ADELT: UNSUPERVISED TRANSPILATION BETWEEN DEEP LEARNING FRAMEWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

We propose **A**dversarial **DE**ep **L**earning **T**ranspiler (**ADELT**) for source-to-source transpilation between deep learning frameworks. Unlike prior approaches, we decouple the transpilation of code skeletons and the mapping of API keywords (an API function name or a parameter name). ADELT transpile code skeletons using few-shot prompting on big language models. Based on contextual embeddings extracted by a BERT for code, we train aligned API embeddings in a domain-adversarial setting, upon which we generate a dictionary for keyword translation. The model is trained on our unlabeled DL corpus from web crawl data, without using any hand-crafted rules and parallel data. Our method outperforms state-of-the-art transpilers on multiple transpilation pairs including PyTorch-Keras and PyTorch-MXNet by 22.76 pts and 22.61 pts respectively evaluated by F1 score, and we have made our code, corpus, and evaluation benchmark publicly available.

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.

The most straightforward way to build such a transpiler is neural machine translation (NMT), where we consider a piece of code written using the legacy deep learning framework as a sentence and then train a sequence-to-sequence (seq2seq) (Sutskever et al., 2014) model or a language model (LM) to transpile. However, as it is rarely the case that a parallel corpus is available for any arbitrary pair of source and target frameworks, only unsupervised NMT methods (Artetxe et al., 2018) are applicable. Sadly, such methods are data-hungry. On the other hand, recent research shows that big language models (Brown et al., 2020) pretrained on web crawl data can do translation in a few-shot or even zero-shot manner. Such methods are suitable for transpilation between DL frameworks where labeled training data is unavailable. In our early experiments, we use Codex (Chen et al., 2021) to transpile DL programs using few-shot prompting. The transpiled programs usually have the correct skeletons but are inaccurate on API-specific details, such as API function names and mappings of function parameters.

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 *decouple* the transpilation of skeletal codes from the mapping of API keywords. The transpilation of skeletal codes is the easier part, and large LMs 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 (Fig. 1), a method that leverages this insight to transpile DL code. ADELT outperforms the state-of-the-art end-to-end transpilers. The canonicalized source code is decoupled into two parts: the code skeleton and the API keywords. ADELT transpiles the code skeleton using a pretrained big language model by few-shot prompting. Then each API keyword occurrence is embedded into a vector by PyBERT, a BERT pre-trained 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.

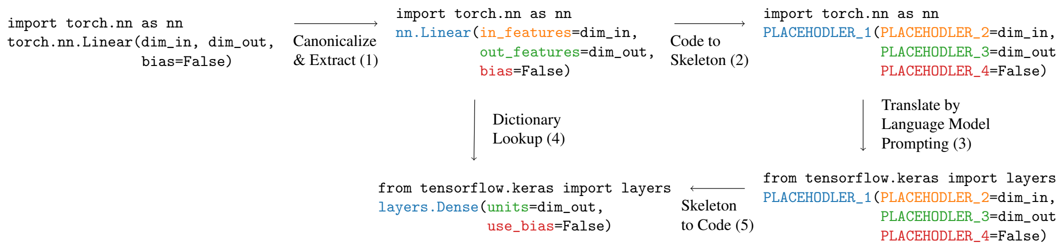

Figure 1: An example of ADELT's pipeline: an import statement in the code skeleton is transpiled from PyTorch to Keras by a language model via few-shot prompting; a linear fully-connected layer is transpiled by removing the argument `in_features` and renaming other API keywords according to the learned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

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 look up each API keyword occurrence in the dictionary and put them back into the transpiled code skeleton.

In summary, this paper makes the following contributions:

- We present ADELT for transpilation between deep learning frameworks without training on any labeled data. ADELT outperforms seq2seq models and other big language models on multiple transpilation pairs, achieving 85.72 F1 score and 95.32 BLEU on PyTorch-Keras transpilation, which is 22.76 pts higher than the state-of-the-art big language model.
- We pre-train a Transformer encoder, PyBERT, on a large-scale corpus of Python code. It can extract high-quality contextual representations of Python code fragments and is part of ADELT.
- To demonstrate our technique, we construct a PyTorch-Keras-MXNet corpus of deep learning code from various Internet sources, containing 19,796 PyTorch modules, 3,703 Keras layers/models, and 1,783 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 big 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 Fig. 1:

1. Extract *API calls* from the source code. Such API calls can be automatically extracted with the `ast` library, a Python built-in module. 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 language model (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 pre-trained 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.

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

2.1 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 $i = 1, 2, 3, \ldots$. The code skeleton is then translated by a big language model (e.g., Codex) using few-shot prompting. The full prompt is shown in Appendix A.5.

2.3 Domain-Adversarial Training

Once the code skeleton is transpiled, we then transpile API keywords. We train the aligned embeddings of the 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 $\mathcal{X}^{(2)}$.

Fig. 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:

[1] `https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy`
[2] `https://docs.python.org/3/library/inspect.html#inspect.Signature.bind`

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.

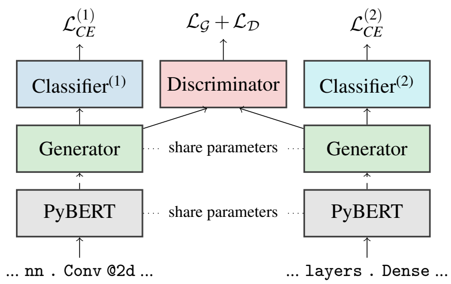

Figure 2: ADELT's domain-adversarial training with contextual embeddings from a PyBERT. The generator and the PyBERT are shared between different DL frameworks. We do not fine-tune the PyBERT during adversarial training.

PyBERT. *PyBERT* is our pre-trained 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 $\mathrm{PyBERT}(x^{(l)})$ 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 multilayer 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 $\mathcal{G}(\mathrm{PyBERT}(\mathcal{X}^{(1)}))$ and $\mathcal{G}(\mathrm{PyBERT}(\mathcal{X}^{(2)}))$ 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 $\mathrm{Pr}_{\mathcal{D}}(\mathrm{pred} = l|\mathbf{z})$ 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 is shown in Eq. (1).

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

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

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 Eq. (2), which is equivalent to the cross-entropy loss of $m^{(l)}$-way softmax-based classification. Here $m^{(l)}$ is the number of distinct API keywords in the DL framework l.

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 is equivalent 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 $\mathcal{L}_{\mathrm{CE}}^{(l)}$ with mini-batch SGD. The joint optimization is crucial, as updating the generator to minimize $\mathcal{L}_{\mathrm{CE}}^{(l)}$ 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 similarity between $w_i^{(1)}$ and $w_j^{(2)}$, denoted by $s_{i,j}$. We can either measure the similarity by dot product or cosine similarity.

Given the scoring matrix, ADELT generates an API keyword dictionary that maps each API keyword in one deep learning framework to an API keyword in another DL framework. In word translation of natural languages (Conneau et al., 2018), *greedy matching* is used to generate a dictionary, where each source word is matched to the target word with the highest similarity score. Unlike natural language words, API keywords are structured. We leverage structures of API keywords to reduce the number of candidates so that the model can choose the correct translation with high probability.

Specifically, we classify all API keywords into two types based on their associated AST node: *callables* (functions or classes), and *parameter names* (names of keyword arguments). In dictionary generation, we do not allow callable names to be translated to callable names. We only allow parameter names to be translated to callable names 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.

Leveraging the structures of API keywords, we propose a *hierarchical API dictionary generation* algorithm: **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.** Apply greedy matching to groups 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[3]. Compared with end-to-end neural networks, our model has better performance.

3.1 Language Model for Skeletal Code Transpilation

We use Codex (Chen et al., 2021) for transpiling code skeletons. Codex is based on GPT-3 (Brown et al., 2020), an autoregressive language model (LM) trained on massive web crawl data. It can be applied to translation tasks with few-shot demonstrations specified purely via text interaction with the model. Codex is a fine-tuned version of GPT-3 using publicly available code from GitHub. The prompt design is similar to the code translation setup of Codex. The prompt consists of a single input-output example and three instructions to make the LM keep placeholders unchanged. Details about prompt designs are shown in Appendix A.5.

3.2 Training Setup

DL corpus. Unsupervised translation usually relies on large unlabeled corpora (Artetxe et al., 2018; Lachaux et al., 2020). For our study, we gather as much relevant source code as possible. We consider 4 data sources **GitHub**, **JuiCe** (Agashe et al., 2019), **Kaggle** (Quaranta et al., 2021), and **Web**. Details are shown in Appendix A.1.

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

[3]We tried to evaluate using JAX (Bradbury et al., 2018). Sadly, JAX is a new DL framework and the GitHub corpus on BigQuery (based on a historical snapshot of GitHub) contains very few (318) examples of JAX.

all sources. Finally, we collect all files into our *DL Corpus* containing 19,796 PyTorch modules, 3,703 Keras layers/models, and 1,783 MXNet modules.

PyBERT. We train the Transformer encoders with the masked language modeling (MLM) (Devlin et al., 2019) objective on all open-source Python files from the GitHub dataset on BigQuery (50.6GB). We call our pre-trained model PyBERT and report results on two model sizes: PyBERT$_{\text{SMALL}}$ (6-layer, 512-d, 45M params) and PyBERT$_{\text{BASE}}$ (12-layer, 768-d, 125M params). The models are pre-trained with the RoBERTa (Liu et al., 2019) pipeline in `fairseq`[4] codebase. We pre-train each PyBERT on the GitHub dataset. On a NVIDIA DGX-2, it takes 8.2 hours and 23.1 hours to train PyBERT$_{\text{SMALL}}$ and PyBERT$_{\text{BASE}}$, respectively. Detailed pre-training hyperparameters are described in Appendix A.2.

Adversarial training. The generator and the discriminator of ADELT are multilayer perceptrons. The activation function is ReLU for the generator and Leaky-ReLU for the discriminator. Dropout and label smoothing are applied for regularization. We train our generator, discriminator, and API keyword embeddings with Adam (Kingma & Ba, 2017) on 1,536,000 samples. There is a linear learning rate warmup over the first 10% of steps, and then we set the LR according to the invert square root decay rule. The maximum learning rate is searched from `[2e-4, 5e-4, 1e-3]`, and the batch size is searched from `[64, 128, 256]` according to the unsupervised validation criterion *"average cosine similarity"* (Conneau et al., 2018) of the generated dictionary, which quantifies the consistency between the learned API keyword embeddings and the generated keyword translations. We set other hyperparameters according to prior works (Conneau et al., 2018) (See Appendix A.3 for details).

3.3 Evaluation Benchmark

We evaluate our method onN transpiling a code snippet from one DL framework to another. We identify potentially matched pairs in the corpus using heuristics (see Appendix A.4) and then manually curate a clean evaluation benchmark of 30 PyTorch-Keras pairs and 25 PyTorch-MXNet pairs.

Following the standard practice of machine translation, we report **BLEU scores**. However, BLEU score is unsuitable for this task because (a) a syntactically incorrect program can have a high BLEU score, while a semantically equivalent program can have a low BLEU score (Ren et al., 2020); (b) a trivial identical mapping baseline already achieves a BLEU score of 57.82 and 61.14 for PyTorch-Keras and Keras-PyTorch, respectively, but it does not give meaningful translations. Therefore, we also report **F1 scores** as a better measurement of the overlap between the prediction and the ground truth: we treat the prediction and the ground truth as bags of function calls; for each test example, let the number of exactly matched calls be n_{match}, the number of predicted calls be n_{pred}, and the number of calls in the ground truth be n_{truth}; the F1 score of this example is defined as $2n_{\text{match}}/(n_{\text{pred}} + n_{\text{truth}})$. We report averages of F1 scores over all test examples. We also report a more rigorous metric, **Exact Match (EM) score**. For each code snippet, a model's transpilation is considered to be an exact match if and only if it is exactly equivalent to the ground truth. The EM score is the number of exact matches divided by the number of examples in the eval set.

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 PyBERT$_{\text{SMALL}}$ and ADELT using 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 GPT-3/Codex baselines, where the entire piece of source code, instead of the code skeleton, is fed into the language model to generate the transpiled target program. For source-to-source translation, we randomly give the LM 5 examples as demonstrations. The prompt design is similar to the code translation setup of Codex, and we add a few tricks to improve the performance. Details about prompt designs and hyperparameter setup are shown in Appendix A.6.

[4] `https://github.com/facebookresearch/fairseq`

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.

	PyTorch-Keras			PyTorch-MXNet		
	BLEU	F1	EM	BLEU	F1	EM
GPT-3 (Brown et al., 2020)	62.73 60.85	30.58 35.62	26.00 29.20	62.12 61.21	25.84 32.79	23.42 25.10
Codex (Chen et al., 2021)	66.76 68.53	62.96 70.72	56.00 60.00	65.84 67.44	57.42 68.96	53.20 56.46
NMT (Lachaux et al., 2020)	61.32 62.76	27.54 26.77	10.37 10.10	59.60 62.25	20.38 21.67	10.46 10.15
Edit Distance (Cased)	78.81 78.31	35.83 34.38	23.34 20.83	79.11 78.30	37.65 35.73	22.85 21.07
Edit Distance (Uncased)	76.29 78.31	27.50 34.38	16.67 20.83	76.40 78.77	30.77 35.97	18.47 20.12
ADELT (Small)	93.83 **92.13**	80.67 80.90	72.67 71.67	92.39 **89.80**	76.63 70.59	66.52 62.88
ADELT (Base)	**95.32** 91.29	**85.72 82.01**	**75.33 72.50**	**93.93** 88.62	**80.03 72.05**	**69.98 63.67**

Neural machine translation (NMT). Lachaux et al. (2020) consider source-to-source transpilation as translation between sentences of code tokens. They train an seq-to-seq unsupervised neural translator, similar to the practice in unsupervised NMT for natural languages (Artetxe et al., 2018). We train an NMT baseline for our task using this method, where the model is initialized with $\text{PyBERT}_{\text{BASE}}$ and trained on our DL corpus.

Edit distance. We consider a 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 standard edit distance (cased) and edit distance between lower-cased keywords (uncased).

The result is shown in Table 1. ADELT, our fully unsupervised approach, consistently outperforms other methods with respect to all metrics, and it benefits from a larger pre-trained PyBERT embedder. Moreover, even if Codex used more examples (5 versus 1) for few-shot supervision, ADELT still consistently outperforms the end-to-end Codex baseline.

The end-to-end unsupervised neural machine translation baseline does not work well when a large monolingual corpus is unavailable. The DL corpus is only 61MB, which is orders of magnitude smaller than the corpus for training TransCoder (Lachaux et al., 2020). Even a simple heuristic baseline using edit distance outperforms NMT, because the API keywords with similar semantics tend to have similar textual representations in different deep learning frameworks (e.g., `stride` of `Conv2d` in PyTorch vs. `strides` of `Conv2D` in Keras).

3.6 Case Studies

Table 2 shows four examples of PyTorch-Keras transpilation together with hypotheses of Codex and ADELT (Base). 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.6 for details). 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.

[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.

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.

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()`

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 having 90.00% precision@1 and 97.73% precision@5 for Keras-to-PyTorch. Developers can even add additional rules to the transpilation algorithm. 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` where `activation` is set) 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.9 for details), 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. We conduct ablation studies on both source-to-source transpilation and API keyword translation. **API keyword translation** involves retrieving the translation of given API keywords. We create a high-quality dictionary by manually translating the first 50 most frequent API keywords in PyTorch and Keras, respectively. Following the standard practice of word translation, we measure how many times the correct translation of a source word is retrieved (**precision@**k for $k = 1, 5$) and the **mean reciprocal rank** of the correct translation (MRR). The results are shown in Table 3.

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. As shown in Table 3, this change significantly harms the performance of ADELT. This justifies the use of PyBERT, a high-quality pre-trained 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 Eq. (2). The result in Table 3 shows that adversarial training contributes ∼6 pts in source-to-source transpilation, showing the effectiveness of adversarial training.

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.

	Keyword						Source-to-Source			
	P@1		P@5		MRR		BLEU		F1	
ADELT (Small)	82.92	90.00	**91.67**	**97.73**	86.97	94.04	93.83	**92.13**	80.67	80.90
ADELT (Base)	**87.08**	90.00	**91.67**	**97.73**	**89.67**	93.96	**95.32**	91.29	**85.72**	**82.01**
Domain-adversarial training										
w/o PyBERT (Small)	52.08	63.64	70.00	85.91	60.54	72.84	78.92	78.66	38.83	45.76
w/o PyBERT (Base)	45.00	54.55	70.42	79.55	56.81	65.99	79.19	76.35	35.33	40.56
w/o Adv Loss (Small)	80.42	88.64	90.00	**97.73**	85.30	93.06	88.89	90.52	68.67	76.32
w/o Adv Loss (Base)	86.25	90.45	**91.67**	**97.73**	89.31	94.27	94.31	85.19	79.89	75.07
Measure for dictionary generation										
Inner Product (Small)	81.25	79.55	**91.67**	90.00	86.34	85.38	93.24	88.49	78.67	77.08
Inner Product (Base)	85.42	**93.18**	**91.67**	**97.73**	88.84	**95.71**	94.38	91.75	82.17	81.46

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). Measures based on 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.

Inspired by unsupervised neural machine translation (NMT) (Artetxe et al., 2018), unsupervised translation of programming languages is made possible recently. Lachaux et al. (2020) use the similar approach to train a transpiler between Python, Java, and C++. However, such methods relies heavily on a massive in-domain unlabeled corpus. For example, Lachaux et al. (2020) train their model on 744GB of source code on GitHub, and Roziere et al. (2022) train their model on a dataset synthesized using 333,542 curated Java functions. However, code related to deep learning available on the Internet is orders of magnitude smaller than this corpus, and we show in Section 3.5 that such method does not work for transpilation between DL frameworks.

Language models are few shot learners. GPT-3 (Brown et al., 2020) is a language model (LM) with 175B parameters trained on massive web crawl data. GPT-3 can be applied to many NLP tasks without any gradient updates or fine-tuning, with tasks 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. In constrast, ADELT is trained in a domain-adversarial setting, and the code generation step is keyword substitution instead of autoregressive generation. ADELT outperforms GPT-3 and Codex in PyTorch-Keras 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* instead of API keyword embeddings.

5 Conclusion

We presented ADELT, a code transpilation algorithm for deep learning frameworks. ADELT formulates the transpilation problem as API keyword mapping, and uses domain-adversarial training to generate the map. Using our collected Pytorch-Keras and PyTorch-MXNet benchmarks, our evaluation shows that ADELT can significantly outperform state-of-the-art transpilers.

Reproducibility Statement

We make our code, corpus, and evaluation benchmark available in the supplementary material. We plan to release them publicly when the paper is accepted. We include training and evaluation setups in Section 3.2 and Section 3.3, respectively. Please refer to Appendix A.9 and Appendix A.3 for more detailed hyperparameter settings. We also release full results with error bars in Appendix A.8.

References

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. URL `http://arxiv.org/abs/1910.02216`. arXiv: 1910.02216.

Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised neural machine translation. *arXiv:1710.11041 [cs]*, Feb 2018. URL `http://arxiv.org/abs/1710.11041`. arXiv: 1710.11041.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

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. URL `http://arxiv.org/abs/2005.14165`. arXiv: 2005.14165.

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. URL `http://arxiv.org/abs/2107.03374`. arXiv: 2107.03374.

Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. *arXiv:1710.04087 [cs]*, Jan 2018. URL `http://arxiv.org/abs/1710.04087`. arXiv: 1710.04087.

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. URL `http://arxiv.org/abs/1810.04805`. arXiv: 1810.04805.

Georgiana Dinu, Angeliki Lazaridou, and Marco Baroni. Improving zero-shot learning by mitigating the hubness problem. *arXiv:1412.6568 [cs]*, Apr 2015. URL `http://arxiv.org/abs/1412.6568`. arXiv: 1412.6568.

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. URL `http://arxiv.org/abs/2002.08155`. arXiv: 2002.08155.

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. URL `http://arxiv.org/abs/1505.07818`. arXiv: 1505.07818.

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. URL `http://arxiv.org/abs/1406.2661`. arXiv: 1406.2661.

Herve Jegou, Cordelia Schmid, Hedi Harzallah, and Jakob Verbeek. Accurate image search using the contextual dissimilarity measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1):2–11, Jan 2010. ISSN 1939-3539. doi: 10.1109/TPAMI.2008.285.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. *arXiv:2001.00059 [cs]*, Aug 2020. URL http://arxiv.org/abs/2001.00059. arXiv: 2001.00059.

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pp. 173–184. Association for Computing Machinery, Oct 2014. ISBN 9781450332101. doi: 10.1145/2661136.2661148. URL https://doi.org/10.1145/2661136.2661148.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980 [cs]*, Jan 2017. URL http://arxiv.org/abs/1412.6980. arXiv: 1412.6980.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv:2006.03511 [cs]*, Sep 2020. URL http://arxiv.org/abs/2006.03511. arXiv: 2006.03511.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, Feb 1966. URL https://ui.adsabs.harvard.edu/abs/1966SPhD...10..707L. ADS Bibcode: 1966SPhD...10..707L.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. Jul 2019. URL https://arxiv.org/abs/1907.11692v1.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. Jan 2013. URL https://arxiv.org/abs/1301.3781v3.

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, pp. 651–654. Association for Computing Machinery, Aug 2013. ISBN 9781450322379. doi: 10.1145/2491411.2494584. URL https://doi.org/10.1145/2491411.2494584.

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)*, pp. 550–554, May 2021. doi: 10.1109/MSR52588.2021.00072. URL http://arxiv.org/abs/2103.10558. arXiv: 2103.10558.

Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Hubs in space: Popular nearest neighbors in high-dimensional data. *Journal of Machine Learning Research*, 11(86):2487–2531, 2010. ISSN 1533-7928. URL http://jmlr.org/papers/v11/radovanovic10a.html.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv:2009.10297 [cs]*, Sep 2020. URL http://arxiv.org/abs/2009.10297. arXiv: 2009.10297.

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. URL http://arxiv.org/abs/2102.07492. arXiv: 2102.07492.

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. URL http://arxiv.org/abs/2110.06773. arXiv: 2110.06773.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv:1508.07909 [cs]*, Jun 2016. URL http://arxiv.org/abs/1508.07909. arXiv: 1508.07909.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *arXiv:1409.3215 [cs]*, Dec 2014. URL http://arxiv.org/abs/1409.3215. arXiv: 1409.3215.

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. URL https://arxiv.org/abs/1706.03762v5.

A APPENDIX

A.1 DATA SOURCES OF OUR PYTORCH-KERAS CORPUS

- **GitHub**: The GitHub public dataset available on Google BigQuery.[6] We keep `py` and `ipynb` files that contain `torch`, `keras`, or `mxnet` in the `main` and `master` branch of the repository. (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 (25.0GB)
- **Kaggle**: All files in KGTorrent (Quaranta et al., 2021), a dataset of Jupyter Notebooks from Kaggle.[7] (54.4GB)
- **Web**: Python code snippets in web pages of relevant websites. We extract texts in `<pre>` tags of HTML files scraped from Stack Overflow[8] (60MB) and PyTorch Forums[9] (25MB).

A.2 PYBERT PRE-TRAINING HYPERPARAMETERS

Table 4 shows the pre-training hyperparemters of PyBERT$_{\text{SMALL}}$ and PyBERT$_{\text{BASE}}$. We first pre-train each model on the Github dataset and then fine-tune it on our canonicalized PyTorch-Keras corpus. The learning rate is decayed according to the inverse square root schedule. We do not use early stopping — we use the last PyBERT checkpoint in ADELT.

Table 4: Pre-training hyperparameters of PyBERT

Hyperparameter	**PyBERT$_{\text{SMALL}}$**	**PyBERT$_{\text{BASE}}$**
Number of layers	6	12
Hidden size d_b	512	768
FFN inner hidden size	2048	3072
Attention heads	8	12
Attention head size	64	64
Dropout	0.1	0.1
Attention dropout	0.0	0.0
FFN dropout	0.0	0.0
Adam β_1	0.9	0.9
Adam β_2	0.98	0.98
Adam ϵ	1e-6	1e-6
Weight decay	0.01	0.01
Gradient clipping	-	-
Peak learning rate	5e-4	5e-4
Batch size	2,048	2,048
Warmup steps	10,000	10,000
Total steps	125,000	125,000

A.3 DOMAIN-ADVERSARIAL TRAINING HYPERPARAMETERS

For domain-adversarial training, we search the peak learning rate from `[2e-4, 5e-4, 1e-3]` and the batch size from `[64, 128, 256]`. Other hyperparameters are shown in Table 5 (top). The learning rates and the batch sizes selected in the hyperparameter search are shown in Table 5 (bottom). The total number of training steps is "total samples" (1,536,000) divided by the searched batch size, which is 6,000 steps for ADELT (Small) and 12,000 steps for ADELT (Base).

A.4 EVALUATION DATA COLLECTION

Our PyTorch-Keras corpus contains some matched PyTorch-Keras pairs. They usually come from open-source projects on GitHub aimed at comparing the performance of PyTorch and Keras using similar neural network architectures. We identify these pairs using a heuristic based on the names of Python classes. Specifically, we consider all pairs of PyTorch module and Keras model/layer that **(a)** share the same class name and **(b)** have a

[6] `https://console.cloud.google.com/marketplace/details/github/github-repos`
[7] `https://kaggle.com`
[8] `https://stackoverflow.com/`
[9] `https://discuss.pytorch.org/`

Table 5: The hyperparameters of domain-adversarial training

Generator hidden size	2,048
Generator layers	1
Discriminator hidden size	2,048
Discriminator layers	1
Discriminator LeakyReLU slope	0.2
Dropout	0.1
Label smoothing	0.2
Adam β_1	0.9
Adam β_2	0.999
Adam ϵ	1e-8
Weight decay	0.001
Discriminator iterations per step	1
Total samples	1,536,000
Peak learning rate (Small)	2e-4
Batch size (Small)	128
Peak learning rate (Base)	5e-4
Batch size (Base)	256

BLEU score greater than 65. Then we manually curate each pair by extracting code segments relevant to deep learning API calls. The resulting parallel corpus has 50 examples, which is too small to be used as a training dataset. So we use the corpus for evaluation of source-to-source transpilation.

A.5 Details of Skeletal Code Transpilation

Table 6 shows by example how we transpile skeletal codes using Codex few-shot prompting.

1. Each API keyword in the canonicalized source program is replaced with an distinct placeholder, numbered from 1 to n (the number of API keywords). The program after this step is called the code skeleton of the source program.
2. We append the code skeleton to the natural language prompt, *# Translate from PyTorch to Keras*, and four input-output pairs. The first three input-output pairs prompt the model to keep placeholders unchanged during transpilation. Our experiments show that three input-output pairs are required for 100% skeletal code transpilation correctness. Also, Codex can generalize to an arbitrary number of placeholders even if only three is given. The last input-output pair is a real example of PyTorch-Keras skeletal code transpilation.
3. This entire piece of input is fed into Codex, and Codex will complete this input by generating tokens after `# Keras`. The output of Codex is considered as the code skeleton of the target program.
4. Each placeholder is replaced with the API keyword in the target DL framework, by querying each API keyword before replacement (step 1) in the API keyword dictionary learned with ADELT.

If the number of placeholders in the source skeleton and the number of placeholders in Codex's output do not match, it is considered a failed example in evaluation. However, in practice, the success rate of skeletal code transpilation is 100% in our experiments. We attribute that to the fact that skeletal code in DL programs, in comparison to arbitrary Python code, tend to be high structured with fairly predictable import statements, constructors, and how the different DL layers are constructed and connected to each other.

A.6 Evaluation Setup of GPT-3 and Codex

Following the practices in Brown et al. (2020) and Chen et al. (2021), we use the *"completion"* endpoint of GPT-3 or Codex for transpilation. We input some text as a prompt with few-shot demonstrations, and the model will generate a completion that attempts to match the prompt. Table 7 shows two examples illustrating how we leverage GPT-3 and Codex for our task.

For source-to-source transpilation, prompt engineering is straightforward. In the PyTorch-Keras transpilation example, we tell the model to "`# Translate from PyTorch to Keras`" and then give 5 demonstrations from our evaluation dataset. Next, we input a piece of source code and "`# Keras`" and let the model generate a code completion starting from the following line. To prevent answers from being leaked to the language model, we do not allow any demonstration to share common API functions with the current evaluation example.

Prompt engineering of API keyword translation is trickier because there are two types of keywords. We represent callable names by one line containing its textual representation, and we represent parameter names by two lines, where the first line is the name of the callable that the parameter belongs to, and the second line is the name of the parameter. We give 10 demonstrations from our evaluation dataset.

Although GPT-3 and Codex have strong capabilities in generating code related to our prompt, we find that they sometimes fail to follow our instructions to transpile between deep learning frameworks. We discuss this problem in Section 3.5. We try several approaches to mitigate this issue:

1. Use the *Instruct* version[10] of GPT-3/Codex: `text-davinci-001` and `code-davinci-001`.
2. Add a prefix to the input prompt based on simple rules. For example, if the source code starts with `nn.` in PyTorch, add `layers.` to the prompt and let the model generate a code completion after it. This trick is applicable to two examples shown in Table 7.
3. Mask the logits of tokens that usually leads to irrelevant generations. Specifically, we find that the model tends to generate irrelevant extra code after a line break or random comments. So we add a bias of -100 to the logits of the hash mark "`#`". We also add a bias of -100 to the logits of the line break if the source code contains no line breaks.

We find that these measures significantly improve the performance of GPT-3 and Codex on deep learning transpilation. All results of GPT-3 and Codex reported in Section 3.5 are from the LMs with all these tricks turned on.

A.7 Cross-Domain Local Scaling (CSLS)

Cross-Domain Local Scaling (CSLS) is a similarity measure for creating a dictionary based on high-dimensional embeddings. CSLS was proposed by Conneau et al. (2018) for word translation between natural languages. Empirical results by Conneau et al. (2018) show that using a pairwise scoring matrix (e.g. cosine similarity, dot product) in dictionary generation suffers from the *hubness* problem (Radovanović et al., 2010), which is detrimental to generating reliable matching pairs as some vectors, dubbed *hubs*, are the nearest neighbors to many other vectors according to s, while others (anti-hubs) are not nearest neighbors of any point. This problem is observed in various areas (Jegou et al., 2010; Dinu et al., 2015). CSLS is proposed to mitigate the hubness problem.

We also conduct an experiment to verify the effectiveness of CSLS in API keyword translation between deep learning frameworks. Specifically, we denote by $\mathcal{N}_s^{(l)}(w)$ the *neighborhood* of API keyword w, a set consisting of K elements with the highest similarity scores with w in DL framework $\mathcal{X}^{(l)}$. We calculate the average similarity score of $w_i^{(1)}$ to its neighborhood in DL framework $\mathcal{X}^{(2)}$ and denote it by $r_i^{(2)}$. Likewise, we denote by $r_j^{(1)}$ the average similarity score of $w_j^{(2)}$ to its neighborhood in DL framework $\mathcal{X}^{(1)}$. Then we define a new similarity measure CSLS of $w_i^{(1)}$ and $w_i^{(2)}$ by subtracting $r_i^{(2)}$ and $r_j^{(1)}$ from their (doubled) similarity score $s_{i,j}$, as shown in Eq. (3).

$$
\begin{aligned}
r_i^{(2)} &= \frac{1}{K} \sum_{k \in \mathcal{N}_s^{(2)}(w_i^{(1)})} s_{i,k} \\
r_j^{(1)} &= \frac{1}{K} \sum_{k \in \mathcal{N}_s^{(1)}(w_j^{(2)})} s_{k,j} \\
\mathrm{CSLS}_{i,j} &= 2s_{i,j} - r_i^{(2)} - r_j^{(1)}
\end{aligned}
\tag{3}
$$

CSLS can be induced from a parameter K and any similarity measure, including dot product and cosine similarity. Intuitively, compared with the score matrix of similarity measure s, the score matrix of CSLS assigns higher scores associated with isolated keyword pairs and lower scores of keywords lying in dense areas.

Given the (cosine similarity) scoring matrix scaled by CSLS, we then apply the hierarchical dictionary generation algorithm (Section 2.4) to generate the API keyword dictionary. We search K in $\{5, 10, 20\}$ according to the unsupervised evaluation metric, and the result is similar, where $K = 5$ gives a slightly better result. Table 8 shows the result of cosine-CSLS compared with cosine similarity.

Table 8 shows that replacing cosine similarity with cosine-CSLS-5 does not impact the F1 score of transpiling PyTorch to Keras significantly, but it hurts the F1 score of transpiling Keras to PyTorch. The reason is that the vocabulary of API keywords is smaller than a natural language vocabulary. Hubness is not a problem for generating API keyword dictionaries; instead, penalizing the top-K may hurt the performance when there are relatively few valid candidates (e.g. Keras-to-PyTorch transpilation). Therefore, we do not use CSLS for ADELT.

A.8 Full Results with Error Bars

Table 9 shows full results with error bars for PyTorch-Keras API keyword translation and source-to-source transpilation. The table includes the results of both the main comparison with GPT-3/Codex and ablation studies.

[10] `https://help.openai.com/en/articles/5832130-what-s-changed-with-engine-names-and-best-practices`

We also add the results of GPT-3 and Codex on API keyword translation, where we randomly give the GPT-3 and Codex 10 examples as demonstrations. Details about prompt designs and hyperparameter setup are shown in Appendix A.6. We do not calculate precision@5 and mean reciprocal rank for GPT-3 and Codex because the API provided by OpenAI does not support ranking a large number of generations cost-efficiently.

A.9 ADELT +

We created a new model, ADELT +, which is based on ADELT but trained on a synthetic dataset. Our goal is to evaluate whether our method can generalize to one-to-many mappings of APIs given enough data.

As we discussed in Section 2.4, we allow parameter names to be translated to callable names when the weight passes a threshold τ. In this case, this parameter will be dropped and a new API call will be generated. This mechanism allows ADELT to transpile `layers.Dense(..., activation="relu")` into two layers: `nn.Linear(...)` and `nn.ReLU()`, and similarly `layers.Conv2D(..., activation="relu")` into `nn.Conv2D(...)` and `nn.ReLU()`. However, such cases are rare in transpiling between deep learning frameworks, making it difficult to evaluate our model's ability to transpile one-to-many mappings in practice. Therefore, we create a synthetic dataset, where we replace all consecutive calls of `layers.Dense` and `layers.ReLU` in our dataset with `layers.Dense(..., activation="relu")`, and we replace all consecutive calls of `layers.Conv2D` and `layers.ReLU` with `layers.Conv2D(..., activation="relu")`. Then we train a new model, ADELT +, using our synthetic dataset.

We then evaluated ADELT + using our evaluation dataset. The value of the threshold τ is set heuristically to 5 (95% of values in the score matrix lies in -7 to 7). Table 2 in Section 3.6 and Table 10 in the appendix show that ADELT + can model one-to-many mappings of APIs. For instance, Table 10 shows that ADELT can transpile `layers.Conv2D` with `activation='relu'` into two API calls: `nn.Conv2d` and `nn.ReLU`.

A.10 More Case Studies

In this section, we select two PyTorch-Keras cases in our evaluation dataset for illustration. They are examples of the average length of all evaluation examples in the evaluation set.

A.10.1 Case 1

```python
# Source Program
import torch.nn as nn
class BasicBlock(nn.Module):
    def __init__(self, dim):
        super.__init__()
        self.bn1 = nn.BatchNorm2d(dim)
        self.act1 = nn.LeakyReLU(0.2)
        self.conv1 = nn.Conv2d(dim, dim, 3)
        self.pool1 = nn.MaxPool2d(3, 2)


# Transpiled by ADELT
import tensorflow.keras.layers as layers
class BasicBlock(layers.Layer):
    def __init__(self, dim):
        super.__init__()
        self.bn1 = layers.BatchNormalization()
        self.act1 = layers.LeakyReLU(alpha=0.2)
        self.conv1 = layers.Conv2D(filters=dim, kernel_size=3)
        self.pool1 = layers.MaxPooling2D(pool_size=3, stride=2)


# Ground Truth
import tensorflow.keras.layers as layers
class BasicBlock(layers.Layer):
    def __init__(self, dim):
        super.__init__()
        self.bn1 = layers.BatchNormalization()
        self.act1 = layers.LeakyReLU(0.2)
        self.conv1 = layers.Conv2D(dim, 3)
        self.pool1 = layers.MaxPooling2D(3, 2)
```

A.10.2 Case 2

```python
# Source Program
import torch.nn as nn
```

```python
class AttentionBlock(nn.Module):
    def __init__(self, args):
        super().__init__()
        self.attn = nn.MultiheadAttention(
            args.d_model, args.n_heads, dropout=args.att_dropout)
        self.drop1 = nn.Dropout(args.dropout)
        self.norm1 = nn.LayerNorm(args.d_model)
```

```python
# Transpiled by ADELT
import tensorflow.keras.layers as layers
class AttentionBlock(layers.Layer):
    def __init__(self, args):
        super().__init__()
        self.attn = layers.MultiHeadAttention(
            num_heads=args.n_heads, key_dim=args.d_model, dropout=args.att_dropout)
        self.drop1 = layers.Dropout(rate=args.dropout)
        self.norm1 = layers.LayerNormalization()
```

```python
# Ground Truth
import tensorflow.keras.layers as layers
class AttentionBlock(layers.Layer):
    def __init__(self, args):
        super().__init__()
        self.attn = layers.MultiHeadAttention(
            args.n_heads, args.d_model, dropout=args.att_dropout)
        self.drop1 = layers.Dropout(args.dropout)
        self.norm1 = layers.LayerNormalization()
```

In each case, ADELT makes the correct transpilation. The only textual difference is that ADELT's transpilation only contains keyword arguments while the ground truth still contains positional arguments. However, because the prediction and the ground truth are the same after canonicalization, we consider each case as an exact match during evaluation.

A.11 Deep Learining Transpilation across Different Programming Languages

In the main paper, all experiments are conducted on Python due to the scarcity of deep learning programs written in other programming languages such as Java or C. Despite that, in this section we show that ADELT is not limited to the same source and target languages by transpiling code written against the PyTorch library in Python 2 to Keras in Python 3.

To do so, we first canonicalize all PyTorch programs into Python 2 and all Keras programs into Python 3. Then we run ADELT on this modified training data to learn the API keyword dictionary. During inference, we transpile the code skeleton with Codex using the prompt shown in Table 11. Besides adding hint words such as "Python2" and "Python3" into the natural language prompt, we also find it necessary to add to the prompt some examples showing differences between Python 2 and Python 3, such as different `print` statements and different integer division operators. As is shown in Table 11, the skeletal codes were successfully transpiled from Python 2 and Python 3 along with the API keywords.

Table 6: Example inputs we give to Codex for skeletal code transpilation. We also show the expected outputs of the language model.

Canonicalized Source Program

```
import torch.nn as nn
dense = nn.Linear(in_features=dim_in, out_features=dim_out, bias=False)
```

Code Skeleton

```
import torch.nn as nn
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in, PLACEHOLDER_3=dim_out, PLACEHOLDER_4=False)
```

Codex Input

```
# Translate from PyTorch to Keras

# PyTorch
PLACEHOLDER_1

# Keras
PLACEHOLDER_1

# PyTorch
PLACEHOLDER_2

# Keras
PLACEHOLDER_2

# PyTorch
PLACEHOLDER_3

# Keras
PLACEHOLDER_3

# PyTorch
import torch.nn as nn
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = PLACEHOLDER_1(PLACEHOLDER_2=16, PLACEHOLDER_3=32, PLACEHOLDER_4=3)
        self.layer2 = PLACEHOLDER_5()

    def forward(self, x):
        x = self.layer1(PLACEHOLDER_6=x)
        x = self.layer2(PLACEHOLDER_7=x)
        return x

# Keras
import tensorflow.keras.layers as layers
class Model(layers.Layer):
    def __init__(self):
        super().__init__()
        self.layer1 = PLACEHOLDER_1(PLACEHOLDER_2=16, PLACEHOLDER_3=32, PLACEHOLDER_4=3)
        self.layer2 = PLACEHOLDER_5()

    def call(self, x):
        x = self.layer1(PLACEHOLDER_6=x)
        x = self.layer2(PLACEHOLDER_7=x)
        return x

# PyTorch
import torch.nn as nn
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in, PLACEHOLDER_3=dim_out, PLACEHOLDER_4=False)

# Keras
```

Expected Codex Output

```
import tensorflow.keras.layers as layers
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in, PLACEHOLDER_3=dim_out, PLACEHOLDER_4=False)
```

Target Program

```
import tensorflow.keras.layers as layers
dense = layers.Dense(units=dim_out, use_bias=False)
```

Table 7: Example inputs we give to GPT-3 or Codex for source-to-source transpilation and API keyword translation. We also show the expected outputs of the language models.

Source-to-Source Transpilation

```
# Translate PyTorch to Keras

# PyTorch
max_len = 512
self.embed_tokens = nn.Embedding(
  n_words, dim_emb)
# Keras
max_len = 512
self.embed_tokens = layers.Embedding(
  n_words, dim_emb, input_length=max_len)

# PyTorch
nn.Linear(dim_in, dim_out)
# Keras
layers.Dense(dim_out)
```

(2 demonstrations omitted)

```
# PyTorch
F.log_softmax(logits, dim=-1)
# Keras
tf.nn.log_softmax(logits, axis=-1)

# PyTorch
nn.Conv2d(64, 128, 3)
# Keras
layers.
```

Expected output:

```
Conv2D(128, 3)
```

Keyword Translation

```
# Translate PyTorch to Keras

# PyTorch
F.log_softmax
# Keras
tf.nn.log_softmax

# PyTorch
nn.MaxPool2d
stride
# Keras
layers.MaxPooling2D
strides
```

(7 demonstrations omitted)

```
# PyTorch
F.relu
# Keras
tf.nn.relu

# PyTorch
nn.Conv2d
out_channels
# Keras
layers.
```

Expected output:

```
Conv2D
filters
```

Table 8: **Results of CSLS.** By default, ADELT computes similarity scores using *cosine similarity* to generate an API keyword dictionary. In this experiment, we replace cosine similarity with *inner product* or *cosine-CSLS-5* to compare different similarity measures. There are two numbers in each table cell: the first one is for transpiling PyTorch to PyTorch, and the second one is for transpiling Keras to PyTorch.

	Keyword						Source-to-Source			
	P@1		P@5		MRR		BLEU		F1	
ADELT (Small)	82.92	90.00	91.67	**97.73**	86.97	94.04	93.83	**92.13**	80.67	80.90
ADELT (Base)	**87.08**	90.00	91.67	**97.73**	89.67	93.96	**95.32**	91.29	**85.72**	**82.01**
Inner Product (Small)	81.25	79.55	91.67	90.00	86.34	85.38	93.24	88.49	78.67	77.08
Inner Product (Base)	85.42	**93.18**	91.67	**97.73**	88.84	**95.71**	94.38	91.75	82.17	81.46
cos-CSLS-5 (Small)	84.17	83.18	**97.92**	93.64	89.89	89.12	94.24	90.43	83.17	76.60
cos-CSLS-5 (Base)	**87.08**	89.55	97.50	**97.73**	**90.63**	93.75	95.20	90.27	85.39	76.18

Table 9: **Full results with 95% confidence intervals**. For each experiment, we run five experiments with different random seeds. Each cell has two intervals: the first one is for transpiling PyTorch to Keras, and the second one is for transpiling Keras to PyTorch. Each interval is the 95% confidence interval according to the Student's t-Test, where we assume that the result of the five experiments follows a normal distribution.

	Keyword		Source-to-Source			
	P@1		BLEU		F1	
LM few shot						
GPT-3 (Brown et al., 2020)	35.42 ± 6.07	39.09 ± 4.19	62.73 ± 4.82	60.85 ± 2.59	30.58 ± 6.35	35.62 ± 8.88
Codex (Chen et al., 2021)	67.50 ± 8.30	79.09 ± 7.83	66.76 ± 2.32	68.53 ± 2.68	62.96 ± 3.44	70.72 ± 2.65
ADELT						
ADELT (Small)	82.92 ± 1.16	90.00 ± 1.55	93.83 ± 0.67	92.13 ± 0.48	80.67 ± 2.78	80.90 ± 1.83
ADELT (Base)	87.08 ± 1.16	90.00 ± 2.52	95.32 ± 0.44	91.29 ± 1.29	85.72 ± 1.13	82.01 ± 3.08
w/o PyBERT (Small)	52.08 ± 2.59	63.64 ± 4.46	78.92 ± 2.47	78.66 ± 6.19	38.83 ± 10.2	45.76 ± 6.04
w/o PyBERT (Base)	45.00 ± 3.92	54.55 ± 5.28	79.19 ± 3.67	76.35 ± 5.55	35.33 ± 9.24	40.56 ± 3.84
w/o Adv Loss (Small)	80.42 ± 1.42	88.64 ± 2.00	88.89 ± 0.40	90.52 ± 1.13	68.67 ± 2.27	76.32 ± 1.89
w/o Adv Loss (Base)	86.25 ± 1.42	90.45 ± 2.36	94.31 ± 0.56	85.19 ± 4.80	79.89 ± 2.58	75.07 ± 5.09
Dot product (Small)	82.92 ± 4.48	90.00 ± 7.19	93.24 ± 0.70	88.49 ± 2.39	78.67 ± 3.32	77.08 ± 4.93
Dot product (Base)	87.08 ± 1.16	90.00 ± 2.00	94.38 ± 0.73	91.45 ± 0.31	82.17 ± 0.62	81.46 ± 0.94

Table 10: **A synthetic example of convolution layer from the evaluation dataset of the Keras-PyTorch transpilation task.** We show the Keras code, ground truth PyTorch 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.

Source	`in_dim = 64`
	`out_dim = 128`
	`layers.Conv2D(filters=out_dim,`
	`  kernel_size=3,`
	`  activation="relu")`

Truth	`in_dim = 64`
	`out_dim = 128`
	`nn.Conv2d(in_dim, out_dim, 3)`
	`nn.ReLU()`

Codex ✗	`in_dim = 64`
	`out_dim = 128`
	`nn.Conv2d(in_dim, out_dim, 3)`

ADELT ✗	`in_dim = 64`
	`out_dim = 128`
	`nn.Linear(in_features=in_dim,`
	`  out_features=out_dim,`
	`  kernel_size=3)`

ADELT + ✓	`in_dim = 64`
	`out_dim = 128`
	`nn.Conv2d(in_channels=in_dim,`
	`  out_channels=out_dim,`
	`  kernel_size=3)`
	`nn.ReLU()`

Table 11: Example of transpiling from PyTorch in Python 2 to Keras in Python 3.

Canonicalized Source Program in Python 2

```
import torch.nn as nn
dense = nn.Linear(in_features=dim_in / 2, out_features=dim_out / 2, bias=False)
```

Code Skeleton

```
import torch.nn as nn
print dim_in, dim_out
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in / 2, PLACEHOLDER_3=dim_out / 2, PLACEHOLDER_4=False)
```

Codex Input

```
# Translate from PyTorch in Python2 to Keras in Python3

# PyTorch in Python2
PLACEHOLDER_1

# Keras in Python3
PLACEHOLDER_1

# PyTorch in Python2
PLACEHOLDER_2

# Keras in Python3
PLACEHOLDER_2

# PyTorch in Python2
PLACEHOLDER_3

# Keras in Python3
PLACEHOLDER_3

# PyTorch in Python2
import torch.nn as nn
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        print "Building Model"
        self.layer1 = PLACEHOLDER_1(PLACEHOLDER_2=16 / 2, PLACEHOLDER_3=32, PLACEHOLDER_4=3)
        self.layer2 = PLACEHOLDER_5()

    def forward(self, x):
        x = self.layer1(PLACEHOLDER_6=x)
        x = self.layer2(PLACEHOLDER_7=x)
        return x

# Keras in Python3
import tensorflow.keras.layers as layers
class Model(layers.Layer):
    def __init__(self):
        super().__init__()
        print("Building Model")
        self.layer1 = PLACEHOLDER_1(PLACEHOLDER_2=16 // 2, PLACEHOLDER_3=32, PLACEHOLDER_4=3)
        self.layer2 = PLACEHOLDER_5()

    def call(self, x):
        x = self.layer1(PLACEHOLDER_6=x)
        x = self.layer2(PLACEHOLDER_7=x)
        return x

# PyTorch in Python2
import torch.nn as nn
print dim_in, dim_out
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in / 2, PLACEHOLDER_3=dim_out / 2, PLACEHOLDER_4=False)

# Keras in Python3
```

Codex Output

```
import tensorflow.keras.layers as layers
print(dim_in, dim_out)
dense = PLACEHOLDER_1(PLACEHOLDER_2=dim_in // 2, PLACEHOLDER_3=dim_out // 2, PLACEHOLDER_4=False)
```

Target Program in Python 3

```
import tensorflow.keras.layers as layers
print(dim_in, dim_out)
dense = layers.Dense(units=dim_out // 2, use_bias=False)
```