Neural Networks is a broad field. We'll focus on a class of supervised, feed-forward neural networks called Multilayer Perception (MLP).

**Agenda**
1. Functional form of MLP
2. MLP as Universal Approximator
3. Training via Back-propagation
4. Practical issues: regularization, speed-up

Suppose a tough dataset (not linearly separable). What classifier to use?
- Perceptron, logistic Regression, simple Gaussian classifiers may not work.
- Good choices:
  - SVM with nonlinear kernels
  - KNN
  - MLP
  - Boosting with stumps

To handle tough datasets, we must expand the expressiveness of our model:

1. \( \hat{y} = f \left( \sum_{i=1}^{P} w_i x_i \right) \) ← Linear model (e.g., perceptron, logistic regression)

   nonlinear function like sign/sigmoid/etc.

2. \( \hat{y} = f \left( \sum_{j=1}^{M} w_j \phi_j(x) \right) \) ← Project x into M nonlinear space (similar in spirit to kernels in SVM)

   Nonlinear basis functions

3. MLP: Parameterize \( \phi_j \) - allows it to be trained.
A Functional form of MLP

\[ z_j = h \left( \sum_{i=1}^{D} w_{ji}^u x_i \right) = h \left( a_j^u \right) \]

\[ y_k = \sigma \left( \sum_{j=1}^{M} w_{kj}^u z_j \right) = \sigma \left( \sum_{j=1}^{M} w_{kj}^u h \left( \sum_{i=1}^{D} w_{ji}^u x_i \right) \right) \]

\[ h: \text{nonlinear, differentiable function (e.g., sigmoid, tanh)} \]

\[ \Gamma: \text{depends on task} \]
- regression: \( y_k = a_k \) \( \sigma = \text{identity} \)
- binary classification: \( y = \sigma \left( a \right) \)
- multiclass: softmax
\[ y_k = \frac{\exp(a_k)}{\sum_k \exp(a_k)} \]

Remarks:

1. Naming - MLP because it's like a cascade of perceptrons (but the layers aren't)
   - 1 layer, 2 layer, or 3 layer net, technically perceptrons

2. View of first hidden layer as feature extractor.
   (In fact, if we make \( h \) linear, we can get PCA.)

3. Many variations possible:
   - multiple hidden layers
   - \# of nodes per layer variable
   - different \( h \)
   - skip layer, sparse connection (as long as it's feedforward)

4. Bias term should be included for each node.
1. Two hidden layers are sufficient to create classification boundaries of arbitrary shape. Boundaries are piecewise linear, but smooth boundaries can be approximated by enough units.

2. One hidden layer can represent nonconvex, disjoint regions.

3. Capacity is related to both depth and size of hidden layers.
Training MLPs

Error function:

\[ \text{MSE: } E(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \]

Cross-entropy: Assume class + \( y = 1 \), class - \( y = 0 \).

Let output be modeled by sigmoid: \( \hat{y} = \sigma(x) = \frac{1}{1 + \exp(-x)} \)

\[ P(y_{\text{true}} | \mathbf{w}) = \hat{y}^y (1 - \hat{y})^{1 - y} \]

\[ E(\mathbf{w}) = -\log P(y_{\text{true}} | \mathbf{w}) = -\frac{1}{n} \sum_{i=1}^{n} (y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)) \]

Gradient Descent optimization:

- Initialize \( \mathbf{w} \)
- While (! convergence)
  1. Compute gradient \( \nabla E(\mathbf{w}) \)
  2. Update \( \mathbf{w} = \mathbf{w} - \eta \nabla E(\mathbf{w}) \) \( \eta \) stepsize/learning rate.

\( \nabla E(\mathbf{w}) \) is computed by "Back-propagation" procedure.

Sometimes the entire gradient descent algo is called "Back-propagation" too.
Calculating derivatives via Backpropagation

Recall: \( \hat{y}_k = \sigma \left( \sum_{j=1}^{N_m} w_{kj}^{(e)} h \left( \sum_{i=1}^{D} w_{ji}^{(u)} x_i \right) \right) \)

\[ a_j \]
\[ z_j \]

Assume: \( E(w) = \frac{1}{2} \sum \frac{1}{n} \sum \left( y_k^{(o)} - y_k^{(d)} \right)^2 \Rightarrow E_n(w) = \frac{1}{2} \sum \left( \hat{y}_k - y_k \right)^2 \)

\( \sigma \): identity \( \hat{y}_k = \sigma (a_k) = a_k \)

Output layer

\[ \frac{dE_n}{dw_{kj}^{(e)}} = \frac{dE_n}{dak} \frac{dak}{dw_{kj}^{(e)}} = \frac{dE_n}{dak} \frac{\sum_j \frac{dak}{dz_j} \frac{dz_j}{dw_{kj}^{(e)}} \hat{y}_j}{\sum_j \frac{dak}{dz_j} \frac{dz_j}{dw_{kj}^{(e)}} \hat{y}_j} = \frac{dE_n}{dak} \frac{\sum_j \frac{dak}{dz_j} \hat{y}_j}{\sum_j \frac{dak}{dz_j} \hat{y}_j} \]

"error" of node \( k \)

\[ \frac{dE_n}{dw_{kj}^{(e)}} = \sum_j \frac{dak}{dz_j} \hat{y}_j \]

What does this mean?

- If \( \hat{z}_j > 0 \) & \( \hat{z}_j \) is on: \( w_{kj}^{(e)} \) down
- If \( \hat{z}_j < 0 \) & \( \hat{z}_j \) is on: \( w_{kj}^{(e)} \) up
- If \( \hat{z}_j \) is off: do nothing.

Hidden layer

\[ \frac{dE_n}{dw_{ji}^{(u)}} = \sum_k \frac{dE_n}{dak} \frac{dak}{dz_j} \frac{dz_j}{dw_{ji}^{(u)}} = \sum_k \frac{dE_n}{dak} \frac{\sum_k \frac{dak}{dz_j} \hat{y}_j}{\sum_k \frac{dak}{dz_j} \hat{y}_j} \frac{w_{kj}^{(e)} h(\sigma)}{w_{kj}^{(e)} h(\sigma)} \frac{x_i}{w_{kj}^{(e)} h(\sigma)} \]

"error of node \( j \)"

\[ \frac{dE_n}{dw_{ji}^{(u)}} = \sum_k \frac{dak}{dz_j} \hat{y}_j \frac{w_{kj}^{(e)} h(\sigma)}{w_{kj}^{(e)} h(\sigma)} \frac{x_i}{w_{kj}^{(e)} h(\sigma)} \]

For sigmoid \( h \):

Note: \( \hat{z}_j \) is a sum of \( \sum_k \hat{y}_k^{(o)} h' \)

where errors of nodes \( k \) it connects to are weighted by connection strengths \( w_{kj}^{(e)} \)

\[ h'(\sigma) \]
Backprop procedure:
1. Input training sample. Compute $Z_j, Y_k$ using current $W$.
2. Calculate error on output node: $E_k$
3. Calculate error on hidden node:
   $$E_j = (\sum E_k W_{kj}) h'(a_j)$$
   Iterate if more layers available.
4. Compute gradient:
   $$\frac{dE_n}{dW_{ji}} = E_j z_i \text{ or } E_j x_i$$
5. Update $W = W - \eta E_n$

- Backprop is basically an efficient use of the chain rule: $O(\# \text{weights})$.

- Note: Batch update can also be done:
  $$\overline{VE} = \frac{1}{n} \sum_{i=1}^{n} E_i$$

Single-sample (online) update may help avoid local optimum but batch update characterizes error gradient better.
Practical issues

△ Regularization
  - Use weight decay (or pruning)
    \[ E = E + \| w \| \Rightarrow \text{update: } w = w - \eta E \cdot w \| w \| \leq 1 \]
  - Use early stopping (or check on dev set)

△ Initialization
  - Random init to break symmetry
  - Make sure range is set s.t. saturation doesn't occur.

△ Learning rate
  - Variable or small constant