Gluon Fit API - Tech Design

Note: extended design for callbcaks here

Problem and Goals

Background

Training a model in Gluon requires users to write the training loop, this is useful because of its imperative nature, however repeating the same code across multiple models can become tedious and repetitive with boilerplate code. The training loop can also be overwhelming to some users new to deep learning. Users have asked for a simple Fit API, similar to APIs available in SKLearn and Keras (example forum ask) as a way to simplify model training and reduce boilerplate code and complexity.

Target Users

1. Beginners who are new to deep learning and/or to Gluon.
2. Applied scientists and deep learning practitioners training models with low complexity and non-custom requirements.

Goals and Deliverables

- Introduce a new Gluon “Fit” API that eliminates the need to code a training loop for simple model use cases, thus reduces manual errors and friction.
- Support Fit API handlers that enable to customize the training loop for things like checkpointing, logging, early stopping and metrics inspired by Keras Callbacks.
- Maintain backwards compatibility: the existing Gluon way to train a model will be supported and maintained - it is needed for complex models and full imperative control by the user.
- The new Fit API will cover beginners use-cases including canonical CV and NLP models, full list is in appendix. For advanced users and complex models, the recommended path is to use the existing training loop.
- Test coverage: 100% unit test coverage and 100% integration test coverage for the example models in Appendix .
- Educate Gluon users via: (1) Blog post (2) Example (3) Tutorial

Existing user experience

Currently in Gluon because of its imperative style of programming, users write the entire training loop which requires multiple steps. To see a code example, see appendix A.
Writing the custom loop involves:

1. Set-up the train/validate data
2. Do a forward pass on batch of data
3. Calculate loss
4. Calculate gradients after back-propagation
5. Update weights
6. Evaluate metrics

7. Update logs
8. Save models

The above steps are generic and remains same across many models being trained. This is a repetitive work and we can simplify this by providing a simple “fit” API for the user that caters for 80% of modeling use-cases especially for novice users. We prefer advanced users continue to write their own custom training loops which gives them more flexibility.

Proposed Approach

We propose to add a simple “fit” API for Gluon by offering a new Estimator class which includes a fit method - similar to SKLearn Classifier API. Gluon Estimator will hold details of the model training like training statistics, training network and event handlers.
We will have EventHandler base class that exposes methods for the stages of the training loop viz. **train_begin, train_end, epoch_begin, epoch_end, batch_begin and batch_end** which gives users flexibility to override and customize different stages of training. We will also provide default event handlers will for common actions such as Logging, Metrics, EarlyStopping, Checkpointing.
So, the “fit” method will run forward pass, calculate loss and gradients, log, checkpoint and update metrics.

new user experience

The new API described below reduces the number of lines of code to be written by the user. In cases where the user implements logging and checkpointing, the number of lines is reduced from ~40 to ~6

Below is an example for the Fit API implementing similar functionality to the one using the existing training loop in Appendix A.

```
import mx.gluon.estimator as est
net = get_model() ## get the network
loss = gluon.loss.CrossEntropy()
e = est(net, lossfn = loss)
## training
trainers = [gluon.Trainer('sgd',{'learning_rate':0.001})]
e.fit(train_data, val_data, epochs, trainers, context)
```

Error Handling

We will provide the following checks to make sure the fit method is robust and easy to debug. We aim to let user know clearly what's wrong with clear error messages.

1. Checking required arguments and optional arguments, notify users if default values are used.
2. Checking the number of inputs from DataLoader and the number of losses.
3. Check the number of outputs from network and the number of inputs for losses and metrics.

All error handling will be covered in unit tests.

Estimator class

Estimator is the new class that will encapsulate training and expose a Fit method.

```
class Estimator:
     def __init__(self, net,  lossfn=gluon.loss.CrossEntropy(),
                  metrics=[mx.metric.TopKAccuracy(), mx.metric.RMSE()]):
         self._train_stats = {"lr" : [], "epoch":[],"train_metric1" : [], "val_metric1" : [],
                              "train_loss1" : [], "val_loss1" : [] , "time":[] ...}
         self._net = net
         self._loss = lossfn
         self._metrics = metrics
         self._loggingHandler= LoggingHandler(self)
         self._checkpointingHandler= CheckpointHandler(self)
         self._metricsHandler= MetricHandler(self)
         self._additionalHandlers= [] ##can be the custom eventhandlers


    def fit(self, train_data_loader,
                  val_data_loader,
                  epochs,
                  trainers=[gluon.Trainer('sgd',{'learning_rate':0.001})],
                  context=[mx.Context.default_ctx],
                  batch_size=None):
         pass

     @property
     def metrics:
        return self._metrics

     @property
     def additionalHandlers:
        return self._additionalHandlers

     @property
     def loggingHandler:
      return self._loggingHandler

     @property
     def checkpointhandler:
        return self._checkpointHandler

     @property
     def metricHandler:
        return self._metricHandler

     @property
     ##Loss fn should take predictions and labels as input and return a scalar loss
     def loss:
        return self._loss

     def plot_loss_graph(self):
        pass
        ## plan to support MXboard in next phase
```

EventHandler CLASS

EventHandler is a new class to offer customizing the training process by offering callbacks for handling different stages of the training.
We will implement standard handlers for logging, checkpointing and more. The user can extend EventHandler and implement their own custom handlers.

```
class EventHandler:
     def __init__(self,estimator):
        self._train_stats= estimator.train_stats

def train_begin(self):
pass
def train_end(self):
pass
def batch_begin(self):
pass
def batch_end(self):
pass
def epoch_begin(self):
pass
def epoch_end(self):
pass
```

```
class LoggingHandler(EventHandler):
    def __init__(self, estimator, log_loc = './'):
        # setup logging
    def epoch_end:
         ## log the train stats to log location

 class CheckpointHandler(EventHandler):
    def __init__(self, estimator, checkpoint_interval=5 , ckpt_loc='./', monitor= "val_loss"):
        super.__init__()
        train_stats = {"lr" = [0.1], "train_acc" = [0.85], "val_acc" = [0.99], ... }
     def epoch_end:
       ## save the model params to the checkpointing location

class MetricHandler(EventHandler):
    def __init__(self, estimator):
        super.__init__()
        train_stats = {"lr" = [0.1], "train_acc" = [0.85], "val_acc" = [0.99], ... }
    def epoch_end:
       ## calculate and update metrics for thr training dataset
       ## update_metrics(pred, labels)- default implementation can be overriden in case of multi-
output cases
       ## update validation metrics for validation dataset

class EarlyStopping(EventHandler):
    def __init__(self, monitor= "val_loss", min_delta=0, patience=0, mode="auto", baseline=None,
restore_best_params=False):
        # setup early stopping rules based on the metric/loss monitor and the mode
        # e.g. if "acc" use greater mode else use lesser
    def on_epoch_end:
        # if metric improved, record the best value
        # else wait n epochs(n=patience) and stop trainning
        # restore net parameters from the best epoch accordingly
    def on_train_end:
        # let user know if early stopping is triggered
```

Addition of New APIs

The design adds new classes Estimator and EventHandler which solves the problem at hand.

Backward compatibility

The design doesn't alter any existing APIs and so the design is backward compatible.

Performance Considerations

The design proposes APIs to substitute training loops in Gluon and shouldn't have any performance regressions.
We can add tests to compare training using training loops vs the new APIs and compare the training times to understand to make sure that their is no regression.

Test Plan

We will implement 100% unit test coverage for the new API and event handlers.

1. A test with all possible parameters
2. A test with default parameters
3. A test with missing parameters

We will add integration tests covering all of the models in the release goals (Appendix). See comment section for integration test plan.

Technical Challenges / Open Questions

- **Using metric update function instead of MetricEventHandle**r:

MetricEventHandler requires to access net outputs and the labels to calculate metric and update the metrics which are available in the Estimator class. This EventHandler is important while dealing with multiple outputs/multiple metrics as the logic to associate outputs with metrics lies here. As an alternative we can have metric_update function to estimator which can be customized by the users.

```
- ##sample estimator class
  class Estimator:
```

```
....
...
..
@property
def metric_updatefn:
## Metric update fn should take predictions and labels as
## input and wrap the logic of how to update metrics in case
## of multi-output/ special cases.
return self._metricupdate_fn
```

- **the APIs should cover the use cases like Multi-task learning and SSD**. This can be done in 2 ways.
  - Either make the general fit API flexible enough to accommodate all the use-cases (like multi-output, multi-loss, multi-metric) - this has a disadvantage when providing a lot of flexibility to cater all use-cases will make the API overwhelming with many handlers that requires overriding MetricHandler, LossFunction which has most of the logic of mapping outputs of the network with metric and losses.
  - or should we provide custom metric and loss functions for use-cases like ObjectDetection, Multi-task learning, Neural Machine Translation which can be used off the shelf- there are already some task specific loss functions in GluonCV which do not have uniform signatures and hence we will just duplicate the APIs to fit our use case.

References

1. MXNet-Module APIs- https://mxnet.incubator.apache.org/api/python/module/module.html, https://github.com/apache/incubator-mxnet/blob/master/python/mxnet/module/base_module.py
2. Tensorflow estimators- https://ai.google/research/pubs/pub46369
3. Keras Model API- (fit, predict, evaluate)- https://keras.io/models/model/
4. https://discuss.pytorch.org/t/supermodule-for-keras-like-training-with-callbacks-constraints-and-progress-bar/2075
5. Scikit learn- https://scikit-learn.org/stable/
6. https://github.com/ecs-vlc/torchbearer#quick: Torchbearer is a framework for doing easy fit, evaluate and predict on Pytorch. It is implemented using state objects which hold states of all the callback functions. The training parameters (loss, optimizer, metric) are passed as callbacks. They use association between states and callbacks for making the fit method work. There are similar frameworks for Pytorch like skorch, PyToune, ignite, TorchNetTwo (TNT), Inferno. Skorch uses Scikit learn internally and we don't want to add additional dependencies

APPENDIX A - Gluon Training Loop Example

current user experience

```
##Current training loop in gluon
############################
#  Only one epoch
############################
num_epochs = 1

trainer = gluon.Trainer(net.collect_params(), optimizer, optimizer_params)
L = gluon.loss.SoftmaxCrossEntropyLoss()
best_val_score = 1

for epoch in range(num_epochs):
            tic = time.time()
            train_metric.reset()
            btic = time.time()

            for i, batch in enumerate(train_data):
                data, label = batch_fn(batch, ctx)


                with ag.record():
                    outputs = [net(X) for X in data]
                    loss = [L(yhat, y) for yhat, y in zip(outputs, label)]
                for l in loss:
                    l.backward()
                lr_scheduler.update(i, epoch)
                trainer.step(batch_size)

                train_metric.update(label, outputs)

                if log_interval and not (i+1)%log_interval:
                    train_metric_name, train_metric_score = train_metric.get()
                    logger.info('Epoch[%d] Batch [%d]\tSpeed: %f samples/sec\t%s=%f\tlr=%f'%(
                                epoch, i, batch_size*log_interval/(time.time()-btic),
                                train_metric_name, train_metric_score, trainer.learning_rate))
                    btic = time.time()

            train_metric_name, train_metric_score = train_metric.get()
            throughput = int(batch_size * i /(time.time() - tic))

            err_top1_val, err_top5_val = test(ctx, val_data)

            logger.info('[Epoch %d] training: %s=%f'%(epoch, train_metric_name,
train_metric_score))
            logger.info('[Epoch %d] speed: %d samples/sec\ttime cost: %f'%(epoch, throughput,
time.time()-tic))
            logger.info('[Epoch %d] validation: err-top1=%f err-top5=%f'%(epoch, err_top1_val,
err_top5_val))

            if err_top1_val < best_val_score:
                best_val_score = err_top1_val
                net.save_parameters('%s/%.4f-imagenet-%s-%d-best.params'%(save_dir,
best_val_score, model_name, epoch))
                trainer.save_states('%s/%.4f-imagenet-%s-%d-best.states'%(save_dir,
best_val_score, model_name, epoch))

            if save_frequency and save_dir and (epoch + 1) % save_frequency == 0:
                net.save_parameters('%s/imagenet-%s-%d.params'%(save_dir, model_name, epoch))
                trainer.save_states('%s/imagenet-%s-%d.states'%(save_dir, model_name, epoch))

        if save_frequency and save_dir:
            net.save_parameters('%s/imagenet-%s-%d.params'%(save_dir, model_name, opt.num_epochs-
1))
            trainer.save_states('%s/imagenet-%s-%d.states'%(save_dir, model_name, opt.num_epochs-
1))
```

Fit - Proposed implementation

```
##sample fit function

def fit(net, train_data_loader, val_data_loader, epochs, loss_fn, trainers, context):
   EventHandlers= [self.LoggingHandler, self.CheckpointHandler, self.MetricHandler]
   EventHandlers = EventHandlers + self.additionalHandlers

   for handlers in EventHandlers:
       handlers.train_begin
   while not exit condition():
        for handlers in EventHandlers:
            handlers.epoch_begin
            for each epoch:
              for handlers in EventHandlers:
                  handlers.batch_begin
              ##do a split and load for multigpu
              x,y = split_and_load(train_dataloader)
     y=net(x) ## forward pass
              calculate loss using loss_fn
              backward pass
              for handlers in EventHandlers:
                handlers.batch_end
     for handlers in EventHandlers:
                  handlers.epoch_end

     for handlers in EventHandlers:
         handlers.train_end
```

APPENDIX B - Supported Models

By supporting the following models, we believe we can cover most basic use cases for Gluon users

Domain	Category	Model	Reference	Feature Required	Note
CV	Image Classification	AlexNet	Gluon Book	net, dataloader, batch_size, trainer, ctx, num_epochs	mlp, lenet, vgg are similar, example: train_ch5()
CV	Image Augmentation + Classification	ResNet18	Gluon Book	net, dataloader, batch_size, trainer, ctx, num_epochs	example: train_ch5()
CV	Semantic Segmentation	FCN	Gluon Book	more data_transformation, multi-gpu	example: train()
CV	Object Detection	SSD	Gluon Book	multiple lables, losses, and metrics	training script from Gluon CV
NLP	Text Sentiment Classification	BiRNN	Gluon Book	same as 1 &2	example: train()
NLP	Text Sentiment classification	TextCNN	Gluon Book	same as 1 &2	example: train()
NLP	Neural Machine Translation	encoder-decoder and attention mechanism.	Gluon Book	multiple trainer, different inputs for loss	
Various	Various	LR	Kaggle Blog	LR and XGBoost is most used besides CV and NLP models	XGBoost is not in scope and not supported

APPENDIX C - Tensorflow estimators

https://www.tensorflow.org/guide/estimators
Tensorflow estimators are objects in TF which provided three methods- Train, eval and predict.
There are two options on how to use the estimators- Custom estimators and pre-defined estimators
For each of the estimators, the model and algorithm is defined model using model_fn and three methods train, evaluate and predict are defined for using those models. Train_hooks and eval_hooks are part of train/predict methods callbacks to run codes within training loop/prediction code.

Some **pre-defined estimators**: DNN, Linear Regressor, etc
An example of how to define pre-defined estimator. There are multiple options available for DNN classifies

```
# estimator using the ProximalAdagradOptimizer optimizer with
# regularization.
estimator = DNNClassifier(
    feature_columns=[categorical_feature_a_emb, categorical_feature_b_emb],
    hidden_units=[1024, 512, 256],
    optimizer=tf.train.ProximalAdagradOptimizer(
      learning_rate=0.1,
      l1_regularization_strength=0.001
    ))
```

For customizing a specific part of pre-defined estimator, we need to re-create a new estimator with the customized module and then use it.
https://towardsdatascience.com/how-to-extend-a-canned-tensorflow-estimator-to-add-more-evaluation-metrics-and-to-pass-through-ddf66cd3047d

Custom-estimators examples:

To implement a typical model function, you must do the following:

- Define the model which contains all the details including loss, optimizer, metric
- Specify additional calculations for each of the three different modes
    - Predict
    - Evaluate
    - Train

Appendix D - Keras fit api

Example usage in of Keras fit API:
https://keras.io/getting-started/sequential-model-guide/#training

```
import kerasfrom keras.models
import Sequentialfrom keras.layers
import Dense, Dropout, Activation
from keras.optimizers import SGD
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)
model = get_model()
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
```

Keras fit API is implemented using Callback (custom object) which exposes methods to be called at
(i) beginning of training
(ii) end of training
(iii) beginning of epoch
(iv) end of epoch
(v) batch_begin
(vi) batch_end
It has a list of default implementation of callbacks like History, BaseLogger, CSVLogger, ModelCheckpoint, EarlyStopping, LRScheduler, Tensorboard. It also has an option of customizable callback which user may define as required.