Machine Learning - FMAN45
Assignment 3, spring 2019

Backpropagation and Neural Networks
Classifying Handwritten Digits with CNN
Classifying Tiny Images with CNN

Hicham Mohamad - hi8826mo-s
hsmo@kth.se

April 5, 2020

1 Introduction

In this assignment, we are going to work on the Backpropagation algorithm in `Artificial Neural Network`. In the first part, we consider the implementation of the forward and backward passes. In the second part, we deal with classification tasks, that involves the `MNIST` image dataset of handwritten digits using the convolutional approach of neural networks CNN.

2 Objectives

The objectives of this assignment are to:

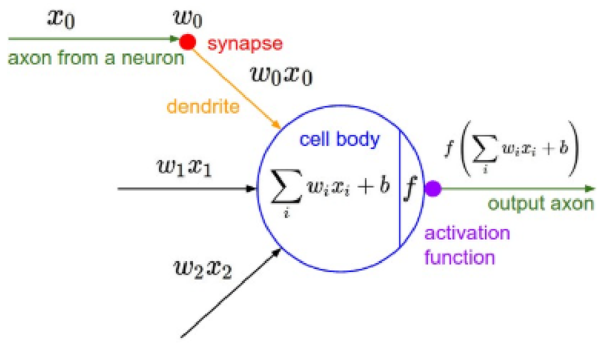
1. simplify the expressions for $\frac{\partial L}{\partial \mathbf{x}}$, $\frac{\partial L}{\partial \mathbf{A}}$ and $\frac{\partial L}{\partial \mathbf{b}}$.
2. vectorize $\frac{\partial L}{\partial \mathbf{x}}$, $\frac{\partial L}{\partial \mathbf{A}}$ and $\frac{\partial L}{\partial \mathbf{b}}$, and implement the forward and backward steps.
3. drive the backpropagation expression for $\frac{\partial L}{\partial \mathbf{x}_i}$ using the rectified linear unit Relu. .
4. compute expression for $\frac{\partial L}{\partial \mathbf{x}_i}$ using the Softmax loss.
5. implement gradient descent with momentum in `training.m`.
6. validate the 98% accuracy on a simple baseline `mnist_starter.m`
7. improve the accuracy of the baseline network `cifar10_starter`.

3 Background - Artificial Neural Netorks, ANN

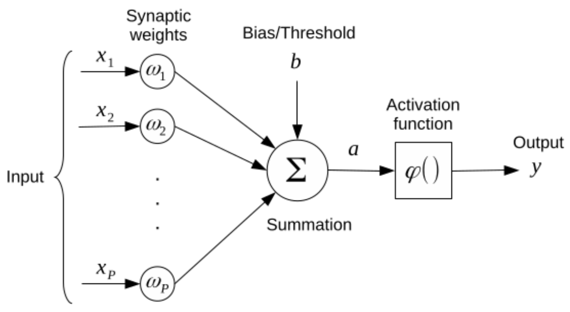
Artificial neural networks (ANN) is a collection name for many algorithms that (almost) all have in common that they are inspired from the construction and functioning of the human brain. Figure **??** shows the basic element of an ANN, the artificial neuron. The output of the neuron is calculated as

$$a = \sum_{k=1}^{P} w_k x_k + b, \qquad y = \phi(a) \tag{1}$$

So what neurons do is a *weighted summation* of its incoming signals and then outputs a new signal. The weights w_k can be both positive and negative. The function $\phi(x)$ have many names, e.g. transfer function, *activation function* or node function. Below a list of a few activation functions are plotted in Figure **??** .



(a)



(b)

Figure 1: a) Illustration of a pyramidal cortical neuron. b) The basic element of the ANN, the mathematical neuron model. The basic computational unit of the brain is a neuron. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately $10^{14} - 10^{15}$ *synapses. Each neuron receives input signals from its dendrites and produces output signals along its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons. In the computational model of a neuron, the signals that travel along the axons (e.g.* x_0*) interact multiplicatively (e.g.* w_0x_0*) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g.* w_0*).*

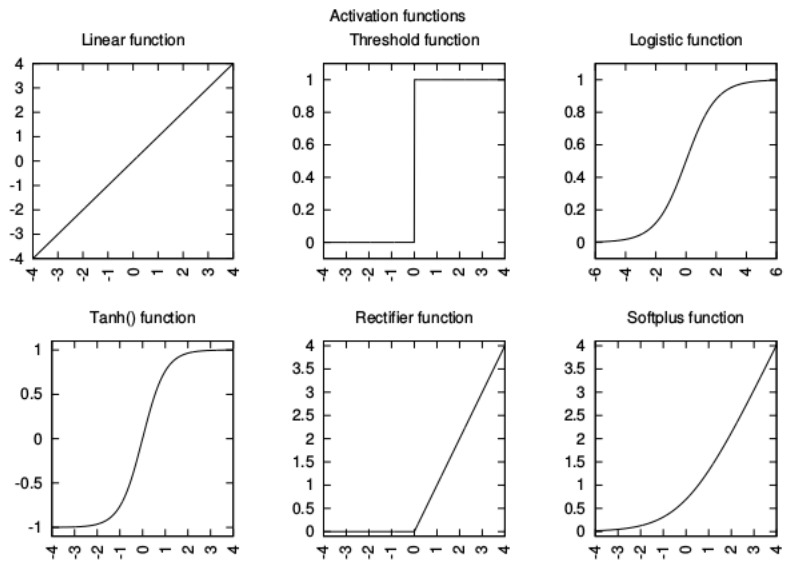


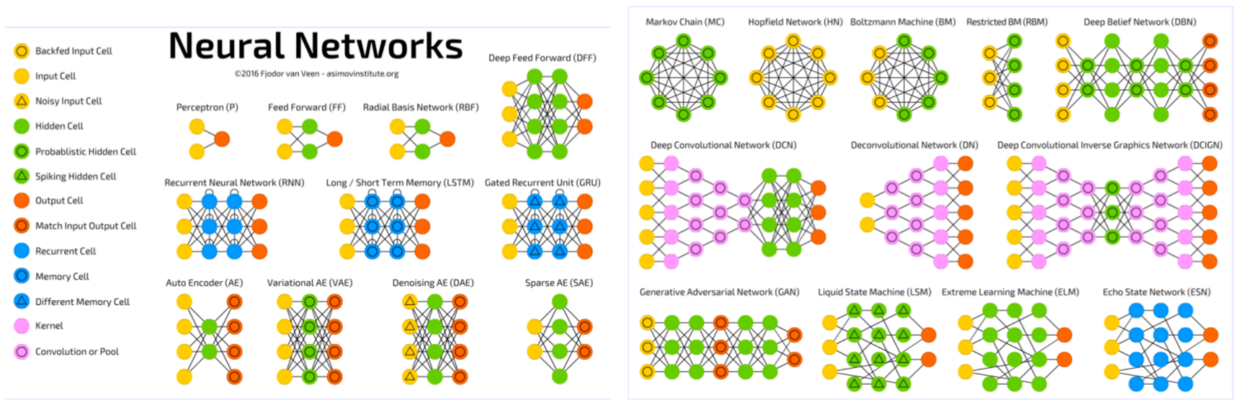
Figure 2: Examples of activation functions.

Neural Network architectures - The multilayer perceptron (MLP)

The most successful model that use the parametric forms for the basis functions in which the parameter values are adapted during training, is the **feed-forward neural network**, also known as the multilayer perceptron MLP.

The multilayer perceptron is constructed by adding one or more *hidden layers* of nodes. The MLP can have an arbitrary number of hidden layers, but most common are one or possible two hidden layers of nodes. The activation functions for the hidden nodes are usually non-linear. In fact there is no meaning in having linear activation functions in many hidden layers. The common activation functions are shown in Figure **??**.

A chart containing many of neural network architectures, composed by Asimov Institute, is illustrated in Figure **??**. Composing a complete list is practically impossible, as new architectures are invented all the time.



(a) (b)

Figure 3: The Neural Network Zoo - Asimov Institute

Convolutional neural networks, CNN

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery.

CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually refer to **fully connected networks**, that is, each neuron in one layer is connected to all neurons in the next layer. *The ”fully-connectedness” of these networks make them prone to overfitting data.* Typical ways of regularization includes adding some form of magnitude measurement of weights to the loss function. However, CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.

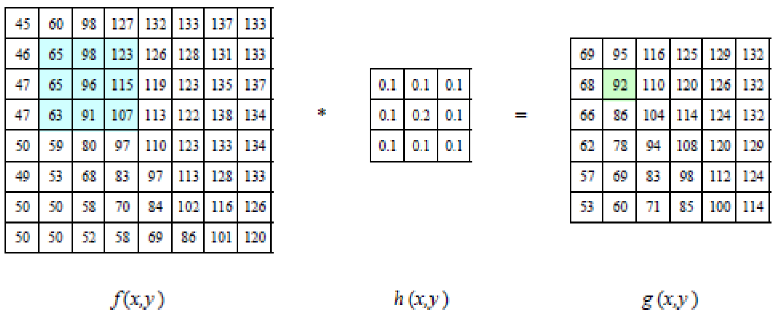
Convolutional Neural Networks (CNNs) is the most popular neural network model being used for image classification problem. The big idea behind CNNs is that a local understanding of an image is good enough. The practical benefit is that having fewer parameters greatly improves the time it takes to learn as well as reduces the amount of data required to train the model. Instead of a fully connected network of weights from each pixel, a CNN has just enough weights to look at a small patch of the image. It’s like reading a book by using a magnifying glass; eventually, you read the whole page, but you look at only a small patch of the page at any given time.

Filtering

To understand how the CNN architecture works we look at the concept of image filters. Assume the our image is given by $f(i,j)$, for simplicity we can assume that each pixel is a scalar. A “filtered” image is mathematically described by the process of **convolution**. The filter is represented by a **kernel**, $h(n,m)$ and the convolved (filtered) image $g(i,j)$ is given by,

$$g = f * h \implies g(i,j) = \sum_u \sum_v f(i-u, j-v)h(u,v) \tag{2}$$

Actually, the process of applying the filter is usually referred to as convolution. This can be seen clearly in Figure **??** using an example.



45	60	98	127	132	133	137	133
46	65	98	123	126	128	131	133
47	65	96	115	119	123	135	137
47	63	91	107	113	122	138	134
50	59	80	97	110	123	133	134
49	53	68	83	97	113	128	133
50	50	58	70	84	102	116	126
50	50	52	58	69	86	101	120

$f(x,y)$

*

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

$h(x,y)$

=

69	95	116	125	129	132
68	92	110	120	126	132
66	86	104	114	124	132
62	78	94	108	120	129
57	69	83	98	112	124
53	60	71	85	100	114

$g(x,y)$

Figure 4: Convolution: The image on the left is convolved with the filter in the middle to yield the image on the right.

Here is a small numerical example where a 4×4 image matrix is filtered by a 2×2 kernel.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \star \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} (-1+2+0+6) & (-2+3+0+7) & (-3+4+0+8) \\ (-5+6+0+10) & (-6+7+0+11) & (-7+8+0+12) \\ (-9+10+0+14) & (-10+11+0+15) & (-11+12+0+16) \end{bmatrix} = \begin{bmatrix} 7 & 8 & 9 \\ 11 & 12 & 13 \\ 15 & 16 & 17 \end{bmatrix}$$

We notice that the filtered image is smaller than the starting image. For CNNs there is also a choice of moving the filter more than one pixel when convolving the image. The amount of pixels the filter is moved is called **stride**. How can we make sure that the filtered image has the same dimension as the original image. The solution is called padding where **zero-padding** being the most common strategy. *Zero-padding consists of adding a boundary of zeros to the original image so that the resulting filtered images has the desired dimensions*, as shown in Figure **??**.

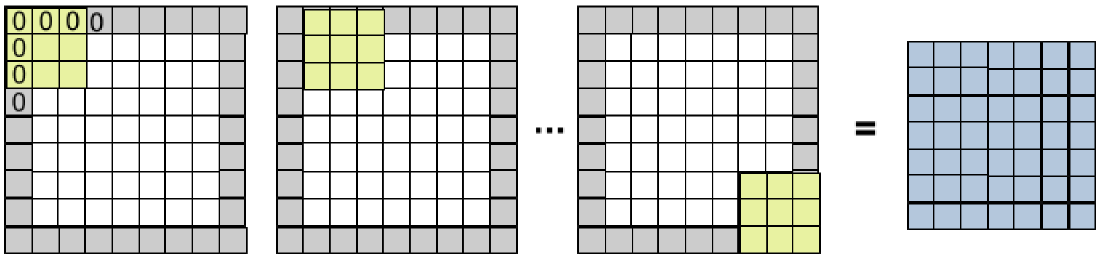
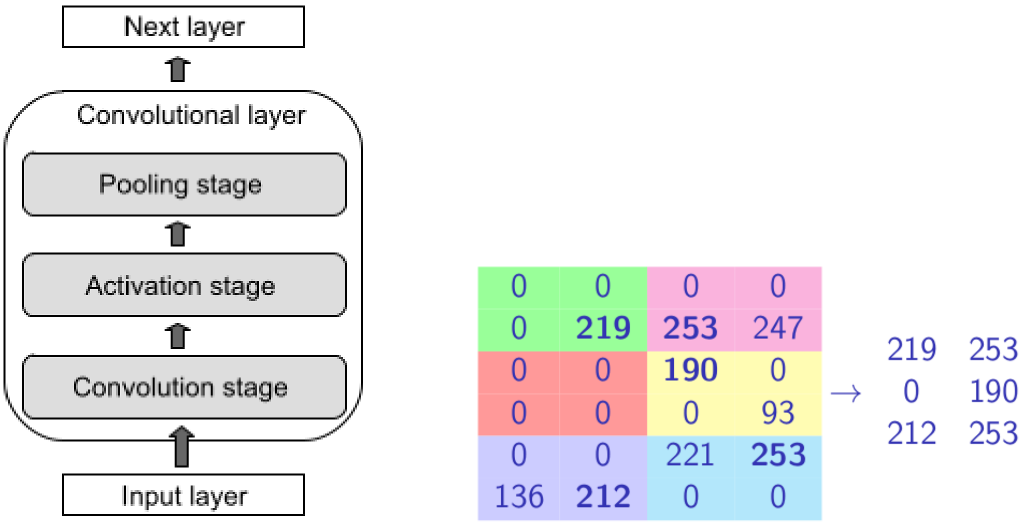


Figure 5: Zero-padding the images with a boundary of zeros will make sure that the filtered image has the same size as the original image.

CNN building blocks

Following the terminology used in the Deep Learning Book, a convolutional layer consists of three blocks, as shown in Figure **??**, the convolution stage, the activation stage and the pooling stage. The first two are vital, but the pooling stage can many times be omitted.

The **activation** stage typically consists of using the rectified linear (ReLU) activation for the hidden nodes. The **pooling** stage consists of replacing the hidden layer with a new layer that can be said to summarize the neighborhood of a given hidden node. A very common pooling operation is **max pooling**. *Max pooling consists of replacing the hidden node value by the maximum value of a small neighborhood around the hidden node.* The term **Subsampling** is generally used and carried out using the max-pooling operation.



(a) (b)

Figure 6: a) The three building blocks of a convolutional layer in a CNN. The convolutional stage, the activation stage and the pooling stage. b) An example of maxpooling by using a tile of size (2, 2), we have reduced the image size from (6, 4) to (3, 2).

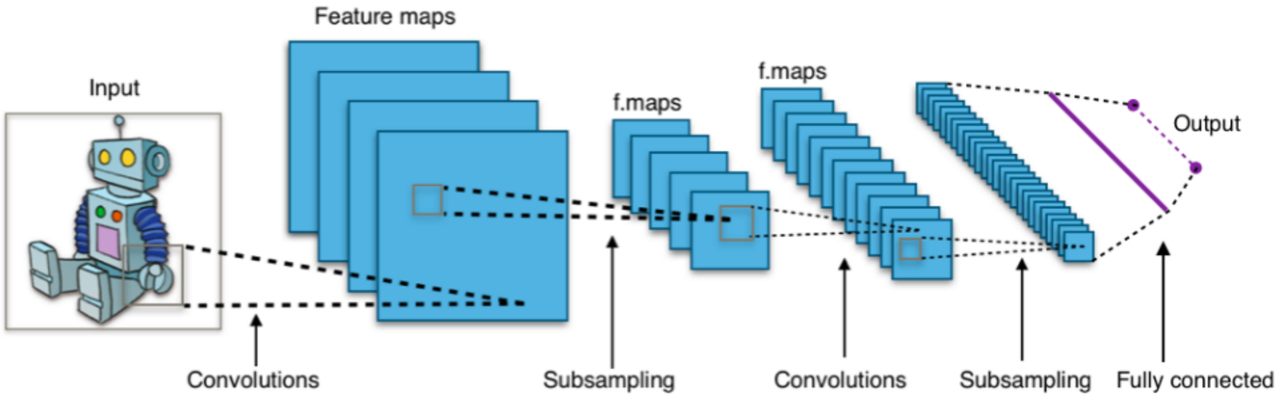


*Figure 7: A CNN architecture. The term **subsampling** used in the figure is the same thing as the pooling layer. In the figure, the network will learn three kernels in the first layer; in the second layer, it will subsample the images, keep one pixel every four pixels, for instance.*

Backpropagation: Learning algorithm

The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams.

Backpropagation is often used to as a name for training an neural networks. We will however see below that back-propagation is just the way we compute gradients in an MLP. Given an input $\mathbf{x}_1$ we first do the forward pass and compute the layer activations x_2, x_3 and the loss L. The loss depends on the ground truth y. When we go in the other direction the goal is to obtain the gradients with respect to the parameters θ_2 and θ_3. First we compute the gradients with respect to the layer activations $\frac{\partial L}{\partial \mathbf{x}_3}$ and $\frac{\partial L}{\partial \mathbf{x}_2}$. Finally the gradients with respect to the parameters $\frac{\partial L}{\partial \theta_3}$ and $\frac{\partial L}{\partial \theta_2}$ can be obtained from $\frac{\partial L}{\partial \mathbf{x}_3}$ and $\frac{\partial L}{\partial \mathbf{x}_2}$ respectively.

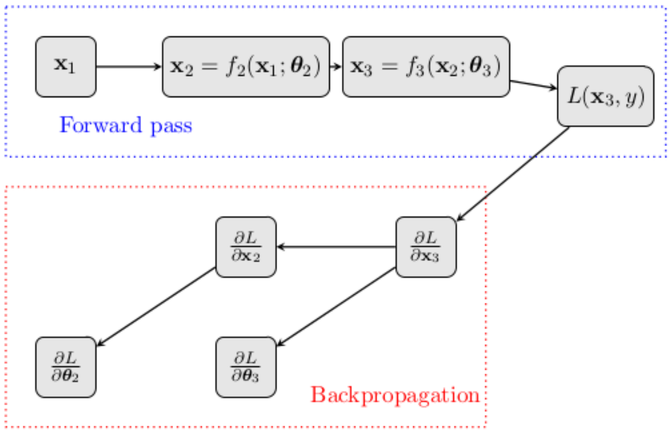

$\mathbf{x}_1$
$\mathbf{x}_2 = f_2(\mathbf{x}_1; \boldsymbol{\theta}_2)$
$\mathbf{x}_3 = f_3(\mathbf{x}_2; \boldsymbol{\theta}_3)$
$L(\mathbf{x}_3, y)$
Forward pass
$\frac{\partial L}{\partial \mathbf{x}_2}$
$\frac{\partial L}{\partial \mathbf{x}_3}$
$\frac{\partial L}{\partial \boldsymbol{\theta}_2}$
$\frac{\partial L}{\partial \boldsymbol{\theta}_3}$
Backpropagation

Figure 8: Illustration of the computational graph for a simple network with two layers and a loss.

4 Common Layers and Backpropagation

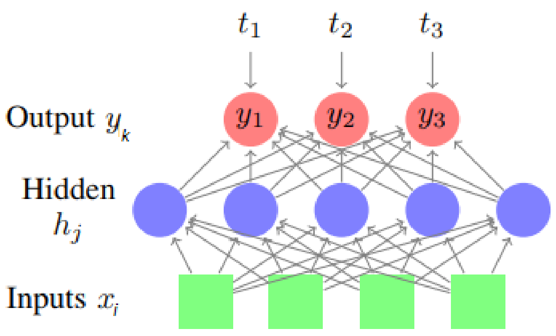


Figure 9: Fully connected layer where we can see the subscript k denotes the output layer , j denotes the hidden layer and i denotes the input layer..

In this section we need tol implement forward and backward steps for a few common neural network layers. When we train a neural net you have to evaluate the gradient $\frac{\partial L}{\partial \theta}$ with respect to all the parameters in the network. The algorithm to do this is called **backpropagation** and it is illustrated in Figure **??**.

4.1 Fully Connected Layer

In this task, we consider a vector $\mathbf{x} \in \mathbb{R}^n$ and we define a mapping to $\mathbf{y} \in \mathbb{R}^m$

$$y_i = \sum_{j=1}^{n} A_{ij} x_j + b_i \tag{3}$$

Using matrix notation we have

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}.$$

where $\mathbf{A}$ and $\mathbf{b}$ are parameters that are trainable. Comparing with figure 1, $\mathbf{x}$ might correspond to $\mathbf{x}_1$, $\mathbf{y}$ to $\mathbf{x}_2$ and the parameter $\boldsymbol{\theta}_2 = \{\mathbf{A}, \mathbf{b}\}$. Thus several such mappings, or layers, are concatenated to form a **full network** such as the one shown in Figure **??**. To derive the *backpropagation equations* we want to express $\frac{\partial L}{\partial \mathbf{x}}$ as an expression containing $\frac{\partial L}{\partial \mathbf{y}}$. Using the **chain rule** we get

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{4}$$

$$\frac{\partial L}{\partial A_{ij}} = \sum_{k=1}^{m} \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial A_{ij}} \tag{5}$$

$$\frac{\partial L}{\partial b_i} = \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial b_i} \tag{6}$$

Exercise 1 - Finding simplified matrix expressions for the gradients

In this task, we need to give the simplified expressions for $\frac{\partial L}{\partial \mathbf{x}}$, $\frac{\partial L}{\partial \mathbf{A}}$ and $\frac{\partial L}{\partial \mathbf{b}}$, where the answers should all be given as matrix expressions without any explicit sums. Here we should remember that in the backward pass we are already given $\frac{\partial L}{\partial \mathbf{y}}$.

From Equation **??**, we write the following sequence of equations

$$\begin{aligned}
y_1 &= A_{11}x_1 + A_{12}x_2 + A_{13}x_3 + \cdots + A_{1m}x_m + b_1 \\
y_1 &= A_{21}x_1 + A_{22}x_2 + A_{23}x_3 + \cdots + A_{2m}x_m + b_1 \\
y_1 &= A_{31}x_1 + A_{32}x_2 + A_{33}x_3 + \cdots + A_{1m}x_m + b_3 \\
&\vdots \\
y_m &= A_{m1}x_1 + A_{m2}x_2 + A_{m3}x_3 + \cdots + A_{nm}x_m + b_m
\end{aligned} \tag{7}$$

Then, with these equations and the equations in **??**, **??**, **??**, we can express the gradient of the loss L with respect to $\mathbf{x}$ and the parameters $\mathbf{A}$ and $\mathbf{b}$ using the chain rule.

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{x}} &= \begin{pmatrix} \frac{\partial L}{\partial x_1} \\ \frac{\partial L}{\partial x_2} \\ \vdots \\ \frac{\partial L}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_1} \\ \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_2} \\ \vdots \\ \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_m} \end{pmatrix} = \{\frac{\partial y_j}{\partial x_i} = A_{ji}\} = \begin{pmatrix} \frac{\partial L}{\partial y_1}A_{11} + \frac{\partial L}{\partial y_2}A_{21} + \frac{\partial L}{\partial y_3}A_{31} + \cdots + \frac{\partial L}{\partial y_m}A_{m1} \\ \frac{\partial L}{\partial y_1}A_{12} + \frac{\partial L}{\partial y_2}A_{22} + \frac{\partial L}{\partial y_3}A_{32} + \cdots + \frac{\partial L}{\partial y_m}A_{m2} \\ \vdots \\ \frac{\partial L}{\partial y_1}A_{1m} + \frac{\partial L}{\partial y_2}A_{2m} + \frac{\partial L}{\partial y_3}A_{3m} + \cdots + \frac{\partial L}{\partial y_m}A_{nm} \end{pmatrix} \\
&= \begin{pmatrix} A_{11} & A_{21} & A_{31} & \cdots & A_{m1} \\ A_{12} & A_{22} & A_{32} & \cdots & A_{m2} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ A_{1m} & A_{2m} & A_{3m} & \cdots & A_{nm} \end{pmatrix} \begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{pmatrix} = \mathbf{A}^T \frac{\partial L}{\partial \mathbf{y}}
\end{aligned} \tag{8}$$

$$\frac{\partial L}{\partial \mathbf{b}} = \begin{pmatrix} \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial b_2} \\ \vdots \\ \frac{\partial L}{\partial b_m} \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial b_1} \\ \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial b_2} \\ \vdots \\ \sum_{j=1}^{m} \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial b_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial L}{\partial y_1} \\ \frac{\partial L}{\partial y_2} \\ \vdots \\ \frac{\partial L}{\partial y_m} \end{pmatrix} = \frac{\partial L}{\partial \mathbf{y}} \tag{9}$$

$$\frac{\partial L}{\partial \mathbf{A}} = \begin{pmatrix} \frac{\partial L}{\partial A_{11}} & \frac{\partial L}{\partial A_{12}} & \cdots & \frac{\partial L}{\partial A_{1m}} \\ \frac{\partial L}{\partial A_{21}} & \frac{\partial L}{\partial A_{22}} & \cdots & \frac{\partial L}{\partial A_{2m}} \\ \vdots & \vdots & & \\ \frac{\partial L}{\partial A_{m1}} & \frac{\partial L}{\partial A_{m2}} & \cdots & \frac{\partial L}{\partial A_{mm}} \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{11}} & \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{12}} & \cdots & \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{1m}} \\ \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{21}} & \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{22}} & \cdots & \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{2m}} \\ \vdots & \vdots & & \\ \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{m1}} & \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{m2}} & \cdots & \sum_{k=1}^{m} \frac{\partial L}{\partial y_k}\frac{\partial y_k}{\partial A_{mm}} \end{pmatrix} =$$

$$= \begin{pmatrix} \frac{\partial L}{\partial y_1}x_1 & \frac{\partial L}{\partial y_1}x_2 & \cdots & \frac{\partial L}{\partial y_1}x_m \\ \frac{\partial L}{\partial y_2}x_1 & \frac{\partial L}{\partial y_2}x_2 & \cdots & \frac{\partial L}{\partial y_2}x_m \\ \vdots & \vdots & & \\ \frac{\partial L}{\partial y_m}x_1 & \frac{\partial L}{\partial y_m}x_2 & \cdots & \frac{\partial L}{\partial y_m}x_m \end{pmatrix} = \frac{\partial L}{\partial \mathbf{y}}\mathbf{x}^T \tag{10}$$

Gradients - Finite differences

When using gradients in the implementation, *it is very important to check that the gradients are correct.* A bad way to conclude that the gradient is correct is to manually look at the code and convince yourself that it is correct or just to see if the function seems to decrease when you run optimization; it might still be a descent direction even if it is not the gradient. A much better way is to check **finite differences** using the formula

$$\frac{\partial f}{\partial x_i} \approx \frac{f(\mathbf{x}+\epsilon\mathbf{e}_i) - f(\mathbf{x}-\epsilon\mathbf{e}_i)}{2\epsilon} \tag{11}$$

where $\mathbf{e}_i$ is a vector that is all zero except for position i where it is 1, and ϵ is a small number.

Training neural network using batch

When training a neural network, *it is common to evaluate the network and compute gradients not with respect to just one element but N elements in a **batch**.* We use superscripts to denote the elements in the batch. For the fully connected layer above, we have multiple inputs $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \cdots, \mathbf{x}^{(N)}$ and we wish to compute $\mathbf{y}^{(1)} = \mathbf{A}\mathbf{x}^{(1)} + \mathbf{b}, \mathbf{y}^{(2)} = \mathbf{A}\mathbf{x}^{(2)} + \mathbf{b}, \cdots, \mathbf{y}^{(N)} = \mathbf{A}\mathbf{x}^{(N)} + \mathbf{b}$.

Thus, in the code for the **forward pass** you can see that the input array first is reshaped to a matrix $\mathbf{X}$ where each column contains all values for a single batch, that is,

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(N)} \end{pmatrix} \tag{12}$$

For instance, if $\mathbf{x}^{(1)}$ an image of size $5 \times 5 \times 3$ it is reshaped to a long vector with length 75. We wish to compute

$$\mathbf{Y} = \begin{pmatrix} \mathbf{y}^{(1)} & \mathbf{y}^{(2)} & \cdots & \mathbf{y}^{(N)} \end{pmatrix} = \begin{pmatrix} \mathbf{A}\mathbf{x}^{(1)} + \mathbf{b} & \mathbf{A}\mathbf{x}^{(2)} + \mathbf{b} & \cdots & \mathbf{A}\mathbf{x}^{(N)} + \mathbf{b} \end{pmatrix} \tag{13}$$

Vectorizing back-propagation

*Vectorization is the process of rewriting a loop so that instead of processing a single element of an array N times, it processes **several or all elements** of the array **simultaneously**.* Using for-loops on the dataset, is too slow, and does not take advantage of modern **parallelism** in CPU and GPU. In order to achieve high performance, we need to transform the dataset into a **matrix representation**. If we take the column-based representation, every input from our dataset is copied to a column in the matrix.

When we are backpropagating to $\mathbf{X}$ we have to compute

$$\frac{\partial L}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial L}{\partial \mathbf{x}^{(1)}} & \frac{\partial L}{\partial \mathbf{x}^{(2)}} & \cdots & \frac{\partial L}{\partial \mathbf{x}^{(N)}} \end{pmatrix} \tag{14}$$

using the given partial derivative of the loss L in respect to $\mathbf{Y}$

$$\frac{\partial L}{\partial \mathbf{Y}} = \begin{pmatrix} \frac{\partial L}{\partial \mathbf{y}^{(1)}} & \frac{\partial L}{\partial \mathbf{y}^{(2)}} & \cdots & \frac{\partial L}{\partial \mathbf{y}^{(N)}} \end{pmatrix} \tag{15}$$

Here we need to use the expression obtained in exercise 1 and simplify to **matrix operations**.

For the parameters, we have that both $\mathbf{A}$ and $\mathbf{b}$ influence all elements $\mathbf{y}^{(i)}$, so for the parameters we compute $\frac{\partial L}{\partial \mathbf{A}}$ using the chain rule with respect to each element in each $\mathbf{y}^{(i)}$ and just sum them up. More formally,

$$\frac{\partial L}{\partial A_{ij}} = \sum_{l=1}^{N} \sum_{k=1}^{m} \frac{\partial L}{\partial y_k^{(l)}} \frac{\partial y_k^{(l)}}{\partial A_{ij}} \tag{16}$$

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{N} \sum_{j=1}^{m} \frac{\partial L}{\partial y_j^{(l)}} \frac{\partial y_j^{(l)}}{\partial b_i} \tag{17}$$

We note that the inner sum is the same as that we have computed in the previous exercise, so we just sum the expression obtained in the previous exercise over all elements in the batch.

Exercise 2 - Vectorizing the backpropagation equations

In this task, we need to vectorize $\frac{\partial L}{\partial \mathbf{x}}$, $\frac{\partial L}{\partial \mathbf{A}}$ and $\frac{\partial L}{\partial \mathbf{b}}$, where the answers should all be proved matimatically. Here we should remember that in the backward pass we are already given $\frac{\partial L}{\partial \mathbf{y}}$. It is possible to implement the forward and backward passes with **for-loops**, but in this assignment we should use **matrix operations** to vectorize the code. Then, we need to implement the functions `fully_connected_forward` and `fully_connected_backward`, and check that the implementation is correct by running `test_fully_connected`. There are potentially useful functions in Matlab lkie `bsxfun`, `sub2ind`, `ind2sub`, `reshape` and `repmat`.

Using the result obtained before in **??** and Equation**??**, we can write

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{X}} &= \begin{pmatrix} \frac{\partial L}{\partial \mathbf{x}^{(1)}} & \frac{\partial L}{\partial \mathbf{x}^{(2)}} & \cdots & \frac{\partial L}{\partial \mathbf{x}^{(N)}} \end{pmatrix} = \begin{pmatrix} A^T \frac{\partial L}{\partial \mathbf{y}^{(1)}} & A^T \frac{\partial L}{\partial \mathbf{y}^{(2)}} & \cdots & A^T \frac{\partial L}{\partial \mathbf{y}^{(N)}} \end{pmatrix} \\ &= A^T \begin{pmatrix} \frac{\partial L}{\partial \mathbf{y}^{(1)}} & \frac{\partial L}{\partial \mathbf{y}^{(2)}} & \cdots & \frac{\partial L}{\partial \mathbf{y}^{(N)}} \end{pmatrix} = A^T \frac{\partial L}{\partial \mathbf{Y}} \end{aligned} \tag{18}$$

By plugging the result obtained before in **??** in Equation**??**, we can write

$$\frac{\partial L}{\partial b_i} = \sum_{l=1}^{N} \sum_{j=1}^{m} \frac{\partial L}{\partial y_j^{(l)}} \frac{\partial y_j^{(l)}}{\partial b_i} = \sum_{l=1}^{N} \frac{\partial L}{\partial y^{(l)}} = \frac{\partial L}{\partial \mathbf{Y}} \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = sum\left(\frac{\partial L}{\partial \mathbf{Y}}, 2\right) \tag{19}$$

Finally, by plugging the result obtained before in **??** in Equation**??**, we can write

$$\frac{\partial L}{\partial A_{ij}} = \sum_{l=1}^{N} \sum_{k=1}^{m} \frac{\partial L}{\partial y_k^{(l)}} \frac{\partial y_k^{(l)}}{\partial A_{ij}} = \sum_{l=1}^{N} \frac{\partial L}{\partial y^{(l)}} (x^{(l)})^T = \frac{\partial L}{\partial \mathbf{Y}} \mathbf{X}^T \tag{20}$$

In the following listing, we can see the relevant code to the implemented and tested functions `fully_connected_forward` and `fully_connected_backward`.

Listing 1: The implementation of fully connected forward).

```
%error('Implement this!');
    %b = reshape(b, [features, batch]);
    Y = A * X + b;
```

Listing 2: The implementation of fully connected backward).

```
%error('Implement this!');
    %dldX = zeros(sz);
    dldX = A' * dldY;
    dldX = reshape(dldX, sz);

    %szA = size(A);
    %dldA = zeros(szA);
    dldA = dldY * X';

    %szb = size(b);
    %dldb = zeros(szb);
    dldb = sum(dldY,2)
```

4.2 ReLU - Rectified linear unit

The most commonly used function as a nonlinearity is the rectified linear unit (relu). It is defined as

$$y_i = \max(x_i, 0) \tag{21}$$

In other words, the activation is simply thresholded at zero, as illustrated in Figure **??**. *It is one of the most popular activation functions for deep neural networks, especially for convolutional neural networks.* A smooth approximation to the ReLU is given by the so called **softplus** function,

$$g(x) = \ln(1 + e^x)$$

where the derivative is given by the **logistic function**,

$$\frac{\partial}{\partial x} g(x) = \frac{e^x}{(e^x + 1)} = \frac{1}{(1 + e^{-x})}$$

Both ReLU and softplus are shown in figure 6.1.

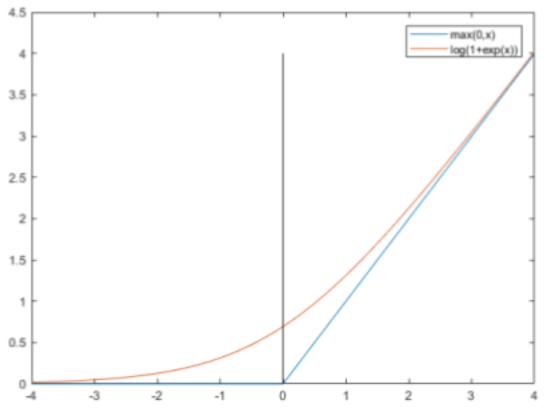


Figure 10: Illustration of The ReLU activation function and a smooth version called softplus.

Why has ReLU become so popular? There are some definite advantages of using this activation function, such as,

- Sparse activation. Suppose we have normalized inputs and uniform initialization of the weights. For a given input pattern about 50% of the nodes will have a zero output and the rest will be linear.
- No vanishing gradients.
- Fast computation.
- More biological plausible.

Exercise 3 - Backpropagation expression using Relu

In this task, we need to derive the backpropagation expression for $\frac{\partial L}{\partial x_i}$ and find a full solution. Then, we implement the layer in `relu_forward` and `relu_backward` and test them with `test_relu`.

$$\frac{\partial y_j}{\partial x_i} = \begin{cases} 0 & \text{when} \quad i \neq j \\ 1 & \text{when} \quad i = j \end{cases} \tag{22}$$

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial x_i} = \begin{cases} 0 & \text{when} \quad i \neq j \\ \\ \frac{\partial L}{\partial y_j} & \text{when} \quad i = j \end{cases} \tag{23}$$

In the following listing, we can see the relevant code to the implemented and tested functions `relu_forward` and `relu_backward`.

Listing 3: The implementation of relu forward).

```
%error('Implement this!');
    y = max(x,0);
```

Listing 4: The implementation of relu backward).

```
%error('Implement this!');
    x(x>0) = 1;
    x(x<=0) = 0;
    dldx = x .* dldy;
```

4.3 Softmax Loss

In this task, we need to derive the backpropagation expression for $\frac{\partial L}{\partial x_i}$ and find a full solution. Then, we implement the layer in `softmaxloss_forward` and `softmax_backward` and test them with `test_softmaxloss`.

Softmax Activation Function

The softmax activation function is often placed at the output layer of a neural network. It's commonly used in **multi-class** learning problems where a set of features can be related to one-of-K classes. For example, in the CIFAR-10 image classification problem, given a set of pixels as input, we need to classify if a particular sample belongs to one-of-ten available classes: i.e., cat, dog, airplane, etc. In particular, a common use of **softmax** appears in logistic regression: the softmax "layer", wherein we apply softmax to the output of a fully-connected layer.

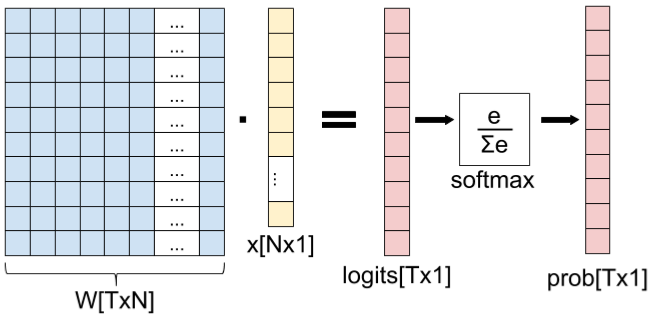


Figure 11: Illustration of the softmax layer. In this diagram, we have an input x with N features, and T possible output classes. The weight matrix W is used to transform x into a vector with T elements (called "logits" in ML folklore), and the softmax function is used to "collapse" the logits (scores) into a vector of probabilities denoting the probability of x belonging to each one of the T output classes.

Suppose we have a vector $x = [x_1, \cdots, x_n]^T \in \mathbb{R}^n$. The goal is to classify the input as one out of n classes and x_i is a **score** for class i and a larger x_i is a higher score. By using the softmax function we can interpret the scores x_i as probabilities. Let

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{24}$$

In this way, we just have to compute for the normalized exponential function of all the units in the layer. Intuitively, what the softmax does is that it squashes a vector of size K between 0 and 1. Furthermore, because it is a normalization of the exponential, the sum of this whole vector equates to 1. We can then interpret the output of the softmax as the probabilities that a certain set of features belongs to a certain class. Thus, given a three-class example below in Figure ??, the scores y_i are computed from the forward propagation of the network. We then take the softmax and obtain the probabilities.

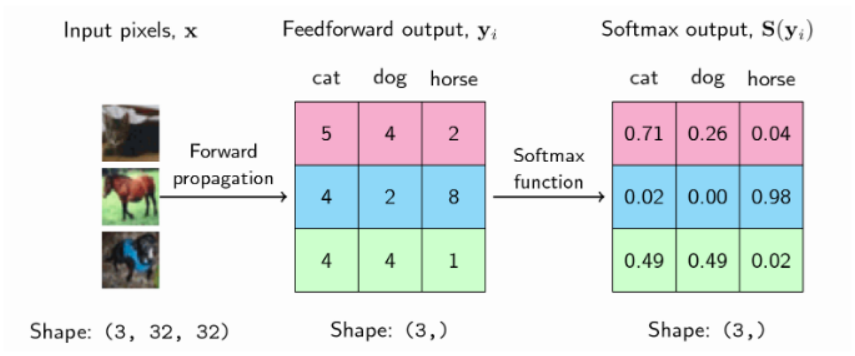


Figure 12: Example of Softmax Computation for three classes: for this example, the neural network assigns a confidence of 0.71 that it is a cat, 0.26 that it is a dog, and 0.04 that it is a horse. The same goes for each of the samples above.

Negative Log-Likelihood (NLL)

Since now y_i are probabilities we can define the loss L to be minimized as the negative log likelihood,

$$L(\mathbf{x}, c) = -\log(y_c) = -\log\left(\frac{e^{x_c}}{\sum_{j=1}^{n} e^{x_j}}\right) = -x_c + \log\left(\sum_{j=1}^{n} e^{x_j}\right) \tag{25}$$

When computing this loss, we can then see that higher confidence at the correct class leads to lower loss and vice-versa.

Exercise 4 - Backpropagation using Softmax

In this task, we need to derive the backpropagation expression for $\frac{\partial L}{\partial x_i}$ and find a full solution. Then, we implement the layer in `softmaxloss_forward` and `softmax_backward` and test them with `test_softmaxloss`.

By recalling the quotient rule for derivatives, we can write

$$y_j = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} \implies \frac{\partial y_j}{\partial x_i} = \frac{e^{x_j}\sum_{i=1}^{n} e^{x_i} - e^{x_j}e^{x_j}}{(\sum_{i=1}^{n} e^{x_i})^2} = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} - \left(\frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}\right)^2 \quad \textbf{when} \quad i = j \tag{26}$$

otherwise, we have

$$y_j = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} \implies \frac{\partial y_j}{\partial x_i} = -\frac{e^{x_i}e^{x_j}}{(\sum_{i=1}^{n} e^{x_i})^2} \quad \textbf{when} \quad i \neq j \tag{27}$$

In addition, from ??, we can write

$$\frac{\partial L}{\partial y_j} = \frac{\partial}{\partial y_j}(-\log(y_j)) = \frac{-1}{y_j} \tag{28}$$

Then we can find the result by using

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{29}$$

But it is actually also possible to solve the problem directly by differentiating the equation in ??, i.e.

$$\frac{\partial L}{\partial x_i} = \frac{\partial}{\partial x_i}\left(-x_c + \log\left(\sum_{j=1}^{n} e^{x_j}\right)\right) = \begin{cases} y_i - 1 & \text{when} \quad c = i \\ \\ y_i & \text{when} \quad c \neq i \end{cases} \tag{30}$$

where

$$\frac{\partial}{\partial x_i}\log\left(\sum_{j=1}^{n} e^{x_j}\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} = y_i \tag{31}$$

In the following listing, we can see the relevant code to the implemented and tested functions `softmax_forward` and `softmax_backward`.

Listing 5: The implementation of softmax forward).

```
        %error('Implement this!');
    xc = x(sub2ind(sz, labels', 1:batch));
    L = -xc + log(sum(exp(x)));
    L = mean(L);
    %L = mean(-x + log(sum(exp(x))));
%     Lsum = sum(-labels + log(sum(exp(x))));
%     L = mean(Lsum);
```

Listing 6: The implementation of softmax backward).

```
 %error('Implement this!');
    %y = exp(x) ./ sum(exp(x));
    sumExpx = sum(exp(x), 1);
    %sumExpx = bsxfun(@plus, exp(x), 1:batch )  % XXXXX
    %y = exp(x) ./ sumExpx; % here can use bsxfun(@rdivide, exp(x), sumExpx
        )
    y = bsxfun(@rdivide, exp(x), sumExpx);
    y(sub2ind(sz, labels', 1:batch)) = y(sub2ind(sz, labels', 1:batch)) -
        1;
    dldx = y;
    dldx = dldx / batch;
```

5 Training Neural Network

The function we are trying to minimize when we are training a neural net is

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}; \boldsymbol{\theta}) \tag{32}$$

where θ are all the parameters of the network, $\mathbf{x}^{(i)}$ is the input and $y^{(i)}$ the corresponding ground truth and $L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}; \theta)$ is the loss for a single example, given for instance by a neural net with a softmax loss in the last layer, as illustrated in Figure **??**. *In practice, N is very large and we never evaluate the gradient over the entire sum. Instead we evaluate it over a batch with $n << N$ elements because the network can be trained much faster if you use batches and update the parameters in every step.*

Gradient descent enhancements

If we are using **gradient descent** to train the network, the way the parameters are updated is

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \alpha\frac{\partial L}{\partial \boldsymbol{\theta}} \tag{33}$$

where the **learning rate** α is a hyperparamete that controls the size of the update. The gradient descent method is easy to implement and fast, but not always the most effcient one. It can suffer from the problem of **local minima** and regions with **small gradients**, as shown in Figure **??**. Furthermore, it is not always easy to set the value of the learning rate α. Even though the stochastic gradient descent approach helps avoiding local minima, there are still other improvements. Below there is one of them.

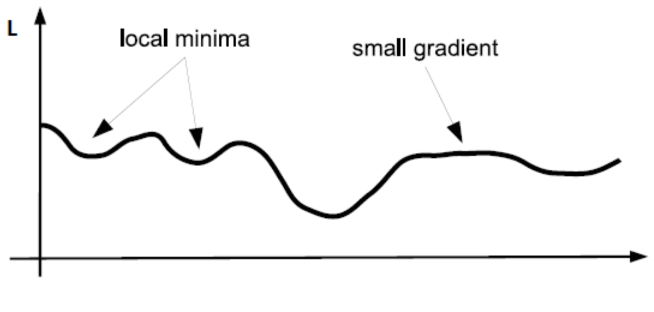


Figure 13: Illustration of the problem of local minima and small gradients.

Stochastic Gradient Descent with Momentum

The stochastic gradient descent algorithm can oscillate along the path of steepest descent towards the optimum. Adding a momentum term to the parameter update is one way to reduce this oscillation . The idea behind gradient descent with momentum is to average the gradient estimations over time and use the smoothed gradient to update the parameters. The stochastic gradient descent with momentum (SGDM) update is

$$\mathbf{m}_n = \mu\ \mathbf{m}_{n-1} + (1-\mu)\frac{\partial L}{\partial \boldsymbol{\theta}} \tag{34}$$

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \alpha \mathbf{m}_n \tag{35}$$

where $\mathbf{m}_n$ is a moving average of the gradient estimations and μ is a hyper-parameter in the range $0 < \mu < 1$ controlling the smoothness.

Exercise 5 - Gradient Descent with Momentum

Listing 7: The implementation of the gradient descent with momentum in the function training().

```
        % momentum and update
    if isfield(opts, 'momentum')
        % We loop over all layers and then all parameters.
        % We use momentum{i}.(s) as the momentum for
        % parameter s in layer number i. Note that theta in
        % the assignment is just a convenient placeholder
        % meaning all parameters in all layers. You can see
        % the code for normal gradient descent below.
        % Remember to include weight decay as param <-
            param
        % - lr*(momentum + weight_decay*param)

        if it==1
            momentum{i}.(s) = zeros(size(net.layers{i}.
                params.(s)));
        end

        %error('Implement this!');
        mu = opts.momentum;
        momentum{i}.(s) = mu*momentum{i}.(s) + (1-mu)*grads
            {i}.(s); % XXXXXX
        net.layers{i}.params.(s) = net.layers{i}.params.(s)
            - ...
            opts.learning_rate * (momentum{i}.(s) + ...
                opts.weight_decay * net.layers{i}.params.(s
                ));
    else
        % run normal gradient descent if
        % the momentum parameter is not specified
        net.layers{i}.params.(s) = net.layers{i}.params.(s)
            - ...
            opts.learning_rate * (grads{i}.(s) + ...
                opts.weight_decay * net.layers{i}.params.(s
                ));
    end
```

6 Classifying Handwritten Digits

In thisl task, we work on the `MNIST` image dataset of handwritten digits, consisting of data-target pairs with images $\mathbf{X}_i \in [0,1]^{28\times 28}$ and targets $t_i \in \{0, 1, \cdots, 9\}$. We split the training set into training data and validation data such that we use 2000 of the 60000 training images as validation data, where for both sets the images have been stacked column-wise, i.e., $\mathbf{x}_i = vec(\mathbf{X}_i) \in \mathbb{R}^{28\times 28}$.

Precision and Recall

There are different kinds of measures to evaluate the performance of machine learning techniques, for instance:

- Precision and recall in information retrieval and natural language processing;
- The receiver operating characteristic (ROC) in medicine.

For a given class, the **recall** is defined as the true positives divided by the sum of the true positives and false negatives.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{36}$$

For a given class, the **precision** is defined as the true positives divided by the true and false positives.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{37}$$

Exercise 6 - Convolutional network `mnist_starter`

In this exercise, we need to validate an accuracy of 98% on the test set using a provided **convolutional** feed-forward network `mnist_starter` to classify the MNIST dataset. In this way, when we can make sure the code we have written so far is correct, we need to analyse this convnet by showing relevant plots, investigating the confusion matrix and getting the number of parameters for the layers.

In the MNIST dataset the images are greyscale and have size 28x28, so the dimension is $28 \times 28 \times 1$. We should consider that the *batch indexing* will be the last dimension such that, using a batch of 16 examples, the true dimension will be [28, 28, 1, 16].

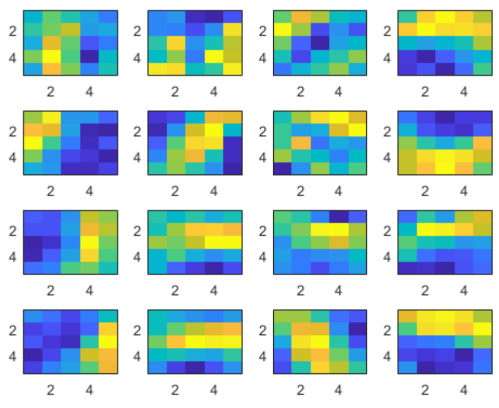
Here we use **filters** of size $5 \times 5 \times 1$. In total there are 16 such filters so the output will have 16 channels. Each filter has a bias, so the size of the bias vector is 16×1. We use 2×2 padding, so the output will have size $(28 - 5 + 1 + 2 \times 2) \times (28 - 5 + 1 + 2 \times 2) = (28 \times 28)$, i.e. it will be the same.

Max pooling, the input size is $28 \times 28 \times 16 \times (batchsize)$. This function uses a stride of 2, so the output will have size $14 \times 14 \times 16 \times (batchsize)$.

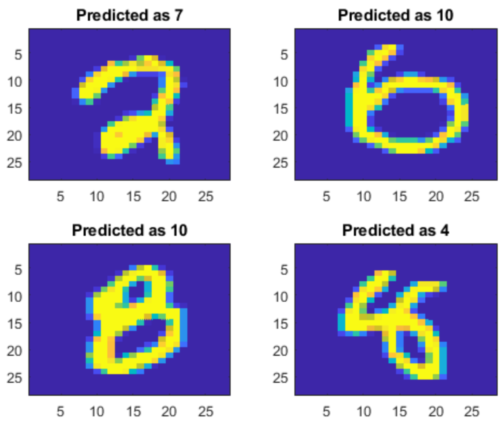
Filters plot for the first convolutional layer

Here we get the plot of the 16 filters of size 5×5 for the first convolutional layer, as illustrated in Figrure **??**, by running the following script in Matalb

```
% Plot the filters the first convolutional layer learns.
convLayer1 = net.layers{2}.params.weights;
for i = 1:size(convLayer1,4)
    subplot(4,4,i)
    imagesc(convLayer1(:,:,1,i))
end
```



(a)



(b)

Figure 14: a) Plot of the filters for the first convolutional layer. b) Plot of few images that are missclassified for MNIST.

6.0.1 Confusion matrix - Recall and Precision

classes	0	1	2	3	4	5	6	7	8	9
0	1114	6	2	0	0	0	2	11	0	0
1	0	1018	4	0	0	0	5	5	0	0
2	0	1	1000	0	3	0	4	2	0	0
3	0	2	0	959	0	1	3	9	8	0
4	0	2	19	0	864	2	1	2	0	2
5	3	1	1	3	1	939	0	3	0	7
6	1	10	3	0	1	0	1009	4	0	0
7	0	1	2	1	0	0	1	965	0	4
8	2	0	6	9	14	0	12	19	941	6
9	0	1	0	0	1	0	1	2	0	975

Table 1: Confusion Matrix: Test data. True label is on the vertical axis and the predicted label on the horizontal axis.

Now we can compute the recall and precision values according to the definitions in **??** and **??**. The results are shown in Table **??**

classes	precision	recall
class1	0.9946	0.9815
class 2	0.9770	0.9864
class 3	0.9643	0.9901
class 4	0.9866	0.9766
class 5	0.9774	0.9686
class 6	0.9968	0.9802
class 7	0.9721	0.9815
class 8	0.9442	0.9908
class 9	0.9916	0.9326
class 10	0.9809	0.9949

Table 2: Precision and Recall for all digits.

The number of parameters

To start with a proposed CNN is provided to solve this classification problem. It consists of the following:

1. Layer 1, The **first layer** is a placeholder for the input we give the network, size (28 28 1 16)

2. Layer 2, **first convolutional layer** consisting of 16 kernels of size 5×5
3. Layer 3, (relu) size (28 28 16 16)
4. Layer 4, (maxpooling) size (14 14 16 16)
5. Layer 5, **second convolutional layer** of 16 kernels of size 5×5
6. Layer 6, (relu) size (14 14 16 16)
7. Layer 7, (maxpooling) size (7 7 16 16)
8. Layer 8, **fully connected** (dense) with 10 nodes
9. Layer 9, (softmaxloss) size (1 1)

Looking at the architecture of the CNN we can compute the parameters as:

First layer:16 kernels of size (5, 5) and an intercept: $(5 \cdot 5 + 1) \cdot 16 = 416$ parameters

The second layer has $(5 \cdot 5 \cdot 16 + 1) \cdot 16 = 6416$ parameters.

The fully connected layer $(784 + 1) \times 10 = 7850$ parameters

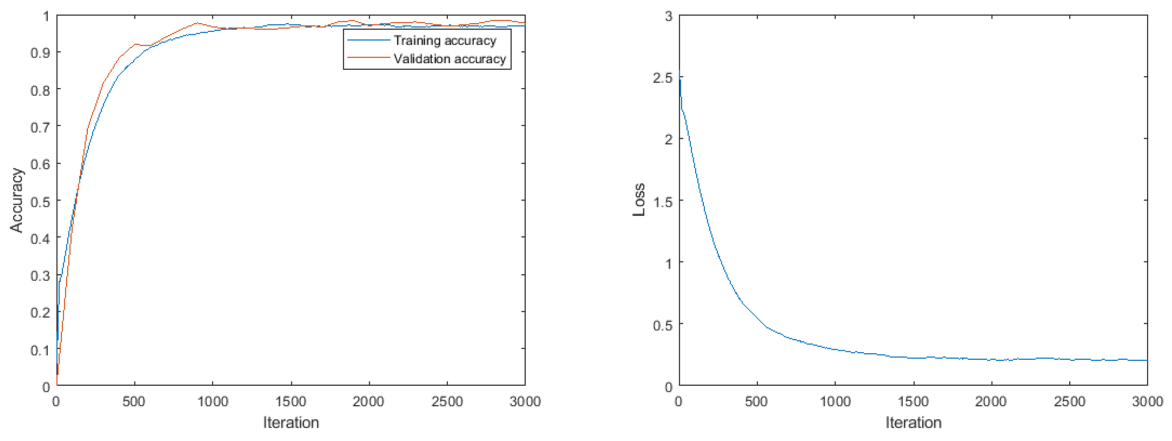
Total parameters =14682

Accuracy results

The obtained test result is

```
Iteration 3000:
Classification loss: 0.084850
Weight decay loss: 0.105513
Total loss: 0.190363
Training accuracy: 0.976196
Validation accuracy: 0.971606

Accuracy on the test set: 0.979300
```



(a) (b)

Figure 15: a) Plots illustrating the accuracy in a) and the loss in b), after training the network `mnist_starter` *on the MNIST dataset .*

7 Classifying Tiny Images - CIFAR-10 Dataset

In this section, we need to train a simple network `cifar10_starter.m` on the CIFAR-10 datatset. This baseline give an accuracy of about 48% after training for 5000 iterations. We should mention that it is much harder to get good classification results on this dataset than MNIST, mainly due to signicant *intraclass variation*. The task here is to improve the accuracy of the baseline network.

The CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 colour images in 10 classes, with 6000 images per class, as illustrated in Figure **??**. There are 50000 training images and 10000 test images. Each image has a 32×32 dimension and represents an object among 10 categories numbered from 0 to 9: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The images have three color channels: red, green, and blue. The input shape is then (32, 32, 3), where the last dimension is to take into account the colors.

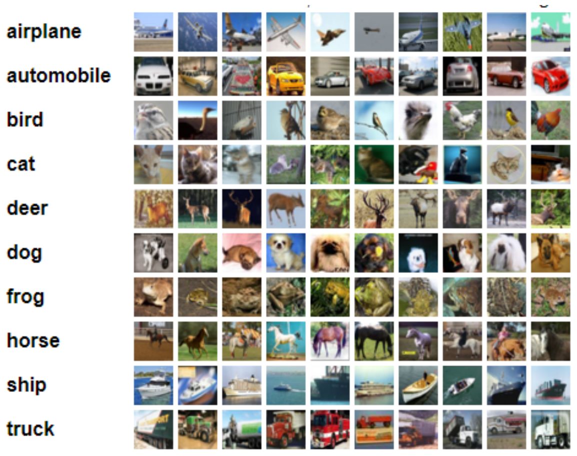
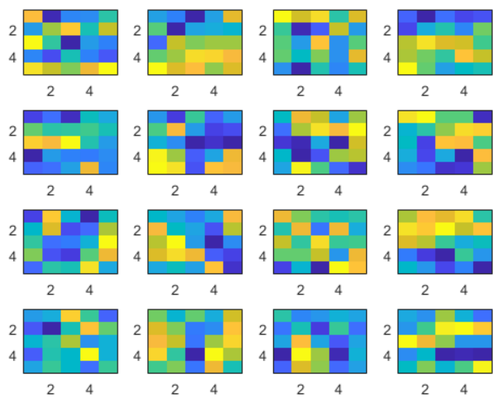
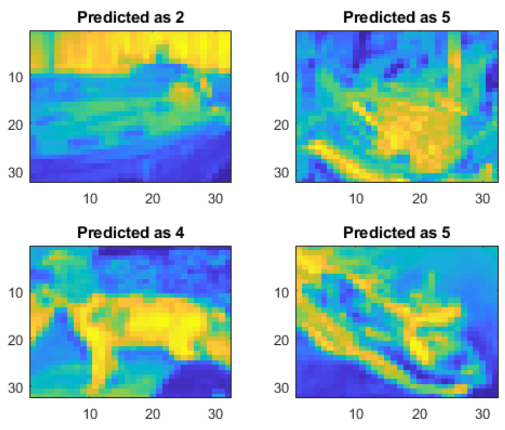


Figure 16: Here are the classes in the dataset CIFAR-10, as well as 10 random images from each

7.1 Exercise 7 - Improving the the accuracy of the convNet `cifar10_starter`



(a)



(b)

Figure 17: a) Plot of the filters for the first convolutional layer. b) Plot of few images that are missclassified for cifar10.

7.1.1 Confusion matrix - Recall and Precision

classes	0	1	2	3	4	5	6	7	8	9
0	619	84	36	25	23	10	11	18	101	73
1	16	782	2	13	10	3	6	7	25	136
2	94	35	387	106	106	51	61	104	18	38
3	34	40	65	409	65	122	92	113	6	54
4	35	33	136	79	408	23	67	183	11	25
5	15	17	95	271	49	330	33	143	13	34
6	13	27	62	88	94	17	613	44	4	38
7	14	17	24	69	72	40	8	707	3	46
8	122	126	7	29	10	8	8	10	582	98
9	32	221	6	21	8	8	11	28	20	645

Table 3: Confusion Matrix: Test data. True label is on the vertical axis and the predicted label on the horizontal axis.

Now we can compute the recall and precision values according to the definitions in **??** and **??**. The results are shown in Table **??**

classes	precision	recall
class1	0.9946	0.9815
class 2	0.9770	0.9864
class 3	0.9643	0.9901
class 4	0.9866	0.9766
class 5	0.9774	0.9686
class 6	0.9968	0.9802
class 7	0.9721	0.9815
class 8	0.9442	0.9908
class 9	0.9916	0.9326
class 10	0.9809	0.9949

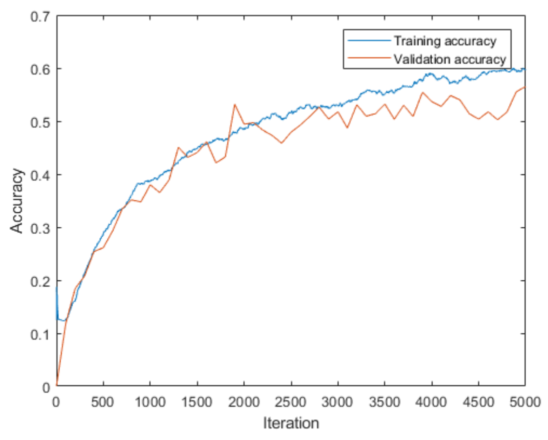
Table 4: Precision and Recall for all digits.

Accuracy results - cifar10

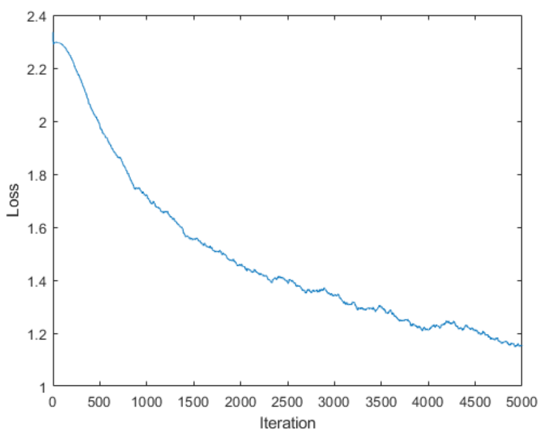
The obtained test result is

```
Iteration 5000:
Classification loss: 1.141692
Weight decay loss: 0.011528
Total loss: 1.153220
Training accuracy: 0.596877
Validation accuracy: 0.565154

Accuracy on the test set: 0.550000
```



(a)



(b)

Figure 18: a) Plots illustrating the accuracy in a) and the loss in b), after training the network `cifar10_starter` *on the cifar10 dataset .*

`https://towardsdatascience.com/the-4-convolutional-neural-network-models-that-can-classify-your-fas`

Train a CNN using the following CNN:

16x(5x5 kernel) - maxpool - 16x(5x5 kernel) - no maxpool - (Flatten)-Dense(10) - Dense(1)

Make sure that your trained model gives good test results (i.e.> 95% accuracy).

To control the complexity it is often better to use a regularization term added to the error function. In this way, we control the complexity by adding an L2 regularizer).We should modify this value until we find the "near optimal" validation performance.

Present an MLP with associated hyperparameters that maximize the validation performance:
The typical goal is to have a high accuracy (i.e., the fraction of correctly classified cases). During training, we typically monitor possible overfitting by looking at the log loss of the validation data, since this is the error used during training. One can, however, have a situation where the log loss increases for the validation data but the accuracy does not decrease (why?). This means that monitoring the accuracy may be better during the model selection.

The idea behind gradient descent with momentum is to average the gradient estimations over time and use the smoothed gradient to update the parameters.

Performance Improvement techniques

- Dataset Augmentation More data is almost always a good way to improve a machine learning system to generalize better. We can sometimes create new data by augmenting what data we have.
- Hyperparameter Tuning Learning rate: If the loss is diverging, decrease the learning rate. If the loss decreases slowly, you should probably increase the learning rate. If the loss has plateaued, decrease the learning rate.

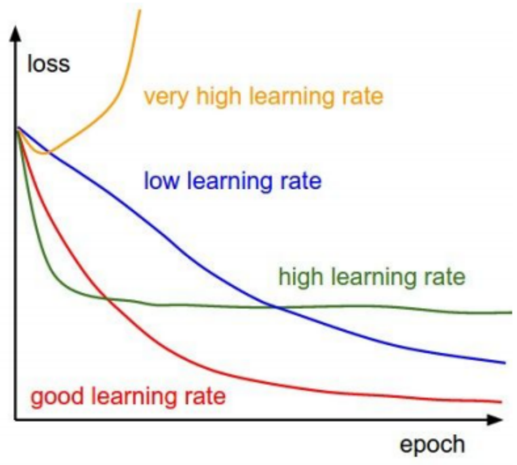


Figure 19: Different learning rate cases.

- Regularization: Is there a big gap between training and validation accuracy? Increase regularization. Is the validation accuracy increasing? Keep training.

References

[1] Bishop, C. M.: Pattern Recognition and Machine Learning. Springer, 2006. Online version available at https://www.microsoft.com/en-us/research/people/cmbishop/#!prml-book (March 2019).

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *Deep Learning.* MIT Press, 2016, ISBN: 9780262035613. HTLM version available at https://www.deeplearningbook.org/ (March 2019).

[3] Introduction to convolutional neural networks (CNN) for classification. Available online at http://www.robots.ox.ac.uk/~vgg/practicals/cnn/

[4] R. Sutton, A.G. Barto: Reinforcement Learning, An Introduction. MIT Press, 2017. Draft available at http://incompleteideas.net/sutton/book/the-book-2nd.html (March 2019).

[5] T. Hastie, R. Tibshirani, J. Friedman: The elements of statistical learning, data mining, inference and prediction, 2nd edition, Springer, 2009, online version available at https://web.stanford.edu/~hastie/Papers/ESLII.pdf (March 2019).

[6] Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/

[7] Andrew Ng, Stanford course CS229: Machine Learning. Available online at cs229.stanford.edu/notes/cs229-notes1.pdf

[8] Roger Grosse, UToronto course CSC 411 Fall 2018: Machine Learning and Data Mining. Available online at http://www.cs.toronto.edu/~rgrosse/courses/csc411_f18/

[9] The Neural Network Zoo, Asimov Institute. Available online at http://www.asimovinstitute.org/neural-network-zoo/

[10] Y. LeCun et al. Efficient BackProp, Neural Networks: Tricks of the Trade, 1998.