

CARTE ML Workshop

Neural networks and Backpropagation

Neural Network for classification

Vector function with tunable parameters \mathbf{W}

$$\mathbf{f}(\cdot; \mathbf{W}) : \mathbb{R}^N \rightarrow (0, 1)^K$$

Neural Network for classification

Vector function with tunable parameters \mathbf{W}

$$\mathbf{f}(\cdot; \mathbf{W}) : \mathbb{R}^N \rightarrow (0, 1)^K$$

Sample s in dataset S :

- input: $\mathbf{x}^s \in \mathbb{R}^N$
- expected output: $y^s \in [0, K - 1]$

Neural Network for classification

Vector function with tunable parameters \mathbf{W}

$$\mathbf{f}(\cdot; \mathbf{W}) : \mathbb{R}^N \rightarrow (0, 1)^K$$

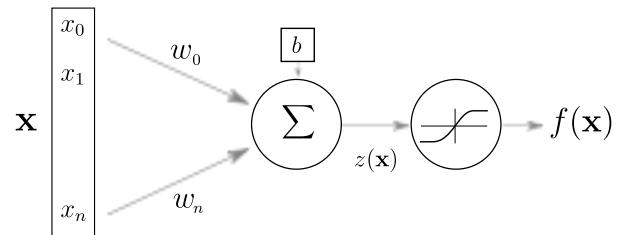
Sample s in dataset S :

- input: $\mathbf{x}^s \in \mathbb{R}^N$
- expected output: $y^s \in [0, K - 1]$

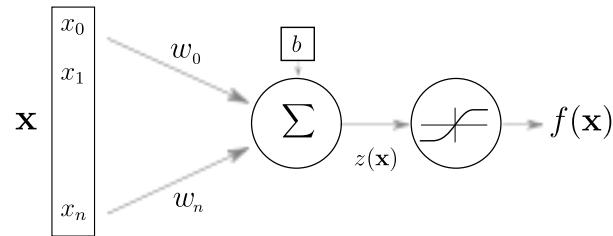
Output is a conditional probability distribution:

$$\mathbf{f}(\mathbf{x}^s; \mathbf{W})_c = P(Y = c | X = \mathbf{x}^s)$$

Artificial Neuron



Artificial Neuron



$$z(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$$

$$f(\mathbf{x}) = g(\mathbf{w}^T \mathbf{x} + b)$$

- $\mathbf{x}, f(\mathbf{x})$ input and output
- $z(\mathbf{x})$ pre-activation
- \mathbf{w}, b weights and bias
- g activation function

Concrete Example

- Say we have two input dimensions x_1 and x_2 and one output dimension $f(x)$
(sometimes, \hat{y} - the predicted value of y - is used instead of $f(x)$)

Concrete Example

- Say we have two input dimensions x_1 and x_2 and one output dimension $f(x)$ (sometimes, \hat{y} - the predicted value of y - is used instead of $f(x)$)
- Our weights and biases could be $\mathbf{W} = [3, -2]$ and $b = 1$

Concrete Example

- Say we have two input dimensions x_1 and x_2 and one output dimension $f(x)$ (sometimes, \hat{y} - the predicted value of y - is used instead of $f(x)$)
- Our weights and biases could be $\mathbf{W} = [3, -2]$ and $b = 1$
- Our non-linearity could be ReLU: $g(z) = \max(0, z)$

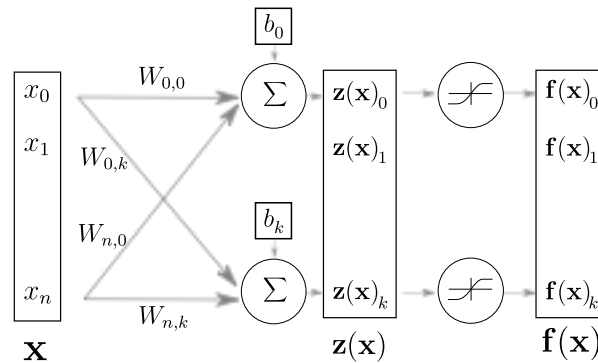
Concrete Example

- Say we have two input dimensions x_1 and x_2 and one output dimension $f(x)$ (sometimes, \hat{y} - the predicted value of y - is used instead of $f(x)$)
- Our weights and biases could be $\mathbf{W} = [3, -2]$ and $b = 1$
- Our non-linearity could be ReLU: $g(z) = \max(0, z)$
- Now $z(x) = 3x_1 - 2x_2 + 1$ and $f(x) = \max(0, 3x_1 - 2x_2 + 1)$

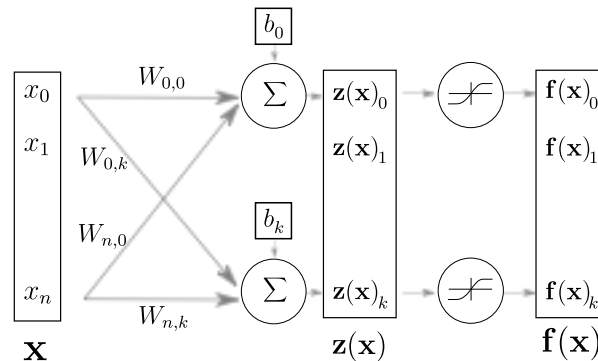
Concrete Example

- Say we have two input dimensions x_1 and x_2 and one output dimension $f(x)$ (sometimes, \hat{y} - the predicted value of y - is used instead of $f(x)$)
- Our weights and biases could be $\mathbf{W} = [3, -2]$ and $b = 1$
- Our non-linearity could be ReLU: $g(z) = \max(0, z)$
- Now $z(x) = 3x_1 - 2x_2 + 1$ and $f(x) = \max(0, 3x_1 - 2x_2 + 1)$
- *Every neuron* in a neural network is a function like this!

Layer of Neurons (Vectorization)



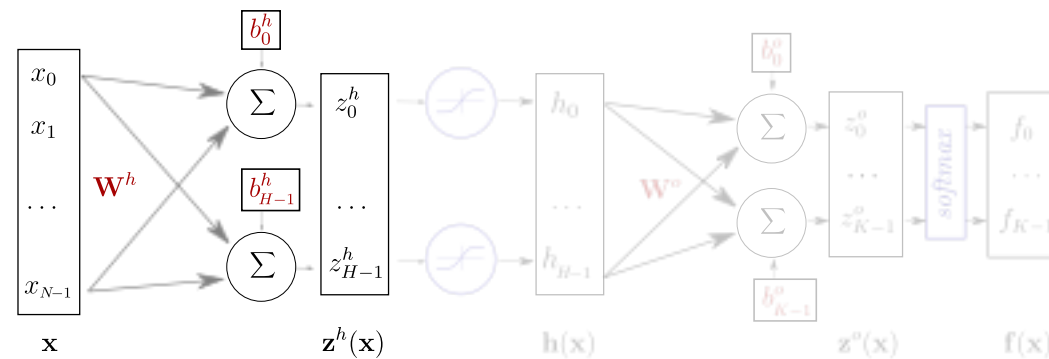
Layer of Neurons (Vectorization)



$$\mathbf{f}(\mathbf{x}) = g(\mathbf{z}(\mathbf{x})) = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

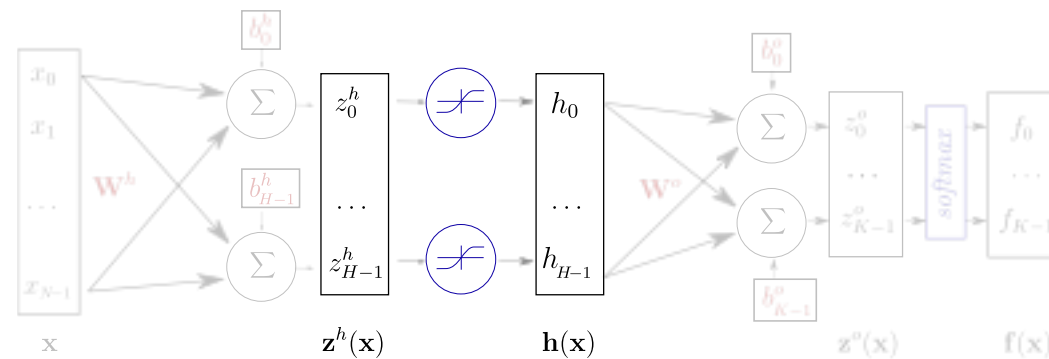
- \mathbf{W}, \mathbf{b} now matrix and vector

One Hidden Layer Network



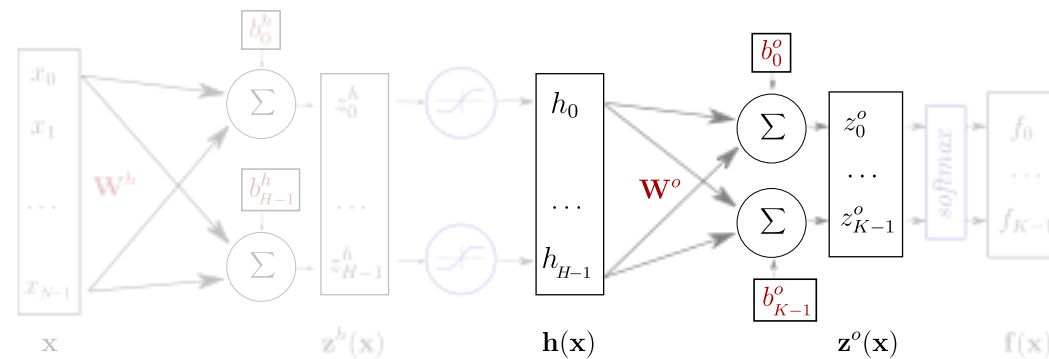
- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h \mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h \mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = \text{softmax}(\mathbf{z}^o) = \text{softmax}(\mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o)$

One Hidden Layer Network



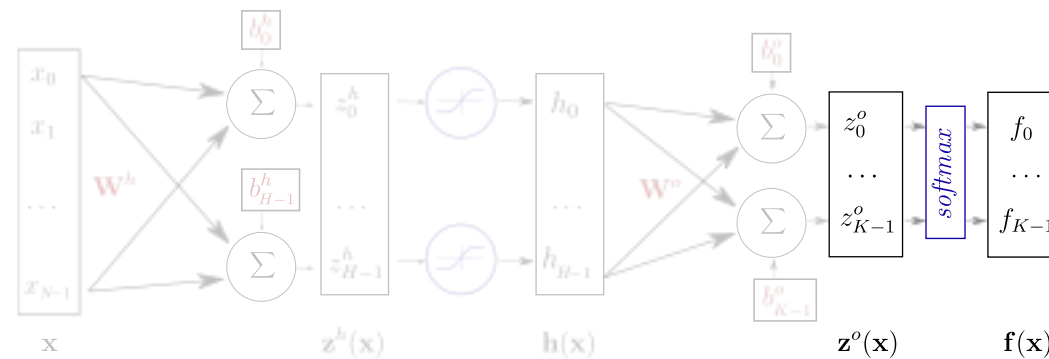
- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h \mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h \mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = \text{softmax}(\mathbf{z}^o) = \text{softmax}(\mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o)$

One Hidden Layer Network



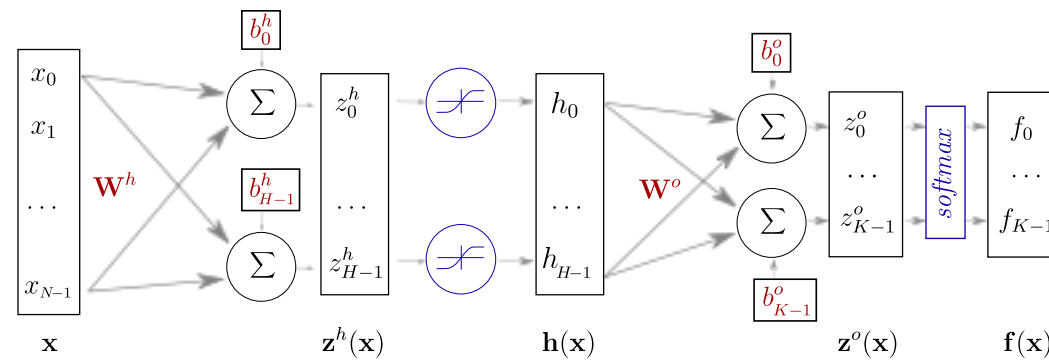
- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h \mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h \mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = \text{softmax}(\mathbf{z}^o) = \text{softmax}(\mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o)$

One Hidden Layer Network

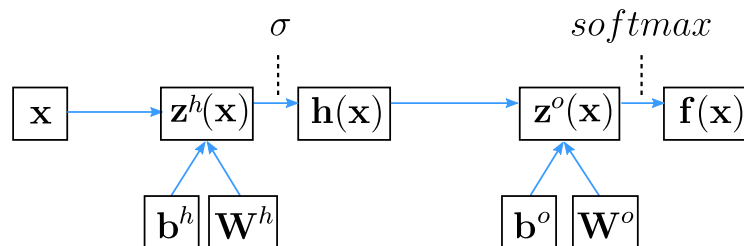


- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h \mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h \mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = \text{softmax}(\mathbf{z}^o) = \text{softmax}(\mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o)$

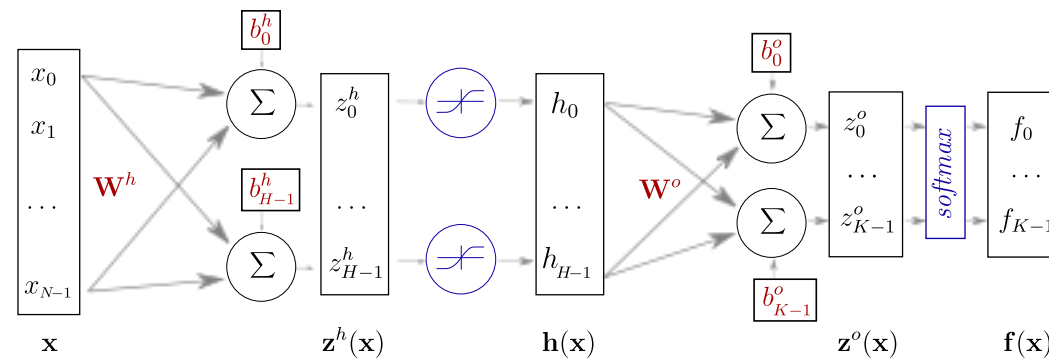
One Hidden Layer Network



Alternate representation



One Hidden Layer Network



Keras implementation

```
model = Sequential()
model.add(Dense(H, input_dim=N)) # weight matrix dim [N * H]
model.add(Activation("tanh"))
model.add(Dense(K)) # weight matrix dim [H x K]
model.add(Activation("softmax"))
```

Softmax function

$$\textit{softmax}(\mathbf{x}) = \frac{1}{\sum_{i=1}^n e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

Softmax function

$$\text{softmax}(\mathbf{x}) = \frac{1}{\sum_{i=1}^n e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

- vector of values in (0, 1) that add up to 1
- for example, $\mathbf{x} = [1, 2, 3]$ becomes $\frac{1}{e^1 + e^2 + e^3} \cdot [e^1, e^2, e^3]^T = [0.09, 0.24, 0.67]$
- $p(Y = c | X = \mathbf{x}) = \text{softmax}(\mathbf{z}(\mathbf{x}))_c$
- the pre-activation vector $\mathbf{z}(\mathbf{x})$ is often called "the logits"

Training the network

Find parameters that minimize the negative log likelihood (or [cross entropy](#))

Training the network

Find parameters that minimize the negative log likelihood (or [cross entropy](#))

The loss function for a given sample $s \in S$:

$$l(\mathbf{f}(\mathbf{x}^s; \theta), y^s) = nll(\mathbf{x}^s, y^s; \theta) = -\log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s}$$

Training the network

Find parameters that minimize the negative log likelihood (or [cross entropy](#))

The loss function for a given sample $s \in S$:

$$l(\mathbf{f}(\mathbf{x}^s; \theta), y^s) = nll(\mathbf{x}^s, y^s; \theta) = -\log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s}$$

example $y^s = 3$

$$l(\mathbf{f}(\mathbf{x}^s; \theta), y^s) = l \left(\begin{array}{c} f_0 \\ \dots \\ f_3 \\ \dots \\ f_{K-1} \end{array}, \begin{array}{c} 0 \\ \dots \\ 1 \\ \dots \\ 0 \end{array} \right) = -\log f_3$$

Training the network

Find parameters $\theta = (\mathbf{W}^h; \mathbf{b}^h; \mathbf{W}^o; \mathbf{b}^o)$ that minimize the negative log likelihood (or [cross entropy](#))

The loss function for a given sample $s \in S$:

$$l(\mathbf{f}(\mathbf{x}^s; \theta), y^s) = nll(\mathbf{x}^s, y^s; \theta) = -\log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s}$$

The cost function is the negative likelihood of the model computed on the full training set (for i.i.d. samples):

$$L_S(\theta) = -\frac{1}{|S|} \sum_{s \in S} \log \mathbf{f}(\mathbf{x}^s; \theta)_{y^s}$$

Training the network

- Now we have a mathematical function representing the network
- And we have a way of measuring how good it is
- How do we find the *parameters* that minimize the *loss*?

Gradient Descent

- Let's imagine we only have one parameter θ

Gradient Descent

- Let's imagine we only have one parameter θ
- We can compute the derivative of the loss with respect to θ

Gradient Descent

- Let's imagine we only have one parameter θ
- We can compute the derivative of the loss with respect to θ
- The derivative, $\frac{dL}{d\theta}$, tells us the slope of the loss function at a given point

Gradient Descent

- Let's imagine we only have one parameter θ
- We can compute the derivative of the loss with respect to θ
- The derivative, $\frac{dL}{d\theta}$, tells us the slope of the loss function at a given point
- If $\frac{dL}{d\theta} > 0$, increasing θ will increase the loss, and vice versa

Gradient Descent

- Let's imagine we only have one parameter θ
- We can compute the derivative of the loss with respect to θ
- The derivative, $\frac{dL}{d\theta}$, tells us the slope of the loss function at a given point
- If $\frac{dL}{d\theta} > 0$, increasing θ will increase the loss, and vice versa
- To minimize loss, we adjust θ in the opposite direction of $\frac{dL}{d\theta}$

Gradient Descent

- Let's imagine we only have one parameter θ
- We can compute the derivative of the loss with respect to θ
- The derivative, $\frac{dL}{d\theta}$, tells us the slope of the loss function at a given point
- If $\frac{dL}{d\theta} > 0$, increasing θ will increase the loss, and vice versa
- To minimize loss, we adjust θ in the opposite direction of $\frac{dL}{d\theta}$
- This is done using the update rule: $\theta = \theta_{\text{old}} - \eta \frac{dL}{d\theta}$

Gradient Descent

- We can use gradient descent to play "guess what number I'm thinking of"
- If your guess is too high, you decrease it
- If your guess is too low, you increase it
- The error function is a parabola
- By finding the lowest point on the parabola, you find the best guess

Implementing Gradient Descent

- Start with an initial guess for θ
- Calculate $\frac{dL}{d\theta}$ using the current value of θ
- Update θ using the update rule
- Repeat the process until the change in loss is below a threshold or a set number of iterations is reached
- The choice of learning rate η is crucial: too high, and we may overshoot the minimum; too low, and convergence will be slow

Stochastic Gradient Descent

- Traditional Gradient Descent uses the entire dataset to compute the gradient, which can be computationally expensive
- Stochastic Gradient Descent (SGD) updates the parameters using only a single data point (or a small batch)
- In SGD, for each iteration, a data point (or batch) is randomly selected to compute the gradient
- Since only a subset of data is used, the gradient estimation can be noisy, leading to a less smooth path towards the minimum
- However, SGD is much faster than traditional gradient descent

Stochastic Gradient Descent

Initialize θ randomly

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)
 - Compute gradients: $\Delta = \nabla_{\theta} L_B(\theta)$

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)
 - Compute gradients: $\Delta = \nabla_{\theta} L_B(\theta)$
 - Update parameters: $\theta \leftarrow \theta - \eta \Delta$

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)
 - Compute gradients: $\Delta = \nabla_{\theta} L_B(\theta)$
 - Update parameters: $\theta \leftarrow \theta - \eta \Delta$
- Repeat until the epoch is completed (all of S is covered)

Stochastic Gradient Descent

Initialize θ randomly

For E epochs perform:

- Randomly select a small batch of samples ($B \subset S$)
 - Compute gradients: $\Delta = \nabla_{\theta} L_B(\theta)$
 - Update parameters: $\theta \leftarrow \theta - \eta \Delta$
- Repeat until the epoch is completed (all of S is covered)

Stop when reaching criterion:

- nll stops decreasing when computed on validation set

Computing Gradients

Output Weights: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial W_{ij}^o}$

Output bias: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial b_i^o}$

Hidden Weights: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial W_{ij}^h}$

Hidden bias: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial b_i^h}$

Computing Gradients

Output Weights: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial W_{ij}^o}$

Output bias: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial b_i^o}$

Hidden Weights: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial W_{ij}^h}$

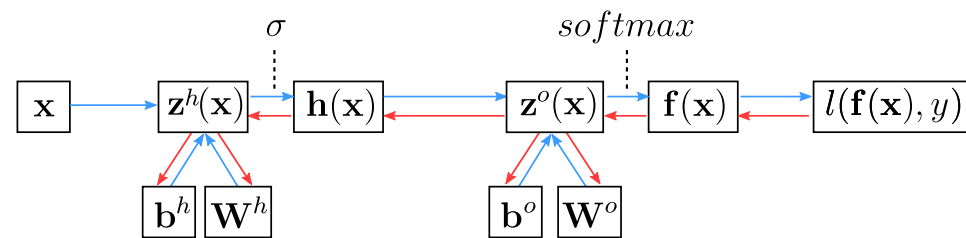
Hidden bias: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial b_i^h}$

- The network is a composition of differentiable modules
- We can apply the "chain rule"

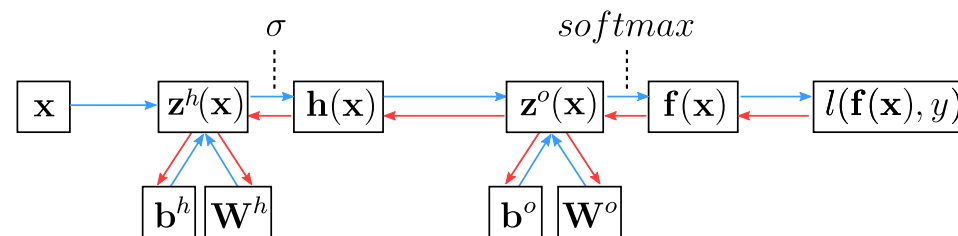
Chain rule

- Mathematical theorem that lets us compute derivatives when functions are inside other functions
- Remember, our neural network is a composition of functions: $f(x) = g(h(x))$
- The chain rule tells us how to compute $\frac{df}{dx}$
- $\frac{df}{dx} = \frac{df}{dg} \frac{dg}{dh} \frac{dh}{dx}$
- In English: The derivative of the overall network with respect to its input is the product of derivatives of each function in the network

Backpropagation



Backpropagation



- Compute partial derivatives of the loss
- For any given function in the network, we can compute how changing its parameters will affect the loss
- In other words, we can find how much each parameter's value contributes to the loss

Initialization and Learning Tricks

Initialization and normalization

- Input data should be normalized to have approx. same range:
 - standardization or quantile normalization

Initialization and normalization

- Input data should be normalized to have approx. same range:
 - standardization or quantile normalization
- Initializing weights:
 - Zero is a saddle point: no gradient, no learning

Initialization and normalization

- Input data should be normalized to have approx. same range:
 - standardization or quantile normalization
- Initializing weights:
 - Zero is a saddle point: no gradient, no learning
 - Constant init: all neurons compute the same function

Initialization and normalization

- Input data should be normalized to have approx. same range:
 - standardization or quantile normalization
- Initializing weights:
 - Zero is a saddle point: no gradient, no learning
 - Constant init: all neurons compute the same function
 - Solution: random init, ex: $w \sim \mathcal{N}(0, 0.01)$

Initialization and normalization

- Input data should be normalized to have approx. same range:
 - standardization or quantile normalization
- Initializing weights:
 - Zero is a saddle point: no gradient, no learning
 - Constant init: all neurons compute the same function
 - Solution: random init, ex: $w \sim \mathcal{N}(0, 0.01)$
 - Better inits: Xavier Glorot and Kaming He & orthogonal

Initialization and normalization

- Input data should be normalized to have approx. same range:
 - standardization or quantile normalization
- Initializing weights:
 - Zero is a saddle point: no gradient, no learning
 - Constant init: all neurons compute the same function
 - Solution: random init, ex: $w \sim \mathcal{N}(0, 0.01)$
 - Better inits: Xavier Glorot and Kaming He & orthogonal
- Biases can (should) be initialized to zero

SGD learning rate

- Very sensitive:
 - Too high → early plateau or even divergence
 - Too low → slow convergence

SGD learning rate

- Very sensitive:
 - Too high → early plateau or even divergence
 - Too low → slow convergence
 - Try a large value first: $\eta = 0.1$ or even $\eta = 1$
 - Divide by 10 and retry in case of divergence

SGD learning rate

- Very sensitive:
 - Too high \rightarrow early plateau or even divergence
 - Too low \rightarrow slow convergence
 - Try a large value first: $\eta = 0.1$ or even $\eta = 1$
 - Divide by 10 and retry in case of divergence
- Large constant LR prevents final convergence
 - multiply η_t by $\beta < 1$ after each update

SGD learning rate

- Very sensitive:
 - Too high \rightarrow early plateau or even divergence
 - Too low \rightarrow slow convergence
 - Try a large value first: $\eta = 0.1$ or even $\eta = 1$
 - Divide by 10 and retry in case of divergence
- Large constant LR prevents final convergence
 - multiply η_t by $\beta < 1$ after each update
 - or monitor validation loss and divide η_t by 2 or 10 when no progress
 - See [ReduceLROnPlateau](#) in Keras

Momentum

Accumulate gradients across successive updates:

$$m_t = \gamma m_{t-1} + \eta \nabla_{\theta} L_{B_t}(\theta_{t-1})$$
$$\theta_t = \theta_{t-1} - m_t$$

γ is typically set to 0.9

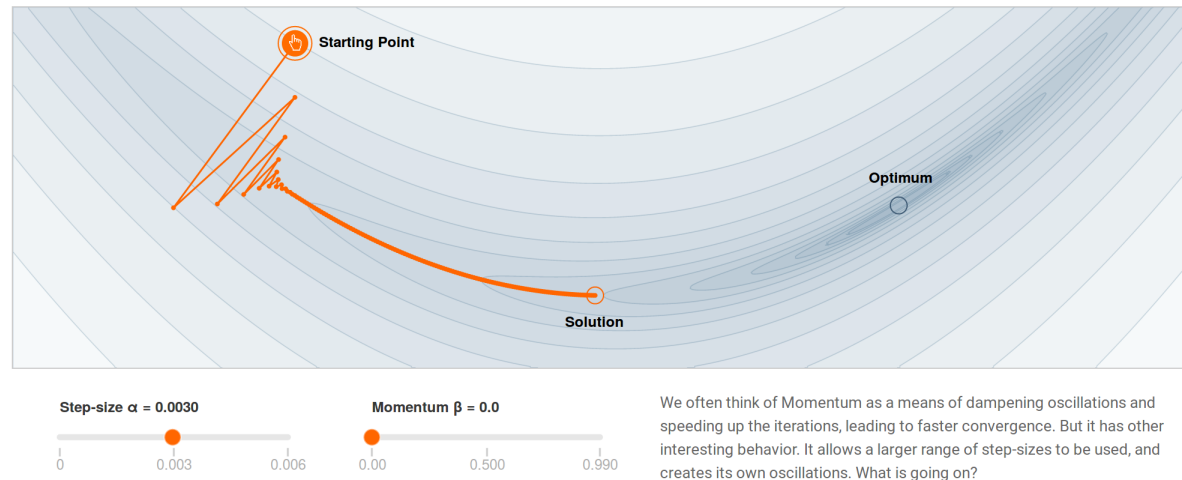
Momentum

Accumulate gradients across successive updates:

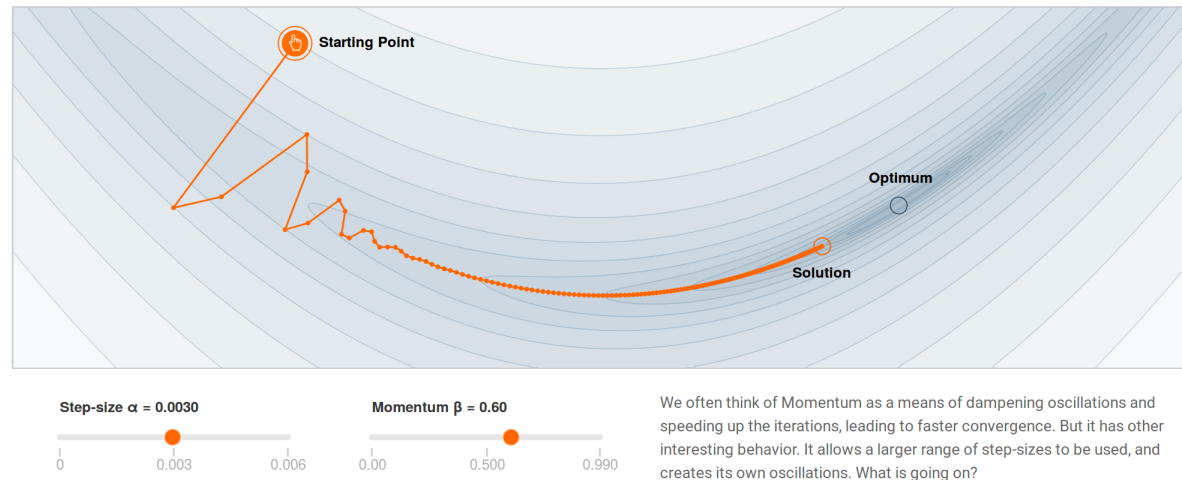
$$m_t = \gamma m_{t-1} + \eta \nabla_{\theta} L_{B_t}(\theta_{t-1})$$
$$\theta_t = \theta_{t-1} - m_t$$

γ is typically set to 0.9

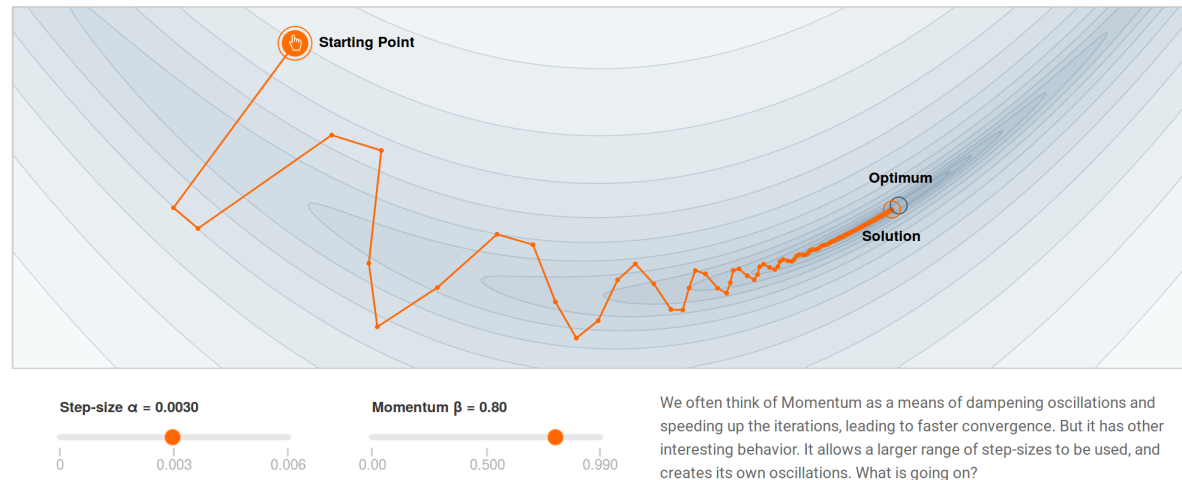
Larger updates in directions where the gradient sign is constant to accelerate in low curvature areas



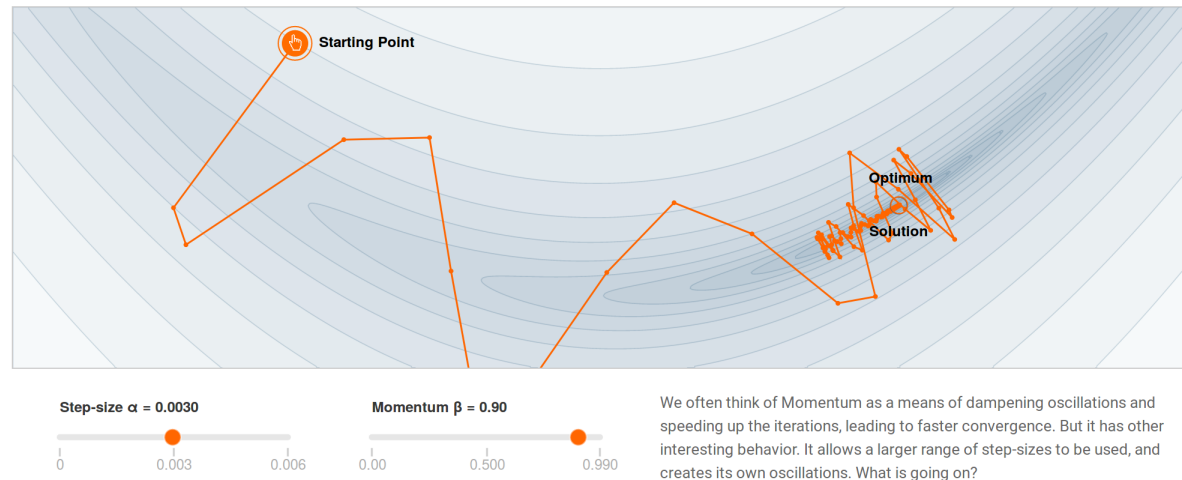
[Why Momentum Really Works](#)



Why Momentum Really Works



[Why Momentum Really Works](#)



Why Momentum Really Works

Alternative optimizers

- SGD (with momentum)
 - Simple to implement
 - Very sensitive to initial value of η
 - Need learning rate scheduling

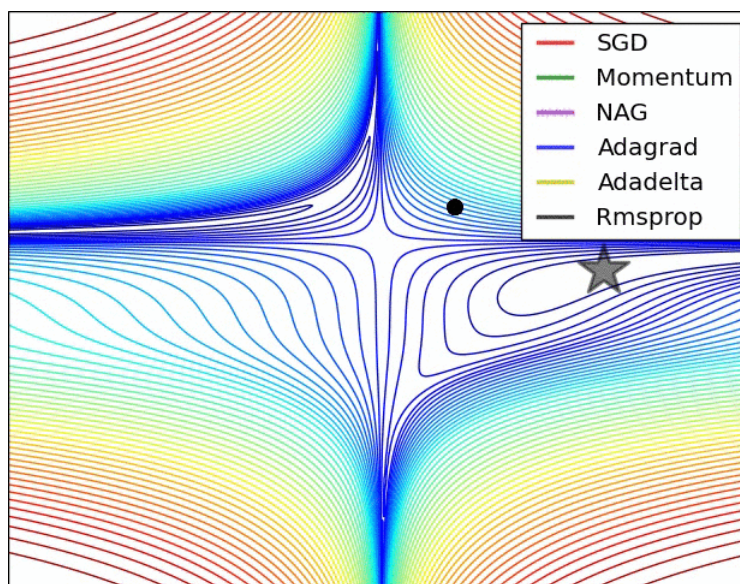
Alternative optimizers

- SGD (with momentum)
 - Simple to implement
 - Very sensitive to initial value of η
 - Need learning rate scheduling
- Adam: adaptive learning rate scale for each param
 - Global η set to $3e-4$ often works well enough
 - Good default choice of optimizer (often)

Alternative optimizers

- SGD (with momentum)
 - Simple to implement
 - Very sensitive to initial value of η
 - Need learning rate scheduling
- Adam: adaptive learning rate scale for each param
 - Global η set to $3e-4$ often works well enough
 - Good default choice of optimizer (often)
- Many other promising methods:
 - RMSProp, Adagrad, Adadelata, Nadam, ...
 - Often takes some experimentation to find the best one

Optimizers around a saddle point



Credits: Alec Radford