Deep Learning

COSC440

Andy Ming / Quelldateien

Table of contents

Details

Science of Arrays

Use Arrays wisely

Don't loop over elements in a array. Use numpy functions to do elementwise operations:

```
# Elementwise sum; both produce an array
z = x + y
z = np.add(x, y)
```

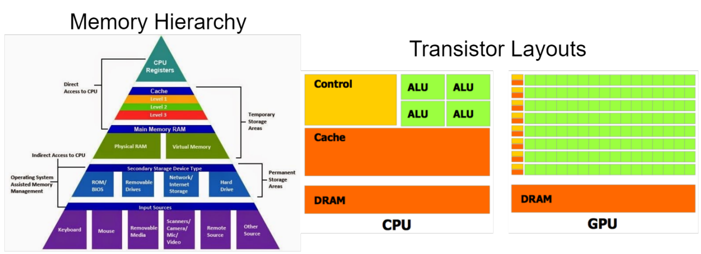
Use Boradcasting to work with arrays of different sizes:

```
# We will add the vector v to each row of the
↪ matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10,
↪ 11, 12]])
v = np.array([1, 0, 2])
y = x + v.T  # Add v to each row of x using
↪ broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

Do **Matrix Multiplications**, remember that matrices of shape 100x20 × 20x40 equal a output shape of 100x40:

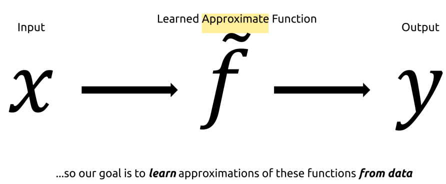
```
C = np.dot(A,B)
F = np.matmul(D,E)
```

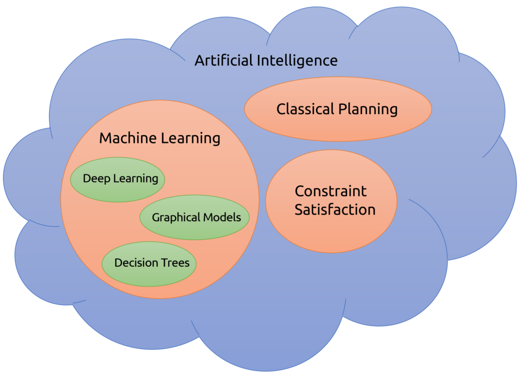
Reason:



Machine Learning Concepts

Machine Learning == Function Approximation



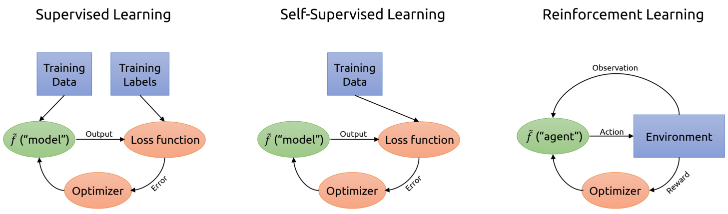


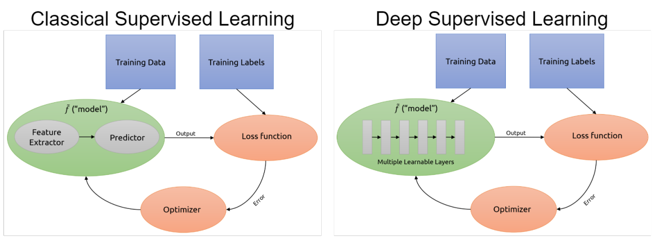
- Repeat for N iterations, or until the weights no longer change:
  - For each training example $\mathbf{x}^k$ with label a^k:
    1. Calculate the prediction error:
       * If $a^k - f(\mathbf{x}^k) = 0$, continue (no change to weights).
    2. Otherwise, update each weight w_i using:

$$w_i = w_i + \lambda \left(a^k - f(\mathbf{x}^k)\right) x_i^k$$

- where λ is a value between 0 and 1, representing the learning rate.

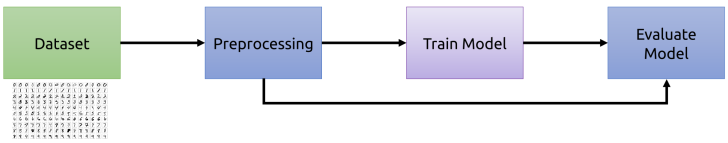
Types of Learning





Types of Problems

Maschine Learning Pipeline



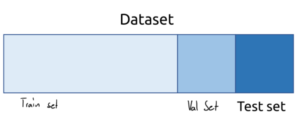
Dataset

Annotated Datasets like MNIST (Handwritten digits).

Preprocessing

Split the dataset into **Train, Validation, and Test sets**

- ***Train set*** — used to adjust the parameters of the model
- ***Validation set*** — used to test how well we're doing as we develop
  - Prevents ***overfitting***, something you will learn later!
- ***Test set*** — used to evaluate the model once the model is done



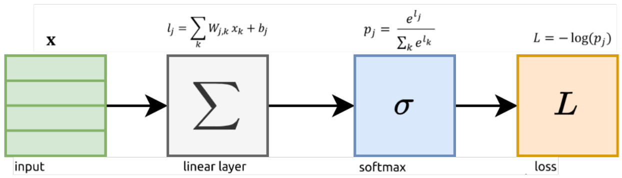
Train Model

1. **Initialization**: Set all weights w_i to 0.
2. **Iteration Process**:

Optimizing with Gradient Descent

Loss Function

Function L which measures how "wrong" a network is. We want our network to answer right with **high probability**.

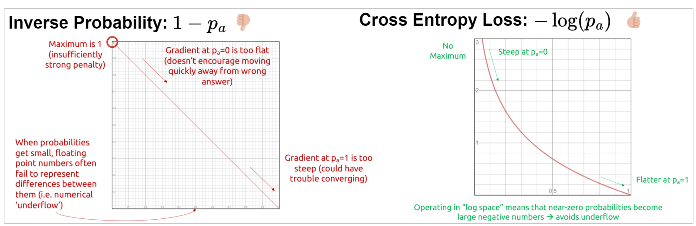


To get a probability for **binary classification**, we introduce a **probability layer**. One of the possible function is **Softmax**

$$p_j = \frac{e^{l_j}}{\sum_k e^{l_k}}$$

For every output j it takes every logit (output of network before activation/probability is applied) l_j in the exponent to ensure positivity. Dividing it by the sum of all logits ensures that $\sum_k p_k = 1$.

To get the loss L we apply a loss-function, *low probability* $\rightarrow$ *high loss*. We use **Cross Entropy Loss**

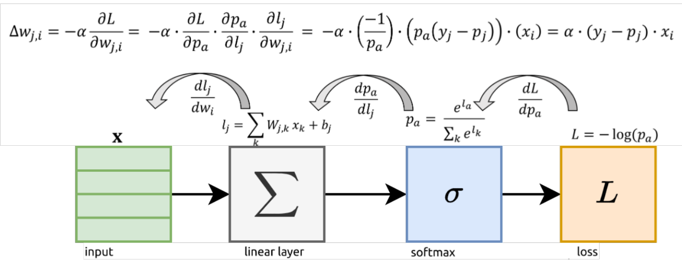


Gradient Descent

$\Delta w_{j,i} = -\alpha \frac{\partial L}{\partial w_{j,i}}$

α: learning rate *(typically 0.1-0.001)*
L: loss function
$w_{j,i}$: one single weight

To compute $-\alpha \frac{\partial L}{\partial w_{j,i}}$ use the chain rule




```
## Backpropagation on batch learning
# y = expected - (f(x)>0)
labels_OH = np.zeros((labels.size, self.num_classes),
↪  dtype=int)
labels_OH[np.arange(labels.size),labels] = 1  #
↪  One-Hot encoding
predictions = np.argmax(outputs, axis=1)
predictions_OH = np.zeros_like(outputs)
predictions_OH[np.arange(outputs.shape[0]),
↪  predictions] = 1
y = labels_OH - predictions_OH
# db = y*1
gradB = np.mean(y, axis=0)   # average over batch
# dW = y*x
y = y.reshape((outputs.shape[0],1,self.num_classes))
inputs =
↪  inputs.reshape((outputs.shape[0],self.input_size[0]*self.input_size[1],1))
dW = inputs*y
gradW = np.mean(dW, axis=0)  # average over batch
```

Stochastic Gradient Descent (SGD)

Train a network on **batches**, small subsets of training data.

```
# Stochastic Gradient Descent
for start in range(0, len(train_inputs),
↪  model.batch_size):
    inputs =
↪  train_inputs[start:start+model.batch_size]
    labels =
↪  train_labels[start:start+model.batch_size]
    # For every batch, compute then descend the
    ↪  gradients for the model's weights
    outputs = model.call(inputs)
    gradientsW, gradientsB =
↪  model.back_propagation(inputs, outputs, labels)
    model.gradient_descent(gradientsW, gradientsB)
```

- Training process is *stochastic* / *non-deterministic*: batches are a random subsample.
- The gradient of a random-sampled batch is a unbiased estimator of the overall gradient of the dataset.
- Pick a large enough batch size for *stable updates*, but small enough to *fit your GPU*

Optimization

Automatic Differentiation

To avoid having to recalculate the whole chain every time a new layer is added, we use *automatic derivation*. There are several options:

Numeric differentiation

- $\frac{df}{dx} \approx \frac{f(x+\Delta x)-f(x)}{\Delta x}$
- Called *finite differences*
- Easy to implement
- Arbitraritly inaccurate/unstable

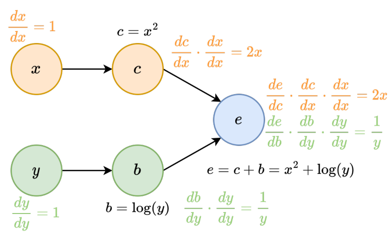
Symbolic differentiation

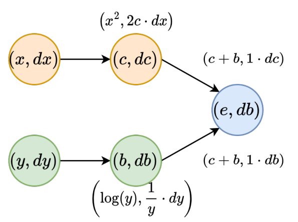
- $\frac{dx^2}{dx} = 2x$
- Computer does algebra and simplifies expressions
- Very exact
- Complex to implement
- Only handles static expressions

Automatic differentiation

- Use the chain rule at runtime
- Gives exact results
- Handles dynamics
- Easier to implement
- Can't simplify expressions

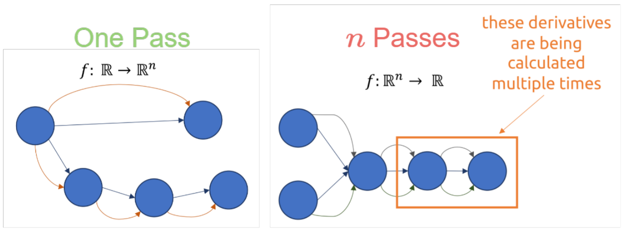
Forward Mode Autodiff Every node stores its (value, derivative) in a tuple, called **dual numbers**. To compute the overall derivative, each derivative can be chained up. This is implemented via **Overloading**, every function / operator has multiple definitions based on the types of the arguments. ML-Framwork functions work on these tuples.



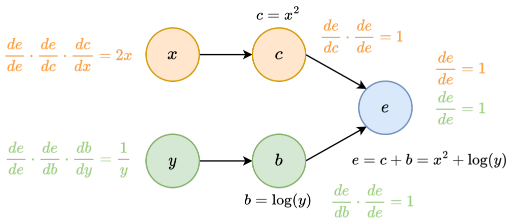


Time Effect: $O(N * M)$ time, $O(1)$ memory, with N = number of inputs, with M = number of nodes

Issue w/ forward mode

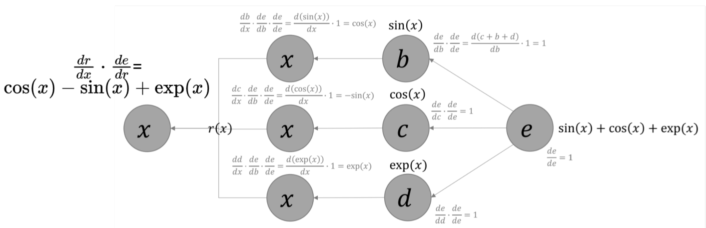


Reverse Mode Autodiff First, run the function to produce the graph, then compute the **derivatives backward**.



- Analog to the forward mode: overload math functions/operators
- Overloaded function return *Node* objects
- Overloaded functions build compute graph while executing
- After forward pass, the operations are recorded
- The backwards pass walks along the graph and computes the derivatives
- **Time Effect**: $O(M)$ time, $O(M)$ memory, with M = number of nodes

Fan-Outs (Reverse) The way to handle fan-out is to **add** the derivatives of the fanned-out nodes through replication $r(x)$.



Diagnosis Problems

Deep Learning Concepts

Common Misconception

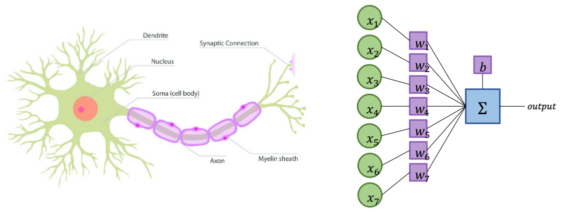
Deep Learning != AI, Just because deep learning algorithms are used doesn't mean there is any intelligence involved.
Deep Learning != Brain, Modern deep nets don't depend solely on *biologically mimiced neural nets* any more. A fully connected layer represents such a neural net the closest.
Deep Learning ==:

1. *Differentiable functions*, composed to more complex diff. func.
2. A deep net is a differentiable function, some inputs are *optimizable parameters*
3. Differentiable functions produce a computation graph, which can be traversed backwards for *gradient-based optimization*

Multi-Dimensional Arrays & Memory Models

Neural Networks

Perceptron



Biological Neuron Artificial Neuron (Perceptron)

Predicting with a Perceptron:

1. Multiply the inputs x_i by their corresponding weight w_i
2. Add the bias b
3. **Binary Classifier**, greater than 0, return 1, else return 0

$$f_\Phi(\mathbf{x}) = \begin{cases} 1, & \text{if } b + \mathbf{w} \cdot \mathbf{x} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Parameters

Weights: "importance of the input to the output"

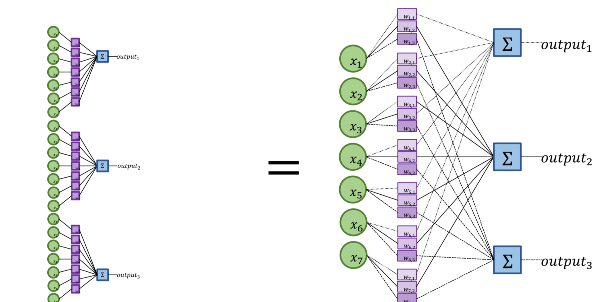
- Weight near 0: Input has little meaning to the output
- Negative weight: Increasing input $\rightarrow$ decreasing output

Bias: "a priori likelihood of positive class"

- Ensures that even if all inputs are 0, there is some result
- Can also be written as a weight for a constant 1 input

$$[x_0, x_1, x_2, \ldots, x_n] \cdot [w_0, w_1, w_2, \ldots, w_n] + b$$
$$= [x_0, x_1, x_2, \ldots, x_n, 1] \cdot [w_0, w_1, w_2, \ldots, w_n, b]$$

Multi-Class Perceptron

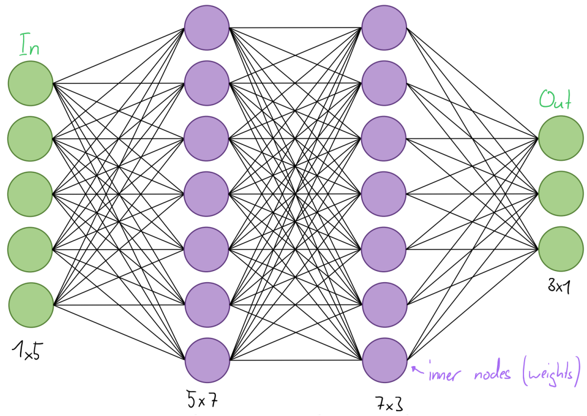


Three separate perceptrons Three perceptrons sharing inputs

Biary Classifier: Only one output can be active $\hat{y} = \operatorname{argmax}\left(f\left(x^k\right)\right)$, thus the update terms are

$$\Delta w_i = \begin{cases} 0, & \text{for } a^k = \hat{y} \\ -x_i^k, & \text{for } \hat{y} = 1, a^k = 0 \\ x_i^k, & \text{for } \hat{y} = 0, a^k = 1 \end{cases}$$

Multi-Layer



Sequential and Recurrent Networks

Latent Space

Transfer Learning

Training Methods and Tricks

Deep Learning Problems, Models & Research

Computer Graphics and Vision

Natural Language

Audio and Video Synthesis

Search using Deep Reinforcment Learning

Anomaly Detection

Irregular Networks