ML Chessboard Recognition

Stephen Huan

Overview

Computer vision component:

- Generate chessboard corner grid (9x9 evenly spaced)
- Rotate, translate, etc. to get “realistic” viewpoints

Self-supervised learning component:

- Remove certain % of grid points and add certain # of superfluous points
- Train NN to filter out extraneous points and fill in missing grid points

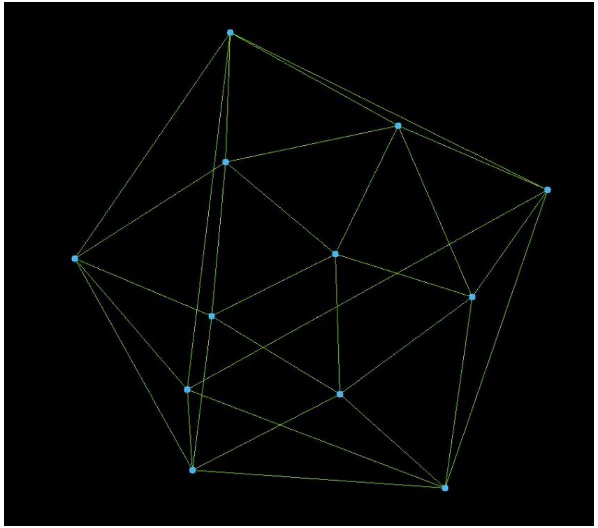
Perspective Projection

Objective

- Goal: generate realistic point grids
  - “Realistic” as in matching the empirical distribution of chessboards
  - Not looking straight down, has perspective effects, etc.
- Mathematical details are messy, therefore hard to generate
  - Also hard to have high-level control of generation
- Why not directly simulate?

Perspective Projection

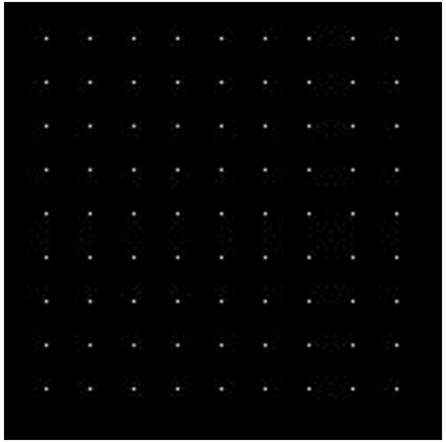
- Render 3D points onto a 2D screen
- Mathematical details are unnecessary
  - Just needs to work
- [Helpful Wikipedia link]



Summary:

- Parameterize position of camera and screen
  - Need camera to be far enough away
  - Otherwise, points might go behind the camera…
  - Distance of screen determines spacing of grid
- Rotate 3D “object” points to rotate chessboard

Details, code



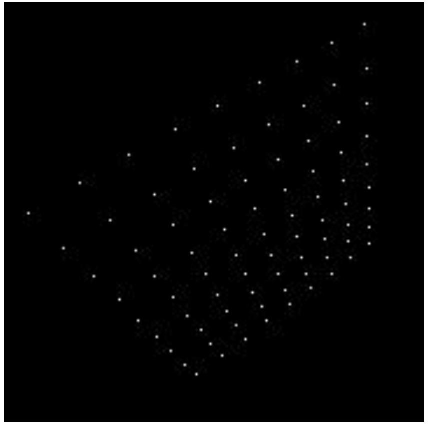
- Start with ideal grid centered at (0, 0), on the plane *z* = 0
  - [(-4, -4, 0), (-4, -3, 0), … (0, 0), … (4, 3, 0), (4, 4, 0)]
- Apply rotation matrix to grid

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & \sin(\theta_x) \\ 0 & -\sin(\theta_x) & \cos(\theta_x) \end{bmatrix} \begin{bmatrix} \cos(\theta_y) & 0 & -\sin(\theta_y) \\ 0 & 1 & 0 \\ \sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix} \begin{bmatrix} \cos(\theta_z) & \sin(\theta_z) & 0 \\ -\sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- This may make *z* != 0 if θ_x or θ_y != 0, hence truly 3D points
  - Need camera far enough from points to avoid points rotating behind camera
  - *z* > 5 is sufficient, for simplicity let *z* = 10 so camera is placed at [0, 0, 10]
- Apply perspective projection transformation matrix

$$\begin{bmatrix} \mathbf{f}_x \\ \mathbf{f}_y \\ \mathbf{f}_w \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{\mathbf{e}_x}{\mathbf{e}_z} \\ 0 & 1 & \frac{\mathbf{e}_y}{\mathbf{e}_z} \\ 0 & 0 & \frac{1}{\mathbf{e}_z} \end{bmatrix} \begin{bmatrix} \mathbf{d}_x \\ \mathbf{d}_y \\ \mathbf{d}_z \end{bmatrix}$$

Generating a Random Grid, code

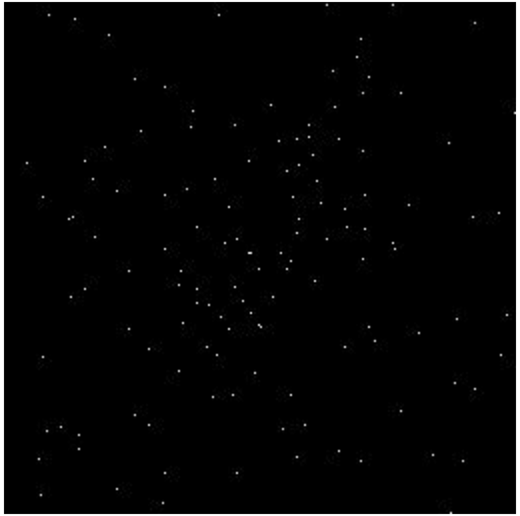


- Sample angle $[\theta_x, \theta_y, \theta_z]$ from $[-\pi/4, \pi/4)$ uniformly
- Recall camera fixed at [0, 0, 10]
- Place screen at [0, 0, *d*], *d* controls the spacing of the grid
  - e.g. if looking directly down d = 50 implies points $d/10$ = 5 pixels apart
- However, rotation will make points closer/farther apart
- Sample *d* from [*H**25/64, *H**25/32] is a good heuristic
  - Assuming height *H* is less than or equal to the width of the image, *W*
- Finally, pick translation such that points are contained within image
- Let x_0, x_1, y_0, y_1 be the bounding box of the points
- Sample translation t_x from $[-x_0, H - x_1]$ and t_y from $[-y_0, W - y_1]$
- Guarantees final points within rectangle (0, 0) to (*H*, *W*)

Self-supervised Learning

Adding Noise, [code]

- Want neural network to identify ground truth grid
- Generate pairs of (noisy grid, ground truth grid)
- (*X*, *y*) training pairs, *y* generated by previous slides
- Start with point list *y*
- Remove random percentage of grid points
  - Between 0 to 0.5 of grid points (arbitrary choice)
- Add random number of random points
  - Between 0 to 100 random points (also mostly arbitrary)



Architecture

- Representation?
  - List of points vs. binary image
- If list of points: use fully connected NN
- Problem: need set point order, not invariant to permutation
- Opinion: Binary image is a nicer representation
  - Also allows for convolution neural network (CNN)
- Image to image prediction
- Sample down with pooling, then upsample with transposed layers
  - See Chapter 14 “Deep Computer Vision Using Convolutional Neural Networks”
  - *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*

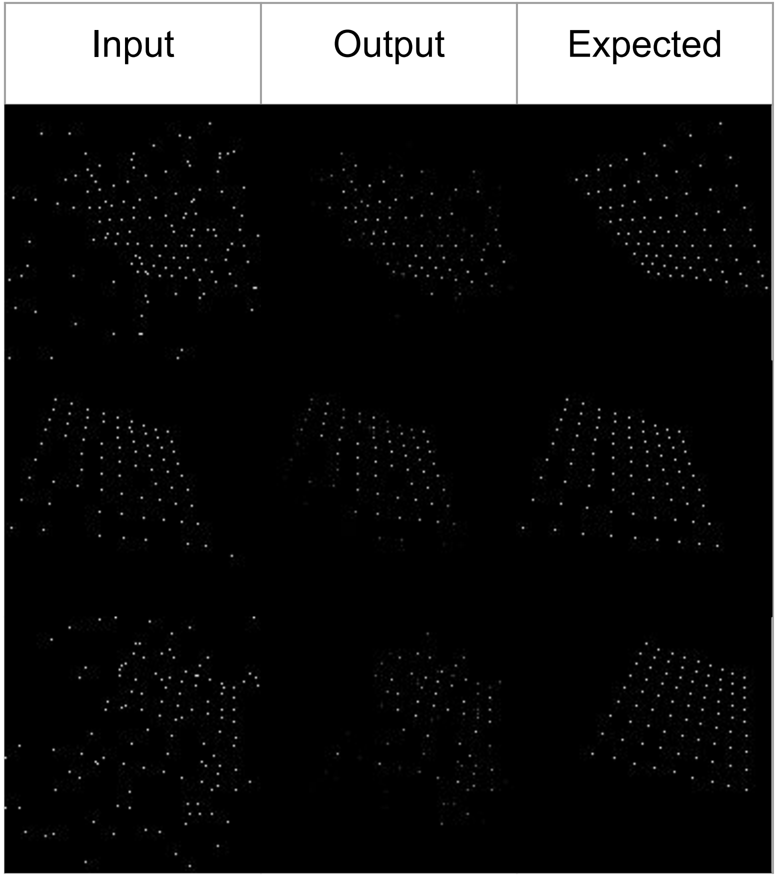
Summary, code

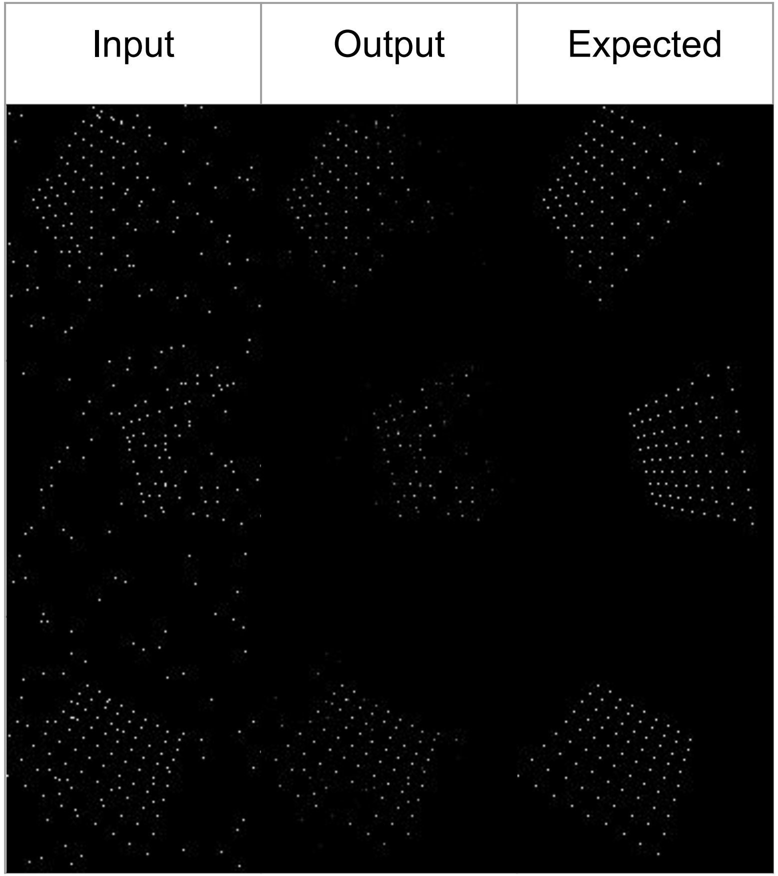
```python
kwargs = {"padding": "same", "activation": "relu"}
model = keras.models.Sequential([
    # downscale
    layers.Conv2D(16, (3, 3), input_shape=X_train[0].shape, **kwargs),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(32, (5, 5), **kwargs),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(32, (7, 7), **kwargs),
    # upscale
    layers.Conv2DTranspose(16, (7, 7), strides=2, **kwargs),
    layers.Conv2D(8, (3, 3), **kwargs),
    layers.Conv2DTranspose(4,  (5, 5), strides=2, **kwargs),
    # flatten to output with 1D convolution, make sure between 0 and 1
    # sigmoid doesn't work too well, use tanh(relu(x))
    layers.Conv2D(1, (1, 1), activation="tanh"),
    layers.ReLU(),
])
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 128, 128, 16)      160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 64, 64, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 64, 32)        12832
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 32)        50208
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 64, 64, 16)        25104
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 8)         1160
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 128, 128, 4)       804
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 128, 128, 1)       5
_________________________________________________________________
re_lu (ReLU)                 (None, 128, 128, 1)       0
=================================================================
Total params: 90,273
Trainable params: 90,273
Non-trainable params: 0
_________________________________________________________________
```

Technical Details

- Need output to be between 0 and 1 to be a valid probability
  - Classic choice would be sigmoid
  - Sigmoid doesn’t work that well, tanh(reLU(*x*)) works better for some reason
- binary_crossentropy would be the standard loss for binary classification
- mean_squared_error works better
- These losses are relatively uninformative, > 99% of the image is black
- Also keep track of:
  - precision: % of predicted grid points that are actually part of the grid
  - recall: % of grid points that were predicted to be part of the grid
- Model usually has low recall (~60%, unable to fill in missing points)
- Decent precision (~80%)


Input
Output
Expected


Input
Output
Expected

Analysis

- Acts more like a “filter” than a generator
    - Able to remove extraneous points but not able to fill in missing points
- Filtering ability is better with more original grid points
    - If given grid with many holes, starts to filter out grid points
- Architectural improvements?
    - Experiment with filter size, pooling, etc.

Code

[Implementation]