Introduction to Neural Networks

Learning like a human

Yordan Darakchiev

Technical Trainer

iordan93@gmail.com

Table of Contents

- sli.do: #neural-nets
- Neural networks – overview, problem statement
- Pros and cons
- Perceptron
- Feed-forward NNs (multi-layer perceptrons)
  - Training, applications for classification
- Convolutional neural networks – overview

Perceptron

The basic unit

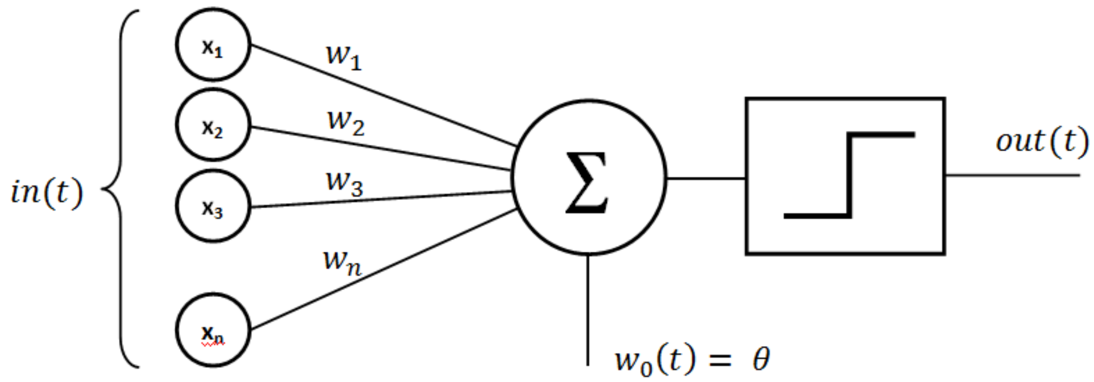
Neural Networks

- Neural networks try to mimic the way the human brain works
  - Series of interconnected artificial neurons (perceptrons)
  - Can do classification, regression, unsupervised learning, etc.
- Perceptrons were "invented" in the 1940s
  - Great development in the recent years
- "Deep learning" – ML algorithms using neural networks
- Cutting-edge applications
  - Machine translation
  - Speech recognition and generation
  - Image recognition
  - Game playing, etc.
- Some examples of deep learning applications

Neural Networks: Pros and Cons

- Can be used to model any datasets
  - Arbitrary dataset complexity
  - One type of algorithm can be used for many applications
- Do not provide any interpretability
  - The classification boundaries are hard to interpret
  - The model is mostly "black box"
  - NNs are not probabilistic (we can't get a confidence metric)
  - "The Dark Secrets at the Heart of AI"
    - Solutions: trying to explain decisions, combining with other algorithms, etc.
- Can be slow
  - Other models usually train a lot faster, even if we use special hardware
- **NNs are not a substitute for understanding the problem deeply**

Perceptron



- The main "NN unit"
- Algorithm
  - Assign each input a weight
  - Sum all weighted inputs
  - Pass the sum to an activation function
    - Decides whether the neuron will activate (i.e. return 1) → binary output
    - Usually sigmoid function or step function
  - As a result, the perceptron returns 0 or 1
- One perceptron is enough to solve any **linearly separable** problem
- Usage in scikit-learn

```
from sklearn.linear_model import Perceptron
perceptron = Perceptron()
```

Perceptron Learning

- The algorithm definition is not enough
  - We have to update the weights
- Adaline (**ADA**ptive **LIN**ear **E**lement) – a perceptron that can learn
  - Implements the same linear model: $y = \sum_{j=1}^{m} w_j x_j + \theta$
    - Reformulation: using a bias term
      $x_{n+1} = 1,\ w_{n+1} = \theta \Rightarrow y = wx$
- Adaline learning – update the weights using $w = w + \eta(\tilde{y} - y)x$
  - w – weights
  - η – learning rate (positive constant, model hyperparameter)
  - $\tilde{y}$ – model output
  - y – desired output

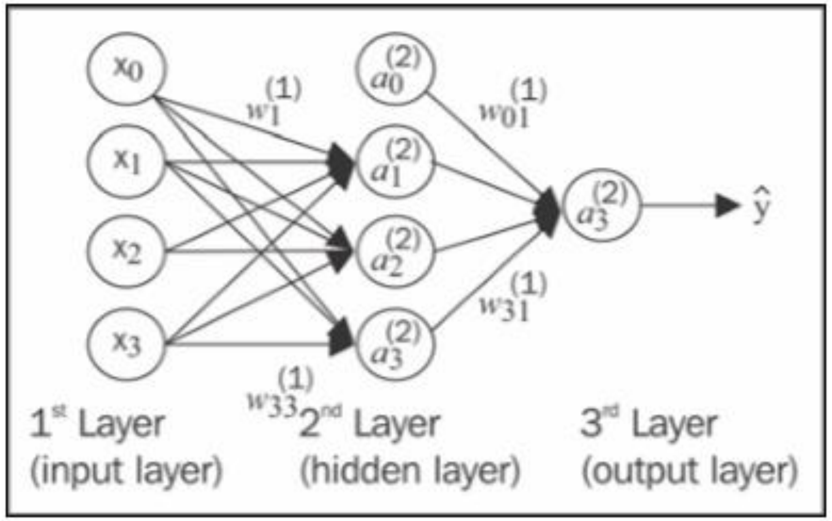
Example: Perceptron

- Generate 2D blobs using scikit-learn
  - Make sure they're linearly separable
  - Make sure you create enough samples, e.g. 10 000
- Use a perceptron to classify samples as belonging to one blob or another
- Test and score the performance
  - Score both in-sample and out-of-sample data
    - It's useful to compare the two metrics – this can sometimes detect underfitting or overfitting
  - Print a confusion matrix (and optionally, other metrics, e.g. ROC curve)
  - * Optionally, perform hyperparameter tuning
    - Especially if your blobs are too close

Neural Networks

Combining perceptrons to achieve glory

Neural Network Architecture



- Neural network layout
  - Input(s) (+ bias unit)
  - "Hidden layers" (+ bias units)
  - Output(s)
- Each "node" is a perceptron
- Each arrow carries 0 or 1, and is assigned a weight
- The layers are fully connected
  - There are no connections within layers
- More than 1 hidden layer → "deep learning" (deep NN)
- How many layers? How many units per layer?
  - We don't know :(⇒ hyperparameter tuning

Neural Network Architecture (2)

- Many layers ⇒ the error gradients become too small
  - "Vanishing gradient problem"
    - Small gradient = too slow learning
  - Special algorithms have been developed to deal with this
    - E.g. pre-train the NN one layer at a time using unsupervised learning,
      use some form of "back-links" (recurrent NNs),
      use a genetic algorithm instead of gradient descent
  - This is the field of deep learning
- Classification using NNs
  - Two classes: one output (0 or 1)
  - Many classes: use one-hot encoding to represent each class, have as many outputs as there are classes

Neural Network Learning

- The type of NN we look at is called a "feed-forward NN"
  - Data flows only forward, there are no "back-links"
- Learning algorithm:
  - Forward propagation / backpropagation
  - Using the data, propagate the patterns from input to output
  - Based on the output, calculate the error (using a cost function)
  - Backpropagate the error (using derivatives), update the model
- We get the "final" weights after repeating the process for several epochs
- The maths is a bit ugly
  - You can read an explanation here

Neural Network Learning (2)

- Classification: just use one-hot encoding
  - MLP = multi-layer perceptron

```
from sklearn.neural_network import MLPClassifier
```

- Regression: no activation function at the output layer

```
from sklearn.neural_network import MLPRegressor
```

- Regularization: parameter `alpha`
  - Increasing = less overfitting
  - A visual comparison of regularization parameters
- Tips
  - A neural network is very sensitive to feature scaling
    - [0; 1], [-1; 1] or Z
    - Use a scaler, e.g. `StandardScaler`
  - Use fine-tuning to optimize `alpha`
    - Usually in the range `10.0 ** -np.arange(1, 7)`

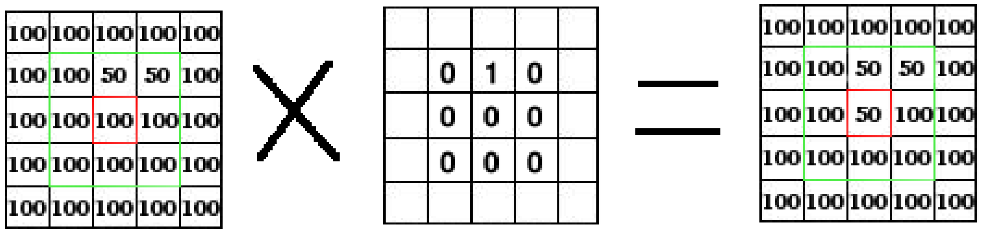
Example: Classifying Handwritten Digits

- Obtain the MNIST dataset of handwritten digits
  - This is a famous dataset for learning and comparing neural networks
  - Each data point represents a 28 x 28 image of a digit (0 – 9)
- Train a simple NN on the MNIST dataset
  - Choose a reasonable number of layers and units per layer, e.g. {3, 3}
- Test, score and evaluate the classification performance
  - E.g. accuracy, precision, recall, F1, confusion matrix, ROC curve
- * Try several other architectures (e.g. more layers, more units per layer, different structure, e.g. 2 + 3 + 2 units, etc.)
- * Compare the results with (an)other classifier(s), e.g. SVM

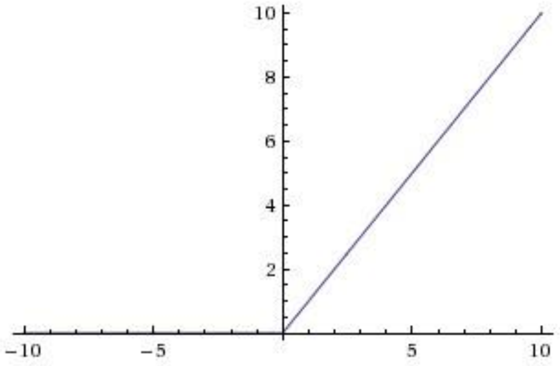
Convolutional Neural Networks

- Feed-forward NNs inspired by the visual cortex
  - Very useful for working on images
    - Recognition and classification
  - Example: https://www.clarifai.com/demo
- Main operations
  - Convolution
  - Non-linearity (ReLU – **Re**ctified **L**inear **U**nits)
  - Pooling (downsampling, subsampling)
  - Classification (fully-connected layer)
- Idea: before proceeding to classification, perform image processing using convolution, pooling and ReLU
  - This process produces the input features of a classical NN

Convolutional Neural Networks (2)



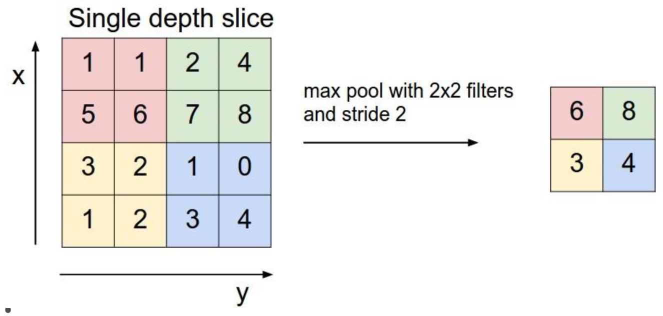
- Convolution: slide a particular matrix (called kernel) over the image
  - This produces a new, smaller matrix representing the convolved feature (called feature map)
    - Many filters => "stacked" matrices (3D feature map)



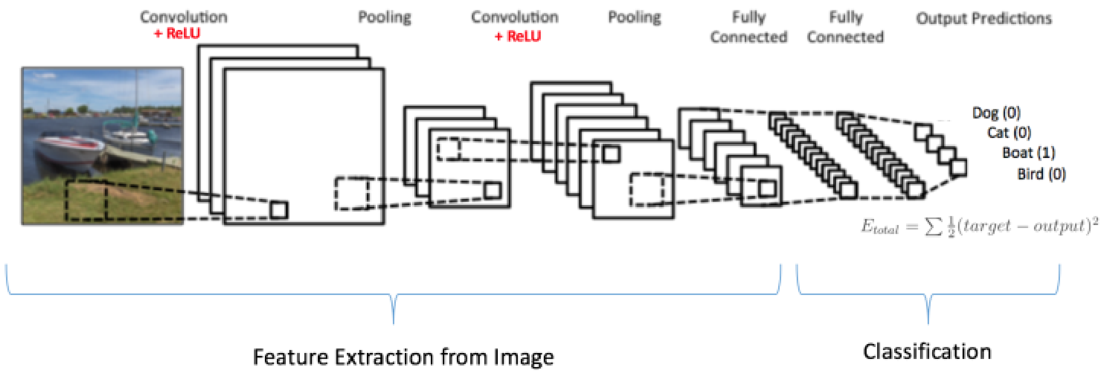
- Rectification (ReLU): $y = \max(0, x)$
  - Performed after each convolution
  - Replaces all negative feature map values with 0
    - Convolution is a linear operation and we want to learn non-linear features; this unit introduces non-linearity
  - It's also possible to use other functions
  - Result from the operation: rectified feature map

Convolutional Neural Networks (3)

- Pooling (downsampling)
  - Reduces the feature map size
  - Types: max, average, sum, etc.
    - The function is applied to a "window" over the image
    - Stride: how many pixels to "skip"



- The combination of convolution, ReLU and pooling layers should output a 1D number vector
  - Represents high-level features of the input image
  - Used as the input to a "standard" feed-forward NN
    - Called a fully-connected layer
  - The NN outputs the final result (e.g. image class)



Summary

- Neural networks – overview, problem statement
- Pros and cons
- Perceptron
- Feed-forward NNs (multi-layer perceptrons)
  - Training, applications for classification
- Convolutional neural networks – overview

Questions?