Introduction to Artificial Neural Networks

Hao Ji, Data Scientist

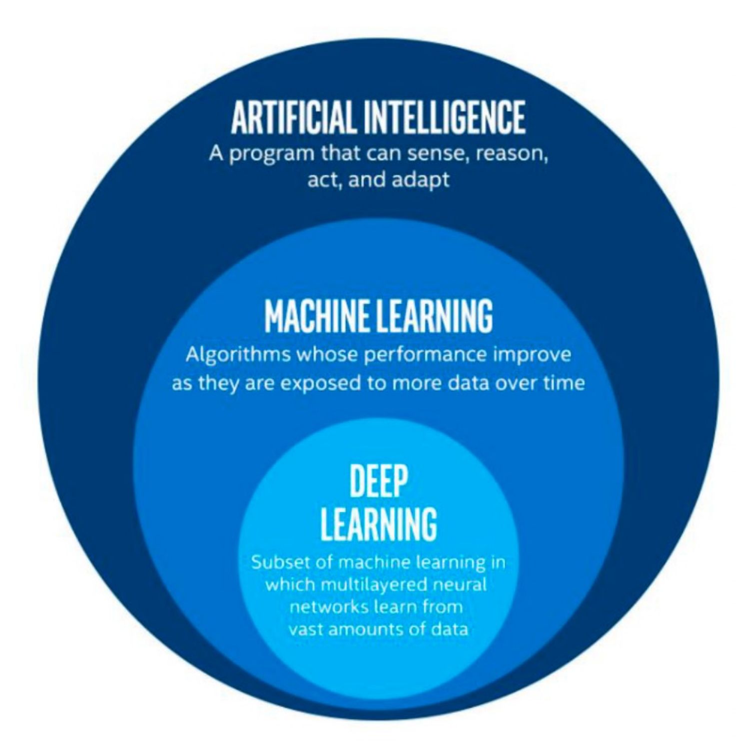
USC Center for Advanced Research Computing

Content

Introduction to Deep Learning

Deep Learning: subfield of traditional machine learning

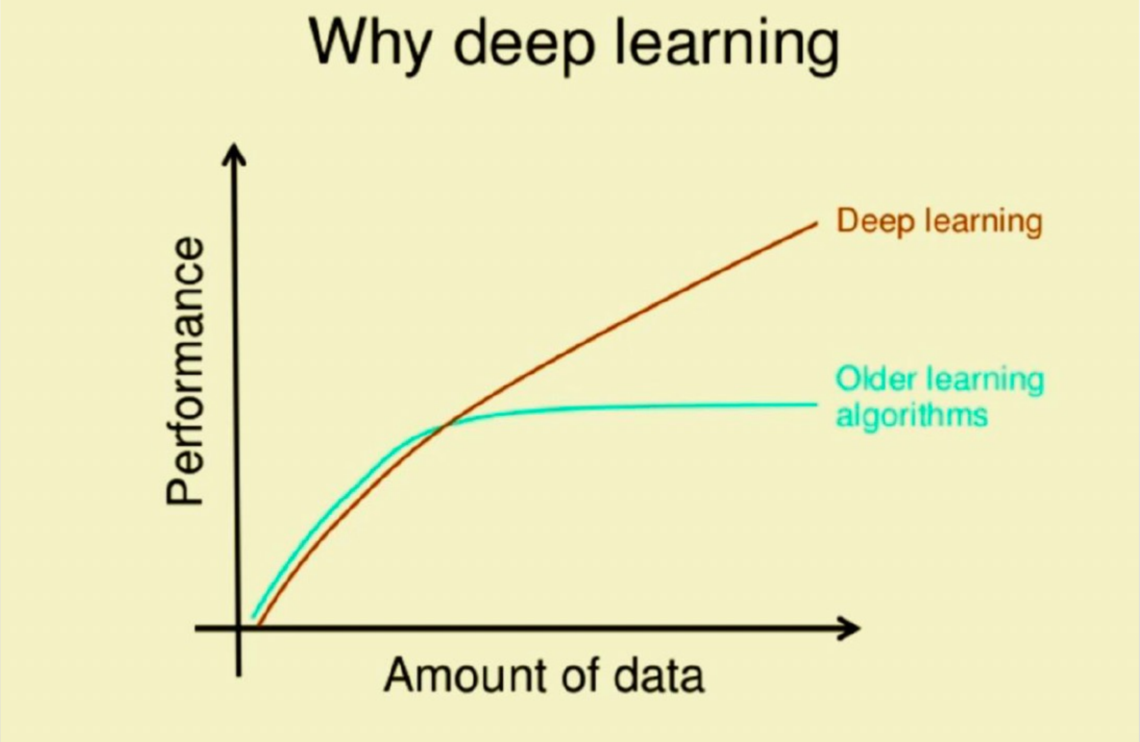
- Inspired by the structure and function of the brain: Artificial Neural Networks
- Computer vision: Tesla recognizing items on a street
- Text Generation: An algorithm trained to create a new Shakespeare piece
- Speech recognition
- Computer Games: AlphaGo



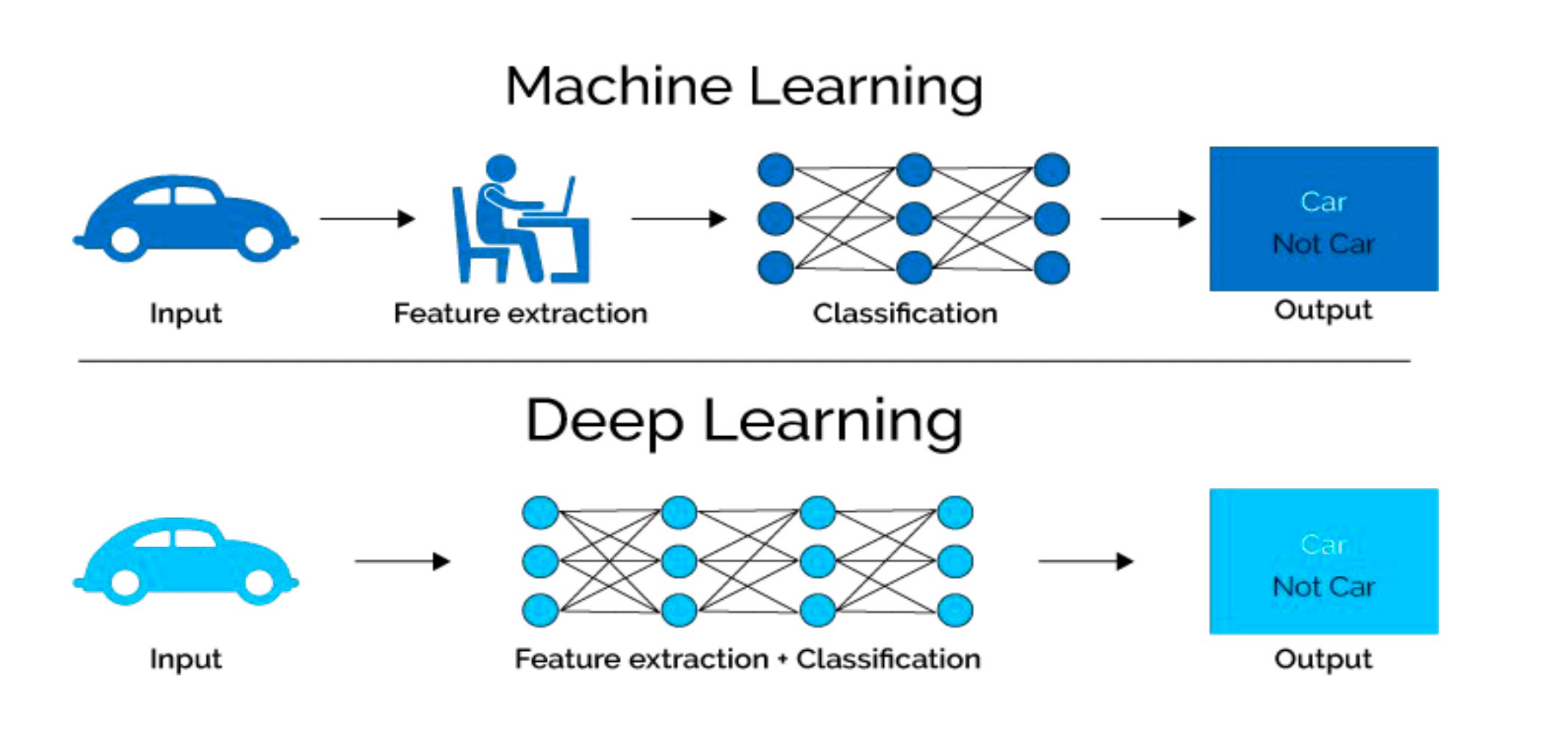
Introduction to Deep Learning

What drives the recent development of Deep Learning?

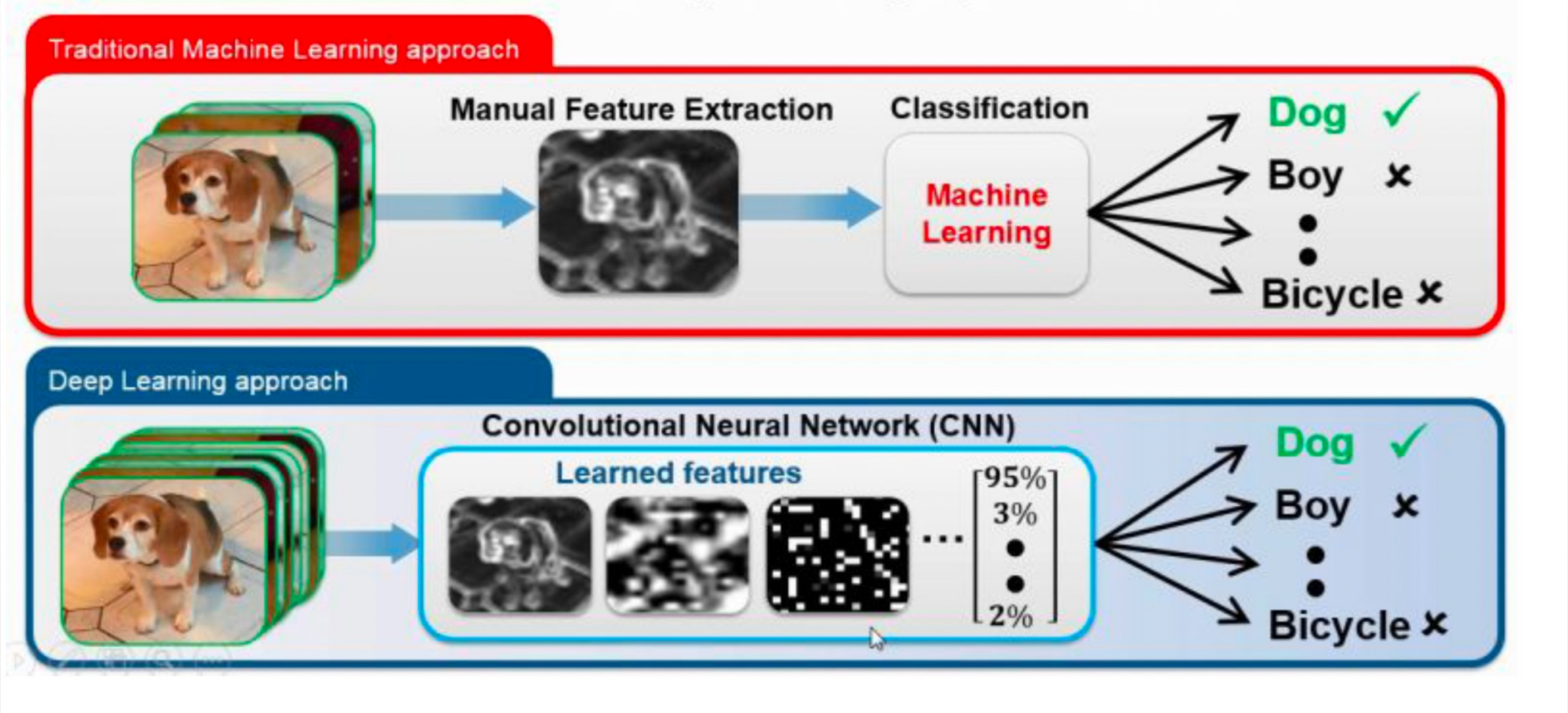
- Larger amounts of data available
- Data Storage
- Strong computation units such as GPU's



Introduction to Deep Learning

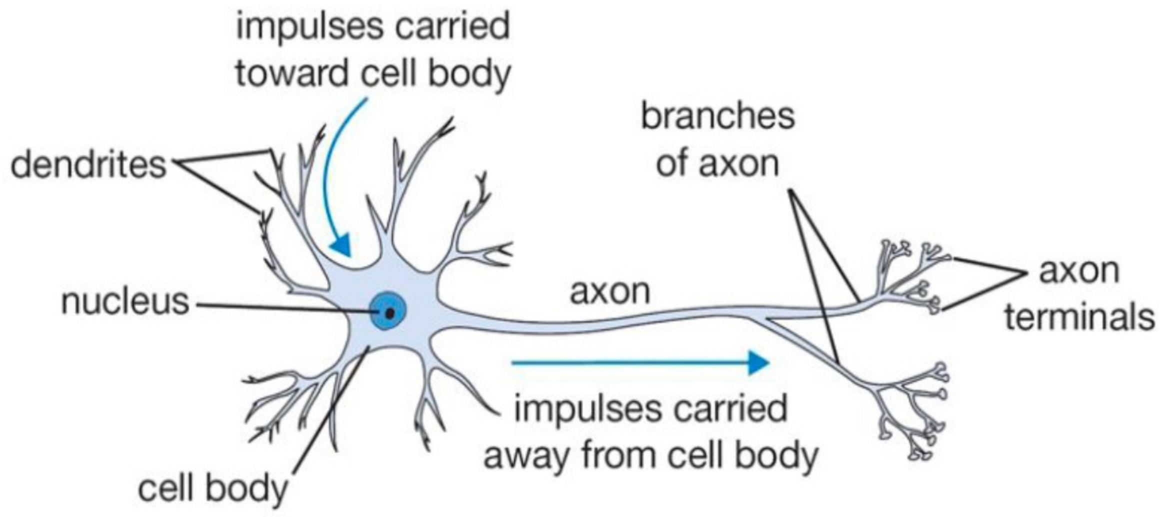


Introduction to Deep Learning

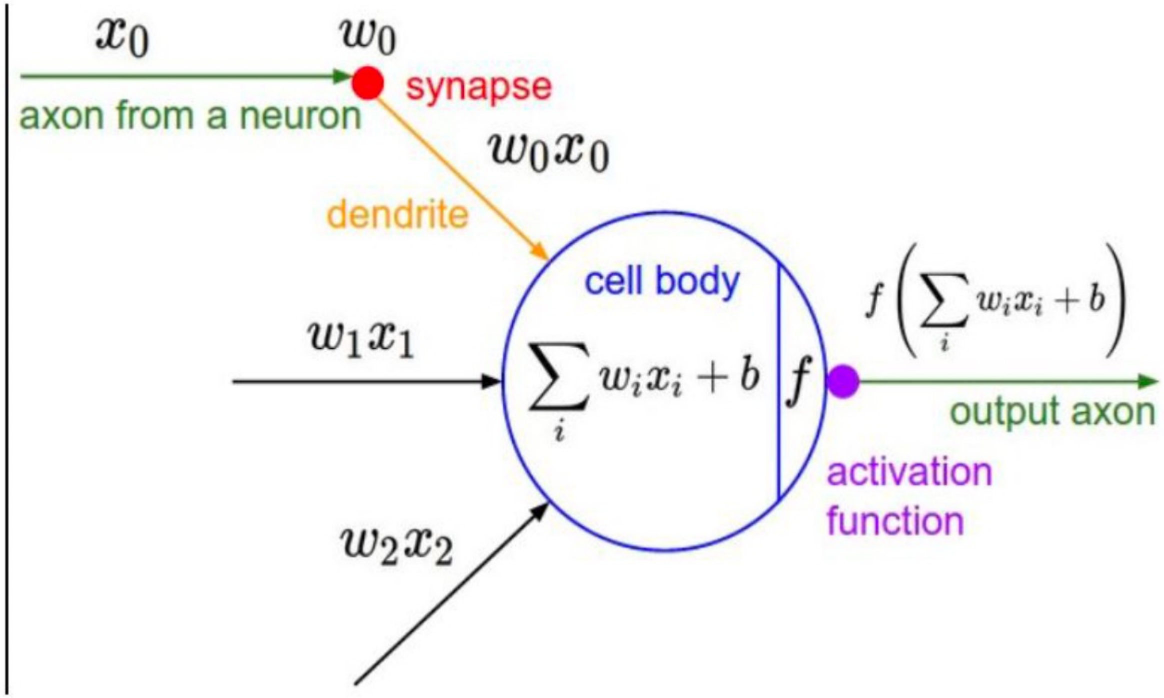


Deep Learning is a machine learning technique that can learn useful representations or features directly from images, text and sound

Neural Networks

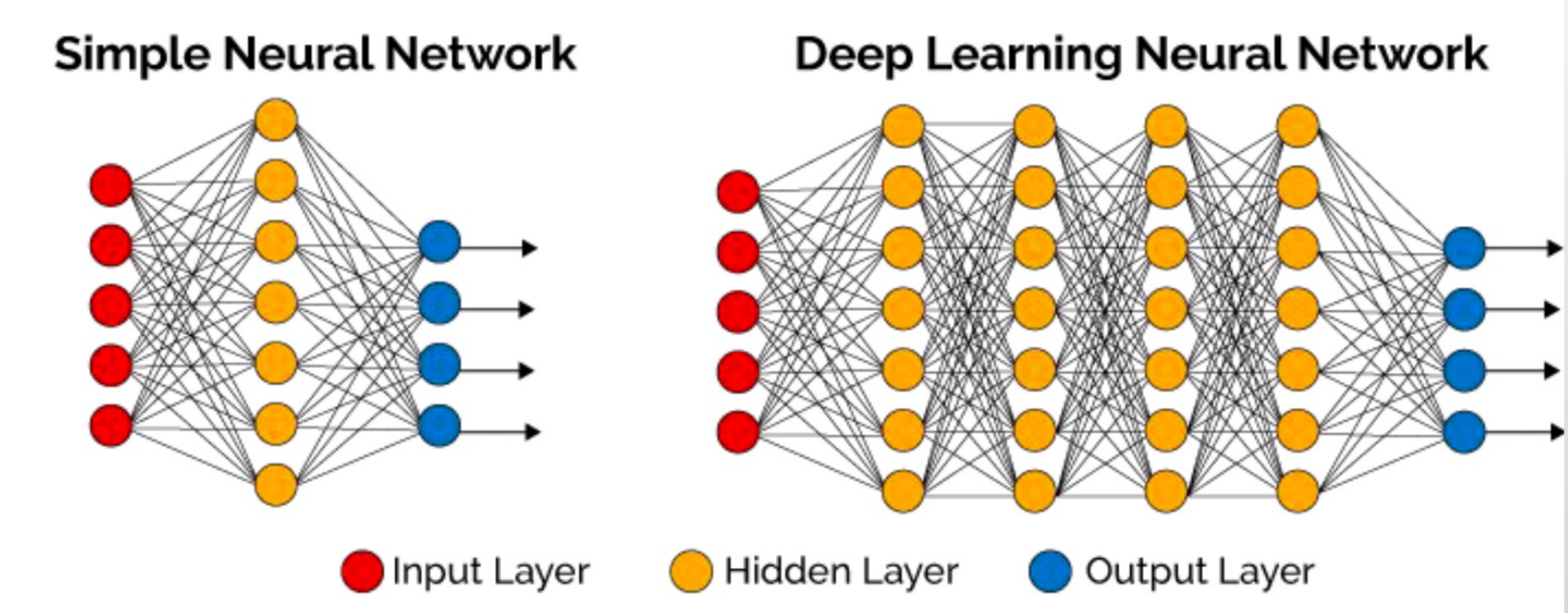


biological neuron



artificial neural networks

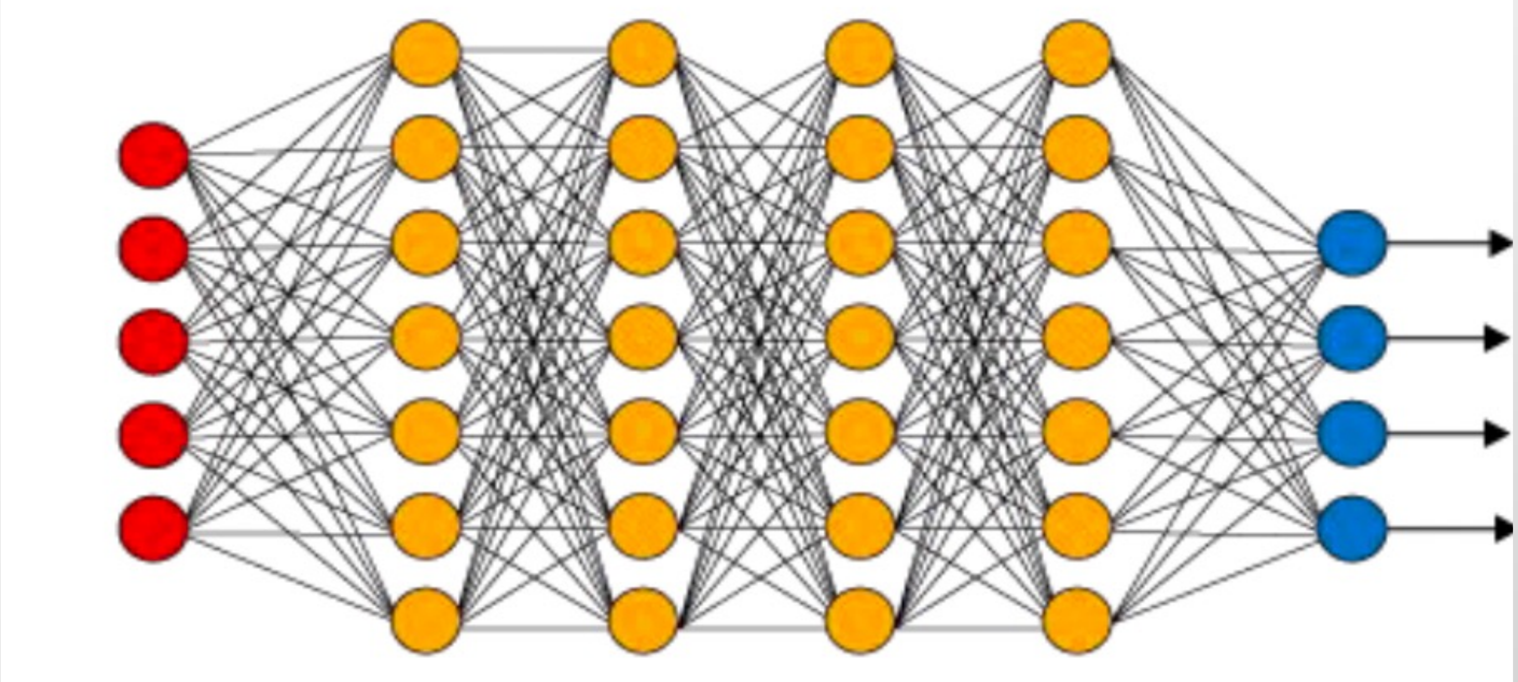
Neural Networks



A neural network (NN) has 3 types of layers:
Input layer Hidden layer Output layer

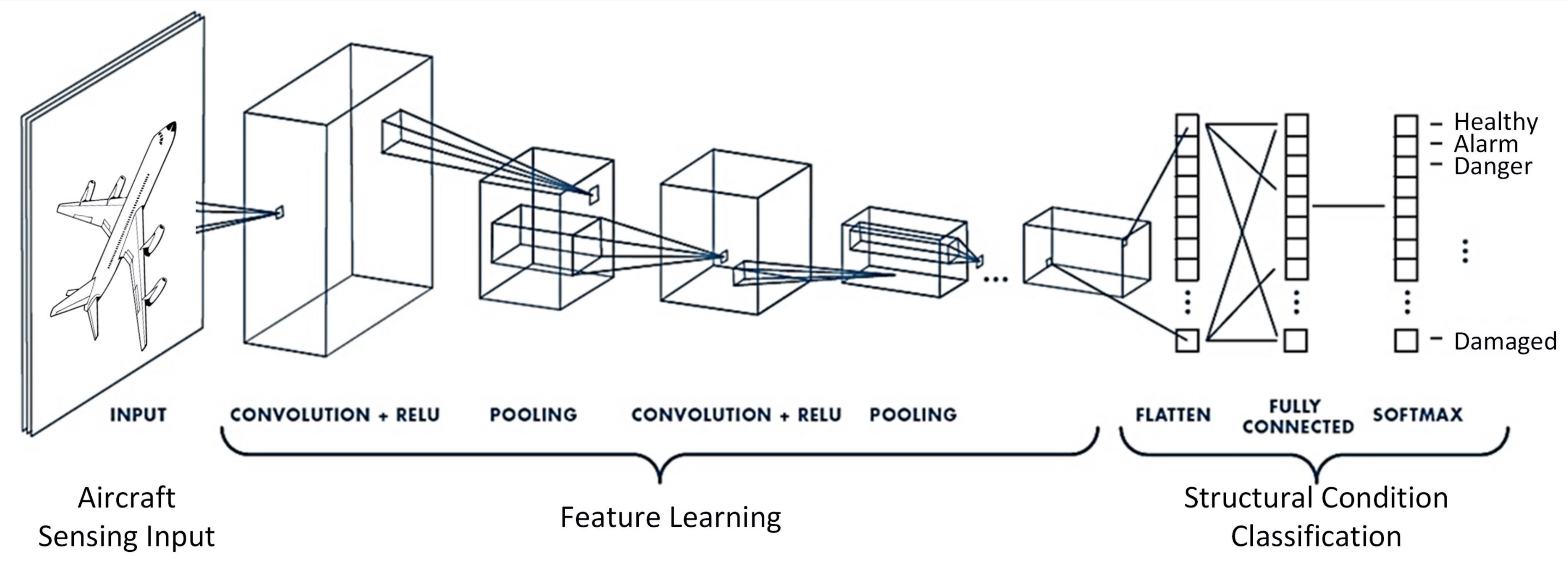
Deep neural networks (DNN) usually has more hidden layers
Still has same 3 types of layers

Different Types of Neural Networks



Multi-layer Perceptron

Different Types of Neural Networks



CNN: Convolutional Neural Network

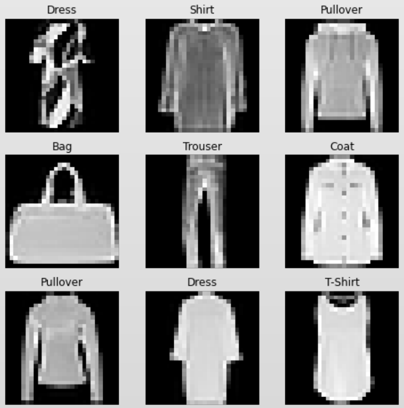
Building Neural Networks

Task: Predict if an input image belongs to one of the following classes: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, or Angle boot.

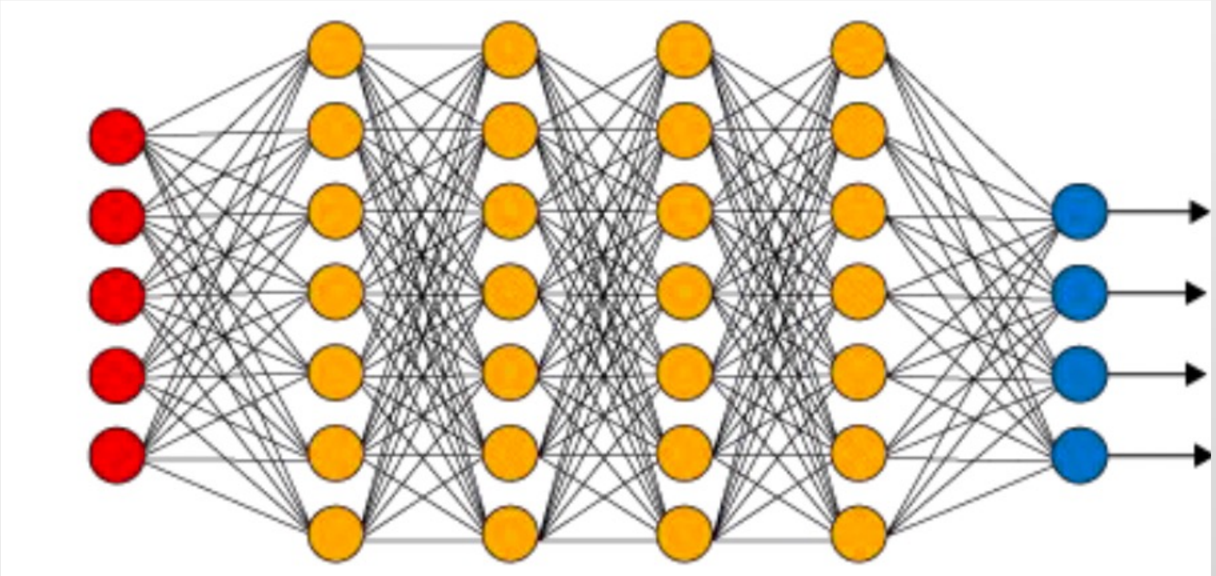
FasionMNIST Dataset

Fashion-MNIST is a dataset comprising of 28×28 grayscale images of 70,000 fashion products from 10 categories, with 7,000 images per category. The training set has 60,000 images and the test set has 10,000 images.

Building Neural Networks

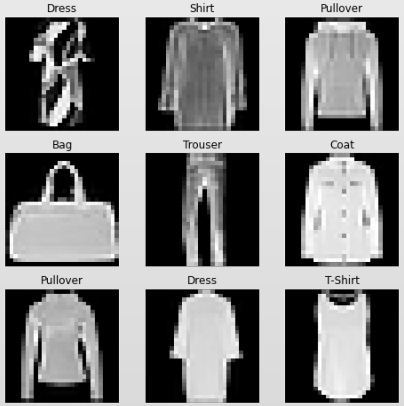


Input X: (28 * 28)

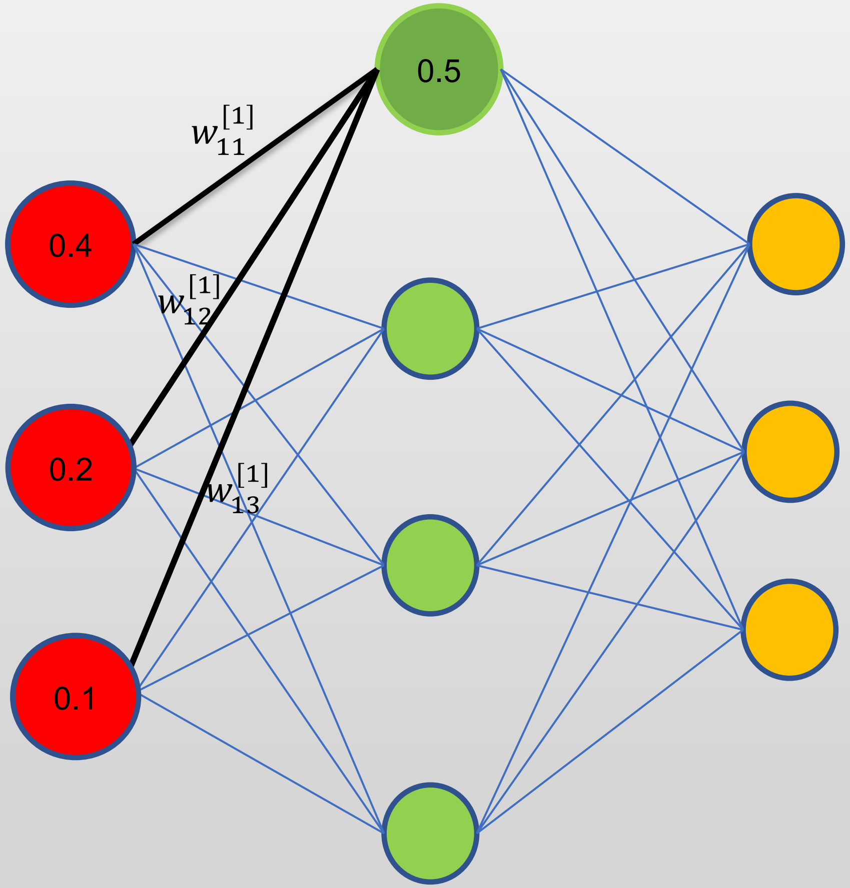


Make Predictions based on Logits

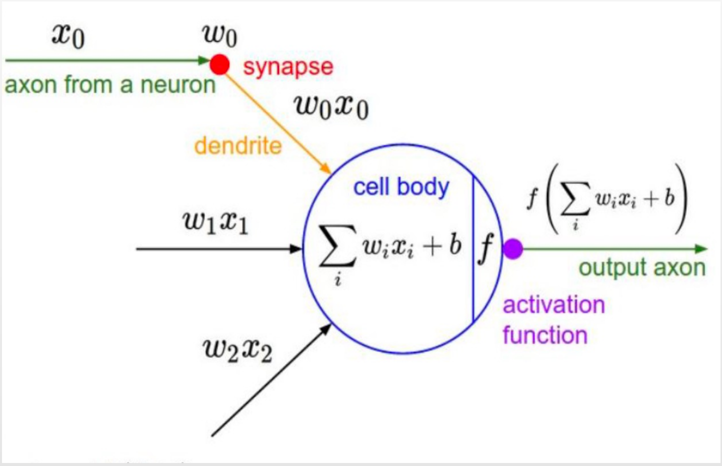
Building Neural Networks



28 * 28 = 784



A_1



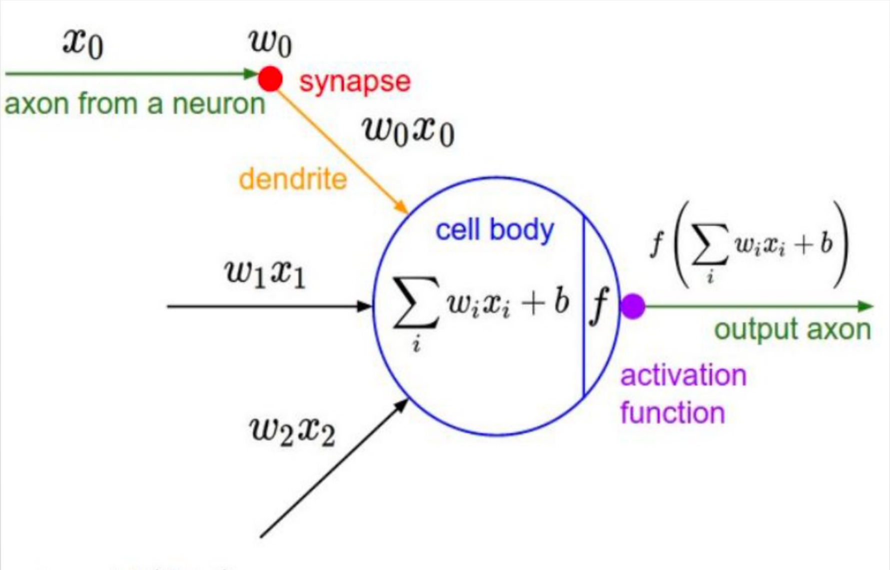
$$Z_1^{[1]} = w_{11}^{[1]}x_1 + w_{12}^{[1]}x_2 + w_{13}^{[1]}x_3 + b_1^{[1]}$$

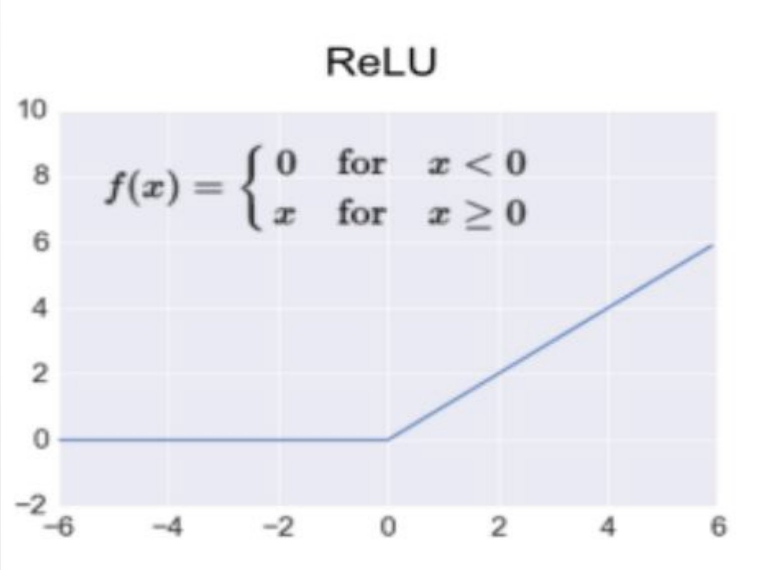
$$= 0.5 * 0.4 + 0.1 * 0.2 + 0.8 * 0.1 + 0.2 = 0.5$$

$$a_1^{[1]} = f(0.5) = ReLU(0.5) = 0.5$$

Weights are initialized randomly

Building Neural Networks



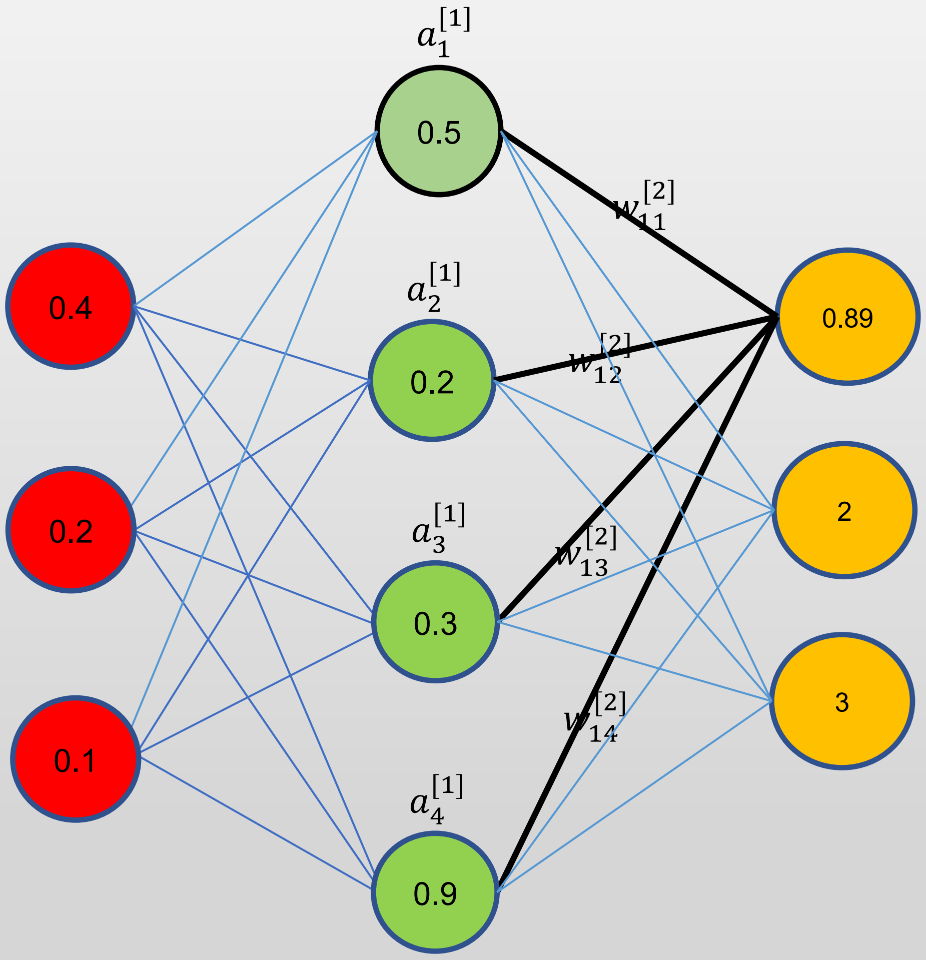


$$Z_1^{[1]} = w_{11}^{[1]}x_1 + w_{12}^{[1]}x_2 + w_{13}^{[1]}x_3 + b_1^{[1]}$$

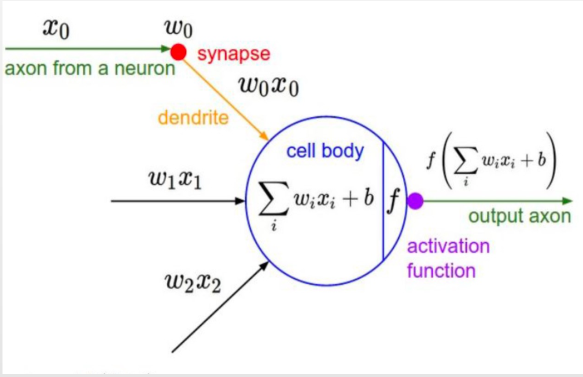
$$= 0.5 * 0.4 + 0.1 * 0.2 + 0.8 * 0.1 + 0.2 = 0.5$$

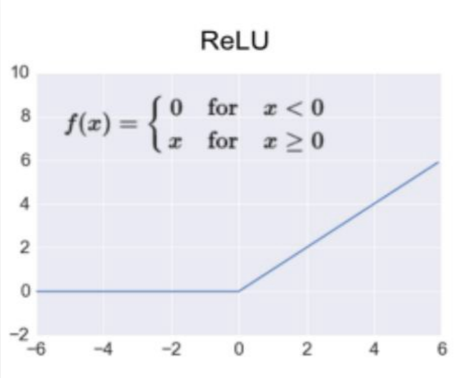
$$a_1^{[1]} = f(0.5) = ReLU(0.5) = 0.5$$

Building Neural Networks



A_1





$$Z_1^{[2]} = w_{11}^{[2]}a_1^{[1]} + w_{12}^{[2]}a_2^{[1]} + w_{13}^{[2]}a_3^{[1]} + w_{14}^{[2]}a_4^{[1]} + b_1^{[2]}$$

$$= 0.3 * 0.5 + 0.1 * 0.2 + 0.2 * 0.3 + 0.4 * 0.9 + 0.3$$

$$= 0.89$$

$$a_1^{[2]} = f\left(Z_1^{[2]}\right) = f(0.89) = ReLU(0.89) = 0.89$$

Neural Networks

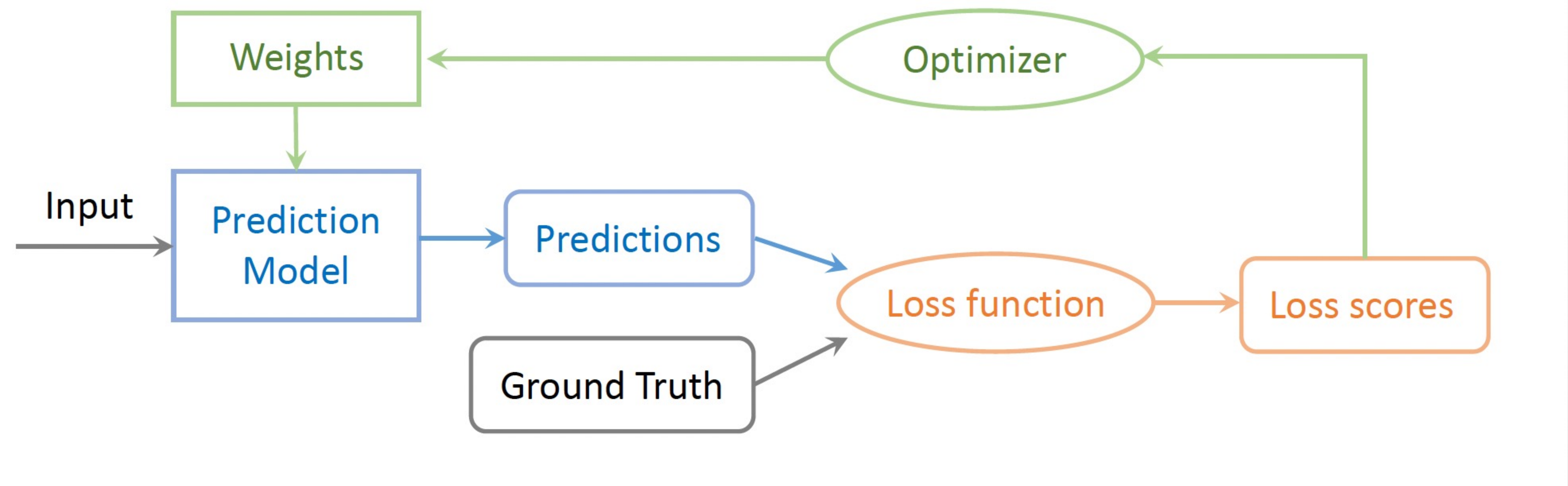
Three steps to training a neural network

1 Forward propagation: push example through the network to get a predicted output

2 Compute the cost: calculate the difference between predicted output and actual data

3 Backward propagation: push back the derivative of the error and apply to each weight, such that next time it will result in a lower error

Training Pipeline



The training pipeline consists of choosing the prediction model, the loss function and the optimizer.

Once these choices are made, we can feed the input data and labels to start the training process.

Deep Learning Applications

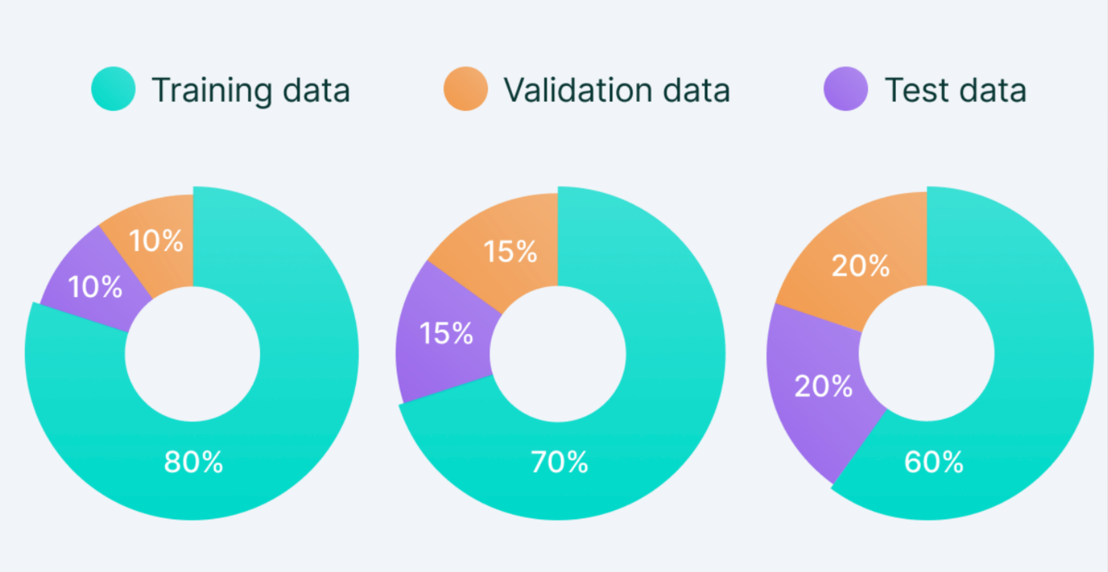
[Deep Learning Applications]

Model Evaluation: Dataset

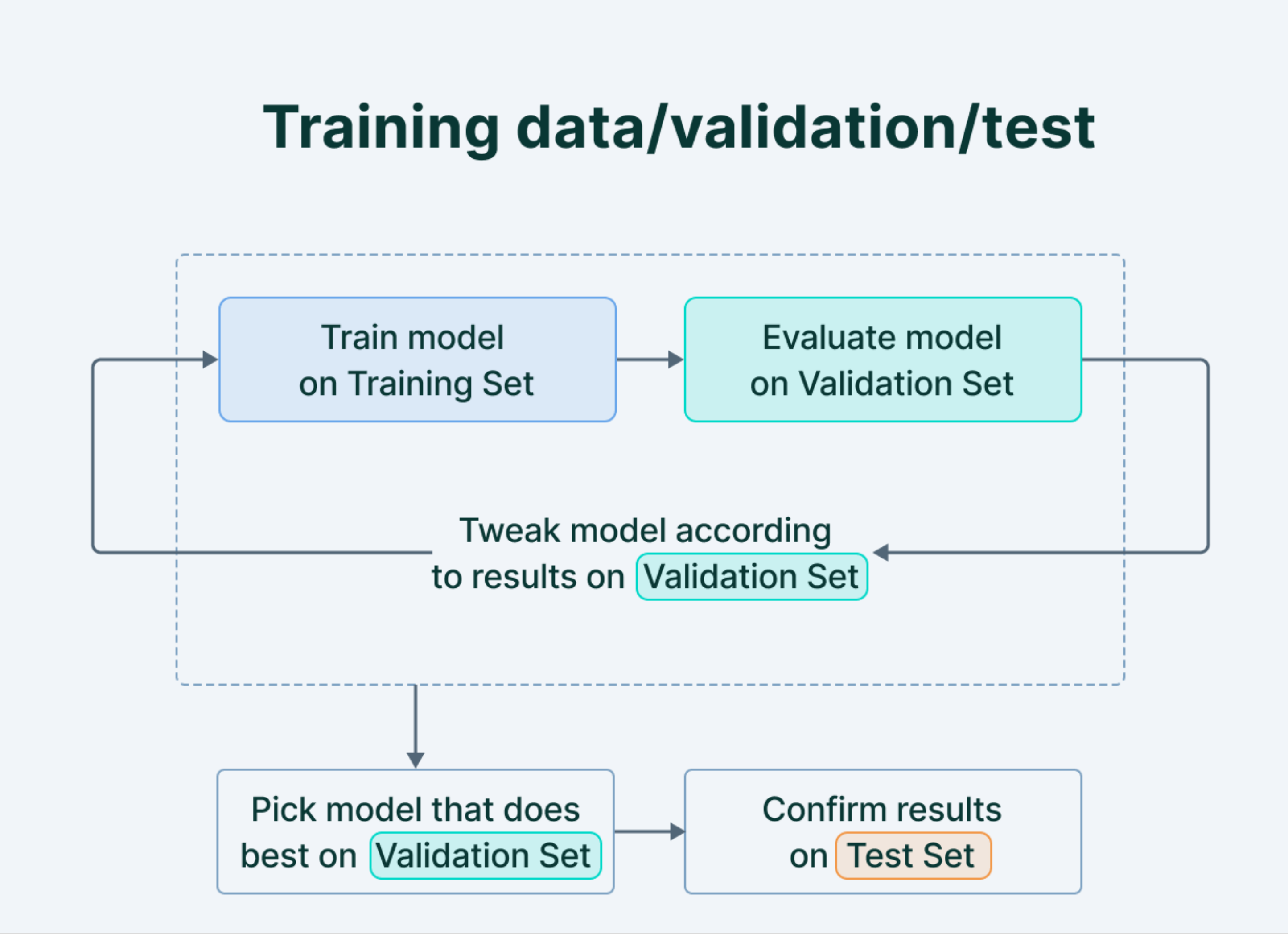
Training Dataset: *The sample of data used to fit the model.*

Validation Dataset*: The sample of data used to provide an evaluation of a model fit on the training dataset while tuning model hyperparameters.*

Test Dataset*: The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.*

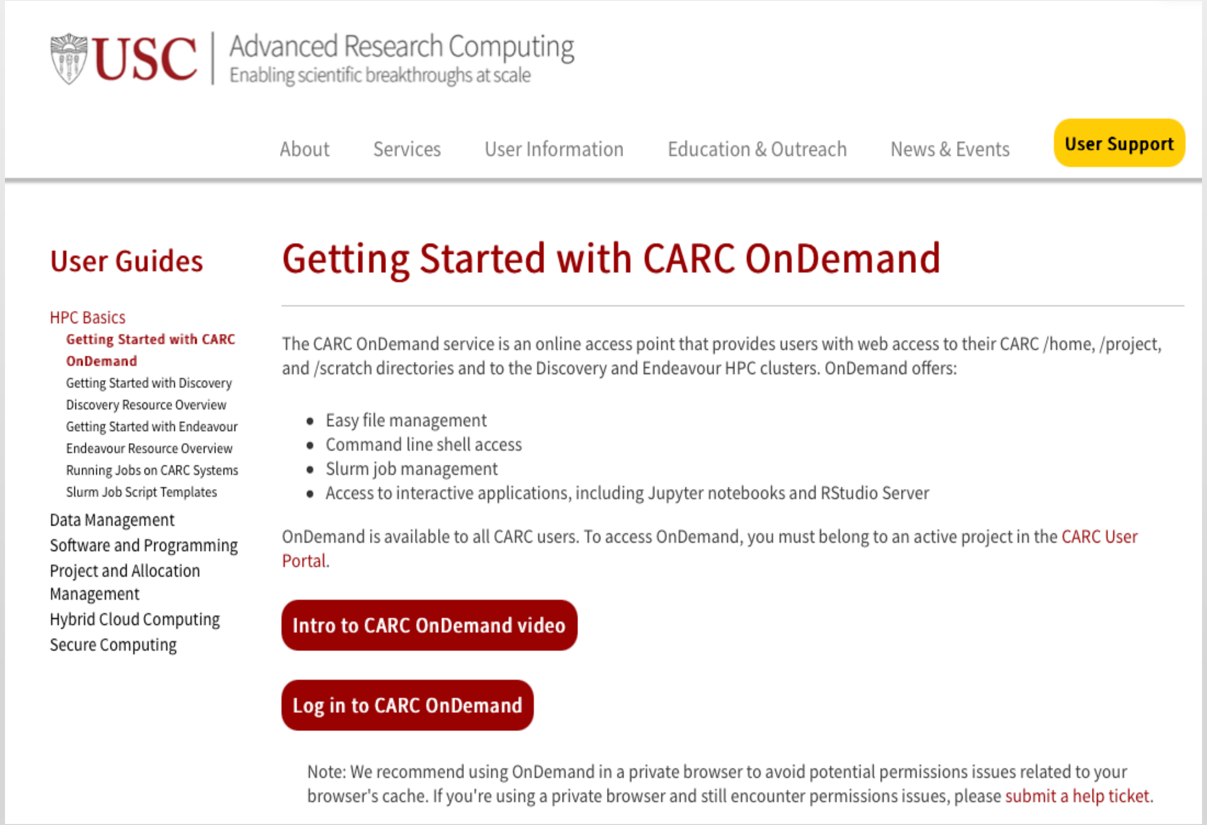


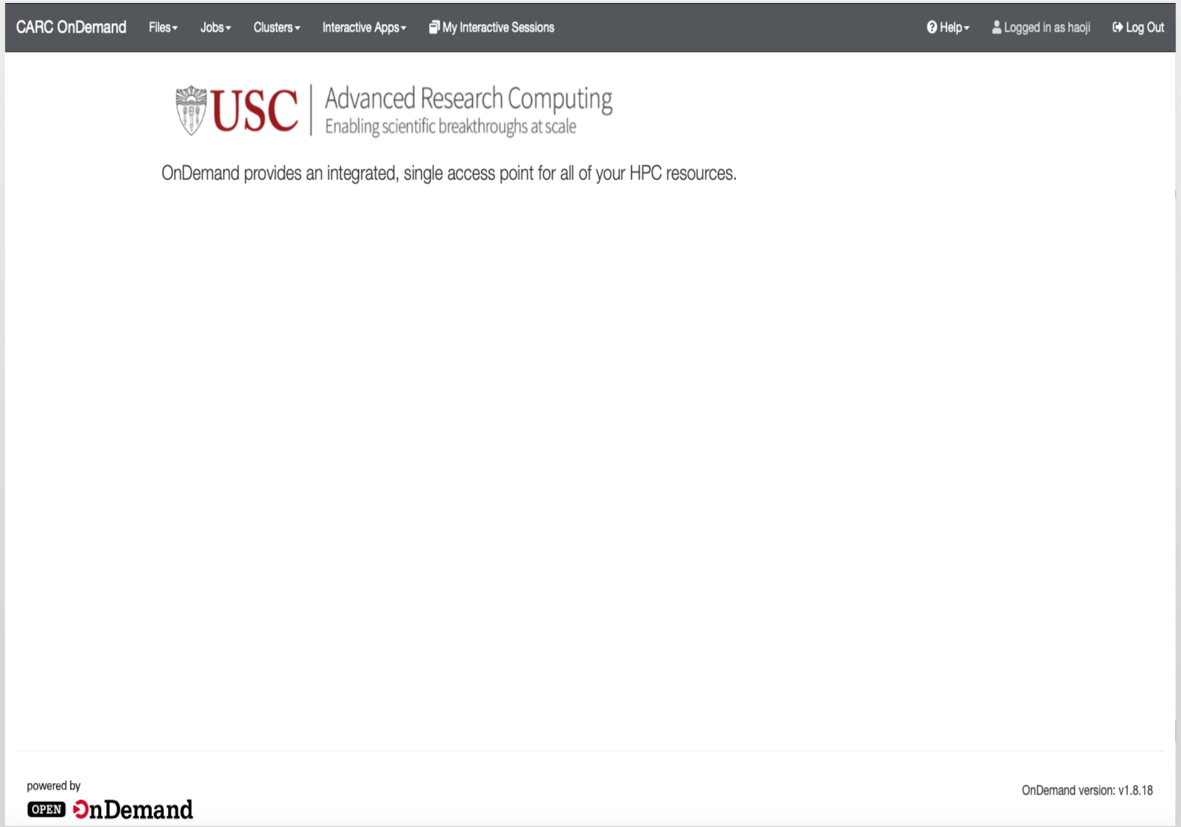
Dataset


Training data/validation/test
Train model on Training Set
Evaluate model on Validation Set
Tweak model according to results on Validation Set
Pick model that does best on Validation Set
Confirm results on Test Set

CARC OnDemand

Web Address: https://ondemand.carc.usc.edu





Using Conda on CARC

Within 'Discovery Cluster Shell Access', Request an interactive session first using salloc

```
salloc --partition=gpu --nodes=1 --ntasks=1 --cpus-per-task=8 --time=1:00:00 --mem=32GB --gres=gpu:1
```

Using Conda on CARC

Anaconda: package and environment manager primarily used for open-source data science packages for the Python and R programming languages.

Using Conda on CARC systems

Begin by logging in. You can find instructions for this in the Getting Started with Discovery or Getting Started with Endeavour user guides.

To use Conda, first load the corresponding module:

```
module purge
module load conda
```

This module is based on the minimal Miniconda installer which includes the package and environment manager Conda that installs and updates packages and their dependencies. This module also provides Mamba, which is a drop-in replacement for most `conda` commands that enables faster package solving, downloading, and installing.

The next step is to initialize your shell to use Conda and Mamba:

```
mamba init bash
source ~/.bashrc
```

This modifies your `~/.bashrc` file so that Conda and Mamba are ready to use every time you log in (without needing to load the module).

If you want a newer version of Conda or Mamba than what is available in the module, you can also install them into one of your directories. We recommend installing either Miniconda or Mambaforge.

Conda can also be configured with various options. Read more about Conda configuration here.

Integrated development environments

JupyterLab, VS Code, RStudio, and other integrated development environments (IDEs) can be used on compute nodes via our OnDemand service. To install Jupyter kernels, see our guide here.

https://www.carc.usc.edu/user-information/user-guides/software-and-programming/conda

Using Anaconda on CARC

https://pytorch.org

PyTorch Build	Stable (2.0.0)	Preview (Nightly)		
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 11.7	CUDA 11.8	ROCm 5.4.2	CPU
Run this Command:	`conda install pytorch torchvision torchaudio pytorch-cuda=11.7 -c pytorch -c nvidia`			

Install PyTorch:

1 mamba create –n torch_env

2 mamba activate torch_env

3 mamba install pytorch torchvision torchaudio pytorch-cuda=11.7 -c pytorch -c nvidia

Test installation: type python and import torch

Using Anaconda & Juypyter Kernel

Creating Jupyter Kernel

A **Jupyter kernel** is a programming language-specific process that executes the code contained in a Jupyter notebook.

https://www.carc.usc.edu/user-information/user-guides/software-and-programming/jupyter-kernels

User Guides

- HPC Basics
- Data Management
- Software and Programming
  - Software Module System
  - Building Code With CMake
  - Using MPI
  - Using GPUs
  - Using Julia
  - Using Python
  - Using Anaconda
  - Using R
  - Using Stata
  - Using MATLAB
  - Using Rust
  - Using Launcher
  - Using Singularity
  - Using Tmux
  - **Installing Jupyter Kernels**
- Project and Allocation Management
- Hybrid Cloud Computing
- Secure Computing

Installing Jupyter Kernels

This user guide provides instructions for installing Jupyter kernels when using CARC OnDemand. For more information about OnDemand and using Jupyter notebooks, see the Getting Started with CARC OnDemand user guide.

A **Jupyter kernel** is a programming language-specific process that executes the code contained in a Jupyter notebook. The following provides installation instructions for a few popular Jupyter kernels, which will be installed in your home directory at `~/.local/share/jupyter/kernels`. Install the kernels when logged in to CARC systems before accessing them via the Jupyter OnDemand interactive app. To learn more about installing software on CARC systems using the software module system, see the Software Module System user guide.

When installing kernels, make sure to use descriptive names in order to distinguish among them. Once installed, when launching Jupyter on OnDemand, the kernels will show up on a Launcher tab (File > New Launcher) and when selecting kernels through other methods.

Many software kernels are available for use with Jupyter. See a full list here: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

Python

The default kernel is for Python 3.9.2, and this is ready to be used when Jupyter is launched. To use other versions of Python, enter a set of commands like the following:

```
module load usc python/<version>
python -m ipykernel install --user --name py376 --display-name "Python 3.7.6"
```

Jupyter Kernel

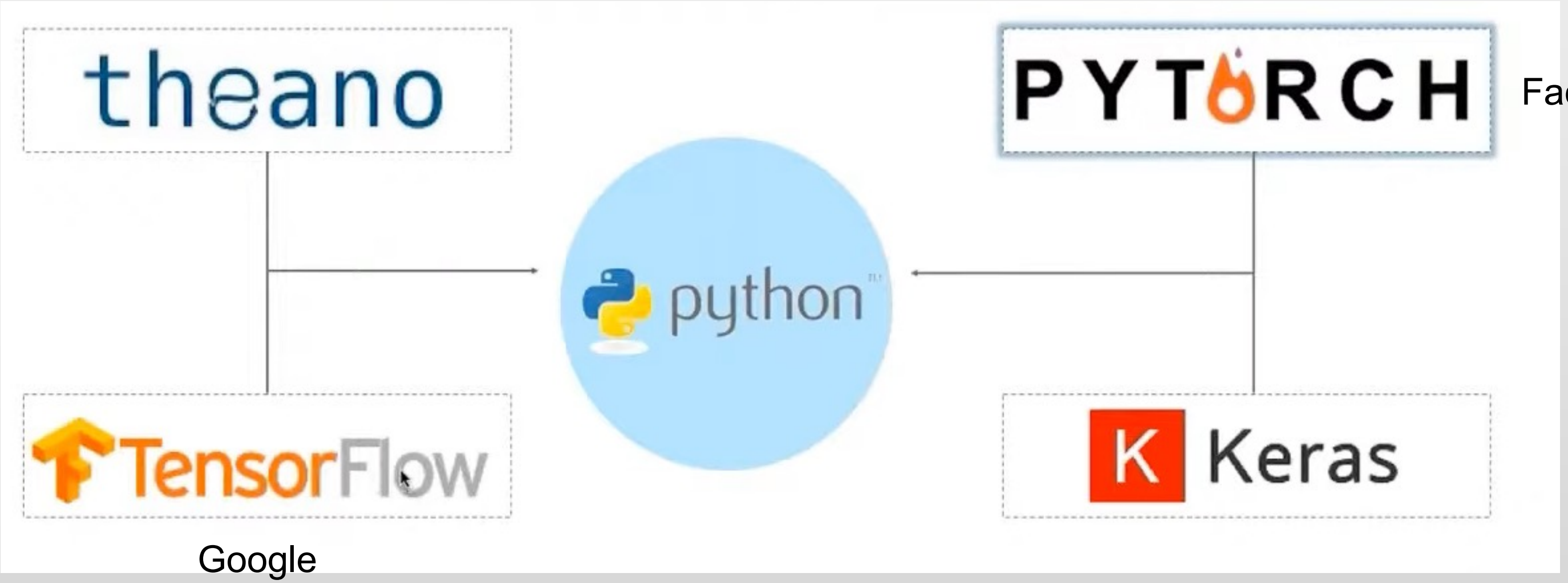
Install Jupyter kernel:

```
module purge
conda activate torch_env
mamba install -c conda-forge ipykernel
python -m ipykernel install --user --name torch_env --display-name "torch_env"
```

Jupyter Kernel

```
git clone https://github.com/jihao2021/workshop_building_deepNN_with_python.git
```

Neural Networks Packages



Facebook

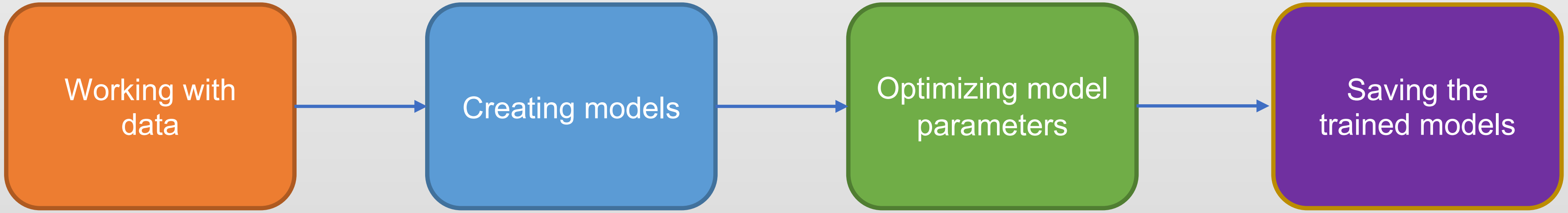
Google

Pytorch Tensors

- Tensors are a specialized data structure similar to arrays and matrices
- PyTorch uses tensors to encode the inputs and outputs of a model, as well as model’s parameters
- Can run on GPUs or other hardware accelerators
- Optimized for automatic differentiation

Pytorch Tutorial

Machine Learning Workflows



Pytorch Tutorial

Predict if an input image belongs to one of the following classes: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, or Angle boot.

FasionMNIST Dataset

Pytorch Tutorial: Working with Data

PyTorch

- Dataset: stores the samples and their corresponding labels TorchText, TorchVision, and TorchAudio

  torchvision.datasets: CIFAR, COCO, FashionMNIST

- DataLoader: wraps an iterable around the Dataset

Working with Data

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
```

Importing Modules

```
# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

Define Training and Test Dataset

Working with Data

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
```

Importing Modules

```
# Download training data from open datasets.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

Define Training and Test Dataset

Working with Data

Pass the Dataset as an argument to DataLoader

Wraps an iterable over dataset and supports automatic batching, sampling, shuffling and multiprocess data loading

```
batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

Define DataLoader

Creating Models

```
# Get cpu or gpu device for training.
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")

# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

Check if GPU is Available

Create Neural Networks Model

Optimizing the Model Parameters

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

Define Training Loop

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Define Test Dataset

```
def test(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            X, y = X.to(device), y.to(device)
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Training Loop

```
epochs = 5
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train(train_dataloader, model, loss_fn, optimizer)
    test(test_dataloader, model, loss_fn)
print("Done!")
```

Saving Models

```
torch.save(model.state_dict(), "model.pth")
print("Saved PyTorch Model State to model.pth")
```

Loading Models

```
model = NeuralNetwork()
model.load_state_dict(torch.load("model.pth"))
```

Make Predictions

```
classes = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
]

model.eval()
x, y = test_data[0][0], test_data[0][1]
with torch.no_grad():
    pred = model(x)
    predicted, actual = classes[pred[0].argmax(0)], classes[y]
    print(f'Predicted: "{predicted}", Actual: "{actual}"')
```