CMPT 733
Deep Learning (I)

Instructor Zhengjie Miao

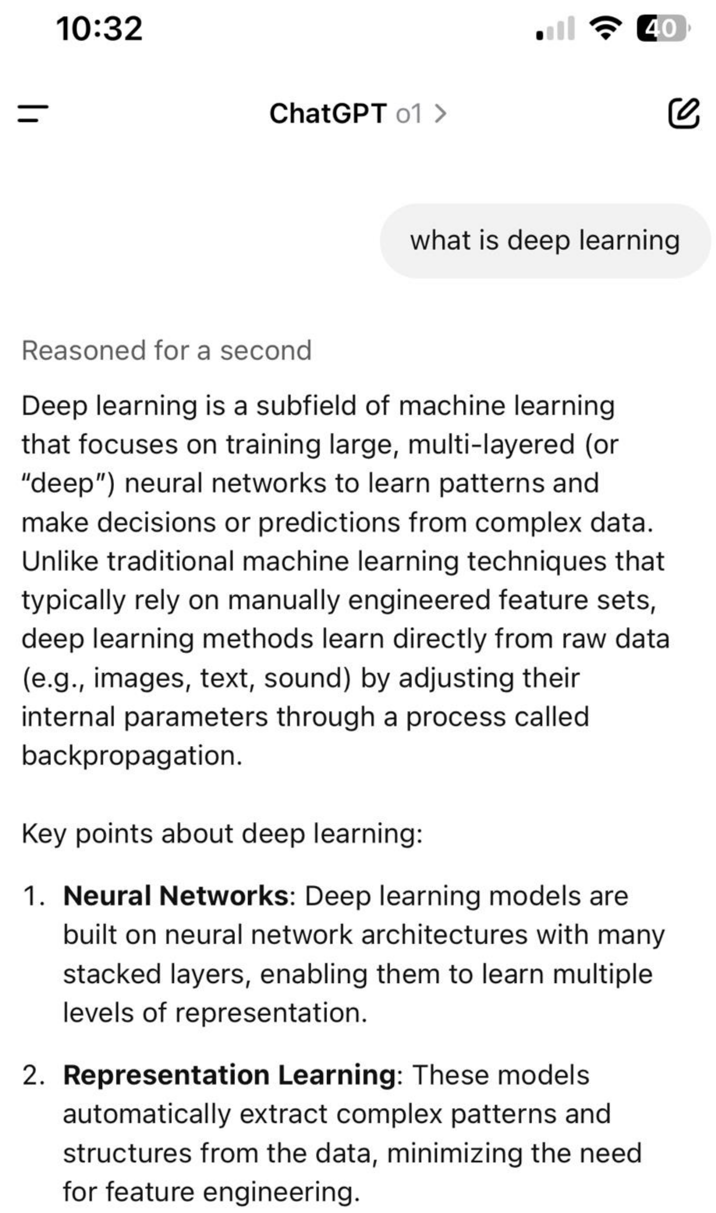
Course website https://coursys.sfu.ca/2025sp-cmpt-733-g1/pages/

DL Applications

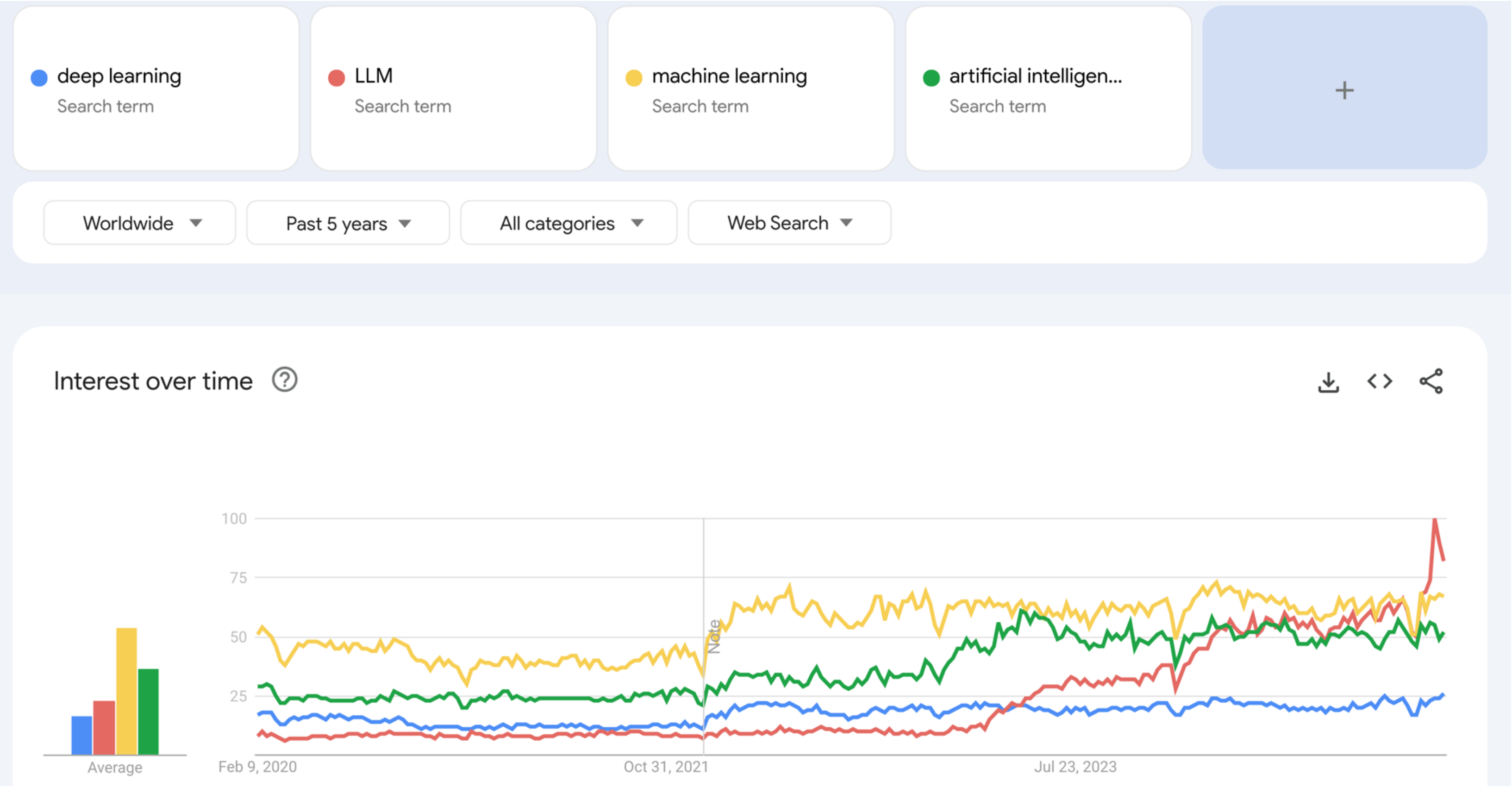
https://en.wikipedia.org/wiki/Amazon_Alexa

https://didyouknowbg8.wordpress.com/2024/02/24/yolov9-a-leap-forward-in-object-detection-performance/

DL Applications



DL & AI trends


deep learning
Search term
LLM
Search term
machine learning
Search term
artificial intelligen...
Search term
+
Worldwide
Past 5 years
All categories
Web Search
Interest over time
100
75
50
25
Note
Average
Feb 9, 2020
Oct 31, 2021
Jul 23, 2023

Overview

- Renaissance of artificial neural networks
  - ML recap
  - Representation learning vs feature engineering
- Background
  - Neural networks
  - Linear Algebra, Optimization
  - Regularization
  - Construction and training of layered models
- Frameworks for deep learning

LeCun, Hinton, Bengio: Deep Learning

- The idea of neural networks had been around for fifty years, but unsuccessful
- Major AI figures had trashed it, even proving that early versions had very limited expressiveness
- Instead, machine learning was based on other models, for example the support vector machine and graphical models. Neural networks did not perform well.

The New York Times

GIVE THE TIMES

Turing Award Won by 3 Pioneers in Artificial Intelligence

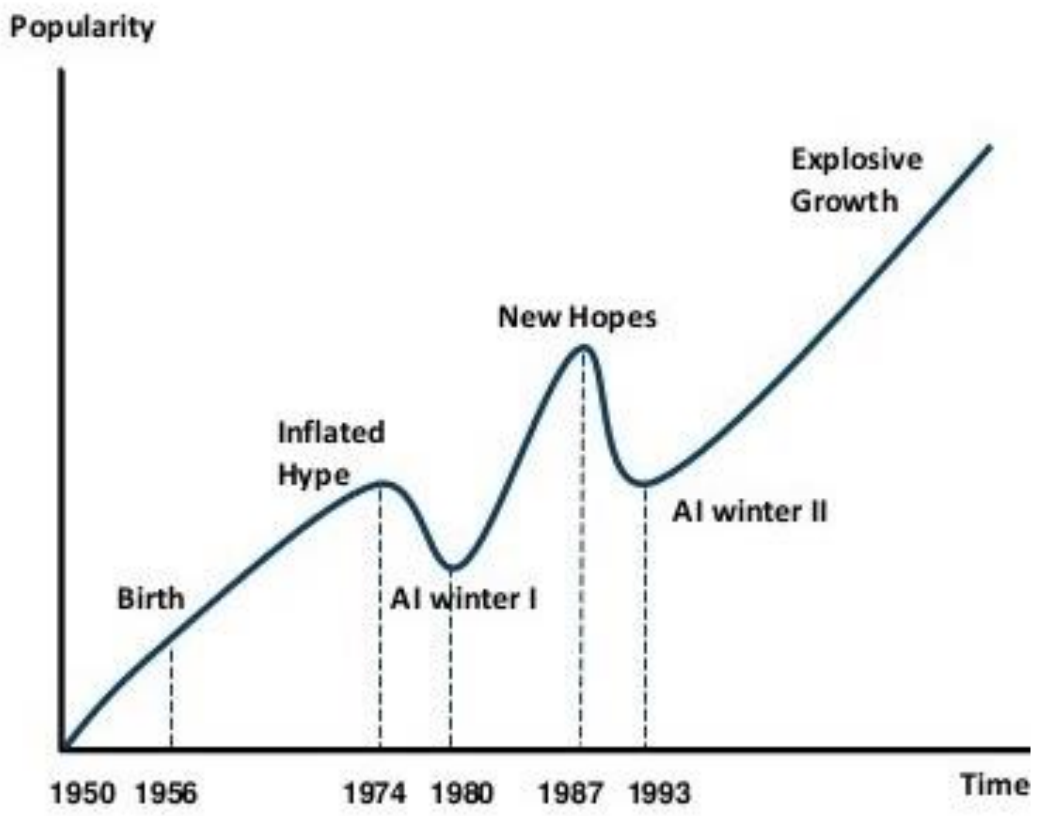
From left, Yann LeCun, Geoffrey Hinton and Yoshua Bengio. The researchers worked on key developments for neural networks, which are reshaping how computer systems are built.
From left, Facebook, via Associated Press; Aaron Vincent Elkaim for The New York Times; Chad Buchanan/Getty Images

By Cade Metz

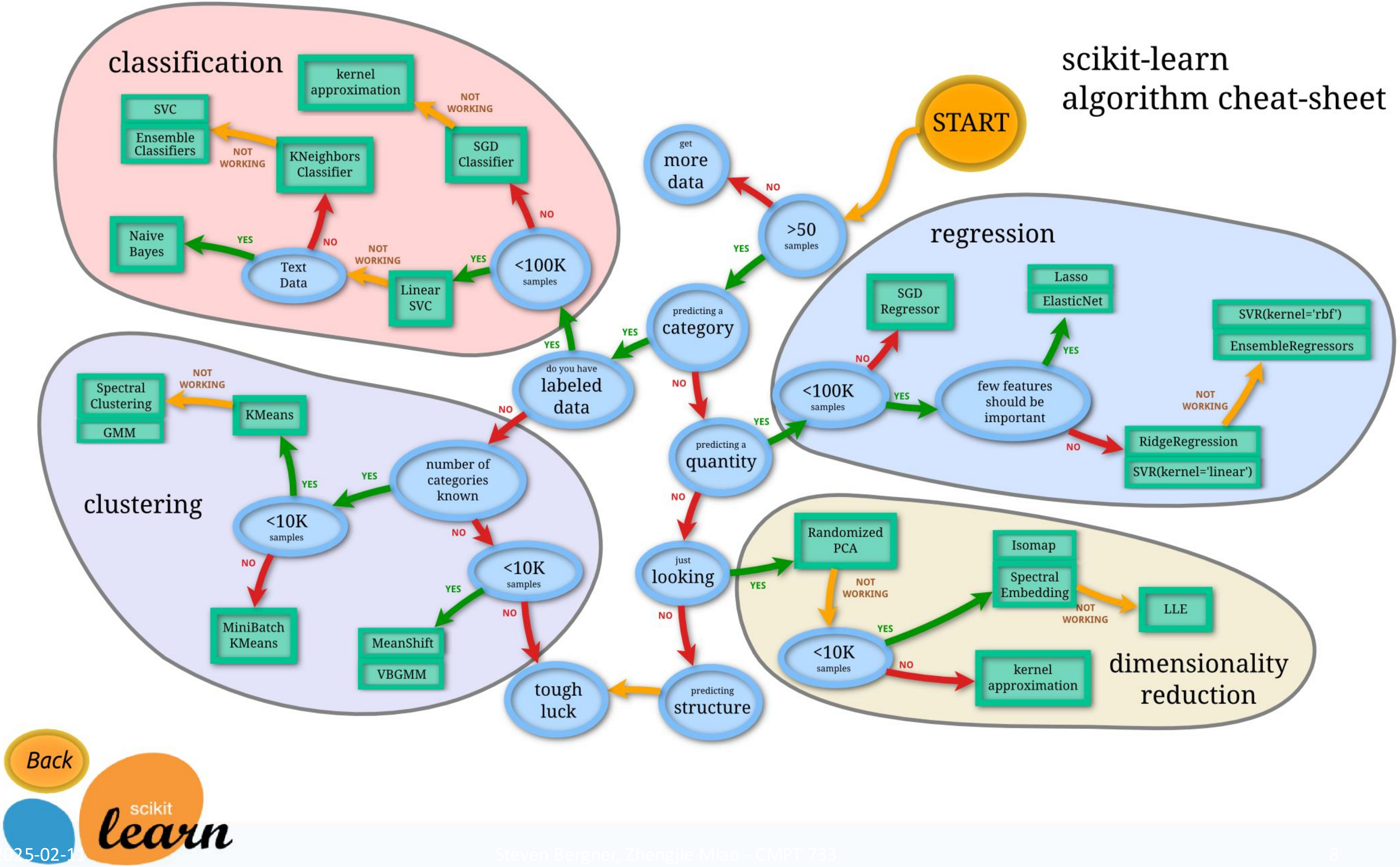
March 27, 2019

LeCun, Hinton, Bengio: Deep Learning

- **“No, let’s do it this way instead:”** these networks learn extremely complex functions, so they need much more data than existing ML approaches, GPUs to train, and algorithms to enable them to learn more effectively
- Around 2010, these models began smashing records in speech and image recognition



Machine Learning Tasks

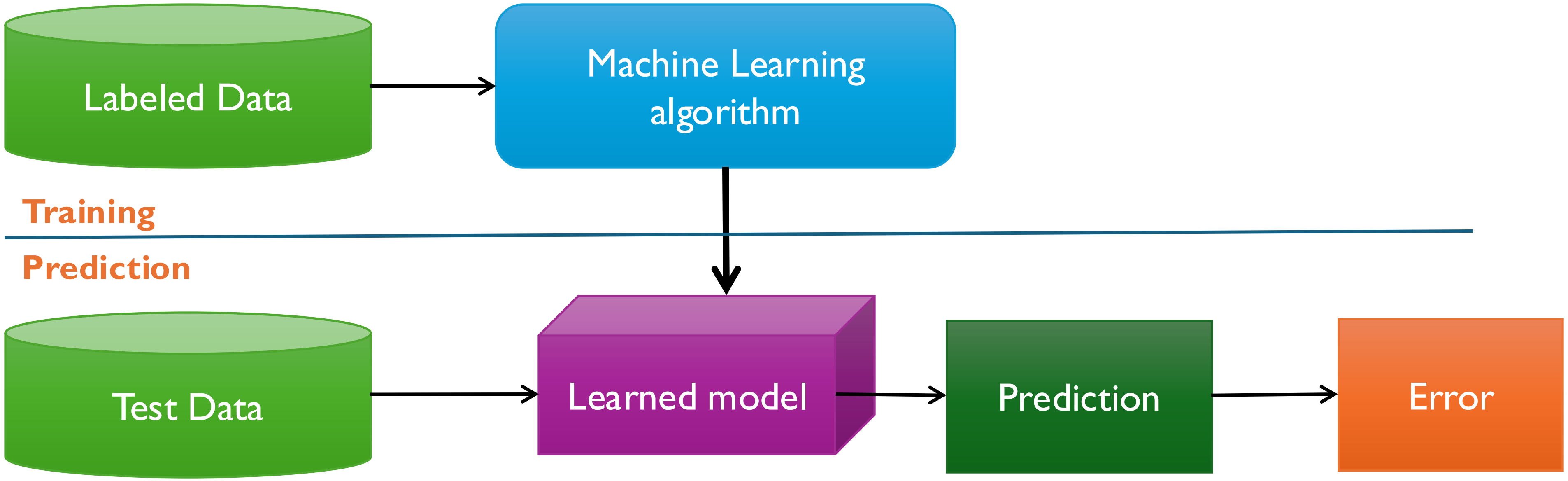


Recap: What is machine learning?

Mathematical principles and computer algorithms exploiting data

- for extracting what is general
- so as to be able to say something meaningful about unseen cases
- to identify which configurations of variables are plausible
- to generate new plausible configurations
- to learn to predict, classify, take decisions

Recap: Supervised Learning Setting



Example: Image Classification

= cat

= cat

= cat

= not cat

Example: Image Classification

- We'd like to learn a cat classifier, which is a function f from the input space to a class
  - In this example, input space = {pictures}, represented as a vector x of pixel values
  - class $\in \{0, 1\}$
- Ideally,
  - $f\left(\right.$  $\left.\right) = 1.$

Recap: Feature Extraction

Raw Data

```
1 in24.inetnebr.com - - [01/Aug/1995:00:00:01 -0400] "GET /shuttle/missions/sts-68/news/sts-68-mcc-05
2 uplherc.upl.com - - [01/Aug/1995:00:00:07 -0400] "GET / HTTP/1.0" 304 0
3 uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/ksclogo-medium.gif HTTP/1.0" 304 0
4 uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/MOSAIC-logosmall.gif HTTP/1.0" 304 0
```

Turning Raw Data into Connection Data

- A connection is a sequence of HTTP requests starting and ending at some well-defined times

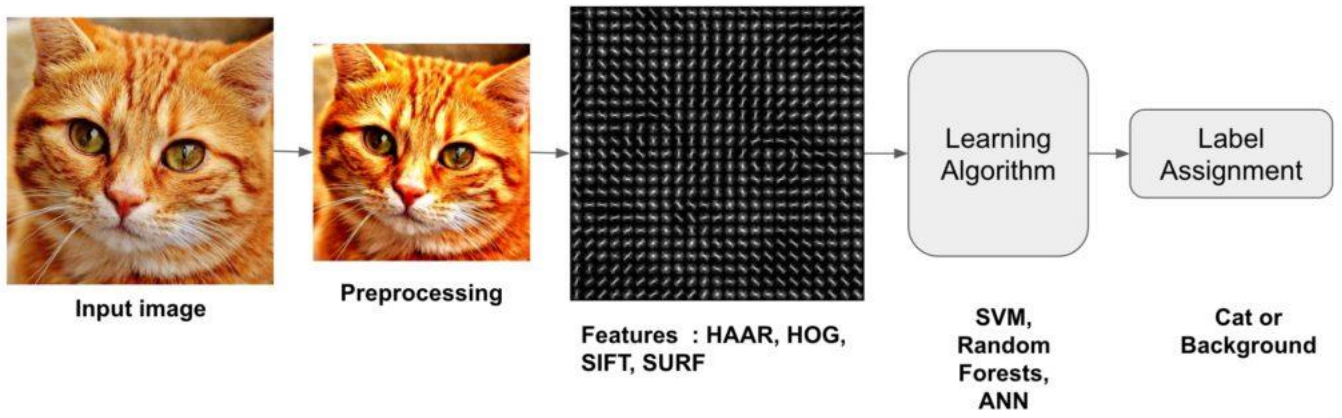
Turning Connection Data into Feature Vectors

- Requiring a fair bit of domain knowledge
- Asking yourself how to distinguish attacks from normal connections (e.g., number of failed login attempts, duration of the connection)

Feature Extraction for Cat Classification

Example Features:

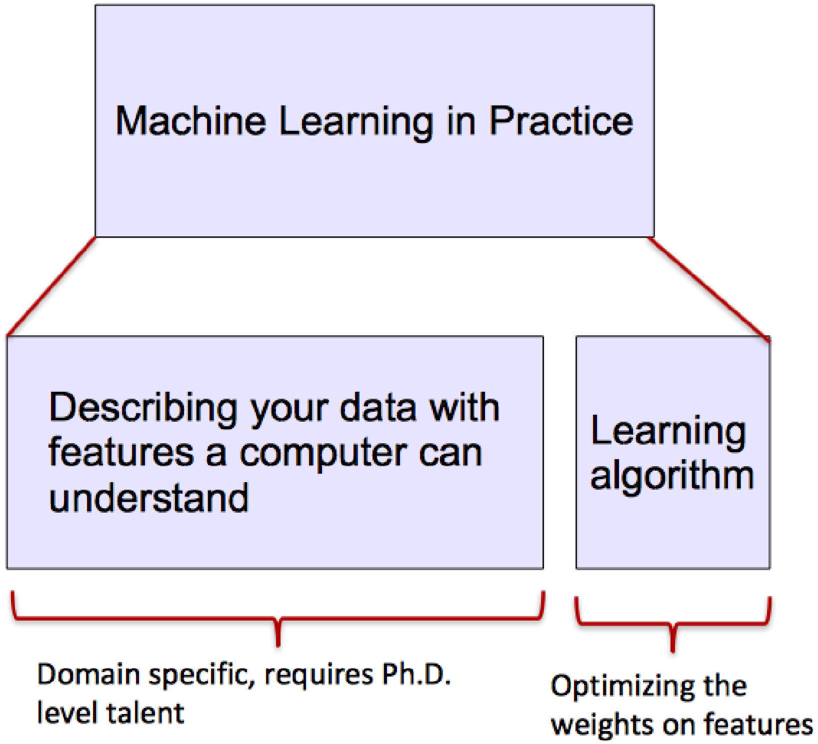
1. Shape-Based Features
   - Ear Shape
   - Face Shape
   - Body Proportions
2. Texture-Based Features
   - Fur Texture
   - Pattern Recognition
3. Color-Based Features
4. Facial Features
   - Eye Shape and Size
   - Nose Structure
5. Whisker Density and Placement



https://learnopencv.com/image-recognition-and-object-detection-part1/

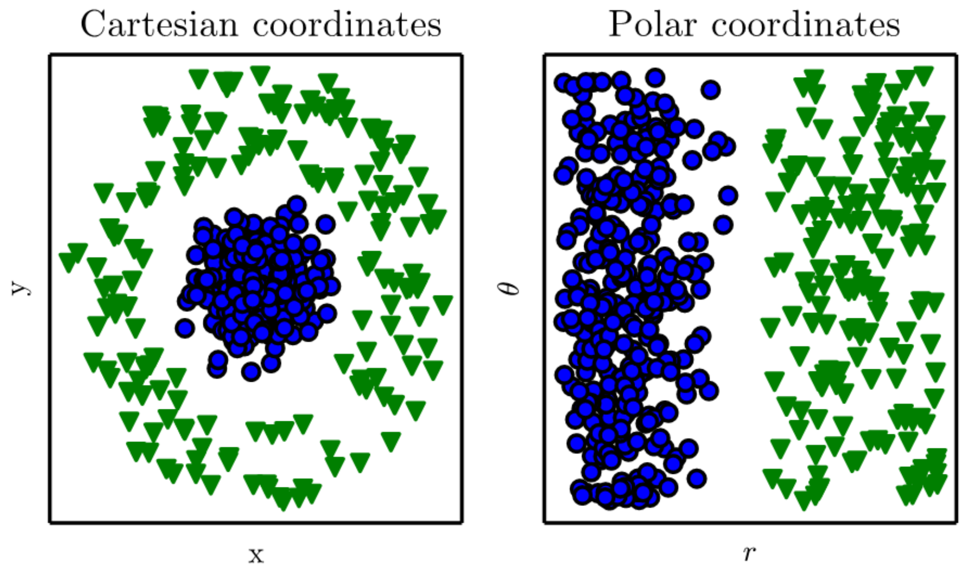
Classical ML vs. Deep Learning

- Many classical machine learning methods work well because of **human-designed input features/data representations**
- ML becomes just **optimizing weights** of the model to best make a final prediction (tuning)



Feature	NER
Current Word	✓
Previous Word	✓
Next Word	✓
Current Word Character n-gram	all
Current POS Tag	✓
Surrounding POS Tag Sequence	✓
Current Word Shape	✓
Surrounding Word Shape Sequence	✓
Presence of Word in Left Window	size 4
Presence of Word in Right Window	size 4

Representations matter



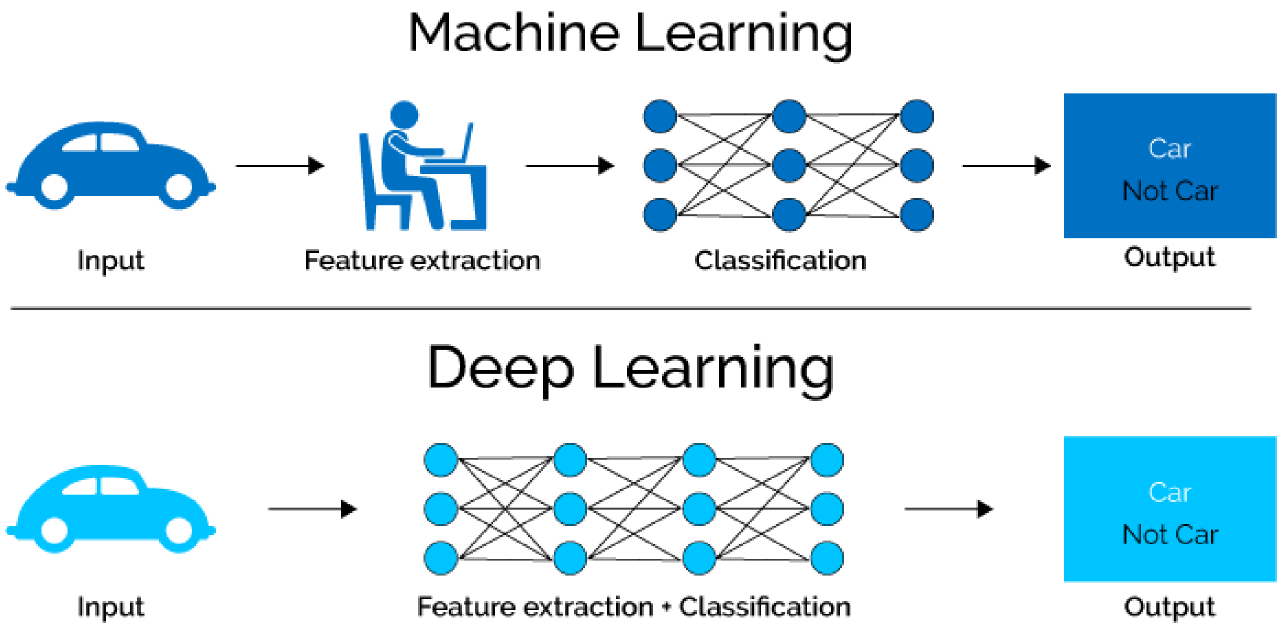
- Transform into the right representation
- Classify points simply by threshold on radius axis
- Single neuron with non-linearity can do this

[Goodfellow, Bengio, Courville 2016]

Deep Learning

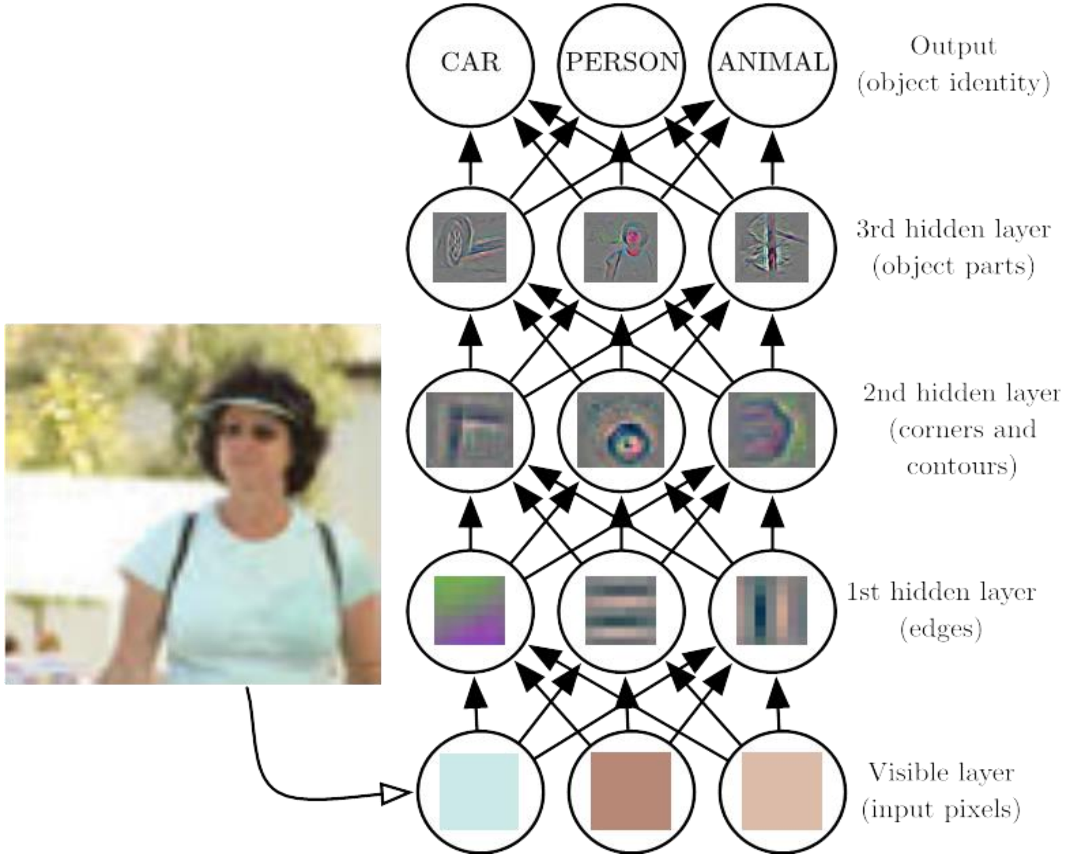
Subfield of machine learning:

- Learn **good representations/extract good features** of data
- Find **good predictors** using these representations/features
- Learn a hierarchy of representations/features that build on each other in layers



https://www.xenonstack.com/blog/static/public/uploads/media/machine-learning-vs-deep-learning.png

Depth: Layered composition

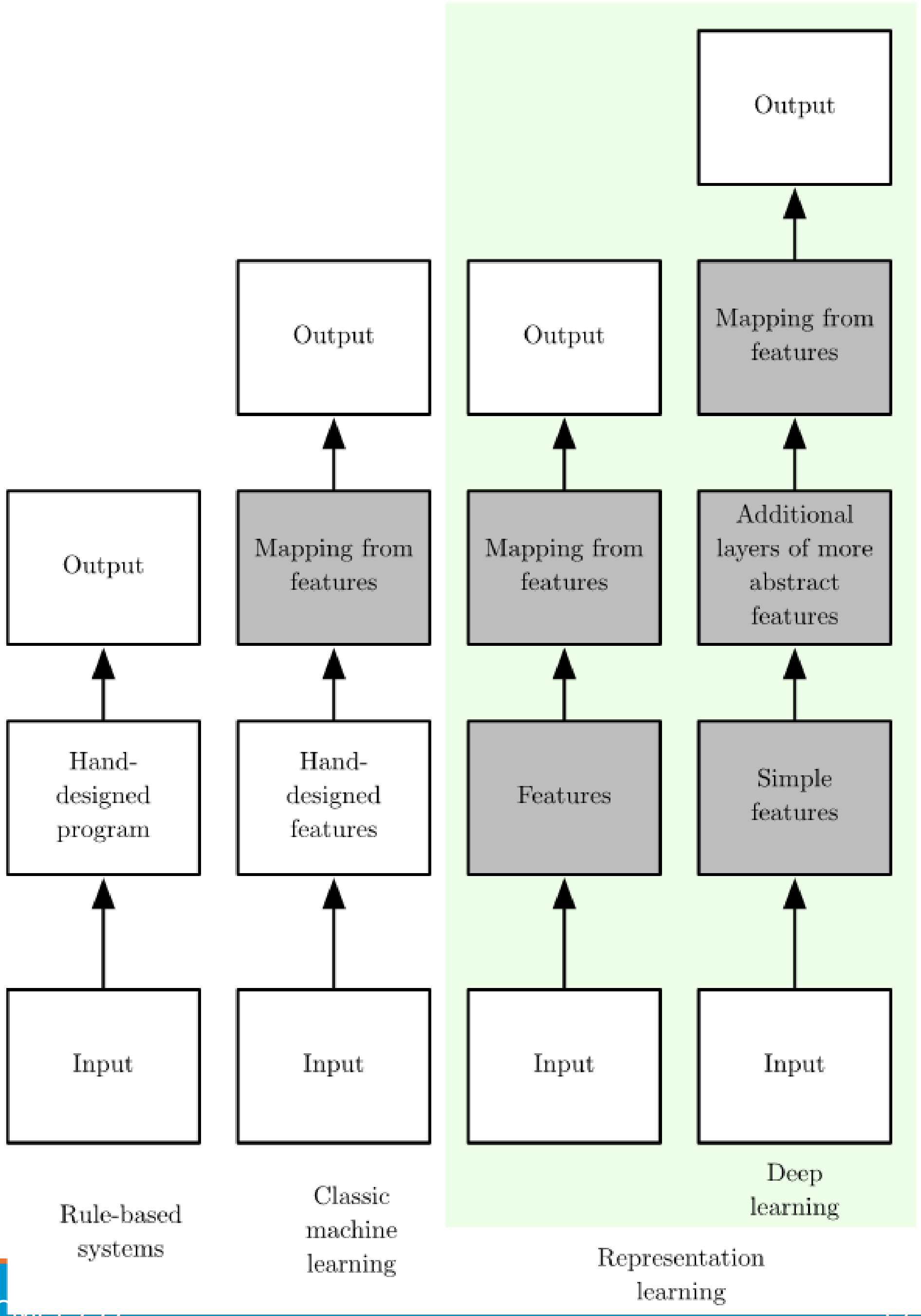


[Goodfellow, Bengio, Courville 2016]

Components of learning

- Hand designed program
  - Input → Output
- Increasingly automated
  - Simple features
  - Abstract features
  - Mapping from features

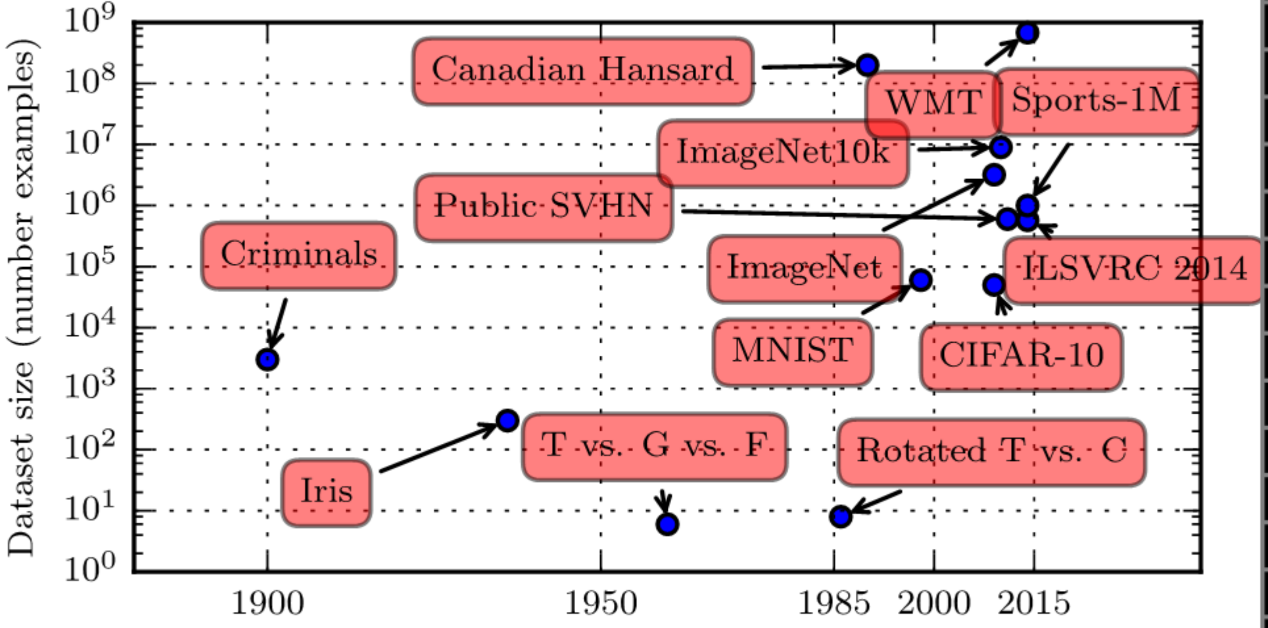
[Goodfellow, Bengio, Courville 2016]



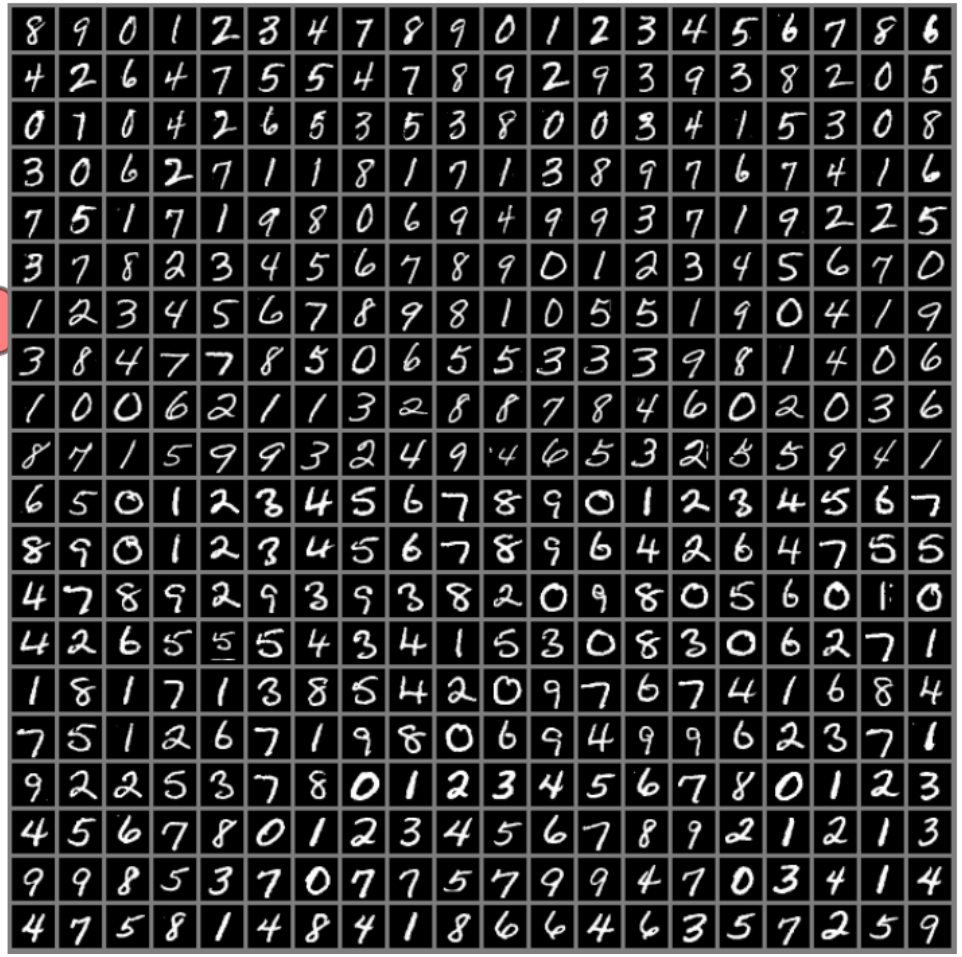
Why is DL useful?

- Manually designed features/representations:
  - require **domain knowledge**
  - may be **incomplete**
  - may take a **long time to design or validate.**
- Deep learning provides a very **flexible** and (almost) **universal** framework for:
  - representing world, visual, and linguistic information
  - creating **end-to-end** joint system learning representations and predictors
  - utilizing large amounts of training data

Growing dataset size

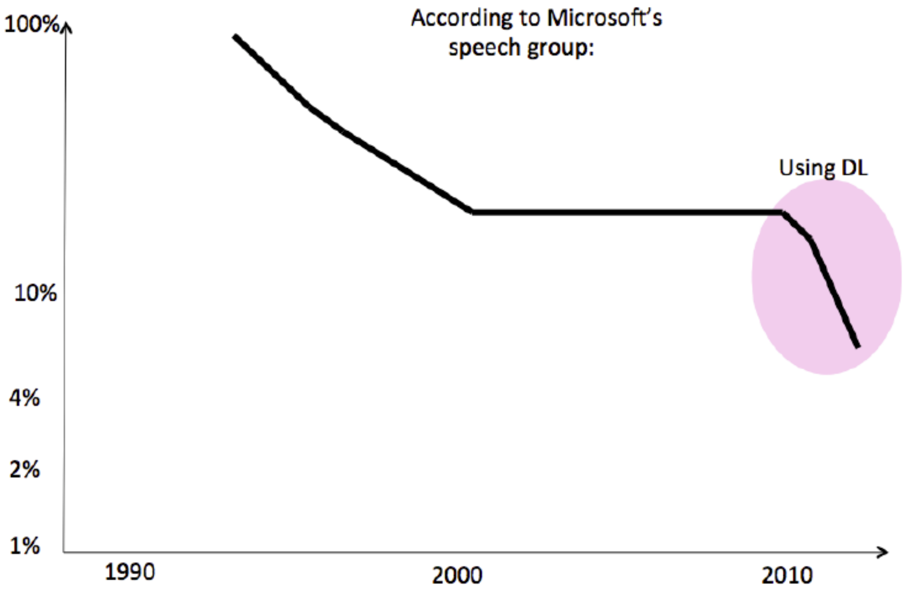


MNIST dataset

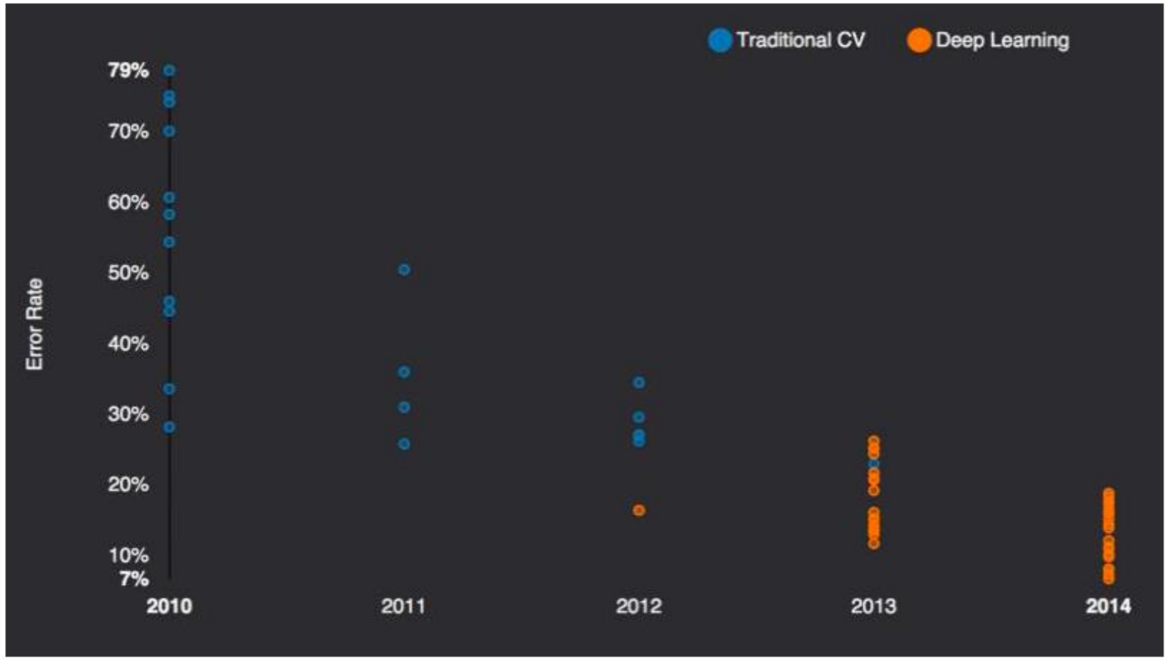


[Goodfellow, Bengio, Courville 2016]

State of the art in …



Deep Learning in Speech Recognition



ImageNet: The "computer vision World Cup"

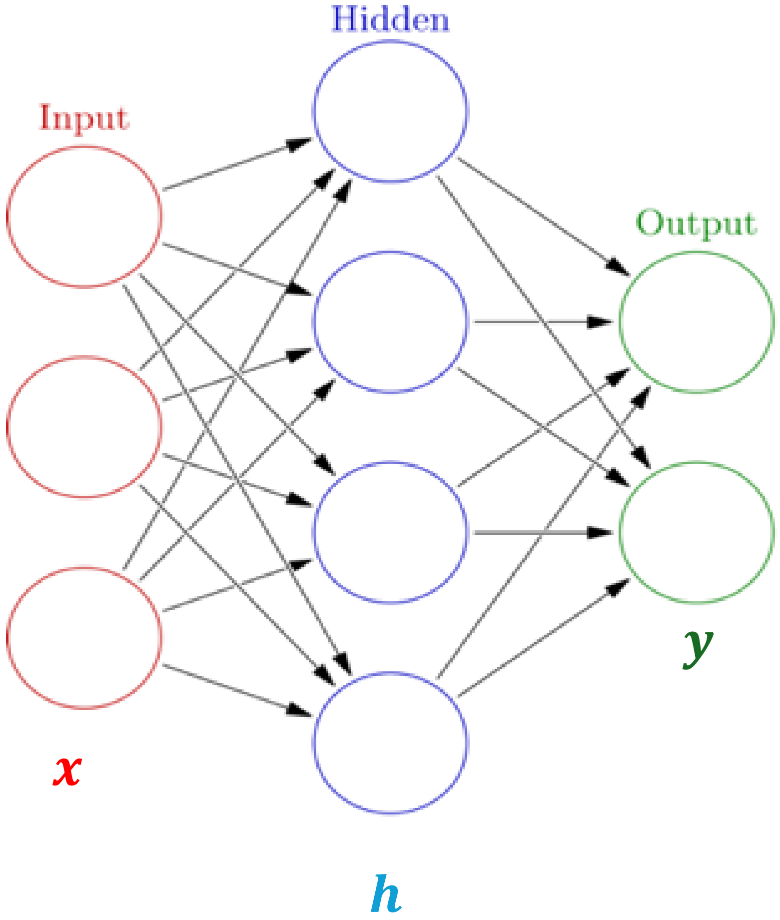
Several big improvements in recent years in NLP

- ✓ Machine Translation
- ✓ Sentiment Analysis
- ✓ Dialogue Agents
- ✓ Question Answering
- ✓ Text Classification …

Basics

Neural Network

Neural Network



Weights

$$\boldsymbol{h} = \boldsymbol{\sigma}_1(\mathbf{W}_1 \boldsymbol{x} + \boldsymbol{b}_1)$$

$$\boldsymbol{y} = \boldsymbol{\sigma}_2(\boldsymbol{W}_2 \boldsymbol{h} + \boldsymbol{b}_2)$$

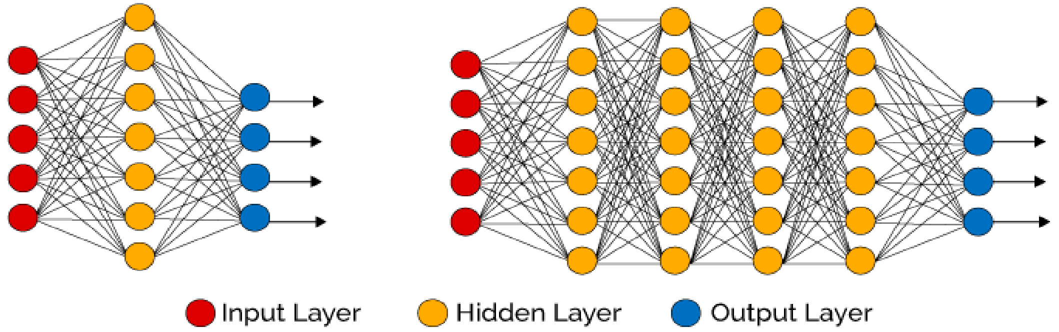
Activation functions

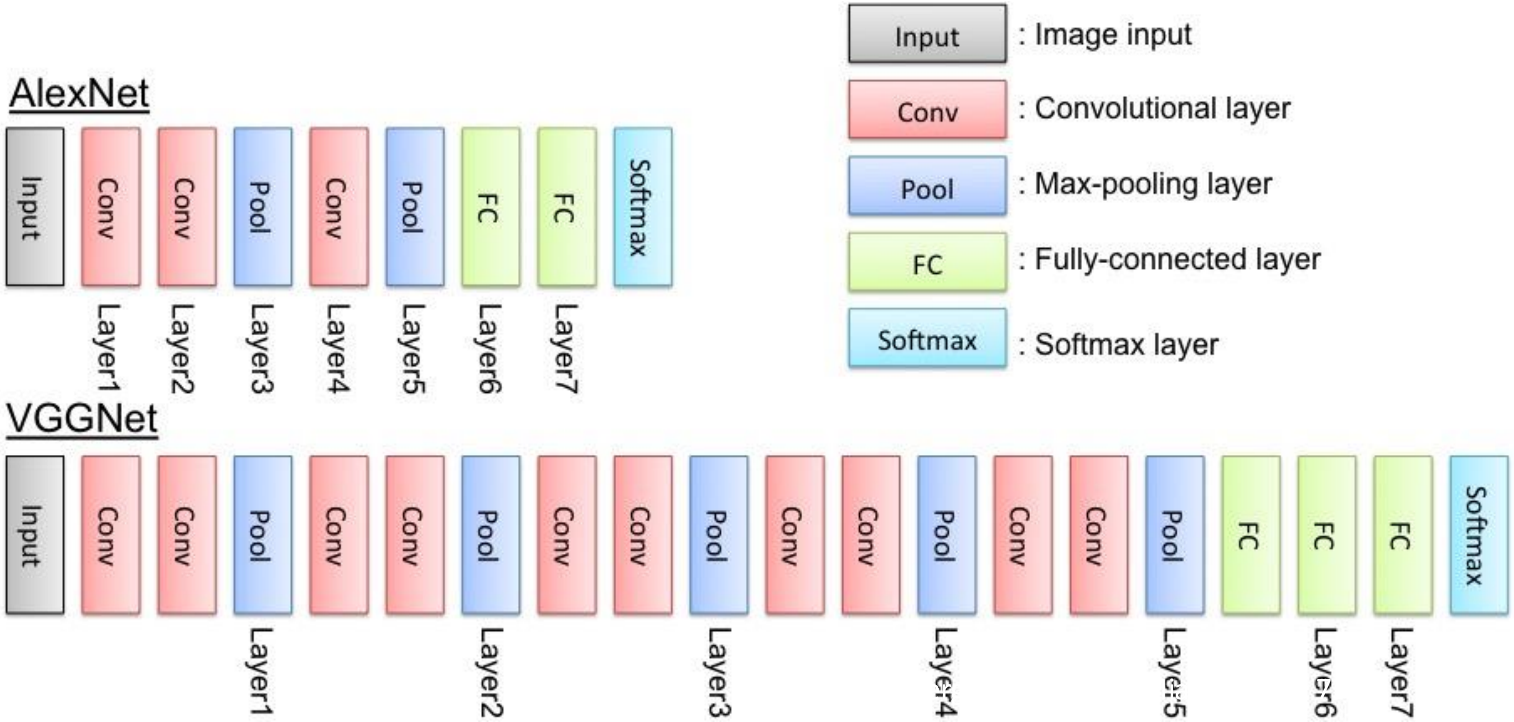
4 + 2 = 6 neurons (not counting inputs)
[3 x 4] + [4 x 2] = 20 weights
4 + 2 = 6 biases

26 learnable **parameters**
Parameter vector: $\boldsymbol{\theta}$

Neural Network Architectures

Simple Neural Network　　Deeper Neural Network

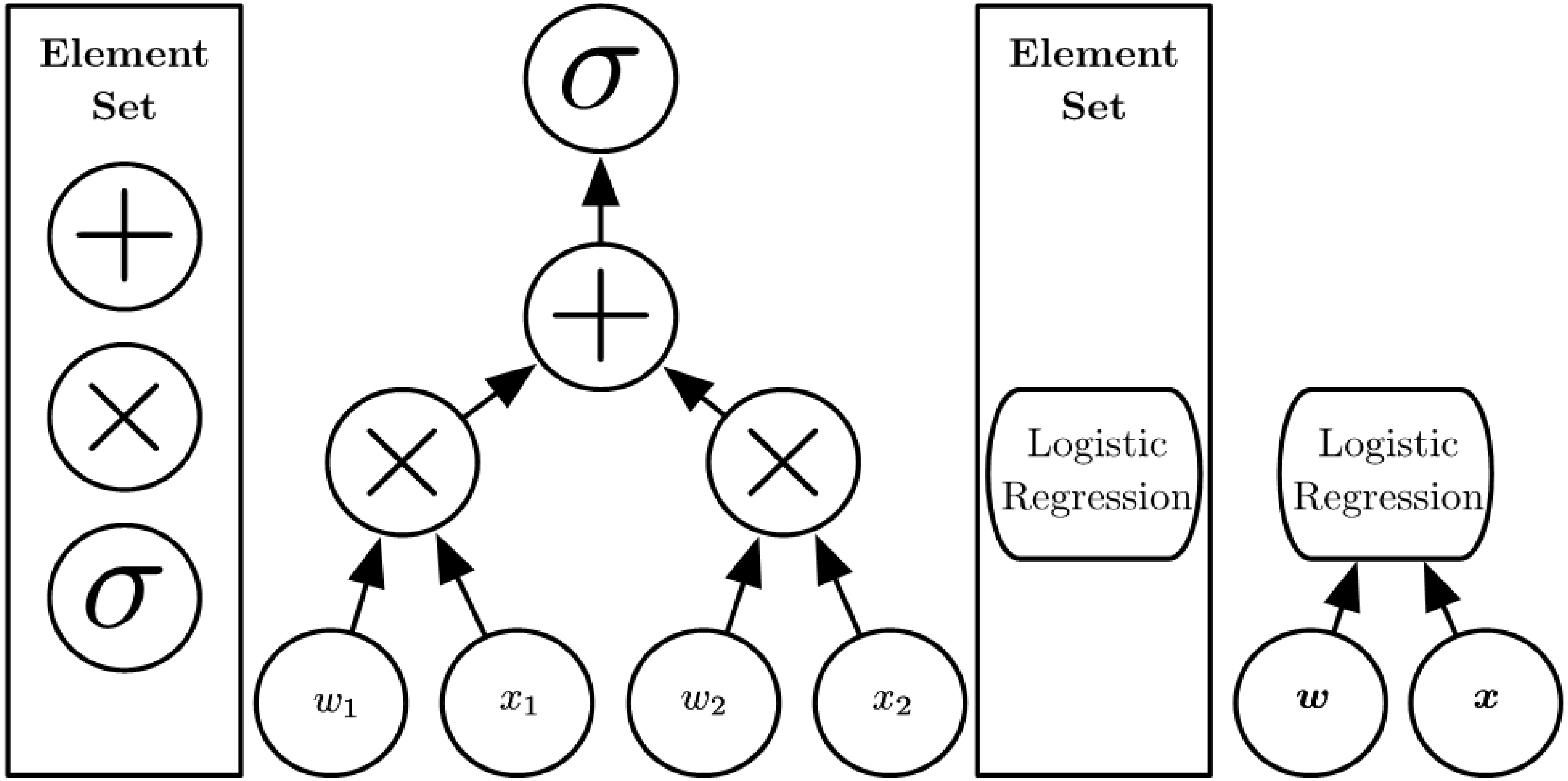




How deep is "deep learning?"

H. Kataoka et al., "Feature evaluation of deep convolutional neural networks for object recognition and detection." arXiv preprint arXiv:1509.07627.

Computational graph



[Goodfellow, Bengio, Courville 2016]

Why Neural Networks?

- Informal Conjecture. For every function f we might want to learn from data, there exists a "not too large" neural network that can represent f.

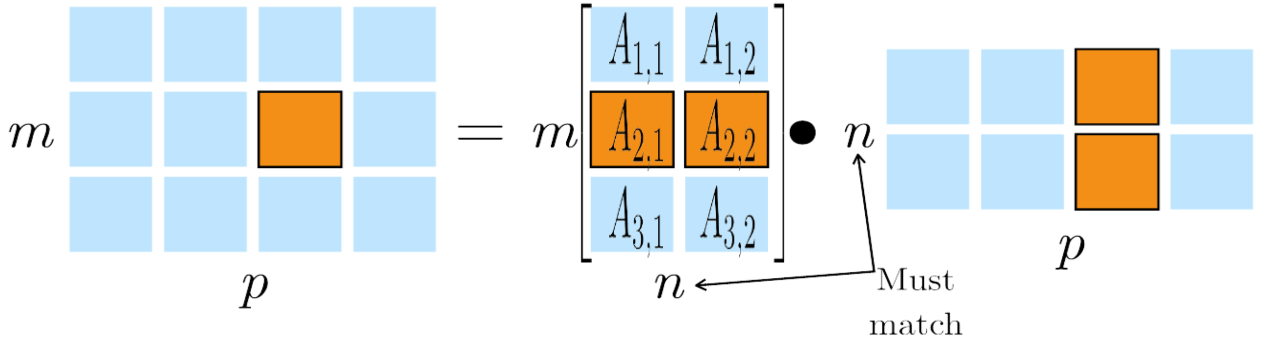
- If true, the upshot is that we don't need to consider any other classes of models in machine learning when we want to learn a function.

Basics

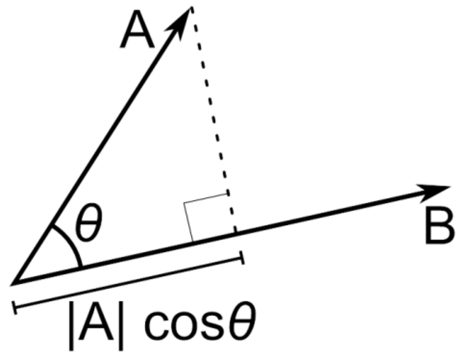
Linear Algebra and Optimization

Linear algebra

- Tensor is an array of numbers
  - Multi-dim: 0d scalar, 1d vector, 2d matrix/image, 3d RGB image
- Matrix (dot) product $\boldsymbol{C} = \boldsymbol{AB}$ $\quad C_{i,j} = \sum_k A_{i,k} B_{k,j}$



- Dot product of vectors A and B
  - (m = p = 1 in above notation)



[Goodfellow, Bengio, Courville 2016]

Linear algebra: Norms

- L^p norm

$$||\boldsymbol{x}||_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}}$$

- Most popular norm: L2 norm, $p{=}2$
- L1 norm, $p{=}1$: $||\boldsymbol{x}||_1 = \sum_i |x_i|.$
- Max norm, infinite p: $||\boldsymbol{x}||_\infty = \max_i |x_i|.$

[Goodfellow, Bengio, Courville 2016]

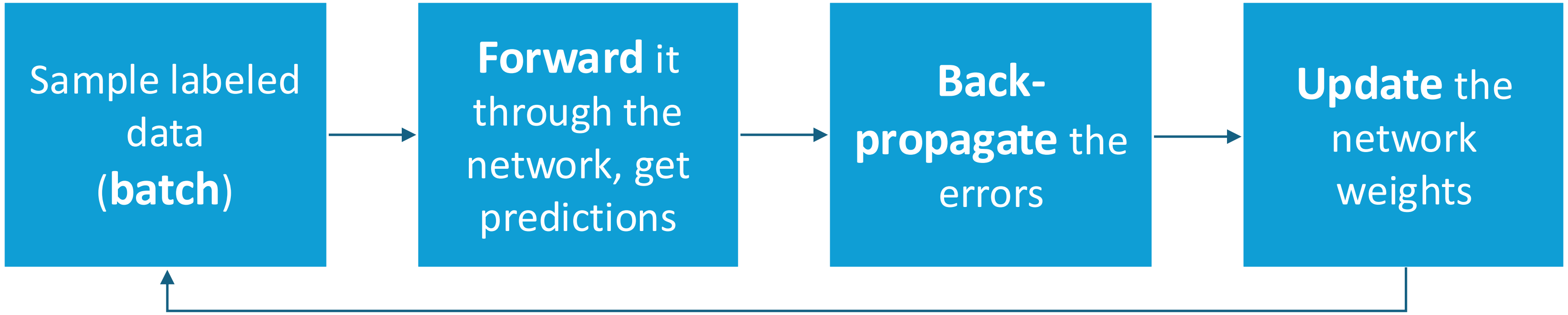
Learning = Optimization

Learning a classifier = optimizing over data with respect to parameters.

Given:

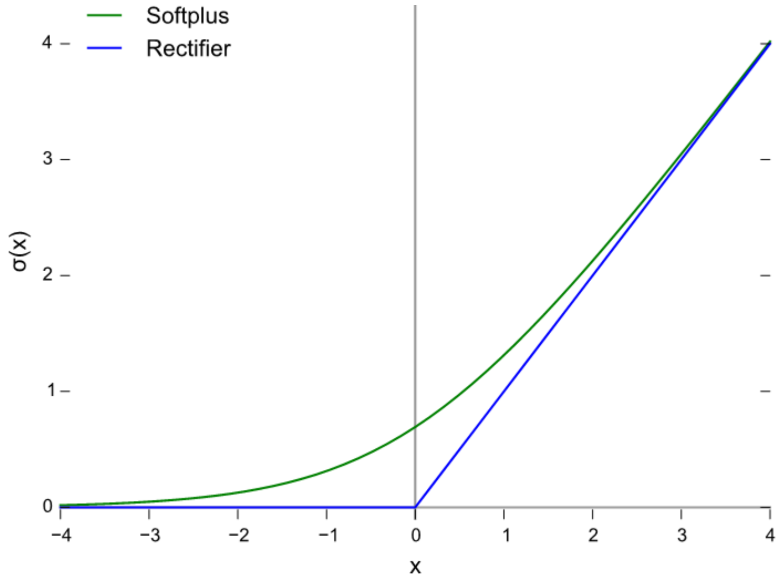
- m labeled samples $S = (\mathrm{x}_1, \mathrm{y}_1), \ldots (\mathrm{x}_\mathrm{m}, \mathrm{y}_\mathrm{m})$
- A loss function to penalize errors (e.g., $l(y, y') = (y - y')^2$)
- A model $f(w, x)$ (e.g., $f(w, x) = w\,x + 1$)

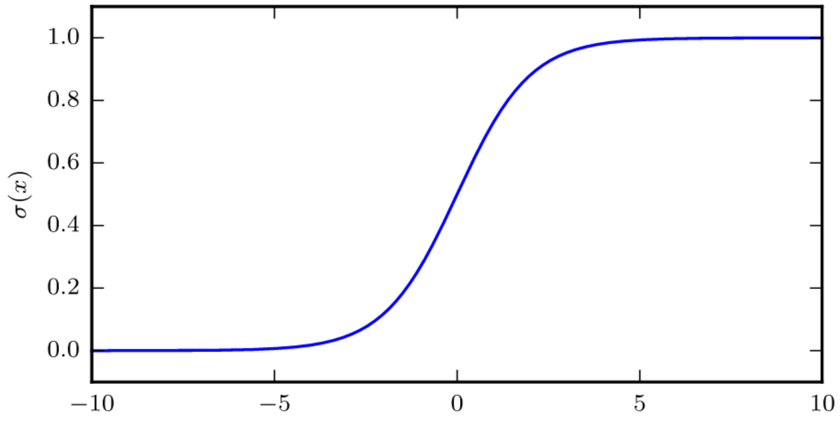
Goal: Minimize l with regard to w: $\mathrm{argmin}_\mathrm{w} \sum_{i=1}^{m} l(yi, f(w, xi))$



Nonlinearities

- ReLU
- Softplus
- Logistic Sigmoid





[Goodfellow, Bengio, Courville 2016]

Forward propagation

- Suppose we start with the input $x_1, x_2, \ldots, x_n$
  - These are the values of the first layer of the network (the input layer)

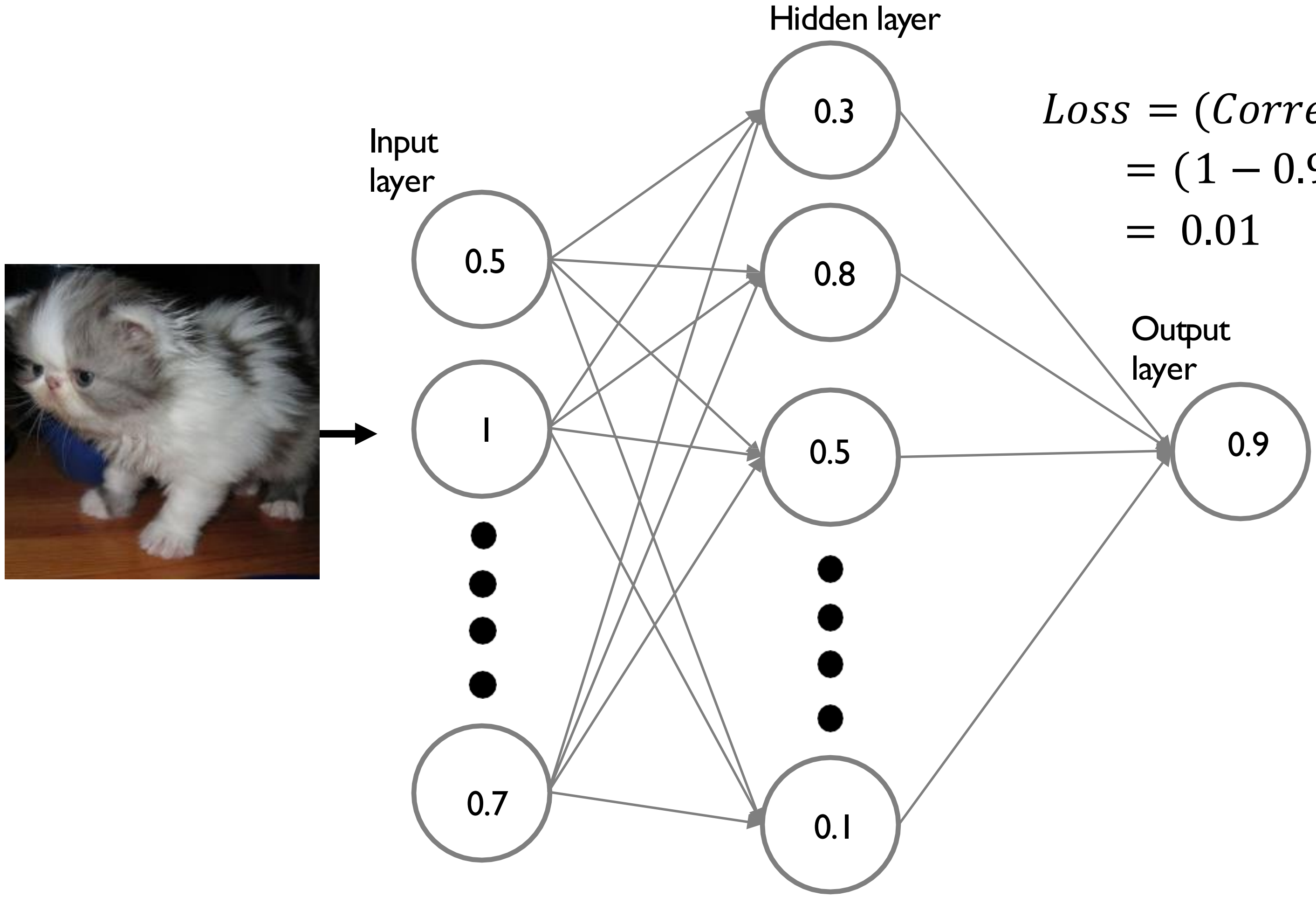
- Compute the value of neuron k in the next layer

  $w_{1,k}x_1 + w_{2,k}x_2 + \ldots + w_{n,k}x_n + b_k$ where $w_{i,k}$ is a weight associated

- Assuming ReLU/Sigmoid as the activation function, neuron k will be set to:

$$\mathrm{Max}(0, w_{1,k}x_1 + w_{2,k}x_2 + \ldots + w_{n,k}x_n + b_k) \text{ or}$$
$$\sigma(w_{1,k}x_1 + w_{2,k}x_2 + \ldots + w_{n,k}x_n + b_k)$$

Calculate the Loss

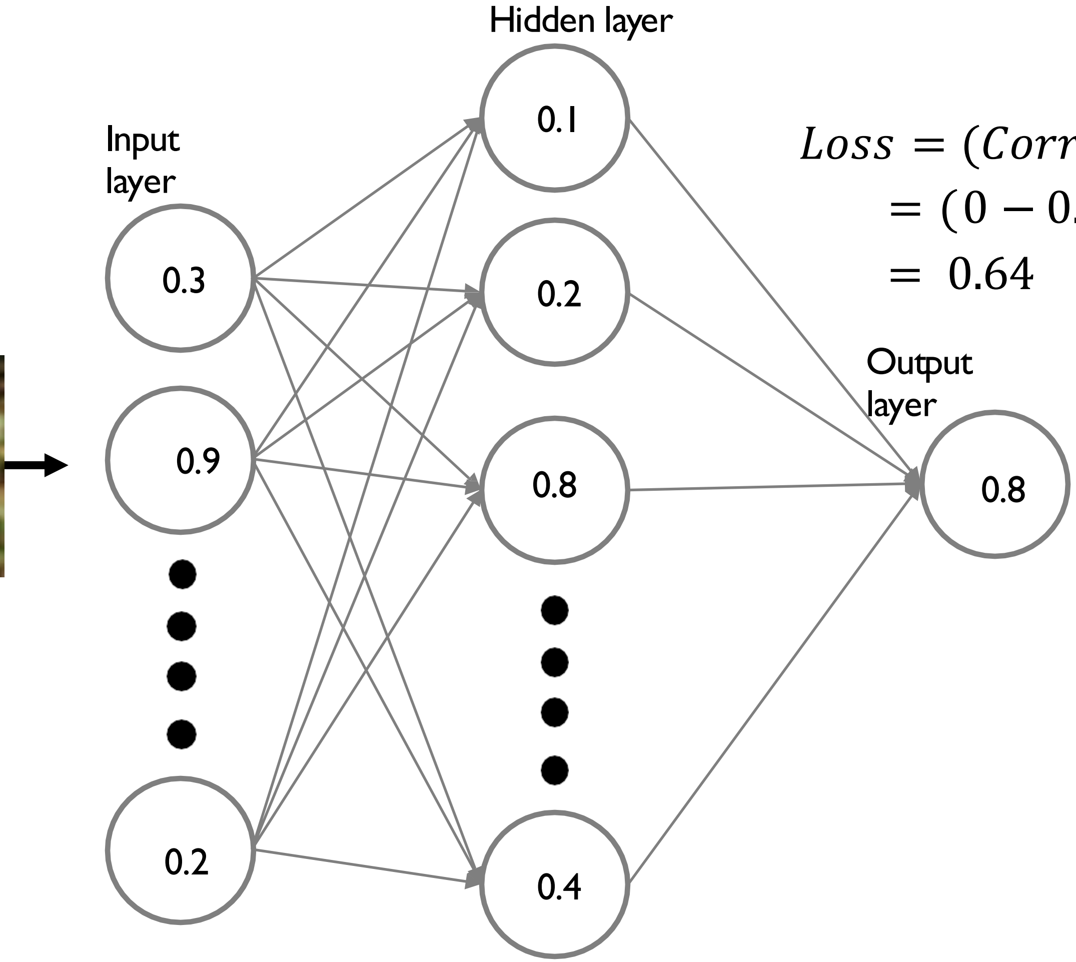


$$Loss = (CorrectAnswer - OutputLayer)^2$$
$$= (1 - 0.9)^2$$
$$= 0.01$$

If large, cat

else, not cat

Calculate the Loss

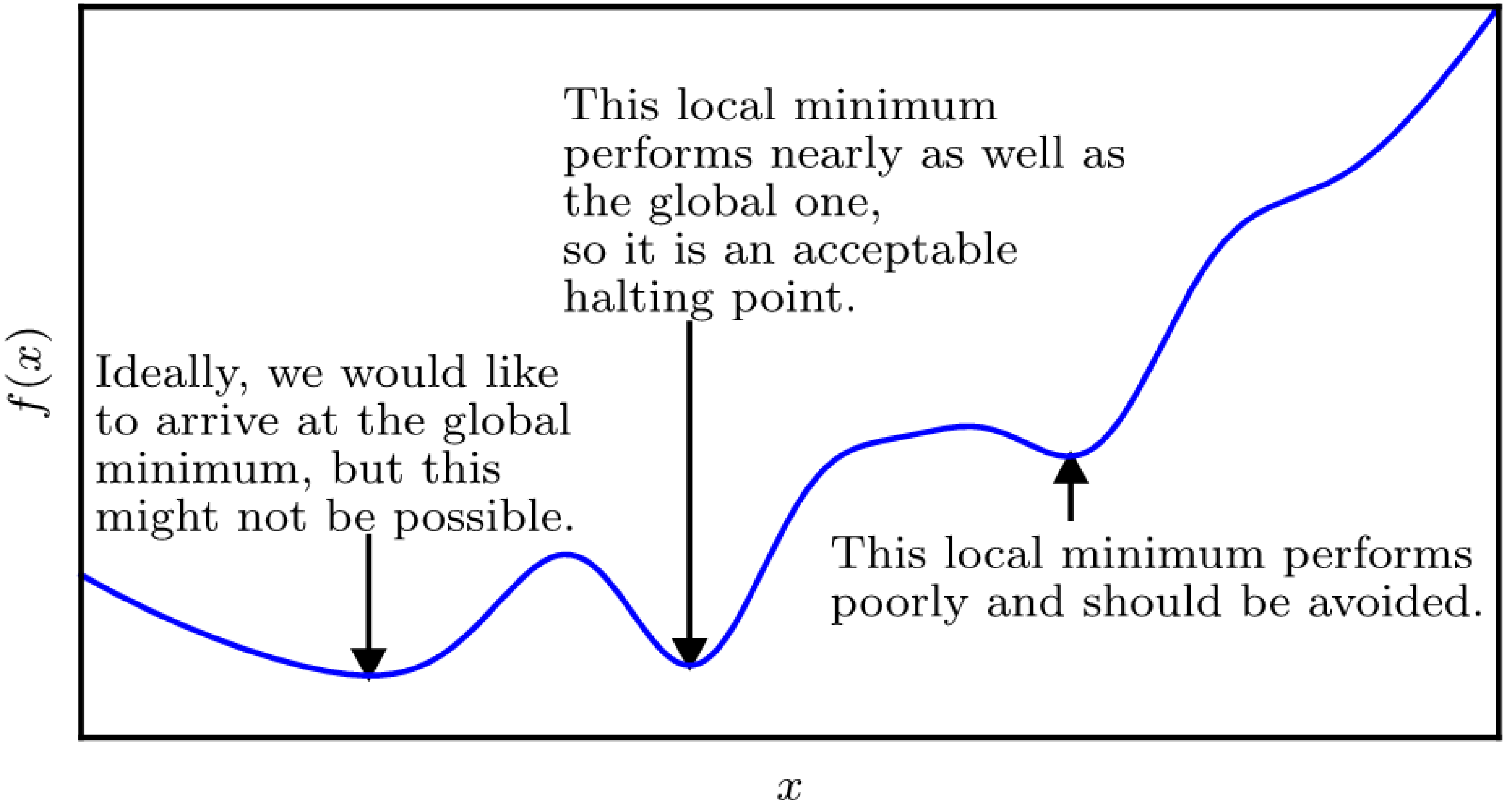


$$Loss = (CorrectAnswer - OutputLayer)^2$$
$$= (0 - 0.8)^2$$
$$= 0.64$$

If large, cat

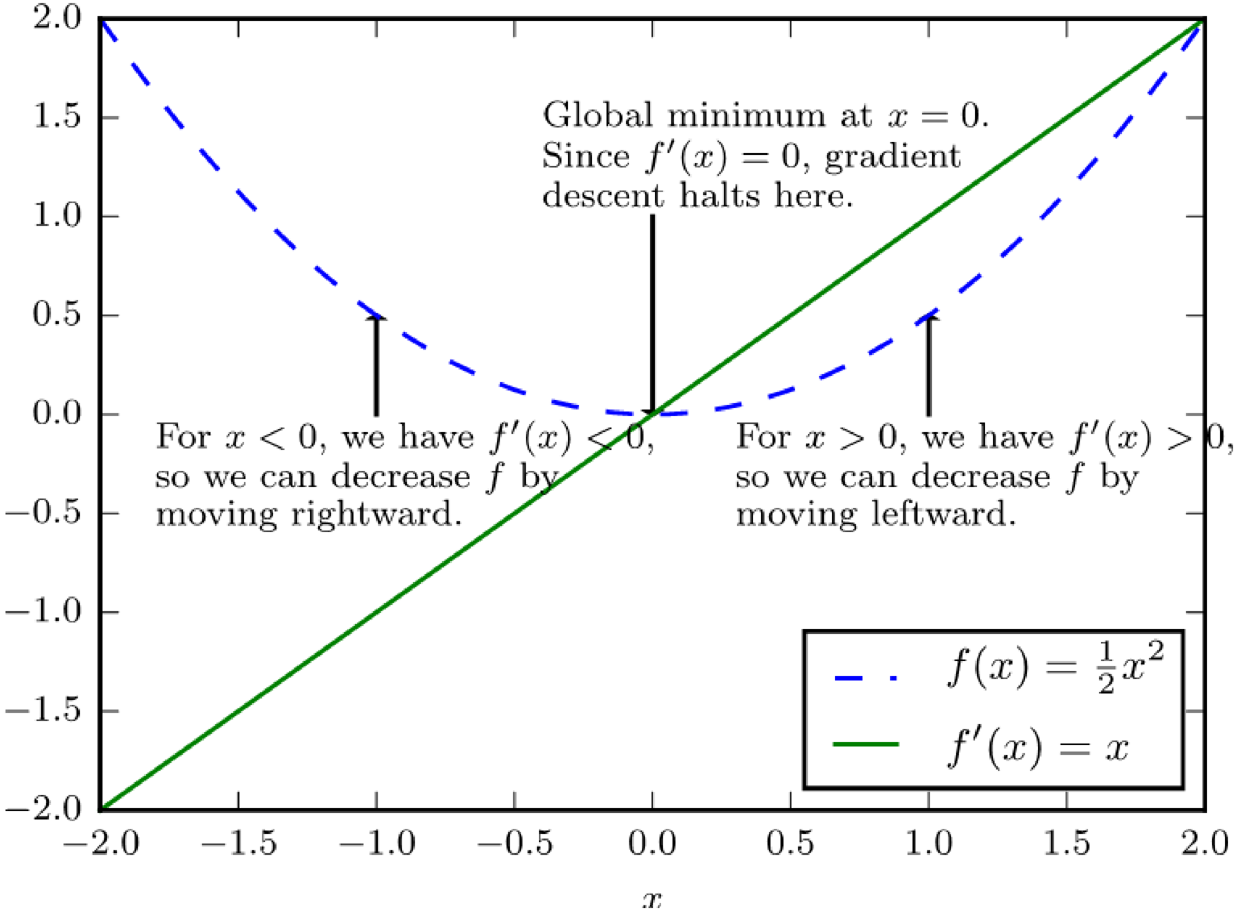
else, not cat

Approximate optimization



[Goodfellow, Bengio, Courville 2016]

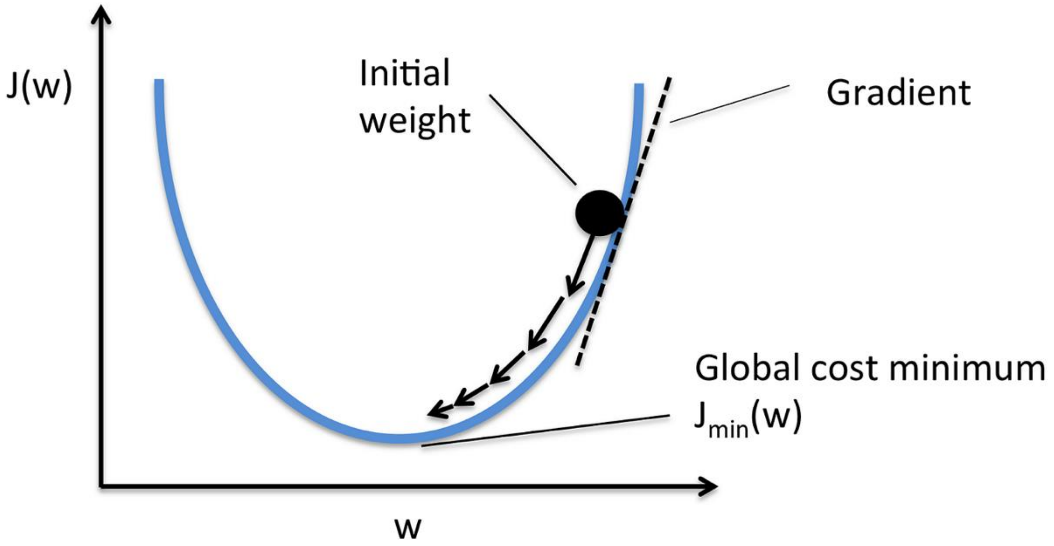
Gradient descent



[Goodfellow, Bengio, Courville 2016]

Gradient descent

- Recall that $\nabla f(\boldsymbol{x})$ is the direction of greatest increase in the function value
- A greedy optimization algorithm that iteratively steps in the negative gradient direction
- More formally, let α be a small step size (the learning rate). The gradient descent algorithm iteratively updates the weights:



$$\forall t: \quad w^{t+1} = w^{t} - \alpha \nabla_w l(w^t)$$

https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

Stochastic Gradient descent

For gradient descent, we need to compute the gradient of

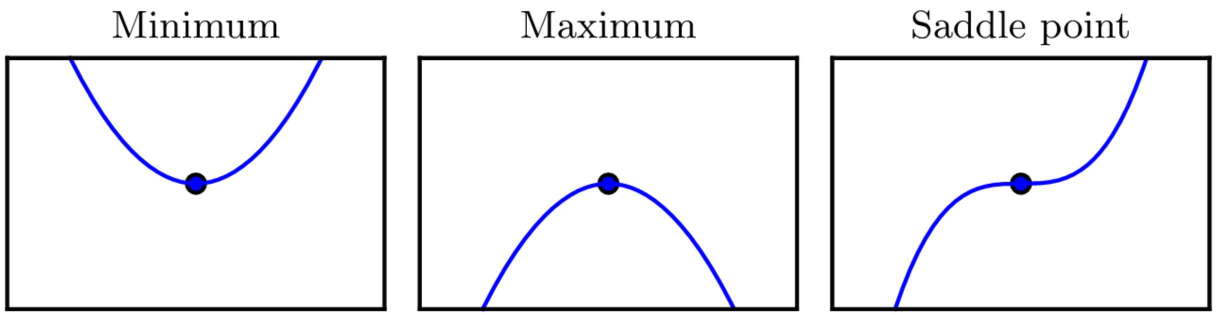
$$l(w^t) = \sum_{i=1}^{m} l(yi, f(w^t, xi))$$

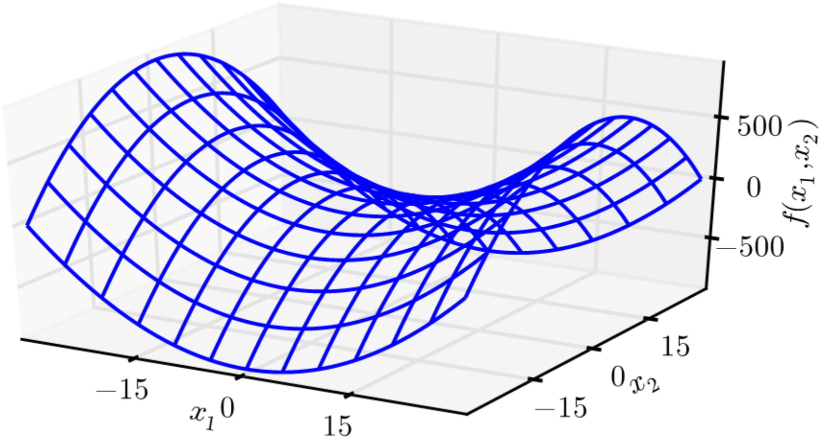
- Slow when we have millions of samples!

SGD: At time t:

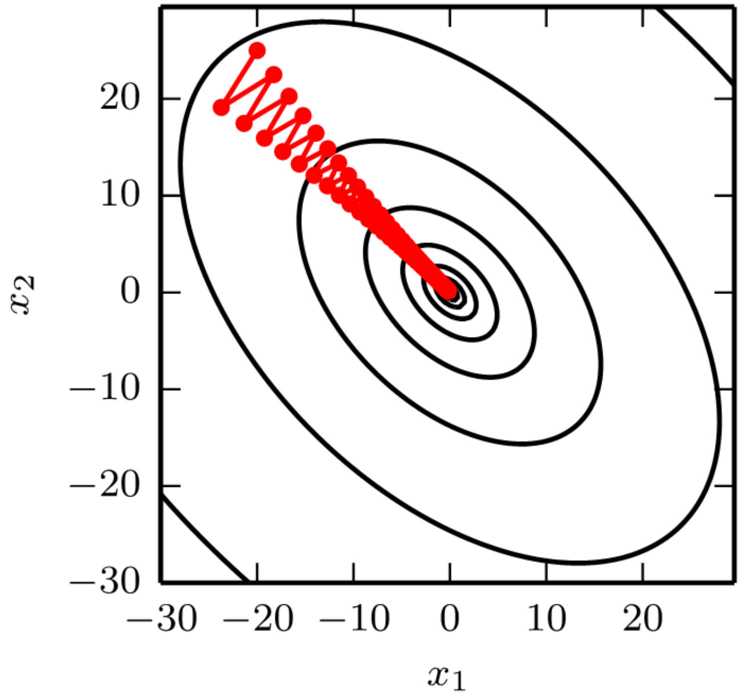
- pick random subset B with b samples of training set
- update: $w_{t+1} = w_t - \alpha \cdot \nabla_w \sum_{i \in B} l(y_i, f(w^t, xi))$

Critical points





Saddle point – 1st and 2nd derivative vanish

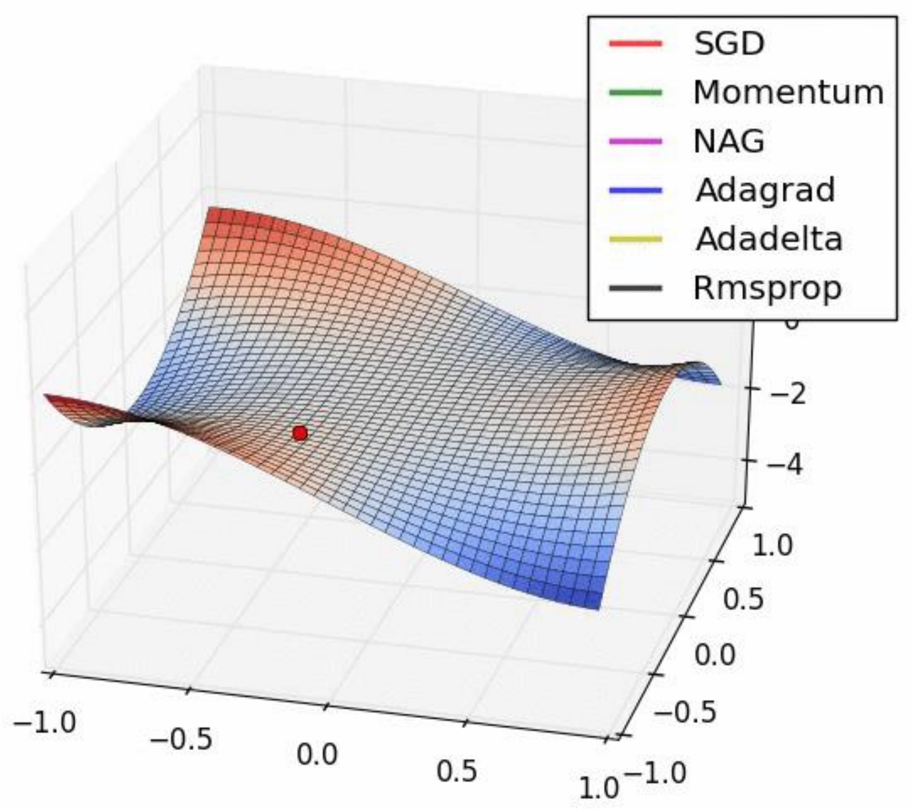


Poor conditioning:
1st derivative large in one and small in another direction

[Goodfellow, Bengio, Courville 2016]

Optimization algorithm

- Lots of variants address choice of learning rate
- See Visualization of Algorithms
- AdaDelta and RMSprop often work well



https://www.ruder.io/optimizing-gradient-descent/#visualizationofalgorithms

Neural network playgrounds

- http://playground.tensorflow.org/
  - Try out simple network configurations on TF Playground

- https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html
  - Visualize linear and non-linear mappings

Back Propagation

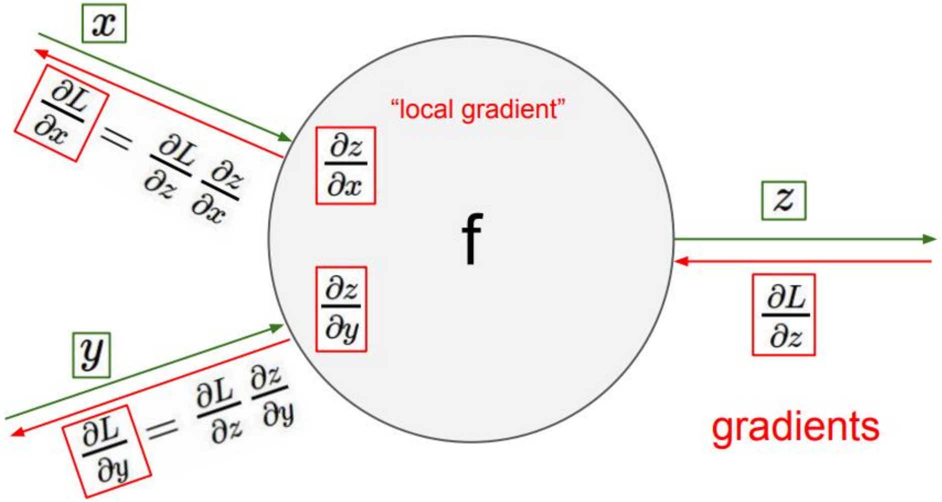
For gradient descent, we need to compute the gradient of

$$l(w^t) = \sum_{i=1}^{m} l^{(i)}(w^t) = \sum_{i=1}^{m} l(yi, f(w^t, xi))$$

Challenge: How to compute partial derivatives of huge network?

Compute gradient of each constituent $l^{(i)}(w^t)$ by recursively applying **chain rule**

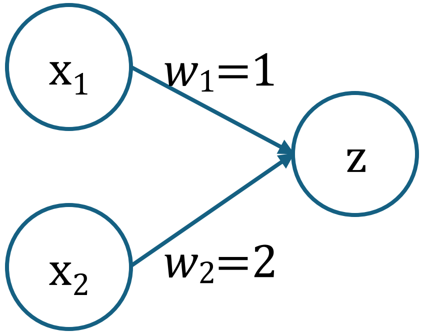
- called the "Backpropagation algorithm"



Back propagation illustration from CS231n Lecture 4

Back Propagation Example

Suppose we have the single training sample $\boldsymbol{x} = (0.5, 1)$, $y = 0$, and the following network:



Denote $z = \sigma(w_1x_1 + w_2x_2) = \sigma(f)$

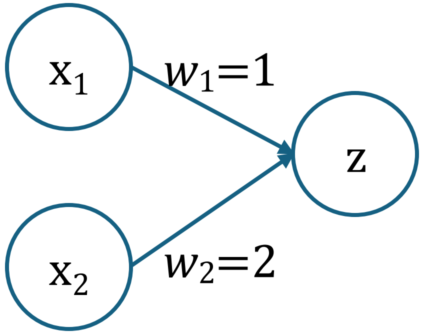
Let $l = (z - y)^2$, then by the chain rule:

$$\frac{\partial l}{\partial w_1} = 2(z-y)\frac{\partial z}{\partial w_1} \quad = 2(z-y)\frac{\partial \sigma(f)}{\partial f}\frac{\partial f}{\partial w_1} = \sigma(f)(1-\sigma(f))x_1$$

$$= 2(z-y)z(1-z)x_1$$

Back Propagation Example

Suppose we have the single training sample $\boldsymbol{x} = (0.5, 1)$, $y = 0$, and the following network:



- First we have a forward propagation pass to calculate the value of z:
- $z = \sigma(w_1 x_1 + w_2 x_2) = \sigma(2.5) \approx 0.924$

$$\frac{\partial l}{\partial w_1} = 2(z - y)z(1 - z)x_1 \approx 0.065, \quad \frac{\partial l}{\partial w_2} = 2(z - y)z(1 - z)x_2 \approx 0.13$$

$$w_1' = w_1 - \alpha \cdot \frac{\partial l}{\partial w_1} = 1 - (0.1)(0.065) = 0.9935$$

$$w_2' = w_2 - \alpha \cdot \frac{\partial l}{\partial w_2} = 2 - (0.1)(0.13) = 1.987$$

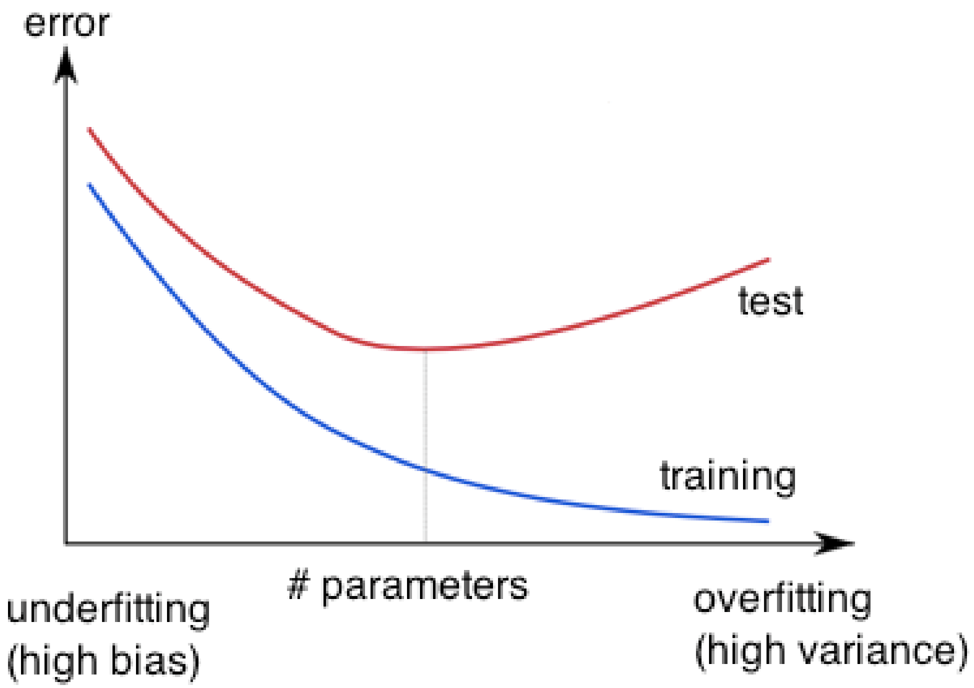
Back Propagation and SGD in Practice

- For a deep neural network, back propagation calculates the partial derivatives necessary for SGD.
- Do I have to program all of this calculus from scratch?
- NO! This is a major part of what deep learning libraries like Pytorch and tensorflow implement for you.

Regularization

Reduced generalization error without impacting training error

Overfitting

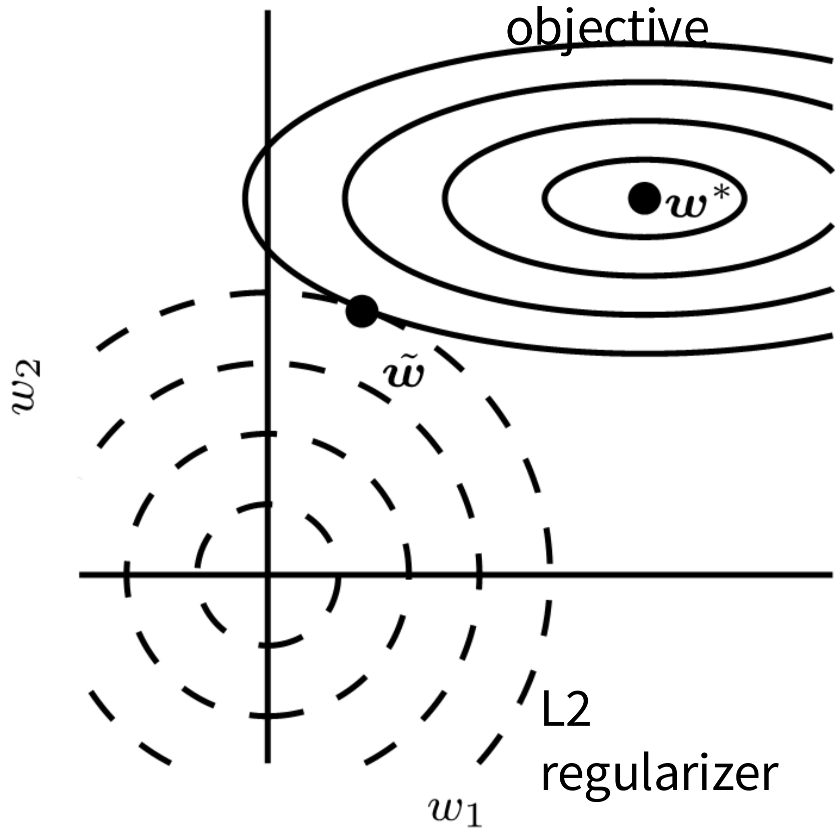


https://www.neuraldesigner.com/images/learning/selection_error.svg

- Learned hypothesis may **fit** training data very well, but fail to **generalize** to new examples (test data)
- To avoid overfitting, use explicit regularization

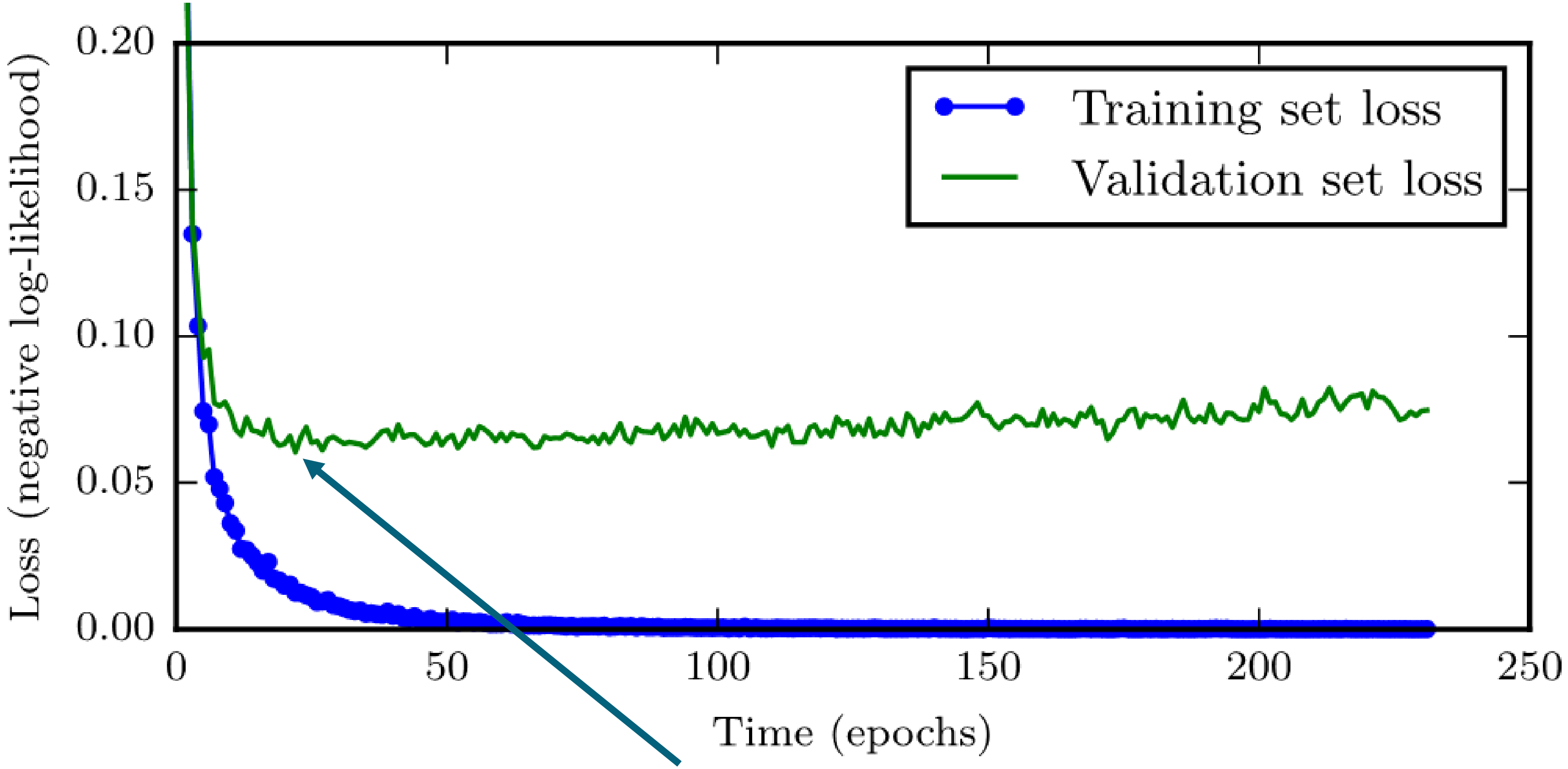
Constrained optimization

- Squared L2 encourages small weights
- L1 encourages sparsity of model parameters (weights)



[Goodfellow, Bengio, Courville 2016]

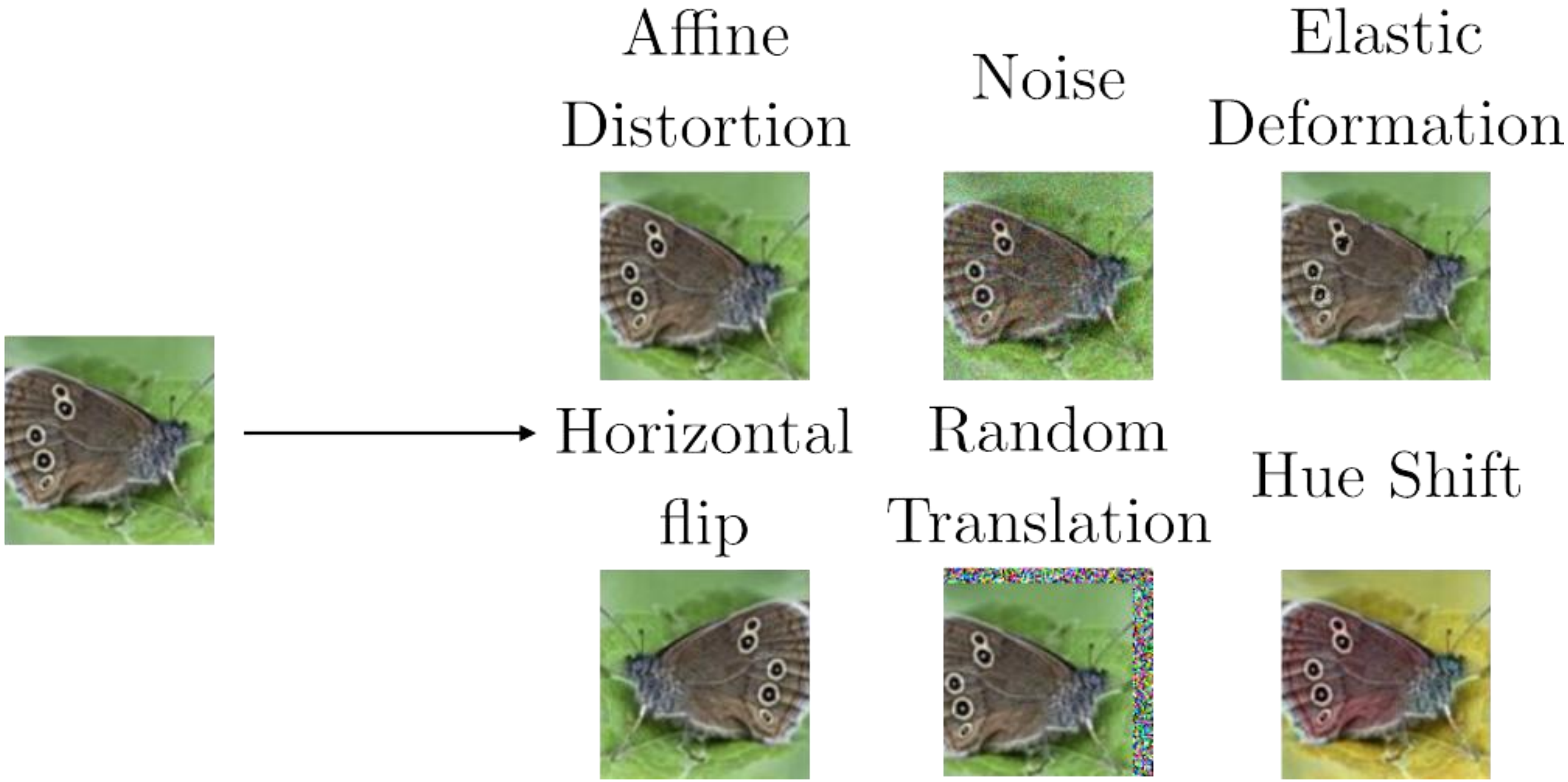
Learning curves



- Early stopping before validation error starts to increase

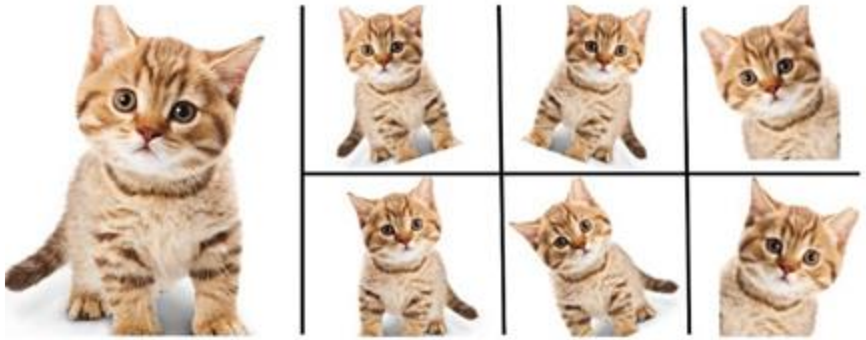
[Goodfellow, Bengio, Courville 2016]

Dataset augmentation



[Goodfellow, Bengio, Courville 2016]

Dataset augmentation



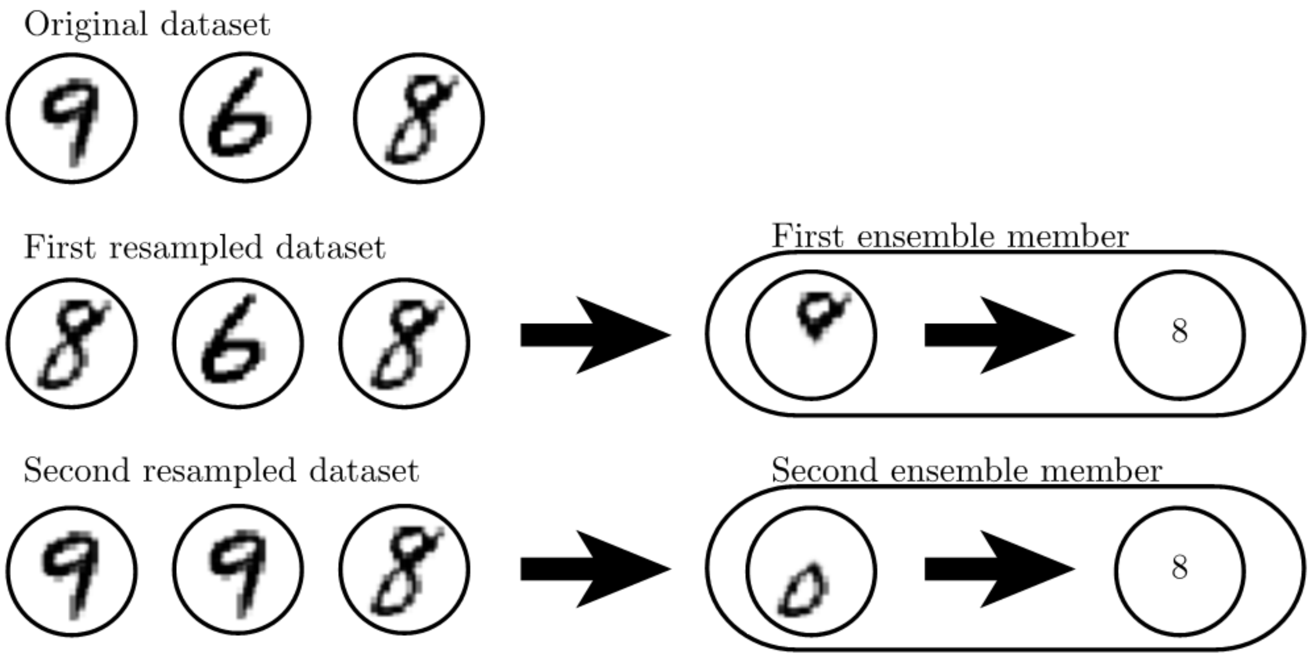
Invariance property

Easy Data Augmentation	Short Example
Random Swap	I am jogging → I tiger jogging
Random Insertion	I am jogging → I am salad jogging
Random Deletion	I am jogging → I jogging
Random Synonym Replacement	I am jogging → I am running

Shorten et al., “Text Data Augmentation for Deep Learning.” Journal of Big Data. 8. 10.1186/s40537-021-00492-0.

Bagging

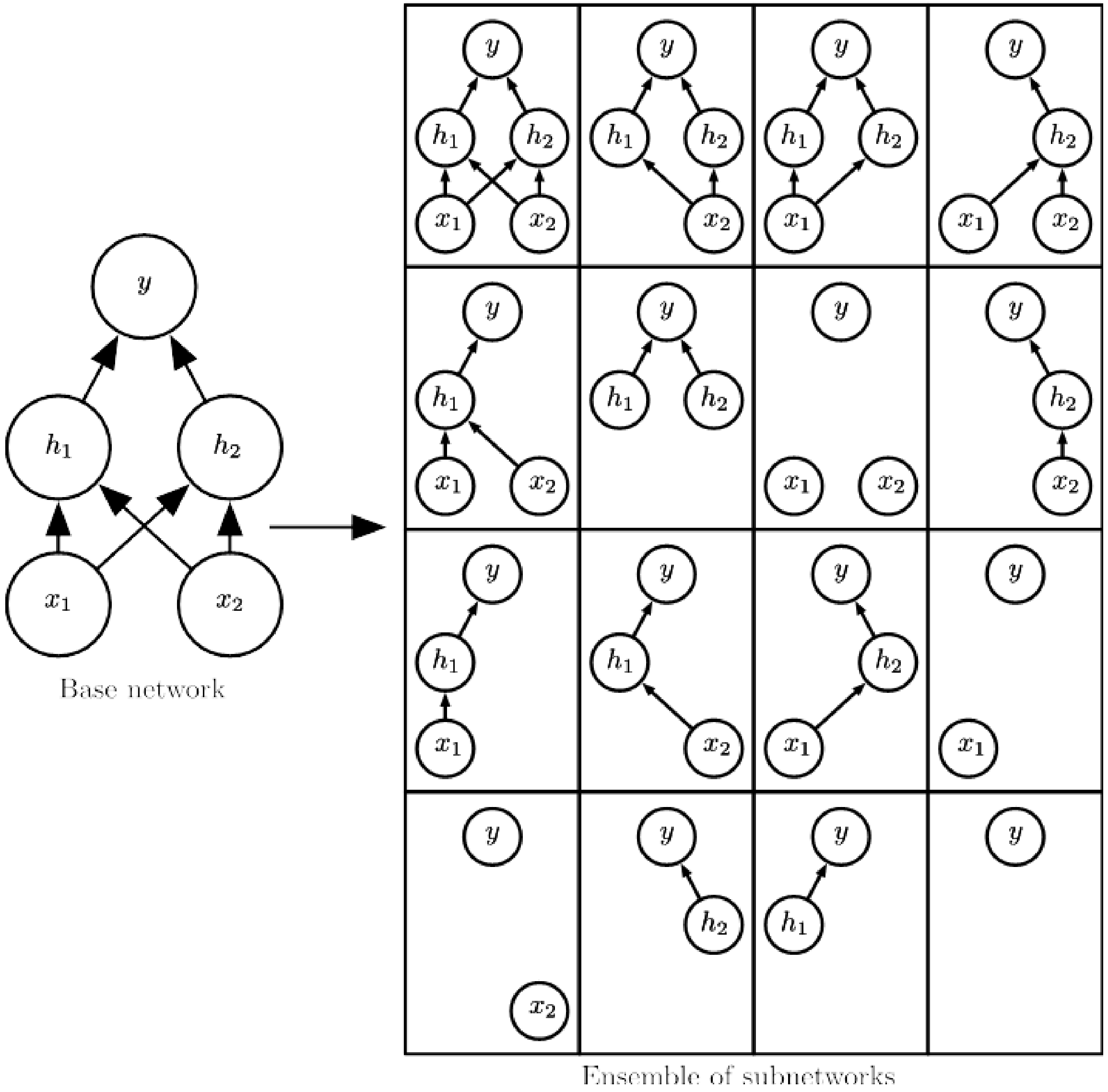
- Average multiple models trained on subsets of the data
- First subset: learns top loop, Second subset: bottom loop



[Goodfellow, Bengio, Courville 2016]

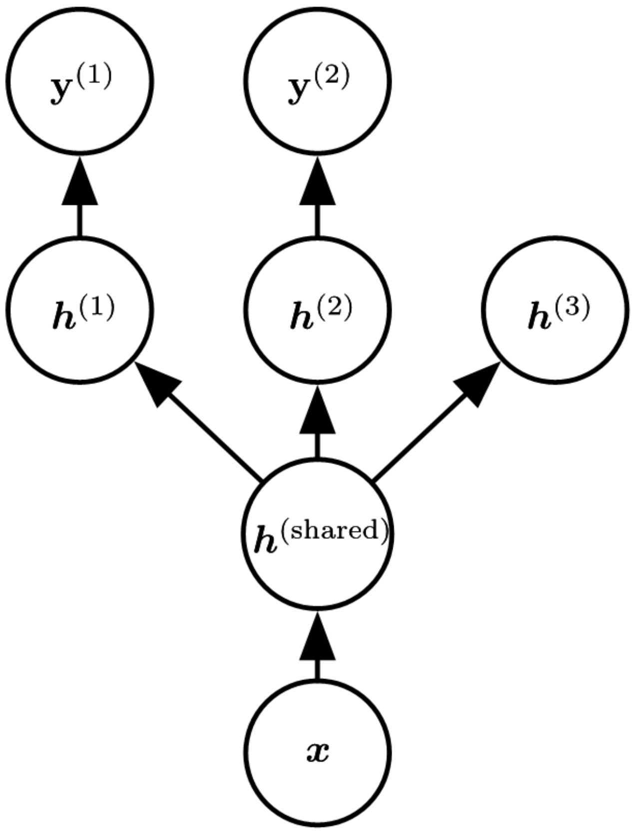
Dropout

- Random sample of connection weights is set to zero
- Train different network model each time
- Learn more robust, generalizable features



[Goodfellow, Bengio, Courville 2016]

Multitask learning



- Shared parameters are trained with more data
- Improved generalization error due to increased statistical strength

[Goodfellow, Bengio, Courville 2016]

Software for Deep Learning

Current Frameworks

- Tensorflow / Keras
- PyTorch
- DL4J
- Caffe (superseded by Caffe2, which is merged into PyTorch)
- And many more
- Most have CPU-only mode but much faster on NVIDIA GPU

Development strategy

- Identify needs: High accuracy or low accuracy?
- Choose metric
  - Accuracy (% of examples correct), Coverage (% examples processed)
  - Precision TP/(TP+FP), Recall TP/(TP+FN)
  - Amount of error in case of regression
- Build end-to-end system
  - Start from baseline, e.g. initialize with pre-trained network
- Refine driven by data

Pytorch example

https://pytorch.org/tutorials/beginner/basics/intro.html

Tensors ~= Numpy Arrays

Directly from data

Tensors can be created directly from data. The data type is automatically inferred.

```
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
```

From a NumPy array

Tensors can be created from NumPy arrays (and vice versa - see Bridge with NumPy).

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

```python
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

Out:

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.8823, 0.9150],
        [0.3829, 0.9593]])
```

Dataset

We load the FashionMNIST Dataset with the following parameters:

- `root` is the path where the train/test data is stored,
- `train` specifies training or test dataset,
- `download=True` downloads the data from the internet if it's not available at `root`.
- `transform` and `target_transform` specify the feature and label transformations

```
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Dataset

```
class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

DataLoader

Preparing your data for training with DataLoaders

The `Dataset` retrieves our dataset's features and labels one sample at a time. While training a model, we typically want to pass samples in "minibatches", reshuffle the data at every epoch to reduce model overfitting, and use Python's `multiprocessing` to speed up data retrieval.

`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Going to try to classify images of clothing

Iterate through DataLoader

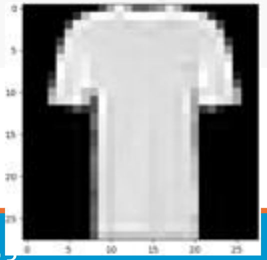
We have loaded that dataset into the `DataLoader` and can iterate through the dataset as needed. Each iteration below returns a batch of `train_features` and `train_labels` (containing `batch_size=64` features and labels respectively). Because we specified `shuffle=True`, after we iterate over all batches the data is shuffled (for finer-grained control over the data loading order, take a look at Samplers).

```
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

Out:

```
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
Label: 5
```

In this example, we want to classify images, e.g., classify this as a shirt



Defining a Multilayer Perceptron

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

`def __init__(self):` is Python syntax for a constructor

Inherits from **nn.module** defined by pytorch

Network architecture. `nn.Linear(input_dim, output_dim)`, this model for 28 by 28 pixel images

Flatten() turns a tensor into a 1d-tensory

Goes through layers in order

https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

Forward Propagation

To use the model, we pass it the input data. This executes the model's `forward`, along with some background operations. Do not call `model.forward()` directly!

Calling the model on the input returns a 2-dimensional tensor with dim=0 corresponding to each output of 10 raw predicted values for each class, and dim=1 corresponding to the individual values of each output. We get the prediction probabilities by passing it through an instance of the `nn.Softmax` module.

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Out:

```
Predicted class: tensor([7], device='cuda:0')
```

Flattening

Let's break down the layers in the FashionMNIST model. To illustrate it, we will take a sample minibatch of 3 images of size 28x28 and see what happens to it as we pass it through the network.

```
input_image = torch.rand(3,28,28)
print(input_image.size())
```

3 gray scale each 28 by 28 pixels images

Out:

```
torch.Size([3, 28, 28])
```

```
flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())
```

Out:

```
torch.Size([3, 784])
```

3 flattened images, each 28*28 = 784 values

Network Layers

nn.Linear

The linear layer is a module that applies a linear transformation on the input using its stored weights and biases.

```
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
```

Out:

```
torch.Size([3, 20])
```

nn.ReLU

Non-linear activations are what create the complex mappings between the model's inputs and outputs. They are applied after linear transformations to introduce *nonlinearity*, helping neural networks learn a wide variety of phenomena.

In this model, we use nn.ReLU between our linear layers, but there's other activations to introduce non-linearity in your model.

```
print(f"Before ReLU: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU: {hidden1}")
```

```
Before ReLU: tensor([[ 0.4158, -0.0130, -0.1144,  0.3960,  0.1476
After ReLU: tensor([[0.4158, 0.0000, 0.0000, 0.3960, 0.1476, 0.0000,
```

Sequential and Softmax

nn.Sequential is an ordered container of modules. The data is passed through all the modules in the same order as defined. You can use sequential containers to put together a quick network like `seq_modules`.

```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)
```

The last linear layer of the neural network returns *logits* - raw values in [-infty, infty] - which are passed to the nn.Softmax module. The logits are scaled to values [0, 1] representing the model's predicted probabilities for each class. `dim` parameter indicates the dimension along which the values must sum to 1.

```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

Training with SGD and Back Propagation

```python
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
```

Hyperparameters:
Learning_rate,
#epochs, batch_size

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

```
Epoch 1
-------------------------------
loss: 2.298730  [   64/60000]
loss: 2.289123  [ 6464/60000]
loss: 2.273286  [12864/60000]
loss: 2.269406  [19264/60000]
loss: 2.249603  [25664/60000]
loss: 2.229407  [32064/60000]
loss: 2.227368  [38464/60000]
loss: 2.204261  [44864/60000]
loss: 2.206193  [51264/60000]
loss: 2.166651  [57664/60000]
```

Prediction and Testing

```
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage for tensors with
requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Sources

- I. Goodfellow, Y. Bengio, A. Courville “Deep Learning” MIT Press 2016 [link]
- Ismini Lourentzou, “Introduction to Deep Learning,” UIUC CS 510
- Brandon Fain, “Everything Data”, Duke CS 216