Deep learning in neuroscience
Tutorial 1

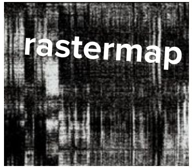
Carsen Stringer, PhD

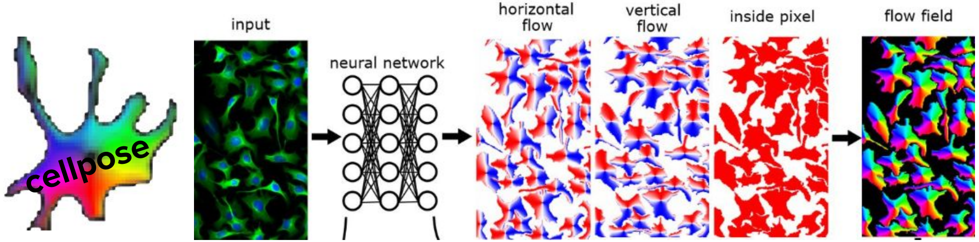
neuromatch academy

About me

HHMI Janelia

How do neurons perform complex, high-dimensional computations?





Thanks to tutorial [redacted]

And thanks to tutorial [redacted]

Roozbeh Farhoodi

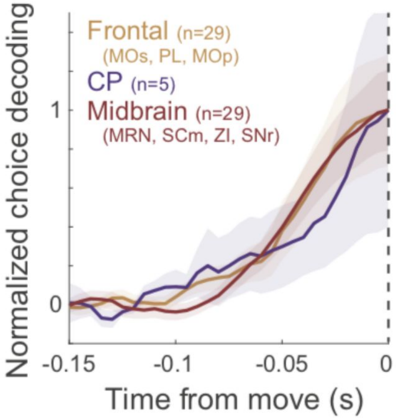
Madineh Sarvestani

Ella Batty

Decoding models vs encoding models

(W1D4 tutorial 2)
neural activity ⇒ external correlate

TUTORIAL 1

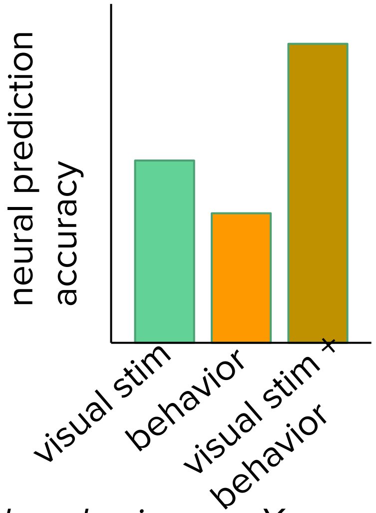


(Steinmetz et al, 2019)

How much information does brain area X have about external correlate Y?

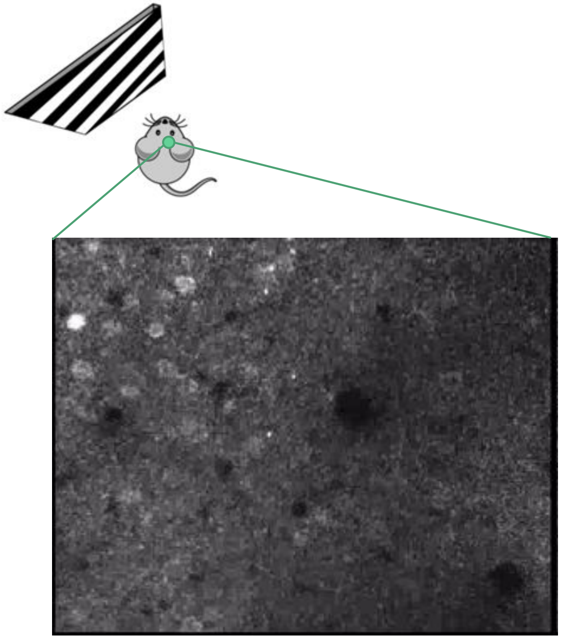
(W1D4 tutorial 1)
external correlate ⇒ neural activity

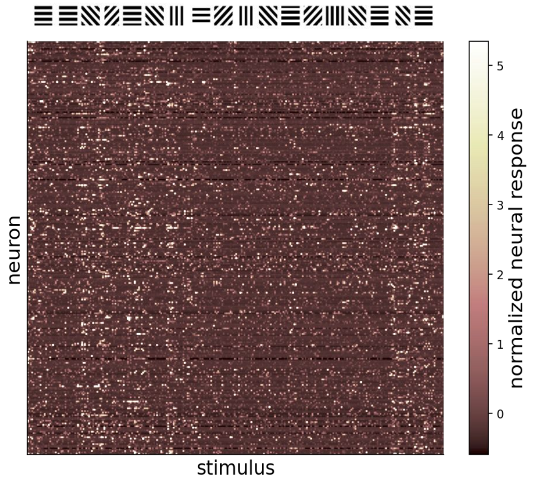
TUTORIAL 2

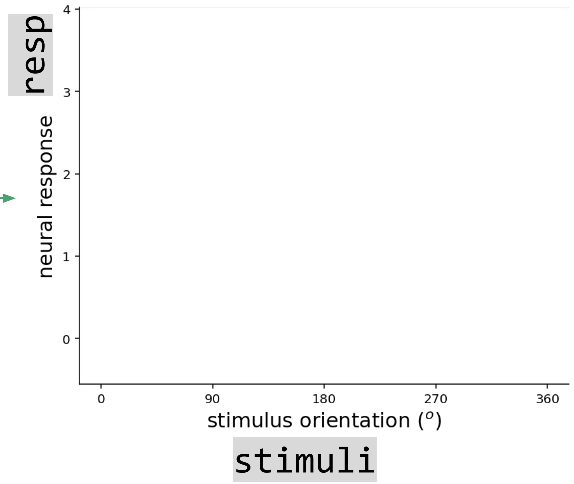


How does brain area X represent external correlate Y?

Neural recordings in mice







`resp contains responses of 23589 neurons to 360 stimuli`

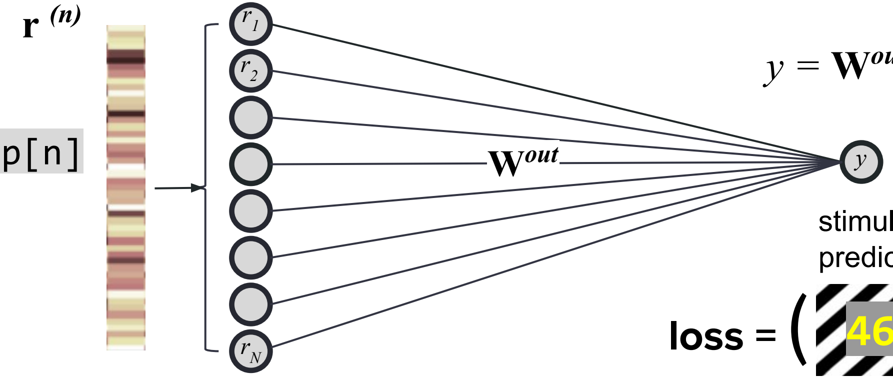
Stringer, Michaelos, Pachitariu, *bioRxiv*, 2019

Building a deep network

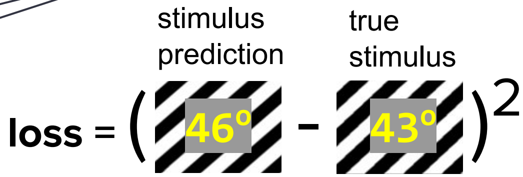
to predict *stimuli* from *neural responses*

Building a linear network

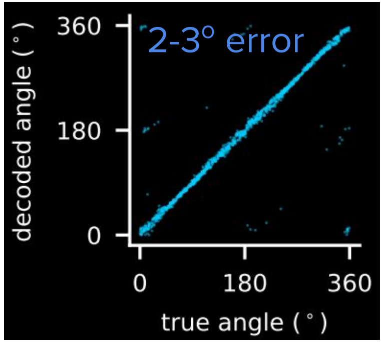
`resp[n]`



$$y = \mathbf{W}^{out}\,\mathbf{r} + \mathbf{b}^{out}$$




```
L = (y - stimuli[n])**2
```



Can we predict better using a deep network?

Building a linear network

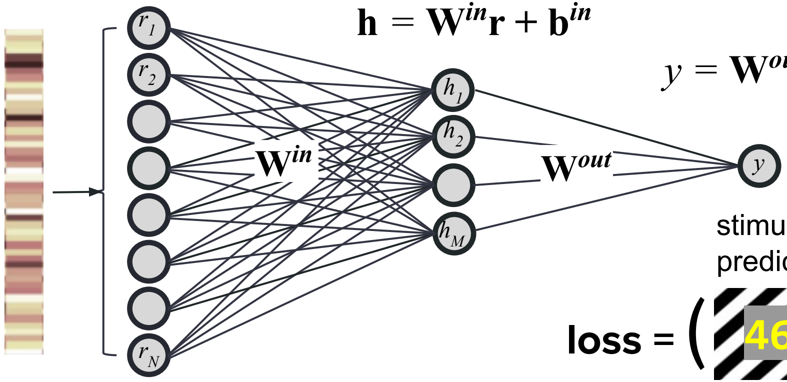
hidden layer $\mathbf{h}$

$\mathbf{h} = \mathbf{W}^{in}\mathbf{r} + \mathbf{b}^{in}$

$y = \mathbf{W}^{out}\mathbf{h} + \mathbf{b}^{out}$

$\mathbf{r}^{(n)}$

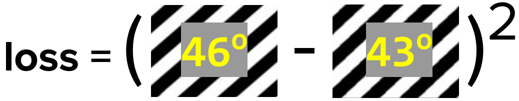
`resp[n]`



depth = 1
width = M

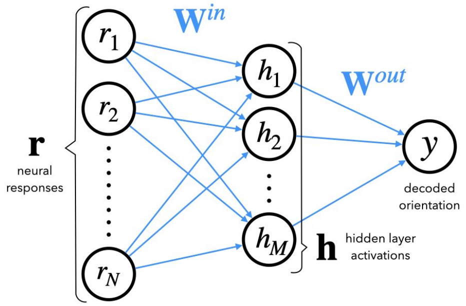
stimulus prediction

true stimulus




```
L = (y - stimuli[n])**2
```

Creating a linear network in pytorch



run the network

```
import torch
from torch import nn

class DeepNet(nn.Module):
  def __init__(self, n_inputs, n_hidden):
    super().__init__()  # needed to invoke the properties of the parent class nn.Module
    self.in_layer = nn.Linear(n_inputs, n_hidden) # neural activity --> hidden units
    self.out_layer = nn.Linear(n_hidden, 1) # hidden units --> output

  def forward(self, r):
    h = self.in_layer(r)  # hidden representation
    y = self.out_layer(h)
    return y

# Initialize a deep network with M=200 hidden units
net = DeepNet(n_neurons, 200)

# Get neural responses to first stimulus in the data set
istim = 0  # index of first stimulus
r = resp[istim]  # neural responses to this stimulus
# Decode orientation from these neural responses using initialized network
out = net(r)  # compute output from network, equivalent to net.forward(r)
```

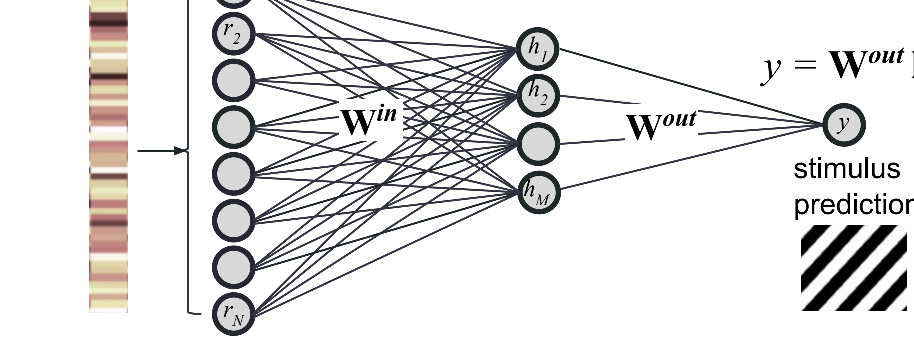
Activation functions

Activation functions

add non-linearities that allow flexible fitting

$$\mathbf{h} = \phi(\mathbf{W}^{in}\mathbf{r} + \mathbf{b}^{in})$$

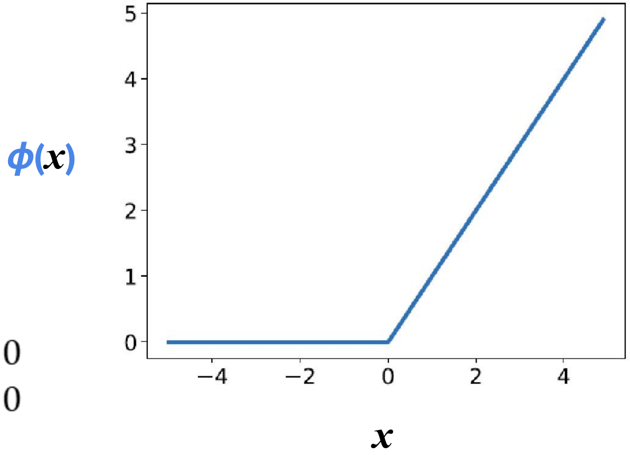
$$y = \mathbf{W}^{out}\mathbf{h} + \mathbf{b}^{out}$$



stimulus prediction

RELU

$$\phi(x) = \max(0, x)$$

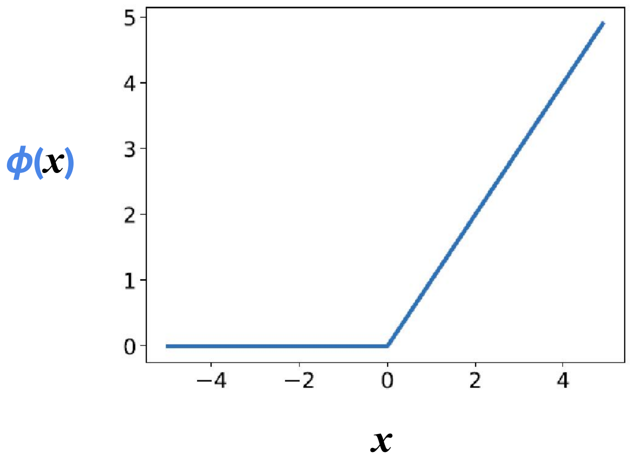


$$\frac{\partial \phi}{\partial x} = \begin{cases} 1, & if \;\; x \geq 0 \\ 0, & if \;\; x < 0 \end{cases}$$

RELU activation function in pytorch

```
class DeepNetReLU(nn.Module):

  def __init__(self, n_inputs, n_hidden):
    super().__init__()
    self.in_layer = nn.Linear(n_inputs, n_hidden)
    self.out_layer = nn.Linear(n_hidden, 1)

  def forward(self, r):
    h = torch.relu(self.in_layer(r))
    y = self.out_layer(h)
    return y
```

$\phi(x) = \max(0, x)$



Optimizing Neural Networks

Loss function

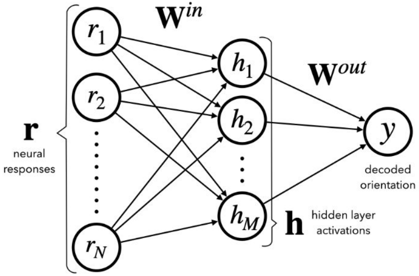
```
# Decode orientation from these neural responses
out = net(r)  # compute output from network

# Initialize PyTorch mean squared error loss
function
loss_fn = nn.MSELoss()

# Evaluate mean squared error
loss = loss_fn(out, ori)
print('mean squared error: %.2f' % loss)
```

```
mean squared error: 12.73
```

Gradient descent

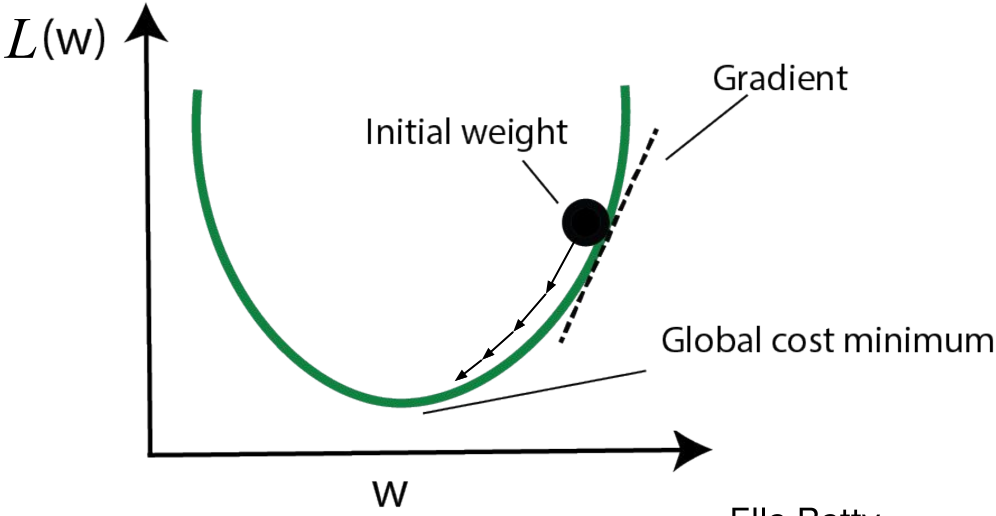
compute gradients wrt loss function L

$$\frac{\partial L}{\partial \mathbf{W}^{in}}, \frac{\partial L}{\partial \mathbf{b}^{in}}, \frac{\partial L}{\partial \mathbf{W}^{out}}, \frac{\partial L}{\partial \mathbf{b}^{out}}$$



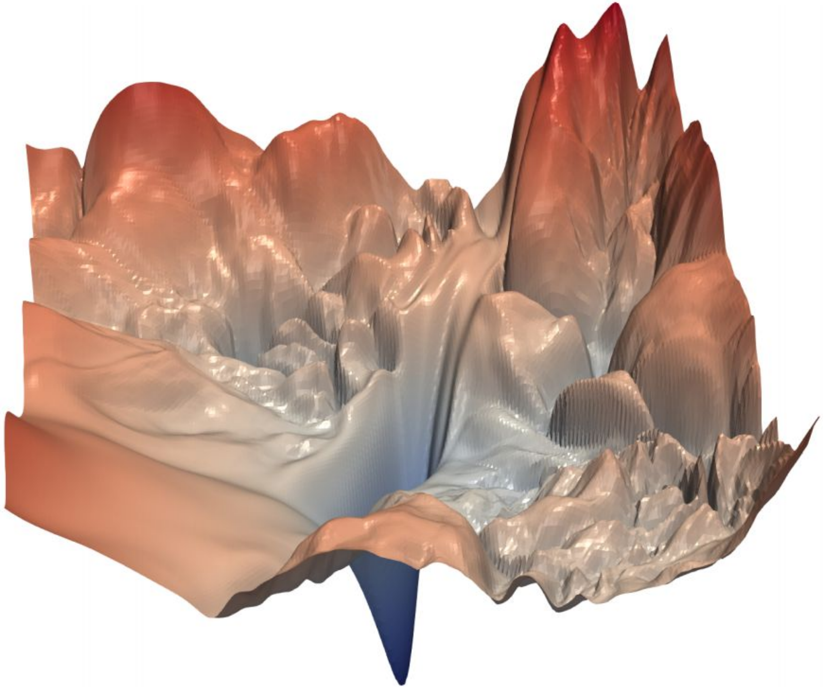
update parameters by gradients on each iteration

$$\mathbf{W}^{in} \leftarrow \mathbf{W}^{in} - \alpha \frac{\partial L}{\partial \mathbf{W}^{in}}$$
$$\mathbf{b}^{in} \leftarrow \mathbf{b}^{in} - \alpha \frac{\partial L}{\partial \mathbf{b}^{in}}$$
$$\mathbf{W}^{out} \leftarrow \mathbf{W}^{out} - \alpha \frac{\partial L}{\partial \mathbf{W}^{out}}$$
$$\mathbf{b}^{out} \leftarrow \mathbf{b}^{out} - \alpha \frac{\partial L}{\partial \mathbf{b}^{out}}$$

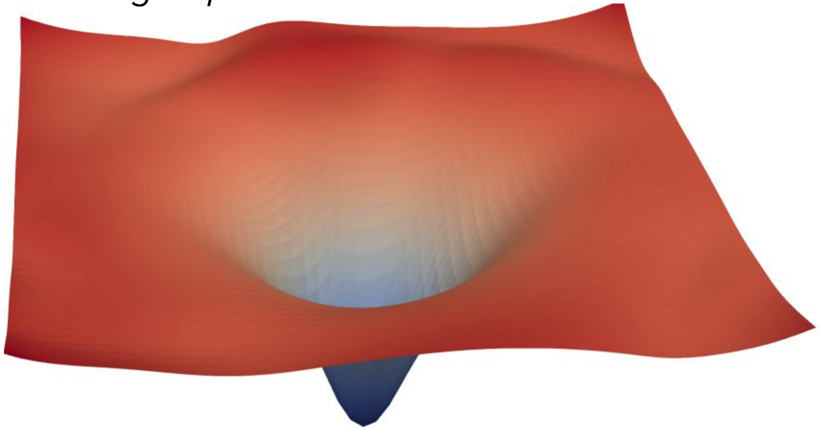


Ella Batty

Loss function



adding skip connections



Li , Xu, Taylor, Studer, Goldstein, *arXiv*, 2018

Gradient descent in pytorch

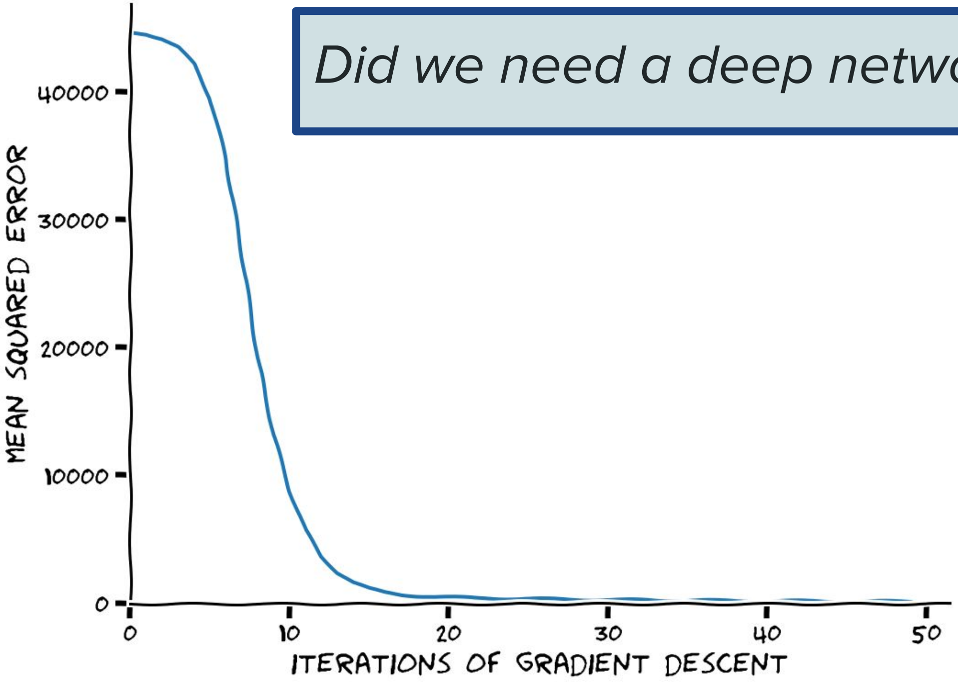
```
optimizer = optim.SGD(net.parameters(), lr=.001)

for i in range(n_iter):
  # Evaluate loss
  out = net(train_data)
  loss = loss_fn(out, train_labels)

  # Compute gradients
  optimizer.zero_grad()  # clear gradients
  loss.backward()

  # Update weights
  optimizer.step()
```

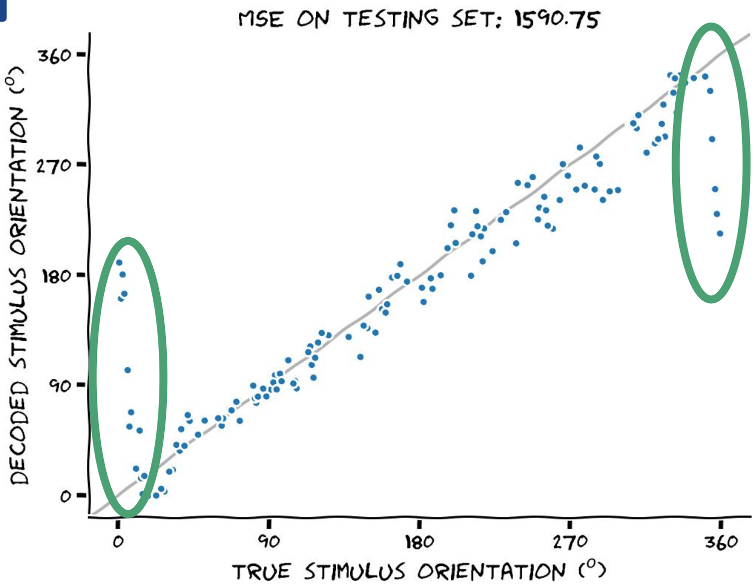
automatic differentiation!

you can use this for any complex model you want to fit!

Gradient descent in pytorch

Did we need a deep network?





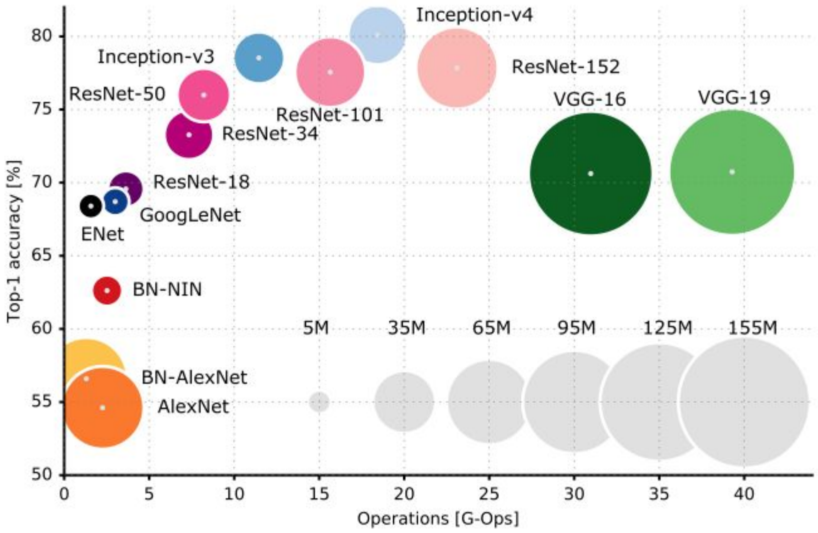
Regularization

Regularizing deep nets

Deep networks can have millions of parameters

How to prevent overfitting? ⇒ Regularization

- L2 regularization (weight decay)
- L1 regularization
- Batch normalization*
- Dropout
- Fewer neurons, fewer layers, use convolutions



Canziani et al, 2016

L2 regularization (weight decay)

$$\mathbf{Cost} = ||Y_{target} - \mathbf{net}(X_{input})||^2 + \lambda \sum ||W||^2$$

$$\partial\mathbf{Cost}/\partial W = \partial/\partial W ||Y_{target} - \mathbf{net}(X_{input})||^2 + 2\lambda \sum W$$

$$W \leftarrow W - \underbrace{\eta}_{\text{learning rate}} \cdot (\ldots + \underbrace{2\lambda W}_{\text{weight decay}})$$

```
L2 = L2_penalty * torch.square(weights).sum()
```

L1 regularization

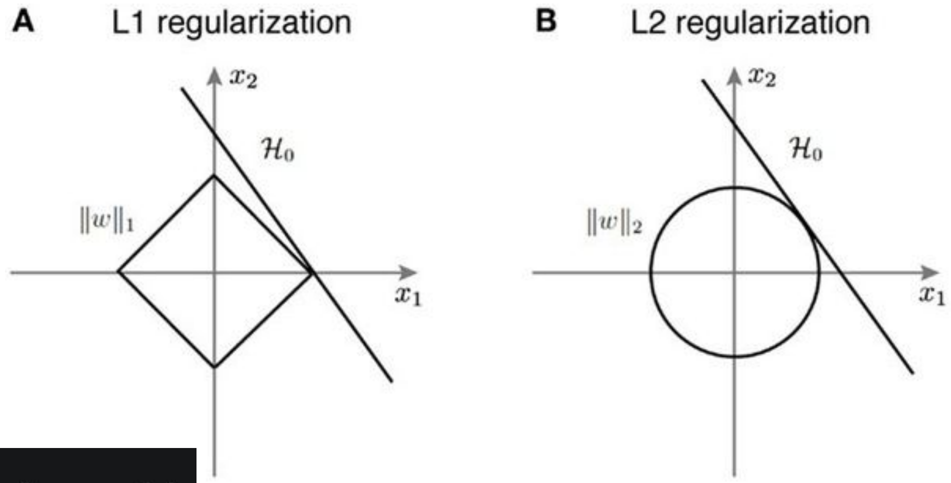
L1 norm = abs

$$\mathbf{Cost} = ||Y_{target} - \mathbf{net}(X_{input})||^2 + \lambda \sum |W|^1$$

$$W \leftarrow W - \eta \cdot (\ldots + \lambda \cdot \mathbf{sign}(W))))$$

- L1 produces sparse solutions
- L1 is more interpretable
- L1 is harder to optimize

```
L1 = L1_penalty * torch.abs(weights).sum()
```



C.V.Krishnakumar