Oppgaver Løsningsforslag - SOK3023

Maskinlæring for økonomer
Laget av Markus J. Aase

January 30, 2025

Dette dokumentet inneholder oppgaver studentene i SOK-3023 kan jobbe med etter tredje uka av kurset. En del av svarene vil dere kunne finne fra de fysiske forelesningene og/eller kompendiet. Mens noen spørsmål krever at dere oppsøker informasjonen selv i dokumentasjon til Tensorflow eller andre sted på internett.

1. **Kostfunksjon vs. Loss-funksjon**

   (a) **Forskjellen mellom kostfunksjon og loss-funksjon**
   Løsning:

   - En **loss-funksjon** (loss function) måler feilen for en enkelt observasjon i datasettet. Den kvantifiserer hvor langt en prediksjon $\hat{y}_i$ er fra den sanne verdien y_i.
   - En **kostfunksjon** (cost function) er en aggregert verdi av loss-funksjonen over hele datasettet. Vanligvis beregnes dette ved å ta gjennomsnittet av alle individuelle tap (loss values) i treningssettet.

   Matematisk kan vi skrive:

   $$\text{Loss-funksjon: } L(y_i, \hat{y}_i)$$

   $$\text{Kostfunksjon: } C(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i)$$

   hvor $C(\theta)$ er kostfunksjonen for parametrene θ, basert på alle n datapunktene.

   (b) **Eksempler på kostfunksjoner for ulike typer maskinlæringsproblemer**
   Løsning:

   - For **regresjon**: Vanlige kostfunksjoner inkluderer Mean Squared Error (MSE) og Mean Absolute Error (MAE):

   $$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2,$$

   $$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|.$$

- For **klassifikasjon**: En vanlig kostfunksjon er kryssentropi (Cross-Entropy Loss):

$$C(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right].$$

NB: Vit at man bruker ulike varianter av Cross-Entropy, avhengig om man gjør binær klassifikasjon (altså, 0 eller 1) eller multiklasse-klassifikasjon (f.eks. hund, katt, esel eller hest).

2. **Hensikten med en kostfunksjon**

(a) **Hva er hovedformålet med en kostfunksjon i maskinlæring?**

Løsning: En kostfunksjon måler hvor godt eller dårlig en modell presterer ved å kvantifisere differansen mellom de sanne verdiene y_i og de predikerte verdiene $\hat{y}_i$. Målet med en kostfunksjon er å gi modellen en numerisk verdi å minimere under trening.

(b) **Hvordan hjelper en godt definert kostfunksjon en modell med å forbedre seg over tid?**

Løsning: Ved å minimere kostfunksjonen under trening kan modellen gradvis justere parametrene sine for å forbedre prediksjonene. Gradient descent eller andre optimaliseringsalgoritmer brukes til å finne de beste parametrene (altså, vekter og biaser: $w_1,\ w_2, \ldots, w_n, b$) som gir lavest mulig kost. Samtidig er det viktig at man definerer en kostfunksjon som passer med dataene og ønsket output (avhengig om man driver med regresjon eller klassifikasjon).

(c) **Hvordan kan valg av kostfunksjon påvirke en modells ytelse?**

Løsning: Valg av kostfunksjon påvirker hvordan modellen håndterer ulike typer feil. For eksempel:

- MSE straffer store feil mer enn MAE, noe som kan gjøre den mer følsom for "uteliggere" (outliers).
- Kryssentropi fungerer godt for klassifikasjonsproblemer ved å skille mellom sannsynligheter.
- Huber-loss kombinerer fordelene med MSE og MAE ved å være robust mot outliers.
- ...

3. **Gradient Descent – Grunnleggende Konsept**

(a) **Hva er gradient descent og hvorfor brukes det i maskinlæring?**

Gradient descent er en optimaliseringsalgoritme som brukes til å minimere en *loss-funksjon* ved å justere modellens parametere gradvis, altså vektene og biasene. Algoritmen beregner gradienten (den deriverte) av loss-funksjonen og oppdaterer modellens parametere i retningen som reduserer feilen mest. Altså, minimerer kost-funksjonen.

I maskinlæring brukes gradient descent fordi:

- Det er en effektiv metode for å finne optimale parametere i modeller med mange variabler.
- Det muliggjør iterativ læring, noe som er nødvendig for komplekse modeller som nevrale nettverk.
- Det hjelper med å minimere feilen mellom modellens prediksjoner og de faktiske verdiene.

Oppdateringsregelen for gradient descent er gitt ved:

$$\mathbf{w} := \mathbf{w} - \alpha \cdot \nabla C(\mathbf{w})$$

hvor:

- $\mathbf{w}$ er modellens parametere.
- α er læringsraten, som bestemmer steglengden for oppdateringene.
- $\nabla C(\mathbf{w})$ er gradienten av loss-funksjonen.

(b) **Forskjellen mellom batch gradient descent, stokastisk gradient descent (SGD) og mini-batch gradient descent**

Det finnes tre hovedvarianter av gradient descent, basert på hvor mange treningsdata som brukes i hver oppdatering:

- **Batch Gradient Descent:** Bruker hele treningssettet til å beregne gradienten og oppdaterer vektene én gang per epoch.
  - **Fordeler:** Presise gradientberegninger, jevn konvergens.
  - **Ulemper:** Treg/ineffektiv for store datasett, krever mye minne.
- **Stokastisk Gradient Descent (SGD):** Oppdaterer vektene etter hver enkelt datapunkt i treningssettet.
  - **Fordeler:** Krever lite minne, raskere oppdateringer, kan unngå lokale minimum.
  - **Ulemper:** Kan være ustabil, større variasjon i gradientene.
- **Mini-Batch Gradient Descent:** En mellomting mellom batch GD og SGD – bruker en liten delmengde (batch) av dataene per oppdatering.
  - **Fordeler:** Balanse mellom effektivitet og stabilitet, fungerer godt med GPU-er.
  - **Ulemper:** Mindre nøyaktig enn batch GD, men mer stabil enn SGD.

Oppsummering:

- *Batch Gradient Descent:* Bruker hele datasettet, gir presise oppdateringer, men er tregt og krever mye minne. Da kan vi skrive

```
history = model.fit(x_train, y_train,
                    epochs=20,
                    batch_size=len(x_train), # GD
                    validation_data=(x_val, y_val),
                    callbacks=[early_stopping])
```

- *SGD:* Oppdaterer etter hvert enkelt datapunkt, raskt men ustabilt.

```
history = model.fit(x_train, y_train,
                    epochs=20,
                    batch_size=1, # SDG
                    validation_data=(x_val, y_val),
                    callbacks=[early_stopping])
```

- *Mini-Batch Gradient Descent:* Kombinerer fordelene ved begge, og brukes oftest i praksis.

```
history = model.fit(x_train, y_train,
                    epochs=20,
                    batch_size=32, # mini-batch
                    validation_data=(x_val, y_val),
                    callbacks=[early_stopping])
```

En ofte brukt 'optimizer' vi bruker i TensorFlow er ADAM - Adaptive Moment Estimation. Som vi ofte skriver i før 'model.fit', slik:

```
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
```

Les mer om ADAM og versjoner av Gradient Descent i Tensorflow sin dokumentasjon.

4. **Gradient Descent – Regneoppgave**

Gitt funksjonen:

$$C(w_1, w_2) = w_1^2 + 2w_1w_2 + w_2^2$$

1. Gradientberegning

Gradienten av $C(w)$ er gitt ved:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \end{bmatrix} = \begin{bmatrix} 2w_1 + 2w_2 \\ 2w_2 + 2w_1 \end{bmatrix}$$

2. Ett steg med gradient descent

Vi starter med de initielle verdiene til $w^{(0)}$, og læringsraten α:

$$w^{(0)} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \quad \alpha = 0.1.$$

Beregning av gradient:

$$\nabla C(2, -1) = \begin{bmatrix} 2 \cdot 2 + 2 \cdot -1 \\ 2 \cdot -1 + 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

Oppdatering av vektene:

$$w^{(1)} = w^{(0)} - \alpha \cdot \nabla C(w^{(0)})$$

$$= \begin{bmatrix} 2 \\ -1 \end{bmatrix} - 0.1 \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

$$= \begin{bmatrix} 2 - 0.2 \\ -1 - 0.2 \end{bmatrix} = \begin{bmatrix} 1.8 \\ -1.2 \end{bmatrix}$$

Svar: Etter ett steg med gradient descent er de oppdaterte vektene:

$$w_1^{(1)} = 1.8, \quad w_2^{(1)} = -1.2.$$

Gjenta prosessen flere ganger, så vil det gå mot et lokalt eller globalt minimum. Prøv å gjøre det et par steg selv, tegn opp grafen i et 3D verktøy - så kan du visualisere selv hva som faktisk skjer her!

5. **Tilbakepropagering (Backpropagation) - for den interesserte**

1. Hva er tilbakepropagering, og hvorfor er det viktig i nevrale nettverk?

Tilbakepropagering er en algoritme som brukes til å trene nevrale nettverk ved å justere vektene basert på feilen mellom prediksjoner og sanne verdier.

Hvorfor viktig?

- Det gjør læring mulig ved å beregne hvordan hver vekt påvirker feilen.
- Effektiv for å optimalisere nevrale nettverk ved hjelp av gradient descent.
- En metode som beregner en meget komplisert gradient av kostfunksjonen, $\nabla C(\mathbf{w})$,som ofte kan ha flere tusen parametere (altså vekter og biaser).

2. Hvordan brukes kjerneregelen (chain rule) for å beregne gradientene?

Kjerneregelen brukes til å beregne hvordan endringer i vektene påvirker feilverdien ved å ta deriverte gjennom de ulike lagene i nettverket.

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \cdot \frac{\partial a^{(L)}}{\partial z^{(L)}} \cdot \frac{\partial C_0}{\partial a^{(L)}}$$

- C_0 er tapet (loss), som representerer forskjellen mellom modellens prediksjoner og de faktiske resultatene.
- $w^{(L)}$ er vektene i det L-te laget, som bestemmer styrken på forbindelsene mellom nevronene i laget.
- $z^{(L)}$ er den lineære kombinasjonen av inputtene og vektene i laget L, dvs. $z^{(L)} = W^{(L)}a^{(L-1)} + b^{(L)}$, hvor $a^{(L-1)}$ er aktiveringsverdiene fra forrige lag, og $b^{(L)}$ er bias.
- $a^{(L)}$ er aktiveringen i laget L, beregnet ved å bruke en aktiveringsfunksjon (for eksempel ReLU, sigmoid eller tanh) på $z^{(L)}$.

3. Hvordan oppdateres vektene i et nevralt nettverk?

Vektene oppdateres ved å bruke gradient descent:

$$\mathbf{w} := \mathbf{w} - \alpha \cdot \nabla C(\mathbf{w})$$

hvor α er læringsraten.

Tilbakepropagering beregner gradienten av tapsfunksjonen ($\nabla C(\mathbf{w})$) med hensyn til vektene i nettverket for et enkelt input–output-eksempel. Dette gjøres på en effektiv måte ved å beregne gradienten lag for lag, ved å iterere baklengs fra det siste laget for å unngå unødvendige beregninger av mellomliggende termer i kjerneregelen. Denne prosessen utledes oftest ved hjelp av dynamisk programmering, og er noe Tensorflow løser for oss.

Steg for tilbakepropagering:

(a) Beregn feilen i utgangslaget.

(b) Bruk kjerneregelen til å spre feilen bakover.

(c) Oppdater vektene basert på gradientene.

Dette gjentas over flere *epochs* til modellen lærer. En intuitiv, dypere forklaring, kan finnes i *anbefalte læringsressurser*, lenke her: `https://www.3blue1brown.com/lessons/backpropagation-calculus`

6. **Forstå Keras-kode**

Nedenfor er en kode for å definere en sekvensiell modell i TensorFlow's Keras API. Beskriv hva hver linje gjør og hvorfor den er viktig.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dropout, Dense
import tensorflow as tf

model = Sequential([
    tf.keras.Input(shape=(28, 28)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.25),
    Dense(10, activation='softmax')
])
```

1. Hva er hensikten med `Flatten()`-laget?

`Flatten()` lager en 1D-vektor (av lengde 784, i dette tilfelle) fra den 2D-inngangen (for eksempel et 28x28 bilde). Dette er nødvendig før et fullt tilkoblet (dense) lag, som krever en vektor som input.

2. Hvorfor brukes `Dropout()` i modellen?

`Dropout()` er en regulariseringsteknikk som tilfeldig setter noen av nevronene til null under trening, for å forhindre overtilpasning (overfitting). Dette tvinger modellen til å lære mer robuste representasjoner.

3. Hva gjør `Dense(10, activation='softmax')` i denne modellen?

`Dense(10, activation='softmax')` er det siste laget i modellen. Det har 10 nevroner (én for hver klasse i en 10-klasses klassifisering), og bruker `softmax` for å beregne sannsynlighetene for hver klasse.

7. **Endre og tilpasse modellen**

1. Hvordan ville du modifisert modellen slik at den har to skjulte lag med henholdsvis 256 og 128 nevroner?

Du kan modifisere modellen som følger:

```
model = Sequential([
    tf.keras.Input(shape=(28, 28)),
    Flatten(),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

Her er to skjulte lag lagt til med 256 og 128 nevroner, begge med `relu` som aktiveringsfunksjon.

2. Endre aktiveringsfunksjonen i det første skjulte laget fra `relu` til `sigmoid`. Hvordan påvirker dette modellens evne til å lære?

Bytt `activation='relu'` til `activation='sigmoid'` i det første skjulte laget. `Sigmoid` gir et utgangsområde mellom 0 og 1, mens `ReLU` kan ha større gradienter, noe som kan føre til raskere læring. Bruken av `sigmoid` kan gjøre treningen langsommere og vanskelige å lære i noen tilfeller, spesielt ved dype nettverk på grunn av problemer som forsvinnende gradienter. Dette kalles *vanishing gradient*, men det er ikke pensum i dette kurset.

3. Hva skjer hvis vi fjerner dropout-lagene? Hvordan kan det påvirke modellens ytelse?

Hvis `Dropout()`-lagene fjernes, kan modellen lettere overtilpasse treningsdataene, spesielt på små datasett eller komplekse modeller. Dette kan føre til dårlig generalisering på nye data (lavere ytelse på testsettet).