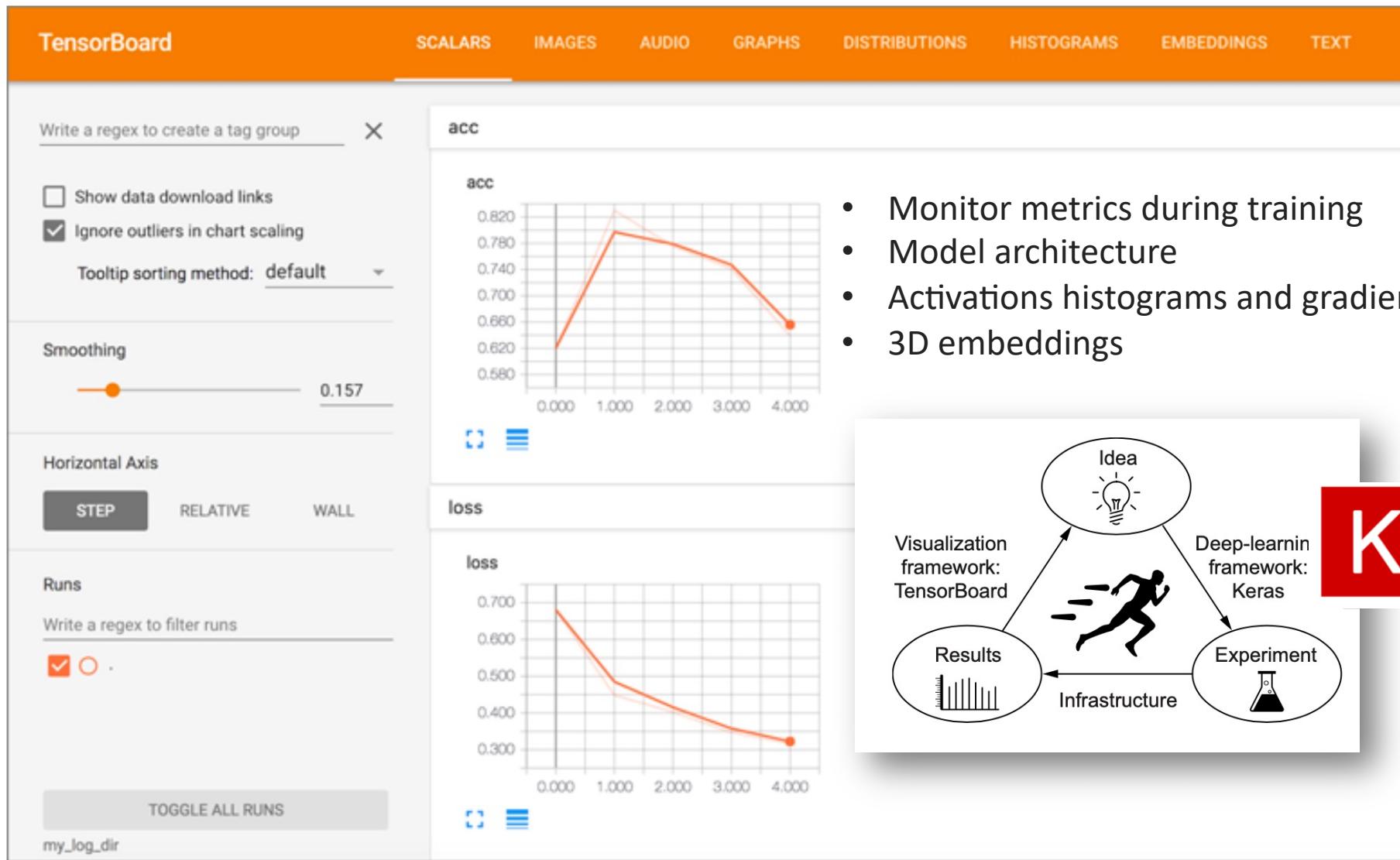
K

Keras callbacks

- *Model checkpointing*—Saving the current weights of the model at different points during training.
- *Early stopping*—Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training).
- *Dynamically adjusting the value of certain parameters during training*—Such as the learning rate of the optimizer.
- *Logging training and validation metrics during training*
- *Visualizing the representations learned by the model as they're updated*

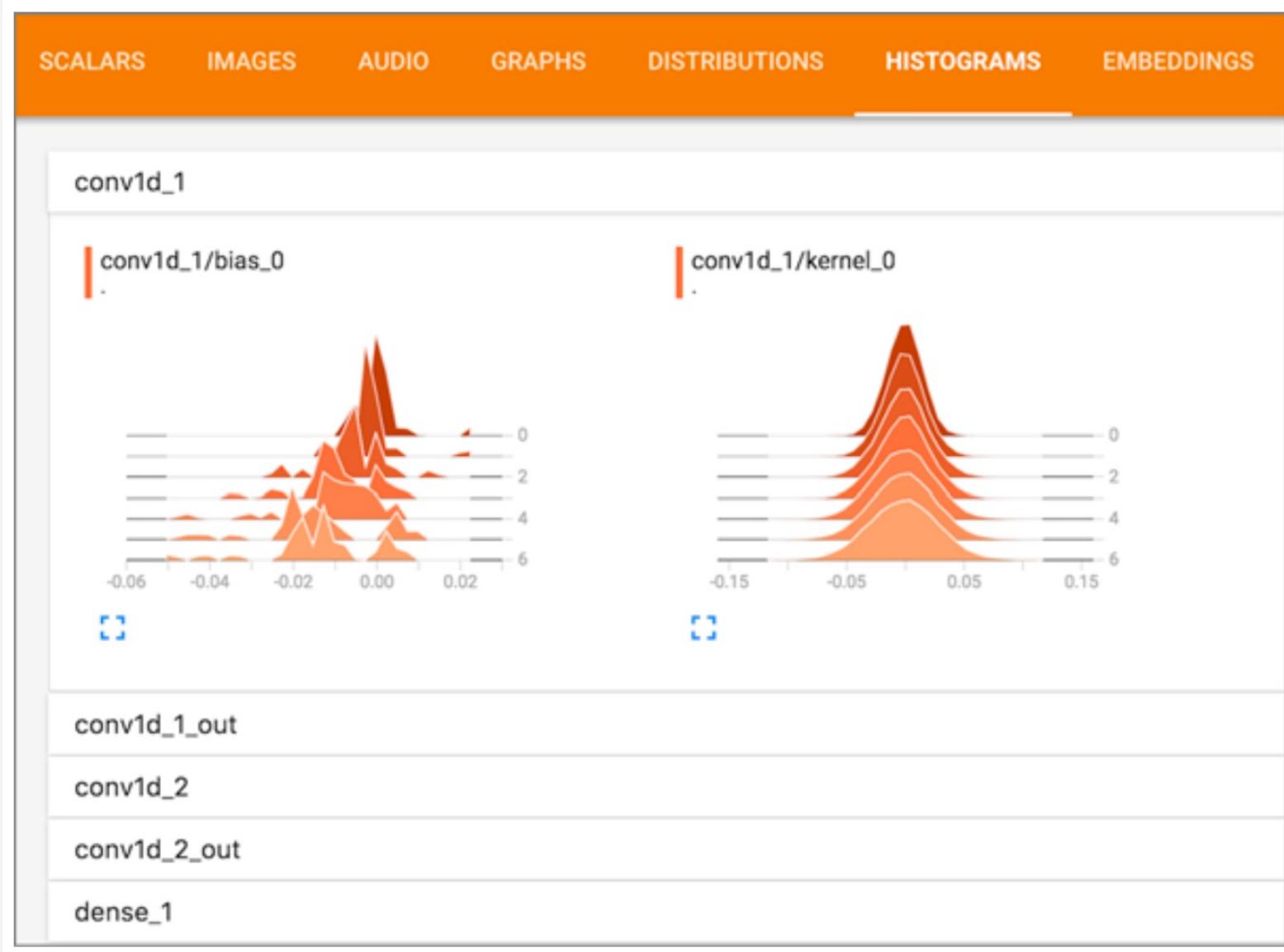
Tensorboard

Training metrics



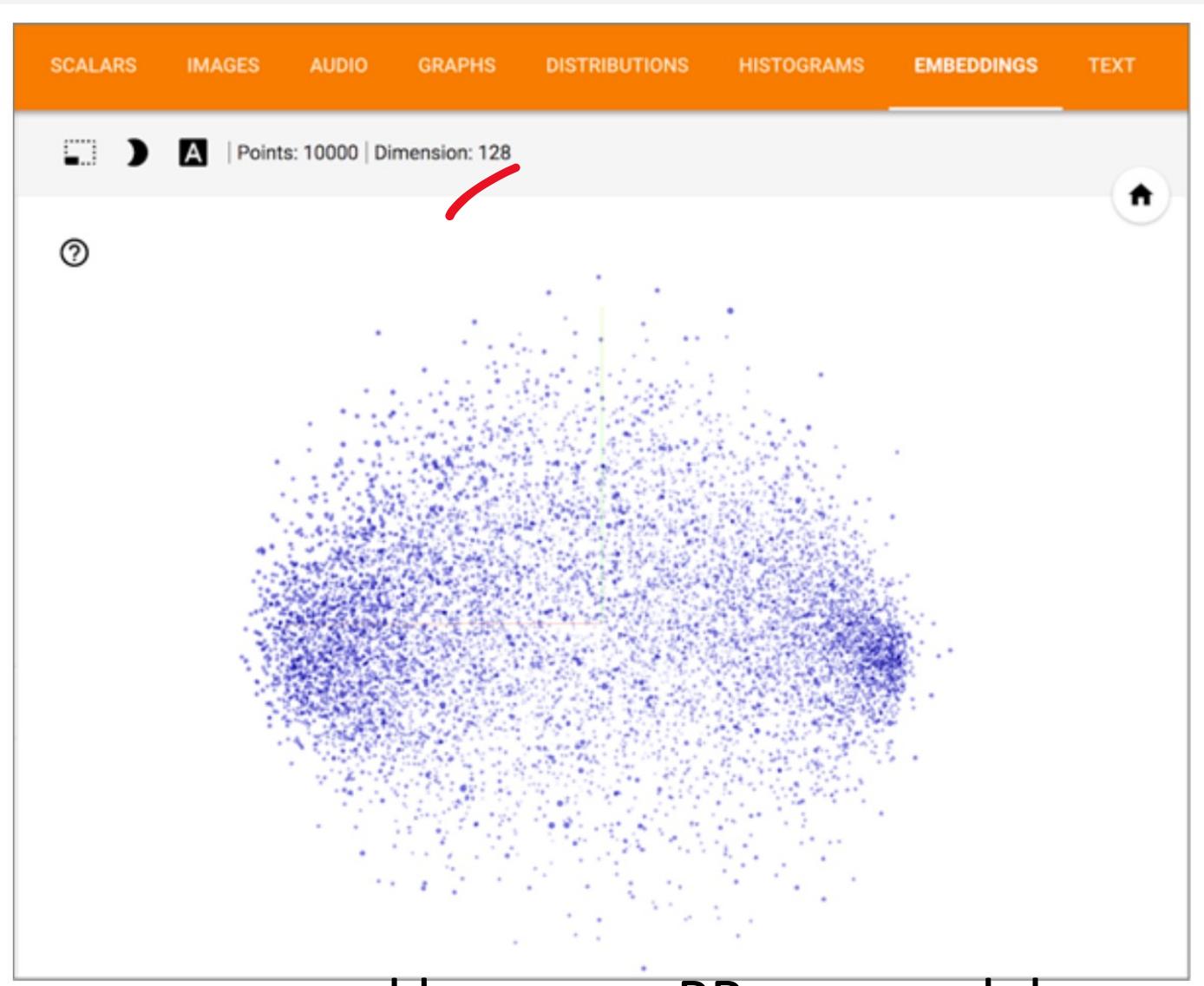
Tensorboard

Activation histograms



Tensorboard Embeddings

Visualize embedding with PCA or t-SNE

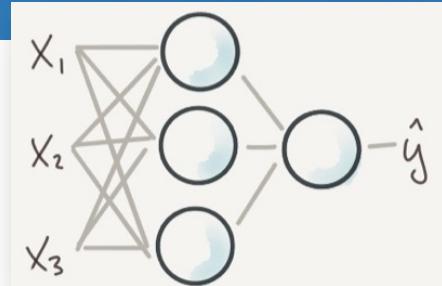


....and hparams, PR curves, debugger, & more



Luis G. Moyano - Fundamentos de ML -
Instituto Balseiro

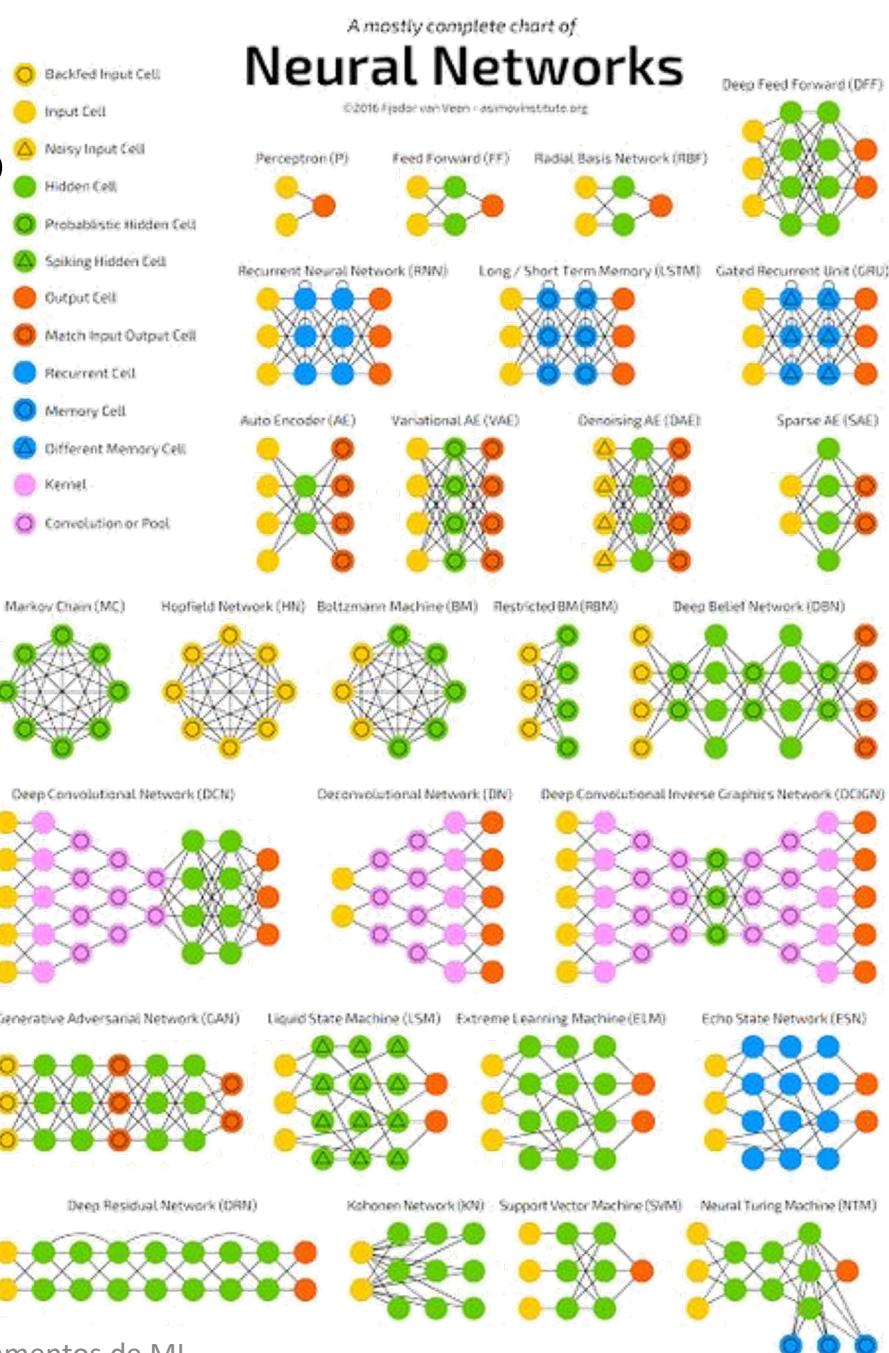
Network architectures



- **Vector data** —Densely connected network (Dense layers).
- **Image data**—2D convnets.
- **Sound data** (for example, waveform)—Either 1D convnets (preferred) or RNNs.
- **Text data**—Either 1D convnets (preferred) or RNNs.
- **Timeseries data**—Either RNNs (preferred) or 1D convnets.
- **Other types of sequence data**—Either RNNs or 1D convnets. Prefer RNNs if data ordering is strongly meaningful (for example, for timeseries, but not for text).
- **Video data**—Either 3D convnets (if you need to capture motion effects) or a combination of a frame-level 2D convnet for feature extraction followed by either an RNN or a 1D convnet to process the resulting sequences.
- **Volumetric data**—3D convnets.

NN architectures

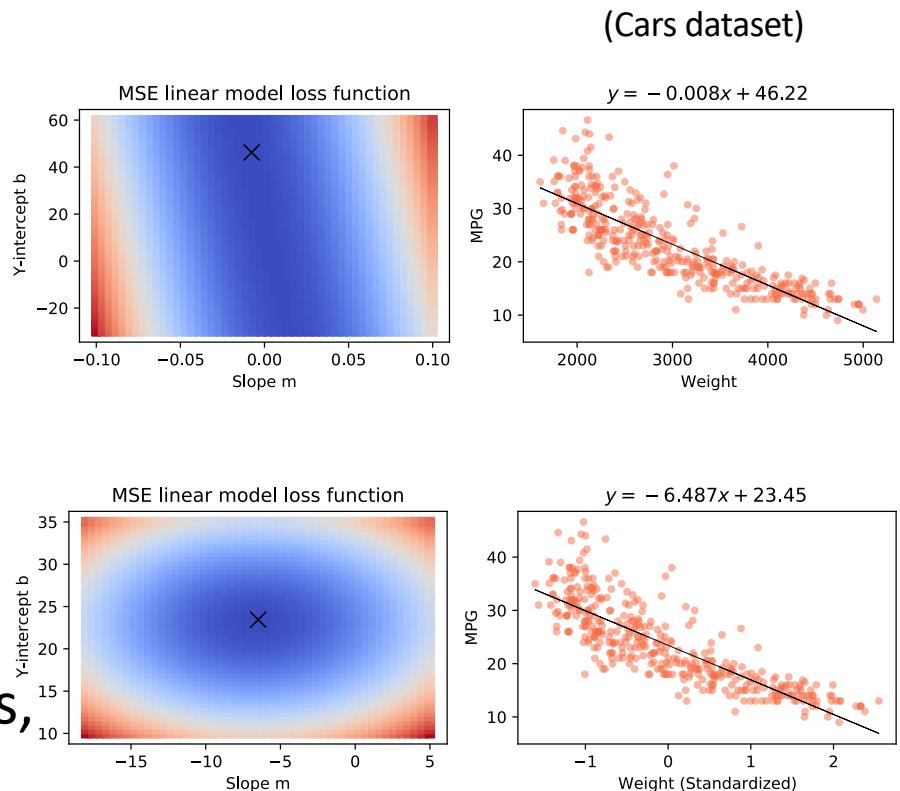
- Standard
- CNN-ConvNets
 - images
- RNN-Recurrent networks
 - sequence of inputs
- GAN-Generative antagonistic networks
 - unsupervised (originally)
- Autoencoders
 - unsupervised (originally)



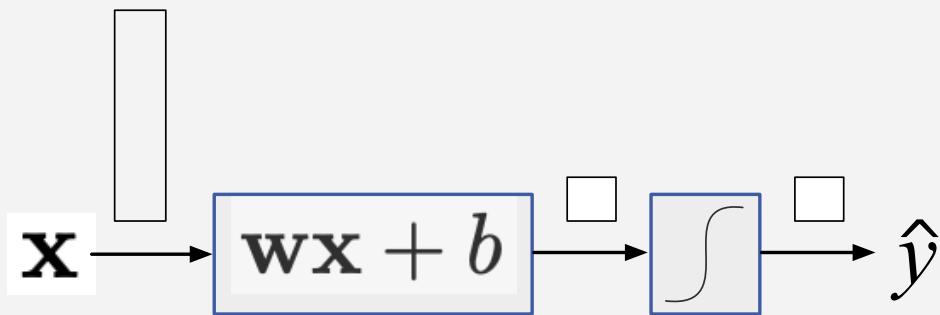
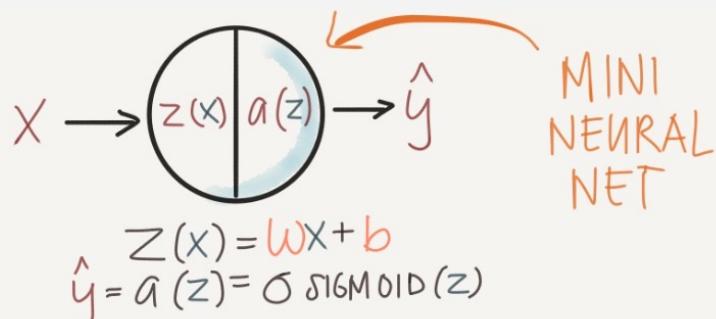
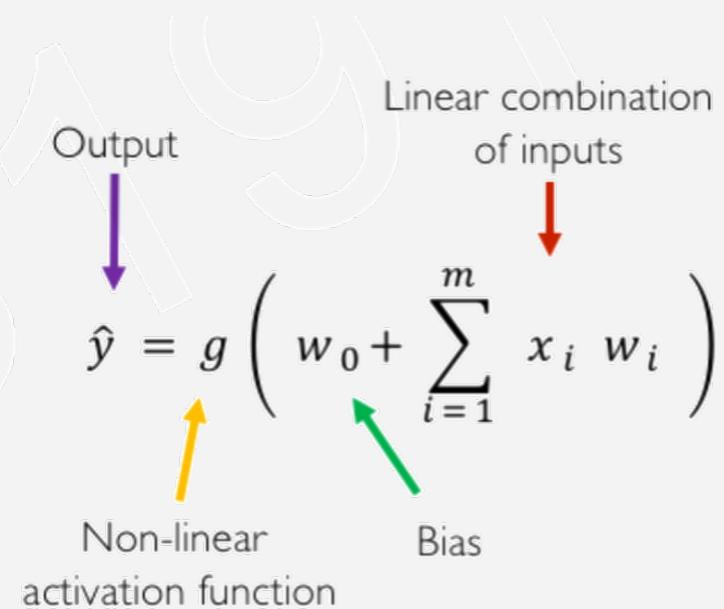
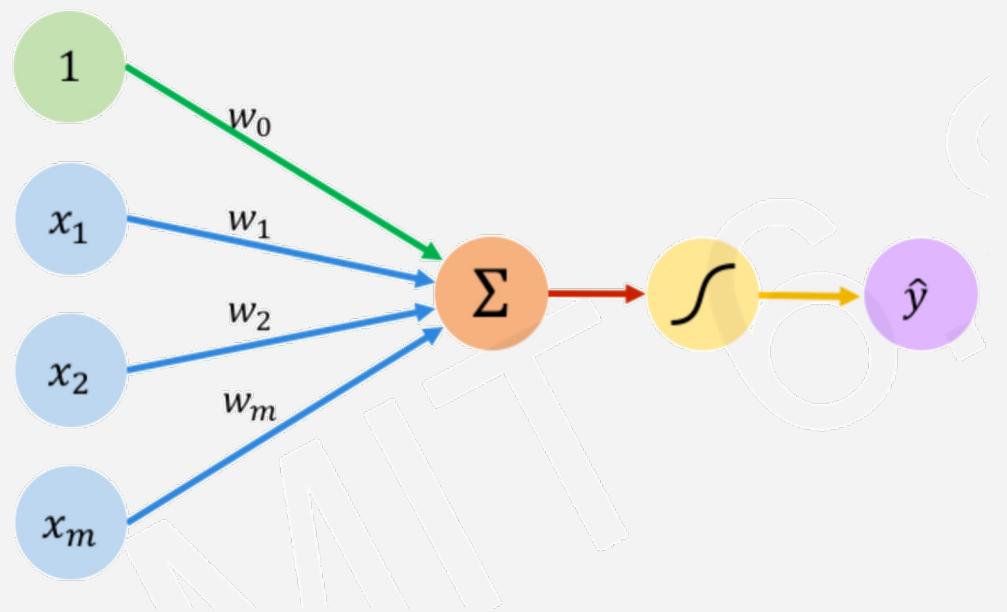
Preparing data

$$\begin{array}{|c|} \hline X \\ \hline \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_n \\ \hline \end{array} \quad \begin{array}{|c|} \hline y \\ \hline y_1 \\ y_2 \\ \vdots \\ y_n \\ \hline \end{array}$$

- Everything must be numeric
- No missing values
- Dummy encode categoricals (`pd.get_dummies()`)
- Should normalize numeric features in X to zero-mean, variance one ("whitening")
 - Speeds up training
- Compare regression equations, loss function surfaces

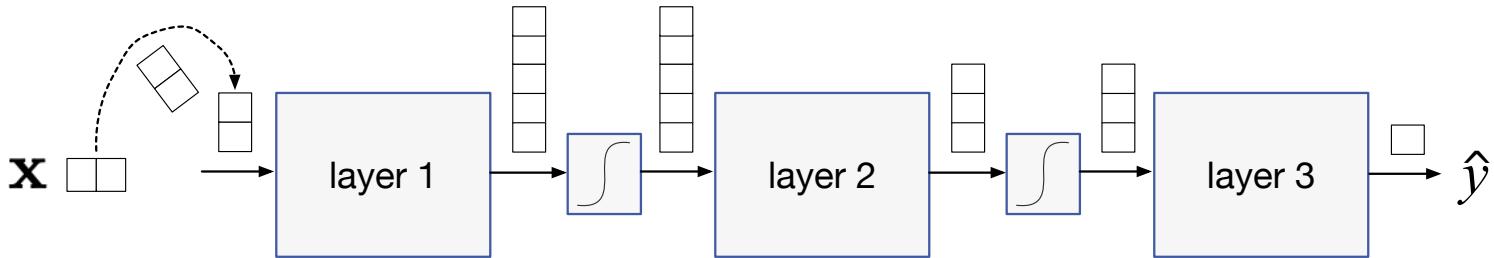


Equivalent Neural Networks depictions



What's a neural network?

- Ignore the neural network metaphor, but know the terminology
- We already saw a neural network is a combination of linear and nonlinear transformations
 - Linear: $z^{[layer]} = W^{[layer]} \mathbf{x}^T + \mathbf{b}^{[layer]}$
 - Nonlinear: $\mathbf{g}^{[layer]} = \sigma(z^{[layer]})$; σ called *activation function*
- Networks have multiple *layers*; layer is a stack of *neurons*



- Transform raw \mathbf{x} vector into better and better features, final linear layer can then make excellent prediction

DL Building blocks

$$\mathbf{w}\mathbf{x} + b$$

linear

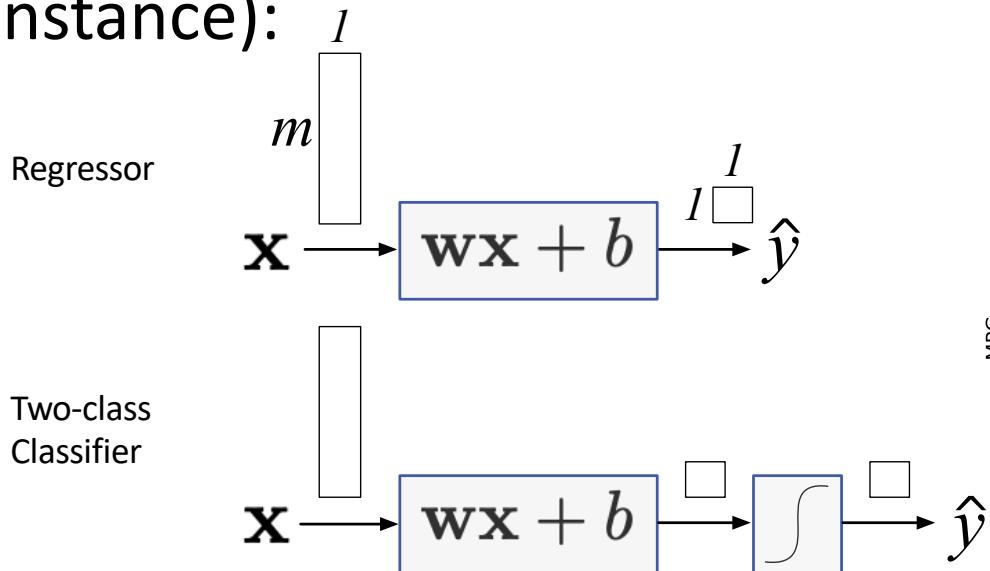


sigmoid

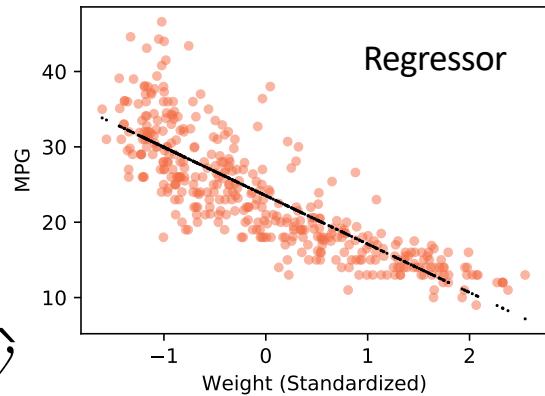


ReLU (rectified linear unit)

- $\hat{y} = w_1x_1 + w_2x_2 + \dots + w_mx_m + b = \mathbf{w}\mathbf{x}^T + b$
for $n \times m$ dimensional X
- Linear/logistic regression equivalents (one x instance):



Assume we magically know w and b



Underfitting a bit here
(need more of a quadratic)

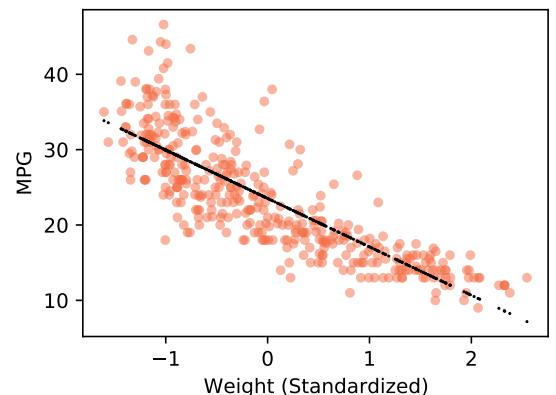
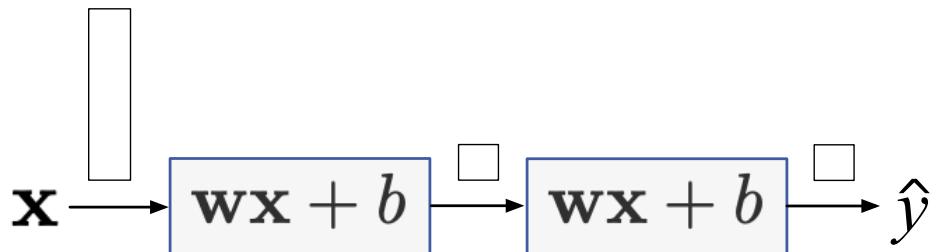
(For simplicity, we are using proper $\mathbf{w}\mathbf{x}^T$ in math but omitting transpose in diagrams)

Try adding layers to get more power

- But, sequence of linear models is just a linear model

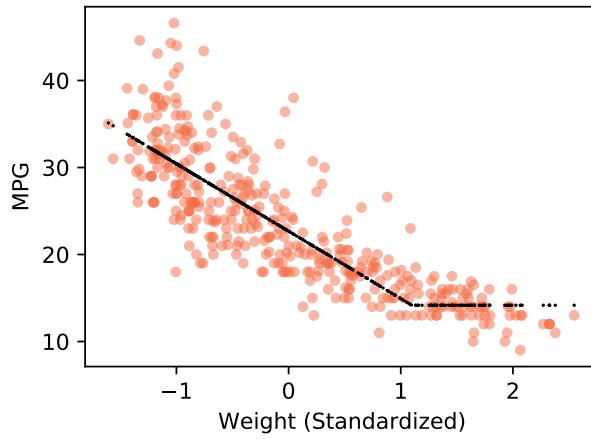
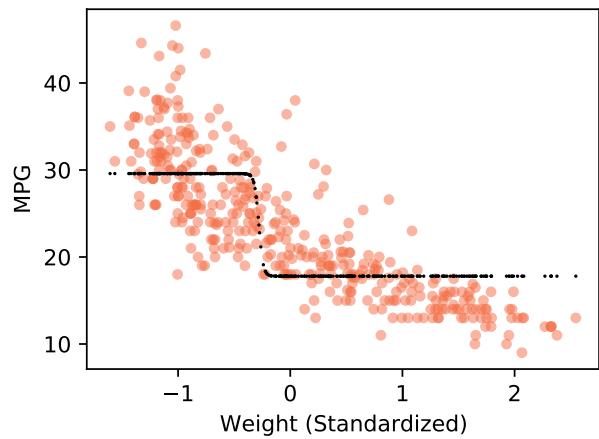
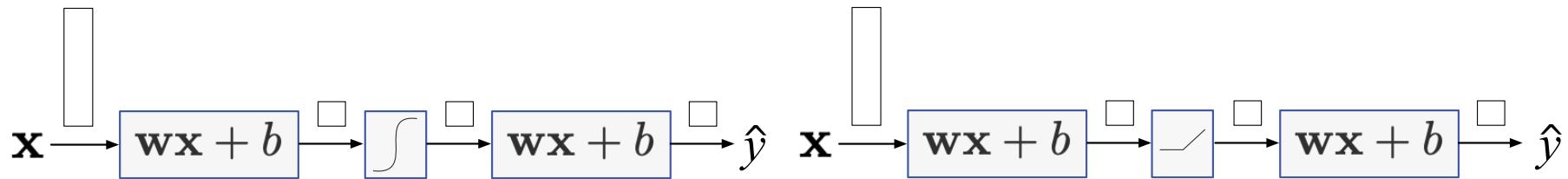
$$\hat{y} = w'(\mathbf{w}\mathbf{x}^T + b) + b' = w'\mathbf{w}\mathbf{x}^T + w'b + b' = \mathbf{w}''\mathbf{x}^T + b''$$

(w' is scalar since $\mathbf{w}\mathbf{x}^T + b$ is scalar)



Still just a line

Must introduce nonlinearity



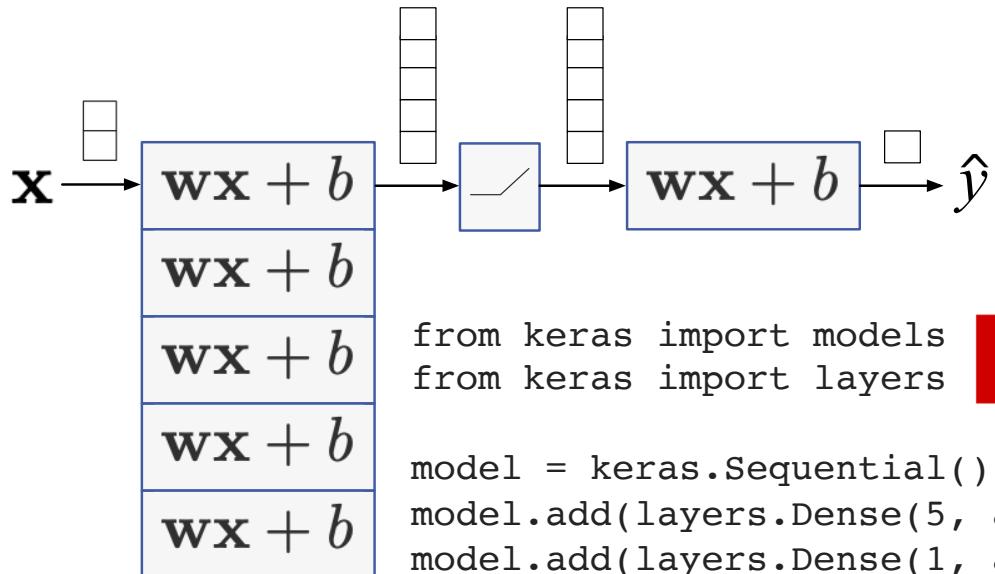
```
from keras import models  
from keras import layers  
model = keras.Sequential()
```



```
model.add(layers.Dense(1, activation='sigmoid', input_shape=(1,)))  
{model.add(layers.Dense(1, activation='relu', input_shape=(1,)))}  
model.add(layers.Dense(1, activation='linear'))
```

Stack linear models (neurons) for more power

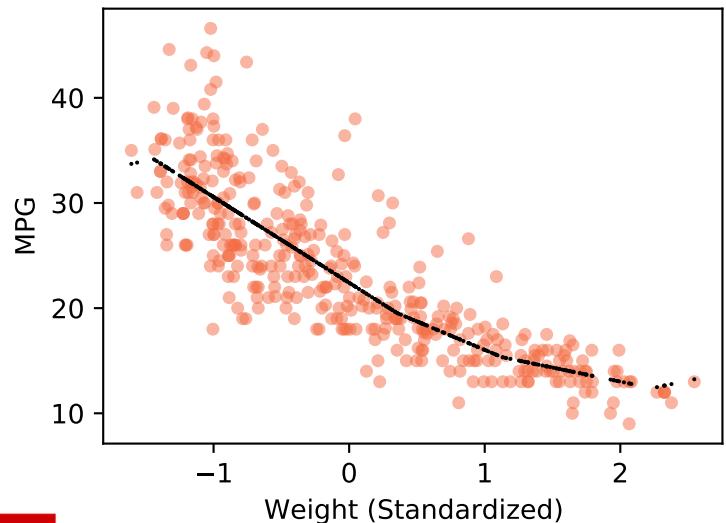
- Stack gives layer: W matrix and \mathbf{b}
- $\mathbf{g}^{[1]} = \text{relu}(W^{[1]}\mathbf{x}^T + \mathbf{b}^{[1]})$
- $\hat{y} = \mathbf{g}^{[2]} = W^{[2]}\mathbf{g}^{[1]} + \mathbf{b}^{[2]}$



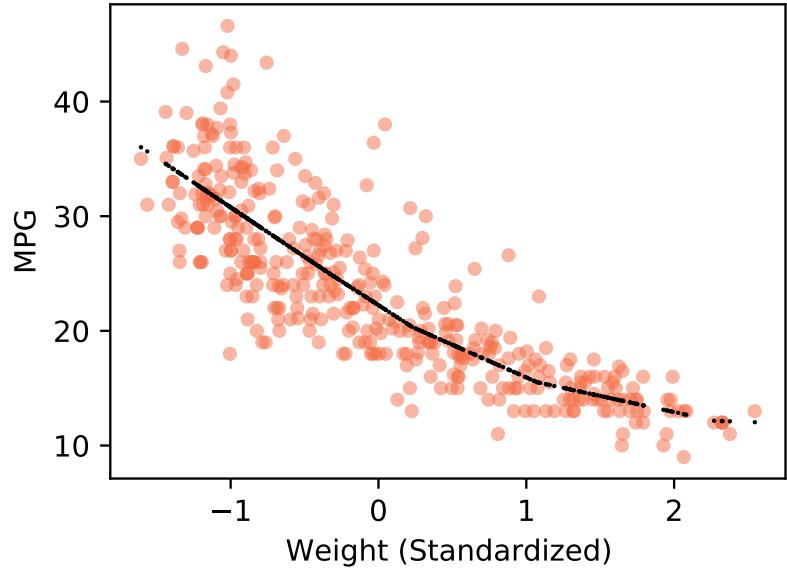
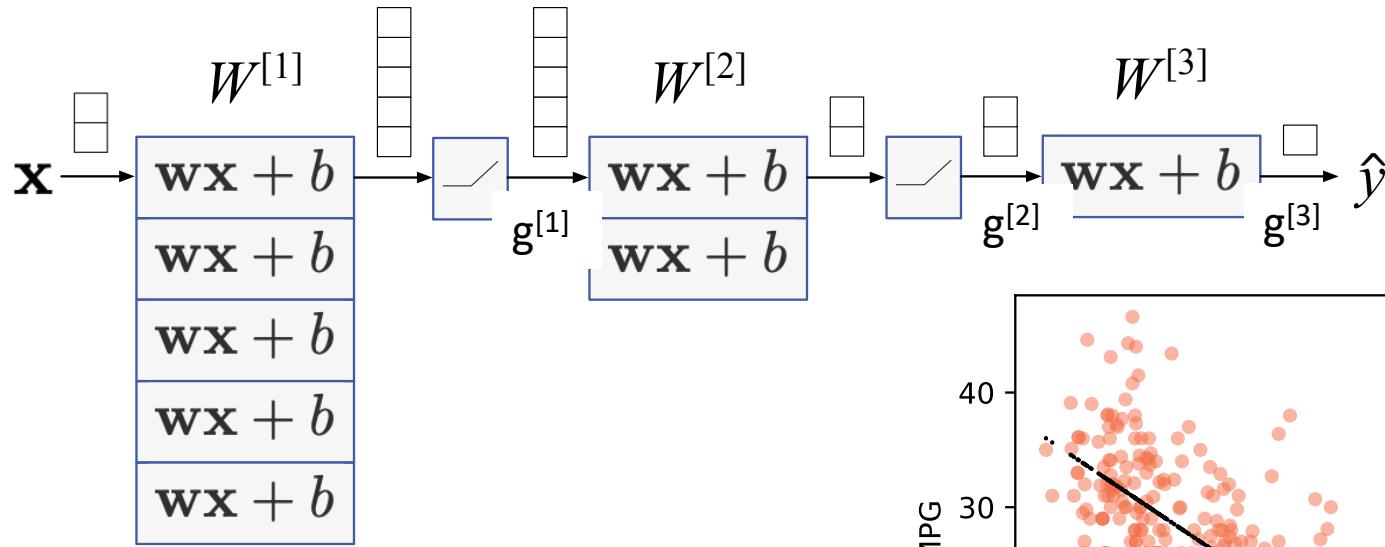
```
from keras import models  
from keras import layers
```

```
model = keras.Sequential()  
model.add(layers.Dense(5, activation='relu', input_shape=(m,)))  
model.add(layers.Dense(1, activation='linear'))
```

All these w and b are different
(how many in total?)



Math for dataset 1D: weight→MPG



```
from keras import models  
from keras import layers
```



```
model = keras.Sequential()  
model.add(layers.Dense(5, activation='relu', input_shape=(m,)))  
model.add(layers.Dense(2, activation='relu'))  
model.add(layers.Dense(1, activation='linear'))
```

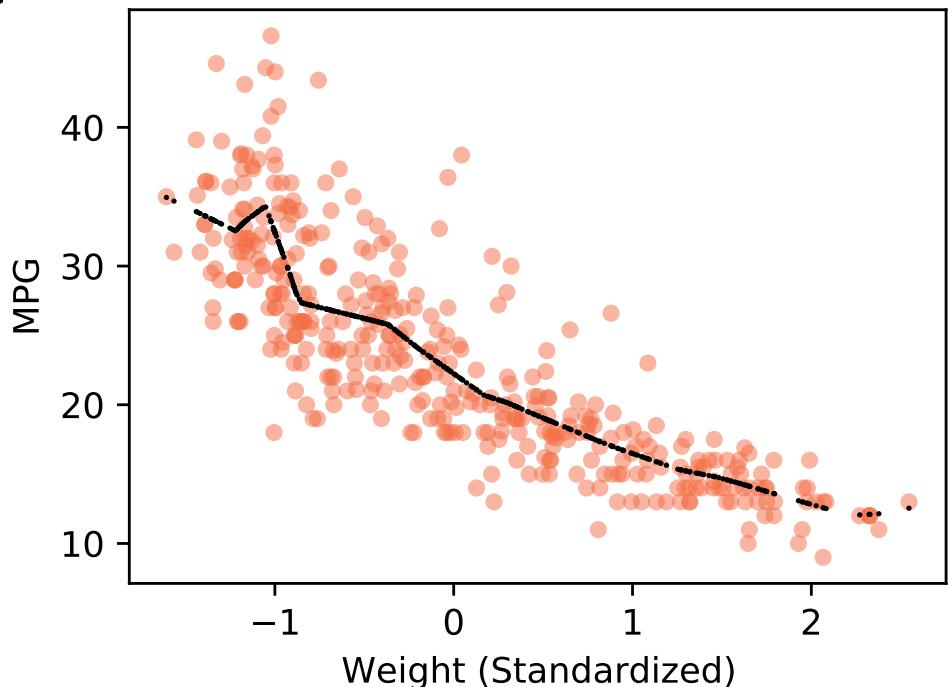
Too much capacity can overfit

- Models with too many parameters will overfit easily, if we train a long time
- Needs regularization

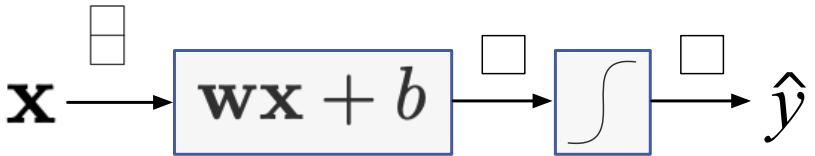
```
from keras import models  
from keras import layers
```



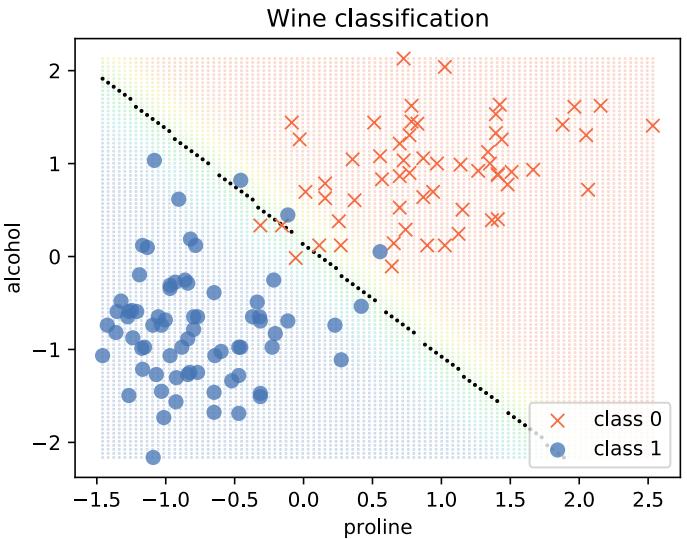
```
model = keras.Sequential()  
model.add(Dense(1000, activation='relu', input_shape=(m,)))  
model.add(layers.Dense(1, activation='linear'))
```



Binary classifiers



- Add sigmoid to regressor and we get a two-class classifier
- Prediction \hat{y} is probability of class 1
- One-layer (hidden) network with sigmoid activation function is just a logistic regression model
- Provides hyper-plane decision surfaces

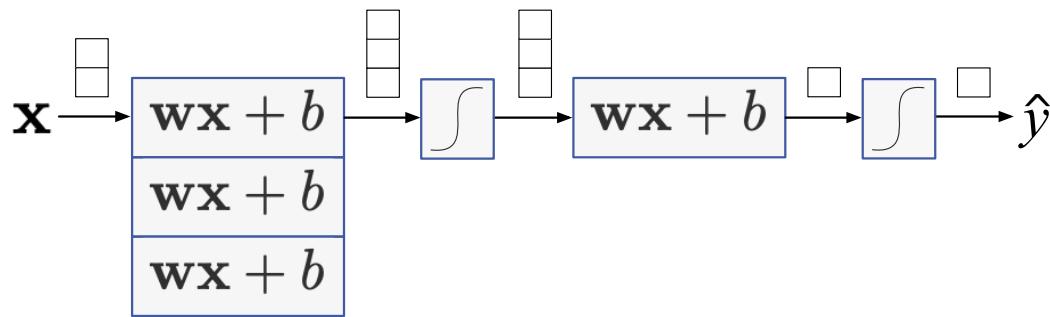


```
from keras import models  
from keras import layers  
model = keras.Sequential()  
  
model.add(layers.Dense(1, activation='sigmoid', input_shape=(m,)))
```

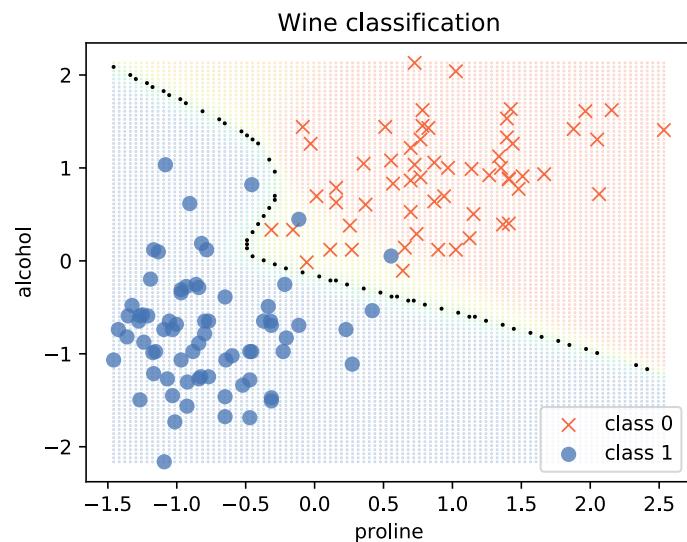
Stack neurons, add layer

See <https://github.com/parrt/fundamentals-of-deep-learning/blob/main/notebooks/5.binary-classifier-wine.ipynb>

- We get a nonlinear decision surface

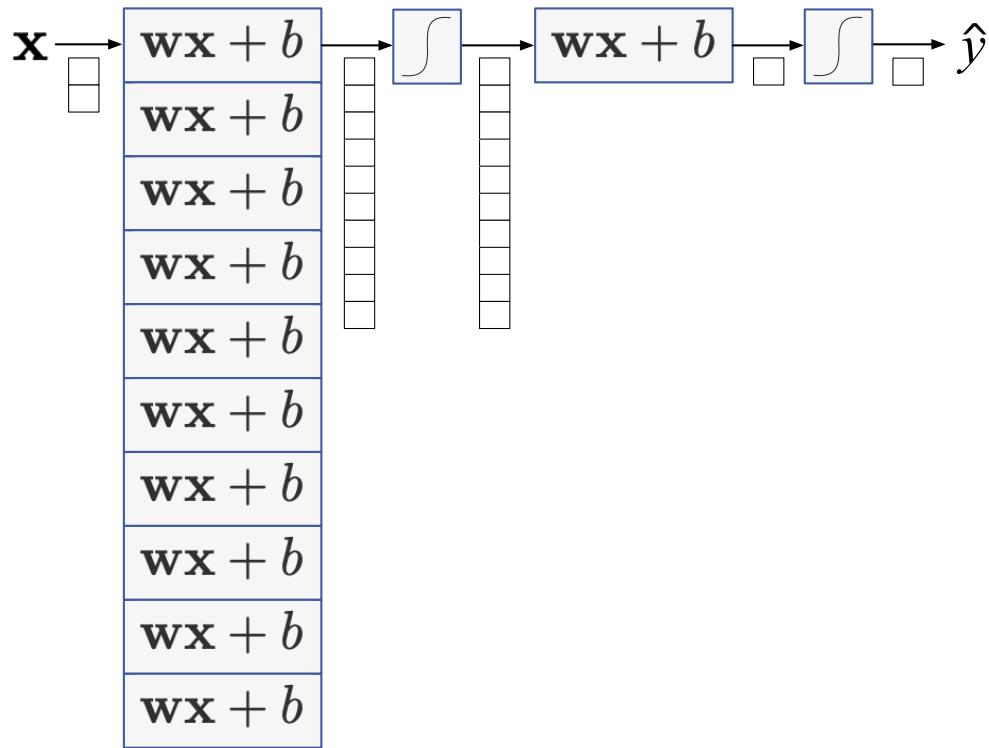


All these w and b are different



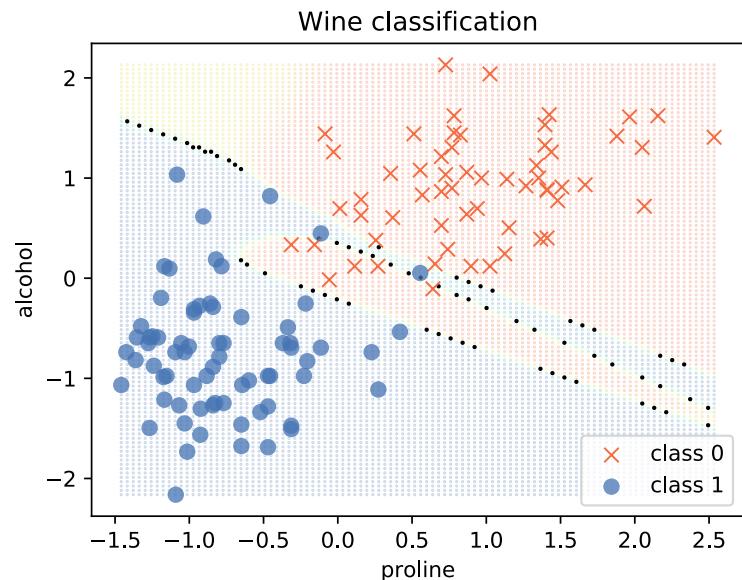
```
from keras import models  
from keras import layers  
  
model = keras.Sequential()  
model.add(layers.Dense(3, activation='sigmoid', input_shape=(1,)))  
model.add(layers.Dense(1, activation='sigmoid'))
```

More neurons: more complex decision surface



```
from keras import models  
from keras import layers
```

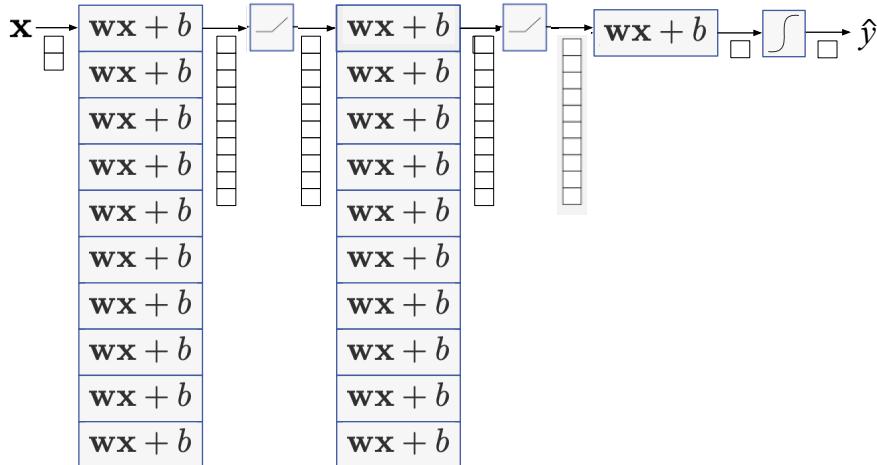
```
model = keras.Sequential()  
model.add(Dense(10, activation='sigmoid', input_shape=(m,)))  
model.add(Dense(1, activation='sigmoid'))
```



Likely overfit

Not only more complex than hyperplane but
non-contiguous regions

Even ReLUs can get "curvy" surfaces

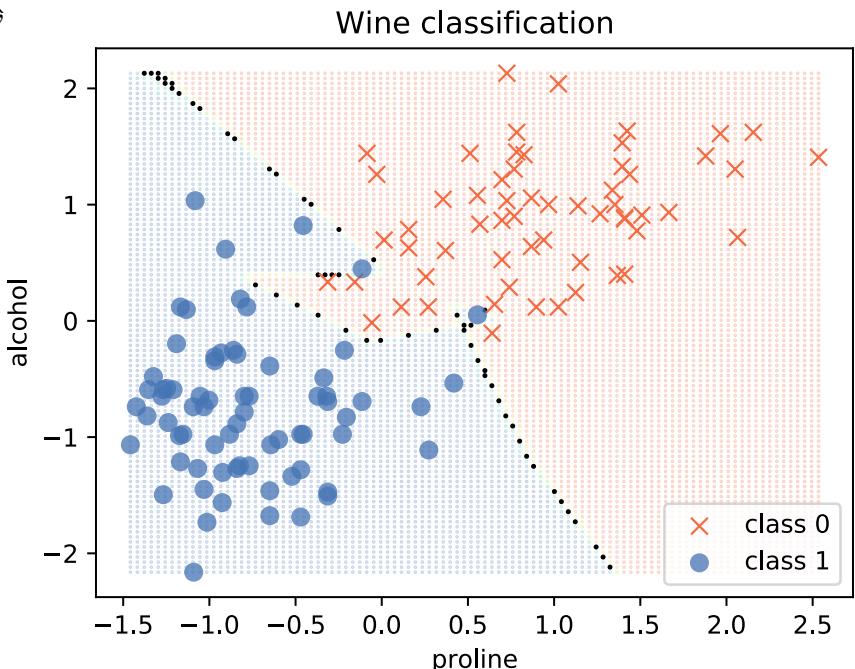


from keras import models
from keras import layers



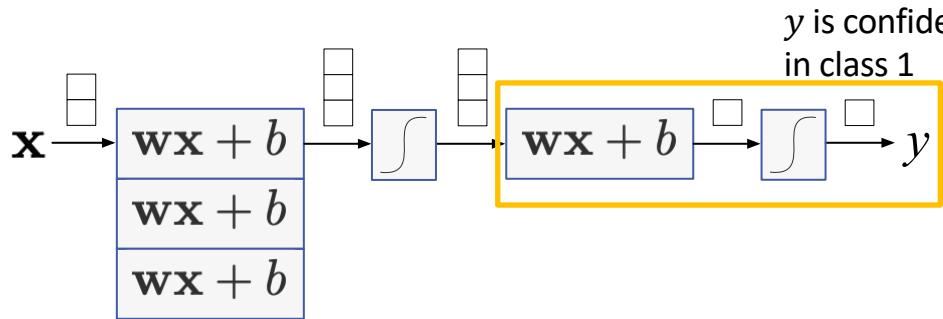
```
model = keras.Sequential()  
model.add(Dense(10, activation='relu', input_shape=(m,)))  
model.add(Dense(10, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))
```

(Last activation function still must be sigmoid)

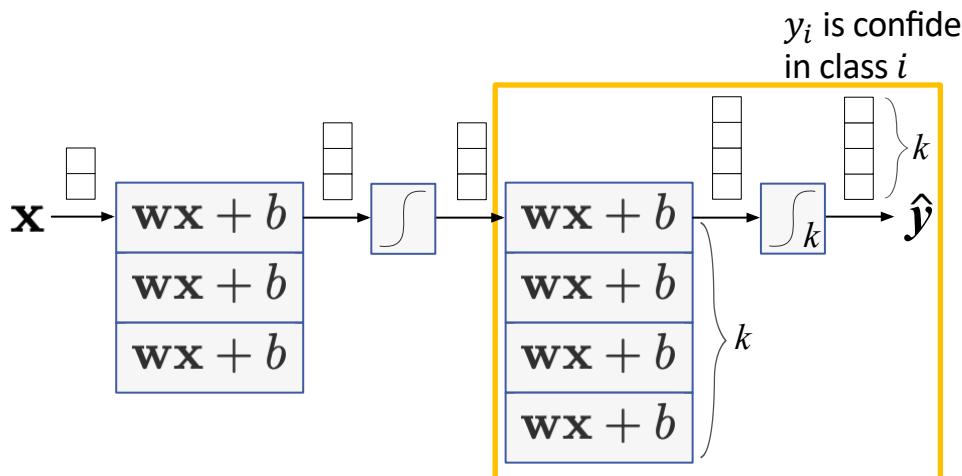


k -class classifiers

- **2-class problems:** final 1 neuron linear layer + sigmoid layer



- **k -class problems:** final k -neuron linear layer + softmax



```
from keras import models  
from keras import layers  
model = keras.Sequential()  
  
model.add(Dense(3, activation='sigmoid', input_shape=(2,)))  
{model.add(Dense(1 activation='sigmoid'))}  
model.add(Dense(4, activation='softmax'))
```

Luis G. Moyano - Fundamentos de ML
Instituto Bakseiro

(Last activation function still must be softmax)

k -class classifiers

- Instead of one neuron in last layer, we use k for k classes
- Last layer has vector output: $\mathbf{z}^{[layer]} = W^{[layer]} \mathbf{x}^T + \mathbf{b}^{[layer]}$
- Instead of sigmoid, we use softmax function
- Vector of k probabilities as activation: $\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{[layer]})$
- Normalized probabilities of k classes
- 'Soft' as in differentiable, thus nice when computing gradients

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^N e^{a_k}} \quad \forall j \in 1..N$$

$$D_j S_i = \begin{cases} S_i(1 - S_j) & i = j \\ -S_j S_i & i \neq j \end{cases}$$

Sample softmax computation

- For layer output vector \mathbf{z} :
 - Any number gets mapped to positive values
 - max gets inflated respect to others, thus, normalization acts as a max function, but 'softer' (i.e., is continuous).
- Similar to the Boltzmann distribution,
 - thus, susceptible of a T parameter, to modulate the impact of smaller values ($T \gg 1$ tends to equiprobability).

```
z = np.array([0.1, 1, 5])
np.exp(z)

array([ 1.10517092,  2.71828183, 148.4131591 ])

np.exp(z) / np.sum(np.exp(z))

array([0.00725956, 0.01785564, 0.9748848 ])
```

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}$$

Stable form:

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^N e^{a_k}} = \frac{C e^{a_j}}{\sum_{k=1}^N C e^{a_k}}$$

$$S_j = \frac{e^{a_j + D}}{\sum_{k=1}^N e^{a_k + D}}$$

$$D = -\max(a_1, a_2, \dots, a_N)$$

Usual Loss functions

- **Regression:** typically mean squared error (MSE); should have smooth derivative, though mean absolute error works despite discontinuity (it's derivative is a V shape)

```
>>> mse = tf.keras.losses.MeanSquaredError()  
>>> mse(y_true, y_pred).numpy()
```



$$loss = -\frac{1}{n} \sum_{i=1}^n y_i \log(p) + (1 - y_i) \log(1 - p_i)$$

- **Classification:** log loss (also called cross entropy)

- Penalizes very confident misclassifications strongly
- Function of actual y and estimated probabilities, not predicted class
- Perfect score is 0 log loss, imperfection gives unbounded scores

```
>>> bce = tf.keras.losses.BinaryCrossentropy(from_logits=True)  
>>> bce(y_true, y_pred).numpy()
```

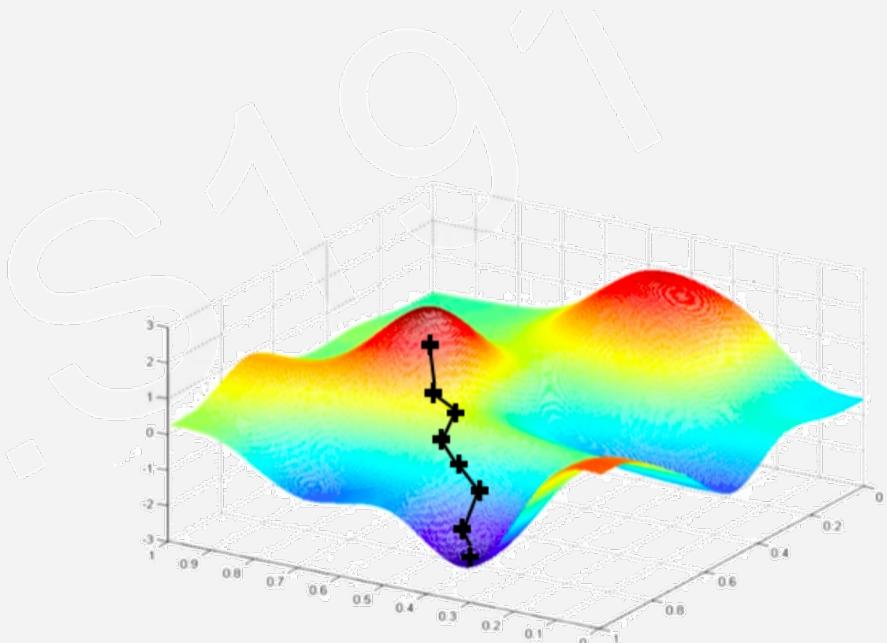


Gradient descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

Batch GD: Compute J over all examples



Mini-Batch gradient descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

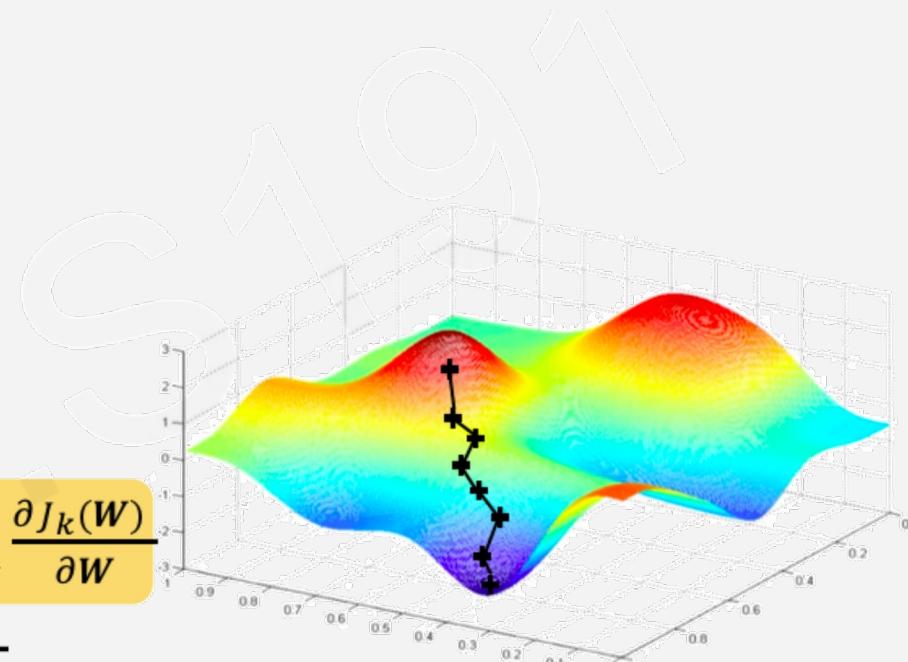
2. Loop until convergence:

3. Pick batch of B data points

4. Compute gradient,
$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$$

5. Update weights,
$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

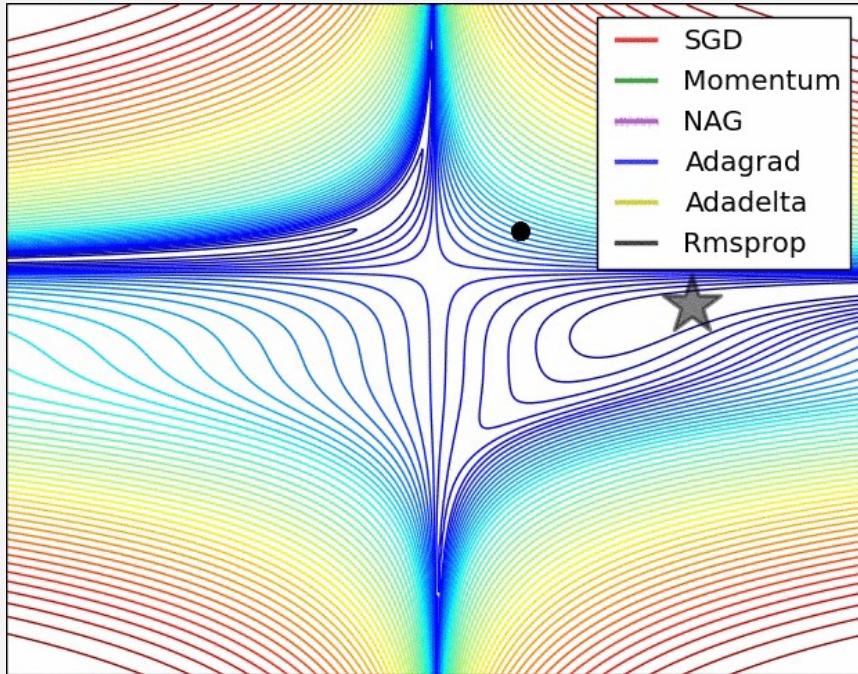
6. Return weights



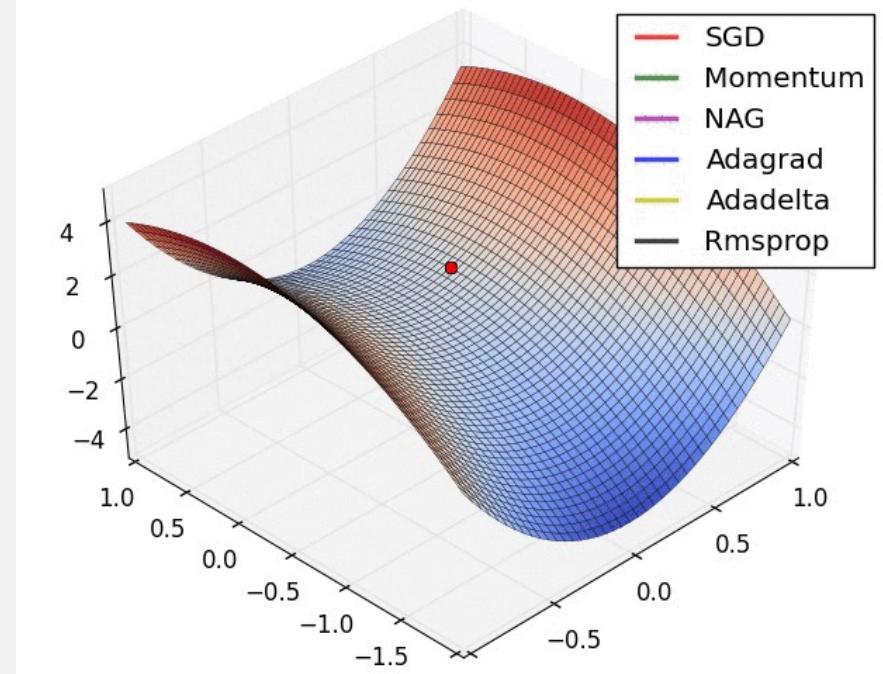
Fast to compute and a much better
estimate of the true gradient!

Stochastic - SGD: Compute J over only one example ($B=1$)
(as opposed to Batch GD, where $B=N$)

Optimization algorithms



<https://cs231n.github.io/neural-networks-3/>



- **SGD:** Stochastic Gradient descent
- **Momentum:** use gradient to change velocity, not position
- [Adadelta](#)
- [Rmsprop](#) – check also [Hinton's lecture 6](#)

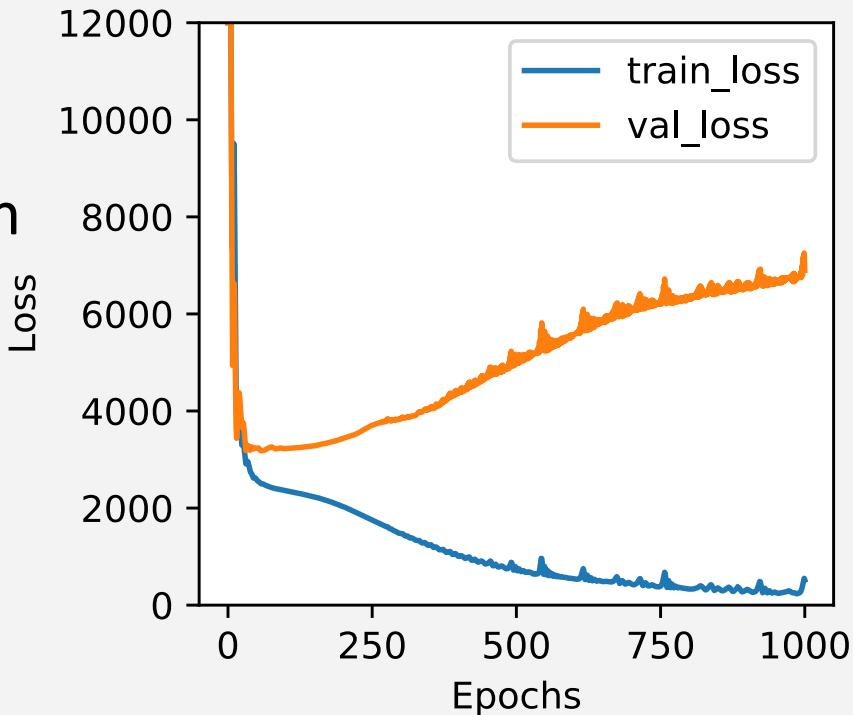
Optimizers in Keras

- SGD
- RMSprop
- Adam
- Adadelta
- Adagrad
- Adamax
- Nadam
- Ftrl

```
>>> opt = tf.keras.optimizers.SGD(learning_rate=0.1)
>>> var = tf.Variable(1.0)
>>> loss = lambda: (var ** 2)/2.0           # d(loss)/d(var1) = var1
>>> step_count = opt.minimize(loss, [var]).numpy()
>>> # Step is ` - learning_rate * grad`
>>> var.numpy()
0.9
```

Common train vs validation loss behavior

- DL networks have so many parameters, we can often get training error down to zero!
- But, we care about generalization
- Unfortunately, validation error often tracks away from training error as the number of epochs increases
- This model is clearly overfitting
- Need to use regularization to improve validation loss



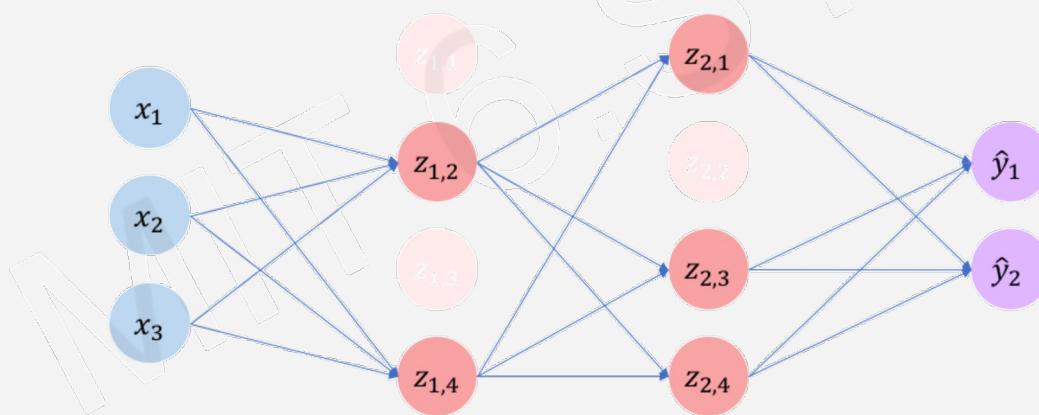
Training process

1. Prepare data
 - normalize numeric variables
 - dummy vars for categoricals
 - conjure up values for missing values
2. Split out at least a validation set from training set
3. Choose network architecture, appropriate loss function
4. Choose hyper-parameters, such as dropout rate
5. Choose a learning rate, number of epochs (passes through data)
6. Run training loop (until validation error goes up or num iterations)
7. Goto 3, 4, or 5 to tweak; iterate until good enough

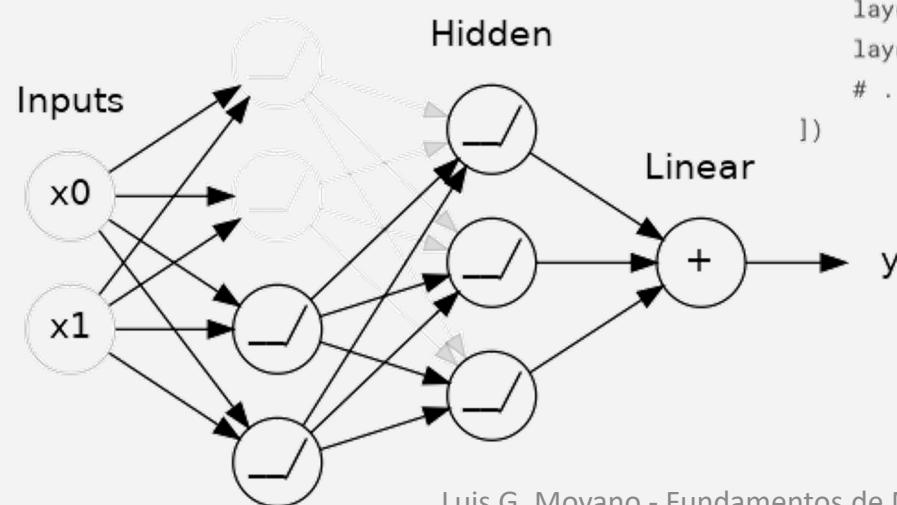
Regularization -Dropout

- During training, randomly set some activations to 0
 - Typically 'drop' 50% of activations in layer
 - Forces network to not rely on any 1 node

 `tf.keras.layers.Dropout(p=0.5)`



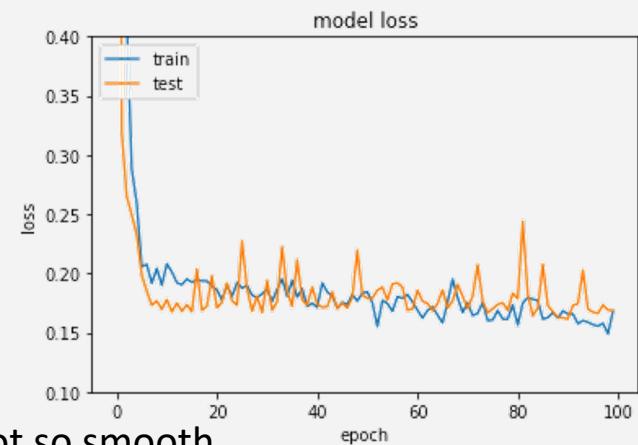
Hidden



```
keras.Sequential([
    # ...
    layers.Dropout(rate=0.3), # apply 30% dropout to the next layer
    layers.Dense(16),
    # ...
])
```

Regularization - Early stopping

- Stop training before we have a chance to overfit



- In Stochastic and Mini-batch Gradient Descent, the curves are not so smooth
- hard to know whether you have reached the minimum or not.
- Solution: stop only after the validation error has been above the minimum for some time
- Roll back the model parameters to the point where the validation error was at a minimum (callbacks)

```
from tensorflow.keras.callbacks import EarlyStopping  
early_stopping = EarlyStopping()  
model.fit(X_train, y_train, epochs=200, validation_split=0.25, batch_size=40, verbose=2, callbacks=[early_stopping])
```

Regularization techniques

- Get more training data; can try augmentation techniques
(more data is likely to represent population distribution better)
- Reduce number of model parameters (i.e., simplify it)
(reduce power/ability to fit the noise)
- Add drop out layers (randomly kill some neurons)
- Weight decay (L2 regularization on model parameters,
restrict model parameter search space)
- Early stopping, when validation error starts to go up
(generally we choose model that yields the best validation error)
- Batch normalization has some small regularization effect
(Force layer activation distributions to be 0-mean, variance 1)
- Stochastic gradient descent tends to land on better
generalizations

Summary

- Vanilla deep learning models are layers of linear regression models glued together with nonlinear functions such as sigmoid/ReLUs
- Regressor: final layer transforms previous layer to single output
- Classifier: add sigmoid to last regressor layer (2-class) or add softmax to last layer of k neurons (k -class)
- Training a model means finding optimal (or good enough) model parameters as measured by a *loss* (cost or error) function; hyper parameters describe architecture and learning rate, amount of regularization, etc.
- We train using (stochastic) gradient descent; tuning model and hyper parameters is more or less trial and error ☹ but experience helps a lot

Deep playground

Tinker With a Neural Network in Your Browser

<https://playground.tensorflow.org>

Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.

Epoch 001,088 Learning rate 0.03 Activation ReLU Regularization None Regularization rate 0 Problem type Classification

DATA
Which dataset do you want to use?

Ratio of training to test data: 50%
Noise: 0
Batch size: 10

FEATURES
Which properties do you want to feed in?
 X_1 , X_2 , X_{12} , X_{22} , X_{1X^2} , $\sin(X^1)$, $\sin(X^2)$

2 HIDDEN LAYERS
+ -
4 neurons 2 neurons
The outputs are mixed with varying weights, shown by the thickness of the lines.
This is the output from one neuron. Hover to see it larger.

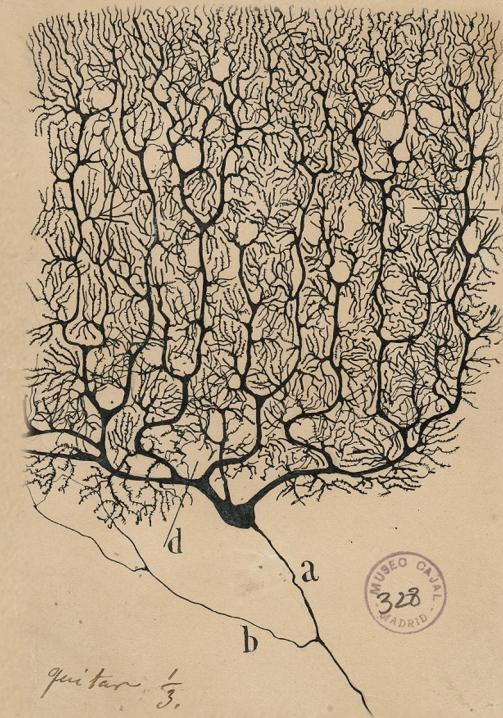
OUTPUT
Test loss 0.434
Training loss 0.411

REGENERATE

Colors shows data, neuron and weight values.
 Show test data Discretize output

ML Fundamentals – Lecture 9

- Perceptron
- Feedforward nets
- Tensors & Keras 
- Architectures
- Training
 - Optimizers
 - Regularization



Next:
Deep learning II