Machine Learning 101

Richard Corrado

Fat Cat Machine Learning

github.com/richcorrado

Goals

- Summarize some types and examples of Machine Learning.
- Examine details of parametric models and how to train them.
- Motivate and discuss Neural Networks (NNs), including Convolutional and Recurrent NNs, along with some applications.

Types of Machine Learning

Supervised Learning

Have a definite **target** that we want to compute from observed **features** of data.

Ex:

- Predict price of something.
- Does patient have some disease?
- Image and facial recognition.

Unsupervised Learning

Have data with observed features, but not a clear idea of what can be predicted.

Approaches include:

- **Clustering**: How do features segment data into groups of similar examples? Ex: Segment population into communities/market segments, also genetic sequencing.
- **Anomaly Detection**: Are some examples considerably different from all of the rest? Ex: Fraud detection, network intrusion.
- **Latent Variables**: Identify better features for modeling or potential targets for supervised learning on data.

Reinforcement Learning:

Train **software agent** to take **actions** in an **environment** to maximize **cumulative reward**.

Ex:

- Game agents (AlphaGo),
- Robot control (autonomous vehicles).

Supervised Machine Learning

From some quantities $\mathbf{x}$ (the **features**), predict a value for a quantity y (the **target**). In principle, there is some functional relationship

$$y = f(\mathbf{x}).$$

However:

- Detailed form of f is unknown (complexity of underlying system).
- Predictors $\mathbf{x}$ may be incomplete or imperfect (complexity of data, randomness). Even if we knew f, could only compute within some margin of error

$$f(\mathbf{x}) = y \pm \epsilon.$$

Machine Learning (ML) includes the study and application of algorithmic models to find approximations

$$\hat{f}(\mathbf{x}) \approx f(\mathbf{x}).$$

Statistical principles of validation, inference, etc., are used to determine the errors of the models.

Regression: Target y is a continuous variable.

- Prices or other expected values.
- Expected demand in number of units (approximately continuous for large numbers).
- Physical dimension of object (area, mass).

Classification: Target y takes some discrete and finite number of values.

- What is the species of the object? (Iris classification)

	sepal_length	sepal_width	petal_length	petal_width	species	species_name
94	5.6	2.7	4.2	1.3	1	versicolor
48	5.3	3.7	1.5	0.2	0	setosa
122	7.7	2.8	6.7	2.0	2	virginica
76	6.8	2.8	4.8	1.4	1	versicolor
4	5.0	3.6	1.4	0.2	0	setosa
40	5.0	3.5	1.3	0.3	0	setosa
92	5.8	2.6	4.0	1.2	1	versicolor
32	5.2	4.1	1.5	0.1	0	setosa
22	4.6	3.6	1.0	0.2	0	setosa
149	5.9	3.0	5.1	1.8	2	virginica

- Image recognition

Parametric Models

In many models $\hat{f}(\mathbf{x}, \boldsymbol{\theta})$ depends on both the features $\mathbf{x}$ and some **parameters** $\boldsymbol{\theta}$.

Examples:

- **Linear models**:

$$\mathbf{x} = (x_1, \ldots, x_F),$$
$$\boldsymbol{\theta} = (\theta_0, \ldots, \theta_F),$$
$$y = \boldsymbol{\theta} \cdot \mathbf{x} = \theta_0 + \theta_1 x_1 + \cdots \theta_F x_F.$$

- **Generalized Linear models (GLMs)**:

$$y = \boldsymbol{\theta} \cdot \boldsymbol{g}(\mathbf{x}) = \theta_0 + \theta_1 g_1(\mathbf{x}) + \cdots \theta_N g_N(\mathbf{x}).$$

Training Parametric Models

Statistical Intuition: Want to maximize the probability that our model predicts the correct target:

$$p(y|\mathbf{x}\boldsymbol{\theta}) = \prod_i p(y_i|\mathbf{x}_i\boldsymbol{\theta}).$$

Technical point: Maximizing this probability is equivalent to minimizing

$$J(\boldsymbol{\theta}) = -\ln p(y|\mathbf{x}\boldsymbol{\theta}) = -\sum_i \ln p(y_i|\mathbf{x}_i\boldsymbol{\theta}).$$

This function is called the **cost/loss function**. Exchanging the product for a sum is a useful computational simplification.

Given training data $\{(\mathbf{x}_i, y_i)\}$, find values $\boldsymbol{\theta}$ that minimize $J(\boldsymbol{\theta})$.

Example: If y is a continuous response, assume that the errors between the true and predicted values

$$\epsilon = \mathbf{y} - \hat{\mathbf{y}}$$

are normally distributed, then

$$p(y|\mathbf{x}) = \mathcal{N}(y; \hat{y}, \sigma^2) = \prod_i \mathcal{N}(y_i; \hat{y}_i, \sigma^2),$$

$$\mathcal{N}(y_i; \hat{y}_i, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left(-\frac{1}{2\sigma^2}|y_i - \hat{y}_i|^2\right).$$

Then

$$J(\boldsymbol{\theta}) = -\sum_i \ln p(y_i|\mathbf{x}_i\boldsymbol{\theta}) \sim \sum_i |y_i - \hat{y}_i|^2$$

We recognize J as the residual sum of squares.

Gradient Descent

Cost function is minimized when

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = 0.$$

At general $\boldsymbol{\theta}$, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \neq 0$, but consider the shift

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

where $\alpha > 0$ is a small number. Then by Taylor expansion

$$J(\boldsymbol{\theta}') \approx J(\boldsymbol{\theta}) - \alpha |\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})|^2 < J(\boldsymbol{\theta}).$$

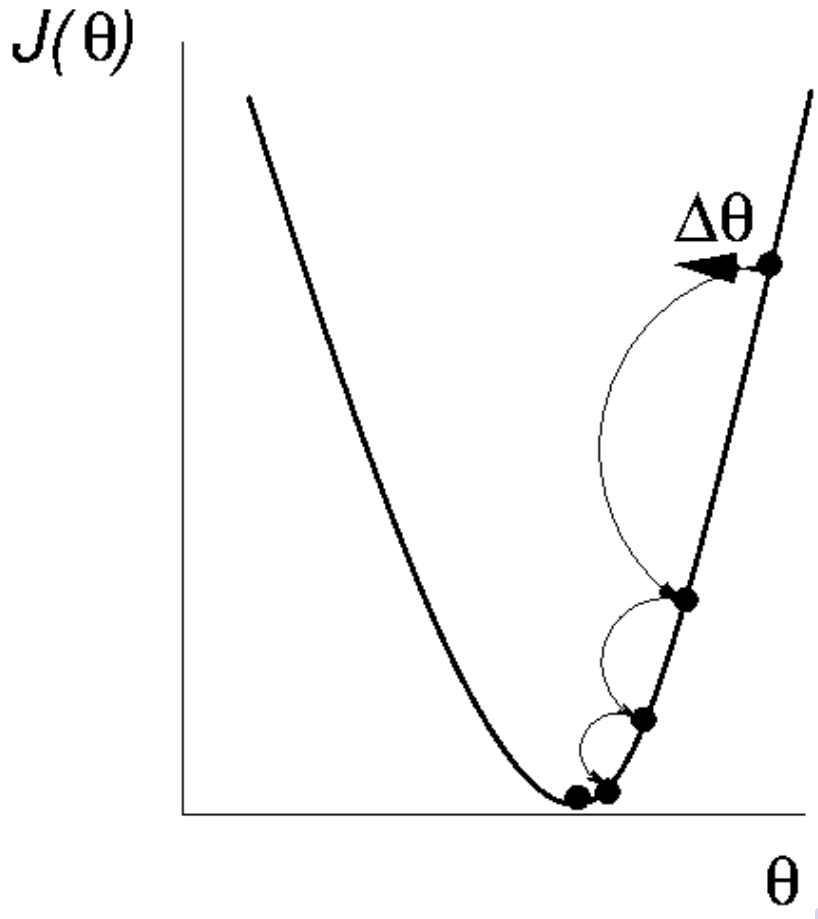
We have reduced the cost function by this change of parameters.

Gradient descent algorithm:

while $J(\boldsymbol{\theta}) > \delta$: $\quad$ # tolerance parameter $\delta > 0$

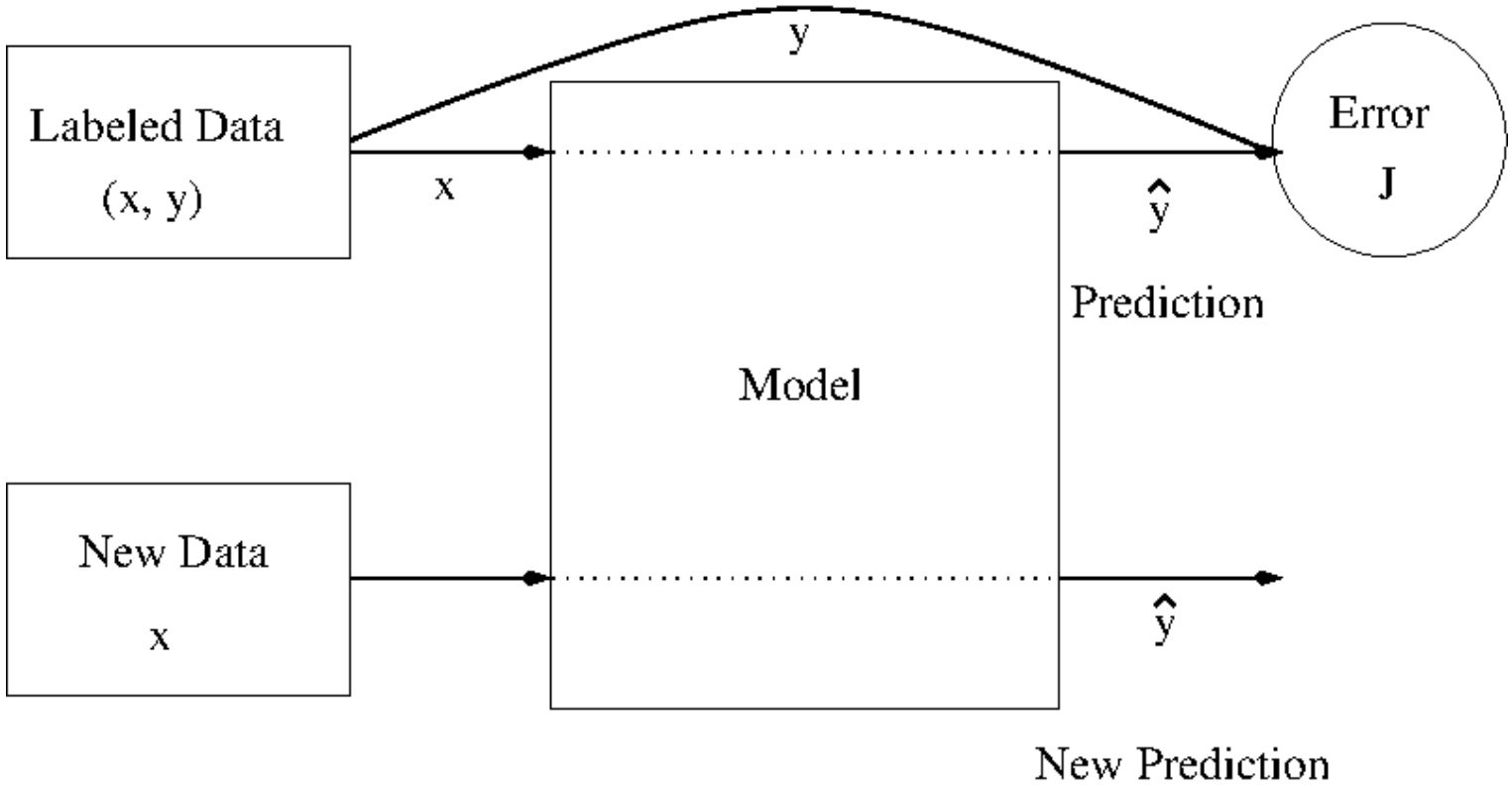
$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

α is usually called the **learning rate**.



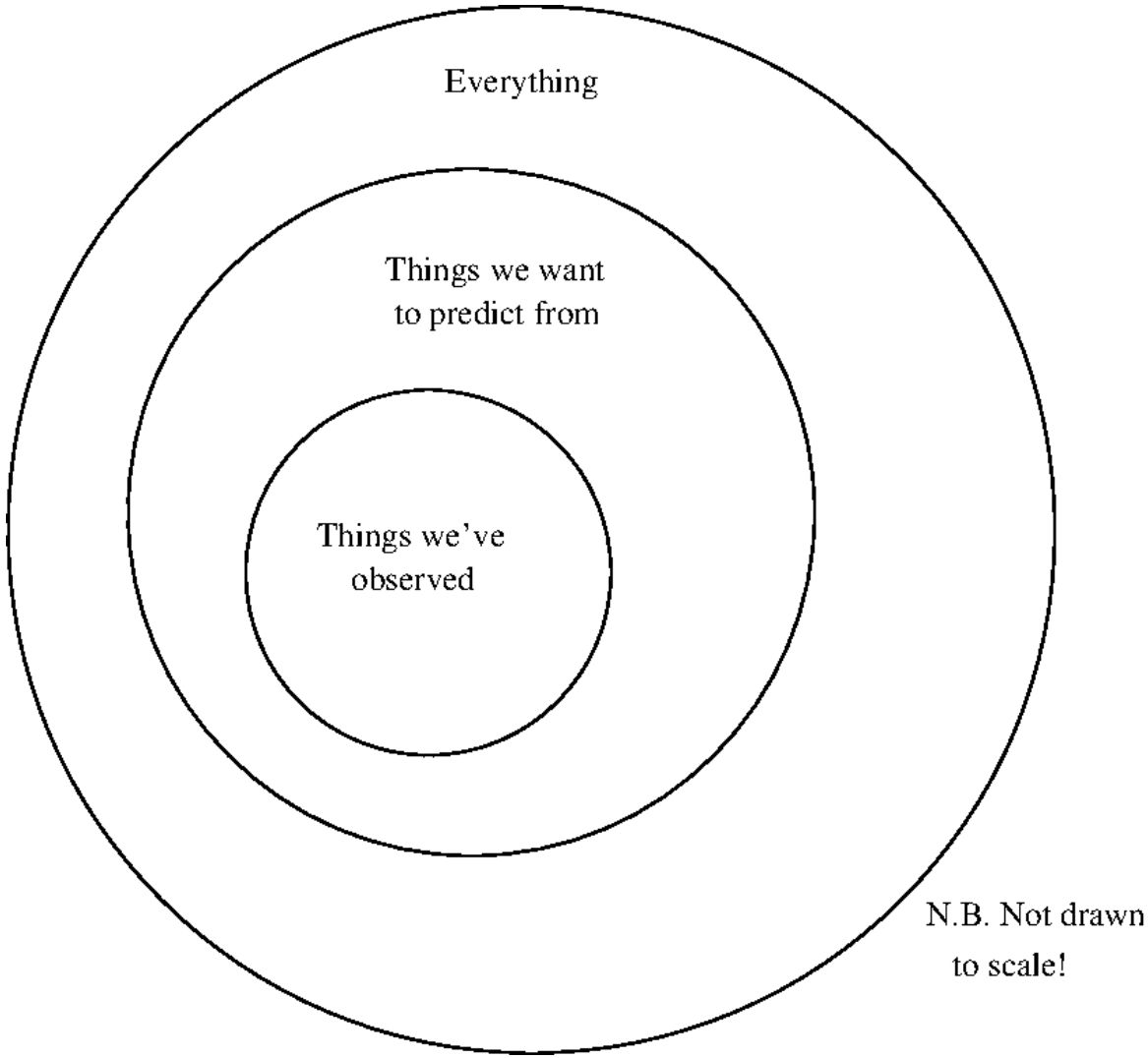
Validation of Models

We train our models because we want to make predictions on new (future) data:

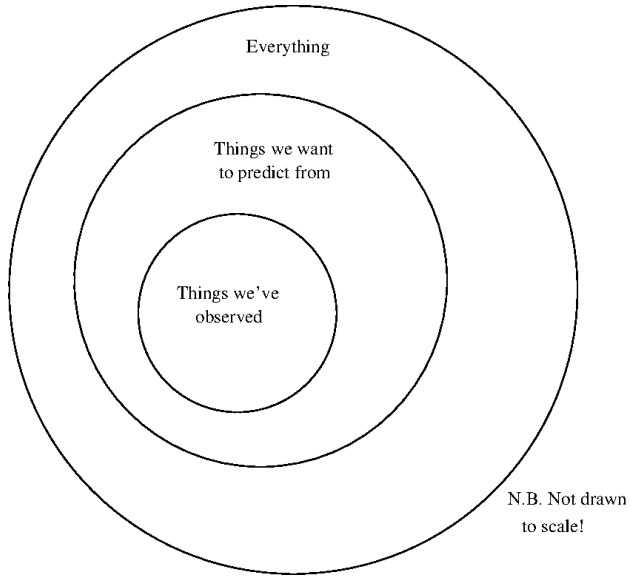


For new data, we don't know the target (wouldn't need a model if we did), so how can we be confident that our prediction is good? $\longrightarrow$ **validation**!

Sampling and Statistics



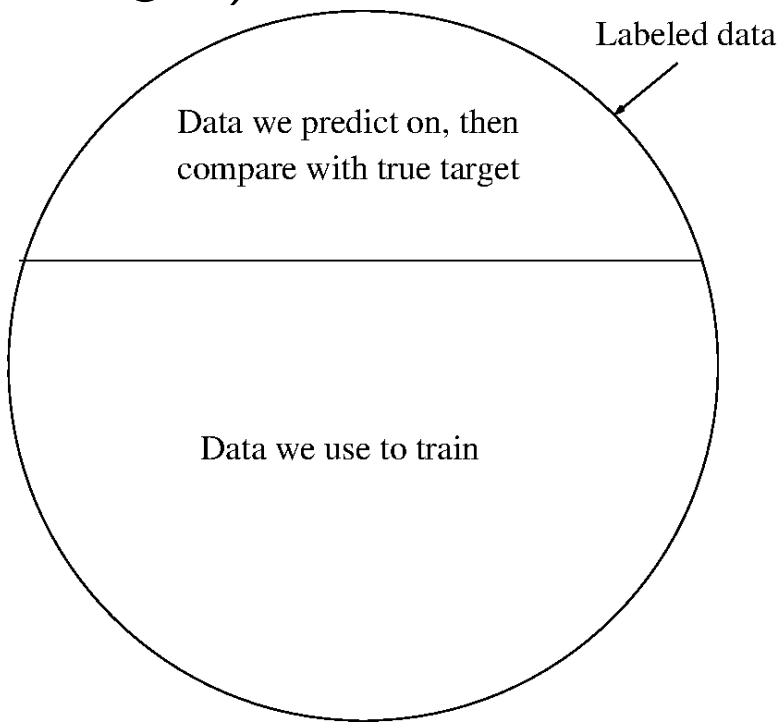
Sampling and Statistics



If observations are chosen randomly, for a large enough sample, statistics like mean, standard deviation should be very close to those of the whole population $\longrightarrow$ want our labeled data to be representative of the population.

Train - Test Split

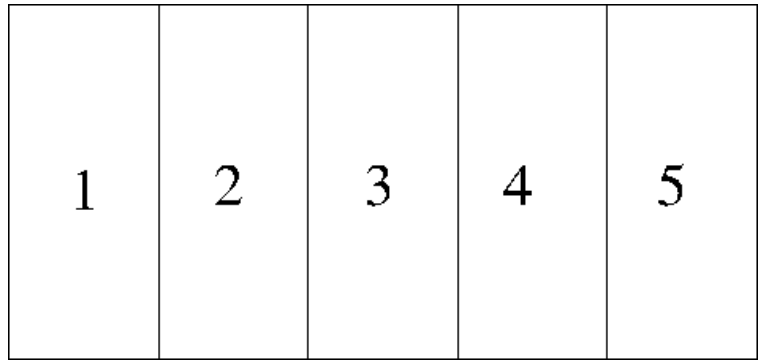
If our labeled data is representative, so will a randomly chosen subset (if "large enough").



Training data $\longrightarrow$ Used to train the model.
Test data $\longrightarrow$ Held out, only used to test the model predictions.

Cross-Validation

More sophisticated schemes are often used, such as **K-fold cross-validation** :



1. Train on $(1, 2, 3, 4)$, compute test error on 5.
2. Train on $(1, 2, 3, 5)$, compute test error on 4.
3. ...

Examine average and standard deviation of test errors.
Pro: Uses all of the labeled data.
Cons: Computational expensive, can't use for time series.

Bias vs. Variance Tradeoff

In any ML problem must look out for:

- **Bias or underfit** if our model is too simple to faithfully match the data. Can also say that the model has a low **capacity** to absorb the information in the training data.

  Ex: Using linear model when data is clearly nonlinear.

- **Variance or overfit** if our model agrees so well with training data that it fails to generalize to new data. Can say that the model has too much capacity and learns the training data too well.

  Ex: Using model with number of parameters that is very large compared to number of features, number of examples in training set.

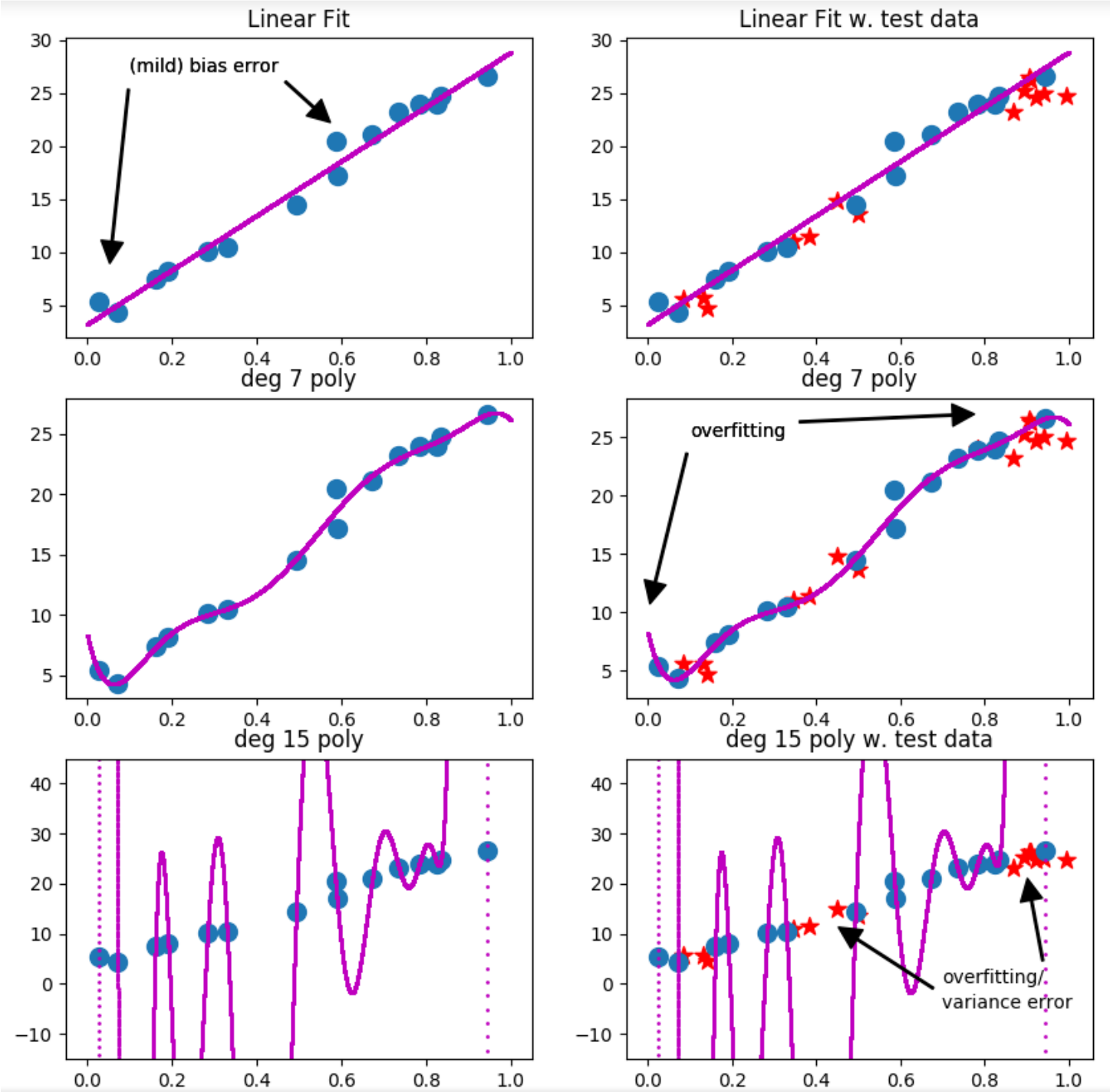
Ex: Polynomial regression on simulated data.

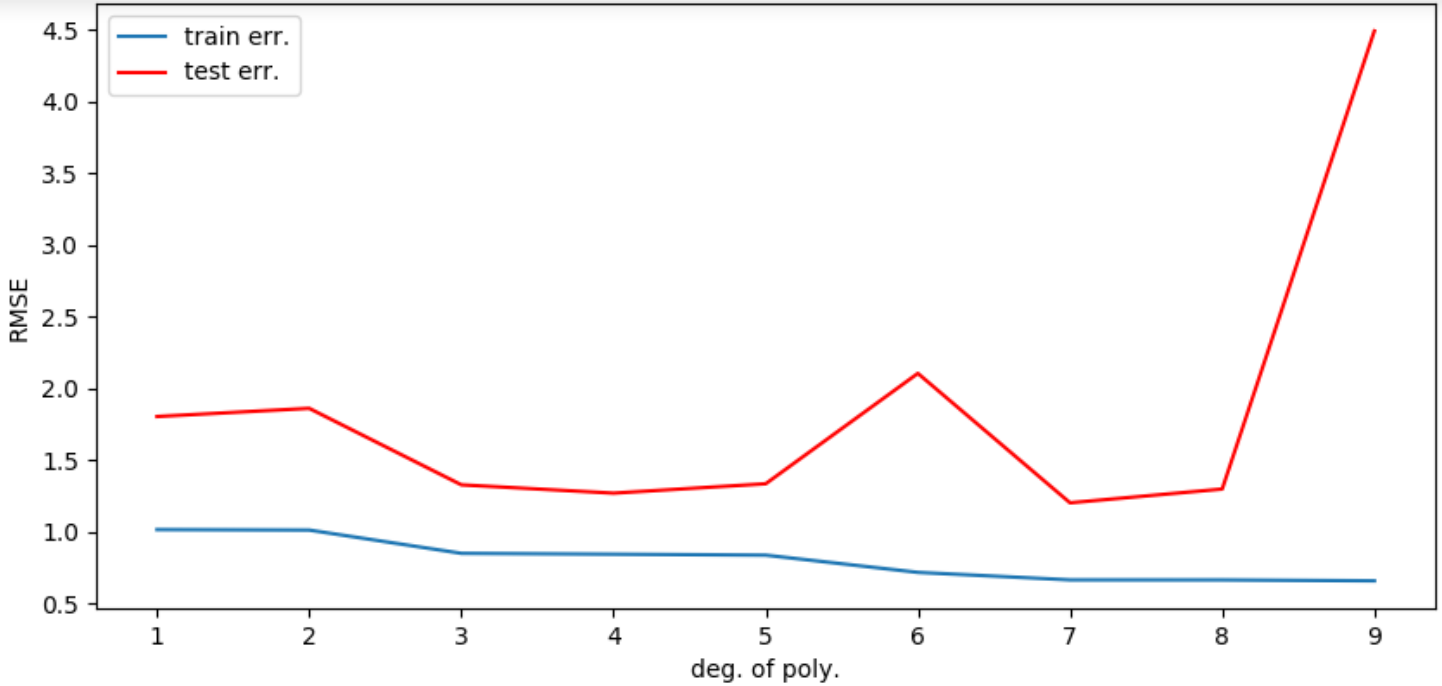
```
n_train = 15
n_test = 15

x_train = np.random.uniform(0, 1, n_train)
x_test = np.random.uniform(0, 1, n_test)
x_grid = np.linspace(0, 1, 100000)

p3 = np.array([-44, 66, -1, 5])
y_train = np.polyval(p3, x_train) + np.random.normal(scale=1.0, size=n_train)
y_test = np.polyval(p3, x_test) + np.random.normal(scale=1.0, size=n_test)
```

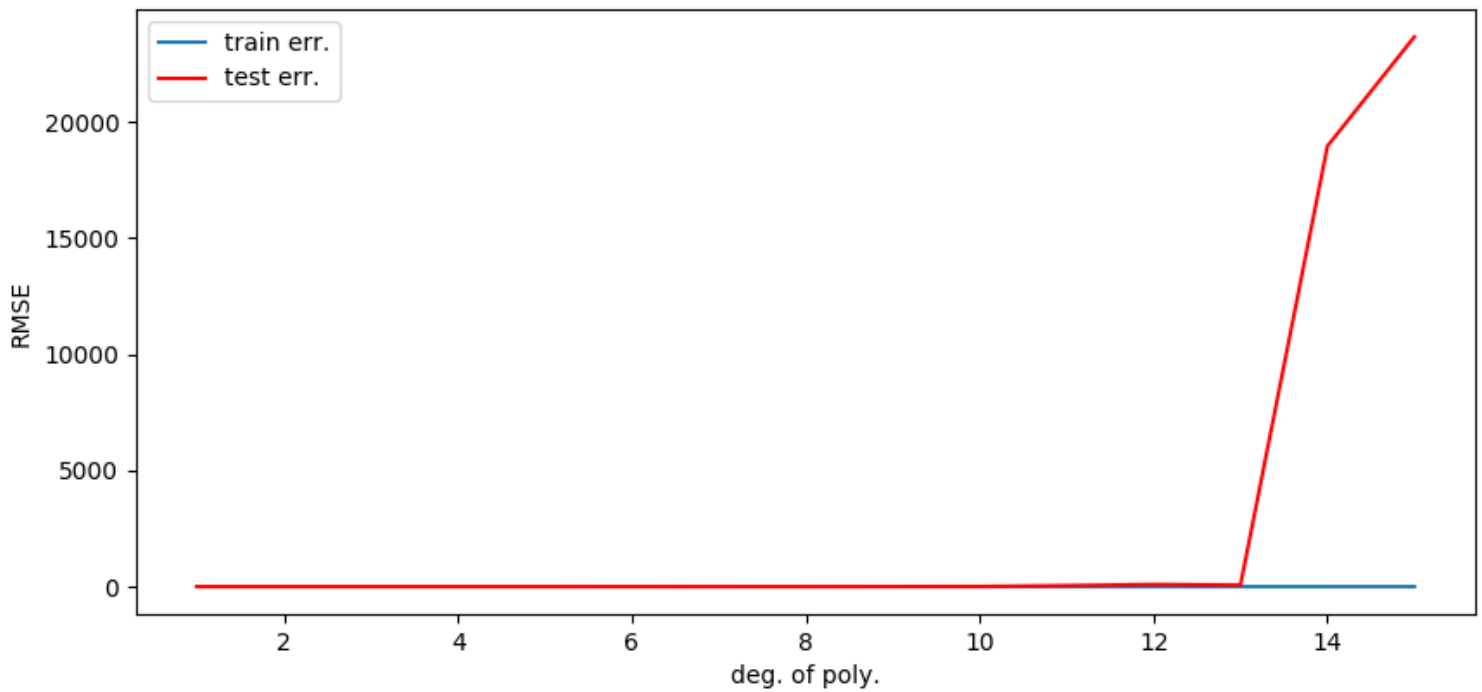
```
lm = LinearRegression()

n_models = n_train

ymodels = []
train_err = []
test_err = []

for i in range(1,n_models+1):
    pmodel = Pipeline([('poly', PolynomialFeatures(degree=i)),
                       ('linear', LinearRegression(fit_intercept=False))])
    pmodel.fit(x_train.reshape(-1, 1), y_train)
    ymodels.append(pmodel.predict(x_grid.reshape(-1, 1)))
    y_train_pred = pmodel.predict(x_train.reshape(-1, 1))
    y_test_pred = pmodel.predict(x_test.reshape(-1, 1))
    train_err.append(np.sqrt(np.mean(np.square(y_train_pred - y_train))))
    test_err.append(np.sqrt(np.mean(np.square(y_test_pred - y_test))))
```

```
print("Optimal training error for degree: ", np.argmin(train_err)+1)
print("Optimal test error for degree: ", np.argmin(test_err)+1)
opt_degree = np.argmin(test_err)+1
```

```
Optimal training error for degree:  15
Optimal test error for degree:  7
```


Linear Fit
(mild) bias error
Linear Fit w. test data
deg 7 poly
deg 7 poly
overfitting
deg 15 poly
deg 15 poly w. test data
overfitting/
variance error


train err.
test err.
RMSE
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
1
2
3
4
5
6
7
8
9
deg. of poly.


train err.
test err.
RMSE
20000
15000
10000
5000
0
2
4
6
8
10
12
14
deg. of poly.

For this data, the linear model

$$y = \theta_0 + \theta_1 x$$

is reasonable, but a generalized linear model

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots \theta_N x^N,$$

with $N = 4, \cdots, 7$ has better performance.

Beyond Generalized Linear Models

When we have many features and a large training set, it is time-consuming to find suitable functions $\boldsymbol{g}(\mathbf{x})$ to use in a generalized linear model.

- Must examine the relationship between each feature x_i and the target y and try to determine a suitable g_a.
- Must also examine relationships between the features x_i, since the best g_a might be multivariate $g(x_1, x_2, \ldots)$.

Note that we can think of the g_a as a new set of features that are "engineered" from the original features x_i.

Neural networks (NNs) can be thought of as providing a set of algorithmic methods to learn **new features**. We hopefully trade expensive manual feature engineering for lower-cost computational resources.

Neural Networks

In order to keep the training process mathematically stable and computationally managable, NNs are based on fairly simple functions:

- Linear functions:

$$\mathbf{z^{(i)}} = \mathbf{f}^{(i-1)}\mathbf{W}^{(i)} + \mathbf{b}^{(i)},$$

$$\mathbf{W}^{(i)}: \quad \text{weights, with shape } (F_{i_1}, F_i),$$

$$\mathbf{b}^{(i)}: \quad \text{biases, with shape } (F_i).$$

- Nonlinear functions, called **activation functions**:

$$\mathbf{f}^{(i)} = g^{(i)}(\mathbf{z}^{(i)}).$$

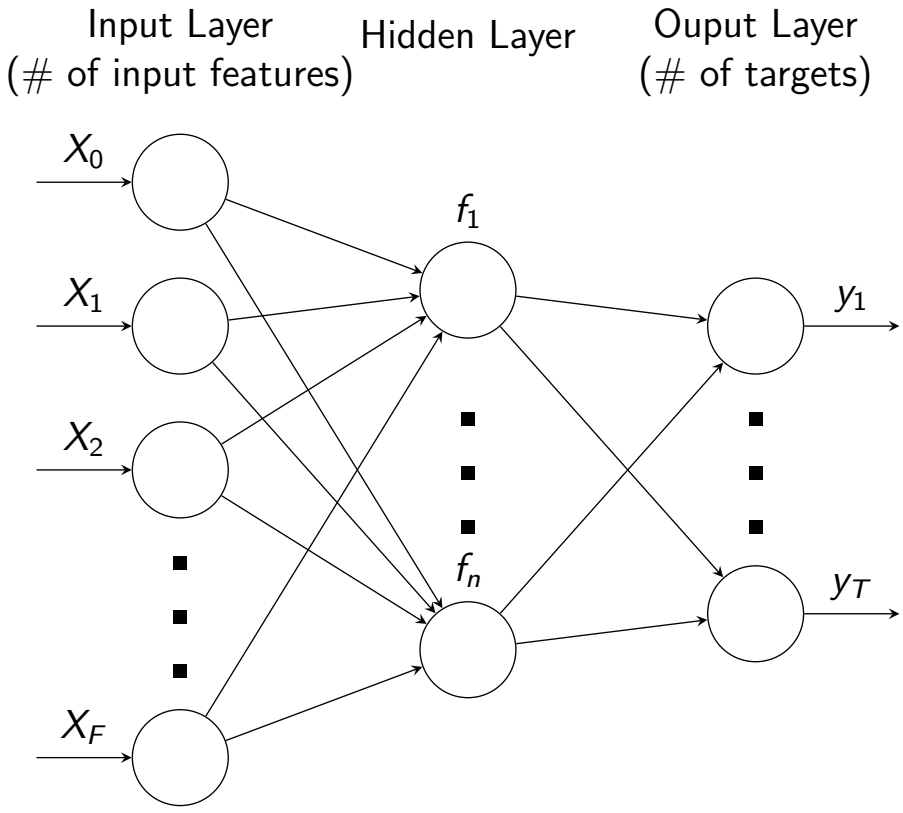
A single layer of a neural network computes new features

$$\mathbf{f}^{(i)} = g^{(i)}\left(\mathbf{f}^{(i-1)}\mathbf{W}^{(i)} + \mathbf{b}^{(i)}\right),$$

where $\mathbf{f}^{(i-1)}$ are the features output by the previous layer.
Terminology for the layers is:

- **Input layer**: The first layer of the network is designed to simply load in the input features $\mathbf{X}$ present in the dataset.
- **Output layer**: The last layer is designed to produce an output that can be compared directly to the targets $\mathbf{y}$.
- **Hidden layers**: The layers in between compute the new features $\mathbf{f}^{(i)}$. Not connected to input or output.
  Width: Number of new features of a single hidden layer.
  Depth: Total number of hidden layers in the network. If depth is ≥ 3 we can say that we have a **deep neural network**.

Single Hidden Layer



A network like this, where every input feature is connected to a new feature is called a **feedforward network** or, in older terms, a **multilevel perceptron** (MLP).

Composition of Functions

Recall that a typical goal of machine learning is to approximate the functional relationship between the input features $\mathbf{X}$ and some target output $\mathbf{y}$:

$$\mathbf{y} = f(\mathbf{X}).$$

NNs naturally arrive at an approximation of this function via the composition of the functions learned at the hidden layers:

$$\hat{f}(\mathbf{X}) = f^{(H)} \circ \cdots \circ f^{(1)}(\mathbf{X}).$$

Activation Functions

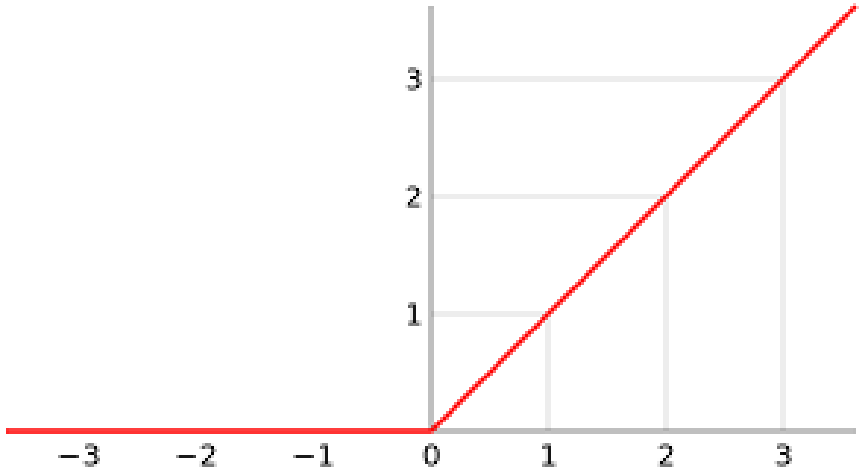
The activation functions play a big role in allowing NNs to learning more general functions than linear models alone. There are many possible choices, but the important characteristics are:

- **Nonlinearity**: Key in allowing approximation of general functions, important for validity of **universal approximation theorem**.
- **Differentiability**: Optimization of cost function uses **backpropagation** through network to update weights and biases during training, need to have stable computations of gradients.

Other characteristics like finite range, monotonicity, smoothness, etc., can be beneficial, but are not considered essential.

Rectified Linear Unit (ReLU)

$$g(z) = \max(0, z) = \begin{cases} z, & z \geq 0 \\ 0, & z < 0. \end{cases}$$



- **Sparse Activation**: Negative inputs are set to zero.
- **Finite Gradients**: Differentiable on $\mathbb{R}\backslash\{0\}$, with $g'(z) = 0, 1$.
- Fast computations.

Universal Approximation Theorem

Theorem: For a suitable choice of activation function, even a NN with a single hidden layer can approximate any function that we could reasonably expect to encounter in real-world machine learning problems (Borel measurable).
Caveat: The approximation error can be made arbitrarily small if the **width** of the hidden layer is allowed to be **arbitrarily large**. An arbitrarily wide NN will have **many free parameters** and will be **prone to overfitting** unless we also have an appropriately large and diverse training set.
Deep Learning: Add many hidden layers, learn larger classes of functions for smaller widths.

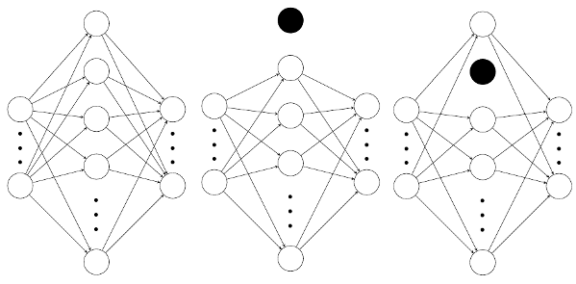
Modifications of Feedforward Architecture

In a fully-connected feed-forward network, each hidden layer feature is connected to a feature from the previous layer

$$z_{a_i}^{(i)} = \sum_{a_{i-1}=0}^{F_{i-1}-1} f_{a_{i-1}}^{(i-1)} W_{a_{i-1}a_i}^{(i)} + b_{a_i}^{(i)}.$$

We can modify the rules for computing the new features, creating new **architectures**, new types of NNs.

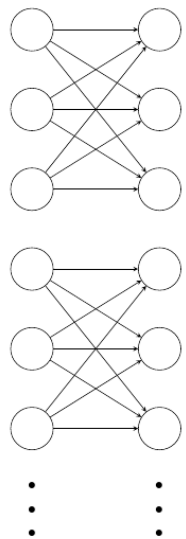
1. We can force some of the weights $\mathbf{W}_{a_{i-1}a_i}^{(i)}$ to zero. This generally results in fully-connected models with a smaller capacity.



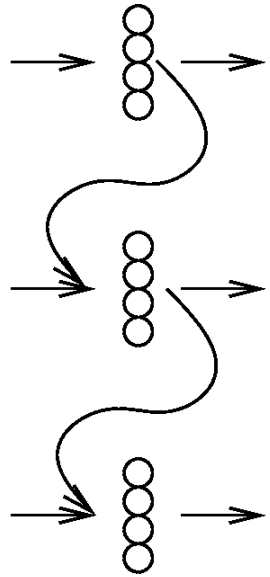
2. We can impose some symmetry on the weights $\mathbf{W}^{(i)}_{a_{i-1}a_i}$, forcing some components to be equal to one another. This is called **weight sharing**.
3. We can remove some connections to the previous layer. This is equivalent to replacing

$$\mathbf{f}^{(i-1)}\mathbf{W}^{(i)} \rightarrow \sum_{a_{i-1}\in D} f^{(i-1)}_{a_{i-1}} W^{(i)}_{a_{i-1}a_i},$$

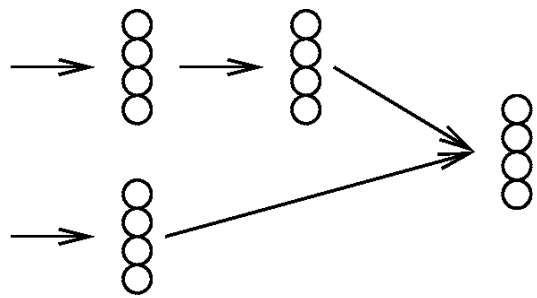
where D is some subset of the index set for the incoming features $\mathbf{f}^{(i-1)}$.



4. Can have more general connections between layers.
   - **Recurrent** networks have a chain structure between layers.

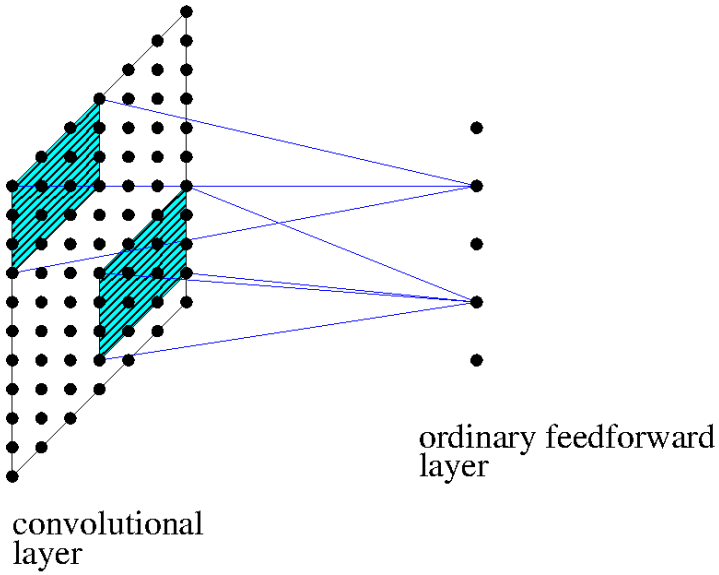


   - **Recursive** networks have a tree structure.



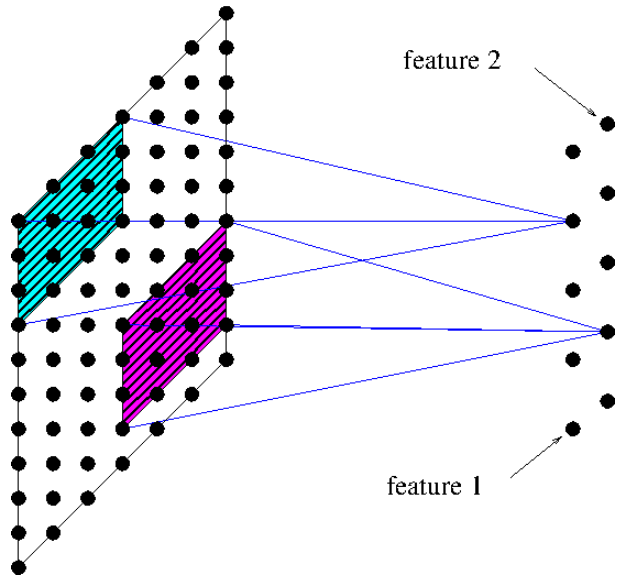
Convolutional Neural Networks

Convolutional NNs (CNNs) are designed to preserve information about how pixels in 2D images are spatially related:

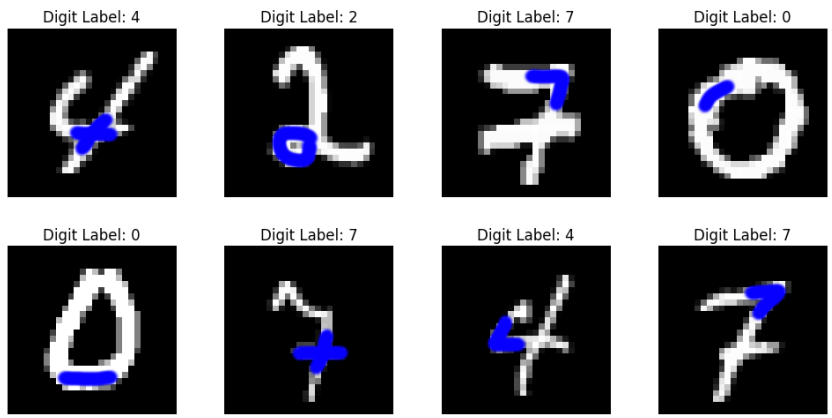


- **Partial connectivity**: Sample nearby pixels using a window.
- **Weight sharing**: Weights are shared over the whole image.

Filter: Collection of weights forming a window. Can use different filters to compute multiple types of features for a given image:



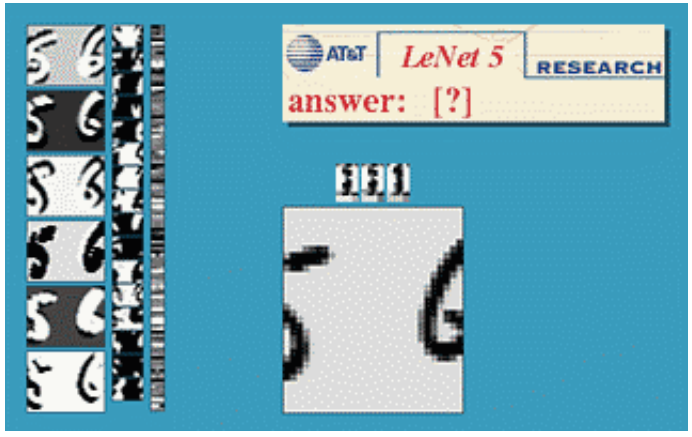
Translational invariance: CNN filters tend to learn geometric shapes, regardless of where they appear in an image:



Computer Vision

Since CNNs are well-suited for processing 2D images (but can also be used for 3D and non-image problems), they are extensively used in machine learning with images.

- **Character recognition**: MNIST digits, zip codes:



Google Streetview: **localization** and character recognition

100 vs. 676


1110 vs. 2641


23 vs. 37

1 vs. 198


4 vs. 332

2 vs 239

1879 vs. 1879-1883

228 vs. 22B

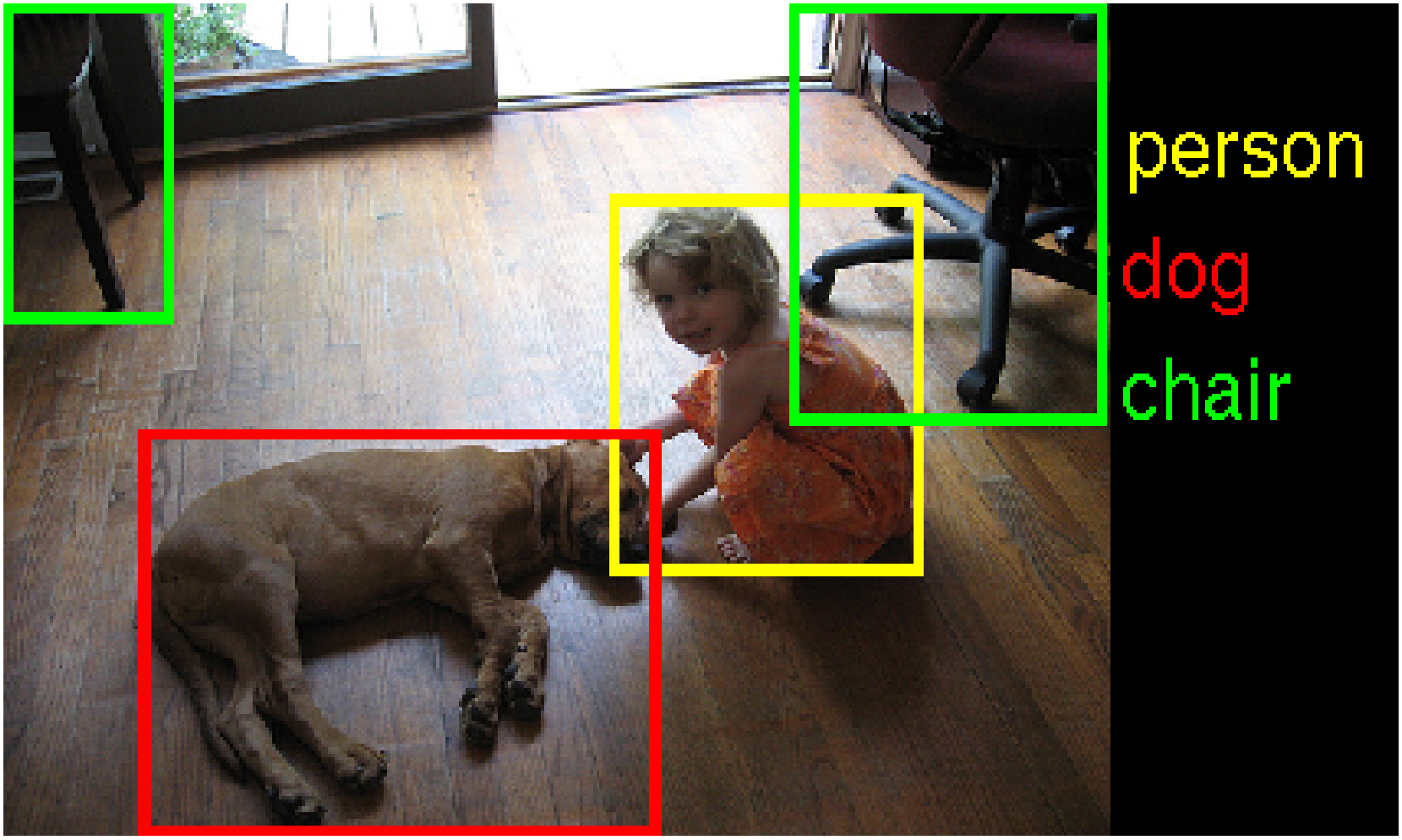
96 vs. 86

1844 vs. 184

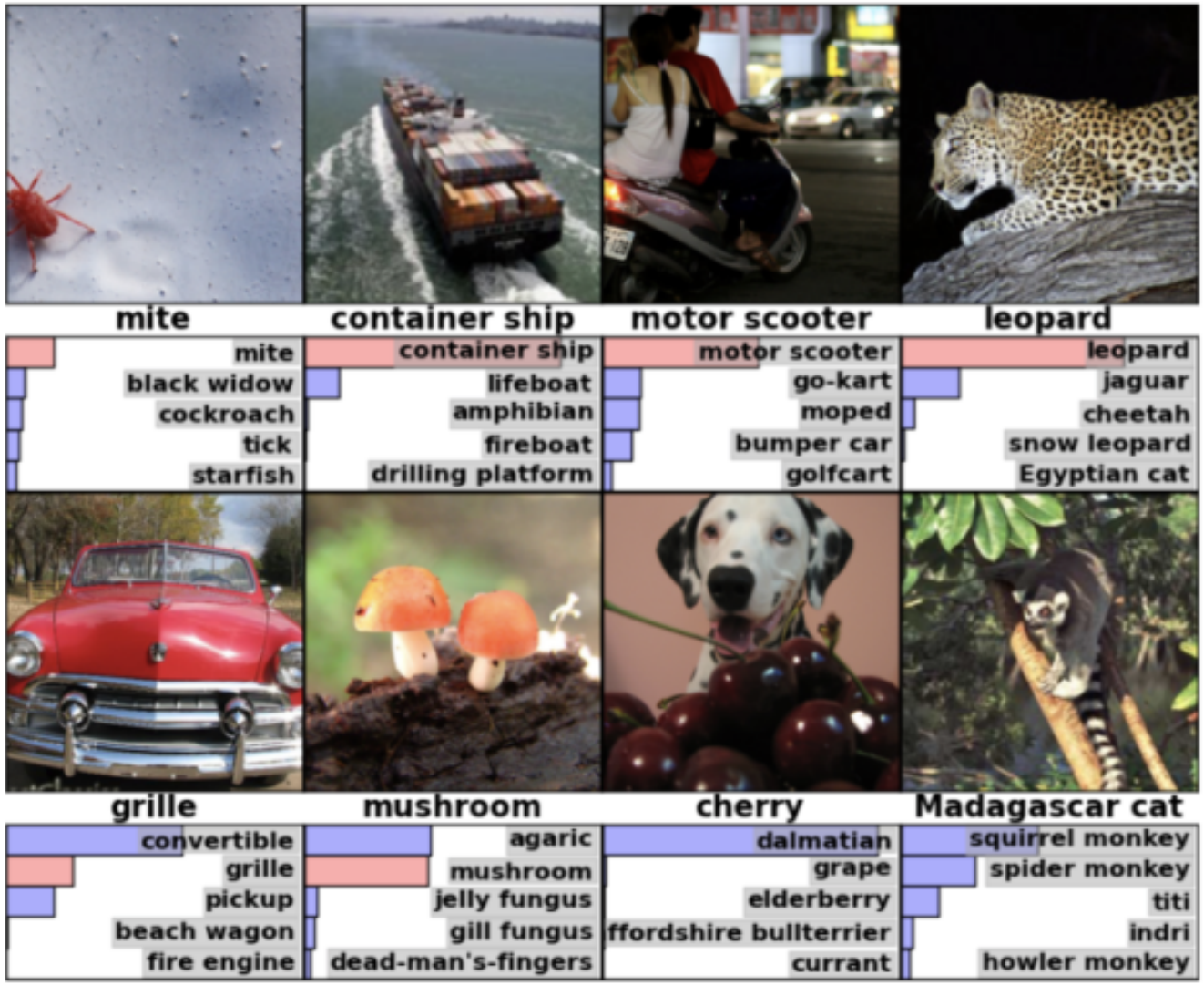
62 vs. 62-37

1180 vs. 1780

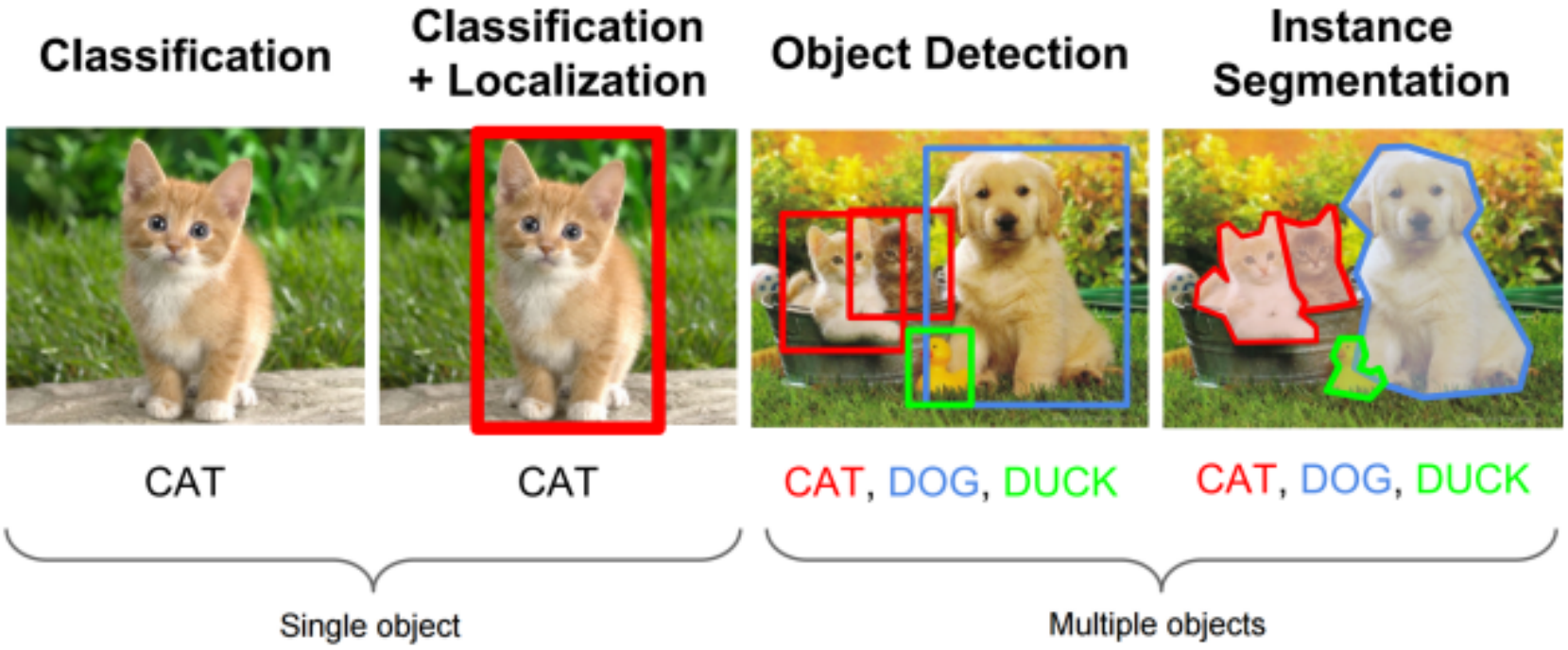
ImageNet: **object detection**



ImageNet: **classification**

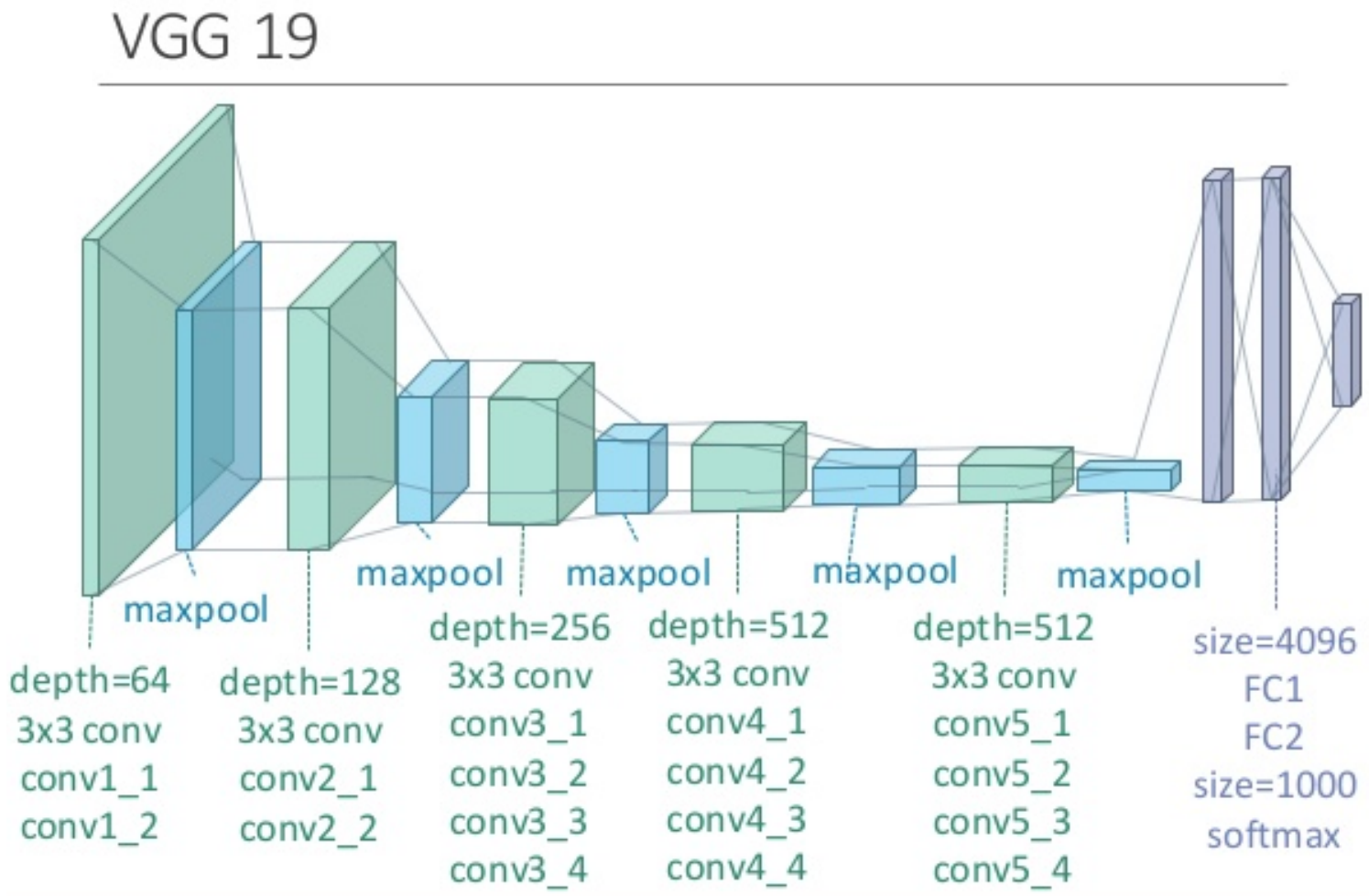


ImageNet: classification + **localization**



VGG-Network

Simonyan and Zisserman, ICLR 2015, arxiv:1409.1556,
1st/2nd in localization/classification, ImageNet ILSVRC-2014.



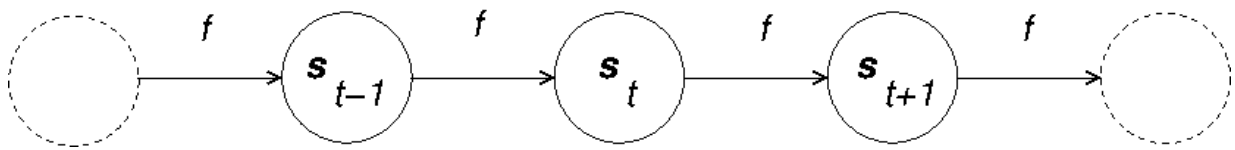
Recurrent Neural Nets

Time-varying data: $(\mathbf{X}_t, \mathbf{y}_t) = \mathbf{s}_t \longrightarrow$ **state** of system.

Causality: state at time τ, $\mathbf{s}_\tau$ must only depend on states at times $t < \tau$.

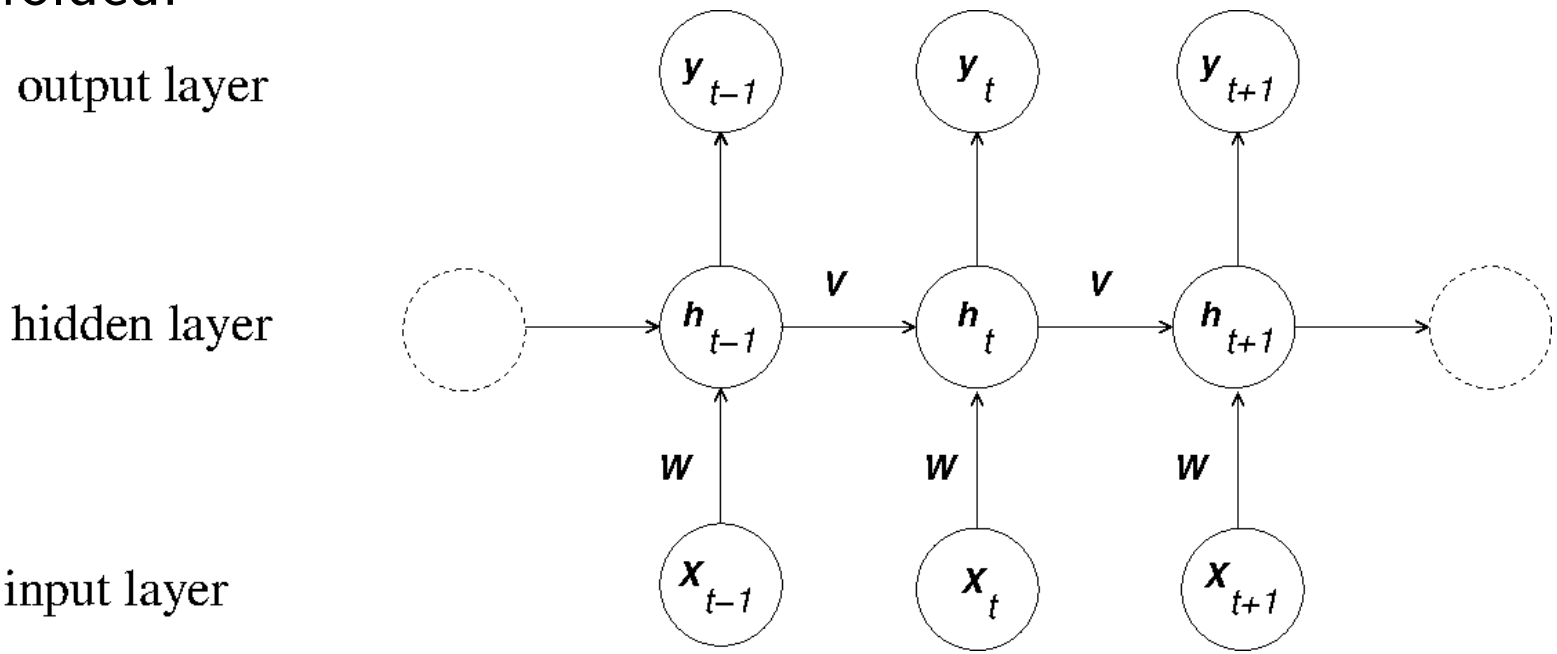
Dynamical system:

$$\mathbf{s}_t = f(\mathbf{s}_{t-1}) = f(f(\mathbf{s}_{t-2})) = \cdots .$$

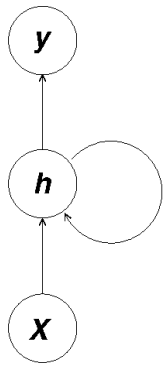


Recurrent Neural Network to Approximate f

Unfolded:



Folded:

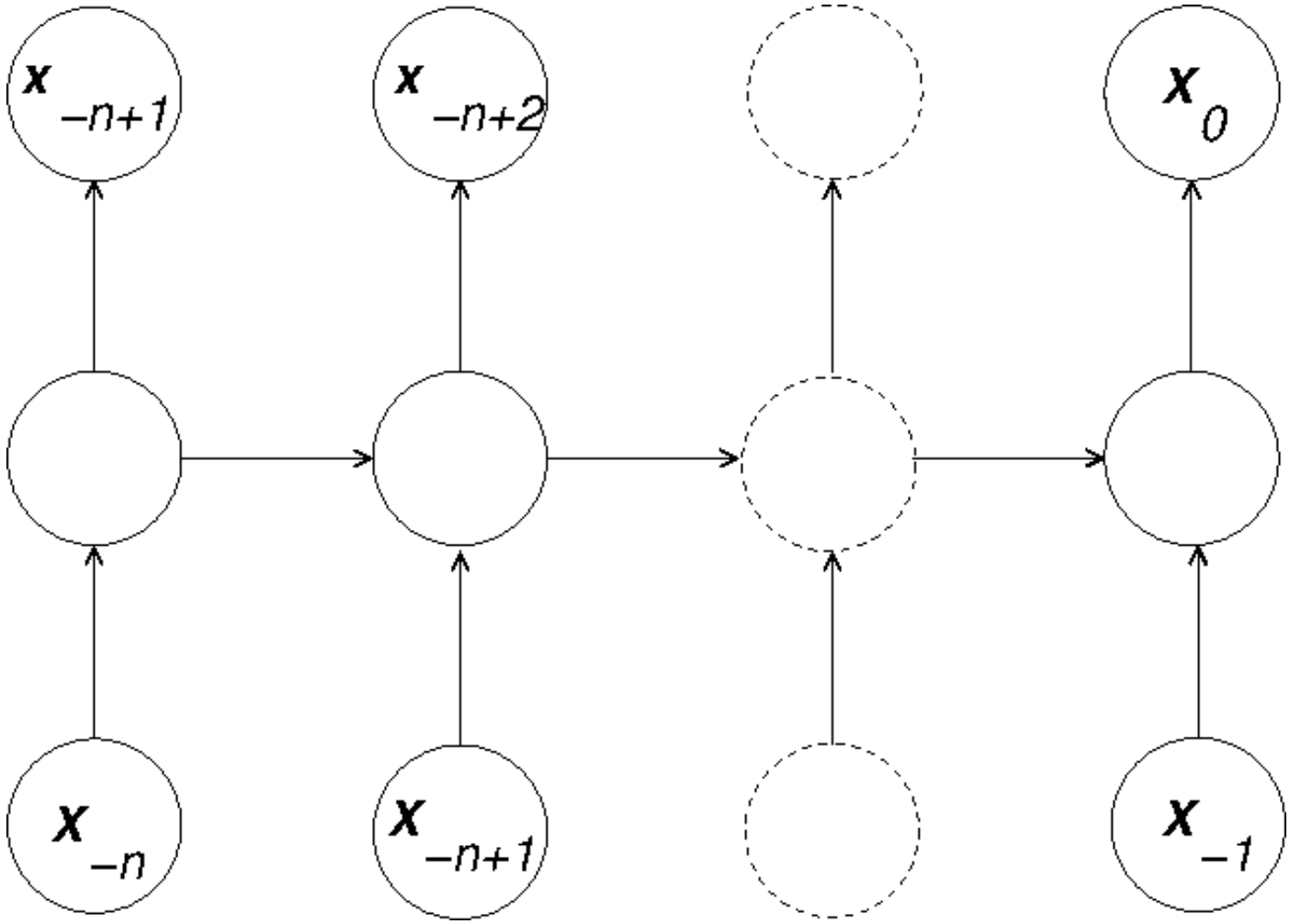


RNN Applications

Important when sequential relationship is important.

- Time series: stock price, weather, resource demand
- Language processing: word order $\rightarrow$ grammar, semantics
  spell/grammar-checking, translation
- Audio: speech recognition, signal processing/detection

For **prediction**: train on $(x_{-n}, \cdots x_{-1})$,
with targets $(x_{-n+1}, \cdots, x_0)$:



Credits:

Several images and other graphics in this presentation are reproduced as a fair use of the original sources:

- The original source of the photo used in the twitter post on page 9 appears to be yamesjames@reddit.
- The image of the rectified linear unit is taken from wikimedia).
- The graphic on page 33 was copied from the section of Yann LeCun's website on LeNet-5.
- The image on page 34 was obtained from Goodfellow et al., arxiv:1312.6082.
- The image on page 35 was obtained from ImageNet.
- The image on page 36 is from Krizhevsky et al., NIPS 2012.
- The image on page 37 is from Stanford CS231n lecture slides, obtained here via C. Körner.
- The figure on page 38 was obtained from slides for a lecture by M. Chang in the Applied Deep Learning course by Y. Chen.

The original creators of the content highlighted and linked to have my thanks and appreciation.