Machine Learning

- Multi-Layer Perceptrons
- Backpropagation Learning
- Convolutional Neural Networks
Linear classifiers (perceptrons)

- Linear Classifiers
  - a linear classifier is a mapping which partitions feature space using a linear function (a straight line, or a hyperplane)
  - separates the two classes using a straight line in feature space
  - in 2 dimensions the decision boundary is a straight line

Linearly separable data  Linearly non-separable data
**Perceptron Classifier (2 features)**

**Classifier** $f(x; \theta)$

- Weighted sum of the inputs
  
  $r = \mu_1 x_1 + \mu_2 x_2 + \mu_0$

- "linear response"

- Threshold Function
  
  $T(r)$

- Output
  
  $f(x; \theta) \{-1, +1\}$ or, $\{0, 1\}$

  = class decision

**Decision Boundary at** $r(x) = 0$

**Solve:**

$X_2 = -w_1/w_2 \cdot X_1 - w_0/w_2 \quad \text{(Line)}$

```python
r = X.dot(theta.T)  # compute linear response
Yhat = 2*(r > 0)-1  # "sign": predict +1 / -1
```
Perceptron Classifier (2 features)

Classifier \( f(x; \theta) \)

- \( r = \mu_1 x_1 + \mu_2 x_2 + \mu_0 \)
- “linear response”
- weighted sum of the inputs

Threshold Function

\( T(r) \)

output = class decision

\( r = X \cdot \text{dot}(\theta) \)

# compute linear response

Yhat = 2*(r > 0)-1

# ”sign”: predict +1 / -1

1D example:

\( T(r) = -1 \) if \( r < 0 \)
\( T(r) = +1 \) if \( r > 0 \)

Decision boundary = “x such that \( T( w_1 x + w_0 ) \) transitions”
Recall the role of features

- We can create extra features that allow more complex decision boundaries
- Linear classifiers
- Features \([1,x]\)
  - Decision rule: \(T(ax+b) = ax + b \geq 0\)
  - Boundary \(ax+b = 0 \Rightarrow \text{point}\)
- Features \([1,x,x^2]\)
  - Decision rule \(T(ax^2+bx+c)\)
  - Boundary \(ax^2+bx+c = 0 \Rightarrow ?\)

What features can produce this decision rule?
Features and perceptrons

• Recall the role of features
  – We can create extra features that allow more complex decision boundaries
  – For example, polynomial features
    \[ \phi(x) = [1 \ x \ x^2 \ x^3 \ldots] \]

• What other kinds of features could we choose?
  – Step functions?

Linear function of features

\[ a \ F1 + b \ F2 + c \ F3 + d \]

Ex: \[ F1 - F2 + F3 \]
Multi-layer perceptron model

- Step functions are just perceptrons!
  - “Features” are outputs of a perceptron
  - Combination of features output of another

Ex: \( F_1 - F_2 + F_3 \)

\[
W^1 = \begin{pmatrix} w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{pmatrix}
\]

\[
W^2 = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}
\]

Linear function of features: \( a F_1 + b F_2 + c F_3 + d \)

Ex: \( F_1 - F_2 + F_3 \)
Multi-layer perceptron model

- Step functions are just perceptrons!
  - “Features” are outputs of a perceptron
  - Combination of features output of another

Linear function of features:
\[ a \cdot F_1 + b \cdot F_2 + c \cdot F_3 + d \]

Ex: \[ F_1 - F_2 + F_3 \]

Regression version:
Remove activation function from output

\[ W^1 = \begin{bmatrix} w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{bmatrix} \]

\[ W^2 = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \]
TODO

- Block layers & color somehow
- Discuss “fully connected”
- Simplified diagram
Features of MLPs

- Simple building blocks
  - Each element is just a perceptron function

- Can build upwards

Perceptron:
- Step function /
- Linear partition

Input Features
Features of MLPs

- Simple building blocks
  - Each element is just a perceptron function

- Can build upwards

2-layer:
  “Features” are now partitions
  All linear combinations of those partitions
Features of MLPs

- Simple building blocks
  - Each element is just a perceptron function

- Can build upwards

3-layer:
- “Features” are now complex functions
- Output any linear combination of those
Features of MLPs

- Simple building blocks
  - Each element is just a perceptron function

- Can build upwards

Current research: “Deep” architectures (many layers)
Features of MLPs

- Simple building blocks
  - Each element is just a perceptron function

- Can build upwards

- Flexible function approximation
  - Approximate arbitrary functions with enough hidden nodes
Neural networks

• Another term for MLPs
• Biological motivation

• Neurons
  – “Simple” cells
  – Dendrites sense charge
  – Cell weighs inputs
  – “Fires” axon

“How stuff works: the brain”
# Activation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
<th>Derivative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistic</td>
<td>$\sigma(z) = \frac{1}{1 + \exp(-z)}$</td>
<td>$\frac{\partial \sigma}{\partial z}(z) = \sigma(z)(1 - \sigma(z))$</td>
</tr>
<tr>
<td>Hyperbolic Tangent</td>
<td>$\sigma(z) = \frac{1 - \exp(-2z)}{1 + \exp(-2z)}$</td>
<td>$\frac{\partial \sigma}{\partial z}(z) = 1 - (\sigma(z))^2$</td>
</tr>
<tr>
<td>Gaussian</td>
<td>$\sigma(z) = \exp(-z^2/2)$</td>
<td>$\frac{\partial \sigma}{\partial z}(z) = -z\sigma(z)$</td>
</tr>
<tr>
<td>ReLU (rectified linear)</td>
<td>$\sigma(z) = \max(0, z)$</td>
<td>$\frac{\partial \sigma}{\partial z}(z) = 1[z &gt; 0]$</td>
</tr>
<tr>
<td>Linear</td>
<td>$\sigma(z) = z$</td>
<td>and many others…</td>
</tr>
</tbody>
</table>
Feed-forward networks

- Information flows left-to-right
  - Input observed features
  - Compute hidden nodes (parallel)
  - Compute next layer...

\[
R = X\cdot W[0] + B[0] \quad \# \text{linear response}
\]

\[
H_1 = \text{Sig}( R ) \quad \# \text{activation f'n}
\]

\[
S = H_1\cdot W[1] + B[1] \quad \# \text{linear response}
\]

\[
H_2 = \text{Sig}( S ) \quad \# \text{activation f'n}
\]
Feed-forward networks

- Information flows left-to-right
  - Input observed features
  - Compute hidden nodes (parallel)
  - Compute next layer…

```python
X1 = _add1(X);  # add constant feature
T  = X1.dot(W[0].T);  # linear response
H  = Sig(T);  # activation f’n

H1 = _add1(H);  # add constant feature
S  = H1.dot(W[1].T);  # linear response
H2 = Sig(S);  # activation f’n
%
...
```

- Alternative: recurrent NNs…
Feed-forward networks

A note on multiple outputs:

• Regression:
  – Predict multi-dimensional y
  – “Shared” representation
    = fewer parameters

• Classification
  – Predict binary vector
  – Multi-class classification
    \[ y = 2 = [0 \ 0 \ 1 \ 0 \ ... ] \]
  – Multiple, joint binary predictions
    (image tagging, etc.)
  – Often trained as regression (MSE),
    with saturating activation
Machine Learning

- Multi-Layer Perceptrons
- Backpropagation Learning
- Convolutional Neural Networks
Training MLPs

- Observe features “x” with target “y”
- Push “x” through NN = output is “ŷ”
- Error: \( (y - ŷ)^2 \)  
  (Can use different loss functions if desired…)
- How should we update the weights to improve?

- Single layer
  - Logistic sigmoid function
  - Smooth, differentiable
- Optimize using:
  - Batch gradient descent
  - Stochastic gradient descent
Gradient calculations

- Think of NNs as “schematics” made of smaller functions
  - Building blocks: summations & nonlinearities
  - For derivatives, just apply the chain rule, etc!

Ex: \( f(g, h) = g^2 h \)

\[
\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f} \cdot 2g(\cdot)h(\cdot) \quad \frac{\partial J}{\partial h} = \frac{\partial J}{\partial f} \cdot g^2(\cdot)
\]

save & reuse info \((g, h)\) from forward computation!
Backpropagation

- Just gradient descent…
- Apply the chain rule to the MLP

\[
\frac{\partial J}{\partial w_{k,j}^2} = -2 \sum_{k'} (y_{k'} - \hat{y}_{k'}) (\partial \hat{y}_{k'}) \\
= -2(y_k - \hat{y}_k) \, \sigma'(s_k) \, h_j
\]

(Identical to logistic mse regression with inputs “h_j”)

Forward pass

Loss function
\[
J_i(W) = \sum_k (y_k^{(i)} - \hat{y}_k^{(i)})^2
\]

Output layer
\[
\hat{y}_k = \sigma(s_k) = \sigma(\sum_j w_{k,j}^2 h_j)
\]

Hidden layer
\[
h_j = \sigma(t_j) = \sigma(\sum_i w_{j,i}^1 x_i)
\]
Backpropagation

- Just gradient descent...
- Apply the chain rule to the MLP

\[
\frac{\partial J}{\partial w^2_{kj}} = -2 \sum_{k'} (y_{k'} - \hat{y}_{k'}) \left( \frac{\partial \hat{y}_{k'}}{\partial h_j} \right) \\
= -2(y_k - \hat{y}_k) \sigma'(s_k) \ h_j \\
= -2(y_k - \hat{y}_k) \sigma'(s_k) \beta^2_k
\]

\[
\frac{\partial J}{\partial w^1_{ji}} = \sum_k -2(y_k - \hat{y}_k) \left( \frac{\partial \hat{y}_k}{\partial h_j} \right) \\
= \sum_k -2(y_k - \hat{y}_k) \sigma'(s_k) \ w^2_{kj} \ \partial h_j \\
= \sum_k -2(y_k - \hat{y}_k) \sigma'(s_k) \ w^2_{kj} \ \sigma'(t_j) \ x_i \\
= \sum_k -2(y_k - \hat{y}_k) \sigma'(s_k) \ w^2_{kj} \ \sigma'(t_j) \ x_i \\
= \sum_k -2(y_k - \hat{y}_k) \sigma'(s_k) \ \beta^2_k
\]

Forward pass

Loss function
\[
J_i(W) = \sum_k (y^{(i)}_k - \hat{y}^{(i)}_k)^2
\]

Output layer
\[
\hat{y}_{k} = \sigma(s_k) = \sigma(\sum_j w^2_{kj} h_j)
\]

Hidden layer
\[
h_j = \sigma(t_j) = \sigma(\sum_i w^1_{ji} x_i)
\]

(Idential to logistic mse regression with inputs “h_j”)
Backpropagation

- Just gradient descent…
- Apply the chain rule to the MLP

\[
\frac{\partial J}{\partial w_{k,j}^{2}} = -2(y_k - \hat{y}_k) \sigma'(s_k) h_j
\]

\[
\frac{\partial J}{\partial w_{j,i}^{1}} = \sum_k -2(y_k - \hat{y}_k) \sigma'(s_k) w_{k,j}^2 \sigma'(t_j) x_i
\]

B2 = \(Y-Yhat\) * dSig(S) \((1xN3)\)

G2 = B2.T.dot( H ) \((N3x1)\) * \((1xN2)\) = \((N3xN2)\)

B1 = B2.dot(W[1]) * dSig(T) \((1xN3)\) * \((N3xN2)\) = \((1xN2)\)

G1 = B1.T.dot( X ) \((N2xN1)\)

Forward pass

Loss function

\[J_i(W) = \sum_k (y_k^{(i)} - \hat{y}_k^{(i)})^2\]

Output layer

\[\hat{y}_k = \sigma(s_k) = \sigma(\sum_j w_{k,j}^2 h_j)\]

Hidden layer

\[h_j = \sigma(t_j) = \sigma(\sum_i w_{j,i}^1 x_i)\]
Example: Regression, MCycle data

- Train NN model, 2 layer
  - 1 input features => 1 input units
  - 10 hidden units
  - 1 target => 1 output units
  - Logistic sigmoid activation for hidden layer, linear for output layer

Data:
+ learned prediction f’n:

Responses of hidden nodes (= features of linear regression): select out useful regions of “x”
Example: Classification, Iris data

- Train NN model, 2 layer
  - 2 input features => 2 input units
  - 10 hidden units
  - 3 classes => 3 output units (y = [0 0 1], etc.)
  - Logistic sigmoid activation functions
  - Optimize MSE of predictions using stochastic gradient
Demo Time!

http://playground.tensorflow.org/
MLPs in practice

- Example: Deep belief nets
  - Handwriting recognition
  - Online demo
  - 784 pixels ⇔ 500 mid ⇔ 500 high ⇔ 2000 top ⇔ 10 labels

\[ x \rightarrow h^1 \rightarrow h^2 \rightarrow h^3 \rightarrow \hat{y} \]

[Hinton et al. 2007]
MLPs in practice

- Example: Deep belief nets
  - Handwriting recognition
  - Online demo
  - 784 pixels ↔ 500 mid ↔ 500 high ↔ 2000 top ↔ 10 labels

$h_1$, $h_2$, $h_3$, $\hat{y}$

[Hinton et al. 2007]
MLPs in practice

- Example: Deep belief nets
  - Handwriting recognition
  - Online demo
  - 784 pixels ⇔ 500 mid ⇔ 500 high ⇔ 2000 top ⇔ 10 labels

Fix output, simulate inputs

(c) Alexander Ihler
Convolutional networks

- Organize & share the NN’s weights (vs “dense”)
- Group weights into “filters”

Input: 28x28 image  Weights: 5x5
Convolutional networks

- Organize & share the NN’s weights (vs “dense”)
- Group weights into “filters” & convolve across input image

Input: 28x28 image  
Weights: 5x5  
24x24 image

$h_1 = \sigma(\sum_{ij} w_{ij} x_{ij})$
Convolutional networks

- Organize & share the NN’s weights (vs “dense”)
- Group weights into “filters” & convolve across input image

Input: 28x28 image  Weights: 5x5

Run over all patches of input ) activation map
Convolutional networks

- Organize & share the NN’s weights (vs “dense”)
- Group weights into “filters” & convolve across input image
- Many hidden nodes, but few parameters!

Input: 28x28 image  
Weights: 5x5  
Hidden layer 1
Convolutional networks

- Again, can view components as building blocks
- Design overall, deep structure from parts
  - Convolutional layers
  - “Max-pooling” (sub-sampling) layers
  - Densely connected layers

LeNet-5 [LeCun 1980]
Ex: AlexNet

- Deep NN model for ImageNet classification
  - 650k units; 60m parameters
  - 1m data; 1 week training (GPUs)

[Krizhevsky et al. 2012]
Hidden layers as “features”

- Visualizing a convolutional network’s filters  [Zeiler & Fergus 2013]

Slide image from Yann LeCun:
https://drive.google.com/open?id=0BxKBnD5y2M8NclFWSXNxa0JIZTg
Dropout

• Another recent technique
  – Randomly “block” some neurons at each step
  – Trains model to have redundancy (predictions must be robust to blocking)

Each training prediction: sample neurons to remove

```python
# ... during training ...
R = X.dot(W[0])+B[0];  # linear response
H1= Sig( R );          # activation f’n
H1 *= np.random.rand(*H1.shape)<p;  # drop out!
```
Neural networks & DBNs

- Want to try them out?
- Matlab “Deep Learning Toolbox”
  https://github.com/rasmusbergpalm/DeepLearnToolbox

Matlab/Octave toolbox for deep learning. Includes Deep Belief Nets, Stacked Autoencoders, Convolutional Neural Nets, Convolutional Autoencoders and vanilla Neural Nets. Each method has examples to get you started.

- PyLearn2
  https://github.com/lisa-lab/pylearn2

- TensorFlow

(c) Alexander Ihler
Summary

• Neural networks, multi-layer perceptrons

• Cascade of simple perceptrons
  – Each just a linear classifier
  – Hidden units used to create new features

• Together, general function approximators
  – Enough hidden units (features) = any function
  – Can create nonlinear classifiers
  – Also used for function approximation, regression, …

• Training via backprop
  – Gradient descent; logistic; apply chain rule. Building block view.

• Advanced: deep nets, conv nets, dropout, …