Tsunami: Training in TVM with Relay

TVMCon 2021 Lightning Talk

Altan Haan
MLSys @ OctoML

OctoML

Current status of TVM training in main

- Gradient coverage for operators
  - Still missing some, but steadily improving in main
- Automatic differentiation (AD) robustness
  - graph AD is now fairly stable
- No framework to handle the “high-level” training features
  - Parameters (initialization, tracking, updating)
  - Training optimizers (SGD, ADAM, etc.)
  - Model definitions (with parameter tracking, serialization)
  - We could try using another framework to handle these, but integration with TVM is tricky (see e.g. https://lernapparat.de/transformers-pytorch-tvm/)

Current status of TVM training in main

- Gradient coverage for operators
  - Still missing some, but steadily improving in main
- Automatic differentiation (AD) robustness
  - graph AD is now fairly stable
- No framework to handle the "high-level" training features
  - Parameters (initialization, tracking, updating)
  - Training optimizers (SGD, ADAM, etc.)
  - Model definitions (with parameter tracking, serialization)
  - We could try using another framework to handle these, but integration with TVM is tricky (see e.g. https://lernapparat.de/transformers-pytorch-tvm/)

Tsunami: a simple TVM training framework

- Provides a modular API for defining models, based off of established training framework principles (e.g. PyTorch's nn.Module)
- Tracks parameters for models, handling initialization and weight updates
- Provides training optimizers
- All computations (model layers, optimizer updates for weights and internal state) are written in Relay
- Backward graph construction is handled behind the scenes, with a simple user-facing API for running a training step

Example code: Module

```python
class BertLMPredictionHead(Module):
    def __init__(self, config: BertConfig):
        super().__init__()
        self.transform = BertPredictionHeadTransform(config)
        self.units_out = config.vocab_size
        self.bias = Parameter((config.vocab_size,), ts.parameter.Constant(0))

    def forward(self, hidden_states, word_embedding_weight):
        hidden_states = self.transform(hidden_states)
        predictions = (
            nn.batched_dense_op(
                hidden_states, word_embedding_weight, units=self.units_out
            )
            + self.bias()
        )
        return predictions
```

sub-module

Trainable parameter (init to 0)

Builds a Relay expression

Example code: Optimizer

```python
class SGD(Optimizer):
    def __init__(self, parameters, device, lr=0.01, momentum=0.0, init=False):
        self.lr = lr
        self.momentum = momentum
        self.has_momentum = momentum ≠ 0.0
        super().__init__(parameters, device, init=init)

    def init(self):
        self.register_state(np.array([self.lr], dtype="float32"), device=self._device, name="lr")
        if self.momentum ≠ 0.0:
            self.has_momentum = True
            self.register_state(
                np.array([self.momentum], dtype="float32"), device=self._device, name="momentum"
            )
            for param in self.parameters:
                self.register_state(
                    np.zeros(param.concrete_shape, param.dtype),
                    device=self._device,
                    name=param.fqn + "_velocity",
                )
```

Example code: Optimizer

```python
def update(self, weights, grads):
    assert len(self.parameters) == len(weights) == len(grads)
    state_updates = []
    weight_updates = []

    # lr doesn't change inside training iteration
    state_updates.append(self.state_var("lr"))
    if self.has_momentum:
        state_updates.append(self.state_var("momentum"))

    for param, weight, grad in zip(self.parameters, weights, grads):
        if self.has_momentum:
            w_v_next = self.state_var("momentum") * self.state_var(
                param.fqn + "_velocity"
            )
            w_v_next = w_v_next - self.state_var("lr") * grad
            state_updates.append(w_v_next)
            w_next = weight + w_v_next
        else:
            w_next = weight - self.state_var("lr") * grad
        weight_updates.append(w_next)

    return weight_updates, state_updates
```

Putting it all together

Compile and initialize the model:

```
model = BertForMaskedLM(config)
optim = SGD(model.parameters(), DEVICE, lr=lr, momentum=0.0)
model.compile_train(
    optim=optim,
    device=DEVICE,
    target=TARGET,
    profiling=False,
    opt_level=3,
    mixed_precision=mixed_precision,
)
```

Inline the optimizer computation

Train the model:

```
loss = model.train([input_ids, token_type_ids, attention_mask], [labels])
optim.step()
```

Mixed Precision Training

Currently we take a simple approach:

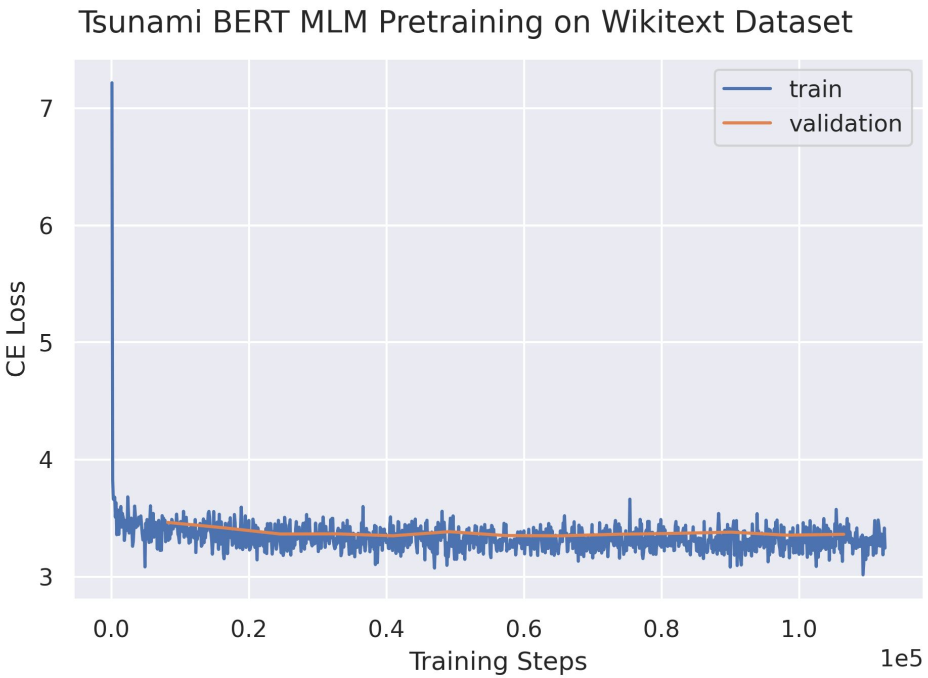
- Transform FP32 GEMMs to FP16 input -> FP32 output mixed precision GEMMs
- Insert FP16 casts before GEMM calls
- Scale the loss value by a given scaling factor (in case of dynamic range issues)
- Scale gradients by inverse of loss scale before performing weight update

This approach is easily realized as a Relay transformation pass in ~50 LOC Python

Possible future extensions:

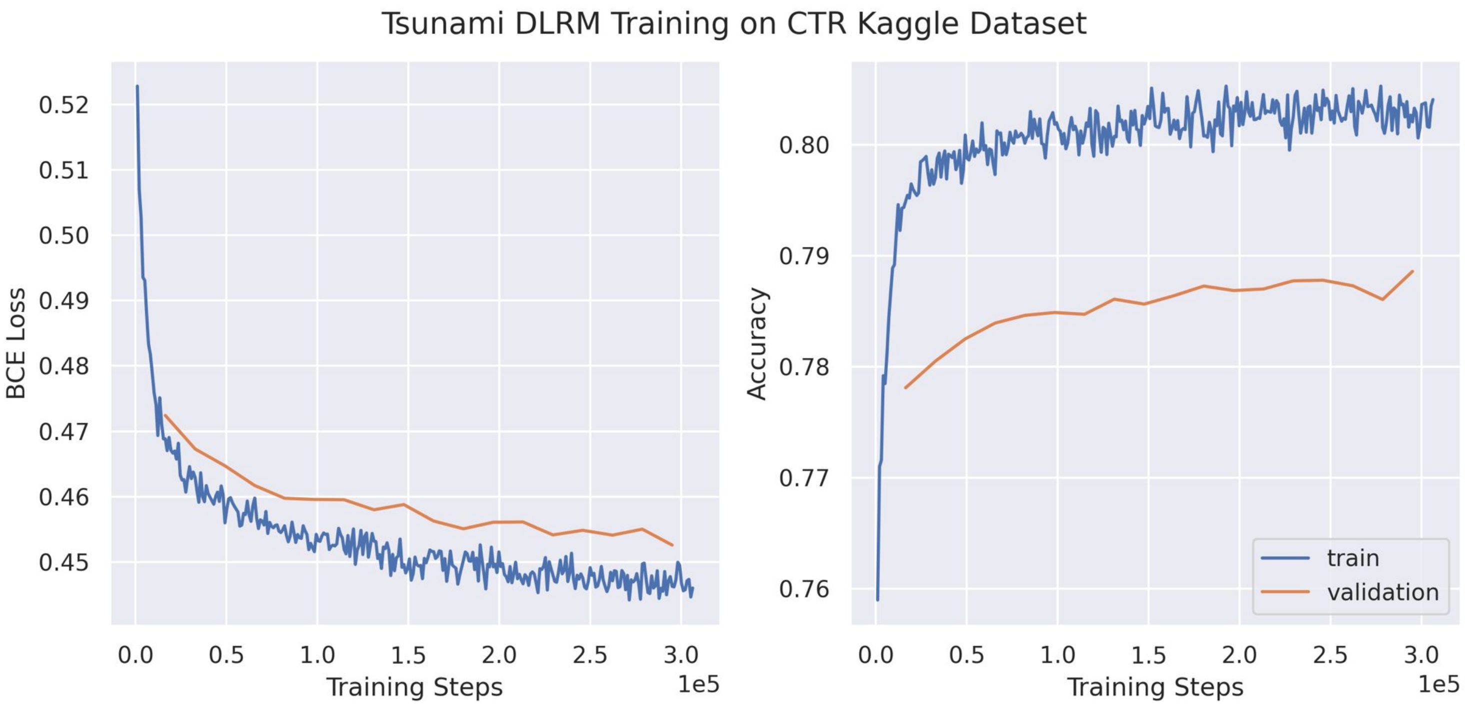
- FP16 support for more ops
- Cast and keep majority of graph in FP16, cast back for numerically sensitive ops
- Try using FP16 -> FP16 GEMMs

Case study: BERT and DLRM



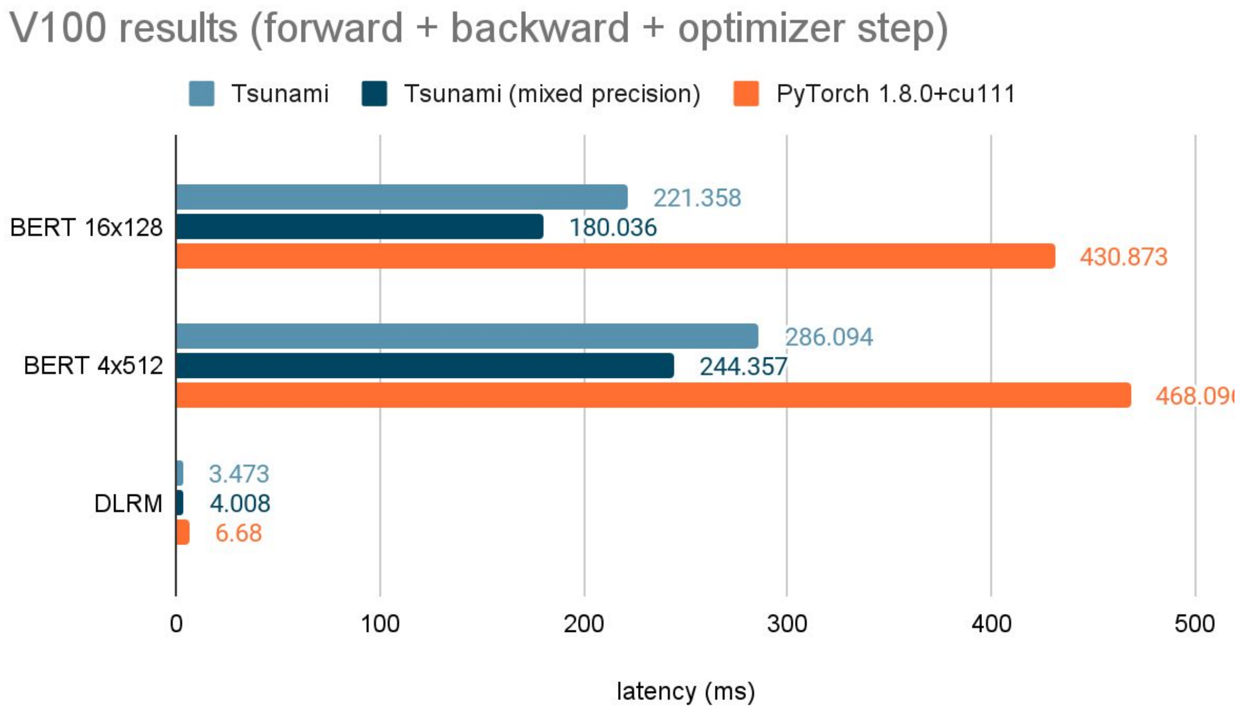
OctoML

Case study: BERT and DLRM



OctoML

Case study: BERT and DLRM



OctoML

Limitations & Future Work

- Stateful operators like batch norm and dropout
- In-place operations like sparse embedding weight updates in DLRM
- Dynamic shape support (most commonly dynamic batch size)
- Non-graph model support
- Check out the ongoing work on **Relax**, which aims to address these limitations of Relay.

Acknowledgements

Thanks to my OctoML collaborators: Tianqi Chen, Tristan Konolige, Wuwei Lin, Thierry Moreau, Jared Roesch, Junru Shao, Chris Sullivan, Bing Xu (now at FB).

This work was done in partnership with Michiel Vermeulen at AMD - we thank them for their support!

Thank you!