LEENet: Learned Early Exit Network

Learning Optimal Early Exit Policy for Efficiency Improvements in DNNs

Austin Chemelli, Anthony Wong, Blake Sanie, Dylan Mace, Dylan Small, Nic Zacharis

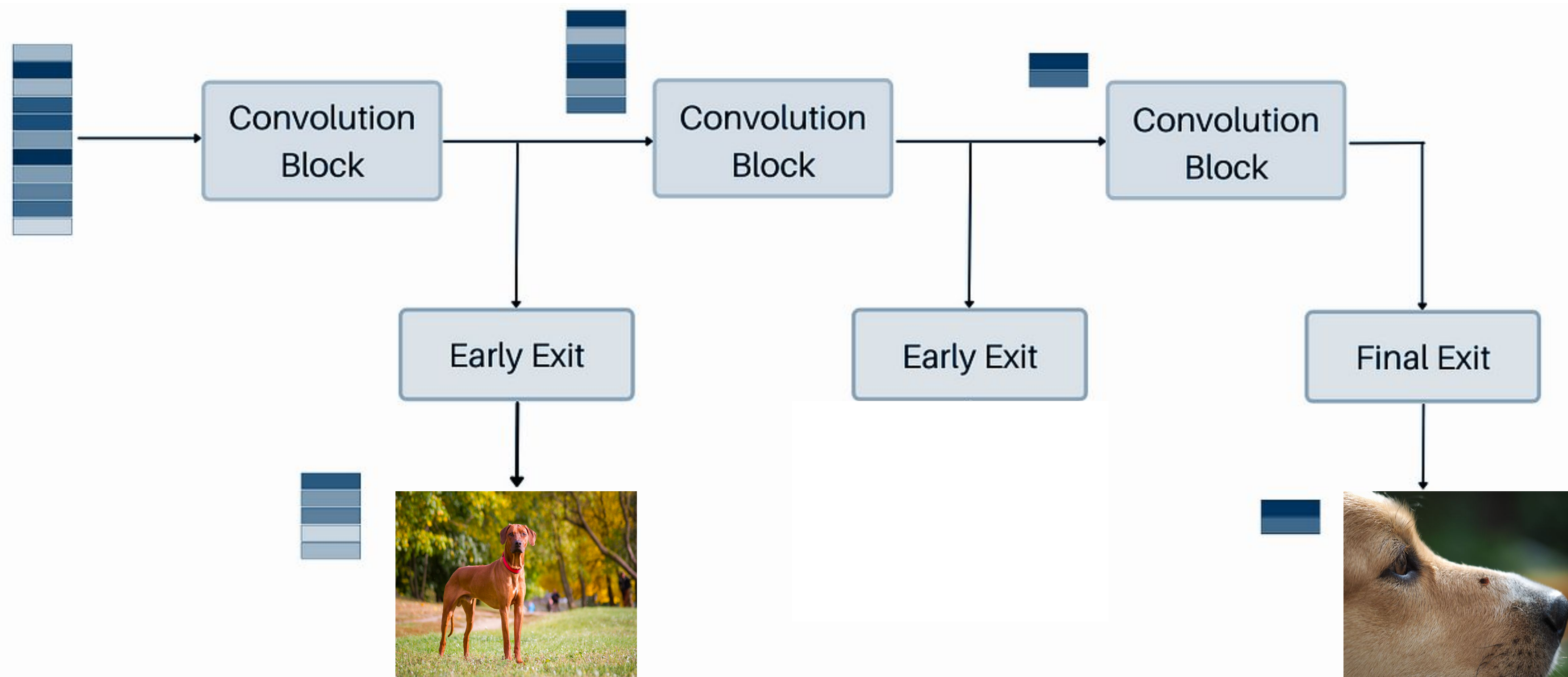
Georgia Tech

Project Information

Georgia Tech

What is Early Exit?

- Place classifiers at **multiple** locations throughout the model
- At each potential exit, a **confidence value** dictates whether to use the exit
  - If threshold met, classify image

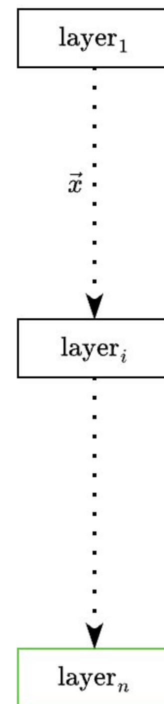


Georgia Tech

Our Approach

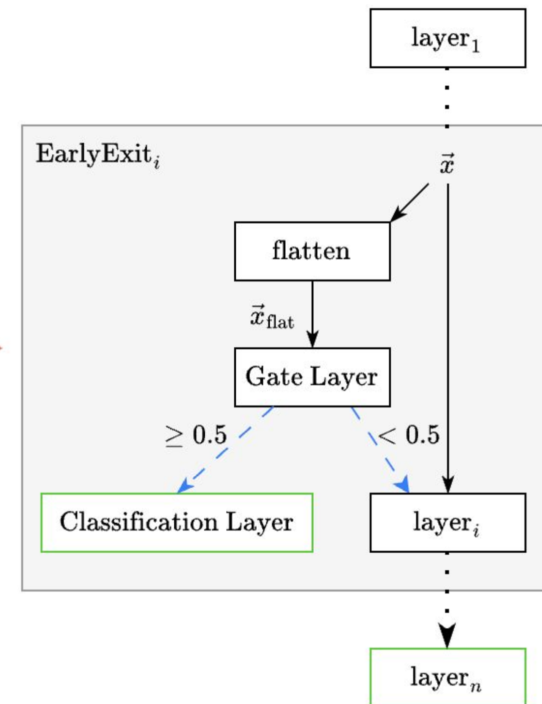
- Insert “gate layer” to decide whether to exit
- Learn gate layer parameters to optimize accuracy/cost tradeoff
- Insert hyperparameters for tuning tradeoff

Base Model



Replaced by

Extended with Early Exit



Legend

Produces $\mathbb{R}^m$ Logits

Indirect layer descendance

Direct layer descendance

Conditional branch

System Architecture
+
Pretrained Components

Open Source Components Utilized

- Model Architectures:
  - ResNet50: TorchVision
  - VGG11: TorchVision
  - DenseNet121: TorchVision
- Datasets
  - ImageNetTE: HuggingFace
    - 12,000 320x320 images, 10 classes
  - CIFAR-10: HuggingFace
    - 6,000 32x32 images, 10 classes
  - CIFAR-100 HuggingFace
    - 60,000 32x32 images, 100 classes
- Previous Early Exit Implementation Code:
  - EENet: Learning to Early Exit for Adaptive Inference (Ilhan et al.)
    - Paper: arXiv
    - Code: GitHub

Georgia Tech

System Architecture

- OptionalExitModule
  - Wrapper class for converting a pretrained layer into an exit layer
- EarlyExitModel
  - Wrapper class for converting a pretrained model into a LEENet model
- EarlyExitTrainer
  - All logic for training classifier heads and gate layers
- main.ipynb
  - Jupyter notebook for creating LEENet models / training exit classifiers
- alpha_tuning.ipynb
  - Jupyter notebook for training gate layers with varying alpha values
- exit_vis.ipynb
  - Jupyter notebook for calculating data metrics and generating test visualizations
- alpha_tuning_results.ipynb
  - Jupyter notebook for generating frontier curves for all trained model alpha values

Code Walkthrough - Model Wrapper

```
class EarlyExitModel(nn.Module):

    def __init__(self, model, num_outputs, device):
```

```
def forward(self, X):
    batch_size, *sample_shape = X.shape

    last_layer_y_hat = None

    # y_hat of layer where there are no remaining images to push forward in the model. All samples have exited
    early_exit_y_hat = None
    try:
        last_layer_y_hat = self.model(X)
    except EarlyExitException as e:
        early_exit_y_hat = e.y_hat

    if self.state == TrainingState.TRAIN_CLASSIFIER_EXIT or self.state == TrainingState.TRAIN_CLASSIFIER_FORWARD:
        if last_layer_y_hat is not None:
            # if forward pass made it to the back of the layer, get the last layer y_hat
            return last_layer_y_hat

        # if exit occured before last classifier, get y_hat where exit occured
        return early_exit_y_hat


    if self.state == TrainingState.TRAIN_EXIT:
        assert last_layer_y_hat is not None, "Forward propagation should have made it to the end of the model"

        y_hats = torch.empty((batch_size, len(self.exit_modules) + 1, self.num_outputs), device=self.device)
        exit_confidences = torch.empty((batch_size, len(self.exit_modules)), device=self.device)

        for i, exit_module in enumerate(self.exit_modules):
            y_hats[:, i] = exit_module.y_hat
            exit_confidences[:, i] = exit_module.exit_confidences

        y_hats[:, -1] = last_layer_y_hat

        return y_hats, exit_confidences
```

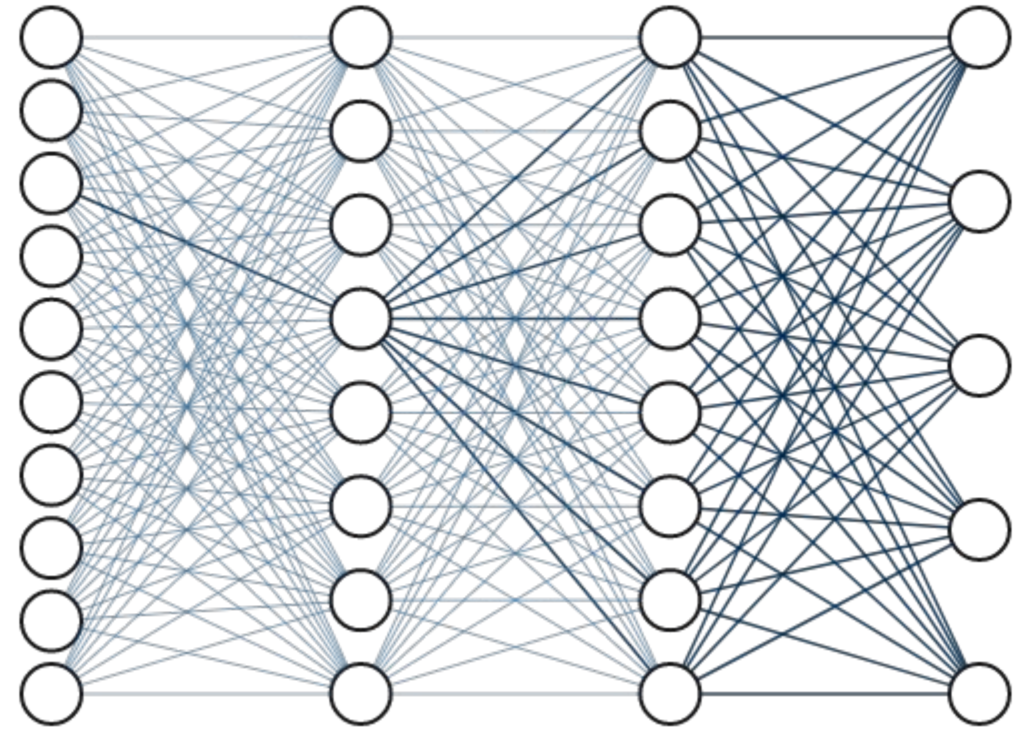
```
    if self.state == TrainingState.INFER:
        if torch.cuda.is_available() and len(self.exit_modules) > 0:
            for stream in [module.stream for module in self.exit_modules if module.stream is not None]:
                stream.synchronize() # wait for all multithreaded classifiers to finish

        y_hat = torch.empty((batch_size, self.num_outputs), device=self.device)

        remaining_idx = torch.arange(batch_size, device=self.device)

        self.num_exits_per_module = []

        for exit_module in self.exit_modules:
            if len(remaining_idx) == 0:
                self.num_exits_per_module.append(0)
                continue

            self.num_exits_per_module.append(exit_module.exit_taken_idx.sum().item())

            # use indices of exits taken in the model's (reduced) batched to obtain the translated original index within the original batch
            original_idx = remaining_idx[exit_module.exit_taken_idx]
            if len(original_idx) == 0: continue
            y_hat[original_idx] = exit_module.y_hat

            # mirroring how the batch is reduced by the exit module, reduce index look up array the same way
            remaining_idx = remaining_idx[~exit_module.exit_taken_idx]

        # if even after going through each early exit layer, there are samples that did not exit, grab the y_hat from terminal classifier
        if len(remaining_idx) > 0:
            y_hat[remaining_idx] = last_layer_y_hat
            self.num_exits_per_module.append(len(remaining_idx))
        else:
            self.num_exits_per_module.append(0)

        return y_hat
```

Training Process

Georgia Tech

Training Process

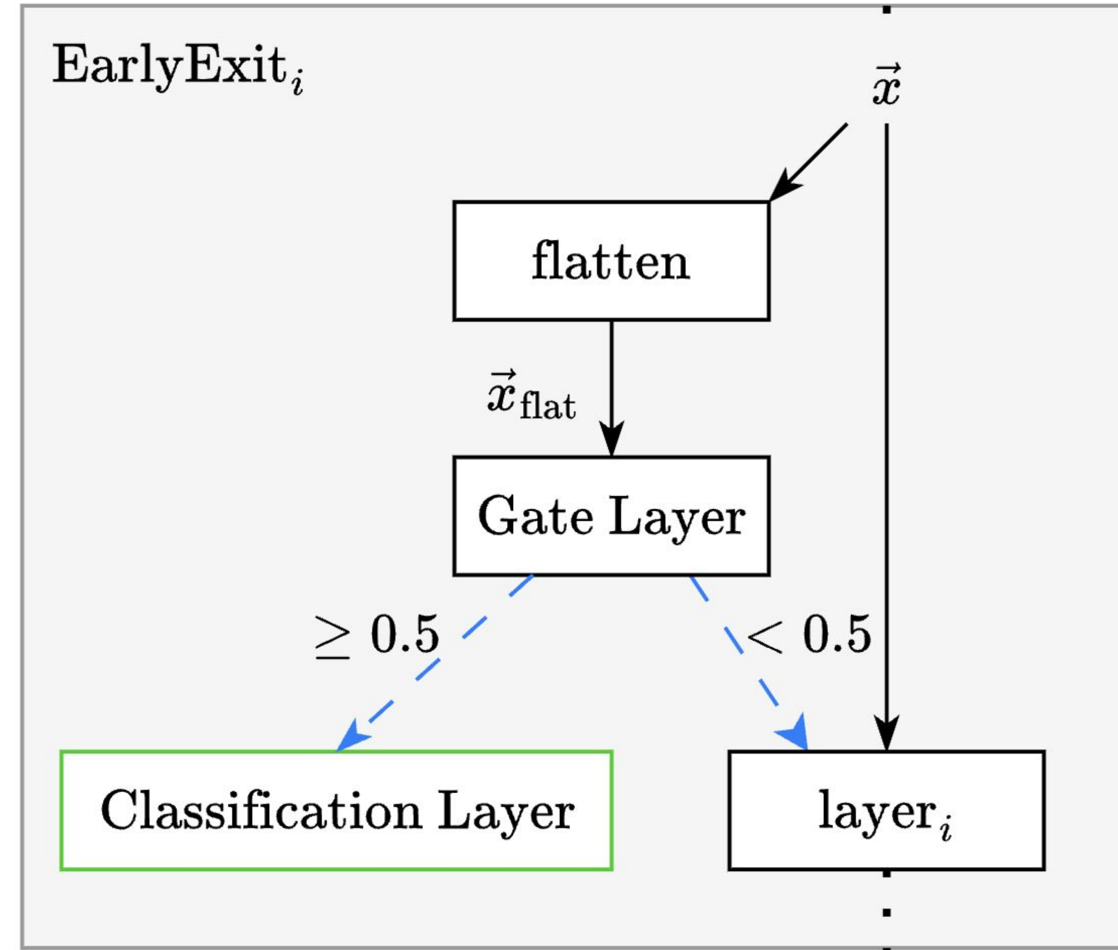
- Train each early exit classifier
  - These can be trained individually since they are unrelated
  - **Loss**: Categorical Cross Entropy
  - **Data Ratio:** 80/20 Train/Test
- Train final classification layer
  - This is just transfer learning onto your dataset
  - **Loss:** Categorical Cross Entropy
  - **Data Ratio:** 80/20 Train/Test
- Train gate layers
  - These have to be trained at the same time
  - **Loss**: Custom Loss Function (next slides)
  - **Data Ratio:** 80/20 Train/Test



Georgia Tech

OptionalExitModule Forward Pass

- Input gets flattened
  - Output Size: (n,)
- Dot product with gate layer parameters
  - Output Size: (1,)
- Decide whether to exit
  - Gate layer output above threshold
- If exit, classify n input to logits
  - Output Size: (n_classes,)
- If not exit, feed through original network



Georgia Tech

Code Walkthrough - Early Exit Module

```python
class OptionalExitModule(nn.Module):

    def __init__(self, module, num_outputs):
```

```python
def forward(self, X):

    # Check the device of input tensor X and move necessary components to the same device
    self.current_device = X.device

    X_flat = torch.flatten(X, start_dim=1).to(self.current_device)
    _, flat_size = X_flat.shape

    # Create exit gate and classifier at runtime to adapt to module input size
    if self.exit_gate is None:
        self.exit_gate = nn.Linear(flat_size, 1).to(self.current_device)
    if self.classifier is None:
        self.classifier = nn.Linear(flat_size, self.num_outputs).to(self.current_device)

    if self.state == TrainingState.TRAIN_CLASSIFIER_EXIT:
        return self.forward_train_classifier_exit(X, X_flat)
    elif self.state == TrainingState.TRAIN_CLASSIFIER_FORWARD:
        return self.forward_train_classifier_forward(X, X_flat)
    elif self.state == TrainingState.TRAIN_EXIT:
        return self.forward_train_exit(X, X_flat)
    if self.state == TrainingState.INFER:
        return self.forward_infer(X, X_flat)
```

Georgia Tech

Code Walkthrough - Classifier Training

```python
# MARK: - Training Classifiers
def train_classifier_epoch(self, train_loader, epoch, validation_loader=None):
    self.model.train()
    net_loss = 0.0
    net_accuracy = 0.0

    validation_loss = 0.0
    validation_accuracy = 0.0

    self.progress_bar = tqdm(train_loader, desc=f'Epoch {epoch}', ncols=100, leave=False)

    for i, (X, y) in enumerate(self.progress_bar):
        X = X.to(self.device)
        y = y.to(self.device)
        y_hat = self.model(X)

        trainable_params = filter(lambda p: p.requires_grad, self.model.parameters())
        optimizer = torch.optim.Adam(trainable_params, lr=0.0001)

        optimizer.zero_grad()

        loss = self.classifier_loss_function(y_hat, y)
        accuracy = self.calculate_accuracy(y_hat, y)

        net_loss += loss.item()
        net_accuracy += accuracy

        loss.backward()
        optimizer.step()

        # Update and display the progress bar at the end of each epoch
        self.progress_bar.set_postfix({"Loss": loss.item(), "Accuracy": accuracy})
```

Code Walkthrough - Gate Training

```
# MARK: - Training Exits
def train_exit_epoch(self, train_loader, lr, epoch, validation_loader=None):
    self.model.train()

    net_loss = 0.0
    validation_accuracy = 0.0
    validation_time = 0.0

    self.progress_bar = tqdm(train_loader, desc=f'Epoch {epoch}', ncols=100, leave=False)

    for i, (X, y) in enumerate(self.progress_bar):
        X = X.to(self.device)
        y = y.to(self.device)
        y_hats, exit_confidences = self.model(X)

        trainable_params = None
        # concatenate all trainable parameters as gate layers
        for exit_layer in self.model.exit_modules:
            if trainable_params is None:
                trainable_params = list(filter(lambda p: p.requires_grad, exit_layer.exit_gate.parameters()))
            else:
                trainable_params += list(filter(lambda p: p.requires_grad, exit_layer.exit_gate.parameters()))

        optimizer = torch.optim.Adam(trainable_params, lr=lr)

        optimizer.zero_grad()

        loss, ce_part, cost_part = self.gate_loss_function(y, y_hats, exit_confidences, self.model.costs)

        net_loss += loss.item()

        loss.backward()
        optimizer.step()

        # Update and display the progress bar at the end of each epoch
        self.progress_bar.set_postfix({"Loss": loss.item()})

        # Optionally, calculate validation metrics
        if validation_loader is not None and i % (len(train_loader) // 10) == 0:
            validation_accuracy, validation_time, exit_idx = self.validate_exit_gates(validation_loader)
```

Georgia Tech

Custom Gate Loss Function (Batch Size *n*)

$$\text{loss} = \frac{1-\alpha}{n}\left[\sum_{(X,y,\hat{y},g)}\left(\sum_{i=0}^{|g|}\left(\text{CE}(y,\hat{y}_i)\cdot g_i\prod_{j=0}^{i-1}\bar{g}_j\right)\right)\right] + \left[\sum_{(X,y,\hat{y},g)}\left(\sum_{i=0}^{|g|}\left(\text{costs}_i\cdot g_i\prod_{j=0}^{i-1}\bar{g}_j\right) + \text{costs}_{|g|}\prod_{j=0}^{|g|}\bar{g}_j\right)\right]\frac{\alpha}{n}$$

Minimize CE Loss (Maximize Accuracy)

Minimize Computational Cost (Maximize Efficiency)

Control Weighting of Accuracy + Cost

Maximizing Accuracy

$$\left[\sum_{(X,y,\hat{y},g)}\left(\sum_{i=0}^{|g|}\left(\mathrm{CE}(y,\hat{y}_i)\cdot g_i\prod_{j=0}^{i-1}\bar{g}_j\right)\right)\right]$$

- Iterate over all images in the batch
- Calculate product summed over all gates:
  - Cross Entropy loss of:
    - **y** : true classification
    - $\hat{\mathbf{y}}_i$: predicted classification logits from gate i
  - Probability of exiting at gate i:
    - $\mathbf{g}_i$: exit confidence for gate i (larger means more confident)
    - $\bar{\mathbf{g}}_j$: forward confidence for gate j (equal to $1\text{-}g_i$)

Minimize this term for higher net model accuracy

Georgia Tech

Maximizing Efficiency

$$\left[\sum_{(X,y,\hat{y},g)}\left(\sum_{i=0}^{|g|}\left(\text{costs}_i \cdot g_i \prod_{j=0}^{i-1}\bar{g}_j\right) + \text{costs}_{|g|}\prod_{j=0}^{|g|}\bar{g}_j\right)\right]$$

- Iterate over all images in the batch
- Calculate product summed over all gates:
  - Cost of exiting at gate i:
    - Derived as **% parameters utilized after the first exit**
    - Value ranges from **[0, 1]**
    - Normalized so **c[0] = 0 and c[-1] = 1**
  - Probability of exiting at gate i:
    - $\mathbf{g_i}$: exit confidence for gate i (larger means more confident)
    - $\mathbf{\bar{g}_i}$: forward confidence for gate i (equal to $1-g_i$)
-
- **Minimize this term for lower net inference cost**

Georgia Tech

Controlling Accuracy/Cost Tradeoff

$$\text{loss} = \frac{1-\alpha}{n}\left[_\text{acc}_\right] + \left[_\text{cost}_\right]\frac{\alpha}{n}$$

- **Alpha** weights both subequations of our loss function
  - In the range [0, 1]
- Changing alpha allows for the loss function to tailor model behavior
  - **α = 0** : weight loss for accuracy only
  - **α = 1** : weight loss for computational efficiency only
  - **α > 0, α < 1** : weight both accuracy and computational cost
- **Optimal alpha depends** on the underlying dataset
  - Different datasets have varying tradeoffs between accuracy and efficiency
- Alpha increases the interpretability of our methodology
  - Single hyperparameter that easily allows for emphasizing model behaviors

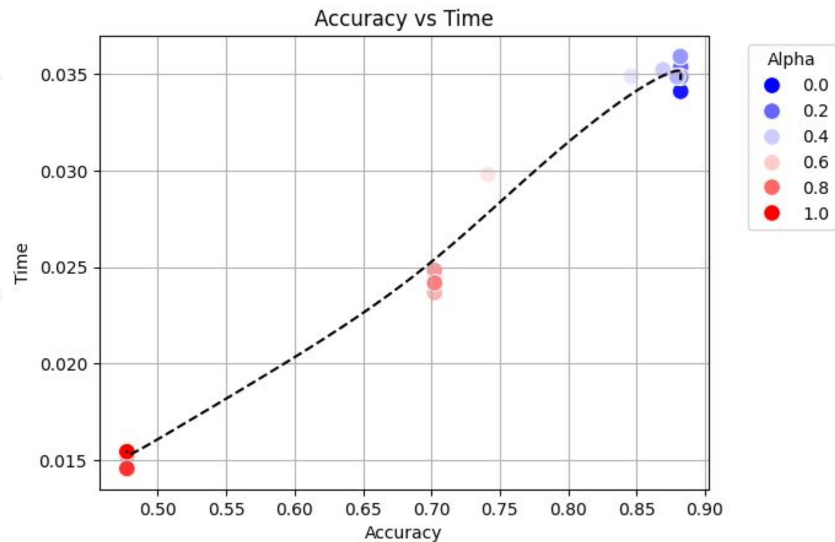
Project Results

(across varying alpha values)

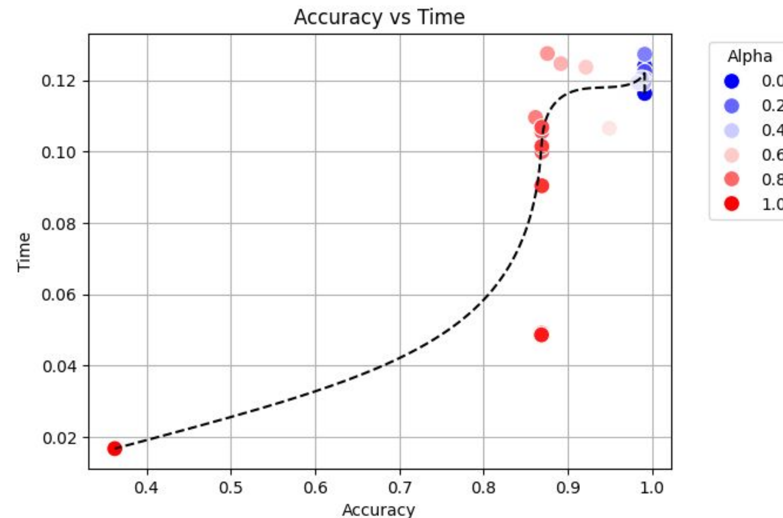
Georgia Tech

Results (Frontier Curves for All Models)

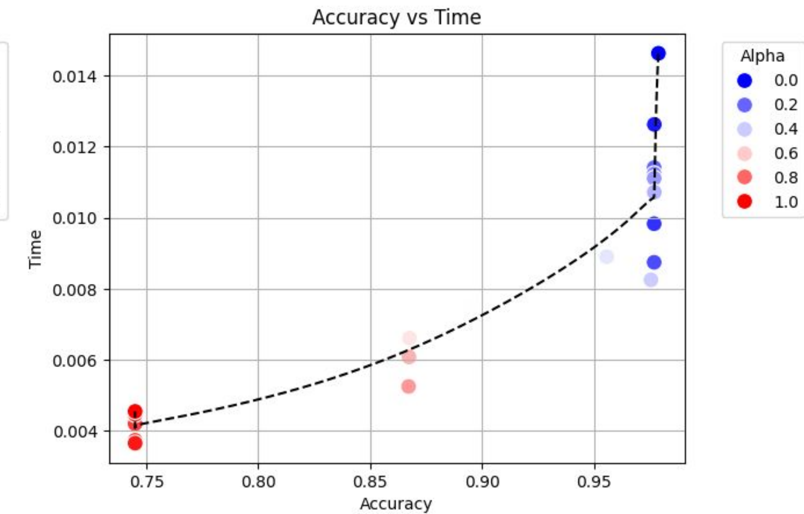
DenseNet121/CIFAR-100



ResNet50/ImageNetTE



VGG11/CIFAR-10



- In the above curves, we can see a general trend of **as accuracy increases, computational cost (time) also increases**.
  - Furthermore, we can observe that **alpha tends to decrease as accuracy and time increase**. This corresponds with with our loss function.
- Another interesting observation is the subtle difference in the accuracy vs time trend lines for each model.
  - DenseNet121 has more of a **linear** shape.
  - ResNet50 **slowly increases and then sharply increases** as alpha begins to decrease , it is also interesting to note most alpha values are clustered in the upper right quadrant of the graph.
  - VGG11 has a **somewhat linear shape**; however **once alpha is below 0.5**, most of the **accuracy stays about the same yet time increases**.

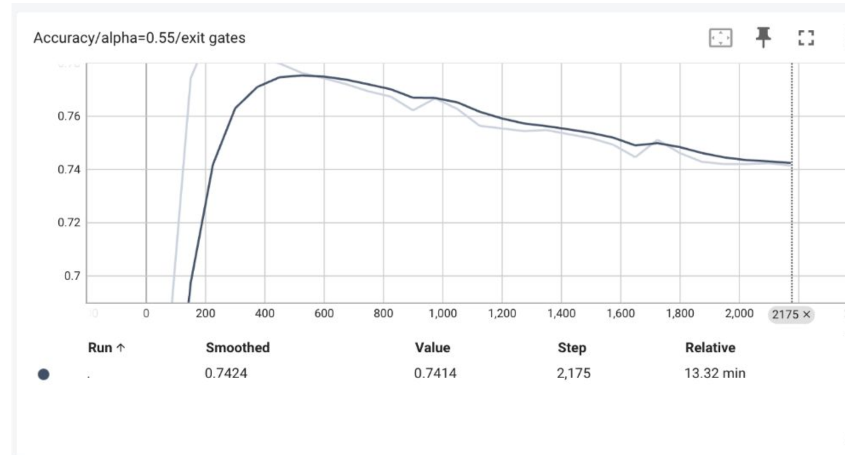
Georgia Tech

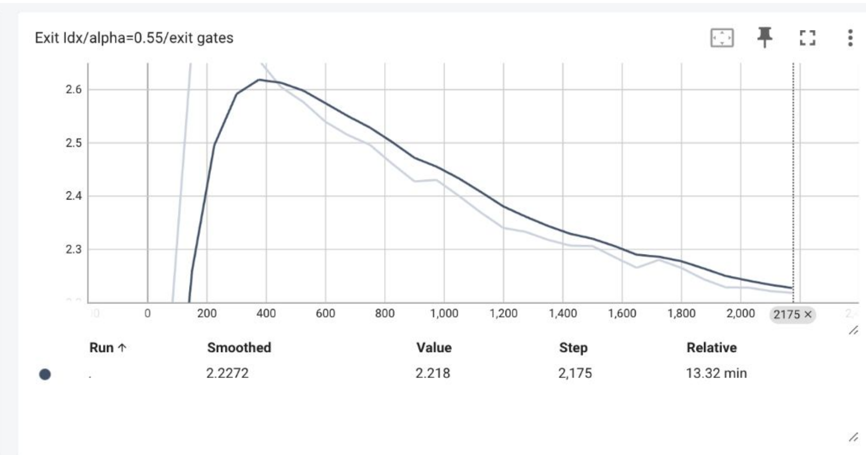
Results (DenseNet121/CIFAR-100, α=0.55)

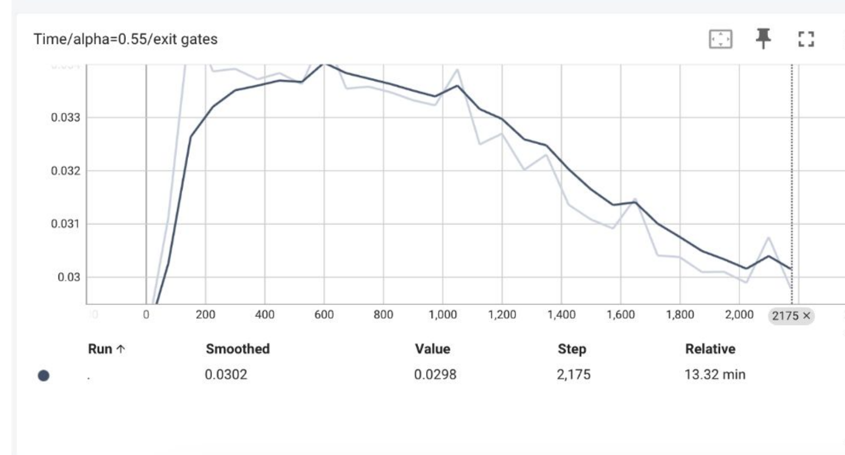
1.47x
Inference Speedup

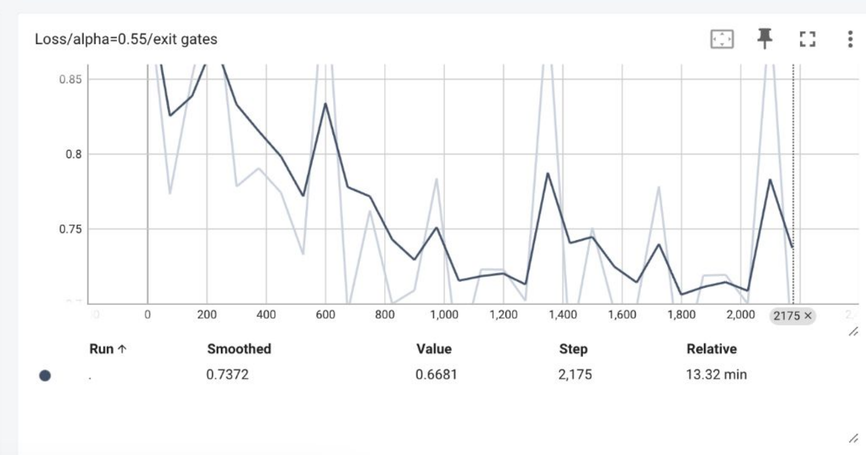
38%
Cost Reduction

10%
Accuracy Reduction









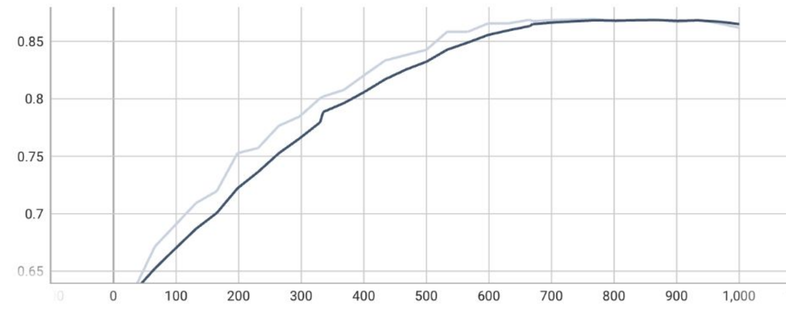
Georgia Tech

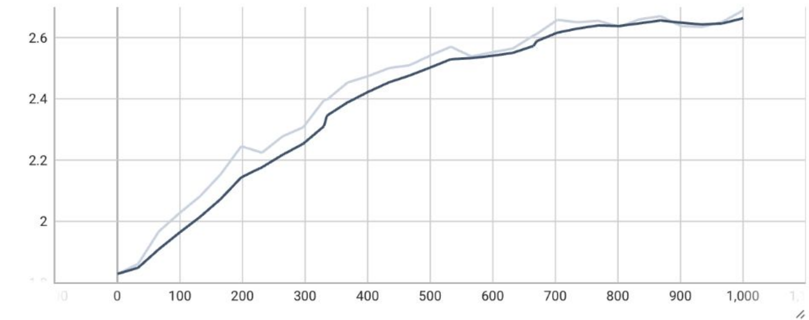
Results (ResNet50/ImageNetTE, α=0.75)

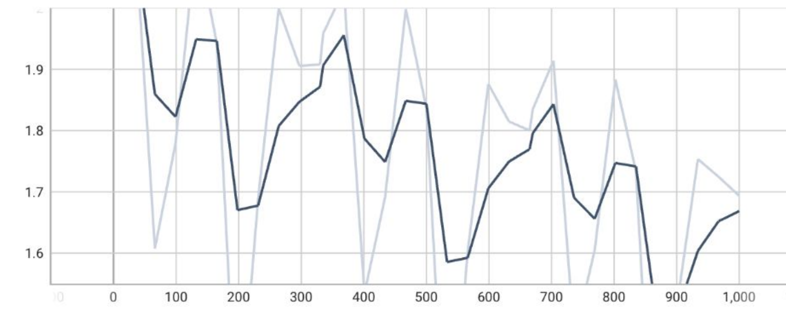
1.43x Inference Speedup

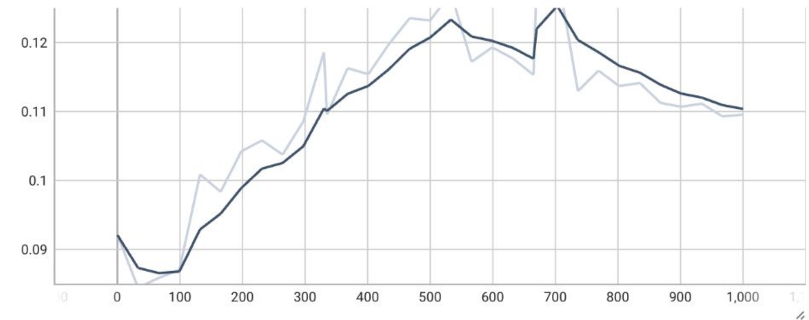
69% Cost Reduction

12% Accuracy Reduction









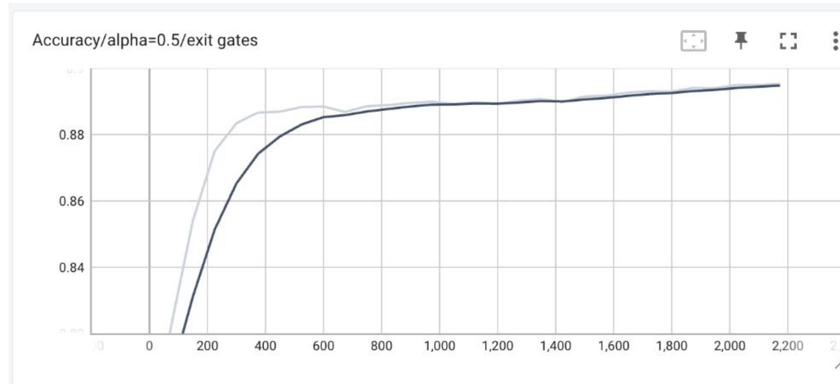
Georgia Tech

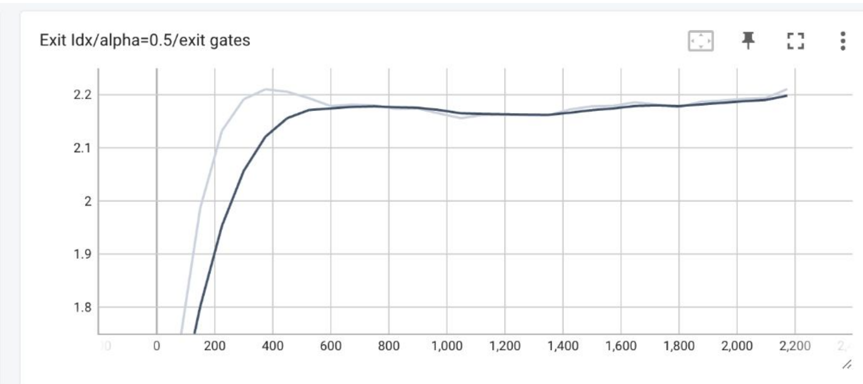
Results (VGG11/CIFAR-10, α=0.5)

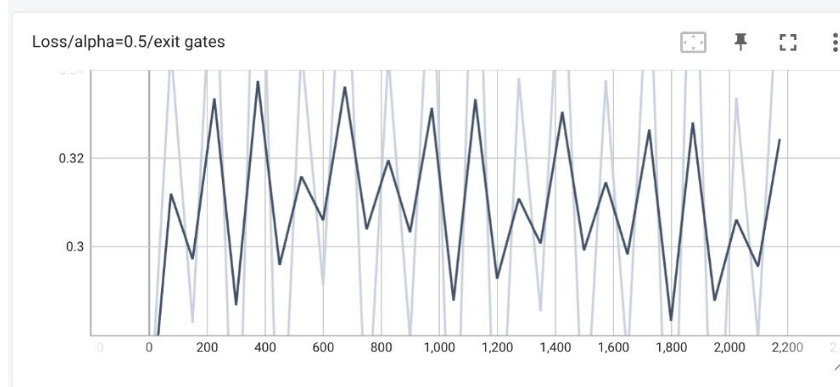
1.08x Inference Speedup

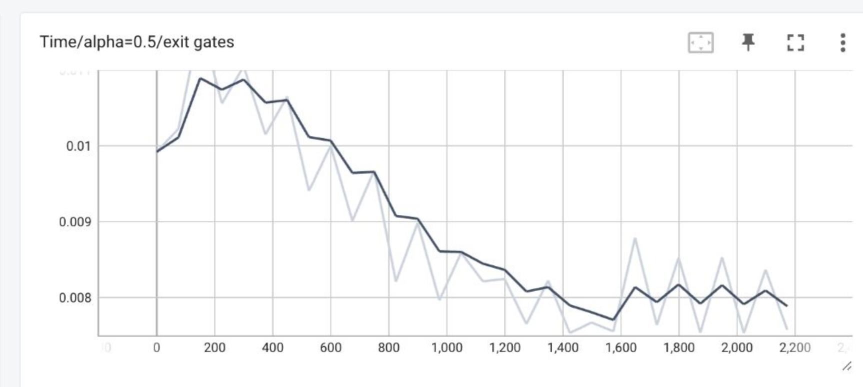
51% Cost Reduction

8% Accuracy Reduction



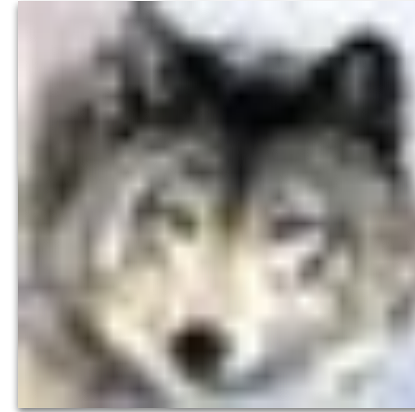






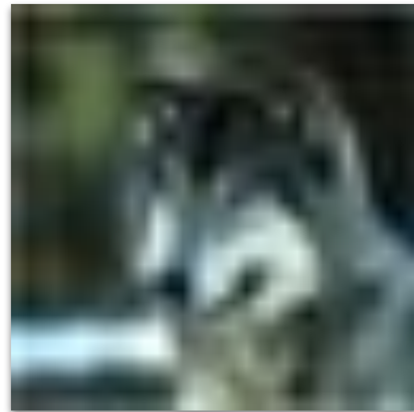
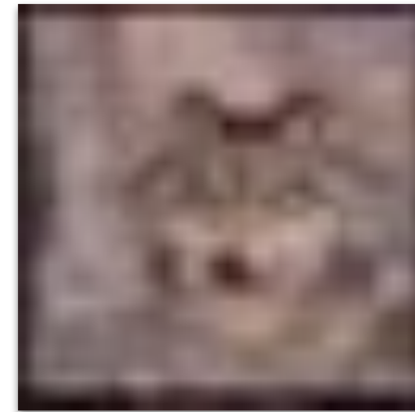
Georgia Tech

Correct Examples: Wolf (VGG11/CIFAR100, α=0.6)



Exit 2 of 5

Consistent Face Scale and Position

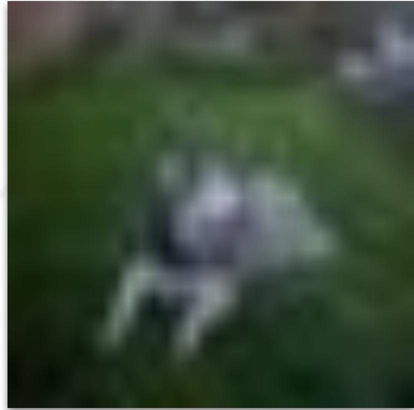



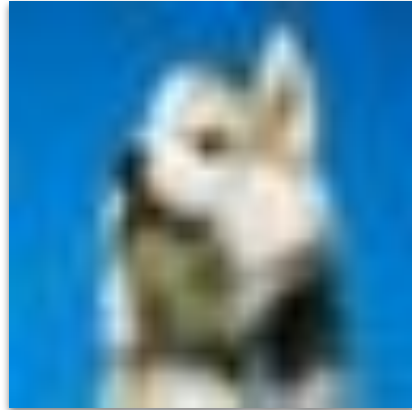
Exit 3 of 5

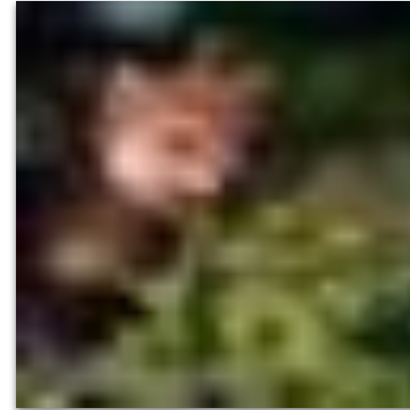
Discoloration, scale and position variation

Georgia Tech

Incorrect Examples: Wolf (VGG11/CIFAR100, α=0.6)


Fox


Seal

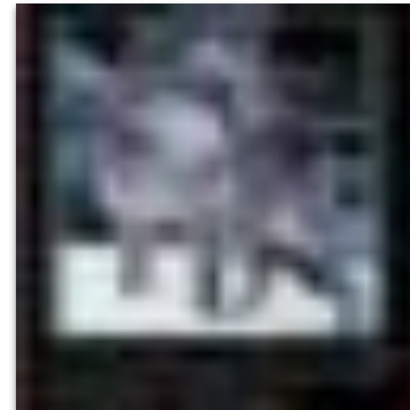

Fox

Exit 2 of 5

Subject/background interference

Rabbit

Flatfish

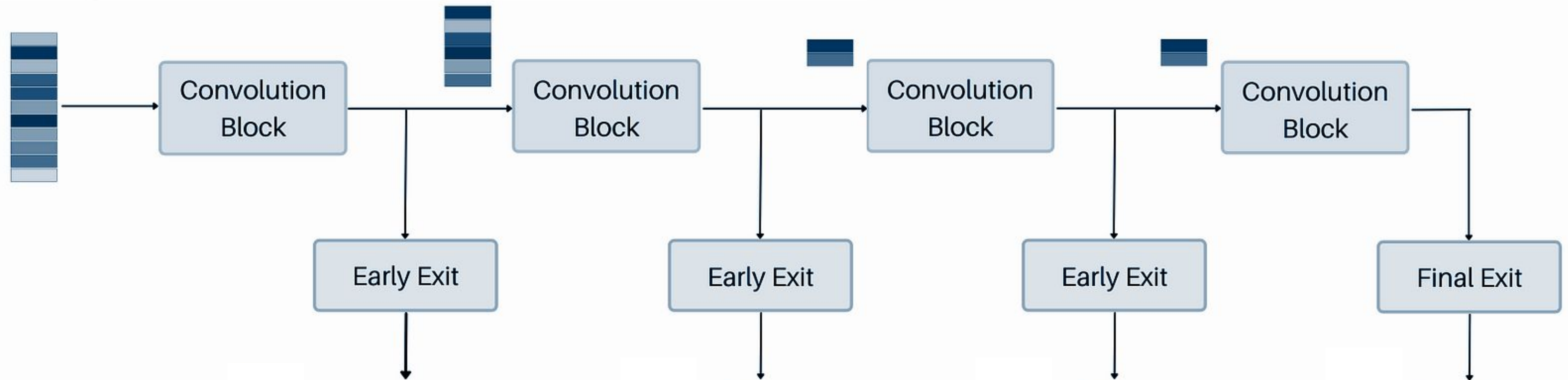

Turtle

Exit 3 of 5

Misleading subject representation

Georgia Tech

DenseNet121/CIFAR-100 Sunflower Predictions (α=0.55)



Sunflower Exit Distribution:

Exit 1: 0% of Images

Exit 2: 69.67% of Images
- *78.9% Correct*
- *21.1% Incorrect*

Exit 3: 24.67% of Images
- *85.1% Correct*
- *14.9% Incorrect*

Exit 4: 5.67% of Images
- *94.1% Correct*
- *5.9% Incorrect*

Georgia Tech

Next Steps

- **Code cleaning + implementation improvements**
  - Parallel processing of classification heads
- **Improve time savings**
  - Changing gate and classifier architectures
- **Allow users to query for max inference budget**
  - Binary search alpha terms to find most accurate model meeting budget constraint
- Train on a large set of different datasets/ML models
  - AlexNet, MSDNet, LLMs(?), ...
- Continue to do comparison studies with previous work

Georgia Tech

Max Inference Budget Optimization (Coming Soon)

- Mirror approach by EENet (Ilhan et al.)
- Convert α to latent variable
- Treat time as the dominant hyperparameter
  - User requests a model that is faster than without Early Exit (ex: 2.0x, 1.5x, ...)
  - User requests a model that on average runs within time budget (ex: 5ms, 3ms, ...)
- Modified model training process
  - Perform binary search over α
  - Begin with $\alpha = 0.5$
  - If speedup is too low, increase alpha -- boosts efficiency, compromises accuracy
  - If speedup is too high, reduce alpha -- boosts accuracy, compromises efficiency

Georgia Tech

Gate Input Dimensionality Reduction (Coming Soon)

- Currently, exit gate infers over all (flattened) module inputs
  - Input size can grow exponentially as model layers accumulate
  - VGG11 first gate layer - over 8k inputs
    - Computational overhead, missed efficiency opportunity
    - Poor decision generalization
- Re-architect exit gate structure
  - Efficiently and uniformly collapse inputs into smaller size
  - Before gate linear layer
    - Avg. Pool / Max Pool
      - Marginalize across channels
      - Perform binning within each channel
      - Combine both
    - Dropout
      - Randomly accept/reject inputs

Parallel Computing Optimization (Coming Soon)

- Early exits operate at batch level
  - Some samples may exit, some continue forward
- Branch formed in computation graph
  - Divergent branches can be computed in parallel
- Implementation considerations
  - Pytorch natively allows for distributed load over multiple GPUs (Cuda)
    - Sequential processing on single GPU
  - Delegating Cuda tasks over Python threads (GIL) causes system errors
    - Tensors do not serialize/deserialize across threads