AutoFedNLP: An efficient FedNLP framework

Dongqi Cai[1], Yaozong Wu[1], Shangguang Wang[1], Felix Xiaozhu Lin[2], Mengwei Xu[1]
Beiyou Shenzhen Institute[1]
University of Virginia[2]

ABSTRACT

Transformer-based pre-trained models have revolutionized NLP for superior performance and generality. Fine-tuning pre-trained models for downstream tasks often requires private data, for which federated learning is the de-facto approach (i.e., FedNLP). However, our measurements show that FedNLP is prohibitively slow due to the large model sizes and the resultant high network/computation cost. Towards practical FedNLP, we identify as the key building blocks *adapters*, small bottleneck modules inserted at a variety of model layers. A key challenge is to properly configure the depth and width of adapters, to which the training speed and efficiency is highly sensitive. No silver-bullet configuration exists: the optimal choice varies across downstream NLP tasks, desired model accuracy, and client resources. To automate adapter configuration, we propose `AutoFedNLP`, a framework that enhances the existing FedNLP with two novel designs. First, `AutoFedNLP` progressively upgrades the adapter configuration throughout a training session; the principle is to quickly learn shallow knowledge by only training fewer and smaller adapters at the model's top layers, and incrementally learn deep knowledge by incorporating deeper and larger adapters. Second, `AutoFedNLP` continuously profiles future adapter configurations by allocating participant devices to trial groups. To minimize client-side computations, `AutoFedNLP` exploits the fact that a FedNLP client trains on the same samples repeatedly between consecutive changes of adapter configurations, and caches computed activations on clients. Extensive experiments show that `AutoFedNLP` can reduce FedNLP's model convergence delay to no more than several hours, which is up to 155.5× faster compared to vanilla FedNLP and 48× faster compared to strong baselines.

1 INTRODUCTION

With the recent rise of transformers and its variants [9, 20, 31, 48, 64, 69, 73, 85], modern NLP models show compelling use cases on client devices. Examples include sentiment analysis, QA, and auto completion [38, 65, 71, 72, 86]. Through careful engineering [9, 31, 48, 64, 69, 85], inference with the NLP models is demonstrated to be affordable on client devices.

Much success of modern NLP comes from its training workflow as illustrated in Figure 1. (1) The pre-training phase initializes a model on large text corpora. The training is self-supervised and time-consuming, often taking hundreds if not thousands of GPU days [14, 20]. Pre-training teaches the model a language's inherent structure, e.g. word distribution. (2) The fine-tuning phase further adapts a pre-trained model for a specific NLP task targeting a specific domain, e.g. to classify sentiments (a task) of user emails (a domain) [20]. Fine-tuning is indispensable to modern NLP training; only through it, the model maps the generic language understanding to the outputs for rich NLP tasks.

FedNLP The two NLP training phases require data of disparate natures. While pre-training is typically done on public text corpora

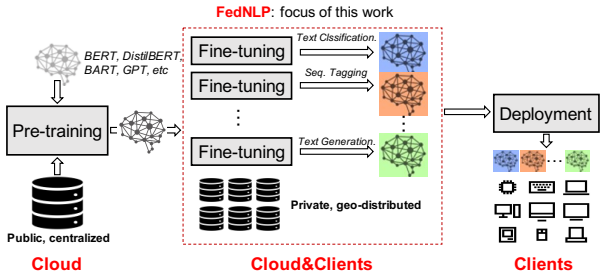


Figure 1: FedNLP and its role in modern NLP

such as Wikipedia articles, fine-tuning requires domain-specific samples such as user reviews, messages, or emails. In personal computing, these samples are generated by end users continually, distributed over client devices, and in many cases considered privacy sensitive.

To fine-tune models on private, distributed data, federated learning is the de-facto approach [13, 47]. In a training session targeting a specific NLP task and domain, a cloud service selects multiple clients to participate in training. A client trains a local copy of the model with its private data and sends the model updates to the cloud. Having aggregated model updates from multiple clients, the cloud sends an updated model to the clients. The training procedure repeats many rounds (typically hundreds or thousands [51, 67, 81]) until the model accuracy reaches a desired level, i.e., convergence.

As such, this paper focuses on NLP model fine-tuning in a federated setting, a core NLP process in personal computing. Such a process is often referred to as FedNLP [47], for which §2.2 will present a detailed system model.

FedNLP overhead Despite the established FedNLP algorithm [47], it was unclear if FedNLP is practical on today's mobile/embedded platforms. This paper's first contribution is a thorough characterization of FedNLP on a suite of benchmarks. Our results in §2.3 show prohibitive overheads in twofold: (1) Communication. In each round, participating devices upload local gradients and then download the updated model, each transferring hundreds of MBs of data. (2) Client computation. Even on an embedded device with GPU, its local computation takes up to several hundred seconds per round. As a result, a fine-tuning session can take as long as a few days. While federated learning has been known for high overhead in general, FedNLP is particularly expensive, primarily because of the large sizes of transformer-based models and NLP task complexity. As a comparison, FedNLP's delay is at least one order of magnitude higher than typical federated training delays reported in literature.

Adapters and their configuration Our primary goal is to reduce FedNLP's training delay to reach a target accuracy, i.e. time to accuracy. We first identify *adapters* [32] as key building blocks for NLP models. As small modules injected between adjacent transformer layers, adapters become the only tunable modules in a pre-trained model, freezing the remaining model parameters (often

>99%) which therefore incur no communication or compute overhead. While adapters have been proposed for parameter-efficient learning in general [32], we are the first to identify their significance for FedNLP and investigate the system implications.

Although adapters significantly reduce tunable parameters and hence the overhead, they do not automatically result in optimal training delays. The challenge is a large configuration space of adapters: to which layer the adapters are injected (depth) and the capacities of individual adapters (width). The adapter configuration has a strong impact on training overhead as well as the model convergence delay. For instance, adding fixed-size adapters to all layers could see up to 2.29× longer training delay as compared to adding the same adapters to fewer hand-picked layers (§3.2). There is no silver-bullet configuration; rather, the optimal configuration depends on many factors: the specific NLP tasks, the target accuracy, and client resources such as network bandwidth and local execution speed. The choice is also dynamic: even within the same training session, the favorable configuration drifts over time, depending on the model's learning progress. Picking a non-optimal configuration could slow down model convergence by up to 4.7x and even underperform training the whole model with no adapters at all.

Our system: `AutoFedNLP` We therefore present a system called `AutoFedNLP`, which speeds up FedNLP with two key designs.

First, `AutoFedNLP` augments the cloud controller with dynamic adapter configuration. The key ideas are twofold. (1) *Progressive training*. `AutoFedNLP` launches a training session with only small adapters inserted at the model's top layers (i.e. close to the model output), which essentially learns shallow knowledge at a low training cost. Only as the model accuracy starts to plateau, `AutoFedNLP` adds bottom-layer adapters to training and increases their widths, which learns deep knowledge at increasingly higher training costs. This resonates with how humans learn knowledge in an incremental fashion and modern learning theories [11]. (2) *Sideline trials*. In addition to training the model with a current configuration, `AutoFedNLP` probes *what* the next configuration will be and *when* to switch to it, for which `AutoFedNLP` continuously profiles multiple candidate configurations. To do so, from all the participating clients in each round, `AutoFedNLP` allocates multiple trial groups, requests them to train the model with different adapter configurations, and compares their learning progresses. If a trial group shows a much higher convergence rate than others as well as the current configuration under training, `AutoFedNLP` will commit to the configuration of this group. While configuration trial distracts some devices from training with the current configuration, it actually speeds up model convergence. This is because the model convergence rate often sees diminishing returns as the population of participant clients grows [51], and our insight is that the surplus clients can better benefit model convergence by profiling next configurations.

Second, `AutoFedNLP` enhances clients with cross-round activation cache. We exploit an observation: by design, a device trains the same set of adapters in repeated runs until the configuration switches; this set of adapters always spans a continuous range of top layers while the bottom layers remain frozen. Exploiting such an opportunity, for a given configuration a client only executes a forward pass *once* through the bottom layers, caches the output activations, and reuses the cached output as the input to the top layers which will undergo forward and backward passes. Caching thus eschews training for the bottom layers, reducing the total training cost by up to one order of magnitude.

Results We implement `AutoFedNLP` atop FedNLP [47], a popular FL framework. We test the resultant implementation on NVIDIA TX2 [1]/Nano [2] and RaspberryPi 4B [3], three development boards with resources similar to mainstream mobile devices. On a diverse set of 4 NLP datasets, `AutoFedNLP` reduces the training (model convergence) delay from 31.1–124.3 hours to 0.2–4.5 hours (up to 155.5× reduction) as against a vanilla fine-tuning approach. During a training session, `AutoFedNLP` reduces the network traffic by 126.7× and per-device energy consumption by 18.4× on average. Compared to more advanced methods such as model quantization [80], layer freezing [23, 47], and their combination, `AutoFedNLP` still brings 4×–48× speedup. Our key designs contribute to the results significantly: compared to a hand-picked configuration which requires exhaustive offline search, `AutoFedNLP`'s online configurator reduces the model convergence delay by 4.6×; `AutoFedNLP`'s caching reduces the delay by 3.3×. `AutoFedNLP` is also resource efficient: it reduces the network traffic by 126.7× and per-device energy consumption by 18.4× on average.

Contributions We have made the following contributions.

- We carry out the first FedNLP measurement on actual embedded hardware and demonstrate its slow convergence.
- We identify adapters as a building block for FedNLP and the major challenge of adapter configuration.
- We design an FL framework `AutoFedNLP` that automatically configures adapters on the fly for fast training and optimizes for client resource usage.
- We demonstrate `AutoFedNLP`'s effectiveness through extensive experiments. For the first time, `AutoFedNLP` makes FedNLP practical for commodity mobile/embedded devices.

2 BACKGROUND AND MOTIVATIONS

2.1 NLP training workflow

The modern NLP training typically consists of two stages: pre-training and fine-tuning. During pre-training, a model is trained on large text datasets, e.g., OSCAR corpora [6] with more than 370 billion words. Those datasets are obtained from public domains, e.g., Wikipedia, Twitter, etc. A pre-trained language model captures the linguistic structure that is ubiquitous and independent of downstream tasks. The pre-training is usually performed in a self-supervised manner and therefore requires no data labels [22]. It needs huge compute resources (a mid/large GPU cluster) [14, 20], typically done by big companies such as Google.

The fine-tuning is to adapt the pre-trained model to various, concrete "downstream" language tasks such as text classification, sequence tagging, text generation, and question answering. This often entails modifying the whole or only top layers of the pre-trained model, and additional training passes to adjust the model weights. Fine-tuning requires *labeled* samples for the given task, and is done in a supervised fashion. The downstream tasks are abundant and keep emerging with time, e.g., new domains, topics or data

distributions. The distribution of new data may differ significantly from the training data. For instance, next word prediction (seq2seq) on messages from user's social network apps [83], where such message corpora were not used for pre-training.

The state-of-the-art NLP models that follow the pre-training workflow are transformer-based, e.g., BERT [20] and its variants [9, 31, 48, 64, 69, 85]. Those models are composed of many transformer blocks, where each block extensively uses attention mechanisms [73]. We refer readers to recent surveys [25, 26, 37] for how those models work internally. This work specifically targets the fine-tuning stage of transformer-based models for its pivotal role in modern NLP services.

2.2 Federated learning

The fine-tuning is often performed on data generated by applications in user devices, e.g., input methods [83], emails [42, 68], and instant messaging [66]. Those data is private by nature and cannot be collected arbitrarily to respect user's privacy concern and legal regulation like GDPR [74]. Federated learning (FL) [35] addresses this need by enabling many devices to collaboratively train a shared model without giving away their data. The key idea is to decentralize the training over devices and only ask them to share model updates instead of raw data. For this reason, the benchmark for NLP training in a federated setting is emerging [47].

System model A pre-trained transformer-based language model is given as input. After that, our task is to fine-tune the model in a federated environment for an unbounded number of unforeseen tasks may emerge. For each task, the fine-tuning initiator (or *developer*) specifies how the last output layer shall be revised (e.g., number of classes).

The fine-tuning mostly follows an FL common practice [13]. In each round, a cloud service (or *aggregator*) selects a fixed number of clients as participants. The model is fine-tuned on each participant and the model updates will be uploaded/aggregated on the cloud. The aggregation is often lightweight and the cloud resources can be flexibly scaled out, so its time cost could be neglected. The fine-tuning speed is mainly bottlenecked by the on-device training time and the network transmission time. The most likely network for FedNLP is WiFi, which is highly unstable and constrained from a hundred Kbps to a few Mbps [82]. We do not consider the device memory to be an obstacle because: (1) Modern mobile devices with many GBs of DRAM can support fine-tuning tasks for BERT (BS=8) according to our experiments; (2) Memory inefficiency can be compensated with acceptable training overhead through advanced techniques [15, 53].

The key metric concerned in this work is time-to-accuracy, a widely adopted metric [17] that indicates the training time taken to reach a target accuracy. This is more practical and comprehensive than a single "time to full convergence" because the accuracy improvement per time slice dramatically decreases when the model approaches full convergence. For instance, when FL fine-tuning on task `AGNEWS`, it takes only 5.2 hours to obtain 80% accuracy but another 25.9 hours to 90%. In our system model, the target accuracy could be preset by the developer so the cloud service will train multiple rounds using FL until the target is met.

We impose no constraint on client selection [40, 43, 45, 52, 76, 81, 88] or training data sampling [44, 76] strategies, making it compatible with a mass of recent FL system literature.

2.3 Preliminary Measurements

We perform preliminary experiments that highlight the motivations to improve FedNLP fine-tuning performance and provide implications for the design of `AutoFedNLP`.

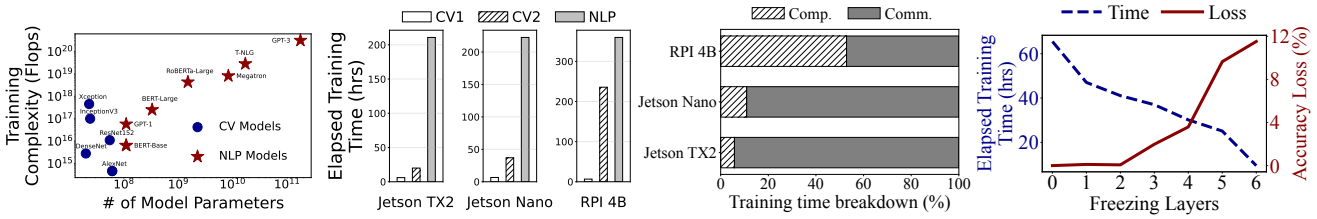
Observation-1: Transformer-based NLP models are highly costly. As illustrated in Figure 2a, transformer-based NLP models are much more expensive than the classic vision models in general in consideration of parameter numbers and computing complexity. BERT large has 330M trainable weights and takes 250,000 PFLOPs to train, which is 6×/23× higher than ResNet-152, respectively.

Observation-2: FedNLP task is extremely slow. Figure 2b shows the end-to-end training time towards full convergence for typical NLP and CV tasks under federated setting. As observed, it takes up to 210.74–359.7 hours to train on `SEMEVAL` dataset, which is 1.53–10.53× longer than training ResNet-56 on CIFAR-100. Note that dataset `SEMEVAL` used for classification tasks has 19 labels, while `CIFAR-100` has 100 labels. It's also worth mentioning that the above CV tasks are launched from scratch while the NLP tasks are fine-tuned atop a well pre-trained model.

Observation-3: network transmission dominates the training delay on high-end devices. The training time spent towards model convergence is dominated by two parts: on-device training and network transmission. Figure 2c shows such breakdown on three kinds of hardware (Jetson TX2 [1], Jetson Nano [2], and Raspberry Pi 4B [3]) that span a wide spectrum of hardware capacity and 1MB/s network bandwidth (both uplink and downlink). It shows that for a high-end edge device like Jetson TX2, the network transmission delay is the major bottleneck (about 94.22%) of FedNLP tasks. On a relatively wimpy device, both two parts contribute nontrivially to the total training time.

Observation-4: existing techniques are inadequate for FedNLP. A common approach to reducing the training cost in NLP fine-tuning is freezing a few bottom transformer blocks [23, 47]. It literally reduces the training computations by early stopping the backward propagation and the network cost by only sending the trainable parameters. Figure 2d shows the tradeoffs between the convergence time and training accuracy loss by tuning the number of frozen transformer blocks on DistilBERT and `ONTONOTES`. Unfortunately, we observed that the profits of such an approach are modest. For instance, to guarantee an acceptable accuracy loss (e.g., $\leq$1%), 2 out of 6 transformer layers can be frozen at most and only 33.3% (2/6) network traffic can be saved.

Implications FedNLP is slow due to the considerable amount of time spent on data transmission and local training. Simply freezing part of the model brings only modest improvement. While cellular network capacity keeps upgrading, their costly nature hinders adoption in federated tasks [16]. To enable practical FedNLP with a tolerable convergence delay (e.g., a few hours), the model structure and training paradigm need to be re-architected.



(a) CV models vs. NLP models (b) FL convegence time (c) Breakdown of FedNLP (d) Layer freezing in FedNLP

Figure 2: The preliminary measurement results of FedNLP. (a) A glance at the complexity of NLP models and traditional CNNs; (b) End-to-end convergence time of CV and NLP models under FL settings (CV1: "Densenet-121 [34] + CelebA [49]"; CV2: "Resnet56 [28] + Cifar-100 [39]"; NLP: "BERT [20] + Semeval [29]"). (c) Training time breakdown of FedNLP tasks on different hardware. Model: BERT; batch size: 4. (d) The performance of layer freezing. Model: DistilBERT [64]. Dataset: ONTONOTES [57]; batch size: 4.

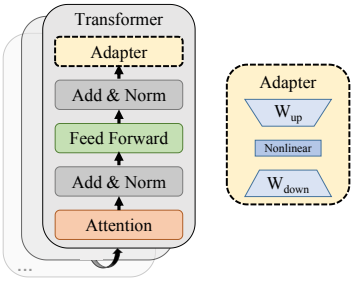


Figure 3: The structure of adapters used.

3 DESIGN

3.1 Plugable Adapters

Transformer adapters For efficient FedNLP, we retrofit adapters – a recently proposed technique for both CV and NLP tasks to achieve parameter efficiency in machine learning [32]. The initial goal of adapters is to reduce the tunable parameters especially in continuous learning [18] scenario where unlimited number of new tasks might emerge. However, it has been seldomly used to tackle system challenges like network cost and convergence speed. As far as we know, `AutoFedNLP` is the first to apply adapters to federated NLP tasks and demonstrate its efficiency under a system context.

The key idea of adapter is to freeze the whole original model but insert a few small modules into different locations inside it. Figure 3 shows the bottleneck architecture of our adapters and how it's applied to the transformer. Recall that each transformer layer contains two primary sub-layers: an attention layer and a feedforward layer, followed immediately by layer normalization. A residual connection is applied across each of the sub-layers. The adapter approach inserts small modules (adapters) between transformer layers. The adapter layer generally uses a down-projection with $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{n \times m}$ to project the input h to a lower-dimensional space specified by bottleneck dimension m, followed by a nonlinear activation function $f(\cdot)$, and a up-projection with $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{m \times n}$. These adapters are surrounded by a residual connection, leading to a final form as:

$$\boldsymbol{h} \leftarrow \boldsymbol{h} + f\left(\boldsymbol{h}\boldsymbol{W}_{\text{down}}\right)\boldsymbol{W}_{\text{up}} .$$

Considering that the same adapter inserted into the feed-forward network always outperforms its attention counterpart [27], we follow prior work [54], a more efficient adapter variant to only insert one adapter module after the second sub-layer, i.e., the feed-forward network "add & layer norm" sublayer. The output of the adapter is then passed directly into the next transformer layer.

The rationales behind adapters Why adapter is able to achieve comparable accuracy with much fewer parameters than freezing the bottom transformer layers without revising the model structure? We reason it with two insights from our experiments and related literature [54, 63].

First, adapters allow modifying a model's hidden state at a low cost. By keeping the whole original model as it is, adapters can maximally preserve the knowledge learned from the pre-training dataset. The pluggable adapters are only used to encode task-specific representations in intermediate layers of the shared model. With a bottleneck architecture, an adapter contains much fewer parameters than a whole transformer block, presuming that adapters can be inserted into "deeper" transformer blocks. While in fine-tuning scenario, the downstream tasks mostly share low-level feature representation with the pre-training task, it's still beneficial to adjust the low and middle-level feature extractor. Second, we observe that using adapters stabilizes the convergence process, while fine-tuning on the full model easily goes to overfitting. Though the overfitting can be remedied by carefully tuning the hyper-parameters such as learning rate and batch size, it also requires non-trivial efforts for each separated fine-tuning task.

Network cost analysis The trainable parameter number per adapter is $2mn + n + m$. Clients only send those parameters and last-layer classifier parameters after on-device training to the cloud aggregator. Therefore the network transmission per round is reduced to $D \times (2mn + n + m) + n \times \#labels$, where D is the total number of transformer blocks of the NLP model. As shown in Table 1, compared to fine-tuning the whole BERT model, the network saving could be more than 99%.

Compute cost analysis The computation FLOPs of each adapter in forward pass is $2 \times m \times n \times seqlen$ (normalized to single data sample), where $seqlen$ is the sequence length (default 256 in BERT). This incurred overhead is trivial compared to the original model complexity, e.g., less than 1% on BERT. On the other hand, since all other parameters are fixed during training, the computation during backward propagation is reduced by skipping calculating most of the weights' gradients. As shown in Table 1, using adapter brings around 40% training time reduction.

Model	Method	Training Time	Updated Paras.
BERT	Full Fine-tuning	1.86 sec	110.01 x 10^6
	Adapter	1.14 sec	0.61 x 10^6
DistilBERT	Full Fine-tuning	0.91 sec	67 x 10^6
	Adapter	0.56 sec	0.32 x 10^6

Table 1: Computation and communication cost of inserting adapters into each transformer block (width=32) and full-model tuning. Batch size: 4. Device: Jetson TX2.

Model	Datasets	Optimal adapter configuration (depth, width) towards different target accuracy				
		99%	**95%**	**90%**	**80%**	**70%**
BERT	**20news**	(2,64)	(2,32)	(2,8)	(2,8)	(2,8)
	agnews	(3,16)	(2,16)	(2,8)	(0,8)	(0,8)
	semeval	(10,8)	(6,8)	(6,8)	(2,8)	(2,8)
	ontonotes	(12, 32)	(12, 32)	(10, 32)	(0, 16)	(0, 16)

Table 2: The optimal adapter configuration (i.e., best time-to-accuracy) for different target accuracy (ratio to the full convergence accuracy) and different datasets.

3.2 The Configuration Challenge

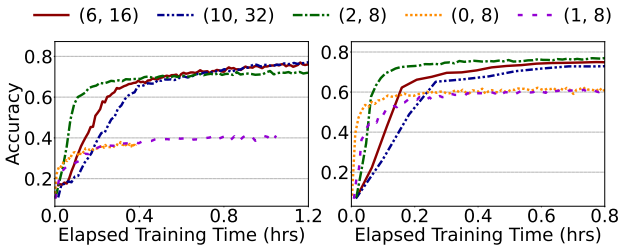
A unique challenge raised by adapters is its sensitivity to the configurations (explained below). Different adapter configurations result in a variety of convergence delays, up to 4.7× gap. Choosing an "optimal" configuration towards fast convergence is fundamentally challenging for the following reasons.

Large adapter configuration space There are two critical parameters of adapters to be determined: depth and width. (1) Similar to the idea of layer freezing, adapters are not necessarily inserted into each transformer block. Reducing the number of adapters inserted into the top blocks (namely *tuning depth*) can effectively reduce the network cost and on-device training time. (2) Apart from the depth, the bottleneck size (*tuning width*), i.e., the target projection dimension of input needs to be carefully set as well. A small width might not suffice to encode the latent features for fine-tuning tasks and thus incurs high accuracy degradation. Yet, a too wide adapter incurs high resource costs and therefore slows down the training (i.e., increased time to accuracy). Overall, the candidate depth spans from 0 to the number of transformer layers even if we only consider inserting adapters at top K consecutive layers, i.e., 12 in BERT, and the valid widths range from 8 to 64 according to our experiments. That results in hundreds of different alternative configurations.

Another dimension of design space is that the configuration can be switched across FL rounds during a training session. §3.3 elaborates how such switching could be realized with the knowledge learned by the old adapter configuration well preserved. But within a round, clients better use the same configuration to facilitate the model aggregation.

Decisions must be online Making a good decision offline is difficult without pre-knowledge about the training dataset – a common setup in fine-tuning scenarios. Even with the same task, with the data distribution drifting over time, the resultant model structure could differ tremendously [31, 43].

No silver bullet configuration A key observation we made from extensive experiments is that *there is no silver-bullet configuration for FedNLP tasks*. Even with a given pre-trained model, there are many factors that affect which configuration shall be picked to achieve the fastest convergence.



(a) SEMEVAL **(b) 20NEWS**

Figure 4: Across different target accuracy and target FedNLP tasks, the optimal adapter configuration (depth, width) varies. Model: BERT; device: Jetson TX2.

• *Targeted accuracy.* Within a training session, different target accuracy favors different configurations. As shown in Figure 4 (a), to achieve the best accuracy possible, using an adapter with depth 6 and width 16 is the best option. If 80% relative accuracy is satisfactory, the adapter with depth 2 and width 8 is 2× faster than the previous configuration.

• *Targeted NLP tasks.* Across different FedNLP tasks, the optimal adapter configuration varies. As shown in Figure 4, using depth 2 and width 8 leads to fast convergence to 80% accuracy on `20NEWS` [41]. However, on `SEMEVAL` [29], the same configuration results in 10% lower convergence accuracy as compared to a more complex configuration.

• *Client resources.* Local clients' training speed and network capacity also make a difference in adapter selection. This is due to the disparate impacts of adapter depth/width on the computation and communication reduction. For instance, a larger tuning depth linearly amplifies the communication and computation cost, while a larger tuning width linearly increases the communication cost but only adds negligible computation cost according to the analysis in §3.1.

Why prior work is inadequate A closely related technique is neural architecture search (NAS) [21], which automatically looks for the best model structures but with a totally different design goal. Essentially, NAS sacrifices the time for good training accuracy, e.g., days to train a single model in a centralized manner [78]. Instead, we pursue fast time-to-accuracy, which is a more practical and affordable setting for FedNLP developers.

3.3 The Online Configurator

We build a configurator that automatically adjusts the tuning depth and width throughout a training session. The goal is fast model convergence: achieving the target model accuracy in the shortest time. Our key ideas are twofold:

• *Progressive training.* The first key idea is, to begin with a shallow tuning configuration (i.e., small depth and width) to quickly boost the model accuracy. When it encounters a "choke point" where more rounds of training no longer provide enough accuracy profit, it "upgrades" to a more complex configuration, i.e., either deeper or wider.

Such upgrading mechanism is inspired by curriculum learning [11], a learning strategy that trains a model beginning from easier data samples to harder ones. Instead of altering the training

samples, we propose to alter the model structure. In the beginning, a simpler adapter configuration can learn fast. This is because, by focusing on fewer compact trainable parameters closer to the model output, the model can rapidly learn the coarse-grained domain-specific knowledge for the downstream tasks, such as new class labels [62]. For simple downstream tasks, fine-tuning without re-learning deep features is enough to obtain satisfactory model accuracy, e.g., depth 2 and width 64 for 20NEWS dataset [41]. As the training proceeds, the model encounters a "choke point" where the learning curve becomes gentle or even totally flat. It demands deeper or wider adapters to learn new features. The experiment results in Table 2 attests to our claim that a higher target accuracy favors deeper and wider adapters.

• *Identifying timing and direction to upgrade configuration through sideline trials.* The learning curve is fundamentally challenging to be estimated or predicted ahead of time. How can a system possibly know it is time to upgrade the adapter configuration and in which direction shall it upgrade (deeper or wider)? In this work, we propose an intuitive approach based on the concept of sideline trials. Its key idea is to ask extra participant clients to attempt different tuning depths and widths, and make a decision on whether and where to upgrade based on the tested accuracy of different directions. In federated settings, such "extra clients" are common because the client-level parallelism of existing FL algorithms is notoriously low. That is, limited by the learning theory [36], a small number of clients (i.e., 5 for `20NEWS`) is enough to saturate the convergence performance (both accuracy and speed) and allocating more clients gives a negligible return. As we observe in extensive experiments, using those extra clients for trial is much more beneficial than asking them to participate in training.

Configurator algorithm in detail Algorithm 1 shows how our `AutoFedNLP` wisely upgrades the configuration of adapters during a training session. Unlike the traditional FL scheme where only one global model with a fixed structure undergoes the training, in `AutoFedNLP` the cloud aggregator periodically dispatches the global model to three groups of clients: one is to train with the current configuration, one with a deeper one and the other with a wider one (line 2–5). After a few rounds of parallel training (line 20–23, 7–9, 18), the aggregator server checks the accuracy of three global models and re-starts the process on the model with the highest accuracy (line 12–16). Note that when the aggregator checks the accuracy, the three global models undergo different numbers of global rounds because the per-round training time and network time depend on the adapter configuration (§3.1). Therefore, the training speed of different tuning depth/width is considered in this mechanism. Except that, the clients and aggregator follow the common FL process in local training (line 20–23) and model aggregation (line 8–9).

As described in Algorithm 1 (line 24–33), the models dispatched to different groups are with different model configurations. Group $Trial_0$ inherits the learned adapter from the previous winner track whereas $Trial_1$ and $Trial_2$ also inherit the old adapters but add extra depth and width, respectively. The step sizes of depth and width are pre-defined, e.g., $S_d = 1$ and $S_w = 8$ by default, respectively. The depths and widths are both added on the fly. All newly added weights are normalized with mean (0.0) and stddev (0.02). We

Algorithm 1: Our Online Configurator

input : Target accuracy, acc;
Trial interval, $trial\_intvl$;
Start-up depth and width, D_0 and W_0;
Step of depth and width, S_d and S_w.
output : Fine-tuned adapter weights, Θ_i (i=1,2…).

1 **Function** *Cloud_controller():*
2 $Trial_0, Trial_1, Trial_2 \leftarrow$ selects $3N$ clients;
3 i=0;
4 T_{trial}=T_{now};
5 Dispatch(i); // init model and client training
6 **while** *Eval()* < acc **do**
7 i++;
8 $\Theta_i^k(n) \leftarrow$ Recieve from clients;
9 $\Theta_i^k \leftarrow$ Fedavg($\Theta_i^k(n)$);
10 $Trial_0, Trial_1, Trial_2 \leftarrow$ selects $3N$ new clients;
11 **if** T_{now} - T_{trial} > $trial\_intvl$ **then**
12 Compare accuracy under different Θ_i^k;
13 $\Theta_i \leftarrow$ The winner track;
14 $D_i, W_i \leftarrow$ The winner setting;
15 T_{trial}=T_{now};
16 Dispatch(i);
17 **else**
18 Send the aggregated model Θ_i^k to $Trial_k$.
19 Exit training.
20 **Function** *Client_training(i,k):*
21 $\Theta_i^k \leftarrow$ Receive global adapter from Server;
22 $\Theta_{i+1}^k(n) \leftarrow$ Train and update local adapter;
23 Send updated adapter $\Theta_{i+1}^k(n)$ to Server.
24 **Function** *Dispatch(i):*
25 **if** *i=0* **then**
26 $F(D,W)$: $\Theta_0 \leftarrow$ Initial with D_0, W_0.
27 **else**
28 $F(D,W)$: $\Theta_i \leftarrow$ Inherit from the winner track with D_i, W_i.
29 $\Theta_i^0 \leftarrow F(D_i,W_i)$;
30 $\Theta_i^1 \leftarrow F(D_i + S_d,W_i)$;
31 $\Theta_i^2 \leftarrow F(D_i,W_i + S_w)$;
32 Sends $\Theta_i^0, \Theta_i^1, \Theta_i^2$ to $Trial_0, Trial_1, Trial_2$ seperately;
33 Parallel: Client_training(i,k).

experiment with two ways to expand the adapter width: vertical and parallel stacking. Our micro experiments show that vertically stacking will outperform parallel stacking with the same amount of parameters. So we always use this method.

Integration with the existing FL framework `AutoFedNLP`'s trial groups are compatible with how existing FL frameworks manage clients for training efficiency, a key system component having received high research attention [40, 44, 45, 52, 76, 81, 88]. This is because the adapters and their configuration scheduler are intentionally designed to be decoupled from which device or data will be involved in per-round training.

4 FEDNLP ACTIVATION CACHE

Using adapter (§3.1) significantly reduces the network cost in FedNLP tasks, exposing client-side computation as the next major bottleneck for model convergence. Layer-freezing reduces the training computations by early-stopping the backward propagation, but does not address the computation cost at forward pass. For instance, with tuning depth as 2 of 12 transformer blocks in BERT, the forward computation takes $12/(12 + 2 * 2) = 75\%$ of the total computation, considering that the backward propagation computation of each layer is approximately 2 times of the forward counterpart [58].

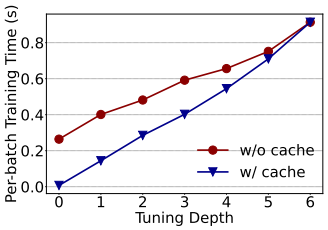


Figure 5: Per-batch training time with and without activation caching. Model: DistilBERT. Device: Jetson TX2.

Opportunities To reduce the client computation, we exploit two unique FedNLP opportunities: (i) Throughout a training session, a device participates in many FL rounds. Across rounds, the device executes local training on the same set of local samples. In typical FL scenarios, it takes tens of thousands of rounds till model convergence [13]. (ii) During on-device training, the weights of the bottom transformer blocks that do not have adapters inserted are fixed. Moreover, those untouched transformer blocks remain the same across FL rounds till `AutoFedNLP` switches the tuning depth (§3.3). In our experiments we observe a tuning depth upgrading interval typically at hundreds to thousands of rounds.

The activation cache Our key idea is to leverage both the static input and the fixed bottom layers so a client's computed activations can be reused across rounds. Assuming the cloud aggregator selects a device to train with depth d_{prev} at round r_{prev}. During the training, the device also extracts the output of the last fixed bottom layer for each input batch, i.e., the output of the $(D - d_{prev})_{th}$ transformer layer and stores them in local storage. For the next time device k is selected, it first inquires the cloud aggregator that how deep the model has been tuned since the last time it's selected, i.e., $d' = \max(d_{prev}, d_{prev+1}, \ldots, d_{now})$. If $d' > d_{prev}$, it means the model has gone deeper. The transformer layers between $(D - d')_{th}$ and $(D - d_{prev})_{th}$ have been updated and the cached activations of $(D - d_{prev})_{th}$ transformer block have "expired". The output of the $(D - d')_{th}$ transformer layer needs to be recomputed and recached. Otherwise, if $d' \leq d_{prev}$, the first $d_{split} = D - d_{prev}$ transformer layers are not touched since round r_{prev}. Therefore, it can directly load the output of the $(D - d_{prev})_{th}$ layer from the cache and feed it into the training process without starting from scratch. The above process repeats when the device participates in training every time.

The cache expiration incurs re-computations of bottom transformer blocks and compromises its profits. Fortunately, `AutoFedNLP`'s design of online configurator (§3.3) orchestrates with the caching technique by monotonously upgrading the tuning depth. The cache only expires when the tuning depth increases, e.g., at most 12 times for BERT, which is negligible in the end-to-end fine-tuning process.

Computation cost analysis Using activation caching reduces the computations by d_{split}/D at the forward pass. Figure 5 shows per-batch training time on devices with various tuning depth. With activation caching, the training time decreases almost linearly with fewer adapter layers to be updated ($D - d_{split}$), and the improvement from caching mechanism increases significantly as well.

Storage and I/O cost analysis Caching the activations takes extra storage, i.e., $seqlen \times n \times BS$ per batch, where $seqlen$ is the max sequence length (default 256), n is the transformer's internal feature size (default 768), and BS is the batch size (default 4). The cache is reloaded from disk per minibatch, taking no more than tens of ms on embedded flash and incurs less than 2% overhead. The total cache size is also proportional to the number of batches samples per client (typically dozens). Assuming 100 training samples, the storage cost is calculated to be around 93.75MB. Such cost is no more than 1% of the storage of a modern mobile/embedded device, ranging from tens to hundreds of GBs. Note that the cache can be cleared once the FL process finishes.

Device	Processor	Per-batch Latency (s)
Jetson TX2 [1]	256-core NVIDIA Pascal™ GPU.	0.88
Jetson Nano [2]	128-core NVIDIA CUDA® GPU.	1.89
RPI 4B [3]	Broadcom BCM2711B0 quad-core A72 64-bit @ 1.5GHz CPU.	18.27

Table 3: Development boards used in experiments.

5 IMPLEMENTATION AND METHODOLOGY

We have fully implemented a `AutoFedNLP` prototype atop `FedNLP` [47] and `Adapterhub` [55]. `FedNLP` is the state-of-the-art framework for evaluating federated learning methods on different NLP tasks. `AdapterHub` is a library that facilitates the integration of pre-trained adapters for different tasks and languages. As prior work [13], we adopt the parameter server (PS) architecture among the clients and central server. At the server side, once job is submitted by the developer, the server initializes the pluggable meta adapter to be trained (through the API of `Adapterhub`) into the pre-trained model. The server also splits the initialized meta adapter into three branches: normal, wider and deeper. The wider branch will stack a few meta adapters parallel to expand the bottleneck size of adapter in single layer. The deeper branch will insert the meta adapter into one more deeper layer. A client selector will sample 3N clients from available devices and shuffle them into 3 groups. We now employ a random client selector (default in most FL literature) but more advanced selection strategies [40, 43–45, 52, 76, 81, 88] can be plugged into our implementation as well. Then, the server sends three branches of adapters to three groups separately via MPI (in standalone mode) or WLAN/Cellular (in distributed mode). Once receiving the adapters, the clients insert the adapter into their local pre-trained model. They fine-tune the model with their own private data. During the fine-tuning process, the cache mechanism is used to avoid duplicate computation. The trained adapters will be collected in the central server and aggregated through *FedAvg* algorithm [51]. All clients run in synchronized mode [30].

Metrics, hardware, and environment We mainly report the time-to-accuracy metric. To evaluate the model accuracy, we divide the dataset of each device into training set (80%) and testing set (20%). For clarity, we pay attention to a few typical accuracy targets, e.g., 99%, 95%, 90% of the full convergence accuracy achievable by the baseline that fine-tunes the whole model. We refer to those accuracy numbers as *relative target accuracy*. For example, the 100% relative target accuracy of BERT is 0.8 (accuracy) for `20NEWS`; 0.9 (accuracy) for `AGNews`; 0.8 (accuracy) for `SEMEVAL`; and 0.75 (token-F1) for `ONTONOTES`. Besides, we also report the resource cost imposed in an FL process, including the total amount of energy consumed on data transmitting and training computation on each client; the total amount of network traffic; and the peak memory usage.

Task	Dataset	# of Clients	Labels	Non-IID	Samples
TC	20NEWS [41]	100	20	/	18.8k
TC	AGNEWS [87]	1,000	4	a=10	127.6k
TC	SEMEVAL [29]	100	19	a=100	10.7k
ST	ONTONOTES [57]	600	37	a=10	5.5k

Table 4: Datasets and settings used in experiments for two tasks: Text Classification and Sequence Tagging. "a" is a parameter that controls the datasets' non-IID level [47].

As prior FL literature [47], our experiments are carried out in an emulation manner on a GPU server with 8x Nvidia A40 and 8x Nvidia V100. The on-device training time is obtained on 3 development boards with similar hardware capacity to mainstream mobile devices, i.e., Jetson TX2 [1], Jetson Nano [2], and Raspberry Pi 4B [3]. The numbers are then plugged into the emulation framework to calculate the wall clock time. The default network bandwidth between clients and server is set to 1MB/s as reported by prior studies [4, 24]. In §6.1, we will quantify the performance of AutoFedNLP under various hardware and bandwidth settings (100Kbps–10Mbps).

Models We use two representative models for FedNLP tasks: BERT [20] (default) and its varient DistilBERT [64]. BERT and DistilBERT is composed of 12 and 6 transformer blocks, respectively. DistilBERT leverages knowledge distillation during the pre-training phase and reduces the size of a BERT model by 40%, while retaining 97% of its language understanding capabilities and being 60% faster. We use BERT for most of our experiments, as all BERT-based variants derive from it. The pre-trained weights of both models are downloaded directly from Hugging Face [79].

Tasks and datasets We evaluate AutoFedNLP on 4 classic NLP downstream datasets as shown in Table 4. We follow the approach in [47] to build the non-IID datasets. (1) 20NEWS (IID) [41] dataset is a collection of approximately 20,000 newsgroup documents. (2) AGNEWS (non-IID) [87] is a collection of 127.6K news articles gathered from more than 2,000 news sources. (3) SEMEVAL (non-IID) [29] is a relation classification datasets which assigning predefined relation labels to the entity pairs that occur in texts. The above 3 datasets are used for text classification (TC) [68] tasks, where the input is a sequence of words (e.g., document contents) and the output is a label in a fixed set of label set (e.g., political, sports, and entertainment). (4) ONTONOTES (non-IID) [57] is a corpus where sentences have annotations for the entity spans and types. We use it for the named entity recognition task, which is fundamental to information extraction and other applications. This dataset is for sequence tagging (ST) [8] task, where the input is a sequence of words and the output is a same-length sequence of tags. ST tasks are widely used in real world such as analyzing syntactic structures, e.g., part-of-speech analysis, phrase chunking, and word segmentation.

Baselines We compare AutoFedNLP to the following alternatives. (1) Vanilla Fine-Tuning (FT) always fine-tunes the whole model on each client. This is the default fine-tuning methodology used in most NLP literature [20, 64]. (2) Fine-Tuning-Quantized (FTQ) quantizes the model parameters from FP32 to lower precision to reduce the network traffic between clients and aggregator. Quantization is one of the most widely adopted approaches to reduce the communication cost and speedup FL process. We use a state-of-the-art quantization algorithm [80] in our FedNLP tasks. According to the algorithm, we observe the NLP models are quantized to INT4 or INT8 adaptively. (3) LayerFreeze-Oracle ($\texttt{LF}_{oracle}$) freezes a few transformer layers at bottom and only fine-tunes the ones above. This is widely used to reduce the fine-tuning cost [23, 47]. The number of freezed layers is selected per task to achieve the best time-to-accuracy at 99% relative accuracy target. (4) LayerFreeze-Quantized-Oracle ($\texttt{LFQ}_{oracle}$) combines the above quantization and freezing techniques and selects the best setting for each task, i.e., the number of freezed layers and the quantized data precision. To be noted, $\texttt{LF}_{oracle}$ and $\texttt{LFQ}_{oracle}$ are impractical in reality as they require prior knowledge to obtain an oracle system parameter. For a fair comparison, all baselines use the same model aggregation algorithm (*FedAvg* [51]) and client sampling (random), which are also the default setting in prior FL literature [47].

Hyper-parameters Unless otherwise stated, AutoFedNLP and all baselines use the same set of hyper-parameters as FedNLP [47] framework: mini-batch size as 4; local training iteration as 1; learning rate as 0.1; max sequence length as 256 for 20NEWS and ONTONOTES, 64 for AGNEWS and SEMEVAL. For the FL configurations at the server side, we follow the prior FedNLP literature to select 15 participants by default for each training round, i.e., 5 clients in each trial group of AutoFedNLP (§3.3).

6 EVALUATION

6.1 End-to-end Performance

In this section, we mainly compare the time-to-accuracy of AutoFedNLP and baselines under various datasets, models, network bandwidths, and device hardware.

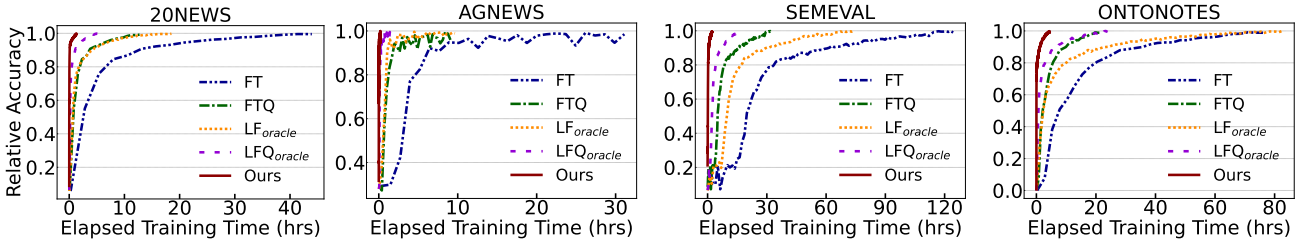
AutoFedNLP reduces model convergence delays significantly, making FedNLP practical. Table 5 summarizes the convergence time and Figure 6a illustrates the convergence process under the default setting. To reach 99% relative target accuracy, AutoFedNLP is 33.8×, 155.5×, 54.0× and 16.9× faster than FT on the four datasets, respectively. With a lower target accuracy such as 90%, the speedup brought by AutoFedNLP is even more significant, i.e., 27.4×–260.0×. It takes at most one hour for AutoFedNLP to reach a usable accuracy.

More competitive baselines $\texttt{LF}_{Oracle}$ and FTQ only bring limited improvement over FT, i.e., 2.4×–3.9× speedup. Dozens of hours are still needed for a single downstream task. $\texttt{LFQ}_{Oracle}$ can benefit from both layer-freezing and quantization, therefore performing better than other baselines. Though, AutoFedNLP can still beat $\texttt{LFQ}_{Oracle}$ nontrivially, especially for reaching a relatively lower target accuracy. For example, AutoFedNLP is 12.8× faster on SEMEVAL to reach 90% relative target accuracy. This is because AutoFedNLP employs an upgrading mechanism on adapter configuration which enables fast boosting of the training accuracy. Note that both $\texttt{LF}_{Oracle}$ and $\texttt{LFQ}_{Oracle}$ are not practical methods as they use the "optimal" tuning depth which is not likely to be known beforehand. If a non-optimal depth is chosen, their training performance will drastically deteriorate.

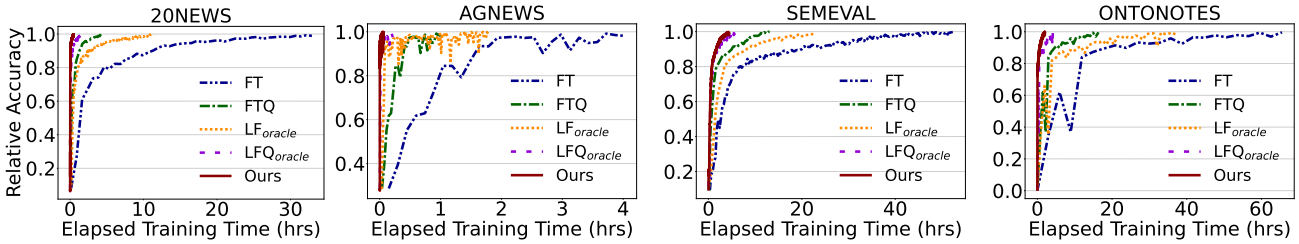
We also extend our experiments to DistilBERT [64], a distilled version of BERT, and illustrate the results in Figure 6b. It shows that AutoFedNLP significantly outperforms the baselines on DistilBERT as well. For instance, AutoFedNLP achieves 14.89×–73.42× speedup over FT to obtain the 99% relative target accuracy. Interestingly, comparing DistilBert to BERT under FT, we find the former achieves

Datasets	20NEWS			AGNEWS			SEMEVAL			ONTONOTES		
Relative Accuracy	99%	95%	90%	99%	95%	90%	99%	95%	90%	99%	95%	90%
FT	44.0	23.4	13.1	31.1	10.1	5.2	124.3	89.9	61.7	76.1	55.9	35.6
FTQ	12.7	6.8	3.8	9.1	2.6	1.7	32.0	23.1	15.9	21.2	15.5	9.9
LF_{oracle}	18.5	8.1	4.3	9.6	1.4	1.1	74.0	46.8	33.2	82.5	43.8	24.5
LFQ_{oracle}	5.2	2.5	1.1	1.6	0.3	0.2	16.8	11.0	7.7	23.9	12.9	7.2
Ours	**1.3**	**0.4**	**0.1**	**0.2**	**0.03**	**0.02**	**2.3**	**1.1**	**0.6**	**4.5**	**2.4**	**1.3**

Table 5: Elapsed training time taken to reach different relative target accuracy. NLP model: BERT-base. Unit: Hour.

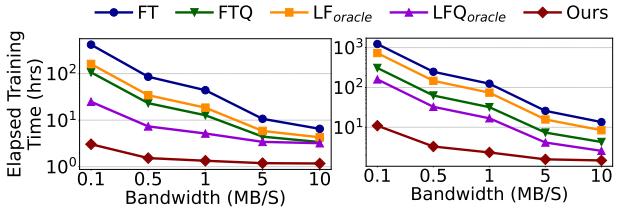


(a) BERT-base



(b) DistilBERT

Figure 6: Time-to-accuracy throughout a training session. AutoFedNLP speeds up model convergence significantly.



(a) 20NEWS. **(b) SEMEVAL**

Figure 7: Model convergence delays under different network bandwidths. Training targets 99% relative target accuracy. AutoFedNLP outperforms under all network bandwidths.

up to 7.7× speedup on AGNEWS on Jetson TX2 since it is more lightweight in consideration of both computation and communication. However, this advantage does not hold for all circumstances. Comparing Figure 6b (d) with Figure 6a (d), we find that BERT has comparable performance with DistilBERT. It is likely attributed that, when the downstream task gets harder, the loss of model's representation capacity during distilling incurs negative impacts. Such a difference will compensate the large model size and computation complexity of the vanilla BERT. Nevertheless, AutoFedNLP performs consistently on both models because it judiciously tunes the pluggable adapters with different depths and widths.

AutoFedNLP outperforms baselines in various network environments. Figure 7 reports the performance of AutoFedNLP and baselines under various network environment from 0.1MB/s to 10MB/s, which cover the typical network capacity for nowaday WiFi and cellular bandwidth. Our key observation is that AutoFedNLP consistently outperforms other baselines with different network conditions, and the improvement is more significant with lower network bandwidth. For instance, with $bandwidth = 10MB/s$, AutoFedNLP reaches the 99% relative target accuracy 5.6× and 9.2× faster than FT on 20NEWS and SEMEVAL, seperately. When the bandwidth goes down to 0.1MB/s, the improvement is as high as 137.7× and 112.5×, respectively. The rationale behind this is that AutoFedNLP brings the most network transmission reduction by inserting tiny adapter modules into the model. Such a micro transmission package makes the communication process fast even with very low network bandwidth. In reality, network fluctuation is common [81]. AutoFedNLP enables a stable fine-tuning process and paves the way for the fundamental solution to stragglers or other possible communication problems [61, 76] that will drag the NLP fine-tuning slow.

AutoFedNLP outperforms baselines on various client hardware. AutoFedNLP also consistently outperforms other baselines with different device capacity as shown in Figure 8. On GPU-powered high-end embedded devices like Jetson TX2 and Jetson Nano, AutoFedNLP reaches the 99% relative target accuracy up tp 32.8× and 79.2× faster than the VanilaFT on 20NEWS and SEMEVAL, respectively. On a much wimpy device RPI 4B, the speedup degrades

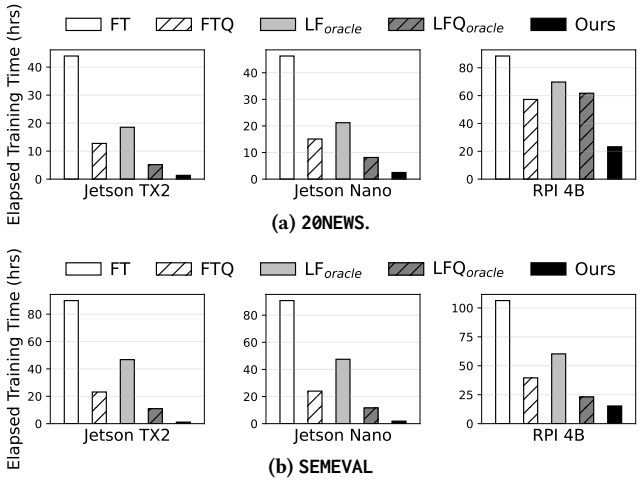


Figure 8: Model convergence delays with a variety of client hardware. Training targets 99% relative target accuracy. AutoFedNLP outperforms the baselines with all client hardware.

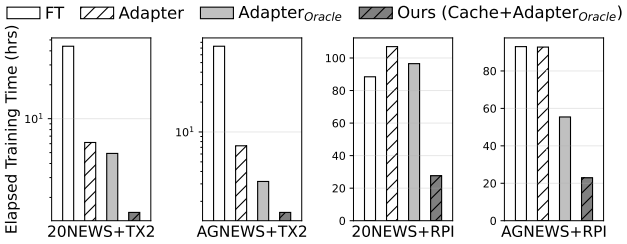


Figure 9: Model convergence delays with and without AutoFedNLP's key designs, showing their significance

to 3.8× and 7.0×, respectively. This is because AutoFedNLP reduces two orders of magnitude in communication cost, but only one order of magnitude in computation cost. Fine-tuning on computationally powerful devices is mostly bottlenecked by communication (Figure 2c), therefore AutoFedNLP brings the most benefit. Nevertheless, AutoFedNLP can still bring remarkable improvement on the wimpy devices. For example, on dataset 20NEWS, AutoFedNLP reaches target accuracy 5.03× and 3.25× faster than the four baselines, respectively. In practice, the participant devices are often with a mixture of heterogeneous hardware capacity [76] and the training is bottlenecked by the slower ones [61]. Extensive research efforts have been invested to mitigate the impacts of such device heterogeneity [40, 43–45, 52, 76, 81, 88] and AutoFedNLP is orthogonal to those techniques.

6.2 Significance of Key Designs

The benefits of AutoFedNLP come from: the use of adapters (§3.1), the activation cache (§4), and the automatic adapter configuration (§3.3). We now quantify their benefits.

Adapter and caching. Figure 9 shows benefit of adapters and caching. Naive use of adapters, e.g. inserting at all layers, may brings notable benefit, e.g., on Jetson TX2 and benchmark AGNEWS, Vanilla-Adapter is 10.1× faster than FT. However, the delays are still too high. On RPI 4B, naive use of adapters slows down the

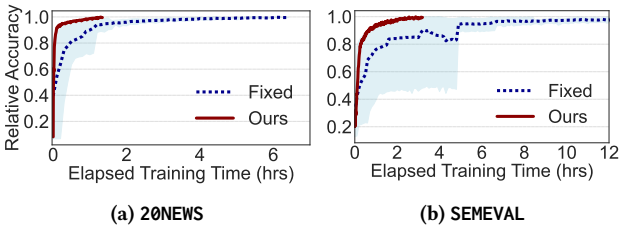


Figure 10: Time-to-accuracy throughout a training session. AutoFedNLP's accuracy (red lines) always outperforms those of fixed adapter configuration (208 in total, aggregated as blue shades, for which blue dotted lines show averages)

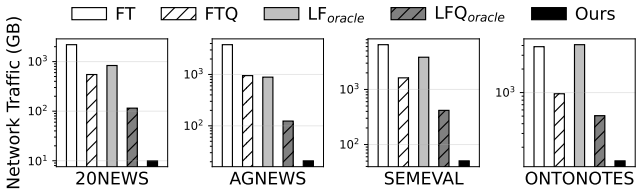


Figure 11: Network traffic (downlink and uplink) of all 15 client devices. Training targets 99% relative target accuracy. AutoFedNLP reduces network traffic significantly

model convergence as compared to fine-tuning the whole model. By using one static, oracle adapter configuration (Adapter$_{oracle}$), the time-to-accuracy is reduced by up tp 23.0× compared to FT. Employing the activation cache technique (Adapter$_{oracle}$+Cache) further brings 2.1×–3.3× speedup by reducing the on-device training time.

Automatic configuration To demonstrate the importance of AutoFedNLP's upgrading mechanism on the adapter's tuning configuration, we exhaustively sweep through all adapter configurations (depth 0–12, width 8,16,..,128, 208 configurations in total) of BERT on 20NEWS, and aggregate their convergence curves as shaded areas shown in Figure 10. The blue line (dotted) is the average time-to-accuracy of all configurations while the red line (solid) is the curve of AutoFedNLP. Note that sweeping all configurations is very expensive: it takes thousands of GPU hours to run the benchmark in a subfigure. The results show that AutoFedNLP almost outperforms every configuration throughout a training session. This is owing to AutoFedNLP switching among different configurations that best suits the current training session. Typically, we observe AutoFedNLP typically uses 8–14 configurations per training session.

6.3 Resource Cost

Network traffic. Figure 11 reports the total network traffic incurred during fine-tuning to reach 99% relative target accuracy. It shows that AutoFedNLP saves 126.7× on average and up to 220.7× (reducing from 2194.3 GB to 9.9 GB) network traffic compared to the FT on dataset 20NEWS. Note that reducing the network traffic not only speeds up the convergence, but also mitigates the overhead on clients and the monetary cost to FL developers, which is billed by the amount of data transmitted on public cloud platforms, e.g., $0.01/GB on AWS [5].

Energy consumption. Figure 12 illustrates the average energy consumed during FedNLP tasks on each device. It shows that AutoFedNLP can save the energy consumption remarkably, e.g.,

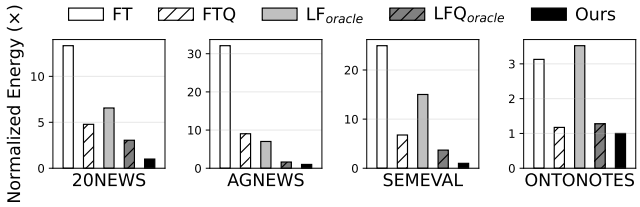
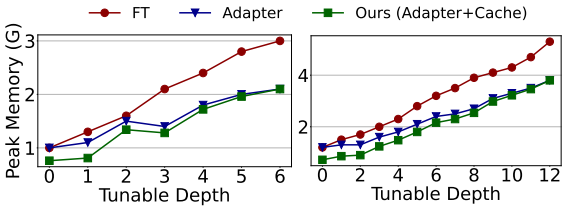


Figure 12: Per-client average energy consumption, normalized to that of AutoFedNLP. Training targets 99% relative target accuracy. AutoFedNLP reduces client energy usage significantly.



(a) DistilBERT. **(b) BERT.**

Figure 13: Peak memory usage of a client device. AutoFedNLP's key designs reduce client memory usage.

1.3×–3.7× reduction compared to LFQ$_{Oracle}$ and 3.1×–32.1× reduction compared to FT, respectively. Such improvement comes from both the reduced network transmission time and the on-device training computations. Again, it is worth mentioning that LFQ$_{Oracle}$ is built atop an unrealistic assumption that the optimal freezing layer number is known ahead of time.

Memory footprint Figure 13 reports the peak memory footprint when fine-tuning on BERT and DistilBERT with different tuning depths. It shows that AutoFedNLP nontrivially reduces the memory usage either with shallow or deep tuning depth. The reasons are twofold. First, AutoFedNLP only updates the parameters of a few adapters, so the gradients of other parameters are not calculated and the associated activations do not need to be stored. Second, our activation caching technique avoids part of the memory to store the parameters at the forward pass.

7 RELATED WORK

Fine-tuning (transfer learning) Inductive transfer learning has greatly advanced NLP research. Howard et al. propose ULMFiT [33], a universal transfer learning method matching the performance of training from scratch. BERT [20] was then introduced and becomes a standard pre-trained model in many NLP downstream tasks for its superior performance and generality. Numerous variants [9, 20, 31, 48, 64, 69, 73, 85] of BERT have since been designed. For instance, Sun et al. explore the space of strategies for fine-tuning BERT for text classification [68]. This work is motivated by those work and specifically targets FedNLP scenario.

FedNLP is a key step towards the adoption of NLP models in practice. However, there is very few literature investigating its implications at system aspect. [47] is the first research benchmark for federated learning in NLP tasks and integrates representative language datasets. AutoFedNLP is built atop it and treats it as a baseline. SEFL [19] is a FedNLP framework that achieves data privacy without any trusted entities. [10] studies how FedNLP can orchestrate with differential privacy. None of above work addresses the high training cost of FedNLP.

Adapters Adapter is extensively studied to achieve parameter efficiency in continuous learning tasks. It was first introduced for vision tasks [59]. The rationale is to encode task-specific representations in intermediate layers while preserving the knowledge learned from the pre-training dataset [54]. Besides, adapter in a reasonably small size steadily achieves convergence to the optimal accuracy and avoids overfitting [63]. Various adapter variants have been proposed to tradeoff trainable parameter numbers and training accuracy in NLP tasks [27, 46, 50, 56, 70]. Despite the popularity, the implications of adapter in FedNLP tasks have not been well examined. For the first time, we treat adapter as a building block to address the training performance issue in FedNLP.

Optimizations for FL Due to the decentralized nature, communication has been recognized as a major bottleneck in FL tasks [13, 84]. Various optimizations [12, 44, 75, 77, 80] have been proposed. Among them, model compression/quantization [12, 80] is the mostly adopted and is directly compared in this work. Apart from network transmission, data and device heterogeneity [60] are also unique challenges introduced in FL. To mitigate the heterogeneity of client devices (therefore stragglers), Abdelmoniem et al. [7] ask each client device to quantize their local model adaptively. Hermes [43] guides different mobile clients to find a small subnetwork through structured pruning for local training. Another line of those work focus on intelligent client selection and data sampling [40, 43–45, 52, 76, 81, 88]. AutoFedNLP instead takes the first fundamental step towards practical FedNLP, and is compatible with above techniques.

8 CONCLUSIONS

AutoFedNLP is a federated learning framework for fast NLP model fine-tuning. AutoFedNLP borrows the wisdom from prior work and uses adapter as the only trainable module in NLP model to reduce the training cost. To identify the optimal adapter configuration on the fly, AutoFedNLP integrates a progressive training paradigm and trail-and-error profiling technique. AutoFedNLP shows superior training speedup over existing approaches through our extensive experiments.

REFERENCES

[1] https://developer.nvidia.com/embedded/jetson-tx2.
[2] https://developer.nvidia.com/embedded/jetson-nano-developer-kit.
[3] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.
[4] The state of wifi vs mobile network experience as 5g arrives. https://www.opensignal.com/sites/opensignal-com/files/data/reports/global/data-2018-11/state_of_wifi_vs_mobile_opensignal_201811.pdf, 2018.
[5] Amazon ec2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand/, 2022.
[6] Julien Abadji, Pedro Javier Ortiz Suárez, Laurent Romary, and Benoît Sagot. Ungoliant: An optimized pipeline for the generation of a very large-scale multilingual web corpus. Proceedings of the Workshop on Challenges in the Management of Large Corpora (CMLC-9) 2021. Limerick, 12 July 2021 (Online-Event), pages 1 – 9, Mannheim, 2021. Leibniz-Institut für Deutsche Sprache.
[7] Ahmed M Abdelmoniem and Marco Canini. Towards mitigating device heterogeneity in federated learning via adaptive model quantization. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 96–103, 2021.
[8] Gizem Aras, Didem Makaroğlu, Seniz Demir, and Altan Cakir. An evaluation of recent neural sequence tagging models in turkish named entity recognition.

Expert Systems With Applications, 182:115049, 2021.

[9] Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*, 2020.

[10] Priyam Basu, Tiasa Singha Roy, Rakshit Naidu, Zumrut Muftuoglu, Sahib Singh, and Fatemehsadat Mireshghallah. Benchmarking differential privacy and federated learning for bert models. *arXiv preprint arXiv:2106.13973*, 2021.

[11] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[12] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018.

[13] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[15] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[16] Xiaoyuan Cheng, Yukun Hu, and Liz Varga. 5g network deployment and the associated energy consumption in the uk: A complex systems' exploration. *Technological Forecasting and Social Change*, 180:121672, 2022.

[17] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *ACM SIGOPS Operating Systems Review*, 53(1):14–25, 2019.

[18] Matthias Delange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Ales Leonardis, Greg Slabaugh, and Tinne Tuytelaars. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[19] Jieren Deng, Chenghong Wang, Xianrui Meng, Yijue Wang, Ji Li, Sheng Lin, Shuo Han, Fei Miao, Sanguthevar Rajasekaran, and Caiwen Ding. A secure and efficient federated learning framework for nlp. *arXiv preprint arXiv:2201.11934*, 2022.

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

[21] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[22] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208. JMLR Workshop and Conference Proceedings, 2010.

[23] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. Spottune: transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4805–4814, 2019.

[24] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 129–143, 2016.

[25] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. A survey on vision transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[26] Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. Transformer in transformer. *Advances in Neural Information Processing Systems*, 34, 2021.

[27] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a unified view of parameter-efficient transfer learning. In *ICML*, 2022.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[29] Iris Hendrickx, Su Nam Kim, Zornitsa Kozareva, Preslav Nakov, Diarmuid O Séaghdha, Sebastian Padó, Marco Pennacchiotti, Lorenza Romano, and Stan Szpakowicz. Semeval-2010 task 8: Multi-way classification of semantic relations between pairs of nominals. *arXiv preprint arXiv:1911.10422*, 2019.

[30] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in neural information processing systems*, 26, 2013.

[31] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. Dynabert: Dynamic bert with adaptive width and depth. *Advances in Neural Information Processing Systems*, 33:9782–9793, 2020.

[32] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[33] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[34] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[35] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.

[36] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[37] Mayara Khadhraoui, Hatem Bellaaj, Mehdi Ben Ammar, Habib Hamam, and Mohamed Jmaiel. Survey of bert-base models for scientific text classification: Covid-19 case study. *Applied Sciences*, 12(6):2891, 2022.

[38] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

[39] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[40] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. Oort: Informed participant selection for scalable federated learning. *arXiv preprint arXiv:2010.06081*, 2020.

[41] Ken Lang. Newsweeder: Learning to filter netnews. In *Machine Learning Proceedings 1995*, pages 331–339. Elsevier, 1995.

[42] Younghoo Lee, Joshua Saxe, and Richard Harang. Catbert: Context-aware tiny bert for detecting social engineering emails. *arXiv preprint arXiv:2010.03484*, 2020.

[43] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. Hermes: an efficient federated learning framework for heterogeneous mobile clients. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 420–437, 2021.

[44] Anran Li, Lan Zhang, Juntao Tan, Yaxuan Qin, Junhao Wang, and Xiang-Yang Li. Sample-level data selection for federated learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[45] Chenning Li, Xiao Zeng, Mi Zhang, and Zhichao Cao. Pyramidfl: A fine-grained client selection framework for efficient federated learning.

[46] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[47] Bill Yuchen Lin, Chaoyang He, Zihang Zeng, Hulin Wang, Yufen Huang, Mahdi Soltanolkotabi, Xiang Ren, and Salman Avestimehr. Fednlp: A research platform for federated learning in natural language processing. *arXiv preprint arXiv:2104.08815*, 2021.

[48] Weijie Liu, Peng Zhou, Zhe Zhao, Zhiruo Wang, Haotang Deng, and Qi Ju. Fastbert: a self-distilling bert with adaptive inference time. *arXiv preprint arXiv:2004.02178*, 2020.

[49] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of the IEEE international conference on computer vision*, pages 3730–3738, 2015.

[50] Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. Compacter: Efficient low-rank hypercomplex adapter layers. *arXiv preprint arXiv:2106.04647*, 2021.

[51] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[52] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE international conference on communications (ICC)*, pages 1–7. IEEE, 2019.

[53] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[54] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning. In *EACL*, 2021.

[55] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. Adapterhub: A framework for adapting transformers. *arXiv preprint arXiv:2007.07779*, 2020.

[56] Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. Mad-x: An adapter-based framework for multi-task cross-lingual transfer. *arXiv preprint arXiv:2005.00052*, 2020.

[57] Sameer Pradhan, Alessandro Moschitti, Nianwen Xue, Hwee Tou Ng, Anders Björkelund, Olga Uryupina, Yuchen Zhang, and Zhi Zhong. Towards robust linguistic analysis using ontonotes. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 143–152, 2013.

[58] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. 2016.

[59] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. Learning multiple visual domains with residual adapters. *Advances in neural information processing systems*, 30, 2017.

[60] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295*, 2020.

[61] Amirhossein Reisizadeh, Isidoros Tziotis, Hamed Hassani, Aryan Mokhtari, and Ramtin Pedarsani. Straggler-resilient federated learning: Leveraging the interplay between statistical accuracy and system heterogeneity. *arXiv preprint arXiv:2012.14453*, 2020.

[62] Youngmin Ro and Jin Young Choi. Autolr: Layer-wise pruning and auto-tuning of learning rates in fine-tuning of deep networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 2486–2494, 2021.

[63] Andreas Rücklé, Gregor Geigle, Max Glockner, Tilman Beck, Jonas Pfeiffer, Nils Reimers, and Iryna Gurevych. Adapterdrop: On the efficiency of adapters in transformers. *arXiv preprint arXiv:2010.11918*, 2020.

[64] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[65] Taihua Shao, Yupu Guo, Honghui Chen, and Zepeng Hao. Transformer-based neural network for answer selection in question answering. *IEEE Access*, 7:26146–26156, 2019.

[66] Ivonne Soldevilla and Nahum Flores. Natural language processing through bert for identifying gender-based violence messages on social media. In *2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE)*, pages 204–208. IEEE, 2021.

[67] Joel Stremmel and Arjun Singh. Pretraining federated text models for next word prediction. In *Future of Information and Communication Conference*, pages 477–488. Springer, 2021.

[68] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune bert for text classification? In *China national conference on Chinese computational linguistics*, pages 194–206. Springer, 2019.

[69] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984*, 2020.

[70] Yi-Lin Sung, Varun Nair, and Colin A Raffel. Training neural networks with fixed sparse masks. *Advances in Neural Information Processing Systems*, 34, 2021.

[71] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[72] Betty Van Aken, Benjamin Winter, Alexander Löser, and Felix A Gers. How does bert answer questions? a layer-wise analysis of transformer representations. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1823–1832, 2019.

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[74] Paul Voigt and Axel Von dem Bussche. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 10(3152676):10–5555, 2017.

[75] Haozhao Wang, Zhihao Qu, Song Guo, Xin Gao, Ruixuan Li, and Baoliu Ye. Intermittent pulling with local compensation for communication-efficient distributed learning. *IEEE Transactions on Emerging Topics in Computing*, 2020.

[76] Su Wang, Mengyuan Lee, Seyyedali Hosseinalipour, Roberto Morabito, Mung Chiang, and Christopher G Brinton. Device sampling for heterogeneous federated learning: Theory, algorithms, and implementation. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[77] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. *Advances in Neural Information Processing Systems*, 31, 2018.

[78] Chen Wei, Chuang Niu, Yiping Tang, Yue Wang, Haihong Hu, and Jimin Liang. Npenas: Neural predictor guided evolution for neural architecture search. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[80] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. Error compensated quantized sgd and its applications to large-scale distributed optimization. In *International Conference on Machine Learning*, pages 5325–5333. PMLR, 2018.

[81] Jie Xu and Heqiang Wang. Client selection and bandwidth allocation in wireless federated learning networks: A long-term perspective. *IEEE Transactions on Wireless Communications*, 20(2):1188–1200, 2020.

[82] Mengwei Xu, Zhe Fu, Xiao Ma, Li Zhang, Yanan Li, Feng Qian, Shangguang Wang, Ke Li, Jingyu Yang, and Xuanzhe Liu. From cloud to edge: a first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 37–53, 2021.

[83] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. Deeptype: On-device deep learning for input personalization service with minimal privacy concern. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(4):1–26, 2018.

[84] Chengxu Yang, Qipeng Wang, Mengwei Xu, Zhenpeng Chen, Kaigui Bian, Yunxin Liu, and Xuanzhe Liu. Characterizing impacts of heterogeneity in federated learning upon large-scale smartphone data. In *Proceedings of the Web Conference 2021*, pages 935–946, 2021.

[85] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE, 2019.

[86] Lei Zhang, Shuai Wang, and Bing Liu. Deep learning for sentiment analysis: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4):e1253, 2018.

[87] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. *Advances in neural information processing systems*, 28, 2015.

[88] Yuxi Zhao and Xiaowen Gong. Quality-aware distributed computation and user selection for cost-effective federated learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6. IEEE, 2021.