TEK300 Maskinlæring Hjemmeeksamen – Gruppe 7

Kandidatnummer: 700020 & 700032

Introduksjon

Denne oppgaven gikk ut på å gjenkjenne siffer fra MNIST datasettet. MNIST datasettet er en stor database med håndskrevne tall som vanligvis blir brukt for å trene opp maskinlærings algoritmer for bildegjenkjenning. Gjennom oppgaven benytter vi forskjellige algoritmer og forklarer hvordan de fungerer og hvilke resultater vi får.

Vi tenkte at for å fullføre oppgaven, så var første steg å researche forskjellige maskinlæring algoritmer og rammeverk. Gjennom vår research kom vi fram til flere ulike rammeverk og algoritmer, blant rammeverkene var det: **Keras, TFLearn, Tensorflow og scikit-learn** og blant algoritmene var det: **artificial neural network(ANN), K-Nearest Neighbour(K-NN), og Convolutional neural network(CNN).**

Vi skal utover oppgaven sammenlikne resultatene og effektiviteten med bruken av disse forskjellige algoritmene. Vi vil også diskutere prosessen hvordan vi gikk fram til resultatet og sluttresultatet. Kode vil også bli diskutert, og resultatene vi kom fram til gjennom skjermbilder.

Hvordan bruke programmet

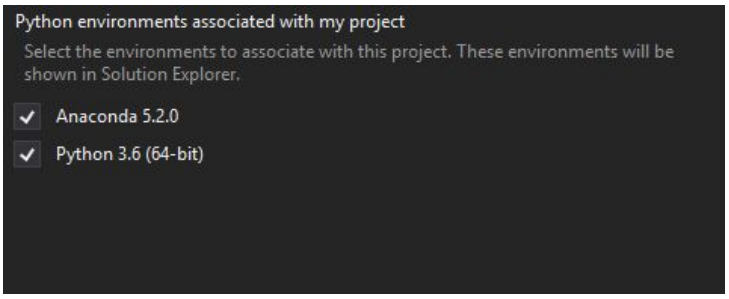
Før man kjører algoritmene i python, må man gjøre klar PCen med nedlasting av biblioteker og rammeverk. Vi vil bruke spesifikke versjoner av disse bibliotekene. Vi har laget en tekstfil kalt requirements.txt som vil laste ned alle versjonene som ble brukt til maskinlæringen.

Når du har tekstfilen lokaliser den med CD ved kommandolinjen å naviger deg der. Etter det installer bibliotekene med følgende kommando: **pip install -r requirements.txt eller kjør pip install "bibliotek pakke" hvis det ikke fungerer.**
requirements.txt:

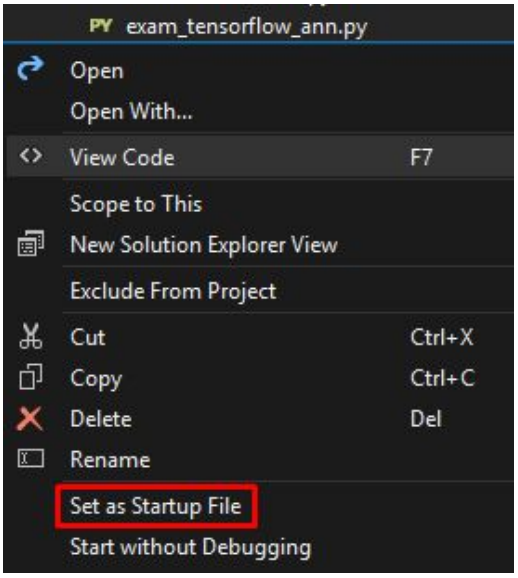
```
image==1.5.20
numpy==1.14.3
tensorflow==1.4.0
keras==2.0.0
opencv-python
```

Merk: Med tanke på at vi benytter Visual Studio 2017 på forhånd, benytter vi også anaconda 5.2.0 pakken som inkluderer bibliotekene: numpy, matplotlib, med mer.

Dette aktiverer man med å høyre klikke "Python Enviorments" på prosjekt filene å velge "Add/Remove Python Environments"



Når man har installert alle requirements og fått åpnet python prosjektet i Visual Studio er det viktig å høyreklikke på python filen man vil kjøre og velge **"Set as Startup File"**. Dette fører til at kun denne python filen kjører.



Prosess

Research:
Ettersom vi hadde gjort akkurat samme oppgaven i forelesning valgte vi å gjøre denne igjen med Tensorflow og nevralt nett. Vi startet først med å bruke en allerede-eksisterende løsning som mal, men vi endret og tilpasset den til bedre bruk for å gjøre den mer forståelig.

Etter å ha fullført det nevrale nettverket med TensorFlow gikk vi videre for å sette opp noe lignende med scikit-learn. For å få dette til å fungere så fant vi en vel-lagd video som både forklarte virkemåte og teorien bak hvordan alt fungerer. Deretter bygde vi det opp og fikk det til å fungere på en tilfredsstillende måte.

Vi testet ut en ny algoritme. Vi brukte igjen scikit-learn, men bruker nå heller en K-NN (K-nærmeste naboer) algoritme.

Til slutt testet vi en helt ny algoritme CNN. Der vi benyttet TensorFlow og Keras som rammeverk. Dette var de løsningene som var mest kompleks for oss å sette opp og forstå, men etter god lesning kom vi fram til et bra resultat.

Problemstillinger:

Et par problemstillinger vi støttet på var nedlastinger av biblioteker og bruk av kode som ikke fungerte gjennom researchen våres. Vi måtte redusere mengde rammeverk vi kunne benytte, inkludert TFLearn.

Men bortsett fra det var selve prosessen ved research enkel med tanke på at det finnes mye materiale og eksempler for MNIST siffer gjenkjennings metoder på nett.

Endringer:

Noen endringer vi gjorde var variabel navn og verdier til TensorFlow nevral nett koden. Vi valgte å endre den så mye som mulig for å gjøre den ulik fra den originale. Slik at vi kan vise vår kode forståelse. Selv om vi gjennom testing og endring av verdiene føler at de gjorde koden så optimal og riktig som mulig.

Eksempel på kode endring av tensorflow nevralt nett løsningen:

```
#Three hidden layers
n_hid1 = 1000
n_hid2 = 250
n_hid3 = 100

n_iterations = 1000
n_batch_size = 100
dropout = 0.6
```

Vi fjerner keep_prob og n_output og n_input siden disse har en fiksert verdi som ikke endres. Vi fjerner også learning_rate siden, den ble ikke engang benyttet gjennom koden, og selv om den ble benyttet var det bare en engangsbruk:

```
X = tf.placeholder("float", [None, 784])
Y = tf.placeholder("float")
```

Her kunne learning_rate variabelen bli brukt.

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y, logits=output_layer))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

Ved bruken av samme algoritme, men med en annen kode struktur og rammeverk(scikit-learn) fant vi fram til en mer enkelt bygd opp kodestruktur. Denne endret vi utrolig mye på for å gjøre den mer effektiv som til slutt ble våres:

Original iterasjon av sklearn nevralt nett løsningen:

```
import numpy as np
import matplotlib.pyplot as pt
import pandas as pd
from sklearn.tree import DecisionTreeClassifier

data = pd.read_csv("mnist_train.csv").as_matrix()
clf = DecisionTreeClassifier()

xtrain = data[0:21000,1:]
train_label = data[0:21000,0]

clf.fit(xtrain, train_label)

# testing data
xtest = data[21000:,1:]
actual_label = data[21000:,0]

p = clf.predict(xtest)

count = 0

for i in range(0,21000):
    count+=1 if p[i] == actual_label[i] else 0
print("Accuracy: ", (count/21000) *100)
```

Kode utvidelse av sklearn nevralt nett løsningen:

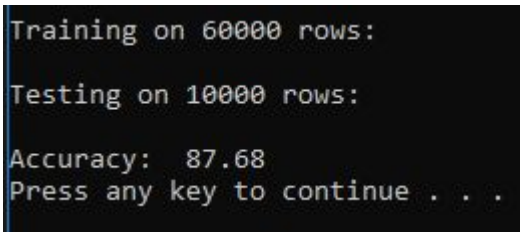
```
import numpy as np
import matplotlib.pyplot as pt
import pandas as pd

from sklearn.tree import DecisionTreeClassifier

in_data_train = pd.read_csv("mnist_train.csv").as_matrix()
in_data_test = pd.read_csv("mnist_test.csv").as_matrix()

n_train_rows = 60000 #Ammount of rows in mnist_train.csv dataset
n_test_rows = 10000 #Ammount of rows in mnist_test.csv dataset
n_correct_predictions = 0 #Correct prediction iterator

clf = DecisionTreeClassifier()

#Training
print("\nTraining on", n_train_rows, " rows: \n")
training_config = in_data_train[0:n_train_rows,1:]
train_label = in_data_train[0:n_train_rows,0]
clf.fit(training_config, train_label)

#Testing
print("\nTesting on", n_test_rows, " rows: \n")
testing_config = in_data_test[0:,1:]
actual_label = in_data_test[0:,0]
prediction = clf.predict(testing_config)

#Predicting
for i in range(0,n_test_rows):
      n_correct_predictions+=1 if prediction[i] == actual_label[i] else 0

#Print the final result
print("Accuracy: ", (n_correct_predictions/n_test_rows)*100)
```

Kode endringen på sklearn løsningen økte vår accuracy verdier mye mer også som beviser økt ytelse.

Original iterasjon accuracy:

```
Accuracy:  83.27619047619048
Press any key to continue . . .
```

Utvidet iterasjon accuracy:



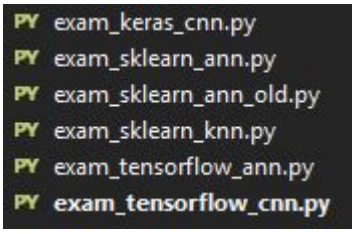
Som vi ser her har vi økt vår accuracy med minst 4 - 5 prosent. Det er inkludert en gammel iterasjon på python kalt **"exam_sklearn_ann_old"** slik at det kan testes ved eget bruk.

Konklusjon og Resultater

Det som gjør de løsningene våre forskjellige er så klart rammeverket, bibliotekene, algoritmene og generelt bare hvordan de fungerer. Noe vi satte fokus på i starten var å få testet ut en rekke forskjellige rammeverk. I stedet for å kun bruke TensorFlow benyttet vi i tillegg Keras. Vi valgte også å benytte oss av scikit-learn biblioteket i noen av løsningene våre.

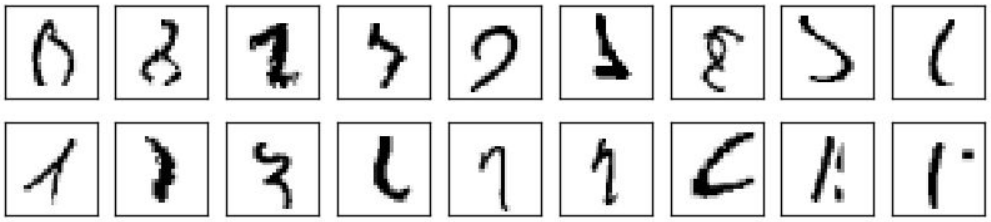
Ettersom at hovedpoenget i oppgaven var å teste ut forskjellige algoritmer så har vi også satt stort fokus på dette. Det er så klart store forskjeller mellom de forskjellige algoritmene, men å forklare det helt perfekt vil kreve for mye tid og tekst. Vi har dermed besluttet å kort beskrive algoritmene vi har brukt og hvordan den fungerte i praksis når vi testet de på MNIST datasettet med de forskjellige maskinlæring algoritmene.

Kode vi skal gå igjennom:



K-Nearest Neighbors(KNN):
Når vi prøver å finne ut av en verdi til en data ser vi på de nærmeste verdiene av dataene(neighbour) som utgangspunkt for å definere hvilken gruppe den tilhører. Algoritmen trenger ikke nødvendigvis ha et klart svar på hva som er riktig, men den ser rett og slett på hva som gir mest mening.

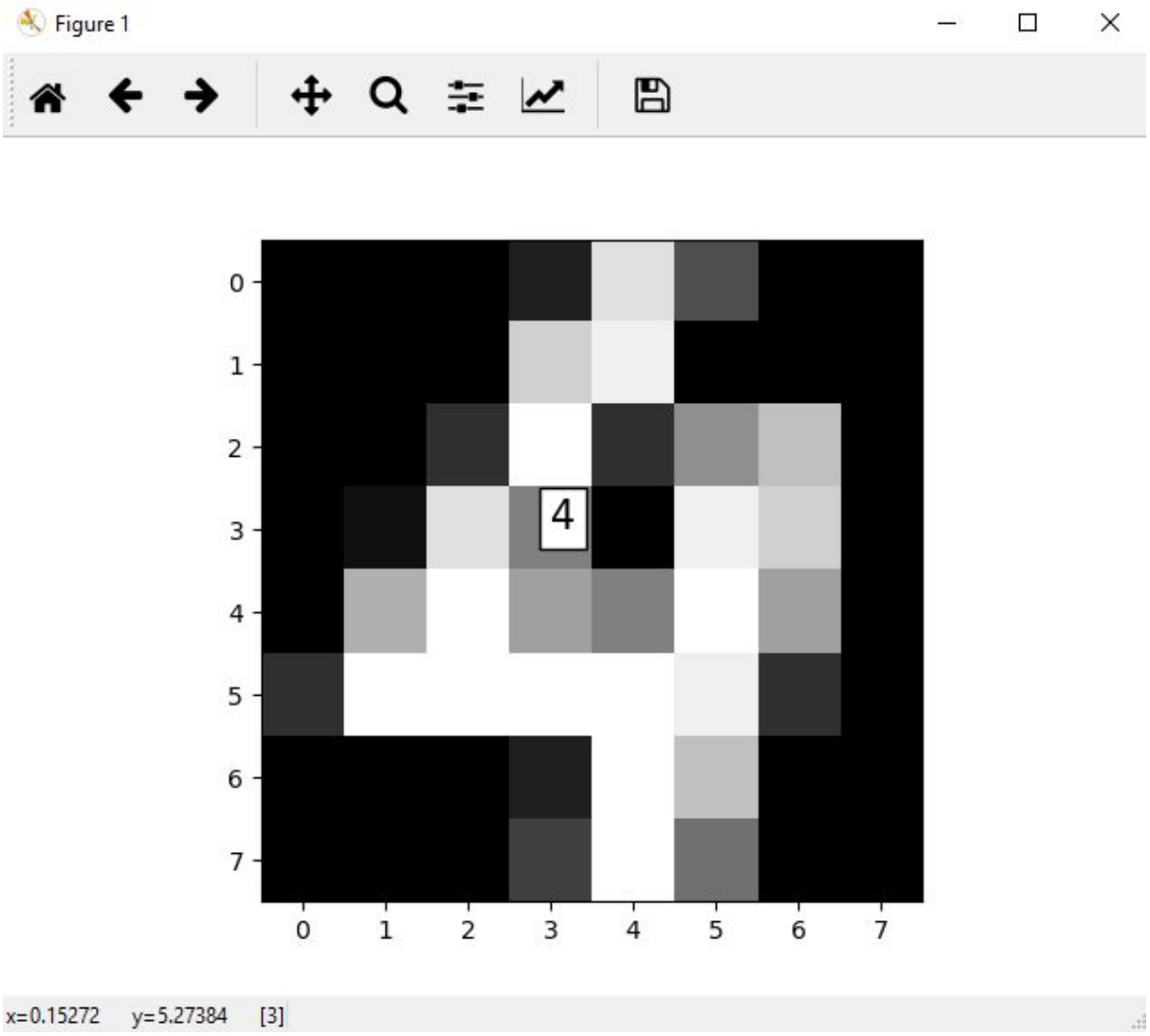
Dette konseptet brukes for tall bildene fra MNIST datasettet. En av problemene er at noen bilder kan være vanskelig for algoritmene å identifisere hvis bildet har mye støy for eksempel:



Men siden vi benytter KNN algoritmen lærer vi at den motvirker denne vanskeligheten mye. Er vi heldig så kan algoritmen få veldig imponerende accuracy score. Vi lærer at selve algoritmen er en lat lære algoritme, og vi burde benytte andre algoritmer hvis vi skal finne fram til mer nøyaktig resultater enn å gjenkjenne 10 bilder av tall.

Resultat etter kjøring av exam_sklearn_knn.py

```
training data points: 1293
validation data points: 144
testing data points: 360
k=1, accuracy=97.92%
k=3, accuracy=97.92%
k=5, accuracy=97.92%
k=7, accuracy=97.22%
k=9, accuracy=97.92%
k=11, accuracy=97.22%
k=13, accuracy=97.22%
k=15, accuracy=96.53%
k=17, accuracy=95.83%
k=19, accuracy=96.53%
k=21, accuracy=96.53%
k=23, accuracy=96.53%
k=25, accuracy=96.53%
k=27, accuracy=95.83%
k=29, accuracy=96.53%
k=1 achieved highest accuracy of 97.92% on validation data
EVALUATION ON TESTING DATA
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        31
           1       0.95      1.00      0.98        42
           2       1.00      1.00      1.00        18
           3       0.97      0.95      0.96        41
           4       1.00      1.00      1.00        37
           5       0.98      0.95      0.96        42
           6       0.98      0.98      0.98        41
           7       1.00      1.00      1.00        38
           8       0.97      0.97      0.97        38
           9       0.97      0.97      0.97        32

avg / total       0.98      0.98      0.98       360

Confusion matrix
[[31  0  0  0  0  0  0  0  0  0]
 [ 0 42  0  0  0  0  0  0  0  0]
 [ 0  0 18  0  0  0  0  0  0  0]
 [ 0  0  0 39  0  1  0  0  1  0]
 [ 0  0  0  0 37  0  0  0  0  0]
 [ 0  0  0  0  0 40  1  0  0  1]
 [ 0  1  0  0  0  0 40  0  0  0]
 [ 0  0  0  0  0  0  0 38  0  0]
 [ 0  1  0  0  0  0  0  0 37  0]
 [ 0  0  0  1  0  0  0  0  0 31]]
i think tha digit is : 4
```



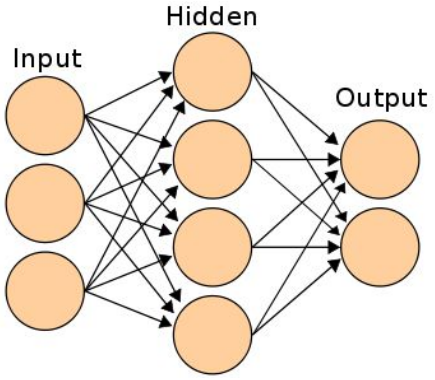
Det som skjer her da er at vi går gjennom datasettet, treningen og testingen flere ganger. Når vi har gjort det ferdig så får vi sett hvem av gangene den var mest nøyaktig. I eksempelet over viser det seg altså at k=1 (og k=3, k=5, k=9) er den mest nøyaktige med 97.92%. Dette er med andre ord en ganske nøyaktig gjenkjenning, men kunne så klart blitt bedre. 2 feil av 100 forsøk er absolutt ikke ille. I vår løsning så prøver algoritmen å predikere tallet 4, og det får den til som vist i bildet ovenfor.

Denne algoritmen tilfredsstiller resultater i forhold til problemstillingen. Den scorer høyt i accuracy, og kjøres kjapt igjennom maskinen uten å kreve store mengder ressurser. Vi tar fremdeles hensyn til det faktum at algoritmen er lat, og kan skape verre resultater for mer komplekse problemstillinger enn tallpredikering.

Artificial neural network(ANN):
Artificial neural network er et kunstig nevralt nettverk som er bygget opp av celler(nevroner) og synapser. Lagene vi trenger til et nevralt nett er inputlag, outputlag og *n* antall skjulte lag(hidden layers) i midten. Vår input er på 784 på grunn av at våre bilder er på 28x28 pixler, mens vår output er på 10 mulige resultater som utgir et array på 10. Ved å gi løsningen våres data genereres output der treningen foregår ved å styrke vektingen til koplinger som

var med på å generere riktig resultat, og svekke vektingen til de som var med på å gi feil resultat(bilder med støy).

Nevralt nettverk input, output og hidden layer eksempel:



Vi gjør klar verdiene for hidden layers, input og output og mer for å kjøre gjennom MNIST datasettet igjen. Denne gangen så modellerer vi litt mer kompleks enn KNN, med å bygge opp vårt nettverk av nevroner og synapser.

Gjennom testingene lærer vi at ANN krever mer krefter for å gi ut sine resultater. Når vi kjørte det gjennom keras rammeverket tok det rundt 3 - 4 minutter for hver iterasjon(epoch). Vi lærer også at algoritmen i vårt nevrale nett bedrer resultatene sine etterhvert som tiden går. Siden den øker sin kunnskap når den går over iterasjonene. Den første iterasjonen er alltid dårlig enn den siste.

Vi forklarer videre resultater av ANN algoritmen når vi tester det ut med flere rammeverk:

Resultat etter kjøring av exam_sklearn_ann.py og exam_sklearn_ann_old.py (som vist før)

```
Training on 60000 rows:

Testing on 10000 rows:

Accuracy:  87.68
Press any key to continue . . .
```

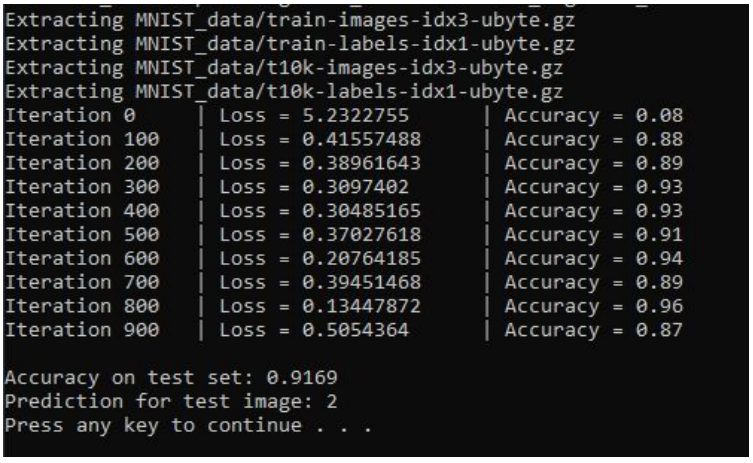
```
Accuracy:  83.27619047619048
Press any key to continue . . .
```

Som man ser på bildet over så gir exam_sklearn_ann.py oss en accuracy på 87.68% etter trening på 60.000 rader og testing på 10.000. Man kan fort tenke at >87% er en god score, men det er det altså ikke i dette eksempelet. Hvis man først vil bruke maskinlæring til å lese tall, noe som da f.eks. kan bli brukt i forhold til penger så er det dumt om 12 av 100 tall er feil. 100,000 i gjeld kan da brått bli til 700,000 i gjeld. Osloveien 1 blir Osloveien 7 på leveringen fra Posten.

Som vi ser så er den gamle versjonen *(exam_sklearn_ann_old.py)* enda litt verre. Her får vi kun litt over 83% i accuracy. I noen scenarioer, f.eks. bildegjenkjenning av mer komplekse ting så kunne dette vært en relativt bra score. I forhold til vår oppgave er det derimot ikke så

bra. Å kun få riktig på 83/100 tall er rett og slett dårlig. Det kunne så klart vært verre, men det endrer ingenting. Denne løsningen burde ikke brukes til noe annet enn å lære og teste.

Resultat etter kjøring av exam_tensorflow_ann.py



Ved kjøring av exam_tensorflow_ann.py ser vi da at etter 1000 iterasjoner så sitter vi igjen med en accuracy på 91.69% på test datasettet vårt. Absolutt ikke helt optimalt. Det vil da si at over 8 av 100 tall blir gjenkjent feil. Vi kunne nok muligens fått en bedre accuracy med å fore inn enda mer treningsdata.

Denne løsningen klarer nesten alltid å predikere at et test bildet er det samme som 2, vi går utifra at lav accuracy har mye med støy bilder å gjøre. Den er ikke ubrukelig i forhold til nøyaktighet, men vi føler at vi kan skape mye bedre resultater med en annen form for rammeverk og algoritme.

Vi lærer at algoritmen er tidsfordrivende, men skaper gode resultater som tilfredsstiller problemstillingen. Det å kjøre test dataene og trenings dataene gjennom iterasjoner flere ganger øker vår accuracy resultater. Når vi kjører gjennom hele datasettet i et kjør som vist fra exam_sklearn_ann.py og exam_sklearn_ann_old.py får vi ikke like gode forventet resultater.

Denne algoritmen kan vi se for oss som brukbar og fornuftig i forhold til mer komplekse bildegjenkjennings oppgaver.

Convolutional neural network(CNN):

Et Convolutional neural network (CNN) er et flerlags nevralt nettverk med en spesiell arkitektur for å oppdage komplekse funksjoner i data. Denne algoritmen kan bli brukt i bildegjenkjenning, selvkjørende biler og mer.

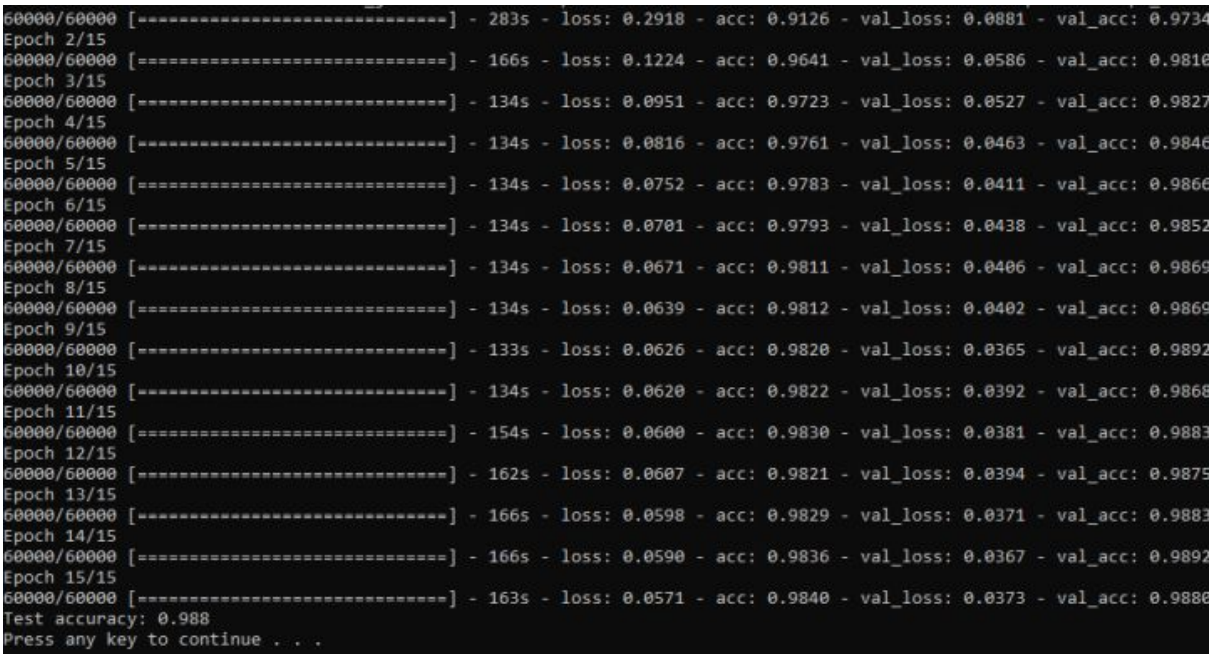
Vi importerer nye biblioteket fra keras som vi ikke er helt kjent med, men lærer mye ut fra resultatene den generer. I denne algoritmen bruker vi to forskjellige kodesnutter å genererer ulike resultater.

Resultat etter kjøring av exam_tensorflow_cnn.py

```
0.8976
step 0, training accuracy 0.1
step 100, training accuracy 0.86
step 200, training accuracy 0.92
step 300, training accuracy 0.9
step 400, training accuracy 0.96
step 500, training accuracy 0.98
step 600, training accuracy 0.92
step 700, training accuracy 0.96
step 800, training accuracy 0.96
step 900, training accuracy 0.96
step 1000, training accuracy 0.94
step 1100, training accuracy 0.92
step 1200, training accuracy 0.98
step 1300, training accuracy 0.98
step 1400, training accuracy 0.96
test accuracy 0.9727
Press any key to continue . . .
```

Algoritmen og resultatene minner mye om ANN, men i forhold til ytelse bruker den mye mer tid til å trene opp det nevrale nettverket. Det er ikke så mye mer å si om denne borsett fra at den har mye mer imponerende accuracy resultater.

Resultat etter kjøring av exam_keras_cnn.py



Denne løsningen var meget nøyaktig. Etter 15 iterasjoner med trening og testing ender vi opp med en accuracy på 98.8%. Det å få trent den opp tar relativt lang tid med tanke på å kjøre gjennom 60.000 rader 15 ganger. Etter at det nevrale nettverket har kjørt gjennom testene og har lært, så gir den 15 generasjonen det beste resultatet av alle.

Selv om vi er svært fornøyd over sluttresultatet, var tiden vi brukte for å komme til resultatet svært tidsfordrivende. I forhold til KNN algoritmen som kjørte umiddelbart, måtte vi vente 30 minutter for å få CNN resultatet.

Algoritmen skaper gode resultater, men svekker ved effektiviteten på grunn av tids fordrivelse.

Forskjell med algoritmene:

ANN og CNN har mye tilfelles, men som nevnt før så er de åpenbare forskjellene mellom disse er ytlese og resultat. ANN genererer kjappere resultater til færre accuracy, mens CNN genrerer utrolig tregere resultater til økt accuracy.

Begge disse algoritmene virker perfekt for andre bildegjengjennings oppsetter, og vi føler at algoritmene kan gi like gode resultater til noe mer enn 10 tall bilder og se igjennom.

KNN i forhold til disse algoritmene er utrolig annerledes, der KNN er en lat algoritme. Som nevnt før så tviler vi på at KNN ville generert like gode resultater på et mer komplekst opplegg.

Konklusjon:
Når det kommer til vårt eget arbeid så føler vi selv at vi har jobbet på en ryddig og logisk måte. Vi begynte det hele med research, fant eksempler og jobbet deretter ut i fra disse. Vi hadde stort fokus på å ikke bare kopiere algoritmene, men å faktisk forstå de og finne gode forklaringer på hvorfor vi fikk de resultatene vi fikk.

Noe vi ikke er helt fornøyd med er antallet algoritmer og rammeverk vi har brukt. Vi hadde et stor ønske om å bruke TFLearn til en av løsningene, men det fikk vi dessverre ikke til. Vi føler også at når en oppgave er like åpen som dette så hadde det vært mer positivt med flere algoritmer enn de tre vi brukte.

De algoritmene vi vurderte var boosted stumps, SVM og linear classifier. Igjen så fikk vi dessverre ikke til flere enn de vi har levert.

Resultatene vi har oppnådd er både imponerende og ikke fullt så imponerende. Dette mener vi er viktig å vise fram ettersom at alt ikke er så perfekt som man alltid vil ha det til.

Igjennom hele denne oppgaven har vi lært oss en del. Vi har forstått forskjellen på forskjellige algoritmer og føler oss nå relativt trygge på de algoritmene vi har brukt. Selv om vi baserte mye av løsningen vår på andres eksempler og kunnskap så har vi fått til å endre mye av koden til vårt behov.

Kilder

tensorflow_ann:
https://www.digitalocean.com/community/tutorials/how-to-build-a-neural-network-to-recognize-handwritten-digits-with-tensorflow

sklern_knn:
https://www.kaggle.com/marwaf/handwritten-digits-classification-using-knn

sklearn_ann og sklearn_ann_old:
https://www.youtube.com/watch?v=aZsZrkIgan0

tensorflow_cnn:
https://towardsdatascience.com/image-classification-using-convolutional-neural-networks-on-mnist-data-set-406db0c265ed

keras_cnn:
https://github.com/kjaisingh/mnist-digits