En kort introduktion til Maskinlæring; Partikelfysik — Er det en elektron?

STEMAcademyML

November 2025

I fysik vil vi ofte gerne bruge eksisterende data til at forudsige ting. Hvis vi har rigtig meget forskelligt data, kan det være svært for os som mennesker at overskue. Her kan vi bruge maskinlæring og kunstig intelligens til at hjælpe os. Det kan ske både inden for klassifikation af himmellegemer, gletcherstørrelser og partikler i partikelfysik, som er det vi vil kigge på her. Dataen som detektoren giver har nogle måske kryptiske navne, men det betyder ikke noget. Vi behøver ikke forstå hvad de betyder, for at få en Machine Learning algoritme til at lære af det, og fortælle os hvilke variable der er vigtige.

1 Introduktion

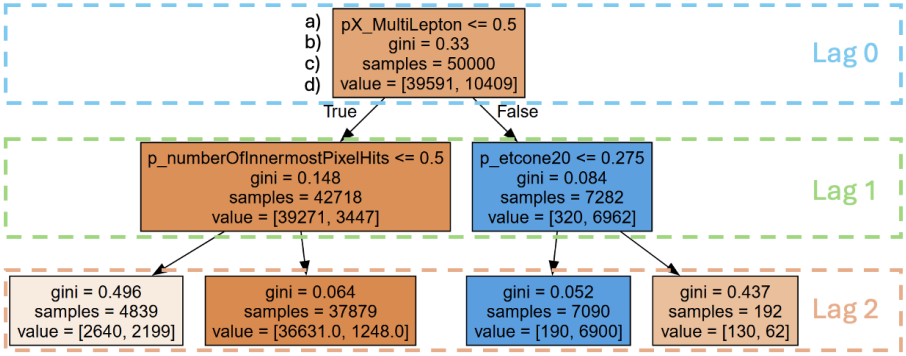
Du har været så heldig at komme i studiepraktik, og har fået lov til at komme med på CERN. Her skal du være med til at samle noget data på LCH (The Large Hadron Collider) og du har fået det fornemme ansvar at skulle vurdere om partiklerne der strømmer gennem detektoren er elektroner eller ej. Den data der kommer ud består dog af en masse navne og variable som du ikke kender, og der er tusindvis af datapunkter. Hvordan skal du dog nogensinde fortolke dette for slet ikke tale om at udnytte det til at forudsige om data stammer fra en elektron eller en af de mange andre mystiske partikler som flyver gennem detektoren hvert sekund? Hvis bare der var en måde at udnytte computerens kraftfulde regnekraft til at takle problemet: Det er her Machine Learning træder ind!

Opgaven du har fået på CERN består i at betragte data der kommer igennem detektoren for de enkelte partikler, notere tendenser i hvordan data opfører sig og forudsige på baggrund af dette om den givne partikel er en elektron. Med så mange forskellige variable og datapunkter ville dette dog være umuligt at gøre i hånden, men heldigvis har vi vores computer til hjælp.

En problematik som denne hvor man skal svarer på et ja-nej spørgsmål, elektron eller ej, kaldes for **Classification**, så det er det vi skal arbejde med her for at løse vores opgave fra CERN. Man kan også bruge Machine Learning til **Regression**, dvs. forudsige et tal. Det kunne for eksempel være at vi skulle forudsige hastigheden eller energien af en partikel.

2 Decision Trees

Et **Decision Tree** er bygget op af lag og grene. Ved hver gren stiller den et spørgsmål, og bevæger sig ned i det næste lag baseret på om spørgsmålet er sandt eller falsk. Og ved at lære af en masse data, kan den finde ud af hvilke spørgsmål der er bedst at stille.



Figur 1: Et decision tree med to lag. I hver boks ser vi: a) hvilket spørgsmål der stilles, b) gini koefficienten, der beskriver hvor 'blandet' dataen i boksen er, c) hvor mange partikler eller hvor meget data der er i gruppen, d) Hvor mange partikler der er i hver kategori [ikke elektron, elektron].

Hvis algoritmen ikke får lov til at stille nogen spørgsmål, kan den kun gætte på en ting, nemlig det der er mest af. I det første lag er der flest ikke-elektroner, så gættet ville være at der ingen elektroner er, hvilket som sådan ville være rigtigt i 80% af tilfældene, da ca. 1/5 partikler er elektroner.

Giver vi den lov til ét lag, kan den stille ét spørgsmål og dele data i to grupper. I vores tilfælde stiller algoritmen spørgsmålet: "Er pX_MultiLepton mindre end eller lig med 0.5?". Nu har vi to grupper med hver et bestemt antal elektroner og ikke-elektroner. Alle partikler i kassen til venstre ville blive vurderet som 'ikke elektroner' og dem til højre ville blive vurderet som 'elektroner'. I det her tilfælde vil der for eksempel være 320 ikke elektroner der bliver classificeret som 'elektroner'. Det er det vi kalder **false positive**. På samme måde vil der også være 3447 elektroner som bliver klassificeret som 'ikke-elektroner, kaldet **false negative**. Alt efter situationen kan det være bedst at fokusere op at minimere false negative eller false positive.

Med to lag kan træet igen stille et nyt spørgsmål til de to grupper. Det her behøver ikke nødvendigvis at være det samme for begge. Igen vil gættet for hvert gruppe være den type partikel der er flest af. Jo flere elektroner (ikke-elektroner) vi har i samme gruppe i forhold til antallet af ikke-elektroner (elektroner) jo bedre - og jo lavere er Ginikoefficienten.

2.1 Optimering: Boosted decision tree (BDT)

Data kan være meget komplekst, med mange variable og stor variation i værdier. Det gør det svært at lave ét træ, der rammer godt. Man skal fx bestemme hvor dybt træet må være, hvor mange partikler der højest/mindst må være i en kasse, og hvor store fejl der er acceptable. På fagsprog kaldes det hyperparametre.

I stedet for at satse på ét perfekt træ, kan man lade algoritmen bygge mange små træer, hvor hvert nyt træ fokuserer på de fejl, de tidligere træer lavede. På den måde forbedres modellen trin for trin. Denne metode kaldes **Boosted Decision Trees**, og den giver typisk meget mere præcise forudsigelser end ét enkelt træ.

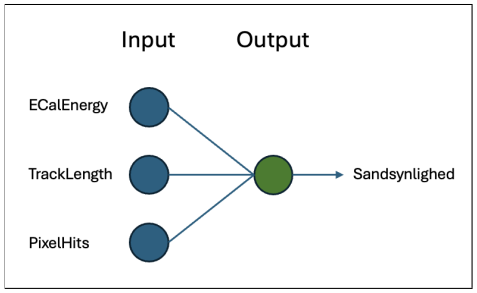
3 Neurale Netværk

En anden type algoritme, som man ofte hører om, er **Neurale Netværk (NN)**. Navnet kommer fra, at opbygningen minder om måden neuroner i hjernen sender signaler på. Ligesom et decision tree består af lag, har NN også lag af noget vi kalder **noder**. I stedet for ja/nej-spørgsmål, fungerer de som justerbare knapper. Hver node har en **vægt**, der bestemmer, hvor meget et input betyder for resultatet.

3.1 Simpelt NN

Lad os tage et helt simpelt netværk, med kun 1 node. Som nævnt i indledningen, vil vi gerne skrive en funktion, der kan fortælle os om en partikel er en elektron eller ej, og det kan et NN hjælpe os med. Hvis vi for eksempel har 3 variable, bliver der i noden lavet ligningen:

$$a \cdot \text{ECalEnergy} + b \cdot \text{TrackLength} + c \cdot \text{PixelHits} + d = \text{sandsynlighed for elektron.} \quad (1)$$



Figur 2: Et simpelt neuralt netværk, hvor vi har 3 input og en enkelt node. I noden laves ligning (1), hvor hver input variabel ganges med en vægt og lægges sammen plus en bias. Outputtet fra noden bruges til at bestemme en sandsynlighed for om partiklen er en elektron.

Her kombineres information fra forskellige målinger i detektoren — for eksempel hvor meget energi partiklen har afsat i elektromagnetiske kalorimeter (*ECalEnergy*), hvor langt et spor den har efterladt i detektoren (*TrackLength*), og hvor mange pixel-hits den har givet i sensoren (*PixelHits*) [1].

[1] her er variablenavnene simplificeret, men ofte vil de hedde noget som p_etcone20, p_deltaEta1 og p_numberOfInnermostPixelHits

Men ligningen her kan give os mange værdier, ikke kun mellem 0 og 1. Så vi skal konvertere de tal vi får ud, til en sandsynlighed. Det gør vi med en **activation function**.

For at få en sandsynlighed som output, vil vi ofte bruge en activation function kaldet **Sigmoid**. Den kan tage en hvilken som helst værdi, og omsætte det til en værdi mellem 0 og 1. Hvis vi kalder vores ligning (1) for g, vil den endelige ligning være

$$\frac{1}{1+e^{-g}} = \text{sandsynlighed}. \tag{2}$$

og vi får et tal mellem 0 og 1 ud. Jo større g er, jo tættere på 1 vil sigmoid funktionen være.

NN arbejder bedst med værdier mellem 0 og 1, så derfor er det vigtigt at dataen først bliver normaliseret. Det vil sige hvis vi havde data med værdier fra 50 til 150, ville alle værdier blive divideret med 150, så den største værdi ville være 1, og alle andre ville være mindre end 1.

Hvis vi tog et netværk med kun 1 node og vi havde 3 variable kunne ligningen være:

$$0.9 \cdot 0.5 + 0.4 \cdot 0.1 + 0.2 \cdot 0.8 + 0.1 = 0.75 \tag{3}$$

$$\frac{1}{1+e^{-0.76}} = 0.68. \tag{4}$$

Vægtene kan fortælle os noget vigtigt, også selvom vi ikke er sikre på hvad variablene er, nemlig deres vigtighed. Her er a relativt høj og c lille, hvilket betyder, at *ECalEnergy* bidrager meget til beslutningen, mens *PixelHits* bidrager mindre. Og når vi bruger vores sigmoid, får vi en sandsynlighed på 68 %. Hvis vi sagde at alle med over 50 % chance ville klassificeres som elektron, ville denne partikel altså ende i den kategori.

3.2 NN med flere lag

Det er en god start, men, ofte vil vi se at der er mange ting at tage højde for. For eksempel kan nogle variable have større betydning for hurtige partikler end for langsomme, eller nogle målinger kan kun være informative, hvis partiklen bevæger sig i en bestemt retning gennem detektoren. Et neuralt netværk med flere lag kan kombinere disse typer information og lære mere komplekse mønstre, som er svære at udtrykke med en enkelt ligning.

Det er her vi introducerer det der kaldes "**Hidden layers**". Et hidden layer er et lag af noder, der ligger mellem input og output (se figur 3). Hver node i "hidden layer", laver samme ligning som (1), med de input den får, og sender så et tal videre til det næste lag. Men inden et tal sendes videre til det næste lag, bruger vi igen en **activation function**, men her har activation function et andet formål. I outputtet er funktionen at omdanne det endelige tal til en sandsynlighed. Men i hidden layers, har funktionen det formål at aktivere forskellige noder.

Her vil man ofte bruge den aktiveringsfunktion der hedder **ReLU** (Rectified Linear Unit):

$$f(x) = max(0, x) \tag{5}$$

Hvis ligning (1) giver en negativ værdi, vil f(x) sende 0 videre til næste lag, mens positive værdier sendes videre som de er. Det betyder, at nogle noder "slukker", og andre "tænder", afhængigt af dataen. På den måde kan forskellige noder lære at reagere på forskellige typer af partikler. For eksempel partikler, der bevæger sig hurtigt, og partikler, der efterlader et langt spor i detektoren. Et lag i netværket kan derfor lære, at visse signaler (som mange pixel-hits) kun er vigtige for bestemte typer partikler, mens andre signaler (som energi i kalorimeteret) spiller en større rolle for andre typer.

3.3 Opbygning af NN

Et neuralt netværk kan for eksempel se sådan ud som i figur 3.

Her har vi 3 input parametre. Herefter to hidden layers, hvor ligninger som (1) og (2) bliver udført med en ReLU activation function, og til sidst et output lag, der giver den endelige sandsynlighed med en sigmoid activation function.

Hvis et hidden layer har n input (noder i det foregående lag) og m noder i sit eget lag, har det

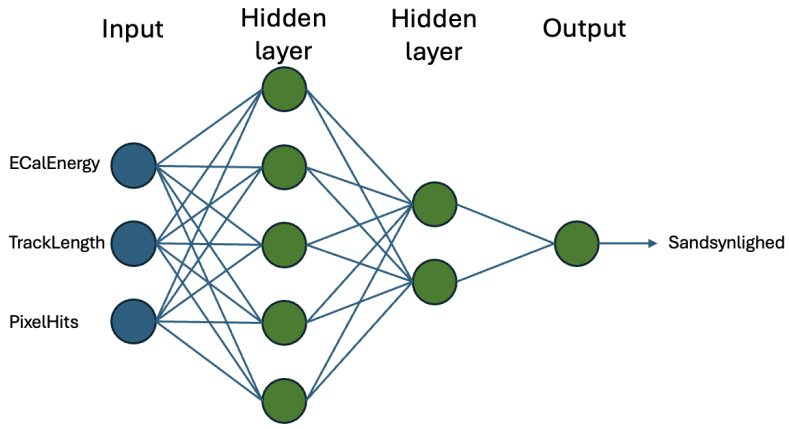
$$n \times m + m$$

parametre.

Så for første skjulte lag har vi $3 \times 5 + 5 = 20$ parametre, for andet skjulte lag $5 \times 2 + 2 = 12$, og for output laget $2 \times 1 + 1 = 3$. I alt altså 35 parametre. Vi kan også sige, at noderne svarer til antallet af **bias**, mens stregerne imellem svarer til antallet af **vægte**.

Til sidst justerer algoritmen alle vægte og biaser, så forskellen mellem den forudsagte sandsynlighed for, at en partikel er en elektron, og den faktiske sandhed bliver så lille som muligt. Det betyder, at modellen hele tiden prøver at blive bedre til at ramme de rigtige svar.

På den måde lærer et neuralt netværk selv at finde mønstre i data, der kan være så komplekse, at vi aldrig ville kunne beskrive dem med en enkelt formel.



Figur 3: Et eksempel på et neuralt netværk med 3 input parametre og to skjulte lag og et output. I hver node laves ligning (1). I noderne i "Hidden Layer"bruger vi typisk ReLU activation function før resultatet sendes videre til næste node, og i den sidste output node bruger vi typisk Sigmoid activation function for at lave det om til en sandsynlighed.

4 Hvordan ved algoritmen hvilket og hvor mange spørgsmål den skal stille?

En algoritme skal bruge noget til at måle, hvor godt den gør det, så den ved hvilke spørgsmål den skal stille, eller hvilke vægte den skal tildele de forskellige parametre. I Machine Learning kalder vi det en **Loss function**.

For et boosted decision tree prøver algoritmen mange mulige spørgsmål og beregner, hvor meget hvert spørgsmål kan reducere loss function. Det spørgsmål, der giver det mindste bidrag, bliver valgt. Processen gentages i hver ny gren, indtil træet er "dybt nok"eller ikke længere forbedres. Bemærk, for et simpelt decision tree bruger man den ovennævnte ginikoefficient.

For et neuralt netværk fungerer princippet på samme måde, men i stedet for at vælge spørgsmål prøver algoritmen forskellige vægtkombinationer i de mange noder. Den beregner herefter sin loss (fejl), og justerer vægtene lidt ad gangen, så fejlen bliver mindre for hver iteration. Denne proces kaldes træning, og den metode, der bruges til at ændre vægtene, kaldes **gradient descent**.

Den hyppigst brugte loss function for classification er **Log Loss** (eller Binary Cross Entropy):

$$\mathcal{L}(\hat{y}, y) = -1\frac{1}{N}\sum_{n=1}^{N}\left[y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right] \tag{6}$$

hvor y som kan være 0 og 1 er den sande værdi og $\hat{y} \in (0, 1)$ er den forudsagte værdi. Vores loss function straffer altså hårdere jo sikrere modellen er i et forkert gæt.

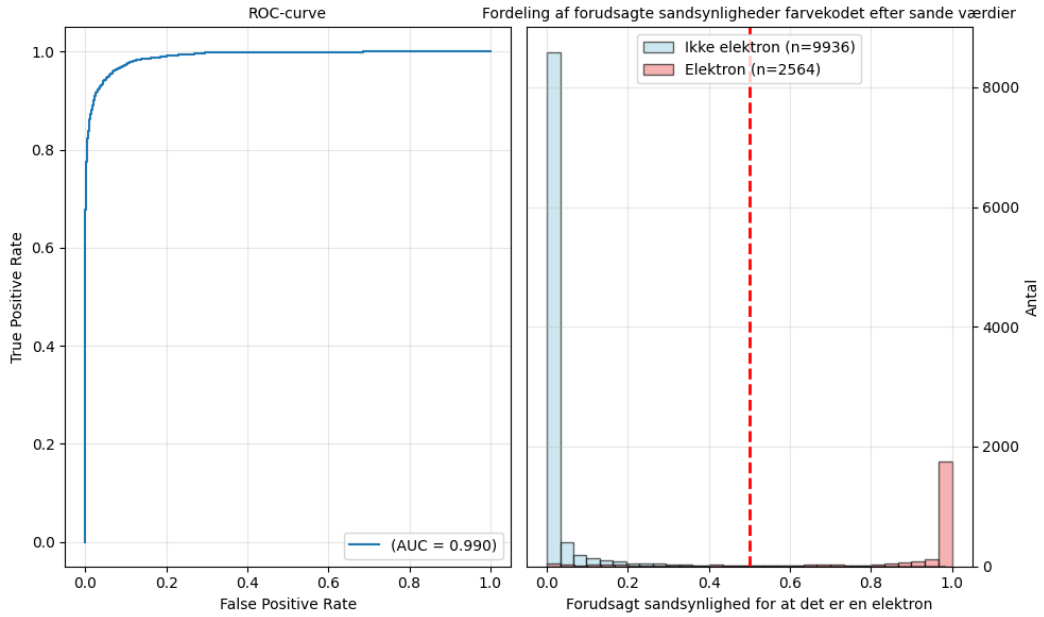
4.1 Hvordan kan vi se vi hvor godt den klarer sig?

Når nu vi har fået opbygget vores decision tree eller neurale netværk, vil vi gerne finde ud af hvor god den er til at forudsige om en partikel er en elektron eller ej. Det gør vi ved at give modellen ny data, som den ikke har trænet på, og bede den om at forudsige, og så sammenligne med de rigtige værdier. Typisk deler man datasættet i to dele: en del til **træning**, hvor modellen får lov at lære

sammenhængene, og en mindre del til **test**, som gemmes væk, indtil modellen skal vurderes. På den måde sikrer vi, at modellen ikke bare husker de præcise svar, men faktisk har lært et mønster, der også virker på ny data.

For classification vil man typisk visualisere performance med en **ROC kurve** (Receiver Operator Characteristic), som vist til venstre i figur 4. Den viser forholdet mellem false positive rate og true positive rate. Vi vil gerne have denne til at ligge så tæt på det venstre øverste hjørne som muligt. **AUC-scoren** (Area Under Curve) i hjørnet skal være så tæt på 1 som muligt. En AUC på 0.5 ville svare til at gætte tilfældigt.

Til højre er **sandsynlighedesfordelingen** plottet. Hvis modellen er god, ligger ikke-elektronerne mest omkring lave sandsynligheder (tæt på 0), og elektronerne ligger mest omkring høje sandsynligheder (tæt på 1). Jo mere de to fordelinger er adskilt, jo nemmere er det at skelne mellem elektron og ikke elektron. Alt til højre for beslutningsgrænsen bliver klassificeret som 'elektron, og alt til venstre som 'ikke-elektron. Hvis fordelingerne er godt adskilt, betyder det, at vi kan vælge en grænse, der laver meget få fejl — altså få elektroner, der bliver classificeret som ikke elektroner og omvendt.



Figur 4: Til venstre ses en ROC-kurve for en classification model. Til højre ses fordelingerne hvad modellen har gættet farvekodet efter deres sande værdier.

5 Er alle informationer lige vigtige?

Når vi har trænet vores model, er det interessant at finde ud af, hvilke variable den lægger mest vægt på. I et **decision tree** kan vi ofte allerede se det ud fra de spørgsmål, træet stiller først, for eksempel at pX_MultiLepton og p_etcone20 har stor betydning for sandsynligheden for at gætte elektron. I et **neuralt netværk** er det sværere at se direkte, men her kan vi kigge på, hvor meget de enkelte vægte påvirker resultatet.

En mere generel metode, der virker for næsten alle typer modeller, kaldes **Permutation Importance**. Her tager man den trænede model og "roder" med én variabel ad gangen. Det vil sige, man blander dens værdier tilfældigt, så den mister sin sammenhæng med resten af dataen. Hvis modellens præcision falder meget, betyder det, at den variabel var vigtig for forudsigelsen. Hvis vi fx bytter rundt på værdierne af pX_MultiLepton og modellens præcision falder markant, betyder det, at denne variable er vigtig. Hvis der næsten ingen forskel er, betyder det, at modellen ikke brugte den variabel særlig meget.

Det fantastiske ved den her metode er, at vi ikke behøver at vide noget som helst om systemet på forhånd. Modellen finder helt selv ud af, hvilke ting der betyder mest, og det kan være mønstre, som folk normalt kun opdager efter mange års erfaring. Pludselig kan vi se, hvad der faktisk bestemmer om partiklen er en elektron, eller hvad der gør forskellen mellem to typer af data, uden at have nogen forhåndsviden. Det er her, Machine Learning virkelig viser sin styrke.