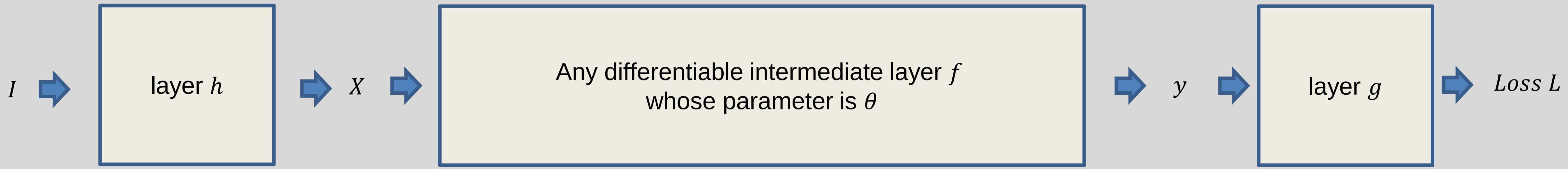
Computer Vision

Lecture 07: Training CNNs with Large Data

Differentiable layers



We need to implement three things for an intermediate layer f :

forward rule:	$y = f(X;\theta)$	for	$g\big(f(X;\theta)\big) = \mathrm{L}$
backward rule:	$\frac{dy}{dX}$	for	$\frac{dL}{d\mathrm{X}} = \frac{dy}{d\mathrm{X}} \times \frac{dL}{dy}$
parameter update rule:	$\frac{dy}{d\theta}$	for	$\theta^{new} = \theta - \varepsilon \frac{dy}{d\theta} \times \frac{dL}{dy}$

Differentiable layers

I ⇨ layer h ⇨ X ⇨ Any differentiable intermediate layer f whose parameter is θ ⇨ y ⇨ layer g ⇨ $Loss\ \mathrm{L}$

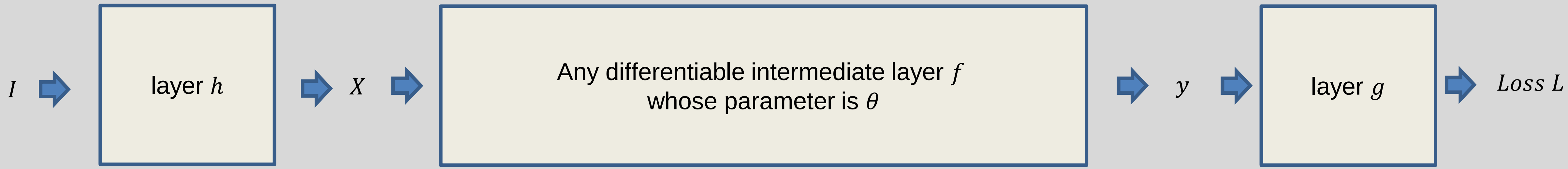
We need to implement three things for an intermediate layer f :

forward rule:	$y = f(X;\theta)$	for	$g\big(f(X;\theta)\big) = \mathrm{L}$
backward rule:	$\frac{dy}{dX}$	for	$\frac{dL}{d\mathrm{X}} = \frac{dy}{d\mathrm{X}} \times \frac{dL}{dy}$
parameter update rule:	$\frac{dy}{d\theta}$	for	$\theta^{new} = \theta - \varepsilon \frac{dy}{d\theta} \times \frac{dL}{dy}$

⇨ PyTorch can do these automatically.

loss.backward()
optimizer.step()

Differentiable layers



Some layers (e.g. pooling, activation) do not have parameters θ. It requires only two:

forward rule: $y = f(X;\theta)$ for $g\big(f(X;\theta)\big) = \mathrm{L}$

backward rule: $\frac{dy}{dX}$ for $\frac{dL}{d\mathrm{X}} = \frac{dy}{d\mathrm{X}} \times \frac{dL}{dy}$

In convolutional layers, parameters are convolutional filter kernel’s weights.

CNNs

RGB image

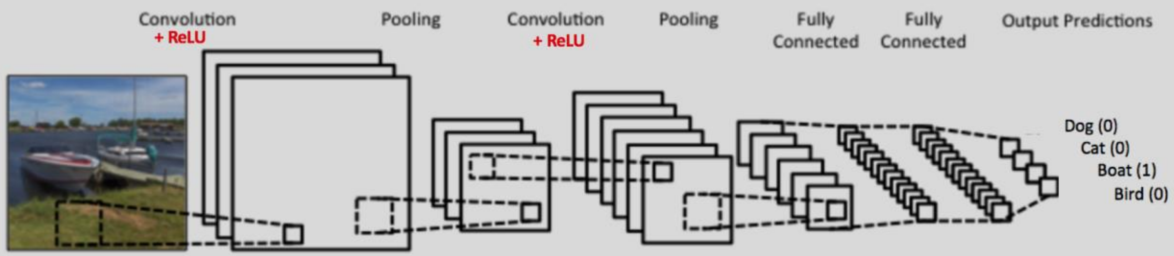
Any Differentiable Layers

cat

Semantic labels

CNNs

RGB image



cat

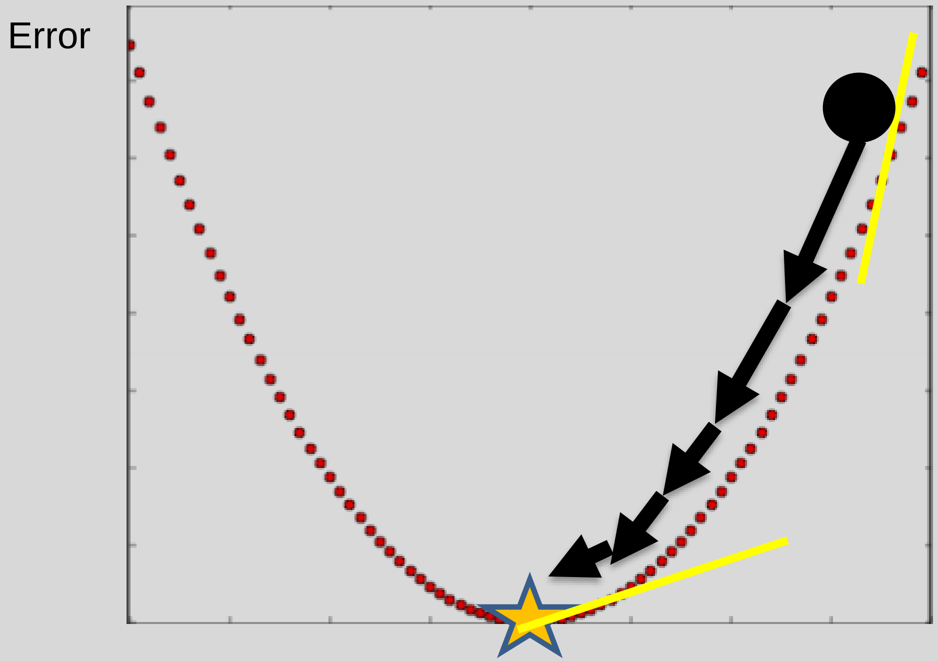
Semantic labels

CNN implementation in PyTorch

```
import torch
import torch.nn

class MyCNN(nn.Module):
  def __init__(self):
    super().__init__()
    self.layer = nn.Sequential(
        nn.Conv2d(1, 16, 5),
        nn.ReLU(),
        nn.Conv2d(16, 32, 5),
        nn.MaxPool2d(2, 2)
        nn.Conv2d(32, 64, 5)
        nn.ReLU()
        nn.MaxPool2d(2,2)
    )
    self.fc_layer = nn.Sequential(
        nn.Linear(64*3*3, 10)
        nn.ReLu()
        nn.Linear(100, 10)
    )

  def forward(self, x):
    out = self.layer(x)
    out = out.view*(batch_size, -1)
    out = self.fc_layer(out)

    return out
```

```
import torch

net = MyCNN()

loss_func = torch.nn.MSELoss()

optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

losses = []

for i in range(num_epoch):

  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)

  loss.backward()

  optimizer.step()

  losses.append(loss.item())
```

Gradient Descent



$$\theta^{(t)} = \{W^{(t)}, b^{(t)}\}$$

```
import torch

loss_func = torch.nn.MSELoss()

optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

for i in range(num_epoch):

  optimizer.zero_grad()

  output = net(x)

  loss = loss_func(output, y)

  loss.backward()

  optimizer.step()
```

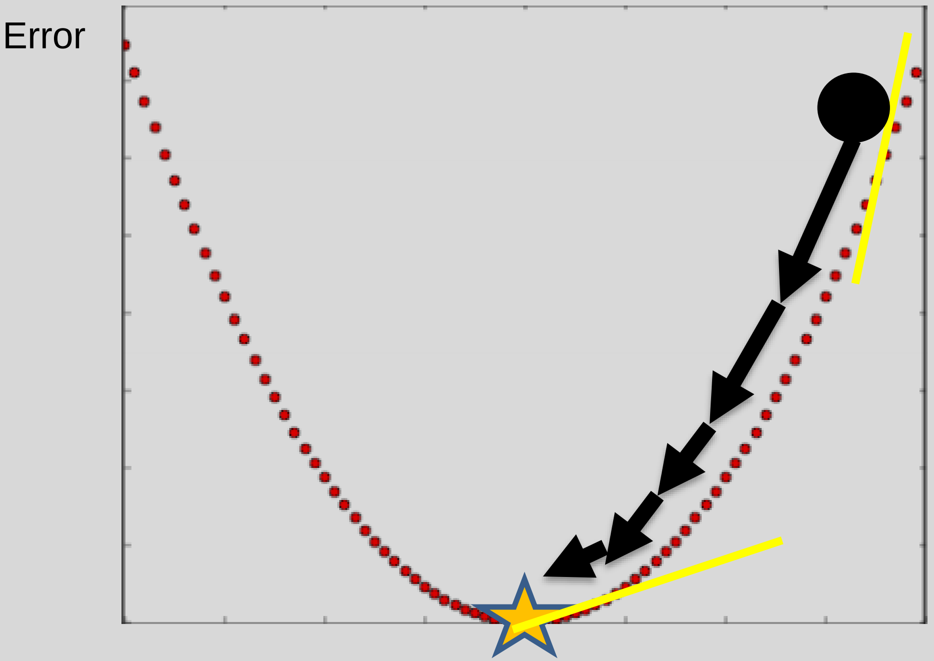
$\text{Error}(W, b) = \frac{1}{3}\sum_{i=1}^{3}(f(x_i; W, b) - y_i)^2,$

$$W^{(t+1)} = W^{(t)} - \epsilon \frac{\partial}{\partial W^{(t)}}\text{Error}(W^{(t)}, b^{(t)})$$

$$b^{(t+1)} = b^{(t)} - \epsilon \frac{\partial}{\partial b^{(t)}}\text{Error}(W^{(t)}, b^{(t)})$$

ϵ : Learning rate (small value e.g. 0.01)

Stochastic Gradient Descent




```
import torch

loss_func = torch.nn.MSELoss()

optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

for i in range(num_epoch):

  for x_batch, y_batch in train_loader:

     optimizer.zero_grad()

     output = net(x_batch)

     loss = loss_func(output, y_batch)

     loss.backward()

     optimizer.step()
```

$$\theta^{(t)} = \{W^{(t)}, b^{(t)}\}$$

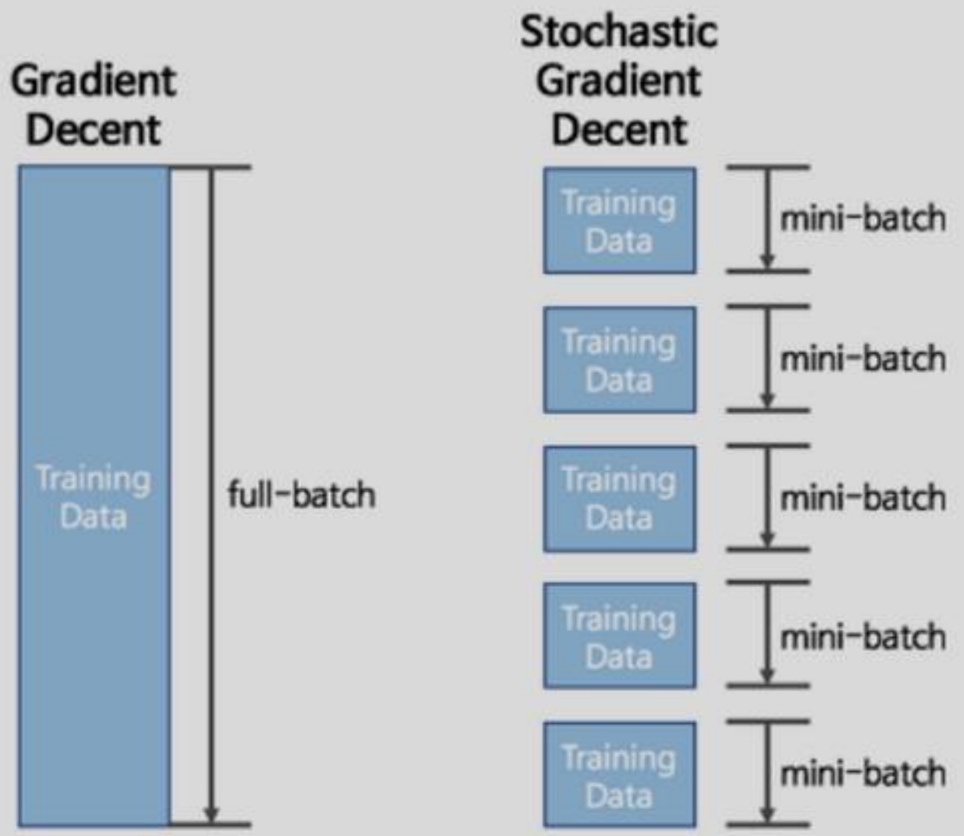
$$\text{Error}(W, b) = \frac{1}{3}\sum_{i=1}^{3}(f(x_i; W, b) - y_i)^2,$$

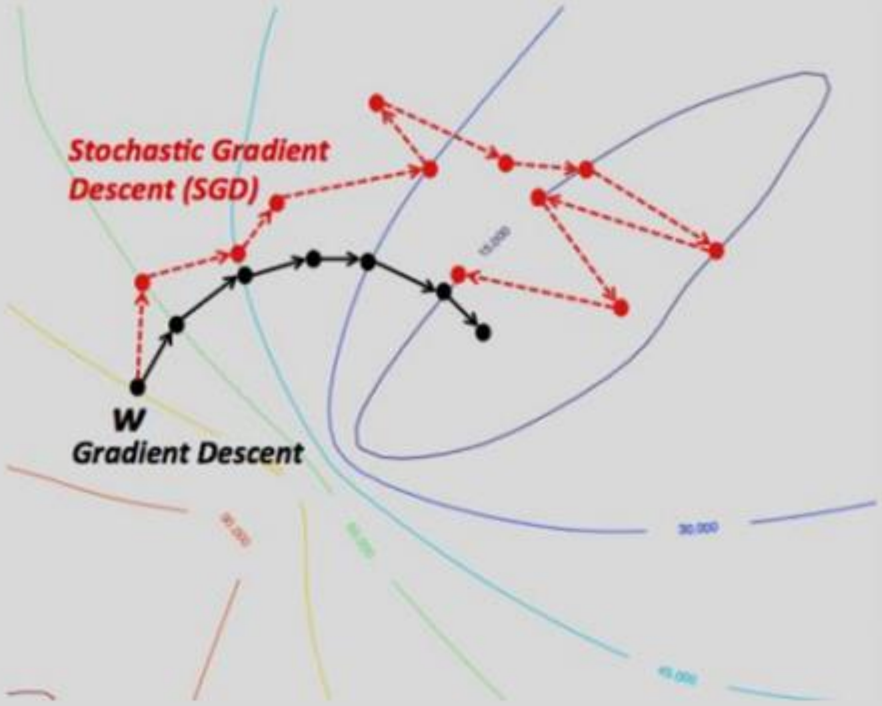
$$W^{(t+1)} = W^{(t)} - \epsilon \frac{\partial}{\partial W^{(t)}} \text{Error}(W^{(t)}, b^{(t)})$$

$$b^{(t+1)} = b^{(t)} - \epsilon \frac{\partial}{\partial b^{(t)}} \text{Error}(W^{(t)}, b^{(t)})$$

ϵ : Learning rate (small value e.g. 0.01)

Stochastic Gradient Descent





Stochastic Gradient Descent

- Gradient descent
  Calculate for all data (takes large amount of time).
  Go 1 optimal step.

- Stochastic gradient descent
  Calculate gradients for partial data (takes small amount of time).
  Go many non-globally-optimal steps, but converges.

Momentum

For the real-world loss function, it is not so clean.

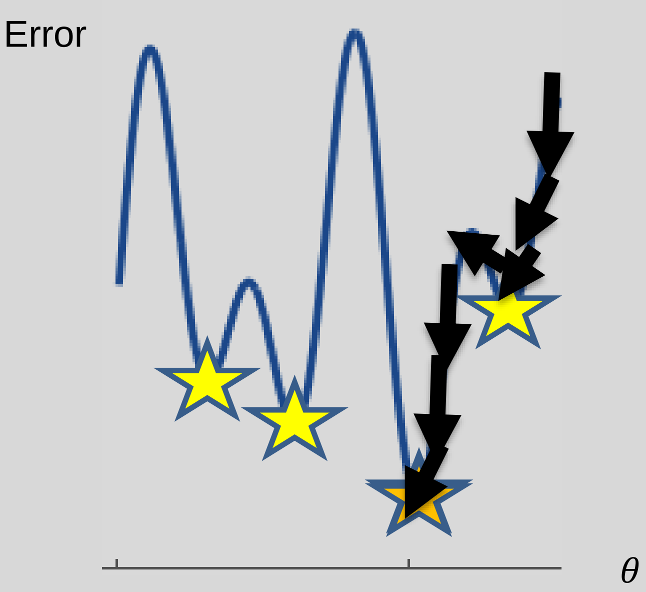
→ There are many local minima.

Functions with many local minima

Momentum

For the real-world loss function, it is not so clean.

→ There are many local minima. → Use momentum to climb up small hills.



Functions with many local minima

$$\mathrm{V}^{(t+1)} = \mathrm{V}^{(t)} * \mathrm{m} - \epsilon \frac{\partial}{\partial W^{(t)}} \mathrm{Error}(W^{(t)}, b^{(t)})$$

$$\mathrm{U}^{(t+1)} = \mathrm{U}^{(t)} * \mathrm{m} - \epsilon \frac{\partial}{\partial b^{(t)}} \mathrm{Error}(W^{(t)}, b^{(t)})$$

$$W^{(t+1)} = W^{(t)} + \mathrm{V}^{(t+1)}$$

$$b^{(t+1)} = b^{(t)} + \mathrm{U}^{(t+1)}$$

Nesterov Gradient Descent

$$\mathrm{V}^{(t+1)} = \mathrm{V}^{(t)} * \mathrm{m} - \epsilon \frac{\partial}{\partial W^{(t)} + \mu \mathrm{V}^{(t)}} \mathrm{Error}(W^{(t)} + \mu \mathrm{V}^{(t)}, b^{(t)} + \mu \mathrm{U}^{(t)})$$

$$\mathrm{U}^{(t+1)} = \mathrm{U}^{(t)} * \mathrm{m} - \epsilon \frac{\partial}{\partial b^{(t)} + \mu \mathrm{U}^{(t)}} \mathrm{Error}(W^{(t)} + \mu \mathrm{V}^{(t)}, b^{(t)} + \mu \mathrm{U}^{(t)})$$

$$W^{(t+1)} = W^{(t)} + \mathrm{V}^{(t+1)}$$

$$b^{(t+1)} = b^{(t)} + \mathrm{U}^{(t+1)}$$

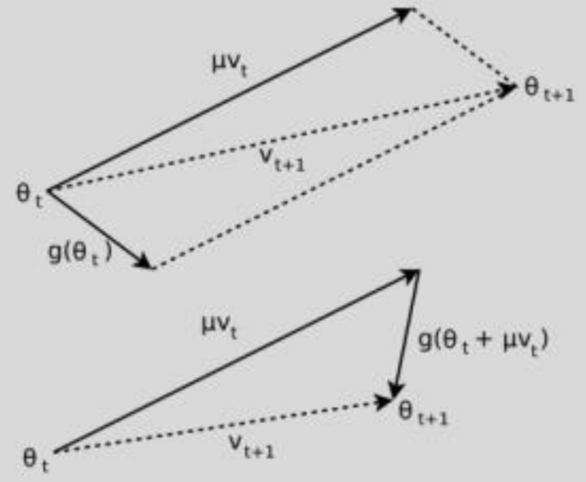
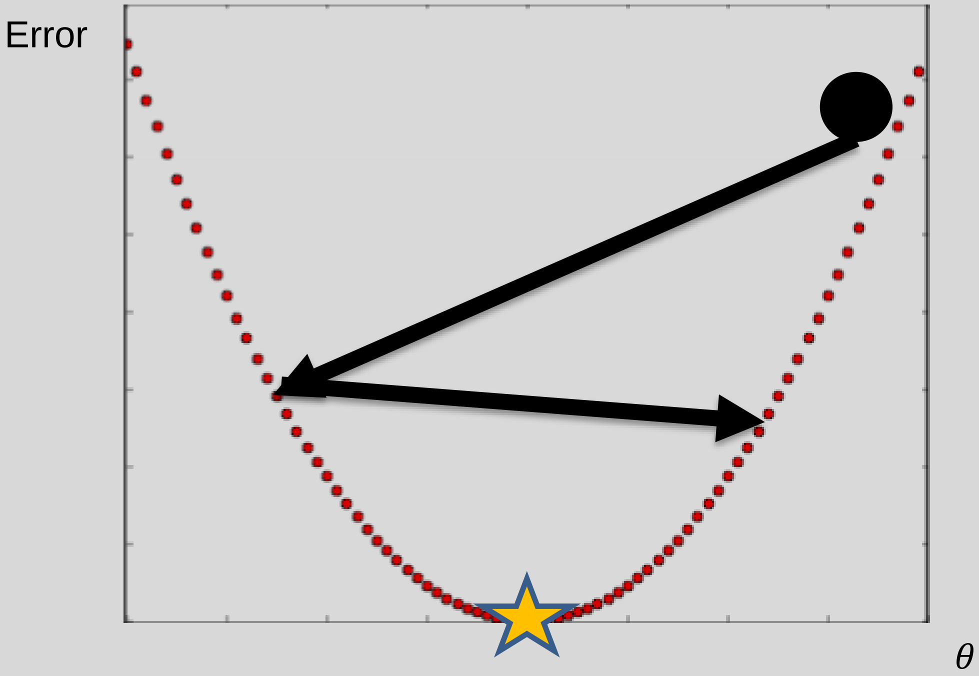


Figure 1. **(Top)** Classical Momentum **(Bottom)** Nesterov Accelerated Gradient

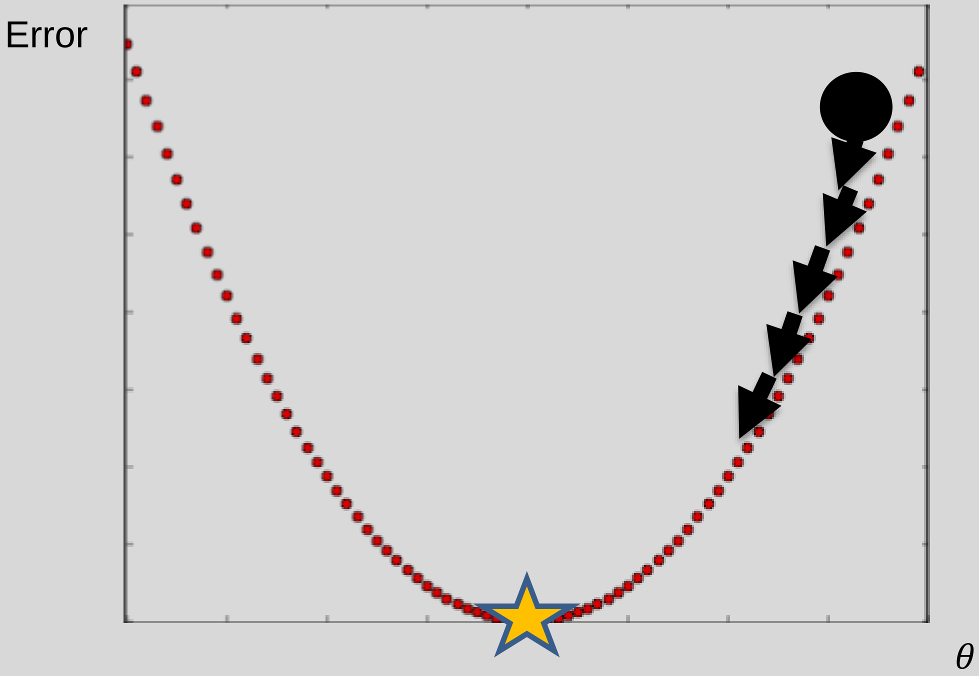
$$\theta_t = \{W^{(t)}, b^{(t)}\}$$

$$v_t = \{V^{(t)}, U^{(t)}\}$$

Learning rate



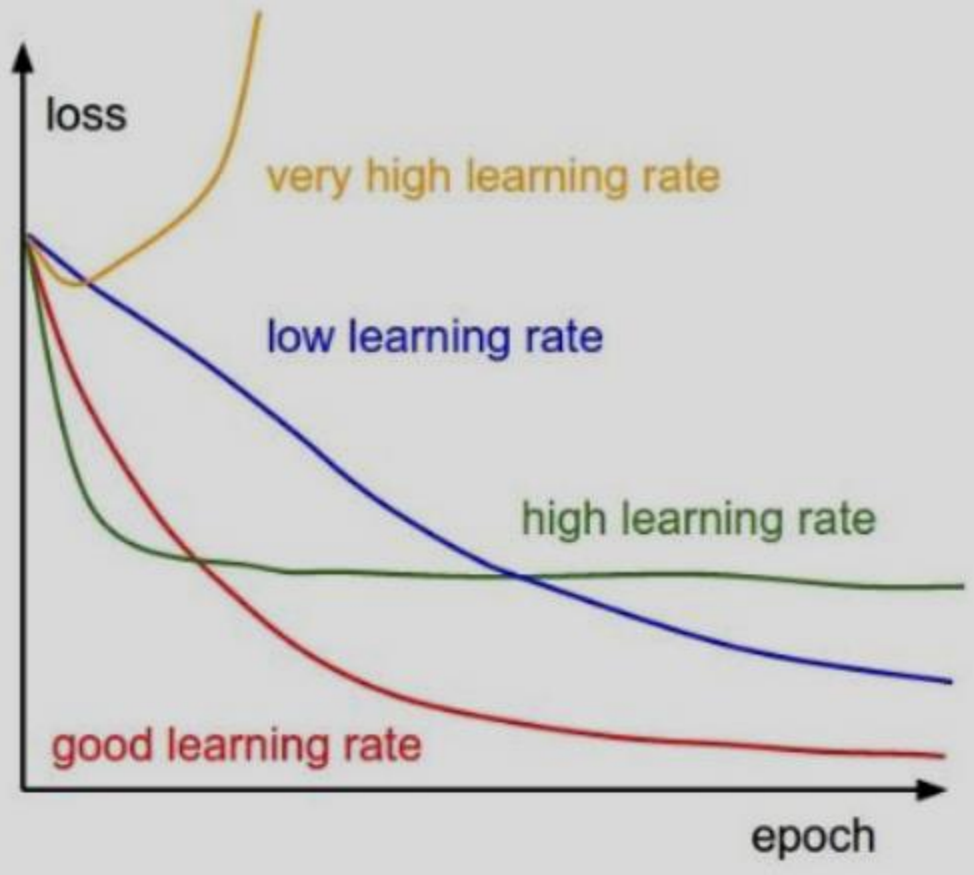
If learning rate is high, it does not converge.



If learning rate is low, too slow to converge.

It is important to decide the proper learning rate.

Learning rate



It is important to decide the proper learning rate.

Learning rate

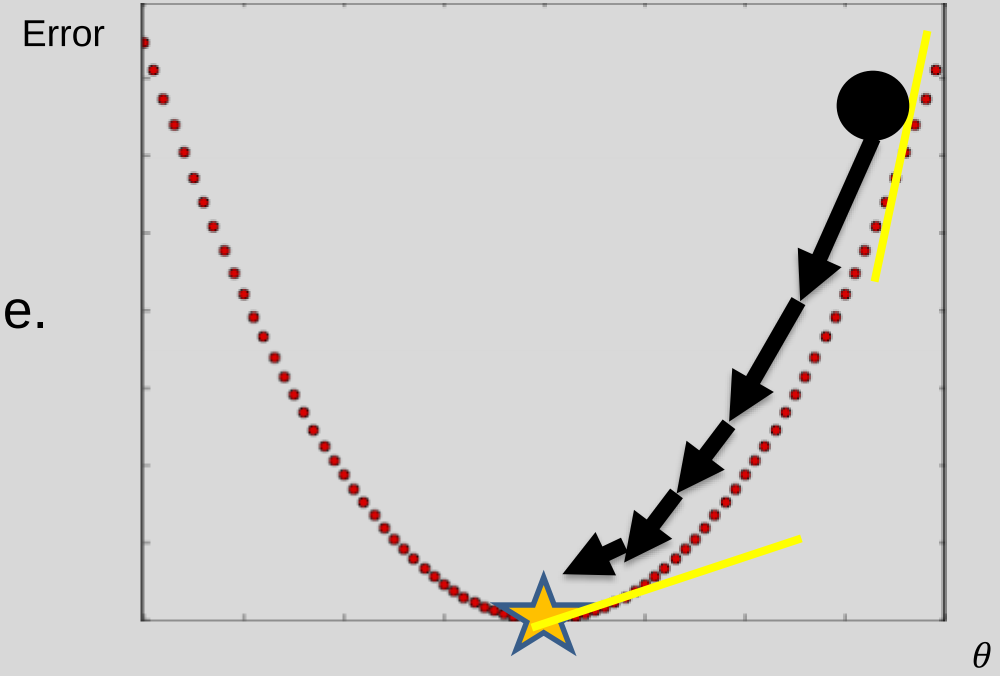
Simple learning rate scheduler defined in the PyTorch:

torch.optim.lr_scheduler.StepLR

→ Multiply gamma(<1) to the learning rate, for every step_size.

torch.optim.lr_scheduler.ExponentialLR

→ Multiply gamma(<1) to the learning rate, for every epoch.



Learning rate

```
import torch
import torch.nn as nn
import torch.optim.lr_scheduler as lr_scheduler

net = nn.Sequential(
    nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 5, stride = 1, padding = 2)
)

optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
scheduler = lr_scheduler.ExponentialLR(optimizer, gamma=0.99)

for i in range(10):
  scheduler.step()
  print(i, scheduler.get_lr())
```

```
0 [0.009801]
1 [0.00970299]
2 [0.0096059601]
3 [0.009509900499]
4 [0.00941480149401]
5 [0.0093206534790699]
6 [0.00922744694427920]
7 [0.00913517247483640]
8 [0.00904382075008804]
9 [0.00895338254258716]
```

Learning rate

```python
import torch
import torch.nn as nn
import torch.optim.lr_scheduler as lr_scheduler

net = nn.Sequential(
    nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 5, stride = 1, padding = 2)
)

optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
scheduler = lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.99)

for i in range(10):
  scheduler.step()
  print(i, scheduler.get_lr())
```

```
0 [0.009801]
1 [0.00970299]
2 [0.0096059601]
3 [0.009509900499]
4 [0.00941480149401]
5 [0.0093206534790699]
6 [0.0092274469442792]
7 [0.009135172474836408]
8 [0.009043820750088045]
9 [0.008953382542587164]
```

Learning rate

```
import torch
import torch.nn as nn
import torch.optim.lr_scheduler as lr_scheduler

net = nn.Sequential(
    nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 5, stride = 1, padding = 2)
)

optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
scheduler = lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.99)

for i in range(10):
  scheduler.step()
  print(i, scheduler.get_lr())
```

```
0 [0.01]
1 [0.01]
2 [0.009801]
3 [0.0099]
4 [0.0099]
5 [0.00970299]
6 [0.009801]
7 [0.009801]
8 [0.0096059601]
9 [0.00970299]
```

AdaGrad

Motivation:

Adaptively update gradients for individual parameters.

Update parameters with a large step, whose parameters haven't been changed a lot.

Update parameters with a small step, whose parameters have been changed a lot.

AdaGrad

SGD: $$\theta^{(t)} = \theta^{(t-1)} - \epsilon \cdot g$$

AdaGrad:

$$\theta^{(t)} = \theta^{(t-1)} - \frac{\epsilon}{\delta + \sqrt{r}} \cdot g$$

$$r = r + g^2$$

RMS Prop

AdaGrad:

$$\theta^{(t)} = \theta^{(t-1)} - \frac{\epsilon}{\delta + \sqrt{r}} \cdot g$$

$$r = r + g^2$$

RMS Prop:

$$\theta^{(t)} = \theta^{(t-1)} - \frac{\epsilon}{\delta + \sqrt{r}} \cdot g$$

$$r = \rho \cdot r + (1 - \rho) \cdot g^2$$

Adam

Motivation:

Jointly tackle `learning rate‘ and `direction’ issues.

Combine RMSProp and Momentum methods.

Adam

$$s^{(t)} = \beta_1 \cdot s^{(t-1)} + (1 - \beta_1) \cdot g$$

$$r^{(t)} = \beta_2 \cdot r^{(t-1)} + (1 - \beta_2) \cdot (g \cdot g)$$

$$\hat{s} = \frac{s}{1 - \beta_1^t}$$

$$\hat{r} = \frac{r}{1 - \beta_2^t}$$

$$\Rightarrow \quad \theta^{(t)} = \theta^{(t-1)} - \epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$$

Adam

PyTorch already offers diverse optimization methods:

```
torch.optim.SGD
torch.optim.RMSProp
torch.optim.AdaGrad
torch.optim.Adam

optimizer = torch.optim.SGD(net.parameters(), lr=0.001)
optimizer = torch.optim.Adam(net.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-08)
```

Remarks

There have been several attempts to optimize CNNs well with varied gradients (direction, learning rate).

Adam is generally the best solution; but it depends on applications.

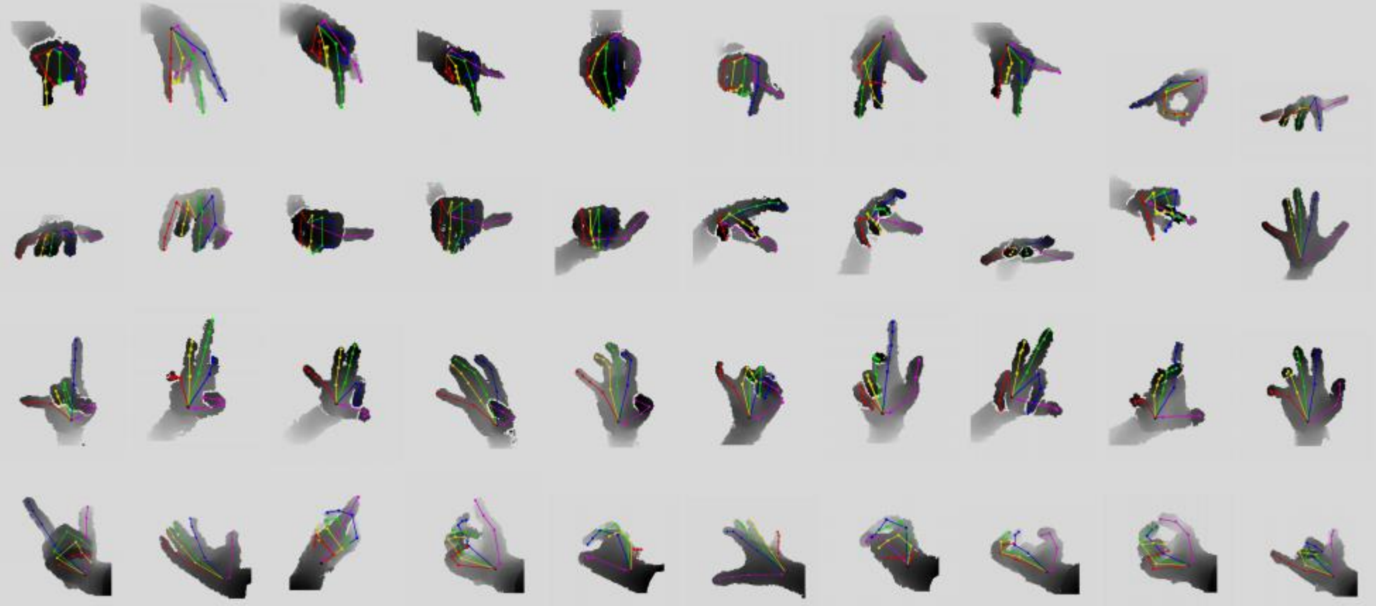
Importance of Data

As the supervised learning method, Deep learning requires many (x, y) pairs.

x: Data,
y: Annotation.

Imagine that making your network overfit to large-scale dataset you can see on Earth.

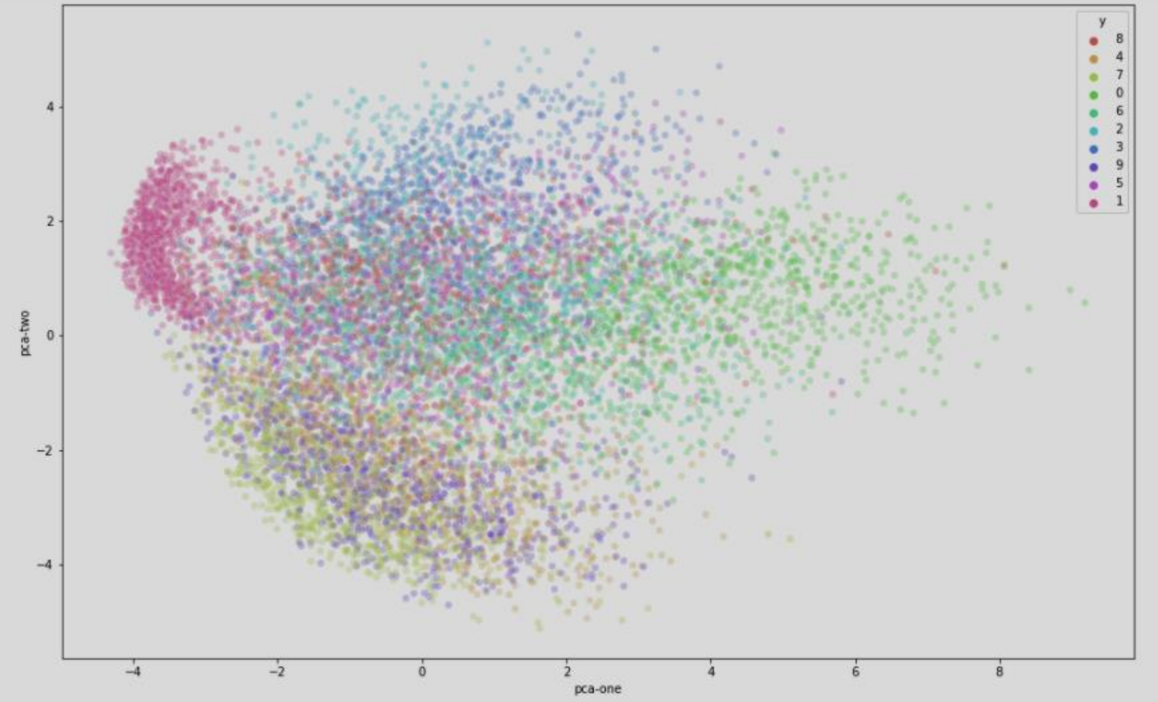
Importance of Data



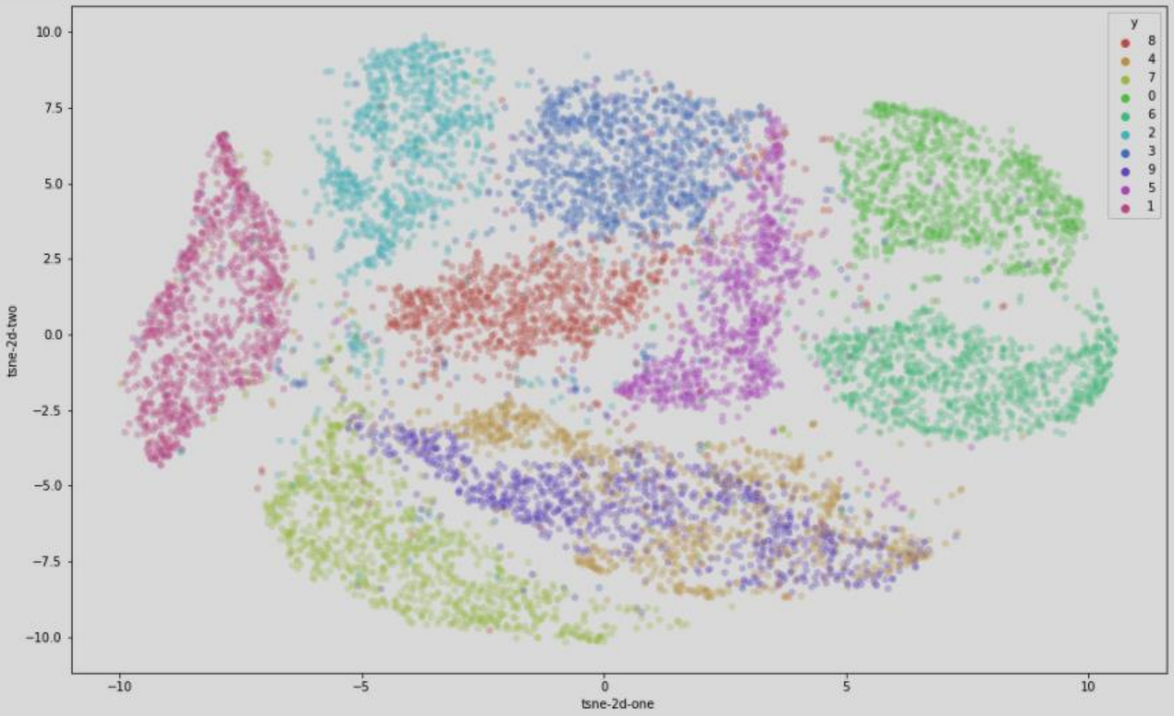
Imagine that your face recognition network learn all human faces on Earth.
Imagine that your hand pose estimation network learn all hand poses on Earth.

Visualizing data

Two popular data visualization techniques: We visualize 784-dimensional data from MNIST in the 2-dimensional space.



PCA



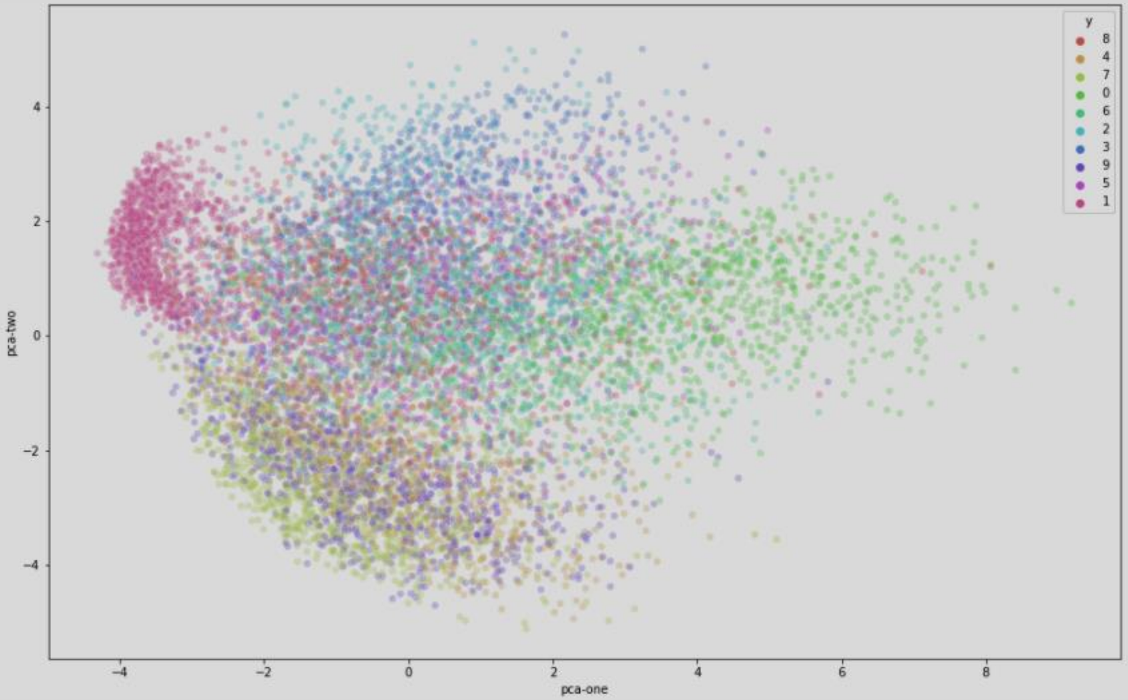
t-SNE

Visualizing data

```
from __future__ import print_function
import time
import numpy as np
import pandas as pd

from sklearn.datasets import fetch_openml
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns

mnist = fetch_openml('mnist_784', version=1, cache=True)
X = mnist.data / 255.0
y = mnist.target
print(X.shape, y.shape)

feat_cols = [ 'pixel'+str(i) for i in range(X.shape[1]) ]
df = pd.DataFrame(X,columns=feat_cols)
df['y'] = y
df['label'] = df['y'].apply(lambda i: str(i))
N = 10000
df_subset = df.loc[rndperm[:N],:].copy()
data_subset = df_subset[feat_cols].values
```

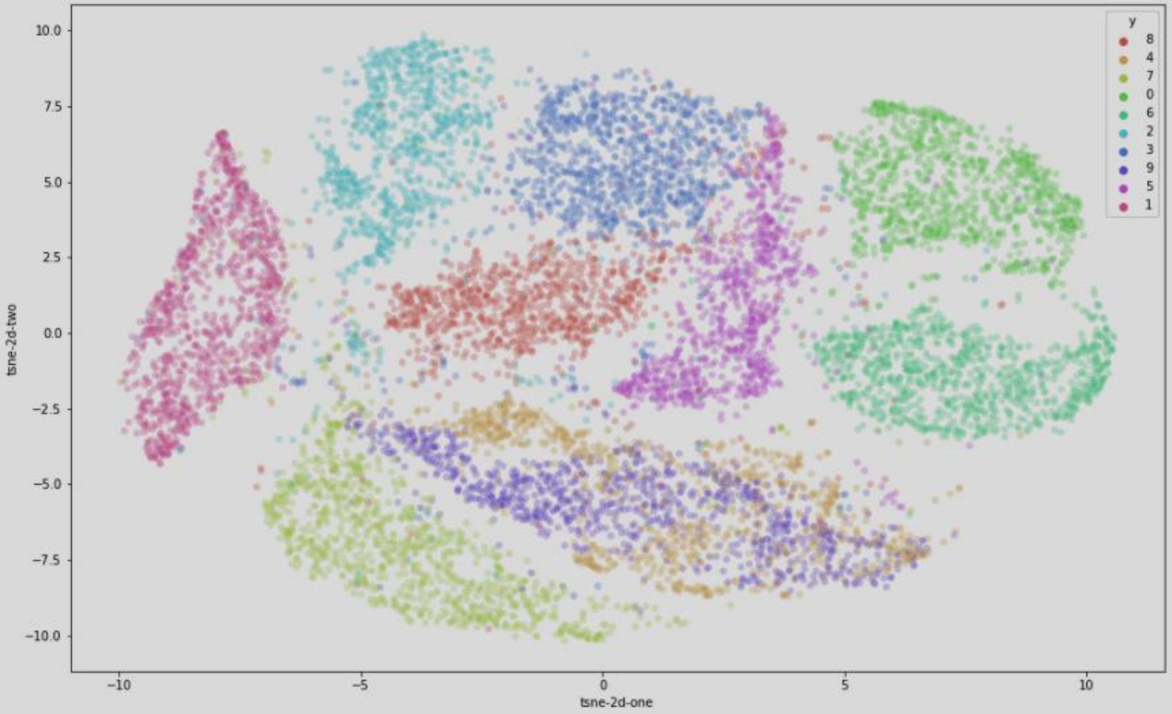
Visualizing data - PCA

```
np.random.seed(42)
rndperm = np.random.permutation(df.shape[0])

pca = PCA(n_components=2)
pca_result = pca.fit_transform(data_subset)
df_subset['pca-one'] = pca_result[:,0]
df_subset['pca-two'] = pca_result[:,1]

plt.figure(figsize=(16,10))
sns.scatterplot(
    x="pca-one", y="pca-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df.loc[rndperm,:],
    legend="full",
    alpha=0.3
)
```

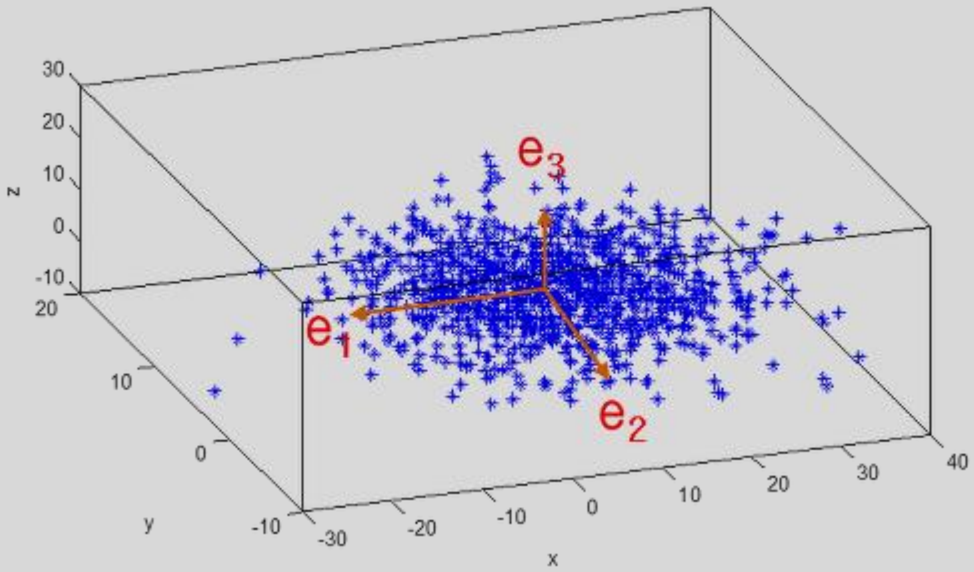


Visualizing data - tSNE

```
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
tsne_results = tsne.fit_transform(data_subset)

df_subset['tsne-2d-one'] = tsne_results[:,0]
df_subset['tsne-2d-two'] = tsne_results[:,1]

plt.figure(figsize=(16,10))
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3
)
```

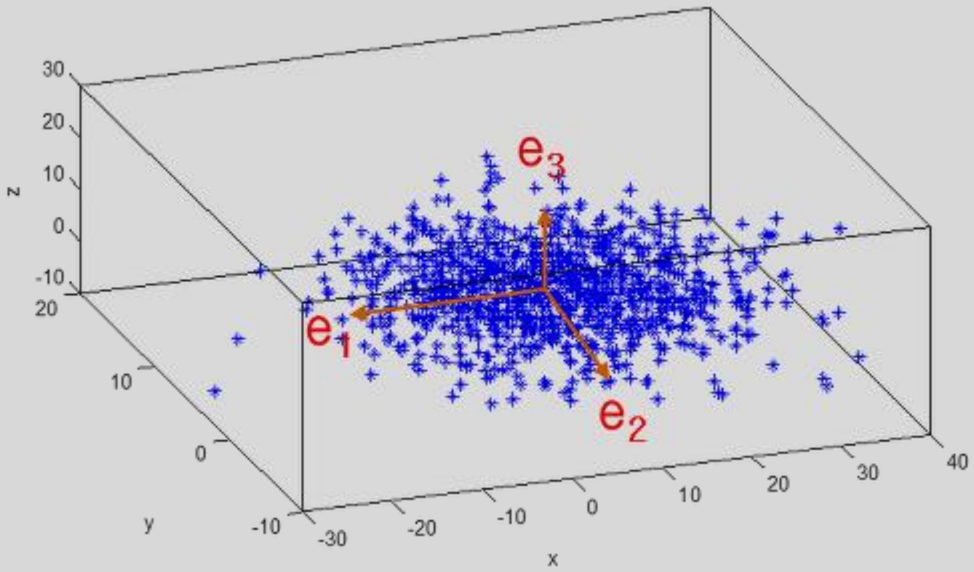


Visualizing the data - PCA



Algorithm to find principal components for data distribution.

Visualizing the data - PCA



What is the principal component here? → Axis, that can represent the largest variance.

Axes are orthogonal to each other.

Visualizing the data - PCA

$$X = \begin{pmatrix} | & | & | & & | \\ X_1 & X_2 & X_3 & \cdots & X_d \\ | & | & | & & | \end{pmatrix} \in \mathbb{R}^{n \times d}$$

X : data matrix

$$X^T X = \begin{pmatrix} — & X_1 & — \\ — & X_2 & — \\ & \cdots & \\ — & X_d & — \end{pmatrix} \begin{pmatrix} | & | & & | \\ X_1 & X_2 & \cdots & X_d \\ | & | & & | \end{pmatrix}$$

$$= \begin{pmatrix} dot(X_1, X_1) & dot(X_1, X_2) & \cdots & dot(X_1, X_d) \\ dot(X_2, X_1) & dot(X_2, X_2) & \cdots & dot(X_2, X_d) \\ \vdots & \vdots & \ddots & \vdots \\ dot(X_d, X_1) & dot(X_d, X_2) & \cdots & dot(X_d, X_d) \end{pmatrix}$$

Visualizing the data - PCA

$$C=\begin{pmatrix} cov(x,x) & cov(x,y) \\ cov(x,y) & cov(y,y) \end{pmatrix}$$
$$=\begin{pmatrix} \frac{1}{n}\sum(x_i-m_x)^2 & \frac{1}{n}\sum(x_i-m_x)(y_i-m_y) \\ \frac{1}{n}\sum(x_i-m_x)(y_i-m_y) & \frac{1}{n}\sum(y_i-m_y)^2 \end{pmatrix}$$

$$\mathrm{Var}[X] = \mathrm{E}\begin{bmatrix} (X_1-\mathrm{E}[X_1])(X_1-\mathrm{E}[X_1]) & \dots & (X_1-\mathrm{E}[X_1])(X_K-\mathrm{E}[X_K]) \\ \vdots & \ddots & \vdots \\ (X_K-\mathrm{E}[X_K])(X_1-\mathrm{E}[X_1]) & \dots & (X_K-\mathrm{E}[X_K])(X_K-\mathrm{E}[X_K]) \end{bmatrix}$$
$$= \begin{bmatrix} \mathrm{E}\left[(X_1-\mathrm{E}[X_1])^2\right] & \dots & \mathrm{E}[(X_1-\mathrm{E}[X_1])(X_K-\mathrm{E}[X_K])] \\ \vdots & \ddots & \vdots \\ \mathrm{E}[(X_K-\mathrm{E}[X_K])(X_1-\mathrm{E}[X_1])] & \dots & \mathrm{E}\left[(X_K-\mathrm{E}[X_K])^2\right] \end{bmatrix}$$
$$= \begin{bmatrix} \mathrm{Var}[X_1] & \dots & \mathrm{Cov}[X_1,X_K] \\ \vdots & \ddots & \vdots \\ \mathrm{Cov}[X_K,X_1] & \dots & \mathrm{Var}[X_K] \end{bmatrix}$$

Visualizing the data - PCA

Eigenvector Decomposition:

$$C = V \Lambda V^T$$

$$= \begin{bmatrix} v_1 & v_2 & \cdots & v_N \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_N \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_N^T \end{bmatrix}$$

$$= \begin{bmatrix} \lambda_1 v_1 & \lambda_2 v_2 & \cdots & \lambda_N v_N \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_N^T \end{bmatrix}$$

Visualizing the data - PCA

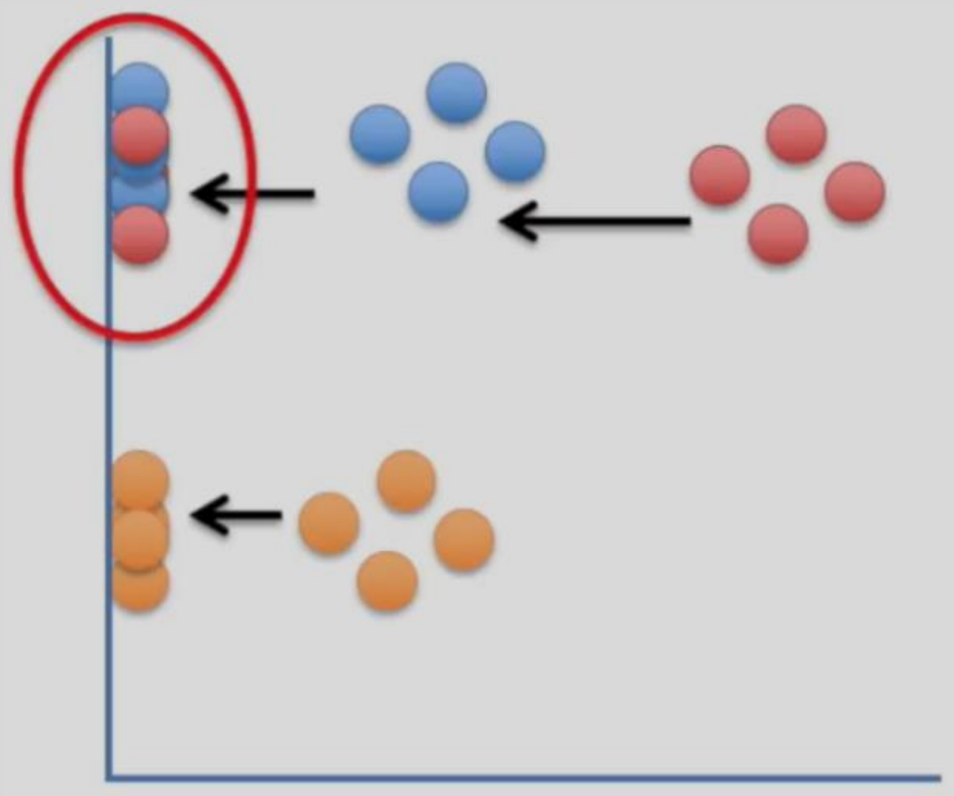
We find eigenvalue/eigenvector for the covariance matrix.

It is mathematically proven that the variance of data samples projected onto an eigenvector is equal to its eigenvalue.

So, we sort eigenvectors in the descending order according to its eigenvalues.

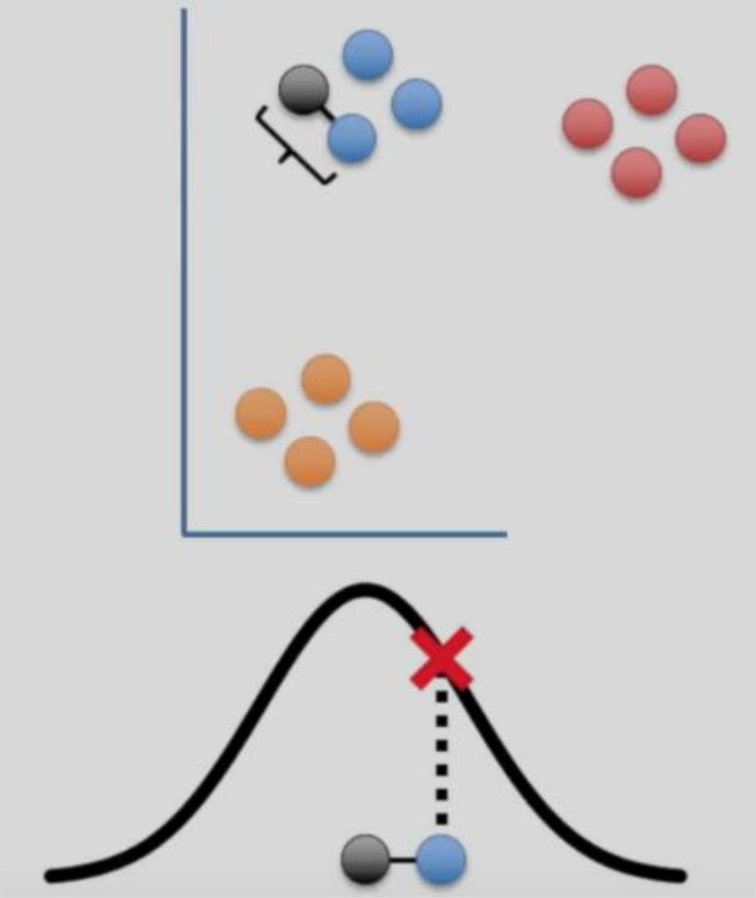
And we choose an eigenvector whose eigenvalue is the largest as the most principled axis e1, choose the second eigenvector as the second principled axis e2 and so on.

Visualizing the data - PCA



Data samples are linearly projected to the principal components.

Visualizing the data - tSNE



$$p_{j|i} = \frac{exp(-|x_i - x_j|^2/2\sigma^2)}{\sum_k exp(-|x_i - x_k|^2/2\sigma^2)}$$

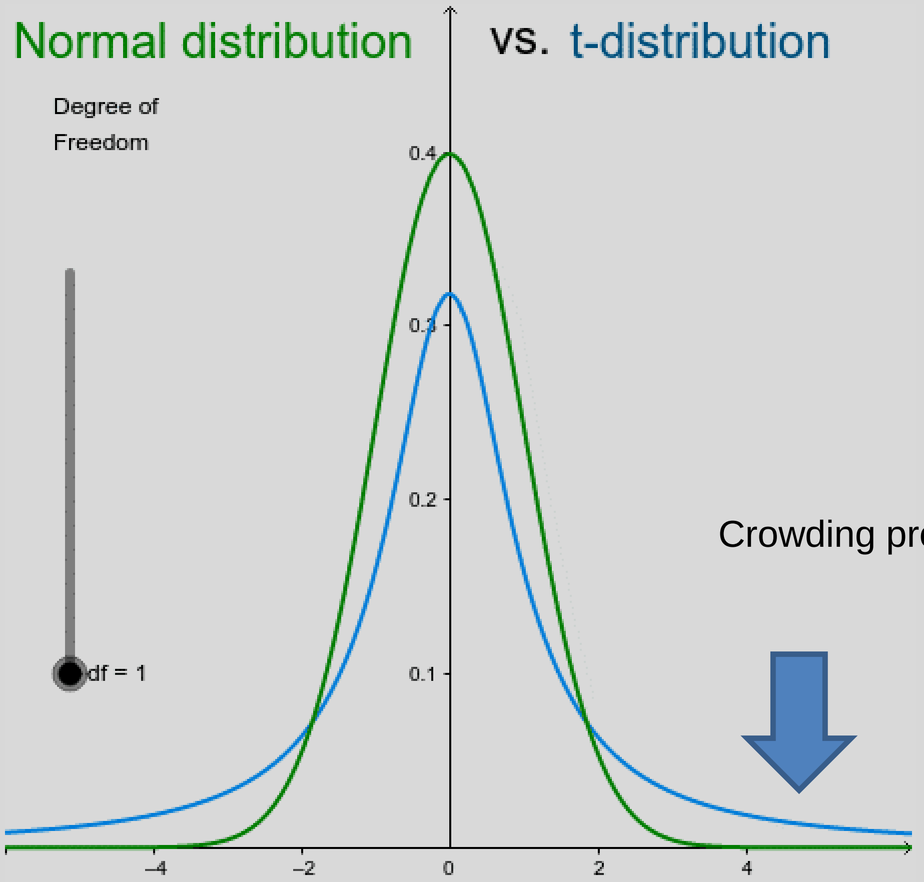
$$q_{j|i} = \frac{e^{-|y_i - y_j|^2}}{\sum_k e^{-|y_i - y_k|^2}}$$

Visualizing the data - tSNE

$$
\begin{aligned}
Cost &= \textstyle\sum_i KL(P_i || Q_i) \\
&= \textstyle\sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}
\end{aligned}
$$

Minimize the discrepancy between locations in original and reduced spaces.

Visualizing the data - tSNE



Crowding problem.

$$q_{j|i} = \frac{e^{-|y_i - y_j|^2}}{\sum_k e^{-|y_i - y_k|^2}}$$

$$q_{j|i} = \frac{\left(1 + |y_i - y_j|^2\right)^{-1}}{\sum_k \left(1 + |y_i - y_k|^2\right)^{-1}}$$

Visualizing large-scale dataset

train \ test	ICVL	NYU	MSRC	Bighand
ICVL	**12.3**	35.1	65.8	46.3
NYU	20.1	**21.4**	64.1	49.6
MSRC	25.3	30.8	**21.3**	49.7
BigHand	**14.9**	**20.6**	**43.7**	**17.1**

Table 3. Cross Benchmark comparison. Cross-benchmark average errors, trained with the *Big Hand* data set, the model performs well on ICVL and NYU, while training on ICVL, NYU, and MSRC does not generalize well to other benchmarks.

Dataset	No. frames
ICVL [26]	17,604
NYU [30]	81,009
MSRA15 [24]	76,375
BigHand	2.2M

http://bjornstenger.github.io/papers/yuan_cvpr2017.pdf

Visualizing large-scale dataset

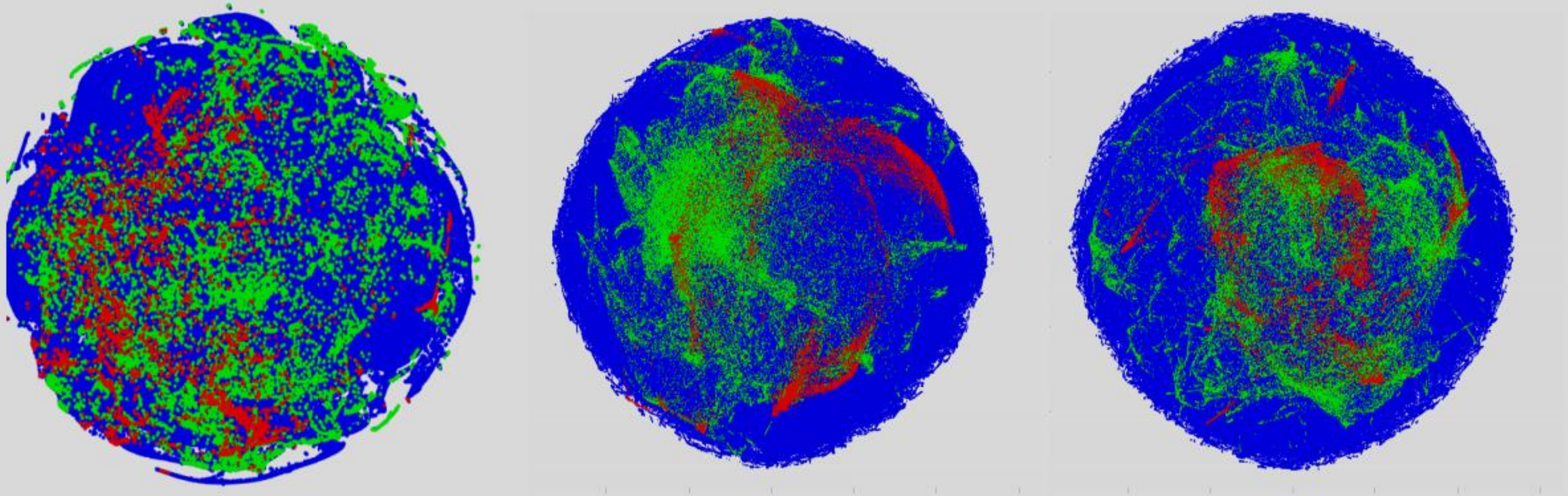
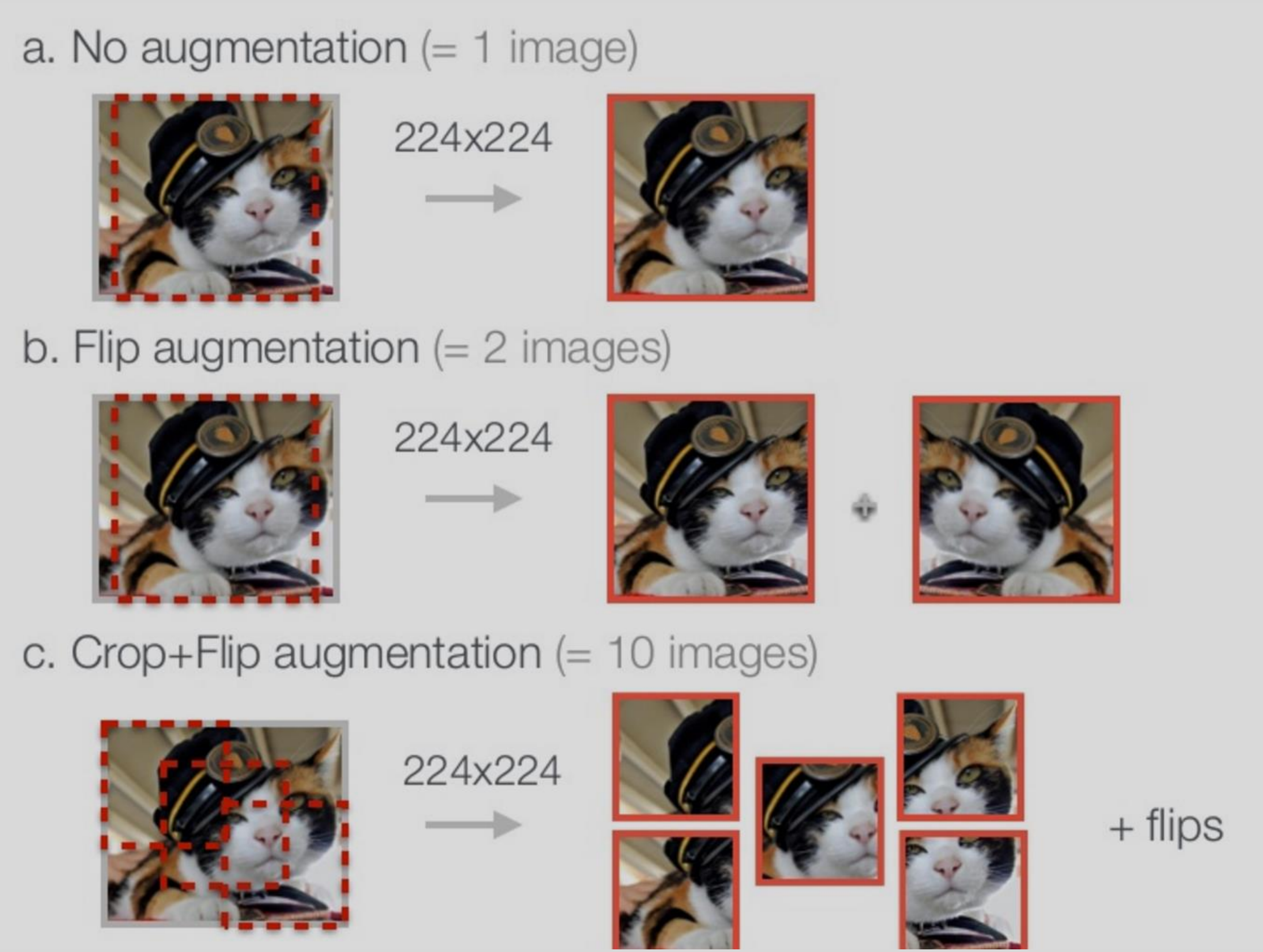


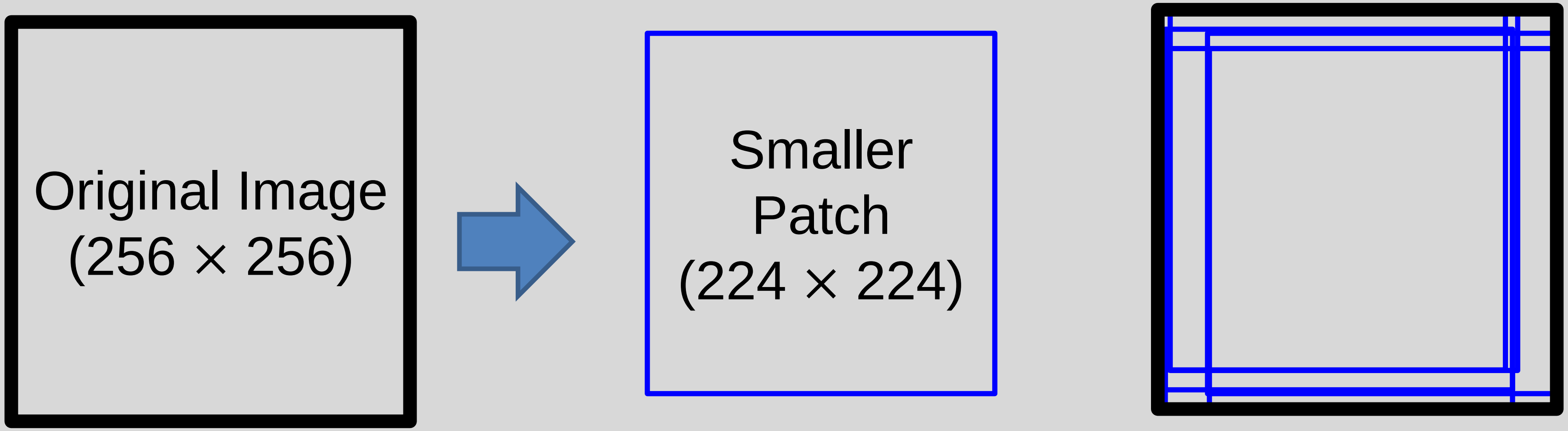
Figure 5. 2D t-SNE embedding of the hand pose space. Big Hand is represented by blue dots, ICVL is represented by red dots. NYU is represented by green dots. The figures show (left) global view point space coverage, (middle) articulation angle space (25D), and (right) hand angle (global orientation and articulation angles) coverage comparison. Compared with existing benchmarks, the *Big Hand* contains a wider range of variation.

http://bjornstenger.github.io/papers/yuan_cvpr2017.pdf

Simple data augmentation



Simple data augmentation



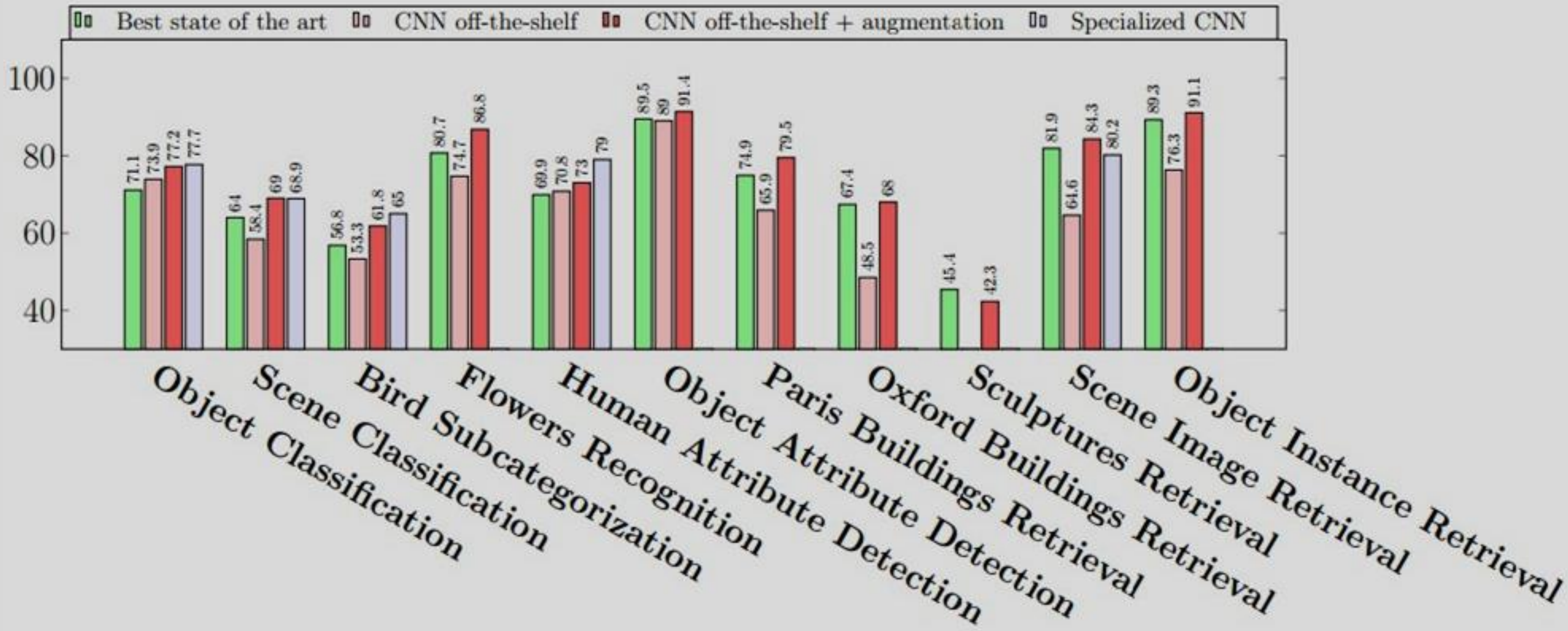
This increases the size of the training set by a **factor of 2048** $(32 * 32 * 2)$.

Simple data augmentation

Original Patch (224×224) $\xrightarrow{\text{Color variation}}$ Altered Patch (224×224)

Probabilistically, not a single patch will be same at the training phase! (a **factor of infinity**!)

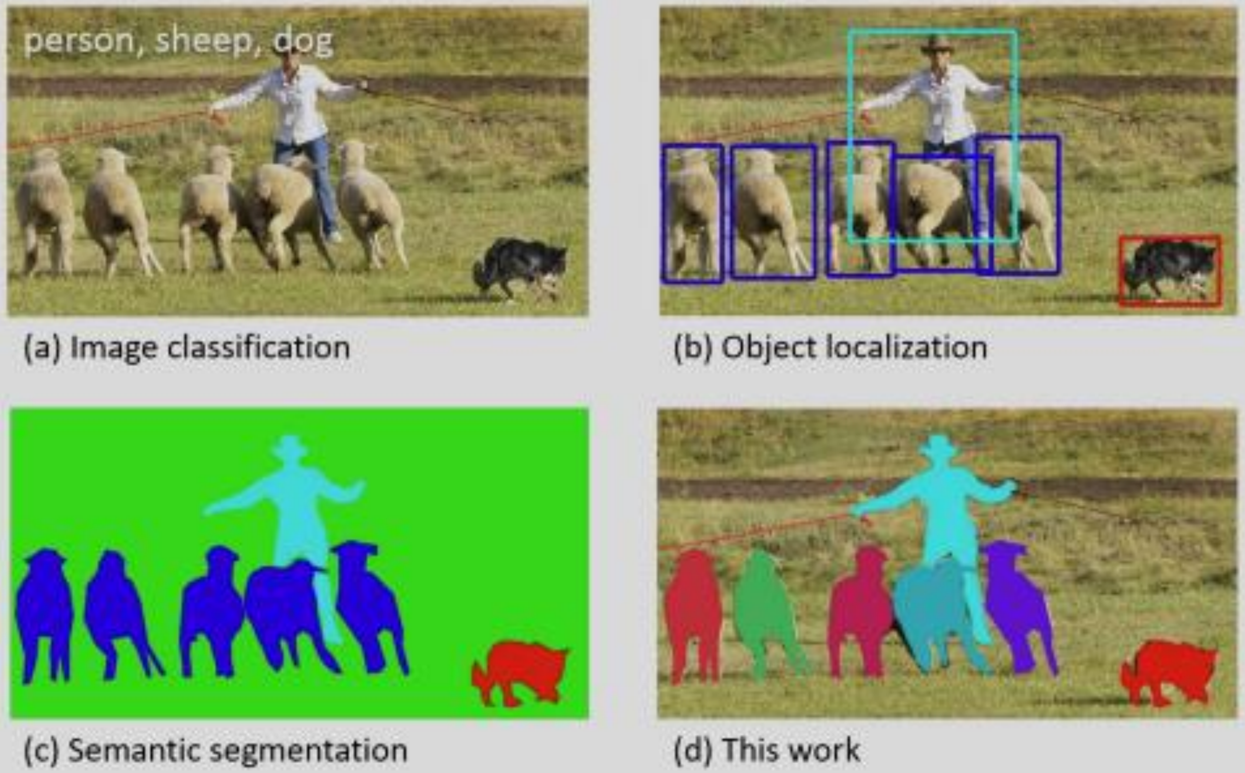
Simple data augmentation



Simple data augmentation

```
import PIL
import numpy as np
import torch
import torchvision
import torchvision.datasets as datasets
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
import cv2

transforms = torchvision.transforms.Compose([
    torchvision.transforms.Resize((224,224)),
    torchvision.transforms.ColorJitter(hue=.05, saturation=.05),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.RandomRotation(20, resample=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor()
])

for i in range(4):
  train_dataset = datasets.MNIST(root = 'mnist_data', train=True, transform=transforms, download=True)
  train_loader = DataLoader(dataset=train_dataset, batch_size=1, shuffle=False)
  for x, y in train_loader:
    break
  R = np.stack((x[0,0]*255.,x[0,0]*255.,x[0,0]*255.), axis=2)
  cv2_imshow(R)
```

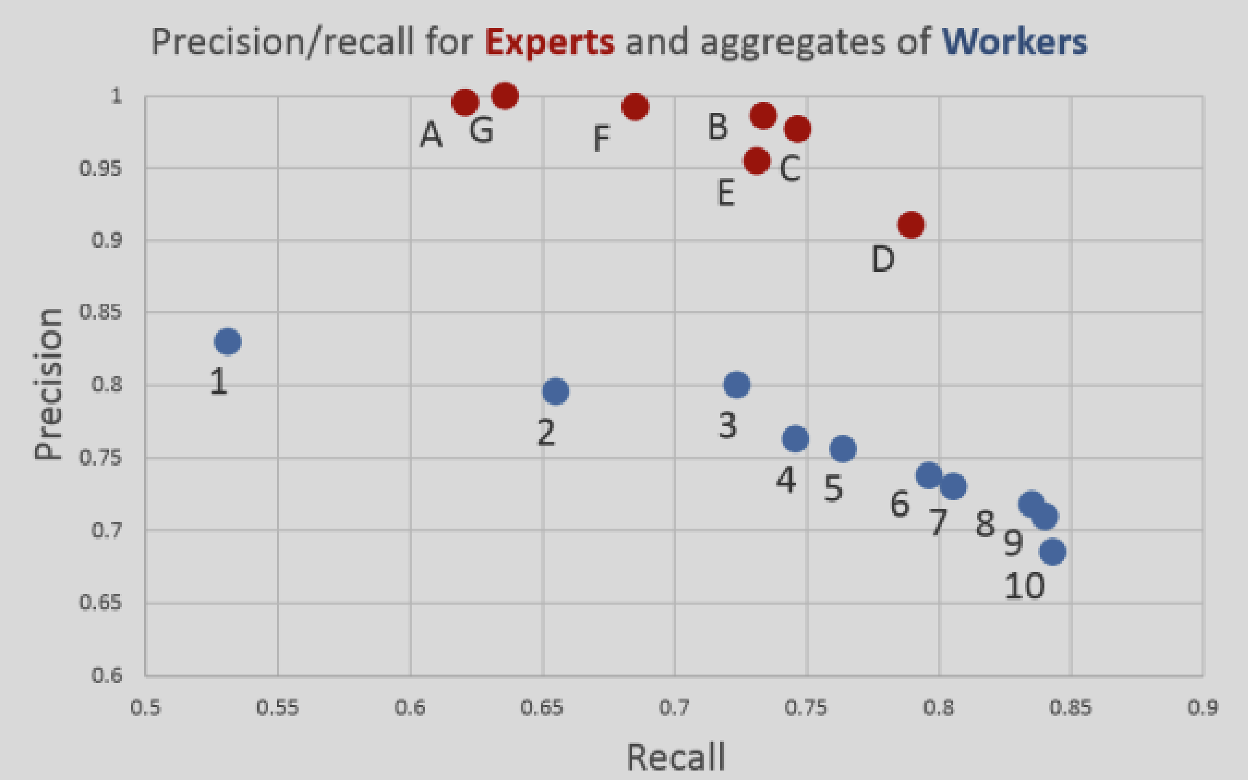
MS COCO dataset



Microsoft COCO: Common Objects in Context, ECCV'14

- 328,000 images, 2.5 million object instances from 91 categories.
- Crowd-sourced data by Amazon’s Mechanical Turk (AMT).

MS COCO dataset



- **8 AMT workers** were used to collect data.
- Independent worker annotate well with over 50% prob. $0.5^8 \rightarrow 0.004$
- Ground-truths are generated by majority vote of experts.

MS COCO dataset

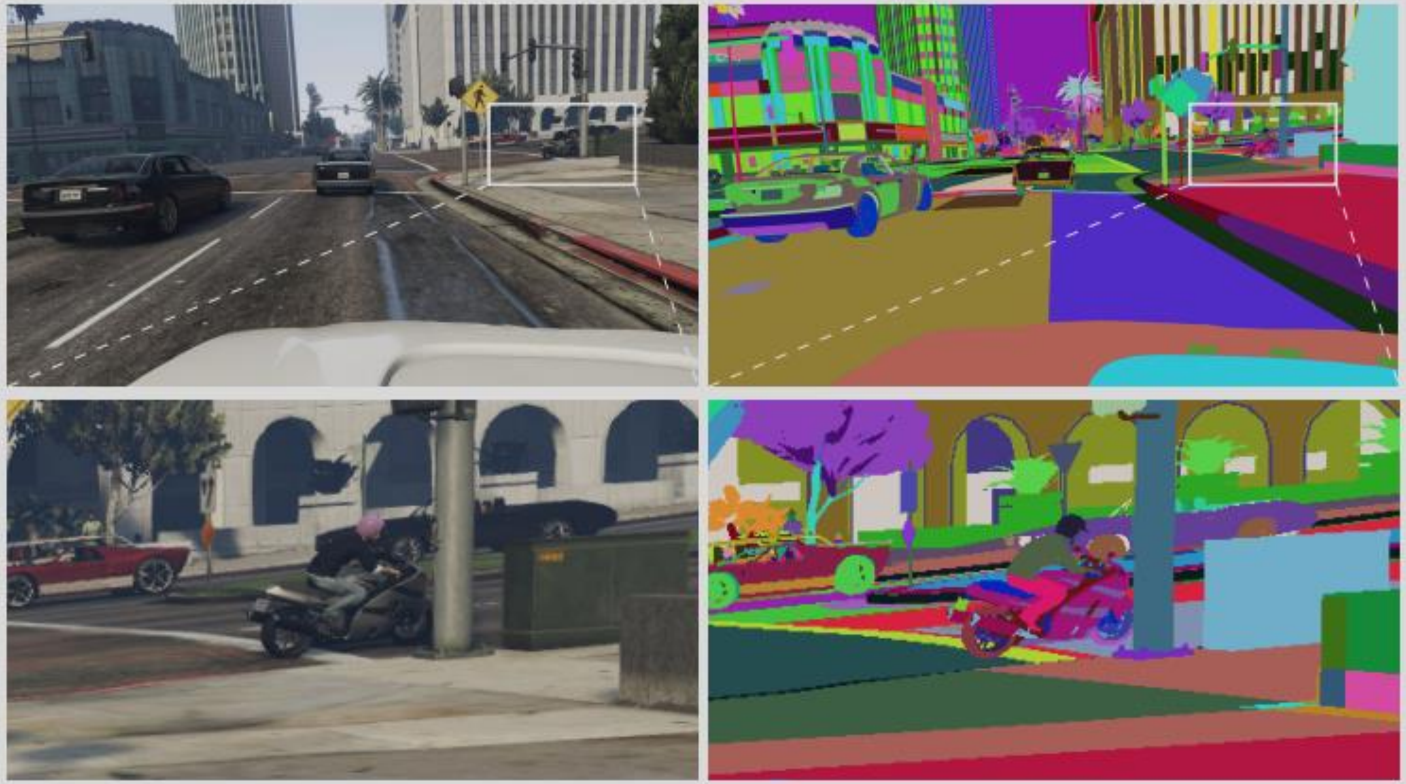
- 22 worker hours per 1,000 segmentations

Synthetic data

Learning from synthetic humans, CVPR’17.

- Use graphics rendering engines to obtain large-scale datasets.

Synthetic data

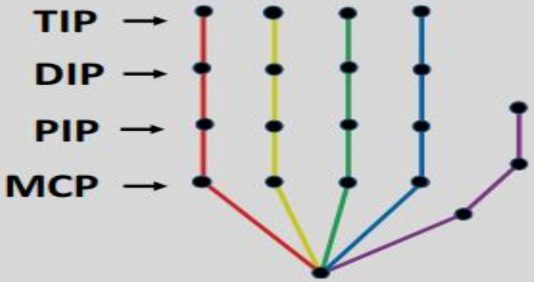


Playing for Data: Ground Truth from Computer Games, ECCV'16.

- Use graphics rendering engines to obtain large-scale datasets.

3D reconstruction task

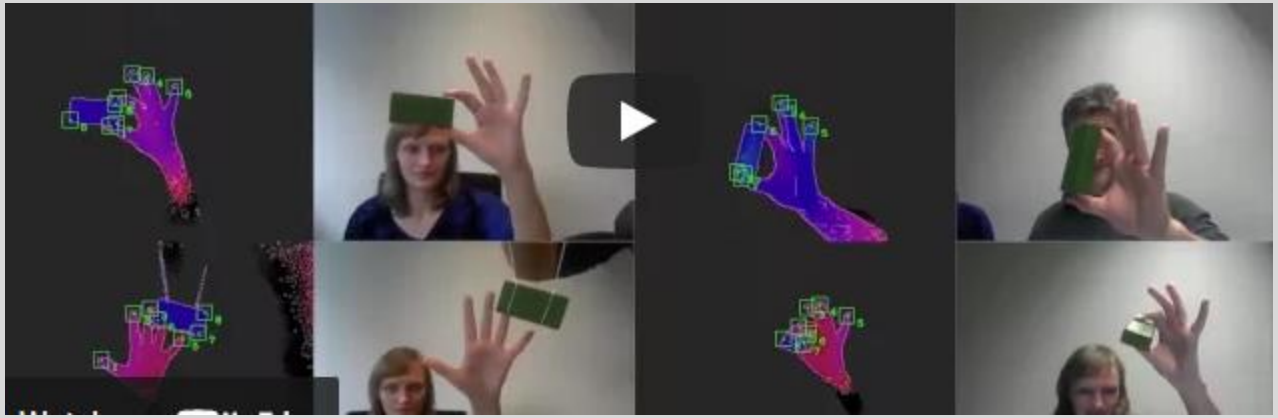

Body Skeleton




Hand Skeletons

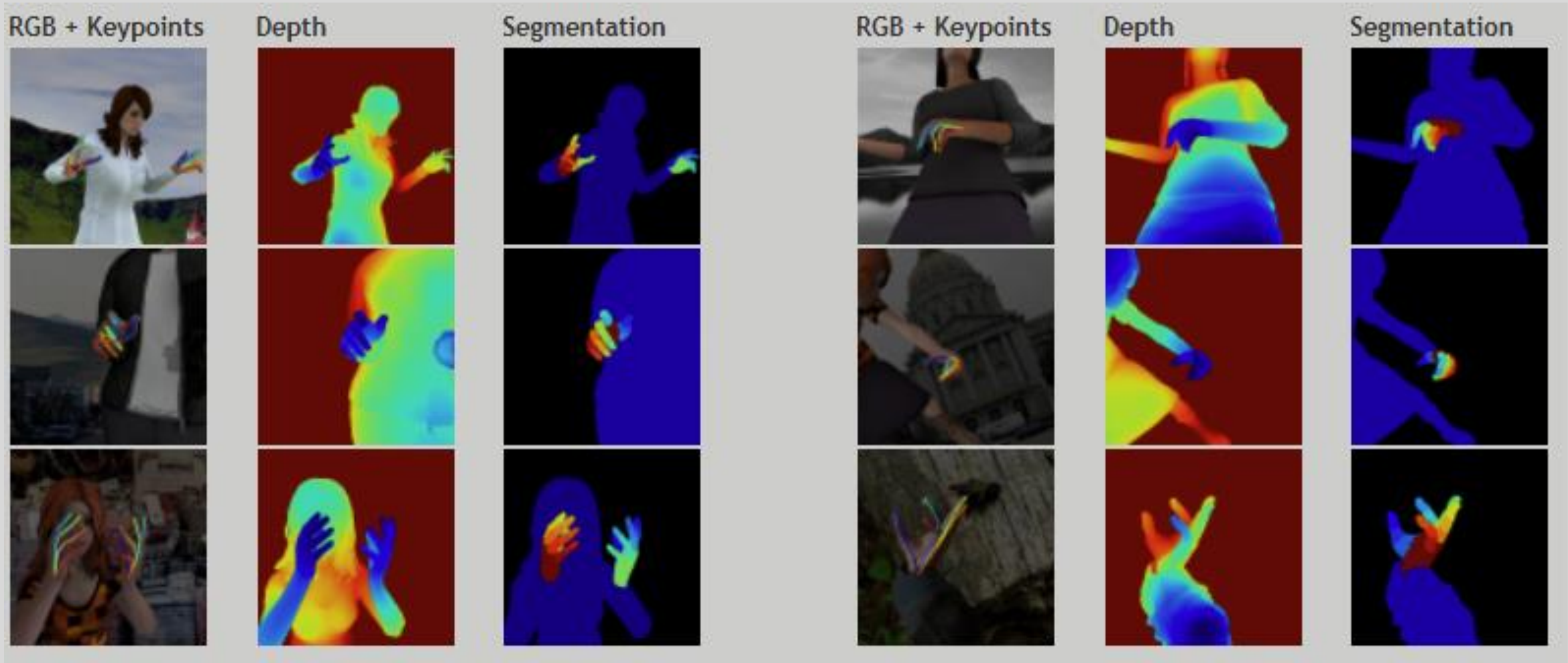

Body Mesh


Hand Mesh


Real-time Joint Tracking of a Hand Manipulating an Object from RGB-D Input, ECCV'16

- Non-trivial to collect large-scale accurate annotations via manual efforts.

Synthetic data



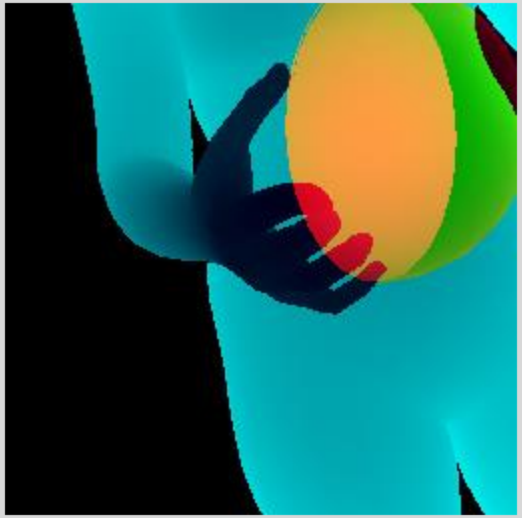
Learning to Estimate 3D Hand Pose from Single RGB Images, ICCV'17.

- Use graphics rendering engines to obtain large-scale datasets.

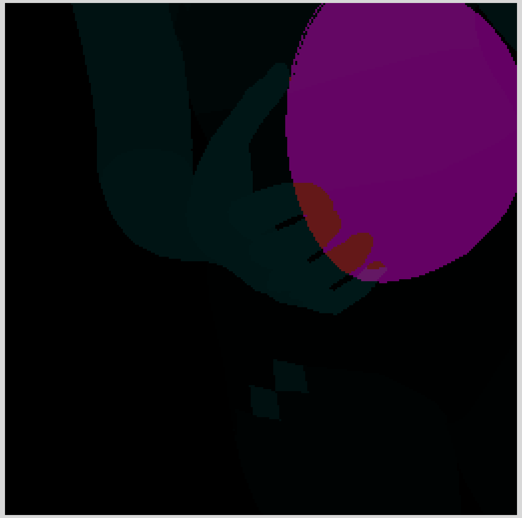
Data generation

RGB image with objects

RGB image without objects

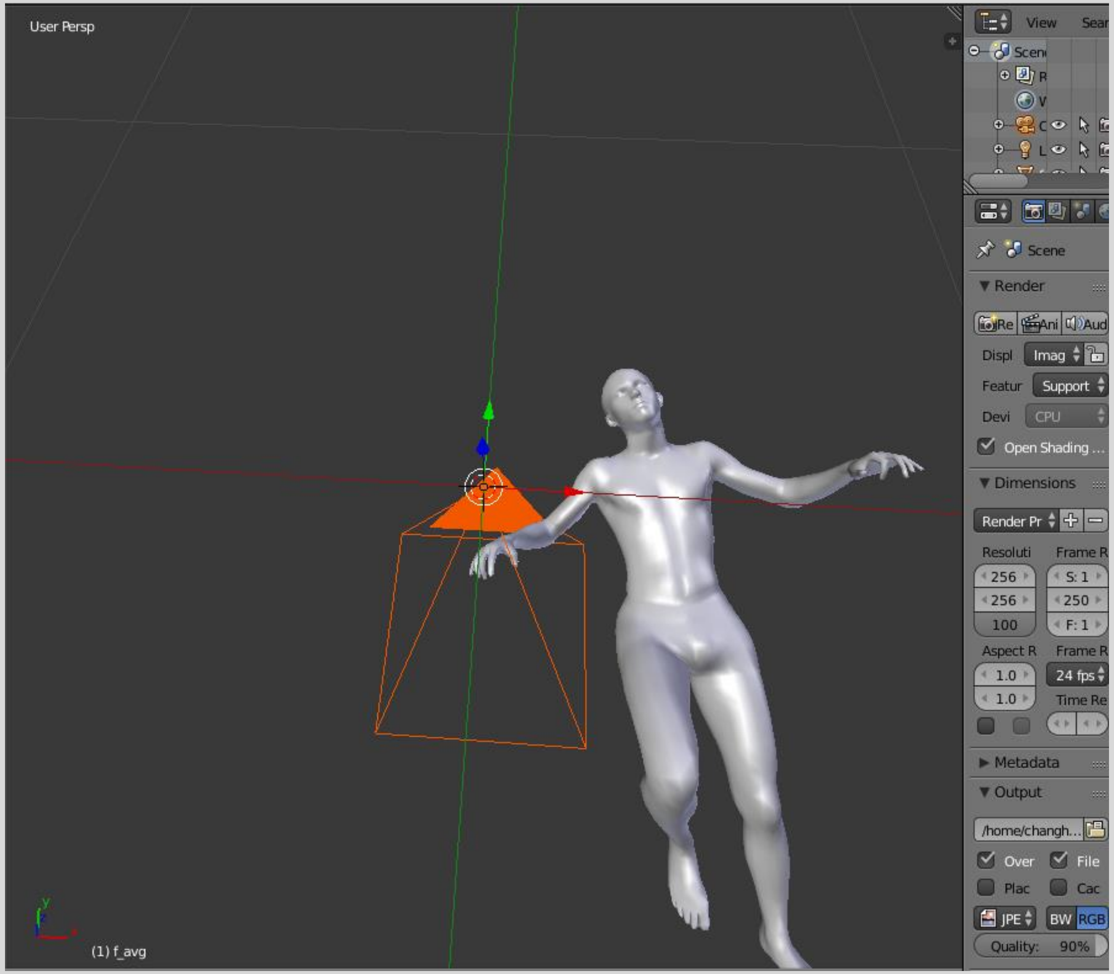


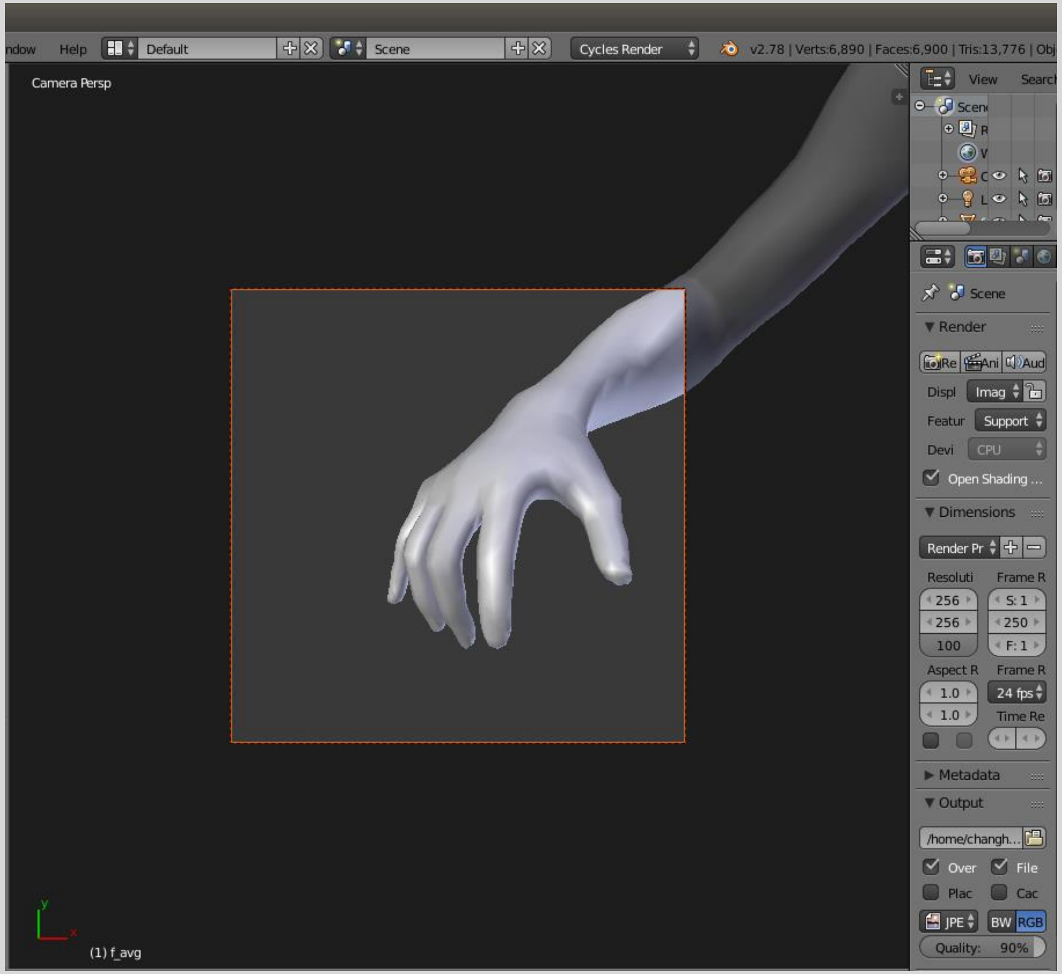
Depth image



Segmentation mask

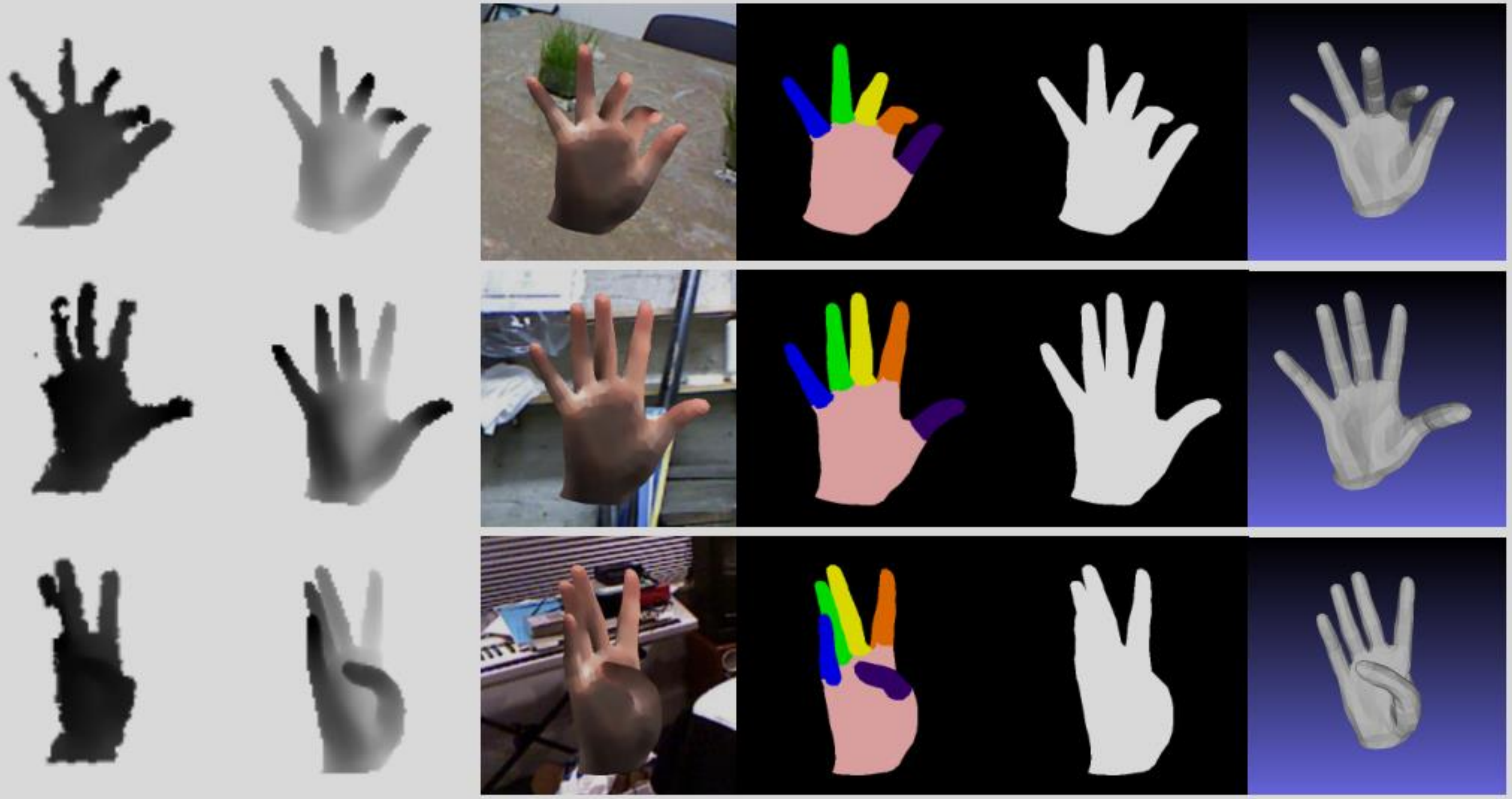
Data generation





Data generation

Graphics model

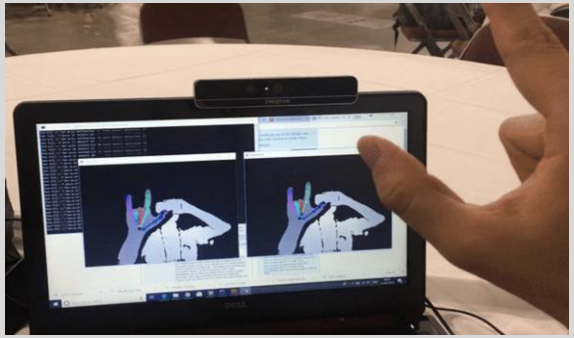


Gap between real and synthetic

1. Testing data should be real; it means we will use the trained machine/deep learning models on the real-world testing dataset.

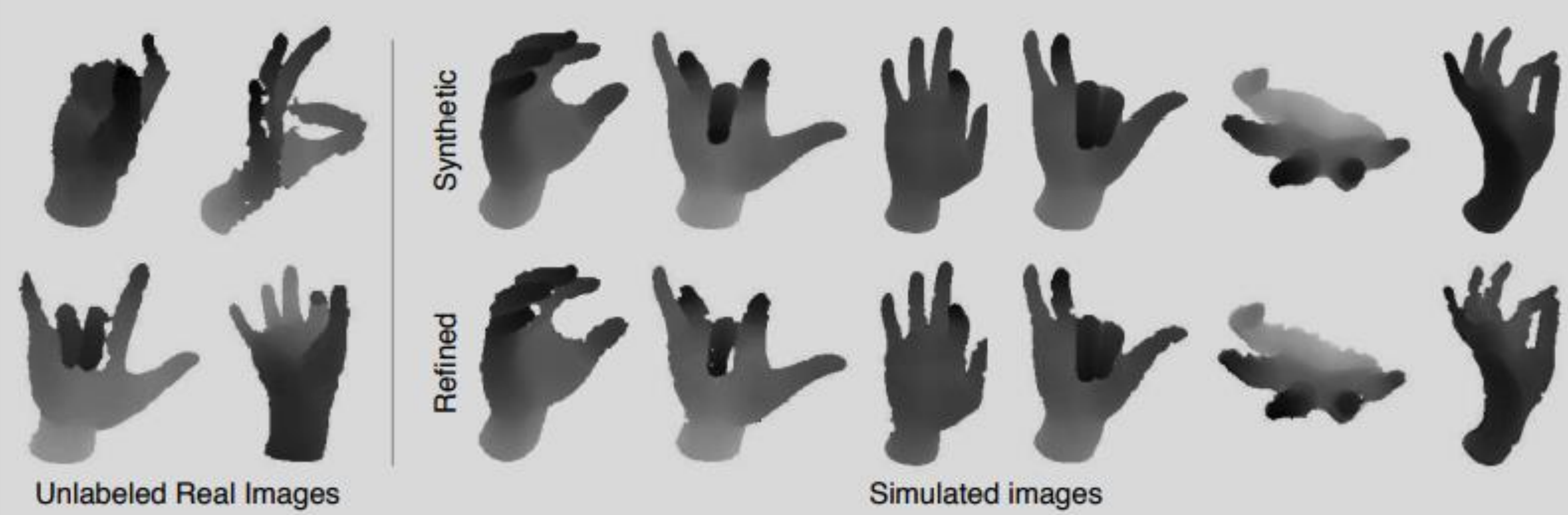
2. Training could involve synthetic dataset; however machine/deep learning models trained on synthetic dataset does not always generalize well to the real-world dataset, due to the domain gap.

Train model using any supervision.
Using synthetic+real datasets.

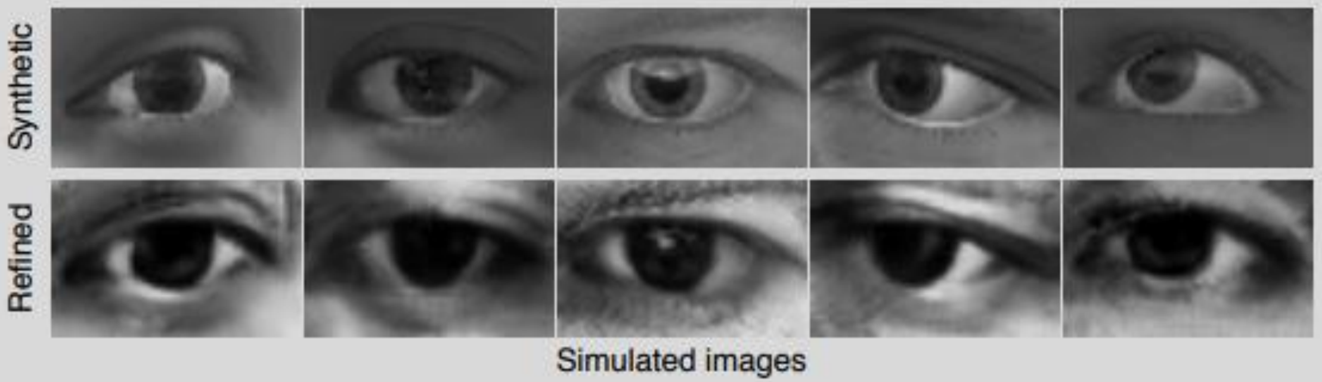


Test for the real hands.

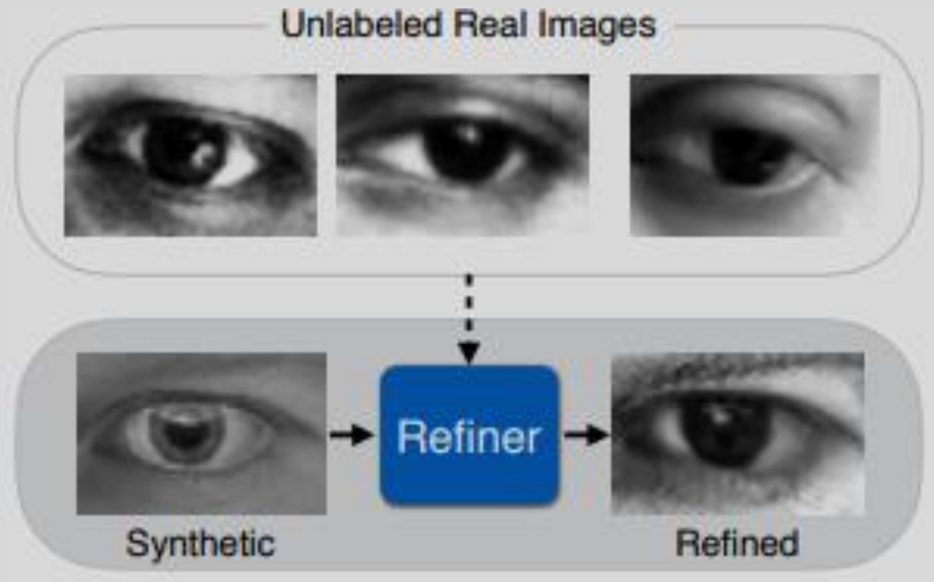
Gap between real and synthetic



https://openaccess.thecvf.com/content_cvpr_2017/papers/Shrivastava_Learning_From_Simulated_CVPR_2017_paper.pdf



Gap between real and synthetic



Training data	% of images within d
Synthetic Data	69.7
Refined Synthetic Data	72.4
Real Data	74.5
Synthetic Data 3x	77.7
Refined Synthetic Data 3x	**83.3**

Table 4. Comparison of a hand pose estimator trained on synthetic data, real data, and the output of SimGAN. The results are at distance $d = 5$ pixels from ground truth.

Thank you!