# Efficient Models

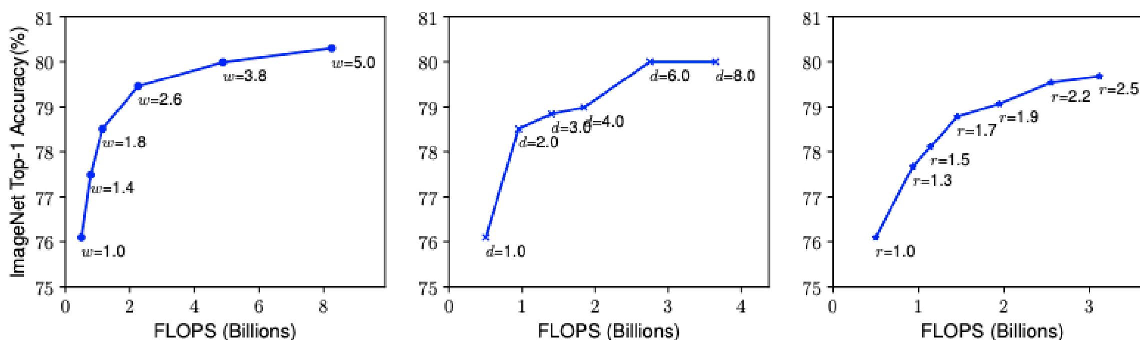
By Yuhang Song

[This Note is recorded from 28th Sep 2022 to ...]

## EfficientNet V1 (2019)

### Idea & Background

The paper[0] proposed that, modern CNNs were developed with more layers(deeper), more channels(wider), and higher quality of input images(hgher resolutions). However, scaling up any of the parameters mentioned monotonically to a large number would not very much benefits the model, in particular, the model might in this case, reaches an accuracy saturation.




As shown above, the paper evaluated CNN models that monotonically scalling up its width $w$, depth $d$ and resolution $r$, the improvements is significant until the scaling reaches a certain limit.

The paper concludes that:

- **Observation 1** – Scaling up any dimension of network width, depth, or resolution improves accuracy, but the accuracy gain diminishes for bigger models.

Intuitively, increased input resolutions needs wider networks that are able to capture more fine-grained patterns with more pixels, as well as the higher depth such that the larger receptive fields would help capture similar features that include more pixels.

The paper anylyzed this idea by scaling network width for different baseline networks. As illustrated below:

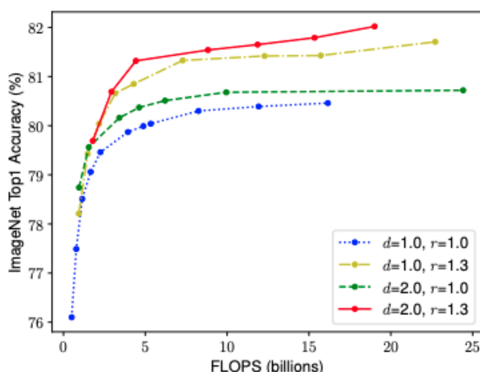



*Figure 4.* **Scaling Network Width for Different Baseline Networks**. Each dot in a line denotes a model with different width coefficient ($w$). All baseline networks are from Table 1. The first baseline network ($d$=1.0, $r$=1.0) has 18 convolutional layers with resolution 224x224, while the last baseline ($d$=2.0, $r$=1.3) has 36 layers with resolution 299x299.

These result lead us to the second observation:

- **Observation 2** – In order to pursue better accuracy and efficiency, it is critical to balance all dimensions of network width, depth, and resolution during ConvNet scaling.

Since the current existing methods are adjusting width, depth, resolutions manually, the paper hence propose an new scaling method that uniformly scales all dimensions of depth/width/resolution using a simple yet highly effective **compound coefficient**.

### Compound Coefficient Scaling

Define a compound coefficient $\phi$ that is used to uniformly scales network width, depth, and resolution in a principled way:

$$\text{depth: } d = \alpha^\phi$$

$$\text{width: } w = \beta^\phi$$

$$\text{resolution: } r = \gamma^\phi$$

$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$

$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1$$

From which, the $\alpha, \beta, \gamma$ are constants that can be determined by a small grid search, $\phi$ is a user-defined coefficient that controls how many more resources are available for model scalling.

The paper point out that, the FLOPSof a regular convolution operation is proportional to $d, w^2, r^2$, that is in other words, 2X network depth will gain 2X FLOPS, but 2X network width or resolution will increase FLOPS by 4X. Also, because convolutional layers are usually dominate the computation cost in ConvNets, scalling a ConvNet with above equation will approximately increase total FLOPS by $(\alpha \cdot \beta^2 \cdot \gamma^2)^\phi$, the method constraints $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$ so that the total FLOPS will approximately increase by $2^\phi$.

### EfficientNet Architecture

The EfficientNet is based on MnasNet (by Platform-aware Neural Architecture Search), except EfficientNet-B0 is slightly bigger with FLOPS targets set to 400M. The proposed EfficientNet-B0 is as following:

| Stage $i$ | Operator $\hat{\mathcal{F}}_i$ | Resolution $\hat{H}_i \times \hat{W}_i$ | #Channels $\hat{C}_i$ | #Layers $\hat{L}_i$ |
|---|---|---|---|---|
| 1 | Conv3x3 | $224 \times 224$ | 32 | 1 |
| 2 | MBConv1, k3x3 | $112 \times 112$ | 16 | 1 |
| 3 | MBConv6, k3x3 | $112 \times 112$ | 24 | 2 |
| 4 | MBConv6, k5x5 | $56 \times 56$ | 40 | 2 |
| 5 | MBConv6, k3x3 | $28 \times 28$ | 80 | 3 |
| 6 | MBConv6, k5x5 | $14 \times 14$ | 112 | 3 |
| 7 | MBConv6, k5x5 | $14 \times 14$ | 192 | 4 |
| 8 | MBConv6, k3x3 | $7 \times 7$ | 320 | 1 |
| 9 | Conv1x1 & Pooling & FC | $7 \times 7$ | 1280 | 1 |

In which, the MBConv block is mobile inverted bottleneck, the SE block is added into each block for optimization.

Starting with EfficientNet-B0, the compound scaling method is performed with 2 steps:

- STEP 1: first fix ϕ = 1, assuming twice more resources available, and do a small grid search of $\alpha, \beta, \gamma$ based on equation shown above. In particular, the paper found the best values for EfficientNet-B0 are $\alpha = 1.2$, $\beta = 1.1$, $\gamma = 1.15$, under constraint of $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$.
- STEP 2: fix $\alpha, \beta, \gamma$ as constants and scale up baseline network with different $\phi$ using above equation, to obtain EfficientNet-B1 to B7.

### Evaluations

EfficientNet B0-B7 with inceased scaling up:

*Table 2.* **EfficientNet Performance Results on ImageNet** (Russakovsky et al., 2015). All EfficientNet models are scaled from our baseline EfficientNet-B0 using different compound coefficient $\phi$ in Equation 3. ConvNets with similar top-1/top-5 accuracy are grouped together for efficiency comparison. Our scaled EfficientNet models consistently reduce parameters and FLOPS by an order of magnitude (up to 8.4x parameter reduction and up to 16x FLOPS reduction) than existing ConvNets.

| Model | Top-1 Acc. | Top-5 Acc. | #Params | Ratio-to-EfficientNet | #FLOPs | Ratio-to-EfficientNet |
|---|---|---|---|---|---|---|
| **EfficientNet-B0** | **77.1%** | **93.3%** | **5.3M** | **1x** | **0.39B** | **1x** |
| ResNet-50 (He et al., 2016) | 76.0% | 93.0% | 26M | 4.9x | 4.1B | 11x |
| DenseNet-169 (Huang et al., 2017) | 76.2% | 93.2% | 14M | 2.6x | 3.5B | 8.9x |
| **EfficientNet-B1** | **79.1%** | **94.4%** | **7.8M** | **1x** | **0.70B** | **1x** |
| ResNet-152 (He et al., 2016) | 77.8% | 93.8% | 60M | 7.6x | 11B | 16x |
| DenseNet-264 (Huang et al., 2017) | 77.9% | 93.9% | 34M | 4.3x | 6.0B | 8.6x |
| Inception-v3 (Szegedy et al., 2016) | 78.8% | 94.4% | 24M | 3.0x | 5.7B | 8.1x |
| Xception (Chollet, 2017) | 79.0% | 94.5% | 23M | 3.0x | 8.4B | 12x |
| **EfficientNet-B2** | **80.1%** | **94.9%** | **9.2M** | **1x** | **1.0B** | **1x** |
| Inception-v4 (Szegedy et al., 2017) | 80.0% | 95.0% | 48M | 5.2x | 13B | 13x |
| Inception-resnet-v2 (Szegedy et al., 2017) | 80.1% | 95.1% | 56M | 6.1x | 13B | 13x |
| **EfficientNet-B3** | **81.6%** | **95.7%** | **12M** | **1x** | **1.8B** | **1x** |
| ResNeXt-101 (Xie et al., 2017) | 80.9% | 95.6% | 84M | 7.0x | 32B | 18x |
| PolyNet (Zhang et al., 2017) | 81.3% | 95.8% | 92M | 7.7x | 35B | 19x |
| **EfficientNet-B4** | **82.9%** | **96.4%** | **19M** | **1x** | **4.2B** | **1x** |
| SENet (Hu et al., 2018) | 82.7% | 96.2% | 146M | 7.7x | 42B | 10x |
| NASNet-A (Zoph et al., 2018) | 82.7% | 96.2% | 89M | 4.7x | 24B | 5.7x |
| AmoebaNet-A (Real et al., 2019) | 82.8% | 96.1% | 87M | 4.6x | 23B | 5.5x |
| PNASNet (Liu et al., 2018) | 82.9% | 96.2% | 86M | 4.5x | 23B | 6.0x |
| **EfficientNet-B5** | **83.6%** | **96.7%** | **30M** | **1x** | **9.9B** | **1x** |
| AmoebaNet-C (Cubuk et al., 2019) | 83.5% | 96.5% | 155M | 5.2x | 41B | 4.1x |
| **EfficientNet-B6** | **84.0%** | **96.8%** | **43M** | **1x** | **19B** | **1x** |
| **EfficientNet-B7** | **84.3%** | **97.0%** | **66M** | **1x** | **37B** | **1x** |
| GPipe (Huang et al., 2018) | 84.3% | 97.0% | 557M | 8.4x | - | - |

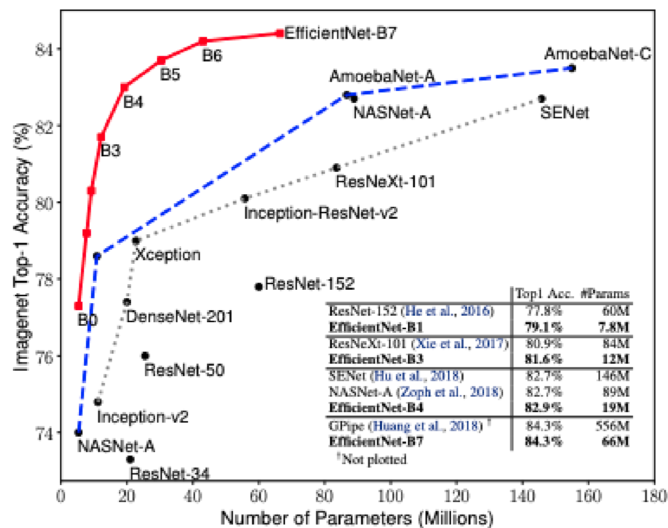
We omit ensemble and multi-crop models (Hu et al., 2018), or models pretrained on 3.5B Instagram images (Mahajan et al., 2018).

Note that the EfficientNet-B7 is with same accuracy of GPipe while maintaining **9X** less parameters.

The paper also performed the compound scales on state-or-art models and obtained:

*Table 3.* **Scaling Up MobileNets and ResNet.**

| Model | FLOPS | Top-1 Acc. |
|---|---|---|
| Baseline MobileNetV1 (Howard et al., 2017) | 0.6B | 70.6% |
| Scale MobileNetV1 by width ($w$=2) | 2.2B | 74.2% |
| Scale MobileNetV1 by resolution ($r$=2) | 2.2B | 72.7% |
| **compound scale (*d*=1.4, *w*=1.2, *r*=1.3)** | **2.3B** | **75.6%** |
| Baseline MobileNetV2 (Sandler et al., 2018) | 0.3B | 72.0% |
| Scale MobileNetV2 by depth ($d$=4) | 1.2B | 76.8% |
| Scale MobileNetV2 by width ($w$=2) | 1.1B | 76.4% |
| Scale MobileNetV2 by resolution ($r$=2) | 1.2B | 74.8% |
| **MobileNetV2 compound scale** | **1.3B** | **77.4%** |
| Baseline ResNet-50 (He et al., 2016) | 4.1B | 76.0% |
| Scale ResNet-50 by depth ($d$=4) | 16.2B | 78.1% |
| Scale ResNet-50 by width ($w$=2) | 14.7B | 77.7% |
| Scale ResNet-50 by resolution ($r$=2) | 16.4B | 77.5% |
| **ResNet-50 compound scale** | **16.7B** | **78.8%** |




# EfficientNet V2 (2021)

## Background & Idea

EfficientNet V2 has got improved training speed and better performance than EfficientNet V1. In this upgraded version, we focus not only on the accuracy and #parameters/FLOPs, but jointly focusing on the training efficiency as well.

The paper[1] identifies several problems of the previouse EfficientNet V1:

- Training with very large image sizes is slow. **(Proposed Solution: Progressive Learning)**
- Depthwise Convolutions are slow in early layers but effective in later layers, since the depthwise convolutions often cannot fully utilize modern accelerators. **(Proposed Solution: Replacing MBConv layers by Fused-MBConv via NAS)**
- Equally scaling up every stage is sub-optimal. **(Proposed Solution: New Scaling Rule and Restriction)**

The Fused-MBConv (better utilize mobile or server accelerators) is replacing the original expension layer and depthwise convolution layer of MBConv by a single 3x3 convolution, the paper evaluated that by using Fused-MBConv in early stage(1-3) helps accelerate the training step with a small overhead on parameters and FLOPs. NAS is used to automatically search for the best combination.

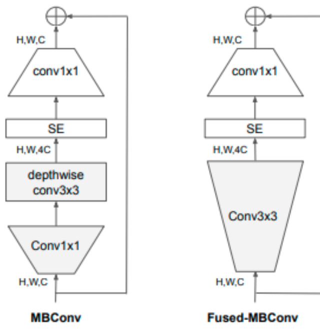



*Figure 2.* Structure of MBConv and Fused-MBConv.

## Training-Aware NAS and Scaling

The Training-Aware NAS is based on Platform-Aware NAS[2], which its search space is also a stage-based factorized space, with the following search options:

Convolution Ops : {MBConv, Fused-MBConv}

Kernel Size: {3x3, 5x5}

Expension Ratio: {1,4,6}

however, in Training-Aware NAS, the paper point out that, they removed unnecessary search options like skip ops, and resued the same channel sizes from the backbone as they aree already searched. In addition, the search reward in Training-Aware NAS conbines model accuracy $A$, the normalized traning step time $S$, and the parameter size $P$, by using a simple weighted product $A \cdot S^w \cdot P^v$, where $w = -.0.07$ and $v = -0.05$, empirically determined to balance the trade-offs similar to [2].

The resulting architecture namely EfficientNetV2-S is given as:

| Stage | Operator | Stride | #Channels | #Layers |
|---|---|---|---|---|
| 0 | Conv3x3 | 2 | 24 | 1 |
| 1 | Fused-MBConv1, k3x3 | 1 | 24 | 2 |
| 2 | Fused-MBConv4, k3x3 | 2 | 48 | 4 |
| 3 | Fused-MBConv4, k3x3 | 2 | 64 | 4 |
| 4 | MBConv4, k3x3, SE0.25 | 2 | 128 | 6 |
| 5 | MBConv6, k3x3, SE0.25 | 1 | 160 | 9 |
| 6 | MBConv6, k3x3, SE0.25 | 2 | 256 | 15 |
| 7 | Conv1x1 & Pooling & FC | - | 1280 | 1 |

As for the scaling part, EfficientNetV2-S is scaled up to EfficientNetV2-M/L using compound scaling with optimizations: (1) Restrict maximum inference image size to 480. (2) Gradually add more layers to later stages to increase the network capacity without adding much runtime overhead.

The comparisions of EfficientNetV2 and many other state-of-the-art models are given as the following:

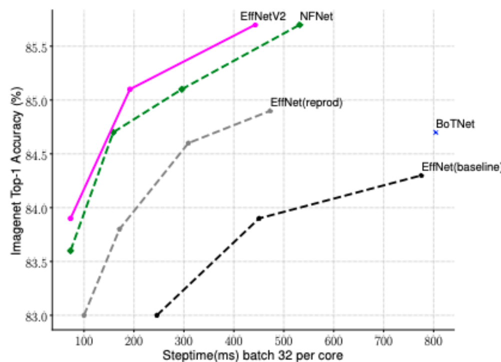



*Figure 3.* ImageNet accuracy and training step time on TPUv3 – Lower step time is better; all models are trained with fixed image size without progressive learning.

## Progressive Learning

The main idea of progressive learning is to increase image size and regularzation magnitude at the same time during the training stage. The paper agure that the loss of accuracy of only progressively

enlarge input image size during training drops due to unbalanced regularization.

The algorithm is shown in below:

**Algorithm 1** Progressive learning with adaptive regularization.

**Input:** Initial image size $S_0$ and regularization $\{\phi_0^k\}$.
**Input:** Final image size $S_e$ and regularization $\{\phi_e^k\}$.
**Input:** Number of total training steps $N$ and stages $M$.
**for** $i = 0$ **to** $M - 1$ **do**
  Image size: $S_i \leftarrow S_0 + (S_e - S_0) \cdot \frac{i}{M-1}$
  Regularization: $R_i \leftarrow \{\phi_i^k = \phi_0^k + (\phi_e^k - \phi_0^k) \cdot \frac{i}{M-1}\}$
  Train the model for $\frac{N}{M}$ steps with $S_i$ and $R_i$.
**end for**

In which $N$ is the total number of training steps, the targeting image size is $S_e$, and $\Phi_e = \phi_e^k$ is a list of regularization magnitude, where $k$ represent a type of regularization such as dropout rate or mixup rate value. The training is divided into $M$ stages, the model is trained with image size $S_i$ where $1 \leq i \leq M$.

### Evaluation

EfficientNetV2-M achieves comparable accuracy to EfficientNet-B7 while training 11x faster using the same computing resources. EfficientNetV2 models also significantly outperform all recent RegNet and ResNeSt, in both accuracy and inference speed.

# Efficient Reinforcement Learning

## Reinforcement Learning

## Model-based RL: Dynamic Programming

In MDP, we want to model the state value function and state-action value functions, besed on which, we can form a strategy that greedly select actions with max state-action value in for each individual state.

We define state value function $v_\pi$ and stage-action value function $q_\pi$ for a certain policy $\pi$ as following:

$$q_\pi(s, a) = \mathbb{E}_{\tau \backsim \pi}[R(\tau)|S_0 = s, A_0 = a]$$
$$v_\pi(s) = \mathbb{E}_{a \backsim \tau}[q_\pi(s, a)]$$

The Bellman Expectation Functions provids us a way to iteratively calculate value functions by decompose them into immediate reward plus discounted value of successor state.

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$
$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}|S_t = s, A_t = a)]$$

The Bellman Equations can be solved directly if we have full information of the environment(i.e. we know the state transformation function), in discrete finite state environment:

$$v = r + \gamma P v$$

From which $v$ and $r$ are scalars, $P$ is state transform probability matrix. Solve it directly we get:

$$v = (I - \gamma P)^{-1} r$$

The Bellman Optimally Equation can be then written as:

$$v_*(s) = max_a \mathbb{E}[R_s^a + \gamma v_*(s')] = max_a R_s^a + \gamma \sum_{s' \in S} p_{ss'}^a v_*(s')$$

$$q_*(s,a) = R_s^a + \gamma \mathbb{E}[max_{a'} q_*(s',a')] = R_s^a + \gamma \sum_{s' \in S} p_{ss'}^a max_{a'} q_*(s',a')$$

The complexity of solving Bellman Expectation equation is $O(n^3)$, where $n$ is the number of states, that means it is hard to solve when having large state space. In such case, we need to use methods like Dynamic Programming, Monte-Carlo Estimation, or Temporal Difference. In other hand, Bellman Optimalityt Equations are non-linear, and has no closed form solution(in general), therefore cannot be directly solved, we need to use other methods.

Dynamic Programming iteratively solves large scale questions by decomposite them into smaller ones, those questions have to be:

- With Optimal Substructure
- Overlapping Subproblems

MDP satisfy both propeerties. We can therefore use DP to solve MDP questions, **note that DP solutions requires Full Knowledge of the MDP, and are hence Model-Based RL methods.**

### Policy Iteration

Policy Iteration method evaluate a given policy $\pi$ by dynamic programming, it iteratively use Bellman Expectations to evaluate the state function of given policy $\pi$. Specifically for each iteration $k$:

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s'))$$

To improve the policy, acting greedily with respect to $v_\pi$:

$$\pi' = greedy(v_\pi) = argmax_{a \in A}\ q_\pi(s,a)$$

The algorithm converges to $v_*(s)$ with greedy policy imrovement, otherwise converges to real $v_\pi(s)$.

### Value Iteration

Based on Principle of Optimality, which states a policy $\pi(s)$ is an optimal policy on state $s$ if and only if $\pi(s)$ achives $v_\pi(s') = v_*(s')$ for any state $s'$ that is reachable from $s$. From which, it implies if we know the solution of $v_*(s')$, we can figure out the optimal solution to any state $s$ by **One-Step Full Backup**.

Formally, if we know the solution to subproblems $v_*(s')$, the solution $v_*(s)$ can be found be one-step lookahead:

$$v_*(s) = max_{a \in A} R_s^a + \gamma \sum_{s' \in S} p_{ss'}^a v_*(s')$$

The algorithm converges to $v_*(s)$.

In summery:

| Problem | Bellman Equation | Method (Algorithm) |
|---|---|---|
| Value Function Prediction | Bellman Expectation Equation | Policy Iteration |
| Control | Bellman Expectation Equation | Policy Iteration + Greedy Policy Improvement |

| Problem | Bellman Equation | Method (Algorithm) |
|---|---|---|
| Control | Bellman Optimality Equation | Value Iteration |

### Asynchronous Dynamic Programming

DP methods described above used synchronous backups, where all states are backed up in parallel. Asynchronous DP backs up states individually, in any order, can significantly reduce computation. It is guaranteed to converge if all states continue to be selected.

Three simple ideas for asynchronous dynamic programming:

- In-place dynamic programming
- Prioritised sweeping
- Real-time dynamic programming

## Model-free Value-based Methods

Dynamic Programming RL methods are all model based methods, in which we need specific environment model to excute them, it is common in real-world RL environment that we dont know environment model, Monte-Carlo / Temporal Difference methods provided algorithms that are model-free to predict value functions.

### Monte-Carlo

Monte-Carlo(MC) methods learn directly from episodes of experience, instead of evaluate policy by expected return(based on environment knowledge), it uses mean return(based on empirical knowledge) to estimate the value function.

The basic idea is, to evaluate state value for $s$, the firstt/every time $t$ when state $s$ is visited in an episode, increment counter $N(s) \leftarrow N(s)+1$, increment total return $S(s) \leftarrow S(s)+G_t$, the estimated state value can be calculated as:

$$V(s) = \frac{S(s)}{N(s)}$$

By law of large numbers, $V(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$.

By incremental MC updates, the final MC evaluation equation can be written as:

$$V(s_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

### Temporal Difference

Temporal Difference(TD) method learns from incomplete eepisodes, in which the agent do not have to wait until finish whole episode to update value function, like did in MC. TD updates value function by leverage the differences between target and estimation in different time step. It uses idea of **Bootstrapping** with a biased estimation. Less precise than MC, but more convinent and with lower variance.

In Monte-Carlo methods, the value function is updated by:

$$V(s_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

Alternatively in TD method, we update value of $V(S_t)$ towards estimated return $R_{t+1} + \gamma V(S_{t+1})$:

$$V(s_t) \leftarrow V(S_t) + \alpha([R_{t+1} + \gamma V(S_{t+1})] - V(S_t))$$

Where $R_{t+1} + \gamma V(S_{t+1})$ called *TD target*, and $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ called *TD error*.

TD can learn before(or without) knowing the final outcome, whereas in order for MC to learn, we need to wait until the termination of the episode which only works in episodic environments.

### On-Policy Value-based Controls

**MC based: Greedy Policy Improvements**

Evaluate state-action value functions $q_\pi(s, a)$ instead of state value function $v_\pi(s)$:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(G_t - q(s_t, a_t))$$

Improve policy by $\epsilon$-greedily selecting $q(s, a)$.

**TD based: Sarsa**

Sarsa algorithm, replace MC by TD in control loop:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(R + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t))$$

### Off-Policy Value-based Controls

Objective:

- Learn from observing humans or other agents.
- Re-use experience generated from old policies $\pi_1; \pi_2, ...., \pi_{t-1}$.
- Learn about optimal policy while following exploratory policy.
- Learn about multiple policies while following one policy.

**MC based: Importance Sampling**

Gater trajectories from another policy distribution to update current distribution using a trick namely:

[Importance Sampling]

Estimate the expectation of distribution Q from P:

$$\mathbb{E}_{X \backsim P}[f(x)] = \sum P(x)f(x) = \sum Q(x)\frac{P(x)}{Q(x)}f(x) = \mathbb{E}_{X \backsim Q}[\frac{P(x)}{Q(x)}f(x)]$$

Define $G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}\frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})}...\frac{\pi(A_T|S_T)}{\mu(A_T|S_T)}G_t$, update value towards corrected return:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{\pi/\mu} - V(S_t))$$

Note that importance sampling can dramatically increase variance. This mechanism can also be applied to TD:

$$V(S_t) \leftarrow V(S_t) + \alpha(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}(R_{t+1} + \gamma V(S_{t+1})) - V(S_t))$$

**TD based: Q-Learning**

Instead of using target value based on current policy $\pi$, the target value in Q-Learning beased on greedy policy over state-action value function:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(R + \gamma\ max_a q(s_{t+1}, a) - q(s_t, a_t))$$

## Model-free Policy Based methods

Advantages:

- Better convergence properties.
- Effective in high-dimensional or continuous action spaces.
- Can learn stochastic policies.

Disadvantages:

- Typically converge to a local rather than global optimum.
- Evaluating a policy is typically inefficient and high variance.

### Gradient Based: Policy Gradient

Define that in a MDP environment, total reward gain from a certain policy $\pi$ can be shown as:

$$\bar{R}_\theta = \sum_\tau R(\tau) p_\theta(\tau) \quad (1)$$

$$where \;\; p_\theta = p(s_1) \prod_{t=1}^{T} p_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \quad (2)$$

In which policy $\pi$ is parameterised by $\theta$, and $\tau$ represents a single trajectory, $p_\theta$ is the probability of the trajectory.

Since equation (1) involves reward of trajectory $R(\tau)$ times the probability of trajectory $\tau$ following policy $\theta$, noted as $p_\theta$, as well as the summing operation $\sum_\tau$, it can be seen as the expectation of reward for a certain policy:

$$\mathbb{E}_{\tau \backsim p_\theta(\tau)}[R(\tau)]$$

Hence we want to maximize the expectation of reward for a certain policy, to achive this, calculate the gradient of the function and perform gredient acsent:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\tau \backsim p_\theta(\tau)}[R(\tau)] &= \nabla_\theta \int_{\tau_t} R_t \; p_\theta(\tau_t) \;\; d\tau_t \\
&= \int_{\tau_t} R_t \; \nabla_\theta \; p_\theta(\tau_t) \;\; d\tau_t \\
&= \int_{\tau_t} R_t \; p_\theta(\tau_t) \; \nabla_\theta \; log p_\theta(\tau_t) \;\; d\tau_t \\
&= \mathbb{E}_{\tau \backsim p_\theta(\tau)}[R_t \; \nabla_\theta \; log p_\theta(\tau_t)]
\end{aligned}$$

$\mathbb{E}_{\tau \backsim p_\theta(\tau)}[R_t \; \nabla_\theta \; log p_\theta(\tau_t)]$ can be approximated by collecting experience as much as possible and compute the average:

$$\mathbb{E}_{\tau \backsim p_\theta(\tau)}[R(\tau) \; \nabla_\theta \; log p_\theta(\tau)] \approx \frac{1}{N} \sum_{n=1}^{N} R(\tau^n) \; \nabla_\theta \; log p_\theta(\tau^n)$$

Since we dont have full model for $p_\theta$, it is not possible to compute equation (2), that is, we dont know $p(s_{t+1}|s_t, a_t)$, because this term depends on the environment. Here for gradient acsent with respect to $\theta$, we only need $\nabla log \; p_\theta(\tau_t)$ instead of the value of $log \; p_\theta(\tau_t)$ itself, therefore simply replace $p_\theta(\tau_t)$ by $\pi_\theta(a_t|s_t)$ we get:

$$\mathbb{E}_{\tau \backsim p_\theta(\tau)}[R(\tau) \; \nabla_\theta \; \sum_{t=1}^{T_n} log \; \pi_\theta(a_t|s_t)] \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} R(\tau^n) \; \nabla_\theta \; log \; \pi_\theta(a_t^n|s_t^n)$$

After obtaining the gradient of objective function, policy parameters $\theta$ are updated by gradient acsent:

$$\theta \leftarrow \theta + \alpha \nabla \bar{R}_\theta$$

$$where\ \ \nabla \bar{R}_\theta = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\ \nabla_\theta\ log\ \pi_\theta(a_t^n|s_t^n)$$

**Tips1-Add a Baseline**

It is possible that in a specific reinforcement learning environment, that $R(\tau^n)$ is always positive. In this case, we might monotoniclly increase the probability of a certain action, this can be solved by adding a baseline to our equation, so that instead of naively taking rewards feedback from environment, we compare it to the average rewards we have and make the reward be relative to all previous rewards:

$$\nabla \bar{R}_\theta = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} (R(\tau^n) - b)\ \nabla_\theta\ log\ \pi_\theta(a_t^n|s_t^n)$$
$$where\ \ b \approx \mathbb{E}[R(\tau)]$$

In other words, instead of rewarding trajectory by only the environment rewards, we reward a trajectory by how looking at how good this trajectory is, comparing with all other collected trajectories, since all actions in a same trajectory are being weighted by same reward, yet those actions might benefits for different amount.

To address this, we weight the $a_t$ by the reward obtained from time $t$, add a discount factor to rewards obtained in later stages.

$$\nabla \bar{R}_\theta = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} ([\sum_{t^{‘}=t}^{T_n} \gamma^{(t^{‘}-t)} \cdot r_{t^{‘}}^n] - b)\ \nabla_\theta\ log\ \pi_\theta(a_t^n|s_t^n)$$
$$where\ \ b \approx \mathbb{E}[R(\tau)]$$

**Tips2-Assign Suitable Credit**

The current version of objective function evaluates the whole trajectory by the total rewards obtained from the environment, it is reasonable, but assumes in-precise correlations between each actions in the trajectory.

## Actor-Critic: Integrating Value-based & Policy-based

The above PG(Policy Gradient) algorithm is evaluating the policy by MC-style critic(i.e. mean expected reward returned by the environment), in Actor-Critic, we define a critic:

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$$

Where the critic approximates state-action value function $Q(s,a)$, the actor approximates the policy $\pi$, there are parameterized by $w$ and $\theta$ respectively.

Actor-critic algorithm follow an approximate policy gradient, the actor network can be updated by:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(s,a) Q_w(s,a)]$$
$$\nabla\theta = \alpha \nabla_\theta log\pi_\theta(s,a) Q_w(s,a)$$
$$\theta_{t+1} \leftarrow \theta_t + \lambda \nabla\theta_t$$

The critic network is based on critic functions, here use Q-function as an example, the critic network can be updated as:

$$\nabla_w J(w) \approx \mathbb{E}_{\pi_\theta}[MSE(Q_t, R_{t+1} + max_a Q_{t+1})]$$
$$\nabla w = MSE(Q_t, R_{t+1} + max_a Q_{t+1})$$
$$w_{t+1} \leftarrow w_t + \lambda \nabla w_t$$

Instead of leting critic to estimate state-value function, we can allow it to alternatively estimates Advantage function $A(s, a) = Q_w(s, a) - V_v(s)$ to reduce the variance. There are many alternative critic function choices.

## Continuous Action Space

Methods proposed so far only solves for environments that are with discrete action space, so that value functions for each actions or the probability distribution of selecting actions could be computed. However, in real world, most of the problem are with continous action space.

### Deep Derministic Policy Gradient

Deep Derministic Policy Gradient is then proposed, it is able to solve RL environments with continuous action space by integrating ideas of DQN and PG. It can be viewed as extended version of DQN that is able to solve problems with continous action space. DDPG algorithm uses Actor-Critic architecture.

In DQN, in order to evaluate value function, we need $max_a Q_{t+1}$, it is not possible to compute such value in continous action space. Instead of inputing only the state into critic network and obtain the Q-values for all actions, DDPG critic network takes next action computed from actor network as well, and evaluate Q value for this certain action. Updating of DDPG critic network is the same to DQN:

$$\nabla_w J(w) \approx \mathbb{E}_{\pi_\theta}[MSE(Q_t, R_{t+1} + Q_{t+1}^{a \backsim Actor})]$$
$$\nabla w = MSE(Q_t, R_{t+1} + Q_{t+1}^{a \backsim Actor})$$
$$w_{t+1} \leftarrow w_t - \lambda \nabla w_t$$

Intuitively, in DDPG, the actor network performs differently as the one in PG, it is not possible for it to compute probability distributions for all actions in continous action space, therefore we alter the network to output a certain action that could be with max Q value. The target of the actor network in DDPG is to maximize the value of $Q_t(s, a)$ evaluated by critic network, therefore to update actor network, we use gradient acsent:

$$pg = \frac{\partial Q(s, \pi(s; \theta), w)}{\partial \theta} = \frac{\partial Q(s, a, w)}{\partial a} \cdot \frac{\partial a}{\partial \theta}$$
$$\theta \leftarrow \theta + \bar{\lambda} \theta$$

Note that in DDPG, tricks like target networks for both AC networks; memory buffer are being used. We also add a environmental noise $N$ when performing actions to allow exploration, as well as off-policy learning.

### Proximal Policy Optimization

Baseline algorithm of OpenAI. PPO allows off-policy learning to policy gradient algorithm. In policy gradient, we update our policy network by compute gradient of the expected reward function with respect to policy parameters $\theta$, and perform gradient acsent to maximize it:

$$\nabla \bar{R}_\theta = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} R(\tau^n) \nabla_\theta \, log \, \pi_\theta(a_t^n | s_t^n)$$
$$= \mathbb{E}_{\tau \backsim \pi_\theta}[R(\tau^n) \nabla_\theta \, log \, \pi_\theta(a_t^n | s_t^n)]$$
$$\theta \leftarrow \theta + \alpha \nabla \bar{R}_\theta$$

In PPO algorithm, instead of sampling trajectories from policy $\pi_\theta$, in order to increase sample efficiency(reuse expirence), we sample trajectories from another policy $\pi_{\theta'}$ and apply a importance sampling method to correct the difference.

$$\nabla \bar{R}_\theta = \mathbb{E}_{\tau \backsim \pi_{\theta'}} [\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} R(\tau^n) \ \nabla_\theta \ log \ \pi_\theta(a_t^n|s_t^n)]$$

In addition, we need to add a regularzation term (or in TRPO, add a constrain) to the objective function to constrain the difference between two distributions, therefore the objective function becomes:

$$J(\theta) = J_{\theta'}(\theta) - \beta KL(\theta, \theta')$$

Where adaptively set the value of $\beta$, specificlly when $KL(\theta, \theta') > KL_{max}$, increase $\beta$; when $KL(\theta, \theta') < KL_{min}$, decrease $\beta$.

## References

[0] Tan, M., & Le, Q. (2019, May). Efficientnet: Rethinking model scaling for convolutional neural networks. In International conference on machine learning (pp. 6105-6114). PMLR

[1] Tan, M., & Le, Q. (2021, July). Efficientnetv2: Smaller models and faster training. In International Conference on Machine Learning (pp. 10096-10106). PMLR.

[2] Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., & Le, Q. V. (2019). Mnasnet: Platform-aware neural architecture search for mobile. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 2820-2828).

[3] Brock, A., De, S., Smith, S. L., & Simonyan, K. (2021, July). High-performance large-scale image recognition without normalization. In International Conference on Machine Learning (pp. 1059-1071). PMLR.