TAMPERE UNIVERSITY OF TECHNOLOGY

SIGNAL INTERPRETATION

Lecture 5: Random Forest, Neural Networks

February 4, 2016

Heikki Huttunen

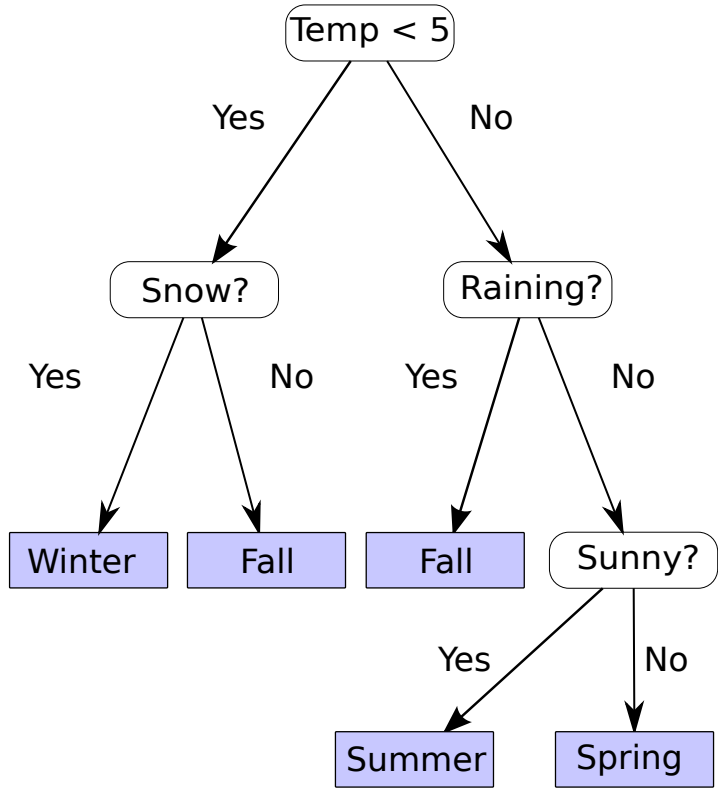
heikki.huttunen@tut.fi

Department of Signal Processing
Tampere University of Technology

RANDOM FOREST
(and other ensemble methods)

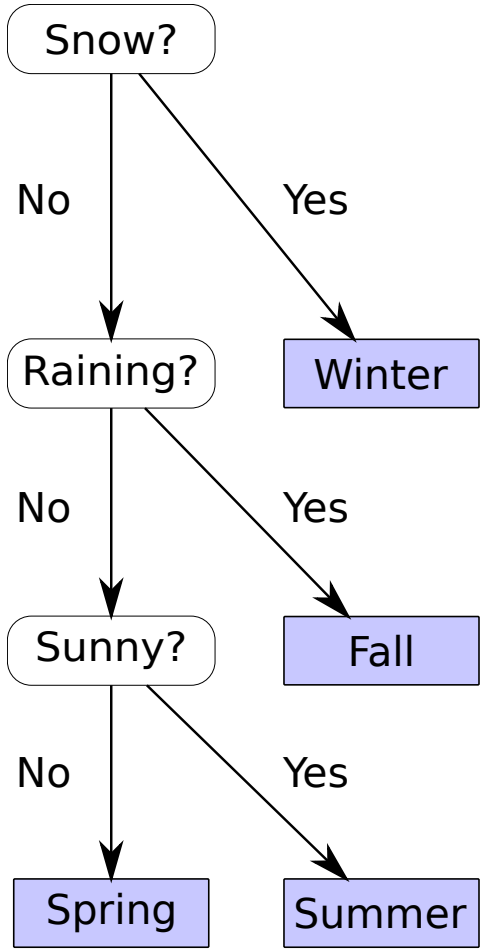
Random Forest

- Random forest (RF) is popular tree based classifier, proposed by Breiman in 1993.
- The RF is based on a collection of *decision trees*.
- A decision tree is a straightforward if-then-else-like diagram that combines individual attributes into a decision.
- Decision trees are easily trained to learn the data.
- For example, the tree on the right predicts the time of year from the following measured attributes: $\{\text{Temperature}, \text{Snow}, \text{Rain}, \text{Sunny}\}$.



Random Forest

- The problem of decision trees is that they *overlearn* the data, *i.e.,* the training data is memorized with a poor ability to generalize.
- With no restrictions, the DT will have one root-leaf-path per sample, which is essentially same as nearest neighbor.
- RF avoids this by training many "imperfect" trees.
- Each tree is trained with a subset of samples and a subset of features, *i.e.,* some of the attributes are hidden from the training.



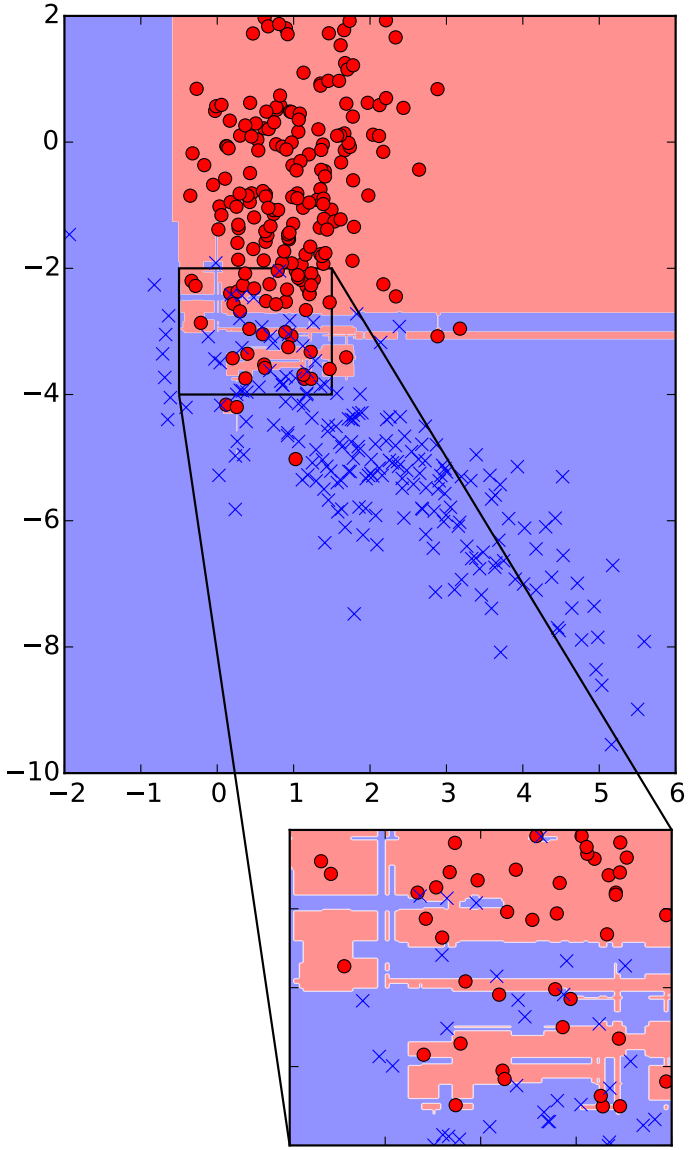
Decision Tree without the *Temperature* attribute.

Random Forest

- The RF extracts values at randomly selected coordinates.
- For example, if we have a data matrix $\mathbf{X} \in \mathbb{R}^{10\times 4}$ (*i.e.,* 10 samples with 4 features each), we might train trees with the following subsets of the data:
  - **Tree 1:** Train using rows {6,4,7,2,6,3} and columns {1,2,4}
  - **Tree 2:** Train using rows {1,10,2,1,7,9} and columns {1,3}
  - **Tree 3:** Train using rows {7,2,7,3,9,3} and columns {2}
  - ...
- Note that rows (samples) are sampled *with replacement*, *i.e.,* rows may appear more than once.
- Columns are sampled *without replacement* (it does not make sense to repeat the same data).
- The picture shows the decision boundary with 10 trees.

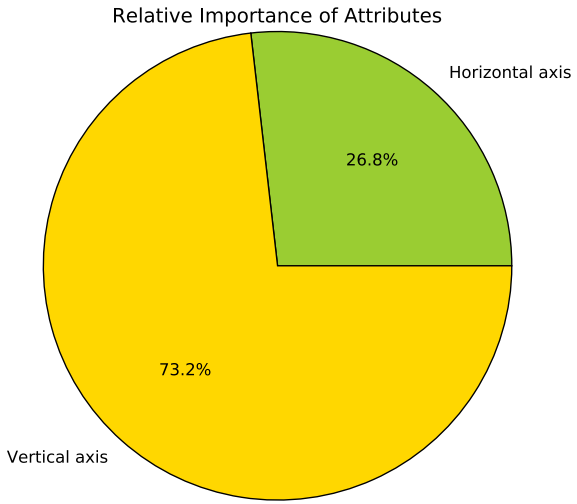
Random Forest

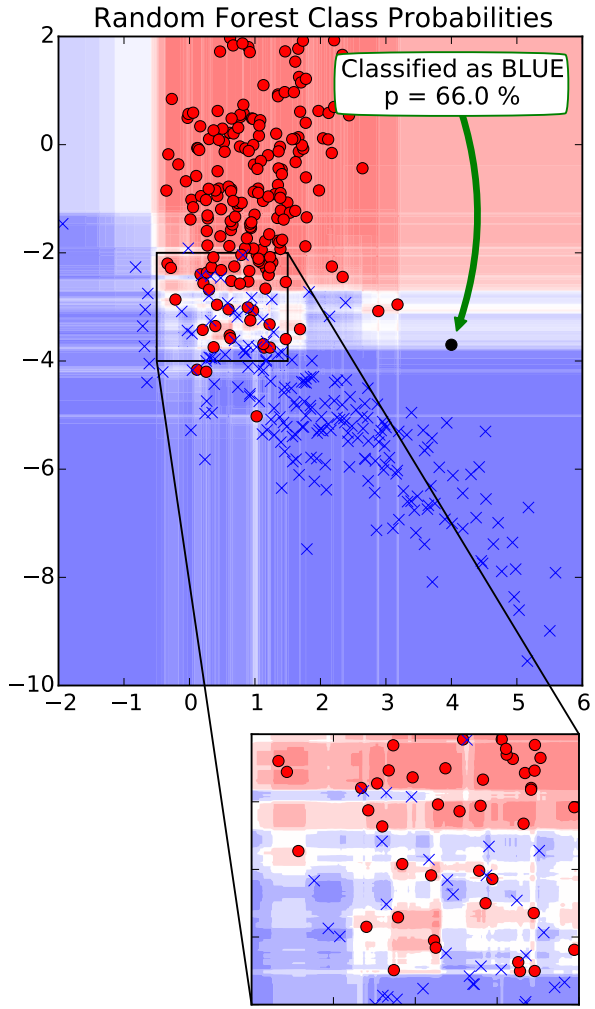
- After training many trees each with different samples and features, the RF predicts the class by taking the majority vote: what is the most frequent label.
- The number of trees varies from a few dozen to few thousand—default in Python is 10.
- The attached picture is produced by a 10-tree random forest.



Random Forest

- The collection of trees gives a natural way of estimating class probabilities: Just use the proportion of trees voting for each class.
- RF's also includes a method for assessing feature importances: randomly shuffle each feature at a time and test how much the accuracy drops.
- Loosing an important feature drops the accuracy a lot.
- Shuffling a non-important feature will not change the result much.





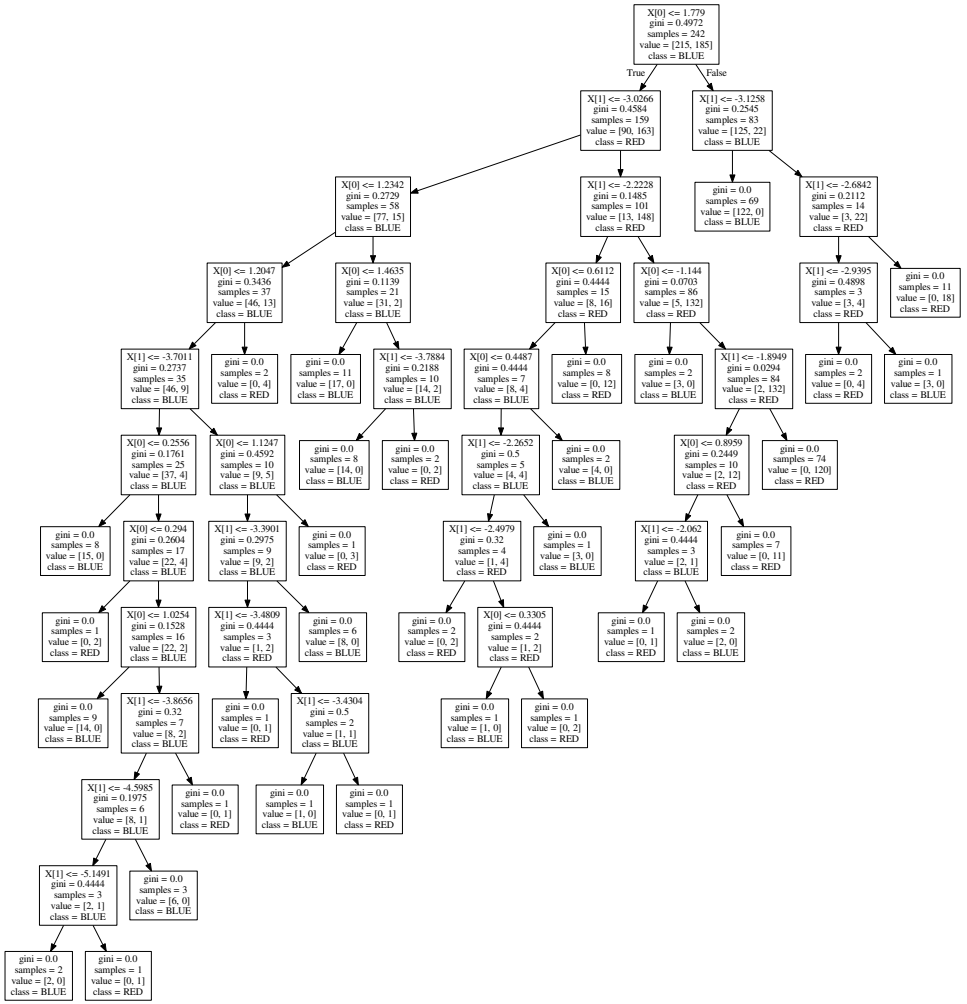
Random Forest in Scikit-Learn

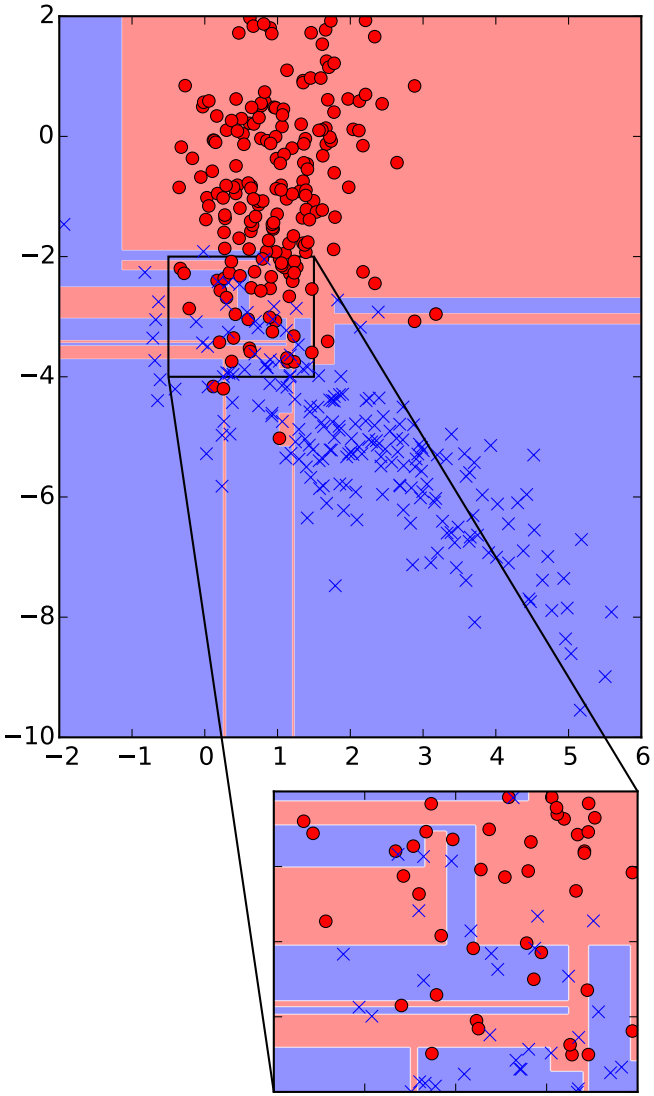
```
# Training code:
from sklearn.ensemble import
       RandomForestClassifier

clf = RandomForestClassifier()
clf.fit(X, y)
```

```
# Testing code:
>>>  clf.predict([0, -4])
array([ 0.])

>>>  clf.predict([0, -2])
array([ 1.])

>>> clf.predict_proba([[0, -4], [0, -2]])
array([[ 0.9,  0.1],
       [ 0.4,  0.6]])

>>> len(clf.estimators_)
10

>>> type(clf.estimators_[0])
sklearn.tree.tree.DecisionTreeClassifier
```

- The `RandomForestClassifier` class is used via the normal interface (`.fit()` and `.predict()`).
- Individual trees can be accessed, as well: `clf.estimators_` is a list of `DecisionTreeClassifier` objects.
- One of the 10 trained trees is visualized on the next slide (plot created using `sklearn.tree.export_graphviz`).

Decision Tree





Other Ensemble Classifiers

- Since the introduction of Random Forest by Breiman in 1993, several extensions have been proposed.
- As a group, these are called *ensemble methods*, because they all consist of an ensemble of weak classifiers,
- Most important ones are briefly summarized in the following slides.

AdaBoost Paradigm

- The **AdaBoost** paradigm was proposed by Freund and Schapire in 1995.
- Also in this case, the classifier consists of a collection of weak classifiers (most often decision trees).
- The difference to other ensemble methods is that trees are grown sequentially:
  1. Assign each sample a weight $w_1 \ldots, w_N$.
  2. Grow a tree minimizing the classification error weighted by w_n. That is, we emphasize the samples with large w_n.
  3. Append the new tree to our ensemble $\mathcal{E}$.
  4. Increase the weights for those samples that were incorrectly classified by $\mathcal{E}$.
  5. If not enough trees, then return to step 2.
- Implemented as `sklearn.ensemble.AdaBoostClassifier`.

Gradient Boosted Regression Trees

- **Gradient Boosted Regression Trees** were proposed by Friedman in 1999.
- Another boosting algorithm similar to AdaBoost.
- In this case, the training sequence is the following.
  1. Initialize the ensemble $\mathcal{E}$ by a single decision tree fit to the data.
  2. Train another tree for correcting the errors made by $\mathcal{E}$, *i.e.,* attempt to predict $\mathbf{y} - F_{\mathcal{E}}(\mathbf{X})$.
  3. Append the new tree to our ensemble $\mathcal{E}$.
  4. If not enough trees, then return to step 2.
- Implemented as `sklearn.ensemble.GradientBoostingClassifier`.

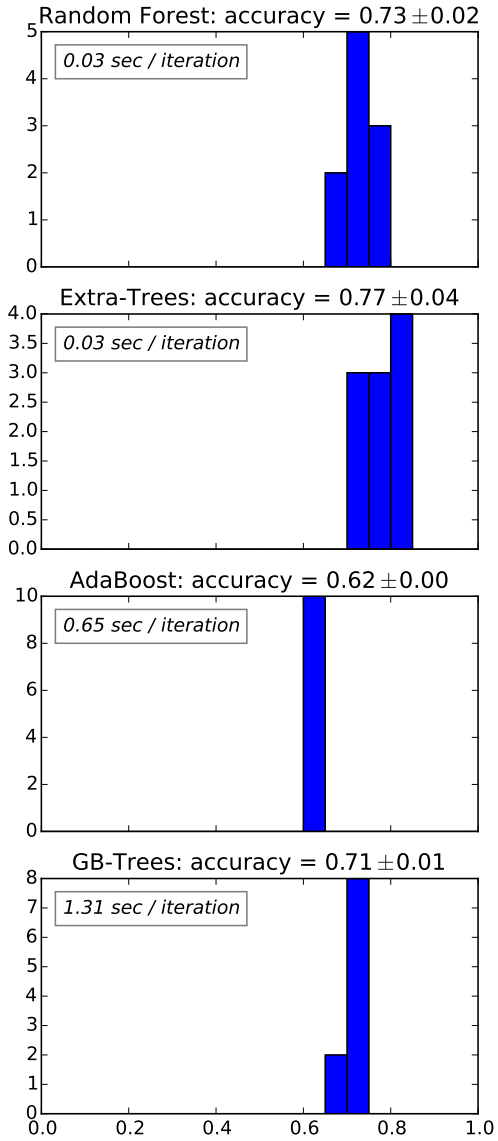
Extremely Randomized Trees

- **Extremely Randomized Trees** were proposed by Geurts *et al.* in 2006.
- The idea is to make the trees even weaker, but to compensate that with the large number of estimators in the ensemble.
- More specifically, not only the features and samples shown to the tree are randomized, but also the growing of individual trees is random.
- In particular the *split point i.e.,* the thresholds of comparisons like $X[1] <= -2.7025$ is randomized (see the graph of a decision tree at an earlier slide.
- Implemented as `sklearn.ensemble.ExtraTreesClassifier`.

Comparison of Methods

- The four ensemble methods were compared with the **arcene dataset**: **https://archive.ics.uci.edu/ml/datasets/Arcene**.
- For randomized algorithms, we iterate the experiment 100 times.

```
# Load Arcene data; 100+100 samples with dimension 10000:
# Mass spectrometer measurements from ovarian cancer patients
# and healthy controls.
X_train, y_train, X_test, y_test = load_arcene()

classifiers = [(RandomForestClassifier(), "Random Forest"),
               (ExtraTreesClassifier(), "Extra-Trees"),
               (AdaBoostClassifier(), "AdaBoost"),
               (GradientBoostingClassifier(), "GB-Trees")]

for clf, name in classifiers:
    clf.n_estimators = 100

    accuracies = []
    for iteration in range(100):

        clf.fit(X_train, y_train)

        y_hat = clf.predict(X_test)
        accuracy = accuracy_score(y_test, y_hat)
        accuracies.append(accuracy)
```



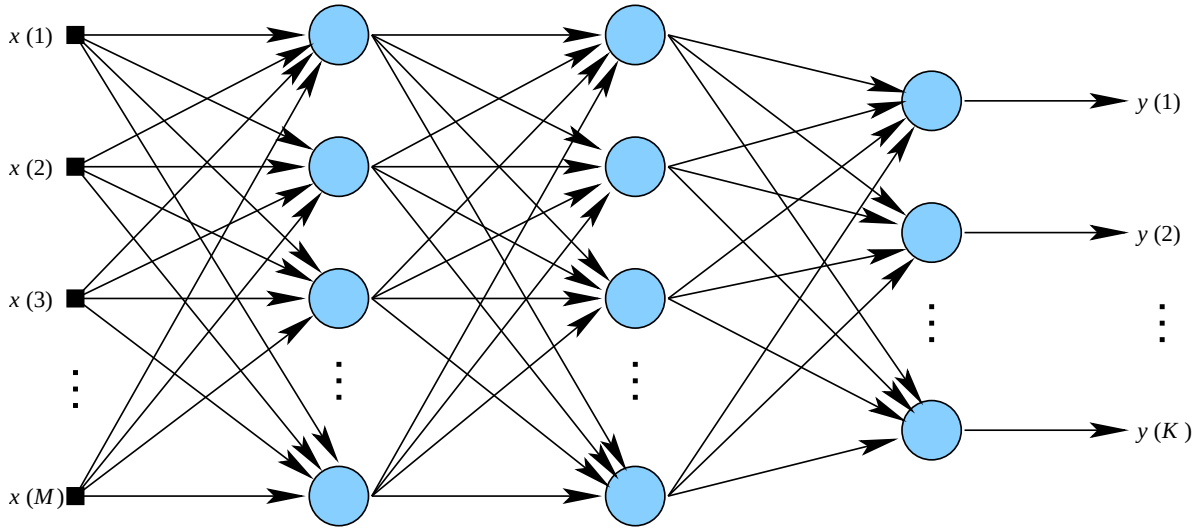
Comparison of Methods

- Let’s add a bunch of other classifiers into our for loop.
  - *1-Nearest Neighbor*: **0.88**
  - *Logistic Regression*: 0.84
  - *Linear SVM*: 0.83
  - *5-Nearest Neighbor*: 0.82
  - *LDA*: 0.79
  - *Extra-Trees*: 0.77 ± 0.04
  - *9-Nearest Neighbor*: 0.73
  - *Random Forest*: 0.73 ± 0.02
  - *GB-Trees*: 0.71 ± 0.01
  - *AdaBoost*: 0.62
  - *SVM with RBF kernel*: 0.56
- It seems that linear models and NN excel with this data.

NEURAL NETWORKS
(and deep learning)

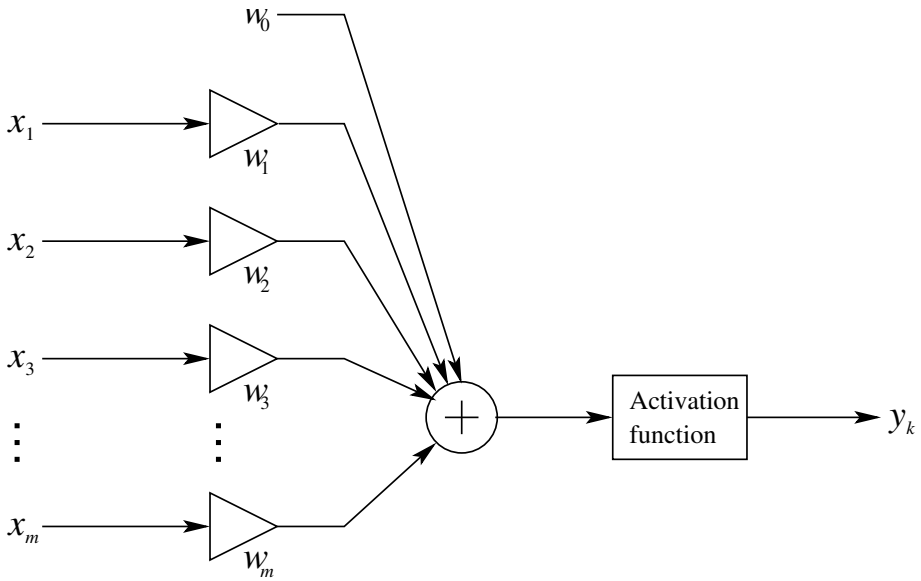
Traditional Neural Networks

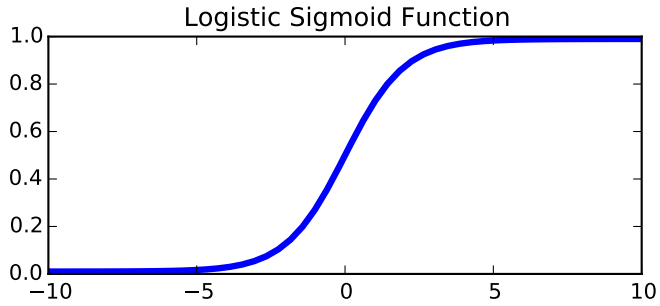
- Neural networks have been studied for decades.
- Traditional networks were *fully connected* (also called *dense*) networks consisting of typically 1-3 layers.
- Input dimensions were typically in the order of few hundred from a few dozen categories.
- Today, input may be 10k...100k variables from 1000 classes and network may have over 1000 layers.

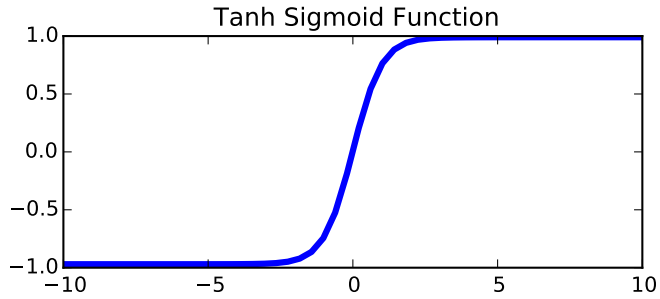


Traditional Neural Networks

- The neuron of a vanilla network is illustrated below.
- In essence, the neuron is a dot product between the inputs $\mathbf{x} = (1, x_1, \ldots, x_n)$ and weights $\mathbf{w} = (w_0, w_1, \ldots, w_n)$ followed by a nonlinearity, most often *logsig* or *tanh*.



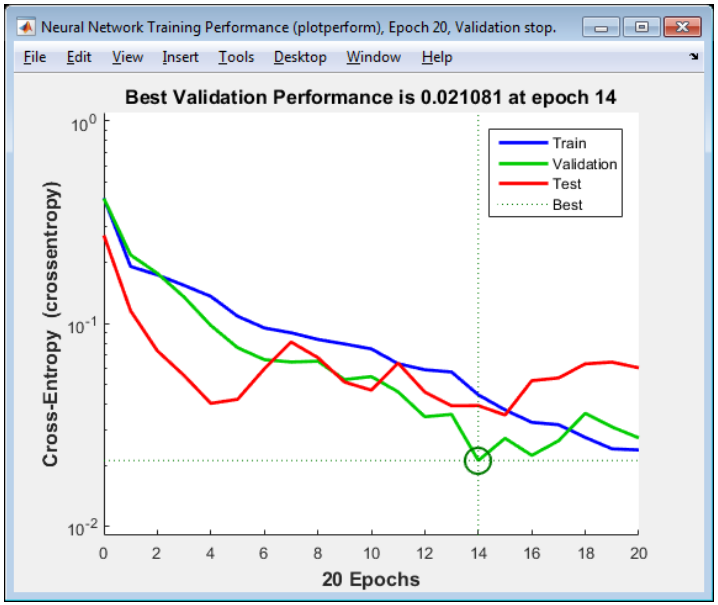


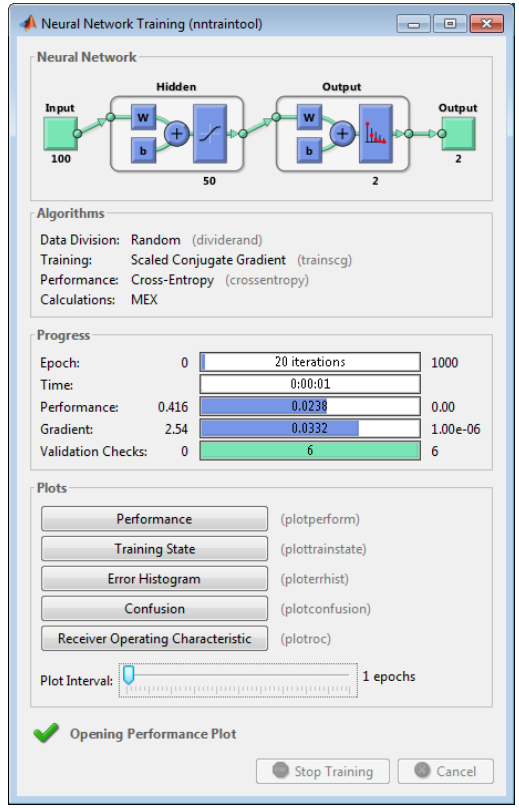


- In other words: this is *logistic regression* model, and the full net is just a stack of logreg models.

Training the Net

- Earlier, there was a lot of emphasis on training algorithms: *conjugate gradient, Levenberg-Marquardt, etc.*
- Today, people mostly use *stochastic gradient descent*.



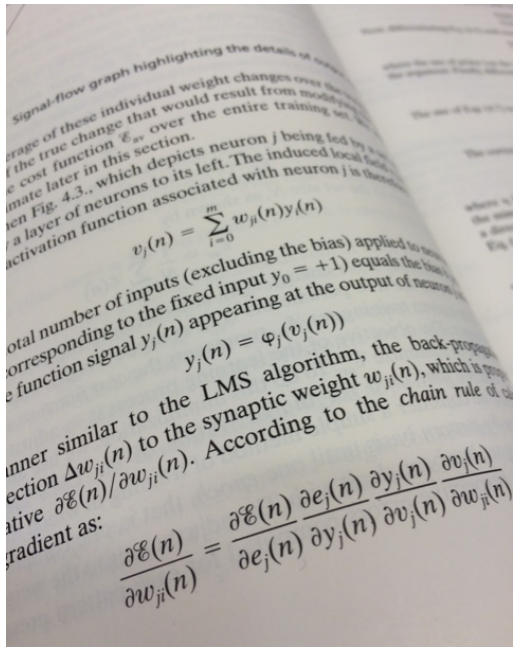


Backpropagation

- The network is trained by adjusting the weights according to the partial derivatives

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{E}}{\partial w_{ij}}$$

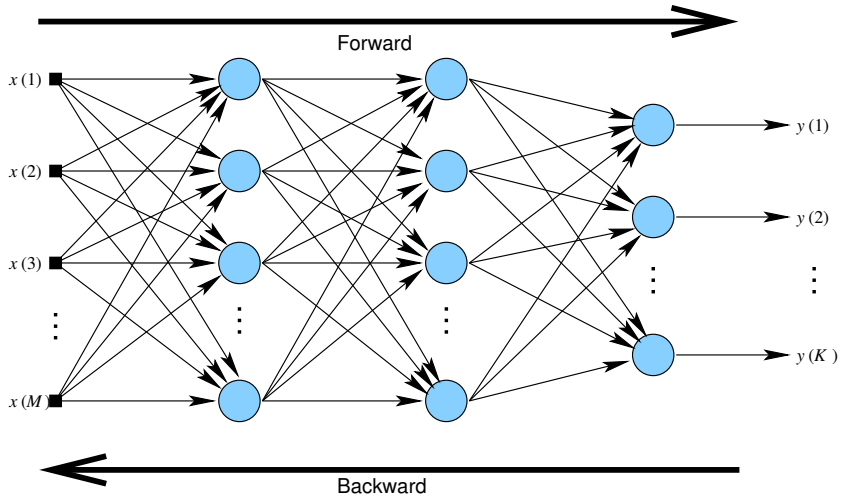
- In other words, the j^{th} weight of the i^{th} node steps towards the negative gradient with step size $\eta > 0$.
- In the 1990's the network structure was rather fixed, and the formulae would be derived by hand.
- Today, the same principle applies, but the exact form is computed symbolically.



Backpropagation in Haykin: Neural networks, 1999.

Forward and Backward

- Training has two passes: *forward pass* and *backward pass*.
- The forward pass feeds one (or more) samples to the net.
- The backward pass computes the (mean) error and propagates the gradients back adjusting the weights one at a time
- When all samples are shown to the net, one *epoch* has passed. Typically the network runs for thousands of epochs.



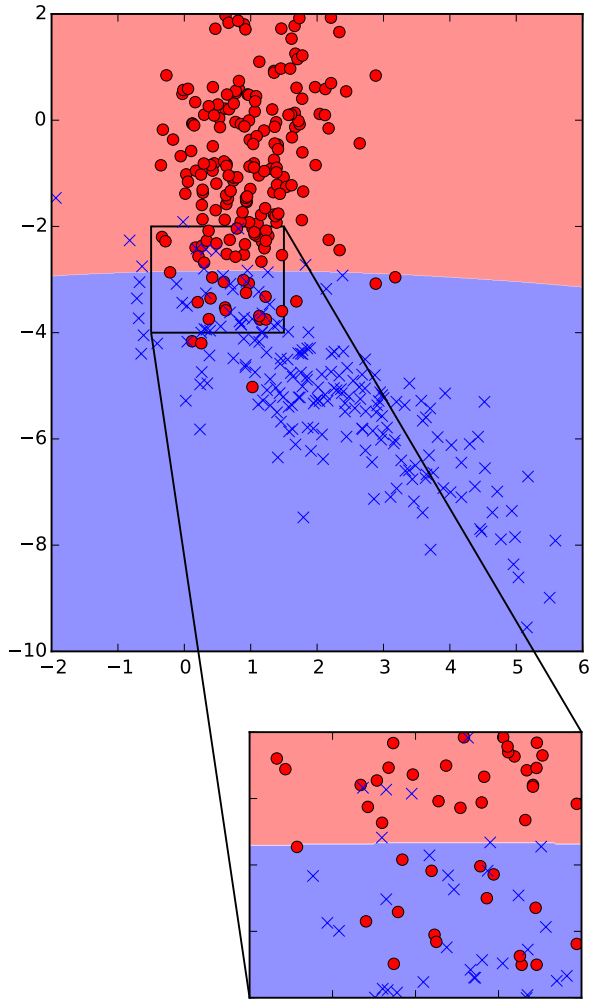
Neural Network Software

- Several packages exist:
  - **Matlab NN Toolbox**: Obsolete.
  - **Caffe**: C++ / CUDA with Python and Matlab interfaces
  - **Theano**: Python based CUDA engine; several front ends available: e.g., **Keras** and Lasagne.
  - **Torch**: Library implemented in Lua language; massively promoted by Facebook
  - **TensorFlow**: Google deep learning engine. Open sourced in 11 / 2015. Supported by **Keras**.
  - Others: VELES (Samsung), Minerva,. . .
  - Most use Nvidia **cuDNN** middle layer.
- The important ones:
  - **Caffe** is very fast and default in image recognition. Good Python and Matlab interface.
  - **Torch** has a large user base and lot of momentum from Facebook.
  - **Keras** is very flexible and readable full-python interface to Theano and Torch. **This is our choice for this course.**
  - **http://keras.io/** and **https://github.com/fchollet/keras**

Train a 2-layer Network with Keras

```
# Training code:
from keras.models import Sequential
from keras.layers.core import Dense, Activation

# First we initialize the model.
# "Sequential" has no loops. "Graph" allows loops etc.
clf = Sequential()

# Add layers one at the time. Each with 100 nodes.
clf.add(Dense(100, input_dim=2))
clf.add(Activation('sigmoid'))

clf.add(Dense(100))
clf.add(Activation('sigmoid'))

clf.add(Dense(1))
clf.add(Activation('sigmoid'))

# The code is compiled to CUDA or C++ (takes about 1.9 s)
clf.compile(loss='mean_squared_error', optimizer='sgd')
clf.fit(X, y, nb_epoch=20, batch_size=16) # takes about 1.6 s
```

```
# Testing code:
# Probabilities
>>> clf.predict(np.array([[1, -2], [-3, -5]]))
array([[ 0.50781795],
       [ 0.48059484]])
# Classes
>>> clf.predict(np.array([[1, -2], [-3, -5]])) > 0.5
array([[ True],
       [False]], dtype=bool)
```



Deep Learning

- The neural network research was rather silent after the rapid expansion in the 1990's.
- The hot topic of 2000's were, *e.g.,* the SVM and *big data*.
- However, at the end of the decade, neural networks started to gain popularity again: A group at Univ. Toronto led by Prof. Geoffrey Hinton studied unconventionally **deep** networks using *unsupervised* pretraining.
- He discovered that training of large networks was indeed possible with an unsupervised pretraining step that initializes the network weights in a layerwise manner.
- Another key factor to the success was the rapidly increased computational power brought by recent Graphics Processing Units (GPU's).

Unsupervised Pretraining

- There were two key problems why network depth did not increase beyond 2-3 layers:
  1. The error has huge **local minima** areas when the net becomes deep: Training gets stuck at one of them.
  2. The **gradient vanishes** at the bottom layers: The logistic activation function tends to decrease the gradient magnitude at each layer; eventually the gradient at the bottom layer is very small and they will not train at all.
- The former problem was corrected by unsupervised pretraining:
  - Train layered models that learned to *represent* the data (no class labels, no classification, just try to learn to reproduce the data).
  - Initialize the network with the weights of the unsupervised model and train in a supervised setting.
  - Common tools: *restricted Boltzmann machine* (RBM), *deep belief network* (DBN), *autoencoders*, etc.

Back to Supervised Training

- After the excitement of deep networks was triggered, the study of fully supervised approaches started as well (purely supervised training is more familiar, well explored and less scary angle of approach).
- A few key discoveries avoid the need for pretraining:
  - New activation functions that better preserve the gradient over layers; most importantly the Rectified Linear Unit[1]: $\mathrm{ReLU}(x) = \max(0, x)$.
  - Novel weight initialization techniques; *e.g.,* Glorot initialization (aka. Xavier initialization) adjusts the initial weight magnitudes layerwise[2].
  - Dropout regularization; avoid overfitting by injecting noise to the network[3]. Individual neurons are shut down at random in the training phase.

---

[1] Glorot, Bordes, and Bengio. "Deep sparse rectifier neural networks."

[2] Glorot and Bengio. "Understanding the difficulty of training deep feedforward neural networks."

[3] Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov. "Dropout: A simple way to prevent neural networks from overfitting."

Convolutional Layers

- In addition to the novel techniques for training, also new network architectures have been adopted.
- Most important of them is *convolutional layer*, which preserves also the topology of the input.
- Convolutional network was proposed already in 1989 but had a rather marginal role as long as image size was small (*e.g.,* 1990's MNIST dataset of size 28×28 as compared to current ImageNet benchmark of size 256×256).

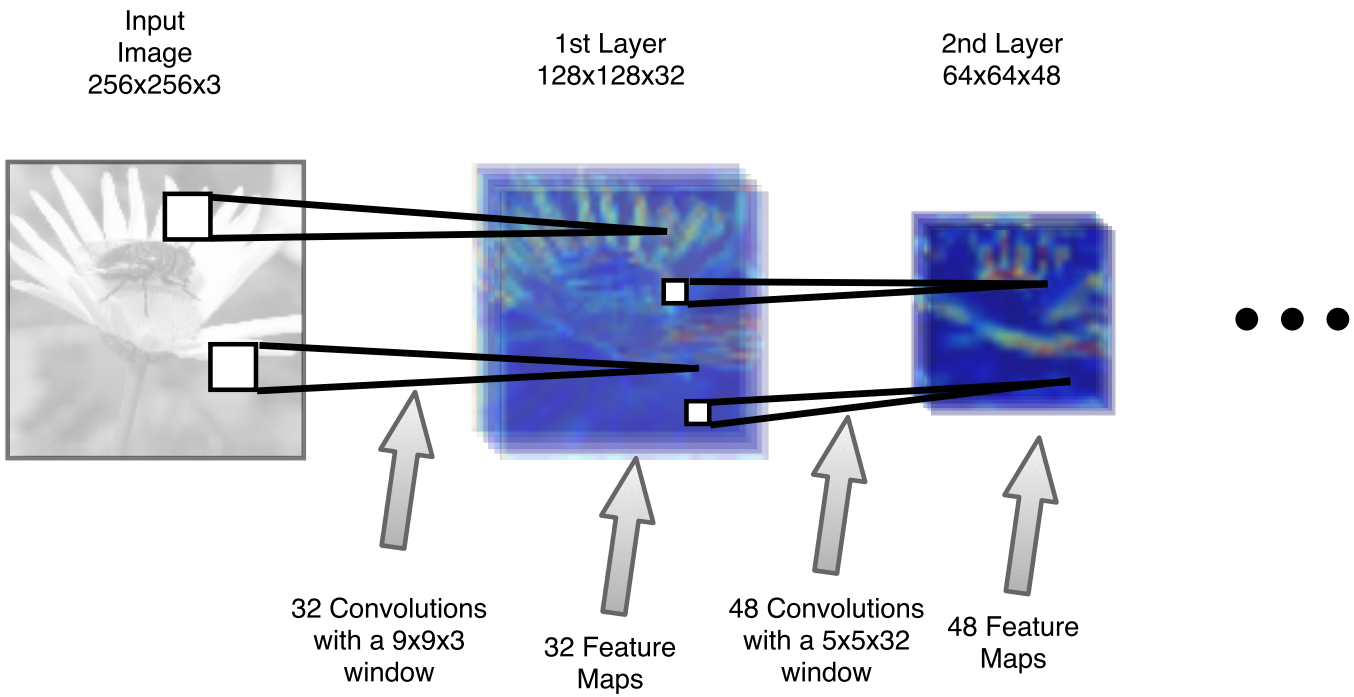
Convolutional Network

- The typical structure of a convolutional network repeats the following elements:

  convolution ⇒ **ReLU** ⇒ **subsampling**

  1. **Convolution** filters the input with a number of convolutional kernels. In the first layer these can be, *e.g.*, 9 × 9 × 3; *i.e.*, they see the local window from all RGB layers.
     - The results are called **feature maps**, and there are typically a few dozen of those.
  2. **ReLU** passes the feature maps through a pixelwise ReLU.
     - In numpy: `y = numpy.clip(x, a_min=0, a_max=np.inf)`.
  3. **Subsampling** shrinks the input dimensions by an integer factor.
     - Originally this was done by averaging each 2 × 2 block.
     - Nowadays, **maxpooling** is more common (take max of each 2 × 2 block).
     - Subsampling reduces the data size and improves spatial invariance.

Convolutional Network



Convolutional Network: Example

```
# Training code (modified from mnist_cnn.py at Keras examples)
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D

# We use the handwritten digit database "MNIST".
# 60000 training and 10000 test images of size 28x28
(X_train, y_train), (X_test, y_test) = mnist.load_data()

model = Sequential()

num_featmaps = 32
num_classes = 10
w, h = 3,3 # conv window size

model.add(Convolution2D(nb_filters, w, h,
                        input_shape=(1, 28, 28),
                        activation = 'relu'))

model.add(Convolution2D(num_featmaps, w, h), activation = 'relu')
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Vectorize the result for dense layers
model.add(Flatten())
model.add(Dense(128), activation = 'relu')
model.add(Dropout(0.5))

model.add(Dense(num_classes), activation = 'softmax')

model.compile(loss='categorical_crossentropy', optimizer='adadelta')
model.fit(X_train, Y_train, nb_epoch=100)
```