ADELT: Transpilation between Deep Learning Frameworks

Linyuan Gong, **Jiayi Wang**, **Alvin Cheung**
University of California, Berkeley
{gly, jiayi_wang, akcheung}@berkeley.edu

Abstract

We propose the **Ad**versarial **DE**ep **L**earning **T**ranspiler (**ADELT**), a novel approach to source-to-source transpilation between deep learning frameworks. ADELT uniquely decouples code skeleton transpilation and API keyword mapping. For code skeleton transpilation, it uses few-shot prompting on large language models (LLMs), while for API keyword mapping, it uses contextual embeddings from a code-specific BERT. These embeddings are trained in a domain-adversarial setup to generate a keyword translation dictionary. ADELT is trained on an unlabeled web-crawled deep learning corpus, without relying on any hand-crafted rules or parallel data. It outperforms state-of-the-art transpilers, improving pass@1 rate by 16.2 pts and 15.0 pts for PyTorch-Keras and PyTorch-MXNet transpilation pairs respectively. We provide open access to our code at https://github.com/gonglinyuan/adelt.

1 Introduction

The rapid development of deep learning (DL) has led to an equally fast emergence of new software frameworks for training neural networks. Unfortunately, maintaining a deep learning framework and keeping it up-to-date is not an easy task. Many deep learning frameworks are deprecated or lose popularity every year, and porting deep learning code from a legacy framework to a new one is a tedious and error-prone task. A *source-to-source transpiler between DL frameworks* would greatly help practitioners overcome this difficulty.

Two promising solutions to source-to-source transpilation between deep learning frameworks are unsupervised neural machine translation (NMT) [Artetxe *et al.*, 2018] and large language models (LLMs). NMT treats deep learning code as a sentence for training sequence-to-sequence [Sutskever *et al.*, 2014] models, but its applicability is limited due to the scarcity of parallel corpora and its notable data hunger. On the other hand, LLMs like GPT-3 [Brown *et al.*, 2020], pretrained on web crawl data, offer potential, performing translation tasks in a few-shot or zero-shot manner. Our early experiments with GPT-4 show its potential in few-shot transpilation of deep learning programs. However, such models struggle with API-specific details, inaccurately handling function names and parameter mappings.

That said, most deep learning framework code is *structured*: each type of layers has its own constructor, and constructing a network involves calling each layer's constructor in a chaining manner. By leveraging the structures of programming languages, we can *decouple* the transpilation of skeletal codes from the mapping of API keywords. The transpilation of skeletal codes is the easier part, and LLMs already do a great job. We only need a separate algorithm to translate the *API keywords*, i.e., the function and parameter names to complete the transpilation.

In this paper, we present ADELT (Figure 1), a method motivated by this insight to transpile DL code. The canonicalized source code is decoupled into two parts: the code skeleton and the API keywords. ADELT transpiles the code skeleton using a pretrained LLM by few-shot prompting. Each API keyword occurrence is then embedded into a vector by PyBERT, a BERT pretrained on Python code. This vector is both the textual and the contextual representation of the API keyword. ADELT then leverages domain-adversarial training to learn a generator that maps the vector to an aligned embedding space. The alignment is enforced by a two-player game, where a discriminator is trained to distinguish between the embeddings from the source DL framework and those from the target DL framework. The API keyword embeddings are trained jointly with the generator as the output embedding matrix of a softmax classifier on the aligned embedding space. After generating a synthetic API keyword dictionary from the embeddings using a two-step greedy algorithm, ADELT then looks up each API keyword occurrence in the dictionary and puts them back into the transpiled code skeleton.

In summary, this paper makes the following contributions:

- We introduce ADELT, a robust solution for transpilation between deep learning frameworks without training on any labeled data. Outperforming large language models, ADELT excels across various transpilation pairs, achieving pass@1 rate of 73.0 and 70.0 for PyTorch-Keras and PyTorch-MXNet transpilations, respectively. These scores surpass those of the state-of-the-art LLM, GPT-4, by 16.2 and 15.0 points respectively.
- For training, we construct a PyTorch-Keras-MXNet corpus

*The appendices can be found at https://arxiv.org/abs/2303.03593.

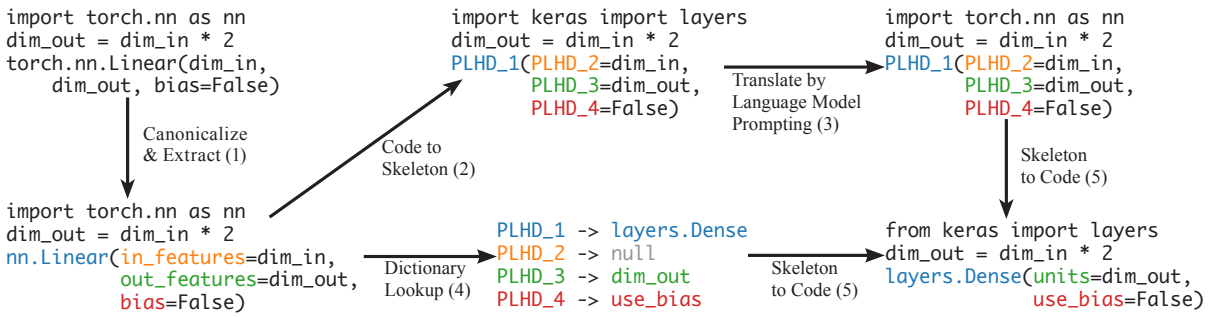

Figure 1: **An example of ADELT's pipeline:** an import statement in the code skeleton is transpiled from PyTorch to Keras by a language model via few-shot prompting; a linear fully-connected layer is transpiled by removing the argument `in_features` and renaming other API keywords according to the learned dictionary. The number (1 to 5) near each arrow label corresponds to the step number in Section 2.

of deep learning code from various Internet sources, containing 49,705 PyTorch modules, 11,443 Keras layers/models, and 4,785 MXNet layers/models. We then build an evaluation benchmark for PyTorch-Keras and PyTorch-MXNet transpilation. The benchmark evaluates both our API keyword mapping algorithm and the overall source-to-source transpilation.

2 Method

ADELT (**A**dversarial **DE**ep **L**earning **T**ranspiler) is an algorithm that transpiles code from a source deep learning framework into an equivalent one in a target framework, by transpiling the skeletal code using a pretrained large language model, and then looking up each keyword in a dictionary learned with unsupervised domain-adversarial training. ADELT applies the following steps to each piece of input code, which we illustrate using the example shown in Figure 1:

1. Extract *API calls* from the source code. Such API calls can be automatically extracted with the Python's built-in `ast` library. We then convert each API call into its canonical form, where each layer/function has a unique name, and all of its arguments are converted to keyword arguments. Finally, we extract all *API keywords* from the canonicalized API call, where an *API keyword* is the name of a layer/function or the name of a keyword argument.
2. Transform the program into its *code skeleton* by replacing each API keyword occurrence with a distinct placeholder.
3. Transpile the code skeleton, where all API keywords are replaced by placeholders, into the target DL framework using a pretrained big LM (e.g., Codex).
4. Look up each API keyword in the *API keyword dictionary*, and replace each keyword with its translation. To generate the API keyword dictionary, we first learn the API embeddings using domain-adversarial training based on contextual embeddings extracted by PyBERT (a BERT pretrained on Python code and then fine-tuned on deep learning code). Next, we calculate the cosine similarity between the embedding vectors. Then we generate the API keyword dictionary using a hierarchical algorithm.
5. Put each API keyword back into the transpiled code skeleton to generate the final output.

We describe each of these steps next in detail.

2.1 Canonicalization & API Keyword Extraction

We first parse the source code into an *abstract syntax tree (AST)* with the Python `ast` module. Then, canonicalization and API call extraction are applied to the AST.

Canonicalization. We canonicalize each API call using the following steps during both domain-adversarial training (Section 2.3) and inference. Each step involves a recursive AST traversal.

1. Unify the different import aliases of each module into the most commonly used name in the training dataset. For example, `torch.nn` is converted to `nn`.
2. Unify different aliases of each layer/function in a DL library into the name in which it was defined. We detect and resolve each alias by looking at its `__name__` attribute, which stores the callable's original name in its definition.[1] For example, `layers.MaxPool2D` is converted to `layers.MaxPooling2D`.
3. Convert each positional argument of an API call into its equivalent keyword argument. Sort all keyword arguments according to the order defined in the function signature. This is done by linking the arguments of each API call to the parameters of its API signature using the `bind` method from Python's `inspect` module.[2]

API keyword extraction. We define *API keyword* as the name of a layer/function or the name of a keyword argument. Once the input code is canonicalized, we locate each API keyword in the AST and then unparse the AST into the canonicalized source code.

2.2 Skeletal Code Transpilation

After canonicalizing the source program, ADELT then replaces all API keywords with a placeholder, turning the source program into its *code skeleton*. Each placeholder has textual form `PLACEHOLDER_i`, where $i = 1, 2, \ldots$. The code skeleton is then translated by Codex using few-shot prompting. The full prompt for is shown in Appendix A.4 [Gong *et al.*, 2024].

[1] https://docs.python.org/3/reference/datamodel.html

[2] https://docs.python.org/3/library/inspect.html#inspect.Signature.bind

Algorithm 1 Pseudo-code for domain-adversarial training.

```
1 for (x_1, y_1), (x_2, y_2) in loader:
2   # N samples from X_1, X_2 respectively
3   # y_1, y_2: API keyword ids
4
5   h_1 = B(x_1).detach()  # contextual embedding
6   h_2 = B(x_2).detach()  # no gradient to PyBERT
7   z_1 = G(h_1)  # generator hidden states
8   z_2 = G(h_2)  # z_1, z_2: N x d
9
10  # dot product of z_l and output embeddings
11  logits_1 = mm(z_1, E_1.view(d, m_1))
12  logits_2 = mm(z_2, E_2.view(d, m_2))
13  L_CE_1 = CrossEntropyLoss(logits_1, y_1)
14  L_CE_2 = CrossEntropyLoss(logits_2, y_2)
15
16  # discriminator predictions
17  pred_1 = D(z_1)
18  pred_2 = D(z_2)
19  labels = cat(zeros(N), ones(N))
20  L_D = CrossEntropyLoss(pred_1, labels)
21  L_G = CrossEntropyLoss(pred_2, 1 - labels)
22
23  # joint update of G and E_l
24  #   to minimize L_CE_l
25  optimize(G + E_1 + E_2, L_CE_1 + L_CE_2)
26  optimize(D, L_D)  # train the discriminator
27  optimize(G, L_G)  # train the generator
```

`B`: PyBERT used as the contextual embedder.
`G`, `D`: the generator $\mathcal{G}$ and the discriminator $\mathcal{D}$.
`E_l`: a d by m_l matrix, where the i-th column vector is the output embedding of API keyword $w_i^{(l)}$.
`mm`: matrix multiplication; `cat`: concatenation.

2.3 Domain-Adversarial Training

Once the code skeleton is transpiled, we then transpile API keywords. We train the aligned embeddings of API keywords in a domain-adversarial setting. In Section 2.4, the embeddings will be used to generate a dictionary that maps an API keyword of the source deep learning framework $\mathcal{X}^{(1)}$ to an API keyword in the target DL framework $\mathcal{X}^{(2)}$.

Figure 2 illustrates the domain-adversarial approach of ADELT, and Algorithm 1 shows the pseudocode. A generator maps the contextual representations extracted by PyBERT into hidden states (line 5-8). The alignment of hidden states from different DL frameworks is enforced by the adversarial loss induced by the discriminator (line 17-21), so that output embeddings learned with these hidden states (line 11-14) are also aligned. Next, we describe each step in detail:

Each training example is a pair of API keyword occurrences with their context in the training corpus, denoted by $(x^{(1)}, x^{(2)})$. Each keyword occurrence $x^{(l)}$ is tokenized and encoded as multiple *byte pair encoding (BPE)* [Sennrich *et al.*, 2016] tokens. In our unsupervised setting, $x^{(1)}$ and $x^{(2)}$ are independent samples from $\mathcal{X}^{(1)}$ and $\mathcal{X}^{(2)}$ in the training dataset, respectively, and they are not necessarily translations of each other.

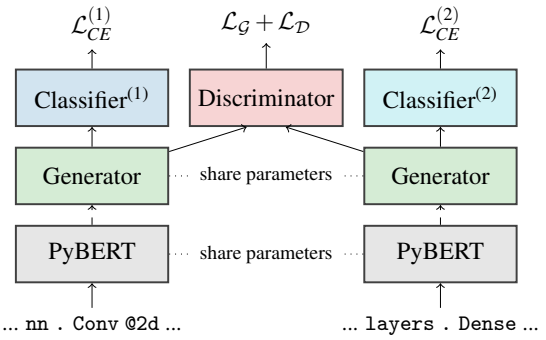

Figure 2: ADELT's domain-adversarial training with contextual embeddings from a PyBERT. The generator and the PyBERT are shared between different DL frameworks. We do not fine-tune the PyBERT during adversarial training.

$$\begin{aligned}\mathcal{L}_{\mathcal{D}} = &- \mathbb{E}_{\text{data}}[\log \Pr_{\mathcal{D}}(\text{pred} = 1|\mathcal{G}(\mathbf{h}^{(1)}))] \\ &- \mathbb{E}_{\text{data}}[\log \Pr_{\mathcal{D}}(\text{pred} = 2|\mathcal{G}(\mathbf{h}^{(2)}))] \\ \mathcal{L}_{\mathcal{G}} = &- \mathbb{E}_{\text{data}}[\log \Pr_{\mathcal{D}}(\text{pred} = 2|\mathcal{G}(\mathbf{h}^{(1)}))] \\ &- \mathbb{E}_{\text{data}}[\log \Pr_{\mathcal{D}}(\text{pred} = 1|\mathcal{G}(\mathbf{h}^{(2)}))]\end{aligned} \tag{1}$$

$$\mathcal{L}_{\text{CE}}^{(l)} = -\mathbb{E}_{(x,y)\sim\text{data}^{(l)}}\left[\log \frac{\exp(\mathbf{z}\cdot\mathbf{e}_y^{(l)})}{\sum_{k=1}^{m^{(l)}}\exp(\mathbf{z}\cdot\mathbf{e}_k^{(l)})}\right] \tag{2}$$

PyBERT. *PyBERT* is our pretrained Transformer [Vaswani *et al.*, 2017; Devlin *et al.*, 2019] for Python code [Feng *et al.*, 2020; Kanade *et al.*, 2020; Roziere *et al.*, 2021]. Given a sequence of BPE tokens that represent an API keyword with its context $x^{(l)}$, PyBERT outputs a sequence of vectors—one vector in $\mathbb{R}^{d_b}$ for each token, where d_b is the hidden dimension size of PyBERT. We average-pool all BPE tokens of the keyword and get a single d_b-dimensional vector as the contextual embedding $\text{PyBERT}(x^{(l)})$ of the API keyword. We denote the contextual embedding of $x^{(1)}, x^{(2)}$ by $\mathbf{h}^{(1)}, \mathbf{h}^{(2)}$ respectively.

Generator and discriminator. We define two multi-layer perceptrons, a generator and a discriminator. A generator $\mathcal{G}$ encodes the contextual embeddings $\mathbf{h}^{(1)}, \mathbf{h}^{(2)}$ into hidden states $\mathbf{z}^{(1)}, \mathbf{z}^{(2)} \in \mathbb{R}^d$, and a discriminator $\mathcal{D}$ is trained to discriminate between $\mathbf{z}^{(1)}$ and $\mathbf{z}^{(2)}$. The generator is trained to prevent the discriminator from making accurate predictions, by making $\mathcal{G}(\text{PyBERT}(\mathcal{X}^{(1)}))$ and $\mathcal{G}(\text{PyBERT}(\mathcal{X}^{(2)}))$ as similar as possible. Our approach is inspired by domain-adversarial training [Ganin *et al.*, 2016], where domain-agnostic representations of images or documents are learned for domain adaptation. In our case, a domain is represented by a DL framework.

Formally, we define the probability $\Pr_{\mathcal{D}}(\text{pred} = l|\mathbf{z})$ that a hidden state $\mathbf{z}$ is from the DL framework l predicted by the discriminator. Note that $\mathbf{z}^{(1)} = \mathcal{G}(\mathbf{h}^{(1)})$ and $\mathbf{z}^{(2)} = \mathcal{G}(\mathbf{h}^{(2)})$. The discriminator loss and the generator loss are computed as the binary cross entropy against the true label and the reversed label, respectively, as shown in Equation (1).

Output embeddings. Our goal is to learn an embedding for each API keyword, but the contextual embedding of each keyword occurrence varies with its context. So we instead train a d-dimensional vector $\mathbf{e}_i^{(l)}$ for each API keyword $w_i^{(l)}$, such that $\mathbf{e}_i^{(l)}$ is similar to the generator hidden states $\mathbf{z}_j^{(l)}$ of this keyword's occurrences and dissimilar to the hidden states $\mathbf{z}_k^{(l)}$ of any other keyword's occurrences. $\mathbf{e}_i^{(l)}$ is considered the *output embedding* of the API keyword $w_i^{(l)}$. With similarity computed using dot product, our optimization objective is shown in Equation (2), equivalent to the cross-entropy loss of $m^{(l)}$-way softmax-based classification.

Adversarial training. During each training iteration, the generator and discriminator are trained successively to minimize $\mathcal{L}_\mathcal{G}$ and $\mathcal{L}_\mathcal{D}$ respectively with mini-batch stochastic gradient descent. Minimizing the adversarial loss equals to minimizing the distance between two distributions of hidden states [Goodfellow *et al.*, 2014]. Therefore, the API keywords from the different DL frameworks will be mapped to an aligned embedding space.

Also, we jointly update the generator and the output embeddings to minimize $\mathcal{L}_{\text{CE}}^{(l)}$ with mini-batch SGD. The joint optimization is crucial, as updating the generator to minimize $\mathcal{L}_{\text{CE}}^{(l)}$ ensures that each generator hidden state $\mathbf{z}^{(l)}$ preserves enough information to recover its original API keyword. As a result, the output embeddings $\{\mathbf{e}_i^{(1)}\}_{i=1}^{m^{(1)}}$ and $\{\mathbf{e}_j^{(2)}\}_{j=1}^{m^{(2)}}$ are also aligned, as they are trained with vectors $\mathbf{z}^{(l)}$ from the aligned embedding space.

We do not fine-tune PyBERT during domain-adversarial training, as fine-tuning PyBERT makes the generator disproportionally strong that results in training divergence.

2.4 Hierarchical API Dictionary Generation

ADELT calculates a *scoring matrix* using the aligned API keyword embeddings trained in Section 2.3. The entry in the i-th row and the j-th column of the matrix is the cosine similarity between $w_i^{(1)}$ and $w_j^{(2)}$, denoted by $s_{i,j}$. Given the scoring matrix, we need to generate an API keyword dictionary that maps each API keyword in one deep learning framework to an API keyword in another DL framework.

Greedy match. Greedy match has been used to generate a dictionary in word translation of natural languages [Conneau *et al.*, 2018], where each source word is matched to the target word with the highest similarity score.

Structure of API keywords. Unlike words in NL, API keywords are *structured*: API keywords can be classified into two types based on their associated AST node: *callables names* (names of functions or classes), and *parameter names* (names of keyword arguments). In dictionary generation, we do not allow callable names to be translated to parameter names. We only allow parameter names to be translated to callable names in a special case when the weight passes a threshold. In this case, this parameter will be dropped and generate a new API call (the last case in Table 2). Another structural property is that the matching of parameters depends on the matching of callables.

Hierarchical API dictionary generation. The algorithm leverages the structure of API keywords to generate a dictionary: **Step 1.** Consider each callable and its parameters as a group and compute the *group similarity* between each pair of groups, by summing up similarity scores in the greedy matching of parameter names, plus the similarity between two callable names. **Step 2.** Match groups greedily based on group similarity scores calculated in step 1.

3 Experiments

We evaluate the effectiveness of ADELT on the task of transpilation between PyTorch, Keras, and MXNet and compare our method with baselines.

3.1 Skeletal Code Transpilation

We use Codex [Chen *et al.*, 2021], a LLM trained on public GitHub code, to transpile code skeletons. As an autoregressive language model trained on massive web data, Codex can handle translation tasks via prompting with few-shot demonstrations. Our prompt design aligns with Codex's code translation setup, comprising a single input-output example and three instructions to keep placeholders unchanged. Appendix A.4 provides further details on this.

3.2 Training Setup

DL corpus. We consider 3 data sources **GitHub**, **JuiCe**, **Kaggle** to build our DL corpus:

- **GitHub**: The GitHub public dataset available on Google BigQuery.[3] We keep `py` and `ipynb` files that contain `torch`, `keras`, or `mxnet` in the `main` and `master` branch of the repository. (69GB of clean Python code before filtering, 2.5GB after filtering).
- **JuiCe**: A code generation dataset [Agashe *et al.*, 2019] based on `ipynb` files from GitHub. JuiCe contains many files absent in the public dataset on Google BigQuery, since the latter is a selected subset of GitHub (10.7GB of clean Python code).
- **Kaggle**: All files in KGTorrent [Quaranta *et al.*, 2021], a dataset of Jupyter Notebooks from Kaggle[4] (22.1GB of clean Python code).

We tokenize all Python source code and extract subclasses of `torch.nn.Module`, `keras.layers.Layer`, or `keras.Model`. Then, we canonicalize (section 2.1) the code of each class definition. We byte-pair encode [Sennrich *et al.*, 2016], merge, and deduplicate codes from all sources. Finally, we collect all files into our *DL Corpus* containing 49,705 PyTorch modules, 11,443 Keras layers/models, and 4,785 MXNet layers/models.

PyBERT. PyBERT is our Transformer encoder pretrained with masked language modeling (MLM) [Devlin *et al.*, 2019] on all open-source Python files from the GitHub dataset. We consider two model sizes: $\text{PyBERT}_{\text{SMALL}}$ (6-layer, 512-d) and $\text{PyBERT}_{\text{BASE}}$ (12-layer, 768-d). Pretraining hyperparameters are detailed in Appendix A.1 [Gong *et al.*, 2024].

[3] https://console.cloud.google.com/marketplace/details/github/github-repos

[4] https://kaggle.com

	PyTorch-Keras						PyTorch-MXNet					
	F1		EM		Pass@1		F1		EM		Pass@1	
GPT-3 [Brown *et al.*, 2020]	26.6	32.0	22.4	26.0	23.4	27.2	25.8	32.8	23.4	25.0	25.0	26.4
Codex [Chen *et al.*, 2021]	59.9	67.1	51.5	54.6	53.4	57.6	57.4	69.0	53.2	56.2	54.2	57.6
GPT-4	67.7	74.9	55.6	64.6	56.8	66.0	60.3	71.8	54.0	60.2	55.0	60.8
Edit Distance (Cased)	31.2	30.1	20.3	16.8	20.3	16.8	37.7	35.7	22.8	21.0	22.8	21.0
Edit Distance (Uncased)	23.9	30.1	12.6	16.8	12.6	16.8	30.8	36.0	18.4	20.0	18.4	20.0
ADELT (Small)	79.0	76.7	70.8	67.6	70.8	67.6	76.7	70.6	66.6	63.0	66.6	63.0
ADELT (Base)	**83.4**	**79.3**	**73.0**	**71.6**	**73.0**	**71.6**	**80.0**	**72.1**	**70.0**	**63.8**	**70.0**	**63.8**

Table 1: **Comparison between ADELT and other methods** on source-to-source transpilation. "ADELT (Small)" is ADELT with $\text{PyBERT}_{\text{SMALL}}$ and "ADELT (Base)" is ADELT with $\text{PyBERT}_{\text{BASE}}$. There are two numbers in each table cell: the first one is for transpiling PyTorch to the other framework (Keras or MXNet), and the second one is for transpiling the other framework to PyTorch. Each number is the average of 5 runs with different random seeds.

Adversarial training. The generator and discriminator of ADELT are multilayer perceptrons. We search the learning rate and batch size according to the unsupervised validation criterion *"average cosine similarity"* [Conneau *et al.*, 2018], which measures the consistency between learned API keyword embeddings and generated keyword translations. Other hyperparameters are set based on previous studies [Conneau *et al.*, 2018] with details described in Appendix A.2.

3.3 Evaluation Benchmark

Our method is evaluated through the task of transpiling code snippets from one DL framework to another. Our benchmark consists of two parts: first, we use heuristics to identify potential matching pairs in the corpus, which were then refined through manual curation to ensure a solid evaluation benchmark; the second part of the benchmark includes a set of expert-transpiled examples, each accompanied by unit tests. For detailed methodology and statistics, please refer to Appendix A.3.

We report results in three evaluation metrics:

- **F1 score** quantifies the overlap between the predicted and ground truth outputs. In this context, we treat each prediction or ground truth as a bag of function calls. For each test case, we determine the number of exactly matched calls n_{match}, predicted calls n_{pred}, and ground truth calls n_{truth}. We define the F1 score for a particular example as $2n_{\text{match}}/(n_{\text{pred}} + n_{\text{truth}})$, and report the average F1 scores across all test cases.
- **Exact Match (EM) score** is a more rigorous metric that evaluates whether a model's transpilation is exactly equivalent to the ground truth for each code snippet. It's calculated as the proportion of exact matches to the total number of examples in the eval set.
- **Pass@1** assesses the proportion of examples for which the first transpilation attempt by the model successfully passes all the unit tests. These unit tests, created by experts for each benchmark example, evaluate the correctness of transpilations by execution.

3.4 Evaluation of Skeletal Code Transpilation

Transpiling code skeletons of DL programs is an easy task, and Codex easily learned transpilation patterns via few-shot prompting. In our evaluation benchmark, the exact match score of skeletal code transpilation using Codex is 100%.

3.5 Comparison with Other Methods

We compare ADELT using $\text{PyBERT}_{\text{SMALL}}$ and ADELT using $\text{PyBERT}_{\text{BASE}}$ with the following baselines. We run all methods 5 times with random seeds `[10, 20, 30, 40, 50]`, and report the arithmetic average of all metrics.

End-to-end language models. We compare ADELT with end-to-end few-shot LLM baselines, including GPT-3, Codex, and GPT-4, where the entire piece of source code, instead of the code skeleton, is fed into the LLM to generate the transpiled target program. For source-to-source translation, we use the *"completion"* endpoint of `code-davinci-002` version for Codex and the *"chat"* endpoint of `gpt-4-0314` for GPT-4. In both cases, we give the LLM a natural language instruction and 5 examples as demonstrations. Details of the prompts are shown in Appendix A.5 [Gong *et al.*, 2024].

Edit distance. We consider a rule-based baseline where we use edit distance [Levenshtein, 1966] as the similarity measure between API keywords, in place of the similarity measures calculated from learned embeddings. We apply hierarchical API dictionary generation exactly as what we do in ADELT. We report the result of both cased and uncased setups for edit distance calculation.

Results. Table 1 shows that **ADELT consistently outperforms other methods across all metrics**. Notably, ADELT outperforms GPT-4 by significant margins, achieving a 16.2 pts lead in pass@1 of PyTorch-Keras translations and a 5.6 pts lead in Keras-PyTorch. The difference is due to ADELT being based on an encoder-only PyBERT model that leverages larger corpora of the *source* DL framework (PyTorch) more effectively, unlike GPT-4, a decoder-only LLM that benefits from larger corpora for the *target* framework. Therefore, ADELT complements traditional end-to-end LLM methods. Additionally, ADELT runs much faster than GPT-4, as it uses a smaller LLM for transpiling code skeletons and a dictionary lookup step that requires only a tiny fraction of the time needed for full LLM inference.

3.6 Case Studies

Table 2 shows four examples of PyTorch-Keras transpilation together with hypotheses of Codex and ADELT (Base).

Source	`nn.Conv2d(64, 128, 3)`	Source	`nn.Embedding(vocab_size, embed_dim)`
Truth	`layers.Conv2D(filters=128, kernel_size=3)`	Truth	`layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)`
Codex ✓	`layers.Conv2D(128, 3)`	Codex ✓	`layers.Embedding(vocab_size, embed_dim)` `self.position_emb = layers.Embedding(...)`
ADELT ✓	`layers.Conv2D(filters=128, kernel_size=3)`	ADELT ✗	`layers.Embedding(embeddings_initializer=embed_dim)`

Source	`nn.MultiheadAttention(model_dim, num_heads=num_heads, dropout=attn_dropout)`	Source	`in_dim = 256`
			`out_dim = 512`
			`layers.Dense(out_dim, activation='relu')`

Truth	`layers.MultiHeadAttention(num_heads=num_heads, key_dim=model_dim, dropout=attn_dropout)`	Truth	`in_dim = 256`
			`out_dim = 512`
			`nn.Linear(in_dim, out_dim)`
			`nn.ReLU()`

Codex ✗	`layers.MultiHeadAttention(model_dim, num_heads, dropout=attn_dropout)`	Codex ✗	`in_dim = 256`
			`out_dim = 512`
			`nn.Linear(in_dim, out_dim)`

ADELT ✓	`layers.MultiHeadAttention(num_heads=num_heads, key_dim=model_dim, dropout=attn_dropout)`	ADELT ✗	`in_dim = 256`
			`out_dim = 512`
			`nn.Linear(in_features=in_dim, out_features=out_dim)`

		ADELT + ✓	`in_dim = 256`
			`out_dim = 512`
			`nn.Linear(in_features=in_dim, out_features=out_dim)`
			`nn.ReLU()`

Table 2: **Examples from the evaluation dataset of the PyTorch-Keras transpilation task and the Keras-PyTorch transpilation task.** We show the source code, ground truth target code, and the outputs from Codex, ADELT, and ADELT +. ✓: the output is the same or equivalent to the ground truth. ✓: the output contains an equivalent of the ground truth, but it also contains incorrect extra code. ✗: the output is incorrect.

Both Codex and ADELT transpile the `nn.Conv2d` to Keras correctly by dropping the first argument `in_channels`. ADELT does not translate the parameter names of `nn.Embedding` to `input_dim` and `output_dim` correctly, while Codex does. However, we notice that Codex sometimes relies on the argument ordering heuristic. In the example of `nn.MultiheadAttention`, where parameters have a different ordering in Keras than in PyTorch, Codex generates the wrong translation, but ADELT successfully constructs the correct mapping between parameters.

Also, in the `nn.Embedding` example, Codex continues to generate code about "positional embeddings" after finishing transpilation. The extra code generated by Codex is relevant to the context.[5] Still, the extra code should not be part of the translation. We have tried various ways to make Codex follow our instructions (see Appendix A.5 [Gong *et al.*, 2024]). However, because Codex is an end-to-end neural language model, our means of changing its predictions are limited, and the result is highly indeterministic. In the end, Codex still occasionally generates extra arguments or unneeded statements.

On the other hand, we decouple neural network training from the transpilation algorithm. ADELT transpiles between deep learning frameworks using deterministic keyword substitution based on a learned API keyword dictionary. The transpiled code is always syntactically correct. If a mistake is found in the dictionary (e.g., the `nn.Embedding` example in Table 2), it can be corrected by simply modifying the dictionary.

Correcting the API keyword dictionary by humans requires much less effort than building the dictionary manually from scratch, as ADELT generates a high-quality dictionary. Developers can even add additional rules to the transpiler. The flexibility of our decoupled design makes ADELT far easier to be integrated into real-world products than end-to-end neural translators/LMs are.

The last case in Table 2 shows an example where an API call (`layers.Dense` with `activation="relu"`) should be transpiled to two calls (`nn.Linear` and `nn.ReLU`). One-to-many mapping is rare in transpilation between deep learning frameworks, but the capability to model such mapping reflects the generality of a transpiler to other APIs. Both ADELT and Codex fail to solve this example because this usage is rarely seen in the training data. Still, if we train ADELT on an additional synthetic dataset ("ADELT +" in Table 2. See Appendix A.8 [Gong *et al.*, 2024]), it successfully solves this case, showing that our method can model one-to-many mappings when enough training data is available.

3.7 Ablation Studies

We conduct ablation studies on PyTorch-Keras transpilation to validate the contribution of each part of ADELT. As illustrated

[5]The definition of positional embeddings usually follows the definition of word embeddings (`nn.Embedding(vocab_size, ...)`) in the source code of a Transformer model.

	Keyword				Source Code	
	P@1		MRR		F1	
ADELT (Small)	82.9	90.0	87.0	94.0	79.0	76.7
ADELT (Base)	**87.1**	90.0	**89.7**	94.0	**83.4**	**79.3**
Domain-adversarial training						
w/o PyBERT (Small)	52.1	63.6	60.5	72.8	37.2	43.0
w/o PyBERT (Base)	45.0	54.6	56.8	66.0	33.0	36.3
w/o Adv Loss (Small)	80.4	88.6	85.3	93.1	65.8	73.6
w/o Adv Loss (Base)	86.3	90.5	89.3	94.3	78.2	72.3
Measure for dictionary generation						
Inner Product (Small)	81.3	79.6	86.3	85.4	74.6	73.2
Inner Product (Base)	85.4	**93.2**	88.8	**95.7**	80.2	78.8

Table 3: **Ablation study results.** By default, ADELT is trained with the adversarial loss on contextual embeddings extracted by PyBERT, and then a dictionary is generated based on cosine similarity scores. We change one component of ADELT (Small) or ADELT (Base) in each experiment to assess its contribution.

in Figure 1, ADELT consists of five steps. Steps 1 (canonicalization and extraction), 2 (code to skeleton), and 5 (skeleton to code) are deterministic and always yield 100% accuracy as they do not rely on machine learning models. The accuracy of Step 3 (code skeleton transpilation) has been discussed in Section 3.4. Therefore, this section focus on the primary error source: the *API keyword translation* in Step 4. This crucial step involves mapping API keywords between frameworks. To evaluate its accuracy, we construct a high-quality dictionary by manually translating the top 50 most frequent API keywords from PyTorch to Keras. We then evaluate the translation's efficacy using standard metrics: *precision@*k (for $k = 1, 5$) and the *mean reciprocal rank* (MRR) of correct translations. The results are shown in Table 3, where we study the following variants of ADELT:

Necessity of contextual embeddings. In *"w/o PyBERT"*, we replace PyBERT with Word2Vec [Mikolov *et al.*, 2013] embeddings of the same dimensions d_b trained on the same corpora. The result in Table 3 shows that this change significantly harms the performance of ADELT. This justifies the use of PyBERT, a high-quality pretrained representation of API keywords that can capture their contexts.

Contribution of adversarial loss. In *"w/o Adv Loss"*, we remove the adversarial loss during training. Instead, we only train the generator and the output embeddings with the cross-entropy loss in Equation (2). The result in Table 3 shows that adversarial training contributes $\sim$6 pts in source-to-source transpilation, showing the effectiveness of adversarial training.

Comparison of similarity measures. By default, ADELT uses cosine similarity as the similarity measure for API dictionary generation. Table 3 shows the results of using dot product (inner). Cosine similarity outperforms dot product by a small margin. This fact implies that the performance of ADELT is insensitive to the choice of similarity measure.

4 Related Work

Source-to-source transpilation. Classical source-to-source transpilers use supervised learning. Nguyen *et al.* [2013] and Karaivanov *et al.* [2014] develop Java-C# transpilers using parallel corpora of open-source code. The dependency on parallel corpora renders these methods inapplicable to transpilation between deep learning frameworks, as parallel corpora are difficult to get.

Drawing inspiration from unsupervised neural machine translation (NMT) [Artetxe *et al.*, 2018], recent advancements have made unsupervised programming language translation possible [Lachaux *et al.*, 2020]. Such approaches, however, require vast amounts of in-domain unlabeled corpora, as evidenced by Lachaux *et al.* [2020] and Roziere *et al.* [2022], who used 744GB of GitHub source code and 333k curated Java functions respectively. The scarcity of DL code hinders their effectiveness for transpilation between DL frameworks.

Metalift [Bhatia *et al.*, 2023] is a transpiler generator for various domain-specific languages (DSLs) [Cheung *et al.*, 2013; Ahmad *et al.*, 2019; Ahmad *et al.*, 2022; Ahmad and Cheung, 2018; Qiu *et al.*, 2024]. It synthesizes target code validated through formal verification. Metalift requires users to explicitly define the target DSL's semantics, while ADELT automatically learns transpilation mappings from data.

Language models are few shot learners. GPT-3 [Brown *et al.*, 2020] is a language model with 175B parameters trained on massive web crawl data. GPT-3 can be applied to many NLP tasks without any task-specific training, with instructions and few-shot demonstrations specified purely via text interaction with the model. Codex [Chen *et al.*, 2021] is a GPT-3 fine-tuned on publicly available code from GitHub, specialized for code generation tasks. GPT-4 is a LLM proficient in both code and NL trained using instruction finetuning. In constrast, the code generation step of ADELT is keyword substitution instead of autoregressive generation. ADELT outperforms GPT-3, Codex, and GPT-4 in PyTorch-Keras transpilation and PyTorch-MXNet transpilation.

Adversarial learning & cross-lingual word embedding. Conneau *et al.* [2018] uses domain-adversarial [Ganin *et al.*, 2016] approach to align the distribution of two word embeddings, enabling natural language word translation without parallel data. The domain-adversarial training in ADELT is inspired by their approach, but we align the distributions of the hidden states of *keyword occurrences*.

5 Conclusion

In this work, we present ADELT, an novel code transpilation algorithm for deep learning frameworks. ADELT decouples code transpilation into two distinct phases: code skeleton transpilation and API keyword mapping. It leverages large language models (LLMs) for the transpilation of skeletal code, while using domain-adversarial training for the creation of an API keyword mapping dictionary. This strategic decoupling harnesses the strengths of two different model types. Through comprehensive evaluations using our specially curated PyTorch-Keras and PyTorch-MXNet benchmarks, we show that ADELT significantly surpasses existing state-of-the-art transpilers in performance.

Acknowledgements

We thank all reviewers for their feedback and support. This work is supported in part by gift from Meta, the U.S. National Science Foundation through grants IIS-1955488, IIS-2027575, ARO W911NF2110339, ONR N00014-21-1-2724, and DOE awards DE-SC0016260, DE-SC0021982.

References

[Agashe *et al.*, 2019] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *arXiv:1910.02216 [cs]*, Oct 2019. arXiv: 1910.02216.

[Ahmad and Cheung, 2018] Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1205–1220. ACM, 2018.

[Ahmad *et al.*, 2019] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Trans. Graph.*, 38(6):204:1–204:13, 2019.

[Ahmad *et al.*, 2022] Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector instruction selection for digital signal processors using program synthesis. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 1004–1016. ACM, 2022.

[Artetxe *et al.*, 2018] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised neural machine translation. *arXiv:1710.11041 [cs]*, Feb 2018. arXiv: 1710.11041.

[Bhatia *et al.*, 2023] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building code transpilers for domain-specific languages using program synthesis. pages 30 pages, 1247897 bytes, 2023.

[Brown *et al.*, 2020] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv:2005.14165 [cs]*, Jul 2020. arXiv: 2005.14165.

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *arXiv:2107.03374 [cs]*, Jul 2021. arXiv: 2107.03374.

[Cheung *et al.*, 2013] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14. ACM, 2013.

[Conneau *et al.*, 2018] Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. *arXiv:1710.04087 [cs]*, Jan 2018. arXiv: 1710.04087.

[Devlin *et al.*, 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805 [cs]*, May 2019. arXiv: 1810.04805.

[Feng *et al.*, 2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *arXiv:2002.08155 [cs]*, Sep 2020. arXiv: 2002.08155.

[Ganin *et al.*, 2016] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *arXiv:1505.07818 [cs, stat]*, May 2016. arXiv: 1505.07818.

[Gong *et al.*, 2024] Linyuan Gong, Jiayi Wang, and Alvin Cheung. ADELT: Transpilation between deep learning frameworks. (arXiv:2303.03593), April 2024. arXiv:2303.03593 [cs].

[Goodfellow *et al.*, 2014] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv:1406.2661 [cs, stat]*, Jun 2014. arXiv: 1406.2661.

[Kanade *et al.*, 2020] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. *arXiv:2001.00059 [cs]*, Aug 2020. arXiv: 2001.00059.

[Karaivanov *et al.*, 2014] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical trans-

lation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, page 173–184. Association for Computing Machinery, Oct 2014.

[Lachaux *et al.*, 2020] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv:2006.03511 [cs]*, Sep 2020. arXiv: 2006.03511.

[Levenshtein, 1966] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, Feb 1966. ADS Bibcode: 1966SPhD...10..707L.

[Mikolov *et al.*, 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. Jan 2013.

[Nguyen *et al.*, 2013] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 651–654. Association for Computing Machinery, Aug 2013.

[Qiu *et al.*, 2024] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Tenspiler: A verified lifting-based compiler for tensor operations. (arXiv:2404.18249), April 2024. arXiv:2404.18249 [cs].

[Quaranta *et al.*, 2021] Luigi Quaranta, Fabio Calefato, and Filippo Lanubile. Kgtorrent: A dataset of python jupyter notebooks from kaggle. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, page 550–554, May 2021. arXiv: 2103.10558.

[Roziere *et al.*, 2021] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv:2102.07492 [cs]*, Oct 2021. arXiv: 2102.07492.

[Roziere *et al.*, 2022] Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv:2110.06773 [cs]*, Feb 2022. arXiv: 2110.06773.

[Sennrich *et al.*, 2016] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv:1508.07909 [cs]*, Jun 2016. arXiv: 1508.07909.

[Sutskever *et al.*, 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *arXiv:1409.3215 [cs]*, Dec 2014. arXiv: 1409.3215.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. Jun 2017.