NB: This lecture should be read in conjunction with the tutorial at:

Learn how to

1) Build (high-dimensional) models of/for neural & behavioral data

\[ \text{output} = F_{[\text{params}]}(\text{input}) \]

formulate \( F \) mathematically on a computer
Learn how to

1) Build (high-dimensional) models of/for neural & behavioral data

\[ \text{output} = F_{[\text{params}]}(\text{input}) \]

formulate \( F \) mathematically on a computer

2) Train such models

\[ \text{data_slice}_2 = F_{[\text{params}]}(\text{data_slice}_1) \]

determine \( \text{params} \) from pair of linked data tensors

Motivation
Learn how to

1) Build (high-dimensional) models of/for neural & behavioral data

\[ \text{output} = F_{\text{params}}(\text{input}) \]

- formulate $F$ mathematically on a computer

2) Train such models

\[ \text{data\_slice\_2} = F_{\text{params}}(\text{data\_slice\_1}) \]

- determine $\text{params}$ from pair of linked data tensors

3) Evaluate and compare such models

\[ \_\text{data\_slice\_2} = F_{\text{params}}(\text{new\_data\_slice\_1}) \]

- this one you’ve already been seeing for several weeks

Motivation
**Key fact:** Amazingly enough, there’s a generic “one-size-fits-all” method— if sometimes suboptimal — for building and training models.
Optimization

loss

parameter space
Optimization

option 1: try many options …

![Diagram showing the parameter space with loss vs. loss(θ₀)]
Optimization

option 1: try many options … pick the best

\[ \text{loss}(\theta_0) \]

\( \theta_0 \) parameter space
Optimization

option 1: try many options … pick the best

```
loss

loss(\theta_0)

\theta_0

parameter space
```

pros: guaranteed to eventually find best minimum; numerically stable

con: infeasible. exhausting. (takes forever)
option 2: start somewhere

![Diagram showing optimization in parameter space with loss function and point $\theta_0$.](image)
option 2: start somewhere, try nearby options
option 2: start somewhere, try nearby options, pick the best, iterate
Optimization

option 2: start somewhere, try nearby options, pick the best, iterate

“Local” derivative-free methods

pro: numerically stable

con: not guaranteed to work, still potentially inefficient
option 2: start somewhere, try nearby options, pick the best, iterate

“Local” derivative-free methods

con: not guaranteed to work because of local minima
option 2: start somewhere, try nearby options, pick the best, iterate

“Local” derivative-free methods

con: not guaranteed to work because of local minima
option 2: start somewhere, try nearby options, pick the best, iterate

“Local” derivative-free methods

con: not guaranteed to work because of local minima
Optimization

option 3: follow the curve

loss

loss(\theta_0)

\theta_0

parameter space
option 3: follow the (negative of) the loss gradient
option 3: follow the (negative of) the loss gradient

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]
option 3: follow the (negative of) the loss gradient

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta \propto - \left. \frac{\partial [\text{loss}(\theta)]}{\partial \theta} \right|_{\theta = \theta_0} \]
option 3: follow the (negative of) the loss gradient

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta \propto - \frac{\partial [\text{loss}(\theta)]}{\partial \theta} \bigg|_{\theta=\theta_0} \]

pros: efficient
option 3: follow the (negative of) the loss gradient

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta \propto - \frac{\partial [\text{loss}(\theta)]}{\partial \theta} \bigg|_{\theta = \theta_0} \]

pros: efficient

con: local minima problem; not necessarily numerically stable
Optimization

option 3: follow the (negative of) the loss gradient

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta \propto -\left. \frac{\partial \text{loss}(\theta)}{\partial \theta} \right|_{\theta=\theta_0} \]

pros: efficient

con: local minima problem; not necessarily numerically stable

we'll see more on this topic shortly
option 3: follow the (negative of) the loss gradient

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta \propto - \frac{\partial [\text{loss}(\theta)]}{\partial \theta} \bigg|_{\theta=\theta_0} \]

ALSO:

1) loss needs to be differentiable (or close to it) in parameters
option 3: follow the (negative of) the loss gradient

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta \propto -\frac{\partial \text{loss}(\theta)}{\partial \theta} \bigg|_{\theta=\theta_0} \]

**ALSO:**

1) loss needs to be differentiable (or close to it) in parameters
2) you actually have to compute the derivative
Optimization

Option 3: follow the (negative of) the loss gradient

Ensuring (1) and getting (2) by hand is annoying. TensorFlow is a tool that makes it easy to do this procedure automatically and generically.

\[ \Delta \theta \propto -\left. \frac{\partial \text{loss}(\theta)}{\partial \theta} \right|_{\theta=\theta_0} \]

**ALSO:**

1) loss needs to be differentiable (or close to it) in parameters
2) you actually have to compute the derivative
Gradient Descent

[Ipynb: Gradient Descent]
Main problem with gradient-based optimization: instability

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta \propto -\left. \frac{\partial [\text{loss}(\theta)]}{\partial \theta} \right|_{\theta=\theta_0} \]
Main problem with gradient-based optimization: instability

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta \propto - \frac{\partial [\text{loss}(\theta)]}{\partial \theta} \bigg|_{\theta = \theta_0} \]
Gradient Descent

Main problem with gradient-based optimization: instability

$$\theta_0 \rightarrow \theta_0 + \Delta \theta$$

$$\Delta \theta \propto - \frac{\partial [\text{loss}(\theta)]}{\partial \theta} \bigg|_{\theta=\theta_0}$$
Gradient Descent

Main problem with gradient-based optimization: instability

\[
\theta_0 \rightarrow \theta_0 + \Delta \theta
\]

\[
\Delta \theta \propto - \frac{\partial [\text{loss}(\theta)]}{\partial \theta} \bigg|_{\theta=\theta_0}
\]

simple solution: take smaller steps
Gradient Descent

Main problem with gradient-based optimization: instability

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta = -\lambda \cdot \left. \frac{\partial \text{loss}(\theta)}{\partial \theta} \right|_{\theta = \theta_0} \]

simple solution: take smaller steps
Gradient Descent

Main problem with gradient-based optimization: instability

simple solution: take smaller steps

Pro: more stable
Gradient Descent

Main problem with gradient-based optimization: instability

\[ \theta_0 \mapsto \theta_0 + \Delta \theta \]

\[ \Delta \theta = -\lambda \cdot \left. \frac{\partial \text{[loss(\theta)]}}{\partial \theta} \right|_{\theta=\theta_0} \]

“learning rate” < 1

simple solution: take smaller steps

Pro: more stable
Con: if you pick LR too small, converges slowly
Gradient Descent

[Ipynb: With Learning Rate]
Gradient Descent

There are multiple improvements on Gradient Descent. One is called the “Momentum” method.
There are multiple improvements on Gradient Descent. One is called the “Momentum” method.

Original gradient descent:

\[ \Delta \theta = -\lambda \cdot \nabla_\theta [\text{loss}(\theta)]|_{\theta=\theta_0} \]
Gradient Descent

There are multiple improvements on Gradient Descent. One is called the “Momentum” method

Original gradient descent:

$$\Delta \theta = -\lambda \cdot \nabla_{\theta}[\text{loss}(\theta)]|_{\theta=\theta_0}$$

Momentum method:

$$\Delta \theta = -\lambda \cdot [\nabla_{\theta}[\text{loss}(\theta)]|_{\theta=\theta_0} + \mu \cdot \text{grad}_{\text{acc}}]$$
Gradient Descent

There are multiple improvements on Gradient Descent. One is called the “Momentum” method.

Original gradient descent:

$$\Delta \theta = -\lambda \cdot \nabla_\theta [\text{loss}(\theta)]|_{\theta=\theta_0}$$

Momentum method:

$$\Delta \theta = -\lambda \cdot \left[ \nabla_\theta [\text{loss}(\theta)]|_{\theta=\theta_0} + \mu \cdot \text{grad\_accum} \right]$$
There are multiple improvements on Gradient Descent. One is called the “Momentum” method.

Original gradient descent:

\[ \Delta \theta = -\lambda \cdot \nabla_\theta [\text{loss}(\theta)]|_{\theta=\theta_0} \]

Momentum method:

\[ \Delta \theta = -\lambda \cdot [\nabla_\theta [\text{loss}(\theta)]|_{\theta=\theta_0} + \mu \cdot \text{grad\_accum}] \]

update \text{grad\_accum} to this for next time step

momentum parameter
Gradient Descent

There are multiple improvements on Gradient Descent. One is called the “Momentum” method

Original gradient descent:

\[
\Delta \theta = -\lambda \cdot \nabla_\theta [\text{loss}(\theta)]|_{\theta=\theta_0}
\]

Momentum method:

\[
\Delta \theta = -\lambda \cdot \left[ \nabla_\theta [\text{loss}(\theta)]|_{\theta=\theta_0} + \mu \cdot \text{grad\_accum} \right]
\]

update grad\_accum to this for next time step

Pros: faster, stabler convergence.  
Cons: nothing (relative to not doing it).
Amazing explanation of why momentum is so good:

https://distill.pub/2017/momentum/

Ours: \( \lambda \)  
\( \mu \)

Theirs: \( \alpha \)  
\( \beta \)

“Here’s a popular story about momentum: gradient descent is a man walking down a hill. He follows the steepest path downwards; his progress is slow, but steady. Momentum is a heavy ball rolling down the same hill. The added inertia acts both as a smoother and an accelerator, dampening oscillations and causing us to barrel through narrow valleys, small humps and local minima. ….”
Gradient Descent

[Intnb: Tensorflow’s Built-in Optimizers]
Gradient Descent

Training

tf.train provides a set of classes and functions that help train models.

Optimizers

The Optimizer base class provides methods to compute gradients for a loss and apply gradients to variables. A collection of subclasses implement classic optimization algorithms such as GradientDescent and Adagrad.

You never instantiate the Optimizer class itself, but instead instantiate one of the subclasses.

- tf.train.Optimizer
- tf.train.GradientDescentOptimizer
- tf.train.AdadeltaOptimizer
- tf.train.AdagradOptimizer
- tf.train.AdagradDAOptimizer
- tf.train.MomentumOptimizer
- tf.train.AdamOptimizer
- tf.train.FtrlOptimizer
- tf.train.ProximalGradientDescentOptimizer
- tf.train.ProximalAdagradOptimizer
- tf.train.RMSPropOptimizer
Second-Order Methods

Taylor series for function $f$:
Taylor series for function $f$:

$$f(x) = f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)\Delta x^2 + \ldots$$
Second-Order Methods

Taylor series for function $f$:

$$f(x_0, \Delta x) = f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)\Delta x^2 + \ldots$$
Second-Order Methods

Taylor series for function $f$:

\[ f(x_0, \Delta x) = f(x_0) + f'(x_0)\Delta x + \frac{1}{2} f''(x_0)\Delta x^2 + \ldots \]

Differentiate by $\Delta x$:
Second-Order Methods

Taylor series for function $f$:

$$f(x_0, \Delta x) = f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)\Delta x^2 + \ldots$$

Differentiate by $\Delta x$:

$$\frac{\partial f(x_0, \Delta x)}{\partial \Delta x} = f'(x_0) + f''(x_0)\Delta x + \ldots$$
Second-Order Methods

Taylor series for function $f$:

$$f(x_0, \Delta x) = f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)\Delta x^2 + \ldots$$

Differentiate by $\Delta x$:

$$\frac{\partial f(x_0, \Delta x)}{\partial \Delta x} = f'(x_0) + f''(x_0)\Delta x + \ldots$$

Set to 0 and ignore HOT:
Second-Order Methods

Taylor series for function $f$:

$$f(x_0, \Delta x) = f(x_0) + f'(x_0) \Delta x + \frac{1}{2} f''(x_0) \Delta x^2 + \ldots$$

Differentiate by $\Delta x$:

$$\frac{\partial f(x_0, \Delta x)}{\partial \Delta x} = f'(x_0) + f''(x_0) \Delta x + \ldots$$

Set to 0 and ignore Higher Order Terms (HOT):

$$0 = f'(x_0) + f''(x_0) \Delta x$$
Taylor series for function $f$:

$$f(x_0, \Delta x) = f(x_0) + f'(x_0) \Delta x + \frac{1}{2} f''(x_0) \Delta x^2 + \ldots$$

Differentiate by $\Delta x$:

$$\frac{\partial f(x_0, \Delta x)}{\partial \Delta x} = f'(x_0) + f''(x_0) \Delta x + \ldots$$

Set to 0 and ignore Higher Order Terms (HOT):

$$0 = f'(x_0) + f''(x_0) \Delta x$$

Solve for $\Delta x$: 
Second-Order Methods

Taylor series for function $f$:

$$f(x_0, \Delta x) = f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)\Delta x^2 + \ldots$$

Differentiate by $\Delta x$:

$$\frac{\partial f(x_0, \Delta x)}{\partial \Delta x} = f'(x_0) + f''(x_0)\Delta x + \ldots$$

Set to 0 and ignore Higher Order Terms (HOT):

$$0 = f'(x_0) + f''(x_0)\Delta x$$

Solve for $\Delta x$:

$$\Delta x = -\frac{f'(x_0)}{f''(x_0)}$$
Second-Order Methods

Taylor series for function $f$:

$$f(x_0, \Delta x) = f(x_0) + f'(x_0) \Delta x + \frac{1}{2} f''(x_0) \Delta x^2 + \ldots$$

Differentiate by $\Delta x$:

$$\frac{\partial f(x_0, \Delta x)}{\partial \Delta x} = f'(x_0) + f''(x_0) \Delta x + \ldots$$

Set to 0 and ignore Higher Order Terms (HOT):

$$0 = f'(x_0) + f''(x_0) \Delta x$$

Solve for $\Delta x$:

$$\Delta x = -f'(x_0) \cdot \frac{1}{f''(x_0)}$$

really correct value for learning rate $\lambda$
Second-Order Methods

Taylor series for function $f$:

$$f(x_0, \Delta x) = f(x_0) + f'(x_0) \Delta x + \frac{1}{2} f''(x_0) \Delta x^2 + \ldots$$

Differentiate by $\Delta x$:

$$\frac{\partial f(x_0, \Delta x)}{\partial \Delta x} = f'(x_0) + f''(x_0) \Delta x + \ldots$$

Set to 0 and ignore Higher Order Terms (HOT):

$$0 = f'(x_0) + f''(x_0) \Delta x$$

Solve for $\Delta x$:

$$\Delta x = -f'(x_0) \cdot \frac{1}{f''(x_0)}$$

really correct value for learning rate $\lambda$
Taylor series for function $f(x_1, \ldots, x_n)$ of $n$ inputs:

$$f(\vec{x}_0, \Delta \vec{x}) = f(\vec{x}_0) + \nabla f(\vec{x}_0) \cdot \Delta \vec{x} + \frac{1}{2} (H[F] \cdot \vec{x}_0) \cdot \Delta \vec{x}^2 + \ldots$$

the gradient $\nabla f = \left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$

the "hessian" matrix $H[f] = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_1 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \frac{\partial^2 f}{\partial x_2 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \frac{\partial^2 f}{\partial x_n \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix}$

shape = $(n, n)$
Second-Order Methods

Taylor series for function $f(x_1, \ldots, x_n)$ of $n$ inputs:

$$f(x_0, \Delta x) = f(x_0) + \nabla f(x_0) \cdot \Delta x + \frac{1}{2} (H[F] \cdot x_0) \cdot \Delta x^2 + \ldots$$

the gradient

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

the “hessian”

$$H[f] = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_1 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \frac{\partial^2 f}{\partial x_2 \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \frac{\partial^2 f}{\partial x_n \partial x_3} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n}
\end{bmatrix}$$

shape = $(n, n)$

matrix-on-vector multiplication
Second-Order Methods

Taylor series for function $f(x_1, \ldots, x_n)$ of $n$ inputs:

$$f(x_0, \Delta x) = f(x_0) + \nabla f(x_0) \cdot \Delta x + \frac{1}{2} (H[F] \cdot \vec{x}_0) \cdot \Delta x^2 + \ldots$$

**single variable:**

$$\Delta x = -\frac{f'(x_0)}{f''(x_0)} \quad \Rightarrow \quad \Delta \vec{x} = - (H[f])^{-1} \cdot \nabla f(\vec{x}_0)$$

**multi-variable:**
Second-Order Methods

Taylor series for function $f(x_1, \ldots, x_n)$ of $n$ inputs:

$$f(x_0, \Delta x) = f(x_0) + \nabla f(x_0) \cdot \Delta x + \frac{1}{2} (H[F] \cdot \vec{x}_0) \cdot \Delta x^2 + \ldots$$

single variable:

$$\Delta x = -\frac{f'(x_0)}{f''(x_0)}$$

⇒

multi-variable:

$$\Delta \vec{x} = -(H[f])^{-1} \cdot \nabla f(\vec{x}_0)$$

$n$ vector $\rightarrow$ $n \times n$ matrix $\rightarrow$ $n$ vector

matrix inverse $\rightarrow$ matrix-on-vector mult
Gradient Descent

[Ipynb: Newton’s Method]
External Optimizers

Don’t have to use optimizers just from Tensorflow, even if using Tensorflow to get derivatives

Really powerful general optimizer for general Python use:
def minimize(loss_func, x0, optimizer_type)

starts at x0, uses whatever specified optimizer to minimize loss_func

doesn't use derivative info
def minimize(loss_func, x0, optimizer_type)

starts at x0, uses whatever specified optimizer to minimize loss_func
doesn't use derivative info

def minimize(loss_func, x0, optimizer_type, jac=True)

loss_func must now return both loss and loss gradient:

    loss_val, grad = loss_func(x)

does use derivative info
External Optimizers

[Ipython: Using External Optimizers]
External Optimizers

Target for learning (with 5-th order polynomial model)
\[ y = \text{np.cosh}(2 \ast x) + \text{np.sin}(x + 1) \]

Not using derivative info:

Using derivative:
Integration with existing tools

[lpyrb: Integrating with cross-validation tools]
Biological learning

Hebb’s Rule:

$$\Delta \theta_{ij} \propto x_i \cdot y_j$$
Biological learning

Hebb’s Rule:

\[ \Delta \theta_{ij} \propto x_i \cdot y_j \]

presynaptic activity

postsynaptic activity

Donald Hebb
Biological learning

Hebb’s Rule:

\[ \Delta \theta_{ij} \propto x_i \cdot y_j \]

- presynaptic activity
- postsynaptic activity

- repeated stimulation
- more dendritic receptors
- more neurotransmitters
- stronger link
Biological learning

Hebb’s Rule:

$$\Delta \theta_{ij} \propto x_i \cdot y_j$$

presynaptic activity  
postsynaptic activity
Biological learning

Hebb's Rule:

$$\Delta \theta_{ij} \propto x_i \cdot y_j$$

presynaptic activity  
postsynaptic activity

Oja's Rule: improves stability of learning convergence

$$\Delta \theta_{ij} \propto y_j \cdot (x_i - y_j \cdot \theta_{ij})$$
Biological learning

Hebb’s Rule:

\[ \Delta \theta_{ij} \propto x_i \cdot y_j \]

- presynaptic activity
- postsynaptic activity

Oja’s Rule: *improves stability of learning convergence*

\[ \Delta \theta_{ij} \propto y_j \cdot (x_i - y_j \cdot \theta_{ij}) \]

Compared to:

\[ \Delta \theta \propto - \frac{\partial \text{[loss}(\theta)]}{\partial \theta} \bigg|_{\theta=\theta_0} \]

... hebbian learning is:

(a) local
(b) not derivative-based
Biological learning

Hebb's Rule:  
\[ \Delta \theta_{ij} \propto x_i \cdot y_j \]

- presynaptic activity
- postsynaptic activity

Oja's Rule: *improves stability of learning convergence*

\[ \Delta \theta_{ij} \propto y_j \cdot (x_i - y_j \cdot \theta_{ij}) \]

Compared to:

\[ \Delta \theta \propto -\frac{\partial [\text{loss}(\theta)]}{\partial \theta} \bigg|_{\theta=\theta_0} \]

- (a) local
- (b) not derivative-based

Donald Hebb

https://www.slideshare.net/mentelibre/hebbian-learning
Direct Feedback Alignment Provides Learning in Deep Neural Networks

Arild Nøkland

(Submitted on 6 Sep 2016 (v1), last revised 21 Dec 2016 (this version, v5))

Random synaptic feedback weights support error backpropagation for deep learning

Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed & Colin J. Akerman
Direct Feedback Alignment Provides Learning in Deep Neural Networks

Arild Nøkland

(Submitted on 6 Sep 2016 (v1), last revised 21 Dec 2016 (this version, v5))

Random synaptic feedback weights support error backpropagation for deep learning

Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed & Colin J. Akerman

Diagram showing different feedback alignment methods:

- **BP (Backpropagation)**: No feedback connections.
- **FA (Feedback Alignment)**: Feedback connections are added to the network.
- **DFA (Differentiable Feedback Alignment)**: Feedback connections are added and are differentiable.
- **IFA (Improved Feedback Alignment)**: Feedback connections are added and are improved for learning.

The diagrams illustrate the flow of information through the layers of a deep neural network, emphasizing the role of feedback in aligning the error backpropagation process.