vBERT 2020: From Pretraining to Production

Liuyi Jin[1,2,3], Rick Battle[1], Josh Simons[2]
[1]VMware Engineering Service Team, VMware Inc., Palo Alto, CA
[2]Office of CTO , VMware Inc., Boston, MA
[3]Department of Computer Science and Engineering, Texas A&M University, College Station, TX

ABSTRACT

BERT has been the state-of-the-art NLP pretraining models since its release, and achieved the state-of-the-art performance in SQuAD, GLUE and MNLI tasks [1]. In VMware, the vBERT 2020 has been built for the need of dealing with large new text data. In this paper, we demonstrate the plug-and-play pipeline covering the pretraining, fine-tuning, and serving for vBERT 2020. It turns out vBERT 2020 outperforms its elder counterpart vBERT 2018 in terms of accuracy and F1 scores for specified downstream NLP tasks. vBERT 2020 can also achieve descent inference performance after deployed onto CPU-only infrastructure DECC.

1 BACKGROUND AND MOTIVATION

Language understanding and modelling is the fundamental natural task the all successful natural language processing (NLP) models want to resolve first. A well-trained language model is capable of predicting next word given an incomplete sentence with high accuracy. Transformer has been the basic building block for large scale language models since its release in 2017 [2]. Dispensing with recurrence and convolutions, Transformer adopts the essential layered multi-header attention networks as shown in Figure 1. In this way, Transformer rapidly reduces the training time. It also improves the machine translation performance by pushing the BLUE core to be 41.8 on the WMT 2014 English-to-French translation task. In 2018, BERT has been an revolutionary language representation models [1]. It achieves new state-of-the-art results on 11 language processing tasks. Transformer is the basic network architecture unit of BERT. BERT achieves high downstream fine-tuning task performance through pre-training. Figure 2 indicates the overall pre-training and fine-tuning procedures for BERT.

The primary goal of this internship project is establishing, automating, and optimizing the pretraining to production pipeline for VMware's BERT: vBERT.

The first motivation of the project lies in the need of newer version of vBERT. After the release of BERT, there are several BERT variants published, e.g., RoBERTa [3], BioBERT [4]. They are optimized counterparts of BERT. VMware has a similar variant: vBERT 2018, which was built based on the VMware text data before 2018. Approximately, we have additional new $10,000$ tokens that needs to be added to the vocabulary since then. This requires us to update the vocabulary. As a consequence, the tokenizing and word embedding process will be changed. This leads to an ultimate newer version of vBERT: vBERT 2020.

Secondly, the fact that an efficient vBERT pipiline covering from pretraining to production has not been built is also a major motivation. Training a good vBERT model is not easy since it requires much manual effort given a fixed data source.

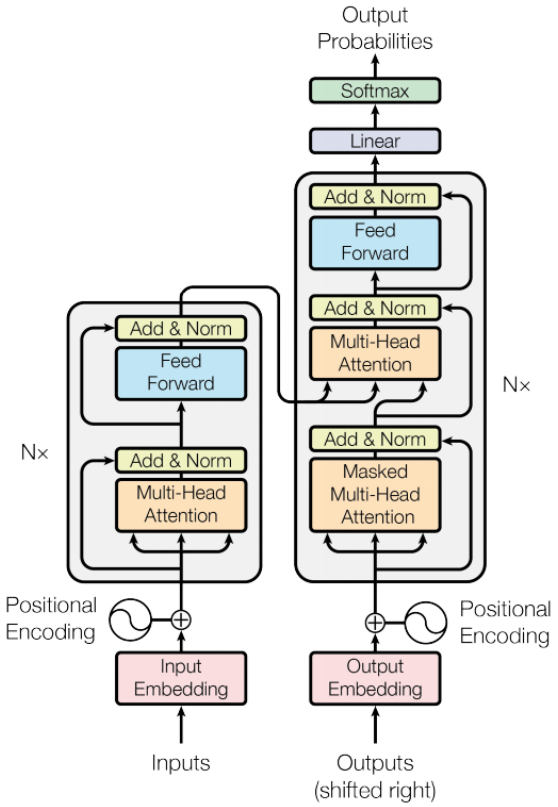

Figure 1: Transformer Architecture Unit

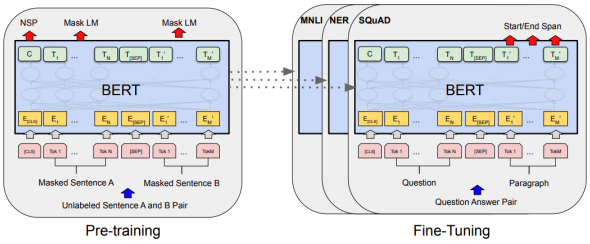

Figure 2: Pretraining and fine-tuning procedures for BERT

During the vBERT pretraining step, we basically generate 20 checkpoints excluding the 0-th checkpoint for maximum sequence length of 128, 2 additional checkpoint for maximum sequence length of 512. For the downstream fine-tuning tasks, we obtained 11 checkpoints in corresponding to each individual pretraining checkpoint. With simple calculation, we need to

train, evaluate and test 242 checkpoints given single pretraining input source tfrecord file. A dozen of data source tfrecord files correspond to thousands of checkpoints that needs to be generated.

Third, there is no mature vBERT services in production at VMware. As we have a vBERT model with good performance, the production-level services heavily rely on the low latency inference time. At VMware, we deploy the fine-tuned vBERT models onto DECC, a CPU-only infrastructure clustuer with vSphere and Kubernetes running on top of it. This additionally requires CPU server inference optimizations during serving stages

2 PIPELINE IMPLEMENTATIONS

In this section, we illustrate the details of the pipeline we've built from vBERT pretraining, fine-tuning, to production deployment. Figure 3 shows the whole pipeline overview. In the following, we provide the details for each component of the pipeline.

It's important to keep in mind that *bert-tensorflow==1.0.1* needs to be installed instead of more recent versions. It may introduce unexpected crash when using more recent version on Colab notebook. Also, all the content in this paper is based on the Linux system.

2.1 Pretraining

2.1.1 Code. The code for vBERT pretraining and its evaluation is named as *Pretrain_BERT.ipynb*. This file should be able to take care of the 2 sections: *Train*, *Eval*. Attention should be paid to the 2nd block of Colab notebook as shown in Figure 4, and Figure 4 is the only part that needs modification for well handling. We can handle the pretraining code by ticking the checking box on the right side. *DOCS_REDDIT* provides the option of if we consider adding the REDDIT texts to the default DOCS texts. *BERT_LARGE* provides the option that if we need to switch to pretrain BERT Large model instead of the default BERT Base model. *MSL512* provides the option that we want pretrain 2 more 512 chpts in addition to 20 128 ckpts. *DEFAULT_VOCAB*, *UNUSED_VOCAB*, *UNUSED_NONASCII_VOCAB* correspond to the 3 vocabulary options: the default vocabulary (gs://vbert/vocab-default.txt) provides by Google[1], the vocabulary with unused tokens replaced (gs://vbert/vocab-vmware-2020-unused.txt), and the vocabulary with unused nonascii tokens replaced (gs://vbert/vocab-vmware-2020-unused-nonascii.txt). It's important to note we must select one of the 3 vocabulary check boxes. The output directory for the pretrained ckpts are automatically generated by the check boxes selection. Take Figure 4 as an example, we add the REDDIT text files to the DOCS, we use BERT base, we additionally pretrain 512 ckpts, and we use the vocabulary with unused nonascii tokens replaced. The directory used to store the corresponding ckpts are gs://vbert/liuyi-vbert-docs-reddit/base/vocab-vmware-unused-nonascii.

EXPORT_DIR is the directory for exported *.pb* models, which is not used in our pretraining pipeline. We need to be cautious of the usage of *DO_DELETE*. *DO_DELETE* is the option to delete all the output directory generated before. The *USE_BUCKET* check box and BUCKET input are used to indicate the BUCKET directory on google cloud.

It's worthy to note that training models with maximum sequence length (MSL) being 512 is disproportionally computation expensive compared to those with 128. In our practice, we pretrained the 10k steps for 512 ckpts starting from every 100k-th 128 ckpts. As shown in the code file *Pretrain_BERT.ipynb*, we set the total pretraining steps for MSL128 to be 200,000. Every 10,000 steps we save a pretrained MSL128 ckpt. In the code, we also saved 2 MSL512 ckpt: one is based on the 100,000 MSL128 step ckpt, the other is based on the 200,000 MSL 128 step ckpt. To give as a data point for reference, a 200,000 steps pretraining for MSL128 and 2 additional 10,000 steps for MSL512 will take more than 9 hours. For

Although this code can be used to take care of the *Train*, *Eval* at the same time, it is recommended to use the 2 section separately, i.e., close the Colab connection after finish each section, especially when we deal with BERT large. This is due to the fact that Google Colab breaks connection every 12 hours. We need to avoid the break of Colab running.

2.1.2 Data. One of the 2 text data sources is VMware Docs only. This includes the general VMware official website documents and product manuals. The other data source includes the extracted Reddit documents into VMware Docs. Before converting the text files into the tfrecord files that would be fed into the pretraining, we apply the vNLP processor first. vNLP is developed by VMware Engineering Service (VES) team for spell checking and compound word splitting. When generating the tfrecord files, we also consider whole word masking and non whole word masking, MSL being 128 and 512 [1]. The specifications for the data sources are shown in Table 1.

```
# Number of training steps.
FLAGS.num_train_steps = 200000

# How often to save the model checkpoint.
FLAGS.save_checkpoints_steps = 10000

# How many steps to make in each estimator
    call.
FLAGS.iterations_per_loop = 10000

# not 128 -> 512
FLAGS.num_train_steps_msl512 =
    FLAGS.iterations_per_loop
```

2.1.3 Pipeline. After we get the data files output from vNLP, we feed them into the google's pretrained ckpt for pretraining. This is shown in the pretraining red box in Figure 3. The output of the pretraining process is vBERT model ready for downstream fine-tuning for better performance on the specific tasks.

2.2 Fine-tuning

2.2.1 Code. The code for vBERT fine-tuning is named as *PR_Esc_PR_Fine_Tune_vBERT.ipynb*. This code file is for fine-tuning the PR2EscalatedPR task. The code structure for fine-tuning is almost the same with that for pretraining except

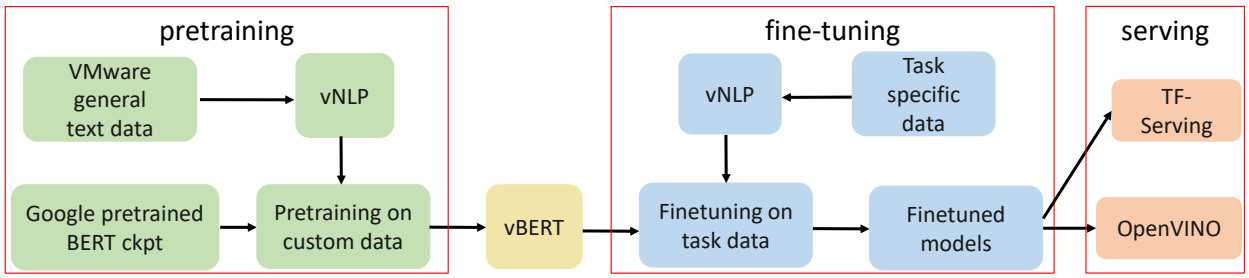

Figure 3: vBERT pipeline: pretraining, fine-tuning and serving

```
# Set the output directory for saving model file
# Optionally, set a GCP bucket location

INPUT_DIR = 'input'#@param {type:"string"}

INPUT_FILE_128 = 'vmware-docs-2020'
INPUT_FILE_512 = 'vmware-docs-2020'
OUTPUT_DIR = 'liuyi-vbert-docs'


DOCS_REDDIT = True #@param {type:"boolean"}
if DOCS_REDDIT:
  INPUT_FILE_128 += '-reddit_'
  INPUT_FILE_512 += '-reddit_'
  OUTPUT_DIR += '-reddit/'
else:
  INPUT_FILE_128 += '_'
  INPUT_FILE_512 += '_'
  OUTPUT_DIR += '/'

BERT_LARGE = False #@param {type:"boolean"}
INPUT_MODEL_DIR = ''
if BERT_LARGE:
  INPUT_MODEL_DIR = 'wwm_uncased_L-24_H-1024_A-16'
  INPUT_FILE_128 += 'wwm_'
  INPUT_FILE_512 += 'wwm_'
  OUTPUT_DIR += 'large/'
else:
  INPUT_MODEL_DIR = 'uncased_L-12_H-768_A-12'
  INPUT_FILE_128 += 'non-wwm_'
  INPUT_FILE_512 += 'non-wwm_'
  OUTPUT_DIR += 'base/'
```

```
INPUT_DIR: "input"
DOCS_REDDIT: ☑
BERT_LARGE: ☐
MSL512: ☑
DEFAULT_VOCAB: ☐
UNUSED_VOCAB: ☐
UNUSED_NONASCII_VOCAB: ☑
EXPORT_DIR: "liuyi_export_pb_model"
Whether or not to clear/delete the directory and create a new one
DO_DELETE: ☐
Set USE_BUCKET and BUCKET if you want to (optionally) store model output on GCP bucket.
USE_BUCKET: ☑
BUCKET: "vbert"
```

Figure 4: Code that needs modification for vBERT pretraining

Table 1: Input data files and corresponding google cloud storage directory

Reddit	Large	MSL512	Default Vocab	Unused	Unused Nonascii	directory
☑	☒	☒	☑	☒	☒	gs://vbert/input/vmware-docs-2020-reddit_non-wwm_msl-128_default-vocab.tfrecord
☑	☒	☒	☒	☒	☑	gs://vbert/input/vmware-docs-2020-reddit_non-wwm_msl-128_vocab-vmware-unused-nonascii.tfrecord
☑	☒	☒	☒	☑	☒	gs://vbert/input/vmware-docs-2020-reddit_non-wwm_msl-128_vocab-vmware-unused.tfrecord
☑	☒	☑	☑	☒	☒	gs://vbert/input/vmware-docs-2020-reddit_non-wwm_msl-512_default-vocab.tfrecord
☑	☒	☑	☒	☒	☑	gs://vbert/input/vmware-docs-2020-reddit_non-wwm_msl-512_vocab-vmware-unused-nonascii.tfrecord
☑	☒	☑	☒	☑	☒	gs://vbert/input/vmware-docs-2020-reddit_non-wwm_msl-512_vocab-vmware-unused.tfrecord
☑	☑	☒	☑	☒	☒	gs://vbert/input/vmware-docs-2020-reddit_wwm_msl-128_default-vocab.tfrecord
☑	☑	☒	☒	☒	☑	gs://vbert/input/vmware-docs-2020-reddit_wwm_msl-128_vocab-vmware-unused-nonascii.tfrecord
☑	☑	☒	☒	☑	☒	gs://vbert/input/vmware-docs-2020-reddit_wwm_msl-128_vocab-vmware-unused.tfrecord
☑	☑	☑	☑	☒	☒	gs://vbert/input/vmware-docs-2020-reddit_wwm_msl-512_default-vocab.tfrecord
☑	☑	☑	☒	☒	☑	gs://vbert/input/vmware-docs-2020-reddit_wwm_msl-512_vocab-vmware-unused-nonascii.tfrecord
☑	☑	☑	☒	☑	☒	gs://vbert/input/vmware-docs-2020-reddit_wwm_msl-512_vocab-vmware-unused.tfrecord
☒	☒	☒	☑	☒	☒	gs://vbert/input/vmware-docs-2020_non-wwm_msl-128_default-vocab.tfrecord
☒	☒	☒	☒	☒	☑	gs://vbert/input/vmware-docs-2020_non-wwm_msl-128_vocab-vmware-unused-nonascii.tfrecord
☒	☒	☒	☒	☑	☒	gs://vbert/input/vmware-docs-2020_non-wwm_msl-128_vocab-vmware-unused.tfrecord
☒	☒	☑	☑	☒	☒	gs://vbert/input/vmware-docs-2020_non-wwm_msl-512_default-vocab.tfrecord
☒	☒	☑	☒	☒	☑	gs://vbert/input/vmware-docs-2020_non-wwm_msl-512_vocab-vmware-unused-nonascii.tfrecord
☒	☒	☑	☒	☑	☒	gs://vbert/input/vmware-docs-2020_non-wwm_msl-512_vocab-vmware-unused.tfrecord
☒	☑	☒	☑	☒	☒	gs://vbert/input/vmware-docs-2020_wwm_msl-128_default-vocab.tfrecord
☒	☑	☒	☒	☒	☑	gs://vbert/input/vmware-docs-2020_wwm_msl-128_vocab-vmware-unused-nonascii.tfrecord
☒	☑	☒	☒	☑	☒	gs://vbert/input/vmware-docs-2020_wwm_msl-128_vocab-vmware-unused.tfrecord
☒	☑	☑	☑	☒	☒	gs://vbert/input/vmware-docs-2020_wwm_msl-512_default-vocab.tfrecord
☒	☑	☑	☒	☒	☑	gs://vbert/input/vmware-docs-2020_wwm_msl-512_vocab-vmware-unused-nonascii.tfrecord
☒	☑	☑	☒	☑	☒	gs://vbert/input/vmware-docs-2020_wwm_msl-512_vocab-vmware-unused.tfrecord

that *PR_Esc_PR_Fine_Tune_vBERT.ipynb* add one more classification layer and includes 3 sections: *Train*, *Eval*, *Test*. The 2nd code block that needs modification is shown in Figure 5. To handle the fine-tuning code, we only need to check/uncheck the checkboxes. For specific information, please refer to Section 2.1.

It should be noted that in fine-tuning, we trained the models for 10 epochs with each fine-tuned ckpt per epoch. We also

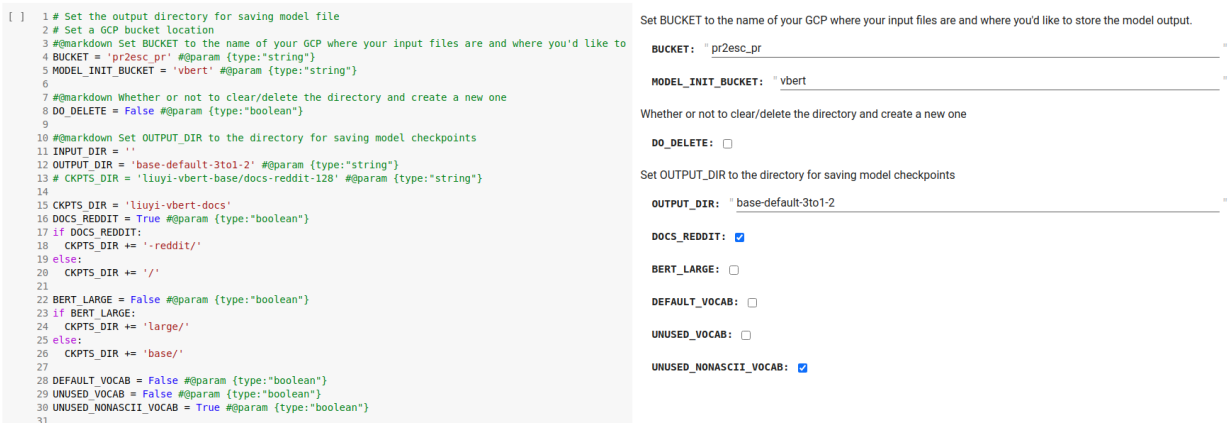

Figure 5: Code that needs modification for vBERT fine-tuning

need to note that, in the following code segment of the *Test* section, we tend to avoid any bugs from google TPU by avoid using *enumerate* for iterating the predicted result. By letting the *next* iterator iterates the *result* generator until the end of result, we want the *result* generator to signal the TPU system to stop and return to CPU. This signal is achieved through the *StopIteration* exception. In this case, *enumerate* will not produce this exception and will require disconnect and reconnect to the Colab runtime.

```
while True:
  try:
    prediction = next(result)
    if num_tests < num_actual_test_examples:
      # raise StopIteration
      probabilities =
          prediction["probabilities"]
      pred = np.argmax(probabilities)
      pred_label = LABEL_NAMES[pred]
      pred = str(pred) # Note: This string is
          still numeric, not the label name!
      pred_correct = ""
      # tf.logging.info("Now we are testing "
          + str(num_tests) + " example in
          test file")
      if pred == test_labels[num_tests]:
        num_correct += 1
        pred_correct = "1"
        if pred_label == TEST_MEASURE_CLASS:
          true_positives += 1
        else:
          true_negatives += 1
      else:
        pred_correct = "0"
        if pred_label == TEST_MEASURE_CLASS:
          false_positives += 1
        else:
          false_negatives += 1
      num_tests += 1
  except StopIteration:
      tf.logging.info("***** StopIteration
          Exception Catch for " +
          str(num_tests) + "-th example *****")
      break
```

2.2.2 Data. Data for fine-tuning tasks heavily depends on the task-specific data. In our internship practice, we are always using *train-3to1.tsv*, *eval-3to1.tsv*, *test-3to1.tsv* for training, evaluation and testing. The directory of the 3 files can be found in gs:pr2esc_pr bucket on the google cloud storage.

2.2.3 Pipeline. As shown in the fine-tuning red box of Figure 3, we feed the processed fine-tuning data into vBERT from pretraining. The output of the fine-tuning are the fine-tuned ckpts ready for serving on the production platforms.

2.3 Serving

In this section, we elaborate serving performance of fine-tuned models developed during our internship project. We are specifically interested in the inference performance on CPU platforms. We used 2 popular production-oriented serving frameworks: TF Serving and OpenVINO Serving. In VMware, DECC provides production environment for many kinds of services with Intel-based CPU support. There are vSphere and Kubernetes running on top of those Intel x86 CPUs. The services are then running on top of Kubernetes. We chose the 2 serving frameworks because they are already optimized for Intel-based x86 CPUs machines.

2.3.1 TF Serving. TensorFlow Serving is a flexible, high-performance serving system for machine learning models, designed for production environments [5]. The overall serving architecture designed in this project is shown in Figure 6.

Given the exported *Fine-tuned-model.pb* file, we loaded it into the docker container. The TF-Server runs as a thread hung within the container. The container further runs with the DECC environment. The TF-Server communicates with the container via the 8501 port, the container communicates with the DECC platform via the 5000 port. When deployed with proper computation resources (e.g., the CPU limit, the memory

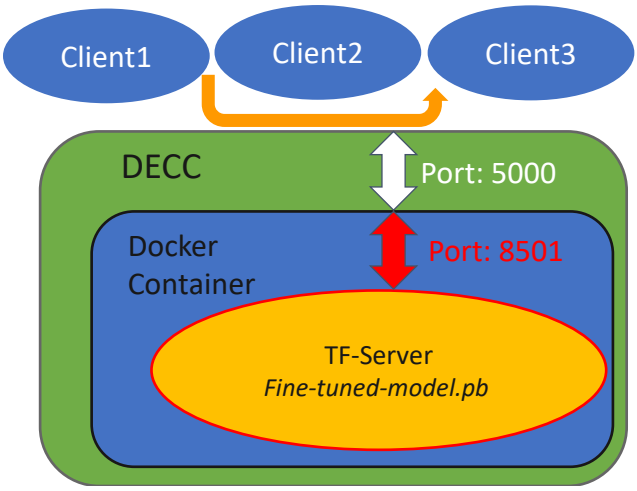

Figure 6: TF Serving architecture designed for vBERT

limit), client can send the request to DECC's 5000 port and get the result from fine-tuned models.

Code: The deployment Gitlab repo is: git@gitlab.eng.vmware.com:gtix-tools/instaml/readyml-services/bertpipeline-deployment.git. Readers can refer to the README file for usage instructions. The key server file is *tf-server.py*. The key code in *tf-server.py* is as follows. This code is the command to hang up the tensorflow server within the docker container.

--port=8500 is the exposed RPC port of TF Server. *--rest_api_port=8501* indicates the restful access port to tensorflow server. *--rest_api_timeout_in_ms=1200000000* is the timeout threshold that TF Server will wait for the model processing, TF Server will stop waiting and return errors if the model processing time is longer than the *rest_api_timeout_in_ms*. *--tensorflow_intra_op_parallelism=6* indicates the CPU cores that TF Server can leverage for parallelism while inferencing. *--enable_batching=true --batching_parameters_file=/app/parameters.conf* enables the TF server to exploit the advantages of batching strategies. It's good to note that TF Server can exploit the most advanced instruction set for inference. For example, on the deployed DECC machine, the TF Server can automatically leverage AVX2 and FMA instruction set for inference acceleration.

```
os.system("nohup tensorflow_model_server
    --port=8500 --rest_api_port=8501
    --rest_api_timeout_in_ms=1200000000
    --tensorflow_intra_op_parallelism=6
    --enable_batching=true
    --batching_parameters_file=
    \"/app/parameters.conf\"
    --model_name=liuyi_sr2pr
    --model_base_path=\"/app/\" >server.log
    2>&1 & sleep 3")
```

2.3.2 OpenVINO Serving. OpenVINO Serving toolkit is designed by Intel. It can be quickly deploy applications and

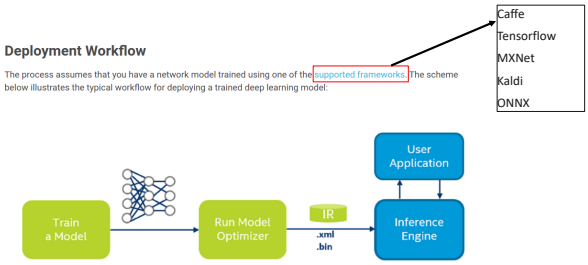

Figure 7: OpenVINO workflow designed for vBERT

solutions across Intel hardware, maximizing performance. Figure 7 [6] illustrates workflow to use OpenVINO. After we obtain the fine-tuned models from fine-tuning as shown in Figure 3, we convert them into Intermediate Representation (IR) format (i.e., .xml and bin) via the *Model Optimizer* component. The inference can then achieved through Inference Engine, which will interact with the client applications.

Code: The code for inference via OpenVINO2020.4 is on the Gitlab repo: git@gitlab.eng.vmware.com:gtix-tools/instaml/readyml-services/bert-openvino-deployment.git. The Code structure is similar to that of TF-Serving except that there is no an additional thread keeping running and carrying the model. It's the FastAPI itself carry the IR models.

It's recommended to directly use images with Intel OpenVINO 2020.4 pre-installed [7] and follow the Dockerfile on the Gitlab. Readers can also follow the detailed installation guide [8] to install OpenVINO locally. After trials and failures for many times, we succeeded to convert the vBERT ckpt model by strictly following the guide described in [9]. We failed to directly convert vBERT pb model with *SavedModel* format into IR format. Due to the fact that Intel provides specific guide on converting BERT ckpts into IR, we assume converting from ckpts, rather than from pb files, is the only feasible way to use with OpenVINO.

In the following, we emphasize how we convert the tensorflow vBERT ckpts into IR format, which is the most challenging part of using OpenVINO:

Step 1: Make sure we have 3 ckpt files, 1 vocabulary file and 1 json file containing bert configurations. Readers can refer to *vBERT_ckpt2IR* folder for the 5 files.
Step 2: git clone https://github.com/google-research/bert.git
Step 3: cd bert
Step 4: git checkout eedf5716c
Step 5: wget https://gist.githubusercontent.com/W4ngatang/60c2bdb54d156a41194446737ce03e2e/raw/17b8dd0d724281ed7c3b2aeeda662b92809aadd5/download_glue_data.py
Step 6: python3 download_glue_data.py --tasks MRPC
Step 7: Open the file modeling.py in the text editor and delete lines 923-924. They should look like this:

```
if not non_static_indexes:
    return shape
```

```
649     with tf.Session(graph=tf.get_default_graph()) as sess:
650         (assignment_map, initialized_variable_names) = \
651             modeling.get_assignment_map_from_checkpoint(tf.trainable_variables(), init_checkpoint)
652         tf.train.init_from_checkpoint(init_checkpoint, assignment_map)
653         sess.run(tf.global_variables_initializer())
654         frozen = tf.graph_util.convert_variables_to_constants(sess, sess.graph_def, ["loss/BiasAdd"])
655         graph_io.write_graph(frozen,'/home/liuyi/Documents/tamu2020summer/VmwareIntern/internship-paper/vBERT_ckpt2IR', 'frozen_loss_BiasAdd.pb', as_text=False)
656     print('BERT frozen model path {}'.format(os.path.join(os.path.dirname(__file__), 'frozen_loss_BiasAdd.pb')))
657     sys.exit(0)
```

Figure 8: Code segment needed in run_classifier.py file for freezing the vBERT ckpt

Step 8: Open the file run_classifier.py and insert the code shown in Figure 8 after the line 645. It's critical to note here that we want to freeze the ckpt to pb file until the *loss/BiasAdd* node although there is a softmax node beyond the *loss/BiasAdd* node. It turns out if we freeze the ckpts until the softmax node, the inference does not work. Alternatively, readers can replace the run_classifier.py file in bert directory with the run_classifier.py in the *vBERT_ckpt2IR*.

Step 9: Set environment variables BERT_BASE_DIR, BERT_REPO_DIR and run the script run_classifier.py to create inference_graph.pb file in the root of the cloned BERT repository. After this step, we could find 2 generated files *frozen_loss_BiasAdd.pb* and *eval.tf_record* in the *vBERT_ckpt2IR* directory.

```
export BERT_BASE_DIR= \
   /path/to/vBERT/ckpts
export BERT_REPO_DIR= \
   /current/working/directory

python3 run_classifier.py \
   --task_name=MRPC \
   --do_eval=true \
   --data_dir= \
      $BERT_REPO_DIR/glue_data/MRPC \
   --vocab_file= \
      $BERT_BASE_DIR/default_vocab.txt \
   --bert_config_file= \
      $BERT_BASE_DIR/bert_config.json \
   --init_checkpoint= \
      $BERT_BASE_DIR/bert_model.ckpt-2559
          \
   --output_dir=./
```

Step 10: Run the Model Optimizer with the following command line parameters to generate vBERT IR. Be careful to run this command in the directory where the OpenVINO is installed. Try to find a similar directory to */opt/intel/openvino_2020.4.287/deployment_tools/model_optimizer*. The *mo_tf.py* is in the directory. Sometimes, we need to change the mod of the *mo_tf.py* file by using *chmod +x mo_tf.py* command.

```
python ./mo_tf.py
   --input_model
   ./vBERT_ckpt2IR/frozen_loss_BiasAdd.pb\
   --disable_nhwc_to_nchw
   --input "IteratorGetNext:0{i32}, \
   IteratorGetNext:1{i32}, \
   IteratorGetNext:4{i32}" \
   --input_shape [1,128],[1,128],[1,128]
   --output_dir ./vBERT_ckpt2IR
```

Step 11: Until now, we should be able to generate the 3 IR files: frozen_loss_BiasAdd.mapping, frozen_loss_BiasAdd.xml, frozen_loss_BiasAdd.bin. After we succeeded to obtain the IR, please put them into the *models* directory on the OpenVINO Gitlab repo for further usages.

3 PERFORMANCE EVALUATION

In this section, we try to evaluate the performance of the fine-tuned models in terms of F1-core, accuracy, recall and precision on test date. We also evaluate the vBERT inference speedup achieved through using TF-Serving and OpenVINO.

3.1 Evaluation for fine-tuned PR2EscPR model

In this section, we mainly deal with the The PR2EscPR task, which aims to predict if we need to escalate a product request to a higher priority. This task is critical in that VMware is developing various new products while engaging maintaining released products, our service teams usually need to prioritize the to-do list of product feature demands from clients. Precise and quick inference essentially improves the efficiency inside VMware.

To give an example, we show the evaluation and test results in Figrue 9 and Figure 10, respectively. If we focus on the F1-score, we can that pretrained model does not have good initial performance. After the fine-tuning, the evaluation and test F1 score get improved to be 0.9 and 0.5, respectively. When we focus on the test F1 score, we still found that several pretrained checkpoints do not work at all.

3.2 Evaluation for Inference Performance

In this section, we provide the inference performance of different serving frames on different hardware platforms. In this section, SR2PR task is employed for inference performance comparisons. Table 2 provides the detailed hardware specifications involved in this evaluation section for reader's reference.

Table 3 gives the detailed comparison of the inference performance of TF-Serving and OpenVINO on 3 different hardware platforms. We compare the performance of 3 different request batch size: 1, 10, 100. *estimator.predict()* is the naive vBERT inference by loading the model first and do the inference later. This is the baseline for comparison.

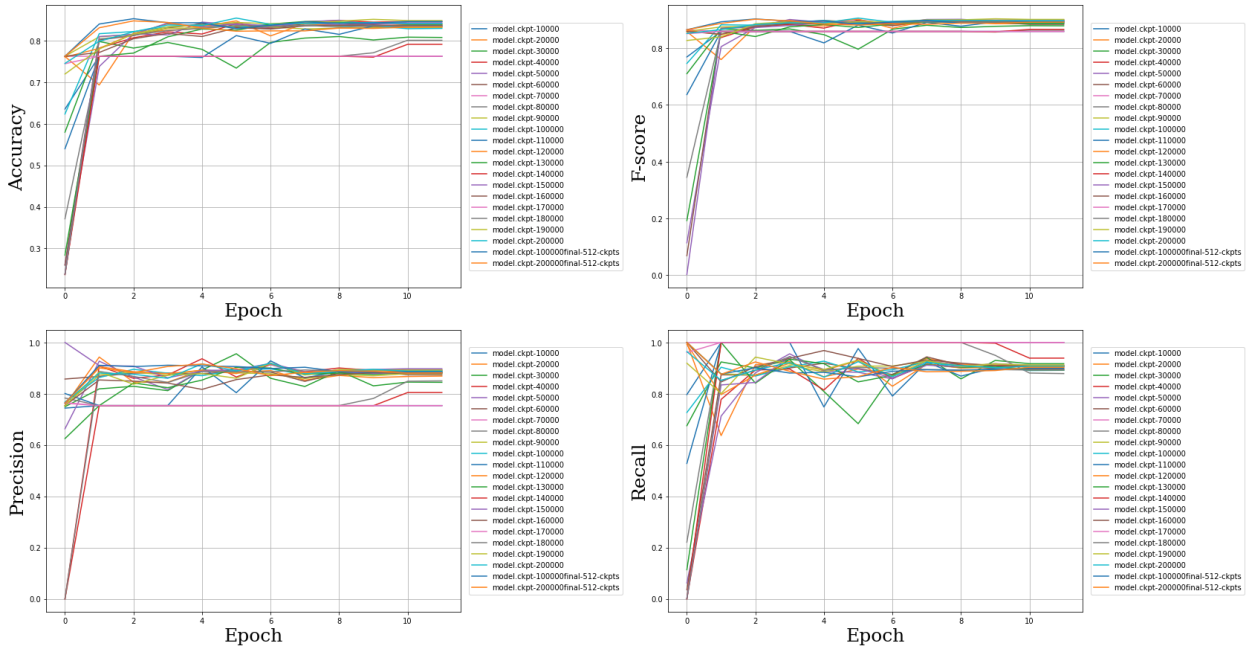

Figure 9: Evaluation Result for PR2EscPR task. The results are generated based on the pretrained model ckpts in directory gs://vbert/liuyi-vbert-docs-reddit/large/default-vocab

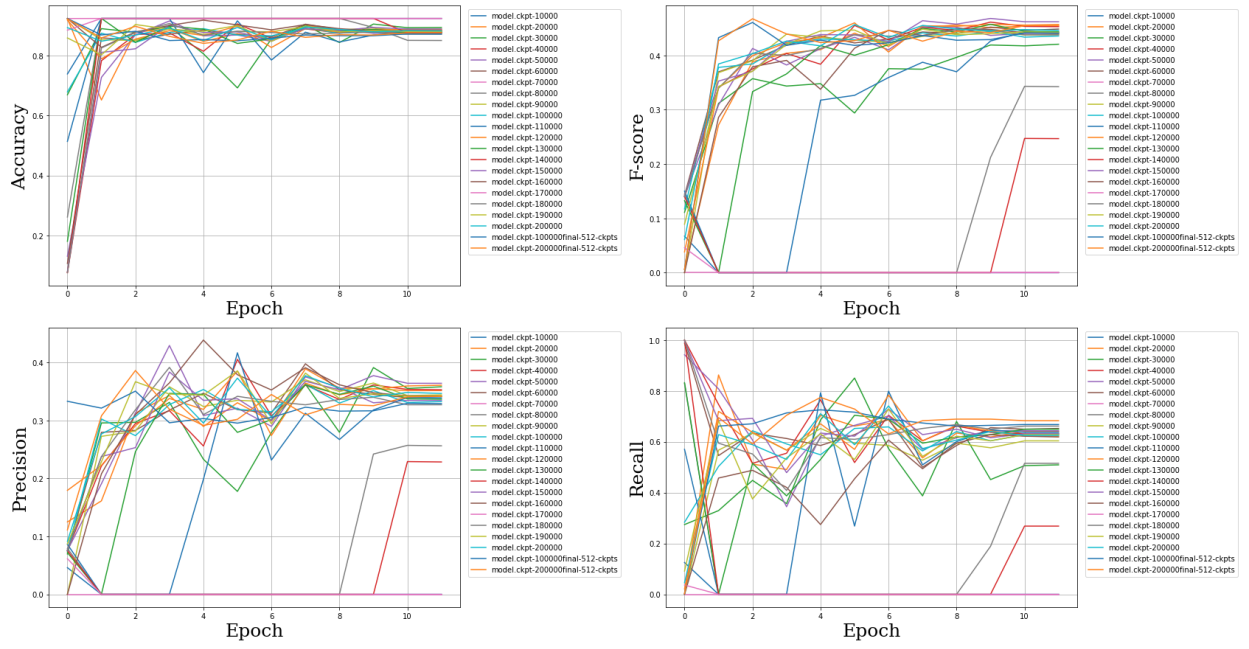

Figure 10: Test Result for PR2EscPR task. The results are generated based on the pretrained model ckpts in directory gs://vbert/liuyi-vbert-docs-reddit/large/default-vocab

Table 2: Hardware Specifications for different platforms

Platforms	Architecture	CPUs	Threads/core	cores/socket	Model Name
Liuyi Local Machine	x86_64(avx)	8	2	4	Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz
DECC (staging)	x86_64(avx2)	6	1	1	Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
HPC Cluster VM	x86_64(avx2)	12	1	1	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz

request_batch_size	1				10				100			
	Feed-forward		End2End		Feed-forward		End2End		Feed-forward		End2End	
Request	TF-Serving	OpenVINO	TF-Serving	OpenVINO	TF-Serving	OpenVINO	TF-Serving	OpenVINO	TF-Serving	OpenVINO	TF-Serving	OpenVINO
1	0.12	0.09	0.49	0.46	0.69	0.51	1.08	0.90	8.27	5.08	8.97	5.78
2	0.13	0.06	0.49	0.43	0.69	0.51	1.08	0.90	8.01	5.04	8.73	5.73
3	0.12	0.06	0.49	0.43	0.68	0.51	1.07	0.90	8.11	5.04	8.84	5.73
4	0.12	0.07	0.48	0.43	0.73	0.51	1.11	0.89	8.32	5.43	9.03	6.13
5	0.12	0.06	0.49	0.43	0.72	0.52	1.11	0.90	7.84	5.07	8.54	5.76
6	0.12	0.06	0.49	0.43	0.72	0.51	1.11	0.90	7.98	5.11	8.71	5.79
7	0.12	0.06	0.49	0.43	0.72	0.51	1.11	0.90	7.92	5.06	8.64	5.76
8	0.12	0.06	0.49	0.43	0.70	0.51	1.09	0.90	8.40	5.06	9.13	5.74
9	0.13	0.06	0.50	0.43	0.71	0.51	1.10	0.90	8.08	5.05	8.80	5.75
10	0.13	0.06	0.49	0.43	0.72	0.51	1.11	0.90	8.10	5.22	8.80	5.91
11	0.12	0.06	0.49	0.43	0.74	0.51	1.12	0.90	7.75	5.06	8.47	5.75
12	0.11	0.06	0.48	0.43	0.69	0.51	1.08	0.90	8.07	5.07	8.78	5.76
13	0.12	0.06	0.49	0.43	0.70	0.51	1.09	0.90	8.06	5.07	8.79	5.76
14	0.12	0.06	0.49	0.43	0.70	0.51	1.09	0.90	8.39	5.07	9.12	5.77
15	0.13	0.06	0.50	0.43	0.70	0.51	1.08	0.90	8.27	5.13	9.05	5.82
16	0.12	0.06	0.59	0.43	0.69	0.52	1.18	0.90	8.27	5.08	9.15	5.78
17	0.11	0.06	0.58	0.43	0.72	0.51	1.21	0.90	8.43	5.07	9.26	5.76
18	0.13	0.06	0.50	0.43	0.72	0.51	1.12	0.89	8.31	5.13	9.05	5.82
19	0.13	0.06	0.50	0.43	0.72	0.51	1.11	0.90	8.22	5.08	8.94	5.77
20	0.11	0.06	0.48	0.43	0.77	0.51	1.17	0.90	8.23	5.06	9.03	5.75
min	**0.11**	**0.06**	**0.48**	**0.43**	**0.68**	**0.51**	**1.07**	**0.89**	**7.75**	**5.04**	**8.47**	**5.73**
max	**0.13**	**0.09**	**0.59**	**0.46**	**0.77**	**0.52**	**1.21**	**0.90**	**8.43**	**5.43**	**9.26**	**6.13**
trim_average	**0.12**	**0.06**	**0.50**	**0.43**	**0.71**	**0.51**	**1.11**	**0.90**	**8.15**	**5.10**	**8.89**	**5.79**
trim_average/request_batch_size	**0.12**	**0.06**	**0.50**	**0.43**	**0.07**	**0.05**	**0.11**	**0.09**	**0.08**	**0.05**	**0.09**	**0.06**
variance	**0.00004**	**0.00004**	**0.00090**	**0.00004**	**0.00045**	**0.00000**	**0.00137**	**0.00001**	**0.0364**	**0.0077**	**0.0458**	**0.0080**

Figure 11: Inference performance of TF Serving and OpenVINO on HPC VM

If we look at the request batch size of 100, we can see that TF-Serving on local machine and OpenVINO on local machine can achieve 3.2 times and 4.7 times speedup, respectively. This turns out deploying vBERT models using serving frameworks can help accelerate the inference to a great extent. OpenVINO outperforms TF Serving on both local machine and HPC VMs. This can be expected since OpenVINO is using Intel hardware-aware optimized model format. The distinctive advantage of OpenVINO is more obvious in Figure 11. For all the 3 batch sizes, OpenVINO outperforms TF-Serving in terms of the both feed-forward time and end2end time. In addition, we find that inference time of OpenVINO is much more stable that of TF-Serving since the time variance for OpenVINO is always near 0.

It's good to note that for TF Serving, we need to enable all kinds of optimization flags to fully exploit the hardware parallelism. OpenVINO automatically leverages the full Intel hardware potential. This can be obviously observed when we compare Figure 12 with Figure 13. Figure 12 is the CPU usage snapshot when TF Server is running on HPC Cluster VM. Figure 13 shows the CPU usage snapshot when OpenVINO is running on HPC cluster VM. TF Server can only exploit 60~80% computation potential while OpenVINO is capable of levergaing all the 100% computation potential. This reasons OpenVINO's advantage as a serving framework over TF Server on intel hardware.

4 CONCLUSIONS AND FUTURE WORK

Here we conclude the paper with 4 conclusions.

1. We built the end-to-end pretraining, fine-tuning, and production inference pipeline with Bert model for downstream tasks, this pipeline is plug-and-play and efficient.

2. We optimized the pretraining and fine-tuning pipeline by adding the regularizer early stopping and extending the maximal sequence length to be 512.

3. Optimized the production inference by leveraging TF-Serving (x3.2) and OpenVINO(x4.7)

4. Deployed the model on staging env for VMware internal service.

ACKNOWLEDGMENTS

Thanks for the support from my teammates, Na Zhang, Giampiero Caprino, Steve Liang and all the professional and nice engineers at VMware.

REFERENCES

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

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

[3] 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, 2019.

[4] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 09 2019.

Table 3: Inference Performance of different serving framework on different platforms(excluding data processing and network routing time latency)

Framework on Platforms	SR/Req(seconds)			Speedup
	1	10	100	
estimator.predict() on local	14	22	90	baseline
TF-Serving on local	0.38	3	28	x3.2
OpenVINO on local	0.19	1.92	19.02	x4.7
TF-Serving on HPC VM	0.12	0.71	8.15	x11.04
OpenVINO on HPC VM	0.06	0.51	5.10	x17.6
TF-Serving on DECC	0.3~0.7	1.5~7	16~22	x5.6~x4.1

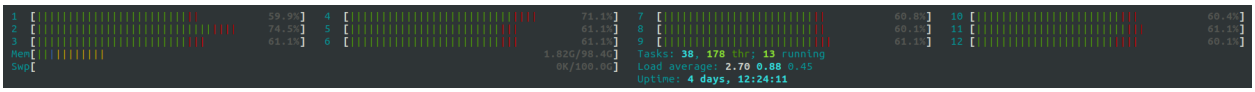

Figure 12: When TF Serving is running, CPU usage on HPC Cluster VM is 60~80%

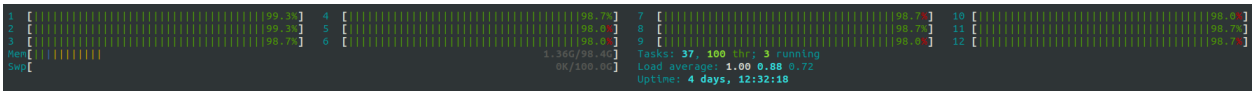

Figure 13: When OpenVINO Serving is running, CPU usage on HPC Cluster VM is ~ 100%

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] Intel. Introduction to intel® deep learning deployment toolkit, Aug 2020.

[7] Dockerhub. Openvino-model-server:2020.4, Aug 2020.

[8] Intel. Install intel® distribution of openvino™ toolkit for linux*, Aug 2020.

[9] Intel. Convert tensorflow* bert model to the intermediate representation, Aug 2020.