# Foundations of Deep Learning

Moustapha

Google AI

## The Ingredients

Learn a mapping from data $(X, Y)$ sampled from a distribution $D$:

- **Inputs**: $X \in \mathbb{R}^{n \times d}$, every $x_i \in \mathbb{R}^d$ represents a data point.
- **Targets** (labels): $Y \in \mathbb{R}^{n \times L}$, every $y_i \in \mathbb{R}^L$ is the target for $x_i$.
- Parameterized function, mapping from $X$ to $Y$, $f_W : \mathbb{R}^{n \times d} \to \mathbb{R}^{n \times L}$
- **Risk**, Measure of success a.k.a **Criteria**:

$$\mathcal{R}(f_W) = \mathbb{E}_{(x,y)\sim D}\ell(f_W(x), y)$$

Since we do not know the generating distribution $D$ we use the observed data $(X, Y)$ and perform **Empirical Risk Minimization:**

$$\tilde{\mathcal{R}}(f_W) = \sum_{i=1}^{n} \ell(f_W(x_i), y_i)$$

## The Ingredients

Learn a mapping from data $(X, Y)$ sampled from a distribution $D$:

- **Inputs**: $X \in \mathbb{R}^{n \times d}$, every $x_i \in \mathbb{R}^d$ represents a data point.
- **Targets** (labels): $Y \in \mathbb{R}^{n \times L}$, every $y_i \in \mathbb{R}^L$ is the target for $x_i$.
- Parameterized function, mapping from $X$ to $Y$, $f_W : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{n \times L}$
- **Risk**, Measure of success a.k.a **Criteria**:

$$\mathcal{R}(f_W) = \mathbb{E}_{(x,y)\sim D}\ell(f_W(x), y)$$

Since we do not know the generating distribution $D$ we use the observed data $(X, Y)$ and perform **Empirical Risk Minimization:**

$$\tilde{\mathcal{R}}(f_W) = \sum_{i=1}^{n} \ell(f_W(x_i), y_i)$$

**Deep Learning** $\rightarrow$ **When** $f_W$ **is a deep neural network**.

## The basics

Foundations of Deep Learning$\rightarrow$Shallow Learning!

## The basics

**Foundations of Deep Learning$\rightarrow$Shallow Learning**!

Linear Regression:

- Linear function: $f_W(X) = XW$
- Criteria is Mean Squared Error: convex $\rightarrow$ global minimum.

$$\min_W \ell(f_W(X), Y) = ||XW - Y||^2$$

## The basics

**Foundations of Deep Learning$\rightarrow$Shallow Learning!**

Linear Regression:

- Linear function: $f_W(X) = XW$
- Criteria is Mean Squared Error: convex $\rightarrow$ global minimum.

$$\min_W \ell(f_W(X), Y) = ||XW - Y||^2$$

**Solution:**

$$\begin{aligned} \nabla_W \ell(X, Y) &= 0 \\ X^T X \cdot W^* - XY &= 0 \\ \mathrm{W}^* &= (\mathrm{X}^\mathrm{T}\mathrm{X})^{-1}\mathrm{X}^\mathrm{T} \cdot \mathrm{Y} \end{aligned}$$

## The basics

**Foundations of Deep Learning$\rightarrow$Shallow Learning!**

Linear Regression:

- Linear function: $f_W(X) = XW$
- Criteria is Mean Squared Error: convex $\rightarrow$ global minimum.

$$\min_W \ell(f_W(X), Y) = ||XW - Y||^2$$

**Solution:**

$$\nabla_W \ell(X, Y) = 0$$
$$X^T X \cdot W^* - XY = 0$$
$$\mathbf{W}^* = (\mathbf{X}^\mathbf{T}\mathbf{X})^{-1}\mathbf{X}^\mathbf{T} \cdot \mathbf{Y}$$

**Problem 1**

- Existence: $(X^T X)$ not always invertible.

## The basics

Foundations of Deep Learning $\rightarrow$ Shallow Learning!

Linear Regression:

- Linear function: $f_W(X) = XW$
- Criteria is Mean Squared Error + $L_2$ Regularization:

$$\min_W \ell(f_W(X), Y) = ||XW - Y||^2 + \lambda \cdot ||W||_F^2$$

## The basics

Foundations of Deep Learning $\rightarrow$ Shallow Learning!

Linear Regression:

- Linear function: $f_W(X) = XW$
- Criteria is Mean Squared Error + $L_2$ Regularization:

$$\min_W \ell(f_W(X), Y) = ||XW - Y||^2 + \lambda \cdot ||W||_F^2$$

**Solution:**

$$\begin{aligned} \nabla_W \ell(X, Y) &= 0 \\ (X^T X + \lambda \cdot I) \cdot W^* - XY &= 0 \\ \mathrm{W}^* = (\mathrm{X}^\mathrm{T}\mathrm{X} + \lambda \cdot \mathbf{I})^{-1}\mathrm{X}^\mathrm{T} \cdot \mathrm{Y} & \end{aligned}$$

$\rightarrow$ Regularization helps with the conditioning.

## The basics

Foundations of Deep Learning $\rightarrow$ Shallow Learning!

Linear Regression:

- Linear function: $f_W(X) = XW$
- Criteria is Mean Squared Error + $L_2$ Regularization:

$$\min_W \ell(f_W(X), Y) = ||XW - Y||^2 + \lambda \cdot ||W||_F^2$$

**Solution:**

$$\begin{aligned} \nabla_W \ell(X, Y) &= 0 \\ (X^T X + \lambda \cdot I) \cdot W^* - XY &= 0 \\ \mathbf{W}^* = (\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda \cdot \mathbf{I})^{-1}\mathbf{X}^\mathsf{T} \cdot \mathbf{Y} \end{aligned}$$

$\rightarrow$ Regularization helps with the conditioning.

**Problems 2**

- Dimensionality: $X \in \mathbb{R}^{n \times d}$, big data and large dimension
  $\rightarrow$ Inversion is $O(n \cdot d^3)$.

## The basics

Foundations of Deep Learning $\rightarrow$ Shallow Learning!

Linear Regression:

- Linear function: $f_W(X) = XW$
- Criteria is Mean Squared Error + $L_2$ Regularization:

$$\min_W \ell(f_W(X), Y) = \sum_{i=1}^{n} (x_i \cdot W - y_i)^2 + \lambda \cdot ||W||_F^2$$

## The basics

**Foundations of Deep Learning $\rightarrow$ Shallow Learning!**

Linear Regression:

- Linear function: $f_W(X) = XW$
- Criteria is Mean Squared Error + $L_2$ Regularization:

$$\min_W \ell(f_W(X), Y) = \sum_{i=1}^{n} (x_i \cdot W - y_i)^2 + \lambda \cdot ||W||_F^2$$

**Solution:** Stochastic Gradient Descent (SGD) instead of closed-form.
Repeat until convergence:

- sample randomly $(x_i, y_i)$
- Perform a gradient update:

$$W_{t+1} \leftarrow W_t - \alpha \cdot \nabla_W \ell(x_i, y_i)$$

$\rightarrow$ SGD improves the scalability: every update is $O(d)$.

## Case study: The basics

**Foundations of Deep Learning** $\rightarrow$ **Shallow Learning**!

Another reason to like SGD: **Logistic Regression** ($Y \in \{+1, -1\}$):

- Linear function: $f_W(X) = XW$
- Logistic loss + $L_2$ Regularization:

$$\min_W \ell(f_W(X), Y) = \frac{1}{n}\sum_{i=1}^{n} \ln(1 + e^{-y_i \cdot f_W(x_i)}) + \lambda \cdot ||W||_F^2$$

**Problem**: No closed form solution.
**Solution:** Smooth convex optimization problem $\rightarrow$ SGD.
Repeat until convergence:

- sample randomly $(x_i, y_i)$
- Perform a gradient update:

$$W_{t+1} \leftarrow W_t - \alpha \cdot \nabla_W \ell(x_i, y_i)$$

$\rightarrow$ SGD allows numerical solutions.

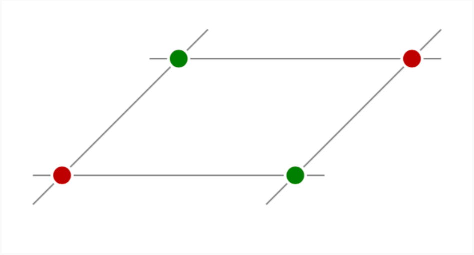
## Recap

For large scale problems we can use:

- **Logistic Regression** for classification.
- **Linear Regression** for regression.
- SGD for scalability and when closed form solutions do not exist
- Regularization for conditioning and stability.

**Question:** Why do we need **deep neural networks**?

## The XOR Problem: designing features

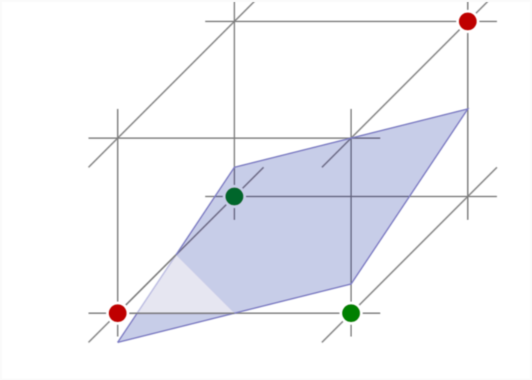


Sometimes the problem is not linearly separable …

## The XOR Problem

Maybe we can preprocess the data to make it separable:

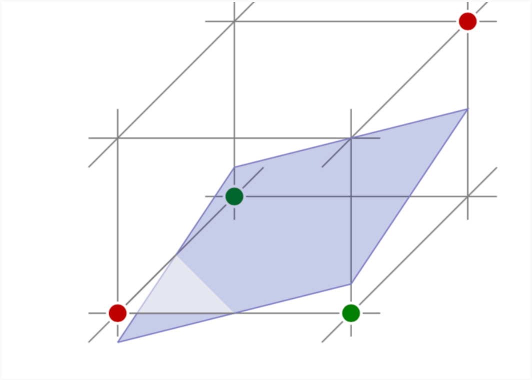
$$\Phi : (x_1, x_2) \to (x_1, x_2, x_3, x_4)$$



## The XOR Problem

Maybe we can preprocess the data to make it separable:

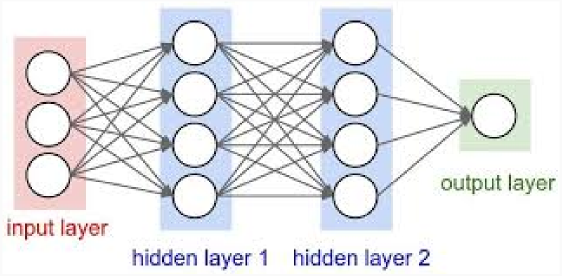
$$\Phi : (x_1, x_2) \to (x_1, x_2, x_3, x_4)$$



**Problem**: It is painful to search for ”the good Φ” for every problem ...
There are few kernel people in room, feel free to ask them :-) .

## Enters the Neural Network

We can learn the features automagically with a multi-layer NN:



Neural Networks are **Universal Approximators**!
With enough capacity you can approximate arbitrarily closely any function.

## Composing functions

A neural network is just a composition of functions:

$$f_W(x) = f_{w_1} \circ f_{w_2} \circ \cdots \circ f_{w_n}(x)$$

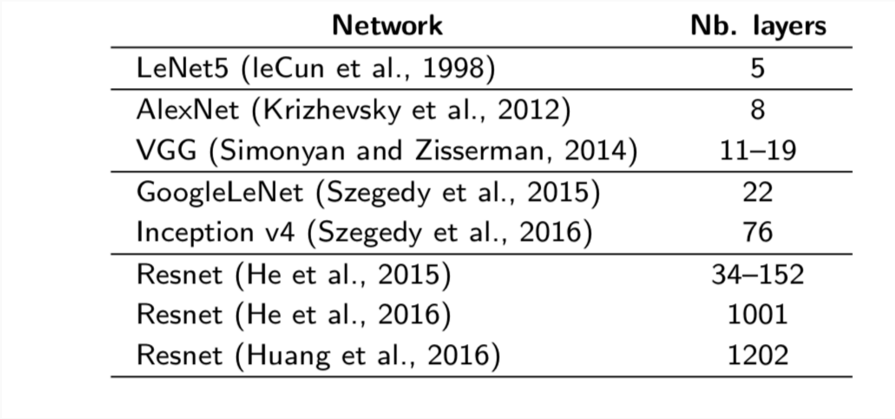
The i-th layer is represented by $f_{w_{i>1}} = \sigma(xw_i + b_i)$ where the **activation function** $\sigma$ is:

- **tanh**: $\sigma(x) = (e^x - e^{-x})/(e^x + e^{-x})$
- **sigmoid**: $\sigma(x) = 1/(1 + e^{-x})$

For classification, the last layer $f_{w_1}$ is a logistic regression.

## Go Deep or Go home!

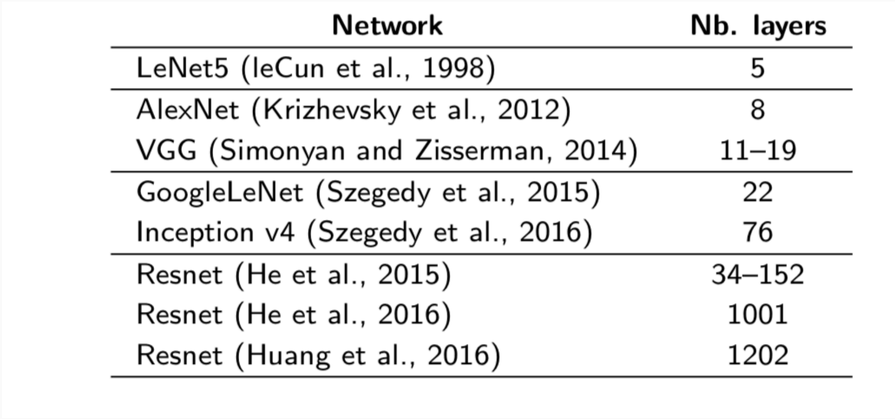
Stack as many layers as your harward can afford and Voila!



| Network | Nb. layers |
|---|---|
| LeNet5 (leCun et al., 1998) | 5 |
| AlexNet (Krizhevsky et al., 2012) | 8 |
| VGG (Simonyan and Zisserman, 2014) | 11–19 |
| GoogleLeNet (Szegedy et al., 2015) | 22 |
| Inception v4 (Szegedy et al., 2016) | 76 |
| Resnet (He et al., 2015) | 34–152 |
| Resnet (He et al., 2016) | 1001 |
| Resnet (Huang et al., 2016) | 1202 |

## Go Deep or Go home!

Stack as many layers as your harward can afford and Voila!



| Network | Nb. layers |
|---|---|
| LeNet5 (leCun et al., 1998) | 5 |
| AlexNet (Krizhevsky et al., 2012) | 8 |
| VGG (Simonyan and Zisserman, 2014) | 11–19 |
| GoogleLeNet (Szegedy et al., 2015) | 22 |
| Inception v4 (Szegedy et al., 2016) | 76 |
| Resnet (He et al., 2015) | 34–152 |
| Resnet (He et al., 2016) | 1001 |
| Resnet (Huang et al., 2016) | 1202 |

OK, this is non-convex and ”complex”, how do we train it ?

A large toolbox to alleviate the issues of non-convexity:

- SGD
- Regularization
- Many new tricks!

## Backprop

Let's consider a simple 2 layer network:

$$f_W(x) = f_{w_1} \circ f_{w_2}(x)$$

To perform SGD on our parameters $W = \{w_i\}_{i=1}^n$ we need to compute $\nabla_W \ell(f_w(x), y)$ for every $(x, y)$. The Chain rule gives use:

$$(f_{w_1} \circ f_{w_2})'(x) = f'_{w_2}(x) \cdot f'_{w_1}(f_{w_2}(x))$$

We have the simple procedure:

- Forward Pass: $f_{w_1} \circ f_{w_2}(x)$
- Backward Pass: Going from the output to the input, the gradient at every layer only depend on the gradients of the layer before and the activations at the current layer: Chain Rule.

## More Regularization: Dropout

A deep regularization method to prevent overfitting:

- Training: randomly zero-out features with probability p
- Test: Multiply activations by p

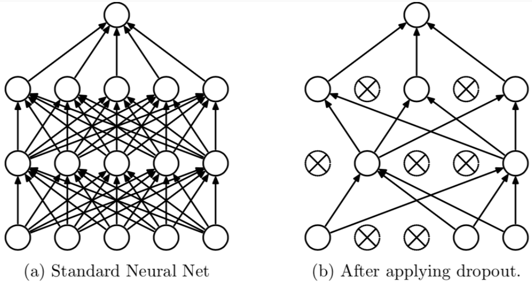
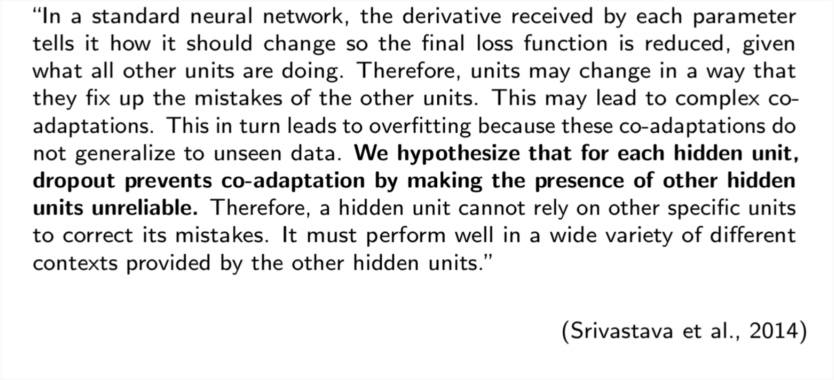


Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

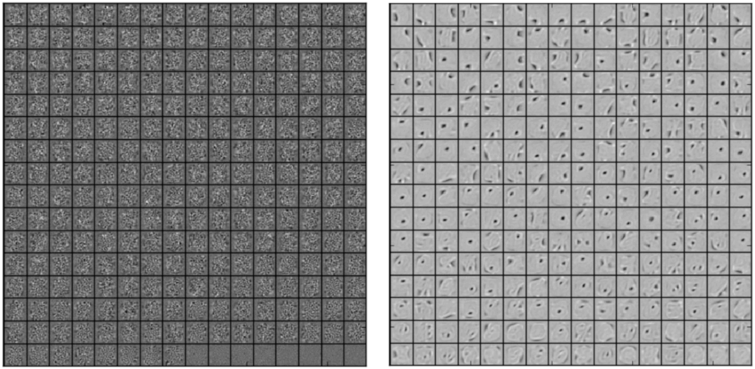
## Dropout



"In a standard neural network, the derivative received by each parameter tells it how it should change so the final loss function is reduced, given what all other units are doing. Therefore, units may change in a way that they fix up the mistakes of the other units. This may lead to complex co-adaptations. This in turn leads to overfitting because these co-adaptations do not generalize to unseen data. **We hypothesize that for each hidden unit, dropout prevents co-adaptation by making the presence of other hidden units unreliable.** Therefore, a hidden unit cannot rely on other specific units to correct its mistakes. It must perform well in a wide variety of different contexts provided by the other hidden units."

(Srivastava et al., 2014)

## Dropout



(a) Without dropout (b) Dropout with $p = 0.5$.

Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.

(Srivastava et al., 2014)

## More Tricks/Regularizers

Deep Learners are fancy:

- Batch-Normalization
- DropConnect
- Layer Normalization
- Spectral Normalization
- Mixup (make sure you try this one)
- ...

An active area of research.

## Improved Optimization: Momentum

SGD is sensitive to initialization and magnitude of the learning rate:

$$W_{t+1} \leftarrow W_t - \alpha \cdot \nabla_W \ell(x_i, y_i)$$

Use a momentum to add inertia in the choice of the direction:

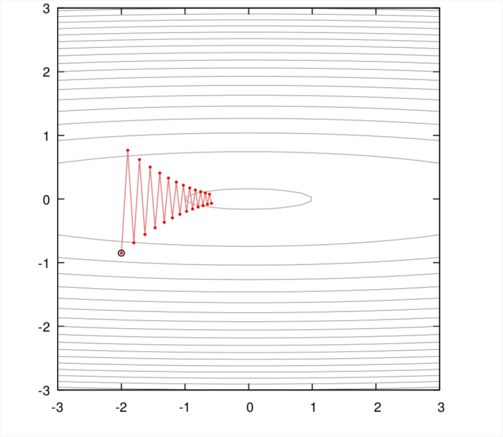
- $u_{t+1} \leftarrow \gamma \cdot u_t - \alpha \cdot \nabla_W \ell(x_i, y_i)$
- $W_{t+1} \leftarrow W_t - u_{t+1}$

When $\gamma = 0$, we recover vanilla SGD.
If $\gamma > 0$ momentum:

- accelerates if the gradient does not change much.
- dampens oscillations in narrow valleys

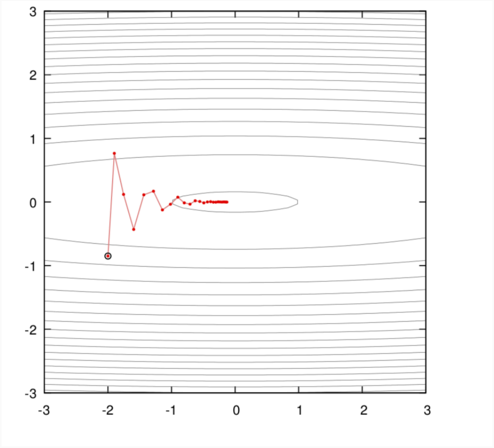
## No Momentum

Slower convergence, oscillations in narrow valleys.



## With Momentum

Faster convergence, no oscillations in narrow valleys.



## More Optimizers

Deep Learners are fancy:

- Adagrad
- Adam
- Adamax
- Nadam
- AdaDelta
- RMSProp
- Natural Gradient
- ...

## More Optimizers

Deep Learners are fancy:

- Adagrad
- Adam
- Adamax
- Nadam
- AdaDelta
- RMSProp
- Natural Gradient
- ...

Still an active area of research.

## New activation function

Sigmoid activation functions are prone vanishing gradient and make optimization more difficult. Use non-saturating activations:

- **tanh**: $\sigma(x) = (e^x - e^{-x})/(e^x + e^{-x})$
- **sigmoid**: $\sigma(x) = 1/(1 + e^{-x})$

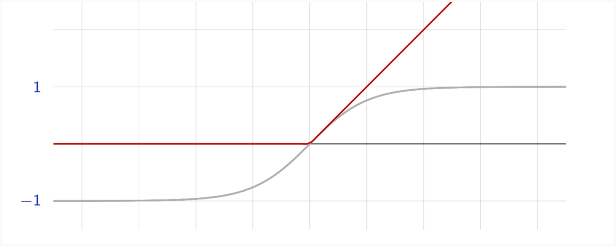
## New activation function

Sigmoid activation functions are prone vanishing gradient and make optimization more difficult. Use non-saturating activations:

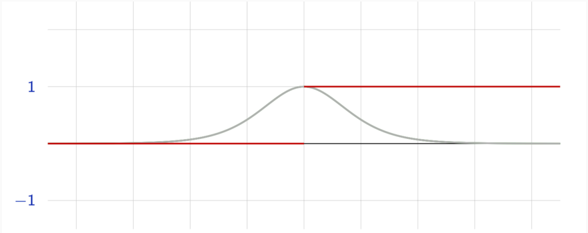
- **tanh**: $\sigma(x) = (e^x - e^{-x})/(e^x + e^{-x})$
- **sigmoid**: $\sigma(x) = 1/(1 + e^{-x})$
- Relu: $\sigma(x) = \max(0, x)$

## Relu vs Tanh

Relus are non-saturating.



Relus' gradient does not vanish



## Two Very Important Ingredients

- Automatic Differentiation.

TensorFlow

- GPUs

## Recap

In summary you need:

- Lots of data
- GPUs
- Software (e.g. TensorFlow, Pytorch)
- Backpropagation
- Optimizers
- Regularizers

## Recap

What I did not talk about:

- Convoluational Neural Networks (Vision)
- Recurrent Networks (Sequences)
- Embeddings (e.g. Text)
- Backpropagation

Questions?

## Backup slides

Sometimes, it is useful to add slides at the end of your presentation to refer to during audience questions.

The best way to do this is to include the `appendixnumberbeamer` package in your preamble and call `\appendix` before your backup slides.

METROPOLIS will automatically turn off slide numbering and progress bars for slides in the appendix.

## References i