DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Gesture Recognition with a Neuromorphic Vision Sensor and Deep Learning

Marten Lienen

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Gesture Recognition with a Neuromorphic Vision Sensor and Deep Learning

Gestenerkennung mit einem neuromorphen Vision-Sensor und Deep Learning

Author:	Marten Lienen
Supervisor:	Prof. Dr.-Ing. Alois Knoll
Advisor:	Guang Chen, Priv.-Doz. Dr. Florian Röhrbein
Submission Date:	July 17, 2017

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, July 17, 2017　　　　　　　　Marten Lienen

Abstract

In 2008 the Institute of Neuroinformatics at UZH/ETHZ published a frame-free, neuromorphic vision sensor. Its output is a continuous-time time series of events that signify a local intensity change at one of its 128x128 pixels. In this thesis, we utilize recurrent neural networks to perform gesture recognition on such time series. Our main contribution is the use of an autoencoder network for sequence compression which improves the performance of downstream recognition systems significantly. We create a segmented, neuromorphic vision dataset and demonstrate a recognition accuracy of 85% on a validation set with a hybrid system of recurrent neural networks and hidden Markov models. In particular, we compare three different representations of the event stream: the raw event data, visual features extracted with a pre-trained neural network from frame reconstructions and learned representations from an autoencoder. Furthermore, we compare two methods for training neural network classifiers, framewise classification and connectionist temporal classification, and two methods for decoding classifications into a label sequence, HMM decoding and a two-step process with HMM segmentation.

Acknowledgements

I would like to thank my advisor Guang Chen for making this thesis possible. I would also like to thank the Human Brain Project group at the I6 chair for providing a DVS and the mathematics chair M3 for the donation of computational resources. Most of all, I would like to thank my family for their continued support throughout my studies.

Contents

1 Introduction

In 2008 Lichtsteiner, Posch, and Delbruck presented a type of image sensor that they called an asynchronous temporal contrast vision sensor [LPD08] which sees the world as a series of light intensity changes through time instead of a sequence of still frames. Since then a spin-off company markets the chip to research institutes around the world under the name Dynamic Vision Sensor (DVS). In this thesis we use GRUs, short for gated recurrent units [Cho+14], to recognize hand gestures in DVS recordings. GRUs are a type of neural network layer for sequence learning and a recent simplification of the influential Long Short-Term Memory (LSTM) layer [HS97].

Computer vision has revolved around images as still frames and series of these, also known as videos, since its inception in the late 60s when Martin Minsky at MIT asked his undergraduate student Gerald Jay Sussman to solve vision as a summer project. Starting in the 70s hordes of researchers have developed methods to track objects through video, reconstruct 3D maps of the world from one or multiple frames, recognize faces in images and much more. Nonetheless, chips like the DVS leave all this behind and break open a new branch of vision – neuromorphic vision.

Neuromorphic means that the DVS draws its inspiration from the way vision happens on the retina of a biological eye, e.g. the human eye. This manifests itself in its eponymous attributes: asynchronous and temporal contrast. The former means that each of the DVS' pixels generates an intensity change event as soon as it is triggered as opposed to the synchronous way in which a traditional camera queries all pixels at once every few milliseconds. The latter refers to the fact that a pixel is triggered when the change in light intensity at its position exceeds a certain threshold. Taken together these properties make the DVS' pixels comparable to retinal ganglion cells.

In recent years, deep learning has taken over the field of computer vision. It started in 2012 when Krizhevsky, Sutskever, and Hinton blew everyone else out of the water at the ILSVRC image classification challenge with a convolutional neural network [KSH12]. Neural networks have been refined ever since and are now even outperforming humans in these types of tasks. The aim of this thesis is to investigate in what capacity these successes can be transferred to neuromorphic vision by the example of gesture recognition. The main challenges are twofold. First, the data a DVS records is inherently temporal and each data point on its own carries little meaning. In comparison, a video is also temporal, yet each frame already captures a lot of the scene on its own. Equally important is the lack of training data. The ubiquity of smartphones and cameras has people collecting ever increasing amounts of visual data. Neuromorphic sensors, on the contrary, are few and far between with a shortage of publicly available data. We handle these difficulties by transforming the event stream into fewer but richer events and recording and labeling our

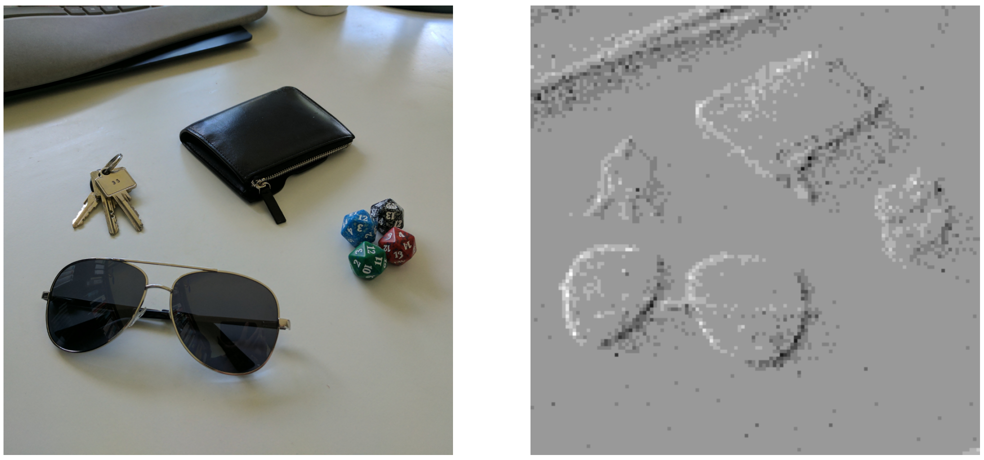


Figure 1.1: Keys, a wallet, dice and sunglasses captured with a camera (left) and with a DVS (right). The righthand picture was generated by accumulating events over 1/30 of a second.

own hand gesture dataset.

In the next chapter, we are going to review other works regarding gesture recognition, the DVS and deep sequence learning. Chapter 3 will introduce the DVS in more depth and discuss its advantages over frame-based cameras. In Chapter 4, we describe our gesture dataset, the process of gathering and labeling it and how we preprocessed the data. Chapter 5 describes two feature extraction methods that we applied to the data and how we used deep learning and hidden Markov models to detect and recognize gestures. Chapter 6 presents the results and Chapter 7 concludes the thesis.

2 Related Work

Ahn, J. H. Lee, Mullen, and Yen were one of the first to use the DVS for gesture recognition when they detected and distinguished between the three throws of the classical rock-paper-scissors game. In a first step they accumulate events into grayscale images in a similar fashion as demonstrated in Figure 1.1. Then they compute hand-designed features over these images, for example hand boundaries and the visible area of the palm, as was common at the time in computer vision with features like the history of gradients. Note, that their work was published in 2011 and thus predates the deep learning era.

The DVS' inventors in Zurich and Samsung's R&D lab in Suwon collaborated over several years on a project to implement gesture recognition with spiking neural networks and leaky integrate-and-fire (LIF) neurons [Lee+12a; Lee+14; Lee+12b]. Spiking neural networks (SNNs) are trainable models of the brain and are thus a good fit for neuromorphic sensors such as the DVS which is itself inspired by the way retina cells communicate with the brain. LIF neurons are one of the building blocks of SNNs and basically keep a count of the events over time in a small part of the DVS' field of view which has a denoising effect. With this setup the group achieved a recognition rate of over 90% with 11 different gestures.

In 2016 another group at Samsung applied deep learning to the problem [Par+16]. First, they super-resolve the event stream with spatiotemporal demosaicing which improved their system's performance on gestures performed more than 4 meters away from the DVS. Next, they extend the naive frame reconstruction technique with temporal information by stacking three successive grayscale images into the color channels of an RGB image. Finally, they train a GoogLeNet CNN to classify these temporal-fusion frames and decode the network output with an LSTM. They achieved a recognition rate of over 90% with 6 gestures and 5 distances.

3 Dynamic Vision Sensor

In 2008 Lichtsteiner, Posch, and Delbruck proposed the design for the asynchronous temporal contrast vision sensor, a neuromorphic, frame-free image sensor. Later a spin-off company called inilabs built this design into the dynamic vision sensor (DVS) which they distribute to labs around the world as part of their neuromorphic product suite. Additionally, they develop the software jAER for recording, replaying and manipulating DVS data.

The DVS produces a stream of illumination change events whereas a pixel generates an event as soon as it detects a change in illumination that exceeds a configurable threshold without synchronization with surrounding pixels. It is this property that makes the sensor comparable to ganglion cells in the human retina insofar as the pixels act asynchronously and react to illuminance changes over time. As a consequence, under constant lighting conditions, the sensor captures most of its signal at the edges of objects because smooth surfaces such as human skin have almost constant illumination when moving in front of the DVS. So, opposed to conventional cameras that transmit the whole picture with each frame, the DVS filters out redundant information from still standing objects which thus incur neither transmission nor computational costs. Therefore the DVS lends itself particularly well to applications that run on battery or have to rely on low-power microcontrollers such as aerial drones. Another proposed application are surveillance scenarios, for example traffic monitoring or security cameras, that primarily detect movement and track objects in video streams and thus could run on cheaper hardware if the vision sensor would filter out movement from the beginning. Finally, the DVS does not exhibit motion blur because each pixel generates an event as soon as a change in illumination is registered. This makes it a good fit for tracking fast moving objects, for example in robotics or human-computer interfaces such as eye trackers or gesture recognition. Furthermore, the device has already been used to track the wings of fruit flies and particles in fluid dynamics experiments.

The sensor has a spatial resolution of 128×128 pixels and a temporal resolution of 10 µs. This means that events are timestamped by a free-running counter ticking up at 100 kHz. Due to the DVS being a CMOS chip, it just consumes 23 mW of power. Each pixel has a logarithmic photoreceptor circuit which lets the DVS operate over a dynamic range of 120 dB. Formally, each pixel circuit tracks the temporal contrast defined as $c = {}^{\mathrm{d}}\!/_{\mathrm{d}t} \log I$, i.e. it is the gradient of the log-intensity of the incoming light. An event is triggered whenever the temporal contrast exceeds a threshold in either direction, i.e. $c < \theta_{\text{off}}$ or $c > \theta_{\text{on}}$. The former condition generates an OFF event while the latter generates an ON event. The type of an event is also called the polarity and is defined as the sign of the temporal contrast that triggered it, $p = \operatorname{sign}(c)$. The whole process has a latency of 15 µs and activates a reset transistor that imposes a refractory period on the pixel during which

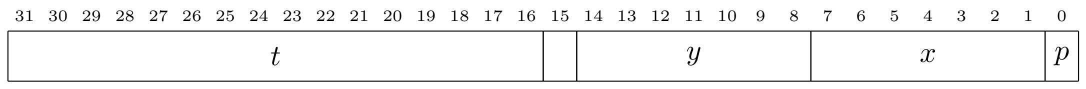


Figure 3.1: Binary format of an address event

it cannot activate again. This ensures fairness between pixels because in its absence a handful of pixels could saturate the communication bus that is limited to about 200 keps (kilo events per second) without timestamping and 60 keps with.

The DVS streams events over USB in address-event representation (AER). In AER each event is a 4-tuple (t, x, y, p) where t is the timestamp, x and y are the coordinates of the event's origin and p is the event's polarity. On the wire each tuple is encoded in 4 bytes as shown in Figure 3.1. The polarity is binary and can be encoded in a single bit. The coordinates range from 0 to 127, thus they can be encoded in 7 bits each since $2^7 = 128$. Interestingly, t is encoded as a 16 bit integer with a maximum value of $2^{16} - 1 = 65535$, even though it is incremented $100,000$ times per second. Consequently, it wraps around up to two times per second. jAER accounts for this by checking for wrap-around and incrementing a separate 32 bit counter accordingly.

There is another interpretation of event streams that is different from the physical one introduced in this chapter. Figure 3.2 shows a 6.5 ms segment from a DVS recording of a hand moving towards the sensor. You can interpret it as the hand moving through this t-x-y space while the DVS samples points from the surface defined by the hand's trajectory. We had originally hoped that this visualization would reveal a tube-like structure through time. However, the samples from the trajectory surface are too sparse for our vision to deduce the tubular shape and therefore we added a projection of the data onto the x-y plane in blue. Without this visual aid, it would be difficult to make out the shape of the moving hand at all.

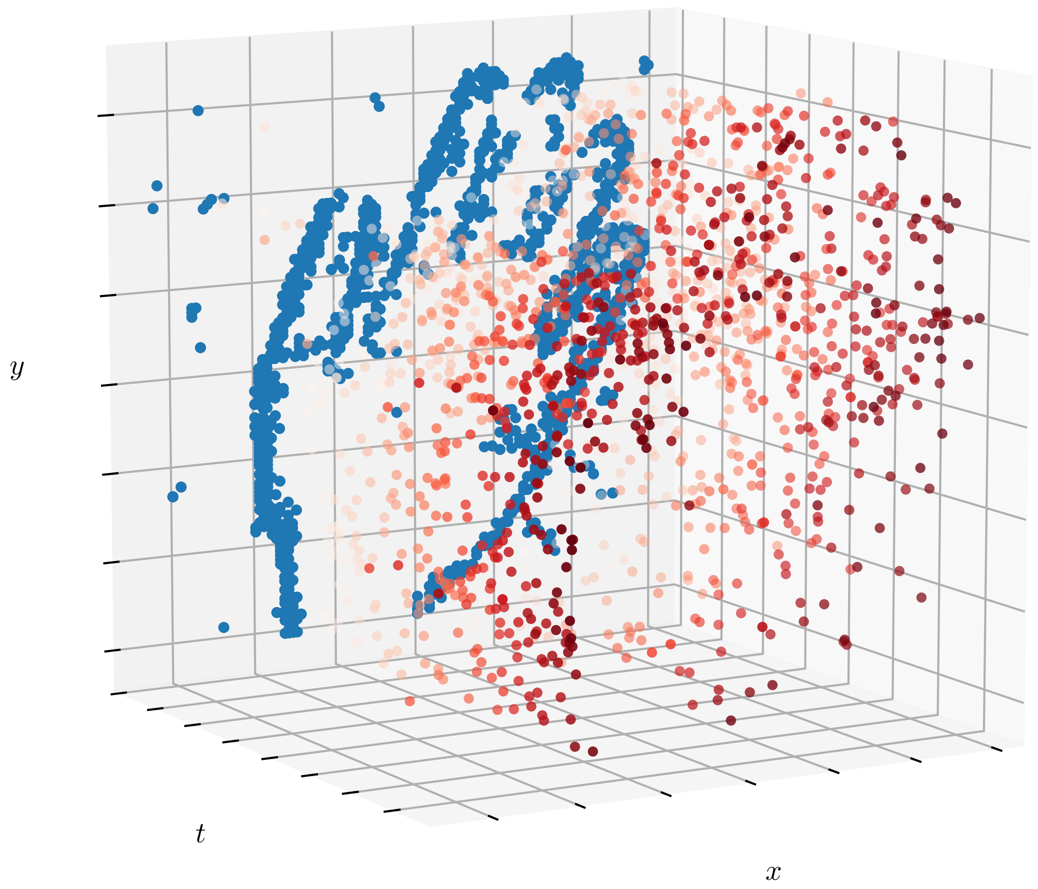


Figure 3.2: Scatter plot of 1409 events in a 6.5 ms long segment of a hand moving towards the sensor. The brightness of the red points indicates their timestamp t while the blue points are a projection of the events onto the x-y-plane.

4 A Gesture Dataset

Murphy defines machine learning as “a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty” [Mur12]. His definition underlines the central role of data in learning and, consequently, the data corpus that we train our deep models on will be of critical importance to their success. One of the important insights of the deep learning revolution in computer vision was that you need truly massive datasets to drive neural networks to peak performance. As a result, research groups have amassed public datasets of visual imagery for general purpose tasks as well as for specialized applications, including gesture recognition. Given the high cost of creating neuromorphic datasets from scratch, researchers have taken the logical next step and thought up a number of methods to convert these image datasets into neuromorphic datasets. One possible way is through simulation. [Bar+16] converted the Middlebury dataset [Sch+14] by feeding a model of a neuromorphic camera with the image data and the provided ground truth optical flow. Secondly, they also applied the same methodology to purely synthetic data generated from a 3-dimensional virtual world created in Blender. However, both of these methods produce insufficient results when compared to actual DVS recordings because the type of noise exhibited by the physical sensor is difficult to synthesize. [Liu+16] developed a method to convert large amounts of image data into event data with realistic noise where they place a DVS in front of a screen that either flashes an image repeatedly or moves it through the DVS’s field of view. Both kinds of presentation excite the DVS’ pixels through apparent motion, yet the generated data unsatisfactory. An object popping in and out of existence is not representative of reality and their second idea has the image moving in discrete steps, i.e. at least one pixel at a time, which produces an unnatural distribution of events. A third and promising idea is derived from a microscopic, twitching movement called a saccade that the human eye performs up to a hundred times a second and which was found to be essential to vision. [Orc+15] recreate this mechanism by mounting a DVS on a pan-tilt base and placing it in front of a screen showing images. The apparatus then pans and tilts cyclically which stimulates the DVS’s pixels continuously as opposed to the discrete excitations from a moving image. After careful consideration, we concluded that these approaches either produced unsatisfactory results or required too much setup work and decided to record our own dataset instead.

Tan, Lallee, and Orchard analyzed computer vision datasets to deduce guidelines for the neuromorphic vision community for creating useful datasets [TLO15]. Their points most relevant to a dataset for this thesis are, first, that a dataset should be representative of the real word and, at the same time, a dataset should be challenging and just outside of the realm of what is currently possible. A corpus that is too easy or clean to the point where it can be solved with existing methods will be irrelevant because it does not further the field. On the other hand, a data corpus that is too hard will not be of any use because new ideas will fail on it regardless of their quality. These arguments also preclude neuromorphic datasets derived from image datasets in any of the ways above.

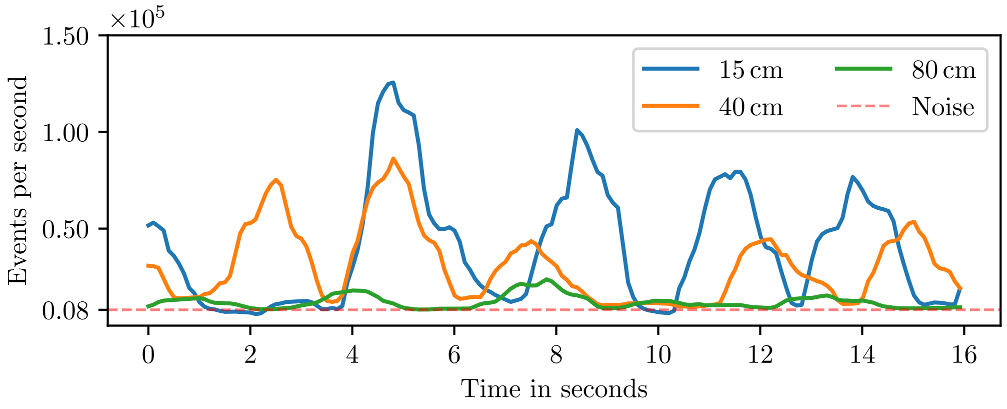


Figure 4.1: Event density for various distances between the hand and the DVS

We kept these guidelines in mind when we designed our dataset. Previous works on neuromorphic gesture recognition worked with 3, 6 or 11 gestures, so we opted for the 25 gestures that [Mol+16] have used for recognition in classical vision. They are listed in Table 4.1. Our first dataset consisted of 22 recordings of one subject with each gesture performed once in each recording. To increase the variation, the subject wore different kinds of clothing and varied the distance to the sensor between each recording. The kind of clothing turned out to be an irrelevant factor because a resolution of 128×128 is too coarse to make out such fine details. On the contrary, the distance is a significant source of variation. Figure 4.1 shows the event rates of three recordings taken at three different distances. In tests we have measured a noise event rate of about 8 keps when the DVS is directed towards a static scene which is the baseline rate between gestures regardless of the distance. The peaks in the event rate show that the event rate above baseline is proportional to the gap between hand and DVS because the main source of events, the hand, takes up less space in the field of view of the camera and stimulates fewer pixels. On the grounds that recordings with a gap length of over 60 cm were at times almost indistinguishable from noise and it was even difficult for a human viewer to tell the gestures apart, we had to exclude them from the dataset. Very close up recordings had to be excluded for a different reason. They have a great signal-to-noise ratio, yet the hand is so close to the sensor that it leaves the camera's field of view for significant stretches of time during some gestures, and we were concerned that this would render the gestures indistinguishable. All things considered, we fixed the distance to about 40 to 50 cm and removed recordings with significantly different gap lengths because that resulted in an acceptable compromise between the two criteria. This reduced the size of the 25-gesture dataset to 18 recordings.

The complete set of 25 gestures listed in Table 4.1 proved problematic nonetheless. We were concerned from the start because the corpus has a high inter-class similarity and we had fewer examples per class than there were classes. For example, `two-fingers-down` and `swipe-down` are almost the same movement with the one difference that the former just uses two fingers instead of four. Indeed, a first complete run through the recognition system revealed these concerns to be justified as you can see in the confusion matrix shown in Figure 4.2. In this visualization, we identified nine gestures that posed particular difficulties for our system. These include all `two-fingers-*` classes as explained above, `extend-*` because they are confused with a tapping motion, `tap-two-fingers` because it is hard to distinguish between tapping with a single or multiple fingers and finally `grab` because it is too similar to `rotate-outward`.

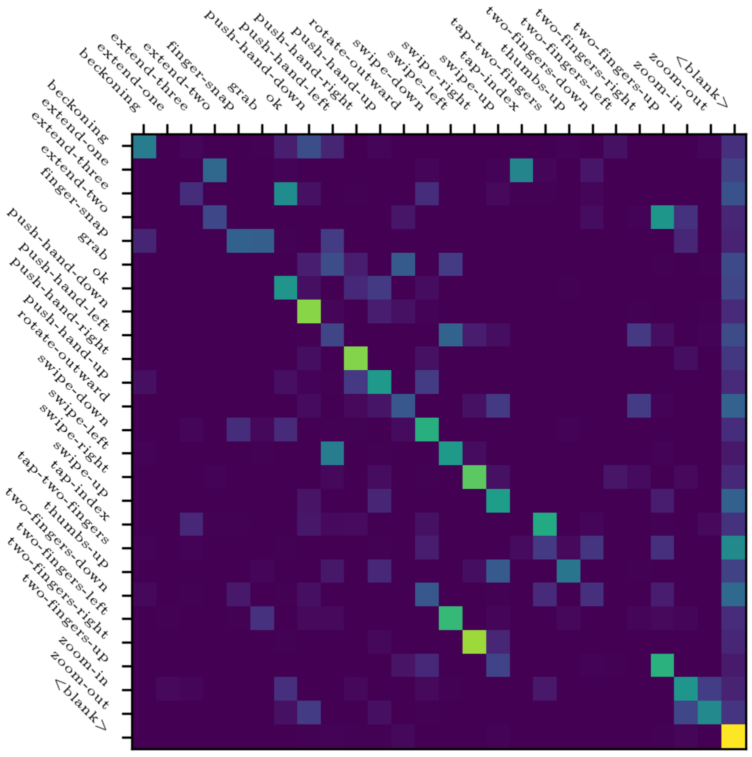


Figure 4.2: Confusion matrix for an initial run on the 25-gesture dataset

At that point, we chose to discard the 25-gesture corpus in favor of a new dataset that would exclude the nine problematic gestures and at the same time have more instances per class performed by more subjects. This second dataset only includes the reduced set of gestures listed in the right-most column of Table 4.1 which are performed by two subjects. Both subjects did twenty recordings each of which contains all sixteen gestures in random order. During the recording, the subject would position themselves so that they could comfortably hold their right hand centrally in front of the DVS at a distance of about 50 cm. This time we did not make the subjects change their appearance in any way between recordings. We also ensured constant illumination by daylight because changing lighting conditions can create virtual edges and therefore events on otherwise smooth surfaces when shadows move across them.

In the end, we collected two datasets, a small one with 25 gestures and a larger one with 16 gestures. However, the first is only used in combination with the latter on an unsupervised learning task described in Section 5.4 because there the actual gestures are only secondary and so we can cheaply profit from the additional data. For everything else, we rely on the 16-gesture set only. We could have tried to reuse parts of the original dataset since, after all, the remaining 16 gestures are a subset of the original 25. Yet we decided against it because the other classes are still present in the data and both options of handling them are unsatisfactory. In a brute-force approach, we could have deleted all events that belong to one of the excluded classes, but that would have resulted in abrupt cuts in the recordings. Another idea is to relabel these classes as one special `other` class, but that class would necessarily be easy to mix up with all other classes.

Each recording is labeled in two ways. First, we have stored the so-called recording log that states the random order in which the subject performed the gestures and which is what we ultimately want the software to reconstruct from the data. Second, we have segmented each recording explicitly with a purpose-built software. In this case segmentation of a time series means a list of start and end timestamps for each gesture. We have divided the dataset into a training and a validation set where the last two recordings of each subject constitute the validation set and the rest becomes the training set.

In Section 4.1 we describe the software we developed for recording and how it came to be. Section 4.2 showcases the software for segmenting the data. Finally, we describe the preprocessing of the data in Section 4.3.

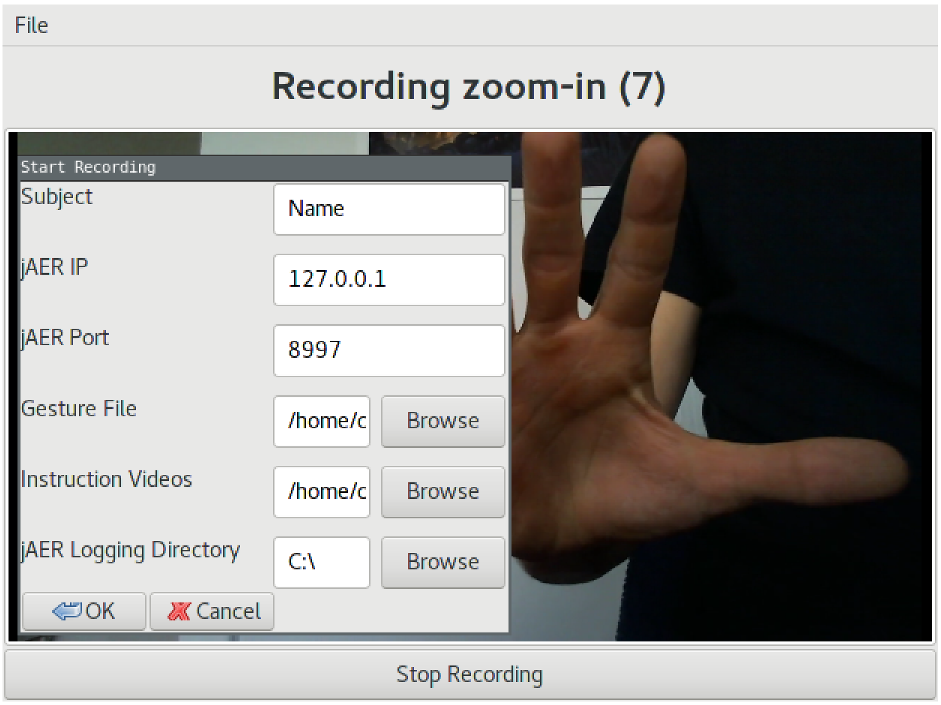
4.1 Recording Software

Our dataset consists of a number of recordings, each of which shows a subject performing each gesture exactly once in random order. In a first attempt, we planned on combining all aspects into one program with the intention of making the experience a smooth one for the subject and time-saving for the instructor. So we wrote a program called `recorder` that was meant to show instructional videos to the subject, instruct them on when to perform a gesture with a countdown and in which order and would also let the instructor segment the time series at the same time. After a first iteration, we saw that we had failed by all accounts. The program was unstable, probably due to the direct device access with a C library, and crashed from time to time, the countdown made the experience unnecessarily stressful for the subject and the segmentations were inaccurate because each subject has their own rhythm in performing when the countdown hits zero. Because of these shortcomings, we scrapped `recorder` and split its functionality into separate programs with well-separated responsibilities.

The successor is called `jaerrec` and you can see its user interface in Figure 4.3. On

Gesture	25 dataset	16 dataset
beckoning	✓	✓
extend-one	✓	
extend-three	✓	
extend-two	✓	
finger-snap	✓	✓
grab	✓	
ok	✓	✓
push-hand-down	✓	✓
push-hand-left	✓	✓
push-hand-right	✓	✓
push-hand-up	✓	✓
rotate-outward	✓	✓
swipe-down	✓	✓
swipe-left	✓	✓
swipe-right	✓	✓
swipe-up	✓	✓
tap-index	✓	✓
tap-two-fingers	✓	
thumbs-up	✓	✓
two-fingers-down	✓	
two-fingers-left	✓	
two-fingers-right	✓	
two-fingers-up	✓	
zoom-in	✓	✓
zoom-out	✓	✓

Table 4.1: List of gestures in the datasets



Subject	Name
jAER IP	127.0.0.1
jAER Port	8997
Gesture File	/home/c
Instruction Videos	/home/c
jAER Logging Directory	C:\

Figure 4.3: User interface of `jaerrec`

the left is the setup dialog for a recording and in the background is the instruction video for `zoom-in`. The recording of the event stream is offloaded to jAER which `jaerrec` controls remotely via UDP commands. This leaves it with the responsibility of playing the instruction videos in the order provided by the recording log that is generated beforehand and is just a random permutation of all gestures. Due to the stress induced by the timing requirements and the resulting inaccuracy the semi-automatic segmentation, we split that part off into a separate program as well.

4.2 Segmentation Software

We have written an extra labeling software called `labeler` shown in Figure 4.4 that is optimized to minimize the time it takes to segment a recording while giving the user the tools to place the gesture boundaries with high precision. It reads the recording log of gestures together with the recorded data and replays it. Then the user guides the segmentation process with just the "New Label". The process is implemented as a finite-state machine with three states and only a single transition going in and out of each state. The default state is `looking-for-gesture` where the playback happens at 150% to quickly skip irrelevant time between gestures. When the user spots a gesture, he or she clicks the button and the machine transitions to a new state `looking-for-start`. In that state the playback runs in reverse at 50% speed so that the user can place the start marker accurately. When he has spotted the beginning of the gesture, he or she clicks

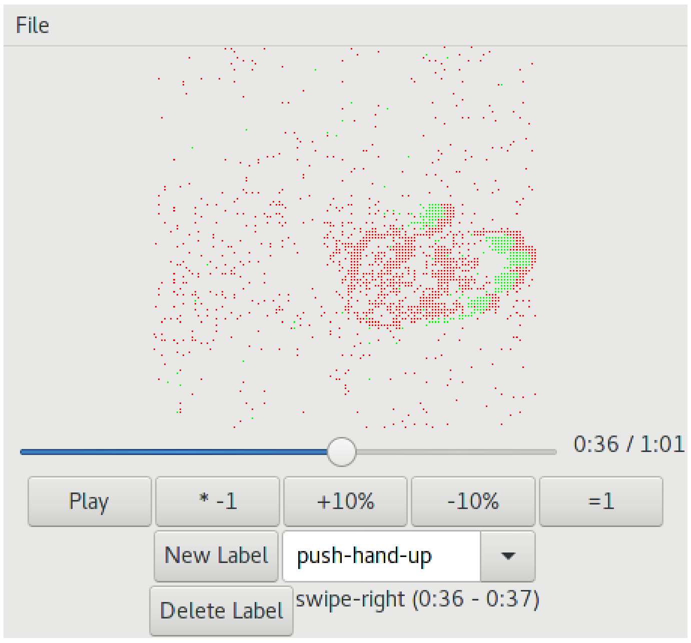


Figure 4.4: User interface of `labeler`

again which marks the start and transitions to the last state `looking-for-end`. Now the playback reverses again though it is still slowed down. One last click at the end of the gesture adds the start and end markers to the segmentation and transitions back to the initial state. The name of the gesture is automatically taken from the recording log by keeping a count of how many gestures have been marked already. This optimized process reduces the user interaction in the general case to the repeated clicking of a single button and allows the segmentation of one minute of recorded material in about three minutes. Figure 4.5 showcases a finished segmentation.

4.3 Preprocessing

Figure 4.6 shows a heatmap of all events in a recording and it looks about as expected. In the left half one can barely make out the outline of the subject, but most events are clustered in the center where the subject held his or her hand. Near the top are two outliers, colored yellow and green, that are significantly brighter than any pixel in their neighborhood. These are most likely faulty circuits that trigger continuously. On the other hand, a handful of pixels in the top right did not generate a single event over the course of this recording and are most likely faulty as well.

Even though the data has the expected form, it is still unfit for further processing for several reasons. First, the timestamps of the events are steadily increasing which means that the event sequence is not time-invariant, i.e. the same gesture performed a second later would generate a different sequence. Second, the time series is not location-invariant

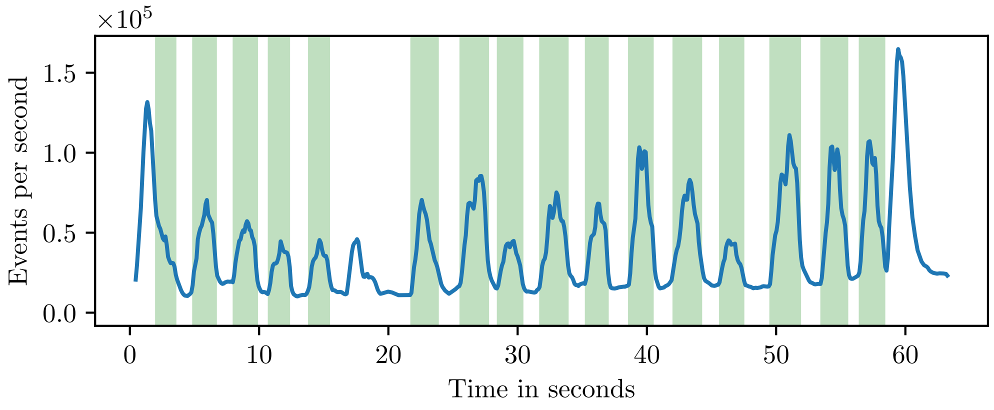


Figure 4.5: A segmentation created with `labeler`. The line plots the event density and the shaded regions mark the extent of the gestures.

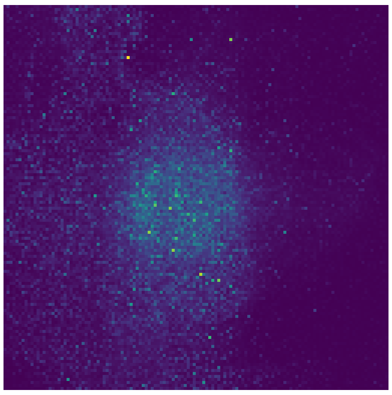


Figure 4.6: Heatmap of pixel activity

because the x and y coordinates are absolute with respect to the origin. This means that a gesture performed in the bottom-left corner of the field of view would produce a time series different from the identical gesture performed near the center. Last, the data is not standardized. Most machine learning algorithms, including deep learning, perform best when the features have mean 0 and standard deviation 1.

We made the event sequence time-invariant by introducing a new feature δt that is defined as the time passed since the previous event, i.e. $\delta t^{(i)} = t^{(i)} - t^{(i-1)}$ with a value of 0 as the base case. In this way, we have replaced an absolute point of reference, some arbitrary point in time, with a point of reference that is defined by the data itself, in this case simply the timestamp of the previous event. To make the data location-invariant we need to introduce a relative reference point for x and y as well. Assume that we would have gone with the same naive idea as we did for t. Then an actual event following a noise event would also receive random nonsense values for δx and δy just as the noise event did, effectively doubling the amount of noise in the data. Therefore, this line of thought necessitated a more stable point of reference. Our solution is to keep track of a mean with exponentially decaying weights because it gives more weight to recent events and can be cheaply computed in a streaming context such as online recognition. Such a mean μ_x that tracks a quantity x through continuous time is defined as

$$\mu_x^{(i)} = \left(1 - \alpha^{t^{(i)} - t^{(i-1)}}\right) \cdot x^{(i)} + \alpha^{t^{(i)} - t^{(i-1)}} \cdot \mu_x^{(i-1)}$$

where $x^{(i)}$ was observed at time $t^{(i)}$. The parameter α controls how much weight is placed in past data. The extreme cases of $\alpha = 0$ and $\alpha = 1$ do not keep track of history at all respectively never change μ_x from its initial value. To make this parameter α more meaningful, we use the half-life reparameterization with the parameter λ. When you plug in the following

$$\alpha = \exp\left(\frac{\log\left(\frac{1}{2}\right)}{\lambda}\right),$$

all values of $x^{(i)}$ that were observed more than λ time ago are assigned a total weight of one half.

We then keep two means for each of x and y, one with $\lambda = 1\,\mathrm{s}$ and another with $\lambda = 50\,\mathrm{ms}$. The first is supposed to track the main movement of the hand, while we intend the second to track fast movement like individual fingers. Figure 4.7 plots both means for the x-coordinate and shows that the intended effect was achieved. For example, the third shaded region in that plot marks a `swipe-right` gesture where the subject first moves the hand slightly left and then in a wavy motion to the right.

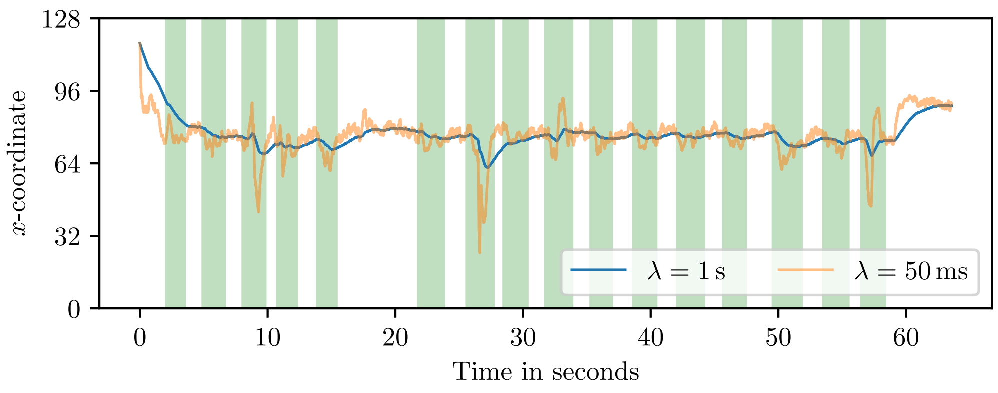


Figure 4.7: Slow and fast tracking means of the x coordinate of an event stream

On the whole the preprocessing maps each event from four features to six features as follows

$$\left(t^{(i)}, x^{(i)}, y^{(i)}, p^{(i)}\right) \mapsto \left(\delta t^{(i)}, \delta x^{(i)}_{\lambda=1\,\mathrm{s}}, \delta x^{(i)}_{\lambda=50\,\mathrm{ms}}, \delta y^{(i)}_{\lambda=1\,\mathrm{s}}, \delta y^{(i)}_{\lambda=50\,\mathrm{ms}}, p^{(i)}\right)$$

where $\delta x^{(i)}_{\lambda=1\,\mathrm{s}} = x^{(i)} - \mu^{(i)}_{x,\lambda=1\,\mathrm{s}}$ et cetera. None of the final features has an absolute point of reference, so they are time- as well as location-invariant.

Finally, we need to standardize the attributes. Let $X \subset \mathbb{R}^6$ be a set of preprocessed events. Then we compute the sample mean μ_X and standard deviation σ_X by

$$\mu_X = \frac{1}{|X|} \sum_{x \in X} x \quad \text{and} \quad \sigma_X = \sqrt{\frac{1}{|X|} \sum_{x \in X} (x - \mu_X)^2}$$

where the squaring and square-root are applied component-wise. Then we replace each x_i by $\bar{x}_i = \frac{x_i - \mu_X}{\sigma_X}$ where the division again is done component-wise. Note that we also standardize the polarity because OFF events outnumber ON events about 60/40, so its mean and standard deviation are not quite 0 and 1, respectively. This final set $\bar{X}$ then constitutes a fully preprocessed training set. Of course, the sample μ and σ are reused from the training set to preprocess the validation and test sets.

At the time, when we designed this process, we were also worried about scale-invariance. The effect of scale is nicely shown in Figure 4.1. In that plot we can see that the event rate depends directly on the distance between the subject and the DVS. So the same gesture performed up close to and bit farther from the sensor would still result in very different recordings. Accounting for this would entail accounting for spatial as well as temporal scale because events from a close-up gesture are spread further across the DVS's field of view as well as take up longer stretches in the event stream. We have experimented with normalizing the data in both regards. We estimated the spatial extent of a gesture with a Kalman filter and scaled the event coordinates proportional to that measure. Regarding the higher event rate, we subsampled the data down to a target event rate. In the end, we decided that our measures regarding scale-invariance were not effective enough and fixed the distance between subject and sensor instead.

Table 4.2 lists statistics about the final dataset. The first thing to note is that not every gesture was performed exactly 40 times as one would expect from two subjects repeating each gesture 20 times. Instead some gestures were performed twice to make sure that they were clearly visible. The average duration of a gesture is between 1.3 and 2 seconds with only a small amount of variation between each instance. However, the average number of events per instance varies considerably between the classes and also between iterations of the same gesture as testified by the large ratio between the means and standard deviations. The data not belonging to any class, labelled `<blank>`, makes up about half of the data measured in seconds yet it is only 29% of the event data. This is due to the reduced event rate between gestures when the subject's movement is minimal.

Gesture	#	Time	Events
beckoning	41	67 s ($\mu = 1.6\,\text{s}, \sigma = 0.2\,\text{s}$)	2,949,776 ($\mu = 71,945, \sigma = 29,515$)
finger-snap	40	59 s ($\mu = 1.5\,\text{s}, \sigma = 0.3\,\text{s}$)	2,203,881 ($\mu = 55,097, \sigma = 19,777$)
ok	40	63 s ($\mu = 1.6\,\text{s}, \sigma = 0.2\,\text{s}$)	3,054,306 ($\mu = 76,357, \sigma = 18,290$)
push-hand-down	41	77 s ($\mu = 1.9\,\text{s}, \sigma = 0.3\,\text{s}$)	4,000,518 ($\mu = 97,573, \sigma = 33,154$)
push-hand-left	40	74 s ($\mu = 1.9\,\text{s}, \sigma = 0.4\,\text{s}$)	3,859,846 ($\mu = 96,496, \sigma = 32,781$)
push-hand-right	39	76 s ($\mu = 1.9\,\text{s}, \sigma = 0.3\,\text{s}$)	5,550,913 ($\mu = 142,331, \sigma = 57,933$)
push-hand-up	41	84 s ($\mu = 2.0\,\text{s}, \sigma = 0.3\,\text{s}$)	5,278,815 ($\mu = 128,751, \sigma = 48,901$)
rotate-outward	40	51 s ($\mu = 1.3\,\text{s}, \sigma = 0.2\,\text{s}$)	2,104,242 ($\mu = 52,606, \sigma = 18,325$)
swipe-down	41	68 s ($\mu = 1.6\,\text{s}, \sigma = 0.3\,\text{s}$)	3,789,590 ($\mu = 92,429, \sigma = 47,715$)
swipe-left	41	66 s ($\mu = 1.6\,\text{s}, \sigma = 0.3\,\text{s}$)	3,709,310 ($\mu = 90,470, \sigma = 36,633$)
swipe-right	40	65 s ($\mu = 1.6\,\text{s}, \sigma = 0.3\,\text{s}$)	3,339,538 ($\mu = 83,488, \sigma = 39,184$)
swipe-up	42	71 s ($\mu = 1.7\,\text{s}, \sigma = 0.3\,\text{s}$)	4,359,013 ($\mu = 103,786, \sigma = 38,998$)
tap-index	40	62 s ($\mu = 1.6\,\text{s}, \sigma = 0.2\,\text{s}$)	2,620,666 ($\mu = 65,516, \sigma = 20,536$)
thumbs-up	40	55 s ($\mu = 1.4\,\text{s}, \sigma = 0.2\,\text{s}$)	1,939,608 ($\mu = 48,490, \sigma = 15,750$)
zoom-in	40	73 s ($\mu = 1.8\,\text{s}, \sigma = 0.3\,\text{s}$)	3,414,950 ($\mu = 85,373, \sigma = 36,047$)
zoom-out	41	74 s ($\mu = 1.8\,\text{s}, \sigma = 0.3\,\text{s}$)	3,425,209 ($\mu = 83,541, \sigma = 28,812$)
<blank>	-	22.18 min	28,865,096
Total	647	40.25 min	84,465,277

Table 4.2: Number of instances, total amount of data in seconds and number of events for each of the 16 gestures in the final dataset. The quantities in parentheses give the average per instance and the standard deviation.

5 From Time Series to Label Sequences

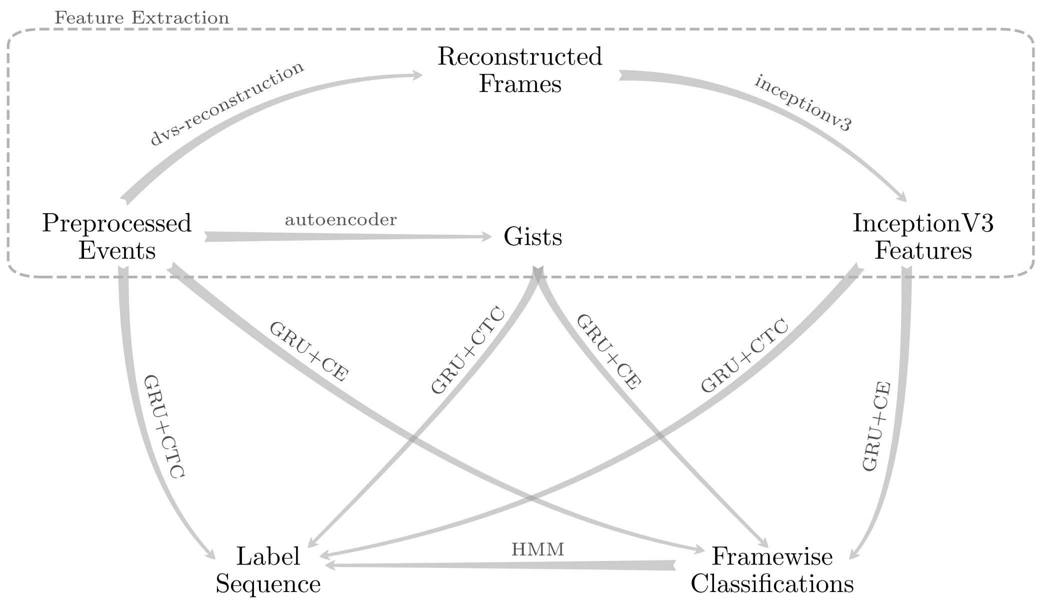


Figure 5.1: Overview of the methods and intermediate representations used in this thesis

There are three types of sequence labeling tasks defined by Graves. The first is sequence classification where the data is a set of sequences each of which belongs to a single class. This problem can be generalized to segment classification. Here a data point is subdivided into one or more segments with known start and end points and each of them needs to be classified. Splitting the inputs along the segment boundaries and interpreting the problem as a sequence classification task is not equivalent because the segment classifier may and often will use surrounding context to disambiguate ambiguous segments. The most general task is called temporal classification in which neither the number of classes per sequence nor any segment boundaries are known. The only assumption is that an input has at most as many labels as the length of the input sequence. Such a classifier has to compute just the label sequence, though some of them can optionally also return a segmentation.

The problem of gesture recognition from address-event data falls into the last category. A recording may contain any number of gestures at arbitrary points in time and each gesture takes up a varying number of events depending on distance between hand and

DVS, speed of execution and lighting conditions. Figure 5.1 depicts all the ways that we tried to solve this temporal classification task. All of them include recurrent neural networks (RNNs) because they are the state of the art in sequence learning. There are two popular ways to compute a label sequence from an input. The older one combines an RNN and a hidden Markov model (HMM). The RNN provides localized classifications of each sequence element and the HMM segments the input on the basis of the RNN output and deduces the most likely label sequence. These are all paths in Figure 5.1 that leave the feature extraction phase and take *framewise classifications* as an intermediate step to *label sequence*. Graves, Fernández, Gomez, and Schmidhuber devised a method called connectionist temporal classification (CTC) that combines both steps, RNN classification and HMM decoding, by means of a new loss function for neural network training. This enables a neural network in theory to solve any temporal classification task end-to-end. This approach is represented in Figure 5.1 by all paths labeled `GRU+CTC` that lead to *label sequence* directly.

Neural networks that are capable of sequence-to-sequence transformations rely on circular connections in their computational graph and are called recurrent neural networks. They process an input sequence element by element and the recurrent connections allow them to carry information, and thus context, from step to step. The particular variant of RNN we use is called a gated recurrent unit, GRU for short, which allows the networks to learn much longer temporal relationships than a naive RNN could. We introduce RNNs, GRUs and their predecessor, the long short-term memory layer, in Section 5.1.

In the beginning, we applied both methods to the 25-gesture dataset, however CTC training did not produce any result at all and framewise classification levelled off at about 40% accuracy, which we deemed to low for successful decoding. The type of data that we are handling consists of very long sequences with only minimal information content per sequence element and a high noise ratio. The recordings average 30,000 events per second and single gestures often span over 100,000 events. Even though we employ gated recurrent units that can learn relationships over hundreds of timesteps, we assumed that the network might be just overwhelmed by the data. Moreover, we conjectured that the continuous time and consequential variance in event density over time made the task more difficult. We introduced two feature extraction methods with the specific goal of overcoming these problems. Both compress sequences of varying length into fixed-length vectors. That lets us transform the continuous-time sequences into discrete-time ones by grouping the events into time windows of fixed length but varying numbers of events and compress them into fixed-length vectors. The transformed sequences should improve learning because the number of data points per second is cut down and the data points are enriched with the information of all events then went into them. On the original data, the network had to rely almost purely on the context, because a single event is uninformative. Afterwards, each data point should be quite meaningful on its own.

The first method reconstructs grayscale images from the event data (Section 5.2) and then uses a pre-trained image classification network to extract visual features (Section 5.3). The second idea, explained in Section 5.4, uses a specific type of network, an autoencoder, to learn an efficient compression directly. We call the representations learned by the autoencoder *gists* because they encode the gist of the input.

Section 5.5 explains framewise classification and the network architecture that we use for that task and all that follow. These local classifications are decoded with an HMM which is described in Section 5.6. In Section 5.7 we split the decoding process into two phases, first the input is segmented which is followed up by segment classification, both with an HMM. Lastly, we introduce connectionist temporal classification in Section 5.8 and explain how it combines the steps of classification and decoding into one.

5.1 Recurrent Neural Networks

The classical neural network is a parametric function approximator that takes an input vector of fixed length and produces an output vector of fixed length. Recurrent neural networks generalize either input, output or both to sequences. Let $x^{(n)}$ be a sequence of input vectors and define a simple network with one hidden layer. So it has a weight matrices W, V, a bias b and in this example we will use a tanh non-linearity. The naive way to produce an output sequence $y^{(n)}$ would be to apply the network elementwise by

$$\begin{aligned} h^{(t)} &= \tanh\left(Wx^{(t)} + b\right) \\ y^{(t)} &= Vh^{(t)} \end{aligned}$$

where $h^{(t)}$ are the activations in the hidden layer for the t-th sequence element. However, this network does not have access to any historical context and so it will have a difficult time to, for example, recognize gestures which are inherently temporal. One solution is to introduce recurrent connections between layers in the form of an additional dependency of $h^{(t)}$ on $h^{(t-1)}$. In precise terms this means that we add another weight matrix U to the network and redefine it as

$$\begin{aligned} h^{(t)} &= \tanh\left(Wx^{(t)} + Uh^{(t-1)} + b\right) \\ y^{(t)} &= Vh^{(t)} \end{aligned}$$

where $h^{(0)}$ is initialized arbitrarily, though with the zero vector most of the time. This network now has memory, so to speak, in the form of $h^{(t)}$ that allows it to take the history of the sequence into account.

Though fine in theory, in practice this type of RNN suffers from the vanishing gradient problem. When such a network is trained with some form of gradient descent, the gradient decays exponentially quickly as it propagates back through time. This makes it hard for such a network to solve tasks that require more than about 10 steps of temporal contrast. Hochreiter and Schmidhuber solved this with the introduction of long short-therm memory (LSTM) layers that they designed to allow constant error flow [HS97]. Instead of carrying over the state of the hidden layer directly, the LSTM architecture replaces the hidden layer with a subnetwork that explicitly controls which memories should be forgotten and which should be overwritten. This keeps the gradient magnitude large and provides sufficient signal for learning over several hundred timesteps. An LSTM layer consists of two persistent states, the cell state $c^{(t)}$ and the hidden state $h^{(t)}$, and three gates, the input, output and forget gates $i^{(t)}$, $o^{(t)}$ and $f^{(t)}$, respectively, which control the flow of information. They are connected as follows where σ is the logistic sigmoid function and $\circ$ denotes an element-wise product.

$$\begin{aligned}
f^{(t)} &= \sigma\left(W_f x^{(t)} + U_f h^{(t-1)} + b_f\right)\\
i^{(t)} &= \sigma\left(W_i x^{(t)} + U_i h^{(t-1)} + b_i\right)\\
o^{(t)} &= \sigma\left(W_o x^{(t)} + U_o h^{(t-1)} + b_o\right)\\
c^{(t)} &= f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tanh\left(W_c x^{(t)} + U_c h^{(t-1)} + b_c\right)\\
h^{(t)} &= o^{(t)} \circ \tanh\left(c^{(t)}\right)
\end{aligned}$$

The forget and input gates control the updates to the cell state while the output gate controls which parts of the cell state are written out into the hidden state and thus into the output of the layer. A later version called Peephole LSTM uses the cell state directly instead of the previous hidden state to compute the input, output and forget gates.

In 2014 K. Cho, Van Merriënboer, Gulcehre, et al. published a simplification of LSTM called a gated recurrent unit (GRU) [Cho+14]. They merge the cell state into the hidden state $h^{(t)}$, combine the input and forget gates into a single update gate $z^{(t)}$ and replace the output gate with a reset gate $r^{(t)}$ that has no equivalent in LSTM. A GRU layer is then defined by

$$\begin{aligned}
r^{(t)} &= \sigma\left(W_r x^{(t)} + U_r h^{(t-1)}\right)\\
z^{(t)} &= \sigma\left(W_z x^{(t)} + U_z h^{(t-1)}\right)\\
h^{(t)} &= z^{(t)} \circ h^{(t-1)} + \left(1 - z^{(t)}\right) \circ \tilde{h}^{(t)}\\
\tilde{h}^{(t)} &= \tanh\left(W x^{(t)} + U\left(r^{(t)} \circ h^{(t-1)}\right)\right).
\end{aligned}$$

where $\tilde{h}^{(t)}$ is an intermediate variable. This architecture has fewer parameters and operations then LSTM which makes it easier to implement and compute which reduces training time. Chung, Gulcehre, K. Cho, and Bengio have found that GRU performance is comparable to LSTM [Chu+14]. For these reasons, we have decided to rely on GRUs in this thesis.

5.2 Frame Reconstruction

The DVS only reacts to changes in light intensity. This removes all the redundant information about stationary objects that makes up the bulk of the data in videos consisting of consecutive frames. Furthermore, the remaining information about the illumination gradient at a point is coarsely discretized and carries just a single bit of information, the sign of the change. Nonetheless, two groups have shown that it is still possible to reconstruct grayscale images from the data [BDL16; RGP16]. Reinbacher, Graber, and Pock interpret the events as points on an event manifold which they reconstruct by minizing an energy including a smoothness regularizer over it with variational methods. The solution can then be interpreted as an intensity image.

They have published an open-source implementation of their algorithm which we used to reconstruct images from the dataset. We created one image every 2.5 ms resulting in 400 frames per second. Figure 5.2 showcases three frames depicting a `swipe-down` gesture. For one thing, the subject's hand has been resolved reasonable well and it has been reconstructed as one smooth surfaces with shadows creating an illusion of depth and even individual digits are visible. At the same time though, artifacts are accumulating in the background regions of the image and over time the images become darker. The reason for this is most likely that we mounted the camera on a stand. As a result it was static relative to the background which therefore produced no events.

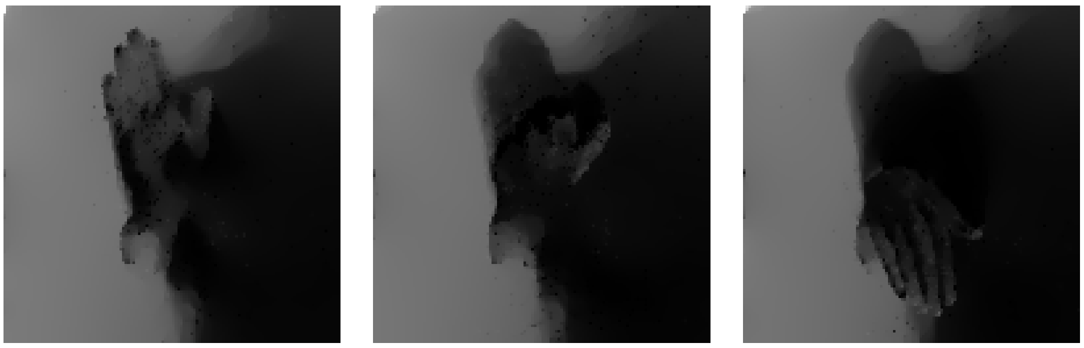


Figure 5.2: Reconstructed frames showing a `swipe-down` gesture

5.3 Feature Extraction with an InceptionV3 Network

It is an emergent property of neural networks trained on image tasks that they develop increasingly abstract features in higher levels of the network. So the bottom layers of a network will learn to recognize simple shapes like edges in various orientations or corners. On top of these, the network learns to recognize more complex shapes like circles or rectangles. This continues through the network and in the highest layers you find neurons that recognize faces or cars. An insight from transfer learning says that these learned features can be reused across a variety of tasks because only a handful of layers at the top will be completely task specific. So by reusing a pre-trained network and replacing the top layers you can reach high performance on a new task in a fraction of the time it would take to train the whole network from scratch. In our case, reusing a network pre-trained on a large dataset has the additional advantage that we can indirectly make use of a lot more training data than what our dataset provides.

The end goal is to compress sequences of events into more expressive data points. The idea for that is that we reconstruct grayscale images from the event data and then feed

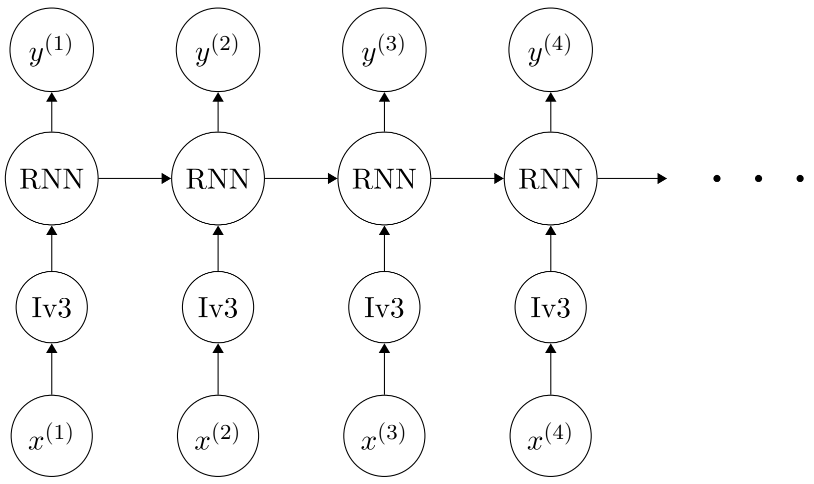


Figure 5.3: Computational graph of an RNN on top of an InceptionV3 network

these to a pre-trained network that is able to extract abstract and meaningful features about the subjects hand position from them. In particular, we chose an InceptionV3 network [Sze+16] implemented in keras [Cho+15] that was pretrained on the ImageNet dataset to 5.6% top-5 accuracy. We cut off the fully connected layers that are specialized on ImageNet and stored the output of the hidden layer at that depth as feature vectors to later train an RNN on. This setup is basically equivalent to the network whose computational graph is sketched in Figure 5.3. There an InceptionV3 network is run on each sequence element with an RNN running on top for temporal context. The difference to the depicted network is that we precompute all the intermediate outputs. The advantage is that the deep network need not be rerun on each epoch. However, its weights are also practically frozen, so they will not be fine-tuned to our problem. One detail worth noting is that the InceptionV3 network expects RGB images with a resolution of 299×299 pixels. To make the 128×128 pixel grayscale images compatible with these requirements we scaled them up and duplicated each of them three times into the three color channels of an RGB image.

5.4 Representation Learning with an Autoencoder

An autoencoder is a neural network that takes an input and reproduces it, so it is approximating the identity function. There are different flavors to this concept, though we are going to focus on the type that is in some form hour-glass shaped. That means that at some intermediate point in the network the data has to flow through a funnel, a layer whose output dimensionality is smaller than the network input. This forces the network to learn a low-dimensional representation of the input data from which it can later reconstruct the input. The advantage of using a neural network for dimensionality reduction is that the mapping can be highly non-linear and it works on sequences.

Our autoencoder follows the architecture proposed in [Cho+14], alongside GRU, for

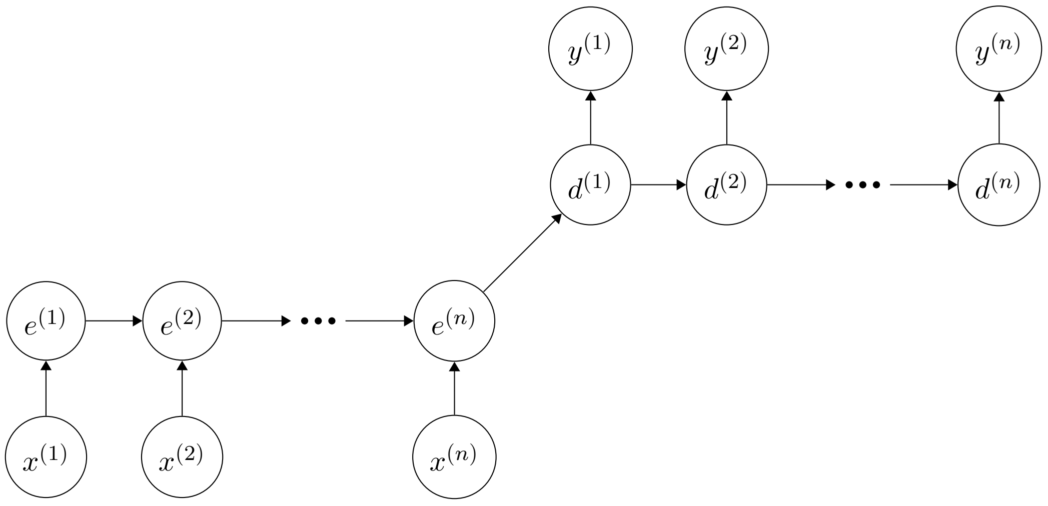


Figure 5.4: Sketch of our autoencoder architecture that encodes an input sequence x of length n into hidden states e. The decider is trained to decode the last hidden state $e^{(n)}$ into a sequence $y^{(1)}, \dots, y^{(n)}$ resembling the input.

neural machine translation. See Figure 5.4 for a sketch. The idea is that the network is split into an encoder and a decoder which share information only along a single edge in the computational graph. This edge initializes the decoder's hidden state with the final hidden state of the encoder. It is the figurative funnel in the network because it has to encode the complete input sequence. For that reason we call these encodings the *gist* of the input.

The specifics of the architecture are influenced by our requirements. We want to use the encoder part to transform variable length event sequences of fixed duration into fixed-length vectors. So the network architecture needs to accomodate inputs of variable length and accordingly also be variable regarding the output length. The preprocessed events are vectors in $\mathbb{R}^5 \times \{-1, 1\}$, so the features are partly continuous. It is common practice to train an autoencoder to produce a probability distribution over sequences instead of sequences directly. The original architecture is designed to work over a finite alphabet and produces parameters for a multinoulli distribution over that alphabet. So each $y^{(i)}$ is a non-negative vector whose entries sum up to 1 and its j-th entry encodes the networks belief that the j-th word should be placed at this point in the output sequence.

The continuity of the features requires a distribution over the real numbers. Graves uses an LSTM-RNN to generate handwriting where each sequence element is a vector in $\mathbb{R}^2 \times \{0, 1\}$ encoding the pen's x and y coordinates and whether it is currently lifted from the blackboard [Gra13]. Their network's output parameterizes a distribution that is a mixture of Gaussians over the real attribute and a Bernoulli distribution over the categorical attribute. They call such networks *mixture density networks.*

Our network handles an input sequence of length n, where n is variable, by encoding the complete sequence first. The it applies the decoder to produce a distribution over a sequence of length n and computing a loss between the two sequences for training. We chose to produce an output sequence of the correct length directly instead of learning a special `end-sequence` output because our inputs are just arbitrary slices of events as opposed to sentences in machine translation that have a meaningful end.

The architecture for our mixture density autoencoder is as follows. Both the encoder and decoder are 3 layers of GRUs each of which has 256 units. Encoder and decoder structure are necessarily symmetrical since the decoder is fed the hidden state from the encoder. The encoder receives preprocessed events in $\mathbb{R}^6$. Remember that the polarity has been standardized as well, so it is not necessarily restricted to $\{-1, 1\}$ anymore. However, for purposes of computing the loss, we rediscretize it again by mapping positive values to 1 and vice versa. The decoder produces parameters for a 10 component mixture distribution of Gaussians with diagonal covariance matrices over $\mathbb{R}^5$ and a single parameter for a Bernoulli distribution over $\{-1, 1\}$. This adds up to 10 component weights, $10 \cdot 5 = 50$ mean parameters, $10 \cdot 5 = 50$ diagonal convariance matrix entries and a single Bernoulli parameter, so 111 parameters in total. To project its 256-dimensional output into the $\mathbb{R}^{111}$, the decoder has a single fully-connected layer with weight matrix and bias term but without non-linearity on top.

The loss is the negative log-likelihood of the input sequence according to the output distribution. To compute this, the outputs, which are arbitrary vectors in $\mathbb{R}^{111}$, need to satisfy certain conditions. Therefore, except for the Gaussian mean vectors μ, all of them are transformed non-linearly. The component weights w_i are transformed with a softmax operation

$$w_i := \frac{\exp(w_i)}{\sum_j \exp(w_j)}$$

that ensures that the weights are between 0 and 1 and sum up to 1. The variance terms σ_{ij} have to be positive so that the covariance matrix is positive-definite. This is achieved with the softmax operation

$$\sigma_{ij} := \log\left(1 + \exp(\sigma_{ij})\right)$$

which ensures that a value is positive and is basically $\max\{0, \sigma_{ij}\}$ with a rounded corner at 0 which also never quite gets to 0 as $\sigma_{ij} \to -\infty$. Finally, the Bernoulli parameter p is squashed into the $(0, 1)$ range by the logistic sigmoid function

$$p := \sigma(p) = \frac{1}{1 + e^{-p}}.$$

The training setup is the same as for all networks trained for this thesis. We use minibatch gradient descent with batches of size 32. The optimizer is Adam, an optimization method that adaptively scales the learning rate for each parameter [KB14]. Though it might be unnecessary with Adam, we still employed exponential learning rate decay with 0.95 to the power of the current epoch. Lastly, we clip the gradients at a norm of 5 which Graves recommends. This also helps with numerical instabilities in case the covariances of the mixture distribution should become really small.

Training recurrent neural networks on long sequences requires lots of memory because all intermediate results over all timesteps need to be stored for back-propagation. To avoid running out of memory, we split input sequences above a maximum length into chunks. A chunk is similar to a batch, though the hidden states are not cleared in between and they are trained on in order. So a batch of sequences of length 100 might be split into five chunks where the first chunk would contain the first twenty elements of all sequences and so forth. In theory, this restricts the maximum distance in time over which the network can learn relationships to the chunk size. In practice, we did not notice anything that we would have traced back to this effect.

The autoencoder was trained on chunks of length 250, while all other networks were trained with chunks of length 1000. The reasons for that are twofold. First, the autoencoder is actually two recurrent networks in sequence, so a chunk size of 250 means the memory requirements of a chunk size of 500. Second, the autoencoder includes more expensive operations and was trained on a lot of data. Because of this, training an autoencoder takes several days, while the other networks can be trained in a matter of hours. Furthermore, the input data of the autoencoder is special. In contrast to all classifier networks, the autoencoder runs on variable length sequences. A batch of 2.5 ms time windows of events has a long-tail distribution of lengths. While most sequences will be quite short, one or two will be significantly longer. The library we use requires us to effectively zero-pad all sequences to the same length, which incurs large and unnecessary memory and computational costs. Chunking allows us to cut down on these by excluding sequences that have already ended in later chunk iterations.

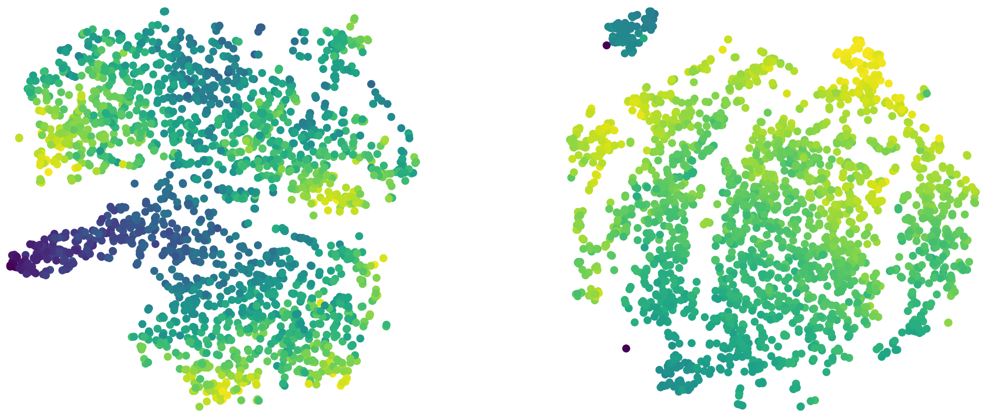

(a) Thumbs up (b) Swipe left

Figure 5.5: t-SNE plots of learned representations of 2.5 ms segments

A last detail about the autoencoder structure is that the decoder is fed the desired output of the previous timestep as input for the current timestep. Using the terminology of Figure 5.4, $d^{(2)}$ would be fed $x^{(1)}$ as input. This is supposed to improve decoding performance because the network can more easily keep track of its current position in the sequence.

An interesting question is if the autoencoder has a denoising effect on the representation. The noise is more or less uniformly distributed over the field of view. Thus, the expected value of the loss incurred by noise should be 0 in expectation and the effect through gradient descent on the network should cancel out in the long run. Therefore one might assume that the learned representation is less noisy than its input.

Figure 5.5 shows two t-SNE plots of learned representations. Each point represents 2.5 ms of events and they are colored by the number of events that went in them where a brighter color denotes more events. Note that the axes carry no meaning in a t-SNE plot and all information is in the distance between points. The key observation is that the points form some high-dimensional structure that encodes, among others, the event density in a segment.

5.5 Framewise Classification

Methods with an explicit decoding step require localized classifications that can be decoded into global classifications. In accord with Graves, we call this *framewise classifications*. Once again we use an RNN for that. This time we are back in the domain of finite alphabets since the network should assign each frame to one of 17 classes, 16 gestures plus the `<blank>` label.

To make our approaches comparable, we will use the same network architecture throughout the rest of the thesis. It consists of three GRU layers with 256 units each, two fully-connected layers with tanh activations and also 256 units each and finally a fully-connected layer without activation function that projects the output down into $\mathbb{R}^{17}$. The output is transformed with softmax to parameterize a multinoulli distribution over the output classes which encodes the networks belief that the current frame belongs to a certain class.

The loss function to optimize is the cross entropy between the outputs and the labels. The cross entropy H between two probability distributions p and q is defined as

$$H(p, q) = \mathrm{E}_p \left[-\log q \right].$$

It measures the difference between the distributions and is minimized with respect to q when $q = p$. In our case both are categorical distributions over 17 categories, where the label can be interpreted as a distribution that is 1 for the correct label and 0 for every other label. This simplifies the cross entropy to $H(p, q) = -\log(q(l))$ where l is the frame's label. When this is minimized, the predicted probability of the correct label is maximized and the probability mass on incorrect labels is necessarily reduced at the same time due to the softmax operation.

Figure 5.6 illustrates the output of a framewise classifier. Since the the `<blank>` class makes up about 50% of the training data, the classifier recognizes non-gesture data with high accuracy. When an activity is detected, the classifier assigns high probabilities to multiple classes at first until it discerns a single label as the correct one. This initial confusion could most likely be alleviated with a backward pass in a bidirectional RNN.

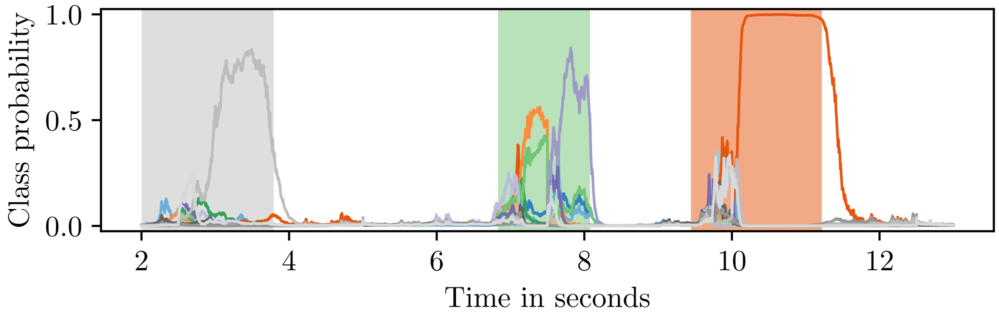


Figure 5.6: Class probabilities attained from a framewise classifier. The shaded regions in the background designate the ground truth.

5.6 Hidden Markov Model Decoding

A framewise classifier tells us which gesture most likely happened at each point in time but a `swipe-down` gesture might be classified as `rotate-outward` for the first few milliseconds, then `swipe-up` for another few and finally as `swipe-down` for the rest of the activity. This sequence of probability distributions now needs to be deciphered into just a single `swipe-down` label. One way of doing just this is with hidden Markov models (HMMs).

A hidden Markov model consists of a Markov chain of hidden states $z^{(t)} \in \{1, \ldots, K\}$ and an observation model $x^{(t)}$. A Markov chain is a probability distribution over sequences of arbitrary length where the transition probability between states only depends on the current state, so

$$p\left(z^{(t)}|z^{(1)}, \ldots, z^{(t-1)}\right) = p\left(z^{(t)}|z^{(t-1)}\right).$$

The transition model is usually written as a transition matrix A where A_{ij} is the probability of transitioning from state i into state j. The initial probabilities for $z^{(1)}$ are given by a vector $\pi \in \mathbb{R}^K$. The observation model specifies $p(x^{(t)}|z^{(t)})$ where the observation probabilities only depend on the current state. In our case, the observation model is categorical as well and specified by an observation matrix B where B_{ij} gives the probability of observation j in state i.

A hidden Markov model models a situation where a state of interest is only indirectly observable through emissions at each timestep. We have a sequence of local classifications of each frame into of 17 classes and would like to derive the true underlying sequence of gestures. An HMM lets us incorporate the knowledge that a state i might be observed as any other state for a short while through the observation matrix B.

An efficient algorithm to decode an observation sequence into the most likely underlying state sequence is *Viterbi decoding.* It produces the most likely sequence of hidden states

$$\begin{aligned}
z^{(1)}, \dots, z^{(n)} &= \operatorname*{arg\,max}_{z^{(1)},\dots,z^{(n)}} p(z^{(1)}, \dots, z^{(n)} | x^{(1)}, \dots, x^{(n)}) \\
&= \operatorname*{arg\,max}_{z^{(1)},\dots,z^{(n)}} p(z^{(1)}) \cdot \prod_{t=2}^{n} p(z^{(t)}|z^{(t-1)}) \cdot \prod_{t=1}^{n} p(x^{(t)}|z^{(t)}) \\
&= \operatorname*{arg\,max}_{z^{(1)},\dots,z^{(n)}} \log \pi_{z^{(1)}} + \sum_{t=2}^{n} \log A_{z^{(t-1)},z^{(t)}} + \sum_{t=1}^{n} \log p(x^{(t)}|z^{(t)})
\end{aligned}$$

given a sequence of observations. A problem is that a framewise classifier produces $p(z^{(t)}|x^{(t)})$ instead of $p(x^{(t)}|z^{(t)})$. We can rewrite the decoding objective with Bayes's theorem.

$$= \operatorname*{arg\,max}_{z^{(1)},\dots,z^{(n)}} \log \pi_{z^{(1)}} + \sum_{t=2}^{n} \log A_{z^{(t-1)},z^{(t)}} + \sum_{t=1}^{n} \left(\log p(z^{(t}|x^{(t)}) + \log p(x^{(t)}) - \log p(z^{(t)}) \right)$$

The $p(x^{(t)})$ term is irrelevant to the arg max because it does not depend on z.

$$\begin{aligned}
&= \operatorname*{arg\,max}_{z^{(1)},\dots,z^{(n)}} \log \pi_{z^{(1)}} + \sum_{t=2}^{n} \log A_{z^{(t-1)},z^{(t)}} + \sum_{t=1}^{n} \left(\log p(z^{(t}|x^{(t)}) - \log p(z^{(t)}) \right) \\
&= \operatorname*{arg\,max}_{z^{(1)},\dots,z^{(n)}} \log \pi_{z^{(1)}} + \sum_{t=2}^{n} \log A_{z^{(t-1)},z^{(t)}} + \sum_{t=1}^{n} \log p(z^{(t}|x^{(t)})
\end{aligned}$$

The last line is true if π is the stationary distribution of the Markov chain A, i.e. the long term distribution over hidden states, because then $p(z^{(1)}) = \pi$ and $p(z^{(t)}) = Ap(z^{(t-1)}) = \pi$. Therefore $\log p(z^{(t)})$ is actually independent of $z^{(t)}$ and can be ignored.

The Viterbi algorithm finds the maximizer by computing the probability of being in state j at time t given that you take the most probable path.

$$\delta_t(j) = \max_{z^{(1)},\dots,z^{(t-1)}} p(z^{(1)}, \dots, z^{(t-1)}, z^{(t)} = j | x^{(1)}, \dots, x^{(t)})$$

The key insight here is that the most probable path to state j at time t must be the one that maximizes the joint probability of being in state k at time $t-1$ and transitioning from k to j, i.e.

$$\delta_t(j) = \max_i \delta_{t-1}(i) \cdot A_{ij} \cdot B_{j,x^{(t)}}$$

If you compute δ for t from 1 to n and store the maximizer i in another table α_{tj}, you can find the most probable final state as $z^{(n)} = \arg\max_i \delta_n(i)$ and work your way back to $t = 1$ by following the $\alpha_{n,z^{(n)}}$ to the predecessor state and so forth. This explanation is summarized from [Mur12].

Since we do not work with the observation matrix B, the process of constructing an HMM decoder from the training data is reduced to finding A and π. The structure of A allows us to include our domain knowledge in the decoding process. We want to encode in

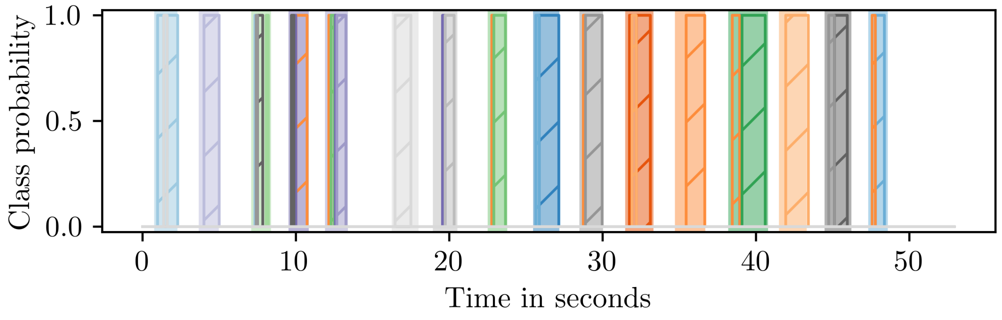


Figure 5.7: A decoding produced by an HMM decoder. The shaded regions in the background denote the ground truth.

A that no two gestures transition directly from one to another, there is always a `<blank>` in between, and the probability of staying in the same state is high because a gesture is at least a thousand frames long. So we define the $A_{i,17}$ entries, the transition probability from gesture i to `<blank>`, as the proportion of frames belonging to class i that transition to `<blank>` and $A_{i,i}$, the self-transition probability, as $1 - Ai, 17$. The transition probability from `<blank>` to any of the gestures is the proportion of gesture frames following blank frames and its self-transition probability is the complement of that. Finally, we choose π as the stationary distribution of A.

5.7 Hidden Markov Model Segmentation

Figure 6.3 shows that an HMM decoder is able to recognize the points of activity in a sequence of local classifications and is also reasonably accurate in decoding them into the correct class. However, there are a lot of spurious labels mixed in with the correct labels. The idea of HMM segmentation is to divide the decoding process into two parts. First, an HMM with just two states, `<gesture>` and `<blank>`, segments the sequence and afterwards a second HMM produces a single label for each segment. The result of this is depicted in Figure 5.8.

The HMM segmenter is constructed in the same way as the decoder in Section 5.6 with the twist that all gestures are combined into a single hidden state `<gesture>`. When the HMM segments a recording, the probability of the `<gesture>` state is the sum of all gesture probabilities. To suppress the remaining spurious activations, we also filter out all segments that are shorter than 500 ms because we know from the dataset statistics that the shortest gesture is over a second long on average. Therefore anything shorter than half of that is most certainly erroneous.

The decoding HMM is particularly simple in that the initial distribution π is the uniform distribution and the transition matrix A is the identity matrix thus disallowing any transitions. After viterbi decoding, the segment label is $z^{(n)}$.

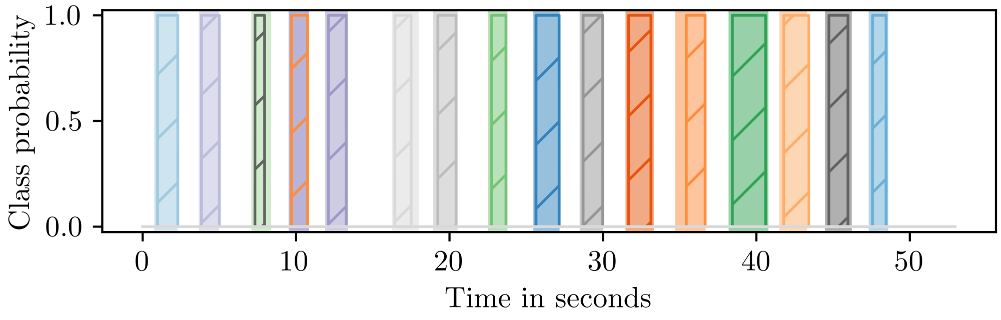


Figure 5.8: A decoding produced by an HMM segmenter followed by segment decoding

5.8 Connectionist Temporal Classification

Graves, Fernández, Gomez, and Schmidhuber solve the problems of classification and decoding with a combined method called connectionist temporal classification (CTC) that trains RNNs for sequence labeling directly [Gra+06]. CTC defines the network output as a probability distribution over label sequences and is trained by minizing the negative log-likelihood of the true label sequences over a training set. The exponential number of possible alignments between a label sequence and an observation sequence is kept in check by an evaluation strategy that is inspired by the forward-backward algorithm for HMMs.

Let $x \in (\mathbb{R}^m)$ be an input sequence of length T and $y \in (\mathbb{R}^n)$ the sequence of outputs produced by a network that classifies each frame into one of $n-1$ classes or blank. Let L^T be the set sequences of length T over the alphabet $\{1, \ldots, n\}$. Then a sequence $\pi \in L^T$ of labels of the same length as x has the probability

$$p(\pi|x) = \prod t = 1^T y_{\pi(t)}^{(t)}$$

conditioned on the observation x. Graves et al. call π a path. Define $\mathcal{B} : L^T \mapsto L^{\leq T}$ as the function that maps a path to a sequence with all blanks and repeated labels removed from the path, for example $\mathcal{B}(aa--ab-) = aab$. Then the probability of any label sequence l of length at most T can be computed by

$$p(l|x) = \sum_{\pi \in \mathcal{B}^{-1}(l)} p(\pi|x).$$

Training by maximizing the log-likelihood with this formula naively would require you to sum over all paths corresponding to a label and “in general there are very many of these” (Graves). However, the combinatorial explosion can be avoided by dynamic programming similar to the forward-backward algorithm for HMMs. For a labelling l define the total probability of its first s labels $l_{1:s}$ at time t as

$$\alpha_t(s) = \sum_{\substack{\pi \in L^T \\ \mathcal{B}(\pi_{1:t}) = l_{1:s}}} \prod_{t'=1}^{t} y_{\pi_{t'}}^{t'}$$

where α is called the *forward variable*. Define a *backward variable* β for suffix probabilities in an analogous way. The important insight is that $\alpha_t(s)$ can be computed recursively from $\alpha_{t-1}(s)$ and $\alpha_{t-1}(s-1)$ and similarly for β. Then for any t the probability of a labelling l can be rewritten as

$$p(l|x) = \sum_{s=1}^{|l|} \frac{\alpha_t(s)\beta_t(s)}{y^t_{l_s}}.$$

From this formula the gradients with respect to the network outputs can be derived.

Graves et al. give several methods for decoding the output of a CTC-trained network where decoding means finding the labelling $l \in L^{\leq T}$ that maximizes $p(l|x)$. The problem is deemed intractable, though they give several approximation methods. The first is called *best path decoding* and chooses $l = \mathcal{B}(\arg\max_\pi p(\pi|x))$. This corresponds to setting π_t to the class with maximum probability at each timestep. A quick glance at Figure 5.9 reveals that this would lead to many spurious labels on our dataset. Another method is called *prefix-search decoding*, which iteratively expands a prefix tree to find the most probable labelling. However, its worst-case runtime is exponential in the length of the input. Because of the length of our inputs, we decided to use the decoding methods given in Sections 5.6 and 5.7 on the CTC output.

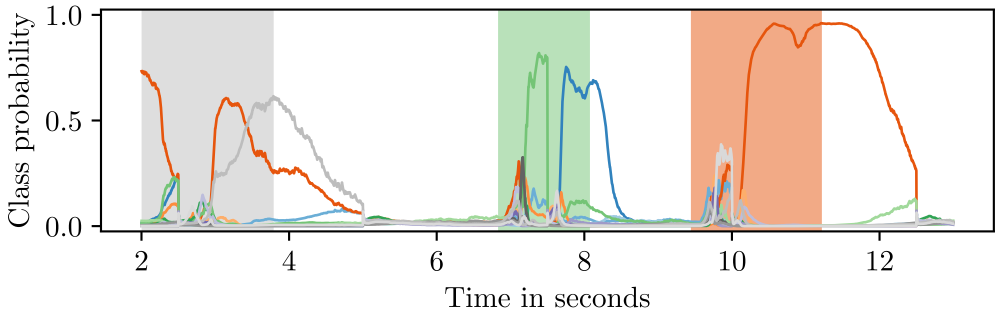


Figure 5.9: Class probabilities attained with CTC

Figure 5.9 plots the output of a CTC-trained network over time on the same input as Figure 5.6 that shows results from a framewise classifier. The output is similar yet different even though both networks share the architecture and were trained on the same data. The framewise classifier easily classifies the first and third gesture but misclassifies the second one. The CTC labeling network classifies the last gesture correctly and gets both the first and second one partially correct.

In the CTC paper the authors showcase an emergent property of CTC training which is that the network would predict the correct class only in short spikes, just long enough to be decoded into the correct label sequence, instead of classifying long streaks of frames as a single class as we observe it in Figure 5.9. This most likely due to an adaptation we had to make because of the very low label to sequence length ratio of our dataset. The average recording is two million events or 24,000 gists long, yet it has only 16 labels. This led to numerical problems during the evaluation of the gradients which we overcame by repeating each label as often as there were events or gists with that label. As a side effect, the output of the network became more akin to that of a framewise classifier.

6 Results

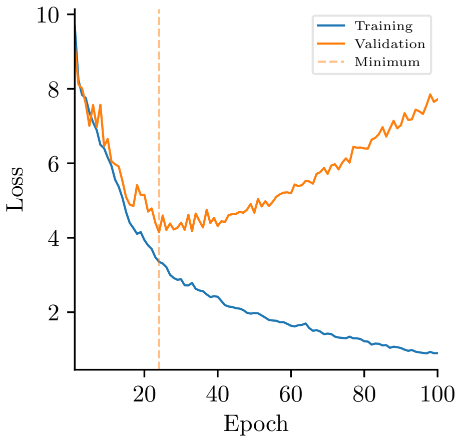
We trained neural networks on each of the three sequence representations, the preprocessed event stream, gists and CNN features from the InceptionV3 network, with each of CTC and cross entropy losses. The training progress for all combinations is depicted in Figure 6.1. The first thing that strikes the eye is that the classification of InceptionV3 features did not work at all. The training error decreases but the validation error either moves randomly or only ever decreases. This is indicative that the extracted features are not meaningful and the network can only memorize the input instead of understanding patterns. The problem can most likely be traced back to the quality of the frame reconstructions which also have to be scaled by a factor of ~2.3.

The other representations both show the normal training progression one would expect. In the beginning both training and validation error decrease almost equally, and at some point the validation error stalls or begins to slightly increase as the network overfits the training set. The latter effect is very pronounced in Figures 6.1a, 6.1e and 6.1f. The plots show that the gist representation allows a network to reach a lower average loss per epoch on the validation set with either loss function, cross entropy or CTC. The networks were trained for different numbers of epochs because of the different costs of training a network. Gists and CNN features compress a sequence of events by a factor of about a hundred in the time dimension, so subsequent classifier networks can be trained in about five minutes per epoch on a Titan X graphics card. Training on the event data takes about half an hour per epoch.

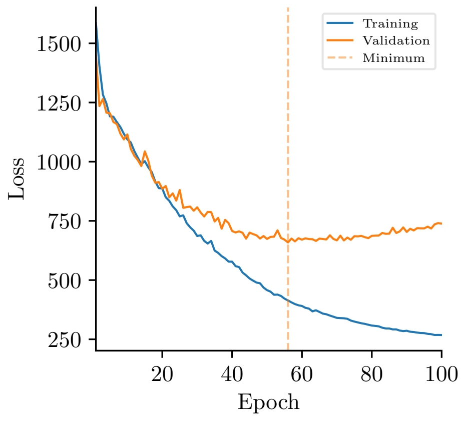
All networks exhibit a relatively large generalization gap at the point of minimum validation loss. This leads us to believe that regularization is viable here, though exploring that is outside the scope of this thesis.

Next, we selected the network with minimum validation loss in each of the six categories to classify the event sequences in the validation set. This opens us up to a kind of overfitting on the validation set because we measure the generalizability with the classifier of which we know beforehand that it performs best. In the best case we would have had a test set recorded from a person whose recordings are not part of the training data at all. However, we think that assessing the performance on the validation set is still indicative of the true generalizability because, after all, no component of the recognition pipeline has seen the validation data directly during training.

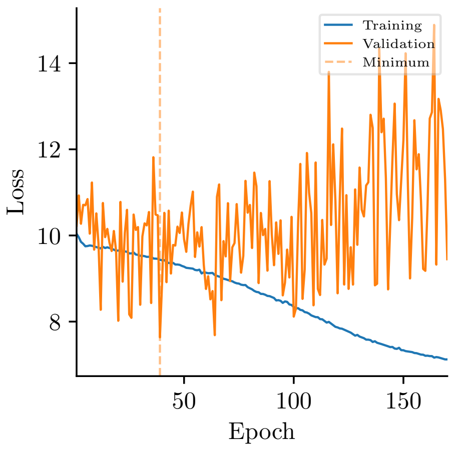
After applying the classifiers, we derived the HMM parameters from the training data as described in the previous chapter. Before presenting the results, we will have to explain how we assess recognition performance. The measure used by Graves for sequence labelling is the label error rate (LER). It is based on the Levenshtein distance between two sequences. Let (a_i) and (b_j) be two label sequences. Then the Levenshtein or edit



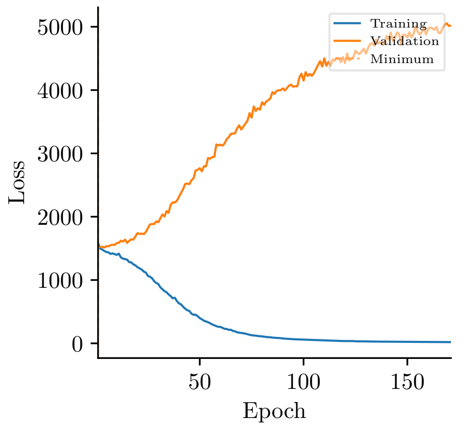
(a) Cross entropy training on gists



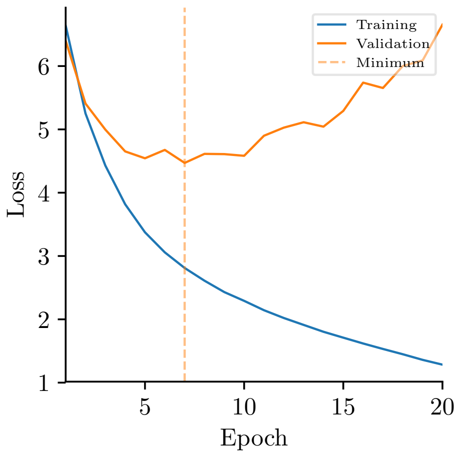
(b) CTC training on gists



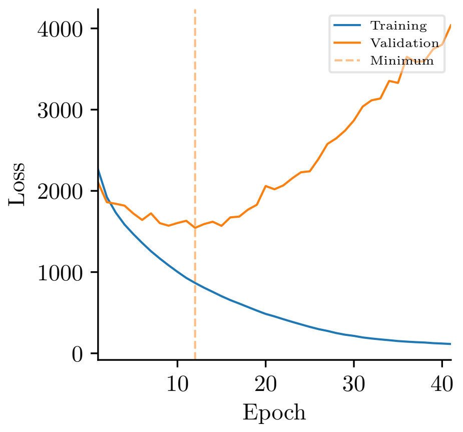
(c) Cross entropy training on InceptionV3 features



(d) CTC training on InceptionV3 features



(e) Cross entropy training on events



(f) CTC training on events

Figure 6.1: Progression of losses on the training and validation set. The dashed line marks the epoch of minimum validation loss.

distance $\mathrm{ED}(a, b)$ between the two sequences is the minimum number of insert, remove and replace operations needed to change one sequence into the other. This is illustrated in Figure 6.2 which shows the label sequence of a recording and what our system recognized side by side. The Levenshtein distance between them is 3 because two labels need to be replaced and one removed. The label error rate is defined in terms of the edit distance as

$$\mathrm{LER} = \frac{1}{\sum_{(l,l')\in Z} |l|} \sum_{(l,l')\in Z} \mathrm{ED}(l, l')$$

where Z is the set of all true label sequences l and predicted label sequences l' in a dataset, in our case the validation dataset. So the LER is the average number of edit operations necessary per true label. If the number of insert and remove operations is small, $1 - \mathrm{LER}$ can informally be treated as an accuracy of the recognition system because in that case the LER is dominated by misclassifications.

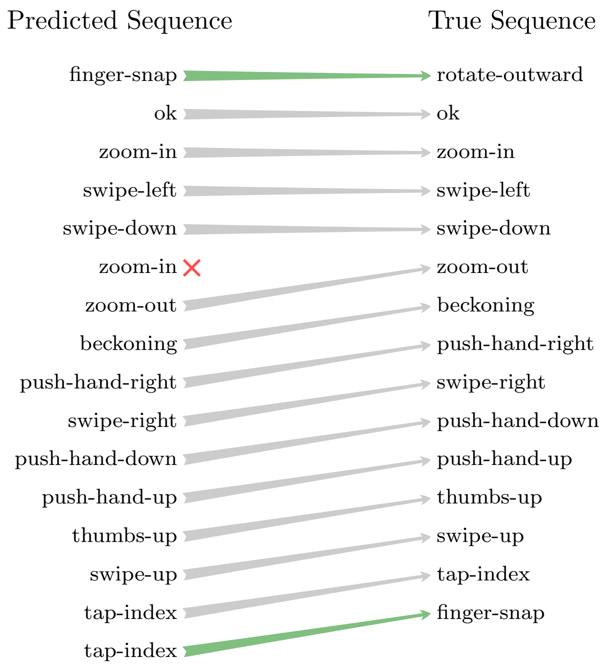


Figure 6.2: Decoded gesture sequence with a Levenshtein distance of 3 to the ground truth

Table 6.1 presents the final results across all twelve combinations of a loss function, input representation and decoding method in the form of mean Levenshtein distance and label error rate. The best result was achieved on gists with a network trained for framewise classification and decoding method with an extra segmentation step. The mean Levenshtein distance is just 2, so of the 16 gestures shown in each of the recordings, it gets 14 correct on average, which is an LER of 0.125 and an accuracy of 87.5%. The accuracy interpretation is approximately correct in this case because the network gets the number of gestures wrong only a single time.

Method	Events	Gists	InceptionV3
CE + HMM Segmentation	6.5	**2.0**	14.5
CTC + HMM Segmentation	16.0	12.5	15.25
CE + HMM Decoding	2104.0	22.0	16.0
CTC + HMM Decoding	3378.25	21.0	16.0

(a) Mean Levenshtein Distance (lower is better)

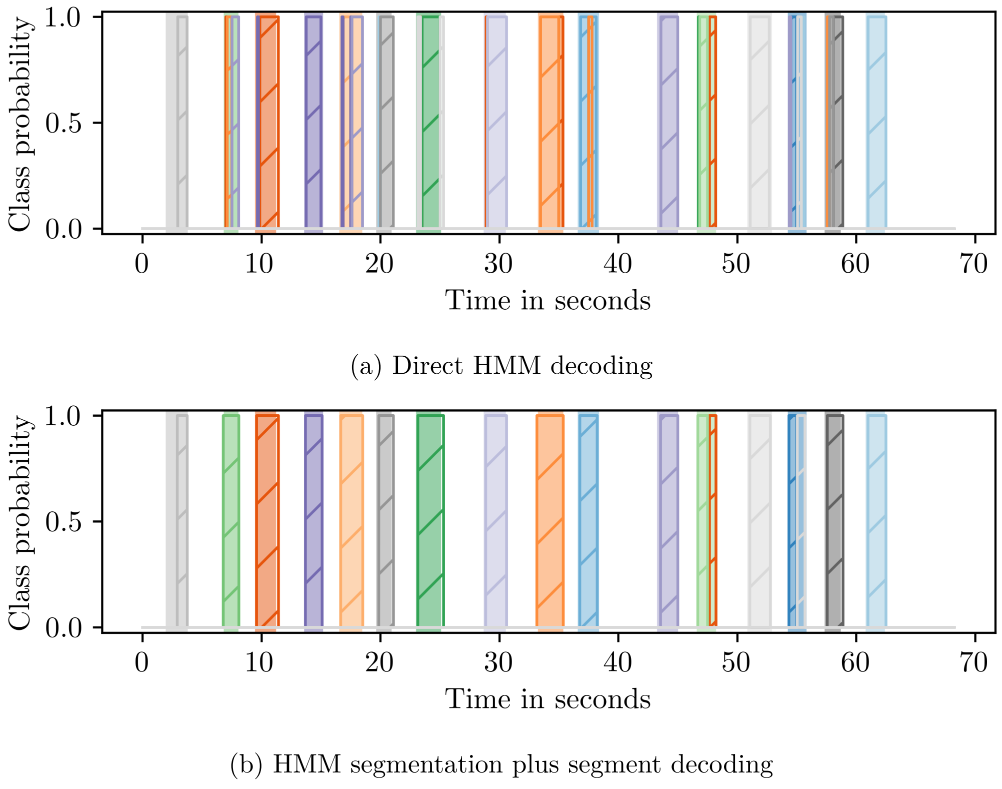
Method	Events	Gists	InceptionV3
CE + HMM Segmentation	0.406	**0.125**	0.906
CTC + HMM Segmentation	1.0	0.781	0.953
CE + HMM Decoding	131.5	1.375	1.0
CTC + HMM Decoding	211.14	1.313	1.0

(b) Label Error Rate (lower is better)

Table 6.1: Performance measured on the validation set

All other combinations perform significantly worse, though segmentation improves the label error rate in every case over simple decoding. To understand why that is, compare the plots in figure 6.3. The direct decoding method has 16 non-blank states that compete to explain the current frame classification. This produces many spurious labels that are only active for a few tenth of a second which in turn drives the Levenshtein distance up with a lot of removal operations. The segmentation-HMM combines all gestures into one non-blank class and is thus able to combine clusters of activations into one segment which is subsequently assigned a single label. The problem is only exacerbated when the pipeline is applied to the event representation. The cause for the dramatically degraded performance on events when comparing a decoder to the segmentation method is that HMMs model discrete time-series but event sequences are continuous. Also, our parameter estimation procedure for HMMs does not account for the variation in event density between gesture and blank. As a result, the HMM "sees" the gestures very drawn out which makes it easy to take the noise in the event classification (see Figure 6.4) for a true signal, if we anthropomorphize the inner workings of an HMM.

The last thing to investigate is why the performance on InceptionV3 features varies so little. However, that question is quickly answered by looking at the classifier's confusion matrix in Figure 6.5a in comparison to the confusion matrix of a framewise classifier for gists in Figure 6.5b. The training data was so uninformative that the classifier just assigned everything to the most numerous class, `<blank>`. Consequently, the output of any decoder is an empty or almost empty sequence which has an edit distance of 16 to the ground truth.


(a) Direct HMM decoding

(b) HMM segmentation plus segment decoding

Figure 6.3: Comparison of the two decoding methods on the same recording

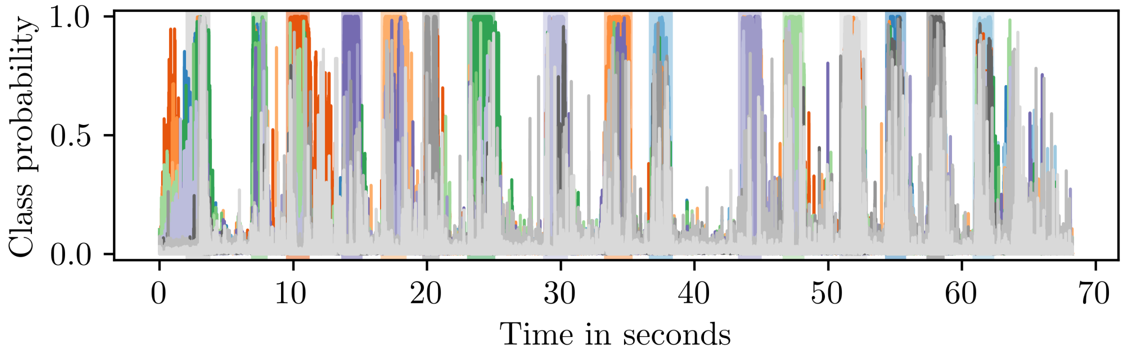
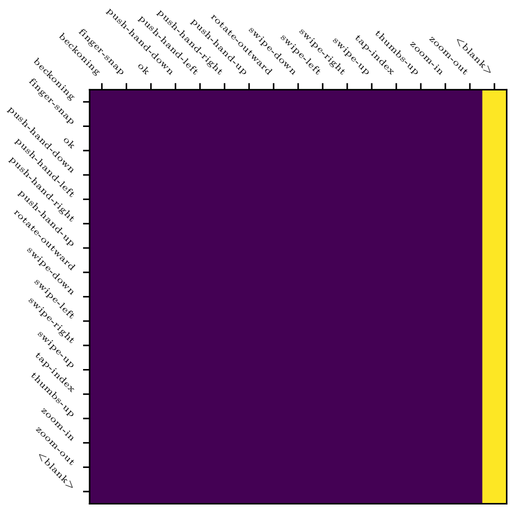
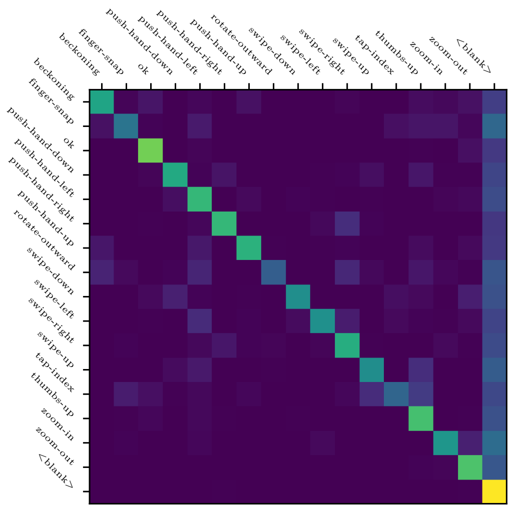


Figure 6.4: Noisy classifications produced by an RNN trained on event data



(a) Trained on InceptionV3 features



(b) Trained on gists

Figure 6.5: Confusion matrices of framewise classifiers

7 Conclusion

In this thesis, we have created a labeled neuromorphic dataset of hand gestures with more than 600 examples of 16 classes. We have compared various preprocessing measures, RNN classifiers and decoders. We were able to achieve a label error rate of 0.125 on a validation set with an autoencoder for sequence compression, a recurrent neural network for framewise classification and an HMM segmenter followed by HMM decoding. We have also tried end-to-end training with CTC and visual feature extraction with an image reconstruction method and a pre-trained CNN, though these methods produced worse results respectively did not work at all.

It would be interesting to investigate the denoising effect of the autoencoder. One could use an autoencoder to reconstruct the input sequence and empirically assess the noise level compared to the original data by replaying it in the segmentation program. Another question is if the autoencoder can hallucinate data in the same way that people use recurrent neural networks to generate texts that read like Shakespeare.

There are several ways in which the recognition performance of this system could be further improved. One idea would be to increase the information content of the learned representations at times of low event density. At the moment, we reset the autoencoder's state to zero between each time window. The effect, as seen in the t-SNE plots of the representations, is that time windows with very few events have so little information that they are not distinguishable. One might be able to improve this by applying the autoencoder in a rolling fashion by not resetting the hidden states between time windows. This could help in classifying stretches of time in gestures of low activity, e.g. the turning point of a swiping gesture.

Another idea would be to use a bidirectional neural network so that the subsequent fully-connected layers can take past as well as future context into account and avoid the phase of confusion at the beginning of a gesture that we have seen in the Figure 5.6.

The decoding process might be improved by turning to higher-order Markov models that can incorporate requirements like a minimum length of a hidden state directly into the model instead of having to post-process the decodings and segmentations.

Of course, recording more training data would surely help. When we went from the 25-gesture dataset to the 16-gesture dataset with double the amount of data, we saw a significant improvement in performance and we would expect the same from more data. Having more subjects would also improve the generalizability of the system and eventually one could reserve the complete set of recordings fro a single subject as a test set.

Finally, the system could be extended to online recognition. In theory, all the components can be run in a streaming context, except for a possible bi-directional extension to the classifier. However, even these can be run quasi-online by introducing a small delay.

List of Figures

List of Tables

Bibliography

[Ahn+11] E. Y. Ahn, J. H. Lee, T. Mullen, and J. Yen. “Dynamic Vision Sensor Camera Based Bare Hand Gesture Recognition.” In: *Computational Intelligence for Multimedia, Signal and Vision Processing (CIMSIVP), 2011 IEEE Symposium on*. IEEE. 2011, pp. 52–59.

[Bar+16] F. Barranco, C. Fermuller, Y. Aloimonos, and T. Delbruck. “A Dataset for Visual Navigation with Neuromorphic Methods.” In: *Frontiers in Neuroscience* 10 (2016).

[BDL16] P. Bardow, A. J. Davison, and S. Leutenegger. “Simultaneous Optical Flow and Intensity Estimation from an Event Camera.” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 884–892.

[Cho+14] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. “Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation.” In: *arXiv preprint arXiv:1406.1078* (2014).

[Cho+15] F. Chollet et al. *Keras*. `https://github.com/fchollet/keras`. 2015.

[Chu+14] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.” In: *arXiv preprint arXiv:1412.3555* (2014).

[Gra+06] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. “Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks.” In: *Proceedings of the 23rd international conference on Machine learning*. ACM. 2006, pp. 369–376.

[Gra13] A. Graves. “Generating Sequences with Recurrent Neural Networks.” In: *arXiv preprint arXiv:1308.0850* (2013).

[HS97] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory.” In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[KB14] D. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization.” In: *arXiv preprint arXiv:1412.6980* (2014).

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet Classification with Deep Convolutional Neural Networks.” In: *Advances in Neural Information Processing Systems*. 2012, pp. 1097–1105.

[Lee+12a] J. H. Lee, P. K. Park, C.-W. Shin, H. Ryu, B. C. Kang, and T. Delbruck. "Touchless Hand Gesture UI with Instantaneous Responses." In: *Image Processing (ICIP), 2012 19th IEEE International Conference on*. IEEE. 2012, pp. 1957–1960.

[Lee+12b] J. Lee, T. Delbruck, P. K. Park, M. Pfeiffer, C.-W. Shin, H. Ryu, and B. C. Kang. "Live Demonstration: Gesture-Based Remote Control Using Stereo Pair of Dynamic Vision Sensors." In: *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 741–745.

[Lee+14] J. H. Lee, T. Delbruck, M. Pfeiffer, P. K. Park, C.-W. Shin, H. Ryu, and B. C. Kang. "Real-Time Gesture Interface Based on Event-Driven Processing from Stereo Silicon Retinas." In: *IEEE transactions on neural networks and learning systems* 25.12 (2014), pp. 2250–2263.

[Liu+16] Q. Liu, G. Pineda-García, E. Stromatias, T. Serrano-Gotarredona, and S. B. Furber. "Benchmarking Spike-Based Visual Recognition: A Dataset and Evaluation." In: *Frontiers in Neuroscience* 10 (2016).

[LPD08] P. Lichtsteiner, C. Posch, and T. Delbruck. "A 128$\times$128 120 dB 15μs Latency Asynchronous Temporal Contrast Vision Sensor." In: *IEEE Journal of Solid-State Circuits* 43.2 (2008), pp. 566–576.

[Mol+16] P. Molchanov, X. Yang, S. Gupta, K. Kim, S. Tyree, and J. Kautz. "Online Detection and Classification of Dynamic Hand Gestures with Recurrent 3D Convolutional Neural Network." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4207–4215.

[Mur12] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[Orc+15] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor. "Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades." In: *Frontiers in Neuroscience* 9 (2015).

[Par+16] P. K. Park, B. H. Cho, J. M. Park, K. Lee, H. Y. Kim, H. A. Kang, H. G. Lee, J. Woo, Y. Roh, W. J. Lee, et al. "Performance Improvement of Deep Learning Based Gesture Recognition Using Spatiotemporal Demosaicing Technique." In: *Image Processing (ICIP), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1624–1628.

[RGP16] C. Reinbacher, G. Graber, and T. Pock. "Real-Time Intensity-Image Reconstruction for Event Cameras Using Manifold Regularisation." In: *arXiv preprint arXiv:1607.06283* (2016).

[Sch+14] D. Scharstein, H. Hirschmüller, Y. Kitajima, G. Krathwohl, N. Nešić, X. Wang, and P. Westling. "High-resolution stereo datasets with subpixel-accurate ground truth." In: *German Conference on Pattern Recognition*. Springer. 2014, pp. 31–42.

[Sze+16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. “Rethinking the Inception Architecture for Computer Vision.” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2818–2826.

[TLO15] C. Tan, S. Lallee, and G. Orchard. “Benchmarking neuromorphic vision: lessons learnt from computer vision.” In: *Frontiers in neuroscience* 9 (2015).