TMA4320 – Prosjekt 2

February 20, 2020

1 Introduksjon

Formålet med dette prosjektet er å introdusere et meget viktig område der vitenskapelige beregninger spiller en stor rolle: nemlig maskinlæring. Maskinlæring inkluderer et bredt spekter av ulike metoder og anvendelser, og i dette prosjektet skal vi fokusere på et lite, men spennende delområde som kalles dyplæring ("deep learning" på engelsk) som uttrykkes gjennom såkalte dype nevrale nettverk. Prosjektet skal bidra til læringsutbytte beskrevet i emnet, for eksempel: (3) implementere utvalgte numeriske algoritmer på en datamaskin; (4) foreta en kritisk vurdering av resultatene; (5) kommunisere dette i form av en skriftlig rapport.

Vi forklarer problemstillingen ved et eksempel: Tenkt deg at du har et foto gitt i digital form, og du ønsker å lage en implementasjon som kan bestemme om det fins en katt eller ikke i bildet, en såkalt binær klassifisering (katt eller ingen-katt). Dette er ingen enkel oppgave, for programmet må ta stilling til at katten kan innta alle mulige orienteringer og posisjoner. Bildet kan vise katten forfra, fra siden, bakfra, hoppende ned fra vinduskarmen, eller utstrakt i en sofa. Den kan fylle hele bildet, eller være en liten del av bildet øverst oppe i et hjørne for eksempel. Det er interessant å merke seg at for oss mennesker er dette vanligvis ganske enkelt, når du ser på et bilde vil du normalt kunne fastslå om det er en katt der eller ikke, bare innen brøkdelen av et sekund. Idéen er at vi forsøker å trene vårt dataprogram til å gjenkjenne en katt med en metode som ligner litt på hvordan hjernen vår virker. Måten vi gjør det på er å hente inn et stort utvalg av bilder, kanskje noen millioner fra en database. I mange av dem fins en katt, da merker vi dem med "katt=1", mens mange har ingen katt, da merker vi dem med "katt=0". Deretter setter vi opp en modell som suksessivt anvender en serie av transformasjoner på det digitale bildet. Hver transformasjon er forbundet med et antall parametre som i utgangspunktet er frie. Etter den siste transformasjonen projiserer vi resultatet på en skalar verdi. Hvis verdien er (nær) 1 så konkluderer vi med at det er en katt på bildet, hvis den er nær 0 så er konklusjonen at det ikke er en katt på bildet. Hvis verdien er ca 0.5 så er konklusjonen mer usikker. Siden vi har et stort antall bilder der vi vet "fasiten" så kan vi regne ut et akkumulert avvik mellom modellens konklusjoner og fasiten. Strategien vår er å søke etter verdier av parametrene assosiert med hver transformasjon som gjør at nevnte avvik blir minst mulig, dette kalles treningsfasen. Når alle parametrene som definerer transformasjonene er bestemt så kan

vi teste modellen på nye bilder som vi ikke brukte i treningsfasen for å se om modellen da finner rett konklusjon. Dette er validering- eller testfasen. Vi formulerer modellen tydeligere i de neste kapitlene, og vi skal bruke bilder som er enklere enn kattefoto.

I kapittel 2 beskriver vi modellen vi skal bruke. Kapittel 3 gir noen tips om implementasjon og angir en overordnet algoritme for treningsmodellen. Kapittel 4 beskriver de to testproblemene vi skal anvende modellen på. Kapittel 5 angir hva du skal ha med i rapporten. I appendix A beskrives utledning av formler som brukes for beregning av gradienten i optimeringsalgoritmen.

2 Beskrivelse av modellen

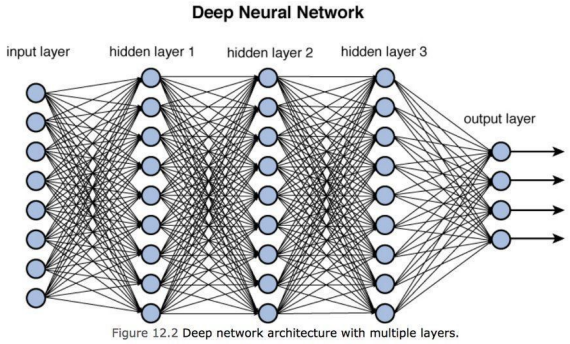

Figure 12.2 Deep network architecture with multiple layers.

Sett fra beskrivelsen kan vi se på et gitt bilde som "input data" og den skalare klassifiseringen som "output data". Lagene merket "hidden layers" svarer til de transformasjonene som er nevnt i introduksjonen. Antallet "hidden layers" kan være stort i dype nevrale nettverk, antallet er det vi refererer til som "dybden" av et dypt nevralt nettverk. I dette prosjektet skal vi holde oss med en forholdsvis beskjeden dybde.

2.1 Inputdata og transformasjoner

Et bilde består av piksler i et $m \times n$ gitter som utgjør bilderommet. Hvert pixel har enten en gråtoneverdi eller tre verdier som svarer til en fargekode (RGB-koding er vanlig). Alle pikselverdiene kan stables opp i en lang vektor, i gråtonetilfellet vil denne vektoren ha lengde $d = m \cdot n$ og man kan for eksempel sortere dem radvis fra bildet. Vi skal derfor alltid anta at hver enkelt input-data i vårt tilfelle er en vektor. I en leketøy-modell som vi også skal bruke, så antar vi at hvert bilde kun består av et punkt i planet, dvs hele bildet er definert av en x-koordinat og en y-koordinat, slik at $d = 2$. Dette vil egentlig utgjøre hovedeksemplet vårt og beskrives i mer detalj senere.

Som tidligere nevnt vil hver transformasjon være assosiert med et sett av parametre. Vi deler parametrene i lag nr k inn i vekter W_k og bias b_k, der W_k er en $d \times d$-matrise og b_k er en d-vektor for hver k. Transformasjonene mellom input og første lag er av samme form som transformasjonen mellom lagene. Vi definerer et bilde til å være $Y_0 \in \mathbb{R}^d$ og bruker transformasjonen

$$Y_{k+1} = Y_k + h\sigma(W_k Y_k + b_k)$$

der h er en skrittlengde som man kan prøve seg litt fram med, et utgangspunkt kan være $h = 0.1$. Her er *aktiveringsfunksjonen* σ en gitt skalar funksjon, $\sigma : \mathbb{R} \to \mathbb{R}$ som anvendes komponentvis på alle elementene i vektoren. Det fins flere populære valg av slike σ, to mye brukte eksempler er

$$\sigma(x) = \tanh x,$$
$$\sigma(x) = \max(0, x)$$

og i dette prosjektet skal vi bruke den første av dem. Den samme transformasjonen brukes for alle bilder vi trener modellen med, dvs samme W_k, b_k for alle bilder Y_0. Det kan være praktisk å kjøre mange eller kanskje alle inputbilder samtidig gjennom modellen. For å skrive dette på en lur måte utvider vi Y_0 fra å være en kolonnevektor med d elementer til å være en $d \times I$-matrise $\mathbf{Y}_0 \in \mathbb{R}^{d \times I}$ der kolonne i inneholder det i'te bildet og I er antall bilder som kjøres gjennom parallelt.. På tilsvarende måte utvider vi Y_k til en matrise $\mathbf{Y}_k$ for alle k. Transformasjonen blir essensielt den samme dvs

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\sigma(W_k \mathbf{Y}_k + b_k), \tag{1}$$

der aktiveringsfunksjonen σ nå virker elementvis på en hel matrise.

Merk at for leketøy-modellen vil hver W_k være en 2×2-matrise og b_k en vektor i planet. I mange modeller vil man kunne velge en annen dimensjon d på de indre lagene enn på input-laget, men i vårt tilfelle bruker vi samme d. La oss anta at det totalt er K lag. Når vi nå skal projisere på en skalar så bruker vi

$$\mathbf{Z} = \eta(\mathbf{Y}_K^T w + \mu \mathbf{1}) \tag{2}$$

der $w \in \mathbb{R}^d$ og $\mu \in \mathbb{R}$ også er parametre på samme måte som W_k og b_k og der $\mathbf{1}$ er vektoren i $\mathbf{R}^I$ der alle elementene er 1. For η kan vi anvende den skalare funksjonen

$$\eta(x) = \frac{e^x}{e^x + 1} = \frac{1}{2}(1 + \tanh \frac{x}{2})$$

elementvis på vektoren i (2). For alle reelle x blir $\eta(x)$ et tall mellom 0 og 1.

2.2 Objektfunksjon og klassifisering

La oss anta at vi har et gitt forslag til verdier på parametrene $(W_k, b_k)_{k=0}^{K-1}$, w, μ. For hvert bilde, dvs kolonne i matrisen $\mathbf{Y}_0$, fra databasen fins en "label" c_i (for eksempel $c_i = 1$ hvis "katt", $c_i = 0$ hvis "ingen katt"). Vi lar $\mathbf{c} = [c_1, \ldots, c_I]^T$ der I er antall bilder. Når $\mathbf{Y}_0$ kjøres gjennom transformasjonene (lagene) som beskrevet og deretter projiseres får vi ut $\mathbf{Z}$. Dersom modellen hadde vært perfekt trent så ville vi hatt for i'te element at $Z_i = c_i$ for alle i, men dette er altfor mye å forlange, det kan også være umulig eller uønsket med en slik perfekt match. Vi kan lage en objektfunksjon eller kostfunksjon som måler hvor langt modellen er unna å klassifisere perfekt. For eksempel, om vi trener modellen på I bilder kan vi sette

$$\mathcal{J} = \frac{1}{2} \sum_{i=1}^{I} |Z_i - c_i|^2 = \frac{1}{2} \|\mathbf{Z} - \mathbf{c}\|^2, \tag{3}$$

men det fins også andre mulige objektfunksjoner man kan bruke.

Det neste vi spør oss om er hva $\mathcal{J}$ *egentlig* avhenger av? Dataene $\mathbf{Y}_0$ kan betraktes som gitt. Faktisk er det kun parametrene $\mathbf{U} = [(W_k, b_k)_{k=0}^{K-1}, w, \mu]$ som betyr noe. Gitt alle disse kan verdien til $\mathcal{J}$ beregnes uten mer informasjon[1], ved å transformere K ganger med (1) og så projisere med (2). Merk at antallet parametre totalt vil være K matriser W_k, hver med d^2 elementer, K vektorer b_k hver med d elementer, w som har d elementer, og μ som er en skalar, altså $K \cdot d^2 + K \cdot d + d + 1$. Vi vil i det følgende ofte skrive $\mathcal{J} = \mathcal{J}(\mathbf{U})$ når vi tenker på denne måten.

2.3 Optimering

For å finne verdien av parametrene $\mathbf{U}$ som minimerer $\mathcal{J}(\mathbf{U})$ bør vi som vi tidligere har lært forsøke sette gradienten til $\mathcal{J}(\mathbf{U})$ med hensyn på $\mathbf{U}$ til null. Dette ville gitt et ikke-lineært ligningssystem som vi kunne ha forsøkt å løse f.eks. med Newton's metode for systemer. Men dessverre er det noen problemer forbundet med dette, blant annet blir Jacobimatrisen ofte singulær og da bryter Newtoniterasjonen sammen. Innen dette fagfeltet er det derfor vanlig bruke enklere og mer robuste metoder i klassen descent-metoder. Siden $\mathcal{J}$ lokalt avtar raskest i retning av $-\nabla\mathcal{J}(\mathbf{U})$ så kan vi i første omgang forsøke med en iterasjon (for $\mathbf{U}$) av typen

$$\mathbf{U}^{(j+1)} = \mathbf{U}^{(j)} - \tau \nabla \mathcal{J}(\mathbf{U}^{(j)}) \tag{4}$$

Parameteren τ forteller altså hvor langt vi skal gå langsmed gradienten, og den kalles *læringsparameteren* av de innvidde. Prøv deg fram til mest effektiv verdi i eksperimentene. Denne enkle metoden blir noen ganger omtalt som "plain vanilla gradient descent" Det fins mange flere avanserte varianter av iterasjoner som bruker gradienten, se for eksempel

`https://ruder.io/optimizing-gradient-descent/`

En av de mest populære algoritmene som er omtalt på denne websiden heter Adam-metoden. Vi beskriver kort denne algoritmen nedenfor, og anbefaler at den benyttes iallfall på Testproblem 2.

Adam descent algorithm

$\beta_1 := 0.9$, $\beta_2 := 0.999$, $\alpha := 0.01$, $\epsilon := 10^{-8}$
$v_0 := \mathbf{0}$, $m_0 := \mathbf{0}$
for $j = 1, 2, \ldots$
$g_j := \nabla_{\mathbf{U}} \mathcal{J}(\mathbf{U}^{(j)})$
$m_j := \beta_1 m_{j-1} + (1 - \beta_1) g_j$
$v_j := \beta_2 v_{j-1} + (1 - \beta_2)(g_j \odot g_j)$
$\hat{m}_j = \frac{m_j}{1 - \beta_1^j}$

[1] ikke helt korrekt, siden vi også må sette en verdi for h

$\hat{v} = \frac{v_j}{1-\beta_2^j}$
$\mathbf{U}^{(j+1)} := \mathbf{U}^{(j)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j}+\epsilon}$

end

Noen presiseringer av notasjon

- β_1^j betyr "β_1 i j'te potens". (Tilsv. for β_2^j)
- $\sqrt{\hat{v}_j}$ er komponentvis kvadratrot og ϵ adderes til alle komponenter
- Divisjonen $\frac{\hat{m}_j}{\sqrt{\hat{v}_j}+\epsilon}$ er også komponentvis

Stochastic gradient descent. Vi nevner også dette som et supplement til Adam-metoden. Dette er en enkel ide. Vi tenker oss generelt at vi har I bilder til rådighet i treningsmodellen. Men det er ikke nødvendig å bruke alle disse I bildene samtidig. Vi kan bestemme oss i koden for bruke et antall `chunk` bilder i hvert sveip gjennom lagene, og så oppdatere parametrene W_k, b_k, w, μ bare basert på disse `chunk` bildene. Neste sveip bruker vi et annet utvalg av bilder osv. Det mest vanlig er å trekke bildene tilfeldig fra databasen uten tilbakelegging, men dette krever litt bokføring. En enklere variant er å kun velge (i hvert sveip) et tilfeldig tall `start` mellom 1 og $I-$`chunk` og så bruke bildene nummeret fra `start` til `start+chunk` dvs trekning av en sammenhengende sekvens av bilder med tilbakelegging.

2.4 Beregning av gradienten

Vi skal finne $\nabla\mathcal{J}(\mathbf{U})$ for alle "delene" av $\mathbf{U}$, dvs vi må finne

$$\frac{\partial\mathcal{J}}{\partial W_k}, \quad \frac{\partial\mathcal{J}}{\partial b_k}, \quad \frac{\partial\mathcal{J}}{\partial w}, \quad \frac{\partial\mathcal{J}}{\partial \mu},$$

alle evaluert i $\mathbf{U}^{(j)}$. Her er det kjerneregelen som gjelder. Starter vi for eksempel med (3) så er det der kun $\mathbf{Z} = (Z_i)$ som avhenger av parametrene $\mathbf{U}$. For eksempel kunne vi beregne ved kjerneregelen

$$\frac{\partial\mathcal{J}}{\partial\mu} = \sum_{i=1}^{I} \frac{\partial\mathcal{J}}{\partial Z_i} \cdot \frac{\partial Z_i}{\partial\mu}$$

Setter vi inn de konkrete uttrykkene for $\mathcal{J}$ og Z_i fra (3) og (2) så får vi

$$\frac{\partial\mathcal{J}}{\partial\mu} = \eta'(\mathbf{Y}_K^T w + \mu\mathbf{1})^T(\mathbf{Z}-\mathbf{c}) \tag{5}$$

som er en skalar størrelse. Beregning av $\frac{\partial\mathcal{J}}{\partial w}$ følger nesten presist samme mal, men husk her at denne deriverte blir en vektor av dimensjon d fordi w har d komponenter vi deriverer med hensyn på. Vi angir bare sluttresultatet

$$\frac{\partial\mathcal{J}}{\partial w} = \mathbf{Y}_K\left[(\mathbf{Z}-\mathbf{c}) \odot \eta'(\mathbf{Y}_K^T w + \mu)\right] \tag{6}$$

der vi har introdusert Hadamard produktet $\odot$ mellom to matriser (vektorer) av samme dimensjon.

$$(A \odot B)_{ij} = A_{ij} \cdot B_{ij}.$$

`numpy.multiply` kan brukes for denne operasjonen, men det er også mulig å rett og slett bruke '*' (til forskjell fra '@'), test gjerne ut med et enkelt eksempel. Merk forøvrig at $A \odot B = B \odot A$. Hvis vi deriverer $\mathcal{J}$ med hensyn på W_k eller b_k for en vilkårlig $k \in \{0, \ldots, K-1\}$ så blir det mer komplisert fordi vi må bruke kjerneregelen mange ganger. Vi presenterer sluttresultatet og henviser til appendiks for en utledning av formlene.

$$\mathbf{P}_K = \frac{\partial \mathcal{J}}{\partial \mathbf{Y}_K} = w \cdot [(\mathbf{Z} - \mathbf{c}) \odot \eta'(\mathbf{Y}_K^T w + \mu \mathbf{1})]^T \tag{7}$$

$$\mathbf{P}_{k-1} = \mathbf{P}_k + h\, W_{k-1}^T \cdot [\sigma'(W_{k-1}\mathbf{Y}_{k-1} + b_{k-1}) \odot \mathbf{P}_k] \tag{8}$$

$$\frac{\partial \mathcal{J}}{\partial W_k} = h\, [\mathbf{P}_{k+1} \odot \sigma'(W_k \mathbf{Y}_k + b_k)] \cdot \mathbf{Y}_k^T \tag{9}$$

$$\frac{\partial \mathcal{J}}{\partial b_k} = h[\mathbf{P}_{k+1} \odot \sigma'(W_k \mathbf{Y}_k + b_k)] \cdot \mathbf{1} \tag{10}$$

Merk at dette fungerer ved at man først beregner $\mathbf{P}_K$ basert på $\mathbf{Y}_K$ som vi fant fra K iterasjoner med (1). Deretter gjør vi såkalt "back propagation" med (8) og finner alle $\mathbf{P}_k$, $k < K$. Da har vi nok informasjon til å finne bitene av gradienten fra (9) og (10). Det blir mer om implementasjonen i neste avsnitt.

3 Implementasjon i Python

Det fins ferdige pakker i Python som man kan bruke til å sette opp og trene nevrale nett med et minimum av kode. Eksempler på slike er `pytorch` og `TensorFlow` (sistnevnte kan også interfaces mot mange andre språk enn Python). I dette kurset vil vi imidlertid kikke litt mer "under panseret" på algoritmene, og derfor skal du utføre det meste av implementasjonen med `numpy`.

Strukturering av koden på en oversiktlig måte er viktig og tellende. Algoritmen er såpass kompleks at det vil være lønnsomt å ha et visst system i kodestrukturen. Det anbefales å bruke informative navn på variabler og funksjoner (og mest mulig i tråd med formlene i oppgaven). Hvis for eksempel koden deles opp i passe små funksjoner, si på 5-15 linjer, ser man at noen variabler alltid brukes sammen. Dette kan utnyttes til å introdusere objektorientering for å gjøre koden enda mer ryddig. Men det trekkes ikke poeng om man ikke bruker objektorientering.

Merk ellers at de fleste operasjoner du skal gjøre på matriser og vektorer kan gjøres uten å lage for-løkker over indeksene i matrisen/vektoren.

3.1 Overordnet algoritme for trening av modellen

Vi presenterer hele treningsprosessen i én bolk, men anbefaler som beskrevet ovenfor at du splitter denne opp i mindre funksjoner for bedre kodestruktur, enklere feilsøking osv.

Algoritme

Sett antall lag K
Sett læringsparameter τ
Sett $\mathbf{Y}_0$ ved å lese inn eller generere data
Sett random startverdier for vekter og bias i alle lag
while not converged
    for $k = 1 : K$
        Beregn $\mathbf{Y}_k$ og lagre i minnet
    end
    Beregn $\mathbf{P}_K$ fra (7) og lagre i minnet
    Beregn bitene av gradienten tilhørende projeksjonssteget (5), (6)
    for $k = K : -1 : 2$
        Beregn $\mathbf{P}_{k-1}$ fra (8)
    end
    for $k = 0 : K - 1$
        Beregn bidrag til gradienten fra (9), (10)
    end
    Oppdater vekter og bias enten som antydet i (4) eller fra Adam-metoden
end (while)

3.2 Hvordan velge parametre?

Det er en del av friheten i oppgaven å gjøre valg av parametre på en fornuftig og interessant måte. Men det kan være vanskelig uten noen som helst idé om hva slags omtrentlige verdier man bør prøve. Vi gir en kort diskusjon som kan tas som et utgangspunkt

1. h: skrittlengde i transformasjonene. Her kan man starte med $h = 0.1$ og eventuelt prøve seg fram videre.

2. τ: Læringsparameter i "plain vanilla" optimering. Forsøk $\tau \in [0.01, 0.1]$ og se hva som konvergerer raskest

3. K: Antall lag i nettverket. Under debugging, bruk et lite antall lag, f.eks. 2-3. Øk så etterhvert. For spiraleksemplet kan du sikkert få gode resultater med 15-20 lag.

4. *Antall data punkter i spiraleksemplet.* Bruk et beskjedent antall, si 10-100 i debuggingsfasen. Sett så opp gradvis ettersom du er tryggere på at koden fungerer. Kjører de endelige resultatene med f.eks. 1000 punkter.

5. *Antall bilder i MNIST - trening.* Det fins totalt 60000 treningsbilder å velge i, så med binærklassifikasjon (kun utvalg på to siffer), så blir det ca 12000 bilder maksimalt. Om du implementerer Adam-optimering så kan du kjøre SGD (Stochastic Gradient Descent) som forklart med bruk av et mer beskjedent antall bilder i hver iterasjon, forsøk f.eks. 50-100 bilder. Disse kan da velges tilfeldig fra hele settet på ca 12000 bilder.

6. *Konvergens – antall iterasjoner i optimeringsfasen.* I mange andre anvendelser av vitenskapelige beregninger så bruker man konvergensestimater som stoppkriterium. I dyplæring er ikke dette nødvendigvis så lett å definere på en god måte. I dette prosjektet kan du for enkelthets skyld bare definere et totalt antall iterasjoner som skal gjøres. Du kan gjerne avpasse dette litt etter hvor lang tid det tar å kjøre. Selv har jeg brukt 40 000 iterasjoner på punkteksemplet, og da tar programmet under ett minutt å kjøre. Men i debuggingsfasen bør du starte med et lavt antall iterasjoner, én kan være nok. Er det en feil i koden så fins den gjerne i alle iterasjoner.

4 Testproblemer og data

4.1 Første testproblem – punkter i planet

Dette er eksemplet nevnt tidligere der $d = 2$, "bildet" er kun et punkt beskrevet av to koordinater (x, y). Dette betyr at $\mathbf{Y}_k$ blir en $2 \times I$-matrise. Vi skiller mellom to typer punkter, de kan være røde eller blå. Det betyr at label c_i tilhørende punkt i er 0 =blå eller 1 =rød. Det suppleres en funksjon `get_data_spiral_2d`. Modellen skal da trenes opp til å avgjøre om et vilkårlig punkt har rød eller blå farge.

4.2 MNIST – siffergjenkjenning

Dette er et klassisk problem i maskinlæring som har eksistert lenge. Her har vi virkelige bilder med de er små, dvs lav oppløsning. Vi snakker om 28×28 piksler, slik at bildedimensjonen blir $d = 28 \cdot 28 = 784$. Disse dataene fins på nettet, se

```
http://yann.lecun.com/exdb/mnist/
```

Der fins the 4 filer som alle kan lastes ned på din lokale pc.

```
train-images-idx3-ubyte.gz:  training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:  training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:   test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:   test set labels (4542 bytes)
```

De to første er henholdsvis treningsdata (bilder) og treningsdata (labels). De to siste er tilsvarende for testing (validering). Man kan laste ned og pakke ut disse 4 filene, og vi supplerer en funksjon som leser inn dataene og legger dem i numpy arrays. Det fins 60000 bilder i treningssettet og 10000 i valideringssettet. Label'ene er siffer fra 0 til 9 som beskriver hvilket håndskrevet siffer som fins på bildet, altså 10 mulige klasser. Dette betyr at klassifisering for MNIST-settet ikke er *binær* klassifisering slik vi ønsker å se på i dette prosjektet. Vi skal derfor begrense oss til å studere kun to forskjellige siffer, f.eks. 4 og 9, så vi bruker et utvalg av dataene som består av to slike valgte siffer.

5 Krav til rapport.

Rapporten skal fortrinnsvis leveres inn som en Jupyter Notebook i ipynb-format.

1. Det bør være en introduksjon som kort forklarer kort hva problemet går ut på og forklarer rapportens struktur, det er ikke nødvendig å reprodusere store deler av prosjektbeskrivelsen.

2. Det skal lages kode som bør være strukturert, ryddig, lett og lese, og ikke minst fungere. Velg gode variabelnavn, gjerne i tråd med prosjektbeskrivelsen. Selve algoritmen bør være laget så generell at det er lett å for eksempel skifte ut data, endre optimeringsmetode etc. Parametre man finner fra treningsdelen bør kunne enkelt lagres til disk og leses inn senere når man vil validere, teste og vise resultater.

3. Presentasjon av resultater fra Testproblem 1 (punkter i planet). Det er opp til gruppen å finne gode måter å gjøre dette på, men noen tips og forslag fins nedenfor.

4. Presentasjon av resultater fra Testproblem 2 (MNIST – siffergjenkjenning). Se flere detaljer om mulig presentasjon av resultater nedenfor.

5.1 Tips angående punkt 3

Her er noe forslag til interessante resultatvurderinger av modellen anvendt på Testproblem 1. Gruppen må selv vurdere hva slags resultater som er interessant å vise fra,

- For et klassifiseringsproblem kan man generelt teste ut den opptrente modellen ved å hente nye data som ikke ble brukt i treningsfasen, og kjøre de gjennom modellen med parametrene fiksert fra treningsfasen. For å teste modellens effektivitet kan du gjøre et nytt kall til `get_data_spiral_2d` der du henter inn et stort antall koordinatpar (x_i, y_i) med label c_i. Så kjører du disse gjennom modellen (med parameterverdier du har funnet) og klassifiserer disse nye punktene. Så teller du til slutt opp hvor stor andel av de nye dataene som klassifiseres korrekt. Denne prosentandelen er interessant å rapportere. Du kan lage suksessrater for litt ulike tilfeller der du for eksempel varierer antall lag. Du kan også lete deg fram til optimale verdier for skrittlengdene h og τ.

- Forslag til visualisering. Siden dette problemet er så enkelt så kan du forsøke å se hva som hender dersom du tester punkter som fyller ut et helt kvadrat, f.eks. $[-1.2, 1.2] \times [-1.2, 1.2]$ der du typisk bruker et par hundre punkter i hver retning. Så kjører du alle disse punktene gjennom modellen og finner klassifikasjonen av hver av dem som et tall mellom 0 og 1 (Z_i). Lag så et plott med en fargeskala som går mellom rødt og blått i henhold til verdiene Z_i. Plott treningsdataene sine utgangskoordinater $\mathbf{Y}_0$ som punkter i lyserød eller lyseblå farge på toppen av dette. Man kan også lage tilsvarende plott, men istedet for å plotte utgangskoordinatene til punktene så plotter man koordinatene fra det siste laget $\mathbf{Y}_K$. Et eksempel på et slikt plott er gitt i figur 1. Nyttige funksjoner en kan gjøre seg bruk av her er: `numpy.linspace, numpy.meshgrid`, `imshow` fra `matplotlib`.

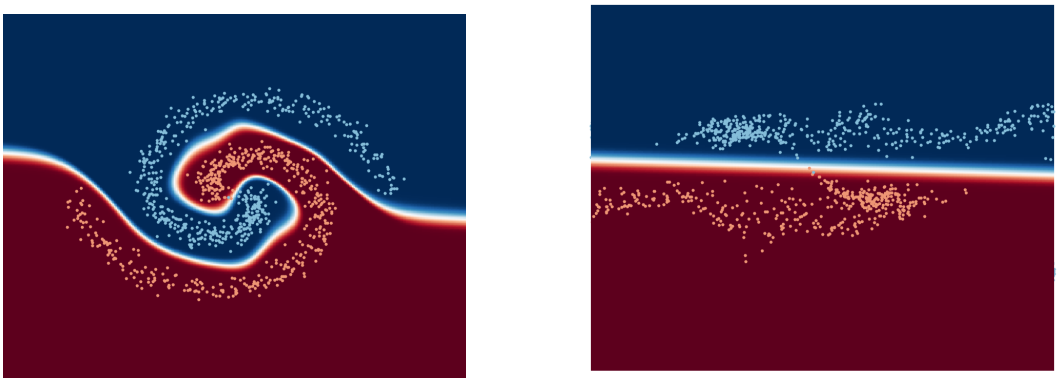

Figure 1: Klassifisering av punkter (testdata) fra hele planet, testdata er representert som et kontinuerlig fargespektrum, og treningsdata ved punkter. Venstre plott viser koordinater til utgangsposisjon og høyre plott til transformerte posisjoner

- Konvergens. vi søker etter å minimere $\mathcal{J}$ med hensyn på $\mathbf{U}$, så det kan være interessant å plotte hvordan $\mathcal{J}(\mathbf{U}^{(j)})$ utvikler seg som funksjon av iterasjonsindeksen j.

5.2 Tips angående punkt 4

Et par av de samme presentasjonene for punkter som nevnt i forrige avsnitt kan brukes også her. Som nevnt i avsnitt 4.2 så trener du modellen med data fra filene

`train-images-idx3-ubyte.gz` og `train-labels-idx1-ubyte.gz`.

Etterpå validerer du med funne verdier for parametrene på testsettet fra

`t10k-images-idx3-ubyte.gz` og `10k-labels-idx1-ubyte.gz`.

Du kan måle suksessrate både på treningssettet og testsettet, man forventer selvsagt høyere rate på treningssettet. Klarer du å få begge ratene mellom 90 og 100 prosent så har du gjort det bra.

Du kan varierer på forskjellige ting, f.eks. hvilke to siffer du vil klassifisere, hvor mange lag du bruker i modellen etc. Merk at simuleringene her kan ta ganske lang tid og det anbefales å bruke Adam-optimering for å øke konvergensen av algoritmen.

I tillegg til suksessrater kan man også her plotte konvergens av objektfunksjonen $\mathcal{J}$ som funksjon av iterasjonsindeksen.

A Utledning av gradientformler

I Matematikk 2 lærer vi om retningsderiverte av funksjoner av flere variable. La $H : \mathbb{R}^m \to \mathbb{R}$ være en funksjon av m variable. La $\mathbf{x}$ og $\boldsymbol{\delta}$ være vektorer i $\mathbb{R}^m$. Den retningsderiverte av H i punktet $\mathbf{x}$ i retning $\boldsymbol{\delta}$ kan beregnes som

$$\frac{d}{d\varepsilon}\bigg|_{\varepsilon=0} H(\mathbf{x}+\varepsilon\boldsymbol{\delta}) = \nabla H(\mathbf{x})\cdot\boldsymbol{\delta} = \langle \frac{\partial H}{\partial \mathbf{x}}(\mathbf{x}), \boldsymbol{\delta}\rangle$$

Den siste likheten viser bare en annen skrivemåte som vi skal bruke her. Vi skal finne den retningsderiverte av funksjonen $\mathcal{J}$ med hensyn på vekter og bias i lagene. La oss velge å se på den retningsderiverte i W_k langs δW_k. Men siden $\mathcal{J}$ avhenger bare indirekte (via $\mathbf{Y}_K$) av W_k, så starter vi med å se på den retningsderiverte av $\mathcal{J}$ med hensyn på en endring $\delta\mathbf{Y}_K$, dvs som ovenfor ser vi på

$$\delta\mathcal{J} = \frac{d}{d\varepsilon}\bigg|_{\varepsilon=0} \mathcal{J}(\mathbf{Y}_K+\varepsilon\delta\mathbf{Y}_K) = \langle \frac{\partial \mathcal{J}}{\partial \mathbf{Y}_K}, \delta\mathbf{Y}_K\rangle \tag{11}$$

Den våkne leser reagerer nå kanskje på at argumentet til funksjonen $\mathcal{J}$ er en matrise og ikke en vektor slik vi først forklarte med $H(\mathbf{x})$, men dette spiller ingen rolle, du kan alltids "pakke ut" f.eks. kolonnene i $\mathbf{Y}_K$ i en lang vektor og jobbe med denne istedet. Og vi skal se at det faktisk gjør formlene enklere å implementere dersom vi "beholder matriseformen". Inspirert av (7) kan vi introdusere $\mathbf{P}_K := \frac{\partial \mathcal{J}}{\partial \mathbf{Y}_K}$.

Nå kan vi fortsette å nøste, og se hva endringen $\delta\mathbf{Y}_K$ blir uttrykt ved $\delta\mathbf{Y}_{K-1}$ siden vi har formelen (1)

$$\mathbf{Y}_K = \mathbf{Y}_{K-1} + h\sigma(W_{K-1}\mathbf{Y}_{K-1}+b_{K-1}) \equiv \mathbf{Y}_{K-1} + hf(\mathbf{Y}_{K-1}; W_{K-1}, b_{K-1})$$

der vi for enkelhets skyld har definert $f(\mathbf{Y}; W, b) = \sigma(W\mathbf{Y}+b)$. Vi får da

$$\delta\mathbf{Y}_K = \left(I + h\frac{\partial f}{\partial \mathbf{Y}}(\mathbf{Y}_{K-1}; W_{K-1}, b_{K-1})\right)\delta\mathbf{Y}_{K-1}.$$

Dette kan vi sette inn i (11) og så husker vi på at for indreprodukt gjelder $\langle v, Aw\rangle = \langle A^T v, w\rangle$ der $v \in \mathbb{R}^n, w \in \mathbb{R}^m$ er vilkårlige vektorer og $A \in \mathbb{R}^{n\times m}$ en vilkårlig matrise. Vi får dermed at

$$\delta\mathcal{J} = \langle \mathbf{P}_K, \left(I + h\frac{\partial f}{\partial \mathbf{Y}}(\mathbf{Y}_{K-1}; W_{K-1}, b_{K-1})\right)\delta\mathbf{Y}_{K-1}\rangle = \langle \mathbf{P}_K + h\frac{\partial f}{\partial \mathbf{Y}}(\cdots)^T\mathbf{P}_K, \delta\mathbf{Y}_{K-1}\rangle$$

Vi definerer derfor generelt

$$\mathbf{P}_{k-1} = \mathbf{P}_k + h\frac{\partial f}{\partial \mathbf{Y}}(\mathbf{Y}_{k-1}; W_{k-1}, b_{k-1})^T\mathbf{P}_k. \tag{12}$$

Denne ligningen viser seg å ikke være noe annet enn (8) når vi setter inn definisjonen av $f(Y; W, b)$ ovenfor. Vi har kommet fram til at $\delta\mathcal{J} = \langle \mathbf{P}_{K-1}, \delta\mathbf{Y}_{K-1}\rangle$, og kan fortsette argumentet induktivt og vil da etterhvert komme til at

$$\delta\mathcal{J} = \langle \mathbf{P}_{k+1}, \delta\mathbf{Y}_{k+1}\rangle \tag{13}$$

Nå gjenstår bare desserten, vi har fra (1) at $\mathbf{Y}_{k+1} = \mathbf{Y}_k + hf(\mathbf{Y}_k; W_k, b_k)$ og nå er vi rede til å uttrykke $\delta\mathbf{Y}_{k+1}$ ved δW_k

$$\delta\mathbf{Y}_{k+1} = h\frac{\partial f}{\partial W}(\mathbf{Y}_k; W_k, b_k)\,\delta W_k$$

som vi setter inn i (13) og får

$$\delta\mathcal{J} = \langle h\frac{\partial f}{\partial W}(\mathbf{Y}_k; W_k, b_k)^T\mathbf{P}_{k+1}, \delta W_k\rangle$$

Da kan vi konkludere med at gradienten av $\mathcal{J}$ med hensyn på W_k er gitt som

$$\frac{\partial\mathcal{J}}{\partial W_k} = h\frac{\partial f}{\partial W}(\mathbf{Y}_k; W_k, b_k)^T\mathbf{P}_{k+1} \tag{14}$$

og der $\mathbf{P}_K$ er gitt ved (7) og der $\mathbf{P}_k,\ k < K$ beregnes fra (12). Helt tilsvarende fås naturligvis

$$\frac{\partial\mathcal{J}}{\partial b_k} = h\frac{\partial f}{\partial b}(\mathbf{Y}_k; W_k, b_k)^T\mathbf{P}_{k+1} \tag{15}$$

Vi må også verifisere de konkrete uttrykkene (7–10) som fås når vi setter inn uttrykket for $f(\mathbf{Y}; W, b)$ og bruker definisjonen av $\mathcal{J}$.

A.1 Uttrykket for $\frac{\partial\mathcal{J}}{\partial\mathbf{Y}_K}$

For å finne $\frac{\partial\mathcal{J}}{\partial\mathbf{Y}_K}$ undertrykker vi indeksen K og ser på

$$\frac{d}{d\varepsilon}\bigg|_{\varepsilon=0}\mathcal{J}(\mathbf{Y} + \varepsilon\delta\mathbf{Y}) = \frac{1}{2}\,\frac{d}{d\varepsilon}\bigg|_{\varepsilon=0}\langle\mathbf{Z}(\varepsilon) - \mathbf{c}, \mathbf{Z}(\varepsilon) - \mathbf{c}\rangle = \langle\mathbf{Z}(0) - \mathbf{c}, \dot{\mathbf{Z}}(0)\rangle$$

der $\mathbf{Z}(\varepsilon) = \eta((\mathbf{Y} + \varepsilon\delta\mathbf{Y})^T w + \mu\mathbf{1})$. Vi finner ved kjerneregelen at

$$\dot{\mathbf{Z}}(0) = \eta'(\mathbf{Y}^T w + \mu\mathbf{1})\,\delta\mathbf{Y}^T w$$

Da følger det at

$$\langle\mathbf{Z}(0) - \mathbf{c}, \dot{\mathbf{Z}}(0)\rangle = \langle w[(\mathbf{Z} - c)\odot\eta'(\mathbf{Y}^T w + \mu\mathbf{1})]^T, \delta\mathbf{Y}\rangle$$

og (7) følger. Hvis man er utrygg på denne siste overgangen så kan det være lurt å gjøre utregningen komponentvis

$$\begin{aligned}\sum_i(\mathbf{Z}(0) - \mathbf{c})_i\,\dot{\mathbf{Z}}(0)_i &= \sum_{i,j}(\mathbf{Z}(0) - \mathbf{c})_i\,\eta'((\mathbf{Y}^T w + \mu\mathbf{1})_i)\,\delta\mathbf{Y}_{ji}\,w_j \\ &= \sum_{j,i} w_j\left[(\mathbf{Z}(0) - \mathbf{c})\odot\eta'((\mathbf{Y}^T w + \mu\mathbf{1})\right]_i\ \delta\mathbf{Y}_{ji} \\ &\quad \langle w[(\mathbf{Z} - c)\odot\eta'(\mathbf{Y}^T w + \mu\mathbf{1})]^T, \delta\mathbf{Y}\rangle\end{aligned}$$

A.2 Utledning av (8) fra (12)

Sammenligner vi disse to ligningene så ser vi at det holder for oss å verifisere at

$$\frac{\partial f}{\partial \mathbf{Y}}(\mathbf{Y}_{k-1}; W_{k-1}, b_{k-1})^T \mathbf{P}_k = W_{k-1}^T \cdot [\sigma'(W_{k-1}\mathbf{Y}_{k-1} + b_{k-1}) \odot \mathbf{P}_k]$$

når $f(\mathbf{Y}; W, b) = \sigma(W\mathbf{Y} + b)$. Vi undertrykker lagindeksene $k, k-1$. Indeksene i det følgende står for 'komponenter' Vi beregner

$$\begin{aligned}
\langle \frac{\partial f}{\partial \mathbf{Y}}(\cdot)^T \mathbf{P}, \delta\mathbf{Y} \rangle &= \langle \mathbf{P}, \frac{\partial f}{\partial \mathbf{Y}}(\cdot)\delta\mathbf{Y} \rangle = \langle \mathbf{P}, \frac{d}{d\varepsilon}\Big|_{\varepsilon=0} f(\mathbf{Y} + \varepsilon\delta\mathbf{Y}; W, b) \rangle \\
&= \langle \mathbf{P}, \frac{d}{d\varepsilon}\Big|_{\varepsilon=0} \sigma(W(\mathbf{Y} + \varepsilon\delta\mathbf{Y}) + b) \rangle \\
&= \sum_{i,j,k} \mathbf{P}_{ij}\, \sigma'((W\mathbf{Y} + b)_{ij})\, W_{ik}\, \delta\mathbf{Y}_{kj} = \sum_{k,j} \delta\mathbf{Y}_{kj} \sum_i W_{ki}^T \left[P \odot \sigma'(W\mathbf{Y} + b)\right]_{ij} \\
&\langle W^T \left[\sigma'(W\mathbf{Y} + b) \odot \mathbf{P}\right], \delta\mathbf{Y} \rangle
\end{aligned}$$

A.3 (9) og (10)

Starter vi fra (14) og dropper igjen lagindeksene så finner vi

$$\begin{aligned}
\langle \frac{\partial \mathcal{J}}{\partial W}, \delta W \rangle &= h\langle \frac{\partial f}{\partial W}(\mathbf{Y}; W, b)^T \mathbf{P}, \delta W \rangle = h\langle \mathbf{P}, \frac{\partial f}{\partial W}(\mathbf{Y}; W, b)\delta W \rangle \\
&= h\langle \mathbf{P}, \frac{d}{d\varepsilon}\Big|_{\varepsilon=0} f(\mathbf{Y}; W + \varepsilon\delta W, b) \rangle = h\langle \mathbf{P}, \frac{d}{d\varepsilon}\Big|_{\varepsilon=0} \sigma((W + \varepsilon\delta W)\mathbf{Y} + b) \rangle \\
&= h \sum_{i,j,k} \mathbf{P}_{ij}\, \sigma'((W\mathbf{Y} + b)_{ij}) \delta W_{ik} \mathbf{Y}_{kj} = h \sum_{i,k} \delta W_{ik} \sum_j [\mathbf{P} \odot \sigma'((W\mathbf{Y} + b)]_{ij} \mathbf{Y}_{jk}^T \\
&= h\langle [\mathbf{P} \odot \sigma'((W\mathbf{Y} + b)]\mathbf{Y}^T, \delta W \rangle
\end{aligned}$$

som beviser (9). Vi dropper beviset av (10) som er helt tilsvarende.