Recap: Decision Tree Training

• **Goal**
  - Select the query (=split) that decreases impurity the most
  \[
  \Delta t(s_j) = i(s_j) - P_L i(s_{jL}) - (1 - P_L) i(s_{jR})
  \]

• **Impurity measures**

  - **Entropy impurity (information gain):**
    \[
    i(s_j) = - \sum_k p(c_k | s_j) \log_2 p(c_k | s_j)
    \]
  
  - **Gini impurity:**
    \[
    i(s_j) = \sum_k p(c_k | s_j) p(c_k | s_j) = \frac{1}{2} \left[ 1 - \sum_k p^2(c_k | s_j) \right]
    \]

Recap: Randomized Decision Trees

• **Decision trees: main effort on finding good split**
  - Training runtime: \( O(DN^2 \log N) \)
  - This is what takes most effort in practice.
  - Especially cumbersome with many attributes (large \( D \)).

• **Idea: randomize attribute selection**
  - No longer look for globally optimal split.
  - Instead randomly use subset of \( K \) attributes on which to base the split.
  - Choose best splitting attribute e.g. by maximizing the information gain (= reducing entropy):
    \[
    \Delta E = \sum_{k=1}^K \frac{S_k}{N} \sum_{j=1}^N p_j \log_2(p_j)
    \]
Random Forests (Breiman 2001)

- General ensemble method
  - Idea: Create ensemble of many (very simple) trees.
  - Empirically very good results
    - Often as good as SVMs (and sometimes better)!
    - Often as good as Boosting (and sometimes better)!
- Standard decision trees: main effort on finding good split
  - Random Forests trees put very little effort in this.
  - CART algorithm with Gini coefficient, no pruning.
  - Each split is only made based on a random subset of the available attributes.
  - Trees are grown fully (important!).
- Main secret
  - Injecting the "right kind of randomness".

Random Forests – Algorithmic Goals

- Create many trees (50 – 1,000)
- Inject randomness into trees such that
  - Each tree has maximal strength
    - i.e. a fairly good model on its own
  - Each tree has minimum correlation with the other trees.
    - i.e. the errors tend to cancel out.
- Ensemble of trees votes for final result
  - Simple majority vote for category.
  - Alternative (Friedman)
    - Optimally reweight the trees via regularized regression (lasso).

Random Forests – Injecting Randomness (1)

- Bootstrap sampling process
  - Select a training set by choosing \( N \) times with replacement from all \( N \) available training examples.
  - On average, each tree is grown on only ~63% of the original training data.
  - Remaining 37% "out-of-bag" (OOB) data used for validation.
    - Provides ongoing assessment of model performance in the current tree.
    - Allows fitting to small data sets without explicitly holding back any data for testing.
    - Error estimate is unbiased and behaves as if we had an independent test sample of the same size as the training sample.

Random Forests – Injecting Randomness (2)

- Random attribute selection
  - For each node, randomly choose subset of \( K \) attributes on which the split is based (typically \( K = \sqrt{N} \)).
  - Faster training procedure
    - Need to test only few attributes.
    - Minimizes inter-tree dependence
      - Reduce correlation between different trees.
  - Each tree is grown to maximal size and is left unpruned
    - Trees are deliberately overfit
    - Become some form of nearest-neighbor predictor.

Bet You’re Asking…

How can this possibly ever work???
Different trees induce different partitions on the data.

By combining them, we obtain a finer subdivision of the feature space...

...which at the same time also better reflects the uncertainty due to the bootstrapped sampling.

Summary: Random Forests

- Properties
  - Very simple algorithm.
  - Resistant to overfitting – generalizes well to new data.
  - Faster training
  - Extensions available for clustering, distance learning, etc.

- Limitations
  - Memory consumption
    - Decision tree construction uses much more memory.
  - Well-suited for problems with little training data
    - Little performance gain when training data is really large.

Today's Topic

Deep Learning
Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

A Brief History of Neural Networks

1957  Rosenblatt invents the Perceptron
- And a cool learning algorithm: “Perceptron Learning”
- Hardware implementation “Mark I Perceptron” for 20×20 pixel image analysis

1969  Minsky & Papert
- Showed that (single-layer) Perceptrons cannot solve all problems.
- This was misunderstood by many that they were worthless.

1980s Resurgence of Neural Networks
- Some notable successes with multi-layer perceptrons.
- Backpropagation learning algorithm
- But they are hard to train, tend to overfit, and have unintuitive parameters.
- So, the excitement fades again…

1995+ Interest shifts to other learning methods
- Notably Support Vector Machines
- Machine Learning becomes a discipline of its own.
A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron
1969 Minsky & Papert
1980s Resurgence of Neural Networks
1995+ Interest shifts to other learning methods
  - Notably Support Vector Machines
  - Machine Learning becomes a discipline of its own.
  - The general public and the press still love Neural Networks.

I'm doing Machine Learning.
So, you're using Neural Networks?
Actually...

Come on. Get real!
Are you using Neural Networks?

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron
1969 Minsky & Papert
1980s Resurgence of Neural Networks
1995+ Interest shifts to other learning methods
2005+ Gradual progress
  - Better understanding how to successfully train deep networks
  - Availability of large datasets and powerful GPUs
  - Still largely under the radar for many disciplines applying ML

Perceptrons (Rosenblatt 1957)

- Standard Perceptron
  - Output layer
  - Weights
  - Input layer
  - Hand-designed features based on common sense
- Outputs
  - Linear outputs
    \[ y(x) = w^T x + w_0 \]
  - Logistic outputs
    \[ y(x) = \sigma(w^T x + w_0) \]
- Learning = Determining the weights \( w \)

Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

Extension: Multi-Class Networks

- One output node per class
  - Output layer
  - Weights
  - Input layer
- Outputs
  - Linear outputs
    \[ y_k(x) = \sum_{i=0}^{d} W_{ki} x_i \]
  - Logistic outputs
    \[ y_k(x) = \sigma \left( \sum_{i=0}^{d} W_{ki} x_i \right) \]

⇒ Can be used to do multidimensional linear regression or multiclass classification.
Extension: Non-Linear Basis Functions

- Straightforward generalization
  \[ y_k(x) = \sum_{i=0}^{d} W_{ki} \phi_i(x) \]
  \[ y_k(x) = \sigma \left( \sum_{i=0}^{d} W_{ki} \phi_i(x) \right) \]
  Output layer
  Weights
  Feature layer
  Mapping (fixed)
  Input layer

- Outputs
  - Linear outputs
  - Logistic outputs
  \[ y_k(x) = \sum_{i=0}^{d} W_{ki} \phi_i(x) \]
  \[ y_k(x) = \sigma \left( \sum_{i=0}^{d} W_{ki} \phi_i(x) \right) \]

Remarks
- Perceptrons are generalized linear discriminants!
- Everything we know about the latter can also be applied here.
- Note: feature functions \( \phi(x) \) are kept fixed, not learned!

Perceptron Learning

- Very simple algorithm
- Process the training cases in some permutation
  - If the output unit is correct, leave the weights alone.
  - If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
  - If the output unit incorrectly outputs a one, subtract the input vector from the weight vector.
- This is guaranteed to converge to a correct solution if such a solution exists.

Loss Functions

- We can now also apply other loss functions
  - L2 loss
    \[ L(t, y(x)) = \sum_n (y(x_n) - t_n)^2 \] \( \Rightarrow \) Least-squares regression
  - L1 loss:
    \[ L(t, y(x)) = \sum_n |y(x_n) - t_n| \] \( \Rightarrow \) Median regression
  - Cross-entropy loss
    \[ L(t, y(x)) = -\sum_n \left( t_n \ln y_n + (1 - t_n) \ln (1 - y_n) \right) \] \( \Rightarrow \) Logistic regression
  - Hinge loss
    \[ L(t, y(x)) = \sum_n \left[ 1 - t_n y_n \right]_+ \] \( \Rightarrow \) SVM classification
  - Softmax loss
    \[ L(t, y(x)) = -\sum_n \sum_k \left( (t_n = k) \ln y_k(x_n) \right) \] \( \Rightarrow \) Multi-class probabilistic classification
Regularization

- In addition, we can apply regularizers
  - E.g., an L2 regularizer
    \[ E(w) = \sum L(t_n, y(x_n; w)) + \lambda ||w||^2 \]
  - This is known as weight decay in Neural Networks.
  - We can also apply other regularizers, e.g. L1 \(\rightarrow\) sparsity
  - Since Neural Networks often have many parameters, regularization becomes very important in practice.
  - We will see more complex regularization techniques later on...

Limitations of Perceptrons

- What makes the task difficult?
  - Perceptrons with fixed, hand-coded input features can model any separable function perfectly...
    - ...given the right input features.
  - For some tasks this requires an exponential number of input features.
    - E.g., by enumerating all possible binary input vectors as separate feature units (similar to a look-up table).
    - But this approach won’t generalize to unseen test cases!
  - It is the feature design that solves the task!
    - Once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn.
      - Classic example: XOR function.

Wait...

- Didn’t we just say that...
  - Perceptrons correspond to generalized linear discriminants
  - And Perceptrons are very limited...
  - Doesn’t this mean that what we have been doing so far in this lecture has the same problems???
  - Yes, this is the case.
    - A linear classifier cannot solve certain problems (e.g., XOR).
    - However, with a non-linear classifier based on the right kind of features, the problem becomes solvable.
      - So far, we have solved such problems by hand-designing good features \(\phi\) and kernels \(\phi'\).
      - Can we also learn such feature representations?

Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

Multi-Layer Perceptrons

- Adding more layers
  - Output
    \[ y_L(x) = g^{(2)} \left( \sum_{i=1}^{h} W^{(2)}_{k_i} g^{(1)} \left( \sum_{j=0}^{d} W^{(1)}_{j} x_j \right) \right) \]
  - Activation functions \(g^{(1)}\):
    - For example: \(g^{(1)}(a) = \sigma(a), g^{(2)}(a) = a\)
  - The hidden layer can have an arbitrary number of nodes
    - There can also be multiple hidden layers.
  - Universal approximators
    - A 2-layer network (1 hidden layer) can approximate any continuous function of a compact domain arbitrarily well!
      (assuming sufficient hidden nodes)
Learning with Hidden Units

- Networks without hidden units are very limited in what they can learn
  - More layers of linear units do not help \( \Rightarrow \) still linear
  - Fixed output non-linearities are not enough.
- We need multiple layers of adaptive non-linear hidden units. But how can we train such nets?
  - Need an efficient way of adapting all weights, not just the last layer.
  - Learning the weights to the hidden units – learning features
  - This is difficult, because nobody tells us what the hidden units should do.

  \( \Rightarrow \) Main challenge in deep learning.

Gradient Descent

- Two main steps
  1. Computing the gradients for each weight \( \text{today} \)
  2. Adjusting the weights in the direction of the gradient \( \text{next lecture} \)

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation
  - Compute the gradients for each variable analytically.
  - What is the problem when doing this?

Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions
  \[
  \Delta z = \frac{dz}{dz} \Delta y = \frac{dy}{dy} \Delta x = \frac{dx}{dx} \Delta z
  \]
Excurision: Chain Rule of Differentiation

- Multi-dimensional case: Total derivative

\[
\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x} + \ldots
\]

\[
= \sum_{i=1}^{k} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}
\]

⇒ Need to sum over all paths that lead to the target variable \( x \).

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation

\[
y_1(x), y_2(x), y_k(x)
\]

\[
W_{i}^{(1)}, \ldots, W_{i}^{(l)}
\]

\[
z_0 = 1, \ldots, z_{l}, x_{l}
\]

⇒ Compute the gradients for each variable analytically.

⇒ What is the problem when doing this?

⇒ With increasing depth, there will be exponentially many paths!

⇒ Infeasible to compute this way.

Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

Obtaining the Gradients

- Approach 2: Numerical Differentiation

\[
y_1(x), y_2(x), y_k(x)
\]

\[
W_{i}^{(1)}, \ldots, W_{i}^{(l)}
\]

\[
z_0 = 1, \ldots, z_{l}, x_{l}
\]

⇒ Given the current state \( W^{(l)} \), we can evaluate \( E(W^{(l)}) \).

⇒ Idea: Make small changes to \( W^{(l)} \) and accept those that improve \( E(W^{(l)}) \).

⇒ Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset!

Obtaining the Gradients

- Approach 3: Incremental Analytical Differentiation

\[
y_1(x), y_2(x), y_k(x)
\]

\[
W_{i}^{(1)}, \ldots, W_{i}^{(l)}
\]

\[
z_0 = 1, \ldots, z_{l}, x_{l}
\]

⇒ Idea: Compute the gradients layer by layer.

⇒ Each layer below builds upon the results of the layer above.

⇒ The gradient is propagated backwards through the layers.

⇒ Backpropagation algorithm
### Backpropagation Algorithm

**Core steps**

1. **Forward Pass**
   
   \[ E = \frac{1}{2} \sum_{j \in \text{output}} (t_j - y_j)^2 \]
   
   \[ \frac{\partial E}{\partial y_j} = -(t_j - y_j) \]

2. **Compute error derivatives in each hidden layer from error derivatives in the layer above.**

3. **Use error derivatives w.r.t. activities to get error derivatives w.r.t. the incoming weights.**

### Efficient propagation scheme

- Propagate back the gradient from layer \( k \) and multiply with \( y^{(k-1)} \).

### Summary: MLP Backpropagation

- **Forward Pass**
  
  \[
  \begin{align*}
  y^{(0)} &= x \\
  \text{for } & k = 1, \ldots, l \text{ do} \\
  z^{(k)} &= W^{(k)} y^{(k-1)} \\
  y^{(k)} &= g_k(z^{(k)}) \\
  \text{endfor} \\
  y &= y^{(l)} \\
  E &= L(t, y) + \lambda \Omega(W)
  \end{align*}
  \]

- **Backward Pass**
  
  \[
  \begin{align*}
  h &\leftarrow \frac{\partial E}{\partial y} = \frac{\partial E}{\partial y_j} L(t, y) + \lambda \frac{\partial \Omega}{\partial h} \\
  \text{for } & k = l, l-1, \ldots, 1 \text{ do} \\
  h &\leftarrow \frac{\partial E}{\partial W^{(k)}} = h \circ g'(y^{(k)}) \\
  &\quad + \lambda \frac{\partial \Omega}{\partial (W^{(k)})} \\
  &\quad \text{endfor} \\
  y &= y^{(l)} \\
  E &= L(t, y) + \lambda \Omega(W)
  \end{align*}
  \]

- **Notes**
  
  - For efficiency, an entire batch of data \( X \) is processed at once.
  - \( \circ \) denotes the element-wise product
Analysis: Backpropagation

- Backpropagation is the key to make deep NNs tractable
  - However...

- The Backprop algorithm given here is specific to MLPs
  - It does not work with more complex architectures, e.g. skip connections or recurrent networks!
  - Whenever a new connection function induces a different functional form of the chain rule, you have to derive a new Backprop algorithm for it.
  - Tedious...

- Let’s analyze Backprop in more detail
  - This will lead us to a more flexible algorithm formulation
  - Next lecture...

References and Further Reading

- More information on Neural Networks can be found in Chapters 6 and 7 of the Goodfellow & Bengio book