Why KERAS?

We'll be using Keras because

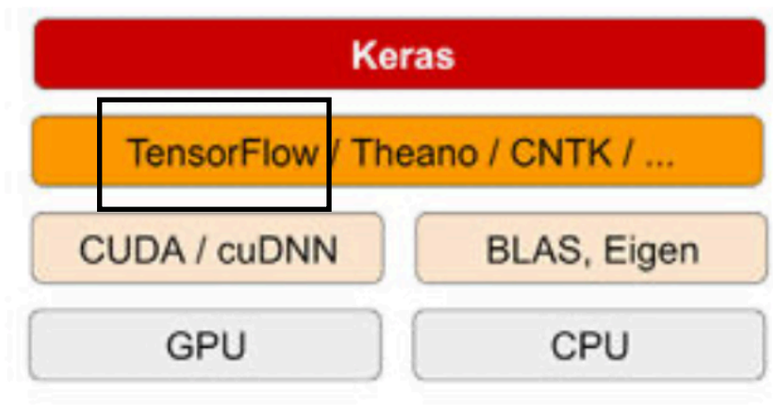
- Can be used with several frameworks
- Simplifies design and implementation of networks
- Good documentation
- Easy to learn and build models

An important reason to use Keras is that while backends change over time, Keras is a wrapper and is unlikely to be supplanted given it's ease of use

That said, it does have some drawbacks for building more customized models but at that point it is probably a good time to switch over to learning a DL backend



Figure 7: *Keras was conceived by Francis Chollet. In early 2018 it was integrated with Tensforflow*



2020-Chollet: Deep learning with Python

-model level development
-high level building block
-doesn't do tensor manipulation/Diff

→ **seamlessly switch between backend and low level linear-algebra library**

Typical workflow

-define data (training-validation data, input/target data)
-define network of layers (model)
-configure learning process (loss+optimizer)
-iterate (fit) (sequential class: linear stack of layer)
(functional acyclic graphs API: arbitrary architecture)

CPU VS GPU

GPUs rather than CPUs are used in DL

CPUs are fast calculators:

- They process information serially
- Benefit from multi-core distribution of computation
- Speedup of non-serial operations will be ~linear however are limited by the number of cores

Deep learning benefits from being able to parallelize operations, in particular matrix, batching and gradient calculations

CPUs while fast are limited by the number of cores/threads that distribute matrix operations

In contrast to CPUs, GPUs have thousands of cores (that are much slower) but can distribute and collect data more efficiently

- GPU bandwidth is quicker than CPU
- PCIe communication is important for sending data to the CPU and writing to disk (checkpointing)

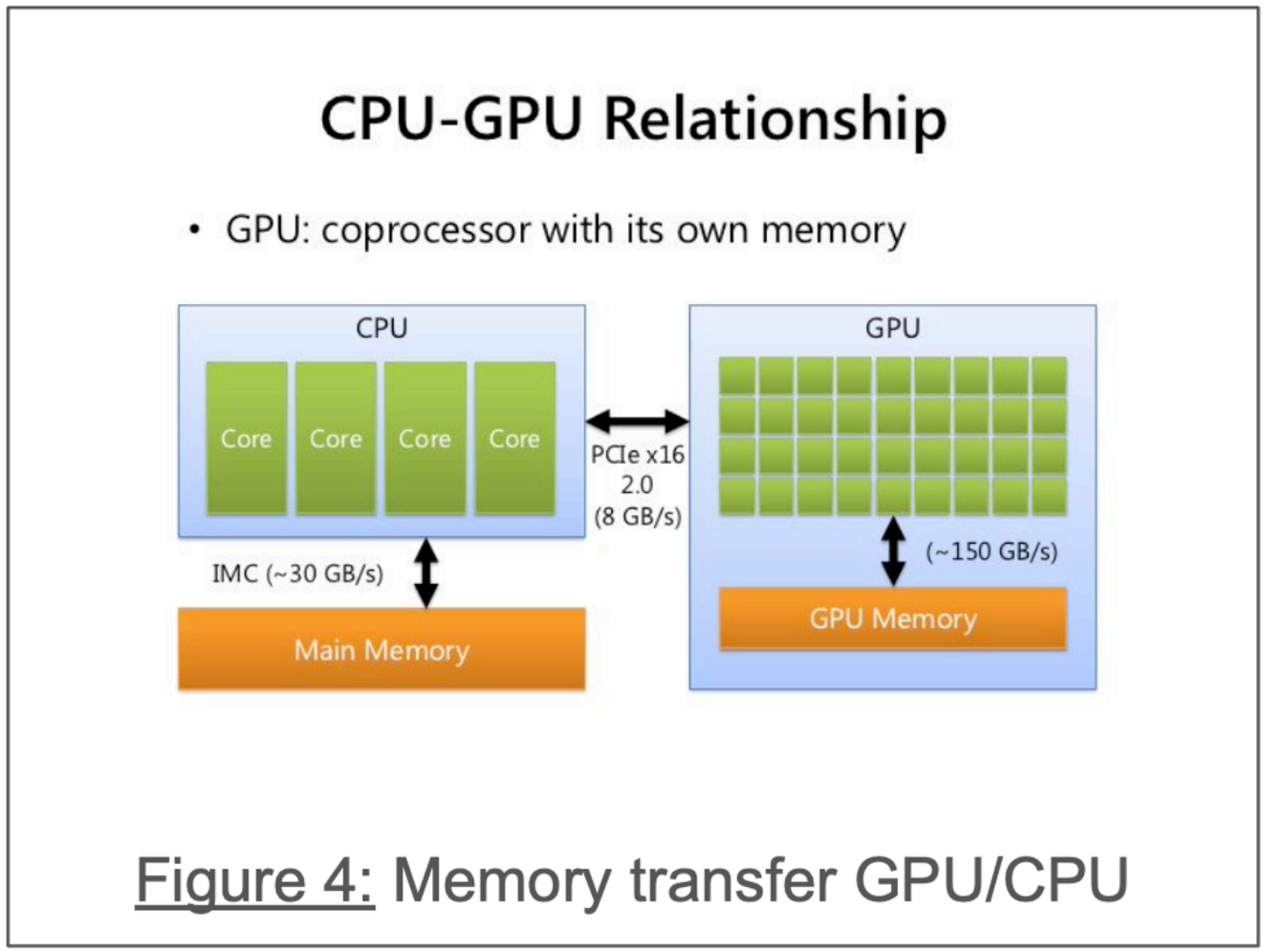
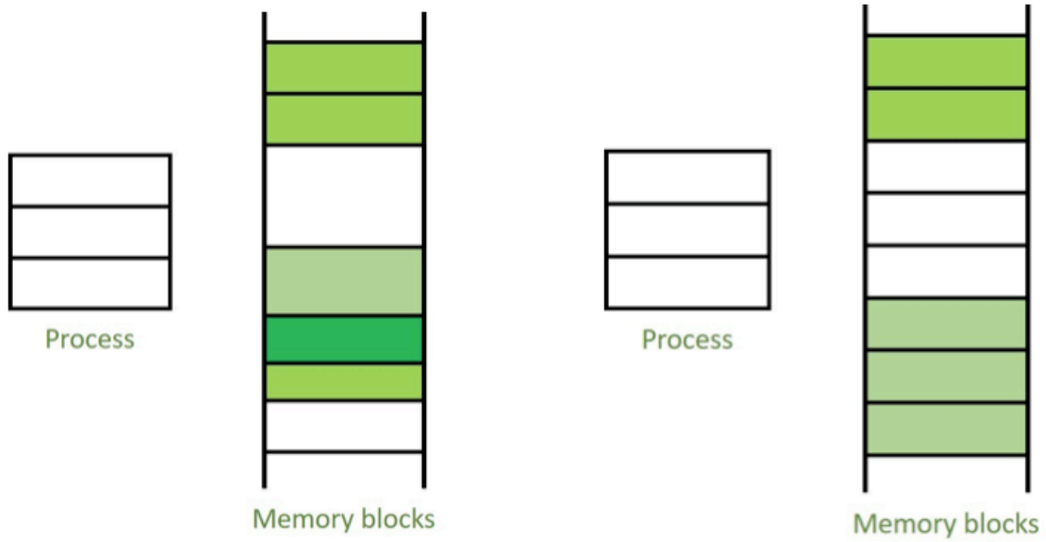
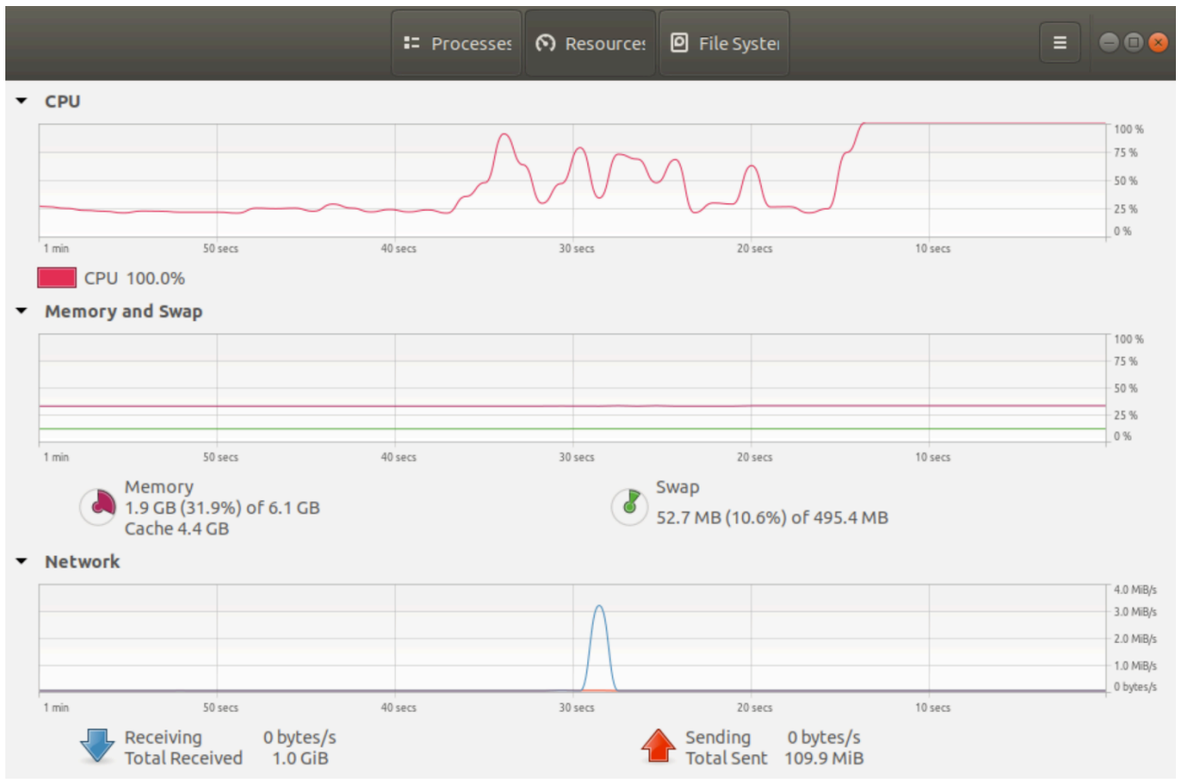


Figure 4: Memory transfer GPU/CPU

Code Demo:

```
IS CONTIGUOUS: True
IS CONTIGUOUS: True
NUMPY took: 0.0160 s
LOOPS took: 10.2502 s
Numpy is  639.0652555221928  X FASTER
```






```python
#CODE MODIFIED FROM:
# chollet-deep-learning-in-python

import numpy as np
import time


# #COMPARE TIMES FOR NUMPY (VECTORIZED) VS LOOPS
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x


# #MAKE BIGGISH ARRAYS
x = np.random.random((100, 1000))
y = np.random.random((100, 1000))

# #NOTICE THAT THEY ARE CONTIGOUS IN MEMOERY
print("IS CONTIGUOUS:",x.data.contiguous)
print("IS CONTIGUOUS:",y.data.contiguous)

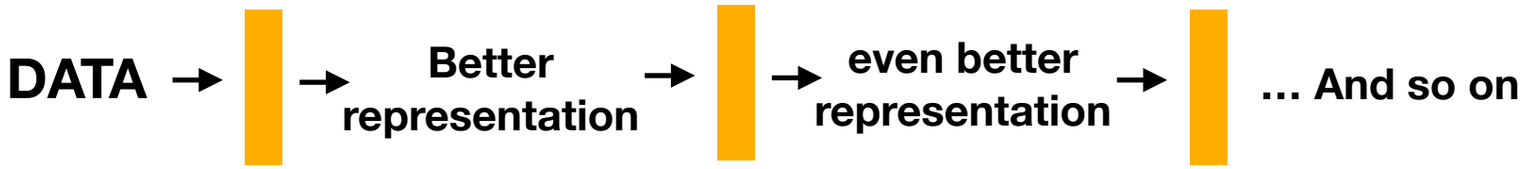

#ADD THEM 1000 TIMES WITH NUMPY (VECTORIZED)
num_add=1000
t0 = time.time()
for _ in range(num_add):
    z = x + y
time_numpy=time.time() - t0
print("NUMPY took: {0:.4f} s".format(time_numpy))

time.sleep(5)

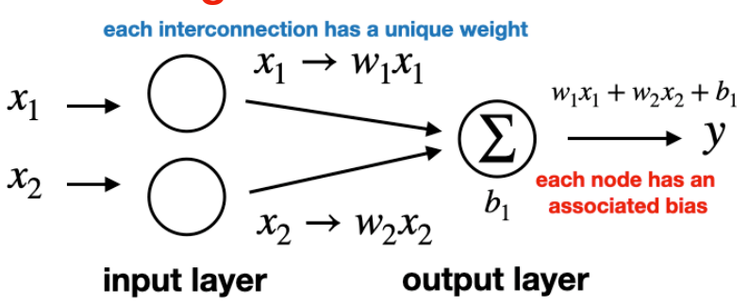
#ADD THEM 1000 TIMES WITH LOOPS
t0 = time.time()
for _ in range(num_add):
    z = naive_add(x, y)
time_loop=time.time() - t0
print("LOOPS took: {0:.4f} s".format(time_loop))
print("Numpy is ",time_loop/time_numpy," X FASTER")
```

KERAS SUMMARY

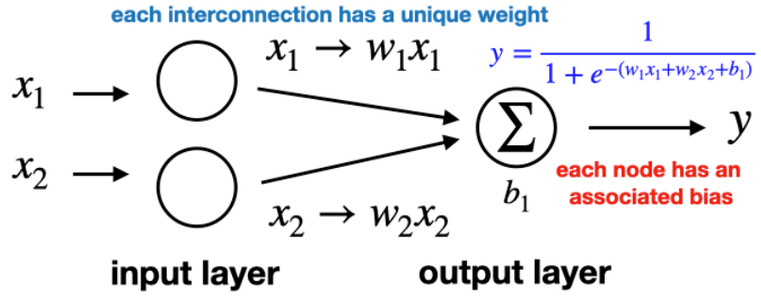
KERAS models (Deep learning models generally have a layered structure)



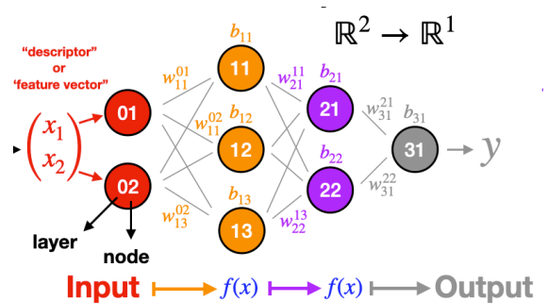
Linear regression



Logistic regression



Neural network



The Model class

- `Model` class
- `summary` method
- `get_layer` method

Keras → `tf.keras.Model()`

`Model` groups layers into an object with training and inference features.

Arguments

- **inputs**: The input(s) of the model: a `keras.Input` object or list of `keras.Input` objects.
- **outputs**: The output(s) of the model. See Functional API example below.
- **name**: String, the name of the model.

https://keras.io/api/models/model/

`summary` method

```
Model.summary(line_length=None, positions=None, print_fn=None)
```

Prints a string summary of the network.

`get_layer` method

```
Model.get_layer(name=None, index=None)
```

Retrieves a layer based on either its name (unique) or index.

Modeling in Keras

- *Layers*, which are combined into a *network* (or *model*)
- The *input data* and corresponding *targets*
- The *loss function*, which defines the feedback signal used for learning
- The *optimizer*, which determines how learning proceeds

1. Define your training data: input tensors and target tensors.
2. Define a network of layers (or *model*) that maps your inputs to your targets.

`Sequential class`

(only for linear stacks of layers)

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```



functional API
can build completely arbitrary architectures

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

applying layers to this tensor as if they were functions

3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
4. Iterate on your training data by calling the `fit()` method of your model.

KERAS SUMMARY

Keras API reference

Keras

Models API

Layers API

Callbacks API

Data preprocessing

Optimizers

Metrics

Losses

Built-in small datasets

Keras Applications

Utilities

Keras Tuner

The Model class

- Model class
- summary method
- get_layer method

The Sequential class

- Sequential class
- add method
- pop method

Model training APIs

- compile method
- fit method
- evaluate method
- predict method
- train_on_batch method
- test_on_batch method
- predict_on_batch method
- run_eagerly property

Keras layers API

- Creating custom layers
- Layers API overview
  - The base Layer class
  - Layer activations
  - Layer weight initializers
  - Layer weight regularizers
  - Layer weight constraints
  - Core layers
  - Convolution layers
  - Pooling layers
  - Recurrent layers
  - Preprocessing layers
  - Normalization layers
  - Regularization layers
  - Attention layers
  - Reshaping layers
  - Merging layers
  - Locally-connected layers
  - Activation layers

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

Regression losses

- MeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- mean_squared_error function
- mean_absolute_error function
- mean_absolute_percentage_error function
- mean_squared_logarithmic_error function
- cosine_similarity function
- Huber class
- huber function
- LogCosh class
- log_cosh function

Probabilistic losses

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- Poisson class
- binary_crossentropy function
- categorical_crossentropy function
- sparse_categorical_crossentropy function
- poisson function
- KLDivergence class
- kl_divergence function

Model saving & serialization APIs

- save method
- save_model function
- load_model function
- get_weights method
- set_weights method
- save_weights method
- load_weights method
- get_config method
- from_config method
- model_from_config function
- to_json method
- model_from_json function
- clone_model function

https://keras.io/api/models/

Sequentially building a model

`Sequential` class

https://keras.io/api/models/sequential/

```
tf.keras.Sequential(layers=None, name=None)
```

`Sequential` groups a linear stack of layers into a `tf.keras.Model`.



`add` method

```
Sequential.add(layer)
```

Adds a layer instance on top of the layer stack.

`pop` method

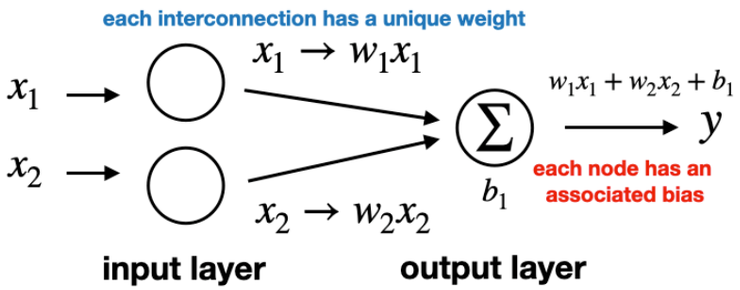
Neural network

```
Sequential.pop()
```

Removes the last layer in the model.

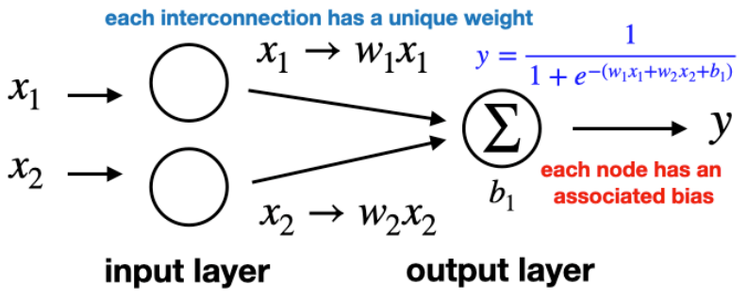
Examples:

Linear regression



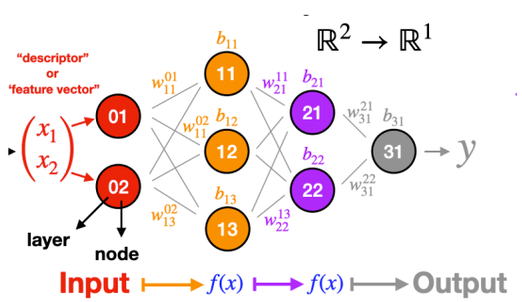

```
model = keras.Sequential([
layers.Dense(1, activation='linear', input_shape=(1,)),
```

Logistic regression




```
#LOGISTIC REGRESSION MODEL
model = keras.Sequential([
    layers.Dense(1, activation='sigmoid', input_shape=(10000,)),
])
```

Neural network




```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

Model training

https://keras.io/api/models/model_training_apis/

Compiling the model → Configures the model for training

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=None,
    steps_per_execution=None,
    **kwargs
)
```

-This is done via the KERAS “compile” method which acts on a model object

Example

```
model.compile(optimizer=tf.keras.optimizer.Adam(learning_rate=1e-3),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.BinaryAccuracy(),
                       tf.keras.metrics.FalseNegatives()])
```

Fitting the model → Trains the model for a fixed number of epochs (iterations on a dataset)

```
Model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose="auto",
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```

-This is done via the KERAS “fit” method which acts on a model object

Returns

A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **RuntimeError**: 1. If the model was never compiled or, 2. If `model.fit` is wrapped in `tf.function`.
- **ValueError**: In case of mismatch between the provided input data and what the model expects or when the input data is empty.

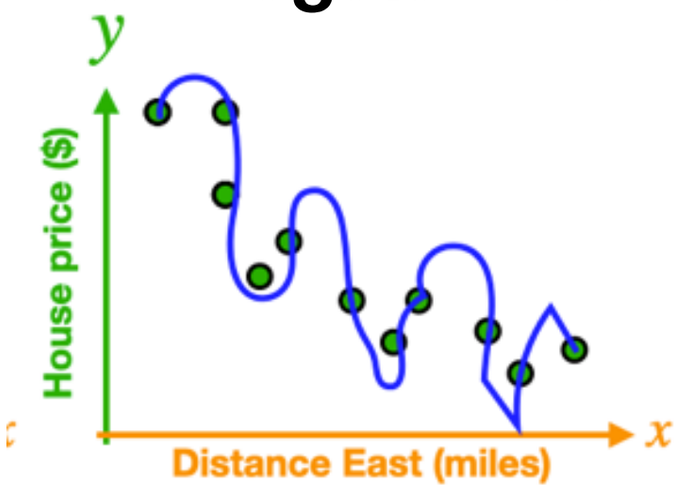
These method introduce various “hyper-parameters” (More on this next week)

REGULARIZATION

Overview

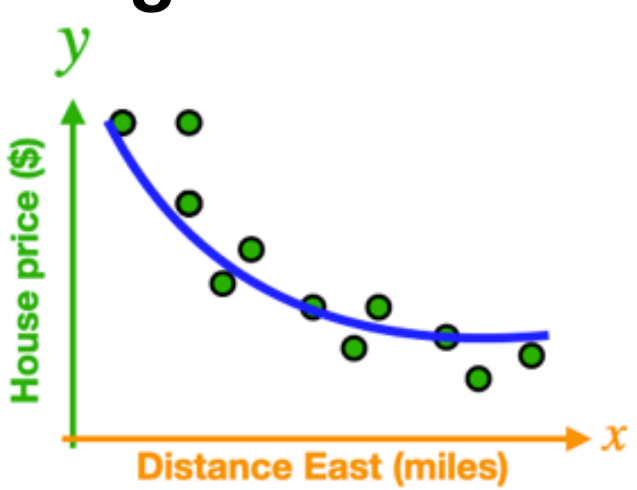
Why?

un-regularized

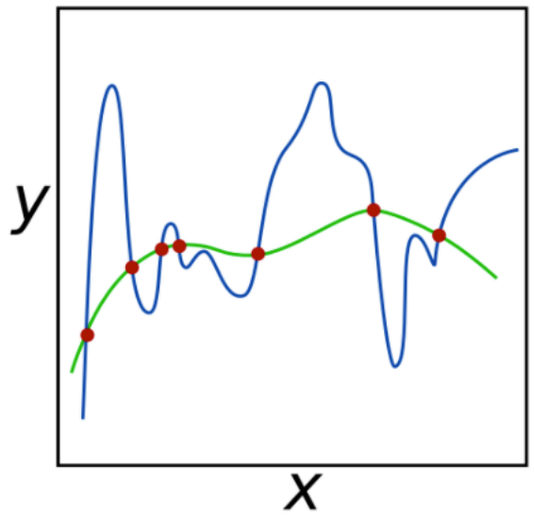


over-fitting

regularized



optimal



Cost = Error + Regularization terms



Regression losses

- MeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- mean_squared_error function
- mean_absolute_error function
- mean_absolute_percentage_error function
- mean_squared_logarithmic_error function
- cosine_similarity function
- Huber class
- huber function
- LogCosh class
- log_cosh function

Probabilistic losses

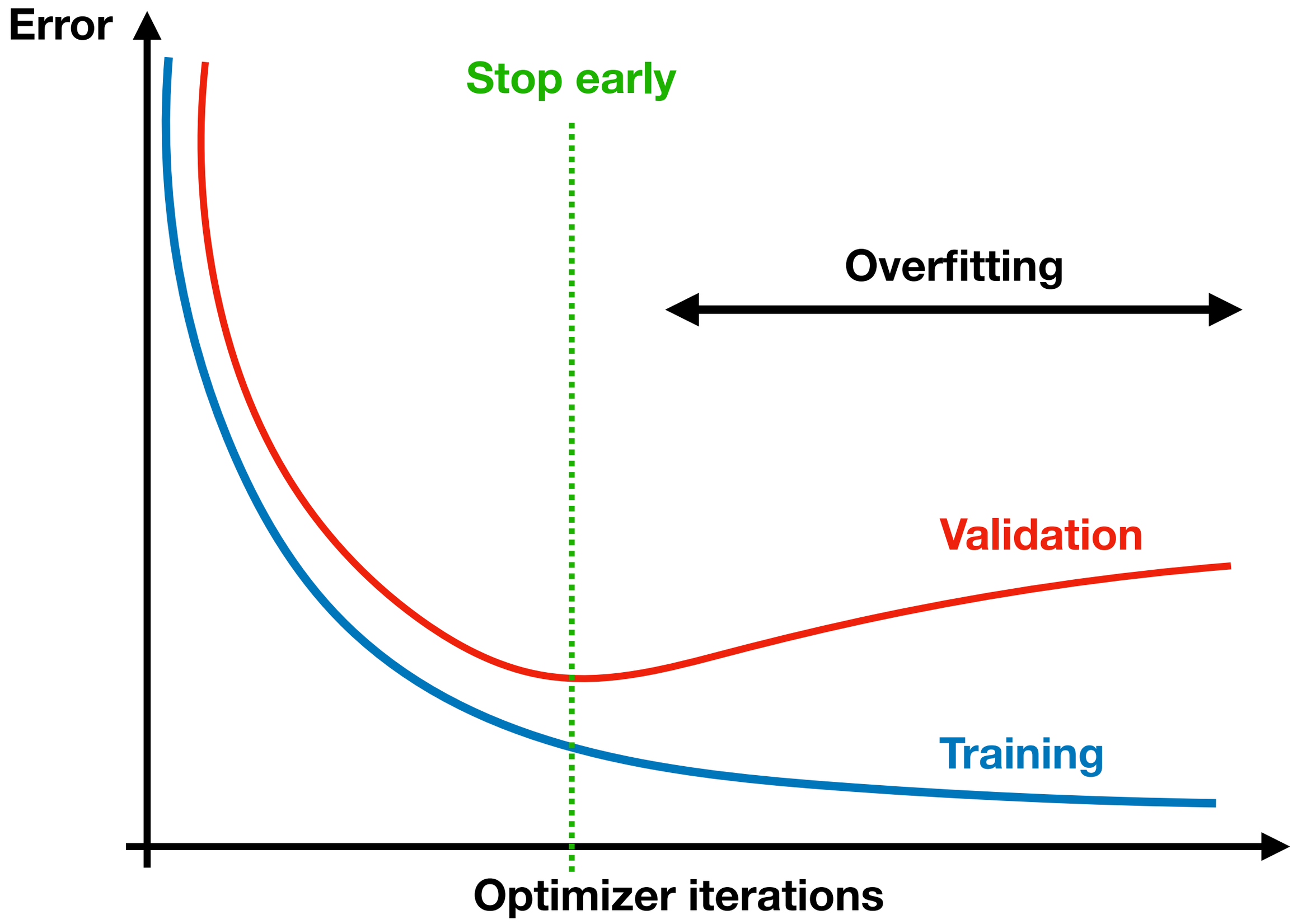
- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- Poisson class
- binary_crossentropy function
- categorical_crossentropy function
- sparse_categorical_crossentropy function
- poisson function
- KLDivergence class
- kl_divergence function

Hinge losses for "maximum-margin" classification

- Hinge class
- SquaredHinge class
- CategoricalHinge class
- hinge function
- squared_hinge function
- categorical_hinge function

- **Most common**
  - L1 (lasso regression)
  - L2 (ridge regression)
- **Other methods**
  - Early stoping
  - Data Augmentation
  - Dropout
  - … more on this next week

Early stopping



L1 and L2 regularization

A regression model that uses L1 regularization technique is called ***Lasso Regression*** and model which uses L2 is called ***Ridge Regression***.

Cost function

β = **model fitting parameters**

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \qquad L1$$

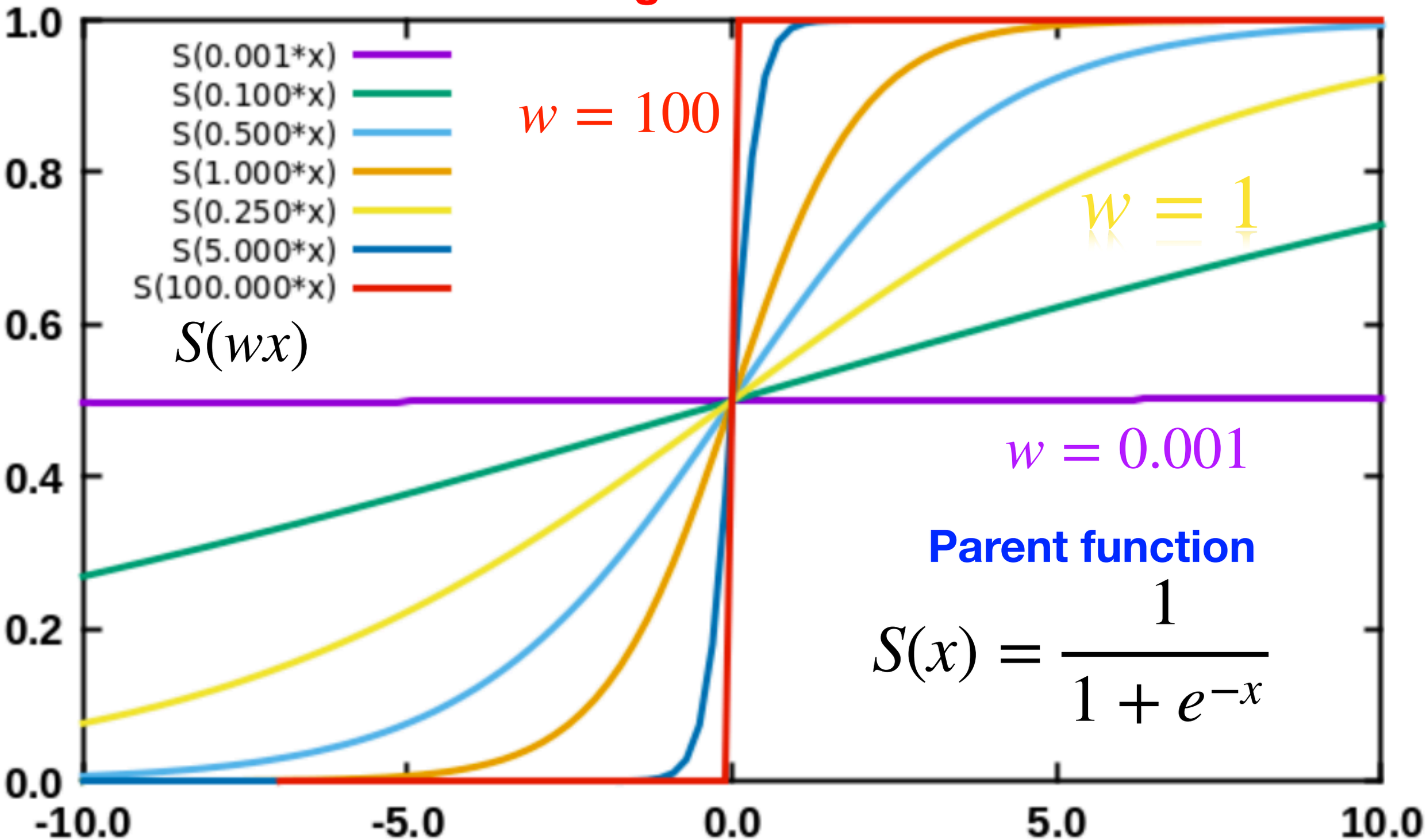
$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \qquad L2$$

Hyper parameter that controls “strength” of regularization

https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c

Why does it work?

non-rigid transform



Parent function

$$S(x) = \frac{1}{1 + e^{-x}}$$

```
gnuplot> set key left; S(x)=1/(1+exp(-x)); plot S(0.001*x) lw 3, S(0.100*x) lw 3, S(0.500*x) lw 3,S(1.000*x) lw 3,S(0.250*x) lw 3,S(5.000*x) lw 3, S(100.000*x) lw 3
```

SMALL —> smooth LARGE —> sharp

Keras regularizers

Layer weight regularizers

- ▹ Available regularizers
  - `L1` class
  - `L2` class
  - `l1_l2` function
- ▹ Creating custom regularizers
  - Simple callables
  - `Regularizer` subclasses

Keras

```
from tensorflow.keras import layers
from tensorflow.keras import regularizers

layer = layers.Dense(
    units=64,
    kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
    bias_regularizer=regularizers.l2(1e-4),
    activity_regularizer=regularizers.l2(1e-5)
)
```

Regularization penalties are applied on a per-layer basis.

Create a regularizer that applies both L1 and L2 penalties.

The L1 regularization penalty is computed as: `loss = l1 * reduce_sum(abs(x))`

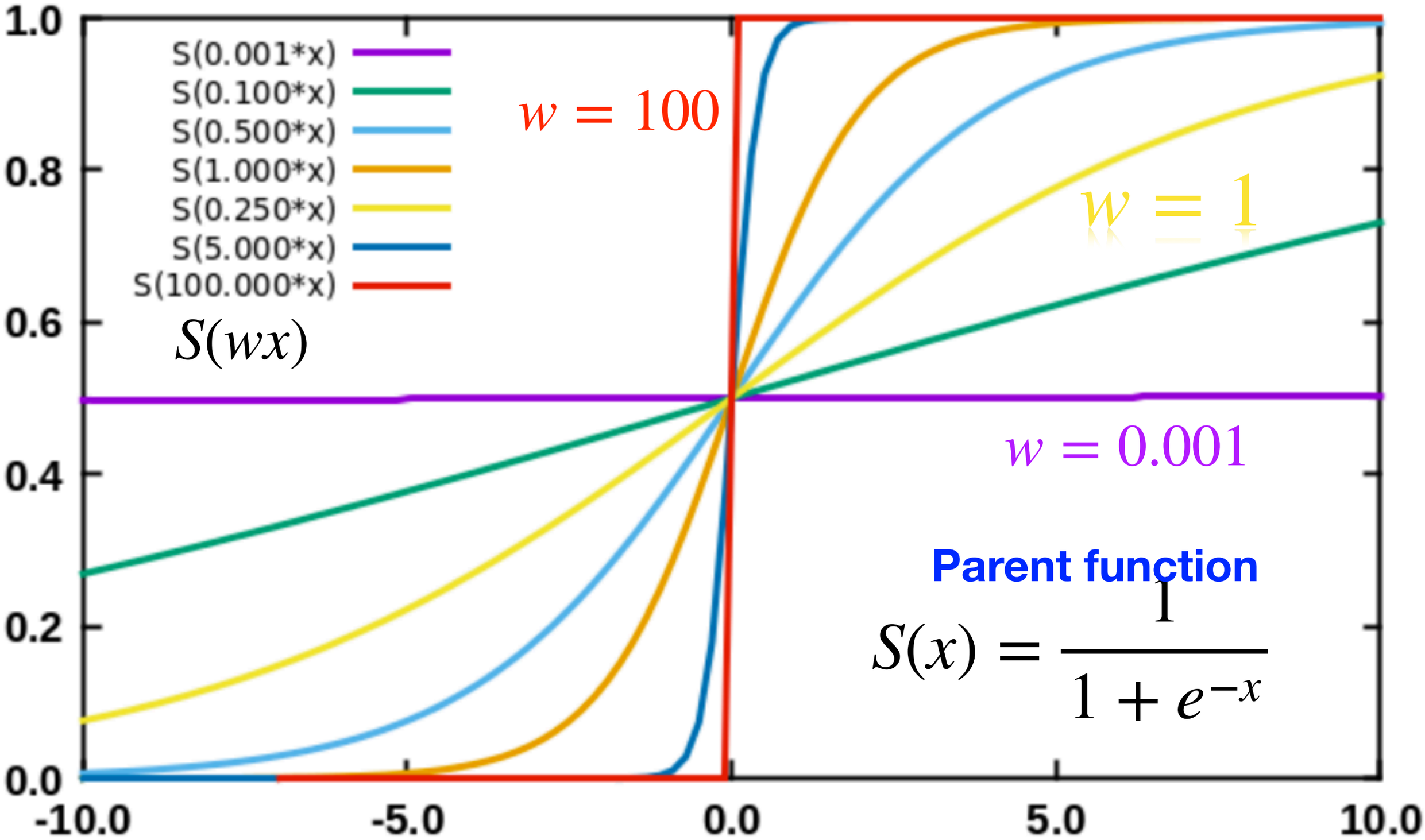
The L2 regularization penalty is computed as: `loss = l2 * reduce_sum(square(x))`

Creating custom regularizers

```
def my_regularizer(x):
    return 1e-3 * tf.reduce_sum(tf.square(x))
```

https://keras.io/api/layers/regularizers/#l1_l2-function

Basic idea (L1 and L2)



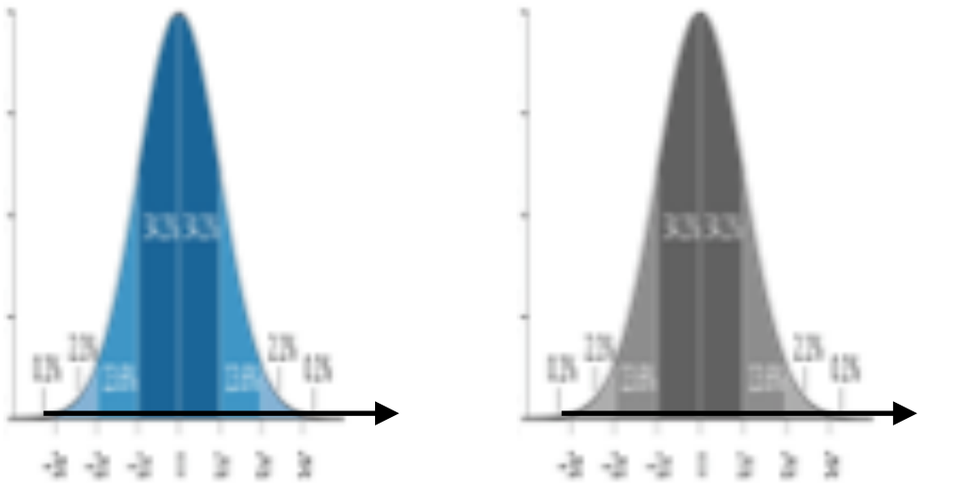
gnuplot> set key left; S(x)=1/(1+exp(-x)); plot S(0.001*x) lw 3, S(0.100*x) lw 3, S(0.500*x) lw 3,S(1.000*x) lw 3,S(0.250*x) lw 3,S(5.000*x) lw 3, S(100.000*x) lw 3

SMALL WEIGHTS—> smooth ANN (GOOD)

LARGE WEIGHTS—> sharp jagged ANN (BAD)

Basic idea (L1 and L2)

No regularization



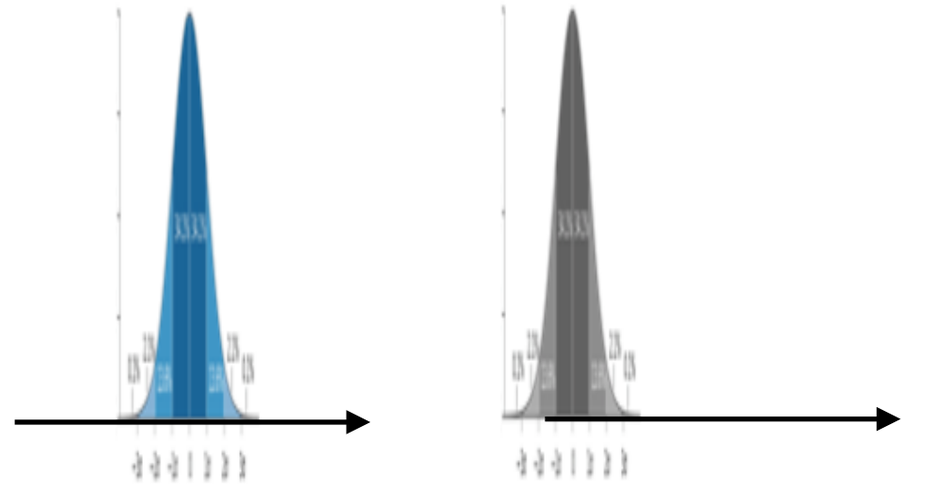
fitting parameter-1 distribution

fitting parameter-2 distribution

more variablity → more complexity

With Regularization

Squared L2 norm: $\|\mathbf{w}\|_2^2 = \sum_i^n w_i^2$ L1 norm: $\|\mathbf{w}\|_1 = \sum_i^n |w_i|$

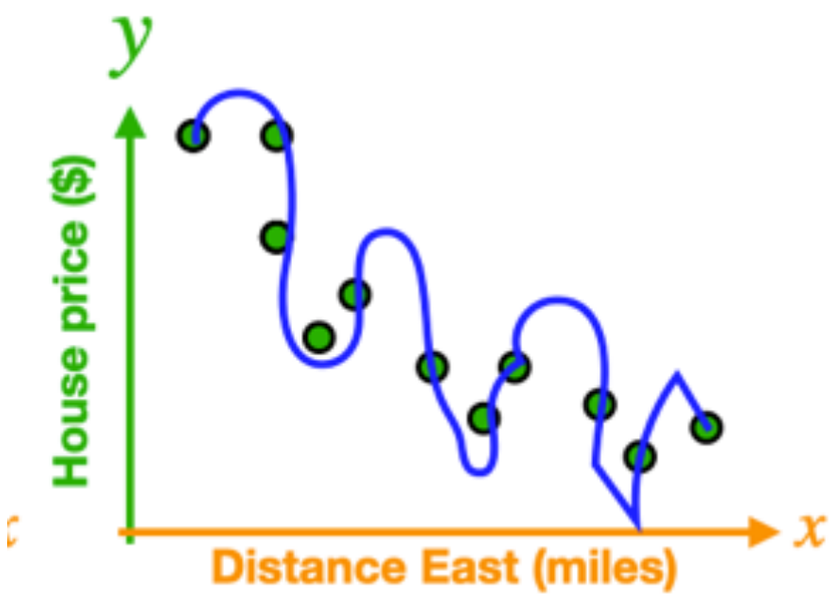


fitting parameter-1 distribution

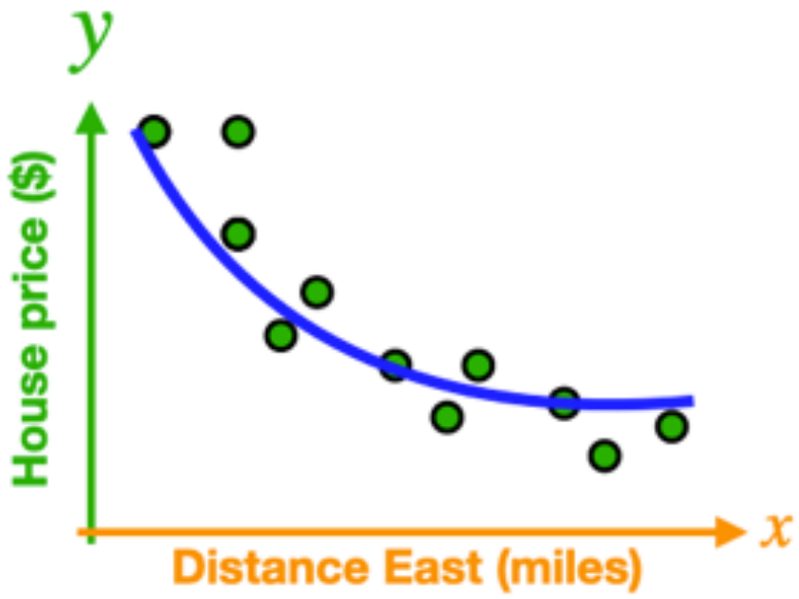
fitting parameter-2 distribution

less variablity → less complexity

Simpler models are less likely to over- fit than complex ones.



over-fitting

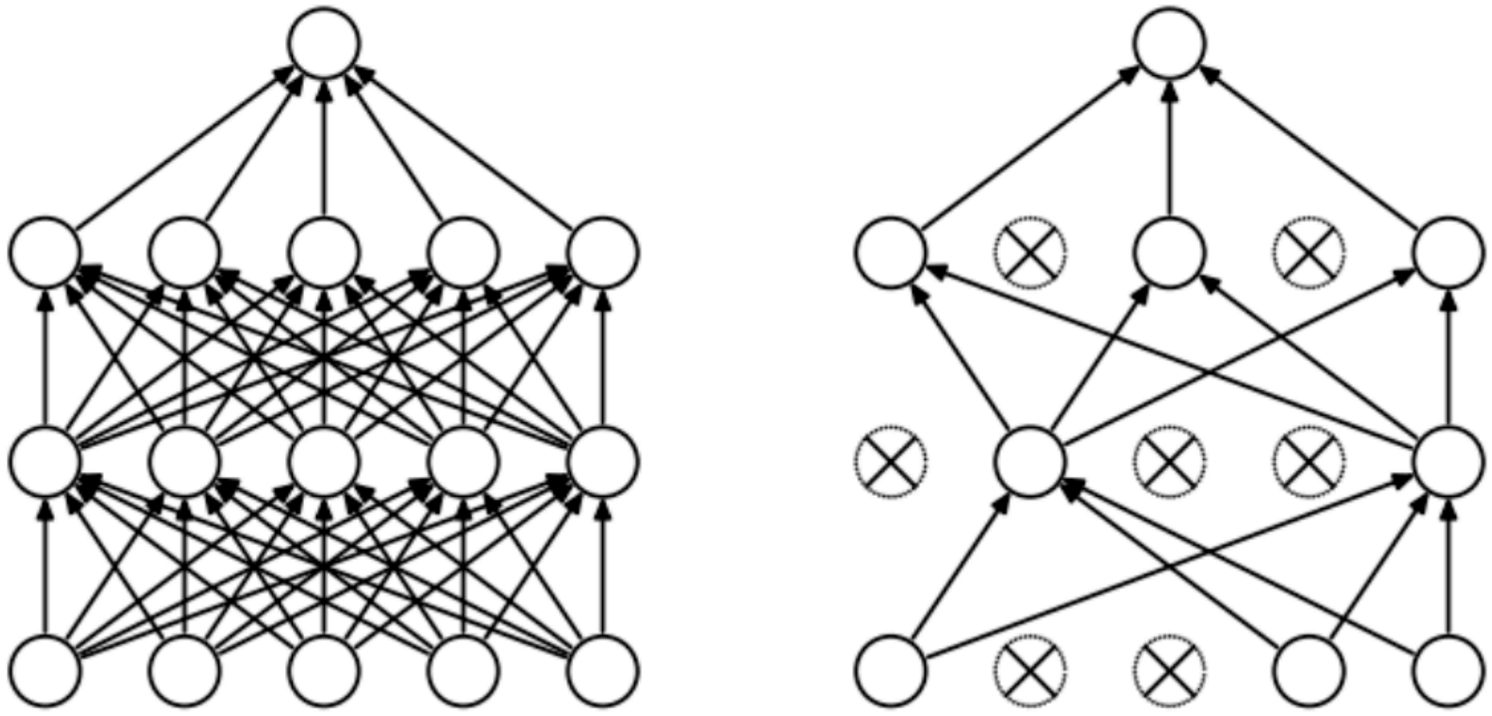


optimal

Dropout

Hinton, Geoffrey E.; Srivastava, Nitish; Krizhevsky, Alex; Sutskever, Ilya; Salakhutdinov, Ruslan R. (2012). "Improving neural networks by preventing co-adaptation of feature detectors". arXiv:1207.0580

- **Randomly drop out (set to zero) a number of output features of the layer during training.**



(a) Standard Neural Net (b) After applying dropout.

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

Simpler models are less likely to over- fit than complex ones.

- **Dropout rate: fraction of the features zeroed out (~0.2 and 0.5)**
- **Once trained units aren't dropped out;**
  - **layer's output are scaled down the dropout rate**
  - **Because more units are active than during training**
- **One of the most effective/common regularization techniques**
- **Why does it work? Introducing noise can break up "happenstance" patterns that aren't significant**

Dropout in Keras

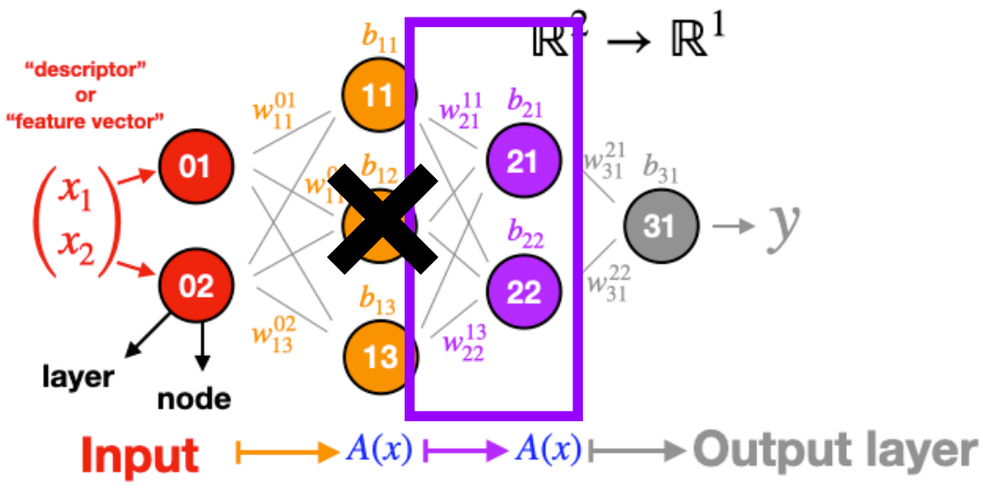
Dropout layer

```
tf.keras.layers.Dropout(rate, noise_shape=None, seed=None, **kwargs)
```

Applies Dropout to the input.

Arguments

- **rate**: Float between 0 and 1. Fraction of the input units to drop.
- **noise_shape**: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape `(batch_size, timesteps, features)` and you want the dropout mask to be the same for all timesteps, you can use `noise_shape=(batch_size, 1, features)`.
- **seed**: A Python integer to use as random seed.



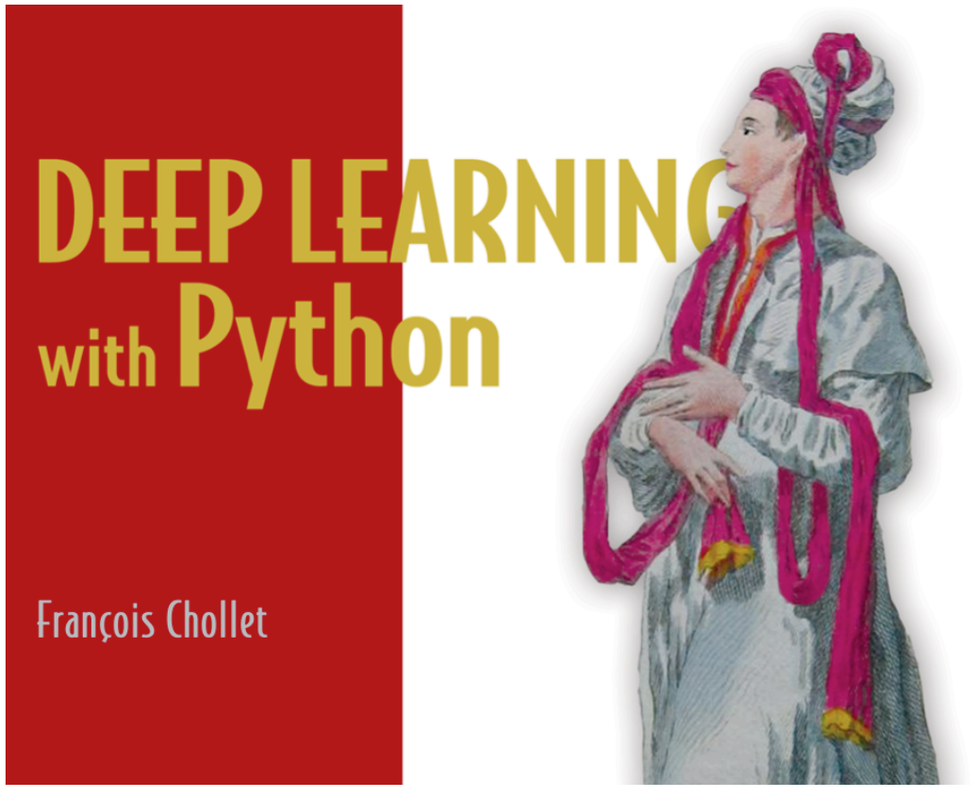
No dropout

$$\begin{pmatrix} O_{21} \\ O_{22} \end{pmatrix} = A\left(\begin{pmatrix} w_{21}^{11} & w_{21}^{12} & w_{21}^{13} \\ w_{22}^{11} & w_{22}^{12} & w_{22}^{13} \end{pmatrix} \begin{pmatrix} O_{11} \\ O_{12} \\ O_{13} \end{pmatrix} + \begin{pmatrix} b_{21} \\ b_{22} \end{pmatrix} \right)$$

With dropout

$$\begin{pmatrix} O_{21} \\ O_{22} \end{pmatrix} = A\left(\begin{pmatrix} w_{21}^{11} & w_{21}^{12} & w_{21}^{13} \\ w_{22}^{11} & w_{22}^{12} & w_{22}^{13} \end{pmatrix} \begin{pmatrix} O_{11} \\ \boxed{0.0} \\ O_{13} \end{pmatrix} + \begin{pmatrix} b_{21} \\ b_{22} \end{pmatrix} \right)$$

Reading Assignment (week-1)



3 *Getting started with neural networks* 56

3.1 Anatomy of a neural network 58

Layers: the building blocks of deep learning 58 ▪ *Models: networks of layers* 59 ▪ *Loss functions and optimizers: keys to configuring the learning process* 60

3.2 Introduction to Keras 61

Keras, TensorFlow, Theano, and CNTK 62 ▪ *Developing with Keras: a quick overview* 62

3.3 Setting up a deep-learning workstation 65

Jupyter notebooks: the preferred way to run deep-learning experiments 65 ▪ *Getting Keras running: two options* 66

Running deep-learning jobs in the cloud: pros and cons 66
What is the best GPU for deep learning? 66

3.4 Classifying movie reviews: a binary classification example 68

The IMDB dataset 68 ▪ *Preparing the data* 69
Building your network 70 ▪ *Validating your approach* 73
Using a trained network to generate predictions on new data 76 ▪ *Further experiments* 77 ▪ *Wrapping up* 77

3.5 Classifying newswires: a multiclass classification example 78

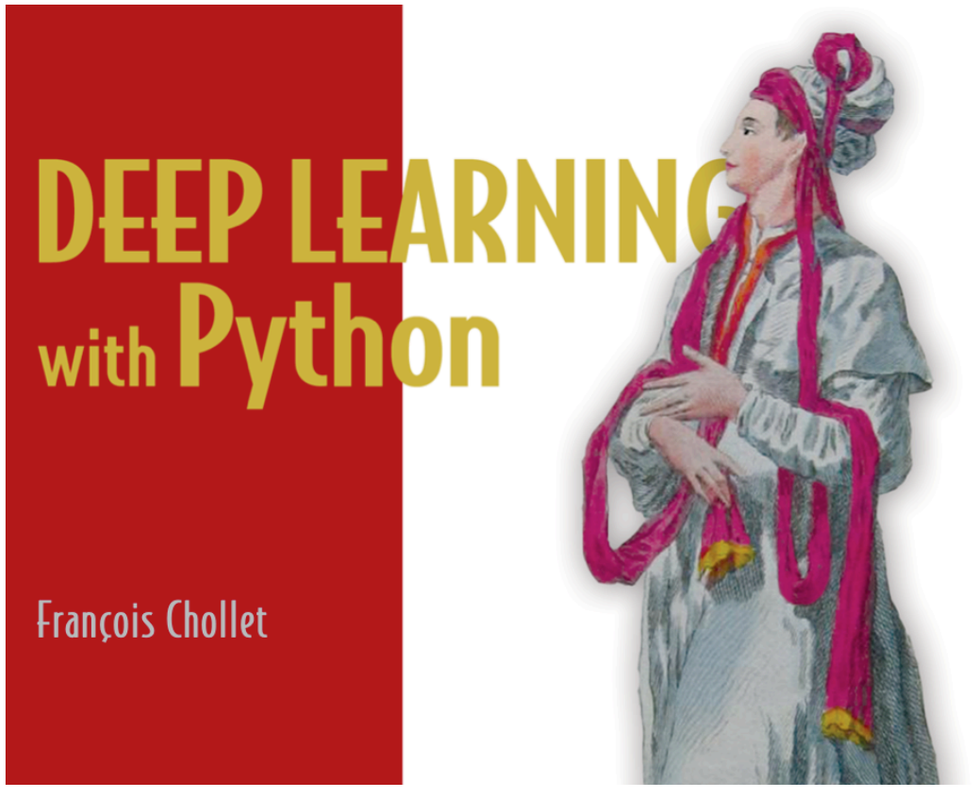
The Reuters dataset 78 ▪ *Preparing the data* 79
Building your network 79 ▪ *Validating your approach* 80
Generating predictions on new data 83 ▪ *A different way to handle the labels and the loss* 83 ▪ *The importance of having sufficiently large intermediate layers* 83 ▪ *Further experiments* 84 ▪ *Wrapping up* 84

3.6 Predicting house prices: a regression example 85

The Boston Housing Price dataset 85 ▪ *Preparing the data* 86 ▪ *Building your network* 86 ▪ *Validating your approach using K-fold validation* 87 ▪ *Wrapping up* 91

3.7 Chapter summary 92

Reading Assignment (week-2)



4 *Fundamentals of machine learning 93*

4.1 Four branches of machine learning 94

Supervised learning 94 ▪ Unsupervised learning 94
Self-supervised learning 94 ▪ Reinforcement learning 95

4.2 Evaluating machine-learning models 97

Training, validation, and test sets 97 ▪ Things to keep in mind 100

4.3 Data preprocessing, feature engineering, and feature learning 101

Data preprocessing for neural networks 101 ▪ Feature engineering 102

4.4 Overfitting and underfitting 104

Reducing the network's size 104 ▪ Adding weight regularization 107 ▪ Adding dropout 109

4.5 The universal workflow of machine learning 111

Defining the problem and assembling a dataset 111
Choosing a measure of success 112 ▪ Deciding on an evaluation protocol 112 ▪ Preparing your data 112
Developing a model that does better than a baseline 113
Scaling up: developing a model that overfits 114
Regularizing your model and tuning your hyperparameters 114

4.6 Chapter summary 116