Exam 2 preview

Glenn Bruns
CSUMB

much of this material is based on Geron's "Hand-on" text

Coverage of exam

All material up to CNNs

Focus on material since the last exam:

- dimensionality reduction
- TensorFlow and neural networks

However, exam will also include:

- linear algebra, training models, support vector machines, ensemble learning

Structure of exam

1. Concepts and theory (25 mins)
   - conceptual questions
   - questions about the math
   - paper and pencil – no notes or other resources
2. Practical (45 mins)
   - add your code to starter iPython notebook
   - use any resources you like

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

Topics since exam 1

- Dimensionality reduction
  - PCA (part of chap 8)
- Neural nets
  - TensorFlow (chap 9)
  - Neural nets (chap 10)
  - Training deep neural nets (chap 11)
  - CNNs (chap 13)

Dimensionality reduction: PCA

- 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

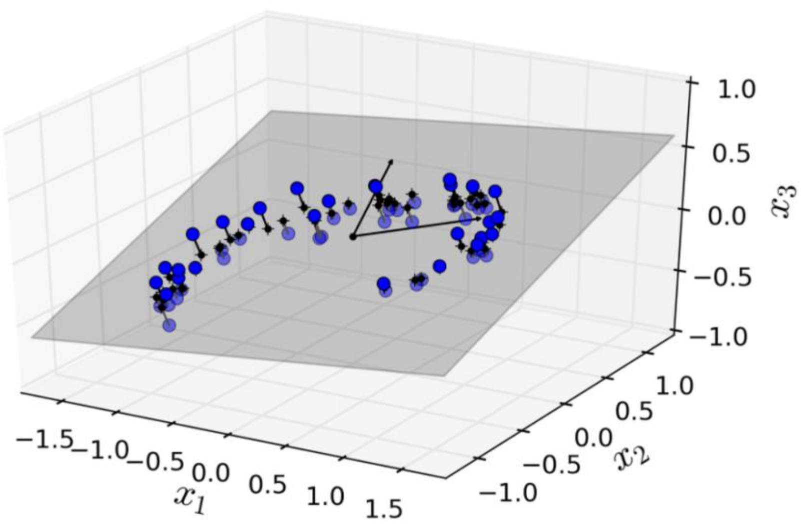
Dimensionality reduction

Two 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?

Principal component analysis

Main idea:

- The single best axis is the one with most variance
- The next best axis is the one, of those orthogonal to the best axis, with the most variance

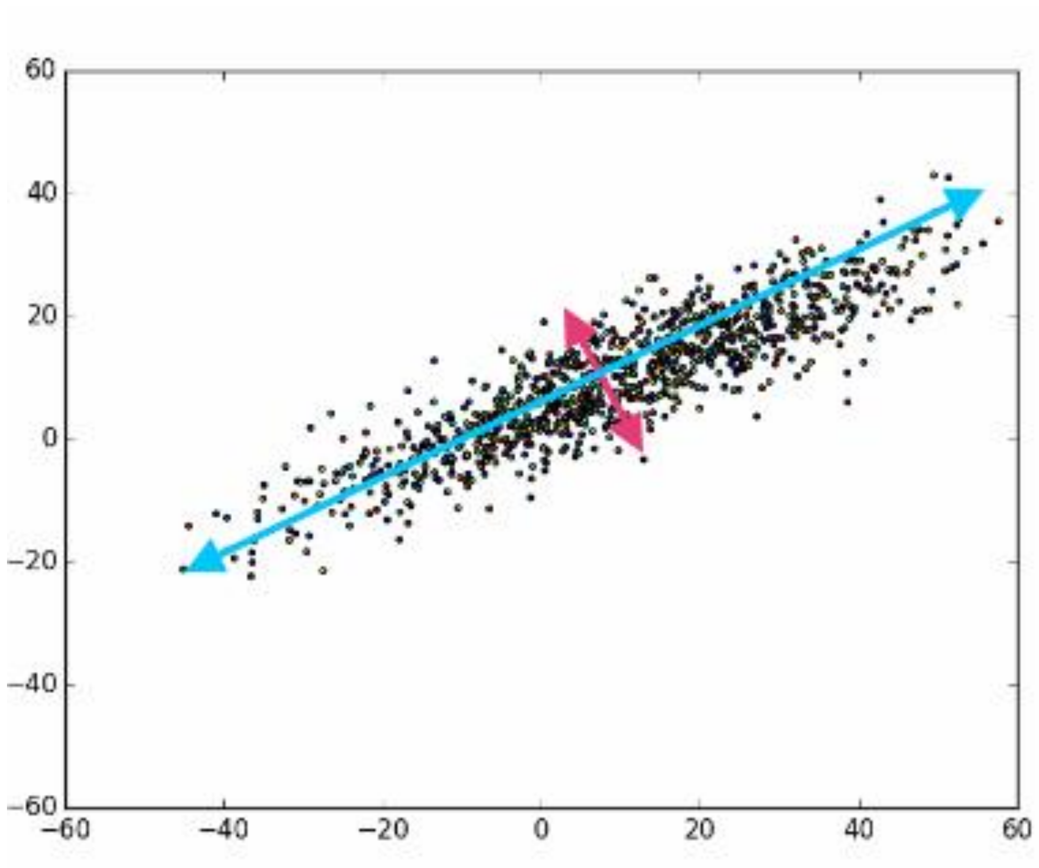


figure: austingwalters.com/pca-principal-component-analysis/

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:

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

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.

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

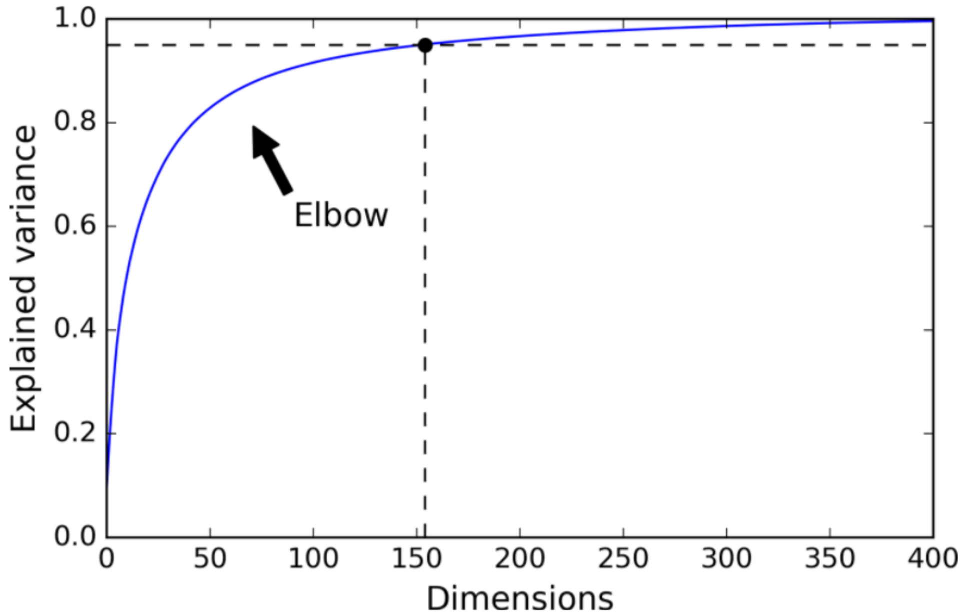
an original feature vector

first principal component

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
```



Kernel PCA

PCA provides a linear transformation of the original data.

What if we need a non-linear transformation?

	SVM	PCA
linear case	linear SVM	PCA
non-linear case	SVM with polynomial or RBF kernel	kernel PCA

Tuning kernel PCA, example

A nice example from Géron showing the use of a Pipeline as a predictor in GridSearchCV.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
    ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

Computational complexity

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

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

LLE: $O(m \log(m)) \log(k)$ (find k-nearest neighbors)

$+ O(mnk^3)$ (reconstruction cost)

$+ O(dm^2)$ (embedding cost)

Recall:

m number of instances

n number of features

k number of nearest neighbors

d # of dimensions in lower-dimensional space

Neural nets: introduction to TF

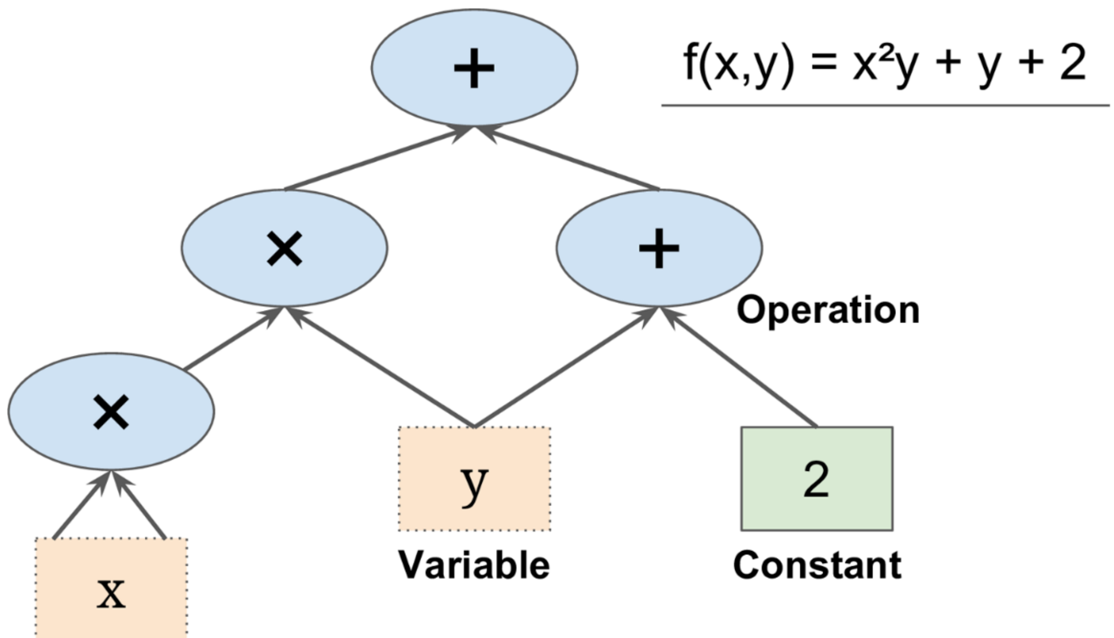
- computation model
- handling sessions
- evaluating a node
- linear regression example

What is TensorFlow?

It's an open source library for numerical computation.

Principle:

1. define a computation graph
2. execute the graph efficiently with C++ code



You can even break the graph into pieces and run them on separate CPUs or GPUs.

Neural nets: linear regression with TF

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

Using TensorFlow's gradient descent

```
…
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = tf.gradients(mse, [theta])[0]
training_op = tf.assign(theta, theta - learning_rate * gradients)
```

You can use one of TensorFlow's built-in optimizers instead:

```
…
y_pred = tf.matmul(X, theta, name="predictions")
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)
```

Switching to an alternative TF optimizer is super-easy.

Neural nets: intro

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

Artificial neurons

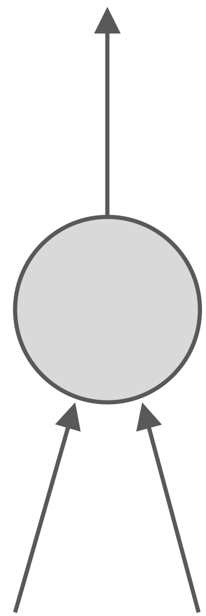
A first mathematical model of neurons, by McCulloch and Pitts.

artificial neuron:

- one or more binary inputs
- one binary output
- output activated when a certain number of inputs are active

example:

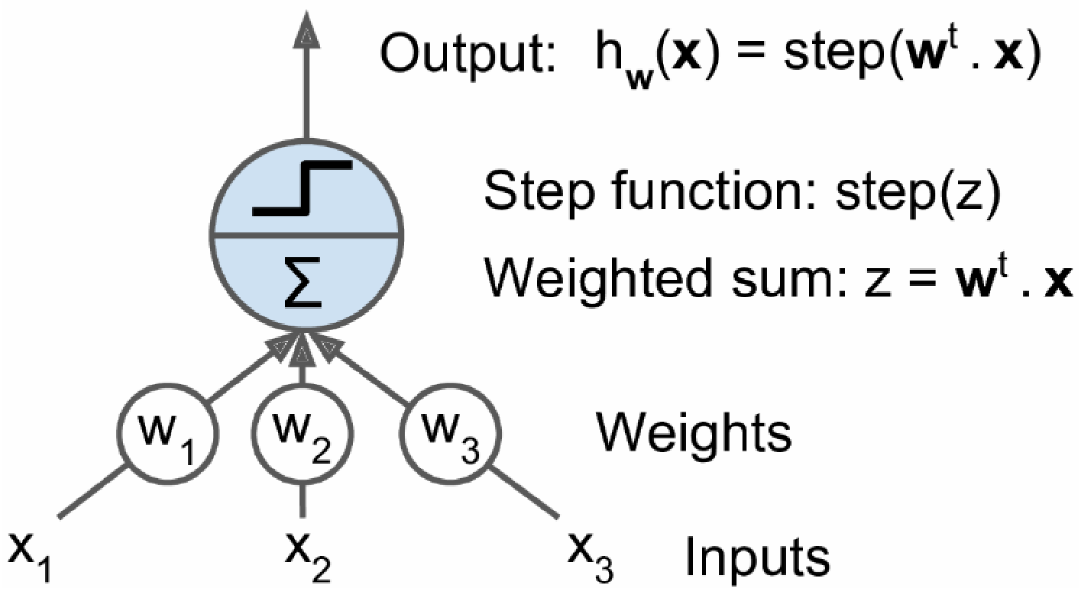
- one input, two outputs
- activate output when both inputs active



Linear Threshold Unit LTU

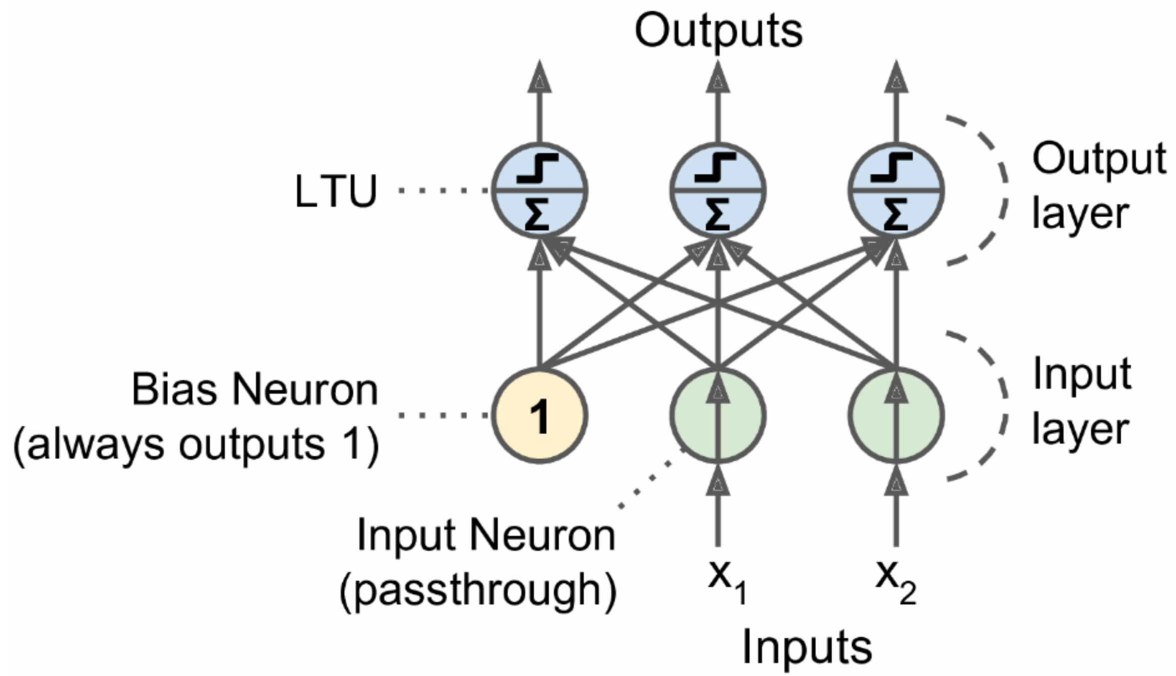
An LTU is another kind of artificial neuron.

- It has numeric, not boolean, inputs and outputs.
- It computes a weighted sum of its inputs.
- The output is a step function applied to the weighted sum.



Perceptrons

A perceptron is a single layer of LTUs, with each neuron connected to all the inputs.



The figure shows a perceptron with 2 inputs and 3 outputs.
The "input neurons" output whatever input they are fed.

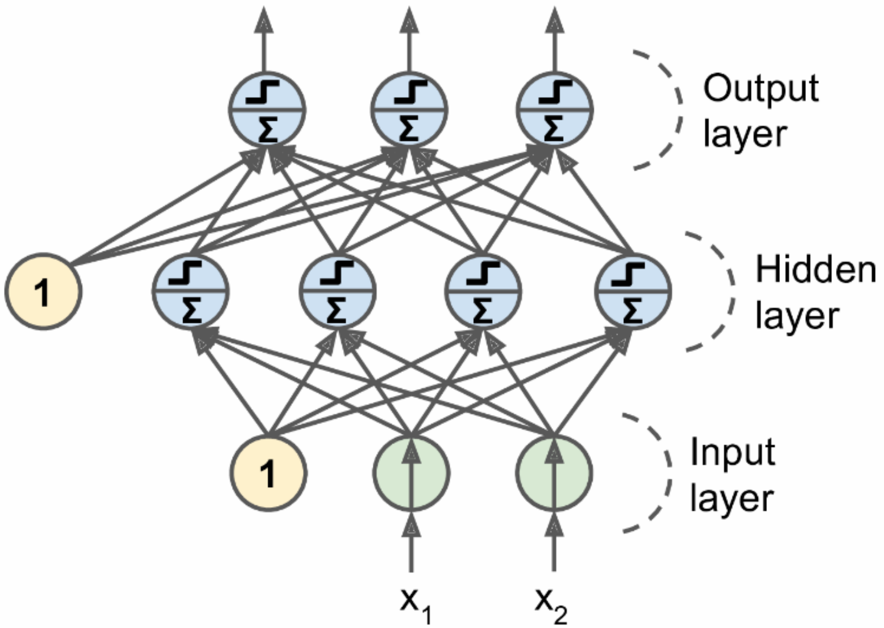
Multi-Layer Perceptron

A multi-layer perceptron has:

- one input layer
- one or more layers of LTUs (hidden layers)
- one final layer of LTUs (output layer)

Every layer but the output layer:

- includes a bias neuron
- is fully connected to the next layer



Activation functions

Key to backprop:

- instead of step function, use logistic function: $\boldsymbol{\sigma}(\boldsymbol{z}) = \frac{1}{1+\exp(-z)}$

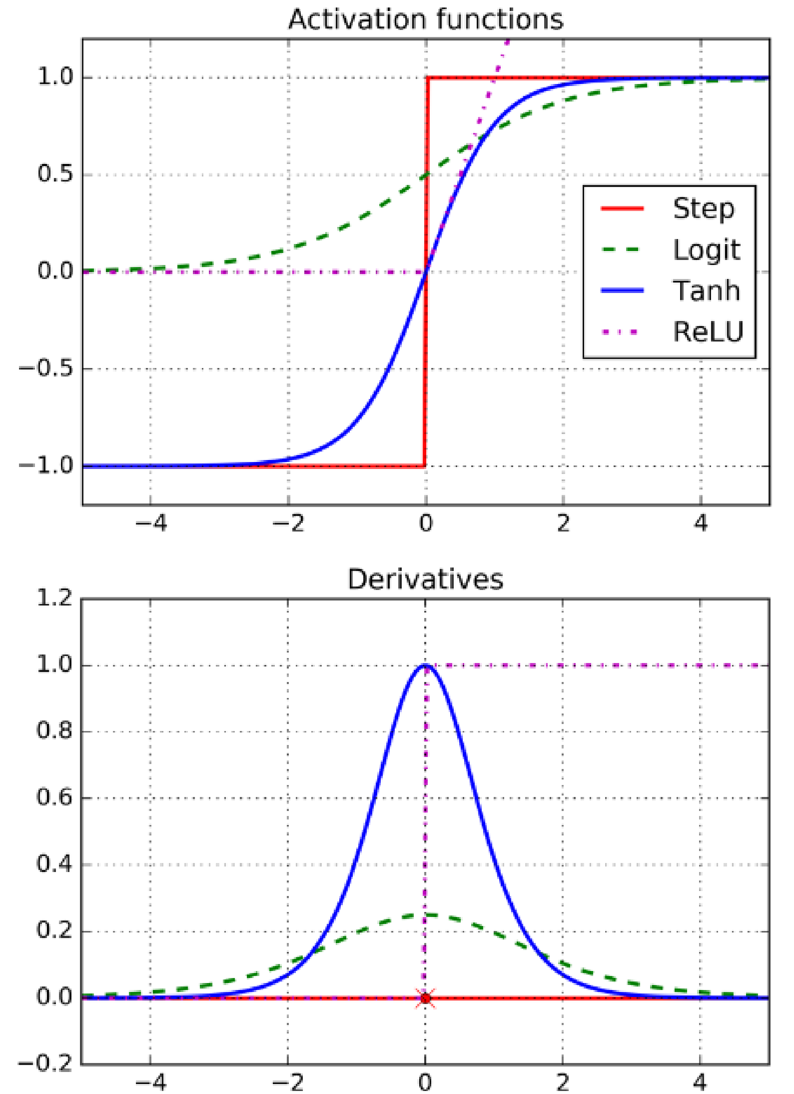
This is called an activation function. Other popular activation functions:

- hyperbolic tangent function:

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

- ReLU function:

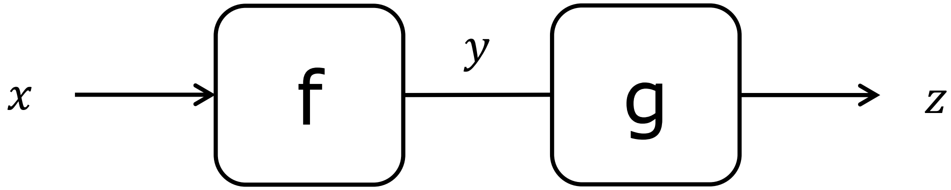
  $$\boldsymbol{ReLU}(\boldsymbol{z}) = \max(0, z)$$



Neural nets: backpropagation

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

Backprop: example with two nodes



$z = y^2$

$y = x + x^\wedge 3$

Approach 1: combine f and g to get $z = (x + x^3)^2$ and then do the same thing as the last example.

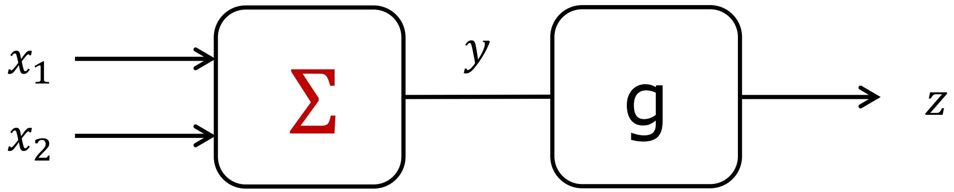
Approach 2: get $\frac{dz}{dx}$ by using the chain rule: $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

Example: Let the input x be 1 (so y is 2 and z is 4). How to adjust x to make z smaller?

Work out that $\frac{dz}{dy}$ is $2y$, and $\frac{dy}{dx}$ is $1 + 3x^2$. Then the chain rule says that the slope of the f,g combined, at x=1, is:

$\frac{dz}{dy}(2) * \frac{dy}{dx}(1) = 4 * 4 = 16.$ For new x, use $x - \eta \frac{dz}{dx}(x)$

A weighted sum node



$z = y^2$

$y = \mathrm{w}_1\, x_1 + w_2\, x_2$

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 $w_1 - \eta \frac{\delta z}{\delta w_1}(0.5, 2.0)$

Notes from Goodfellow et al

- Multilayer perceptrons (MLPs) are also called deep feedforward networks
- back-propagation is <u>not</u> gradient descent – it refers only to the method for computing the gradient
- back-propagation is a very general technique:
  - is not limited to the gradient of a cost function with respect to its parameters
  - not limited to neural networks

source: Deep Learning, by Goodfellow, Bengio, and Courville

Neural nets: training

- list some of the APIs built on top of TensorFlow
- build and train a DNN using TensorFlow
- tune the hyperparameters of your neural net

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

Training DNNs: vanishing gradients

- diagnose and address the vanishing and exploding gradients problem

Causes of the problem

"Understanding the Difficulty of Training Deep Feedforward Neural Networks" (Glorot and Bengio, 2010), diagnosed the issue.

1. Use of the sigmoid activation function
2. Method used to initialize network parameters
   - random initialization with mean 0, std dev 1

Idea:

- with 1 and 2 above, the outputs of a layer have greater variance than the inputs of a layer
- activation function then saturates in top layers

Idea 2: dump the sigmoid activation

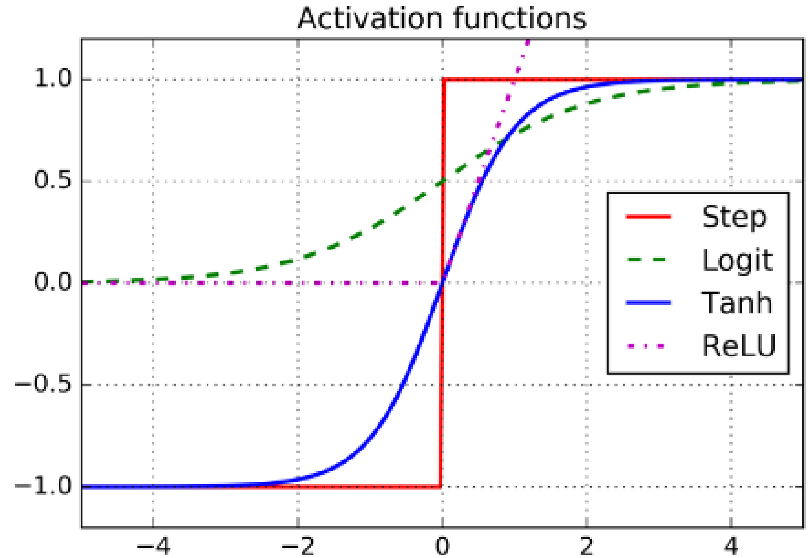
Replacement candidate 1:
ReLU activation function

pros:

- doesn't saturate for positive values
- fast to compute

cons:

- dying ReLU problem: many neurons in network stop outputting anything other than 0
- if weighted sum of neuron's inputs is negative, ReLU gives 0
- once this happens, it tends to stay that way

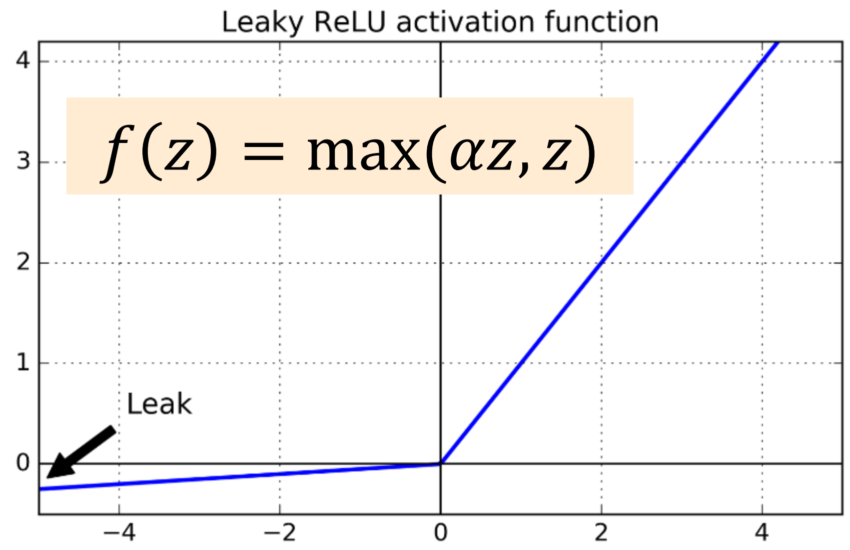


Dump the sigmoid (cont'd.)

Replacement candidate 2: leaky ReLU

pros:

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



α is a hyperparameter; typically set to 0.01

values of up to 0.2 may work even better

Other cousin candidates:

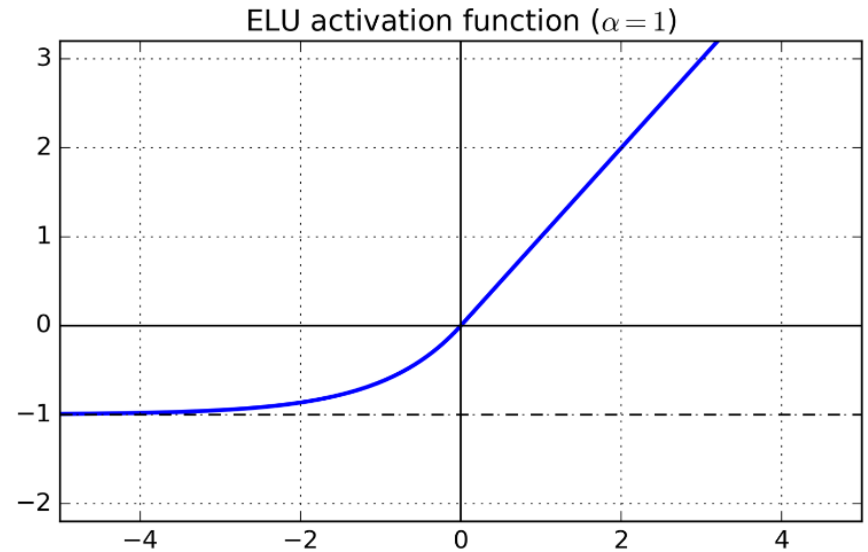
- randomized leaky ReLU
  - α is picked randomly during training
- parametric leaky ReLU
  - α is learned during training

Dump the sigmoid (cont'd.)

Replacement candidate 3: exponential linear unit (ELU)

pros:

- outperformed all ReLU variants in a recent study
- training time reduced; performance better
- function is smooth everywhere, helping with gradient descent
- negative value when z < 0, avoids vanishing gradients
- non-zero gradient when z < 0, avoids dying units



$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

hyperparameter usually set to 1

cons: slower to compute than ReLU and its variants

Solution idea 3: Batch normalization

He initialization plus better activation function especially helps with vanishing/exploding gradients <u>during beginning of training</u>.

A newer technique (2015) is batch normalization.

Idea:

- add operation just before activation function
- this operation lets the model learn the optimal scale and mean of the inputs for each layer
- inventors claim many benefits, including reduction in vanishing gradients, and better performance

See textbook for using batch normalization in TensorFlow

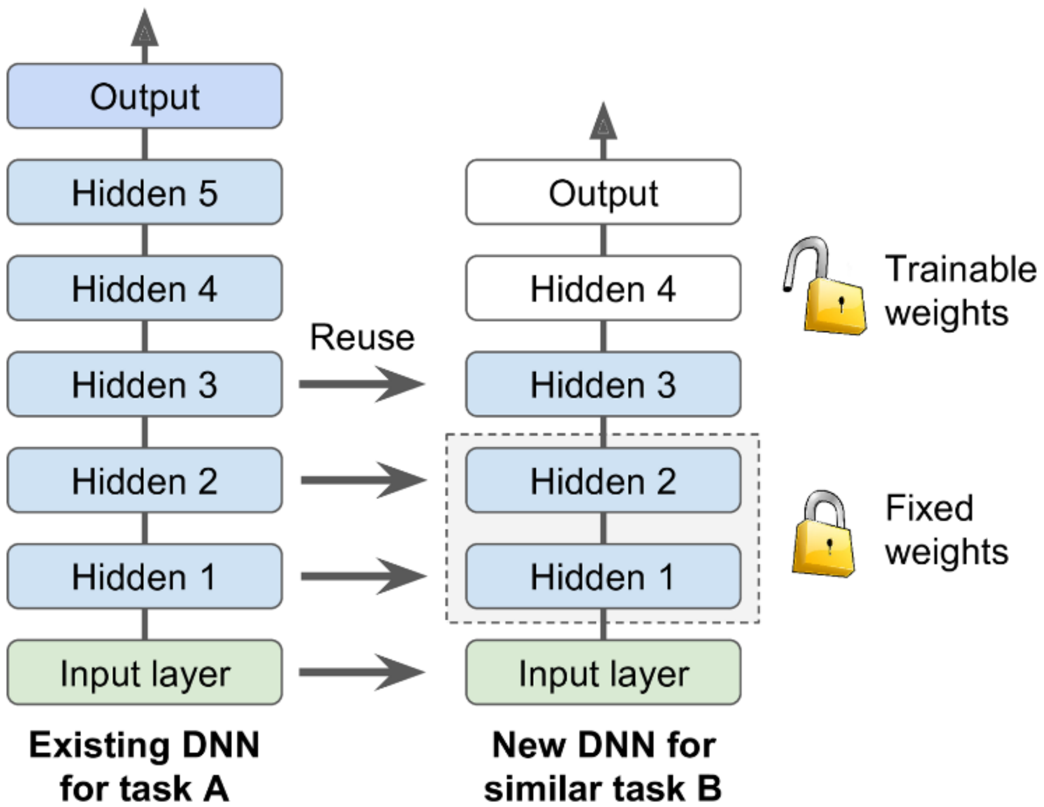
Training DNNs: optimizers

- reuse parts of TensorFlow models
- select a fast optimizer

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.

Caching layers

If the reused layers are frozen, you only need to compute the output of the topmost frozen layer once.

Strategy:

- run the whole training set through the lower levels
- don't batch training data – batch outputs from the topmost frozen layer
- use these during training

Strategies for tweaking reuse

- we know we don't want to reuse the output layer of original model
- how many of the hidden layers should be reused?
  - the higher the level, the less likely to be used
- strategy:
  - start by freezing all borrowed layers
  - train your model, see how it performs
  - if performance not very good, unfreeze one or two top borrowed layers
  - if performance still not very good, drop or replace the topmost borrowed layers

Momentum optimization

Idea: gain momentum on downhill slopes and "roll past" local minima

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

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

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

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

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

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

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

Nesterov Accelerated Gradient

- A variant of Momentum optimization
- Idea: measure gradient at position slightly ahead of direction of momentum

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

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

- In plain Momentum optimization we had

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

"look where you're going"

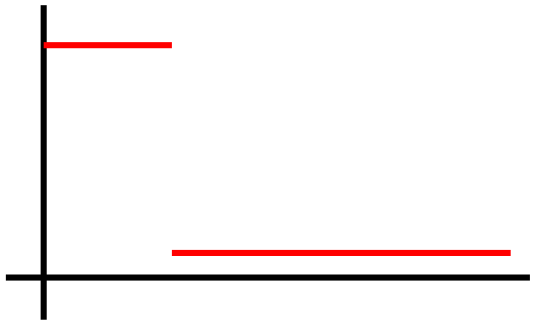
Training DNNs: learning rate, regl'zn

- learning rate scheduling in DNNs
- regularization in DNNs

Learning rate scheduling

Predetermined piecewise constant learning rate:

- reduce learning rate every so many epochs
- example: initially $\eta = 0.1$, then 0.001 after 50 epochs



Performance scheduling:

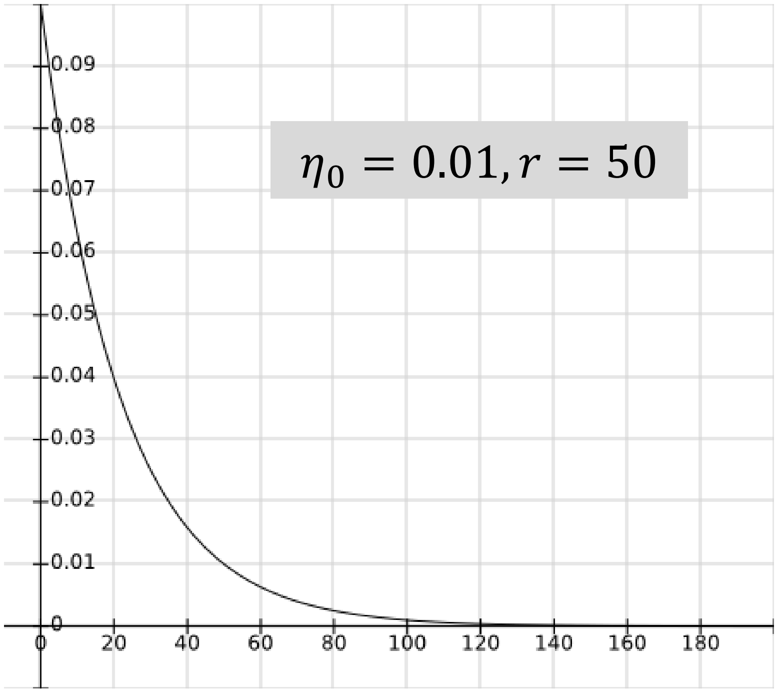
- measure test error every N steps
- reduce learning rate by a factor of λ when the error stops dropping

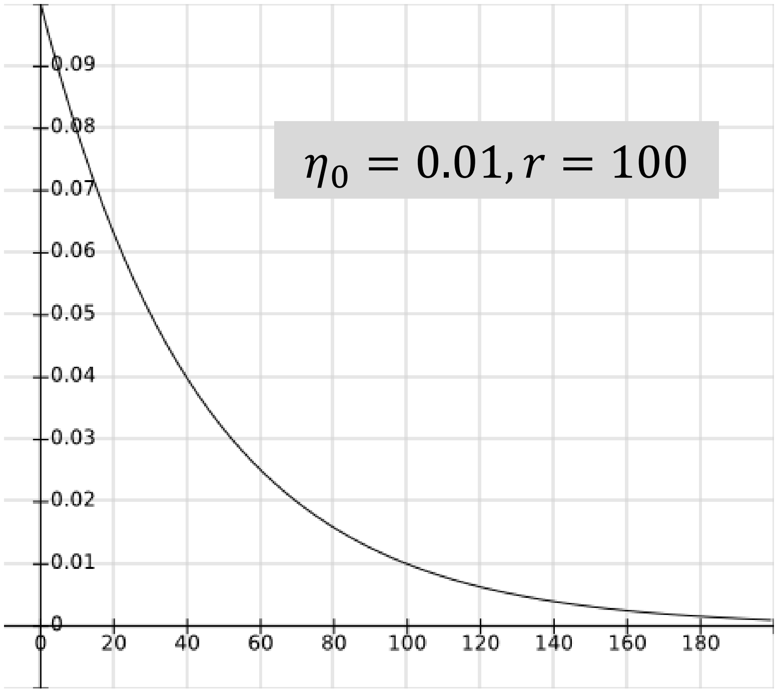
Exponential scheduling

- learning rate a function of iteration number:

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

- two tuning parameters: η_0, r



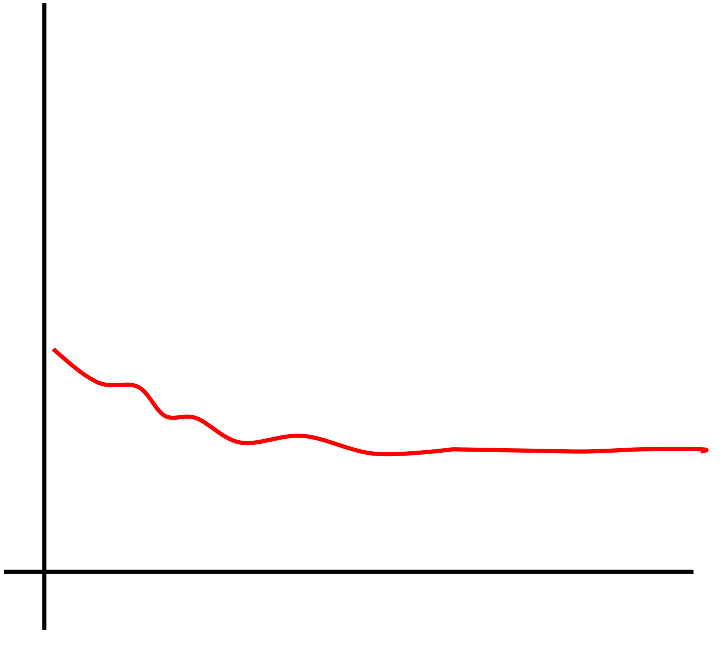


Early stopping

Stop training when performance on test set starts dropping

One way, in TensorFlow:

- periodically evaluate model on test set
- save a 'winner' snapshot if it outperforms earlier winner snapshots
- stop training if number of steps since last winner exceeds a threashold (like 2000 steps)



Dropout

Super effective and popular.

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

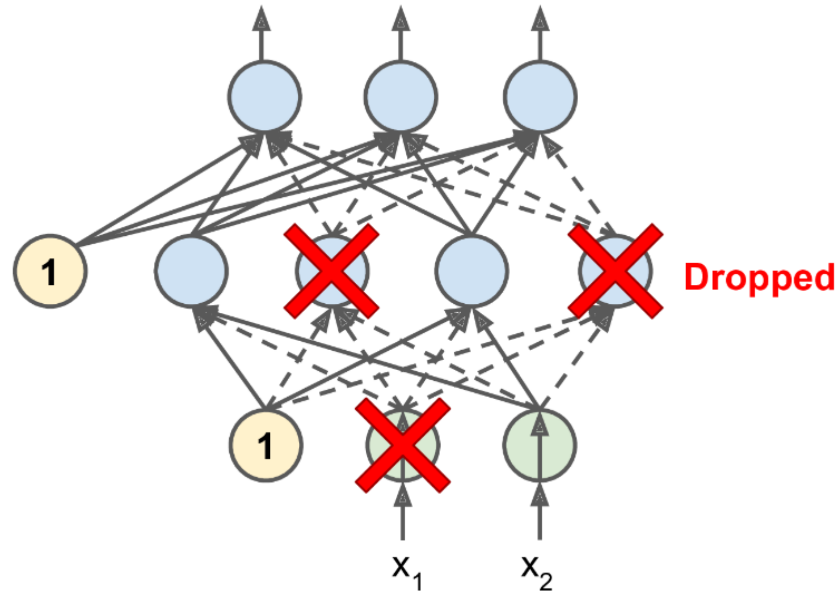


figure source: Géron

Practical guidelines

Configure your DNN like this by default:

- initialization: He initialization
- activation function: ELU
- normalization: batch normalization
- regularization: dropout
- optimizer: Adam
- learning rate schedule: none

These guidelines are straight out of Géron's book.

Practical guidelines, cont'd.

Tweaking your configuration:

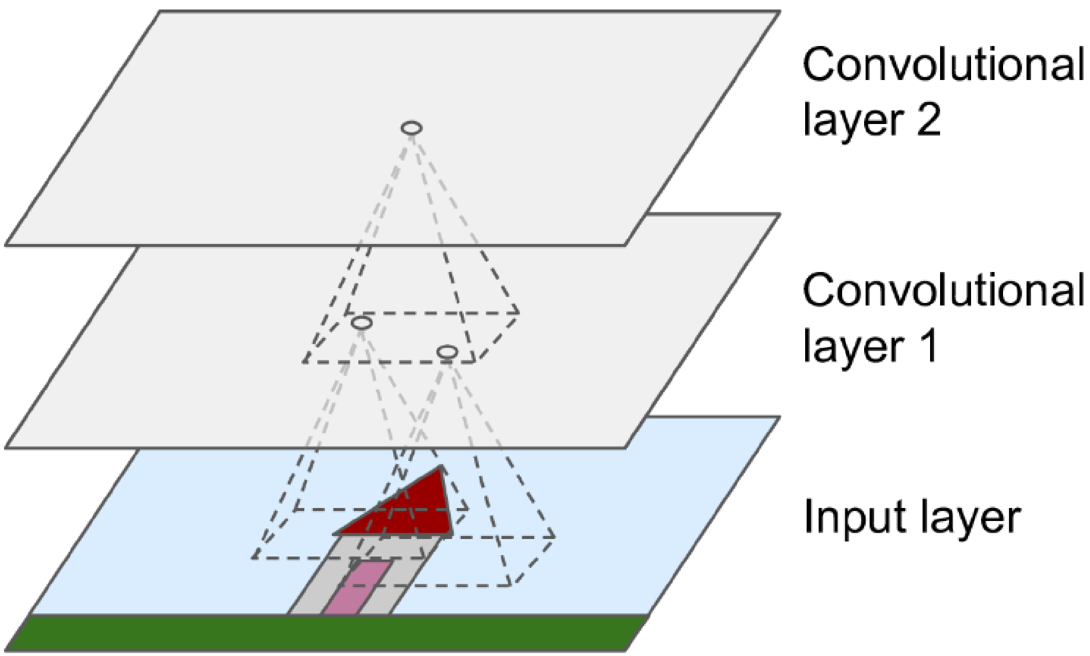
- Can't find good learning rate? Try a learning schedule, such as exponential delay
- Training set too small? Try data augmentation
- Need a sparse model? Add l_1 regularization, or FTRL instead of Adam optimization
- Need a fast model at runtime? Drop batch normalization, replace ELU with leaky ReLU

Convolutional neural nets

- explain the biological motivation for convolutional neural nets
- define the concepts of:
  - convolutional layer
  - padding
  - stride
  - filter
  - feature map
- write the expression defining the output of a neuron in a convolutional layer

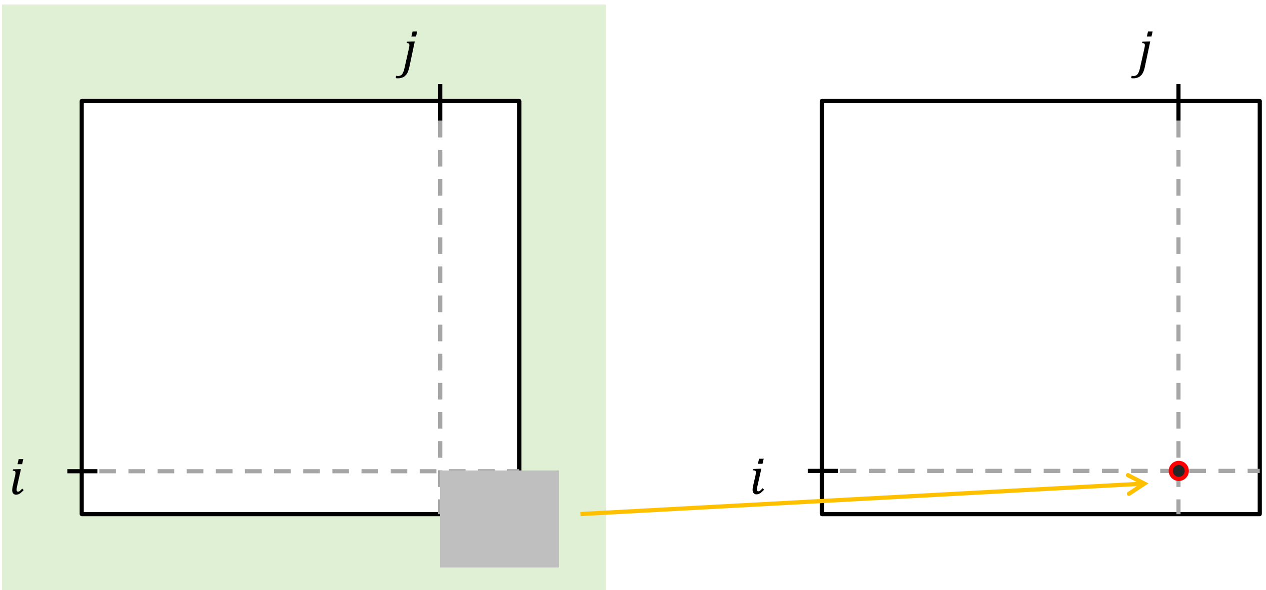
Convolutional layer

- 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

How to deal with this situation?



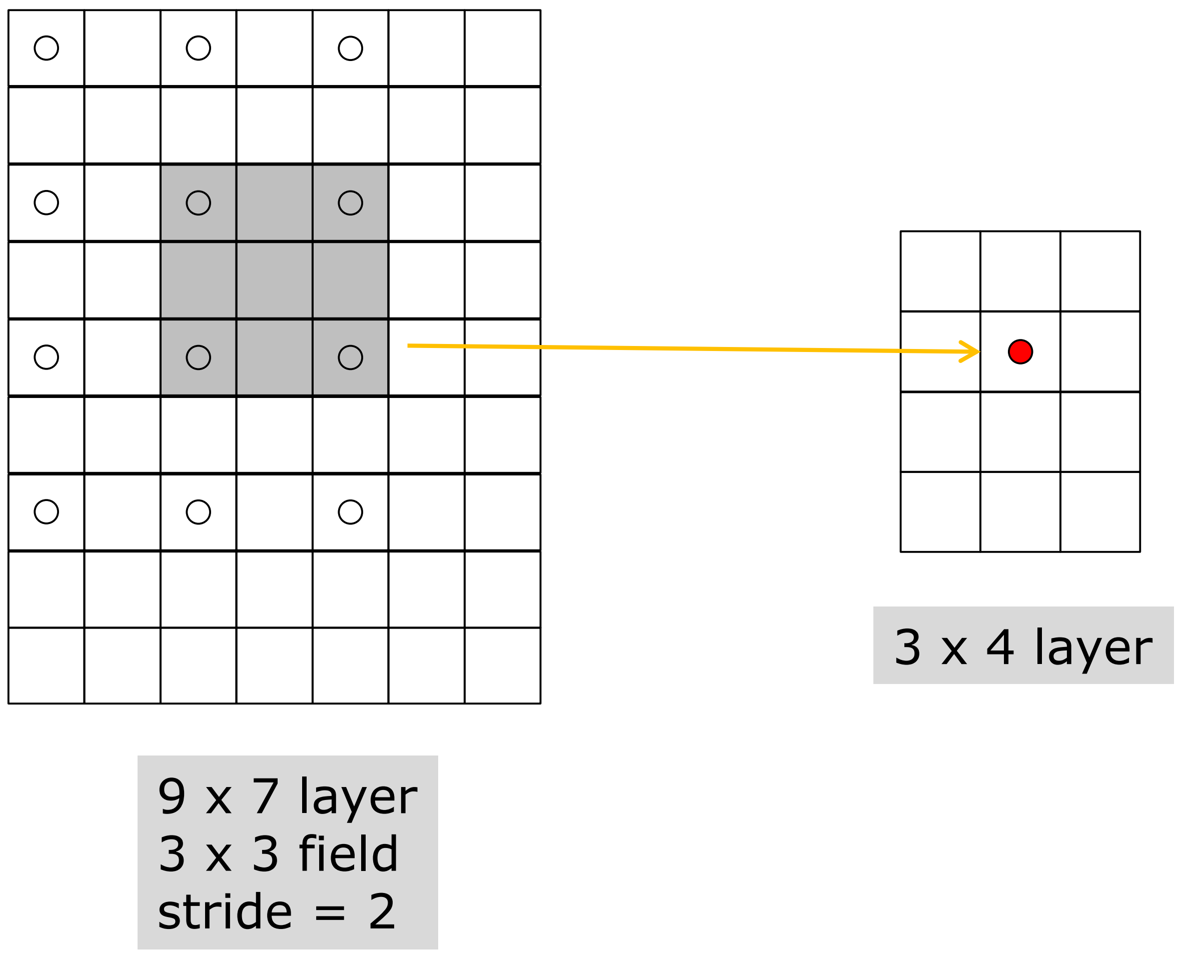
Idea 1: use a second layer that is smaller than the first

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

"zero padding"

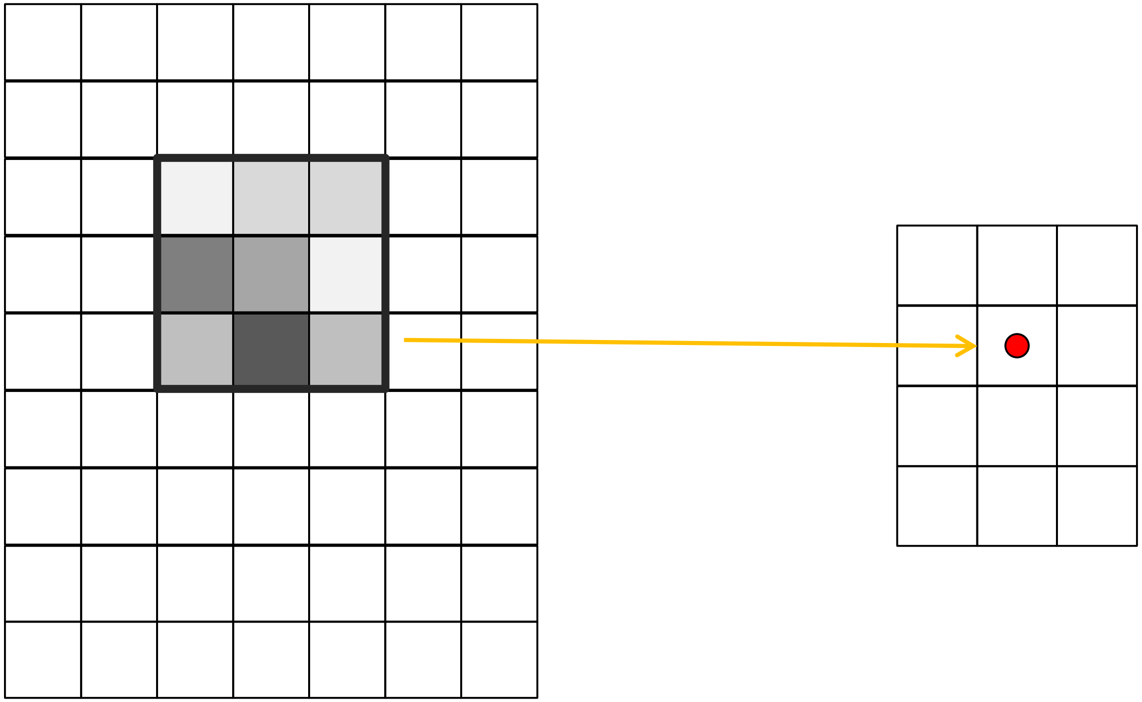
Stride

A receptive field can skip over rows and columns of a layer



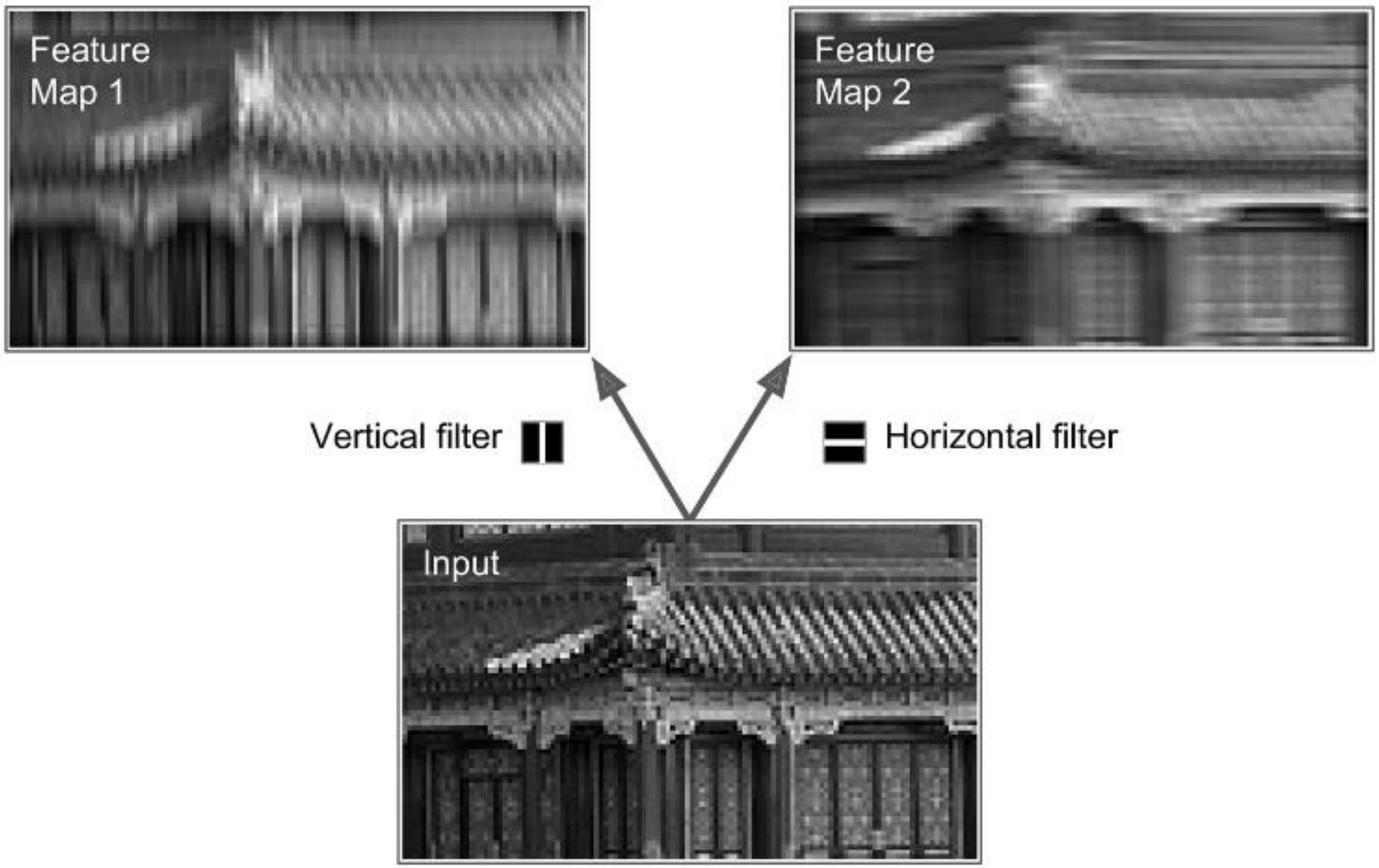
Filters

The weights associated with neuron on the right can be shown as an image – like a heatmap

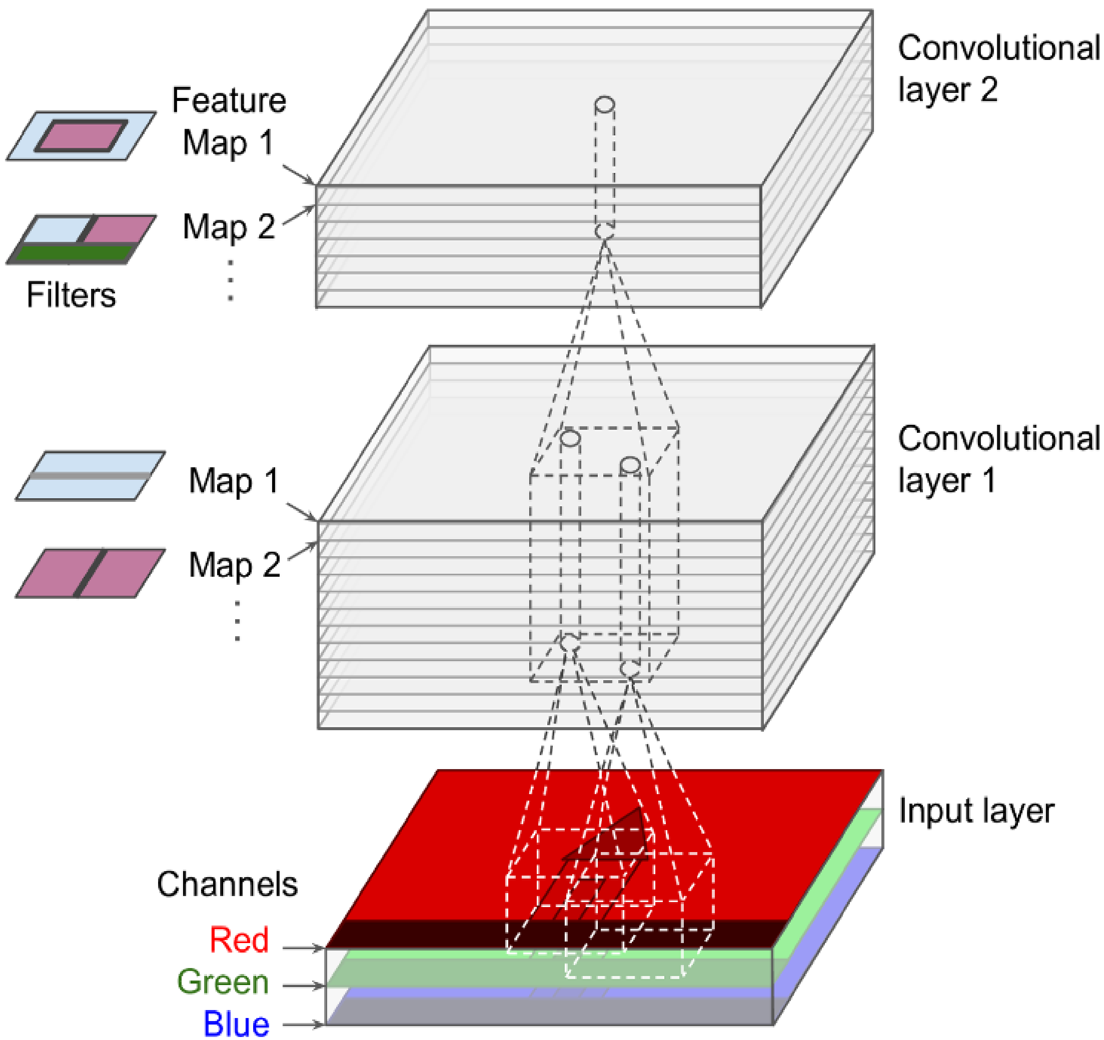


Feature maps

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

Pooling

Basically: aggregate over receptive field

As with convolution, you must specify:

- size of the rectangle
- stride
- padding type

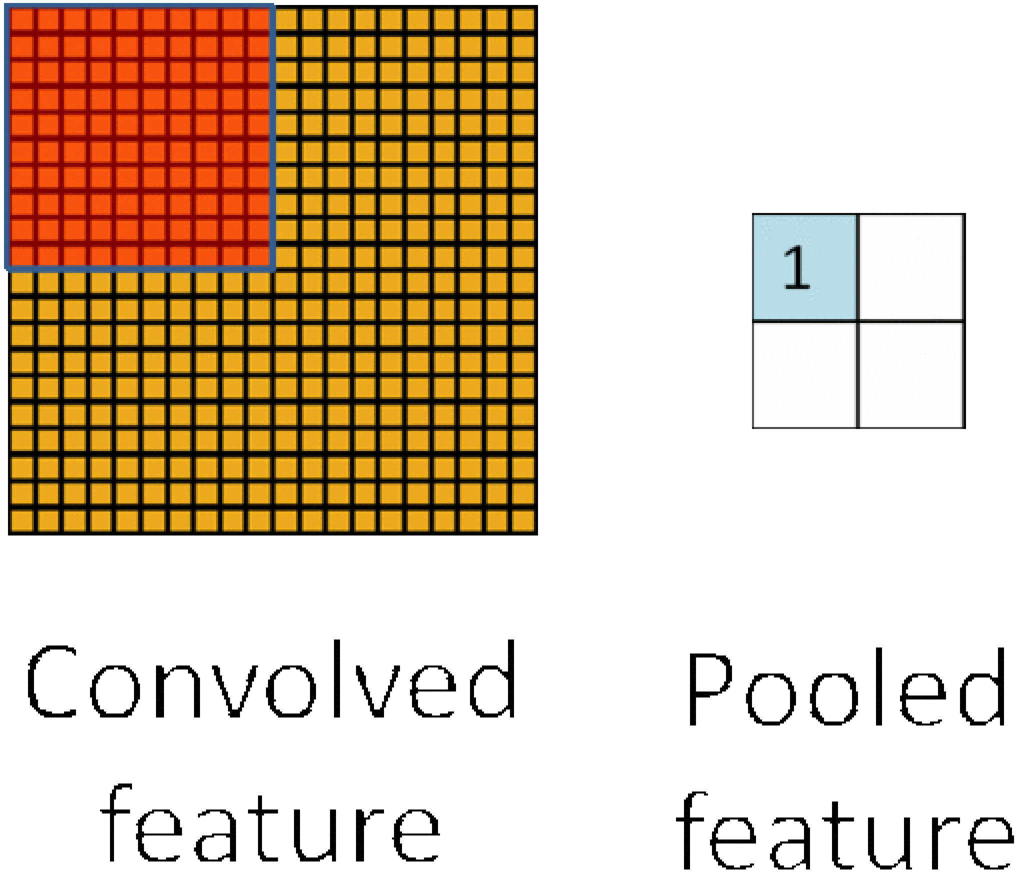
But now, instead of weights, also specify:

- aggregation function (max, mean, etc.)

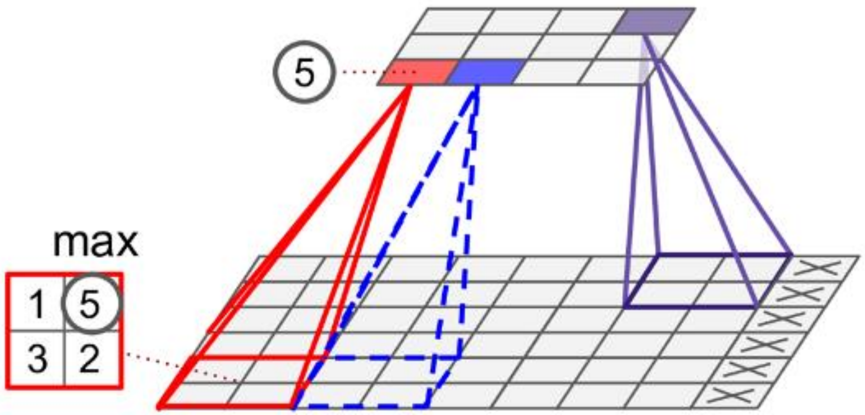
Reduces computational load, memory usage, and number of parameters

Question: how is overfitting affected?

Pooling



Max pooling example

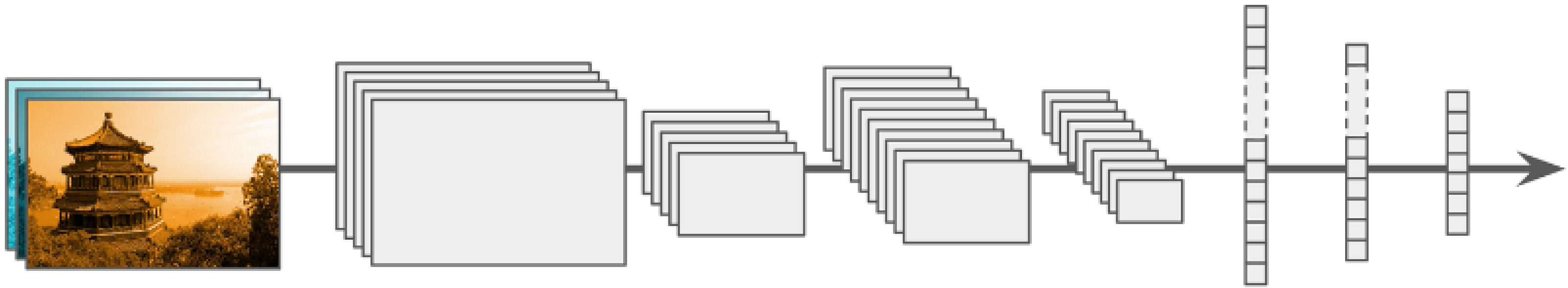


Notes:

- 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)