Deep Learning Workshop: Applied Computer Vision

Jeremy Pinto, jeremy.pinto@mila.quebec

Pierre-Luc St-Charles, pierreluc.stcharles@mila.quebec

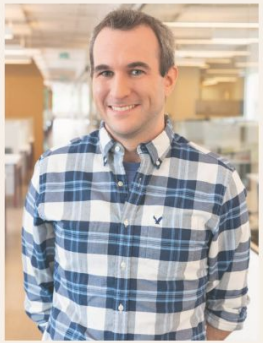
Course Overview

Introduction

This course is meant to be an **overview** of key deep learning concepts applied to computer vision.

We will be going over **high-level** theoretical concepts and intuitions and applying them to **hands-on** examples.

Jeremy Pinto
Senior Applied Research Scientist

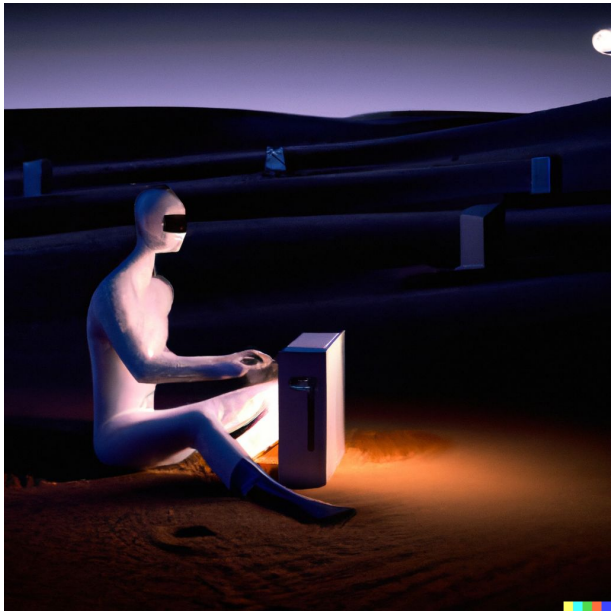


Pierre-Luc St-Charles
Senior Applied Research Scientist

Course Content

In this course, we will:

- Look at how computers process images using **deep learning**
- Review **state-of-the-art** models and architectures
- Implement **classification** tasks using **modern** tools and libraries



“A computer understanding that it can understand itself, digital art” Generated by DALL·E

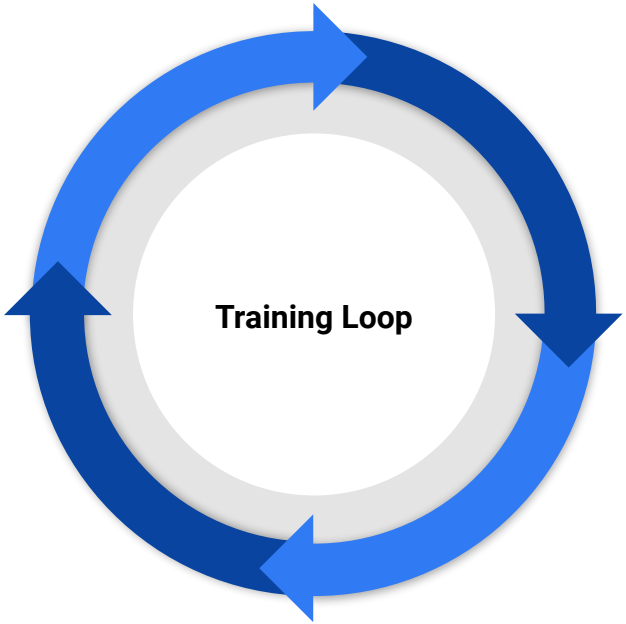
Course Structure

The course is spread over **4 sessions**, with a mix of theory and practical content.

Session 1: Theory	Session 2: Practical	Session 3: Theory	Session 4: Practical
Introduction to computer vision and deep learning concepts	**Implementation of a classification algorithm on a rock, paper, scissors dataset**	**Review of state-of-the-art models, vision transformers, object detection**	**Overview of an applied project using modern tools and libraries and best practices**

Session 1 Content

- Introduction to computer vision
- Image representations
- Image classification
- Neural networks
- Losses + optimizers
- Training + evaluating models



Computer Vision

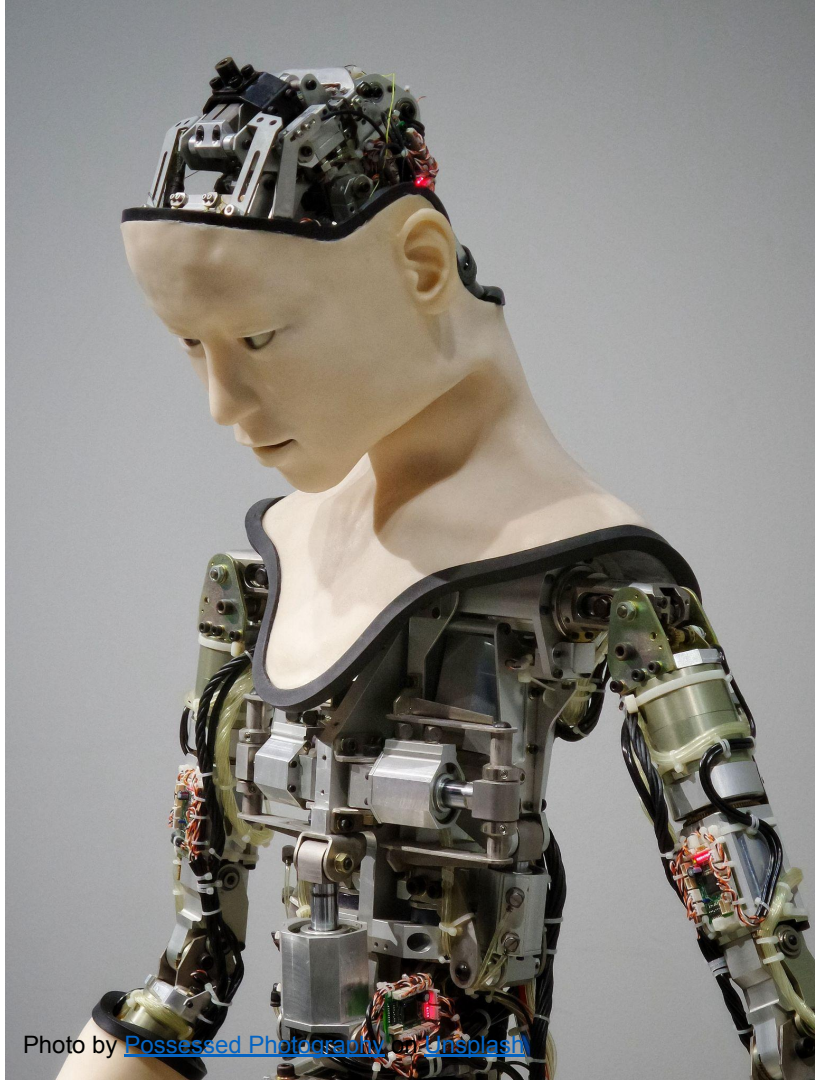
Human Vision

The human vision system allows us to **navigate the world** to perform complex tasks:

- Recognize objects
- Avoid danger
- Identify predators
- Manipulate tools
- Drive cars
- etc.

Computer Vision

The field of computer vision attempts to **enable** computers with **human-like** vision abilities.



Computer Vision

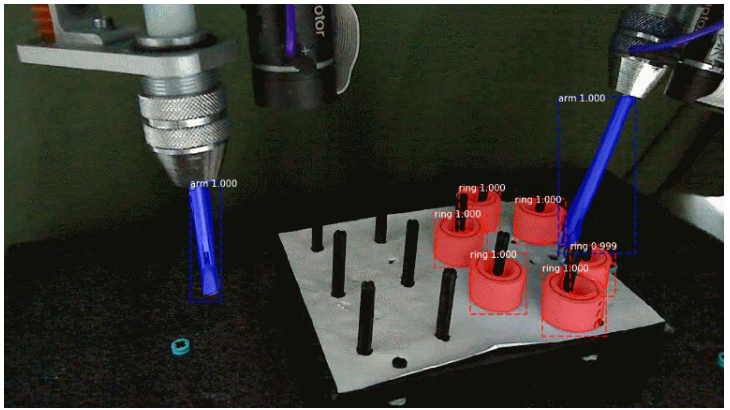
Ultimately, computer vision seeks to **imitate** (even surpass) human ability at everyday tasks:

- Driving Cars
- Recognizing objects and people
- Navigating the physical world
- Avoiding obstacles
- etc.

Computer Vision

Computer vision remains an **active area of research**. Topics include:

- Image classification
- Object detection
- Scene understanding
- Image generation
- etc.



Source

Image Representations

Image Representations

Engineers of the first computers opted for a simple **grid-like** format for images.

Squares on the grid could take **binary** values:

0 → **OFF**

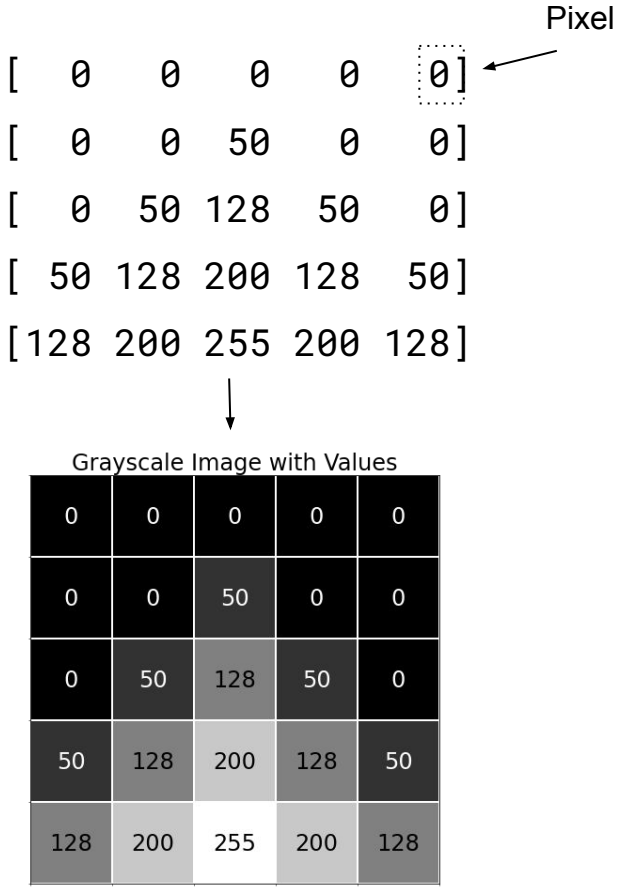
1 → **ON**

First digital image created by scanning an analogue photograph, 1957.

Image Representations

This format is still largely used today.
Individual squares of a grid are called **pixels**.

Each pixel can take any **integer value** in the range [0, 255].



colab

IVADO Mila

Image Representations

Can you figure out who this is?

```
[  0   0   0 100 100 100 100 100   0   0   0   0]
[  0   0 100 100 100 100 100 100 100 100 100   0]
[  0   0 128 128 128 255 255 128 255   0   0   0]
[  0 128 255 128 255 255 255 128 255 255 255   0]
[  0 128 255 128 128 255 255 255 128 255 255 128]
[  0 128 128 255 255 255 255 128 128 128 128   0]
[  0   0   0 255 255 255 255 255 255 255   0   0]
[  0   0 100 100 200 100 100 100 100   0   0   0]
[  0 100 100 100 200 100 100 200 100 100 100   0]
[100 100 100 100 200 200 200 200 100 100 100 100]
[255 255 100 200 255 200 200 255 200 100 255 255]
[255 255 255 200 200 200 200 200 200 255 255 255]
[255 255 200 200 200 200 200 200 200 200 255 255]
[  0   0 200 200 200   0   0 200 200 200   0   0]
[  0 128 128 128   0   0   0   0 128 128 128   0]
[128 128 128 128   0   0   0   0 128 128 128 128]
```

colab

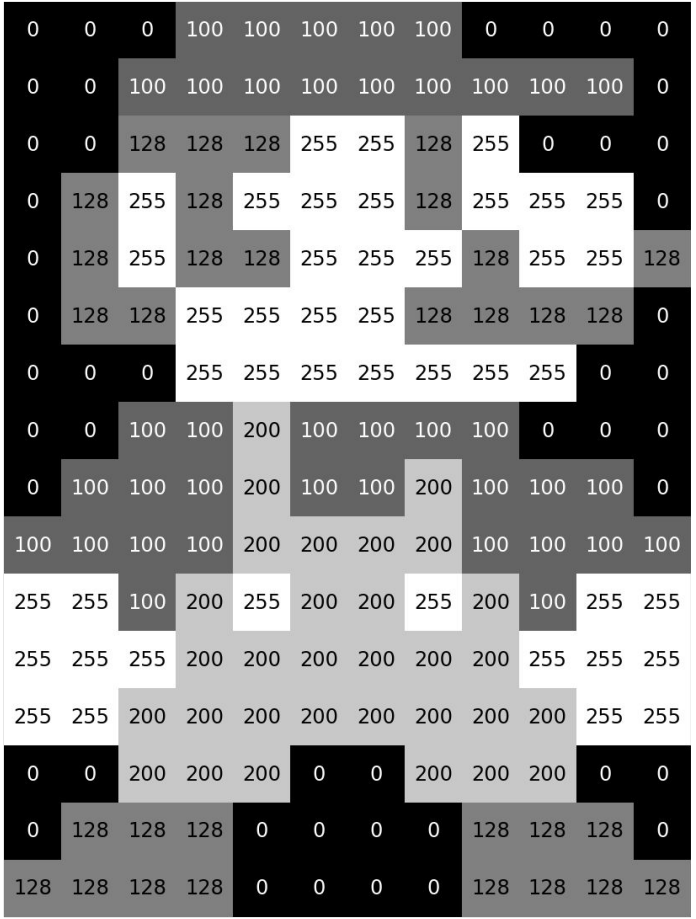
Image Representations

Can you figure out who this is?

colab

Image Representations

Can you figure out who this is?



colab

Image Representation

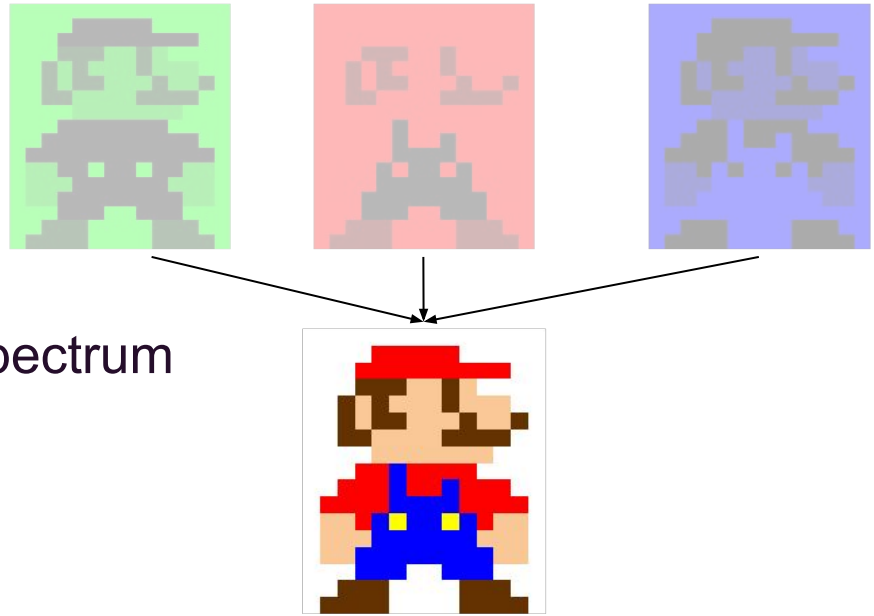
To display colours, we use can use an **RGB** representation.

Red, **G**reen, and **B**lue **colour channels** are used to define the different **intensities** and mixed to give all the colours our eyes can see.

A picture of Mohammed Alim Khan (1880-1944), Emir of Bukhara, taken in 1911

Image Representation

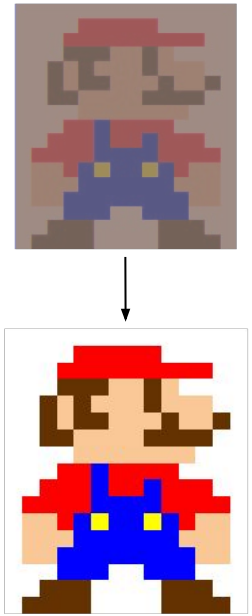
By **superimposing** the three colour channels, we can recreate an entire spectrum of colours in images.



colab

Image Representation

By **superimposing** the three colour channels, we can recreate an entire spectrum of colours in images.

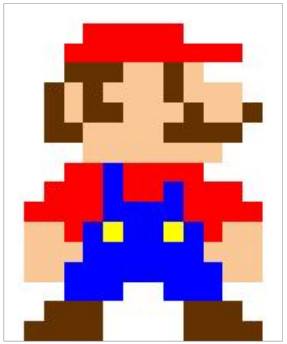


colab

Image Representation

By defining the RGB values as **triplets** in [0, 255], we can numerically represent images as a combination of

Red, **G**reen, and **B**lue.



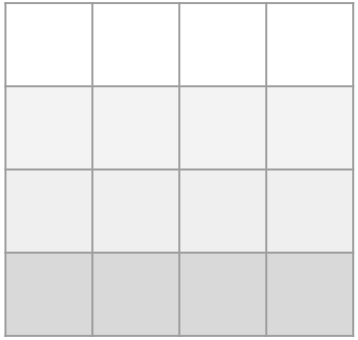
```
W = [255, 255, 255] # White
R = [255,   0,   0] # Red
B = [  0,   0, 255] # Blue
Y = [255, 255,   0] # Yellow
S = [250, 200, 150] # Beige
C = [100,  50,   0] # Brown
```

```
[[W, W, W, W, W, W, W, W, W, W, W, W, W, W],
 [W, W, W, W, R, R, R, R, R, W, W, W, W, W],
 [W, W, W, R, R, R, R, R, R, R, R, R, W, W],
 [W, W, W, C, C, C, S, S, C, S, W, W, W, W],
 [W, W, C, S, C, S, S, S, C, S, S, S, W, W],
 [W, W, C, S, C, C, S, S, S, C, S, S, C, W],
 [W, W, C, C, S, S, S, S, C, C, C, C, W, W],
 [W, W, W, W, S, S, S, S, S, S, S, W, W, W],
 [W, W, W, R, R, B, R, R, R, R, W, W, W, W],
 [W, W, R, R, R, B, R, R, B, R, R, R, W, W],
 [W, R, R, R, R, B, B, B, B, R, R, R, R, W],
 [W, S, S, R, B, Y, B, B, Y, B, R, S, S, W],
 [W, S, S, S, B, B, B, B, B, B, S, S, S, W],
 [W, S, S, B, B, B, B, B, B, B, B, S, S, W],
 [W, W, W, B, B, B, W, W, B, B, B, W, W, W],
 [W, W, C, C, C, W, W, W, W, C, C, C, W, W],
 [W, C, C, C, C, W, W, W, W, C, C, C, C, W],]
```

Image Representation

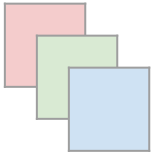
Grayscale

(1,)

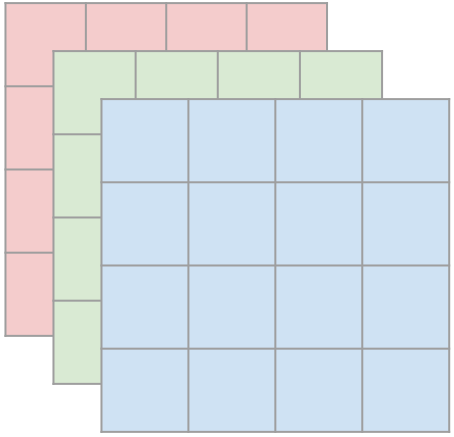


(4,4)

RGB



(3,)



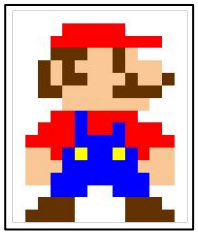
(4,4,3)

Resolution

The more pixels and colours we combine, the higher the **resolution** of the images we can have.

12 px

16 px



480 px

600 px

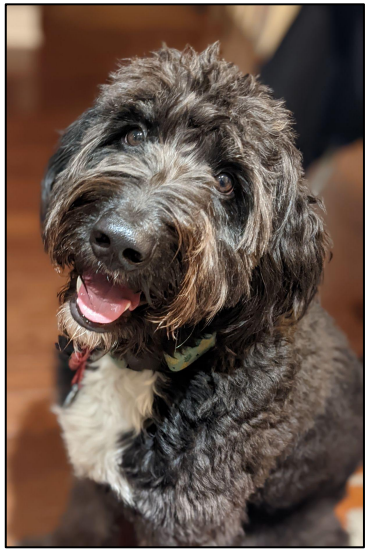


Image Representation

To display colour on **screens**, triplets of RGB channels display colours. Each channel can take a value between [0, 255].

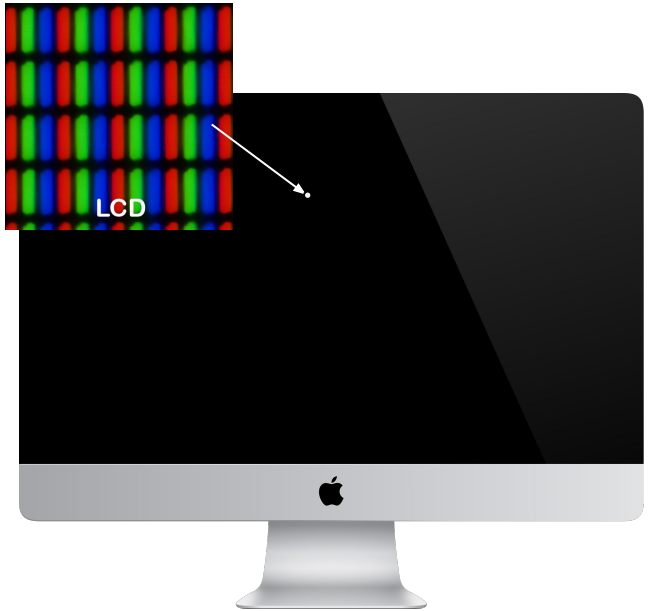


Image Representations

Using these digital representations of images, we can use all sorts of numerical operations to **manipulate** and **process** images.

Image Classification

Image Classification

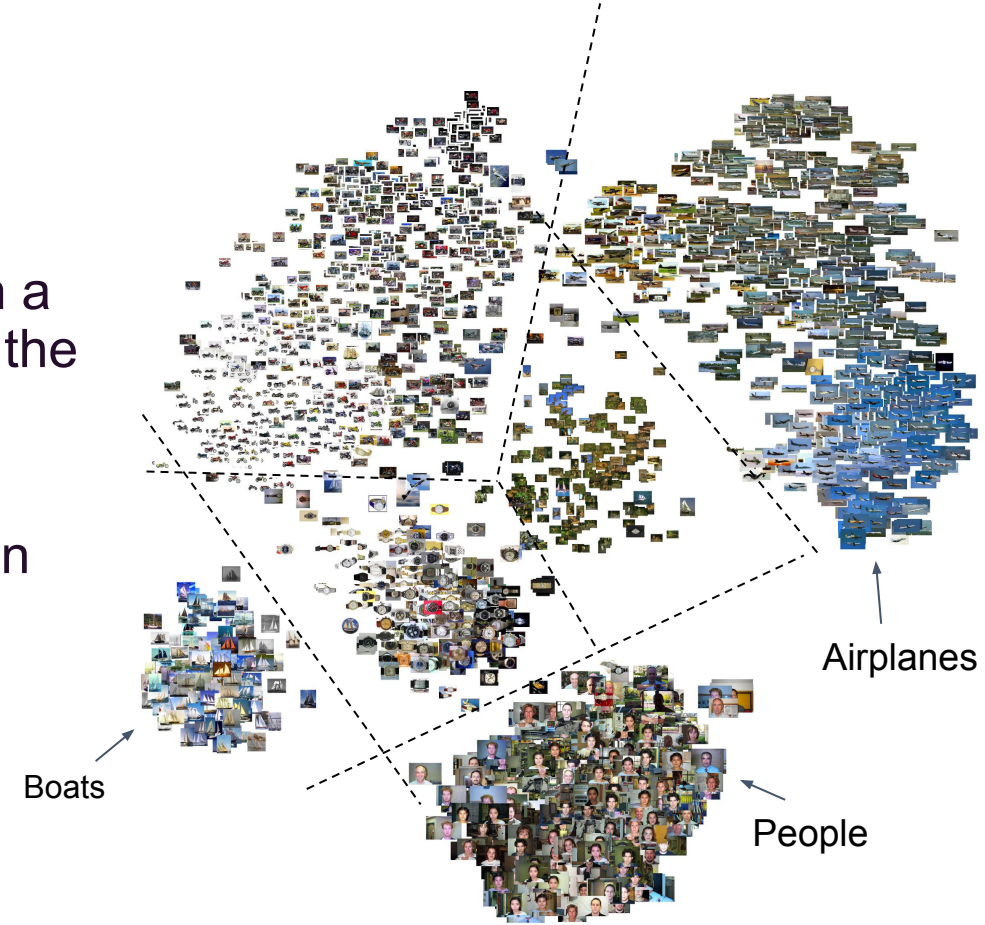
Image classification is a classic computer vision task:

➔ Given an **image**, we want to determine which **category** it belongs to.

Image Classification

The goal is to identify a mapping on a collection of data which **minimizes** the overall classification error.

In this course, we will be focusing on **deep learning** to implement these mappings.



Image

Image Classification

The key ingredients needed for implementing an image classification algorithm are:

1. A labelled dataset
2. A model
3. A loss function
4. An optimizer

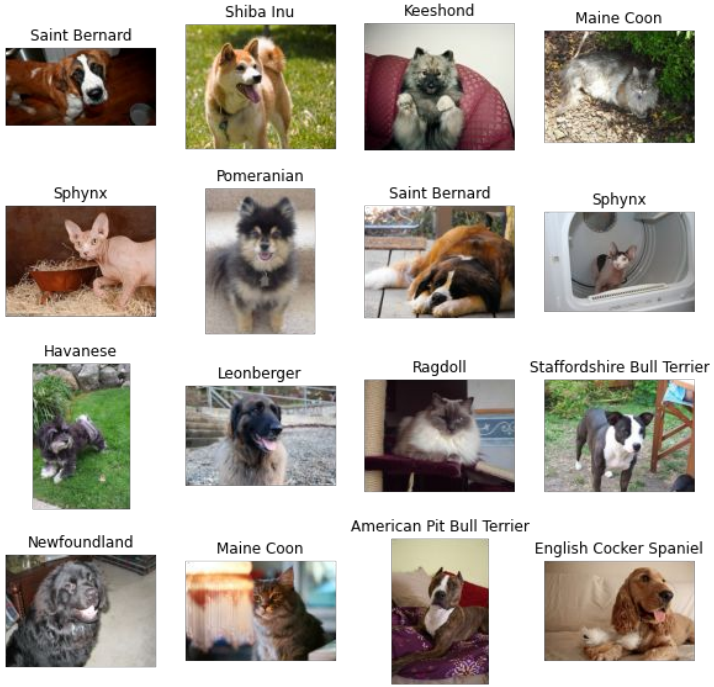
Image Classification

The key ingredients needed for implementing an image classification algorithm are:

1. **A labelled dataset**
2. A model
3. A loss function
4. An optimizer

Image Classification

Oxford III Pet dataset



A **labelled dataset** generally consists of

- Features (**X**)
  - Images (pixels)
- Labels (**y**)
  - Categories
  - Manually annotated and curated

colab

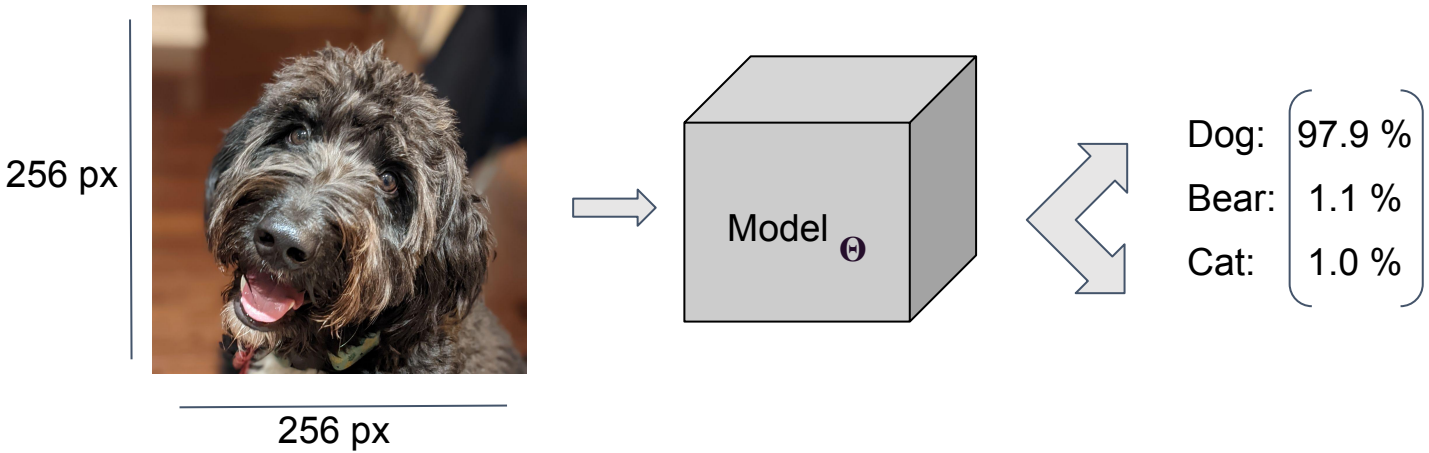
Image Classification

The key ingredients needed for implementing an image classification algorithm are:

1. A labelled dataset
2. **A model**
3. A loss function
4. An optimizer

Model

A classification model **maps** the input (images) to outputs (probabilities).
Neural networks consists of learnable parameters Θ (weights and biases).

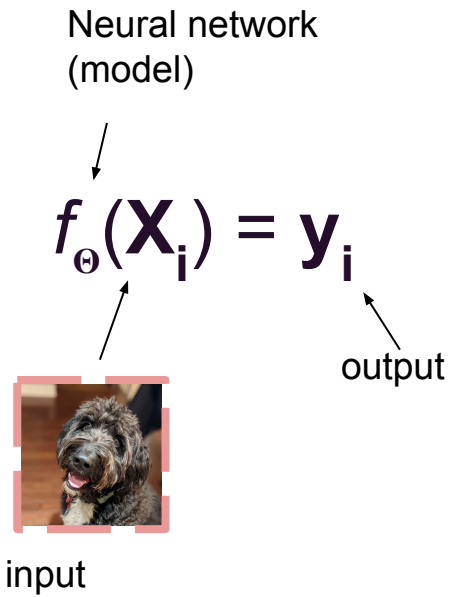


Neural Networks

Neural Networks

Neural networks are a family of models that can map an input $\mathbf{X}_i$ to an output $\mathbf{y}_i$ via learnable parameters $\boldsymbol{\Theta}$.

The goal is to automatically learn a function $\boldsymbol{f}_{\boldsymbol{\Theta}}$ that best fits observed data. This is known as **supervised learning**.

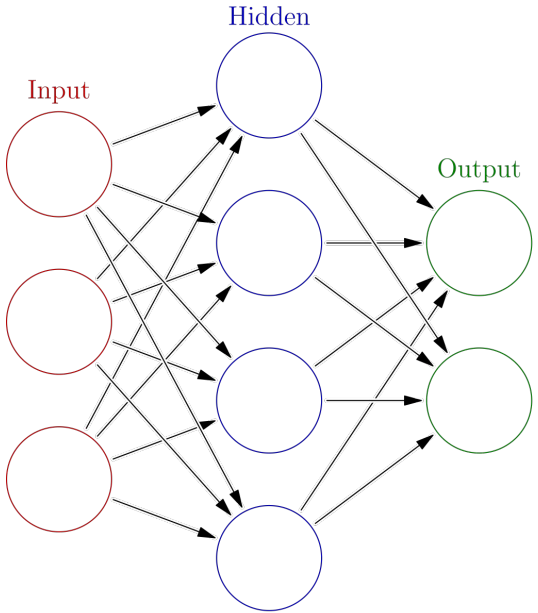


Neural Networks

The most common type of neural networks is the **multi-layer perceptron** (MLP).

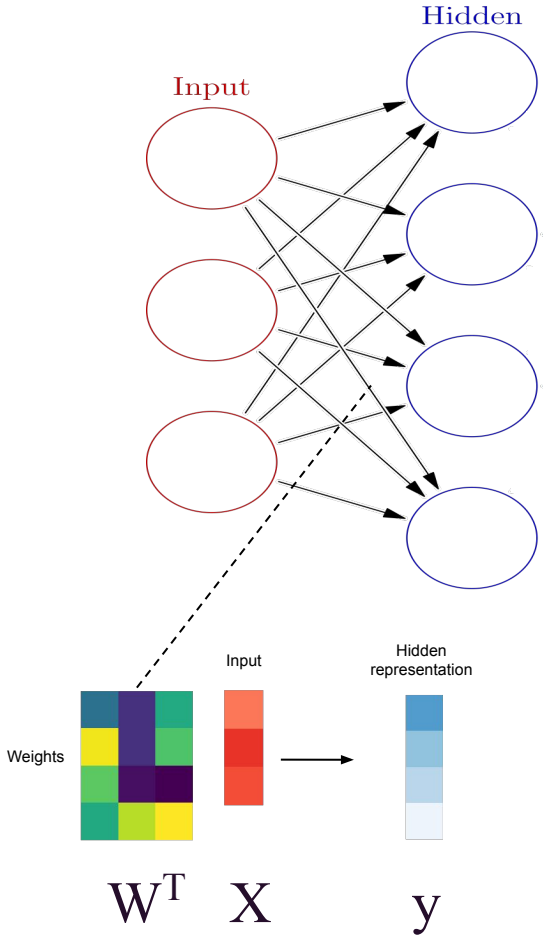
Every layer between the input and the output layers are called **hidden layers**.

MLPs transform a 1xN **input** to a 1xM **output** via matrix multiplication.



Neural Networks

The input to each layer is the output of the **previous** layer.

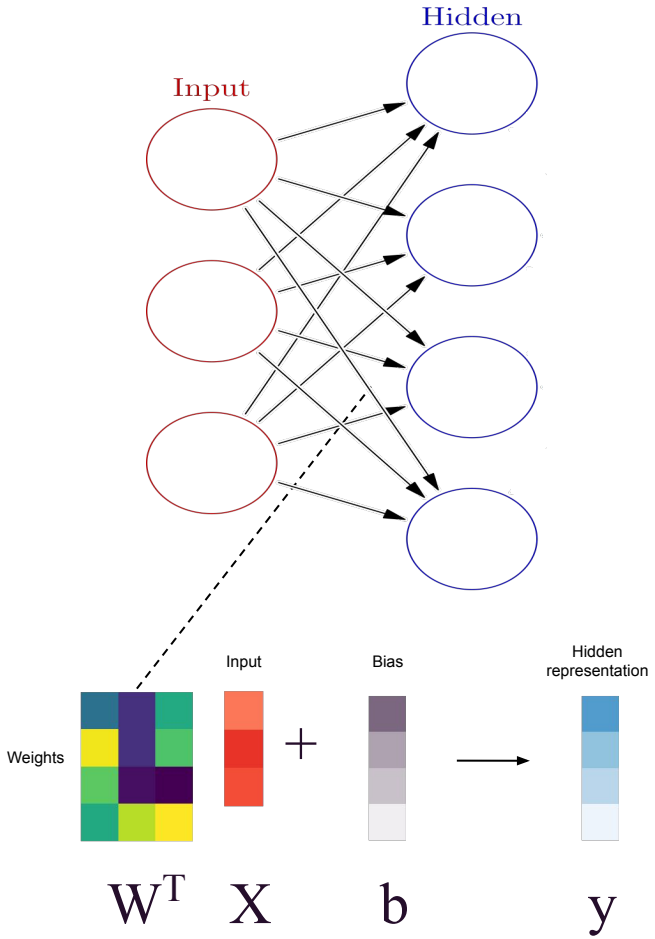


$$y = W^T x$$

Neural Networks

The input to each layer is the output of the **previous** layer.

A **bias** term is added after each matrix multiplication.

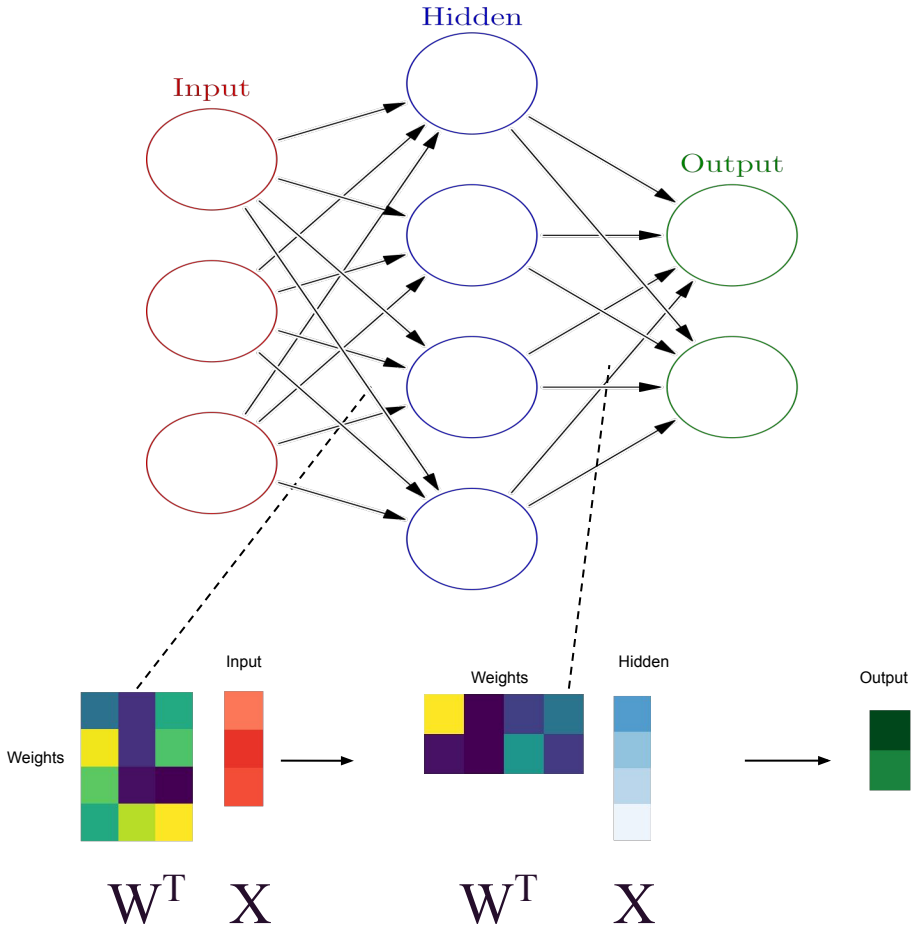


$$y = W^T x + b$$

Neural Networks

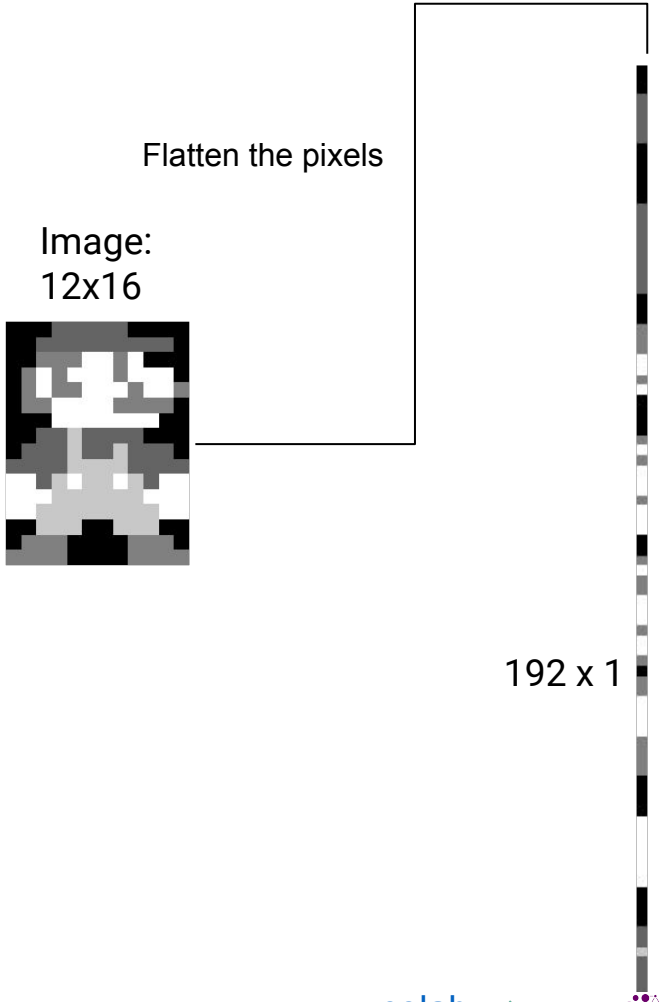
The input to each layer is the output of the **previous** layer.

The outputs are chained successively, hence the **multi-layer** nature.



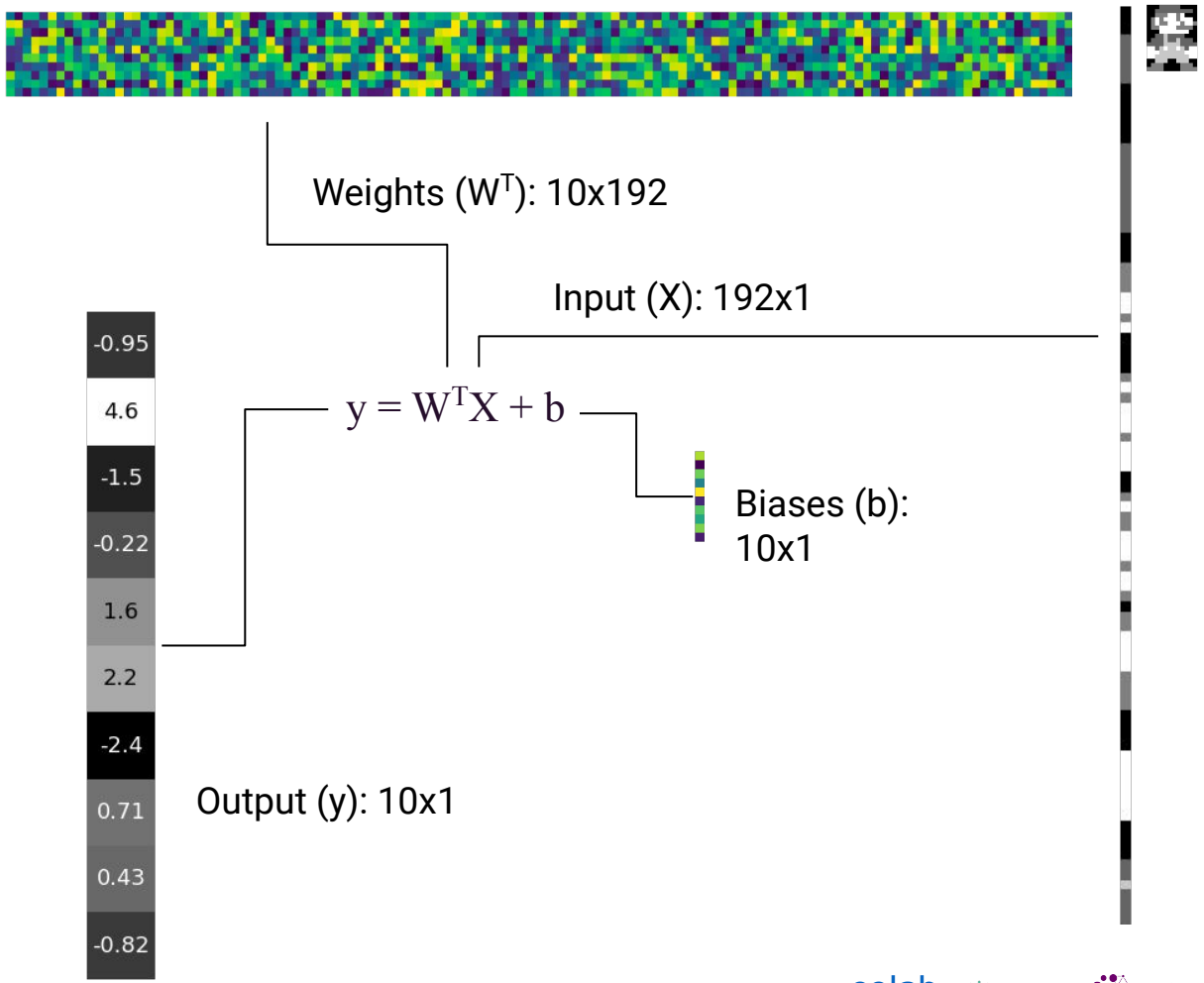
Neural Networks

To process an image in an MLP, we first **flatten** the image to turn it into a vector.



Neural Networks

Next, we multiply it by **weights** and add the **bias** to get our output.

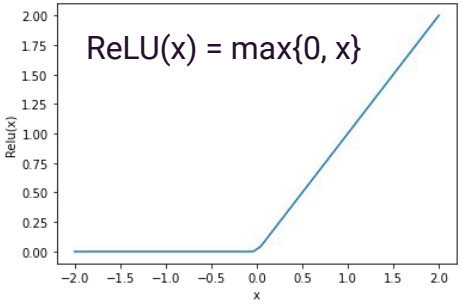


Neural Networks

The representational power of MLPs come from the **non-linear operation** that follows the matrix multiplication.

$$\mathbf{y} = \sigma(\mathbf{W}^{\mathrm{T}}\mathbf{x} + \mathbf{b})$$

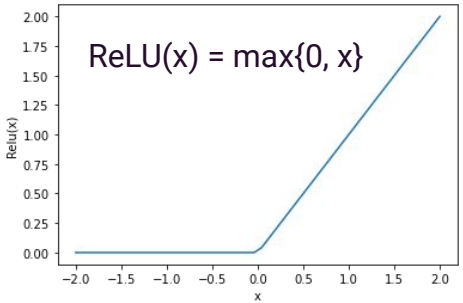
Element-wise **non-linear** activation function (e.g. ReLU)



[Why Neural Networks can learn (almost) anything]

Neural Networks

The representational power of MLPs come from the **non-linear operation** that follows the matrix multiplication.



$W^T x + b =$ -0.95 4.6 -1.5 -0.22 1.6 2.2 -2.4 0.71 0.43 -0.82

σ (-0.95 4.6 -1.5 -0.22 1.6 2.2 -2.4 0.71 0.43 -0.82)

Element-wise non-Linear Activation (e.g. ReLU)

$ReLU(W^T x + b)$

0 4.6 0 0 1.6 2.2 0 0.71 0.43 0

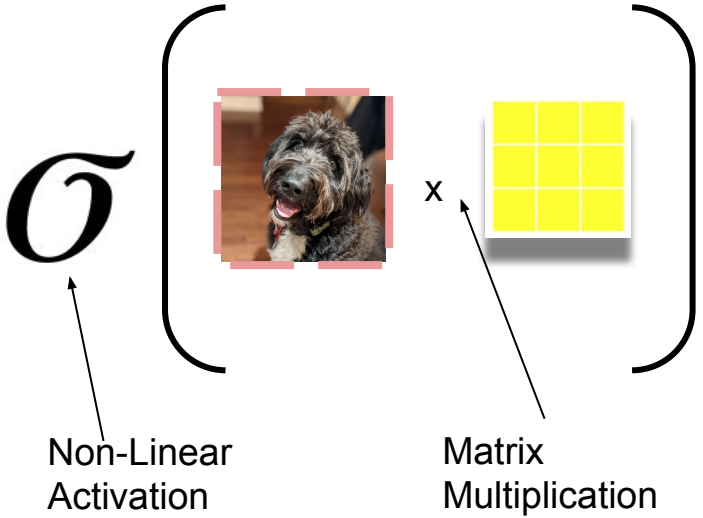
colab

Why Neural Networks can learn (almost) anything

IVADO Mila

Neural Networks

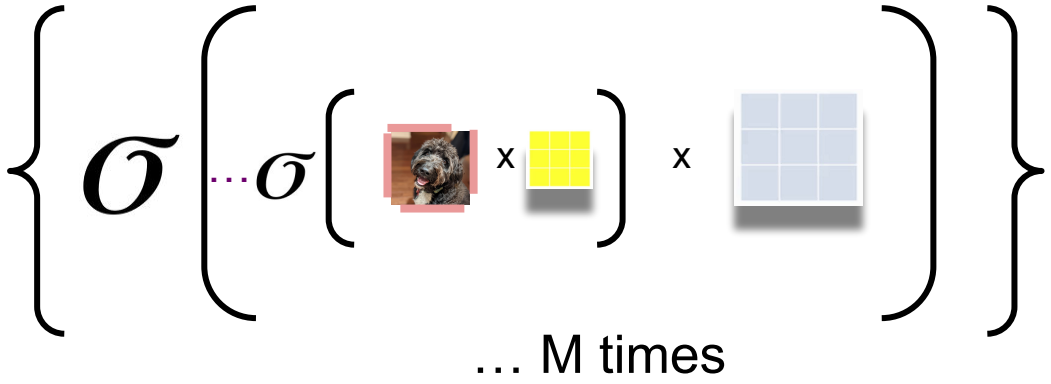
Each **layer** of an MLP consists of a matrix multiplication followed by a non-linear activation.

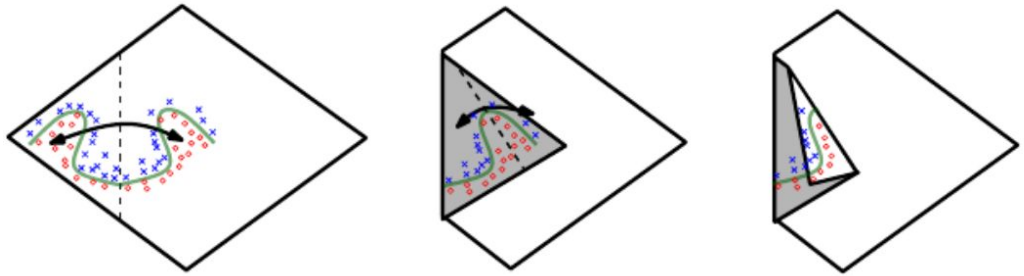


Neural Networks

By **chaining** these layers, we can effectively represent any function.

This is known as the [universal approximation theorem](#).





Logits

The output of an MLP, **logits**, will be a 1xM vector of real numbers in [-inf, +inf].

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

To convert logits to a probability distribution **p**, the **softmax** function can be used after the last linear layer.

$$softmax\left(\begin{array}{cccccccccc} 0 & 2 & 1.4 & 0 & 1.6 & 1.2 & 0 & 4.6 & 0 & 0.66 \end{array}\right)$$

$$\mathbf{p} = \begin{array}{cccccccccc} 0.008 & 0.06 & 0.033 & 0.008 & 0.038 & 0.028 & 0.008 & 0.79 & 0.008 & 0.015 \end{array}$$

$$\sum(\mathbf{p}) = 1$$

Predicted class

The **predicted class** is the index with the **maximum value** (argmax).

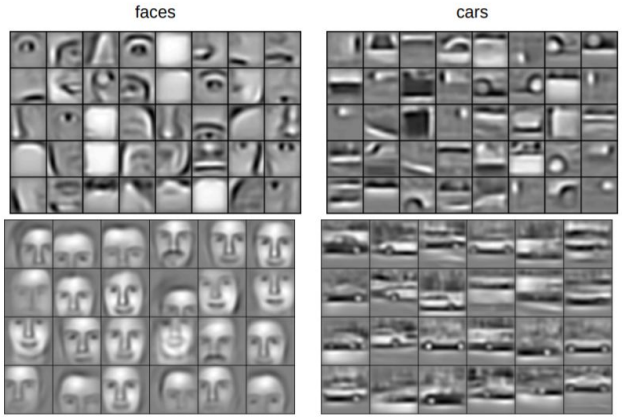
colab

Convolutional Neural Networks

Motivations for Convolutions

Convolutional Neural Networks, or **CNNs** were introduced to encourage neural networks to focus on **local features**, or “patches”, in images.

Think of a detector that scans images looking for “eye-like” and “nose-like” features to detect a human.

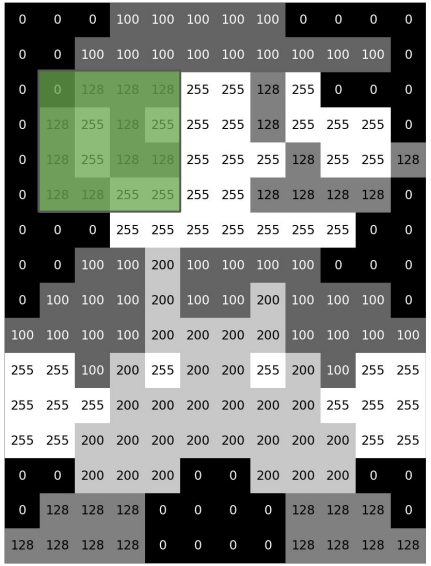


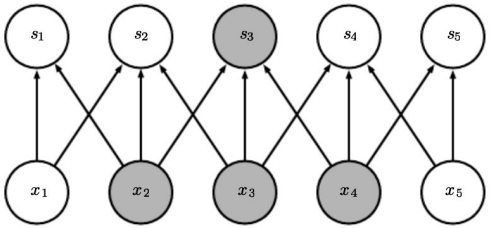
http://robotics.stanford.edu/~ang/papers/icml09-ConvolutionalDeepBeliefNetworks.pdf

Motivations for Convolutions

This is particularly useful in grid-like structures, like images:

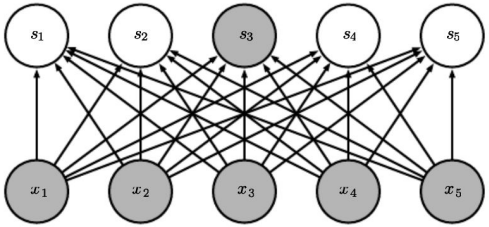
- **Local pixels** tend to be more important than distant pixels
- Features tend to be useful **throughout** images (parameter sharing)





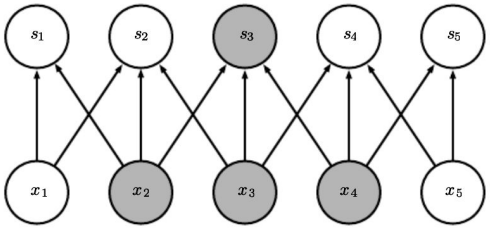
Motivations for Convolutions

By design, each input in an MLP **contributes equally** to every output node, which can be inefficient.



MLP

Convolutional Neural Networks, or **CNNs** were introduced to encourage neural networks to focus only on **local features**.

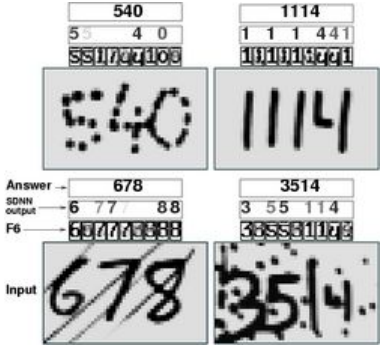


CNN

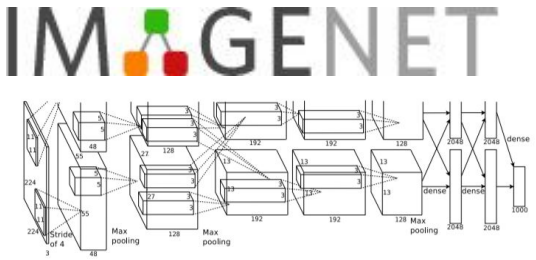
source

Motivations for Convolutions

CNNs were some of the **first deep learning models** to be shown to **generalize** well to "useful" tasks.



Modern interest in CNNs came about when Alex Krizhevsky et al. won the 2012 **ImageNet** object recognition challenge using **AlexNet**, a CNN with many layers.



Convolution

Convolutions are the **mathematical** operation behind CNNs.
Convolutions are expressed using the $*$ operator:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

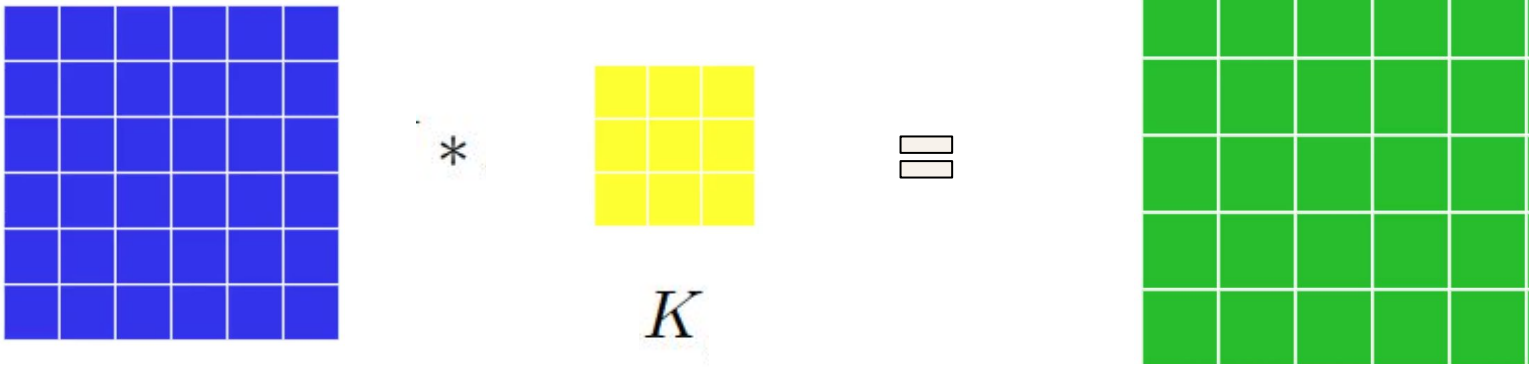
where $\mathbf{S}$ is said to be the convolution of $\mathbf{I}$ with $\mathbf{K}$. In the context of CNNs, $\mathbf{I}$ is typically referred to as the *input*, **K** as the *kernel* and **S** as the output or *feature map*.

2D Convolution

Let’s look at a **2D convolution** animation:

2D Convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n).$$



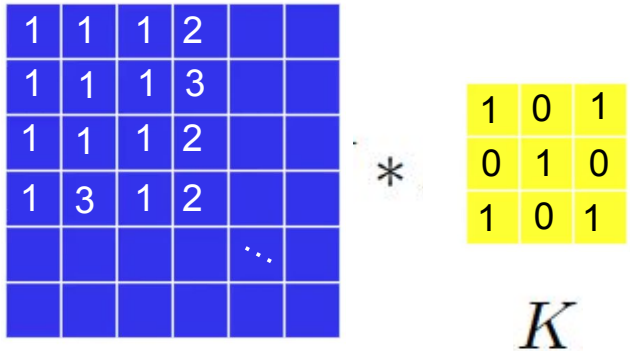
Input

Kernel (weights)

Output

2D Convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n).$$



$*$

1	0	1
0	1	0
1	0	1

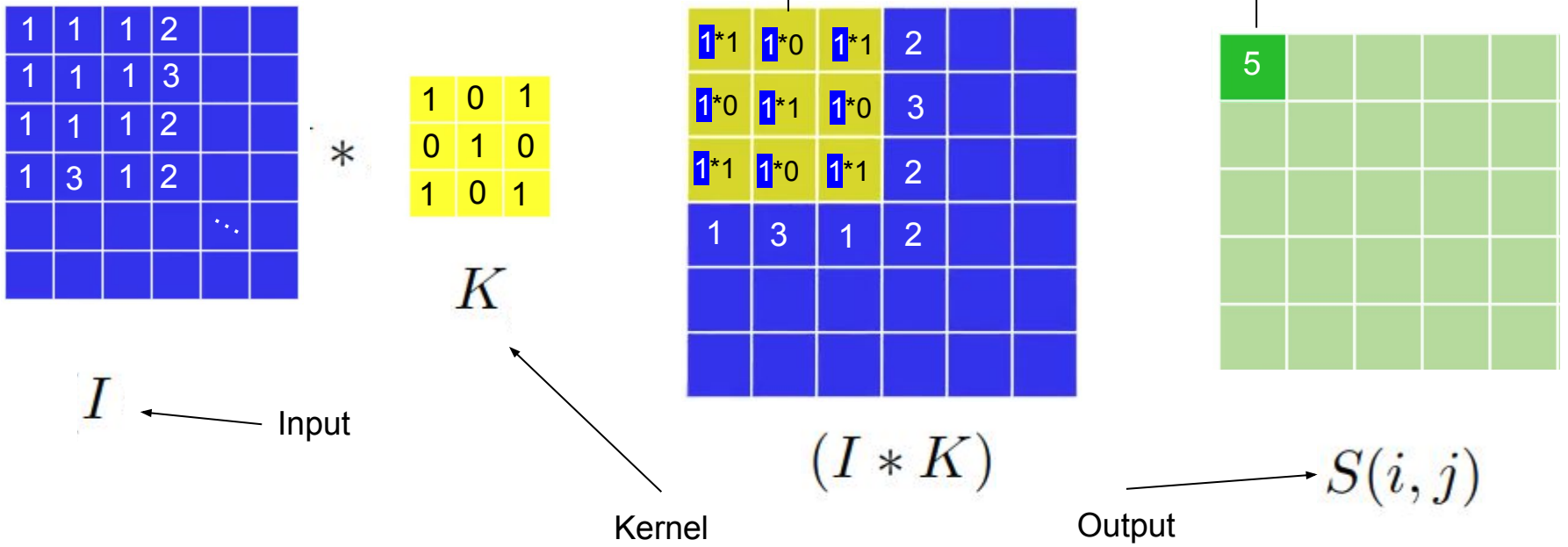
K

I ← Input

Kernel (weights)

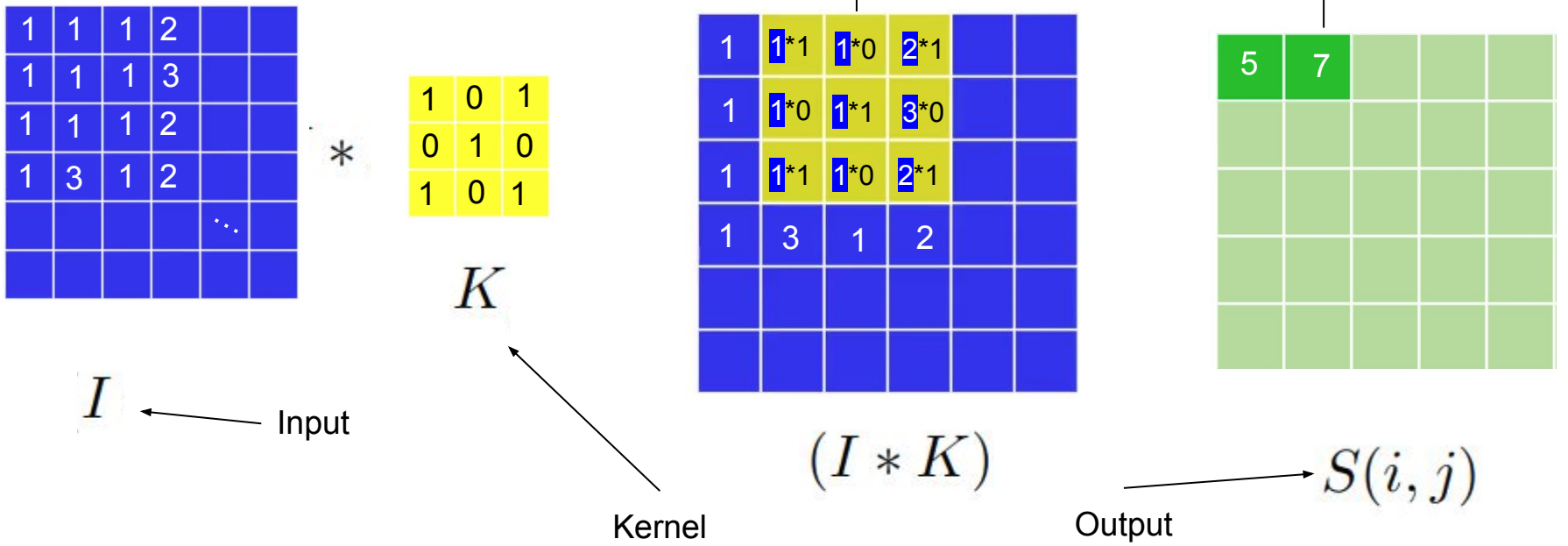
2D Convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n).$$



2D Convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n).$$

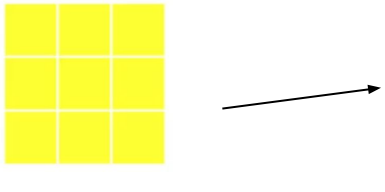


2D Convolution

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n).$$

2D Convolution

The idea is that the network can automatically **learn** the optimal parameters of the **kernel** to solve a given task.



K

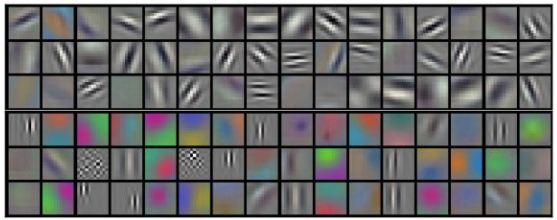
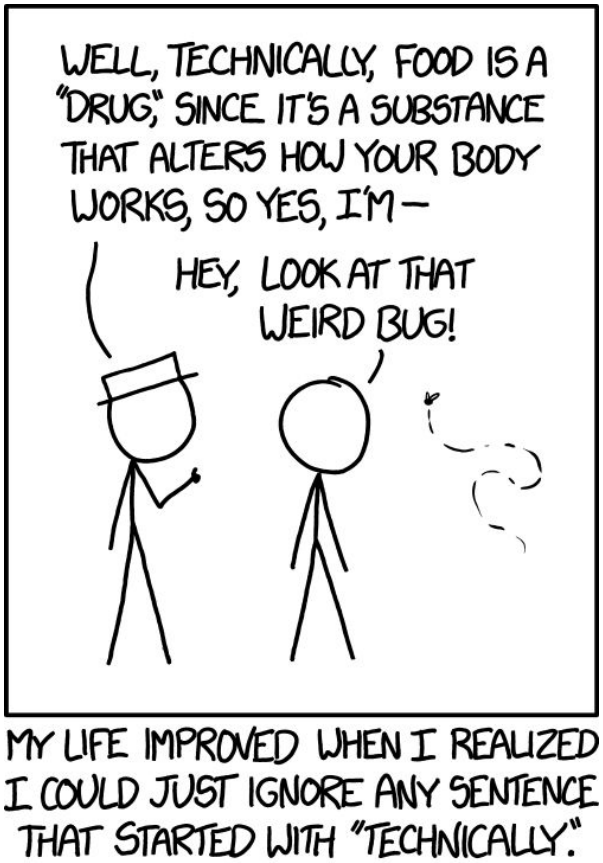


Figure 3: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images. The

Pedantic timeout

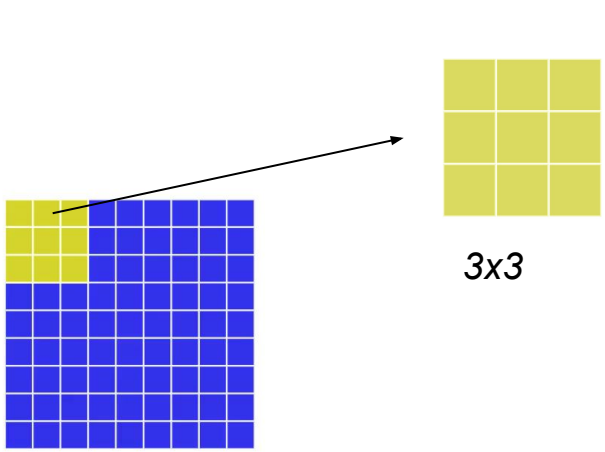
Technically, convolutions in neural networks are really implemented as **cross-correlations** (the kernels are not flipped as they should be).



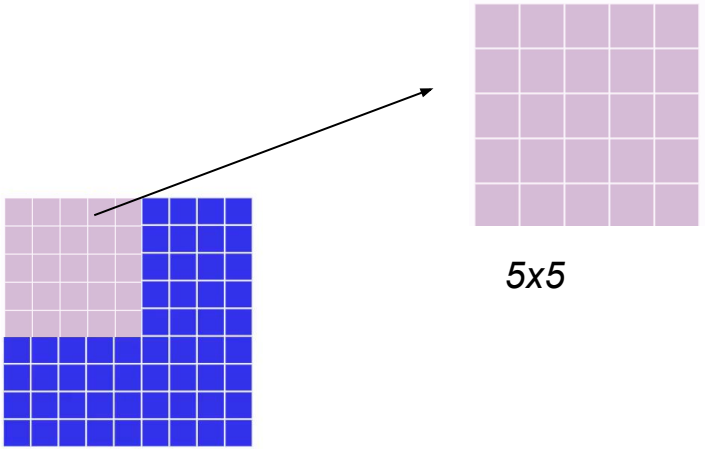
https://xkcd.com/1475/

Spatial Extent

The **size** of the kernel defines how many pixels the kernel spans. Kernels are typically square in size.



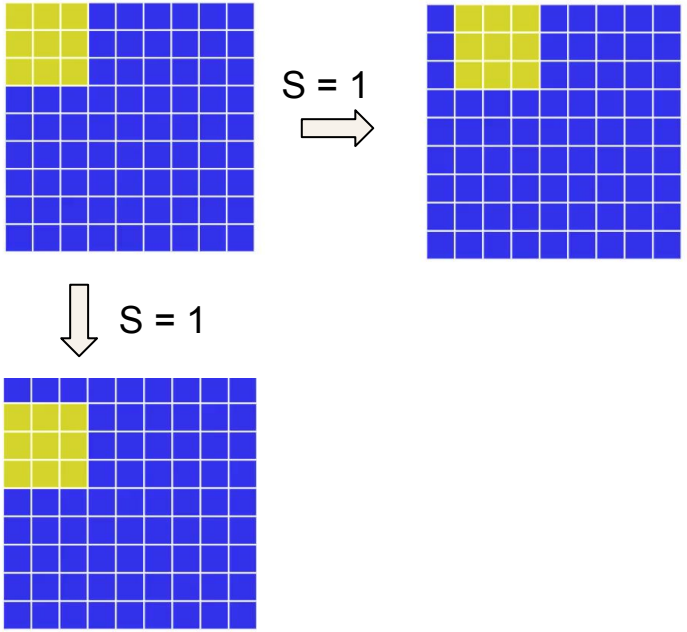
3x3

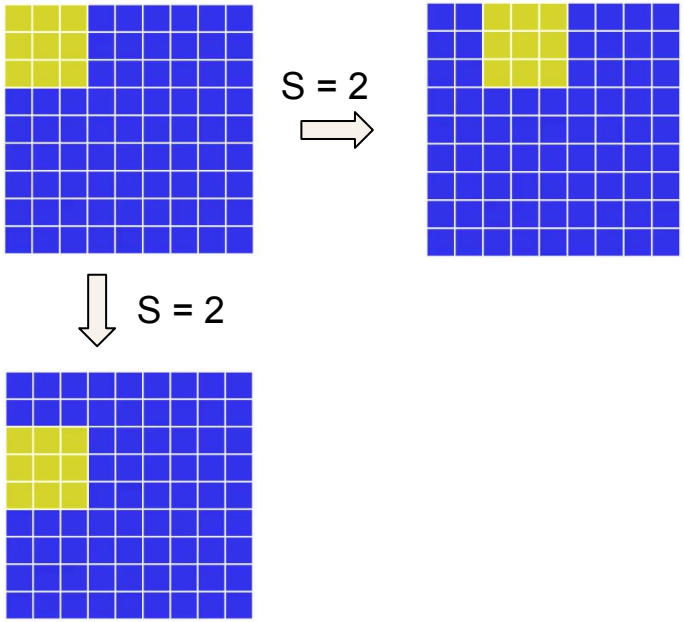


5x5

Stride

The **stride** *S*, represents the number of pixels to move in (x,y) before each operation in the convolution.





Stride

The **stride** *S,* represents the number of pixels to move in (x,y) before each operation in the convolution.

Stride = 1 in both x and y

Stride = 2 in both x and y

Stride

Stride = 2 in both x and y

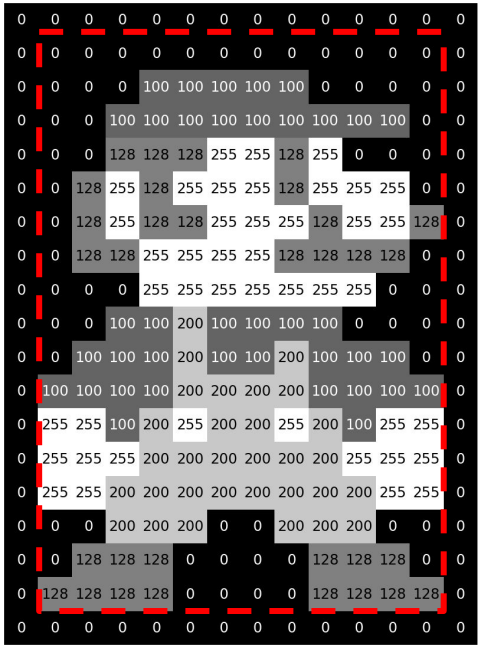
Padding

Padding defines some strategy to pad the borders (gray) of our inputs to modulate the output size.

Padding allows to maintain the same sizes at the input and output.

Padding Strategies

The most common padding strategy is **zero padding**, in which zeros are added at the borders.



Padding

Here are some of the default ways to specify padding:

Valid convolution

The kernel is allowed to visit only positions where the kernel is contained entirely within the input (i.e. no padding)

Padding

Here are some of the default ways to specify padding:

Same convolution

Just enough padding is added to keep the size of the output equal to the size of the input

Depth

We can stack kernels to compute features in **parallel**. The number of kernels will determine the number of channels of the output. The resulting feature maps are **stacked** to produce a volume.

```
Channels_in = 1
Kernel_size = 3x3
Channels_out = 5
```

Bias

Similar to MLPs, we add a constant **bias** *b* to every feature map for every kernel. The value of the bias is learned by the network.

$$y = W^{\mathsf{T}} x + b$$

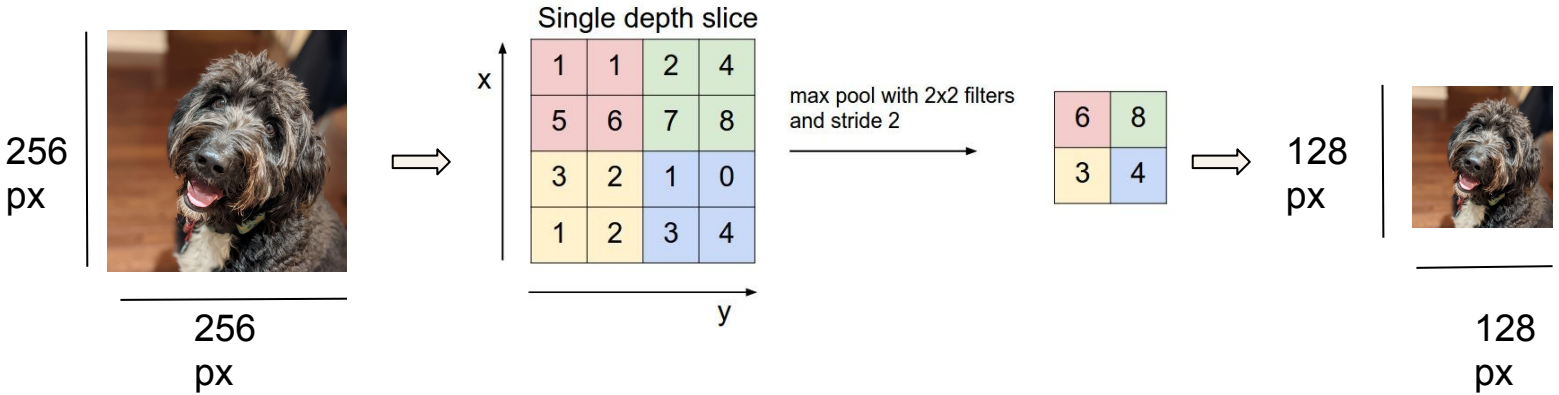
MLP: Matrix multiplication

$$y = W * x + b$$

CNN: Convolution

Pooling

Pooling is used to downsample an image. The most common type of pooling is **max-pooling**. Pooling makes the network **less sensitive** to fine details.



http://cs231n.github.io/convolutional-networks/

Pooling

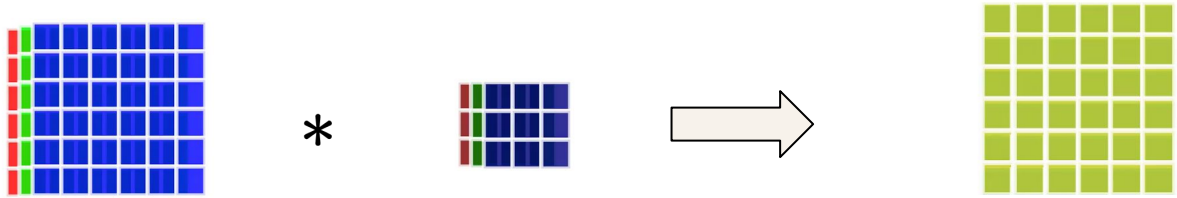
2x2 max-pooling with stride 2

1x1 Convolutions

1x1 convolutions are useful in the context of non-linear channel reduction. They allow us to preserve the width and height and reduce on the depth with few parameters.

Convolution on an RGB Image

Each kernel will have the same number of channels as the input. The output will be the sum of the feature-maps element-wise.



Input:
W_{in} x H_{in} x C_{in}

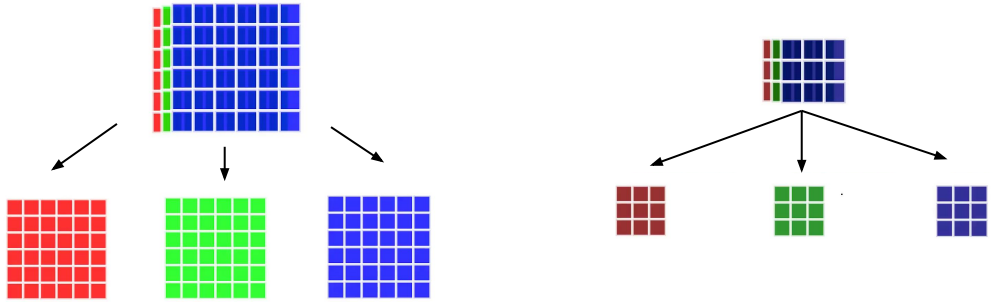
Kernel
Volume : *3* x *3* x C_{in}

Output (feature map)
Volume: W_{out} x H_{out} x 1.

Additional material

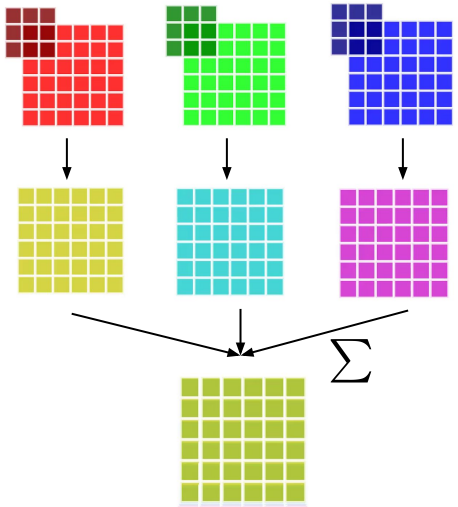
Convolution

Separate the inputs and kernels by channel



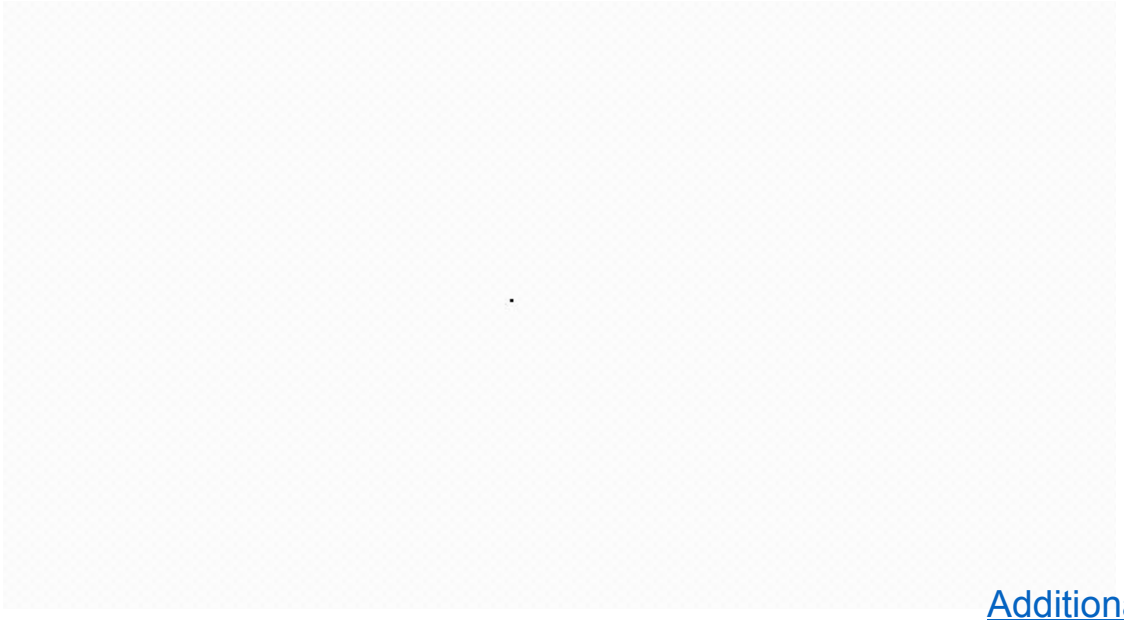
Perform 2D convolution on each channel

Sum the results to get a W_{out}x H_{out} feature map



Convolution on an RGB Image

Here is an animation showing the entire procedure of a convolution for a single kernel:



Additional material

From volume to volume

The number of kernels D determines the number of output channels C_{out}. Here, we are using 4 kernels, each of volume $3x3xC_{in}$. We are going from an *6x6x3* RGB image to an 6x6x4 feature map.

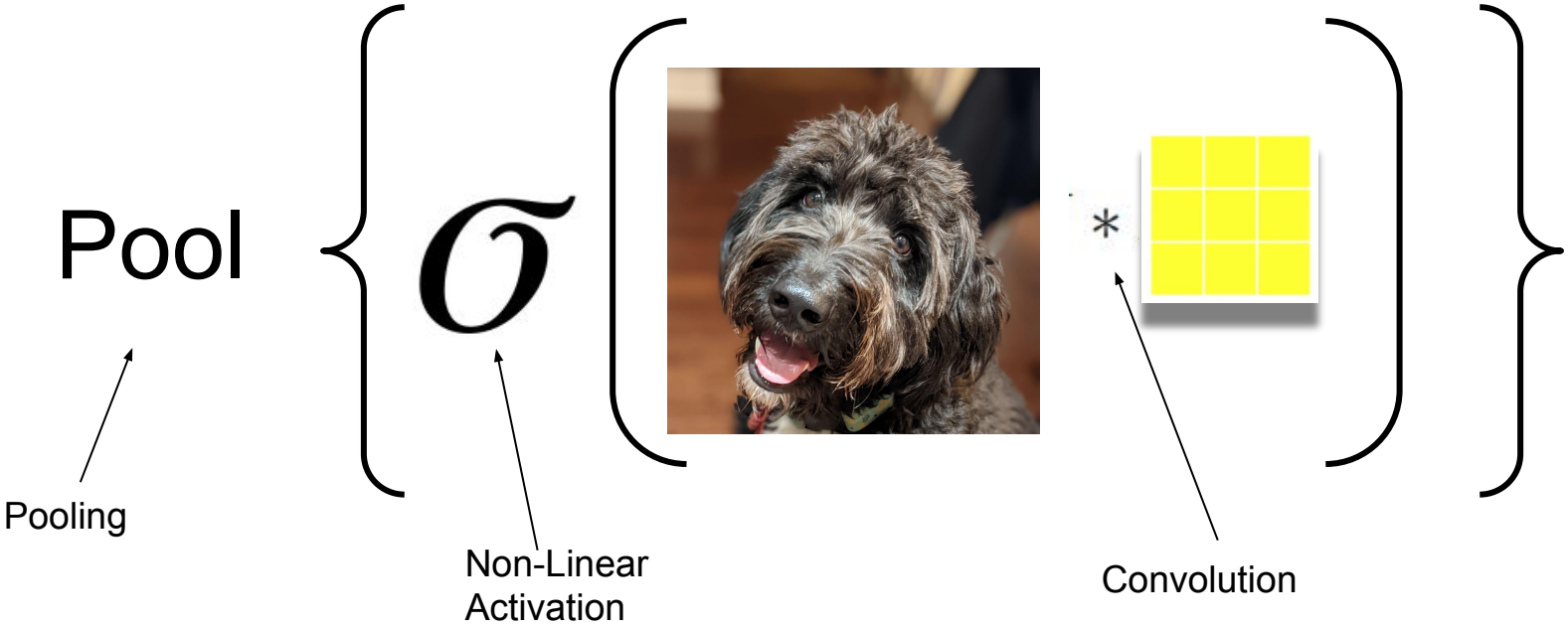
Input (RGB image),
W_{in} x H_{in} x C_{in}
(6x6x3)

Kernels
D x *3* x *3* x C_{in}=
(***4*** x 3x3x3)

Feature Map
W_{out} x H_{out} x D =
(6x6x**4**)

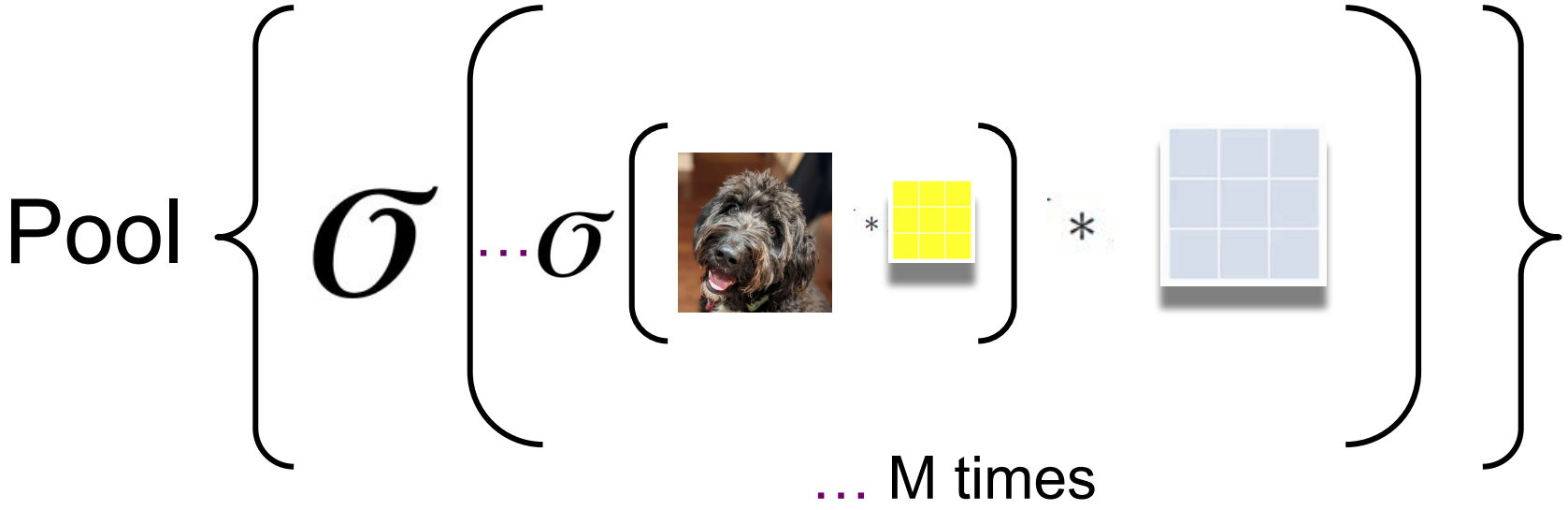
Building Blocks

Here is a common building block for CNNs:



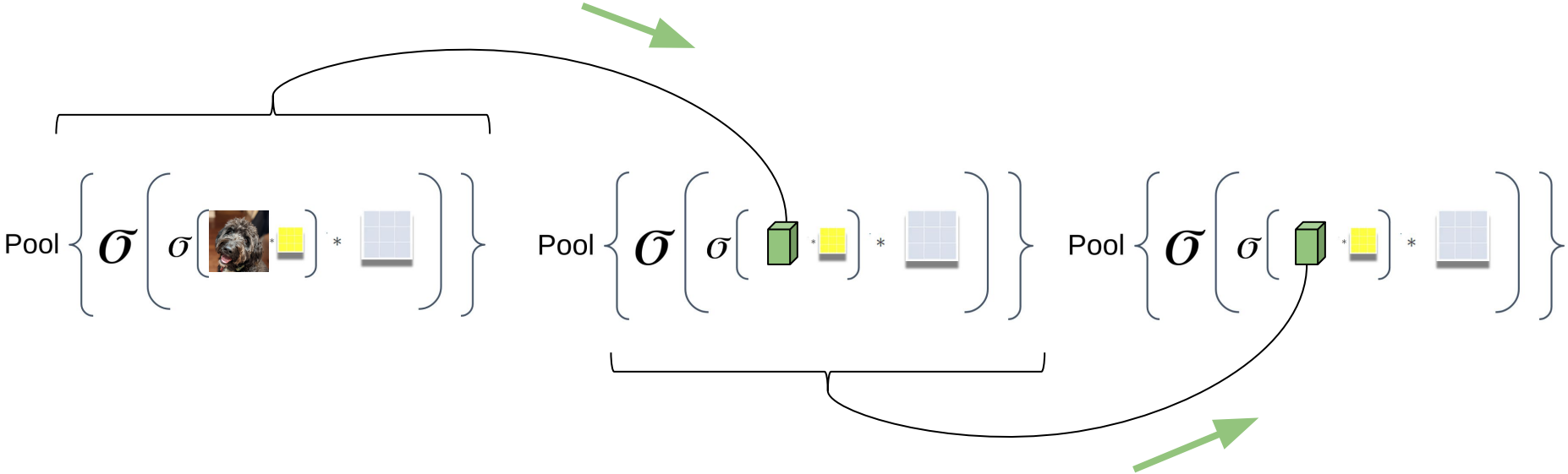
Building Blocks

We can have many *Convolutions* + *Activations* chained before pooling:



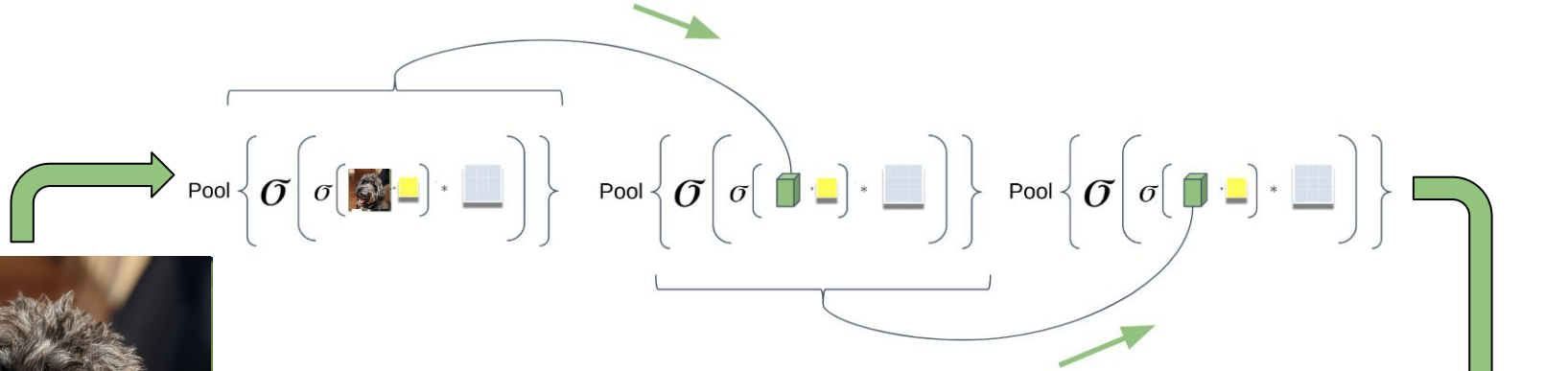
Building Blocks

… And we can have many of these blocks chained together



Building Blocks

These blocks allow us to map from one volume (RGB) to another.

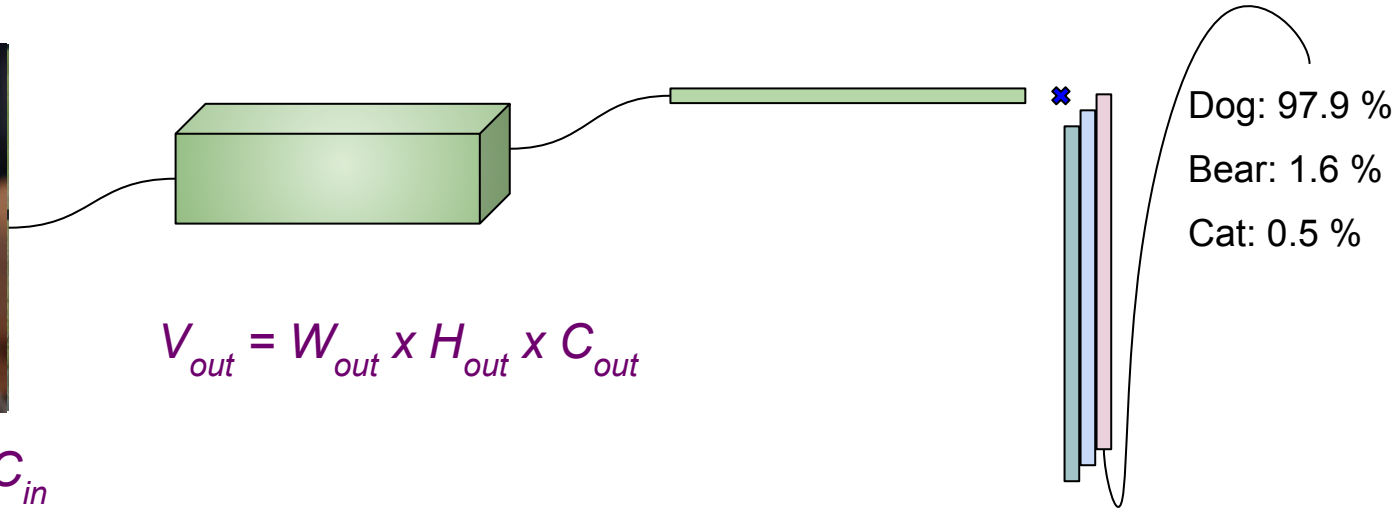


$W_{in} \times H_{in} \times C_{in}$

$W_{out} \times H_{out} \times C_{out}$

Building Blocks

To get to a probability distribution, we can **aggregate** the feature map (average or flatten) and then use an MLP followed by a **softmax** to obtain a probability distribution that sums to 1.



$V_{out} = W_{out} \times H_{out} \times C_{out}$

$V_{in} = W_{in} \times H_{in} \times C_{in}$

LeNet

One of the first successful applications of CNNs was the **LeNet** architecture. It was used to classify handwritten digits and consisted of 2 convolutional layers and 2 fully-connected (MLP) layers.

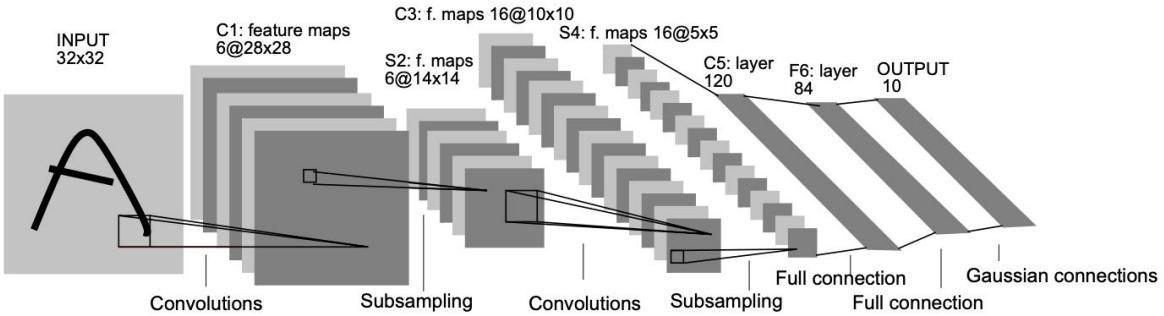


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

source

Loss Function

The key ingredients needed for implementing an image classification algorithm are:

1. A labelled dataset
2. A model
3. **A loss function**
4. An optimizer

Loss Functions

Loss Function

A **loss function** quantifies the performance (scalar) of the model on a given task. The loss function needs to be **differentiable** w.r.t model parameters.

The goal is to **minimize** the loss $L(\theta)$.

Loss Function

The **binary cross entropy** (BCE) loss is used in binary classification problems. It can be defined as:

$$L(y, p) = -[ylog(p) + (1 - y)log(1 - p))]$$

Where ***y*** is the class label, and ***p*** is the predicted probability (model output) of belonging to the class.

Loss Function

```
Cat: 0
Dog: 1
```

The loss indicates how well the prediction **p** of our model *f* performs on a given image $\mathbf{X_i}$ with label $\mathbf{y_i}$.

Here, the model predicts **correctly** resulting in a **low loss**.

$$f_\Theta(\mathbf{X_i}) = \mathbf{p}$$

$\mathbf{y_i} = 1$ (dog)

Dog: 97.9 %
Cat: 2.1%

$$L(y, p) = -[ylog(p) + (1 - y)log(1 - p))]$$

$$L(1, 0.979) = 0.0212$$

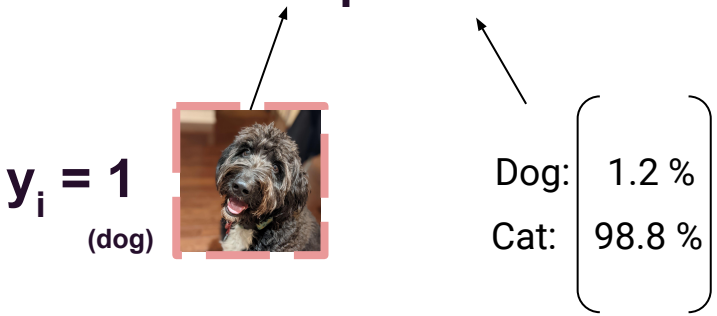
Loss Function

Cat: 0
Dog: 1

The loss indicates how well the prediction **p** of our model *f* performs on a given image $\mathbf{X_i}$ with label $\mathbf{y_i}$.

Here, the model predicts **incorrectly** resulting in a **high loss**.

$$f_\Theta(\mathbf{X_i}) = \mathbf{p}$$



$$L(y, p) = -[ylog(p) + (1 - y)log(1 - p))]$$

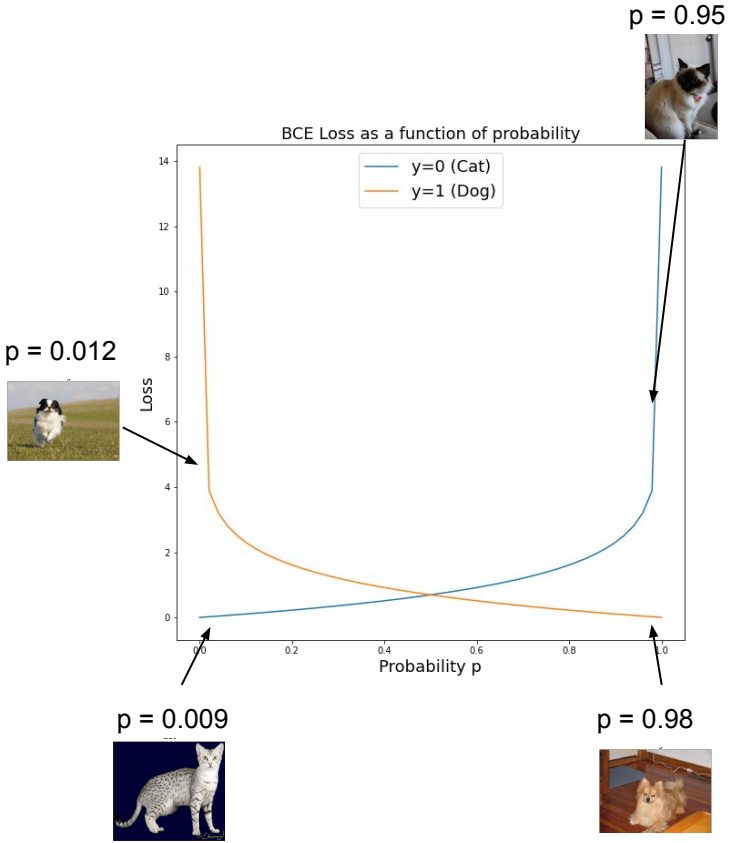
$$L(1, 0.012) = 4.42$$

Loss Function

By **summing** the loss over different samples, we can capture the **performance** of our model and **optimize** for this value.

Here, the loss of our model is

```
L(0, 0.009) + L(0, 0.950) +
L(1, 0.012) + L(1, 0.980) = 7.44
```



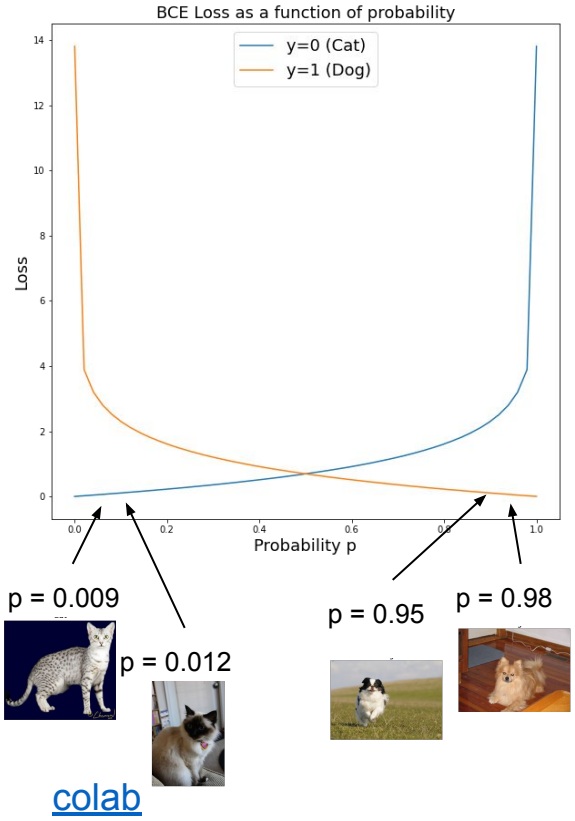
colab

Loss Function

By **summing** the loss over different samples, we can capture the **performance** of our model and **optimize** for this value.

Here, the loss of our model is

```
L(0, 0.009) + L(1, 0.950) +
L(0, 0.012) + L(1, 0.980) = 0.092
```



colab

Loss Function

Binary cross-entropy can be extended to handle **multiple classes**.

Using the more **general** cross-entropy definition, multiple classes are possible, but only **one class** can be the correct class.

$$-\sum_{c=1}^{N} y_c log(p_c)$$

Loss Function

Binary cross-entropy can be extended to handle **multiple classes**.

We can predict **multiple classes** using parallel binary cross-entropy objectives when classes are not mutually exclusive.

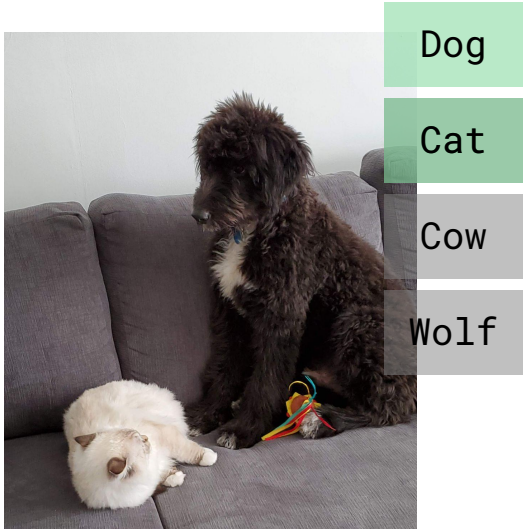


Image Classification

The key ingredients needed for implementing an image classification algorithm are:

1. A labelled dataset
2. A model
3. A loss function
4. **An optimizer**

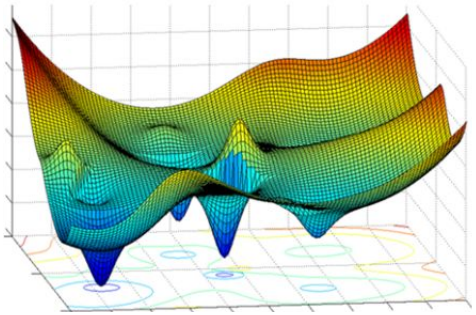
Optimizers

Optimizers

So how do we find the **optimal** set of parameters (weights and biases) for a model?

Optimizers

Using an **optimizer**, we can find model parameters Θ that **minimize** the loss function by using the **gradient of the loss** w.r.t model parameters.



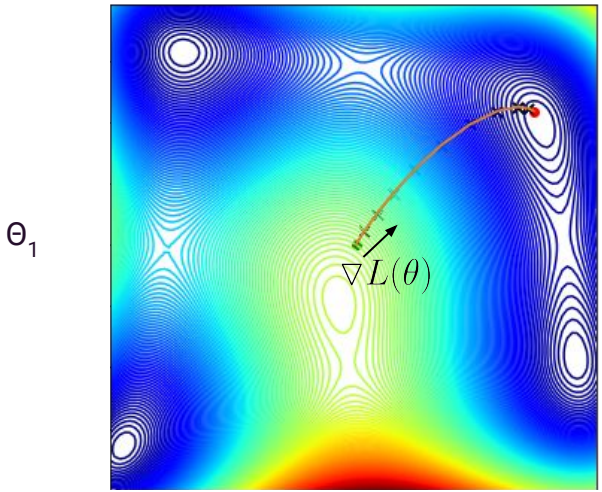
Gradient Descent

$$L(\theta_1, \theta_2) = \left(\theta_1^2 + \theta_2 - 11\right)^2 + \left(\theta_1 + \theta_2^2 - 7\right)^2$$

This can be done using e.g. **gradient descent** which uses the gradients of the loss to find optimal parameters:

$$\theta := \theta - \eta \nabla L(\theta)$$

Where η represents the **learning rate** or the step size and $\nabla L(\theta)$ represents the gradient w.r.t. model parameters.

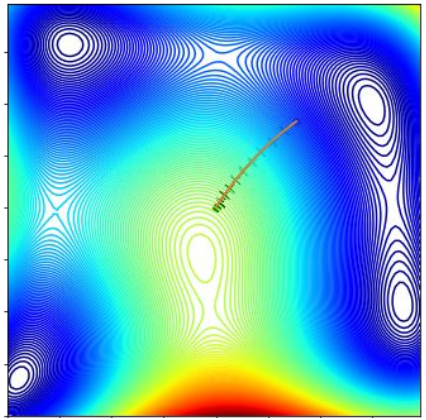


colab

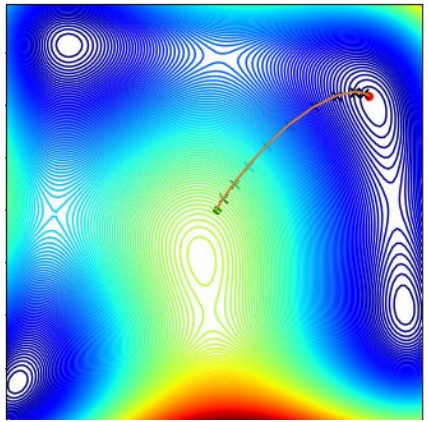
Learning Rate

Finding an **adequate** learning rate is crucial for model convergence.

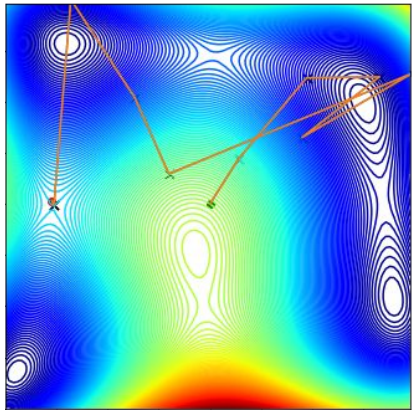
Too low (lr = 0.005)



Good (lr = 0.01)



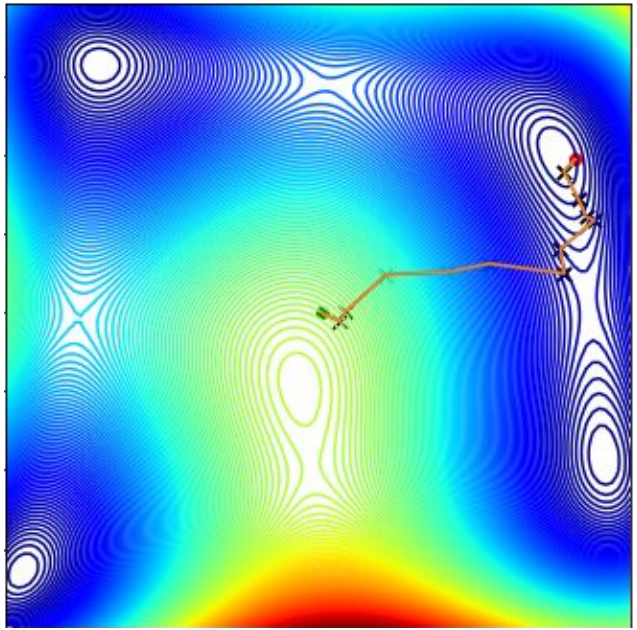
Too high (lr = 0.04)



Stochastic Gradient Descent

Optimizers like Stochastic Gradient Descent (SGD) **approximate** gradient descent by using **batches of data** to approximate the gradient $\nabla L(\theta)$.

This makes the actual gradient computation **faster** but also **noisier**. This can help escape saddle points and local minima.

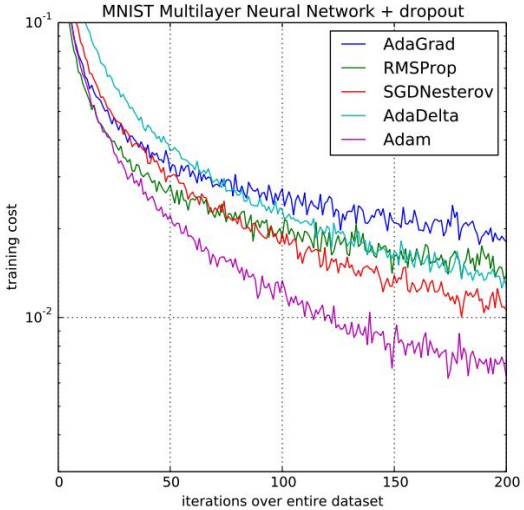


colab

Adam

There have been many **improvements** to SGD over the years to make it more robust and sample efficient.

Adam is a popular variation which has been shown to work very well in practice compared to SGD.



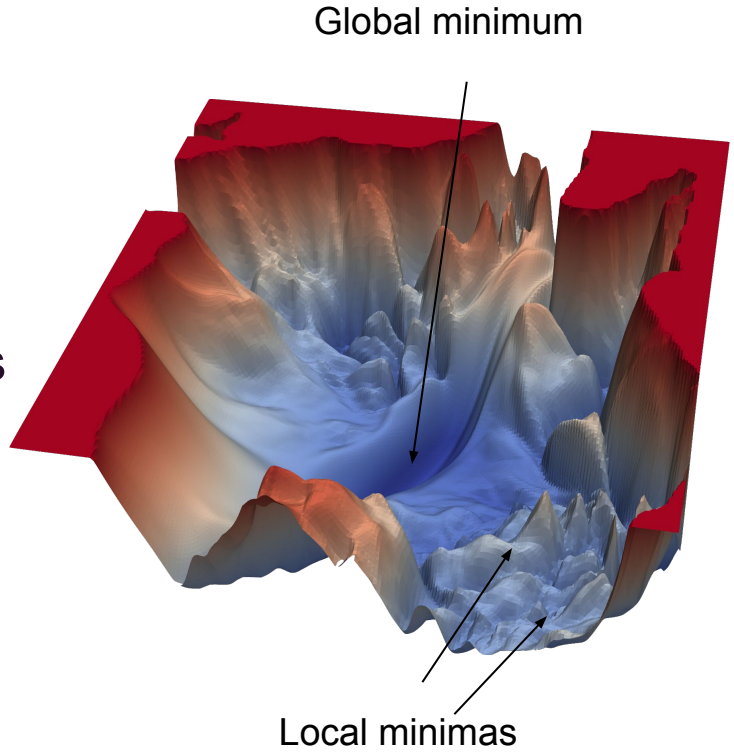
https://arxiv.org/abs/1412.6980

Loss Surface

Loss functions of neural networks are rarely smooth and can exhibit **complex surfaces**.

There can be many **local minimas**, and it is not necessarily possible to know if we are in a global minimum.

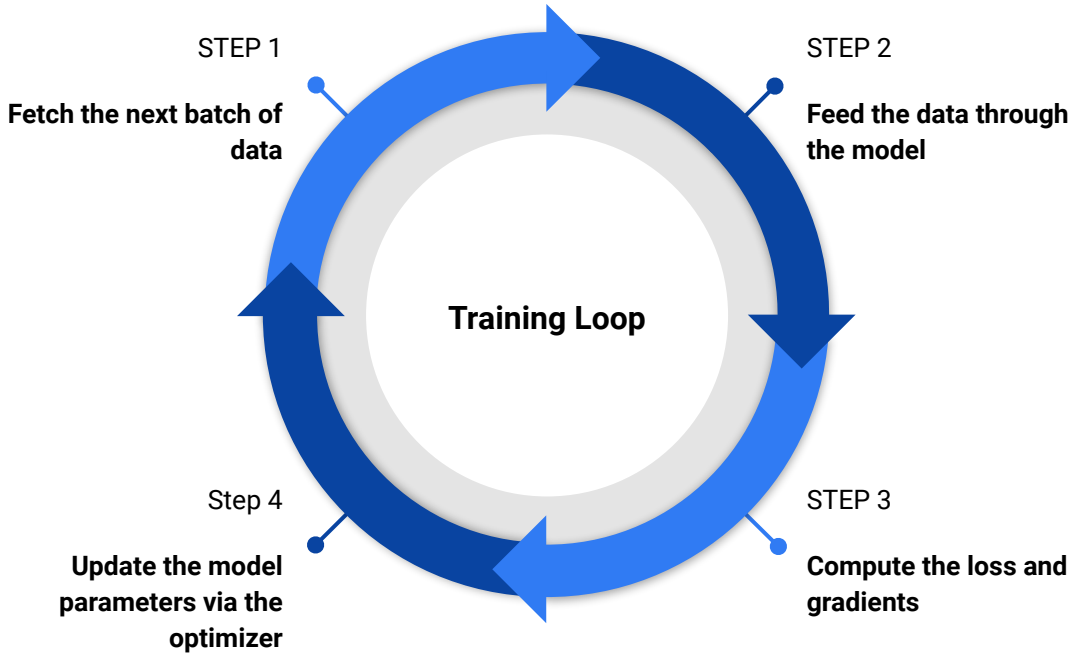
The **stochasticity** in optimizers can help escape local minimas.



Training Models

Training Loop

Once we have defined a model, a dataset, a loss function and an optimizer, we can implement a **training loop**.



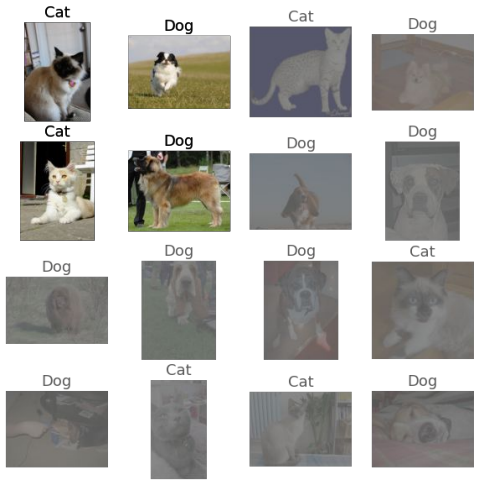
Step 1

Fetch a subset of data, **a batch,** from our dataset, which is defined by a **`batch_size`** parameter.

Here, **`batch_size`** = 4.

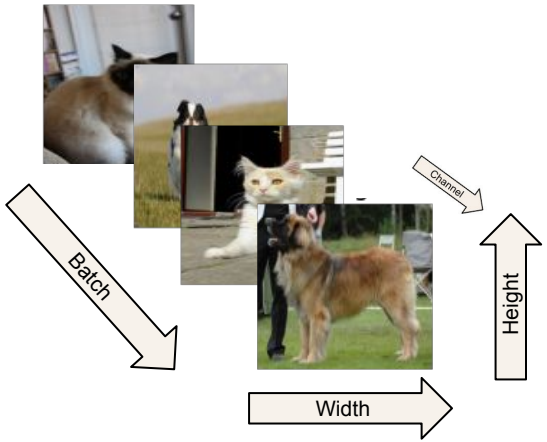
Step 1

The images in a batch are **collated** across a **batch** dimension. Images should be resized for all dimensions to agree.



4D Tensor of shape:

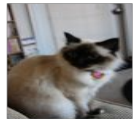
```
[batch, width, height, channel]
   4,    128,  128,       3
```

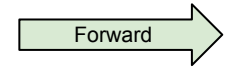


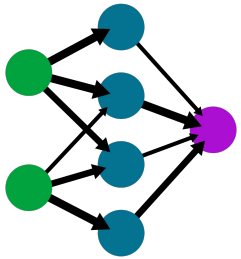
IVADO Mila

Step 2

Predictions for each image are processed in **parallel** by the model in a forward pass.







$f(X_1) = 0.221$ ✅

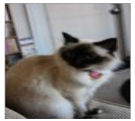
$f(X_2) = 0.813$ ✅

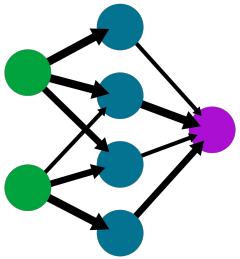
$f(X_3) = 0.793$ ❌

$f(X_4) = 0.121$ ❌

Step 3

The **loss** and **gradients** for the batch are computed.





Backward

Forward

$f(X_1) = 0.221$ ✅

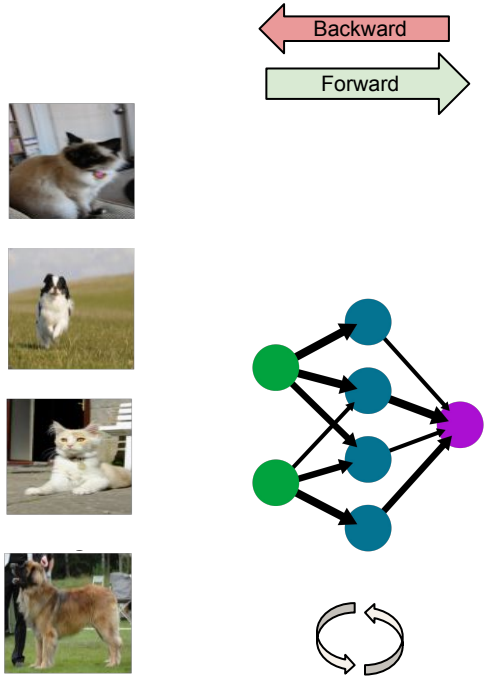
$f(X_2) = 0.813$ ✅

$f(X_3) = 0.793$ ❌

$f(X_4) = 0.121$ ❌

Step 4

The model parameters are **updated** by the optimizer for the batch.

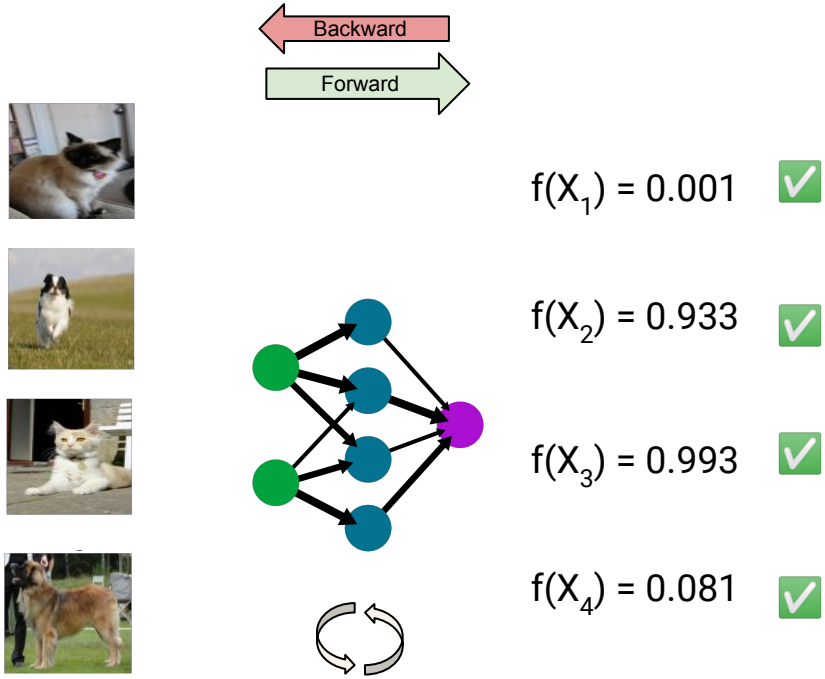


$$\theta := \theta - \eta \nabla L(\theta)$$

Step 4

Each pass on the dataset is called an **epoch**.

The training procedure is repeated on the entire training dataset over **many epochs** until convergence.



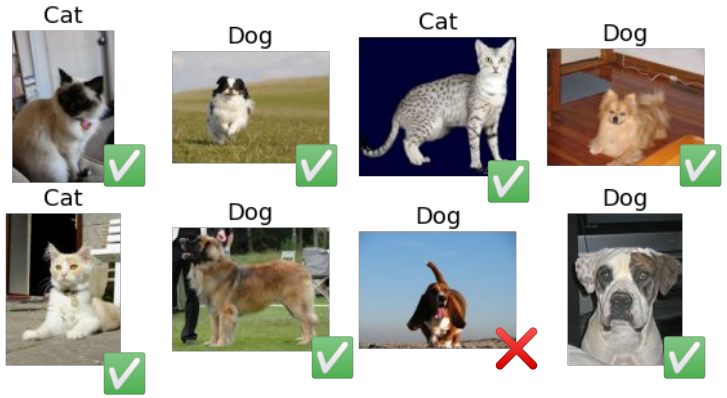
$$\theta := \theta - \eta \nabla L(\theta)$$

Generalization

Generalization

To goal of a classification model is to be able to **generalize** to previously unseen data.

Training Set

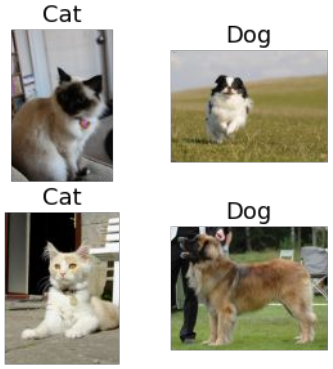


Unseen Data

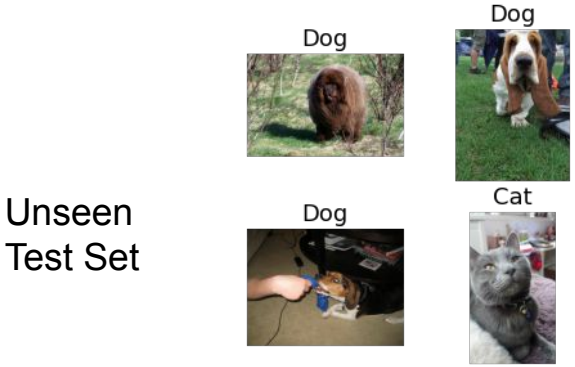
Test Set

To approximate the generalization of a model on a task, it is evaluated on a separate **test dataset** that is independent from the training data.

If the model performs similarly on the training data as on the test data, we say the model has **generalized** to the task.



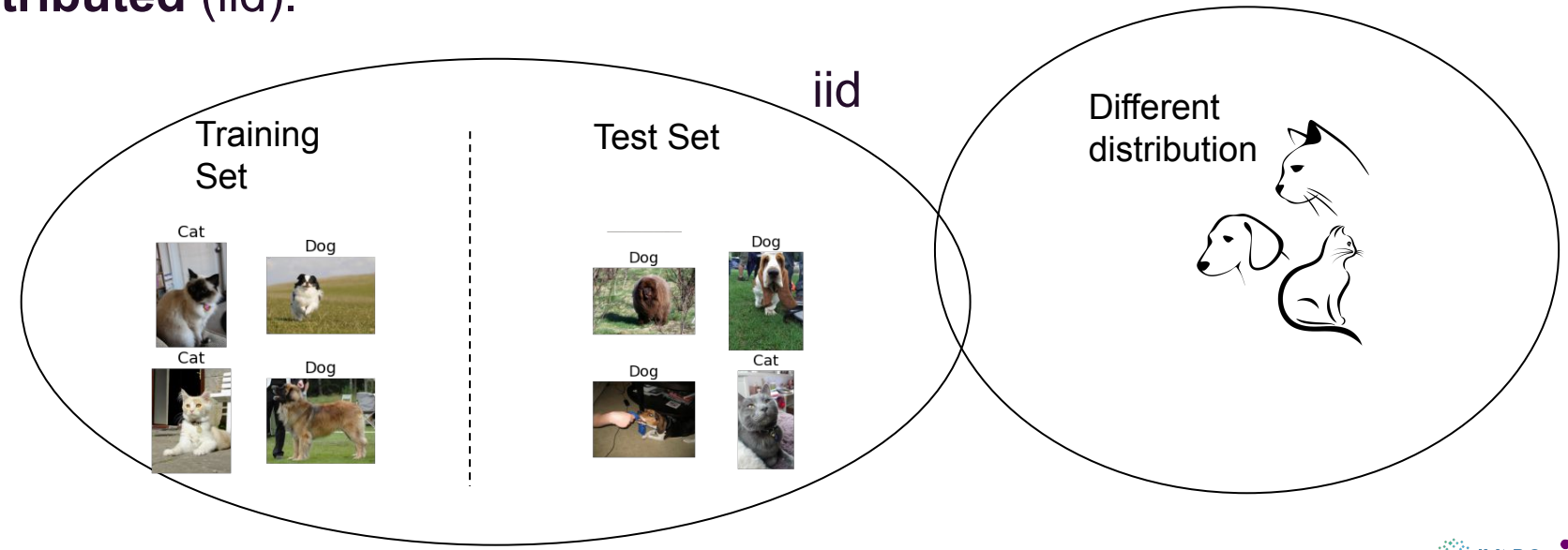
Training Set



Unseen Test Set

IID Assumption

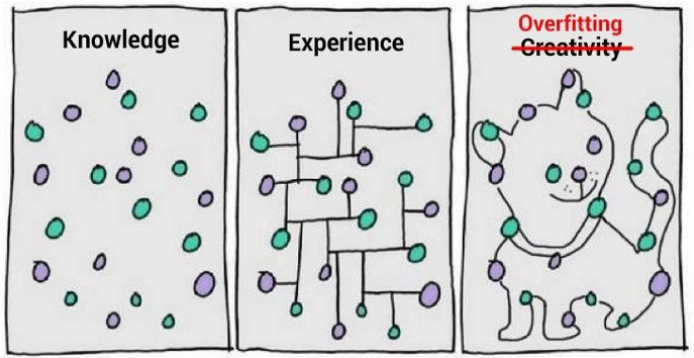
We must assume that the training set and test set come from the same data distribution. We assume they are **independent** and **identically distributed** (iid).



Overfitting

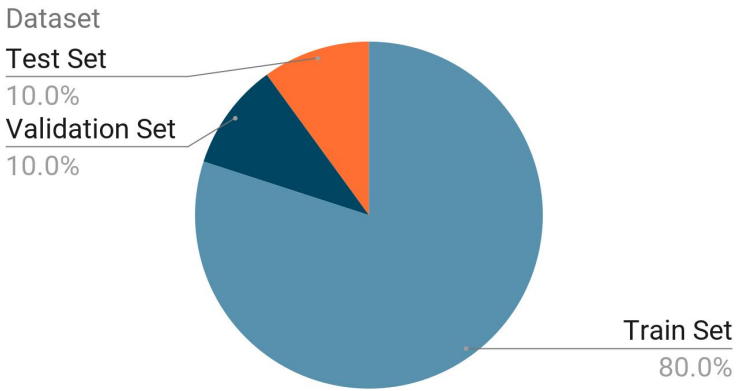
When training a model, it is possible for the model to effectively memorize the training set. This is called **overfitting**.

A model will typically **overfit** the training set if trained long enough and if it has enough capacity. This is the opposite of generalization.



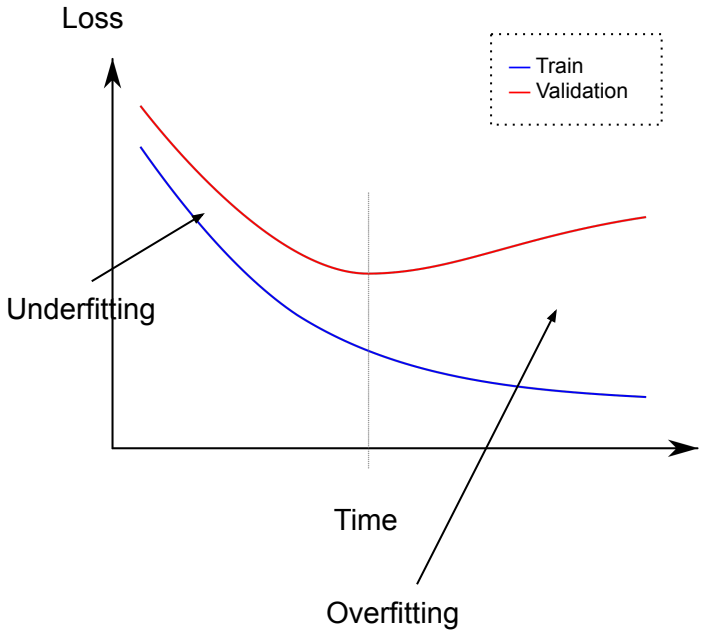
Data Splits

In order to **monitor** overfitting during training, a portion of the training data can be used as a **validation set** (~20-40%).



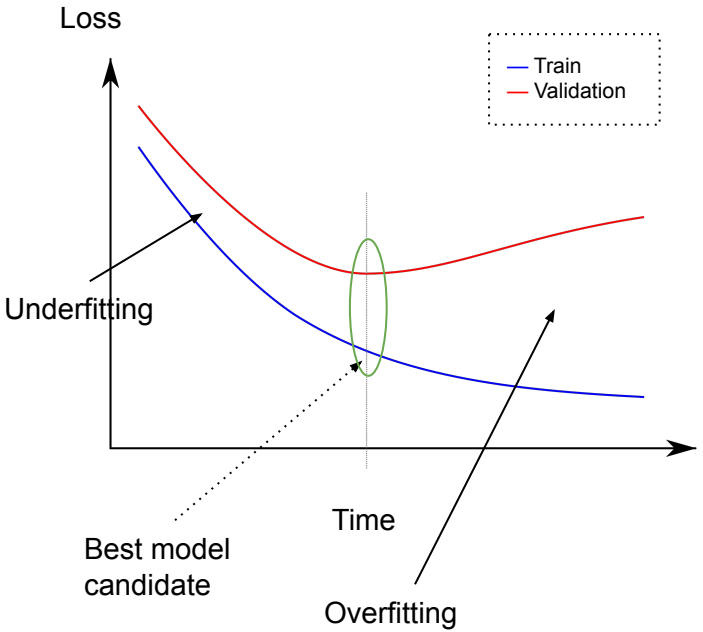
Overfitting

When the error between train and validation starts **increasing**, the model is likely overfitting.



Overfitting

We can use the model which minimized the error on the validation set to evaluate the **test set** and measure its **generalization**.

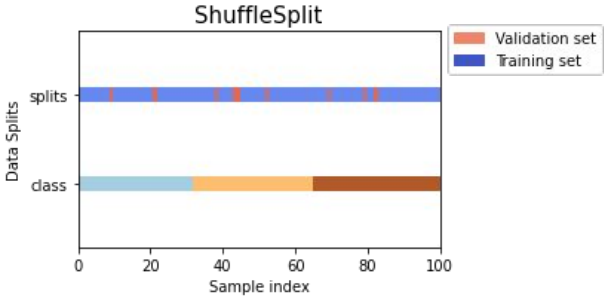


Data Splitting

When constructing a validation set, it is important to keep original **data distributions** in mind.

Ideally, the data should be **distributed similarly** in the validation as in the training set (IID hypothesis).

Assuming we have **balanced classes**, simply **shuffling** the data can be sufficient.

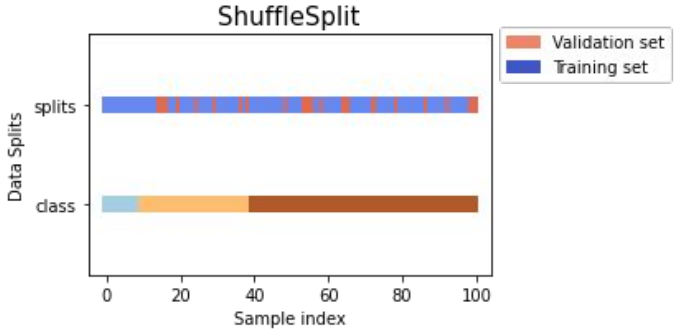


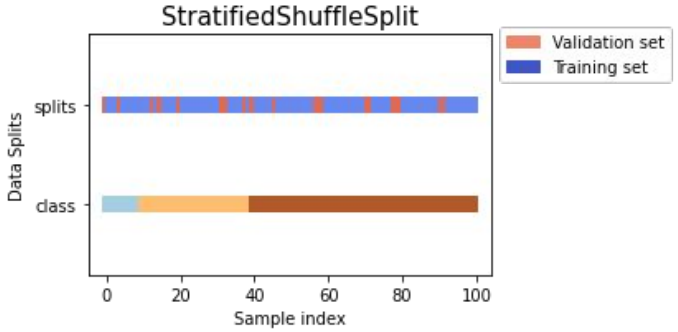
colab

IVADO Mila

Data Splitting

If the classes aren't balanced, other splitting techniques like **stratification** can be used to mimic the original distribution.

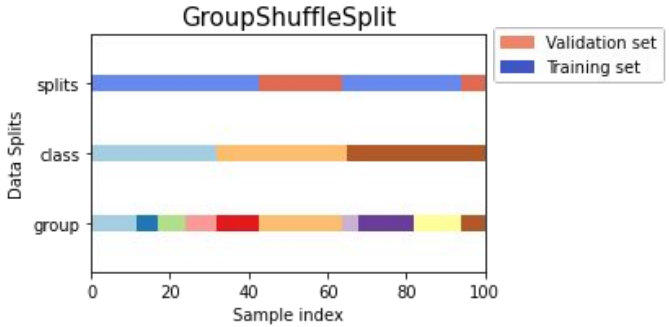




colab

Group Splitting

If there exists **groups** within the data (e.g. individuals with multiple data points), it can be useful to **stratify** on the groups themselves.



colab

IVADO Mila

Model evaluation

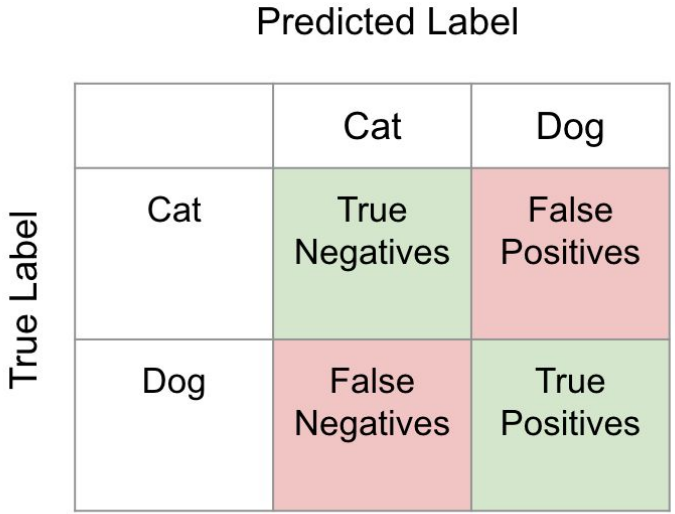
Classification Outcomes

In a **binary classification** task, we can have 4 possible outcomes for a given prediction and target.

Target	Prediction	
🐈	🐶	❌
🐶	🐈	❌
🐈	🐈	✅
🐶	🐶	✅

Confusion Matrix

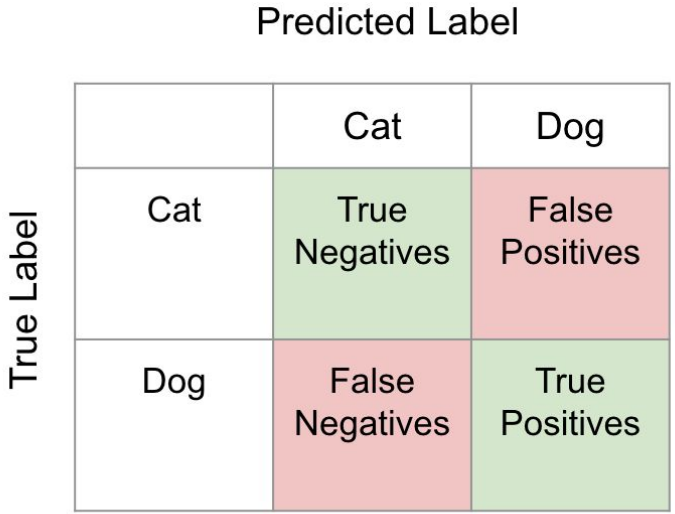
We can keep track of the different classification outcomes in a **confusion matrix**.



Accuracy

Accuracy computes the ratio of correct predictions with overall predictions made.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

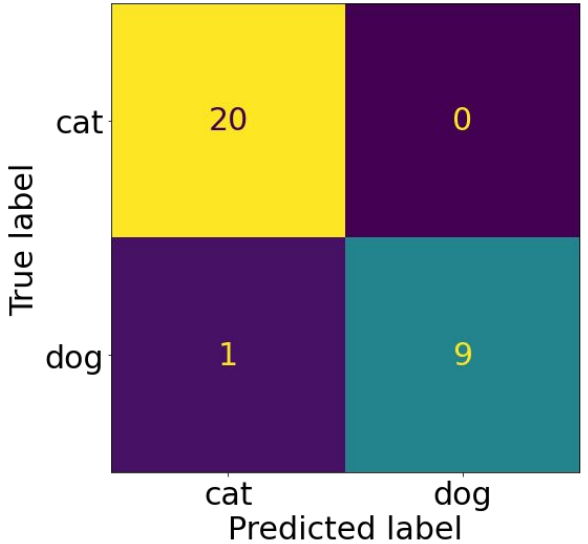


Accuracy

In the following confusion matrix, we have a total of 30 samples that were evaluated.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

= (20 + 9) / (20 + 9 +1 + 0) = 96%

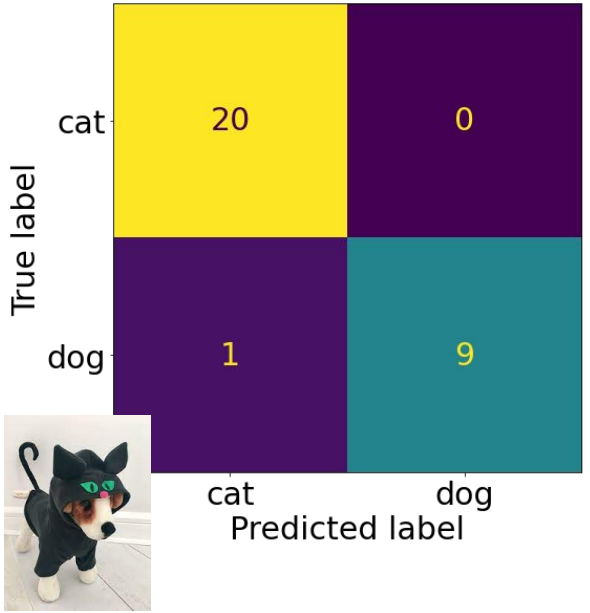


Accuracy

In the following confusion matrix, we have a total of 30 samples that were evaluated.

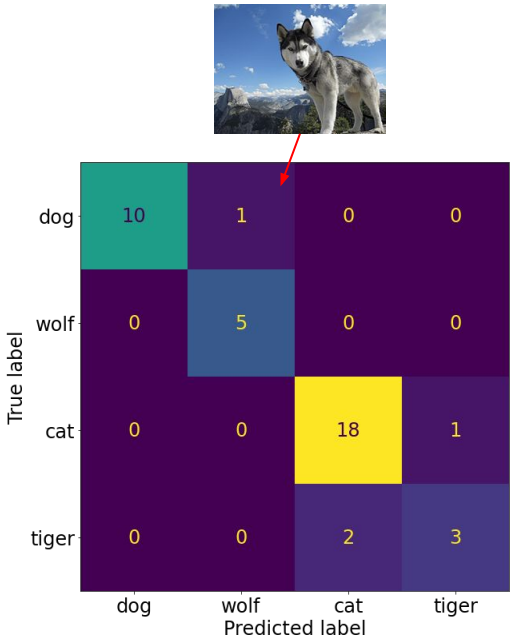
$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

= (20 + 9) / (20 + 9 +1 + 0) = 96%



“A dog that was wrongly predicted as a cat”

Confusion Matrix



Confusion matrices can be extended to visualize multiple classes and help diagnose problems/confusions of the classifier.

Here, accuracy = 90%.

Metrics

In cases with unbalanced data, such as rare disease predictions, accuracy can be **misleading**.

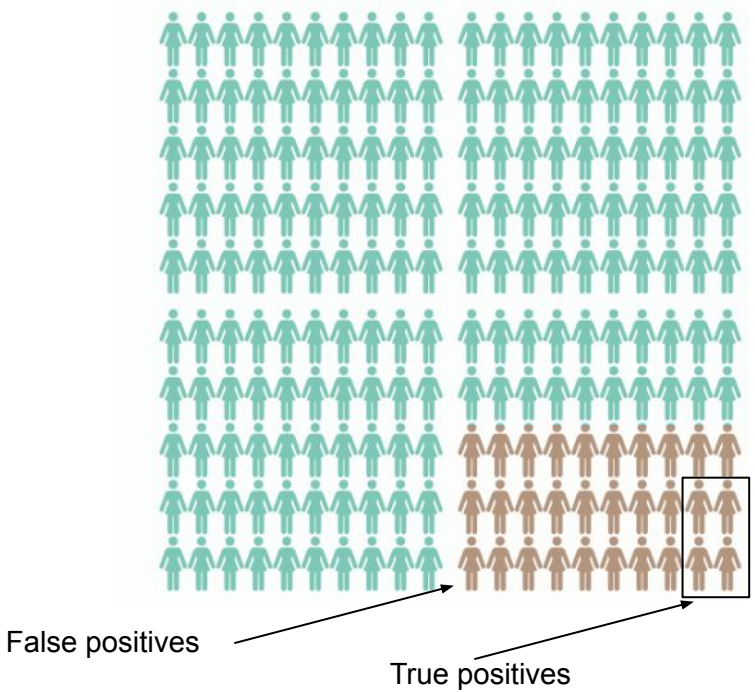
Consider a rare disease detector that is **hard-coded** to always predict “Healthy”:

Here, accuracy is 196/200 = **98%**, but the model is effectively **useless**.

Metrics

Diagnosing too many healthy people as diseased, **false positives**, can also lead to patients receiving treatment they don't otherwise need.

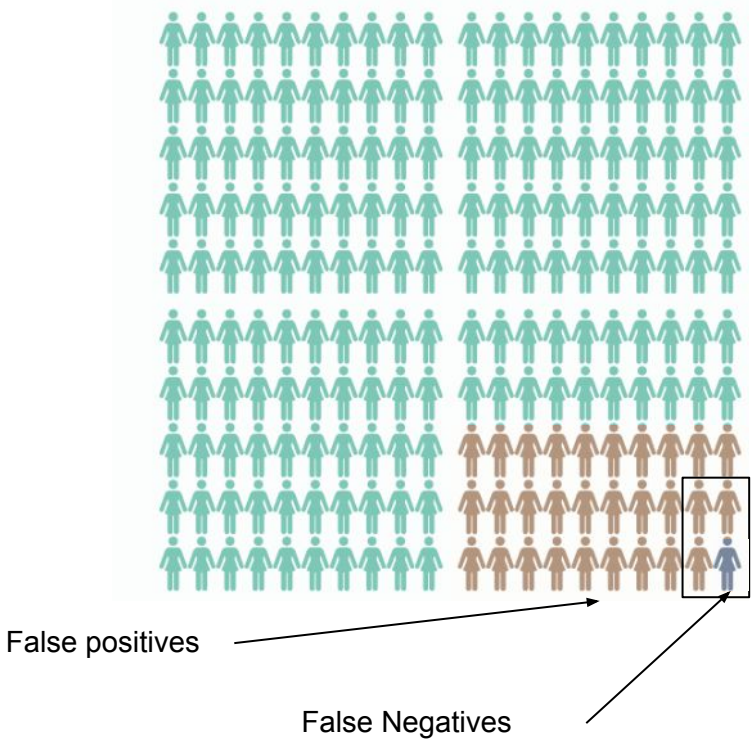
Here, accuracy = 174 / 200 = 87%



Metrics

Missing someone with the disease, **false negatives**, can also lead to disastrous outcomes for that patient.

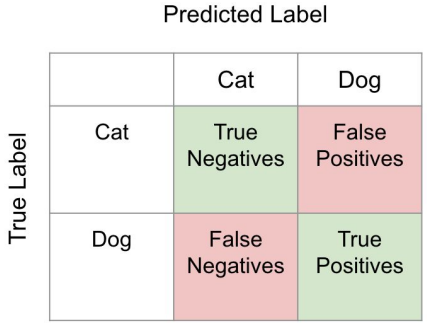
Here, accuracy = 173 / 200 = 86.5%



Metrics

When dealing with unbalanced data, other metrics such as **precision** and **recall** can be considered.

- **Precision** is the fraction of detections reported by the model that were correct.
- **Recall** (sensitivity) is the fraction of true events that were detected.
- **F1 score** is the geometric mean of precision and recall.



$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

$$F_1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{\text{tp}}{\text{tp} + \frac{1}{2}(\text{fp} + \text{fn})}$$

more metrics

Data Augmentation

Data Augmentation

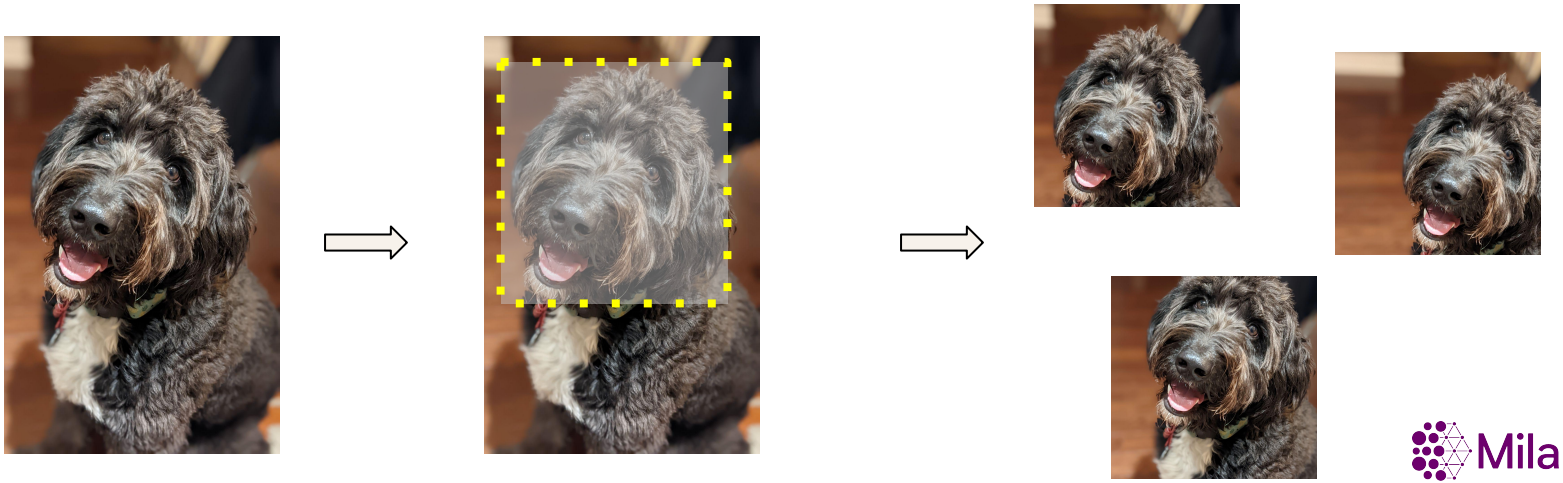
One trick to help **artificially** augment the size of a dataset is **data augmentation**.

We apply **transformations** to our inputs such that the associated labels remain unchanged.

Data Augmentation

Cropping:

- We randomly pick a square patch in our image and use the crop as input to our network.



Mila

Data Augmentation

Mirroring:

- We mirror the image along a given axis (here along the vertical axis). This only works when orientation doesn't matter.

Mila

Data Augmentation

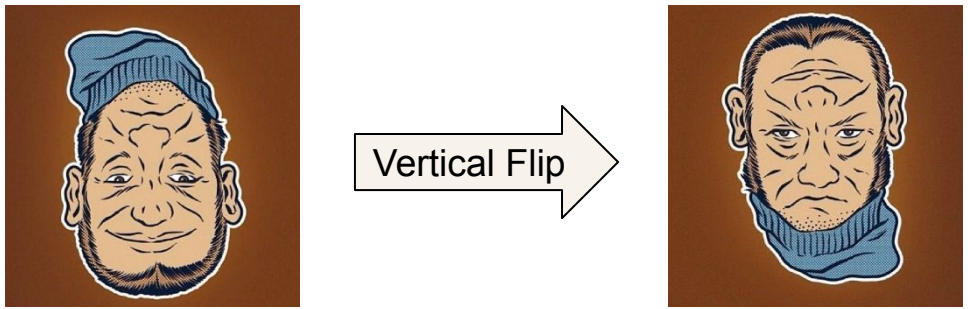
Rotation:

- We first rotate our image then crop a patch of the rotated image. Padding can be used.

Mila

Data Augmentation

There exists many other types of augmentations. It is important to ensure that the transformations will not **change** the predictions that will follow:



"happy"

"happy"?

Coding Activity

Rock, paper, scissors

In the next session, we will be implementing the concepts covered today in a **programming activity**.

We will be training **neural networks** to classify images of hands playing rock, paper, scissors.

Questions?

jeremy.pinto@mila.quebec
pierreluc.stcharles@mila.quebec