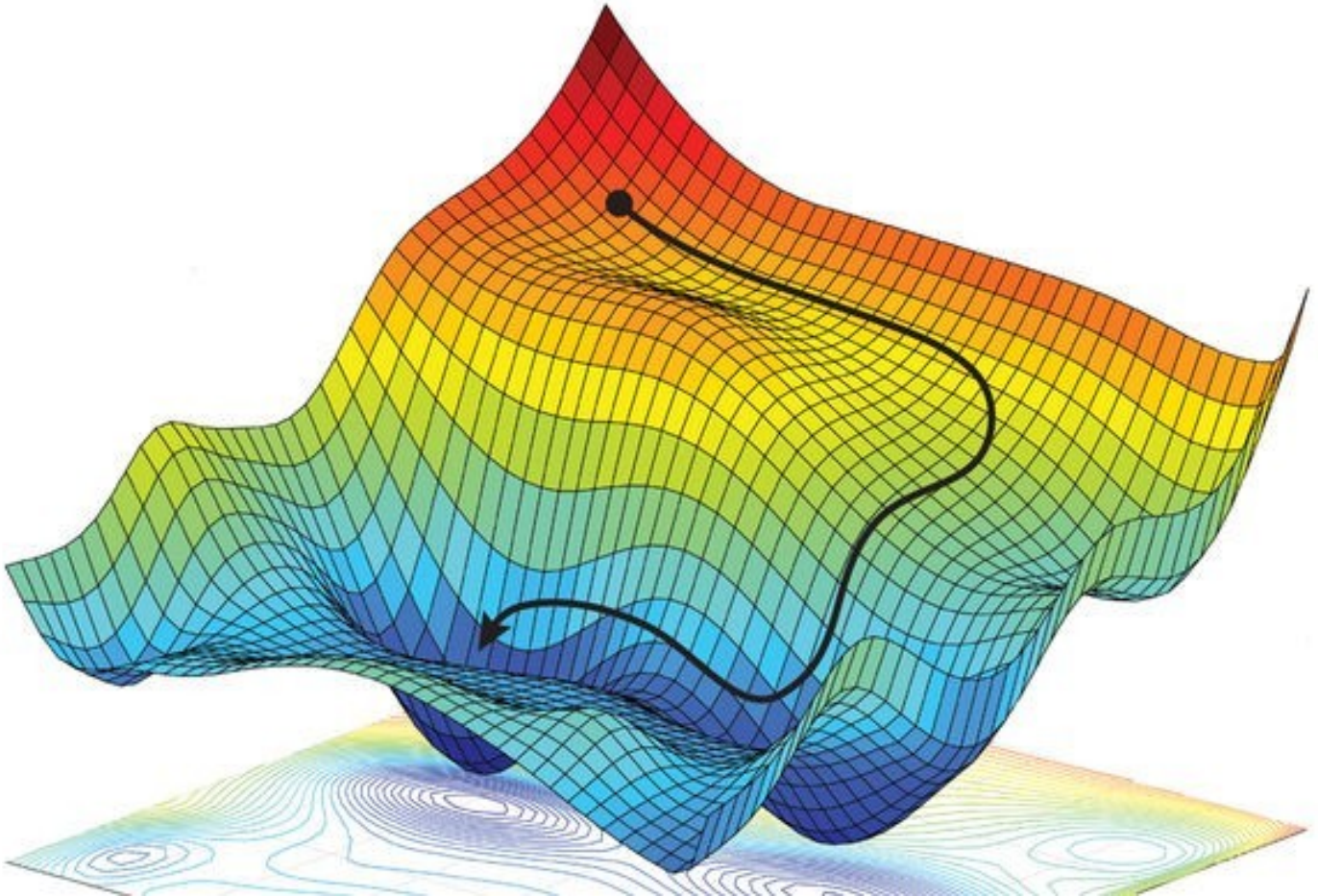
Optimizers



Optimizers

Recall:
Cost function: *How good is your neural network?*

Then:
Optimizer: *Let's make your neural network better*

Optimizers

Methods

- *Gradient descent*
- *Stochastic gradient descent*

Optimizers

Gradient descent

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

where

$J(\theta)$ = cost function

θ = the parameters (i.e. network weights)

α = learning rate

Optimizers

Gradient descent

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

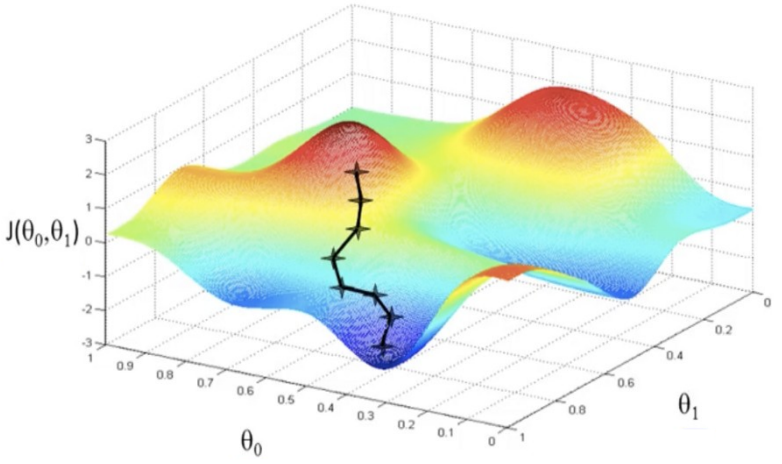
}

where

$J(\theta)$ = cost function

θ = the parameters (i.e. network weights)

α = learning rate



Optimizers

Gradient descent

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

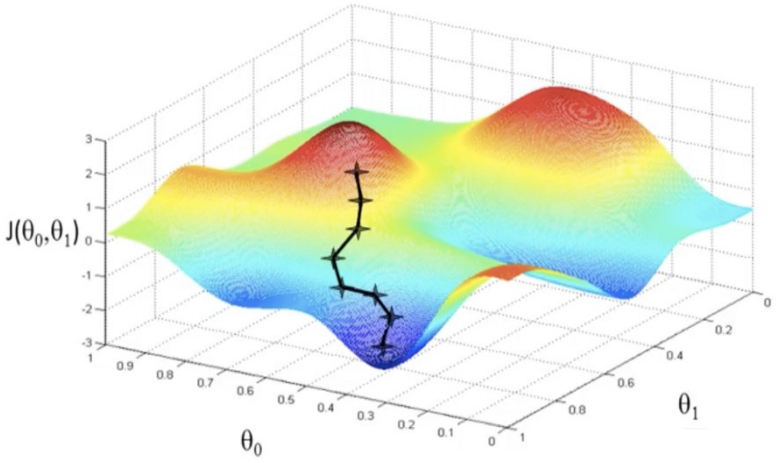
}

where

$J(\theta)$ = cost function

θ = the parameters (i.e. network weights)

α = learning rate



$$\nabla J(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} J(\theta) \\ \frac{\partial}{\partial \theta_1} J(\theta) \end{pmatrix}$$

Optimizers

Gradient descent

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

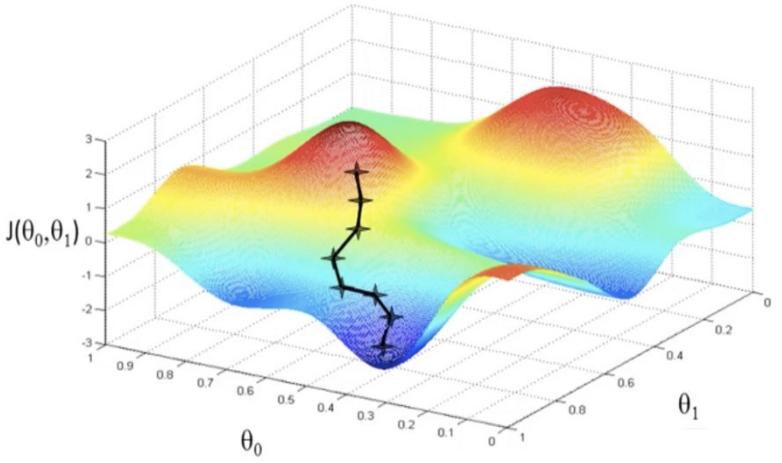
}

where

$J(\theta)$ = cost function

θ = the parameters (i.e. network weights)

α = learning rate



$$\nabla J(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} J(\theta) \\ \frac{\partial}{\partial \theta_1} J(\theta) \end{pmatrix}$$

$$\theta_0 \leftarrow \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta)\Big|_{\theta_0,\theta_1}$$

$$\theta_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta)\Big|_{\theta_0,\theta_1}$$

Optimizers

Gradient descent

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

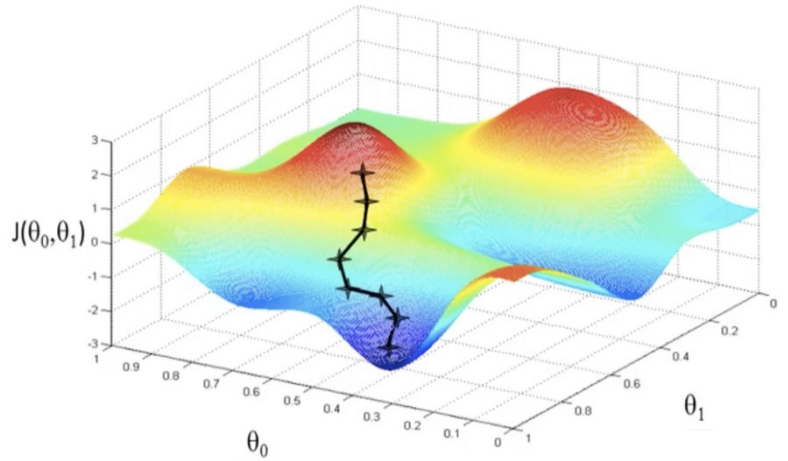
}

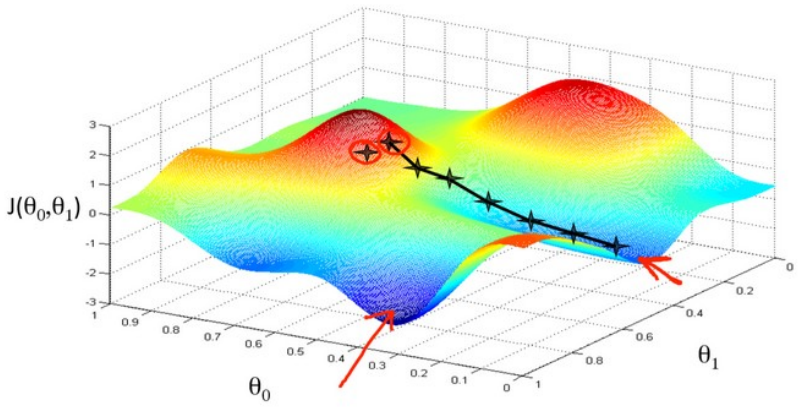
where

$J(\theta)$ = cost function

θ = the parameters (i.e. network weights)

α = learning rate





Optimizers

Stochastic gradient descent

- *Computationally cheaper (important for large datasets)*
- *Avoids issues of finding local minima*

Idea

*Rather than calculate the gradient using the entire dataset (which could be massive), use a **randomly selected subset** of the data (i.e. a mini-batch)*

Hence stochastic

Note

'True' SGD actually performs an update after every single datapoint. In practice using mini-batches performs significantly better

Optimizers

Stochastic gradient descent

*Rather than calculate the gradient using the entire dataset (which could be massive), use a **randomly selected subset** of the data (i.e. a mini-batch)*

Recall

*The **cost function** is the **average of the losses***

$$Cost = \frac{1}{n}\sum_{i=1}^{n} L(x_i, y_i; \theta)$$

$$\nabla_\theta Cost = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L(x_i, y_i; \theta)$$

For stochastic GD, only use a subset of the x_i and y_i

n is the batch size

Optimizers

Gradient descent example

Suppose we want to fit a straight line $\hat{y} = w_1 + w_2 x$ to a training set with input data $(x_1, x_2, \ldots, x_n)$ and true 'labels' $(y_1, y_2, \ldots, y_n)$. The generated label estimates are $(\hat{y_1}, \hat{y_2}, \ldots, \hat{y_n})$.

Use the Mean Squared Error cost function:

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{n} (\hat{y_i} - y_i)^2 = \sum_{i=1}^{n} (w_1 + w_2 x_i - y_i)^2$$

Then the update rule for the parameters w_1 and w_2 is:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial}{\partial w_1} (w_1 + w_2 x_i - y_i)^2 \\ \frac{\partial}{\partial w_2} (w_1 + w_2 x_i - y_i)^2 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \eta \begin{bmatrix} 2(w_1 + w_2 x_i - y_i) \\ 2x_i(w_1 + w_2 x_i - y_i) \end{bmatrix}$$

- To do **vanilla gradient descent**, use all the x_i and y_i for each update
- To do **true gradient descent**, only use a single x_i and y_i for each update
- To do **standard** (i.e. batch) **gradient descent**, use a subset of the x_i and y_i for each update

Optimizers

Adam

- *‘Adaptive moment estimation’*
- *Extends SGD*
- *“Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients”*

Adam: A Method for Stochastic Optimization (Kingma & Ba, 2015)

- *Combines and extends AdaGrad and RMSProp:*
  - ***AdaGrad** (Adaptive Gradient Algorithm) maintains a per-parameter learning rate that improves performance on problems with **sparse gradients** (e.g. natural language and computer vision problems).*
  - ***RMSProp** (Root Mean Square Propagation) also maintains per-parameter learning rates that are adapted based on the **average of recent gradients** (i.e. how quickly it is changing). This means the algorithm does well on **online and non-stationary** (i.e. noisy) problems.*

- *Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, **Adam also makes use of the average of the second moments of the gradients** (the uncentered variance).*

Optimizers

Adam

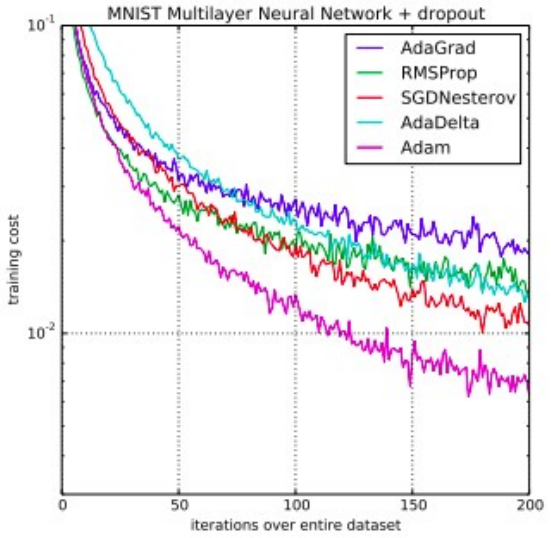
- *Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp,* ***Adam also makes use of the average of the second moments of the gradients*** *(the uncentered variance).*

- *Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.*

Optimizers

Adam

- *“The paper is quite readable and I would encourage you to read it if you are interested in the specific implementation details.”*



- *“In practice Adam is currently recommended as the default algorithm to use”*

Stanford CNNs course delivered by Karpathy

Optimizers

Adam

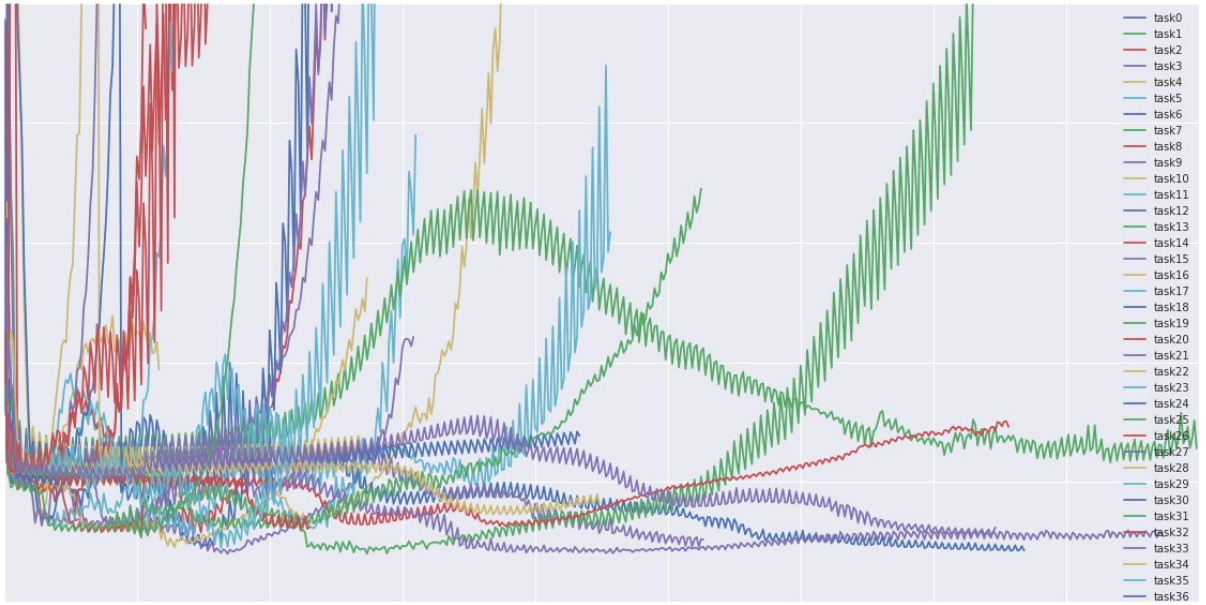
- *Parameters:*
  - ***alpha:*** *[Initial] learning rate or step size. Larger values result in faster initial learning before the rate is updated*
  - ***beta1:*** *The exponential decay rate for the first moment estimates.*
  - ***beta2:*** *The exponential decay rate for the second-moment estimates. Should be set close to 1.0 on problems with sparse gradients (e.g. NLP and computer vision problems).*
  - ***epsilon:*** *A very small number to prevent any division by zero in the implementation.*

- *In Pytorch the default values are as recommended in the original paper:*
  - ***alpha*** *= 0.001*
  - ***beta1*** *= 0.9*
  - ***beta2*** *= 0.999*
  - ***epsilon*** *= 1e-8*

Optimizers

Further reading

- *https://ruder.io/optimizing-gradient-descent/*



Optimizers

Beyond gradient descent...

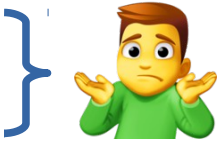
I Googled ‘Are there optimization algorithms that don’t use gradient descent’ and this is what I got:

Optimizers

Beyond gradient descent...

I Googled ‘Are there optimization algorithms that don’t use gradient descent’ and this is what I got:

- *Usually **you want to use the gradient** to optimize neural networks in a supervised setting because it is **significantly faster than derivative-free optimization**.*
- *There are numerous **gradient-based optimization algorithms** that are used to optimize NNs:*
  - *Stochastic Gradient Descent*
  - *Nonlinear Conjugate Gradient*
  - *L-BFGS*
  - *Levenberg-Marquardt Algorithm (LMA)*



Optimizers

Beyond gradient descent...

I Googled 'Are there optimization algorithms that don't use gradient descent' and this is what I got:

- *Usually **you want to use the gradient** to optimize neural networks in a supervised setting because it is **significantly faster than derivative-free optimization**.*
- *There are numerous **gradient-based optimization algorithms** that are used to optimize NNs:*
  - *Stochastic Gradient Descent*
  - *Nonlinear Conjugate Gradient*
  - *L-BFGS*
  - *Levenberg-Marquardt Algorithm (LMA)*

- *The following are good **global optimisation algorithms** that navigate well through huge search spaces, **do not need any information about the gradient** and can be used with **black-box objective functions** and problems that require running simulations:*
  - *Simulated Annealing*
  - *Particle Swarm Optimisation*
  - *Genetic Algorithms*

Danielle?