Network Architecture Search

AutoML and NAS

KU-BIG

AutoML

Network Architecture Search (NAS)

1. AutoML
2. Network Architecture Search
3. Randomly Wired Neural Networks
4. Genetic Algorithm
5. Weight Agnostic Neural Networks

AutoML

Machine Learning(ML)로 설계하는 ML

크게 아래와 같은 세가지 방향으로 연구가 진행

1. Automated Feature Learning
2. Architecture Search
3. Hyperparameter Optimization

1. Feature Extraction, Feature Engineering

- 학습 모델에 Input을 그대로 사용하지 않고, 유의미한 Feature를 추출해서 사용
- 기존에는 각 모델마다 사람이 직접 실험적으로 Feature를 추출

Input -> Feature Engineering -> Evaluation의 무한 반복..

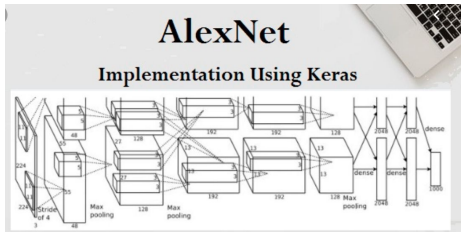
AutoML

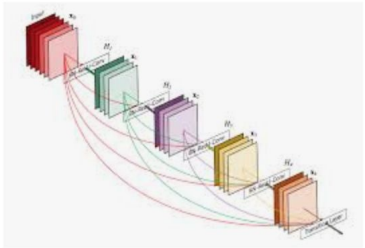
Machine Learning(ML)로 설계하는 ML

크게 아래와 같은 세가지 방향으로 연구가 진행

1. Automated Feature Learning
2. Architecture Search
3. Hyperparameter Optimization

2. Network Architecture Search





Densenet | PyTorch

최근에는 Genetic Algorithm을 이용한 연구 활발

AutoML

Machine Learning(ML)로 설계하는 ML

크게 아래와 같은 세가지 방향으로 연구가 진행

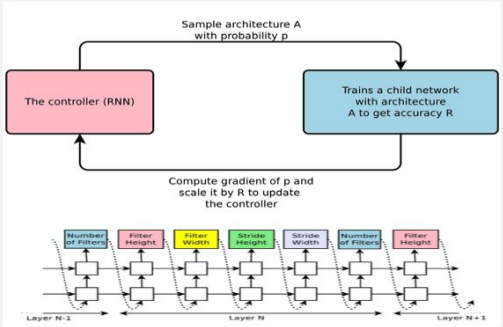
1. Automated Feature Learning
2. Architecture Search
3. Hyperparameter Optimization

3. Hyperparameter Search

- 학습을 시키기 위해 필요한 Hyperparameter들을 학습을 통해 추정
- Learning rate, Mini-batch size 등..

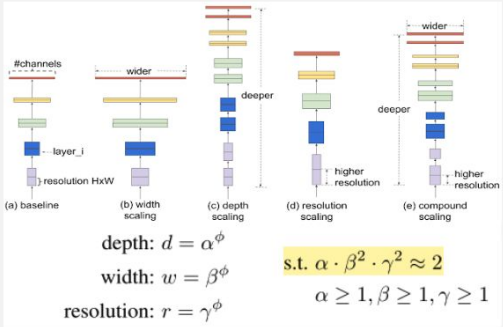
Network Architecture Search

NAS Net with RL
B. Zoph et al(2017) – Google Brain



- Neural Net의 Architecture를 **Reinforcement Learning** 기반으로 탐색하는 모델
- **RNN Controller**를 활용해 Child Network의 Hyperparameter Set을 생성
- **Child Net.**의 **Validation Accuracy**를 RL의 Reward Signal로 활용해 Controller Update
- 제한된 구조의 Search Space 내에서 Controller의 Parameter 최적화

Efficient Net
Tan et al(2019) – Google Brain



- Tan et al(2018)의 **Mnas-Net**을 Baseline Model로 활용
- Conv. Net 모델의 세가지 Scaling 방법(Width, Depth, Resolution)에 대한 실험 결과 제시
- 위의 3가지 Scaling Factor를 동시에 고려하는 **Compound Scaling** 제시
- 모델 사이즈 별 Performance SOTA 달성

NAS with Genetic Algorithm
E Real et al(2018) / David et al(2019) in ICML

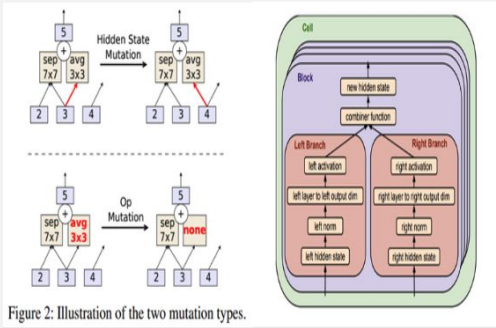
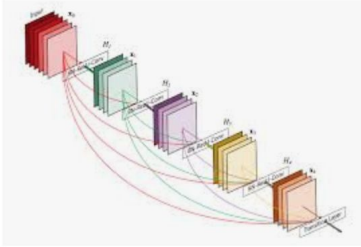


Figure 2: Illustration of the two mutation types.

- RNN Controller 대신에 **Genetic Algorithm**의 일종인 **Tournament Search**를 통해 Optimal Hyperparameter 탐색
- Conv.와 Transformer의 최적 형태를 찾고자 집중
- Image Classification과 NLP Task에 적용

특정 구조의 제한적인 Search Space → Cell(Node)들 간의 연결구조가 단조롭다는 한계가 나타남

Randomly Wired Neural Networks (RWNN)



Densenet | PyTorch

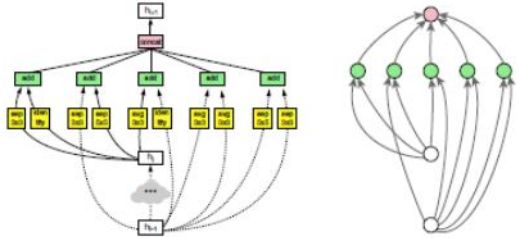


Figure 7. Mapping a NAS cell (left, credit: [56]) to a graph (right).

1. How computational networks are wired is crucial for building intelligent machines: 네트워크가 wired(연결되는) 방식이 중요하다!
2. ResNet and DenseNet, that are effective in large part because of how they are wired: 네트워크 내에서 연결되는 방식이 좋았기 때문에 ResNet이나 DenseNet이 좋은 성능을 보였다. (ResNet처럼 skip connection)을 해준다던지, DenseNet에서처럼 노드들이 pairwise하게 연결 된다던지)
3. wiring 하는 것이 중요하다는 트렌드를 기반으로, NAS(Neural Architecture Search)라는 방법론 출현

- NAS에서도 입력을 어디에서 가져올 것인지는 LSTM에서 뽑아줌(여기서 랜덤성을
가진다고 볼 수 있음)

- 그러나 두개의 operation이 add된다던지, 5개의 노드를 받아 concat된다던지의 design은
사람이 함.
이 constraint를 줄이고 새로운 network generators를 만들어보자!

- NAS와 같은 방법론 하나를 놓고 여기에서 최적의 network architecture를

Randomly Wired Neural Networks (RWNN)

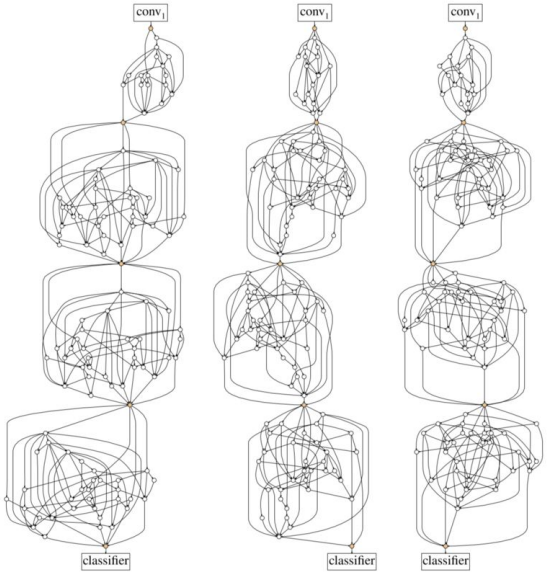


Figure 1. **Randomly wired neural networks** generated by the classical Watts-Strogatz (WS) [50] model: these three instances of random networks achieve (left-to-right) 79.1%, 79.1%, 79.0% classification accuracy on ImageNet under a similar computational budget to ResNet-50, which has 77.1% accuracy.

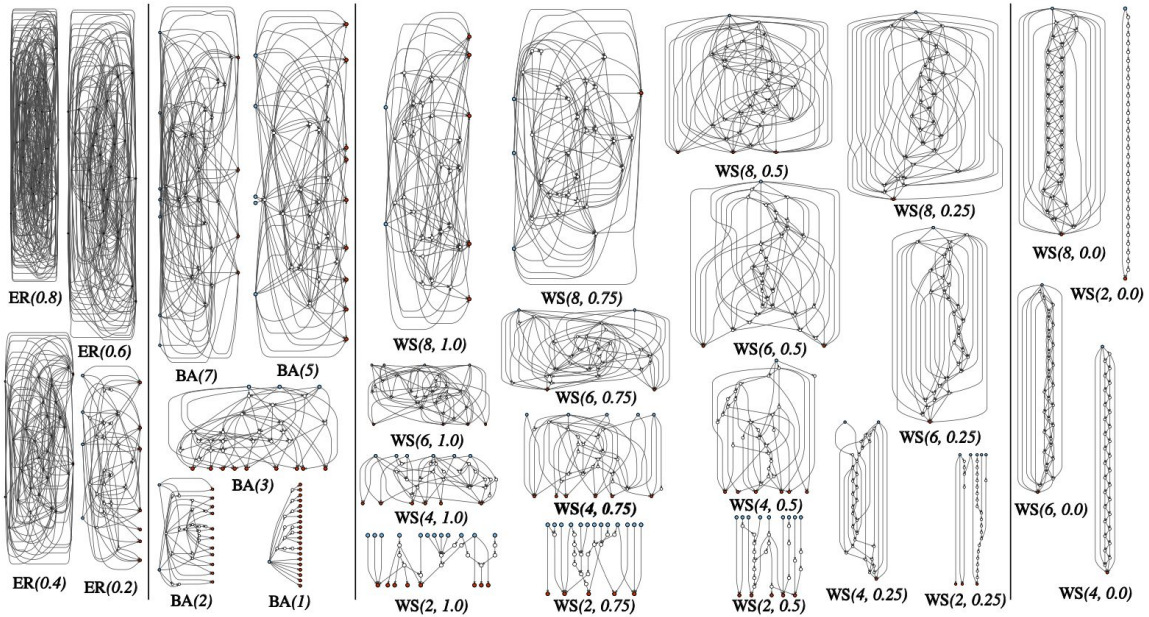


Figure 4. **Visualization of the random graphs generated by ER, BA, and WS**. Each plot represents one random graph instance sampled by the specified generator. The generators are those in Figure 3. The node count is N=32 for each graph. A blue/red node denotes an input/output node, to which an extra unique input/output node (not shown) will be added (see §3.2).

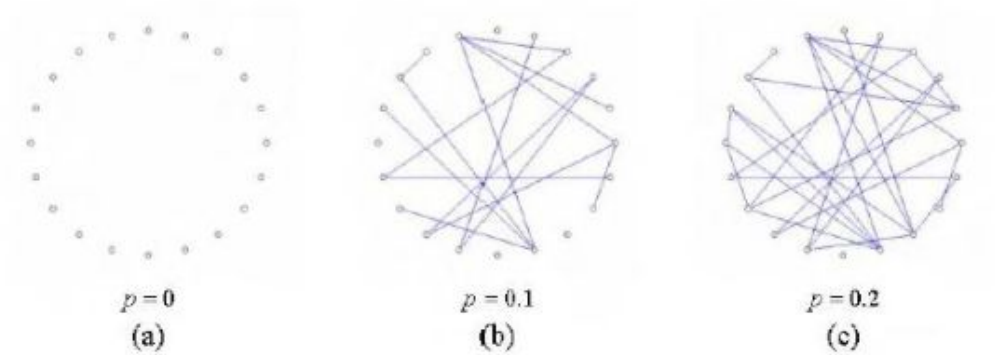
Random Graph Models

[Erdos-Renyi(ER)]

N개의 노드를 잘 깔아놓고, P의 확률로 2개의 노드를 연결할지 말지 결정

P가 유일한 parameter, ER(P) = ER generation model

N개의 노드로 구성된 어떠한 그래프도 이 ER 모델을 활용하여 만들 수 있다.

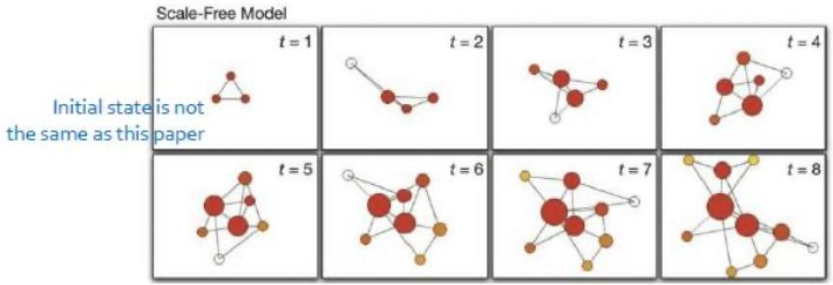


Random Graph Models

[Barabasi-Albert(BA)]

- 어떠한 edge도 없이 M개의 노드를 깔아 놓는다.
- 하나의 node를 더하면서 M개의 새로운 edges를 잇는다: 새로운 edge를 이을 때, 이미 존재하는 node v의 degree에 비례하는 확률로 잇는다. (많은 노드가 연결되어 있는 노드일수록 edge가 이어질 가능성이 크다). Edge가 겹치지 않게 긋는다.
- 전체 노드의 개수가 N개가 될 때까지 반복한다.
- 이 방법은 전 단계에서 그래프가 그려진 상태에서, 노드 추가 후 edges를 그리는 것이기 때문에 prior가 있다고 말할 수 있다.

BA(M) -> M(N-M) edges를 가진다.



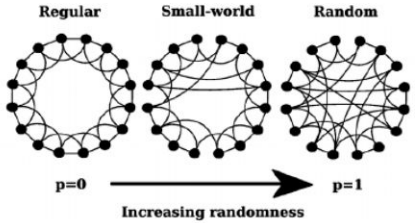
Random Graph Models

[Watts-Strogats(WS)]

"Small World" Model

High Clustering, Small diameter일수록 좋다. -> 클러스터링이 많을수록 좋다. real world에서 몇다리 건너지 않으면 바로 아는 사람들이 나오는 것처럼, 가장 멀리 거쳐야 하는 거리가 적을수록 좋다!

Regular Graph에서 시작: N개의 노드가 배치되어 있고, 각 노드의 양쪽으로 K/2개의 edges가 있음.
시계방향으로 회전하면서: 노드 v에 대해, v와 연결되어 있는 edge를 끊고, P의 확률로 나머지 노드들 중 아무거나와 연결 (v가 아닌 노드를 uniform하게 choose)
한 노드에 대해 K/2번 반복



Converting Undirected Graphs into DAGS

Index를 매겨서, 작은 index에서 큰 index로 방향이 향하게 함 -> cycle이 생기지 않음

*index 매기는 방법

ER = 랜덤하게 index를 매김
BA = 처음에 M개의 index를 매기고, 노드가 add되는 순서대로 index를 매김
WS = 시계방향 순서로 index를 매김

Randomly Wired Neural Networks (RWNN)

network	top-1 acc.	top-5 acc.	FLOPs (M)	params (M)
MobileNet [15]	70.6	89.5	569	4.2
MobileNet v2 [40]	74.7	-	585	6.9
ShuffleNet [54]	73.7	91.5	524	5.4
ShuffleNet v2 [30]	**74.9**	92.2	591	7.4
NASNet-A [56]	74.0	91.6	564	5.3
NASNet-B [56]	72.8	91.3	488	5.3
NASNet-C [56]	72.5	91.0	558	4.9
Amoeba-A [34]	74.5	92.0	555	5.1
Amoeba-B [34]	74.0	91.5	555	5.3
Amoeba-C [34]	**75.7**	**92.4**	570	6.4
PNAS [26]	74.2	91.9	588	5.1
DARTS [27]	73.1	91.0	595	4.9
RandWire-WS	**74.7**$_{\pm 0.25}$	**92.2**$_{\pm 0.15}$	583$_{\pm 6.2}$	5.6$_{\pm 0.1}$

Table 2. **ImageNet: small computation regime** (*i.e.*, <600M FLOPs). RandWire results are the mean accuracy (±std) of 5 random network instances, with WS$(4, 0.75)$. Here we train for 250 epochs similar to [56, 34, 26, 27], for fair comparisons.

network	top-1 acc.	top-5 acc.	FLOPs (B)	params (M)
ResNet-50 [11]	77.1	93.5	4.1	25.6
ResNeXt-50 [52]	78.4	94.0	4.2	25.0
RandWire-WS, C=109	**79.0**$_{\pm 0.17}$	**94.4**$_{\pm 0.11}$	4.0$_{\pm 0.09}$	31.9$_{\pm 0.66}$
ResNet-101 [11]	78.8	94.4	7.8	44.6
ResNeXt-101 [52]	79.5	94.6	8.0	44.2
RandWire-WS, C=154	**80.1**$_{\pm 0.19}$	**94.8**$_{\pm 0.18}$	7.9$_{\pm 0.18}$	61.5$_{\pm 1.32}$

Table 3. **ImageNet: regular computation regime** with FLOPs comparable to ResNet-50 (top) and to ResNet-101 (bottom). ResNeXt is the 32×4 version [52]. RandWire is WS$(4, 0.75)$.

Randomly Wired Neural Networks (RWNN)

- Wiring이 중요하다는 견해를 제시

- Random하게 초기화된 Connection을 가지고 Evaluation을 진행하였지만, 해당 Connection이 Optimal이라는 보장이 없음

- Connection의 Fine Tuning을 진행하지 않아 성능 향상의 여지가 있음

Optimal한 Connection이 무엇인가?

Finding Optimal Connection

Optimal Connection이란?

- Task에 따라 달라질 수 있음
- e.g.
  - Image Classification : Best Accuracy
  - Novelty Detection: Best AUPRC
  - and so on..

Sparse Network의 Connection에 대한 Optimization Problem

- Optimal Connection Search란
  Connection matrix를 parameter로 하여
  Task에 맞는 Score를 Maximize하는 것!

Genetic Algorithm

1. 소개

유전 알고리즘은 생물체가 환경에 적응하면서 진화해가는 모습을 모방하여 최적해를 찾아내는 검색 방법이다. 유전 알고리즘은 이론적으로 전역 최적점을 찾을 수 있으며, 수학적으로 명확하게 정의되지 않은 문제에도 적용할 수 있기 때문에 매우 유용하게 이용된다. 일반적으로 유전 알고리즘에 대해 알고리즘이라는 표현을 이용하지만, 유전 알고리즘은 특정한 문제를 풀기 위한 알고리즘이라기 보다는 최적화 문제를 풀기 위한 방법론에 가깝다. 즉, 모든 문제에 적용 가능한 하나의 알고리즘이나 소스 코드가 있는 것이 아니기 때문에 유전 알고리즘의 원리를 이해하고, 이를 자신이 원하는 문제에 적용할 수 있도록 하는 것이 중요하다. 유전 알고리즘을 정의하기 위해 아래와 같은 개념들을 정의한다.

Genetic Algorithm

- **염색체 (chromosome)**: 생물학적으로는 유전 물질을 담고 있는 하나의 집합을 의미하며, 유전 알고리즘에는 하나의 해 (solution)를 표현한다. 어떠한 문제의 해를 염색체로 인코딩 (encoding)하는 것에 대한 예시는 [5. 예제] 항목에서 자세하게 서술한다.
- **유전자 (gene)**: 염색체를 구성하는 요소로써, 하나의 유전 정보를 나타낸다. 어떠한 염색체가 [A B C]라면, 이 염색체에는 각각 A, B 그리고 C의 값을 갖는 3개의 gene이 존재한다.
- **자손 (offspring)**: 특정 시간 t에 존재했던 염색체들로부터 생성된 염색체를 t에 존재했던 염색체들의 자손이라고 한다. 자손은 이전 세대와 비슷한 유전 정보를 갖는다.
- **적합도 (fitness)**: 어떠한 염색체가 갖고 있는 고유값으로써, 해당 문제에 대해 염색체가 표현하는 해가 얼마나 적합한지를 나타낸다. 적합도에 대한 예시는 [5. 예제] 항목에서 자세하게 서술한다.

Genetic Algorithm

2. 알고리즘 구조

유전 알고리즘은 t에 존재하는 염색체들의 집합으로부터 적합도가 가장 좋은 염색체를 선택하고, 선택된 해의 방향으로 검색을 반복하면서 최적해를 찾아가는 구조로 동작한다. 유전 알고리즘의 동작을 단계별로 표현하면 아래와 같다.

1) 초기 염색체의 집합 생성

2) 초기 염색체들에 대한 적합도 계산

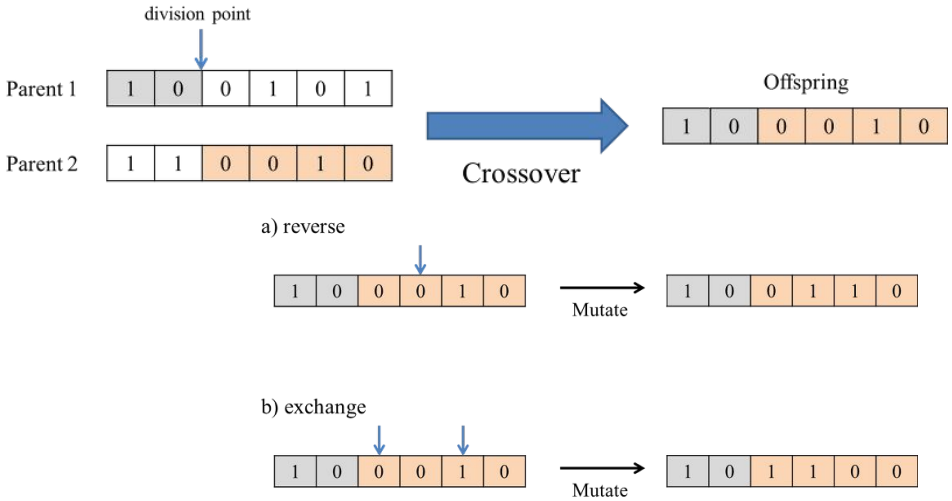
3) 현재 염색체들로부터 자손들을 생성

4) 생성된 자손들의 적합도 계산

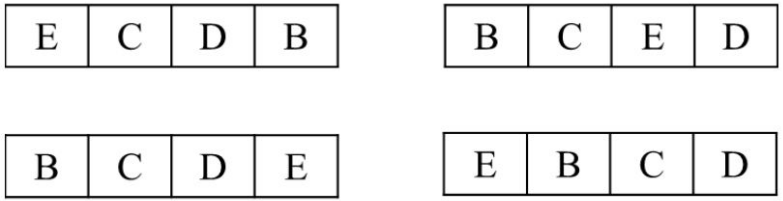
5) 종료 조건 판별

6-1) 종료 조건이 거짓인 경우, (3)으로 이동하여 반복

6-2) 종료 조건이 참인 경우, 알고리즘을 종료

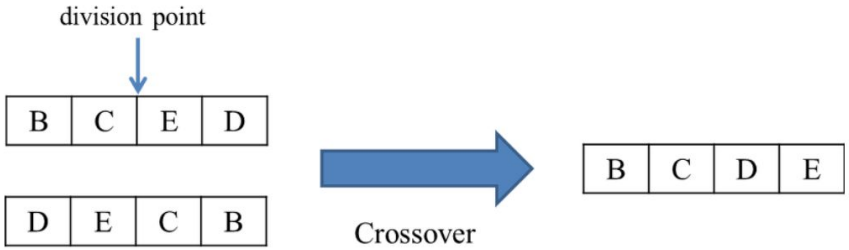


Genetic Algorithm

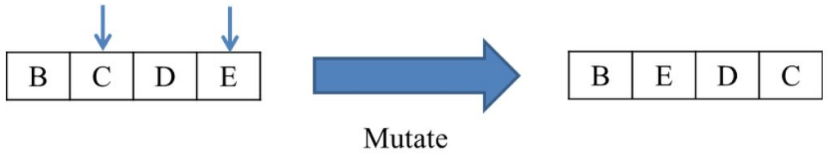


[그림 6] 초기 염색체의 집합

[그림 7] 초기 염색체들에 대한 적합도 계산



[그림 8] 두 염색체로부터 자손 생성



[그림 9] 돌연변이 연산

Genetic Algorithm

[그림 10] 새롭게 생성된 자손들에 대한 적합도 계산

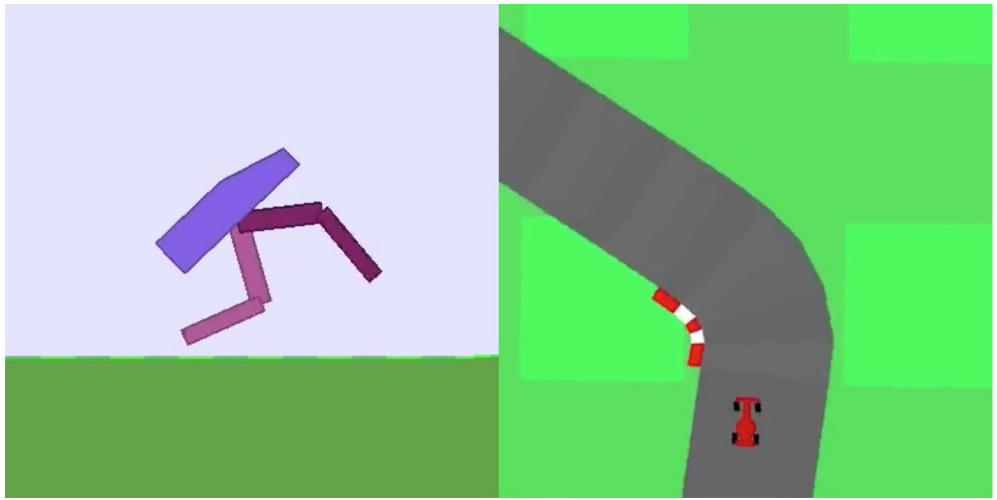
Weight Agnostic Neural Network (WANN)

[Introduction]

- 랜덤으로 Weight가 초기화 되어 있어도 특정한 Task를 수행할 수 있는 뉴럴넷 구조를 찾는 방법론
- 동물이 태어나자마자 걸을 수 있다는 것에서 영감을 받은 연구

[Idea]

- 모든 가중치를 동일하게 설정하여 Topology의 성능을 평가하여 기존의 NAS(Network Architecture Search)에 비해 빠르게 네트워크 아키텍처를 탐색함



https://arxiv.org/pdf/1906.04358.pdf

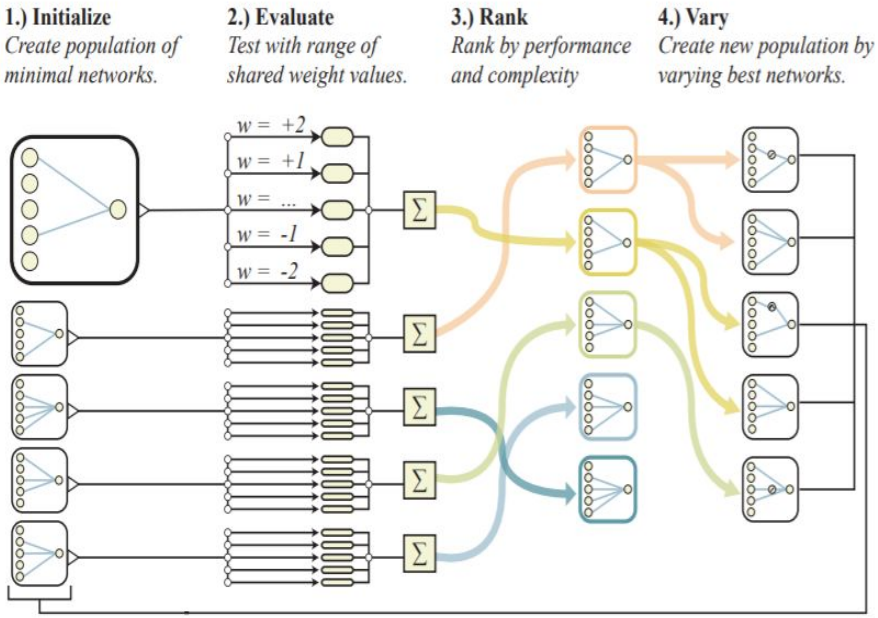
WANN

- weight training이 아닌 **weight random sampling**
- 퍼포먼스는 weight와는 무관하게 **네트워크의 형태(topology)**에 따라 결정됨
- 아키텍쳐 스스로 문제를 푸는 과정을 좀 더 효과적으로 만들기 위해, weight를 최소화하고자 → **하나의 공유된 weight** → computational cost 감소 → 계산이 더욱 빨라짐.

- 즉, weight training을 통해 optimal한 **weight를 찾는 과정이 costly**하다는 문제점을 개선하고자 함.
- 아키텍쳐가 weight에 관대하도록 진화하게 됨.
- WANN에서 **simple**의 정의
  - soft-weight sharing을 통해 모델을 단순화
  - weight의 정보량을 감소
  - weight의 search space를 단순화한다.

WANN Topology Search

1. 작은 단위의 뉴럴 네트워크를 형성.
2. 각 네트워크가 서로 다르지만 공유된 weight값들로 테스트됨
3. NEAT (Genetic Algorithm의 개선된 버전)으로 Topology 탐색
   a. 퍼포먼스와 모델의 복잡도에 따라 네트워크들의 순위가 매겨짐
   b. 가장 높은 랭크를 차지한 네트워크 모형이 변형되어 새로운 뉴럴 네트워크가 만들어짐
   c. 성능이 비슷할 경우에는 # of params. 가 작은 Topology를 선택



https://arxiv.org/pdf/1906.04358.pdf

WANN 성능 평가 & 실험계획

WANN	Test Accuracy
Random Weight	*82.0% ± 18.7%*
Ensemble Weights	*91.6%*
Tuned Weight	*91.9%*
Trained Weights	*94.2%*

ANN	Test Accuracy
Linear Regression	*91.6%* [62]
Two-Layer CNN	*99.3%* [15]

Figure 6: *Classification Accuracy on MNIST.*

A.2.4 Using backpropagation to fine-tune weights of a WANN.

We explored the use of autograd packages such as JAX [24] to fine-tune individual weights of WANNs for the MNIST experiment. Performance improved, but ultimately we find that black-box optimization methods such as CMA-ES and population-based REINFORCE can find better solutions for the WANN architectures evolved for MNIST, suggesting that the various activations proposed by the WANN search algorithm may have produced optimization landscapes that are more difficult for gradient-based methods to traverse compared to standard ReLU-based deep network architectures.

Fine-Tuning을 Backpropagation을 통해 했을때 성능이 별로였다

- REINFORCE라는 방법으로 업데이트
- 다양한 Activation Function을 사용하는 와중에 Gradient Vanishing이 일어나서?
- Input에서 Output으로 가는 Route의 길이가 제각각인 것도 학습을 방해하지 않을까?

https://arxiv.org/pdf/1906.04358.pdf

WANN 성능 평가 & 실험계획

WANN	Test Accuracy
Random Weight	*82.0% ± 18.7%*
Ensemble Weights	*91.6%*
Tuned Weight	*91.9%*
Trained Weights	*94.2%*

ANN	Test Accuracy
Linear Regression	*91.6%* [62]
Two-Layer CNN	*99.3%* [15]

Figure 6: *Classification Accuracy on MNIST.*

Backpropagation으로 Fine-Tuning했을때 얼마나 성능이 떨어질까?

- 제시한 성능은 REINFORCE를 쓰고도 Linear Regression과 크게 차이 나지 않으며 Two-Layer CNN보다 떨어짐
- Backpropagation 했을때는 어느 정도?

실험계획

- 실험에서 제시한 동일한 세팅으로 Backpropagation 후 성능 비교
- ReLU만 사용해서 동일한 Task에 대한 Topology 찾고 Accuracy, # of Params. 비교
- Gradient Vanishing을 크게 일으키는 Activation Function들 제외하고 탐색후 성능 비교

https://arxiv.org/pdf/1906.04358.pdf

WANN 성능 평가 & 실험계획

WANN	Test Accuracy
Random Weight	*82.0% ± 18.7%*
Ensemble Weights	*91.6%*
Tuned Weight	*91.9%*
Trained Weights	*94.2%*

ANN	Test Accuracy
Linear Regression	*91.6%* [62]
Two-Layer CNN	*99.3%* [15]

Figure 6: *Classification Accuracy on MNIST.*

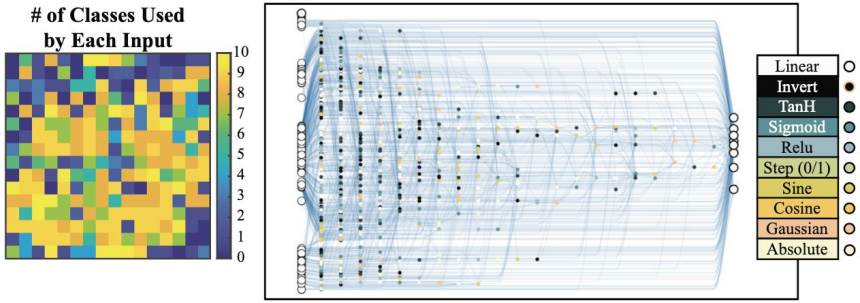


Figure 7: *MNIST classifier network (1849 connections)*

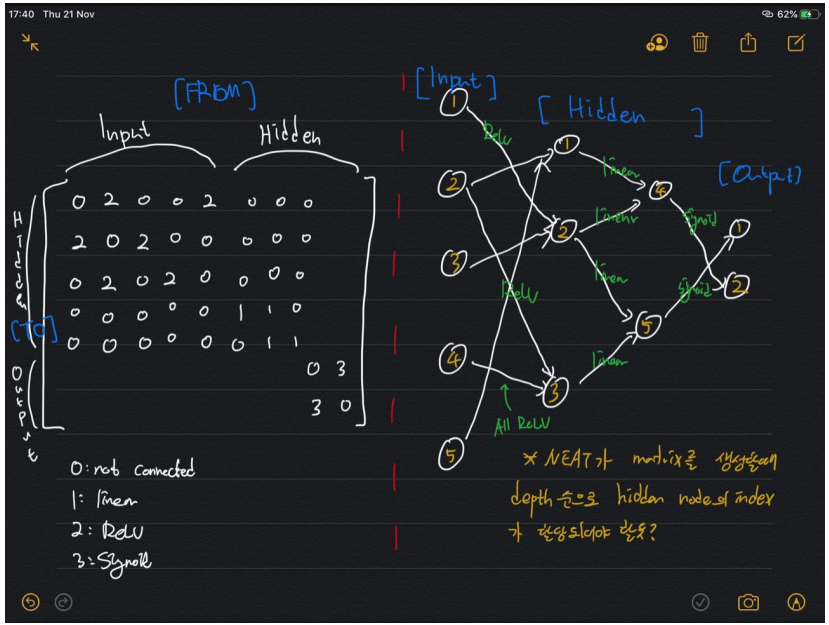
실험계획

- 1849개의 Connection을 가진 Fully connected neural network와는 얼마나 성능 차이가 나는지?

https://arxiv.org/pdf/1906.04358.pdf

WANN PyTorch Implementation

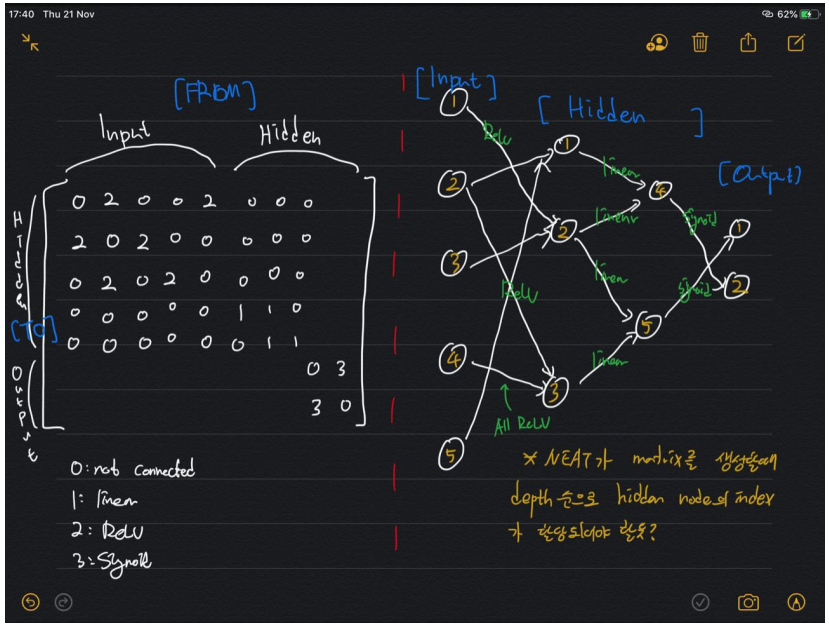
1. Class converts matrix representing graph to PyTorch NN model




```
[2]: class MatrixForWANN() :
         def __init__(self, mat, in_dim, out_dim) :

             # get when intiliazed
             self.mat = mat
             self.in_dim = in_dim
             self.out_dim = out_dim

             # calculate
             self.num_hidden_nodes = self.mat.shape[1]-self.in_dim

             # when matrix has hidden layer
             if mat.shape[1] != in_dim :
                 self.hidden_dim = self.get_hidden_dim()
             else :
                 self.hidden_dim = []

         def get_hidden_dim(self):
             in_dim = self.in_dim
             out_dim = self.out_dim
             mat_mask = self.mat
             hidden_dim_list = []
             start_col_idx = 0
             finish_col_idx = in_dim-1

             while(True):

                 if finish_col_idx >= mat_mask.shape[1]:
                     break

                 for i in range(start_col_idx, len(mat_mask)):
                     if (mat_mask[i,start_col_idx:(finish_col_idx + 1)].sum() == 0):
                         hidden_dim = i - sum(hidden_dim_list)
                         hidden_dim_list += [hidden_dim]
                         start_col_idx = finish_col_idx + 1
                         finish_col_idx += i
                         break
             return hidden_dim_list
```

WANN PyTorch Implementation

1. Class converts matrix representing graph to PyTorch NN model




```python
class WANNFCN(nn.Module) :
    def __init__(self, mat_wann, activations) :
        super(WANNFCN, self).__init__()
        self.mat = mat_wann.mat
        self.in_dim = mat_wann.in_dim
        self.out_dim = mat_wann.out_dim
        self.num_hidden_nodes = mat_wann.num_hidden_nodes
        self.hidden_dim = mat_wann.hidden_dim

        self.activations = activations

        self.nodes = {}
        '''
        nodes라는 dictionary 안에 아래와 같이 저장됨
        'hidden_1' : 해당 노드
        ...
        'output_1' : 해당 output 노드, hidden node로부터 연결되어있음
        'output_2' : 해당 output 노드, input node, hidden node로부터 연결되어있음
        '''
    def forward(self, x) :
        # hidden node가 한개라도 있을때
        if self.num_hidden_nodes != 0 :
            self.to_hidden(x)
        # output은 반드시 있음
        outputs = self.to_output(x)
        print(self.nodes)

        return outputs

    def to_hidden(self, x) :

        return

    def to_output(self, x) :
        # input layer와 모든 이전 hidden layer를 탐색
        # 그렇지 않으면 skip connection을 놓칠수 있음
        # 모든 node와 connection은 dictionary self.nodes에 저장

        outputs = 'test'
        return outputs
```