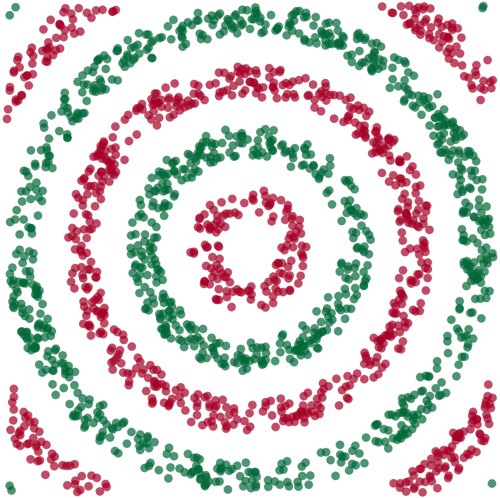
DSC 190

Machine Learning: Representations

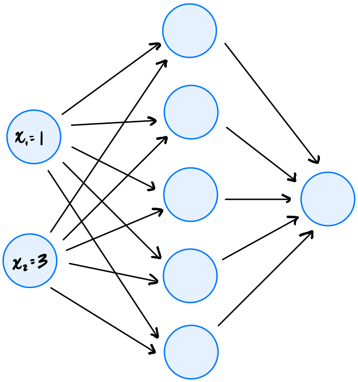
Lecture 15 | Part 1

NNs and Representations



NNs and Representations

- Hidden layer transforms to new **representation**.
  - Maps $\mathbb{R}^2 \to \mathbb{R}^5$
- Output layer makes prediction.
  - Maps $\mathbb{R}^5 \to \mathbb{R}^1$
- Representation optimized for classification!

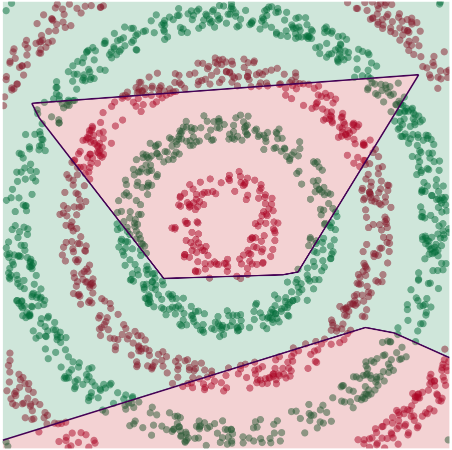


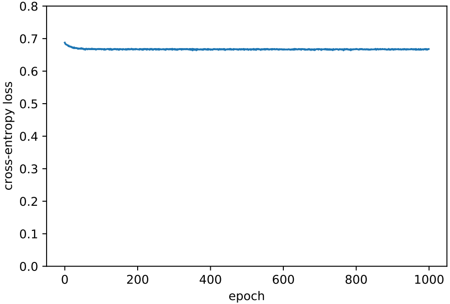
NN Design

- Design a network for classification.
- Hidden layer activations: ReLU
- Output layer activation: sigmoid
- Loss function: cross-entropy

```python
from tensorflow import keras

inputs = keras.Input(shape=2)
hidden_1 = keras.layers.Dense(5, activation='relu')(inputs)
outputs = keras.layers.Dense(1, activation='sigmoid')(hidden_1)

model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=.01),
    loss=keras.losses.BinaryCrossentropy()
)

history = model.fit(X, y, epochs=1000, verbose=1)
```

Results

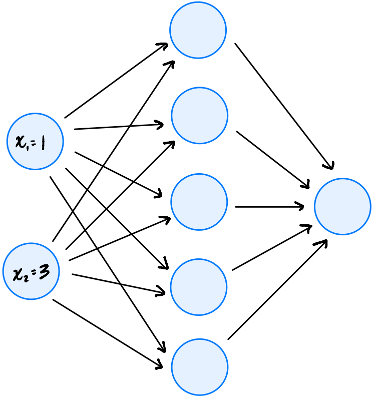




0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
cross-entropy loss
0
200
400
600
800
1000
epoch

NNs and Representations

- Data has complex structure.
- Only 5 hidden neurons not enough to learn a good representation.



DSC 190

Machine Learning: Representations

Lecture 15 | Part 2

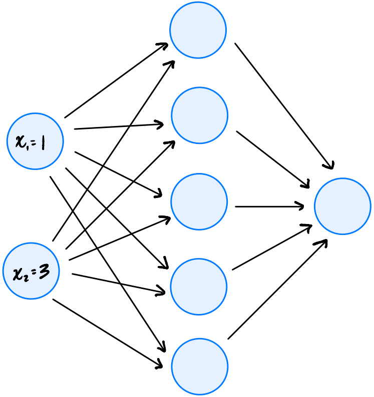
Architecture

Architecture

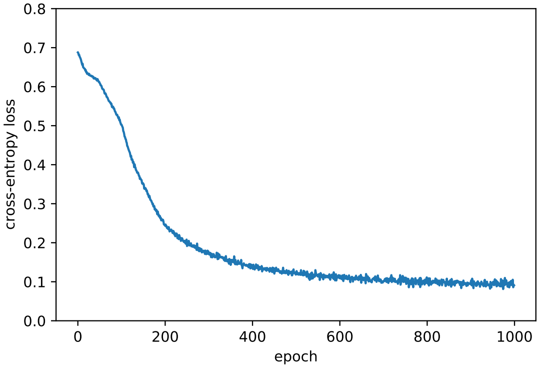
- We can increase complexity in two ways:
- Increasing **width**.
- Increasing **depth**.

Increasing Width

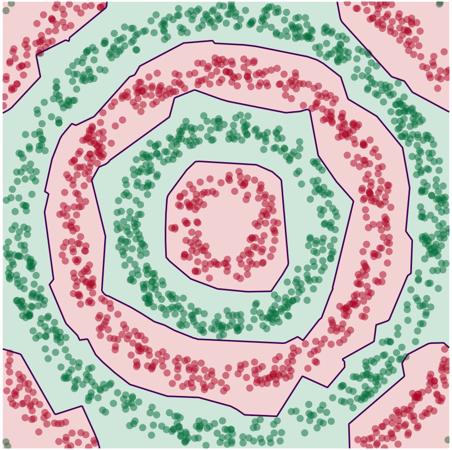
- Use a single hidden layer.
- But with 50 hidden neurons instead of 5.
- I.e., map to $\mathbb{R}^{50}$, then predict.



Loss


0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
cross-entropy loss
0
200
400
600
800
1000
epoch

Result



Universal Approximation Theorem

- A neural network f is a special type of function.
- Given another function g, can we make a neural network f so that $f(\vec{x}) \approx g(\vec{x})$?
- **Yes!** Assuming:
  - f has a hidden layer with a suitable activation function (ReLU, sigmoid, etc.)
  - the hidden layer has **enough** neurons
  - g is not too wild.

Main Idea

A network with a single hidden layer is able to approximate any (not-too-wild) function arbitrarily well as long as it has enough neurons in the hidden layer.

So what?

- Nature uses some function g to assign class labels to data.
- We don't see this function. But we see $g(\vec{x})$ for a bunch of points.
- Our goal is to learn a function f approximating g using this data.

The Challenge

- NNs are universal approximators (so are RBF networks, etc.)
- But just because it *can* approximate any function, doesn't mean we can *learn* the approximation.

Number of Neurons

- UAT says one hidden layer works well with “enough neurons”
- What is enough?
- Unfortunately, it can be a lot!

DSC 190

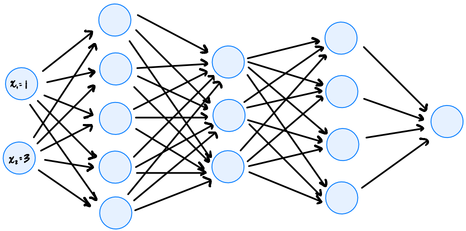
Machine Learning: Representations

Lecture 15 | Part 3

Deep Networks

Deep Networks

- Use a **multiple** hidden layers.
- Hidden layers transform to a new representation.
- Composition of simple transformations.
- Output layer performs prediction.



Main Idea

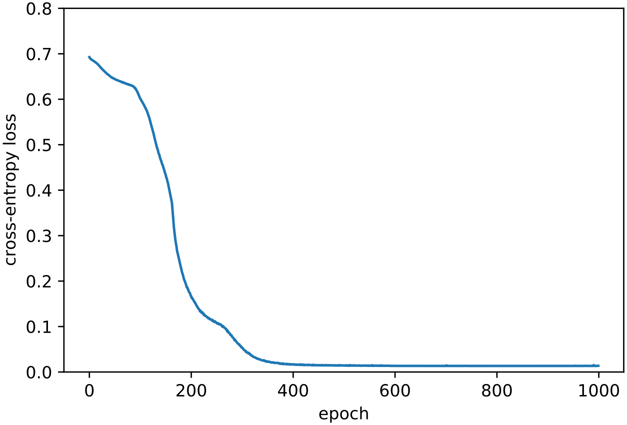
In machine learning, “deep” means “more than one hidden layer”. Deep models are useful for **learning** simpler representations.

Designing a Deep NN

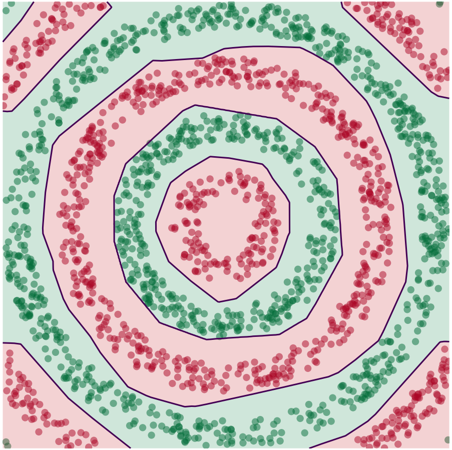
- Pick a number of hidden layers.
- Pick width of each hidden layer.
- There's not much theory to help us here.
- Experiment with different choices.

```python
inputs = keras.Input(shape=2)
hidden_1 = keras.layers.Dense(15, activation='relu')(inputs)
hidden_2 = keras.layers.Dense(20, activation='relu')(hidden_1)
hidden_3 = keras.layers.Dense(2, activation='relu')(hidden_2)
outputs = keras.layers.Dense(1, activation='sigmoid')(hidden_3)

model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=.001),
    loss=keras.losses.BinaryCrossentropy()
)

history = model.fit(X, y, epochs=1000, verbose=1)
```

Loss

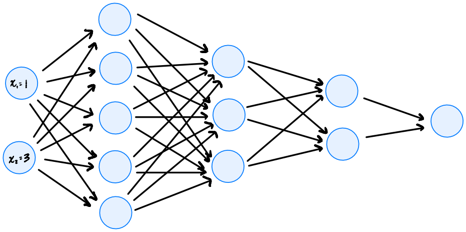

0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
cross-entropy loss
0
200
400
600
800
1000
epoch

Result

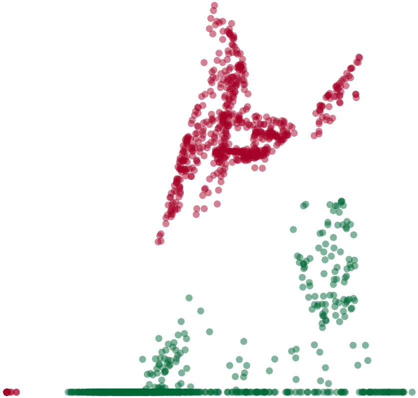


Deep Networks

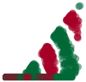
- Hidden layers map input to new representation.
- We can see this new representation!
- Plug in $\vec{x}$ and see activations of last hidden layer.

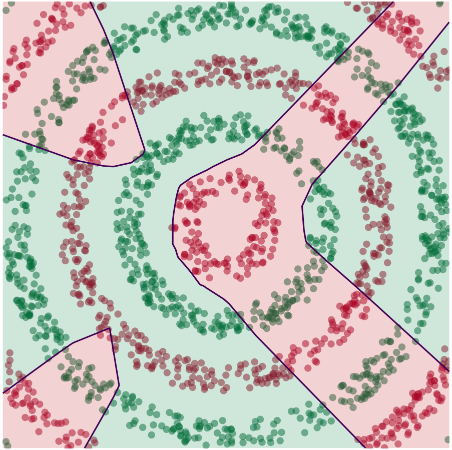


The New Representation

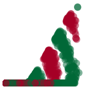


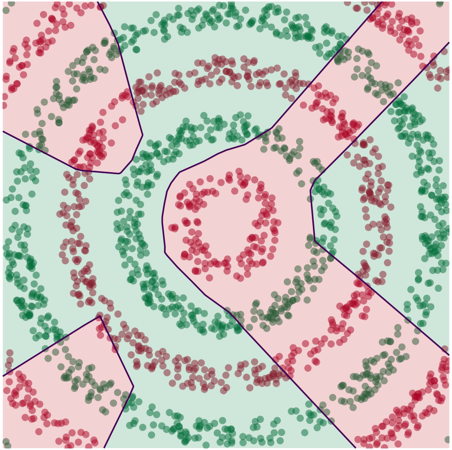
Learning a New Representation



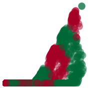


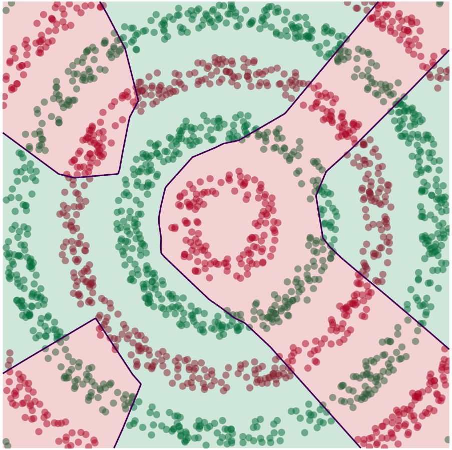
Learning a New Representation



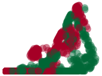


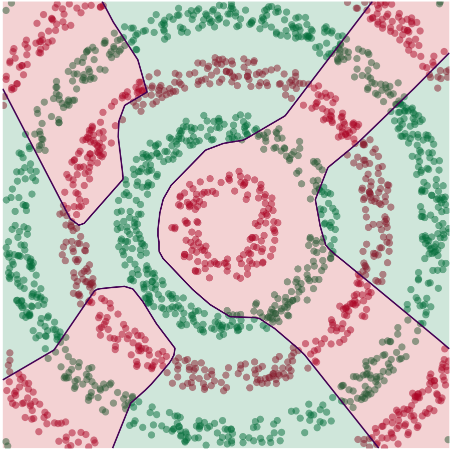
Learning a New Representation



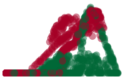


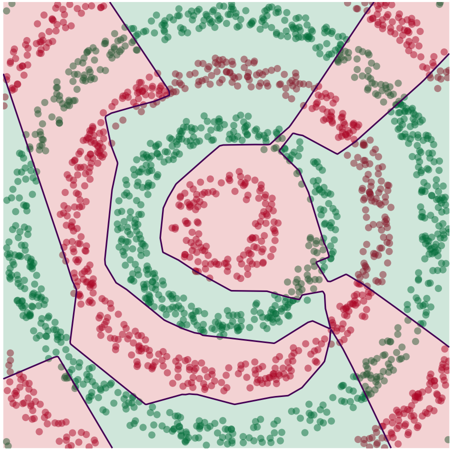
Learning a New Representation



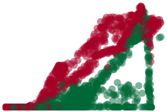


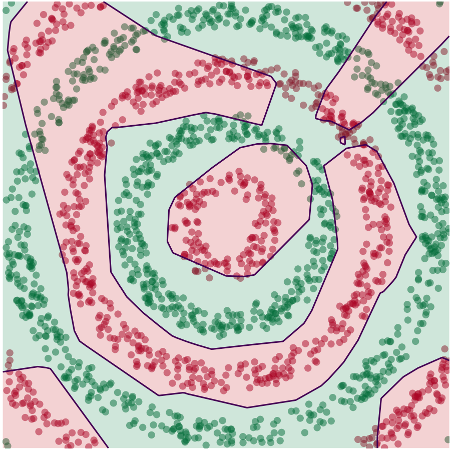
Learning a New Representation



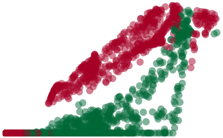


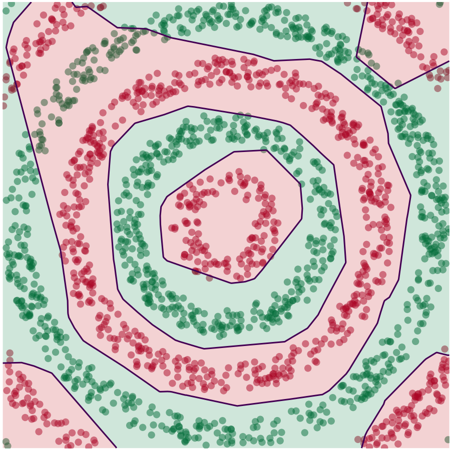
Learning a New Representation



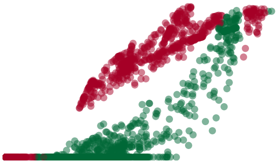


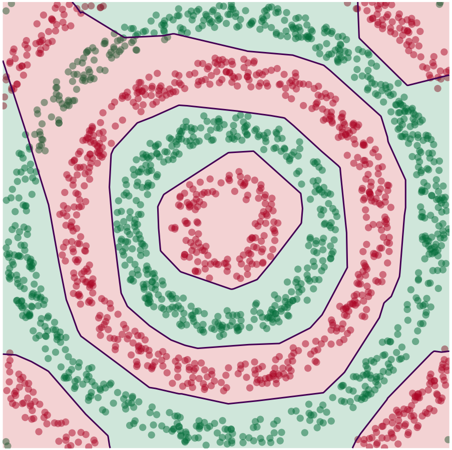
Learning a New Representation



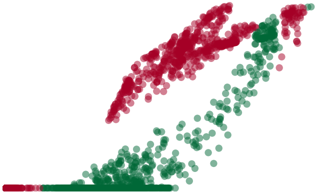


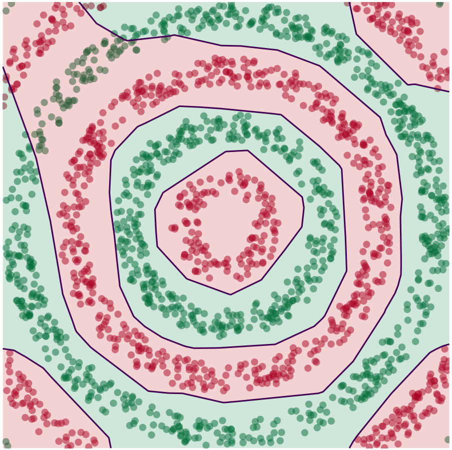
Learning a New Representation



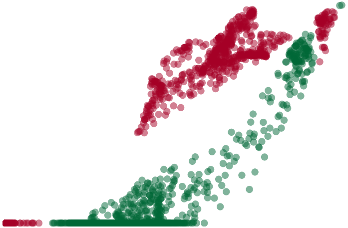


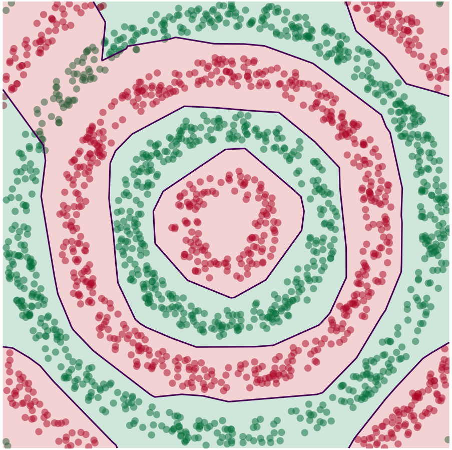
Learning a New Representation



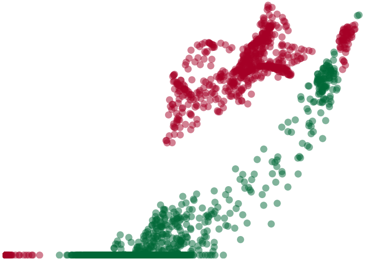


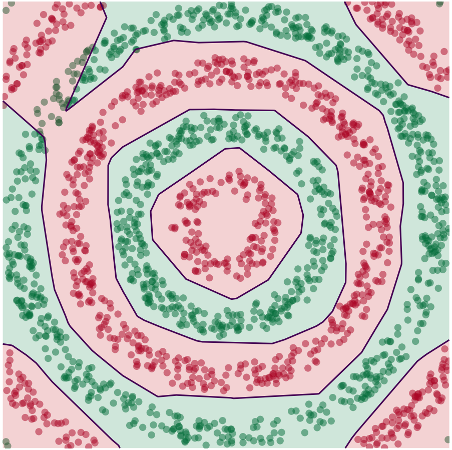
Learning a New Representation



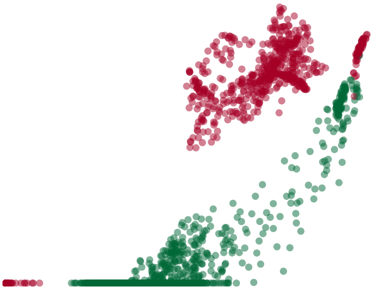


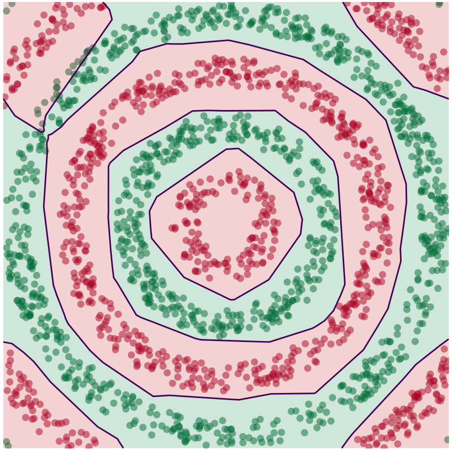
Learning a New Representation



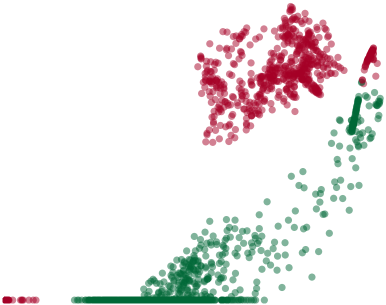


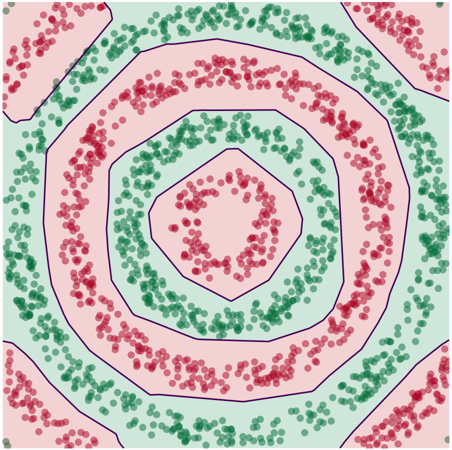
Learning a New Representation



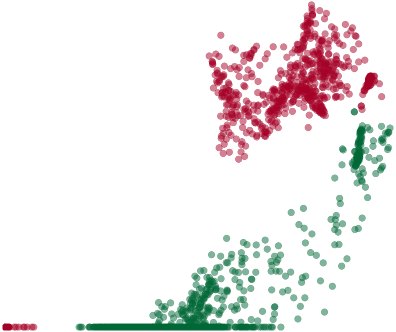


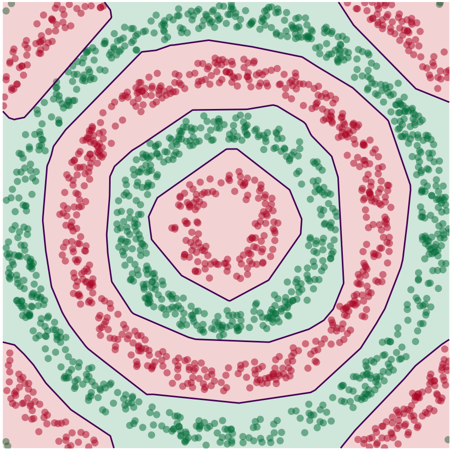
Learning a New Representation



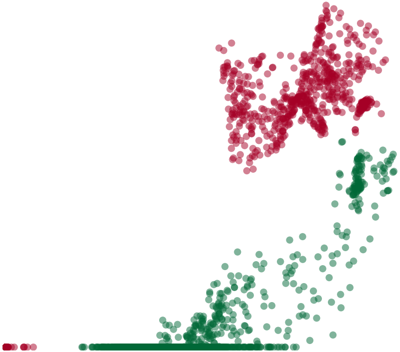


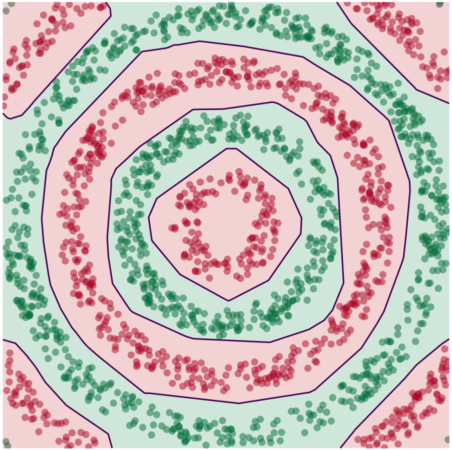
Learning a New Representation



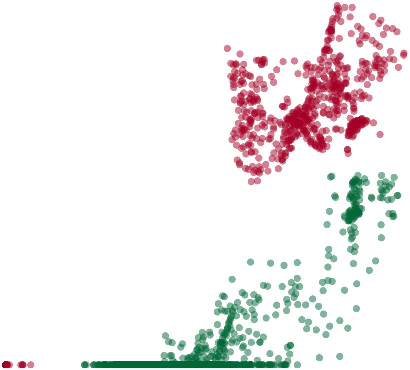


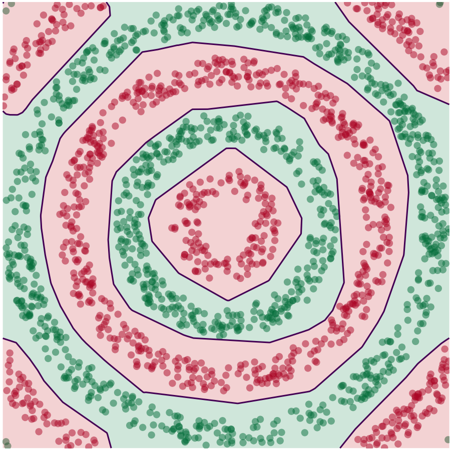
Learning a New Representation



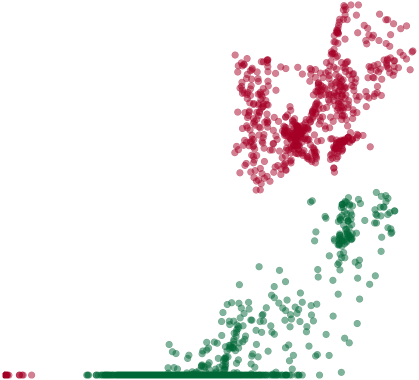


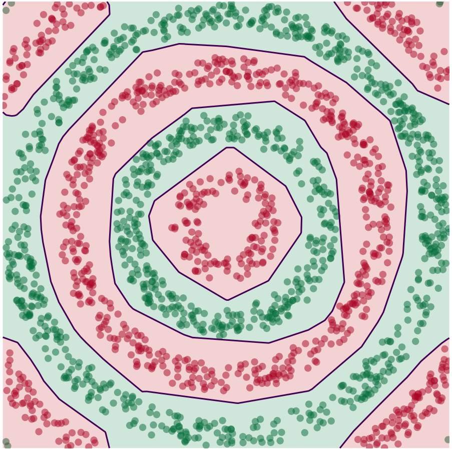
Learning a New Representation



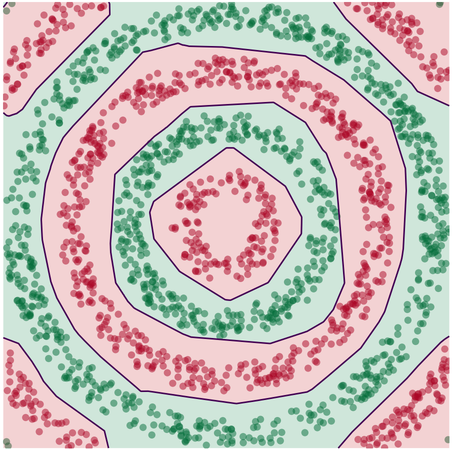


Learning a New Representation

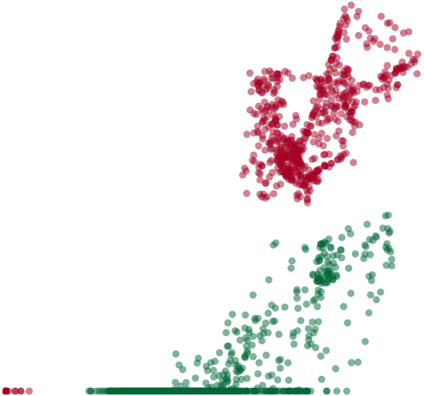


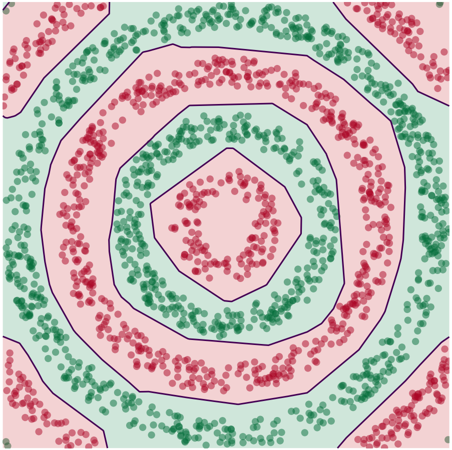


Learning a New Representation

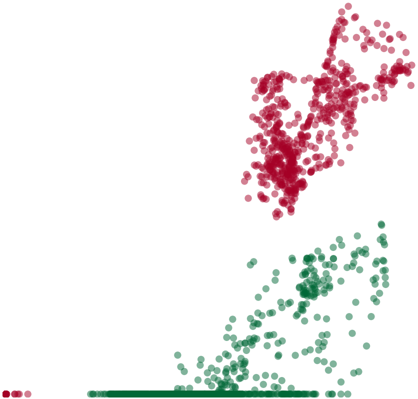


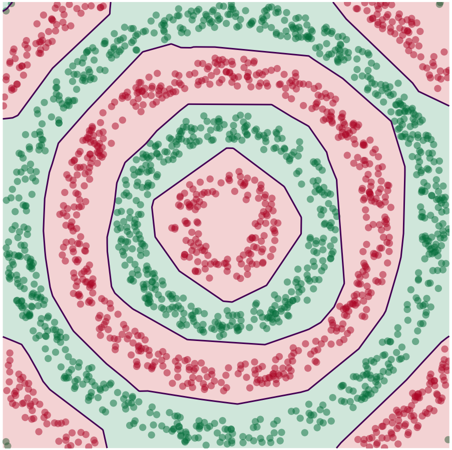
Learning a New Representation



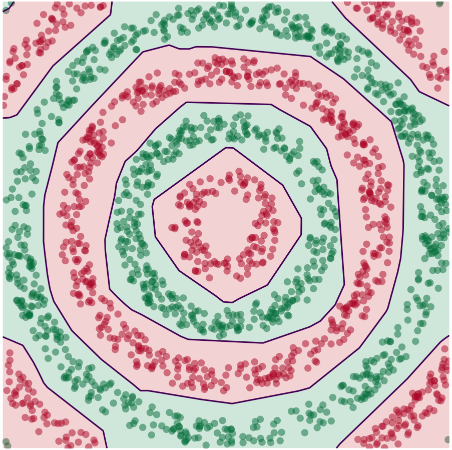


Learning a New Representation

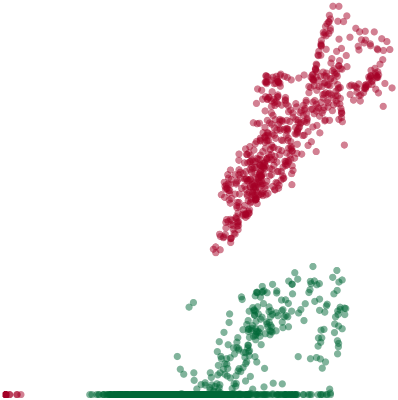


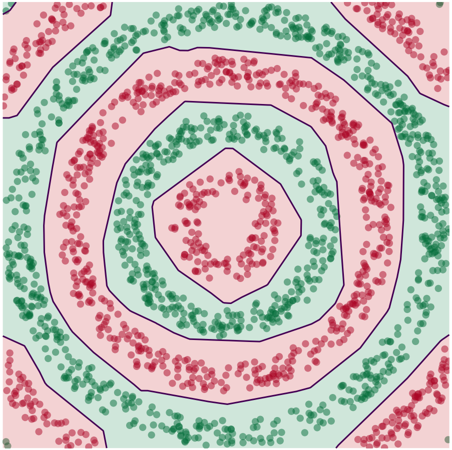


Learning a New Representation

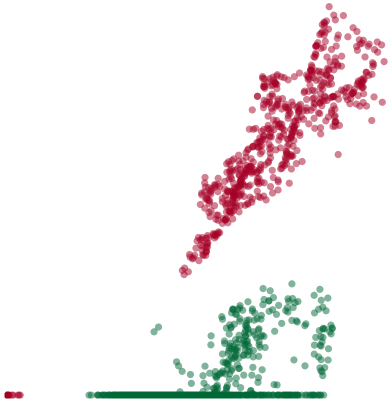


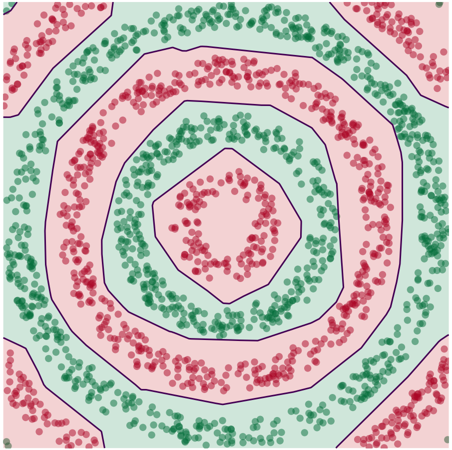
Learning a New Representation



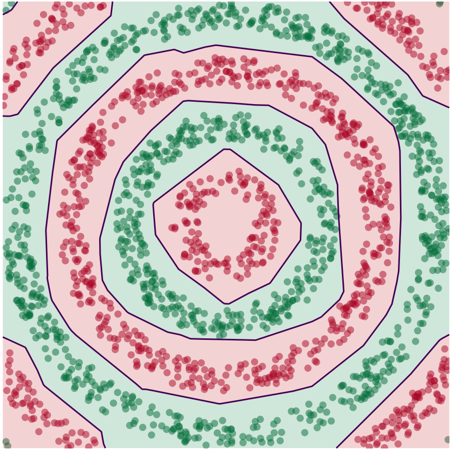


Learning a New Representation

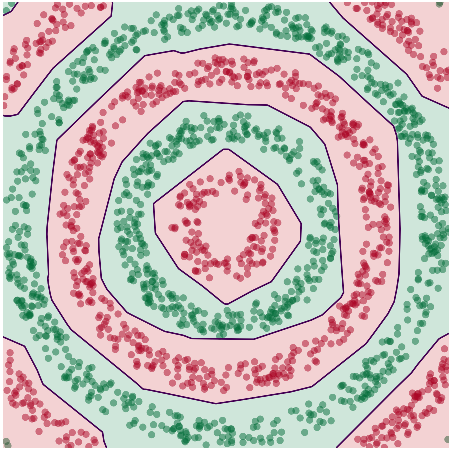




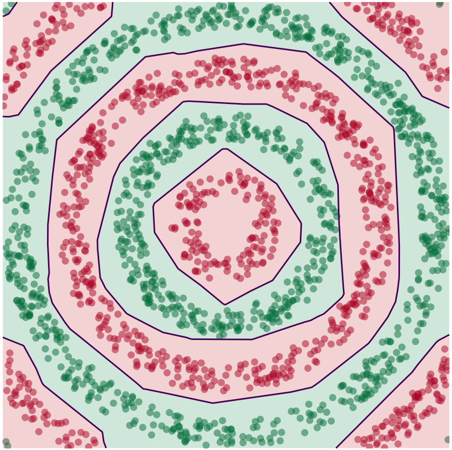
Learning a New Representation



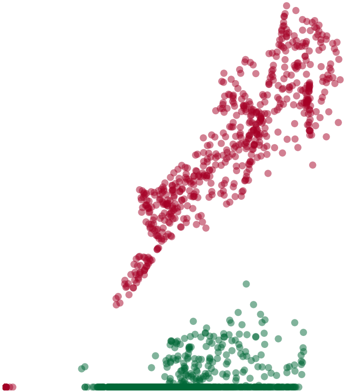
Learning a New Representation

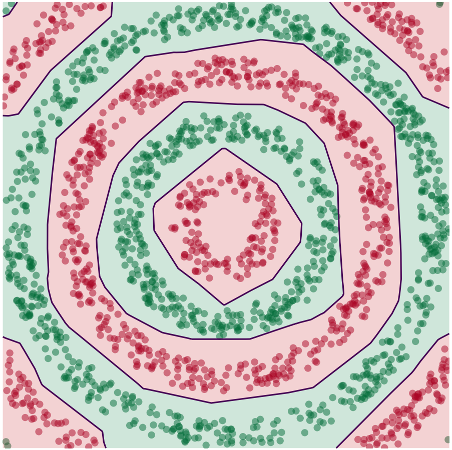


Learning a New Representation

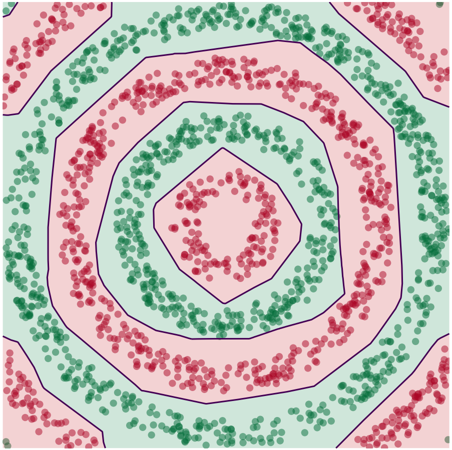


Learning a New Representation

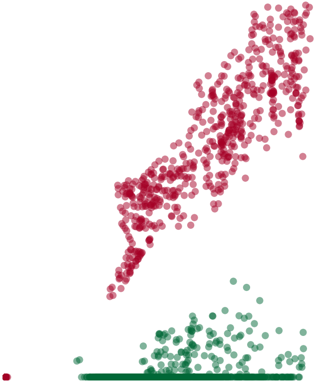


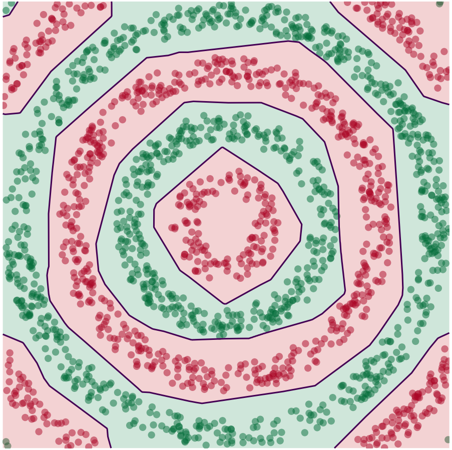


Learning a New Representation

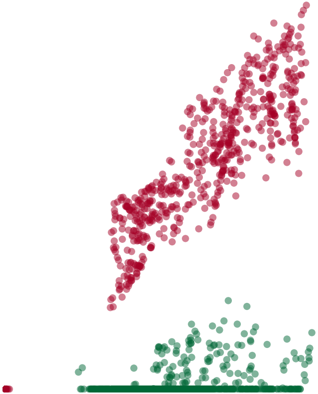


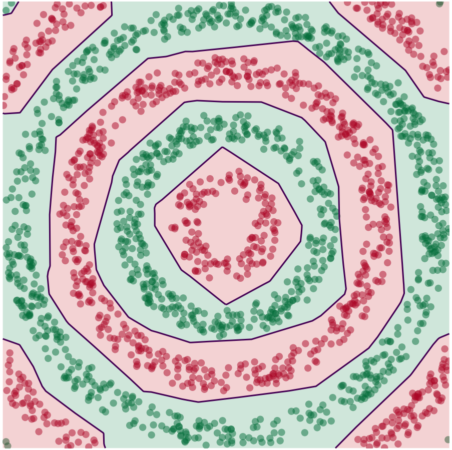
Learning a New Representation





Learning a New Representation





Deep Networks and Approximation

- Deep networks are also universal approximators.
- May require fewer nodes and/or parameters than single hidden layer.
- I.e., there exist functions which require an exponential number of nodes to approximate with a single hidden layer, but not with several layers.

Challenges

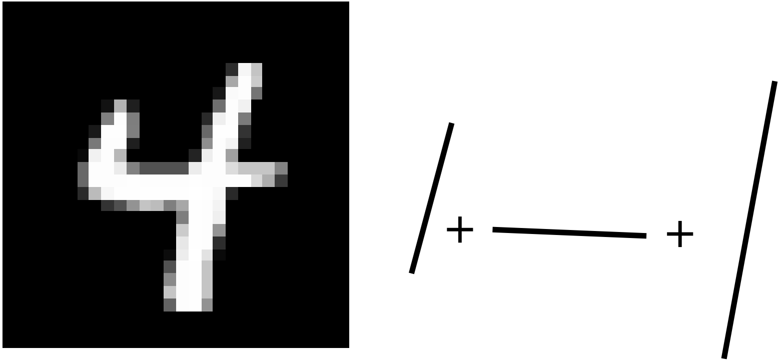
- The deeper the network, the weaker the gradient gets.
- Very non-convex!
- Deeper networks are harder to learn.

DSC 190

Machine Learning: Representations

Lecture 15 | Part 4

Convolutions

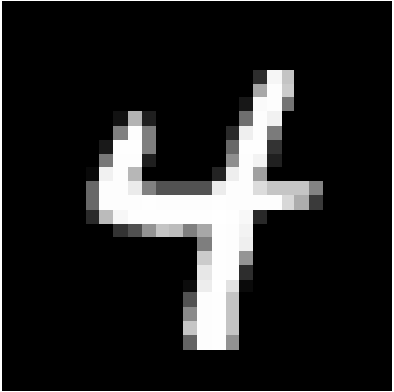


From Simple to Complex

- Complex shapes are made of simple patterns
- The human visual system uses this fact
- Line detector → shape detector → … → face detector
- Can we replicate this with a deep NN?

Edge Detector

- How do we find **vertical edges** in an image?
- One solution: **convolution** with an **edge filter**.

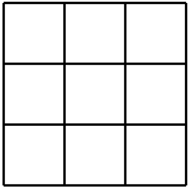


Vertical Edge Filter

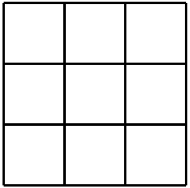
Idea

- Take a patch of the image, same size as filter.
- Perform “dot product” between patch and filter.
- If large, this is a (vertical) edge.

image patch:

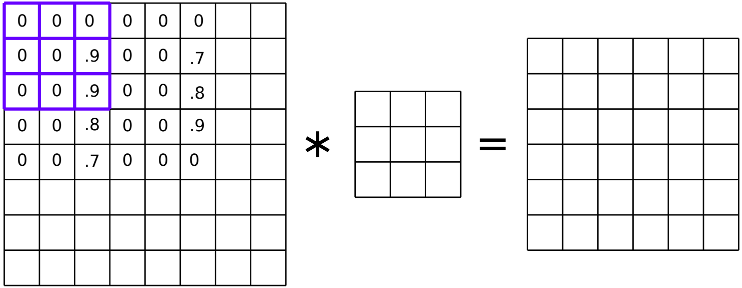


filter:



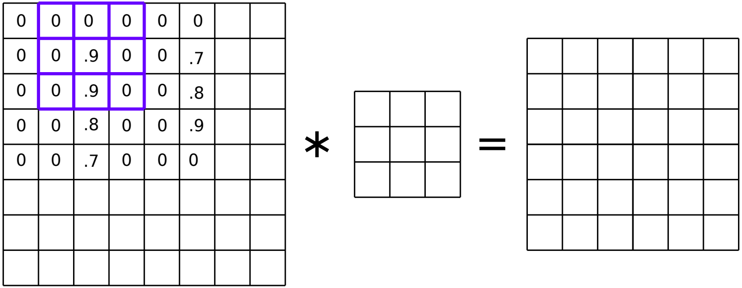
Idea

- Move the filter over the entire image, repeat procedure.



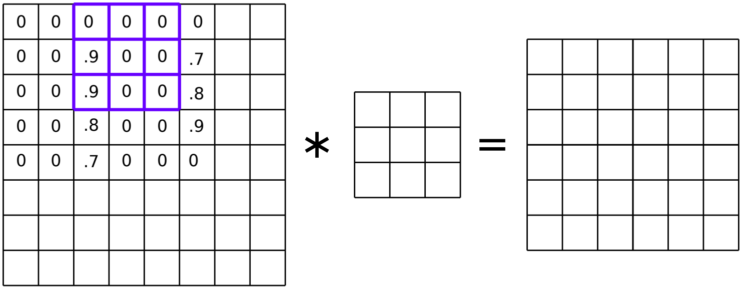
Idea

- Move the filter over the entire image, repeat procedure.



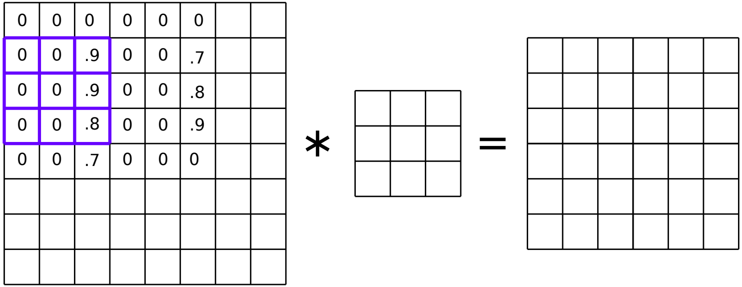
Idea

- Move the filter over the entire image, repeat procedure.



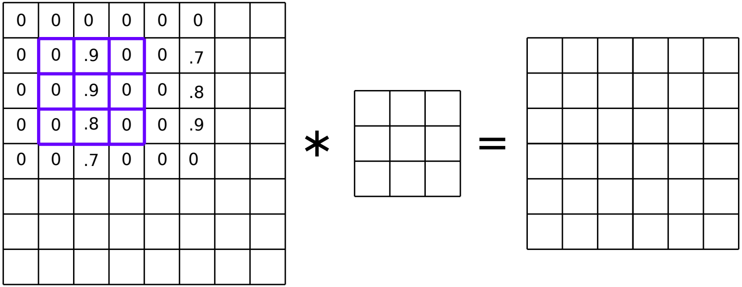
Idea

- Move the filter over the entire image, repeat procedure.



Idea

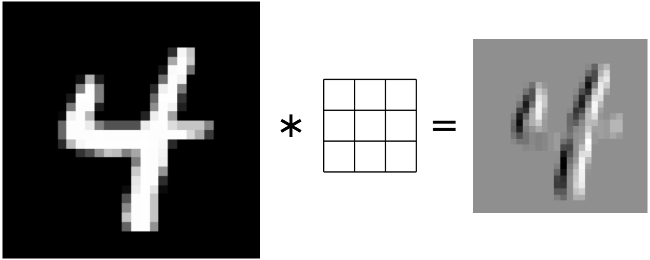
- Move the filter over the entire image, repeat procedure.



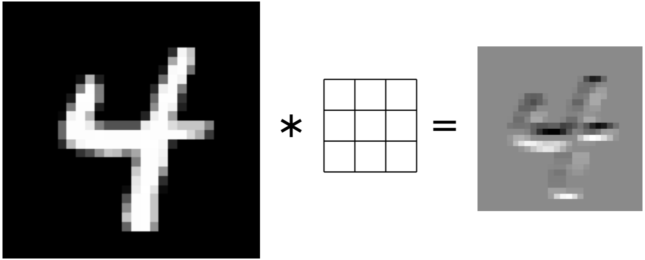
Convolution

- The result is the (2d) **convolution** of the filter with the image.
- Output is also 2-dimensional array.
- Called a **response map**.

Example: Vertical Filter



Example: Horizontal Filter



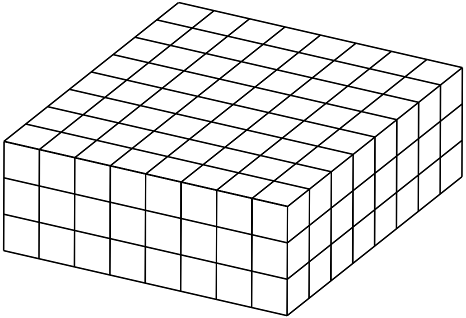
More About Filters

- Typically 3×3 or 5×5.
- Variations: different **stride**, image **padding**.

3-d Filters

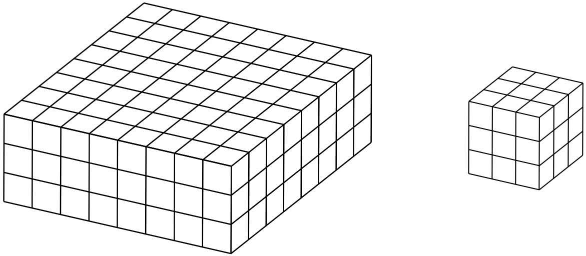
- Black and white images are 2-d arrays.
- But color images are 3-d arrays:
  - a.k.a., **tensors**
  - Three color **channels**: red, green, blue.
  - height × width × 3
- How does convolution work here?

Color Image

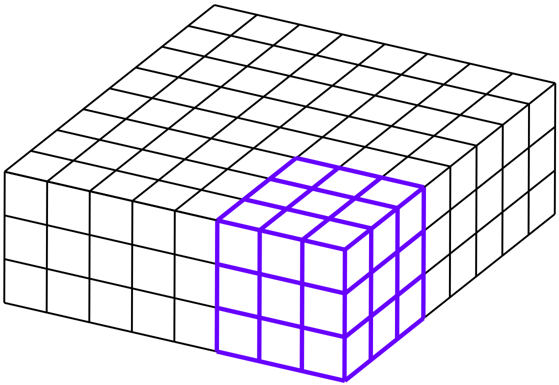


3-d Filter

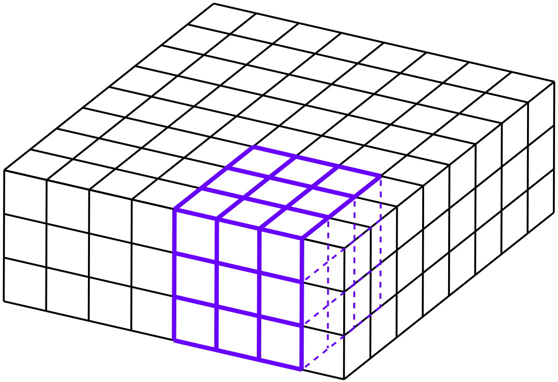
- The filter must also have three channels:
  - $3 \times 3 \times 3$, $5 \times 5 \times 3$, etc.



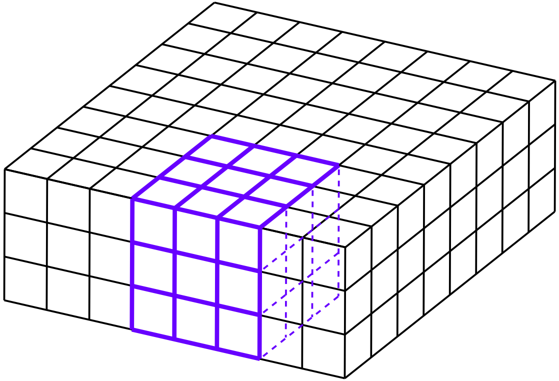
3-d Filter



3-d Filter



3-d Filter



Convolution with 3-d Filter

- Filter must have same number of channels as image.
  - 3 channels if image RGB.
- Result is still a 2-d array.

General Case

- Input "image" has k channels.
- Filter must have k channels as well.
  - e.g., $3 \times 3 \times k$
- Output is still $2-d$