Mathematisch-
Naturwissenschaftliche
Fakultät

Human-Computer Interaction

Bachelorarbeit

Evaluation of ML methods for online use in the browser

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Human-Computer Interaction
Hao,Liu, h.liu@uni-tuebingen.de, 2020

Bearbeitungszeitraum: von 01.08.2020 - bis

Betreuer/Gutachter: Prof. Dr. Enkelejda Kasneci
Zweitgutachter: Dr. Wolfgang Fuhl, Universität Tübingen

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Hao,Liu (Matrikelnummer 4187178), October 30, 2020

Abstract

Machine learning continues to be an increasingly integral component of our lives, whether we are applying the techniques to research or business problems. Machine learning models ought to be able to give accurate predictions in order to create real value for a given organization. At the same time Machine learning training by running algorithms on the browser has gradually become a trend. As the closest link to users in the Internet, the web front-end can also create a better experience for our users through AI capabilities. This article will focus on how to evaluate machine learning algorithms and deploy machine learning models in the browser. We will use "Cars", "MNIST" and "Cifar-10" datasets to test LeNet, AlexNet, GoogLeNet and ResNet models in the browser. On the other hand, we will also test emerging lightweight models such as MobileNet. By trying, comparing, and comprehensively evaluating regression and classification tasks, we can summarize some excellent methods/models and experiences suitable for machine learning in the browser.

Acknowledgments

Here I would like to thank Google Cloud and Alibaba Cloud for their data support for this article, and Dr. Wolfgang Fuhl for helping me during the creation of the thesis.

Contents

1 Introduction

The core idea behind machine learning is to design a program so that it can learn a certain task, rather than implementing an algorithm by hand for a fixed behavior. Machine learning includes the definition of a variety of problems and provides many different algorithms to solve various problems in different fields. Some examples are the semantic segmentation [FGRK19, FRK19, FCZ+18], the classification [HCE+17] or the regression of values [MWL16, FK19]. The fields of application are manifold, such as human computer interaction [Fuh20], the automotive industry [XAJ+18, AAB+17], medicine [ESF+17, EFK17, EHF+17, BFG+16], data analysis as well as eye tracking [Fuh19] and many more. In the field of eye tracking, neural networks are used for pupil detection [FSR+16, FKH+17, FGS+18, FGK20b, FSK17b, FEH+18, FKS+15b, FSKK16, FTBK16], eyelid segmentation [FSG+16, FSG+17, FSK17a], eye movement classification [FRE20, FSK+18, FCK18a, FCK18b, FK18, FRE20], gaze vector regression [FGK20a], scanpath analysis [FBH+19, FCK+19] or for basic data cleansing for visualizations [FKB+18, FKS+15a, GFSK17, FKSK18]. My work is a cornerstone for a software which allows eye tracking studies via the browser. Using the browser for machine learning has unparalleled advantages [PSL+16]. Accessing web pages through a browser has a lower threshold and a wider spread of content. The ability to integrate intelligent factors into web pages will give us experience a pair of flying wings. Traditional intelligent effects are mostly implemented on the server side due to the size of the model and the computing power of the equipment and machines, but this requires multiple information interactions with the server on the network, and the user experience is discounted.

1.1 Advantages in a browser-based environment

Easy to share : A major motivation behind this is the ability to run ML in standard browsers, without any additional installations, therefore the used models have to be computationally cheap [FRM+20, FKRK20, SHZ+18]. Models and applications through the browser are easily shared on the web, lowering the barrier to entry for machine learning. This is particularly important for educational use cases and for increasing the diversity of contributors to the field.

Streamline interaction : From a machine learning perspective, the interactive nature of web browsers and versatile capabilities of Web APIs open the possibility for a wide range of novel user-centric ML applications which can serve both education and research purposes. Visualizations of neural networks such as [Ola14] and [SW] have been popular to teach the basic concepts of machine learning.

Local device call: Lastly, standardized access to various components of device hardware such as the web camera, microphone, and the accelerometer in the browser allow easy integration between ML models and sensor data. An important result of this integration is that user data can stay on-device and preserve user-privacy, enabling applications in the medical, accessibility, and personalized ML domains. For example, speech-impaired users can use their phones to collect audio samples to train a personalized model in the browser. Another technology, called Federated Learning [MMR+16], enables devices to collaboratively train a centralized model while keeping sensitive data on device. Browsers are a natural a platform for this type of application.

In this article, I will mainly use two ideas to implement the browser-side machine learning algorithm.Then the two ways are compared and analyzed,at the same time the results are obtained according to different models and different neural network structures, and finally we evaluate these results.

Convert existing models through converters.

As the name suggests, the first method I want to introduce is to convert the existing model through an appropriate converter, so as to convert it into a program that can be run in a browser. Traditionally, we first need to simplify and refine the model we want to train, and then build the corresponding neural network according to different needs.Then according to the framework of our choice (the framework used in this article is the open source framework **"tensorflow"** [ABC+16] and **"tensorflow.js"** from Google) to refine and select the features and use **python** or **C++** to implement locally. Later we need to select and build a suitable converter to call our previous model python so that it can be unimpeded in the **javascript script**. It is precisely because we can call the training model through javascript/html scripts, which finally allows us to implement machine learning operation and evaluation in the browser. It is very convenient to use Python to train the model, because reading data and GPU acceleration in the Python environment are relatively easy to implement, so we only need to solve the problem of porting the Python trained model to the js environment.

Directly use Javascript to build a model for training.

This is another way that i also would like to introduce.Because this article will use a tensorflow.js framework, TensorFlow.js is the JavaScript version of TensorFlow, which supports GPU hardware acceleration, and it also can run in **Node.js** or browser environments. Tensorflow.js itself comes with many tools and packages, which makes model training directly in TensorFlow.js more convenient and intuitive. (The specific details about Tensorflow and Tensorflow.js will be mentioned below)

2 Fundamentals

Fundamentally speaking, no matter what kind of method and what kind of idee, we can never be separated from the level of machine learning. So we still have to clarify the tasks of our machine learning and the elements needed in the process. Successful machine learning has four elements: data, a model to transform the data, a loss function for the size of the model, and an algorithm to adjust the weight of the model to minimize the loss function.
Data: The more the better. In fact, data is at the core of the deep learning renaissance, because complex nonlinear models require more data than other machine learning. Examples of data include:
1. Picture: For example, your phone picture, which may contain cats, dogs, dinosaurs, high school reunions or yesterday's dinner.
2. Text: Mail, news and WeChat chat history.
3. Sound: audio books and phone records.
4. Structural data: Jupyter notebook (with text, pictures and codes), web pages, car rental bills and electricity bills.
Model: Usually the data is far from what we ultimately want. For example, we want to know if the person in the photo is happy, so we need to turn 10 million pixels into a probability value of happiness. Usually we need to apply several non-linear functions (such as neural networks) to the data.
Loss function: We need to compare the error between the output of the model and the true value. The loss function helps us decide if Amazon stock will be worth 1,500 dollars at the end of 2020. Depending on whether we want to be short-term or long-term, this function can be very different.
Training: Usually there are many parameters in a model. We learn these parameters by minimizing the loss function. Unfortunately, even if we do well, there is no guarantee that we can still do well on new, unseen data.
Training error: This is the error of the model in evaluating the data set used to train the model. This is similar to the score we got on the mock test paper before the exam. There is a certain direction, but the real test scores are not necessarily guaranteed.
Test error: This is the error of the model on new data that has not been seen before, and may be different from the training error (statistically called over-fitting). This is similar to getting high scores in the pre-exam model test, but it actually made a mistake in the test. (One of the authors used to get high scores every time when doing the GRE real test. I was happy to recite the red book and then I really went to the exam. In the end, I got a low score that was just enough. Later I realized that this was because of the red book. The book contains a lot of real questions.)

Figure 2.1
Tensorflow

2.1 Framework

The machine learning framework is like a tool to help us in deep learning. In short, it is a library. When programming, we need to use "import".

Let us make a simple analogy, a set of machine learning framework is a set of building blocks of this brand. Each component is a part of a certain model or algorithm. We can design how to use the building blocks to build blocks that fit your data set. The advantage is that we don't have to reinvent the wheel because the model is known. We can assemble it directly, but different assembly methods, that is, different data sets, are up to us.
This article will focus on two machine learning frameworks that have been used: "Tensorflow" and "Tensorflow.js".

2.1.1 Tensorflow

Tensorflow is an open source software library that uses data flow graphs for numerical operations. It implements data flow graphs. Among them, "tensors" can be processed by a series of graph-describe algorithms, and the changes of data in the system are called "flows". Hence the name. The data stream can be coded in C++ or Python and run on a CPU or GPU device [Dev16]. In fact, TensorFlow provides a very rich API related to deep learning. It can be said that all the current deep learning frameworks provide the most complete API, including basic vector matrix calculation, various optimization algorithms, convolution neural networks and the realization of the basic unit of cyclic neural network, and the auxiliary tools of visualization, etc.

In this article, all models built in python are based on the tensorflow framework.

Basic use of TensorFlow
· Use graphs to represent computational tasks.

· The graph is executed in a context called a session.
· Use tensor to represent data.
· Maintain the state through Variables.
· Use feed and fetch to assign values to or retrieve data from arbitrary operations.

TensorFlow overview

TensorFlow is a programming system that uses graphs to represent computing tasks. The nodes in the graph are called **op** (short for operation). An **op** gets 0 or more Tensors, performs calculations, and produces 0 or more Tensors. Each tensor is a typed multidimensional array. For example, you can represent a small set of images as a four-dimensional array of floating-point numbers. The four dimensions are [batch, height, width, channels].

A TensorFlow graph describes the process of calculation. In order to perform calculations, the graph must be started in a session. The session distributes the op of the graph to devices such as CPU or GPU, and provides methods to execute the **op**. After these methods are executed, the generated tensor is returned. In the Python language, the returned tensor is a numpy nd-array object; in C and C++ languages, the returned tensor is an instance.

TensorFlow calculation graph

TensorFlow programs are usually organized into a construction phase and an execution phase. In the construction phase, the execution steps of the **op** are described as a graph. In the execution phase, the **op** in the execution graph is executed using the session.
For example, a graph is usually created in the construction phase to represent and train the neural network, and then the training **op** in the graph is repeatedly executed in the execution phase.

TensorFlow construction graph

The first step in building a graph is to create a source op. The source op does not require any input, such as Constant. The output of the source op is passed to other ops for calculation.
In the Python library, the return value of the op constructor represents the output of the constructed op, and these return values can be passed to other op constructors as input.

The following is a simple example of matrix multiplication to illustrate how to complete the construction graph:

```python
import tensorflow as tf
```

```
# Create a constant op to produce a 1x2 matrix.This op is used as a node
# Add to the default picture.

# The return value of the constructor represents the return value of the constant op.
matrix1 = tf.constant([[3., 3.]])

# Create another constant op to generate a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])

# Create a matrix multiplication matmul op with'matrix1' and'matrix2' as input.
# The return value'product' represents the result of matrix multiplication.
product = tf.matmul(matrix1, matrix2)
```

Now we default the graph has three nodes, two **constant()** op, and one **matmul()** op. In order to actually perform the matrix multiplication operation and get the result of the matrix multiplication, we must activate this diagram in the session.

```
# Start the default image.
sess = tf.Session()

# Call the'run()' method of sess to perform matrix multiplication op, and pass in'product' as the method parameter.
# As mentioned above,'product' represents the output of matrix multiplication op, passing it in is to indicate to the method that we want to get it back
# The output of matrix multiplication op.

# The return value'result' is a numpy ‘ndarray‘ object.
result = sess.run(product)
print result
# ==> [[ 12.]]

# Task completed, close the session.
sess.close()
```

Interactive call

Here is a Python example to explain interactive calls. We use a session to start the graph, and call the **Session.run()** method to perform the operation.

```
# Enter an interactive TensorFlow session.

import tensorflow as tf
sess = tf.InteractiveSession()

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])

```

```
# Use initializer initializer op's run() method to initialize'x'

x.initializer.run()

# Add a subtraction sub op, subtract'a' from'x'. Run the subtraction
    op, and output the result.

sub = tf.sub(x, a)
print sub.eval()
# ==> [-2. -1.]
```

In order to adapt to the Python interactive environment, we can use Interactive Session instead of the Session class, and use the Tensor.eval() and Operation.run() methods instead of Session.run(). This avoids using a variable to hold the session.

Tensor
TensorFlow program uses the tensor data structure to represent all data. In the calculation graph, the data passed between operations are all tensors. We can think of TensorFlow tensor as an n-dimensional array or list. A tensor contains a static type rank and a shape.
The above content introduces the basic use of the Tensorflow framework in this article

2.1.2 Tensorflow.js

The second framework that this article focuses on is Tensorflow.js. **TensorFlow.js** is the JavaScript version of TensorFlow, supports GPU hardware acceleration, and can run in Node.js or browser environments. It not only supports developing, training, and deploying models from scratch based entirely on JavaScript, but also can be used to run existing Python version TensorFlow models, or continue training based on existing models. This undoubtedly gives us a lot of support and help in the development of the web.

Use TensorFlow.js in the browser
The most convenient way to load TensorFlow.js in the browser is to directly quote the installed JavaScript code in the NPM package released by TensorFlow.js in the HTML.

```
<html>
<head>
    <script src="http://unpkg.com/@tensorflow/tfjs/dist/tf.min.js">
    </script>
```

If we need to use it on the server side, we need to ensure that the latest version of Node.js is installed.

```
#Initialize the project management file package.json
$ npm init -y

# Install tfjs library, pure JavaScript version
$ npm install @tensorflow/tfjs

# Install tfjs-node library, C Binding version
$ npm install @tensorflow/tfjs-node

# Install tfjs-node-gpu library to support CUDA GPU acceleration
$ npm install @tensorflow/tfjs-node-gpu
```

Load the Python model in the browser
Generally, TensorFlow models will be stored in **SavedModel** format. This is also the best way for us to save models in python programs. The **SavedModel** format can be converted to a format that can be directly loaded by TensorFlow.js through the **tensorflowjs-converter**, so that it can be used in the JavaScript language.
Below we take a "Mobilenet" as an example to see how to convert model files, and store the model files that can be loaded by TensorFlow.js in the $/mobilenet/tfjs\_model$ directory.

We first need to convert the **SavedModel**: convert $/mobilenet/saved\_model$ to $/mobilenet/tfjs\_model$.

```
tensorflowjs_converter \
    --input_format=tf_saved_model \
    --output_node_names='Mobilenet/Predictions/Reshape_1' \
    --saved_model_tags=serve \
    /mobilenet/saved_model \
    /mobilenet/tfjs_model
```

The converted model is saved as two types of files:

$model.json$: model architecture

$group1-shard*of*$: model parameters

For example, the files we converted for MobileNet are as follows:

$/mobilenet/tfjs\_model/model.json/mobilenet/tfjs\_model/group1-shard1of5$
$/mobilenet/tfjs\_model/group1-shard2of5$
...
$/mobilenet/tfjs_model/group1-shard5of5$

And next in order to load the converted model file, we need to install the *tfjs−converter* and *@tensorflow/tfjs_modules*

```
$ npm install @tensorflow/tfjs:
```

Finally, we can load the TensorFlow model through JavaScript.

```
import * as tf from '@tensorflow/tfjs'

const MODEL_URL = '/mobilenet/tfjs_model/model.json'

const model = await tf.loadGraphModel(MODEL_URL)

const cat = document.getElementById('cat')
model.execute(tf.browser.fromPixels(cat))
```

Build the model directly through TensorFlow.js
Based on the comprehensive functions and configuration of tensorflow.js, we can bulid a simple **HTML** page to call TensorFlow.js and its trained model, so that in our user's browser, we can directly perform browser online learning.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport"
  content="width=device-width,␣initial-scale=1.0">
  <title>TensorFlow.js Example</title>

  <!-- Import TensorFlow.js -->
  <script
          src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@1.0.0/dist/tf.min.js"></script>
  <!-- Import tfjs-vis -->
  <script
          src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis@1.0.2/dist/tfjs-vis.umd.min.js"></script>

  <!-- Import the data file -->
  <script src="data.js" type="module"></script>

  <!-- Import the main script file -->
  <script src="script.js" type="module"></script>

</head>

<body>
</body>
</html>
```

In this html file, the *script.js* file is the training model we built using tensorflow.js through javascript, and the *data.js* file is the data we want to use and process. One thing to note here, the two files need to be placed in the same folder as the above HTML file.

TensorFlow.js Model library

In addition to the above loading and construction functions, TensorFlow.js also provides a series of pre-trained models to facilitate us to quickly introduce artificial intelligence capabilities to the program.

The model classification includes image recognition, speech recognition, human gesture recognition, object recognition, text classification and so on.(Google Cloud)

Figure 2.2
Tensorflow.js

2.2 Optimization Algorithm

When constructing a neural network model and training in browser, the best optimizer is selected so as to quickly converge and learn correctly, while adjusting internal parameters to minimize the loss function to the greatest extent.In order to better simplify the neural network and improve efficiency, so that our model can be called in the browser, it is necessary to choose **Mini-batch Gradient Descent** and **Adam** optimization algorithm.

2.2.1 Three forms of gradient descent

In general, mini-batch gradient descent algorithm is a variant of **gradient descent algorithm**. Gradient descent Algorithm is the most commonly used optimization algorithm in machine learning. It has three different forms: "batch gradient descent", "stochastic gradient descent", "mini-batch gradient descent". Among them, the mini-batch gradient descent algorithm also often carries out model training in deep learning. Next, we will understand these three different gradient descent methode and explain why we choose mini-bach gradient Algorithm.

Batch Gradient Descent(BGD)

Optimization algorithms that use the entire training set are called batch or deterministic gradient algorithms because they process all samples simultaneously in a large batch.
The batch gradient descent method is the most primitive form, which refers to using all samples to update the gradient at each iteration.

The loss function for all samples is:

$$J(\theta) = J(\theta_0, \theta_1, \theta_2 ......\theta_i) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^i) - y^i)$$

The parameter update is as follows:

$$\theta_j \leftarrow \theta_j - \eta . \nabla J(\theta)$$

θ is an n-dimensional weight vector
$J(\theta)$ is the loss function.
$\nabla J(\theta)$ is the gradient of the parameter.
The step η is also called the learning rate.

Advantages:
. During the training process, it use a fixed learning rate without worrying about the decline of the learning rate.
. The direction determined by the full data set can better represent the sample population, so as to be more accurate toward the direction of the extreme value. When the objective function is a convex function, it must converge to the global minimum, and if the objective function is not convex, it will converge to the local minimum.
. Its estimation of the gradient is unbiased. The more samples, the lower the standard deviation.

Disadvantages:
.Although vectorization calculation is used in the calculation process, it still takes a lot of time to traverse all samples, especially when the data set is very large (millions or even hundreds of millions), it is a bit powerless.

. Each update occurs after traversing all the samples. Only then will some examples be found to be redundant and have little effect on the parameter update.
. Sometimes unbalanced data sets are also problematic.

Stochastic Gradient Descent(SGD)

Stochastic gradient descent is different from batch gradient descent. Stochastic gradient descent uses only one sample to update the parameters at each iteration (mini-batch size = 1).

The loss function for one samples (x^i, y^i) is:

$$J(\theta;x^i;y^i) = J(\theta_0,\theta_1,\theta_2......\theta_i) = \frac{1}{2}(h_\theta(x^i) - y^i)$$

The parameter update is as follows:

$$\theta_j \leftarrow \theta_j - \eta . \nabla J(\theta;x^i;y^i)$$

Advantages:
. Since it is not a loss function on all training data, but in each iteration, the loss function on a certain training data is randomly optimized, so that the update speed of each round of parameters is greatly accelerated.

Disadvantages:
.A single sample does not represent the trend of all samples.
. When encountering a local minimum or saddle point, SGD will get stuck at a gradient of zero.Because randomness will also cause convergences becoming complex, even when it reaches the minimum, SGD will over-optimize, which makes the process full with fluctuation and overfitting.

Mini-batch Gradient Descen(MBGD)

Most gradient descent algorithms for deep learning are using more than one but not all training samples. These will be referred to as mini-batch or mini-batch stochastic methods, but now they are usually simply referred to as stochastic methods. For the deep learning model, what people call "stochastic gradient descent, SGD" is actually a random batch based on mini-batch.

Specifically: At each step of the algorithm, we randomly draw a mini-batch sample $X = (x^1, x^2.....x^{m'})$ from the training set with m samples (the order of the samples has been shuffled). The number of small batches m' is usually a relatively small number (from 1 to several hundred). Importantly, as the training set size m grows, [formula] is usually fixed. When fitting billions of samples, we may use only a few hundred samples for each update calculation (Sebastian Rruder,2016).

The loss function for m' samples $X = (x^1, x^2.....x^{m'})$ is:

$$J(\theta; x^{i:m'}; y^{i:m'}) = J(\theta_0, \theta_1, \theta_2......\theta_i) = \frac{1}{2m'}\sum_{i=1}^{m'}(h_\theta(x^i) - y^i)$$

The parameter update is as follows:

$$\theta_j \leftarrow \theta_j - \eta \,\textbf{.}\, \nabla J(\theta; x^{i:m'}; y^{i:m'})$$

Advantages:
. The speed is faster than Batch Gradient Descent, because the update can be performed only by traversing part of the samples.
. Randomly selecting samples is helpful to avoid repeating redundant samples and samples that contribute less to parameter updates.
. Using only one batch at a time can greatly reduce the number of iterations those required for convergence, and at the same time can make the convergence result closer to the effect of gradient descent.

Disadvantages:
. Improper selection of batch-size may cause some problems. Just like maybe there will be more oscillations in the learning process. Or maybe sometimes Stuck at the local minimum.

2.2.2 Challenges

Although we have learned about the three variants of the gradient descent algorithm, when we do machine learning in the browser, the situation is often a lot more complicated because it cannot guarantee good convergence. The following proposes some things we need to solve challenge:
(1) Choosing a proper learning rate can be difficult. A learning rate that is too small

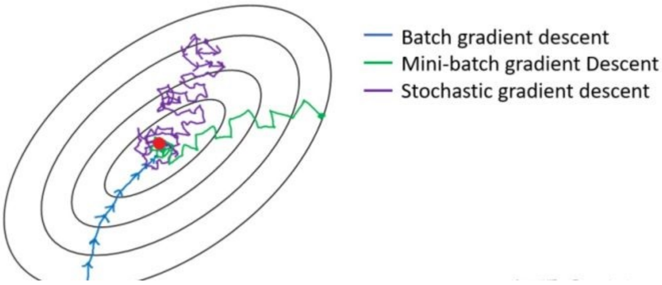


Figure 2.3
Convergence of three gradient descent algorithms

leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
(2)Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
(3)Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima.That the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions. In the following, I will explain some of the algorithms widely used in the deep learning community, and explain the reasons for finally using **adam** to solve the above challenges.

Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another [Sut86],.which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Figure 3.

Momentum [Qia] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in the Figure 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

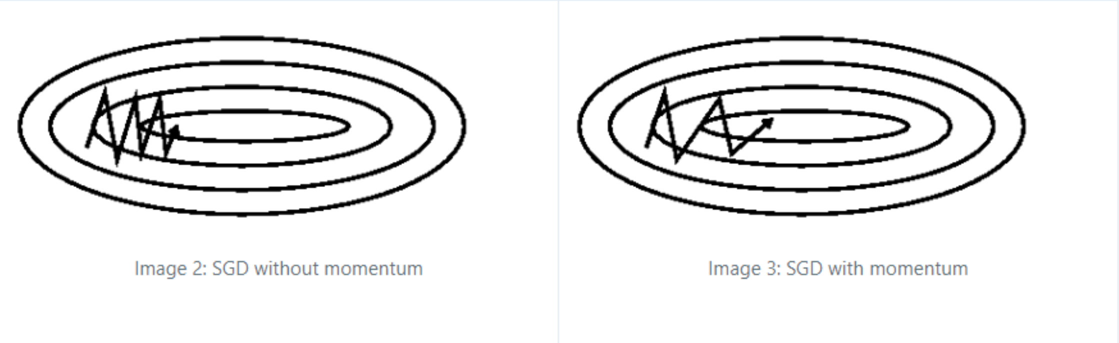


Figure 2.4: SGD without momentum and SGD with momentum [Rru16]

$$v_t \leftarrow \gamma v_{t-1} + \eta \mathbin{\textbf{.}} \nabla J(\theta)$$
$$\theta_j \leftarrow \theta_j - v_t$$

Note: Some implementations exchange the signs in the equations. The momentum term γ is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma < 1$) The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation [Rru16].

Nesterov accelerated gradient

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Nesterov accelerated gradient (NAG) [Nes83]is a way to give our momentum term this kind of prescience. We know that we will use our momentum term γv_{t-1} to move the parameters θ.Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

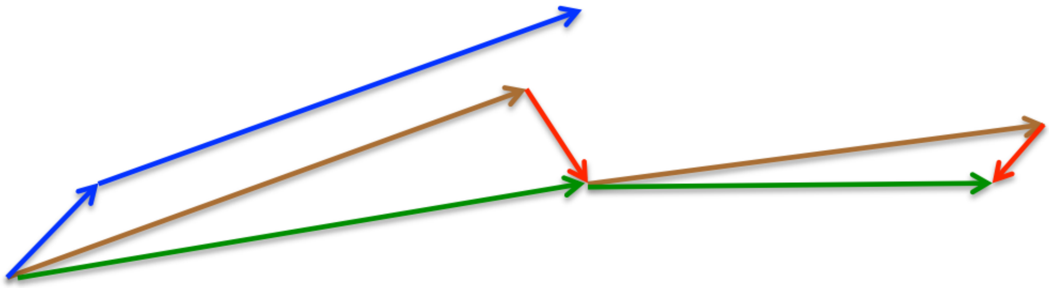


Figure 2.5: While Momentum first computes the current gradient (small blue vector) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector),NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (green vector).

$$v_t \leftarrow \gamma v_{t-1} + \eta \cdot \nabla J(\theta - \gamma v_{t-1})$$

$$\theta_j \leftarrow \theta_j - v_t$$

Again, we set the momentum term γ to a value of around 0.9. And we could take a look at the following example:
This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks[BBLP12].

Adagrad

Adagrad[Duc11]is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features.

Previously, we performed an update for all parameters θ at once as every parameter θ_i used the same learning rate η. As Adagrad uses a different learning rate for every parameter θ_i at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we use g_t to denote the gradient at time step t. $g_{t,i}$ is then the partial derivative of the objective function w.r.t. to the parameter θ_i at time step t:

$$g_{t,i} \leftarrow \nabla J(\theta_{t,i})$$

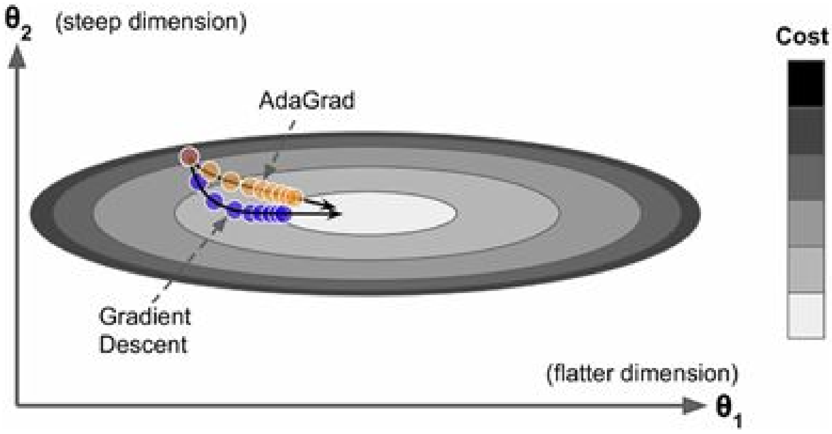


Figure 2.6: This figure shows the performance comparison between Adagrad and ordinary gradient descent in n-dimensional parameters. It is obvious that Adagrad curve drops slowly and converges better.
AdaGrad update

The SGD update for every parameter θ_i at each time step t then becomes:

$$\theta_{t+1,i} \leftarrow \theta_{t,i} - \eta g_{t,i}$$

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i :

$$\theta_{t+1,i} \leftarrow \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}+\epsilon}} g_{t,i}$$

$G_t \in \mathbb{R}^{dxd}$here is a diagonal matrix where each diagonal element i,i is the sum of the squares of the gradients w.r.t. $\theta_i$up to time step t, while ϵ is a smoothing term that avoids division by zero. Interestingly, without the square root operation, the algorithm performs much worse.

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that. Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

RMSprop

RMSprop is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, RMSprop restricts the window of accumulated past gradients to some fixed size w. Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)E[g^2]_t$$

We now simply replace the diagonal matrix $G_t \in \mathbb{R}^{dxd}$ with the decaying average over past squared gradients $E[g^2]_t$:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_{t,i}$$

We set γ to a similar value as the momentum term, around 0.9,while a good default value for the learning rate η is 0.001.

Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t ,similar to momentum [Kin15]. Where as momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface [Heu17]. We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the not centered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and $\beta_2$are close to 1).
They counteract these biases by computing bias-corrected first and second moment estimates:

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t}$$

$$\widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

They then use these to update the parameters just as we have seen in RMSprop, which yields the Adam update rule:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\widehat{v_t}} + \epsilon} \widehat{m_t}$$

We propose default values of 0.9 for β_1 , 0.999 for β_2 and 10^{-8} for ϵ.
That shows empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

2.3 Convolutional Neural Networks in Tensorflow.js

As we all know, Neural Network is the foundation of deep learning, and it is no exception when we perform machine learning in the browser. The basic concepts include: neurons, layers, back propagation and so on. If I go into detail, I guess there are no five to ten articles, then it will be endless. Simply put, it simulates the way brain neurons work, using a model that combines multiple neurons into a network structure to classify data.
When we use a browser for machine learning, it is unavoidable that we will be exposed to a lot of time series data (which can be considered as a one-dimensional grid formed by regular sampling on the time axis) and image data (which can be seen as two Dimensional pixel grid), then the best way to process these data is to use a convolution neural network. Because it is a neural network designed to process data with a similar grid structure. So let's specifically consider how to use tensorflow to build a CNN and process it in the browser.

2.3.1 Convolution and Kernels

The principle of CNN actually simulates how the human visual nerve recognizes images. Each optic nerve is only responsible for processing a small picture of different sizes, and processing different information at different neural levels.

We can see that the rectangle on the left side of the figure below is the input data, which is the tensor representation of the image we want to process. The rectangle in the middle is the kernel, and the rectangle on the right is the result of convolution. The kernel function moves from left to right and top to bottom, scanning the image by moving one pixel at a time, and calculating the result matrix of the convolution sum.

And the calculation process of convolution is as follows:

Calculation methods are very intuitive,simple multiplication and addition.
We can see that the kernel function is actually a weight. For each small image, different checks have different weights for different regions. For every singel convolution , the pixel values covered by the kernel are multiplied with the corresponing kernel values and the products are summated. The result is placed in the new image at the point corresponding to the centre of the kernel. The kernel is moved over by one pixel and this process is repated until all of the possible locations in the image are filtered. Notice that there is a border of empty values around the convolved image. This is because the result of convolution is placed at the centre of the kernel. To deal

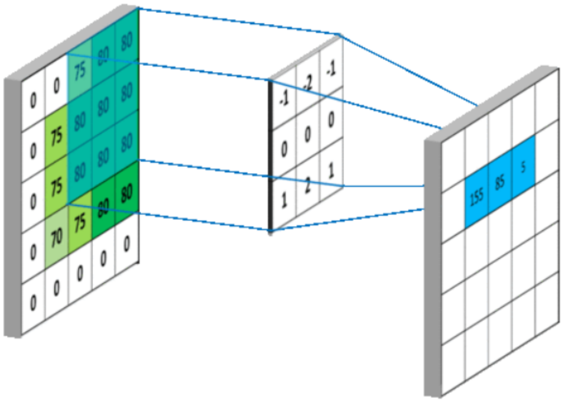


Figure 2.7: Convolution kernel multiplies and sums the layers.
Convolution operation graph

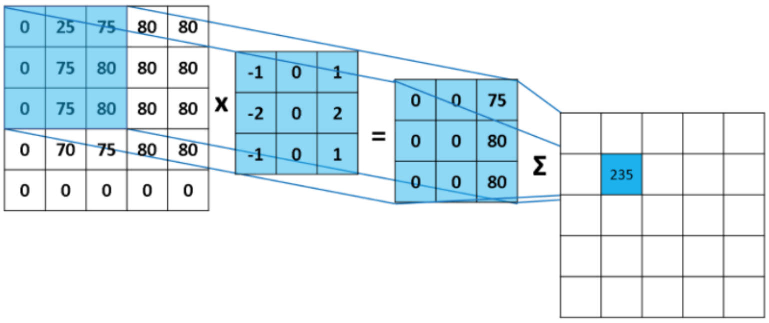


Figure 2.8: Flat display of convolution kernel operation.
Convolution operation graph

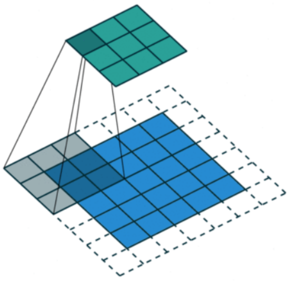


Figure 2.9
The input is 5x5, 1 grid is filled, stride is 2, output is 3x3

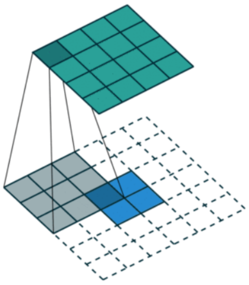


Figure 2.10
Input is 2x2, 2 grids are filled, stride is 1, output is 4x4

with this, a process called 'padding' or more commonly 'zero-padding' is used. This simply means that a border of zeros is placed around the original image to make it a pixel wider all around. The convolution is then done as normal, but the convolution result will now produce an image that is of equal size to the original.

From this we can see that,each pixel is a feature, and each feature is an input node. The result of each convolution is input to the hidden node of the next layer. The weight of the core connects the input layer and the hidden layer. The input layer filled with 0 can output convolution results of different shapes. At the same time, we can adjust the stride of the scan.

We can see that increasing the padding will increase the number of hidden layer nodes, and increasing the stride can make the hidden layer nodes less.

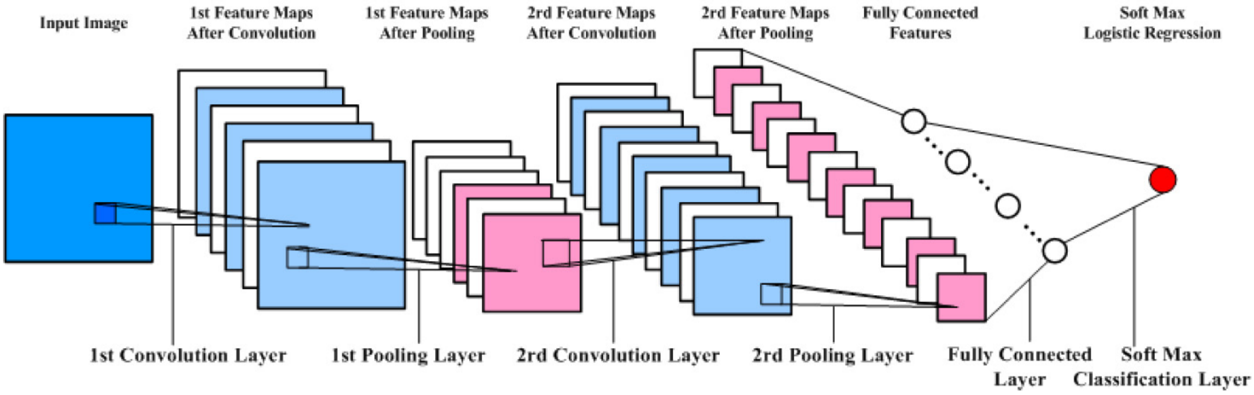

Input Image
1st Feature Maps After Convolution
1st Feature Maps After Pooling
2rd Feature Maps After Convolution
2rd Feature Maps After Pooling
Fully Connected Features
Soft Max Logistic Regression
1st Convolution Layer
1st Pooling Layer
2rd Convolution Layer
2rd Pooling Layer
Fully Connected Layer
Soft Max Classification Layer

Figure 2.11: The whole Architecture of CNN

2.3.2 CNN architecture

Generally, the CNN network consists of the layers shown in the Figure2.11.
Input Layer: The input image is placed into this layer. It can be a single-layer 2D image (gray scale), 2D 3-channel image (RGB colour) or 3D.
Convolution Layer: Convolution calculation, function activation (usually the most popular ReLU is a kind of activation function). In the previous article, we have studied the role of the convolution layer, and we will not repeat it here.
Pooling Layer: The pooling layer is usually a layer following the convolution, which is usually the average or maximum value of the region. Pooling is similar to sampling, which greatly reduces the data to be processed by the next layer of neural network. Reduce the parameters of the entire network to prevent over fitting. Usually our pooling strategy is to take the maximum or average.
Fully Connected (Dense) Layer: The fully connected layer mainly refits the features to reduce the loss of feature information.
Softmax Classification Layer: The softmax layer is similar to the softmax classifier in that it is the final output classification probability, so it is often used as the last layer in the deep learning network of the classification problem.
Output Layer: Information is transmitted, analyzed, and weighed in neuron links to form output results and is also called the output vector.

2.3.3 Implementation in Tensorflow.js

After understanding the basic convolutional network concept, let's take a look at how to implement a CNN in TensorflowJS.
Here is the code for a model example:

```
function cnn() {
  const model = tf.sequential();
  model.add(tf.layers.conv2d({
    inputShape: [28, 28, 1],
    kernelSize: 5,
    filters: 8,
    strides: 1,
    activation: 'relu',
    kernelInitializer: 'varianceScaling'
    }));
  model.add(tf.layers.maxPooling2d(
            {poolSize: [2, 2], strides: [2, 2]}));
  model.add(tf.layers.conv2d({
    kernelSize: 5,
    filters: 16,
    strides: 1,
    activation: 'relu',
    kernelInitializer: 'varianceScaling'
    }));
  model.add(tf.layers.maxPooling2d({poolSize: [2, 2], strides: [2,
      2]}));
  model.add(tf.layers.flatten());
  model.add(tf.layers.dense(
    {units: 10, kernelInitializer: 'varianceScaling',
                activation: 'softmax'}));
  return model;
}
```

tf.sequential() creates a continuous neural network and automatically creates the input layer.
tf.layers.conv2d is the first convolution layer, and the input 28*28*1 is the length, height, and color channels of the image. The size of the core is 5*5, and the stride is 1. Let's ignore the other parameters first.
tf.layers.maxPooling2d is the next pooling layer, which is to pool the convolution result with a small 2*2 window.
And then there is another convolution and pooling layer.
tf.layers.flatten() is to flatten the previous result.
The last is a softmax classification layer *tf.layers.dense*

CNN is a very popular deep learning model, widely used in image-related related fields, from "Alpha go" to autonomous driving, he is everywhere.When we want to use it, the last point to note is that we should consider the advantages and disadvantages of CNN when choosing a model.

Advantages:
1. Shared convolution kernel, no pressure on high-dimensional data processing
2. No need to manually select features, just train the weights to get the features.
3. Good classification effect.

Disadvantages:
1. People need to adjust parameters, need a large sample size and the training is best to use GPU.
2. The physical meaning is not clear. With the stacking of Convolutions, the Feature Map becomes more and more abstract, which is difficult for humans to understand[HL09].

3 Model and Evaluation

In the previous article, we have introduced a lot of key knowledge about how to perform machine learning in the browser. And in the following article, I want to guide you through the method of machine learning in the browser to solve some practical problems. Therefore, I will try my best to use some classic machine learning models like **LeNet [LHBB99], AlexNet [KSH12], GoogLeNet [SLJ$^+$15], ResNet [HZRS16], MobileNet [SHZ$^+$18]** on different datasets,so as to achieve the purpose of evaluating our machine learning models on the Web.

3.1 Model

3.1.1 LeNet

The LeNet network is trained for gray scale images. The input image size is 32*32*1. There are 7 layers in total without the input layer, and each layer contains trainable parameters (connection weights). In this model, the basic framework of the convolution neural network is already in place. The basic components of convolution, activation, pooling and full connection are complete.

Features:

1. Convolution networks use a three-layer sequence: convolution, pooling, and nonlinearity. These three characteristics lay the foundation for subsequent deep learning.
2. Local perception. Use convolution to extract local features. People's perception of the outside world is from local to global, and neighboring local pixels are closely connected. Each neuron does not need to perceive the global image, it only needs to perceive the part, and then the higher layer combines the local information to obtain the global information.
3. Parameter sharing. Each convolution kernel is a way to extract features, which greatly reduces the number of parameters.
4. Down sampling using average pooling.
5. Use multi-layer neural network (MLP) as the final classifier. ([YLJ89])

3.1.2 AlexNet

The AlexNet network has a total of 5 convolution layers, 3 pooling layers, 3 fully connected layers (including the output layer), and finally 1000 classifications. The last layer is a softmax output layer.
Features:
1. Using ReLU as a nonlinear activation function solves the gradient dispersion problem of sigmoid when the network is deep.
2. The fully connected layer uses dropout technology to selectively ignore individual neurons in training to avoid over fitting.
3. Use overlap max pooling to avoid the blur effect of average pooling.
4. Use multi-GPU training, greatly accelerate the training speed.
5. Proposed the local response normalization LRN (Local Response Normalization), although later proved to be of little effect, it gradually gave way to BatchNorm.
6. Data enhancement. Randomly intercept an area of 224*224 size (and the mirror image flipped horizontally) from 256*256 data, which greatly reduces over-fitting and enhances generalization ability[AKH12].

3.1.3 GoogLeNet:

The main innovation of this model lies in its Inception, which is a Network In Network structure, that is, the original node is also a network.
Features:
1. While controlling the amount of calculation and parameters, a good classification performance is obtained.
2. Removed the last fully connected layer and used a global average pooling layer.
3. Use Inception Module to improve parameter utilization.
4. Use different sizes of convolution kernels to increase diversity.
5. Introduce auxiliary classifiers (auxiliary classifiers), that is, the output of an intermediate layer is used as a classification, and a small weight (0.3) is added to the final classification.
6. Replace the large convolution kernel with multiple small convolution kernels.
7. n*n convolution kernel is changed to 1*n and n*1 convolution kernel, saving parameters, accelerating.
calculation, reducing over-fitting, and adding a layer of non-linear expansion model expression ability[CS15].

3.1.4 ResNet:

ResNet designs a residual module that allows us to train deeper networks. The proposal of this network is essentially to solve the problem that cannot be trained

when the level is deeper. It is similar to GoogLeNet, ResNet finally uses a mean pooling layer.
Features:
1. Using the residual module, a 152-layer residual network can be trained. Its accuracy is higher than GoogLeNet.
2. Make the learning result more sensitive to the fluctuation of the network weight.
3. The residual results are more sensitive to data fluctuations[HZRS15].

3.1.5 MobileNet

MobileNet introduces the concept of separable convolution. In simpler terms, it decomposes the two-dimensional convolution kernel into two separate convolution kernels, namely the depth direction (responsible for collecting the spatial information of each individual channel) and the point direction (responsible for processing the inter-channel Interactive).
Features:
1. Lightweight network. Compression is performed on the trained model, so that the network carries fewer network parameters, thereby solving the memory problem and the speed problem.
2. Separable convolution reduces network parameters without losing network performance[NOF20].

3.2 Dataset

In order to facilitate our better observation and evaluation of the use of machine learning algorithms and models in the browser, here are three data sets to be tested for different tasks.

3.2.1 Cars

(View here [RD83]) This was one of a ASA(American Statistical Association) Data Exposition dataset. For analysis, data was on mpg, cylinders, displacement, etc. (8 variables) for 406 different cars. The dataset also includes the names of the cars. Later we will use this data set for our regression task prediction.

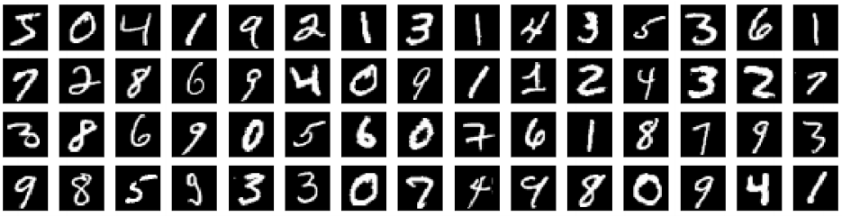


Figure 3.1
Images from the MNIST training set.

3.2.2 MNIST

“MNIST", a handwritten digit recognition data set. It is one of the most basic data sets used to test performance of neural network models and learning techniques. Using 60,000 images as the training set, a $97\% - 98\%$ accuracy could easily be achieved on the test set of 10,000 images, with learning methods such as k-nearest neighbors (KNN), random forests, support vector machines (SVM)and simple neural networks models[SA08].

In this article, there are a total of 65,000 images, we will use up to 55,000 images to train the model, saving 10,000 images that we can use to test the model's performance once we are done. Each image is 28px wide 28px high and has a 1 color channel as it is a gray scale image. So the shape of each image is [28, 28, 1]. And we are going to do all of that in the browser.

3.2.3 Cifar-10

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.
This dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.
Here are the classes in the dataset, as well as 10 random images from each:

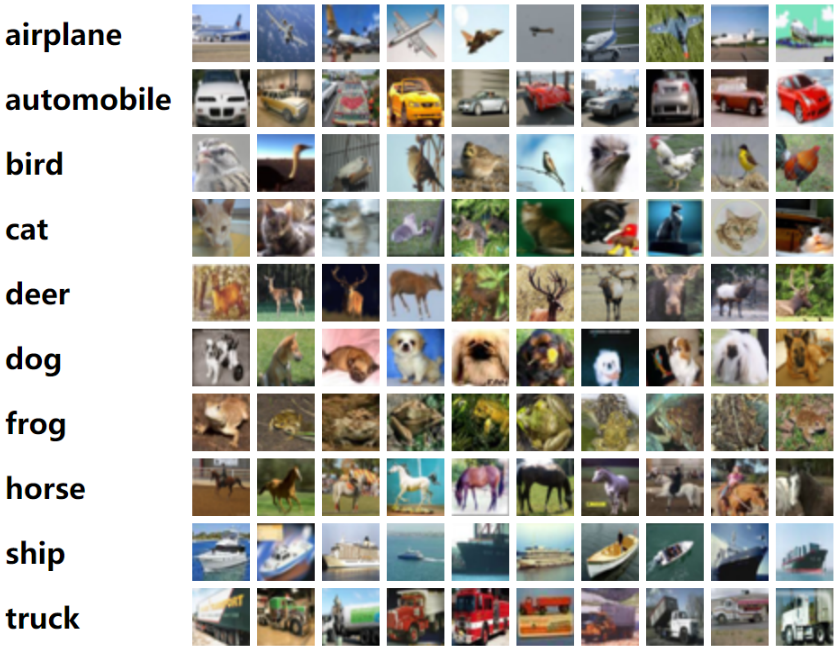


Figure 3.2
Images from the Cifar-10 training set.

3.3 Training task

3.3.1 Regression

By using machine learning (ML), we want to solve a real problem. If we think about the problem, what would be the characteristics of a car to predict its price? The Weight? The year when it was produced? The point is that there are endless of characteristics for a car that could contribute to the price of a house. These characteristics of one date (car) in a data set (cars) are called features. For instance, the horsepower and miles per gallon are features. In order to simplify the process of learning ML in the article by using linear regression and gradient descent [MWL16, FK19], let's assume that the only feature that affects the miles per gallon is its horsepower. That way, we can apply an univariate linear regression that simplifies the algorithm, because we are only using one feature.
Here i will load the **"Cars"** data-set,(which can be found below). It contains many different features for a given car. For this model, we want to only extract data about Horsepower and Miles Per Gallon.
First we will also remove any entries that do not have either miles per gallon or horsepower defined. Let's also plot this data in a scatter plot to see what it looks like.
We can see from the plot(Figure 3.3) that there is a negative correlation between horsepower and MPG, i.e. as horse power goes up, cars generally get fewer miles

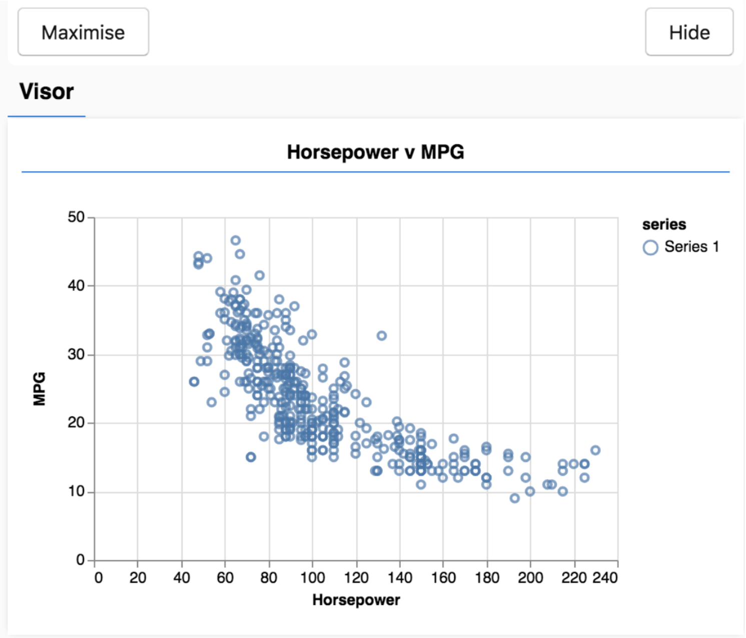


Figure 3.3: Horsepower V MPG

per gallon.
Second add an input layer to our network, which is automatically connected to a "dense" layer with a hidden unit. The dense layer is a type of layer that multiplies its input by a matrix (called weights), and then adds a number (called bias).Next we can set the size of the weight matrix in the layer. By setting it to 1 here, we could say that the weight of each input feature of the data is 1. To get the performance benefits of TensorFlow.js that make training machine learning models practical, we need to convert our data to tensors. We will also perform a number of transformations on our data that are best practices, namely shuffling and normalization.
Before we train this model,we have to 'compile' the model. To do so, we need to specify a number of very important things: There are many optimizers available in TensorFlow.js. Here we have picked the **adam** optimizer as it is quite effective in practice and requires no configuration. At the same time for loss function, we use mean Squared Error to compare the predictions made by the model with the true values. Batch sizes tend to be in the range 32-512 and we will take 50 iterations through the dataset.

We will see in Figure3.4that they display the loss and mse, averaged over the whole dataset, at the end of each epoch.When training a model we want to see the loss go down. In this case, because our metric is a measure of error, we want to see it go down as well.
Now that our model is trained, we want to make some predictions and evaluate the

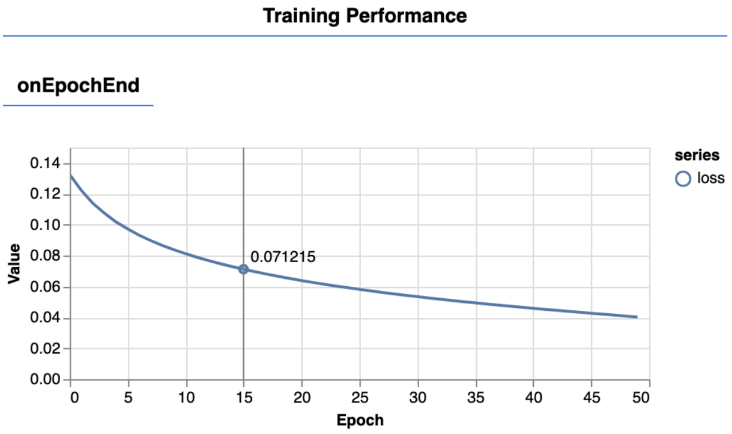


Figure 3.4: This displays the loss and mse, averaged over the whole dataset, at the end of each epoch.

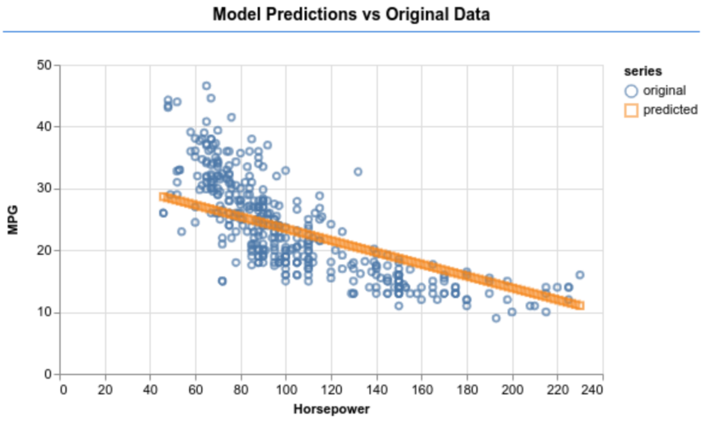


Figure 3.5: The "slash" indicates the prediction performance of the model, which tries to fit a line to the trend present in input data.

model by seeing what it predicts for a uniform range of numbers of low to high horsepowers.
This Figure3.5 currently performs what is known as linear regression which tries to fit a line to the trend present in input data.

3.3.2 Classifition

For evaluating classification tasks in machine learning, in this module we will make a web page that uses TensorFlow.js to train a model in the browser [HCE+17]. Given a black and white image of a particular size it will classify which digit appears in the image.
First, we'll train the classifier by having it "look" at thousands of hand written digit images and their labels. Then we'll evaluate the classifier's accuracy using test data that the model has never seen. This task is considered a classification task as we are training the model to assign a category (the digit that appears inthe image) to the input image. We will train the model by showing it many examples of inputs along with the correct output. This is referred to as supervised learning.
Similarly, after loading the **MNIST** training data, we need to configure the parameters in our model: The size of the sliding convolution filter windows to be applied to the input data (kernel Size) could be 5, which specifies a square, 5x5 convolution window. We will apply 8 filters to the data. Here, we also specify strides of 1, which means that the filter will slide over the image in steps of 1 pixel. After the convolution is complete. In this case, we are applying a **Rectified Linear Unit (ReLU)** function as activation function, which is a very common activation function in ML models.
When we compile the model, we need to specify the optimizer, loss function and indicator to be tracked. Contrary to our first regression task, here we use **categorical Crossentropy** as the loss function. As the name implies, this name will be used when the output of our model is a probability distribution. The categorical Crossentropy measures the error between the probability distribution generated by the last layer of the model and the probability distribution given by the real label.We could see Figure 3.6 and Figure 3.7 reporting training progress.

After training the model, we need to make some predictions. Here, we will take 500 images and predict the number of bits in them (Of course, we can also increase this number to test larger image sets).
Notably the **argmax** function is what gives us the index of the highest probability class. Remember that the model outputs a probability for each class. Here we find out the highest probability and assign use that as the prediction. We should notice that we can do predictions on all 500 examples at once. This is the power of vectorization that TensorFlow.js provides.

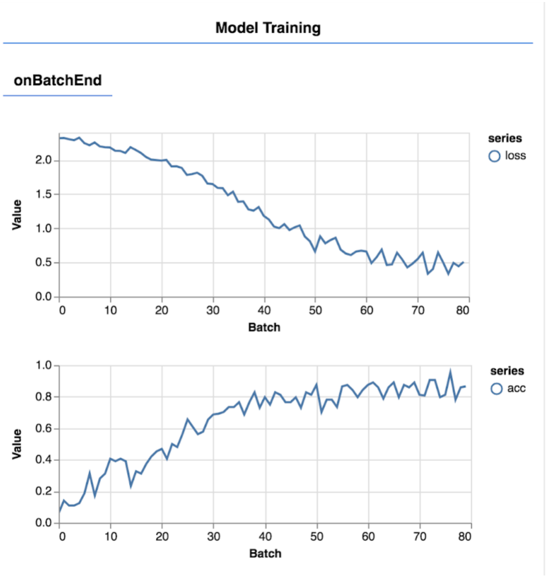


Figure 3.6: When training a model we want to see the loss go down. In this case, because our metric is a measure of error, we want to see it go down as well.

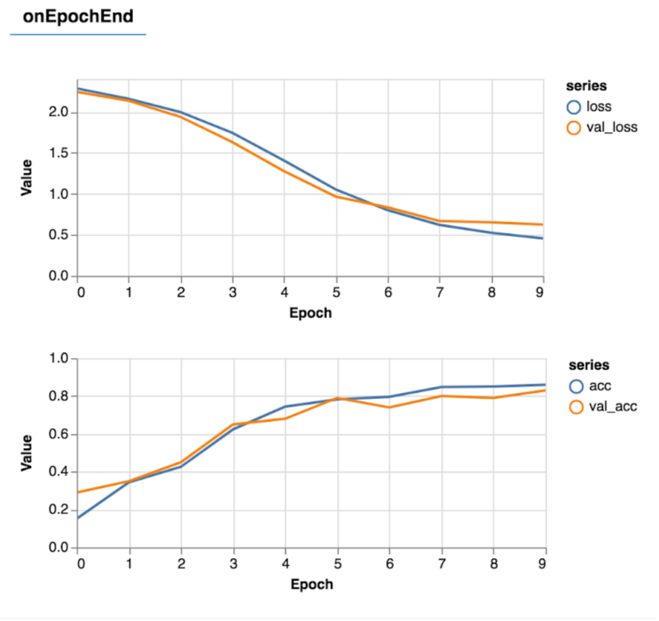
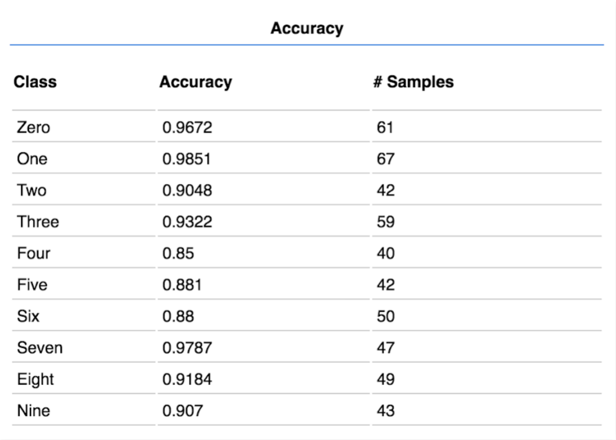


Figure 3.7: As the number of epochs of our training increases, our accuracy value keeps increasing, and the loss value is gradually decreasing



Accuracy

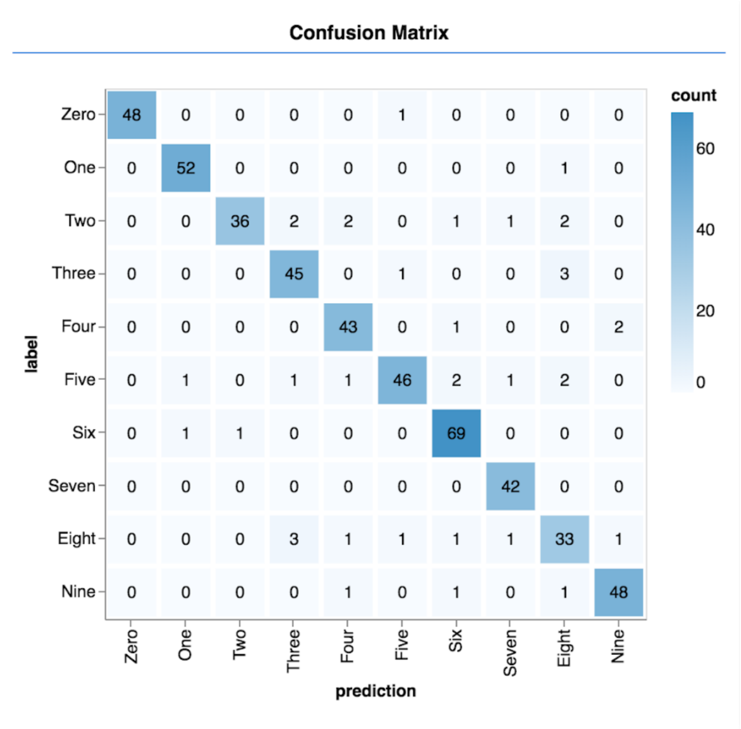
Class	Accuracy	# Samples
Zero	0.9672	61
One	0.9851	67
Two	0.9048	42
Three	0.9322	59
Four	0.85	40
Five	0.881	42
Six	0.88	50
Seven	0.9787	47
Eight	0.9184	49
Nine	0.907	43

Figure 3.8: The performance is good, and the highest accurate prediction rate for the number "Zero" reaches 98.51%

Show per class accuracy
With a set of predictions and labels we can calculate accuracy for each class. We could see a display in our webpage that looks like the Figure3.8.

Show a confusion matrix
A confusion matrix in the Figure3.9 is similar to per class accuracy but further breaks it down to show patterns of misclassification. It allows us to see if the model is getting confused about any particular pairs of classes. In the results, we can clearly see that the prediction accuracy of the MNIST dataset in the browser is amazing for machine learning anyway [FKB+18, FKS+15a, GFSK17, FKSK18].

For the **Cifar-10** data set, the steps we need to configure the neural network parameters are similar to MNIST. The same is to train the model, then store the model and use **tfjs−converter** to convert to a model that can be run on an html/javascript scripts.
But this dataset is more difficult and it takes longer to train a network. Data augmentation includes random flipping and random image shifts by up to 2px horizontally and vertically. We need to notice that by default, we will use Adadelta, which is one of the per-parameter adaptive step size methods, so we don't have to worry about the learning rate or momentum that changes over time. However, there is also the possibility of using SGD + Momentum trainer.
Here I will show some results based on last 200 test images in Figure3.10.



	Zero	One	Two	Three	Four	Five	Six	Seven	Eight	Nine
Zero	48	0	0	0	0	1	0	0	0	0
One	0	52	0	0	0	0	0	0	1	0
Two	0	0	36	2	2	0	1	1	2	0
Three	0	0	0	45	0	1	0	0	3	0
Four	0	0	0	0	43	0	1	0	0	2
Five	0	1	0	1	1	46	2	1	2	0
Six	0	1	1	0	0	0	69	0	0	0
Seven	0	0	0	0	0	0	0	42	0	0
Eight	0	0	0	3	1	1	1	1	33	1
Nine	0	0	0	0	1	0	1	0	1	48

Figure 3.9: The highest count in the confusion matrix is 69 for the number “Six", and the lowest count is 33 for the number "Eight".

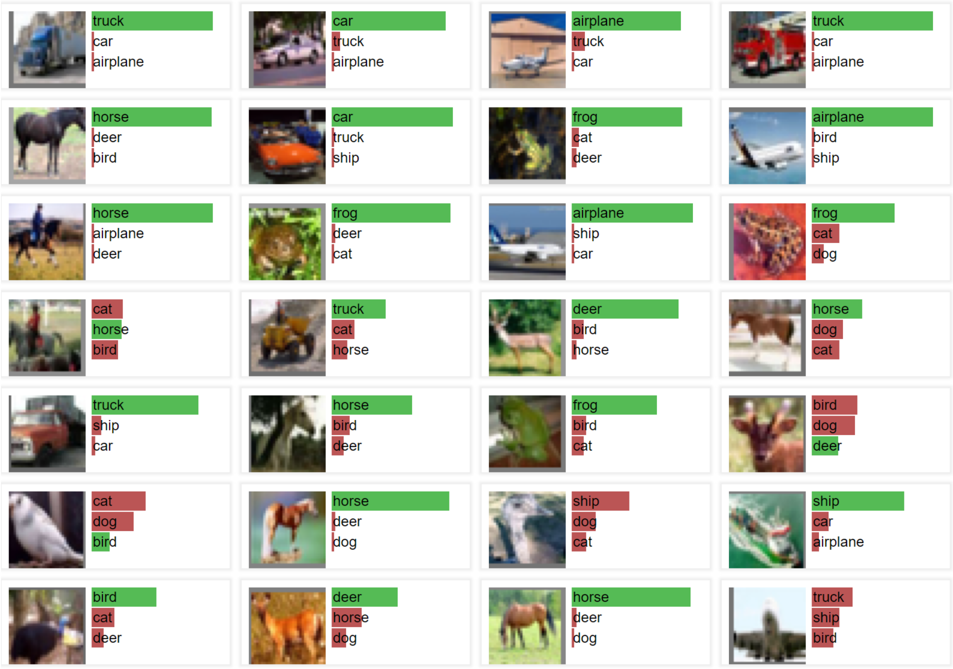


Figure 3.10: The green rectangles in each object represent examples of successful and accurate predictions, and the red represent incorrect predictions

From the results, the test accuracy nearly reached 80%.

3.4 Performance comparison

When we deploy the above machine learning model in the browser, we want to know the difference between different parameters and different models on the efficiency of machine learning. In other words, what kind of model is most suitable for browser-side machine learning? How does it perform? What effect will we have if we change the parameters of the neural network model? In this section, we will apply different models for our data set in order to observe their respective performance.

For the **"Cars"** dataset in the regression task, we want to change its model parameters and observe its performance changes. First, we try to change the number of epochs, the new epochs number should be doubled. Second, we try to increase the units in the hidden layer and try to add more hidden layers between the first hidden layer added and the final output layer. The code for these additional layers should look like the following.

```
model.add(tf.layers.dense({units: 50, activation: 'sigmoid'}));
```

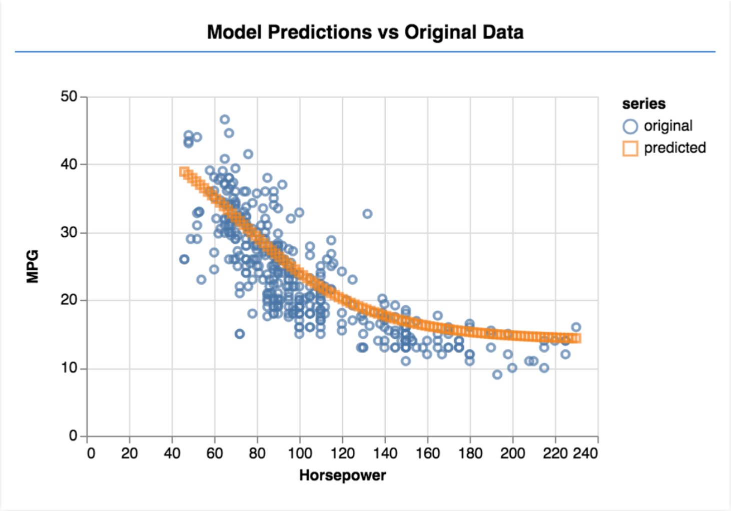


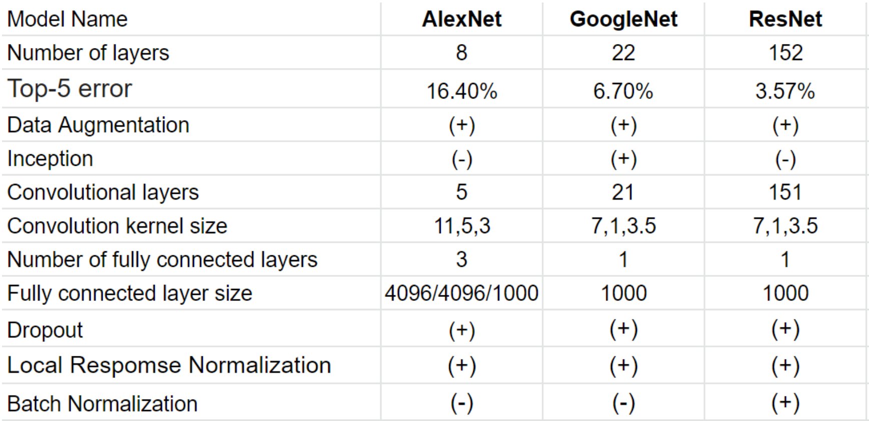
Figure 3.11: When we add more hidden layers between the first hidden layer added and the final output layer in the models of "Cars" dataset. There will be better convergence of data and trends.

Here we increase the unit in the added hidden layer to 50 and use the **sigmoid** function to activate. The most important new thing about these hidden layers is that they introduce a non-linear activation function, in this case sigmoid activation. And the test results are as follows.

In this Figure3.11, we can clearly see that the fitting curve between horsepower and MPG becomes more smoother, and the performance and effect have become better than before. It turns out that when we add more hidden layers and choose a suitable activation function, it will have a profound impact on the performance of our machine learning in the browser.

For our MNIST dataset, we want to deploy several classic CNN models(such as **AlexNet, GoogLeNet, and ResNet**)to compare their respective performance.
According to the comparison(Figure3.12), we found that ResNet performed the best,with top-5 error of only 3.57%. ResNet has more layers, deeper convolution layers and larger convolution kernels than AlexNet and GoogleNet. At the same time,the number of its fully connected layers is more simplified and the size is more appropriate. And more valuable is that ResNet has both Local Response Normalization and Batch Normalization.
So let's take a look at the performance of these networks on **Cifar-10** data (Figure3.13). When we train with the **LeNet** model for 100 epochs, the accuracy can reach at 66%. When we train with the **AlexNet** model for 100 epochs, the effect is really improved. Finally, the performance of the **ResNet** is the best, basically reached at 80%.

Therefore, it is not difficult to see that in the classic CNN heavyweight models,



Model Name	**AlexNet**	**GoogleNet**	**ResNet**
Number of layers	8	22	152
Top-5 error	16.40%	6.70%	3.57%
Data Augmentation	(+)	(+)	(+)
Inception	(-)	(+)	(-)
Convolutional layers	5	21	151
Convolution kernel size	11,5,3	7,1,3.5	7,1,3.5
Number of fully connected layers	3	1	1
Fully connected layer size	4096/4096/1000	1000	1000
Dropout	(+)	(+)	(+)
Local Respomse Normalization	(+)	(+)	(+)
Batch Normalization	(-)	(-)	(+)

Figure 3.12: Comparison of three models parameters from shallow to deep, among them ResNet ranks first with the least Top-5 error rate and deeper layers structure.

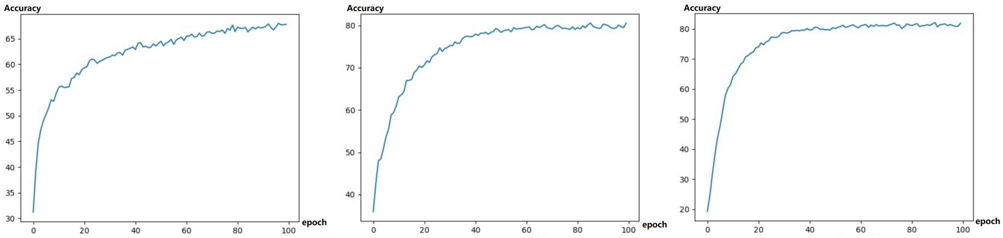


Figure 3.13: From left to right, LeNet, AlexNet, and ResNet deploy on the Cifar-10 model in turn. After completing 100 epochs, ResNet reached the highest accuracy rate of 80%

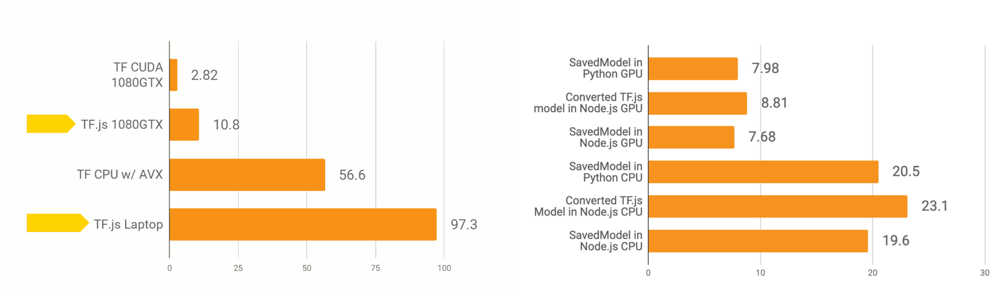


Figure 3.14: The performance of MobileNet in different devices and different environments.We can see that the performance on the mobile side is more prominent.

ResNet has a deeper number of layers and better performance. It is a brilliant model we can choose and trust.
After discussing the classic heavyweight models, the lightweight models such as **MobileNet** in recent years have become more and more popular. But how effective is the lightweight model on mobile and web side? According to Google Cloud data[Goo], we can make an intuitive evaluation based on **MobileNet**,which can be used as a reference for us.
Here we will use the same model to run 200 times in python through tensorflow and run in the browser with javascript through the tensorflow.js(Figure3.14).
In the browser, TensorFlow.js can use WebGL for hardware acceleration and use GPU resources. When we run an inference in the browser with tensorflow.js:
It takes 97ms on the CPU
It takes 10ms on GPU (WebGL)
Compared with the Python code benchmark, the running time of TensorFlow.js in the browser is 1.7 times that of the CPU, and the running time of the GPU (WebGL) is 3.8 times.
In Node.js, we can load the converted model with JavaScript through tensorflw.js, or use C++ Binding of the TensorFlow, which approach and surpass the performance of Python respectively.
When we run an inference in the browser on Node.js through tensorflow.js:
Running native model time on CPU is 19.6ms
Running native model time on GPU (CUDA) is 7.68ms
Compared with the Python code benchmark, it runs 4% faster than the benchmark on both CPU and GPU.

4 Conclusion

In this thesis we evaluated different models for different tasks. This makes it possible to select the appropriate model for a mobile web-based application. The results of this thesis allow to consider the accuracy of the model as well as its runtime with and without GPU.

Machine learning inn the browser has gradually become a trend. Using the browser for machine learning has unparalleled advantages. Because compared to traditional local deployment, accessing web pages through a browser has a lower threshold and a wider spread. When we only use local CPU/GPU resources to perform the machine learning we gain more flexible useage of AI applications in the browser: We do not need to install unnecessary software or drivers; More convenient human-computer interaction can be carried out through the browser; We can call various sensors of the phone hardware (such as GPS, electronic compass, acceleration sensor, camera, etc.) through the phone browser; Our user data can be completed locally without uploading to the server.

Internet is a very powerful medium. It is cross-platform and can be used with a variety of different devices, from mobile devices, tablets to desktop devices, and different operating systems (Android, iOS and Mac, Windows, etc.), with only one link. Different from common applications, it does not require any installation process nor complex configurations.

When we deploy the machine learning model in the browser, it is particularly important to configure the appropriate neural network parameters and select the appropriate model. Among the heavyweight models, **ResNet** is unique with its excellent performance. However, compared to the flexibility and speed of the lightweight model, **MobileNet** is more suitable for our deployment in the browser due to the low ressource consumption and low runtime.

Bibliography

[AAB$^+$17] Nassim Ammour, Haikel Alhichri, Yakoub Bazi, Bilel Benjdira, Naif Alajlan, and Mansour Zuair. Deep learning approach for car detection in uav imagery. *Remote Sensing*, 9(4):312, 2017.

[ABC$^+$16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[AKH12] I. Sutskever A. Krizhevsky and G. Hinton. Imagenet classification with deep convolutional neural networks. 2012.

[BBLP12] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks, 2012.

[BFG$^+$16] H. Bahmani, W. Fuhl, E. Gutierrez, G. Kasneci, E. Kasneci, and S. Wahl. Feature-based attentional influences on the accommodation response. In *Vision Sciences Society Annual Meeting Abstract*, 2016.

[CS15] Yangqing Jia Pierre Sermanet Scott E. Reed Dragomir Anguelov Dumitru Erhan Vincent Vanhoucke Andrew Rabinovich Christian Szegedy, Wei Liu. Going deeper with convolutions. 2015.

[Dev16] Desale. Devendra. Top 15 frameworks for machine learning experts. 2016.

[Duc11] Hazan E. Singer Y. Duchi, J. Adaptive subgradient methods for online learning and stochastic optimization. 2011.

[EFK17] Shahram Eivazi, Wolfgang Fuhl, and Enkelejda Kasneci. Towards intelligent surgical microscopes: Surgeons gaze and instrument tracking. In *Proceedings of the 22st International Conference on Intelligent User Interfaces, IUI 2017*. ACM, 03 2017.

[EHF$^+$17] S. Eivazi, A. Hafez, W. Fuhl, H. Afkari, E. Kasneci, M. Lehecka, and R. Bednarik. Optimal eye movement strategies: a comparison of neurosurgeons gaze patterns when using a surgical microscope. *Acta Neurochirurgica*, 2017.

[ESF$^+$17] Shahram Eivazi, Michael Slupina, Wolfgang Fuhl, Hoorieh Afkari, Ahmad Hafez, and Enkelejda Kasneci. Towards automatic skill evaluation in microsurgery. In *Proceedings of the 22st International Conference on Intelligent User Interfaces, IUI 2017*. ACM, 03 2017.

[FBH$^+$19] Wolfgang Fuhl, Efe Bozkir, Benedikt Hosp, Nora Castner, David Geisler, Thiago C., and Enkelejda Kasneci. Encodji: Encoding gaze data into emoji space for an amusing scanpath classification approach ;). In *Eye Tracking Research and Applications*, 2019.

[FCK18a] W. Fuhl, N. Castner, and E. Kasneci. Histogram of oriented velocities for eye movement detection. In *International Conference on Multimodal Interaction Workshops, ICMIW*, 2018.

[FCK18b] W. Fuhl, N. Castner, and E. Kasneci. Rule based learning for eye movement type detection. In *International Conference on Multimodal Interaction Workshops, ICMIW*, 2018.

[FCK$^+$19] W. Fuhl, N. Castner, T. C. Kübler, A. Lotz, W. Rosenstiel, and E. Kasneci. Ferns for area of interest free scanpath classification. In *Proceedings of the 2019 ACM Symposium on Eye Tracking Research & Applications (ETRA)*, 06 2019.

[FCZ$^+$18] W. Fuhl, N. Castner, L. Zhuang, M. Holzer, W. Rosenstiel, and E. Kasneci. Mam: Transfer learning for fully automatic video annotation and specialized detector creation. In *International Conference on Computer Vision Workshops, ICCVW*, 2018.

[FEH$^+$18] W. Fuhl, S. Eivazi, B. Hosp, A. Eivazi, W. Rosenstiel, and E. Kasneci. Bore: Boosted-oriented edge optimization for robust, real time remote pupil center detection. In *Eye Tracking Research and Applications, ETRA*, 2018.

[FGK20a] W. Fuhl, H. Gao, and E. Kasneci. Neural networks for optical vector and eye ball parameter estimation. In *ACM Symposium on Eye Tracking Research & Applications, ETRA 2020*. ACM, 01 2020.

[FGK20b] W. Fuhl, H. Gao, and E. Kasneci. Tiny convolution, decision tree, and binary neuronal networks for robust and real time pupil outline estimation. In *ACM Symposium on Eye Tracking Research & Applications, ETRA 2020*. ACM, 01 2020.

[FGRK19] W. Fuhl, D. Geisler, W. Rosenstiel, and E. Kasneci. The applicability of cycle gans for pupil and eyelid segmentation, data generation and image refinement. In *International Conference on Computer Vision Workshops, ICCVW*, 11 2019.

[FGS$^+$18] W. Fuhl, D. Geisler, T. Santini, T. Appel, W. Rosenstiel, and E. Kasneci. Cbf:circular binary features for robust and real-time pupil center detection. In *ACM Symposium on Eye Tracking Research & Applications*, 06 2018.

[FK18] W. Fuhl and E. Kasneci. Eye movement velocity and gaze data generator for evaluation, robustness testing and assess of eye tracking software and visualization tools. In *Poster at Egocentric Perception, Interaction and Computing, EPIC*, 2018.

[FK19] W. Fuhl and E. Kasneci. Learning to validate the quality of detected landmarks. In *International Conference on Machine Vision, ICMV*, 11 2019.

[FKB+18] W. Fuhl, T. C. Kübler, H. Brinkmann, R. Rosenberg, W. Rosenstiel, and E. Kasneci. Region of interest generation algorithms for eye tracking data. In *Third Workshop on Eye Tracking and Visualization (ETVIS), in conjunction with ACM ETRA*, 06 2018.

[FKH+17] W. Fuhl, T. C. Kübler, D. Hospach, O. Bringmann, W. Rosenstiel, and E. Kasneci. Ways of improving the precision of eye tracking data: Controlling the influence of dirt and dust on pupil detection. *Journal of Eye Movement Research*, 10(3), 05 2017.

[FKRK20] W. Fuhl, G. Kasneci, W. Rosenstiel, and E. Kasneci. Training decision trees as replacement for convolution layers. In *Conference on Artificial Intelligence, AAAI*, 02 2020.

[FKS+15a] W. Fuhl, T. C. Kübler, K. Sippel, W. Rosenstiel, and E. Kasneci. Arbitrarily shaped areas of interest based on gaze density gradient. In *European Conference on Eye Movements, ECEM 2015*, 08 2015.

[FKS+15b] W. Fuhl, T. C. Kübler, K. Sippel, W. Rosenstiel, and E. Kasneci. Excuse: Robust pupil detection in real-world scenarios. In *16th International Conference on Computer Analysis of Images and Patterns (CAIP 2015)*, 09 2015.

[FKSK18] W. Fuhl, T. Kübler, T. Santini, and E. Kasneci. Automatic generation of saliency-based areas of interest. In *Symposium on Vision, Modeling and Visualization (VMV)*, 09 2018.

[FRE20] Wolfgang Fuhl, Yao Rong, and Kasneci Enkelejda. Fully convolutional neural networks for raw eye tracking data segmentation, generation, and reconstruction. In *Proceedings of the International Conference on Pattern Recognition*, pages 0–0, 2020.

[FRK19] W. Fuhl, W. Rosenstiel, and E. Kasneci. 500,000 images closer to eyelid and pupil segmentation. In *Computer Analysis of Images and Patterns, CAIP*, 11 2019.

[FRM+20] Wolfgang Fuhl, Yao Rong, Thomas Motz, Michael Scheidt, Andreas Hartel, Andreas Koch, and Enkelejda Kasneci. Explainable online validation of machine learning models for practical applications. *arXiv*, 08 2020.

[FSG$^+$16] W. Fuhl, T. Santini, D. Geisler, T. C. Kübler, W. Rosenstiel, and E. Kasneci. Eyes wide open? eyelid location and eye aperture estimation for pervasive eye tracking in real-world scenarios. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct publication – PETMEI 2016*, 09 2016.

[FSG$^+$17] W. Fuhl, T. Santini, D. Geisler, T. C. Kübler, and E. Kasneci. Eyelad: Remote eye tracking image labeling tool. In *12th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2017)*, 02 2017.

[FSK17a] W. Fuhl, T. Santini, and E. Kasneci. Fast and robust eyelid outline and aperture detection in real-world scenarios. In *IEEE Winter Conference on Applications of Computer Vision (WACV 2017)*, 03 2017.

[FSK17b] W. Fuhl, T. Santini, and E. Kasneci. Fast camera focus estimation for gaze-based focus control. In *CoRR*, 2017.

[FSK$^+$18] W. Fuhl, T. Santini, T. Kuebler, N. Castner, W. Rosenstiel, and E. Kasneci. Eye movement simulation and detector creation to reduce laborious parameter adjustments. *arXiv preprint arXiv:1804.00970*, 2018.

[FSKK16] W. Fuhl, T. Santini, T. C. Kübler, and E. Kasneci. Else: Ellipse selection for robust pupil detection in real-world environments. In *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research & Applications (ETRA)*, pages 123–130, 03 2016.

[FSR$^+$16] W. Fuhl, T. Santini, C. Reichert, D. Claus, A. Herkommer, H. Bahmani, K. Rifai, S. Wahl, and E. Kasneci. Non-intrusive practitioner pupil detection for unmodified microscope oculars. *Elsevier Computers in Biology and Medicine*, 79:36–44, 12 2016.

[FTBK16] Wolfgang Fuhl, Marc Tonsen, Andreas Bulling, and Enkelejda Kasneci. Pupil detection for head-mounted eye tracking in the wild: An evaluation of the state of the art. In *Machine Vision and Applications*, pages 1–14, 06 2016.

[Fuh19] W. Fuhl. *Image-based extraction of eye features for robust eye tracking*. PhD thesis, University of Tübingen, 04 2019.

[Fuh20] Wolfgang Fuhl. From perception to action using observed actions to learn gestures. *User Modeling and User-Adapted Interaction*, pages 1–18, 08 2020.

[GFSK17] D. Geisler, W. Fuhl, T. Santini, and E. Kasneci. Saliency sandbox: Bottom-up saliency framework. In *12th Joint Conference on Computer Vision,* *Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2017)*, 02 2017.

[Goo] Google. Cloud data.

[HCE$^+$17] Shawn Hershey, Sourish Chaudhuri, Daniel PW Ellis, Jort F Gemmeke, Aren Jansen, R Channing Moore, Manoj Plakal, Devin Platt, Rif A Saurous, Bryan Seybold, et al. Cnn architectures for large-scale audio classification. In *2017 ieee international conference on acoustics, speech and signal processing (icassp)*, pages 131–135. IEEE, 2017.

[Heu17] Ramsauer H. Unterthiner T. Nessler B. Hochreiter S Heusel, M. Gans appendix a. reference. 2017.

[HL09] R. Ranganath A. Y. Ng H. Lee, R. Grosse. Convolutional deeb belief networks for scalable unsupervised learning of hierarchical representation. 2009.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[Kin15] Ba J. L. Kingma, D. P. Adam: a method for stochastic optimization. 2015.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LHBB99] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.

[MMR$^+$16] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. 2016.

[MWL16] Shun Miao, Z Jane Wang, and Rui Liao. A cnn regression approach for real-time 2d/3d registration. *IEEE transactions on medical imaging*, 35(5):1352–1363, 2016.

[Nes83] Y Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). doklady ansssr (translated as soviet.math.docl.), vol. 269, pp. 543-547. 1983.

[NOF20] Anna Nguyen, Adrian Oberföll, and Michael Färber. Right for the right reason: Making image classification robust, 2020.

[Ola14] C Olah. Neural networks, manifolds, and topology. 2014.

[PSL+16] Alexandra Papoutsaki, Patsorn Sangkloy, James Laskey, Nediyana Daskalova, Jeff Huang, and James Hays. Webgazer: Scalable webcam eye tracking using user interactions. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence-IJCAI 2016*, 2016.

[Qia] N. Qian. On the momentum term in gradient descent learning algorithms.

[RD83] Ernesto Ramos and David Donoho. 1983.

[Rru16] Sebastian Rruder. An overview of gradient descent optimization algorithms. 2016.

[SA08] Sanglee Park Heerin Yang Jungmin So Sanghyeon An, Minjun Lee. An ensemble of simple convolutional neural network models for mnist digit recognition. 2008.

[SHZ+18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[Sut86] R. S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. 1986.

[SW] Carter S. Sculley D. Viegas F. Smilkov, D. and M Wattenberg. Direct-manipulation visualization of deep networks.

[XAJ+18] Lele Xie, Tasweer Ahmad, Lianwen Jin, Yuliang Liu, and Sheng Zhang. A new cnn-based method for multi-directional car license plate detection. *IEEE Transactions on Intelligent Transportation Systems*, 19(2):507–517, 2018.

[YLJ89] J. S. Denker D. Henderson R. E. Howard W. Hubbard Y. LeCun, B. Boser and L. D. Jackel. "backpropagation applied to handwritten zip code recognition". 1989.