Neuronal Networks

What Is Machine Learning?

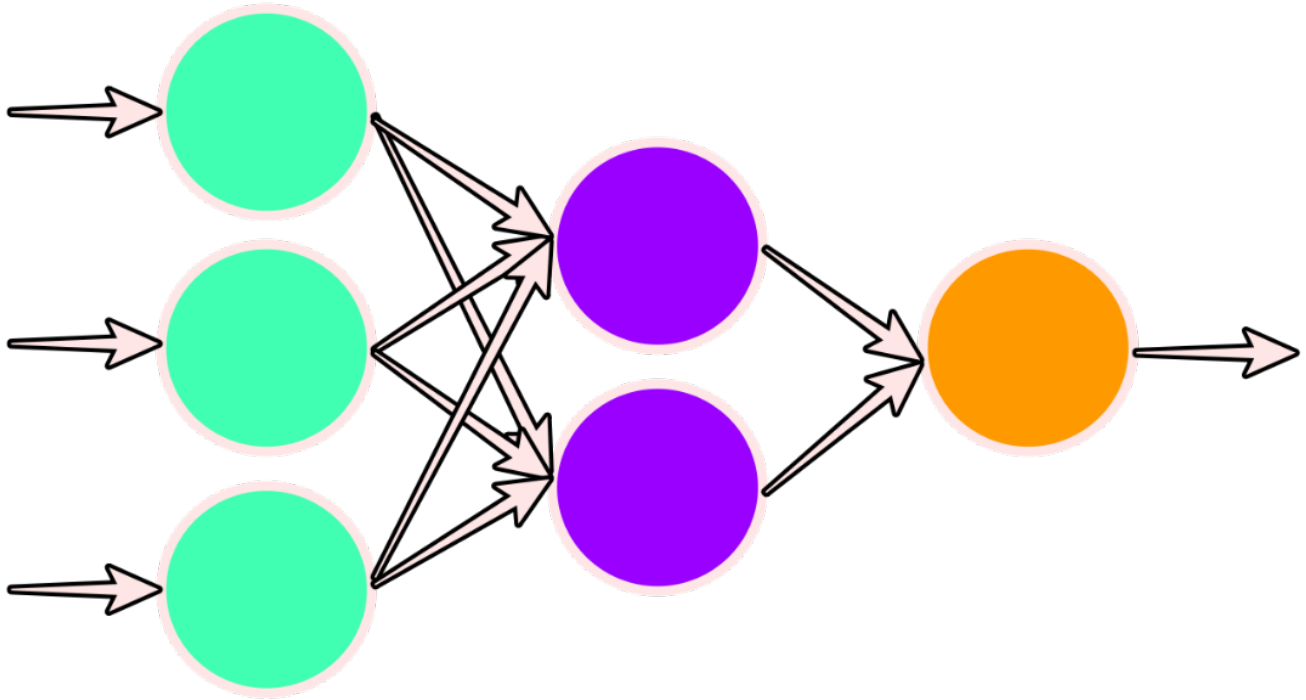
Machine learning is the practice of using algorithms to analyze data, learn from that data, and then make a determination or prediction about new data.

```
// pseudocode
let positive = [
   "happy",
   "thankful",
   "amazing"
];

let negative = [
   "can't",
   "won't",
   "sorry",
   "unfortunately"
];
```

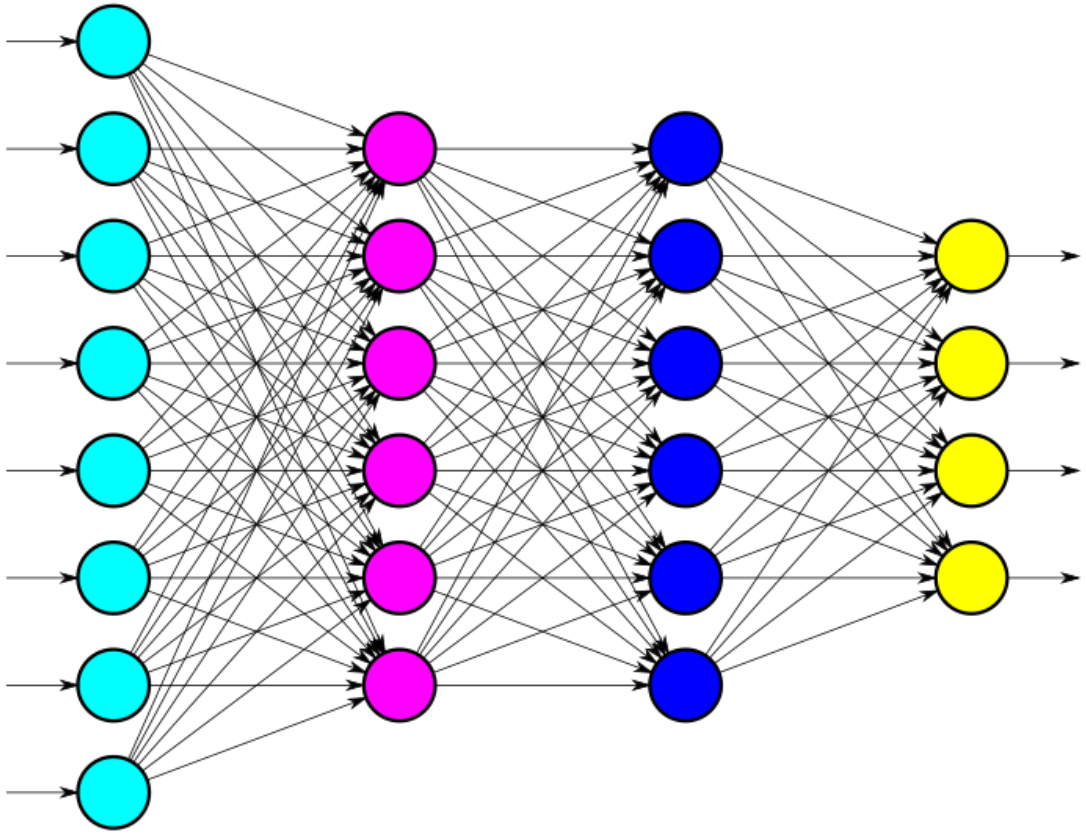
```
// pseudocode
let articles = [
   {
      label: "positive",
      data: "The lizard movie was great! I really liked..."
   },
   {
      label: "positive",
      data: "Awesome lizards! The color green is my fav..."
   },
   {
      label: "negative",
      data: "Total disaster! I never liked..."
   },
   {
      label: "negative",
      data: "Worst movie of all time!..."
   }
];
```

What Is Deep Learning?

Deep learning is a sub-field of machine learning that uses algorithms inspired by the structure and function of the brain's neural networks.

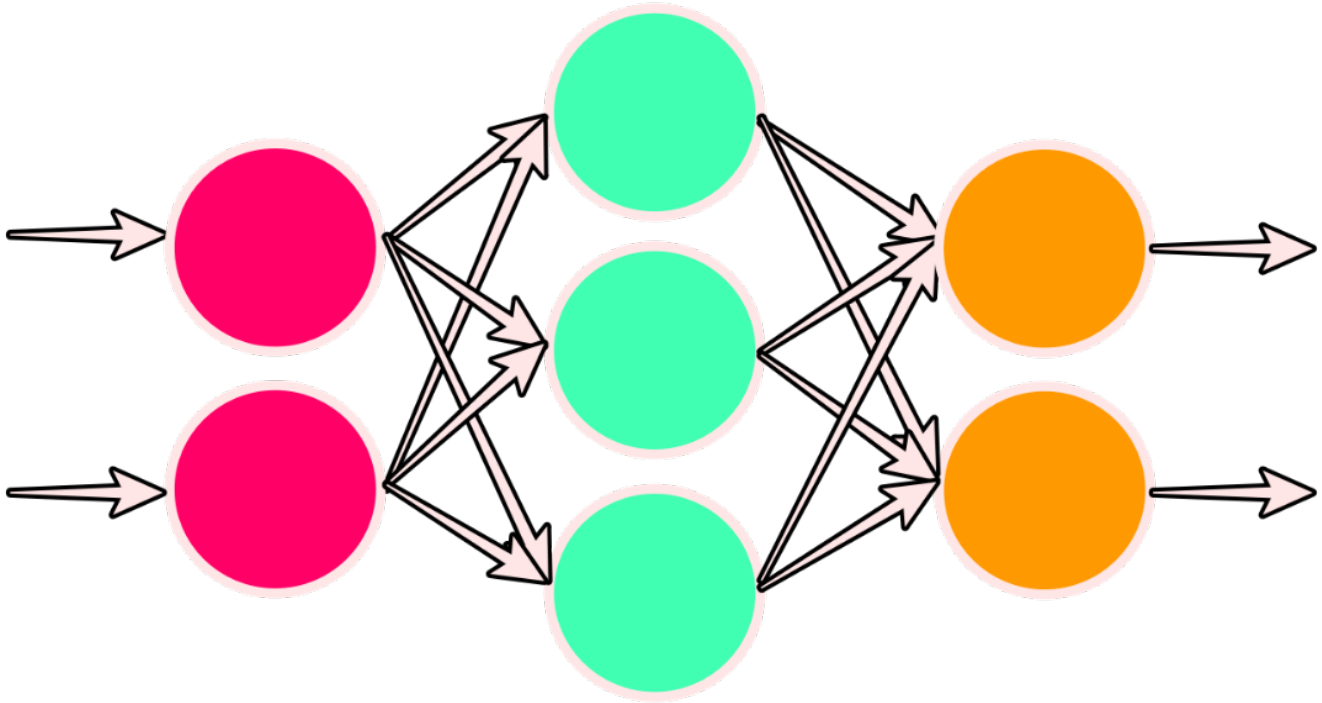


What Is Deep Learning?



1. ANNs are built using what we call neurons.
2. Neurons in an ANN are organized into what we call layers.
3. Layers *within* an ANN (all but the input and output layers) are called hidden layers.
4. If an ANN has more than one hidden layer, the ANN is said to be a deep ANN.

An Artificial neuronal network



1.Input layer (left): 2 nodes
2.Hidden layer (middle): 3 nodes
3.Output layer (right): 2 nodes

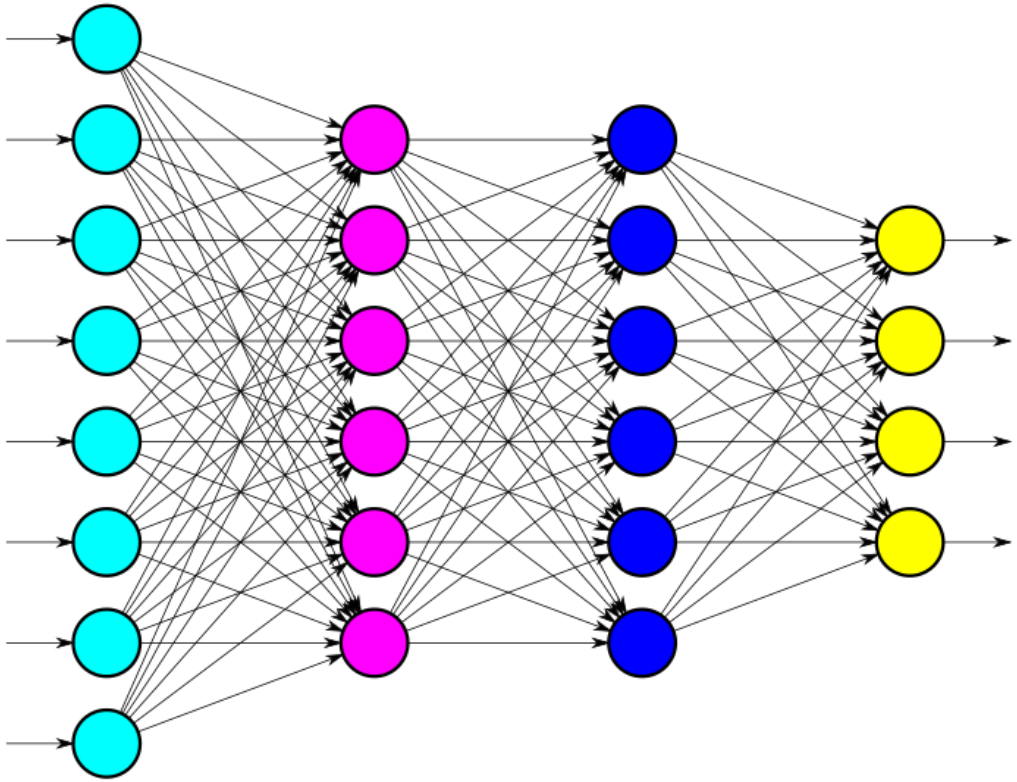
Implementation in Keras:

[7]:
```
# Keras is now a part of tensorflow. Tensorflow is a machine learning platform
import tensorflow
import keras
```

[8]:
```
from keras.models import Sequential
from keras.layers import Dense, Activation
```

[11]:
```
# layers will be used in the instatiation of object Sequential
layers = [
    Dense(units=3, input_shape=(2,), activation='relu'),
    Dense(units=2, activation='softmax')
]

model = Sequential(layers)
```

- Dense (or fully connected) layers
- Convolutional layers
- Pooling layers
- Recurrent layers
- Normalization layers



```
layers = [
    Dense(units=6, input_shape=(8,), activation='relu'),
    Dense(units=6, activation='relu'),
    Dense(units=4, activation='softmax')
]
```

Layer Weights

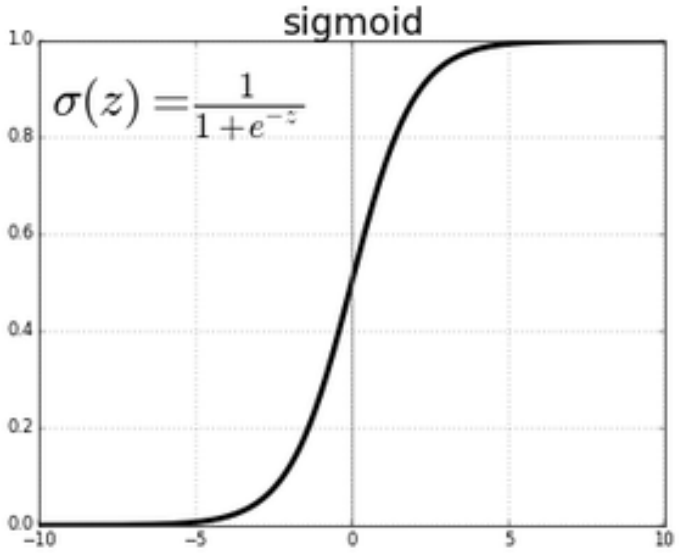
node output = activation(weighted sum of inputs)

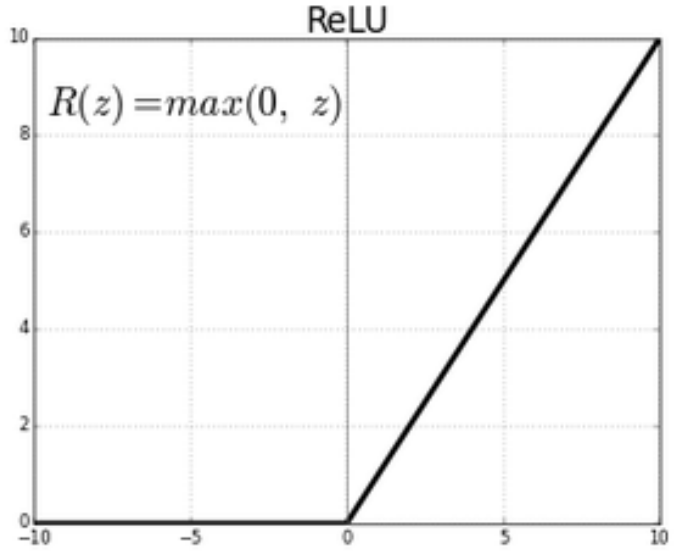
Learning: Finding The Optimal Weights

Activation Functions In A Neural Network

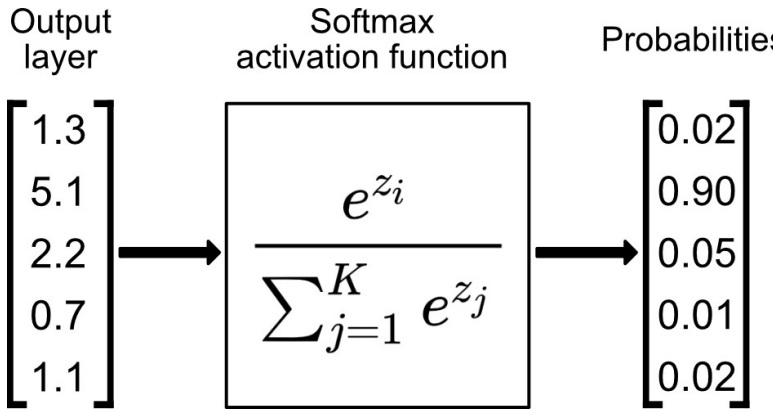
function that maps a node's inputs to its corresponding output.

node output = activation(weighted sum of inputs)





node output = relu(weighted sum of inputs)

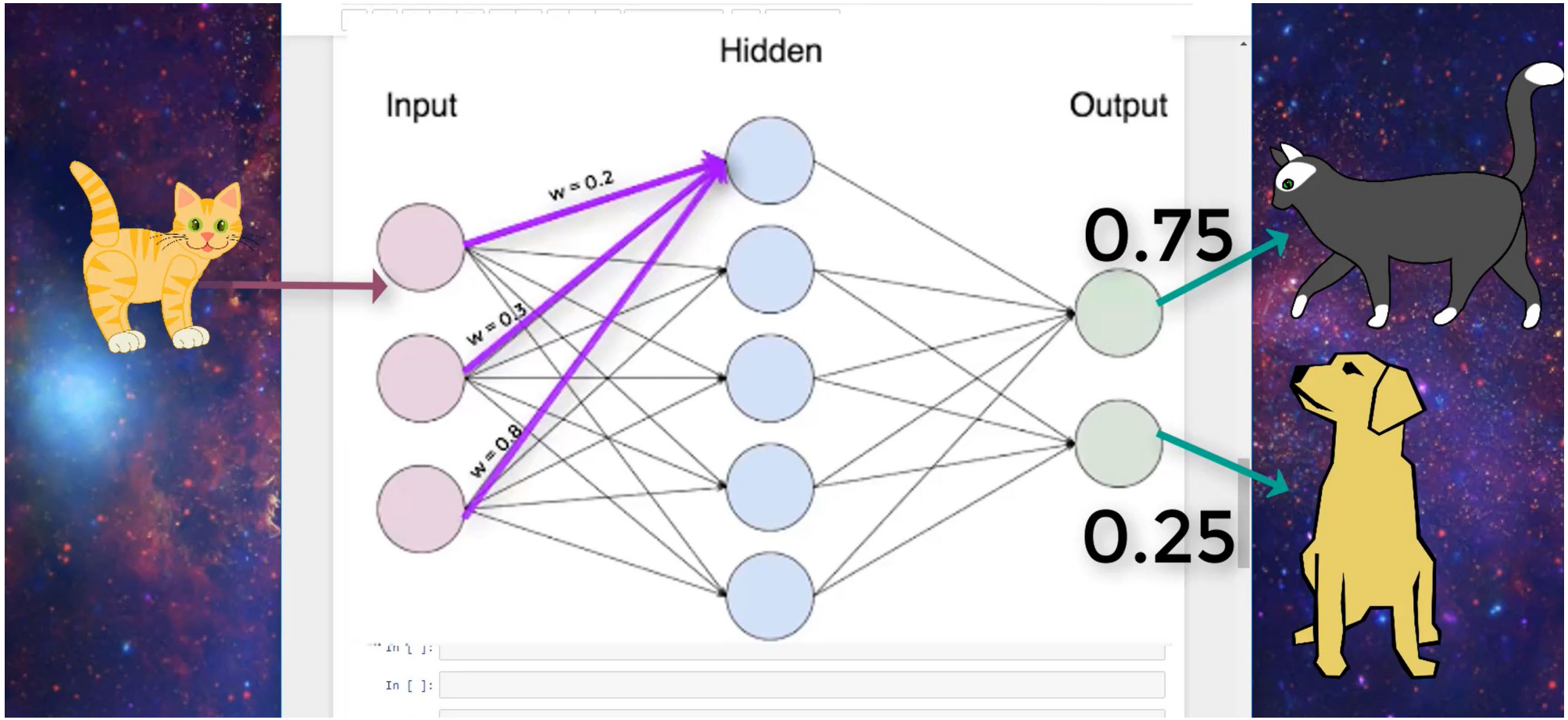


Training An Artificial Neural Network

- Provide input AND output, optimize weights (fit) to best account for all the training

Optimization Algorithm: Often Stochastic Gradient descent

Loss Function: Is the measure against which network is optimized.



Learning In Artificial Neural Networks

Gradient Of The Loss Function

```
[*]: import keras
     from keras.models import Sequential
     from keras.layers import Activation
     from keras.layers.core import Dense
     from keras.optimizers import Adam
     from keras.metrics import categorical_crossentropy
```

```
[*]: model = Sequential([
         Dense(units=16, input_shape=(1,), activation='relu'),
         Dense(units=32, activation='relu'),
         Dense(units=2, activation='sigmoid')
     ])
```

```
[*]: model.compile(
         optimizer=Adam(learning_rate=0.0001),
         loss='sparse_categorical_crossentropy',
         metrics=['accuracy']
     )
```

```
[*]: model.fit(
         x=scaled_train_samples,
         y=train_labels,
         batch_size=10,
         epochs=20,
         shuffle=True,
         verbose=2
     )
```

The learning rate tells us how large of a step we should take in the direction of the minimum.

new weight = old weight - (learning rate * gradient)

```
Epoch 1/20 0s - loss: 0.6400 - acc: 0.5576
Epoch 2/20 0s - loss: 0.6061 - acc: 0.6310
Epoch 3/20 0s - loss: 0.5748 - acc: 0.7010
Epoch 4/20 0s - loss: 0.5401 - acc: 0.7633
Epoch 5/20 0s - loss: 0.5050 - acc: 0.7990
Epoch 6/20 0s - loss: 0.4702 - acc: 0.8300
Epoch 7/20 0s - loss: 0.4366 - acc: 0.8495
Epoch 8/20 0s - loss: 0.4066 - acc: 0.8767
Epoch 9/20 0s - loss: 0.3808 - acc: 0.8814
Epoch 10/20 0s - loss: 0.3596 - acc: 0.8962
Epoch 11/20 0s - loss: 0.3420 - acc: 0.9043
Epoch 12/20 0s - loss: 0.3282 - acc: 0.9090
Epoch 13/20 0s - loss: 0.3170 - acc: 0.9129
Epoch 14/20 0s - loss: 0.3081 - acc: 0.9210
Epoch 15/20 0s - loss: 0.3014 - acc: 0.9190
Epoch 16/20 0s - loss: 0.2959 - acc: 0.9205
Epoch 17/20 0s - loss: 0.2916 - acc: 0.9238
Epoch 18/20 0s - loss: 0.2879 - acc: 0.9267
Epoch 19/20 0s - loss: 0.2848 - acc: 0.9252
Epoch 20/20 0s - loss: 0.2824 - acc: 0.9286
```

Loss Functions In Neural Networks

MSE(input) = (output - label)(output - label)

- We use sparse_categorical_crossentropy $$CE = -\sum_{neuron=1}^{classes} y_{true_{neuron}} * \ln\left(y_{pred_{neuron}}\right)$$

But many available:

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity

Introducing The Learning Rate

new weight = old weight - (learning rate * gradient)

- Typically 0.0001 – 0.01
- Too large -> Oscillations,
- Too low, slow convergence

Train, Test, & Validation Sets Explained

The training set is what it sounds like. It's the set of data used to train the model.

The test set provides a final check that the model is generalizing well before deploying the model to production.

The validation set allows us to see how well the model is generalizing during training.

Dataset	Updates Weights	Description
Training set	Yes	Used to train the model. The goal of training is to fit the model to the training set while still generalizing to unseen data.
Validation set	No	Used during training to check how well the model is generalizing.
Test set	No	Used to test the model's final ability to generalize before deploying to production.

Using A Keras Model To Get A Prediction

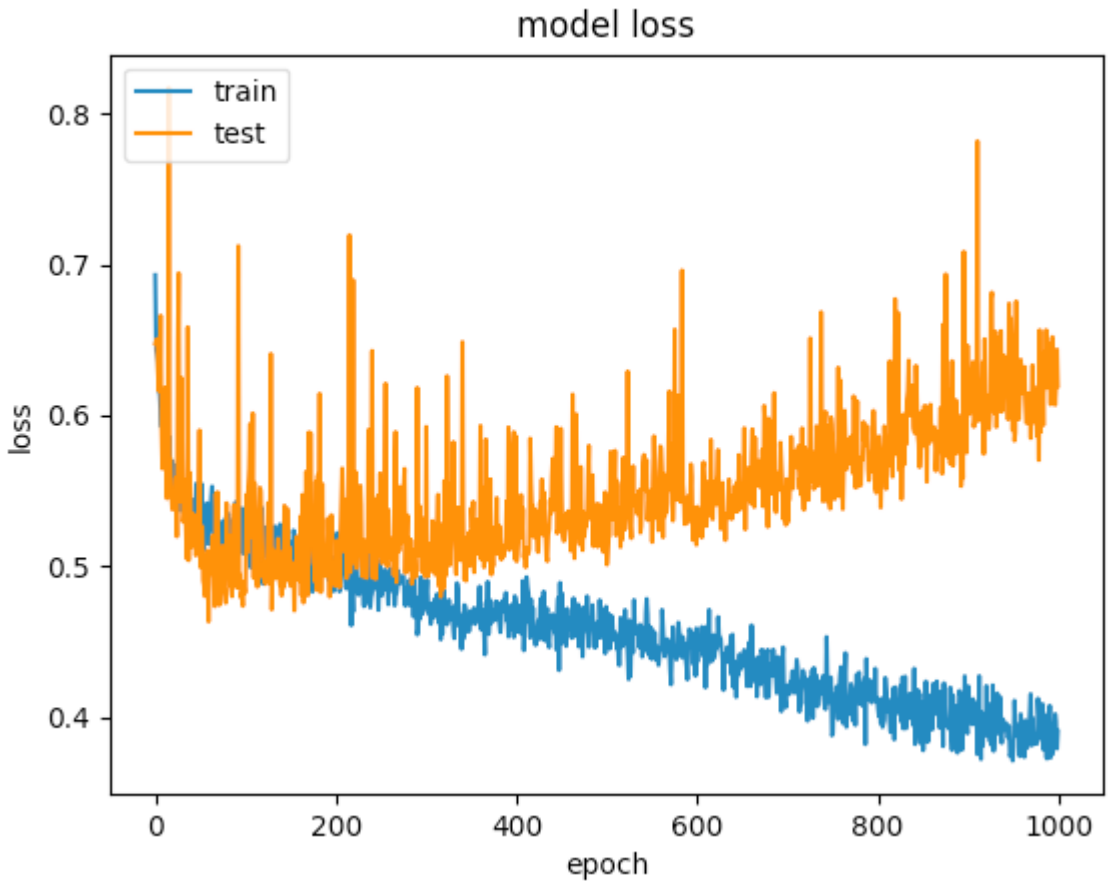
```
predictions = model.predict(
    x=scaled_test_samples,
    batch_size=10,
    verbose=0
)
```

```
for p in predictions:
        print(p)
```

```
[ 0.7410683 0.2589317]
[ 0.14958295 0.85041702]
...
[ 0.87152088 0.12847912]
[ 0.04943148 0.95056852]
```

Overfitting In A Neural Network

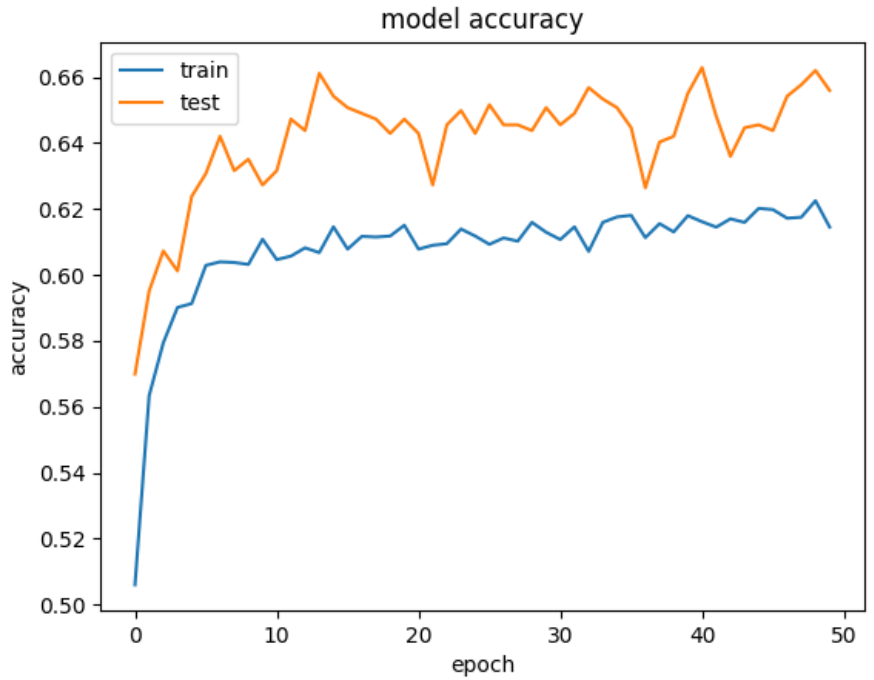
```
model.fit(scaled_train_samples, train_labels, validation_split = 0.20, ba
          epochs=20, shuffle=True, verbose=2)

Train on 1680 samples, validate on 420 samples
Epoch 1/20
0s - loss: 0.6994 - acc: 0.4970 - val_loss: 0.6960 - val_acc: 0.5000
Epoch 2/20
0s - loss: 0.6906 - acc: 0.5774 - val_loss: 0.6815 - val_acc: 0.6952
Epoch 3/20
0s - loss: 0.6754 - acc: 0.7179 - val_loss: 0.6613 - val_acc: 0.7857
Epoch 4/20
0s - loss: 0.6548 - acc: 0.7720 - val_loss: 0.6341 - val_acc: 0.8333
Epoch 5/20
0s - loss: 0.6296 - acc: 0.7958 - val_loss: 0.6001 - val_acc: 0.8571
Epoch 6/20
0s - loss: 0.5951 - acc: 0.8161 - val_loss: 0.5516 - val_acc: 0.8714
Epoch 7/20
0s - loss: 0.5545 - acc: 0.8250 - val_loss: 0.5004 - val_acc: 0.8881
Epoch 8/20
0s - loss: 0.5091 - acc: 0.8446 - val_loss: 0.4400 - val_acc: 0.9119
Epoch 9/20
0s - loss: 0.4637 - acc: 0.8726 - val_loss: 0.3886 - val_acc: 0.9310
Epoch 10/20
0s - loss: 0.4283 - acc: 0.8798 - val_loss: 0.3454 - val_acc: 0.9381
Epoch 11/20
0s - loss: 0.3997 - acc: 0.8863 - val_loss: 0.3096 - val_acc: 0.9524
Epoch 12/20
0s - loss: 0.3776 - acc: 0.8917 - val_loss: 0.2805 - val_acc: 0.9524
Epoch 13/20
0s - loss: 0.3602 - acc: 0.8988 - val_loss: 0.2572 - val_acc: 0.9643
Epoch 14/20
0s - loss: 0.3467 - acc: 0.9083 - val_loss: 0.2369 - val_acc: 0.9643
Epoch 15/20
0s - loss: 0.3364 - acc: 0.9113 - val_loss: 0.2214 - val_acc: 0.9643
Epoch 16/20
0s - loss: 0.3288 - acc: 0.9119 - val_loss: 0.2083 - val_acc: 0.9714
Epoch 17/20
0s - loss: 0.3227 - acc: 0.9155 - val_loss: 0.1971 - val_acc: 0.9643
Epoch 18/20
0s - loss: 0.3182 - acc: 0.9137 - val_loss: 0.1889 - val_acc: 0.9786
Epoch 19/20
0s - loss: 0.3144 - acc: 0.9167 - val_loss: 0.1809 - val_acc: 0.9714
Epoch 20/20
0s - loss: 0.3115 - acc: 0.9161 - val_loss: 0.1749 - val_acc: 0.9714
```



Underfitting In A Neural Network Explained

```
model.fit_generator(train_batches, steps_per_epoch=4,
                    validation_data=valid_batches, validation_steps=4, epochs=5, verbose=2)
```

```
Epoch 1/5
6s - loss: 8.7262 - acc: 0.4500 - val_loss: 8.0590 - val_acc: 0.5000
Epoch 2/5
5s - loss: 8.0590 - acc: 0.5000 - val_loss: 11.0812 - val_acc: 0.3125
Epoch 3/5
3s - loss: 8.0590 - acc: 0.5000 - val_loss: 8.0590 - val_acc: 0.5000
Epoch 4/5
2s - loss: 8.0590 - acc: 0.5000 - val_loss: 9.0664 - val_acc: 0.4375
Epoch 5/5
4s - loss: 8.0590 - acc: 0.5000 - val_loss: 7.0517 - val_acc: 0.5625
```



Supervised Learning For Machine Learning

```
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

In [9]:
```
# weight, height
train_samples = np.array([
    [150, 67],
    [130, 60],
    [200, 65],
    [125, 52],
    [230, 72],
    [181, 70]
])
```

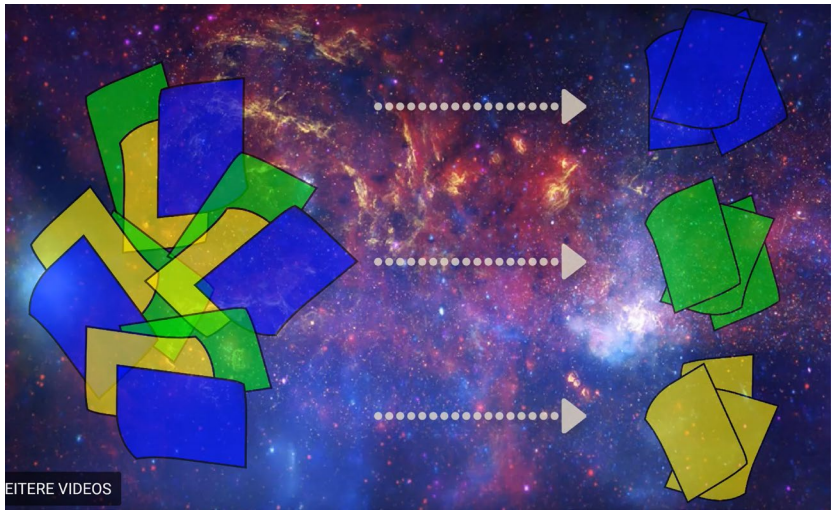
In [10]:
```
# 0: male
# 1: female
train_labels = np.array([1, 1, 0, 1, 0, 0])
```

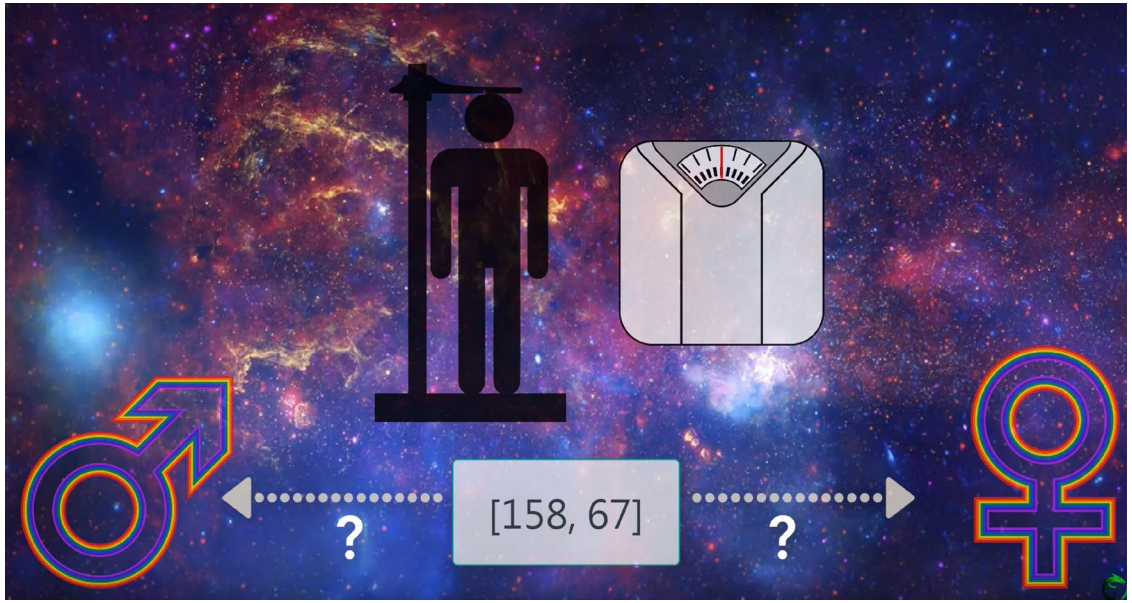
In [11]:
```
model.fit(
    x=train_samples,
    y=train_labels,
    batch_size=3,
    epochs=10,
    shuffle=True,
    verbose=2
)
```

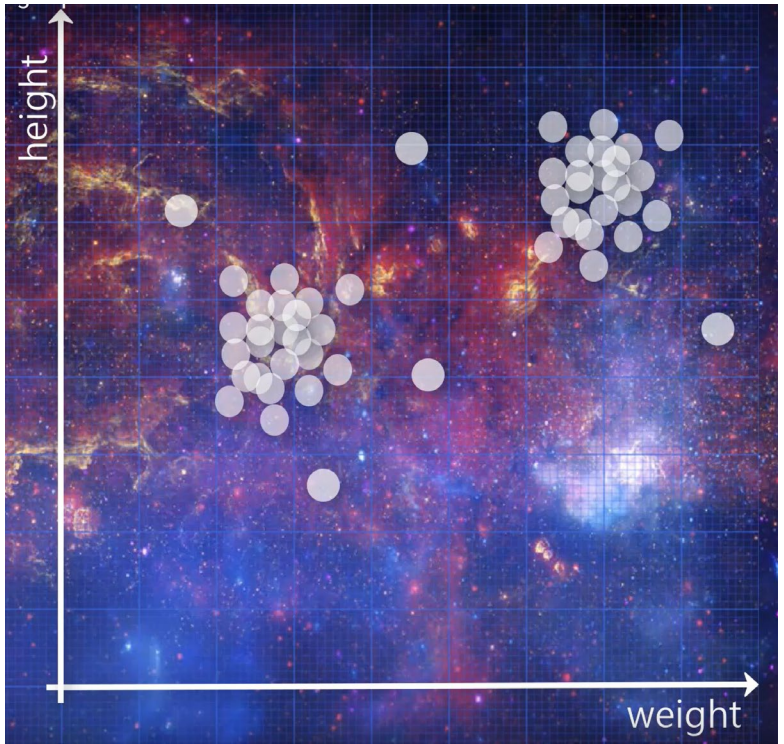
```
Epoch 1/10
2/2 - 0s - loss: 1.0512 - accuracy: 0.5000
Epoch 2/10
2/2 - 0s - loss: 1.0362 - accuracy: 0.5000
Epoch 3/10
2/2 - 0s - loss: 0.9790 - accuracy: 0.5000
Epoch 4/10
2/2 - 0s - loss: 0.9555 - accuracy: 0.5000
Epoch 5/10
2/2 - 0s - loss: 0.9416 - accuracy: 0.5000
Epoch 6/10
2/2 - 0s - loss: 0.9212 - accuracy: 0.5000
Epoch 7/10
2/2 - 0s - loss: 0.9009 - accuracy: 0.5000
Epoch 8/10
2/2 - 0s - loss: 0.8747 - accuracy: 0.5000
Epoch 9/10
2/2 - 0s - loss: 0.8615 - accuracy: 0.5000
Epoch 10/10
2/2 - 0s - loss: 0.8532 - accuracy: 0.5000
```

Out[11]: <tensorflow.python.keras.callbacks.History at 0x14b7fc7c2b0>

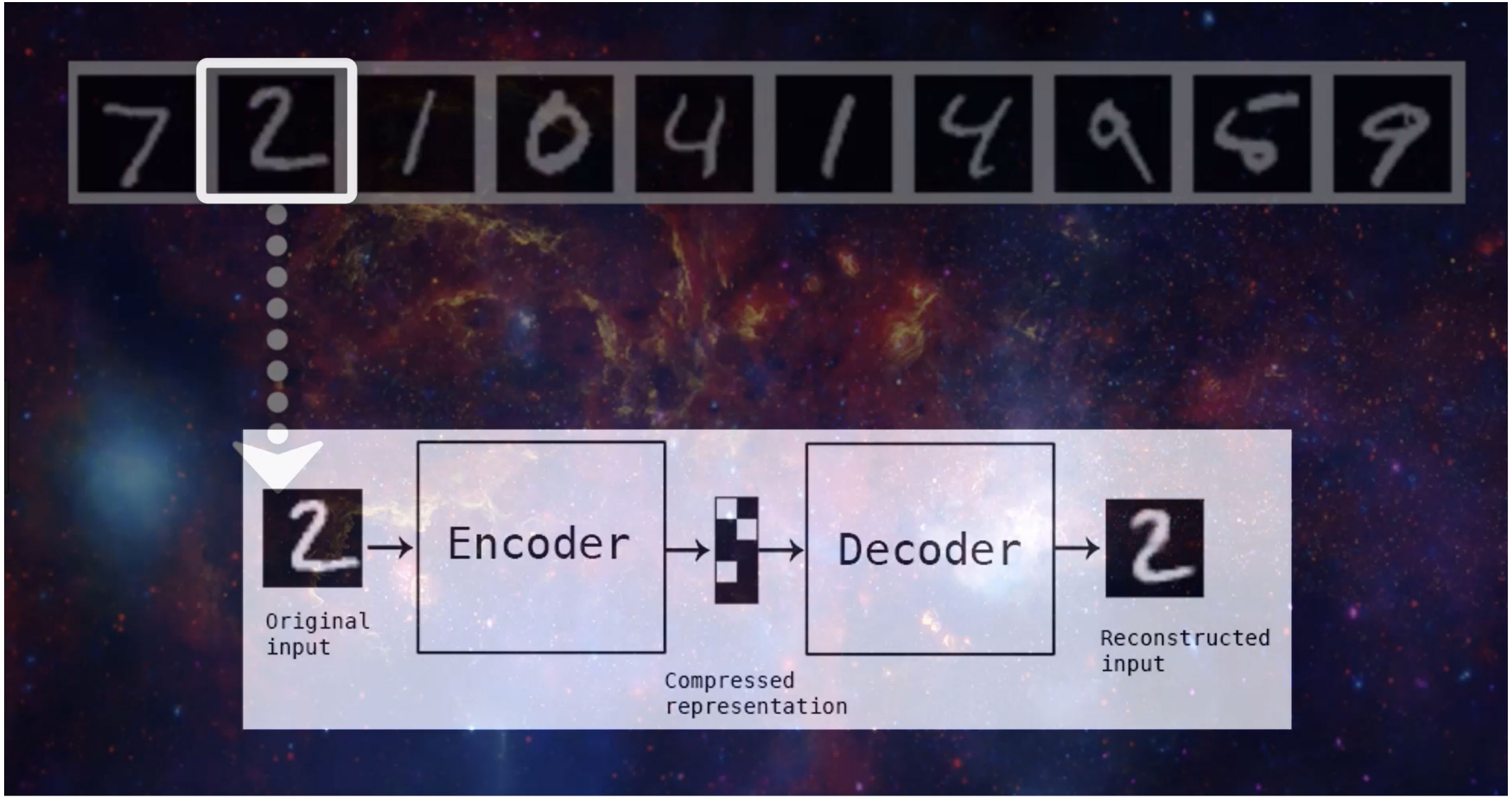
Unsupervised Learning For Machine Learning



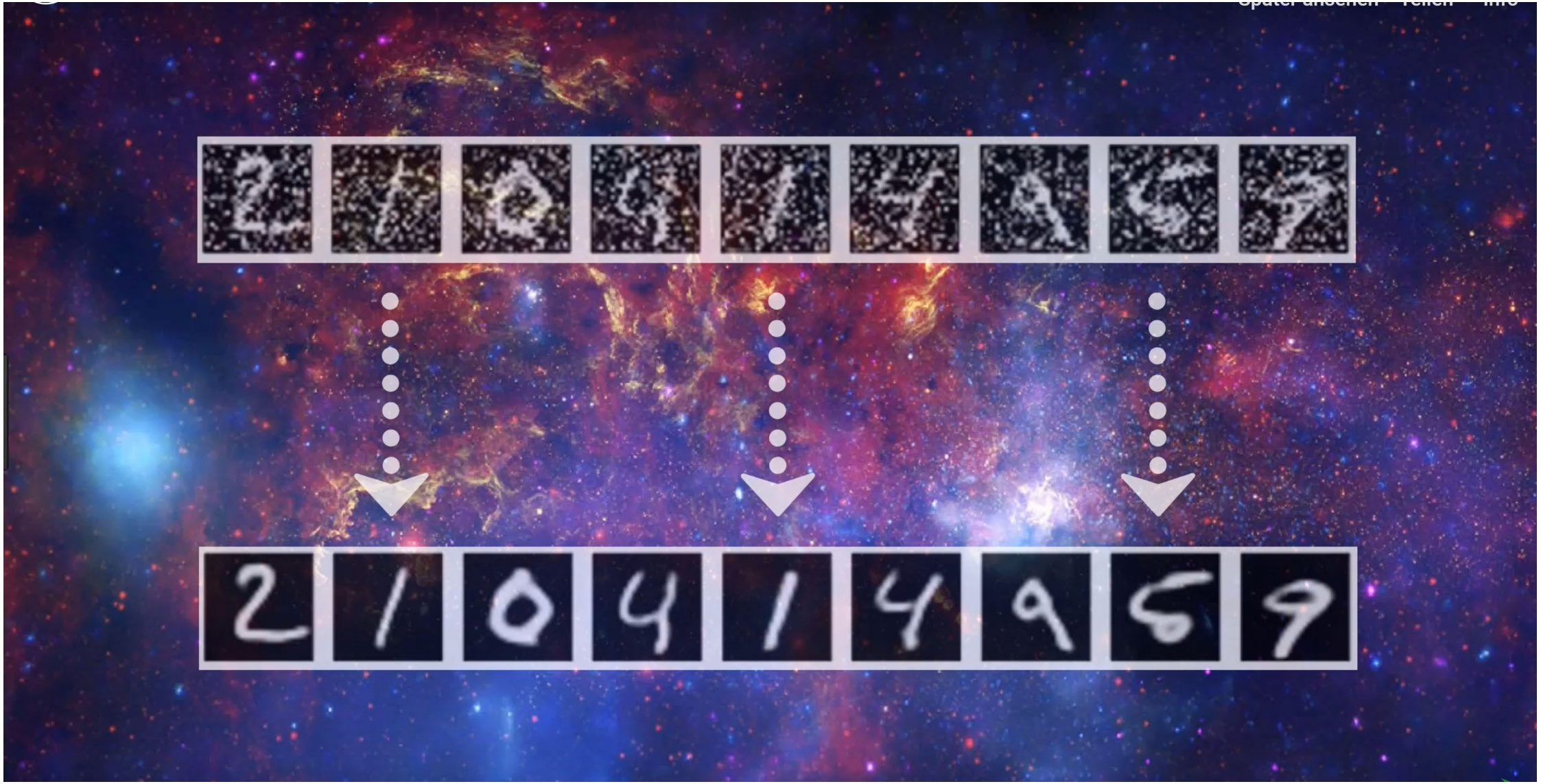




Unsupervised Learning For Machine Learning
Autoencoder



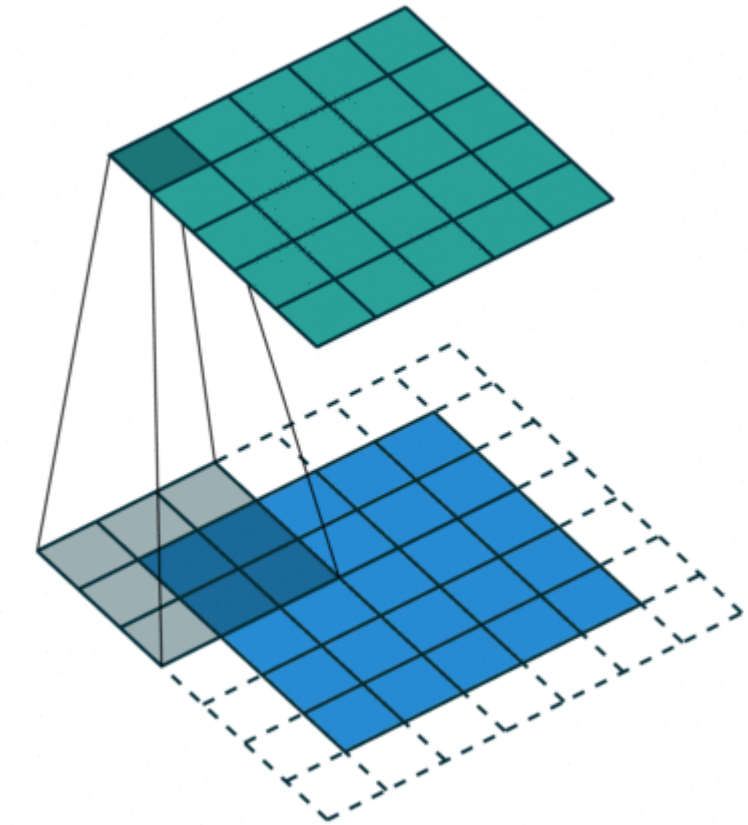
Unsupervised Learning For Machine Learning
Autoencoder



Deep Learning With Convolutional Neural Networks

Convolutional Layers: Filters that work on convolution to detect patterns

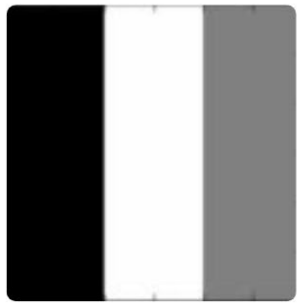
- edges
- shapes
- textures
- curves
- objects
- colors



Deep Learning With Convolutional Neural Networks

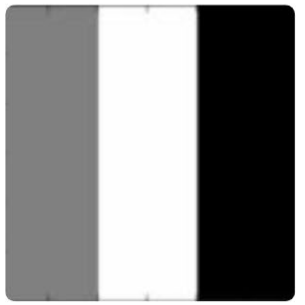
-1	-1	-1
1	1	1
0	0	0

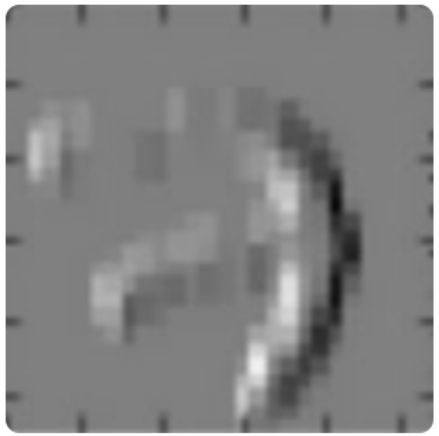
-1	1	0
-1	1	0
-1	1	0

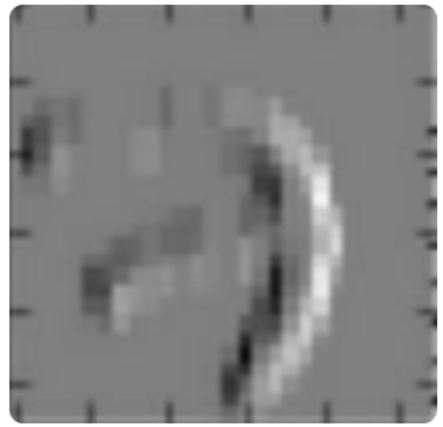


0	0	0
1	1	1
-1	-1	-1

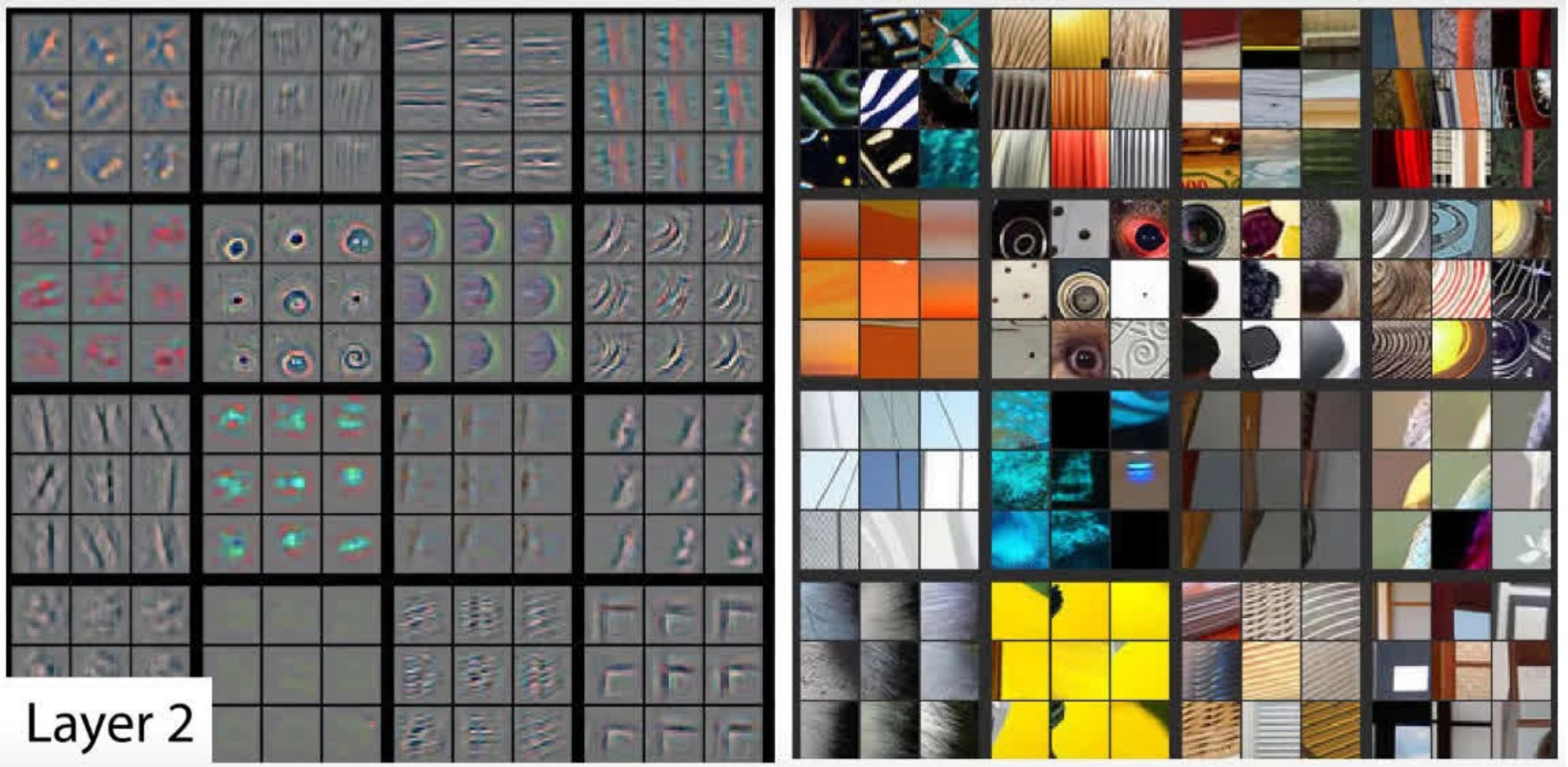
0	1	-1
0	1	-1
0	1	-1



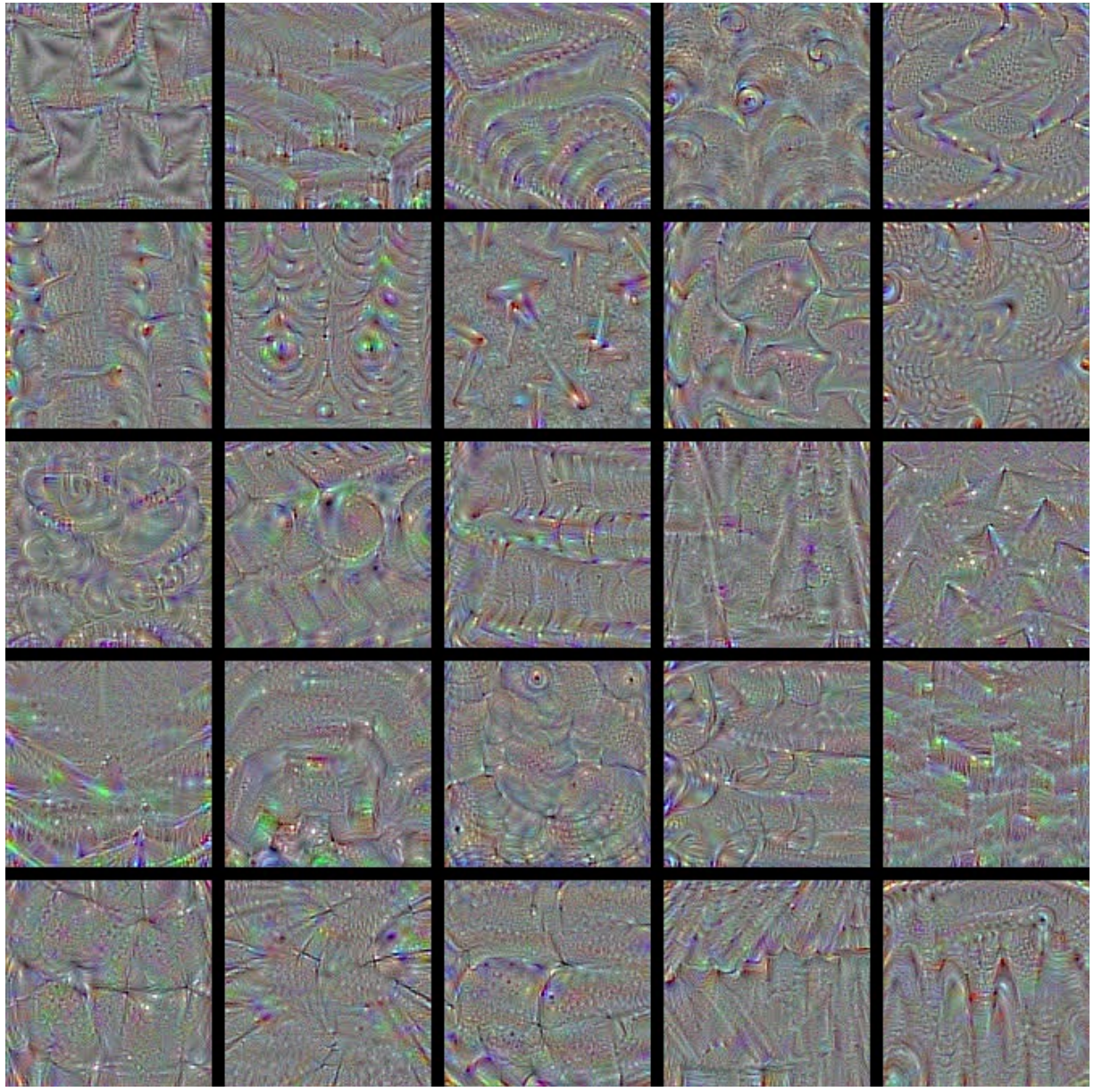


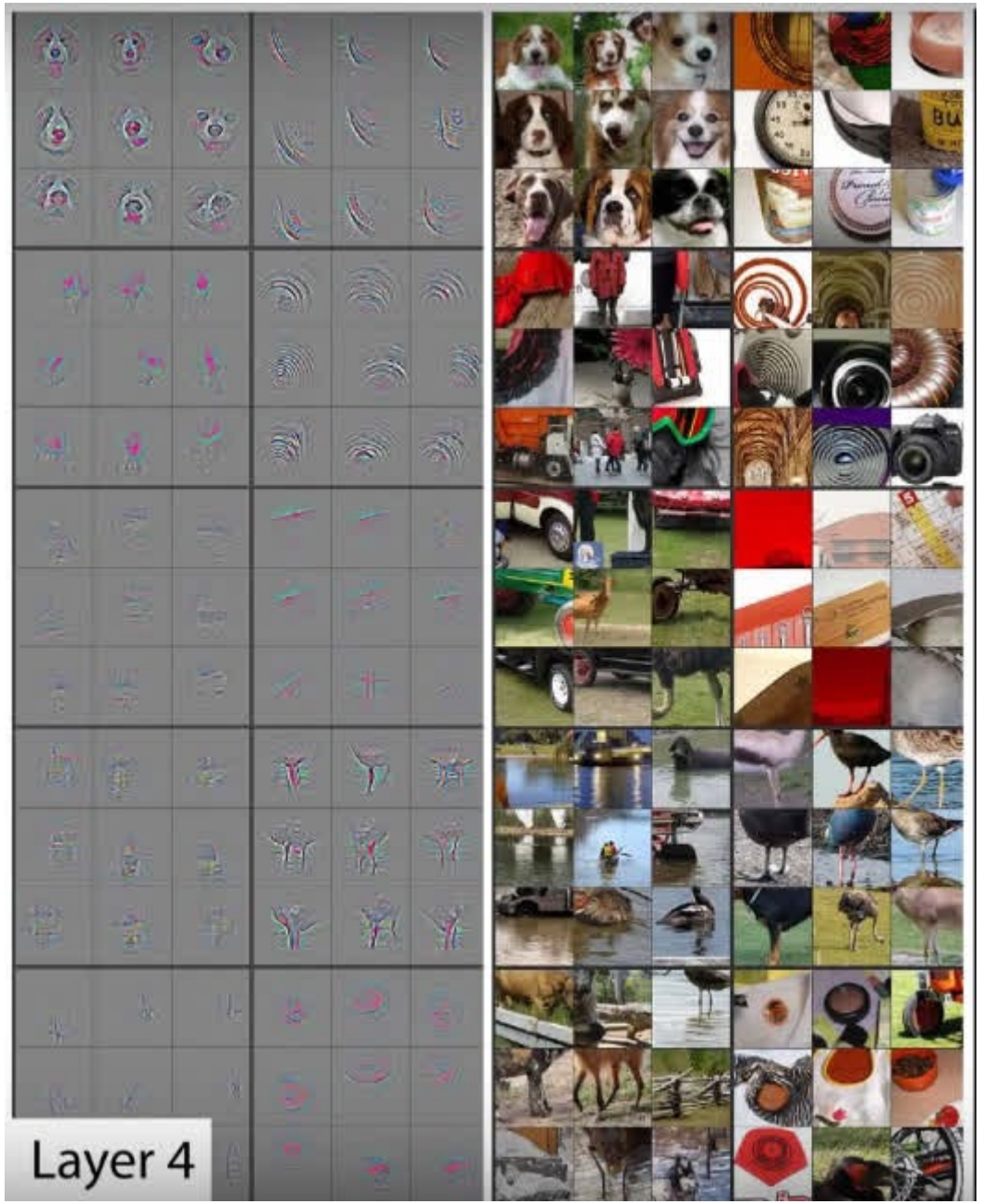


Deep Learning With Convolutional Neural Networks



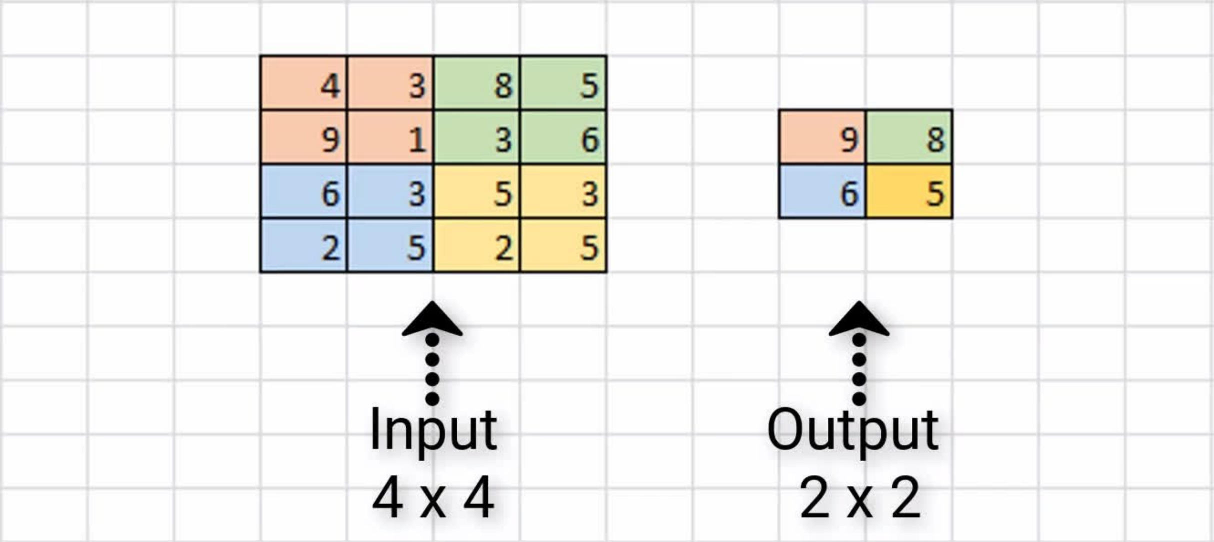
Visualizing Convolutional Neural Networks





Max-Pooling in CNNs

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.4	0.6	0.7	0.5	0.4	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.3	0.6	1.2	1.4	1.6	1.6	1.6	1.6	1.9	1.9	2.2	2.3	2.1	2.0	1.7	0.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.5	1.2	1.8	2.6	2.7	3.0	3.0	3.0	3.0	3.4	3.5	3.8	4.0	3.7	3.6	3.2	2.3	1.5	0.5	0.1	0.0	0.0	0.0	0.0	0.0	0.0
1.1	2.1	3.2	4.2	4.4	4.7	4.7	4.5	4.2	4.0	3.8	3.9	3.9	4.1	4.5	4.7	4.1	3.1	1.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0
1.1	2.0	3.1	3.6	3.3	3.2	3.2	3.1	2.9	2.7	2.5	2.5	2.5	2.7	3.0	3.9	4.4	4.1	2.9	1.4	0.3	0.0	0.0	0.0	0.0	0.0
0.9	1.4	2.1	2.2	1.8	1.7	1.7	1.5	1.1	0.8	0.5	0.5	0.5	0.8	1.3	2.4	3.7	4.5	4.0	2.4	1.0	0.0	0.0	0.0	0.0	0.0
0.1	0.3	0.3	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	1.3	2.8	4.2	4.7	2.8	1.6	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.2	2.9	3.9	5.1	3.1	2.2	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.4	1.0	1.3	1.6	1.9	2.4	3.7	4.4	5.2	3.8	2.5	0.7	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.5	1.1	1.7	2.3	2.7	3.0	3.4	3.7	4.6	4.9	5.2	4.1	2.5	1.2	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.1	0.7	1.3	1.9	2.6	3.2	4.0	4.4	4.8	4.4	4.2	4.5	4.8	5.2	4.5	2.7	1.6	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.4	1.0	1.8	2.6	3.3	3.8	3.9	3.8	3.6	3.4	3.0	2.9	3.6	4.1	5.0	3.8	2.5	1.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.8	1.7	3.0	3.5	3.7	3.3	3.0	2.5	2.2	1.9	1.3	1.3	2.4	3.3	4.8	3.4	2.3	0.6	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	2.0	2.7	3.2	2.6	1.8	1.3	0.7	0.4	0.1	0.0	0.4	2.2	3.3	4.6	3.0	2.0	0.2	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.7	1.4	1.6	1.7	0.7	0.2	0.0	0.0	0.0	0.0	0.0	0.8	2.5	3.7	4.2	2.6	1.5	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.1	0.5	0.2	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.7	1.7	3.3	4.0	3.6	2.2	0.8	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.3	2.3	4.0	3.9	2.8	1.6	0.2	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	2.3	3.1	4.5	3.4	2.0	0.8	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	2.6	3.4	3.8	2.5	1.2	0.2	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.2	2.0	2.8	2.4	1.5	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.3	2.0	1.3	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.3	0.6	0.7	0.4	0.0	0.0	0.0	0.0	0.0
1.2	2.6	3.0	3.0	3.4	3.8	4.0	3.6	2.3	0.5	0.0	0.0	0.0
2.1	4.2	4.7	4.7	4.2	3.9	4.1	4.7	4.4	2.9	0.3	0.0	0.0
1.4	2.2	1.8	1.7	1.1	0.5	0.8	2.4	4.5	4.7	1.6	0.0	0.0
0.0	0.0	0.0	0.0	0.1	1.0	1.6	2.4	4.4	5.2	2.5	0.0	0.0
0.0	0.0	0.1	1.3	2.6	4.0	4.8	4.4	4.9	5.2	2.7	0.0	0.0
0.0	0.0	1.7	3.5	3.8	3.9	3.6	3.0	4.1	5.0	2.5	0.0	0.0
0.0	0.0	2.0	3.2	2.6	1.3	0.4	0.8	3.7	4.6	2.0	0.0	0.0
0.0	0.0	0.5	0.5	0.0	0.0	0.0	2.3	4.0	3.6	0.8	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.9	3.4	4.5	2.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.2	2.8	2.4	0.3	0.0	0.0	0.0

Setting up CNNs

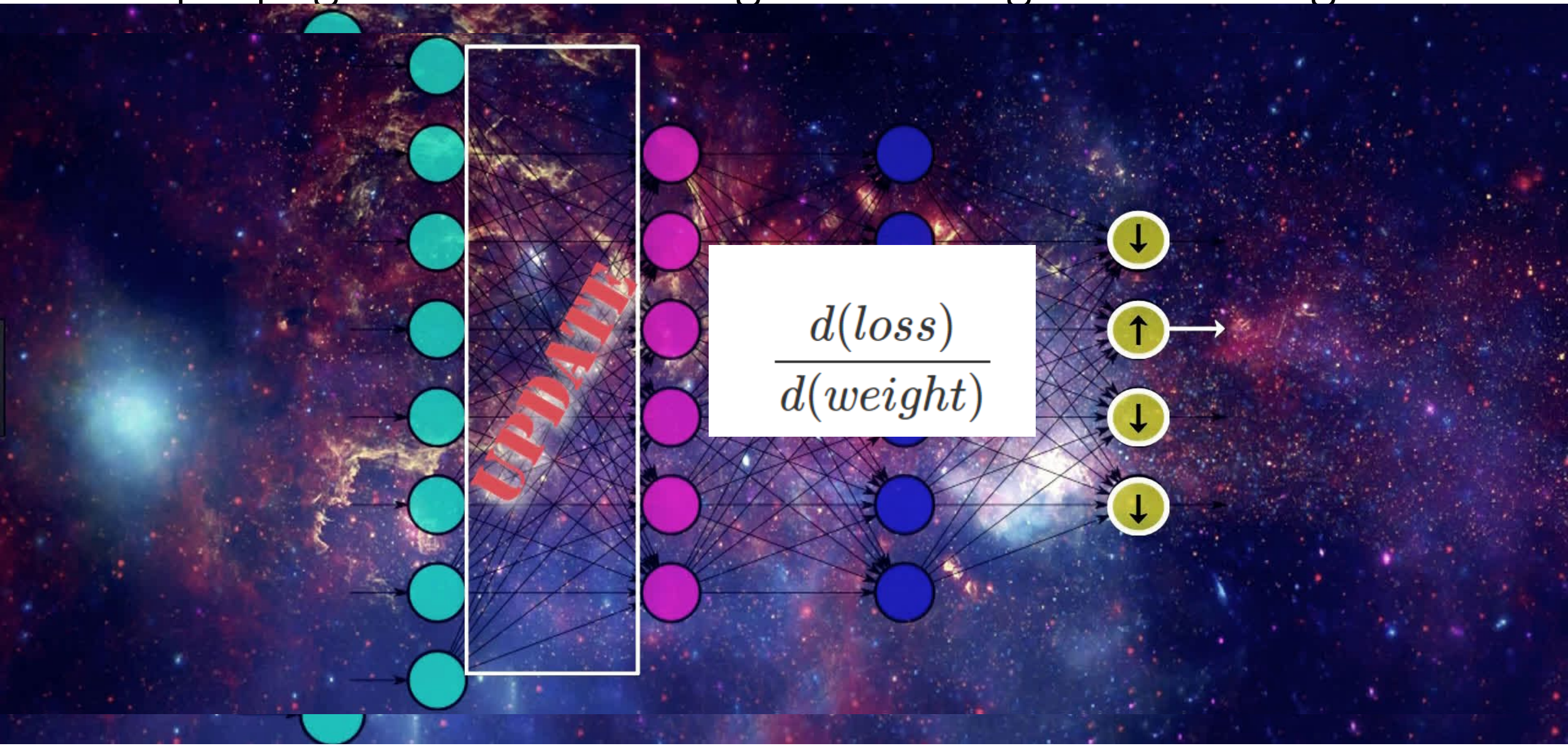
In [13]:
```
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense, Flatten
from keras.layers.convolutional import *
from keras.layers.pooling import *
```

In [14]:
```
model_valid = Sequential([
    Dense(16, input_shape=(20,20,3), activation='relu'),
    Conv2D(32, kernel_size=(3,3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'),
    Conv2D(64, kernel_size=(5,5), activation='relu', padding='same'),
    Flatten(),
    Dense(2, activation='softmax')
])
```

In [15]:
```
model_valid.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_7 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d (Conv2D)              (None, 20, 20, 32)        4640
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 64)        51264
_________________________________________________________________
flatten (Flatten)            (None, 6400)              0
_________________________________________________________________
dense_8 (Dense)              (None, 2)                 12802
=================================================================
Total params: 68,770
Trainable params: 68,770
Non-trainable params: 0
_________________________________________________________________
```

Backpropagation in the fitting…. The magic of learning:



And some math!

Definitions and Notation

We define

L = number of layers in the network
Layers are indexed as $l = 1, 2, \cdots, L-1, L$
Nodes in a given layer l are indexed as $j = 0, 1, \cdots, n-1$
Nodes in layer $l-1$ are indexed as $k = 0, 1, \cdots, n-1$

y_j = the desired value of node j in the output layer L for a single training sample

C_0 = loss function of the network for a single training sample (sum of squared errors)

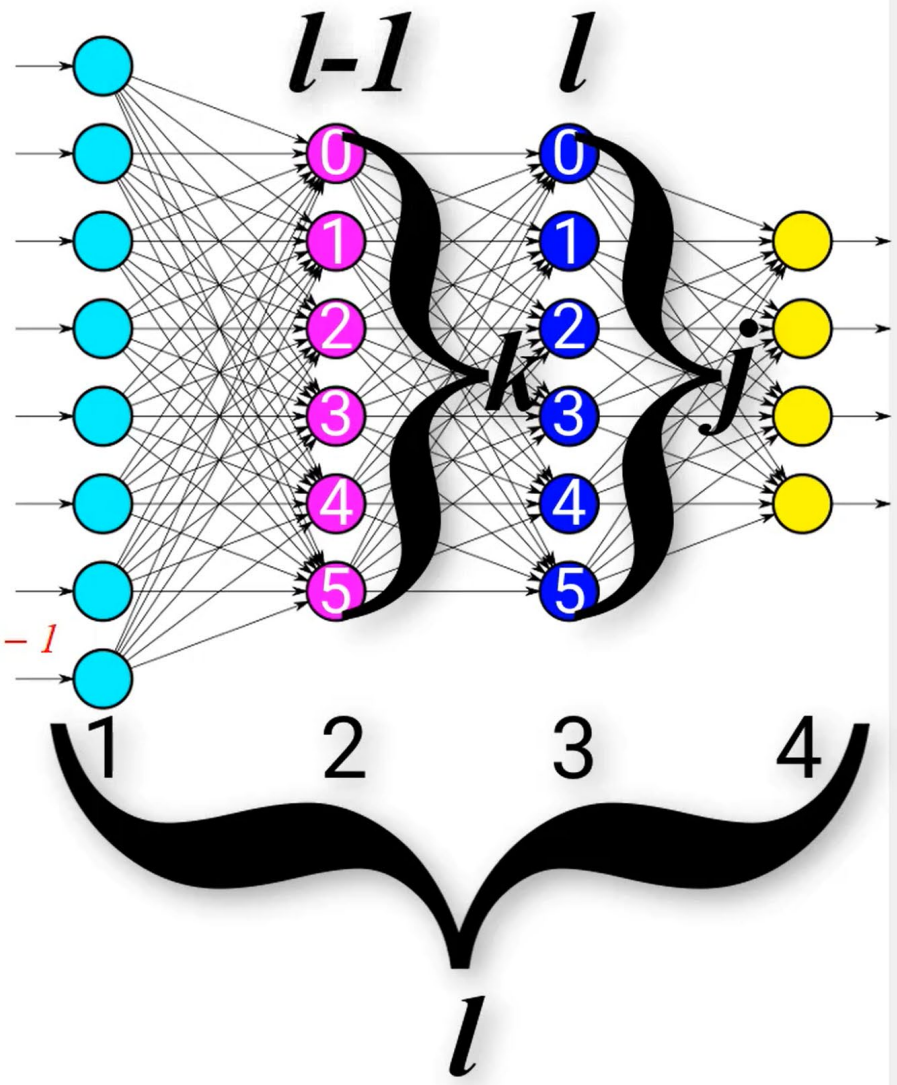
$w_{jk}^{(l)}$ = the weight of the connection that connects node k in layer $l-1$ to node j in layer l

$w_j^{(l)}$ = the vector that contains all weights connected to node j in layer l by each node in layer $l-1$

$z_j^{(l)}$ = the input for node j in layer l

$g^{(l)}$ = the activation function used for layer l

$a_j^{(l)}$ = the activation output of node j in layer l



And some math!

Input $z_j^{(l)}$

We know that the input for node j in layer l is the weighted sum of the activation outputs from the previous layer $l-1$.

An individual term from the sum looks like this:

$$w_{jk}^{(l)} a_k^{(l-1)}$$

So, the input for a given node j in layer l is expressed as

$$z_j^{(l)} = \sum_{k=0}^{n-1} w_{jk}^{(l)} a_k^{(l-1)}.$$

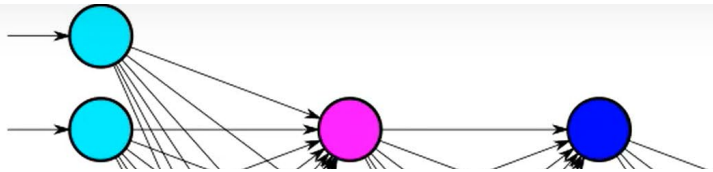
$$C_0 = \sum_{j=0}^{n-1} \left(a_j^{(L)} - y_j \right)^2.$$

$$C_{0_j} = C_{0_j}\left(a_j^{(L)}\left(z_j^{(L)}\left(w_j^{(L)} \right) \right) \right).$$

$$C_0 = \sum_{j=0}^{n-1} C_{0_j},$$

And some math!

Calculations



$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right)$$

$$= 2\left(a_1^{(L)} - y_1\right)\left(g'^{(L)}\left(z_1^{(L)}\right)\right)\left(a_2^{(L-1)}\right)$$

This is expressed as

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right).$$

Let's break down each term from the expression on the right hand side of the above equation.

And some math!

Derivative of the loss with respect to activation outputs

Motivation

We left off seeing how we can calculate the gradient of the loss function with respect to any weight in the network.

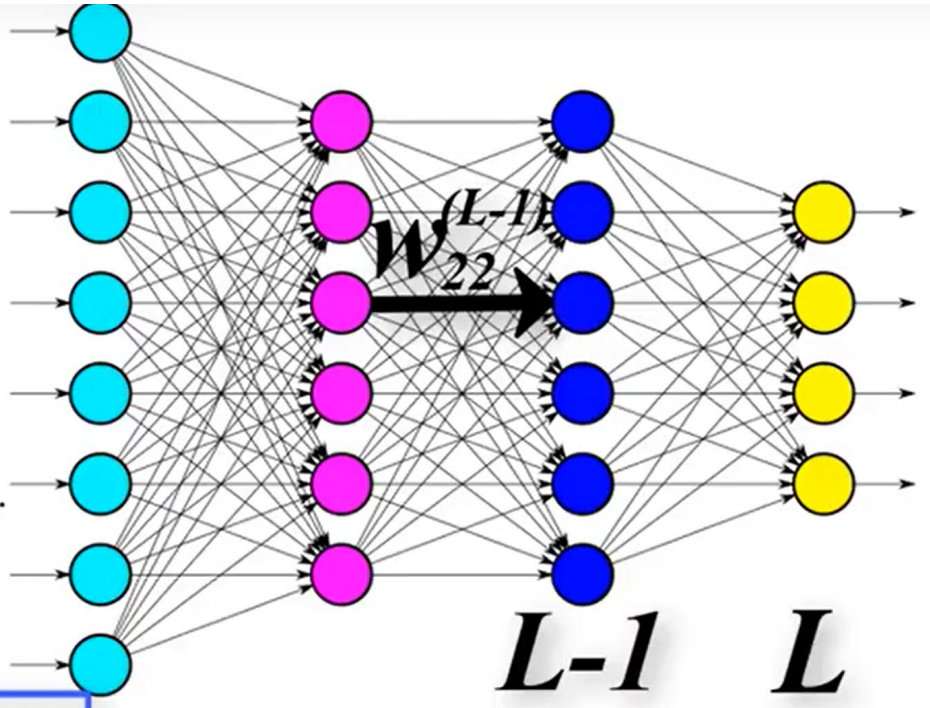
Recall, the weight we chose to work with to explain this idea was $w_{12}^{(L)}$, and we saw that

$$\frac{\partial C_0}{\partial w_{12}^{(L)}} = \left(\frac{\partial C_0}{\partial a_1^{(L)}}\right)\left(\frac{\partial a_1^{(L)}}{\partial z_1^{(L)}}\right)\left(\frac{\partial z_1^{(L)}}{\partial w_{12}^{(L)}}\right).$$

Suppose we choose to work with a weight that is not in the output layer, like $w_{22}^{(L-1)}$.

Then the gradient of the loss with respect to this weight would be

$$\frac{\partial C_0}{\partial w_{22}^{(L-1)}} = \left(\frac{\partial C_0}{\partial a_2^{(L-1)}}\right)\left(\frac{\partial a_2^{(L-1)}}{\partial z_2^{(L-1)}}\right)\left(\frac{\partial z_2^{(L-1)}}{\partial w_{22}^{(L-1)}}\right).$$



The second and third terms on the right hand hand side, $\frac{\partial C_0}{\partial a_2^{(L-1)}}$, will not be calculated in t

We need to understand how to calculate this *not* in the output layer.

The calculation of this term will be our focu

$$\frac{\partial C_0}{\partial a_2^{(L-1)}} = \sum_{j=0}^{n-1}\left(\left(\frac{\partial C_0}{\partial a_j^{(L)}}\right)\left(\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}\right)\left(\frac{\partial z_j^{(L)}}{\partial a_2^{(L-1)}}\right)\right).$$