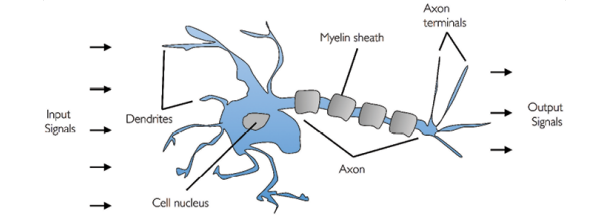
**2. Rosenblattův perceptron**

**2.1 Inspirace a historický kontext**

Umělé neuronové sítě jsou inspirovány skutečným uspořádáním nervové soustavy. Abychom správně pochopili jejich architekturu a fungování, zamysleme se nejdříve nad fungováním lidského mozku. Ten může obsahovat až 100 miliard mozkových buněk (neuronů – viz obr. 2.2), které považujeme za jeho základní stavební jednotky. Neurony jsou mezi sebou propojeny synapsemi a předávají si chemické a elektrické signály přijímané dendrity. Pokud intenzita elektrického potenciálu přesáhne určitý práh, neuron vypálí signál dále po axonu a přenáší tak informaci. V případě opakujících se vzorců chování jsou zodpovědné synapse posíleny.3 Tento koncept v roce 1943 poprvé představili Warren McCulloch a Walter Pitts s cílem navrhnout fungující umělou inteligenci. Mozkovou buňku popsali jako prostý rozhodovací mechanismus s binárním výstupem (jejich úspěšný model nese název *MCP – McCulloch-Pitts neuron* – viz obr. 2.1).4

****

Obr. 2.1 Model MCP neuronu

Obsah obrázku strom, posazený, exteriér, větev

Popis byl vytvořen automatickyJen o pár let později publikoval Frank Rosenblatt na základě MCP neuronu první koncept nejjednoduššího modelu jednovrstvé neuronové sítě, takzvaný perceptron. Tento algoritmus se řídí velmi jednoduchými pravidly inspirovanými skutečnými mozkovými buňkami. Jeho cílem je optimalizovat váhové koeficienty, kterými jsou násobeny příznaky vstupních dat pro výpočet „intenzity signálu“. Jedná se o první umělý neuron, který poskytuje základ pro všechny složitější architektury umělých neuronových sítí. Podobnosti mezi umělými a biologickými neurony budou stále viditelnější při detailnějším vysvětlení v dalších kapitolách.

Obr. 2.2 Mikroskopický snímek neuronu

**2.2** **Formální definice**

Perceptronový algoritmus funguje dobře zejména při řešení problému binární klasifikace. Představme si, že chceme naše data na základě jejich příznaků rozdělit do dvou tříd, a . Perceptron pro každý příklad vypočítá lineární kombinaci vektoru příznaků a vektoru váhových koeficientů .

Tato práce důsledně využívá maticový zápis formulí. Tedy zvýrazněný font znamená, že pracujeme s vektorovou, případně maticovou (zejména matice vah ) veličinou.

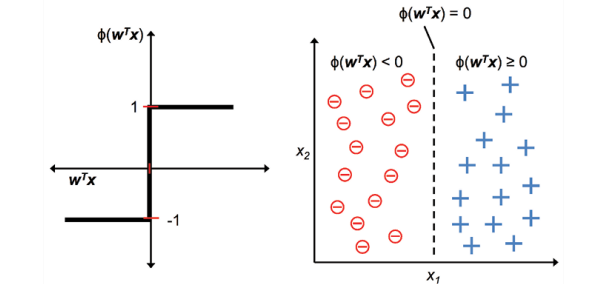
Pokud výsledná hodnota přesáhne stanovený práh klasifikujeme příklad jako třídu (neuron pálí), pokud bude nižší, připadne příklad třídě (neuron není aktivován). Výsledné hodnotě lineární kombinace říkáme lineární vstup (z anglického *net input*). O tom, zda lineární vstup přesáhne stanovený práh rozhoduje *skoková funkce* **,** na kterou zatím nelze použít gradientní přístup optimalizace (kvůli její nespojitosti – viz kapitola 3). Funkcím, které mapují lineárním vstupem neuronu na jiné hodnoty, se v kontextu umělých neuronových sítí říká *aktivační funkce*.5 Budu používat variantu skokové funkce, která převádí kladné proměnné na hodnotu 1 a záporné na -1 (obvykle jsou záporné proměnné převáděny na hodnotu 0, pro můj příklad se však více hodí tato varianta – viz obr. 2.3):

Neboli:

Následně můžeme z rovnice práh odstranit úplně tím, že prvnímu váhovému koeficientu přiřadíme hodnotu a korespondující první příznaky hodnotu 1:

Negativní práh nazýváme bias. Ten můžeme považovat za základní hodnotu neuronu, kterou budeme vždy přičítat k lineární kombinaci. Zápis rovnice pro výpočet lineárního vstupu můžeme nyní napsat v kompaktní vektorové formě1:

Proces klasifikace vstupních dat podle perceptronu můžeme tedy jednoduše shrnout do těchto kroků:

1. Výpočet lineární kombinace vektoru váhového vektoru a vektoru příznaků příkladu a přičtení jednotky biasu
2. Vyhodnocení klasifikace lineárního vstupu podle skokové aktivační funkce (nebo **)**

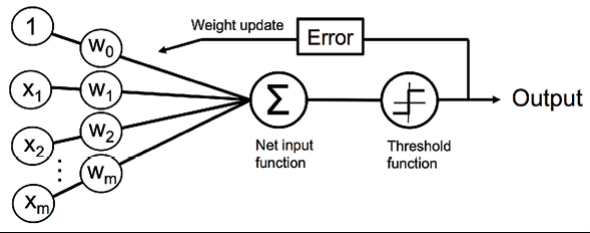
Obr. 2.3 Skoková aktivační funkce vytvářející predikce

**2.3 Optimalizace váhových koeficientů**

Cílem nejen perceptronu, ale i všech neuronových sítí je optimalizovat váhové koeficienty pro co nejlepší výsledky. Vektor těchto koeficientů je na začátku generován zcela náhodně v rámci určeného rozmezí (například 0 až 1). Při prvních iteracích během učení tedy není příliš veliká pravděpodobnost, že perceptron klasifikuje data správně. Algoritmus zkrátka nejdříve vůbec neví, co dělá, a tak prostě tipuje. Aby dokázal váhové koeficienty optimalizovat, je mu pro porovnání poskytnut vektor cílových tříd výsledků .

Tento přístup nazýváme *učení s učitelem* (z anglického *supervised learning*).2 Formálně můžeme aktualizaci každého váhového koeficientu zapsat jako:

Pro výpočet si nejdříve musíme stanovit *rychlost učení (learning rate)* **η.** Tu obvykle určíme jako konstantu s hodnotou mezi 0 až 1. Aktualizaci každého váhového koeficientu (neboli proces, který nazýváme jako *trénování* nebo *učení*) poté definujeme jako násobek rychlosti učení , závislého příznaku a rozdílu mezi cílovou () a predikovanou ( třídou:

Během klasifikace každého příkladu z trénovacího vzorku takto postupně aktualizujeme každý váhový koeficient. Perceptron se obvykle trénuje na celkovém datovém souboru několikrát. Pro počet trénovacích iterací se používá název *epochy.2* Pořadí příkladů ve vzorku je typicky určeno náhodně pro každou epochu, protože algoritmus by si jinak mohl „zapamatovat“ který příklad koresponduje s jakým výstupem a nebyl by schopen pracovat s novými daty – tento problém nazýváme *přetrénování (overfitting)*.1 Schéma percetronu je dobře vidět na obrázku 2.4.

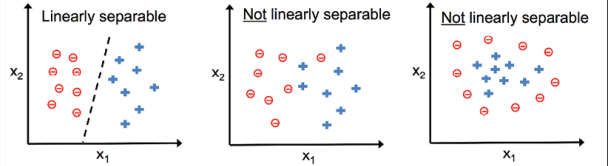
Obr. 2.4 Schéma perceptronového algoritmu

**2.4 Souhrn**

Cílem perceptronu je aplikovat jisté učební pravidlo a s jeho pomocí optimalizovat váhové koeficienty tak, aby dosahoval co nejlepších výsledků při klasifikaci vstupních dat. To je proces, kterému říkáme *učení* (odsud pojem *strojové/hluboké učení*) nebo *trénování*. Ideálně se snažíme o to, aby model dosahoval dobrých výsledků nejen na trénovací množině, ale aby dokázal pracovat efektivně také s novými daty (k validaci slouží *testovací datový soubor* – viz implementace). Celý proces učení můžeme shrnout těmito kroky:

1. Náhodná inicializace vektoru váhových koeficientů
2. Určení prahu à převod prahu na bias
3. Iterace přes každý příklad trénovací datové množiny a počet epoch
   1. Výpočet lineárního vstupu a výstupu skokové funkce
   2. Aktualizace váhových koeficientů na základě učebního pravidla

Po dovršení počtu epoch nebo dosažení určené přesnosti (pokud je skutečně dosažitelná) je perceptronový model naučený řešit problém binární klasifikace a může být aplikován na nové příklady stejného typu mimo trénovací množinu. Je však důležité, aby byla poskytnutá data lineárně separovatelná (úplné oddělení dat v ruznách výstupních třídách lineárním útvarem, například přímkou ve 2D případě – viz obr. 2.5), jinak by je perceptron nebyl schopen klasifikovat.



Obr. 2.5 Příklady lineárně rozdělitelných a nerozdělitelných datových bodů

**2.5 Implementace**

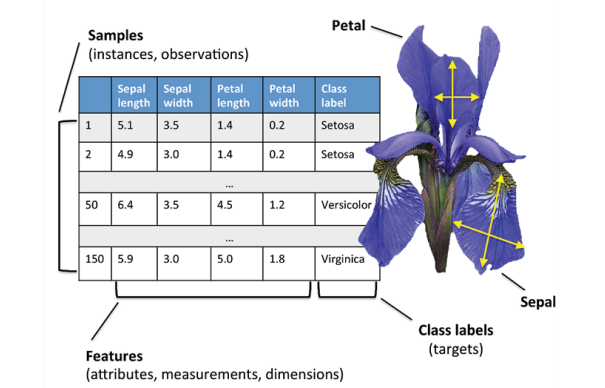
Implementace perceptronu v Pythonu je velmi jednoduchá, jelikož se jedná pouze o jeden neuron. Kompletní kód je k dispozici v příloze, zde představím pouze klíčové metody. Ty jsou součástí třídy *Perceptron()*:



Kód 1 - Perceptron.py

O trénování klasifikátoru se stará metoda *fit(X, y),* která jako parametry vyžaduje příznaky příkladů v trénovacím datovém souboru **X** a jejich cílové třídy **Y** pro porovnání. Na základě výše definovaného učebního pravidla optimalizuje váhové koeficienty.

Na otestování perceptronu využiji datový soubor Iris (viz obr. 2.6), který obsahuje míry okvětních lístků tří druhů kosatce – setosa, versicolor a virginica (každý druh je zastoupen 50 příklady – soubor dohromady obsahuje 150 příkladů).



Obr. 2.6 Popsaná reprezentace datového souboru Iris

Celý soubor můžeme zapsat jako matici , kde každý řádek *i* reprezentuje jeden příklad a každý sloupec *j* představuje daný příznak :

Cílové třídy příkladů zase zapíšeme jako vektor :

Jelikož perceptron je binární klasifikátor, budu pracovat pouze se druhy setosa a versicolor, takže trénovací datový soubor bude obsahovat pouze 100 příkladů (stejně tak vektor cílových tříd bude mít velikost )1:

Následující kód načte datový soubor Iris a extrahuje z něj prvních 100 příkladů. Vzorkům označeným jako versicolo*r* bude ve vektoru cílových tříd přiřazena hodnota 1, těm označeným jako setosa zase -1:

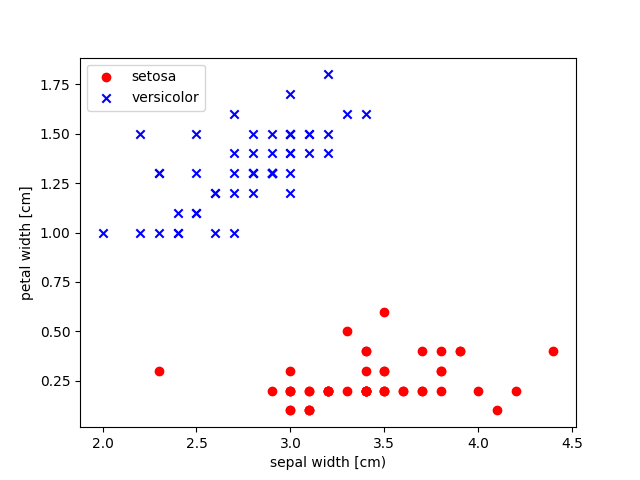
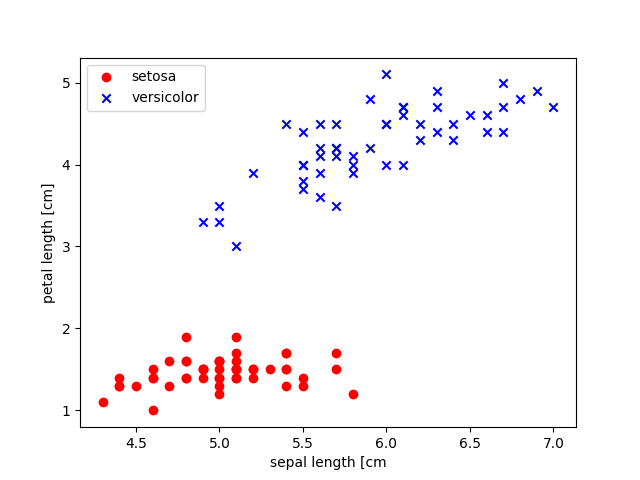


Kód 2 – perceptron\_iris\_binary.py

Datový soubor bude dále rozdělen na 80 trénovacích vzorků (40 od každého druhu) a 20 pro testování natrénovaného modelu (10 od každého druhu):



Kód 2 – perceptron\_iris\_binary.py

Z následujících grafů (viz obr. 2.7 a 2.8) můžeme jednoznačně určit, že data z datového souboru Iris jsou dobře lineárně rozdělitelná (podle délek a šířek lístků korun a kalichů obou druhů), takže perceptron by neměl mít s klasifikací žádné problémy:

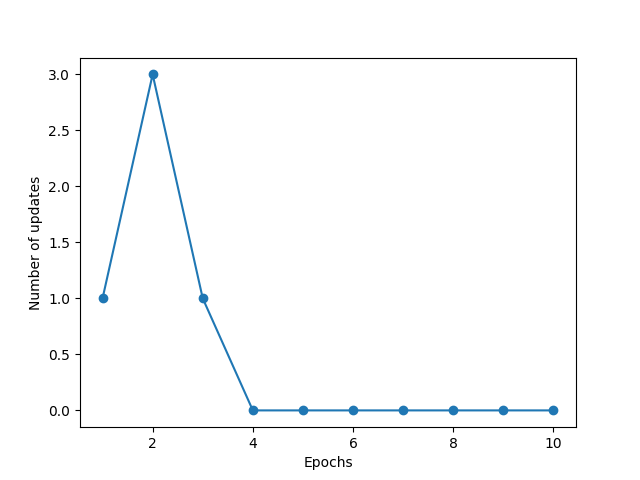
Obr. 2.8 Rozdělení jednotlivých druhů podle délky okvětních lístků

Obr. 2.7 Rozdělení jednotlivých druhů podle šířky okvětních lístků

Nakonec importujeme výše uvedenou třidu *Perceptron()* a předáme trénovací množinu dat metodě *fit(X, y).* Kvůli jednoduššímu 2D zobrazení grafů pracuji pouze se dvěma příznaky, a to sice s délkami okvětních lístků koruny a kalichu.Rychlost učení jsem zvolil o velikosti 0,1 a počet epoch nastavil na 10:



Kód 2 – perceptron\_iris\_binary.py

Jak ukazuje tento graf (viz obr. 2.9), perceptronový algoritmus dosáhl cíle a přestal optimalizovat váhové koeficienty ve čtvrté epoše:

Obr. 2.9 Graf znázorňující počet aktualizací vah v každé epoše

Natrénovaný model poté správně klasifikoval všech 10 příkladů z testovacího souboru.

Při trénování neuronových sítí obecně je již zcela běžnou praxí rozdělit datový soubor

na trénovací a testovací množinu2 (stejně jako jsem to v tomto příkladu udělal já). Získáme tak mnohem lepší představu o skutečné přesnosti natrénovaného modelu (v tomto případě 100 %), protože při špatně zvolených parametrech může jednoduše dojít k již zmíněnému přetrénovaní.