ITMAL - Journal 2

Jeppe Stærk - 201271201

Forår 2019

Litteratur

[1] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618, 9780262035613.

[2] A. Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition, 2017. ISBN 1491962291, 9781491962299.

Indholdsfortegnelse

L05 gradient_descent 1

Den grundlæggende idé bag Gradient Descent metoden, er at man iterativt forbedrer sine parametre for at optimere ud fra en kost-funktion.

1.1 Qa The Gradient Descent Method (GD)

I linje 22 fyldes θ med tilfældige værdier, dette kaldes *Random Initialization*. Det giver et udgangspunkt som man forsøger gradvist at forbedre, indtil der konvergeres på et minimum - eller i det her tilfælde bruger man præcist 1000 iterationer på at komme så tæt på den optimale løsning som muligt (linje 20). Man kunne forbedre implementationen ved at afbryde yderligere iterationer, når forskellen mellem en iteration og den forrige kommer under en defineret tolerance.

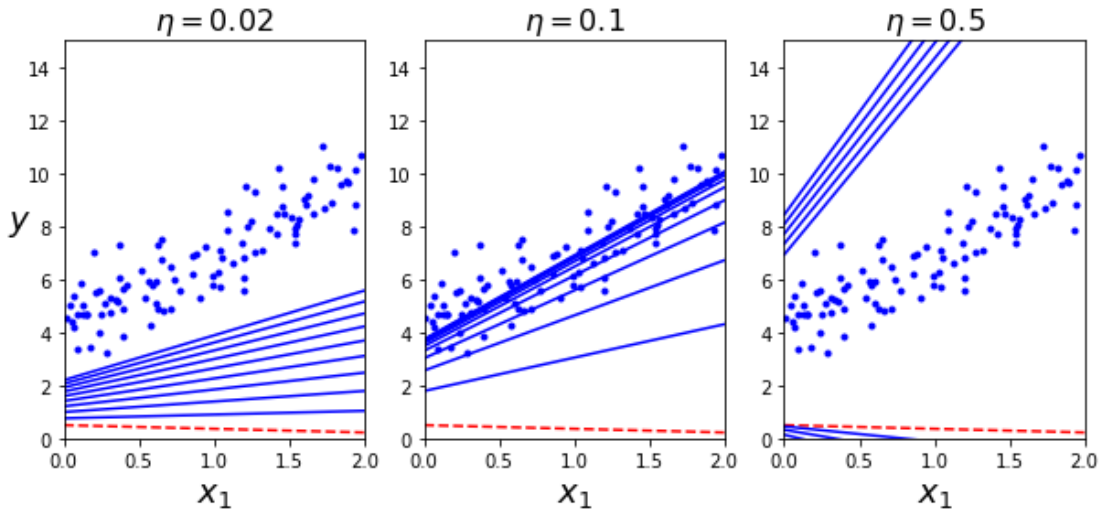
m værdien som sættes til 100 i linje 21 er antallet af data punkter, hvilket passer godt sammen med størrelserne af de vektorer der genereres i `GenerateData` funktionen.

η værdien som sættes til 0.1 i linje 19 er vores *step størrelse*, eller *learning rate*. Når gradienterne fra et givent punkt er beregnet, bruges step størrelse til at afgøre hvor stort et spring der tages i den retning hvor der er den stejleste hældning. Man kan også bruge en adaptiv step størrelse, hvor størrelsen varierer fra den ene iteration til den næste. I dette tilfælde er der dog valgt en statisk step størrelse på 0.1.

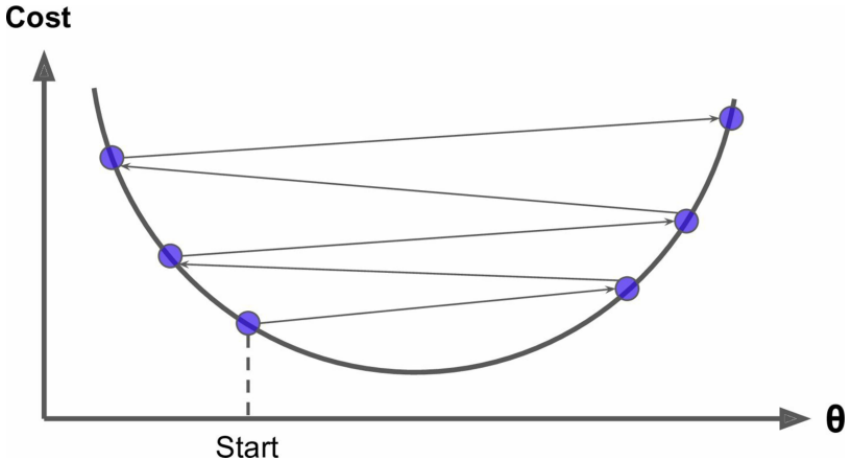
De 3 figurer der plottes viser hvordan Gradient Descent metoden konvergerer på et datasæt, med en η på hhv. 0.02, 0.1 og 0.5. Det er kun de første 10 iterationer der plottes, men det ses tydeligt at en step størrelse på 0.02 godt nok vil konvergere til den korrekte værdi, men det går meget langsomt i forhold til en step størrelse på 0.1. Har vi derimod en for høj step størrelse risikerer vi at vi aldrig konvergerer, men i stedet skyder forbi målet ved hver eneste iteration, som det ses med $\eta = 0.5$. Figur 1.1 illustrerer hvordan en for stor step størrelse kan resultere i at vi kommer længere og længere fra det ønskede resultat.

```python
def GenerateData():
    X = 2 * np.random.rand(100, 1)
    y = 4 + 3 * X + np.random.randn(100, 1)
    X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
    return X, X_b, y

X, X_b, y = GenerateData()

eta = 0.1
n_iterations = 1000
m = 100
theta = np.random.randn(2,1)

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

```
stochastic gradient descent theta=[4.20831857 2.79226572]
```




```
OK
```



Figur 1.1: Illustration af for høj learning rate ved Gradient Descent. Kilde: [2]

1.2 Qb The Stochastic Gradient Descent Method (SGD)

Det grundlæggende forskel ved Stochastic Gradient Descent metoden, er at man i stedet for at bruge hele sættet af træningsdata i hver iteration, kun bruger et tilfældigt udvalgt punkt fra sættet af træningsdata. Det betyder dels at den stokastiske metode er langt hurtigere pr. iteration, men også at den er bedre til at håndtere større mængder træningsdata hvor det kan være svært at rumme træningsdata i RAM.

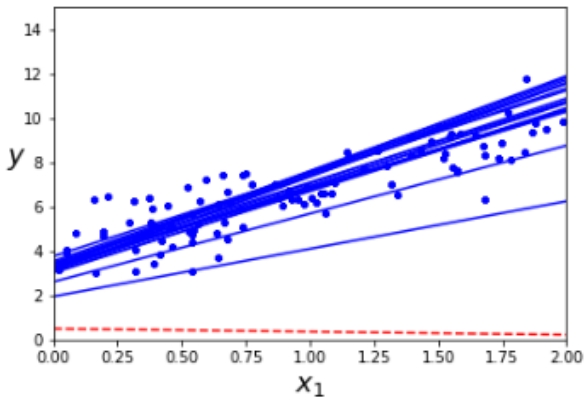
Hver iteration er til gengæld mindre præcis, og nogle steps vil endda være i den forkerte retning, så den stokastiske tilgang kræver dermed mange flere iterationer. Hver iteration er dog så meget hurtigere at udføre, at det ofte godt kan betale sig at lave et større antal hurtige approximerende iterationer, sammenlignet med et mindre antal langsomme men præcise iterationer.

I implementationen her er der lavet adaptiv step størrelse, hvilket ses i funktionen `learning_schedule(t)`. Funktionen kaldes med stigende `t` værdier, og step størrelsen vil derfor blive mindre for hver iteration.

`np.random.randint(m)` i linje 20 udvælger et tilfældigt data punkt, som bruges til at beregne hældninger og deraf afgøre retningen af næste step.

```
1  theta_path_sgd = []
2  m = len(X_b)
3  np.random.seed(42)
4
5  n_epochs = 50
6  t0, t1 = 5, 50  # learning schedule hyperparameters
7
8  def learning_schedule(t):
9      return t0 / (t + t1)
10
11 theta = np.random.randn(2,1)  # random initialization
12
13 for epoch in range(n_epochs):
14     for i in range(m):
15         if epoch == 0 and i < 20:
16             y_predict = X_new_b.dot(theta)
17             style = "b-" if i > 0 else "r--"
18             plt.plot(X_new, y_predict, style)
19
20         random_index = np.random.randint(m)
21         xi = X_b[random_index:random_index+1]
22         yi = y[random_index:random_index+1]
23
24         gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
25         eta = learning_schedule(epoch * m + i)
26         theta = theta - eta * gradients
27         theta_path_sgd.append(theta)
28
29         plt.plot(X, y, "b.")
30
31
32 from sklearn.linear_model import SGDRegressor
33 sgd_reg = SGDRegressor(max_iter=50, tol=-np.infty, penalty=None, eta0=0.1, random_state=42)
34 sgd_reg.fit(X, y.ravel())
```

```
stochastic gradient descent theta=[3.84208754 3.03831083]
Scikit-learn SGDRegressor "thetas": sgd_reg.intercept_=[3.83534891], sgd_reg.coef_=[2.96948592]
```



OK

1.3 Qd Mini-batch Gradient Descent Method

Mini-batch Gradient Descent metoden bruger til hver iteration en tilfældig valgt delmængde af trænings datasættet, lidt ligesom SGD (Stochastic Gradient Descent). Forskellen i forhold til SGD er at man i Mini-batch bruger et mindre sæt data, i stedet for kun et enkelt datapunkt, til at beregne hældninger. Dermed får man en mellemting mellem SGD og Batch Gradient Descent, hvor hver iteration er hurtigere end Batch, men langsommere end SGD - hvor der kræves flere iterationer end Batch, men færre end SGD for at konvergere på en god løsning.

I implementationen herunder ses det i linje 15 at man blander (shuffler) datasættet, for så at bruge et mini-batch på 20 datapunkter (antallet er defineret i linje 4) til beregning af hældninger.

Igen bruges der her en adaptiv step størrelse (η), defineret i `learning_schedule` funktionen.

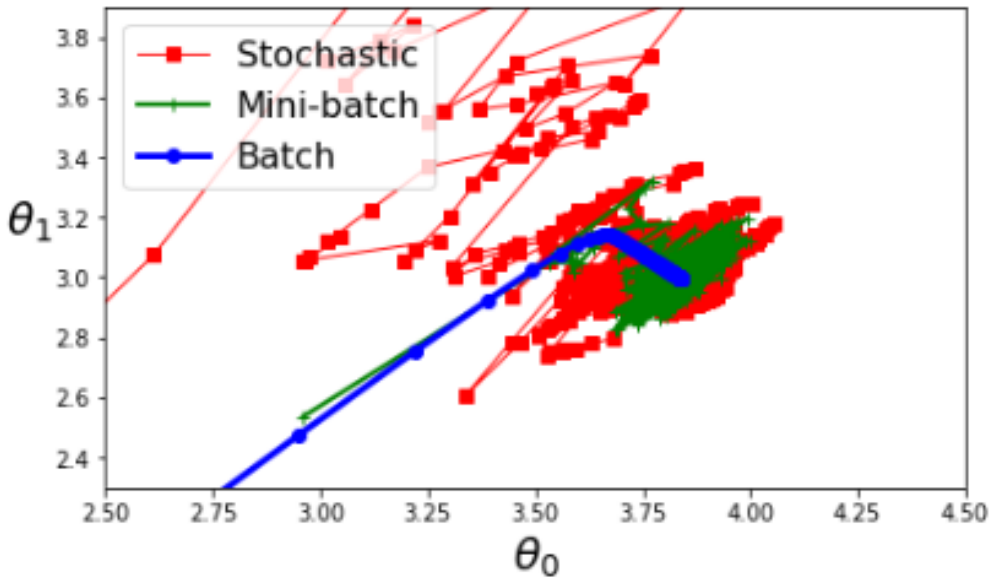
```python
theta_path_mgd = []

n_iterations = 50
minibatch_size = 20

np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

t0, t1 = 200, 1000
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(m)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, m, minibatch_size):
        t += 1
        xi = X_b_shuffled[i:i+minibatch_size]
        yi = y_shuffled[i:i+minibatch_size]

        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(t)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)

print(f'mini-batch theta={theta.ravel()}')
print('OK')
```

```
mini-batch theta=[3.86814831 3.02878845]
OK
```

1.4 Qe Choosing a Gradient Descent Method

I figuren herunder ses plot af hver iteration i træningen af hhv. Gradient Descent (også kaldt “Batch Gradient Descent", BGD), Stochastich Gradient Descent (SGD) og Mini-batch Gradient Descent (MGD). Det ses tydeligt at hvor BGD tager en meget direkte vej, så springer både SGD og MGD meget omkring, og de sidste mange iterationer springer både SGD og MGD rundt omkring i nærheden af det punkt hvor BGD konvergerer.

Der er fordele og ulemper ved hver metode, så hvilken man skal vælge afhænger af hvad man har af træningsdata. Når først man har trænet sin classifier og opnået et passende fit, er det stort set ligegyldigt hvilken metode man har brugt til at træne den. Har man derfor et begrænset datasæt til træning, og man skal aldrig eller kun sjældent træne sin classifier på ny, så er BGD den simple løsning. Har man et større datasæt og evt. behov for at gen-træne sin classifier, så er SGD eller MGD oplagte valg, hvor SGD er at foretrække hvis man prioriterer hastighed over præcision.


Stochastic
Mini-batch
Batch
θ_1
θ_0
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.50
2.75
3.00
3.25
3.50
3.75
4.00
4.25
4.50

L05 capacity_under_overfitting 2

2.1 Qa Explain the polynomial fitting via code review

`def true_fun(x)` definerer en cosinus kurve, som er den model man skal forsøge at modellere.

`def GenerateData(n_samples)` genererer et sæt træningsdata, som ligger tæt op ad `true_fun`, men med en smule støj tilføjet.

`for` loop i linje 24 looper igennem de 3 valgte grader af polynomie som skal afprøves: 1, 4 og 15 (linje 20).

I linje 28 laves der en matrice med de polynomial features svarende til graden af polynomien vi ser på i den givne iteration af for-loopen. I linje 30 laves der en *Ordinary least squares Linear Regression* predictor/estimator. I linjer 31-34 sættes matricen med polynomial features og Linear Regression predictor sammen i en *PipeLine*, som er en scikit learn klasse der kombinerer en række "transformers"med en predictor, for at kunne anvende dem på et datasæt sekventielt. Brugen af Pipeline gør det let at krydsvalidere på transformers og predictor som samlet enhed. I linje 35 trænes Linear Regression predictor på det genererede sæt test data.

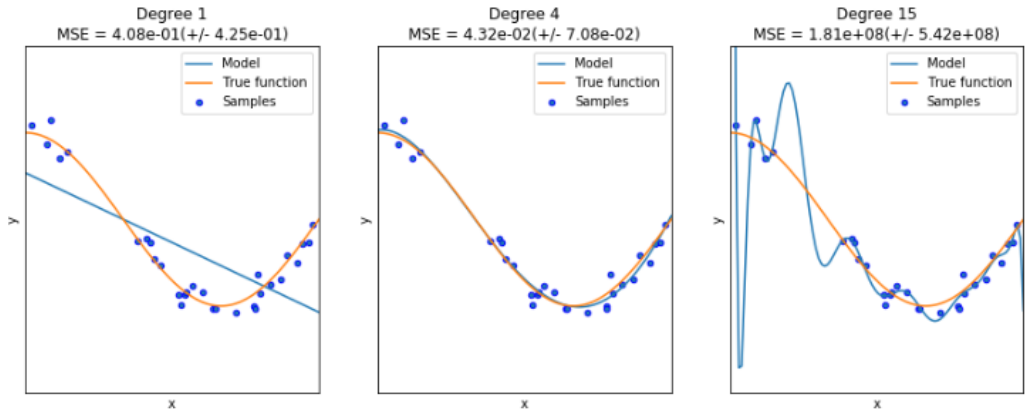
`cross_val_score` kaldes med pipeline og sættet af træningsdata som input, samt argumenter der specificerer at score skal beregnes som negativ mean squared error (*scoring* parameter), og at valideringen skal laves med en K-fold cross-validator, med $K = 20$ (*cv* parameter).

I linje 40 tages negativ `mean()` af resultatet fra krydsvalideringen, hvilket giver *Mean Squared Error* (MSE).

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

def true_fun(X):
    return np.cos(1.5 * np.pi * X)

def GenerateData(n_samples = 30):
    X = np.sort(np.random.rand(n_samples))
    y = true_fun(X) + np.random.randn(n_samples) * 0.1
    return X, y

np.random.seed(0)

X, y = GenerateData()
degrees = [1, 4, 15]

print("Iterating...degrees=",degrees)
plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i], include_bias=False)

    linear_regression = LinearRegression()
    pipeline = Pipeline([
            ("polynomial_features", polynomial_features),
            ("linear_regression", linear_regression)
        ])
    pipeline.fit(X[:, np.newaxis], y)
```

```

    # Evaluate the models using crossvalidation
    scores = cross_val_score(pipeline, X[:, np.newaxis], y, scoring="neg_mean_squared_error", cv=10)

    score_mean = -scores.mean()
    print(f"  degree={degrees[i]:4d}, score_mean={score_mean:4.2f},  {polynomial_features}")

    X_test = np.linspace(0, 1, 100)
    y_pred = pipeline.predict(X_test[:, np.newaxis])

    # Plotting details
    plt.plot(X_test, y_pred          , label="Model")
    plt.plot(X_test, true_fun(X_test), label="True function")
    plt.scatter(X, y, edgecolor='b', s=20, label="Samples")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title("Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(degrees[i], -scores.mean(), scores.std()))

plt.show()
print('OK')
```

```
Iterating...degrees= [1, 4, 15]
  degree=   1, score_mean=0.41,  PolynomialFeatures(degree=1, include_bias=False, interaction_only=False)
  degree=   4, score_mean=0.04,  PolynomialFeatures(degree=4, include_bias=False, interaction_only=False)
  degree=  15, score_mean=180526263.32,  PolynomialFeatures(degree=15, include_bias=False, interaction_only=False)
```



OK

2.2 Qb Explain the capacity and under/overfitting concept

2.2.1 Kapacitet

Modellens kapacitet svarer til antallet af polynomier som den kan repræsentere.

2.2.2 Under/Overfitting

Underfitting er når modellen ikke har kapacitet nok til at danne en god approximation af vores `true function`, som det ses i figuren fra forrige opgave *Qa* ved førstegradspolynomien, så er modellen en dårlig approximation da den ikke kan rumme kompleksiteten. Derimod har 15.gradspolynomium modellen så meget kapacitet at den kan rumme sættet af træningsdata nærmest perfekt, men den *overfitter* nu i sådan grad at den i nogle data områder vil være en meget dårlig predictor.

L05 generalization_error 3

3.1 Qa On Generalization Error

Forkaring af figur 3.1:

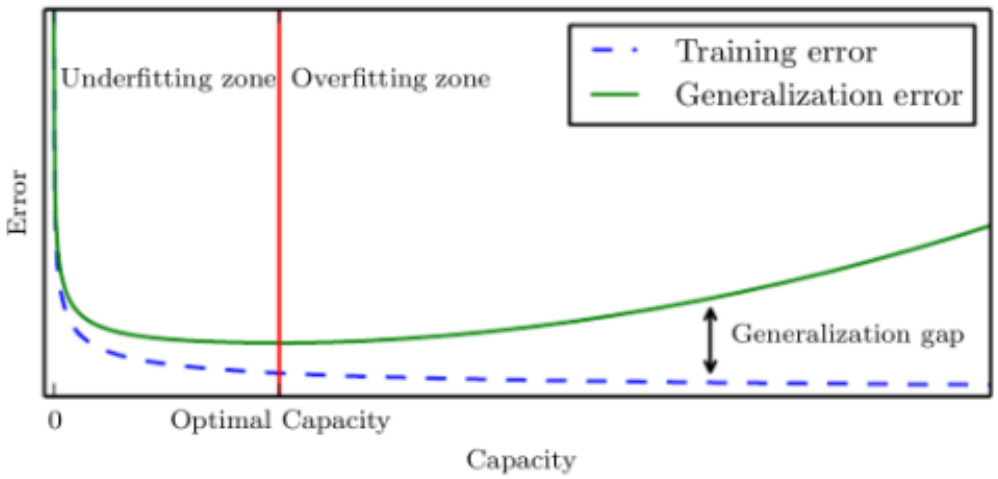
training/generalization error Den stiplede blå linje indikerer hvor god fit vi har på træningsdata, og den solide grønne linje indikerer hvor god fit vi har på virkeligheden vi prøver at modellere.

underfit/overfit zone I underfit zonen har vi ikke udnyttet træningsdata i ligeså stort omfang som vi kunne, for at få en model som er et godt fit på virkeligheden. Når vi kommer ind i overfit zonen, har vi en model som fitter vores træningsdata rigtig godt, men bliver en dårligere og dårligere approximation af virkeligheden.

optimal capacity Den optimale kapacitet er modellen er der hvor vi får maximal udnyttelse af træningsdata, uden at overfitte.

generalization gap Generaliserings-gap er spændet mellem den fejl vi har på fit til træningsdata, og den fejl vi har på den virkelige verden.

axes: x/capacity, y/error X-aksen symboliserer vores models kapacitet, og y-aksen hvor præcist vores model approximerer hhv. træningsdata og den virkelige verden. Jo større kapacitet vores model har, desto lavere fejl har den på træningsdata, men den bliver dårligere til at generalisere.

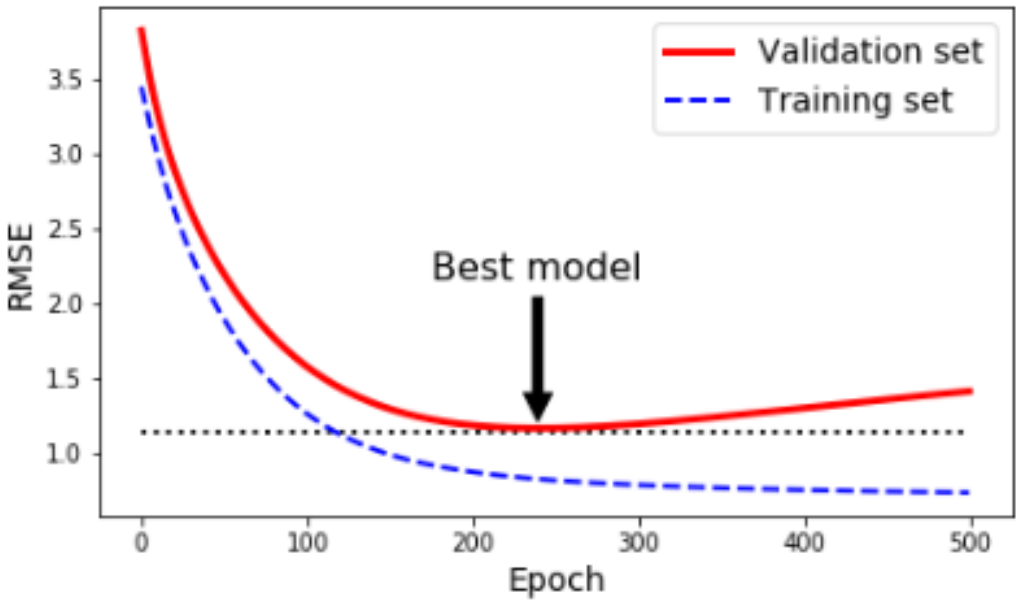


Figur 3.1: Forholdet mellem kapacitet og fejl. Kilde: [1]

3.2 Qb A MSE-Epoch/Error Plot

En *epoch* er en enkelt iteration af træning, hvor hele sættet af træningsdata er anvendt. Det tilfældigt genererede data (`GenerateData()`) er splittet i et træningssæt og et valideringssæt, hvor hvert sæt er lige stort. *mse_train* og *mse_val* er modellens MSE sammenlignet med hhv. træningsdata og valideringsdata.

Baseret på figuren herunder kan man konkludere at den bedste model er opnået efter ca. 230-250 epoch.



3.3 Qc Early Stopping

I det her tilfælde kunne man implementere *Early Stopping* efter *Validation Based Early Stopping* princippet, hvor man stopper med at træne så snart MSE på valideringssættet er højere end det var i forrige epoch, og så bruger man parametre/model fra forrige epoch som sin optimale model.

Output herunder viser udviklingen af *mse_train* og *mse_val* omkring epoch 251-254, hvor man ser at *mse_val* stiger fra 1.35 til 1.36 i epoch 253. Det vil derfor være den model man opnår i epoch 252 som man bruger, og epoch 254 og derefter vil aldrig blive eksekveret.

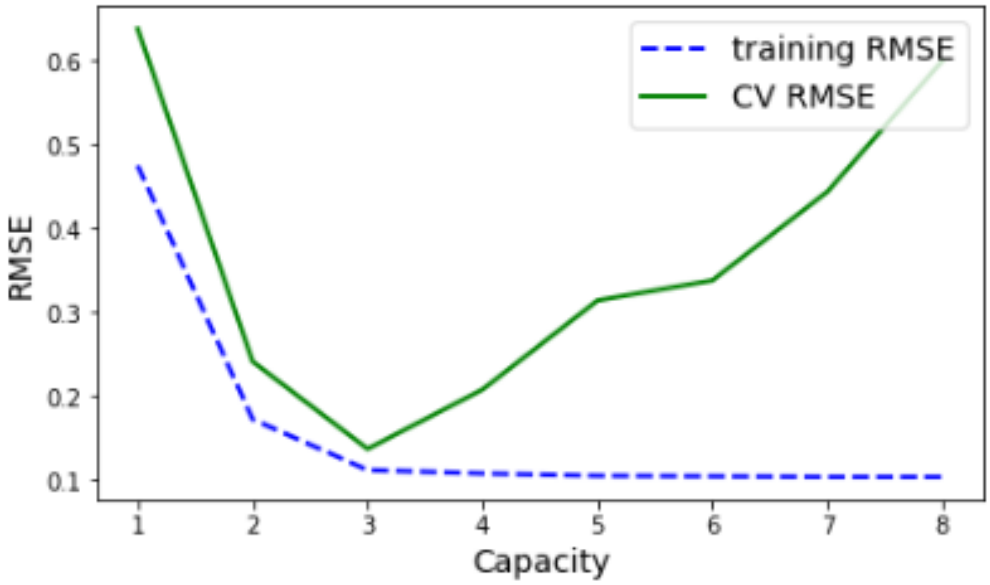
```
epoch= 251, mse_train=0.66, mse_val=1.35
epoch= 252, mse_train=0.66, mse_val=1.35
epoch= 253, mse_train=0.66, mse_val=1.36
epoch= 254, mse_train=0.66, mse_val=1.36
```

3.4 Qd Explain the Polynomial RSME-Capacity plot

I figuren herunder med kapacitet 1-8 ses at RMSE for krydsvalideringen begynder at stige markent når modellen går fra at være tredjegrads til fjerdegradspolynomie. Det indikerer

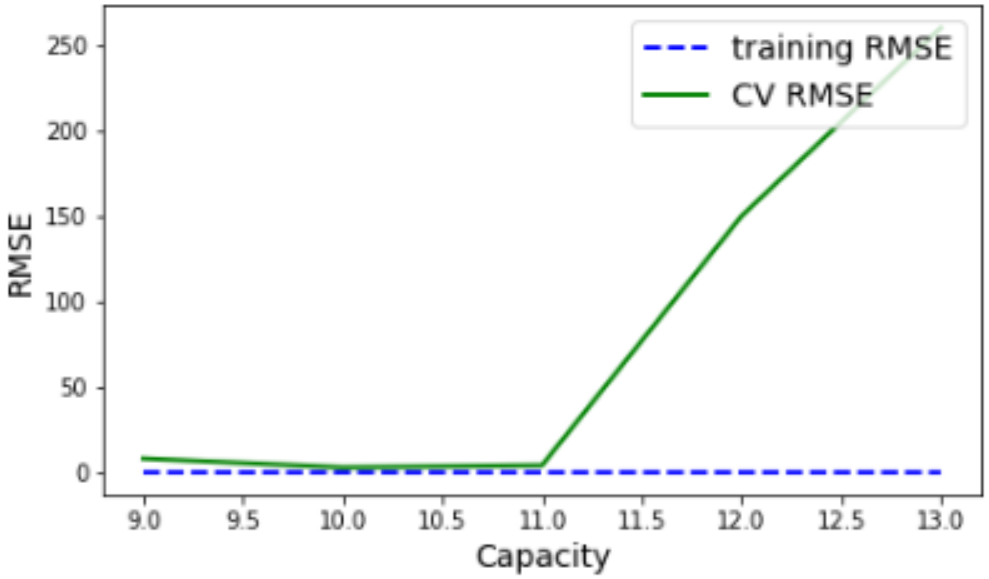
at modellen får for meget kapacitet i forhold til det data den skal approximere. RMSE på træningsdata fortsætter med at falde, fordi modellen *overfitter*.

```
Iterating...degrees= range(1, 9)
  degree=   1, rmse_training=0.48, rmse_cv=0.64
  degree=   2, rmse_training=0.17, rmse_cv=0.24
  degree=   3, rmse_training=0.11, rmse_cv=0.14
  degree=   4, rmse_training=0.11, rmse_cv=0.21
  degree=   5, rmse_training=0.10, rmse_cv=0.31
  degree=   6, rmse_training=0.10, rmse_cv=0.34
  degree=   7, rmse_training=0.10, rmse_cv=0.44
  degree=   8, rmse_training=0.10, rmse_cv=0.60
```



Hvis vi ser på endnu større kapacitet, som i figuren herunder hvor vi ser på grader fra 9-13, så begynder vi at få en nærmest ekspontentiel stigning af RMSE på krydsvalideringen.

```
Iterating...degrees= range(9, 14)
  degree=   9, rmse_training=0.09, rmse_cv=8.21
  degree=  10, rmse_training=0.08, rmse_cv=2.95
  degree=  11, rmse_training=0.08, rmse_cv=4.21
  degree=  12, rmse_training=0.08, rmse_cv=149.71
  degree=  13, rmse_training=0.08, rmse_cv=259.74
```



L06 Naive Bayes Classifier on MNIST 4

I koden herunder er der anvendt 4 forskellige implementationer af naiv bayes classifiers fra Scikit learn biblioteket. Baseret på resultaterne er *MultinomialNB* det bedste valg til MNIST, da den har en høj score, på niveau med *BernoulliNB*, og så er den næsten dobbelt så hurtig som *BernoulliNB*.

```python
from time import time
from sklearn.naive_bayes import BernoulliNB, GaussianNB, MultinomialNB, ComplementNB
from sklearn.metrics import confusion_matrix

BernoulliNB_model = BernoulliNB()
start = time()
BernoulliNB_model.fit(X_train, y_train)
t = time()-start
BernoulliNB_score = BernoulliNB_model.score(X_test,y_test)
print(f'BernoulliNB Score: {BernoulliNB_score}, Time: {t}')

GaussianNB_model = GaussianNB()
start = time()
GaussianNB_model.fit(X_train, y_train)
t = time()-start
GaussianNB_score = GaussianNB_model.score(X_test,y_test)

print(f'GaussianNB Score: {GaussianNB_score}, Time: {t}')

MultinomialNB_model = MultinomialNB()
start = time()
MultinomialNB_model.fit(X_train, y_train)
t = time()-start
MultinomialNB_score = MultinomialNB_model.score(X_test,y_test)
print(f'MultinomialNB Score: {MultinomialNB_score}, Time: {t}')

ComplementNB_model = ComplementNB()
start = time()
ComplementNB_model.fit(X_train, y_train)
t = time()-start
ComplementNB_score = ComplementNB_model.score(X_test,y_test)
print(f'ComplementNB Score: {ComplementNB_score}, Time: {t}')
```

```
BernoulliNB Score: 0.89, Time: 0.24962687492370605
GaussianNB Score: 0.7269523809523809, Time: 0.34546470642089844
MultinomialNB Score: 0.8960952380952381, Time: 0.15369224548339844
ComplementNB Score: 0.8311428571428572, Time: 0.138916015625
```

L07 regulizers 5

5.1 Qa The Penalty Factor

Penalty funktionen $\omega^T\omega$ bliver med numpy i Python

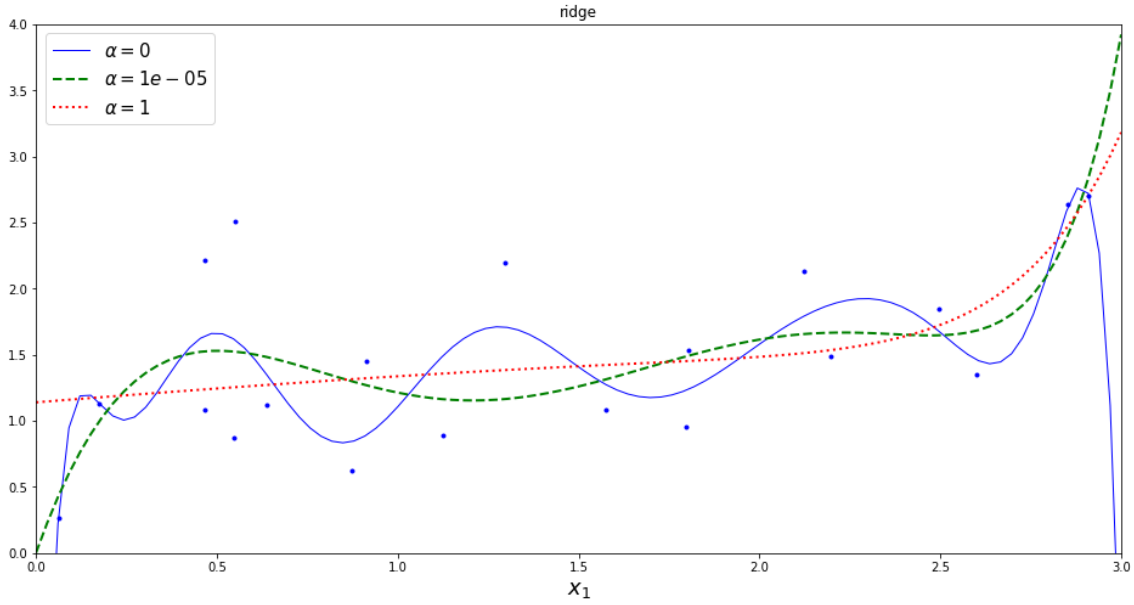
`np.transpose(w[1:]).dot(w[1:])`.

```python
import sys,os
sys.path.append(os.path.abspath('')+'/..')
import numpy as np
from libitmal import utils as itmalutils
from keras.regularizers import l2

def Omega(w):
    return np.transpose(w[1:]).dot(w[1:])

# weight vector format: [w_0 w_1 .. w_d], ie. elem. 0 is the 'bias'
w_a = np.array([1, 2, -3]) #
w_b = np.array([1E10, -3E10])
w_c = np.array([0.1, 0.2, -0.3, 0])

p_a = Omega(w_a)
p_b = Omega(w_b)
p_c = Omega(w_c)

print(f"P(w0)={p_a}")
print(f"P(w1)={p_b}")
print(f"P(w2)={p_c}")

# TEST VECTORS
e0 = 2*2+(-3)*(-3)
e1 = 9e+20
e2 = 0.13
itmalutils.AssertInRange(1.0*p_a,1.0*e0)
itmalutils.AssertInRange(1.0*p_b,1.0*e1,eps=1E5)
itmalutils.AssertInRange(1.0*p_c,1.0*e2)
print("Assertions OK")
```

```
P(w0)=13
P(w1)=9e+20
P(w2)=0.13
Assertions OK
```

5.2 Qb Explain the Ridge Plot

α værdien i *Ridge* plot herunder er *Regularization strength* parameteren som gives til Scikit learn. En større α giver en større regularisering, dvs. en større penalty på store værdier.



5.3 Qc Explain the Ridge, Lasso and ElasticNet Regulized Methods

Ved *Ridge* regularisering er straffen (penalty) lig med summen af den kvadrerede værdi af vægte. Konsekvensen er at egenskaber med lille vægt får meget lille betydning for modellens kompleksitet, hvor større egenskaber har enorm betydning.

Ved *Lasso* (*Least Absolute Shrinkage and Selection Operator Regression*) regularisering er straffen (penalty) lig med summen af den absolutte værdi af vægte. *Lasso* regularisering vil få små vægte til at gå i nul, og kan dermed bruges til at udvælge de signifikante koefficienter.

ElasticNet regularisering er en kombination af L1 (*Lasso*) og L2 (*Ridge*) regularisering, hvor L1 reducerer antallet af vægte, og L2 begrænser størrelsen af vægte.

5.4 Qd Regularization and Overfitting

Regularisering bidrager til at undgå overfitting, ved at trække modellen i retning af en simpel løsning - dels ved at filtrere svage datapunkter fra (L1), og ved at undgå at outliers i træningsdata får for stor betydning (L2).

L07 gridsearch 6

6.1 Qa Explain GridSearchCV

`GridSearchCV` tager en estimator som første argument, hvilket her er en *Support Vector Classifier* trænet på et sæt træningsdata fra IRIS datasættet. Den tager yderligere en krydsvalideringsstrategi, i det her tilfælde sendes tallet '5' ind, hvilket betyder der laves en K-fold krydsvalidering med $K = 5$. Parameteren *scoring* sættes til 'f1_micro', hvilket er en scoring strategi der tæller det totale antal *True Positive*, *False Negative* og *False Positive*. `n_jobs` argumentet indikerer hvor mange tråde der skal bruges, hvor -1 betyder der skal køres en tråd pr. processor kerne der er til rådighed. `iid` argumentet er en boolean der indikerer om score skal være et gennemsmnit på tværs af folder, vægtet efter antallet af data punkter i hver fold (`iid=True`, eller om score skal være et simpelt gennemsnit på tværs af folder (`iid=False`.

I linje 19 herunder kaldes `fit` metoden på `GridSearchCV` objektet, hvilket sætter gang i at træne modellen med det givne input som træningsdata. Modellen trænes for hver mulige kombination af de givne tuning parametre, som er defineret i linjer 8-11. Hver træning af modellen evalueres med $K - fold$ krydsvalidering og scores ud fra 'f1_micro' metoden. Efter kaldet til `fit` indeholder objektet informationer om hvordan de forskellige kombinationer af tuning parametre klarede sig, og man kan dermed nemt evaluere hvilken kombination er bedst egnet til det data man arbejder med.

```
1  # Setup data
2  X_train, X_test, y_train, y_test = LoadAndSetupData('iris') # or 'moon', or 'mnist'
3
4  # Setup search parameters
5  model = svm.SVC(gamma=0.001) # NOTE: gamma="scale" does not work in older Scikit-learn frameworks,
6                               # FIX:  replace with model = svm.SVC(gamma=0.001)
7
8  tuning_parameters = {
9      'kernel':('linear', 'rbf'),
10     'C':[1, 10]
11 }
12
13 CV=5
14 VERBOSE=0
15
16 # Run GridSearchCV for the model
17 start = time()
18 grid_tuned = GridSearchCV(model, tuning_parameters, cv=CV, scoring='f1_micro', verbose=VERBOSE, n_jobs=-1, iid=True)
19 grid_tuned.fit(X_train, y_train)
20 t = time()-start
21
22 # Report result
23 b0, m0= FullReport(grid_tuned , X_test, y_test, t)
24 print('OK')
```

6.2 Qb Hyperparameter Grid Search using an SGD classifier

```
from sklearn import linear_model

model = linear_model.SGDClassifier(max_iter=1000, tol=1e-3, eta0=0.1)

tuning_parameters = {
    'learning_rate':('constant', 'optimal', 'invscaling', 'adaptive'),
    'penalty':('none', 'l2', 'l1', 'elasticnet')
}

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

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

```
SEARCH TIME: 7.64 sec

Best model set found on train set:

        best parameters={'learning_rate': 'adaptive', 'penalty': 'l1'}
        best 'f1_micro' score=0.9809523809523809
        best index=14

Best estimator CTOR:
        SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.1, fit_intercept=True,
       l1_ratio=0.15, learning_rate='adaptive', loss='hinge',
       max_iter=1000, n_iter=None, n_iter_no_change=5, n_jobs=None,
       penalty='l1', power_t=0.5, random_state=None, shuffle=True,
       tol=0.001, validation_fraction=0.1, verbose=0, warm_start=False)

Grid scores ('f1_micro') on development set:
        [ 0]: 0.810 (+/-0.202) for {'learning_rate': 'constant', 'penalty': 'none'}
        [ 1]: 0.686 (+/-0.354) for {'learning_rate': 'constant', 'penalty': 'l2'}
        [ 2]: 0.667 (+/-0.182) for {'learning_rate': 'constant', 'penalty': 'l1'}
        [ 3]: 0.848 (+/-0.251) for {'learning_rate': 'constant', 'penalty': 'elasticnet'}
        [ 4]: 0.800 (+/-0.116) for {'learning_rate': 'optimal', 'penalty': 'none'}
        [ 5]: 0.829 (+/-0.173) for {'learning_rate': 'optimal', 'penalty': 'l2'}
        [ 6]: 0.848 (+/-0.192) for {'learning_rate': 'optimal', 'penalty': 'l1'}
        [ 7]: 0.790 (+/-0.291) for {'learning_rate': 'optimal', 'penalty': 'elasticnet'}
        [ 8]: 0.848 (+/-0.087) for {'learning_rate': 'invscaling', 'penalty': 'none'}
        [ 9]: 0.848 (+/-0.055) for {'learning_rate': 'invscaling', 'penalty': 'l2'}
        [10]: 0.857 (+/-0.072) for {'learning_rate': 'invscaling', 'penalty': 'l1'}
        [11]: 0.810 (+/-0.170) for {'learning_rate': 'invscaling', 'penalty': 'elasticnet'}
        [12]: 0.971 (+/-0.048) for {'learning_rate': 'adaptive', 'penalty': 'none'}
        [13]: 0.971 (+/-0.048) for {'learning_rate': 'adaptive', 'penalty': 'l2'}
        [14]: 0.981 (+/-0.049) for {'learning_rate': 'adaptive', 'penalty': 'l1'}
        [15]: 0.962 (+/-0.070) for {'learning_rate': 'adaptive', 'penalty': 'elasticnet'}

Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       1.00      0.83      0.91        18
           2       0.79      1.00      0.88        11

   micro avg       0.93      0.93      0.93        45
   macro avg       0.93      0.94      0.93        45
weighted avg       0.95      0.93      0.93        45


CTOR for best model: SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.1, fit_intercept=True,
       l1_ratio=0.15, learning_rate='adaptive', loss='hinge',
       max_iter=1000, n_iter=None, n_iter_no_change=5, n_jobs=None,
       penalty='l1', power_t=0.5, random_state=None, shuffle=True,
       tol=0.001, validation_fraction=0.1, verbose=0, warm_start=False)

best: dat=iris, score=0.98095, model=SGDClassifier(learning_rate='adaptive',penalty='l1')
```

6.3 Qc Hyperparameter Random Search using an SGD classifier

Hvis man sammenligner $f1\_micro$ score for *GridSearchCV* og *RandomizedSearchCV* så ender begge modeller ud med præcis samme score:

0.9809523809523809.

```
model = linear_model.SGDClassifier(max_iter=1000, tol=1e-3, eta0=0.1)

tuning_parameters = {
    'learning_rate':('constant', 'optimal', 'invscaling', 'adaptive'),
    'penalty':('none', 'l2', 'l1', 'elasticnet')
}

CV=5
VERBOSE=0

# Run GridSearchCV for the model
start = time()
random_tuned = RandomizedSearchCV(model, tuning_parameters, random_state=42, n_iter=20, cv=CV,
                                  scoring='f1_micro', verbose=VERBOSE, n_jobs=-1, iid=True)
random_tuned.fit(X_train, y_train)
t = time()-start

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

```
SEARCH TIME: 6.35 sec

Best model set found on train set:

        best parameters={'penalty': 'l1', 'learning_rate': 'adaptive'}
        best 'f1_micro' score=0.9809523809523809
        best index=14

Best estimator CTOR:
        SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.1, fit_intercept=True,
       l1_ratio=0.15, learning_rate='adaptive', loss='hinge',
       max_iter=1000, n_iter=None, n_iter_no_change=5, n_jobs=None,
       penalty='l1', power_t=0.5, random_state=None, shuffle=True,
       tol=0.001, validation_fraction=0.1, verbose=0, warm_start=False)

Grid scores ('f1_micro') on development set:
        [ 0]: 0.819 (+/-0.195) for {'penalty': 'none', 'learning_rate': 'constant'}
        [ 1]: 0.781 (+/-0.243) for {'penalty': 'l2', 'learning_rate': 'constant'}
        [ 2]: 0.790 (+/-0.204) for {'penalty': 'l1', 'learning_rate': 'constant'}
        [ 3]: 0.724 (+/-0.162) for {'penalty': 'elasticnet', 'learning_rate': 'constant'}
        [ 4]: 0.848 (+/-0.223) for {'penalty': 'none', 'learning_rate': 'optimal'}
        [ 5]: 0.819 (+/-0.211) for {'penalty': 'l2', 'learning_rate': 'optimal'}
        [ 6]: 0.819 (+/-0.166) for {'penalty': 'l1', 'learning_rate': 'optimal'}
        [ 7]: 0.914 (+/-0.031) for {'penalty': 'elasticnet', 'learning_rate': 'optimal'}
        [ 8]: 0.743 (+/-0.080) for {'penalty': 'none', 'learning_rate': 'invscaling'}
        [ 9]: 0.857 (+/-0.136) for {'penalty': 'l2', 'learning_rate': 'invscaling'}
        [10]: 0.810 (+/-0.135) for {'penalty': 'l1', 'learning_rate': 'invscaling'}
        [11]: 0.886 (+/-0.067) for {'penalty': 'elasticnet', 'learning_rate': 'invscaling'}
        [12]: 0.971 (+/-0.048) for {'penalty': 'none', 'learning_rate': 'adaptive'}
        [13]: 0.971 (+/-0.048) for {'penalty': 'l2', 'learning_rate': 'adaptive'}
        [14]: 0.981 (+/-0.049) for {'penalty': 'l1', 'learning_rate': 'adaptive'}
        [15]: 0.981 (+/-0.049) for {'penalty': 'elasticnet', 'learning_rate': 'adaptive'}

Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       1.00      0.83      0.91        18
           2       0.79      1.00      0.88        11

   micro avg       0.93      0.93      0.93        45
   macro avg       0.93      0.94      0.93        45
weighted avg       0.95      0.93      0.93        45


CTOR for best model: SGDClassifier(alpha=0.0001, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.1, fit_intercept=True,
       l1_ratio=0.15, learning_rate='adaptive', loss='hinge',
       max_iter=1000, n_iter=None, n_iter_no_change=5, n_jobs=None,
       penalty='l1', power_t=0.5, random_state=None, shuffle=True,
       tol=0.001, validation_fraction=0.1, verbose=0, warm_start=False)

best: dat=iris, score=0.98095, model=SGDClassifier(learning_rate='adaptive',penalty='l1')
```

6.4 Qd Search Quest

```
from sklearn.ensemble import RandomForestClassifier

X_train, X_test, y_train, y_test = LoadAndSetupData('mnist')
model = RandomForestClassifier(random_state=0)

tuning_parameters = {
    'n_estimators':(50, 100),
    'max_depth':[1, 2]
}

CV=5
VERBOSE=0

# Run GridSearchCV for the model
start = time()
random_tuned = RandomizedSearchCV(model, tuning_parameters, random_state=42, n_iter=20, cv=CV,
                                  scoring='f1_micro', verbose=VERBOSE, n_jobs=-1, iid=True)
random_tuned.fit(X_train, y_train)
t = time()-start

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

```
SEARCH TIME: 46.20 sec

Best model set found on train set:

        best parameters={'n_estimators': 100, 'max_depth': 2}
        best 'f1_micro' score=0.6306122448979592
        best index=3

Best estimator CTOR:
        RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=2, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
            oob_score=False, random_state=0, verbose=0, warm_start=False)

Grid scores ('f1_micro') on development set:
        [ 0]: 0.526 (+/-0.009) for {'n_estimators': 50, 'max_depth': 1}
        [ 1]: 0.541 (+/-0.022) for {'n_estimators': 100, 'max_depth': 1}
        [ 2]: 0.620 (+/-0.023) for {'n_estimators': 50, 'max_depth': 2}
        [ 3]: 0.631 (+/-0.017) for {'n_estimators': 100, 'max_depth': 2}

Detailed classification report:
        The model is trained on the full development set.
        The scores are computed on the full evaluation set.

              precision    recall  f1-score   support

           0       0.59      0.97      0.74      2008
           1       0.50      0.98      0.66      2412
           2       0.71      0.61      0.65      2045
           3       0.66      0.62      0.64      2082
           4       0.61      0.68      0.64      2078
           5       1.00      0.02      0.05      1920
           6       0.87      0.66      0.75      2078
           7       0.70      0.83      0.76      2191
           8       0.85      0.40      0.55      2093
           9       0.55      0.45      0.50      2093

   micro avg       0.63      0.63      0.63     21000
   macro avg       0.71      0.62      0.59     21000
weighted avg       0.70      0.63      0.60     21000


CTOR for best model: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=2, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
            oob_score=False, random_state=0, verbose=0, warm_start=False)

best: dat=mnist, score=0.63061, model=RandomForestClassifier(max_depth=2,n_estimators=100)
```

L08 PCA Assignment 7

7.1 Qa Speed up by compression

```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.95)
X_train_reduced = pca.fit_transform(X_train)
X_test_reduced = pca.transform(X_test)

logisticRegr = LogisticRegression(solver = 'lbfgs',max_iter = 1000, multi_class = 'multinomial')

time_start = time.time()
logisticRegr.fit(X_train_reduced, y_train)
print('logisticRegr done! Time elapsed: {} seconds'.format(time.time()-time_start))

logisticRegr.score(X_test_reduced, y_test)
```

```
logisticRegr done! Time elapsed: 33.34248995780945 seconds
```

```
/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:758: ConvergenceWarning: lbfg
erge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
```

```
0.9202285714285714
```

7.2 Qa Noise reduction

```
from sklearn.svm import LinearSVC
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score
import time

print("Case 1: The noisy data")

clf = SGDClassifier(random_state=45, max_iter=1000, tol=1e-3)

t0 = time.time()
clf.fit(X_train_noisy, y_train)
t1 = time.time()
print("  Training took {:.2f}s".format(t1 - t0))

y_pred = clf.predict(X_test_noisy)
print("  Accuracy score {:.8f}".format(accuracy_score(y_test, y_pred)))
```

```
Case 1: The noisy data
 SGDClassifier
  Training took 161.40s
  Accuracy score 0.62988571
```

```
print("Case 2: A PCA reduced version of the noisy data")

X_reduced_train = pca.fit_transform(X_train_noisy)
X_reduced_test = pca.fit_transform(X_test_noisy)

clf = SGDClassifier(random_state=45, max_iter=1000, tol=1e-3)

t0 = time.time()
clf.fit(X_reduced_train, y_train)
t1 = time.time()
print("  Training took {:.2f}s".format(t1 - t0))

y_pred = clf.predict(X_reduced_test)
print("  Accuracy score {:.8f}".format(accuracy_score(y_test, y_pred)))
```

```
Case 2: A PCA reduced version of the noisy data
 SGDClassifier
  Training took 22.14s
```

L09 neuron 8

8.1 Qa The Biological Neurons and the Human Brain

En menneskehjerne består af mange milliarder nerveceller, også kaldt neuroner. Neuronerne er indbyrdes forbundet af forbindelser kaldt synapser. En neuron kan have op til 5.000-10.000 indgående synapser, og et tilsvarende antal udgående synapser. Information er lagret i hjernen gennem forbindelsesmønstrene som neuronerne kan danne.

8.2 Qb On Cognition

Kognition er den mentale handling hvor viden og erfaring opbygges. Den viden og erfaring lagres som tidligere nævnt i et forbindelsesmønster af neuroner.

L09 perceptron 9

9.1 Qa Using a Perceptron on the Moon-data

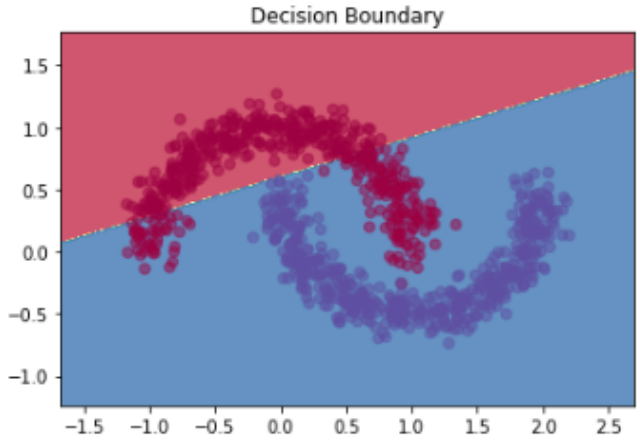
Default score på *Perceptron* fra scikit learn er den gennemsnitlige præcision på det givne data (*mean accuracy*).

```
1  import sys,os
2  sys.path.append(os.path.abspath('')+'/..')
3
4  from sklearn.linear_model import Perceptron, SGDClassifier
5  from sklearn.model_selection import train_test_split
6
7  from libitmal import dataloaders_v3 as itmaldataloaders
8
9  X, y = itmaldataloaders.MOON_GetDataSet(n_samples=1000)
10
11 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
12
13 model = Perceptron(tol=1e-3, random_state=42)
14 model.fit(X_train, y_train)
15 print(model.score(X_test, y_test))
```

```
0.7933333333333333
```

9.2 Qb Plot the Decision Boundary

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

# Helper function to plot a decision boundary.
def plot_decision_boundary(pred_func):
    # Set min and max values and give it some padding
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral,alpha=.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Spectral, alpha=.5)

# Predict and plot decision boundary
plot_decision_boundary(lambda x: model.predict(x))
plt.title("Decision Boundary")
plt.show()
```



9.3 Qc The Perceptron on an XOR-problem

Det er ikke muligt at få en tilfredsstillende predictor til MOON datasættet med en *Perceptron*, da den som udgangspunkt er en lineær predictor.

Det er dog muligt at opbygge en *Multi-Layer Perceptron* med flere lag *Perceptron*, hvor outputtet fra første lag bruges som input til det næste lag.

9.4 Qd Compare the Perceptron to the SGD

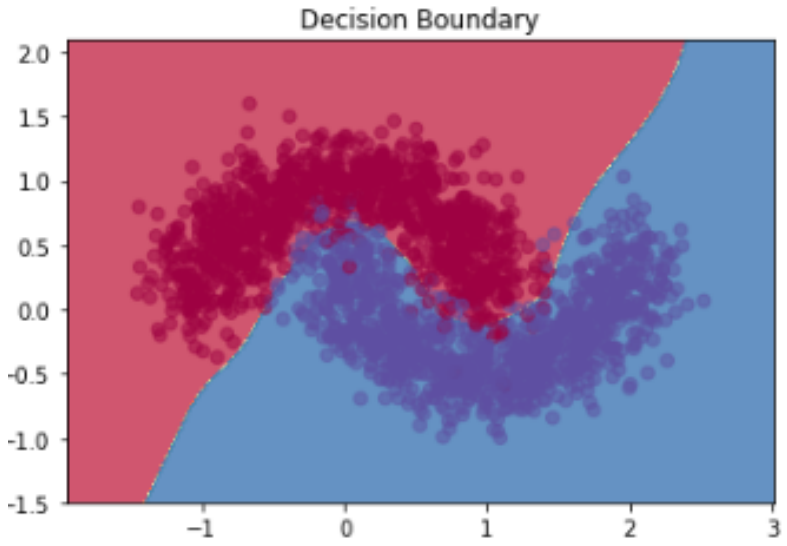
En *SGDClassifier* bruger gennemsnitlig præcision som score metric, hvilket er det samme som *Perceptron*. Den opnåede score er 100% magen til det vi så med *Perceptron* i opgave Qa.

```
model = SGDClassifier(loss='perceptron', eta0=1, learning_rate='constant', penalty=None, tol=1e-3, random_state=42)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

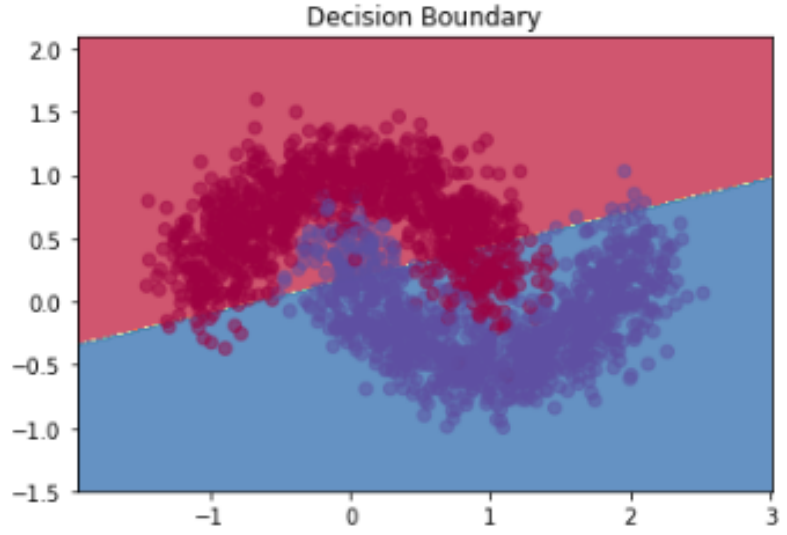
```
0.7933333333333333
```

L09 keras_mlp_moon 10

10.1 Qa Using a Keras MLP on the Moon-data



Klassificeringsgrænse for *Adam* optimizer.



Klassificeringsgrænse for *SGD* optimizer.

I figuren herunder er der lavet et par ændringer for at forbedre performance af SGD

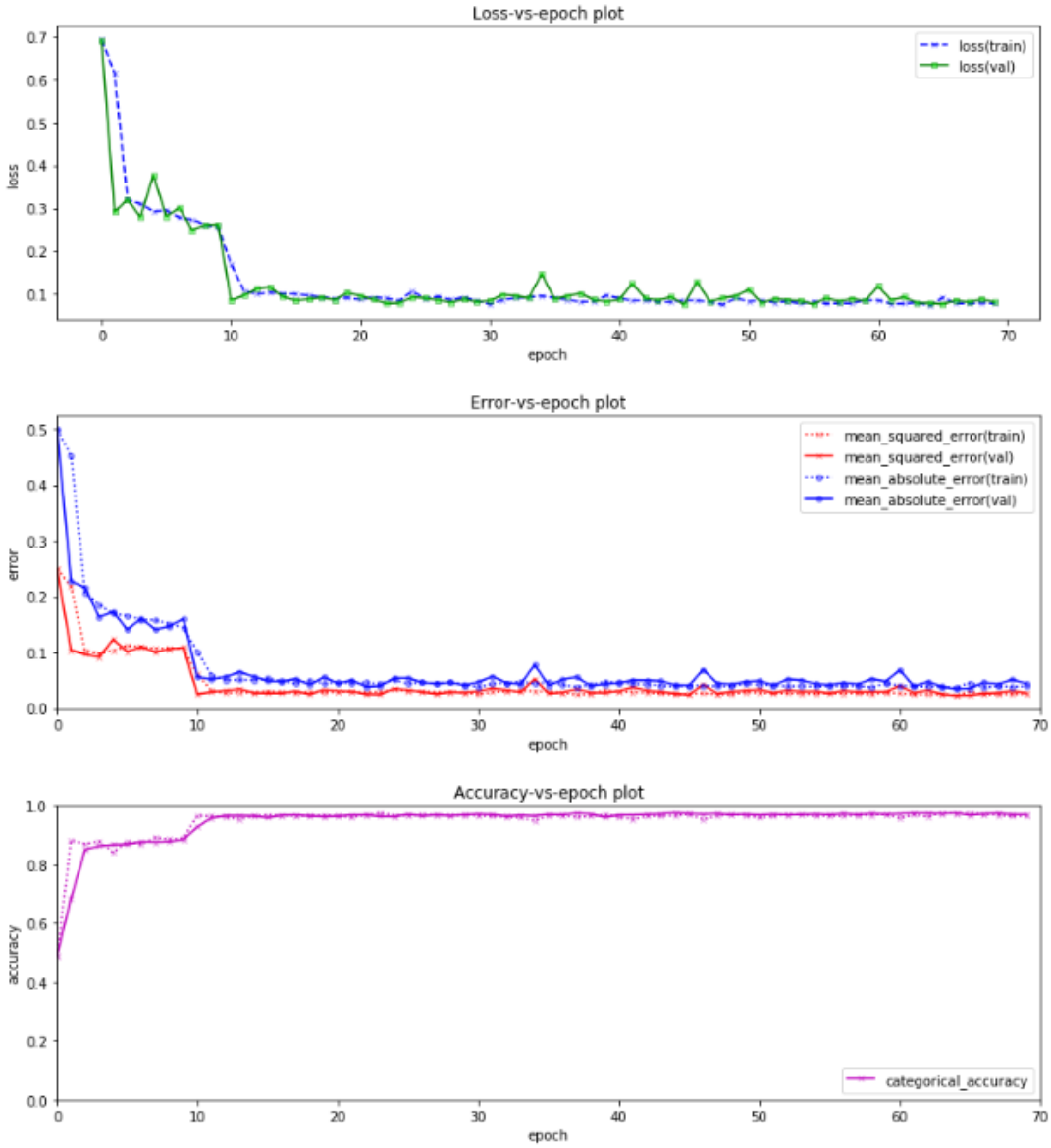
optimizeren. Der er tilføjet 2 mellem-lag af neuroner, *activation* er ændret til *Rectified Linear Unit* for mellemlag, og til *sigmoid* for output laget.

```
# Build Keras model
model = Sequential()
model.add(Dense(input_dim=2, units=16, activation="relu", kernel_initializer="normal"))
model.add(Dense(input_dim=16, units=32, activation="relu", kernel_initializer="normal"))
model.add(Dense(input_dim=16, units=32, activation="relu", kernel_initializer="normal"))
model.add(Dense(units=2, activation="sigmoid"))

optimizer = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
#optimizer = Adam(lr=0.1)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['categorical_accuracy', 'mean_squared_error', 'mean_absolute_error'])

# Make data
X, y = datasets.make_moons(2000, noise=0.20, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

y_train_binary = to_categorical(y_train)
y_test_binary  = to_categorical(y_test)

assert y.ndim==1
assert y_train_binary.ndim==2
assert y_test_binary.ndim ==2

# Train
VERBOSE     = 0
EPOCHS      = 70

start = time()
history = model.fit(X_train, y_train_binary, validation_data=(X_test, y_test_binary), epochs=EPOCHS, verbose=VERBOSE)
t = time()-start

print(f"OK, training time={t:0.1f}")
```

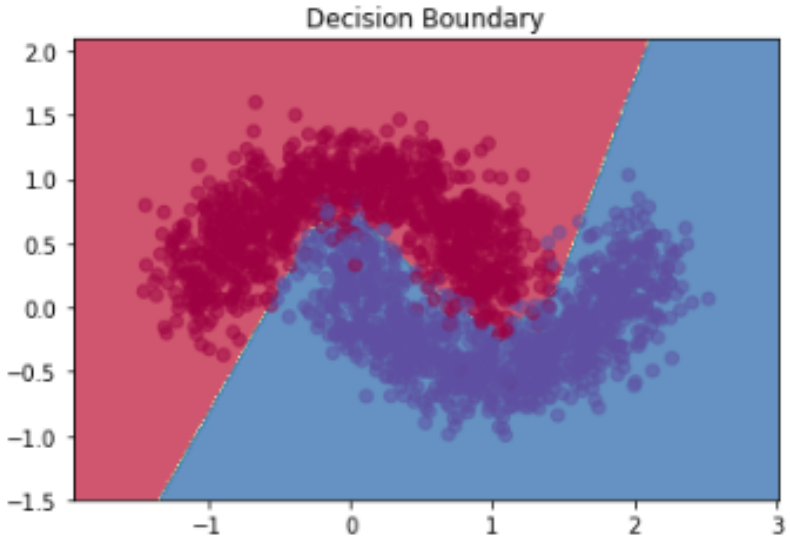
```
OK, training time=7.3
```

Den opnåede præcision med ovenstående ændringer er 0.9633.

```
Training time: 7.3 sec
Test loss:     0.08104073598980904
Test accuracy: 0.9633333325386048
All scores in history: [0.08104073598980904, 0.9633333325386048, 0.02761055441573262, 0.04441211725274722]
```



Herunder er klassificeringsgrænsen illustreret med SGD inklusiv ændringerne til keras modellen.



10.2 Qb Keras and Classification Categories

One-hot encoding er når man tager et array med værdier der repræsenterer kategorier, og konverterer det til en matrice.

Hvis man f.eks. forestiller sig at et input kan være fra 0-3, og hvert tal repræsenterer en specifik by. Den numeriske værdi har dermed ingen betydning andet end at symbolisere en kategori, og det kan derfor være vildledende at give de numeriske tal som input til en ML metode.

Herunder ses et konkret eksempel af input/output til `to_categorical` funktionen. Matricen har 0 på alle positioner, undtagen den ene position i hver række som svarer til den numeriske repræsentation af kategorien, så 1 bliver til [0.1.0.0.] osv.

```
1 print(to_categorical([1, 2, 0, 1, 2, 0, 3]))
```

```
[[0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]]
```

`categorical_accuracy` er beregnet som metric til evaluering af kategori inputs, og den evaluerer om index af den største vægt i forudsigelsesmatricen svarer til index af den største vægt i valideringsmatricen.

10.3 Qc Optimize the Keras Model

Increasing/decreasing the number of epochs En fordobling af epochs gør ikke nogen væsentlig forskel, men sætter man den højt nok er det ikke længere muligt at generere plots.

Adding more neurons per layer Der er ingen forskel på at fordoble antallet af neuroner pr. lag.

adding whole new layers Tilføjer man et skjult lag neuroner mere, med samme antal input og neuroner som det eksisterende skjulte lag, så falder score en smule. Tilføjer man endnu et skjult lag med samme konfiguration så falder score meget. Ændrer man parametre til et lavere antal input og antal neuroner så kommer *test accuracy* score tilbage op omkring de 96%.

changing the activation functions in the layers *softmax* ser udemærket ud. *tanh*, *sigmoid* og *softsign* klarer sig godt i forhold til score, men på loss/errror/accuracy vs epoch graferne ligner det at modellen overfitter. *selu*, *elu*, *softplus*, *hard_sigmoid* og *exponential* er alle et meget dårligt fit, med en score på 50%.

Changing the output activation from activation="sigmoid"to something else Hvis output aktiveringsfunktionen ændres til softmax får man en model der overfitter, med *hard_sigmoid* bliver fejlen meget stor efter de første 30 epoch.

L09 keras_mlp_mnist 11

11.1 Qa Using a Keras MLP on the MNIST-data

```
from __future__ import print_function
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

batch_size = 128
num_classes = 10
epochs = 12

img_rows, img_cols = 28, 28

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

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
36
37 model = Sequential()
38 model.add(Conv2D(32, kernel_size=(3, 3),
39                  activation='relu',
40                  input_shape=input_shape))
41 model.add(Conv2D(64, (3, 3), activation='relu'))
42 model.add(MaxPooling2D(pool_size=(2, 2)))
43 model.add(Dropout(0.25))
44 model.add(Flatten())
45 model.add(Dense(128, activation='relu'))
46 model.add(Dropout(0.5))
47 model.add(Dense(num_classes, activation='softmax'))
48
49 model.compile(loss=keras.losses.categorical_crossentropy,
50               optimizer=keras.optimizers.Adadelta(),
51               metrics=['accuracy'])
52
53 model.fit(x_train, y_train,
54           batch_size=batch_size,
55           epochs=epochs,
56           verbose=1,
57           validation_data=(x_test, y_test))
58 score = model.evaluate(x_test, y_test, verbose=0)
59 print('Test loss:', score[0])
60 print('Test accuracy:', score[1])
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 114s 2ms/step - loss: 0.2583 - acc: 0.9217 - val_loss: 0.0543 - val_acc: 0.9818
Epoch 2/12
60000/60000 [==============================] - 111s 2ms/step - loss: 0.0890 - acc: 0.9734 - val_loss: 0.0394 - val_acc: 0.9868
Epoch 3/12
60000/60000 [==============================] - 113s 2ms/step - loss: 0.0650 - acc: 0.9802 - val_loss: 0.0341 - val_acc: 0.9886
Epoch 4/12
60000/60000 [==============================] - 112s 2ms/step - loss: 0.0537 - acc: 0.9835 - val_loss: 0.0307 - val_acc: 0.9900
Epoch 5/12
60000/60000 [==============================] - 108s 2ms/step - loss: 0.0474 - acc: 0.9860 - val_loss: 0.0289 - val_acc: 0.9908
Epoch 6/12
60000/60000 [==============================] - 107s 2ms/step - loss: 0.0421 - acc: 0.9872 - val_loss: 0.0273 - val_acc: 0.9914
Epoch 7/12
60000/60000 [==============================] - 107s 2ms/step - loss: 0.0352 - acc: 0.9891 - val_loss: 0.0265 - val_acc: 0.9912
Epoch 8/12
60000/60000 [==============================] - 107s 2ms/step - loss: 0.0337 - acc: 0.9895 - val_loss: 0.0261 - val_acc: 0.9909
Epoch 9/12
60000/60000 [==============================] - 107s 2ms/step - loss: 0.0320 - acc: 0.9901 - val_loss: 0.0269 - val_acc: 0.9912
Epoch 10/12
60000/60000 [==============================] - 106s 2ms/step - loss: 0.0287 - acc: 0.9914 - val_loss: 0.0285 - val_acc: 0.9904
Epoch 11/12
60000/60000 [==============================] - 104s 2ms/step - loss: 0.0273 - acc: 0.9916 - val_loss: 0.0273 - val_acc: 0.9913
Epoch 12/12
60000/60000 [==============================] - 104s 2ms/step - loss: 0.0248 - acc: 0.9923 - val_loss: 0.0251 - val_acc: 0.9919
Test loss: 0.025122384592889468
Test accuracy: 0.9919
```

11.2 Qb Repeat Grp10's Go at the Search Quest

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Conv2D(30, (5, 5), activation='relu', input_shape=(28,28,1)))
model.add(Flatten())
model.add(Dense(20, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(70, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(70, activation='relu'))
model.add(Dense(50, activation='relu'))
model.add(Dense(num_classes, activation='sigmoid'))
model.summary()

sgd = keras.optimizers.SGD(lr=0.001, decay=1e-08, momentum=0.9, nesterov=True)
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 24, 24, 30)        780
_________________________________________________________________
flatten_2 (Flatten)          (None, 17280)             0
_________________________________________________________________
dense_3 (Dense)              (None, 20)                345620
_________________________________________________________________
dense_4 (Dense)              (None, 50)                1050
_________________________________________________________________
dense_5 (Dense)              (None, 70)                3570
_________________________________________________________________
dense_6 (Dense)              (None, 100)               7100
_________________________________________________________________
dense_7 (Dense)              (None, 70)                7070
_________________________________________________________________
dense_8 (Dense)              (None, 50)                3550
_________________________________________________________________
dense_9 (Dense)              (None, 10)                510
=================================================================
Total params: 369,250
Trainable params: 369,250
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
60000/60000 [==============================] - 18s 299us/step - loss: 2.2748 - acc: 0.2843 - val_loss: 2.2051 - val_acc: 0.3950
Epoch 2/12
60000/60000 [==============================] - 18s 293us/step - loss: nan - acc: 0.4850 - val_loss: nan - val_acc: 0.0980
Epoch 3/12
60000/60000 [==============================] - 18s 293us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 4/12
60000/60000 [==============================] - 18s 292us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 5/12
60000/60000 [==============================] - 18s 292us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 6/12
60000/60000 [==============================] - 18s 292us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 7/12
60000/60000 [==============================] - 17s 292us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 8/12
60000/60000 [==============================] - 18s 294us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 9/12
60000/60000 [==============================] - 18s 293us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 10/12
60000/60000 [==============================] - 18s 292us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 11/12
60000/60000 [==============================] - 18s 292us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Epoch 12/12
60000/60000 [==============================] - 18s 294us/step - loss: nan - acc: 0.0987 - val_loss: nan - val_acc: 0.0980
Test loss: nan
Test accuracy: 0.098
```