Stochastic gradient descent
Neural networks
Batch versus Online Learning

Many learning algorithms fit pattern of *batch* minimization:

- Objective for all data:
  \[ F(w) = \frac{1}{N} \sum_{n=1}^{N} f(w, (x_n, t_n)) \]

- Objective for example n:
  \[ f(w, (x_n, t_n)) = - \log p(t_n | x_n, w) \]

  Effective but requires computation & storage linear in data size: \( O(N) \)

Alternative *online learning* via *stochastic gradient* descent:

- Sampled data point:
  \[ z_k = (x_i, t_i) \sim \text{Unif}\left((x_1, t_1), \ldots (x_N, t_N)\right) \]

- Update via gradient step:
  \[ w_{k+1} = w_k - \eta_k \nabla f(w_k, z_k) \]

  Cheap computation cost for each update: \( O(1) \).

  No need to store all \( N \) examples in memory. Load on demand.
Select one example at random from dataset

\[ z_k = (x_i, t_i) \sim \text{Unif}\left( (x_1, t_1), \ldots, (x_N, t_N) \right) \]

Take step in the direction of gradient at chosen example

\[ w_{k+1} = w_k - \eta_k \nabla f(w_k, z_k) \]

Why does it work?

Sampled step direction has right expected value.

\[ \mathbb{E}_{z \sim p(z)} \left[ \nabla f(w, z) \right] = \nabla F(w) \]
SGD for linear regression

Sometimes called least-mean-squares (LMS) algorithm

\[
f(w, x_n, t_n) = \frac{1}{2}(t_n - w^T \phi(x_n))^2 \]

\[
w_{k+1} = w_k - \eta_k(w_k^T \phi(x_k) - t_k)\phi(x_k)\]

Training error (residual sum-of-squares) vs iteration

\[
\text{RSS vs iteration} \quad 
\begin{array}{c}
\text{training error (residual sum-of-squares)} \\
\text{iterations } k
\end{array}
\]
Stochastic Gradient: Tips and Tricks

\[ w_{k+1} = w_k - \eta_k \nabla f(w_k, z_k) \quad \mathbb{E}[f(w_k, z_k)] = F(w_k) \]

How can we produce a single parameter estimate?

\[ \bar{w}_k = \frac{1}{k} \sum_{i=1}^{k} w_i = \bar{w}_{k-1} + \frac{1}{k}(w_k - \bar{w}_{k-1}) \]

Polyak-Ruppert averaging

How should we set the step size \( \eta_k \)?

\[ \eta_k = (\tau_0 + k)^{-\kappa} \quad \tau_0 \geq 0 \quad \kappa \in (0.5, 1] \]

Robbins-Monro conditions

Conventional batch step size rules fail (in theory & practice)
Stochastic Gradient: Using small batches

- Refinement: Take batches of data for each step: \[ 1 < B \ll N \]

\[
\begin{align*}
    w_{k+1} &= w_k - \frac{\eta_k}{B} \sum_{b \in \mathcal{B}} \nabla f(w_b, z_b)
\end{align*}
\]

\( \mathcal{B} \) is a random subset of \( \{1, \ldots, N\} \) of size \( B \).

- Each gradient step \( B \) times more costly to compute, but often compensated by reduction in estimate noise
- How does this work in practice?

*Often excellent for big datasets!*

*But several tricky tuning parameters:*
step size decay schedule, number of batches
SGD for Logistic Regression

\[ p(t_n \mid x_n, w) = \text{Ber}(t_n \mid \sigma(w^T \phi(x_n))) \]

\[ \mu_n = \sigma(w^T \phi(x_n)) \]

\[ F(w) = -\sum_{n=1}^{N} t_n \log \mu_n + (1 - t_n) \log(1 - \mu_n) \]

Gradient for example chosen at iteration \( k \):

\[ \nabla f(w, x_k, t_k) = (\mu_k - t_k) \phi(x_k) \]

Stochastic Gradient Descent:

\[ w_{k+1} = w_k - \eta_k (\mu_k - t_k) \phi(x_k) \]
### Threshold Decision Rule:

\[ y_n = \mathbb{I}(w^T \phi(x_n) > 0) \]

### Perceptron Learning Rule:

- If \( y_k = t_k \), \( w_{k+1} = w_k \)
- If \( y_k \neq t_k \), \( w_{k+1} = w_k + \tilde{t}_k \phi(x_k) \)
  
  \[ \tilde{t}_k = 2t_k - 1 \in \{ +1, -1 \} \]
Perceptron Algorithm Convergence

- Black arrow points in direction favoring red over blue
- Green circles indicate points with incorrect predictions, which produce modified weights
- Once all points are perfectly classified, algorithm halts
Perceptron Algorithm Properties

**Strengths**
- Guaranteed to converge if data linearly separable (in feature space; reduces angle to true separators)
- Simple algorithm is easy to generalize to more complex models

**Weaknesses**
- May be slow to converge (worst-case performance poor)
- If data not linearly separable, will never converge
- Solution depends on order data visited; no notion of a “best” separating hyperplane
- Non-probabilistic: No measure of confidence in decisions, difficult to generalize to other problems
Stochastic gradient descent
Neural networks
Neural Networks

- Feed-forward
- Transform input into hidden, non-linear, tunable feature representation
- Use this hidden representation to produce output
- Size of hidden layer \( M \) and weights can all be optimized.
Neural Networks: From Hidden to Output

Output depends on hidden layer via linear/logistic regression

\[ \phi_0(z_n) = [1, z_{n1}, \ldots, z_{nM}]^T \]

**Classification:** \( t_n \in \{0, 1\} \)
\[ p(t_n \mid x_n, w) = \text{Ber}(t_n \mid \sigma(w_0^T \phi_0(z_n))) \]

**Regression:** \( t_n \in \mathbb{R} \)
\[ p(t_n \mid x_n, w) = \text{Norm}(t_n \mid w_0^T \phi_0(z_n), \beta^{-1}) \]

\( z_n \in \mathbb{R}^M \)
Hidden units: deterministic, tunable, non-linear functions of inputs

\[ \phi_m(x_n) = [1, x_{n1}, \ldots, x_{nD}]^T \]

\[ z_{nm} = h(w_m^T \phi_m(x_n)) \]

- If \( h(\cdot) \) is smooth squashing function, approximates any smooth function arbitrarily well (in bounded region) with large enough \( M \)

- Linear activation functions are not useful. Why?

Non-linear activation function \( h() \):

\[
\begin{align*}
\tanh(\eta) &= \frac{e^\eta - e^{-\eta}}{e^\eta + e^{-\eta}} \\
\sigma(\eta) &= \left(1 + e^{-\eta}\right)^{-1} \\
\tanh(\eta) &= 2\sigma(2\eta) - 1
\end{align*}
\]
Neural Networks: Training Objective

MAP with Gaussian prior on all weights:

\[ f(w) = \frac{\alpha}{2} \sum_{m=0}^{M} w_m^T w_m - \sum_{n=1}^{N} \log p(t_n | x_n, w) \]

\[ p(t_n | x_n, w) = \text{Ber}(t_n | \sigma(w_0^T \phi_0(z_n))) \]

\[ z_{nm} = h(w_{m}^T \phi_m(x_n)) \]

Matches the batch minimization pattern, so

**We can do gradient descent or SGD!**
**Neural Networks: Training**

**Back-propagation:** Gradient descent using the chain rule
- Gradient expressions have simple recursive form
- Can be computed in linear time!

Update for input node
- $\frac{\partial z}{\partial x}$

Update for hidden node
- $\frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$

Convert hidden to input

(Bengio, KDD 2014 Tutorial)
Neural Networks: Training

**Back-propagation:** Gradient descent using the chain rule
- Gradient expressions have simple recursive form
- Can be computed in linear time!

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

Update for input node
Update for hidden node
Convert hidden to input

(Bengio, KDD 2014 Tutorial)
Back-propagation: Chain rule of derivatives gives gradients a simple form, which is efficiently computed.

SGD works very well!

Likelihood is highly non-convex, many local optima

Non-identifiable and hard to interpret: Order of features and signs of outputs arbitrary, so for any weight vector, number of equivalent models is

\[ 2^M \cdot M! \]
Multiple Layers and Deep Networks

**LeNet5:** Convolutional Neural Net for Digit Classification (LeCun et al., 1998)
Deep Learning for Object Recognition

**ImageNet dataset:** 15 million images
22,000 categories

**AlexNet** (Alex Krizhevsky et al, NIPS 2012)
Deep convolutional neural network, trained via backprop on multiple GPUs.
Deep Learning for Object Recognition

State-of-the-art (2012) on ImageNet:
>25% error

AlexNet:
16% error

(Rob Fergus, NIPS 2013 Tutorial)
Deep Learning for Object Recognition

Try it out: http://deeplearning.cs.toronto.edu

(Bengio, KDD 2014 Tutorial)
Deep Learning for Speech Recognition

(Bengio, KDD 2014 Tutorial)