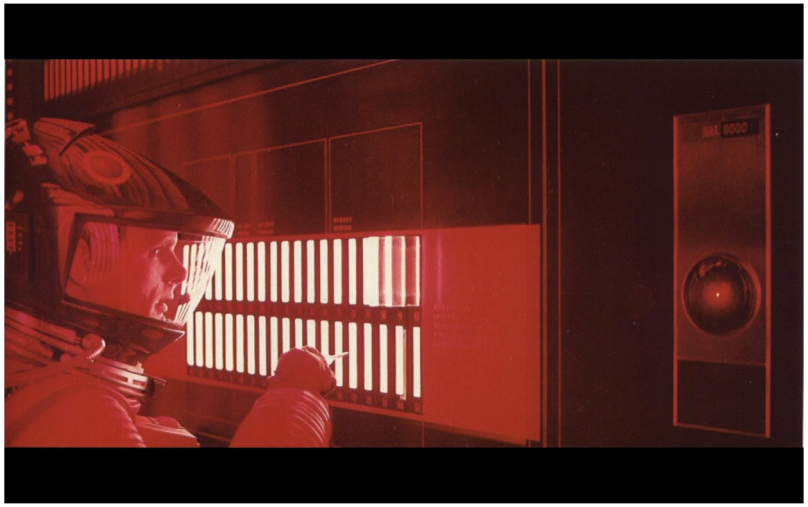
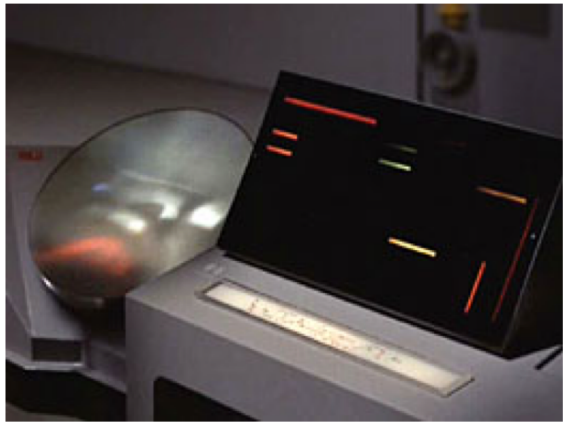
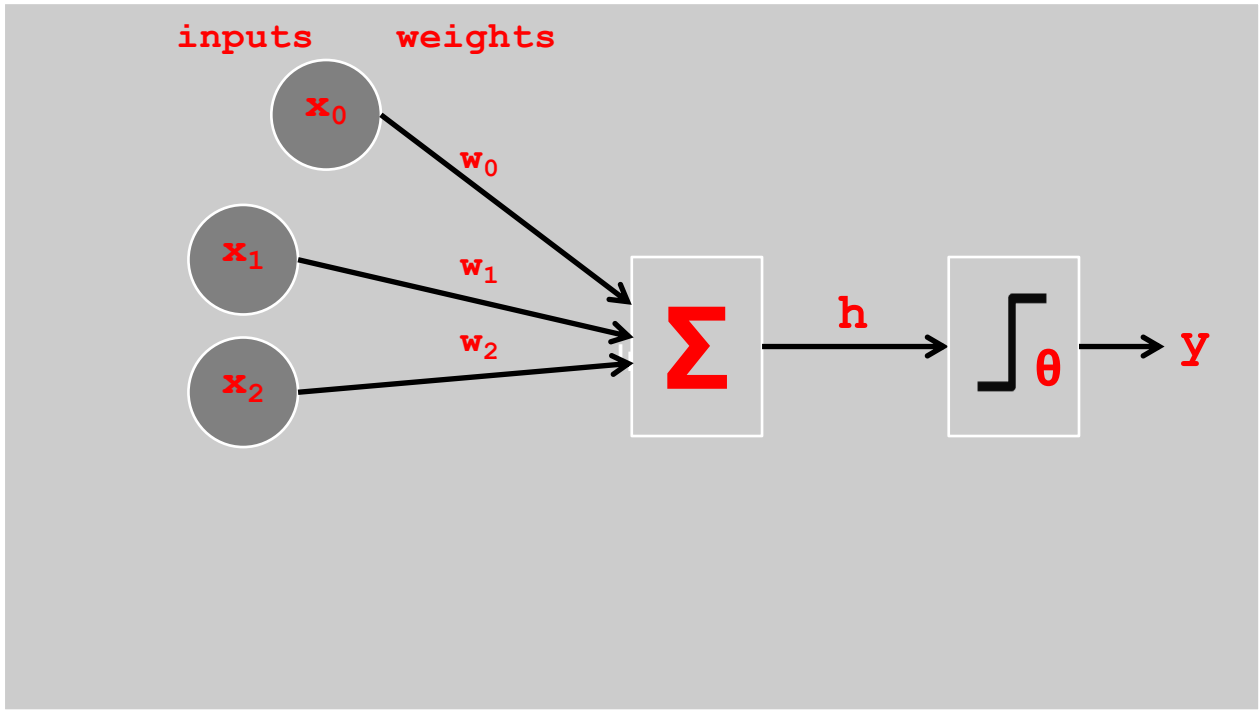
Perceptrons are The Future!

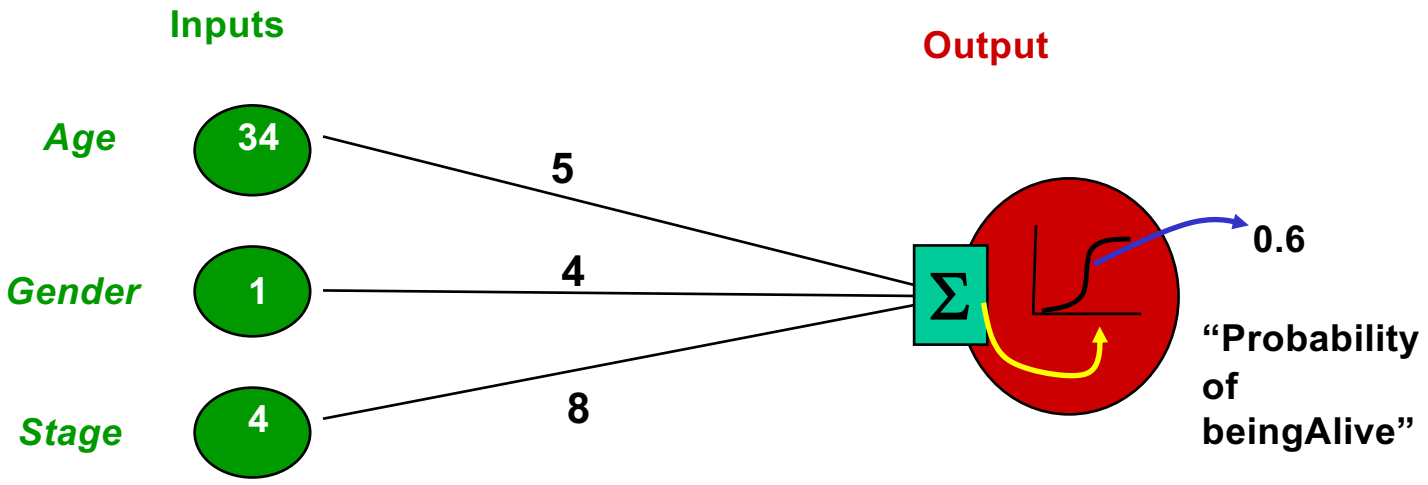




Perceptron Networks



Logistic Regression Model



Independent variables

x1, x2, x3

Coefficients

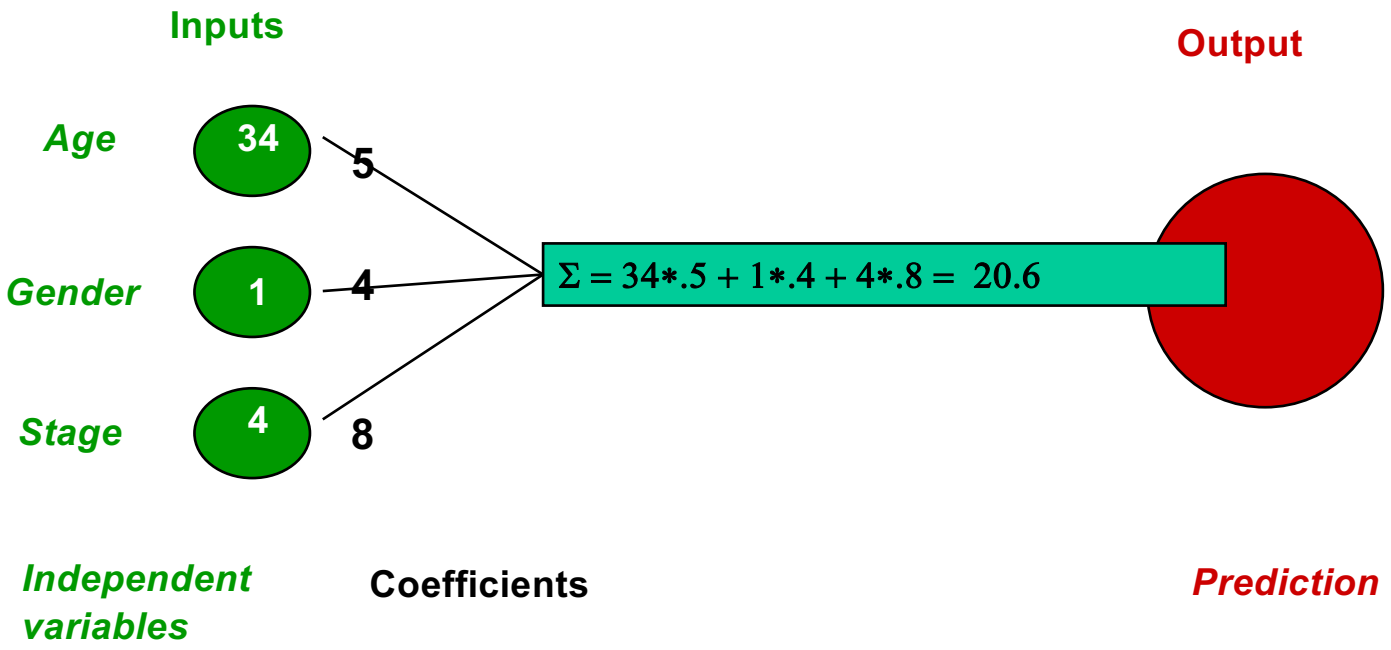
a, b, c

Dependent variable

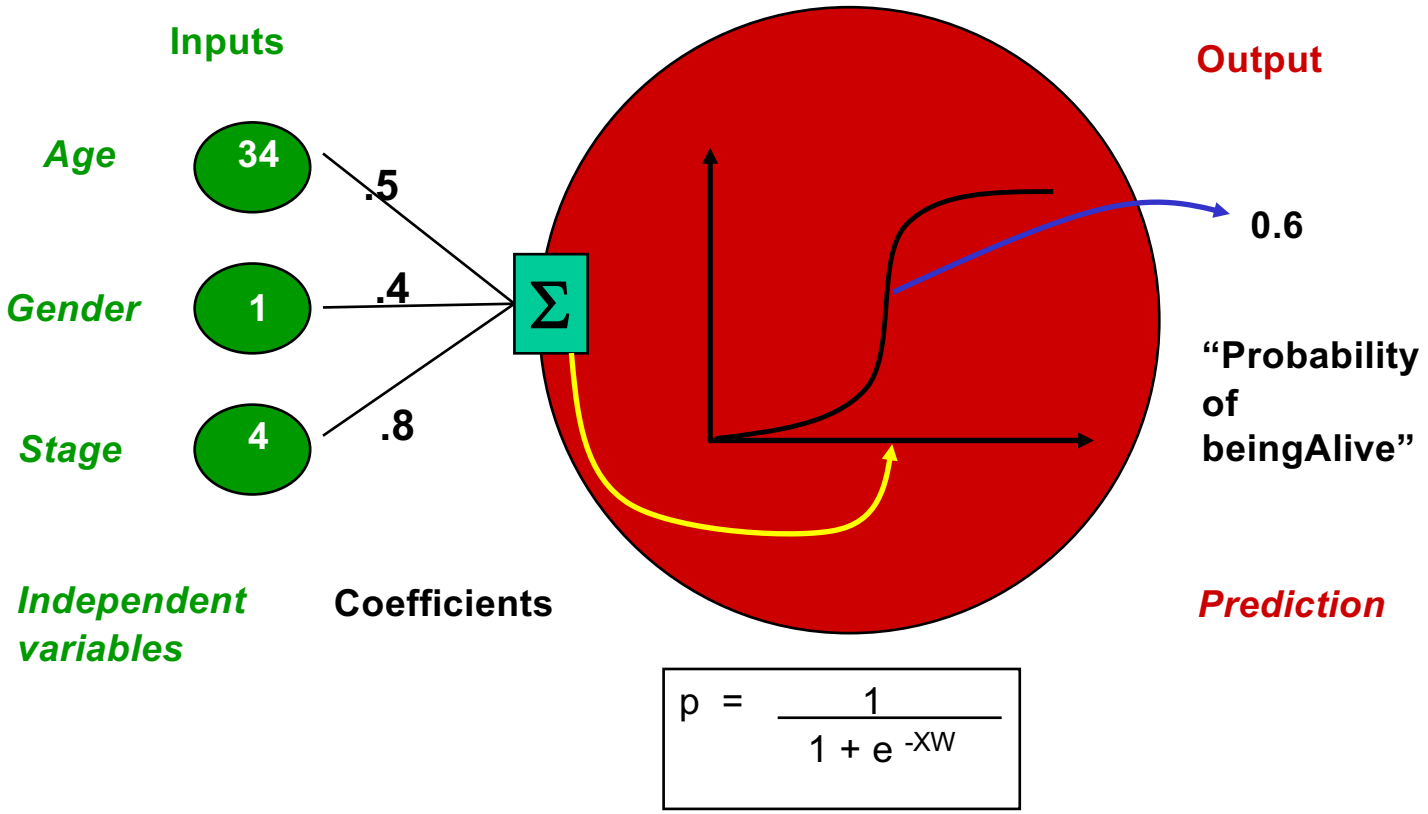
p

Prediction

Σ is the sum of inputs * weights



Logistic function

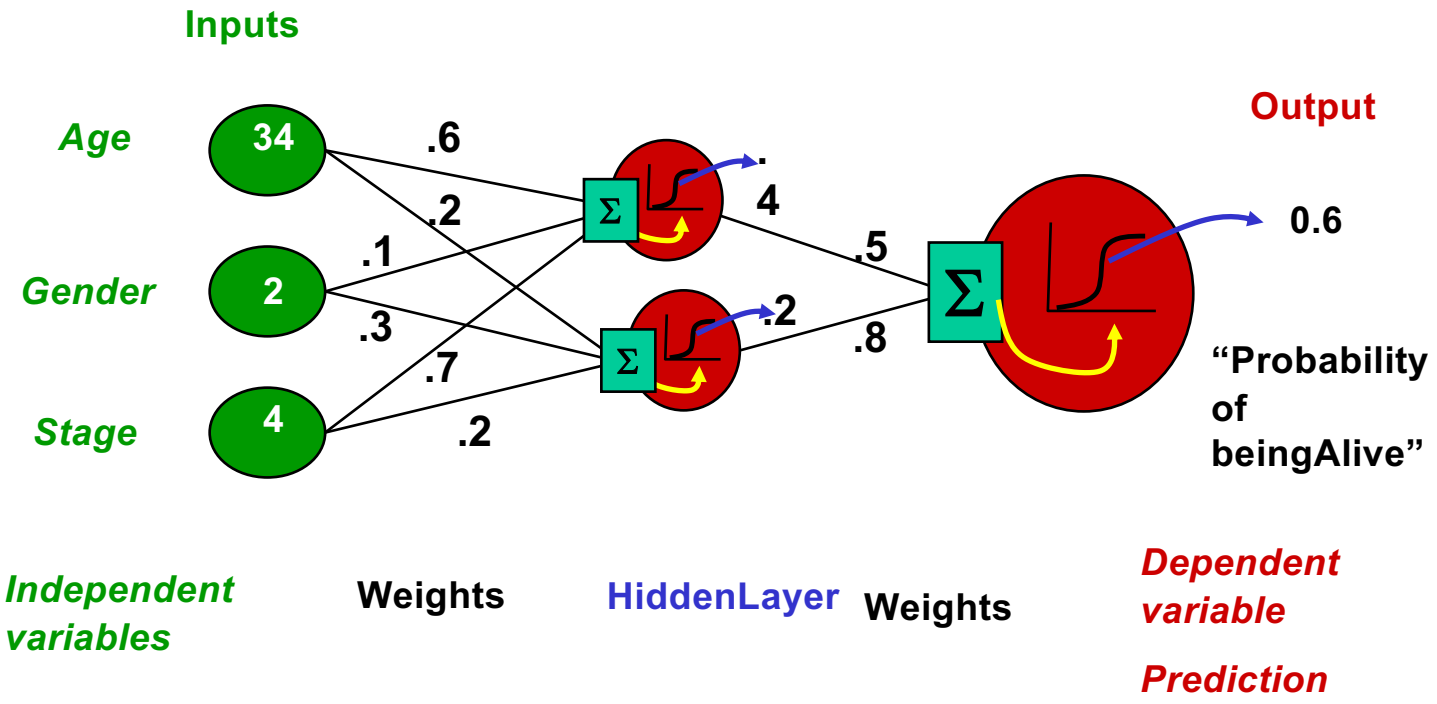


Activation Functions...

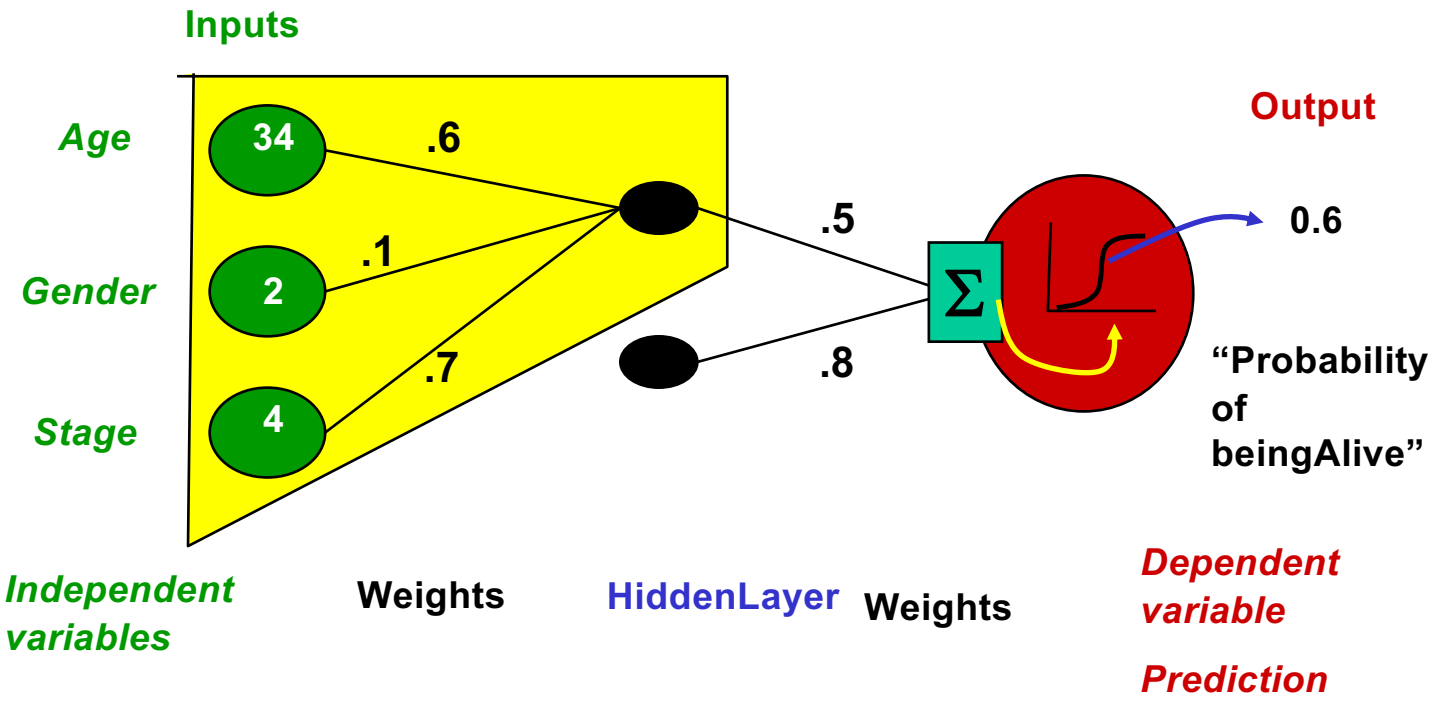
- Linear

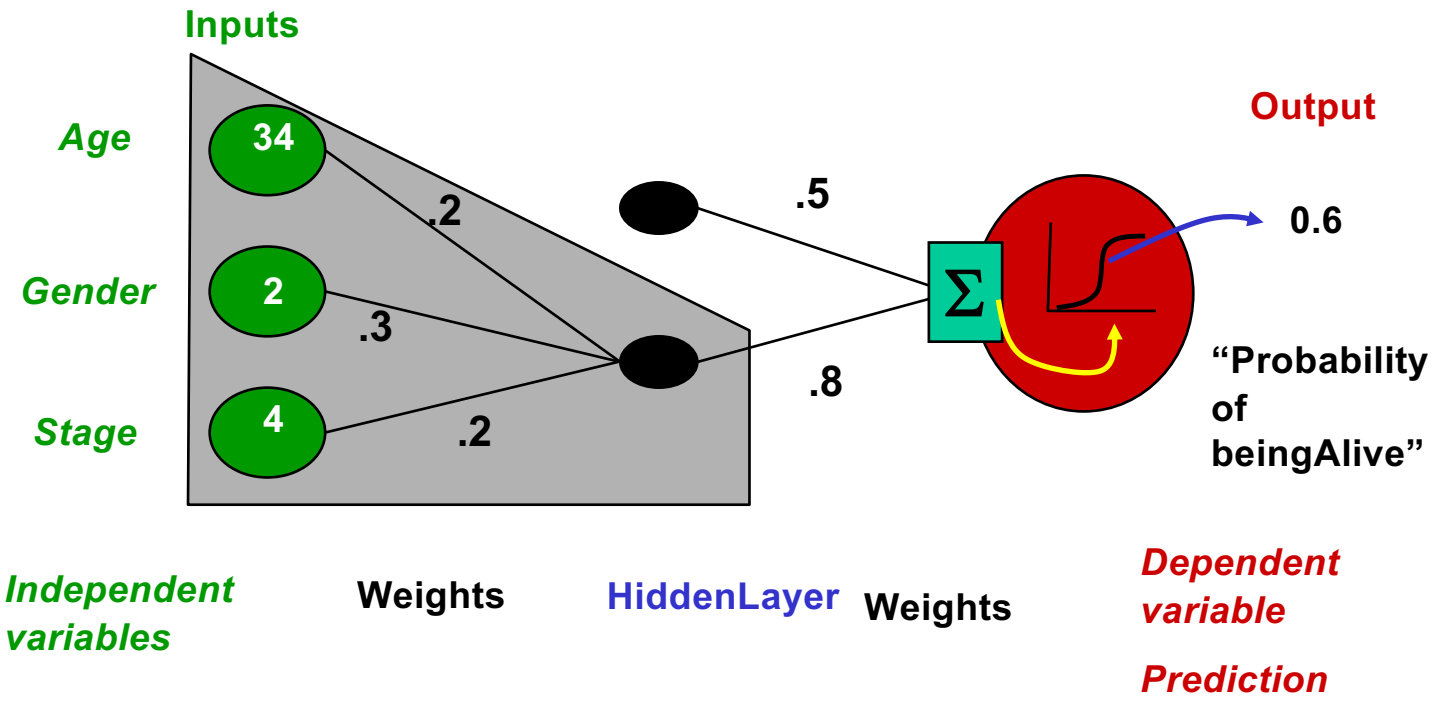
- Threshold or step function
- Logistic, sigmoid, “squash”
- Hyperbolic tangent

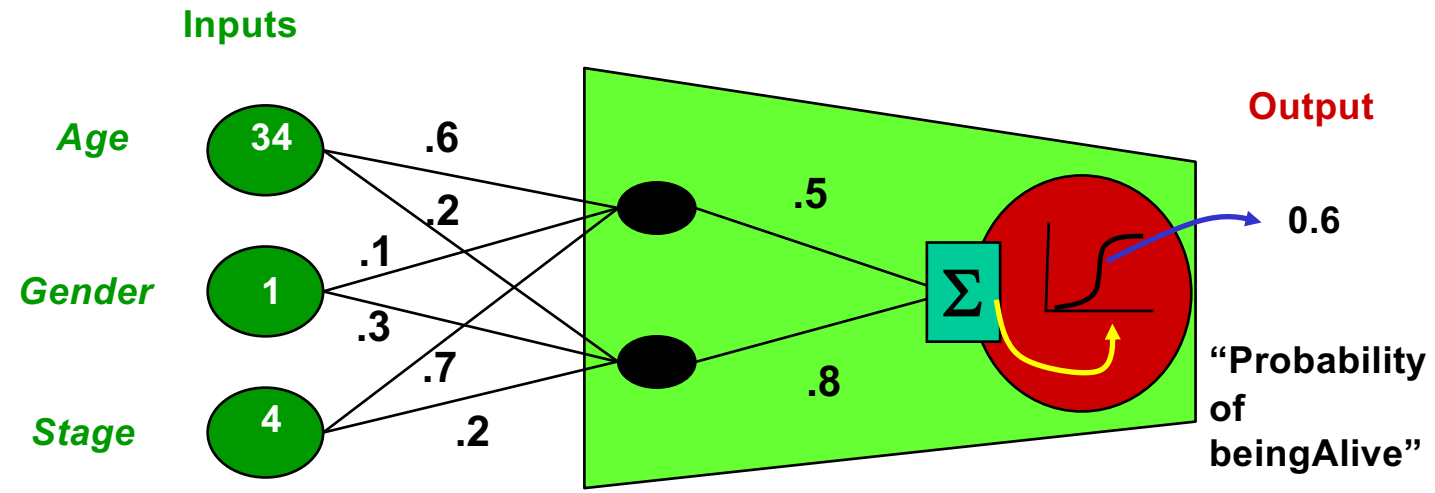
Neural Network Model



“Combined logistic models”

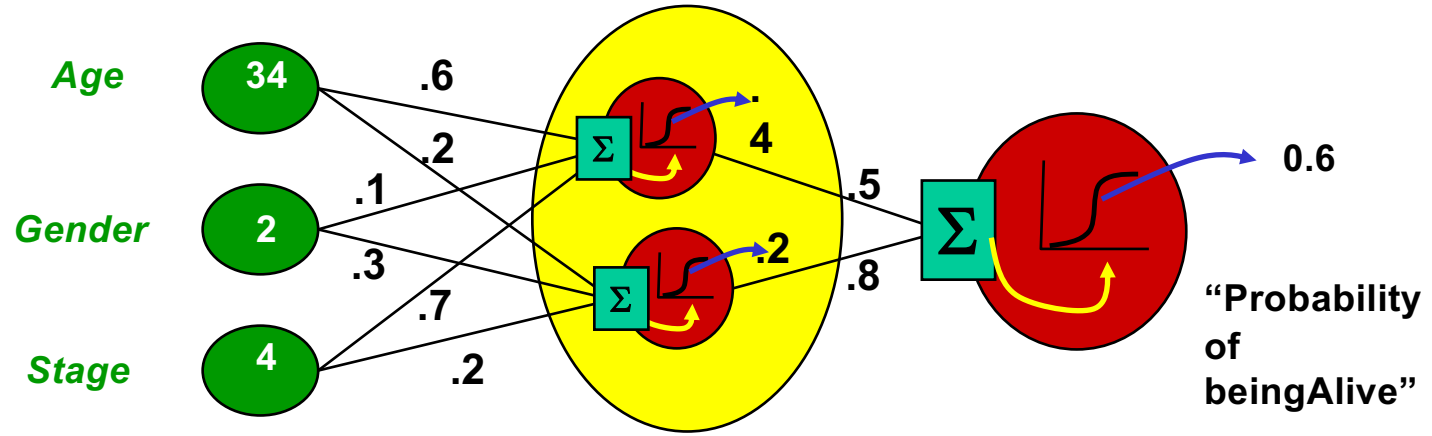



Inputs
Age
Gender
Stage
34
2
4
.2
.3
.2
.5
.8
Σ
Output
0.6
"Probability of beingAlive"
Independent variables
Weights
HiddenLayer
Weights
Dependent variable
Prediction

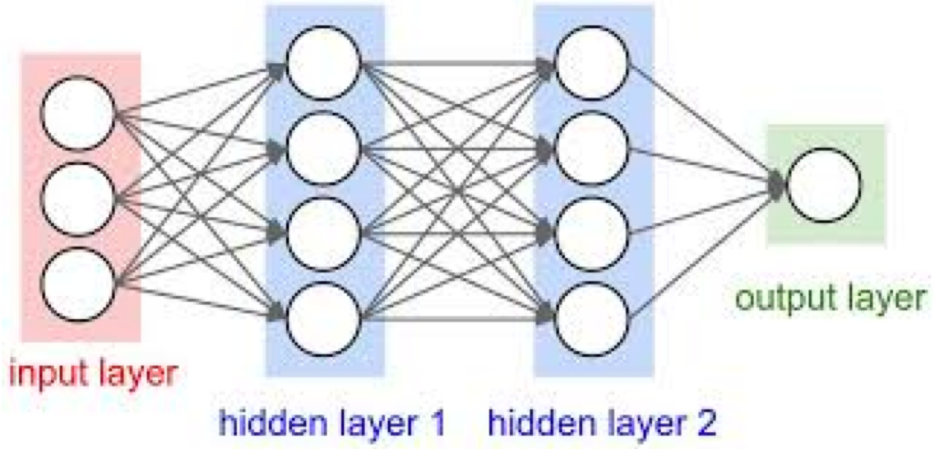

Inputs
Age
34
Gender
1
Stage
4
.6
.2
.1
.3
.7
.2
.5
.8
Σ
Output
0.6
"Probability of beingAlive"
Independent variables
Weights
HiddenLayer
Weights
Dependent variable
Prediction

Not really, no target for hidden units...

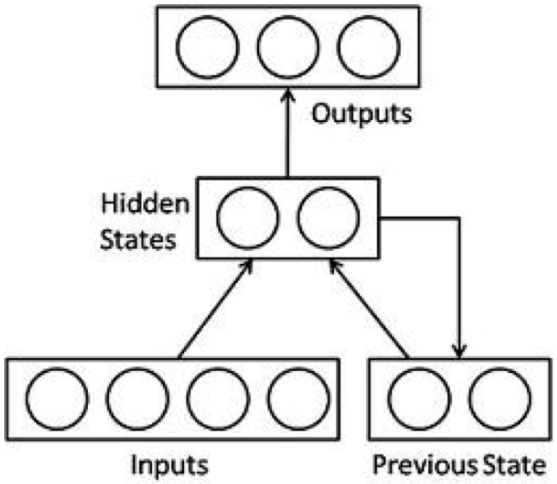
Hidden neurons help us model feature interactions
See manifold section below for more details

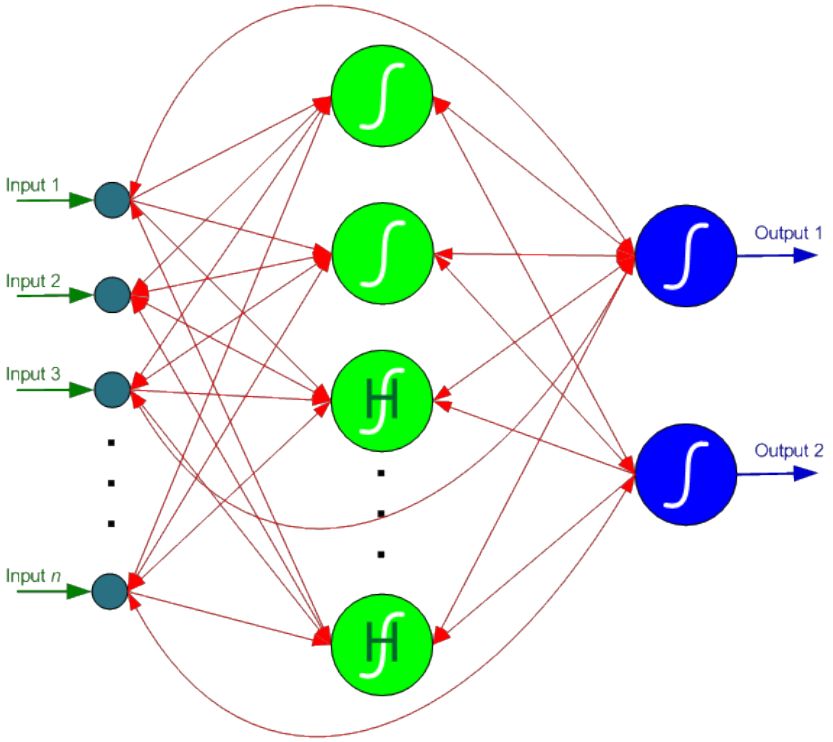


MLP, feed forward neural network



Neural Network Architectures
Recurrent Neural Networks





*H: indicates a "hidden" neuron without a target output

Neural Network Architectures

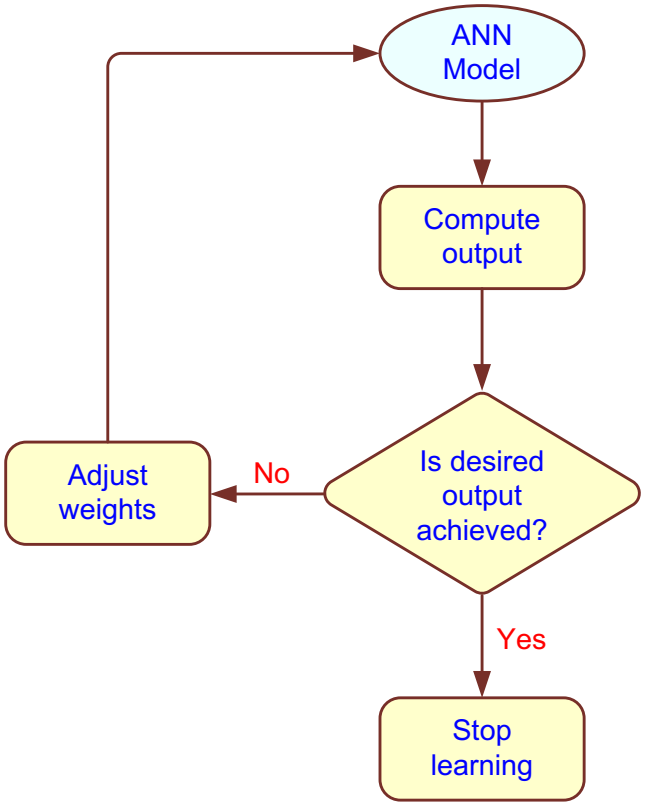
- **Architecture of a neural network is driven by the task it is intended to address**
  - Classification, regression, clustering, general optimization, association, ….
- **Most popular architecture: Feedforward, multi-layered perceptron with backpropagation learning algorithm**
  - Used for both classification and regression type problems

Learning Algos

- Supervised
- Unsupervised
- Semi-Supervised

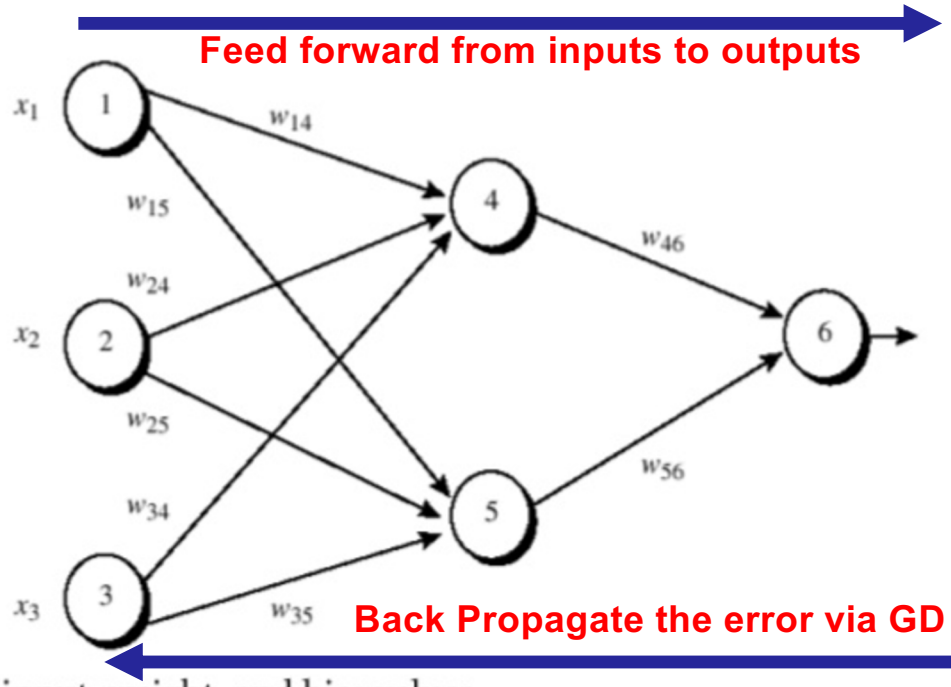
- BackProp

A Supervised Learning Process



Three-step process:

1. Compute temporary outputs
2. Compare outputs with desired targets
3. Adjust the weights and repeat the process



Δ rule

change weights to decrease the error

Bias terms for nodes 4,5,6

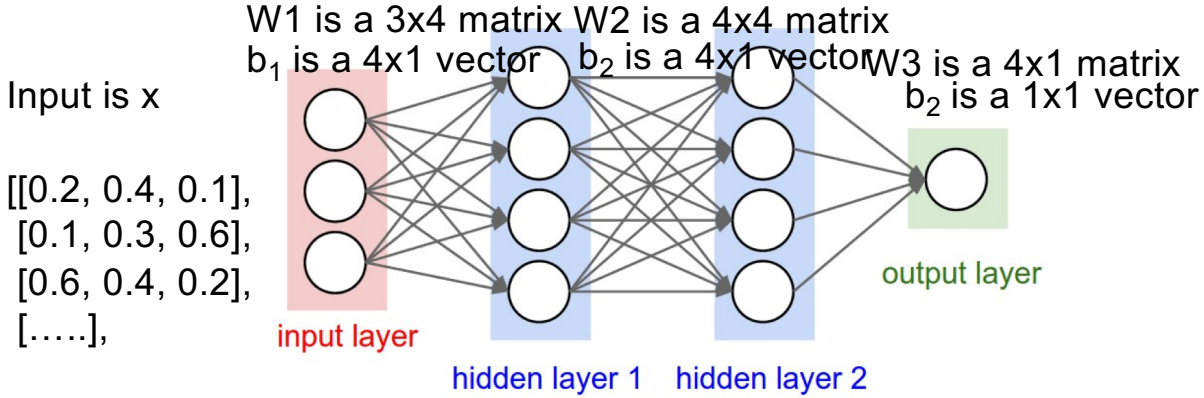
Initial input, weight, and bias values.

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

Unit j	Net input, I_j	Output, O_j
4	$0.2+0-0.5-0.4=-0.7$	$1/(1+e^{0.7})=0.332$
5	$-0.3+0+0.2+0.2=0.1$	$1/(1+e^{-0.1})=0.525$
6	$(-0.3)(0.332)-(0.2)(0.525)+0.1=-0.105$	$1/(1+e^{0.105})=0.474$

Layers enable vectorized computations

Example Feed-forward computation of a Neural Network




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Outline

1. **Introduction**
2. **ML and deep learning review**
    1. Perceptron → MLP
    2. Keras
    3. BackPropagation
3. **What is computer vision?**
    1. Computer vision
    2. Convolutional Neural Nets (CNNs)
4. **Deep NN Architectures for CV Tasks**
    1. Backbone networks
5. **Solving edge-based IoT**
6. **Conclusions and Next steps**

Deep Learning EcoSystem

- **High Performance GPU-Acceleration for Deep Learning**

- **What is Keras?**
  - Neural Network library written in Python
  - Designed to be minimalistic & straight forward yet extensive Built on top of either Theano or newly TensorFlow
- **Why use Keras?**
  - Simple to get started, simple to keep going
  - Written in python and highly modular; easy to expand Deep enough to build serious models

Keras is minimal, modular and extensible

- **Keras** is an open source neural network library written in Python. It is capable of running on top of either Tensorflow or Theano.[1] Designed to enable fast experimentation with deep neural networks, it focuses on being minimal, modular and extensible.
- The library contains numerous implementations of commonly used neural network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier.
- As of Sept 16, 2016, Keras is the second fastest growing deep learning framework after Google's Tensorflow.
- As of September 2017, Keras a core part of Tensorflow

Keras: a model-level library

- Keras is a model-level library, providing high-level building blocks for developing deep learning models.
  - It does not handle itself low-level operations such as tensor products, convolutions and so on.
  - Instead, it relies on a specialized, well-optimized tensor manipulation library to do so, serving as the "backend engine" of Keras. Rather than picking one single tensor library and making the implementation of Keras tied to that library, Keras handles the problem in a modular way, and several different backend engines can be plugged seamlessly into Keras.
- At this time, Keras has two backend implementations available:
  - the TensorFlow backend and the Theano backend.
    - TensorFlow is an open-source symbolic tensor manipulation framework developed by Google, Inc.
    - Theano is an open-source symbolic tensor manipulation framework developed by LISA/MILA Lab at Université de Montréal.

Switching from one backend to another

If you have run Keras at least once, you will find the Keras configuration file at:

`~/.keras/keras.json`

If it isn't there, you can create it.

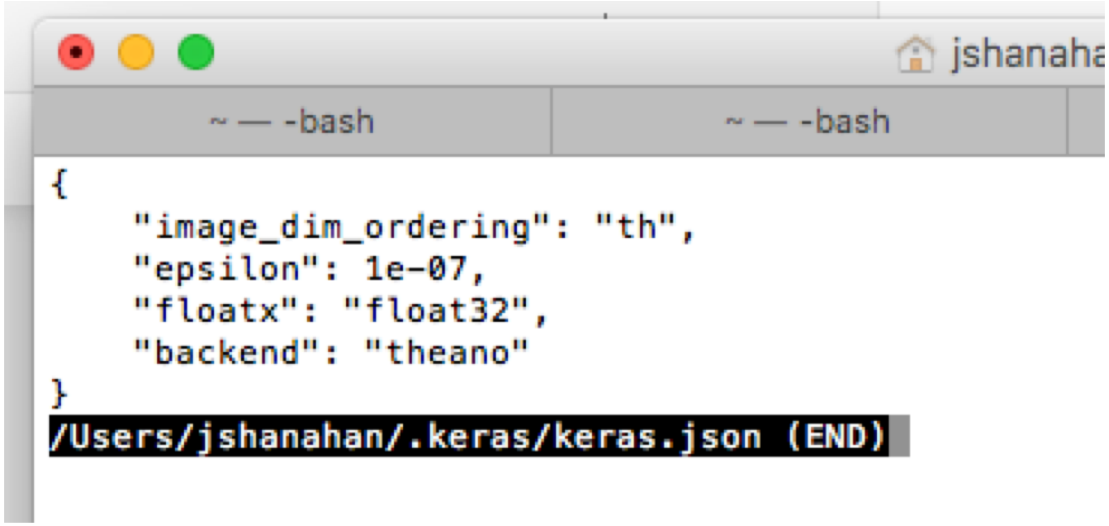
The default configuration file looks like this:

```
{
    "image_dim_ordering": "tf",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

Simply change the field `backend` to either `"theano"` or `"tensorflow"`, and Keras will use the new configuration next time you run any Keras code.

Switching from one backend to another

- Simply change the field backend to either "theano" or "tensorflow", and Keras will use the new configuration next time you run any Keras code.
-



Outline

- **Neural Networks Introduction**
  - Perceptron
  - MLP
  - Learning via Gradient Descent
- **Keras**
- **5 steps to create a NN/deep learning model**
- **Labs**
- **BackProp Algorithm**

5 steps to create a NN/deep learning model in Keras

5 steps to create a NN/deep learning model

1: Load data and do EDA

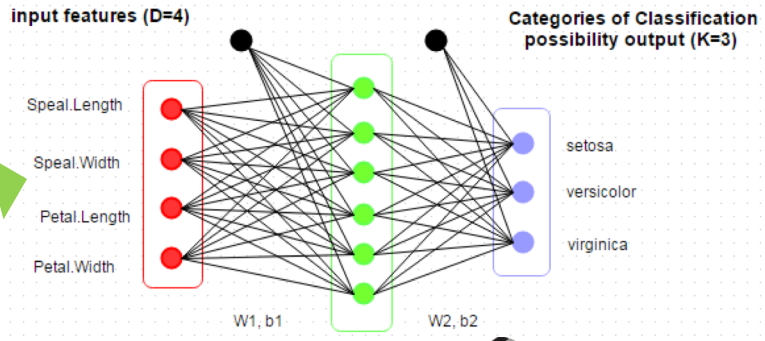
2: Define Network

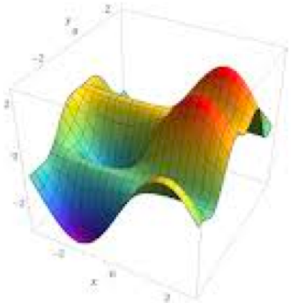
3: Compile Network

4: Fit Network

5: Evaluate network

6: Deploy: Make predictions





5 steps to create a NN/deep learning model

- Create a neural network model using the powerful Keras Python library for deep learning

5 Steps

1. How to load data.
2. How to define neural network in Keras.
3. How to compile a Keras model using the efficient numerical backend (such as Tensorflow or Theano)
4. How to train a model on data.
5. How to evaluate a model on data.
6. *Make predictions (deploy)*

Binary cross entropy example

- Keras library
  - Keras Tutorial
  - Example Notebook
    - Binary cross entropy example
      - http://machinelearningmastery.com/binary-classification-tutorial-with-the-keras-deep-learning-library/
    - MLP

Classification Notebooks

- **Pima diabetes: binary classification problem**
  - http://localhost:8890/notebooks/shared/Dropbox/Projects/UniversityOFGhent-2017/Src/Untitled%20Folder/Pima-Diabetes-First-NN.ipynb
- **Iris: Three class classification problem**
  - http://localhost:8890/notebooks/shared/Dropbox/Projects/UniversityOFGhent-2017/Src/U

Notebook hidden (quiz)

Main Steps in learning a NN

1. **Load data and do EDA**

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

2. **Define the model architecture**

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
```

SoftMax

3. **Compile the model using a numerical backend (and set objective)**

```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

4. **Fit the model**

```
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

5. **Evaluate**

```
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

5 steps to create a NN/deep learning model

1: Load data and do EDA

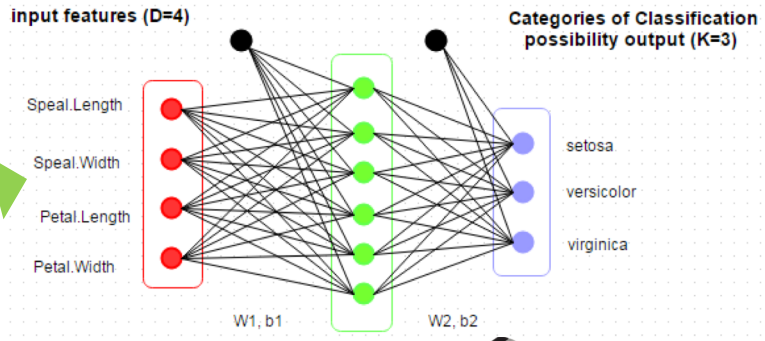
2: Define Network

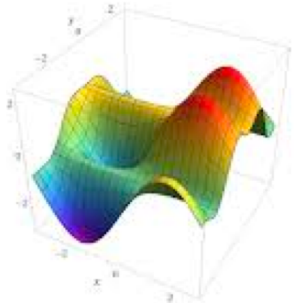
3: Compile Network

4: Fit Network

5: Evaluate network

6: Make predictions





Datasets

PIMA diabetes

File size: 23.31 KB

Description: **Predict the Onset of Diabetes**

Attributes:

Pregnancies: Number of times pregnant

Glucose: Plasma glucose concentration a 2 hours in an oral glucose tolerance test

BloodPressure: Diastolic blood pressure (mm Hg)

SkinThickness: Triceps skin fold thickness (mm)

Insulin: 2-Hour serum insulin (mu U/ml)

BMI: Body mass index (weight in kg/(height in m)^2)

DiabetesPedigreeFunction: Diabetes pedigree function

Age: Age (years)

Outcome: Class variable (0 or 1)

Table [768 females, 9]

https://www.kaggle.com/uciml/pima-indians-diabetes-database

File Preview:

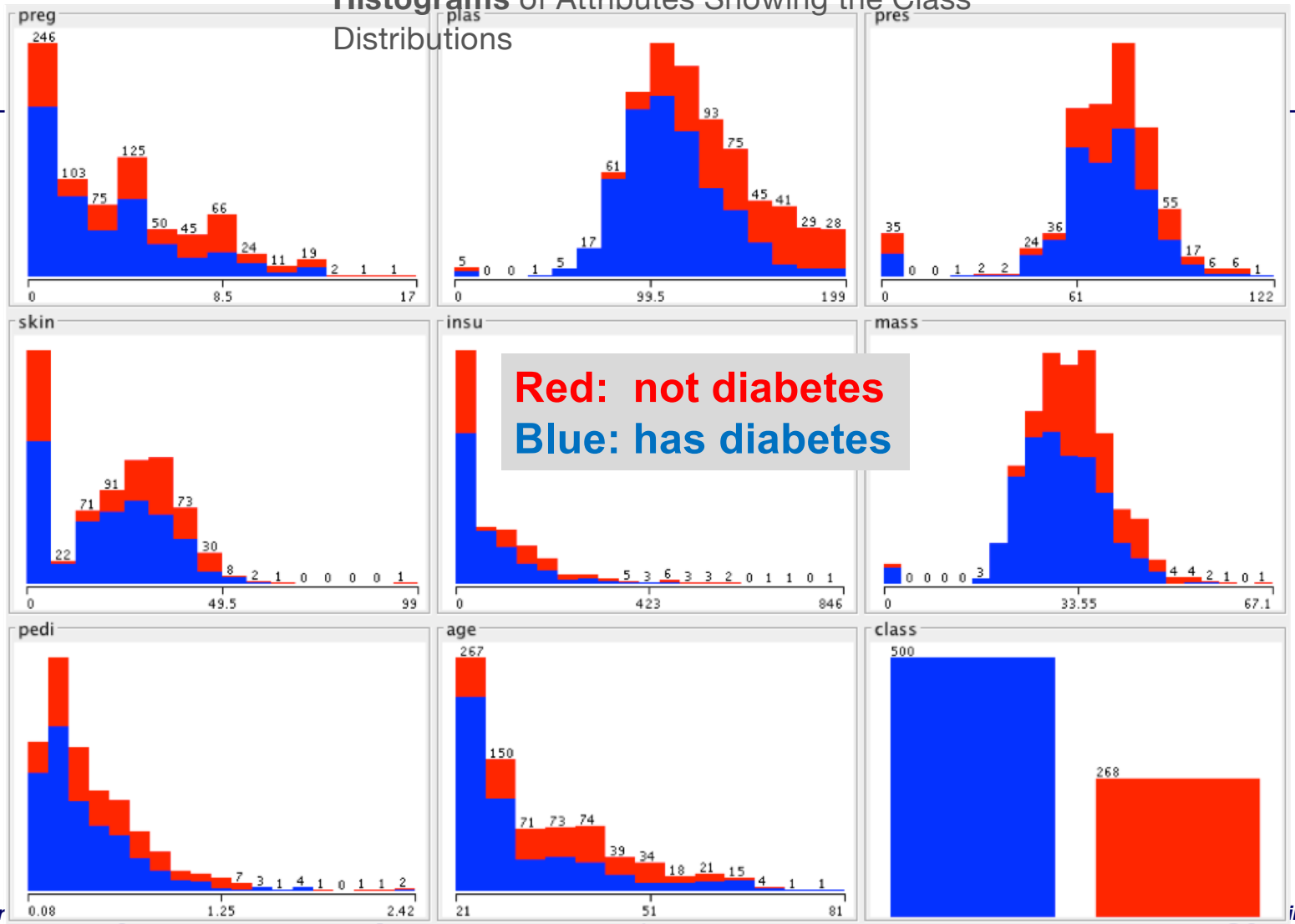
Edit column descriptions

Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288		

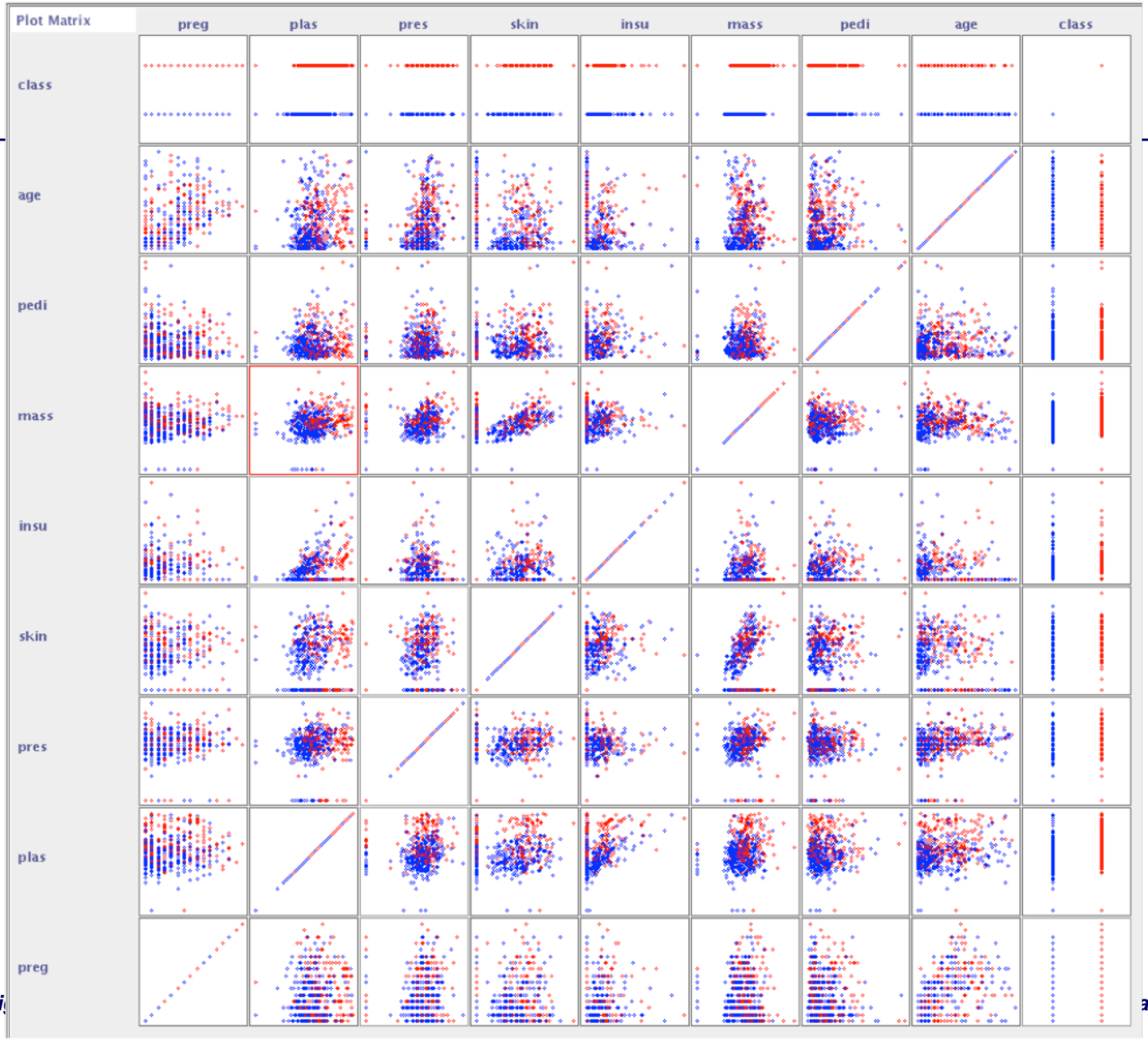
Evaluation of various ML Algorithms on PIMA Diabetes

- **See this article**
  - http://machinelearningmastery.com/case-study-predicting-the-onset-of-diabetes-within-five-years-part-2-of-3/

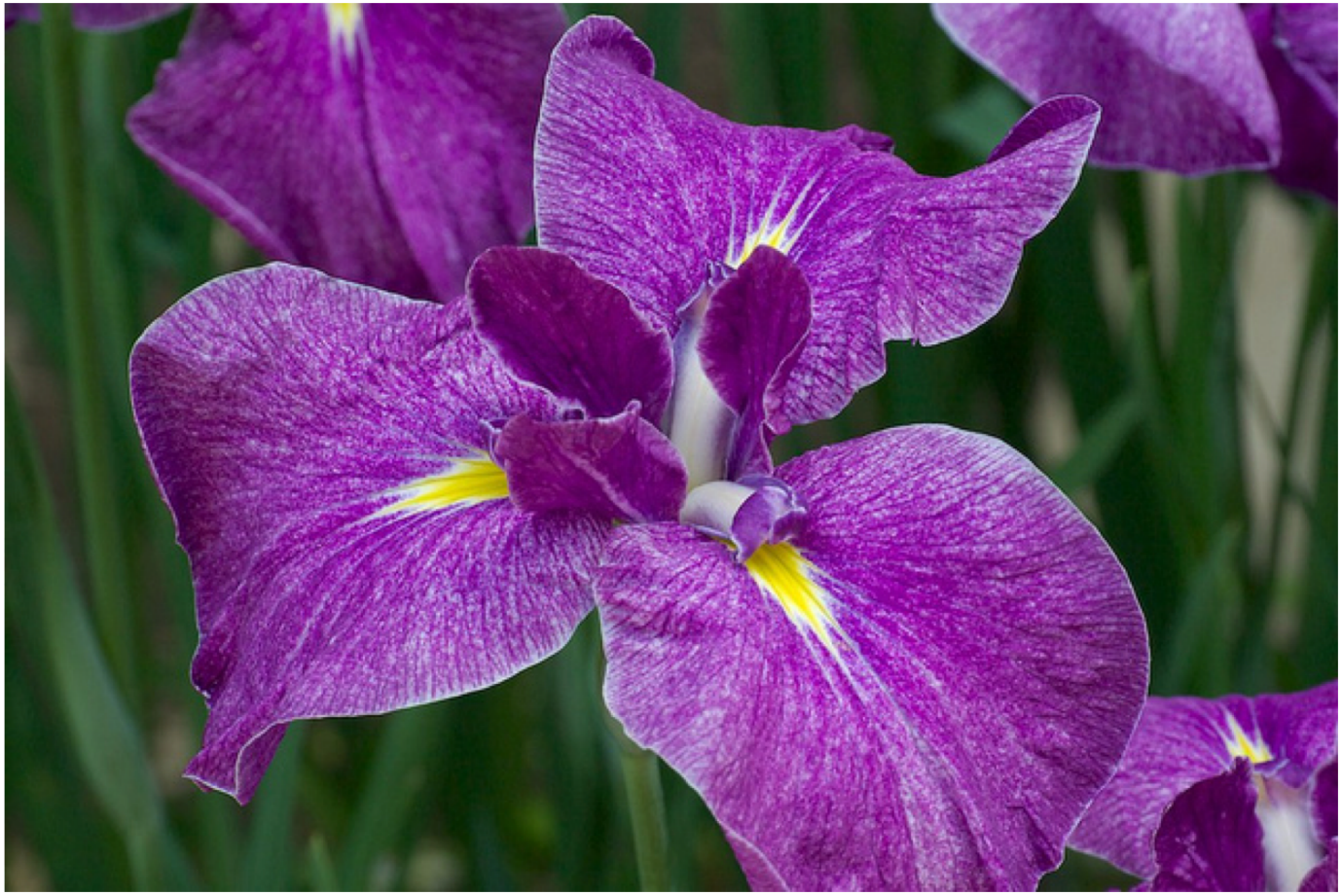
Histograms of Attributes Showing the Class Distributions

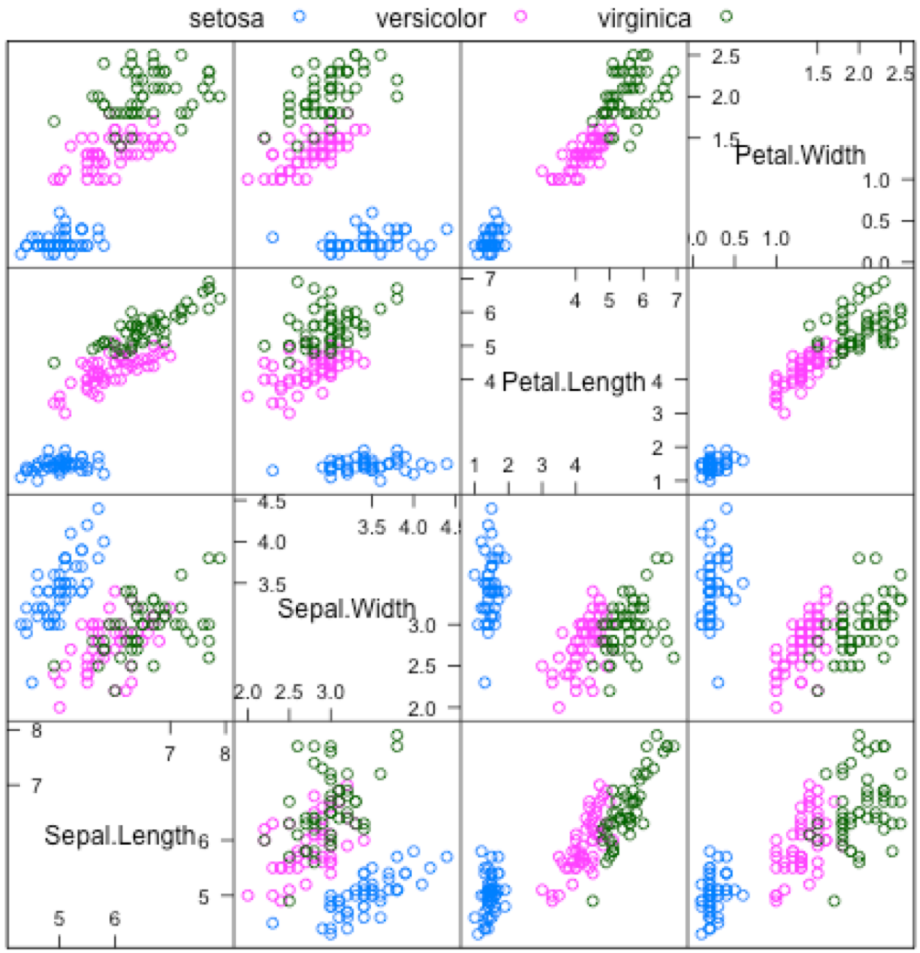


Red: not diabetes
Blue: has diabetes


Plot Matrix
preg
plas
pres
skin
insu
mass
pedi
age
class
class
age
pedi
mass
insu
skin
pres
plas
preg

Iris Dataset:
4 real-val$^{d.}$ inputs; 150 examples; 3 categories





Scatter Plot Matrix

5 steps to create a NN/deep learning model

1: Load data and do EDA

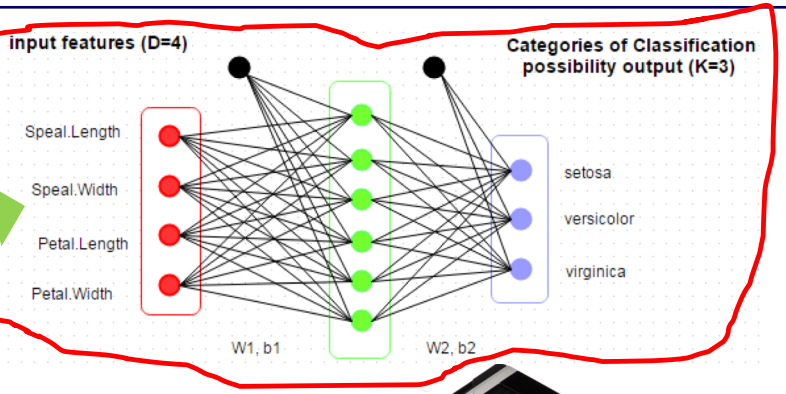
2: Define Network

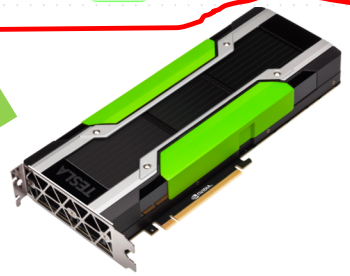
3: Compile Network

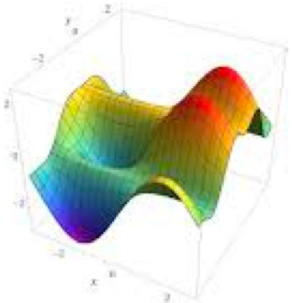
4: Fit Network

5: Evaluate network

6: Make predictions







Define your neural network in Keras: Sequence container PLUS Layers

- **The first step is to define your neural network as a Sequence container PLUS Layers.**
  - Sequence of layers
    - Neural networks are defined in Keras as a sequence of layers.
  - Sequential container
    - The container for these layers is the Sequential class.

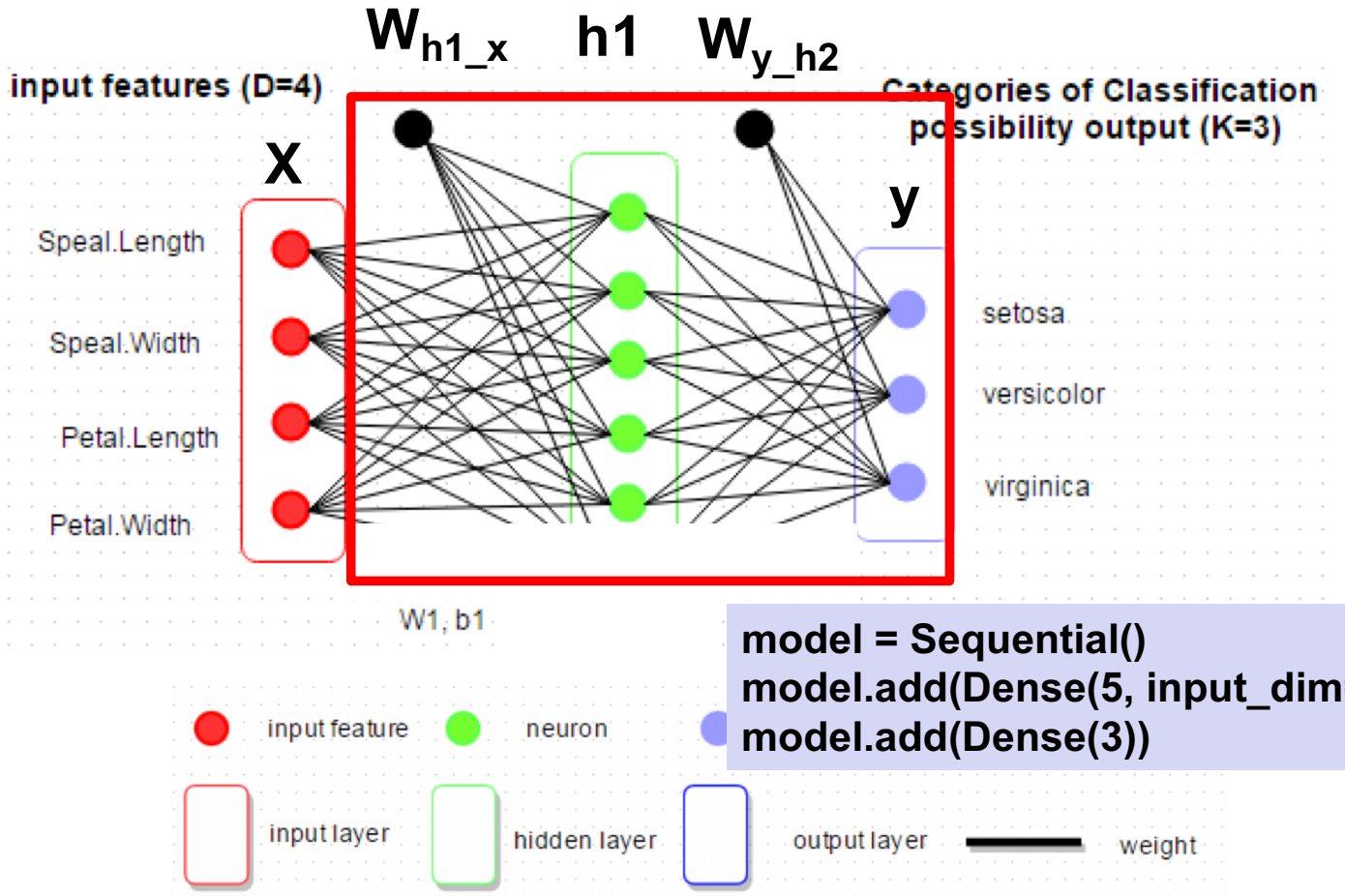
1. **The first step is to create an instance of the Sequential class.**
2. **Then you can create your layers and add them in the order that they should be connected.**

- **For example**,
  - a small Multilayer Perceptron model with 2 inputs in the visible layer, 5 neurons in the hidden layer and one neuron in the output layer can be defined as:

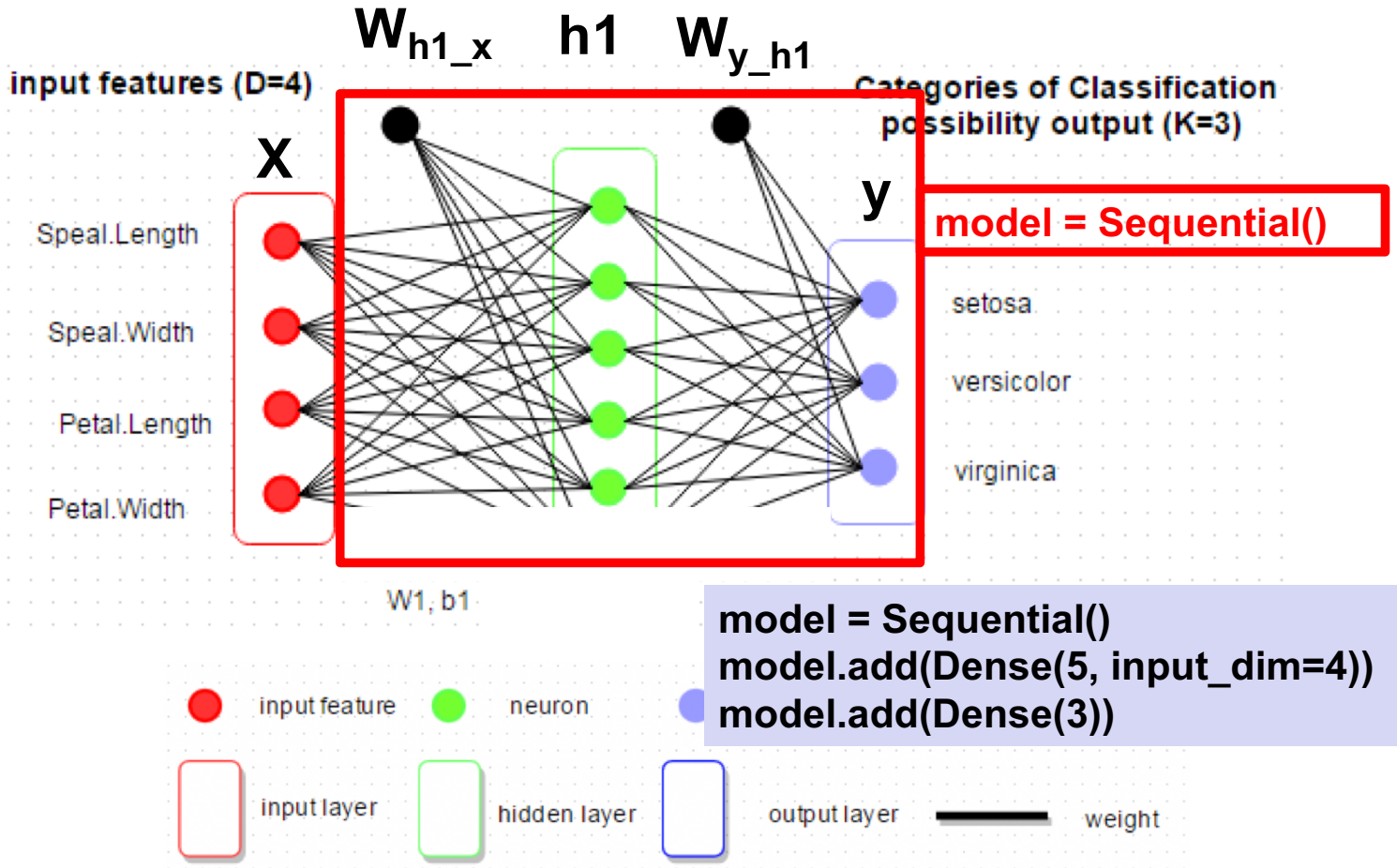
2-5-1

```
1 model = Sequential()
2 model.add(Dense(5, input_dim=2))
3 model.add(Dense(1))
```

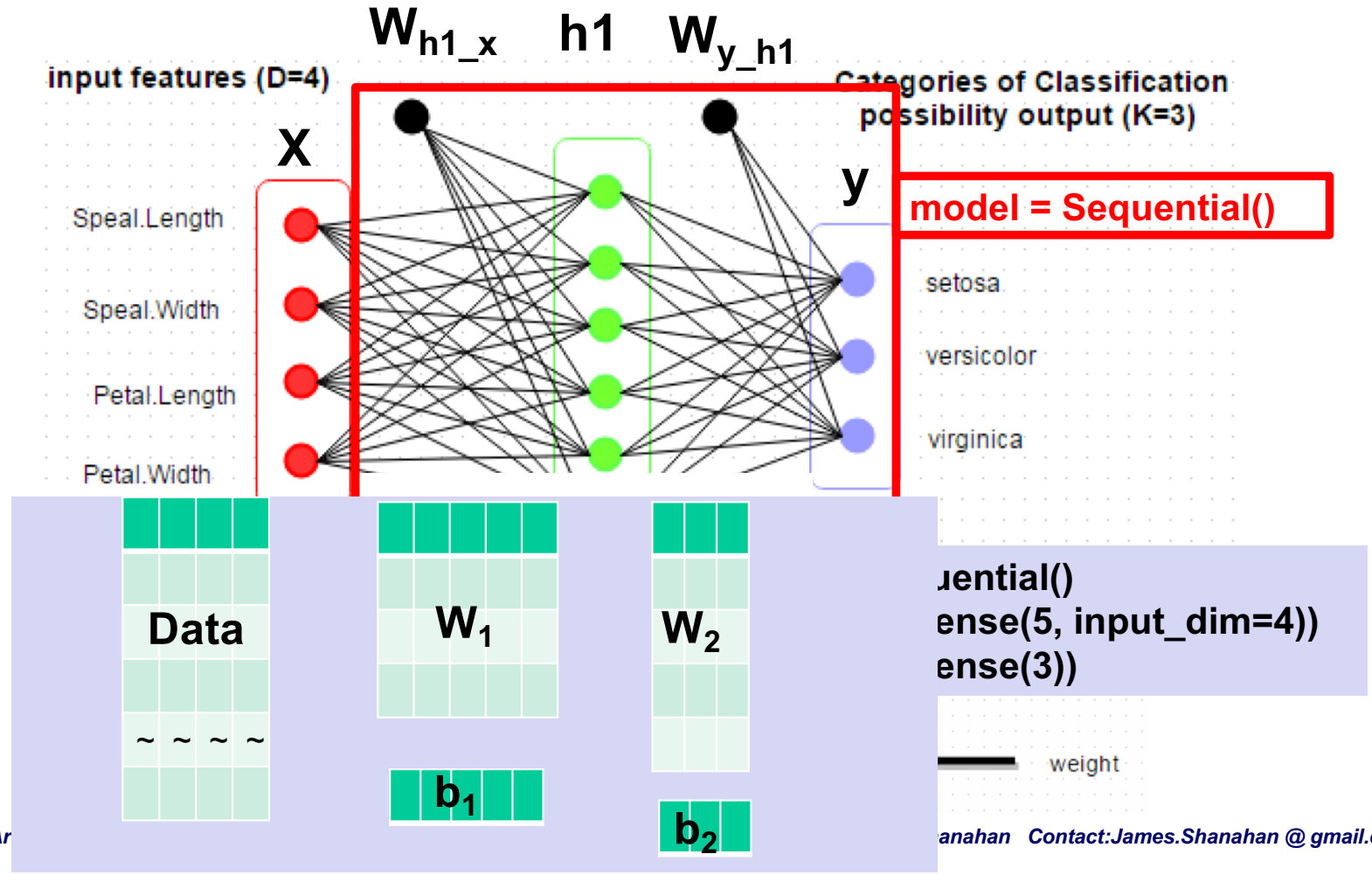
Let's work with a 4 – 5 – 3 MLP



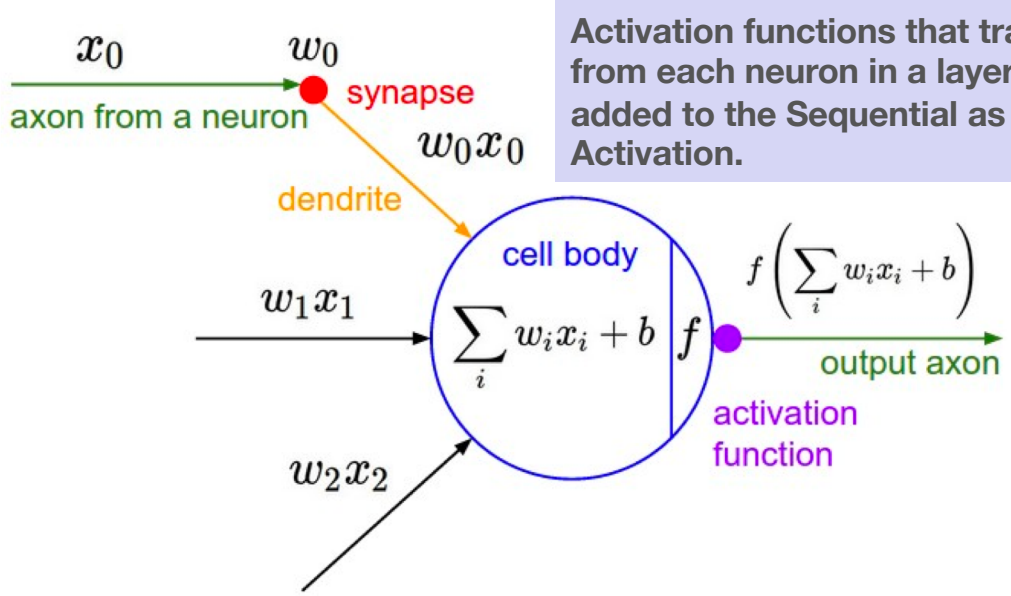
Let's work with a 4 – 5 – 3 MLP



Let's work with a 4 – 5 – 3 MLP



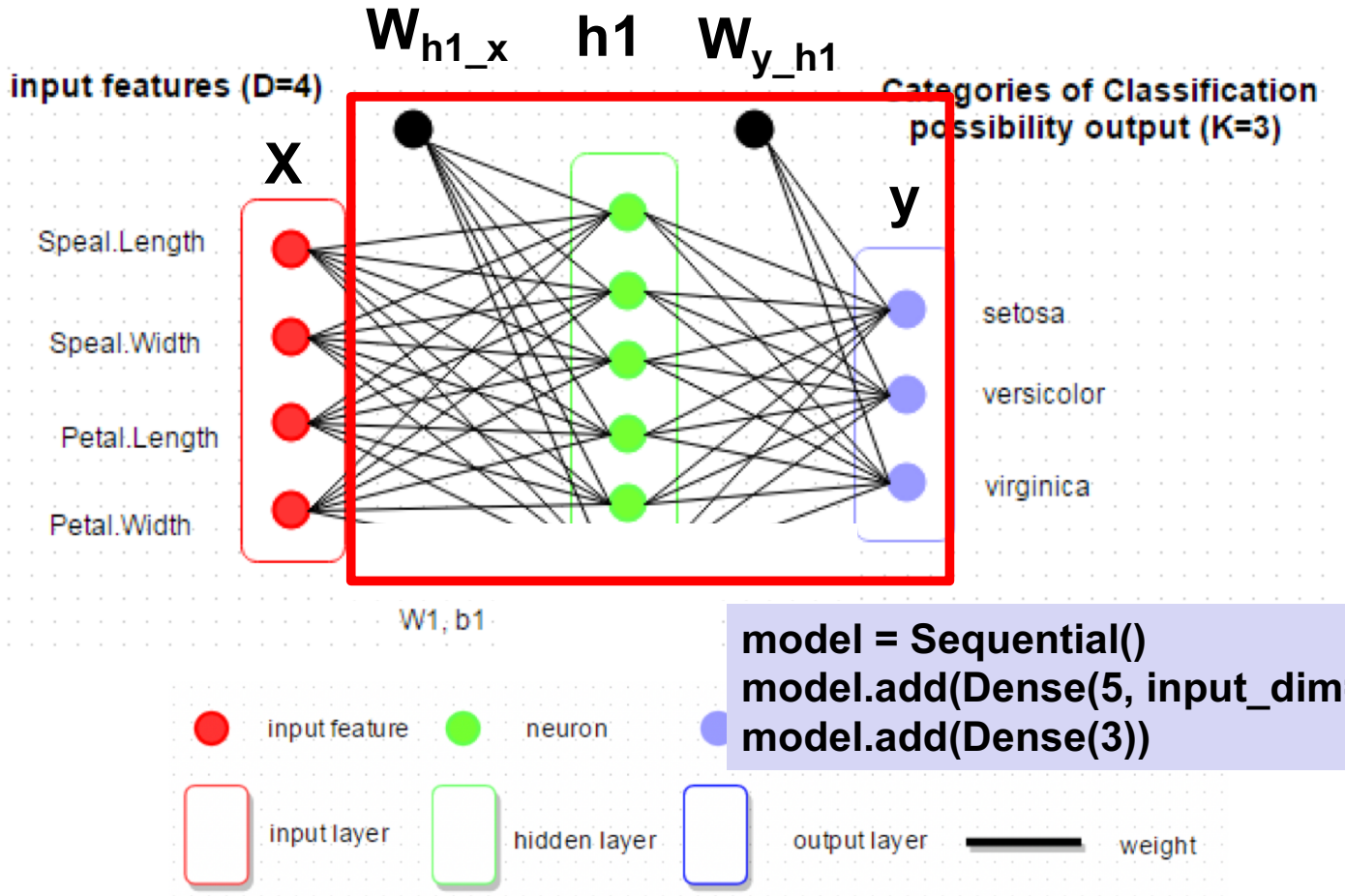
Layer in the activation functions



Activation functions that transform a summed signal from each neuron in a layer can be extracted and added to the Sequential as a layer-like object called Activation.

```
1  model = Sequential()
2  model.add(Dense(5, input_dim=2))
3  model.add(Activation('relu'))
4  model.add(Dense(1))
5  model.add(Activation('sigmoid'))
```

Activaton functions for hidden outer nodes



Activation functions for hidden nodes versus outer nodes

- **Inner node activation functions**
  - Sigmoid, Tanh, ReLU, etc.

- **Outer node activation functions (and loss function objective)**
  - **Regression**: Linear activation function or 'linear' and the number of neurons matching the number of outputs.
  - **Binary Classification** (2 class): Logistic activation function or 's MSE id' and one neuron the output layer.
  - **Multiclass Classification** (>2 class): Softmax activation functi Cross entropy ' and one output neuron per class value, assuming a one-hot encoded output pattern.

Cross entropy

Output layer activation function: Regression versus Classification

- **The choice of activation function is most important for the output layer as it will define the format that predictions will take.**
- **For example, some common predictive modeling problem types and the structure and standard activation function that you can use in the output layer:**
  - **Regression**: Linear activation function or 'linear' and the number of neurons matching the number of outputs.
  - **Binary Classification** (2 class): Logistic activation function or 'sigmoid' and one neuron the output layer.
  - **Multiclass Classification** (>2 class): Softmax activation function or 'softmax' and one output neuron per class value, assuming a one-hot encoded output pattern.

Main Steps in learning a NN

1. **Load data and do EDA**

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

2. **Define the model architecture**

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
```

SoftMax

3. **Compile the model using a numerical backend (and set objective)**

```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

4. **Fit the model**

```
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

5. **Evaluate**

```
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

5 steps to create a NN/deep learning model

1: Load data and do EDA

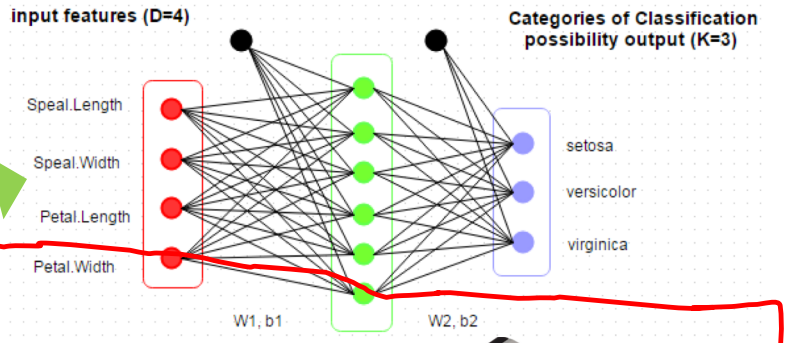
2: Define Network

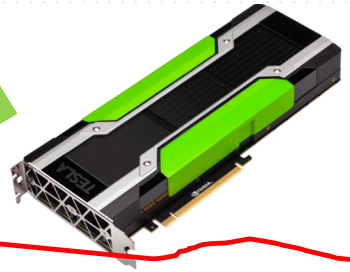
3: Compile Network

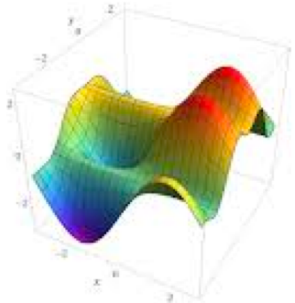
4: Fit Network

5: Evaluate network

6: Make predictions







Step 3: Compile the model

- **Compile the model using a numerical backend (and set objective)**
- **Compilation is an efficiency step.**
  - It transforms the simple sequence of layers that we defined into a highly efficient series of matrix transforms in a format intended to be executed on your GPU or CPU, depending on how Keras is configured.
- **Think of compilation as a precompute step for your network.**

Compilation is done before training

- **Compilation is always required after defining a model**
  1. Compile to fit hardware
  2. Optimization and Loss
- **Compilation is done before training and before transfer learning (CPU/GPU efficiencies)**
  - This includes both before training it using an optimization scheme as well as loading a set of pre-trained weights from a save file.
  - The reason is that the compilation step prepares an efficient representation of the network that is also required to make predictions on your hardware.
- **Compiling requires the optimization algorithm and loss**
  - Specify the optimization algorithm to use to train the network and
  - Specify the loss function used to evaluate the network that is minimized by the optimization algorithm.

Compilation is done before training

- **Compilation is always required after defining a model.**

- **Compilation is done before training and before transfer learning (CPU/GPU efficiencies)**
  - This includes both before training it using an optimization scheme as well as loading a set of pre-trained weights from a save file.
  - The reason is that the compilation step prepares an efficient representation of the network that is also required to make predictions on your hardware.
- **Compiling requires the optimization algorithm and loss**
  - Specify the optimization algorithm to use to train the network and
  - Specify the loss function used to evaluate the network that is minimized by the optimization algorithm

```
model.compile(optimizer='sgd', loss='mse')
```

Loss functions for DL

- **The type of predictive modeling problem imposes constraints on the type of loss function that can be used.**
- **For example, below are some standard loss functions for different predictive model types:**
  - **Regression**: Mean Squared Error or **'mse'.**
  - **Binary Classification (2 class):** Logarithmic Loss, also called cross entropy or **'binary_crossentropy'.**
  - **Multiclass Classification (>2 class):** Multiclass Logarithmic Loss or **'categorical_crossentropy' (CX).**

```
model.compile(optimizer='sgd', loss='mse')
```

Objective Functions (for output layer)

- **Objectives**
  - https://keras.io/objectives

Available objectives

- **mean_squared_error / mse**
- **mean_absolute_error / mae**
- **mean_absolute_percentage_error / mape**
- **mean_squared_logarithmic_error / msle**
- **squared_hinge**
- **hinge**
- **binary_crossentropy**: Also known as logloss.
- **categorical_crossentropy**: Also known as multiclass logloss. **Note**: using this objective requires that your labels are binary arrays of shape `(nb_samples, nb_classes)`.
- **sparse_categorical_crossentropy**: As above but accepts sparse labels. **Note**: this objective still requires that your labels have the same number of dimensions as your outputs; you may need to add a length-1 dimension to the shape of your labels, e.g with `np.expand_dims(y, -1)`.
- **kullback_leibler_divergence / kld**: Information gain from a predicted probability distribution Q to a true probability distribution P. Gives a measure of difference between both distributions.
- **poisson**: Mean of `(predictions - targets * log(predictions))`
- **cosine_proximity**: The opposite (negative) of the mean cosine proximity between predictions and targets.

One hot encode (OHE) multi-class output variable

Note: when using the `categorical_crossentropy` objective, your targets should be in categorical format (e.g. if you have 10 classes, the target for each sample should be a 10-dimensional vector that is all-zeros expect for a 1 at the index corresponding to the class of the sample). In order to convert *integer targets* into *categorical targets*, you can use the Keras utility `to_categorical`:

```
from keras.utils.np_utils import to_categorical

categorical_labels = to_categorical(int_labels, nb_classes=None)
```

Optimization Algorithms and hyperparameters

- Perhaps the most commonly used optimization algorithms because of their generally better performance are:
  - **Stochastic Gradient Descent** or '*sgd*' that requires the tuning of a learning rate
  - **ADAM** or '*adam*' that requires the tuning of learning rate and momentum.
  - **RMSprop** or '*rmsprop*' that requires the tuning of learning rate and momentum.

Track learning by collecting metrics

- Finally, you can also specify metrics to collect while fitting your model in addition to the loss function. Generally, the most useful additional metric to collect is accuracy for classification problems.
- The metrics to collect are specified by name in an array.

```
1    model.compile(optimizer='sgd', loss='mse', metrics=['accuracy'])
```

Main Steps in learning a NN

1. **Load data and do EDA**

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

2. **Define the model architecture**

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
```

SoftMax

3. **Compile the model using a numerical backend (and set objective)**

```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

4. **Fit/Train the model**

```
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

5. **Evaluate**

```
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

5 steps to create a NN/deep learning model

1: Load data and do EDA

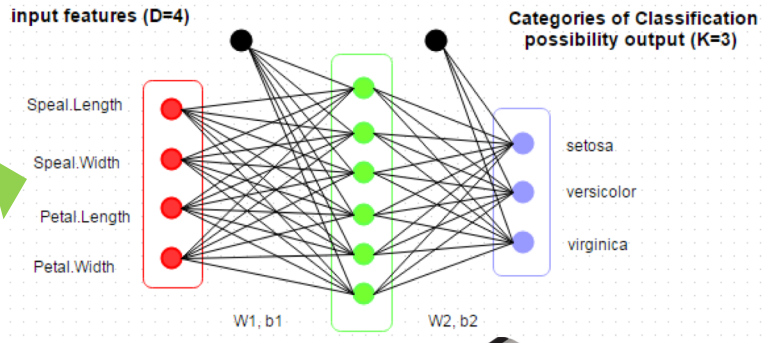
2: Define Network

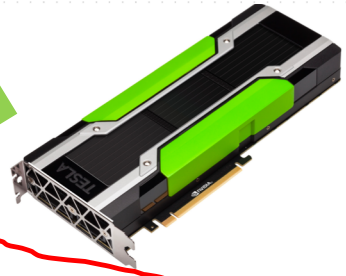
3: Compile Network

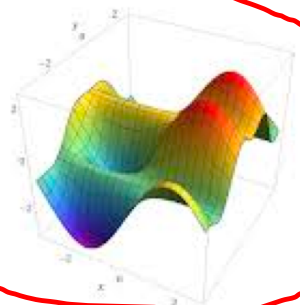
4: Fit Network

5: Evaluate network

6: Make predictions





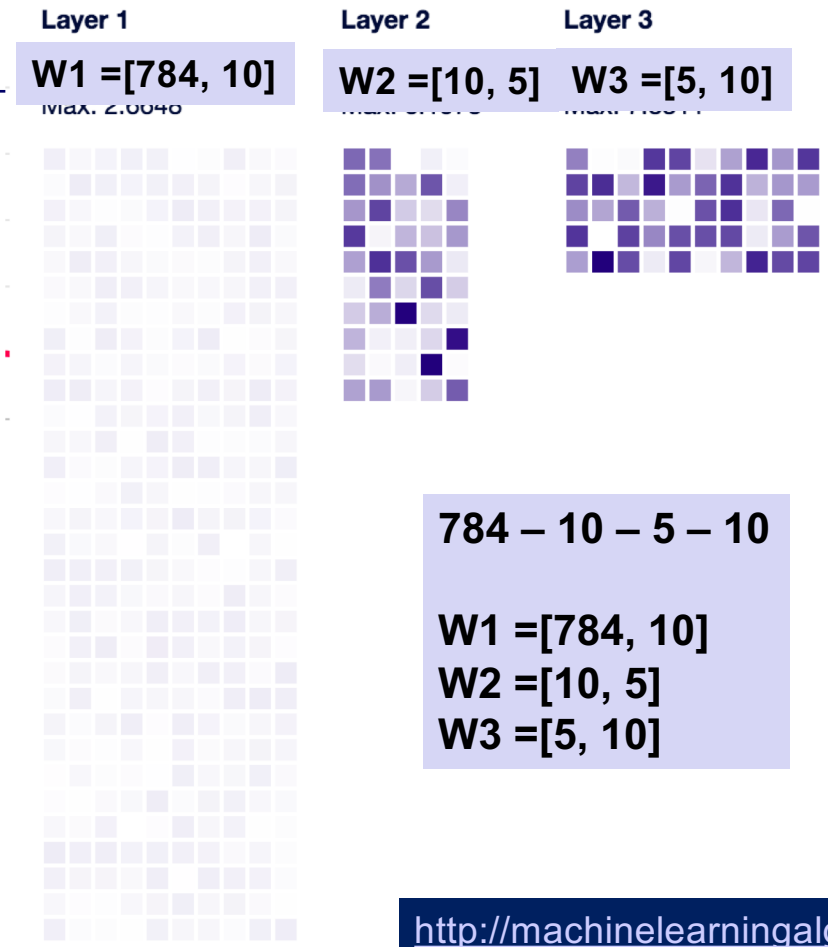


Step 4: Fit the model(AKA Train)

- Once the network is compiled, it can be fit, which means adapt the weights on a training dataset.
- The network is trained using the backpropagation algorithm and optimized according to the optimization algorithm and loss function specified when compiling the model.

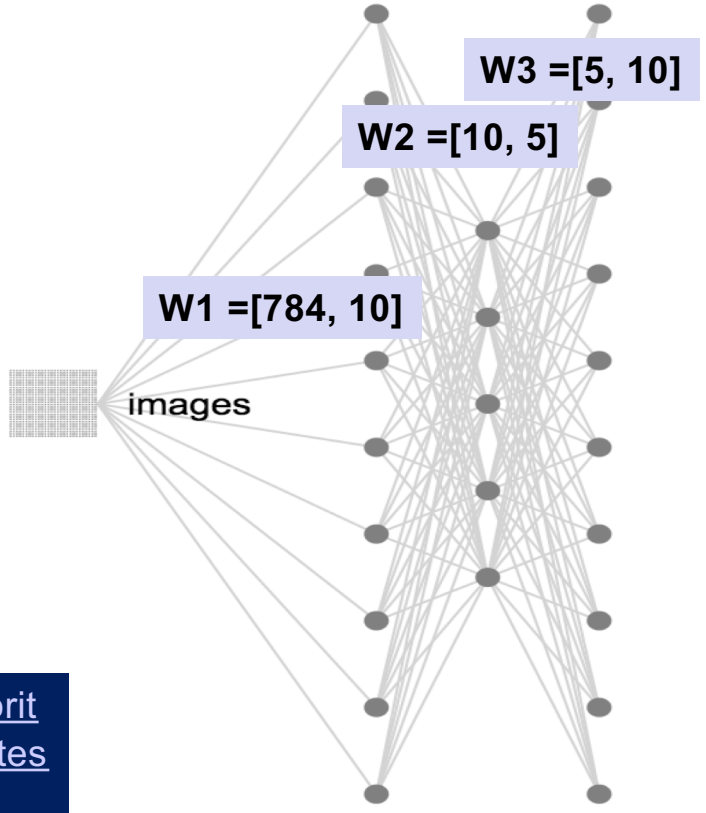
```
1    history = model.fit(X, y, batch_size=10, nb_epoch=100)
```

Weight Matrices for image-based input



784 – 10 – 5 – 10

W1 =[784, 10]
W2 =[10, 5]
W3 =[5, 10]



http://machinelearningalgorithmsillustrated.azurewebsites.net/

Main Steps in learning a NN

1. **Load data and do EDA**

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

2. **Define the model architecture**

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
```

SoftMax

3. **Compile the model using a numerical backend (and set objective)**

```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

4. **Fit/Train the model**

```
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

5. **Evaluate**

```
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

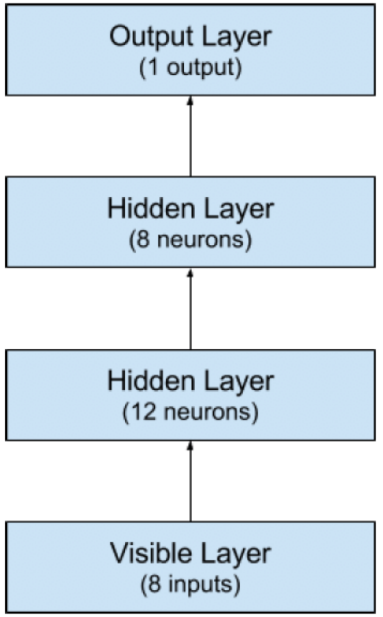
Binary Classification Notebook

- **Pima diabetes: binary classification problem**
  - http://localhost:8890/notebooks/shared/Dropbox/Projects/UniversityOFGhent-2017/Src/Untitled%20Folder/Pima-Diabetes-First-NN.ipynb

- **http://localhost:8892/notebooks/shared/Dropbox/Projects/UniversityOFGhent-2017/Src/01-Pima-Diabetes-First-NN/Pima-Diabetes-First-NN.ipynb**

Pima Diabetes Detector: Binary Classifier

```
1  # Create first network with Keras
2  from keras.models import Sequential
3  from keras.layers import Dense
4  import numpy
5  # fix random seed for reproducibility
6  seed = 7
7  numpy.random.seed(seed)
8  # load pima indians dataset
9  dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
10 # split into input (X) and output (Y) variables
11 X = dataset[:,0:8]
12 Y = dataset[:,8]
13 # create model
14 model = Sequential()
15 model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
16 model.add(Dense(8, init='uniform', activation='relu'))
17 model.add(Dense(1, init='uniform', activation='sigmoid'))
18 # Compile model
19 model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
20 # Fit the model
21 model.fit(X, Y, nb_epoch=150, batch_size=10)
22 # evaluate the model
23 scores = model.evaluate(X, Y)
24 print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```



```
...
Epoch 143/150
768/768 [==============================] - 0s - loss: 0.4614 - acc: 0.7878
Epoch 144/150
768/768 [==============================] - 0s - loss: 0.4508 - acc: 0.7969
Epoch 145/150
768/768 [==============================] - 0s - loss: 0.4580 - acc: 0.7747
Epoch 146/150
768/768 [==============================] - 0s - loss: 0.4627 - acc: 0.7812
Epoch 147/150
768/768 [==============================] - 0s - loss: 0.4531 - acc: 0.7943
Epoch 148/150
768/768 [==============================] - 0s - loss: 0.4656 - acc: 0.7734
Epoch 149/150
768/768 [==============================] - 0s - loss: 0.4566 - acc: 0.7839
Epoch 150/150
768/768 [==============================] - 0s - loss: 0.4593 - acc: 0.7839
768/768 [==============================] - 0s
acc: 79.56%
```

Main Steps in learning a NN

1. **Load data and do EDA**

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

2. **Define the model architecture**

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
```

SoftMax

3. **Compile the model using a numerical backend (and set objective)**

```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

4. **Fit/Train the model**

```
# Fit the model
model.fit(X, Y, nb_epoch=150, batch_size=10)
```

5. **Evaluate**

```
# evaluate the model
scores = model.evaluate(X, Y)
print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Vanilla logistic regression

4-3 for iris data

have 3 output units.

```
]: inp = Input(shape=(4,))
   out = Dense(3, activation="softmax")(inp)
```

Multinomial Logistic Regression uses softmax activation in multiclass case instead of binary logit function. Also we use cı
as a loss function.

```
]: model = Model(inputs=[inp], outputs=[out])
   model.compile(loss="categorical_crossentropy", optimizer="sgd", metrics=["accuracy"])
```

Split into train and test set

Python For Data Science *Cheat Sheet*

Keras

Learn Python for data science Interactively at www.DataCamp.com

Keras

Keras is a powerful and easy-to-use deep learning library for Theano and TensorFlow that provides a high-level neural networks API to develop and evaluate deep learning models.

A Basic Example

```
>>> import numpy as np
>>> from keras.models import Sequential
>>> from keras.layers import Dense
>>> data = np.random.random((1000,100))
>>> labels = np.random.randint(2,size=(1000,1))
>>> model = Sequential()
>>> model.add(Dense(32,
                    activation='relu',
                    input_dim=100))
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
>>> model.fit(data,labels,epochs=10,batch_size=32)
>>> predictions = model.predict(data)
```

Data

Also see NumPy, Pandas & Scikit-Learn

Your data needs to be stored as NumPy arrays or as a list of NumPy arrays. Ideally, you split the data in training and test sets, for which you can also resort to the `train_test_split` module of `sklearn.cross_validation`.

Keras Data Sets

```
>>> from keras.datasets import boston_housing,
                               mnist,
                               cifar10,
                               imdb

>>> (x_train,y_train),(x_test,y_test) = mnist.load_data()
>>> (x_train2,y_train2),(x_test2,y_test2) = boston_housing.load_data()
>>> (x_train3,y_train3),(x_test3,y_test3) = cifar10.load_data()
>>> (x_train4,y_train4),(x_test4,y_test4) = imdb.load_data(num_words=20000)
>>> num_classes = 10
```

Other

```
>>> from urllib.request import urlopen
>>> data = np.loadtxt(urlopen("http://archive.ics.uci.edu/
ml/machine-learning-databases/pima-indians-diabetes/
pima-indians-diabetes.data"),delimiter=",")
>>> X = data[:,0:8]
>>> y = data [:,8]
```

Model Architecture

Sequential Model

```
>>> from keras.models import Sequential
>>> model = Sequential()
>>> model2 = Sequential()
>>> model3 = Sequential()
```

Multilayer Perceptron (MLP)

Binary Classification

```
>>> from keras.layers import Dense
>>> model.add(Dense(12,
                    input_dim=8,
                    kernel_initializer='uniform',
                    activation='relu'))
>>> model.add(Dense(8,kernel_initializer='uniform',activation='relu'))
>>> model.add(Dense(1,kernel_initializer='uniform',activation='sigmoid'))
```

Multi-Class Classification

```
>>> from keras.layers import Dropout
>>> model.add(Dense(512,activation='relu',input_shape=(784,)))
>>> model.add(Dropout(0.2))
>>> model.add(Dense(512,activation='relu'))
>>> model.add(Dropout(0.2))
>>> model.add(Dense(10,activation='softmax'))
```

Regression

```
>>> model.add(Dense(64,activation='relu',input_dim=train_data.shape[1]))
>>> model.add(Dense(1))
```

Convolutional Neural Network (CNN)

```
>>> from keras.layers import Activation,Conv2D,MaxPooling2D,Flatten
>>> model2.add(Conv2D(32,(3,3),padding='same',input_shape=x_train.shape[1:]))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(32,(3,3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))

>>> model2.add(Conv2D(64,(3,3), padding='same'))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(64,(3, 3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))

>>> model2.add(Flatten())
>>> model2.add(Dense(512))
>>> model2.add(Activation('relu'))
>>> model2.add(Dropout(0.5))
>>> model2.add(Dense(num_classes))
>>> model2.add(Activation('softmax'))
```

Recurrent Neural Network (RNN)

```
>>> from keras.klayers import Embedding,LSTM
>>> model3.add(Embedding(20000,128))
>>> model3.add(LSTM(128,dropout=0.2,recurrent_dropout=0.2))
>>> model3.add(Dense(1,activation='sigmoid'))
```

Inspect Model

`>>> model.output_shape`	Model output shape
`>>> model.summary()`	Model summary representation
`>>> model.get_config()`	Model configuration
`>>> model.get_weights()`	List all weight tensors in the model

Compile Model

MLP: Binary Classification

```
>>> model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
```

MLP: Multi-Class Classification

```
>>> model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

MLP: Regression

```
>>> model.compile(optimizer='rmsprop',
                  loss='mse',
                  metrics=['mae'])
```

Recurrent Neural Network

```
>>> model3.compile(loss='binary_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
```

Model Training

```
>>> model3.fit(x_train4,
               y_train4,
               batch_size=32,
               epochs=15,
               verbose=1,
               validation_data=(x_test4,y_test4))
```

Evaluate Your Model's Performance

```
>>> score = model3.evaluate(x_test,
                            y_test,
                            batch_size=32)
```

Prediction

```
>>> model3.predict(x_test4, batch_size=32)
>>> model3.predict_classes(x_test4,batch_size=32)
```

Save/ Reload Models

```
>>> from keras.models import load_model
>>> model3.save('model_file.h5')
>>> my_model = load_model('my_model.h5')
```

Model Fine-tuning

Optimization Parameters

```
>>> from keras.optimizers import RMSprop
>>> opt = RMSprop(lr=0.0001, decay=1e-6)
>>> model2.compile(loss='categorical_crossentropy',
                   optimizer=opt,
                   metrics=['accuracy'])
```

Early Stopping

```
>>> from keras.callbacks import EarlyStopping
>>> early_stopping_monitor = EarlyStopping(patience=2)
>>> model3.fit(x_train4,
               y_train4,
               batch_size=32,
               epochs=15,
               validation_data=(x_test4,y_test4),
               callbacks=[early_stopping_monitor])
```

Preprocessing

Also see NumPy & Scikit-Learn

Sequence Padding

```
>>> from keras.preprocessing import sequence
>>> x_train4 = sequence.pad_sequences(x_train4,maxlen=80)
>>> x_test4 = sequence.pad_sequences(x_test4,maxlen=80)
```

One-Hot Encoding

```
>>> from keras.utils import to_categorical
>>> Y_train = to_categorical(y_train, num_classes)
>>> Y_test = to_categorical(y_test, num_classes)
>>> Y_train3 = to_categorical(y_train3, num_classes)
>>> Y_test3 = to_categorical(y_test3, num_classes)
```

Train and Test Sets

```
>>> from sklearn.model_selection import train_test_split
>>> X_train5,X_test5,y_train5,y_test5 = train_test_split(X,
                                                         y,
                                                         test_size=0.33,
                                                         random_state=42)
```

Standardization/Normalization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(x_train2)
>>> standardized_X = scaler.transform(x_train2)
>>> standardized_X_test = scaler.transform(x_test2)
```

DataCamp
Learn Python for Data Science Interactively

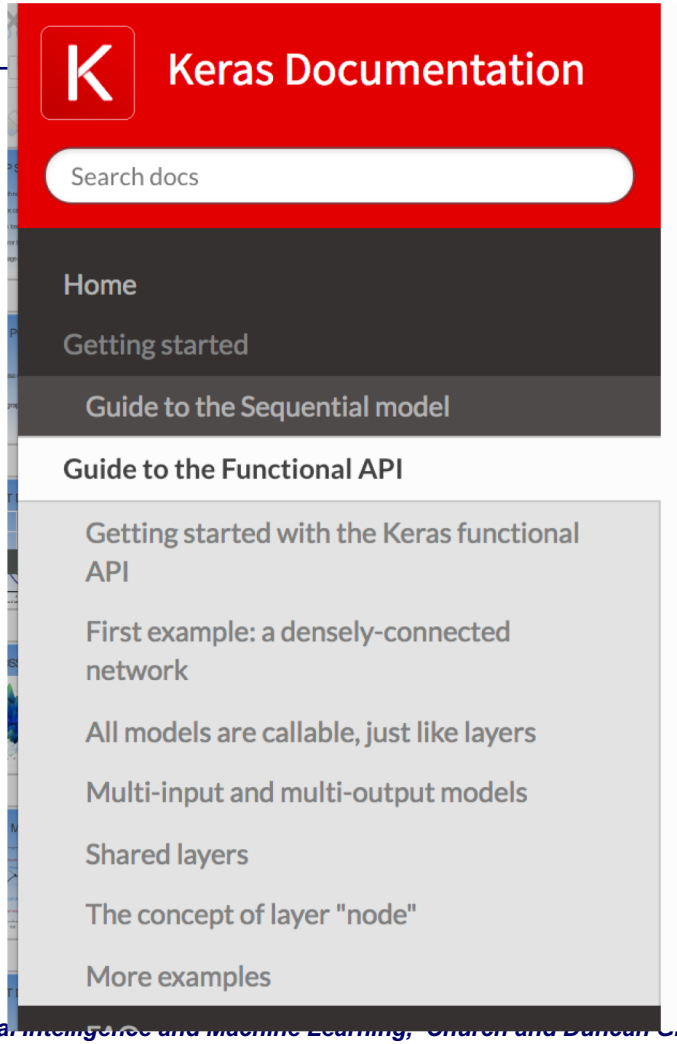
Convert Networks from TF → MXNet/etc.

- **MMdnn**
- **https://github.com/Microsoft/MMdnn**

- **Lab: Sequential API versus Functional/Model API**

Regression: Predict the price of houses in Boston

Programming Deep Neural Networks in Keras or TensorFlow



- **Sequential programming**
  - The Sequential model is a linear stack of layers.
- **Functional programming**
  - The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

https://keras.io/getting-started/sequential-model-guide/

Sequential definition of a model

4 Train a Neural Network Model (8-12-8-1)

- Train a 8-12-8-1 network
- Extract probabiolities

```
## Train a Neural Network Model (8-12-8-1)
# Sample Multilayer Perceptron Neural Network in Keras
from keras.models import Sequential
from keras.layers import Dense
import numpy

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.30, random_state=42)

# 1. define the network
model = Sequential()
model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))
model.add(Dense(8, init='uniform', activation='relu'))
model.add(Dense(1, init='uniform', activation='sigmoid'))
# 2. compile the network
#time it
%time model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Keras functional API

First example: a densely-connected network

The `Sequential` model is probably a better choice to implement such a network, but it helps to start with something really simple.

784 – 64 – 64 – 10

- A layer instance is callable (on a tensor), and it returns a tensor

x = Dense(64, activation =“relu”)(inputs)

1. **Creates layer Dense(64, activation =“relu”) creates a layer**
2. **Calls the callable method on this Dense layer Dense(64, activation =“relu”)(inputs)**
   Connect the layer inputs to to the Dense 64 layer and returns Dense(64)

```
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

Reuse model architecture and weights

- **All models are callable, just like layers**
- With the functional API, it is easy to re-use trained models: you can treat any model as if it were a layer, by calling it on a tensor.
- Note that by calling a model you aren't just re-using the ***architecture*** of the model, you are also re-using its **weights**.

```
x = Input(shape=(784,))
# This works, and returns the 10-way softmax we defined above.
y = model(x)
```

Callable object

- A **callable object**, in computer programming, is any object that can be called like a function.

In Python any object with a `__call__()` method can be called using function-call syntax.

```
# Python 3 Syntax
class Foo(object):
    def __call__(self):
        print("Called.")

foo_instance = Foo()
foo_instance()  # This will output "Called." to the screen.
```

[2]

Another example:

```
class Accumulator(object):
    def __init__(self, n):
        self.n = n

    def __call__(self, x):
        self.n += x
        return self.n
```

Callable object

- A **callable object**, in computer programming, is any object that can be called like a function.

`__call__` makes any object be callable as a function.

This example will output 8:

```
class Adder(object):
  def __init__(self, val):
    self.val = val

  def __call__(self, val):
    return self.val + val

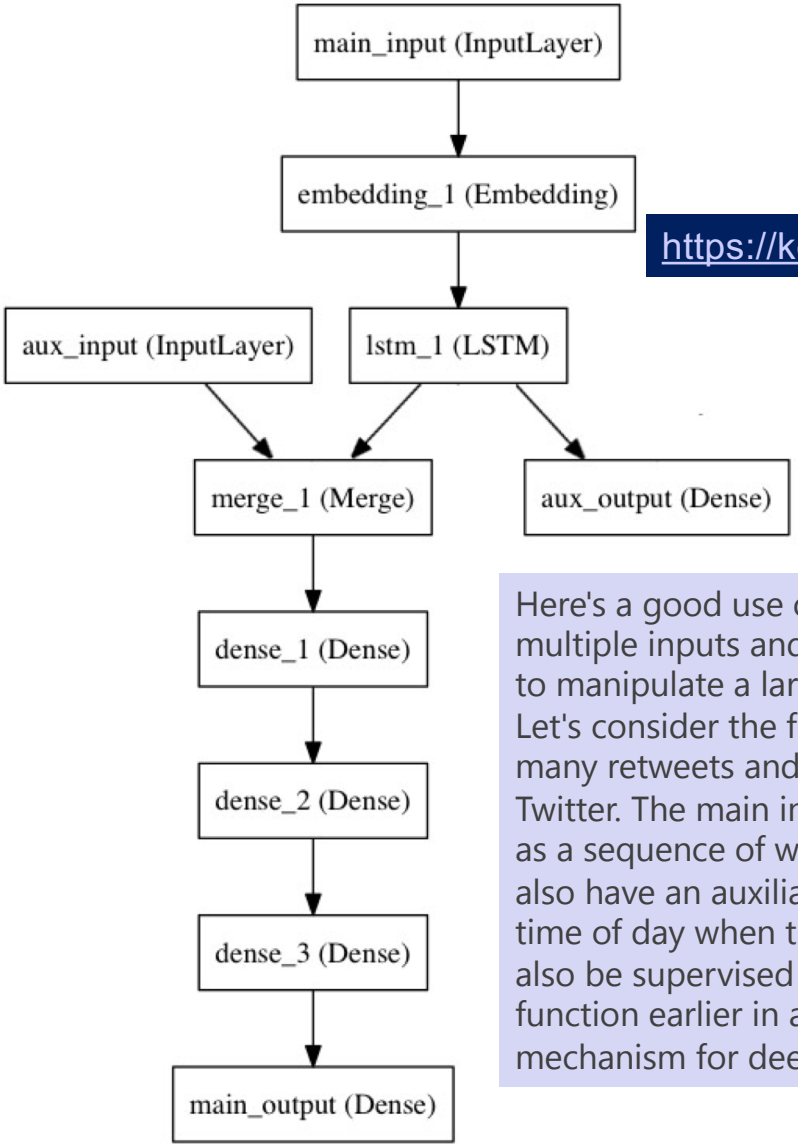
func = Adder(5)
print func(3)
```

Dense Layer

- https://github.com/keras-team/keras/blob/master/keras/layers/core.py

```
878         def call(self, inputs):
879             output = K.dot(inputs, self.kernel)
880             if self.use_bias:
881                 output = K.bias_add(output, self.bias, data_format='channels_last')
882             if self.activation is not None:
883                 output = self.activation(output)
884             return output
885
```

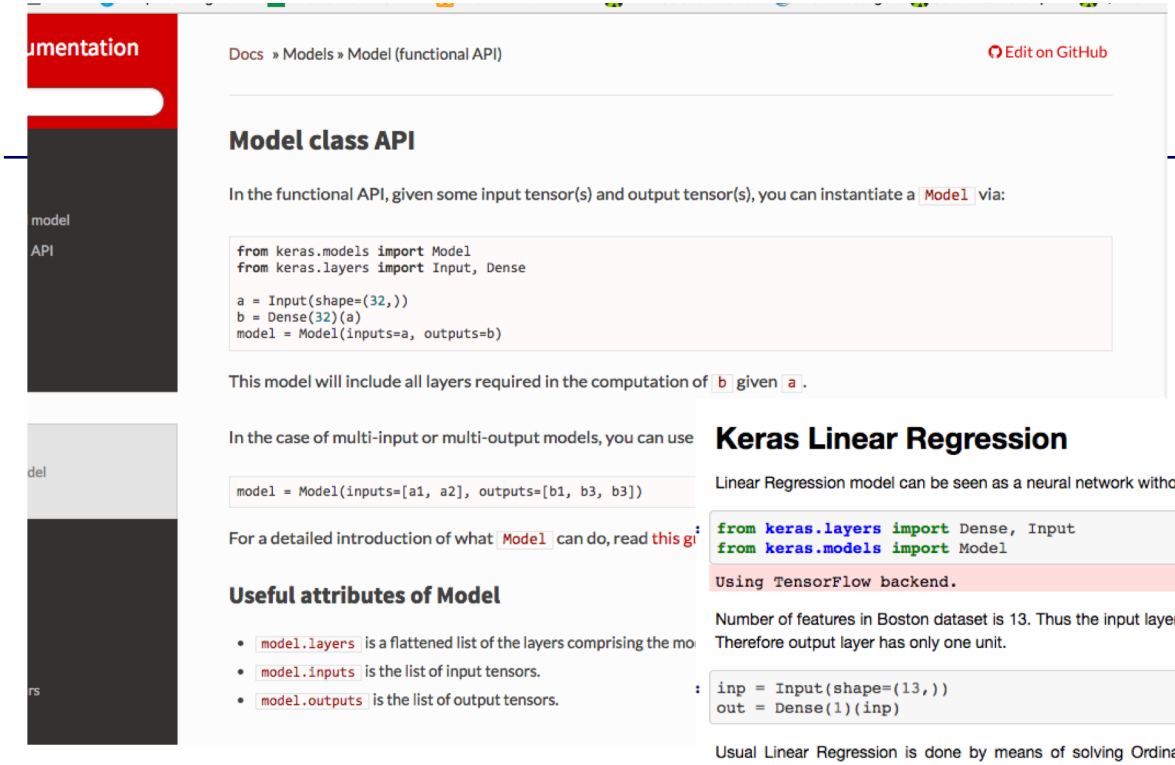
Multi-input and multi-output models



https://keras.io/getting-started/functional-api-guide/

Here's a good use case for the functional API: models with multiple inputs and outputs. The functional API makes it easy to manipulate a large number of intertwined datastreams. Let's consider the following model. We seek to predict how many retweets and likes a news headline will receive on Twitter. The main input to the model will be the headline itself, as a sequence of words, but to spice things up, our model will also have an auxiliary input, receiving extra data such as the time of day when the headline was posted, etc. The model will also be supervised via two loss functions. Using the main loss function earlier in a model is a good regularization mechanism for deep models.

Model Class API



umentation

model

API

del

rs

Docs » Models » Model (functional API)

Edit on GitHub

Model class API

In the functional API, given some input tensor(s) and output tensor(s), you can instantiate a `Model` via:

```
from keras.models import Model
from keras.layers import Input, Dense

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(inputs=a, outputs=b)
```

This model will include all layers required in the computation of `b` given `a`.

In the case of multi-input or multi-output models, you can use

```
model = Model(inputs=[a1, a2], outputs=[b1, b3, b3])
```

For a detailed introduction of what `Model` can do, read this gi

Useful attributes of Model

- `model.layers` is a flattened list of the layers comprising the mo
- `model.inputs` is the list of input tensors.
- `model.outputs` is the list of output tensors.

Keras Linear Regression

Linear Regression model can be seen as a neural network withour hidden layer and any activation function

```
from keras.layers import Dense, Input
from keras.models import Model
```

```
Using TensorFlow backend.
```

Number of features in Boston dataset is 13. Thus the input layer has 13 units. We want to predict only one value - price of the house. Therefore output layer has only one unit.

```
inp = Input(shape=(13,))
out = Dense(1)(inp)
```

Usual Linear Regression is done by means of solving Ordinary Least Square Error problem which means that one should one squared (mse) loss.

Here we also choose the simplest optimizer - usual stohastic gradient descent.

```
model_keras = Model(inputs=[inp], outputs=[out])
model_keras.compile(loss="mse", optimizer="sgd", metrics=["mae"])
```

Split into train and test set (with the same random_state which means we can compare results)

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Scaling

```
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

__call__ method for an object: default execution is to execution

- **Used in SciKit Learn**
- **In Keras to connect layers into a network**
- **myInstance(var1, var2)**
  - Will result in the method call being executed

__call__

```
839     def call(self, inputs):
840         output = K.dot(inputs, self.kernel)
841         if self.use_bias:
842             output = K.bias_add(output, self.bias)
843         if self.activation is not None:
844             output = self.activation(output)
845         return output
846
```

Keras Linear Regression

Linear Regression model can be seen as a neural network withour hidden layer and any activation function

In [31]:
```
from keras.layers import Dense, Input
from keras.models import Model
```

```
Using TensorFlow backend.
```

Number of features in Boston dataset is 13. Thus the input layer has 13 units. We want to predict only one value - price of the house. Therefore output layer has only one unit.

In [32]:
```
inp = Input(shape=(13,))
out = Dense(1)(inp)
```

inp is callable

Usual Linear Regression is done by means of solving Ordinary Least Square Error problem which means that one should one squared (mse) loss.

Here we also choose the simplest optimizer - usual stohastic gradient descent.

In [33]:
```
model_keras = Model(inputs=[inp], outputs=[out])
model_keras.compile(loss="mse", optimizer="sgd", metrics=["mae"])
```

Split into train and test set (with the same random_state which means we can compare results)

In [34]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Scaling

In [35]:
```
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Scale data

Sequential assumes a single input and a single output and single path thru graph
However, using Keras Model/functional API, we can build sophisticated graphs with multiple inputs and outputs

Fitting

In [36]:
```
hist = model_keras.fit(X_train, y_train,
                       validation_data=(X_test, y_test),
                       epochs=300, verbose=0)
```

```
/opt/conda/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:2252: UserWarning: Expe
cted no kwargs, you passed 1
kwargs passed to function are ignored with Tensorflow backend
  warnings.warn('\n'.join(msg))
```

```
created by the layer, and `bias` is a bias vector created by the layer
(only applicable if `use_bias` is `True`).

Note: if the input to the layer has a rank greater than 2, then
it is flattened prior to the initial dot product with `kernel`.

# Example

```python
    # as first layer in a sequential model:
    model = Sequential()
    model.add(Dense(32, input_shape=(16,)))
    # now the model will take as input arrays of shape (*, 16)
    # and output arrays of shape (*, 32)

    # after the first layer, you don't need to specify
    # the size of the input anymore:
    model.add(Dense(32))
```
```

https://github.com/fchollet/keras/blob/mas

Learning for... | Twelve Month Cale... | Dashboards - All D... | Data Science Home... | AT&T Messages | Senior Leadership... | Q3 2014 Priority U

```
        self.input_spec = InputSpec(min_ndim=2, axes={-1: input_dim})
        self.built = True

    def call(self, inputs):
        output = K.dot(inputs, self.kernel)
        if self.use_bias:
            output = K.bias_add(output, self.bias)
        if self.activation is not None:
            output = self.activation(output)
        return output

    def compute_output_shape(self, input_shape):
        assert input_shape and len(input_shape) >= 2
        assert input_shape[-1]
        output_shape = list(input_shape)
        output_shape[-1] = self.units
        return tuple(output_shape)

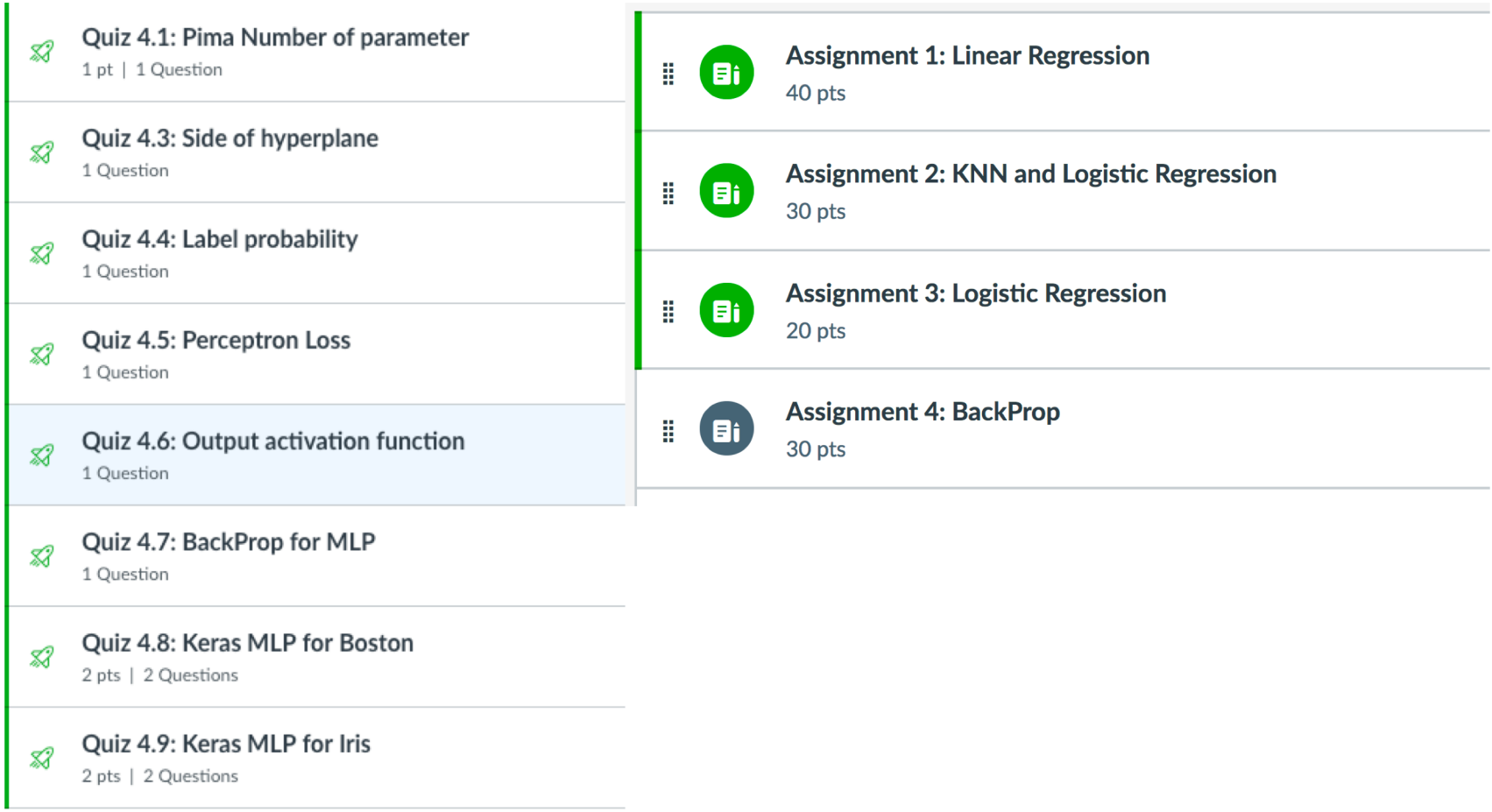
    def get_config(self):
```

- **Model functional API versus sequential**

- https://www.quora.com/Why-would-one-use-the-special-method-__call__-instead-of-defining-a-new-method-for-an-object-in-Python

Lab: Iris: Three class classification problem

Quizzes and HW Assignments



Quiz 4.9

Quiz Instructions

Run the MLPIris and do the task in the end of the notebook.

Question 1

What is the test Accuracy for MLP with the described architecture?

Report an answer with **2 decimal places (using np.round()).**

Question 2

How many parameters does the described model have?

Quiz: Bonus exercise

- **Implement vanilla logistic regression in Keras (use the code from you 4-20-3 network as a basis)**

Vanilla logistic regression

4-3 for iris data

have 3 output units.

```
]: inp = Input(shape=(4,))
   out = Dense(3, activation="softmax")(inp)
```

Multinomial Logistic Regression uses softmax activation in multiclass case instead of binary logit function. Also we use cı
as a loss function.

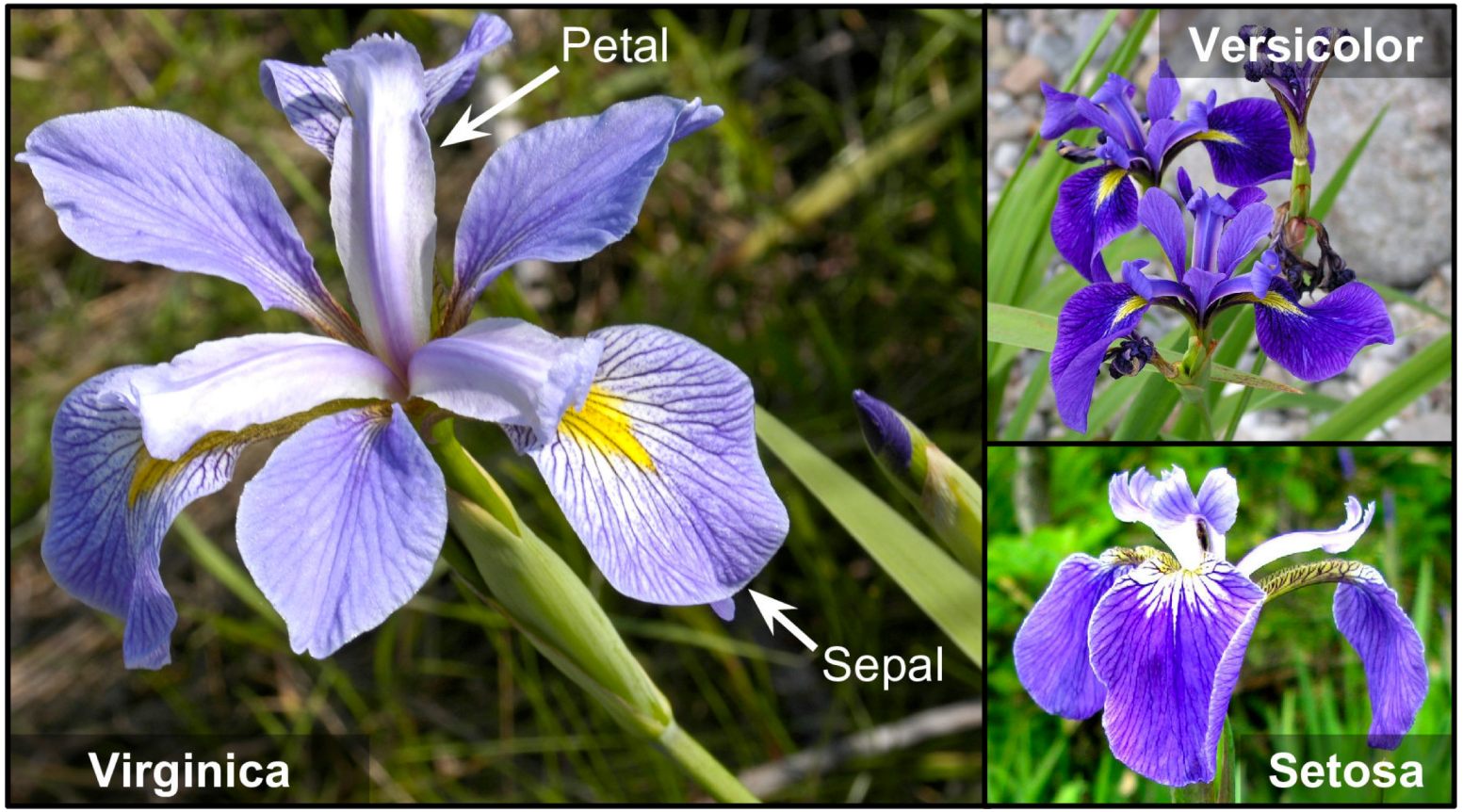
```
]: model = Model(inputs=[inp], outputs=[out])
   model.compile(loss="categorical_crossentropy", optimizer="sgd", metrics=["accuracy"])
```

Split into train and test set

Labs

- **Here are the links to the notebooks for in-class work:**
  - Pima
  - Boston
  - Iris

A sepal is a part of the flower of angiosperms (flowering plants). Usually green, sepals typically function as protection for the flower in bud, and often as support for the petals when in bloom



Quiz: IrisTask

Keras MLP (TASK)

Iris dataset has 4 features per sample. Thus we have 4 units in the input layer. Our problem has 3 classes and therefore we should have 3 output units.

Now let's add one more hidden layer with 20 unit between input and output and $\tanh$ activation function.

The syntax is the following (Keras API):

<name> = <Layer>(<args>)(<name of layer to connect to>)

In [ ]:
```
inp = Input(shape=(4,))
# add Dense layer called "hidden" with 20 neurons and "relu" activation and connect it to the "in
p" layer
out = Dense(3, activation="softmax")(hidden)
```

MLP

In [ ]:
```
model = Model(inputs=[inp], outputs=[out])
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
```

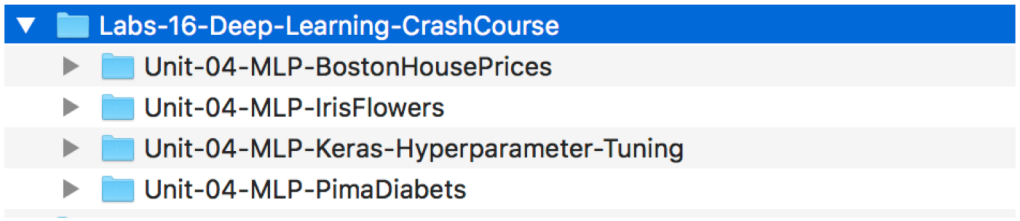
Model description

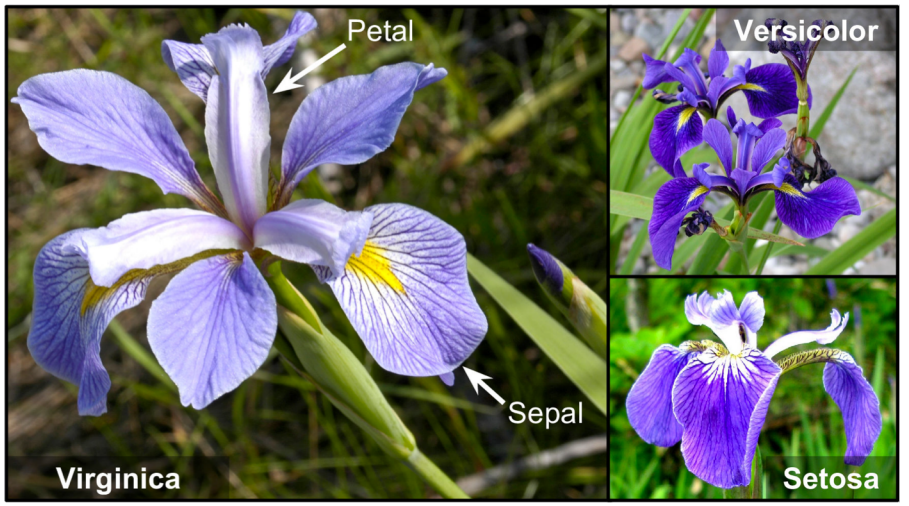
In [ ]:
```
model.summary()
```

Notebooks

- **Pima diabetes: binary classification problem**
  - GitHub: /Labs/Unit-16
    - Src/Dev/Labs/Unit-16

- **QUIZ: Iris Notebook Colab**
  - Iris: Three class classification problem
    - https://drive.google.com/file/d/1AcVGV64AU0-gFM8qW0t0Zy5JspTQi1vk/view?usp=sharing





Lab: Regression: Predict the price of houses in Boston

Regression: Predict the price of houses in Boston

- The problem that we will look at in this tutorial is the Boston house price dataset. *Shape [506, 14]*
- The dataset describes 13 numerical properties of houses in Boston suburbs and is concerned with modeling the price of houses in those suburbs in thousands of dollars.
- Input attributes include things like crime rate, proportion of nonretail business acres, chemical concentrations and more.

Quiz 4.8 Boston MLP regression

Quiz Instructions

Run t MLPBoston and do the task in the end of the notebook.

Question 1

What is the test Accuracy for MLP with the described architecture?

Report an answer with **2 decimal places (using np.round()).**

Question 2

How many parameters does the described model have?

MLP Regression Model 13-13Relu-1

- **Boston dataset**

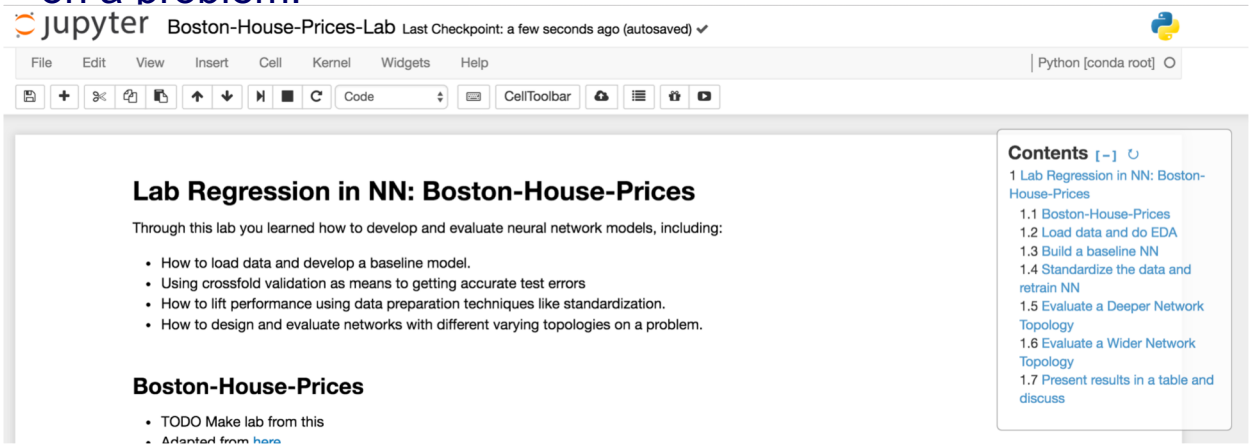
```
# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_dim=13, kernel_initializer='normal', activation='relu
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

MSE for Boston Housing dataset

- Reasonable performance for models evaluated using Mean Squared Error (MSE) are around 20 in squared thousands of dollars
- Or a mean absolute error of $4,500 if you take the square root of MSE (not totally accurate).
- This is a nice target to aim for with our neural network model.

In class Lab: NN for regression

- Through this lab you learned how to develop and evaluate neural network models, including:
  - How to load data and develop a baseline model.
  - Using crossfold validation as means to getting accurate test errors
  - How to lift performance using data preparation techniques like standardization.
  - How to design and evaluate networks with different varying topologies on a problem.



Quiz: Boston Task

Keras MLP (TASK)

Number of features in Boston dataset is 13. Thus the input layer has 13 units. We want to predict only one value - price of the house. Therefore output layer has only one unit.

Now let's add one more hidden layer with 20 unit between input and output and $\tanh$ activation function.

The syntax is the following (Keras API):

$\text{

```
inp = Input(shape=(13,))
# add Dense layer called "hidden" with 20 neurons and "relu" activation and connect it to the "in
p" layer
out = Dense(1, activation="linear")(hidden)
```

MLP

```
model_mlp = Model(inputs=[inp], outputs=[out])
model_mlp.compile(loss="mse", optimizer="sgd", metrics=["mae"])
```

Model description

Quiz: Boston housing in linear regression

- **Add home linear regression model in here**

- **https://github.com/cadgip/DL_course/blob/master/Basics/Regression/LinRegrBoston.ipynb**

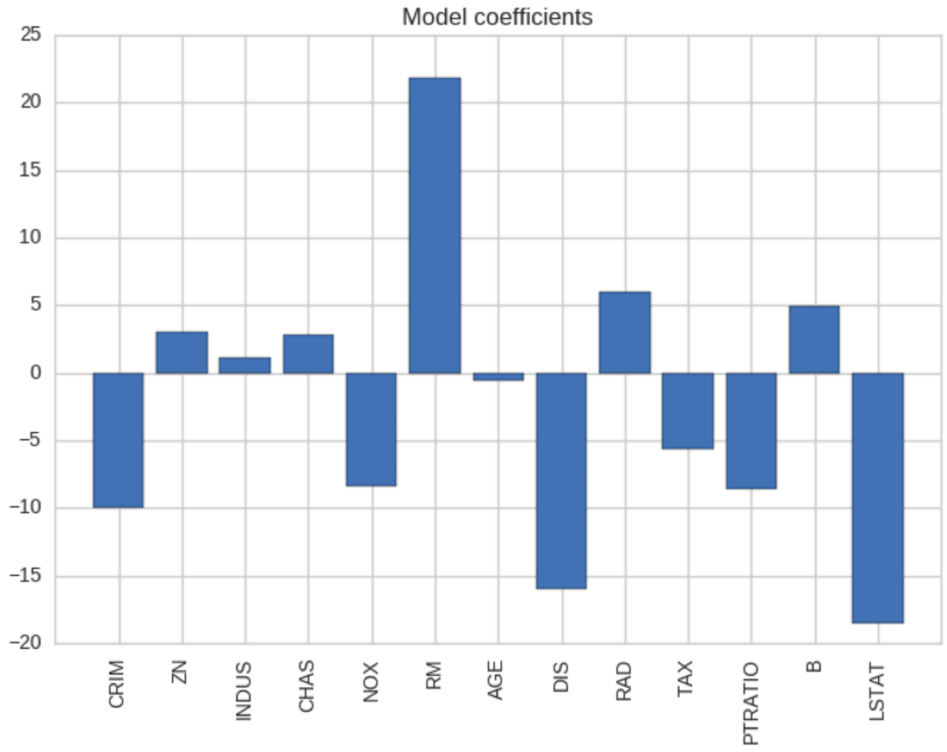
```
In [15]: model.fit(X_train, y_train)
```

```
Out[15]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Evaluation

Let's see what features are significant for the model

```
In [16]: plt.bar(np.arange(model.coef_.shape[0]) - 0.4, model.coef_)
         plt.xticks(np.arange(model.coef_.shape[0]), X.columns, rotation='vertical')
         plt.xlim([-1, model.coef_.shape[0]])
         plt.title("Model coefficients")
         plt.show()
```



KerasClassifier class

- **Keras in pipeline fashion**
- **Uses the** KerasClassifier class
  - Iris data example
    - https://machin
      classification-t

Wrappers for the Scikit-Learn API

You can use `Sequential` Keras models (single-input only) as part of your Scikit-Learn workflow via the wrappers found at `keras.wrappers.scikit_learn.py`.

There are two wrappers available:

`keras.wrappers.scikit_learn.KerasClassifier(build_fn=None, **sk_params)`, which implements the Scikit-Learn classifier interface,

`keras.wrappers.scikit_learn.KerasRegressor(build_fn=None, **sk_params)`, which implements the Scikit-Learn regressor interface.

Arguments

- **build_fn**: callable function or class instance
- **sk_params**: model parameters & fitting parameters

`build_fn` should construct, compile and return a Keras model, which will then be used to fit/predict. One of the following three values could be passed to `build_fn`:

The network topology of this simple one-layer neural network can be summarized as:

```
4 inputs -> [8 hidden nodes] -> 3 outputs
```

Note that we use a "*softmax*" activation function in the output layer. This is to ensure the output values are in the range of 0 and 1 and may be used as predicted probabilities.

Finally, the network uses the efficient Adam gradient descent optimization algorithm with a logarithmic loss function, which is called "*categorical_crossentropy*" in Keras.

```
# define baseline model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(8, input_dim=4, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accu
    return model
```

We can now create our KerasClassifier for use in scikit-learn.

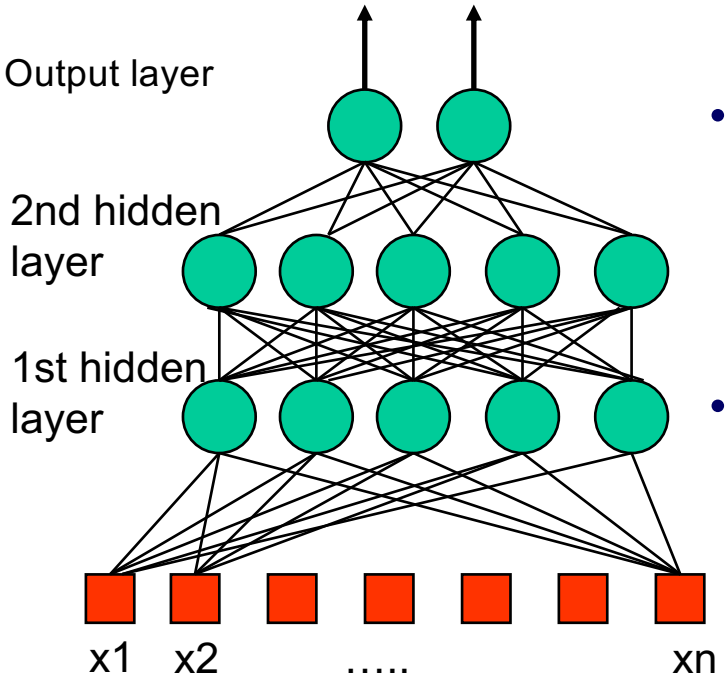
We can also pass arguments in the construction of the KerasClassifier class that will be passed on to the fit() function internally used to train the neural network. Here, we pass the number of epochs as 200 and batch size as 5 to use when training the model. Debugging is also turned off when training by setting verbose to 0.

```
estimator = KerasClassifier(build_fn=baseline_model, epochs=200, batch_size=5, verbose=0)
```

Outline

1. **Introduction**
2. **ML and deep learning review**
    1. Perceptron → MLP
    2. Keras
    3. BackPropagation
3. **What is computer vision?**
    1. Computer vision
    2. Convolutional Neural Nets (CNNs)
4. **Deep NN Architectures for CV Tasks**
    1. Backbone networks
5. **Solving edge-based IoT**
6. **Conclusions and Next steps**

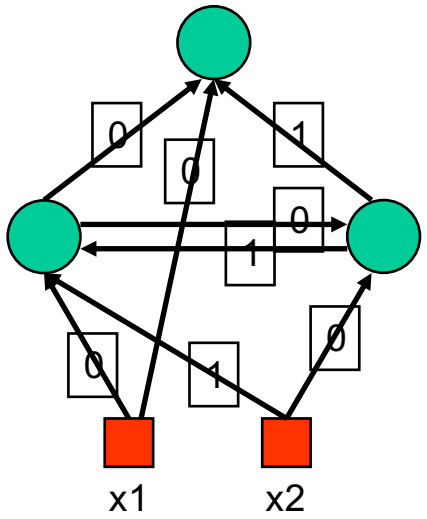
Feed Forward Neural Networks



- The information is propagated from the inputs to the outputs
- Computations of No non-linear functions from n input variables by compositions of Nc algebraic functions
- Time has no role (NO cycle between outputs and inputs)

Recurrent Neural Networks

- **Can have arbitrary topologies**
- **Can model systems with internal states (dynamic ones)**
- **Delays are associated to a specific weight**
- **Training is more difficult**
- **Performance may be problematic**
  - Stable Outputs may be more difficult to evaluate
  - Unexpected behavior (oscillation, chaos, …)



MLP Regression Model: 13-13Relu-1

- **Boston dataset**

```
# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_dim=13, kernel_initializer='normal', activation='relu
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

$MSE(X; MLP_{Regression}) = \frac{1}{2n}(I(W_2\ (\sigma(W_1\ X))) - t_k)^2$ where I is identity

Input. W1 Hidden W2 output

MLP Softmax Model

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)\mathbf{x}^{(i)}$$

MSE(X; $MLP_{Regression}$) = $\frac{1}{2n}(I(W_2\ (\sigma(W_1\ X)))\ - t_k)^2$ where I is identity

CrossEntropy(X; $MLP_{Regression}$) = $\frac{1}{2n}(\sigma(W_2\ (\sigma(W_1\ X)))_k\ - t_k)^2$.

where $\sigma(.)_k$ is defined as follows:

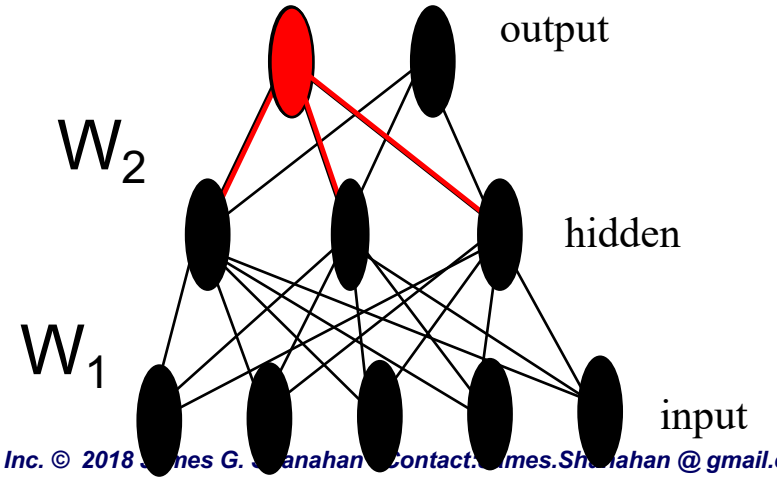
$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K}\exp\left(s_j(\mathbf{x})\right)}$$

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k given the scores of each class for that instance.

General (two-layer) feedforward network

$$MLP(X) = g(f(XW_1) W_2)$$

- The hidden units with their activation functions can express non-linear functions
- The activation functions can be different at neurons (but the same one is used in practice)



Backpropagation Algorithm

- **Initialize weights (typically random!)**
- **Keep doing epochs**
  - For each example e in training set do
    - **forward pass** to compute
      - O = neural-net-output(network,e)
      - miss = (T-O) at each output unit
    - **backward pass** to calculate deltas to weights
    - update all weights
  - end
- **until tuning set error stops improving**

Forward pass explained earlier

Backward pass explained in next slide

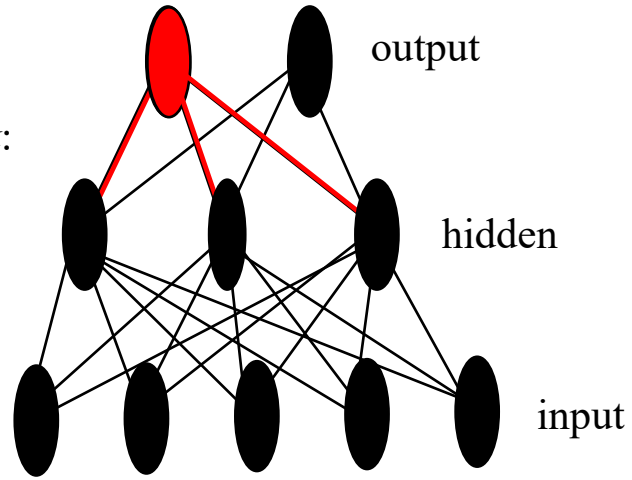
Backpropagation

Calculate the error signal for the output neurons and update the weights between the output and hidden layers

$$\delta_k = (t_k - z_k) f'(net_k)$$

update the weights to k:

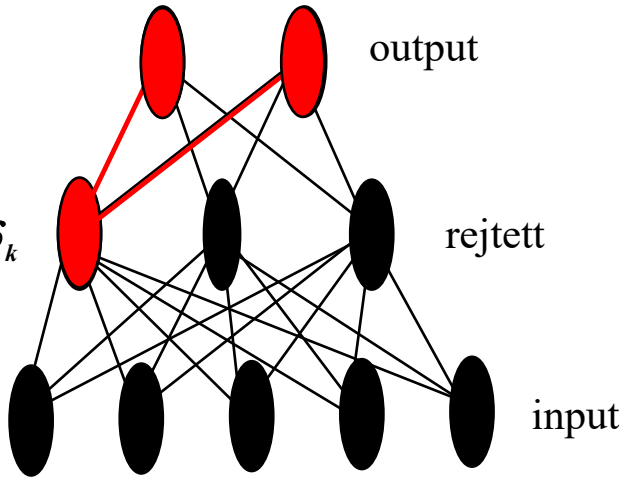
$$\Delta w_{ki} = \eta \delta_k z_i$$



Backpropagation

Calculate the error signal for hidden neurons

$$\delta_j \equiv f'(net_j)\sum_{k=1}^{c} w_{kj}\delta_k$$

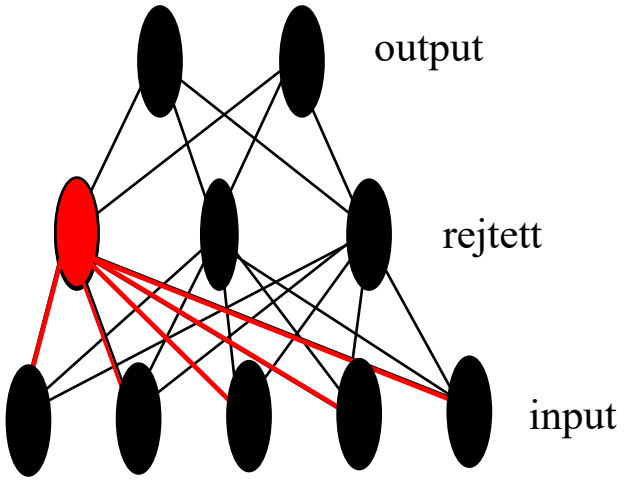


Backpropagation

Update the weights between the input and hidden neurons

updating the ones to j

$$\Delta w_{ji} = \eta \delta_j x_i$$



BackProp Algo.

Backpropagation algorithm: A History

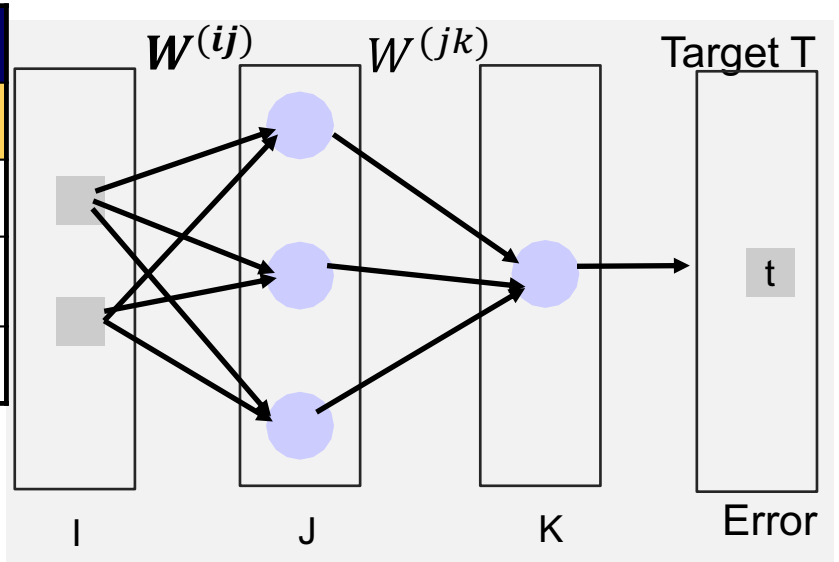
- The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams.
  - That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble.
- Today, the backpropagation algorithm is the workhorse of learning in neural networks.

Why study backprop?

- **At the heart of backpropagation is an expression for the partial derivative $\partial C/\partial w$ of the cost function C with respect to any weight w (or bias b) in the network.**
- **The expression tells us how quickly the cost changes when we change the weights and biases.**
  - And while the expression is somewhat complex, it also has a beauty to it, with each element having a natural, intuitive interpretation.
- **And so backpropagation isn't just a fast algorithm for learning. It actually gives us detailed insights into how changing the weights and biases changes the overall behaviour of the network.**
- **That's well worth studying in detail.**

Regression via MLP

Instance\Attr	x_1	x_2	…	x_n	t
1	3	0	..	7	-1
2					-4
…	…	…	…	…	…
m	0	4	...	8	23



$MLP(x) = g\left(f\left(XW^{(ij)}\right) W^{(jk)}\right)$ #MLP Regression

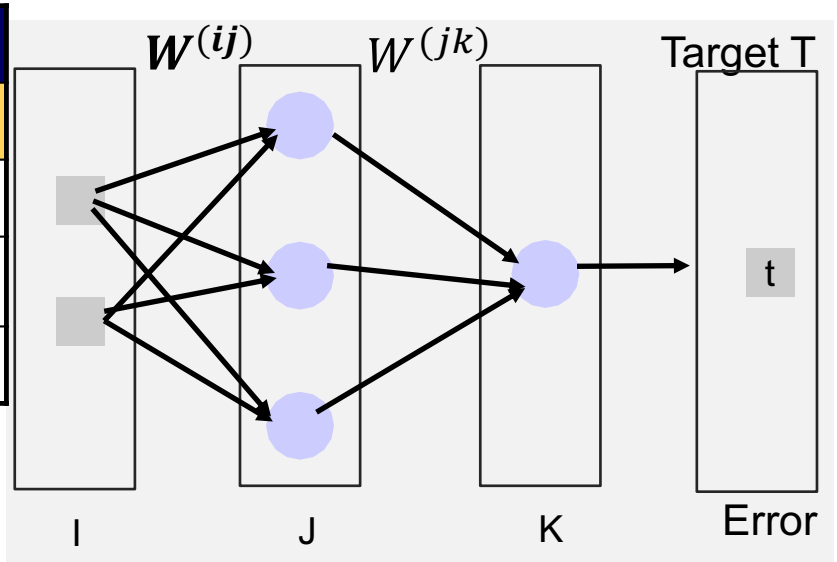
$MLP\_regression(X) = I\left(\sigma\left(XW^{(ij)}\right) W^{(jk)}\right)$ #old fashioned sigmoid

$$MSE = \frac{1}{2m}\sum_{i=1:m}(O_k - t_k)^2$$

$$MSE = \frac{1}{2m}\sum_{i=1:m}(MLP\_regression(X_i) - t_i)^2$$

Learn an MLP for Regression via BackProp (Gradient descent and chain rule of calculus)

Instance\Attr	x_1	x_2	…	x_n	t
1	3	0	..	7	-1
2					-4
…	…	…	…	…	…
m	0	4	...	8	23



$MLP(x) = g\left(f\left(XW^{(ij)}\right)W^{(jk)}\right)$ #MLP Regression

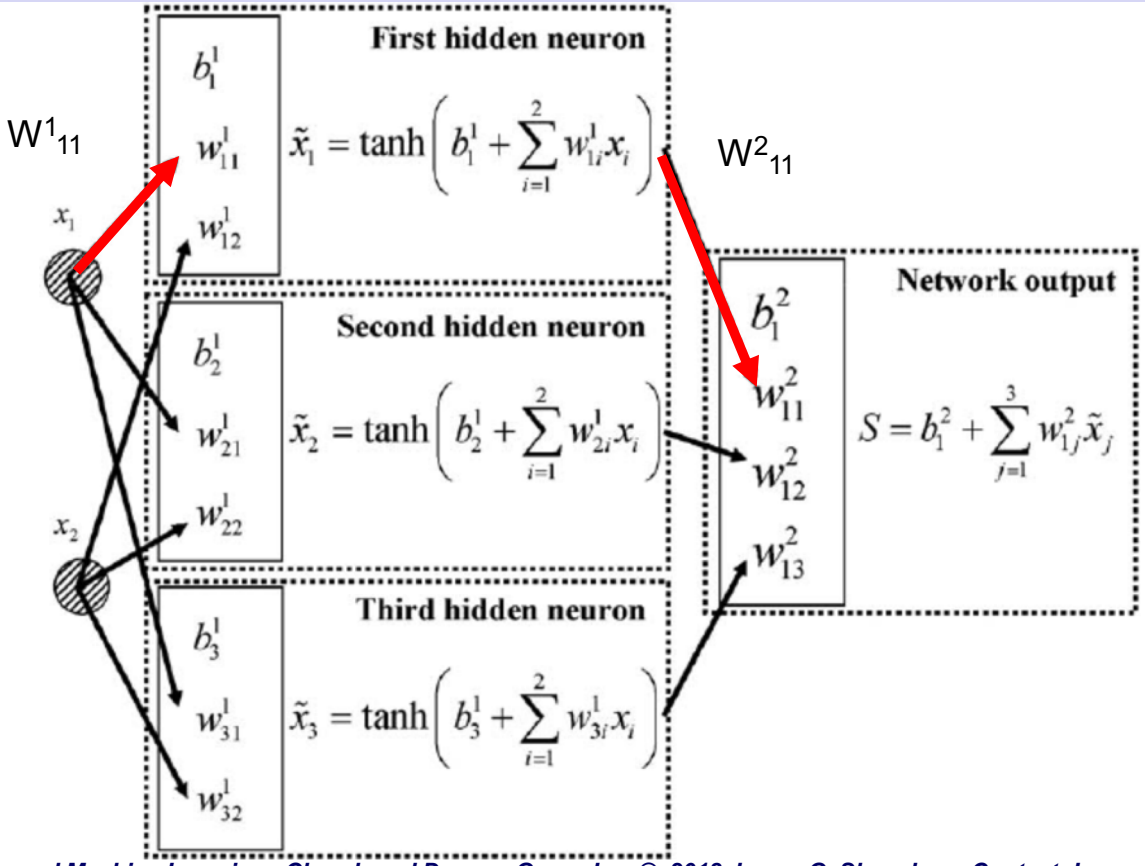
$MLP\_regression(X) = \sigma\left(\sigma\left(XW^{(ij)}\right)W^{(jk)}\right)$ #old fashioned sigmoid

$$MSE = \frac{1}{2n}\sum_{i=1:m}(O_k - t_k)^2$$

$$MSE = \frac{1}{2n}\sum_{i=1:m}(MLP\_regression(X_i) - t_i)^2$$

Learn via backpropagation algorithm (Gradient descent and chain rule of calculus)

Activation functions in NN: 2-3-1



Gradient Descent for NN

- **Given a set of training data points with target t_k and output layer output O_k we can write the error as**

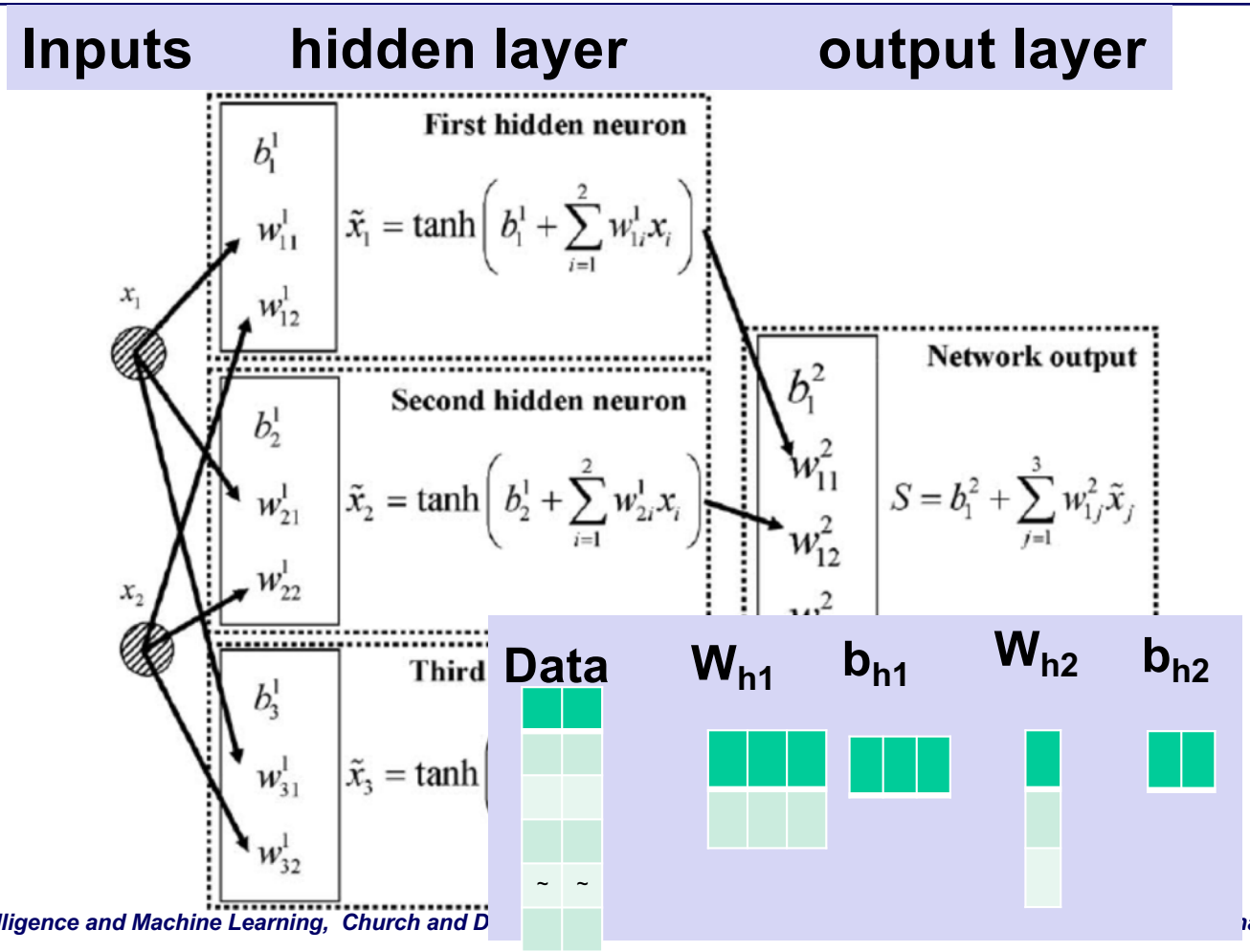
$$E = \frac{1}{2}\sum_{k \in K}(O_k - t_k)^2$$

- **How can we train the network via Gradient descent? We need to calculate the Gradient∇ (the vector of partial derivatives)**

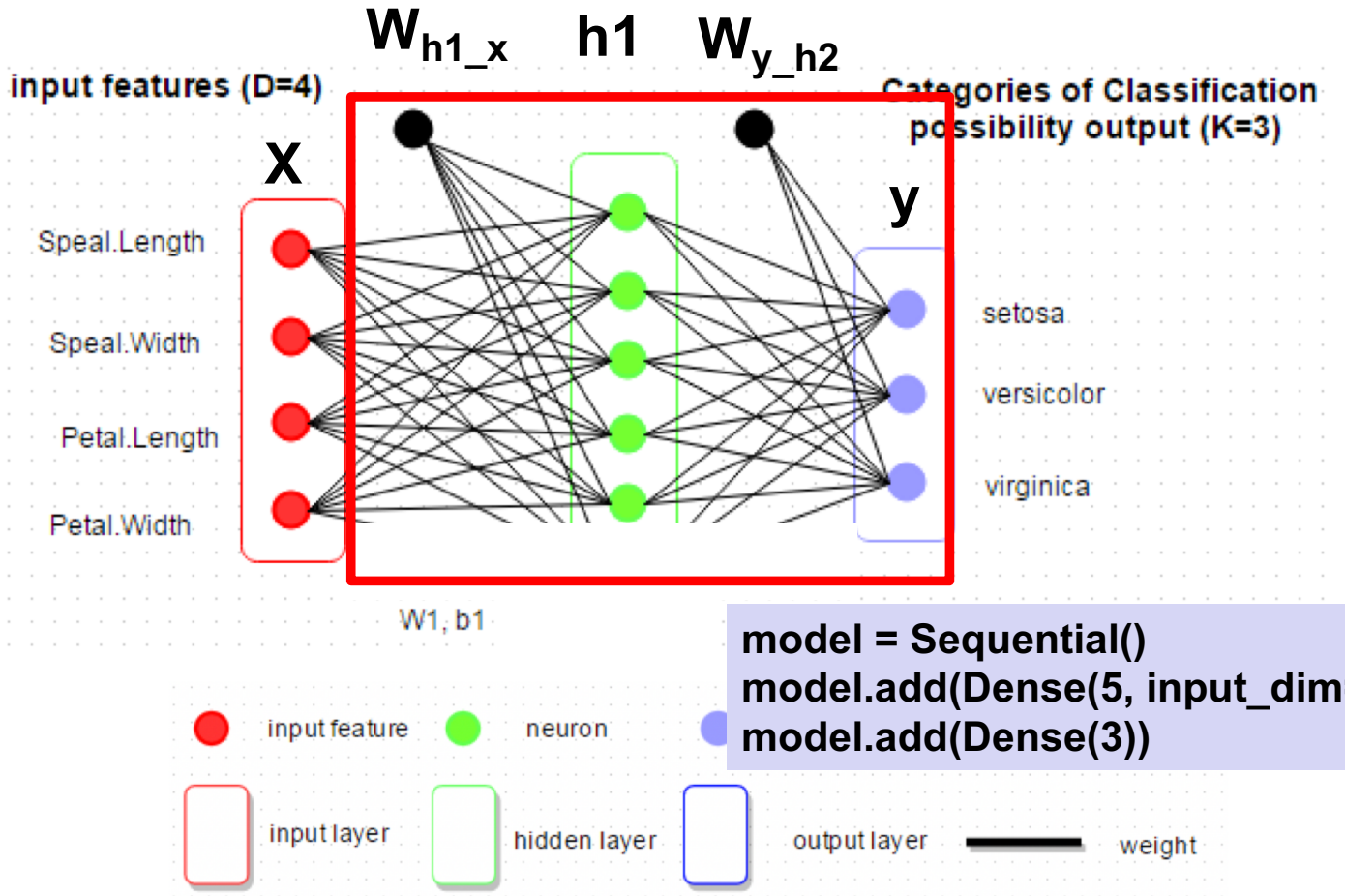
$$\frac{\partial E}{\partial W_{ij}}$$

- **Now we consider two cases:**
  - Weights associated with an output layer node
  - Weights associated with a hidden layer node...

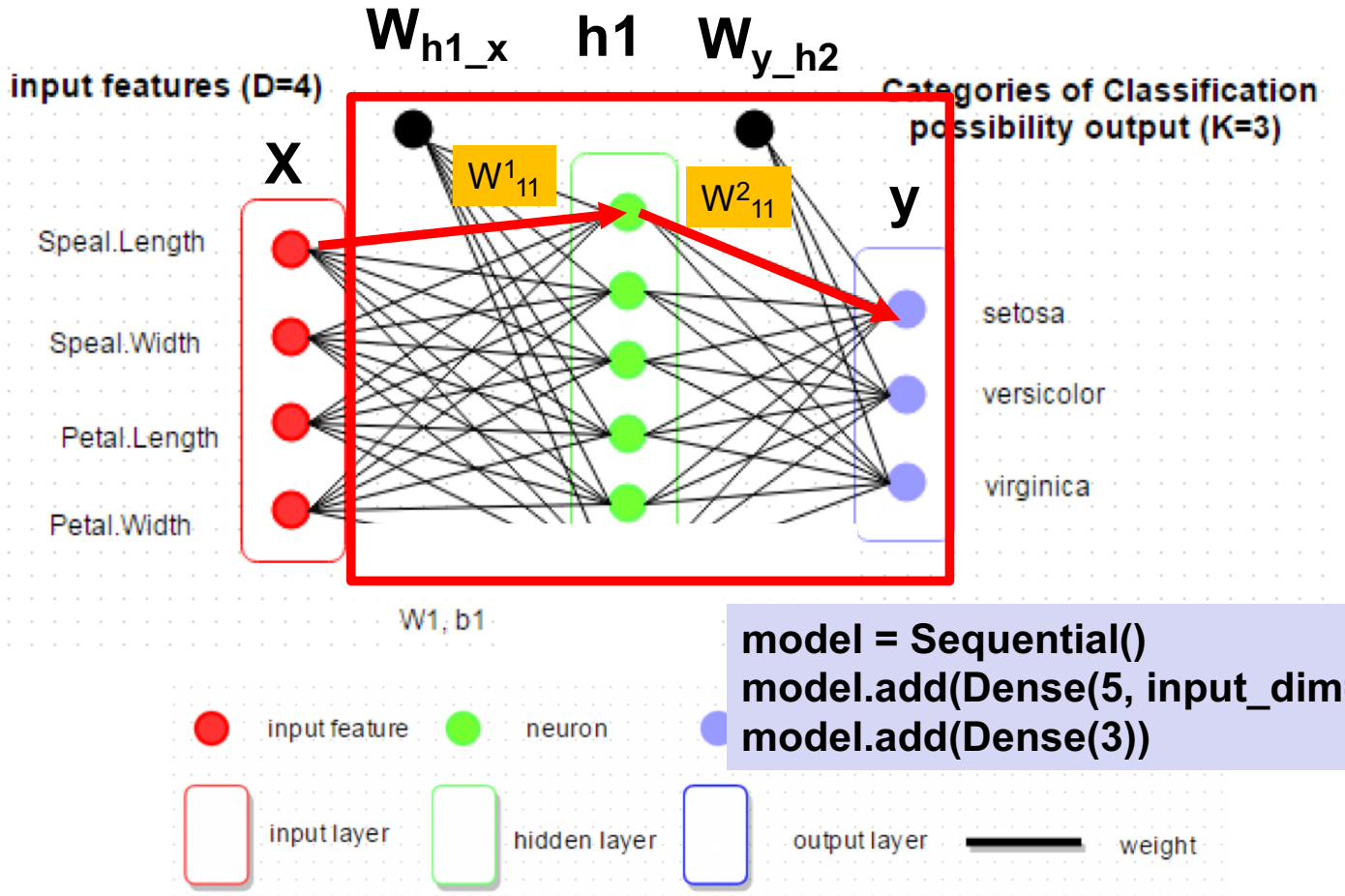
Activation functions in NN: 2-3-1



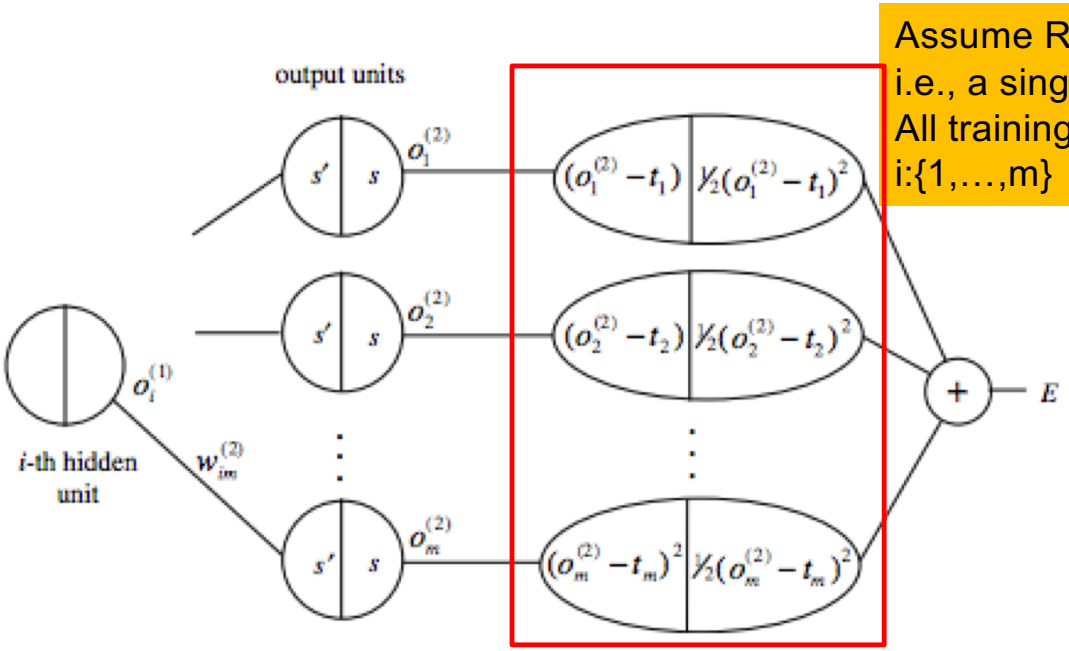
Let's work with a 4 – 5 – 3 MLP



Let's work with a 4 – 5 – 3 MLP



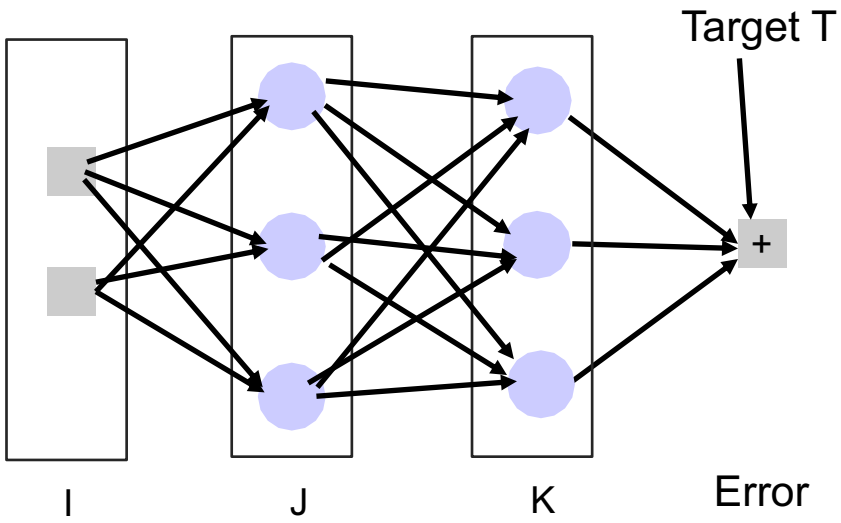
Extended multilayer network with Error



Assume Regression,
i.e., a single output
All training data
i:{1,…,m}

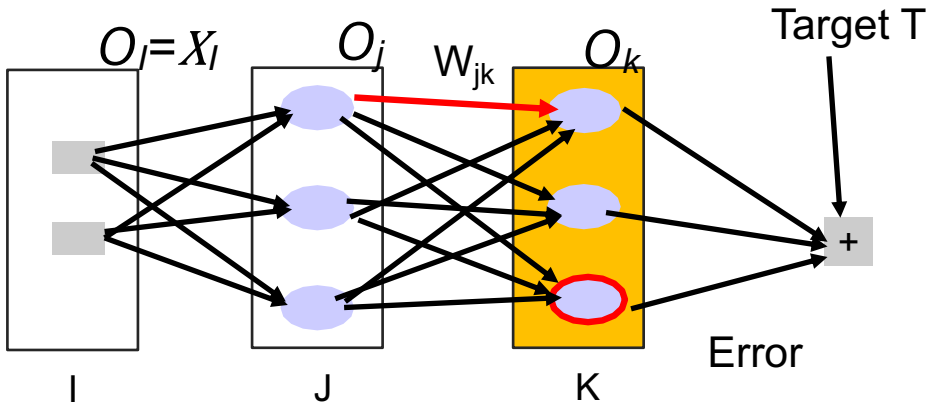
Fig. 7.18. Extended multilayer network for the computation of E

Extended multilayer network with Error

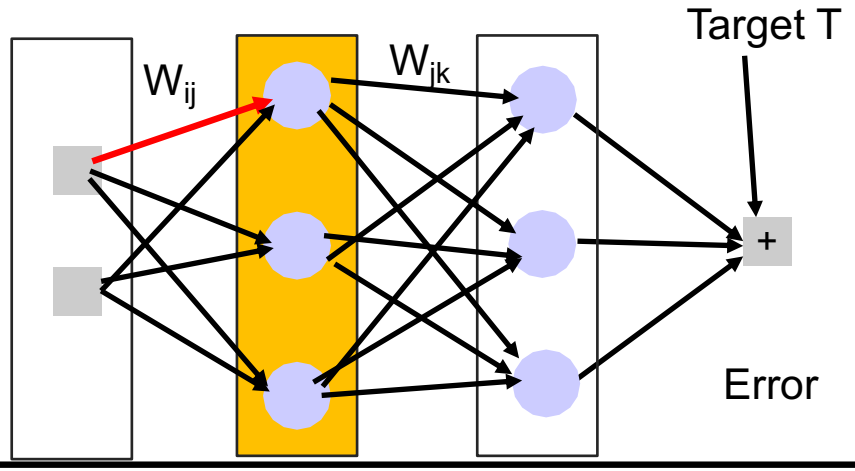


BackPropagation: Gradient Descent/Chain Rule

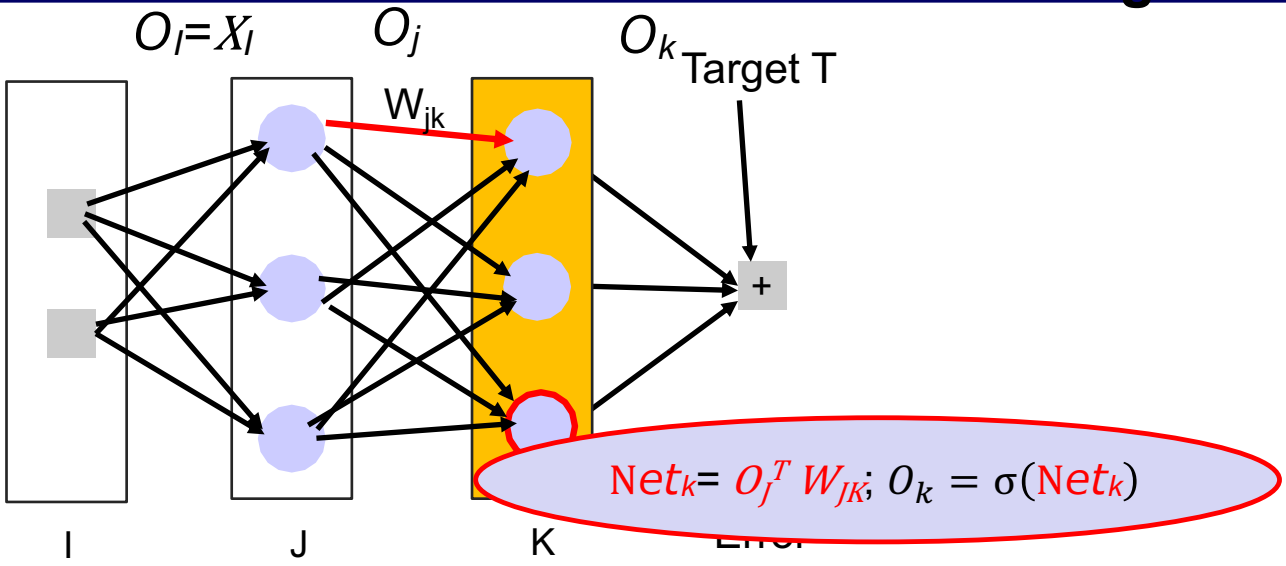
- **Case 1:Outer node weight**



- **Case 2:hidden node weight**



Gradient Descent
Case 1:Outer node weight



$(\sigma(WX) - t_k)^2 = (\sigma(net_k) - t_k)^2$

Notation:

Each layer , I, J, K, consists of node has:

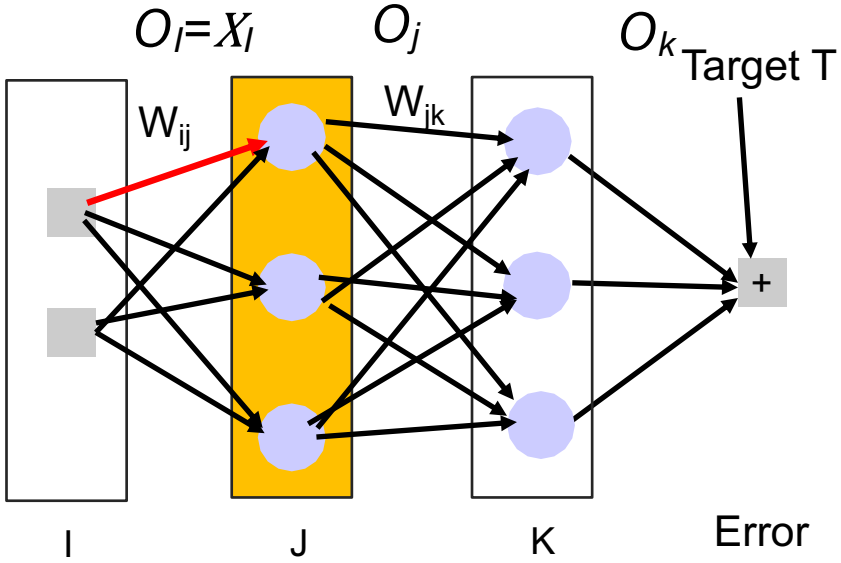
- *Weighted sum is denoted as Net_J (i.e.,$X^T W_J$)*

$\sigma(X_k)$

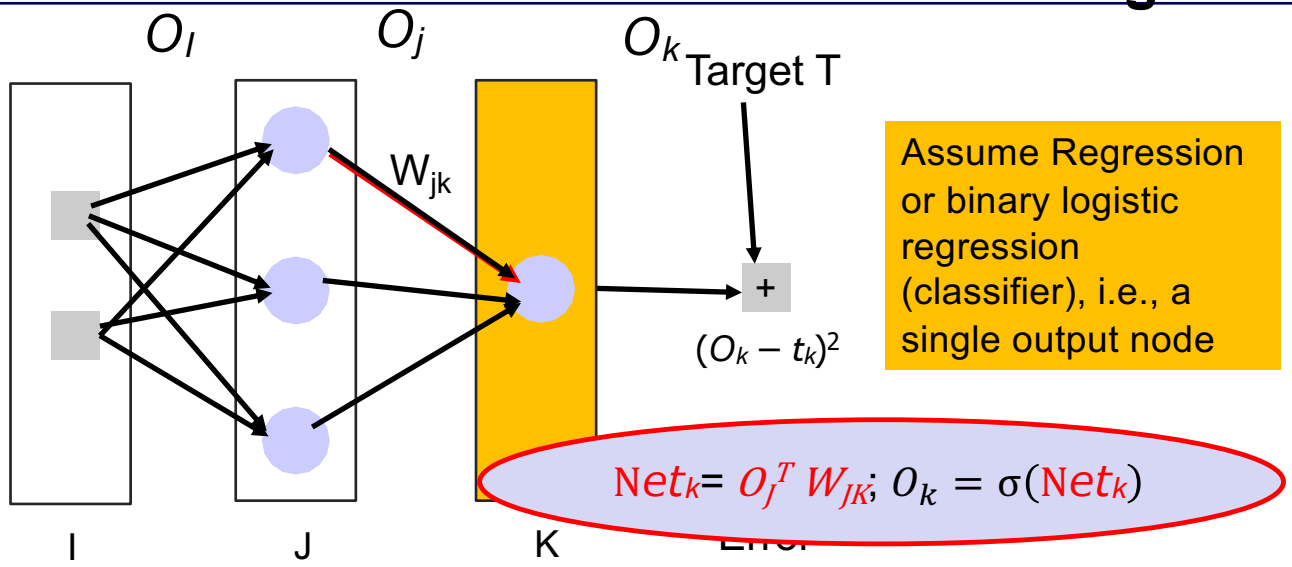
Error term, δ_k

Δ, δ delta

Gradient Descent
Case 2:hidden node weight



Gradient Descent
Case 1:Outer node weight



Notation:

Each layer, *I, J,* and *K*, consists of nodes that have in forward prop mode (prediction):

- A Weighted sum is denoted as Net_J (i.e.,$X^T W_J$);
- an activation $\sigma(X_k)$ where the activation func is sigmoid

In BackProp mode: each node has error term, δ_k

Instance\Attr	x_0	x_1	…	x_n	y
1	1	-3	..	-7	-1
2	1				+1
…	1	…	…	…	…
L (aka m)	1	0	…	8	-1

W
x_0
x_1
…
x_n

Linear Regression via GD

MSE = $\sum_{n=1:Train} \frac{1}{2}$(Prediction – target)^2

$$MSE = \sum_{n=1:Train} \frac{1}{2}(X_i\ w - t_i)^2$$

$$MSE = (XW\ - Y)^2 \quad \#vectorized$$

All training data $\frac{\partial E}{\partial w} = \frac{\partial}{\partial w}\sum_{n=1:Train} \frac{1}{2}(X_i\ w - t_i)^2$

One example $\frac{\partial E}{\partial w} = \frac{\partial}{\partial w}\frac{1}{2}(o - t)^2 \quad \#Let\ o = X_i\ w - t_i$

$$\frac{\partial E}{\partial w} = (o - t)\frac{\partial}{\partial w}(o - t)$$

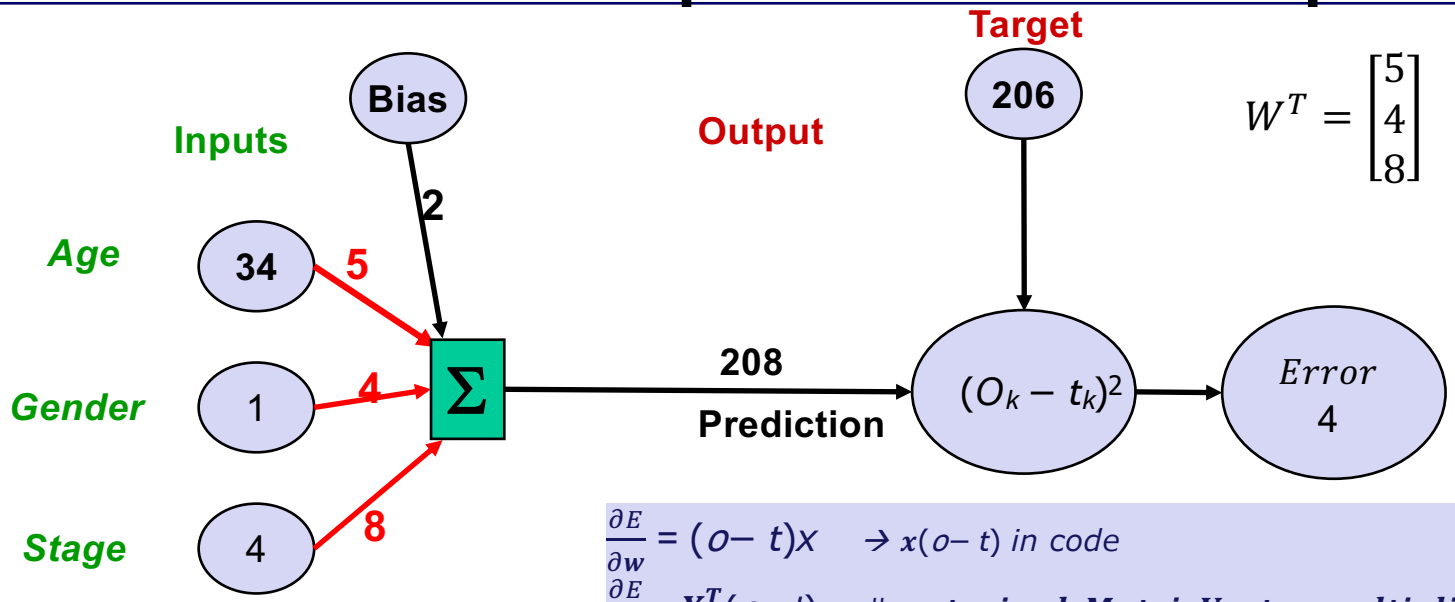
$$\frac{\partial E}{\partial w} = (o - t)\frac{\partial}{\partial w}(xw - t)$$

$$\frac{\partial E}{\partial w} = (o - t)x$$

All training data $\frac{\partial E}{\partial w} = X^T(o - t) \quad \#vectorized; MatrixVector\ multiplication$

$$w = w - \alpha\frac{\partial E}{\partial w} \quad AKA. \quad w = w - \alpha\nabla_w$$

Linear Regression Model: Gradient update for one example



$$W^T = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix}$$

$\frac{\partial E}{\partial w} = (o - t)x$ → $x(o - t)$ *in code*

$\frac{\partial E}{\partial w} = X^T(o - t)$ $\#vectorized; MatrixVector\ multiplication$

$w = w - \alpha \frac{\partial E}{\partial w}$ $AKA.\ \ w = w - \alpha \nabla_w$

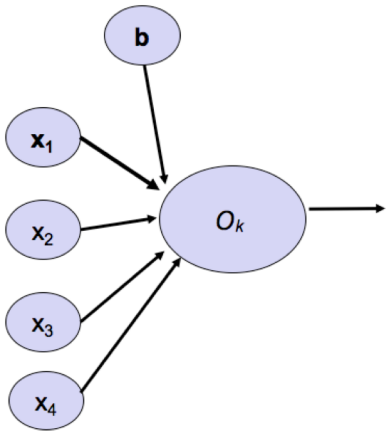
$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} 34 \\ 1 \\ 4 \end{bmatrix}$$

$4.32 = 5 - 0.01 \times (2 \times 34)$ for $i = 1$

Weight Gradient for this training example[34,1,4]; b = 2

Assume a learning rate α=0.01

Activation matrices and model matrices for a single target variable



4 1 1 Prediction

150 X 4 W = 150 P − T

W = W $-\alpha\nabla_W$

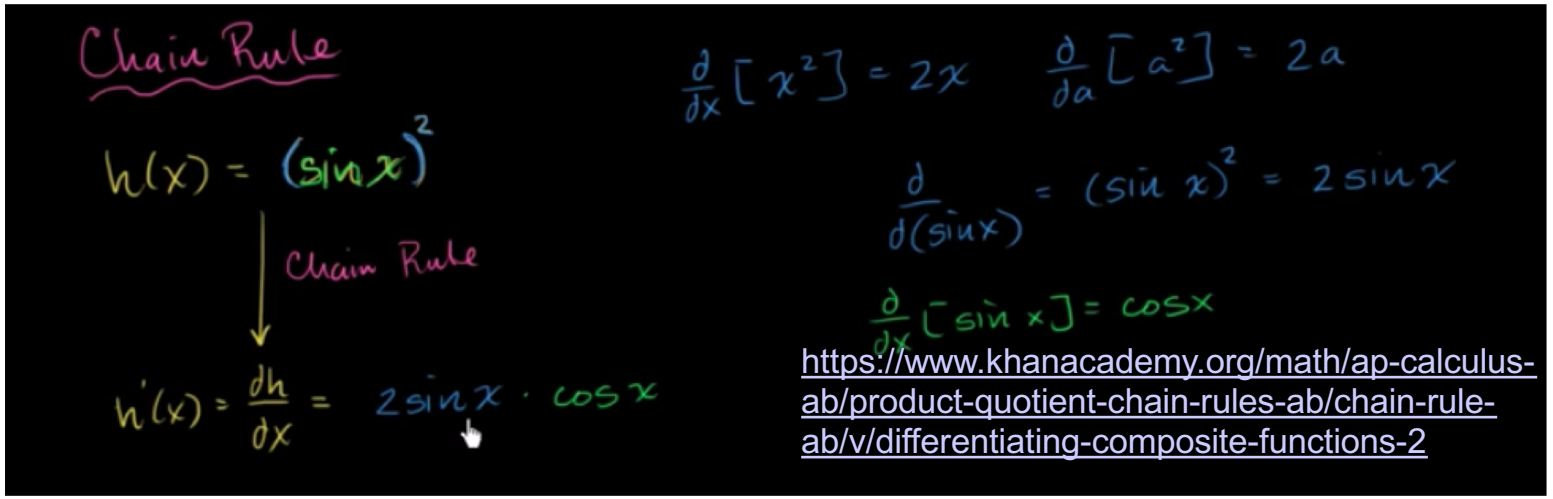
∇_W = Weighted Sum × E

Model

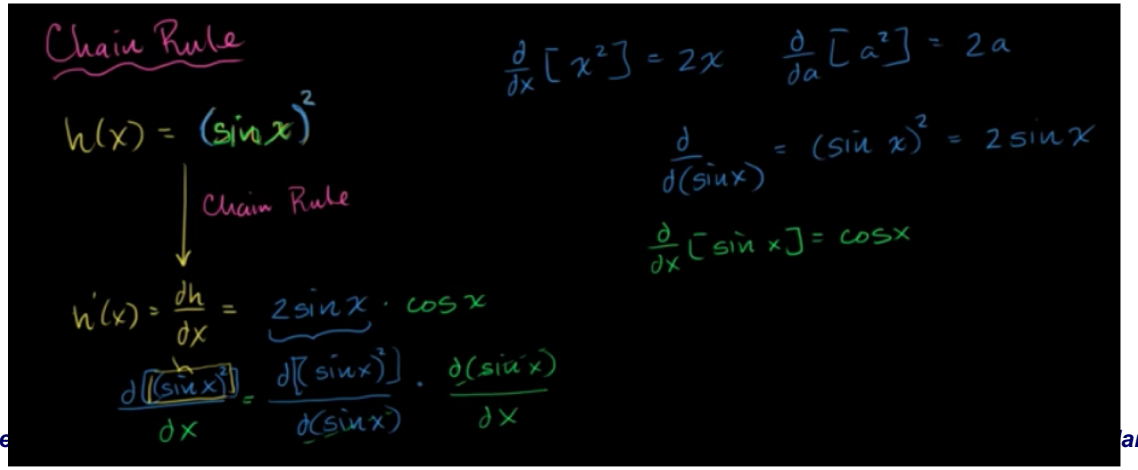
Data

$$\begin{bmatrix}4.32\\3.98\\7.92\end{bmatrix} = \begin{bmatrix}5\\4\\8\end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix}34\\1\\4\end{bmatrix}$$

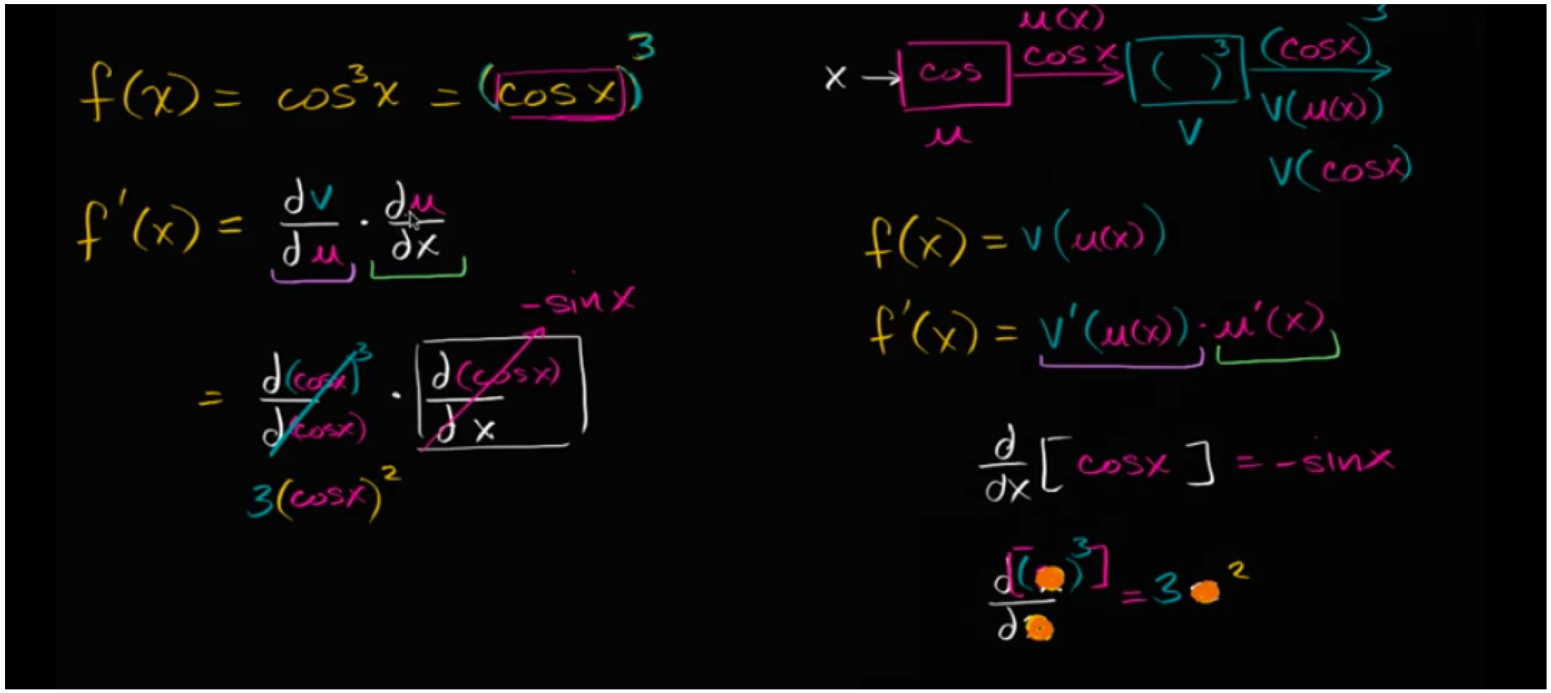
Chain Rule (Khan Academy)



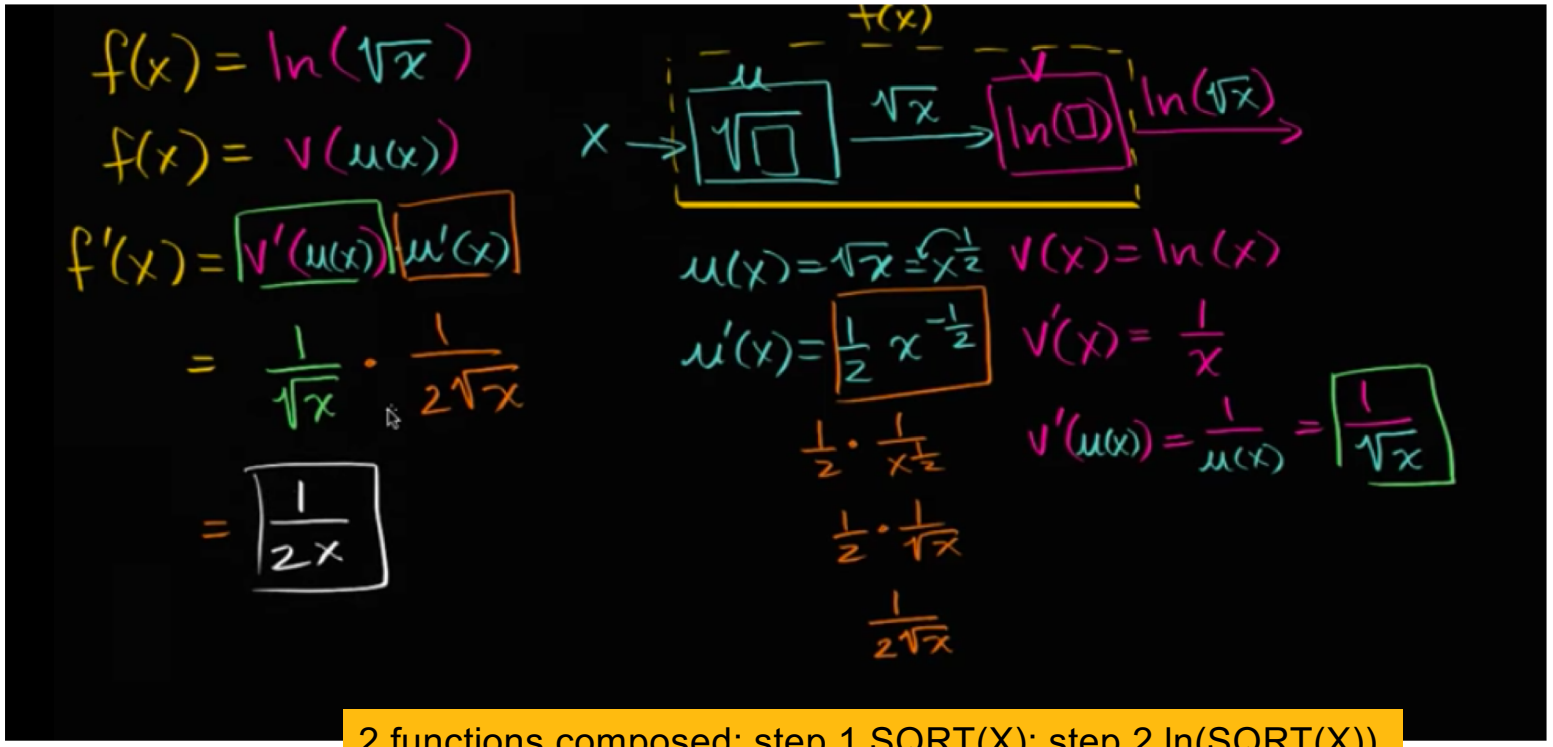
https://www.khanacademy.org/math/ap-calculus-ab/product-quotient-chain-rules-ab/chain-rule-ab/v/differentiating-composite-functions-2



Chain Rule Example 2



Chain Rule example 3



2 functions composed: step 1 SQRT(X); step 2 ln(SQRT(X))
Derivative of the outside function wrtt inside function
Derivative of the inside function wrt X

MLP Regression Model: 13-13Relu-1

- **Boston dataset**

```
# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(13, input_dim=13, kernel_initializer='normal', activation='relu
    model.add(Dense(1, kernel_initializer='normal'))
    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

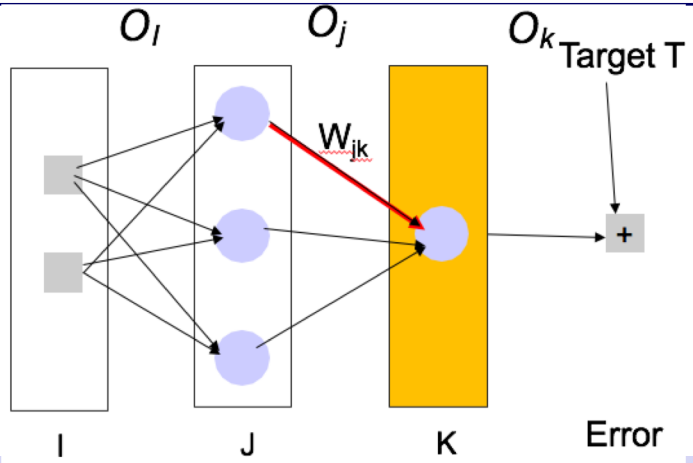
$MSE(X; MLP_{Regression}) = \frac{1}{2n}(I(W_2\ (\sigma(W_1\ X)))\ - t_k)^2$ where I is identity

$MSE(X; MLP_{Regression}) = \frac{1}{2n}(\sigma(W_2\ (\sigma(W_1\ X)))\ - t_k)^2\ \ Where\ target\ \epsilon[0,1]$

Below we assume a sigmoid activation function in the output layer. HERESY!

Given a NN where $Target\ \epsilon[0,1]$

Sigmoid Regression!



MSE(X; $MLP_{Regression}$) = $\frac{1}{2n}(I(W_2\ (\sigma(W_1\ X)))\ -\ t_k)^2$ #where I is identity

MSE(X; $MLP_{Regression}$) = $\frac{1}{2n}(\sigma(W_2\ (\sigma(W_1\ X)))\ -\ t_k)^2$ #Here we assume $Target\ \epsilon[0,1]$

O_k= = $\frac{1}{2n}(\sigma(W_2\ (\sigma(W_1\ X)))\ -\ t_k)^2$

$$\frac{\partial E}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}}\frac{1}{2}(O_k - t_k)^2$$ MSE

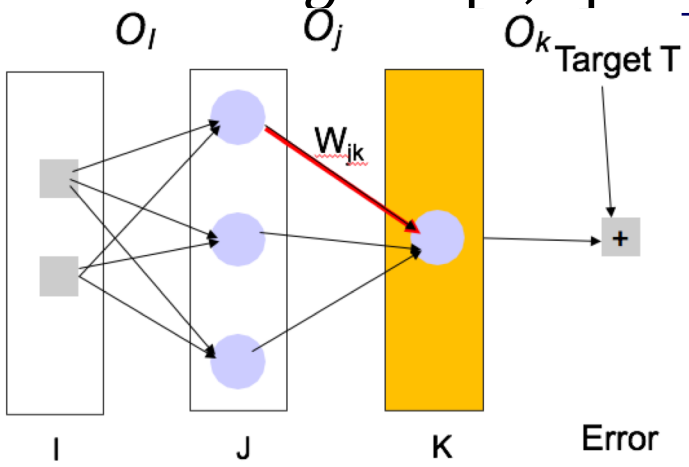
$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\ \frac{\partial}{\partial W_{jk}}\ (O_k - t_k)$$ After applying chain rule once

Need to unpack the network using the chain rule

Given MSE MLP:

Assume MSE loss where Target $\epsilon[0,1]$

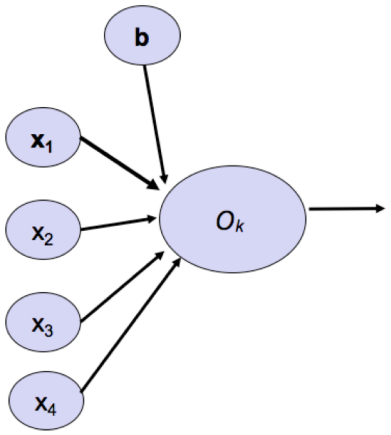
$\text{Net}_k = O_J W_{JK}$; $O_k = \sigma(\text{Net}_k)$



$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\frac{\partial}{\partial W_{jk}}(O_k - t_k)$$

$$\textit{Assume MSE loss where Target } \epsilon[0,1]\ (\sigma(XW) - t_k)^2 = (\sigma(\text{net}_k) - t_k)^2$$

Activation matrices and model matrices for a single target variable



$$\sigma\left(\underset{150}{\mathbf{X}}\ \underset{4}{\mathbf{W}}\right) = \underset{150}{\mathbf{P}}\ \underset{1}{\mathbf{T}}$$

4 (X) 1 (W) 1 Prediction

$$\mathbf{W} = \mathbf{W} - \alpha\ \nabla_W$$

Model

Data

$$\nabla_W = \mathbf{X}^{\mathbf{T}}\ \mathbf{O}\ \delta_K$$

Weighted Sum

$$\frac{\partial E}{\partial W_{jk}} = O_j \delta_K$$

$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} 34 \\ 1 \\ 4 \end{bmatrix}$$

$Assume\ MSE\ loss\ where\ Target\ \epsilon[0,1]\ (\sigma(WX) - t_k)^2 = (\sigma(net_k) - t_k)^2$

Output layer node: single output MSE Loss (0-1 range)

$$\frac{\partial E}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}} \frac{1}{2}(O_k - t_k)^2$$

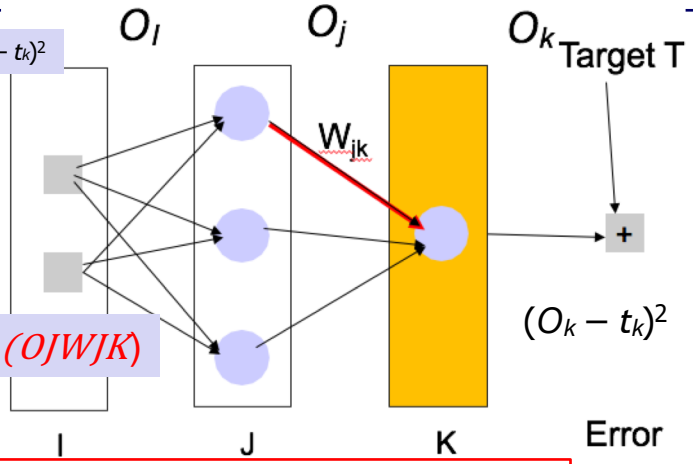
Chain rule where u = $(O_k - t_k)^2$

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\ \frac{\partial}{\partial W_{jk}}\ (O_k - t_k)$$

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\ \frac{\partial}{\partial W_{jk}}\ (O_k)$$

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\ \frac{\partial}{\partial W_{jk}}\ (\sigma(net_k))$$

$\sigma(net_k)=\sigma(OJWJK)$



$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\ (\sigma(net_k)(1-\sigma(net_k))\ \frac{\partial}{\partial W_{jk}}\ net_k$$

Chain rule where u = net_k

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)(\sigma(net_k)(1-\sigma(net_k))O_j$$

$simplify\ as\ O_k = \sigma(Net_k)$

Simplify

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)O_k(1-O_k)O_j$$

Error Vector (for weighted sum in the gradient calc.) for node output node k

Let $\delta_K = (O_k - t_k)O_k(1-O_k)$

For terms involving k simplify and replace with one term δ_K

$$\frac{\partial E}{\partial W_{jk}} = O_j\delta_K$$

$Net_k = O_J^T\ W_{JK};\ O_k = \sigma(Net_k)$

$(\sigma(WX) - t_k)^2 = (\sigma(net_k) - t_k)^2$ View net_k as the input to node k

Output layer node: single output; single example

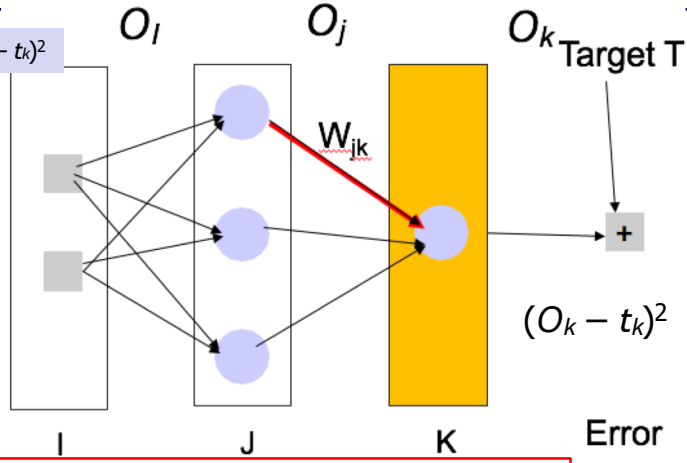
$$\frac{\partial E}{\partial W_{jk}} = \frac{\partial}{\partial W_{jk}} \frac{1}{2}(O_k - t_k)^2$$

Chain rule where u = $(O_k - t_k)^2$

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\frac{\partial}{\partial W_{jk}}(O_k - t_k)$$

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\frac{\partial}{\partial W_{jk}}(O_k)$$

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)\frac{\partial}{\partial W_{jk}}(\sigma(net_k))$$



$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)(\sigma(net_k)(1-\sigma(net_k))\frac{\partial}{\partial W_{jk}} net_k$$

Chain rule where u = net_k

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)(\sigma(net_k)(1-\sigma(net_k))O_j$$

$$\frac{\partial E}{\partial W_{jk}} = (O_k - t_k)O_k(1-O_k)O_j$$

Simplify

Let $\delta_K = (O_k - t_k)O_k(1-O_k)$

For terms involving k simplify and replace with one term δ_K

Think of this as error (weight) for output node k

$$\frac{\partial E}{\partial W_{jk}} = O_j\delta_K$$

Weight update

X → Y in linear regression

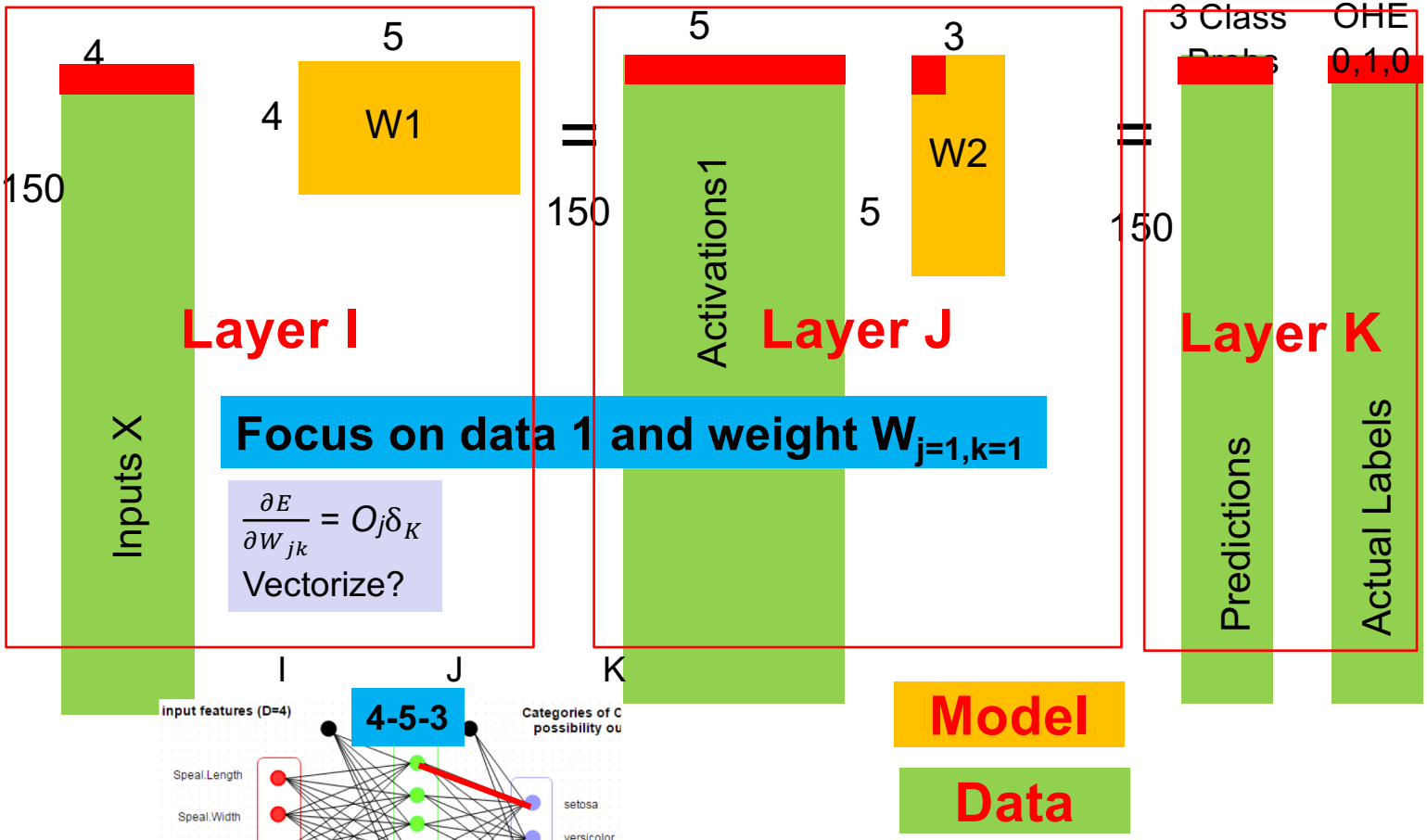
$O_i \rightarrow O_k$ in NN

o be the input vector in the case of

gression but now it means the

as from layer J here

Activation matrices and weight matrices for Iris NN with a 4-5-3 architecture



∇ the gradient chorus

- **What is the gradient for linear regression?**

- **Chorus**
  - The gradient is the weighted sum of the training data, where the weights are proportional to the error (for each example) !

weight *example*

$$\frac{\partial E}{\partial W} = (O - t)\,\boldsymbol{X}$$

$$W = W - \alpha \times \frac{\partial E}{\partial W}$$

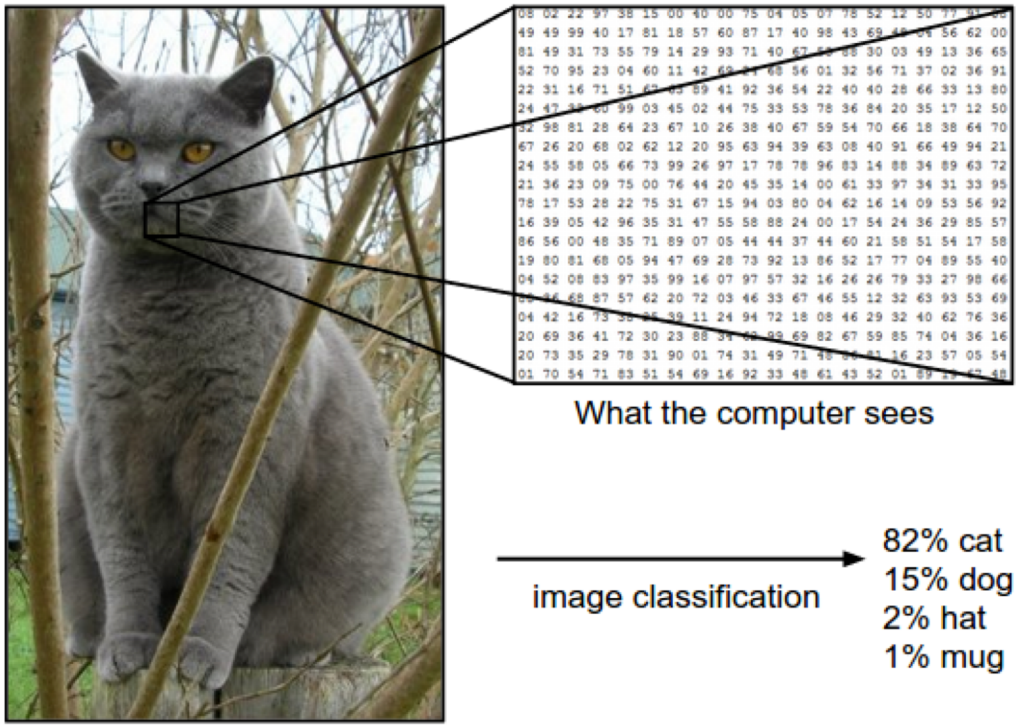
$$\begin{bmatrix} 4.32 \\ 3.98 \\ 7.92 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \\ 8 \end{bmatrix} - 0.01 \times 2 \times \begin{bmatrix} \mathbf{34} \\ \mathbf{1} \\ \mathbf{4} \end{bmatrix}$$

$$4.32 = 5 - 0.01 \times (2 \times 34) \text{ for } i = 1$$

Outline

1. **Introduction**
2. **ML and deep learning review**
3. **What is computer vision?**
    1. Computer vision
    2. Convolutional Neural Nets (CNNs)
4. **Deep NN Architectures for CV Tasks**
    1. Backbone networks
5. **Solving edge-based IoT**
6. **Conclusions and Next steps**

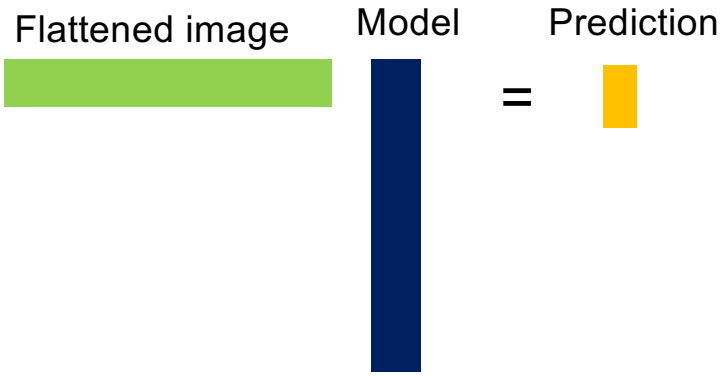
Flatten image and learn a logistic regression classifier



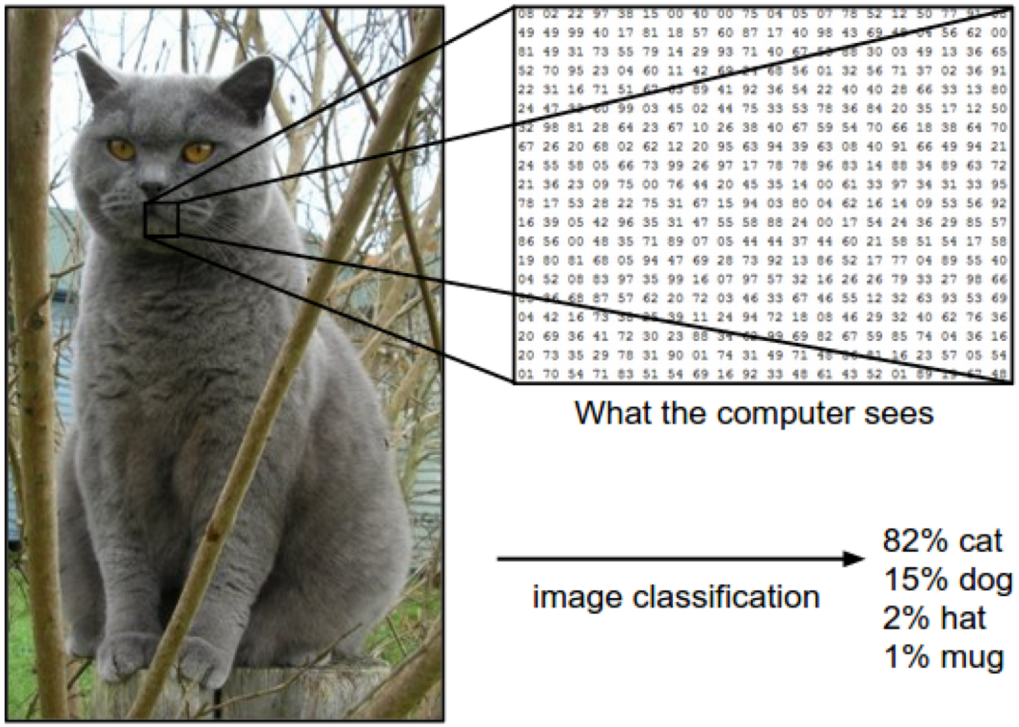
What the computer sees

image classification

82% cat
15% dog
2% hat
1% mug



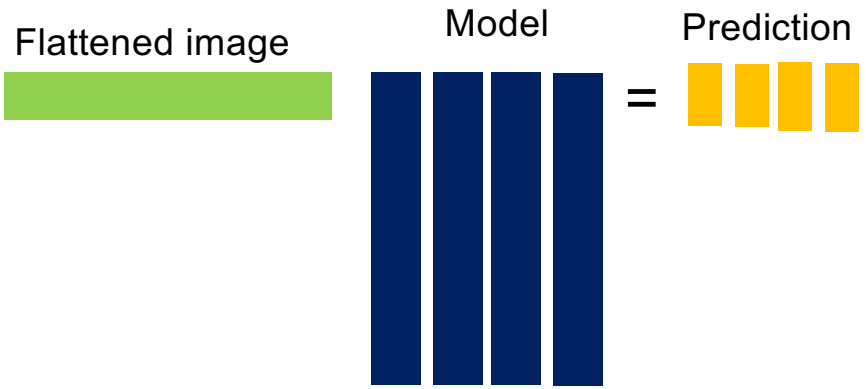
Flatten image and learn a logistic regression classifier



What the computer sees

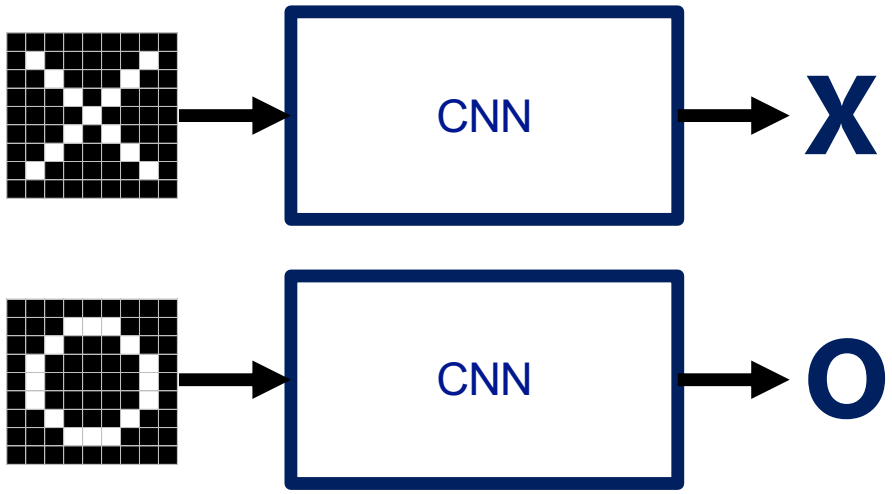
image classification

82% cat
15% dog
2% hat
1% mug



Classify these images:
Ignores the spatial relations

Classify these images: take into account the spatial relations



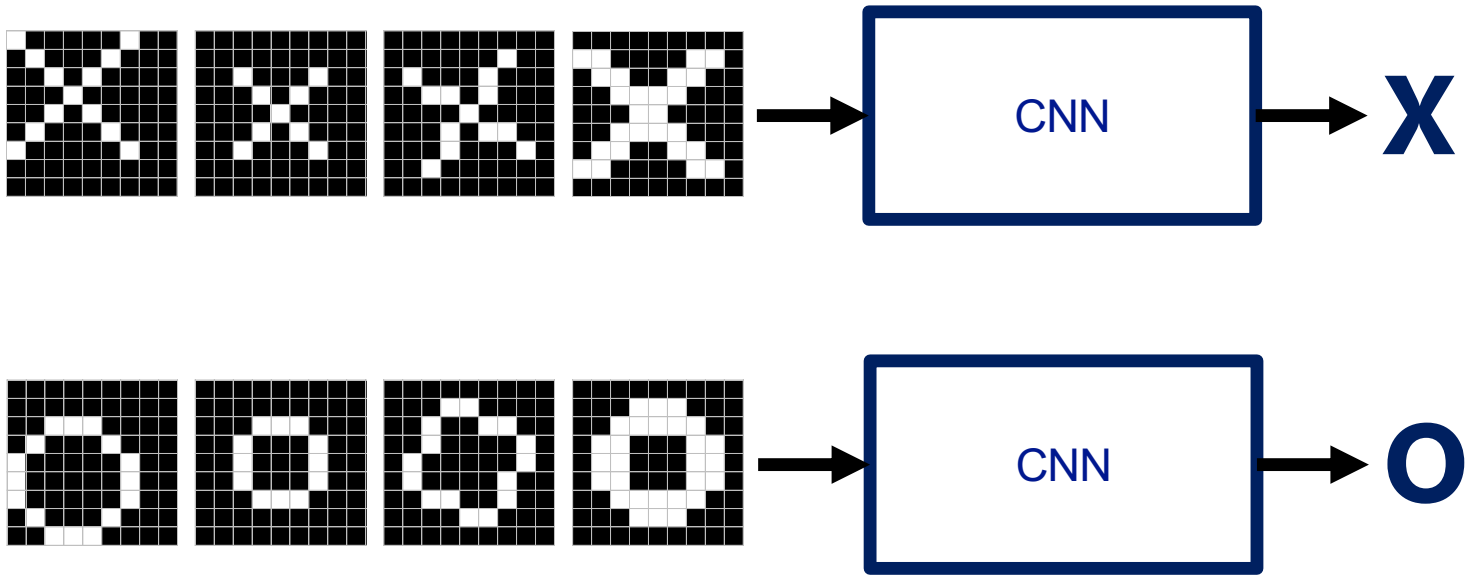
Slides and YouTube Video by Brandon Rohrer [August 2016]

https://www.youtube.com/watch?v=FmpDIaiMIeA

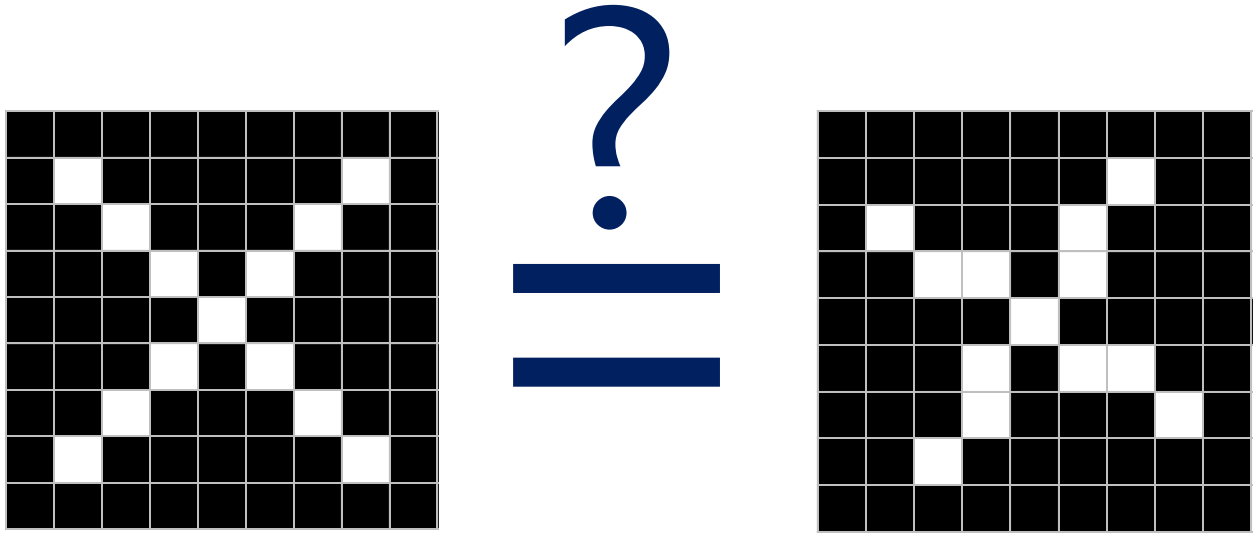
SLIDES

https://github.com/brohrer/public-hosting/blob/master/how_CNNs_work.pptx?raw=true

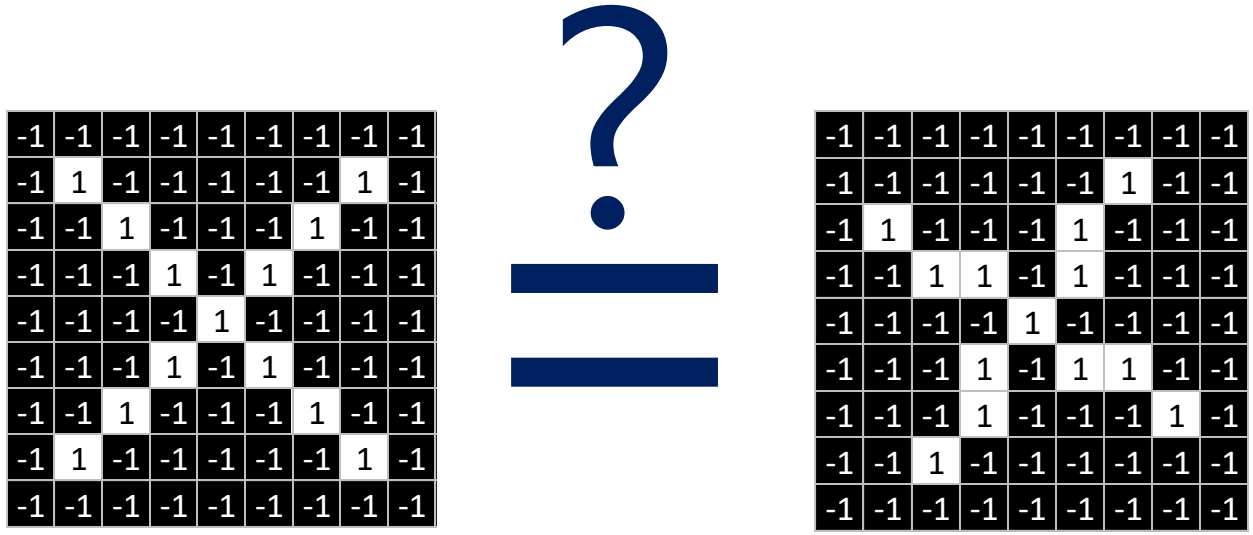
Trickier cases



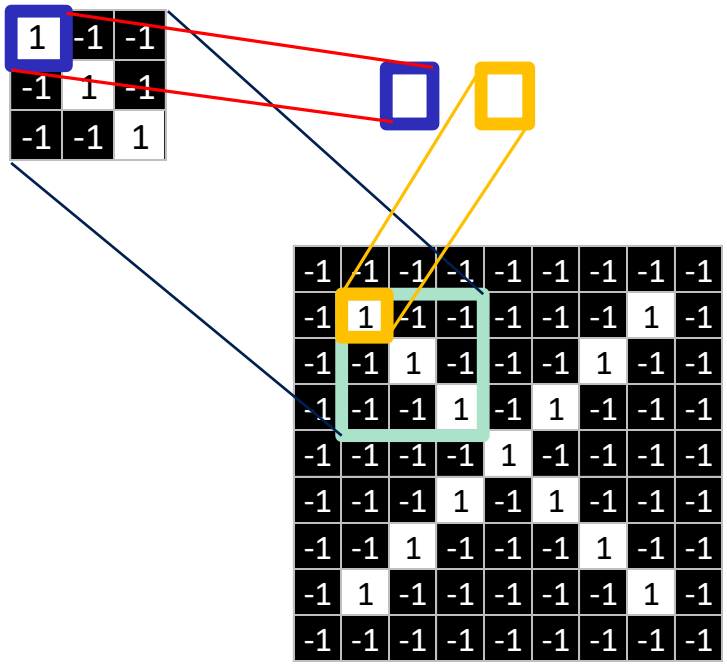
Deciding is hard



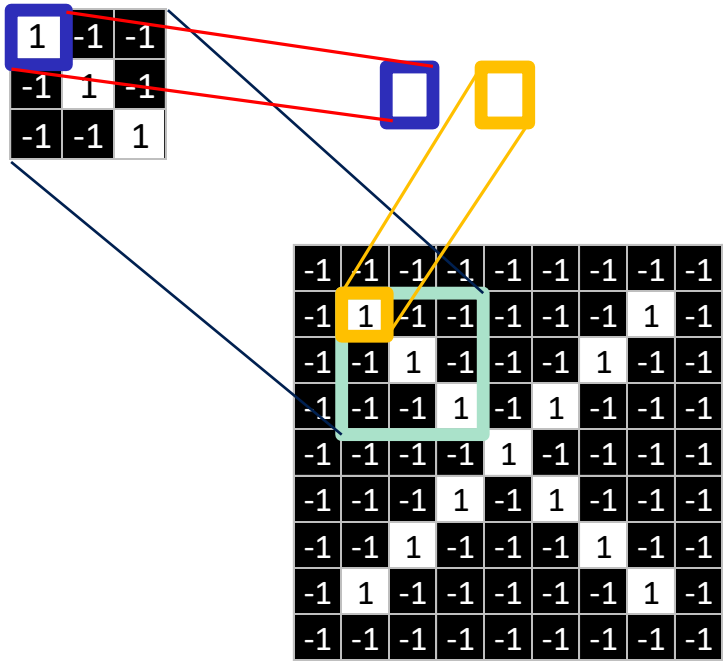
What computers see

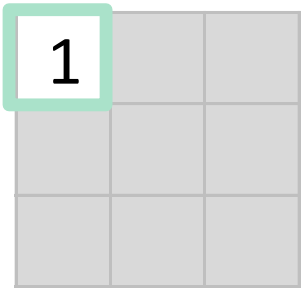


Filtering: The math behind the match

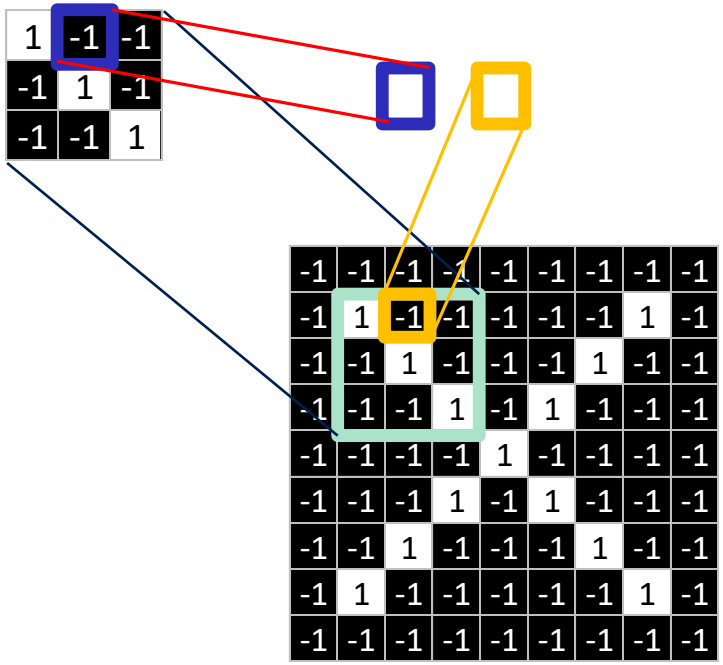


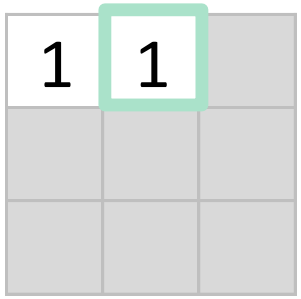
Filtering: The math behind the match



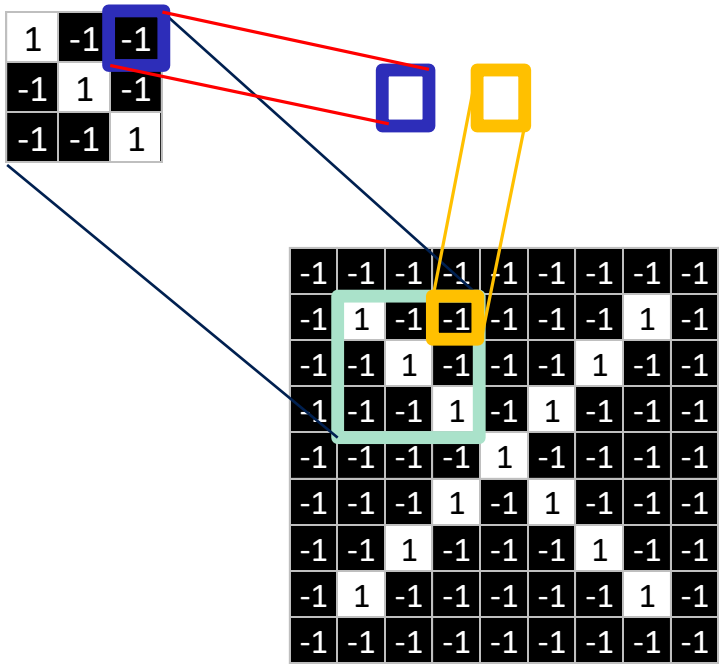


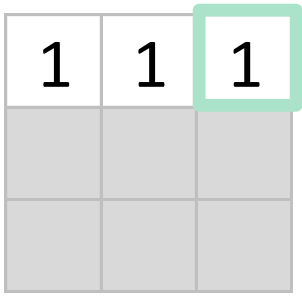
Filtering: The math behind the match



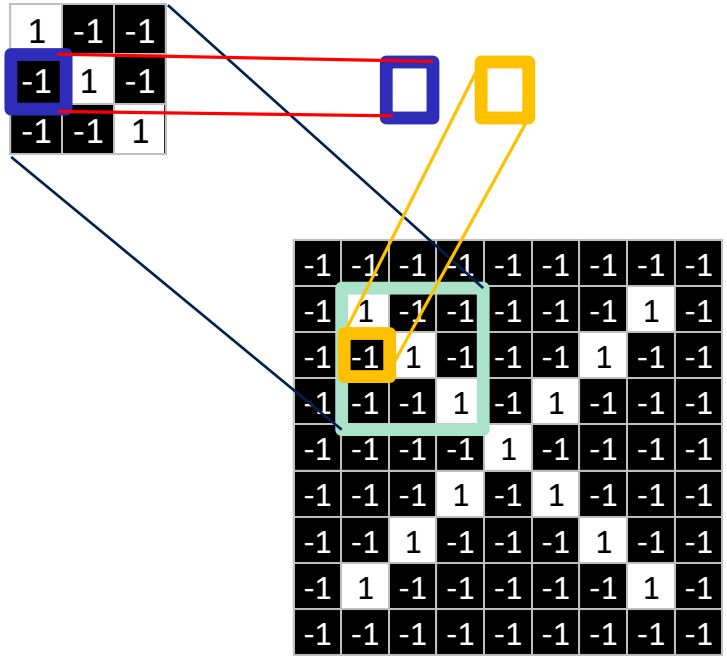


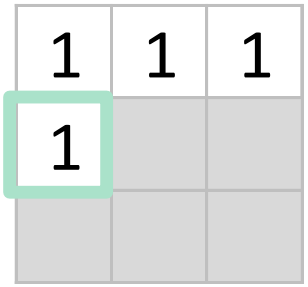
Filtering: The math behind the match



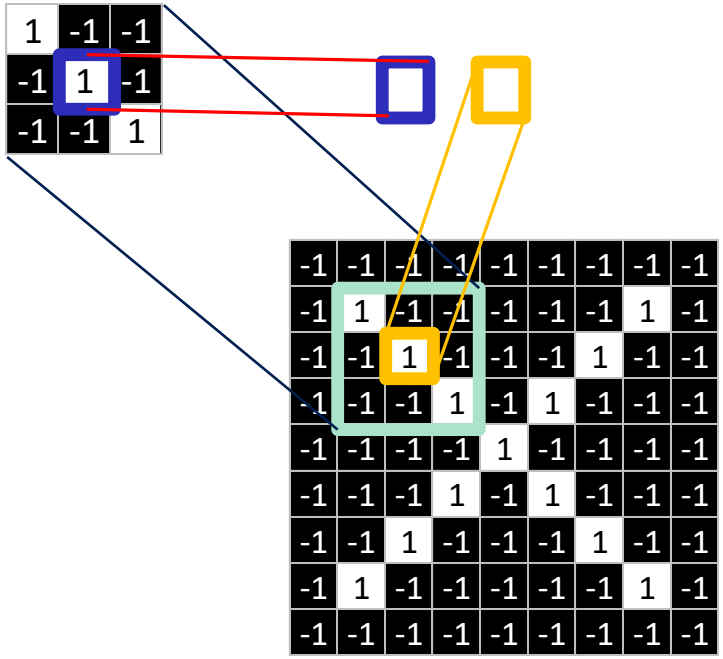


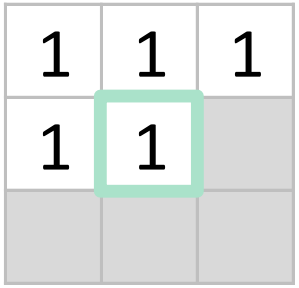
Filtering: The math behind the match



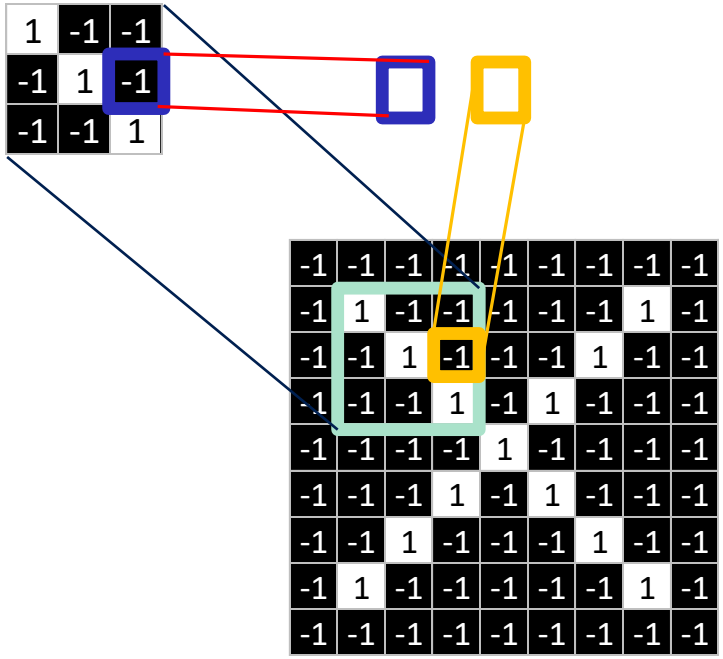


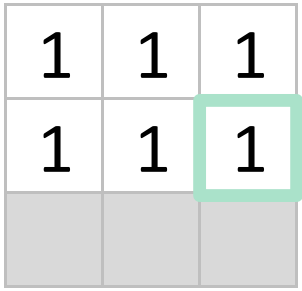
Filtering: The math behind the match



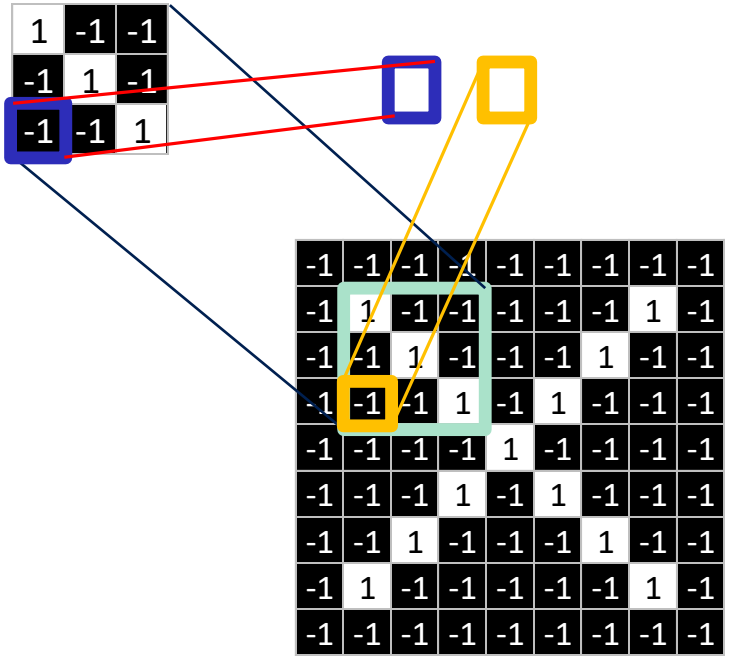


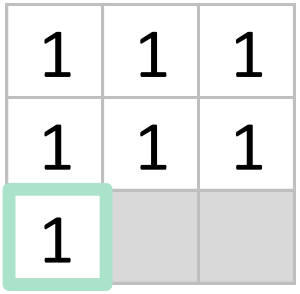
Filtering: The math behind the match



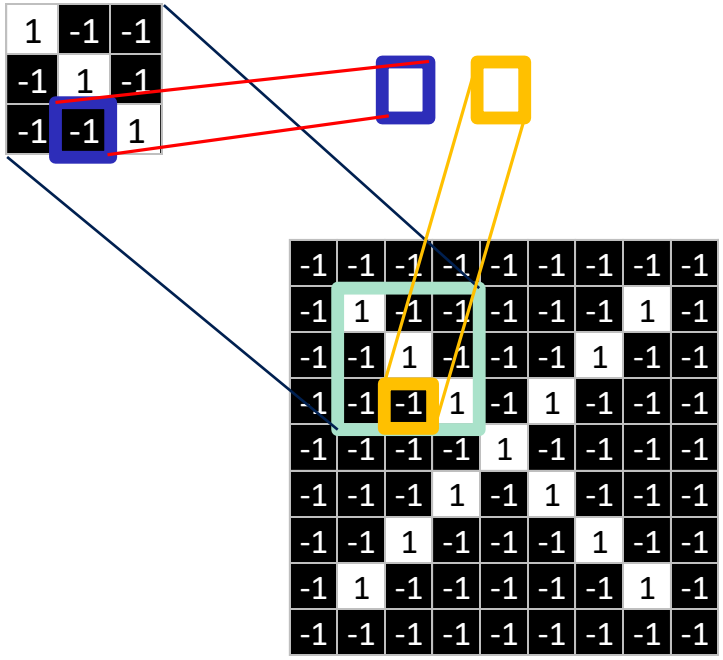


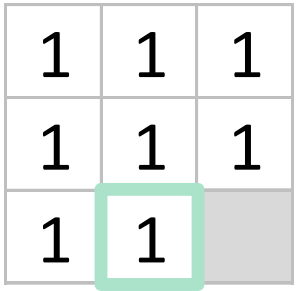
Filtering: The math behind the match



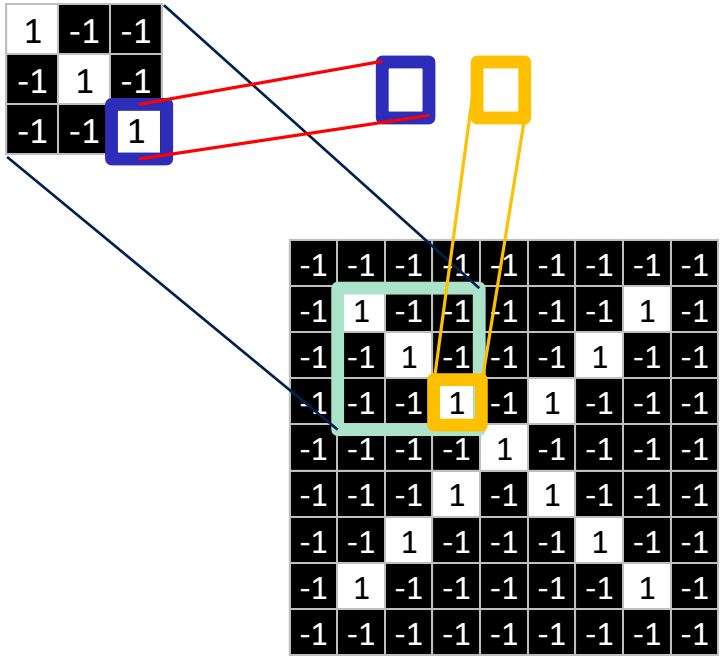


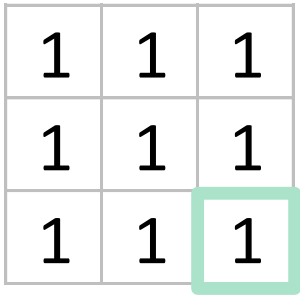
Filtering: The math behind the match



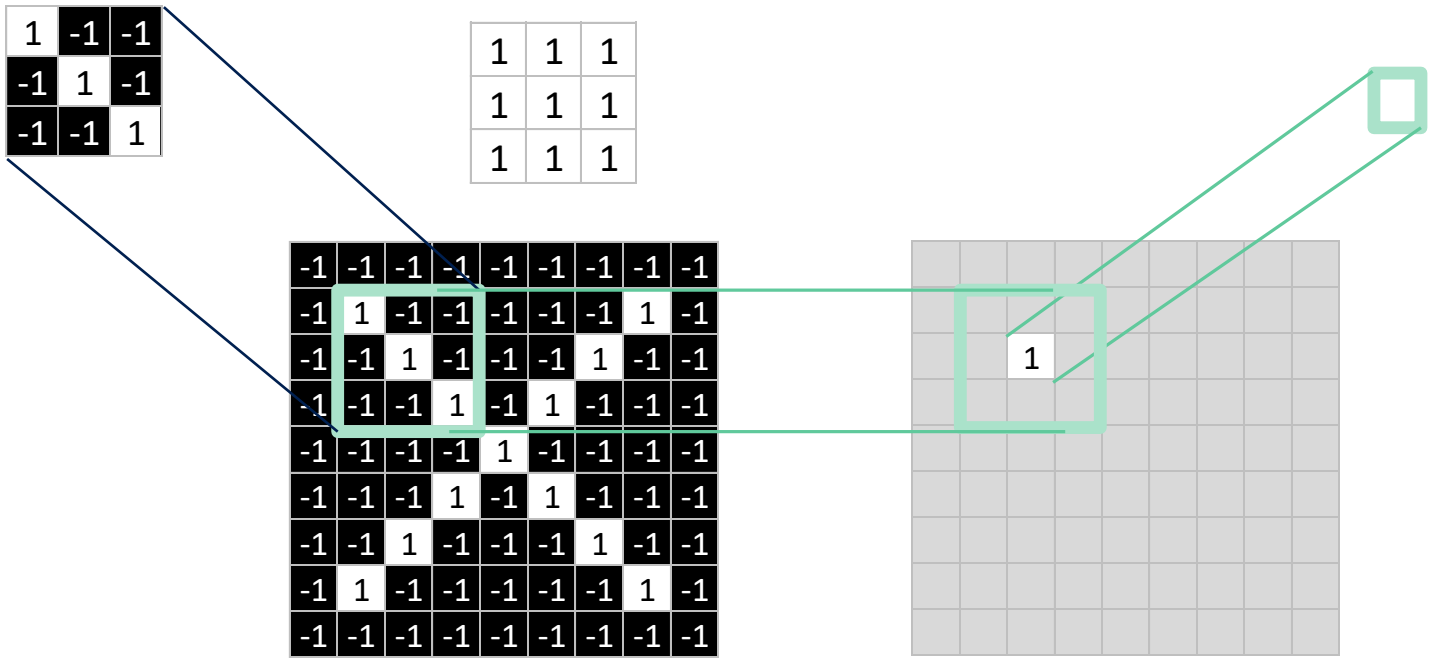


Filtering: The math behind the match

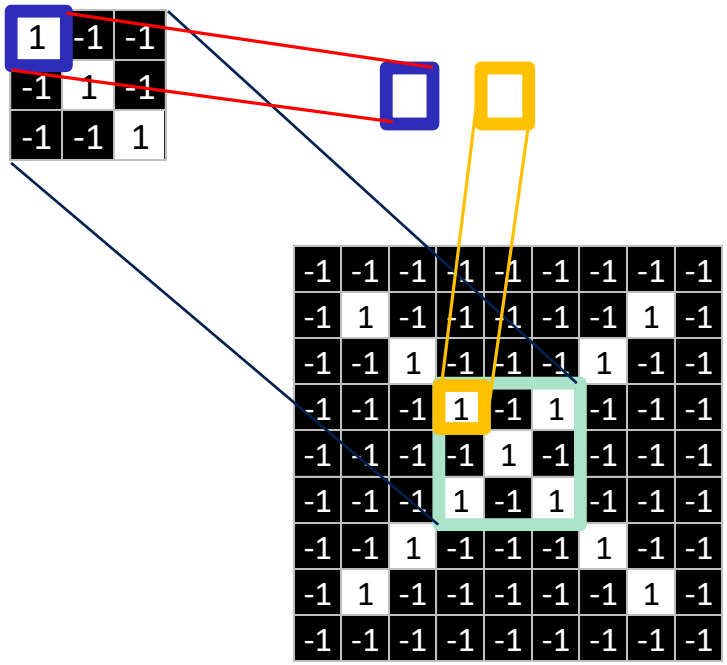


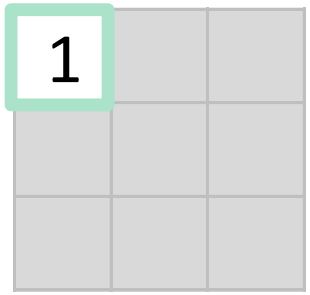


Filtering: The math behind the match

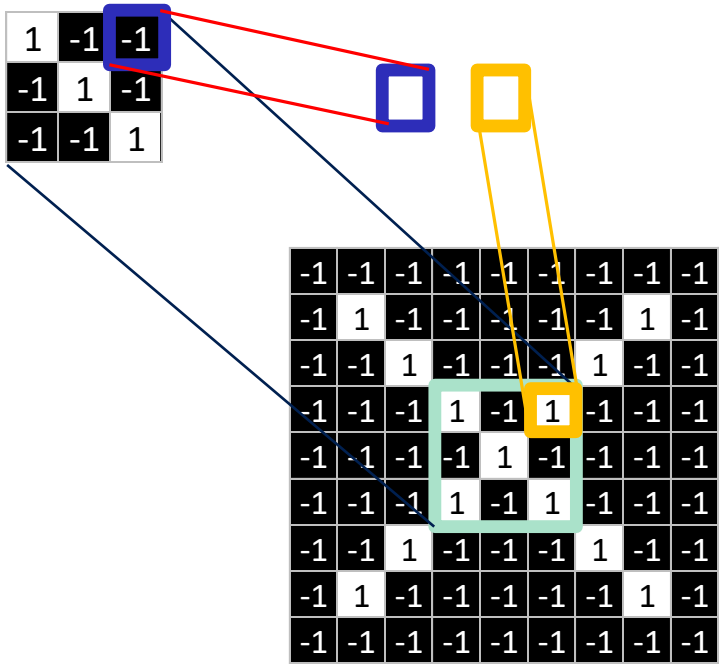


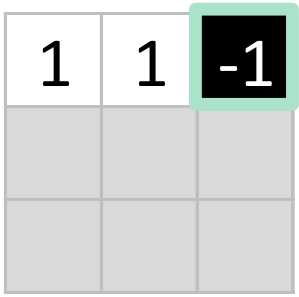
Filtering: The math behind the match



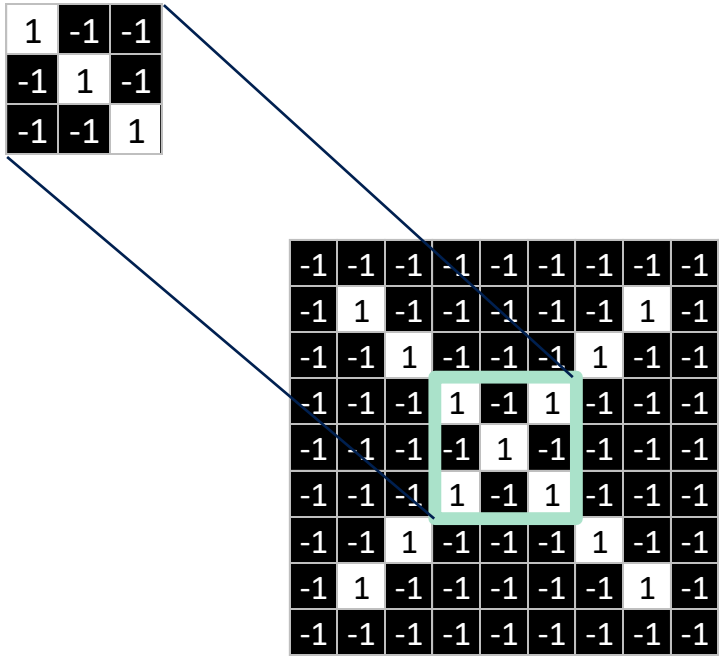


Filtering: The math behind the match



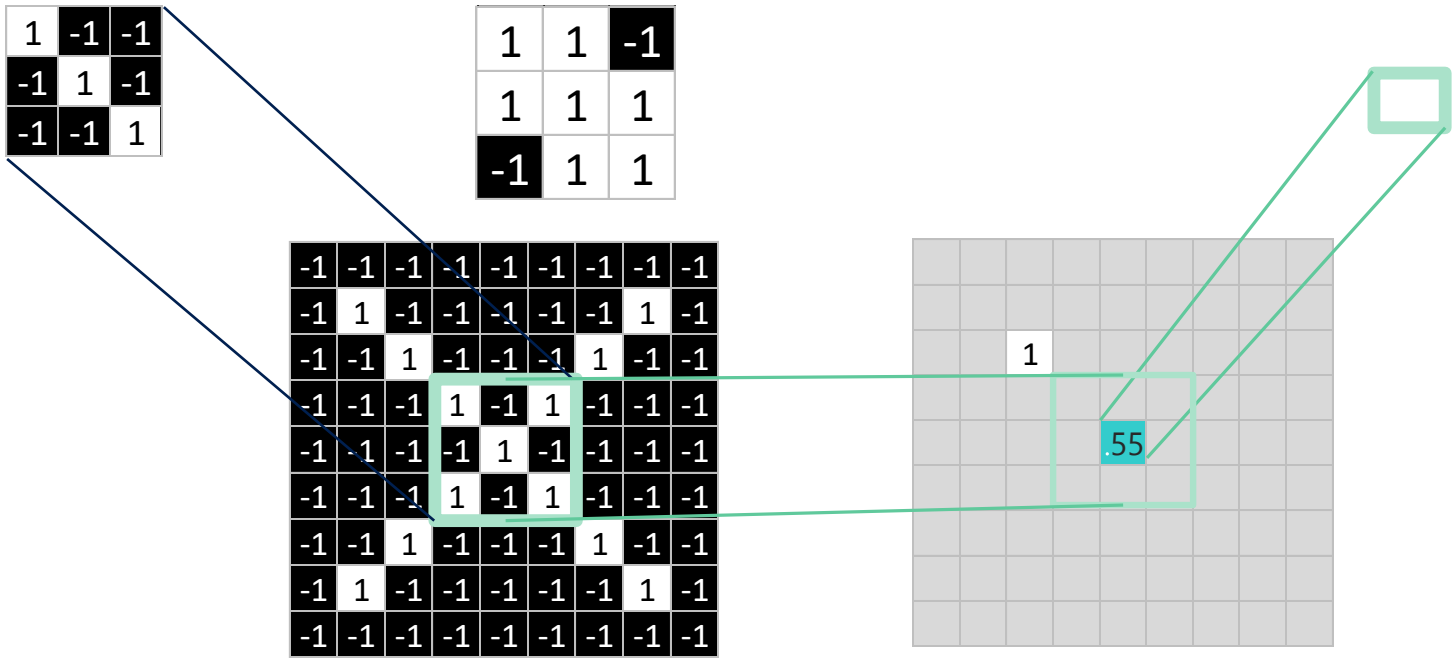


Filtering: The math behind the match



1	1	-1
1	1	1
-1	1	1

Filtering: The math behind the match



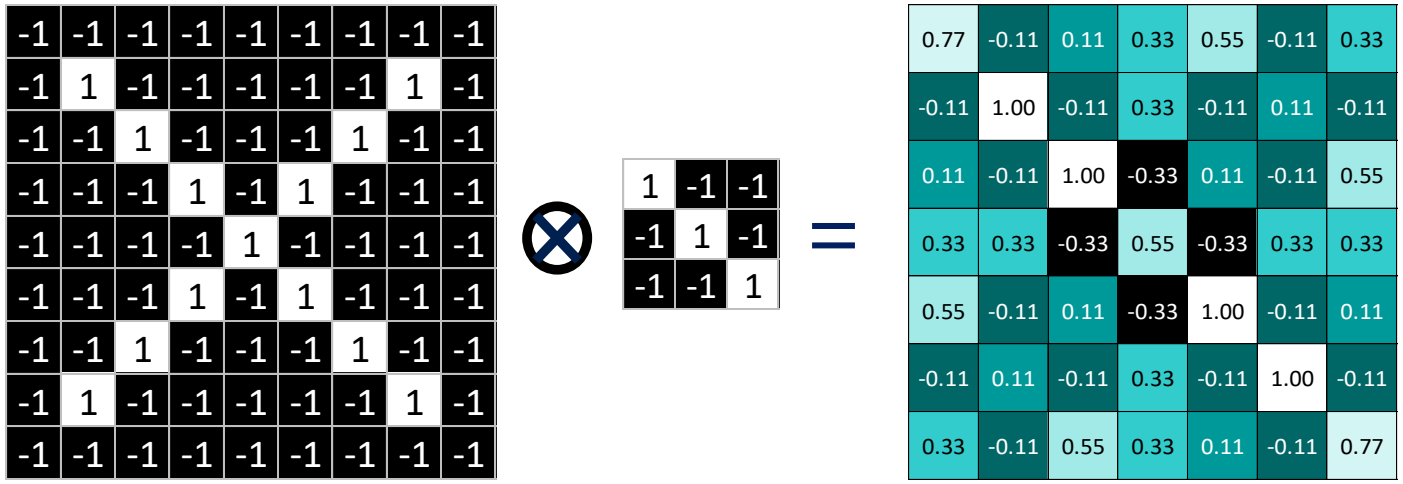
Convolution: Trying every possible match

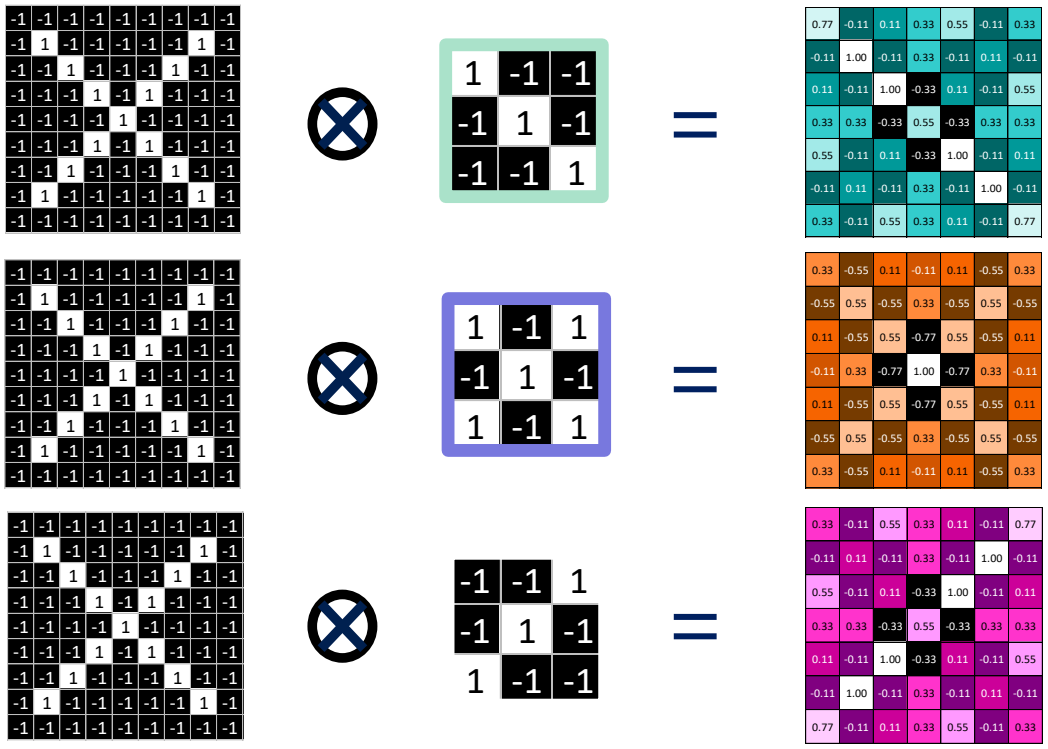
1	-1	-1
-1	1	-1
-1	-1	1

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

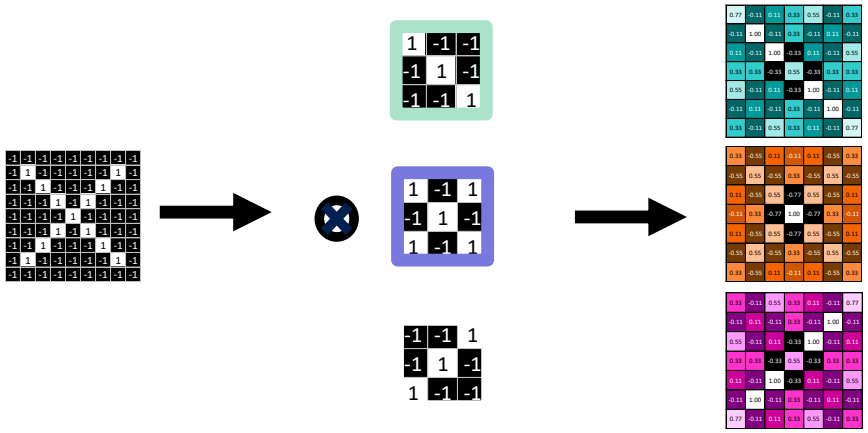
Convolution: Trying every possible match




-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 1 -1 -1 -1 -1 -1 1 -1
-1 -1 1 -1 -1 -1 1 -1 -1
-1 -1 -1 1 -1 1 -1 -1 -1
-1 -1 -1 -1 1 -1 -1 -1 -1
-1 -1 -1 1 -1 1 -1 -1 -1
-1 -1 1 -1 -1 -1 1 -1 -1
-1 1 -1 -1 -1 -1 -1 1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
1 -1 -1
-1 1 -1
-1 -1 1
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1 -1 1
-1 1 -1
1 -1 1
0.33 -0.55 0.11 -0.11 0.11 -0.55 0.33
-0.55 0.55 -0.55 0.33 -0.55 0.55 -0.55
0.11 -0.55 0.55 -0.77 0.55 -0.55 0.11
-0.11 0.33 -0.77 1.00 -0.77 0.33 -0.11
0.11 -0.55 0.55 -0.77 0.55 -0.55 0.11
-0.55 0.55 -0.55 0.33 -0.55 0.55 -0.55
0.33 -0.55 0.11 -0.11 0.11 -0.55 0.33
-1 -1 1
-1 1 -1
1 -1 -1
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33

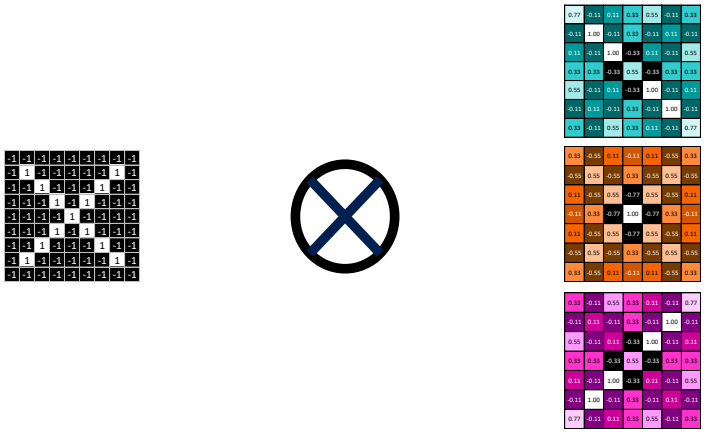
Convolution layer

One image becomes a stack of filtered images



Convolution layer

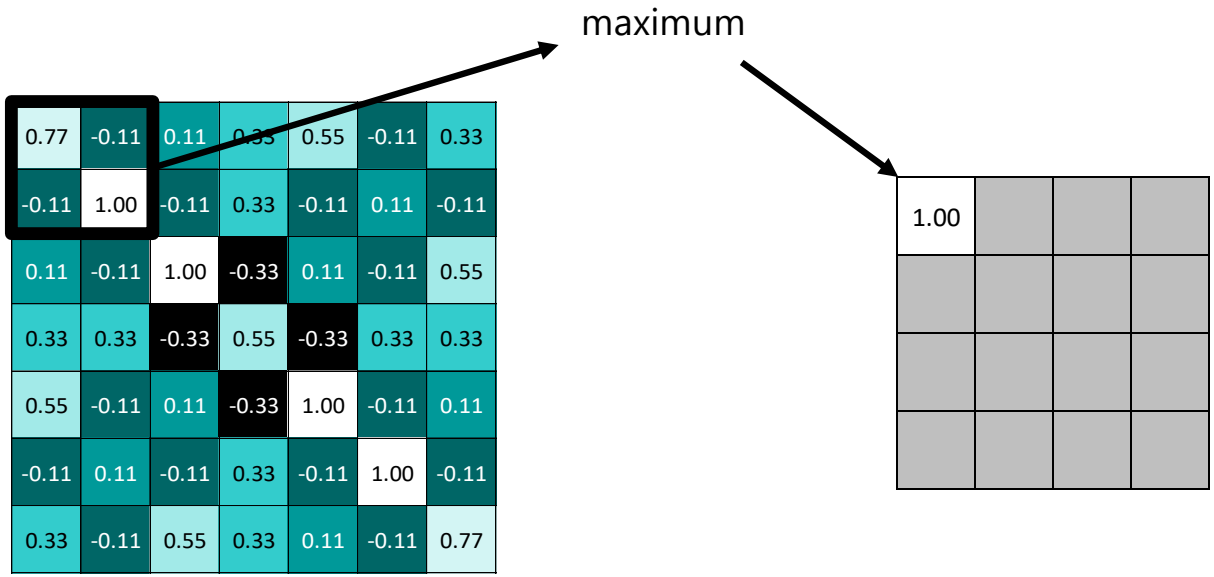
One image becomes a stack of filtered images



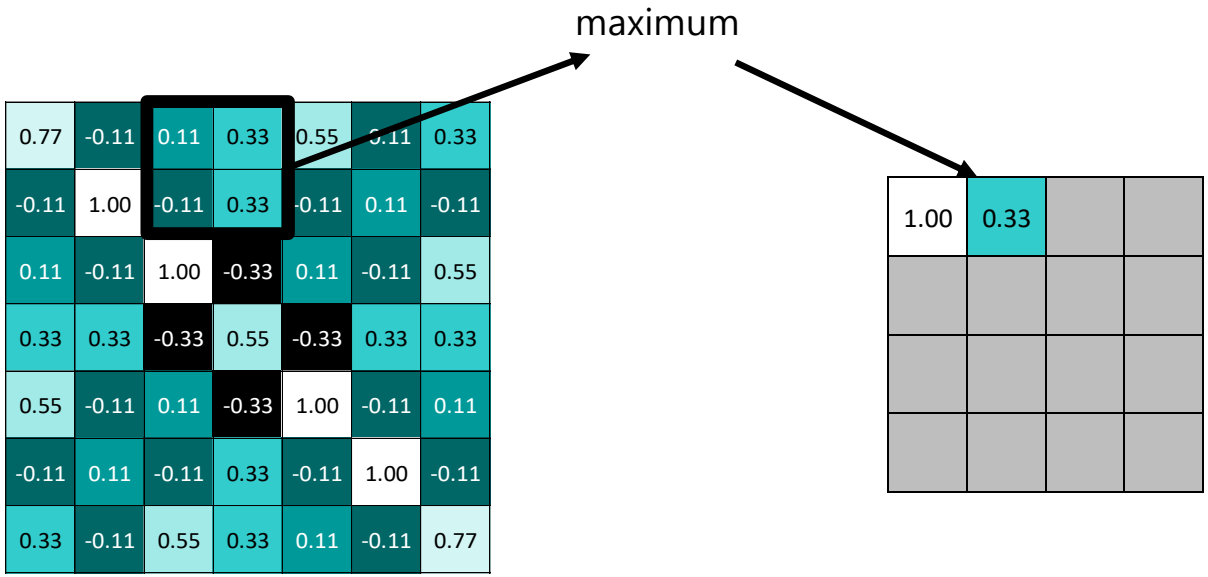
Pooling: Shrinking the image stack

1. Pick a window size (usually 2 or 3).
2. Pick a stride (usually 2).
3. Walk your window across your filtered images.
4. From each window, take the maximum value.

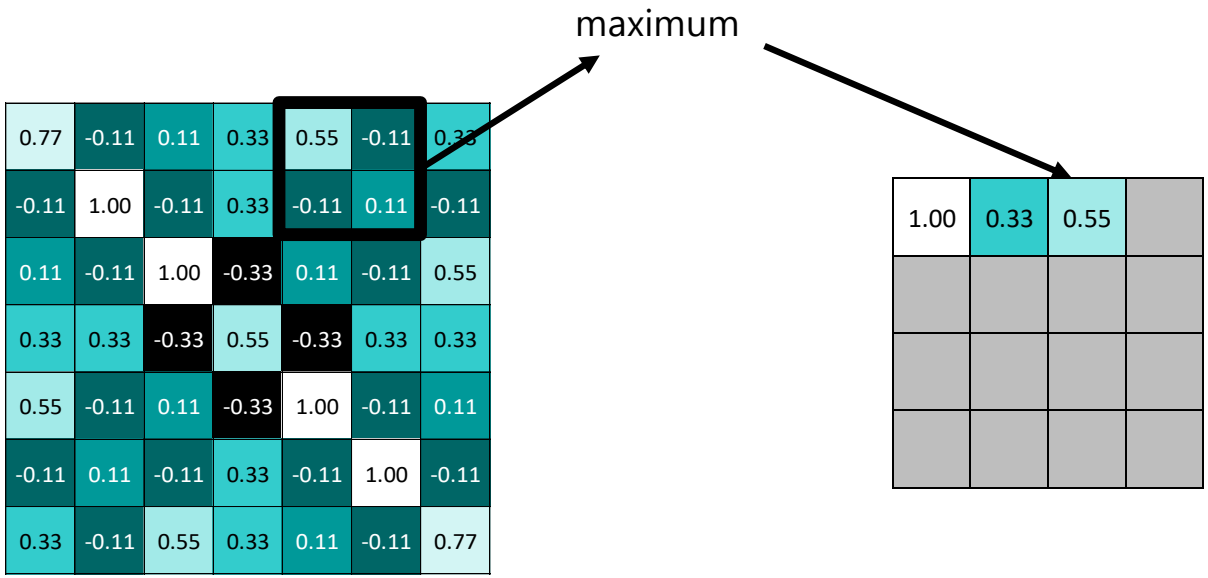
Pooling


maximum
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1.00

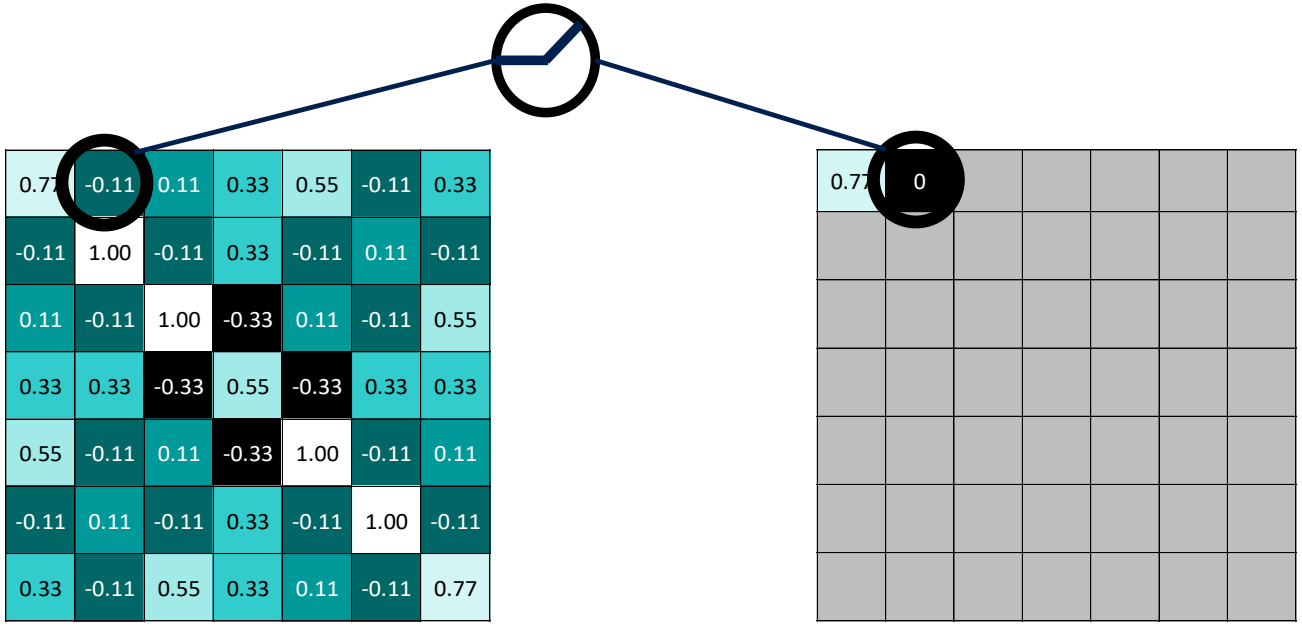
Pooling


maximum
0.77 -0.11 0.11 0.33 0.55 0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1.00 0.33

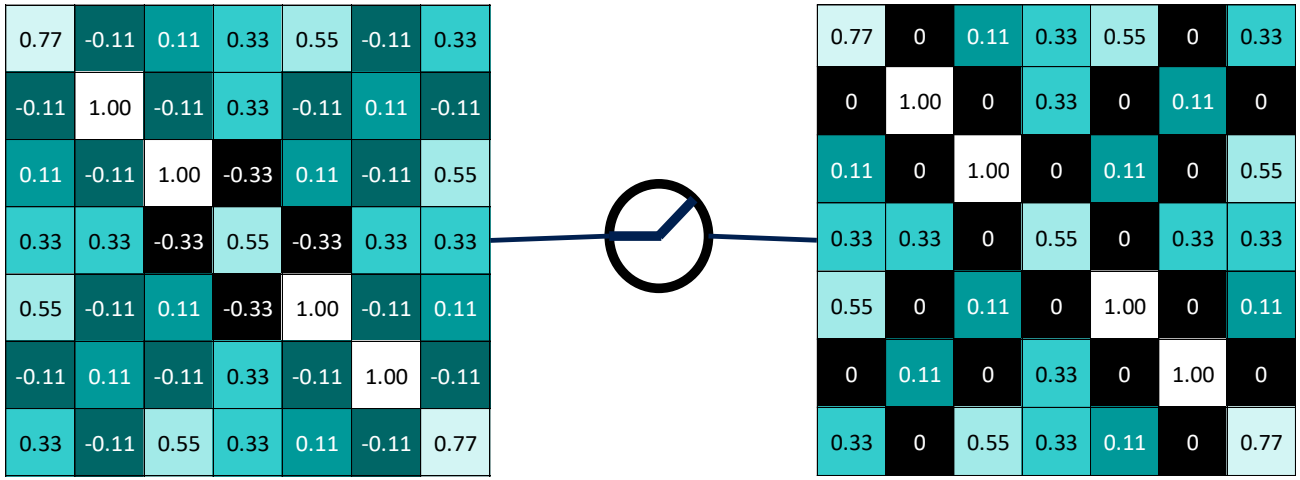
Pooling


maximum
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
1.00 0.33 0.55

Nonlinear Activation function: Rectified Linear Units (ReLUs)

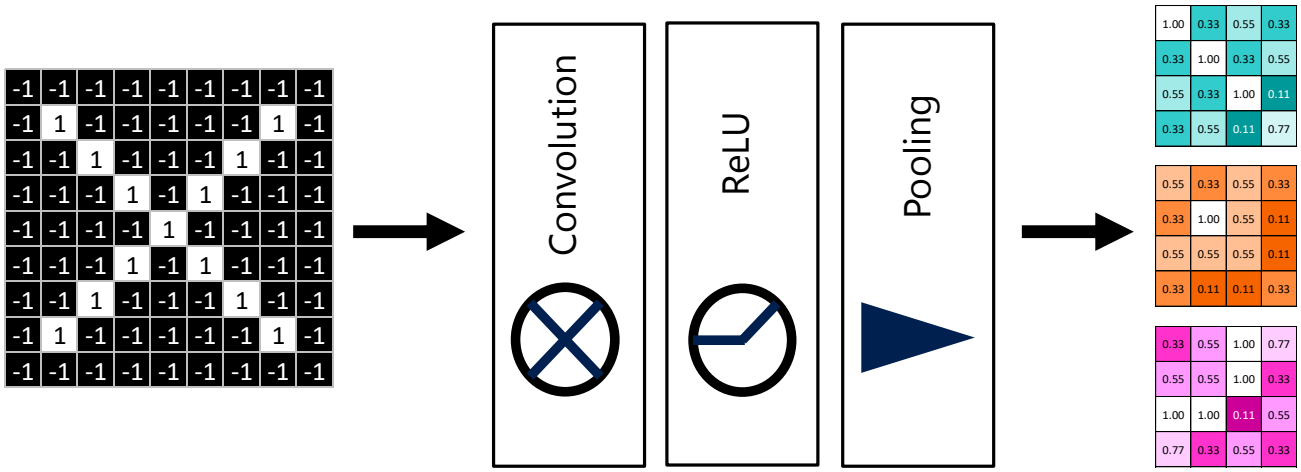


Rectified Linear Units (ReLUs)



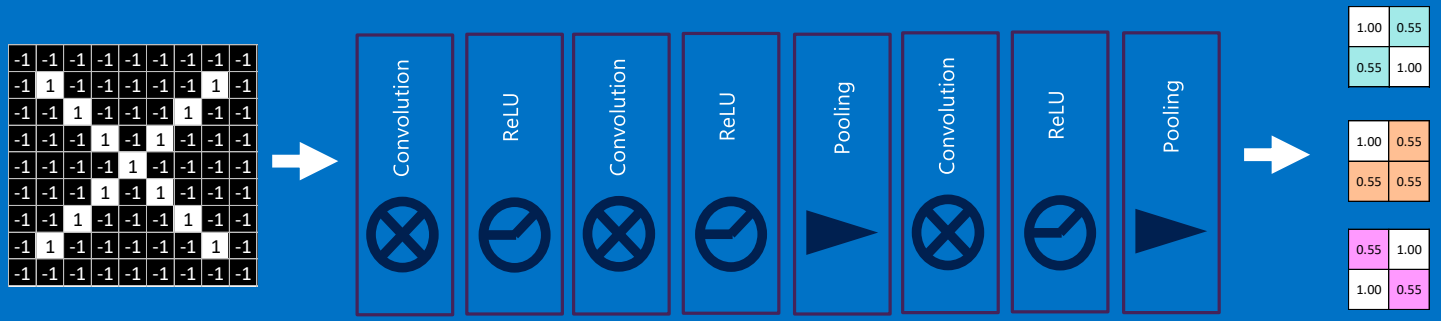
Layers get stacked

The output of one becomes the input of the next.



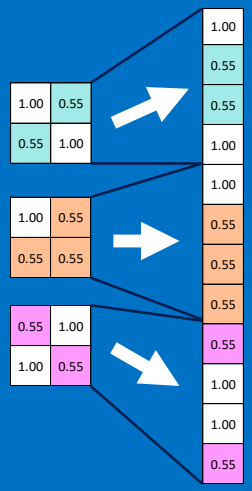
Deep stacking

Layers can be repeated several (or many) times.



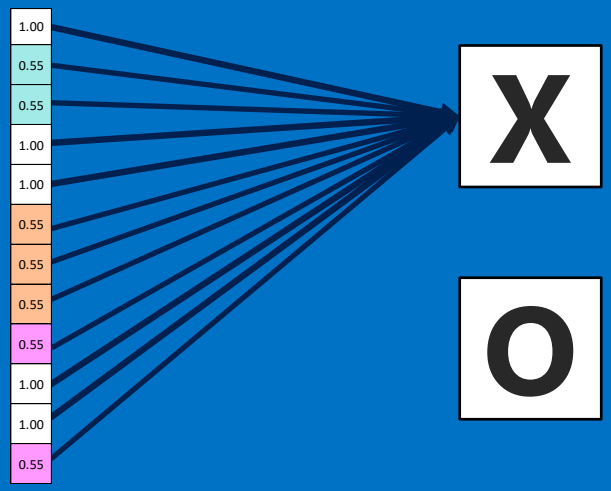
Fully connected layer

Every value gets a vote



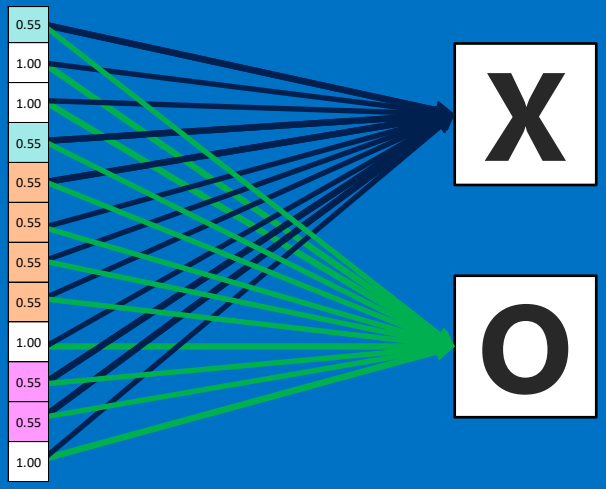
Fully connected layer

Vote depends on how strongly a value predicts X or O



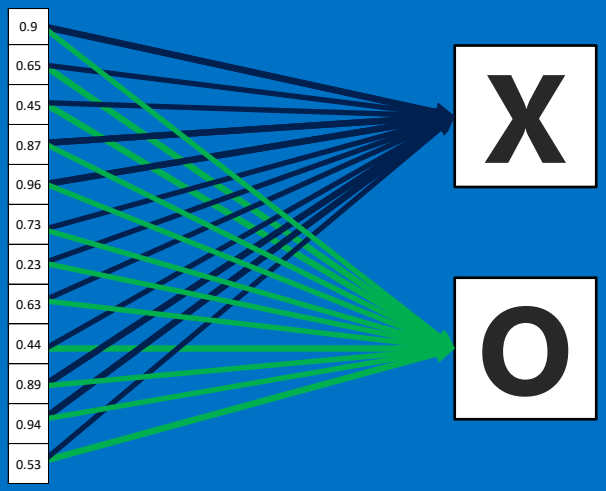
Fully connected layer

Vote depends on how strongly a value predicts X or O



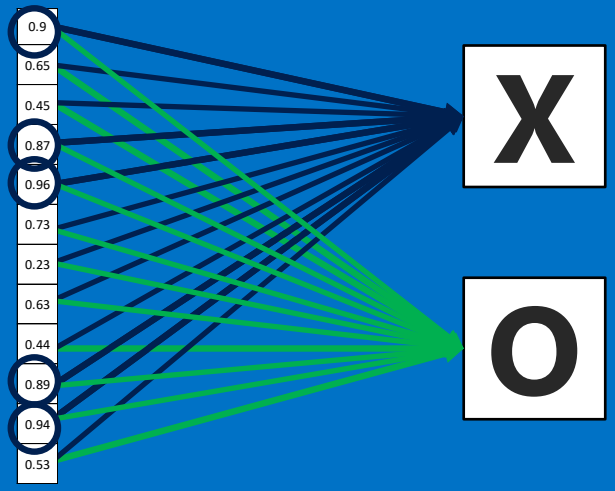
Fully connected layer

Future values vote on X or O



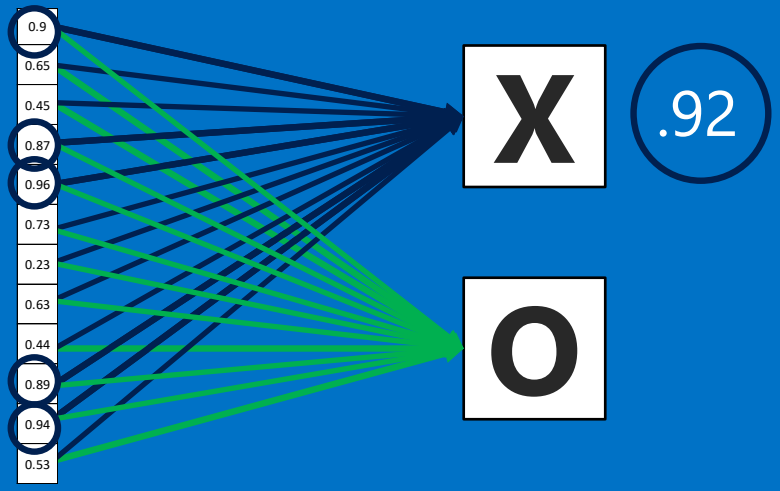
Fully connected layer

Future values vote on X or O



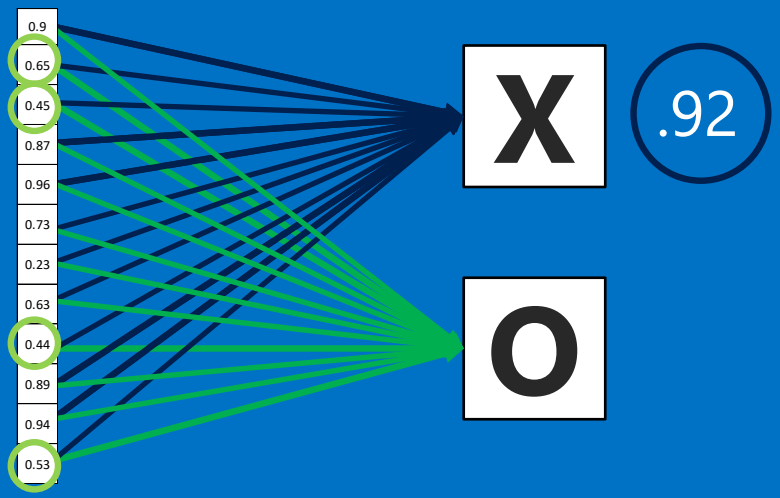
Fully connected layer

Future values vote on X or O



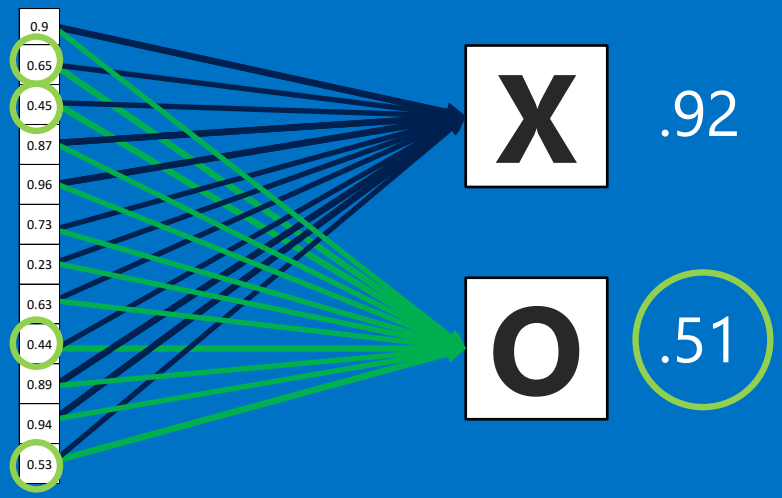
Fully connected layer

Future values vote on X or O



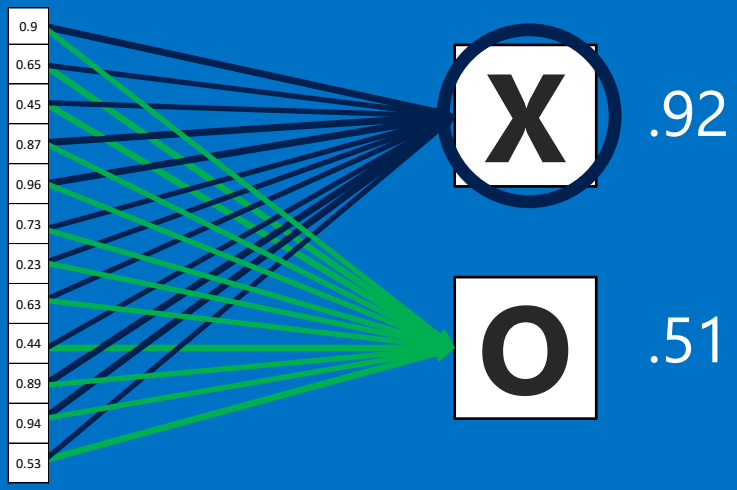
Fully connected layer

Future values vote on X or O



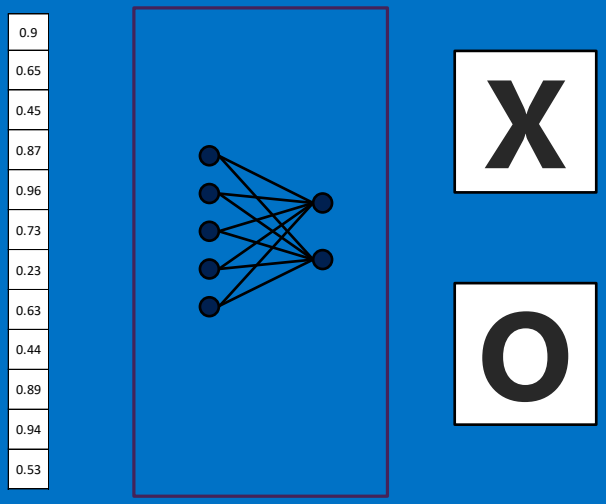
Fully connected layer

Future values vote on X or O



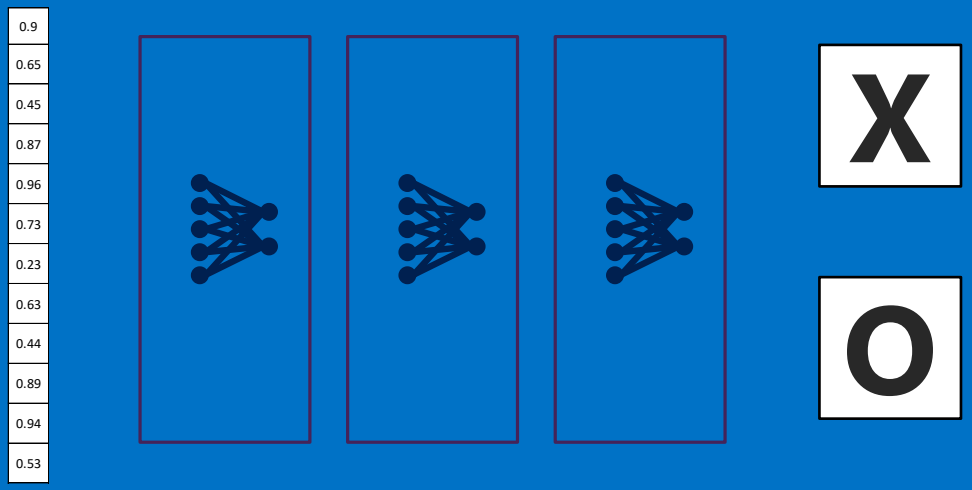
Fully connected layer

A list of feature values becomes a list of votes.



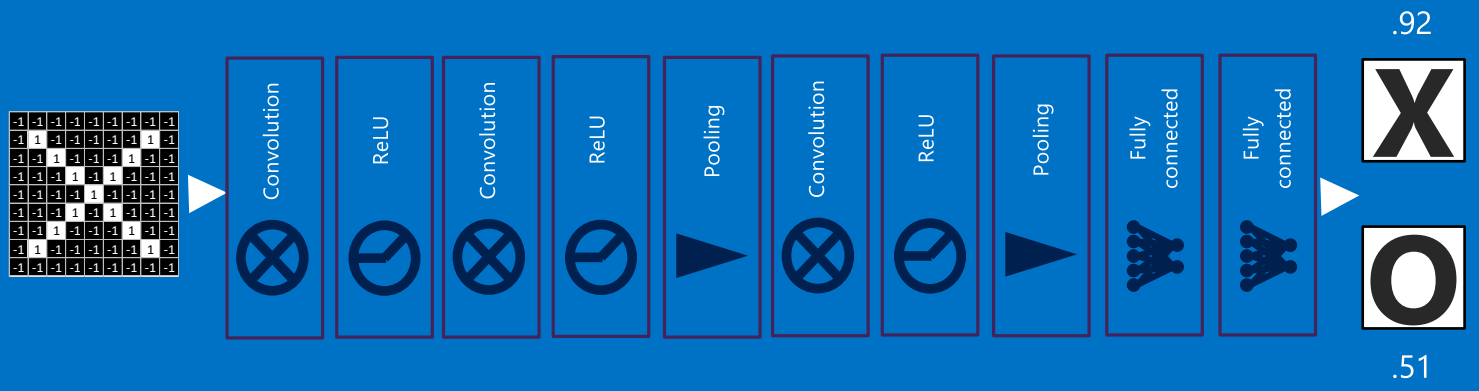
Fully connected layer

These can also be stacked.



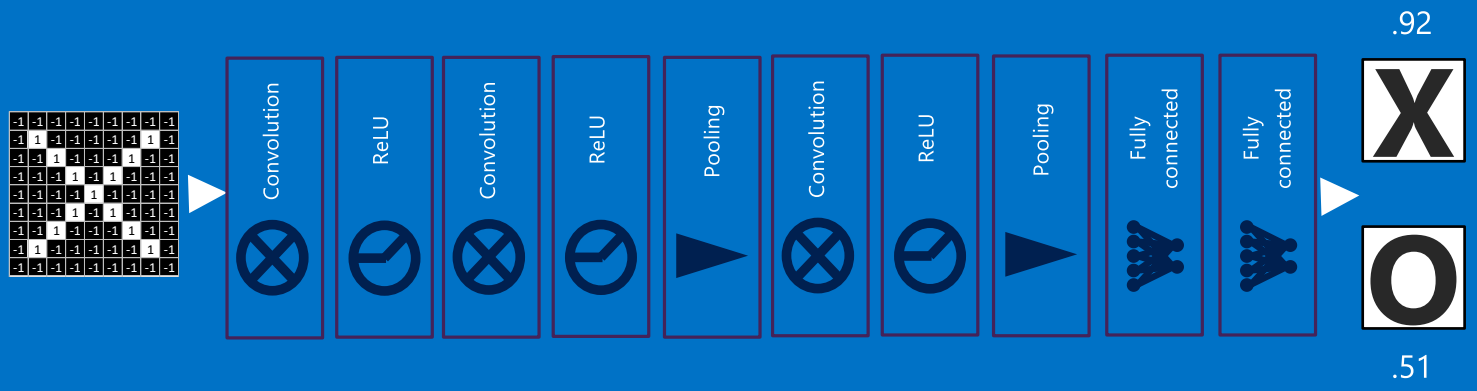
Putting it all together

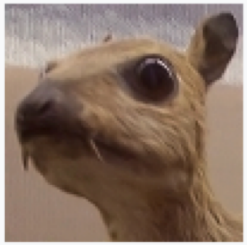
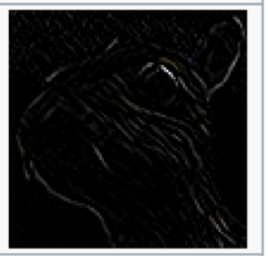
A set of pixels becomes a set of votes.

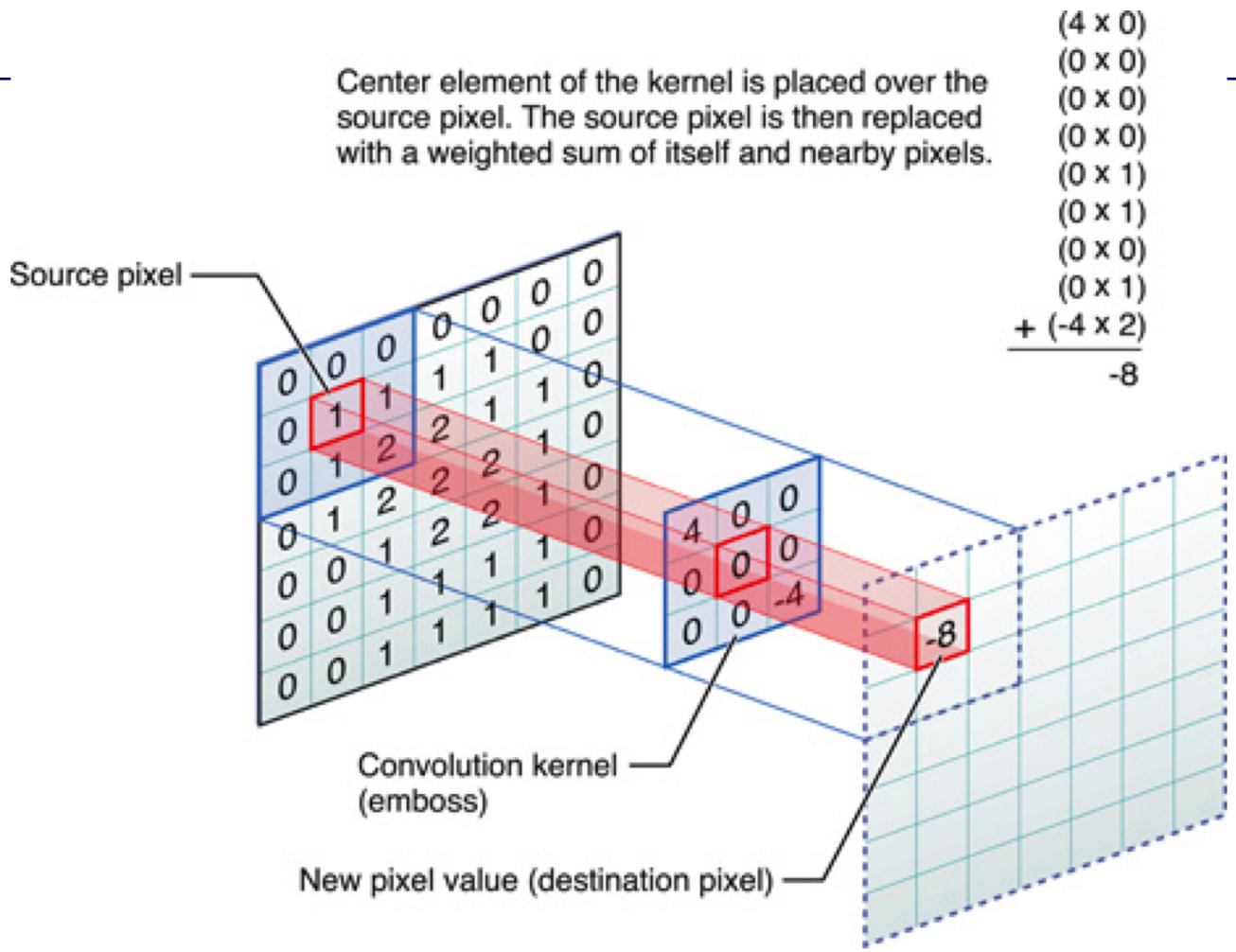


Backprop (gradient descent) to learn weights/kernels

	Right answer	Actual answer	Error
X	1	0.92	0.08
O	0	0.51	0.49
		Total	0.57



Operation	Kernel	Image result
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)