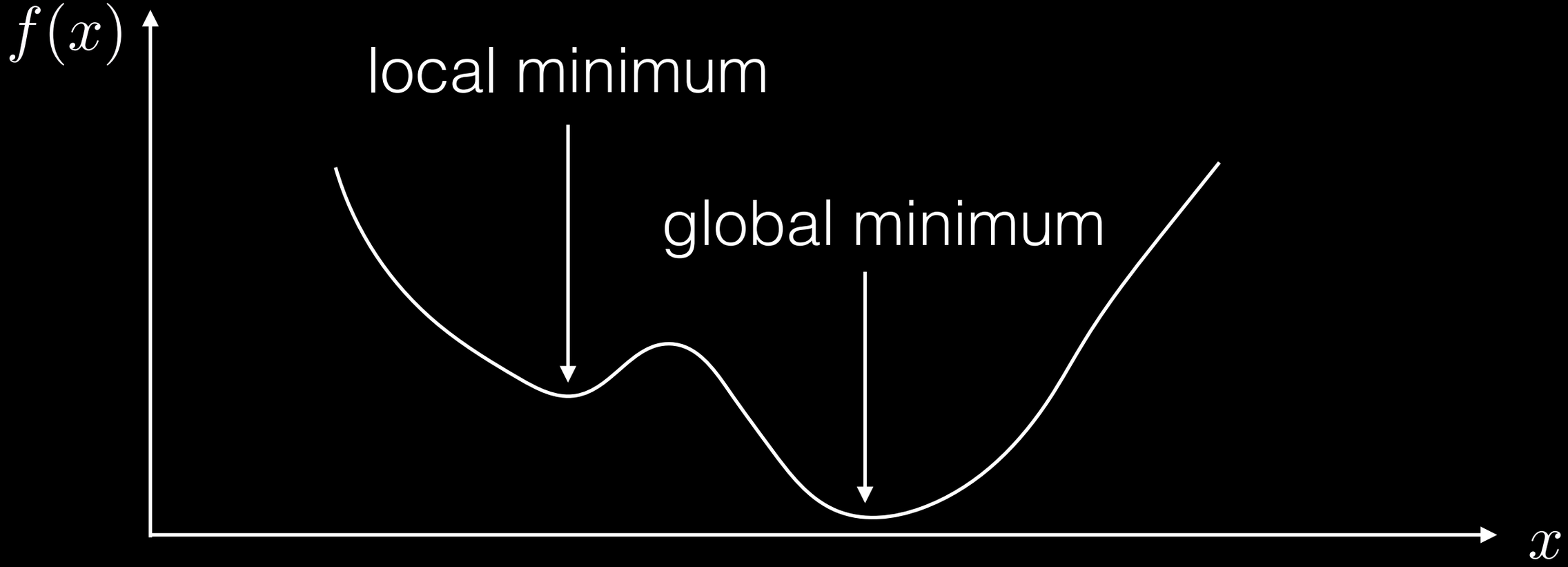
Deep Learning for Computer Vision

Lecture 8: Optimization

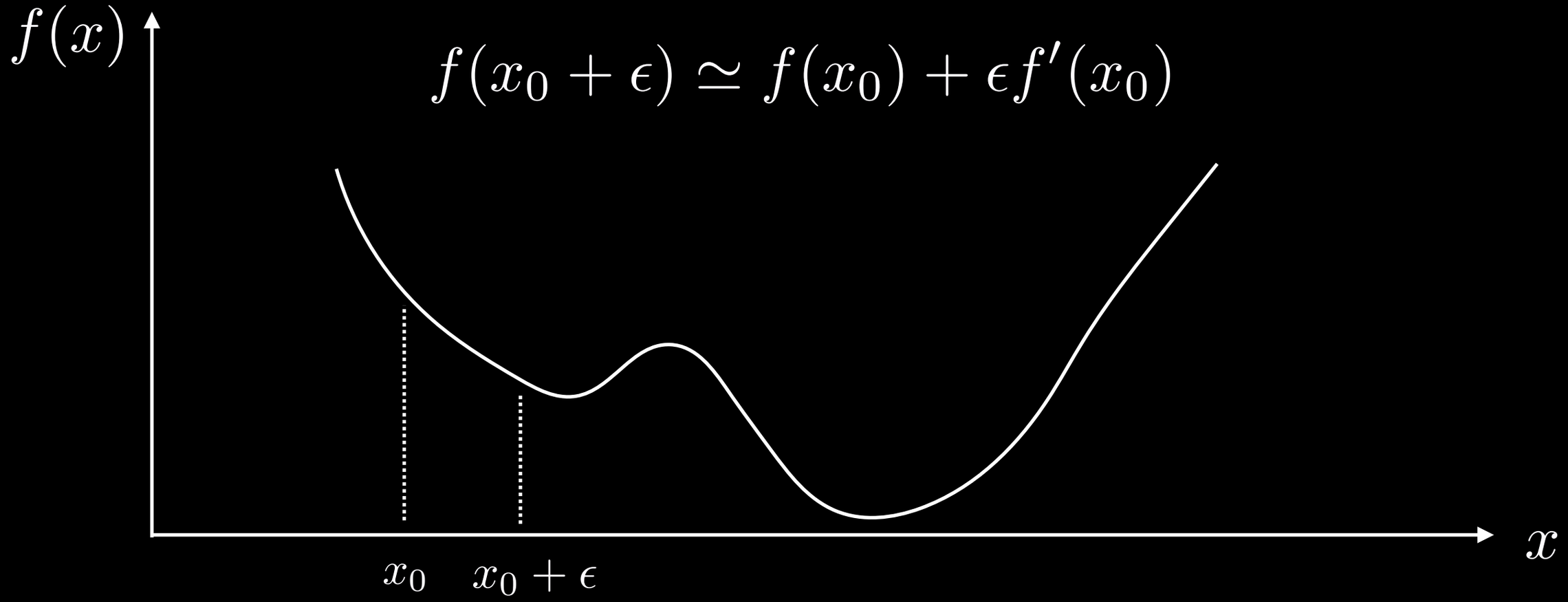
Peter Belhumeur

Computer Science
Columbia University

Gradient-Based Optimization



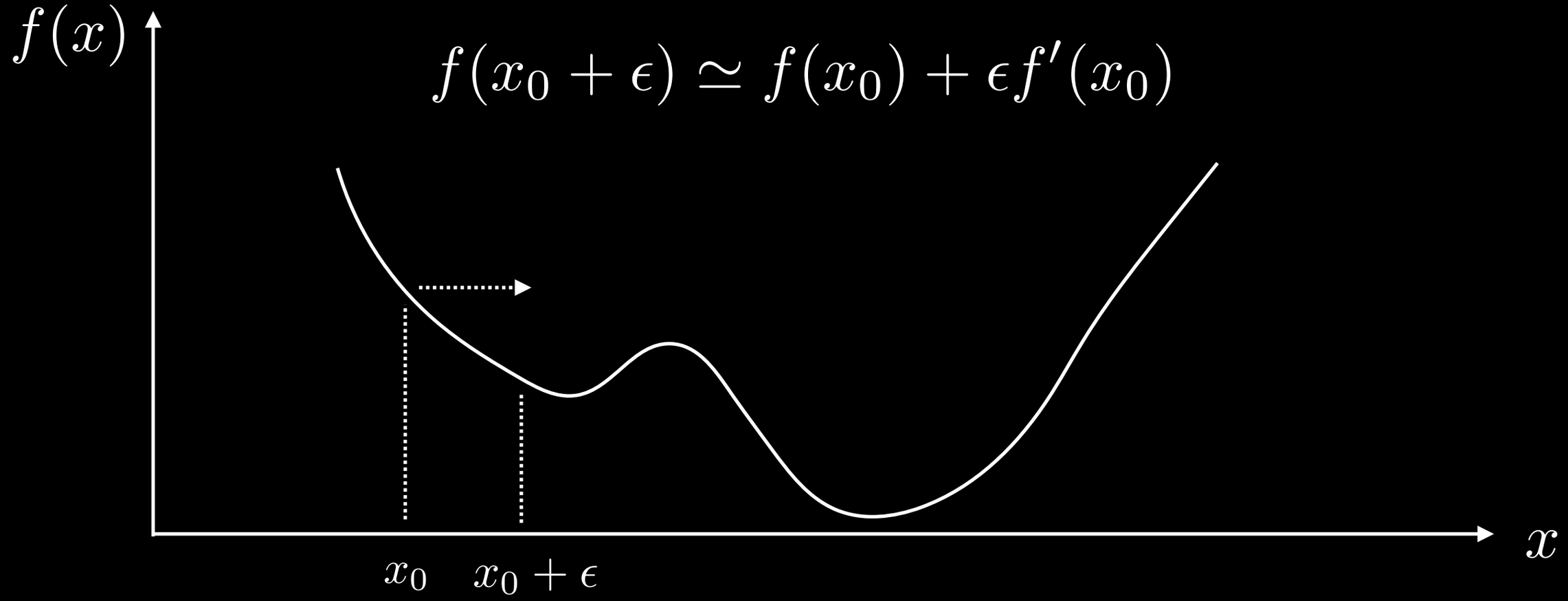
Gradient-Based Optimization



$$f' = \frac{df}{dx}$$

Gradient Descent

$$f(x_0 + \epsilon) \simeq f(x_0) + \epsilon f'(x_0)$$

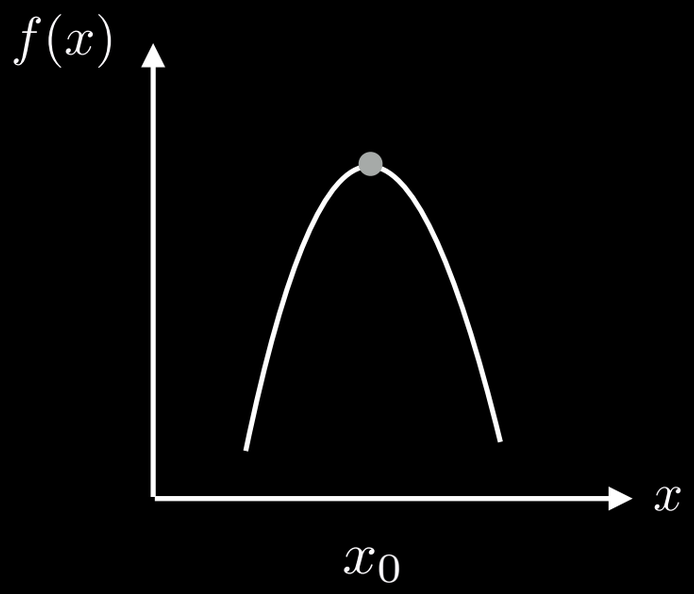


Note that f' is negative, so going in positive direction decreases the function.

Critical Points

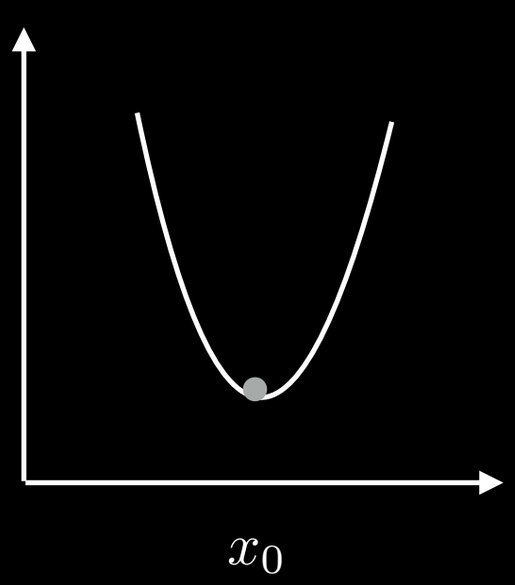
$$f'(x_0) = 0$$

Maximum



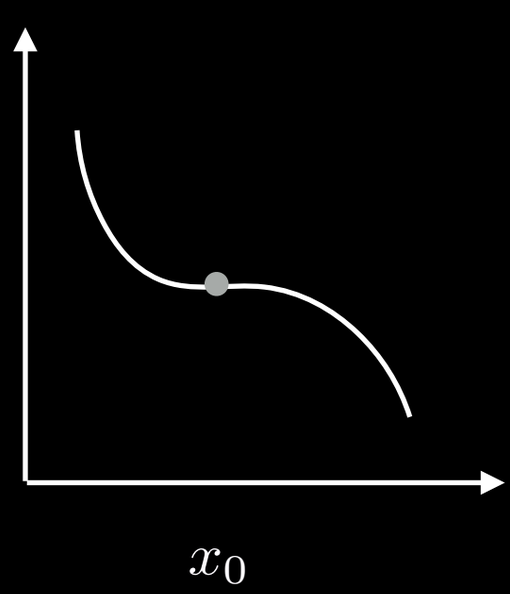
$$f''(x_0) < 0$$

Minimum



$$f''(x_0) > 0$$

Saddle Point



$$f''(x_0) = 0$$

What if our input is a vector?

- Let $f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$
- The ***directional derivative*** is the slope of the function in direction $\mathbf{u}$
- We can find this as $\frac{\partial}{\partial \eta} f(\mathbf{x} + \eta \mathbf{u}) \quad \text{at} \quad \eta = 0$
- …or after the chain rule yields $\nabla_{\mathbf{x}} f(\mathbf{x})^T \, \mathbf{u}$

PLEASE DON’T FORGET

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \cdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

Gradient Descent

- So if we move in direction $\mathbf{u}$ the slope is $\nabla_{\mathbf{x}} f(\mathbf{x})^T \mathbf{u}$
- So in what direction is the slope most negative?
- Clearly in the **OPPOSITE** direction of the gradient!
- And if we traverse the fcn this way then we are doing ***steepest descent*** or ***gradient descent***.

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla_{\mathbf{x}} f(\mathbf{x_t})$$

The Hessian Matrix

- Now we just saw that the gradient is the vector of partial derivatives wrt each of the input variables
- What if we took second derivatives?
- To do this, we can compute the Jacobian of the gradient.
- This beast is called the Hessian!

The Hessian Matrix

$$\mathbf{H}(f)(\mathbf{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x})$$

The Hessian Matrix

At a critical point $(\nabla_{\mathbf{x}} f = \mathbf{0})$:

- if all of the eigenvalues of H are positive, then point is a local minimum.
- if all of the eigenvalues of H are negative, then the point is a local maximum.
- if one or more are positive and one or more are negative, then the point is a saddle point.

Taylor Series Expansion

- Taylor series expansion gives us this approximation:

$$f(\mathbf{x}_{t+1}) \approx f(\mathbf{x}_t) + (\mathbf{x}_{t+1} - \mathbf{x}_t)^T \mathbf{g} + \frac{1}{2}(\mathbf{x}_{t+1} - \mathbf{x}_t)^T \mathbf{H} (\mathbf{x}_{t+1} - \mathbf{x}_t)$$

- So if we update as: $\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\, \mathbf{g}$

- Then we expect this change in the function:

$$f(\mathbf{x}_{t+1}) \approx f(\mathbf{x}_t) - \eta\, \mathbf{g}^T \mathbf{g} + \frac{1}{2} \eta^2\, \mathbf{g}^T \mathbf{H} \mathbf{g}$$

And that's not all…

- Note that the second order term involving the Hessian tells us what to expect if we move in the direction of the opposite gradient.
- If the second order term is positive then the decrease in loss is diminished, and if negative it is accelerated:

$$f(\mathbf{x}_{t+1}) \approx f(\mathbf{x}_t) - \eta\, \mathbf{g}^T \mathbf{g} + \frac{1}{2}\eta^2\, \mathbf{g}^T \mathbf{H} \mathbf{g}$$

Optimization for Deep Nets

- Deep learning optimization is a type global optimization where the optimization is usually expressed as a loss summed over all the training samples.
- Our goal is not so much find the parameters (or weights) that minimize the loss but rather to find parameters that produce a network with the desired behavior.
- Note that there are LOTS of solutions to which our optimization could converge to—with very different values for the weights—but each producing a model with very similar behavior on our sample data.
- For example, consider all the permutations of the weights in a hidden layer that produce the same outputs.

Optimization for Deep Nets

- Although there is a seemingly endless literature on global optimization, here we consider only gradient descent-based methods.
- Our optimizations for deep learning are typically done in very high dimensional spaces, were the parameters we are optimizing can run into the millions.
- And for these optimizations, when starting the training from scratch (i.e., some random initialization of the weights) we will needs LOTS of labeled training data.
- The process of learning our model from this labeled data is referred to as ***supervised learning***. Although, supervised learning is more general than the deep learning algorithms we will consider.

Deterministic vs. Stochastic Methods

- If we performed our gradient descent optimization using all the training samples to compute each step in our parameter updates, then our optimization would be ***deterministic***.
- Confusingly, ***deterministic*** gradient descent algorithms are sometimes referred to as ***batch*** algorithms
- In contrast, when we use a subset of randomly selected training samples to compute each update, we call this ***stochastic gradient descent*** and refer to the subset of samples as a ***mini-batch***.
- And even more confusingly, we often call this mini-batch the “batch” and refer to its size as the “batch size.”

Deterministic vs. Stochastic Methods

- In general, we will have too many training samples to use deterministic methods, as it will be too computationally costly to process all samples with each update.

- Also, processing a random mini-batch serves as a type of regularization and helps prevent overfitting.

- So we will, restrict ourselves to stochastic gradient descent (SGD) from here on.

Stochastic Gradient Descent

The SGD algorithm could not be any simpler:

1. Choose a learning rate schedule.
2. Choose stopping criterion.
3. Choose batch size.
4. Randomly select a mini-batch.
5. Propagate it forward through the network and then backward through the network computing the gradient wrt the weights using back propagation.
6. Update the weights by moving in the direction opposite the gradient where the step size is given by the learning rate.
7. Repeat 4, 5, and 6 until the stopping criterion is satisfied.

Stochastic Gradient Descent

The SGD algorithm could not be any simpler:

1. Choose a learning rate schedule η_t.
2. Choose stopping criterion.
3. Choose batch size m.
4. Randomly select mini-batch $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)}\}$
5. Forward and backpropagation
6. Update $\theta_{t+1} = \theta_t - \eta_t \, g \qquad \mathbf{g} = \frac{1}{m} \sum_i^m \nabla_\theta L(\mathbf{x}^{(i)}, y^i)$
7. Repeat 4, 5, 6 until the stopping criterion is satisfied.

SGD with Momentum

Update rule with momentum:

1. Compute the gradient: $\mathbf{g} = \frac{1}{m} \sum_i \nabla_\theta L(\mathbf{x}^{(i)}, y^i)$
2. Compute the velocity: $v_t = \alpha_t \, v_{t-1} - \eta_t \, g$
3. Update: $\theta_{t+1} = \theta_t + v_t$

Note: α_t starts small and increases with time (typically)
η_t starts large and decreases with time

Learning Rate

- Choosing a learning rate has so far eluded science and remains a bit of an art.
- A typical learning rate schedule might look like:

for $t < \tau, \;\; \eta_t = \left(1 - \frac{t}{\tau}\right) \eta_0 + \frac{t}{\tau} \eta_\tau$

for $t \geq \tau, \;\; \eta_t = \eta_\tau$

with τ equivalent to 100 - 1000 passes through the data

and $\eta_\tau = 0.01 \, \eta_0$

Learning Rate

- But this is just one choice for a learning schedule
- One might use an exponential decay
- Or use an adaptive learning rate…

AdaGrad [Duchi et al. 2011]

- Let the learning rate for a model parameter be inversely proportional to the square root of the sum of the square of all past values for that model parameter's partial derivative.
- So parameters with a history of large partial derivatives get smaller step sizes, and vice versa.
- Works well sometimes, but large initial gradients can slow down the learning rates too much.

AdaGrad

AdaGrad update rule:

1. Compute the gradient: $\mathbf{g} = \frac{1}{m} \sum_{i}^{m} \nabla_\theta L(\mathbf{x}^{(i)}, y^i)$
2. Accumulate: $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g} \odot \mathbf{g}$
3. Update parameters: $\theta_{t+1} = \theta_t - \frac{\eta}{\delta + \sqrt{\mathbf{s}_t}} \odot \mathbf{g}$

RMSProp [Hinton 2012]

- Similar to AdaGrad but introduces an exponential decay on the accumulation.

- Adds another hyperparameter specifying the decay rate.

- Frequently used learning rate in practice.

RMSProp

RMSProp update rule:

1. Compute the gradient: $\mathbf{g} = \frac{1}{m} \sum_{i}^{m} \nabla_\theta L(\mathbf{x}^{(i)}, y^i)$
2. Accumulate: $\mathbf{s}_t = \beta\, \mathbf{s}_{t-1} + (1-\beta)\, \mathbf{g} \odot \mathbf{g} \qquad \beta < 1$
3. Update parameters: $\theta_{t+1} = \theta_t - \frac{\eta}{\delta + \sqrt{\mathbf{s}_t}} \odot \mathbf{g}$

Possible defaults: $\beta = 0.9 \qquad \eta = 0.001$

Adam [Kingma and Ba 2014]

- Name comes from "**Ada**ptive **m**oments"
- Typically not too sensitive to choice of hyperparameters.
- Frequently used learning rate in practice.

Adam

Adam update rule:

1. Compute the gradient: $\mathbf{g}_t = \frac{1}{m} \sum_i^m \nabla_\theta L(\mathbf{x}^{(i)}, y^i)$
2. Update first moment: $\mathbf{r}_t = \beta_1 \, \mathbf{r}_{t-1} + (1 - \beta_1) \mathbf{g}_t$
3. Correct bias: $\hat{\mathbf{r}}_t = \frac{\mathbf{r}_t}{1 - \beta_1^t}$
4. Update second moment: $\mathbf{s}_t = \beta_2 \, \mathbf{s}_{t-1} + (1 - \beta_2) \, \mathbf{g}_t \odot \mathbf{g}_t$
5. Correct bias: $\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}$
6. Update parameters: $\theta_{t+1} = \theta_t - \frac{\eta \, \hat{\mathbf{r}}_t}{\delta + \sqrt{\hat{\mathbf{s}}_t}}$

Possible defaults: $\beta_1 = 0.9 \quad \beta_2 = 0.999 \quad \eta = 0.001 \quad \delta = 10^{-8}$

Batch Normalization

[Ioffe and Szegedy 2015]

- Training deep nets is often tricky. Updates in weights in one layer can get compounded as stuff propagates through network.
- A recent major advance in training these networks was to normalize each batch at each unit of each layer so that it has mean = 0 and variance = 1.
- This ***batch normalization*** is usually done right before a layer's nonlinearity.
- Scaling and bias offsets can be added back in after the normalization as explicitly learned parameters.
- Batch normalization makes training more stable and is now widely adopted.

Batch Normalization

- Let's say we have the input to a layer $X_{[d_{in} \times m]}$ where d_{in} is the input dimension and m is the mini-batch size.
- Let the mini-batch pass through the linear part of the layer $W^T X = X'_{[d_{out} \times m]}$
- Note we don't have any bias here as this will be stripped by the normalization.

Batch Normalization

- At this point in the network—right before the ReLu—we are going to insert batch normalization.
- To do this, we are going to process every row of X' so that is has mean = 0 and variance = 1.
- Note that each unit—row of X'—is normalized separately to produce a new matrix X''
- Finally, we rescale and shift each row by broadcasting vectors γ and β to produce $\gamma X'' + \beta$

Batch Normalization

1. $X' = W^T X$ Put input through linearity
2. $\mu = \frac{1}{m} \sum_{i=1}^{m} X'_{:,i}$ Find the mean of each unit
3. $\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (X'_{:,i} - \mu)^2$ Find the variance of each unit
4. $X''_{:,i} = \frac{X'_{:,i} - \mu}{\sqrt{\sigma^2 + \epsilon}}$ Normalize each unit
5. $X'''_{:,i} = \gamma \odot X''_{:,i} + \beta$ Rescale and shift each unit

Batch Normalization

- Batch normalization just becomes another layer that can be added to the network.

- The layer is **subject to both forward and back propagation**!

- The scaling γ and offset β are learned like all other weights.