TensorFlow.js and Brain.js – Project report

Abstract – This project delves into the possibilities of browser-based image recognition and object detection by utilizing cutting-edge frameworks such as TensorFlow.js, Brain.js, and MediaPipe's models for hand recognition. The primary focus is on precise hand detection, extending pretrained models through transfer learning to explore the potential of developing a specialized model for recognizing Macedonian sign language. This research highlights the growing accessibility and versatility of machine learning technologies in real-time, web-based applications.

Introduction

Image recognition is the process of identifying an object or a feature in an image or video. It is used in many applications like defect detection, medical imaging, and security surveillance. This is a crucial technique that can find usage in many applications, and it is the main component in deep learning applications such as Visual Inspection, Robotics, Automated Driving, Image Classification and so on.

Image recognition is a technique that usually goes hand in hand with Object detection. There are similarities between the two, but Object detection is a technique that uses neural networks to localize and classify objects in images. The key difference is that Image recognition focuses on classifying the entire image, while Object detection identifies and locates multiple objects within an image.

One widely used technique in image recognition is deep learning, which leverages convolutional neural networks (CNNs) to automatically learn and extract key features from large sets of sample images. By training on these features, CNNs enable the system to accurately identify and classify similar patterns in new images. The typical workflow starts with gathering and preparing training data, followed by creating and training the deep learning model, and finally testing the model with new data.

How do CNNs work?

Let's start from the beginning: what exactly is a CNN? A convolutional neural network (or CNN for short) is a category of machine learning model, namely a type of deep learning algorithm well suited to analyzing visual data. By using convolution operations from linear algebra, CNNs extract features and identify patterns in images. While primarily used for image processing, CNNs can also be adapted for audio and other types of signal data.

The architecture of CNNs is inspired by the connectivity patterns found in the human brain, particularly the visual cortex, which is crucial for processing visual stimuli. In a CNN, artificial neurons are arranged to effectively interpret visual information, allowing the model to analyze entire images efficiently.

A CNN typically consists of several layers each of which detects different features of an input image. These layers can be broadly categorized into three groups: convolutional layers, pooling layers and fully connected layers. As data passes through these layers, the complexity of the CNN increases, which let's

the CNN successively identify larger portions of an image and more abstract features.

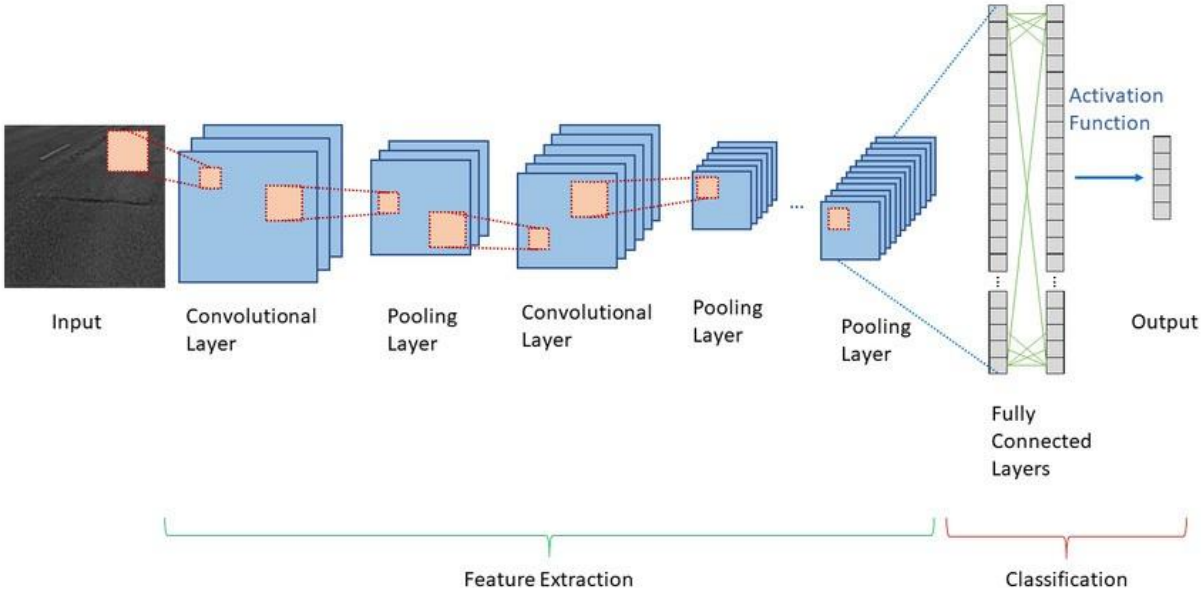

Figure 1: Architecture of a CNN [17]

The **convolutional layer** is the core component of a CNN, where most of the computation happens. It uses a filter or kernel, a small matrix of weights, to slide over an input image and detect specific features. This involves computing the dot product between the kernel and the image pixels to produce feature maps, which indicate the presence and intensity of features at various locations.

CNNs typically stack multiple convolutional layers to progressively interpret visual information. Early layers detect basic features like edges and textures, while deeper layers build on these to identify more complex patterns and objects.

The **pooling layer** is also a crucial component of a CNN with a different functionality and just like the convolutional layer, it involves a sweeping process across the input image. The pooling layer aims to reduce dimensionality of the input data while retaining critical information, thus improving the network's overall efficiency and generalization ability. This is typically achieved through downsampling: decreasing the number of data points in the input (or in our case: reducing the number of pixels used to represent the image).

The most common type of pooling is max pooling, which retains the maximum value within a certain window (i.e. the kernel size) while discarding other values. Another common technique is average pooling. This type of pooling takes a similar approach, but uses the average value instead of the maximum. Other types of pooling include: global and min pooling.

The **fully connected layer** is used in the final stages of a CNN, where it is responsible for classifying images based on the features extracted in the previous layers. This means it maps the features to specific classes or outcomes. The term *fully connected* means that each neuron in one layer is connected to each neuron in the subsequent layer. Each input from the previous layer connects to each activation unit in the fully connected layer, enabling the CNN to simultaneously consider all features when making a final classification decision.

Not all CNN layers are fully connected. Using fully connected layers throughout the network can lead to excessive parameter density, increased overfitting risk, and higher memory and computation costs. By limiting these layers, the network achieves a balance between computational efficiency, generalization, and the ability to learn complex patterns.

Machine Learning in the browser

As Machine Learning continues to evolve, its complexity increases, highlighting the importance of frameworks and libraries that simplify browser-based development. Historically, successful technologies have relied on such frameworks for efficient implementation. Mastering machine learning frameworks designed for the browser not only accelerates development but also enhances the efficiency of integrating ML models into web applications.

There are many good options out there when it comes to Machine Learning frameworks, but the ones we decided to focus on for this project are Brain.js and TensorFlow.js.

Brain.js

Brain.js is a popular open-source JavaScript library for building neural networks. It is designed to be easy to use and provides a simple API for creating and training various types of neural networks, including feedforward networks, recurrent networks, and convolutional networks. It can be used for build machine learning models that can be used for a variety of tasks, such as text classification, image recognition, and time series analysis. The library is written entirely in JavaScript and runs in the browser as well as on the server-side using Node.js.

Some of the key features of Brain.js are support for multiple types of neural networks, simple API, support for CPU and GPU training, ability to save and load trained models and support for transfer learning. In comparison to other alternatives, it has its own strengths and weaknesses. Here is a quick overview:

Brain.js is a lightweight neural network library for JavaScript, designed for simplicity and ease of use. It is suitable for quick prototyping and smaller machine learning tasks.

Some of its strengths include:

1. **Simplicity:** Brain.js is easy to set up and use, making it accessible for beginners and those looking to quickly prototype neural networks.
2. **Ease of Integration:** It integrates well with other JavaScript projects and can be run both in the browser and on the server with Node.js.
3. **Visualization:** Brain.js provides straightforward tools for visualizing neural networks and their outputs.

Of course, Brain.js also has its limitations:

1. **Performance:** Brain.js is not optimized for large-scale machine learning tasks or deep learning models. It lacks the advanced optimizations and performance benefits seen in more comprehensive libraries.

2. **Limited Functionality:** It supports basic neural networks but lacks the extensive features and flexibility offered by more advanced libraries like TensorFlow.js.

Installation

When working with a Node.js application Brain.js can be installed into your project using one of two methods:

1. Installing with NPM

```
npm install brain.js
```

Make sure the following dependencies are installed and up to date then run:

```
npm rebuild
```

2. Serve over CDN

```
<script src="//unpkg.com/brain.js"></script>
```

As you can see, Brain.js is an excellent choice for implementing neural networks directly in the browser. Its simplicity and ability to run models client-side make it ideal for integrating machine learning into web applications efficiently, handling tasks like pattern recognition and predictions with ease in a JavaScript environment. Brain.js is a great starting point for beginners and smaller projects due to its simplicity and ease of use. It excels in scenarios where quick setup and straightforward implementation are more important than advanced features.

TensorFlow.js

TensorFlow.js is a powerful library that enables the development and execution of machine learning algorithms directly within JavaScript. It allows machine learning models to run both in the browser and in the Node.js environment, making it highly versatile for web and server-side applications. As part of the broader TensorFlow ecosystem, TensorFlow.js provides a robust set of APIs that are compatible with Python's TensorFlow, allowing seamless portability of models between the Python and JavaScript ecosystems. This has significantly expanded the reach of machine learning, empowering a vast community of JavaScript developers to build, train, and deploy sophisticated models with ease. Moreover, TensorFlow.js has opened up new possibilities for on-device computation, providing real-time machine learning capabilities that can be leveraged in web applications without the need for server-side processing.

Tensorflow.js is not designed to train large models, but rather it is built for inference. Machine learning models trained in Python can be loaded into Tensorflow.js and used for inference in the user's browser.

Training machine learning models can require a large amount of computational resources. However, once a model has been trained, it can be reused multiple times for inference without requiring as much computation.

To put it simply: TensorFlow.js is a more comprehensive and powerful library for machine learning in JavaScript, developed by Google. It allows you to develop and execute ML models in the browser and Node.js, leveraging TensorFlow’s powerful ecosystem. Some of its strengths include:

1. **Performance:** TensorFlow.js is optimized for high-performance machine learning tasks, including GPU acceleration (via WebGL) for computationally intensive operations.
2. **Flexibility:** It supports a wide range of machine learning models, from simple neural networks to complex deep learning architectures.
3. **Pre-trained Models:** TensorFlow.js offers access to numerous pre-trained models that can be easily integrated and fine-tuned for specific tasks.
4. **Extensive Ecosystem:** It benefits from the broader TensorFlow ecosystem, including tools for model training, deployment, and visualization.

Like any technology, TensorFlow.js has its limitations. Here are a few to consider:

1. **Complexity:** TensorFlow.js has a steeper learning curve compared to Brain.js. Its extensive functionality can be overwhelming for beginners.
2. **Size:** TensorFlow.js is a larger library, which can impact load times and resource usage in browser-based applications.

The Tensor class

One of the core features of TensorFlow.js is its Tensor class, which represents the main data structure used for computation. A tensor is essentially a multi-dimensional array, capable of storing numerical data across one or more dimensions. For example, a 1D tensor functions like a vector, while a 2D tensor resembles a matrix. Tensors can also extend to higher dimensions, making them ideal for complex data structures like images, videos, or time-series data. Importantly, tensors in TensorFlow.js are immutable, meaning their values cannot be changed once created. Instead, new tensors are generated when applying transformations, ensuring stability and consistency in machine learning operations.

TensorFlow.js offers a variety of methods to manipulate and work with tensors, such as reshaping, slicing, and broadcasting to align data for computation. Additionally, it supports a wide range of mathematical functions, from basic operations like addition and multiplication to more advanced linear algebra. These methods are optimized for performance and can leverage WebGL for GPU acceleration, making TensorFlow.js an effective tool for developing machine learning models directly in the browser. Tensors, therefore, play a crucial role in enabling efficient data representation and computation, forming the backbone of all machine learning processes in TensorFlow.js.

Installation

Same as with Brain.js, there are multiple ways to set up TensorFlow.js in your project.

1. Installing with NPM

```
npm install @tensorflow/tfjs
```

2. Serve over CDN

```
<script
src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest/dist/tf.min.js"
></script>
```

In conclusion, TensorFlow.js is a powerful and flexible library suited for more complex and performance-critical machine learning tasks. It offers a rich set of features and access to the extensive TensorFlow ecosystem, making it ideal for production-level applications and advanced machine learning projects.

Documentation and examples

In this project the possibilities of machine learning in the browser are explored through several examples about hand pose detection and recognition. For the development of this project, **React** was chosen as the primary framework for building the user interface. Additionally, **npm** (Node Package Manager) and **Node.js** were utilized to manage dependencies and facilitate the development environment.

As mentioned, the project includes **multiple code examples**, each **organized into separate branches** within the Git repository. These branches represent distinct features of the project, allowing for isolated development and easier management of different aspects. Each branch includes specific examples and implementations showcasing various functionalities, which will be examined in detail in subsequent sections.

Transfer learning

One of the main focuses of this project is Transfer learning. Transfer learning is a machine learning technique in which knowledge gained through one task or dataset is used to improve model performance on another related task and/or different dataset. In other words, transfer learning uses what has been learned in one setting to improve generalization in another setting. We humans do this all the time. We have a lifetime of experiences contained in our brains that we can use to help recognize new things we have never seen before.

There are many advantages and reasons why transfer learning should be used. Transfer learning reduces computational costs by allowing users to repurpose pretrained models for new tasks, thereby cutting down on training time, data requirements, and processor resources. This approach can significantly shorten the training process, often needing fewer epochs to reach optimal learning. It also addresses the challenge of acquiring large datasets, as it leverages existing models to perform well even with limited new data. Additionally, transfer learning enhances a model's generalizability by incorporating knowledge from multiple datasets, which helps improve performance on diverse data and reduces the risk of overfitting.

This project utilizes the advantages of transfer learning by leveraging **pre-trained models** from **TensorFlow.js** and **MediaPipe** as the foundation for the examples.

Gesture recognition

The initial example in this project can be found on the **feature/handpose-detection** branch, serving as a starting point for our research. This example was build using guidance from Nicholas Renotte's two tutorials (following his videos about Real time AI Hand pose estimation and Real time AI Gesture recognition). These resources provided valuable insights into TensorFlow.js and established a solid groundwork for developing our own examples.

In this example the **handpose model** from TensorFlow.js is used. It can detect and track hand gestures in real-time using a webcam or other video input. It can also identify the position of key points on the hand, such as fingertips and joints, making it useful for applications involving gesture recognition and hand interaction.

The model uses a convolutional neural network (CNN) to analyze input images or video frames. It processes these frames to predict the positions of 21 key points on each hand, which correspond to various parts of the hand, including fingers and joints. The predictions are used to generate a hand mesh or keypoint coordinates, which can then be used to infer gestures or interact with applications.

The handpose model is pre-trained on a large dataset of hand images, which allows it to accurately detect and track hand key points out of the box. This pre-training means that it can be used for real-time hand pose detection without needing to be train from scratch.

The **fingerpose library** is also used in the Gesture recognition example. It is a library designed for recognizing hand gestures using hand landmark data. It provides tools to define and recognize custom hand gestures based on key points detected by models like TensorFlow.js's handpose. Its only constraint is that it is optimized for single hand detections.

In the code example this model is used to recognize two gestures: the thumbs up and victory gesture. The fingerpose library is used to recognize these specific gestures, and these recognized gestures are mapped to emojis which are then displayed on the screen.

The drawHand function

The drawHand function is used in all code examples in the project to visualize the detected hands and it is adjusted appropriately to the hand detecting model being used. It uses the <canvas> element to visualize hand landmarks and their connections. It maps each finger to its corresponding joints and draws lines between these joints to represent the finger's shape. Additionally, it plots circles at each landmark's position to highlight them. The drawing is done using the canvas's 2D context, with lines and points styled for clear visualization.

Macedonian sign language recognition examples

Building on insights gained from the previous example, we proceeded to develop two additional examples. Both examples are designed to collect samples and train models for recognizing Macedonian sign language. One model is focused on recognizing signs made with one hand, while the other is trained to recognize signs made with both hands. It's important to note that these models can be adapted for various classification tasks beyond the ones explored here, which were selected as initial use cases for development.

Due to the challenge of collecting classified samples, both code examples share a common foundation. In both instances, the user interface includes five buttons, each assigned to collect labeled samples corresponding to the first five letters of the Macedonian alphabet: 'А', 'Б', 'В', 'Г', and 'Д'. We opted to train the model on only these five letters to avoid extending the training phase significantly. When a button is clicked, a frame from the webcam is processed and saved in a corresponding format as a labeled training sample. Additionally, there is a button that initiates the training and prediction phases. Once training is complete, the webcam stream is used to classify hand gestures in real time. It is crucial

to remember that this button should not be clicked repeatedly, as doing so may disrupt the ongoing training process and lead to issues.

One handed Macedonian sign language recognition model

In this research project, the one-handed Macedonian sign language recognition model can be found on the branch named **feature/one-hand-msl-detection**. This model leverages TensorFlow.js and the **HandPose model** as its foundational base. By incorporating **Brain.js** and applying transfer learning techniques, the model is trained to recognize the first five letters from the one-handed Macedonian sign language alphabet. This approach allows for the adaptation of pre-existing hand gesture recognition capabilities to specific, user-defined gestures.

The code example begins by initializing a neural network model using the **NeuralNetwork** class from Brain.js, which features a single hidden layer comprising 128 neurons. The hidden layer uses ReLU (Rectified Linear Unit) as the activation function introducing non-linearity to capture complex patterns. The output layer, which corresponds to the number of signs (classes), employs the softmax activation function to generate a probability distribution across these classes. For hand landmark detection, it integrates the handpose model from TensorFlow.js, which provides the capability to identify and analyze hand positions and gestures through the webcam.

Figure 2: Macedonian Sign Language Alphabet [18]

In the data collection phase, the gatherDataForClass function captures hand landmarks from the webcam video feed for specified signs, such as "А", "Б", "В", "Г", and "Д". This process runs over a defined duration, during which landmarks are converted into input arrays and recorded as training samples with associated gesture labels. The data collection is performed in intervals and continues until the specified duration is reached, ensuring a comprehensive dataset for model training.

Once data collection is complete, the application transitions to hand detection and prediction. The detectHand function operates continuously to detect hand landmarks and make predictions if the model is trained. By utilizing the neural network, it classifies the detected gestures based on the input landmarks, with the results logged to the console for evaluation.

Training the model is handled by the handleTrainModel function, which employs the collected training data to train the neural network. The training process is configured with parameters such as iterations and logging details to monitor progress. Following successful training, the application initiates the prediction process to begin classifying gestures in real-time.

To ensure the smooth operation of the application, the checkWebcamReady function verifies that the webcam feed is properly initialized before starting the detection and training phases. This function periodically checks the webcam status and initializes the handpose model only after confirming that the webcam is operational.

The user interface of the React component, as mentioned before, includes interactive buttons for initiating data collection for specific signs and for training the model. Additionally, status updates are displayed to keep the user informed about ongoing processes and any issues that may arise.

Two handed Macedonian sign language recognition model

The code for this model is on the branch named **feature/two-hand-msl-detection**. The gestures for the two-handed signs were learned from the excellent resource Talking hands, which is a website dedicated to teaching Macedonian Sign Language. These are the signs that will be sampled and utilized for making predictions with the model.

In this example TensorFlow.js is used to load a pre-trained base model, specifically **MobileNet**, to generate image features that can be used in transfer learning, and then to create and define a multi-layer perceptron that takes the image features and learns to classify new objects using them. On the other hand, this example uses the **HandLandmarker model** from **MediaPipe** to detect the landmarks on both hands.

MobileNet is a lightweight deep learning model developed by Google, designed specifically for mobile and edge devices. It is built on an efficient architecture that uses depthwise separable convolutions to reduce computational demands and memory usage. MobileNet is pre-trained on the ImageNet dataset, which contains millions of labeled images across thousands of categories. This pre-training allows MobileNet to recognize a wide variety of objects and features in images. Users can leverage MobileNet for transfer learning, adapting the model to new tasks by fine-tuning it on additional datasets or specific use cases.

The MediaPipe HandLandmarker model is a real-time computer vision tool from Google's MediaPipe framework, designed to detect and track 21 key hand landmarks with high accuracy. It operates efficiently on both images and video streams, making it ideal for interactive applications like gesture recognition. MediaPipe itself is an open-source framework that offers a range of pre-trained models and tools for various computer vision tasks, optimized for real-time performance across multiple platforms, including web and mobile.

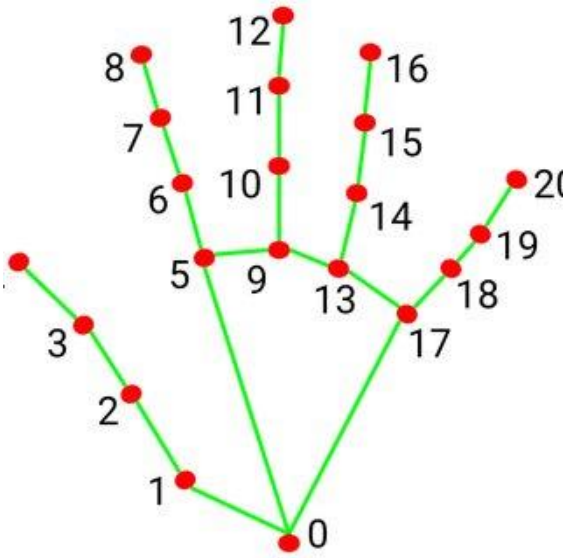

Figure 3: Hand landmarks [19]

In the code example the MediaPipe model is loaded and warmed up in the function loadMobileNetFeatureModel. With larger models like this, the first time it is used, it can take a moment to set everything up. Therefor it helps to pass zeros through the model to avoid any waiting in the future where timing may be more critical.

On the other hand, the model, that will be trained, is structured as a sequential model, which allows us to add layers sequentially. We start by adding a dense input layer with an input shape of 1024, corresponding to the output size from the MobileNet v3 feature extraction. This layer includes 128 neurons and uses ReLU as its activation function. Following the input layer, we add an output layer. The number of neurons in this layer matches the number of classes we intend to predict, which is 5 in our case. We use the softmax activation function in this output layer, as it is commonly used in classification

problems to provide probabilities for each class. When compiling the model, the Adam optimizer is used, accuracy is set as the evaluation metric, and categorical crossentropy is chosen as the loss function.

To prepare the output data for training, the output array is converted to a 1D tensor of type int32, which is then used for one-hot encoding. The tf.oneHot() function is applied with the number of classes to produce a one-hot encoded tensor. The input array of tensors is then transformed into a 2D tensor using tf.stack(), which stacks the tensors to form a batch suitable for training.

The model is trained with model.fit(), using the prepared input and one-hot encoded output tensors. Configuration parameters include shuffling the data, setting the batch size, specifying the number of epochs, and defining a callback function for logging training progress. After training, the temporary tensors are disposed of. Prediction functionality is re-enabled, and live predictions are initiated with the prediction loop function. A logging function is also defined to track and print training progress.

Conclusion

In conclusion, the field of machine learning (ML) is rapidly expanding, with increasing applications of these technologies within web browsers. This trend highlights the growing potential and versatility of ML in accessible environments. A particularly significant application is in the realm of sign language translation, where ML models can facilitate real-time interpretation and enhance communication for the deaf and hard-of-hearing communities. As browser-based ML technologies continue to evolve, they promise to provide even more resources and tools for developing and refining sign language translation systems, thereby broadening their impact and accessibility.

References and resources

1. MathWorks. *Image Recognition*. https://www.mathworks.com/discovery/image-recognition-matlab.html [1]
2. Lev Craig. TechTarget. (Jan. 2024). *Convolutional Neural Network (CNN)*. https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network [1, 2, 3]
3. Scribbler. (Apr.30, 2023). *Using Brain.js for Machine Learning*. https://scribbler.live/2023/04/30/Machine-Learning-in-JavaScript-using-Brainjs.html#top [3, 4, 5, 6]
4. Obinna Ekwuno. LogRocket. (Nov.15, 2019). * An introduction to deep learning with Brain.js *. https://blog.logrocket.com/an-introduction-to-deep-learning-with-brain-js/ [4]
5. Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, Stan Bileschi, Michael Terry, Charles Nicholson, Sandeep N. Gupta, Sarah Sirajuddin, D. Sculley, Rajat Monga, Greg Corrado, Fernanda B. Viégas, Martin Wattenberg. Arxiv. (Feb.28, 2019). * TensorFlow.js: Machine Learning for the Web and Beyond*. https://arxiv.org/abs/1901.05350 [4]
6. Youdiowei Eteimorde. Dev. (Mar.18, 2023). * Introduction to Tensorflow.js*. https://dev.to/eteimz/introduction-to-tensorflowjs-443h [5]
7. TensorFlow. (Aug.15, 2024). *Introduction to Tensors*. https://www.tensorflow.org/guide/tensor [5]
8. TensorFlow. *Get started with TensorFlow.js*. https://www.tensorflow.org/js/tutorials [4]

9. Jacob Murel Ph.D., Eda Kavlakoglu. IBM. (Feb.12, 2024). *What is transfer learning?*. https://www.ibm.com/topics/transfer-learning [6]
10. Nicholas Renotte. YouTube. (Sep.27, 2020). *Real Time AI HAND POSE Estimation with Javascript, Tensorflow.JS and React.JS*. https://www.youtube.com/watch?v=f7uBsb-0sGQ&list=PLgNJO2hghbmhqne2KldbiWfzMGJSB6mQK&index=2 [7]
11. Nicholas Renotte. YouTube. (Oct.11, 2020). *Real Time AI GESTURE RECOGNITION with Tensorflow.JS + React.JS + Fingerpose*. https://www.youtube.com/watch?v=9MTiQMxTXPE&list=PLgNJO2hghbmhqne2KldbiWfzMGJSB6mQK&index=5 [7]
12. Valentin Bazarevsky, Ivan Grishchenko, Eduard Gabriel Bazavan, Andrei Zanfir, Mihai Zanfir, Jiuqiang Tang, Jason Mayes, Ahmed Sabie, Google. TensorFlow Blog. (Nov.15, 2021). *3D Hand Pose with MediaPipe and TensorFlow.js*. https://blog.tensorflow.org/2021/11/3D-handpose.html [9]
13. npm. (Oct.29, 2021). *Fingerpose*. https://www.npmjs.com/package/fingerpose [7]
14. tensorflow. GitHub. (Last update: May, 2024). https://github.com/tensorflow/tfjs-models/tree/master/hand-pose-detection [9]
15. Anonymous. Google. (Mar.31, 2022). *TensorFlow.js:Make your own "Teachable Machine" using transfer learning with TensorFlow.js*. https://codelabs.developers.google.com/tensorflowjs-transfer-learning-teachable-machine#0 [7, 8, 9, 10]
16. TensorFlow. *tf.keras.applications.MobileNetV3Small*. https://www.tensorflow.org/api_docs/python/tf/keras/applications/MobileNetV3Small [9]
17. Figure 1: https://www.linkedin.com/pulse/decoding-cnn-architecture-unveiling-power-precision-neural-moustafa
18. Figure 2: https://mk.wikipedia.org/wiki/%D0%9C%D0%B0%D0%BA%D0%B5%D0%B4%D0%BE%D0%BD%D1%81%D0%BA%D0%B0_%D0%B7%D0%BD%D0%B0%D0%BA%D0%BE%D0%B2%D0%BD%D0%B0_%D0%B0%D0%B7%D0%B1%D1%83%D0%BA%D0%B0
19. Figure 3: https://mediapipe.readthedocs.io/en/latest/solutions/hands.html