Deep Learning

Neural networks and Backpropagation

Alex Olson

Adapted from material by Charles Ollion & Olivier Grisel

Neural Network for classification

Vector function with tunable parameters θ

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

Neural Network for classification

Vector function with tunable parameters θ

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

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 θ

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

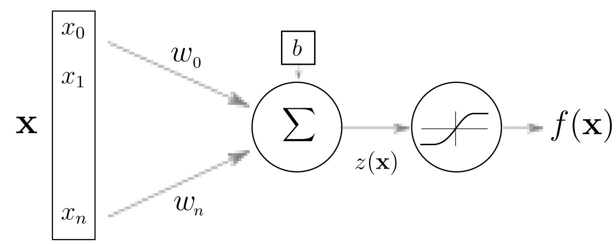
Sample s in dataset S:

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

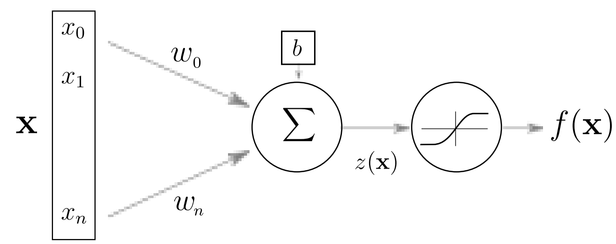
Output is a conditional probability distribution:

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

Artificial Neuron



Artificial Neuron

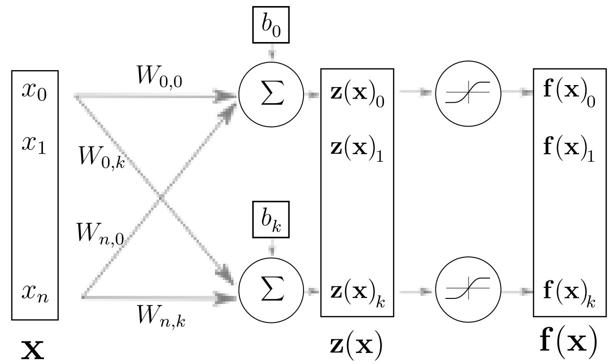


$$z(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$$

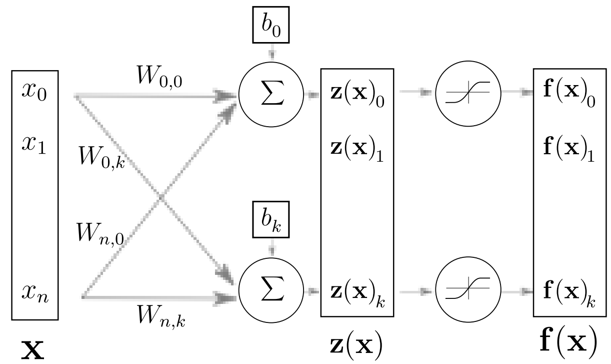
$$f(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + b)$$

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

Layer of Neurons



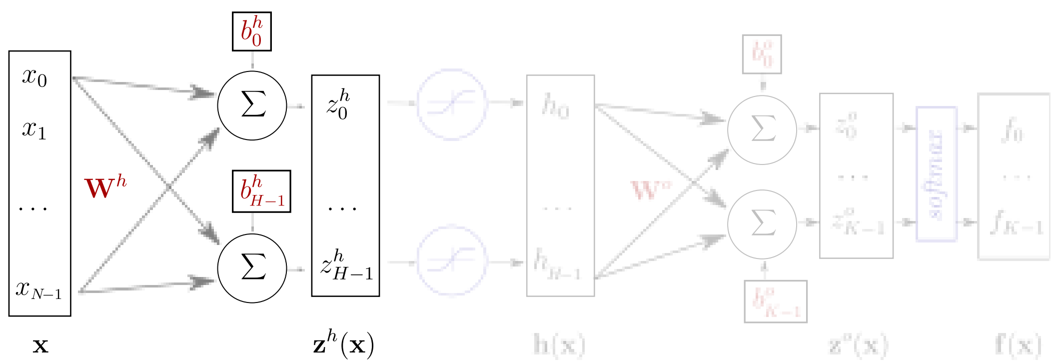
Layer of Neurons



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

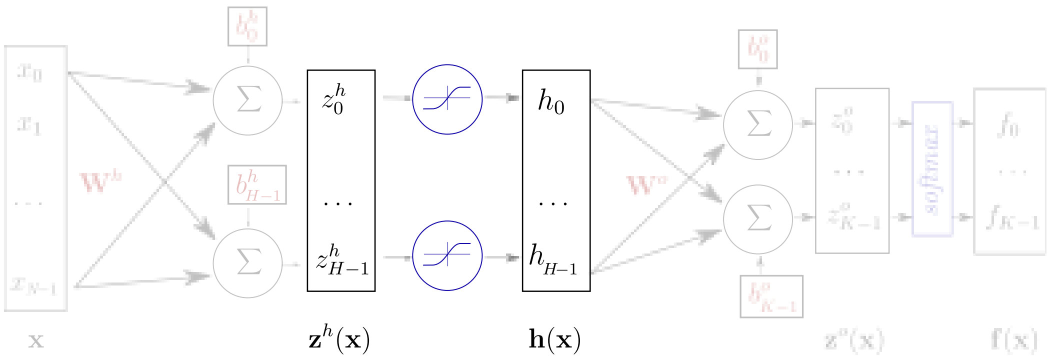
- $\mathbf{W}, \mathbf{b}$ now matrix and vector

One Hidden Layer Network



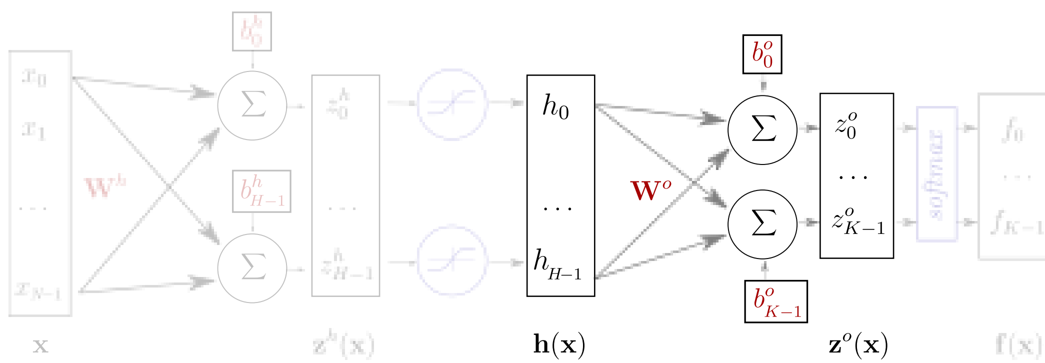
- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h \mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h \mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o \mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = softmax(\mathbf{z}^o)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

One Hidden Layer Network



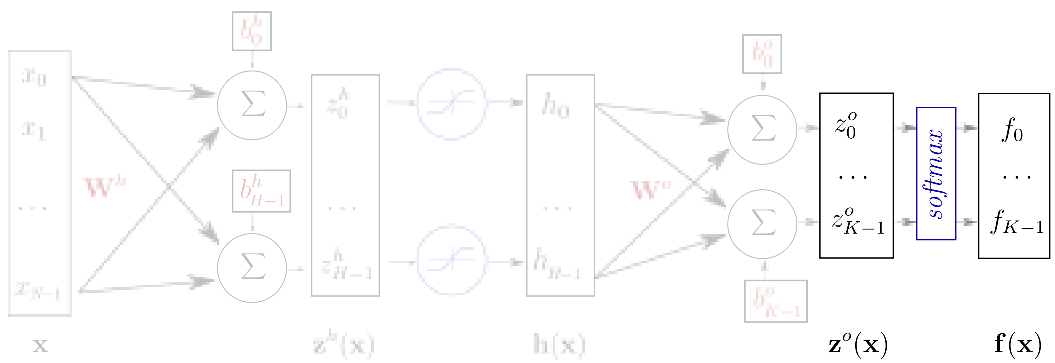
- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h\mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h\mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = softmax(\mathbf{z}^o) =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

One Hidden Layer Network



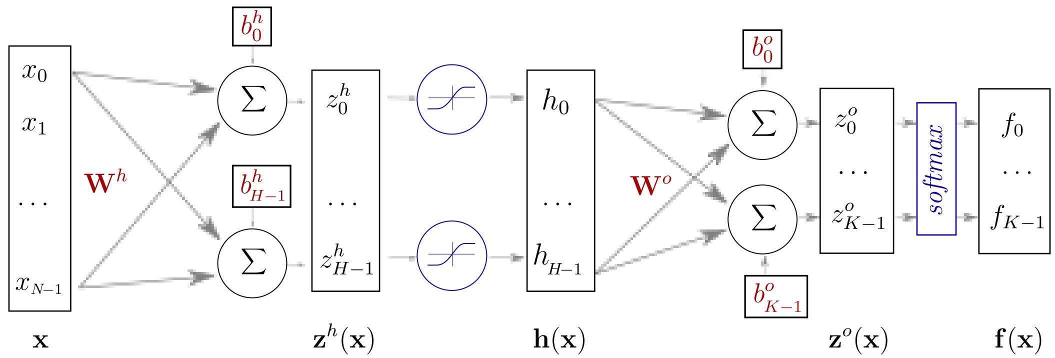
- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h\mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h\mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = softmax(\mathbf{z}^o) =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

One Hidden Layer Network

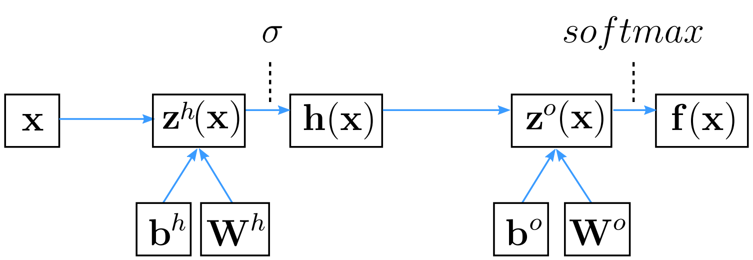


- $\mathbf{z}^h(\mathbf{x}) = \mathbf{W}^h\mathbf{x} + \mathbf{b}^h$
- $\mathbf{h}(\mathbf{x}) = g(\mathbf{z}^h(\mathbf{x})) = g(\mathbf{W}^h\mathbf{x} + \mathbf{b}^h)$
- $\mathbf{z}^o(\mathbf{x}) = \mathbf{W}^o\mathbf{h}(\mathbf{x}) + \mathbf{b}^o$
- $\mathbf{f}(\mathbf{x}) = softmax(\mathbf{z}^o) =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

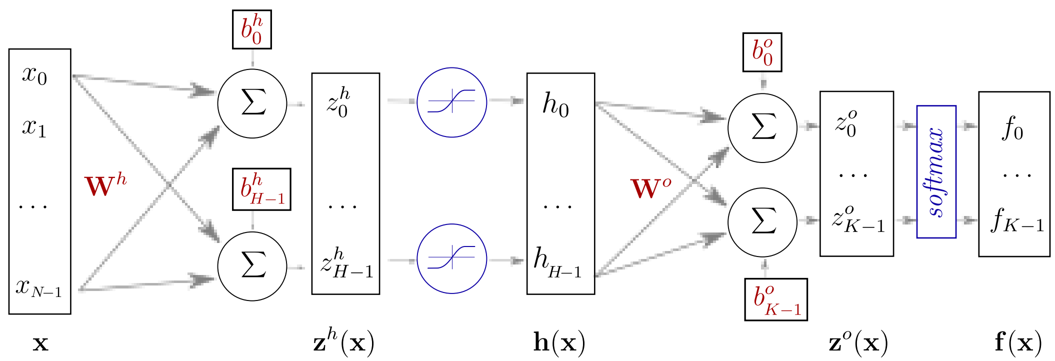
One Hidden Layer Network



Alternate representation



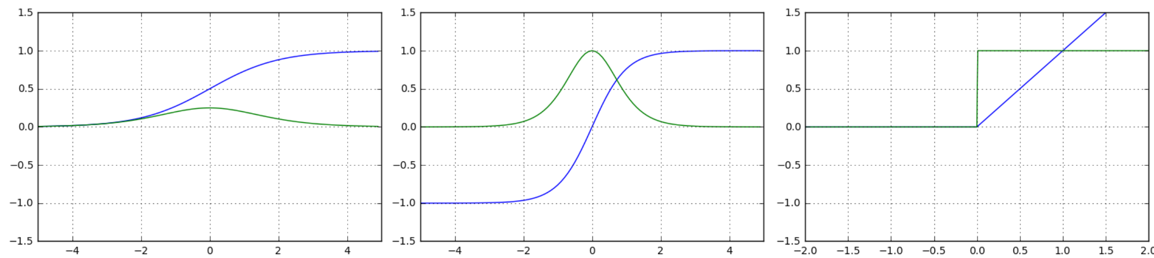
One Hidden Layer Network



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"))
```

Element-wise activation functions



$$\mathrm{sigm}(x) = \frac{1}{1+e^{-x}} \qquad \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \qquad \mathrm{relu}(x) = \max(0, x)$$

$$\mathrm{sigm}'(x) = \mathrm{sigm}(x)(1-\mathrm{sigm}(x)) \qquad \tanh'(x) = 1 - \tanh(x)^2 \qquad \mathrm{relu}'(x) = 1_{x>0}$$

- blue: activation function
- green: derivative

Softmax function

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

$$\frac{\partial softmax(\mathbf{x})_i}{\partial x_j} = \begin{cases} softmax(\mathbf{x})_i \cdot (1 - softmax(\mathbf{x})_i) & i = j \\ -softmax(\mathbf{x})_i \cdot softmax(\mathbf{x})_j & i \neq j \end{cases}$$

Softmax function

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

$$\frac{\partial softmax(\mathbf{x})_i}{\partial x_j} = \begin{cases} softmax(\mathbf{x})_i \cdot (1 - softmax(\mathbf{x})_i) & i = j \\ -softmax(\mathbf{x})_i \cdot softmax(\mathbf{x})_j & i \neq j \end{cases}$$

- vector of values in $(0, 1)$ that add up to 1
- $p(Y = c|X = \mathbf{x}) = \text{softmax}(\mathbf{z}(\mathbf{x}))_c$
- the pre-activation vector $\mathbf{z}(\mathbf{x})$ is often called "the logits"

Training the network

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

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 $s \in S$:

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

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 $s \in S$:

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

example $y^s = 3$

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

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):

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

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: $\theta \leftarrow \theta - \eta\Delta$
  - $\eta > 0$ is called the learning rate

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: $\theta \leftarrow \theta - \eta\Delta$
  - $\eta > 0$ is called the learning rate
- 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: $\theta \leftarrow \theta - \eta\Delta$
  - $\eta > 0$ is called the learning rate
- 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: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial W^o_{i,j}}$

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

Hidden Weights: $\frac{\partial l(\mathbf{f}(\mathbf{x}), y)}{\partial W^h_{i,j}}$

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

Computing Gradients

Output Weights: $\frac{\partial l(\mathbf{f}(\mathbf{x}),y)}{\partial W^o_{i,j}}$

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

Hidden Weights: $\frac{\partial l(\mathbf{f}(\mathbf{x}),y)}{\partial W^h_{i,j}}$

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

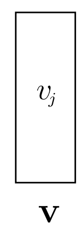
- The network is a composition of differentiable modules
- We can apply the "chain rule"

Chain rule

$$x \rightarrow \mathbf{v} \rightarrow z$$

$$z = u(\mathbf{v}(x))$$

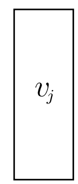
$$\frac{\partial z}{\partial x} = \text{?}$$



Chain rule

$$x \longrightarrow \mathbf{v} \longrightarrow z$$

$$z = u(\mathbf{v}(x))$$



$\mathbf{v}$

chain-rule

$$\frac{\partial z}{\partial x} = \sum_j \frac{\partial z}{\partial v_j} \frac{\partial v_j}{\partial x}$$

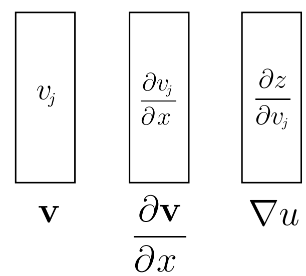
Chain rule

$$x \longrightarrow \mathbf{v} \longrightarrow z$$

$$z = u(\mathbf{v}(x))$$

chain-rule

$$\frac{\partial z}{\partial x} = \sum_j \frac{\partial z}{\partial v_j}\frac{\partial v_j}{\partial x} = \nabla u . \frac{\partial \mathbf{v}}{\partial x}$$



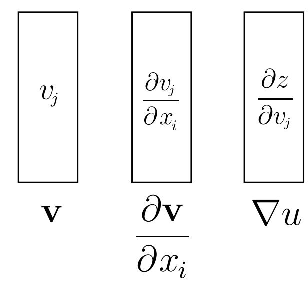
Chain rule

$$\boxed{\mathbf{x}} \longrightarrow \boxed{\mathbf{v}} \longrightarrow \boxed{z}$$

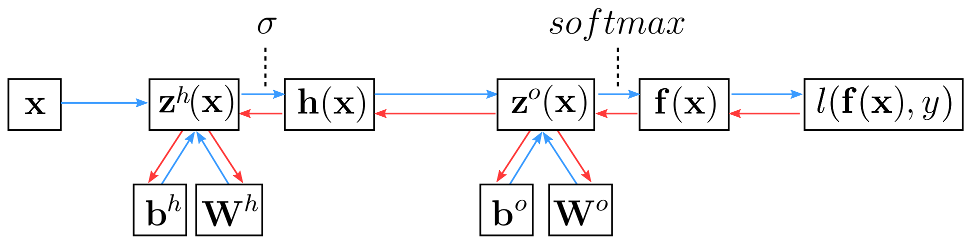
$$z = u(\mathbf{v}(\mathbf{x}))$$

chain-rule

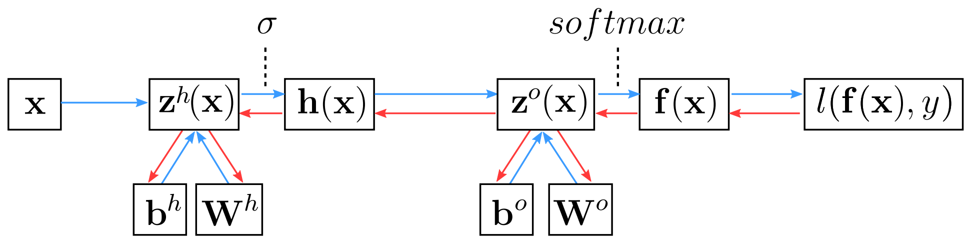
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial v_j}\frac{\partial v_j}{\partial x_i} = \nabla u . \frac{\partial \mathbf{v}}{\partial x_i}$$



Backpropagation



Backpropagation



Compute partial derivatives of 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 → 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
- 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:

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

γ is typically set to 0.9

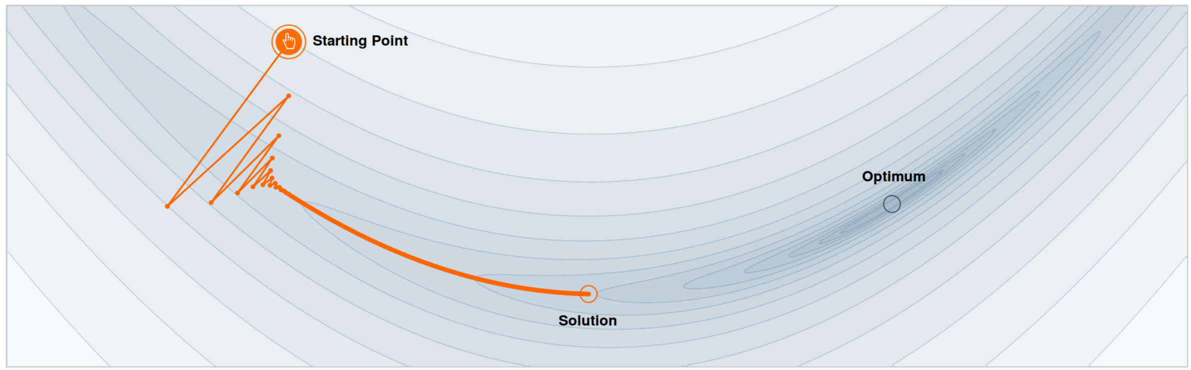
Momentum

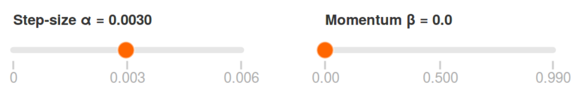
Accumulate gradients across successive updates:

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

γ is typically set to 0.9

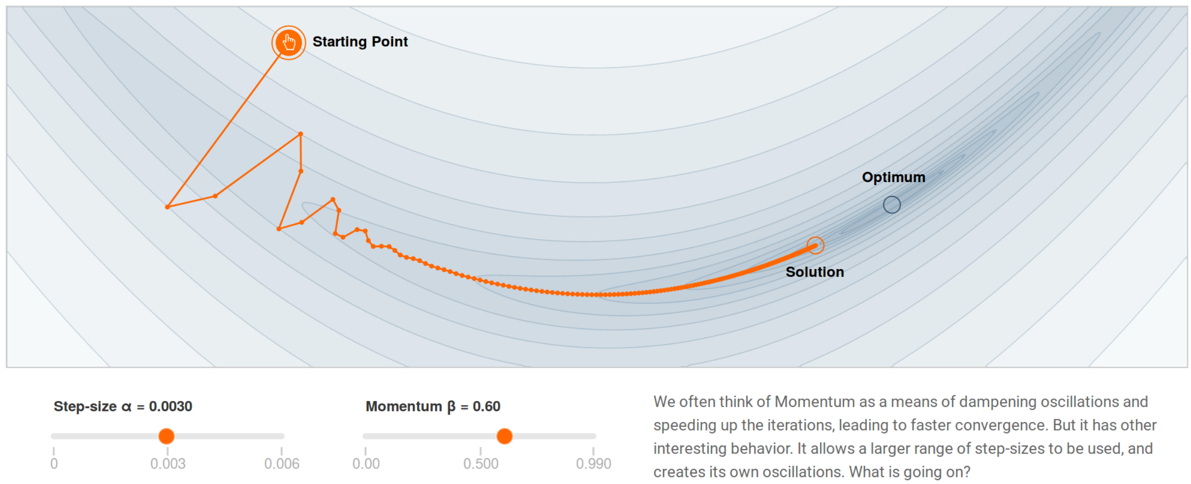
Larger updates in directions where the gradient sign is constant to accelerate in low curvature areas





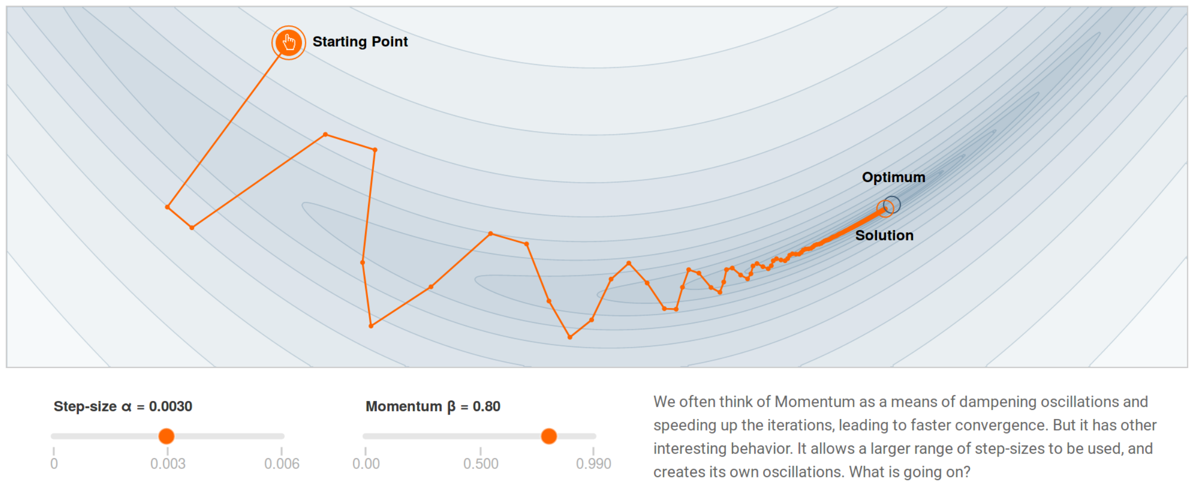
We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

[Why Momentum Really Works]



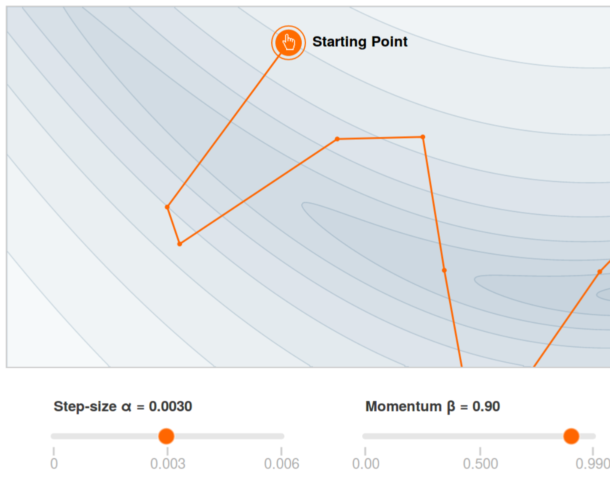
We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

[Why Momentum Really Works]



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

[Why Momentum Really Works]



We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

[Why Momentum Really Works]

Alternative optimizers

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

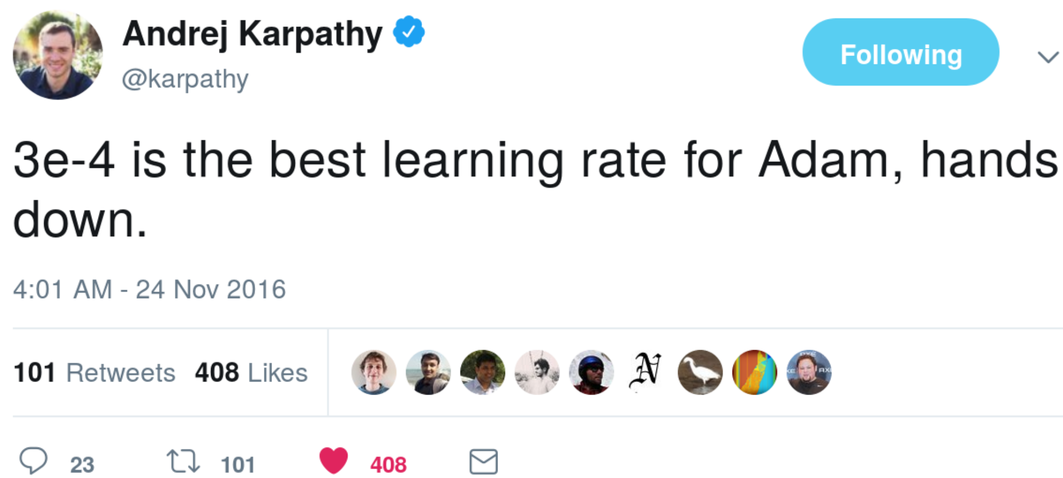
Alternative optimizers

- 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)

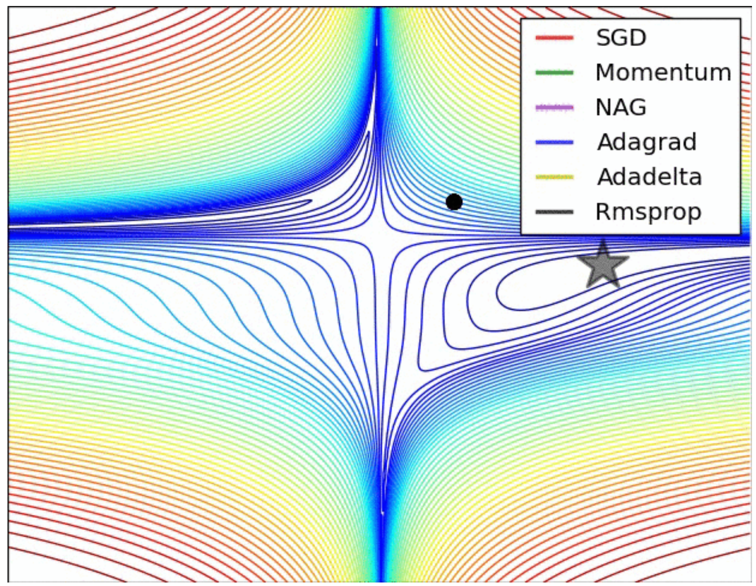
Alternative optimizers

- 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, Adadelta, Nadam, ...
  - Often takes some experimentation to find the best one

The Karpathy Constant for Adam



Optimizers around a saddle point



Credits: Alec Radford

Next: Lab 2!