S&DS 365 / 665

Intermediate Machine Learning

Convolutional Neural Networks

(Continued)

February 16

Reminders

- Assignment due a week from today
- Quiz 1 available after class; 15 minutes, on Canvas for 24 hours
- Any material covered in class

Topics for today

- Finish up convolutional networks
- Jump-starting Problem 4 on Assn 4
- A few other comments on ANNs

Convolution

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

Continuous:

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

- Same operation we saw with smoothing kernels
- If K is a Gaussian bump, this blurs or smooths out the values.
- K is trained to encode predictive features of the input

Convolution

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

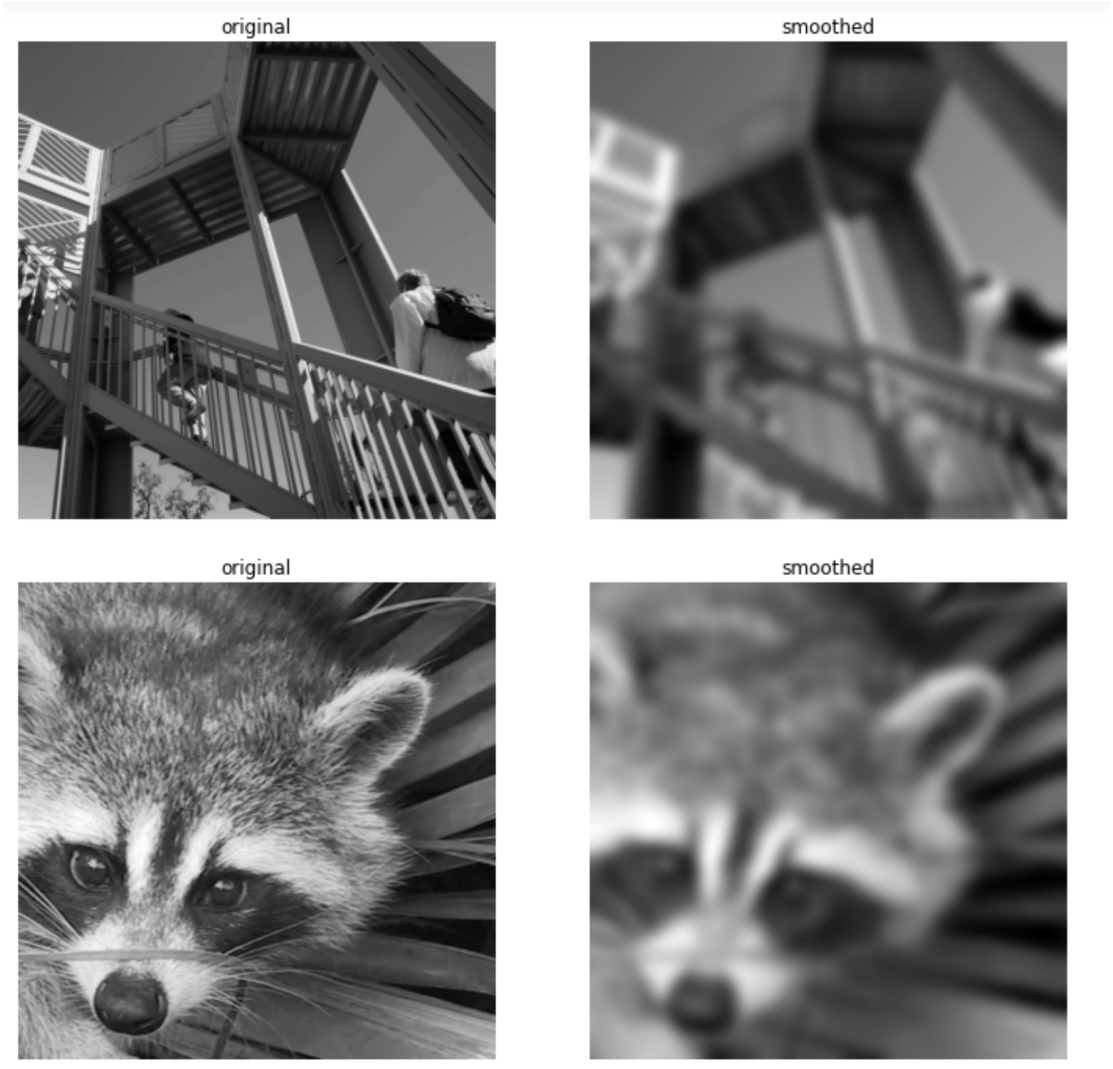
Discrete:

$$(K \star f)(x_j) = \sum_i K(x_j - x_i)\, f(x_i)$$

- Same operation we saw with smoothing kernels
- If K is a Gaussian bump, this blurs or smooths out the values.
- K is trained to encode predictive 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

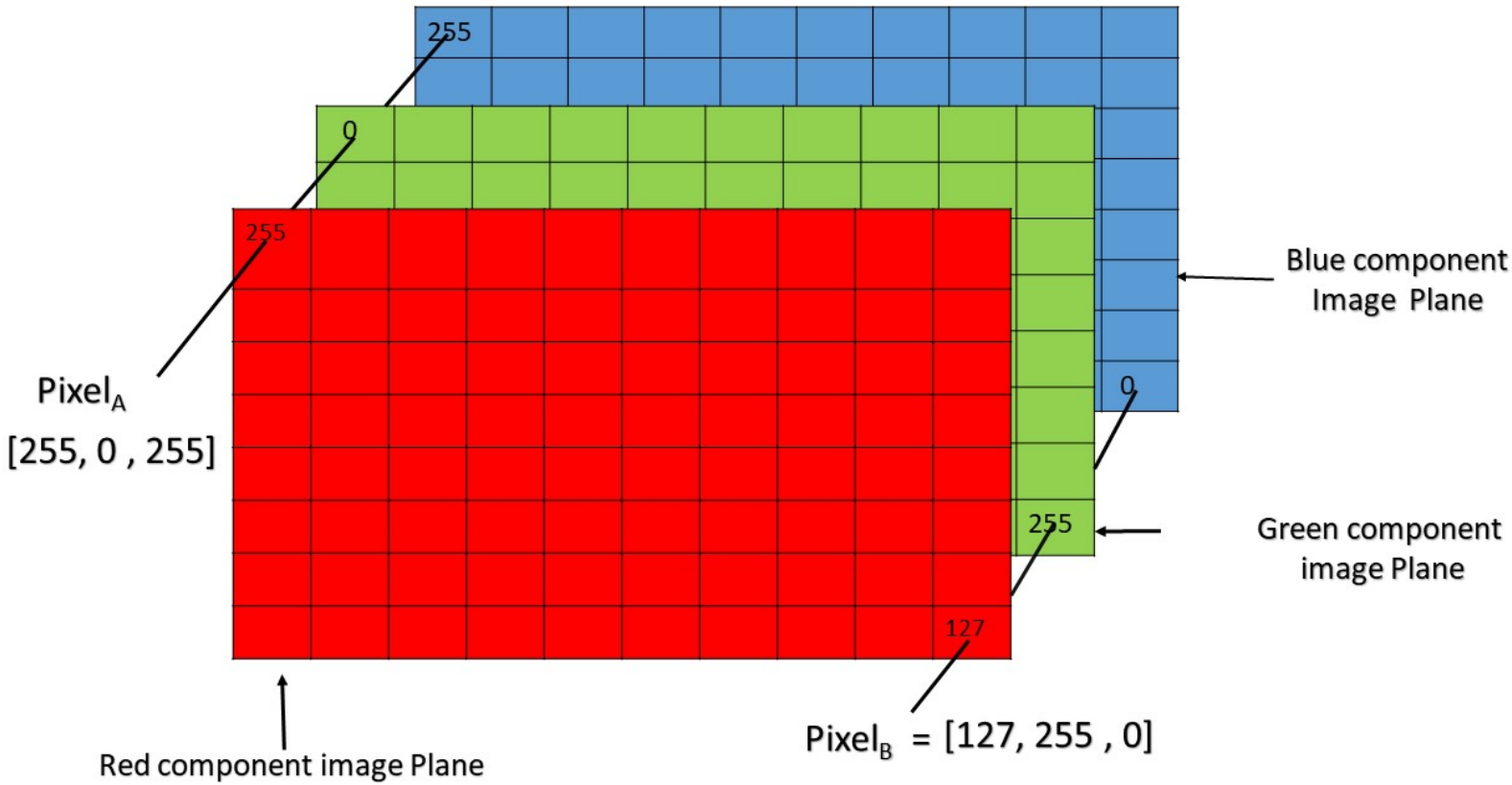
```
x = np.array(np.arange(24)).reshape(2,3,4)
print(x)
print(x[:,:,0])
```

```
[[[ 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

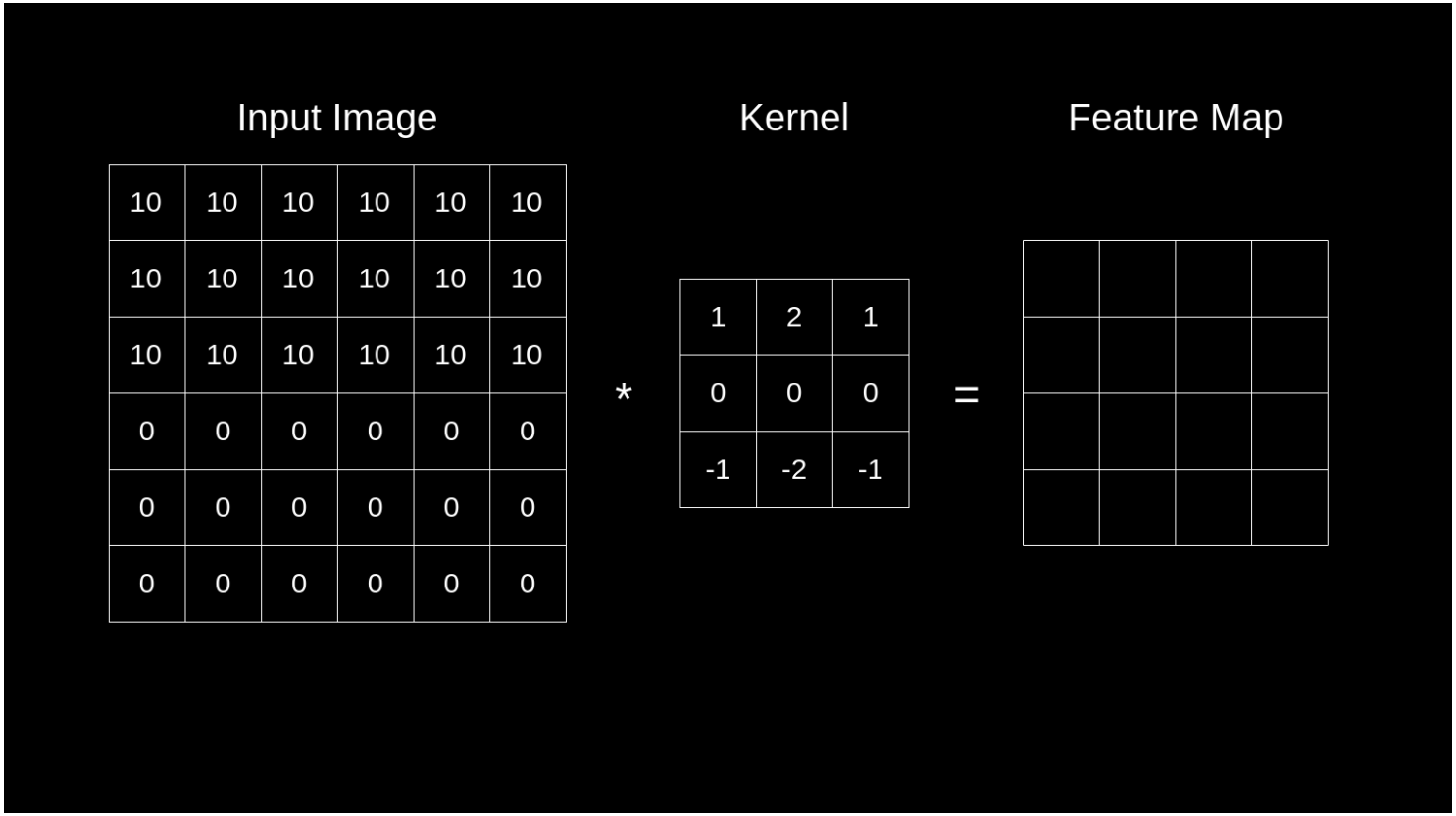


Pixel of an RGB image are formed from the corresponding pixel of the three component images

Convolutional layer

- In a convolutional layer, a set of *kernels* are “swept over” the input
- The kernels are themselves tensors with entries that are trained
- 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 “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

Computing the feature map

After multiplying the input tensor by the kernel, we add in the intercept and apply the activation function

Example: Suppose input and kernel are

0	1	2	3	4
-5	-6	-7	-8	-9
1	2	3	-2	1
4	3	2	1	0
-9	-8	-7	-6	-5

2	-1
-1	2

Also suppose intercept is $b = 1$ and the activation function is $\text{ReLU}(u) = \max\{0, u\}$.

Computing the feature map

After multiplying the input tensor by the kernel, we add in the intercept and apply the activation function

The feature map at position [2,1] is then

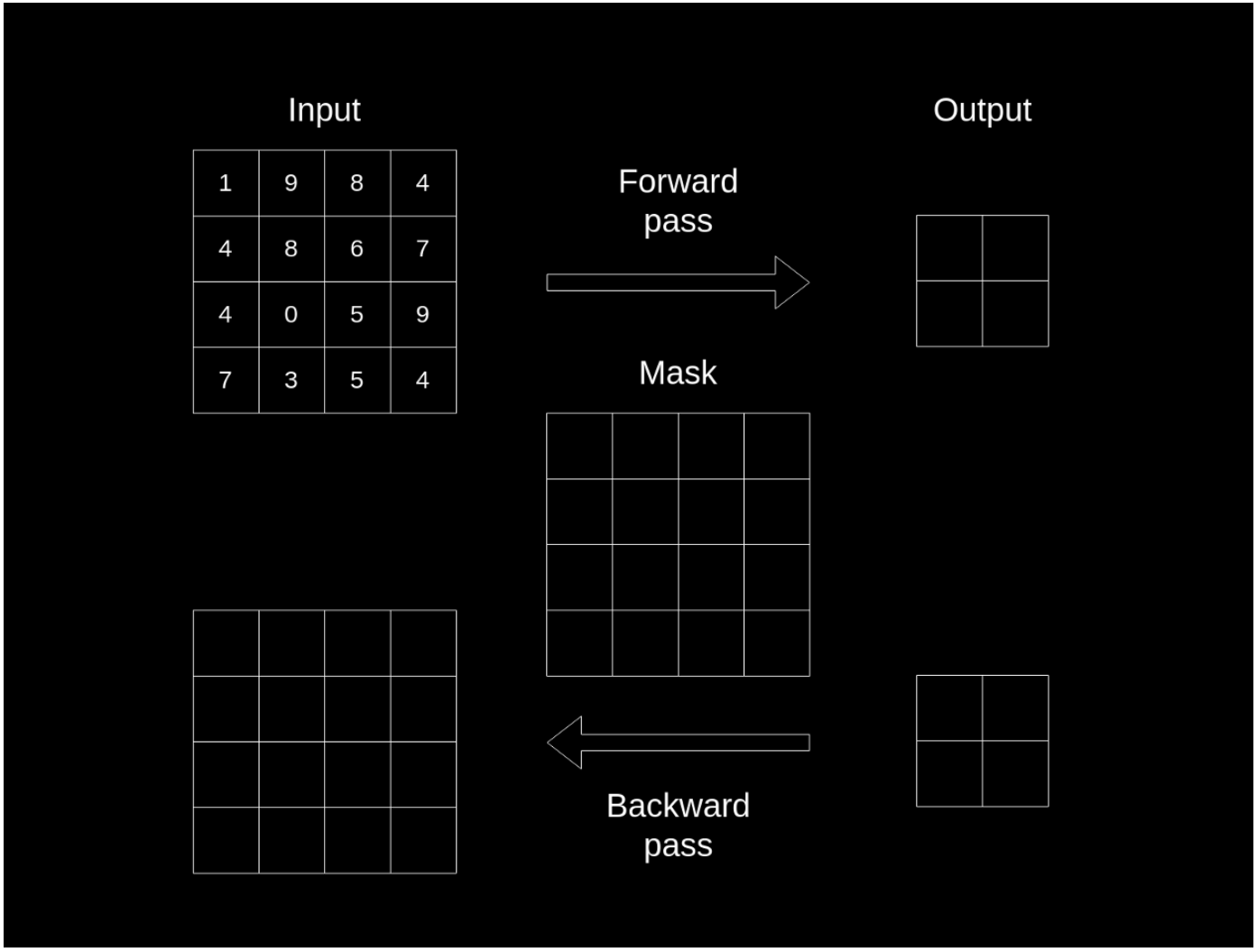
$$\begin{aligned}
&\text{ReLU}\left(\left\langle \begin{pmatrix} 2 & 3 \\ 3 & 2 \end{pmatrix}, \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} \right\rangle + 1\right) \\
&\quad = \max\{2 \cdot 2 + 3 \cdot (-1) + 3 \cdot (-1) + 2 \cdot 2 + 1, 0\} \\
&\quad = 3
\end{aligned}$$

Here $\langle A, B \rangle \equiv \text{trace}(A^T B) = \sum_{ij} A_{ij} B_{ij}$ is the component-wise product and sum of the entries.

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

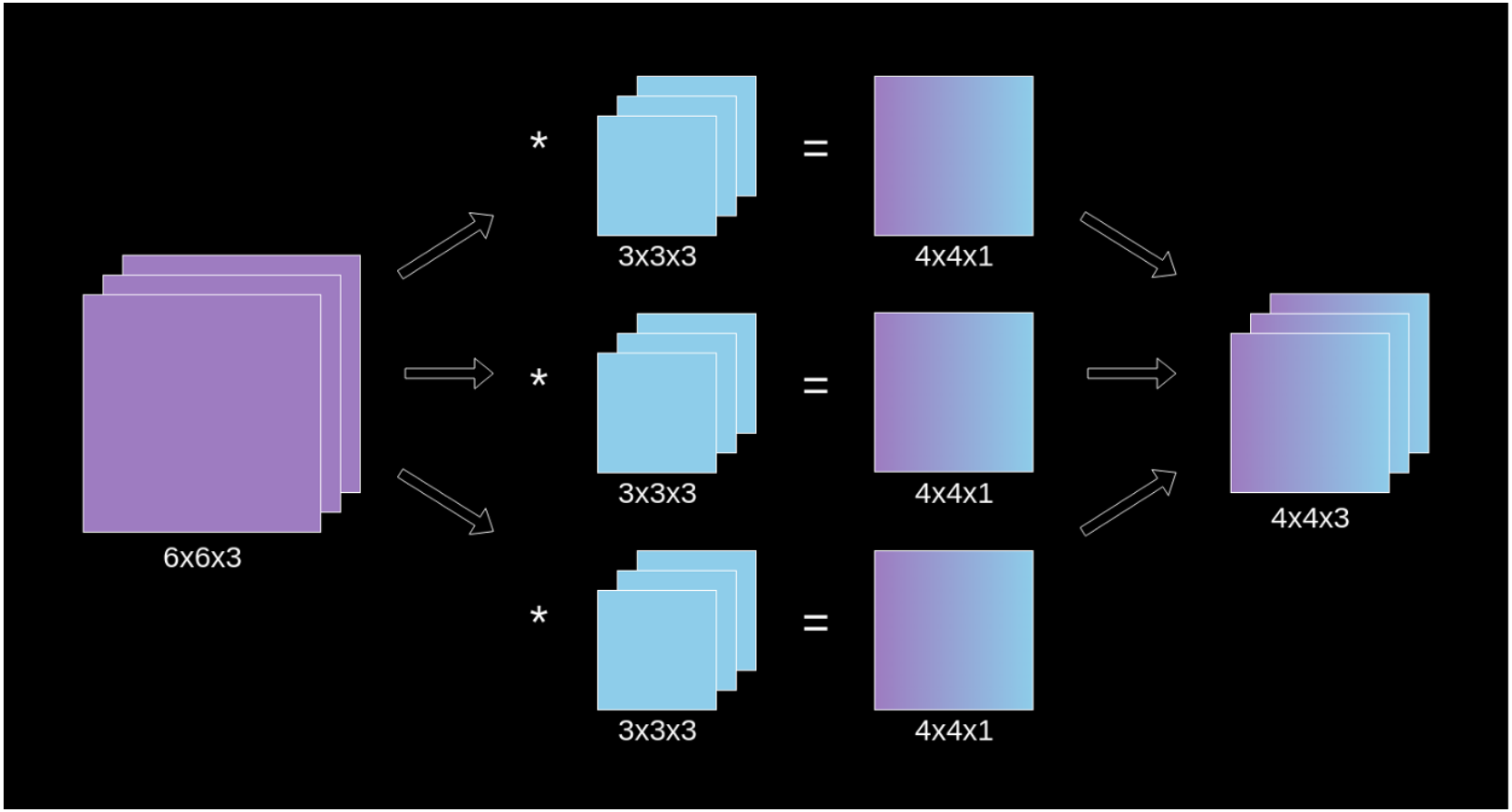
Pooling


Input
Output
1 9 8 4
4 8 6 7
4 0 5 9
7 3 5 4
Forward pass
Mask
Backward pass

Applying kernels across channels

- 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

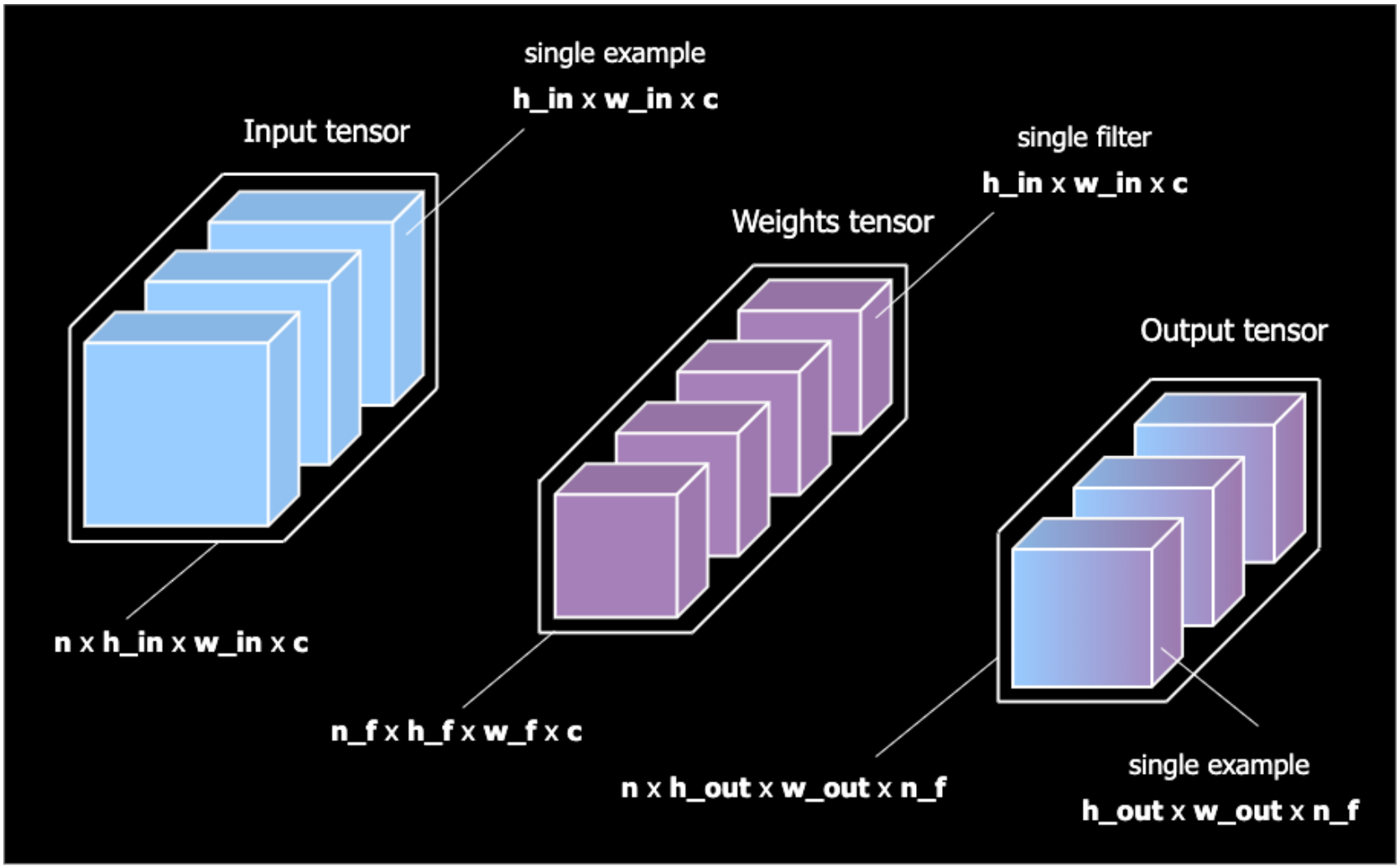


If there were 100 kernels in this layer, the output tensor would have shape $4 \times 4 \times 100$

Convolutional layer: Tensor shapes

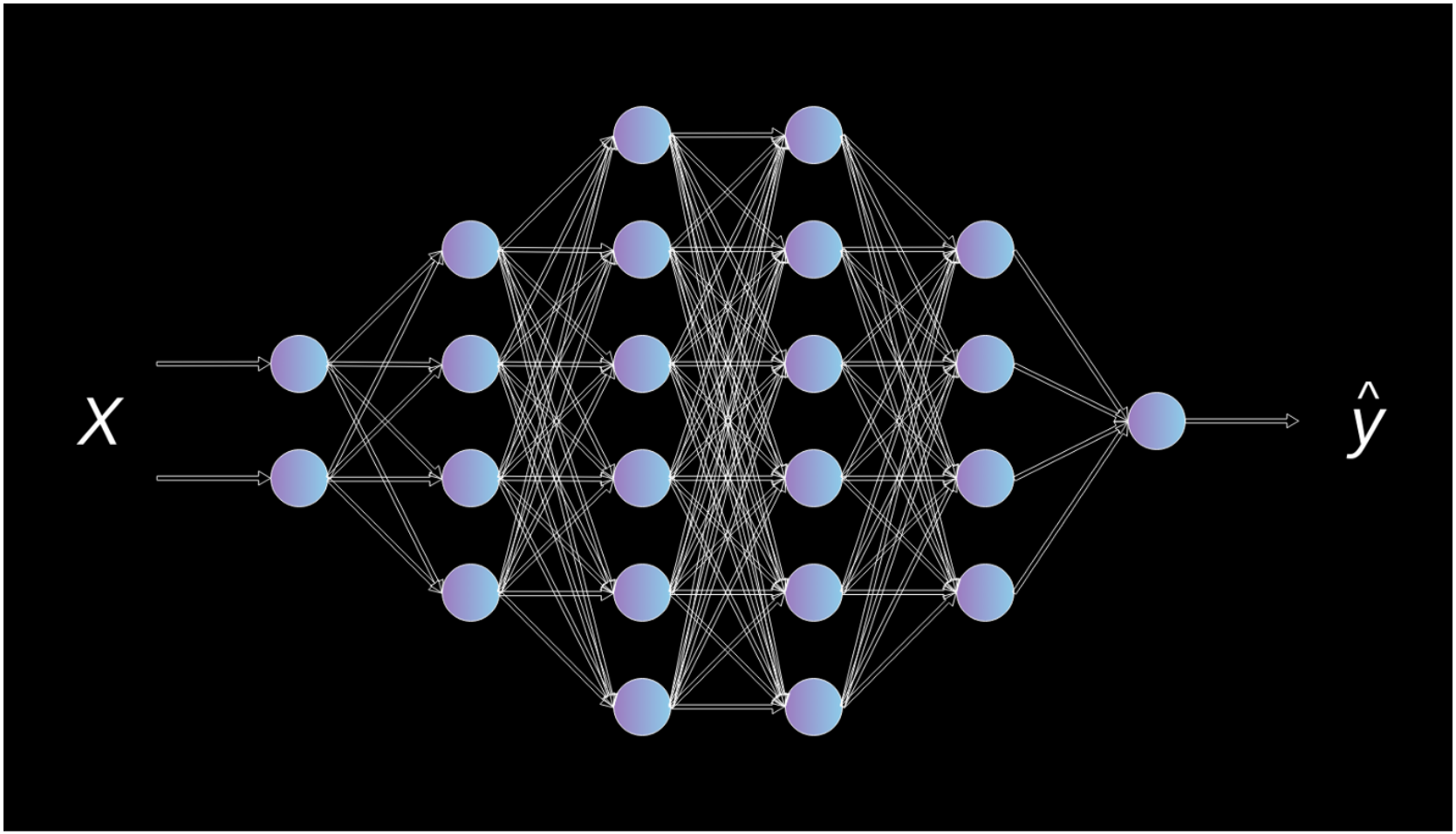
- Data points are processed in “mini-batches” of n items
- The first dimension of the input and output tensors at each layer is always n
- The last dimension of the output is the number of kernels (filters)

Convolutional layer: Tensor shapes

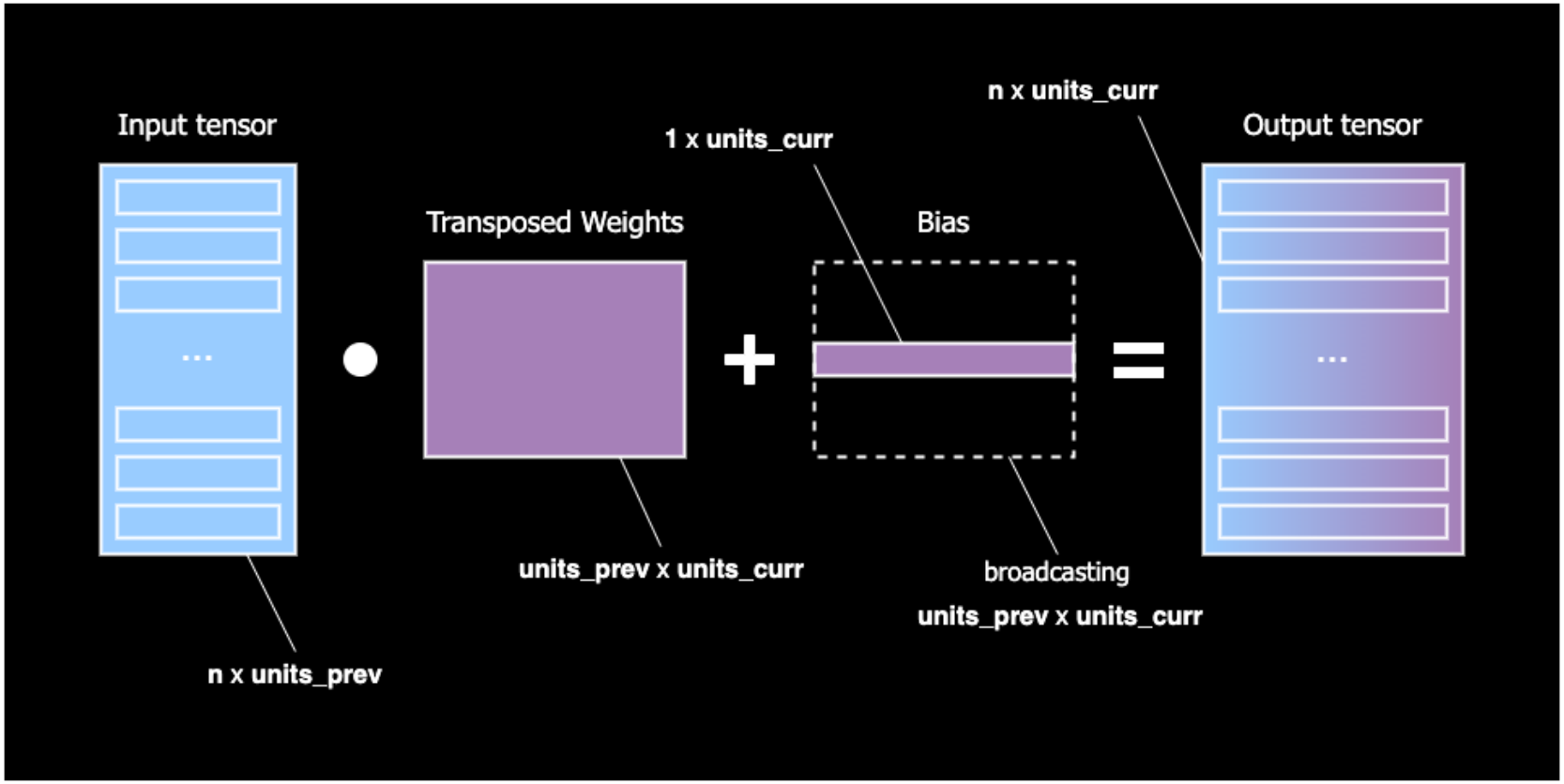


Can you find the typo in this figure?

Dense layers



Dense operations



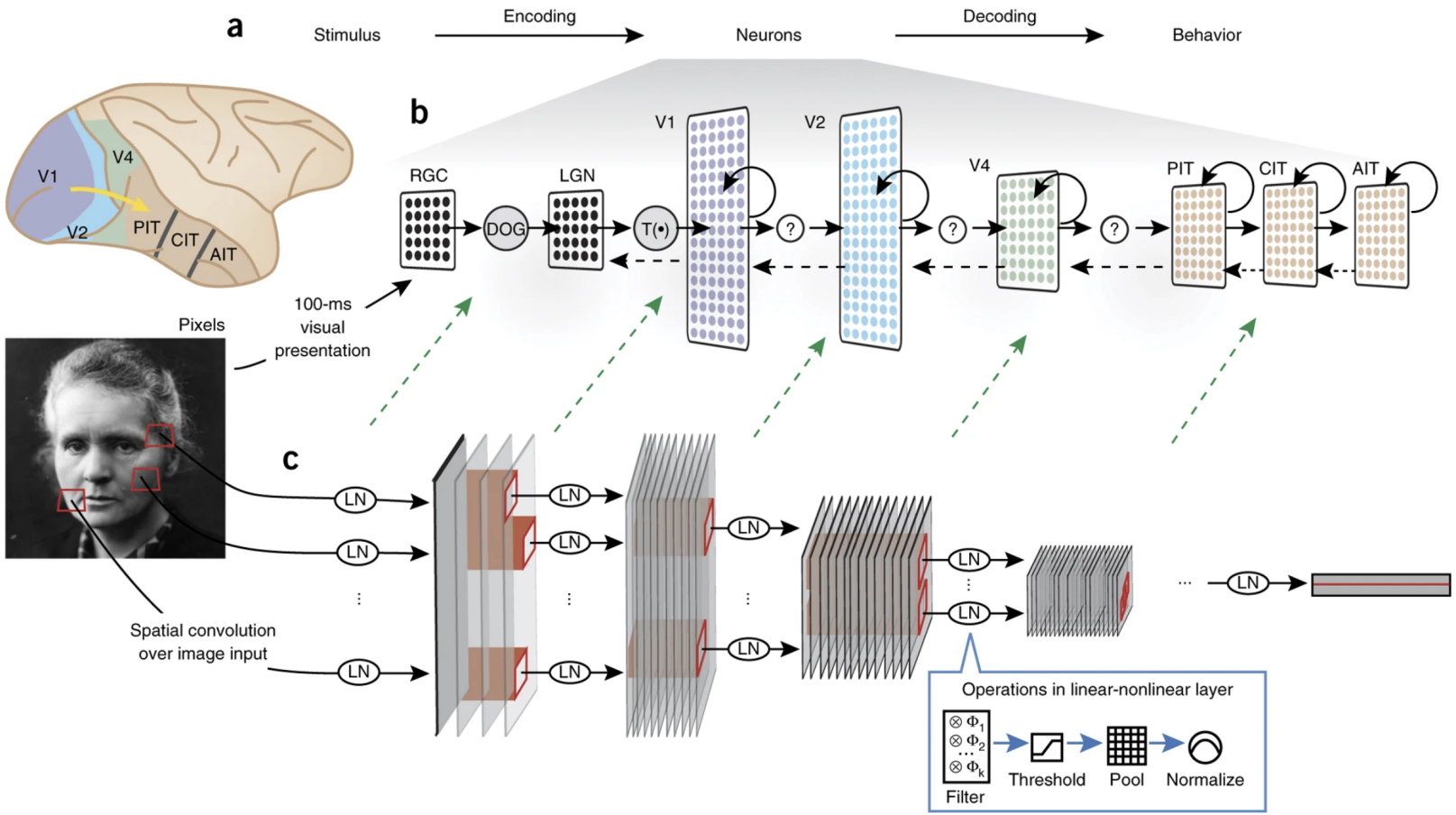
We'll review standard dense layers shortly...

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.

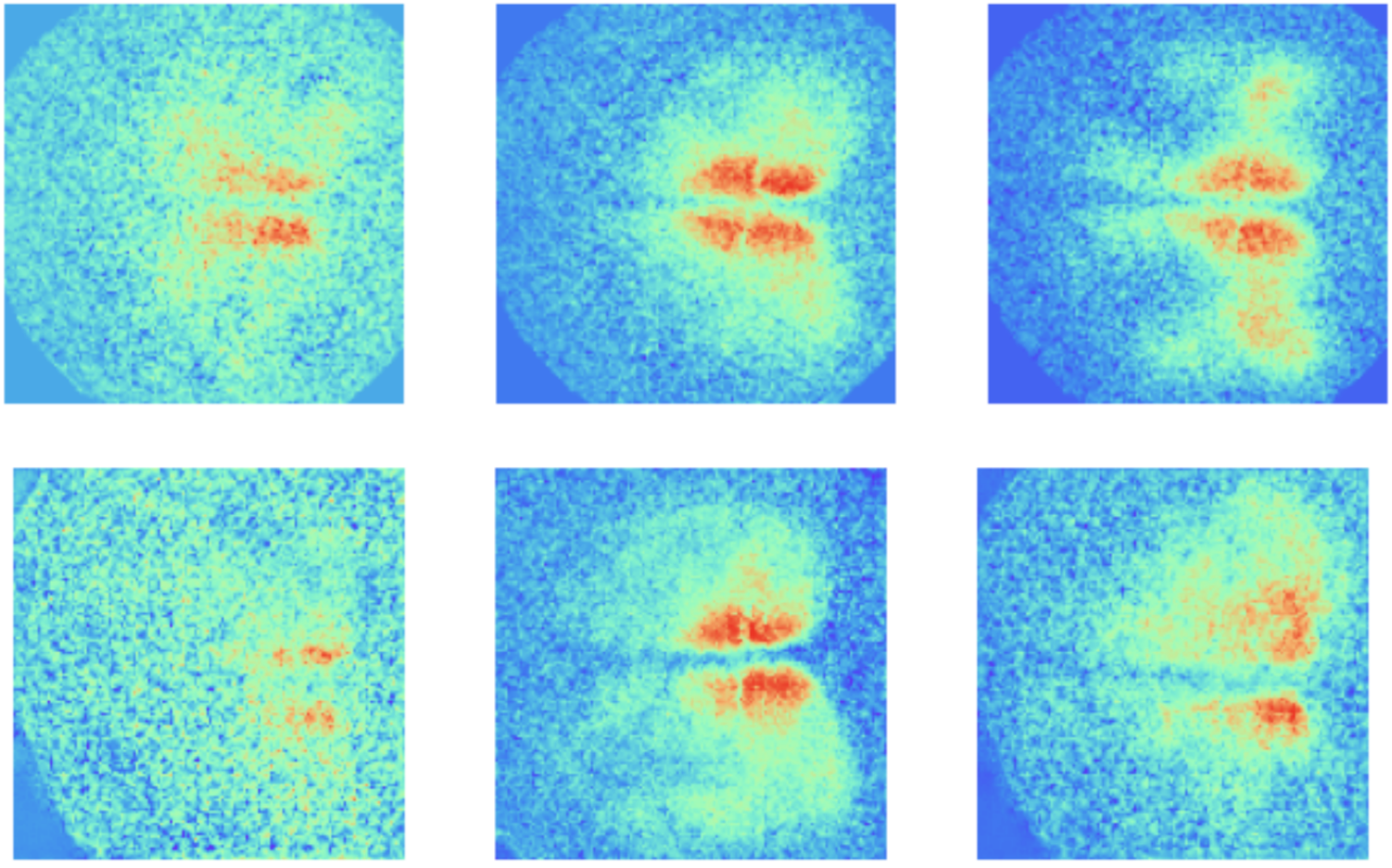
The classification task is to discriminate real optical images of brain activity in mice from fake images that were constructed using a generative adversarial network (GAN). A paper on the underlying imaging technologies developed by Yale researchers is here.

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 #
=================================================================
conv2d_20 (Conv2D)           (None, 124, 124, 32)      832
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 30, 30, 32)        4128
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 7, 7, 32)          0
_________________________________________________________________
flatten_11 (Flatten)         (None, 1568)              0
_________________________________________________________________
dense_17 (Dense)             (None, 128)               200832
_________________________________________________________________
dense_18 (Dense)             (None, 2)                 258
=================================================================
Total params: 206,050
Trainable params: 206,050
Non-trainable params: 0
_________________________________________________________________
```

More on neural nets

Let’s review some material from the slides posted last week

Nonlinearities

Add nonlinearity

$$h = \varphi(Wx + b)$$

applied component-wise.

For regression, the last layer is just linear:

$$f = \beta^T h + \beta_0$$

Nonlinearities

Commonly used nonlinearities:

$$\varphi(u) = \tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$$

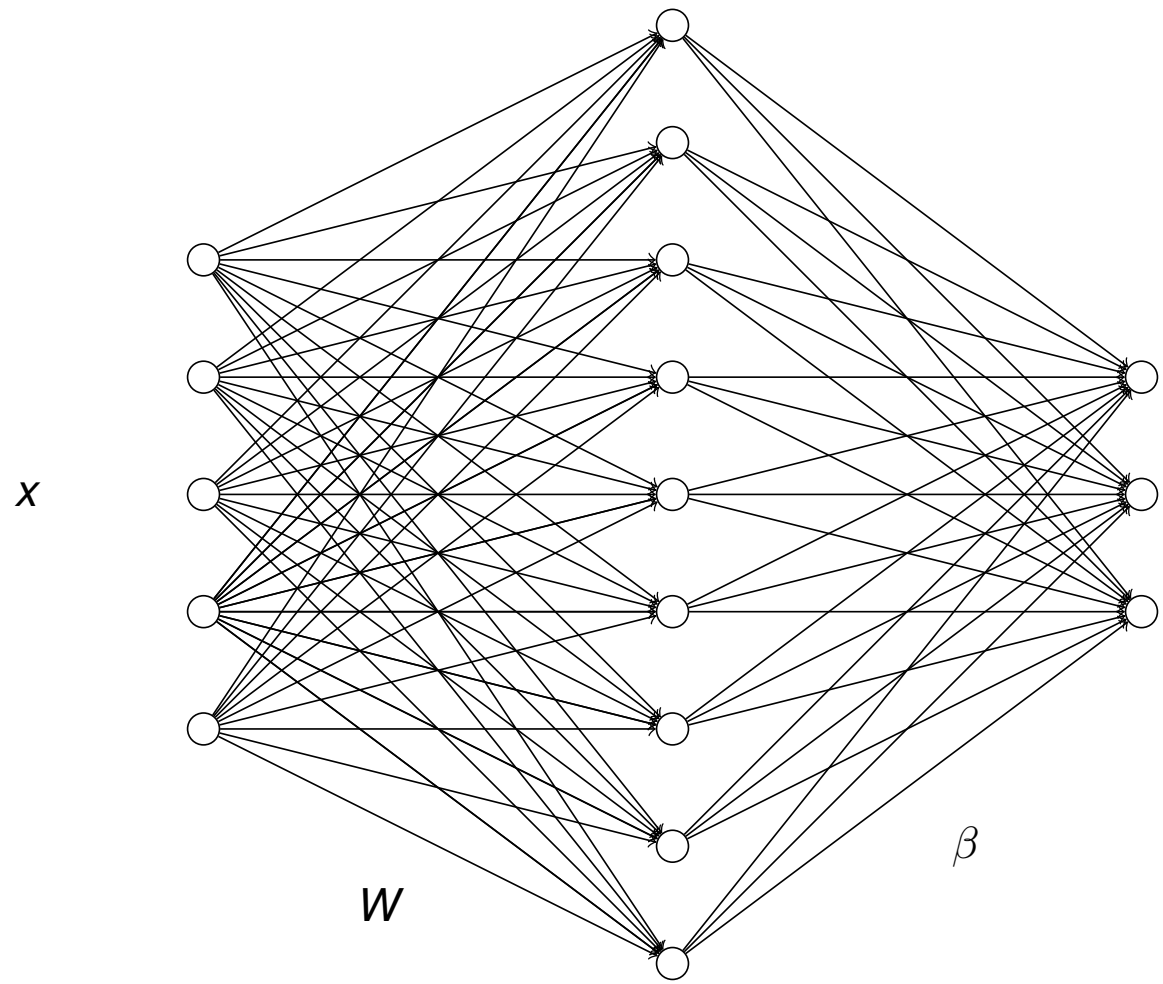
$$\varphi(u) = \text{sigmoid}(u) = \frac{e^u}{1 + e^u}$$

$$\varphi(u) = \text{relu}(u) = \max(u, 0)$$

Nonlinearities

So, a neural network is nothing more than a parametric regression model with a restricted type of nonlinearity

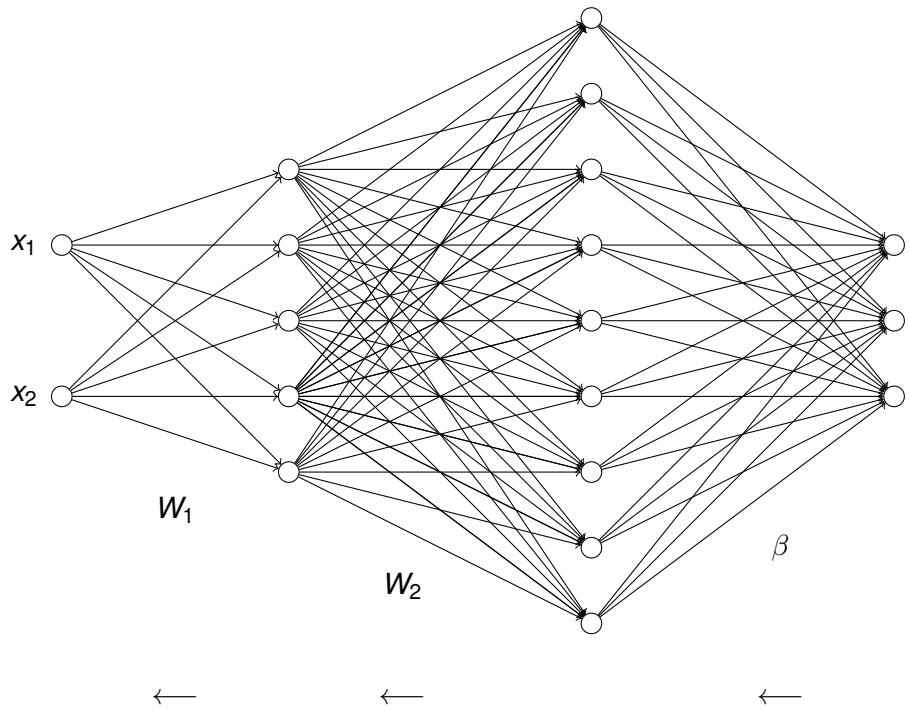
Two-layer dense network



Training

- The parameters are trained by stochastic gradient descent.
- To calculate derivatives we just use the chain rule, working our way backwards from the last layer to the first.

High level idea



Start at last layer, send error information back to previous layers

Start simple

Loss is

$$\mathcal{L} = \frac{1}{2}(y - f)^2$$

The change in loss due to making a small change in output f is

$$\frac{\partial \mathcal{L}}{\partial f} = (f - y)$$

We now send this backward through the network

Example

So if $f = Wx + b$ then

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial f} x^T$$
$$= (f - y) x^T$$

Example

So if $f = Wx + b$ then

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial f}$$
$$= (f - y)$$

Two layers

Now add a layer:

$$f = W_2 h + b_2$$
$$h = W_1 x + b_1$$

Then we have

$$\frac{\partial \mathcal{L}}{\partial W_2} = \frac{\partial \mathcal{L}}{\partial f} \, h^T$$
$$= (f - y) \, h^T$$

$$\frac{\partial \mathcal{L}}{\partial h} = W_2^T \, \frac{\partial \mathcal{L}}{\partial f}$$
$$= W_2^T \, (f - y)$$

Two layers

Now send this back (backpropagate) to the first layer:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_1} &= \frac{\partial \mathcal{L}}{\partial h}\, x^T \\ &= W_2^T\, \frac{\partial \mathcal{L}}{\partial f}\, x^T \\ &= W_2^T\, (f - y)\, x^T\end{aligned}$$

Adding a nonlinearity

Remember, this just gives a linear model! Need a nonlinearity:

$$h = \varphi(W_1 x + b_1)$$

$$f = W_1 h + b_2$$

Adding a nonlinearity

If $\varphi(u) = ReLU(u) = \max(u, 0)$ then this just becomes

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_1} &= \mathbb{1}(h > 0)\ \frac{\partial \mathcal{L}}{\partial h}\ x^T \\ &= \mathbb{1}(h > 0)\ W_2^T\ \frac{\partial \mathcal{L}}{\partial f}\ x^T \\ &= \mathbb{1}(h > 0)\ W_2^T\ (f - y)\ x^T\end{aligned}$$

where

$$\mathbb{1}(u) = \begin{cases} 1 & u > 0 \\ 0 & \text{otherwise} \end{cases}$$

See notes on backpropagation from iML for details

---

https://github.com/YData123/sds265-fa21/raw/main/notes/backprop.pdf

Classification

For classification we use softmax to compute probabilities

$$(p_1, p_2, p_3) = \frac{1}{e^{f_1} + e^{f_2} + e^{f_3}} \left(e^{f_1}, e^{f_2}, e^{f_3} \right)$$

The loss function is

$$\mathcal{L} = -\log P(y \mid x) = \log \left(e^{f_1} + e^{f_2} + e^{f_3} \right) - f_y$$

So, we have

$$\frac{\partial \mathcal{L}}{\partial f_k} = p_k - \mathbb{1}(y = k)$$

Interactive examples

https://playground.tensorflow.org/

Neural tangent kernel

There is a kernel view of neural networks that has been useful in understanding the dynamics of stochastic gradient descent for neural networks.

This is based on the *neural tangent kernel (NTK)*

Parameterized functions

Suppose we have a parameterized function $f_\theta(x) \equiv f(x;\theta)$

Almost all machine learning takes this form — for classification and regression, these give us estimates of the regression function

For neural nets, the parameters θ are all of the weight matrices and bias (intercept) vectors across the layers.

Feature maps

Suppose we have a parameterized function $f_\theta(x) \equiv f(x;\theta)$

We then define a *feature map*

$$x \mapsto \varphi(x) = \nabla_\theta f(x;\theta) = \begin{pmatrix} \dfrac{\partial f(x;\theta)}{\partial \theta_1} \\ \dfrac{\partial f(x;\theta)}{\partial \theta_2} \\ \vdots \\ \dfrac{\partial f(x;\theta)}{\partial \theta_p} \end{pmatrix}$$

This defines a Mercer kernel

$$K(x,x') = \varphi(x)^T \varphi(x') = \nabla_\theta f(x;\theta)^T \nabla_\theta f(x';\theta)$$

NTK and SGD

- The NTK has been used to study the dynamics of stochastic gradient descent
- Upshot: As the number of neurons in the layers grows, the parameters in the network barely change during training, even though the training error quickly decreases to zero
- For a colorful blog on the NTK, see `https://rajatvd.github.io/NTK/`

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
- Parameters are trained using backpropagation
- NTK is a type of Mercer kernel used to analyze training dynamics of neural nets