KIG4068 : Machine Learning

Week 6: Artificial Neural Networks (ANNs)

Semester 2, Session 2023/2024

Introduction to Neural Networks

Artificial neural networks (ANNs).

- ML model inspired by networks of biological neurons found in brains.
- ANNs have become quite different from their biological cousins.
- ANNs are at the very core of Deep Learning (DL).

Ideal for tackling large and highly complex tasks.

- Classifying billions of images (Google Images).
- Speech recognition services (Apple's Siri).
- Recommending the best videos to (YouTube).
- Super-human gaming (DeepMind's AlphaGo).

A Brief History of ANNs

Interest in ANNs has come in waves.

- First wave of interest kicked off by **McCulloch and Pitts (1943)**.
- Computational model of how biological neurons could perform complex computations using *propositional calculus*.
- Early success led to lots of hype! But by 1960s the hype had died down and ANNs fell into disuse as ANNs research entered its first "winter" period.

A second wave of interest in ANNs kicked off in the early 1980s.

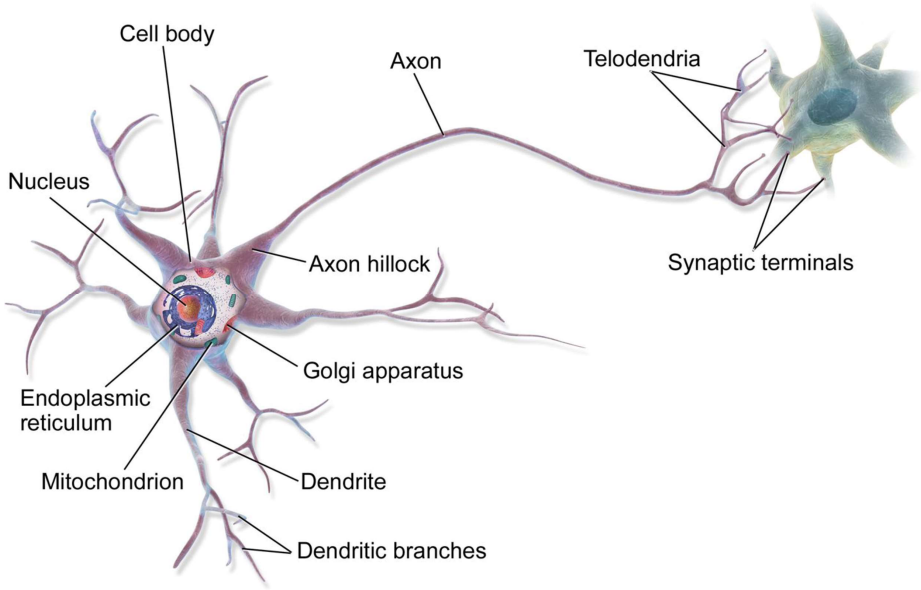
- In 1980s new ANN architectures invented, better training techniques developed. Lots of hype!
- In 1990s other powerful ML techniques were invented, such as SVMs, which seemed to offer better results than ANNs.
- Hype died down and ANN research entered its second "winter" period.

A Brief History of ANNs

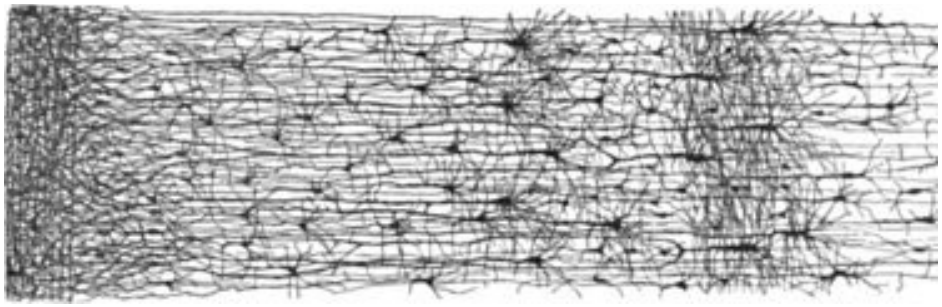
Now experiencing a third wave of interest in ANNs. Will we see a repeat of the past? Or is this time different?

- Huge quantities of data now available to train ANNs.
- Better training algorithms have been developed.
- Theoretical issues of ANNs have turned out to be benign in practice.
- ANNs often outperform ML techniques on large and complex problems.
- Increase in computing power (GPUs, TPUs, etc.) makes it possible to train large ANNs efficiently.

Biological Neurons



A biological neuron

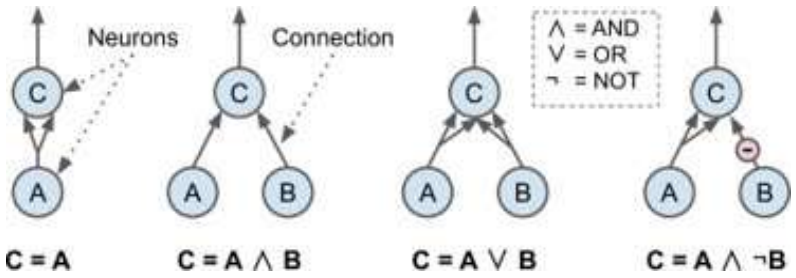


Multiple layers in a biological neural network (human cortex)

Logical Computations with ANNs

McCulloch and Pitts (1943) developed a simple model of a biological neuron.

- Artificial neuron has one or more binary inputs and one binary output.
- Artificial neuron activates when more than certain number of inputs are active.
- Proved that one build ANNs to compute any logical proposition.

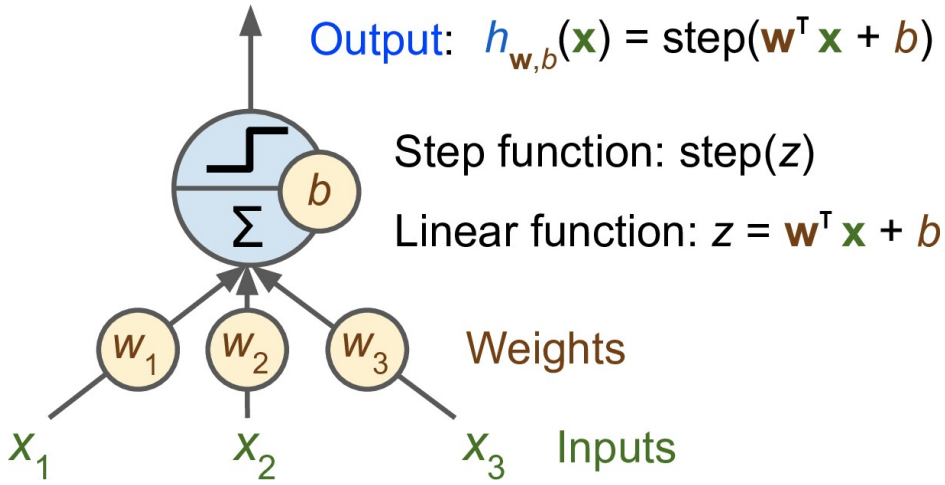


- The first network is the identity function.
- The second network performs a logical AND.
- The third network performs a logical OR.
- the fourth network computes a slightly more complex logical proposition.

The Perceptron - TLU

Threshold Logic Unit (TLU)

- Artificial neuron called a *threshold logic unit* (TLU).
- Inputs and output are numbers (or vectors).
- Each input connection is associated with a weight.



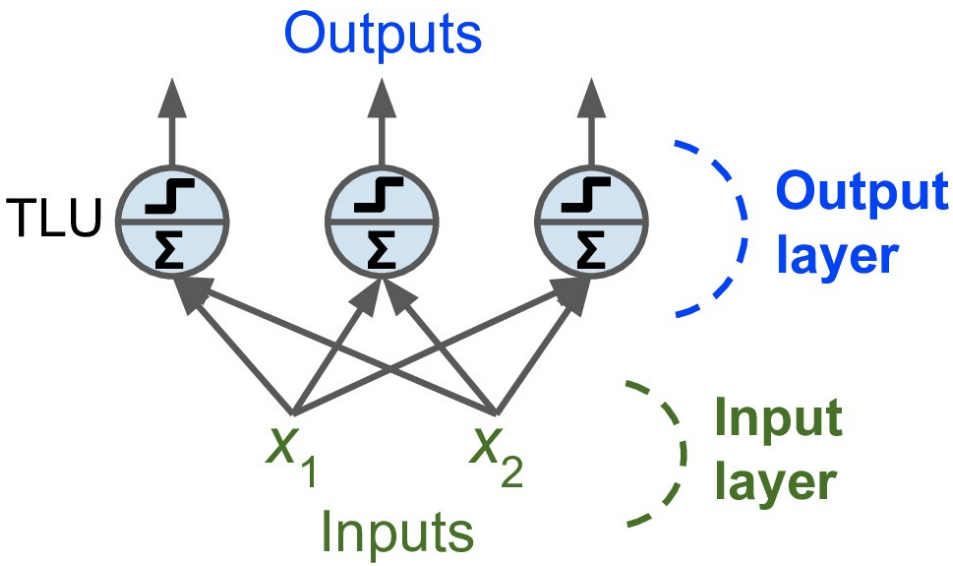
The Perceptron - TLU as a Binary Classifier

TLUs can perform linear binary classification.

- TLU first computes a linear function of its inputs and then applies a *step function* to the result.
- Most common step function used in TLU is the *Heaviside step function*.
- If result exceeds a threshold, then output positive class (else output the negative class).

The Perceptron

- Composed of one or more TLUs organized in a single layer: every TLU is connected to every input.
- Such layers are called *fully connected layer* or *dense layer*.
- Inputs form an "*input layer*"; since the layer of TLUs also produces the final output, it is also the *output layer*.



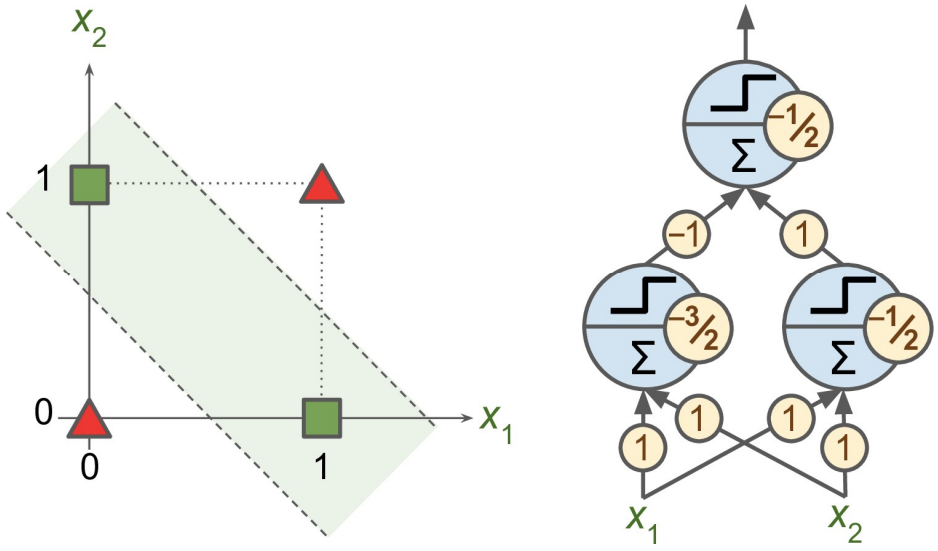
Architecture of a perceptron with two inputs and three output neurons

Perceptron in Scikit-Learn

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0)  # Iris setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new)  # predicts True and False for these 2 flowers
```

Issues with Perceptron

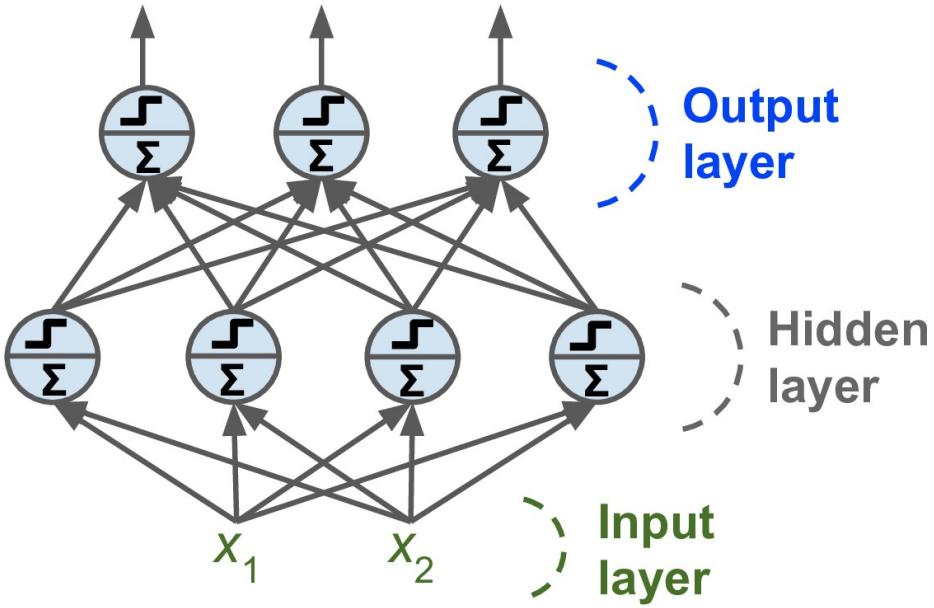
- Single perceptron can't solve the simple XOR classification problem.
- However, a two-layer perceptron *can* solve the XOR classification problem.
- Other limitations of perceptron can be addressed by adding multiple layers.



XOR classification problem and an MLP that solves it

The Multi-layer Perceptron (MLP)

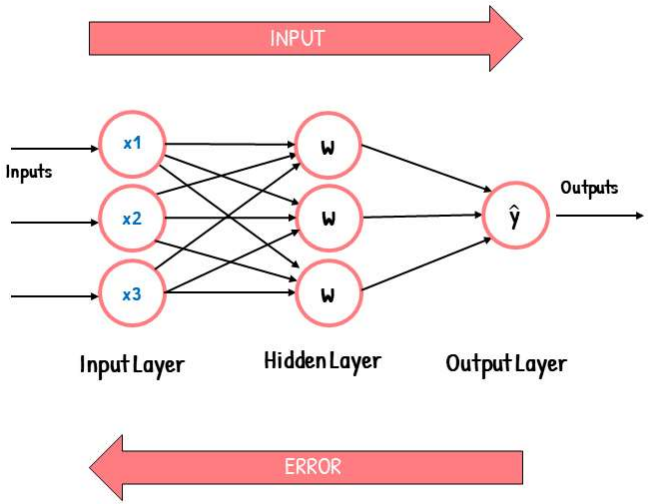
- An input layer, one or more layers of TLUs, called *hidden layers*, one final layer of TLUs called the *output layer.*
- ANN with a deep stack of hidden layers is called a *deep neural network* (DNN).
- A *feedforward neural network* (FNN) is when signal flows from inputs to the outputs.



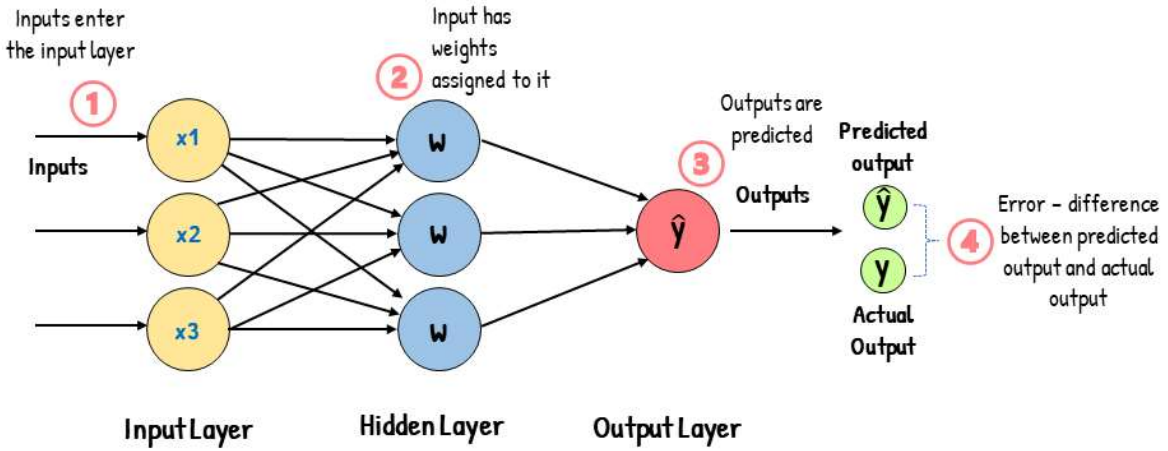
Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons

Backpropagation

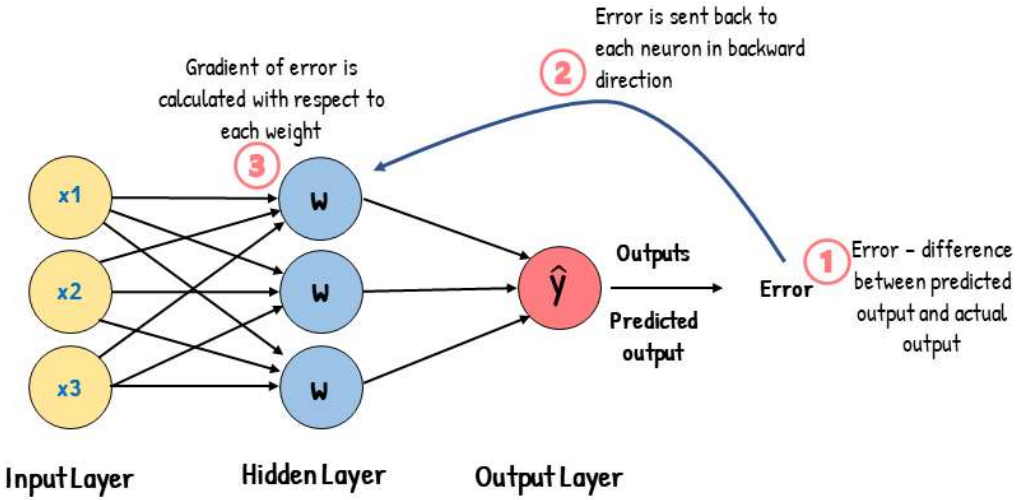
Neural Network Architecture



Feed-Forward Neural Network

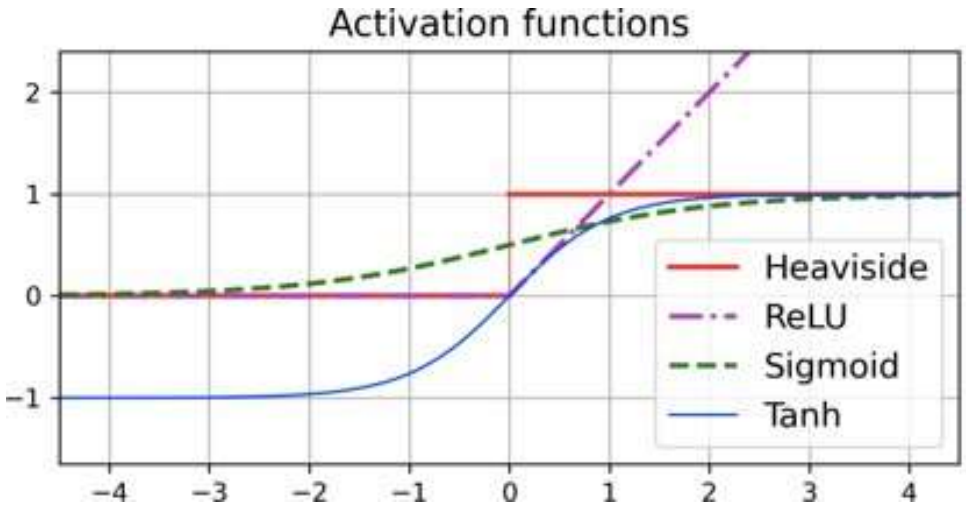


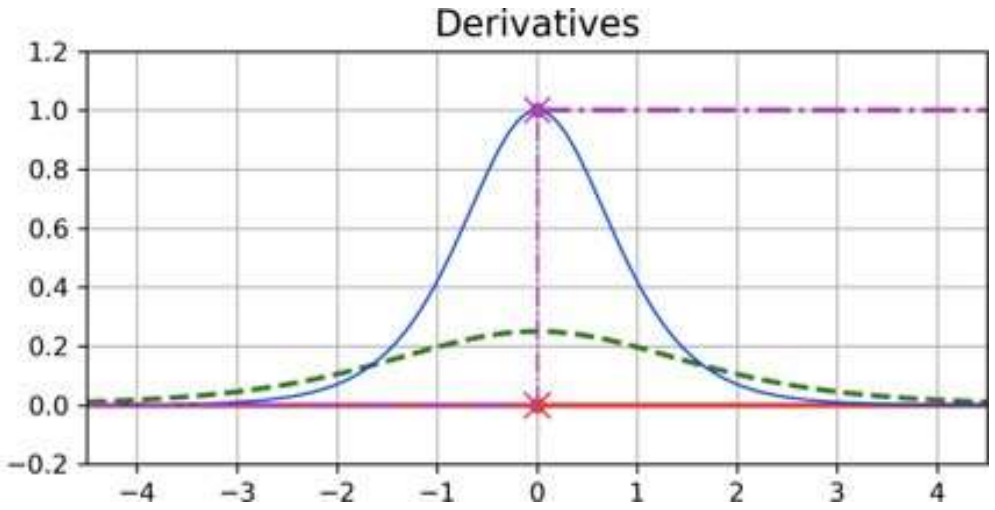
Backpropagation



Activation Functions

- Generalizes the step-function used in TLUs.
- Without non-linear activation functions perceptron would only be capable of approximating linear functions! e.g. f(x) = 2x+3 and g(x)=5x-1, which will give f(g(x))=2(5x-1)+3 = 10x+1….
- Popular activation functions are sigmoid, RELU, and hyperbolic tangent.
- A large enough perceptron with non-linear activations can approximate any continuous function.





MLPs for Regression

- If you want to predict a single value, then you just need a single output neuron: its output is the predicted value.
- To predict multiple values at once, you need one output neuron per output dimension.
- Often does not use any activation function for the output layer, so it's free to output any value it wants.
- Typically use (root) mean squared error loss but other loss functions are possible as well.

Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

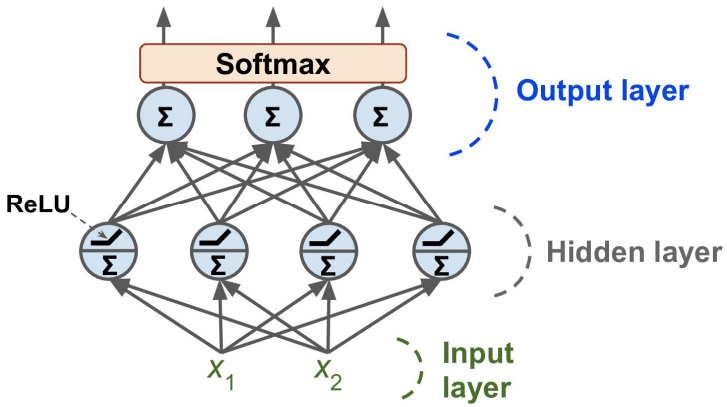
Typical regression MLP architecture

Regression MLPs in Scikit Learn

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)  # about 0.505
```

MLPs for Classification

- For a binary classification you need a single output neuron and the sigmoid activation function.
- For multi-label binary classification tasks, you need one output neuron per output dimension.
- For multi-class classification tasks, you need one output neuron per class and the (Log)SoftMax activation function.



A modern MLP (including ReLU and softmax for classification

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy

Typical classification MLP architecture

Classification MLPs in Scikit Learn

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

iris = load_iris()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    iris.data, iris.target, test_size=0.1, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, test_size=0.1, random_state=42)

mlp_clf = MLPClassifier(hidden_layer_sizes=[5], max_iter=10_000,
                        random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_clf)
pipeline.fit(X_train, y_train)
accuracy = pipeline.score(X_valid, y_valid)
accuracy
```

Tuning MLP Hyperparameters

- **Number of hidden layers:** start with a single hidden layer and then try adding a few hidden layers. Stop when you model is overfitting. Deep networks have a much higher *parameter efficiency* than shallow ones
- **Number of neurons per hidden layer:** Current "best practice" is to use the same number of neurons in all your hidden layers instead of one per layer. *Favor increasing the number of layers instead of the number of neurons per layer.*
- Try to build a model with slightly more layers and neurons than you need, then use early stopping and other regularization techniques to prevent it from overfitting too much.

Tuning MLP Hyperparameters

Learning Rate: *very* important hyperparameter. Basic search algorithm for "optimal" learning rate:

1. Train the model for a few hundred iterations, starting with a very low learning rate (e.g., 1e–5) and gradually increasing it up to a very large value (e.g., 10).
2. Plot the loss as a function of the learning rate (using a log scale for the learning rate).
3. The "optimal" learning rate will be a *lower* than the point at which the loss starts to climb (typically about 10 times lower than the turning point).

Tuning MLP Hyperparameters

- **Optimizer:** Don't just rely on SGD (or it's simple variants). Trying different optimizers can sometimes make a significant difference.
- **Batch Size:** Many researchers and practitioners recommend using the largest batch size that can fit in GPU RAM. But the optimal learning rate depends on the batch size so don't forget to tune your learning rate after changing your batch size.
- **Activation Functions:** The ReLU activation function will be a good default for all hidden layers, but for the output layer it really depends on your task

Sequential API (Keras – Tensorflow)

```python
import tensorflow as tf

# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

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

# Train the model
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y_valid))

# Evaluate the model
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_acc}')

# Make predictions
predictions = model.predict(X_test)
```

Optimizer: SGD, RMSprop, Adagrad,
refer here for more (https://keras.io/api/optimizers/)

Loss(reg): MAE, MSE etc
Loss(classification): Binary Cross-Entropy,
Categorical Cross-Entropy

Metrics: Accuracy, Precision, Recall etc