Machine Learning Basics

Niloy Mitra  Iasonas Kokkinos  Federico Monti  Emanuele Rodolà  Michael Bronstein  Or Litany  Leonidas Guibas

UCL  UCL  USI Lugano  La Sapienza  Imperial College USI Lugano  Stanford University Facebook  Stanford University

http://geometry.cs.ucl.ac.uk/dl_for_CG/
Course Information (slides/code/comments)

http://geometry.cs.ucl.ac.uk/dl_for_CG/
# Timetable

| Sessions: A. 9:00-10:30 (coffee) B. 11:00-12:30 [LUNCH] C. 13:30-15:00 (coffee) D. 15:30-17:00 |
|---|---|---|---|---|
| Theory/Basics | Niloy | Federico | Iasonas | Emanuele |
| Introduction | 9:00 | X | X | X | X |
| Machine Learning Basics | ~9:05 | X | | |
| Neural Network Basics | ~9:35 | | X | |
| Alternatives to Direct Supervision (GANs) | ~11:00 | | | X |
| Image Domain | ~11:45 | X | | |
| 3D Domains (extrinsic) | ~13:30 | X | | |
| 3D Domains (intrinsic) | ~14:15 | | | X |
| Physics and Animation | ~16:00 | X | | |
| Discussion | ~16:45 | X | X | X | X |
Machine Learning Variants

• **Supervised**
  • Classification
  • Regression
  • Data consolidation

• **Unsupervised**
  • Clustering
  • Dimensionality Reduction

• **Weakly supervised/semi-supervised**
  Some data supervised, some unsupervised

• **Reinforcement learning**
  Supervision: sparse reward for a sequence of decisions
Machine Learning Variants

- **Supervised**
  - *Classification*
  - Regression
  - Data consolidation

- **Unsupervised**
  - Clustering
  - Dimensionality Reduction

- **Weakly supervised/semi-supervised**
  - Some data supervised, some unsupervised

- **Reinforcement learning**
  - Supervision: sparse reward for a sequence of decisions
Classification Examples

- Digit Recognition

![Classification Examples Image]
Classification Examples

- Digit Recognition

- Spam Detection

Digit Recognition Example:

```
32222227
```

Spam Detection Example:

```
Subject: ***XIXX XAIL*** Don’t waste your time on discarded by health!
From: Pete-Louise2008@candle@inspection.com
Date: 2009-01-01
To: bill@institute.com <bill@institute.com>

5 reasons of quit smoking! [http://www.markthill.com/]
```
Classification Examples

• Digit Recognition

• Spam Detection

• Face detection
uld we give a loan to this customer?
Segmentation + Classification in Real Images
Segmentation + Classification in Real Images

Evaluation measures: Confusion matrix, ROC curve, precision, recall, etc.
`Faceness’ Function: Classifier

depth

background

face

Deep Learning for CG & Geometry Processing
`Faceness’ Function: Classifier

background

decision boundary

face
Machine Learning Variants

- **Supervised**
  - Classification
  - **Regression**
  - Data consolidation

- **Unsupervised**
  - Clustering
  - Dimensionality Reduction

- **Weakly supervised/semi-supervised**
  - Some data supervised, some unsupervised

- **Reinforcement learning**
  - Supervision: sparse reward for a sequence of decisions
Human Face/Pose Estimation

- Human estimation: from image to vector-valued pose estimate

[Blanz and Vetter, Siggraph, 1999]
Human Face/Pose Estimation

- Human estimation: from image to vector-valued pose estimate

[Blanz and Vetter, Siggraph, 1999]
Machine Learning Variants

• **Supervised**
  • Classification
  • Regression
  • Data consolidation

• **Unsupervised**
  • *Clustering*
  • Dimensionality Reduction

• **Weakly supervised/semi-supervised**
  Some data supervised, some unsupervised

• **Reinforcement learning**
  Supervision: sparse reward for a sequence of decisions
Clustering: Group Points According to X

• Break a set of data into coherent groups
  • Labels are ‘invented’
Clustering: Group Points According to X

• Break a set of data into coherent groups
  • Labels are ‘invented’

![Diagram showing clustering results](image)
Clustering: Group Points According to X

• Break a set of data into coherent groups
  • Labels are ‘invented’
Clustering Examples: Image Segmentation using NCuts

Normalized Cuts - based clustering of natural images
Clustering Examples

• Spotify recommendations

[Chu et al., TVCG, 2009]  [Zheng et al., Eurographics, 2014]
Machine Learning Variants

- **Supervised**
  - Classification
  - Regression
  - Data consolidation

- **Unsupervised**
  - Clustering
    - **Dimensionality Reduction**

- **Weakly supervised/semi-supervised**
  - Some data supervised, some unsupervised

- **Reinforcement learning**
  - Supervision: sparse reward for a sequence of decisions
Dimensionality Reduction (Manifold Learning)

Isomap

Tenenbaum et al., Science, 2000

Yang et al., TOG, 2011

Face Manifold

Averkiou et al., Eurographics, 2014
Example of **Nonlinear** Manifold: Faces

$x_1$

$x_2$
Example of **Nonlinear** Manifold: Faces

\[ \frac{1}{2}(x_1 + x_2) \]
Example of **Nonlinear** Manifold: Faces

\[ \frac{1}{2}(x_1 + x_2) \]

\[ x_1 \quad x \quad x_2 \]
Moving Along Learned Face Manifold

Trajectory along the “male” dimension

[Lample et. al. Fader Networks, NIPS 2017]
Moving Along Learned Face Manifold

- Trajectory along the “male” dimension
- Trajectory along the “young” dimension

[Lample et. al. Fader Networks, NIPS 2017]
PCA Basis

• All eigenvalues of symmetric matrices are real.
• Any real symmetric n×n matrix has a set of \( n \) mutually orthogonal eigenvectors.

\[
y = Ax
\]

\[
Ae_i = \lambda_i e_i
\]

\[
T = [v_1 \ v_2 \ldots]
\]

\[
T^{-1}AT = \text{diag}(\lambda_1, \lambda_2, \ldots)
\]
Code Example
Code Example
Code Example

```python
rng = np.random.RandomState(10)
X = np.dot(rng.rand(2, 2), rng.randn(2, 500)).T

mean_vec = np.mean(X, axis=0)
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)
```
Deep Learning for CG & Geometry Processing

```python
rng = np.random.RandomState(10)
X = np.dot(rng.randn(2, 2), rng.randn(2, 500)).T
mean_vec = np.mean(X, axis=0)
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)
```
Code Example

```
rng = np.random.RandomState(10)
X = np.dot(rng.rand(2, 2), rng.randn(2, 500)).T
mean_vec = np.mean(X, axis=0)
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)
eig_vals, eig_vecs = np.linalg.eig(cov_mat)
```
Code Example

```
mean_vec = np.mean(X, axis=0)
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)
matU, sigma, matV = np.linalg.svd(cov_mat)
```
Morphable Faces
Morphable Faces
Regression: Continuous Output

![Graph showing a scatter plot with red dots representing data points. The x-axis ranges from -6 to 6, and the y-axis ranges from -2 to 4. The dots form a trend line that increases as the x-value increases.]
Regression: Continuous Output
Neural BTF Compression and Interpolation

[Rainer et al. 2019, EG]
Learning a Function

\[ y = f_w(x) \]

- Prediction
- Method
- Parameters
- Input

Calculus:
\[ x \in \mathbb{R} \]

Vector calculus:
\[ \mathbf{x} \in \mathbb{R}^d \]
Learning a Function

\[ y = f_w(x) \]

Calculus

\[ x \in \mathbb{R} \]

Vector calculus

\[ \mathbf{x} \in \mathbb{R}^d \]

*Machine learning: can work also for discrete inputs, strings, images, meshes, animations, ...*
Learning a Function

\[ y = f_w(x) \]

Calculus: \( x \in \mathbb{R} \)

Vector calculus: \( \mathbf{x} \in \mathbb{R}^d \)

Classification: \( y \in \{0, 1\} \)

Machine learning: can work also for discrete inputs, strings, images, meshes, animations, ...
Learning a Function

\[ y = f_w(x) \]

Calculus

\[ x \in \mathbb{R} \]

Vector calculus

\[ \mathbf{x} \in \mathbb{R}^d \]

Classification:

\[ y \in \{0, 1\} \]

Regression:

\[ y \in \mathbb{R} \]

Machine learning: can work also for discrete inputs, strings, images, meshes, animations, …
Learning a **Linear** Separator/Classifier

![Diagram of a linear separator/classifier with nodes and a separating hyperplane]

separating hyperplane
Learning a **Linear** Separator/Classifier

A separating hyperplane is shown between two points $x_1$ and $x_2$. The hyperplane divides the space into two regions, each representing a class.
Learning a **Linear** Separator/Classifier

![Diagram](image)

- Variables: $x_1$, $x_2$, $y$
- Separating hyperplane
Learning a **Linear** Separator/Classifier

\[
\begin{align*}
    y &= w_1 x_1 + w_2 x_2 \\
    \text{separating hyperplane}
\end{align*}
\]
Learning a **Linear** Separator/Classifier

\[ y = f(w_1 x_1 + w_2 x_2) \]
Learning a **Linear** Separator/Classifier

\[ y = f(w_1 x_1 + w_2 x_2) = \mathcal{H}(w_1 x_1 + w_2 x_2) \]
Learning a **Linear** Separator/Classifier

\[
y = f(w_1 x_1 + w_2 x_2) = \mathcal{H}(w_1 x_1 + w_2 x_2)
\]
Learning a **Linear** Separator/Classifier

\[ y = f(w_1 x_1 + w_2 x_2) = \mathcal{H}(w_1 x_1 + w_2 x_2) \]

- **Fixed non-linearity**
- **Learned**
- **Separating hyperplane**
Combining Simple Functions/Classifiers

2 layers of trainable weights

convex polygon region
Combining Simple Functions/Classifiers

3 layers of trainable weights

composition of polygons: convex regions

Eurographics 2019
Regression

1. Least Squares fitting

2. Nonlinear error function and gradient descent

3. Perceptron training (simple neural network)
Regression

1. Least Squares fitting

2. Nonlinear error function and gradient descent

3. Perceptron training (simple neural network)
Reminder: Linear Classifier
Reminder: Linear Classifier

Supervised setting

\[ y_t = \begin{cases} +1 \\ -1 \end{cases} \]
Reminder: Linear Classifier

\[ y_t = \begin{cases} +1 \\ -1 \end{cases} \]
Reminder: Linear Classifier

$x_i$ positive: $x_i \cdot w \geq 0$

$x_i$ negative: $x_i \cdot w < 0$

Supervised setting

labelled input

$y_t = \begin{cases} 
+1 \\ -1
\end{cases}$
Which Line to Pick?

$x_i$ positive: $x_i \cdot w \geq 0$

$x_i$ negative: $x_i \cdot w < 0$

supervised setting

labelled input

$y_t = \begin{cases} +1 & \text{red} \\ -1 & \text{blue} \end{cases}$
Sum of Square Errors (\textit{MSE without the mean})

\[ y^i = w^T x^i + \epsilon^i \]
Sum of Square Errors \((MSE \text{ without the mean})\)

\[ y^i = w^T x^i + \epsilon^i \]

Loss function: sum of squared errors

\[
L(w) = \sum_{i=1}^{N} (\epsilon^i)^2
\]
Sum of Square Errors (MSE without the mean)

\[ y^i = w^T x^i + \epsilon^i \]

Loss function: sum of squared errors

\[ L(w) = \sum_{i=1}^{N} (\epsilon^i)^2 \]

In two variables:

\[ L(w_0, w_1) = \sum_{i=1}^{N} [y^i - (w_0 x_0^i + w_1 x_1^i)]^2 \]
Sum of Square Errors \((MSE \text{ without the mean})\)

\[ y^i = w^T x^i + \epsilon^i \]

Loss function: sum of squared errors

\[ L(w) = \sum_{i=1}^{N} (\epsilon^i)^2 \]

In two variables:

\[ L(w_0, w_1) = \sum_{i=1}^{N} \left[ y^i - (w_0 x^i + w_1 x^i) \right]^2 \]

Question: what is the best (or least bad) value of \(w\)?
Calculus 101

\[ f(x) \]

\[ x^* \]
Calculus 101

\[ f(x) \]

\[ x^* = \text{argmax}_x f(x) \]
Local Extrema Condition

\[ f(x) \]

\[ x^* = \text{argmax}_x f(x) \]
Local Extrema Condition

\[ x^* = \arg\max_x f(x) \quad \Rightarrow \quad f'(x^*) = 0 \]
Local Extrema Condition

\[ x^* = \arg\max_x f(x) \quad \rightarrow \quad f'(x^*) = 0 \]
Vector Calculus 101

\[ f(x) \]

2D function graph
Vector Calculus 101

2D function graph

\[ f(x) \]

isocontours

\[ f(x) = c \]
Vector Calculus 101

2D function graph

$\mathbf{\nabla} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$
Vector Calculus 101

2D function graph

isocontours

gradient field

\[ f(x) \]

\[ f(x) = c \]

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

\[ \bullet \text{ at minimum of function: } \nabla f(x) = 0 \]
**Vector Calculus 101**

- **2D function graph**: $f(x)$
- **isocontours**: $f(x) = c$
- **gradient field**: $\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$

- At minimum of function: $\nabla f(x) = 0$
LS Solution for Linear Regression

\[ y = Xw + \epsilon \]

\[ L(w) = \epsilon^T \epsilon \]

\[ X = \begin{bmatrix} (x^1)^T \\ (x^2)^T \\ \vdots \\ (x^N)^T \end{bmatrix} \]
LS Solution for Linear Regression

\[ y = Xw + \epsilon \]

\[ L(w) = \epsilon^T \epsilon \]

\[ X = \begin{bmatrix}
(x^1)^T \\
(x^2)^T \\
\vdots \\
(x^N)^T
\end{bmatrix} \]

\[ w^* \leftarrow (X^T X)^{-1} X^T y \]
Code Example

\[ w^* \leftarrow (X^T X)^{-1} X^T y \]
import numpy as np
from numpy import array
from numpy import matmul
from numpy.linalg import inv
from numpy.random import rand
from matplotlib import pyplot

# generate data on a line perturbed with some noise
noise_margin = 2
w = rand(2,1) # w[0] is random constant term (offset from origin), w[1] is random linear term (slope)
x = np.linspace(-5, 5, 20)
y = w[0] + w[1]*x + noise_margin*rand(len(x))

# create the design matrix: the x data, and add a column of ones for the constant term
X = np.column_stack([np.ones([len(x), 1]), x.reshape(-1, 1)])

# These are the normal equations in matrix form: w = (X'X)^-1 X' y
w_est = matmul(inv(matmul(X.transpose(), X)), X.transpose()).dot(y)

# For ridge regression, use regularizer
weight = 0.01
w_est = matmul(inv(matmul(X.transpose(), X) + weight*np.identity(2)), X.transpose()).dot(y)

# evaluate the x values in the fitted model to get estimated y values
y_est = w_est[0] + w_est[1]*x

# visualize the fitted model
pyplot.scatter(x, y, color='red')
pyplot.plot(x, y_est, color='blue')
pyplot.show()
import numpy as np
from numpy import array
from numpy import matmul
from numpy.linalg import inv
from numpy.random import rand
from matplotlib import pyplot

# generate data on a line perturbed with some noise
noise_margin = 2
w = rand(2,1) # w[0] is random constant term (offset from origin), w[1] is random linear term (slope)
x = np.linspace(-5, 5, 21)
y = w[0] + w[1]*x + noise_margin*rand(len(x))

# create the design matrix: the x data, and add a column of ones for the constant term
X = np.column_stack([np.ones(len(x)), x.reshape(-1, 1)]

# These are the normal equations in matrix form: \( w = (X'X)^{-1}X'y \)
w_est = matmul(matmul(X.transpose(),X),X.transpose()).dot(y)

# For ridge regression, use regularizer
#weight = 0.01
#w_est = matmul(matmul(X.transpose(),X) + weight*np.identity(2),X.transpose()).dot(y)

# evaluate the x values in the fitted model to get estimated y values
y_est = w_est[0] + w_est[1]*x

# visualize the fitted model
pyplot.scatter(x, y, color='red')
pyplot.plot(x, y_est, color='blue')
pyplot.show()
Linear Regression (Line/Plane Fitting)
Linear Regression (Line/Plane Fitting)
LS Solution for Regression

\[ L(\mathbf{w}) = \sum_{i=1}^{N} (y^i - \mathbf{w}^T \mathbf{x}^i)^2 = \sum_{i=1}^{N} (\epsilon^i)^2 \]

\[ L(\mathbf{w}) = \begin{bmatrix} \epsilon^1 & \epsilon^2 & \ldots & \epsilon^N \end{bmatrix} \begin{bmatrix} \epsilon^1 \\ \epsilon^2 \\ \vdots \\ \epsilon^N \end{bmatrix} \]

\[ \mathbf{y} = \mathbf{Xw} + \epsilon \]
LS Solution for Regression

\[ L(w) = \sum_{i=1}^{N} (y^i - w^T x^i)^2 = \sum_{i=1}^{N} (\epsilon^i)^2 \]

\[ L(w) = \begin{bmatrix} \epsilon^1 & \epsilon^2 & \ldots & \epsilon^N \end{bmatrix} \begin{bmatrix} \epsilon^1 \\ \epsilon^2 \\ \vdots \\ \epsilon^N \end{bmatrix} \]

\[ L(w) = \epsilon^T \epsilon \]

\[ y = Xw + \epsilon \]
Generalized Linear Regression

\[ x \rightarrow \phi(x) = \begin{bmatrix} \phi_1(x) \\ \vdots \\ \phi_M(x) \end{bmatrix} \]
Generalized Linear Regression

\[ x \rightarrow \phi(x) = \begin{bmatrix} \phi_1(x) \\ \vdots \\ \phi_M(x) \end{bmatrix} \]

known nonlinearity
1D Example: k-th Degree Polynomial Fitting

\[ \phi(x) = \begin{bmatrix} 1 \\ x \\ \vdots \\ (x)^K \end{bmatrix} \]

\[ \langle w, \phi(x) \rangle = w_0 + w_1 x + \ldots + w_k (x)^K \]
Generalized Linear Regression

\[ L(w) = \sum_{i=1}^{N} (y_i - w^T \phi(x_i))^T = \sum_{i=1}^{N} (\epsilon^i)^2 \]
Generalized Linear Regression

\[ L(w) = \sum_{i=1}^{N} (y^i - w^T \phi(x^i))^2 = \sum_{i=1}^{N} (\epsilon^i)^2 \]

\[
\begin{bmatrix}
y^1 \\
y^2 \\
\vdots \\
y^N
\end{bmatrix}_{\text{Nx1}} = 
\begin{bmatrix}
\phi(x^1)^T \\
\phi(x^2)^T \\
\vdots \\
\phi(x^N)^T
\end{bmatrix}_{\text{NxM}} 
\begin{bmatrix}
w_1 \\
w_2 \\
\vdots \\
w_M
\end{bmatrix}_{\text{Mx1}} + 
\begin{bmatrix}
\epsilon^1 \\
\epsilon^2 \\
\vdots \\
\epsilon^N
\end{bmatrix}_{\text{Nx1}}
\]

\[ \phi(x) : \mathbb{R}^D \rightarrow \mathbb{R}^M \]
LS Solution for Generalized Linear Regression

\[ y = \Phi w + \epsilon \]

\[ \Phi = \begin{bmatrix}
\phi(x^1)^T \\
\phi(x^2)^T \\
\vdots \\
\phi(x^N)^T
\end{bmatrix} \]
LS Solution for Generalized Linear Regression

\[ y = \Phi w + \epsilon \]

\[ L(w) = \epsilon^T \epsilon \]

\[ \Phi = \begin{bmatrix} \phi(x^1)^T \\ \phi(x^2)^T \\ \vdots \\ \phi(x^N)^T \end{bmatrix} \]

\[ w^* = (\Phi^T \Phi)^{-1} \Phi y \]
```python
import numpy as np
from numpy import array
from numpy.matlib import matmul
from numpy.linalg import inv
from numpy.random import rand
from matplotlib import pyplot

# generate data on a line perturbed with some noise
noise_margin = 3
w = 2*rand(3,1)  # w[0] is random constant term (offset from origin), w[1] is random linear term, w[2] is random quadratic term
x = np.linspace(-5,5,20)
y = w[0] + w[1]*x + w[2]*x**2 + noise_margin*rand(len(x))

# create the design matrix: the x data, and add a column of ones for the constant term
X = np.column_stack([np.ones([len(x),1]), x.reshape(-1, 1), (x**2).reshape(-1, 1)])

# These are the normal equations in matrix form: w = (X’ X)**-1 X’ y
w_est = matmul(inv(matmul(X.transpose(),X)),X.transpose()).dot(y)

# evaluate the x values in the fitted model to get estimated y values
y_est = w_est[0] + w_est[1]*x + w_est[2]*x**2

# visualize the fitted model
pyplot.scatter(x, y)
pyplot.plot(x, y_est, color='red')
pyplot.show()
```
Code Example

\[ w^* = (\Phi^T \Phi)^{-1} \Phi y \]
Hyperparameter: Underfitting vs. Overfitting

Underfitting

classification

regression
Hyperparameter: Underfitting vs. Overfitting

Underfitting

Overfitting

classification

regression

Deep Learning for CG & Geometry Processing
Hyperparameter: Underfitting vs. Overfitting

Underfitting

Overfitting

just right

classification

regression

Deep Learning for CG & Geometry Processing
Hyperparameter Tuning
Hyperparameter Tuning

![Graph showing the relationship between prediction error and model complexity. The graph illustrates the trade-off between high bias and low variance, and low bias and high variance. The x-axis represents model complexity (low to high), and the y-axis represents prediction error. There are two curves: one for the training sample and one for the test sample. The training sample curve shows a decrease in error with increasing complexity, while the test sample curve shows a peak at medium complexity before decreasing. This indicates the need for tuning to find the optimal complexity.](image)
Hyperparameter Tuning

The diagram illustrates the relationship between prediction error and model complexity. The x-axis represents model complexity, ranging from low to high. The y-axis shows prediction error, which decreases as model complexity increases.

The graph highlights two main regions:
- **High Bias, Low Variance**: Models in this region are overfitting (low training error, high test error).
- **Low Bias, High Variance**: Models here are underfitting (high training error, low test error).

The **sweet spot** is where both training and test errors are minimized, indicating an optimal balance between bias and variance.

- **Training Sample**: Data used to train the model.
- **Test Sample**: Data used to evaluate the model's performance.

This diagram is crucial for understanding how to tune hyperparameters to achieve the best model performance.
Selecting \( \lambda \) with Cross-validation
Selecting $\lambda$ with Cross-validation

- Cross validation technique
  - Exclude part of the training data from parameter estimation
  - Use them only to predict the test error
Selecting $\lambda$ with Cross-validation

• Cross validation technique
  • Exclude part of the training data from parameter estimation
  • Use them only to predict the test error

• K-fold cross validation:
  • K splits, average K errors
Selecting $\lambda$ with Cross-validation

- **Cross validation technique**
  - Exclude part of the training data from parameter estimation
  - Use them only to predict the test error

- **K-fold cross validation:**
  - K splits, average K errors

- **Use cross-validation for different values of $\lambda$**
  - pick value that minimizes cross-validation error
Selecting $\lambda$ with Cross-validation

- Cross validation technique
  - Exclude part of the training data from parameter estimation
  - Use them only to predict the test error

- K-fold cross validation:
  - K splits, average K errors

- Use cross-validation for different values of $\lambda$
  - pick value that minimizes cross-validation error

Least glorious, most effective of all methods.
Regression

1. Least Squares fitting

2. Nonlinear error function and gradient descent

3. Perceptron training (simple neural network)
Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness
Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

\[ g(\alpha) = \frac{1}{1 + \exp(-\alpha)} \]
Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

\[ g(\alpha) = \frac{1}{1 + \exp(-\alpha)} \]
Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

\[ g(\alpha) = \frac{1}{1 + \exp(-\alpha)} \]
Extension #1: Logistic Regression

Using squashing (sigmoidal) function for robustness

\[ g(\alpha) = \frac{1}{1 + \exp(-\alpha)} \]
Extension #2: Handling Multiple (2+) Classes

C classes: one-of-c coding (or one-hot encoding)

4 classes, i-th sample is in 3\textsuperscript{rd} class:
\[ y^i = (0, 0, 1, 0) \]
C classes: one-of-c coding (or one-hot encoding)

4 classes, i-th sample is in 3rd class:
\[ y^i = (0, 0, 1, 0) \]

Matrix notation:
\[
Y = \begin{bmatrix}
  y^1 \\
  \vdots \\
  y^N
\end{bmatrix} = \begin{bmatrix}
  y_1 & \cdots & y_C
\end{bmatrix}
\]

where
\[
y_c = \begin{bmatrix}
  y^1_c \\
  \vdots \\
  y^N_c
\end{bmatrix}
\]

\[
W = \begin{bmatrix}
  w_1 & \cdots & w_C
\end{bmatrix}
\]

Loss function:
\[
L(W) = \sum_{c=1}^{C} (y_c - Xw_c)^T (y_c - Xw_c)
\]
Extension #2: Handling Multiple (2+) Classes

C classes: one-of-c coding (or one-hot encoding)

Matrix notation:

\[ Y = \begin{bmatrix} y^1 \\ \vdots \\ y^N \end{bmatrix} = \begin{bmatrix} y_1 & \cdots & y_C \end{bmatrix} \]

4 classes, i-th sample is in 3rd class:
\[ y^i = (0, 0, 1, 0) \]

\[ y_c = \begin{bmatrix} y^1_c \\ \vdots \\ y^N_c \end{bmatrix} \]

Loss function:
\[ L(W) = \sum_{c=1}^{C} (y_c - Xw_c)^T (y_c - Xw_c) \]

Least squares fit (decouples per class):
\[ w^*_c = (X^T X)^{-1} X^T y_c \]
Logistic vs Linear Regression, n>2 classes
Logistic vs Linear Regression, n>2 classes

Linear regression
Logistic vs Linear Regression, n>2 classes
Logistic vs Linear Regression, n>2 classes

Logistic regression does not exhibit the masking problem
Gradient of Cross-entropy Loss

\[ L(w) = - \sum_{i=1}^{N} y^i \log g(w^T x^i) + (1 - y^i) \log(1 - g(w^T x^i)) \]
Gradient of Cross-entropy Loss

\[ L(w) = - \sum_{i=1}^{N} y^i \log g(w^T x^i) + (1 - y^i) \log(1 - g(w^T x^i)) \]

\[ \nabla L(w^*) = 0 \]

nonlinear system of equations!!
Gradient Descent Minimization

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

gradient at any point gives direction of fastest increase
Gradient Descent Minimization

Gradient at any point gives direction of fastest increase

Idea: start at a point and move in the direction opposite to the gradient

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]
Gradient Descent Minimization

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

gradient at any point gives direction of fastest increase
Idea: start at a point and move in the direction opposite to the gradient
Gradient Descent Minimization

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

gradient at any point gives direction of fastest increase

Idea: start at a point and move in the direction opposite to the gradient
Gradient Descent Minimization

Gradient at any point gives direction of fastest increase

Idea: start at a point and move in the direction opposite to the gradient

Initialize: \( x_0 \)

Update: \( x_{i+1} = x_i - \alpha \nabla f(x_i) \) for \( i = 0 \)
Gradient Descent Minimization

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

Update:

\[ x_{i+1} = x_i - \alpha \nabla f(x_i) \quad \text{for } i=1 \]
Gradient Descent Minimization

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

Update:

\[ x_{i+1} = x_i - \alpha \nabla f(x_i) \quad i=1 \]
Gradient Descent Minimization

Update:

\[ x_{i+1} = x_i - \alpha \nabla f(x_i) \]

\[ i = 2 \]
Gradient Descent Minimization

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

Update:

\[ x_{i+1} = x_i - \alpha \nabla f(x_i) \]

\[ i=2 \]
Gradient Descent Minimization

\[ \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} \]

Update:

\[ x_{i+1} = x_i - \alpha \nabla f(x_i) \quad i=3 \]
Gradient Descent Minimization

Initialize: \( x_0 \)

Update: \( x_{i+1} = x_i - \alpha \nabla f(x_i) \)
Gradient Descent Minimization

Initialize: \( x_0 \)

Update: \( x_{i+1} = x_i - \alpha \nabla f(x_i) \)

We can always make it converge for a convex function.
Gradient Descent Minimization

Initialize: $\mathbf{x}_0$

Update: $\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i)$

We can always make it converge for a convex function.
Gradient Descent Minimization

Initialize: \( x_0 \)

Update: \( x_{i+1} = x_i - \alpha \nabla f(x_i) \)

We can always make it converge for a convex function.
XOR Problem
## XOR Problem

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
XOR Problem

\[ y = f(x_1, x_2) \]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
XOR Problem

\[ y = f(x_1, x_2) \]

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
XOR Problem

\[ y = f(x_1, x_2) \]

\[
\begin{array}{c|c|c}
 x_1 & x_2 & y \\
\hline
 0 & 0 & 0 \\
 0 & 1 & 1 \\
 1 & 0 & 1 \\
 1 & 1 & 0 \\
\end{array}
\]

\[ y = f(w_0, w_1, w_2) = \mathcal{H}(w_0 + w_1 x_1 + w_2 x_2) \]
XOR Problem

\[ y = f(x_1, x_2) \]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ y = f(w_0, w_1, w_2) = \mathcal{H}(w_0 + w_1 x_1 + w_2 x_2) \]
XOR Problem

\[ y = f(x_1, x_2) \]

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ y = f(w_0, w_1, w_2) = \mathcal{H}(w_0 + w_1 x_1 + w_2 x_2) \]
Regression

1. Least Squares fitting

2. Nonlinear error function and gradient descent

3. Perceptron training (simple neural network)
Lifting to Higher Dimensions
Lifting to Higher Dimensions

\[ g : X \rightarrow X' \]
Lifting to Higher Dimensions

\[ g : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ f_\theta(x) = \begin{cases} 
1 & \text{if } w g(x) + b \geq 0 \\
0 & \text{if } w g(x) + b < 0 
\end{cases} \]
Building A Complicated Function

Given a library of simple functions

\[ \sin(x), \log(x), \cos(x), x^3, \exp(x) \]

Compose into a complicated function

Slide Credit: Marc'Aurelio Ranzato, Yann LeCun
Building A Complicated Function

Given a library of simple functions

\[ \sin(x) \quad \log(x) \quad \cos(x) \quad x^3 \quad \exp(x) \]

Compose into a complicated function

**Idea 1: Linear Combinations**
- Boosting
- Kernels
- ...

\[ f(x) = \sum_{i} \alpha_i g_i(x) \]
Building A Complicated Function

Given a library of simple functions

$$\sin(x) \quad \log(x) \quad \cos(x) \quad x^3 \quad \exp(x)$$

Compose into a complicated function

$$f(x) = g_1(g_2(\ldots(g_n(x)\ldots)))$$

Idea 2: Compositions

- Decision Trees
- Deep Learning
Building A Complicated Function

Given a library of simple functions

\[
\sin(x) \quad \cos(x) \quad x^3 \quad \exp(x) \quad \log(x)
\]

Compose into a complicated function

\[
f(x) = \log(\cos(\exp(\sin^3(x))))
\]

Idea 2: Compositions

- Decision Trees
- Deep Learning
‘Neuron’: Cascade of Linear and Nonlinear Function

basic building block

[Diagram of a cascade of linear and nonlinear function blocks]
‘Neuron’: Cascade of Linear and Nonlinear Function

A basic building block of a neuron involves an axon from a neuron, dendrites, cell body, and an output axon. The input data $x_0$ is multiplied by a weight $w_0$ at the synapse, and the weighted sum $w_0x_0$ is then passed through an activation function $f$ of the form $f\left(\sum_i w_ix_i + b\right)$. Further inputs $w_1x_1$ and $w_2x_2$ are added to the weighted sum, completing the neuron's computation.
‘Neuron’: Cascade of Linear and Nonlinear Function

basic building block

\[ f \left( \sum_i w_i x_i + b \right) \]

Sigmoidal activation

\[ f(x) = \frac{1}{1 + e^{-x}} \]

Deep Learning for CG & Geometry Processing
Activation Functions

Step ("perceptron")
\[ g(a) = \begin{cases} 
0 & a < 0 \\
1 & a \geq 0 
\end{cases} \]

Sigmoidal ("logistic")
\[ g(a) = \frac{1}{1 + \exp(-a)} \]

Hyperbolic tangent
\[ g(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} \]

Rectified Linear Unit (RELU)
\[ g(a) = \max(0, a) \]

Image Credit: Olivier Grisel and Charles Ollion
Multi-Layer Perceptrons (~1985)

\[ u_i = g \left( \sum_{k \in \mathcal{N}(i)} w_{k,i} g \left( \sum_{m \in \mathcal{N}(k)} w_{m,k} u_m + b_k \right) + b_i \right) \]
Reminder: Non-linear Decision Boundaries
Reminder: Non-linear Decision Boundaries

\[ g : \mathbb{X} \rightarrow \mathbb{X}' \]
Reminder: Non-linear Decision Boundaries

\[ g : \mathbb{X} \rightarrow \mathbb{X}' \]

\[ f_\theta(x) = \begin{cases} 
1 & \text{if } w g(x) + b \geq 0 \\
0 & \text{if } w g(x) + b < 0
\end{cases} \]
Reminder: Non-linear Decision Boundaries

This is what the hidden layers should be doing!

\[
g : \mathbb{X} \longrightarrow \mathbb{X}'
\]

\[
f_\theta(x) = \begin{cases} 
1 & \text{if } w \ g(x) + b \geq 0 \\
0 & \text{if } w \ g(x) + b < 0
\end{cases}
\]
From Non-separable to Linearly Separable

Non-linearly separable data

Decision function

From Non-separable to Linearly Separable

Non-linearly separable data

Data mapped to learned space

Decision function

Hidden Layers: What do They Do?

Intuition: learn “dictionary” for objects

“Distributed representation”: represent (and classify) objects by mixing & mashing reusable parts

\[
\begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & \ldots
\end{bmatrix}
\] truck feature
Deep Learning ~ Hierarchical Composition

```
Low-Level Feature -> Mid-Level Feature -> High-Level Feature -> Trainable Classifier
```

“car”
Deep Learning ~ Hierarchical Composition

Low-Level Feature → Mid-Level Feature → High-Level Feature → Trainable Classifier → “car”

Deep Learning ~ Hierarchical Composition
Deep Learning ~ Hierarchical Composition

Deep Learning for CG & Geometry Processing
Deep Learning ~ Hierarchical Composition

- Low-Level Feature
- Mid-Level Feature
- High-Level Feature
- Trainable Classifier

"car"
MLP Demo: playground.tensorflow.org
Neural Network Training: Old and New Tricks

• Old
  • Backpropagation algorithm
  • Stochastic gradient, momentum, weight decay

• New
  • Dropout
  • Relu
  • Batch Norm(alization), GroupNorm, Spectral Normalization
  • Res(idual) Net(work)
Training Goal

Our network implements a parametric function:

\[ f_\theta : \mathbb{X} \rightarrow \mathbb{Y} \quad \hat{y} = f(x; \theta) \]
Our network implements a parametric function:

\[ f_\theta : \mathbb{X} \rightarrow \mathbb{Y} \quad \hat{y} = f(x; \theta) \]

During training, we search for parameters that minimize a loss:

\[ \min_\theta L_f(\theta) \]
Training Goal

Our network implements a parametric function:

\[ f_\theta : \mathbb{X} \rightarrow \mathbb{Y} \quad \hat{y} = f(x; \theta) \]

During training, we search for parameters that minimize a loss:

\[ \min_\theta L_f(\theta) \]

Example: L2 regression loss given target \((x^i, y^i)\) pairs:

\[ L_f(\theta) = \sum_i \| f(x^i; \theta) - y^i \|_2^2 \]
Multiple Local Minima: Based on Initialization

empirically all are almost equally good
All You Need is Gradients

Forward
All You Need is Gradients

Forward

Backward

\[
\frac{\partial L}{\partial X} \left\{ \frac{\partial Z}{\partial X}, \frac{\partial Z}{\partial \theta} \right\} \quad \frac{\partial L}{\partial Z}
\]
Chain Rule

Given \( y(x) \) and \( \frac{dL}{dy} \),
What is \( \frac{dL}{dx} \)?
Chain Rule

\[ \frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx} \]

Given \( y(x) \) and \( dL/dy \),
What is \( dL/dx \)?
‘Another Brick in the Wall’

Given \( y(x) \) and \( \frac{dL}{dy} \),

What is \( \frac{dL}{dx} \)?

\[
\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}
\]
‘Another Brick in the Wall’

Given \( y(x) \) and \( \frac{dL}{dy} \),

What is \( \frac{dL}{dx} \)?

\[
\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}
\]
Computation Graph and Automatic Differentiation
Multi-Layer Perceptrons (~1985)

\[ u_i = g \left( \sum_{k \in \mathcal{N}(i)} w_{k,i} g \left( \sum_{m \in \mathcal{N}(k)} w_{m,k} u_m + b_k \right) + b_i \right) \]
Multi-Layer Perceptrons (~1985)

Slide credit: G. Hinton
Multi-Layer Perceptrons (~1985)

Compare outputs with current answer to get error signal

Slide credit: G. Hinton
Multi-Layer Perceptrons (~1985)

- Compare outputs with current answer to get error signal.
- Back-propagate error signal to get derivatives for learning.

Slide credit: G. Hinton
Training Goal

Our network implements a parametric function:

\[ f_\theta : \mathbb{X} \rightarrow \mathbb{Y} \quad \hat{y} = f(x; \theta) \]

During training, we search for parameters that minimize a loss:

\[
\min_\theta L_f(\theta)
\]

Example: L2 regression loss given target \((x^i, y^i)\) pairs:

\[
L_f(\theta) = \sum_i \| f(x^i; \theta) - y^i \|^2
\]
A Neural Network for Multi-way Classification

\[ x_n \xrightarrow{V} a_n \xrightarrow{g} z_n \xrightarrow{W} b_n \xrightarrow{h} \hat{y}_n \]

**Inputs**

\[ x_{nD}, x_{ni}, x_{n1} \]

**Hidden layer**

\[ z_{n1}, \ldots, z_{nj}, \ldots, z_{nH} \]

**Outputs**

\[ y_{n1}, \ldots, y_{nk}, y_{nC} \]

Parameters:

\[ \theta = \{v, w\} \]
A Neural Network in Forward Mode

\[ a = Vx \]
A Neural Network in Forward Mode

\[ z = g(a) \]

\[ z_k = \frac{1}{1 + \exp(-a_k)} \]
A Neural Network in Forward Mode

\[ b = Wz \]
A Neural Network in Forward Mode

\[ \hat{y} = h(b) \]
Objective for Linear Regression

\[ h(b) = b \]

\[ \hat{y} = h(b) \quad y \]

Ground truth

\[ \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \]

L2 loss

\[ l(\hat{y}, y) = \sum_{c=1}^{C} (y_c - \hat{y}_c)^2 \]
Objective for Multi-class Classification

Softmax unit

\[ \hat{y}_k = \frac{\exp(b_k)}{\sum_{c=1}^{C} \exp(b_c)} \]

\[ \hat{y} = h(b) \]

\[
\begin{bmatrix}
0 \\
1 \\
0
\end{bmatrix}
\]

Ground truth

‘Cross-entropy’ loss

\[ l(\hat{y}, y) = \sum_{c=1}^{C} y_c \log(\hat{y}_c) \]
A Neural Network for Multi-way Classification

\[ x_n \xrightarrow{V} a_n \xrightarrow{g} z_n \xrightarrow{W} b_n \xrightarrow{h} \hat{y}_n \]

Inputs

\[ x_{nD}, x_{ni}, x_{n1} \]

Hidden layer

\[ z_{n1}, z_{nj}, z_{nH} \]

Outputs

\[ y_{n1}, y_{nk}, y_{nC} \]

Parameters:

\[ \theta = \{ v, w \} \]

Deep Learning for CG & Geometry Processing
Neural Network in Forward Mode: Recap

Network output:

$$\hat{y} = f(x; \mathbf{v}, \mathbf{w})$$

Loss (prediction error):

$$l(\hat{y}, y)$$

What we need to compute for gradient descent:

$$\frac{\partial l(\hat{y}, y)}{\partial v_i}$$

$$\frac{\partial l(\hat{y}, y)}{\partial w_j}$$
A Neural Network in Backward Mode

\[
\begin{bmatrix}
\vdots \\
\end{bmatrix}
\]
A Neural Network in Backward Mode

Hidden layer

\[ b = Wz \]

\[
\begin{bmatrix}
    y \\
    0 \\
    1 \\
    0
\end{bmatrix}
\]

\[ \frac{\partial l}{\partial w_{jk}} = \text{?} \]
A Neural Network in Backward Mode

\[ b = Wz \]

This we want

\[
\frac{\partial l}{\partial z_j} = ?
\]
Linear Layer in Forward Mode: All For One

\[ b_m = \sum_{h=1}^{H} z_h w_{h,m} \]
Linear Layer in Backward Mode: All For One

\[ b_m = \sum_{h=1}^{H} z_h w_{h,m} \]

\[ \frac{\partial L}{\partial b_m} \]
Linear Layer in Backward Mode: All For One

\[ b_m = \sum_{h=1}^{H} z_h w_{h,m} \]

\[ \frac{\partial L}{\partial b_m} \]

\[ \frac{\partial L}{\partial z_h} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial z_h} \]
Linear Layer in Backward Mode: All For One

\[ b_m = \sum_{h=1}^{H} z_h w_{h,m} \]

\[
\frac{\partial L}{\partial b_m} \]

\[
\frac{\partial L}{\partial z_h} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial z_h} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} w_{h,c}
\]
Linear Layer Parameters in Backward: 1-to-1

\[ b_m = \sum_{h=1}^{H} z_h w_{h,m} \]

\[ \frac{\partial L}{\partial b_m} \]
Linear Layer Parameters in Backward: 1-to-1

\[ b_m = \sum_{h=1}^{H} z_h w_{h,m} \]

\[ \frac{\partial L}{\partial b_m} \]

\[ \frac{\partial L}{\partial w_{h,m}} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial w_{h,m}} \]
Linear Layer Parameters in Backward: 1-to-1

\[ b_m = \sum_{h=1}^{H} z_h w_{h,m} \]

\[ \frac{\partial L}{\partial b_m} \]

\[ \frac{\partial L}{\partial w_{h,m}} = \sum_{c=1}^{C} \frac{\partial L}{\partial b_c} \cdot \frac{\partial b_c}{\partial w_{h,m}} = \frac{\partial L}{\partial b_m} z_h \]
A Neural Network in Backward Mode

This we want

\[ \frac{\partial l}{\partial w_{jk}} = \sum_m \frac{\partial l}{\partial b_m} \frac{\partial b_m}{\partial w_{jk}} \]

This we have

\[ b = Wz \]

This we computed

\[ y \left[ \begin{array}{c} 0 \\ 1 \\ 0 \end{array} \right] \]
A Neural Network in Backward Mode

\[ b = Wz \]

This we want

\[ \frac{\partial l}{\partial w_{jk}} = \sum_m \frac{\partial l}{\partial b_m} \frac{\partial b_m}{\partial w_{jk}} = \frac{\partial l}{\partial b} z_j \]

This we have

This we computed

\[ y \]

\[ \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \]
A Neural Network in Backward Mode

$$b = Wz$$

$$\frac{\partial l}{\partial z_j} = \sum_m \frac{\partial l}{\partial b_m} \frac{\partial b_m}{\partial z_j} = \sum_m \frac{\partial l}{\partial b_m} w_{j,m}$$
A Neural Network in Backward Mode

\[ z_k = \frac{1}{1 + \exp(-a_k)} \]

\[
\begin{bmatrix}
\vdots \\
x_{ni} & v_{ij} & z_{nj} & w_{jk} & x_{n1} \\
\vdots \\
z_{n1} & y_{n1} & y_{nk} & y_{nc} \\
\end{bmatrix}
\]

\[
y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}
\]
A Neural Network in Backward Mode

\[
z_k = \frac{1}{1 + \exp(-a_k)}
\]

\[
\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m} \frac{\partial z_m}{\partial a_k}
\]

[0 
 1 
 0]
A Neural Network in Backward Mode

\[
z_k = \frac{1}{1 + \exp(-a_k)}
\]

\[
\frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m} \frac{\partial z_m}{\partial a_k} = \frac{\partial l}{\partial z_n} g'(a_k)
\]

\[
y = \begin{bmatrix}
0 \\
1 \\
0
\end{bmatrix}
\]

Deep Learning for CG & Geometry Processing
A Neural Network in Backward Mode

\[ z_k = \frac{1}{1 + \exp(-a_k)} \]

\[ \frac{\partial l}{\partial a_k} = \sum_m \frac{\partial l}{\partial z_m} \frac{\partial z_m}{\partial a_k} = \frac{\partial l}{\partial z_k} g'(a_k) = \frac{\partial l}{\partial z_k} g(a_k)(1 - g(a_k)) \]

\[ y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \]
A Neural Network in Backward Mode

\[ a = \mathbf{V} \mathbf{x} \]

\[ z_k = \frac{1}{1 + \exp(-a_k)} \]

\[ y \left[ \begin{array}{c} 0 \\ 1 \\ 0 \end{array} \right] \]
A Neural Network in Backward Mode

\[ a = Vx \]

\[ z_k = \frac{1}{1 + \exp(-a_k)} \]

\[ \frac{\partial l}{\partial v_{ij}} = \sum_k \frac{\partial l}{\partial a_k} \frac{\partial a_k}{\partial v_{ij}} \]

Deep Learning for CG & Geometry Processing

Eurographics 2019
A Neural Network in Backward Mode

\[ a = Vx \]

\[ z_k = \frac{1}{1 + \exp(-a_k)} \]

\[ \frac{\partial l}{\partial v_{ij}} = \sum_k \frac{\partial l}{\partial a_k} \frac{\partial a_k}{\partial v_{ij}} = \frac{\partial l}{\partial a_j} x_i \]
Neural Network Training: Old and New Tricks

• Old
  • Backpropagation algorithm
  • Stochastic gradient, momentum, weight decay

• New
  • Dropout
  • Relu
  • Batch Norm(alization), GroupNorm, Spectral Normalization
  • Res(idual) Net(work)
Training Objective for N training samples

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} l(y^i, \hat{y}^i) + \sum_{l} \lambda_l \sum_{k,m} (W_{k,m}^l)^2 \]

Per-sample loss  Per-layer regularization
Training Objective for N training samples

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} l(y^i, \hat{y}^i) + \sum_{l} \lambda_l \sum_{k,m} (W^{l}_{k,m})^2 \]

Per-sample loss
Per-layer regularization

Gradient descent: \[ W_{t+1} = W_t - \epsilon \nabla_W L(W_t) \]
Training Objective for N training samples

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} l(y^i, \hat{y}^i) + \sum_l \lambda_l \sum_{k,m} (W^l_{k,m})^2$$

- Per-sample loss
- Per-layer regularization

Gradient descent:

$$W_{t+1} = W_t - \epsilon \nabla_W L(W_t)$$

(l,k,m) element of gradient vector:
Training Objective for N training samples

\[
L(W) = \frac{1}{N} \sum_{i=1}^{N} l(y^i, \hat{y}^i) + \sum_{l} \lambda_l \sum_{k,m} (W^l_{k,m})^2
\]

Per-sample loss \hspace{1cm} Per-layer regularization

Gradient descent: \( W_{t+1} = W_t - \epsilon \nabla_W L(W_t) \)

(\( l,k,m \)) element of gradient vector:

\[
\frac{\partial L}{\partial W^l_{k,m}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i)}{\partial W^l_{k,m}} + 2\lambda_l W^l_{k,m}
\]
Training Objective for N training samples

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} l(y^i, \hat{y}^i) + \sum_{l} \lambda_l \sum_{k,m} (W^l_{k,m})^2 \]

Per-sample loss \hspace{1cm} Per-layer regularization

Gradient descent: \[ W_{t+1} = W_t - \epsilon \nabla_W L(W_t) \]

\((l,k,m)\) element of gradient vector:

\[ \frac{\partial L}{\partial W^l_{k,m}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i)}{\partial W^l_{k,m}} + 2\lambda_l W^l_{k,m} \]

Back-prop for i-th example
Training Objective for N training samples

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} l(y^i, \hat{y}^i) + \sum \lambda_l \sum (W^l_{k,m})^2 \]

- **Per-sample loss**
- **Per-layer regularization**

Gradient descent:

\[ W_{t+1} = W_t - \epsilon \nabla W L(W_t) \]

(I,k,m) element of gradient vector:

\[ \frac{\partial L}{\partial W^l_{k,m}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i)}{\partial W^l_{k,m}} + 2\lambda_l W^l_{k,m} \]

If N=10^6, to update W once need to back-prop 10^6 times!
Regularization in SGD: Weight Decay

Gradient: \textbf{Batch}: [1..N]

\[
\frac{\partial L}{\partial W^l_{k,m}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i)}{\partial W^l_{k,m}} + 2\lambda_l W^l_{k,m}
\]
Regularization in SGD: Weight Decay

**Gradient:** \[ \frac{\partial L}{\partial W_{k,m}^l} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i)}{\partial W_{k,m}^l} + 2\lambda_l W_{k,m}^l \]

**Noisy (‘Stochastic’) Gradient:**

**Minibatch:** B elements  
\[ b(1), b(2), ..., b(B): \text{randomly} \text{ sampled from } [1,N] \]
Regularization in SGD: Weight Decay

Gradient: \( \text{Batch: } [1..N] \)

\[
\frac{\partial L}{\partial W^l_{k,m}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i)}{\partial W^l_{k,m}} + 2\lambda_l W^l_{k,m}
\]

Noisy (‘Stochastic’) Gradient: 

Minibatch: \( B \) elements

\( b(1), b(2), ..., b(B): \text{randomly sampled from } [1,N] \)

\[
\frac{\partial L}{\partial W^l_{k,m}} \approx \frac{1}{B} \sum_{i=1}^{B} \frac{\partial l(y^{b(i)}, \hat{y}^{b(i)})}{\partial W^l_{k,m}} + 2\lambda_l W^l_{k,m}
\]

Back-prop on minibatch
Regularization in SGD: Weight Decay

Gradient: Batch: [1..N]

\[
\frac{\partial L}{\partial W_{k,m}^l} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i)}{\partial W_{k,m}^l} + 2\lambda_l W_{k,m}^l
\]

Noisy (‘Stochastic’) Gradient:

Minibatch: B elements

b(1), b(2),..., b(B): randomly sampled from [1,N]
**Regularization in SGD: Weight Decay**

**Gradient:**  
Batch: [1..N]

\[
\frac{\partial L}{\partial W_{k,m}^l} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial l(y^i, \hat{y}^i)}{\partial W_{k,m}^l} + 2\lambda_l W_{k,m}^l
\]

**Noisy (‘Stochastic’) Gradient:**

Minibatch: B elements  
b(1), b(2),..., b(B): *randomly* sampled from [1,N]

\[
\frac{\partial L}{\partial W_{k,m}^l} \approx \frac{1}{B} \sum_{i=1}^{B} \frac{\partial l(y^{b(i)}, \hat{y}^{b(i)})}{\partial W_{k,m}^l} + 2\lambda_l W_{k,m}^l
\]

**Epoch:** N samples, N/B batches

Deep Learning for CG & Geometry Processing
Learning Rate

\[ W_{t+1} = W_t - \epsilon \nabla_W L(W_t) \]
(S)GD with Adaptable Stepsize

Too small: converge very slowly

Too big: overshoot and even diverge
(S)GD with Adaptable Stepsize

- Too small: converge very slowly
- Too big: overshoot and even diverge
- Reduce size over time
(S)GD with Adaptable Stepsize

Too small: converge very slowly

Too big: overshoot and even diverge

Reduce size over time

\[ \epsilon_t = \frac{C}{t} \]
Main idea: retain long-term trend of updates, drop oscillations

\[ \mathbf{W}_{t+1} = \mathbf{W}_t - \epsilon_t \nabla_{\mathbf{W}} L(\mathbf{W}_t) \]
(S)GD with Momentum

Main idea: retain long-term trend of updates, drop oscillations

\[ W_{t+1} = W_t - \epsilon_t \nabla W L(W_t) \]
(S)GD with Momentum

Main idea: retain long-term trend of updates, drop oscillations

\[
(S)\text{GD} \quad W_{t+1} = W_t - \epsilon_t \nabla_W L(W_t)
\]
**Main idea:** retain long-term trend of updates, drop oscillations

\[(S)GD\]  
\[W_{t+1} = W_t - \epsilon_t \nabla_W L(W_t)\]

\[(S)GD + \text{momentum}\]  
\[V_{t+1} = \mu V_t + (1 - \mu) \nabla_W L(W_t)\]  
\[W_{t+1} = W_t - \epsilon_t V_{t+1}\]
Step-size Selection & Optimizers

- Nesterov’s Accelerated Gradient (NAG)
- R-prop
- AdaGrad
- RMSProp
- AdaDelta
- Adam
- …
Course Information (slides/code/comments)

http://geometry.cs.ucl.ac.uk/dl_for_CG/