Lecture 4

Neural Networks
ICS 273A UC Irvine
Instructor: Max Welling
Neurons

• Neurons communicate by receiving signals on their dendrites. Adding these signals and firing off a new signal along the axon if the total input exceeds a threshold.

• The axon connects to new dendrites through synapses which can learn how much signal is transmitted.

• McCulloch and Pitt ('43) built a first abstract model of a neuron.

\[ y = g\left(\sum_{i} W_i x_i + b\right) \]
Neurons

- We have about $10^{11}$ neurons, each one connected to $10^4$ other neurons on average.

- Each neuron needs at least $10^{-3}$ seconds to transmit the signal.

- So we have many, slow neurons. Yet we recognize our grandmother in $10^{-1}$ sec.

- Computers have much faster switching times: $10^{-10}$ sec.

- Conclusion: brains compute in parallel!

- In fact, neurons are unreliable/noisy as well. But since things are encoded redundantly by many of them, their population can do computation reliably and fast.
Classification / Regression

• Neural nets are a parameterized function $Y = f(X; W)$ from inputs ($X$) to outputs ($Y$).

• If $Y$ is continuous: regression, if $Y$ is discrete: classification.

• We adapt the weights so as to minimize the error between the data and the model predictions. Or, in other words: maximize the conditional probability of the data (output, given attributes)

• E.g. 2-class classification ($y=0/1$)

\[
P(y | x) = \prod_{n=1}^{N} f(x_n)^{y_n} (1 - f(x_n))^{1-y_n}
\]

\[
error = - \sum_{n=1}^{N} y_n \log f(x_n) + (1 - y_n) \log(1 - f(x_n))
\]

looks familiar?
• If we use the following model for $f(x)$, we obtain logistic regression!

$$y_n = f(x_n; W) = \sigma(\sum_i W_i x_{in} + b)$$

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

• This is called a “perceptron” in neural networks jargon.

• Perceptrons can only separate classes linearly.
Regression

- Probability of output given input attributes is Normally distributed.

\[ p(y \mid x) = c \prod_{n=1}^{N} \prod_{i=1}^{d_{in}} \exp \left[ -\frac{1}{2} \gamma (y_{in} - \sum_{j} W_{ij} x_{jn} - b_i)^2 \right] \]

- Error is negative log-probability = squared loss function:

\[ \text{error} = \sum_{n=1}^{N} \sum_{i=1}^{d_{out}} (y_{in} - \sum_{j=1}^{d_{in}} W_{ij} x_{jn} - b_i)^2 \]
Optimization

- We use stochastic gradient descent: pick a single data-item, compute the contribution of that data-point to the overall gradient and update the weights.

Repeat:
1) Pick random data-item \((y_n, x_n)\)

\[ 2a) \frac{d \text{error}_n}{dW_{ij}} \propto -(y_{in} - \sum_k W_{ik} x_{kn}) x_{jn} \]

\[ 2b) \frac{d \text{error}_n}{db_i} \propto -(y_{in} - \sum_k W_{ik} x_{kn}) \]

\[ 3a) W_{ij} \leftarrow W_{ij} - \eta \frac{d \text{error}_n}{dW_{ij}} \]

\[ 3b) b_i \leftarrow b_i - \eta \frac{d \text{error}_n}{db_i} \]
Stochastic Gradient Descent

- Stochastic gradient descent does not converge to the minimum, but “dances” around it.
- To get to the minimum, one needs to decrease the step-size as one get closer to the minimum.
- Alternatively, one can obtain a few samples and average predictions over them (similar to bagging).
• Single layers can only do linear things. If we want to learn non-linear decision surfaces, or non-linear regression curves, we need more than one layer.
• In fact, NN with 1 hidden layer can approximate any boolean and cont. functions.
Back-propagation

• How do we learn the weights of a multi-layer network?
  Answer: Stochastic gradient descent. But now the gradients are harder!

example:

\[
error = \sum_{in} y_{in} \log \sigma_{in} + (1 - y_{in}) \log (1 - \sigma_{in})
\]

\[
\frac{d error}{d W_{jk}^2} = \sum_{in} \frac{d error_{n}}{\sigma_{in}} \frac{d \sigma_{in}}{d W_{jk}^2} = \sum_{in} \frac{d error_{n}}{\sigma_{in}} \sigma_{in} (1 - \sigma_{in}) \frac{d}{d W_{jk}^2} \left( \sum_{j} W_{ij}^3 h_{jn}^2 + b_{j}^3 \right)
\]

\[
\sum_{in} \frac{d error_{n}}{\sigma_{in}} \sigma_{in} (1 - \sigma_{in}) W_{ij}^3 \frac{d h_{jn}^2}{d W_{jk}^2} = \sum_{in} \frac{d error_{n}}{\sigma_{in}} \sigma_{in} (1 - \sigma_{in}) W_{ij}^3 \sigma_{jn} (1 - \sigma_{jn}) \frac{d}{d W_{jk}^2} \left( \sum_{k} W_{jk}^2 h_{kn}^1 + b_{j}^2 \right)
\]

\[
\sum_{in} \frac{d error_{n}}{\sigma_{in}} \sigma_{in} (1 - \sigma_{in}) W_{ij}^3 \sigma_{jn} (1 - \sigma_{jn}) h_{kn}^1 = \sum_{in} \frac{d error_{n}}{\sigma_{in}} \sigma_{in} (1 - \sigma_{in}) W_{ij}^3 \sigma_{jn} (1 - \sigma_{jn}) \sigma(\sum_{l} W_{kl}^1 x_{in} + b_{k}^1)
\]

\[
\sum_{in} \frac{d error_{n}}{\sigma_{in}} \sigma_{in} (1 - \sigma_{in}) W_{ij}^3 \sigma_{jn} (1 - \sigma_{jn}) \sigma(\sum_{l} W_{kl}^1 x_{in} + b_{k}^1)
\]
Back Propagation

\[
\hat{y}_i = \sigma(\sum_j W_{ij}^3 h_j^2 + b_i^3)
\]

\[
h_i^2 = \sigma(\sum_j W_{ij}^2 h_j^1 + b_i^2)
\]

\[
h_i^1 = \sigma(\sum_j W_{ij}^1 x_j + b_i^1)
\]

\[
\delta_{in}^3 = \hat{y}_i (1 - \hat{y}_i) \frac{d \text{error}_{in}}{d \sigma_{in}}
\]

\[
\delta_{jn}^2 = h_j^2 (1 - h_j^2) \sum_{\text{upstream } i} W_{ij}^3 \delta_{in}^3
\]

\[
\delta_{kn}^1 = h_k^1 (1 - h_k^1) \sum_{\text{upstream } j} W_{jk}^2 \delta_{jn}^2
\]
Back Propagation

\[ \delta^3_{in} = \hat{y}_{in}(1 - \hat{y}_{in}) \frac{d \text{error}_{in}}{d \sigma_{in}} \]

\[ \frac{d \text{error}}{dW^2_{jk}} = \sum_{in} \frac{d \text{error}_{in}}{\sigma_{in}} \sigma_{in}(1 - \sigma_{in})W^3_{ij} \sigma_{jn}(1 - \sigma_{jn})\sigma_{kn} \]

\[ = \delta^2_{jn}h^1_{kn} \]

\[ W^2_{jk} \leftarrow W^2_{jk} - \eta \delta^2_{jn}h^1_{kn} \]

\[ b^2_{j} \leftarrow b^2_{j} - \eta \delta^2_{jn} \]
This hidden unit detects a mildly left sloping road and advises to steer left.

How would another hidden unit look like?
Weight Decay

• NN can also overfit (of course).

• We can try to avoid this by initializing all weights/biases terms to very small random values and grow them during learning.

• One can now check performance on a validation set and stop early.

• Or one can change the update rule to discourage large weights:

\[
W_{jk}^2 \leftarrow W_{jk}^2 - \eta \delta_{jn}^2 h_{kn}^1 - \lambda W_{jk}^2 \\
\]

\[
b_j^2 \leftarrow b_j^2 - \eta \delta_{jn}^2 - \lambda b_j^2
\]

• Now we need to set $\lambda$ using X-validation.

• This is called “weight-decay” in NN jargon.
Momentum

• In the beginning of learning it is likely that the weights are changed in a consistent manner.

• Like a ball rolling down a hill, we should gain speed if we make consistent changes. It’s like an adaptive stepsize.

• This idea is easily implemented by changing the gradient as follows:

\[
\Delta W_{jk}^2(\text{new}) = \eta \delta_{jn}^2 h_{kn}^1 + \gamma \Delta W_{jk}^2(\text{old})
\]

(\text{and similar to biases})

\[
W_{jk}^2 \leftarrow W_{jk}^2 - \Delta W_{jk}^2(\text{new})
\]
Conclusion

• NN are a flexible way to model input/output functions

• They can be given a probabilistic interpretation

• They are robust against noisy data

• Hard to interpret the results (unlike DTs)

• Learning is fast on large datasets when using stochastic gradient descent plus momentum.

• Overfitting can be avoided using weight decay or early stopping

• There are also NN which feed information back (recurrent NN)

• Many more interesting NNs: Boltzman machines, self-organizing maps,...