

S&DS 365 / 665
Intermediate Machine Learning

Convolutional Neural Networks

February 14

Yale

Reminders

- Assignment 1 is out, due a week from Wednesday
- Quiz 1 this Wednesday; 15 minutes, online, 24 hours
- Any material covered in class (up to and including today)
- OH schedule posted to Canvas/EdD
- Questions or concerns?

Topics for today

- Mechanics of convolutional networks
- Filters and pooling and flattening (oh my!)
- Example: Classifying Ca²⁺ brain scans
- Jumpstarting Problem 4 on Assn 1
- Other examples

Notes

- Animations from tutorial “Let’s Code Convolutional Neural Network in plain NumPy: Mysteries of Neural Networks” by Piotr Skalski
- [https://towardsdatascience.com/
lets-code-convolutional-neural-network-in-plain-numpy-ce48e732f5d5](https://towardsdatascience.com/lets-code-convolutional-neural-network-in-plain-numpy-ce48e732f5d5)
- To see the animations, view this pdf using Adobe Acrobat

Convolution

Mathematically, a *convolution* sweeps a function $K(u)$ over the values of another function f :

Continuous:

$$K * f(x) = \int K(x - u) f(u) du$$

- Same operation we saw with smoothing kernels
- If K is a Gaussian bump, this blurs or smooths out the values.
- For CNNs, K will have small support, encode features of the input

Convolution

Mathematically, a *convolution* sweeps a function $K(u)$ over the values of another function f :

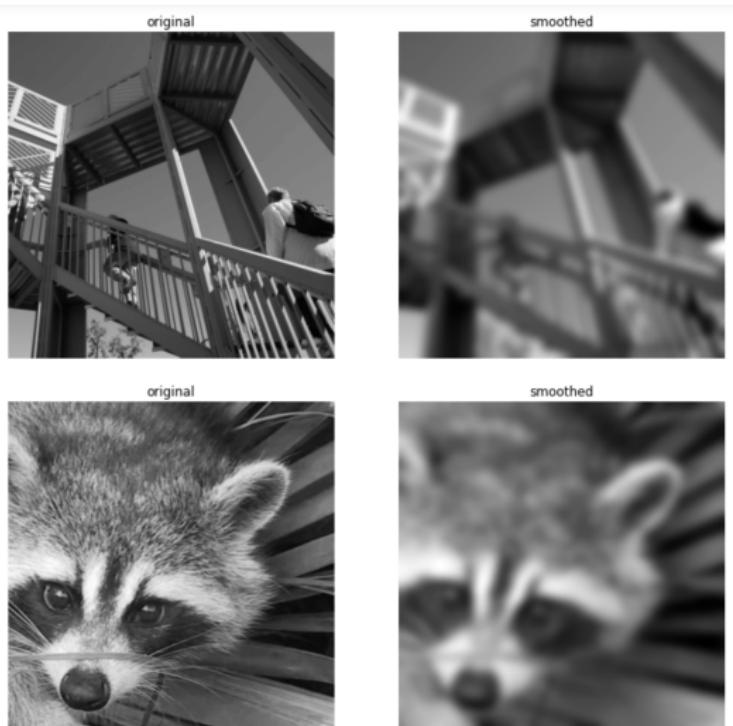
Discrete:

$$K * f(x_j) = \sum_i K(x_j - x_i) f(x_i)$$

- Same operation we saw with smoothing kernels
- If K is a Gaussian bump, this blurs or smooths out the values.
- For CNNs, K will have small support, encode features of the input

Examples

convolve_demo.ipynb gives examples



Two principles

Two of the ideas behind convolutional neural networks (CNNs):

- ① Parameter tying: Let neurons share the same weights
- ② Kernel learning: The kernels should not be fixed, but rather *learned* from the data

CNNs are most commonly used for images; can be used whenever data have some kind of spatial structure (e.g. DNA)

Tensors

A tensor is a multidimensional array*—like a matrix, but with more than just a pair of indices for row and columns.

In Python, tensors are represented in `numpy` using the type `numpy.ndarray` (“*n*-dimensional array”)

TensorFlow also has built-in types to represent tensors

* In mathematics, there is a little more structure to tensors.

Tensors

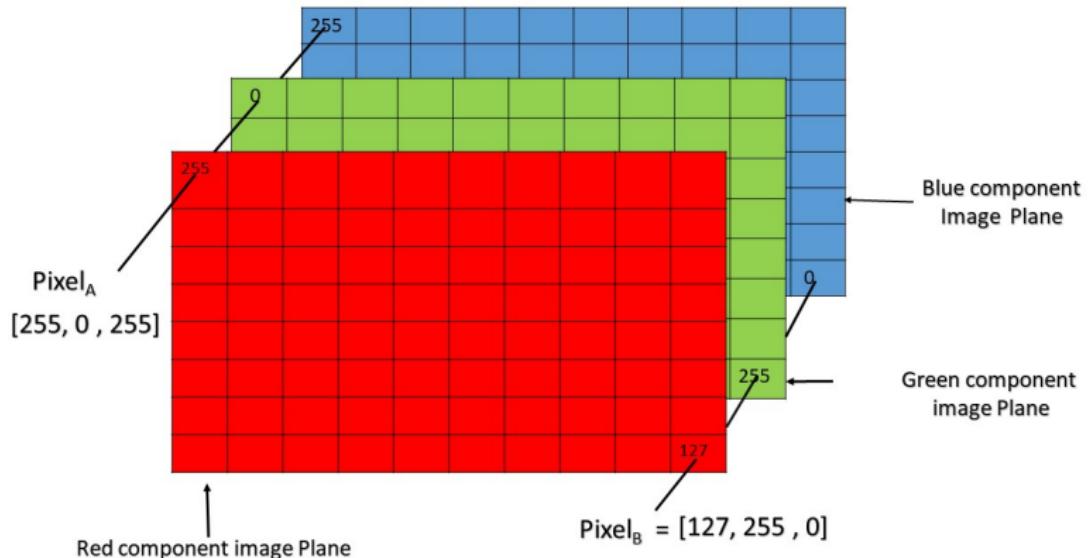
```
x = np.array(np.arange(24)).reshape(2, 3, 4)
print(x)
print(x[:, :, 0])

[[[ 0   1   2   3]
 [ 4   5   6   7]
 [ 8   9  10  11]]

 [[12  13  14  15]
 [16  17  18  19]
 [20  21  22  23]]]

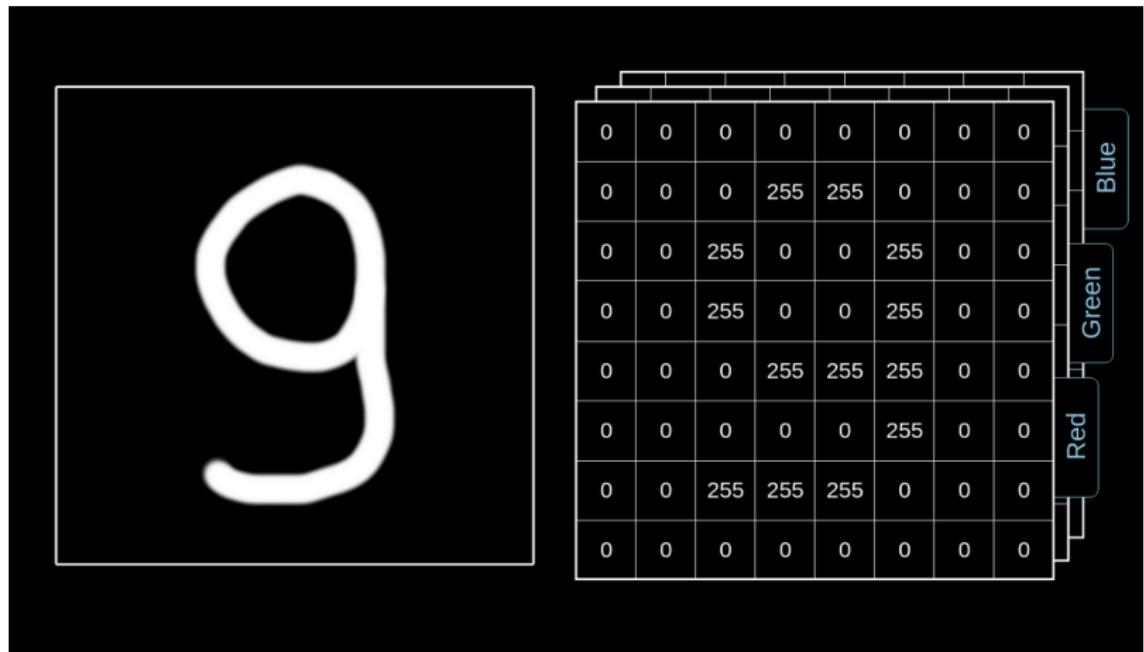
[[ 0   4   8]
 [12  16  20]]
```

Images as tensors: Channels



Pixel of an RGB image are formed from the corresponding pixel of the three component images

Images as tensors: Channels



Convolutional layer

- In a convolutional layer, a set of *kernels* are “swept over” or convolved over the input
- The kernels are themselves tensors with entries that are trained weights
- The result of convolving the kernel over the input tensor is called a *feature map*
- A convolutional layer is equivalent to a standard fully connected layer, but where the weights are constrained to be sparse and “tied” (constrained to be equal) across different neurons

Convolutional layer

Adding a nonlinearity

- So far everything is linear
- An activation function is used to add a nonlinearity
- Each filter also has an intercept (a single number), added before applying the activation function
- The same activation function is used for all of the kernels in a layer

Pooling

- The point of a convolutional layer is to learn “features” of the input
- We don’t care too much exactly where they appear
- *Pooling* the feature map reduces the dimension and encodes the approximate location of features
- Two common types: max pooling and average pooling
- When using max pooling, to update weights in backpropagation, the locations of the features can be noted

Pooling

How features are built up

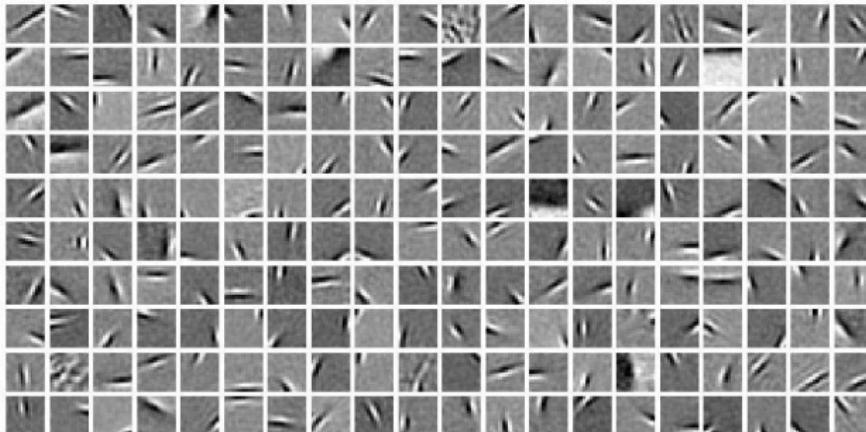
- Each kernel has a weight for each spatial position and channel
- Features from previous layers are recorded in the channels
- So the next layer can have a kernel that says

“Oh, I see that feature 17 from the previous layer is active in position (1,1), feature 42 is in position (1,2), feature 5 is in position (2,1), and feature 11 is in position (2,2)”

- This enables compound features to be built up

How features are built up

- Filters in the first layer may look like this:



- Filters in second layer can then capture more complex shapes

Pooling and backpropagation

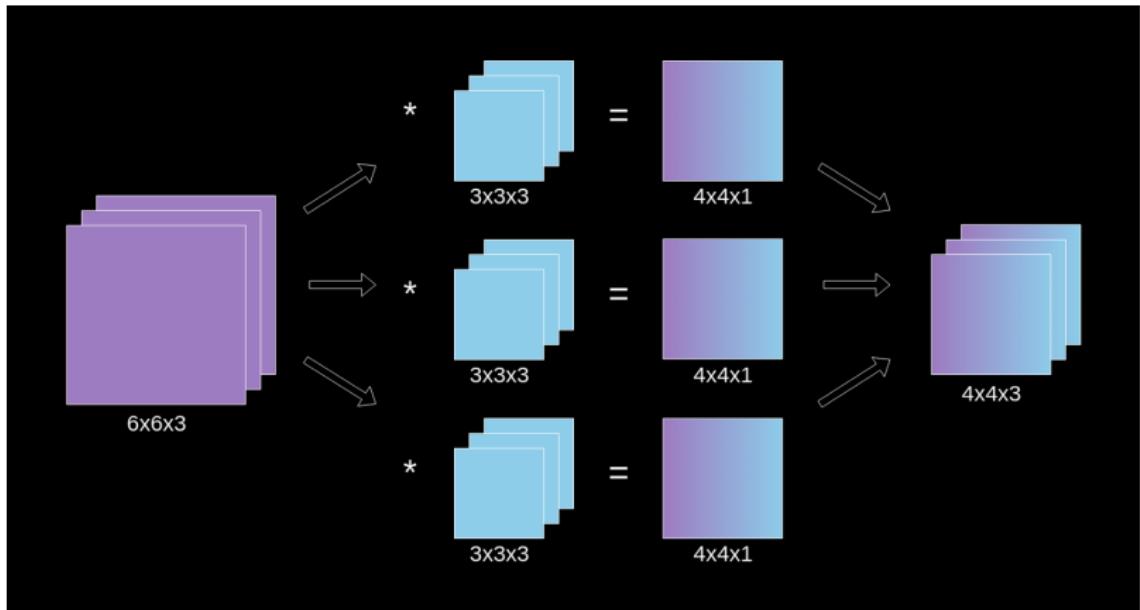
- A forward pass through the layers is used to make predictions, and store intermediate results to be used later
- A backward pass propagates the errors in prediction back through the layers, and computes changes to the weights for the gradient step

Forward and backward passes

Applying kernels across channels

- Typically each kernel (filter) has the same number of channels as the input tensor
- A kernel is applied at each location by multiplying component-wise and then summing
- The feature map for each kernel is then one channel in the output tensor
- So, the number of channels in the output is the number of kernels

Applying kernels across channels

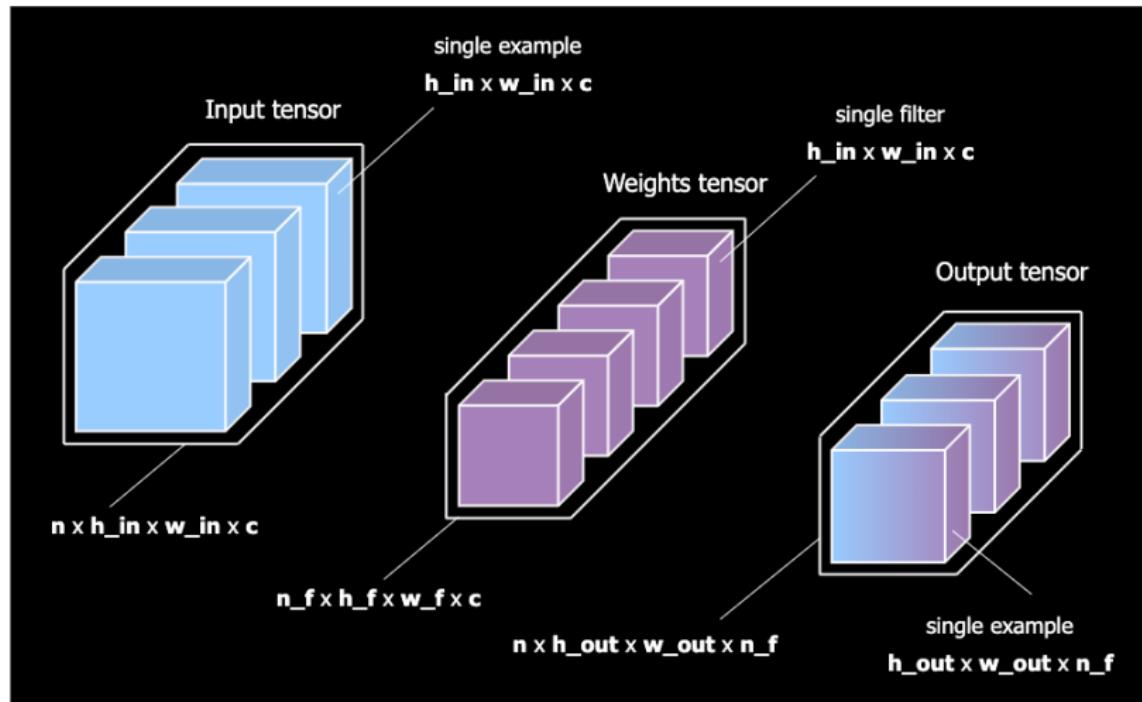


Convolutional layer: Tensor shapes

- Data points are processed in “mini-batches” of n items
- The first dimension of the input and output tensors at each layer is always n
- The last dimension of the output is the number of kernels (filters)

Can you find the typo in the next figure?

Convolutional layer: Tensor shapes



Can you find the typo in this figure?

Dropout

- “Dropout” is a type of regularization that is easy to incorporate
- Idea: Randomly zero out entries in a tensor
- Helps guard against overfitting
- Usual form:
 - ▶ Independently zero out each entry with probability ε
 - ▶ Divide by $(1 - \varepsilon)$

Dropout

This illustrates something a bit different than the standard dropout

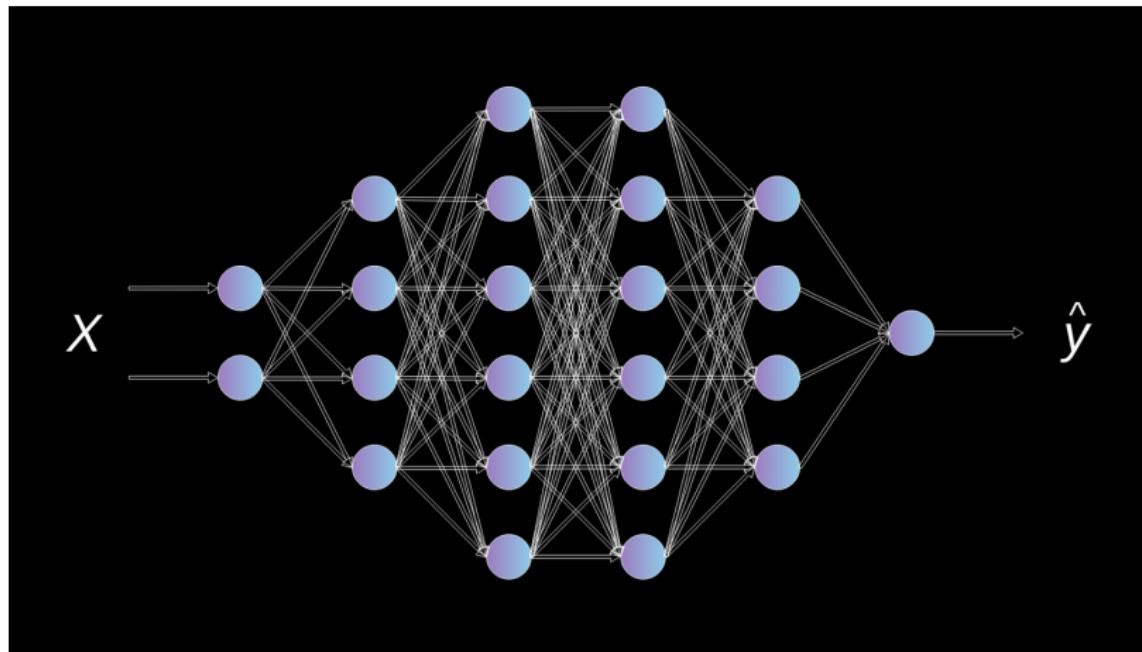
Flattening

- After one or more convolutional layers, we usually add “dense” layers
- These are the standard types of layers in a neural network, with each output neuron connected to each input neuron, with a tuneable weight
- In CNNs, the convolutional layers build up a set of “features” and then these features are used to make a prediction

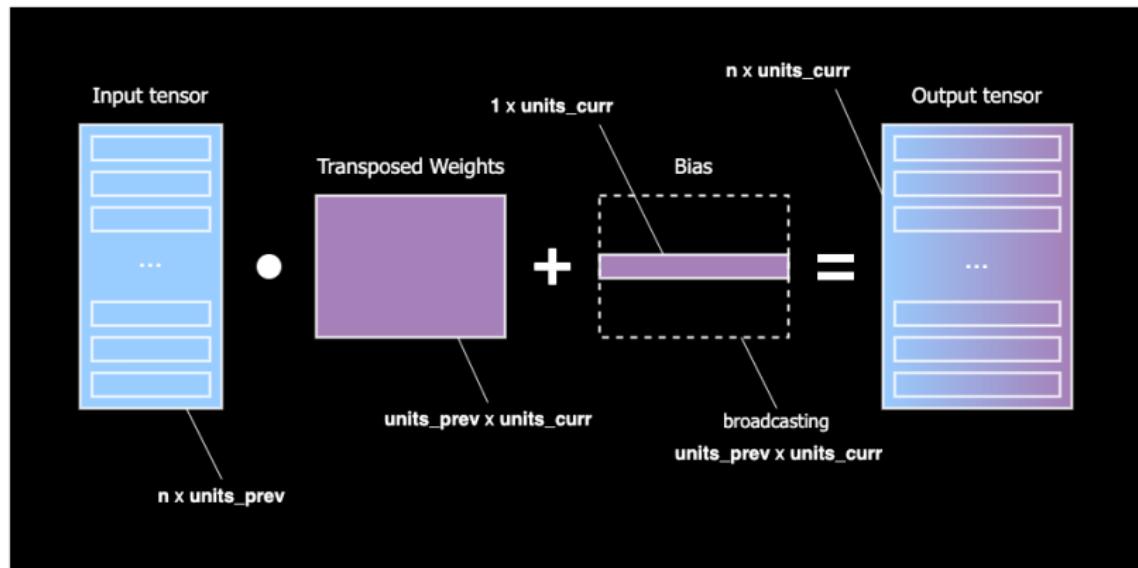
Flattening

This is the same as the `numpy.flatten()` operation

Dense layers



Dense operations

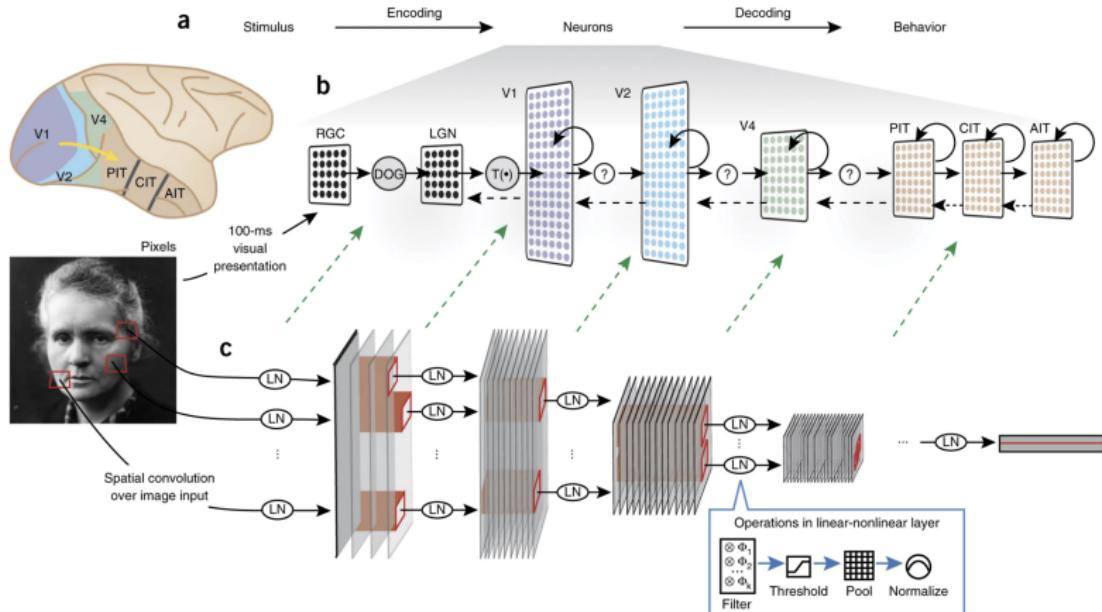


Again, this is not showing the use of an activation function

CNNs: Biological analogies

- It has been argued that common architectures for CNNs used for image processing mirror the organization and function of visual cortex in many species

CNNs: Biological analogies



Using goal-driven deep learning models to understand sensory cortex, Yamins and DiCarlo, 2016,
<https://www.nature.com/articles/nn.4244>

Problem 4: Brain food (20 points)

This problem gives you some experience with convolutional neural networks for image classification using [TensorFlow](#).

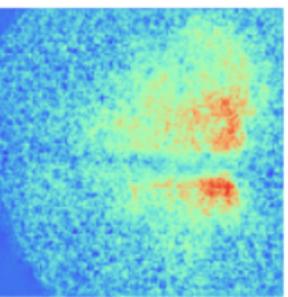
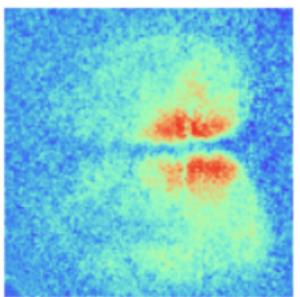
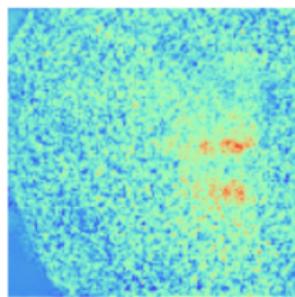
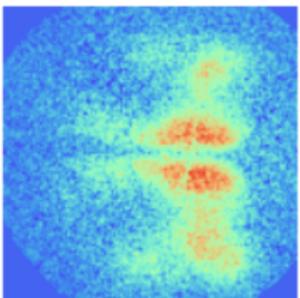
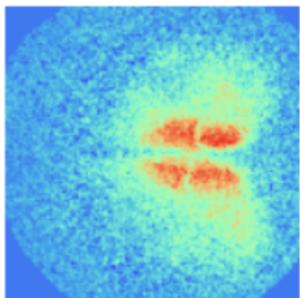
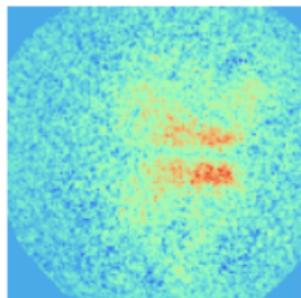
The classification task is to discriminate real optical images of brain activity in mice from fake images that were constructed using a [generative adversarial network \(GAN\)](#). A paper on the underlying imaging technologies developed by Yale researchers is [here](#).

For this problem we'll walk you through the following steps:

- Downloading the data
- Loading the data
- Displaying some sample images
- Building a classification model using a simple CNN

After this, your task will be to improve upon this baseline model by building, training, and evaluating two more CNNs.

Task: Real or Fake?



Task: Real or Fake?

- The real images are optical images of neural activity at the surface of the cortex in transgenic mice
- The fake images were generated by a “deconvolutional” neural network trained as part of a GAN
- The discriminator network is similar to the networks you’ll train

Baseline model

```
model = models.Sequential()
model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(128, 128, 1)))
model.add(layers.MaxPooling2D((4, 4)))
model.add(layers.Flatten())
model.add(layers.Dense(2))
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 124, 124, 32)	832
max_pooling2d (MaxPooling2D)	(None, 31, 31, 32)	0
flatten (Flatten)	(None, 30752)	0
dense (Dense)	(None, 2)	61506

Total params: 62,338

Trainable params: 62,338

Non-trainable params: 0

Baseline model

First layer:

- Input: 128×128 images, 1 channel
- First layer: 32 filters, each of size 5×5
- Output of first layer: tensor of shape $124 \times 124 \times 32$
- Number of parameters: $(5 \times 5 + 1) \times 32 = 832$
- Activation function: ReLU
- Output of first layer: tensor of shape $124 \times 124 \times 32$

Baseline model

Second layer:

- Maximum of 4×4 regions in feature map
- Output: Tensor of shape $31 \times 31 \times 32$ since $124/4 = 31$
- Number of parameters: Zero!

Baseline model

Third layer:

- Flattened version of previous layer
- Number of neurons: $31 \times 31 \times 32 = 30,752$
- Number of parameters: Zero!

Baseline model

Final layer:

- Dense connections
- Two neurons in output (“fake” and “real”)
- Number of parameters: $(30,752 + 1) \times 2 = 61,506$

Baseline model

Overall:

- Number of parameters: $832 + 61,506 = 62,338$
- Most come from dense layer (98.7%)

Let's build another model

```
model = models.Sequential()
model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(128, 128, 1)))
model.add(layers.MaxPooling2D((4, 4)))
model.add(layers.Conv2D(32, (2, 2), activation='relu'))
model.add(layers.MaxPooling2D((4,4)))
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(2))
model.summary()
```

Model: "sequential_11"

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_20 (Conv2D)	(None, 124, 124, 32)	832
<hr/>		
max_pooling2d_20 (MaxPooling)	(None, 31, 31, 32)	0
<hr/>		
conv2d_21 (Conv2D)	(None, 30, 30, 32)	4128
<hr/>		
max_pooling2d_21 (MaxPooling)	(None, 7, 7, 32)	0
<hr/>		
flatten_11 (Flatten)	(None, 1568)	0
<hr/>		
dense_17 (Dense)	(None, 128)	200832
<hr/>		
dense_18 (Dense)	(None, 2)	258
<hr/>		
Total params: 206,050		
Trainable params: 206,050		
Non-trainable params: 0		

Summary for today

- A convolutional layer applies a set of kernels (filters) across the input
- Each kernel is a tensor of smaller dimension than the input
- The values of the kernels are learned by backpropagation
- Successive convolutional layers build increasingly rich features
- The convolutional layers define a set of features that are then used to make predictions in one or more dense layers
- CNNs are attractive whenever the data have “spatial” structure where the same features might appear at many different locations