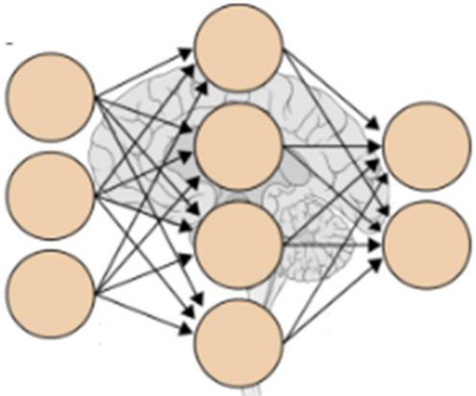
APR - Aprendizagem Profunda e por Reforço - 2025/26

Deep and Reinforcement Learning

P31 – CNN & CNN

Jorge Henriques, João Nuno Correia

jh@dei.uc.pt, jncor@dei.uc.pt



Departamento de Engenharia Informática

Faculdade de Ciências e Tecnologia

Main Goals

- Use our library on other problems (e.g., the Iris dataset)
- Introduction to the PyTorch library

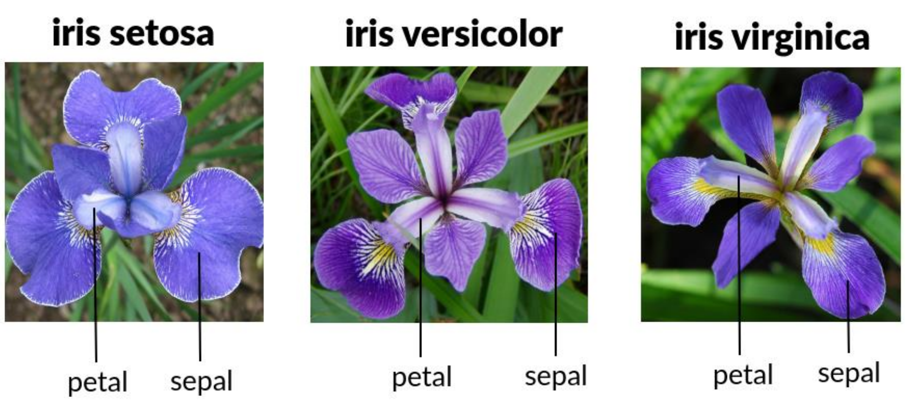
- PyTorch to define and train a Multilayer Neural Network (**MLN**)
- PyTorch to define and train a Convolutional Neural Network (**CNN**)

Contents

Iris dataset

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
Y = iris.target
```

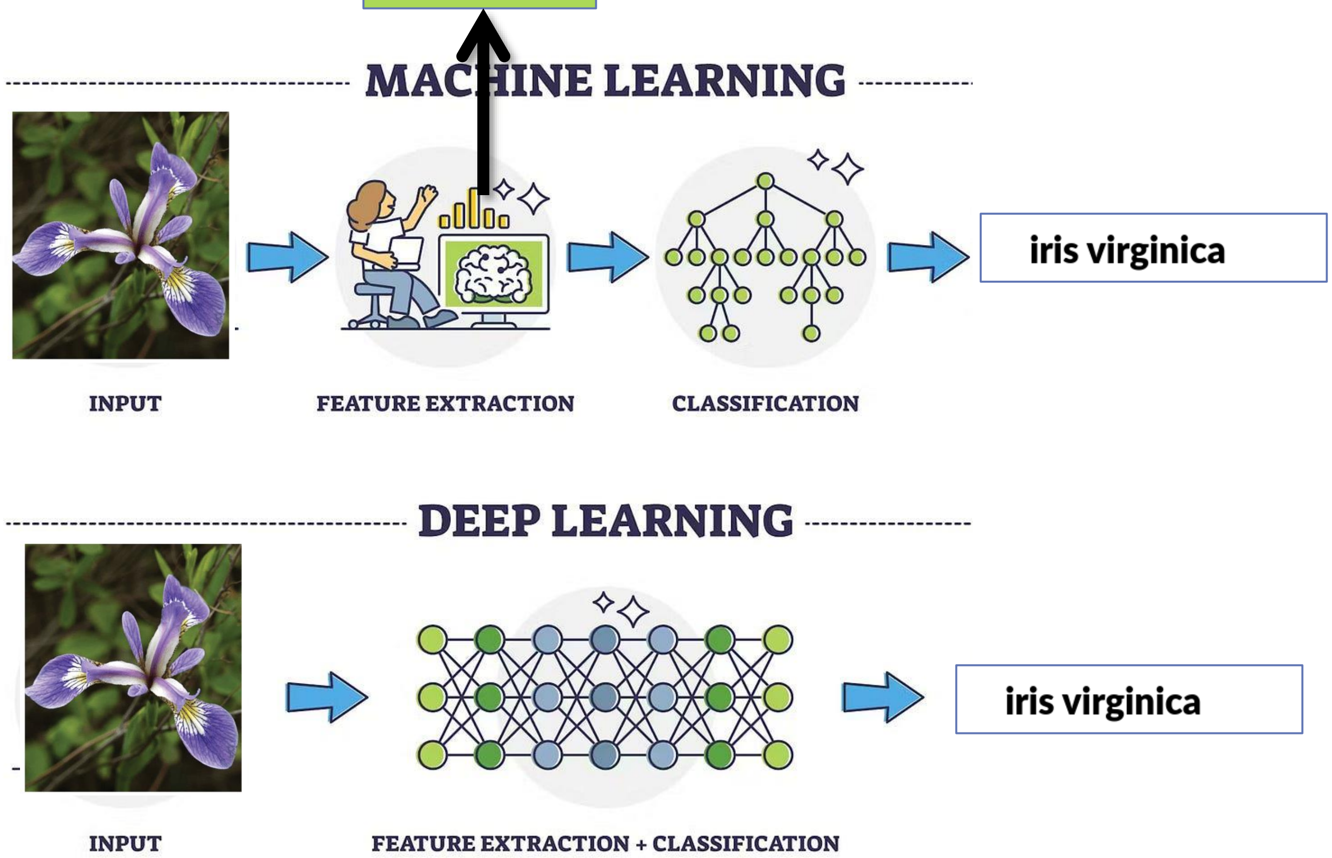
- N=150
- Features: 4 (sepal length, sepal width, petal length, petal width)
- Classes: 3 (Setosa, Versicolor, Virginica)



- Machile Learning / Deep learning models

Manual feature extraction

sepal length,
sepal width,
petal length,
petal width



Dataset: train and test

```
X_train, X_test, y_train, y_test = train_test_split(X, Y,
  test_size=0.2, random_state=10)
```

- Splits the dataset into two parts: **Training set** (80%), **Test set** (20%)

- 0.8*150=120

$$X\_train \ (120,4)$$
$$Y\_train \ (120,1)$$

- 0.2*150=30

$$X\_test \ (30,4)$$
$$Y\_test \ (30,1)$$

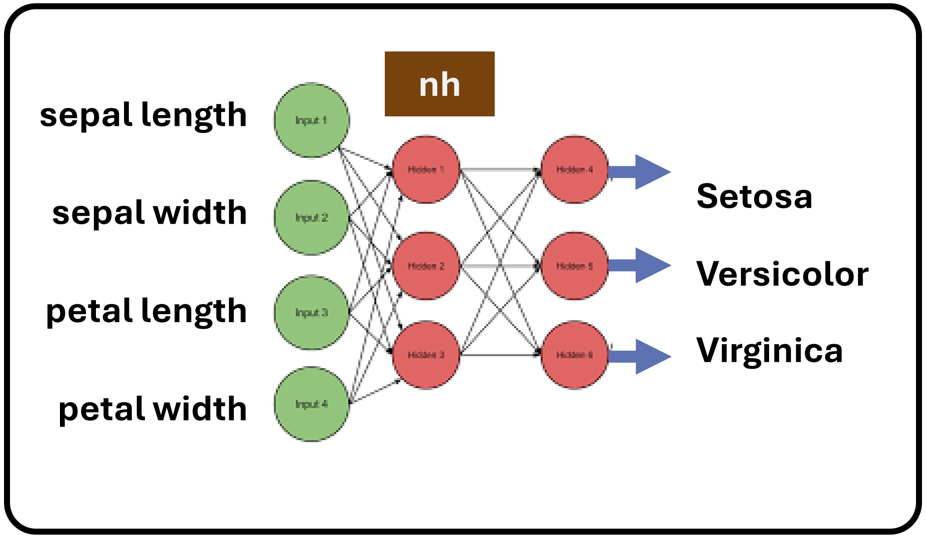
- Classes

```
three = np.zeros((y_train.shape[0],3))    # (120,3)
three[np.where(y_train == 0),0] = 1
three[np.where(y_train == 1),1] = 1
three[np.where(y_train == 2),2] = 1
```

- Output 1D to clsses 3D

T	*three* (120,3)	
[0	[1 0 0	If target=0
…	…	
1	0 1 0	If target=1
…	…	
2]	0 0 1]	If target=2

Neural network structure



- Normalization

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
scaler.transform(X_train);
X_train = np.array(scaler.transform(X_train))
X_test  = np.array(scaler.transform(X_test))
```

- StandardScaler is used for feature scaling, which standardizes features by removing the mean and scaling to unit variance.
- Returns a scaled array where each **feature has mean 0 and variance 1**.

$$X\_scaled = \frac{X - \mu}{\sigma}$$

- Define and train a MLNN

Inputs=4, outputs=3

Nh=30

```
an = MultiLayerPerceptron(4, 3, 30, number_of_iterations=1000,
  output_type='logistic', learning_rate = 0.001)

an.train(X_train, three)
```

```
Y= an.predict (X_test)
test_predictions = np.argmax(Y, axis=1)     # {0,1,2}


print("Confusion Matrix\n", confusion_matrix(y_test, test_predictions))
print("Accuracy: %.2f %%" % (accuracy_score(y_test, test_predictions) * 100))
```

```
Confusion Matrix
 [[10  0  0]
 [ 0 13  0]
 [ 0  0  7]]
Accuracy: 100.00 %
```

Contents

PyTorch

- PyTorch is an open-source machine learning library primarily developed by Facebook's AI Research lab (FAIR).

- Popular among researchers and developers for building and training deep learning models.

- This feature is particularly useful for tasks such as **natural language processing and computer vision**.

PyTorch

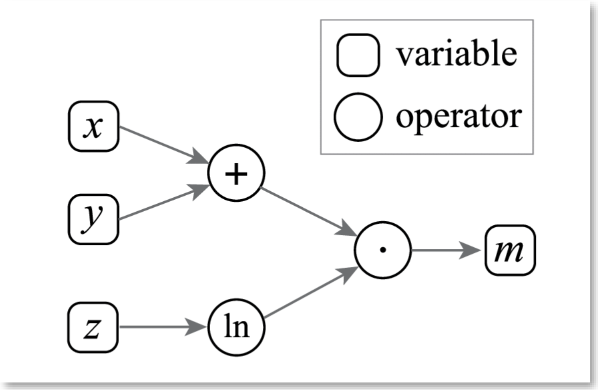
Key features of PyTorch include:

- **Tensors**:
  - Mmulti-dimensional array called a tensor,
  - Similar to NumPy arrays
  - Added benefit of **GPU acceleration** for numerical computations.

- **Neural Network Module**:
  - **torch.nn** module, providing pre-defined layers, loss functions, and optimization algorithms to facilitate the construction of neural network architectures

- **Autograd**:
  - Automatic differentiation library called Autograd, which **automatically computes gradients for tensors during backpropagation**.
  - This simplifies the process of training neural networks.

PyTorch



- A **computational graph** is a structure that represents **all the operations and dependencies** in a neural network.

- PyTorch builds its **computational graph dynamically**,
  - It creates the graph as your code runs.
  - This is called “define-by-run”: you don’t have to define the whole network in advance.
  - **Easy debugging**: You can use normal Python tools like print() to inspect what’s happening.
  - **Flexible models**: You can write loops, conditionals, or work with inputs of different sizes.

1. Creating Tensors

```
import torch


x = torch.tensor([1.0, 2.0, 3.0])
x = torch.zeros(2, 3)

x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)*


# Convert numpy to a PyTorch tensor
np_array = np.array([[1, 2], [3, 4]])
tensor = torch.from_numpy(np_array)


* requires_grad=True → PyTorch will compute gradients for this tensor.
```

2. Basic Tensor Operations

```
a = torch.tensor([1.0, 2.0])
b = torch.tensor([3.0, 4.0])
c = a + b
d = a * b

#------------------------------------ Mean, sum
mean_val = a.mean()
sum_val = b.sum()

#------------------------------------ # Matrix multiplication
x = torch.rand(2, 3)
y = torch.rand(3, 2)
z = torch.matmul(x, y)
```

3. Using GPU (CUDA)

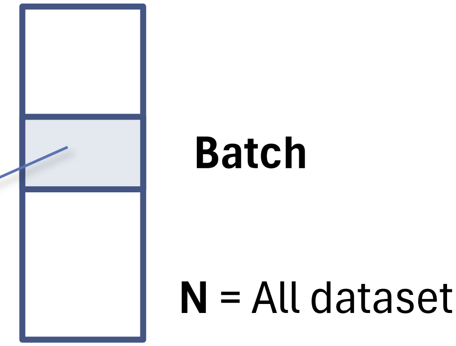
```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = torch.device("cpu")

#----------------- Move tensors to device
x = x.to(device)

#----------------  Move a model to device
model.to(device)
```

CUDA Compute Unified Device Architecture.
Parallel computing platform and API created by NVIDIA that lets software (PyTorch, TensorFlow, etc.) use the GPU (Graphics Processing Unit) to do computations

4. Neural networks: definitions

```
import torch
import torch.nn as nn                           # Neural network modules
import torch.optim as optim                     # Optimizers


#----------------- Loss functions

criterion = nn.MSELoss()                         # For regression

criterion = nn.CrossEntropyLoss()                # For classification


#----------------- Optimizers

optimizer = optim.SGD(model.parameters(), lr=0.01)

optimizer = optim.Adam(model.parameters(), lr=0.001)
```

5. Neural networks: Train Loop




```
Num_epochs=100

for epoch in range(num_epochs):                     # iterates over all epochs
    for X_batch, y_batch in train_loader:           # Over the dataset in mini-batches
                                                      using a Data Loader

            X_batch = X_batch.to(device)            # To device=CPU/GPU
            y_batch = y_batch.to(device)

            outputs = model(X_batch)                # Step 1: Forward pass
            loss = criterion(outputs, y_batch)      # Step 2: Compute loss

            optimizer.zero_grad()                   # Step 3: Clear old gradients
            loss.backward()                         # Step 4: Backpropagation
            optimizer.step()                        # Step 5: Update weights
```

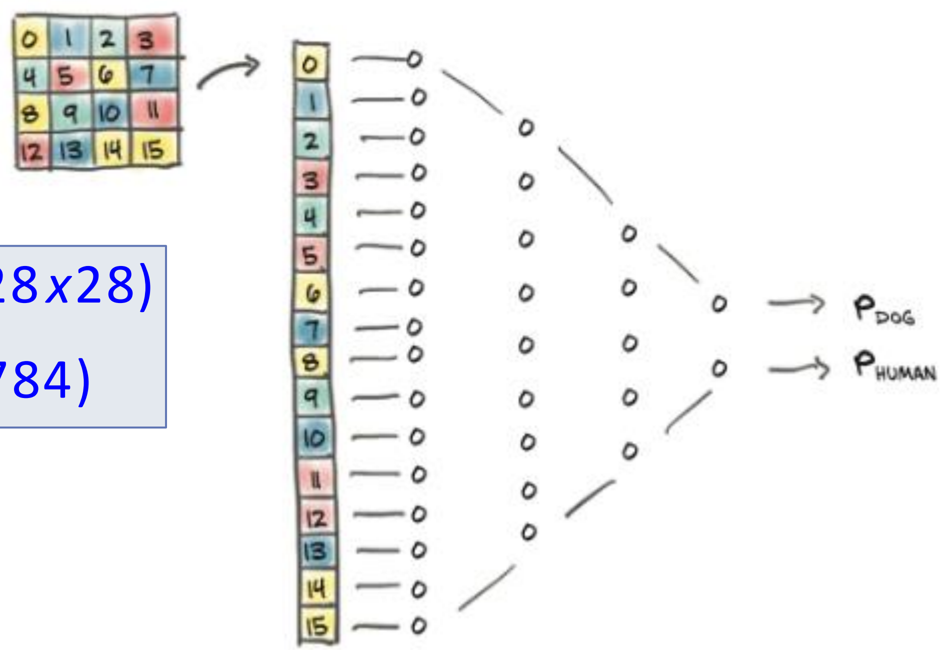
Contents

- Load Data –> numpy

```
#--------------------------------------- LOAD Train/test data
#--------------------------------------- as a tensor
#  Train = 60000, 28x28,   Test = 10000 28x28

mnist_trainset =
datasets.MNIST(root='./data', train=True, download=True, transform=None)
mnist_testset =
datasets.MNIST(root='./data', train=False, download=True, transform=None)


#--------------------------------------- Tensor to numpy
X_train = mnist_trainset.data.numpy()
y_train = mnist_trainset.targets.numpy()
X_test = mnist_testset.data.numpy()
y_test = mnist_testset.targets.numpy()
```

Binary images (black and white) & Flattening

```
#----------------------------------------- Image [0 .. 255]
#----------------------------------------- Convert to [0..1]
X_train = np.where(X_train>0.5,1,0)
X_test = np.where(X_test>0.5,1,0)

# Flatten the input (i.e., convert 2D images to 1D vectors)
X_train = X_train.reshape(X_train.shape[0], -1)
X_test  = X_test.reshape( X_test.shape[0], -1)
```



$$X\_train \quad (60\,000, 28x28)$$

$$X\_train \quad (60\,000, 784)$$

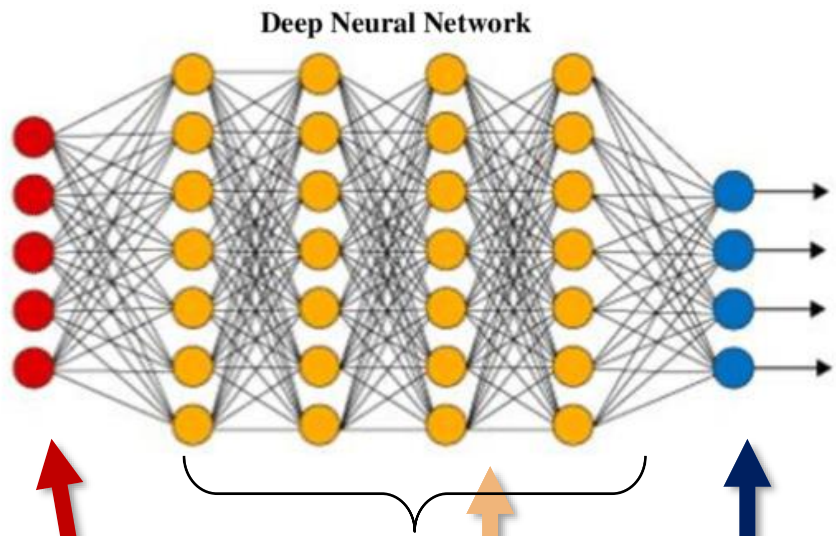
- Data to tensor &
- Number of inputs and outputs=classes

```
#---------------------------------------- Image [0 .. 255]
X_train = torch.tensor(X_train).float()
y_train = torch.tensor(y_train).long()
X_test  = torch.tensor( X_test).float()
y_test  = torch.tensor( y_test).long()


num_inputs = X_train.shape[1]                    # 784
num_classes = int(y_train.max().item() + 1)      # 10 classes
```

- Device

```
if torch.backends.mps.is_available():     # **
        device = torch.device("mps")

elif torch.cuda.is_available():
        device = torch.device("cuda")
else:
        device = torch.device("cpu")


# MPS --  Apple's Metal Performance Shaders
```

- MLNN - Deep CNN




```
class DNN(nn.Module):
       def __init__(self, input_size, hidden_sizes, num_classes):
       self.layers = nn.ModuleList()
       self.activations = nn.ModuleList()

# --------------------  Example
NumInp= 5
NHidden=[7,7,7,7]  # same size !!!
NumCla= 4

dnn = DNN(input_size=numInp, hidden_sizes=NHidden, num_classes=numCla)
```

- TODO

```
self.layers = nn.ModuleList()           # from torch
self.activations = nn.ModuleList()

self.layers      is a list of linear layers, e.g. nn.Linear
self.activations is a list of activation functions, e.g. nn.ReLU()
```

```
def forward(self, x):
#----- TODO: implement the forward pass
  out = x
  for i in range(len(self.layers)):               # All layers
        out = self.layers[i](out)                 # out = W out
        if i < len(self.activations):
                 out = self.activations[i](out)   # out = activation(out)

return out
```

- Train the MLNN

```
def fit(X_train, y_train, nn, criterion, optimizer, n_epochs,
  to_device=True, batch_size=32):
….
return loss_values, nn

# X_train         Input training data
# y_train         Target training data
# nn              DNN – MLNN
# criterion       Loss functions
# optimizer       Backpropagtion optimizer
# n_epochs        Number of epochs
# batch_size      Dimension of each batch for training
```

Application to MNIST

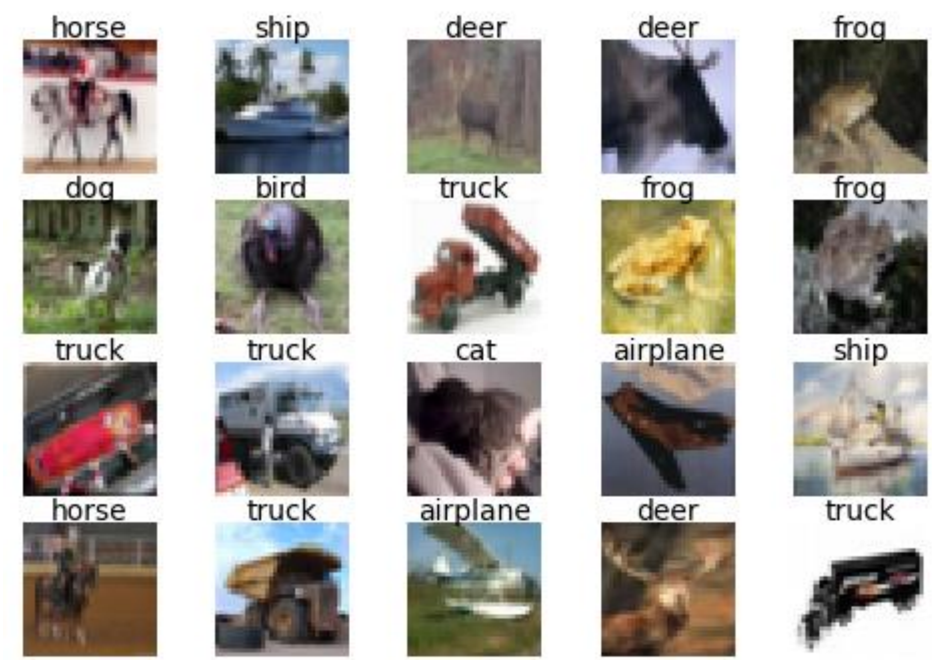
$$nh = \frac{Num\,inputs + Num\,Classes}{2}$$

$$nh = \frac{784 + 10}{2} = 397$$

```
num_inputs = 28*28
num_clases = 10
n_epochs = 100
n_layers = 3
BATCH_SIZE = 100

# MUST HAVE THE SAME SIZE !!
# Determine the number of hidden_layer_sizes = [397,397,397,397]
hidden_layer_sizes = ((num_inputs + num_classes) // 2,)*n_layers

#------------------------ Defnne the MLNN
dnn =
DNN(input_size=num_inputs, hidden_sizes=hidden_layer_sizes, num_classes=num_classes)

#------------------------ Definitions
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(dnn.parameters(), lr=0.1)

#------------------------ Train MLNN
loss_values, dnn =
 fit(X_train, y_train, dnn, criterion, optimizer, n_epochs, batch_size=BATCH_SIZE,
  to_device=False)
```

Contents

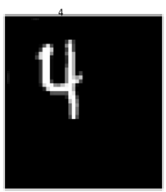
- CIFAR-10
  - Simillar to MNIST,
  - Classification problem with 10 possible classes.
  - Images are more complex, representing real life scenes instead of hand-written digits.



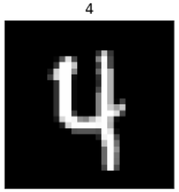
0 - airplane
1 - automobile
2 - bird
3 - cat
4 - deer
5 - dog
6 - frog
7 - horse
8 - ship
9 - truck

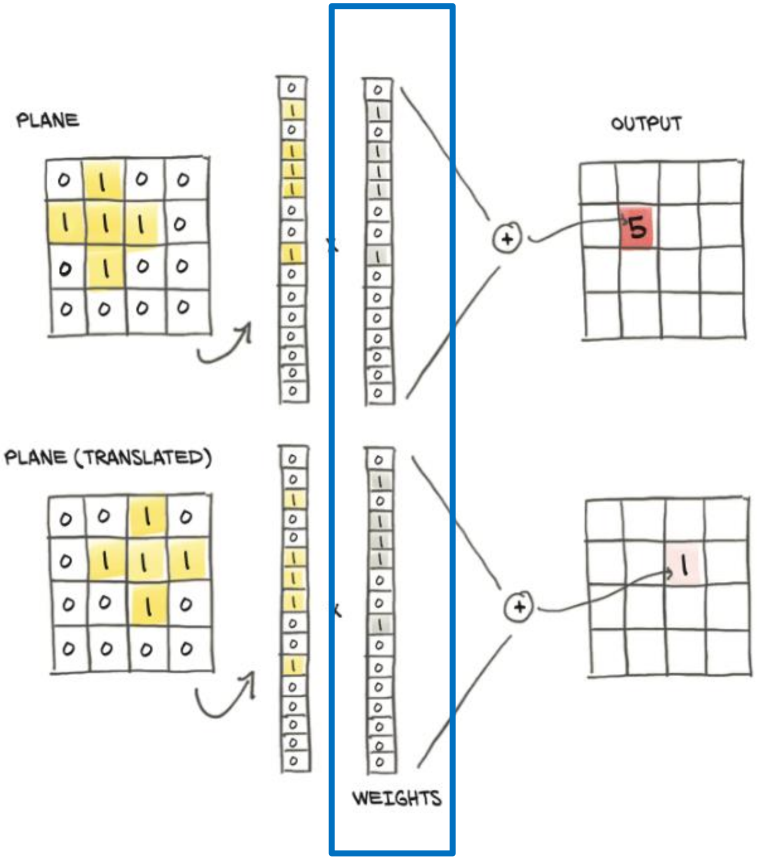
- MLNN
  - Main problem !!
  - The use of pixel value values make the model **sensible to translations**

Same filter



Same pattern



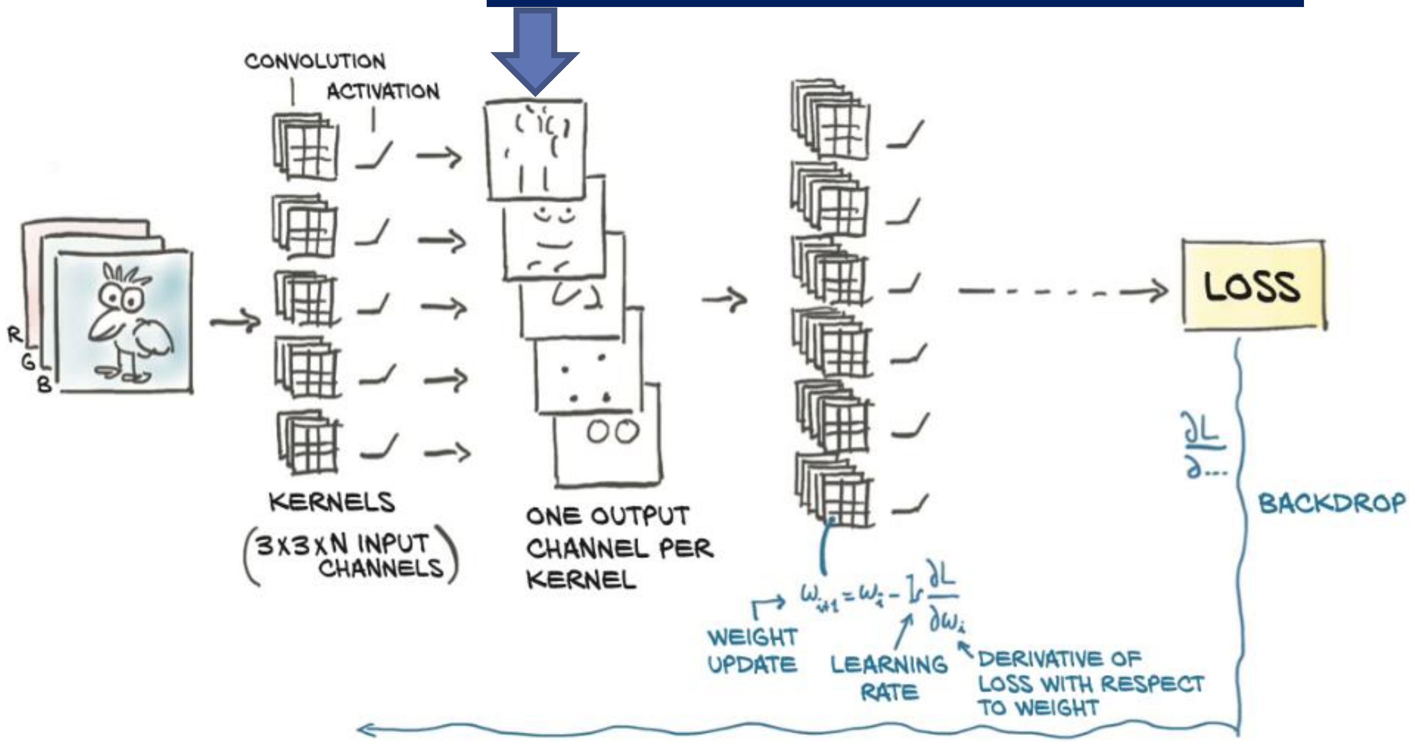


Different output !

Contents

- **CNN – Convolutional neural networks**
  - CNN uses this operations to **automatically compute features** on an input image producing feature maps.
  - Translation-invariant characteristics, no matter the variations in position, orientation, scale, or translation
  - **Filters (weights) are adjusted to minimize a loss function**

Num filters(kernels) = num of features



Buld a CNN

RBG

$32x32$

```
class CNN(nn.Module):
 def __init__(self, input_channels=3, num_classes=10):

  #------------------------------- Two Convolutional layers
  self.conv1 =
  nn.Conv2d(input_channels, 16, kernel_size=3, padding=1)

  self.conv2 =
  nn.Conv2d(16, 8, kernel_size=3, padding = 1)

  #------------------------------- Two fully connect layers
  self.fc1 = nn.Linear(8*8*8, 32)
  # 8x8x8=512 (height × width × channels) the tensor after the last convolution

  self.fc2 = nn.Linear(32, num_classes)
```

16 filters

3x3

$32 \rightarrow pool(2x2) = 16$

3x3

8 filters

$16 \rightarrow pool(2x2) = 8$

```
myCNN=CNN(3,10)
```

Train a CNN

```
cnn_loss_values, cnn =
fit(X_train, y_train, cnn, criterion, optimizer, n_epochs, to_device=True,
  batch_size=BATCH_SIZE)
```