Sukkertoppen Gymnasium

next UDDANNELSE KØBENHAVN

Studieområdeprojekt

Elev: Syed Muhammad Amin *Klasse: 3.n*

Fag og niveau	Vejleder:
Programmering B & Matematik A	Henrik van Bedaf Sterner & Niels Henning Dahl

Emne: Klassifikation vha. Machine Learning
Opgaveformulering: • Redegør for brugen af neurale netværk på en selvvalgt case. • Analyser hvorledes et neuralt netværk fungerer hvor der er særligt fokus på de matematiske aspekter. Det gælder bl.a.: Lineær algebra herunder operationer på matricer og vektorer, funktionsanalyse af funktioner med flere variable herunder partielt afledede, kædereglen og gradient-metoden. • Analyser med udgangspunkt i principperne fra computationel tænkning hvorledes man kan designe og konstruere en simpel prototype på et neuralt netværk i et selvvalgt programmeringssprog. • Vurder hvor godt dit neurale netværk fungerer på din selvvalgte case og overvej de to fags muligheder og begrænsninger i forhold til at undersøge problemet. Evt. vedlagt bilag:

Studieområdeprojektet

“Hvordan kan jeg vha. en softmax klassifikationsmodel klassificér billeder.”

Klasse: 3.N
Navn: Syed Muhammad Amin
Fag og Niveau: Programmering B & Matematik A
Gymnasium: NEXT Sukkertoppen

Afleveringsdato: 19/12/2024

RESUME

Dette projekt omhandler klassifikation af billeder. Interessen bag projektet ligger i emnet om lineær neurale netværker, og hvordan de fungerer. Ydermere, gik projektet ud på at undersøge, hvordan en lineær neural netværk kan klassificér billeder. Mere specifikt, en softmax classifier. Idét, at jeg tog udgangspunkt i MNIST datasættet. Ved at redegøre for hvordan en softmax classifier er opbygget, kunne jeg konstruere en selv. Dernæst analyserede jeg dens output, ved at plotte en graf. Grafen viste dens læringskurve over tid. Idét at den var hurtige til at tilpasse sit data. Den kunne dog ikke genkende håndskrevet tal på papir. Derfor blev det konkluderede, at modellen var ikke god til at genkende billeder uden for MNIST's scope. Til gengæld er problemformuleringen, om hvordan man kan klassificér billeder vha. blevet løst. Dette skyldes, at de følgende ting er blevet opnået under projektet: En softmax classifiers bestanddele og processer, hvordan man konstruerer en softmax classifier vha. programmering, og hvordan man kan teste den. Idét at man bruger den til at klassificere billeder.

2 INDHOLDSFORTEGNELSE

3 INDLEDNING

Gennem tiden, har folk opbygget statistisk og sandsynlighedsmodeller, for at kunne forudsige noget om et stykke data. Videnskabsmænd har også prøvet at definere ordet information, og dens betydning indenfor IT og matematik. Hvad nu med billeder? Emnet om at studere billeder, såsom seismisk aktivitet, eller forfalskede underskrifter findes som jobs i virkeligheden. Med den nye revolutionerende AI front, kan man få det bedste af begge verdener. Vha. Maskinlæringsteknikker og principper, kan man undersøge specifikke detaljer som billedet gemmer fra øjet.

Maskinlæring er et ret bredt emne, der dækker over flere underemner. Et af dem er Deep Learning. Idét er der komplekse neurale netværker, såsom ANN's, CNN's, RNN's osv. En neural netværk, er et program, eller model, der laver beslutninger lignende til menneskets hjerne. Ved at efterligne måden biologiske neuroner fungerer, og identificere fænomener, og evt. vægte muligheder, og lave konklusioner (What is a neural network?, n.d.).

Givet, at opgavens omfang kun er 20 normalsider, Har jeg valgt at undersøge lineær neurale netværker. Lineær neurale netværker er forsimplede designs af de senere, mere moderne og komplekse netværker. Mere specifikt, er de modeller der kun gør brug af lineære funktioner til at transformér deres input. Jeg ville arbejde med at klassificér tal fra MNIST datasættet vha. en lineær neurale netværk. MNIST, som er meget velkendt, er et datasæt bestående af håndskrevet tal. Idét at den består af 70,000 forskellige gråskaleret $28px \times 28px$ billeder. MNIST er oftest brugt som en basis for at teste billede bearbejdelses systemer (MNIST Dataset, n.d.).

På baggrund af det, har jeg valgt at arbejde med en specifikt lineær neural netværk, som er, softmax classifieren. Givet, at det er en meget interesserende model at undersøge, givet hvor akkurat den er. For at kunne undersøge emnet, og få en grundlæggende forståelse af den, har jeg kommet frem til min problemformulering: *"Hvordan kan jeg vha. en softmax klassifikationsmodel klassificér billeder."*

4 EMPIRI OG METODE

For at forstå lineære neurale netværker, kræver det en del matematisk viden. I dette indgår der, sandsynlighed- og statisklære, lineær algebra og calculus. Online kan man finde flere kilder inkluderende kurser, som universiteter, såsom Stanford, tilbyder. I denne opgave ville jeg bruge de følgende kilder.

Dive into Deep Learning af Aston Zhang, Zachary C. Lipton, Mu Li og Alexander J. Smola
Jeg ville forholde mig til denne bog, som en primær kilde. Dvs., at de fleste koncepter, som jeg ville inddrage, og en del af teorien ville stamme fra denne bog. Til den redegørende del af mit projekt, ville jeg introducere nogle notationer og formler, som bogen bruger til at redegøre for lineær neurale netværk.

Machine Learning Introduction af Stanford - Andrew Ng
For at få noget mere information om emnet neurale netværk, ville jeg forholde mig til denne kilde. Kilden er god til at forklare komplekse aspekter og en del af matematikken. Jeg ville bruge denne kilde som supplerende stof til mit emne om klassificering vha. softmax regression.

Selvskrevet kode + graf over training-loss, validation-loss og -accuracy. Af Syed M. Amin
Til den analyserende del, ville jeg på baggrund af teorien om Softmax regression og klassifikation, konstruér et program, der ville kunne genkende tal baseret på MNIST datasættet. Programmet ville bl.a. printe dens accuracy, training- og validation-loss. Ude fra dette data, ville jeg plotte en graf, der så viser modellens accuracy, training-loss og validation-loss baseret på MNIST datasættet.

4.1 UNDERSPØRGSMÅL TIL DE TAKSONOMISKE NIVEAUER

For at kunne løse problemformuleringen, kommer jeg til at dele problemet op i mindre problemer. Hermed ved at stille 1-3 spørgsmål til hvert taksonomiske niveau, ville jeg kunne dele og herske problemet som helhed. Dvs. at jeg bruger løsningerne fra de opdelte problemer til at løse problemformuleringen.

4.1.1 Redegørende spørgsmål

- Redegør for vektorer, matricer, skalarer, og andre elementer fra lineær algebra der bruges i lineær neurale netværker.
- Redegør for brugen af statistik og sandsynlighedsteori i en softmax classifier.
- Redegør for brugen af softmax regression klassifikation i billedklassifikation.

4.1.2 Analyserende spørgsmål

- Analyser hvorledes et neuralt netværk fungerer, hvor der er særlig fokus på de matematiske aspekter
- Analyser med udgangspunkt i principperne fra komputationel tænkning hvorledes man kan designe og konstruere en simpel prototype på et neuralt netværk i et selvvalgt programmeringssprog.

4.1.3 Diskuterende spørgsmål

- Vurder, hvor godt dit neurale netværk fungerer på din selvvalgte case.
- Overveje de to fags muligheder og begrænsninger i forhold til at undersøge problemet.

5 HVAD ER EN SOFTMAX CLASSIFIER?

Machine Learning er en bred kategori, der her alt det at gøre med en computer program, der kan imitere den måde et menneske lærer (What is a neural network?, n.d.). I Machine Learning findes der 4 kategorier. Disse er, supervised learning, unsupervised learning, reinforcement learning og deep learning.

Supervised learning, er når man skaber en model til at lave forudsigelser på baggrund af noget data. Dataet, som modellen trænes på opdeles i 2 underkategorier. Disse er features og labels. Features er nogle parametre, som datapunkterne har. Dette kunne f.eks. være grundarealet af et hus, eller en elevs alder. Labels er det empiri, der knytter sig til dataets features. Her kunne man have en helt dataset over elevernes højder baseret på alder i 3.N, eller en dataset med priserne af huse i Valby baseret på grundareal. Her har en Supervised maskinlæringsprogram til formål at finde mønsteret i dette data, for at kunne lave nogle forudsigelser. Her kunne det være værd at vide, hvad prisen af en lejlighed i Valby ville være, hvis den har et grundareal på $82\ m^2$. Eller hvor høj den næste elev kommer til at være, hvis han er 21 år gammel.

En softmax classifier er en supervised maskinlæringsalgoritme, der på baggrund af nogle billeder og labels, kan forudsige et billedes kategori. De fleste Machine Learning algoritmer baserer sig på de grundlæggende koncepter om tensorregning og sandsynlighedslære. I den første del, ville jeg gennemgå de grundlæggende aspekter af lineær algebra, som bidrager til neurale netværk som en helhed. Dette indgår datatyper og senere hen forskellige slags operationer.

5.1 Skalarer, Vektorer, og Matricer.

En tensor er et flerdimensionelt dataobjekt, der indeholder numeriske værdier. Programmer såsom PyTorch, Tensorflow, Jax osv. Bruger disse programmer til at lave deres udregninger. Der findes 4 kategorier af tensors. Der er tensors der kun indeholder 1 numerisk værdi. Disse kaldes for skalarer. Skalarer repræsenteres med et lille kursivt bogstav, f.eks. x. Der er én dimensionale (dvs. 1D) tensors, som kaldes for vektorer. Vektorer er det der kommer til at være en af de essentielle brikker indenfor Maskinlæring. Man bruger det følgende matematiske notation til at betegne en vektor '$\mathbf{x}$'.

Indenfor lineær algebra, findes der to slags vektorer. Rækkevektorer, som stammer fra det engelske fagterm row vector, og kolonnevektorer, som stammer fra fagbegrebet column vector. De ser ud som følgende:

$$\mathbf{x} = [x_1, x_2, x_3, x_4 \ldots . x_n]$$

Figur 4.1.1, en kolonnevektor

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

Figur 4.1.2, en rækkevektor.

For at vise deres størrelse, bruger man den følgende matematiske notation; $\mathbf{x} \in \mathbb{R}^n$. Her påstår man, at vektor $\mathbf{x}$ er n elementer lang. Ordet element henviser sig til en numeriske værdi, der er i en tensor. Dermed ville man sige, at en tensor har n-antal elementer i sig. n indikerer også hvor lang vektoren er. Dvs. $||\mathbf{x}|| = n$. For at referere til et element i vektor $\mathbf{x}$, skriver man så x_2. Her referer man til det anden element i rækkefølgen (se figur 4.1.1).

Der er 2D tensors, som kaldes for matricer. Matricer kan indeholde data ligesom en tabel. Det smarte ved at bruge matricer er, at man kan beregne på flere forskellige datapunkter under et variabel. Den matematiske notation for en matrice er '$\mathbf{X}$'. En matrice har flere rækker og kolonner (se figur 4.1.3)

24	98	52	12
2	1	98	67
3	6	9	6
4	7	9	14

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{2m} & \ldots & b_{mn} \end{bmatrix}$$

Figur 4.1.3, sammenligning mellem en tabel og en matrice.

For at beskrive en matrices dimensionalitet, dvs. vise dens størrelse i rækker og kolonner, bruger man den følgende notation $\mathbf{B} \in \mathbb{R}^{m \times n}$. Her siger man, at matricen $\mathbf{B}$ har m rækker og n kolonner (se Figur 4.1.3).

Der findes tensors, der kan være i flere dimensioner. Det kunne f.eks. være 10, 12 eller måske 100 forskellige dimensioner. Givet, at tensors, der er over 2 dimensioner ikke er med i softmax klassifikation[1], ville jeg ikke gå i dybden med det.

5.2 OPERATIONER

Prikproduktet, også kaldt skalarproduktet, er summen af produktet mellem vektorernes elementer. Dvs., at givet vektor $\mathbf{a}$ og $\mathbf{b}$ med størrelsen n; $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, ville prikproduktet være som vist i nedenunder.

$$\mathbf{a^T b} = \sum_{i=1}^{n} a_i b_i$$

Operatøren $\mathbf{T}$ indikerer to ting. 1) at der er tale om et prikprodukt af to vektorer, og 2) der er tale om en *transpose*. En transpose, svarer til det omvendte af en vektor eller matrice. Dvs. at man bytter rundt på mængden af rækker og kolonner. Som f.eks., man har en matrice med følgende dimensioner $\mathbf{B} \in \mathbb{R}^{m \times n}$. Ved at tage dens transpose får man så $\mathbf{B^T} \in \mathbb{R}^{n \times m}$.

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 1 \end{bmatrix}^{\mathbf{T}} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 1 \end{bmatrix}$$

Når man arbejder med Machine Learning, ville man oftest støde på matrice-vektor multiplikationer og matrice-vektor multiplikationer. Dvs. at man ganger en matrice med en anden matrice, eller med en vektor. Begge to er meget vigtige fundamenter indenfor maskinlæring. Givet en matrice $\mathbf{A}$ og en vektor $\mathbf{x}$, ville det returnere en rækkevektor, hvor hvert element svarer til prikproduktet mellem $\mathbf{x}$ og hvert element i matricen $\mathbf{A}$ (Zhang, Lipton, Li, & Smola, 2.3 Linear Algebra, n.d.). For at få bedre indblik, kan man visualisér det som det følgende:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \cdot a_{11} + \cdots + x_n a_{1n} \\ x_2 \cdot a_{21} + \cdots + x_n a_{2n} \\ \vdots \\ x_m \cdot a_{m1} + \cdots + x_n a_{mn} \end{bmatrix}$$

For at gør det simplere, kan man så omskrive det til prikproduktet af en kolonnevektor og $\mathbf{X}$.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \mathbf{a_1^T x} \\ \mathbf{a_2^T x} \\ \vdots \\ \mathbf{a_m^T x} \end{bmatrix}$$

[1] I kontekst med lineær neurale netværk.

Her er der krav om at $\mathbf{x}$ skal være på længde med n. Dette skyldes, at man tager prikproduktet mellem kolonnevektoren $\mathbf{a_i} \in \mathbb{R}^n$, hvor i er et indeks der beskriver hvilken række der er tale om.

Matrice-matrice multiplikation bygger på det samme koncept. Her tager man en rækkevektor og en kolonnevektor og finder deres prikprodukt. F.eks., har man en række $\mathbf{a_i}$, fra matrice $\mathbf{A}$, og en kolonne $\mathbf{b_j}$ fra $\mathbf{B}$. Dem ganger vi sammen, imens i og j iterer igennem deres respektive matricer. Dette resulterer i en ny matrice $\mathbf{C}$.

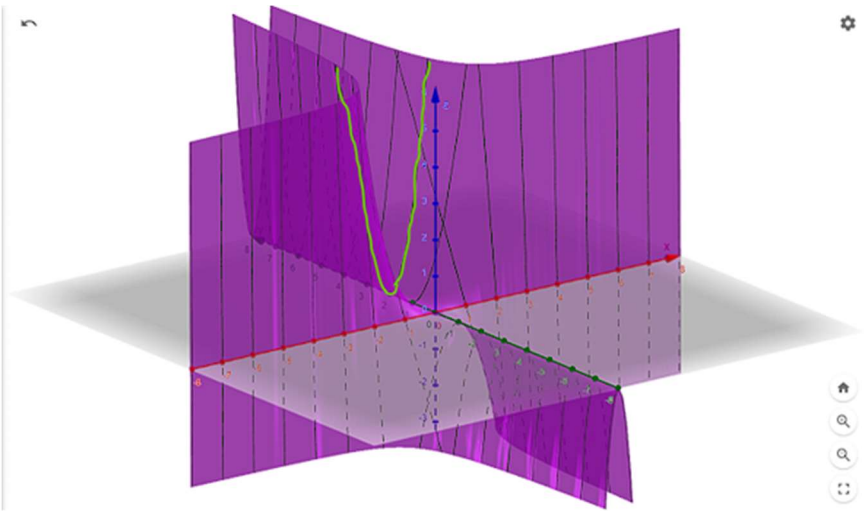
$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} \mathbf{a_1} & \mathbf{a_2} & \mathbf{a_3} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{b_1} \\ \mathbf{b_2} \\ \mathbf{b_3} \end{bmatrix} = \begin{bmatrix} \mathbf{a_1^T b_1} & \mathbf{a_1^T b_2} & \mathbf{a_1^T b_3} \\ \mathbf{a_2^T b_1} & \mathbf{a_2^T b_2} & \mathbf{a_2^T b_3} \\ \mathbf{a_3^T b_1} & \mathbf{a_3^T b_2} & \mathbf{a_3^T b_3} \end{bmatrix}$$

5.3 PARTIEL AFLEDTE FUNKTIONER OG GRADIENTEN.

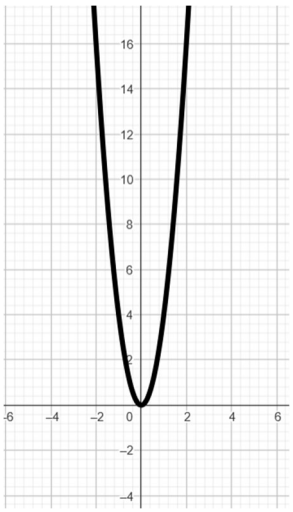
Et meget vigtigt koncept indenfor, som er gradient descent, underbygger sig af disse 3 koncepter. I forbindelse med gradient descent, er der meget fokus på funktioner med flere parametre og evt. 3D grafer.

Partialer bygger op på konceptet om afledte funktioner, men her tager den højde for tredimensionale funktioner. I det scenarie har f flere parametre, f.eks. $f(x, y) = 2xy^2$. Hvis vi bare tager den afledte funktion af f, ender vi med at ignorere en af parametrene. Dette ville ikke være særlige smart. Derfor bruger vi så partialer.

Partialen af f er hældningen af en given funktion f, med hensyn til en given parameter, f.eks. x. Her tager man et udsnit af grafen, hvor x er et givent tal, som f.eks. $x = 2$. Her differentierer man så parameteren y. Partialens operatør betegnes med ∂.



Figur 4.3.1, 3D graf, som viser f(x, y)



Figur 4.3.2, 2D udsnit af f(x, y), hvor $x = 2$

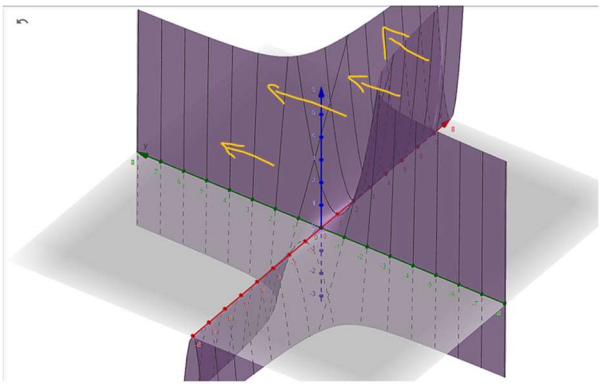
Hvis vi ville finde hældningen af vores funktion f, med en given x-værdi, ville det se ud som følgende:

$$\frac{\partial}{\partial_y} f(2, y) = \frac{\partial}{\partial_y} (2 \cdot (2) \cdot y^2) = (4 \cdot y^2)' = 8y$$

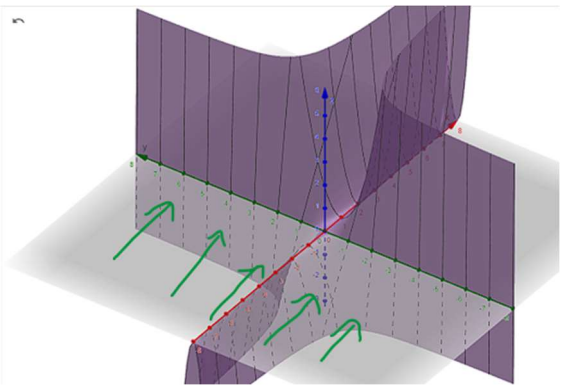
Her har vi fundet partialen af $f(x, y)$, med respekt til y, hvor $x = 2$. Hvis vi ville finde hældningen for alle punkter på y-aksen, gør man så det følgende.

$$\frac{\partial}{\partial_y} f(x,y) = \frac{\partial}{\partial_y}(2x \cdot y^2) = 2x \cdot (y^2)' = 2x \cdot 2y = 4xy$$

I stedet for at sætte x en værdi, kan man finde et udtryk, der beskriver hældningen på alle punkter på y-aksen. Dette kan godt være en svært koncept at forstå, men for at gør tingene simplere, kan man tænke på det som sådan: Hvis man tager partialen af f med respekt til y kigger man på hældningen fra y's retning.



Figur 4.3.3, hældning med respekt til x's retning



Figur 4.3.4, hældning med respekt til y's retning

Hvis man tager partialen af f med hensyn til x, læse man hældningen fra x's retning. Definitionen for partialen af f med hensyn til x, er som følgende (Partial Derivative):

$$\frac{\partial}{\partial_x} = \lim_{h \to 0} \frac{f(x+h,y) - f(x,y)}{h}$$

Partialen giver os kun lov til at se på hældningen af vores funktion f, en parameter ad gangen. Derfor har vi så gradienten. Gradienten viser den stejleste vej op ad grafen afhængige af dens parametre. Den beskrives som en 2D-vektor, der viser en punktforskydning. Afhængige af ens position, kan den vise den stejleste vej opad. Givet at vi kommer til at arbejde med funktioner, med flere parameter, kan vi referere til parametrene vha. en multidimensionale vektor.

$$\mathbf{p} = (p_1, p_2, \dots, p_n)$$

Hermed er gradienten skrevet som det følgende:

$$\nabla_{\mathbf{p}} f(\mathbf{p}) = [\partial_{p_1} f(\mathbf{p}), \dots, \partial_{p_n} f(\mathbf{p})]$$

Så hvis vi har en funktion $f(x,y)$, ville dens gradient være:

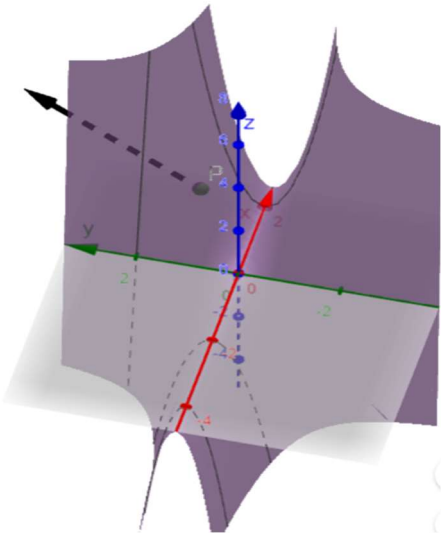
$$\nabla_{x,y} f(x,y) = [\partial_x f(x,y), \partial_y f(x,y)]$$

For at holde os til det originale eksempel, svarer det til at gradienten er:

$$\nabla_{x,y} f(x,y) = [2y^2, 4xy]$$

Afhængige af et tilfældigt punkt på grafen P, som ligger i en given funktion f, kan vi finde dens gradient. Som f.eks., $P = [1,1,f(1,1)]$, ville gradienten være det følgende.

$$[2 \cdot 1^2, 4 \cdot 1 \cdot 1] = [2,1]$$



Figur 4.3.5, gradienten mht. punktet P.

Givet at gradienten er en 2D-vektor, er den så begrænsede til xy-planen. Dvs. at den ikke har en z-akse. Hvis man ville differentiere en funktion med flere parametre, dvs. $f'(x,y)$, skal man benytte kædereglen. For funktioner med 1 parameter, er kædereglen som følgende.

$$(fog)' = f'\big(g(x)\big) \cdot g'(x)$$

Kædereglen for funktioner med flere parametre, er som følgende.

$$\frac{d}{dt} f\big(x(t), y(t)\big) = \frac{\partial f}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt}$$

Kædereglen kan også bruges, for at finde en indre funktions parametre. Dvs. hvis vi har en funktion ind i en funktion $f(g(x), y)$, hvor $f(z,y) = 2z + y$, og $g(x) = x^2 + 9x$, kan vi så finde partialen af f med hensyn til en af $g's$ parametre, som f.eks. x.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial x}$$

5.4 SANDSYNLIGHEDSREGNING

Når det kommer til Maskinlæring er uklarhed en af de vigtigste aspekter. I supervised learning prøver vi at forudsige noget ukendt på baggrund af den givne information (Zhang, Lipton, Li, & Smola, 2.6 Probability and statistics, n.d.). Det kunne f.eks. være prisen af et hus, givet at vi kender dens grundareal. Vi ville også gerne kvantificere vores uvidenhed, som f.eks., ville vi gerne vide hvad chancen for at en patient for en hjerteanfald baseret på hans/huns oplysninger.

Sandsynlighed er et felt indenfor matematik, der har til formål at gør uvidenheden klart. Som f.eks., har man to almindelige 6-sidet terninger. Hvis vi kaster dem, har vi ingen idé hvad vi får. Ved at lave en lille tabel over summerne, kan man så konkludere, at der er en høj sandsynlighed, at man ruller 7. Dette skyldes, at de fleste summer leder op til et 7-tal.

	1	**2**	**3**	**4**	**5**	**6**
1	2	3	4	5	6	7
2	3	4	5	6	7	8

3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Tabel 4.4.1

For at beskrive disse tilfældigheder, bruger man oftest tilfældige variabler. Tilfældige variabler, er variabler, der repræsenterer ægte handlinger i en matematisk form. Der findes som regel to slags tilfældige variabler: Diskrete tilfældige variabler og Kontinuerte tilfældige variabler.

Diskrete tilfældige variabler, er variabler, der er begrænset til separate værdier (Khan, u.d.). Dvs., at hvert enkelte element af variablen er forskellige fra den anden. Man bruger den følgende notation for at betegne en tilfældige variabel $\boldsymbol{B}$. F.eks., hvis vi har en tilfældige variabel $\boldsymbol{X}$, som kun returnerer 1 eller 0, helt afhængige af resultatet af en mønt, der bliver slået. Her kunne 0 være plat, 1 være krone.

$$\boldsymbol{X} = \begin{cases} 0\ lander\ på\ plat \\ 1\ lander\ på\ krone \end{cases}$$

En kontinuert tilfældige variabel, er en variabel, der kan have hvilken som helst værdi i et givent interval. F.eks., hvis man har en tilfældige variabel $\boldsymbol{Y}$, der beskriver en tilfældig persons IQ, ville det se ud som følgende.

$$\boldsymbol{Y} = IQ\ af\ en\ tilfældig\ person\ i\ København$$

$$\boldsymbol{Y} \in]55; 200[$$

Intervallet viser den laveste IQ 55 og den højeste IQ 200[2]. Det der er specielt ved kontinuerte værdier, er at der kan være uendelig mange IQ niveauer. Dette gør, at man ikke kan tælle hvor mange der er, givet at det er et interval. Hvis man har en diskrete variabel, kan man tælle de forskellige udfald, som variablen repræsenterer. Dette kan man ikke gør her. Dette skyldes, at den tilfældige variabel $\boldsymbol{Y}$, kan antage hvilken som helst værdi i intervallet (Khan, u.d.).

5.4.1 Sandsynlighedsfunktioner

Sandsynligheden for, at en begivenhed b, ville ske, kan skrives op som det følgende $P(\boldsymbol{B} = b)$. Som f.eks., $P(\boldsymbol{X} = plat)$, viser sandsynligheden for at mønten lander på plat. Dette kaldes for en marginal sandsynlighedsfunktion. Alle sandsynlighedsfunktioner returnerer værdier inden for intervallet $[0; 1]$, hvor 0 er mindre sandsynligt og 1 er mest sandsynligt. Dvs., $P(\boldsymbol{X} = x) \in [0; 1]$. Hvis man har flere probabiliteter i en givet sample-space[3], ville summen af alle disse sandsynligheder give 1 (Das, n.d.). Som f.eks., hvis $P(\boldsymbol{A} = 1)$, $\boldsymbol{A} =$ $En\ terning\ som\ bliver\ slået$, hvad er sandsynligheden for at terningen lander på 1? Sandsynligheden giver naturligvis $\frac{1}{6}$. Men det gør det også, for alle andre udfald, såsom 2, 3 osv. Ved at summe alle disse sandsynligheder får vi 1. Dette skrives som det følgende:

[2] Intervallet er ikke akkurat på nogen måde, men er hellere skabt for at fremme forståelsen af pointen bag kontinuerte tilfældige variabler.
[3] Et statistisk miljø med flere begivenheder I sig.

$$1 = \sum_{i \in \boldsymbol{A}} P(\boldsymbol{A} = i)$$

Betinget sandsynlighedsfunktioner, viser sandsynligheden af en begivenhed baseret på en anden begivenhed. De skrives som det følgende.

$$P(\boldsymbol{A}|\boldsymbol{B})$$

Her er et eksempel. Givet at man har et kortsæt, med 52 kort. 26 af dem er røde, og resten er sort. Sandsynligheden for, at man trækker en rød 4 er som vist nedenunder:

$$P(4|rødt) = \frac{2}{26} = 7{,}6\%$$

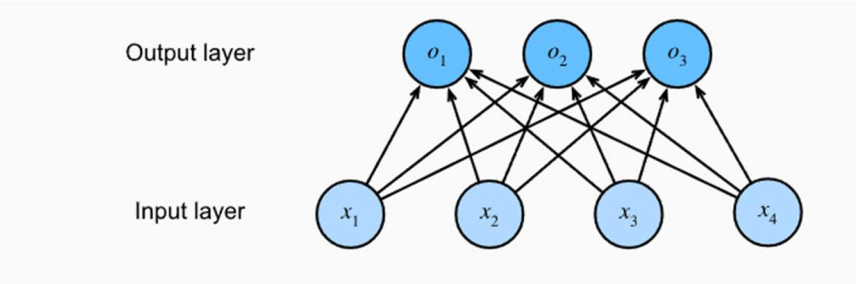
5.5 Intuitionen

I et klassifikationsproblem, ville vi gerne klassificér et datapunkt på baggrund af nogle kategorier. Når modellen laver sin udregninger, får man en liste over sandsynligheder. Her er den højeste sandsynlighed, som beskrives som $\max_k \hat{\mathbf{y}}$, den største match for vores datapunkt.

Tage et eksempel med et billede af en bil. Her har man 4 kategorier; cykel, bil og fly. De noteres som følgende;

$$y \in \{(1,0,0), (0,1,0), (0,0,1)\}$$

Her bruger man ikke normale tal, men en one-hot encodning, for at vise at der ikke er en rækkefølge. Her er (1,0,0) en cykel, (0,1,0) = bil, osv. En Softmax klassifikationsmodel er opbygget som vist i figur 4.5.1. Input lageren, som kan beskrives som en vektor $\mathbf{x}$, indeholder mængden af værdier, som objektet, der skal klassificeres indeholder.



Figur 4.5.1, klassifikationsmodel for 4 kategorier.

5.6 Weights og Bias

Som figur 4.5.1 viser, er hvert inputelement i netværket forbundet til en output element. Elementerne i outputlageren kaldes for en logit. En logit er et skalar, der indeholder rå data fra den forrige lag. Idét at den ikke er blevet omdannet til en sandsynlighed (Bowen, n.d.).

Når data bliver sendt til outputlaget, foregår der 2 operationer. En af dem er en lineær funktion, som tilføjer vægten og bias til inputtet. Det repræsenteres med den følgende formel.

$$o_n = \left(\sum_{i=0}^{k} x_i w_{in} \right) + b_n$$

Her er et tilfældigt logit o_n lige med summen af input dataet ganget med vægterne og evt. en tilføjet bias. Som modellen viser, er vægterne og biasen forskelligt pr. forbindelse mellem input og outputknude. Dvs., at hvis o_1 er forbundet til x_1 og x_2, ville vægtene og biasen være forskellige mellem o_1 til x_1 og o_1 til x_2. Dette kan omskrives til det følgende.

$$o_n = \mathbf{x}^{\mathbf{T}}\mathbf{w_n} + b_n$$

Her er vektoren $\mathbf{x}$, vores features, og $\mathbf{w_n}$, en vektor fra matricen $\mathbf{W}$, som bliver taget med i udregningen. Denne proces kaldes for vektorisation. Her omskriver man en formel, eller ligning for at yde komputationel effektivitet[4] (Zhang, Lipton, Li, & Smola, 4.1 Softmax Regression, n.d.). For at vektorisér den yderligere, kan vi tage hensyn til alle outputelementerne, ved at skrive det følgende.

$$\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Her tager man højde for alle outputknuderne, ved at inddrage alle inputknuderne og deres vægte + bias. Her implementerer man strukturen af en lineær funktion. Lægge mærke til, at vægten er en matrice $\mathbf{W}$. Givet, at der kan være flere forskellige forbindelser, ville det være optimalt at sætte alle vægtene i en matrice.

Det der gør vægt og bias specielt, er den måde den styrer vigtigheden af forbindelsen. Givet at vægterne og biasen starter med tilfældige værdier. Kan de så ændres for at tilpasse/fitte deres data. Ved at ændre på vægtene og bias, sørger vi for, at vores model kan skelne mellem underbukser og sko.

5.7 SOFTMAX REGRESSION

Det vi får i outputlaget, kan ikke rigtige bruges til noget endnu, givet at de ikke er blevet omdannet til sandsynligheder. Dette ville ikke kunne bruges til at klassificér et billede. Tricket her, er at kunne begrænse disse værdier mellem $[0; 1]$, og dermed omdanne dem til en sandsynlighed. For at gøre det, skal vi have en funktion, der kan tage alle vores logitter, og lave dem om til noget der kan bruges. Den bedste idé er at tage eksponenten af en værdi, dvs. e^{o_i} $eller$ $\exp(o_i)$, og dividér den med summen af eksponenterne af alle logitter. Idét, at den ville være proportionelle med en sandsynlighed $P(y = i|\mathbf{x}) \propto \exp(o_i)$. Dette betyder, at sandsynligheden for at i er en kategori indenfor kategorierne y, givet vores features x.

I retur får vi en sandsynlighed. Dette kaldes for softmax regression, og skrives op som det følgende.

$$\hat{y}_i = softmax(\mathbf{o})_i = \frac{\exp(o_i)}{\sum_{k=0}^{||\mathbf{o}||} \exp(o_k)}$$

Vi laver en ny vektor $\hat{y}_i$, som er de forudset værdier. Disse værdier kan vi bruge til at se om vores input tilhører en specifikke klasse. Idét at den højeste sandsynlighed, viser at inputtet tilhører den givne kategori.

[4] Hurtigere udregninger

5.8 Overraskelse, Entropi og Cross-Entropy Loss Funktionen

For at måle, hvor akkurat en model er, benytter man sig af en loss-funktion. Loss funktionen har til formål at måle forskellen på en models labels $\mathbf{y}$, og forudsete værdier $\hat{\mathbf{y}}$. Cross-Entropy eller tværs entropi stammer fra en gren af datalogi, der hedder informationsteori.

Informationsteori drejer sig om at måle, hvor brugbart information data indeholder. Det hele starter med *overraskelse*. Overraskelse er et mål for, hvor overrasket man ville være på baggrund af noget information. Sige at man slår en mønt. 999 gange lander den på krone. Hvis nu man slår møntet igen, og den lander på plat. Ville man ikke være overrasket?

Overraskelse beskrives som at være invers proportionel med sandsynlighed. Dvs., sandsynligheden for at mønten lander på krone er $P(\boldsymbol{X} = krone) = \frac{999}{1000} = 99{,}9\%$. I dette tilfælde ville overraskelsesniveauet være ret lavt. Dette skyldes, at det var forventet i forvejen. Men sandsynligheden for plat, som er $P(\boldsymbol{X} = plat) = \frac{1}{1000} = 0{,}1\%$, ville give et højt overraskelsesniveau. Deres forhold kan beskrives som det følgende.

$$s(\boldsymbol{X} = plat) \propto \frac{1}{P(\boldsymbol{X} = plat)}$$

Her er s[5], som stammer fra den engelske ord $surprise$, vores overraskelsesniveau. Overraskelsesniveauet svarer til den negative logaritme af $P(x)$. Dvs.

$$s(\boldsymbol{X} = plat) = \log_2 \frac{1}{P(\boldsymbol{X} = plat)} = -\log_2 P(\boldsymbol{X} = plat)$$

Logaritmen er en base 2 logaritme. Dette skyldes, at der kun er 2 tilfælde; plat eller krone. Hvis der var flere tilfælde, ville logaritmen være $\log_b P(x)$, hvor b er mængden af udfald. Som f.eks., en terning slag $\log_6 P(\boldsymbol{X} = x)\,, \boldsymbol{X} = En\ terning\ bliver\ slået.$

Entropien er den gennemsnitlig overraskelse. Lad os antage at $P(plat) = 0.2$, og $P(krone) = 0.8$. Ved at udregne overraskelsesniveauerne efter 100 kast, får vi det følgende.

$$P(plat) \cdot s(plat) \cdot 100 + P(krone) + s(krone) \cdot 100 = 72.19$$

Dette giver så en stor overraskelse, at efter 100 kast, hvor 20 af dem var plat og 80 krone. Den gennemsnitlig overraskelse, eller *entropien,* ville så være $\frac{72.18}{100} \approx 0.7218$. Entropien er et mål for, hvor uforudsigeligt dataet er. Ved at dividere med 100, ville entropien rent faktisk kunne opskrives som sådan.

$$P(plat) \cdot s(plat) + P(krone) \cdot s(krone) = 0.7218$$

På baggrund af dette, kan man opskrive en formel for entropi, som ser ud som det følgende.

[5] Notation Hentet fra *Technical notes on information theory* af Yang Xu.

$$H(\boldsymbol{X}) = \sum_{x \in \boldsymbol{X}} P(x)s(x) = \sum_{x \in \boldsymbol{X}} -P(x)\log_b P(x)$$

Cross-entropy, eller tværsentropien er et mål for, hvor uforudsigeligt dataet er sammenlignet med en anden distribution[6] Q. Formlen opskrives som det følgende.

$$H(P,Q) = -\sum_j P(j)\log\left(Q_j\right)$$

j i dette tilfælde, er så datapunkterne fra både $\mathbf{y}$ og $\hat{\mathbf{y}}$. Vi kan bruge Cross-Entropy til at måle modellens accuracy. Dette skyldes, at vores labels $\boldsymbol{y}$ forudset værdier $\hat{\mathbf{y}}$ består af sandsynligheder, og ville derfor virke perfekt med loss-funktionen.

$$l(\mathbf{y},\hat{\boldsymbol{y}}) = -\sum_{\mathrm{j}} y_j \log \hat{y}_j$$

5.9 Minibatch Stochastic Gradient Descent (SGD)

Gradient descent benytter sig af loss-funktionen til at formindske parametrene $\mathbf{W}$ og $\mathbf{b}$. Gradient descent, har til formål at finde den negative gradient af en funktion. Ved at finde den negative gradient af vores loss-funktion, har vi fundet en succesfuld måde at minimere dens parametre $\mathbf{W}$ og $\mathbf{b}$. Husk, at $\hat{y}_j$ stammer fra $softmax(\mathbf{o})_j = softmax(\mathbf{W}\mathbf{x} + \mathbf{b})_j$.

Vi kan udnytte forståelsen af l's partial hældning til at få en bedre forståelse af gradient descent funktionen. Partialen af loss-funktionen med hensyn til en logit[7] o_j er det følgende.

$$\frac{\partial}{\partial o_j} l(\mathbf{y},\hat{\boldsymbol{y}}) = softmax(\mathbf{o})_j - y_j$$

Lægge mærke til, at hældningen er rent faktisk forskellen mellem $\hat{y}_i$ og y_j. Dette gør det nemmere at udregne gradienten (Zhang, Lipton, Li, & Smola, 4.1 Softmax Regression, n.d.). Vi kan lave en ny loss-funktion. Den kalder vi L, og den defineres som det følgende.

$$L^{(i)}(\mathbf{W},\mathbf{b}) = softmax(\mathbf{W}\mathbf{x} + \mathbf{b})_i - y_i$$

Nedenunder er formlen for Minibatch Stochastic Gradient Descent implementerede på lineær regression. Ved lineær regression, måler man også forskellen mellem punkterne. Idét at man tager den kvadreret gennemsnitstab (Ng, Coursera.org, n.d.).

$$l^{(i)}(\mathbf{w},b) = \frac{1}{2}(\hat{y}_i - y_i), \qquad \hat{y}_i = \mathbf{w}^{\mathbf{T}}\mathbf{x}^{(\mathrm{i})} + b_i$$

$$(\mathbf{w},b) - \frac{\eta}{|B|}\sum_{i \in B} \partial_{(\mathbf{w},\mathbf{b})} l^{(i)}(\mathbf{w},b) \rightarrow (\mathbf{w},b)$$

Lægge mærke til, at den også tager højde for forskellen mellem $\hat{y}_i$ og $\hat{y}_i$. Med vores opdaterede loss funktion, kan man indsætte den i formlen. Dermed har man et udtryk for Minibatch SGD mht. softmax klassifikation. Idét, at man tager hensyn til vægtmatricen $\mathbf{W}$ og biasen $\mathbf{b}$.

[6] Diskret eller kontinuert variabel
[7] En Outputværdi

$$(\mathbf{W}, \mathbf{b}) - \frac{\eta}{|B|} \sum_{i \in B} \partial_{(\mathbf{W},\mathbf{b})} L^{(i)}(\mathbf{W}, \mathbf{b}) \rightarrow (\mathbf{W}, \mathbf{b})$$

Ved at opdele vores træningsdataset i flere del, og træne modellen pr. batch B([8]), kan vi opnå en meget hurtigere, og mere effektiv måde at minimere vægtene og biasen. En batch, er et datasegment, der indeholder en specifik mængde af observationer (Zhang, Lipton, Li, & Smola, 3.1 Linear Regression, n.d.). I et eksempel med billedklassifikation, kunne det indeholde 64 billeder og labels. Dette afhænger af, hvor stor man sætter batch størrelsen til.

Læringsraten η, svarer til skridtlængden af vores minibatch SGD. Skridtlængden svarer til hvor stort et skridt man ville tage ned af bakken, eller i dette scenarie, ned til funktionens lokalt minimum. Hvis skridtlængden er lille, ville man garanteret ende i funktionens minimum, men det ville tage ret langt tid. Hvis skridtlængden er stor, ville gradient descent ramme over, og evt. aldrig nå frem til funktionens lokalt minimum (Ng, Learning Rate, n.d.).

6 HVORDAN KONSTRUERER MAN EN SOFTMAX CLASSIFIER?

På baggrund af teorien, kommer jeg til at konstruere en lineær neuralt netværk, som kan klassificér tal. Her gør jeg brug af MNIST's datasæt, som består af 60.000 billeder af håndskrevne tal. Hvert billede har en resolution på 28×28 pixels. Dette svarer til 784 pixels. Givet at vi ville tage højde for hver pixel, ville vi have en model, der har 784 inputs. Jeg kommer til at bruge Python, da det er perfekt til datavidenskab og maskinlæringsprogrammer. Her ville jeg analysere, hvordan modellen lærer igennem sin træning. Dette gør jeg ved at tage dens output, og lav en graf over den, som jeg analyserer. Dernæst ville jeg selv teste modellen, ved at se hvor godt den kan genkende håndskrevet tal.

Til at starte med skal vi designe en model. Modellen, skal kunne tage 784 inputs, dvs. én for hver pixel. Idét, skal den også anvende en lineær funktion vha. forward propagation. Forward propagation, er processen, hvor inputværdierne bliver sendt videre til den næste lag, og evt. blive transformeret vha. vægtene og biasen (What is forward propagation in neural networks?, n.d.). Givet, at vi kun har 2 lag; input og output, svarer det til at udregne vores outputlag $\mathbf{o}$.

$$\mathbf{x} \in \mathbb{R}^{784}$$

$$\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Dette kan gøres ved at opstille en klasse, der hedder SoftmaxClassifier. Klassen ville være programmets hovedmodel, som ville kunne klassificere billeder på baggrund af dens input $\mathbf{x}$. Den ville dermed transformere den til outputvektoren $\mathbf{o}$.

[8] Et sæt med features og labels.

```
13  class SoftmaxClassifier(torch.nn.Module):
14      def __init__(self, n_inputs, n_outputs):
15          super().__init__()
16          self.linear = torch.nn.Linear(n_inputs, n_outputs)
17          self.flatten = torch.nn.Flatten()
18
19      def forward(self, x):
20          x = self.flatten(x)
21          return self.linear(x)
```

Figur 5.1, SoftmaxClassifier.

Ved linje 16, opstiller man forbindelsen, som vi kalder for et lineær neural netværk. Ved linje 17, opstiller man et objekt af klassen Flatten(). Det den har til formål at omdanne inputtet til en vektor, så den kan udfylde udsagnet $\mathbf{x} \in \mathbb{R}^{1\times 784}$. Funktionen $forward()$, sørger for, at når inputværdierne skal udregnes til outputværdier, ville det ske forward propagation.

Ved linjer 36 og 37 definerer vi vores loss-funktion og gradient descent metode. Givet at der findes flere gradient descent metoder, har vi valgt den der er mest simpel (når det kommer til forklaring). Loss-funktionen $CrossEntropyLoss()$, indeholder både softmax regression og cross-entropy funktionen.

```
36  lossfunc = torch.nn.CrossEntropyLoss()
37  optim = torch.optim.SGD(model.parameters(), lr=LR)
```

Figur 5.2, opsætning af cross-entropy loss og minibatch SGD.

Givet at $CrossEntropyLoss()$, gør begge ting, kunne man opstille formlen for loss-funktionen som det følgende.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{\mathrm{j}} y_j \log \hat{y}_j$$

$$\Leftrightarrow l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{\mathrm{j}} y_j \cdot \log \frac{\exp(o_J)}{\sum_{k=1}^{q} \exp(o_k)}$$

$$\Leftrightarrow l(\mathbf{y}, \hat{\mathbf{y}}) = \log \sum_{k=1}^{q} \exp\,(o_k) - \sum_{j=1}^{q} y_j o_j$$

Givet at man så kan indsætte labels $\mathbf{y}$ og predictions $\hat{\mathbf{y}}$, får man mængden af tab/loss efter outputværdierne $\mathbf{o}$, er blevet konverteret til sandsynligheder, som vist i nedenstående udtryk.

$$softmax : \mathbb{R}^d \rightarrow [0; 1]$$

Når det kommer til optræningen af en maskinlæringsmodel, er der 2 dele. Træningsdelen, som består af korrigering af $\mathbf{W}$ og $\mathbf{b}$. Dernæst er valideringsfasen. Her evaluerer man modellen baseret på mængden af fejl og accuracy/korrekthed. Eventuelt, kan man have en testfase til sidst, hvor man overordnet evaluerer modellens fejl og korrekthed. Givet at vi arbejder med

PyTorch, skal vi selv implementere modellens trænefunktion. Vi kan udarbejde funktionen baseret på noget pseudokode.

```
start
create classifier class as model
setup GPU

lossfunc = CrossEntropyLoss()
optim = optim.SGD(model.parameters())

load MNIST_DATASET into train_set AND test_set
train_size = 0.8 * len(train_set)
val_size = len(train_set) - train_size

train_subset, val_subset = split train_set according to sizes

create dataloaders from subsets with batch_size{64}:
    train_loader from train_subset
    val_loader   from val_subset

    # optional
    test_loader  from test_subset

model_fit(epochs):
    for epoch in epochs:
        for images and labels in train_loader:
            output = predict(images)
            loss = lossfunc(output, labels)

            backpropogate loss
            optim.step()
            store loss

        for images and labels in val_loader:
            output = predict(images)
            calculate val_loss <- lossfunc(output, labels)
            measure accuracy
            store val_loss and accuracy


save model
plot loss val_loss and accuracy

end
```

Figur 5.3, selvskrevet pseudokode fra bilaget.

Først skal man loade nogle dataloaders. Dataloaders er objekter, der bruges til at træne en model. Vores største fokus er $model\_train(epochs)$. For hver batch, udregner den vores forudsigelser $\hat{\mathbf{y}}$, hvor den dernæst udregner loss vha. loss-funktionen. For at minimere mængden af loss, bruger vi så backpropogation. Backpropogation i maskinlæring, er en

metode, der udregner den negative gradient[9] for vores batch (What is a backpropogation algorithm, n.d.). Ved at kalde $optim.step()$, bliver det så indført. ***store loss*** og ***store val_loss and accuracy*** er variabler, der bliver gemt i en liste. Disse bruges senere til at plotte modellens fremgang.

Nedenunder er implementeringen af pseudokodens første del, træningsdelen. For at holde overblik over træningssessionen, bliver training_loss, printet hver epoch. Ved at få vores forudsigelser `outputs`, og eventuelt udregne dens loss. Kan vi så opdatere vores parametre **W** og **b.** Dette gøres vha. *loss.backward()* og *optim.step(). optim.zero_grad()* sørger for, at slette dens historik om gradienter. Dette er meget brugbart, da vi ikke gider at have vores nuværende gradient blive påvirket af tidligere gradienter. Ellers ender det med forkerte gradienter og evt. forkerte parametre.

```
def model_fit(epochs):
    for epoch in range(epochs):
        total_loss = 0

        print(f"epoch {epoch+1}/{epochs}")

        batch_idx = 0
        epoch_loss = 0
        for images, labels in train_loader:
            outputs = model(images.to(device))              # prediction
            loss = lossfunc(outputs, labels.to(device))     # calculating
loss

            optim.zero_grad()           # resetting gradient history
            loss.backward()             # calculating the gradient
            optim.step()                # implementing gradient descent

            total_loss += loss.item()   # Getting the value of the loss
instead of entire tensor
            epoch_loss += loss.item()
            batch_idx += 1

        epoch_loss /= len(train_loader)
        print(f"epoch nr. {epoch+1}: training_loss: {epoch_loss}")
        loss_plots.append(epoch_loss)

        model.eval()
        l, acc = eval_model(val_loader)

        print(f"epoch nr. {epoch+1}: validation_loss: {l}")
        print(f"epoch nr. {epoch+1}: validation_acc: {acc}")

        acc_plots.append(acc)
        val_loss_plots.append(l)
```

Figur 5.4, model_fit fra classifier_scratch2.py.

Efter træningsdelen, skal man så putte modellens opdaterede parametre til testen. Dette gøres, ved at måle modellens præstation med en valideringssæt. I figur 5.4, ser man at metoden

[9] I stedet for at pege mod funktionens global eller lokal maksimumspunkt, leder den så efter minimumspunktet.

model_fit gør brug af *eval_model*, når den evaluerer. Givet, at man kan evaluere en model pr. epoch på valideringssættet, og give en overordnet validering med testsættet, var det optimalt at lave en metode, der kunne kaldes flere gange. Som vist i figur 5.4, returnerer metoden 2 variabler: validation-loss og mængden af forudsigelser den fik korrekt.

```
def eval_model(loader):
    model.eval()
    total_loss = 0
    total_corr = 0
    total_samples = 0

    with torch.no_grad():
        for batch in loader:
            images, labels = batch
            outputs = model(images.to(device))
            loss = lossfunc(outputs, labels.to(device))
            _, predicted = torch.max(outputs, 1)  # taking the most
accurate prediction .. max(P(X = x))
            correct = (predicted == labels.to(device)).sum().item()

            total_loss += loss
            total_corr += correct

            total_samples += batch[1].size(0)

    # calculating average loss and accuracy based on sample size
    avg_loss = total_loss / total_samples
    accuracy = total_corr / total_samples

    return avg_loss, accuracy
```

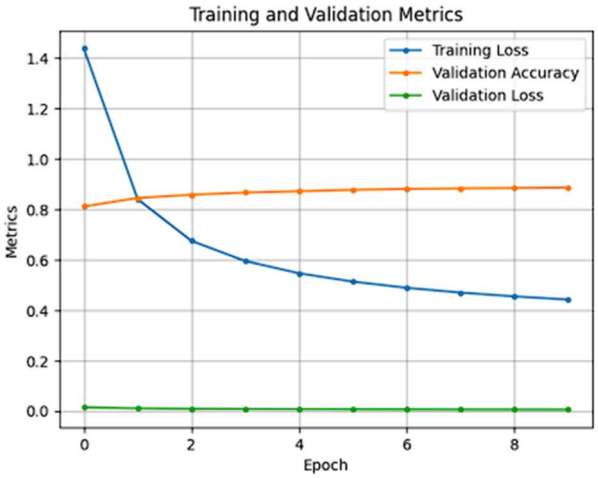
Figur 5.5, eval_model fra classifier_scratch2.py.

eval_model er ligesom træningsdelen. I stedet for at opdatér **W** *og* **b**, laver den en score over mængden af korrekte forudsete labels. Dvs., at programmet tæller mængden af labels modellen, fik korrekt, og returnere det og loss-værdien. Ved at implementér dette i koden, kan vi nu få en oversigt over modellens præstation.

```
epoch 1/10
epoch nr. 1: training_loss: 1.4367210834026336
epoch nr. 1: validation_loss: 0.015453736297786236
epoch nr. 1: validation_acc: 0.8108333333333333
epoch 2/10
epoch nr. 2: training_loss: 0.8400400834878285
….
```

Figur 5.6

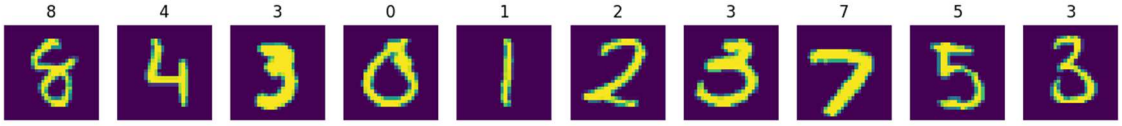
Programmet generer noget data. Dataet, som tidligere nævnt, viser modellens læringskurve, i forbindelse med træningstab, valideringsnøjagtighed og valideringsstab. For at få en bedre forståelse, kan man lave en graf (se figur 5.7). Grafen viser også det samme. Ved at kigge nærmere på



Figur 5.7, graf over data

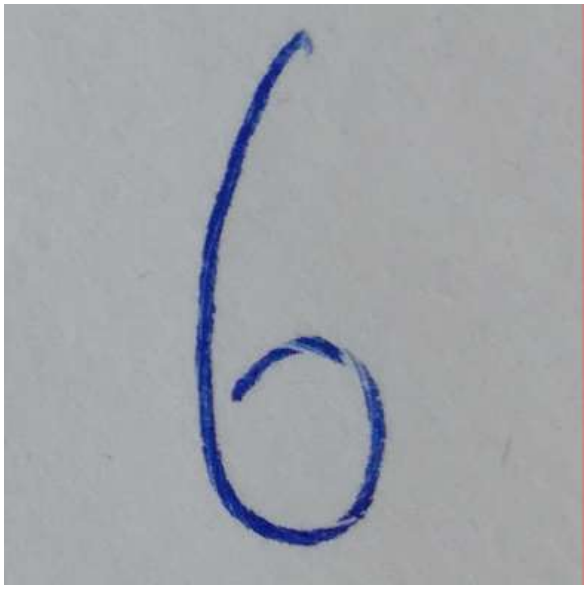
grafen, kan man se, at modellen bliver mere akkurat senere hen i tidslinjen. Givet at den bliver mere akkurat, begynder dens training_loss at falde. Dette skyldes, at den kan genkende flere af eksemplerne, idét at den bliver finpudset under træningen. Modellens accuracy bliver målt som mængden af korrekte forudsigelser ud af mængden af samples. Dvs., $accuracy = \frac{amount\ correct}{amount\ of\ samples}$. Hvis man undersøger grafen i figur 5.7, ville man kunne se, at modellen ville kunne genkende et tal nærmest 90% ad gangen. Dette betyder også, at den ville kunne genkende det næste tal med en 90% sandsynlighed. $P(\boldsymbol{X} = et\ tal\ fra\ 0-9) \approx 90\%$

For at teste dette, kan man få programmet til at klassificér nogle værdier. Lad os først tage udgangspunkt i testsættet se (figur 5.8). Lægge mærke til, at den kan genkende alle de håndskrevne tal på figur 5.8. Dette er ikke ved noget tilfælde. Efter at blive trænet på 60,000 28×28 håndskrevne tal. Har den fået en validation accuracy på 90%. Men dette betyder dog ikke, at den ikke har svært ved at genkende nogle tal.

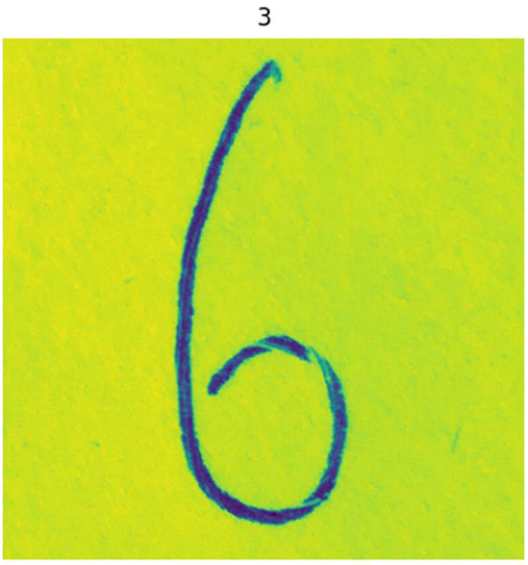


Figur 5.8, test af model.

Lad os nu prøve at se, om vores model kan genkende et håndskrevet tal på papir. (se figur 5.9 og figur 5.10). Som Figurerne nedenfor viser, er tallet på figur 5.9 seks. Dette kunne modellen ikke regne ud. Som tidligere nævnt, har modellen en validation accuracy på 90%. Dette er en validation accuracy på dens eget data, som er MNIST datasættet.



Figur 5.9, håndskrevet tal på papir **(**https://i.sstatic.net/CF1ze.jpg**).**



Figur 5.10, modellens forudsigelse.

Hvis man sammenligner figur 5.10, med tallene i figur 5.8, ligner de ikke hinanden på nogen måde. Dette skyldes nogle faktorer. For det første er baggrunden i figur 5.10 lysgrønt. Dette svarer til foregrundsfarven af tallene i figur 5.8. Sekstallet i figur 5.9 og 5.10 er skrevet med kuglepen. Farven af kuglepennen er stadigvæk i figur 5.10. Dette skyldes, den måde billedet er blevet bearbejdet. Men dette fremmer også en meget vigtig pointe. Modellen er ikke i stand til at genkende et tal på baggrund af farver. Modellen i helhed, kan godt genkende tal, som er

monokrom og stammer fra MNIST datasættet, men har svært ved at genkende andre håndskrevne tal.

Som afrunding, kan man godt konstruere en simpel prototype, der kan nogenlunde genkende håndskrevne tal. Med en nøjagtighed på 90%, kan den genkende tal, der stammer fra MNIST datasættet. Man skal dog være opmærksom på, at den har en lavere chance for at klare sig med håndskrevne tal udenfor MNIST datasættet.

7 DISKUSSION

Som tidligere nævnt, fungerer modellen ret godt på sit eget data. I modsætning til dette, fungerer den mindre godt på en *real-world* eksempel. Dette er helt afhængige af hvordan modellen trænes, og hvad den trænes med. Givet at MNIST datasættet består af 60,000 træningsbilleder, ville den ikke have svært ved at genkende billeder fra MNIST. Men derimod, have svært ved at genkende et tal skrevet på papir.

Det afhænger også af, hvor meget modellen trænes. Indenfor maskinlæring, er der 2 vigtige koncepter, som er over og underfitting. Givet at man underfitter en model, ville modellen ikke kunne genkende sit eget data, og heller ikke eksempler uden for datasættet. Derimod, ville en overfittede model kunne genkende sit eget data meget nemt. Den ville ikke kunne genkende data fra den virkelig verden. Disse to fagbegreber er et slags mål for hvor meget man træner sin model. I vores eksempel, blev modellen trænet på 10 epochs, dvs. 10 gentagelser. I virkeligheden er det ikke særlig meget. Men til gengæld var dens valideringsnøjagtighed ret høj (se figur 5.7). Dette kunne også skyldes variationen i datasættet, givet at de fleste tal ligner næsten hinanden. Dette er også nok en grund til den var så akkurat på sit eget data. Men modellen i det hele taget fungerer ikke godt med billeder udenfor MNIST's scope.

I forbindelse med valg af fag, var der en del positive ting ved dette. Givet at neurale netværk er i virkeligheden sandsynlighedsmodeller, ville det dumt at overse, hvordan de rent faktisk fungerer. Ved at vælge Matematik A og Programmering B, har jeg haft muligheden for at dykke ned og forstå, hvordan et neuralt netværk er opbygget, hvad idéen bag det var, og hvorfor det virker. Desuden med programmering, har jeg haft mulighed for at kunne implementere det selv. Idét at man selv kunne opbygge en prototype, med fuld forståelse af dens funktionalitet. På baggrund af problemformuleringen, ville jeg sige, at faget ikke har nogen decideret begrænsninger. Problemformuleringen er henrettet mod at finde en løsning til hvordan man laver et program, der vha. en lineær neural netværk, kan klassificér billeder. Dermed er Matematik A og Programmering B de mest optimale valg, for at løse problemformuleringen.

Som afrunding, er modellen god til at genkende data fra MNIST datasættet, men har besvær med at eksempler udenfor MNIST datasættet. Hermed kan man konkludere at modellen i det hele taget er god til at genkende billeder. I forbindelse med valg af fag, var Programmering B og Matematik A ret de mest optimale valg. Dette skyldes, at de er gode til at belyse problemet fra flere vinkler. Idét, at man både forstår, hvordan man løser problemet, men også hvordan den implementeres i et system.

8 SAMLET KONKLUSION

Lineær klassifikationsmodeller er ret gode til at forudsige deres eget data, men til gengæld kan have svært ved at genkende noget uden for deres træningssæt. Givet at den kunne genkende 90% af eksempler, der stammer fra MNIST-datasættet. I figur 5.7, ser man, at modellen var ret hurtig om at blive trænet. Idét at dens nøjagtighed var ret højt, og dens træningstab minimeret. Ved at teste den på nogle af eksemplerne fra MNIST-datasættet, så man, at modellen var ret akkurat med sin klassifikationer. Det modsatte kan siges, da den blev testet med et håndskrevet tal på papir. Det kunne dermed diskuteres om der var over eller underfitting. Noget andet at overveje, er hvordan billedet blev bearbejdet. Dvs., hvordan den blev transformerede til læseligt data. Som regel, blev det konkluderede, at det heller var et problem med variansen i MNIST-datasættet.

For at konstruér en softmax klassificeringsmodel, kræver det en minimal forståelse af den bagvedliggende matematik og koden. Lineær klassifikationsmodeller fungerer ret simpelt, da de består af lineære transformationer, softmax regressors og cross-entropy loss-funktioner. En stor afgørende del er lineær algebra og differentiering. Disse er 2 af de vigtigste komponenter, da de viser hvordan en lineær neural netværk laver sin udregninger.

Ved at lave en softmax klassifikationsmodel med 784 inputs og 10 outputs, gjorde at man kunne klassificér billeder. Ved at træne modellen på 28×28 billeder af tal, kunne man vise, hvordan modellen kan klassificér billeder. Idét, at man redegjorde for hvert step i softmax klassifikationsmodellen. Men også analyserede hvordan man bygget en lineær klassifikationsmodel. Dermed kan det konkluderes, at problemformuleringen er blevet besvaret.

9 PERSPEKTIVERING

I forbindelse med analysen, kunne man havde undersøgt en anden form for gradient descent. Der findes andre mere effektive løsninger, såsom Adam-metoden. Adam metoden drejer sig om at bruge en mere momentums baseret fremgangsmåde for at finde et lokalt minimum. Idét at man så optimerer mængden af tid krævet for at træne modellen, og bedst tilpasset parametre. Dvs. man kan optimere sin model, og dermed

Et andet aspekt, er at undersøge andre modeller for klassifikation. Problemformuleringen begrænser mængden af modellerne til nogle få lineære modeller. En af disse er SVM's, dog er de ikke særlige præcise. Ved at kigge på nogle af de større ikke lineære modeller ville en CNN model være ret optimalt. CNN's gør brug af pooling og convolutions til at ekstrahere features. Dvs. at den ekstraherer kendetegn ved billedet. Givet, at softmax klassifikationsmodellen ikke kunne genkende det håndskrevne eksempel på papir, skyldes at den ikke kunne forstå billedet. Dvs., at den ikke kunne udpege formen af sekstallet.

Dette siger også en del om lineære klassifikationsmodeller. De er ikke designet til at ekstrahere features fra billeder, men hellere udføre udregninger på baggrund af de indsatte pixels. Dog kan man godt modificér dem. Idét at man tilføjer en pooling og feature extraction lager. Dermed ville softmax klassifikationsmodellen modtage features. Dette gør, at den ville kunne genkende tallene på baggrund af dens features.

10 REFERENCER

Bowen, E. (u.d.). *Understanding logits in AI and neural networks*. Hentet fra Telnyx: https://telnyx.com/learn-ai/logits-ai

Das, S. M. (u.d.). *Probability Theory 101 for Dummies like Me*. Hentet fra Sangeet - Github.io: https://sangeetm.github.io/projects/2019/10/13/Probability-Theory-101-for-Dummies-like-Me/

Khan, S. A. (u.d.). *Discrete and continuous random variables*. Hentet fra Khan Academy: https://www.khanacademy.org/math/statistics-probability/random-variables-stats-library/random-variables-discrete/v/discrete-and-continuous-random-variables

MNIST Dataset. (u.d.). Hentet fra Engati: https://www.engati.com/glossary/mnist-dataset#:~:text=a%20good%20dataset%3F-,What%20is%20MNIST%20dataset%3F,the%20field%20of%20machine%20learning.

Ng, A. (u.d.). *Coursera.org*. Hentet fra Cost function formula: https://www.coursera.org/learn/machine-learning/lecture/1Z0TT/cost-function-formula

Ng, A. (u.d.). *Learning Rate*. Hentet fra Coursera: https://www.coursera.org/learn/machine-learning/lecture/OoP3Y/learning-rate

Partial Derivative. (u.d.). Hentet fra Newcastle University: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/core-mathematics/calculus/partial-derivatives.html

What is a backpropogation algorithm. (u.d.). Hentet fra Informa TechTarget: https://www.techtarget.com/searchenterpriseai/definition/backpropagation-algorithm

What is a neural network? (u.d.). Hentet fra IBM: https://www.ibm.com/think/topics/neural-networks

What is forward propagation in neural networks? (u.d.). Hentet fra Geeks for Geeks: https://www.geeksforgeeks.org/what-is-forward-propagation-in-neural-networks/

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (u.d.). *2.3 Linear Algebra*. Hentet fra Dive into Deep Learning: https://d2l.ai/chapter_preliminaries/linear-algebra.html

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (u.d.). *2.6 Probability and statistics*. Hentet fra Dive into Deep Learning: https://d2l.ai/chapter_preliminaries/probability.html

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (u.d.). *3.1 Linear Regression*. Hentet fra Dive Into Deep Learning: https://d2l.ai/chapter_linear-regression/linear-regression.html

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (u.d.). *4.1 Softmax Regression*. Hentet fra Dive Into Deep Learning: https://d2l.ai/chapter_linear-classification/softmax-regression.html

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (u.d.). *Probability and statistics*. Hentet fra Dive into Deep Learning: https://d2l.ai/chapter_preliminaries/probability.html

10.1 Figurer

Figur 5.9 - Hentet fra (https://i.sstatic.net/CF1ze.jpg) d. 18/12/24. Forfatter - ukendt.

11 Bilag 1 - Data

```
epoch 1/10
epoch nr. 1: training_loss: 1.4367210834026336
epoch nr. 1: validation_loss: 0.015453736297786236
epoch nr. 1: validation_acc: 0.8108333333333333
epoch 2/10
epoch nr. 2: training_loss: 0.8400400834878285
epoch nr. 2: validation_loss: 0.011431700550019741
epoch nr. 2: validation_acc: 0.84475
epoch 3/10
epoch nr. 3: training_loss: 0.6747297169367472
epoch nr. 3: validation_loss: 0.00976547785103321
epoch nr. 3: validation_acc: 0.8575
epoch 4/10
epoch nr. 4: training_loss: 0.59485276222229
epoch nr. 4: validation_loss: 0.008836435154080391
epoch nr. 4: validation_acc: 0.8661666666666666
epoch 5/10
epoch nr. 5: training_loss: 0.5463407789866129
epoch nr. 5: validation_loss: 0.008221527561545372
epoch nr. 5: validation_acc: 0.8714166666666666
epoch 6/10
epoch nr. 6: training_loss: 0.5132817226847013
epoch nr. 6: validation_loss: 0.007782710250467062
epoch nr. 6: validation_acc: 0.877
epoch 7/10
epoch nr. 7: training_loss: 0.48883247760931653
epoch nr. 7: validation_loss: 0.007458352483808994
epoch nr. 7: validation_acc: 0.8804166666666666
epoch 8/10
epoch nr. 8: training_loss: 0.46990906343857447
epoch nr. 8: validation_loss: 0.007189915515482426
epoch nr. 8: validation_acc: 0.8826666666666667
epoch 9/10
epoch nr. 9: training_loss: 0.4549017608563105
epoch nr. 9: validation_loss: 0.006977986078709364
epoch nr. 9: validation_acc: 0.8840833333333333
epoch 10/10
epoch nr. 10: training_loss: 0.4423167727192243
epoch nr. 10: validation_loss: 0.00680222362279892
epoch nr. 10: validation_acc: 0.8859166666666667
```

12 Bilag 2 - Pseudokode

```
start
```

```
create classifier class as model
setup GPU

lossfunc = CrossEntropyLoss()
optim = optim.SGD(model.parameters())

load MNIST_DATASET into train_set AND test_set
train_size = 0.8 * len(train_set)
val_size = len(train_set) - train_size

train_subset, val_subset = split train_set according to sizes

create dataloaders from subsets with batch_size{64}:
    train_loader from train_subset
    val_loader   from val_subset

    # optional
    test_loader  from test_subset

model_fit(epochs):
    for epoch in epochs:
        for images and labels in train_loader:
            output = predict(images)
            loss = lossfunc(output, labels)

            backpropogate loss
            optim.step()
            store loss

        for images and labels in val_loader:
            output = predict(images)
            calculate val_loss <- lossfunc(output, labels)
            measure accuracy
            store val_loss and accuracy


save model
plot loss val_loss and accuracy

end
```

13 BILAG 3 - KODE

13.1 CLASSIFIER_BASE2.PY

```
"""
* FILE      : classifier_base2.py
* AUTHOR    : SYED M. AMIN
* PROJECT   : SOP
* DESC      : a file containing all the softmax classifier class and some
transforms for image preproccesing
"""
```

```
import torch

from torchvision import transforms

class SoftmaxClassifier(torch.nn.Module):
    def __init__(self, n_inputs, n_outputs):
        super().__init__()
        self.linear = torch.nn.Linear(n_inputs, n_outputs)
        self.flatten = torch.nn.Flatten()

    def forward(self, x):
        x = self.flatten(x)
        return self.linear(x)

# preproccessing template
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

all_transforms = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,),
                         std=(0.5,))
])
```

13.2 CLASSIFIER_SCRATCH2.PY

```
"""
* FILE      : classifier_scratch2.py
* AUTHOR    : SYED M. AMIN
* PROJECT   : SOP
* DESC      : the program that trains the softmax classifier and saves it.
"""


import torch
import matplotlib.pyplot as plt

from torchvision import transforms
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader, random_split
from classifier_base2 import SoftmaxClassifier, transform

LR = 1e-3
num_classes = 10
num_epochs = 10
batch_size = 64

"""
model = torch.nn.Sequential(
    torch.nn.Flatten(),
    torch.nn.Linear(28*28, 10)
)"""


model = SoftmaxClassifier(784, num_classes)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
lossfunc = torch.nn.CrossEntropyLoss()
optim = torch.optim.SGD(model.parameters(), lr=LR)

# loading dataset
train_dataset = MNIST(root='../data', train=True, transform=transform,
download=True)

# preparing sizes
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset = random_split(train_dataset, [train_size,
val_size])

train_loader = DataLoader(train_subset, batch_size=batch_size,
shuffle=True)
val_loader = DataLoader(val_subset, batch_size=batch_size, shuffle=False)

# plots list
loss_plots = []
acc_plots = []
val_loss_plots = []

def eval_model(loader):
    model.eval()
    total_loss = 0
    total_corr = 0
    total_samples = 0

    with torch.no_grad():
        for batch in loader:
            images, labels = batch
            outputs = model(images.to(device))
            loss = lossfunc(outputs, labels.to(device))
            _, predicted = torch.max(outputs, 1)  # taking the most
accurate prediction .. max(P(X = x))
            correct = (predicted == labels.to(device)).sum().item()

            total_loss += loss
            total_corr += correct

            total_samples += batch[1].size(0)

    # calculating average loss and accuracy based on sample size
    avg_loss = total_loss / total_samples
    accuracy = total_corr / total_samples

    return avg_loss, accuracy
def model_fit(epochs):
    for epoch in range(epochs):
        total_loss = 0
        total_acc = 0

        print(f"epoch {epoch+1}/{epochs}")

        batch_idx = 0
        epoch_loss = 0
        for images, labels in train_loader:
            outputs = model(images.to(device))              # prediction
            loss = lossfunc(outputs, labels.to(device))     # calculating
loss
```

```
            optim.zero_grad()               # resetting gradient history
            loss.backward()                 # calculating the gradient
            optim.step()                    # implementing gradient descent

            total_loss += loss.item()    # Getting the value of the loss
instead of entire tensor
            epoch_loss += loss.item()
            batch_idx += 1

        epoch_loss /= len(train_loader)
        print(f"epoch nr. {epoch+1}: training_loss: {epoch_loss}")
        loss_plots.append(epoch_loss)

        model.eval()
        l, acc = eval_model(val_loader)

        print(f"epoch nr. {epoch+1}: validation_loss: {l}")
        print(f"epoch nr. {epoch+1}: validation_acc: {acc}")

        acc_plots.append(acc)
        val_loss_plots.append(l)

model_fit(num_epochs)
torch.save(model.state_dict(), 'sofm_cls_scratch.params')

plt.plot(range(0, len(loss_plots)), loss_plots, marker='.', linestyle='-',
label='Training Loss')
plt.plot(range(0, len(acc_plots)), acc_plots, marker='.', linestyle='-',
label='Validation Accuracy')
plt.plot(range(0, len(val_loss_plots)), val_loss_plots, marker='.',
linestyle='-', label='Validation Loss')

plt.xlabel('Epoch')
plt.ylabel('Metrics')
plt.title('Training and Validation Metrics')
plt.legend()
plt.grid()
plt.show()
```

13.3 SOFTMAX_TEST.PY

```
"""
* FILE      : softmax_test.py
* AUTHOR    : SYED M. AMIN
* PROJECT   : SOP
* DESC      : Testing the softmax classifier on some handwritten digits
from MNIST test dataset plus 1 image of a hand drawn number taken from the
internet
"""

import torch
import matplotlib.pyplot as plt
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
from PIL import Image

from classifier_base2 import SoftmaxClassifier, transform, all_transforms

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = SoftmaxClassifier(784, 10)
```

```
model.load_state_dict(torch.load("sofm_cls_scratch.params",
weights_only=True, map_location=device))
model = model.to(device)

test_dataset = MNIST("../data", False, transform, download=True)
test_loader = DataLoader(test_dataset, 64, True)

img = Image.open("./test_digit.jpg").convert("L")
img_tensor = all_transforms(img)
img_tensor = img_tensor.to(device)

def plot_test_images(dataloader, num_img=10):
    images_shown = 0

    fig, axes = plt.subplots(1, num_img, figsize=(15, 5))

    for images, labels in dataloader:
        for idx in range(num_img):
            image = images[idx]
            _, pred = torch.max(model(image.to(device)), 1)

            image = image.permute(1, 2, 0)

            axes[idx].imshow(image.cpu().numpy())
            axes[idx].axis("off")
            axes[idx].set_title(f"{pred.item()}")

            images_shown += 1

            if images_shown == num_img:
                plt.show()
                return

model.eval()

_, pred = torch.max(model(img_tensor), 1)

plt.imshow(img)
plt.axis("off")
plt.title(pred.item())
plot_test_images(test_loader)
```