# MAL O3 Journal

ITMAL-01 Forår 2022

Journal over opgaver fra L07-L09


**Afleveret af: Gruppe 21**

| Retning | Navn | | Studie ID | AU-nummer |
|---|---|---|---|---|
| E | Lukas Kezic | LKE | 201906917 | AU637735 |
| E | Rasmus Holm Lund | RHL | 201900058 | AU630898 |
| E | Jakob Peter Aarestrup | JPA | 201907898 | AU632998 |
| E | Frederik Thomsen | FT | 201906146 | AU637451 |

**Kontaktperson:**

| Retning | Navn | Studie ID | Email |
|---|---|---|---|
| E | Rasmus Holm Lund | 201900058 | 201900058@post.au.dk |



Aarhus Universitet
19. april 2022

# Indholdsfortegnelse

# 1 L07 - CNN

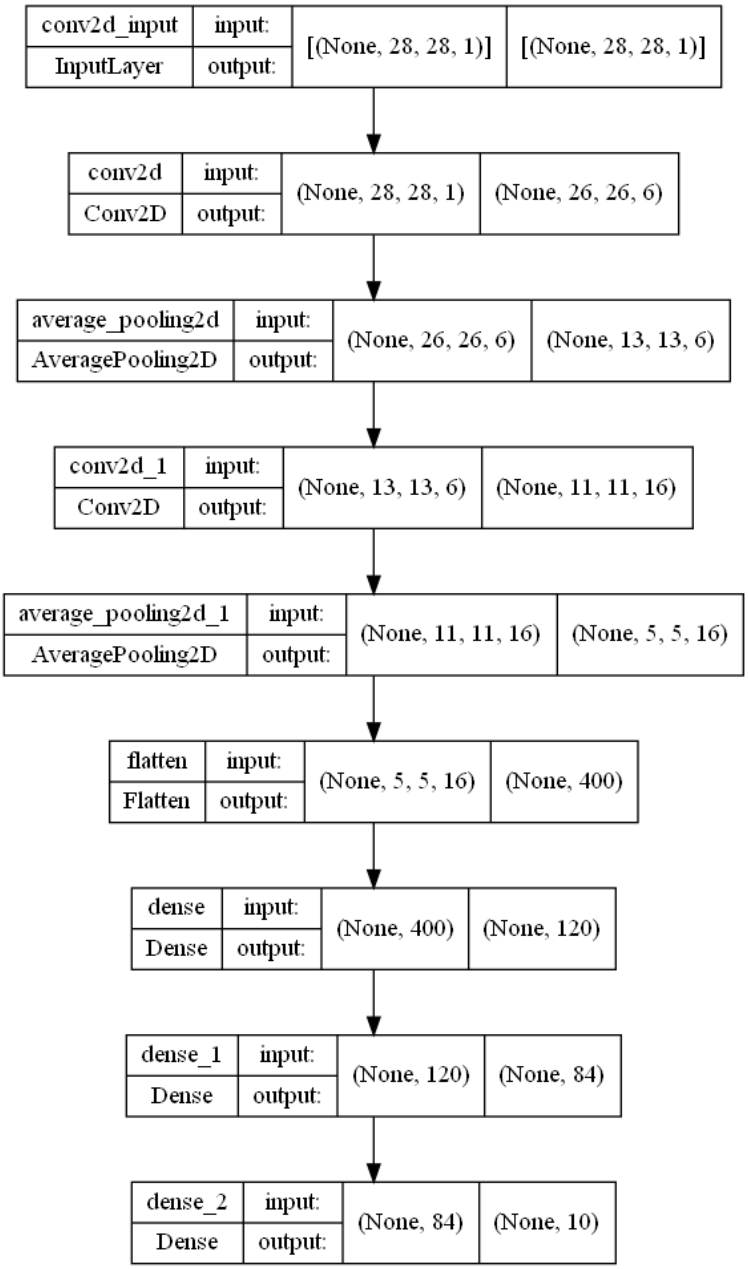
## 1.1 Convolutional Neural Networks

CNN står for Convolutional Neural Network. Når CNN bliver brugt handler det primært om de her convolution layers som bliver brugt til at filterer og processerer f.eks. et billede. For hvert convolution layer bliver dataen filtereret yderligere indtil det ønskede resultat opnås. Dvs. det kunne være et program som skal genkende koala ansigter. Hvor nødvendigt at have flere convolution layers for at opnå bedre nøjagtighed med at genkende f.eks. øjne, øre og mund, inden man når det Neurale Network.

Det var meget vigtigt at lave så layer størrelsen og modellen var samme størrelse. Idet at koden ikke ville køre uden at de var samme størrelse. Derudover var det vigtigt at matche de forskellige dimensioner af testdata for at opnå vores ønskede resultat. Desto større dimensioner der blev regnet på desto mindre nøjagtigt var programmet.

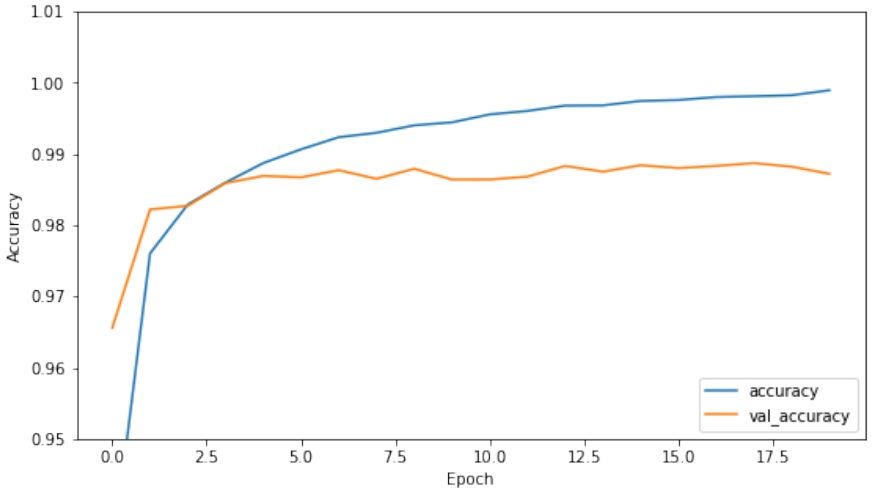
Der er gjort brug af keras funktionen mnist.load$_data()$, $hvorvisplitterdatasettethhv.itestdataogtraindata.Dissebliverbrugtsa$

CNN setuppen er visualiseret på Figur 1, hvor de forskellige lag, der er lagt i modellen er opdelt i kasser. Her kan der eksempelvis i første kasse ses conv2d_input-laget med en input størrelse på (28,28,1) og et output på (28,28,1). Dvs størrelsen på de billeder der er indsat i modellen og derefter hvad den konverteres til. I dette tilfælde var der ingen ændring, men kigger man i næste kasse kan der nu ses en ændring i outputtet til (26,26,6). Vores data skal altså igennem alle lagene før vi kan bruge den endelige CNN-model.



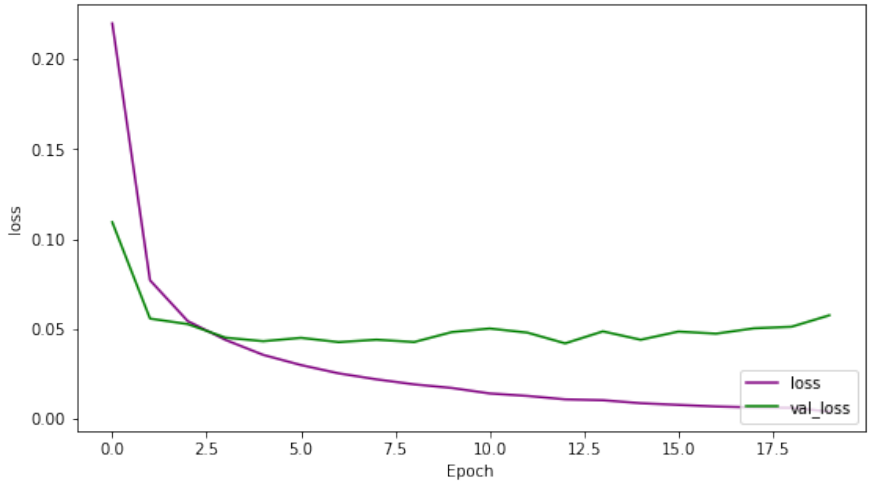

**Figur 1:** Diagram af CNN modellen

Som en start blev vores model fittet med 5 epochs. Vi kunne dog se at præcisionen steg desto flere epochs vi brugte og derfor endte vi ud med at køre fitten med 20 epochs. Dette kan også ses beskrevet i Figur 2, hvor plottet tydeligt viser en stigning i accuracy pr. epoch. Dog kan man også se at val_accuracy er relativt stationær ift. accuracy.




**Figur 2:** CNN accurary plot

Kigger vi på Figur 3 kan vi ligeledes se at vores loss falder drastisk efter den første epoch og fortsætter med at være langsomt nedadgående derefter. Val_loss agerer til gengæld hovedsageligt ligesom valog svinger let, men er som udgangspunkt stationær.




**Figur 3:** CNN loss plot

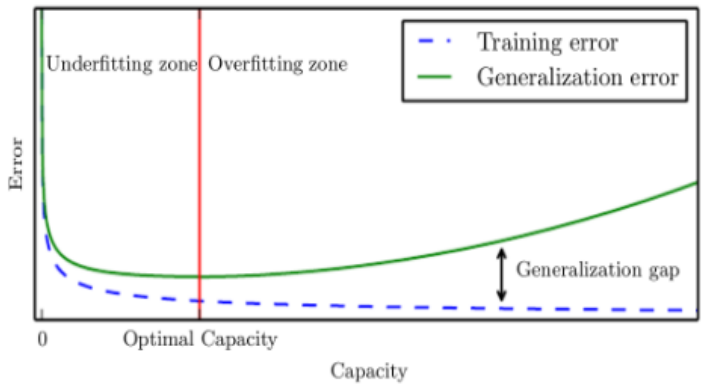
## 1.2 Konklusion

Der er i opgaven gjort brug af datasættet MNIST til at bygge et convolutional neural network (CNN), hvor der er forsøgt at få den højeste nøjagtighed. Dette er lykkedes ved at opbygge CNN'et i forskellige lag, der gennemgår dataen. Derudaf er der kommet en score på 98,77% nøjagtighed og et loss på 0,056. Der er udarbejdet plots, der tydeligt viser udviklingen pr. epoch, men også hvordan at forbedringen stagnerer efter relativt hurtigt.

# 2 L08 - Generalization error

## 2.1 Qa - Beskrivelse af Generalization error

Figur 4 viser forholder mellem et datasæts kapacitet og dets forventede fejl. På figuren er forholdet mellem træningsfejlen og generaliseringsfejlen blevet plottet. Træningsfejlen er den fejl vi har beregnet ud fra vores træningssæt, og generaliseringsfejlen er den fejl vi får, hvis vi bruger vores model på noget ukendt data. Det kan ses, at man ved lav data-kapacitet får meget store træningsfejl og generaliseringsfejl, fordi kapaciteten er så lille. Dette fænomen kaldes for underfitting. Når data-kapaciteten derimod bliver for stor, begynder generaliseringsfejlen at blive større, mens træningsfejlen bliver mindre, som vil medføre et stor generaliseringrum. Dette fænomen kaldes for overfitting. Man vil typisk vælge en optimal kapacitet, der ligger lige mellem vores underfitting-zone og vores overfitting-zone, da dette giver den bedste repræsentation af vores data.




**Figur 4:** Forhold mellem kapacitet og forventet fejl

## 2.2 Qb - Et MSE-Epoch/Error Plot

Nedenfor er de forskellige dele af den kode, der plotter forholdet mellem RMSE og kapacitet blevet forklaret. Derudover er koden også vedlagt.

Part I: Først genereres noget tilfældigt data. Herefter opdeles denne data i test- og træningsdata, hvorefter det skaleres, og til sidst bliver det plottet.

```
# Run code: Qb(part I)
# NOTE: modified code from [GITHOML], 04_training_linear_models.ipynb

%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
import numpy as np

from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import SGDRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

np.random.seed(42)

def GenerateData():
    m = 100
    X = 6 * np.random.rand(m, 1) - 3
    y = 2 + X + 0.5 * X**2 + np.random.randn(m, 1)
    return X, y

X, y = GenerateData()
X_train, X_val, y_train, y_val = \
    train_test_split( \
        X[:50], y[:50].ravel(), \
        test_size=0.5, \
        random_state=10)

print("X_train.shape=",X_train.shape)
print("X_val__.shape=",X_val.shape)
print("y_train.shape=",y_train.shape)
print("y_val__.shape=",y_val.shape)

poly_scaler = Pipeline([
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)
            ),
        ("std_scaler", StandardScaler()),
    ])
```

**Listing 1:** Part I

Part II: Herefter defineres det, at det er en SGDRegressor model, der skal bruges. Derefter bruges der fit på træningsdataen, og predict på både træning- og testdataen. Derudover beregnes der en Mean Squared Error på både trænings- og testdataen, hvor værdien gemmes i mse_train og mse_val, hvorefter de udprintes. Dette gøres i et loop med epochs, som også udprintes, som svarer til kapapciteten af dataen. Formålet med dette er at kunne se forholdet mellem kapaciteten af dataen og RMSE-fejlen for træningsdataen og for testdaten, som også kaldes for træningsfejlen og generaliseringsfejlen.

```
# Run code: Qb(part II)

def Train(X_train, y_train, X_val, y_val, n_epochs, verbose=False):
    print("Training...n_epochs=",n_epochs)

    train_errors, val_errors = [], []

    sgd_reg = SGDRegressor(max_iter=1,
                           penalty=None,
                           eta0=0.0005,
                           warm_start=True,
                           early_stopping=False,
                           learning_rate="constant",
                           tol=-float("inf"),
                           random_state=42)

    for epoch in range(n_epochs):

        sgd_reg.fit(X_train, y_train)

        y_train_predict = sgd_reg.predict(X_train)
        y_val_predict   = sgd_reg.predict(X_val)

        mse_train=mean_squared_error(y_train, y_train_predict)
        mse_val  =mean_squared_error(y_val  , y_val_predict)

        train_errors.append(mse_train)
        val_errors  .append(mse_val)
        if verbose:
            print(f"  epoch={epoch:4d}, mse_train={mse_train:4.2f}, mse_val
                ={mse_val:4.2f}")

    return train_errors, val_errors

n_epochs = 500
train_errors, val_errors = Train(X_train_poly_scaled, y_train,
    X_val_poly_scaled, y_val, n_epochs, True)

print('OK')
```

**Listing 2:** Part II

Part III: RMSE-fejlen ift. kapactiten for træningsdataen og testdaten bliver nu plottet vha. plt.plot funktionen. Derudover plottes den bedste RMSE-værdi, som er fundet ud fra den mindste Mean Squared Value værdi.

```
# Run code: Qb(part III)

best_epoch = np.argmin(val_errors)
best_val_rmse = np.sqrt(val_errors[best_epoch])

plt.figure(figsize=(10,5))
plt.annotate('Best model',
             xy=(best_epoch, best_val_rmse),
             xytext=(best_epoch, best_val_rmse + 1),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.05),
             fontsize=16,
            )

best_val_rmse -= 0.03  # just to make the graph look better
plt.plot([0, n_epochs], [best_val_rmse, best_val_rmse], "k:", linewidth=2)
plt.plot(np.sqrt(train_errors), "b-", linewidth=2, label="Training set")
plt.plot(np.sqrt(val_errors), "g-", linewidth=3, label="Validation set")
plt.legend(loc="upper right", fontsize=14)
plt.xlabel("Epoch", fontsize=14)
plt.ylabel("RMSE", fontsize=14)
plt.show()

print('OK')
```

**Listing 3:** Part III

## 2.3 Qc - Tidlig stop

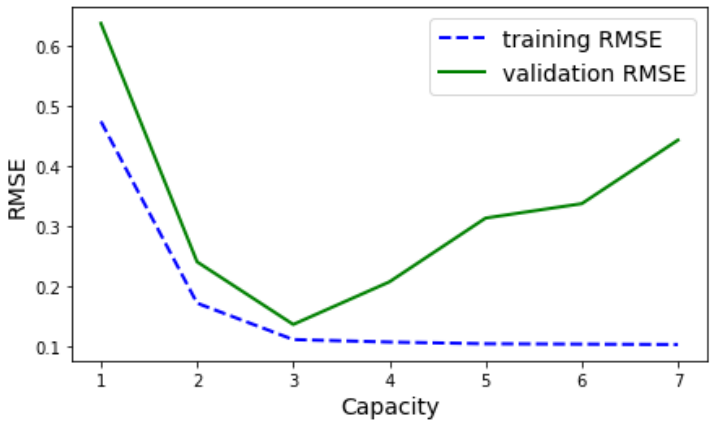
Early stopping er en reguleringsmetode man bruger for at undgå at ens data bliver for underfit eller overfit. Man kan implemtere early stopping i koden ovenfor ved at indsætte en if-sætning, der tjekker om valideringsfejlen val_error er mindre end datasættets minimale fejl minimum_error_value. Når dette sker, er den bedste model nået, da valideringsfejlen har nået sit minimum. Pseudokoden er vist nedenfor. Early stopping metoden er implementeret mellem linje 27 og 32.

```
for epoch in range(number of epochs)
{
    train_errors, val_errors = [], []

    sgd_reg = SGDRegressor(max_iter=1,
                           penalty=None,
                           eta0=0.0005,
                           warm_start=True,
                           early_stopping=False,
                           learning_rate="constant",
                           tol=-float("inf"),
                           random_state=42)

  minimum_error_value = 1;

  fit(x_train,y_train)

  y_train_predict = sgd_reg.predict(X_train)
    y_val_predict   = sgd_reg.predict(X_val)

  mse_train=mean_squared_error(y_train, y_train_predict)
    mse_val  =mean_squared_error(y_val  , y_val_predict)

    train_errors.append(mse_train)
    val_errors  .append(mse_val)

  if(val_error < minimum_error)
  {
    minimum_error_value = val_error
    best_epoch = epoch
    best_model = clone(sgd_reg)
  }

}
```

**Listing 4:** Pseduokode

## 2.4 Qd - Forklaring af Polynomial RMSE-Capacity plot

Nedenfor på Figur 5 ses forholdet mellem et datasæts kapacitet og RMSE (Root Mean Sqaured Error) for et datasæt, hvor polynomisk regression er blevet anvendt. Det kan ses, at valideringsfejlen begynder at stige ved en kapacitet på 3, mens træningsfejlen bliver ved med at falde. Valideringsfejlen stiger ved en kapacitet på 3, som betyder at dataen er blevet overfittet. Overfitting betyder at den regressionsmodel der bruges passer så godt til datasættet, at den ikke vil kunne forudse fremtidige resultater. En større kapacitet på 3 vil altså medføre en større valideringsfejl, men en mindre træningsfejl.




**Figur 5:** Forhold mellem kapacitet og RMSE med polynomisk regression

## 2.5 Konklusion

Det kan konkluderes at det har været muligt at eftervise og beskrive koncepterne omkring træning i Machine Learning. Heriblandt generaliseringsfejl, træningsfejl, underfitting, overfitting, kapacitet, optimal kapacitet og RMSE. Derudover kan det konkluderes, at det er vigtigt, at ens datasæt har den rigtige kapacitet, så man ikke ender med en model, der er i en underfit eller overfit zone.

# 3 L09 - Gridsearch

## 3.1 Qa - Forklaring af GridSearchCV

I opgavesættet er der vist 2 kode celler, som indeholder forskelligt kode. Vi starter med at kigge på celle 2, eftersom det er den vigtigste at forstå. I denne celle har vi først en linje af kode, der vælger hvilket dataset der arbejdes med. Her er der 3 muligheder "iris", "moon"eller "mnist". I denne opgave vil der kun blive fokuseret på iris og mnist, eftersom de er mest relevant. Den næste linje af kode er hvor vi vælger vores model, denne linje kan ændres alt efter hvilken model man gerne vil bruge. Den næste del af koden er vores tuning parameter, her bestemmer vi de parameter, som vi gerne vil ændre. Man skal altså indsætte de hyper parameter, som man ønsker at teste på ens model. Den sidste og vigtigste funktion i denne kode er vores gridsearch. Denne funktion tester automatisk de tuning parameter vi valgte tidligere. Denne funktion har dog selv nogle parameter, som man skal tage stilling til. Man fitter så ens gridsearch og kører den igennem en funktion. Denne funktion er defineret i celle 1. Celle 1 består hovedsageligt af en masse definitioner, som blandt andet loader det valgte dataset. Vi vil ikke gå i dybden med hvordan disse definationer er opbygget. Alt vi vil nævne er at de bliver brugt i celle 2, til at printe et pænt resultat opsætning af en gridsearch.

## 3.2 Qb - Hyperparameter Grid Search vha. SDG classifier

Nu hvor vi kender til måden man opsætter gridsearch, kan vi prøve at lave vores egen. Her prøver vi at lave et gridsearch på en SGDClassifier model. Vi starter altså med at ændre vores model om til en SGDClassifier, under vores setup parameter. Bagefter tilføjer vi en masse hyper parameter til vores tuning parameter, så den ved hvad den skal teste blandt.

```
1 # Setup search parameters
2 model = SGDClassifier()
3
4 tuning_parameters = {
5     'alpha': [ 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5],
6     'max_iter': [0.1, 1, 10, 100, 1000, 10000, 100000],
7     'learning_rate':('constant','optimal','invscaling','adaptive'),
8     'eta0' : [0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5]
9     'power_t' : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
10 }
```

Her har vi tilføjet 5 hyper parameter, som hver især har forskellige antal af parameter, som gridsearch kan vælge imellem. Disse parameter er manuelt indskrevet og fundte igennem scikitlearns wiki. Alt der mangler er at køre koden, for at finde de optimale hyper parameter. Efter koden har kørt i over 2 minutter, får vi et resultat på:

```
1 CTOR for best model: SGDClassifier(eta0=0.1, learning_rate='adaptive',
      max_iter=100, power_t=0.6)
2
3 best: dat=iris , score=0.99048, model=SGDClassifier(alpha=0.0001,eta0=0.1,
      learning_rate='adaptive',max_iter=100,power_t=0.6)
4
5 OK(grid-search)
```

Dermed har vi fundet de bedste hyperparameter med en score på 0.99, ved hjælp af gridsearch.

### 3.3 Qc - Hyperparameter Random Search vha. SDG classifier

Nu skal vi gentage koden fra tidligere men istedet for et gridsearch, skal der bruges randomsearch. Randomsearch, har det samme formål som gridsearch, om at finde de bedste hyperparameter. De fungere dog på to vidt forskellige måder. Gridsearch kører alle tilgængelige kombinationer af hyperparameter igennem trin for trin. Mens randomsearch angiver man antal af forskellige kombinationer den skal prøve, hvor den vælger disse kombinationer tilfældigt. Så lad os starte med at ændre gridsearch om til randomsearch i koden.

```
# Run GridSearchCV for the model
start = time()
random_tuned = RandomizedSearchCV(
    model,
    tuning_parameters,
    n_iter=20,
    random_state=42,
    cv=CV,
    scoring='f1_micro',
    verbose=VERBOSE,
    n_jobs=-1
)
```

Den sættes op ligesom gridsearch, den har dog nogle andre parameter, som f.eks. n_iter, som bestemmer antal af kombinationer den skal køre igennem før den stopper. Efter den har kørt de 20 kombinationer igennem, vælger den dén kombination som scorede højest. Så hvis vi kører koden får vi dette resultat:

```
CTOR for best model: SGDClassifier(alpha=0.01, eta0=0.5, learning_rate='
    invscaling', max_iter=10, power_t=0.8)

best: dat=iris , score=0.86667, model=SGDClassifier(alpha=0.01,eta0=0.5,
    learning_rate='invscaling',max_iter=10,power_t=0.8)

OK(random-search)
```

Det skal dog nævnes at eftersom randomsearch, er baseret på tilfældighed, betyder det at den ikke kommer med samme resultat hvergang. Man kan kører denne funktion, 10 gange og have 10 forskellige resultater. Det betyder altså at randomsearch ikke er den bedste metode, men den hurtigste. Så hvis man bare gerne vil tjekke nogle forskellige kombinationer af hyper parameter, er randomsearch et god redskab. Det tog 0.33 sekunder at gennemføre koden med randomsearch, hvor gridsearch brugte 131 sekunder. Gridsearch er altså betydelig hurtigere, men dette skyldes at randomsearch kun kører n_inter kombinationer. Vores randomsearch fik en kombination med en score på 0.87, hvor vores gridsearch fik 0.99. Gridsearch er den mest pålidelige, mens randomsearch er den hurtigste, plus der vil altid være en chance for at randomsearch får den samme score som gridsearch.

## 3.4 Qd - MNIST Search Quest II

I den sidste del skal de tidligere funktioner testes på Mnist dataen. Så her starter vi ud med at ændre vores data om til mnist.

```
1 # Setup data
2 X_train, X_test, y_train, y_test = LoadAndSetupData('mnist')
```

Nu hvor vi har ændret hvilket dataset, vi bruger, kan vi kører koden igen. Her skal vi dog være opmærksom på størrelsen af vores dataset, eftersom mnist dataen er betydelig større end iris. Dette betyder at vores kode, vil tage meget længere tid at køre end tidligere. Derfor er det vigtigt at man ikke tester for mange hyper parameter. Vi kører altså koden ligesom før og venter på at den er færdig.

Med de hyperparameter vi har brugt fra tidligere, tager det koden ca. 35161,26 sekunder at køre. Det betyder altså, at efter 9,7 timer har koden kørt igennem alle mulige kombinationer. Det giver os en score på 0.89912, denne score kan dog forbedres, men det kræver flere hyperparameter og derfor mere tid.

## 3.5 Konklusion

Der blev dog testet med forskellige hyperparameter, ovenstående var dog den bedste, eftersom de andre enden gav en lavere score, eller tog alt for lang tid til at køre. Der blev også testet med alle hyper parameter, dette kunne dog ikke lade sig gøre eftersom, det enden tog alt for lang tid (mere end 48 timer), eller fordi pc'en ikke kunne håndtere det. Her kunne man dog have brugt noget preprocessing, til at fortynde mængden af arbejdet. Der er altså blevet lært hvordan man kan teste forskellige hyperparameter automatisk.

# 4 Appendix

## 4.1 L07

```
import keras
from keras import layers
from keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(x_train,y_train),(x_test,y_test) = keras.datasets.mnist.load_data()

y_train_labels = to_categorical(y_train, dtype="uint8")
y_test_labels = to_categorical(y_test, dtype ="uint8")

print(y_train_labels[0])
print(y_test_labels[0])

print(x_train.shape)
print(x_test.shape)
print(y_train_labels.shape)
print(y_test_labels.shape)
```

**Listing 5:** L07 - 1

```
model = keras.Sequential()

model.add(layers.Conv2D(filters=6, kernel_size=(3,3), activation='relu',
    input_shape=(28,28,1)))
model.add(layers.AveragePooling2D())
model.add(layers.Conv2D(filters=16, kernel_size=(3,3), activation='relu',))
model.add(layers.AveragePooling2D())
model.add(layers.Flatten())
model.add(layers.Dense(units=120, activation='relu'))
model.add(layers.Dense(units=84, activation='relu'))
model.add(layers.Dense(units=10, activation='softmax'))

model.build()

model.summary()
```

**Listing 6:** L07 - 2

```
from keras.utils.vis_utils import plot_model
model.compile(optimizer="sgd", metrics=['accuracy'],loss = keras.losses.
    CategoricalCrossentropy
                (
from_logits=False,
    label_smoothing =0.0,
    axis = 1,
    reduction ="auto",
    name ="categorical_crossentropy"
))

cnn = model.fit(x_train, y_train_labels, epochs = 20, validation_data=(
    x_test,y_test_labels))
model.evaluate(x_test, y_test_labels)
plot_model(model, to_file='model_plot.png', show_shapes=True,
    show_layer_names=True)
```

**Listing 7:** L07 - 3

```
import matplotlib.pyplot as plt

plt.figure(figsize=(9,5))
plt.plot(cnn.history['accuracy'], label='accuracy')
plt.plot(cnn.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.95, 1.01])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(x_test, y_test_labels, verbose=2)
plt.show()

plt.figure(figsize=(9,5))
plt.plot(cnn.history['loss'], label='loss',color="purple")
plt.plot(cnn.history['val_loss'], label = 'val_loss',color="green")
plt.xlabel('Epoch')
plt.ylabel('loss')
#plt.ylim([0.95, 1.01])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(x_test, y_test_labels, verbose=2)
plt.show()
```

**Listing 8:** L07 - 4

## 4.2 L08

```
for epoch in range(number of epochs)
{
    train_errors, val_errors = [], []

    sgd_reg = SGDRegressor(max_iter=1,
                           penalty=None,
                           eta0=0.0005,
                           warm_start=True,
                           early_stopping=False,
                           learning_rate="constant",
                           tol=-float("inf"),
                           random_state=42)

  minimum_error_value = 1;

  fit(x_train,y_train)

  y_train_predict = sgd_reg.predict(X_train)
    y_val_predict   = sgd_reg.predict(X_val)

  mse_train=mean_squared_error(y_train, y_train_predict)
    mse_val  =mean_squared_error(y_val  , y_val_predict)

    train_errors.append(mse_train)
    val_errors  .append(mse_val)

  if(val_error < minimum_error)
  {
    minimum_error_value = val_error
    best_epoch = epoch
    best_model = clone(sgd_reg)
  }

}
```

**Listing 9:** L08 - 3

### 4.3 L09

```
1  from time import time
2  import numpy as np
3
4  from sklearn import svm
5  from sklearn.linear_model import SGDClassifier
6
7  from sklearn.model_selection import GridSearchCV, RandomizedSearchCV,
       train_test_split
8  from sklearn.metrics import classification_report, f1_score
9  from sklearn import datasets
10
11 from libitmal import dataloaders as itmaldataloaders # Needed for load of
       iris, moon and mnist
12
13 currmode="N/A" # GLOBAL var!
14
15 def SearchReport(model):
16
17     def GetBestModelCTOR(model, best_params):
18         def GetParams(best_params):
19             ret_str=""
20             for key in sorted(best_params):
21                 value = best_params[key]
22                 temp_str = "'" if str(type(value))=="<class 'str'>" else ""
23                 if len(ret_str)>0:
24                     ret_str += ','
25                 ret_str += f'{key}={temp_str}{value}{temp_str}'
26             return ret_str
27         try:
28             param_str = GetParams(best_params)
29             return type(model).__name__ + '(' + param_str + ')'
30         except:
31             return "N/A(1)"
32
33     print("\nBest model set found on train set:")
34     print()
35     print(f"\tbest parameters={model.best_params_}")
36     print(f"\tbest '{model.scoring}' score={model.best_score_}")
37     print(f"\tbest index={model.best_index_}")
38     print()
39     print(f"Best estimator CTOR:")
40     print(f"\t{model.best_estimator_}")
41     print()
42     try:
43         print(f"Grid scores ('{model.scoring}') on development set:")
44         means = model.cv_results_['mean_test_score']
45         stds  = model.cv_results_['std_test_score']
46         i=0
47         for mean, std, params in zip(means, stds, model.cv_results_['params
               ']):
48             print("\t[%2d]: %0.3f (+/-%0.03f) for %r" % (i, mean, std * 2,
                   params))
49             i += 1
50     except:
51         print("WARNING: the random search do not provide means/stds")
```



**Listing 10:** L09 - 1 - part 1

```
1
2       global currmode
3       assert "f1_micro"==str(model.scoring), f"come␣on,␣we␣need␣to␣fix␣the␣
            scoring␣to␣be␣able␣to␣compare␣model-fits!␣Your␣scoreing={str(model.
            scoring)}...remember␣to␣add␣scoring='f1_micro'␣to␣the␣search"
4       return f"best:␣dat={currmode},␣score={model.best_score_:0.5f},␣model={
            GetBestModelCTOR(model.estimator,model.best_params_)}", model.
            best_estimator_
5
6   def ClassificationReport(model, X_test, y_test, target_names=None):
7       assert X_test.shape[0]==y_test.shape[0]
8       print("\nDetailed␣classification␣report:")
9       print("\tThe␣model␣is␣trained␣on␣the␣full␣development␣set.")
10      print("\tThe␣scores␣are␣computed␣on␣the␣full␣evaluation␣set.")
11      print()
12      y_true, y_pred = y_test, model.predict(X_test)
13      print(classification_report(y_true, y_pred, target_names))
14      print()
15
16  def FullReport(model, X_test, y_test, t):
17      print(f"SEARCH␣TIME:␣{t:0.2f}␣sec")
18      beststr, bestmodel = SearchReport(model)
19      ClassificationReport(model, X_test, y_test)
20      print(f"CTOR␣for␣best␣model:␣{bestmodel}\n")
21      print(f"{beststr}\n")
22      return beststr, bestmodel
23
24  def LoadAndSetupData(mode, test_size=0.3):
25      assert test_size>=0.0 and test_size<=1.0
26
27      def ShapeToString(Z):
28          n = Z.ndim
29          s = "("
30          for i in range(n):
31              s += f"{Z.shape[i]:5d}"
32              if i+1!=n:
33                  s += ";"
34          return s+")"
```

**Listing 11:** L09 - 1 - part 2

```
1
2       global currmode
3       currmode=mode
4       print(f"DATA: {currmode}..")
5
6       if mode=='moon':
7           X, y = itmaldataloaders.MOON_GetDataSet(n_samples=5000, noise=0.2)
8           itmaldataloaders.MOON_Plot(X, y)
9       elif mode=='mnist':
10          X, y = itmaldataloaders.MNIST_GetDataSet(load_mode=0)
11          if X.ndim==3:
12              X=np.reshape(X, (X.shape[0], -1))
13      elif mode=='iris':
14          X, y = itmaldataloaders.IRIS_GetDataSet()
15      else:
16          raise ValueError(f"could not load data for that particular mode='{
                mode}', only 'moon'/'mnist'/'iris' supported")
17
18      print(f'  org. data:  X.shape      ={ShapeToString(X)}, y.shape      ={
            ShapeToString(y)}')
19
20      assert X.ndim==2
21      assert X.shape[0]==y.shape[0]
22      assert y.ndim==1 or (y.ndim==2 and y.shape[1]==0)
23
24      X_train, X_test, y_train, y_test = train_test_split(
25          X, y, test_size=test_size, random_state=0, shuffle=True
26      )
27
28      print(f'  train data: X_train.shape={ShapeToString(X_train)}, y_train.
            shape={ShapeToString(y_train)}')
29      print(f'  test data:  X_test.shape ={ShapeToString(X_test)}, y_test.
            shape ={ShapeToString(y_test)}')
30      print()
31
32      return X_train, X_test, y_train, y_test
33
34  print('OK(function setup, hope MNIST loads works, seem best if you got
        Keras or Tensorflow installed!)')
```

**Listing 12:** L09 - 1 - part 3

```
# Setup data
X_train, X_test, y_train, y_test = LoadAndSetupData('iris')  # 'iris', '
    moon', or 'mnist'

# Setup search parameters
model = svm.SVC(gamma=0.001)
# NOTE: gamma="scale" does not work in older Scikit-learn frameworks,
# FIX:  replace with model = svm.SVC(gamma=0.001)

tuning_parameters = {
    'kernel': ('linear', 'rbf'),
    'C': [0.1, 1, 10]
}

CV = 5
VERBOSE = 0

# Run GridSearchCV for the model
start = time()
grid_tuned = GridSearchCV(model,
                          tuning_parameters,
                          cv=CV,
                          scoring='f1_micro',
                          verbose=VERBOSE,
                          n_jobs=-1)
grid_tuned.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(grid_tuned, X_test, y_test, t)
print('OK(grid-search)')
```

**Listing 13:** L09 - 2

```
# Setup data
X_train, X_test, y_train, y_test = LoadAndSetupData(
    'iris') # 'iris', 'moon', or 'mnist'

# Setup search parameters
model = SGDClassifier()

tuning_parameters = {
    'alpha': [ 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5],
    'max_iter': [0.1, 1, 10, 100, 1000, 10000, 100000],
    'learning_rate':('constant','optimal','invscaling','adaptive'),
    'eta0' : [0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5],
    'power_t' : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
}

CV = 5
VERBOSE = 0

# Run GridSearchCV for the model
start = time()
grid_tuned = GridSearchCV(model,
                          tuning_parameters,
                          cv=CV,
                          scoring='f1_micro',
                          verbose=VERBOSE,
                          n_jobs=-1)

grid_tuned.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(grid_tuned, X_test, y_test, t)
print('OK(grid-search)')
```

**Listing 14:** L09 - 3

```
# Setup search parameters
model = SGDClassifier()

tuning_parameters = {
    'alpha': [ 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5],
    'max_iter': [0.1, 1, 10, 100, 1000, 10000, 100000],
    'learning_rate':('constant','optimal','invscaling','adaptive'),
    'eta0' : [0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5],
    'power_t' : [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
}

CV = 5
VERBOSE = 0

# Run GridSearchCV for the model
start = time()
random_tuned = RandomizedSearchCV(
    model,
    tuning_parameters,
    n_iter=20,
    random_state=42,
    cv=CV,
    scoring='f1_micro',
    verbose=VERBOSE,
    n_jobs=-1
)

random_tuned.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(random_tuned, X_test, y_test, t)
print('OK(random-search)')
```

**Listing 15:** L09 - 4

```
# Setup data
X_train, X_test, y_train, y_test = LoadAndSetupData(
    'mnist')  # 'iris', 'moon', or 'mnist'

# Setup search parameters
model = SGDClassifier()

tuning_parameters = {
    'alpha': [ 0.0001, 0.001, 0.01, 0.1],
    'max_iter': [0.1, 1, 10, 100, 1000, 10000],
    'learning_rate':('constant','optimal','invscaling','adaptive'),
    'eta0' : [0.001, 0.01, 0.1],
    'power_t' : [0.1, 0.2, 0.3, 0.4, 0.5]
}

CV = 5
VERBOSE = 0

# Run GridSearchCV for the model
start = time()
grid_tuned = GridSearchCV(model,
                          tuning_parameters,
                          cv=CV,
                          scoring='f1_micro',
                          verbose=VERBOSE,
                          n_jobs=-1)

grid_tuned.fit(X_train, y_train)
t = time() - start

# Report result
b0, m0 = FullReport(grid_tuned, X_test, y_test, t)
print('OK(grid-search)')
```

**Listing 16:** L09 - 5