Language Models with Transformers

Chenguang Wang **Mu Li** **Alexander J. Smola**
Amazon Web Services
{chgwang, mli, smola}@amazon.com

Abstract

The Transformer architecture is superior to RNN-based models in computational efficiency. Recently, GPT and BERT demonstrate the efficacy of Transformer models on various NLP tasks using pre-trained language models on large-scale corpora. Surprisingly, these Transformer architectures are suboptimal for language model itself. Neither self-attention nor the positional encoding in the Transformer is able to efficiently incorporate the word-level sequential context crucial to language modeling.

In this paper, we explore effective Transformer architectures for language model, including adding additional LSTM layers to better capture the sequential context while still keeping the computation efficient. We propose Coordinate Architecture Search (CAS) to find an effective architecture through iterative refinement of the model. Experimental results on the PTB, WikiText-2, and WikiText-103 show that CAS achieves perplexities between 20.42 and 34.11 on all problems, i.e. on average an improvement of 12.0 perplexity units compared to state-of-the-art LSTMs. The source code is publicly available [1].

1 Introduction

Modeling the sequential context in language is the key to success in many NLP tasks. Recurrent neural networks (RNNs) (Mikolov et al., 2010) memorize the sequential context in carefully designed cells. The sequential nature of these models, however, makes computation expensive (Merity et al., 2017; Yang et al., 2017), and therefore it is difficult to scale to large corpora.

The Transformer architecture (Vaswani et al., 2017) replaces RNN cells with self-attention and point-wise fully connected layers, which are highly parallelizable and thus cheaper to compute. Together with positional encoding, Transformers are able to capture long-range dependencies with vague relative token positions. This results in a coarse-grained sequence representation at sentence level. Recent works such as GPT (or GPT-2) (Radford et al., 2018, 2019) and BERT (Devlin et al., 2018) show that the representations learned on large-scale language modeling datasets are effective for fine-tuning both sentence-level tasks, such as GLUE benchmark (Wang et al., 2018), and token-level tasks that do not rely on word order dependency in the context, such as question answering and NER.

Despite the fact that both GPT and BERT use language models for pre-training, neither of them achieves state-of-the-art performance in language modeling. Language model aims to predict the next word given the previous context, where fine-grained order information of words in context is required. Neither self-attention nor positional encoding in the existing Transformer architecture is effective in modeling such information.

A second challenge (and opportunity) arises from the fact that we may often have access to models pre-trained on related, albeit not identical tasks. For instance, neither GPT or BERT is tuned for WikiText and neither of them aims to minimize perplexity directly. In fact, the architectures may not even be useful directly: BERT provides estimates of $p(w_i|\text{context})$ rather than $p(w_i|\text{history})$. This shows that there is a need for us to design algorithms which systematically explore the space of networks that can be derived (and adapted) from such tasks. This generalizes the problem of making use of pre-trained word embeddings for related tasks, only that in our case we do not have vectors but rather entire networks to deal with.

Lastly, the problem of architecture search per-se has received great interests. However, the size of

[1] https://github.com/cgraywang/gluon-nlp-1/tree/lmtransformer/scripts/language_model

the datasets where training a single model for GPT or BERT can cost in excess of $10,000, makes it prohibitively expensive to perform a fully-fledged model exploration with full retraining. Instead, we propose to use architecture search in a much more restricted (and economical) manner to investigate refining a trained architecture. This is much cheaper. Our pragmatic approach leads to improvements on the state-of-the-art in language modeling. Our contributions are as follows:

1. We propose a Transformer architecture for language model. It works by adding LSTM layers after all Transformer blocks (a result of the search algorithm). This captures fine-grained word-level sequential context.
2. We describe an effective search procedure, Coordinate Architecture Search (CAS). This algorithm randomly generates variants of the Transformer architecture, based on the current best found architecture. Due to its greedy nature, CAS is simpler and faster than previous architecture search algorithms (Zoph and Le, 2016; Pham et al., 2018; Liu et al., 2018).
3. We show how this can be used to incorporate substantial prior knowledge in the form of GPT or BERT. Using this information via brute force architecture search would be prohibitively expensive.

Contributions 2 and 3 are general and apply to many cases beyond NLP. Contribution 1 is arguably more language specific. We evaluate CAS on three popular language model datasets: PTB, WikiText-2 and WikiText-103. The BERT-based CAS achieves in average 12.0 perplexity gains compared to the state-of-the-art LSTM-based language model AWD-LSTM-MoS (Yang et al., 2017).

2 Transformers for Language Models

Our Transformer architectures are based on GPT and BERT. We will reuse the pre-trained weights in GPT and BERT to fine-tune the language model task. During fine-tuning, we modify and retrain the weights and network used by GPT and BERT to adapt to language model task.

2.1 GPT and BERT

GPT (Radford et al., 2018) uses a variant of the Transformer architecture (Vaswani et al., 2017). That is, it employs a multi-layer Transformer decoder based language model. The original paper provides a pre-trained architecture with 12-layer Transformer decoder-only blocks. Each block has hidden size 768 and 12 self-attention heads. The weights are trained on BooksCorpus. This allows it to generate $p(w_i|\text{history})$, one word at a time.

BERT is a multi-layer bidirectional Transformer encoder (Devlin et al., 2018). The original paper provides two BERT structures: BERT-Base, consists of 12-layer bidirectional Transformer encoder block with hidden size 768 and 12 self-attention heads; BERT-Large includes 24-layer bidirectional Transformer encoder blocks with hidden size 1024 and 16 self-attention heads. The weights are trained on BooksCorpus and the English Wikipedia. Unless stated otherwise, we mean BERT Base when mentioning BERT.

Relation between GPT and BERT. Both models use virtually the same architecture. In fact, GPT and BERT-Base even use the same number of layers and dimensions. The only difference is that BERT is bidirectional since it tries to fill in individual words given their context, whereas GPT uses masked self-attention heads.

2.2 Adapting GPT and BERT for Sub-word Language Model

GPT needs little modification, unless we want to explore different architectures. After all, it is already trained as a language model. At a minimum, during fine-tuning we add a linear layer with hidden size equal to the vocabulary size. These weights are tuned and fed into the softmax to generate a probability distribution of the target word over the vocabulary. Masked self-attention ensures that only causal information flow can occur.

Recall the objective of BERT: masked language model and next sentence prediction. The masked language model uses bidirectional contextual information and randomly masks some tokens during training. Based on that it tries to infer the identity of the masked word. Unfortunately, estimating $p(w_i|w_1, \dots w_{i-1}, w_{i+1}, \dots w_n)$ is not conducive to building an effective text generator: We would need to design a Gibbs sampler to sample $w_i|w^{-i}$, i.e. w_i given its context w^{-i} iteratively and repeatedly for all i to use a variant of this aspect directly.

The next sentence prediction aims to capture the binarized relationship between two sentences. Again, this is not directly useful for LM. We thus remove the objective and replace it by a log-likelihood measure during fine-tuning. Similar to

GPT, we add an output linear layer and replace the self-attention heads with masked self-attention to prevent leftward information flow.

Note that GPT and BERT pre-trained weights are re-used in the language model fine-tuning process to save the costs of a full retraining. We are thus conducting the language model in the sub-word level since the sub-word tokenization is used in both GPT and BERT. More details will be described in Section 4.

2.3 Fine-tuning Transformer Weights

GPT and BERT tune the weights of their respective models for the tasks mentioned above. For instance, BERT doesn't use windowing by default. Hence it makes sense to adjust the weights when fine-tuning for language modeling. However, updating all weights could lead to overfitting since datasets such as WikiText or Penn Tree Bank are over an order of magnitude smaller than the data used to train GPT and BERT.

To address this dilemma we propose to update only a subset of layer weights during fine-tuning. Since both GPT and BERT have 12 Transformer blocks, each of which contains a self-attention and a point-wise fully connected layer, it is not straightforward to choose the subset of layers whose parameters should be fixed. Instead, we will automatically search the subset which is most effective for the language model task. The search algorithm will be discussed in Section 3.

2.4 Adding an LSTM

The positional encoding via a Fourier base in the Transformer only provides vague relative position information, forcing the layers to reinvent trigonometry at *each* layer for specific word access. This is problematic since LM requires strong word-level context information to predict the next word. RNNs explicitly model this sequential information. We therefore propose to add LSTM layers to the Transformer architecture.

In theory we could add LSTM layers anywhere, even interleaving them with Transformers. However, LSTMs add significant computational efficiency penalties, since they prevent parallel computation. Our reasoning is analogous to that guiding the design of the SRU (simple recurrent unit) (Lei et al., 2018). Hence we propose to add an LSTM layer either before all basic Transformer blocks or after these blocks. For the former, we add the LSTM layer immediately after the embedding layer and remove the positional and segment embedding, because we believe the LSTM layer is able to encode sufficient sequential information. For the latter, we insert the LSTM layer between the last Transformer block and the output linear layer. We determine the best location for the LSTM by automatic search.

3 Coordinate Architecture Search

Now that we have the basic components, let's review the network transformations and the associated search procedures to obtain a well-performing architecture.

3.1 Network Transformations

Transformations modify a network. A modification could be adding a new layer or fixing the parameters during fine-tuning. In Section 2, we proposed multiple transformations. Let us define them formally below with randomization and practical constraints.

AddLinear adds a linear output layer with hidden size equal to the vocabulary size. It then randomly initializes its parameters. If such a linear layer already exists, this step is skipped.

AddLSTM adds an LSTM layer if no such layer already exists. It attaches the LSTM either before or after all Transformer blocks. For the former we remove both positional embedding and segment embedding. If there exist fewer than 3 LSTM layers, we append another LSTM layer to the LSTM block. We randomly initialize parameters for the newly added layer.

FixSubset Given n Transformer blocks, pick $k \in [0, n]$ uniformly at random. Accordingly pick k blocks uniformly at random among the $\{1, \dots n\}$ layers and fix the parameters for each selected block during fine-tuning.

3.2 Sampling a Search Candidate

We need to generate architecture candidates during search. To illustrate that restricted search is competitive to a full-fledged brute force reinforcement learning (or genetic algorithms) search, we adopt an exceedingly simple procedure: uniform random sampling. At each time we sample transformations uniformly at random (as per Algorithm 1) from the set of modifications of a base architecture until termination, as indicated

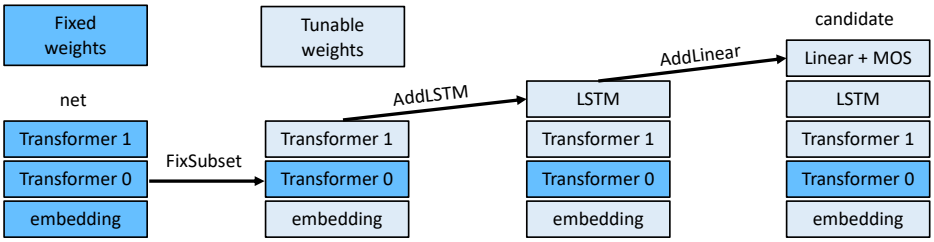

Figure 1: Search candidate sampling. *net* is the base architecture and *candidate* is returned in the next step. Transformers, Embeddings, LSTMs and Linear output transformations are as stated. Lightly shaded blocks are variable, dark blocks are fixed. See Algorithm 1 for details.

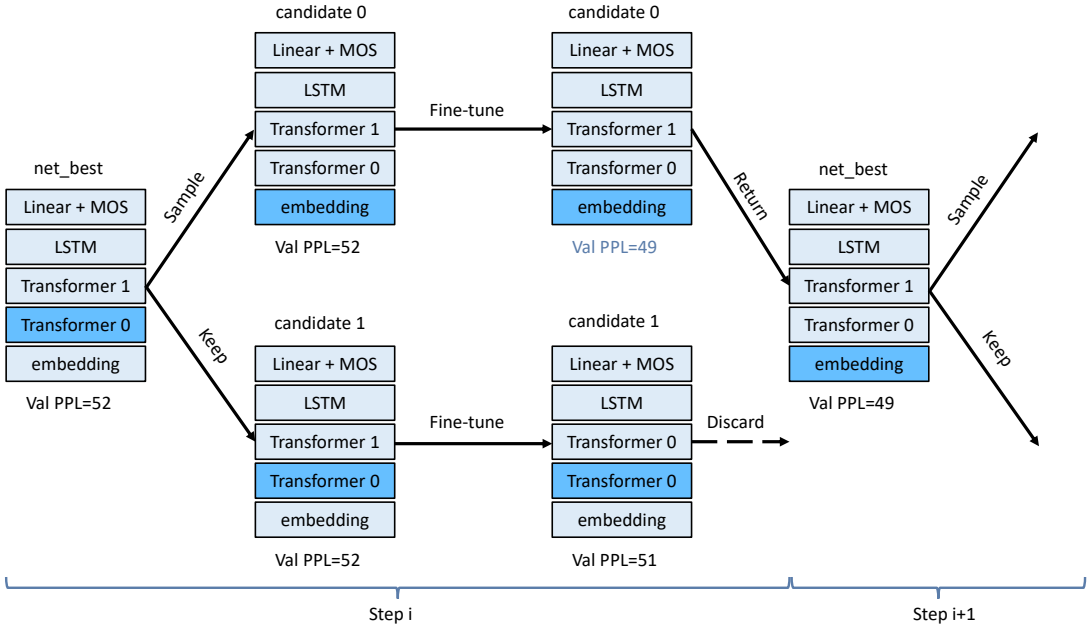

Figure 2: Coordinate architecture search. *net_best* is the best architecture at step i of the search. We sample search candidates and keep the one that performs best, as measured by perplexity (Val PPL) on the target dataset after fine-tuning. See details in Algorithm 2.

Algorithm 1 Search Candidate Sampling

Input: Base architecture *net*
Output: A new architecture *candidate*

1: *candidate* $\leftarrow$ *net*
2: **repeat**
3: Sample a tranformation $\mathcal{T}$ uniformly from $\{AddLinear, AddLSTM, FixSubset\}$.
4: Apply $\mathcal{T}$ to *candidate*
5: **until** $\mathcal{T} = AddLinear$
6: **return** *candidate*

by adding an emissions layer $AddLinear$. This means that we have a valid architecture. See Figure 1 for an example.

3.3 Coordinate Architecture Search

We use a simple greedy strategy for architecture search. Starting with either GPT or BERT as pre-trained model we repeat the search n times. Each time we sample a candidate, then fine-tune it and update the best candidate if necessary. See Algorithm 2 for a description and Figure 2 for an illustration. At each (successful) step we fine-tune the variable parameters of the architecture.

4 Experiments

To illustrate the effectiveness of the Transformer architectures found using coordinate search we present results on both WikiText and Penn TreeBank datasets. We also provide details about its speed relative to existing neural search strategies.

4.1 Datasets and Evaluation Metric

We evaluate the proposed methods on three widely-used language model benchmark datasets. **Penn TreeBank (PTB)**: we use the preprocessed

Algorithm 2 Coordinate Architecture Search

Input: Initial architecture *net*, search steps n, fine-tuning dataset
Output: Best architecture net_{best}

$net_{\text{best}} \leftarrow net$;
for $i = 1$ **to** n **do**
  Draw $candidate$ from net_{best} using Algorithm 1.
  Fine-tune $candidate$ on dataset
  if $\text{PPL}(candidate) < \text{PPL}(net_{\text{best}})$ **then**
    $net_{\text{best}} \leftarrow candidate$
  end if
end for
return net_{best}

version of (Mikolov et al., 2010), which contains 100M tokens. **WikiText-2 (WT-2)** is a small pre-processed version of Wikipedia, containing 200M tokens (Merity et al., 2016). **WikiText-103 (WT-103)** contains 1B tokens of the same origin as WT-2. We use the commonly adopted training, validation and test splits.

To illustrate the flexibility of our approach, we explore two pre-trained Transformers for sub-word level language models, i.e., BERT and GPT. For BERT related model architectures, we use WordPiece embedding (Wu et al., 2016) to tokenize the training/validation/test split of the PTB, WT-2 and WT-103 respectively. The resulting sub-word vocabulary size is 30k, denoted as *BERTVocab*. The split word pieces are denoted with ## following (Devlin et al., 2018). For the model architectures based on GPT, the three datasets are tokenized based on bytepair encoding (BPE) (Sennrich et al., 2016), where the sub-word vocabulary size is 40k based on (Radford et al., 2018), denoted as *GPTVocab*. Note that BERT and the WordPiece embedding in BERT are trained on BooksCorpus and Wikipedia, whereas GPT and its BPE are trained only on BooksCorpus. Note the sub-word level vocabulary size is different from the word-level vocabulary size obtained on the training splits of the datasets. We use perplexity (PPL) to evaluate the sub-word language model results.

4.2 Training Details

We evaluate CAS (Algorithm 2) with both BERT and GPT pre-trained as the initial architecture, and trained on all three datasets. The same training configuration is used across all datasets. We pick $n = 10$ search steps. In a fine-tuning task, the number of epochs is 50, the gradients are computed using truncated back-propagation through time, and ADAM (Kingma and Ba, 2014) is used to update parameters. The perplexity on the validation dataset is used to choose architectures. We report results on the respective test datasets.

For GPT based architectures the hyperparameters of the Transformer decoder and embedding blocks are the same as in (Radford et al., 2018). If LSTM layers are added, we set the dropouts of the LSTM layers to 0.1. DropConnect is not applied. All other LSTM hyperparameters follow (Merity et al., 2017). The final linear layer is with dropout rate 0.1. Following (Yang et al., 2017), we use a mixture of softmax (MoS) to replace the standard softmax with 15 components. We set 64 as sequence length and 16 as minibatch size. ADAM with learning rate $6.25 \cdot 10^{-5}$ and L2 weight decay of 0.01 are used.

For BERT based architectures the hyperparameters of the Transformer encoder blocks and the embedding blocks are set the same as the original implementation (Devlin et al., 2018). The hyperparameters of the LSTM layers and linear layer are the same with GPT configuration. As with GPT we use MoS with 15 components. We pick 128 as sequence length and 16 as minibatch size. ADAM with learning rate 10^{-4}, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and L2 weight decay of 0.01 are used.

Lastly, for AWD-LSTM-MoS with BERT or GPT sub-word setting, we largely follow the parameter settings in the original implementation (Yang et al., 2017). We use NT-ASGD (Merity et al., 2017) to train 50 epochs on training datasets.

Since the goal of this work is to discover best-performing language model from the architecture perspective, we do not employ post-training methods such as neural cache model (Grave et al., 2016) or dynamic evaluation (Krause et al., 2018). We expect that such methods would potentially improve the perplexity of all models.

4.3 Comparing CAS to Other Methods

We compare CAS, denoted by **BERT-CAS** and **GPT-CAS** respectively to three other models.

BERT and GPT. This is straightforward. The only change needed is that we update the last output layer during fine-tuning.

AWD-LSTM-MoS-{BERT, GPT}Vocab. This is a state-of-the-art language model, based on

Model	Datasets					
	PTB		WT-2		WT-103	
	Val	Test	Val	Test	Val	Test
AWD-LSTM-MoS-BERTVocab	43.47	38.04	48.48	42.25	54.94	52.91
BERT	72.99	62.40	79.76	69.32	109.54	107.30
BERT-CAS (Our)	39.97	34.47	38.43	34.64	40.70	39.85
BERT-Large-CAS (Our)	**36.14**	**31.34**	**37.79**	**34.11**	**19.67**	**20.42**
AWD-LSTM-MoS-GPTVocab	50.20	44.92	55.03	49.77	52.90	51.88
GPT	79.44	68.79	89.96	80.60	63.07	63.47
GPT-CAS (Our)	**46.24**	**40.87**	**50.41**	**46.62**	**35.75**	**34.24**

Table 1: Performance of Coordinate Architecture Search (CAS). 'Val' and 'Test' denote validation and test perplexity respectively.

LSTMs, improving on (Yang et al., 2017) due to a more careful handling of tokens. For a fair comparison, instead of using word level vocabulary in the original implementation of AWD-LSTM-MoS (Yang et al., 2017), we use the sub-word vocabularies of BERT and GPT separately. Our implementation uses BERTVocab or GPTVocab to replace the word based vocabulary used in the original implementation of AWD-LSTM-MoS (Yang et al., 2017). Note that on PTB and WT-2, both AWD-LSTM-MoS-BERTVocab and AWD-LSTM-MoS-GPTVocab outperform the original AWS-LSTM-MoS models by 17.8 and 10.6 perplexity points respectively. This is likely due to the change in word vocabulary to a sub-word vocabulary.

The results are shown in Table 1 and illustrated in Figure 3. First note that GPT and BERT are significantly worse than AWD-LSTM-MoS. It confirms our hypothesis that neither BERT nor GPT are effective tools for language modeling. Applying them naively leads to significantly worse results compared to AWS-LSTM-MoS on three datasets. It demonstrates that language modeling requires strong capabilities in modeling the word order dependency within sentences. However, due to the combination of self-attention and positional encoding, both GPT and BERT primarily capture coarse-grained language representation but with limited word-level context.

On the other hand, the Transformer architectures picked by CAS (BERT-CAS) outperform AWS-LSTM-MoS on all datasets. The average test perplexity improvement with BERT pre-trained models is 8.09 and 8.28 with GPT pre-trained models. The results demonstrate that 1) CAS is able to locate an effective Transformer architecture for language model; and 2) that the combination of fixing a subset weights and adding LSTM layers is capable of capturing the word-level context.

Furthermore, we apply CAS to BERT-Large (i.e., BERT-Large-CAS). Compare to BERT-CAS, the architectures generated achieve on average 7.70 perplexity gains, which are competitive results with recent approaches such as Transformer-XL (Dai et al., 2019) and GPT-2 (Radford et al., 2019). This shows that robustness of the CAS method, which indicates that a stronger pre-trained model would potentially produce a better language model.

In addition, BERT-CAS outperforms GPT-CAS on datasets PTB and WT-2, but is worse on WT-103. The reason is twofold. First, the GPT's BPE vocabulary is 10k larger than BERT's Word-Piece vocabulary, since the original word vocabulary size of WT-103 is around 10 times larger compared to PTB and WT-2, thus we infer that BPE vocabulary has stronger ability to represent large vocabulary. Second, unlike GPT, the pre-trained BERT weights are not based on a language modeling objective. Thus BERT based architectures may need more epochs to converge on large corpora. This is likely due to the fact that masking is not a part of BERT training. Its introduction amounts to a more significant change in the covariates, thus requires more adaptation.

4.4 Ablation Study

To elucidate the effects of different model improvements we compare CAS to the following three variants:

{BERT, GPT}-CAS-Subset applies Algorithm 2 without adding LSTM layers.

Model	Datasets					
	PTB		WT-2		WT-103	
	Val	Test	Val	Test	Val	Test
BERT-CAS-Subset	42.53	36.57	**51.15**	**44.96**	**44.34**	**43.33**
BERT-CAS-LSTM	**40.22**	**35.32**	53.82	47.00	53.66	51.60
GPT-CAS-Subset	47.58	41.85	54.58	50.08	**35.49**	**35.48**
GPT-CAS-LSTM	**47.24**	**41.61**	**50.55**	**46.62**	36.68	36.61

Table 2: Ablation study. Compare CAS with not adding LSTM layers (CAS-Subset) and not updating Transformer block parameters (CAS-LSTM).

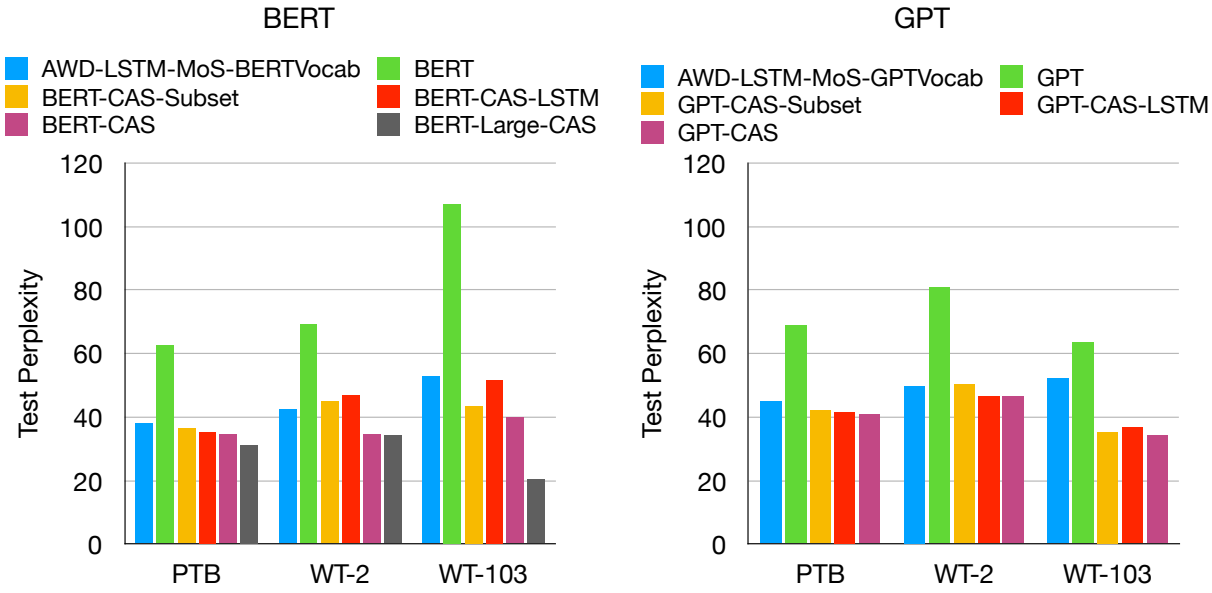

Figure 3: Comparison of test perplexities between CAS and other models (left: using BERT pre-trained models; right: using GPT pre-trained models). In particular, 'Subset' indicates variants without LSTMs and 'LSTM' corresponds to models without updating the transformer blocks.

{BERT, GPT}-CAS-LSTM applies Algorithm 2 but it fixes all Transformer blocks during fine-tuning.

See Table 2 and Figure 3 for details of the results. As can be seen, both CAS-Subset and CAS-LSTM improve significantly upon a naive use of BERT and GPT. This is to be expected since fine-tuning improves performance. On the smaller dataset, i.e. PTB, adding LSTMs is more effective. This might be due to the overfitting incurred in updating Transformers. On the other hand, on the larger datasets, i.e. WT-103, adding an LSTM is less effective, which means that adapting the Transformer parameters for better sentence-level representation is more important. Combing both together leads to further improvement. CAS outperforms AWD-LSTM-MoS on all three datasets.

Next, we unfreeze the pre-trained weights of BERT to allow fully fine-tuning including the last linear output layer (BERT-All) on PTB data as an example, to illustrate the over-fitting issue. From the results in Table 3, we can see that, by leveraging CAS, we marginally relieve the over-fitting issue by fixing a subset of weights of the full Transformer architecture.

Model	Validation	Test
BERT-All	79.14	67.43
BERT-CAS	**39.97**	**34.47**

Table 3: Over-fitting example on PTB data. BERT-All: BERT with fully fine-tuning including the last layer. BERT-CAS: BERT with coordinate architecture search.

Let's look into the details of adding LSTMs. There are 4 cases:

Only-LSTM implements a model consisting only

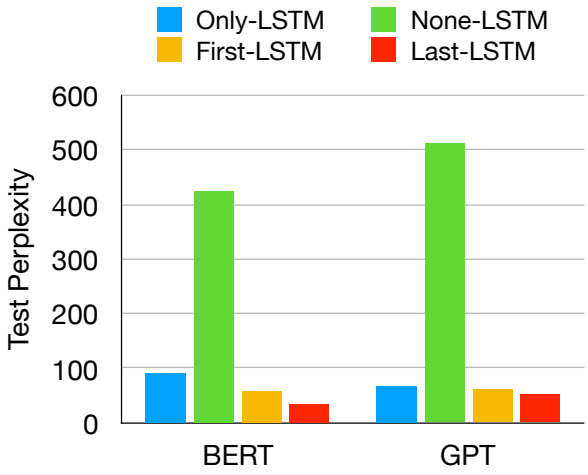

Figure 4: LSTM variants for Penn TreeBank. We study whether to add LSTMs before or after the transformer layers (or none at all).

of LSTM layers. To make up for the loss of expressiveness due to removing all Transformer blocks we add a total of 6 LSTM layers.

None-LSTM adds no LSTM layer at all. Instead, we add another stack of Transformer blocks. This effectively doubles the number of blocks to 24.

First-LSTM adds LSTM layers only before all Transformer blocks.

Last-LSTM adds LSTM layers only after all Transformer blocks.

The results are shown in Table 4 and in Figure 4. As can be seen, neither purely transformer blocks nor purely LSTM layers are effective for language modeling. The former is likely unsuitable due to the comparatively large number of parameters relative to the tuning set. Adding LSTM layers properly into Transformer architecture significantly improves the perplexity. In addition, adding LSTMs before the output linear layer outperforms replacing positional and segment embeddings with LSTM layers. These results confirm our intuition and indicate that we need to first preserve the coarse-grained representation using fixed subset weights; subsequently LSTMs can be used to model the word order dependency.

4.5 Efficiency Analysis

Lastly, we compare CAS with other existing neural network search methods in terms of search cost. The main distinction being that we significantly constrain the architectures to be investigated. This allows us to obtain significant computational savings.

Constraints			Validation	Test
BERT	Only	LSTM	107.47	89.82
	None		491.75	425.32
	First		67.85	57.71
	Last		**39.97**	**34.47**
GPT	Only	LSTM	75.76	66.56
	None		579.77	510.49
	First		70.05	60.82
	Last		**46.24**	**40.87**

Table 4: Effects of different search constraints for placing the LSTM on perplexity on the PTB data.

NAS by (Zoph and Le, 2016) is a reinforcement learning based search method, which uses a recurrent network to generate the model descriptions of neural networks and minimizes the expected perplexity of the generated architectures on the PTB validation set.

ENAS by (Pham et al., 2018) also leverages a reinforcement learning search method. ENAS's search space is the superposition of all possible child models in the NAS search space, which allows parameters to be shared among all child models.

DARTS by (Liu et al., 2018) is a recently proposed neural architecture search algorithm based on gradient descent.

We evaluate the efficiency of the methods using GPU days. The search costs of NAS, ENAS and DARTS are obtained from (Liu et al., 2018).

The reported search costs of the above methods compared to CAS are shown in Table 5. As can be seen, BERT-CAS is cheaper than all others. The results indicate that by leveraging the prior knowledge of the design of the neural networks for specific tasks, we could only optimize the architectures in a small confined sub-space, that leads to speed up the search process. For example, BERT-CAS is directly based on BERT, applying search upon such effective neural networks could facilitate the adaptation to similar tasks.

The reason of the search cost of GPT-CAS on WT-2 is higher than ENAS is three-fold:

1. ENAS is directly transferring an architecture searched based on PTB to WT-2. Instead we apply coordinate search to find one from scratch;

Search Method	Search Cost (GPU days)		Method Class
	PTB	WT-2	
NAS (Zoph and Le, 2016)	1,000 CPU days	n.a.	reinforcement
ENAS (Pham et al., 2018)	0.5	0.5	reinforcement
DARTS (first order) (Liu et al., 2018)	0.5	1	gradient descent
DARTS (second order) (Liu et al., 2018)	1		gradient descent
BERT-CAS (Our)	**0.15**	**0.38**	greedy search
GPT-CAS (Our)	0.23	0.53	greedy search

Table 5: Efficiency of different search methods on PTB and WT-2.

Model	Parameters	Datasets		
		PTB	WT-2	WT-103
GPT-2	345M	47.33	22.76	26.37
	762M	40.31	19.93	22.05
	1542M	35.76	**18.34**	**17.48**
BERT-Large-CAS	395M	**31.34**	34.11	20.42

Table 6: Compare model parameter size and results with GPT-2. The GPT-2 model size and results are from (Radford et al., 2019).

Model	Training Data	Tokens
GPT-2	WebText	14.0B
BERT-Large-CAS	PTB	**0.1B**
	WT-2	**0.2B**
	WT-103	**1.0B**

Table 7: Compare training data size with GPT-2.

2. The model size of GPT-CAS is 149M, which is much larger compared to the size 37M from ENAS;
3. The GPT vocabulary size is 10k larger compared to the ENAS's vocabulary.

We note that the difference in vocabularies might affect the results, since the results of NAS, ENAS and DARTS are from the original implementations. The original implementations are based on basic word tokenization (such as space splitter) of the PTB and WT-2. Instead, we are using the sub-word tokenization (WordPiece and BPE respectively) for BERT and GPT architecture exploration. However, the vocabulary size after basic tokenization processing is similar to the results after the sub-word tokenization, which are all around 30k-40k. Given that, we consider the performance comparison as fair [2].

4.6 Comparison with GPT-2

We specifically compare the proposed model with the recent state-of-the-art language model GPT-2 (Radford et al., 2019) on three dimensions: results[3], parameter size, and scale of the training data. From the results shown in Table 6, we conclude that with comparable size of the models' parameters, BERT-Large-CAS outperforms GPT-2 (345M) by on average 10.97 PPL on PTB and WT-103. More surprisingly, the proposed method performs better than GPT-2 (1542M) which has around 4 times more parameters. On WT-103, BERT-Large-CAS is better than GPT-2 (762M) which has around 2 times more parameters. Note that on WT-2, our method performs worse than GPT-2, we suspect the reason is that the WebText still contains the texts that are similar to the Wikipedia. WT-2 is quite small in terms of scale. In contrast, we regard the results on WT-103 (50 times larger than WT-2) as a more reasonable comparison with GPT-2.

The training data described in Table 7 suggests

[2]The PPL results are not comparable since the vocabularies are different (i.e., sub-word versus word level), we omit the comparison here.

[3]The results comparison is fair since GPT-2's vocabulary is also based on sub-word tokenization.

that, with significantly smaller training datasets, the proposed method generates competitive results. Once GPT-2 models are released, we expect CAS could generalize to the GPT-2 models to obtain better results for language model task.

5 Related Work

Architecture search has shown promising results in tasks such as image classification (Zoph and Le, 2016; Liu et al., 2017a,b; Real et al., 2018; Zoph et al., 2018; Liu et al., 2018), object detection (Zoph et al., 2018) as well as language modeling (Zoph and Le, 2016; Pham et al., 2018; Liu et al., 2018) in NLP. Existing neural architecture search studies focus on leveraging different methods to build the neural network from scratch. For example, NAS (Zoph and Le, 2016) uses reinforcement learning to obtain an architecture for CIFAR-10 and ImageNet. Designing the architecture from scratch using reinforcement learning is very costly. Many follow-up studies focus on speeding up the search process by weight-sharing across child models (Pham et al., 2018; Cai et al., 2018), by incorporating a particular structure into the search space (Liu et al., 2017a,b), or by enabling weights prediction for each architecture (Brock et al., 2017; Baker et al., 2017). Different from the above methods, the proposed coordinate search does not involve any controllers.

Recent studies start to explore using the idea of network transformation within reinforcement learning (Cai et al., 2018) or via Bayesian optimization (Jin et al., 2018) or simple greedy search (Elsken et al., 2017). DARTS (Liu et al., 2018) enables gradient descent to optimize the architecture. Compared to these methods, the coordinate search is more straightforward and more efficient due to the direct incorporation of the pre-defined Transformer architecture. Notably, the major difference of the proposed search algorithm compared to the existing methods is that we focus on *adapting* an existing well-trained Transformer architecture with minimum changes in the task of language model, whereas a majority of the existing work focus on generating variants of RNN cells *from scratch* for better results.

Language models have been studied extensively in NLP. Neural language models have supplanted traditional n-gram models in recent years (Bengio et al., 2003; Mnih and Hinton, 2007; Mikolov et al., 2010). Particularly, recurrent neural networks (Inan et al., 2016; Merity et al., 2017; Melis et al., 2017; Krause et al., 2018), such as LSTMs have achieved state-of-the-art results on various benchmark datasets with different regularization techniques and post-training methods (Grave et al., 2016; Krause et al., 2018). The mixture of softmax (Yang et al., 2017) has helped address the low-rank embedding problem for word prediction. We used this in our model, too. It provides some improvement over a more conventional model.

The recently proposed GPT-2 (Radford et al., 2019) is a deeper Transformer decoder based language model trained on a 40GB dataset. In contrast, the proposed model generates competitive results but with significantly less training cost and smaller model size. Transformer-XL (Dai et al., 2019) is a word level language model that also delivers good results by incorporating longer context. The proposed method is a sub-word level language model thus the results are not comparable. We expect to generalize CAS to pre-trained Transformer-XL models as well to achieve better results. The adaptive input representations idea proposed in (Baevski and Auli, 2018) could be combined with the proposed method to further speed up.

Network transformations were introduced in the context of the transfer learning (Chen et al., 2015). The main purpose of the transformations is to make networks deeper and wider. Often stagewise training accelerates training and architecture search. Recent studies (Wei et al., 2016; Cai et al., 2018; Elsken et al., 2017) focus on extending the set of the network transformations to handle additional operations such as non-linear activation functions and skip connections. We instead introduce simple network modifications to perform modest modifications of an existing network. They allow us to treat a pre-trained Transformer block in a manner similar to that of a large pre-trained embedding vector.

6 Conclusion

We study the problem of finding an effective Transformer architecture for language model. We identify the issues of existing Transformer architectures, such as BERT and GPT, that are not able to capture the strong word-level context required in language model. We proposed two approaches to address this issue: we fine-tune a subset of pa-

rameters to improve the coarse-grain representations obtained from the pre-trained Transformer models. Secondly, we add LSTM layers to capture the fine-grained sequence. We then propose a coordinate architecture search (CAS) algorithm to select an effective architecture based on fine-tuning results. It uses a greedy search strategy to accelerate architecture search. We experimentally show that CAS outperforms the state-of-the-art language models on three language model benchmark datasets.

Although we only show the effectiveness of CAS when applying Transformer architectures to the language model task, we feel it is possible to apply CAS to both other neural network architectures and fine-tuning other NLP tasks that require strong word-level context as well.

References

Alexei Baevski and Michael Auli. 2018. Adaptive input representations for neural language modeling. *arXiv preprint arXiv:1809.10853*.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2017. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *JMLR*, 3(Feb):1137–1155.

Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. 2017. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*.

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. 2018. Efficient architecture search by network transformation. AAAI.

Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. 2015. Net2net: Accelerating learning via knowledge transfer. *CoRR*.

Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2017. Simple and efficient architecture search for convolutional neural networks. *CoRR*.

Edouard Grave, Armand Joulin, and Nicolas Usunier. 2016. Improving neural language models with a continuous cache. *CoRR*.

Hakan Inan, Khashayar Khosravi, and Richard Socher. 2016. Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462*.

Haifeng Jin, Qingquan Song, and Xia Hu. 2018. Efficient neural architecture search with network morphism. *arXiv preprint arXiv:1806.10282*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. 2018. Dynamic evaluation of neural sequence models. In *ICML*, pages 2771–2780.

Tao Lei, Yu Zhang, Sida I Wang, Hui Dai, and Yoav Artzi. 2018. Simple recurrent units for highly parallelizable recurrence. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4470–4481.

Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2017a. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2017b. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: differentiable architecture search. *CoRR*.

Gábor Melis, Chris Dyer, and Phil Blunsom. 2017. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*.

Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *CoRR*.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *CoRR*.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*.

Andriy Mnih and Geoffrey Hinton. 2007. Three new graphical models for statistical language modelling. In *ICML*, pages 641–648.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient neural architecture search via parameter sharing. In *ICML*, pages 4092–4101.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/research-covers/language-unsupervised/language_ understanding_paper. pdf*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2018. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *ACL*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*, pages 5998–6008.

Alex Wang, Amapreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. 2016. Network morphism. In *ICML*, pages 564–572.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W. Cohen. 2017. Breaking the softmax bottleneck: A high-rank RNN language model. *CoRR*.

Barret Zoph and Quoc V. Le. 2016. Neural architecture search with reinforcement learning. *CoRR*.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning transferable architectures for scalable image recognition. In *CVPR*, pages 8697–8710.