

COSC440



1

Science of Arrays 1

1

Types of Learning	2
Types of Problems	2
Machine Learning Pipeline	2
Dataset	2
Preprocessing	2
Train Model	2
Optimizing with Gradient Descent	2
Loss Function	2
Gradient Descent	2
Stochastic Gradient Descent (SGD)	3
Optimization	3
Automatic Differentiation	3
Diagnosis Problems	3

3

Multi-Dimensional Arrays & Memory Models	3
Neural Networks	3
Perceptron	3
Multi-Layer	4
Sequential and Recurrent Networks	4
Latent Space	4
Transfer Learning	4
Training Methods and Tricks	4

4

Computer Graphics and Vision	4
Natural Language	4
Audio and Video Synthesis	4
Search using Deep Reinforcement Learning	4
Anomaly Detection	4
Irregular Networks	4

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

Use Broadcasting to work with arrays of different sizes:

Do **Matrix Multiplications**, remember that matrices of shape $100 \times 20 \times 20 \times 40$ equal a output shape of 100×40 :

```
F = np.matmul(D,E)
```

Memory Hierarchy

The diagram illustrates the memory hierarchy of a CPU, structured as a pyramid with layers representing different memory types and their access characteristics. The layers, from top to bottom, are:

- GPU Registers:** The top layer, with "Direct Access to CPU".
- Cache:** Divided into "Level 1" and "Level 2".
- Main Memory SDRAM:** Divided into "Physical SDRAM" and "Virtual Memory".
- Secondary Storage Device Type:** Divided into "SSD/HDD/BSS", "Non-volatile Memory", "Network/Internet Storage", and "Hard Drive".
- Input Sources:** The base layer, including "Keyboard", "Mouse", "Removable Media", "Memory/Camera/Video", "Network Source", and "Other Source".

Annotations on the right side of the pyramid indicate:

- Temporary Non-Storage Access:** Points to the Cache and Main Memory SDRAM layers.
- Increased Access Times:** Points to the Secondary Storage Device Type layer.

Text on the left side of the pyramid indicates:

- Operating System Assisted Memory Management:** Points to the bottom layers of the hierarchy.

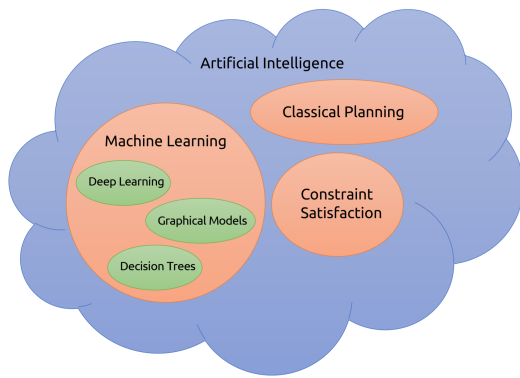
Transistor Layouts

The diagram shows the transistor layouts for a CPU and a GPU. The CPU layout is divided into three main sections: Control, Cache, and DRAM. The GPU layout is divided into two main sections: Control and DRAM. The Control section of the CPU is yellow and contains four ALU units. The Cache section is orange. The DRAM section is orange. The GPU layout shows a large array of green rectangles representing transistors, with a small orange section labeled "DRAM" at the bottom.

CPU	GPU
Control (Yellow)	Control (Green)
Cache (Orange)	Cache (Green)
DRAM (Orange)	DRAM (Orange)

Machine Learning == Function Approximation



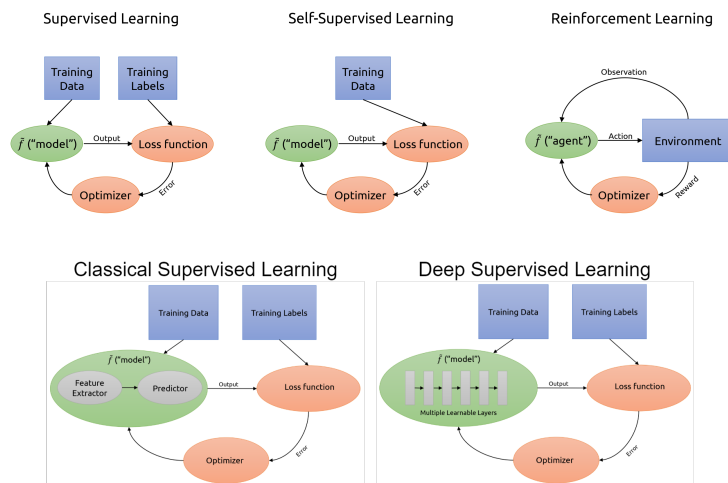


- Repeat for N iterations, or until the weights no longer change:
 - For each training example \mathbf{x}^k with label a^k :
 - Calculate the prediction error:
 - If $a^k - f(\mathbf{x}^k) = 0$, continue (no change to weights).
 - Otherwise, update each weight w_i using:

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

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

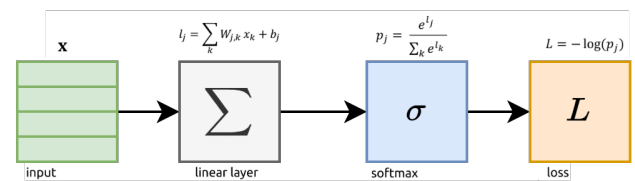
Types of Learning



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**.

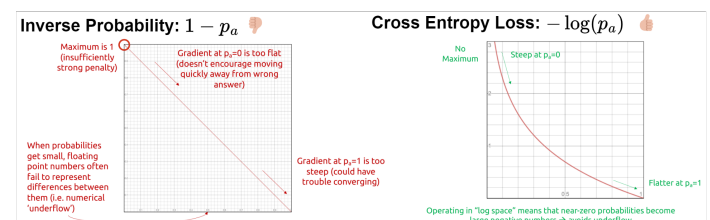


To get a probability for **binary classification**, we introduce a **probability layer**. One of the possible function is **Softmax**

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

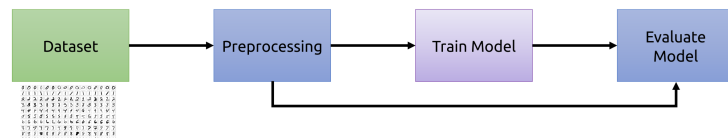
For every output j it takes every logit (output of network before activation/probability is applied) l_j in the exponent to ensure positivity. Dividing it by the sum of all logits ensures that $\sum_k p_k = 1$.

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



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

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

Gradient Descent

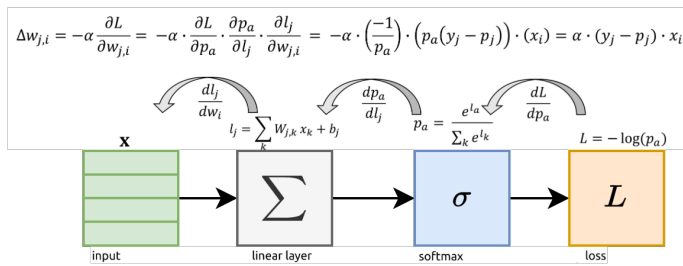
$$\Delta w_{j,i} = -\alpha \frac{\partial L}{\partial w_{j,i}}$$

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*x
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

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 mimicked neural nets* any more. A fully connected layer represents such a neural net the closest.

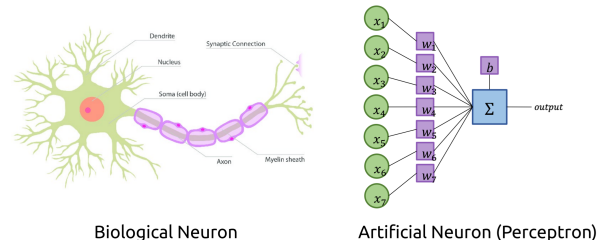
Deep Learning ==:

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



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"

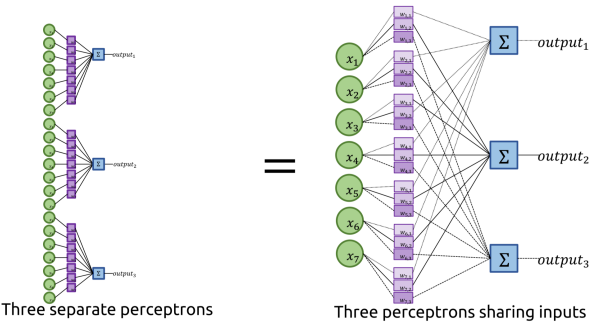
- Weight near 0: Input has little meaning to the output
- Negative weight: Increasing input \rightarrow decreasing output

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

$$\begin{aligned} &[x_0, x_1, x_2, \dots, x_n] \cdot [w_0, w_1, w_2, \dots, w_n] + b \\ &= [x_0, x_1, x_2, \dots, x_n, 1] \cdot [w_0, w_1, w_2, \dots, w_n, b] \end{aligned}$$

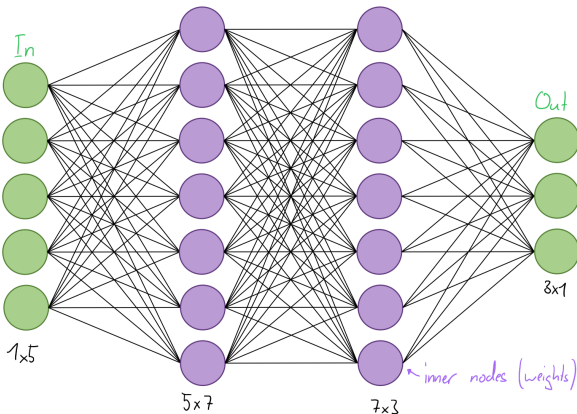
Multi-Class Perceptron



Biary Classifier: Only one output can be active $\hat{y} = \operatorname{argmax} (f(x^k))$, 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 Reinforcement Learning

Anomaly Detection

Irregular Networks