

Challenges to Automatic Differentiation

Barak A. Pearlmutter

Hamilton Institute
NUI Maynooth
Co. Kildare
Ireland

2nd European Workshop on Automatic Differentiation

Problematic Topics for Current AD Systems

Problematic Topics for Current AD Systems

- ▶ Things we know how to do
 - ▶ High-level transformation of high-level routines
 - ▶ Iterate-to-fixedpoint loops
 - ▶ integrators: $\frac{d}{dx} \int_0^1 f(x, y) dy$
 - ▶ optimisers: $\frac{d}{dx} \operatorname{argmin}_y E(x, y)$
 - ▶ ODE solvers, PDE solvers
 - ▶ AD & just-in-time compilation & specialisation
 - ▶ Nested application of AD operators
 - ▶ “*Almost* a matter of running make.”
 - ▶ Modularity: *callee* derives (separation of concerns)
 - ▶ No need for constant manual validation (trust like GCC)

Problematic Topics for Current AD Systems

- ▶ Things we know how to do ... manually ...
 - ▶ High-level transformation of high-level routines
 - ▶ Iterate-to-fixedpoint loops
 - ▶ integrators: $\frac{d}{dx} \int_0^1 f(x, y) dy$
 - ▶ optimisers: $\frac{d}{dx} \operatorname{argmin}_y E(x, y)$
 - ▶ ODE solvers, PDE solvers
 - ▶ AD & just-in-time compilation & specialisation
 - ▶ Nested application of AD operators
 - ▶ “*Almost* a matter of running make.”
 - ▶ Modularity: *callee* derives (separation of concerns)
 - ▶ No need for constant manual validation (trust like GCC)

Problematic Topics for Current AD Systems

- ▶ Things we know how to do ... manually ...
 - ▶ High-level transformation of high-level routines
 - ▶ Iterate-to-fixedpoint loops
 - ▶ integrators: $\frac{d}{dx} \int_0^1 f(x, y) dy$
 - ▶ optimisers: $\frac{d}{dx} \operatorname{argmin}_y E(x, y)$ such that $c(x, y) = 0$
 - ▶ ODE solvers, PDE solvers
 - ▶ AD & just-in-time compilation & specialisation
 - ▶ Nested application of AD operators
 - ▶ “*Almost* a matter of running make.”
 - ▶ Modularity: *callee* derives (separation of concerns)
 - ▶ No need for constant manual validation (trust like GCC)

Problematic Topics for Current AD Systems

- ▶ Things we know how to do ... manually ...
 - ▶ High-level transformation of high-level routines
 - ▶ Iterate-to-fixedpoint loops
 - ▶ integrators: $\frac{d}{dx} \int_0^1 f(x, y) dy$
 - ▶ optimisers: $\frac{d}{dx} \operatorname{argmin}_y E(x, y)$ such that $c(x, y) = 0$
 - ▶ ODE solvers, PDE solvers
 - ▶ AD & just-in-time compilation & specialisation
 - ▶ Nested application of AD operators
 - ▶ “~~Almost~~ a matter of running make.”
 - ▶ Modularity: *callee* derives (separation of concerns)
 - ▶ No need for constant manual validation (trust like GCC)

Problematic Topics for Current AD Systems

- ▶ Things we know how to do ... manually ...
 - ▶ High-level transformation of high-level routines
 - ▶ Iterate-to-fixedpoint loops
 - ▶ integrators: $\frac{d}{dx} \int_0^1 f(x, y) dy$
 - ▶ optimisers: $\frac{d}{dx} \operatorname{argmin}_y E(x, y)$ such that $c(x, y) = 0$
 - ▶ ODE solvers, PDE solvers
 - ▶ AD & just-in-time compilation & specialisation
 - ▶ Nested application of AD operators
 - ▶ ~~“Almost~~ a matter of running make.”
 - ▶ Modularity: *callee* derives (separation of concerns)
 - ▶ No need for constant manual validation (trust like GCC)
- ▶ Those that seem like there should be a way to ...
 - ▶ Annealed gradient systems (*e.g.* stochastic Boltzmann Machine)
 - ▶ EM & Generalised EM models
 - ▶ The *kernel trick* (Support Vector Machines, Kernel PCA)
 - ▶ Data structures (auto-adaptive grids, polygonal surface approximations)

Simple Integrator

Define $\mathcal{I}_u\{g(x, u)\} = \int_0^1 g(x, u) \, du$

Simple Integrator

Define $\mathcal{I}_u\{g(x, u)\} = \int_0^1 g(x, u) \, du$

Sophisticated implementation, $\mathcal{I}_u\{g(x, u)\}$ evaluates $g(x, u)$ at irregular points u_i where $g(x, u)$ is wiggly in u .

Simple Integrator

Define $\mathcal{I}_u\{g(x, u)\} = \int_0^1 g(x, u) \, du$

Sophisticated implementation, $\mathcal{I}_u\{g(x, u)\}$ evaluates $g(x, u)$ at irregular points u_i where $g(x, u)$ is wiggly in u .

Ultimate output is weighted sum: $\sum_i \alpha_i g(x, u_i)$

Simple Integrator

Define $\mathcal{I}_u\{g(x, u)\} = \int_0^1 g(x, u) \, du$

Sophisticated implementation, $\mathcal{I}_u\{g(x, u)\}$ evaluates $g(x, u)$ at irregular points u_i where $g(x, u)$ is wiggly in u .

Ultimate output is weighted sum: $\sum_i \alpha_i g(x, u_i)$

Consider $\frac{d}{dx} \mathcal{I}_u\{g(x, u)\}$

Simple Integrator

Define $\mathcal{I}_u\{g(x, u)\} = \int_0^1 g(x, u) \, du$

Sophisticated implementation, $\mathcal{I}_u\{g(x, u)\}$ evaluates $g(x, u)$ at irregular points u_i where $g(x, u)$ is wiggly in u .

Ultimate output is weighted sum: $\sum_i \alpha_i g(x, u_i)$

Consider $\frac{d}{dx} \mathcal{I}_u\{g(x, u)\}$

Naive approach, current systems: $\sum_i \alpha_i \frac{d}{dx} g(x, u_i)$.

Simple Integrator

Define $\mathcal{I}_u\{g(x, u)\} = \int_0^1 g(x, u) \, du$

Sophisticated implementation, $\mathcal{I}_u\{g(x, u)\}$ evaluates $g(x, u)$ at irregular points u_i where $g(x, u)$ is wiggly in u .

Ultimate output is weighted sum: $\sum_i \alpha_i g(x, u_i)$

Consider $\frac{d}{dx} \mathcal{I}_u\{g(x, u)\}$

Naive approach, current systems: $\sum_i \alpha_i \frac{d}{dx} g(x, u_i)$.

Problem: $\frac{d}{dx} g(x, u)$ not necessarily wiggliest at the u_i .

Simple Integrator

Define $\mathcal{I}_u\{g(x, u)\} = \int_0^1 g(x, u) \, du$

Sophisticated implementation, $\mathcal{I}_u\{g(x, u)\}$ evaluates $g(x, u)$ at irregular points u_i where $g(x, u)$ is wiggly in u .

Ultimate output is weighted sum: $\sum_i \alpha_i g(x, u_i)$

Consider $\frac{d}{dx} \mathcal{I}_u\{g(x, u)\}$

Naive approach, current systems: $\sum_i \alpha_i \frac{d}{dx} g(x, u_i)$.

Problem: $\frac{d}{dx} g(x, u)$ not necessarily wiggliest at the u_i .

Better (more accurate) result: $\mathcal{I}_u\{\frac{d}{dx} g(x, u)\}$.

Allows sophisticated integration routine to do its job.

Simple Integrator

Define $\mathcal{I}_u\{g(x, u)\} = \int_0^1 g(x, u) du$

Sophisticated implementation, $\mathcal{I}_u\{g(x, u)\}$ evaluates $g(x, u)$ at irregular points u_i where $g(x, u)$ is wiggly in u .

Ultimate output is weighted sum: $\sum_i \alpha_i g(x, u_i)$

Consider $\frac{d}{dx} \mathcal{I}_u\{g(x, u)\}$

Naive approach, current systems: $\sum_i \alpha_i \frac{d}{dx} g(x, u_i)$.

Problem: $\frac{d}{dx} g(x, u)$ not necessarily wiggliest at the u_i .

Better (more accurate) result: $\mathcal{I}_u\{\frac{d}{dx} g(x, u)\}$.

Allows sophisticated integration routine to do its job.

Direct high-level AD transforms of $y = \mathcal{I}_u\{g(x, u)\}$

Forward: $\dot{y} = \mathcal{I}_u\{\overrightarrow{\mathcal{J}}_x\{g(x, u)\}(\dot{x})\}$

Reverse: $\dot{x} = \mathcal{I}_u\{\overleftarrow{\mathcal{J}}_x\{g(x, u)\}(\dot{y})\}$

Iterate-to-Fixedpoint Loop Notation: \mathcal{F}

When we write

$$y = \mathcal{F}_u^\varepsilon \{ g(x, u) \}$$

we mean

```
do {   $u_{\text{prev}} = u;$   
       $u = g(x, u);$  }  
until ( $u \approx_\varepsilon u_{\text{prev}}$ );  
 $y = u;$ 
```

Iterate-to-Fixedpoint Loop Notation: \mathcal{F}

When we write

$$y = \mathcal{F}_u^x \{ g(x, u) \}$$

we mean

```
do {   $u_{\text{prev}} = u;$   
       $u = g(x, u);$  }  
until ( $u \approx_{\epsilon} u_{\text{prev}}$ );  
 $y = u;$ 
```

AD Transforms of \mathcal{F}

Primal:

$$y = \mathcal{F}_u\{g(x, u)\}$$

AD Transforms of \mathcal{F}

Primal:

$$y = \mathcal{F}_u\{g(x, u)\}$$

Forward:

$$\dot{y} = \mathcal{F}_{\dot{u}}\{\overrightarrow{\mathcal{J}}_y\{g(x, y)\}\}(\dot{u}) + \overrightarrow{\mathcal{J}}_x\{g(x, y)\}(\dot{x})\}$$

AD Transforms of \mathcal{F}

Primal:

$$y = \mathcal{F}_u\{g(x, u)\}$$

Forward:

$$\dot{y} = \mathcal{F}_{\dot{u}}\{\overrightarrow{\mathcal{J}}_y\{g(x, y)\}\}(\dot{u}) + \overrightarrow{\mathcal{J}}_x\{g(x, y)\}(\dot{x})\}$$

Reverse:

$$\dot{x} = \overleftarrow{\mathcal{J}}_x\{g(x, y)\}(\mathcal{F}_{\dot{u}}\{\overleftarrow{\mathcal{J}}_y\{g(x, y)\}\}(\dot{u}) + \dot{y})\}$$

High-level AD of Optimisation

Fun derivation: commutative diagram, lift rule for \mathcal{F} .

$$\begin{array}{ccc} \operatorname{argmin}_{\mathbf{u}} g(\mathbf{x}, \mathbf{u}) & \xrightarrow{\vec{\mathcal{J}}} & \operatorname{argmin}_{\hat{\mathbf{u}}} \tilde{g}(\dots) \\ \text{gradient descent} \downarrow & & \text{corresponding minimisation} \uparrow \\ \mathcal{F}_{\mathbf{u}}\{\mathbf{u} - \eta \nabla_{\mathbf{u}} g(\mathbf{x}, \mathbf{u})\} & \xrightarrow{\vec{\mathcal{J}}} & \mathcal{F}_{\hat{\mathbf{u}}}\{\hat{\mathbf{u}} - \eta \nabla_{\hat{\mathbf{u}}} \tilde{g}(\dots)\} \end{array}$$

High-level AD of Optimisation: Formal Transforms

Primal:

$$y = \underset{u}{\operatorname{argmin}}\{g(x, u)\}$$

Forward:

$$\begin{aligned} \hat{y} = \underset{\hat{u}}{\operatorname{argmin}}\{ & \frac{1}{2} \overrightarrow{\mathcal{J}}_y\{\overrightarrow{\mathcal{J}}_y\{g(x, y)\}(\hat{u})\}(\hat{u}) \\ & + \overrightarrow{\mathcal{J}}_y\{\overrightarrow{\mathcal{J}}_x\{g(x, y)\}(\hat{x})\}(\hat{u}) \} \end{aligned}$$

Reverse:

$$\begin{aligned} \hat{x} = -\overleftarrow{\mathcal{J}}_x\{\overleftarrow{\mathcal{J}}_y\{g(x, y)\}(1)\}(\\ \underset{\hat{u}}{\operatorname{argmin}}\{\frac{1}{2}(\overleftarrow{\mathcal{J}}_y\{\overleftarrow{\mathcal{J}}_y\{g(x, y)\}(1)\}(\hat{u}))^\top \hat{u} - \hat{y}^\top \hat{u}\}) \end{aligned}$$

Modularity: Separation of Concerns

User's job: call optimisation routine where appropriate.

Implementer's job: write good optimisation routine.

Modularity: Separation of Concerns

User's job: call optimisation routine where appropriate.

Implementer's job: write good optimisation routine.

Modularity: either can make decisions (change mind) in isolation.

Modularity: Separation of Concerns

User's job: call optimisation routine where appropriate.

Implementer's job: write good optimisation routine.

Modularity: either can make decisions (change mind) in isolation.

Problem: in current systems, the *caller* must pass desired derivatives to optimisation routine.

Modularity: Separation of Concerns

User's job: call optimisation routine where appropriate.

Implementer's job: write good optimisation routine.

Modularity: either can make decisions (change mind) in isolation.

Problem: in current systems, the *caller* must pass desired derivatives to optimisation routine.

Solution: the *callee* should be able to get whatever derivatives are needed (do some ADing) without the caller's cooperation.

Modularity: Separation of Concerns

User's job: call optimisation routine where appropriate.

Implementer's job: write good optimisation routine.

Modularity: either can make decisions (change mind) in isolation.

Problem: in current systems, the *caller* must pass desired derivatives to optimisation routine.

Solution: the *callee* should be able to get whatever derivatives are needed (do some ADing) without the caller's cooperation.

Also creates additional opportunities for optimisation.

Nested Application of AD Operators

Users should be able to write code compositionally without bumping against implementation restrictions.

If

$$\operatorname{argmin} : (\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n$$

is defined, then users should be able to confidently define

$$\operatorname{argmax}(f : \mathbb{R}^n \rightarrow \mathbb{R}) = \operatorname{argmin}(\lambda x \ -f(x))$$

$$\max(f) = f(\operatorname{argmax}(f))$$

$$\begin{aligned} \operatorname{saddlepoint}(f : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}) \\ = \max(\lambda x f(x, (\operatorname{argmin}(\lambda y f(x, y)))))) \end{aligned}$$

Now for something we *really* don't know how to do

Now for something we *really* don't know how to do

Kernel AD

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Consider optimisation of $\langle E(\mathbf{w}, \mathbf{x}_i, y_i) \rangle_i$

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Consider stochastic optimisation of $\langle E(\mathbf{w}, \mathbf{x}_i, y_i) \rangle_i$

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Consider stochastic optimisation of $\langle E(\mathbf{w}, \mathbf{x}_i, y_i) \rangle_i$ via small $\eta > 0$

do { choose i ; update $\mathbf{w} += -\eta \nabla_{\mathbf{w}} E(\mathbf{w}, \mathbf{x}_i, y_i)$ }
until convergence of \mathbf{w}

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Consider stochastic optimisation of $\langle E(\mathbf{w}, \mathbf{x}_i, y_i) \rangle_i$ via small $\eta > 0$

do { choose i ; update $\mathbf{w} += -\eta(\mathbf{w} \cdot \mathbf{x}_i - y_i)\mathbf{x}_i$ }
until convergence of \mathbf{w}

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Consider stochastic optimisation of $\langle E(\mathbf{w}, \mathbf{x}_i, y_i) \rangle_i$ via small $\eta > 0$

do { choose i ; update $\mathbf{w} += -\eta(\mathbf{w} \cdot \mathbf{x}_i - y_i)\mathbf{x}_i$ }
until convergence of \mathbf{w}

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Consider stochastic optimisation of $\langle E(\mathbf{w}, \mathbf{x}_i, y_i) \rangle_i$ via small $\eta > 0$

do { choose i ; update $\mathbf{w} += -\eta(\mathbf{w} \cdot \mathbf{x}_i - y_i)\mathbf{x}_i$ }
until convergence of \mathbf{w}

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Transform algorithm to use this dual representation:

do { choose i ; update $\mathbf{w} += -\eta(\mathbf{w} \cdot \mathbf{x}_i - y_i)\mathbf{x}_i$ }
until convergence of \mathbf{w}

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Consider stochastic optimisation of $\langle E(\mathbf{w}, \mathbf{x}_i, y_i) \rangle_i$ via small $\eta > 0$

do { choose i ; update $\mathbf{w} += -\eta(\mathbf{w} \cdot \mathbf{x}_i - y_i)\mathbf{x}_i$ }
until convergence of \mathbf{w}

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Transform algorithm to use this dual representation:

do { choose i ; update $\mathbf{w} += -\eta(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i - y_i)\mathbf{x}_i$ }
until convergence of \mathbf{w}

Simple Kernelisable Algorithm

LMS (Least Mean Squares)

Let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$

Let \mathbf{w} be a weight vector

Define $E(\mathbf{w}, \mathbf{x}, y) = \frac{1}{2} \|\mathbf{w} \cdot \mathbf{x} - y\|^2$

Consider stochastic optimisation of $\langle E(\mathbf{w}, \mathbf{x}_i, y_i) \rangle_i$ via small $\eta > 0$

do { choose i ; update $\mathbf{w} += -\eta(\mathbf{w} \cdot \mathbf{x}_i - y_i)\mathbf{x}_i$ }
until convergence of \mathbf{w}

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Transform algorithm to use this dual representation:

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

Kernelisation

Algorithm uses input data (in \mathbb{R}^n) inside dot products:

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

and can therefore be kernelised.

Kernelisation

Algorithm uses input data (in \mathbb{R}^n) inside dot products:

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

and can therefore be kernelised.

Map each $\mathbf{x}_i \mapsto \phi(\mathbf{x}_i)$ where $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{10^{100}}$ and $n \ll 10^{100}$ but
 $K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ is fast to calculate anyway.

Kernelisation

Algorithm uses input data (in \mathbb{R}^n) inside dot products:

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

and can therefore be kernelised.

Map each $\mathbf{x}_i \mapsto \phi(\mathbf{x}_i)$ where $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{10^{100}}$ and $n \ll 10^{100}$ but
 $K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ is fast to calculate anyway.

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

Kernelisation

Algorithm uses input data (in \mathbb{R}^n) inside dot products:

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

and can therefore be kernelised.

Map each $\mathbf{x}_i \mapsto \phi(\mathbf{x}_i)$ where $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{10^{100}}$ and $n \ll 10^{100}$ but
 $K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ is fast to calculate anyway.

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

Can pre-compute $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$

Kernelisation

Algorithm uses input data (in \mathbb{R}^n) inside dot products:

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

and can therefore be kernelised.

Map each $\mathbf{x}_i \mapsto \phi(\mathbf{x}_i)$ where $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{10^{100}}$ and $n \ll 10^{100}$ but
 $K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ is fast to calculate anyway.

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

Can pre-compute $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ and throw away the \mathbf{x}_i

Kernelisation

Algorithm uses input data (in \mathbb{R}^n) inside dot products:

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

and can therefore be kernelised.

Map each $\mathbf{x}_i \mapsto \phi(\mathbf{x}_i)$ where $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{10^{100}}$ and $n \ll 10^{100}$ but
 $K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ is fast to calculate anyway.

do { choose i ; update $\alpha_i += -\eta(\sum_j \alpha_j K(\mathbf{x}_j, \mathbf{x}_i) - y_i)$ }
until convergence of $\alpha_1, \dots, \alpha_p$

Can pre-compute $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ and throw away the \mathbf{x}_i (except \mathbf{x}_i for which $\alpha_i \neq 0$ after convergence, assuming you plan to use it on off-sample inputs later.)

Digression: Sample Kernels

Digression: Sample Kernels

Polynomial kernels: consider all quadratic terms, so $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{n^2}$

Digression: Sample Kernels

Polynomial kernels: consider all quadratic terms, so $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{n^2}$

$$\phi(\mathbf{x}) = (x_1^2, \dots, x_n^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \dots, \sqrt{2}x_{n-1}x_n)$$

$$\begin{aligned}K(\mathbf{x}, \mathbf{y}) &= \phi(\mathbf{x}) \cdot \phi(\mathbf{y}) \\&= x_1^2y_1^2 + \dots + x_n^2y_n^2 \\&\quad + 2x_1x_2y_1y_2 + 2x_1x_3y_1y_3 + \dots + 2x_{n-1}x_ny_{n-1}y_n \\&= (\mathbf{x} \cdot \mathbf{y})^2\end{aligned}$$

Digression: Sample Kernels

Polynomial kernels: consider all quadratic terms, so $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{n^2}$

$$\phi(\mathbf{x}) = (x_1^2, \dots, x_n^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \dots, \sqrt{2}x_{n-1}x_n)$$

$$\begin{aligned}K(\mathbf{x}, \mathbf{y}) &= \phi(\mathbf{x}) \cdot \phi(\mathbf{y}) \\&= x_1^2y_1^2 + \dots + x_n^2y_n^2 \\&\quad + 2x_1x_2y_1y_2 + 2x_1x_3y_1y_3 + \dots + 2x_{n-1}x_ny_{n-1}y_n \\&= (\mathbf{x} \cdot \mathbf{y})^2\end{aligned}$$

Generalises: can get **all** terms up to order m , so
 $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{O(n^m)}$, using $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^m$

Digression: Sample Kernels

Polynomial kernels: consider all quadratic terms, so $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{n^2}$

$$\phi(\mathbf{x}) = (x_1^2, \dots, x_n^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \dots, \sqrt{2}x_{n-1}x_n)$$

$$\begin{aligned}K(\mathbf{x}, \mathbf{y}) &= \phi(\mathbf{x}) \cdot \phi(\mathbf{y}) \\&= x_1^2y_1^2 + \dots + x_n^2y_n^2 \\&\quad + 2x_1x_2y_1y_2 + 2x_1x_3y_1y_3 + \dots + 2x_{n-1}x_ny_{n-1}y_n \\&= (\mathbf{x} \cdot \mathbf{y})^2\end{aligned}$$

Generalises: can get **all** terms up to order m , so

$$\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{O(n^m)}, \text{ using } K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^m$$

Hilbert space: roughly $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^\infty$

$$K(\mathbf{x}, \mathbf{y}) = \exp -\|\mathbf{x} - \mathbf{y}\|^2$$

Digression: Sample Kernels

Polynomial kernels: consider all quadratic terms, so $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{n^2}$

$$\phi(\mathbf{x}) = (x_1^2, \dots, x_n^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \dots, \sqrt{2}x_{n-1}x_n)$$

$$\begin{aligned}K(\mathbf{x}, \mathbf{y}) &= \phi(\mathbf{x}) \cdot \phi(\mathbf{y}) \\&= x_1^2y_1^2 + \dots + x_n^2y_n^2 \\&\quad + 2x_1x_2y_1y_2 + 2x_1x_3y_1y_3 + \dots + 2x_{n-1}x_ny_{n-1}y_n \\&= (\mathbf{x} \cdot \mathbf{y})^2\end{aligned}$$

Generalises: can get **all** terms up to order m , so

$$\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{O(n^m)}, \text{ using } K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^m$$

Hilbert space: roughly $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^\infty$

$$K(\mathbf{x}, \mathbf{y}) = \exp -\|\mathbf{x} - \mathbf{y}\|^2$$

Mercer's Theorem: If $\mathbf{K} = (K(\mathbf{x}_i, \mathbf{x}_j))_{ij}$ is non-negative definite then

$$\exists \phi \text{ such that } K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$$

Kernels and Modularity: If K is a kernel then cK is a kernel, $c \geq 0$.

If K_1 and K_2 are kernels then $K_1 + K_2$ is a kernel.

Other Kernels: \exists sensible K for written text, biosequences, bitmaps,

trees, etc.

Digression: Sample Kernels

Polynomial kernels: consider all quadratic terms, so $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{n^2}$

$$\phi(\mathbf{x}) = (x_1^2, \dots, x_n^2, \sqrt{2}x_1x_2, \sqrt{2}x_1x_3, \dots, \sqrt{2}x_{n-1}x_n)$$

$$\begin{aligned}K(\mathbf{x}, \mathbf{y}) &= \phi(\mathbf{x}) \cdot \phi(\mathbf{y}) \\&= x_1^2y_1^2 + \dots + x_n^2y_n^2 \\&\quad + 2x_1x_2y_1y_2 + 2x_1x_3y_1y_3 + \dots + 2x_{n-1}x_ny_{n-1}y_n \\&= (\mathbf{x} \cdot \mathbf{y})^2\end{aligned}$$

Generalises: can get **all** terms up to order m , so

$$\phi : \mathbb{R}^n \rightarrow \mathbb{R}^{O(n^m)}, \text{ using } K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + 1)^m$$

Hilbert space: roughly $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^\infty$

$$K(\mathbf{x}, \mathbf{y}) = \exp -\|\mathbf{x} - \mathbf{y}\|^2$$

Mercer's Theorem: If $\mathbf{K} = (K(\mathbf{x}_i, \mathbf{x}_j))_{ij}$ is non-negative definite then

$$\exists \phi \text{ such that } K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$$

Kernels and Modularity: If K is a kernel then cK is a kernel, $c \geq 0$.

If K_1 and K_2 are kernels then $K_1 + K_2$ is a kernel.

Other Kernels: \exists sensible K for written text, biosequences, bitmaps,

trees, etc. But I digress.

Kernel AD Fails

AD takes the gradient in the wrong space, in \mathbb{R}^n (input space) instead of $\mathbb{R}^{10^{100}}$ (feature space).

AD therefore changes all α s on each step, instead of only α_i .

And it doesn't even change them the right amount.

Conclusions

- ▶ AD has issues which impede its practical application.
- ▶ Treating library functions as primitives may help.
- ▶ Higher-order functions (operators) yield nice equations.
- ▶ APIs are important; allowing and exporting good APIs might help AD attract a non-specialist user community.
- ▶ The kernel trick is really cool, but ...

Conclusions

- ▶ AD has issues which impede its practical application.
- ▶ Treating library functions as primitives may help.
- ▶ Higher-order functions (operators) yield nice equations.
- ▶ APIs are important; allowing and exporting good APIs might help AD attract a non-specialist user community.
- ▶ The kernel trick is really cool, but ...
no one has a clue how to kernelise AD.

Conclusions

- ▶ AD has issues which impede its practical application.
- ▶ Treating library functions as primitives may help.
- ▶ Higher-order functions (operators) yield nice equations.
- ▶ APIs are important; allowing and exporting good APIs might help AD attract a non-specialist user community.
- ▶ The kernel trick is really cool, but ...
no one has a clue how to kernelise AD.
- ▶ Secret weapon: **programming language theory**.

Things Jeff knows how to do automatically

- ▶ High-level transformation of high-level routines: fixedpoint iterators, optimisers, integrators, ODE solvers, PDE solvers, ...
- ▶ AD & just-in-time compilation & specialisation
- ▶ Nested application of AD operators
- ▶ “A matter of running make.”
- ▶ Modularity: *callee* derives (separation of concerns)
- ▶ No need for constant manual validation (trust like GCC)

thank you

Linear Threshold: Perceptron Learning Rule

let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, with $y_i \in \{-1, +1\}$

let \mathbf{w} be a weight vector, b be a scalar bias, and $\eta > 0$ be small

Repeat

 choose i

 if $(\mathbf{w} \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b

Linear Threshold: Perceptron Learning Rule

let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, with $y_i \in \{-1, +1\}$

let \mathbf{w} be a weight vector, b be a scalar bias, and $\eta > 0$ be small

Repeat

 choose i

 if $(\mathbf{w} \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Linear Threshold: Perceptron Learning Rule

let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, with $y_i \in \{-1, +1\}$

let \mathbf{w} be a weight vector, b be a scalar bias, and $\eta > 0$ be small

Repeat

 choose i

 if $(\mathbf{w} \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Transform algorithm to use this dual representation:

Repeat

 choose i

 if $(\mathbf{w} \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b .

Linear Threshold: Perceptron Learning Rule

let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, with $y_i \in \{-1, +1\}$

let \mathbf{w} be a weight vector, b be a scalar bias, and $\eta > 0$ be small

Repeat

 choose i

 if $(\mathbf{w} \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Transform algorithm to use this dual representation:

Repeat

 choose i

 if $(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b .

Linear Threshold: Perceptron Learning Rule

let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, with $y_i \in \{-1, +1\}$

let \mathbf{w} be a weight vector, b be a scalar bias, and $\eta > 0$ be small

Repeat

 choose i

 if $(\mathbf{w} \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Transform algorithm to use this dual representation:

Repeat

 choose i

 if $(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\alpha_i += \eta y_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b .

Linear Threshold: Perceptron Learning Rule

let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, with $y_i \in \{-1, +1\}$

let \mathbf{w} be a weight vector, b be a scalar bias, and $\eta > 0$ be small

Repeat

 choose i

 if $(\mathbf{w} \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

until convergence of \mathbf{w} and b

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Transform algorithm to use this dual representation:

Repeat

 choose i

 if $(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\alpha_i += \eta y_i$ and $b += \eta y_i$

until convergence of $\alpha_1, \dots, \alpha_p$ and b .

Linear Threshold: Perceptron Learning Rule

let $(\mathbf{x}_1 \mapsto y_1, \dots, \mathbf{x}_p \mapsto y_p)$ be a training set, with $y_i \in \{-1, +1\}$

let \mathbf{w} be a weight vector, b be a scalar bias, and $\eta > 0$ be small

Repeat

 choose i

 if $(\mathbf{w} \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\mathbf{w} += \eta y_i \mathbf{x}_i$ and $b += \eta y_i$

 until convergence of \mathbf{w} and b

Represent \mathbf{w} using $\alpha_1, \dots, \alpha_p$, as $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$

Transform algorithm to use this dual representation:

Repeat

 choose i

 if $(\sum_j \alpha_j \mathbf{x}_j \cdot \mathbf{x}_i + b)y_i \leq 0$

 update $\alpha_i += \eta y_i$ and $b += \eta y_i$

 until convergence of $\alpha_1, \dots, \alpha_p$ and b .

sparsity: if never get \mathbf{x}_i wrong, then $\alpha_i = 0$