Welcome!
#pod-031
Week #3, Day 4

(Reviewed by: Deepak)

neuromatch academy

facebook Reality Labs

Agenda

- Tutorial 1 (Decoding neural responses)
  - 3 exercises
- Tutorial 2 (encoding neural responses)
  - 2 exercise + 1 bonus
- Tutorial 3 (normal encoding models)
  - 1 Exercise + 3 bonus

Tutorial #1 Explanations

objective

- use deep learning to decode stimulus information from the responses of sensory neurons. Specifically, look at the activity of ~20,000 neurons in mouse primary visual cortex responding to oriented gratings recorded in this study. Our task will be to decode the orientation of the presented stimulus from the responses of the whole population of neurons.

  Why deep learning?

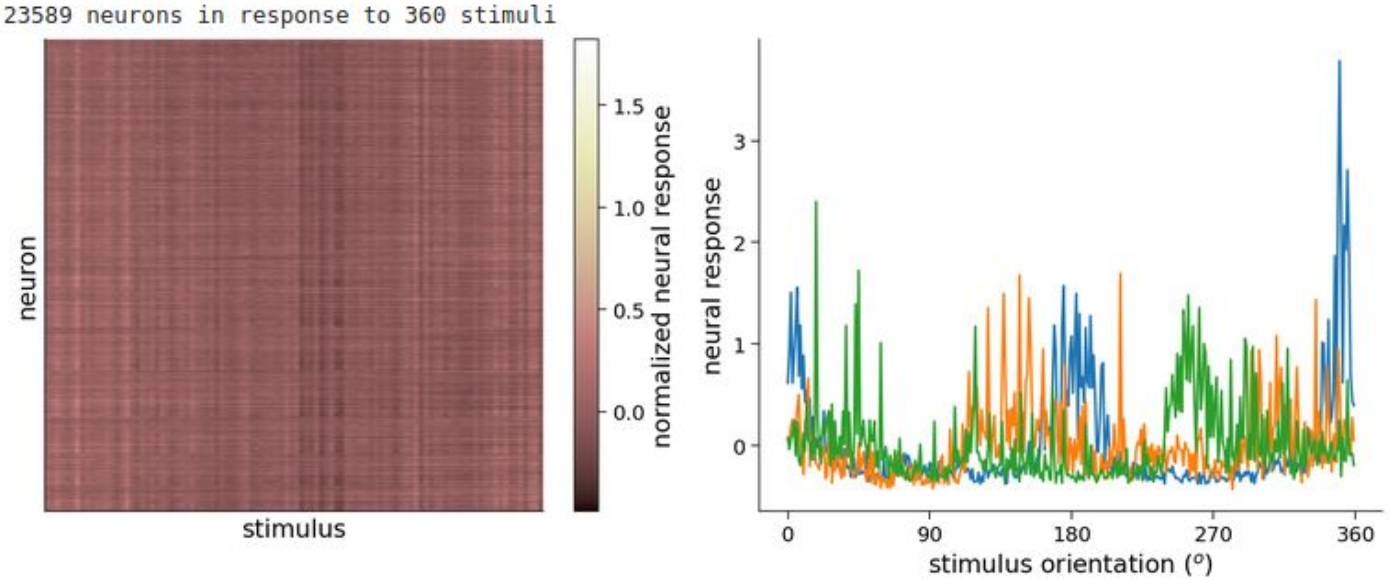
- The data are very high-dimensional: the neural response to a stimulus is a ~20,000 dimensional vector. Many machine learning techniques fail in such high dimensions, but deep learning actually thrives in this regime, as long as you have enough data
- different neurons can respond quite differently to stimuli. This complex pattern of responses will, therefore, require non-linear methods to be decoded, which we can easily do with non-linear activation functions in deep networks.
- Deep learning architectures are highly flexible, meaning we can easily adapt the architecture of our decoding model to optimize decoding.

goal:

- Build a deep feed-forward network using PyTorch
- Evaluate the network's outputs using PyTorch built-in loss functions
- Compute gradients of the loss with respect to each parameter of the network using automatic differentiation
- Implement gradient descent to optimize the network's parameters

Load and visualize data

Algorithm: load the data and plot the matrix of neural responses, then plot the tuning curves of three randomly selected neurons.



Architectures

Split data into a training set and test set. Have a training set of orientations (stimuli_train) and the corresponding responses (resp_train). Testing set will have held-out orientations (stimuli_test) and the corresponding responses (resp_test).

Simple deep neural network with one hidden layer: takes as input a vector of neural responses and outputs a single number representing the decoded stimulus orientation.

$$r^{(n)} = \begin{bmatrix} r_1^{(n)} & r_2^{(n)} & \cdots & r_N^{(n)} \end{bmatrix}^T \quad \} \text{(neurons } 1 \ldots N\text{)}$$

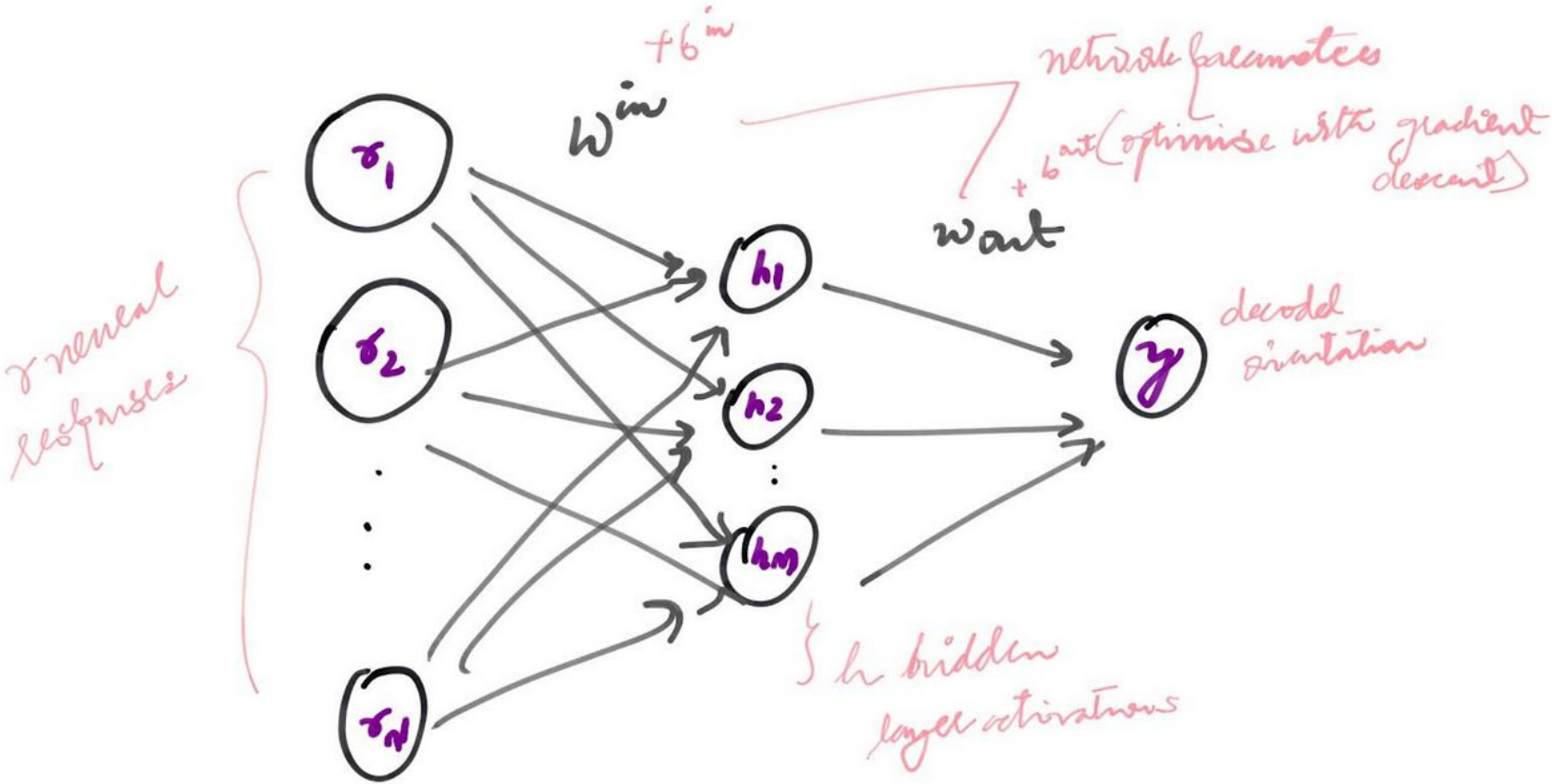
denotes vector of neural responses to the nth stimulus

activations of hidden layer of n/w

$$h^{(n)} = W^{in} r^{(n)} + b^{in} \qquad [W^{in} : M \times N]$$

$$y^{(n)} = W^{out} h^{(n)} + b^{out} \qquad (W^{out} : 1 \times M)$$

Scalar o/p of network } decoded orientation of nth stimulus


$+b^{in}$
W^{in}
network parameters
$+b^{out}$ (optimise with gradient descent)
W^{out}
r_1
r_2
r_N
h_1
h_2
h_M
y
r neural response
decoded orientation
h hidden layer activations

Intro to pytorch - 1

Use the PyTorch package to build, run, and train deep networks in Python. There are two core components to the PyTorch package:

1. The first is the torch.Tensor data type used in PyTorch. torch.Tensor's are effectively just like a numpy arrays, except that they have some important attributes and methods needed for automatic differentiation. They also come along with infrastructure for easily storing and computing with them on GPU's.

Intro to pytorch - 2

2. The second core ingredient is the PyTorch nn.Module class for constructing deep networks, so that we can then easily train them using built-in PyTorch functions.
nn.Module classes can actually be used to build, run, and train any model -- not just deep networks! It contains three key ingredients:

- __init__() method to initialize its parameters, like in any other Python class. In this case, it takes two arguments:
  - n_inputs: the number of input units. This should always be set to the number of neurons whose activities are being decoded (i.e. the dimensionality of the input to the network).
  - n_hidden: the number of hidden units. This is a parameter that we are free to vary in deciding how to build our network.
  - nn.Linear modules, which are built-in PyTorch classes containing all the weights and biases for a given network layer. This class takes two arguments to initialize: # of inputs to that layer, # of outputs from that layer

Intro to pytorch ~3

- For the input layer, for example, we have:
  - # of inputs = # of neurons whose responses are to be decoded (*M*, specified by n_inputs)
  - # of outputs = # of hidden layer units (*M*, specified by n_hidden)
- PyTorch will initialize all weights and biases randomly.
- forward() method, which takes as argument an input to the network and returns the network output. In our case, this comprises computing the output y from a given input **r**

Intro to pytorch - 4

Initializing and running this network:

decode stimulus orientation from a vector of neural responses to the very first stimulus. Note that when the initialized network class is called as a function on an input (e.g. net(r)), its .forward() method is called. This is a special property of the nn.Module class.

Note that the decoded orientations at this point will be nonsense, since the network has been initialized with random weights. Optimize these weights for good stimulus decoding.

Non linear activation functions

$$y^{(n)} = W^{out}\left(W^{in}x^{(n)} + b^{(in)}\right) + b^{out}$$

output is weighted sum of elements in input

$$= W^{out}W^{in}x^{(n)} + \left(W^{out}b^{(in)} + b^{out}\right)$$

Non linear activation function in hidden units.

(extends set of computable i/o transformations to more than just weighted sums).

activation function

$$h^{(n)} = \phi\left(W^{in}x^{(n)} + b^{in}\right)$$

Using a non-linear activation function will ensure that the hidden layer performs a non-linear transformation of the input, which will make our network much more powerful (or expressive, cf. appendix). In practice, deep networks always use non-linear activation functions.

$$\phi(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{else} \end{cases}$$

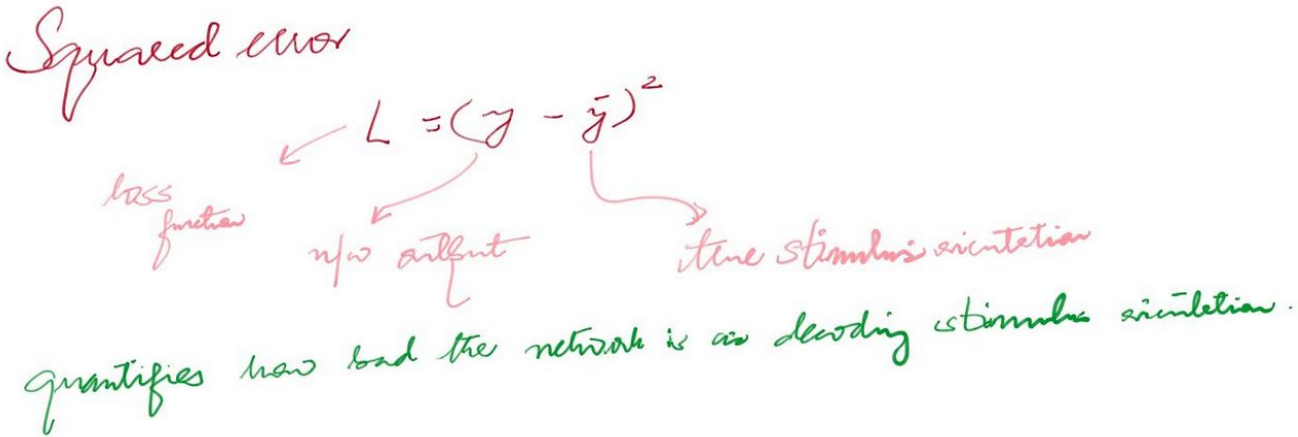
linear rectification function:

hidden layers with this activation function are typically referred to as rectified linear units (ReLU).

Loss functions

Weights of the network are currently randomly chosen, the outputs of the network: the decoded stimulus orientation is nowhere close to the true stimulus orientation.

When the decoded stimulus orientation is far from the true stimulus orientation, L will be large.



Mean squared error

squared error is called nn.MSELoss(). This will take as arguments a **batch** of network outputs

$$y1, y2, \ldots, yP$$

and corresponding target outputs

$$y\sim1, y\sim2, \ldots, y\sim P$$

and compute the **mean squared error (MSE)**

MSE :

$$L = \frac{1}{P} \sum_{n=1}^{P} \left(y^{(n)} - \bar{y}^{(n)} \right)^2$$

Optimization with gradient descent - Step 1

goal is now to modify the weights to make the mean squared error loss L as small as possible over the whole data set.

use the **gradient descent (GD)** algorithm, which consists of iterating three steps:

1. **Evaluate the loss** on the training data,

```
out = net(train_data)

loss = loss_fn(out, train_labels)
```

where train_data are the network inputs in the training data (neural responses), and train_labels are the target outputs for each input (true stimulus orientations).

Optimization with gradient descent - Step 2

2. Compute the gradient of the loss with respect to each of the network weights. In PyTorch, we can do this with one line of code:

```
loss.backward()
```

This command tells PyTorch to compute the gradients of the quantity stored in the variable loss with respect to each network parameter using automatic differentiation. These gradients are then stored behind the scenes.

Optimization with gradient descent - Step 3

Update the network weights by descending the gradient. In Pytorch, we can do this using built-in optimizers. We'll use the optim.SGD optimizer (documentation here) which updates parameters along the negative gradient, scaled by a learning rate. To initialize this optimizer, we have to tell it

- which parameters to update, and
- what learning rate to use

For example, to optimize all the parameters of a network net using a learning rate of .001, the optimizer would be initialized as follows

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

where .parameters() is a method of the nn.Module class that returns a Python generator object over all the parameters of that nn.Module class ($\mathbf{W}_{in}$, $\mathbf{b}_{in}$, $\mathbf{W}_{out}$, $\mathbf{b}_{out}$).

Parameters gradients

After computing all the parameter gradients in step 2, we can then update each of these parameters using the .step() method of this optimizer,

```
optimizer.step()
```

Extract all the gradients computed with .backward() and execute the SGD updates for each parameter given to the optimizer.

Gradients of each parameter need to be cleared before calling .backward(), or else PyTorch will try to accumulate gradients across iterations; can again be done using built-in optimizers via the method zero_grad(), as follows:

```
optimizer.zero_grad()
```

Pytorch code details

```
Get outputs from network
Evaluate loss# Compute gradients

optimizer.zero_grad()  # clear gradients

loss.backward()

# Update weights

optimizer.step()
```

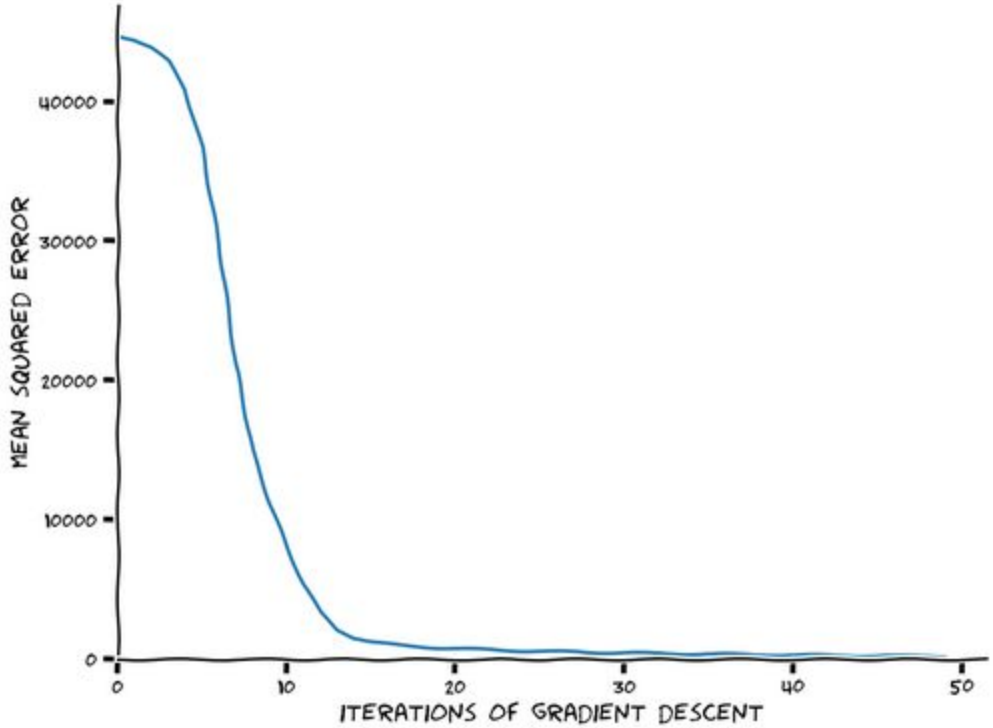
Gradient descent in PyTorch

train() function takes as input arguments

- net: the PyTorch network whose weights to optimize
- loss_fn: the PyTorch loss function to use to evaluate the loss
- train_data: the training data to evaluate the loss on (i.e. neural responses to decode)
- train_labels: the target outputs for each data point in train_data (i.e. true stimulus orientations)

Train a neural network on our data and plot the loss (mean squared error) over time. When we run this function, behind the scenes PyTorch is actually changing the parameters inside this network to make the network better at decoding, so its weights will now be different than they were at initialization.

```
iteration 10/50 | loss: 11225.866
iteration 20/50 | loss: 796.539
iteration 30/50 | loss: 449.642
iteration 40/50 | loss: 285.460
iteration 50/50 | loss: 186.747
```

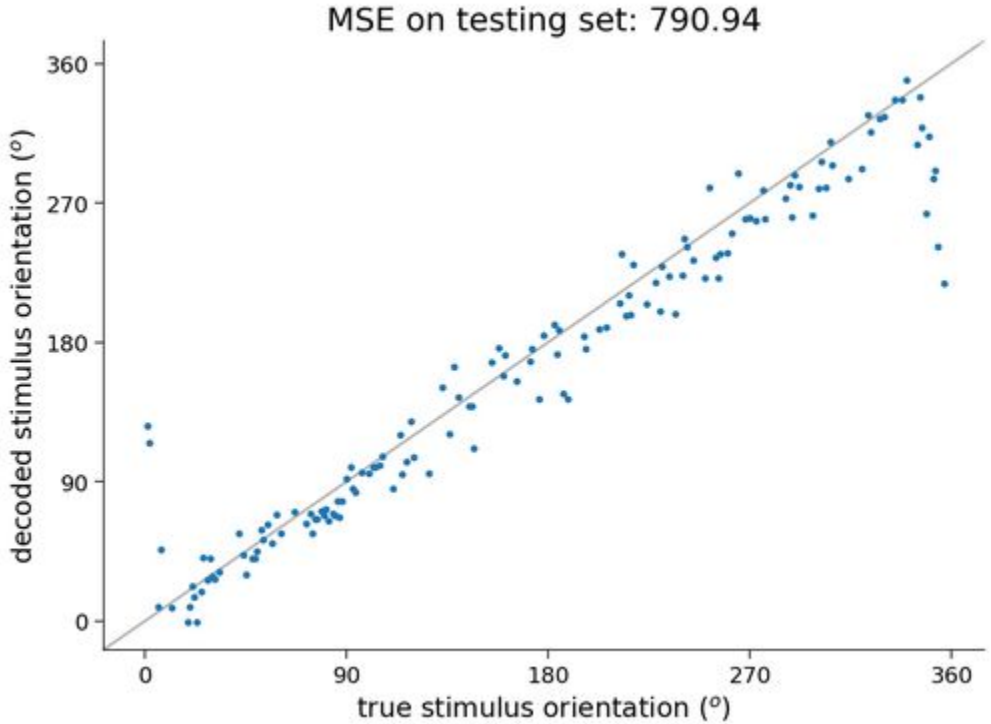


Generalization performance with test data

Note that gradient descent is essentially an algorithm for fitting the network's parameters to a given set of training data. Selecting this training data is thus crucial for ensuring that the optimized parameters ***generalize*** *to unseen data they weren't trained on. Make sure that our trained network is good at decoding stimulus orientations from neural responses to any orientation, not just those in our data set.*

To ensure this, we have split up the full data set into a ***training set*** *and a* ***testing set****. Traind a deep network by optimizing the parameters on a training set. Now evaluate how good the optimized parameters are by using the trained network to decode stimulus orientations from neural responses in the testing set. Good decoding performance on this testing set should then be indicative of good decoding performance on the neurons' responses to any other stimulus orientation through* ***cross-validation****.*

Generalization performance with test data



PyTorch Note:

An important thing to note for plotting the decoded orientations is the .detach() method.
The PyTorch nn.Module class: each of the variables inside it are linked to each other in a computational graph, for the purposes of automatic differentiation (the algorithm used in .backward() to compute gradients). As a result, for anything that is not a torch operation to the parameters or outputs of an nn.Module class, first "detach" it from its computational graph. T

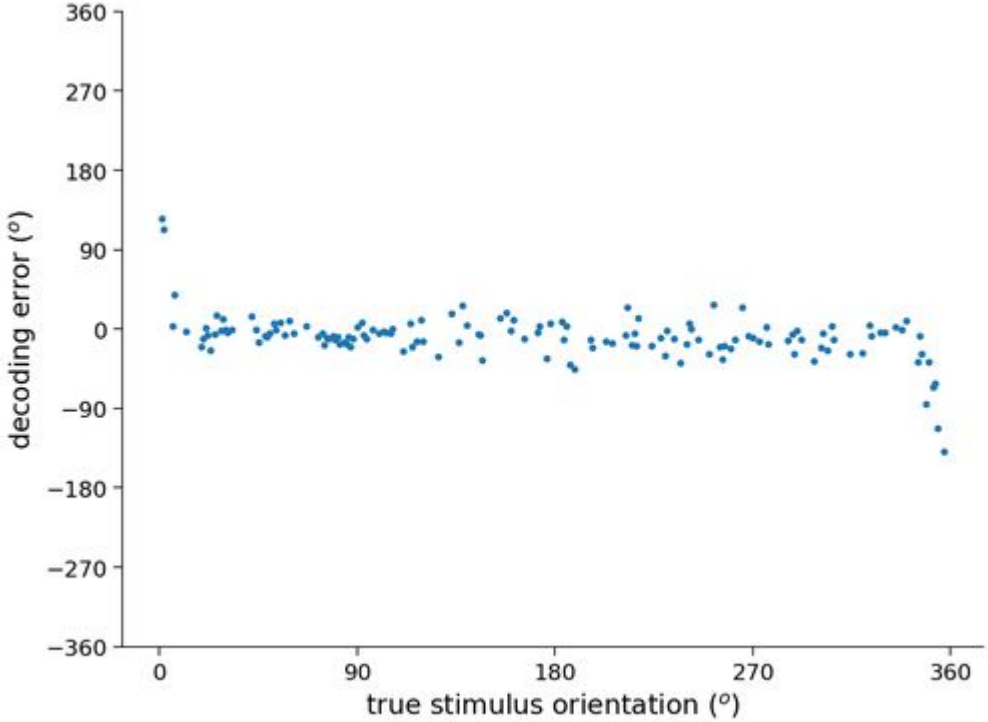
Tutorial #1 bonus Explanations

Model criticism

Plot the *decoding error* for each neural response in the testing set.

The decoding error is defined as the decoded stimulus orientation minus true stimulus orientation

Decoding error= $Y(n) - Y\sim(n)$



Food for thought

Plot decoding error as a function of the true stimulus orientation.

- *Are some stimulus orientations harder to decode than others? If so, in what sense? Are the decoded orientations for these stimuli more variable and/or are they biased?*

It appears that the errors are larger at 0 and 360 degrees. The errors are biased in the positive direction at 0 degrees and in the negative direction at 360 degrees. This is because the 0 degree stimulus and the 360 degree stimulus are in fact the same because orientation is a circular variable. The network therefore has trouble determining whether the stimulus is 0 or 360 degrees.

- *Can you explain this variability/bias? What makes these stimulus orientations different from the others?*
- *Can you think of a way to modify the deep network in order to avoid this?*

We can modify the deep network to avoid this problem in a few different ways. One approach would be to predict a sine and a cosine of the angle and then taking the predicted angle as the angle of the complex number $sin(\theta) + i cos(\theta)$.
An alternative approach is to bin the stimulus responses and predict the bin of the stimulus. This turns the problem into a classification problem rather than a regression problem, and in this case you will need to use a new loss function.

Squared error

The squared error is not a good loss function for circular quantities like angles, since two angles that are very close (e.g. 1 and 359) might actually have a very large squared error.

<u>Decoding problem as a classification problem:</u>

Rather than estimating the exact angle of the stimulus, we'll now aim to construct a decoder that classifies the stimulus into one of C classes, corresponding to different bins of angles of width B

$b = \frac{360}{C}$ ← no of classes.

true class $\bar{y}^{(n)}$ of stimulus i is given by -

$$\bar{y}^{(n)} = \begin{cases} 1 & \text{if angle of stimulus } n \text{ is in range } [0, b] \\ 2 & \text{——"——} \quad [b, 2b] \\ \vdots & \vdots \\ C & \text{if angle of stimulus } n \text{ is in range } [(C-1)b, 360] \end{cases}$$

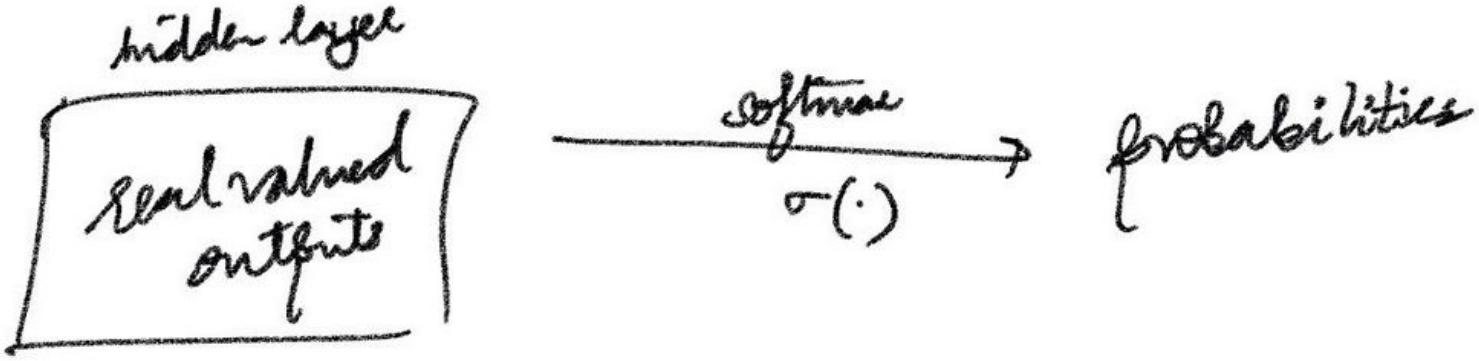
decode stimulus class from neural responses

→ use deep network

$$p = [p_1 \; p_2 \; \cdots \; p_C]$$

vector of probabilities

C dimensions



$$h^{(n)} = \phi\left(W^{in} x^{(n)} + b^{in}\right) \qquad \left[W^{in} : M \times N\right]$$

$$p^{(n)} = \sigma\left(W^{out} h^{(n)} + b^{out}\right) \qquad \left[W^{out} : C \times M\right]$$

decoded stimulus class given by assigned highest probability by the n/w:

$$y^{(n)} = \arg\max_i p_i$$

probabilities are too small ⇒ log probabilities { monotonic

$$I^{(n)} = \log\left(p^{(n)}\right)$$

$$y^{(n)} = \arg\max_i p_i^{(n)} = \arg\max_i \log p_i^{(n)} = \arg\max_i l_i^{(n)}$$

Objective : max log probability of true stimulus class $\bar{y}^{(n)}$ under class probabilities predicted by the network.

$$\log\left(\begin{array}{c}\text{predicted probability of stimulus } n \\ \text{being of class } \bar{y}^{(n)}\end{array}\right) = \log P^{(n)}_{\bar{y}^{(n)}}$$

$$= l^{(n)}_{\bar{y}^{(n)}}$$

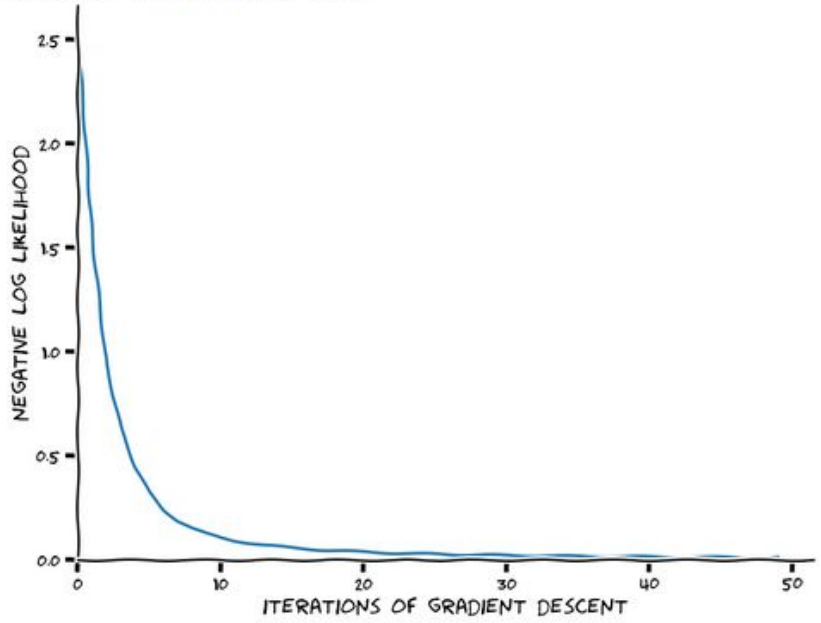
convert to loss function to be minimized
$\equiv$ min negative log probability

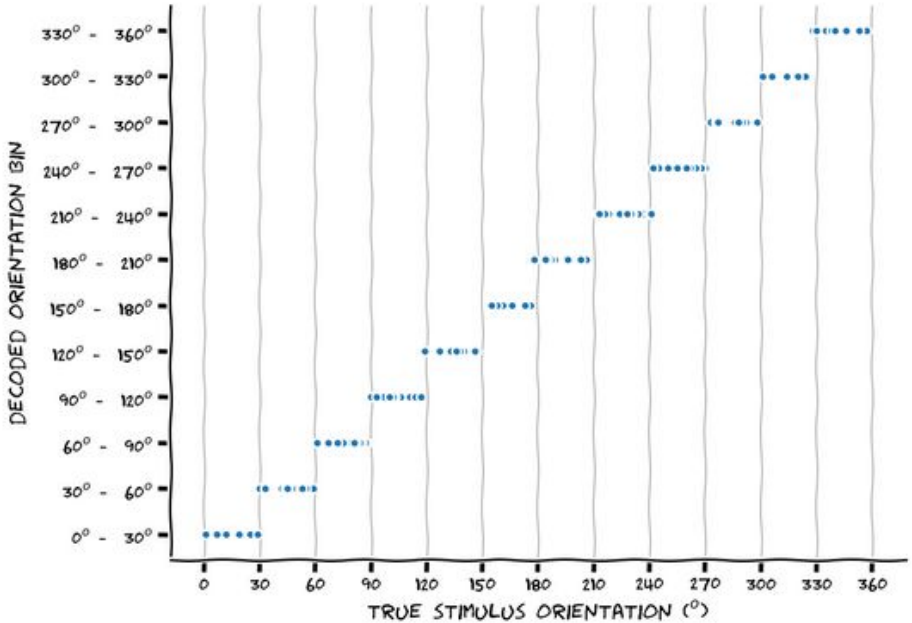
loss function : $L = -\sum_{n=1}^{P} \log P^{(n)}_{\bar{y}^{(n)}} = -\sum_{n=1}^{P} l^{(n)}_{\bar{y}^{(n)}}$

loss function ≡ cross entropy ≡ negative log likelihood

Objective: decode stimulus orientation via classification, by minimizing negative log likelihood.

```
iteration 10/50 | loss: 0.126
iteration 20/50 | loss: 0.042
iteration 30/50 | loss: 0.023
iteration 40/50 | loss: 0.016
iteration 50/50 | loss: 0.012
```





SUMMARY

We have now covered a number of common and powerful techniques for applying deep learning to decoding from neural data, some of which are common to almost any machine learning problem:

- Building and training deep networks using the **PyTorch** nn.Module class and built-in **optimizers**
- Choosing and evaluating **loss functions**
- Testing a trained model on unseen data via **cross-validation**, by splitting the data into a **training set and testing set**

train() function: Note that it can be used to train *any* network to minimize *any* loss function on *any* training data. As long as its parameters and computations involve only torch.Tensor's, and the model is differentiable, we can optimize the parameters of this model.

If we can decode the stimulus well from visual cortex activity, that means that there is information about this stimulus available in visual cortex. Whether or not the animal uses that information to make decisions is not determined from an analysis like this. In fact mice perform poorly in orientation discrimination tasks compared to monkeys and humans, even though they have information about these stimuli in their visual cortex.

See this paper for some potential hypotheses (https://www.biorxiv.org/content/10.1101/679324v2), but this is totally an open question!

Neural network depth, width and expressivity #1

Two important architectural choices that always have to be made when constructing deep feed-forward networks:

- the number of hidden layers, or the network's *depth*
- the number of units in each layer, or the layer *widths*

Here, we restricted ourselves to networks with a single hidden layer with a width of M units, but this code could be adapted to arbitrary depths. Adding another hidden layer simply requires adding another nn.Linear module to the `__init__()` method and incorporating it into the .forward() method.

The depth and width of a network determine the set of input/output transformations that it can perform, often referred to as its *expressivity*. The deeper and wider the network, the more *expressive* it is; that is, the larger the class of input/output transformations it can compute. In fact, it turns out that an infinitely wide or infinitely deep networks can in principle compute (almost) any input/output transformation.

Neural network depth, width and expressivity #2

A classic mathematical demonstration of the power of depth is given by the so-called XOR problem. This problem demonstrates how even a single hidden layer can drastically expand the set of input/output transformations a network can perform, relative to a shallow network with no hidden layers. The key intuition is that the hidden layer allows you to represent the input in a new format. The *wider* this hidden layer, the more flexibility you have in this representation. In particular, if you have more hidden units than input units, then the hidden layer representation of the input is higher-dimensional than the raw data representation. This higher dimensionality effectively gives you more "room" to perform arbitrary computations in. It turns out that even with just this one hidden layer, if you make it wide enough you can actually approximate any input/output transformation you want. See here for a neat visual demonstration of this. In practice, however, it turns out that increasing depth seems to grant more expressivity with fewer units than increasing width does (for reasons that are not well understood). It is for this reason that truly *deep* networks are almost always used in machine learning, which is why this set of techniques is often referred to as *deep* learning.

That said, there is a cost to making networks deeper and wider. The bigger your network, the more parameters (i.e. weights and biases) it has, which need to be optimized! The extra expressivity afforded by higher width and/or depth thus carries with it (at least) two problems:

- optimizing more parameters usually requires more data
- a more highly parameterized network is more prone to overfit to the training data, so requires more sophisticated optimization algorithms to ensure generalization

Gradient descent equations (Algorithm)

Step 1 : evaluate loss

mean squared error loss

$$L = \frac{1}{P} \sum_{n=1}^{P} \left(y^{(n)} - \bar{y}^{(n)} \right)^2$$

stimulus orientation decoded from population response $\bar{y}^{(n)}$ to n^{th} stimulus in training data

true orientation of n^{th} stimulus

no of data samples in training set

Step 2: compute gradient of loss (wrt n/w weights)

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

using pytorch (automatic differentiation)

Gradients

More specifically, when this function is called on a particular variable (e.g. loss, as above), PyTorch will compute the gradients with respect to each network parameter. These are computed and stored behind the scenes, and can be accessed through the .grad attribute of each of the network's parameters.

Step 3 : Update n/w weights (using descent of gradient)

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

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

learning rate

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

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

Hyperparameter

This hyperparameter of the SGD algorithm controls how far we descend the gradient on each iteration. It should be as large as possible so that fewer iterations are needed, but not too large so as to avoid parameter updates from skipping over minima in the loss landscape.

Stochastic gradient descent (SGD) vs. gradient descent (GD) #1

The key difference is in the very first step of each iteration, where in the GD algorithm we evaluate the loss at every data sample in the training set. In SGD, on the other hand, we evaluate the loss only at a random subset of data samples from the full training set, called a **mini-batch**.

At each iteration, we randomly sample a mini-batch:

The P data samples $x(n), y(n)$ denote a mini-batch of P random samples from the training set, rather than the whole training set.

Stochastic gradient descent (SGD) vs. gradient descent (GD) #2

There are several reasons why one might want to use SGD instead of GD.

1. The training set might be too big, so that we actually can't actually evaluate the loss on every single data sample in it. In this case, GD is simply infeasible, so we have no choice but to turn to SGD, which bypasses the restrictive memory demands of GD by sub-sampling the training set into smaller mini-batches.
2. But, even when GD is feasible, SGD turns out to be generally better. The stochasticity induced by the extra random sampling step in SGD effectively adds some noise in the search for local minima of the loss function. This can be really useful for avoiding potential local minima, and enforce that whatever minimum is converged to is a good one. This is particularly important when networks are wider and/ or deeper, in which case the large number of parameters can lead to overfitting.

Here, we used only GD because (1) it is simpler, and (2) it suffices for the problem being considered here. Because we have so many neurons in our data set, decoding is not too challenging and doesn't require a particularly deep or wide network. The small number of parameters in our deep networks therefore can be optimized without a problem using GD.

Tutorial #2
Explanations

Objective

Use deep learning to build an encoding model from stimuli to neural activity. Specifically, we'll be looking at the activity of ~20,000 neurons in mouse primary visual cortex responding to oriented gratings recorded in this study.

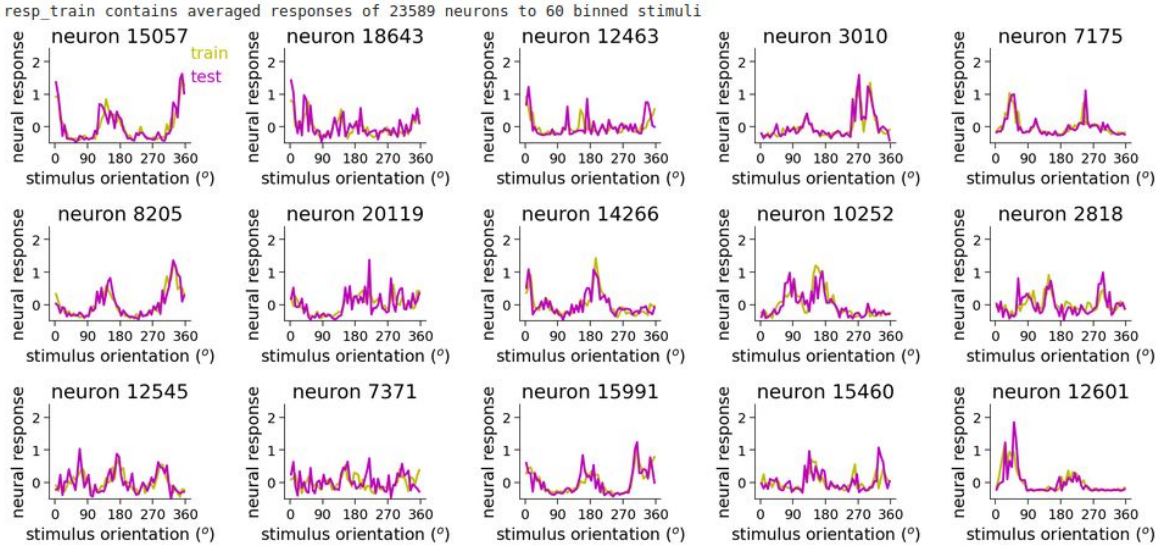
Because the stimuli are 1D and the neurons respond with smooth tuning curves, we will model the neural responses as a 1D convolutional operation on the stimulus.

- *Understand the basics of convolution*
- *Build and train a convolutional neural network to predict neural responses using PyTorch*
- *Visualize and analyze its internal representations*

Neural Tuning curves

Plot the turning curves of a random subset of neurons. We have binned the stimuli orientations. Look at different example neurons and observe the diversity of tuning curves in the population.

How can we fit these neural responses with an encoding model?



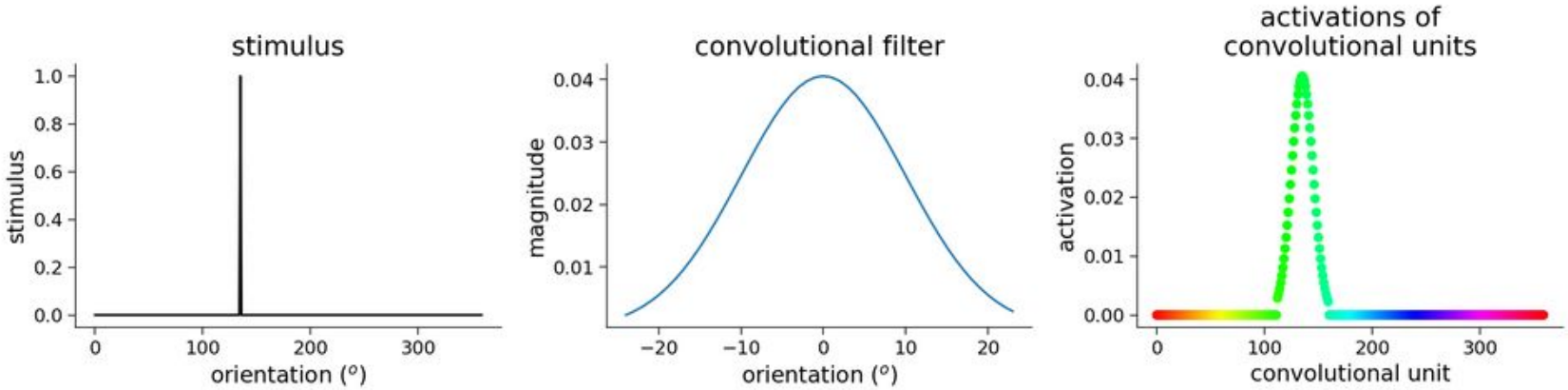
1D convolution

$$a_x = \sum_{i=-K/2}^{K/2} f_i \cdot s_{x-i}$$

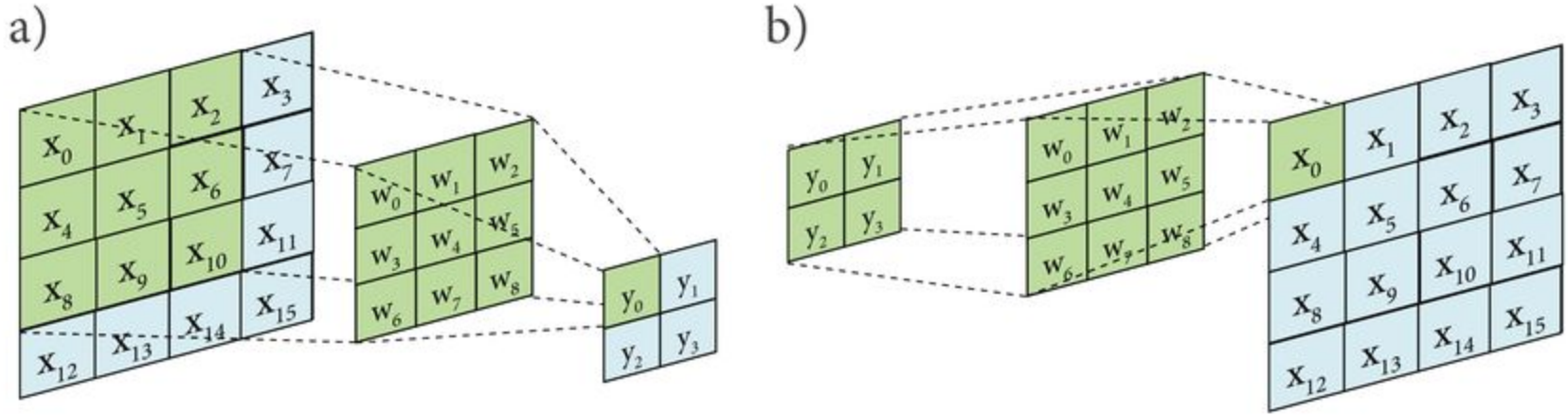
stimulus input

convolutional o/p @ pos. x

filter/kernels (size: K)


stimulus
stimulus
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
orientation (°)
convolutional filter
magnitude
0.04
0.03
0.02
0.01
−20
−10
0
10
20
orientation (°)
activations of
convolutional units
activation
0.04
0.03
0.02
0.01
0.00
0
100
200
300
convolutional unit

Convolution



convolutional layer

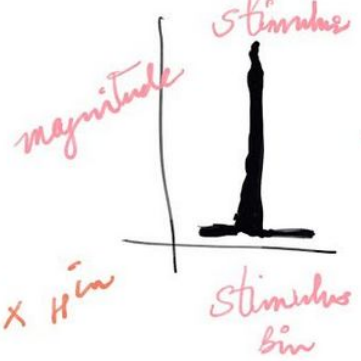
learned how to compute what is called a single convolutional ***channel****: a single filter applied to the input resulting in several units, where the number of units depends on the stride you set.*

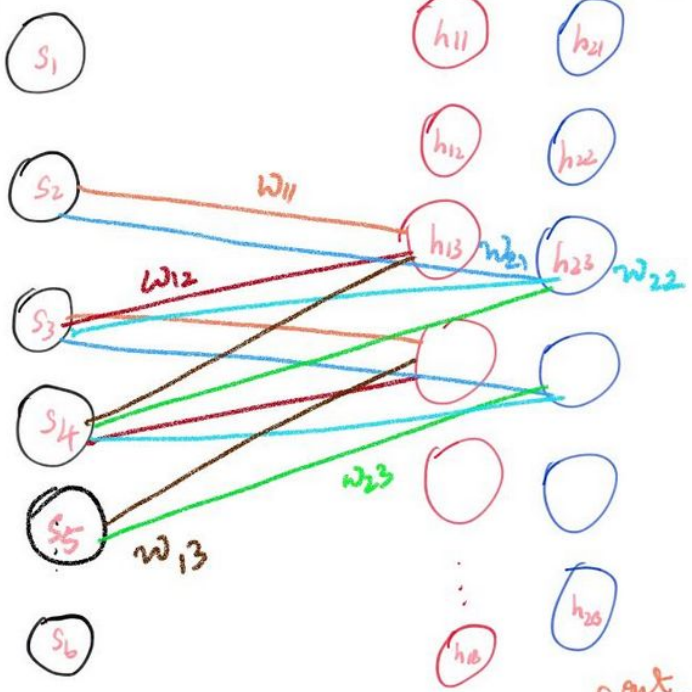
(Note if filter size K is odd and you set the pad=K//2 and stride=1 (as is the default above), you get a ***channel*** *of units that is the same size as the input.)*

Food for thought: How does a neuron potentially combine those activation units and create the tuning curves they have? Will we need more than one convolutional filter to recreate all the responses we see?

We can think of all neurons being composed of the same shaped peak but at different/multiple locations for each neuron. However, it appears that different neurons have peaks that are shaped differently, or even the same neuron has two peaks with different shapes. Therefore we may want multiple different convolutional filters.

C^{in} = no of output channels





$h = w^{in} * s + b^{in}$

↓ = size of stimulus bins

k = size of filter / Size of C^{out} different Convolutional filters

$C^{out} = \begin{cases} \text{no of o/p channels} \\ \text{no of convolutional channels/filters} \end{cases}$

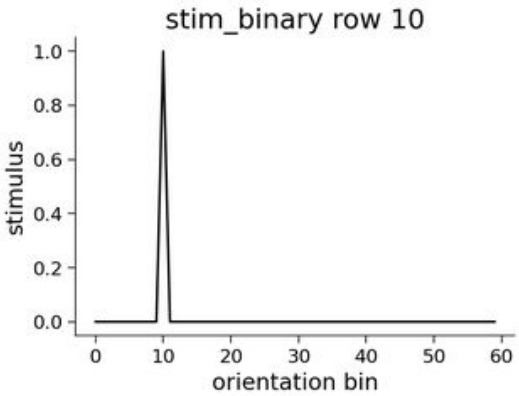
$M = BC^{out}$ = No. of hidden units

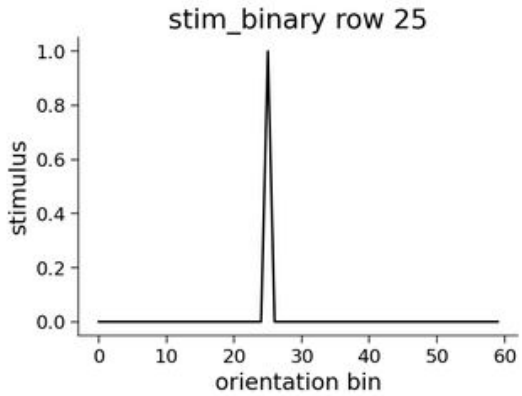
layer of convolutional channels

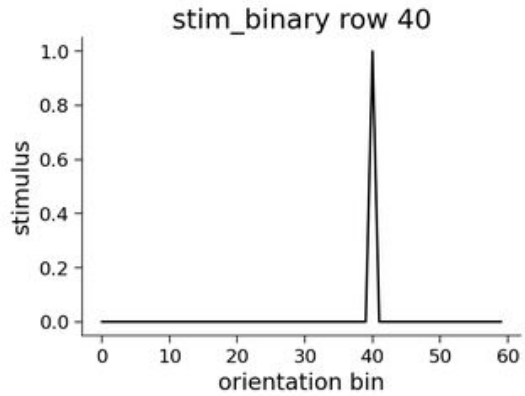
layer of convolutional channels can be implemented using the PyTorch class nn.Conv1d()

When you run the network, you can input a stimulus of arbitrary length (Hin), but it needs to be shaped as a 2D input Cin× Hin. In our case, Cin=1 because there is only one orientation input and Hin is the number of stimulus bins B.

1D convolution: In particular, we will use the binary stimuli (stim_binary), which is a 60 x 60 tensor where each row contains the binary stimuli for one orientation (all zeros except for a one at that orientation). This tensor is size 60 instead of 360 because we have binned the orientations. Each row of this matrix is a different example orientation that we want to convolve.


stim_binary row 10
stimulus
orientation bin
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60


stim_binary row 25
stimulus
orientation bin
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60

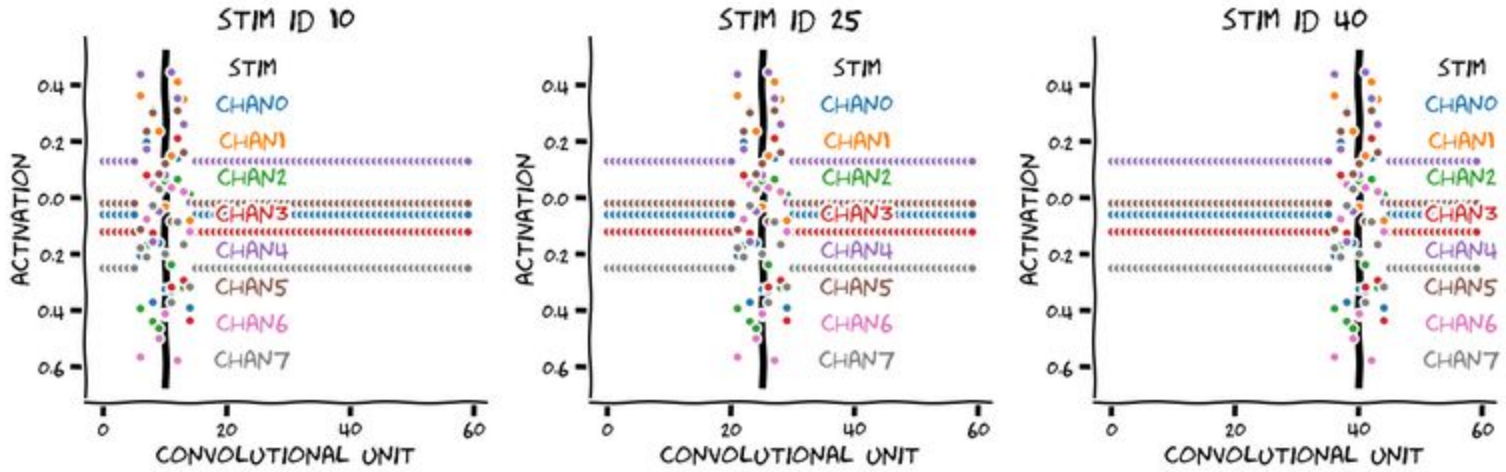

stim_binary row 40
stimulus
orientation bin
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60

nn.Conv1d

nn.Conv1d takes in a tensor of size (N, Cin, Hin) where
N is the number of examples,
Cin is the number of input channels, and
Hin is the number of stimulus bins B.

Since our stimulus has only one input channel, the ConvolutionalLayer class adds the Cin dimension for us: we need to input an (N,Hin)stimulus, which stim_binary is!

We will plot the outputs of the convolution. convout is a tensor of size (N,Cout, Hin) where
N is the number of examples and
Cout are the number of convolutional channels.


STIM ID 10
STIM ID 25
STIM ID 40
STIM
CHAN0
CHAN1
CHAN2
CHAN3
CHAN4
CHAN5
CHAN6
CHAN7
ACTIVATION
0.4
0.2
0.0
0.2
0.4
0.6
0
20
40
60
CONVOLUTIONAL UNIT

Food for thought

- Why are the convolutional activations for a given channel the same for many units?

  the stimulus is zero everywhere except for in one bin, and so the convolutional activations are constant everywhere except for around the stimulus. The constant offset is from a bias term that each convolutional channel has.

- What is the width of the non-constant activations (i.e. how many units in a given channel would differ from the constant)?

  The width of non-constant activations is the kernel size K.

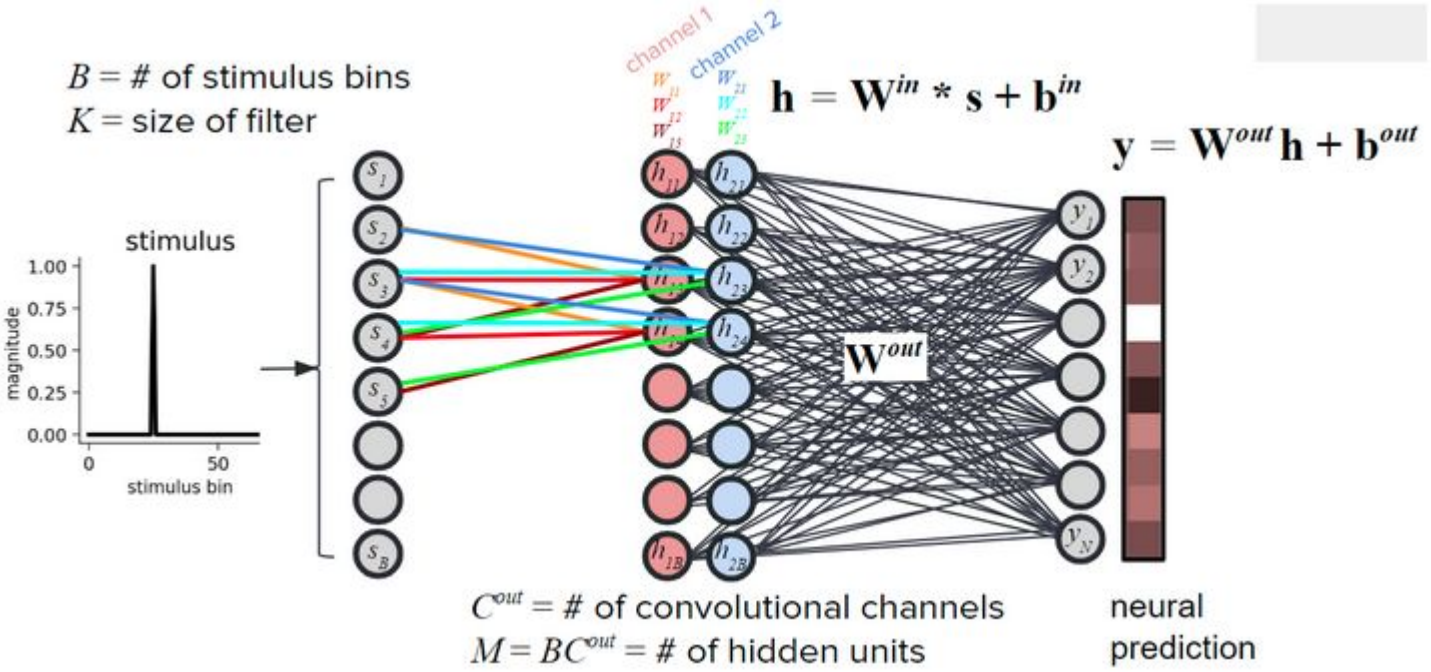
- How many weights does this convLayer have?

  The convLayer has K*C_out weights.

- How many would it have if it were a fully connected layer?

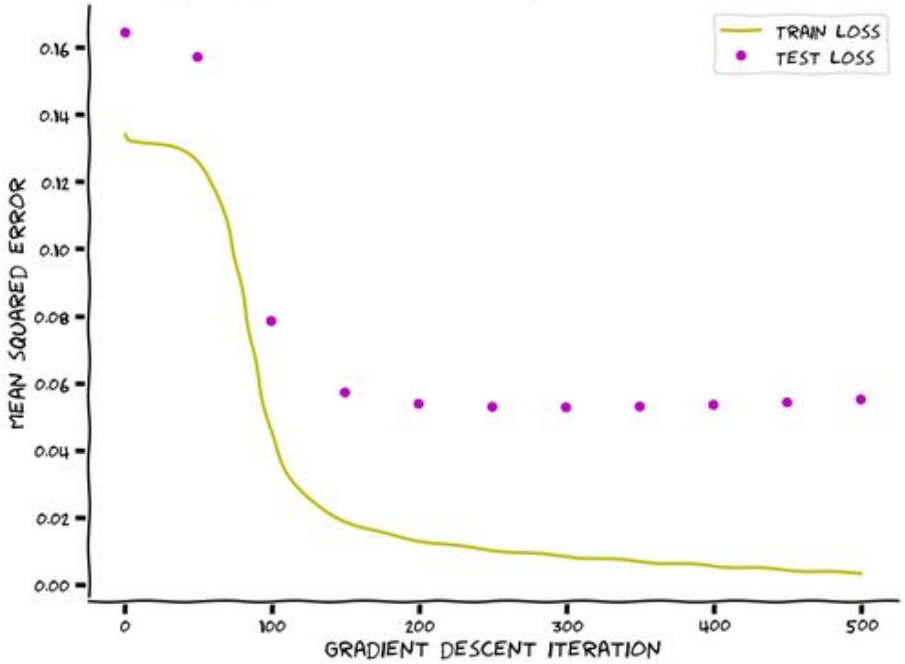
  A fully connected layer would have (B^2)*C_out weights.

Convolutional layer & fully connected layer



Implement encoding model

```
iteration 1/500 | train loss: 0.1343 | test loss: 0.1646
iteration 50/500 | train loss: 0.1263 | test loss: 0.1573
iteration 100/500 | train loss: 0.0455 | test loss: 0.0787
iteration 150/500 | train loss: 0.0191 | test loss: 0.0574
iteration 200/500 | train loss: 0.0133 | test loss: 0.0540
iteration 250/500 | train loss: 0.0105 | test loss: 0.0531
iteration 300/500 | train loss: 0.0086 | test loss: 0.0529
iteration 350/500 | train loss: 0.0071 | test loss: 0.0532
iteration 400/500 | train loss: 0.0057 | test loss: 0.0537
iteration 450/500 | train loss: 0.0045 | test loss: 0.0544
iteration 500/500 | train loss: 0.0034 | test loss: 0.0553
```



Observations

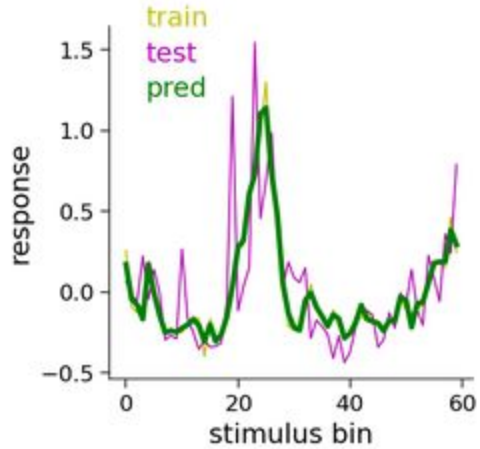
We trained this network to predict the neural responses. The training loss goes down throughout training but the testing loss doesn't -- We are overfitting to the NOISE in the training set.

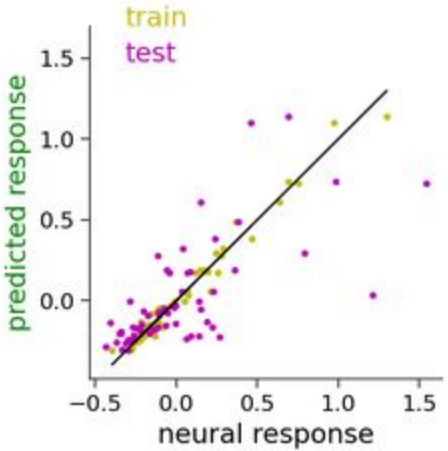
The yellow curve is the training data, the pink curve is the testing data and the prediction is in green - because the prediction has fit so well to the training data. However, some of what it has fit is noise.

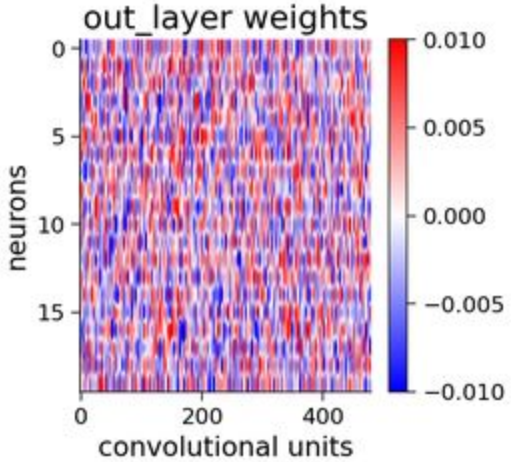
If we look at the weight matrix, we see that the weights are all positive or negative.

We see many sharp peaks in the training set, but these peaks are not always present in the test set, suggesting that they are due to noise. Therefore, ignoring this noise, we might have expected the neural responses to be a sum of only a few different filters and only at a few different positions. This would mean that we would expect the `out_layer` weight matrix to be sparse, not dense.

```
output shape:  torch.Size([60, 23589])
output weights shape:  torch.Size([23589, 480])
```







Food for thought

What does each dimension of the output and output weights correspond to?

The output is shape 60 by 23589 where 60 is the number of stimulus bins and 23589 is the number of predicted neurons. The out_layer weights have shape 23589 by 480 where 23589 is the number of predicted neurons and 480 is the number of convolutional units in the conv layer. There are 480 convolutional units because the stimulus is length 60 and we used 8 C^{out} convolutional channels.

Regularisation

If we think of a neuron as a sum of a few convolutional filters, we might expect the weight matrix of the fully-connected layer to be sparse. Therefore, we can also apply an L1 regularization penalty to enforce sparsity.

Tutorial #2 bonus Explanations

classic L_2 regularisation penalty R_{L_2}

Sum of squares of each weight in the network

$$\sum_{ij} W_{ij}^{out\,2}$$

L_1 regularisation penalty R_{L_1}

enforce sparsity of the weights

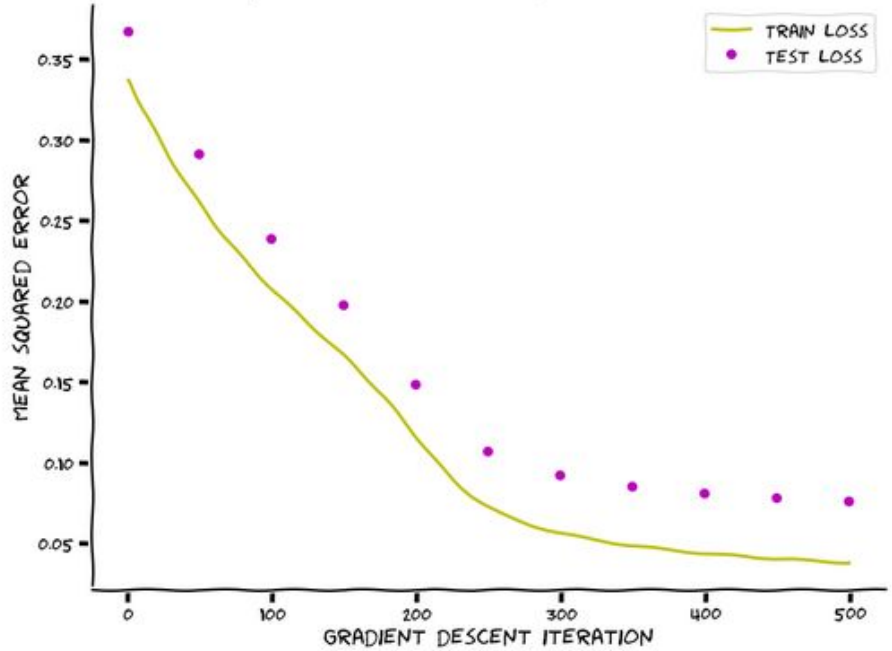
Sum of absolute values of the weights

$$\sum_{ij} |W_{ij}^{out}|$$

adding to loss function (inputs to transformer)

$$L = (y - \bar{y})^2 + R_{L_2} + R_{L_1}$$

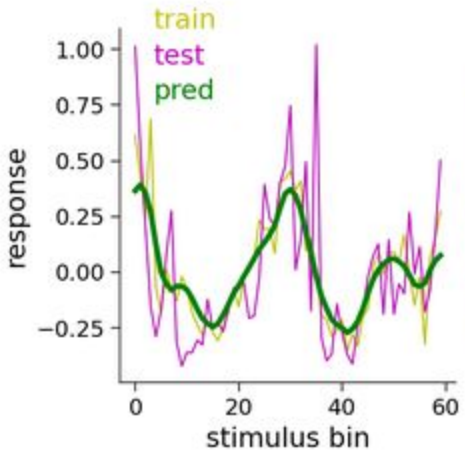
```
iteration 1/500 | train loss: 0.3374 | test loss: 0.3672
iteration 50/500 | train loss: 0.2613 | test loss: 0.2913
iteration 100/500 | train loss: 0.2083 | test loss: 0.2388
iteration 150/500 | train loss: 0.1668 | test loss: 0.1978
iteration 200/500 | train loss: 0.1168 | test loss: 0.1486
iteration 250/500 | train loss: 0.0730 | test loss: 0.1071
iteration 300/500 | train loss: 0.0568 | test loss: 0.0924
iteration 350/500 | train loss: 0.0490 | test loss: 0.0853
iteration 400/500 | train loss: 0.0441 | test loss: 0.0811
iteration 450/500 | train loss: 0.0408 | test loss: 0.0782
iteration 500/500 | train loss: 0.0383 | test loss: 0.0762
```

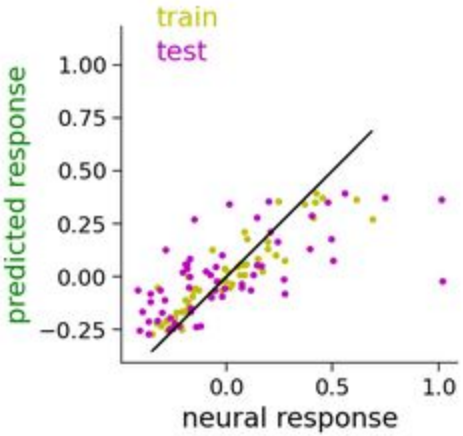


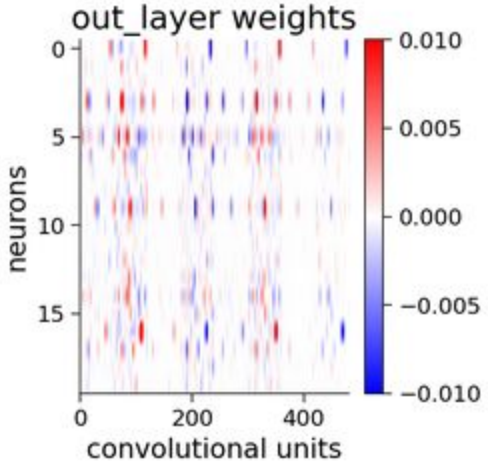
Observations

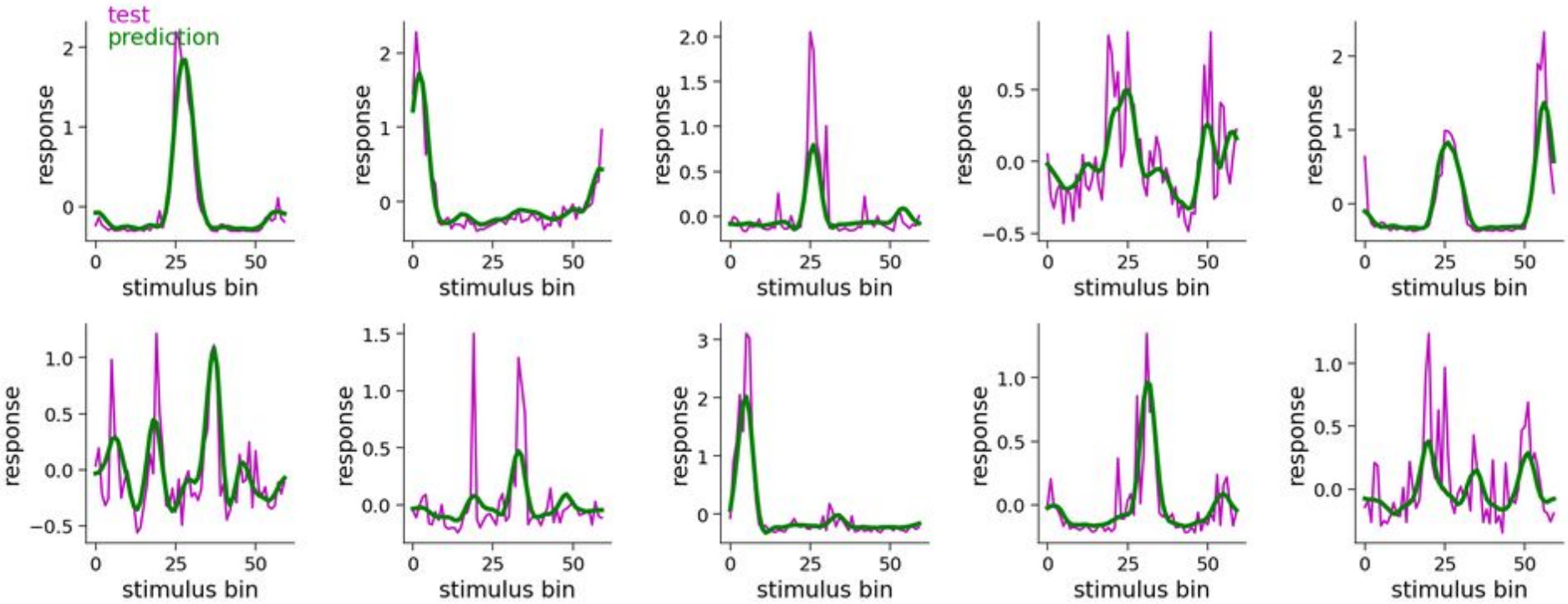
If we now train the network with these regularization penalties we find that the train and test loss are similar throughout training: both continue decreasing.

We can see below that the prediction is much smoother than before! This is because the weight matrix is in fact sparser


train
test
pred
response
stimulus bin


train
test
predicted response
neural response


out_layer weights
neurons
convolutional units


test
prediction
response
stimulus bin

Summary

In this notebook, we built and evaluated a neural network based encoding model to predict neural responses from stimuli:

- implemented a basic convolution filter
- implemented and trained a convolutional neural network with multiple filters to predict neural responses using PyTorch
- learned about and implemented L2/L1 regularization to avoid overfitting

What does this mean?

What can this tell us about the representation of oriented gratings in mouse visual cortex? Maybe we can think of interpreting each of the convolutional channels as a computation performed by a single group of neurons in thalamus, and each visual cortical neuron combines various groups of thalamic neurons. But we'd have to test hypotheses like these by, for instance, recording thalamic neurons.

Why CNN's? #1

CNN models are particularly well-suited to modeling the visual system for a number of reasons:

1. *Distributed computation: like any other neural network, CNN's use distributed representations to compute -- much like the brain seems to do. Such models, therefore, provide us with a vocabulary with which to talk and think about such distributed representations. Because we know the exact function the model is built to perform (e.g. orientation discrimination), we can analyze its internal representations with respect to this function and begin to interpret why the representations look the way they do. Most importantly, we can then use these insights to analyze the structure of neural representations observed in recorded population activity. We can qualitatively and quantitatively compare the representations we see in the model and in a given population of real neurons to hopefully tease out the computations it performs.*

Why CNN's? #2

2. Hierarchical architecture: like in any other deep learning architecture, each layer of a deep CNN comprises a non-linear transformation of the previous layer. Thus, there is a natural hierarchy whereby layers closer to the network output represent increasingly more abstract information about the input image. For example, in a network trained to do object recognition, the early layers might represent information about edges in the image, whereas later layers closer to the output might represent various object categories. This resembles the hierarchical structure of the visual system, where lower-level areas (e.g. retina, V1) represent visual features of the sensory input and higher-level areas (e.g. V4, IT) represent properties of objects in the visual scene. We can then naturally use a single CNN to model multiple visual areas, using early CNN layers to model lower-level visual areas and late CNN layers to model higher-level visual areas.

Relative to fully connected networks, CNN's, in fact, have further hierarchical structure built-in through the max pooling layers. Recall that each output of a convolution + pooling block is the result of processing a local patch of the inputs to that block. If we stack such blocks in a sequence, then the outputs of each block will be sensitive to increasingly larger regions of the initial raw input to the network: an output from the first block is sensitive to a single patch of these inputs, corresponding to its "receptive field"; an output from the second block is sensitive to a patch of outputs from the first block, which together are sensitive to a larger patch of raw inputs comprising the union of their receptive fields. Receptive fields thus get larger for deeper layers (see here for a nice visual depiction of this). This resembles primate visual systems, where neurons in higher-level visual areas respond to stimuli in wider regions of the visual field than neurons in lower-level visual areas.

Why CNN's? #3

3. Convolutional layers: through the weight sharing constraint, the outputs of each channel of a convolutional layer process different parts of the input image in exactly the same way. This architectural constraint effectively builds into the network the assumption that objects in the world typically look the same regardless of where they are in space. This is useful for modeling the visual system.

Why CNN's? #3a and #3b

- *Firstly, this assumption is generally valid in mammalian visual systems, since mammals tend to view the same object from many perspectives. Two neurons at a similar hierarchy in the visual system with different receptive fields could thus end up receiving statistically similar synaptic inputs, so that the synaptic weights developed over time may end up being similar as well.*
- *Secondly, this architecture significantly improves object recognition ability. Object recognition was essentially an unsolved problem in machine learning until the advent of techniques for effectively training deep convolutional neural networks. Fully connected networks on their own can't achieve object recognition abilities anywhere close to human levels, making them bad models of human object recognition. Indeed, it is generally the case that the better a neural network model is at object recognition, the closer the match between its representations and those observed in the brain. That said, it is worth noting that our much simpler orientation discrimination task can be solved by relatively simple networks.*

Tutorial #3
Explanations

Objective

In this tutorial, we'll be using deep learning to build an encoding model of the visual system, and then compare its internal representations to those observed in neural data.

Importantly, the encoding model is different from the encoding models. Its parameters won't be directly optimized to fit the neural data. Instead, we will optimize its parameters to solve a particular visual task that we know the brain can solve. We therefore refer to it as a "normative" encoding model, since it is optimized for a specific behavioral task.

To then evaluate whether this normative encoding model is actually a good model of the brain, we'll analyze its internal representations and compare them to the representations observed in mouse primary visual cortex. Since we understand exactly what the encoding model's representations are optimized to do, any similarities will hopefully shed light on why the representations in the brain look the way they do.

- Visualize and analyze the internal representations of a deep network
- Quantify the similarity between distributed representations in a model and neural representations observed in recordings, using Representational Similarity Analysis (RSA)

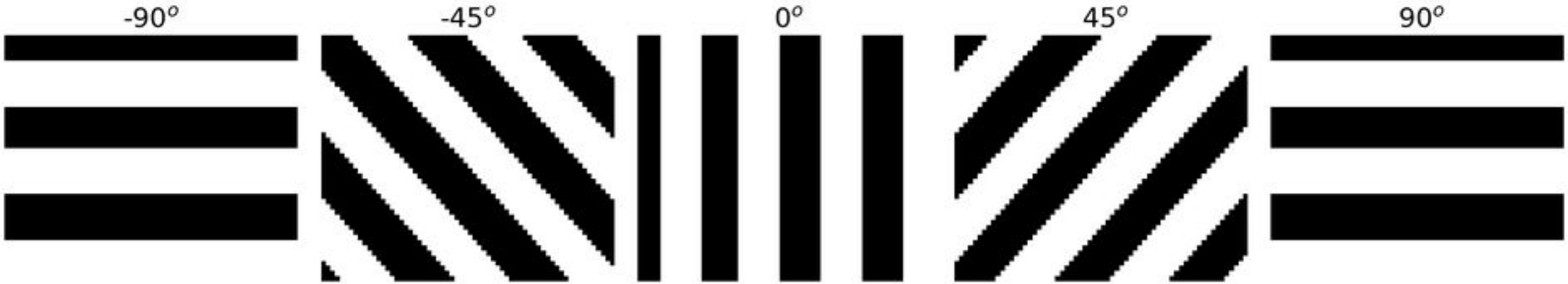
Orientation discrimination task

We will build our normative encoding model by optimizing its parameters to solve an orientation discrimination task.

Task is to tell whether a given grating stimulus is tilted to the "right" or "left"; that is, whether its angle relative to the vertical is positive or negative, respectively.

Note that this is a task that we know many mammalian visual systems are capable of solving. It is therefore conceivable that the representations in a deep network model optimized for this task might resemble those in the brain. To test this hypothesis, we will compare the representations of our optimized encoding model to neural activity recorded in response to these very same stimuli, courtesy of Stringer et al 2019.

stimulus size: 48 x 64



A deep network model of orientation discrimination

Our goal is to build a model that solves the orientation discrimination task outlined above. The model should take as input a stimulus image and output the probability of that stimulus being tilted right.

Use a **convolutional neural network (CNN)** that performs two-dimensional convolutions on the raw stimulus image (which is a 2D matrix of pixels), rather than one-dimensional convolutions on a categorical 1D vector representation of the stimulus. CNNs are commonly used for image processing.

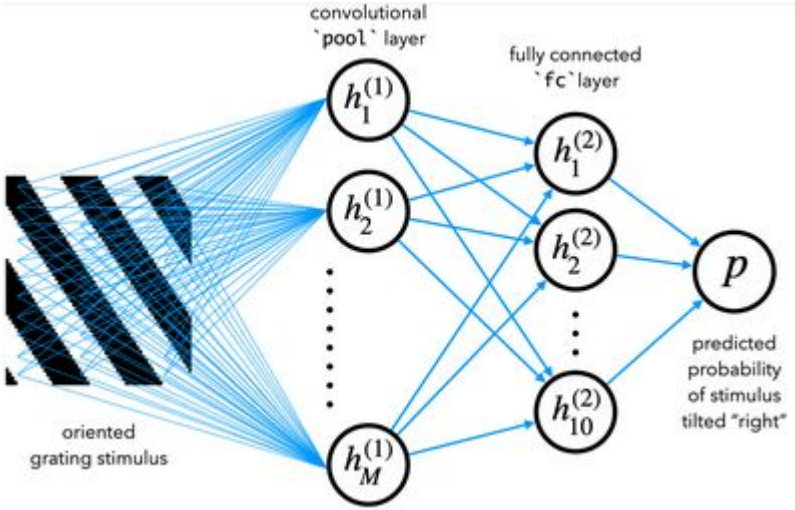
The particular CNN we will use here has two layers:

1. a convolutional layer, which convolves the images with a set of filters
2. a fully connected layer, which transforms the output of this convolution into a 10-dimensional representation

Finally, a set of output weights transforms this 10-dimensional representation into a single scalar p, denoting the predicted probability of the input stimulus being tilted right.

Training network and analyzing its internal representations.

After initializing our CNN model, it builds a dataset of oriented grating stimuli to use for training it. These are then passed into a function called train() that uses SGD to optimize the model's parameters,
taking similar arguments as the train()
function



Comparing CNNs to neural activity

Analyze the internal representations of our deep CNN model of orientation discrimination and qualitatively compare them to population responses in mouse primary visual cortex.

Load data

Loading in some data from this paper, which contains the responses of about ~20,000 neurons in mouse primary visual cortex to grating stimuli like those used to train our network. These data are stored in two variables:

- resp_v1 is a matrix where each row contains the responses of all neurons to a single stimulus.
- ori is a vector with the orientations of each stimulus, in degrees. As in the above convention, negative angles denote stimuli tilted to the left and positive angles denote stimuli tilted to the right.

We will then extract our deep CNN model's representations of these same stimuli (i.e. oriented gratings with the orientations in ori). We will run the same stimuli through our CNN model and use the helper function get_hidden_activity() to store the model's internal representations. The output of this function is a Python dict, which contains a matrix of population responses (just like resp_v1) for each layer of the network specified by the layer_labels argument. We'll focus on looking at the representations in

- the output of the first convolutional layer, stored in the model as 'pool' (see the appendix for the details of the CNN architecture to understand why it's called this way)
- the 10-dimensional output of the fully connected layer, stored in the model as 'fc'

Representational Similarity Analysis (RSA)

Similarities and differences between the population responses in mouse primary visual cortex and in different layers in our model.

Use a quantification technique called Representational Similarity Analysis. The idea is to look at the similarity structure between representations of different stimuli. We can say that a brain area and a model use a similar representational scheme if stimuli that are represented (dis)similarly in the brain are represented (dis)similarly in the model as well.

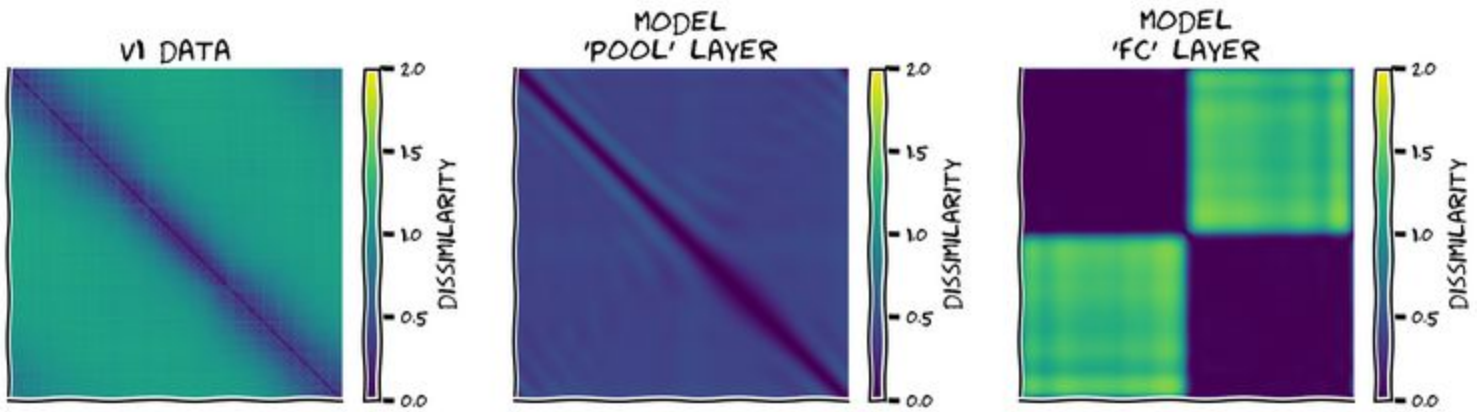
Representational dissimilarity matrix

$$M = 1 - \frac{1}{N} ZZ^T$$

correlation coefficient
two population
responses to each
stimulus.

Z scored responses.

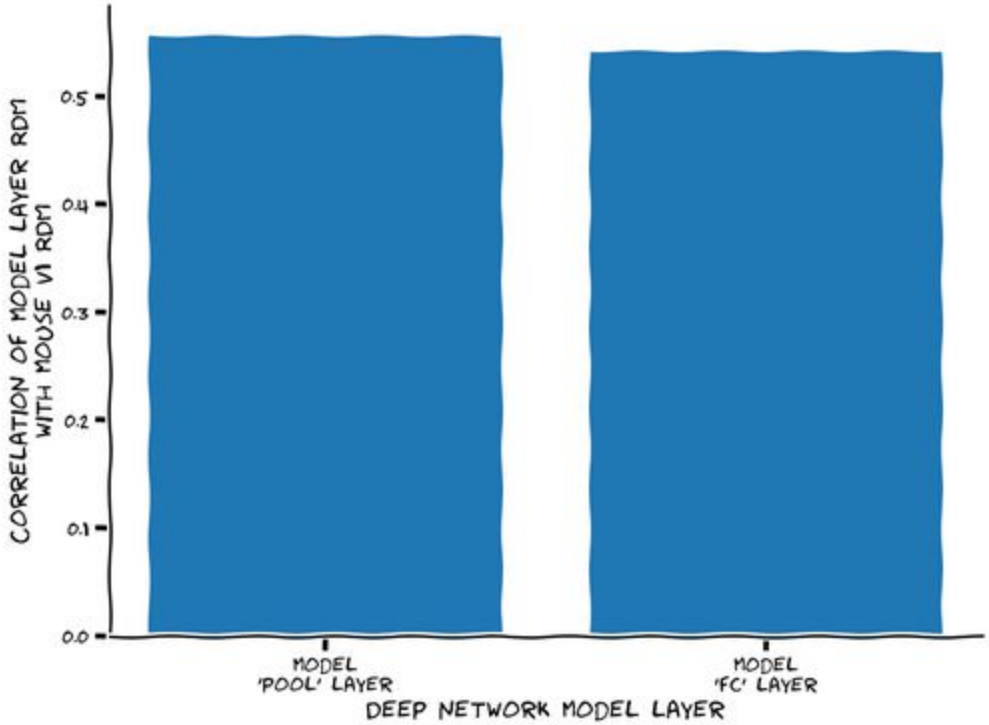
no of neural units


V1 DATA
2.0
1.5
1.0
0.5
0.0
DISSIMILARITY
MODEL
'POOL' LAYER
2.0
1.5
1.0
0.5
0.0
DISSIMILARITY
MODEL
'FC' LAYER
2.0
1.5
1.0
0.5
0.0
DISSIMILARITY

Tutorial #3 bonus Explanations

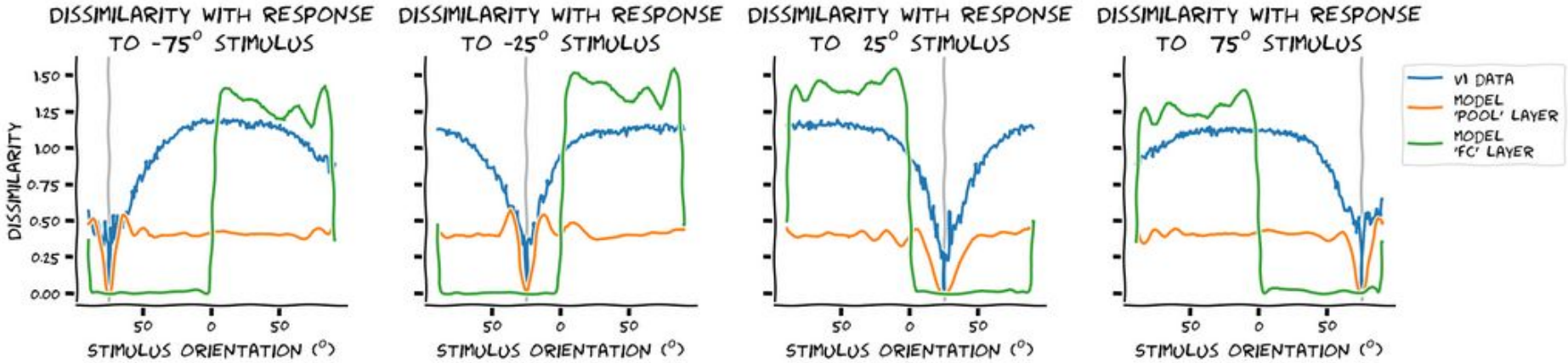
Correlate RDMs

To quantify how similar the representations are, we can simply correlate their dissimilarity matrices. For this, we'll again use the correlation coefficient. Note that dissimilarity matrices are symmetric ($M_{ss'} = M_{s's}$), so we should only use the off-diagonal terms on one side of the diagonal when computing this correlation to avoid overcounting. Moreover, we should leave out the diagonal terms, which are always equal to 0, so will always be perfectly correlated across any pair of RDM's.


CORRELATION OF MODEL LAYER RDM WITH MOUSE V1 RDM
0.0
0.1
0.2
0.3
0.4
0.5
MODEL 'POOL' LAYER
MODEL 'FC' LAYER
DEEP NETWORK MODEL LAYER

Plot rows of RDM

To better understand how these correlations in RDM's arise, plot individual rows of the RDM matrix. The resulting curves show the similarity of the responses to each stimulus with that to one specific stimulus.


DISSIMILARITY WITH RESPONSE TO -75° STIMULUS
DISSIMILARITY WITH RESPONSE TO -25° STIMULUS
DISSIMILARITY WITH RESPONSE TO 25° STIMULUS
DISSIMILARITY WITH RESPONSE TO 75° STIMULUS
DISSIMILARITY
1.50
1.25
1.00
0.75
0.50
0.25
0.00
50
0
50
STIMULUS ORIENTATION (°)
V1 DATA
MODEL 'POOL' LAYER
MODEL 'FC' LAYER

Qualitative comparisons of CNNs and neural activity

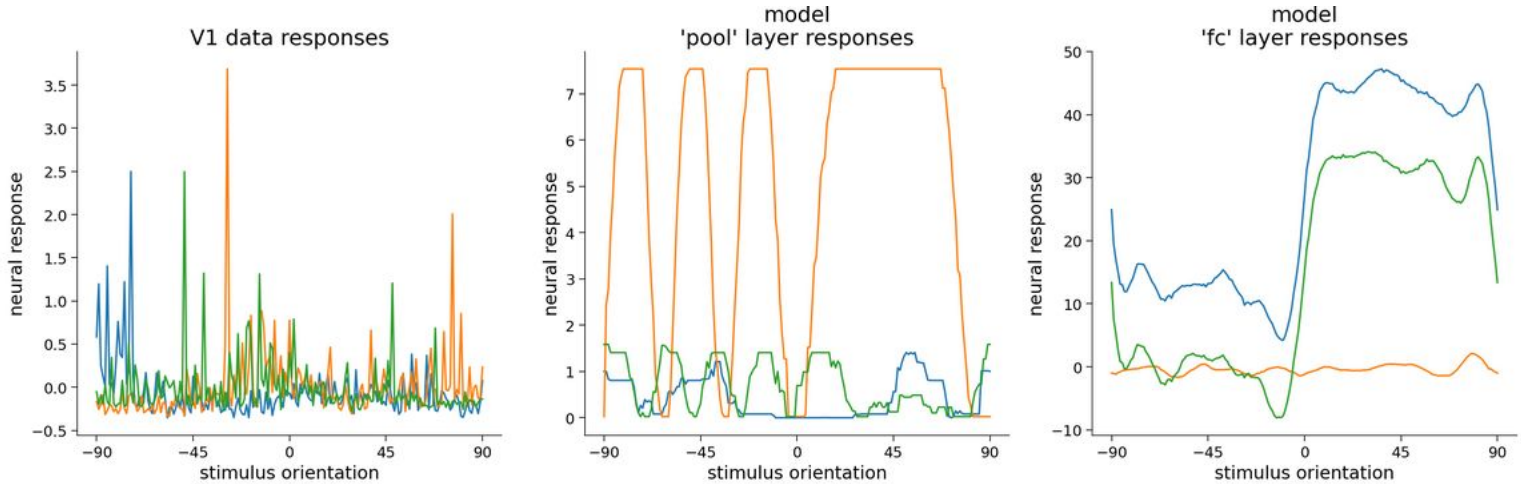
To visualize the representations in the data and in each of these model layers, we'll use two classic techniques from systems neuroscience:

1. **tuning curves**: plotting the response of single neurons (or units, in the case of the deep network) as a function of the stimulus orientation

2. **dimensionality reduction**: plotting full population responses to each stimulus in two dimensions via dimensionality reduction. Use the non-linear dimensionality reduction technique t-SNE for this.

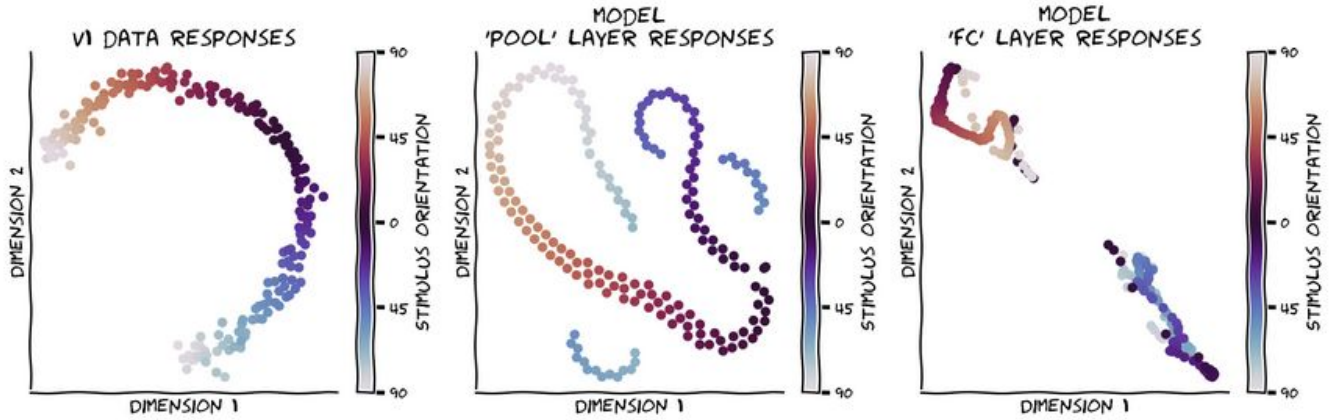
Tuning curves

Tuning curves for different neurons and units in the CNN: to get an idea of shared properties in the tuning curves of the neurons within each population.



Dimensionality reduction of representations

We can visualize a dimensionality-reduced version of the internal representations of the mouse primary visual cortex or CNN internal representations in order to potentially uncover informative structure. Use PCA to reduce the dimensionality to 20 dimensions, and then use tSNE to further reduce dimensionality to 2 dimensions. (Use the first step of PCA so that tSNE runs faster)



Observations #1

Why do these representations look the way they do?

How are the population responses similar / different between the model and the data? Can you explain these population-level responses from the single neuron responses seen in the previous exercise, or vice-versa?

The single unit activations of the 'pool' layer in the model have peaks at various orientations, but they appear to have more peaks than the tuning curves from the original data do. When we look at the population level responses we see that they are not quite as smooth across orientations in the t-SNE embedding, which is likely due to the fact that the 'pool' layer activations do not have localized responses to orientations like the neural data.

Observations #2

- How do the representations in the different layers of the model differ, and how does this relate to the orientation discrimination task the model was optimized for?

  The representations in the fully-connected 'fc' layer appear to be much more clustered than the 'pool' layer. Stimuli which correspond to the same choice are clustered together. It seems like the 'pool' layer is still working hard to represent information about the orientation of the stimulus (much like the V1 population), whereas the 'fc' layer only cares about tilt category, representing all the stimuli with the same category in a similar way regardless of their different orientations.

Observations #3

Which layer of our deep network encoding model most closely resembles the V1 data?

From this analysis, it appears that the 'pool' layer is more similar to the neural data at the population level.

Summary

- use deep learning to build a normative encoding model of the visual system
- use RSA to evaluate how the model's representations match to those in the brain

Our approach was to optimize a deep convolutional network to solve an orientation discrimination task.

Firstly, there are many other "normative" ways to solve this orientation discrimination task. We could have used different neural network architectures, or even used a completely different algorithm that didn't involve a neural network at all, but instead used other kinds of image transformations (e.g. Fourier transforms). Neural network approaches, however, are special in that they explicitly uses abstract distributed representations to compute, which feels a lot closer to the kinds of algorithms the brain uses.

Secondly, our choice of visual task was mostly arbitrary. For example, we could have trained our network to directly estimate the orientation of the stimulus, rather than just discriminating between two classes of tilt. Or, we could have trained the network to perform a more naturalistic task, such as recognizing the rotation of an arbitrary image. Or we could try a task like object recognition.

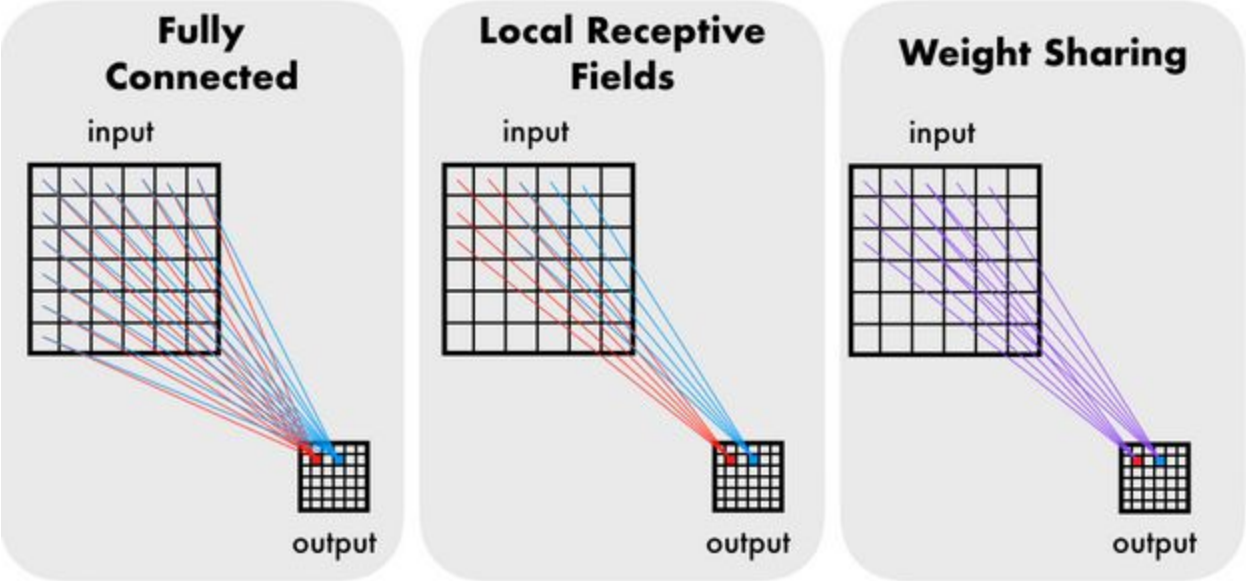
Training on different tasks could lead to different representations of the oriented grating stimuli, which might match the observed V1 representations better or worse.

Appendix

Convolutional Neural Networks (CNN's)

Convolutional layers are different from their fully connected counterparts in two ways (see figure below):

- In a fully connected layer, each unit computes a weighted sum over all the input units. In a convolutional layer, on the other hand, each unit computes a weighted sum over only a small patch of the input, referred to as the unit's **receptive field**. When the input is an image, the receptive field can be thought of as a local patch of pixels.
- In a fully connected layer, each unit uses its own independent set of weights to compute the weighted sum. In a convolutional layer, all the units (within the same channel) **share the same weights**. This set of shared weights is called the **convolutional filter or kernel**. The result of this computation is a convolution, where each unit has computed the same weighted sum over a different part of the input.


Fully Connected
input
output
Local Receptive Fields
input
output
Weight Sharing
input
output

Fully connected layers

In a fully connected layer, each unit computes a weighted sum over all the input units and applies a non-linear function to this weighted sum.

Specifically, its output is the predicted probability of the input image being tilted right. To ensure that its output is a probability (i.e. a number between 0 and 1), we use a sigmoid activation function to squash the output into this range (implemented with torch.sigmoid()).

Convolutional layers

In a convolutional layer, each unit computes a weighted sum over a two-dimensional $K \times K$ patch of inputs. Units are arranged in **channels**, whereby units in the same channel compute the same weighted sum over different parts of the input, using the weights of that channel's **convolutional filter (or kernel)**. The output of a convolutional layer is thus a three-dimensional tensor of shape

$C_{out} \times H \times W$, where

C_{out} is the number of channels (i.e. the number of convolutional filters/kernels), and

H and W are the height and width of the input.

CONVOLUTIONAL LAYER

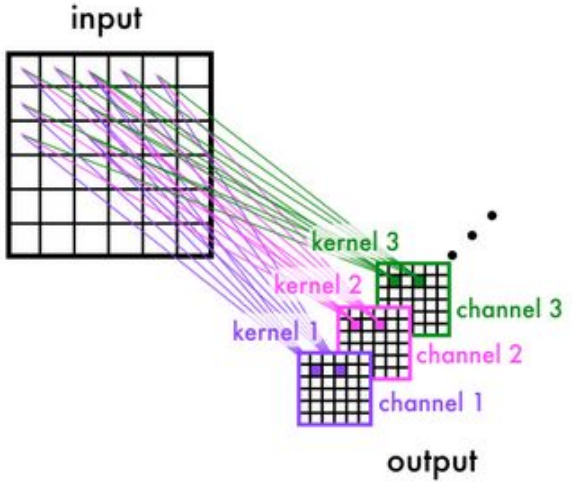
Code incorporating a convolutional layer with 8 convolutional filters of size 5 x 5 into our above fully connected network. Note that we have to flatten the multi-channel output in order to pass it on to the fully connected layer.

***Note:** as is also the case for the nn.Conv1d class, the inputs to nn.Conv2d layers must have a channel dimension in their first dimension. Thus, the input to a nn.Conv2d layer must be a 3D tensor of shape Cin x H x W*

where

Cin is the number of input channels and

H, W their height and width, respectively.



Max pool layer

In a max pooling layer, each unit computes the maximum over a small two-dimensional Kpool × Kpool patch of inputs. Given a multi-channel input of dimensions C×H×W, the output of a max pooling layer has dimensions C×Hout×Wout. Max pooling layers can be implemented with the PyTorch nn.MaxPool2d class, which takes as a single argument the size Kpool of the pooling patch. We need to calculate the dimensions of its output in order to set the dimensions of the subsequent fully connected layer.

Max pool layers

rounding down

$$H^{out} = \left\lfloor \frac{H}{K^{pool}} \right\rfloor$$

$$W^{out} = \left\lfloor \frac{W}{K^{pool}} \right\rfloor$$

Conv + pool

This pooling layer completes the CNN model trained above to perform orientation discrimination. We can think of this architecture as having two primary layers:

1. *a convolutional + pooling layer*
2. *a fully connected layer*

We group together the convolution and pooling layers into one, as they really form one full unit of convolutional processing, where each patch of the image is passed through a convolutional filter and pooled with neighboring patches. It is standard practice to follow up any convolutional layer with a pooling layer, so they are generally treated as a single block of processing.

Orientation discrimination as a binary classification problem

What loss function should we minimize to optimize orientation discrimination performance? We first note that the orientation discrimination task is a **binary classification problem**, where the goal is to classify a given stimulus into one of two classes: being tilted left or being tilted right.

Our goal is thus to output a high probability of the stimulus being tilted right (i.e. large p) whenever the stimulus is tilted right, and a high probability of the stimulus being tilted left (i.e. large $1-p \Leftrightarrow$ small p) whenever the stimulus is tilted left.

goal: output high probability of stimulus being tilted (large p) in a particular direction (right/left) whenever the stimulus is tilted in a particular direction.

$y^{(n)}$: label of the nth stimulus in a mini-batch (indicating the tilt)

$$= \begin{cases} 1 & \text{if stim n is tilted right} \\ 0 & \text{---''---} \quad \text{left} \end{cases}$$

$$\log(\text{predicted probability of stimulus } n \text{ being in class } \bar{y}^{(n)}) = \begin{cases} \log \bar{p}^{(n)} & \text{if } \bar{y}^{(n)} = 1 \\ \log(1-\bar{p}^{(n)}) & \text{if } \bar{y}^{(n)} = 0 \end{cases}$$

$$= \bar{y}^{(n)} \log \bar{p}^{(n)} + (1 - \bar{y}^{(n)}) \log (1 - \bar{p}^{(n)})$$

predicted probability of stimulus being tilted right

predicted probability of stimulus being tilted left

mean predicted probability of true class $\bar{y}^{(n)}$.

Log likelihood

Log likelihood of the Bernoulli distribution under the predicted probability $p(n)$. This is the same quantity that is maximized in logistic regression, where the predicted probability $p(n)$ is just a simple linear sum of its inputs (rather than a complicated non-linear operation).

log probability $\xrightarrow{*(-1)}$ loss function

(negative log likelihood / binary cross entropy)

Binary cross entropy loss:

$$L = -\sum_{n=1}^{P} y^{(n)} \log p^{(n)} + \left(1 - y^{(n)}\right) \log\left(1 - p^{(n)}\right)$$

Sum over P samples in a batch

Binary cross entropy

The binary cross-entropy loss can be implemented in PyTorch using the nn.BCELoss() loss function (cf. documentation).

Because the CNN's used here have lots of parameters, we have to use two tricks:

1. We have to use stochastic gradient descent (SGD), rather than just gradient descent (GD).
2. We have to use momentum in our SGD updates. This is easily incorporated into our PyTorch implementation by just setting the momentum argument of the built-in optim.SGD optimizer.

RDM Z-score explanation

response of i-th neuron to s-th stimulus

$$M_{ss'} = 1 - \frac{\mathrm{cov}\left[x_i^{(s)}, x_i^{(s')}\right]}{\sqrt{\mathrm{var}\left[x_i^{(s)}\right]\mathrm{var}\left[x_i^{(s')}\right]}}$$

$$= 1 - \frac{\sum_{i=1}^{N}\left(x_i^{(s)} - \bar{x}^{(s)}\right)\left(x_i^{(s')} - \bar{x}^{(s')}\right)}{\sqrt{\sum_{i=1}^{N}\left(x_i^{(s)} - \bar{x}^{(s)}\right)^2 \sum_{i=1}^{N}\left(x_i^{(s')} - \bar{x}^{(s')}\right)^2}}$$

$$\bar{r}^{(s)} = \frac{1}{N} \sum_{i=1}^{N} r_i^{(s)}$$

efficiently computed using z scored reference

$$z_i^{(s)} = \frac{r_i^{(s)} - \bar{r}^{(s)}}{\sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(r_i^{(s)} - \bar{r}^{(s)} \right)^2}}$$

$$\Rightarrow M_{ss'} = 1 - \frac{1}{N} \sum_{i=1}^{N} z_i^{(s)} z_i^{(s')}$$

Such that matrix is computed through matrix multiplication

$$M = I - \frac{1}{N} ZZ^T$$

$$Z = \begin{bmatrix} z_1^{(1)} & z_2^{(1)} & \cdots & z_N^{(1)} \\ z_1^{(2)} & z_2^{(2)} & \cdots & z_N^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ z_1^{(S)} & z_2^{(S)} & \cdots & z_N^{(S)} \end{bmatrix}$$

total no of stimuli

$S \times N$

$M : S \times S$