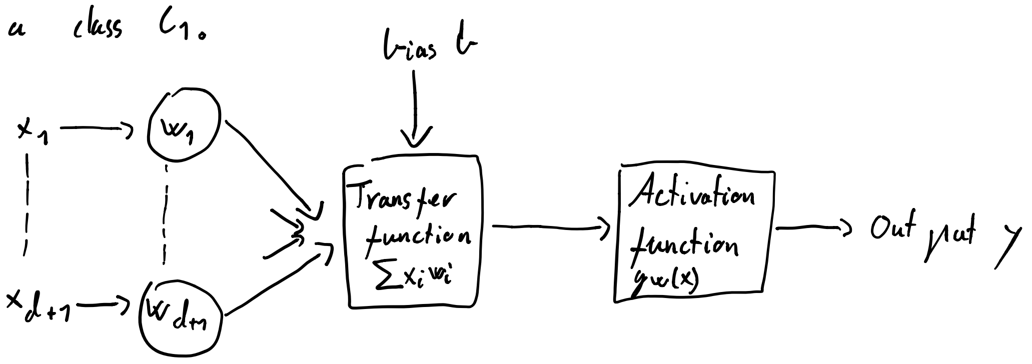
## 1 Logistic Regression.

Logistic regression is a tool for binary classification that "uses" one neuron to determin the probability that input x belong in a class $C_1$.


bias b
$x_1$
$w_1$
$x_{d+1}$
$w_{d+1}$
Transfer function $\sum x_i w_i$
Activation function $g_w(x)$
Output y

We have : $P(x \in C_1 | x) = g_w(x) = \dfrac{1}{1+e^{-w^T x}}$

$P(x \in C_2 | x) = 1 - P(x \in C_1 | x) = 1 - g_w(x)$

$g_w(x) \in [0,1]$

Loss function $E(w) = -\dfrac{1}{N} \sum_{n=1}^{N} t^n \ln(y^n) + (1-t^n)\ln(1-t^n)$

measures how well our hypothesis function $g_w(x) = y^n$ does on $N$ data points. $t^n$ = target value for example n

Goal: $t_n = y^n$ for all n.

* Softmax regression is a generalization of logistic regression to a multi-class classification.

## 1.1

Show that $-\frac{\partial E^n(w)}{\partial w_j} = (t^n - y^n)\, x_j^n$

We have $E^n(w) = -t^n \ln(g_w^n) + (1-t^n)\ln(1-t^n)$

$$\Longrightarrow \quad \frac{\partial E^n(w)}{\partial w_j} = -t^n \frac{1}{g_w^n} \frac{\partial g_w^n}{\partial w_j} \qquad \text{Applies chain rule.}$$

Using the hint : $\frac{\partial E^n(w)}{\partial w_j} = -t^n \frac{1}{\cancel{g_w^n}} x_j^n \cancel{g_w^n} (1 - g_w^n)$

$$-\frac{\partial E^n(w)}{\partial w_j} = t^n x_j^n (1 - g_w^n) = (t^n - y^n)\, x_j^n$$

## 1.2

We have for the Soft max regression cost function

$$E = -\sum_{n=1}^{N}\sum_{K'=1}^{C} t_{K'}^n \ln(y_{K'}^n) + t_K^n \ln(y_K^n) \quad , \quad K' \neq K$$

where : $y_K^n = \frac{e^{a_K^n}}{\sum_{K'} e^{a_{K'}^n}}$ is the Softmax function with

$a_K^n = w_K^T x^n$ called net input to $y_K$ and

$y_K^n$ is the probability that $x$ is a member of class $K$. Note $\sum^K y_K^n = 1$

We have two cases : $K \neq K'$ and $K = K'$,

hence we get :

Forsøk 3:

$$E^n(w) = -\overbrace{\sum_{k'=1}^{c} t^n_{k'} \ln(y^n_{k'})}^{\text{Case } k' \neq k} + \overbrace{t^n_k \ln(y^n_k)}^{\text{Case } k' = k}$$

$$= -\sum_{k'=1}^{c} t^n_{k'} \left( \ln(e^{w^T_{k'} x^n}) - \ln\left( \sum_{k'} e^{w^T_{k'} x^n} \right) \right) + t^n_k \ln\left( \frac{e^{w^T_k x^n}}{\sum_{k'} e^{w^T_{k'} x^n}} \right)$$

$$\frac{\partial E^n(w)}{\partial w_{kj}} = -\sum_{k=1}^{c} t^n_{k'} \left( \cancel{\left( \frac{1}{\cancel{e^{w^T_k x^n}}} \cdot \cancel{e^{w^T_k x^n}} \cdot x^n_j \right)} - \left( \frac{1}{\sum e^{w^T_{k'} x^n}} e^{w^T_k x^n} x^n_j \right) \right) +$$

$$t^n_k \left( \left( \frac{1}{\cancel{e^{w^T_k x^n}}} \cdot \cancel{e^{w^T_k x^n}} \cdot x^n_j \right) - \left( \frac{1}{\sum e^{w^T_{k'} x^n}} e^{w^T_k x^n} x^n_j \right) \right)$$

$$\frac{\partial E^n(w)}{\partial w_{kj}} = -\sum_{k'=1}^{c} t^n_{k'} \left( x^n_j y^n_k \right) + t^n_k x^n_j (1 - y^n_k)$$

$$-\frac{\partial E^n(w)}{\partial w_{kj}} = x^n_j y^n_k \sum_{k'=1}^{c} t^n_{k'} + t^n_k x^n_j - x^n_j y^n_k$$

$$= x^n_j y^n_k \left( \cancel{\sum t^n_{k'}} - 1 \right) + t^n_k x^n_j$$

$$= \underline{\underline{x^n_j (t^n_k - y^n_k)}}$$

## Task 2.2

$$C(w) = \|w_{i,j}\|^2 = \sum_{i,j} w_{i,j}^2$$

$$\frac{\partial C(w)}{\partial w_{ij}} = 2 w_{i,j}$$

Gradient descent with regularization then becomes:

$$dw = \frac{dE(w)}{dw} + \lambda \frac{d\,C(w)}{dw}$$

# TDT 4265 Assignment 1

Kristian Haga

January 2019

## 1 Task 2

### 1.1 Preparing data sets

After downloading the MNIST data set using the handout code the solution normalizes the data set. Since the image set contains images with pixel values 0-255, the set is normalized by divinding all values by 255. This ensures that all pixel values is in the interval $[0, 1]$. Normalizing the code avoids large input values to carry a heavier weight when calculating the gradient descent and such. This again enhance learning and thus the result.

Then the inputs are filtered for 2's and 3's to satisfy the preconditions for the task, before setting the initial weights as a $1x784$ vector where each entry is randomly set as a value in the interval $(-1, 1)$. Sets weights random to ensure that weights will be updated.

### 1.2 Training loop

The training loop consist of a nested for-loop. The outer loop represent an epoch, while the inner loop represent what happens in each epoch.

Before entering any of the loops, the solution divides the data set into a training set and an validation set. This is to avoid data snooping when validating the loss of the network.

#### 1.2.1 Outer loop

In each iteration the training data is shuffled intentionally, but n this specific task I observed a larger loss when shuffling the sets.

After running the inner loop, which updates the weights, the solution computes and logs the loss function on the whole training set, the validation set and the training set.

Finally the outer loop implements early stopping by checking if the computed losses in the three last epochs have consistently risen. If this occurred the training is ended and the weights resulting in minimal loss is returned. Early stopping is a regularization mechanism to avoid over fitting. We did not observe early stopping ever kicking in during the training, even though it was implemented.

### 1.2.2 Inner loop

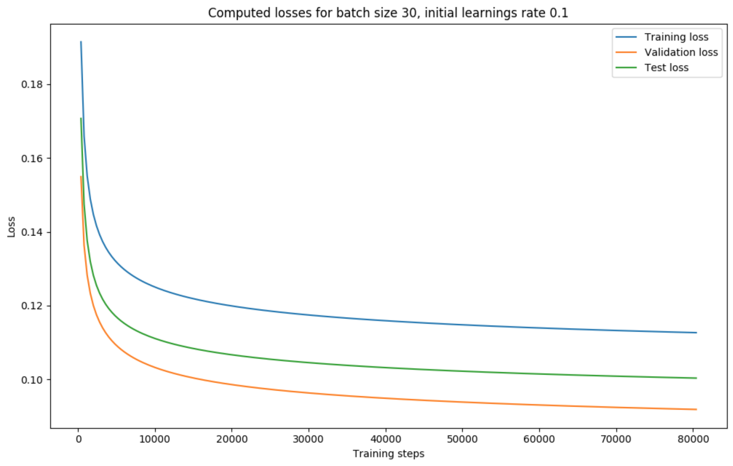
In the inner loop the training set is divided into a mini batch. Using mini batches the solution implements the trade-of between spending a lot of time and memory computing the gradient for the whole training set giving an optimal descent direction, and rapidly, over a smaller input set, computing a non-optimal gradient descent direction. This can be viewed as a person walking in a straight line from A to B, compared to a drunk stumbling back and forth, left and right from A to B, but the drunk does in fact reach B. The use of mini batches introduces a noise in the gradient descent which will ensure that the solution not get stuck in local minimum and saddle point.

The inner loop then computes the gradient descent with L2 regularization and updating the weights. The learning rate used to update the weights is found by applying the annealing learning rate function. We found the best initial learning rate to be 0.1. This learning rate gave a loss of 0.10024 on the test set. The T value in the annealing rate function gave best results when being set to the number of epochs. Here 200.

0.001 -¿ 0.7305926371057812 0.01 -¿ 0.28026005341781535 0.1 -¿ 10034499334081994

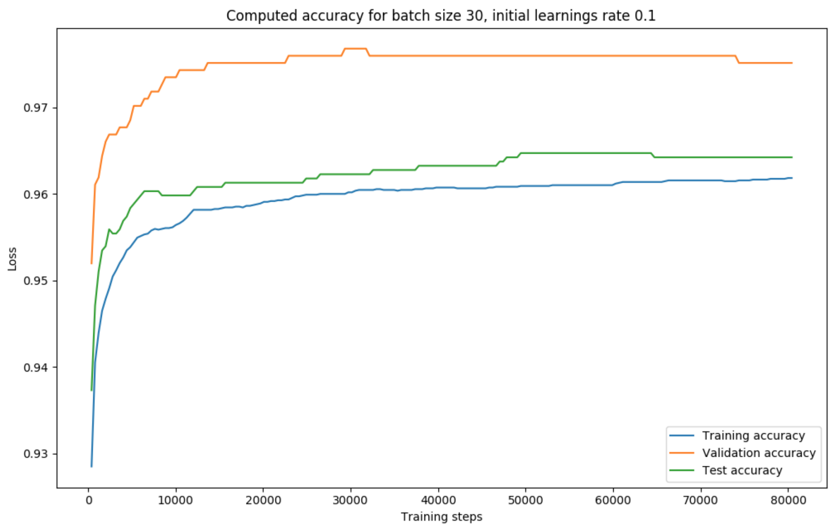
## 1.3 Task 2.1 a

We obtained the following losses when using the initial learning rate of 0.1, batch size 30 and the number of epochs set to 200.



## 1.4 Task 2.1 b

Plot showing accuracy of the solution with same parameters as in 2.1 a

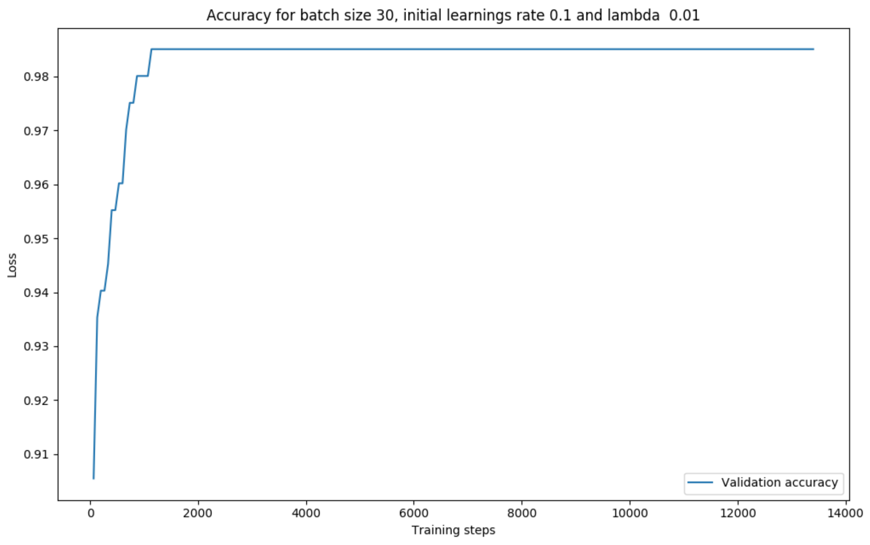


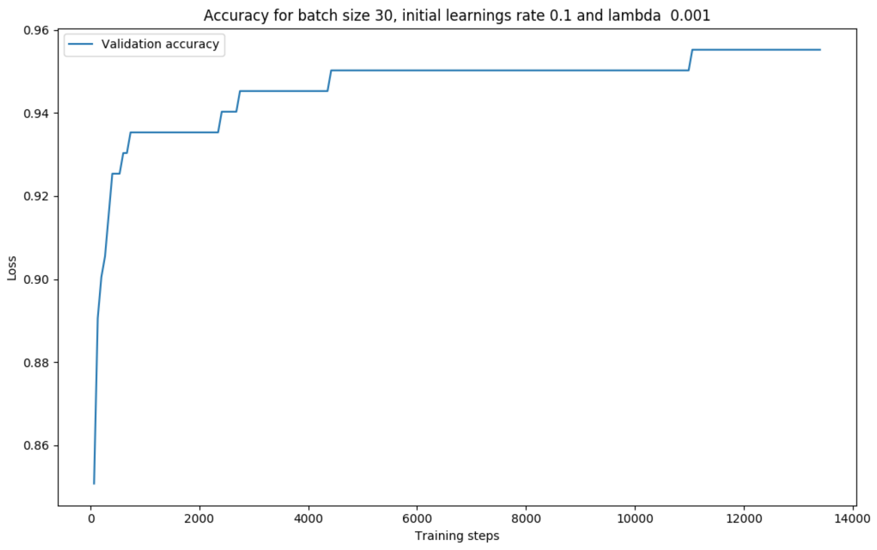
## 1.5 Task 2.2 a

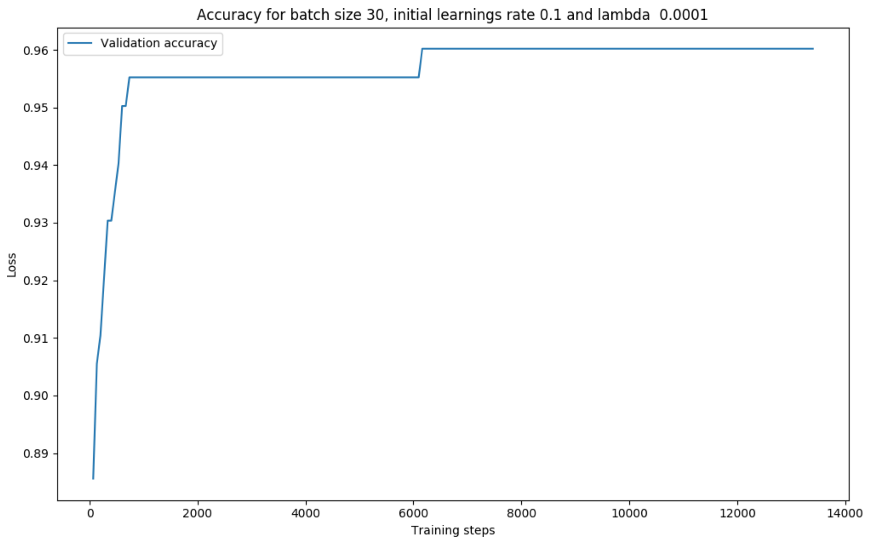
This task was solved in the hand written pages at the beginning of this document.

## 1.6 Task 2.2 b

Plots showing the accuracy on the validation set with different values for lambda in the regularization function.



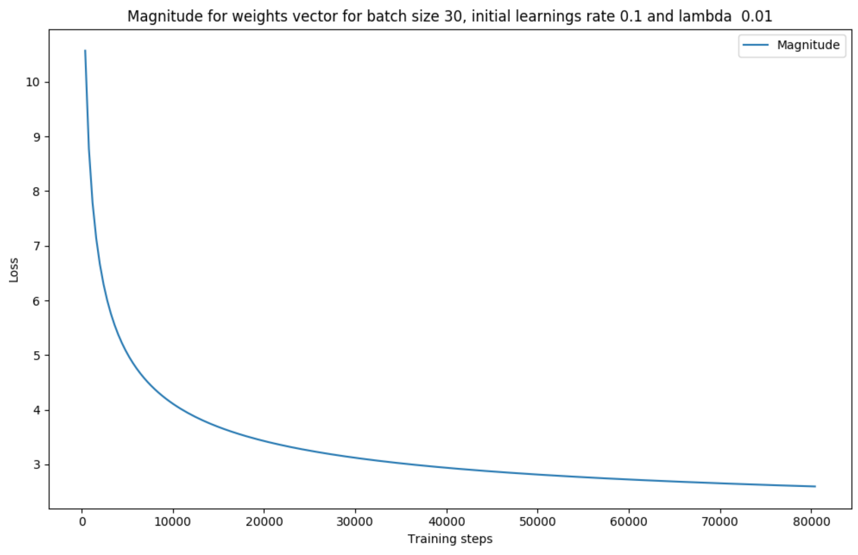


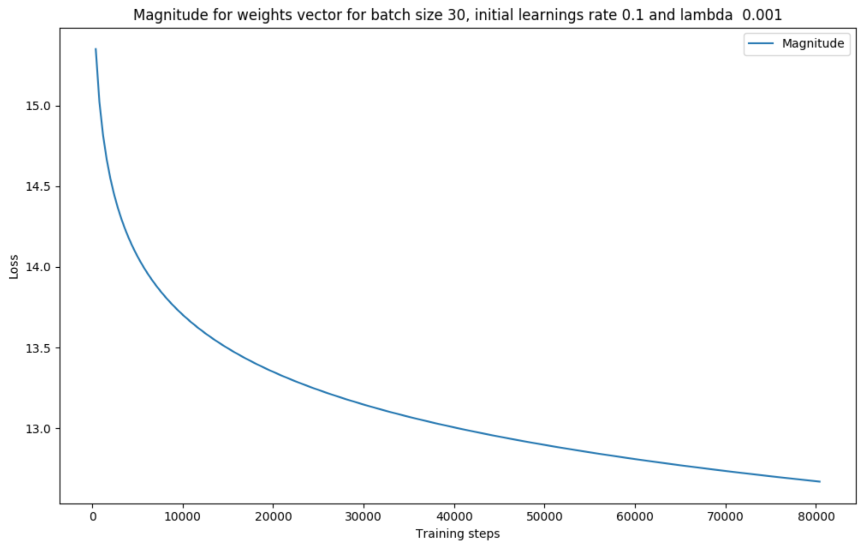


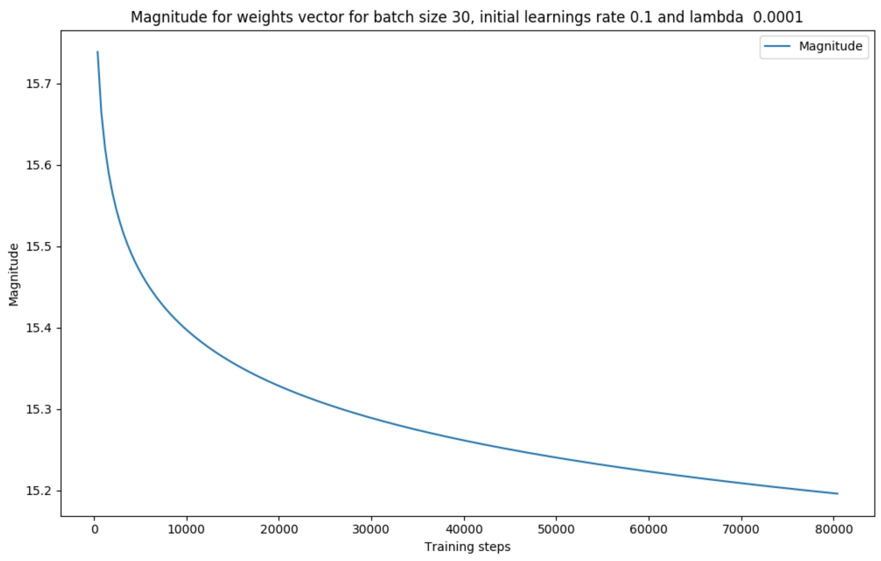
We observe that the smaller lambda, the better.

## 1.7 Task 2.2 c

Plot showing the magnitude of the weights vector during training for different lambdas. Do note that the y-axis is labeled loss in the two first figures. This should be labeled magnitude, but there is no time.


Magnitude for weights vector for batch size 30, initial learnings rate 0.1 and lambda 0.01
Magnitude
Loss
Training steps
0
10000
20000
30000
40000
50000
60000
70000
80000
3
4
5
6
7
8
9
10

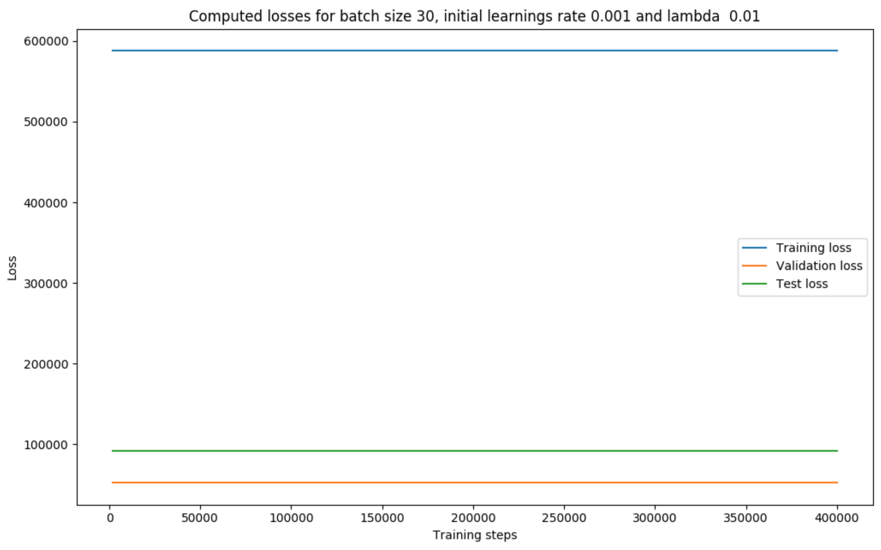

Magnitude for weights vector for batch size 30, initial learnings rate 0.1 and lambda 0.001
Magnitude
Loss
Training steps
0
10000
20000
30000
40000
50000
60000
70000
80000
13.0
13.5
14.0
14.5
15.0



We observe that the magnitude of the weights vector increase as lambda increases.

## 2 Task 3 Softmax

Running the code for the Softmax function i receive extremely high losses and there is clearly something wrong. Hope your review of the code will score me some points.


Computed losses for batch size 30, initial learnings rate 0.001 and lambda 0.01
Loss
Training steps
Training loss
Validation loss
Test loss