Anime Meets Reality

Or: How to Think Weebier Than The Possible

Alex Grabanski
10/8/2019

Genesis

- > Be me, around HackCWRU, 2018.
- Am taking Computational Intelligence I and ML
- Watched a few anime shows, mostly usual stuff like Cowboy Bebop, DBZ, and Cory in the House
- Honestly kind of depressed with the state of the real world.
- What to do?

Freudian Theory of The Mind

Realization: Deep down inside, *everyone* wants to be a cute anime girl.

Seriously

Like Actually

How Can You Not Want To Be This?

A Political Revolution

Political candidates *always* promise to make anime real

I will <u>succeed</u> where they have <u>failed</u>

BERNIE SANDERS

WILL MAKE ANIME REAL

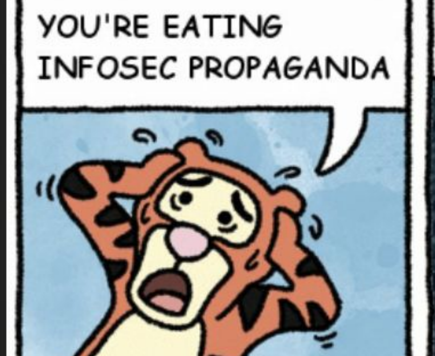
Making Anime Real By Making The Real Anime

Idea: Create an *augmented-reality* phone application which replaces all people in a live video feed with pre-specified anime characters.

How, You Ask?

Using...

MACHINE LEARNING

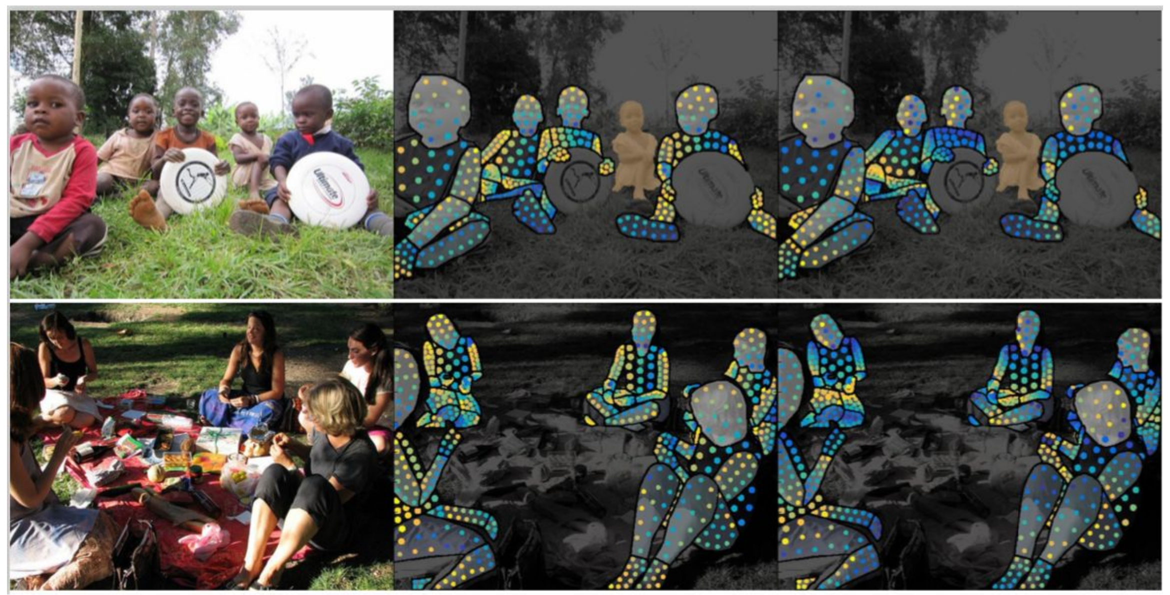


Albert Einstein: Insanity Is Doing the Same Thing Over and Over Again and Expecting Different Results

Machine learning:

Basic Proposed Pipeline

1. Annotate Live Video Feed with dense [per-pixel] pose data



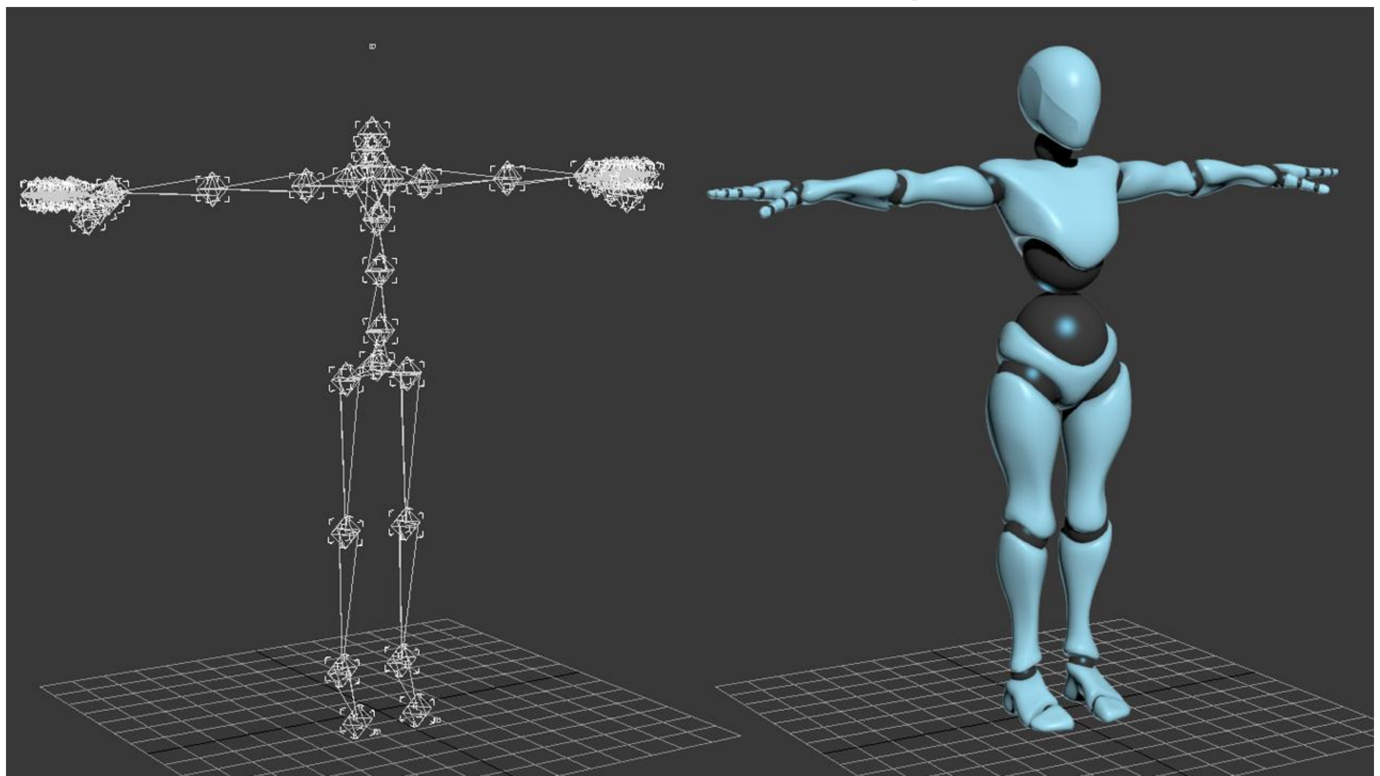
Basically a mapping from 2D video coordinate space to a normalized 3D body ”texture map” space, also with a channel with a binary mask including the people (and only the people) present in a given frame.

Basic Proposed Pipeline

2. (Re-)position a number of schematic ”dolls” (basic human body templates) within an internally maintained 3d space such that their (known, ground-truth) per-pixel pose data most closely matches the per-pixel pose data of the live video feed.

Basic Proposed Pipeline

3. Read off the transforms on the doll model bones, and transfer those to a 3d anime figure so that it matches the dimensions and pose of the doll.



Basic Proposed Pipeline

4. Render cel-shaded 3d anime figures against a transparent background, and composite that onto the live view.

DukTape9001 (HackCWRU Repo)

Four d00ds.

48 Hours.

A destiny to fulfill that is grander than they can imagine.

HackCWRU Implementation

Used OpenPose (Python/C++ library)

Yields 2D Keypoints per person*frame

Slow even on beefy computers (~4fps)

Just keypoints, **no dense pose info**

Extracting 3D pose takes some work

We tried to extract 3D poses using some hacky estimation of skeleton bone lengths and some trigonometry

Result of HackCWRU Implementation

Not very far from the schematic diagram on the right.

Really.

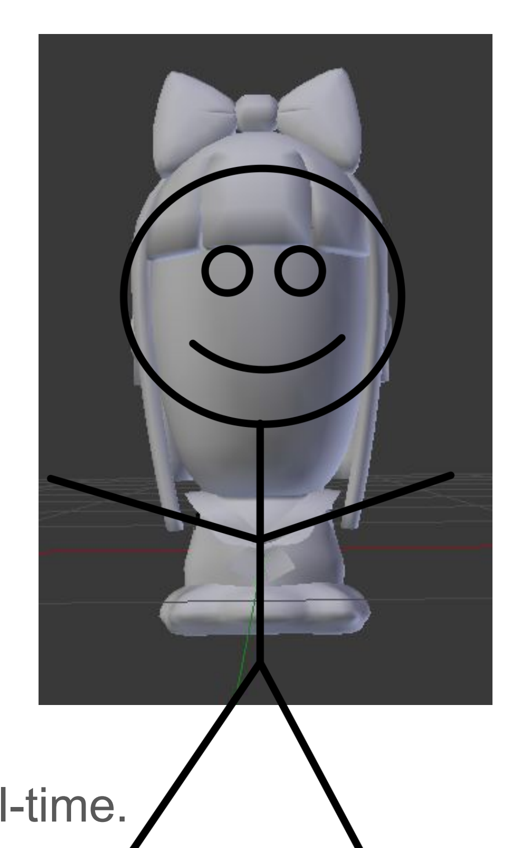
I wish I still had videos. It was *godawful*.

Everything was confined to a 2d plane

And it only tracked the very endpoints of arms and legs

(Elbows and knees be damned)

Also 4 frames per second on my desktop -- far cry from real-time.



Luckily...

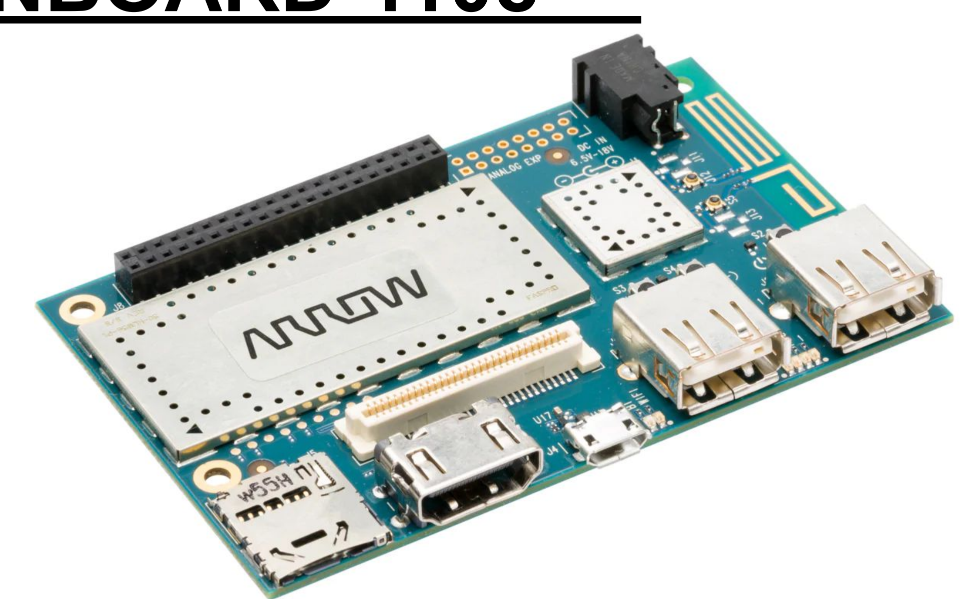
While my boi Coby and I were working on making anime real visually,

Our other teammates made an anime voice changer running on the…

QUALCOMM DRAGONBOARD 410c ™

It was the only project that used it.

As a result, we won a prize.



And For Our Prize...

QUALCOMM DRAGONBOARD 410cs ™

Imaaaaaaaagineeeeeeeeeeee

The HackCWRU implementation didn't work out, but I could see that the task was at least *possible*, if just barely out of the reach of current technology.

Think *Beyond* The Possible

Although, it wasn't enough to just make anime real… No.

I wanted to also constructively prove that (market) socialism could work.

Worker's co-operative:

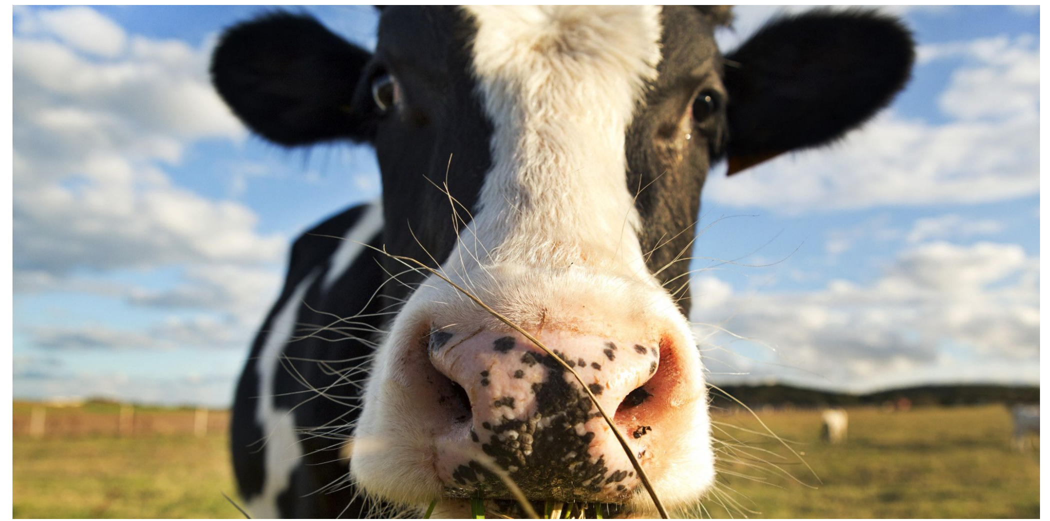
Every worker has equal power to influence the direction of the firm.

Market Socialism:

Economic system where co-operatives dominate over other kinds of biz entities.

Home Again, Home Again

‘Sconsin



(pictured above: some strange kind of dog)

(pictured on right: typical Wisconsinite fridge)

The Catch?

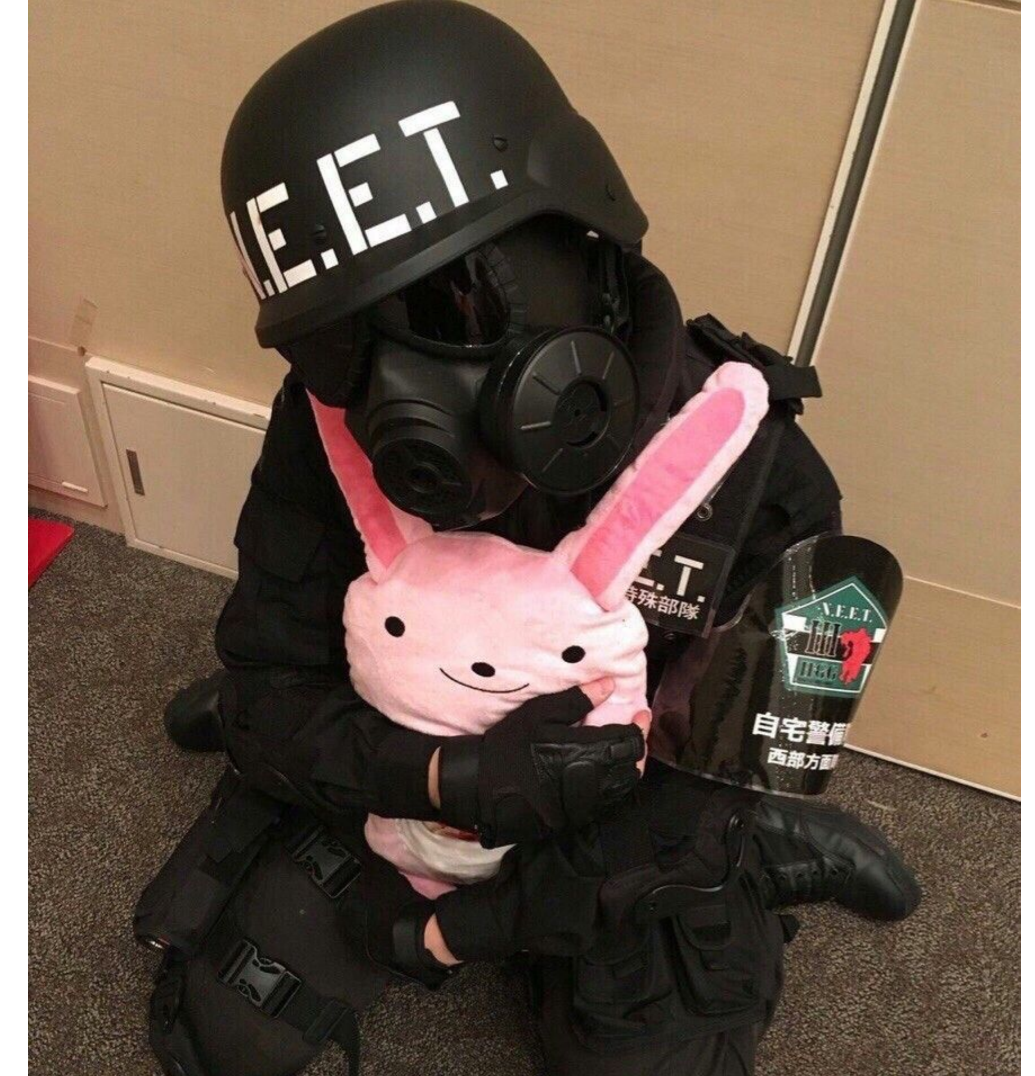
Living With Parents,

Technically a NEET,

Also a weeb NEET at that.

Ability to take self seriously: 0

But hey, free housing.



Gettin’ Techy With It

Now onto the technical details...

What is a CNN?

Using them for pose detection?

How did you get the BIG DATA?

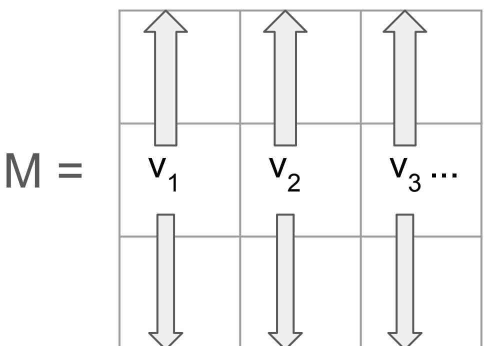
How do you use Tensorflow?

Neural Networks - Background

Some Linear Algebra: Suppose we have a full-rank NxN matrix M,

Written column-wise as $[v_1 v_2 v_3 ...]$

M =



Then, $M e_i = v_i$ for any i,

Where e_i is the ith standard basis vector

[0, 0, 0, …. 0, 1, 0, …. 0, 0, 0]

ith position

Morally speaking:
The columns of a matrix describe what the standard basis vectors map to under multiplication.
Once that's determined, the behavior of M on all other vectors is describable using linearity.

Neural Networks - Background

More Linear Algebra: In the situation from the previous slide, let's consider $\mathbf{M}^{-1}$

$\mathbf{M}^{-1}\mathbf{v}_i = \mathbf{e}_i$ …Okay, what's the point?

Well, if we think of the $\mathbf{v}_i$'s as "features we want to recognize",

Then $\mathbf{M}^{-1}$ is the linear transformation which takes any vector $\mathbf{v}$ and packs

"the degree of $\mathbf{v}_i$-ness" of $\mathbf{v}$ into the $\mathbf{i}$th coordinate of the output.

Morally speaking: Linear transformations can re-arrange vector spaces into a format where each coordinate corresponds to a "feature".

Neural Networks - Background

One issue with this intuition:

Can’t express prior knowledge of feature rarity

E.g: Looks like gold?

Probably not actually gold.

It’s *probably* just some dumb shiny rock.

Solution: Add a *bias* term.

Previously: **Mv** ⟹ Now: **Mv + b**

Previously: **Linear Transformations** ⟹ Now: **Affine Transformations**

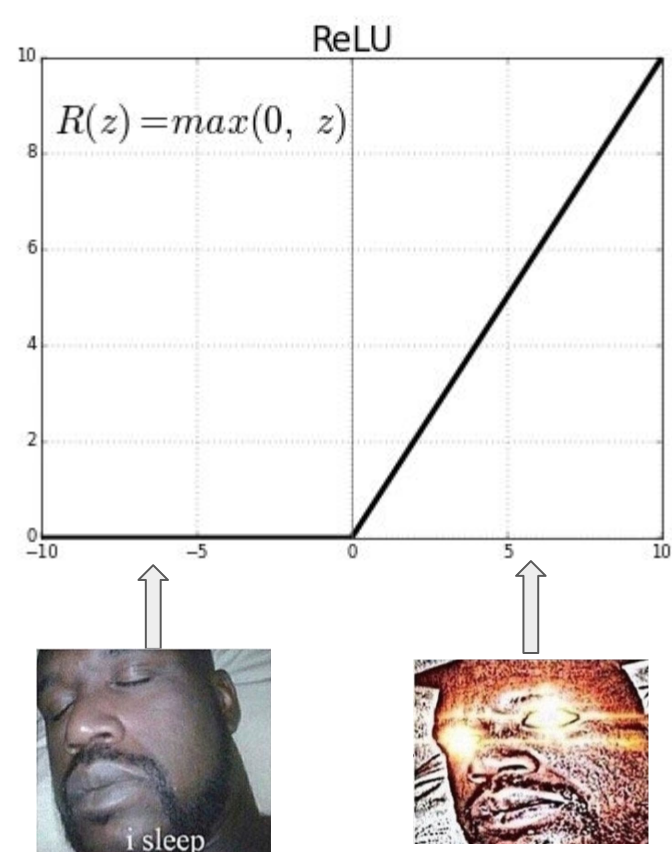
Neural Networks - Background

Another issue with this intuition: We can't *forget* negative information about features

E.g: If something is very much not a cat, we wouldn't say it's a negative cat.

Exactly this kind of thing would happen if we used affine transformations alone.

Solution: Run each coordinate through a nonlinear activation function, like ReLU



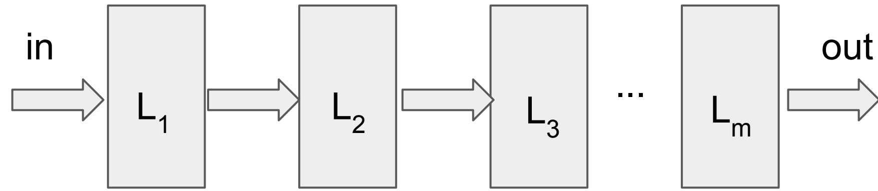
(Simple) Feedforward Neural Networks

Call the composite function $\boldsymbol{\sigma}_i \circ \mathbf{A}_i$ the ith *layer* $\mathbf{L}_i$,

Where $\boldsymbol{\sigma}_i$ is some activation function, and $\mathbf{A}_i$ is some affine transformation

Feedforward neural net: $\mathbf{L}_m \circ \ldots \mathbf{L}_2 \circ \mathbf{L}_1$

Morally:
"Recognize $\mathbf{L}_1$-level features
Use those to recognize $\mathbf{L}_2$-level features
And use those to recognize $\mathbf{L}_3$-level features
And so on, and output $\mathbf{L}_m$-level features.



(2D) Convolutional Neural Networks

Just feedforward networks

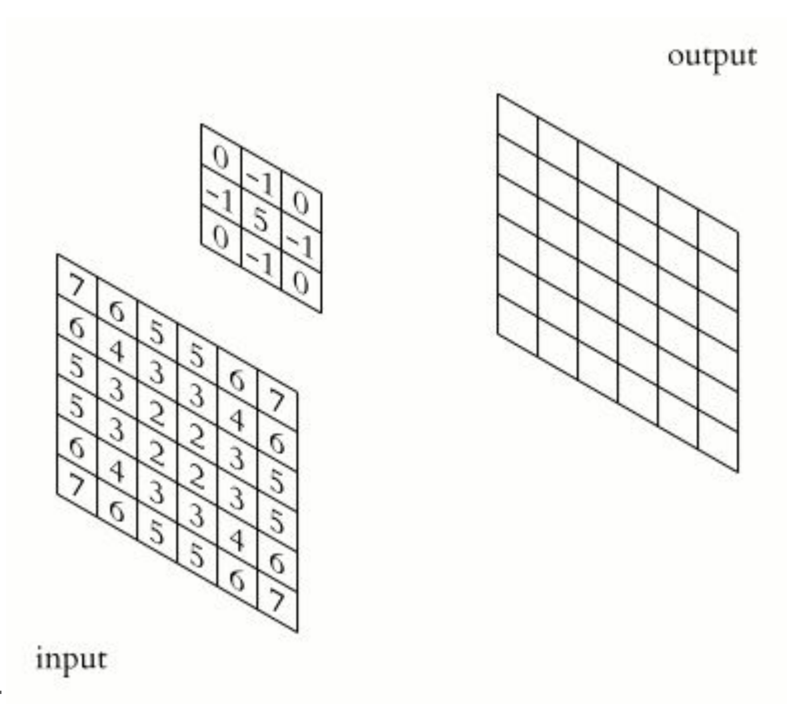
with a <u>specific</u> kind of linear transformations

<u>Slide a pattern to recognize over the image</u>

(Expressed as a “convolution kernel”)

Each position in the result is set to how closely

the pattern matches at that position in the input



CNNs: Feature Maps

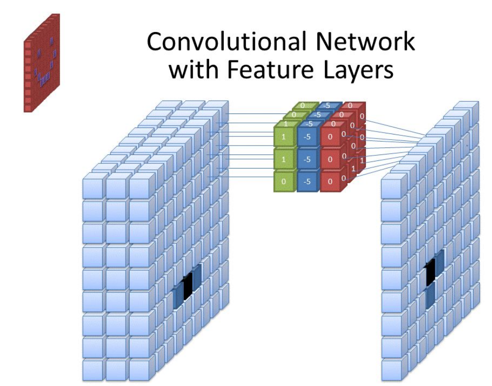
Previous animation: grayscale (single-channel) images, and a single *feature*.

<u>More generally</u>: Deal with stacks of images, one channel per *feature*

E.G: Input could be “Red/Green/Blue” at each position in the image

Output could be “Catness/Dogness/Car-ness/Boat-ness” at each position

Call each channel of each intermediate stack of images a “*feature map*”.



Note: From this intuition, we can see that we should apply our biases uniformly upon each channel to express relative rarity of features.

CNNs: Training

Convolutional Neural Networks: Just glorified iterated image pattern matching

...But how do we know what patterns to look for?

Supervised learning: Have a collection of known input/output pairings: *Data Set*

Training set: A subset of the data set that we try to fine-tune our patterns to.

Validation set: A subset of the data we use to estimate when to stop fine-tuning.

Evaluation set: Subset we use to estimate goodness-of-fit outside of the training set.

CNNs: Training With Stochastic Gradient Descent

1. Initialize convolution kernels randomly
2. Jibble the parameters a bit so they work a bit better on the training set
3. Keep jibblin’ on step 2 with the jibblin’ path that helps the training set ‘till we don’t get any better on the validation set.
4. Evaluate how good our final patterns are.

Yes, it really is that stupid.

But we don’t know anything much better.

CNNs for Pose Detection

Two Primary Output Formats:

1. Keypoint Heatmaps

   Stack of feature maps describing “probability of keypoint” at each pixel

2. Segmentation + Dense pose map

   One channel which indicates where people are in the image

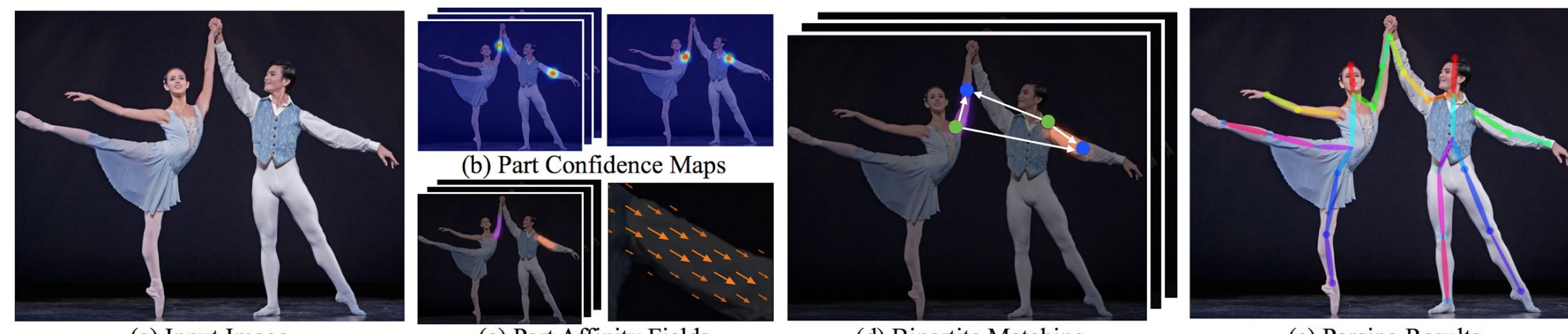
   Two to three other channels for expressing body surface positions

What OpenPose Does

Output from CNN: Keypoint heatmaps and *Part Affinity Fields*

Part Affinity Fields: Basically vector fields between adjacent body keypoints

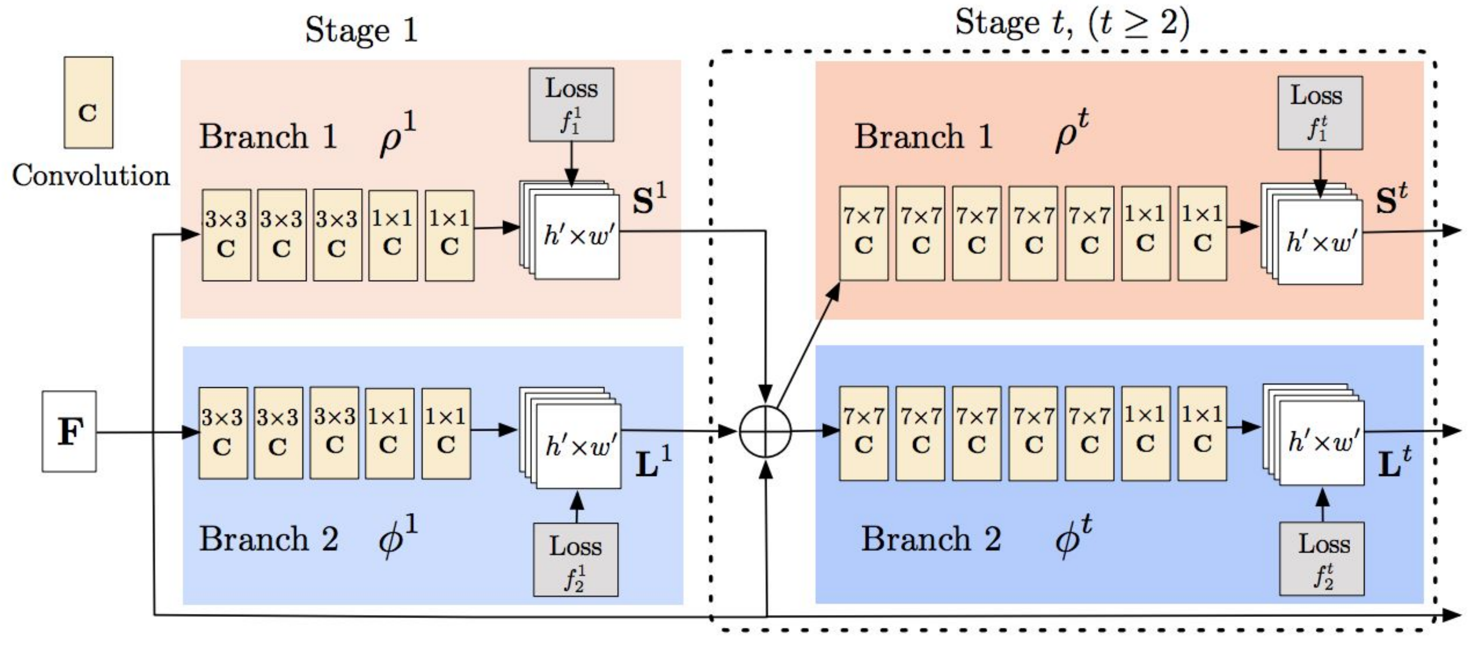
Allows them to do pose detection on *multiple people* in the same image



(a) Input Image (c) Part Affinity Fields (d) Bipartite Matching (e) Parsing Results

What OpenPose Does: CNN Architecture

Literally just two long CNNs (one for keypoint maps, one for part affinity maps) duct-taped together in a few places.



So… You Used OpenPose, Right?

No!

Why?

Well, the technical issues encountered during HackCWRU, but also...

Licensing.

OpenPose’s License

Dual-licensed -- commercial, and open source for research purposes :D

But...

Non-Exclusive Commercial License

Here is a copy of the license template: https://flintbox.com/file/download/13057

Important points from the license:

- The non-exclusive commercial license requires a non-refundable $25,000 USD **annual** royalty.

non-refundable $25,000 USD **annual** royalty.

e $25,000 USD **annual**

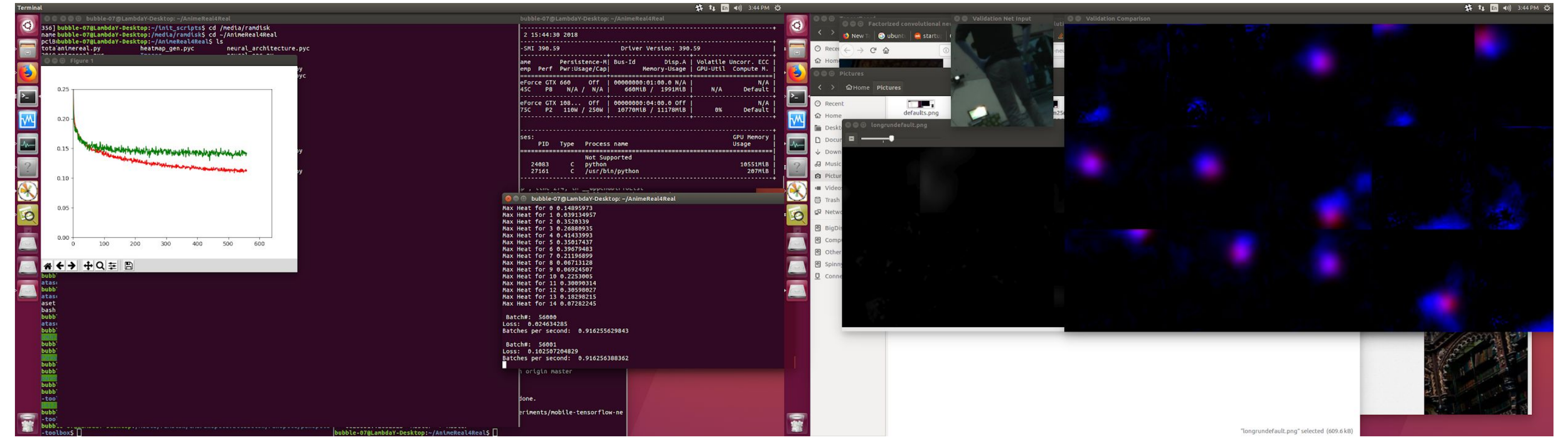
Needless to say, **screw that**.

So… You Clean-Room Re-engineered OpenPose?

No again.

Initially, I did try to use the “keypoint heatmap” output format.

However, there’s one big problem with this approach...



Computational Complexity of CNN Evaluation

Multiplying a nxm matrix by a vector (pragmatically):

O(nm)

Applying a cxc convolution kernel with n input channels and m output channels:

$O(c^2nm)$

More Channels: More expressive, but *much* slower

Fewer Channels: Faster, but carries less information

18 Body Keypoints with Part Affinity Maps? Need like 56 feature maps at each layer.

Not at all practical for mobile devices.

DensePose To The Rescue!

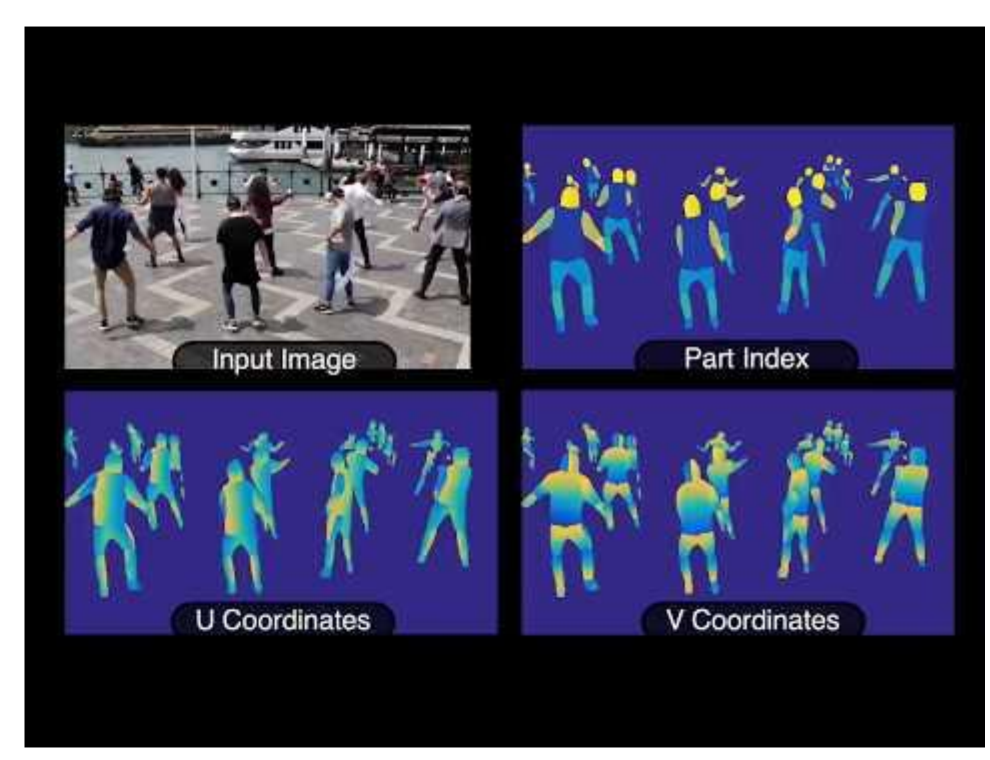
Feb 2018 FB Research Paper

Good-looking results,

CNNs still monstrously-sized

(Hundreds of layers)

But there's at least room to shrink!



CNN Architectures For Pose Detection

Previous Slides: All Pretty Simple Architectures -- Good baselines!

But the *state of the art* in 2016 was actually Stacked Hourglass Networks

Downscaling step: Scale down feature maps by power-of-two-side-length reductions

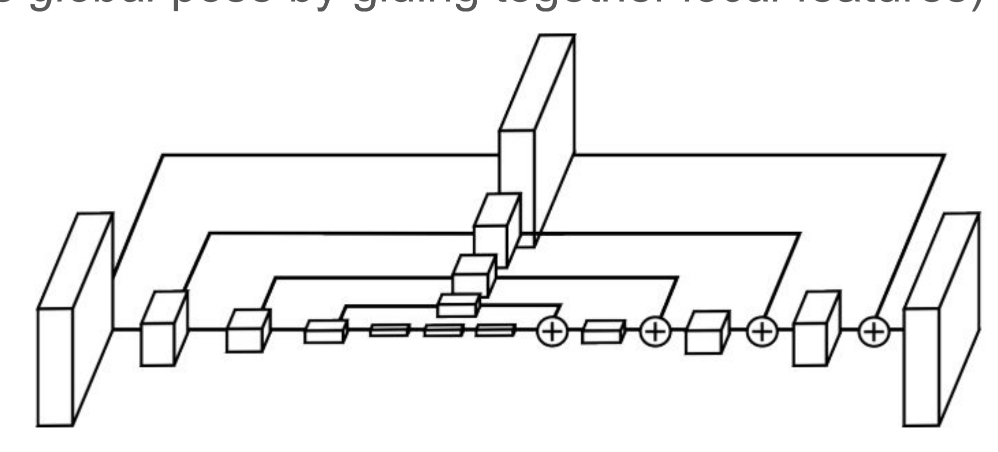
Intuitively: “*Bottom up*” pass (describe *global* pose by gluing together *local* features)

Upscaling step: Just the opposite

“*Top down*” pass

(*global* info -> *local* info)

Information loss? Nah, uses residuals



My Idea: Scale-Convolutional Neural Networks

Why not keep feature maps in *all scales* at all layers?

Better, why not use the *same maps* on all of them?

Me trying to keep SCNNs a trade secret (2018, Colorized)

⇒ loss

(could also do intermediate supervision, I suppose -- would work the same as usual)

two variants: - All wts same down columns (conv variant)
- Possible differences (fc variant)

SCNN connectivity diagram (August 2018)

Scale-Convolutional Neural Networks

Fundamental idea: Scale-convolutional kernels.

Input: a **4r x 4r x L** set of feature maps, a **2r x 2r x L** set, and a **r x r x L** set

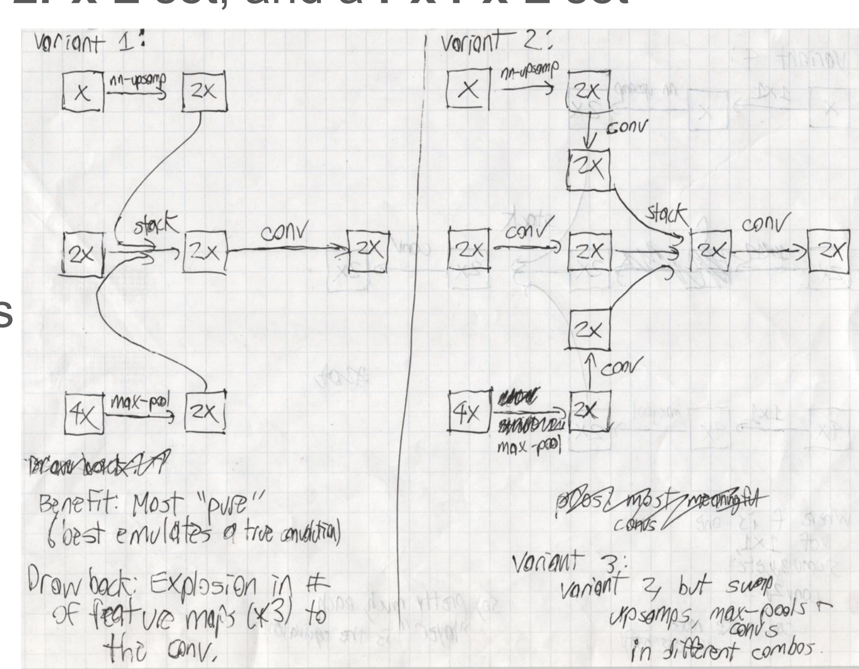
Output: A **2r x 2r x L** set of feature maps

Also called them “*trapezoids*”

Same one applied to each such triple of scales

Many different possible designs

Varying efficiency/expressivity



The Gambit

I *believed* I had a technical advantage with SCNNs on mobile devices

Main Barrier: Obtaining a training dataset that may be used for commerical purposes

Training data-sets with dense pose information are not common.

Most pose data-sets are only licensed for academic use.

Honestly pretty frustrating -- many, *many* hours of searching with no real results.

To academics: *Please stop* doing garbage like this. It makes things unnecessarily hard for small players, and no more difficult for the big players in the tech market. At least dual-license datasets if you have permission to do so.

Data Collection: The Initial Plan

With no data-sets commercially available to use, I was left with no other options.

<u>Flow:</u>

1. Film people with Microsoft Kinects striking a variety of poses.
2. Capture a variety of video backgrounds to composite in.
3. ??? (Translated: find *some* way to annotate the images with the ground-truth)
4. Profit

Kinect v2 specs: 60FPS, 512 x 424 Depth Stream, 1080p Video Stream

Actually initially went with Kinect 1… Way *cheaper*, but *worse*

The Filming Set-Up: Part I

Got one green-screen

Set up around the dining room in my parents' house

They were *impressed*

But also not *pleased*

Dimensions: **8'x8'** on the wall, and around ~**6'x8'** on the floor in front of it.

The Filming Set-Up: Part I

‘Nother pic of this monstrosity

The Filming Set-Up: Part I

<u>The Main Filming Rig</u>

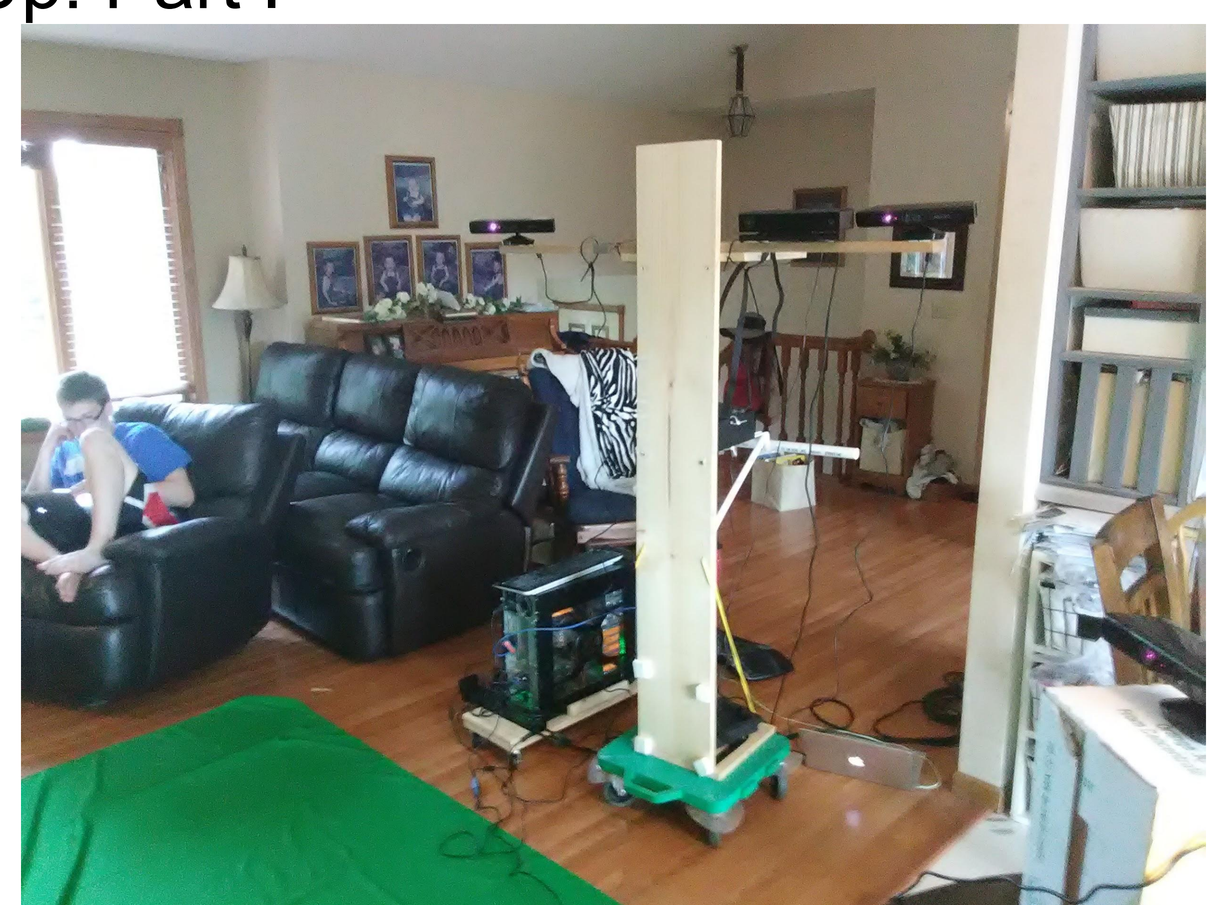
Three Kinects (1x**v2**, 2x**v1**)

Boards and dollies and stuff

Desktop computer

Also a laptop -- loads of USB bandwidth usage!

Wrote a networked application in Python so the two could coordinate filming



The Filming Set-Up: Part I

More details:

Filming rig: on-board router, mobile backup battery

Tried to use **DragonBoard 410c**

Turns out it sucks -> laptop time

Made a calibration cube to align RGB and depth images

Usually threw in a third Kinect **v1** at a very wide angle as well

Code For The Filming Set-Up: Part I

Libraries:

Libfreenect2 for Kinect **v2**,
Libfreenect for Kinect **v1**

OpenCV for image ops

Msgpack for data serialization

ZeroMQ for networking

Client-server architecture

Kinect **v2** Server main loop:

```
while True:
    frames = listener.waitForNewFrame()

    rgb = frames["color"].asarray(np.uint8)
    depth = frames["depth"].asarray(np.float32)
    rgb = cv2.resize(rgb, (int(1920 / 2), int(1080 / 2)))

    #Great -- just send those straight over the network

    #TODO: Share code with the other KinectImageServer code
    #this is awful copy-pasta right now

    #Get the current UNIX timestamp in milliseconds
    timestamp = long((time.time() + 0.5) * 1000)

    out = messages.frame_to_bytes(camera_num, timestamp, depth, rgb)

    print "Got frame at time %s" % str(timestamp)

    socket.send(out)
    print "Sent frame at time %s" % str(timestamp)
    listener.release(frames)
```

What about the Client?

Initially, a Python Client. *Easy*, but...

Lack of *good* async I/O support

Critical in this kind of application

Multiple GBs over the wire at an alarming rate

All need to be saved *to the HDD* (slow!)

Compression would just make it compute-bound

Frequently dropped entire cameras' streams

```
try:
    k = win.getkey()
    if (k == 'q'):
        break
except curses.error:
    pass
raw_bytes = socket.recv()

final_numpy_load = np.load(io.BytesIO(raw_bytes))
timestamp = final_numpy_load['time_stamp']
depth_img = final_numpy_load['depth_img']
rgb_img = final_numpy_load['rgb_img']
camera_num = final_numpy_load['camera_num']
win.clear()
win.addstr("Received image at timestamp %s on camera %s \n" % (timestamp, camera_num))
collected_rgb_imgs[camera_num].append(rgb_img)
collected_depth_imgs[camera_num].append(depth_img)
collected_timestamps[camera_num].append(timestamp)
```

Just Sprinkle Some Java On It

Threw the whole kitchen sink at it

FileChannels, Multithreading, BlockingQueues, ZeroMQ

Three communicating kinds of threads, all pulling from shared queues:

Network message listener threads

Message interpretation threads

Data writing threads

Full HDD Write speed achieved

Frames packed 100-at-a-time into a raw binary format -- rgb_i.dat and depth_i.dat

```
//Message decoding threads
Function<BlockingQueue<byte[]>, Runnable> messageInterpreter = (messageQueue) -> { return () -> {
    while (1 == 1) {
        byte[] message = null;
        try {
            message = messageQueue.take();
        }
        catch (java.lang.InterruptedException e) {
            System.err.println("Message interpreter operation interrupted!");
            e.printStackTrace();
            System.exit(0);
            return;
        }

        if (message.length == 0) {
            //Termination signal
            return;
        }
        Frame frame = null;
        try {
            frame = Frame.fromMsgPack(message);
        }
        catch (java.io.IOException e) {
            System.err.println("Frame reading failed!");
            e.printStackTrace();
            System.exit(0);
        }
        //Adjust frame number of the frame object appropriately
        int cam_num = frame.getCameraNum();
        frame.setFrameNum(frameNumbers[cam_num]);
        frameNumbers[cam_num]++;
        //Great, now put the frame into the appropriate frame queue
        try {
            frameQueues.get(cam_num).put(frame);
        }
        catch (java.lang.InterruptedException e) {
            System.err.println("Frame writer queue putting interrupted!");
            e.printStackTrace();
            System.exit(0);
        }
    }
};
};
```

```
Function<Integer, Runnable> frameWriter = (frameQueueNumber) -> { return () -> {
    BlockingQueue<Frame> frameQueue = frameQueues.get(frameQueueNumber);
    while (1 == 1) {
        Frame frame = null;
        try {
            frame = frameQueue.take();
        }
        catch (java.lang.InterruptedException e) {
            System.err.println("Frame writer queue operation interrupted!");
            e.printStackTrace();
            System.exit(0);
            return;
        }
        if (frame.isTerminationSignal()) {
            //Exit the thread, but before we do so, finalize the output streams.
            frame_write_managers.get(frameQueueNumber).close();
            break;
        }
        int cam_num = frame.getCameraNum();
        //Now, try to write the frame to file
        frame_write_managers.get(cam_num).writeFrame(frame);
    }
};};
```

Data Throughput Problems

Even with HDD write saturated, RAM still filled up under this scheme. Why?

Quick calculation: (512 * 424 + 1920 * 1080 * 0.25) * 60 * 4 * 3 bytes/sec = ~529 MB/s

Depth stream — Half-size video stream — Frames/sec, bytes/pixel, num kinects

My HDD: ~120MB/s peak write speed

Used two 500GB SSDs with ~450MB/s peak write speed in a RAID 1 configuration

(Theoretical write speed: ~900MB/s)

This was finally good enough to not drop any frames nor clog up the RAM.

The Filming Results: Part I

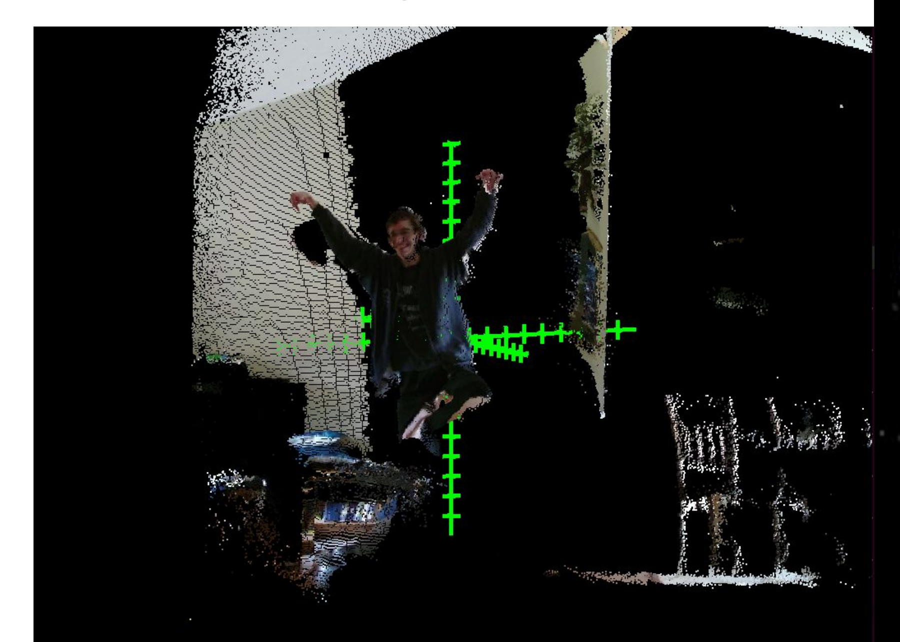
Wrote a small python application to visualize the gathered point clouds

Mostly tested on brothers*

Kinects non-synchronized

Alternating view angles

Aided intuition about Kinects

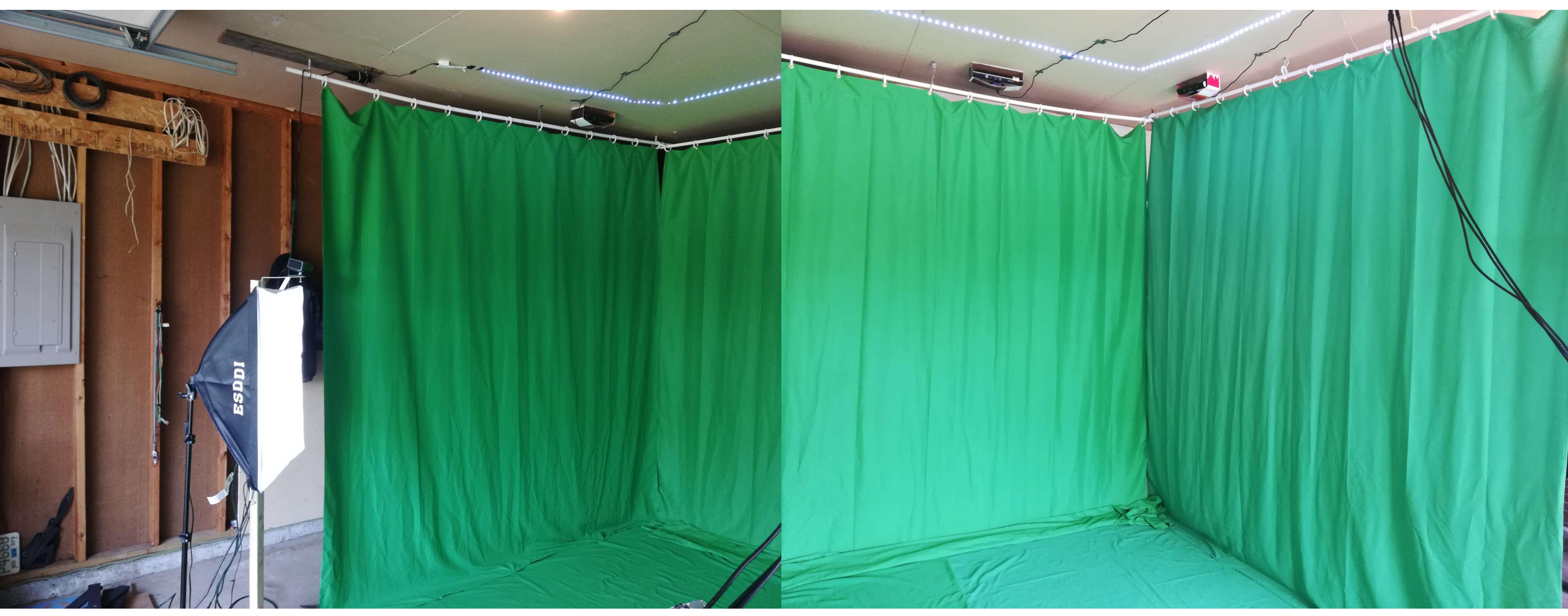


* No brothers were hurt in the filming of this dataset.

Problems With The Initial Filming Setup

- Not a very wide range of viewing angles achievable -- no green screens on the sides, so extreme angles would no longer have a green background
- Kinect **v1**’s interfere with each other -> some noticeably bad depth frames
- Using two computers and having to deal with network issues sucks
- Major inconvenience to family every time it was used
- Floor/wall green screen shared -> movements could pull it away from the wall
- Bump the cart -> need to calibrate again
- Cords to trip on *everywhere*

The Filming Set-Up: Part II



Ahhh… Much better. Although, 3x Kinect **v2** -> needed to install two USB 3.0 PCI-e cards in the floorputer

The Filming Code: Part II

Dropped the network junk, switched to a multi-process Python script

One process per Kinect

Used bgwrite to do async I/O

Achieved full write speed

(Take that, Java)

```
#Construct kinect 2 processes
kinect2_processes = []
for i in range(0, num_kinect2_devices):
    kinect2_serial = kinect2_serials[i]
    camera_num = kinect2_camera_numbers[i]
    camera_init = get_kinect2_init(kinect2_serial)
    kinect2_processes.append(Process(target=mainLoop,
            args=(camera_init, kinect2_framegetter, kinect2_getter_code, kinect2_takedown, camera_num)))

#First, kinect 2 processes, then kinect 1
processes = kinect2_processes + kinect1_processes

#Start all processes
for process in processes:
    process.start()
```

```
obj1 = bgwrite(rgb_file, rgb_payload, ioPrio=ioPriority, closeWhenFinished=True)
obj2 = bgwrite(depth_file, depth_payload, ioPrio=ioPriority, closeWhenFinished=True)
```

Also wrote some scripts to compress (massive, ~200MB) generated files

Also wrote quick scripts to play them back, for QA purposes

The Grabanski Talent Agency

Needed to find *aspiring young actors* who dream of jumping like a frog

How?

Craigslist ad?

Newspaper ad?

Facebook ad?

First, two issues to handle...

Liability

What if someone slips on a banana peel and tries to sue?

Solution: Create a legal entity to shoulder the blame, and have every person filmed sign a model release contract. [it's the American way :')]

Introducing… **Funktor Reactive, LLC** -- A Wisconsin single-member LLC

FORM **502**

ARTICLES OF ORGANIZATION
LIMITED LIABILITY COMPANY
Sec. 183.0202 Wis. Stats.

Executed by the undersigned for the purpose of forming a Wisconsin limited liability company under Ch. 183 of the Wisconsin Statutes:

Article 1. Name of the limited liability company:

Funktor Reactive, LLC

Takeaway: Starting a company is not that hard -- it's just some paperwork!

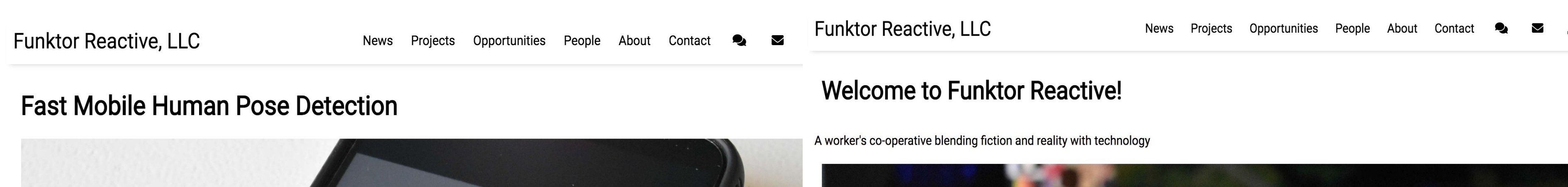
Starting a *successful* company is hard.

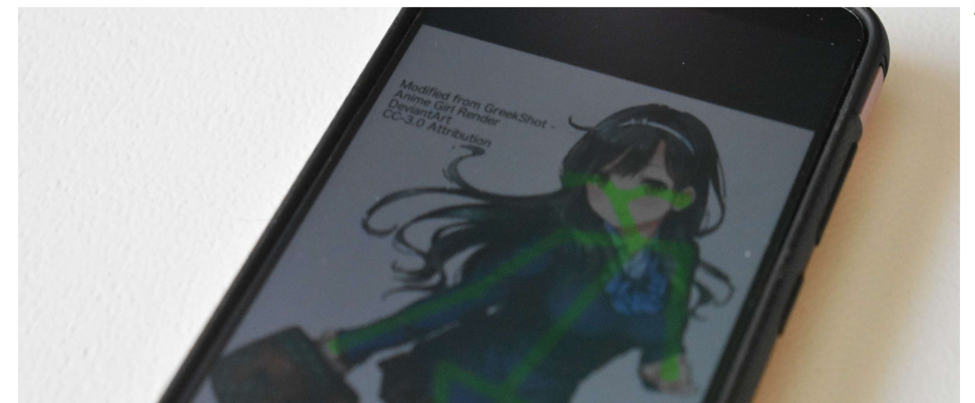
Credibility

Try walking up to a stranger and asking “Can I film you doing weird poses in 3D?”

Need a platform to explain what’s happening… Time to do some **-[w e b d e v]-**

Used Jekyll theme “Millennial” -> www.funktorreactive.com [now-defunct]





Advertising Time!

Posted in Twin Cities Queer Exchange

Knew few people at the beginning

Due to thread-bumps by comments, got about three people or so per week

Network effects quickly set in

$50 for goofing around ain't bad!

Also made some friends

Hello QE!

My name is Alex Grabanski, and I'm representing Funktor Reactive, LLC (http://www.funktorreactive.com/), a Hudson, WI startup worker's co-operative which builds technology to blend fiction and reality. We are currently looking to gather data for our human pose database. As a result, we're willing to offer short-term $50 contracts to participants who are willing to be filmed in a wide variety of fun poses. Each participant will be paid the full amount after filming 15 minutes of footage in our multi-view RGB-D filming setup. The whole process should take under an hour. We also will respect your right to privacy under our default contracting option, as we are not allowed to publicly use your likeness for e.g. advertising without your permission. No qualifications or experience necessary! We plan to contract with the first 20 applicants in a first-come-first-served manner, since our only goal is to capture a very diverse collection of sample pose/image pairs for our machine learning secret sauce to do its work properly. We are also willing to coordinate transportation to/from Hudson for low-income applicants, and will see if we can coordinate car-pooling to be more environmentally-friendly.

Interested, and want more information? Send a private message to https://www.facebook.com/FunktorReactive/ or an e-mail to funktorreactive@gmail.com.

Livin’ The Life

<u>Usual drive:</u> 30 mins -> hour each way.

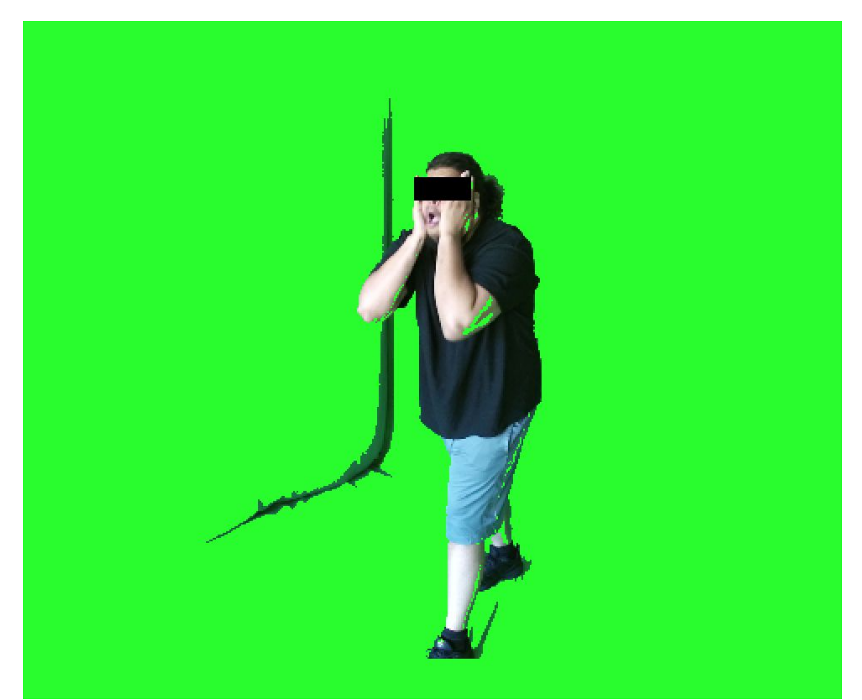
Served cold drinks during hot weather

Some great car conversations

People had tons of fun!

Couples were the best to film, since the second person to go often got their *revenge*

Overall, 10/10, $1k well spent



Still frame of participant doing “The Scream” with basic chroma-key background removal applied

Annotating The Data

Unlabeled 3D point cloud video of people doing weird things - check

How to label the data with ground-truth poses?

A Few Possible Approaches:

1. Set up an Amazon Mechanical Turk task for people to manually annotate images
2. Use some pre-existing classifier to automatically annotate the images

Obviously, option **1** would yield better data...

Machine Learning, Menial Labor, and Mental Illness

Living with OCD: One thing that is *near impossible* for me to do:

Making another person perform a task that will *bore them out of their minds*.

Was legitimately afraid that I was inconveniencing every customer service representative I ever interacted with -> prefer self-checkout, automation strategies

Making people annotate thousands of images was unimaginable to me.

So, on the automation route I go!

Automated Pose Labels

The idea: Wear a morph suit

Color-code each appendage

Color-code points on appendages

Do everything participants did

Augment the data by distorting my bodily proportions in the depth frames

Train a pose detector using depth frame/ processed color data pairings

Use the pose detector trained on depth data to annotate RGB-d participant dataset

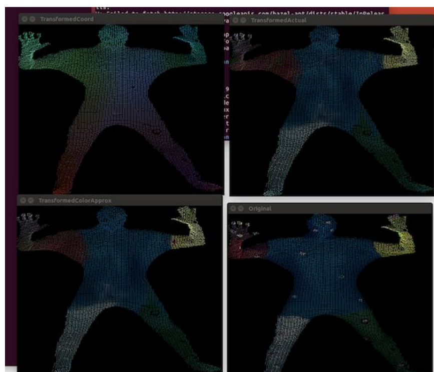
Body Templates

Common body coordinate system?

Scanned myself from a variety of angles

Manually stitched together a point cloud

Added some schematic color-coding (Python)



You may not like it, but this is what peak human performance looks like.

Randemo:

Part loss 7.88746 (in)

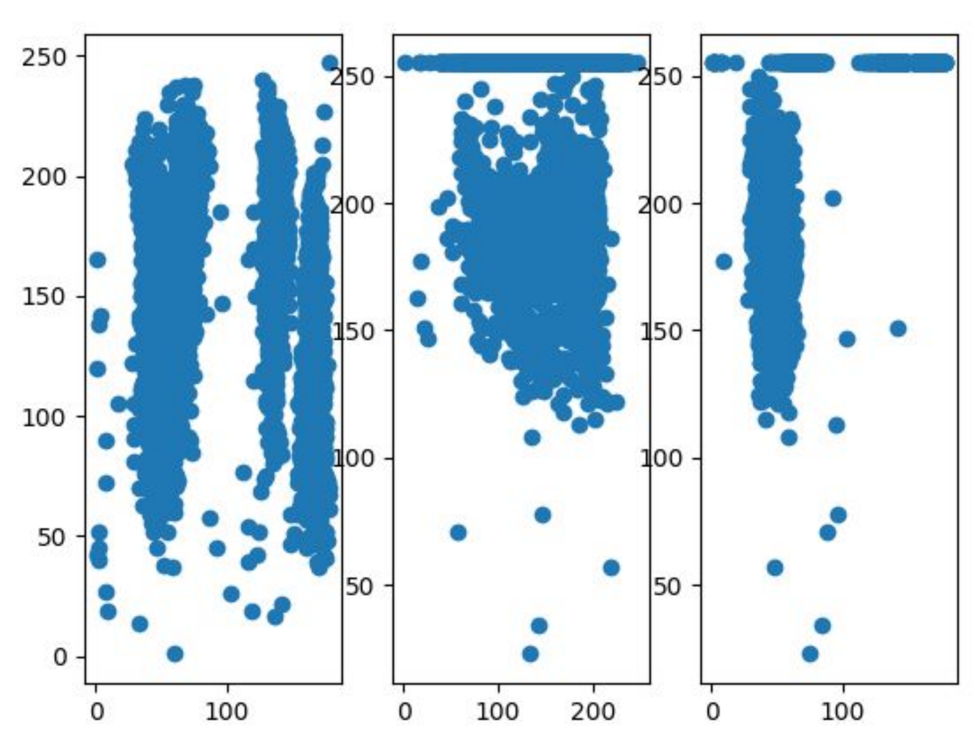
The Finer Points of Chroma-Keying

Nobody will tell you this, but it's actually a royal pain in the *** to get chroma-keying right.

In my case: spent several hours generating plots with MatPlotLib to try to figure out best linear separating hyperplanes between adjacent colors.

HSV is 3D, plots are 2D

Life sucks



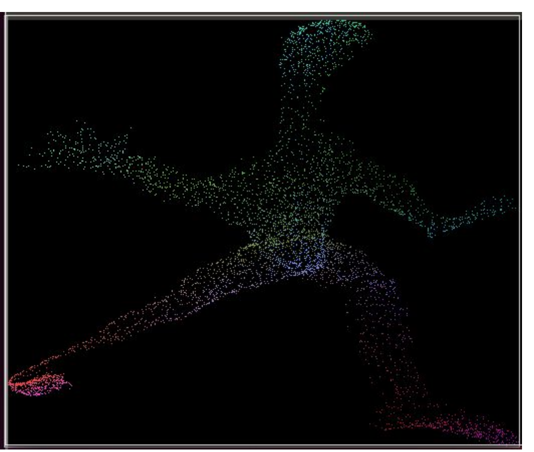
Deformable Registration of Point-Clouds

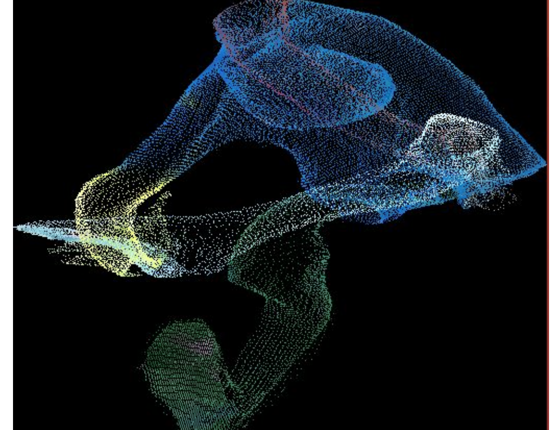
Chroma-keyed points -> accurate locations for *specific points* on my body

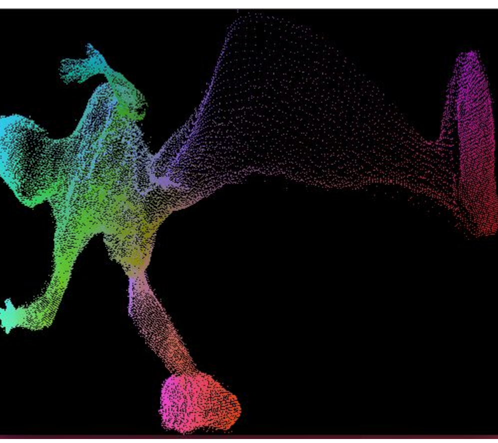
Everything else? No direct ground-truth annotation.

Problem: Using continuous mappings, align template to captured frames

This is a *deformable registration* problem. (s/o to MIM Software)







Deformable Registration of Point-Clouds

Approach: Fit a transformation from (time x template location) -> frame location.

Yields a changing embedding of the template within 3+1D space-time.

Representation of the transformation?

Use a small feed-forward neural network!

Just like before, but with a different evaluation criteria:

Total Loss = “Isometricity loss” + Point distance loss

Where “Isometricity loss” is a measure of the distortion of the template

Code for Deformable Registration of Point Clouds

In Repo as *TemporalRegister.py*

Some notes: Previous slide is simplified

Actually, we try to fit the registration

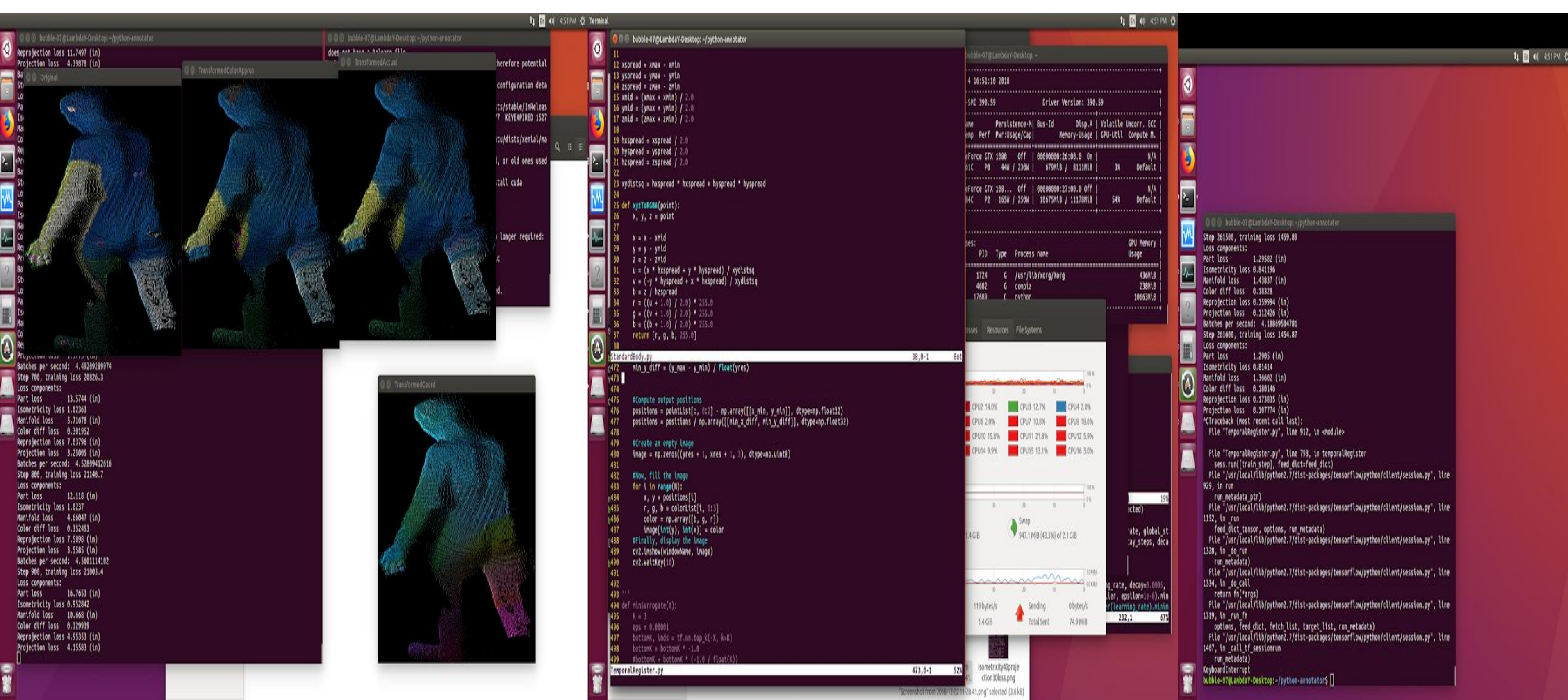
+ Its inverse!

Also, we have loss terms for color differences

when compared with the template

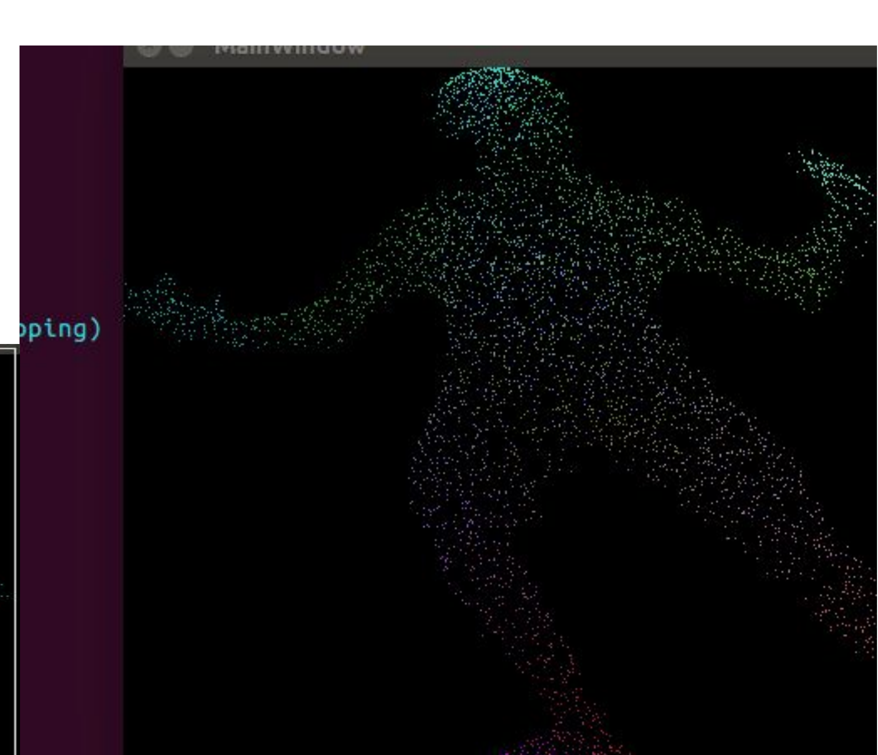
```
373
374 #A network where input vectors are used to determine rigid transformations,
375 #which are then applied to the spatial component of the input vector
376 def fluidTimeNetwork(x, reuse, namePrefix):
377     scale_fac = 10.0
378
379     x = x / scale_fac
380
381     with tf.variable_scope(namePrefix):
382         spatial_positions = x[:, 0:3]
383         #First thing's first: Figure out how we parameterize our rigid transform!
384         #Initial expansion layer
385         with tf.variable_scope(namePrefix + "ExpandInitial") as s:
386             out = fcLayer(x, fluid_network_width, reuse, s)
387         #Middle layers
388         for i in range(fluid_network_layers):
389             with tf.variable_scope(namePrefix + "FC" + str(i)) as s:
390                 with tf.name_scope(namePrefix + "FC" + str(i)):
391                     out = out + fcLayer(out, fluid_network_width, reuse, s)
392         #For the final output, take (1 + fluid_network_segments * 2) * 3
393         #linearly-transformed network outputs and interpret them
394         #as pre-translation followed by a repeating sequence of rotations
395         #and translations
396         seg_slots = 1 + 2 * fluid_network_segments
397         seg_params = seg_slots * 3
398         with tf.variable_scope(namePrefix + "FinalLinear") as s:
399             rigid_params = fcLinLayer(out, seg_params, reuse, s)
400         reshaped_params = tf.reshape(rigid_params, [-1, seg_slots, 3])
401         pre_translate = reshaped_params[:, 0, :]
402
403         out = spatial_positions + pre_translate
404
405         for i in range(1, seg_slots, 2):
406             rotate = reshaped_params[:, i, :]
407             post_translate = reshaped_params[:, i + 1, :]
408             out = rodrigues(rotate, out)
409             out = out + post_translate
410         return out * scale_fac
411
```

Deformable Registration of Point Clouds: Screenie

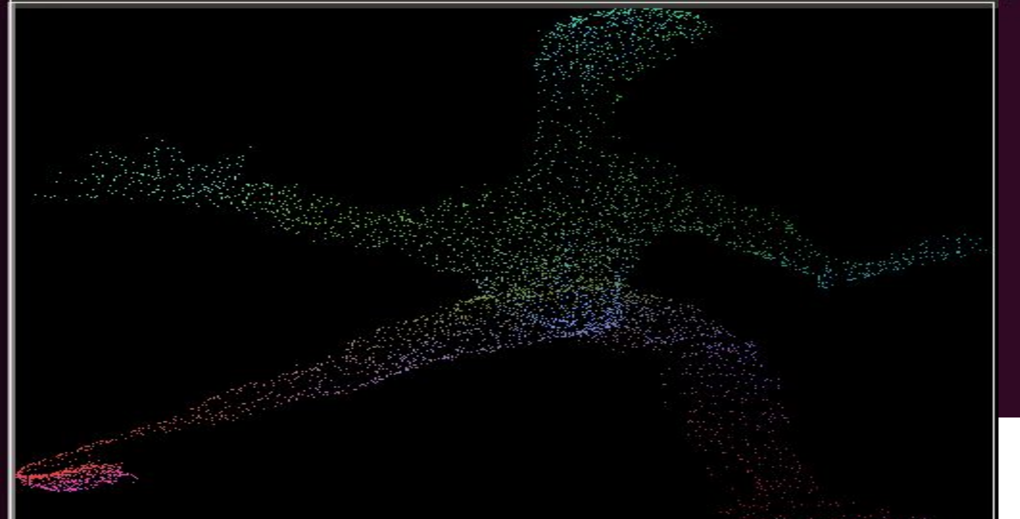


Results of Deformable Registrations:

Pretty good most of the time.



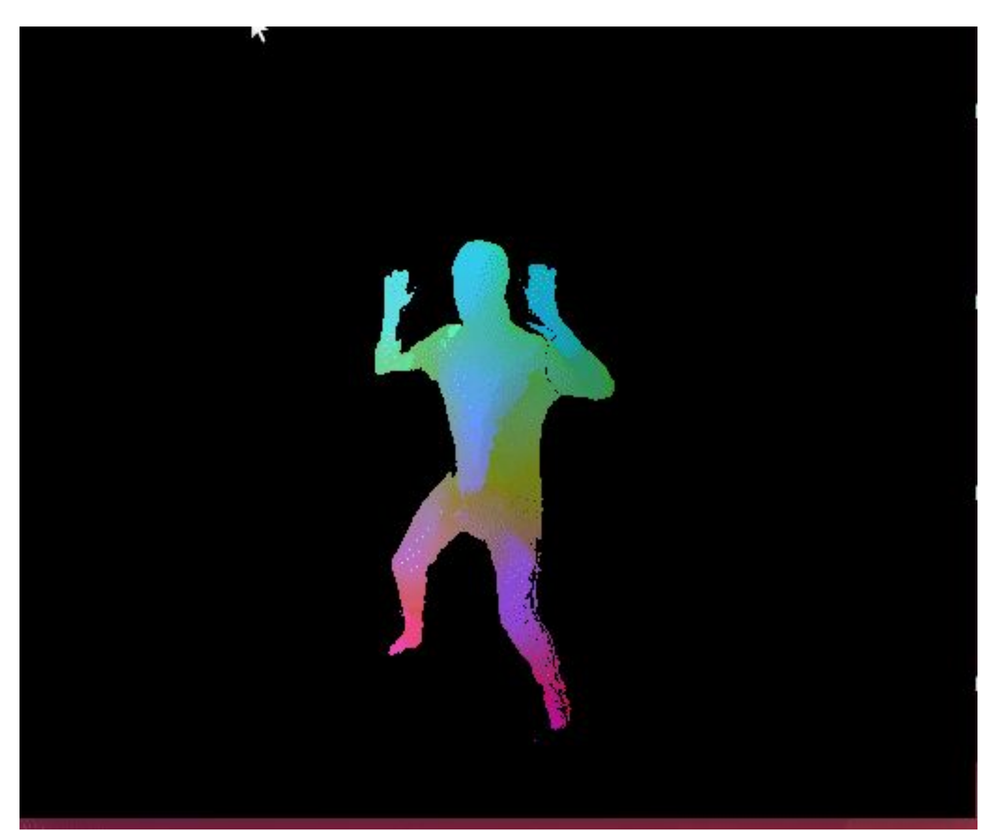
Except, of course, when it wasn't.



Annotating Morph-Suit Depth Images

With deformable registrations in hand, labeling depth images is just* a matter of finding nearest points on the (deformed) template.

Some pretty results:



Annotating Morph-Suit Depth Images: Code

The * on the previous slide?

Well, actually, did *k-nearest neighbors*

And did some post-processing

Code in *AutoLabelTFRecordSaver.py*

```
def advancedGetTemplateIndexImage(pointCloud, labels):

    try:
        #getTemplateIndexImage did things relatively simple, in
        #the sense that there, it was just a one-time assignment
        #of the closest distorted template point from points
        #in the point cloud

        #Here, we'll kinda-sorta do that, but iteratively
        #reassign points in the point cloud array to the average
        #of nearby neighbors (in template-space), weighted by probabilities given
        #by the original distance from the each point to the label cloud
        #We'll then snap those positions to positions on the template manifold,
        #take those as the new coordinates, and do it again.
        #The result is hopefully smoother and more coherent than the original,
        #and hopefully majority-takes-all for mislabelings
```

```
labelKdTree = cKDTree(labels)

cloudPointArray = np.asarray(pointCloud.getPoints(), dtype=np.float32)


K = 8
if (cloudPointArray.shape[0] < K):
    #Special case: not enough points to do the averaging!
    return getTemplateIndexImage(pointCloud, labels)

#Get the indices of a K-neighborhood of every point in the input cloud
cloudKdTree = cKDTree(cloudPointArray)
_, cloudKNeighborhoods = cloudKdTree.query(cloudPointArray, K)


labelDists, labelInds = labelKdTree.query(cloudPointArray)

#Assign probability weights based on a decreasing exponential
#of distance
#Distance (mm) which constitutes a weight reduction to 1/e of what it was
naturalDist = 50.0
probWeights = np.exp(-labelDists / naturalDist)
```

Exports results to .tfrecord data format

Body template position labels are packed in as uint16’s

-> they’re indices into the template’s point array!

The Depth-Frame Pose Detector: Network Internals

<u>Code Architecture of Depth-Frame NN:</u>

Wrote a small library in Python for NNs

Functional programming-y style

<u>Pictured:</u> Definition of network

With F feature maps, L layers, and B is the “block size” for DenseNet-like blocks (full explanation of experimentation there would completely fill a whole other talk.)

```
def generalizedHeatmapGen(F, L, B, in_channels=img_color_channels, out_channels=3, convLayerFunc=dense_sum_scale_layer, inter
    #For when this op runs...
    """Arguments:
        x: an input tensor with the dimensions (N_examples, img_height, img_width, img_color_channels)
        params: the parameters of the model, as expressed by "gen_params" above
       Returns:
        A tuple of tensors of shape (N_examples, height, width, num_field_maps)
        in decreasing order of size (in width and height), from 64x64 down to 8x8
        which stores the detection heatmaps for each body part
    """
    return identity().then(
            iteratedDownsample() #Then, iteratively 2x downsample from 256x256 to 8x8
            ).then(unsplice(3)).then(parallel( #PREPROCESSING: Separate out the 256x256, 128x128 and 64x64 feature maps
                parallel(
                    conv(3, in_channels, feats_128, stride=2).then(
                        conv(3, feats_128, F / 4, stride=2)), #256x256 feature maps get down-conved twice at stride 2
                    conv(3, in_channels, F / 4, stride=2), #128x128 feature maps get down-conved once at stride 2
                    conv(3, in_channels, F / 2) #Generate F/2 64x64 feature maps in the same manner, but stride-1
                ).then(stack_features()).then(  #Stack all of those together to get something of size 64x64
                    conv(1, F, F)       #Convolve the stacked maps to hopefully get agreement with those 32x32 and below
                            ).to_singleton_list()
                , replicate_over_list(
                    lambda: conv(3, in_channels, F), 3) #For the feature maps 32x32 and below, convolve so we can get F chann
            )).then(splice()).then_apply( #That was a mouthful! Put our new 64x64, 32x32, 16x16, and 8x8 feature maps back to
                lambda L: L[::-1]).then( #And reverse the order (8x8 to 64x64 now). SCALE CONVOLUTIONS ARE NEXT
                internalsFunc(F, L, B, convLayerFunc=convLayerFunc) #We can actually do stuff with scale-convolutions!
            ).then(replicate_over_list(
                    lambda: conv(1, F, out_channels, activation=activ_fn_relu6()), 4)).then_apply( #We need heatmaps, not F f
            lambda L : L[::-1]) #Finally, reverse again so we're in increasing order (8x8 to 64x64 heatmaps)
```

The Depth-Frame Pose Detector: Loss Functions

Scale-convolutional nets ->Have 8x8, 16x16, 32x32, … outputs

Loss function?

Iteratively downsample the target image -> Compare each against each

Return a weighted sum of differences

```
#Downsampling loss function -- this takes the mean absolute errors
#between the network's 8x8, 16x16, 32x32 and 64x64 outputs
#and iteratively max-pooled versions of the "ground truth" 256x256
#heatmaps
def downsampling_mabs_loss(net_out, expected):
    #Iteratively max-pool the expected
    expected_out = iterated_max_pool_downsample(expected, 7)[2:-1]

    #Compute pairwise mabs losses
    losses = map(lambda pair : loss_fn(*pair), zip(net_out, expected_out))

    #Multiply by downsampling weighting factors and sum
    weighted_losses = map(lambda pair : tf.multiply(*pair), zip(losses, downsample_weighting_factors()))
    return reduce(tf.add, weighted_losses, 0.0)
```

The Depth-Frame Pose Detector: Data Augmentation

AnnotationSuitDepthTrainer.py

Data Augmentation:

Makes dataset artificially “bigger”

```
if (AUGMENT_TRANSLATE):
    translate_x = tf.random_uniform([], minval=-1.0, maxval=1.0) * AUGMENT_TRANSLATE_X_PIX
    translate_y = tf.random_uniform([], minval=-1.0, maxval=1.0) * AUGMENT_TRANSLATE_Y_PIX
if (AUGMENT_ROTATE):
    rotate = tf.random_normal([], stddev=AUGMENT_ROTATE_ANGLE_STDEV)
if (AUGMENT_UNIFORM_SCALE):
    uniform_scale = tf.random_uniform([], minval=AUGMENT_MIN_UNIFORM_SCALE, maxval=AUGMENT_MAX_UNIFORM_SCALE)
if (AUGMENT_ASPECT_RATIO):
    x_fac = tf.random_uniform([], minval=AUGMENT_ASPECT_X_MIN, maxval=AUGMENT_ASPECT_X_MAX)
    y_fac = tf.random_uniform([], minval=AUGMENT_ASPECT_Y_MIN, maxval=AUGMENT_ASPECT_Y_MAX)

aspect_transform = augmentation.get_aspect_ratio_transform(x_fac, y_fac, x_center, y_center)
rot_scale_xlate_transform = augmentation.get_affine_transform(1.0, rotate, uniform_scale, translate_x, translate_y, x_cente
total_transform = tf.contrib.image.compose_transforms(aspect_transform, rot_scale_xlate_transform)

#Great, now stack all of the things together, and apply the transform!
all_together = tf.concat([image_mask, template_image], 2)
all_transformed = tf.contrib.image.transform(all_together, total_transform)
```

```
def augment_example(depth_image, template_image, template_mask):

    #Before doing anything else, apply distortions that only happen to the depth image
    #These include speckle noise, point omission noise, and small-angle (approx linear)
    #depth adjustments

    #First, do small-angle adjustments
    if (AUGMENT_SMALL_ANGLE):
        #TODO: Should we use a radially uniform distribution instead?
        x_f = tf.random_uniform([], minval=-1.0, maxval=1.0) * AUGMENT_SMALL_ANGLE_MAX_MM
        y_f = tf.random_uniform([], minval=-1.0, maxval=1.0) * AUGMENT_SMALL_ANGLE_MAX_MM

        depth_image += x_f * x_sweep_array
        depth_image += y_f * y_sweep_array
    if (AUGMENT_SMALL_SCALE):
        delta_z = tf.random_uniform([], minval=-1.0, maxval=1.0) * AUGMENT_SMALL_SCALE_MAX_DELTA_Z
        depth_image += delta_z
        depth_image = tf.maximum(0.0, depth_image)
    if (AUGMENT_GAUSS_NOISE):
        noise = tf.random_normal([424, 512, 1], stddev=AUGMENT_GAUSS_NOISE_STDEV)
        depth_image += noise
    if (AUGMENT_OMIT_NOISE):
        roll = tf.random_uniform([424, 512, 1], minval=0.0, maxval=1.0) < AUGMENT_OMIT_PROB
        mask = 1.0 - tf.cast(roll, tf.float32)
        depth_image = depth_image * mask

    #This will make things a bit easier, because when we manipulate the
    #depth image, we'll usually wind up modifying the mask, too
    template_mask = tf.cast(template_mask, tf.float32)
    image_mask = tf.concat([depth_image, template_mask], 2)

    #First thing's first -- determine whether or not to mirror the image
    #and adjust template positions accordingly
    if (AUGMENT_FLIP):
```

The Depth-Frame Pose Detector: Data Pipeline

Tensorflow has a really nice Dataset class -> Abstracts away lots of complexity

```
def build_dataset_from_dir(tfrecordRoot, batch_size):
    tfrecordFiles = [y for x in os.walk(tfrecordRoot) for y in glob(os.path.join(x[0], '*.tfrecord'))]
    raw_dataset = tf.data.TFRecordDataset(tfrecordFiles)

    raw_dataset.shuffle(buffer_size=shuffle_buffer_size, reshuffle_each_iteration=True)

    #Parse from the raw dataset into (depth image, template positions, template mask) format
    dataset_formatted = raw_dataset.map(parse_example_to_sample_label, num_parallel_calls=CPU_PARALLEL_THREADS)

    dataset_augmented = dataset_formatted.map(augment_example, num_parallel_calls=CPU_PARALLEL_THREADS)

    #Okay, great! Now it's batching time!
    result_set = dataset_augmented.batch(batch_size, drop_remainder=True)
    result_set = result_set.repeat()

    result_set = result_set.prefetch(2)

    return result_set
```

Annotating RGB Pose Data

Takes the trained depth image pose detector -> Uses it to annotate the dataset.

Bit of post-processing -- Output from network is arbitrary position in template-space

Snap those output points to actual points on the templates

Encode it in more-or-less the same .tfrecord format as the annotated depth images

Training The RGB-only Pose Detector

Literally the same as the Depth pose detector, just with more input channels

… and a smaller, more compact neural network.

We need to make it run on a mobile device, after all!

The network is *quantized** for extra performance

Exported to *tflite* model format

*Explaining this in detail could also take a while.

Building an Android App

Used

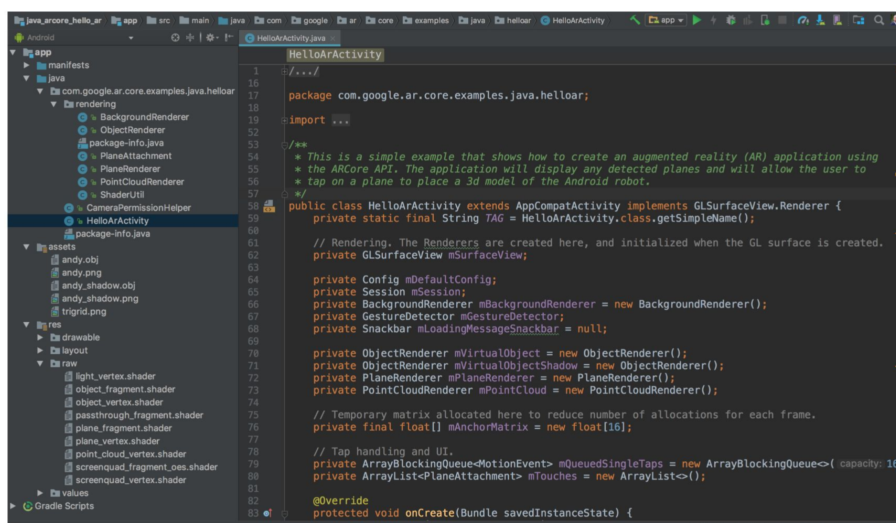
by

Would help with absolute position tracking

Convenient SceneForm rendering

Useful for rendering animated skeletons

Quickstart for Android



Provides some basic 3D math classes: Quaternions, Vectors, Matrices…

I wound up ripping stuff out of a Tensorflow Lite Demo, and out of an ARCore Demo, and duct-taping them together somewhat hastily.

Raycasting And Dolls

How to determine where to render the figure, and in what pose?

Use Gradient Descent with automatic differentiation

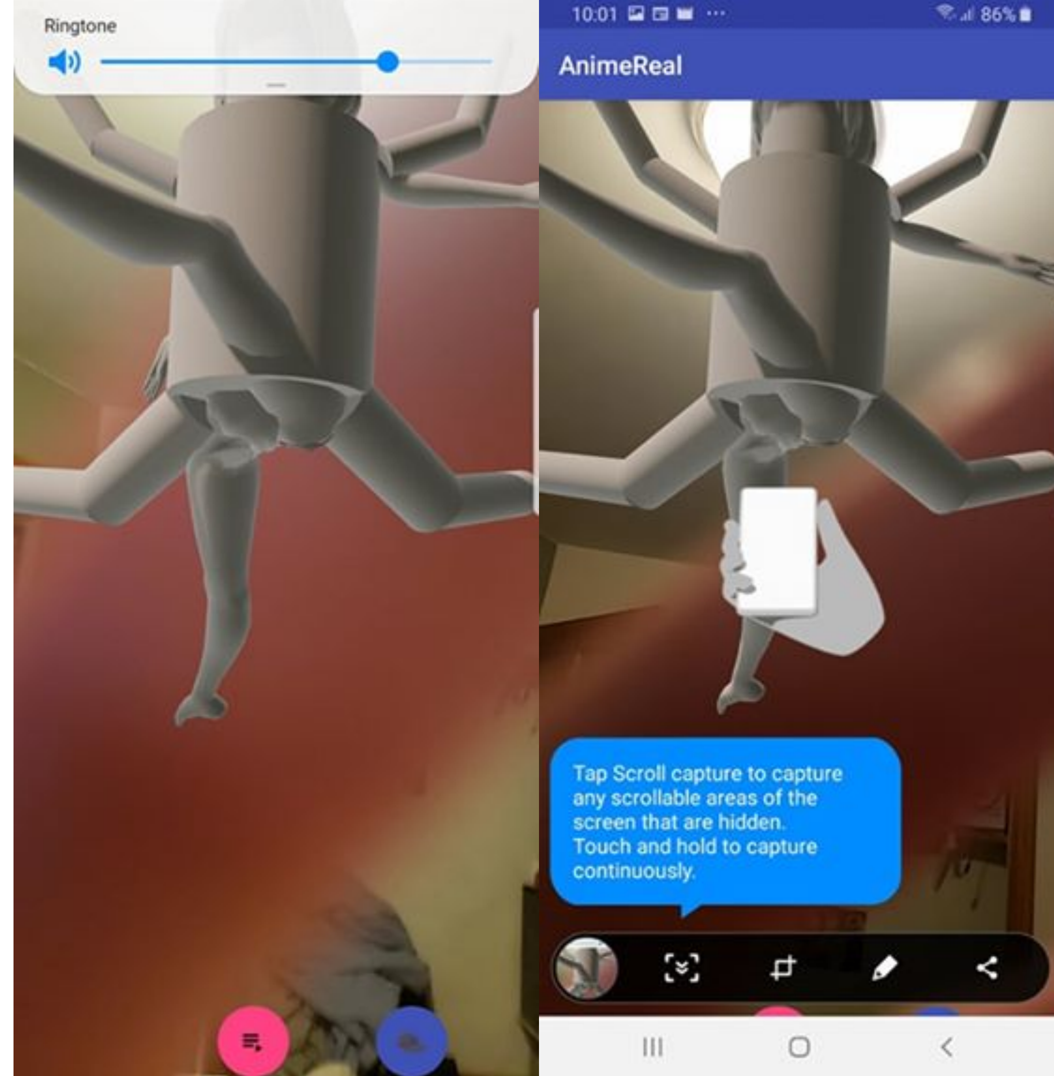
Loss function: Draw rays through each pixel in the current frame’s neural net output. Wherever the ray hits the schematic “doll”, determine what body part it hit, and where that point is on the template. Compare the distance between those two.

Model joint rotation -> represented with unit Quaternions for numerical stability.

Each frame runs a fixed number of gradient descent steps to cap the runtime.

Behold

It's **garbage**



A Retrospective

I managed to build the full pipeline I dreamed of, which is p neat.

However, the individual components could’ve used some improvements.

In particular: A human-labeled set of ground-truth annotations >

Any sort of automated or semi-automated process

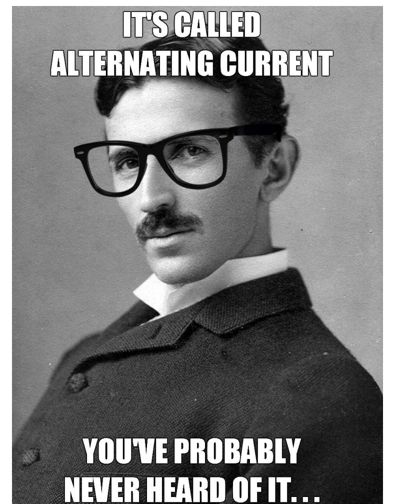
Still was lots of fun, though.

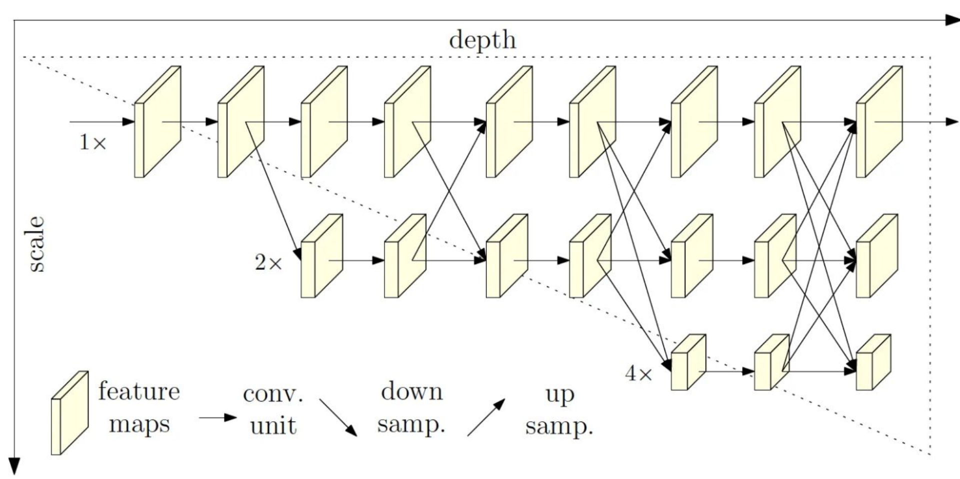
But Wait… Why'd You Stop?

Published On Feb. 25th of 2019: HRNet (Microsoft Research) Paper

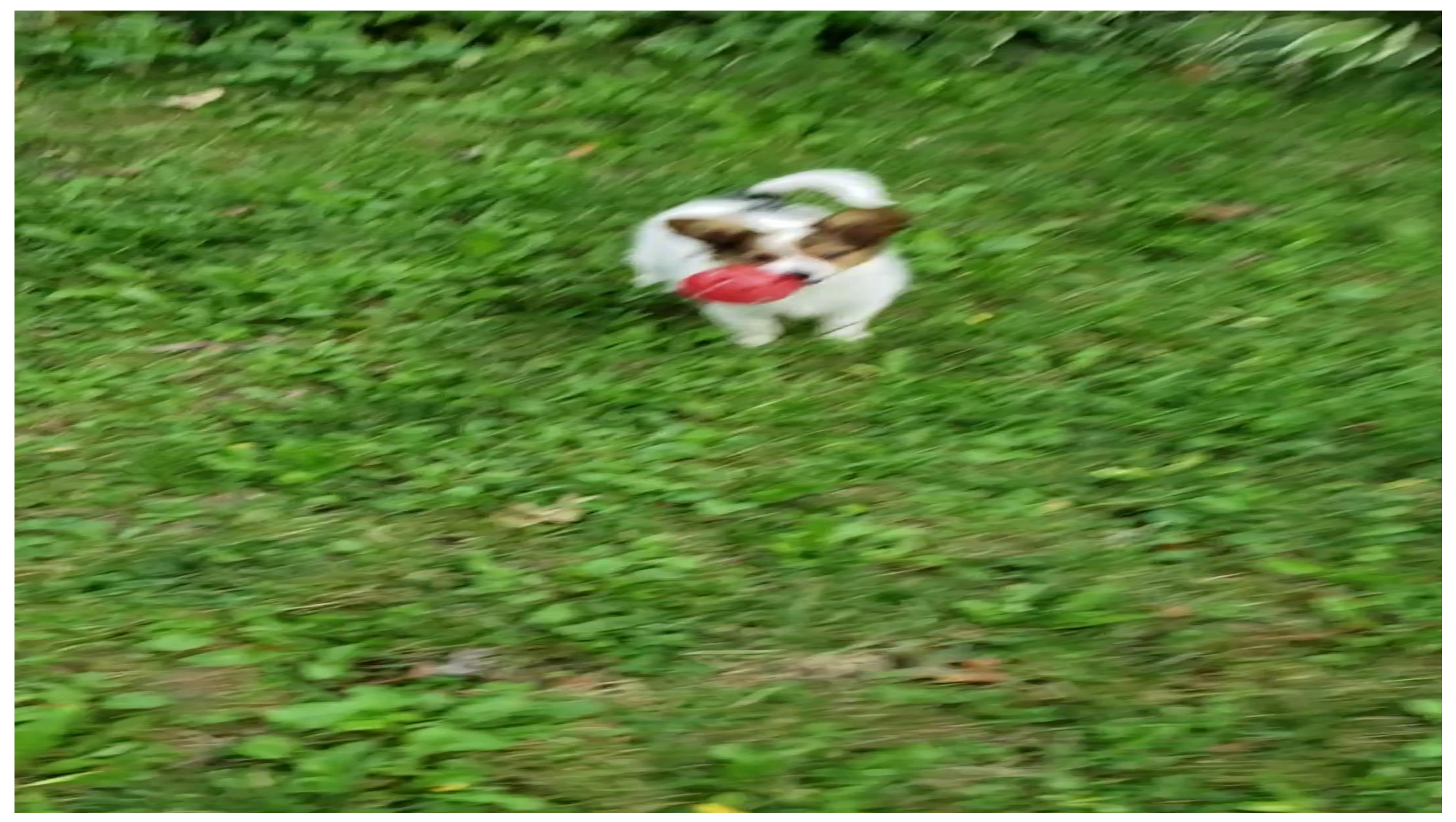
Pretty much the same idea

I was doing it before it was cool!





What I'm Doing Nowadays



All Code Now Open-Source!

Repo: https://github.com/bubble-07/AnimeReal/

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

Alexander-Grabanski Merge branch 'master' of https://github.com/bubble-07/AnimeReal — Latest commit b12193f 3 hours ago

Name	Last commit	Time
AnimeRealDataCollection	Initial adding of source files	3 hours ago
JavaPoseAnnotator	Initial adding of source files	3 hours ago
LocalKinectSaver	Initial adding of source files	3 hours ago
OldAnimeReal	Initial adding of source files	3 hours ago
PythonCalibrator	Initial adding of source files	3 hours ago
animereal-app-src/src/main	Initial adding of source files	3 hours ago
python-annotator	Initial adding of source files	3 hours ago
.gitignore	Update gitignore	3 hours ago
LICENSE	Initial commit	3 hours ago

What About The Datasets?

They’re fairly large (~4TB uncompressed)

If you want them, *please* ask me, and I’ll try to work something out.

Questions?