How to Train Neural Networks

Seth D Temple, PhD
Schmidt AI in Science Postdoc

Agenda (exercises throughout)

1. Generate training data
    a. Modularize simulation code
    b. Data splitting
2. Architectures
    a. Linear layers
    b. Convolutional neural network
    c. Recurrent neural networks
3. Neural networks as objects
4. Training loops
5. Evaluation

https://github.com/sdtemple/zootopia3

Set up your environment

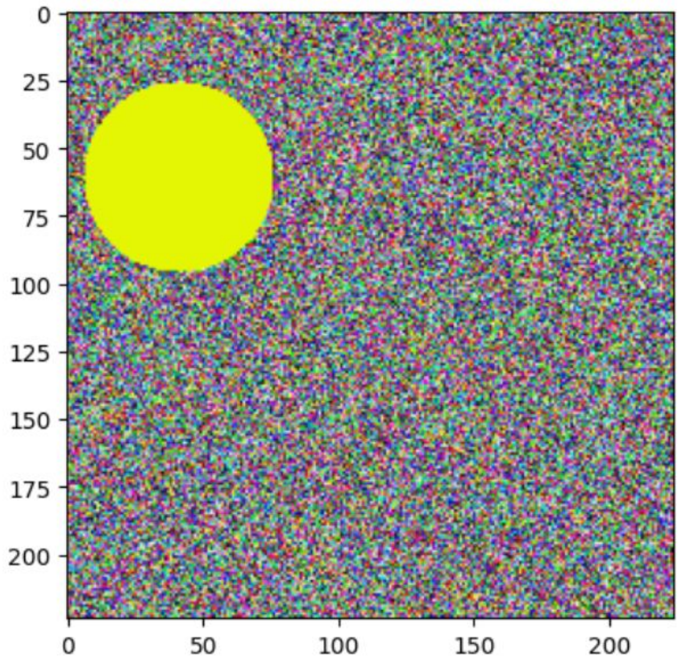
1. Fork the code at github.com/sdtemple/zootopia3
2. Move to IDE and terminal
   a. git clone https://github.com/youraccount/zootopia3.git
   b. Set up and activate environment
      (should be completed in tutorial 1)
3. Peruse the contents of the repository
   a. src/ : the Python package
   b. examples/ : where we will develop models
   c. data/ : where we will store code
4. Turn off AI coding assistant
   a. In VS Code, Ctrl + , then search "editor.inlineSuggest.enabled"

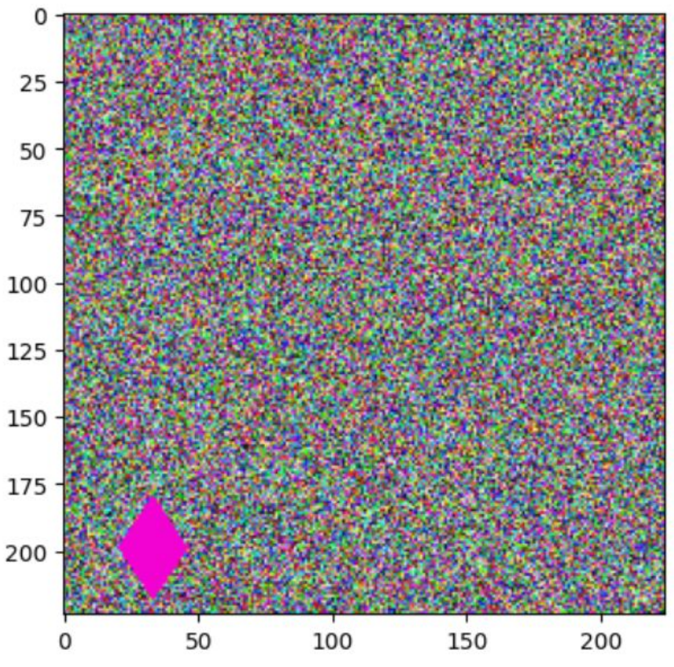
API references

- https://docs.pytorch.org/docs/stable/pytorch-api.html
- https://numpy.org/doc/stable/reference/
- https://scikit-learn.org/stable/api/sklearn.model_selection.html
- https://scikit-learn.org/stable/api/sklearn.preprocessing.html
- https://matplotlib.org/stable/api/pyplot_summary.html

Training data

The main task: classify shapes (4) and colors (6 - 8)





Simulate training data

- answers/simulate-exercise.ipynb
- Modify the parameters to create data
  - Use different file suffixes for different data simulations
  - Start with a small number of simulated data examples
  - After getting a working model, scale up the data size

Neural networks as objects

Models via object-oriented programming

- Neural network is defined as a **Python class** that inherits from `torch.nn.Module`.
- **Encapsulation:** parameters, layers, and forward logic are bundled together
  - Modularity and reusability
- `__init__`: defines layers and components.
- `__forward__`: defines how data flows through them.
- **Composability:** layers are objects
  → combine them to build complex architectures.

The __init__ function

- Declares what the object is!
- Constructor: runs when a model object is created
- Defines and stores the model’s components and hyperparameters
- Establishes the internal state of the model

super() within the method

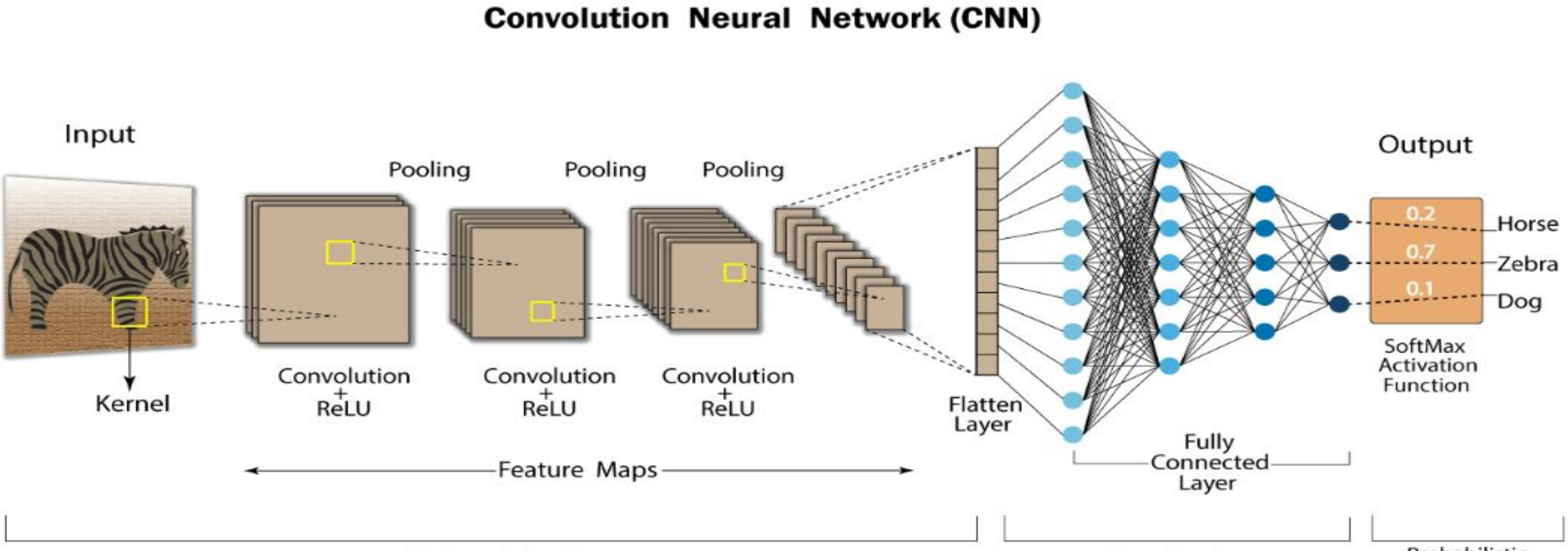
- Calls the parent class constructor torch.nn.Module
- Ensures inherited functionality is properly initialized
- Required for framework features (e.g., parameter tracking)

self

- class is a blueprint, but self stores the specific instance
- self.attribute =
  - Stores data on the object
  - Makes it accessible to other methods
- You always pass self into the class methods!
  - def forward(self, x):
  - def do_something(self, *args, **kwargs)
- Define attributes for the model architecture
  - self.conv1 = nn.Conv2d()
  - self.linear1 = nn.Linear()

The forward(self, x) function

- Specifies the computation graph
    - How inputs are transformed into outputs
    - By invoking the self layer attributes
    - This is called when you run ` model(x) `

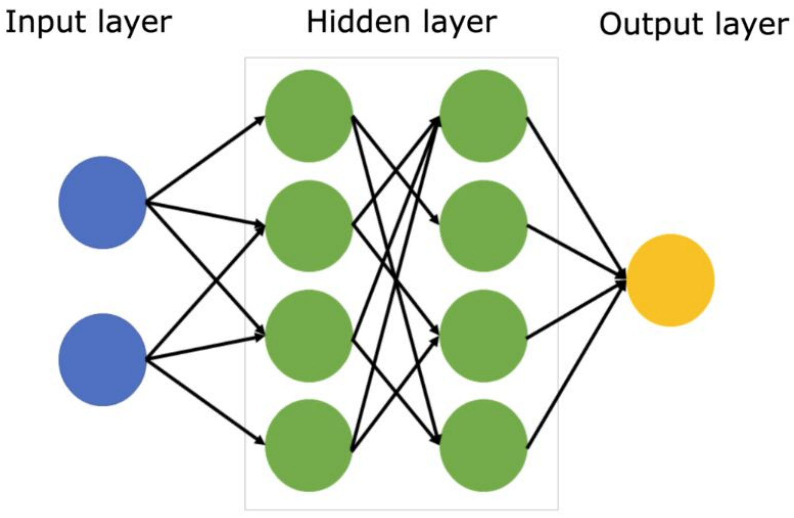


Architectures

examples/modeling-exercise-1.ipynb
examples/modeling-exercise-2.ipynb

Linear() layer

- All-to-all mapping
  - The most amount of parameters!
  - Don’t make this too large
- Dropout and/or batch normalization possible for all but last layer!
- Have to specify the first dimension, which changes in CNNs

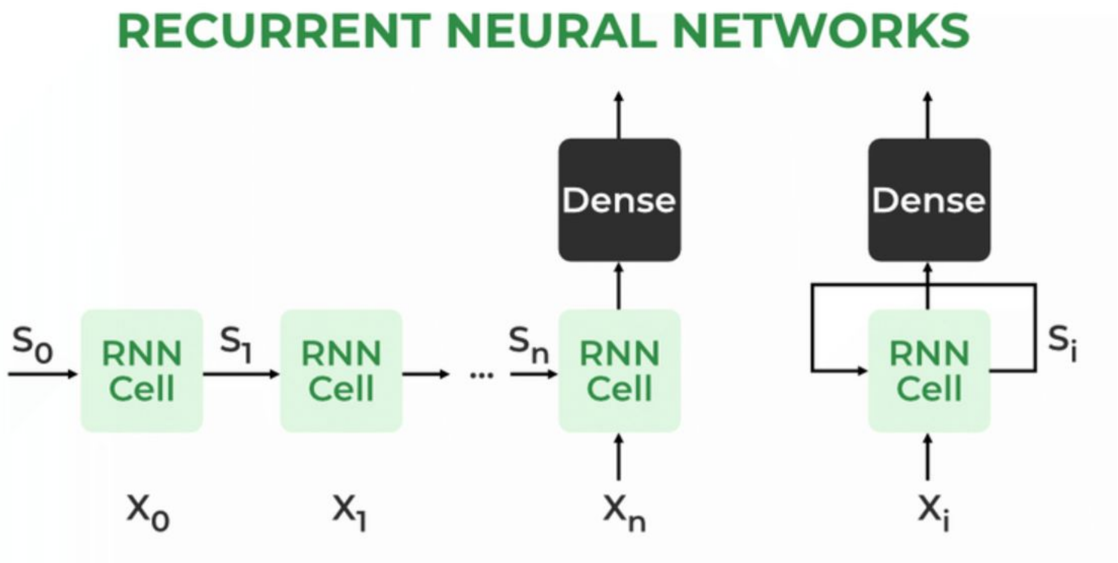


Artificial neural networks

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

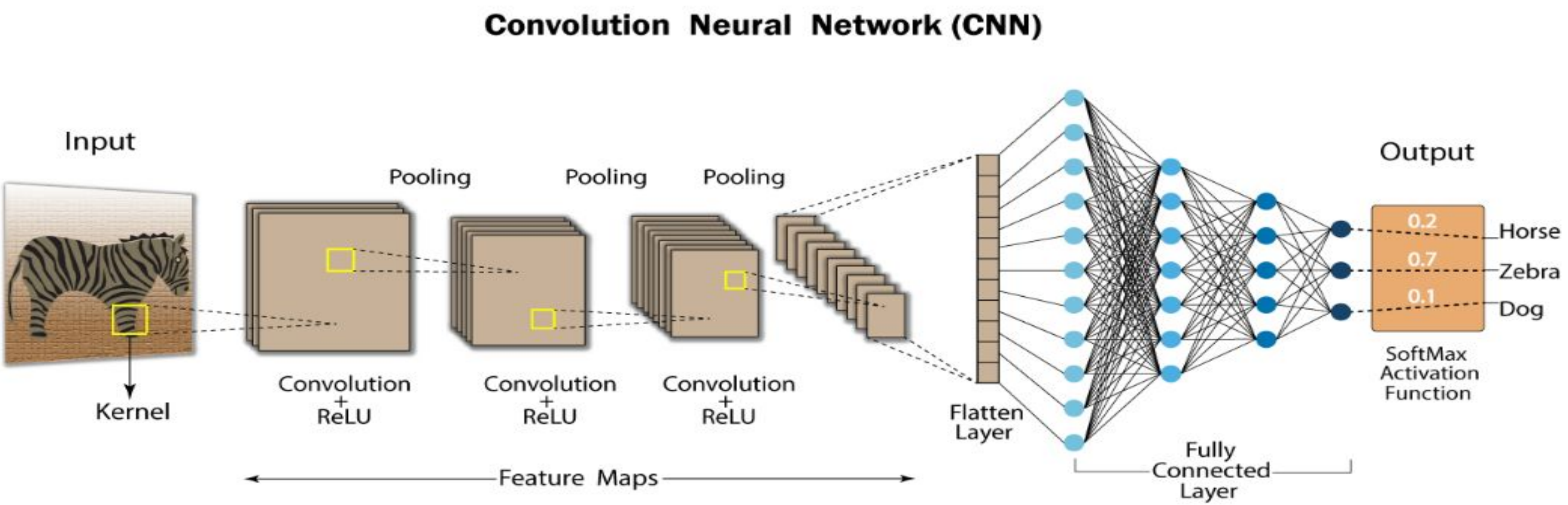
RNN() or LSTM() layer

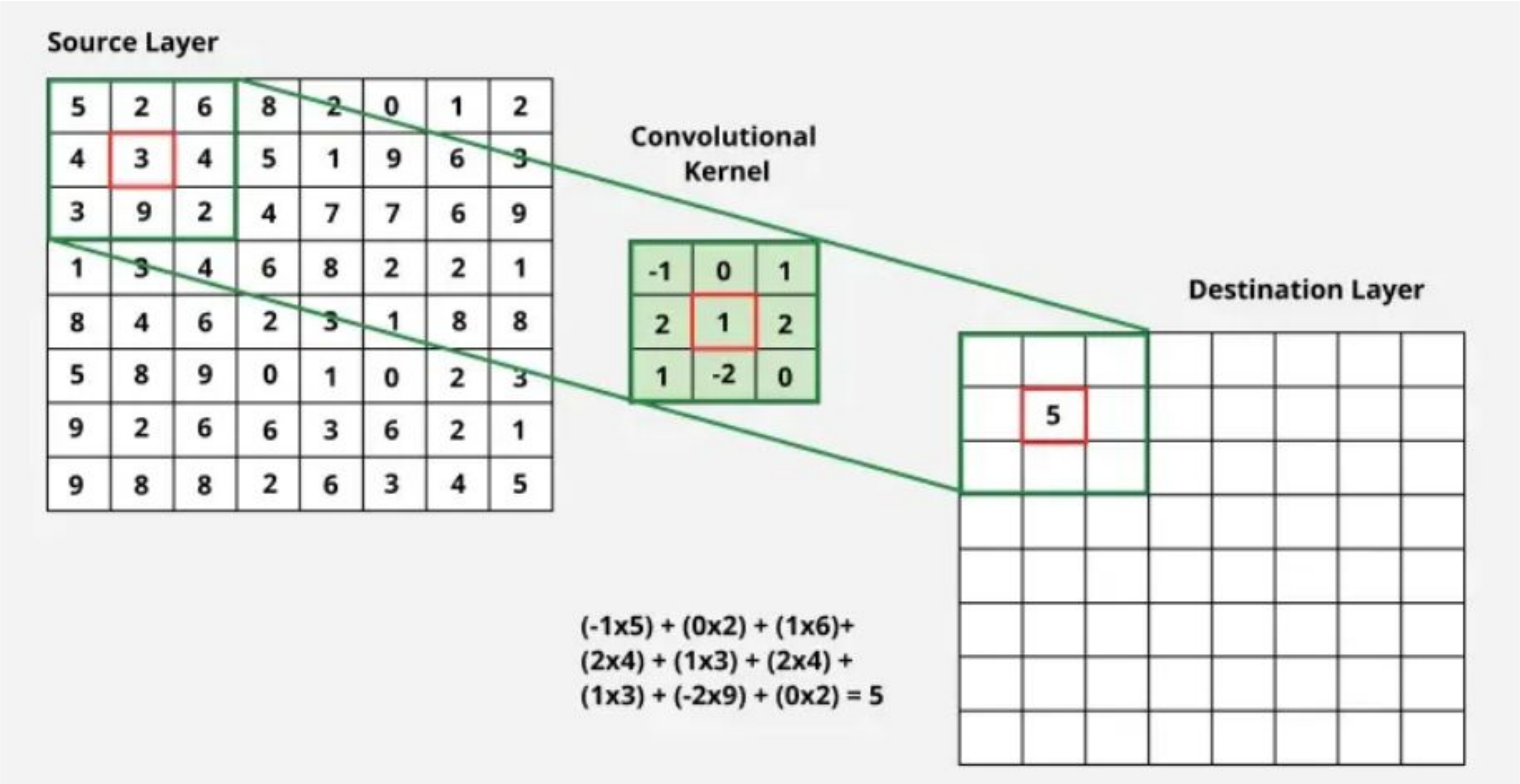
- For sequential data
- Slower to train
- Careful about initializing hidden and cell states
- Careful about input dimensions



Conv2d() layer

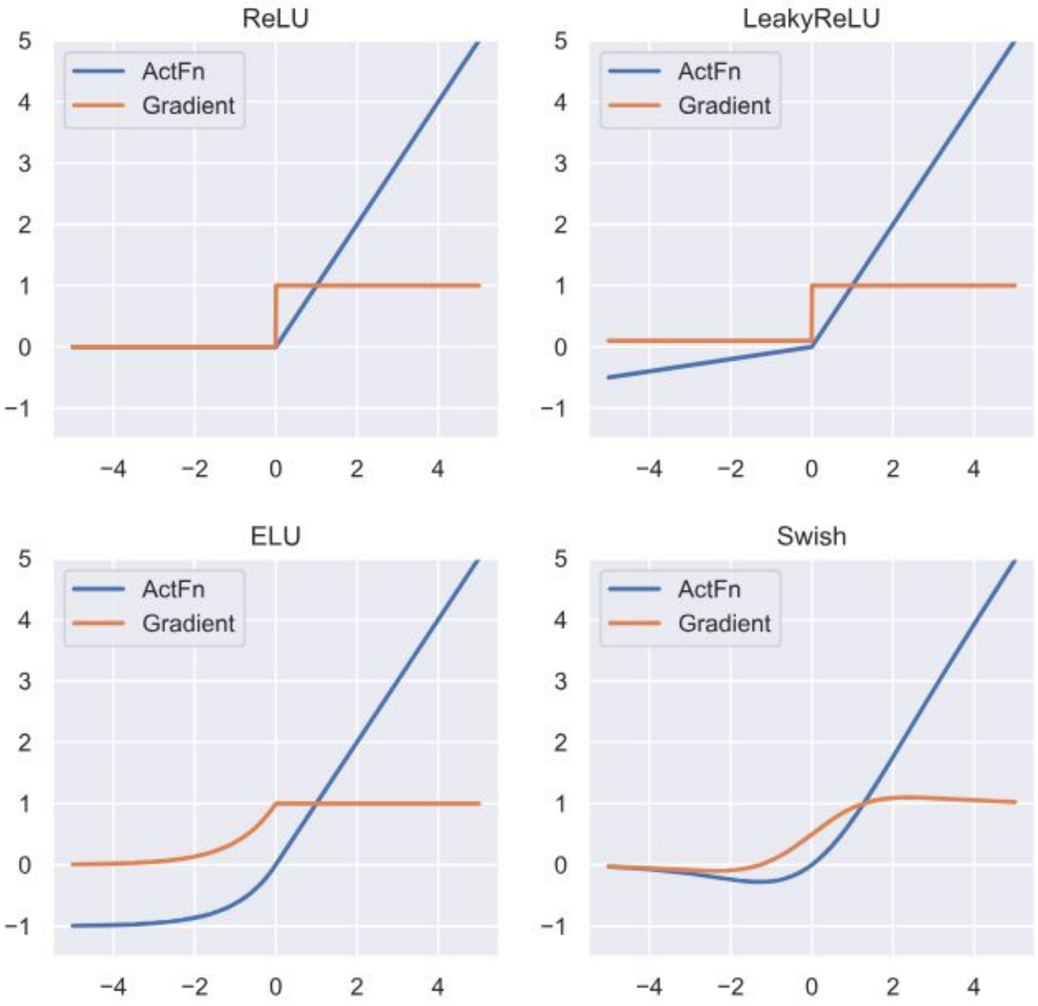
- Input channels (e.g., RGB colors)
- Output channels
  - You often increase from initial (3) channels
  - For instance, 3 -> 32 -> 32 -> Linear()
- kernel_size, stride, padding
- Pooling and normalization layers follows



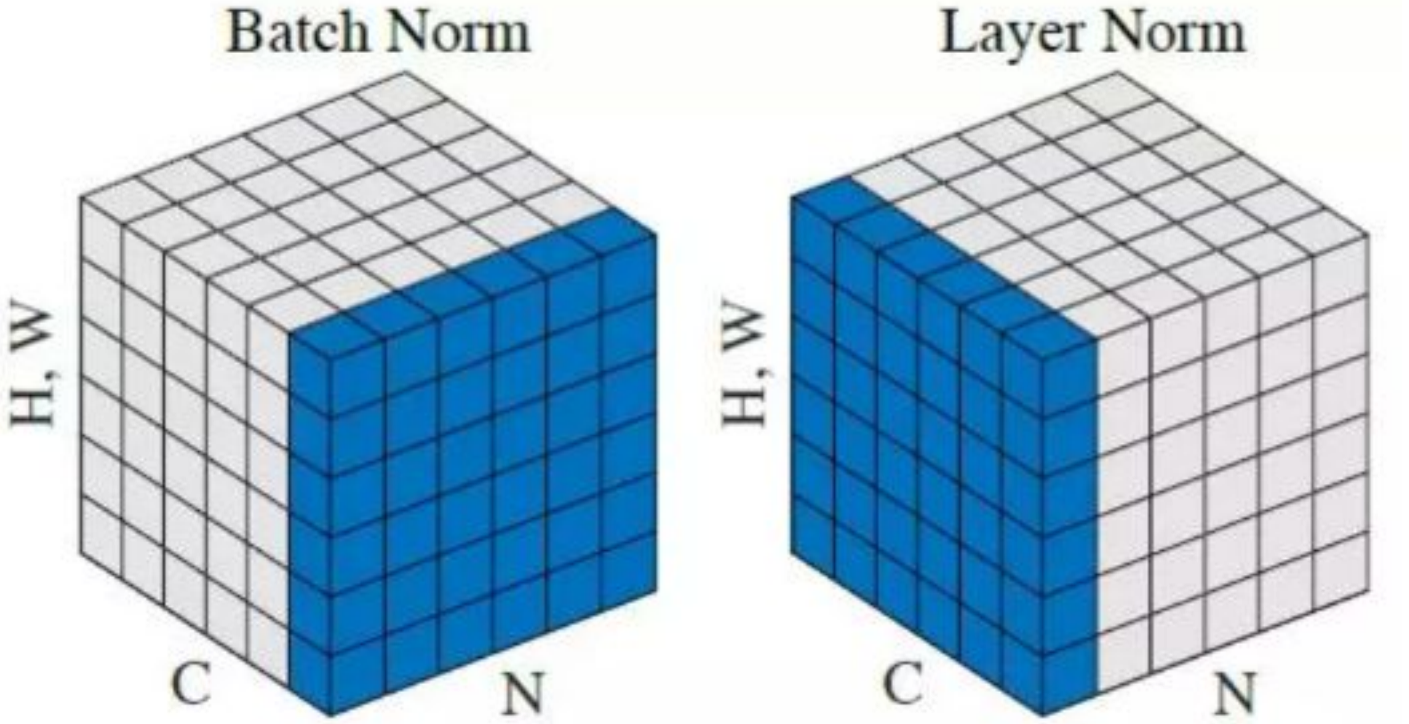

Source Layer
Convolutional Kernel
Destination Layer
(-1x5) + (0x2) + (1x6)+
(2x4) + (1x3) + (2x4) +
(1x3) + (-2x9) + (0x2) = 5

Other layers

- Pooling (same math as Conv2d):
  - MaxPool2d()
  - AvgPool2d()
- Activations
  - https://docs.pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions
- Dropout()
  - This is important to make inferences generalizable past training
- LayerNorm1d()
- BatchNorm2d()
  - Normalization layers can be super important!
  - Often your model will crash without them


ReLU
LeakyReLU
ELU
Swish
ActFn
Gradient
5
4
3
2
1
0
−1
−4
−2
0
2
4

Normalization visualization



Training loop

Standard way to define your training loop

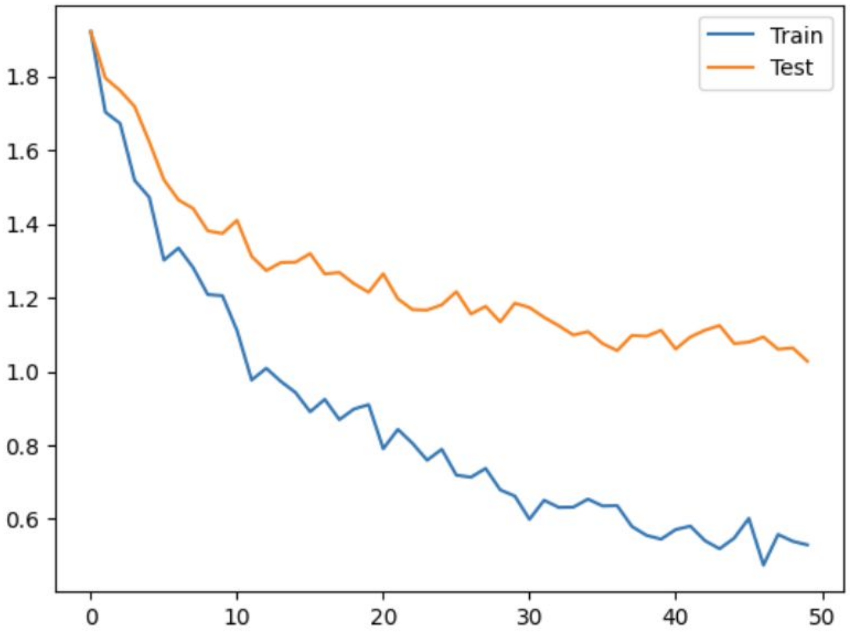
1. for epoch in range(num_epochs)
    a. How many times to pass over the data
2. **model.train()** : turns on the learning state!
3. **for** x, y in **train_loader**:
4. **optimizer.zero_grad()**
5. Invoke ` yhat = model(x) `
6. Invoke ` criterion(y, yhat) `
    a. Care about dimensions of x and model output
    b. Care about data being on the GPU device
7. **loss.backward()**
8. **optimizer.step()**

Core training steps

- `zero_grad()` clears stored gradients from the previous iteration
- `loss.backward()` computes gradients via backpropagation
- `optimizer.step()` updates parameters using those gradients
- These steps implement one optimization iteration
- Order matters and is repeated every batch

Diagnosing issues

- Loss not decreasing
  - Print the per-epoch loss with a running loss from batches
- Training–validation gap
  - Train - test - and validate
  - Keep track of validation loss in model.eval() state
- Exploding / vanishing gradients
  - This number should not be too small or large `print(next(model.parameters()).grad.abs().max())`



Evaluation
(on validation data)

Metrics for categorical classifier

- Accuracy: overall fraction of correct predictions
- Confusion matrix: per-class error structure
  https://en.wikipedia.org/wiki/Confusion_matrix
- Precision & recall: false positives vs. negatives
- ROC–AUC / PR–AUC:
  threshold-independent performance

Confusion Matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

Code snippet, or there is one in the notebooks

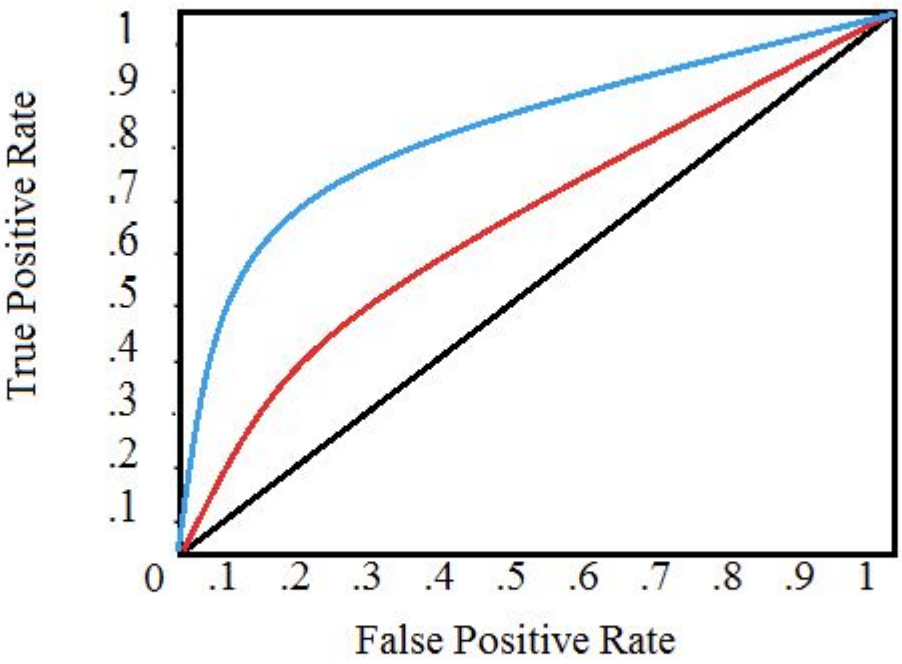
```
import matplotlib.pyplot as plt


def plot_confusion_matrix(df_confusion, title='Confusion matrix', cmap=plt.cm.gray_r):
    plt.matshow(df_confusion, cmap=cmap) # imshow
    #plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(df_confusion.columns))
    plt.xticks(tick_marks, df_confusion.columns, rotation=45)
    plt.yticks(tick_marks, df_confusion.index)
    #plt.tight_layout()
    plt.ylabel(df_confusion.index.name)
    plt.xlabel(df_confusion.columns.name)


df_confusion = pd.crosstab(y_actu, y_pred)
plot_confusion_matrix(df_confusion)
```

Receiver operator curve

- TPR & FPR as we **change hard threshold** of a probabilistic classifier
- Prediction line headed to upper left is good
- Similar idea for PR curve (Precision-Recall)



Demo: train big model on the cluster

The End

Acknowledgement:
- Michigan Institute for Data & AI in Society
- Schmidt Sciences