CSCI S-89C Deep Reinforcement Learning

Harvard Summer School

Dmitry Kurochkin

Summer 2020
Lecture 12

Contents

Contents

Quiz 12

Question 1

4 / 4 pts

Please consider the following Convolutional Neural Network (CNN):

```
model = models.Sequential()
model.add(layers.Conv2D(10, (4, 4), activation='relu',
                        input_shape=(28, 28, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(20, (5, 5), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(30, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```

How many parameters does the first convolutional layer of this CNN have?

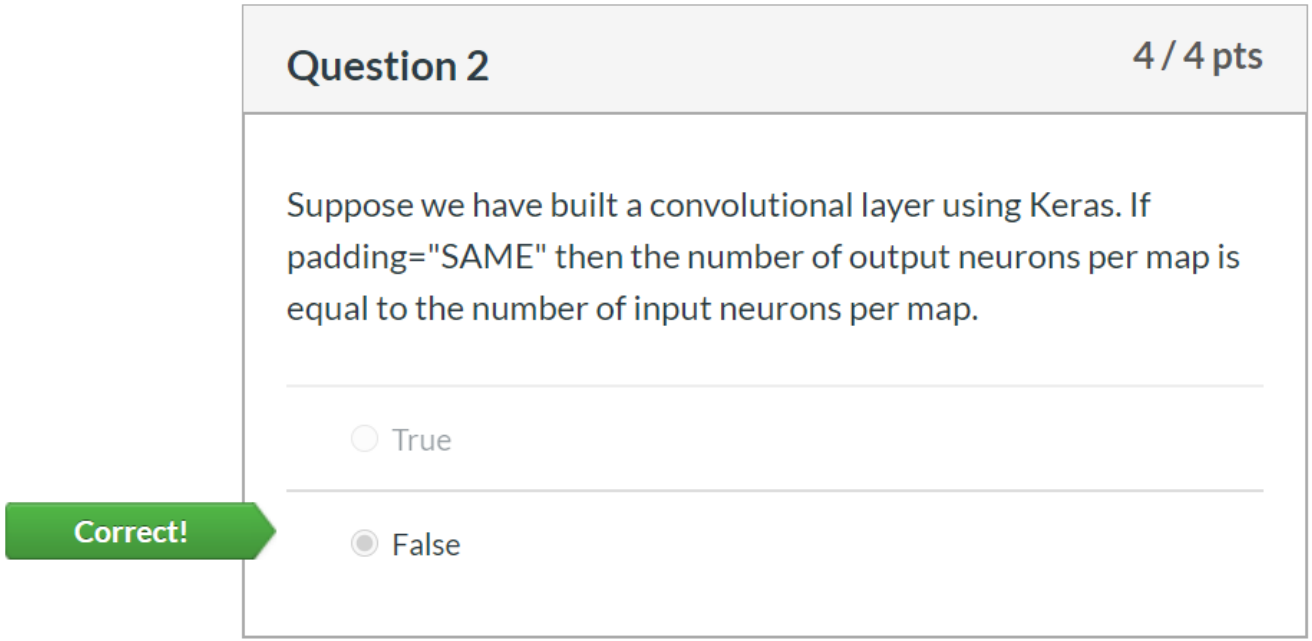
Correct!

490

Correct Answers

490 (with margin: 0)

Quiz 12


Question 2
4 / 4 pts
Suppose we have built a convolutional layer using Keras. If padding="SAME" then the number of output neurons per map is equal to the number of input neurons per map.
True
Correct!
False

Quiz 12

Question 3 4 / 4 pts

Suppose we have built a convolutional layer using Keras. If strides=1 then the number of output neurons per map is equal to the number of input neurons per map.

- ○ True
- ◉ False

Correct!

Quiz 12

Question 4

4 / 4 pts

Please consider the following Recurrent Neural Network (RNN):

```
model = models.Sequential()
model.add(layers.SimpleRNN(2, activation='relu', input_shape=(20,4)))
model.add(layers.Dense(1, activation='linear'))

model.summary()
```

How many parameters does the first layer of recurrent neurons of this RNN have?

Correct!

14

Correct Answers

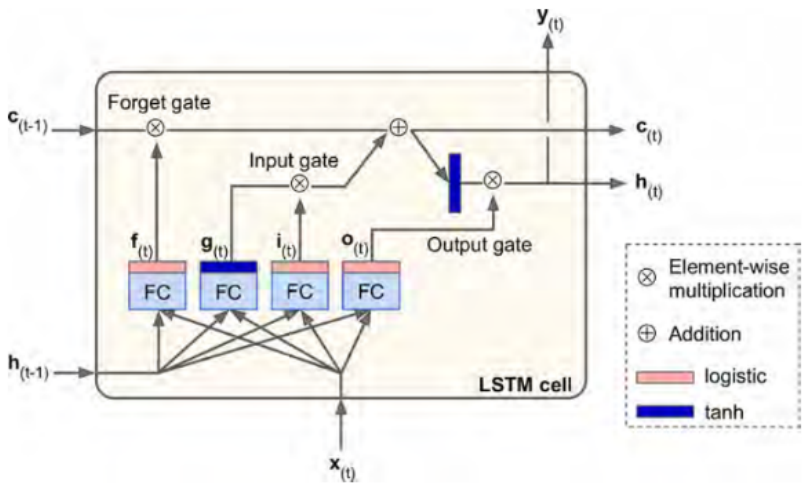
14 (with margin: 0)

Quiz 12

Question 5

4 / 4 pts

"Long-term memory" vector $\mathbf{c}_{(t)}$ and "short-term memory" vector $\mathbf{h}_{(t)}$ in Long Short-Term Memory (LSTM) Cells (please see the figure below) have same dimensions.



Source: Hands-On Machine Learning with Scikit-Learn and TensorFlow by A. Geron

Correct!

- True

Contents

Regularization Penalty

If we wish to discourage overfitting we can add a *regularization penalty*, $R(\boldsymbol{w})$, to the loss function:

$$L(\boldsymbol{w}) = \frac{1}{2} \sum_{m=1}^{M} (\hat{y}_m - y_m)^2 + \lambda R(\boldsymbol{w}).$$

Regularization Penalty

If we wish to discourage overfitting we can add a *regularization penalty*, $R(\boldsymbol{w})$, to the loss function:

$$L(\boldsymbol{w}) = \frac{1}{2}\sum_{m=1}^{M}(\hat{y}_m - y_m)^2 + \lambda R(\boldsymbol{w}).$$

The most common regularization terms are:

- L1-norm (similar to Lasso regression):

$$R(\boldsymbol{w}) \doteq \sum_{\ell}\sum_{i,j}|w_{ij}^{(\ell)}|$$

- L2-norm (similar to Ridge regression):

$$R(\boldsymbol{w}) \doteq \sum_{\ell}\sum_{i,j}\left(w_{ij}^{(\ell)}\right)^2$$

Regularization Penalty

Keras:

L1 Regularization

```
from keras import regularizers
lam = 0.0001

model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l1(lam),
                       activation='relu', input_shape=(X_train.shape[1],)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l1(lam),
                       activation='relu'))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l1(lam),
                       activation='relu'))
#model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(2, activation='softmax'))

model.summary()
```

Regularization Penalty

Keras:

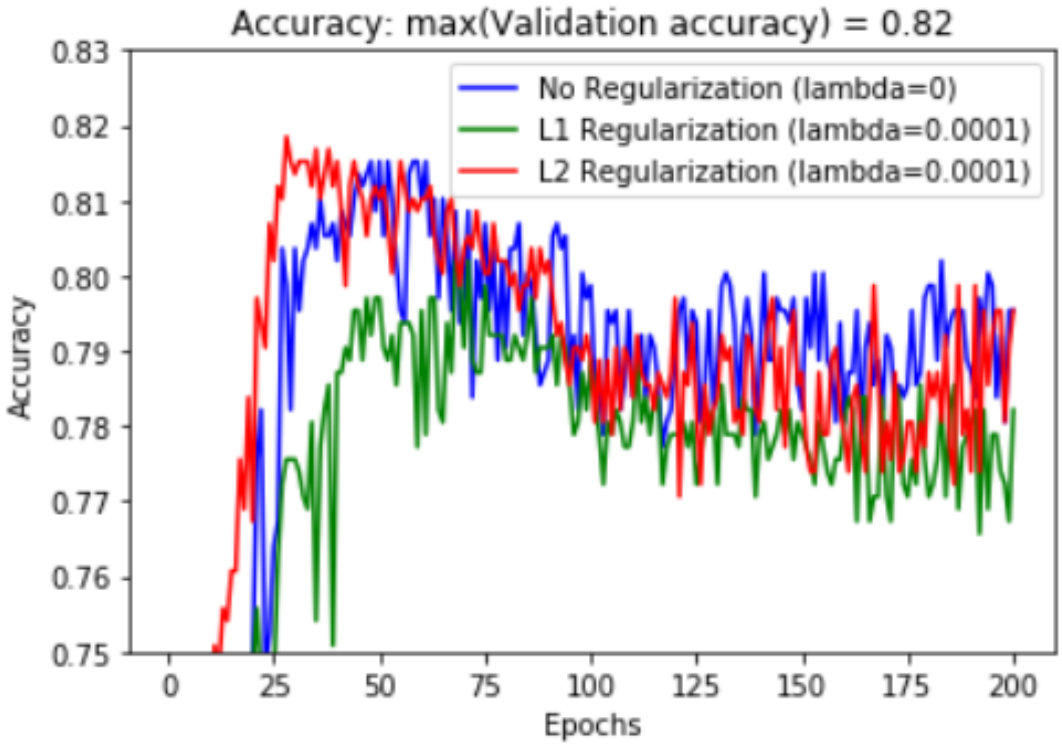
L2 Regularization

```
from keras import regularizers
lam = 0.0001
model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(lam),
                       activation='relu', input_shape=(X_train.shape[1],)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(lam),
                       activation='relu'))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(lam),
                       activation='relu'))
#model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(2,
                       activation='softmax'))

model.summary()
```

Regularization Penalty

Keras:

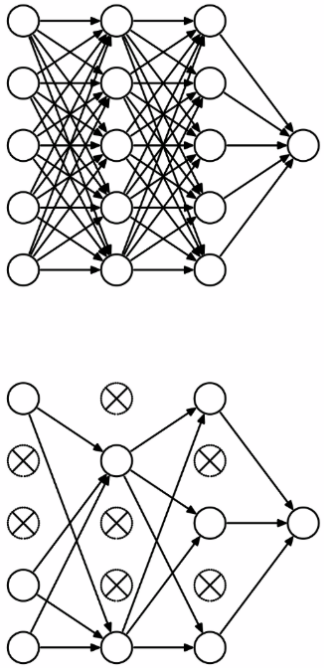


Contents

Dropouts

Dropout is another regularization tool which literally means "dropping out" a random number of neurons during every training step. After training, each neuron's input connection weight needs to be adjusted by a factor of (1-dropout rate).

Effectively, all we need to do is to set to zero a random number of outputs from a given layer:



Dropouts

Keras:

Dropout Regularization

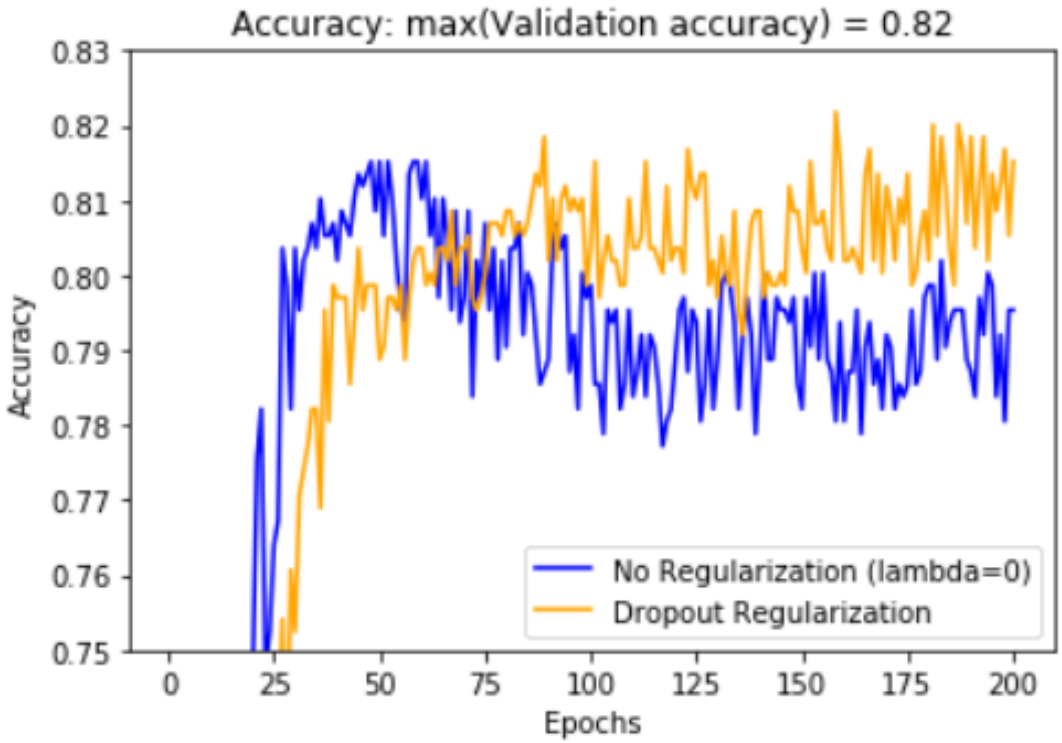
```
dropout_rate = 0.3

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(X_train.shape[1],)))
model.add(layers.Dropout(dropout_rate))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(dropout_rate))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(dropout_rate))
#model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(2, activation='softmax'))

model.summary()
```

Dropouts

Keras:



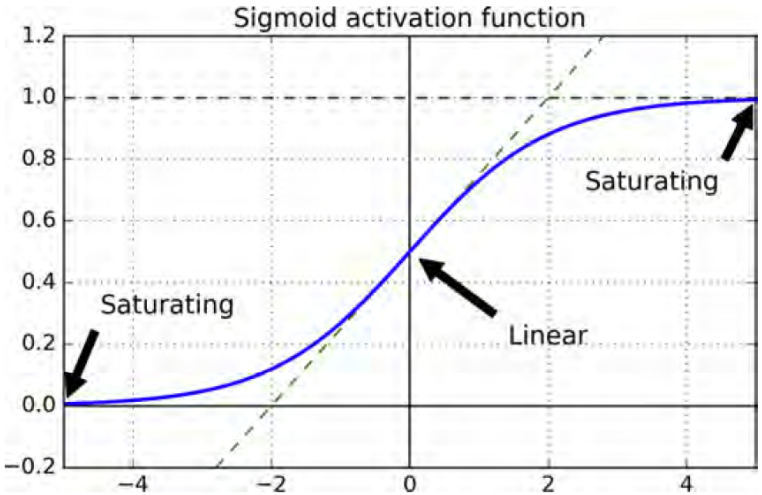
Contents

Vanishing/Exploding Gradients Problems

Unstable gradients:

1. *Vanishing gradients* problem: Given current weights $\boldsymbol{w}$ of the NN and inputs (data), the gradient of the activation function may be very small resulting in the corresponding weights virtually unchanged during the iterations / updates.
2. *Exploding gradients* problem: The weights may one the contrary blow up - this problem is mostly encountered in recurrent neural networks.

Sigmoid function:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

Techniques to Alleviate the Unstable Gradient Problems

Ways to resolve the problems:

1. "Proper" initialization of weights: special initial distribution, reusing pretrained layers, etc.
2. Nonsaturating activations functons: Leaky ReLU, exponential LU (ELU), etc.
3. Batch normalization (BN): scale inputs before each layer during training (two more parameters)
4. Gradient clipping: set a threshold for the gradient

Contents

SGD, mini-batch GD, and GD Optimization: 'sgd'

The SGD, mini-batch GD, and GD Optimization (with learning rate α) are all defined as follows:

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \underbrace{\frac{1}{s} \sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})},$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and

- $s = 1$ in case of Stochastic Gradient Descent (SGD)
- $1 < s < m$ in case of mini-batch Gradient Descent (mini-batch GD)
- $s = m$ in case of Gradient Descent (GD)

Here, m denotes the total number of observations in the data set.

SGD, mini-batch GD, and GD Optimization

Example: Classification via mini-batch GD with $s = 128$ and $\alpha = 0.01$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='sgd')

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

SGD, mini-batch GD, and GD Optimization

Example: Classification via mini-batch GD with $s = 128$ and $\alpha = 0.05$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.SGD(lr=0.05))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Contents

Momentum Optimization

The Momentum Optimization algorithm is defined as follows:
Fist, *momentum vector* $\mathbf{v}$ is initialized at $\mathbf{0}$ and then the updates are

$$\mathbf{v} := -\alpha \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})} + \eta\, \boldsymbol{v}$$

$$\boldsymbol{w} := \boldsymbol{w} + \mathbf{v}$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set.

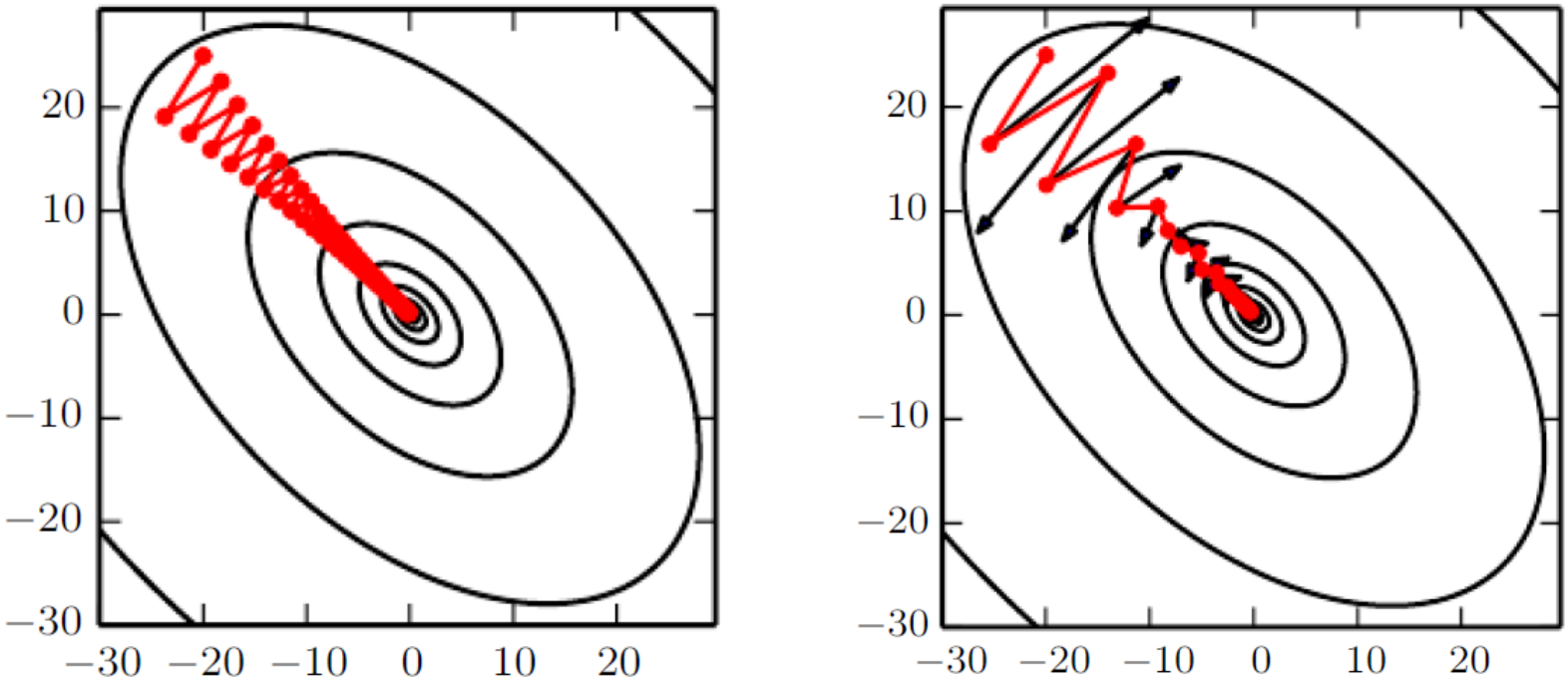
The hyperparameters of the algorithm are

- s - mini-batch size
- α - learning rate
- η - *momentum*, a number between 0 and 1

Momentum Optimization

Example: Path in (w_1, w_2) plane.

Left: no momentum, i.e. $\eta = 0$. Right: Momentum optimization with $\eta > 0$.



Source: *Deep Learning* by I. Goodfellow, Y. Bengio, and A. Courville

Momentum Optimization

Example: Classification via Momentum Optimization with $s = 128$, $\alpha = 0.05$, and $\eta = 0.9$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.SGD(lr=0.05, momentum=0.9))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Nesterov Accelerated Gradient (NAG)

The Nesterov Accelerated Gradient (NAG) algorithm is defined as follows: Fist, *momentum vector* $\mathbf{v}$ is initialized at $\mathbf{0}$ and then the updates are

$$\mathbf{v} := -\alpha \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w} + \eta\, \mathbf{v})}_{\approx \nabla J(\boldsymbol{w})} + \eta\, \boldsymbol{v}$$

$$\boldsymbol{w} := \boldsymbol{w} + \mathbf{v}$$

where $L^{(i)}(\boldsymbol{w} + \eta\, \mathbf{v})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set.

The hyperparameters of the algorithm are

- s - mini-batch size
- α - learning rate
- η - *momentum*, a number between 0 and 1

Nesterov Accelerated Gradient (NAG)

Example: Classification via Nesterov Accelerated Gradient (NAG) with $s = 128$, $\alpha = 0.05$, and $\eta = 0.9$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.SGD(lr=0.05, momentum=0.9, nesterov=True))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

AdaGrad

The AdaGrad algorithm is defined as follows:
Fist, initialize vector $\mathbf{r}$ (with $r_k > 0$) and then the updates are

$$\mathbf{g} := \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})}$$

$$\mathbf{r} := \mathbf{r} + \mathbf{g} \odot \mathbf{g}$$

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{\sqrt{\mathbf{r} + \epsilon}} \odot \mathbf{g}$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set. $\odot$ denotes element-wise multiplication. The hyperparameters of the algorithm are

- s - mini-batch size
- α - learning rate
- ϵ - positive small parameter, typically around 10^{-7}

AdaGrad

Example: AdaGrad with $s = 128$, $\alpha = 0.05$, $\epsilon = 10^{-5}$, and r_k initialized at 0.1.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 3
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.Adagrad(lr=0.05, epsilon=1e-5))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

RMSProp

The RMSProp is defined as follows:
Fist, initialize vector $\mathbf{r}$ (with $r_k > 0$) and then the updates are

$$\mathbf{g} := \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})}$$

$$\mathbf{r} := \rho\, \mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g}$$

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{\sqrt{\mathbf{r}+\epsilon}} \odot \mathbf{g}$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and $1 \leq s \leq m$, where m denotes the total number of observations in the data set. $\odot$ denotes element-wise multiplication. The hyperparameters of the algorithm are

- s - mini-batch size
- α - learning rate
- ϵ - positive small parameter, typically around 10^{-7}
- ρ - decay rate between 0 and 1, typically around 0.9

RMSProp

Example: AdaGrad with $s = 128$, $\alpha = 0.05$, $\epsilon = 10^{-5}$, and $\rho = 0.9$.

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.RMSprop(lr=0.05, rho=0.9, epsilon=1e-07))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Adam

The Adam (Adaptive Momentum) is defined as follows: Fist, initialize *momentum vector* $\mathbf{v} = \mathbf{0}$ and vector $\mathbf{r}$ (with $r_k > 0$), then the updates at iteration step t are

$$\mathbf{g} := \underbrace{\frac{1}{s}\sum_{i=1}^{s} \nabla L^{(i)}(\boldsymbol{w})}_{\approx \nabla J(\boldsymbol{w})},$$

$$\mathbf{v} := (1-\beta_1)\mathbf{g} + \beta_1 \mathbf{v}, \quad \mathbf{v} := \frac{\mathbf{v}}{1-\beta_1^t},$$

$$\mathbf{r} := \beta_2\, \mathbf{r} + (1-\beta_2)\mathbf{g} \odot \mathbf{g}, \quad \mathbf{r} := \frac{\mathbf{r}}{1-\beta_2^t},$$

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{\sqrt{\mathbf{r}+\epsilon}} \odot \mathbf{v},$$

where $L^{(i)}(\boldsymbol{w})$ is based on one observation i and $1 \le s \le m$, where m denotes the total number of observations in the data set. $\odot$ denotes element-wise multiplication. The hyperparameters of the algorithm are

- s is the mini-batch size and α is learning rate
- β_1 - *momentum*, a number between 0 and 1 (analogous to η in Momentum Opt.)
- β_2 - decay rate between 0 and 1, typically around 0.9 (analogous to ρ in RMSProp)
- ϵ - positive small parameter, typically around 10^{-7}

Adam

Example: Adam with $s = 128$, $\alpha = 0.001$, $\epsilon = 10^{-7}$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35
model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Adam

Example: Adam with $s = 128$, $\alpha = 0.05$, $\epsilon = 10^{-5}$, $\beta_1 = 0.85$, and $\beta_2 = 0.95$

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_35 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_36 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_37 (Dense)             (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
nepochs = 35

model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
              optimizer=keras.optimizers.adam(lr=0.05, beta_1=0.85, beta_2=0.95, epsilon=1e-05))

history = model.fit(X_train, y_train,
          batch_size=128, epochs=nepochs,
          verbose=1,
          validation_data=(X_test, y_test))
```

Contents

Policy Parameterization

Recall:
Optimal state-value function:

$$v_*(s) \doteq \max_\pi v_\pi(s)$$
$$= \max_\pi E_\pi \left[G_t | S_t = s \right]$$

Can we try searching over policies directly (whether also estimating the state-values or not)?
$\Rightarrow$ *Policy Gradient Methods*

Policy Parameterization

Assume *parameterized policy* $\pi(a|s, \boldsymbol{\theta})$, where $\boldsymbol{\theta} \in \mathbb{R}^{d'}$.

For example,

- if action space is discrete, one can define

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}},$$

  where $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ are parameterized *numerical preferences* for each state-action pair (s, a);

- if action space is continuous with $a \in \mathbb{R}$, one can define

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{1}{\sigma(s, \boldsymbol{\theta})\sqrt{2\pi}} e^{-\frac{(a-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}},$$

  where $\mu(s, \boldsymbol{\theta}) \in \mathbb{R}$ and $\sigma(s, \boldsymbol{\theta}) \in \mathbb{R}^+$ are some parameterized functions.

Policy Parameterization

Advantages of policy gradient methods:

- Policy approximation methods can have better convergence properties because the action probabilities change smoothly.
- Policy approximation methods can be effective in high-dimensional or even continuous action spaces: no need to estimate individual probabilities for each of the many actions - statistics (such as $\mu(s, \boldsymbol{\theta})$ and $\sigma(s, \boldsymbol{\theta})$) can be learned instead.
- Approximate policy can converge towards a deterministic policy - this can be advantageous in case of deterministic optimal policy (while ε-soft policy with fixed ε, for example, is always stochastic).
- Approximate policy is designed to learn stochastic policies - this can be advantageous in case of stochastic optimal policy (e.g. Nash equilibrium).

Disadvantages:

- Approximate policy may converge to a local optimum.
- Evaluating a policy may be inefficient.

Contents

Policy Gradient Theorem

Theorem

For any differentiable in $\boldsymbol{\theta}$ parameterized policy $\pi(a|s, \boldsymbol{\theta})$ and any state $s_0 \in \mathcal{S}$:

$$\nabla v_\pi(s_0) \propto E_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right],$$

where ∇ denotes the gradient with respect to $\boldsymbol{\theta}$.

Contents

REINFORCE

Assume *parameterized policy* $\pi(a|s, \boldsymbol{\theta})$, where $\boldsymbol{\theta} \in \mathbb{R}^{d'}$.

Let's fix $s_0 \in \mathcal{S}$.

By Policy Gradient Theorem, the Stochastic gradient ascent method that maximizes $v_\pi(s_0)$ is then

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)},$$

where we approximate $q_\pi(S_t, A_t)$ via MC return G_t.

REINFORCE

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Source: *Reinforcement Learning: An Introduction* by R. Sutton and A. Barto

REINFORCE with Baseline

It can be shown that

$$E_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] = E_\pi \left[(q_\pi(S_t, A_t) - b(S_t)) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right]$$

for any *baseline* $b(S_t)$ because

$$\sum_a b(s) \nabla \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla \sum_a \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla 1 = 0.$$

Then the updates can be written as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(G_t - b(S_t) \right) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}.$$

REINFORCE with Baseline

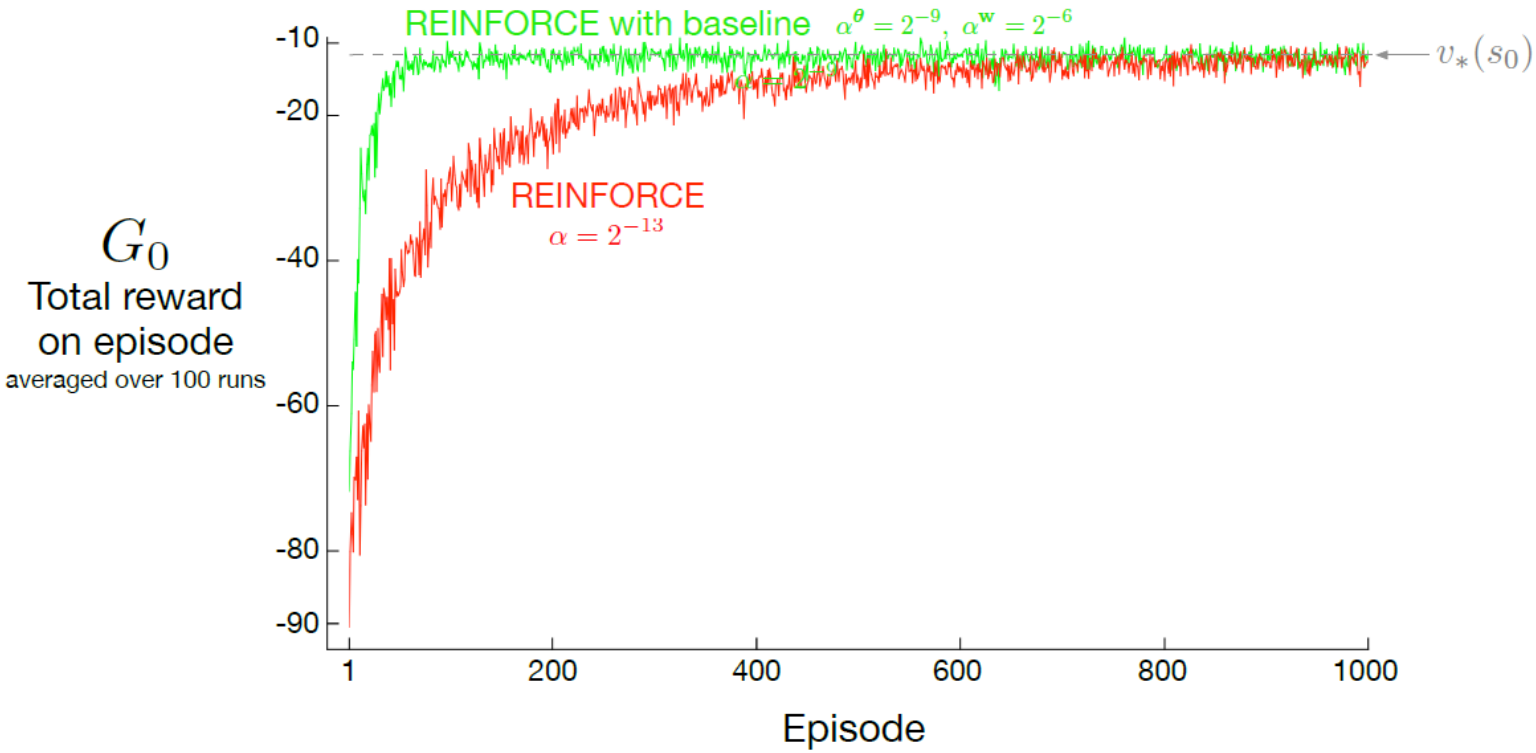
REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Source: *Reinforcement Learning: An Introduction* by R. Sutton and A. Barto

REINFORCE v.s. REINFORCE with Baseline



Source: *Reinforcement Learning: An Introduction* by R. Sutton and A. Barto

Contents

Actor–Critic

Instead of MC return G_t, one can use

- n-step TD return with Approximation
- λ-return with Approximation

These methods are called *Actor-Critic*.

For example, REINFORCE with 1-step TD return becomes:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(\underbrace{R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})}_{G_{t:(t+1)}} - b(S_t) \right) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}.$$

1-step Actor–Critic

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma\,\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\,\delta\,\nabla\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\,I\,\delta\,\nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Source: *Reinforcement Learning: An Introduction* by R. Sutton and A. Barto

Actor–Critic with Eligibility Traces

Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S,\mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + I \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Source: *Reinforcement Learning: An Introduction* by R. Sutton and A. Barto