Q Networks

Learning Objectives

- TD-Gammon
- Deep Q Networks
  - The Loss Function
  - Memory
  - Code

Agenda

TD-Gammon

Deep Q Networks - Loss

Deep Q Networks - Memory

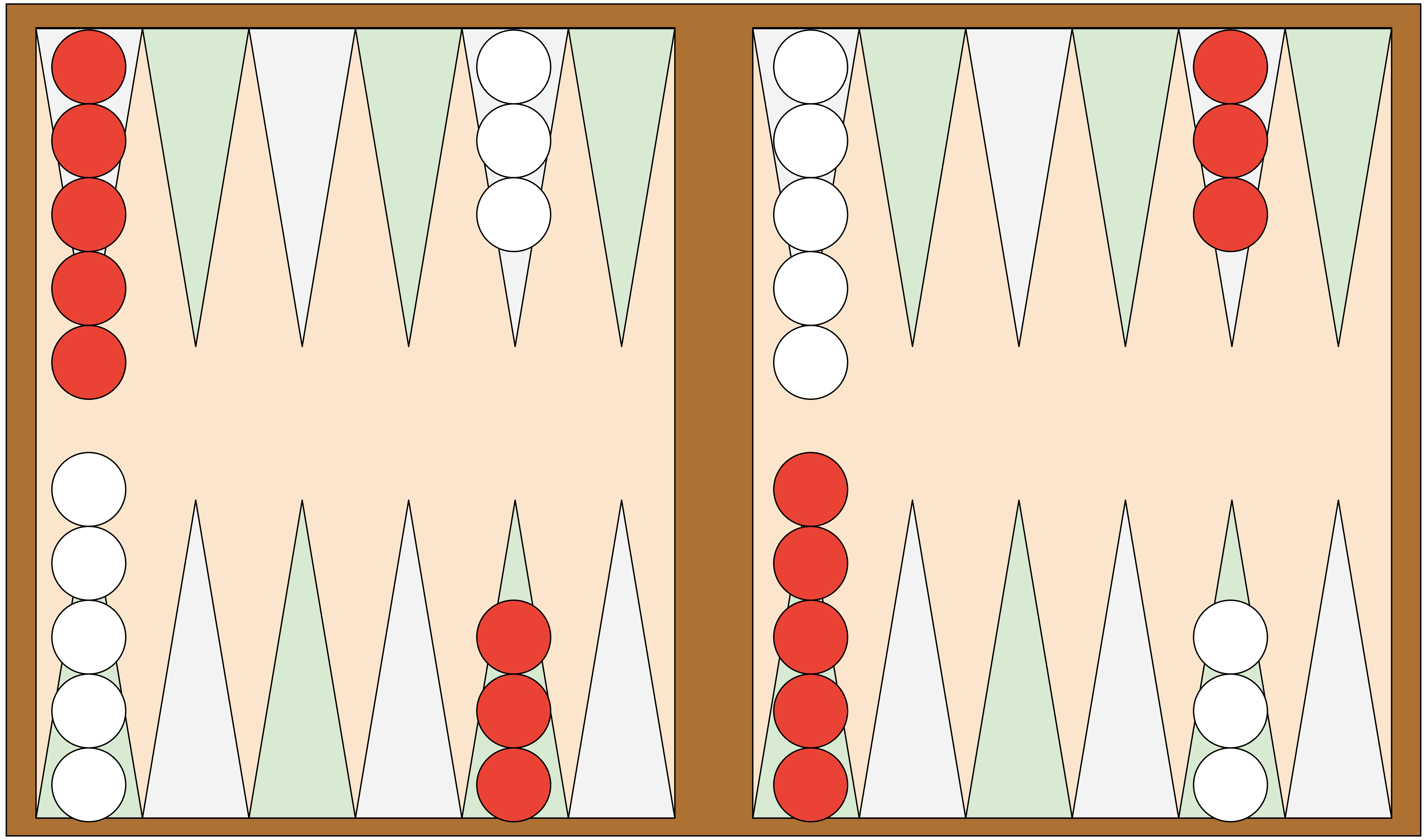
Deep Q Networks - Code

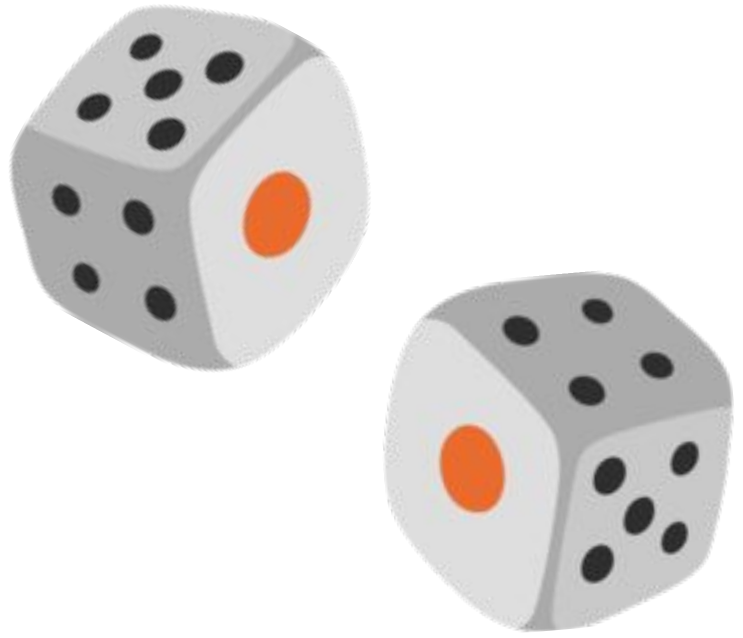
From Chess to Backgammon

From Chess to Backgammon

Google Cloud

From Chess to Backgammon





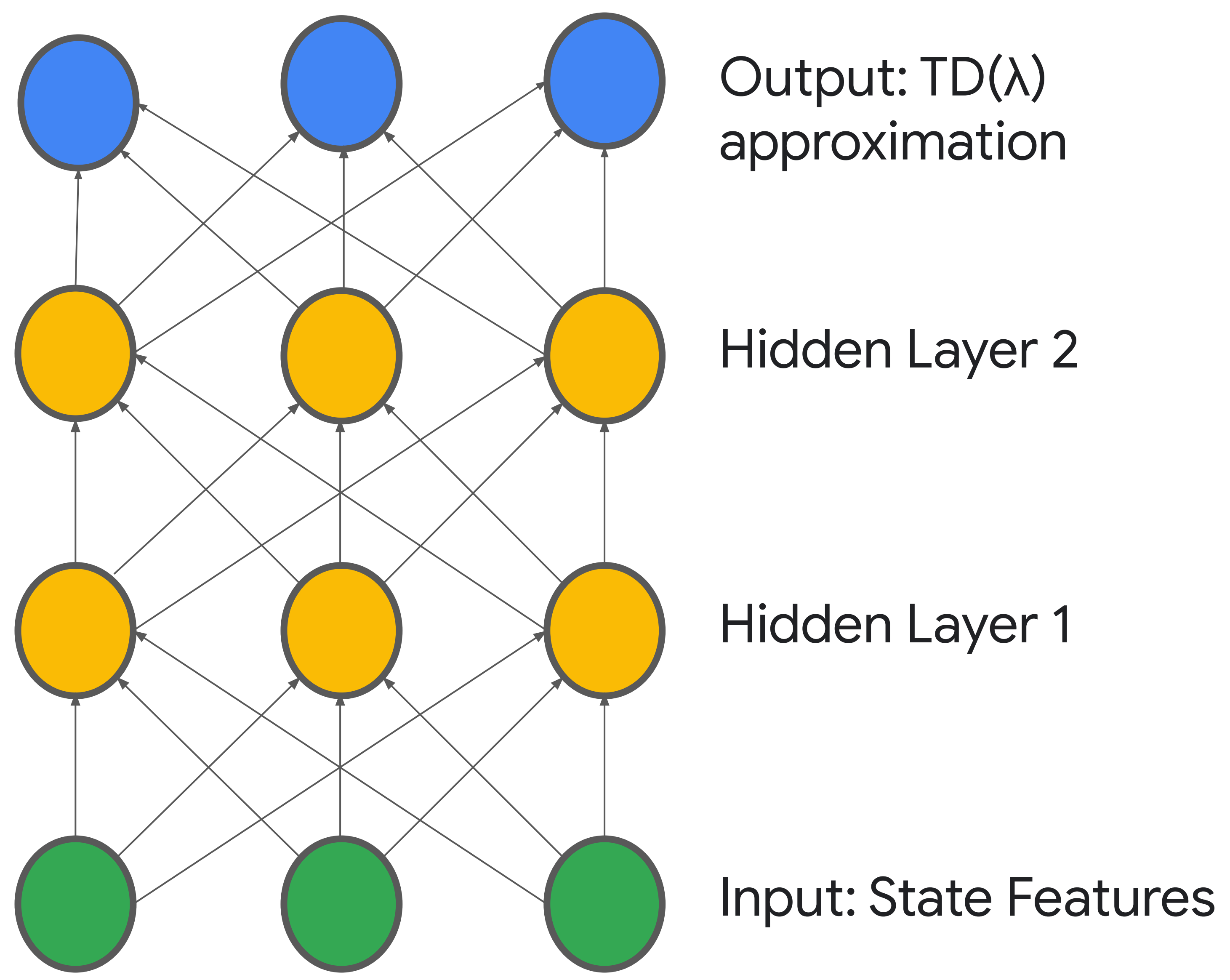
Google Cloud

Q Tables vs Q Networks

Q - table				
	Left	Down	Right	Up
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Q Tables vs Q Networks

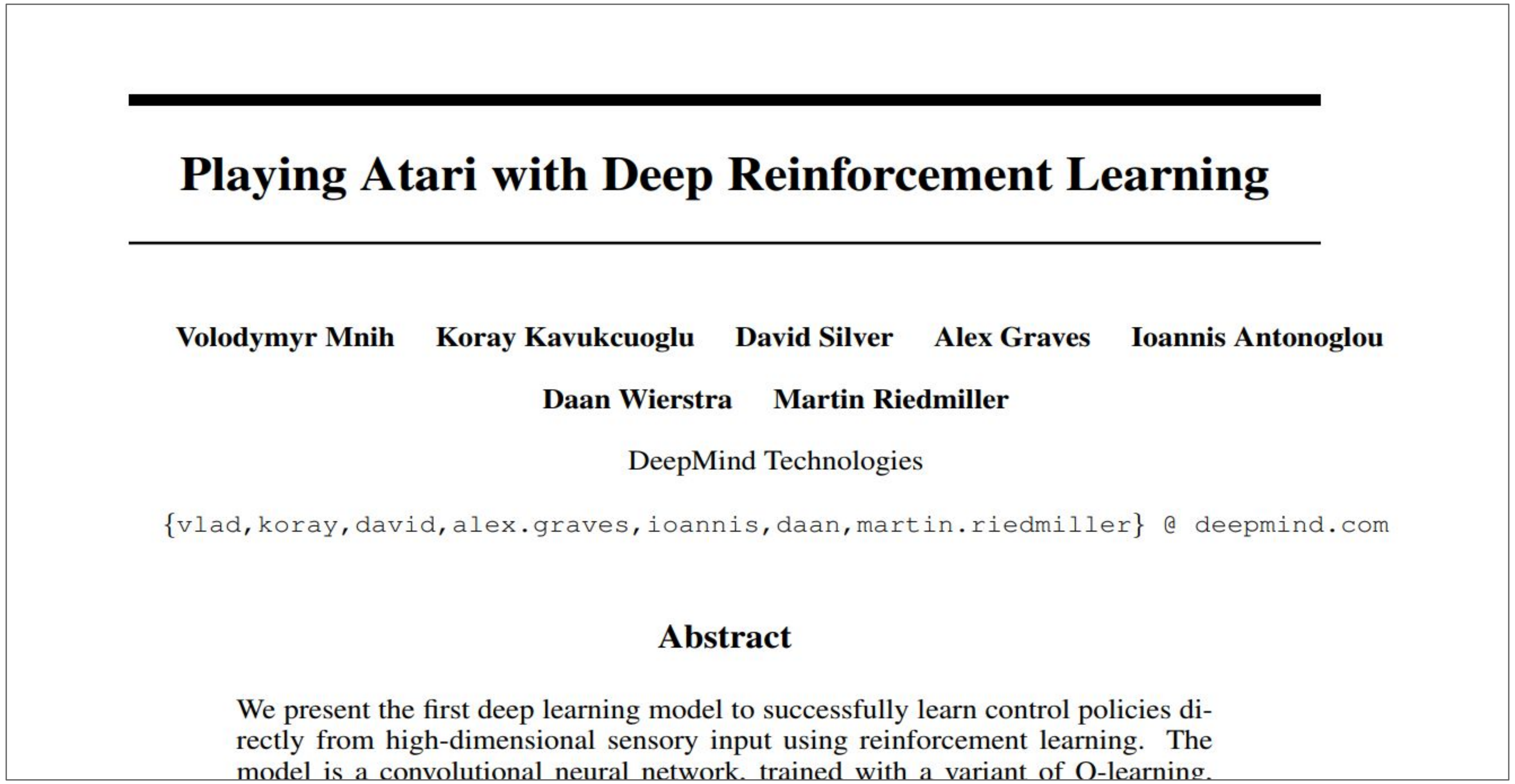
Q - table				
	Left	Down	Right	Up
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0



Agenda

Google Cloud

Deep Q Learning



Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou

Daan Wierstra Martin Riedmiller

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning,

[Deep Reinforcement Learning]

Deep Q Learning - Loss Function

$$Q(s_t, a_t) = Q(s_t, a_t) + \square_t\left(r_t + \gamma \cdot max_a\{Q(s_{t+1}, a)\} - Q(s_t, a_t)\right)$$

Deep Q Learning - Loss Function

$$Q(s_t, a_t) = Q(s_t, a_t) + \square_t \left(r_t + \gamma \cdot max_a \{Q(s_{t+1}, a)\} - Q(s_t, a_t) \right)$$

$$\Delta w = \square \left(r + \gamma \cdot max_a \{Q(s_{t+1}, a, w)\} - Q(s_t, a_t, w) \right) \nabla_w Q(s, a, w)$$

Deep Q Learning - Loss Function

$$Q(s_t,a_t) = Q(s_t,a_t) + \square_t\left(r_t + \gamma \cdot max_a\{Q(s_{t+1},a)\} \underline{- Q(s_t,a_t}\,)\right)$$

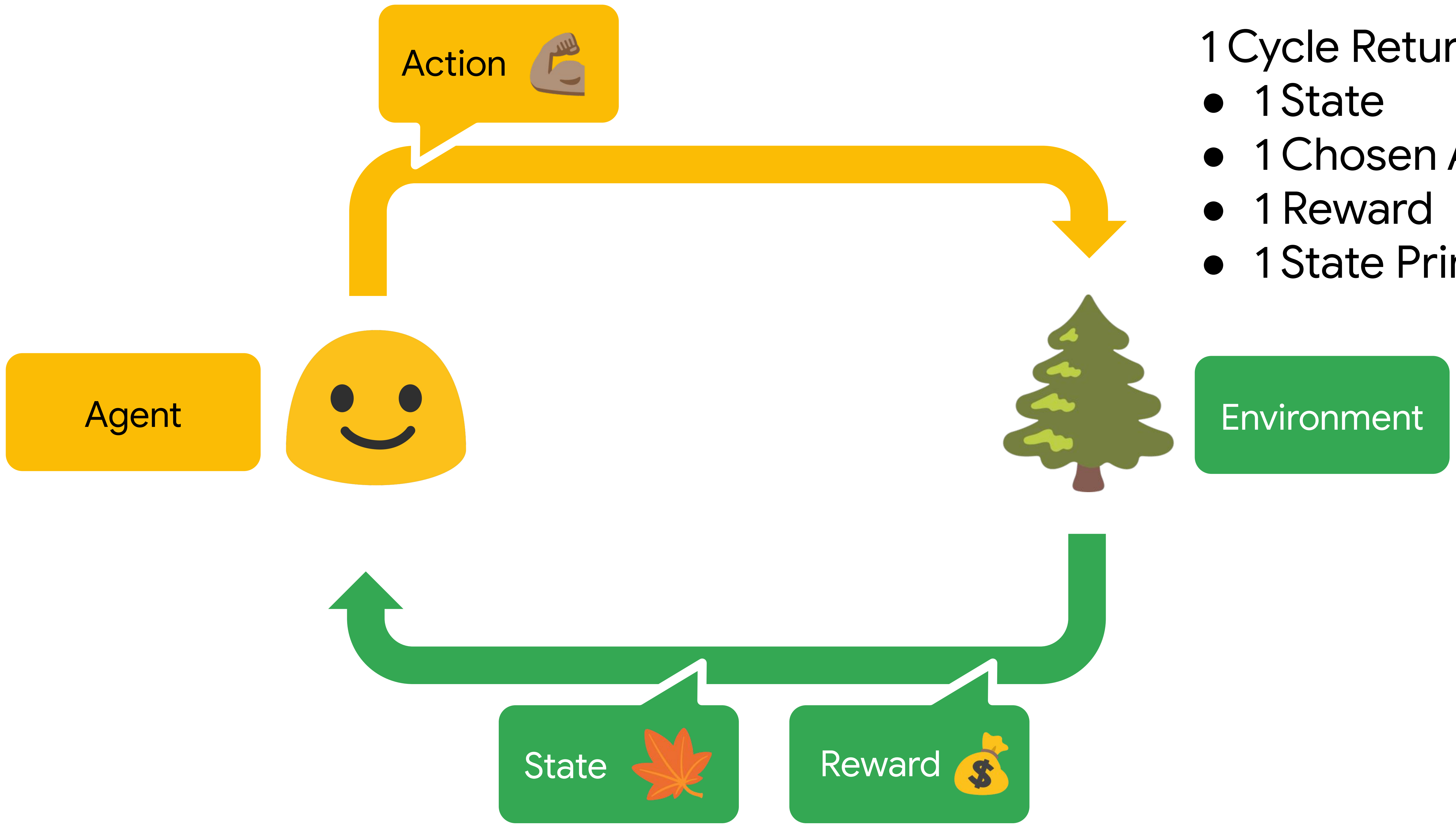
$$\Delta w = \square\left(r + \gamma \cdot max_a\{Q(s_{\underline{t+1},a,w)\}} - Q(s_t,a_t\,,w)\right)\nabla_w Q(s,a,w)$$

Deep Q Learning - Loss Function

$$Q(s_t,a_t) = Q(s_t,a_t) + \underline{\square_t\big(r_t + \gamma \cdot max_a\{Q(s_{t+1},a)\}} - Q(s_t,a_t)\big)$$

$$\Delta \underline{w = \square\big(r + \gamma \cdot max_a}\{Q(s_{t+1},a,w)\} - Q(s_t,a_t,w)\big)\nabla_w Q(s,a,w)$$

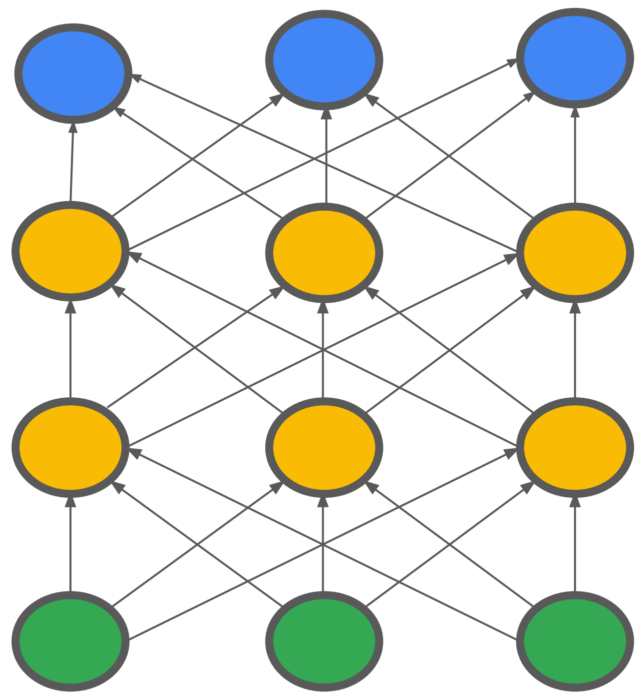
Deep Q Learning - Training



1 Cycle Returns:

- 1 State
- 1 Chosen Action
- 1 Reward
- 1 State Prime

Deep Q Learning - Training

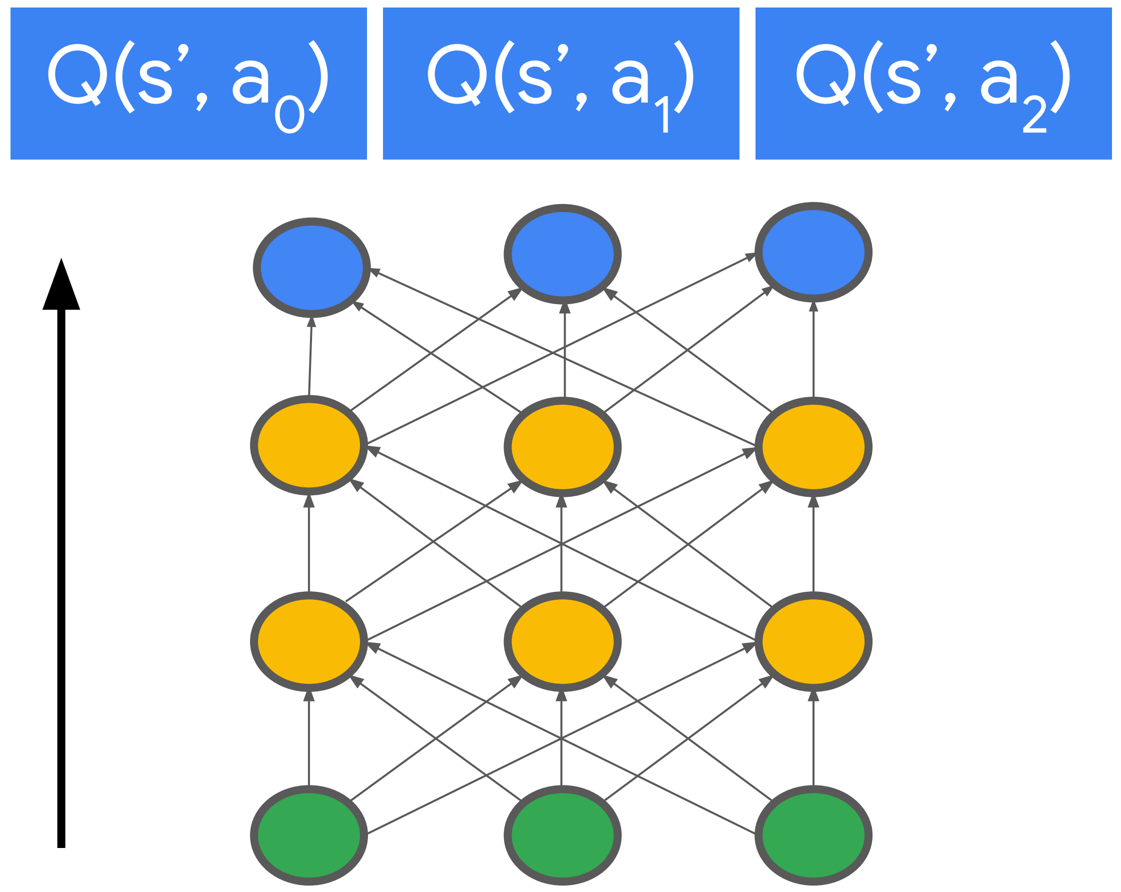


Feed in State Prime

1 Cycle Returns:

- 1 State
- 1 Chosen Action
- 1 Reward
- 1 State Prime

Deep Q Learning - Training

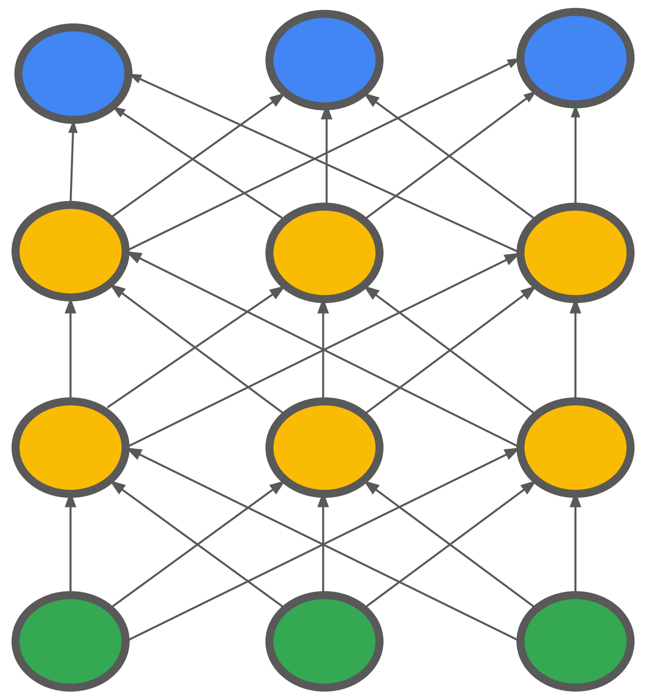


1 Cycle Returns:

- 1 State
- 1 Chosen Action
- 1 Reward
- ~~1 State Prime~~

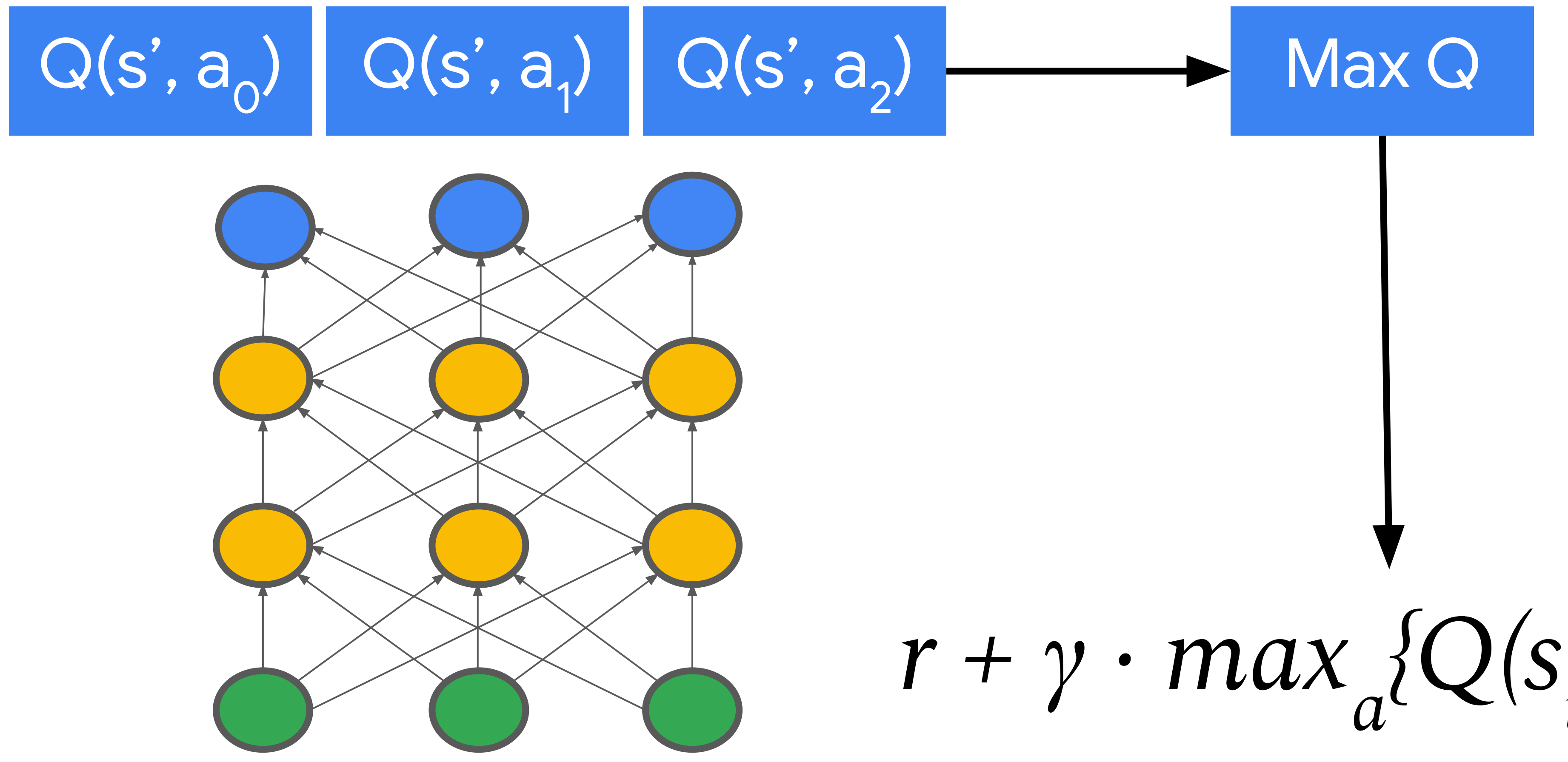
Deep Q Learning - Training

$Q(s', a_0)$ | $Q(s', a_1)$ | $Q(s', a_2)$ | Take Max Value



1 Cycle Returns:
- 1 State
- 1 Chosen Action
- 1 Reward
- ~~1 State Prime~~

Deep Q Learning - Training



$$r + \gamma \cdot max_a\{Q(s_{t+1},a,w)\}$$

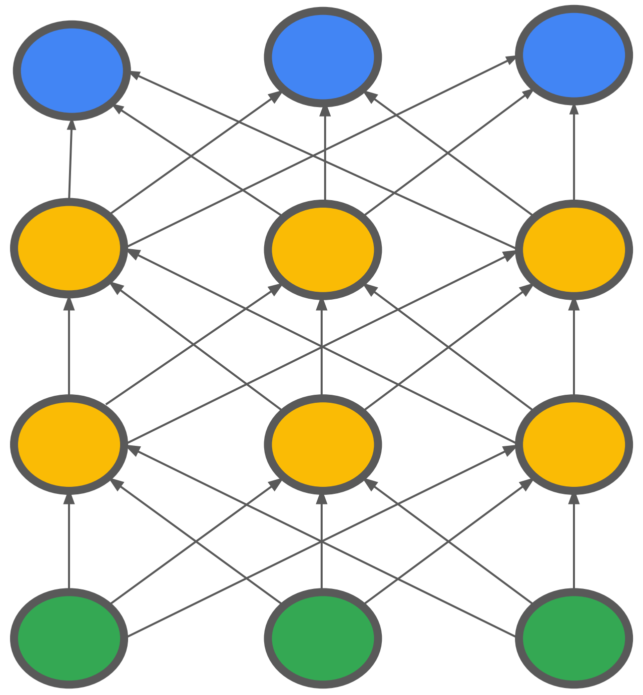
1 Cycle Returns:
- 1 State
- 1 Chosen Action
- 1 Reward
- ~~1 State Prime~~

Calculate Label

Deep Q Learning - Training

0	0	Label

Apply label to action

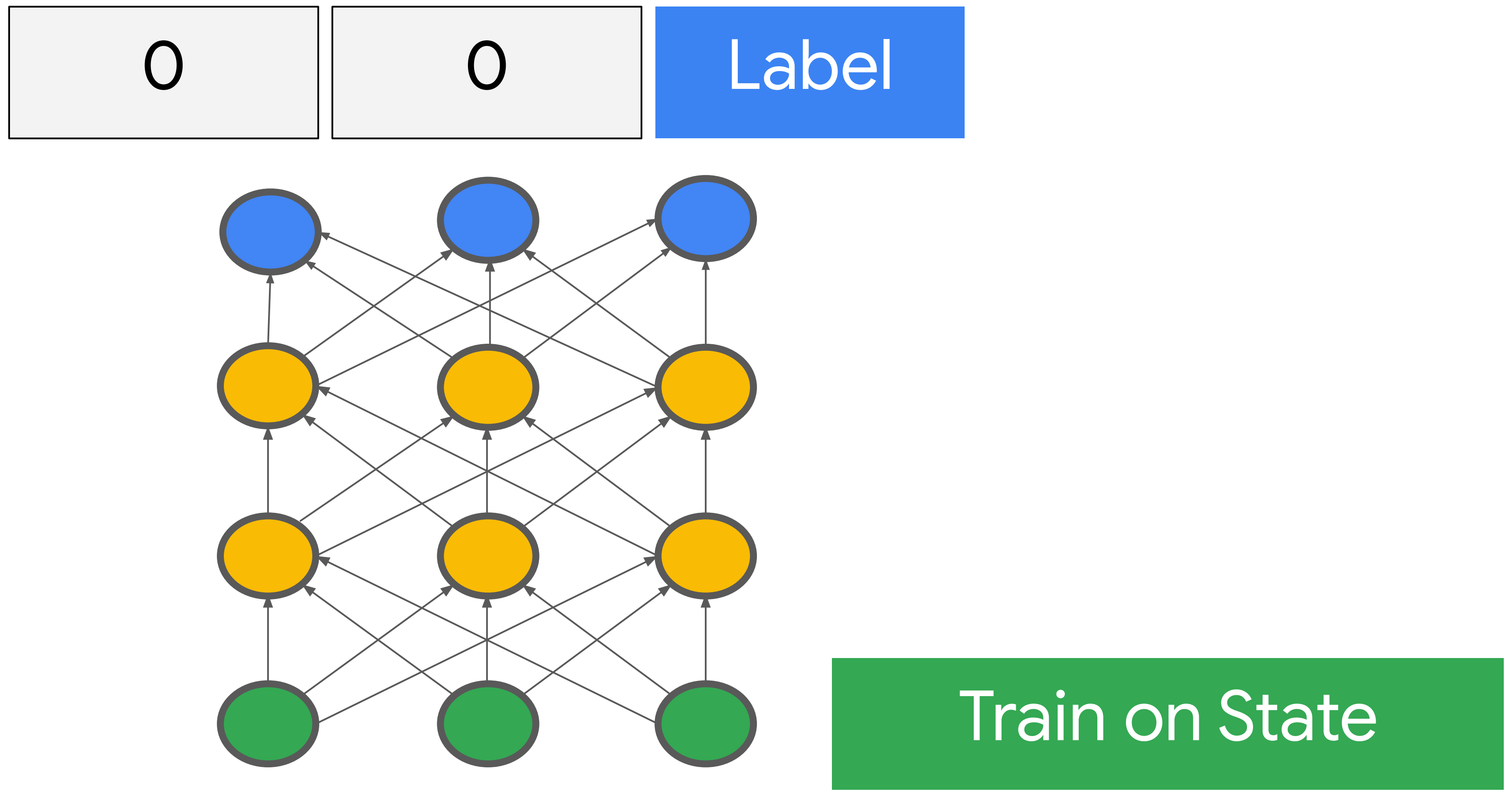


1 Cycle Returns:

- 1 State
- 1 Chosen Action
- ~~1 Reward~~
- ~~1 State Prime~~

Google Cloud

Deep Q Learning - Training



1 Cycle Returns:

- 1 State
- ~~1 Chosen Action~~
- ~~1 Reward~~
- ~~1 State Prime~~

Deep Q Learning - Loss Function

$$\Delta w = \square\left(r + \gamma \cdot max_a\{Q(s_{t+1},a,w)\} - Q(s_t,a_t\ ,w)\right)\nabla_w Q(s,a,w)$$

Label

Predicted Value

Deep Q Learning - Loss Function

$$\Delta w = \square\underline{\left(r + \gamma \cdot max_a\{Q(s_{t+1}}, \underline{a,w)\} - Q(s_t},a_t ,w)\right)\nabla_w Q(s,a,w)$$

Label

Predicted Value

$$Loss = (y - \hat{y})^2$$

Google Cloud

Agenda

TD-Gammon

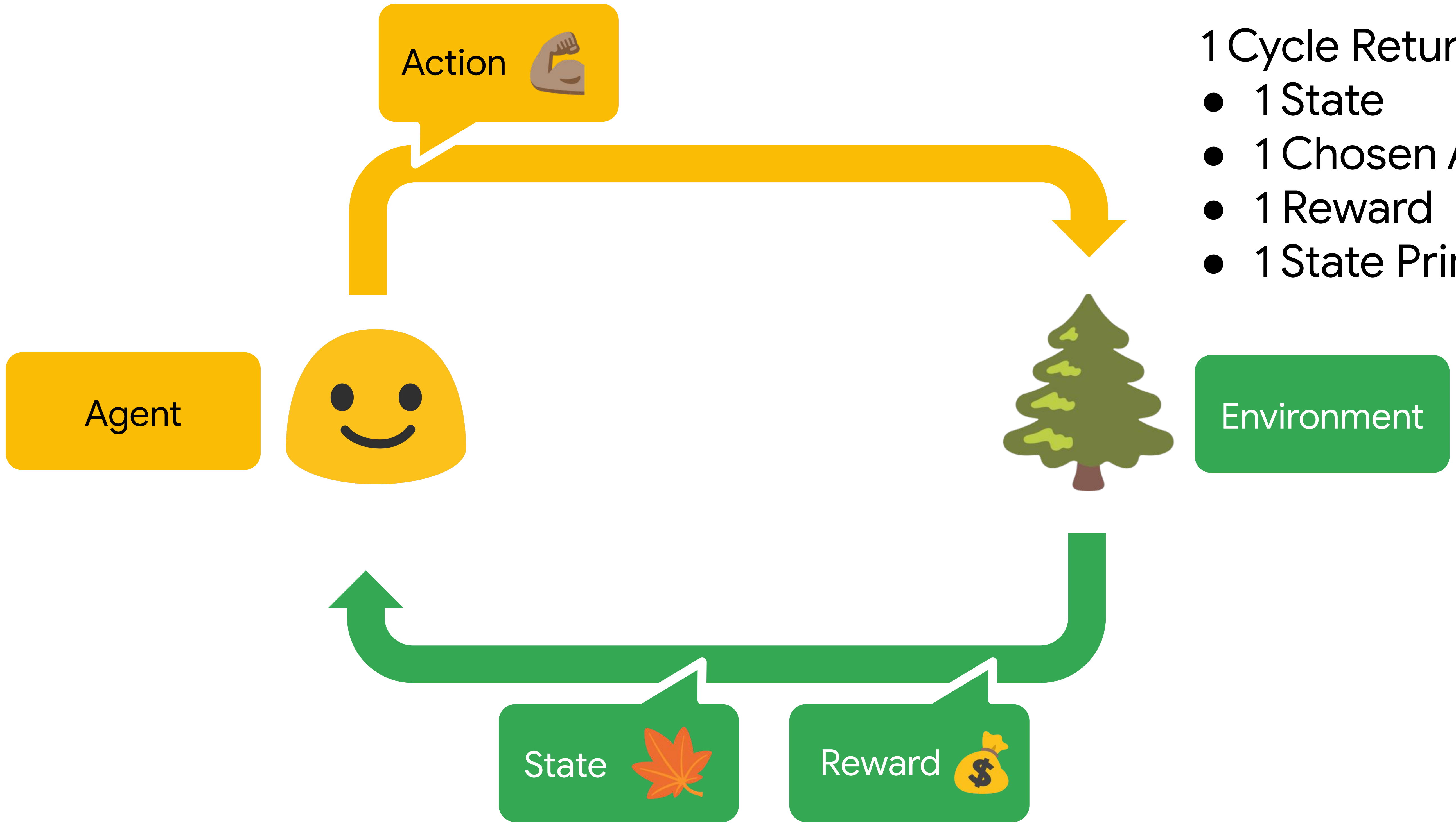
Deep Q Networks - Loss

Deep Q Networks - Memory

Deep Q Networks - Code

Experience Replay

Deep Q Learning - Memory



1 Cycle Returns:

- 1 State
- 1 Chosen Action
- 1 Reward
- 1 State Prime

Deep Q Learning - Memory

Memory Buffer				
Idx	state	action	reward	state prime
0	s_0	a_0	r_0	s_1
1	s_1	a_1	r_1	s_2
2	s_2	a_2	r_2	s_3
...				

1 Cycle Returns:

- 1 State
- 1 Chosen Action
- 1 Reward
- 1 State Prime

Deep Q Learning - Memory

Memory Buffer				
Idx	state	action	reward	state prime
0	s_0	a_0	r_0	s_1
1	s_1	a_1	r_1	s_2
2	s_2	a_2	r_2	s_3
...				

1 Cycle Returns:

- 1 State
- 1 Chosen Action
- 1 Reward
- 1 State Prime

Google Cloud

Deep Q Learning - Memory

Memory Buffer				
Idx	state	action	reward	state prime
0	s_0	a_0	r_0	s_1
1	s_1	a_1	r_1	s_2
2	s_2	a_2	r_2	s_3
...				

Training Sample				
Idx	state	action	reward	state prime
2	s_2	a_2	r_2	s_3
11	s_{11}	a_{11}	r_{11}	s_{12}
25	s_{25}	a_{25}	r_{25}	s_{26}
...				

Google Cloud

The Memory Buffer

```
class Memory():
    def __init__(self, memory_size, batch_size):
        ...

    def add(self, experience):
        ...

    def sample(self):
        ...
```

The Memory Buffer

```python
class Memory():
    def __init__(self, memory_size, batch_size):
        self.buffer = deque(maxlen=memory_size)
        self.batch_size = batch_size

    def add(self, experience):
        ...

    def sample(self):
        ...
```

Google Cloud

The Memory Buffer

```
class Memory():
    def __init__(self, memory_size, batch_size):
        self.buffer = deque(maxlen=memory_size)
        self.batch_size = batch_size

    def add(self, experience):
        # Adds a (state, action, reward, state_prime, done) tuple.
        self.buffer.append(experience)

    def sample(self):
        ...
```

The Memory Buffer

```python
class Memory():
    def __init__(self, memory_size, batch_size):
        self.buffer = deque(maxlen=memory_size)
        self.batch_size = batch_size

    def add(self, experience):
        # Adds a (state, action, reward, state_prime, done) tuple.
        self.buffer.append(experience)

    def sample(self):
        buffer_size = len(self.buffer)
        index = np.random.choice(
            np.arange(buffer_size), size=self.batch_size, replace=False)
        batch = [self.buffer[i] for i in index]
        return batch
```

Google Cloud

Experience Replay

Google Cloud

Agenda

TD-Gammon

Deep Q Networks - Loss

Deep Q Networks - Memory

Deep Q Networks - Code

Deep Q Learning - Network

```
def deep_q_network(state_shape, action_size, learning_rate, hidden_neurons):
    state_input = Input(state_shape, name='frames')

    hidden_1 = Dense(hidden_neurons, activation='relu')(state_input)
    hidden_2 = Dense(hidden_neurons, activation='relu')(hidden_1)
    q_values = Dense(action_size)(hidden_2)

    model = Model(inputs=[state_input], outputs=q_values)
    optimizer = tf.keras.optimizers.RMSprop(lr=learning_rate)
    model.compile(loss='mse', optimizer=optimizer)
    return model
```

Deep Q Learning - Network (advanced)

```python
def deep_q_network(state_shape, action_size, learning_rate, hidden_neurons):
    state_input = Input(state_shape, name='frames')
    actions_input = Input((action_size,), name='mask')

    hidden_1 = Dense(hidden_neurons, activation='relu')(state_input)
    hidden_2 = Dense(hidden_neurons, activation='relu')(hidden_1)
    q_values = Dense(action_size)(hidden_2)
    masked_q_values = Multiply()([q_values, actions_input])

    model = Model(inputs=[state_input, actions_input], outputs=masked_q_values)
    optimizer = tf.keras.optimizers.RMSprop(lr=learning_rate)
    model.compile(loss='mse', optimizer=optimizer)
    return model
```

Deep Q Learning - Network (advanced)

```
def deep_q_network(state_shape, action_size, learning_rate, hidden_neurons):
    state_input = Input(state_shape, name='frames')
    actions_input = Input((action_size,), name='mask')

                                             ation='relu')(state_input)
                                             ation='relu')(hidden_1)
                                             )
                                             , actions_input])

    model = Model(inputs=[state_input, actions_input], outputs=masked_q_values)
    optimizer = tf.keras.optimizers.RMSprop(lr=learning_rate)
    model.compile(loss='mse', optimizer=optimizer)
    return model
```

Training			
a_0	a_1	a_2	a_3
0	0	1	0

Google Cloud

Deep Q Learning - Network (advanced)

```
def deep_q_network(state_shape, action_size, learning_rate, hidden_neurons):
    state_input = Input(state_shape, name='frames')
    actions_input = Input((action_size,), name='mask')

                                                ation='relu')(state_input)
                                                ation='relu')(hidden_1)
                                                )
                                                , actions_input])

    model = Model(inputs=[state_input, actions_input], outputs=masked_q_values)
    optimizer = tf.keras.optimizers.RMSprop(lr=learning_rate)
    model.compile(loss='mse', optimizer=optimizer)
    return model
```

Predicting			
a_0	a_1	a_2	a_3
1	1	1	1

Deep Q Learning - The Act Function

```
def act(self, state, training=False):
    if training:
        # Random actions until enough simulations to train the model.
        if len(self.memory.buffer) >= self.memory.batch_size:
            self.random_rate *= self.random_decay

        if self.random_rate > np.random.rand():
            return random.randint(0, self.action_size-1)

    # If not acting randomly, take action with highest predicted value.
    state_batch = np.expand_dims(state, axis=0)
    predict_mask = np.ones((1, self.action_size,))
    action_qs = self.network.predict([state_batch, predict_mask])
    return np.argmax(action_qs[0])
```

Deep Q Learning - The Act Function

```python
def act(self, state, training=False):
    if training:
        # Random actions until enough simulations to train the model.
        if len(self.memory.buffer) >= self.memory.batch_size:
            self.random_rate *= self.random_decay

        if self.random_rate > np.random.rand():
            return random.randint(0, self.action_size-1)

    # If not acting randomly, take action with highest predicted value.
    state_batch = np.expand_dims(state, axis=0)
    predict_mask = np.ones((1, self.action_size,))
    action_qs = self.network.predict([state_batch, predict_mask])
    return np.argmax(action_qs[0])
```

Google Cloud

Deep Q Learning - The Act Function

```
def act(self, state, training=False):
    if training:
        # Random actions until enough simulations to train the model.
        if len(self.memory.buffer) >= self.memory.batch_size:
            self.random_rate *= self.random_decay

        if self.random_rate > np.random.rand():
            return random.randint(0, self.action_size-1)

    # If not acting randomly, take action with highest predicted value.
    state_batch = np.expand_dims(state, axis=0)
    predict_mask = np.ones((1, self.action_size,))
    action_qs = self.network.predict([state_batch, predict_mask])
    return np.argmax(action_qs[0])
```

Google Cloud

Deep Q Learning - The Act Function

```
def act(self, state, training=False):
    if training:
        # Random actions until enough simulations to train the model.
        if len(self.memory.buffer) >= self.memory.batch_size:
            self.random_rate *= self.random_decay

        if self.random_rate > np.random.rand():
            return random.randint(0, self.action_size-1)

    # If not acting randomly, take action with highest predicted value.
    state_batch = np.expand_dims(state, axis=0)
    predict_mask = np.ones((1, self.action_size,))
    action_qs = self.network.predict([state_batch, predict_mask])
    return np.argmax(action_qs[0])
```

Deep Q Learning - Update Q Function

```
def update_Q(self):
    state_mb, action_mb, reward_mb, state_prime_mb, done_mb = (
        self.memory.sample())

    # Get Q values for state_prime_mb.
    ...

    # Apply the Bellman Equation
    ...

    # Match training batch to network output
    ...
```

Deep Q Learning - Update Q Function

```
def update_Q(self):
    state_mb, action_mb, reward_mb, state_prime_mb, done_mb = (
        self.memory.sample())

    # Get Q values for state_prime_mb.
    predict_mask = np.ones(action_mb.shape + (self.action_size,))
    next_q_mb = self.network.predict([state_prime_mb, predict_mask])
    next_q_mb = tf.math.reduce_max(next_q_mb, axis=1)

    # Apply the Bellman Equation
    ...

    # Match training batch to network output
    ...
```

Google Cloud

Deep Q Learning - Update Q Function

```
def update_Q(self):
    state_mb, action_mb, reward_mb, state_prime_mb, done_mb = (
        self.memory.sample())

    # Get Q values for state_prime_mb.
    predict_mask = np.ones(action_mb.shape + (self.action_size,))
    next_q_mb = self.network.predict([state_prime_mb, predict_mask])
    next_q_mb = tf.math.reduce_max(next_q_mb, axis=1)

    # Apply the Bellman Equation
    target_qs = (next_q_mb * self.memory.gamma) + reward_mb
    target_qs = tf.where(done_mb, reward_mb, target_qs)

    # Match training batch to network output
    ...
```

Deep Q Learning - Update Q Function

```
def update_Q(self):
    state_mb, action_mb, reward_mb, state_prime_mb, done_mb = (
        self.memory.sample())

    # Get Q values for state_prime_mb.
    predict_mask = np.ones(action_mb.shape + (self.action_size,))
    next_q_mb = self.network.predict([state_prime_mb, predict_mask])
    next_q_mb = tf.math.reduce_max(next_q_mb, axis=1)

    # Apply the Bellman Equation
    target_qs = (next_q_mb * self.memory.gamma) + reward_mb
    target_qs = tf.where(done_mb, reward_mb, target_qs)

    # Match training batch to network output
    ...
```

Deep Q Learning - Update Q Function

```
def update_Q(self):
    state_mb, action_mb, reward_mb, state_prime_mb, done_mb = (
        self.memory.sample())

    # Get Q values for state_prime_mb.
    predict_mask = np.ones(action_mb.shape + (self.action_size,))
    next_q_mb = self.network.predict([state_prime_mb, predict_mask])
    next_q_mb = tf.math.reduce_max(next_q_mb, axis=1)

    # Apply the Bellman Equation
    target_qs = (next_q_mb * self.memory.gamma) + reward_mb
    target_qs = tf.where(done_mb, reward_mb, target_qs)

    # Match training batch to network output
    action_mb = tf.convert_to_tensor(action_mb, dtype=tf.int32)
    action_hot = tf.one_hot(action_mb, self.action_size)
    target_mask = tf.multiply(tf.expand_dims(target_qs, -1), action_hot)
    return self.network.train_on_batch([state_mb, action_hot], target_mask)
```

Deep Q Learning - Update Q Function

```python
def update_Q(self):
    state_mb, action_mb, reward_mb, state_prime_mb, done_mb = (
        self.memory.sample())

    # Get Q values for state_prime_mb.
    predict_mask = np.ones(action_mb.shape + (self.action_size,))
    next_q_mb = self.network.predict([state_prime_mb, predict_mask])
    next_q_mb = tf.math.reduce_max(next_q_mb, axis=1)

    # Apply the Bellman Equation
                                              mma) + reward_mb
                                              , target_qs)

                                              t
    action_mb = tf.convert_to_tensor(action_mb, dtype=tf.int32)
    action_hot = tf.one_hot(action_mb, self.action_size)
    target_mask = tf.multiply(tf.expand_dims(target_qs, -1), action_hot)
    return self.network.train_on_batch([state_mb, action_hot], target_mask)
```

Training			
a_0	a_1	a_2	a_3
0	0	1	0

Google Cloud

Deep Q Learning - Update Q Function

```python
def update_Q(self):
    state_mb, action_mb, reward_mb, state_prime_mb, done_mb = (
        self.memory.sample())

    # Get Q values for state_prime_mb.
    predict_mask = np.ones(action_mb.shape + (self.action_size,))
    next_q_mb = self.network.predict([state_prime_mb, predict_mask])
    next_q_mb = tf.math.reduce_max(next_q_mb, axis=1)

    # Apply the Bellman Equation
    target_qs = (next_q_mb * self.memory.gamma) + reward_mb
    target_qs = tf.where(done_mb, reward_mb, target_qs)

    # Match training batch to network output
    action_mb = tf.convert_to_tensor(action_mb, dtype=tf.int32)
    action_hot = tf.one_hot(action_mb, self.action_size)
    target_mask = tf.multiply(tf.expand_dims(target_qs, -1), action_hot)
    return self.network.train_on_batch([state_mb, action_hot], target_mask)
```

Lab

Use Reinforcement Learning in Trading

Google Cloud

Lab Objectives

-
-

Google Cloud

Screencast

Google Cloud

Policy Gradients

Agenda

Policy Gradients

Actor - Critic

Agenda

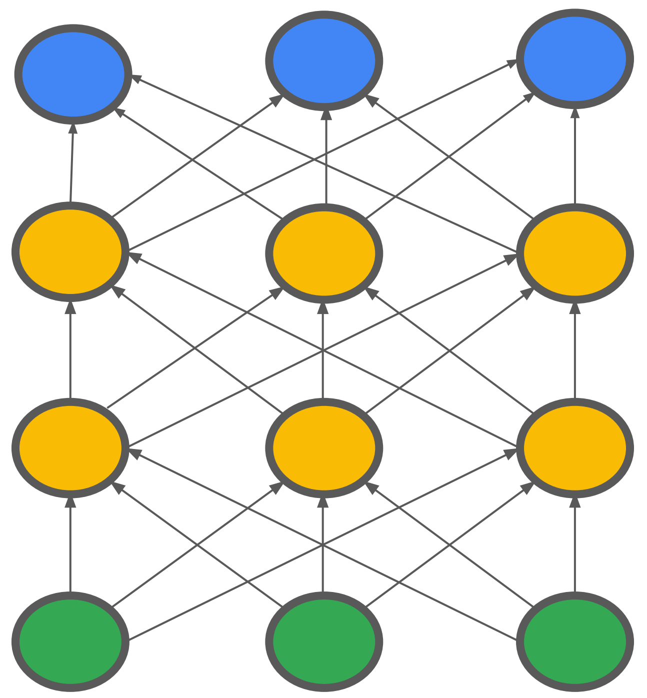
Policy Gradients

Actor - Critic

Deep Q vs Policy Gradients

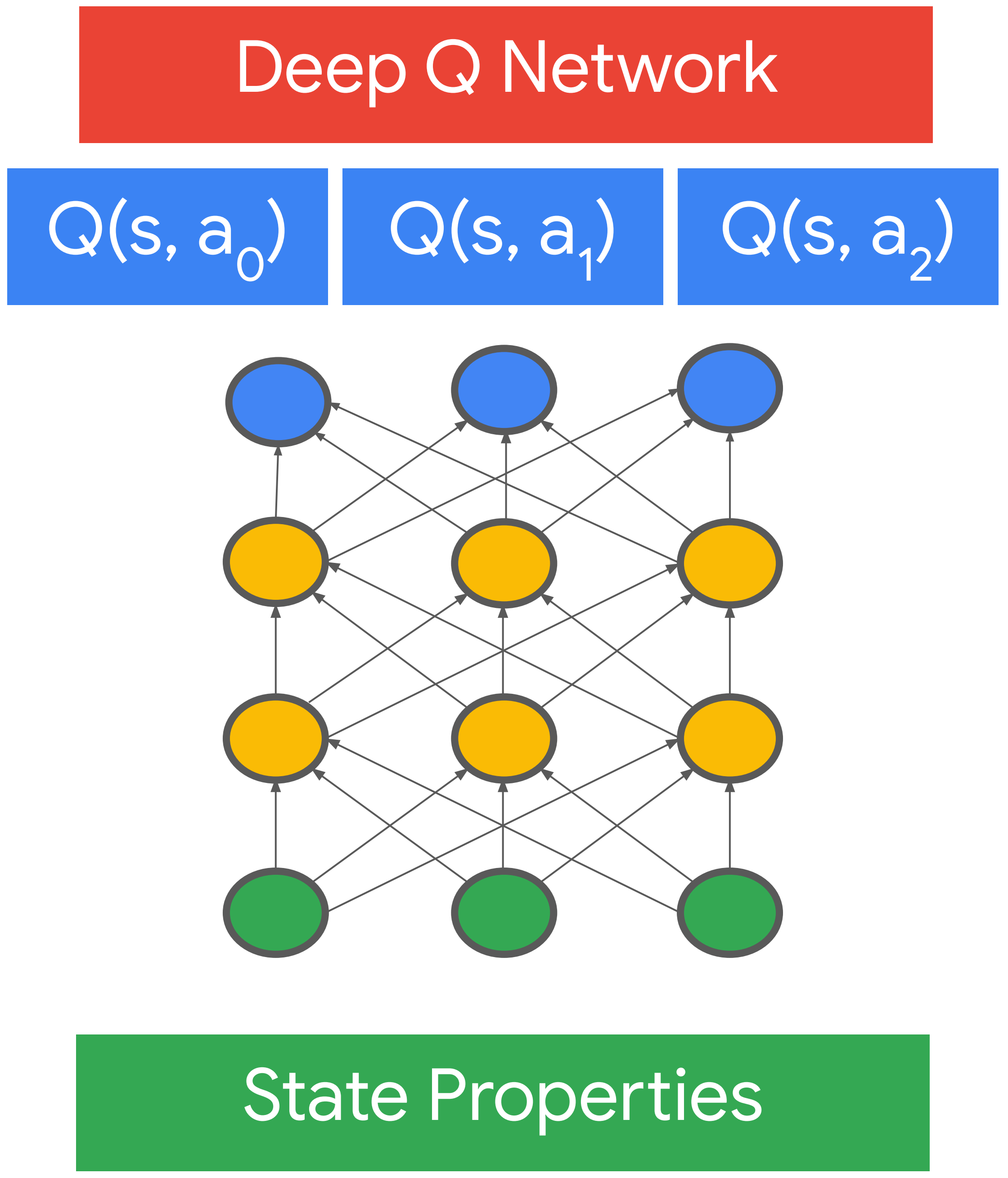
Deep Q Network

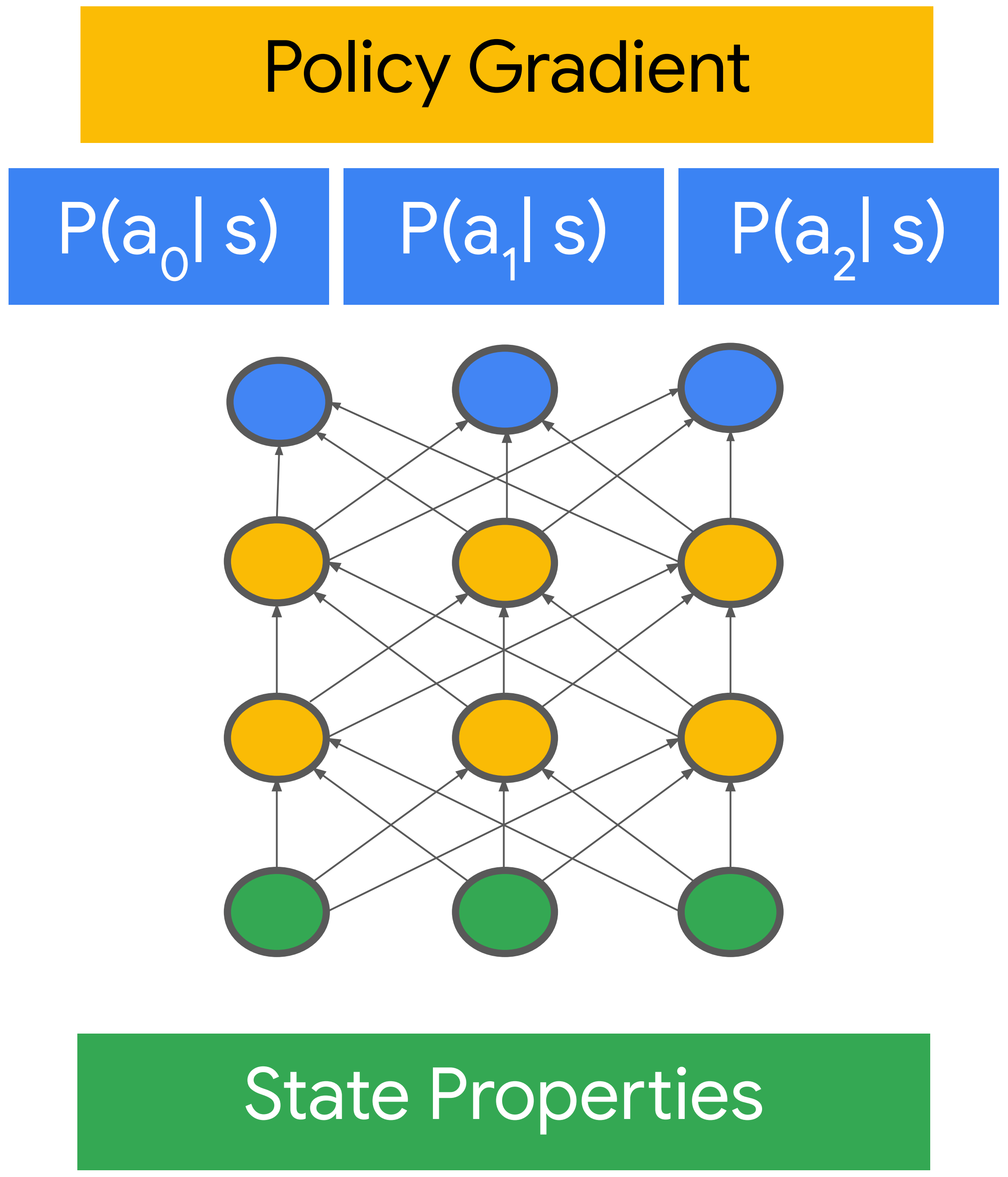
$Q(s, a_0)$ | $Q(s, a_1)$ | $Q(s, a_2)$



State Properties

Deep Q vs Policy Gradients





Google Cloud

Deep Q vs Policy Gradients

Deep Q Network		
$Q(s, a_0)$	$Q(s, a_1)$	$Q(s, a_2)$

Policy Gradient		
$P(a_0 \| s)$	$P(a_1 \| s)$	$P(a_2 \| s)$

Deep Q vs Policy Gradients

Deep Q Network		
$Q(s, a_0)$	$Q(s, a_1)$	$Q(s, a_2)$
.4	.3	.3

Policy Gradient		
$P(a_0 \| s)$	$P(a_1 \| s)$	$P(a_2 \| s)$

Google Cloud

Deep Q vs Policy Gradients

Deep Q Network

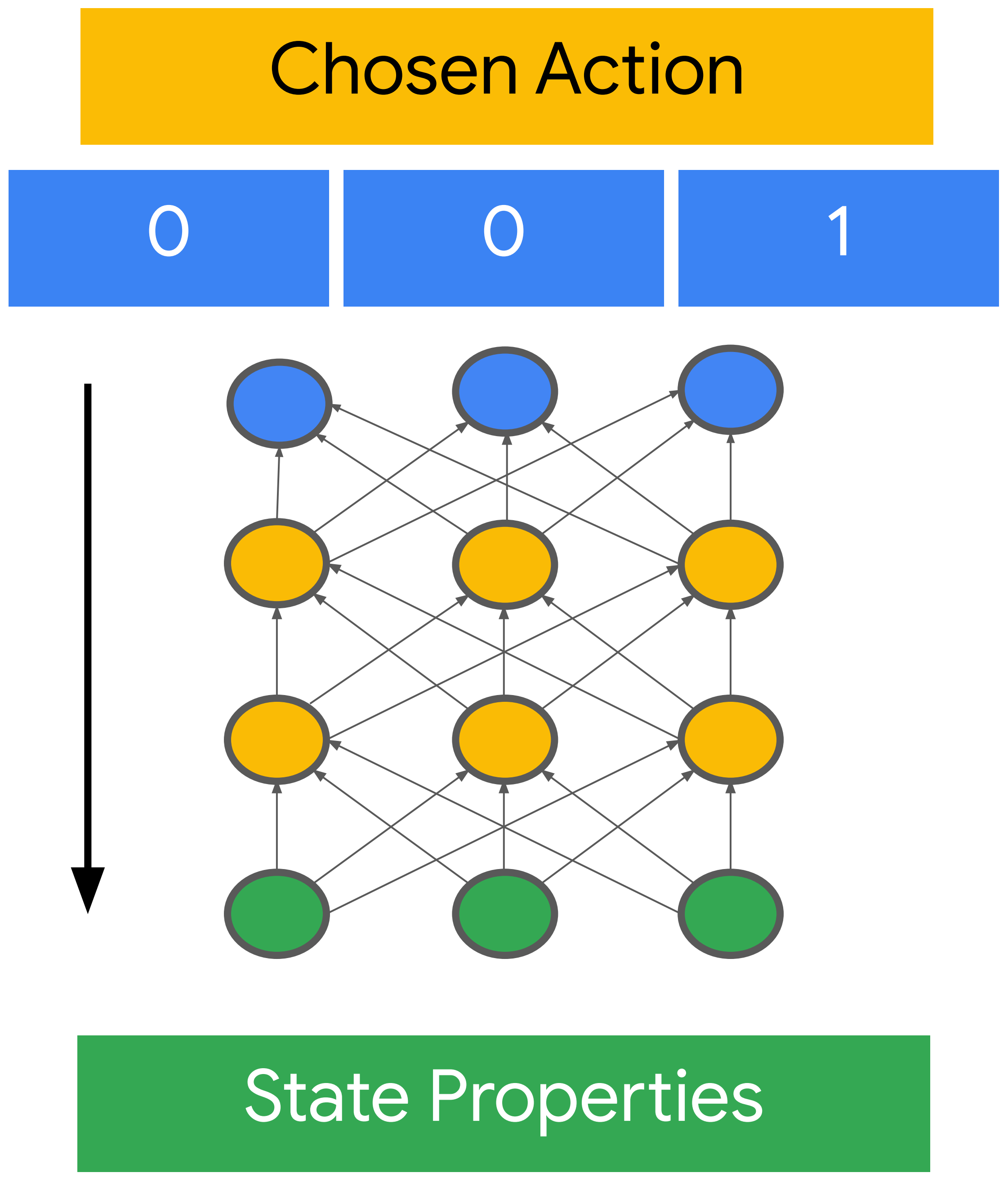
$Q(s, a_0)$	$Q(s, a_1)$	$Q(s, a_2)$

Policy Gradient

$P(a_0 \| s)$	$P(a_1 \| s)$	$P(a_2 \| s)$
.4	.3	.3

Google Cloud

Policy Gradients - Loss



Policy Gradients - Loss

$$\Delta w = \square \nabla \pi_w(a^*, s)$$

Policy Gradients - Loss

$$\Delta w = \square \frac{\nabla \pi_w(a^*, s)}{\pi_w(a^*, s)}$$

Policy Gradients - Loss

$$\Delta w = \square \nabla_w log(\pi_w(a^*, s))$$

Policy Gradients - Loss

$$\Delta w = \Box \nabla_w log(\pi_w(a,s)) \cdot G_t$$

Policy Gradients - Loss

$$\Delta w = \square \nabla_w log(\pi_w(a,s)) \cdot G_t$$

```
def custom_loss(y_true, y_pred):
    y_pred_clipped = K.clip(y_pred, 1e-8, 1-1e-8)
    log_likelihood = y_true * K.log(y_pred_clipped)
    return K.sum(-log_likelihood*g)
```

Policy Gradients - Network

```
def build_networks(state_shape, action_size, learning_rate, hidden_neurons):
    state_input = Input(state_shape, name='frames')
    g = Input((1,), name='G')
    hidden_1 = Dense(hidden_neurons, activation='relu')(state_input)
    hidden_2 = Dense(hidden_neurons, activation='relu')(hidden_1)
    probabilities = Dense(action_size, activation='softmax')(hidden_2)

    def custom_loss(y_true, y_pred):
        # Previous slide.

    policy = Model(
        inputs=[state_input, g], outputs=[probabilities])
    optimizer = Adam(lr=learning_rate)
    policy.compile(loss=custom_loss, optimizer=optimizer)

    predict = Model(inputs=[state_input], outputs=[probabilities])
    return policy, predict
```

Policy Gradients - Memory

```
class Memory():
    def __init__(self, gamma):
        self.buffer = []
        self.gamma = gamma

    def add(self, experience):
        self.buffer.append(experience)

    def sample(self):
        batch = np.array(self.buffer).T.tolist()
        states_mb = np.array(batch[0], dtype=np.float32)
        actions_mb = np.array(batch[1], dtype=np.int8)
        rewards_mb = np.array(batch[2], dtype=np.float32)
        self.buffer = []
        return states_mb, actions_mb, rewards_mb
```

Policy Gradients - Training

```
def learn(self):
    """Trains the Deep Q Network based on stored experiences."""
    # Obtain random mini-batch from memory.
    state_mb, action_mb, reward_mb = self.memory.sample()
    actions = tf.one_hot(action_mb, self.action_size)

    # Normalized TD(1)
    discount_mb = np.zeros_like(reward_mb)
    total_rewards = 0
    for t in reversed(range(len(reward_mb))):
        total_rewards = reward_mb[t] + total_rewards * self.memory.gamma
        discount_mb[t] = total_rewards
    discount_mb = (discount_mb - np.mean(discount_mb)) / np.std(discount_mb)

    self.policy.train_on_batch([state_mb, discount_mb], actions)
```

Policy Gradients - Training

```
def learn(self):
    """Trains the Deep Q Network based on stored experiences."""
    # Obtain random mini-batch from memory.
    state_mb, action_mb, reward_mb = self.memory.sample()
    actions = tf.one_hot(action_mb, self.action_size)

    # Normalized TD(1)
    discount_mb = np.zeros_like(reward_mb)
    total_rewards = 0
    for t in reversed(range(len(reward_mb))):
        total_rewards = reward_mb[t] + total_rewards * self.memory.gamma
        discount_mb[t] = total_rewards
    discount_mb = (discount_mb - np.mean(discount_mb)) / np.std(discount_mb)

    self.policy.train_on_batch([state_mb, discount_mb], actions)
```

Policy Gradients Overview

```python
def act(self, state):
    state_batch = np.expand_dims(state, axis=0)
    probabilities = self.predict.predict(state_batch)[0]
    action = np.random.choice(self.action_size, p=probabilities)
    return action
```

✊ ✋ ✌️ 👍

Agenda

Policy Gradients

Actor - Critic

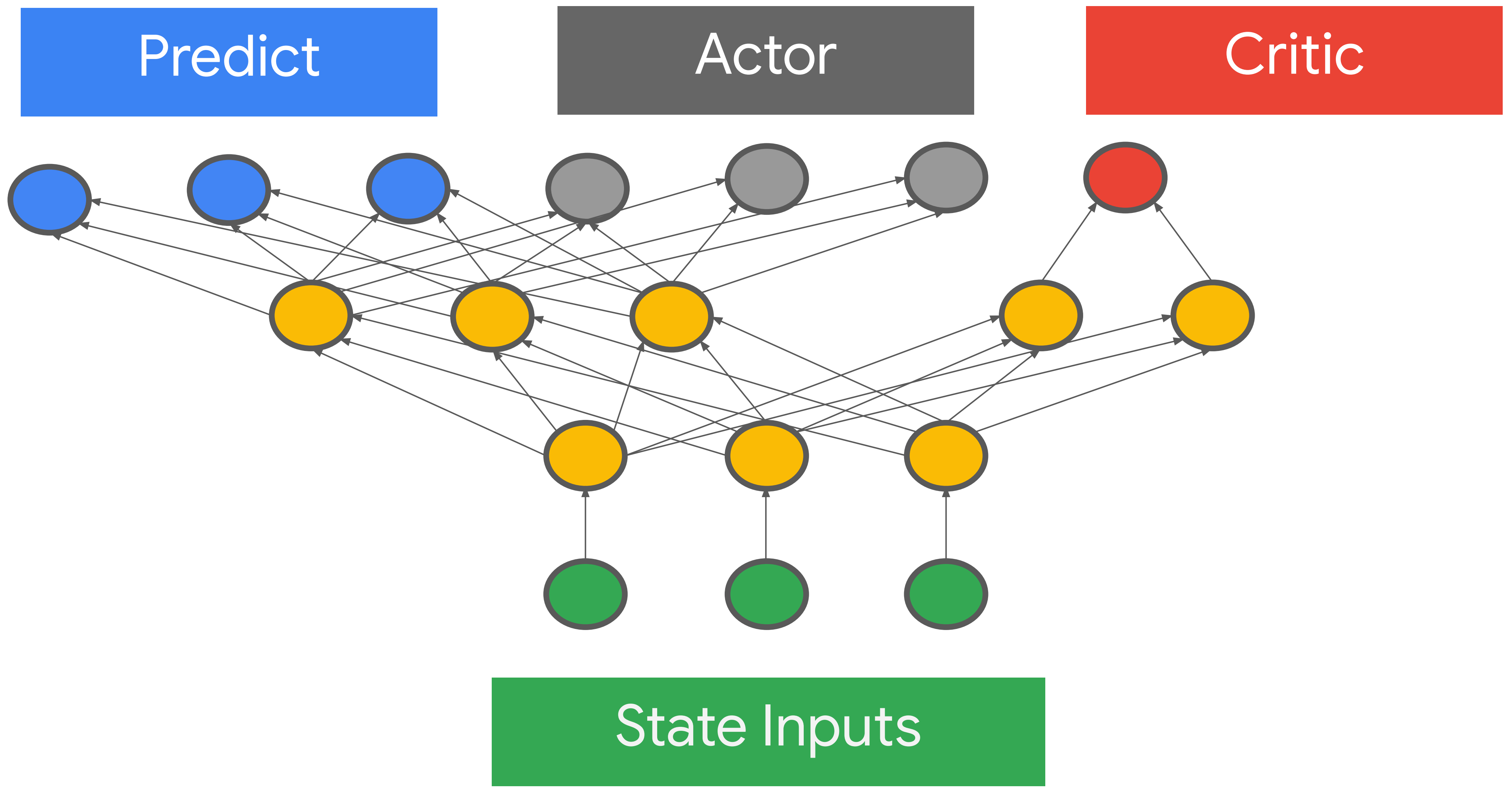
Breaking Down Q

$$Q(s, a) = V(s) + A(s, a)$$

Google Cloud

Breaking Down Q

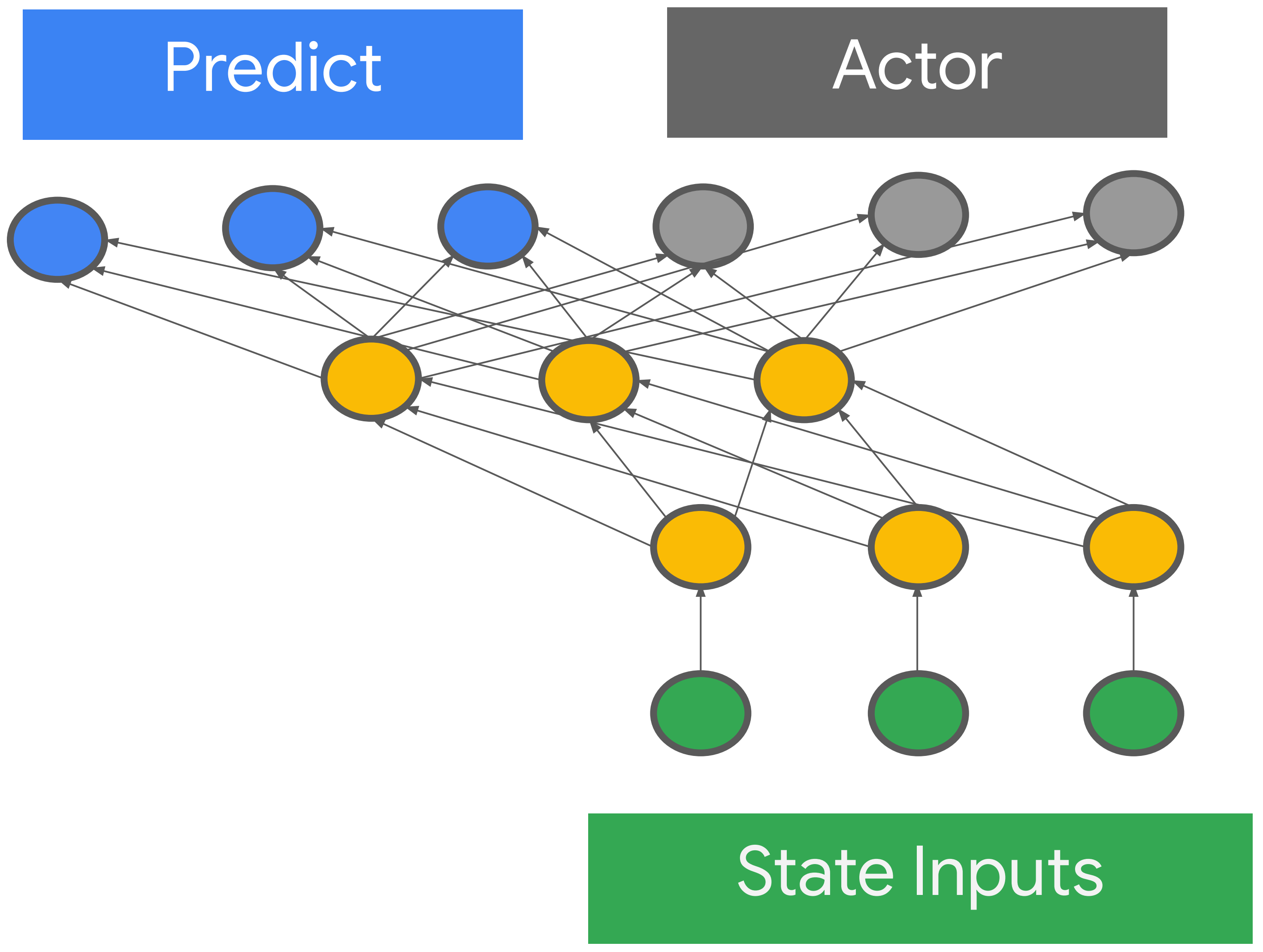
$$Q(s, a) = V(s) + A(s, a)$$



Google Cloud

Breaking Down Q

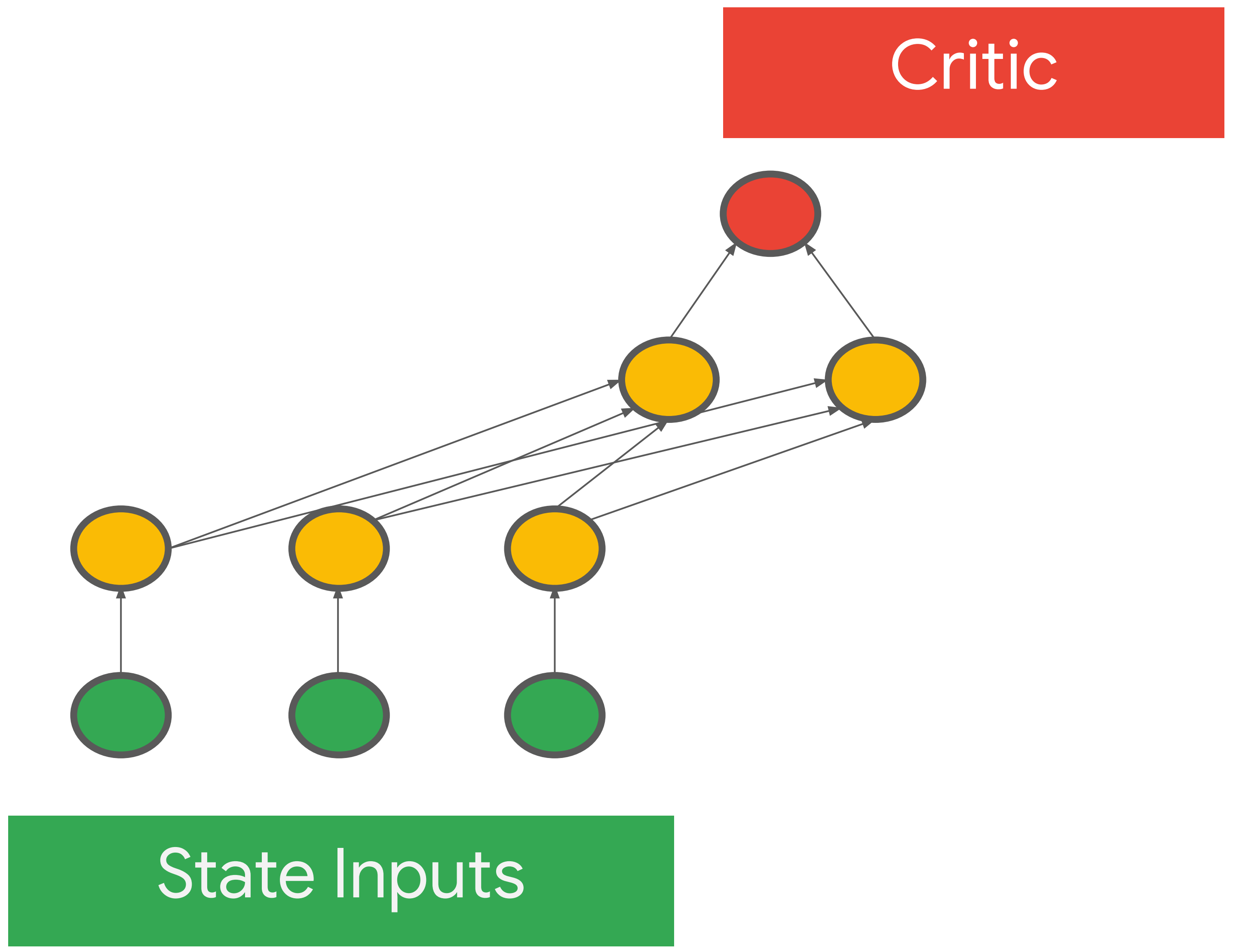
$$Q(s, a) = V(s) + A(s, a)$$



Google Cloud

Breaking Down Q

$$Q(s, a) = V(s) + A(s, a)$$



Google Cloud

A2C - Network

```
def build_networks(state_shape, action_size, actor_lr, critic_lr, neurons):
    state_input = layers.Input(state_shape, name='frames')
    advantage = layers.Input((1,), name='A')  # Now A instead of G.

    hidden_1 = layers.Dense(neurons, activation='relu')(state_input)
    hidden_2 = layers.Dense(neurons, activation='relu')(hidden_1)
    probabilities = layers.Dense(action_size, activation='softmax')(hidden_2)
    value = layers.Dense(1, activation='linear')(hidden_2)

    def custom_loss(y_true, y_pred):
        # Same as before.

    actor = Model(inputs=[state_input, advantages], outputs=[probabilities, values])
    actor.compile(loss=[custom_loss, 'mean_squared_error'], optimizer=Adam(lr=actor_lr))

    critic = Model(inputs=[state_input], outputs=[value])
    predict = Model(inputs=[state_input], outputs=[probabilities])
    return actor, critic, predict
```

Google Cloud

A2C - Training

```
def learn(self):
    """Trains the Deep Q Network based on stored experiences."""
    # Obtain random mini-batch from memory.
    state_mb, action_mb, reward_mb, dones_mb, next_v_mb = self.memory.sample()

    #Apply TD(0)
    discount_mb = reward_mb + next_v_mb * self.memory.gamma * (1 - dones_mb)
    state_values = self.critic.predict([state_mb])
    advantages = discount_mb - np.squeeze(state_values)
    self.actor.train_on_batch([state_mb, advantages], [action_mb, discount_mb])
```

Lab

Use Reinfocement Learning in Trading

Lab Objectives

-
-

Screencast

Google Cloud

What is LSTM?

Daniel Sparing
Machine Learning Solutions Engineer
Google Cloud

Agenda

Sequence Models

DNNs and RNNs for sequences

RNN limitations

LSTM

Applying LSTM to Time Series Data

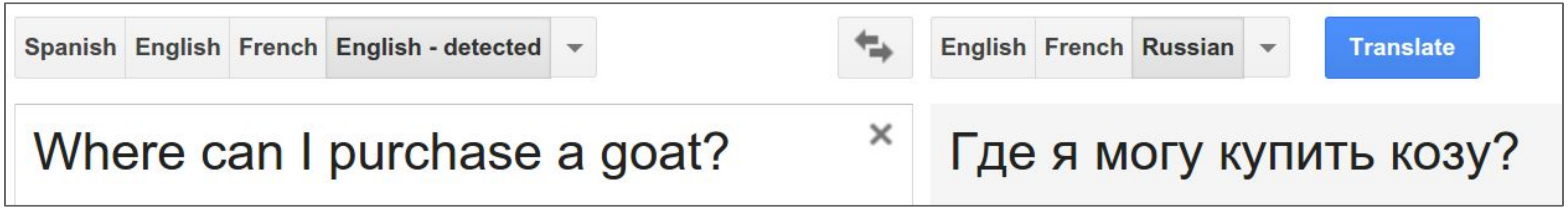
Google Cloud

Why Sequence Models?

Predict the next word

The cat sat on the ______.

Translate

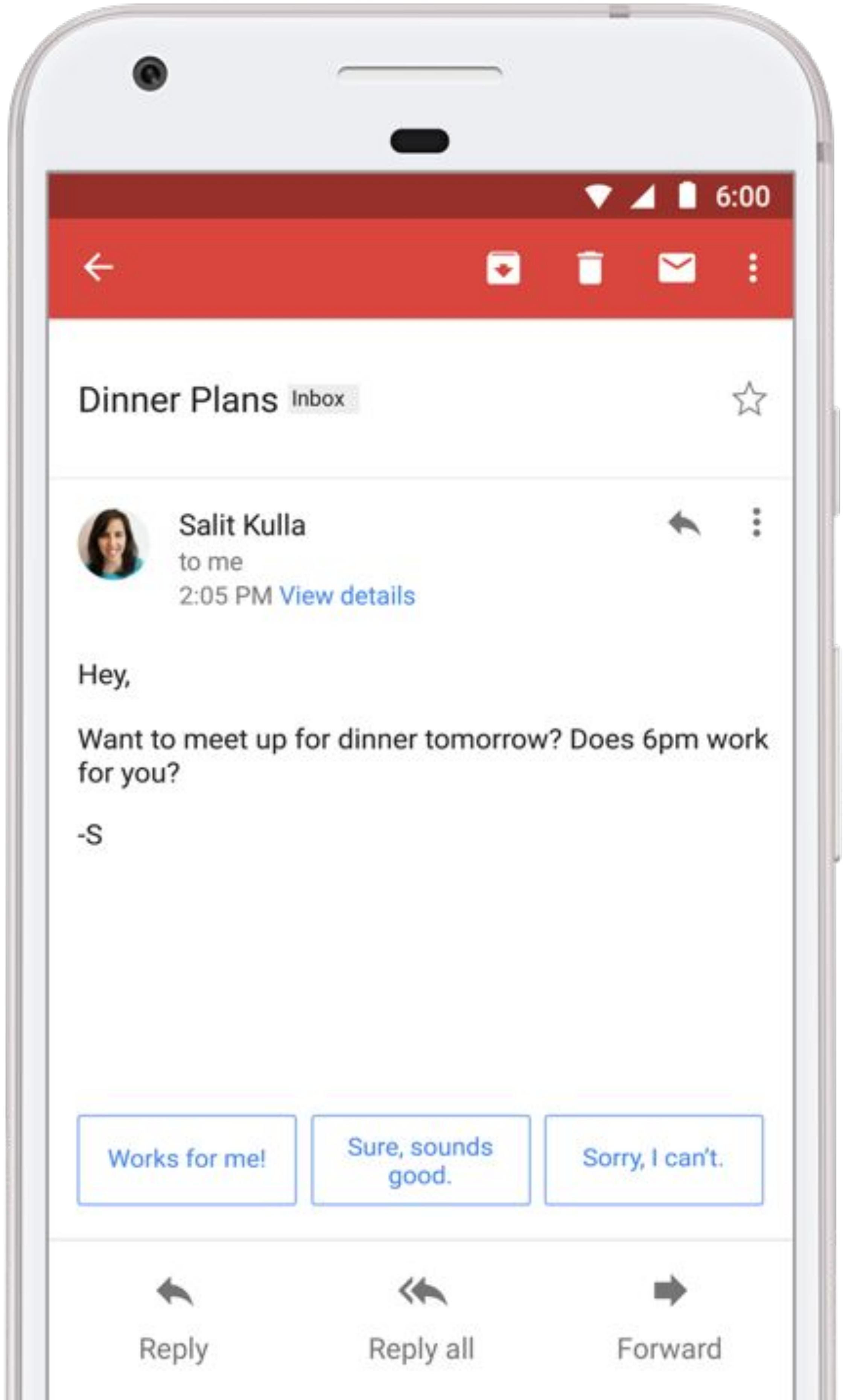


Where can I purchase a goat?

Где я могу купить козу?

Why Sequence Models?

Smart Reply



Salit Kulla
to me
2:05 PM View details

Hey,

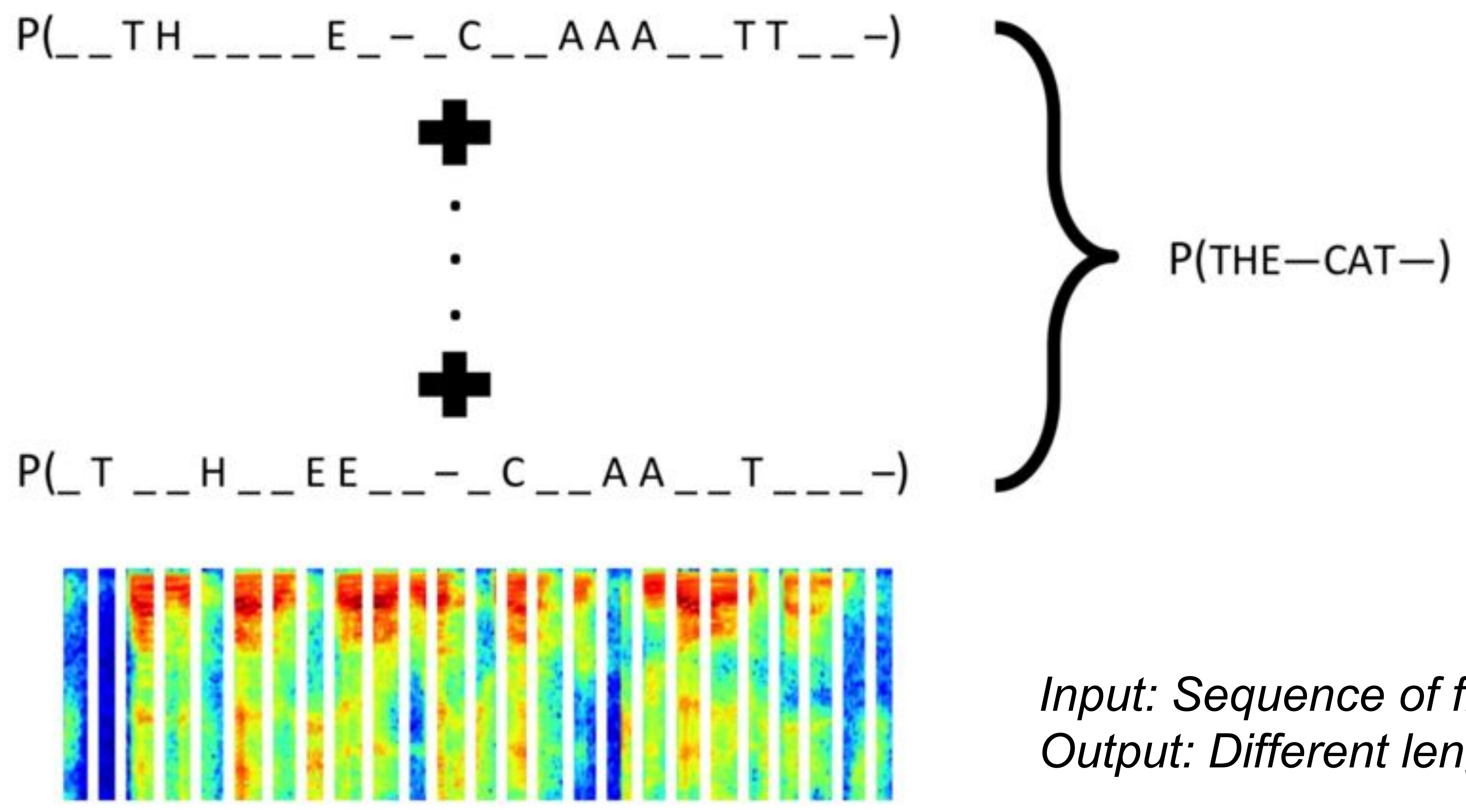
Want to meet up for dinner tomorrow? Does 6pm work for you?

-S

Google Cloud

Why Sequence Models?

Speech recognition



Input: Sequence of float vectors (windowed Fourier Transforms)
Output: Different length sequence of characters

Google Cloud

Agenda

Sequence Models

DNNs and RNNs for sequences

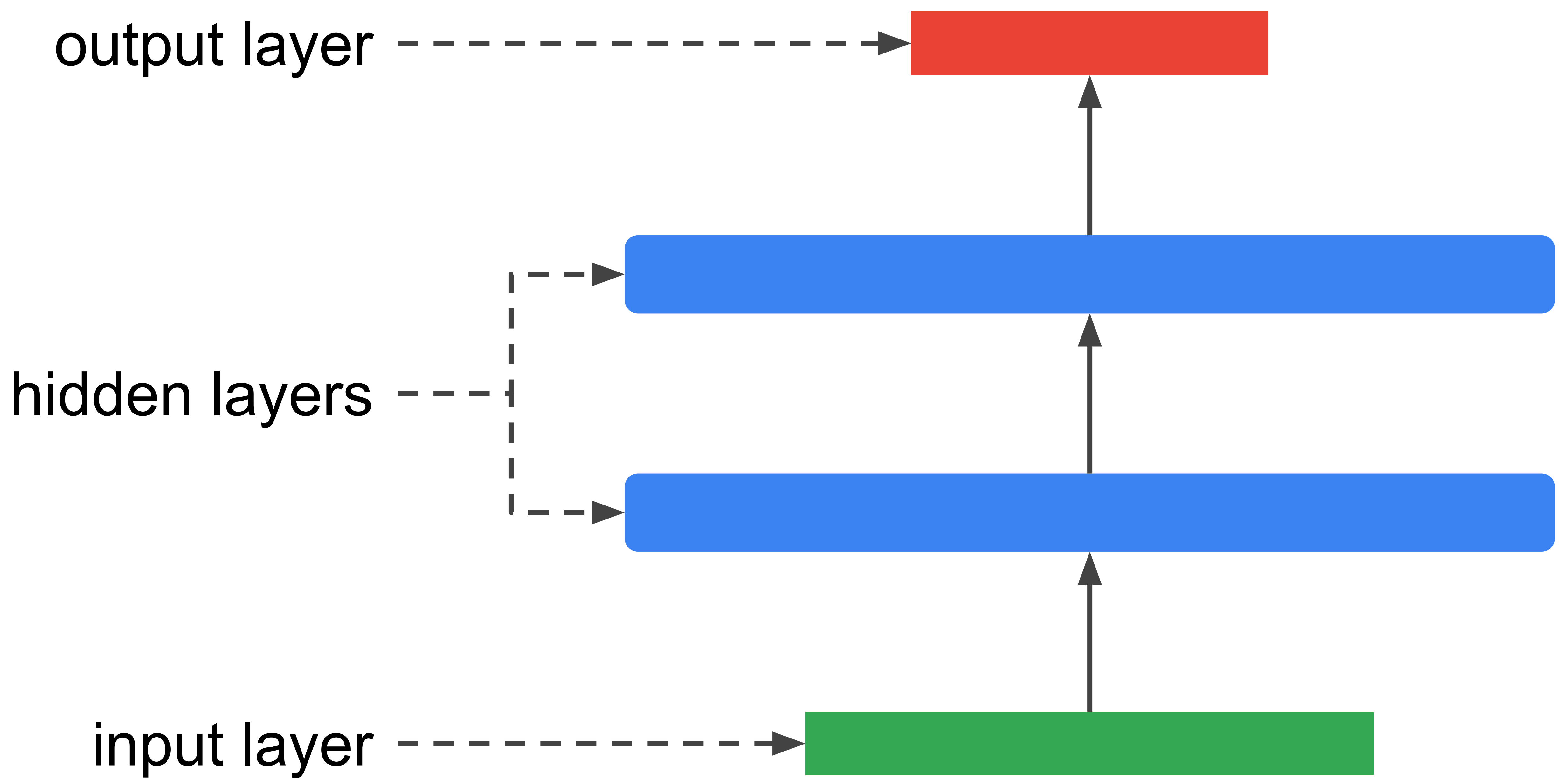
RNN limitations

LSTM

Applying LSTM to Time Series Data

Google Cloud

Feed Forward Networks



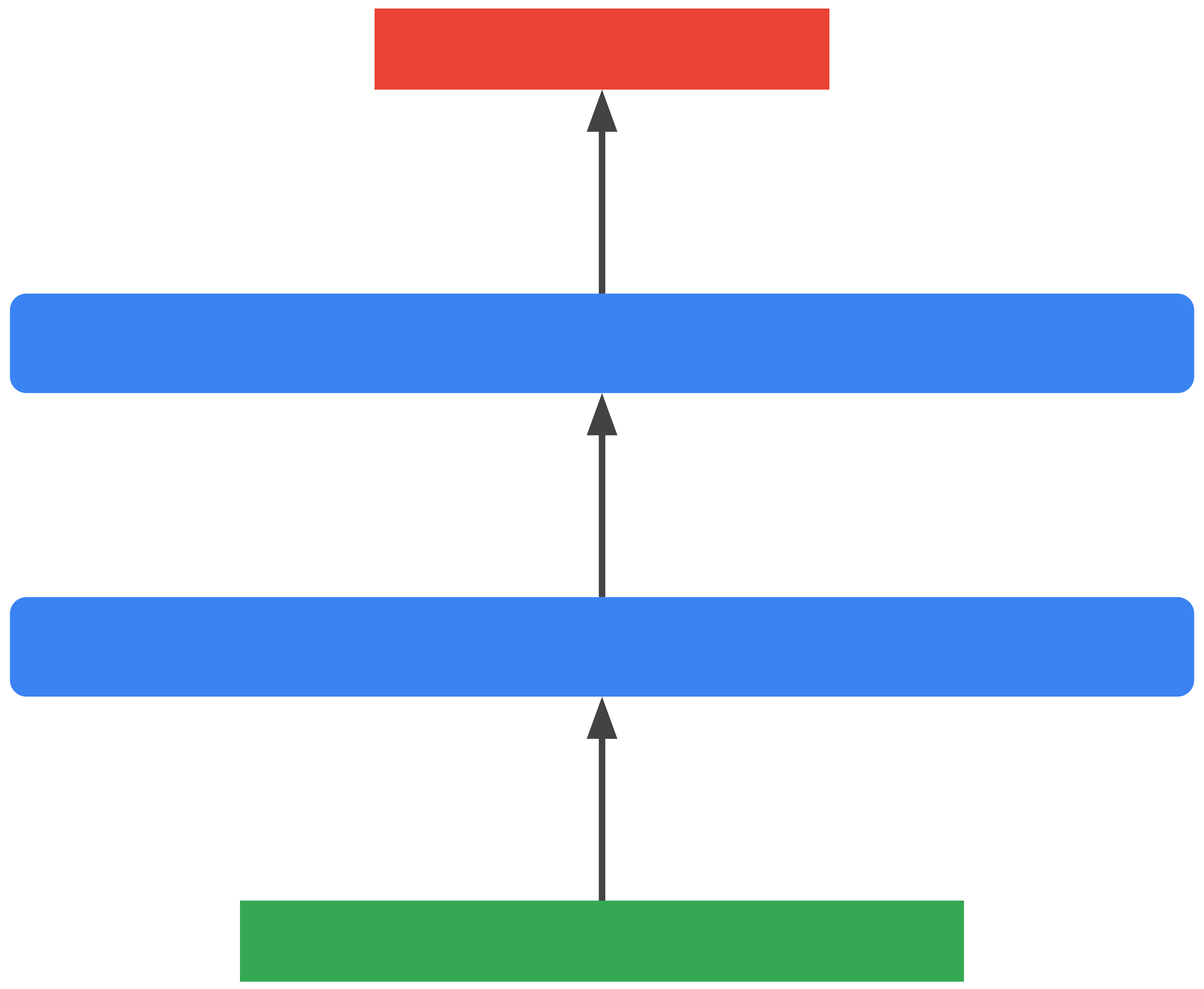
Google Cloud

Feed Forward Networks

Fixed size layers

Inference is stateless

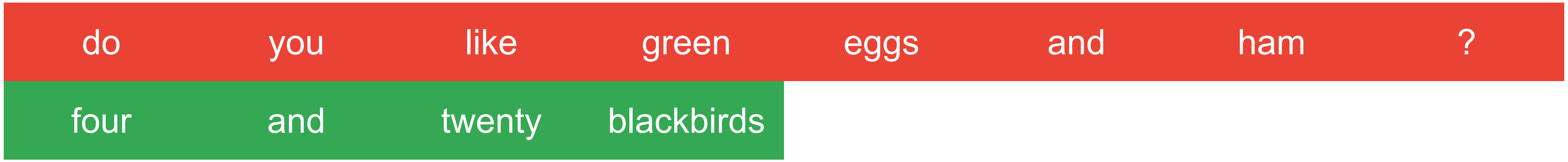
Nodes are unordered



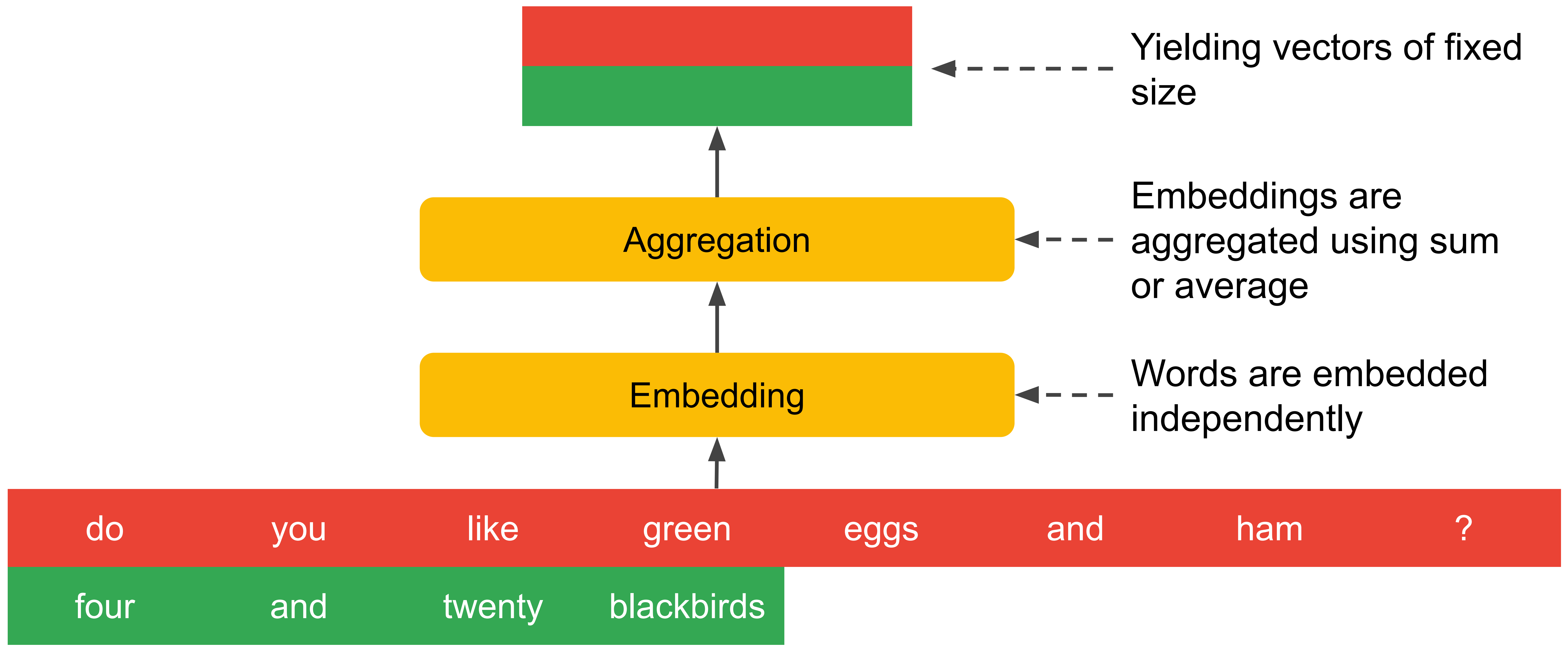
Google Cloud

Language as Input

Input to a language model can have variable length. For example,



Language as Input: the “Typical” Approach

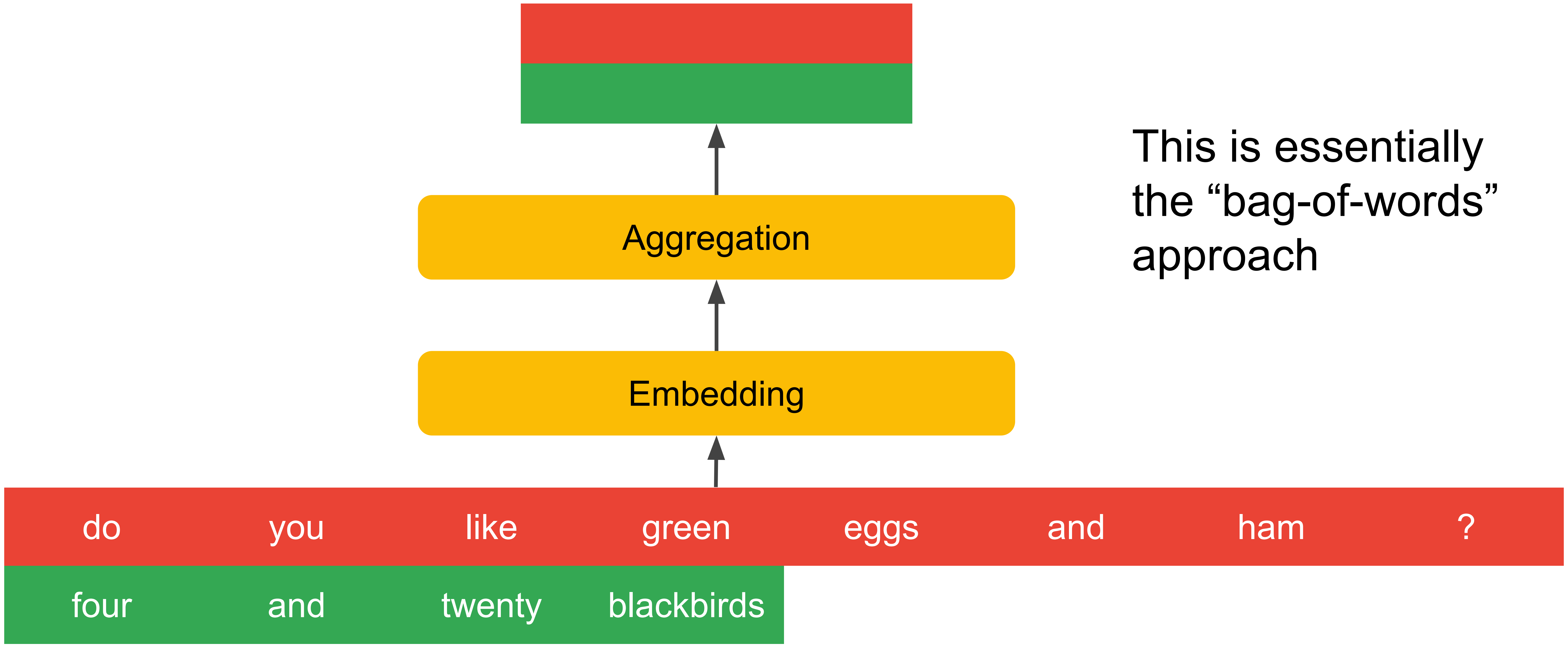


Yielding vectors of fixed size

Embeddings are aggregated using sum or average

Words are embedded independently

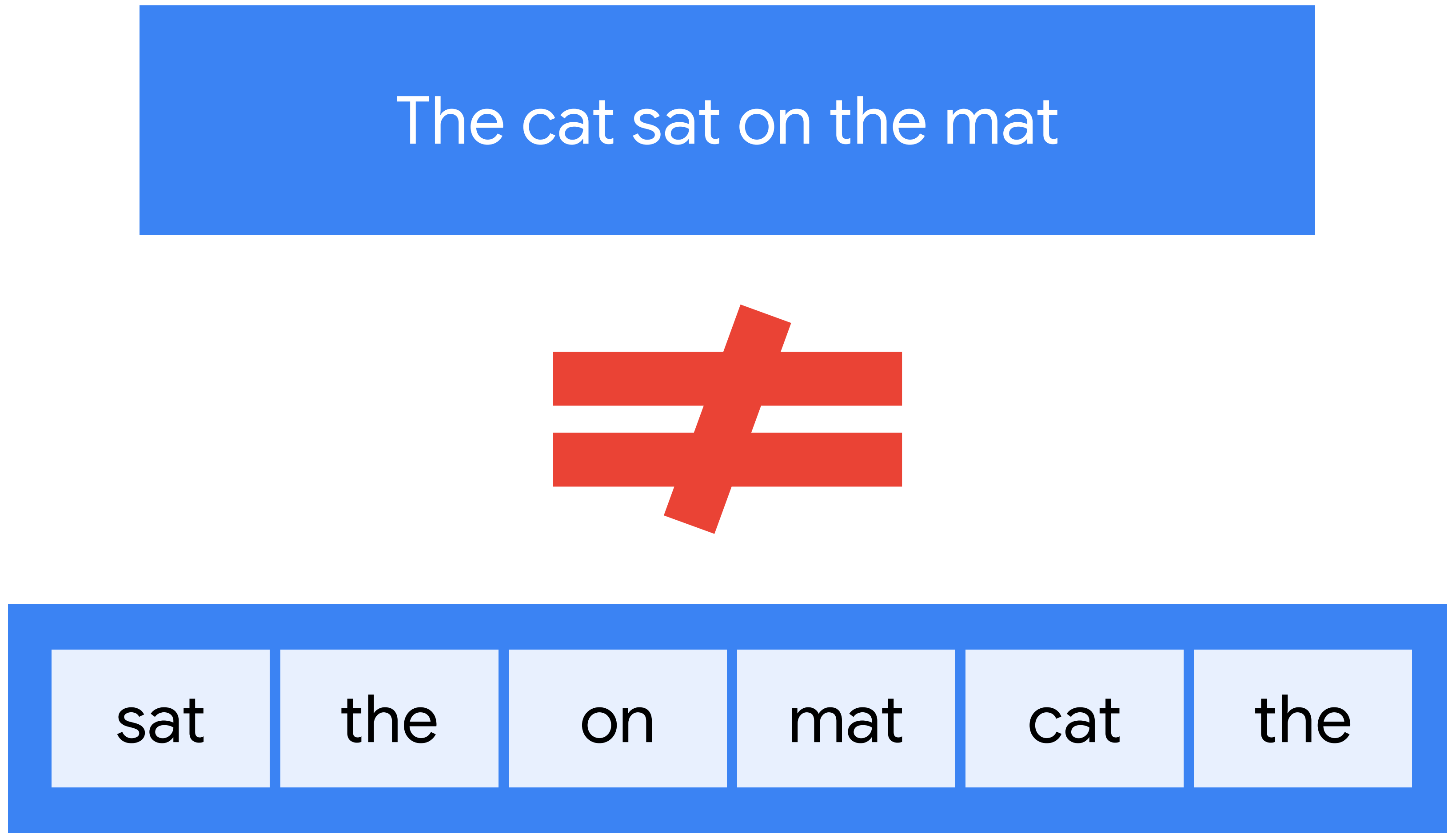
Language as Input: the “Typical” Approach



This is essentially the “bag-of-words” approach

Google Cloud

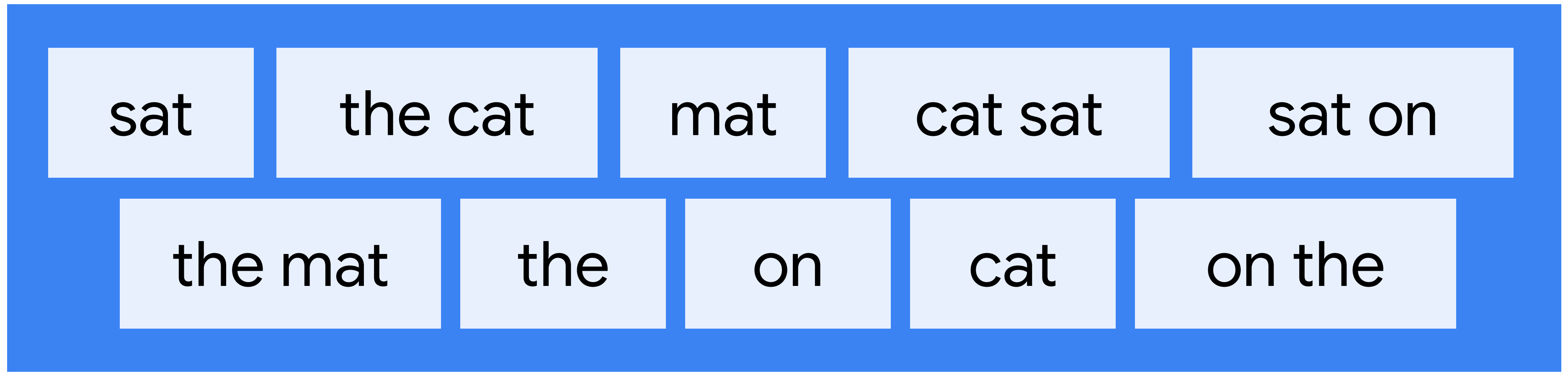
Structure is Important



- Certain tasks, structure is essential:
  - Humor
  - Sarcasm
- Certain tasks, ngrams can get you a long way:
  - Sentiment Analysis
  - Topic detection
- Specific words can be strong indicators
  - Useless, fantastic (sentiment)
  - Hoop, green tea, NASDAQ (topic)

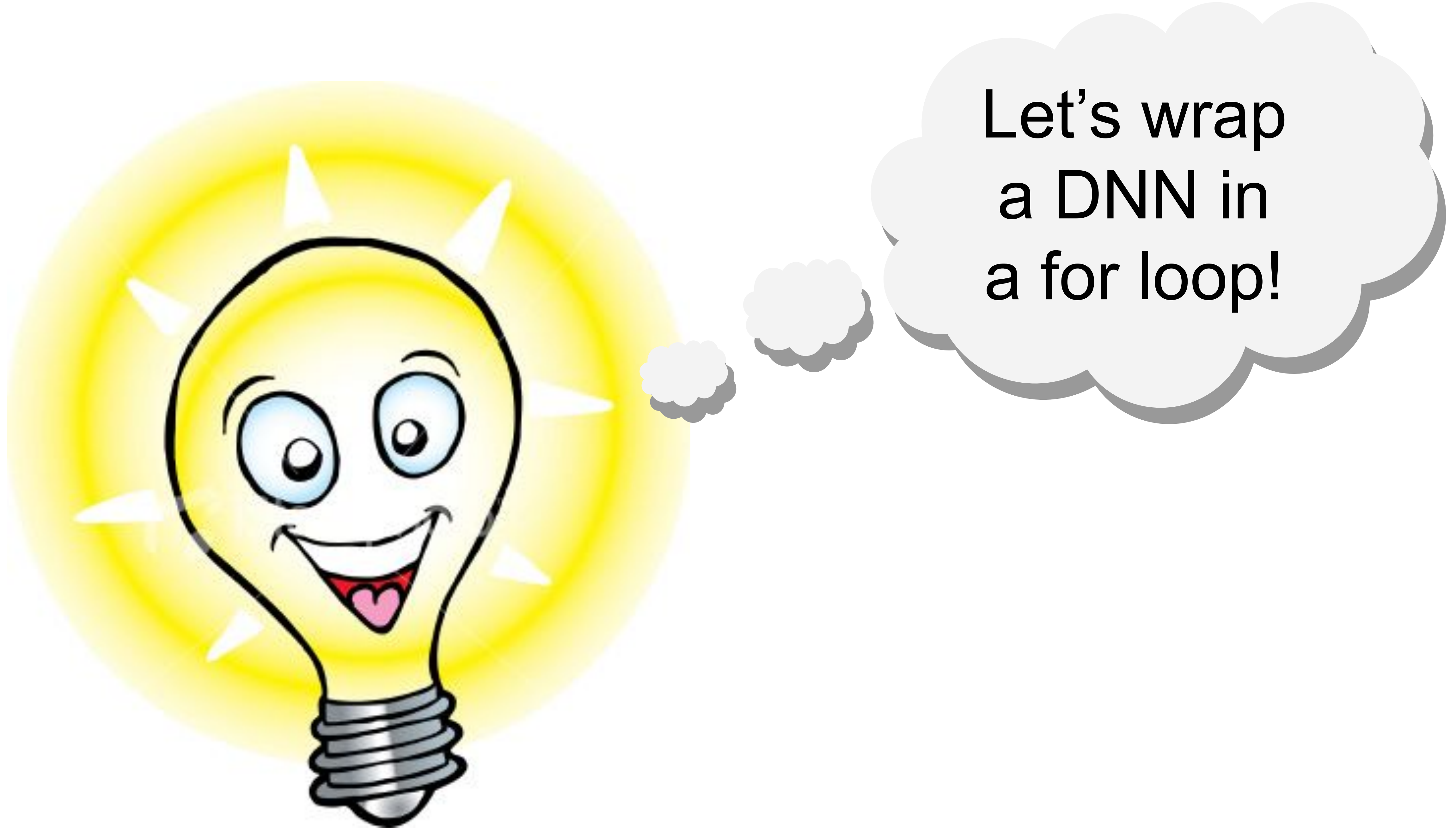
Structure is Hard

Ngrams is typical way of preserving some structure

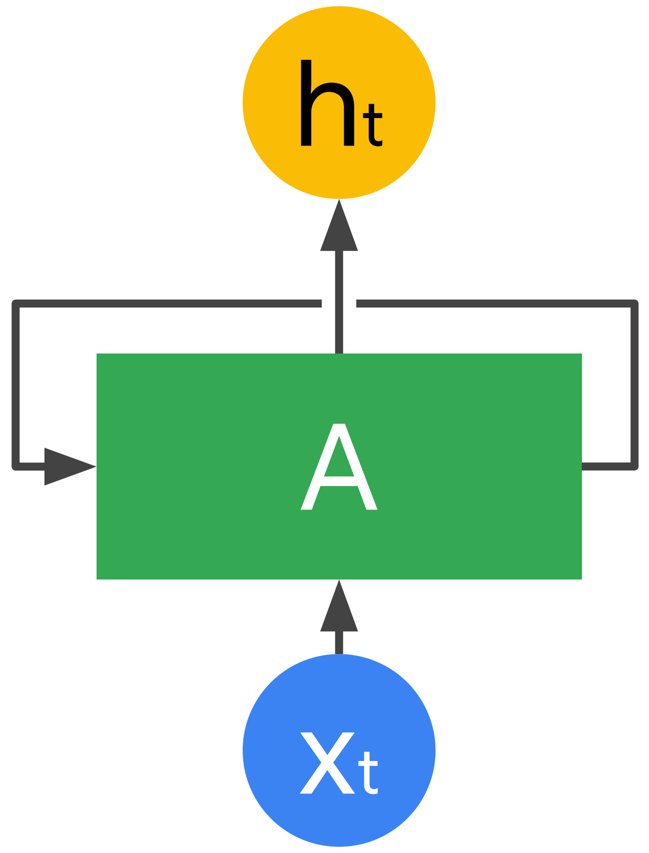


Beyond bi or tri-grams occurrences become very rare and dimensionality becomes huge (1, 10 million + features)

Big Idea of Recurrent Neural Networks



RNNs: Networks with Loops



A: a subgraph of the NN

x(t): RNN input at time *t*

h(t): RNN state at time *t*

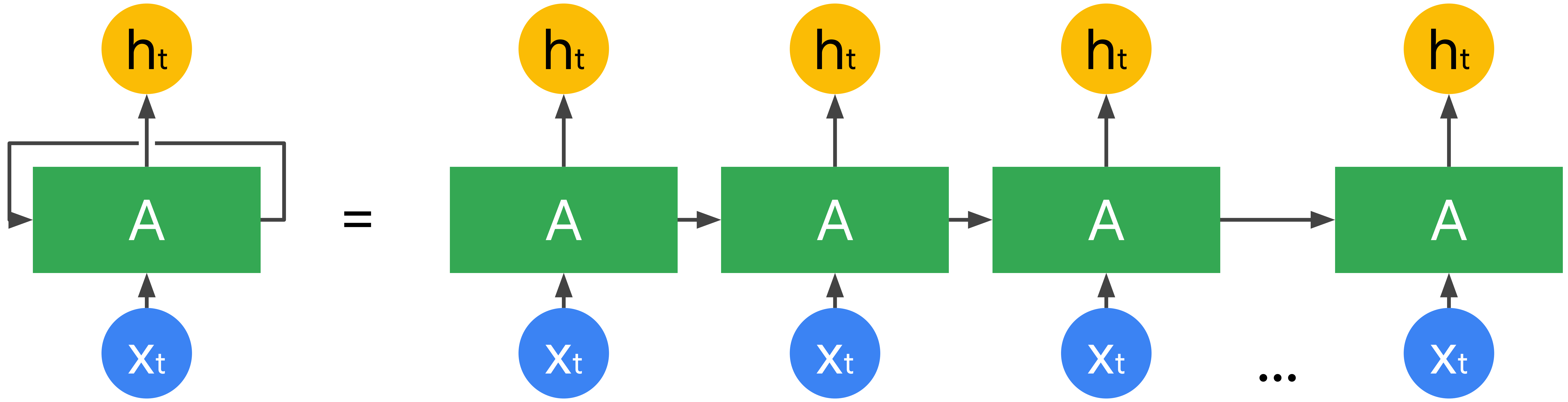
- *h(t) = (hidden state, output)*

```
for t in range(len(x)):
  h_next = A(x[t], h[t-1].hidden)
  h.append(h_next)
loss = sum([loss_fn(y) for y in h.output])
```

Google Cloud

Parts of this slide and the following slides are building on ideas and visualizations from
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Unrolled Recurrent Neural Networks

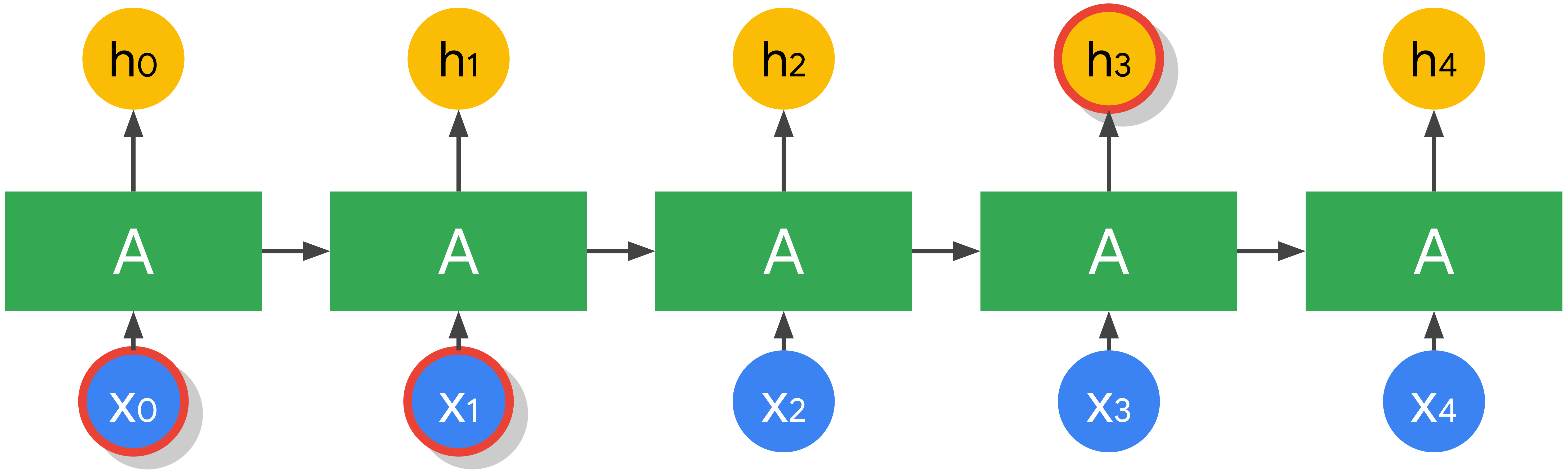


Secret sauce:

- *Tie* (share) weights of A for all *t*.
- Backprop updates same weights for all *t*.

(*sum* gradients from all *t*).

RNNs provide temporal context



I grew up in France… I speak fluent ______.

Agenda

Sequence Models

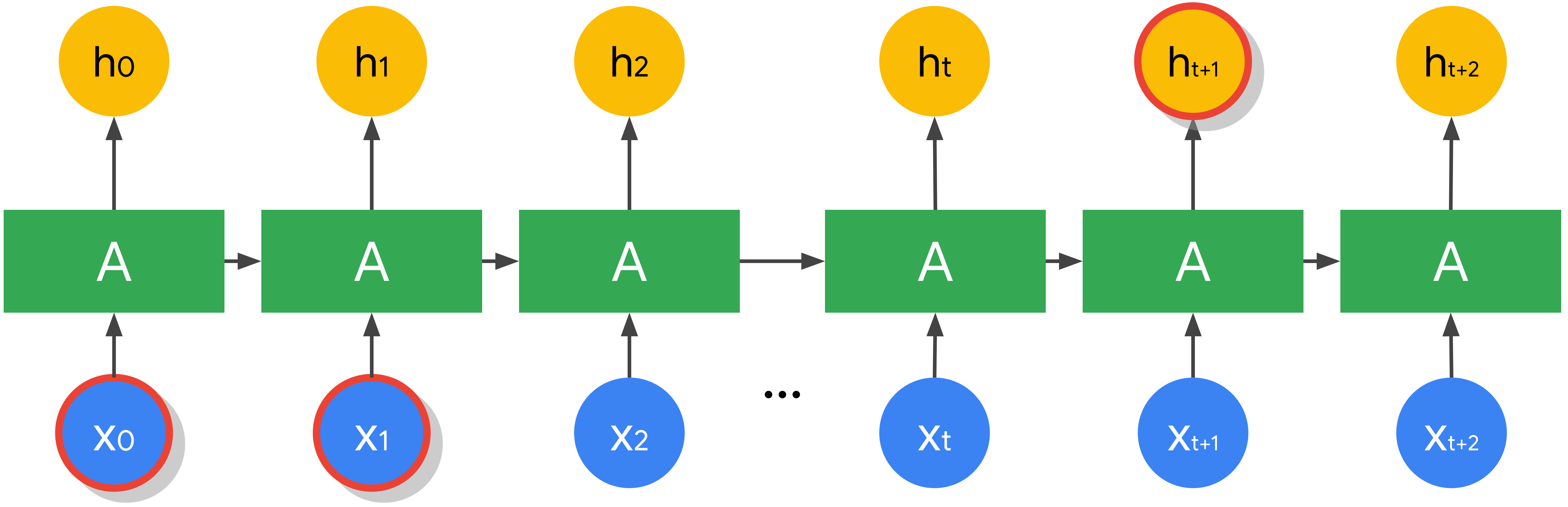
DNNs and RNNs for sequences

RNN limitations

LSTM

Applying LSTM to Time Series Data

Problems with Long-Term RNNs



- Problem 1: Gradients exploding
- Problem 2: Gradients vanishing[1]

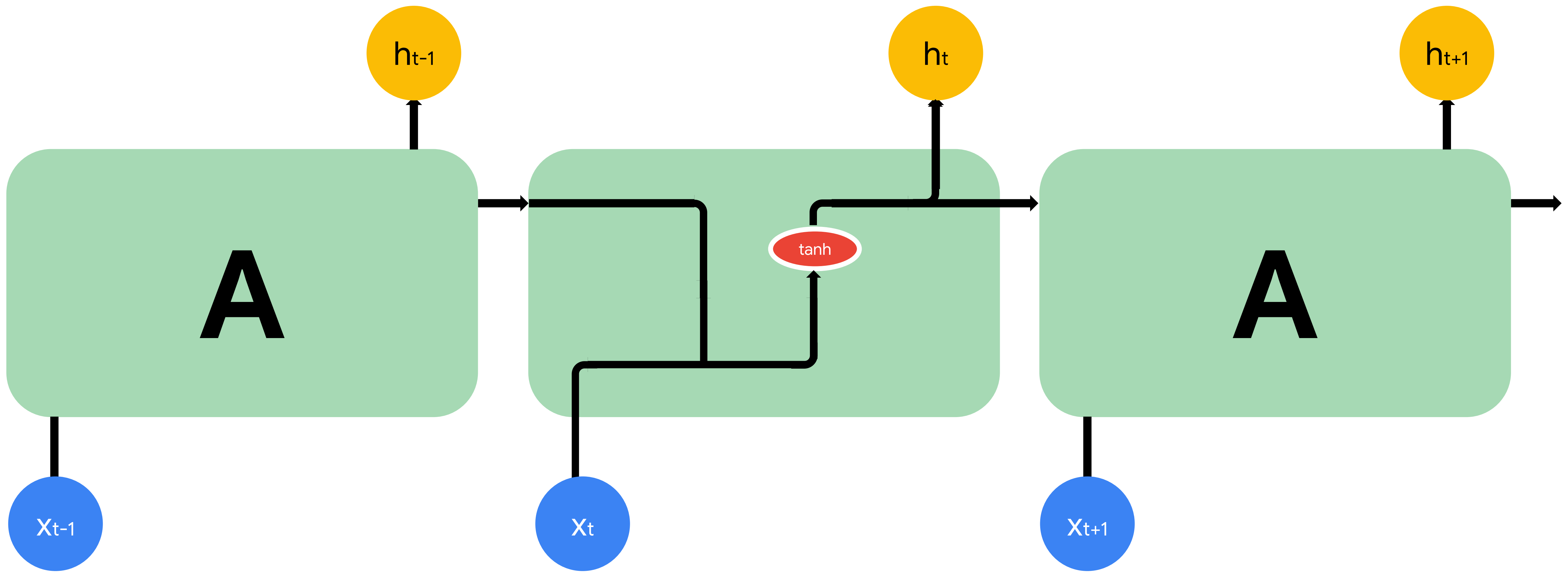
Google Cloud

1. On the difficulty of training recurrent neural networks

Agenda

Vanishing Gradients - Two Weird Tricks

Standard RNN:

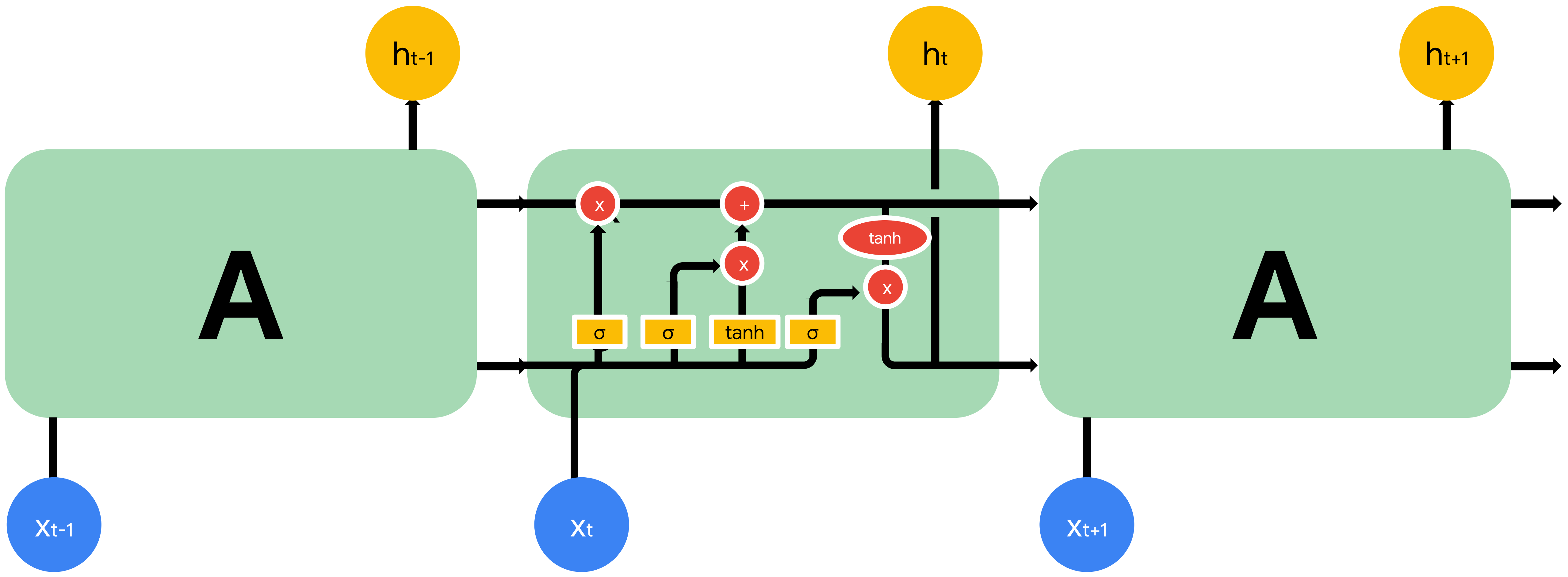


Google Cloud

Vanishing Gradients - Two Weird Tricks

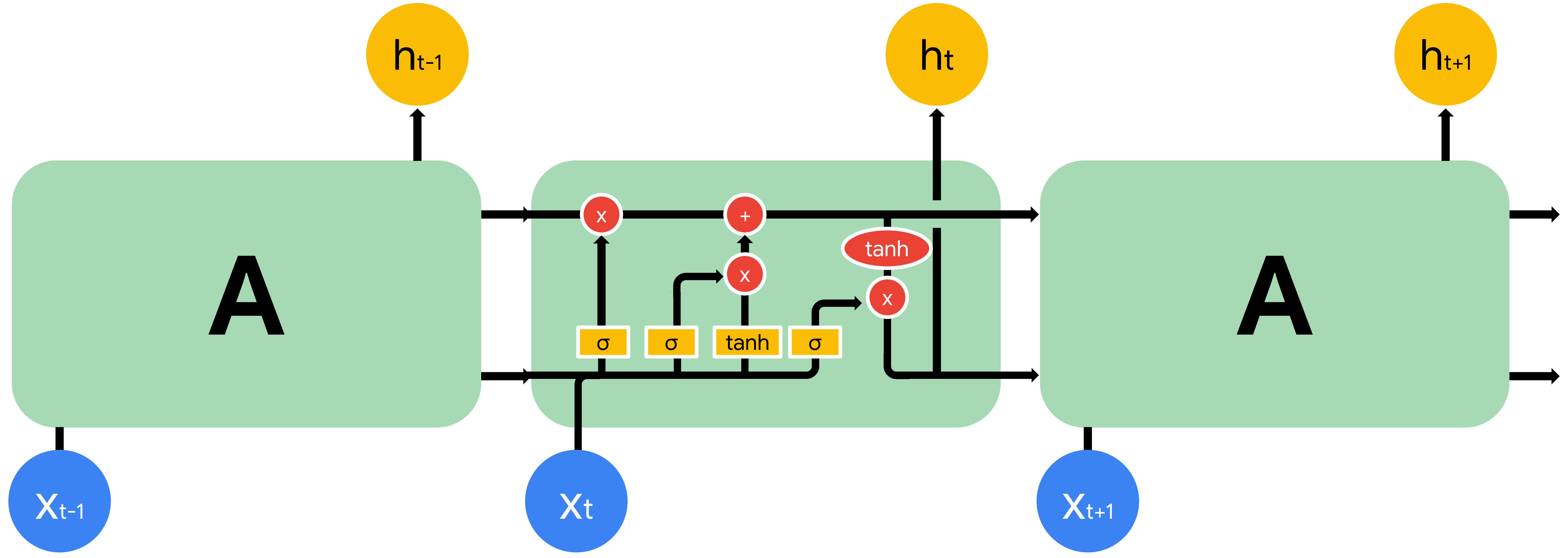
- **LSTM**: “magic” solution to the vanishing gradient problem
- Trick #1: Memory cell carried over time
- Trick #2: Gates that **learn** to manage the memory

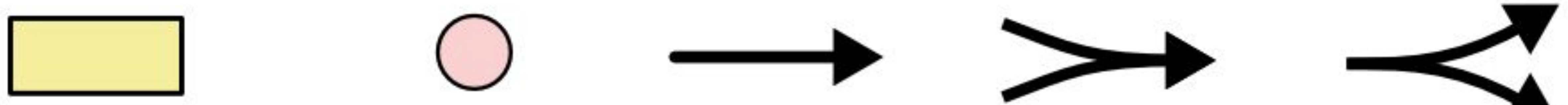
Long Short Term Memory Networks (LSTM)



Google Cloud

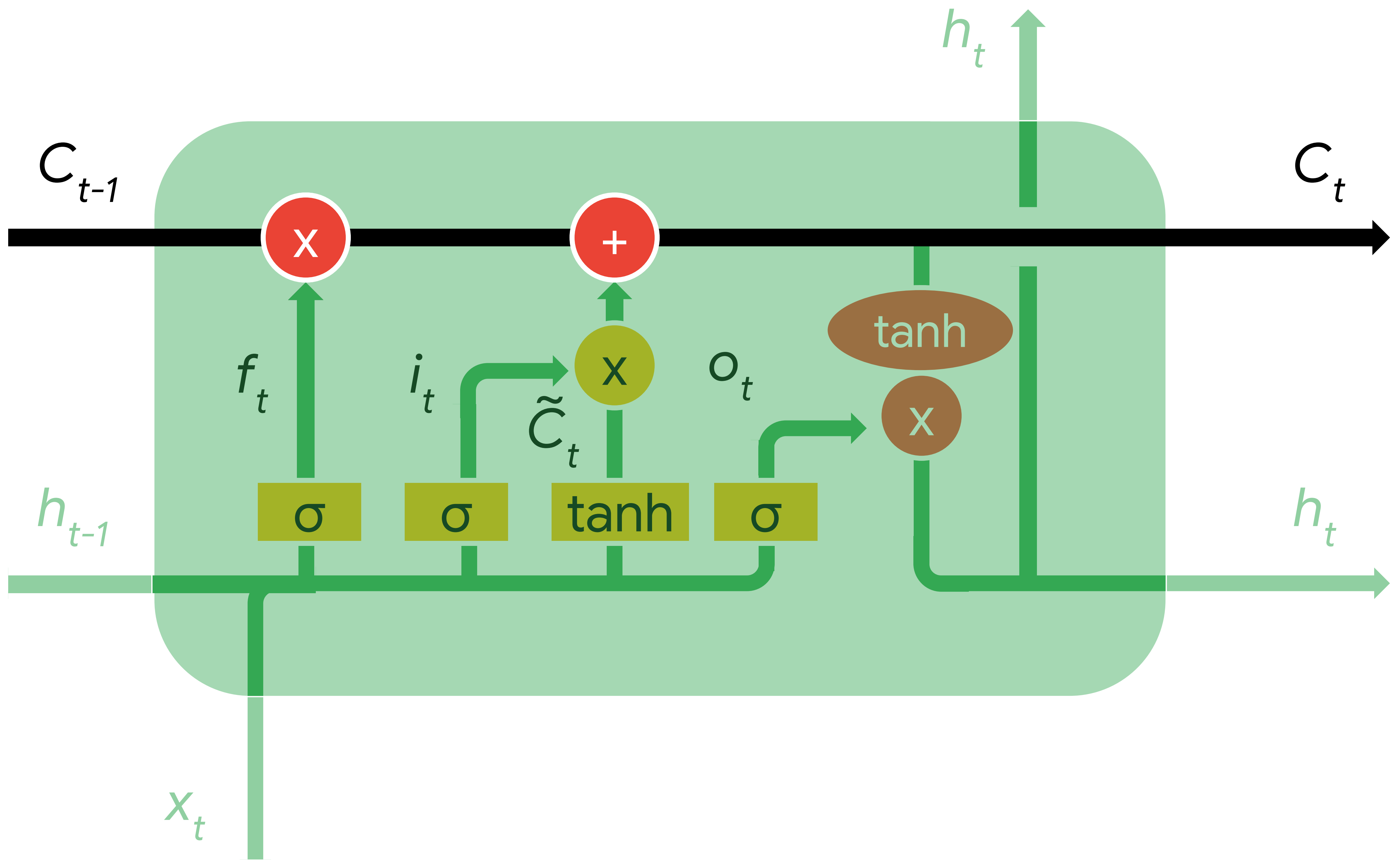
Long Short Term Memory Networks (LSTM)





Google C

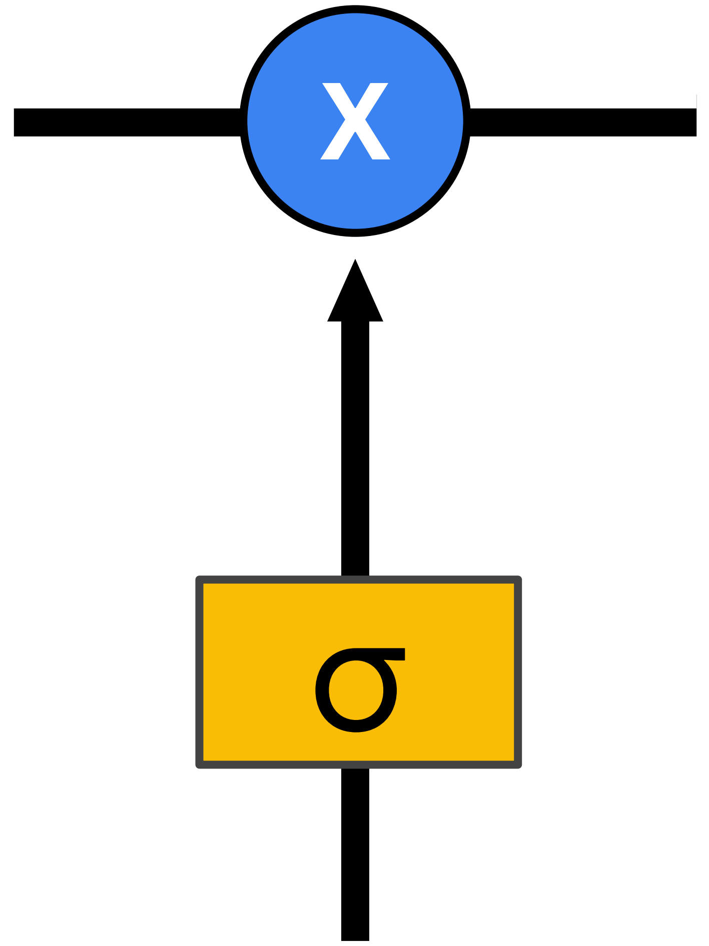
LSTM - Cell State



- “Conveyer belt”
- LSTM can “add” or “remove” information to cell state via *gates*.

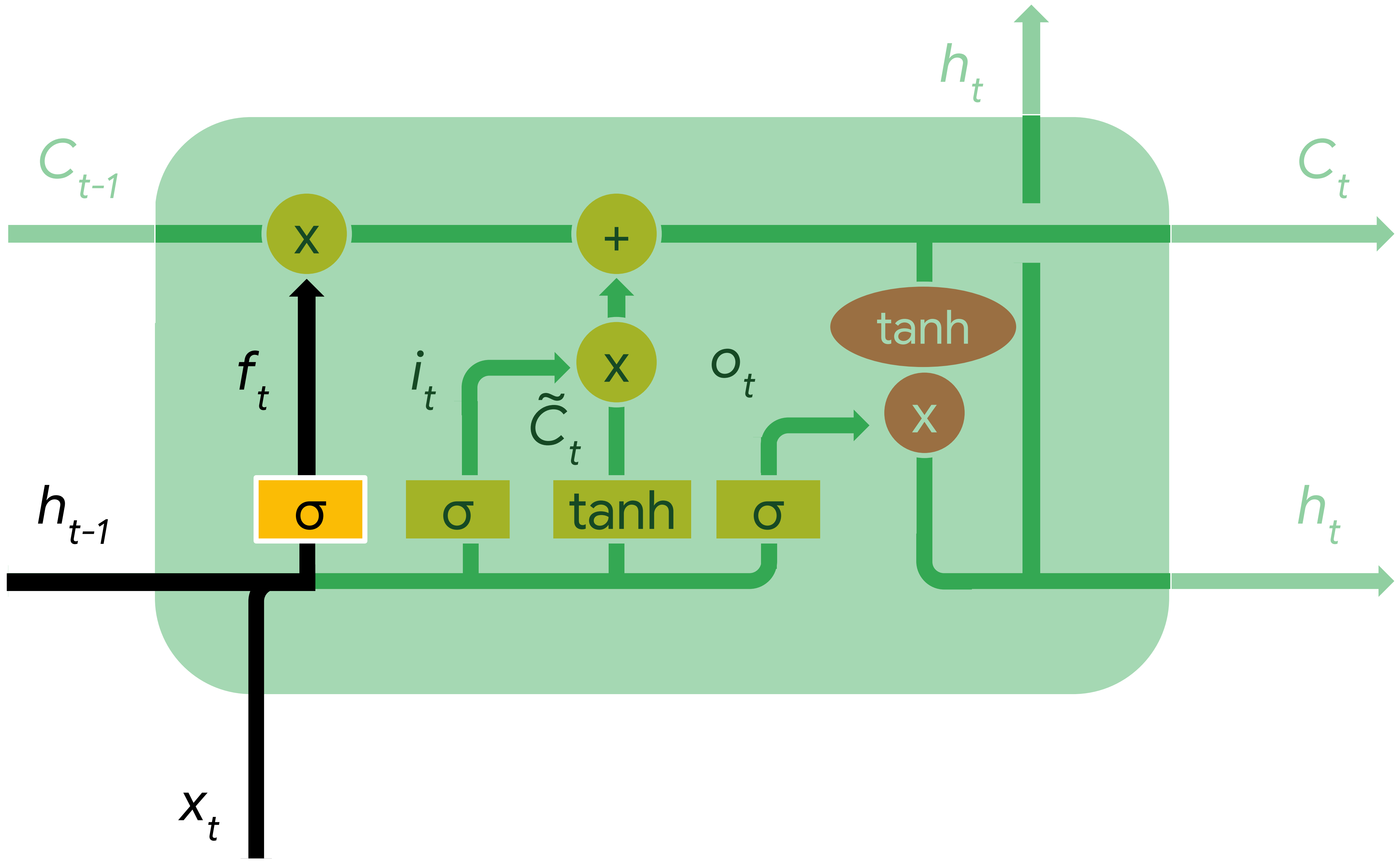
Google Cloud

Gates: Optionally Let Information Through



- Elementwise sigmoid and elementwise multiplication
- Differentiable: trainable
- $\sigma(\cdot) \in [0, 1]$

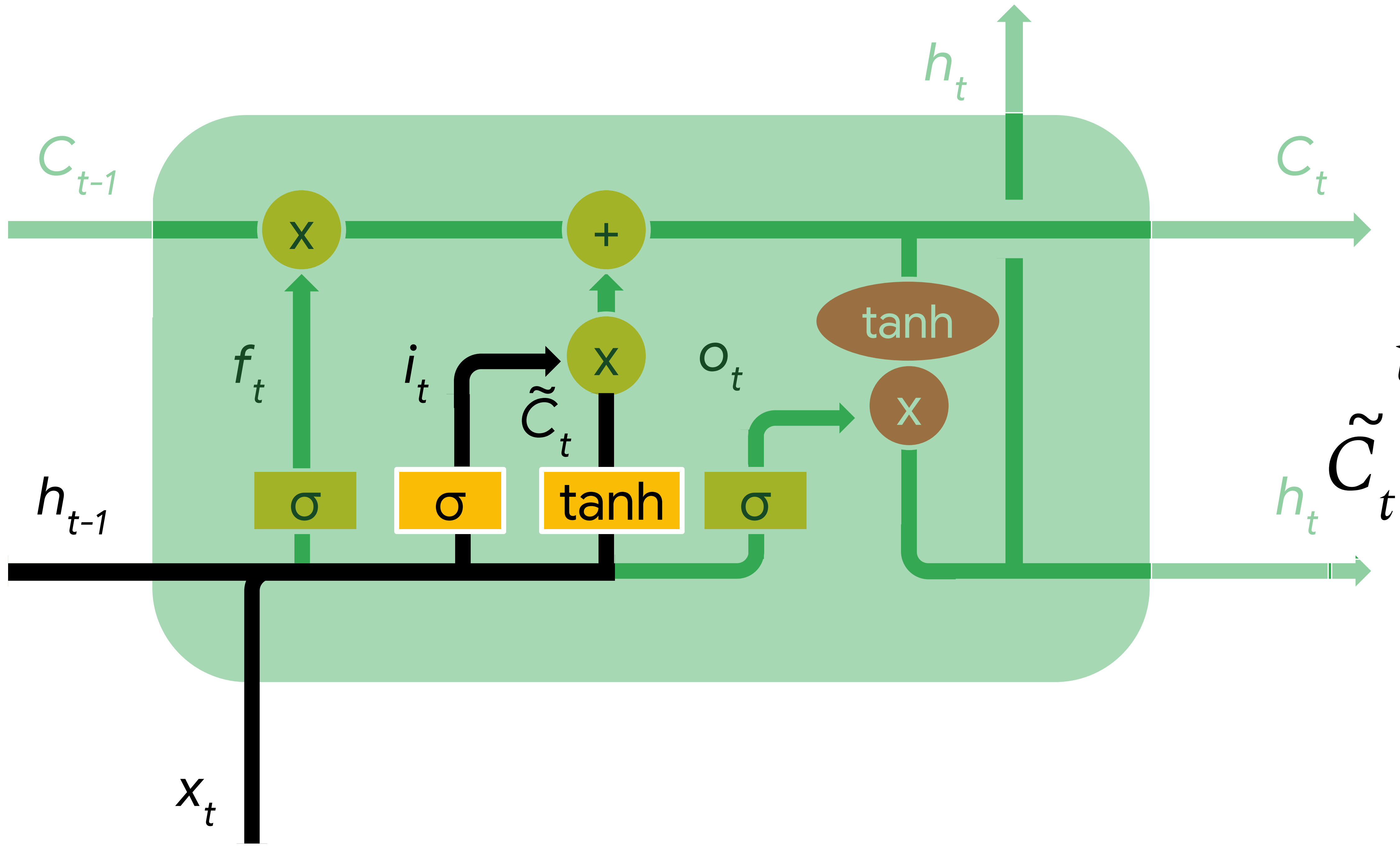
Forget Gate: What were we talking about?



$$f_t = \sigma\,(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Google Cloud

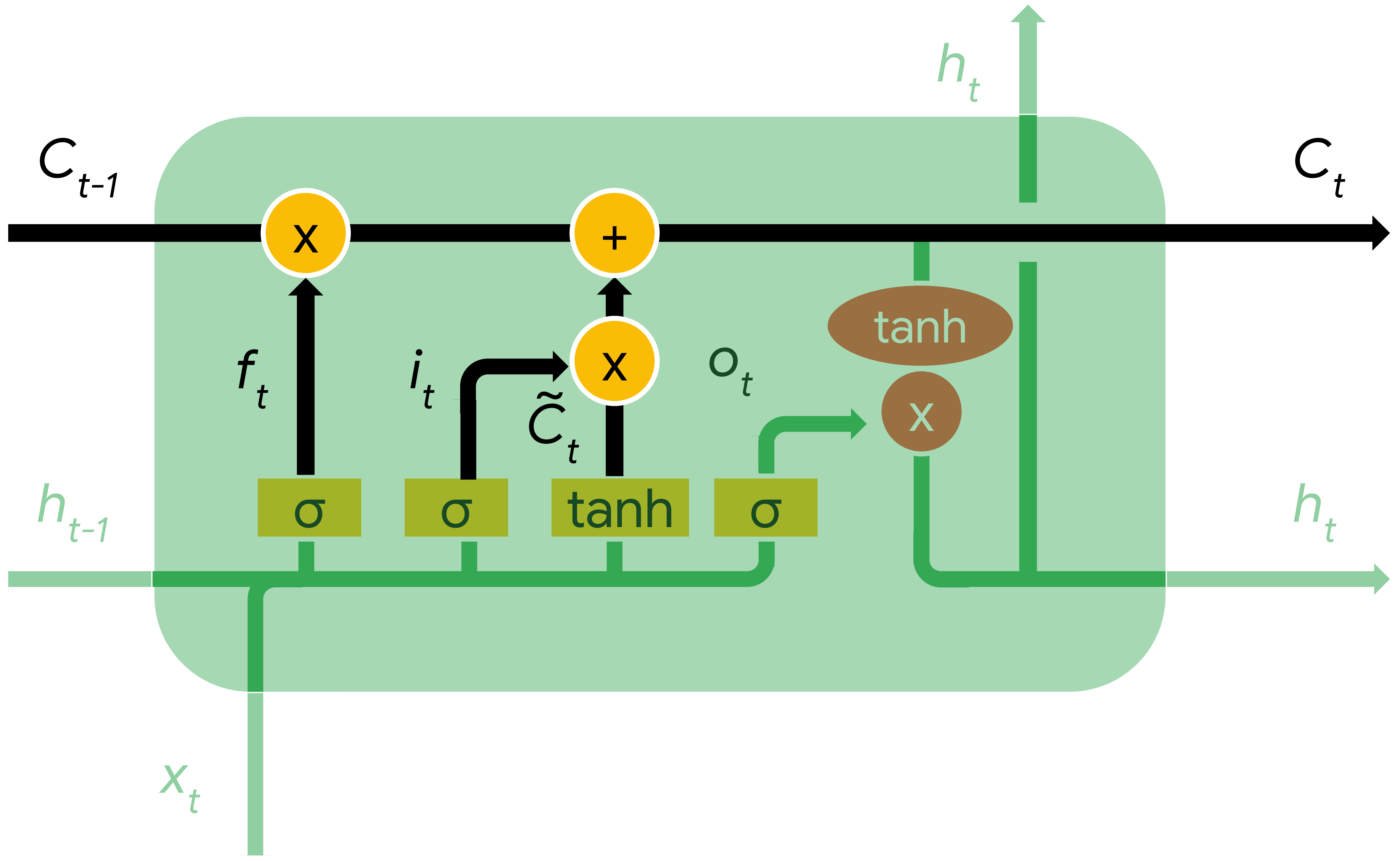
Input Gate and Candidate State



$$i_t = \sigma\,(W_i\,.\,[h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = tanh\,(W_c.\,[h_{t-1}, x_t] + b_C)$$

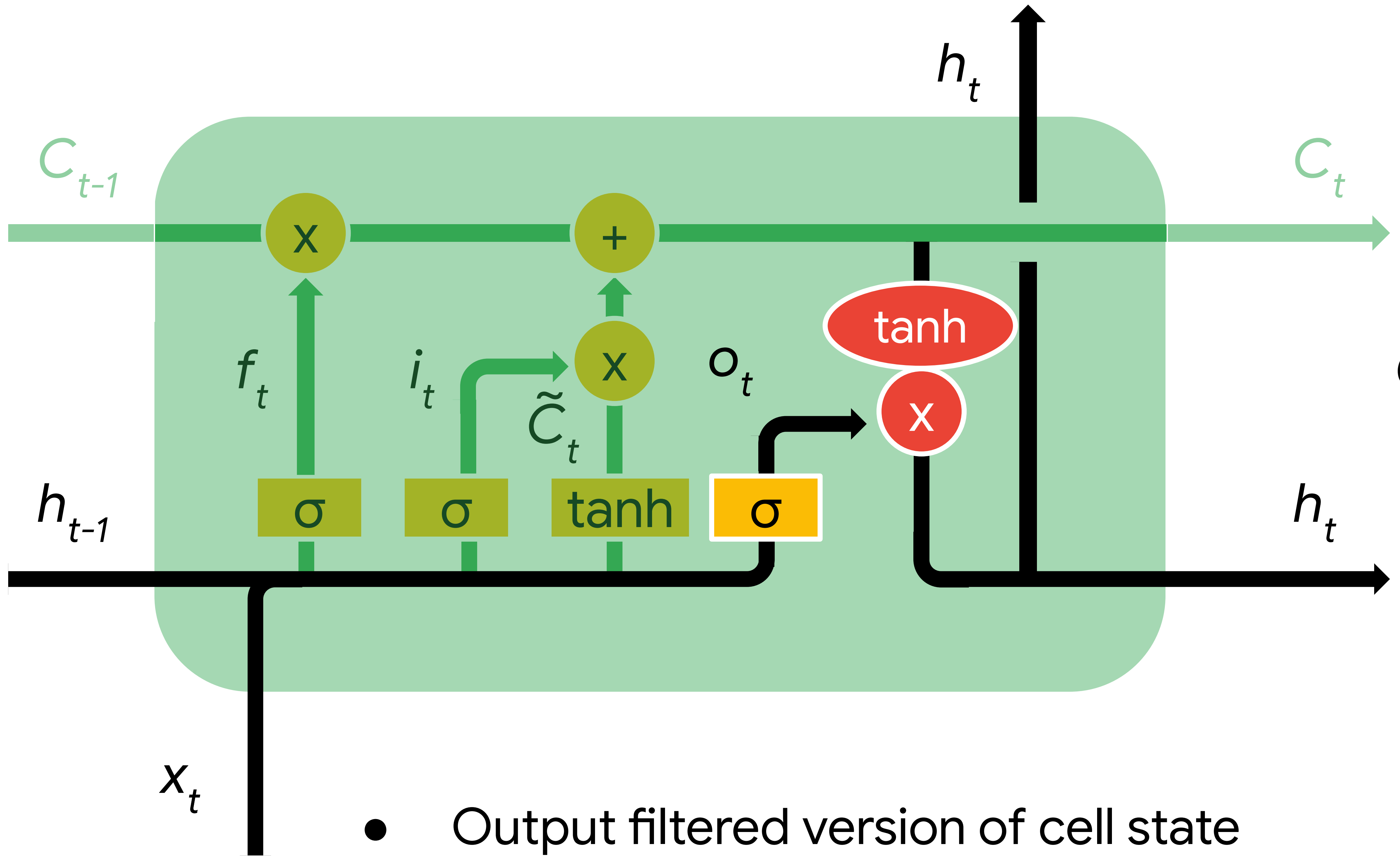
Google Cloud

Update the Cell State



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output Gate

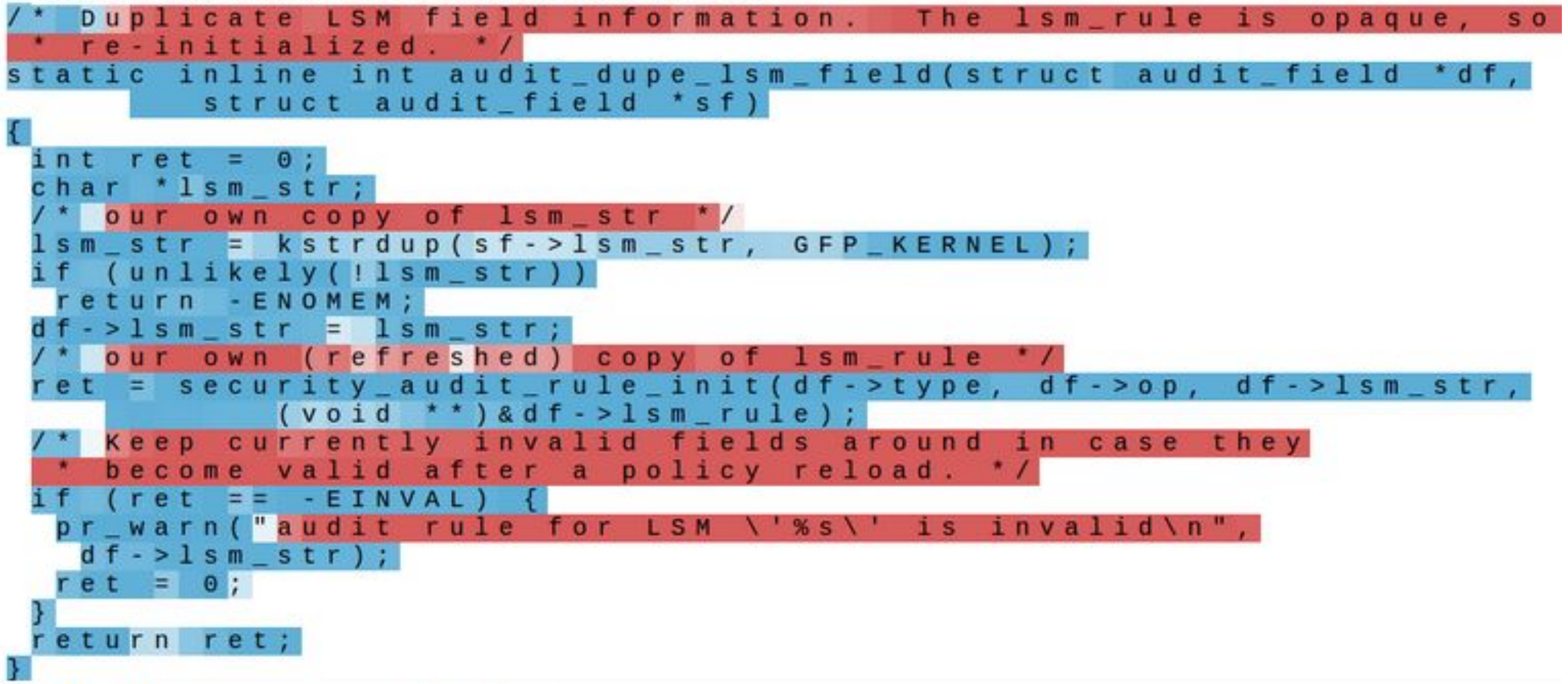


$$o_t = \sigma\,(W_o\,[\,h_{t-1}, x_t\,] + b_o)$$
$$h_t = o_t * tanh\,(C_t)$$

- Output filtered version of cell state
- $\tanh(\cdot) \in [-1, 1]$

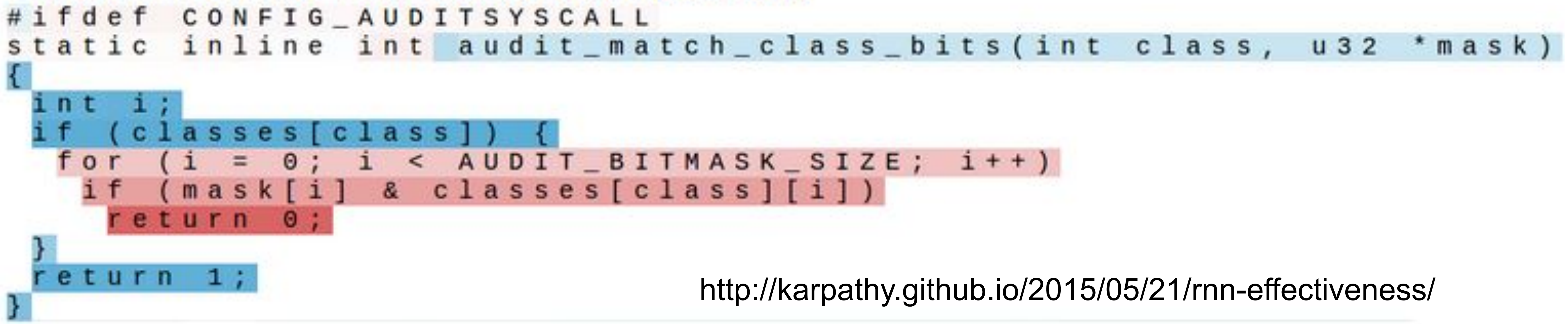
Google Cloud

Cell that turns on inside comments and quotes:



```
/* Duplicate LSM field information.  The lsm_rule is opaque, so
 * re-initialized.  */
static inline int audit_dupe_lsm_field(struct audit_field *df,
                                       struct audit_field *sf)
{
 int ret = 0;
 char *lsm_str;
 /* our own copy of lsm_str */
 lsm_str = kstrdup(sf->lsm_str, GFP_KERNEL);
 if (unlikely(!lsm_str))
  return -ENOMEM;
 df->lsm_str = lsm_str;
 /* our own (refreshed) copy of lsm_rule */
 ret = security_audit_rule_init(df->type, df->op, df->lsm_str,
              (void **)&df->lsm_rule);
 /* Keep currently invalid fields around in case they
  * become valid after a policy reload. */
 if (ret == -EINVAL) {
  pr_warn("audit rule for LSM \'%s\' is invalid\n",
   df->lsm_str);
  ret = 0;
 }
 return ret;
}
```

Cell that is sensitive to the depth of an expression:



```
#ifdef CONFIG_AUDITSYSCALL
static inline int audit_match_class_bits(int class, u32 *mask)
{
 int i;
 if (classes[class]) {
  for (i = 0; i < AUDIT_BITMASK_SIZE; i++)
   if (mask[i] & classes[class][i])
    return 0;
 }
 return 1;
}
```

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Google Cloud

Google Cloud

Apply LSTM to Time Series data

Agenda

Sequence Models

DNNs and RNNs for sequences

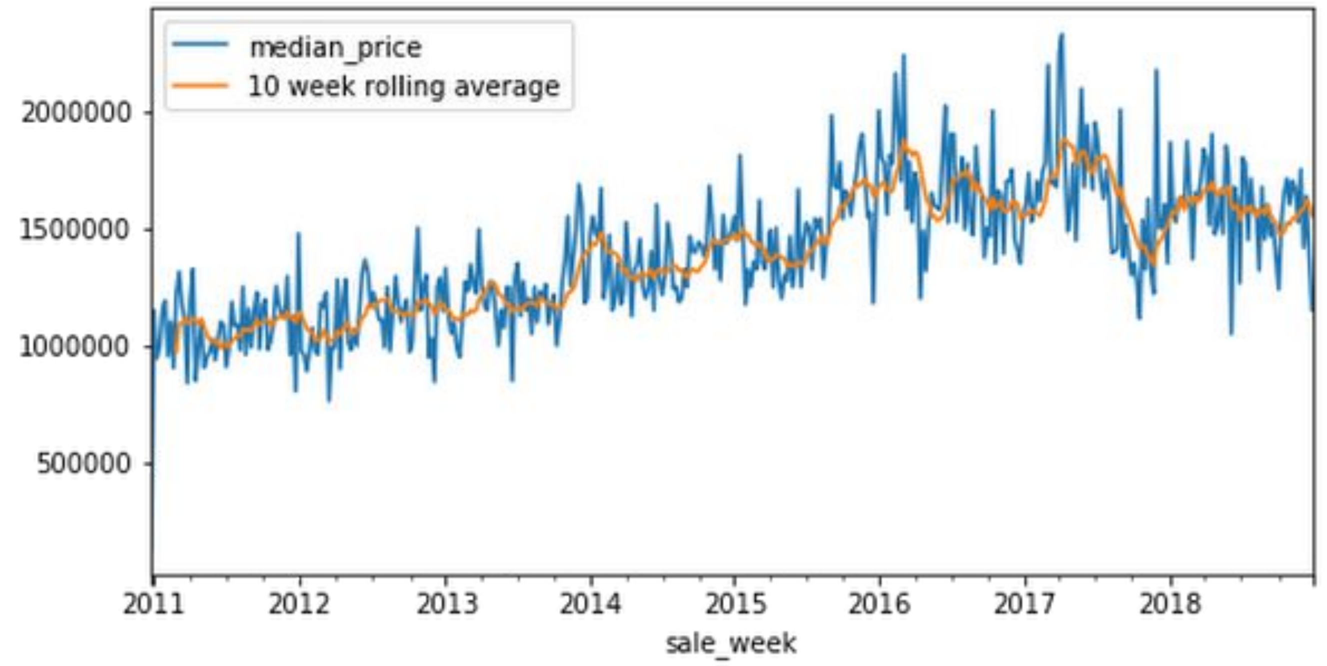
RNN limitations

LSTM

Applying LSTM to Time Series Data

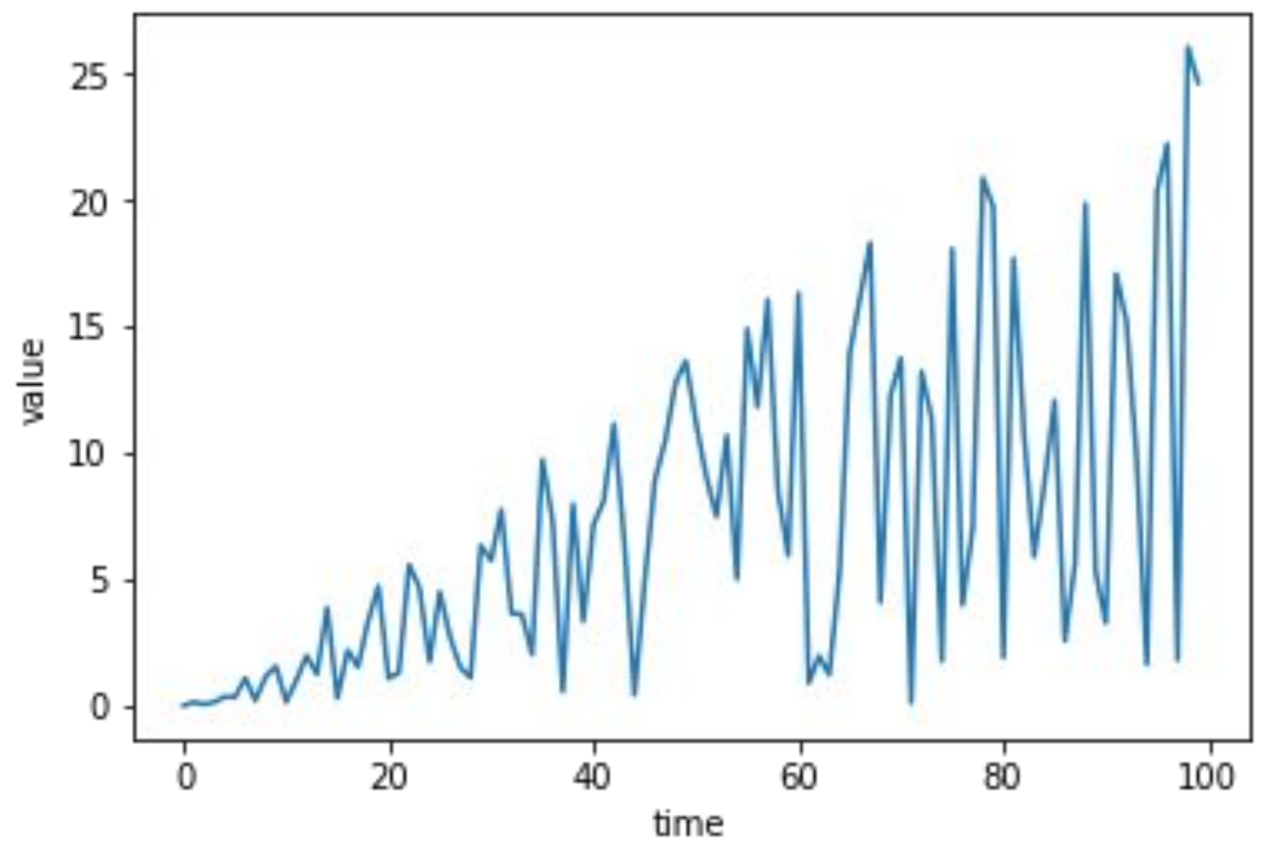
Time-series problems are ubiquitous

- How many items will be sold next week? Next month? Next year?
- What is the likelihood there will be a major earthquake (M>6.7) on the Hayward fault in the next 26 years?
- Is this a fraudulent transaction?



Google Cloud

Training a time-series model can require significant feature engineering

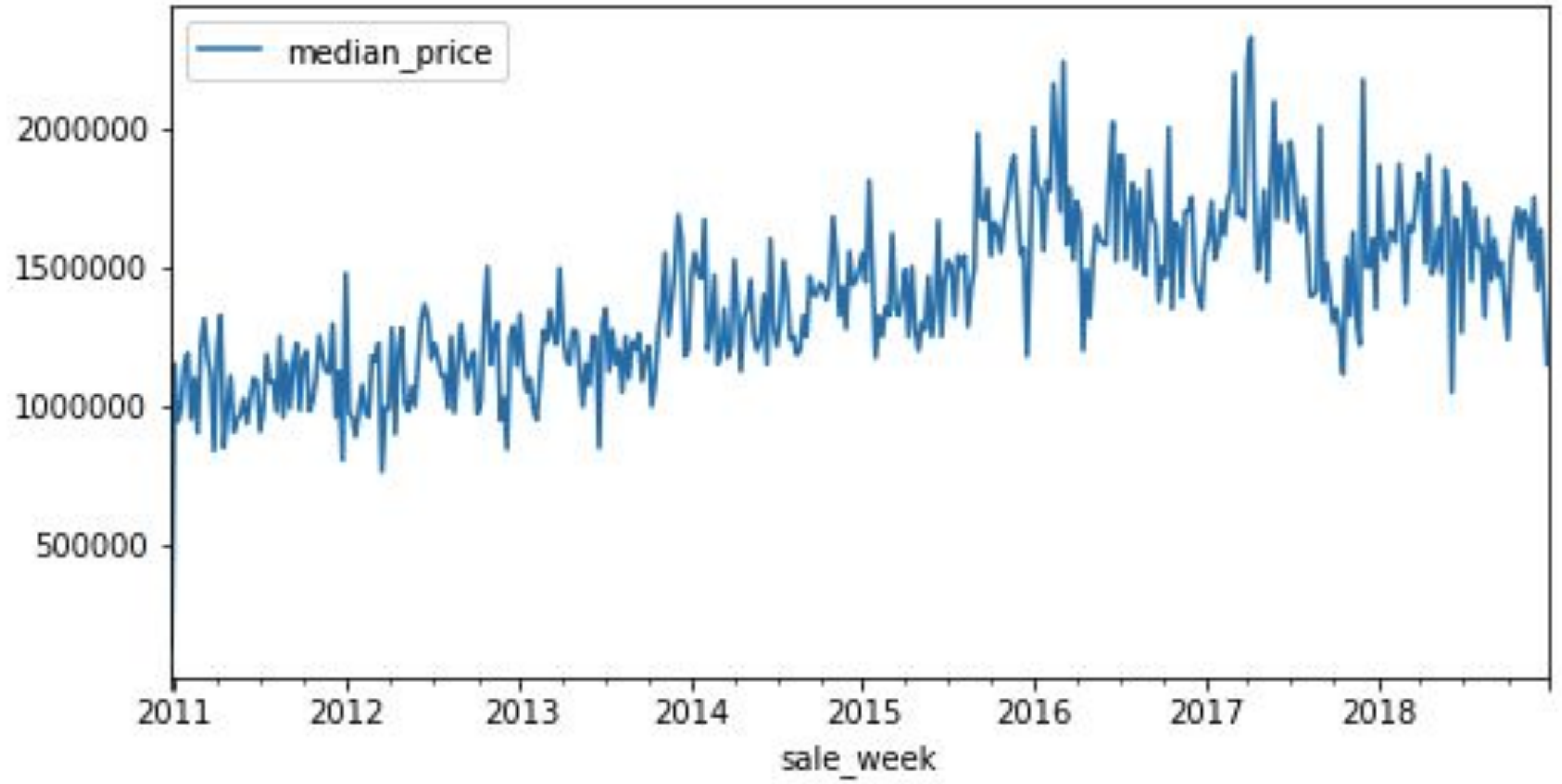


→ Feature (and label!) engineering →

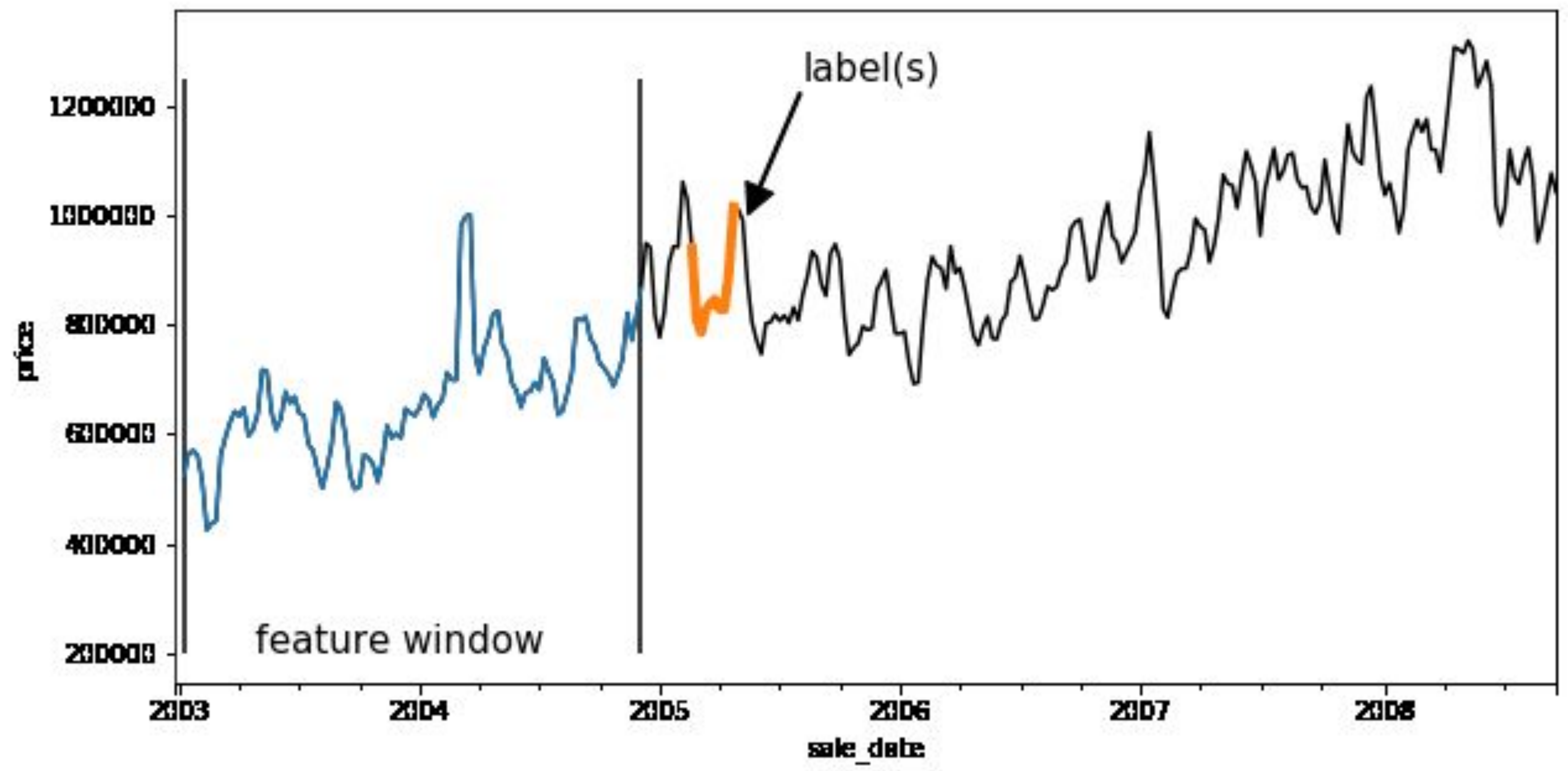
feature1	feature2	...	label
5	20.51	1	1.1
0.8	-0.51	2.9	-0.82
...	...	...	...

NYC real estate data

```
sale_week
2010-12-26     134640
2011-01-02    1150000
2011-01-09     945000
2011-01-16     995000
2011-01-23    1150000
```



Sliding Window to create features and label



Google Cloud

Example: Create a feature table, window_size = 3, horizon = 1

datetime	value
2018-01-01 0:00:00	0.7713206433
2018-01-02 0:00:00	0.02075194936
2018-01-03 0:00:00	0.6336482349
2018-01-04 0:00:00	0.7488038825
2018-01-05 0:00:00	0.4985070123
2018-01-06 0:00:00	0.2247966455
2018-01-07 0:00:00	0.1980628648
2018-01-08 0:00:00	0.7605307122
2018-01-09 0:00:00	0.1691108366
2018-01-10 0:00:00	0.08833981417

Input table

$\longrightarrow$

pred_datetime	-3_steps	-2_steps	-1_steps	label
2018-01-04 0:00:00	0.7713206433	0.02075194936	0.6336482349	0.7488038825
2018-01-05 0:00:00	0.02075194936	0.6336482349	0.7488038825	0.4985070123
2018-01-06 0:00:00	0.6336482349	0.7488038825	0.4985070123	0.2247966455
2018-01-07 0:00:00	0.7488038825	0.4985070123	0.2247966455	0.1980628648
2018-01-08 0:00:00	0.4985070123	0.2247966455	0.1980628648	0.7605307122
2018-01-09 0:00:00	0.2247966455	0.1980628648	0.7605307122	0.1691108366
2018-01-10 0:00:00	0.1980628648	0.7605307122	0.1691108366	0.08833981417

Features, label

Create the features and label

```
import time_series

WINDOW_SIZE = 52 * 1
HORIZON = 4*6

df = time_series.create_rolling_features_label(sales,
                                               window_size=WINDOW_SIZE,
                                               pred_offset=HORIZON)
```

pred_date	-52_steps	-51_steps	...	-2_steps	-1_steps	label
2012-06-03	134640	1150000	...	960000	1125000	805000
2012-06-10	1150000	945000	...	1125000	805000	1476462
2012-06-17	945000	995000	...	805000	1476462	975000
2012-06-24	995000	1150000	...	1476462	975000	960000
2012-07-01	1150000	1190000	...	975000	960000	890000

https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/blogs/gcp_forecasting/time_series.py

Google Cloud

Date features can provide performance lift

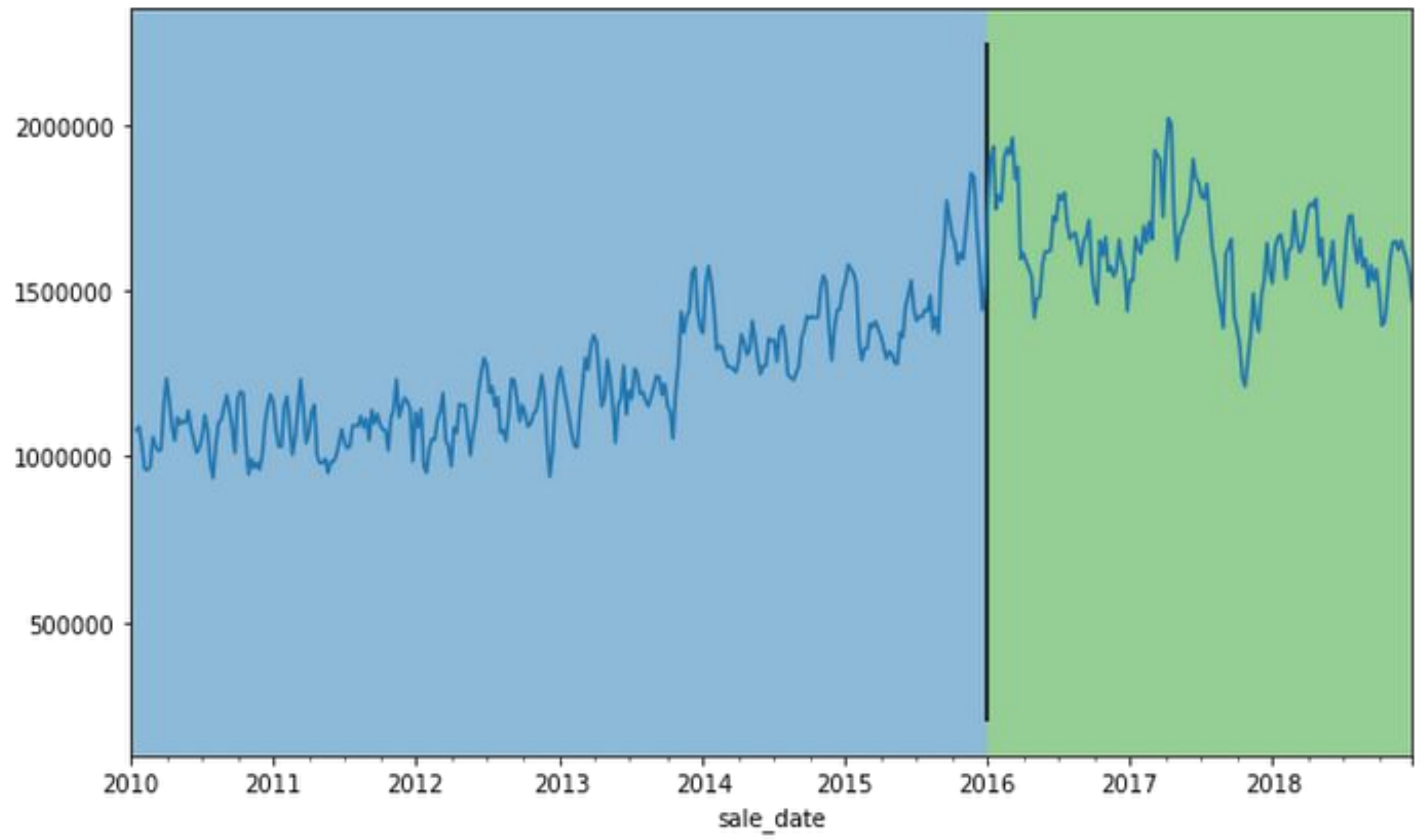
```
dates = df.index
df = time_series.add_date_features(df, dates)
```

doy	dom	month	year	n_holidays
155	3	6	2012	0
162	10	6	2012	0
169	17	6	2012	0
176	24	6	2012	0
183	1	7	2012	1

Train/test set: split temporally

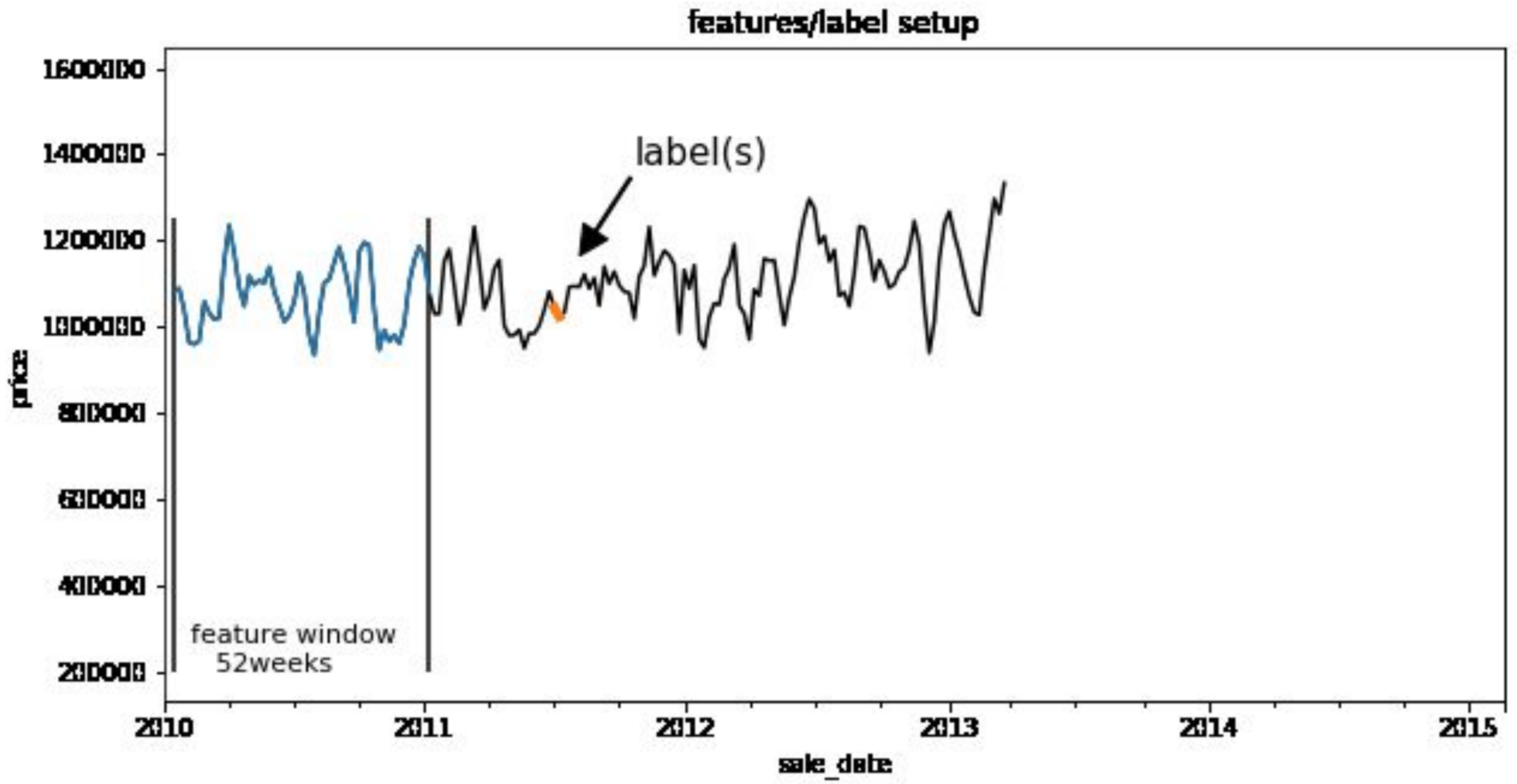
```
# Features, label.
X = df.drop('label', axis=1)
y = df['label']

# Train/test split. Splitting on time.
train_ix = time_series.is_between_dates(y.index,
                                  end='2015-12-30')
test_ix = time_series.is_between_dates(y.index,
                               start='2015-12-30',
                                 end='2018-12-30')
X_train, y_train = X.iloc[train_ix], y.iloc[train_ix]
```



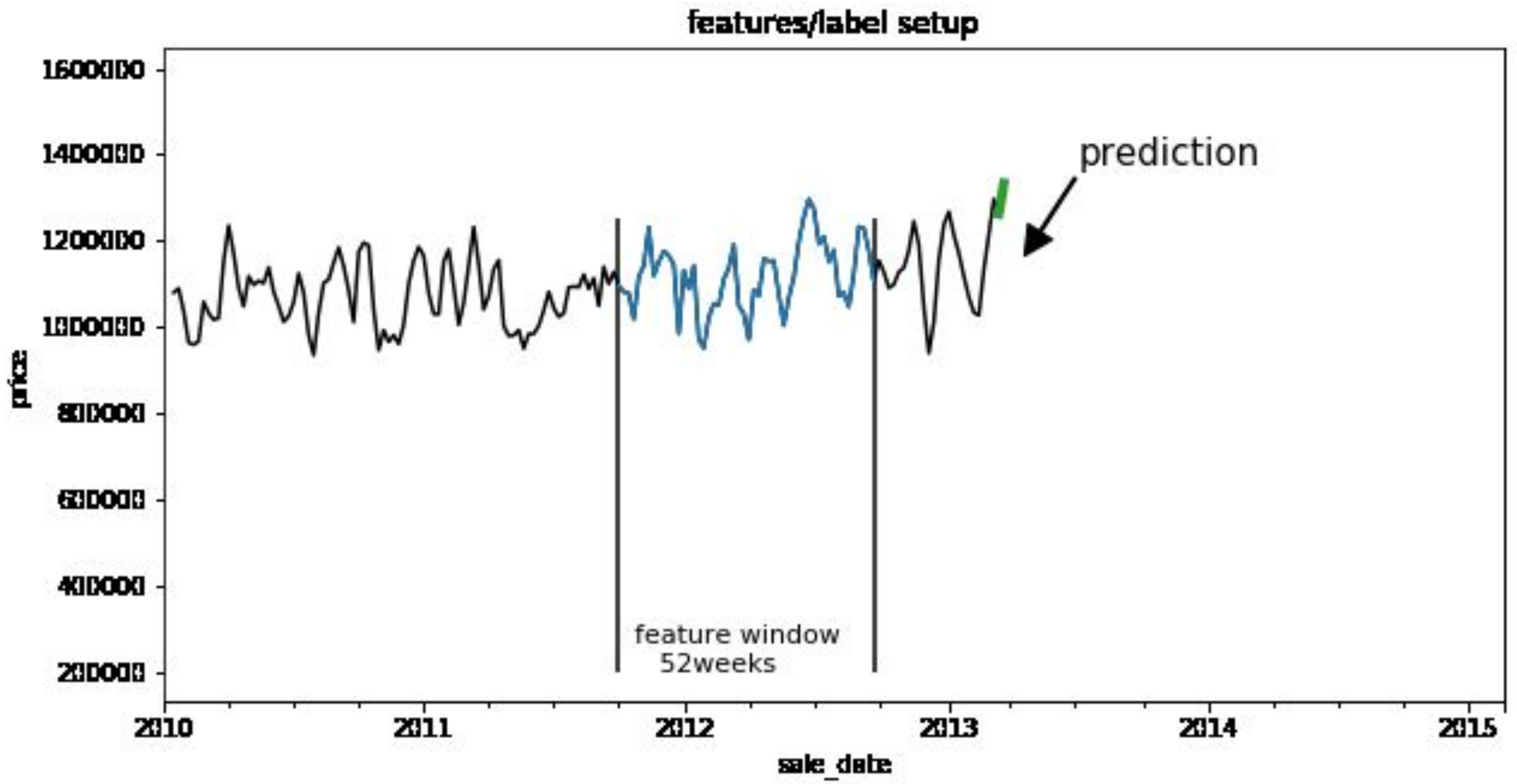
Google Cloud

Training



Google Cloud

Predicting



Google Cloud

Baseline model

Simple model: look at all the history and predicts the next point to be the average of the last 20 observations.

```
import time_series

baseline_global_metrics =
time_series.Metrics(df_baseline.pred,
df_baseline.label)
baseline_global_metrics.report("Global Baseline Model")

"""
Global Baseline Model results
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
RMSE: 376544.261
MAE: 316352.450
MALR: 0.207
"""
```

My bidirectional LSTM successfully trained
It doesn’t beat my baseline model

Machine learn: Random Forest

```
# Train model.
cl = RandomForestRegressor(n_estimators=500,
max_features='sqrt', random_state=10, criterion='mse')
cl.fit(X_train, y_train)
pred = cl.predict(X_test)

random_forest_metrics = time_series.Metrics(y_test,
                                            pred)
random_forest_metrics.report("Forest Model")
"""
Forest Model results
~~~~~~~~~~~~~~~~~~~~
RMSE: 259388.403
MAE: 202647.688
MALR: 0.125
"""
```

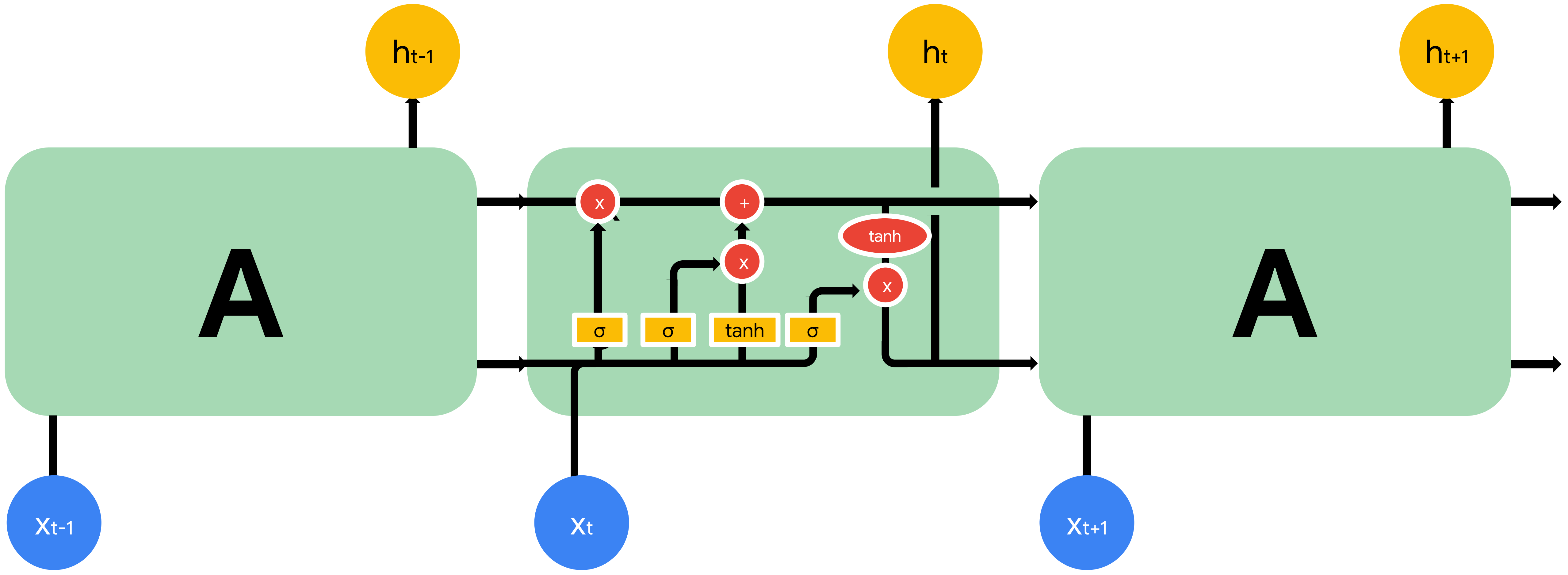
it's working

Google Cloud

Machine learn: using LSTM

Instead of the simple Random Forest model, we can also build an LSTM model on the same prepared dataset to attempt to increase model performance.

See the coming Lab for more details.



Lab

Use LSTM framework to set up a simple Buy/Sell trading model

Lab Objectives

-
-

Screencast