Machine Learning from Scratch:
Stochastic Gradient Descent and Adam Optimizer

JAMES GABBARD AND DANIEL MILLER

18.0851 FINAL PROJECT

Introduction

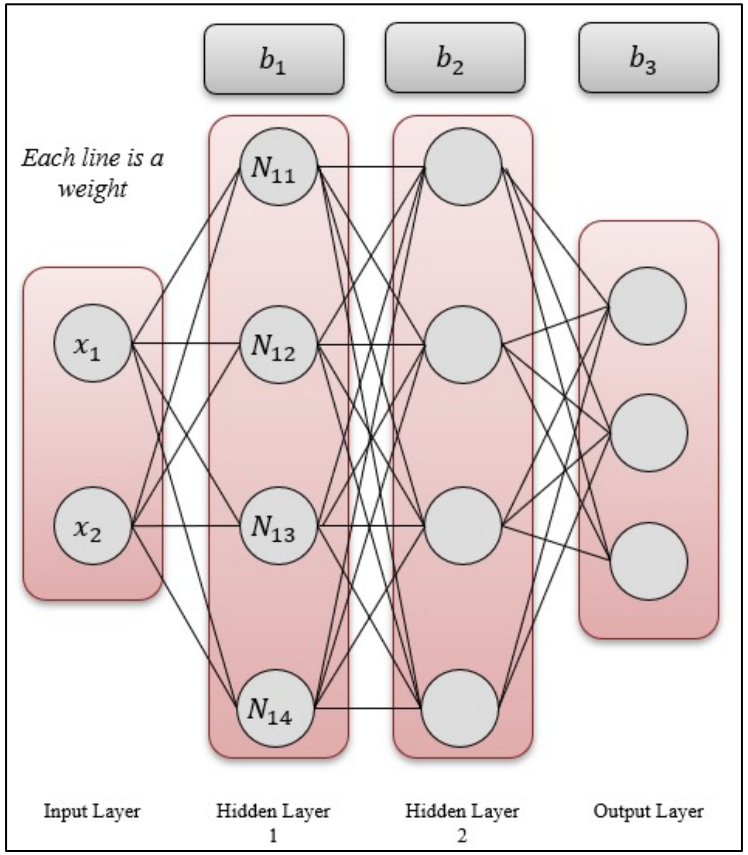
- Neural networks are an increasingly important part of research and daily life.
- While their applications are widely discussed, the algorithms that allow them to learn are not.
- How do they actually learn?
  - Optimization Algorithms:
    - Stochastic Gradient Descent and Backpropagation
    - Adam Optimizer
  - Application test:
    - Tensorflow Playground-style classification problems
    - Reinforcement Learning: CartPole
- All coding completed in MATLAB

Neural Network Overview

- Composed of layers *d* deep and *n* neurons wide
  - Network shown here: 2 layers
- Connected via weights and biases
- Activation functions add nonlinearities

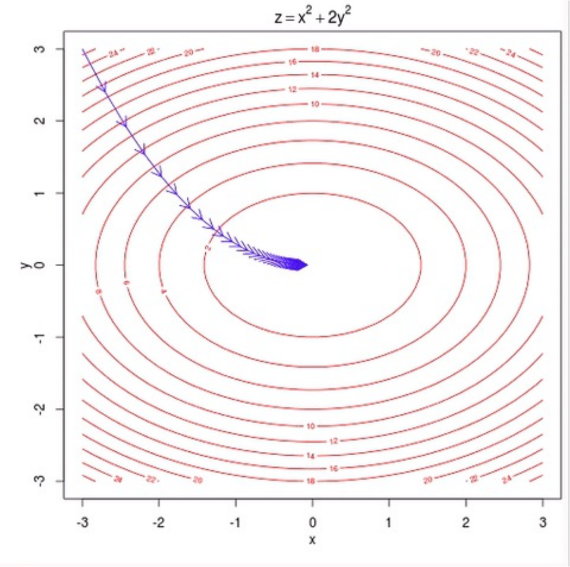
$$N_1 = Activation(W_1 x^T + b_1)$$

- In order for a network to learn, a method must be devised to shape these weights and biases to produce an accurate output.
  - Minimize a cost function.
    Classification problem: Cost function is the error between data labels and neural network classification



Stochastic Gradient Descent (SGD)

- With a set of data X, SGD optimizes minimizing an objective function $J(\theta)$ using a randomly (STOCHASTIC) selected minibatch of data by via GRADIENT DESCENT, $-\nabla_{\theta} J(\theta)$.



Algorithm 2 Stochastic Gradient Descent (SGD)

```
procedure SGD TRAINING(X)
    initialize θ = (W, b) to small random numbers
    randomize order of training examples in X
    while not converged do
        Create minibatch of length B from X
        a_1 = minibatch_j
        ∇_θ = Backpropagation(X, θ)
        for i ← [2, L] do
            θ_i ← θ_i + (η/B) ∇_θ
        end for
    end while
end procedure
```

Backpropagation (In 5 Minutes)

$W^1, \dots W^L$ and $b^1, \dots b^L$ are weight matrices and bias vectors

σ^ℓ is the activation function at layer ℓ

$z^\ell = W^\ell a^{\ell-1} + b^\ell$ is the weighted input to layer ℓ

a^ℓ is the activation at layer ℓ

C is our cost function

$\delta_\ell = \frac{\partial C}{\partial z_\ell}$ is the "error", a quantity central to backpropagation

For elementwise multiplication ("dot star"), we'll use $\odot$

$$c = a \odot b \quad \leftrightarrow \quad c_i = a_i b_i$$

Plan of Attack

Plan of attack:

Forward pass: compute a's (activations) and z's (weighted inputs) for each layer

Backward pass:

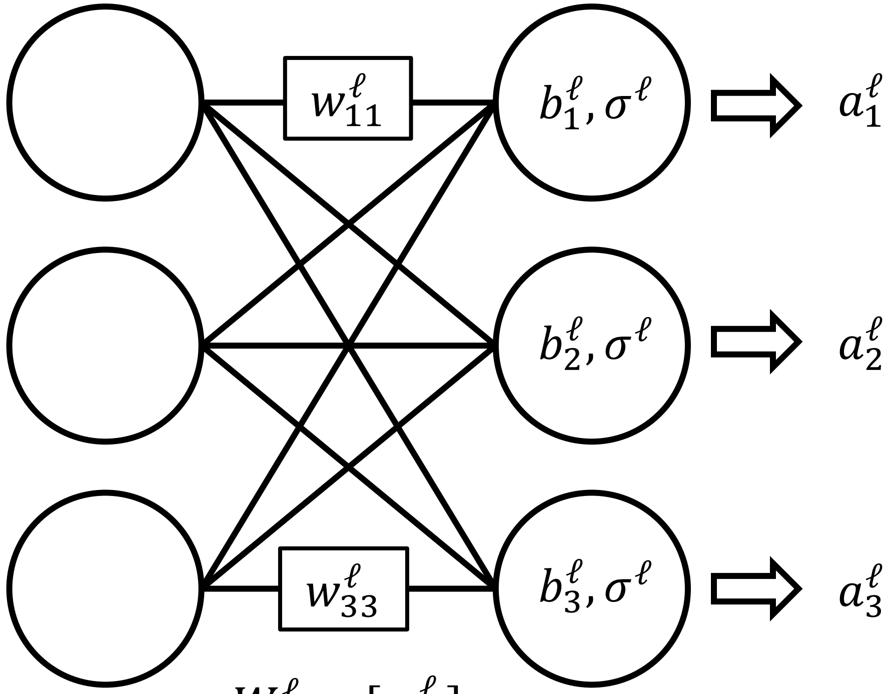
$$\delta^{\ell} = \frac{\partial C}{\partial z_{\ell}} = \frac{\partial C}{\partial a_L} \times \frac{\partial a_L}{\partial z_L} \times \frac{\partial z_L}{\partial a_{L-1}} \times \frac{\partial a_{L-1}}{\partial z_{L-1}} \times \frac{\partial z_{L-1}}{\partial a_{L-2}} \times \cdots$$

Compute Gradients:

$$\frac{\partial C}{\partial b}, \frac{\partial C}{\partial W}$$

Feed Forward: Compute a^ℓ and z^ℓ

Layer $\ell - 1$ Layer ℓ



$W^\ell = [w_{ij}^\ell]$

Feed Forward:

$$z_i^\ell = \sum_j w_{ij}^\ell a_j^{\ell-1} + b_i^\ell$$

$$a_i^\ell = \sigma^\ell(z_i^\ell)$$

Derivatives:

$$\frac{\partial z_i^\ell}{\partial a_j^{\ell-1}} = w_{ij}^\ell$$

$$\frac{da_i^\ell}{dz_i^\ell} = \sigma'(z_i^\ell)$$

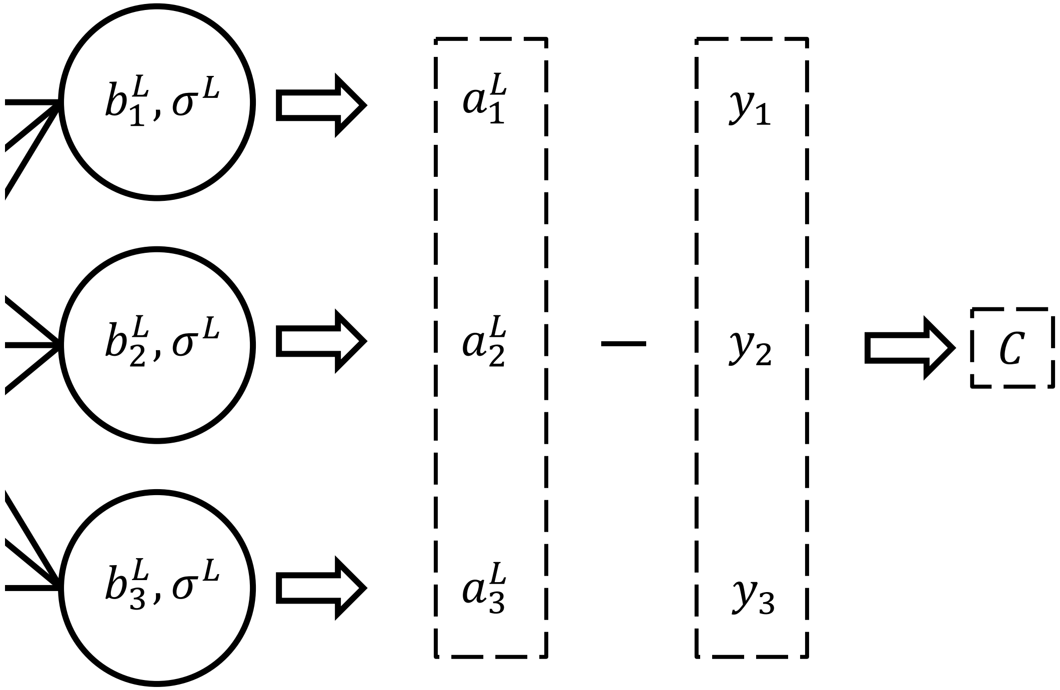
Vector Notation:

$$z^\ell = W^\ell a^{\ell-1} + b^\ell$$

$$a^\ell = \sigma^\ell(z^\ell)$$

The Last Layer: Compute $\delta^L = \partial C / \partial z_L$

Layer L



Quadratic Cost Function:

$$C = \frac{1}{2} \sum_i \left(a_i^L - y_i\right)^2$$
$$= \frac{1}{2} \sum_i \left[\sigma^L\left(z_i^L\right) - y_i\right]^2$$

Taking Derivatives:

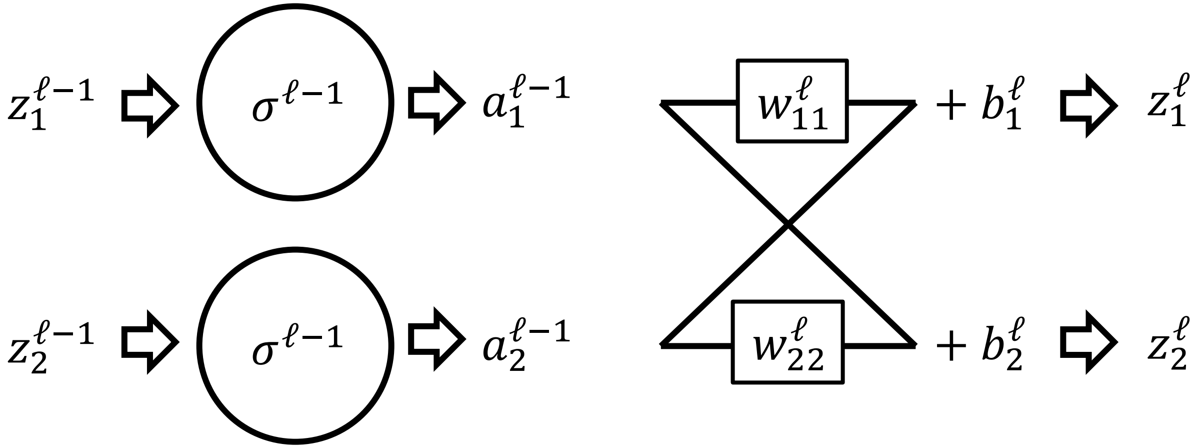
$$\frac{\partial C}{\partial z_i^L} = \left[\sigma^L\left(z_i^L\right) - y_i\right] \sigma'(z_i^L)$$

Substitute:

Vector Notation:

$$\delta_i^L = \left(a_i^L - y_i\right) \sigma'(z_i^L) \Rightarrow \delta^L = (a^L - y) \odot \sigma'(z^L)$$

Propagating Error: Given δ^ℓ, Compute $\delta^{\ell-1}$



$$\frac{\partial z_i^\ell}{\partial a_j^{\ell-1}} = w_{ij}^\ell$$

$$\frac{d a_i^{\ell-1}}{d z_i^{\ell-1}} = \sigma'(z_i^{\ell-1})$$

Chain Rule:

$$\frac{\partial C}{\partial a_i^{\ell-1}} = \sum_j \frac{\partial C}{\partial z_j^\ell} \times \frac{\partial z_j^\ell}{\partial a_i^{\ell-1}}$$

Substitute:

$$\Rightarrow \frac{\partial C}{\partial a_i^{\ell-1}} = \sum_j \delta_j^\ell w_{ji}^\ell$$

Vector Notation:

$$\Rightarrow \frac{\partial C}{\partial a^{\ell-1}} = \left(W^\ell\right)^T \delta^L$$

Chain Rule:

$$\frac{\partial C}{\partial z_i^{\ell-1}} = \frac{\partial C}{\partial a_i^{\ell-1}} \times \frac{d a_i^{\ell-1}}{d z_i^{\ell-1}}$$

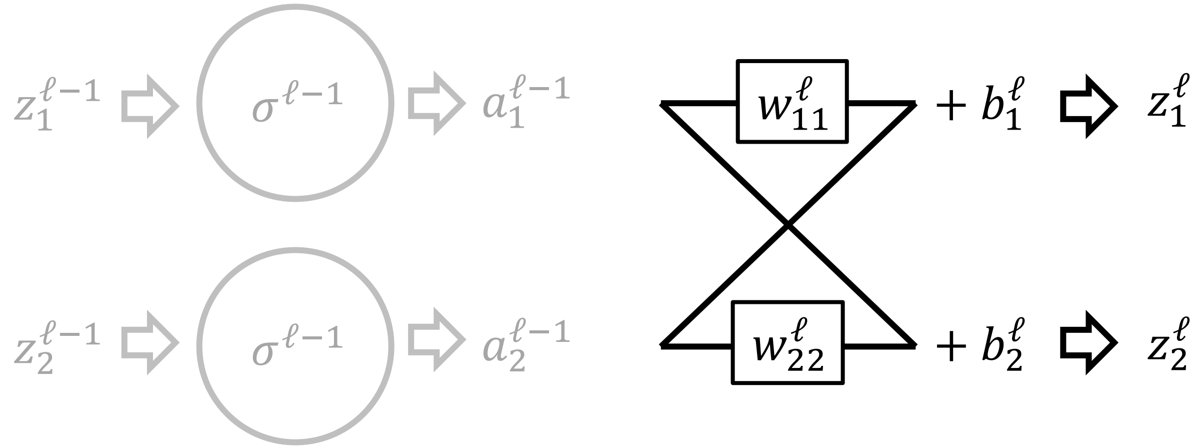
Substitute:

$$\Rightarrow \frac{\partial C}{\partial z_i^{\ell-1}} = \frac{\partial C}{\partial a_i^{\ell-1}} \sigma'(z_i^{\ell-1})$$

Vector Notation:

$$\Rightarrow \delta_i^{\ell-1} = \left(W^\ell\right)^T \delta^\ell \odot \sigma'(z^{\ell-1})$$

Propagating Error: Given δ^ℓ, Compute $\delta^{\ell-1}$



$$\frac{\partial z_i^\ell}{\partial a_j^{\ell-1}} = w_{ij}^\ell$$

$$\frac{da_i^{\ell-1}}{dz_i^{\ell-1}} = \sigma'(z_i^{\ell-1})$$

Chain Rule:

$$\frac{\partial C}{\partial a_i^{\ell-1}} = \sum_j \frac{\partial C}{\partial z_j^\ell} \times \frac{\partial z_j^\ell}{\partial a_i^{\ell-1}}$$

Substitute:

$$\Rightarrow \frac{\partial C}{\partial a_i^{\ell-1}} = \sum_j \delta_j^\ell w_{ji}^\ell$$

Vector Notation:

$$\Rightarrow \frac{\partial C}{\partial a^{\ell-1}} = \left(W^\ell\right)^T \delta^L$$

Chain Rule:

$$\frac{\partial C}{\partial z_i^{\ell-1}} = \frac{\partial C}{\partial a_i^{\ell-1}} \times \frac{da_i^{\ell-1}}{dz_i^{\ell-1}}$$

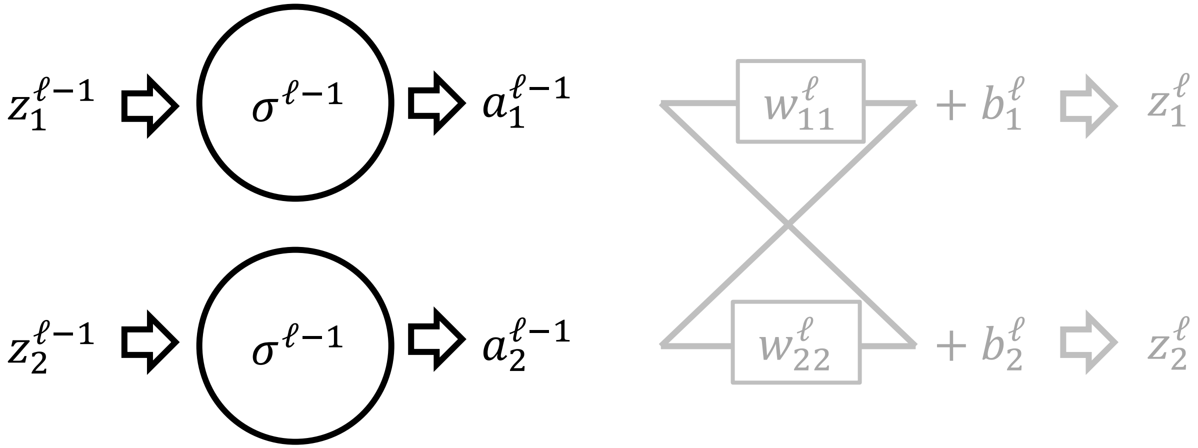
Substitute:

$$\Rightarrow \frac{\partial C}{\partial z_i^{\ell-1}} = \frac{\partial C}{\partial a_i^{\ell-1}} \sigma'(z_i^{\ell-1})$$

Vector Notation:

$$\Rightarrow \delta_i^{\ell-1} = \left(W^\ell\right)^T \delta^\ell \odot \sigma'(z^{\ell-1})$$

Propagating Error: Given δ^{ℓ}, Compute $\delta^{\ell-1}$



$$\frac{\partial z_i^{\ell}}{\partial a_j^{\ell-1}} = w_{ij}^{\ell}$$

$$\frac{d a_i^{\ell-1}}{d z_i^{\ell-1}} = \sigma'(z_i^{\ell-1})$$

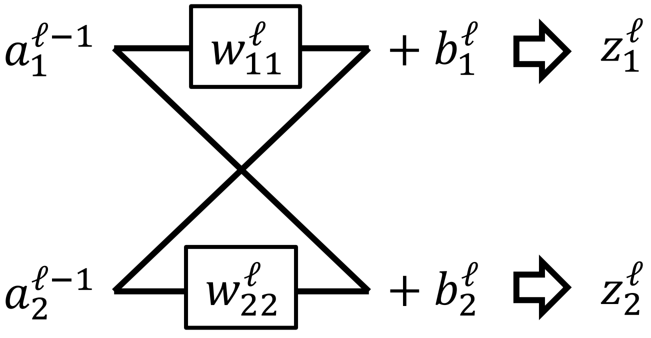
Chain Rule: Substitute: Vector Notation:

$$\frac{\partial C}{\partial a_i^{\ell-1}} = \sum_j \frac{\partial C}{\partial z_j^{\ell}} \times \frac{\partial z_j^{\ell}}{\partial a_i^{\ell-1}} \Rightarrow \frac{\partial C}{\partial a_i^{\ell-1}} = \sum_j \delta_j^{\ell} w_{ji}^{\ell} \Rightarrow \frac{\partial C}{\partial a^{\ell-1}} = \left(W^{\ell}\right)^T \delta^L$$

Chain Rule: Substitute: Vector Notation:

$$\frac{\partial C}{\partial z_i^{\ell-1}} = \frac{\partial C}{\partial a_i^{\ell-1}} \times \frac{d a_i^{\ell-1}}{d z_i^{\ell-1}} \Rightarrow \frac{\partial C}{\partial z_i^{\ell-1}} = \frac{\partial C}{\partial a_i^{\ell-1}} \sigma'(z_i^{\ell-1}) \Rightarrow \delta_i^{\ell-1} = \left(W^{\ell}\right)^T \delta^{\ell} \odot \sigma'(z^{\ell-1})$$

Gradients: Given δ^ℓ, Compute $\partial C/\partial b^\ell$ and $\partial C/\partial W^\ell$



Weighted Input:

$$z_i^\ell = \sum_j w_{ij}^\ell a_j^{\ell-1} + b_i^\ell$$

$\Rightarrow$

Derivatives:

$$\frac{\partial z_j^\ell}{\partial b_i^\ell} = Id_{ij}$$

$$\frac{\partial z_i^\ell}{\partial w_{ij}} = a_j^{\ell-1}$$

Apply Chain Rule:

$$\frac{\partial C}{\partial b_i^l} = \frac{\partial C}{\partial z_j^l} \times \frac{\partial z_j^\ell}{\partial b_i^\ell}$$

$$\frac{\partial C}{\partial w_{ij}^l} = \frac{\partial C}{\partial z_i^l} \times \frac{\partial z_i^\ell}{\partial w_{ij}}$$

$\Rightarrow$

Substitute for derivatives:

$$\frac{\partial C}{\partial b_i^l} = \delta_j^\ell \times Id_{ij} = \delta_i^\ell$$

$$\frac{\partial C}{\partial w_{ij}^l} = \delta_i^\ell \times a_j^{\ell-1}$$

$\Rightarrow$

Vector Notation:

$$\frac{\partial C}{\partial b^\ell} = \delta^\ell$$

$$\frac{\partial C}{\partial W^l} = \delta^\ell \left(a^{\ell-1}\right)^T$$

The Complete Backpropagation Algorithm

Feed Forward:

$$z^\ell = W^\ell a^{\ell-1} + b^\ell$$

$$a^\ell = \sigma^\ell(z^\ell)$$

Backpropagate Error:

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

$$\delta^{\ell-1} = \left(W^\ell\right)^T \delta^\ell \odot \sigma'(z^{\ell-1})$$

Calculate Gradients:

$$\frac{\partial C}{\partial b^\ell} = \delta^\ell$$

$$\frac{\partial C}{\partial W^l} = \delta^\ell \left(a^{\ell-1}\right)^T$$

Translating to MATLAB

Setting up the network

```matlab
% Initializing network by hand: size is [2,3,3,2];
W2 = 0.5*ones(3,2);
b2 = 0.5*ones(3,1);
W3 = 0.5*ones(3,3);
b3 = 0.5*ones(3,1);
W4 = 0.5*ones(2,3);
b4 = 0.5*ones(2,1);

% Cell arrays for each layer
W = {[],W2, W3, W4};
b = {[],b2, b3, b4};
sigma = {[],@ReLU, @ReLU, @sigmoid};
grad = {[],@grad_ReLU, @grad_ReLU, @grad_sigmoid};

% Training data here; one data point for this example
features  = {[2;2]};
labels = {[0.3;0.7]};

% Initalize input and activation arrays
z = {}; % Weighted input to each layer
a = {}; % Activation at each layer

% Max Index, (For readability)
L = length(W);
```

Backpropagation

```matlab
% Set input layer
a{1} = features{1}; %features(1) is a cell

% Forward pass: apply the network to the data
for i = 2:L % Layer 1 is the input layer
    z{i} = W{i} * a{i-1} + b{i};
    a{i} = sigma{i}(z{i});
end

% Calculate error for output layer
delta{L} = (labels{1} - a{L}) .* grad{L}(z{L});
grad_b{L} = delta{L};
grad_W{L} = delta{L} * a{L-1}' ;

% Backward Pass: the chain rule
for i = (L - 1):-1:2 % Looping backwards
    % Backpropagate error
    delta{i} = (W{i+1}'*delta{i+1}).*grad{i}(z{i});

    % Compute gradient for weights and bias
    grad_b{i} = delta{i};
    grad_W{i} = delta{i} * a{i-1}' ;
end
```

Stochastic Gradient Descent (SGD)

- Network update:

$$\theta_i \leftarrow \theta_i + \frac{\eta}{B} \nabla_\theta$$

where η is the learning rate and B is the batch size

- Repeat process until converged

Algorithm 2 Stochastic Gradient Descent (SGD)

procedure SGD TRAINING(X)
    initialize $\theta = (W, b)$ to small random numbers
    randomize order of training examples in $\mathbf{X}$
    while not converged **do**
        Create *minibatch* of length B from X
        $a_1 = minibatch_j$
        $\nabla_\theta =$Backpropagation(X, θ)
        for $i \leftarrow [2, L]$ **do**
            $\theta_i \leftarrow \theta_i + \frac{\eta}{B} \nabla_\theta$
        end for
    end while
end procedure

Adaptive Moment Estimation (Adam) Optimizer

- Modification of SGD
- Introduce *m* and *v*: the first and second moments of inertia of the gradient
  - First moment = running mean
  - Second moment = running uncentered variance
- Update Rule:

$$m_i \leftarrow \beta_1 m_i + (1 - \beta_1)\nabla_\theta$$
$$v_i \leftarrow \beta_2 v_i + (1 - \beta_2){\nabla_\theta}^2$$

$$\beta_1, \beta_2 \sim 1$$

These build momentum!

Algorithm 3 Adam

procedure ADAM(X)
  initialize $\theta = (W, b)$ to small random numbers
  randomize order of training examples in $\mathbf{X}$
  initialize $m \leftarrow 0,\ v \leftarrow 0$
  while not converged **do**
    Create *minibatch* of length B from X
    $a_1 = minibatch_j$
    $\nabla_\theta =$Backpropagation(X, θ)
    for $i \leftarrow [2, L]$ **do**
      $m_i \leftarrow \beta_1 m_i + (1 - \beta_1)\nabla_\theta$
      $v_i \leftarrow \beta_2 v_i + (1 - \beta_2)\nabla_\theta^2$
      $\hat{m}_i = \frac{m_i}{1-\beta_1^t}$
      $\hat{v}_i = \frac{v_i}{1-\beta_2^t}$
      $\theta_i \leftarrow \theta_i - \frac{\alpha}{\sqrt{\hat{v}_i}+\epsilon}\hat{m}_i$
    end for
  end while
end procedure

Adam Optimizer

- Update network parameters:

$$\theta_i \leftarrow \theta_i - \frac{\alpha}{\sqrt{\hat{v}_i} + \varepsilon}\hat{m}_i$$

- To avoid biasing the first few parameter updates, a correction step must be completed.

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^t}$$

$$\hat{v}_i = \frac{v_i}{1 - \beta_2^t}$$

Algorithm 3 Adam

procedure ADAM(X)
  initialize $\theta = (W, b)$ to small random numbers
  randomize order of training examples in $\mathbf{X}$
  initialize $m \leftarrow 0$, $v \leftarrow 0$
  while not converged **do**
    Create *minibatch* of length B from X
    $a_1 = minibatch_j$
    ∇_θ =Backpropagation(X, θ)
    for $i \leftarrow [2, L]$ **do**
      $m_i \leftarrow \beta_1 m_i + (1 - \beta_1)\nabla_\theta$
      $v_i \leftarrow \beta_2 v_i + (1 - \beta_2)\nabla_\theta^2$
      $\hat{m}_i = \frac{m_i}{1-\beta_1^t}$
      $\hat{v}_i = \frac{v_i}{1-\beta_2^t}$
      $\theta_i \leftarrow \theta_i - \frac{\alpha}{\sqrt{\hat{v}_i + \epsilon}}\hat{m}_i$
    end for
  end while
end procedure

Adam Optimizer

Why is the bias correction required? Consider the equation for the second moment.

$$v_i \leftarrow \beta_2 v_i + (1 - \beta_2){\nabla_\theta}^2$$

At some time step t, this moving average can be written as a sum of previous second moments.

$$v_t = (1 - \beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \cdot {\nabla_\theta}^2$$

How does this compare against the true second moment, $\mathbb{E}\left[{\nabla_\theta}^2\right]$?

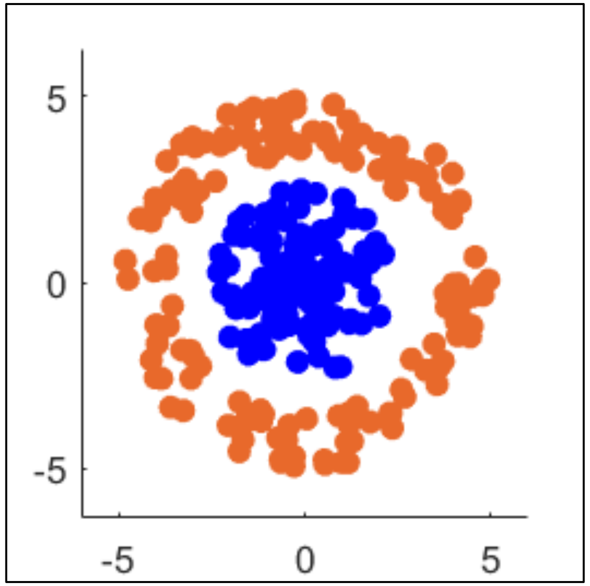
Adam Optimizer

$$
\begin{aligned}
\mathbb{E}[v_t] &= \mathbb{E}\left[(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\cdot{\nabla_\theta}^2\right] \\
&= \mathbb{E}\left[{\nabla_\theta}^2\right]\cdot(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i} \\
&= \mathbb{E}\left[{\nabla_\theta}^2\right]\cdot(1-\beta_2^t)
\end{aligned}
$$

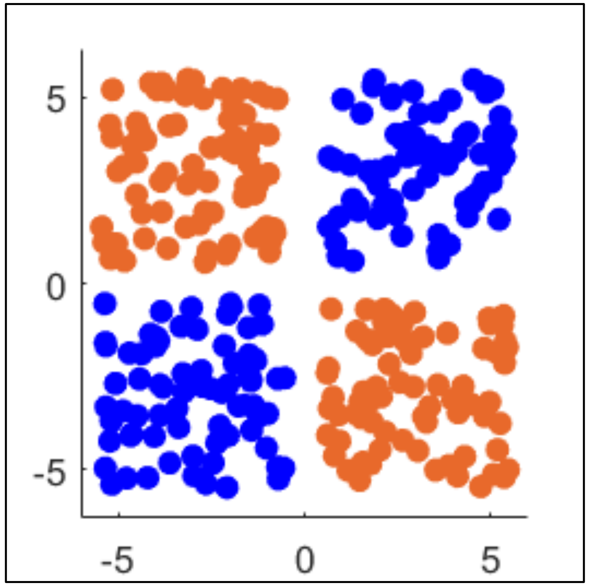
Therefore, to find the true value $\mathbb{E}\left[{\nabla_\theta}^2\right]$,

$$
\mathbb{E}\left[{\nabla_\theta}^2\right] = \hat{v}_i = \frac{v_i}{1-\beta_2^t}
$$

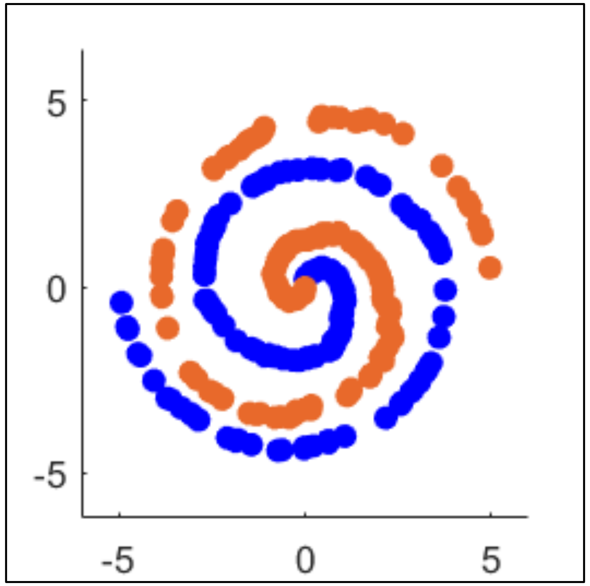
Tensorflow Playground



Circles

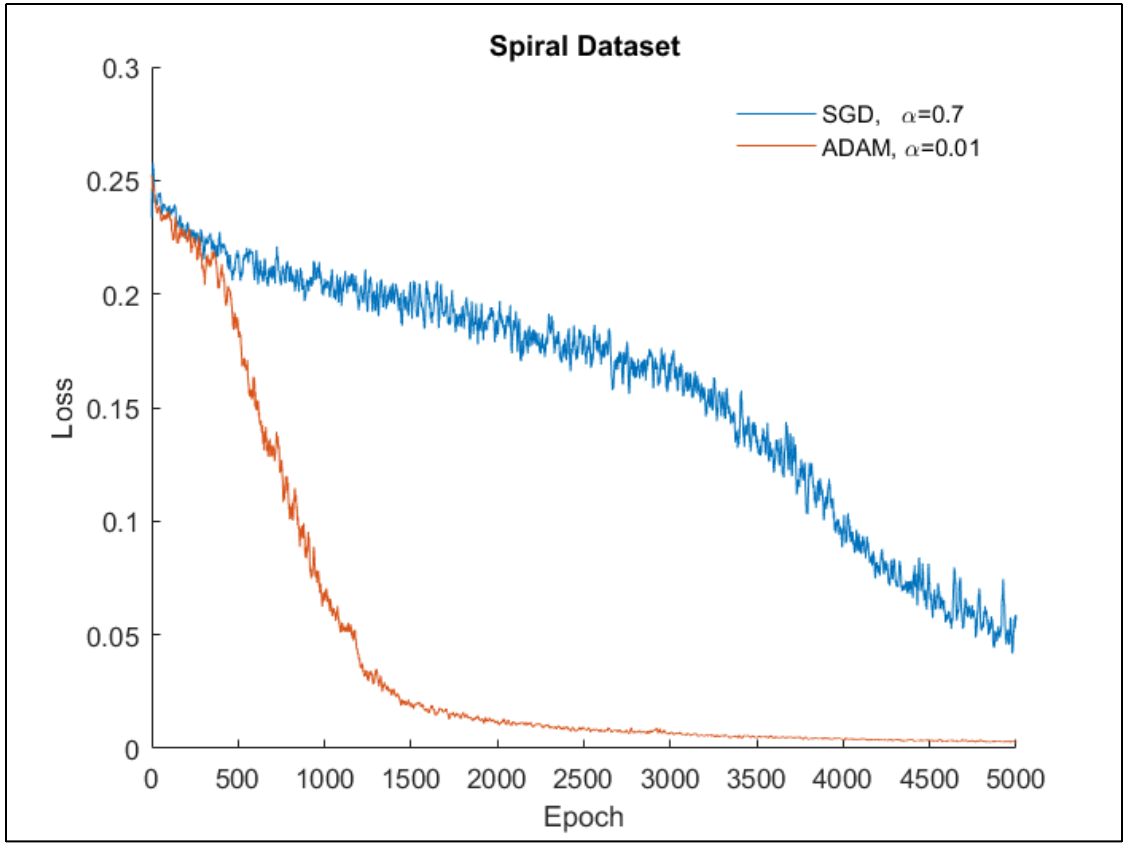


Square Regions

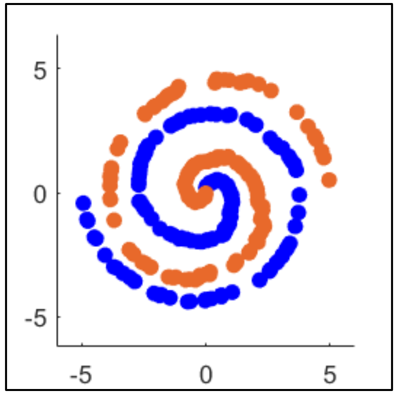


Spirals

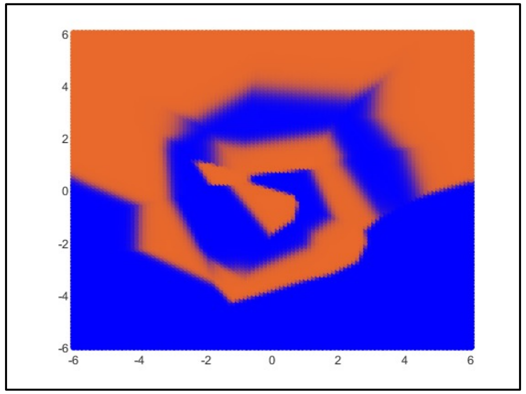
Tensorflow Playground Results



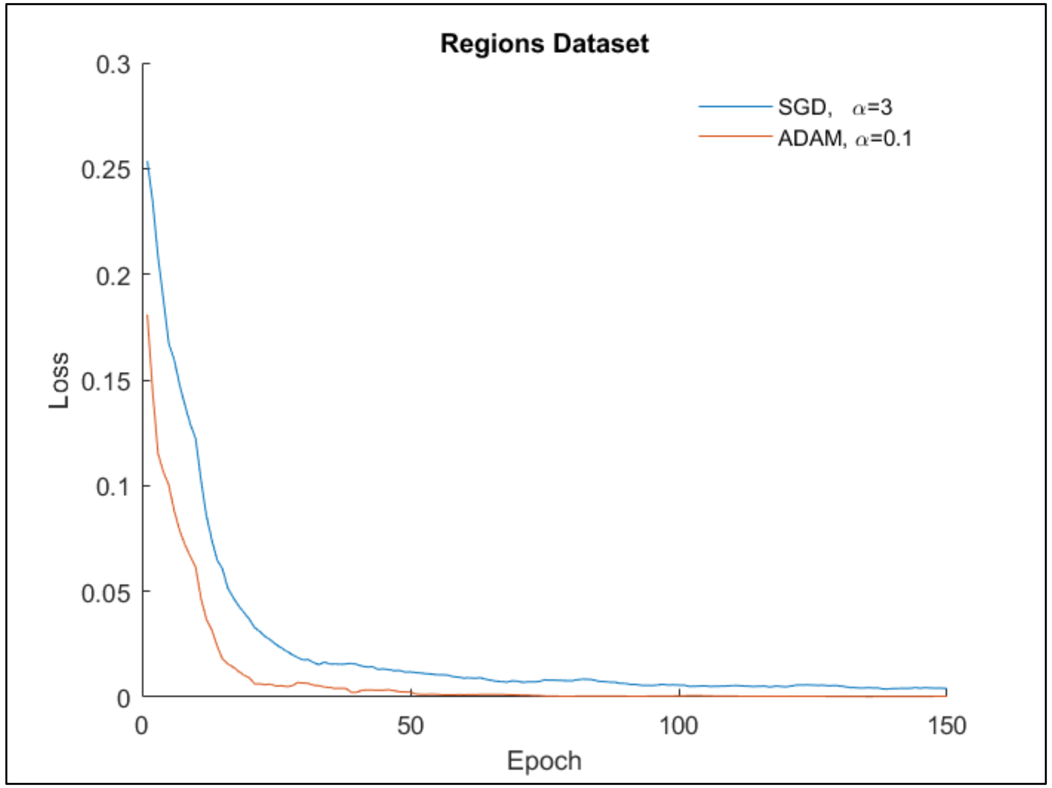
← SGD vs Adam Spiral data set (moving avg.)

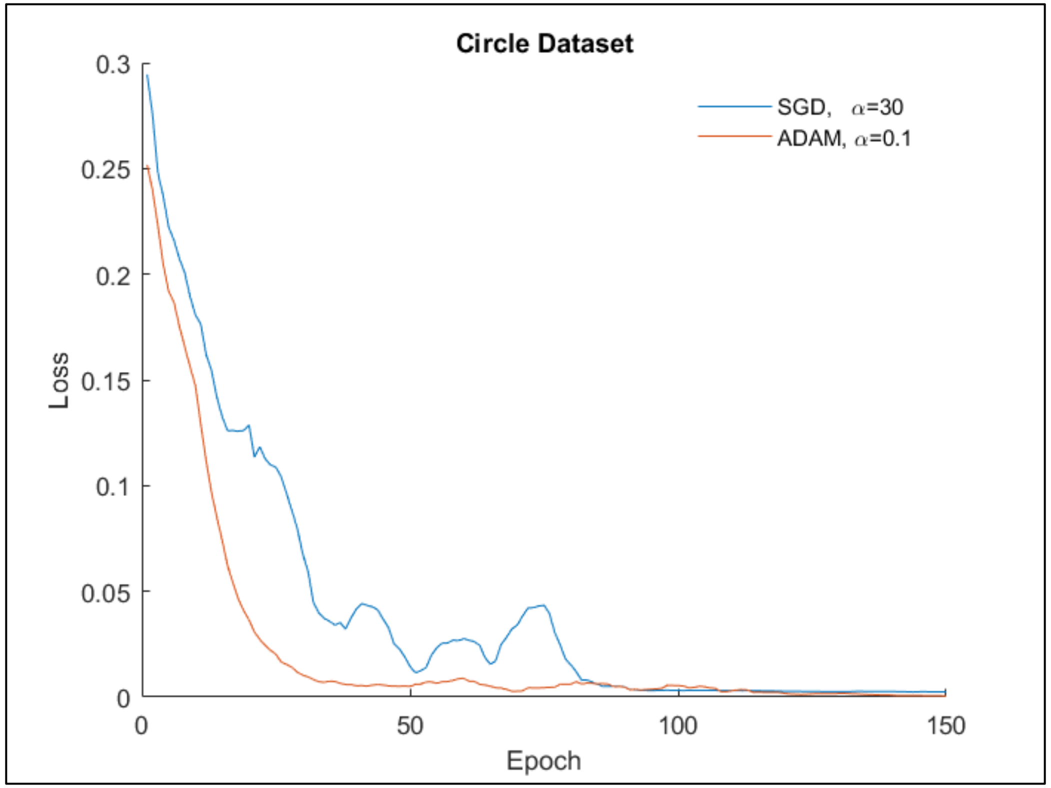


Adam Classification of Spiral data set →



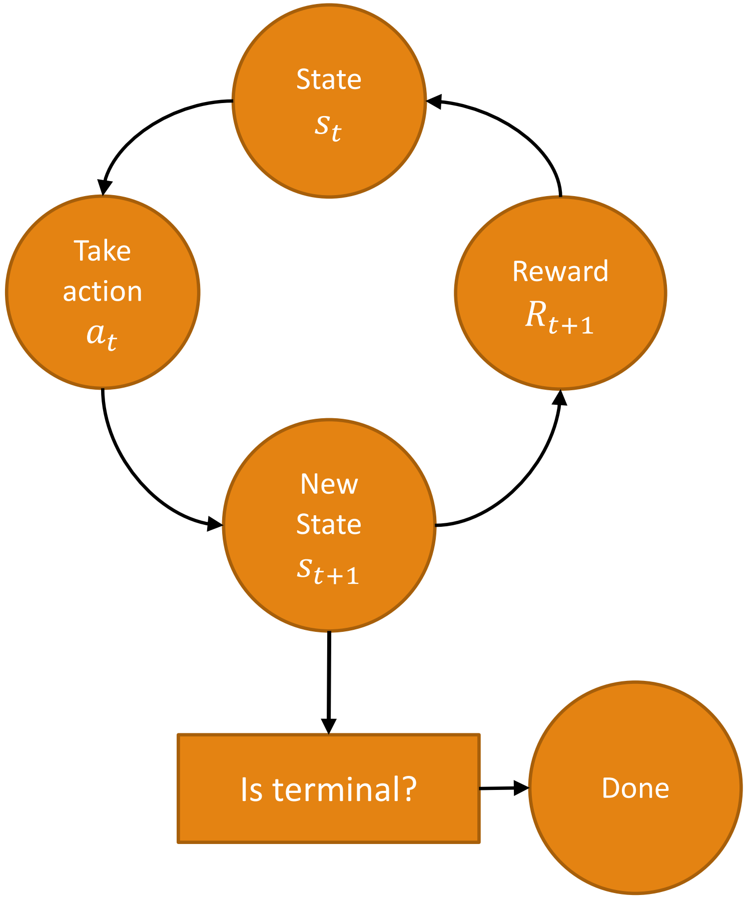
Tensorflow Playground Results





Reinforcement Learning (In 5 Minutes)

- Training an agent (the AI) to learn by giving positive or negative feedback
- Modeling a decision process:
  - Begin in a **state $\boldsymbol{s_t}$** at time t
  - Choose an **action $\boldsymbol{a_t}$** from a list of possible actions
  - Transitition to a **new state $\boldsymbol{s_{t+1}}$**
  - Receive appropriate **reward, $\boldsymbol{R_{t+1}}$**
  - Repeat until entering a terminal state – success or failure
- Goal: Choose actions (policy) to maximize total rewards



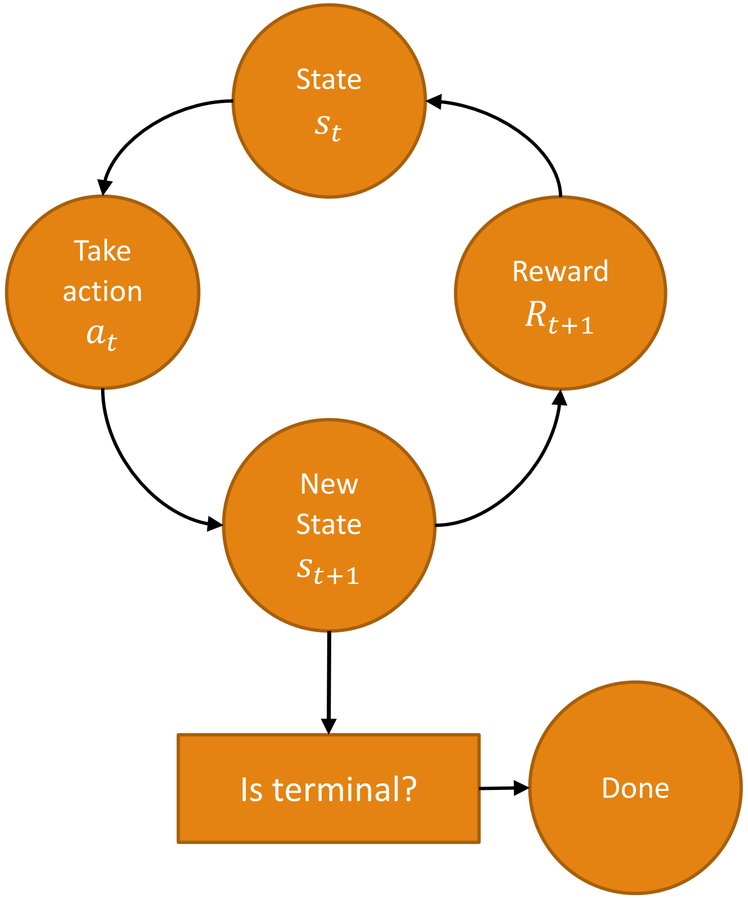
Expected Rewards

- What is total reward?
- Discount factor: $0 < \gamma < 1$
- Total discounted rewards:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots$$

The “quality” of an action:

$$Q(a, s) = \mathbb{E}(G_t \mid a_t = a, s_t = s)$$



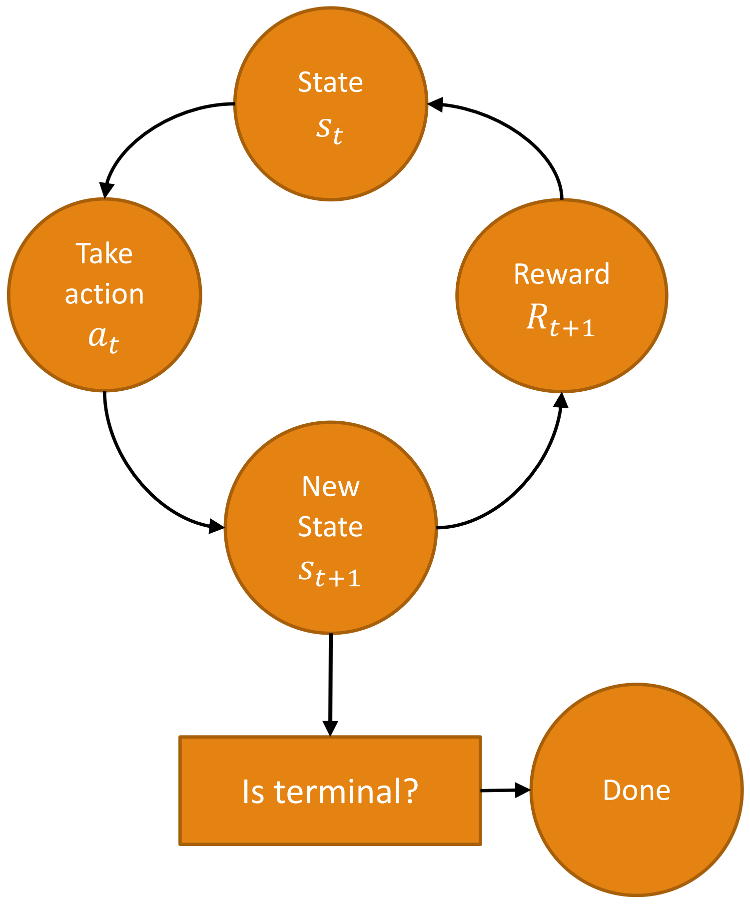
Results on MDP’s

(Markov Decision Processes)

1. An optimal policy is to always choose the highest quality action

2. When following an optimal policy, Q obeys the Bellman Equation:

$$Q(s_t, a_t) = R_{t+1} + \gamma \max_a Q(s_{t+1}, a)$$



Reinforcement Learning

- DQN approximates the quality function using a neural network.
- The network is trained with a cost function based on the Bellman Equation:

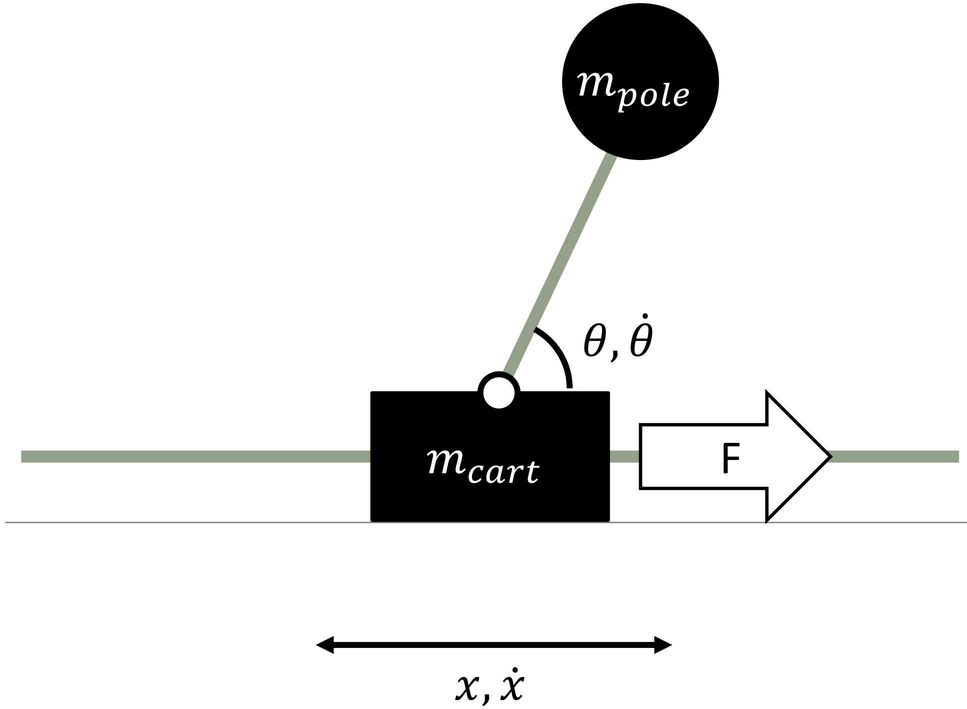
$$C = \left[Q^*(s_t, a_t) - R_{t+1} - \gamma \max_a Q^*(s_{t+1}, a) \right]^2$$

- Every action becomes training data for the network (improves with time)

---

procedure DQN(X)
    Initialize Neural Network to represent action-value function Q_θ
    Initialize Experience Replay memory, M
    Initialize target action-value function $\hat{Q}_{\theta^-}$
    for ep = [1,N] **do**
        $s_t \leftarrow s_0$
        while s_t is not a terminal state **do**
            With probability ϵ select a random action a_t
            Otherwise, select $a_t = argmax_a Q(s_t, a)$
            Receive reward R_{t+1} and new state s_{t+1}
            Store transition $\{s_t, a_t, s_{t+1}, R_{t+1}\}$ in M
            Train Q_θ using random experiences from M
            $s_t \leftarrow s_{t+1}$
        end while
        $\hat{Q}_{\theta^-} \leftarrow Q_\theta$
    end for
end procedure

Sample Problem: Cart Pole



State $s_t = \{x, \dot{x}, \theta, \dot{\theta}\}$

Terminal states:

$|x| > x_{max}$ or

$|\theta| > \theta_{max}$

Actions:

$a_1 \rightarrow F = 10\ N$

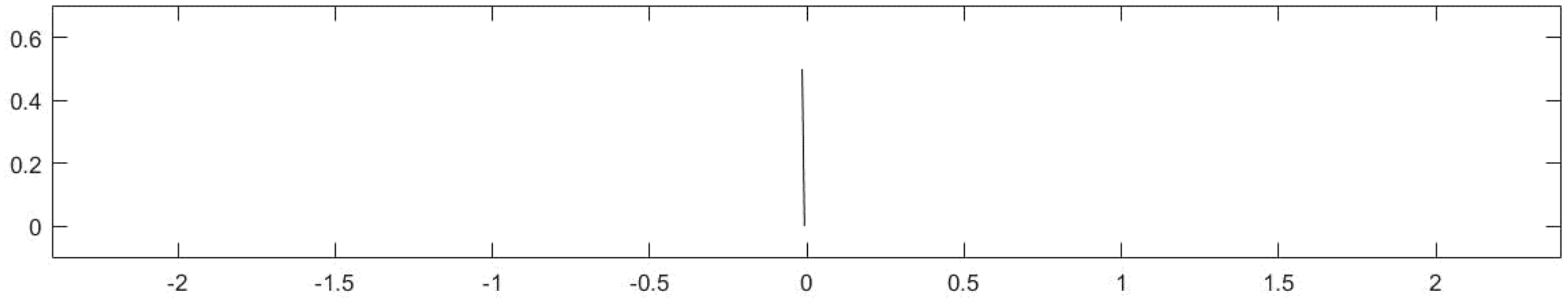
$a_2 \rightarrow F = -10\ N$

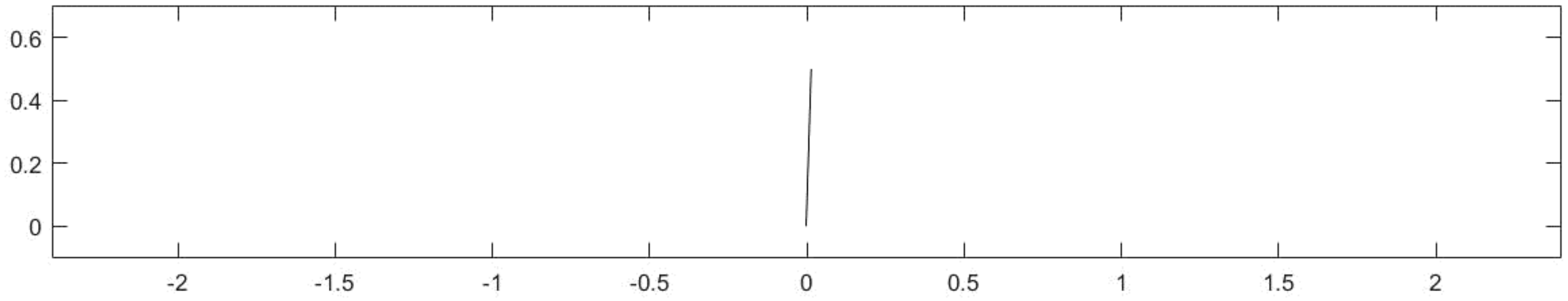
Reward:

$R_t = 0$ for a terminal state

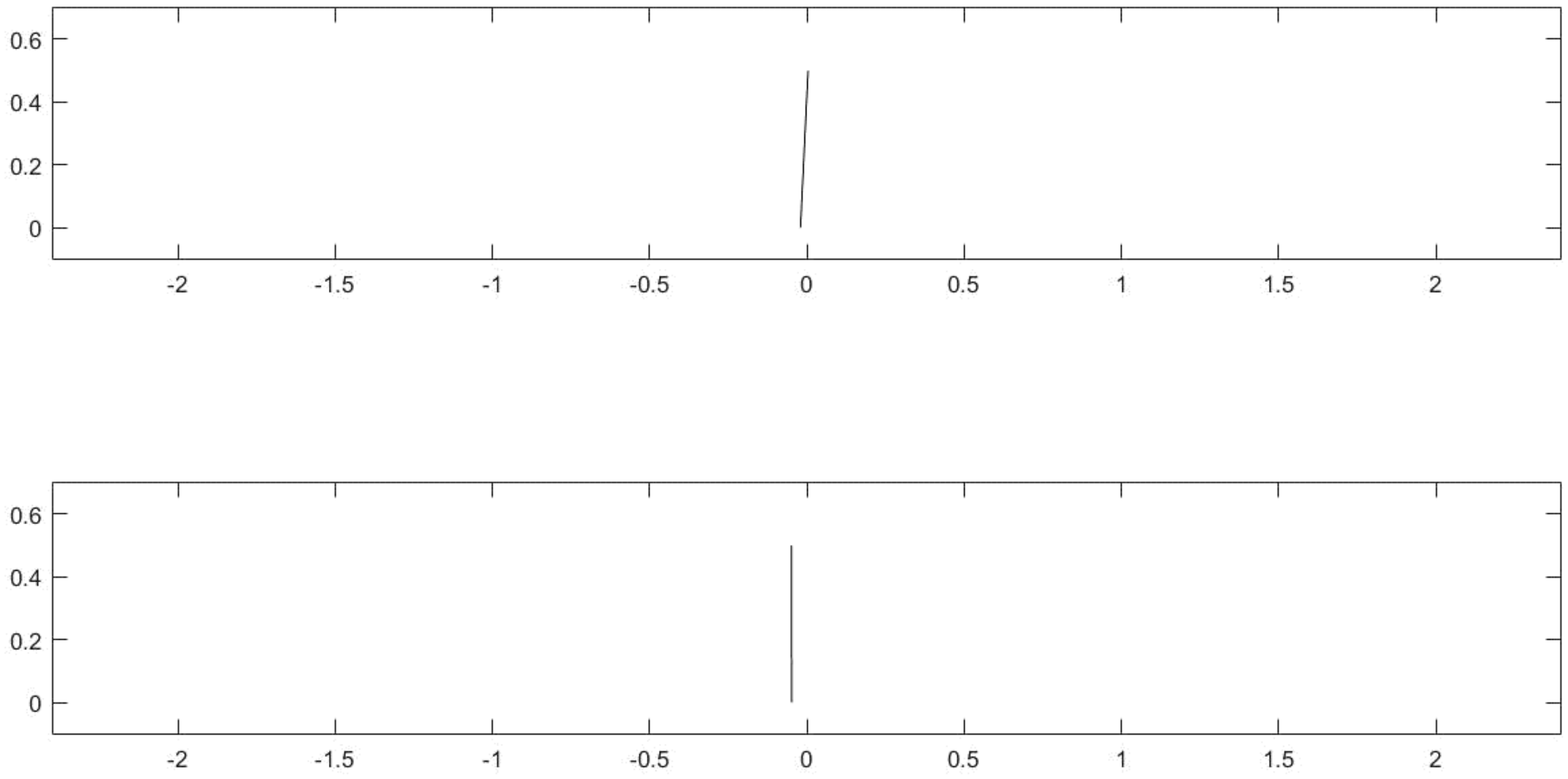
$R_t = 1$ for all other states

Results: Getting Started





Results: Mastery



Thank you!
Any questions?

Kingma, Diederik P, and Jimmy Ba. “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION.” *CoRR*, abs/1412.6980, 2014, arxiv.org/abs/1412.6980.

Mnih, Volodymyr, et al. “Human-Level Control through Deep Reinforcement Learning.” Nature, vol. 518, 26 Feb. 2015, pp. 529–533., doi:https://doi.org/10.1038/nature14236.

Nielson, Michael A. *Neural Networks and Deep Learning*. Determination Press, 2015, neuralnetworksanddeeplearning.com/chap2.html.

Ruder, Sebastian. “An Overview of Gradient Descent Optimization Algorithms.” Sebastian Ruder, Sebastian Ruder, 19 Jan. 2016, ruder.io/optimizing-gradient-descent.

Smilkov, Daniel, and Shan Carter. “Tensorflow - Neural Network Playground.” *A Neural Network Playground*, playground.tensorflow.org/.

Increasing Stability: Target Network

- Q-value: The quality of a state – the discounted reward expected from an action at a given state
- Bellman Equation:

$$Q^*(s_t, a_t) = R_{t+1} + \gamma \max_a Q^*(s_{t+1}, a)$$

- Framed as a cost function:

$$C = Q^*(s_t, a_t) - R_{t+1} + \gamma \max_a Q^*(s_{t+1}, a)$$

Working Network
Optimized every step

Target Network
Copied every ~500 steps

Increasing Stability: Experience Recall

Unstable: Current action immediately becomes training data

Improved: Current action is stored in memory, “recalled” later