**Recap: Decision Tree Training**

- **Goal**
  - Select the query (=split) that decreases impurity the most
  \[
  \Delta i(N) = i(N) - P_L i(N_L) - (1 - P_L) i(N_R)
  \]

- **Impurity measures**
  - Entropy impurity (information gain):
    \[
    i(N) = - \sum_j p(C_j|N) \log_2 p(C_j|N)
    \]
  - Gini impurity:
    \[
    i(N) = \sum_{i \neq j} p(C_i|N)p(C_j|N) = \frac{1}{2} \left[ 1 - \sum_j p^2(C_j|N) \right]
    \]

**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):
    \[
    \Delta E = \sum_{k=1}^K \left[ \hat{S}^k \sum_{j=1}^N p_j \log_2(p_j) \right]
    \]

**Recap: Ensemble Combination**

- **Ensemble combination**
  - Tree leaves \((i, x)\) store posterior probabilities of the target classes.
  \[
  p_{i,x}(C|x)
  \]
  - Combine the output of several trees by averaging their posteriors (Bayesian model combination)
  \[
  p(C|x) = \frac{1}{L} \sum_{l=1}^L p_{i,x}(C|x)
  \]

**Recap: Random Forests (Breiman 2001)**

- **General ensemble method**
  - Idea: Create ensemble of many (50 - 1,000) trees.

- **Injecting randomness**
  - Bootstrap sampling process
    - On average only 63% of training examples used for building the tree
    - Remaining 37% out-of-bag samples used for validation.
  - Random attribute selection
    - Randomly choose subset of \(K\) attributes to select from at each node.
    - Faster training procedure.

- **Simple majority vote for tree combination**

- **Empirically very good results**
  - Often as good as SVMs (and sometimes better)?
  - Often as good as Boosting (and sometimes better)
Recap: Ferns

• Ferns
  - Ferns are semi-naïve Bayes classifiers.
  - They assume independence between sets of features (between the ferns)...
  - ...and enumerate all possible outcomes inside each set.

• Interpretation
  - Combine the tests \( f_1, \ldots, f_{N_f} \) into a binary number.
  - Update the “fern leaf” corresponding to that number.

\[
\begin{align*}
0 & \rightarrow \text{Update leaf 100}_2 = 4 \\
1 & \\
\end{align*}
\]

Recap: Ferns (Semi-Naïve Bayes Classifiers)

• Ferns
  - A fern \( F \) is defined as a set of \( S \) binary features \( \{f_1, \ldots, f_{N_f}\} \).
  - \( M \): number of ferns, \( N_f = S \cdot M \).
  - This represents a compromise:
    - Model with \( M \cdot 2^S \) parameters ("Semi-Naïve").
    - Flexible solution that allows complexity/performance tuning.

Today’s Topic

Deep Learning

Topics of This Lecture

• Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits

• Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units

• Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation

Perceptrons (Rosenblatt 1957)

• Standard Perceptron
  - Input Layer
    - Hand-designed features based on common sense
  - Outputs
    - Linear outputs: \( y(x) = w^\top x + w_0 \)
    - Logistic outputs: \( y(x) = \sigma(w^\top x + w_0) \)
  - Learning = Determining the weights \( w \)

Extension: Multi-Class Networks

• One output node per class
  - Output layer
  - Weighted inputs
  - Outputs
    - Linear outputs: \( y_k(x) = \sum_{i=0}^d W_{ki} x_i \)
    - Logistic outputs: \( y_k(x) = \sigma\left( \sum_{i=0}^d W_{ki} x_i \right) \)

\( \Rightarrow \) Can be used to do multidimensional linear regression or multiclass classification.

Slide adapted from Stefan Roth
Extension: Non-Linear Basis Functions

- Straightforward generalization
  \[ y_i(x) = \sum_{k=0}^{d} W_{ik}\phi(x_i) \]
  \[ y(x) = \sigma\left( \sum_{i=0}^{d} W_{ik}\phi(x_i) \right) \]

- Outputs
  - Linear outputs
  - Logistic outputs
  \[ y_l(x) = \sum_{k=0}^{d} W_{lk}\phi(x_i) \]
  \[ y_l(x) = \sigma\left( \sum_{i=0}^{d} W_{lk}\phi(x_i) \right) \]

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.

Let’s analyze this 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.
- Translation
  \[ w_k^{(\tau+1)} = w_k^{(\tau)} - \eta (y_k(x_n; W) - t_{kn}) \phi_j(x_n) \]

- This is the Delta rule, a.k.a. LMS rule!
- Perceptron Learning corresponds to 1st-order (stochastic) Gradient Descent of a quadratic error function!

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
    \[ L(t, y(x)) = -\sum_{n} (t_n \ln y_n + (1 - t_n) \ln (1 - y_n)) \] \( \Rightarrow \) Logistic regression
  - Hinge loss
    \[ L(t, y(x)) = \sum_{n} \max(0, 1 - t_n y_n) \] \( \Rightarrow \) SVM classification
  - Softmax loss
    \[ L(t, y(x)) = -\sum_{n} \sum_{k} \left( t_n = k \ln \frac{\exp(y_k(x_n))}{\sum_k \exp(y_k(x_n))} \right) \] \( \Rightarrow \) Multi-class probabilistic classification
**Regularization**

- In addition, we can apply regularizers
  - E.g., an L2 regularizer
    \[ E(w) = \sum_{i} L(t_i, y(X_i; w)) + \lambda \| w \|^2 \]
  - 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.
  - More complex regularization techniques exist (and are an active field of research)

**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!
    \[ \Rightarrow \text{It is the feature design that solves the task!} \]
    - 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 \( \phi \cdot \).
  - Can we also learn such feature representations?

**Topics of This Lecture**

- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naïve analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation

**Multi-Layer Perceptrons**

- Adding more layers
  - Output
    \[ y_k(x) = g^{(2)} \left( \sum_{i=0}^{h} W_{ki}^{(2)} g^{(1)} \left( \sum_{j=0}^{d} W_{ij}^{(1)} x_j \right) \right) \]
  - Activation functions \( g^{(k)} \)
    - For example: \( g^{(2)}(a) = \sigma(a) \), \( g^{(1)}(a) = \tanh(a) \)
  - 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 ⇒ 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.

Learning with Hidden Units

- How can we train multi-layer networks efficiently?
  - Need an efficient way of adapting all weights, not just the last layer.

- Idea: Gradient Descent
  - Set up an error function
    \[ E(W) = \sum_{n} L(y_n, y_{n}(x_n; W)) + \lambda \Omega(W) \]
    with a loss \( L(\cdot) \) and a regularizer \( \Omega(\cdot) \).
  - E.g., \( L(y_n; x_n; W) = \sum_{n} (y_n(x_n; W) - t_n)^2 \)
    \( L_2 \) loss
    \[ \Omega(W) = \| W \|^2 \]
    \( L_2 \) regularizer ("weight decay")
  ⇒ Update each weight \( W_{ij}^{(k)} \) in the direction of the gradient \( \nabla E(W) \)

Gradient Descent

- Two main steps
  1. Computing the gradients for each weight  
  2. Adjusting the weights in the direction of the gradient

Topics of This Lecture

- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits

- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units

- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation

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
  \[
  z \xrightarrow{\mathcal{L}} y \xrightarrow{\mathcal{L}} x
  \]
  \[ \Delta z = \frac{dz}{dy} \Delta y + \frac{dz}{dx} \Delta x \]
  \[ \Delta y = \frac{dy}{dz} \Delta z + \frac{dy}{dx} \Delta z \]
  \[ \Delta z = \frac{dz}{dx} + \frac{dz}{dy} \frac{dy}{dx} \Delta x \]
**Excursion: Chain Rule of Differentiation**

- Multi-dimensional case: Total derivative

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

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

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

**Obtaining the Gradients**

- **Approach 1: Naive Analytical Differentiation**

- **Approach 2: Numerical Differentiation**

- **Approach 3: Incremental Analytical Differentiation**

**Topics of This Lecture**

- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits

- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation

- Obtaining the Gradients
  - 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. Convert the discrepancy between each output and its target value into an error derivative.

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

   \[ \frac{\partial E}{\partial y_j} = -(t_j - y_j) \]

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

   \[ \frac{\partial E}{\partial y} \rightarrow \frac{\partial E}{\partial w_{ij}} \]

**Notation**

- \( y^j \) Output of layer \( j \)
- \( z^j \) Input of layer \( j \)

Connections:

- \( z^j = \sum_i w_{ij} y_i \)

**Efficient propagation scheme**

- \( y^j \) is already known from forward pass! (Dynamic Programming)

  Propagate back the gradient from layer \( j \) and multiply with \( y^j \).
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

Ian Goodfellow, Aaron Courville, Yoshua Bengio
Deep Learning
MIT Press, in preparation

https://goodfeli.github.io/dlbook/