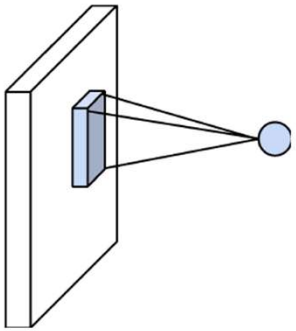
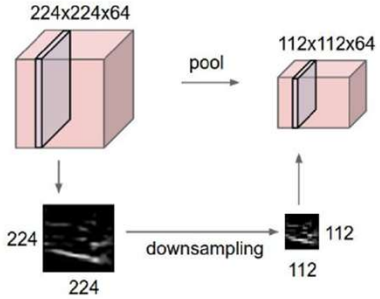
Components of CNNs

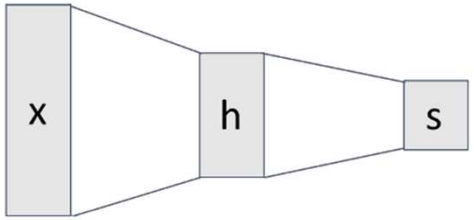
Convolution Layers



Pooling Layers

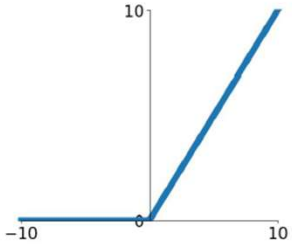


Fully-Connected Layers



Question: How should we put them together?

Activation Function

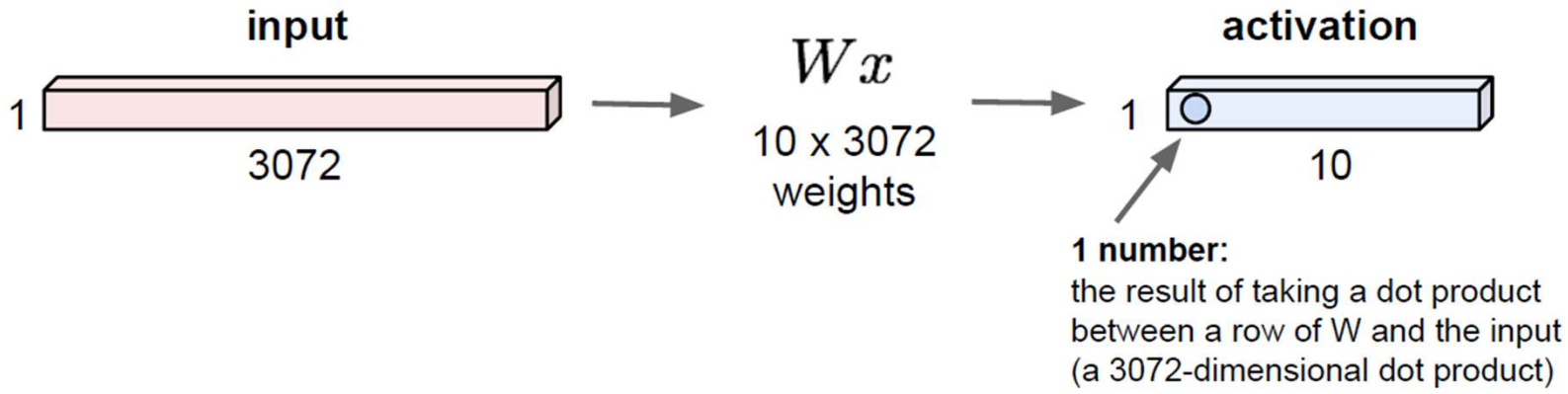


Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Fully Connected Layer

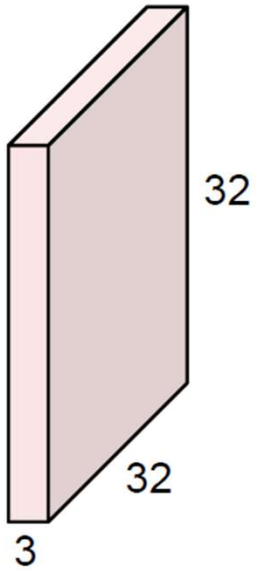
32x32x3 image -> stretch to 3072 x 1



Convolution Layer

Filters always extend the full depth of the input volume

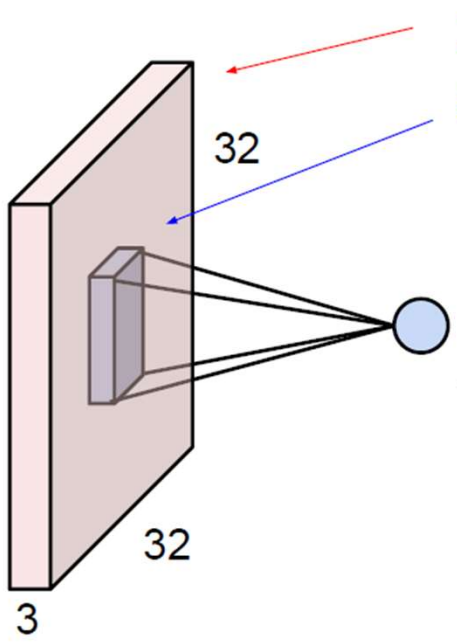
32x32x3 image



5x5x3 filter

Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Convolution Layer

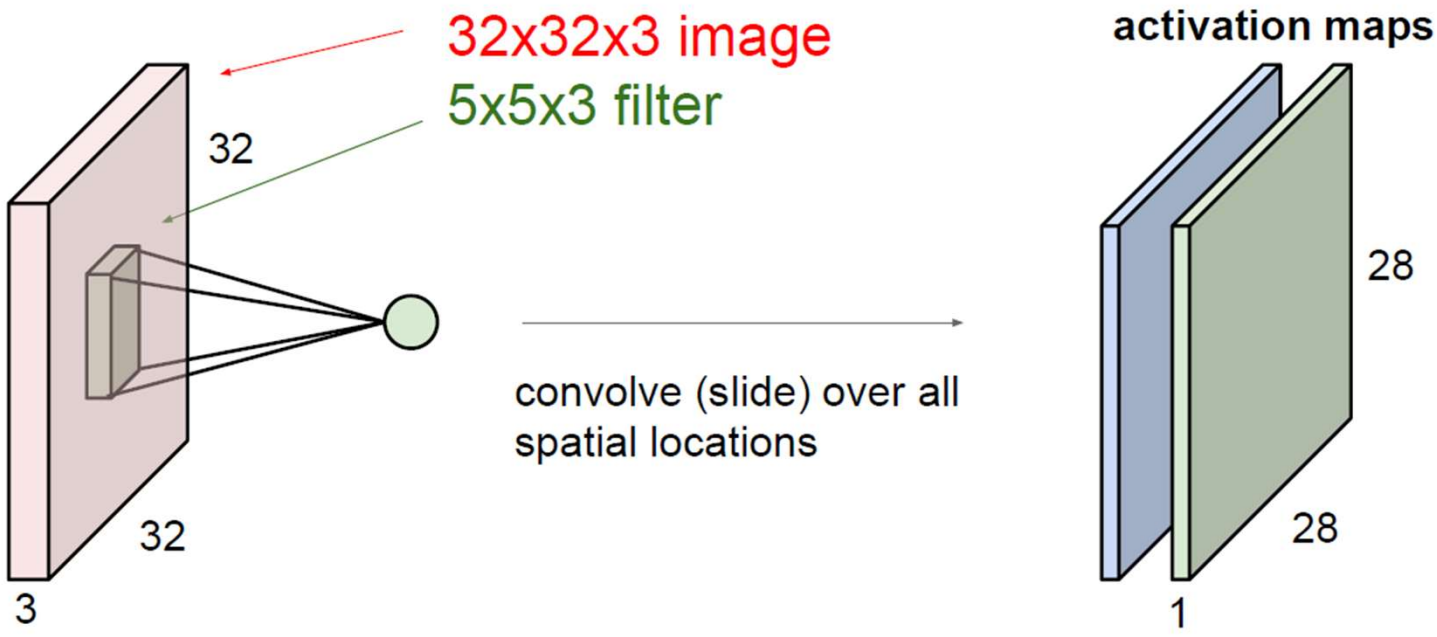


1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

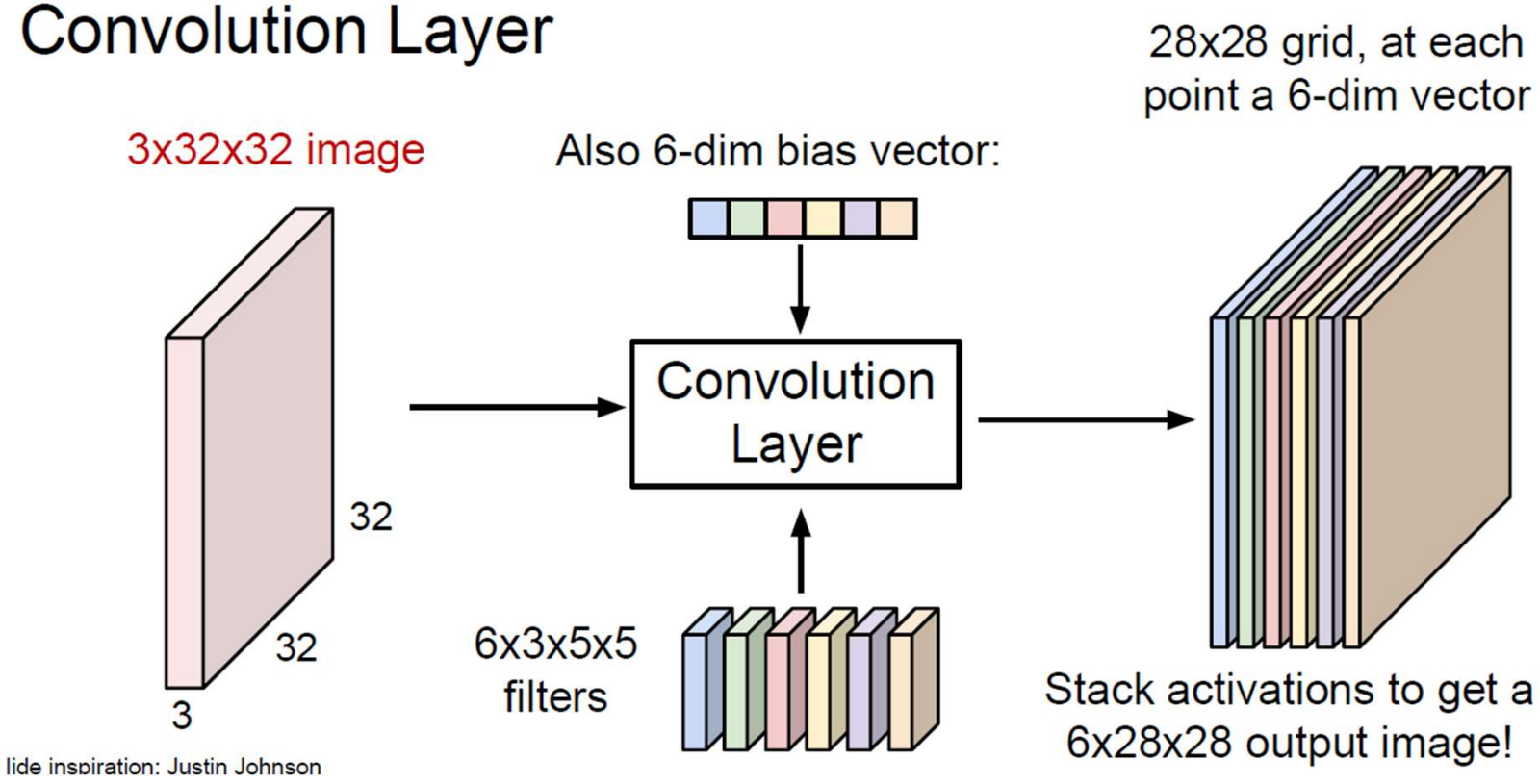
$$w^T x + b$$

Convolution Layer

consider a second, green filter

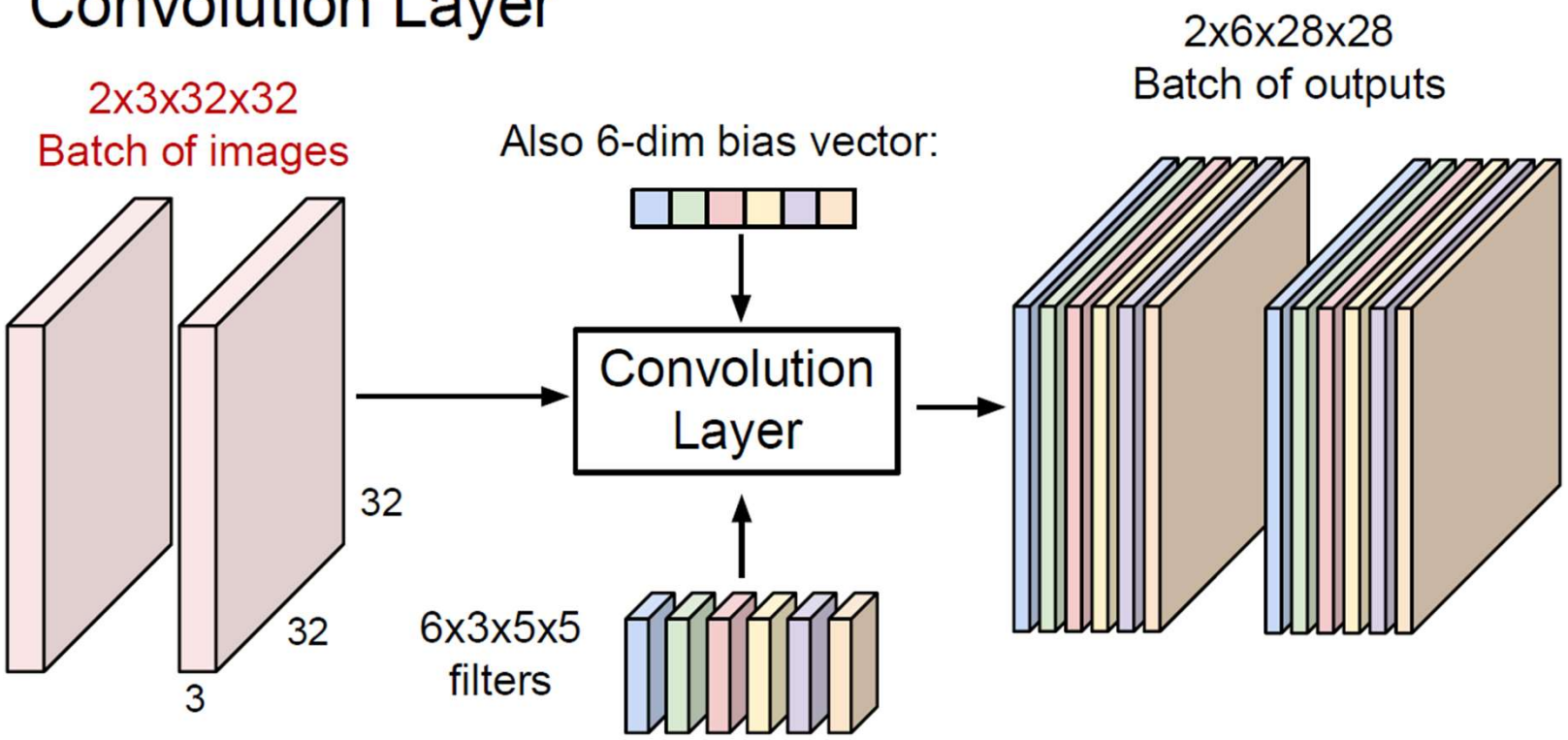


Convolution Layer

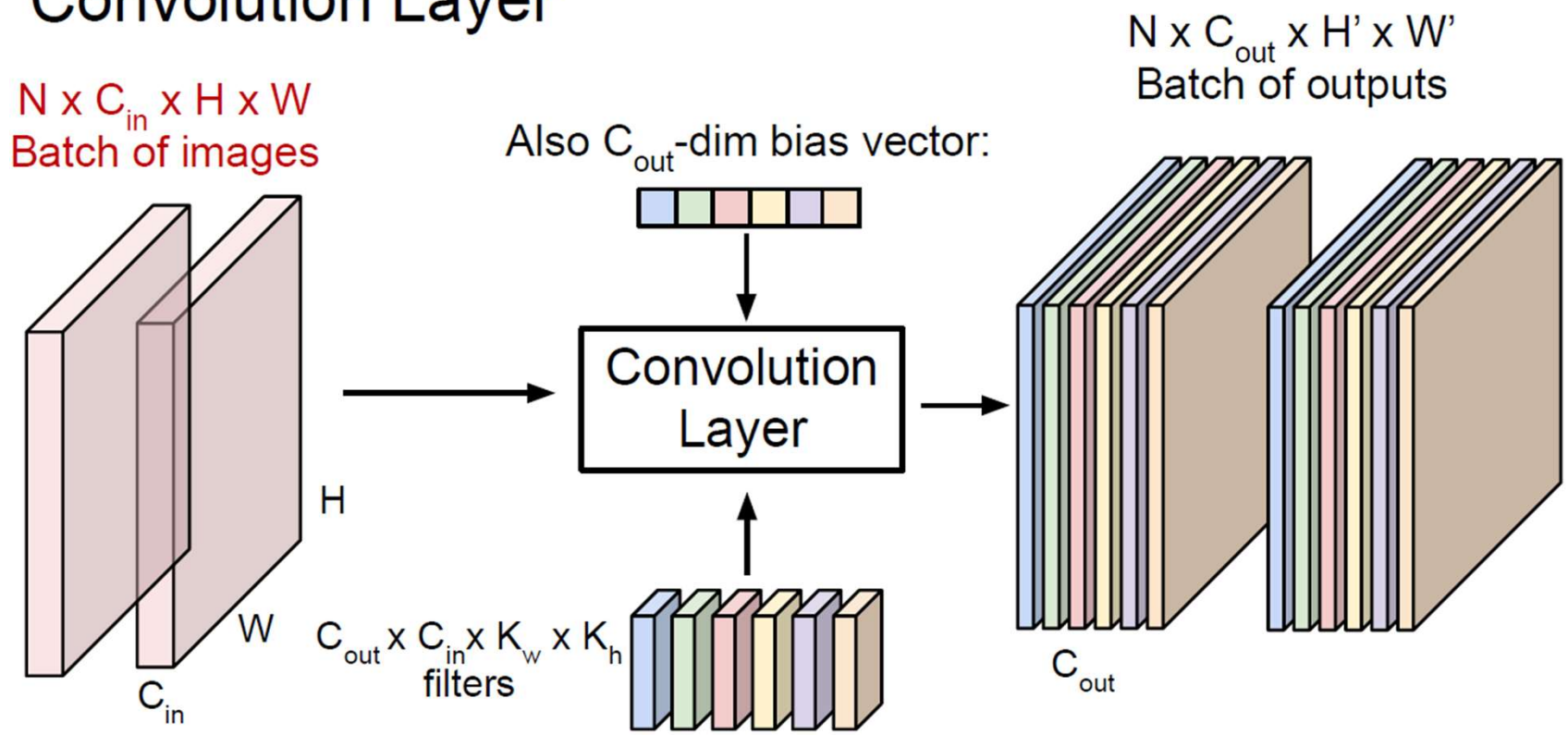


lide inspiration: Justin Johnson

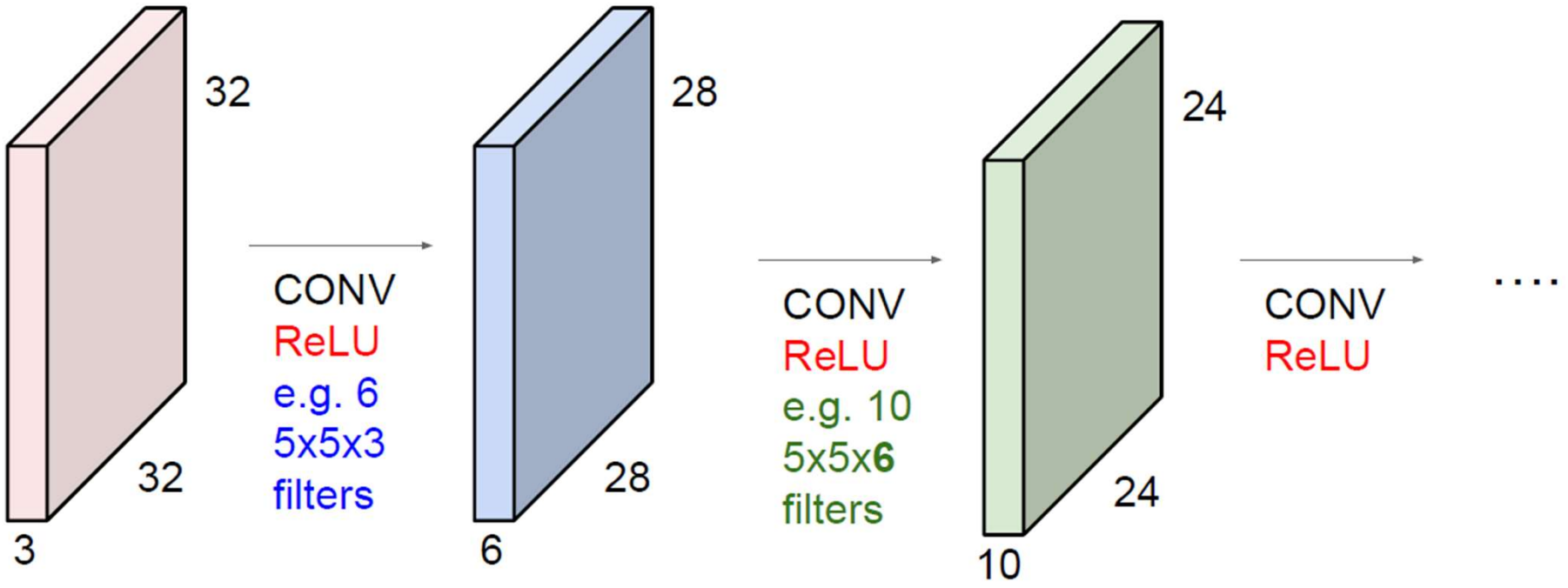
Convolution Layer



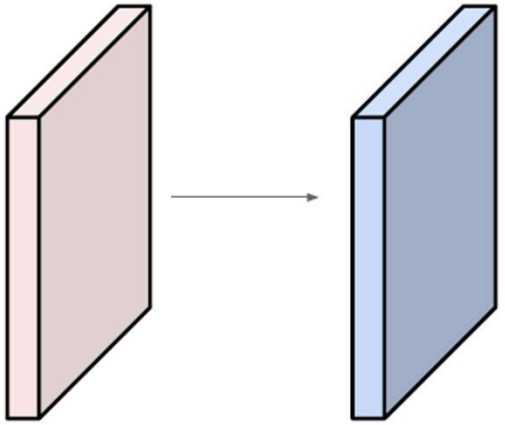
Convolution Layer



Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



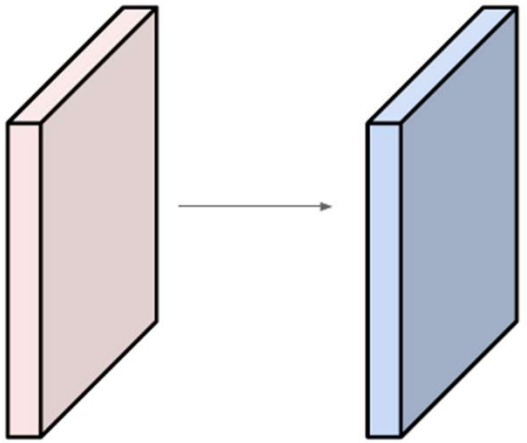
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

Examples time:

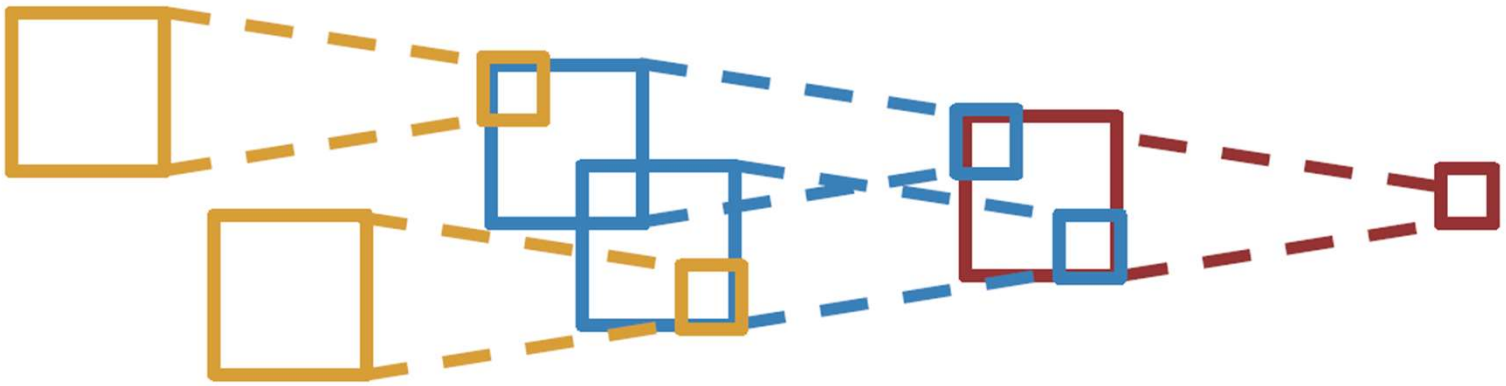


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Receptive Fields

Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)



Input

Output

Problem: For large images we need many layers for each output to "see" the whole image image

Solution: Downsample inside the network

Slide inspiration: Justin Johnson

Convolution layer: summary

Let's assume input is W_1 x H_1 x C

Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of W_2 x H_2 x K

where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

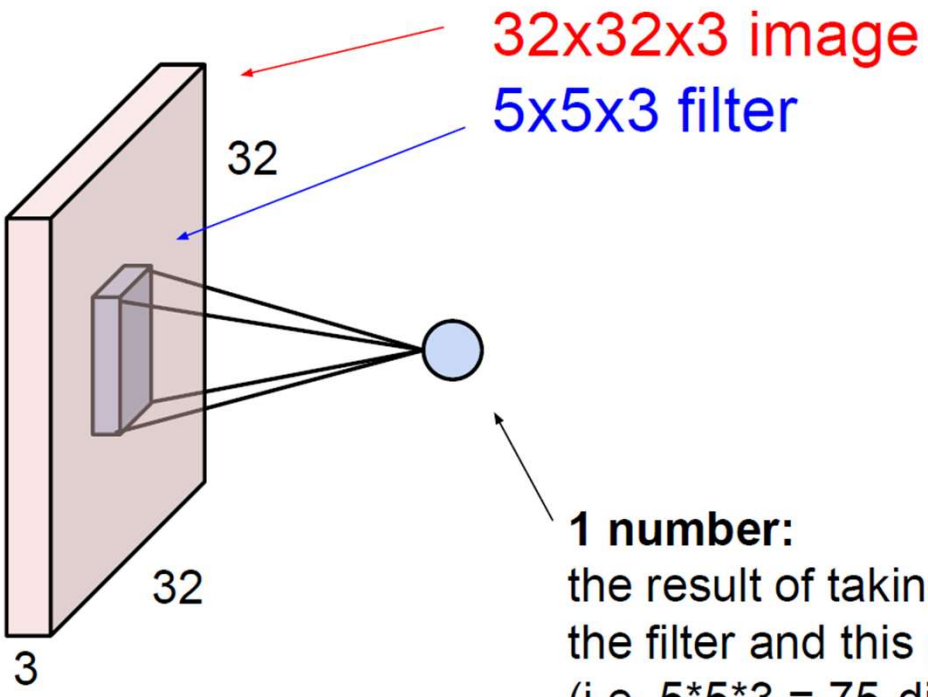
Number of parameters: F^2CK and K biases

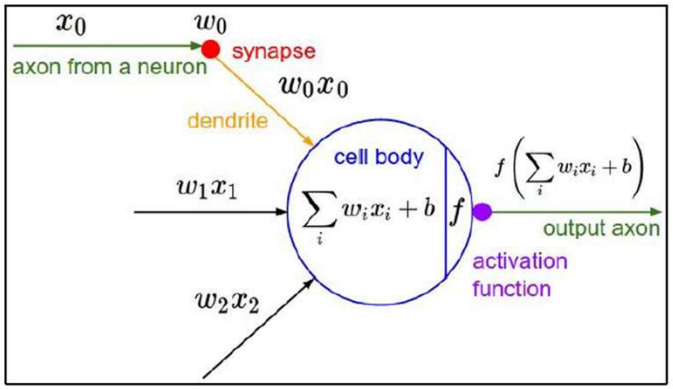
Common settings:

K = (powers of 2, e.g. 32, 64, 128, 512)

- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

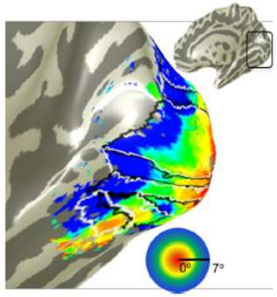
The brain/neuron view of CONV Layer



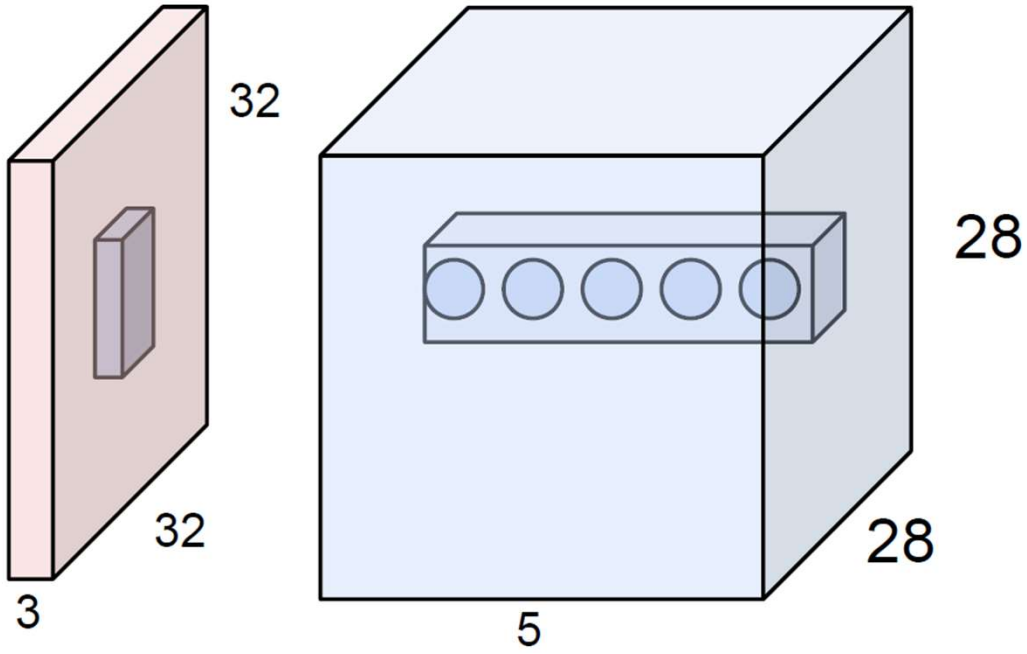


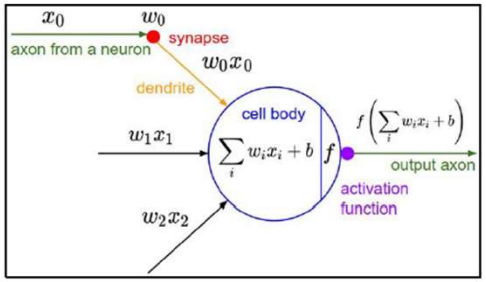
It's just a neuron with local connectivity...

1 number:
the result of taking a dot product between the filter and this part of the image
(i.e. 5*5*3 = 75-dimensional dot product)



The brain/neuron view of CONV Layer



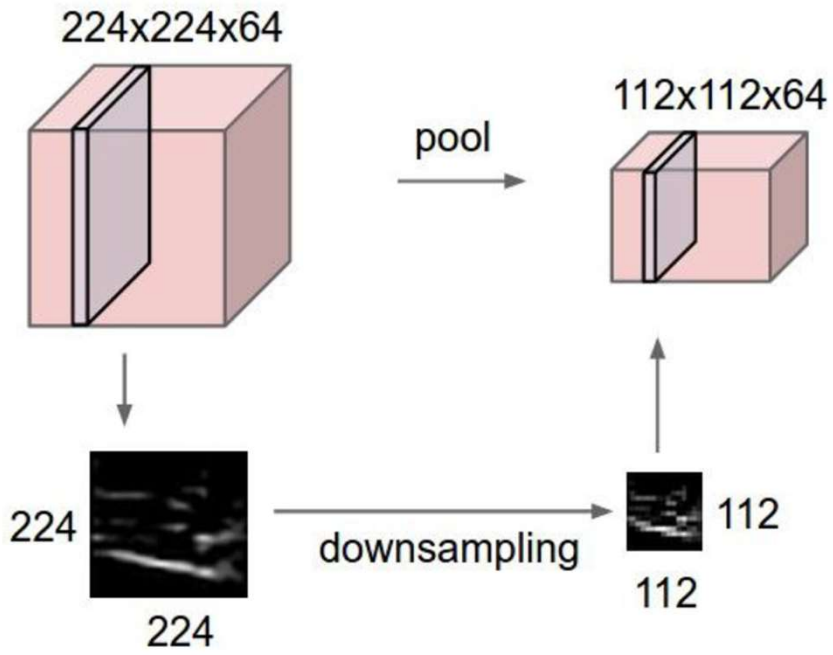


E.g. with 5 filters,
CONV layer consists of neurons arranged in a 3D grid (28x28x5)

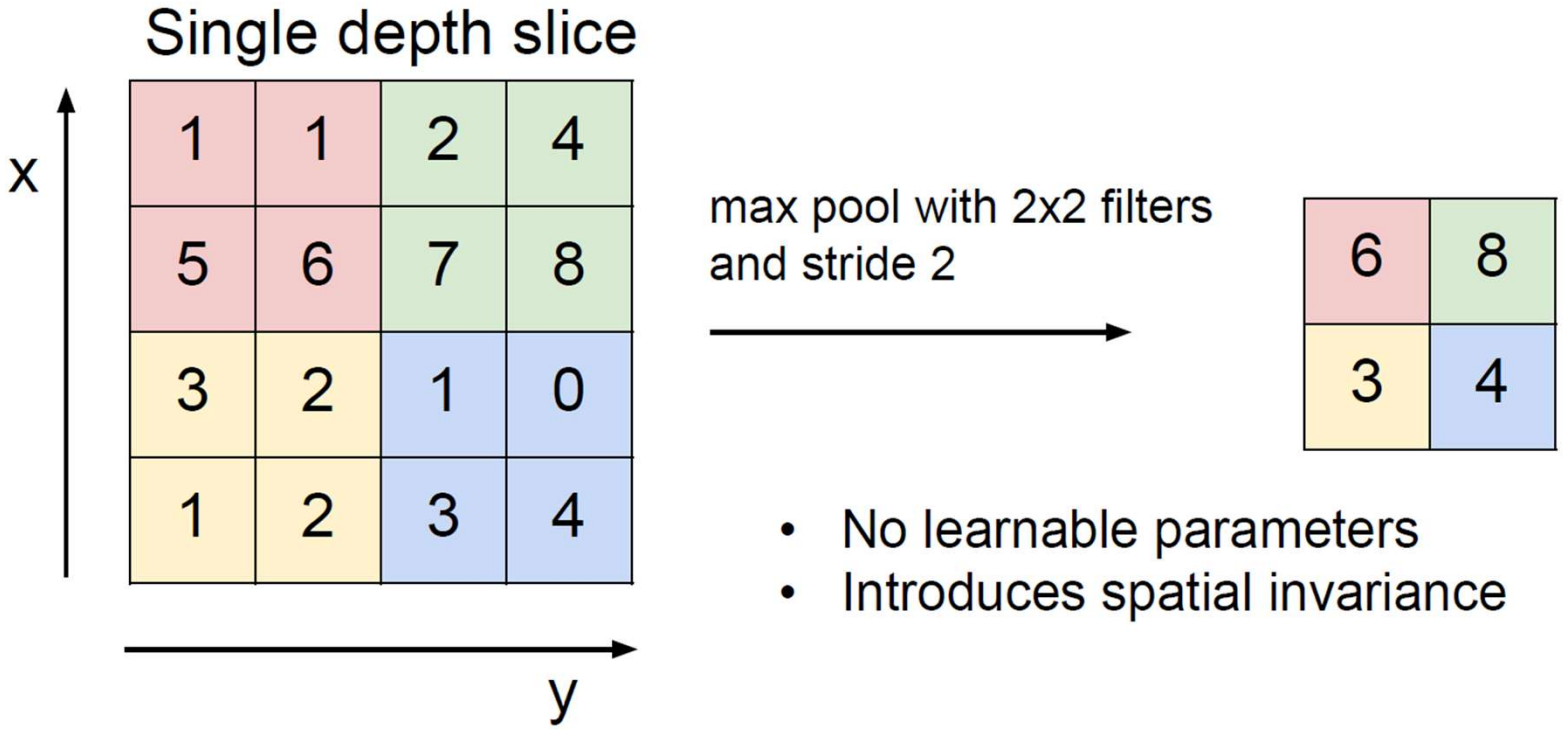
There will be 5 different neurons all looking at the same region in the input volume

Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently



MAX POOLING



- No learnable parameters
- Introduces spatial invariance

Pooling layer: summary

Let's assume input is W_1 x H_1 x C
Conv layer needs 2 hyperparameters:

- The spatial extent **F**
- The stride **S**

This will produce an output of W_2 x H_2 x C where:

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$

Number of parameters: 0

[ConvNetJS demo: training on CIFAR-10]

ConvNetJS CIFAR-10 demo

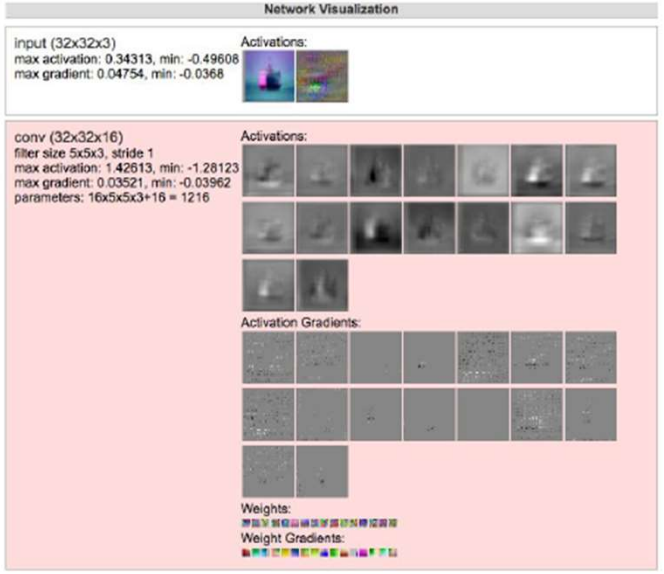
Description

This demo trains a Convolutional Neural Network on the CIFAR-10 dataset in your browser, with nothing but Javascript. The state of the art on this dataset is about 90% accuracy and human performance is at about 94% (not perfect as the dataset can be a bit ambiguous). I used this python script to parse the original files (python version) into batches of images that can be easily loaded into page DOM with img tags.

This dataset is more difficult and it takes longer to train a network. Data augmentation includes random flipping and random image shifts by up to 2px horizontally and verically.

By default, in this demo we're using Adadelta which is one of per-parameter adaptive step size methods, so we don't have to worry about changing learning rates or momentum over time. However, I still included the text fields for changing these if you'd like to play around with SGD+Momentum trainer.

Report questions/bugs/suggestions to @karpathy.



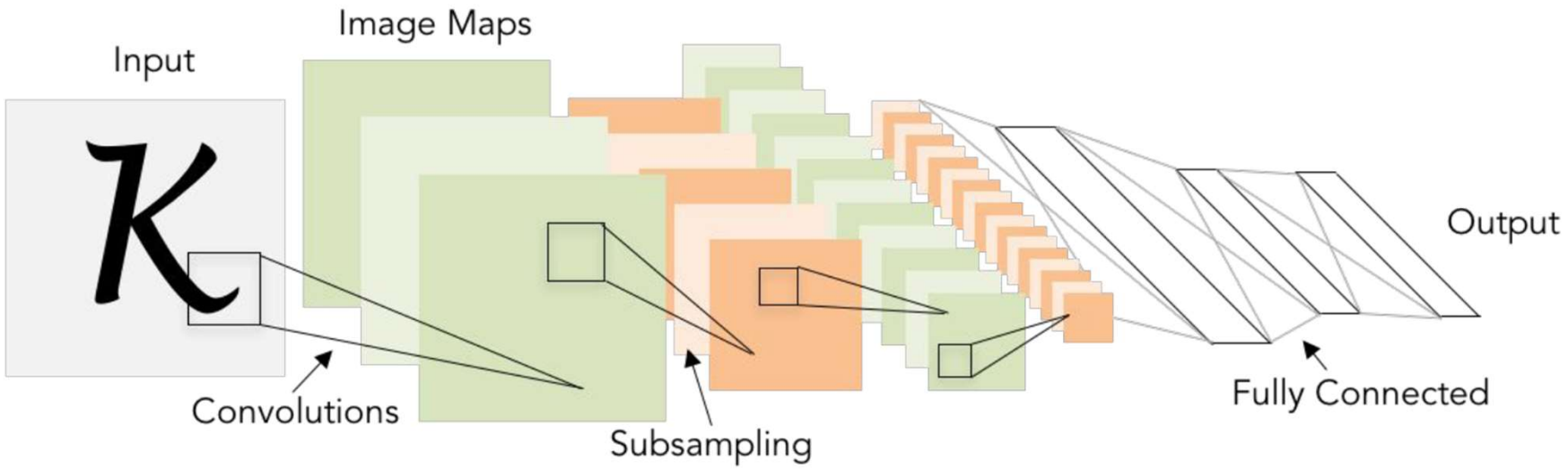
http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

Summary

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Historically architectures looked like
  [(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX
  where N is usually up to ~5, M is large, 0 <= K <= 2.
- But recent advances such as ResNet/GoogLeNet have challenged this paradigm

Review: LeNet-5

[LeCun et al., 1998]



Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

Summary: CNN Architectures

Case Studies

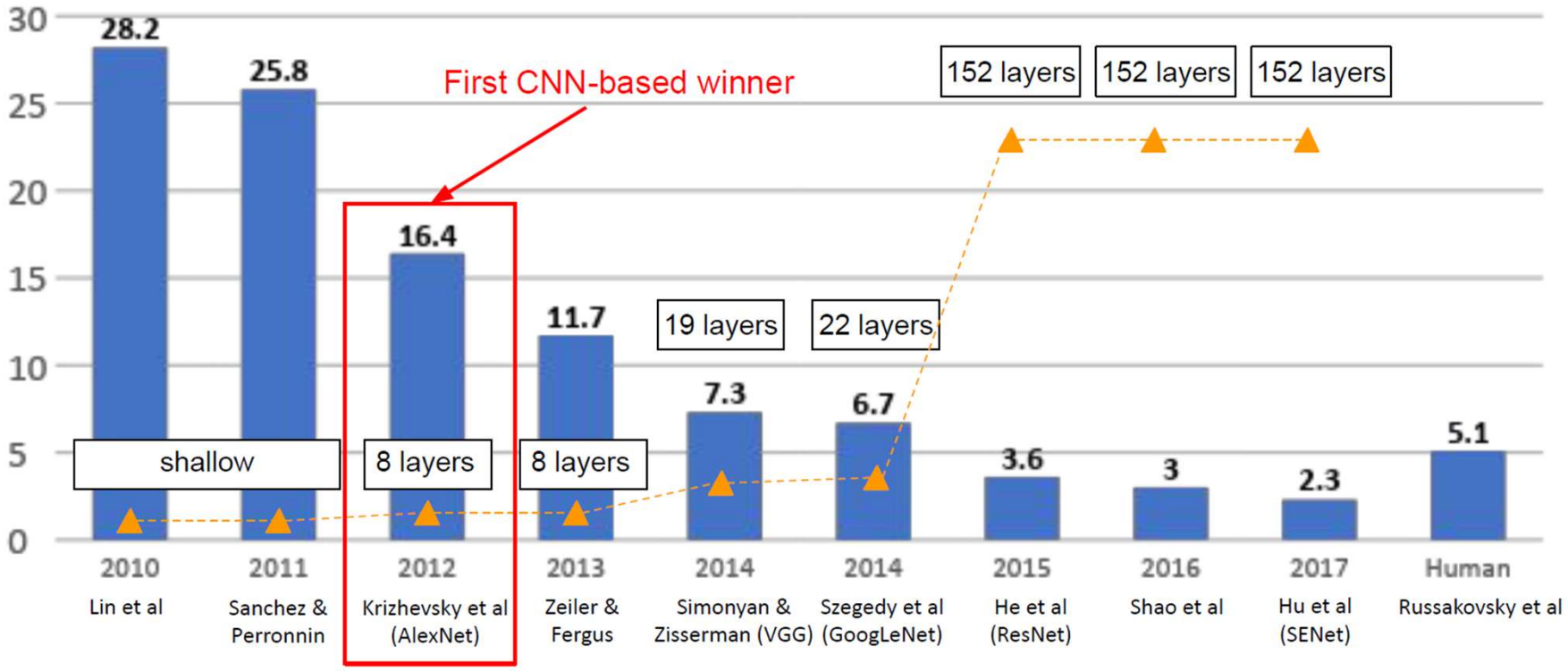
- AlexNet
- VGG
- GoogLeNet
- ResNet

Also....

- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet

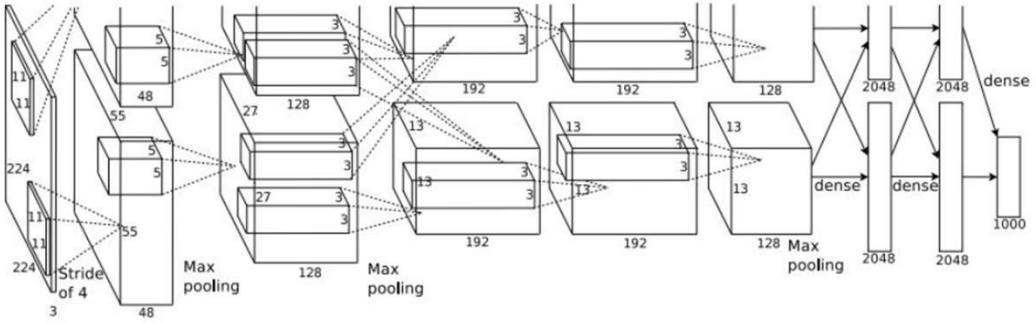
Completion of the challenge:
Annual ImageNet competition no longer held after 2017 -> now moved to Kaggle.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: AlexNet

[Krizhevsky et al. 2012]

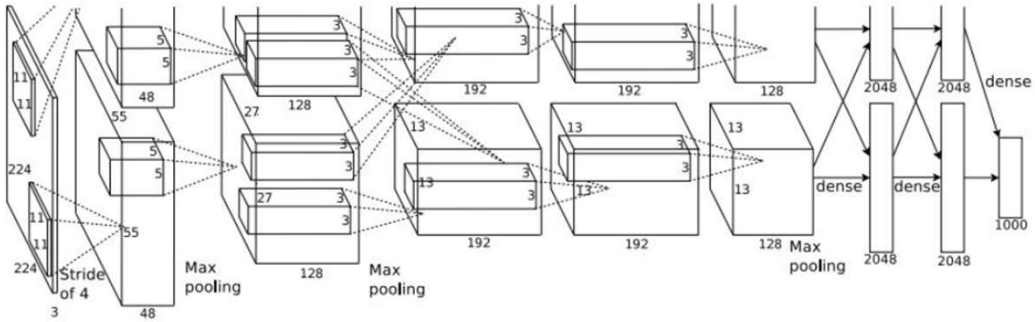


Architecture:

CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

Case Study: AlexNet

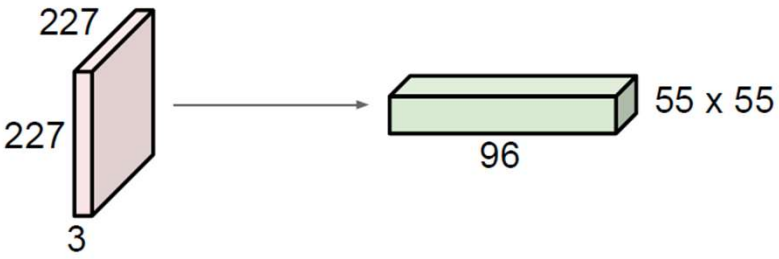
[Krizhevsky et al. 2012]



Input: 227x227x3 images

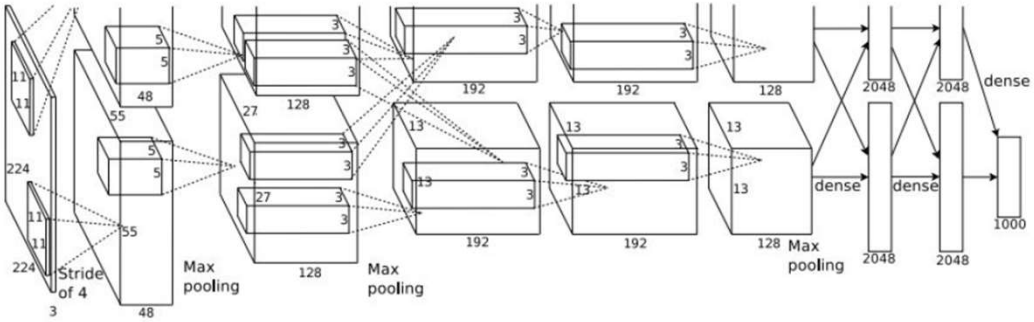
First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**

$W' = (W - F + 2P) / S + 1$



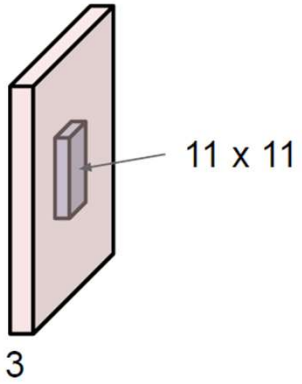
Case Study: AlexNet

[Krizhevsky et al. 2012]



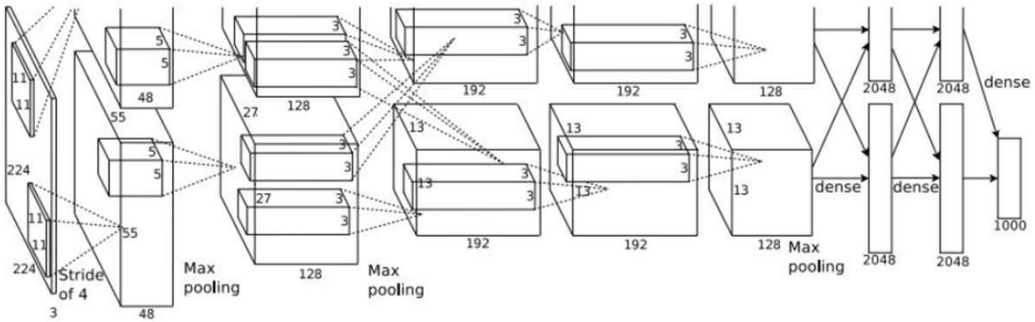
Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**
Parameters: (11*11*3 + 1)*96 = **35K**



Case Study: AlexNet

[Krizhevsky et al. 2012]

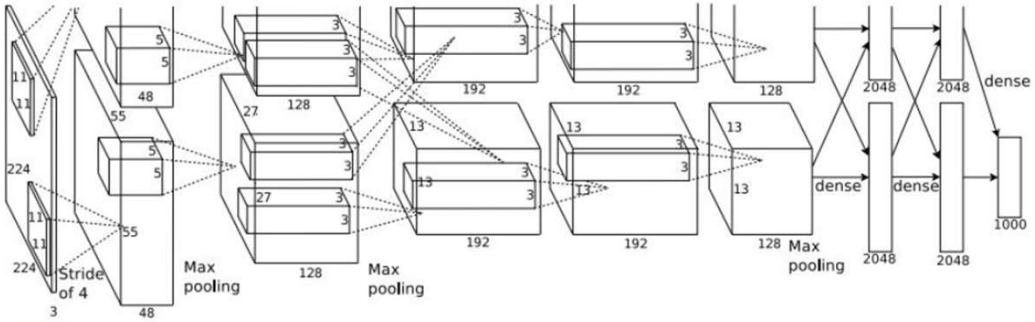


Input: 227x227x3 images
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96
Parameters: 0!

Case Study: AlexNet

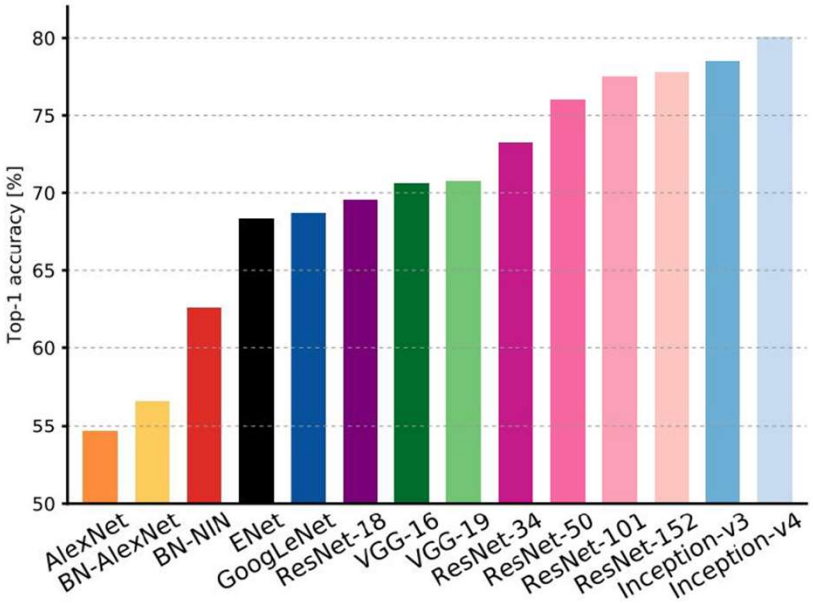
[Krizhevsky et al. 2012]



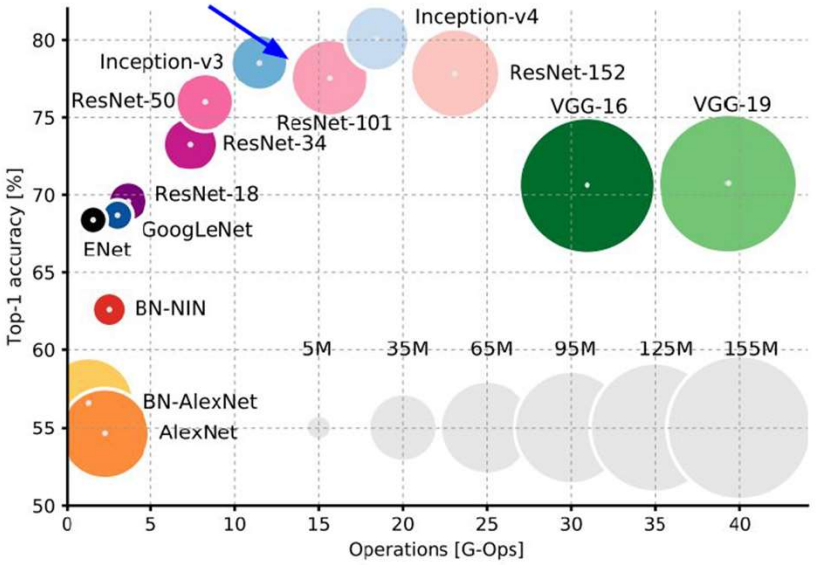
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used LRN layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Comparing complexity...



ResNet:
Moderate efficiency depending on model, highest accuracy



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Main takeaways

AlexNet showed that you can use CNNs to train Computer Vision models.
ZFNet, **VGG** shows that bigger networks work better
GoogLeNet is one of the first to focus on efficiency using 1x1 bottleneck convolutions and global avg pool instead of FC layers
ResNet showed us how to train extremely deep networks

- Limited only by GPU & memory!
- Showed diminishing returns as networks got bigger

After ResNet: CNNs were better than the human metric and focus shifted to Efficient networks:

- Lots of tiny networks aimed at mobile devices: **MobileNet**, **ShuffleNet**

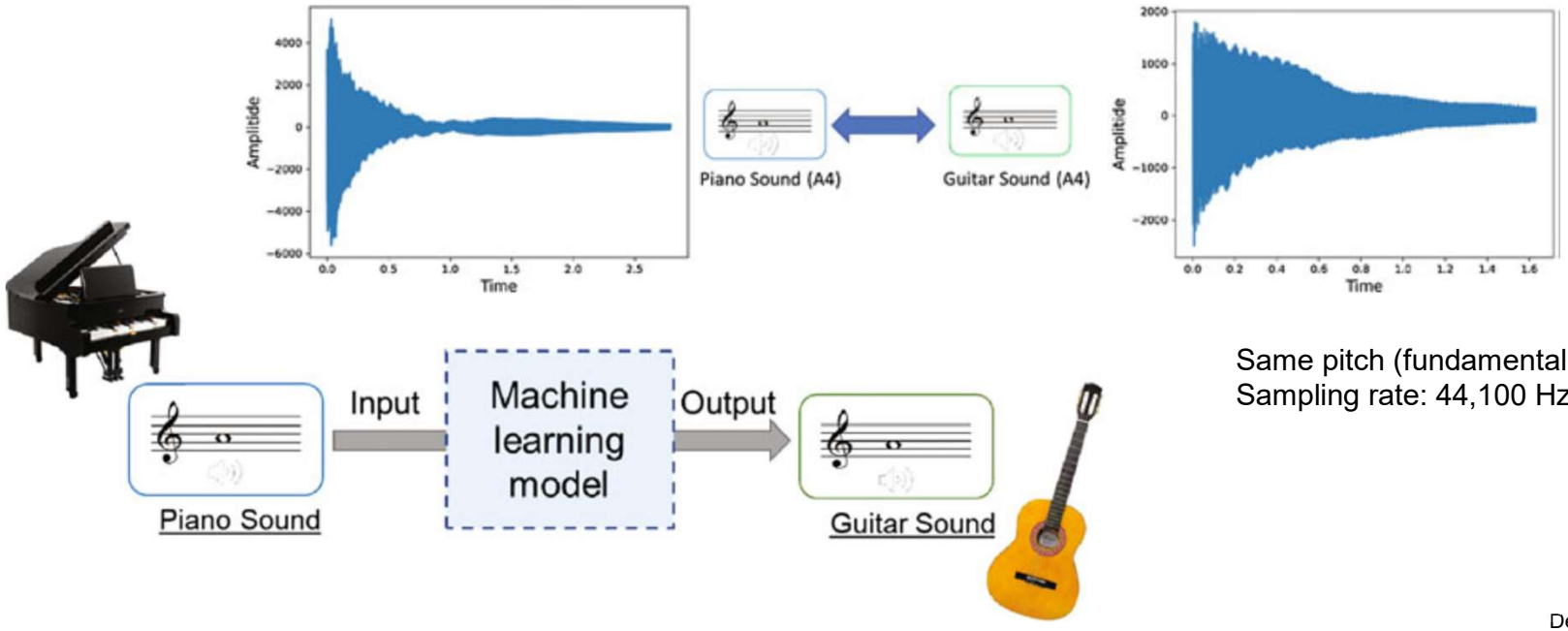
Neural Architecture Search can now automate architecture design

Summary: CNN Architectures

- Many popular architectures are available in model zoos.
- ResNets are currently good defaults to use.
- Networks have gotten increasingly deep over time.
- Many other aspects of network architectures are also continuously being investigated and improved.

Musical Instrument Sound Conversion

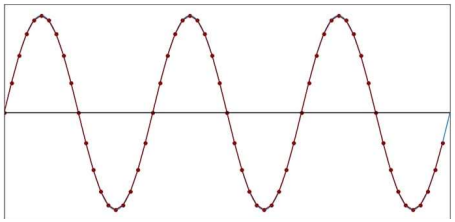
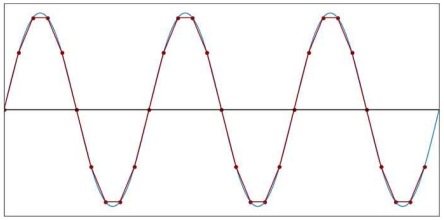
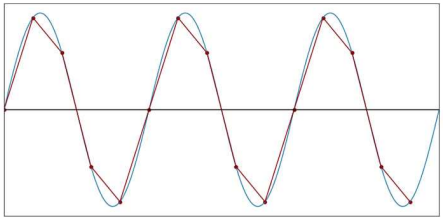
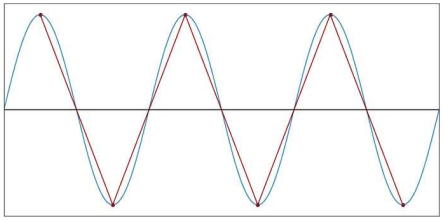
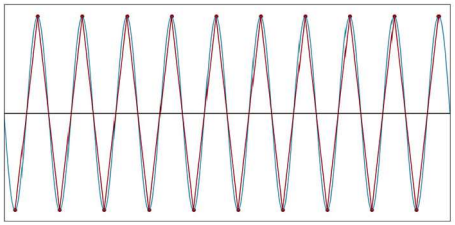
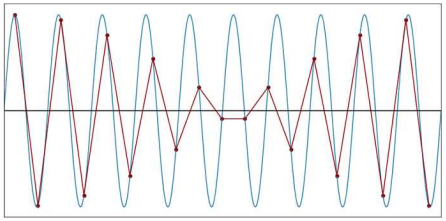
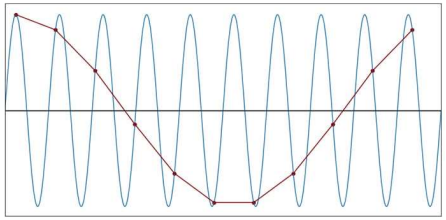
- use mechanistic data science to convert piano sounds to guitar sounds
- Dataset: eight pairs of notes (notes A4, A5, B5, C5, C6, D5, E5, G5) performed by a piano and by a guitar (apronus.com)



Same pitch (fundamental frequency)
Sampling rate: 44,100 Hz

Nyquist-Shannon Theorem

- If a system uniformly samples an analog signal at a rate that exceeds the signal's highest frequency by at least a factor of two, the original analog signal can be perfectly recovered from the discrete values produced by sampling.
- The human hearing range is 20~20,000 Hz. Therefore, a sufficient sampling rate for sound file should be 40,000 Hz
  - A common sampling time interval in music is 44,100Hz. It is higher than 40,000Hz

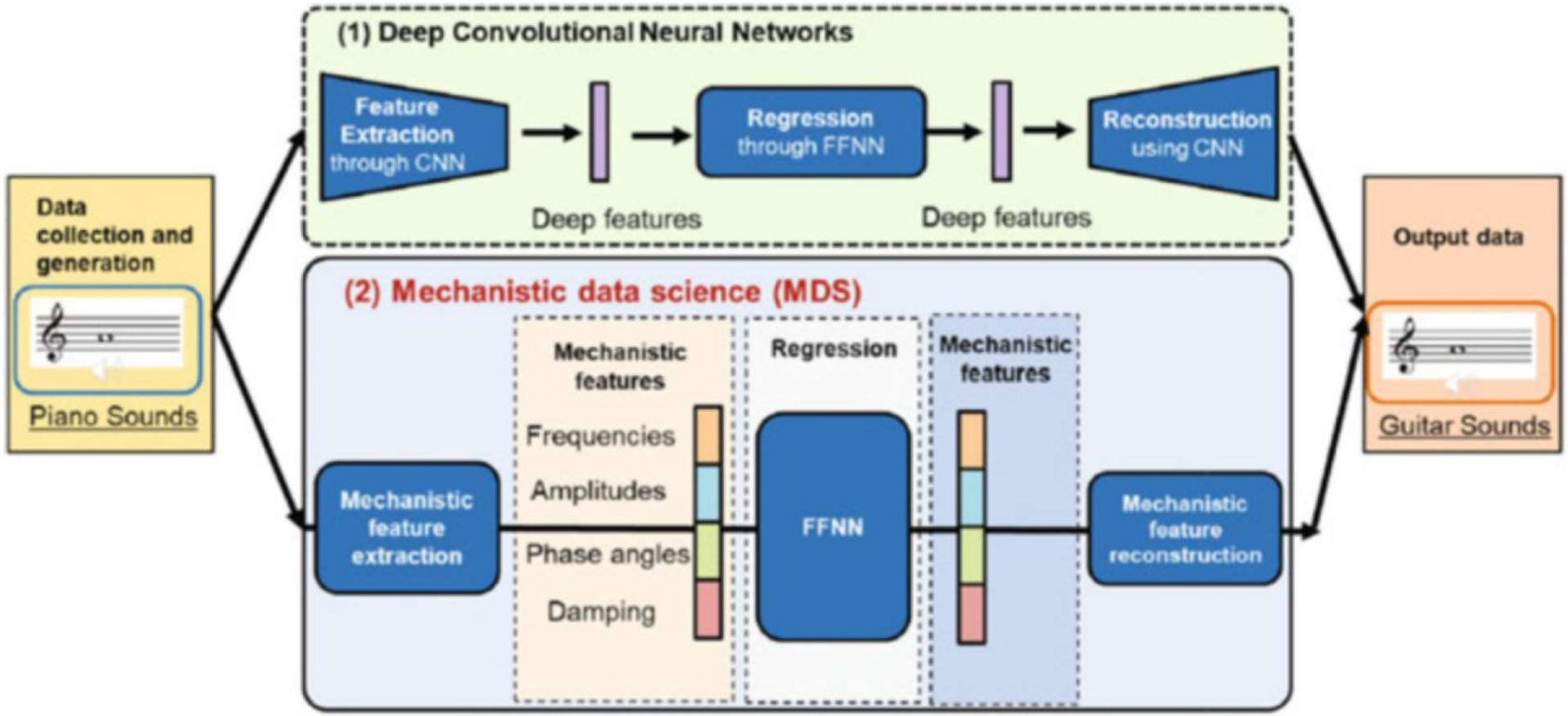








$$f_{\text{sampling}} \geq 2 f_{\max} \xrightarrow[f=\frac{1}{T}]{\omega=2\pi f}$$

$$\frac{1}{t_{\text{sampling}}} \geq 2\left(\frac{\omega_{\max}}{2\pi}\right) \rightarrow$$

$$t_{\text{sampling}} \leq \frac{\pi}{\omega_{\max}}$$

Three Solutions



(3) PCA + NN → Ch.7

Pure CNN Analysis

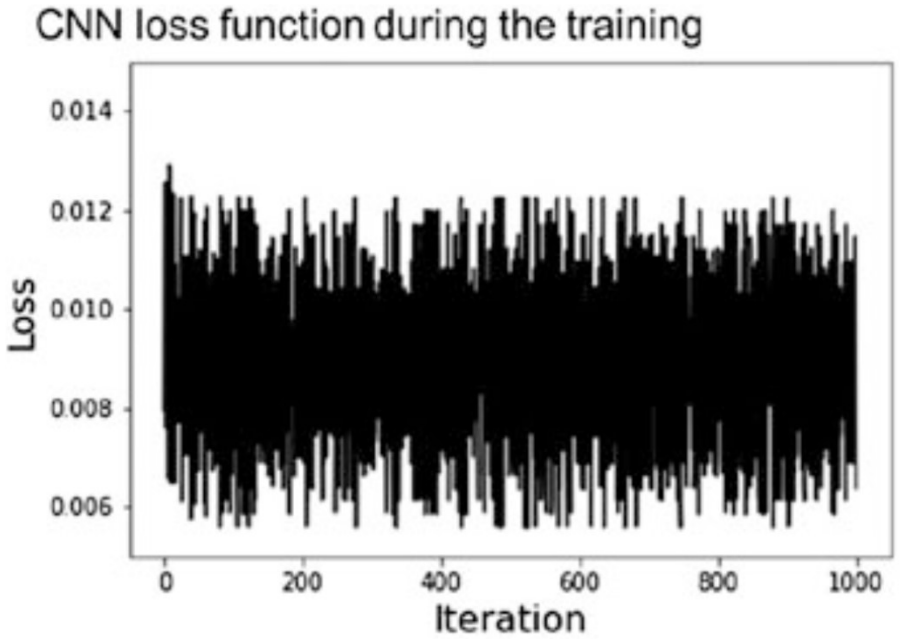
- Input and output are high-dimensional time-amplitude curves
- drawback of this strategy is the high number of trainable parameters involved in the CNN and FFNN structures
- For some applications where the amount of data is small or the quality of data is low, the performance of the CNN is limited
- high-dimensional sound curve can be represented by a set of mechanistic features with a physical meaning
- hyperparameters involved in the model can be significantly decreased from thousands to dozens in this manner

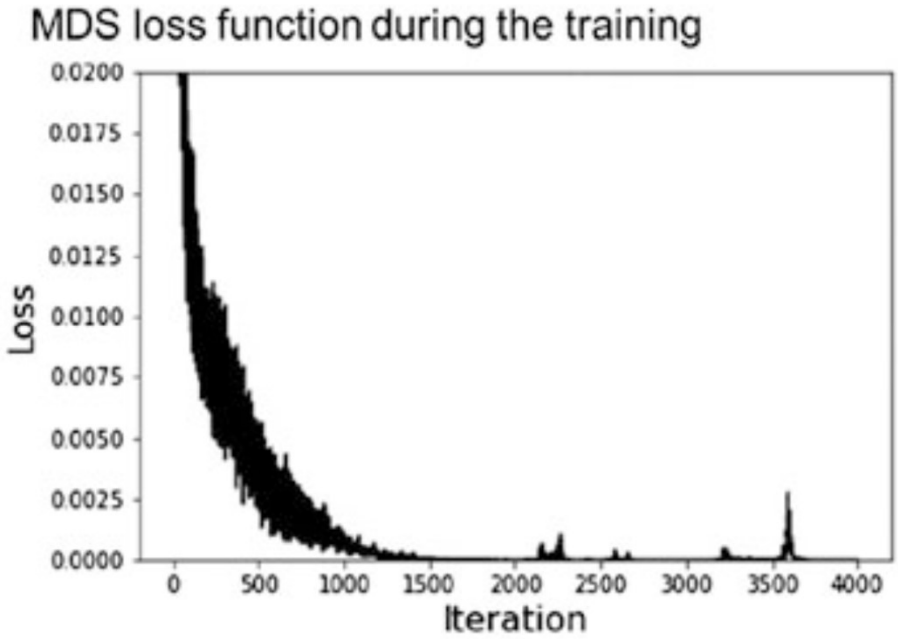
Mechanistic Data Science Analysis

- Instead of extracting deep features by CNN, mechanistic data science extracts mechanistic features based on the underlying scientific principles
- Each signal (high-dimensional sound curve) can be simplified to a set of sine functions, where the mechanistic features are the frequencies, damping coefficients, amplitudes, and phase angles
- Short Time Fourier Transform (STFT) and a regression to fit a mechanistic model, such as a spring-mass-damper model
- hyperparameters involved in the model can be significantly decreased from thousands to dozens in this manner, which reduces the amount of training data required

CNN vs. MDS

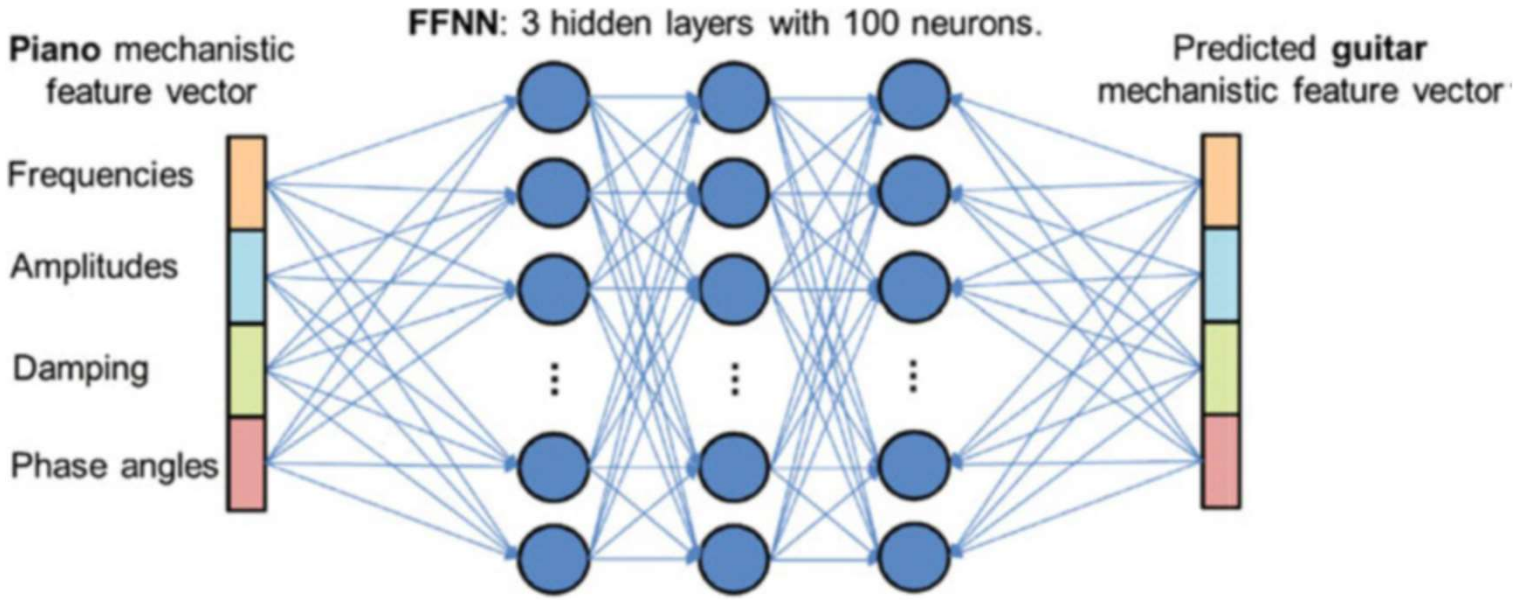
- The loss function of the CNN is volatile and does not converged to zero
- Training a CNN model requires the number of data points to be larger than the dimension of input signal or image
  - In this case, the dimension of input is a million



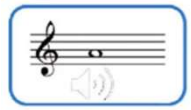


FFNN structure with reduced mechanistic features

- Dataset: 4(features) x 8(sets) = 32 for one note
- activation function: tanh
- Loss function: MSE

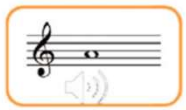


Results of reconstructing a single guitar key A4 from a piano key as input

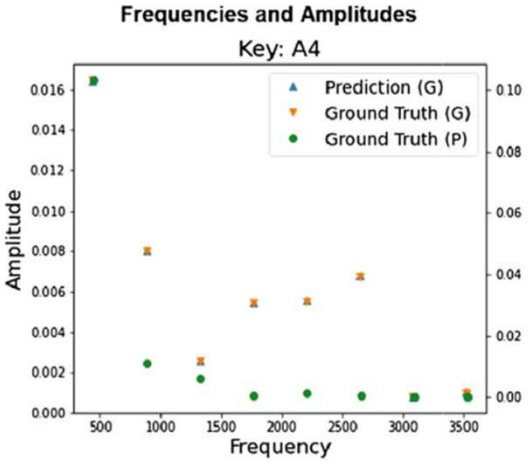


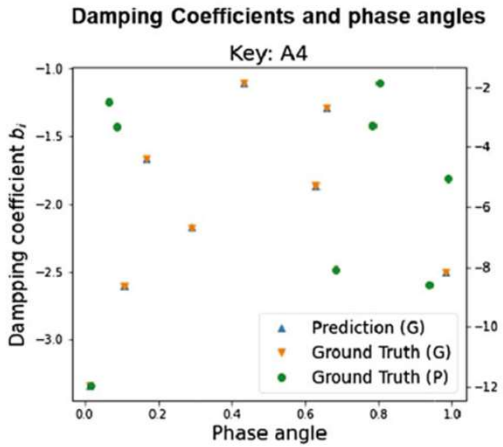
Authentic
Piano Sound (A4)

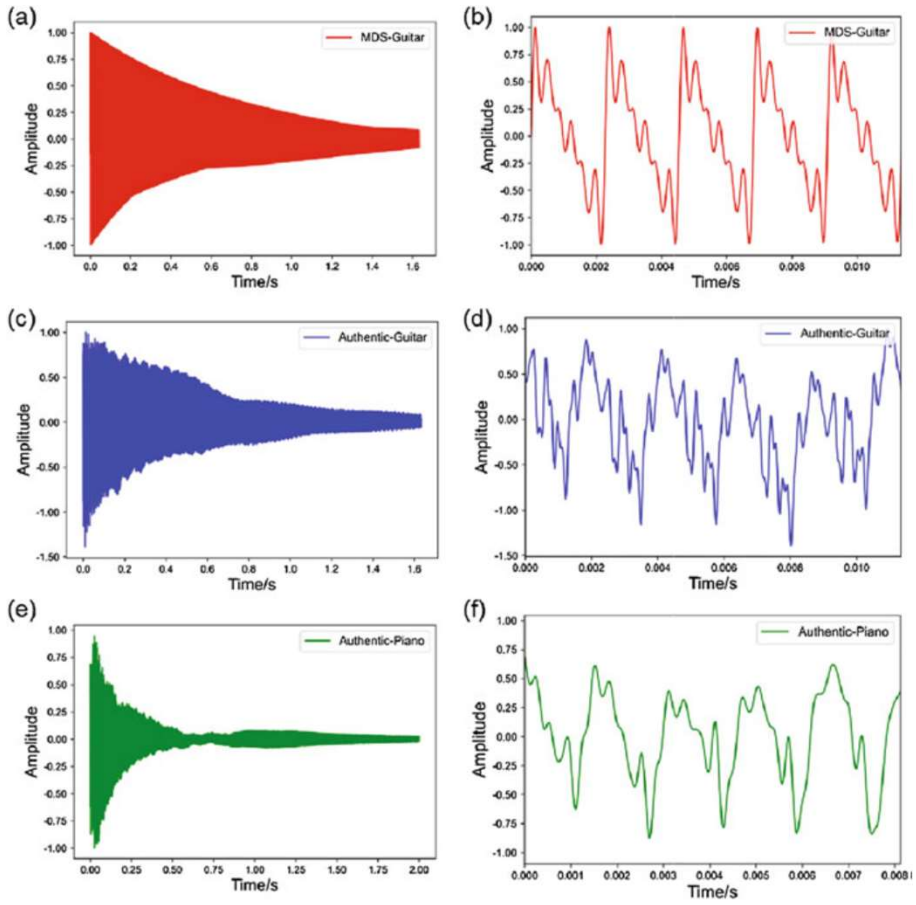
Authentic
Guitar Sound (A4)



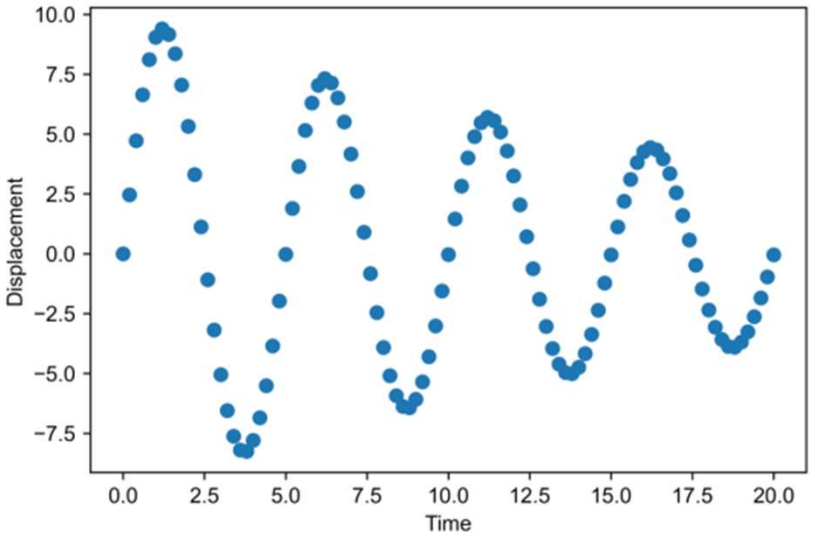
MDS Generated
Guitar Sound (A4)







Use PyTorch to fit spring-mass data (1)



Train a FFNN

Use PyTorch to fit spring-mass data (2)

```
%matplotlib inline
import numpy as np
import torch
import torch.optim as optim
import torch.nn as nn
import matplotlib.pyplot as plt
import math

torch.set_printoptions(edgeitems=2, linewidth=75)
```

Import necessary libraries

```
t=np.linspace(0, 20, 101)
z =10*np.sin(2*3.14/5*t)*np.exp(-5e-2*t)

z = torch.FloatTensor(z).unsqueeze(1) # <1>

t = torch.FloatTensor(t).unsqueeze(1) # <1>
```

Create data

```
from matplotlib import pyplot as plt

t_range = torch.arange(0., 21).unsqueeze(1)

fig = plt.figure(dpi=600)
plt.xlabel("Time")
plt.ylabel("Displacement")
plt.plot(t.numpy(), z.numpy(), 'o')
```

Plot data

Use PyTorch to fit spring-mass data (2)

```
n_samples = t.shape[0]
n_val = int(0.2 * n_samples)
print(n_val)
shuffled_indices = torch.randperm(n_samples)

train_indices = shuffled_indices[0:n_samples-n_val]
val_indices = shuffled_indices[n_samples-n_val:n_samples]

print(train_indices, val_indices)
print(shuffled_indices)
```

Separate data into 80% training data and 20% test data

```
t_train = t[train_indices]
z_train = z[train_indices]

t_val = t[val_indices]
z_val = z[val_indices]

t_n_train = 0.1 * t_train
t_n_val = 0.1 * t_val
```

Use PyTorch to fit spring-mass data (3)

```
def training_loop(n_epochs, optimizer, model, loss_fn, t_train, t_val,
                  z_train, z_val):
    for epoch in range(1, n_epochs + 1):
        z_p_train = model(t_train) # <1>
        loss_train = loss_fn(z_p_train, z_train)

        z_p_val = model(t_val) # <1>
        loss_val = loss_fn(z_p_val, z_val)

        optimizer.zero_grad()
        loss_train.backward() # <2>
        optimizer.step()

        if epoch == 1 or epoch % 1000 == 0:
            print(f"Epoch {epoch}, Training loss {loss_train.item():.4f},"
                  f" Validation loss {loss_val.item():.4f}")
```

Define training process

```
def loss_fn(z_p, z):
    squared_diffs = (z_p - z)**2
    return squared_diffs.mean()
```

Define loss function

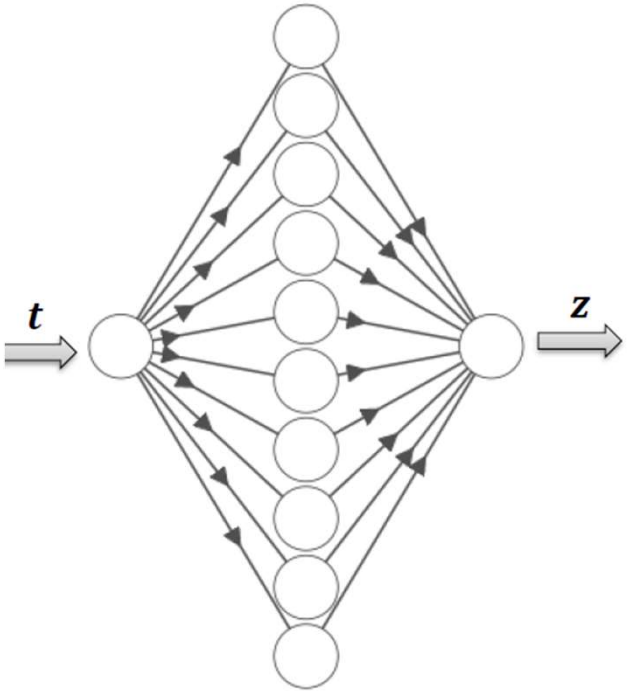
Use PyTorch to fit spring-mass data (4)

```
from collections import OrderedDict
neuron_count = 100
seq_model = nn.Sequential(OrderedDict([
    ('hidden_linear', nn.Linear(1, neuron_count )),
    ('hidden_activation', nn.Tanh()),
    ('output_linear', nn.Linear(neuron_count , 1))
]))

seq_model
```

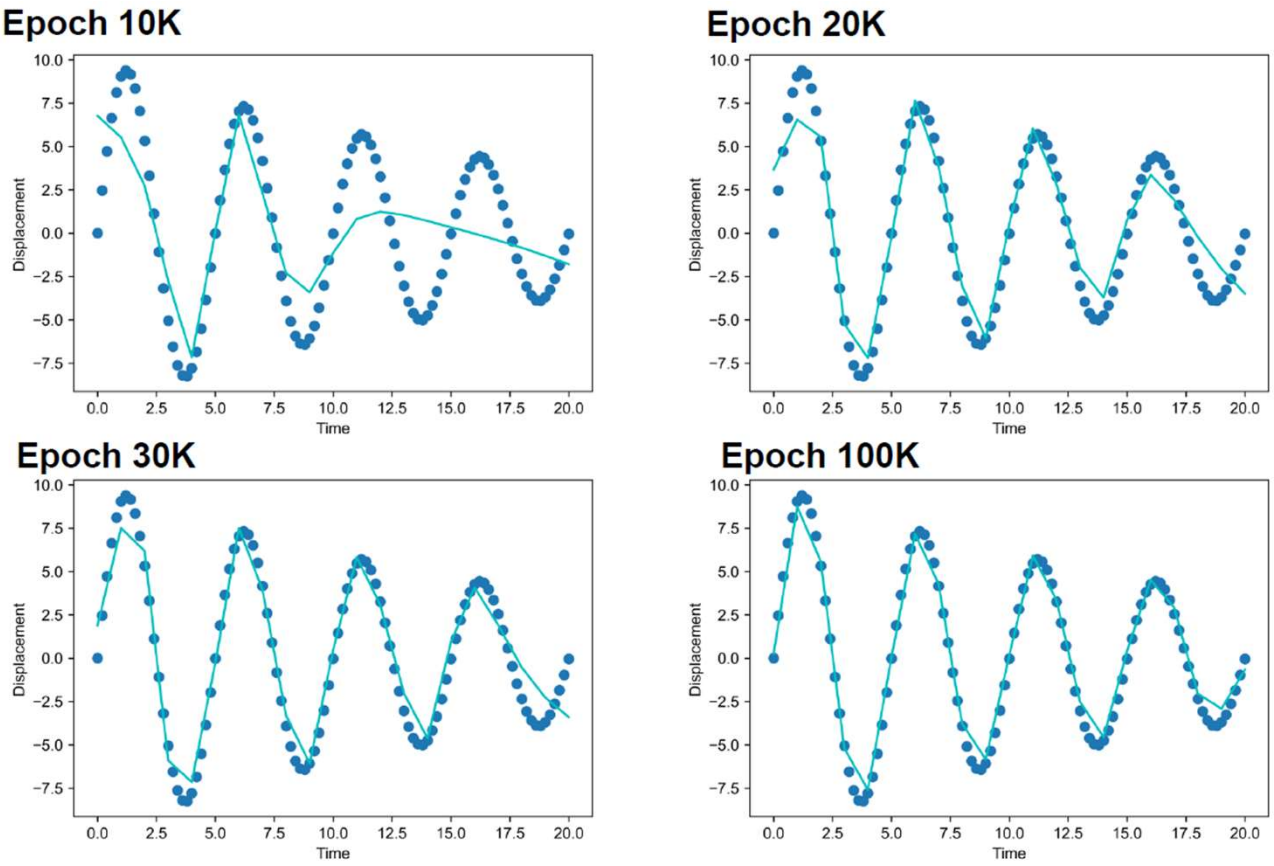
```
optimizer = optim.SGD(seq_model.parameters(), lr=1e-2)

training_loop(
    n_epochs = 10000,
    optimizer = optimizer,
    model = seq_model,
    loss_fn = nn.MSELoss(),
    t_train = t_n_train,
    t_val = t_n_val,
    z_train = z_train,
    z_val = z_val)

print('output', seq_model(t_n_val))
print('answer', z_val)
```

Define network structure

1 layer with 100 neurons

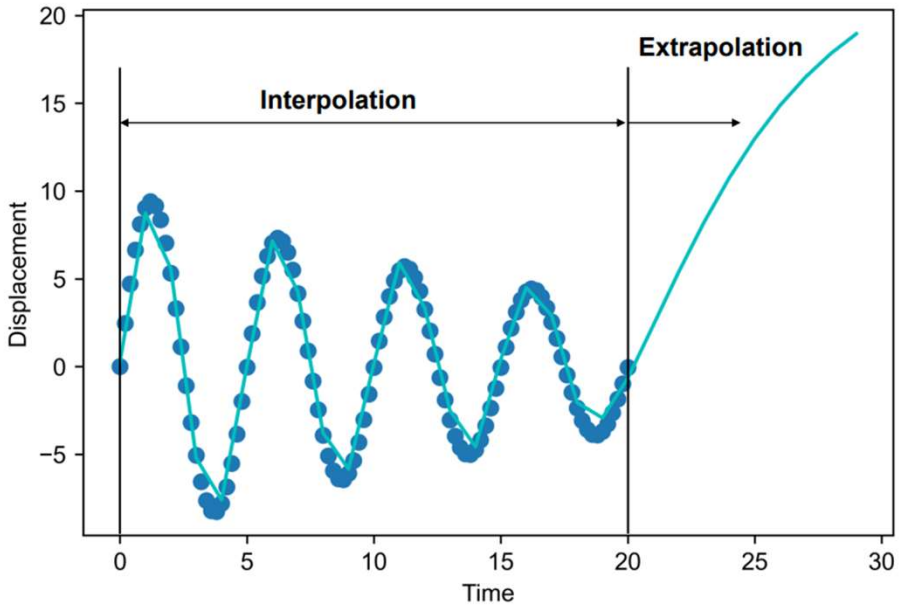


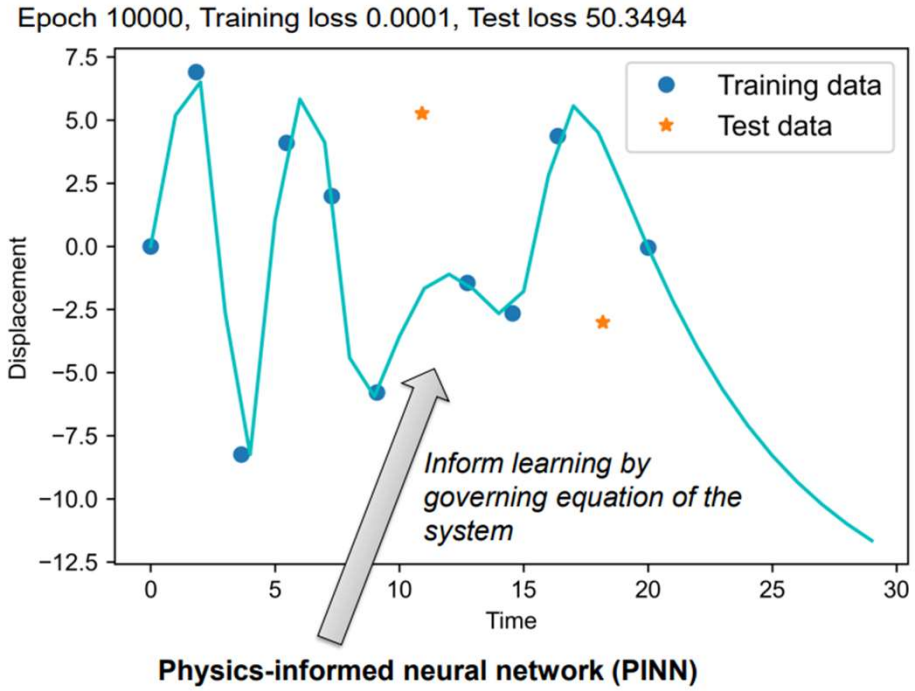
Use PyTorch to fit spring-mass data (3)



Limitations of NN

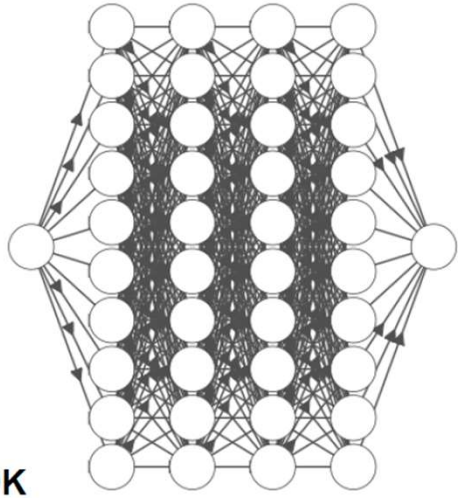
- Good at interpolation, not extrapolation
- May be overfit if there are not enough data





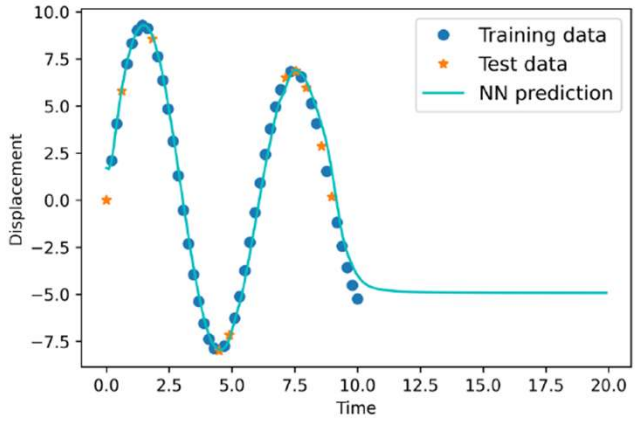
Physics-informed neural network (PINN)

NN:



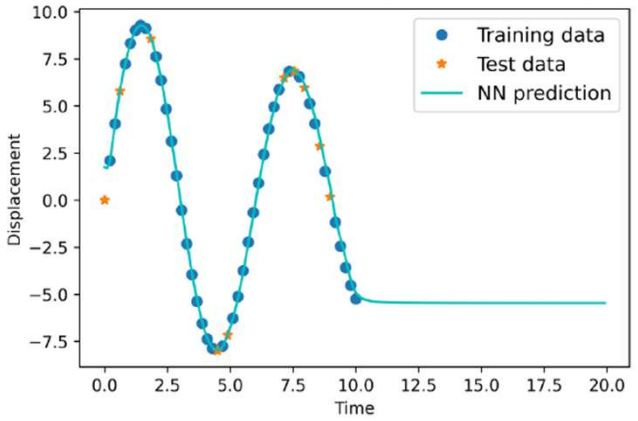
4 hidden layer
(100 neurons per layer)

Epoch 10K



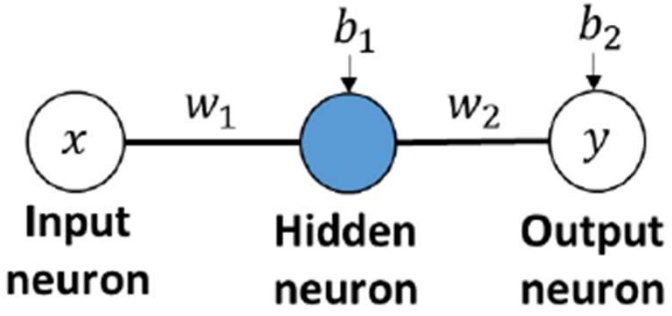
Epoch 20K

Training loss MSE 0.0365
Validation loss MSE 0.5312



Automatic Differentiation

- A trained NN model is differentiable



$$y = \mathcal{A}(w_2 \mathcal{A}(w_1 x + b_1) + b_2)$$

$$A(\ ) = \tanh(\ )$$

$$y = \tanh\left(w_2 \tanh\left(w_1 x + b_1\right) + b_2\right)$$

$$\frac{d}{dx}\left(\tanh(x)\right) = 1 - \tanh^2(x)$$

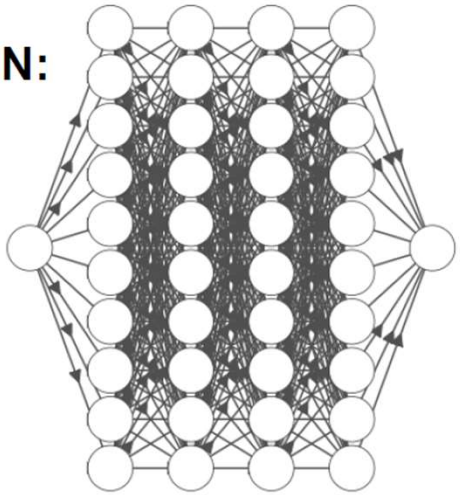
$$\frac{dy}{dx} = \left[1 - \tanh^2\left(w_2 \tanh\left(w_1 x + b_1\right) + b_2\right)\right] \frac{d}{dx}\left[w_2 \tanh\left(w_1 x + b_1\right) + b_2\right]$$

$$= \left[1 - \tanh^2\left(w_2 \tanh\left(w_1 x + b_1\right) + b_2\right)\right] w_2 \left[1 - \tanh^2\left(w_1 x + b_1\right)\right] w_1$$

- PyTorch for automatic differentiation

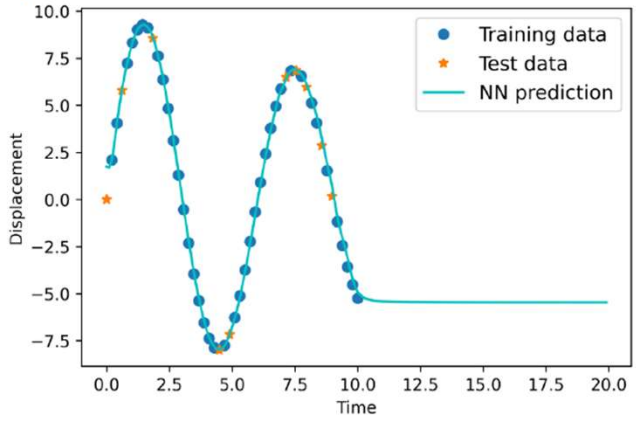
```
torch.autograd.grad(outputs, inputs, grad_outputs=None, retain_graph=None,
create_graph=False, only_inputs=True, allow_unused=False) [SOURCE]
```

NN:

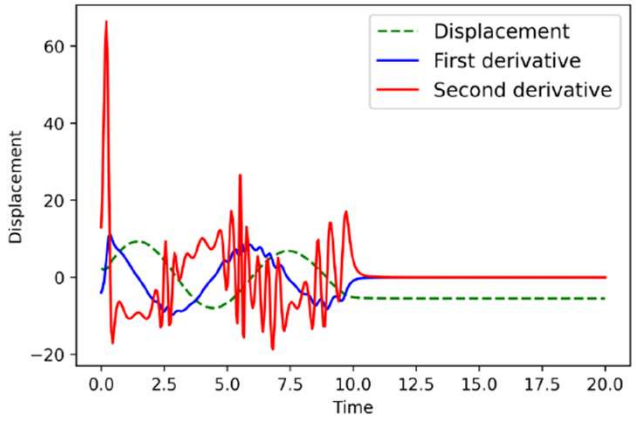


4 hidden layer
(100 neurons per layer)

Epoch 20K



Derivatives are inaccurate



Customized loss function by adding governing equation

data: $D = \left\{ \left(t_i, z_i \right) \middle| i = 1, 2, \ldots, n \right\}$

[Neural Network]

Loss function $= \frac{1}{n} \sum_{i=1}^{n} \left(z_{pi} - z_i \right)^2$: Mean Squared Error (MSE)

```
def loss_fn(z_p, z):
    squared_diffs = (z_p - z)**2
    return squared_diffs.mean()
```

[Physics-Informed Neural Network]

governing equation of spring-mass system: $\ddot{z} + \frac{c}{m}\dot{z} + \frac{k}{m} z = 0 \rightarrow \ddot{z} + 0.1\dot{z} + 1.1z = 0$

Loss function $= \underbrace{\frac{1}{n} \sum_{i=1}^{n} \left(z_{pi} - z_i \right)^2}_{\text{MSE}} + \underbrace{\frac{1}{n_1} \sum_{i=1}^{n_1} \left(\ddot{z}_{pi} + 0.1\dot{z}_{pi} + 1.1 z_{pi} \right)^2}_{\text{equation error}}$

n : number of data points

n_1 : number of sampling points for derivative calculation

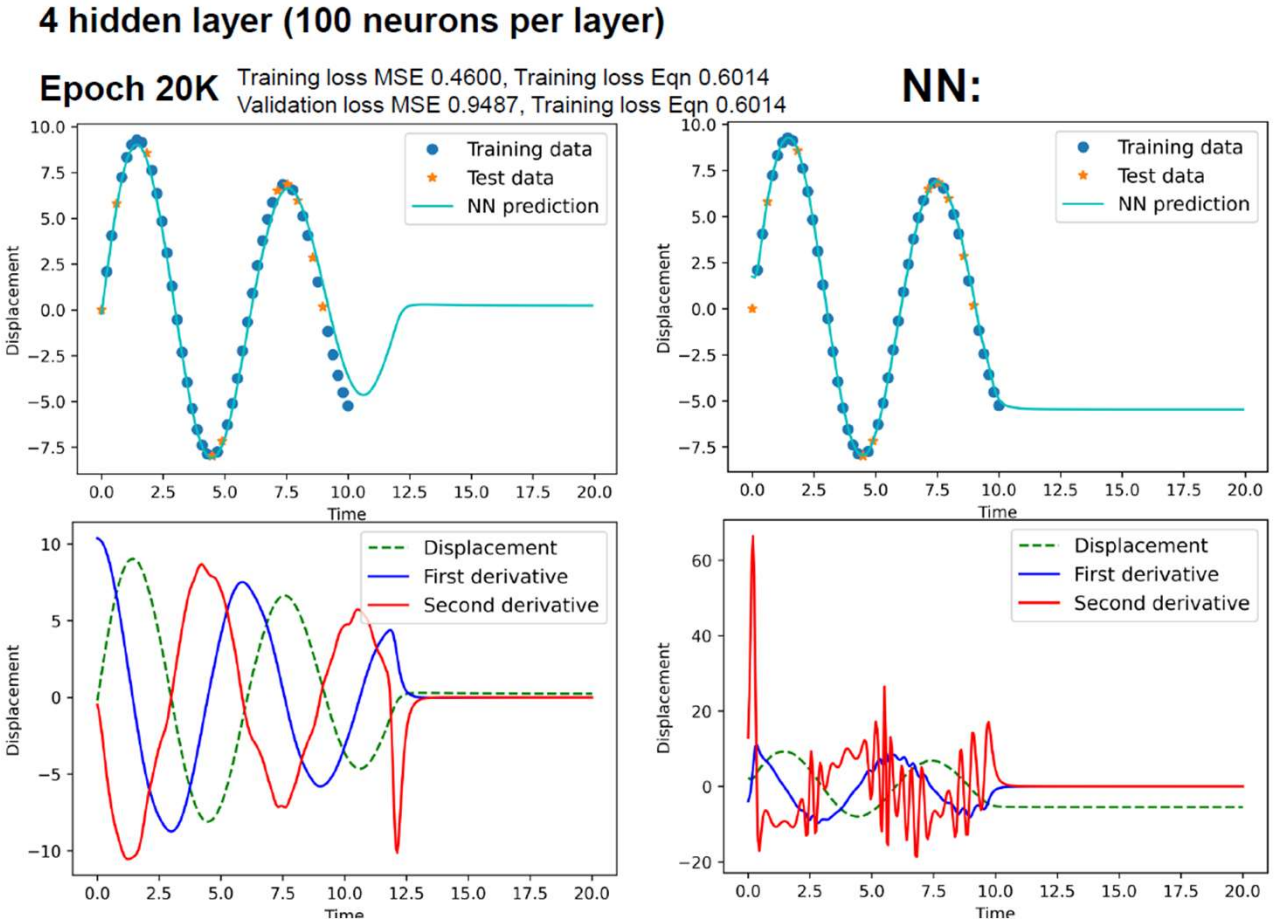
Loss Function for PINN: Code

```python
def loss_fn1(z_p, z):
    squared_diffs = (z_p - z)**2
    return squared_diffs.mean()

def loss_fn2(t_new):

    dzdt = torch.autograd.grad(outputs=seq_model(t_new),
    inputs=t_new,
    grad_outputs=torch.ones_like(seq_model(t_new)),
    retain_graph=True,
    create_graph=True)[0]
    dzdtt= torch.autograd.grad(outputs=dzdt,
    inputs=t_new,
    grad_outputs=torch.ones_like(dzdt),
    retain_graph=True,
    create_graph=True)[0]
    z=seq_model(t_new)
    f=abs(dzdtt+1.1*z+0.1*dzdt)

    #print(f.mean())
    return f.mean()
```

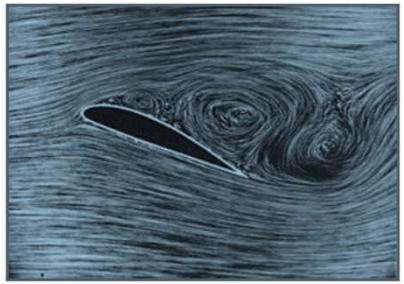
```python
t=np.linspace(0, 10, 50)
z =10*np.sin(2*3.14/6*t)*np.exp(-5e-2*t)
```

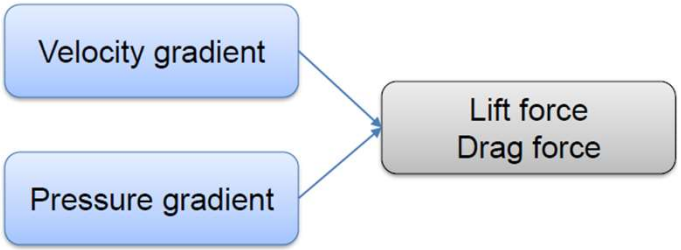
```python
t_new=np.linspace(0, 20, 1000)
```

PINN vs. NN

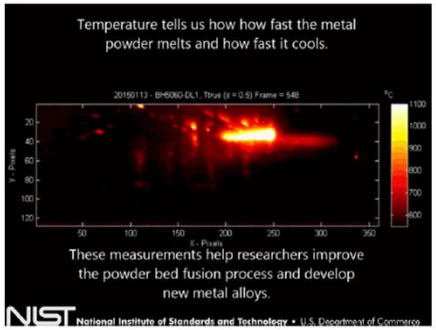

4 hidden layer (100 neurons per layer)
Epoch 20K
Training loss MSE 0.4600, Training loss Eqn 0.6014
Validation loss MSE 0.9487, Training loss Eqn 0.6014
NN:
Training data
Test data
NN prediction
Displacement
First derivative
Second derivative
Time

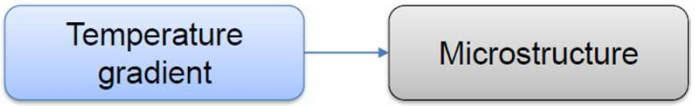
- Derivative/Gradient accuracy is critical in many engineering applications
- Data-driven airfoil design



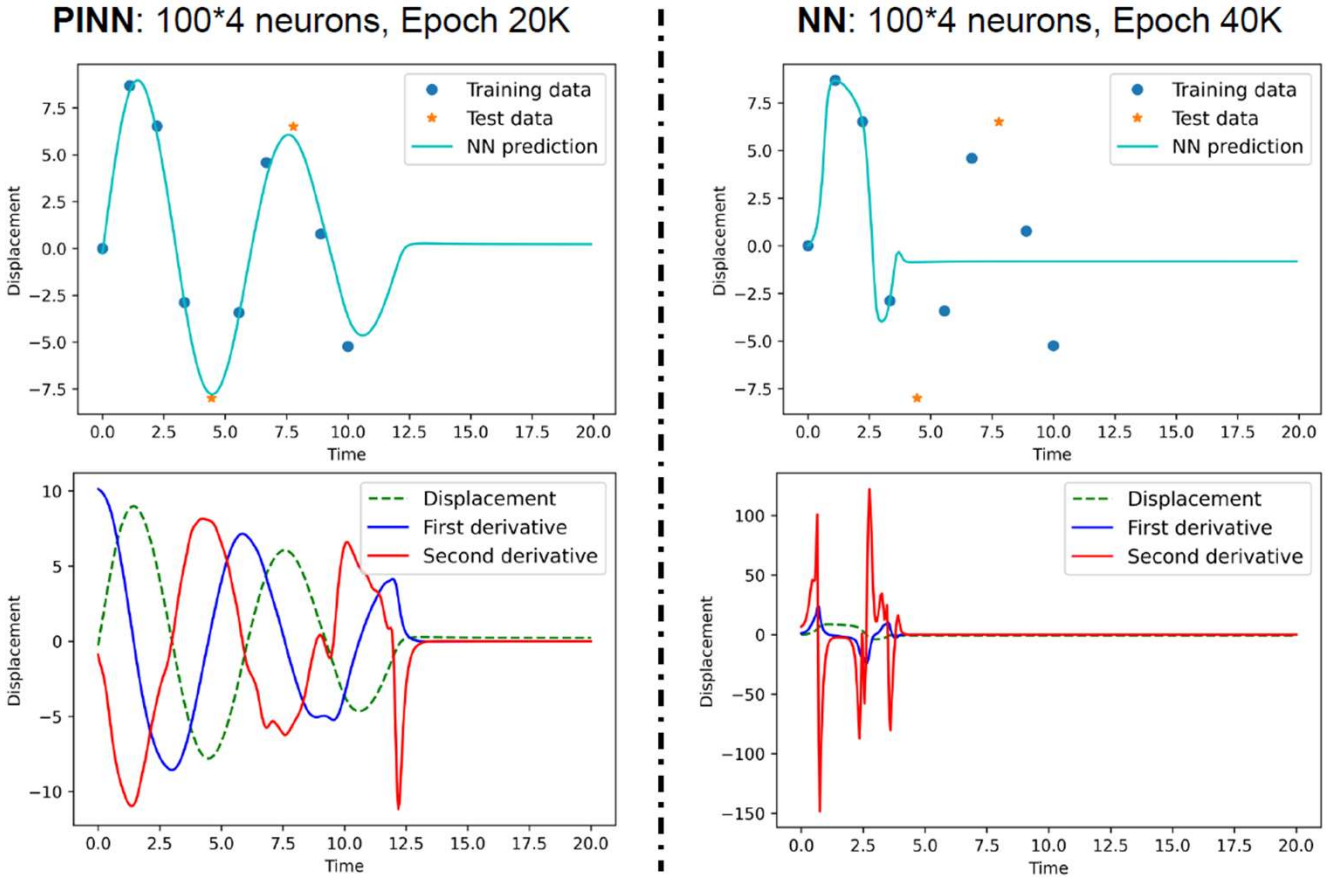


- Pressure-structure linkage in additive manufacturing

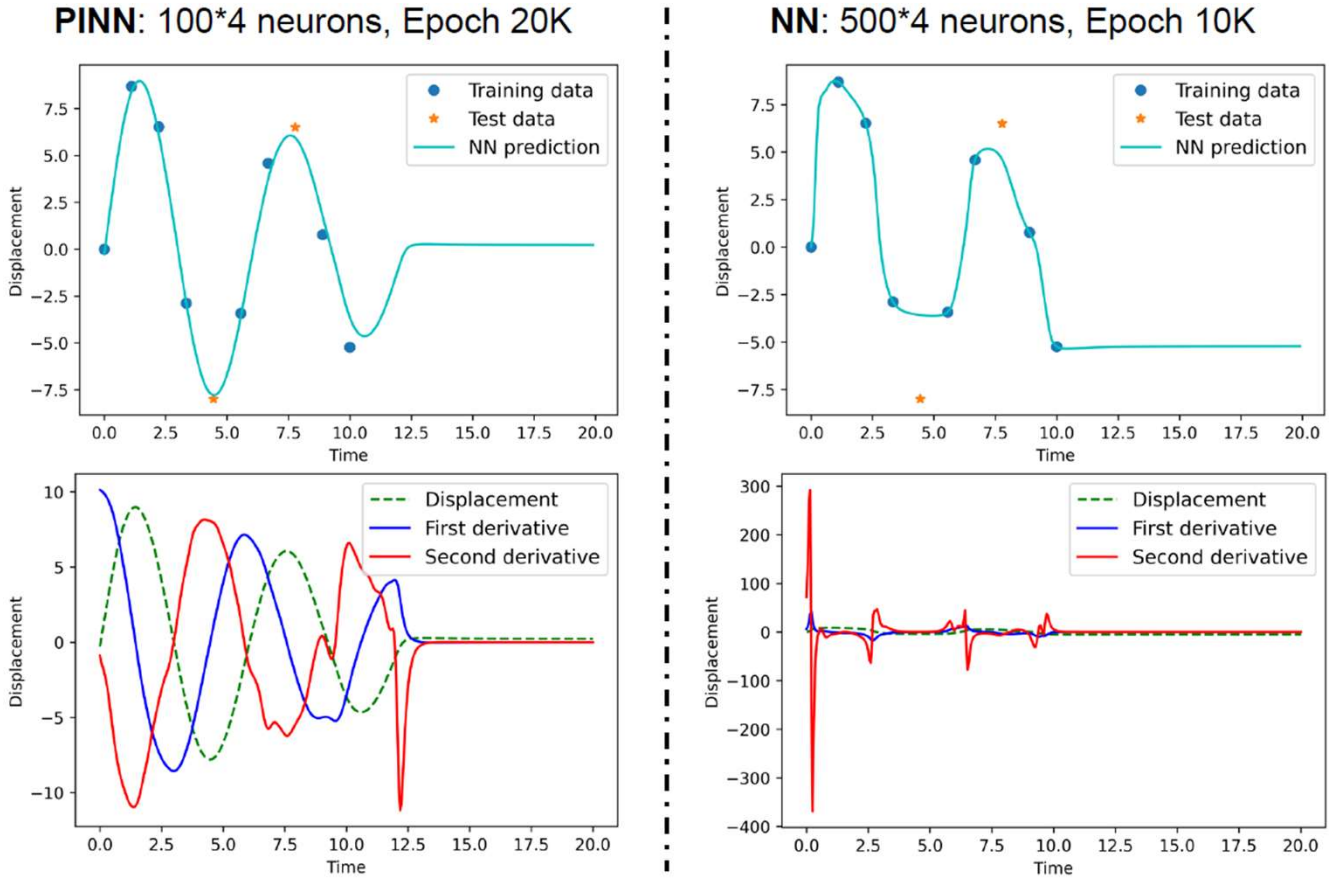




Small Dataset: PINN vs. NN (1)



Small Dataset: PINN vs. NN (2)



Summary

- Adding prior governing equation into NN loss function
  - Can improve prediction accuracy, especially for derivatives
  - Can avoid (to some extent) overfit
  - Works well for small data sets
  - Might improve extrapolation capability
- Physics informed neural networks (PINN)
  - M. Raissi, P. Perdikaris and G.E. Karniadakis . "Physics informed deep learning (Part I): Data driven solutions of nonlinear partial differential equations." arXiv preprint arXiv:1711.10561 (2017)
- Physics guided neural networks (PGNN)
  - A. Karpatne, et al. "Physics guided neural networks (PGNN): An application in lake temperature modeling." arXiv preprint arXiv:1710.11431 2 (2017)
- Physics Aware Neural Networks (PANN)
  - A.S. Zamzam and N.D. Sidiropoulos, IEEE Transactions on Power Systems 35.6 (2020): 4347 4356.
- Mechanistic neural networks (MNN): Embed prior physical knowledge into machine learning
  - S.L Brunton, "Applying Machine Learning to Study Fluid Mechanics.“ arXiv preprint arXiv:2110.02083 (2021)