연구 배경

- 지난 몇 년 동안 음성 인식, 이미지 인식 및 기계 번역과 같은 많은 까다로운 응용 분야에서 심층 신경망의 많은 성공이 있었음.

- 이러한 성공과 함께 SIFT, HOG로부터 AlexNet, VGGNet, GoogleNet, ResNet이 된 것처럼 특징 설계(feature designing)에서 아키텍처 설계(architecture designing)로 패러다임이 전환되었음.

- 하지만 **아키텍처 설계는 여전히 많은 전문 지식과 충분한 시간을 요구함.**

연구 동기

- 하이퍼파라미터 최적화는 기계 학습에서 중요한 연구 주제이며 실제로 널리 사용됨. 성공한 논문 방법은 고정 길이 공간에서만 모델을 검색한다는 점에서 여전히 제한적임. 즉, 네트워크의 구조와 연결성을 지정하는 가변 길이 구성을 생성하도록 요청하기가 어려움. 고정되지 않은 길이의 아키텍처를 검색할 수 있는 베이지안 최적화 방법이 있지만, 덜 일반적이고 덜 유연함.
- 최신 신경 진화 알고리즘은 새로운 모델을 구성하는 데 훨씬 더 유연하지만 대규모에서는 일반적으로 덜 실용적임.
- 이러한 문제점을 해결하여 최적화된 아키텍처를 설계할 수 있는 프레임워크를 만들고자 NAS를 제안하게 되었음.

제안 아이디어

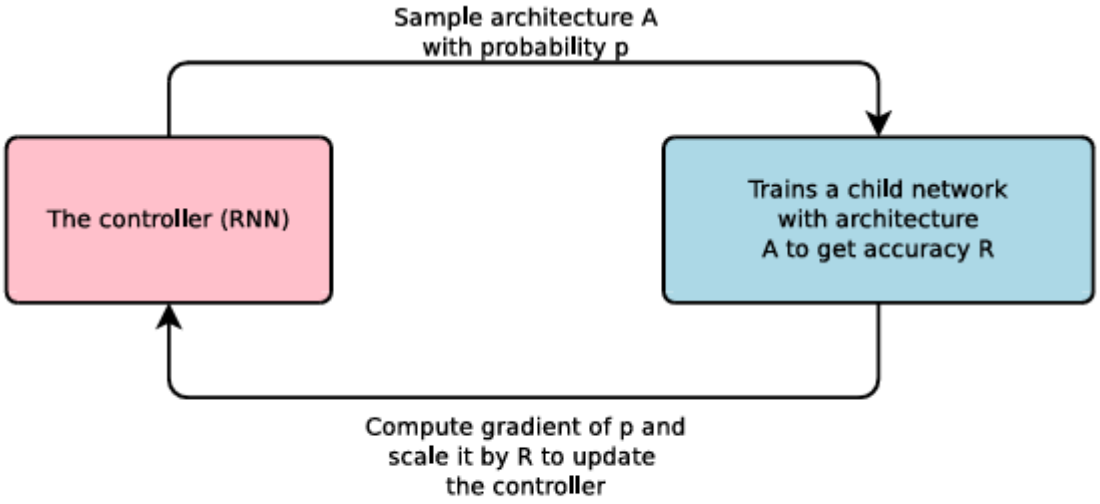


Figure 1: An overview of Neural Architecture Search.

1. 에이전트 담당하는 컨트롤러 (Controller as an Agent): 아키텍처를 생성하는 주체를 '컨트롤러'라고 부르는 에이전트로 정의합니다. 이 컨트롤러는 또 다른 신경망(RNN)으로 구현됩니다.

2. 순차적 행동으로서의 아키텍처 생성 (Architecture Generation as Sequential Actions): 신경망 아키텍처는 '필터 크기: 5x5', '필터 개수: 32' 등과 같은 하이퍼파라미터들의 순차적인 조합으로 표현될 수 있습니다. 컨트롤러는 이러한 하이퍼파라미터들을 순서대로 예측(행동)하여 하나의 완전한 아키텍처를 생성합니다.

3. 보상으로서의 성능 (Performance as Reward): 컨트롤러가 생성한 아키텍처(이를 '자식 네트워크'라 칭함)를 실제로 학습시킨 후, 검증 데이터셋에서의 정확도를 측정합니다. 이 정확도가 컨트롤러에게 주어지는 '보상(Reward)'이 됩니다.

4. 정책 경사도를 이용한 학습 (Learning with Policy Gradient): 컨트롤러는 더 높은 보상(더 좋은 성능의 아키텍처)을 생성했던 행동들의 확률은 높이고, 낮은 보상을 생성했던 행동들의 확률은 낮추는 방향으로 학습합니다. 이 학습 과정에는 강화학습의 핵심 알고리즘인 'REINFORCE' (정책 경사도 이론 기반)가 사용됩니다.

제안 방안 오버뷰

1) 컨트롤러

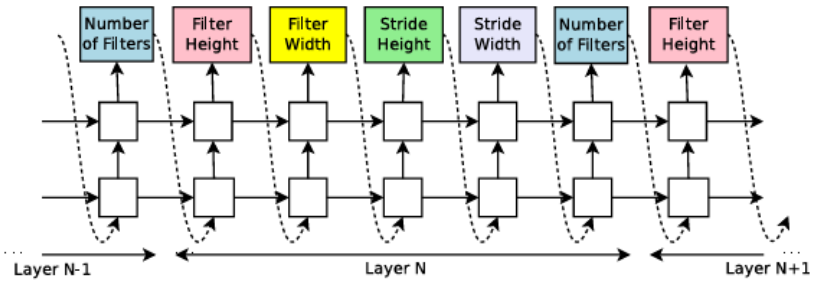


Figure 2: How our controller recurrent neural network samples a simple convolutional network. It predicts filter height, filter width, stride height, stride width, and number of filters for one layer and repeats. Every prediction is carried out by a softmax classifier and then fed into the next time step as input.

- Conv Layer만 있는 FFNN을 예측하고 싶다고 가정하면 컨트롤러를 사용하여 하이퍼파라미터를 토큰 시퀀스로 생성할 수 있음 (Conv layer인 경우 [filter 개수, filer 크기, stride 크기]와 같이 하이퍼파라미터를 토큰 시퀀스로 생성 가능).

- 실험에서 아키텍처 생성 프로세스는 레이어 수가 특정 값을 초과하면 중지됨. 이 값은 훈련이 진행됨에 따라 증가하는 일정을 따름 (실험에서 6개 layer부터 시작해서 점차 layer 증가하는 방식으로 진행함).

- 모든 예측은 softmax classifier를 거쳐서 나오게 되고 이 예측 값은 다음 step의 input으로 들어가게 됨 (예를 들어, 필터 개수를 예측한다 하면 가능한 값으로 [3개, 5개, 8개]로 지정할 수 있고, softmax classifier를 통해 각각의 확률로 [0.5, 0.3, 0.2]와 같이 나오게 됨. 이 확률 분포로부터 필터 개수를 어떤 값으로 할지 샘플링을 하면 그것이 필터 개수에 대한 예측 값이 됨. 이 예측 값은 다음 step의 input으로 들어가게 됨)

- 하이퍼파라미터로 무엇을 세팅할 것인지, 각각의 하이퍼파라미터는 어떤 값들이 가능한지 설정하는 것을 'search space 정의'한다고 하며, 실험에서는 filter 크기로 어떤 값이 가능한지, stride 크기로 어떤 값이 가능하지 미리 지정했음.

- 컨트롤러가 아키텍처 생성을 마치면(a1:T가 다 만들어지면) 이 아키텍처를 가진 신경망이 구축되고 훈련됨 (하나의 자식 네트워크를 검증 데이터셋을 통해 훈련). 이 자식 네트워크가 수렴 시, 검증 데이터셋에 대한 네트워크의 정확도(이 정확도 R을 보상으로 함)가 기록됨. 컨트롤러의 매개변수 θc는 이 정확도 R을 최대화하기 위해 업데이트 됨.

2) REINFORCE로 훈련

- 컨트롤러가 예측하는 토큰 목록은 자식 네트워크의 아키텍처를 설계하기 위한 작업 목록 a1:T ([필터 개수, 필터 크기, stride 크기]라면 [3개, 3x3, 1x1]식의 토큰 목록으로 볼 수 있고, 이게 하나의 자식 네트워크의 아키텍처로도 볼 수도 있음)로 볼 수 있음. 이 때 T는 컨트롤러가 예측해야 하는 총 하이퍼파라미터의 개수를 의미함. 이 자식 네트워크가 검증 데이터셋을 통해 수렴 시 정확도 R을 보상으로 하여 강화 학습을 통해

θc를 업데이트 할 수 있음. 즉, 컨트롤러에 J(θc)로 표시되는 예상 보상을 최대화하도록 하는 것임.

$$J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R]$$

-> 컨트롤러 매개변수 θc가 생성하는 모든 가능한 신경망 구조 각각에 대해, 각 구조가 생성되었을 때 그 구조의 성능(정확도 R)에 대한 기댓값

- 보상 신호 R은 미분할 수 없으므로 정책 경사도 방법을 사용하여 θc를 반복적으로 업데이트 해야 함. 이 연구에서는 Williams의 REINFORCE 규칙을 사용함.

$$\nabla_{\theta_c} J(\theta_c) = \sum_{t=1}^{T} E_{P(a_{1:T};\theta_c)}\big[\nabla_{\theta_c} \log P(a_t|a_{(t-1):1};\theta_c)R\big]$$

-> a1:T가 만들어지는 에피소드에서 이전 예측 값들 a_(t-1):1 상태에서 a_t를 예측하는 행동을 할 확률에 보상 R을 곱해서 가치 함수로 만들고, 그거에 대한 기댓값

- 문제는 모든 경우의 a1:T를 다 탐색하기는 어려우므로 m개만 샘플링해서 평균을 취하는 경험적 근사식을 사용함.

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t|a_{(t-1):1};\theta_c)R_k$$

-> m개의 경우만 샘플링해서 평균을 한 것을 사용할 것임. 이 때 R_k는 k번째 a1:T의 경우에서의 정확도를 의미함.

- 문제는 저 식으로 하다간 gradient 추정 값의 분산이 너무 큼. 따라서 이 분산을 줄이기 위해서 baseline function으로 b를 추가함.

    1. 보상 R 자체가 매우 노이즈가 심함. 아키텍처의 아주 작은 변화(예를 들어 필터 개수를 32에서 48로 변경)가 최종 정확도에 큰 영향을 미칠 수 있음.

    2. 조합할 수 있는 a1:T의 경우는 엄청 많지만 그 중 적은 m개만큼 샘플링해서 진행함. 이 m개가 전체 search space를 대표하기엔 부족함. 따라서 우연히 좋은 a1:T만 샘플링되는 경우와 우연히 나쁜 a1:T만 샘플링되는 경우가 생길 수 있고, 그만큼 추정 값 또한 변동성이 크다는 의미임.

    3. 최종 보상 R이 좋았을 때, 여러 개의 하이퍼파라미터 선택들 전부 '결과가 좋았으니 이 선택 하나하나 다 좋은 선택이었을 것'이라고 가정하고 긍정적으로 강화됨. 선택된 하이퍼파라미터들 중 일부는 실제로는 성능을 저해한 나쁜 선택이

었음에도 불구하고 긍정적으로 강화될 수 있고, 그 반대도 가능함. 이런 문제로 인해 경사 추정치에 엄청난 노이즈가 발생하여 분산을 높일 가능성이 있음.

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T}\nabla_{\theta_c}\log P(a_t|a_{(t-1):1};\theta_c)(R_k-b)$$

-> 보상 R의 절대적인 크기가 아닌, baseline function b를 통해 평균보다 얼마나 더 나은가를 기준으로 업데이트 하도록 만듦. 이 때 b는 이전 아키텍처의 정확도들에 대한 지수 이동 평균 값을 의미함.

3) 병렬 처리 및 비동기 업데이트

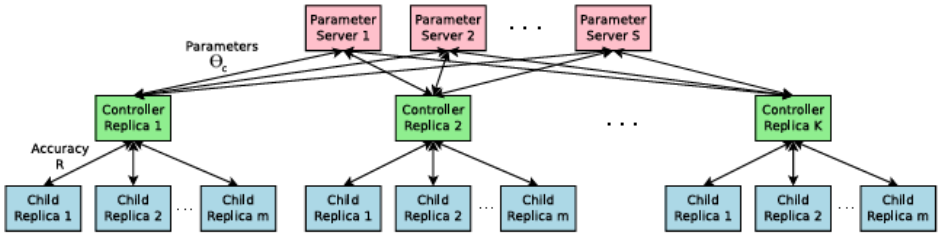


Figure 3: Distributed training for Neural Architecture Search. We use a set of S parameter servers to store and send parameters to K controller replicas. Each controller replica then samples m architectures and run the multiple child models in parallel. The accuracy of each child model is recorded to compute the gradients with respect to θ_c, which are then sent back to the parameter servers.

- 컨트롤러 매개변수 θc에 대해 업데이트 하려면 m개의 자식 네트워크에 대해서 각각의 자식 네트워크마다 수렴하도록 훈련하는 과정이 필요함. 이 훈련하는 과정에서 몇 시간이 걸릴 수 있으므로 분산 훈련 및 비동기 매개변수 업데이트를 사용하여 컨트롤러의 학습 프로세스 속도를 높임.

1. S개의 parameter server에는 하나의 θc가 분산되어 저장됨.

2. K개의 controller replica는 각각 독립적으로 θc에 대해 학습 진행(S개의 parameter server에 저장되어 있는 θc를 pull하고 진행하는 느낌)

3. 각각의 controller replica는 θc로 만들 수 있는 모든 자식 네트워크들 중 m개 만큼 샘플링 해서 gradient 계산.

4. m개의 자식 네트워크는 병렬적으로 훈련 데이터셋을 가지고 훈련을 진행함. 이 때 훈련 에폭 수는 따로 지정함.(병렬 처리)

5. 검증 데이터셋을 통해 각각의 자식 네트워크에 대응하는 정확도 R 구함.

6. m개의 자식 네트워크로부터 나온 m개의 정확도 R을 가지고 controller replica는 gradient 계산함.

7. 각각의 controller replica는 다른 controller replica가 gradient 계산을 완료했던 안했던 상관없이 먼저 gradient 계산이 완료되자마자 바로 그 gradient를 S개의 parameter server로 보냄. (비동기 업데이트) (임의의 controller replica가 S개의 parameter server로 push하는 느낌)

8. S개의 parameter server는 controller replica가 보낸 gradient를 그대로 반영해서 θc를 업데이트 함.

9. gradient 보냈던 controller replica는 다시 S개의 parameter server로 접근해서 최근 버전의 θc를 가져와서 3 부터 다시 시작함. (다시 pull 하는 느낌)

- Q. 그러면 parameter server에서 θc 업데이트 할 때 반영하는 gradient 값이 최신 버의 θc로부터 나온 gradient가 아니라 그보다 오래된 버전의 θc로부터 나온 gradient라면 문제 되는거 아님?

- A. 오래된 버전을 기준으로 계산된 경사도를 stale gradient 라고 함. 이걸로 인해 노이즈 발생할 수도 있는 것이 비동기 업데이트 방식의 문제점임. 그러나 전체적으로 보면 대다수가 가리키는 평균적인 방향은 올바른 방향이고, 동기 업데이트 방식보다 속도적인 면에서 엄청나게 이득이기 때문에 비동기 업데이트 방식을 사용함.

4) 스킵 연결 및 기타 레이어 유형으로 아키텍처 복잡성 증가

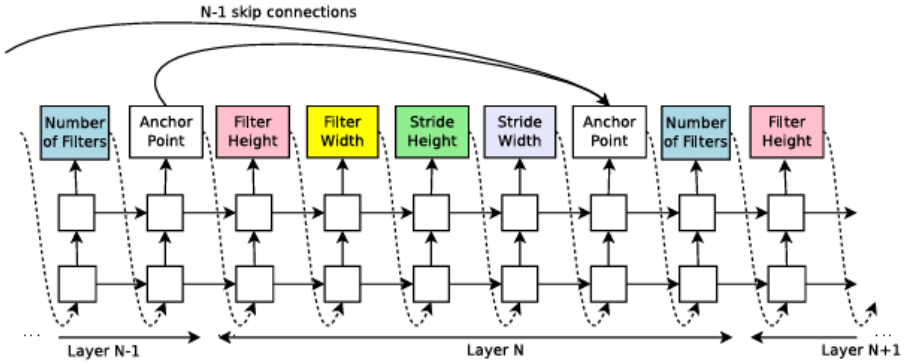


Figure 4: The controller uses anchor points, and set-selection attention to form skip connections.

- GoogleNet이나 ResNet과 같은 최신 아키텍처에서 사용되는 skip connections 또는 branching layers도 search space에 도입할 수 있는 방법에 대해 얘기함.

- 레이어 N에서, 연결할 수 있는 N-1개의 이전 레이어들을 나타내기 위해 N-1개의 콘텐츠 기반 시그모이드를 가진 Anchor Point를 추가함. 각 시그모이드는 컨트롤러의 현재 은닉 상태 h_i와 이전 N-1 Anchor Point의 은닉 상태의 함수 h_j 이다.

$$\mathrm{P}(\text{Layer j is an input to layer i}) = \mathrm{sigmoid}(v^{\mathrm{T}}\tanh(W_{prev} * h_j + W_{curr} * h_i)),$$

-> 레이어 j가 레이어 i의 입력으로 들어갈 확률을 다음의 sigmoid를 통해 나타냄. 이 때 W_prev와 W_curr, v는 학습 가능한 파라미터임. 이 때 j는 0~N-1 범위임.

-> 즉, 이전 레이어들 중 어떤 레이어를 현재 레이어에 연결할 것인지 결정을 이 sigmoid를 통해 샘플링해서 결정하겠다는 의미.

- 한 레이어에 여러 개의 이전 레이어들을 입력 받을 수 있음. 이 경우에는 모든 입력 레이어의 결과들을 깊이 차원에서 concat 함.

- skip connections는 컴파일 실패를 유발할 수 있음

    1. 레이어가 입력 받는 이전 레이어들이 하나도 없다면 이미지가 입력 레이어로 사용됨.

    2. 최종 레이어에서 한번도 연결되지 않은 고립된 모든 레이어의 출력을 가져와 이 최종 은닉 상태를 softmax classifier로 보내기 전에 concat 하고 보냄.

    3. 연결한 입력 레이어의 크기가 다른 경우 크기가 더 작은 레이어를 0으로 padding해서 크기를 맞춰줌.

- 사실 예측 중 하나로 학습률을 추가할 수 있음. 또한 풀링, 로컬 대비 정규화, 배치 정규화를 예측하는 것도 가능. 대신 그만큼 컨트롤러에 단계 추가하고 그와 관련된 하이퍼파라미터도 추가해야 함.

5) 순환 셀 아키텍처 생성

- 지금까지는 CNN과 같은 아키텍처를 생성하는 것에 대해 다룸. 하지만 RNN이나 LSTM과 같은 순환 셀 아키텍처를 예측하기 위해서는 수정해야 할 필요가 있음.

- 순환 셀을 예측하기 위해서는 컨트롤러는 매 time step t마다 x_t와 h_t-1을 입력으로 사용하는 h_t에 대한 함수 형식을 찾아야 함. 가장 간단한 방법은 기본 순환 셀의 공식 사용하거나, 더 복잡한 공식은 널리 사용되는 LSTM 순환 셀 공식임.

- 기본 RNN 및 LSTM 셀에 대한 계산은 x_t와 h_t-1을 입력으로 받아 최종 출력으로 h_t를 생성하는 일종의 트리 구조로 일반화될 수 있음.

- 컨트롤러는 트리의 각 노드마다 조합 방법(덧셈, 요소별 곱셈, ...)과 활성화 함수(tanh, ReLU, ...)를 결정하여 두 입력을 병합하고 하나의 출력을 생성해야 함.

- 이를 위해서 컨트롤러는 트리의 노드를 순서대로 인덱싱(리프 노드부터 인덱싱)하여 컨트롤러는 각 노드를 하나씩 방문하고 필요한 하이퍼파라미터를 label로 지정할 수 있도록 함.

- LSTM 셀의 구성을 생각하면 메모리 상태를 나타내기 위해 셀 변수 C_t-1과 C_t도 필요

함. 이러한 변수를 통합하려면 컨트롤러는 트리의 어떤 노드에 이 C_t-1이나 C_t를 연결할지 예측해야 함. 이러한 예측은 컨트롤러의 마지막 두 블록에서 수행될 수 있음.

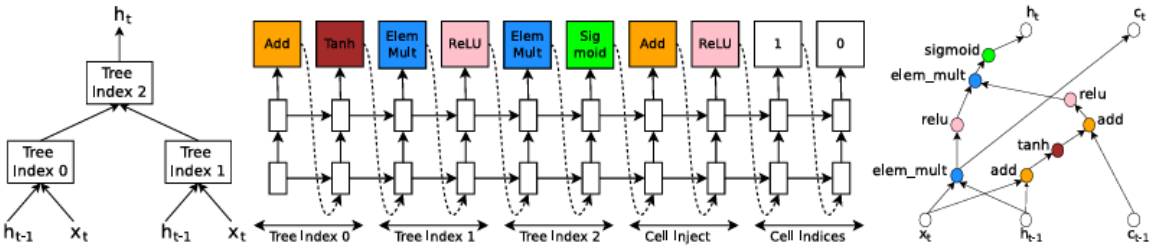
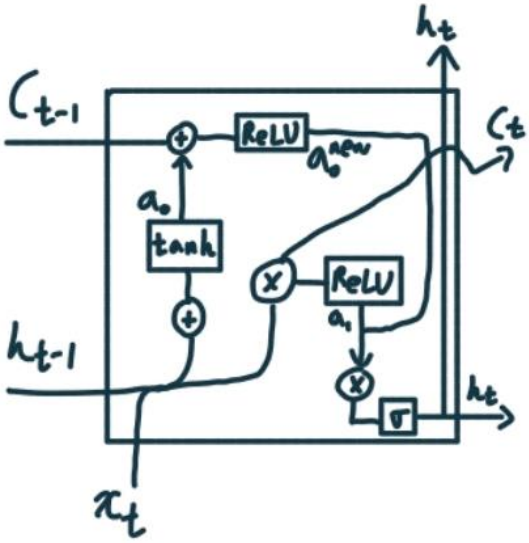


Figure 5: An example of a recurrent cell constructed from a tree that has two leaf nodes (base 2) and one internal node. Left: the tree that defines the computation steps to be predicted by controller. Center: an example set of predictions made by the controller for each computation step in the tree. Right: the computation graph of the recurrent cell constructed from example predictions of the controller.

- 왼쪽의 트리 구조에 대해서 컨트롤러는 예측하려고 함.
- 리프 노드부터 인덱싱해서 0, 1, 2 총 3개의 노드가 존재함.
- 컨트롤러는 이 노드 순서대로 각 노드마다 어떤 조합 방식, 어떤 활성화 함수를 사용할 것인지 예측함.
- 맨 마지막 두 블록은 각각 C_t는 어떤 노드와 연결할지 예측, C_t-1은 어떤 노드와 연결할지 예측 하는 블록임.
- Cell Inject는 C_t-1과 연결할 시 어떤 조합 방식, 어떤 활성화 함수로 연결할 것인지 예측하는 부분임.
- 오른쪽 그림은 컨트롤러가 예측한 순환 셀 구조를 그래프로 나타낸 것인데 굳이 그림으로 표현한다면 다음 그림과 같이 표현할 수 있을 것 같음.



실험 세팅 및 결과

- 제안된 프레임워크의 유효성을 검증하기 위해 두 가지 대표적인 벤치마크 데이터셋에 대한 실험을 수행함.

1. CIFAR-10 이미지 분류 (CNN 탐색)

- 데이터셋: 모든 이미지를 whitening 하고, 각 이미지를 upsampling한 다음 이 upsampling된 이미지에서 랜덤으로 32x32 crop 선택함. 이 32x32 크롭 이미지에 대해 랜덤으로 수평 뒤집기를 함.

- search space: 비선형성으로 수정된 선형 단위, 배치 정규화 및 레이어 간 skip connections가 있는 Convolution 아키텍처로 구성됨. 모든 Conv Layer에 대해 컨트롤러는 필터높이로 [1, 3, 5, 7], 필터 너비로 [1, 3, 5, 7], 필터 개수로 [24, 26, 48, 64] 중 하나를 선택해야 함. Stride 경우 Stride=1로 고정하는 실험과 컨트롤러가 [1, 2, 3]에서 하나 예측하도록 하는 실험의 두 가지 세트를 수행함.

- 훈련 세부 정보: 컨트롤러는 각 레이어에 35개의 은닉 유닛이 있는 2계층 LSTM임. ADAM 옵티마이저로 0.0006의 학습률로 훈련됨. 컨트롤러의 가중치는 -0.08~0.08 사이에서 균일하게 초기화 됨. 분산 훈련의 경우, parameter server 20개, controller replica 100개, child network 8개로 설정함.

- child network는 50 epochs 동안 훈련됨. 보상 R은 마지막 5 epochs의 최대 검증 정확도를 세제곱한 값임. Child network 훈련 설정은 학습률=0.1이고 가중치 감쇠 1e-4, 0.9 momentum을 가진 momentum 옵티마이저 사용하고, Nesterov Momentum 사용함.

- 컨트롤러 훈련 중에 훈련이 진행됨에 따라 자식 네트워크의 레이어 수를 늘리는 schedule을 사용함. 컨트롤러는 6개 레이어에서 시작하여 1600개 샘플마다 레이어 2개 늘림.

- 결과: DenseNet 최고 성능 3.46% 오류율과 필적하는 성능 3.65% 오류율을 달성함. 이는 NAS가 인간 전문가 수준의 아키텍처를 발견할 수 있음을 증명한 것임.

Model	Depth	Parameters	Error rate (%)
Network in Network (Lin et al., 2013)	-	-	8.81
All-CNN (Springenberg et al., 2014)	-	-	7.25
Deeply Supervised Net (Lee et al., 2015)	-	-	7.97
Highway Network (Srivastava et al., 2015)	-	-	7.72
Scalable Bayesian Optimization (Snoek et al., 2015)	-	-	6.37
FractalNet (Larsson et al., 2016)	21	38.6M	5.22
with Dropout/Drop-path	21	38.6M	4.60
ResNet (He et al., 2016a)	110	1.7M	6.61
ResNet (reported by Huang et al. (2016c))	110	1.7M	6.41
ResNet with Stochastic Depth (Huang et al., 2016c)	110	1.7M	5.23
	1202	10.2M	4.91
Wide ResNet (Zagoruyko & Komodakis, 2016)	16	11.0M	4.81
	28	36.5M	4.17
ResNet (pre-activation) (He et al., 2016b)	164	1.7M	5.46
	1001	10.2M	4.62
DenseNet ($L = 40, k = 12$) Huang et al. (2016a)	40	1.0M	5.24
DenseNet($L = 100, k = 12$) Huang et al. (2016a)	100	7.0M	4.10
DenseNet ($L = 100, k = 24$) Huang et al. (2016a)	100	27.2M	3.74
DenseNet-BC ($L = 100, k = 40$) Huang et al. (2016b)	190	25.6M	3.46
Neural Architecture Search v1 no stride or pooling	15	4.2M	5.50
Neural Architecture Search v2 predicting strides	20	2.5M	6.01
Neural Architecture Search v3 max pooling	39	7.1M	4.47
Neural Architecture Search v3 max pooling + more filters	39	37.4M	3.65

Table 1: Performance of Neural Architecture Search and other state-of-the-art models on CIFAR-10.

2. Penn TreeBank 언어 모델링 (RNN Cell 탐색)

- 결론: 신경망 구조 탐색으로 찾은 모델은 이 데이터셋의 다른 최첨단 모델보다 성능이 뛰어나며, 최고의 모델 중 하나는 거의 3.6 복잡도의 이득을 달성합니다. 우리 셀이 더 좋을 뿐만 아니라 64 복잡도를 달성하는 모델은 이전 최고의 네트워크가 타임 스텝당 셀을 10번 실행해야 하기 때문에 두 배 이상 빠릅니다. (Perplexity는 모델이 평균적으로 다음 단어를 예측 할 때, 몇 개의 단어들 사이에서 선택하는지 지표)

Model	Parameters	Test Perplexity
Mikolov & Zweig (2012) - KN-5	2M‡	141.2
Mikolov & Zweig (2012) - KN5 + cache	2M‡	125.7
Mikolov & Zweig (2012) - RNN	6M‡	124.7
Mikolov & Zweig (2012) - RNN-LDA	7M‡	113.7
Mikolov & Zweig (2012) - RNN-LDA + KN-5 + cache	9M‡	92.0
Pascanu et al. (2013) - Deep RNN	6M	107.5
Cheng et al. (2014) - Sum-Prod Net	5M‡	100.0
Zaremba et al. (2014) - LSTM (medium)	20M	82.7
Zaremba et al. (2014) - LSTM (large)	66M	78.4
Gal (2015) - Variational LSTM (medium, untied)	20M	79.7
Gal (2015) - Variational LSTM (medium, untied, MC)	20M	78.6
Gal (2015) - Variational LSTM (large, untied)	66M	75.2
Gal (2015) - Variational LSTM (large, untied, MC)	66M	73.4
Kim et al. (2015) - CharCNN	19M	78.9
Press & Wolf (2016) - Variational LSTM, shared embeddings	51M	73.2
Merity et al. (2016) - Zoneout + Variational LSTM (medium)	20M	80.6
Merity et al. (2016) - Pointer Sentinel-LSTM (medium)	21M	70.9
Inan et al. (2016) - VD-LSTM + REAL (large)	51M	68.5
Zilly et al. (2016) - Variational RHN, shared embeddings	24M	66.0
Neural Architecture Search with base 8	32M	67.9
Neural Architecture Search with base 8 and shared embeddings	25M	64.0
Neural Architecture Search with base 8 and shared embeddings	54M	62.4

Table 2: Single model perplexity on the test set of the Penn Treebank language modeling task. Parameter numbers with ‡ are estimates with reference to Merity et al. (2016).

논문의 공헌

- 좋은 신경망 아키텍처를 자동으로 찾기 위한 새로운 연구 방향을 제시함.

- NAS를 통해 구성된 아키텍처가 실제로 유효한 성능을 낸다는 점을 증명함.