Final preview

Glenn Bruns

CSUMB

many of the figures and code examples in these slides come from Gerón, *Hands-on Machine Learning with Scikit-Learn and TensorFlow* (O'Reilly)

Main learning outcomes

When you finish this class, you should be able to:

- explain the optimization framework of machine learning
- apply numpy/Pandas/Scikit-learn to classic machine learning problems
- apply advanced ML methods (e.g. ensemble methods) intelligently
- write TensorFlow code for tasks like:
  - classic machine learning (feedforward)
  - image classification (CNN)
  - text processing (RNN)
  - time series prediction
- explain and apply ARIMA models for time series forecasting

Coverage of exam

All material including neural nets

About half of exam will be on neural nets

- training deep neural nets
- convolutional neural nets
- recurrent neural nets

The other half will be on earlier material.

All topics in course will be covered.

Structure of exam

1. Concepts and theory (50 minutes)
   - paper and pencil – no notes or other resources
   - conceptual questions
   - questions about the math
   - questions about neural net examples (e.g. text examples, Karpathy code)
2. Practical (65 minutes)
   - add your code to starter iPython notebook
   - use any resources you like (except fellow students)
   - part a: machine learning with Numpy/Pandas/Scikit-Learn
   - part b: neural networks

How to prepare

- Note learning outcomes at the front of each slide deck
  - ask yourself if you can do these things
- Practice on lab and homework problems
- Don’t passively review lecture slides
  - actively review by writing test questions
  - make flash cards for yourself, and use them
  - write, answer, and rate Peerwise questions

Course topics

- end-to-end machine learning
  - use Python ecosystem for machine learning
- linear algebra
- training models
  - optimization methods, regularization
- support vector machines
- ensemble learning
- dimensionality reduction
- deep learning
  - TensorFlow, neural nets, CNNs, RNNs
- time-series data

Linear algebra

Linear algebra, some key points

1. Operations on vectors: addition, multiplication by a scalar, norm, dot product
2. Operations on matrices: addition, multiplication by a scalar, multiplication, matrix inversion, transpose
3. Properties of matrix and vectors and matrix operations, such as commutativity, associativity, distributivity
4. Special matrices: square, diagonal, identity
5. Applications of matrices to ML: creating new features, transforming matrices with matrix mult.
6. Basics of determinants, singular value decomposition, eigenvalues and eigenvectors

Question

If the size of A is $m\ x\ n$, then what is the size of $A^T A$?

answer: n x n

(n x m) (m x n) = (n x n)

Question

What is the norm of vector [2, 2, 1] ? Give a simplified expression?

answer: $\sqrt{2^2 + 2^2 + 1^2} = \sqrt{9} = 3$

Question

If a matrix has an inverse, then the inverse is unique.

answer: true

Extra credit: prove it

answer:

Assume B and C are two inverses of A.

$B = BI = B(AC) = (BA)C = IC = C$

Question

The determinant of a matrix is a:

a) scalar

b) vector

c) matrix

answer: a

You should know the determinant for a 2x2 matrix:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \mathrm{ad} - \mathrm{bc}$$

Training models

Training models, key points

- gradient descent
- gradient descent for multi-variable functions
- types of gradient descent (batch, mini-batch,..)
- regularization (e.g. regularization methods)
- logistic regression
- softmax
- loss functions (log loss, cross entropy)
- extending binary classifier to multi-class classifiers

Machine Learning as optimization

steps	example: linear regression
define a **model**, with parameters, that will be used to make predictions	$\hat{y} = \theta^T \cdot \mathbf{x}$
define a **cost function** to explain what it means for the model to fit the data well	$MSE(\mathbf{X}, \theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$
if possible, find the **partial derivatives of the cost function**	$\frac{\partial}{\partial \theta_j} MSE(\mathbf{X}, \theta) = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T \cdot x^{(i)} - y^{(i)}\right) x_j^{(i)}$
fit the model to your training data by finding the parameters that **minimize the cost function, using gradient descent**	

Question

Typically, which cost function is used in classification problems to see how well estimated class probabilities match the target class?

a) softmax

b) log loss

c) cross entropy

answer: b or c !

softmax is not a loss function; log loss and cross entropy are the same in the case of binary classification

Log loss cost function

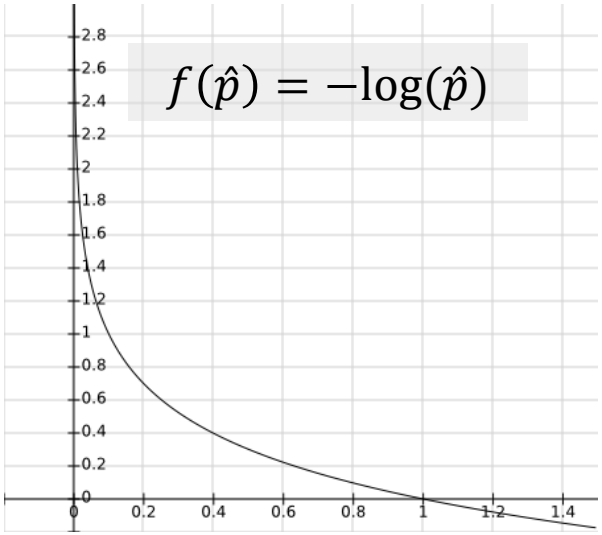
This is used with binary classification, when we predict a single probability value and want to compare to a 0/1 target value

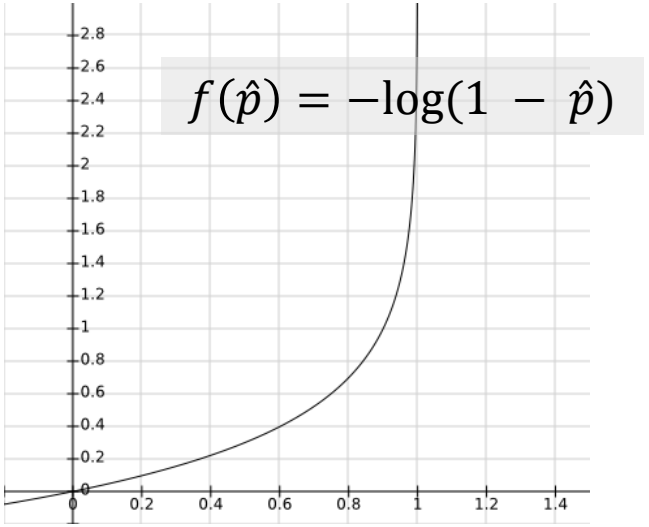
Cost for a single training example:

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

This is equal to:

$$-(y \log(\hat{p}) + (1 - y) \log(1 - \hat{p}))$$





Softmax function

This is used with multi-way classification, to transform a vector of scores to a vector of probability values (that add to 1)

softmax function:

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^{K} \exp(s_j(\mathbf{x}))}$$

Example.

Scores:

$s_1(\mathbf{x})$	$s_2(\mathbf{x})$	$s_3(\mathbf{x})$
1.7	2.4	0.5

$$\frac{\exp(1.7)}{\exp(1.7) + \exp(2.4) + \exp(0.5)} = 0.30$$

After applying softmax:

$\hat{p}_1$	$\hat{p}_2$	$\hat{p}_3$
0.30	0.61	0.09

(remember, $\exp(1.7) = e^{1.7}$)

Cost function in softmax regression

This is used with multi-way classification, when we predict a vector of probabilities and want to compare to a target class. The target class is often one-hot encoded.

Cross-entropy cost function:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{\mathrm{k}}^{(\mathrm{i})} \log(p_k^{(i)})$$

Note connection to log loss.

Example, with three instances:

target class	prediction
(1, 0, 0)	(0.1, 0.5, 0.4)
(0, 0, 1)	(0.1, 0.1, 0.8)
(0, 1, 0)	(0.2, 0.6, 0.2)

cost for this training instance is:

$1 \times \log(0.1) + 0 \times \log(0.5) + 0 \times \log(0.4) = -1$

How to handle >2 classes?

Here are two ways to build a k-way classifier from a bunch of binary classifiers:

- One-vs-all:
  - Train one binary classifier for every class
  - Run every classifier; select class with highest decision score
- One-vs-one:
  - Train a binary classifier for every pair of classes
  - Run every classifier; select class that wins the most "duels"

	C1	C2	C3
score on x	0.5	0.7	0.2

which class do we predict?

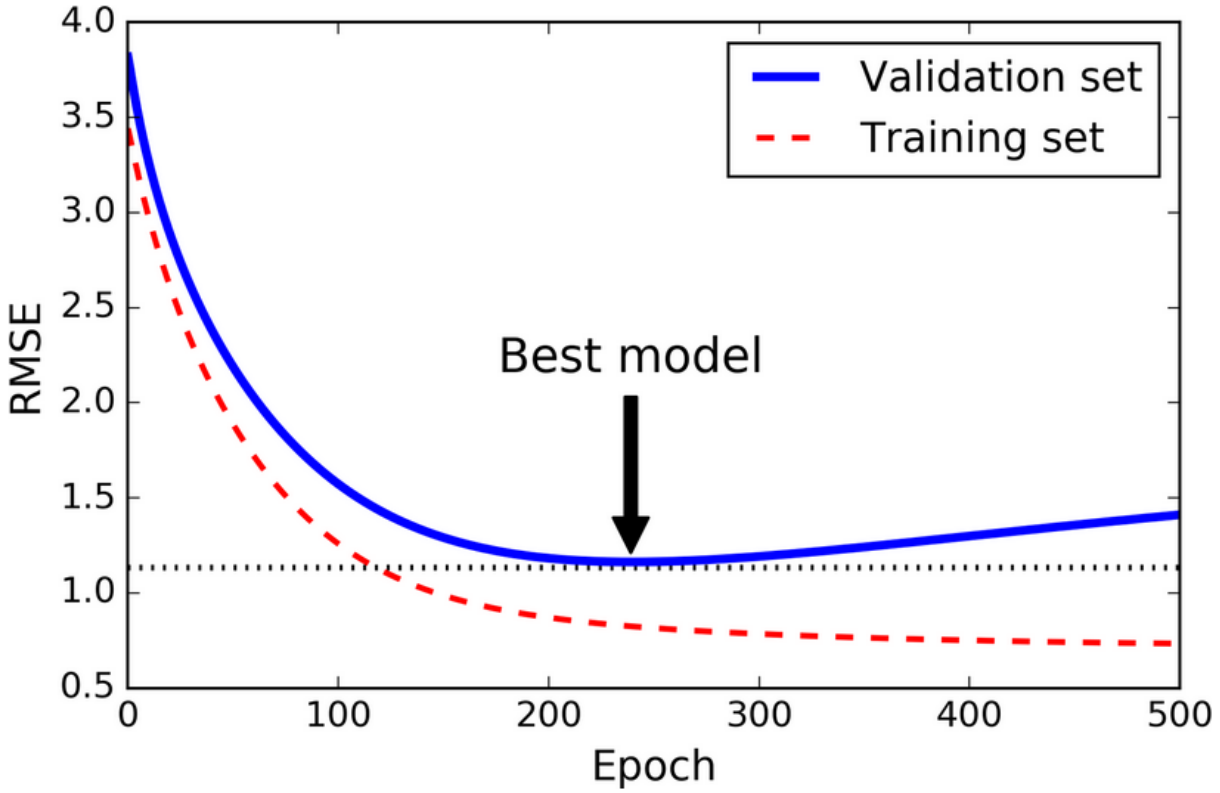
	C12	C13	C23
score on x	0.2	0.6	0.7

which class do we predict?

Question

WIth early stopping, you stop training when _____ error is minimized (fill blank with 'training' or 'test').

answer: test



Support Vector Machines

Unit 1

- hard margin classifier
- soft margin classifier
- formulation as optimization problems

Question

Hyperparameter C is used in:

a) hard margin classification

b) soft margin classification

c) both

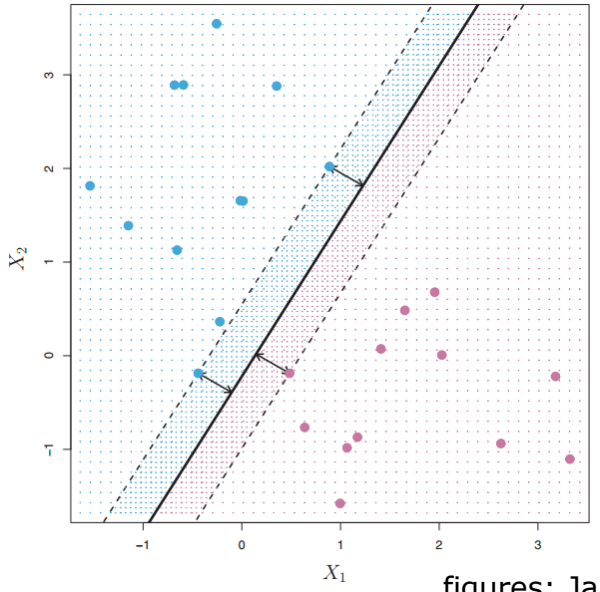
answer: b

Impact of hyperparameter C

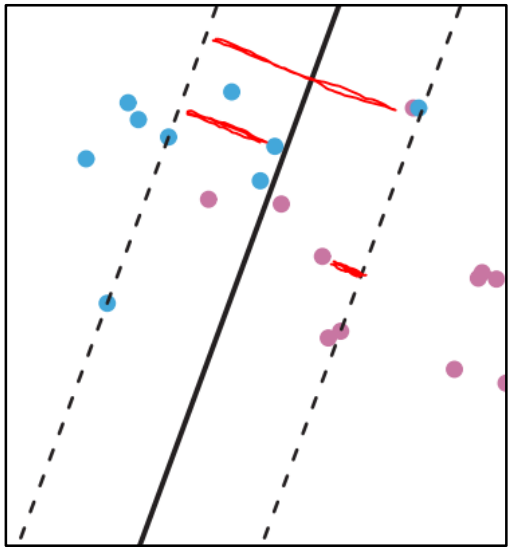
Hard margin classifier: make street as wide as possible while still correctly classifying all training data.

Soft margin classifier: compromise between making street wide and reducing total cost of margin violations.

Hyperparameter C: large value means margin violations have high cost



margin violations



figures: James et al, Intro. to Stat. Learning

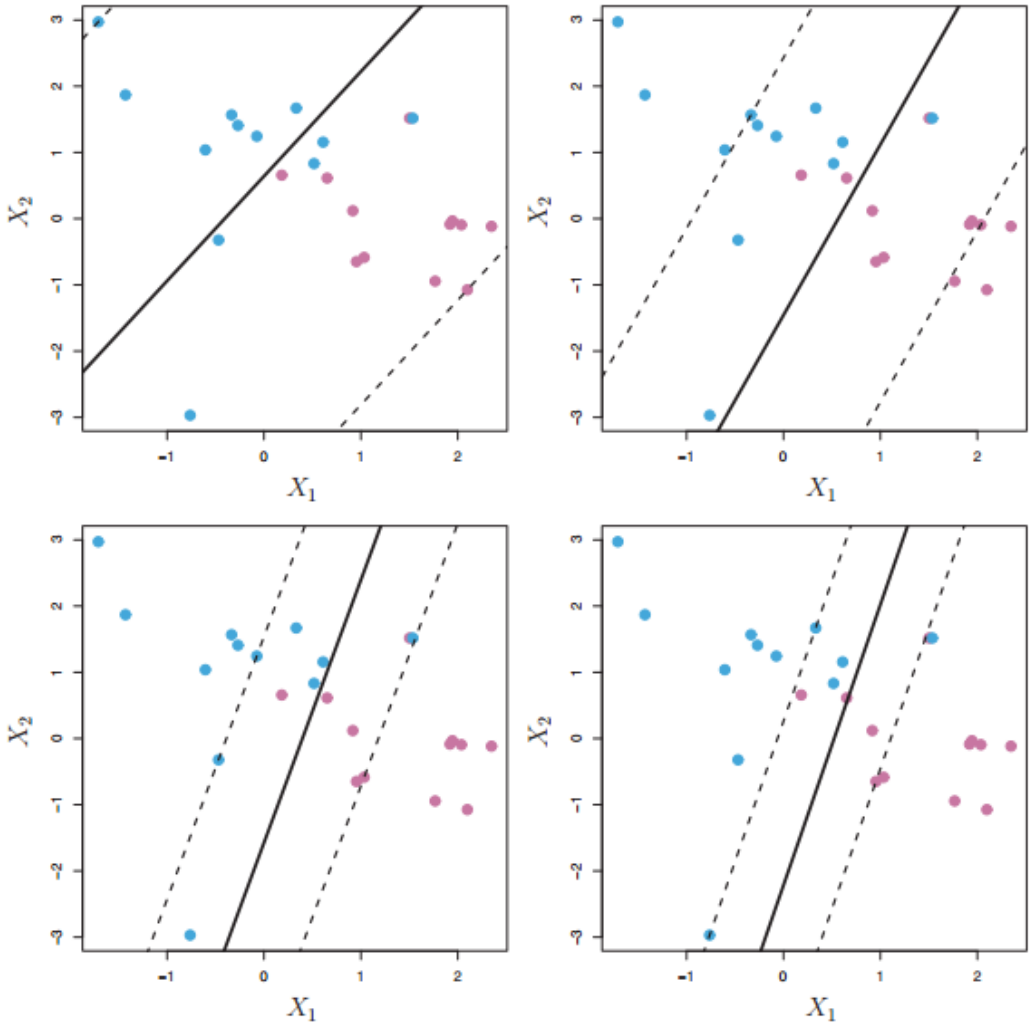
Impact of hyperparameter C

Hyperparameter C: large value means margin violations have high cost

Question: Which case on right indicates highest C value?

Answer: figure on bottom right

smaller C → wider street, more margin violations



source: James et all, intro to Stat. Learning

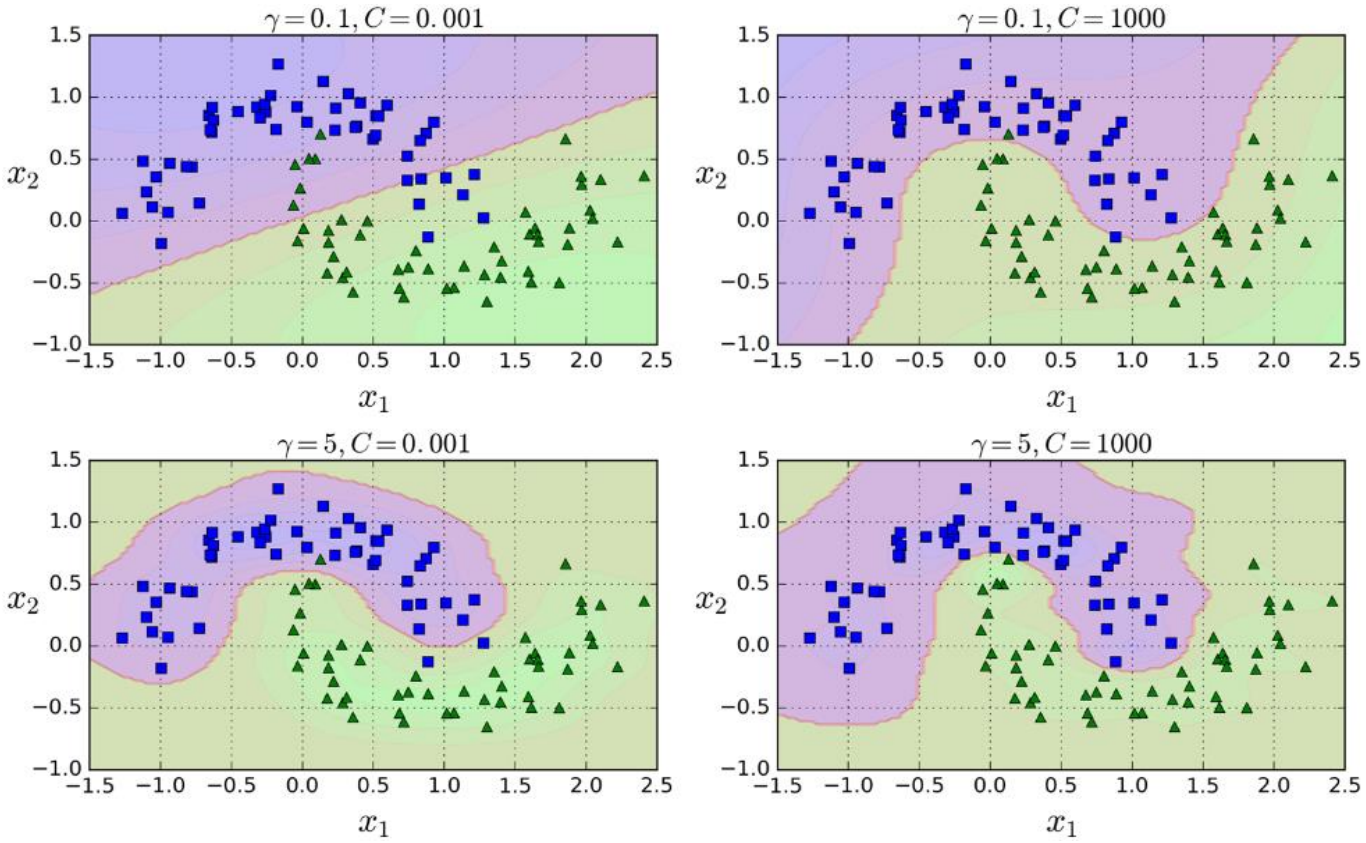
Unit 2

- polynomial features, similarity features
- adding the features manually
- polynomial and Gaussian RBF kernels
- regression with SVMs
- Scikit-Learn classes, and their performance

Similarity features with a kernel

```
rbf_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ))
rbf_kernel_svm_clf.fit(X, y)
```

support vector classifier with a "Gaussian RBF kernel"



γ: larger value makes bell-shaped curver narrower; reduces landmark's "range of influence"

source: Geron text

Ensemble methods

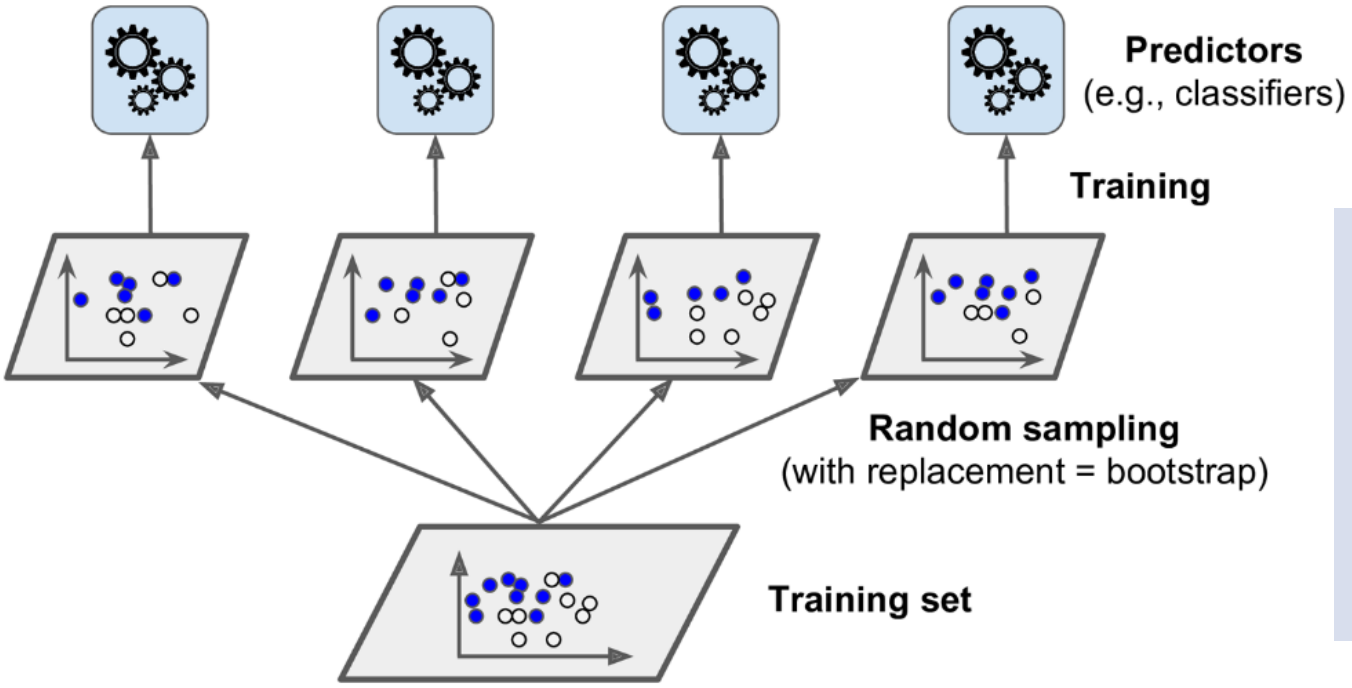
Unit 1: bagging

- hard and soft voting (predictor diversity)
- bagging and pasting (training data diversity)
- bagging and pasting in Scikit-Learn
- out-of-bag evaluation
- random subspaces, random patches

Bagging

In bagging, diversity of predictors is achieved by training them on different versions of the training data.

- for each predictor, train using m random samples from the training data set (with replacement)



Typically, a predictor will see about 63% of the training instances in its own training set.

Unit 2: AdaBoost

- random forests ≅ bagging + decision trees
- AdaBoost
  - train predictors sequentially
  - training instances are weighted, based on errors of previous predictor
- AdaBoost details: predictor error rates, predictor weights, instance weights, making predictions
- AdaBoost in Scikit-Learn

AdaBoost example

training data | initial weight values

x_1	x_2	y	weight	$\hat{y}$	
.2	234	0	0.2	0	✓
.5	43	0	0.2	1	
-.1	54	1	0.2	1	✓
.6	3	0	0.2	0	✓
.3	302	1	0.2	0	

error rate is 0.4

predictor weight is 0.405

updated weight values

predictor 1

x_1	x_2	y	weight	$\hat{y}$	
.2	234	0	0.167		
.5	43	0	0.25		
-.1	54	1	0.167		
.6	3	0	0.167		
.3	302	1	0.25		

predictor 2

Unit 3: gradient boosting

- Concept of gradient boosting:
  - sequentially train predictors
  - a predictor trains on residuals from previous predictor
  - ensemble predicts sum of the base predictions
- Gradient boosting in Scikit-Learn
- Classification with gradient boosting
- Stacking: train a 'blender'

Question

In using gradient boosting for regression, how are the predictions from the base estimators combined?

answer: just add them up;

predictions from second model should "correct" error in first model, predictions from second model should correct error in second model, etc.

A regression example; training

First predictor:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Second predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Third predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

A regression example; prediction

How the ensemble makes a prediction:

```
y_pred = sum(tree.predict(X_new) for tree in
                                (tree_reg1, tree_reg2, tree_reg3))
```

Stacking example

x	y
1	3.4
6	8.9
3	5.1
4	5.3
5	6.5
2	3.7

use half of training set to train predictors

trained predictors make predictions from other half

x	y	$\hat{y}_1$	$\hat{y}_2$	$\hat{y}_3$
4	5.3	5.1	5.3	5.0
5	6.5	5.9	6.7	6.2
2	3.7	3.1	4.2	3.9

combine the training data of previous step with the predictions

x_1	x_2	x_3	y
5.1	5.3	5.0	5.3
5.9	6.7	6.2	6.5
3.1	4.2	3.9	3.7

use this data to train blender

Dimensionality reduction

Unit 1: Principal Component Analysis

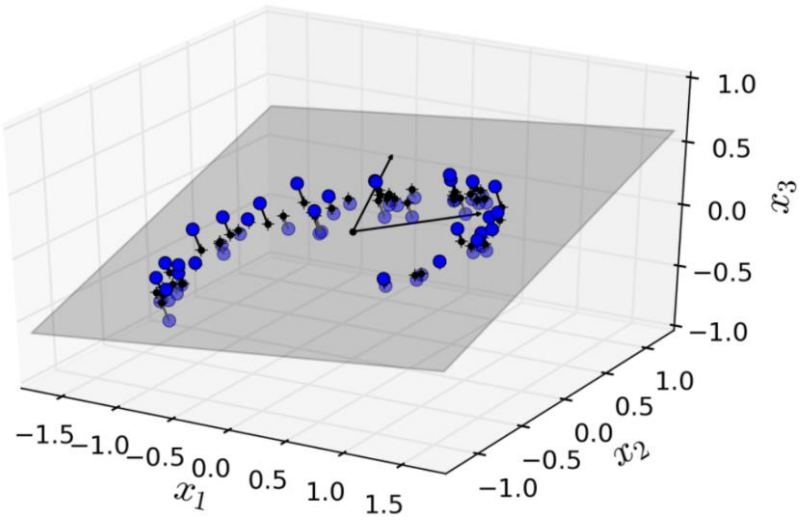
- High-dimensional data in machine learning
- Issues with high-dimensional data
- Feature selection and dimensionality reduction
- Projection and manifold learning
- PCA concept
- Computing the principal components
- PCA with Numpy and with Scikit-Learn
- Deciding on number of components to project to

Main approaches

- projection (PCA and its relatives)
  - assumption: training data is not spread out uniformly across all dimensions – training instances lie in a much lower-dimensional subspace
  - method: project data onto lower-dimensional subspace
- manifold learning (LLE, Isomap, MDS, t-SNE,…)
  - a 2D manifold is a 2D shape that can be bent and twised in a higher-dimensional space
  - assumption: most high-dimension datasets lie close to a much lower-dimension manifold
  - method: model the manifold on which the data lies

Principle component analysis: concept

When we are given 3-D data, the values are relative to a coordinate system, with x, y, z axes.



Think of this as expressing the data relative to three vectors, one for each axis:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We could equally well express our data relative to three other orthogonal axes.

Which other axes are best?

Question

Suppose X is a matrix of feature data, with m rows and n columns.

Suppose W_d is a matrix of the top d principal components, with the first column being the first principal component. How many rows does W_d have? (give a mathematical expression using one or more variables)

answer: n

Computing the principal components

One method is to use Singular Value Decomposition:

$$X = U \Sigma V^T$$

Recall that if X is an $m \times n$ matrix, then V^T is an $n \times n$ orthogonal matrix.

The columns $c_1, \ldots, c_n$ of V are the principal components:

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c_1} & \mathbf{c_2} & \cdots & \mathbf{c_n} \\ | & | & & | \end{pmatrix}$$

c_1 is the first principal component

this is a corrected version of Fig. 8-1 in the Géron text

Projecting down to d dimensions

$$X_d = X\, W_d$$

Here W_d is the first d columns of V. Note the sizes:

$$m \times n \qquad n \times d \qquad \rightarrow \qquad m \times d$$

Example: suppose our data consists of 3 instances, and we want to reduce from 4 to 2 features.

$$\underset{X}{\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \end{pmatrix}} \underset{W_d}{\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}} = \underset{X_\mathrm{d}}{\begin{pmatrix} ? & \end{pmatrix}}$$

an original feature vector

first principal component

Question

Suppose we use PCA, and use all of the principal components. In other words, in the matrix W_d, $d = n$.

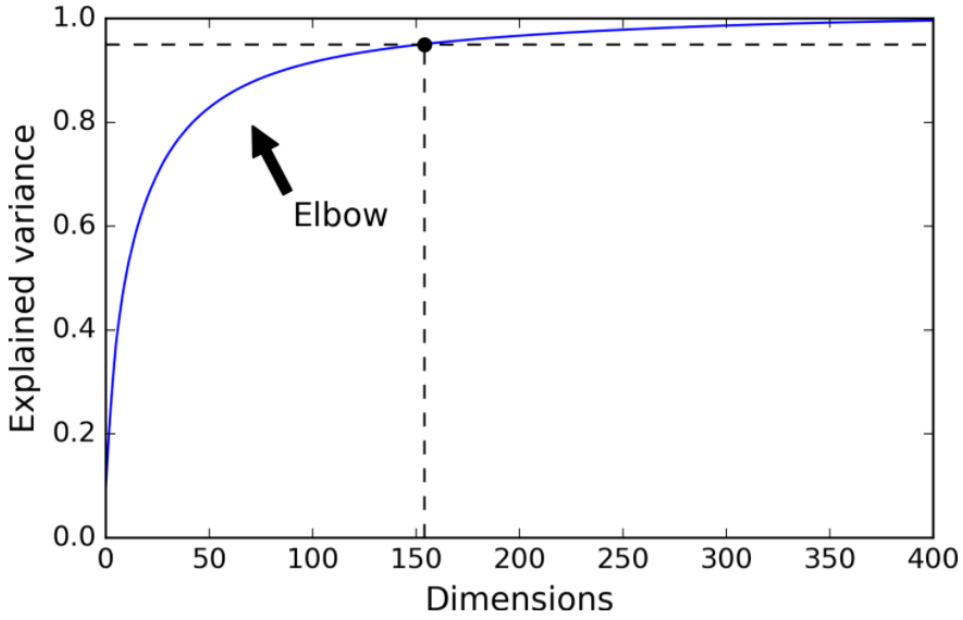
$$X_d = X\,W_d$$

If matrix X of feature data is an m by n matrix, what is the size of X_d?

answer: m by n

How many dimensions to project to?

```
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum > 0.95) + 1

# the easier way
pca = PCA(n_components = 0.95)
X_reduced = pca.fit_transform
```



Computational complexity

PCA: $O(m \times n^2) + O(n^3)$

Randomized PCA: $O(m \times d^2) + O(d^3)$

Recall:

m number of instances

n number of features

d # of dimensions in lower-dimensional space

Neural networks

Unit 1: introduction to TensorFlow

- computation model
- handling sessions
- evaluating a node
- linear regression example

Question

Where is a variable in the TensorFlow graph initialized?

a) in the graph construction phase

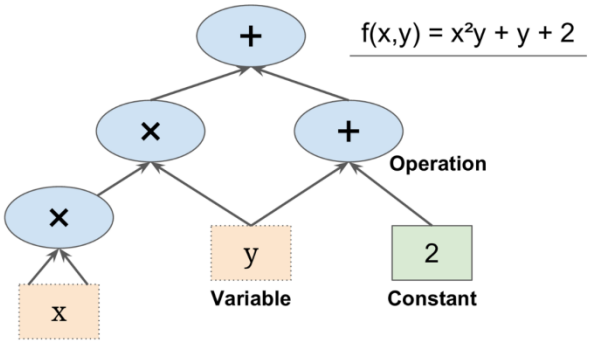
b) in the execution phase

answer: b

A simple TensorFlow program

Construction phase: build a computation graph

```
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```



Execution phase: initialize and evaluate variables

```
sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)
print(result)
```

evaluation is done within a TensorFlow session

output:

```
42
```

Alternatives for session handling

1

```
sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)
print(result)
```

a pain to write sess.run a lot

2

```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
```

sess is default session within block; session is automatically closed at end

3

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
    init.run()
    result = f.eval()
```

initialize all variables at once; first statement is just a node

Unit 2: linear regression with TF

- linear regression as an optimization problem
- TF code for linear regression
  - computing gradients
  - selecting an optimizer
  - mini-batch

Unit 3: Tensorboard

- extend TF code to produce data for Tensorboard
- running the Tensorboard tool
- saving and restoring models
- use language features of TensorFlow to modularize network:
  - name scopes
  - modular graph construction

Name scopes

Reduce clutter in TensorBoard view of graph by grouping related nodes.

```
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42),
name="theta")
y_pred = tf.matmul(X, theta, name="predictions")

# put error and mse within a single name score
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")

optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

This code creates a "loss" name scope that simplifies the graph

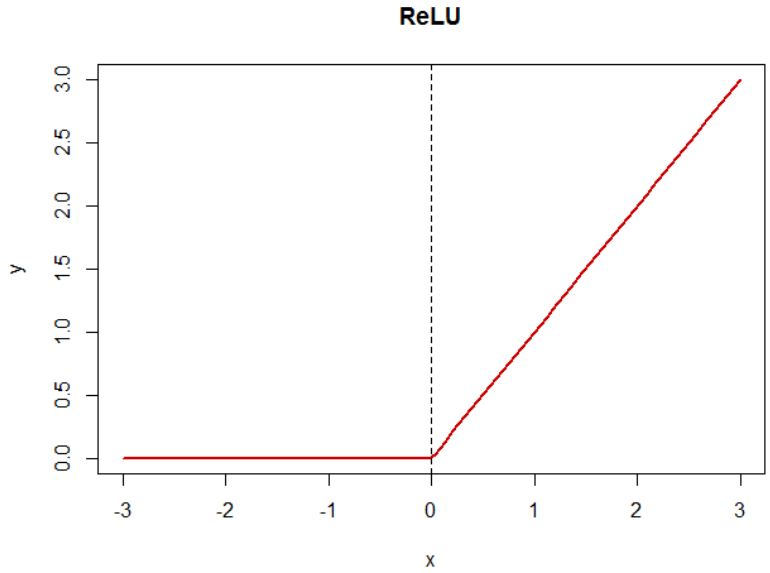
Unit 4: intro to neural nets

- artificial neuron
- perceptron
- linear threshold unit
- multi-layer perceptron
- activation function
- perceptron learning rule

Question

Write the definition of the ReLU activation function.

hint: it looks like this



answer: ReLu(x) = max(0, x)

Activation functions

Key to backprop:

- instead of step function, use logistic function:

$$\boldsymbol{\sigma(z)} = \frac{1}{1+\exp(-z)}$$

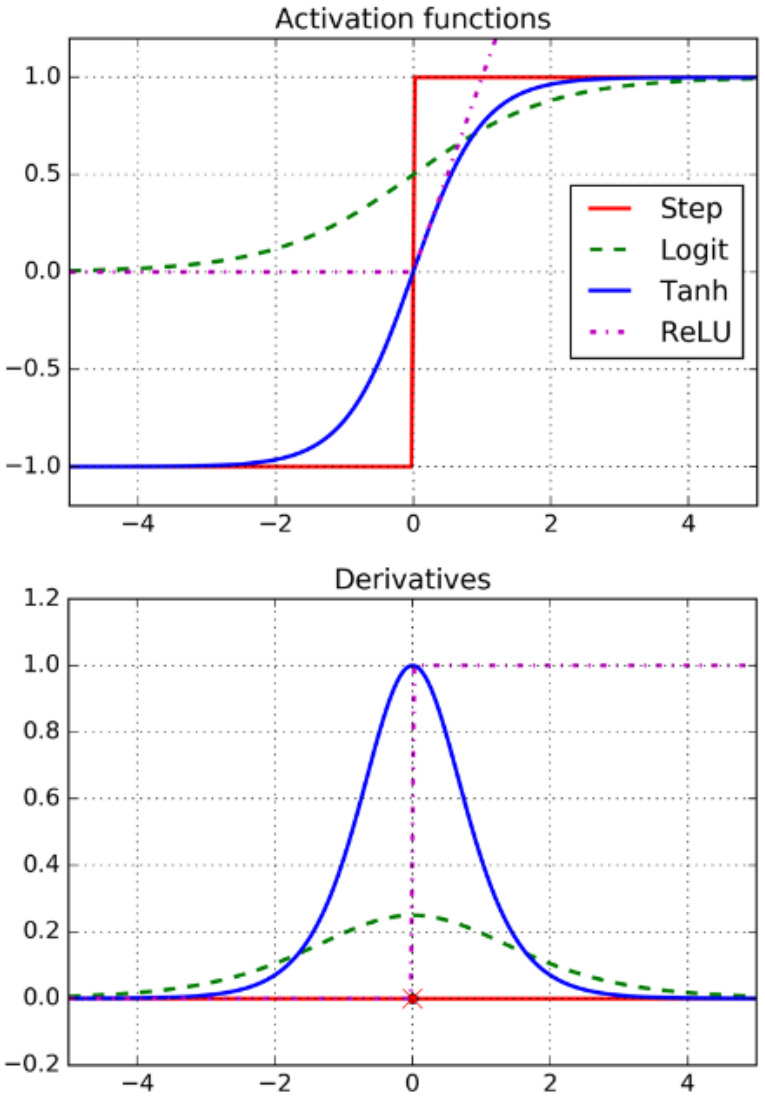
This is called an activation function. Other popular activation functions:

- hyperbolic tangent function:

$$\mathbf{tanh}(\boldsymbol{z}) = 2\sigma(2z) - 1$$

- ReLU function:

$$\boldsymbol{ReLU(z)} = \max(0, z)$$



Unit 5: backpropagation

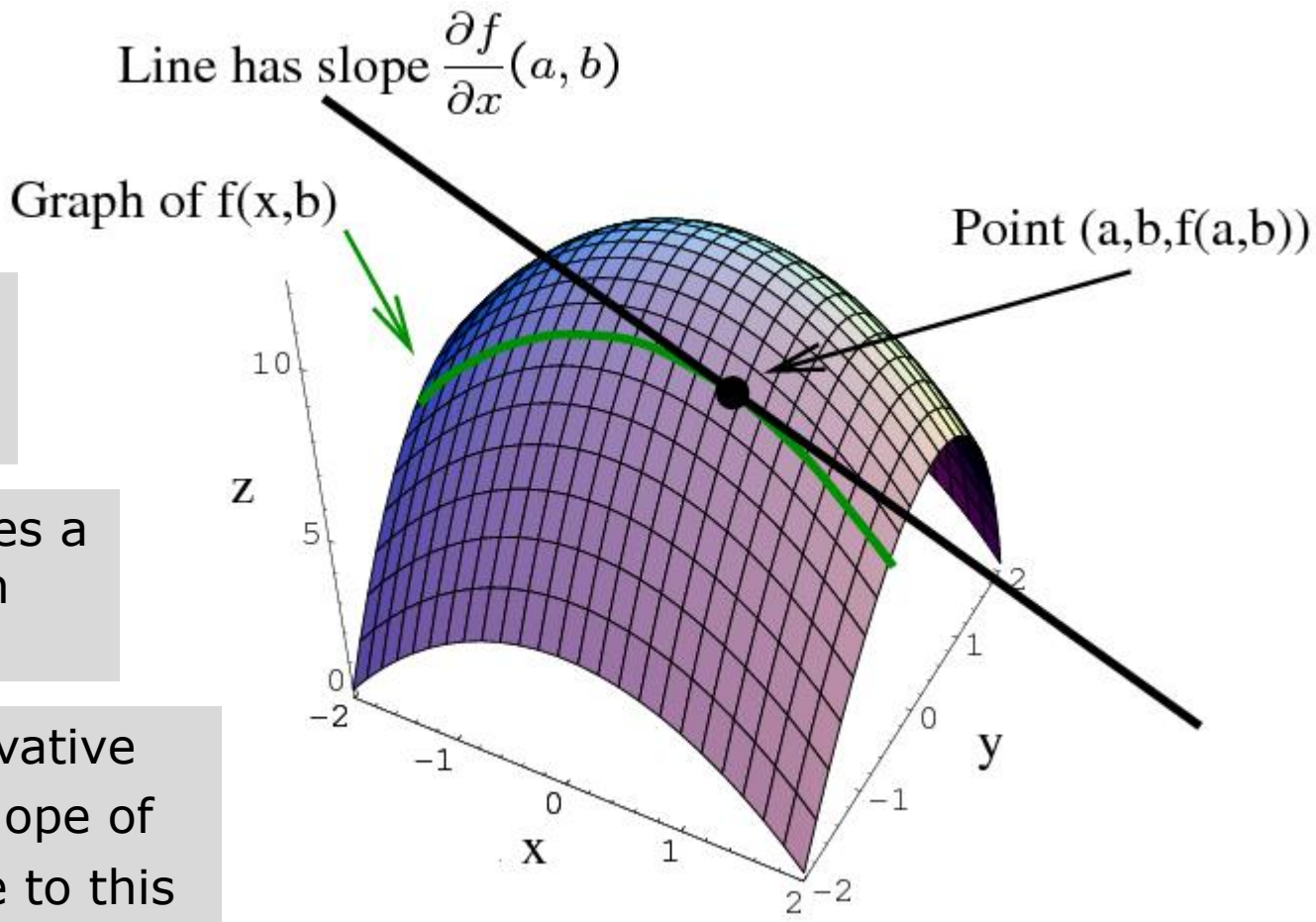
- gradient descent
- gradient descent in a computation graph
- backpropagation algorithm

Recall: partial derivatives

The graph of $f(x, y)$ is a surface.

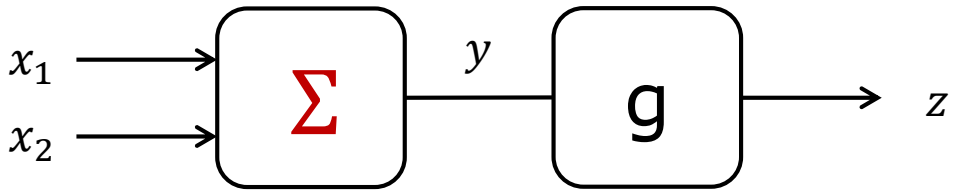
Fixing $y = b$ gives a curve (shown in green

The partial derivative $\frac{\delta f}{\delta x}(a, b)$ is the slope of the tangent line to this curve at the point where $x = a$.



source: https://mathinsight.org/partial_derivative_limit_definition

A weighted sum node



$z = y^2$

$y = w_1\, x_1 + w_2\, x_2$

Node Σ outputs the weighted sum of its inputs. The coefficient values are $w_1 = 0.5$ and $w_2 = 2.0$.

How to modify the coefficients of Σ to make z smaller? (Now treat x_1 and x_2 as constants.)

Use the multi-variable chain rule: $\frac{\partial z}{\partial w_1} = \frac{\partial z}{\partial y}\,\frac{\partial y}{\partial w_1}$ (similarly for w_2)

Example: Let $x_1 = 1$ and $x_2 = 3$ (so $y = 0.5 + 6 = 6.5$).

1. $\frac{\partial z}{\partial y}$ is $2y$, and $\frac{\delta y}{\delta w_1}$ is x_1 (note: $\frac{\delta y}{\delta w_1}$ does not depend on w_1 !)
2. By chain rule: $\frac{\partial z}{\partial w_1}$ at w_1, w_2 is $\frac{dz}{dy}(6.5) * \frac{dy}{dw_1}(0.5,2.0) = 13 * 1 = 13$
3. For new value of w_1, use $w_1 - \eta\, \frac{\delta z}{\delta w_1}(0.5, 2.0)$

Unit 6: training neural nets

- training with TF.Learn
- steps of building and training a feedforward neural net with pure TensorFlow

Evaluating accuracy (MNIST, CNN)

```
fc1 = tf.layers.dense(pool3_flat, n_fc1, activation=tf.nn.relu)

logits = tf.layers.dense(fc1, n_outputs)
Y_proba = tf.nn.softmax(logits)

xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=y)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer()
training_op = optimizer.minimize(loss)

correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Question: Is accuracy evaluated during training of the network? No

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
```

TensorFlow evaluation

```
fc1 = tf.layers.dense(pool3_flat, n_fc1, activation=tf.nn.relu)

logits = tf.layers.dense(fc1, n_outputs)
Y_proba = tf.nn.softmax(logits)

xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=y)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer()
training_op = optimizer.minimize(loss)

correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Question: Is the feed dict below okay?

Yes, y isn't needed for Y_proba

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        probs = Y_proba.eval(feed_dict={X: X_batch})
```

Summary of building the net

1. define inputs
2. specify placeholders for training, target data
3. create the network
4. define a loss function
5. specify an optimizer
6. specify a performance measure
7. make initializer and saver

Questions:

- does the optimizer depend on the loss function?
- does the performance measure depend on the optimizer?

Tuning the network parameters

- number of hidden layers
  - with more layers, exponentially fewer total neurons can be used to model complex functions
  - hierarchical concept
  - start with a couple, ramp up
- number of neurons per hidden layers
  - fewer and fewer neurons per layer, as you move to output layer
- activation functions
  - often, ReLU for hidden layers (good performance)
  - output layer: often softmax for classification, nothing for regression

please read this section of our text carefully for details

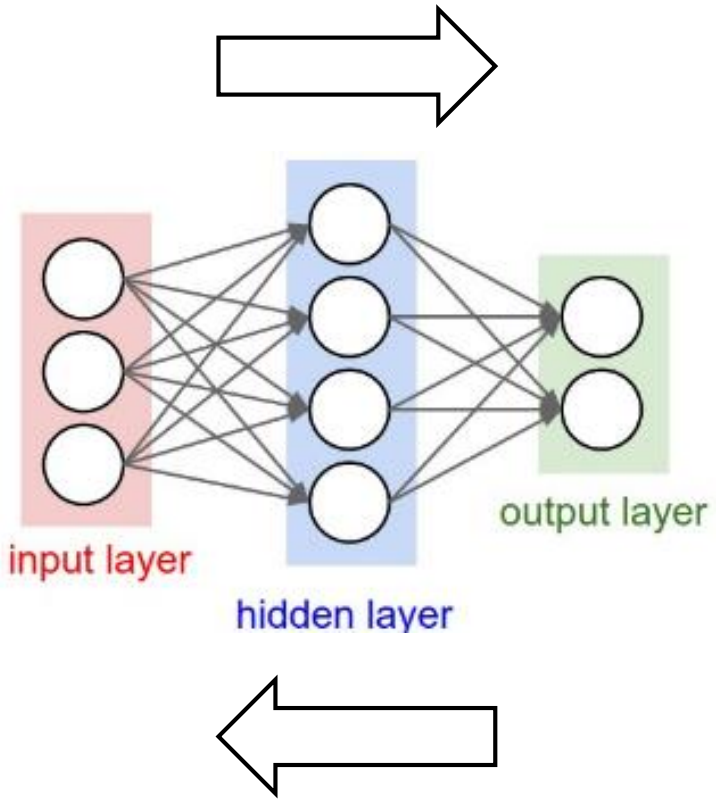
Unit 7: vanishing gradients

- problems in training deep neural nets
  - vanishing/exploding gradients
  - training time
  - overfitting
- cause of vanishing/exploding gradients
- Xavier, He initialization
- Leaky ReLU, ELU activation functions
- batch normalization
- gradient clipping

Vanishing/Exploding gradients (cont'd.)

Vanishing gradients: gradients get smaller and smaller during backward pass

Exploding gradients: gradients get larger and larger during backward pass

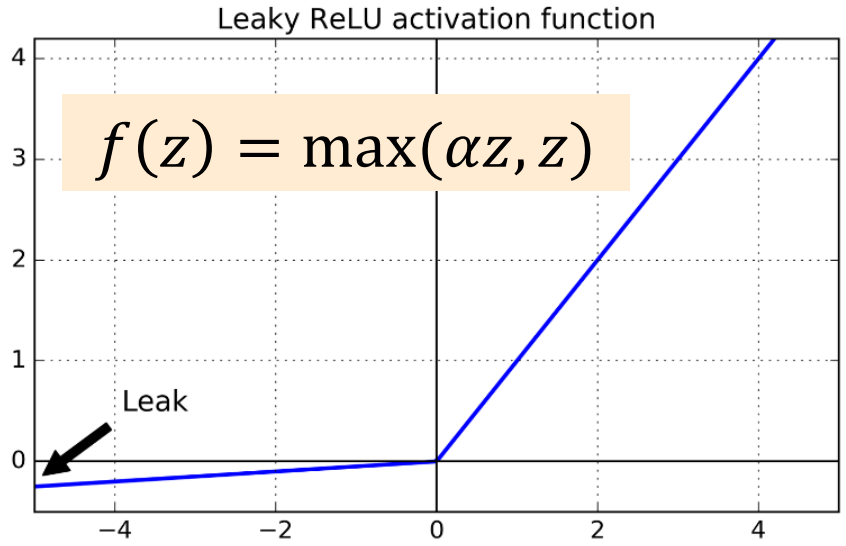


Dump the sigmoid (cont'd.)

Replacement candidate 2: leaky ReLU

pros:

- unlike a ReLU, it can never "die"
- a recent paper claims it always outperforms plain ReLU



Other cousin candidates:

- randomized leaky ReLU
  - α is picked randomly during training
- parametric leaky ReLU
  - α is learned during training

α is a hyperparameter; typically set to 0.01

values of up to 0.2 may work even better

Unit 8: optimizers

Address training time in deep learning

Tricks to speed training:

- reusing layers, freezing layers, caching layers
- strategies for tweaking reuse
- model zoos
- unsupervised pretraining
- pretraining on an auxiliary task

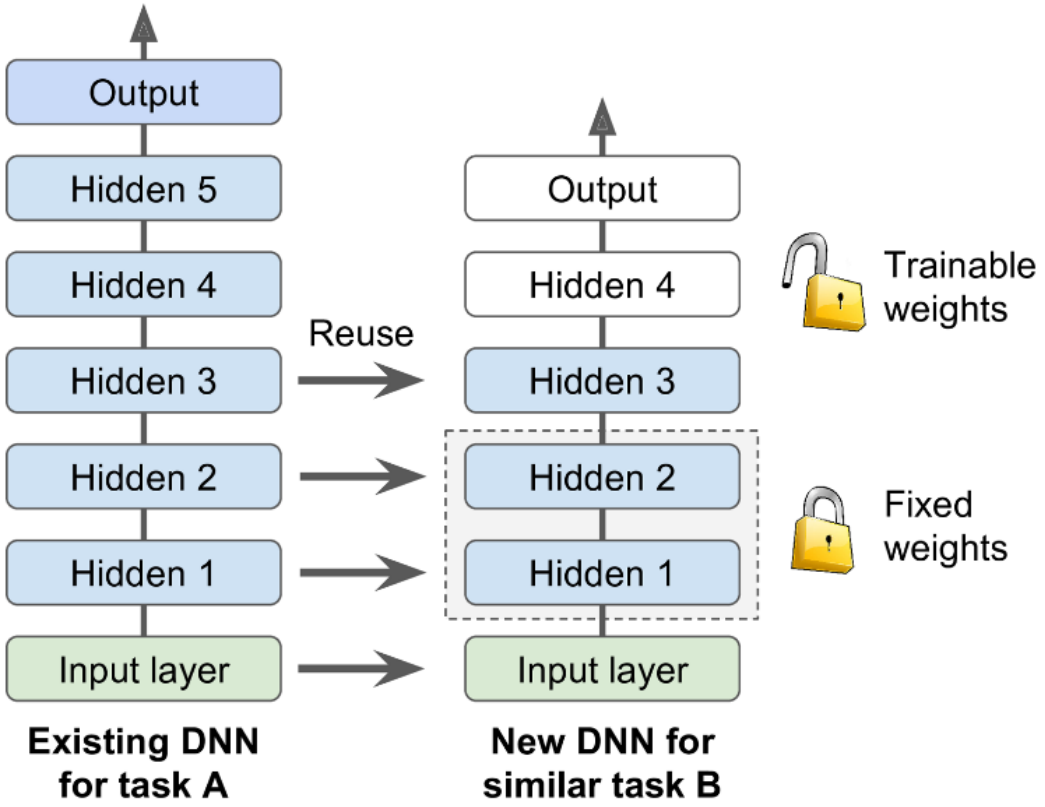
Improved optimization algorithms

- Momentum, Nesterov Accelerated Gradient
- AdaGrad, RMSProp
- Adam Optimization

Reusing layers

- Training large nets takes a long time
- Many nets solve similar tasks

Transfer learning: reuse lower levels of a net solving a problem similar to your own



Example: you want to classify vehicles from images; existing nets classify images of animals, vehicles, etc.

Freezing layers

If you reuse weights from another DNN, you probably want to "freeze" them.

This means their values won't be involved in the training process.

An easy way to do this in TensorFlow: tell optimizer which variables to train:

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")

# variables in hidden layers won't change during training
training_op = optimizer.minimize(loss, var_list=train_vars)
```

(see our text for an alternative method)

Pretraining on an auxiliary task

- Similar to the idea of borrowing layers from another DNN
- Idea 1:
  - you want to build a system to classify faces, but have little labeled training data
  - so grab a bunch of unlabeled face images, and training then to distinguish whether two pictures are of the same person
  - then reuse lower levels of this DNN for your problem
- Idea 2:
  - you are building a language processing DNN
  - grab a bunch of English sentences, label them 'good'
  - corrupt a bunch of English sentences, label them 'bad'
  - train a classifier to predict which sentences are good or bad

Momentum optimization

Idea: gain momentum on downhill slopes and roll over local minima

$$\mathbf{m} = \beta\mathbf{m} + \eta\,\nabla_\theta J(\theta)$$

$$\theta = \theta - \mathbf{m}$$

where $\mathbf{m}$ is the momentum vector.

$0 \le \beta \le 1$ 0 means "high friction", 1 means "no friction"

Suppose gradient is constant, β = 0.9 (typical value)

Example:
$\eta = 0.1$
$\nabla_\theta J(\theta) = (1,1)$
$\mathbf{m} = (0,0)$

$\theta = (0.5, 2)$
$\mathbf{m} = 0 + (0.1, 0.1)$
$\theta = (0.4, 1.9)$
$\mathbf{m} = ?$
$\theta = ?$

$\theta = (0.5, 2)$
$\mathbf{m} = 0 + (0.1, 0.1)$
$\theta = (0.4, 1.9)$
$\mathbf{m} = (0.09 + 0.09) + (0.1, 0.1)$
$= (0.19, 0.19)$
$\theta = (0.21, 1.71)$

Unit 9: regularization

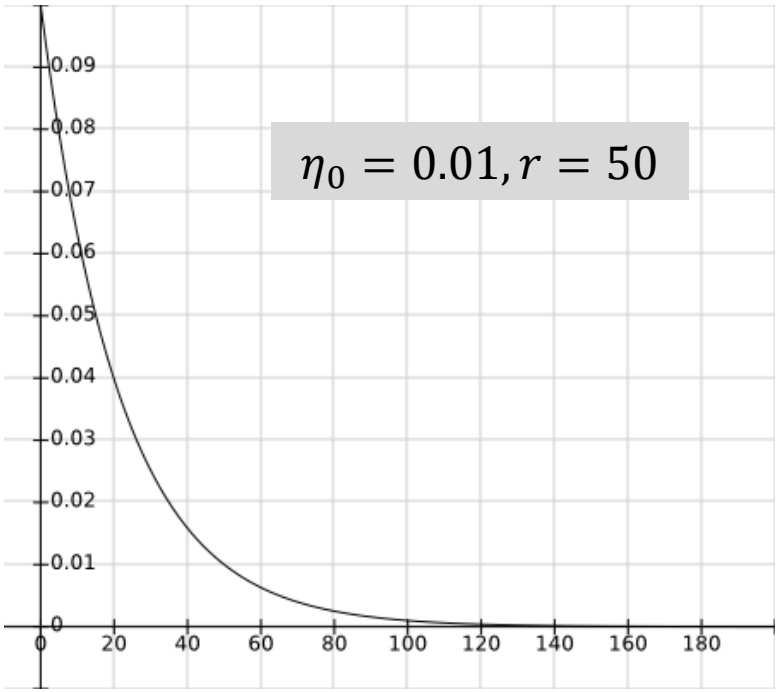
- address training time and overfitting
- learning rate scheduling
  - piecewise constant
  - performance
  - exponential, power
- regularization
  - early stopping
  - l1, l2 regularization
  - dropout
  - data augmentation

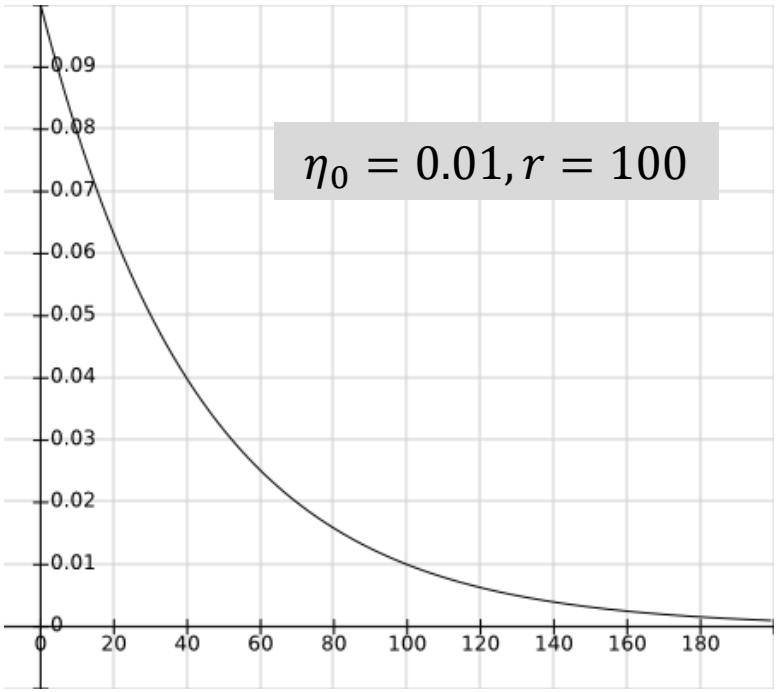
Exponential scheduling

- learning rate a function of iteration number:

$$\eta(t) = \eta_0 10^{-t/r}$$

- two tuning parameters: η_0, r





Dropout

Super effective.

Idea is simple:

- at each training step, each neuron (except output neurons) has some probability p of being ignored
- dropout rate p is typically set to 0.5
- dropout only occurs during training

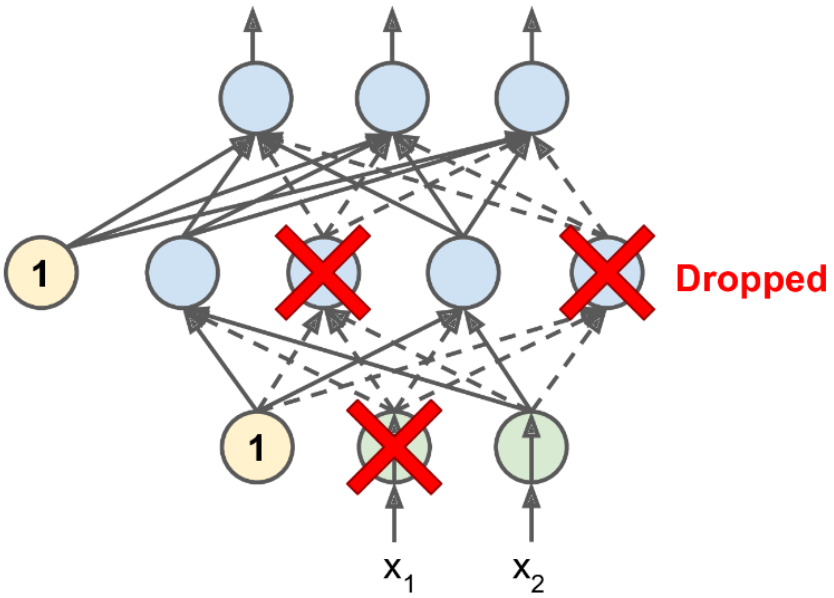


figure source: Géron

Data augmentation

Reduce overfitting by creating addition training examples.

Create the new example by modify existing examples.

For example, if training on images:

- shift every image in training set
- rotate
- resize
- change contrast
- flip image horizontally

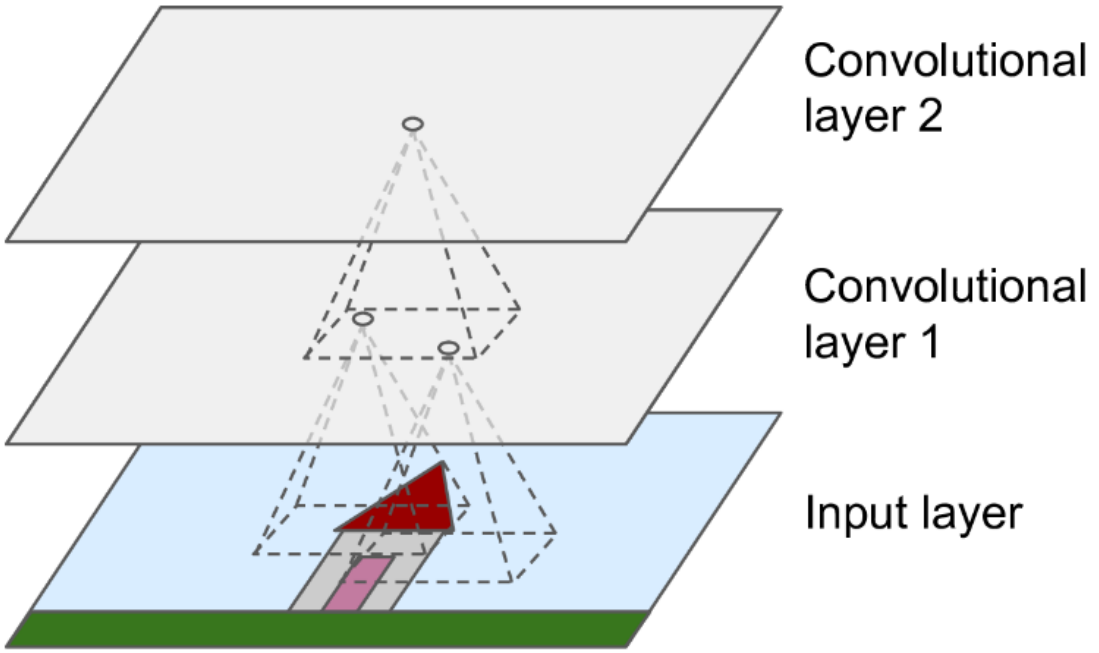
You can do this on the fly rather than creating a huge training set

Unit 10: intro to CNNs

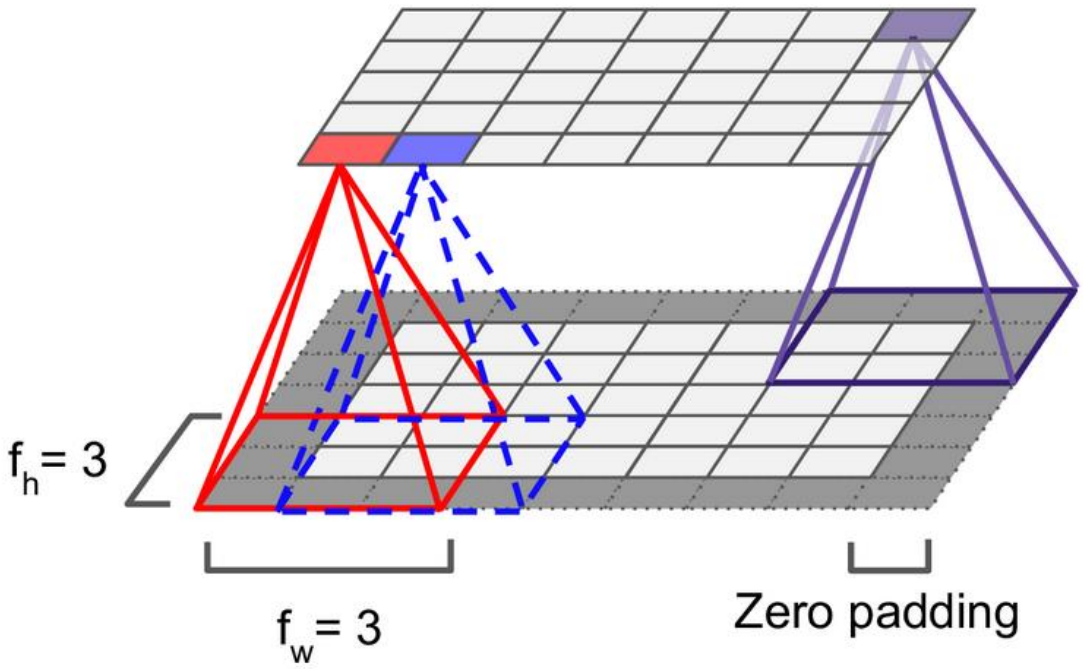
- biological motivation
- convolutional layer
  - padding
  - stride
  - filters
- feature map
  - stacking feature maps
- formula showing output of a CNN neuron

Convolutional layer

- in the first convolutional layer of a Convolutional Neural Network, each neuron "sees" only pixels in its "receptive field"
- in the second convolutional layer, each neuron is connected only to neurons in a small rectangle of the previous layer



Padding



3x3 filter vertical and horizontal stride is 1

Idea 1: use a second layer that is smaller than the first

Idea 2: add padding around the edge of the first layer

"zero padding"

Question

TensorFlow uses the names "valid" and "same" for the two padding options. What does "valid" mean?

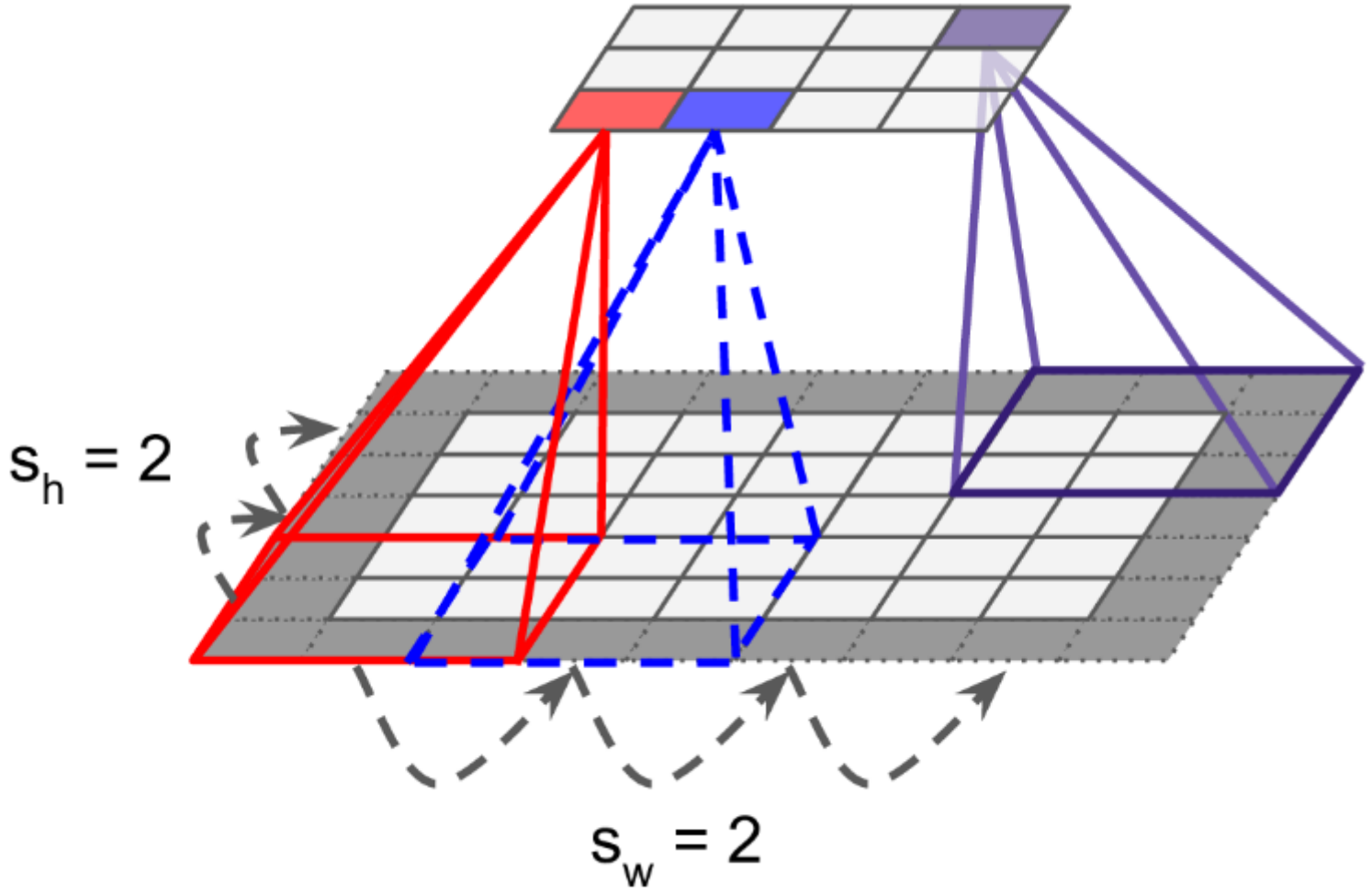
a) use zero padding

b) use no padding

answer: b

I think "same" is supposed to suggest that we use padding so dimensions aren't reduced.

Stride



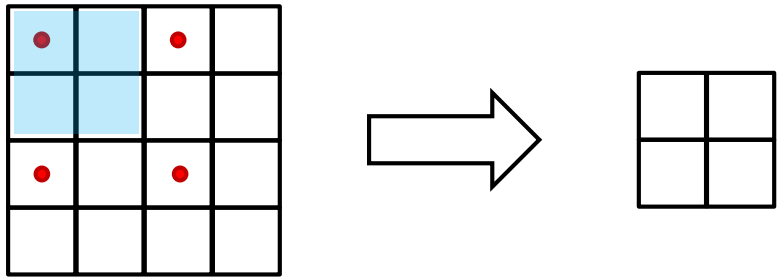
Question

Input is 4 x 4

Apply 2 x 2 filter, with "valid" padding and stride 2.

What is height and width of output?

hint:



answer: 2 x 2

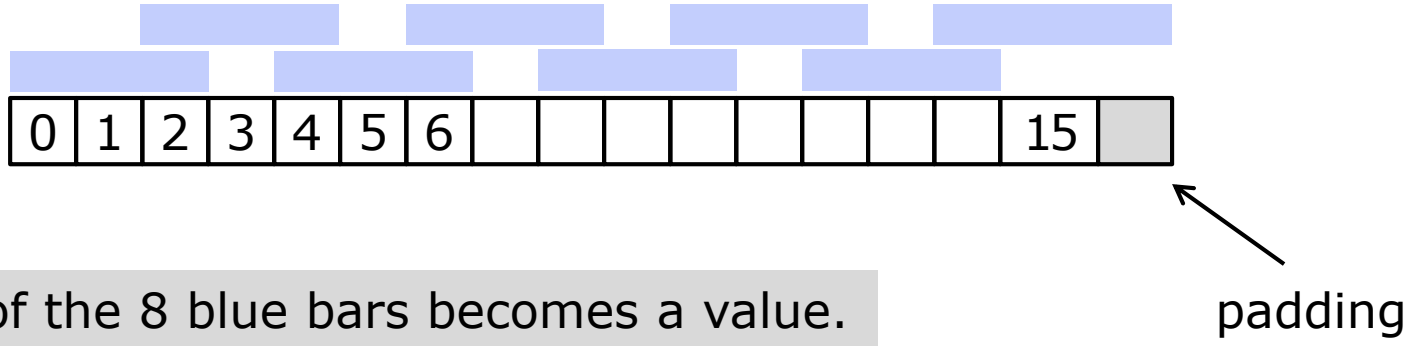
Question

Input is 16 x 16

Apply 3 x 3 filter, with "same" padding and stride 2.

What is height and width of output?

hint: everything is square, so we can think in one dimension

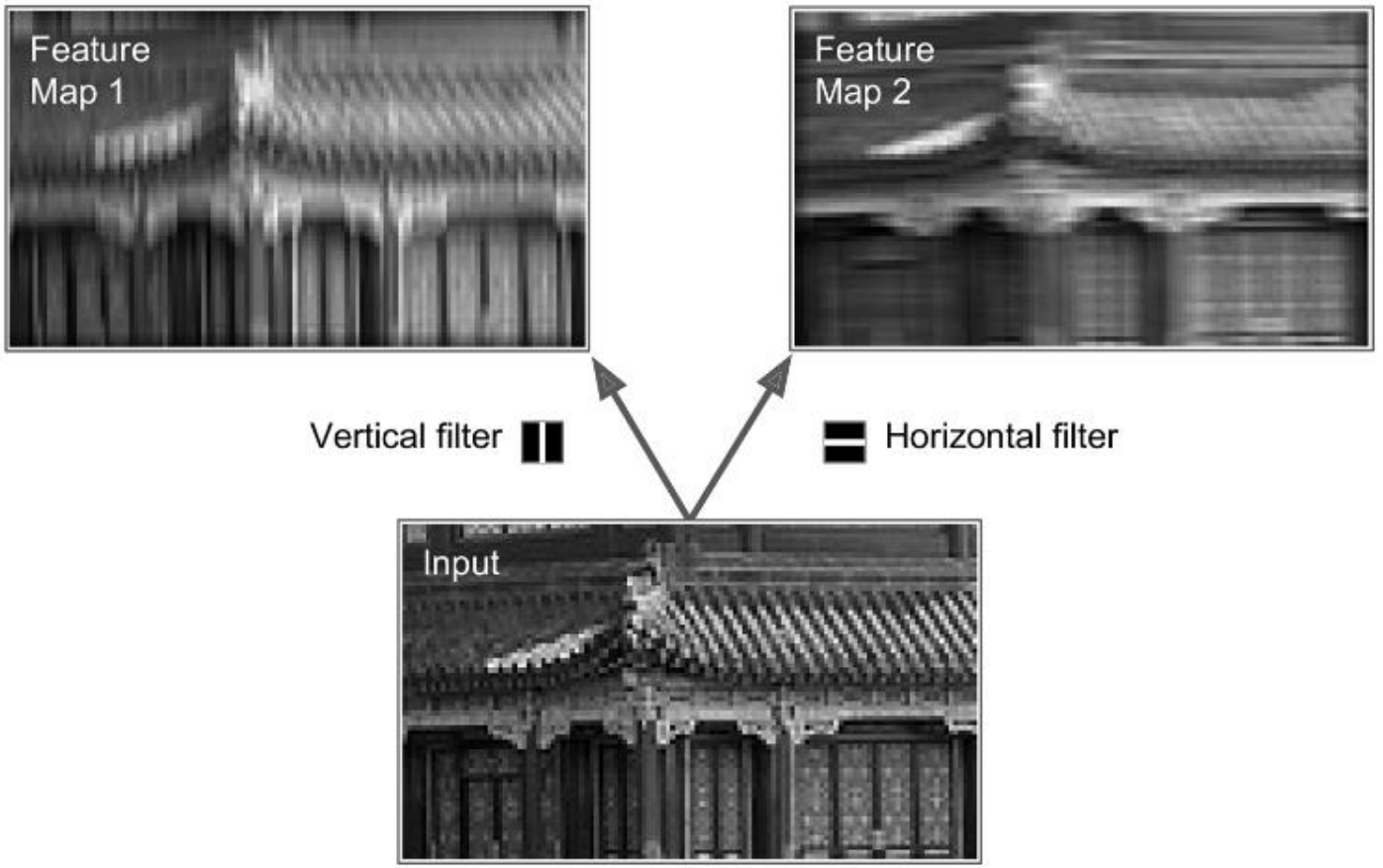


Each of the 8 blue bars becomes a value.

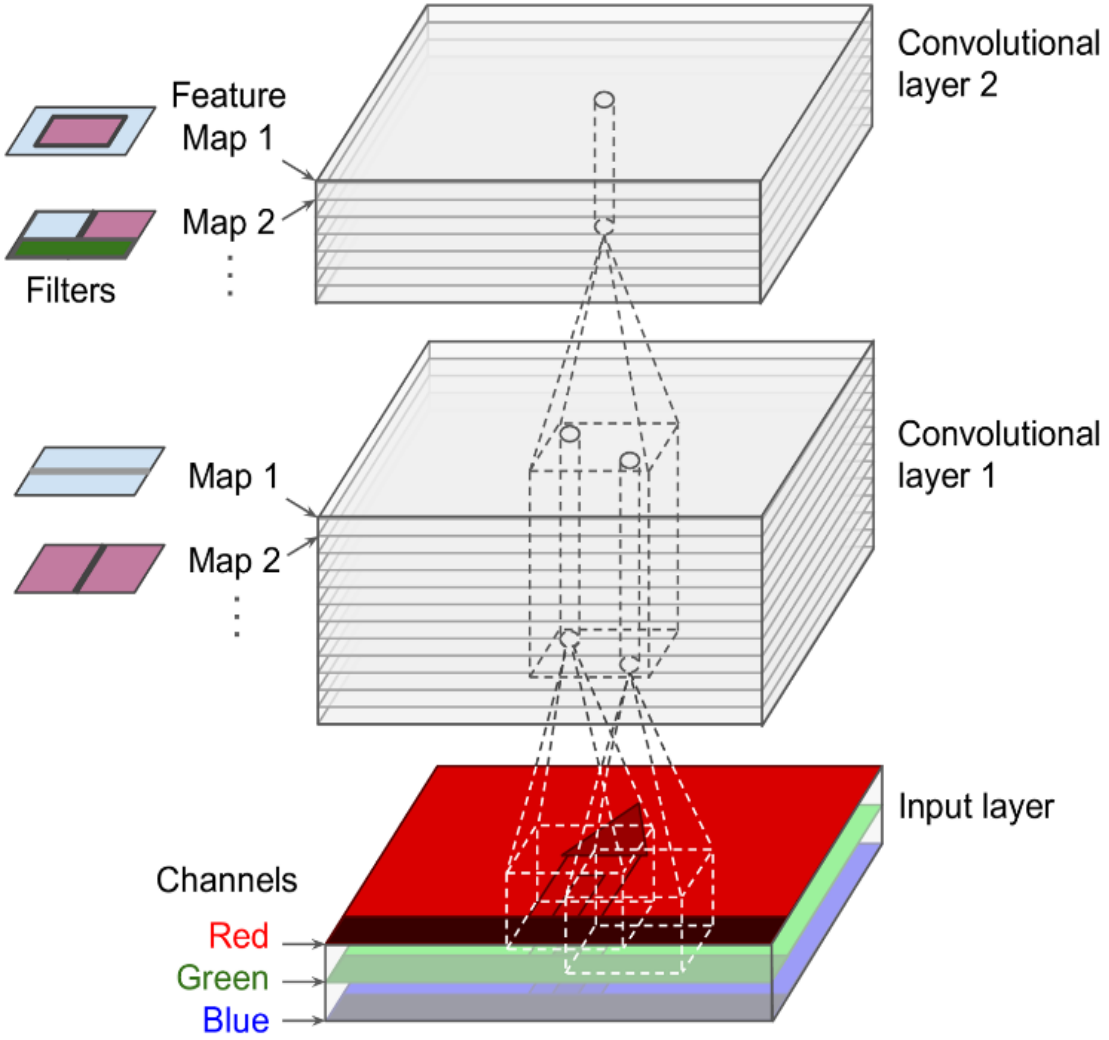
answer: 8 x 8

Feature maps

A layer full of neutrons using the same filter is a feature map.



Stacking feature maps



a convolutional layer applies multiple filters to its inputs

a convolutional layer can then detect multiple features anywhere in its inputs

Computing the output of CNN neuron

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} . w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

$z_{i,j,k}$ is the output of the neuron at row i, column j in feature map k of the convolutional layer

s_h, s_w are the vertical and horizontal strides

f_h, f_w are the height and width of the receptive field

$f_{n'}$ is the number of feature maps in the previous layer

$x_{i',j',k'}$ is the output of the nuron located in the previous layer at row i', column j', feature map k'

b_k is the bias term for feature map k of this layer

$w_{u,v,k',k}$ is the connection weight between a neuron in feature map k of this layer and its input at row u, column v, (relative to neuron's receptive field) and feature map k'

Unit 11: CNNs with TensorFlow

- build a convolutional layer with TF
- pooling layers
- CNN architecture
  - typical structure
  - examples of successful CNNs

Build a convolutional layer in TF

```
# Load sample images
china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")
dataset = np.array([china, flower], dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# 7x7 sized filters, 2 channels, and 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

# Create a graph with one convolutional layer applying the 2 filters
tf.reset_default_graph()
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1],
                           padding="SAME")

# execute
with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})
```

Pooling

Basically: aggregate over receptive field

As with convolution, you must specify:

- size of the rectangle
- stride
- padding type

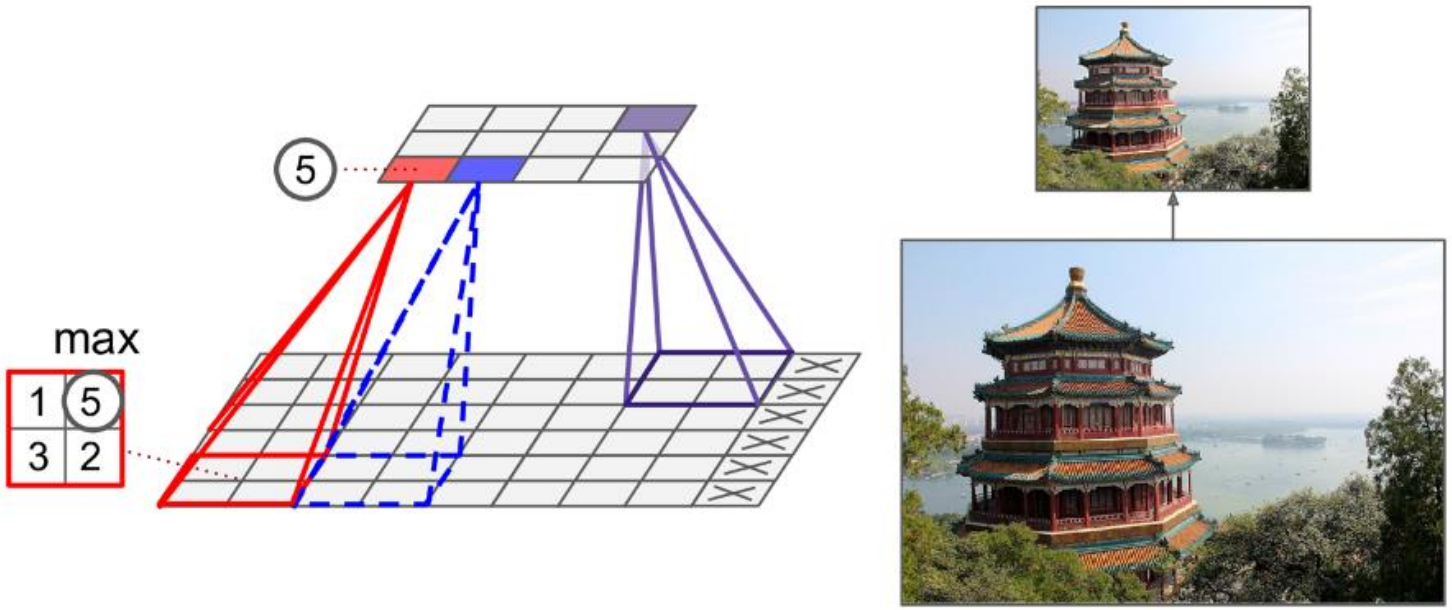
But now, instead of weights, also specify:

- aggregation function (max, mean, etc.)

Reduces computational load, memory usage, and number of parameters

Question: how is overfitting affected?

Max pooling example

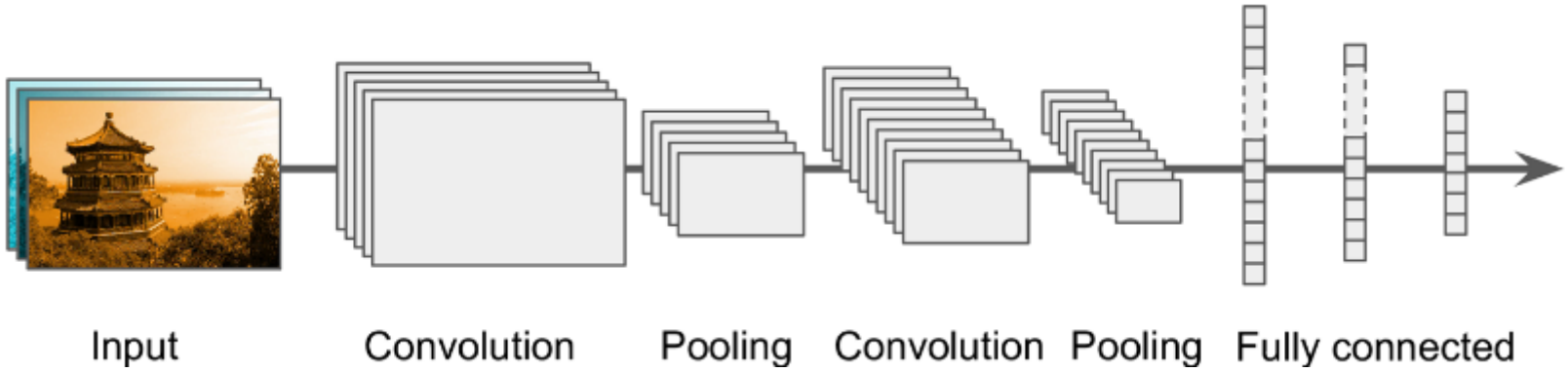


Notes:

- pooling is usually done on each channel independently
- but, pooling can also be done across channels

CNN architecture

How are convolution and pooling combined in common CNNs?



- The convolutional layers usually have ReLU activation
- Image gets smaller but also deeper (more feature maps) as it goes through network
- Final layer outputs prediction (e.g. softmax layer)

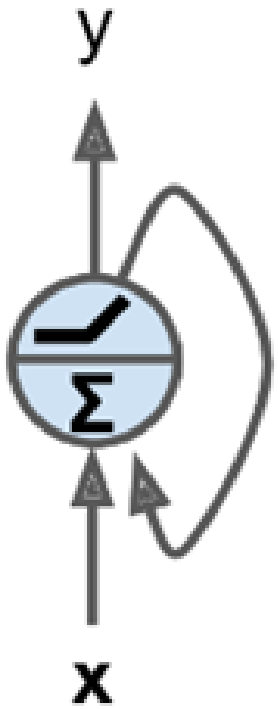
Unit 12: intro to RNNs

- recurrent neurons
- unrolling a RNN
- manually-created RNN

A recurrent neuron

Recurrent neural networks (RNNs) have connections that "point" backwards

At each time step, the recurrent neuron receives input x, as well as output from the previous step.



Output of a single recurrent neuron

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{W}_x^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_y^{T} \cdot \mathbf{y}_{(t-1)} + \mathbf{b}\right)$$

- similar to neurons in feedforward neural nets
- two sets of weights: one for inputs, and one for feedback (output of previous step)
- b is bias term

Unit 13: training RNNs

- letting TF do the unrolling
- variable-length input sequences
- using an RNN to build an MNIST classifier

Dynamic unrolling

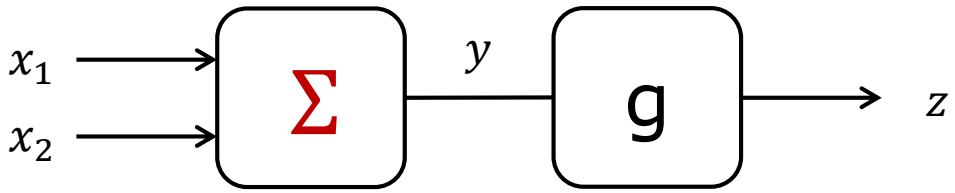
graph construction:

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X,
                                    dtype=tf.float32)

init = tf.global_variables_initializer()
```

execution:

```
X_batch = np.array([  # t=0         t=1
                     [[0, 1, 2], [9, 8, 7]], # instance 1
                     [[3, 4, 5], [0, 0, 0]], # instance 2
                     [[6, 7, 8], [6, 5, 4]], # instance 3
                     [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})
```

A weighted sum node



$z = y^2$

$y = w_1\, x_1 + w_2\, x_2$

Node Σ outputs the weighted sum of its inputs. The coefficient values are $w_1 = 0.5$ and $w_2 = 2.0$.

How to modify the coefficients of Σ to make z smaller? (Now treat x_1 and x_2 as constants.)

Use the multi-variable chain rule: $\frac{\partial z}{\partial w_1} = \frac{\partial z}{\partial y}\,\frac{\partial y}{\partial w_1}$ (similarly for w_2)

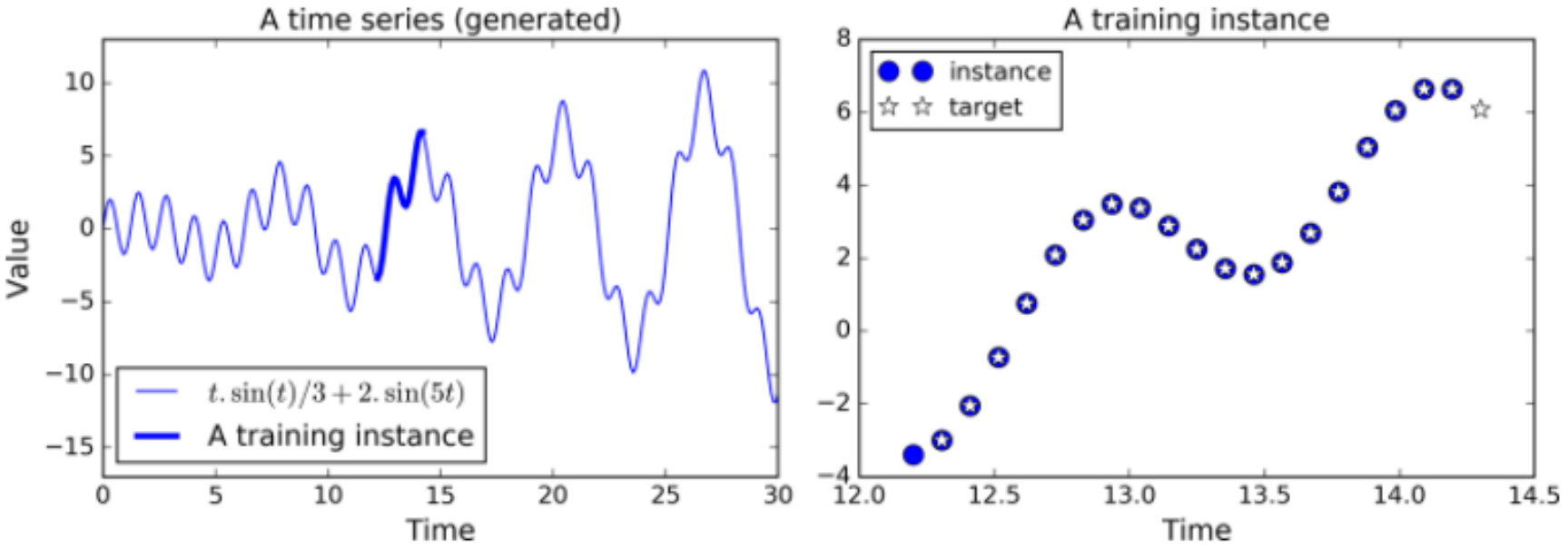
Example: Let $x_1 = 1$ and $x_2 = 3$ (so $y = 0.5 + 6 = 6.5$).

1. $\frac{\partial z}{\partial y}$ is $2y$, and $\frac{\delta y}{\delta w_1}$ is x_1 (note: $\frac{\delta y}{\delta w_1}$ does not depend on w_1 !)
2. By chain rule: $\frac{\partial z}{\partial w_1}$ at w_1, w_2 is $\frac{dz}{dy}(6.5) * \frac{dy}{dw_1}(0.5,2.0) = 13 * 1 = 13$
3. For new value of w_1, use $w_1 - \eta\, \frac{\delta z}{\delta w_1}(0.5, 2.0)$

Unit 14: forecasting with RNNs

- RNN review
- problem set up
  - training instances are sequences
  - targets are 1-shifted training instances
- output projections
  - to get outputs of the right shape

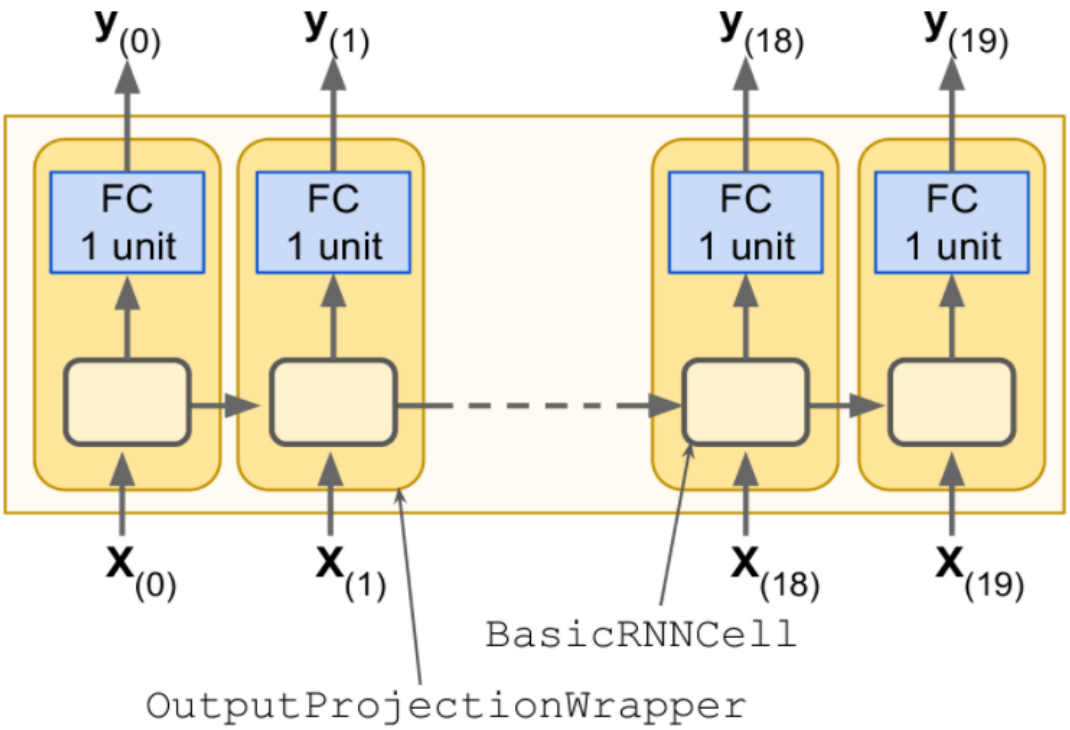
Time series forecasting with RNNs



- training instance: 20 values from the series
- each input has only one feature
- target: 20 values, but shifted over by one step
- 100 recurrent neurons

Output projections

```
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu),
    output_size=n_outputs)
```



The wrapper adds a fully-connected (FC) layer of linear neurons on top of each output.

('linear' means no activiation function.)

```
In [11]: outputs.get_shape()
Out[11]: TensorShape([Dimension(None), Dimension(20), Dimension(1)])
```

Time Series Analysis

Unit 1: time series intro

- examples
- definition of time series
- how to create time series data sets

What is time series data?

Time series: measurements of a variable at regular intervals over some period of time

The above definition is for univariate (single variable) time series.

We can also have multivariate time series, where we have measurements of multiple variables at each time point.

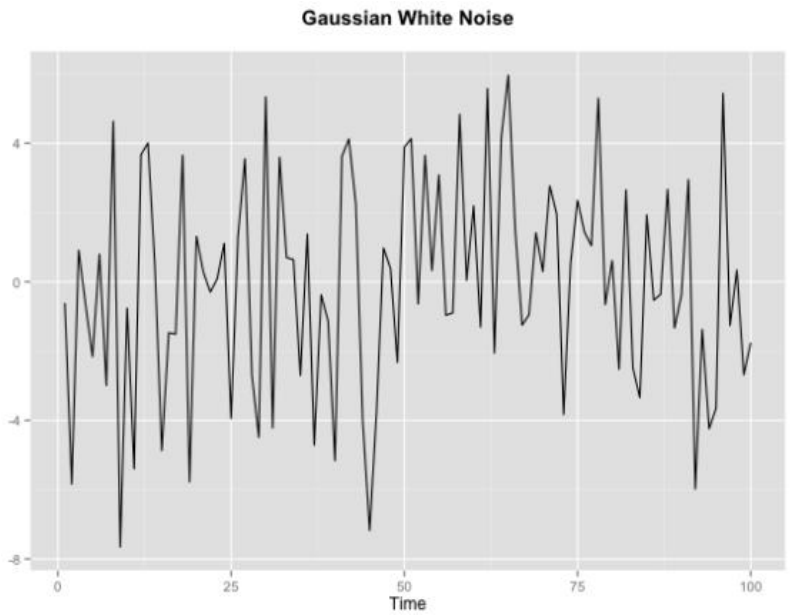
Unit 2: basic concepts

- white noise
- random walk
- autocorrelation function (ACF)
- stationary, weakly stationary time series

White noise

White noise is a time series $x_1, x_2, x_3, \ldots$ in which the random variables are independent and identically distributed (iid), with mean 0.

The figure shows Gaussian white noise, where each variable is normally distributed.



In reality, the variables in a time series are rarely independent.

figure: Forte, Mastering Predictive Analytics in R

Random walk

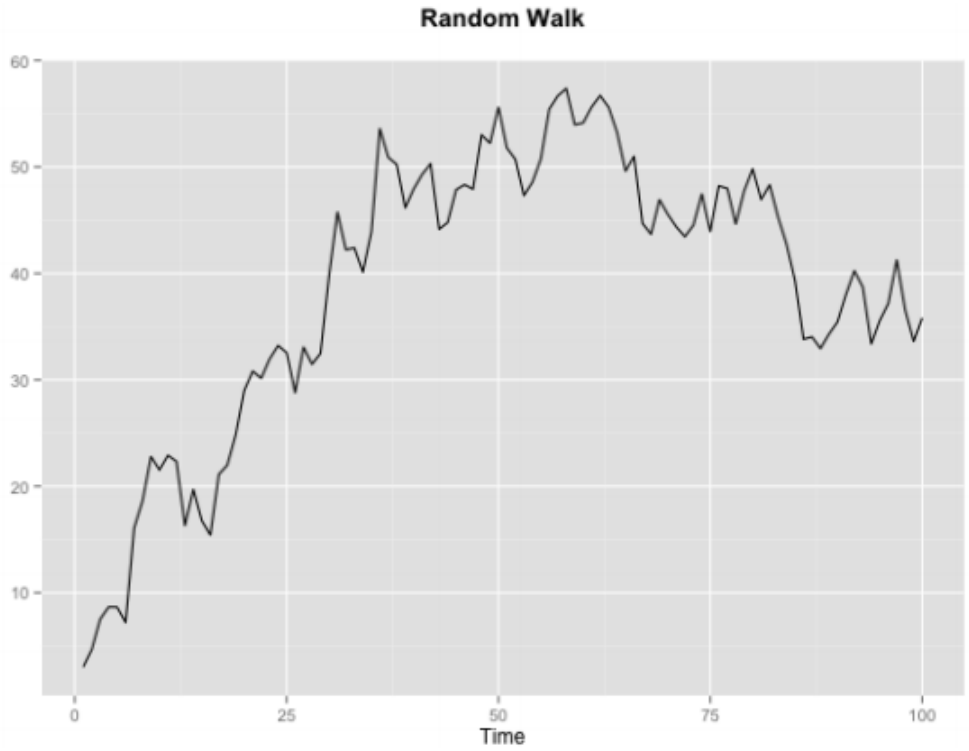
Another basic time series, like white noise.

A random walk is a time series in which the differences between time points is white noise

$$x_1 = e_1$$

$$x_t = x_{t-1} + e_t$$

($e_1, e_2, e_3, \ldots$ is a white noise time series)



some example data:

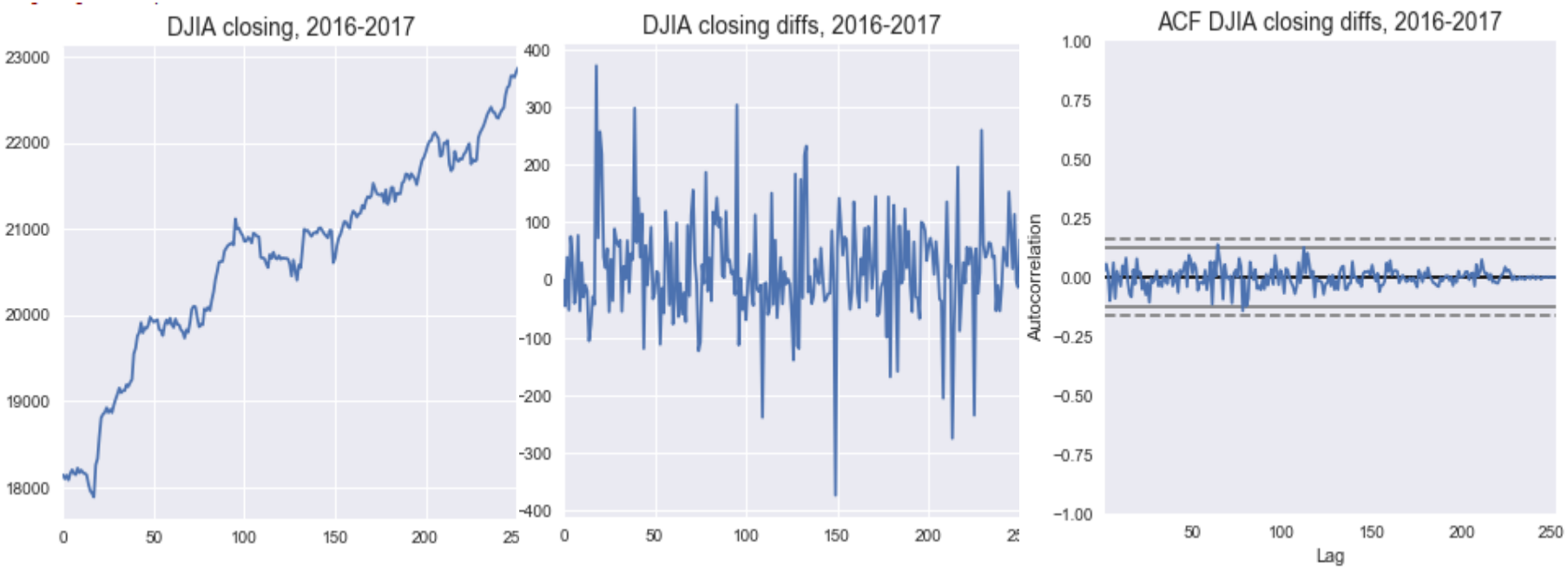
e = 0.2, -0.5, -0.3, 0.4, …

x = 0.2, -0.3, -0.8, 0.4, …

figure: Forte, Mastering Predictive Analytics with R

A random walk down Wall Street

In data from a random walk, the differences between points should be random.

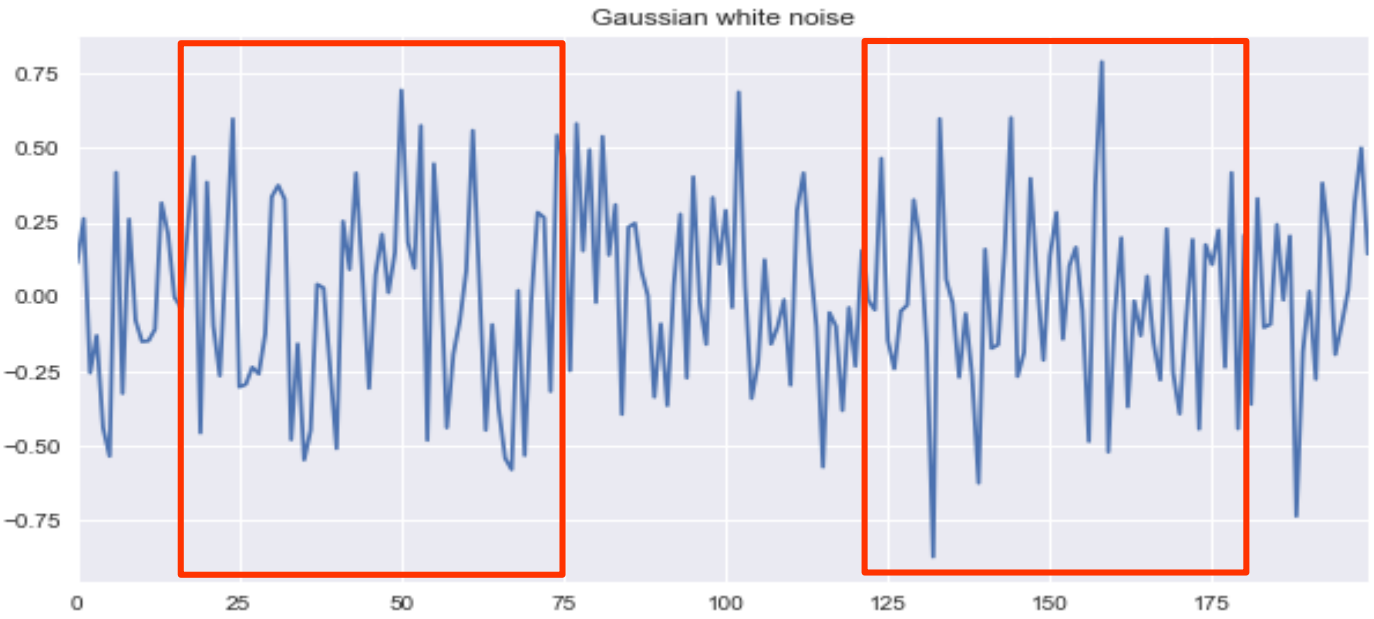


The title of this slide is the name of a famous book by Malkiel. He writes: "A random walk is one in which future steps or directions cannot be predicted on the basis of past history".

Stationarity

Roughly, this means the probabilistic behavior of a time series does not change over time.

A time series is strictly stationary if the probabilistic behavior of $\{x_{t_1}, x_{t_2}, \ldots, x_{t_k}\}$ and $\{x_{t_1+h}, x_{t_2+h}, \ldots, x_{t_k+h}\}$ are the same for all k, all time points $t_1, t_2, \ldots, t_k$ and all h.



Unit 3: time series decomposition

- definition
- identifying and removing trend
- identifying and removing seasonality

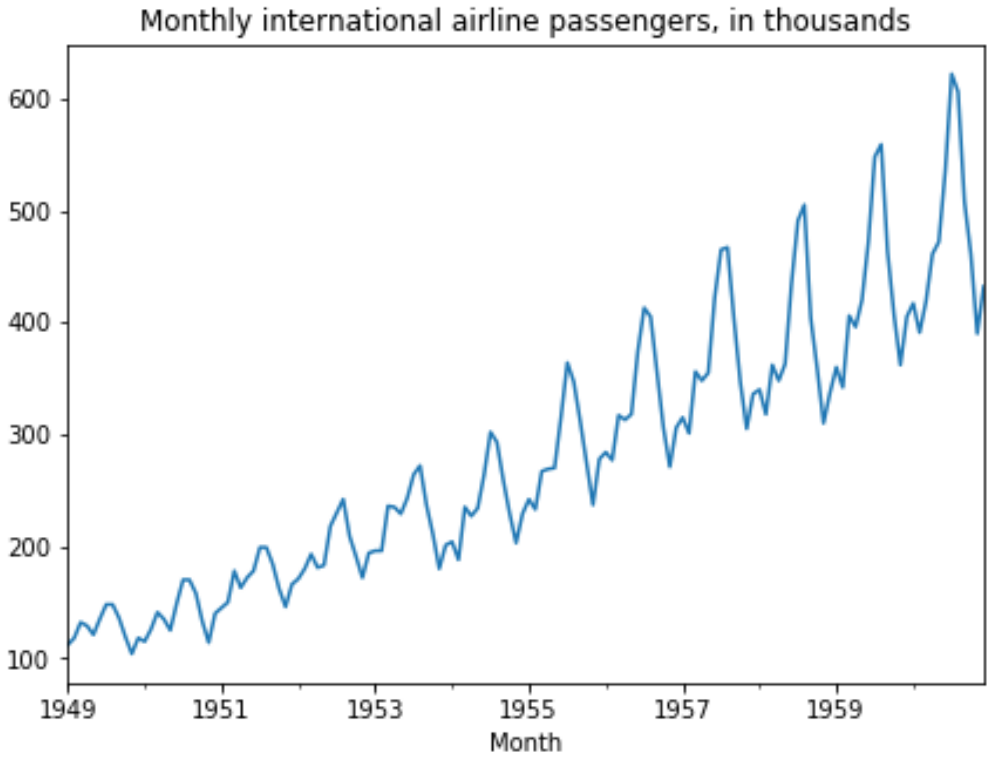
Time series decomposition

It's often useful to break a time series down into:

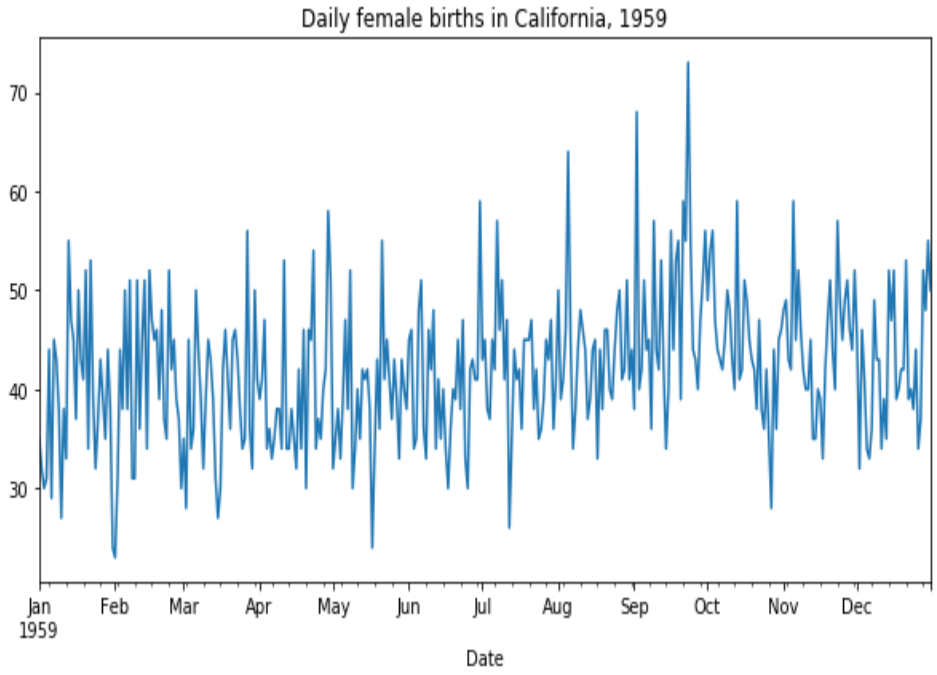
- a seasonal component (weekly, yearly, etc.)
- a trend
- noise

You get the original time series by either:

- adding the parts:
  $x_t = T_t + S_t + N_t$
- multiplying the parts:
  $x_t = T_t \cdot S_t + N_t$



Identifying and removing trends



Identifying trends is useful.

Removing trends is common in traditional time series analysis.

To identify a trend:

- study a plot of the time series
- moving average
- model fitting

To remove a trend:

- differencing
- model fitting

Identifying and removing seasonality

Identifying seasonality is useful – can make relationships clearer.

Removing seasonality is common in traditional time series analysis.

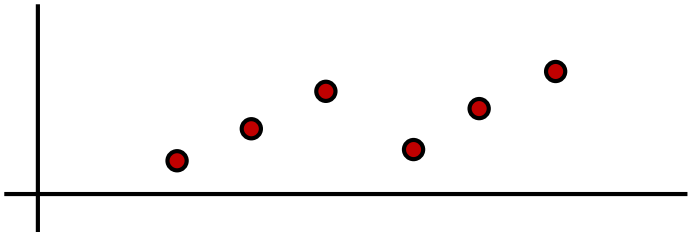
To identify seasonality:

- study a plot of the time series
- use periodicity detection algorithms; spectral analysis

To remove seasonality:

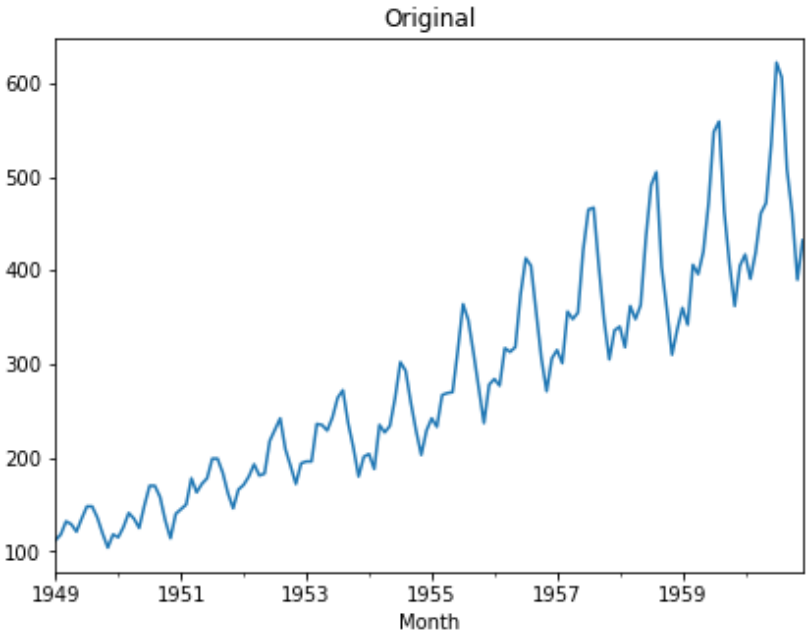
- differencing on the period
- aggregate data to the appropriate level
- use model fitting

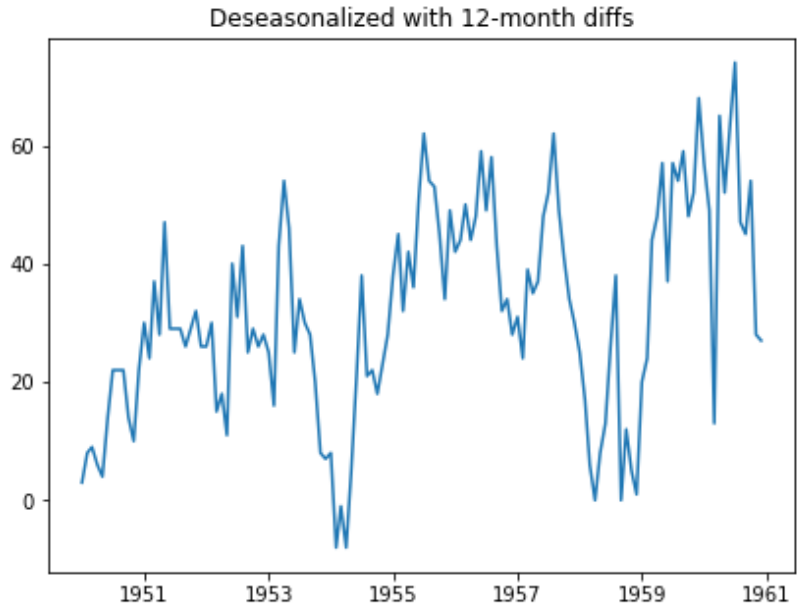
“Deseasonalizing” with differencing



period is length 3, so subtract by value 3 time units in past

```
# monthly international airline passengers data
months_per_year = 12
diff = series.diff(months_per_year)      # pandas
```





Unit 4: ARIMA models

- AR (Autoregressive)
- MA (Moving Average)
- ARMA (Autoregressive Moving Average)
- ARIMA (Autoregressive Integrated Moving Average)

Autoregressive (AR) models

- a linear regression on past observations

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t$$

$\phi_1, \phi_2, \ldots$ are constants, with $\phi_p \neq 0$
w_t is Gaussian white noise with mean 0

- past values influence the current value (e.g., if GDP is high this quarter, we expect it to be high next quarter)
- random walk is an example
- "normally" it is stationary

if p = 1, then we have an AR model of order 1, written AR(1)

if p = 2, AR(2), etc.

Moving average (MA) models

- a linear regression of past white noise values

$$x_t = w_t + \beta_1 w_{t-1} + \beta_2 w_{t-2} + \cdots + \beta_q w_{t-q}$$

w_t is Gaussian white noise with mean 0

- intuitively, the process is responding to random "shocks" (e.g., a severe earthquake affects the economy)
- the influence of a shock can persist (e.g. the earthquake affects the economy not only now but in the near future)
- an MA process is always stationary

if q = 1, then we have an MA model of order 1, written MA(1)

if q = 2, MA(2), etc.

ARIMA

- ARIMA: Autoregressive Integrated Moving Average Models
- ARIMA = ARMA, plus differencing to handle non-stationarity
- A time series follows an ARIMA(p,d,q) process if the d^{th} differences of the series are an ARMA(p,q) process.

Key thing: ARIMA can handle non-stationary time series, as long as stationarity can be achieved by simple differencing

Seasonality can't usually be removed with simple differencing

Unit 5: ARIMA forecasting