Dynamic 3D CNN Pruning: Joint Frame, Channel, and Feature Adaptation for Energy Efficiency on the Edge

Supplementary Material

(a) Nunchucks (367 ms) (b) Javelin Throw (272 ms) (c) Hammering (299 ms)

(d) Boxing Speed Bag (253 ms) (e) Body Weight Squats (277 ms) (f) Apply Eye Makeup (275 ms)

Figure 11. Six sample frames from the UCF101 dataset. These frames show the time it takes to do activities using C3D with adaptive pruning on an edge CPU. The adaptive mechanism makes things faster than the time of 547 ms.

Figure 11 shows six sample frames from the UCF101 activity recognition dataset. It also shows the time it takes to do activities on a Snapdragon CPU. The pruning mechanism makes the C3D model work about two times faster than the model at the average time of the original model. The time it takes to do activities is different. It ranges from 253 ms to 367 ms. This shows that the time it takes to do activities depends on the activity.

8. Latency Profiling on Edge CPU

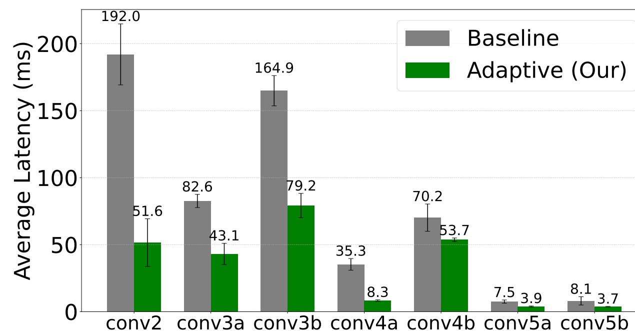


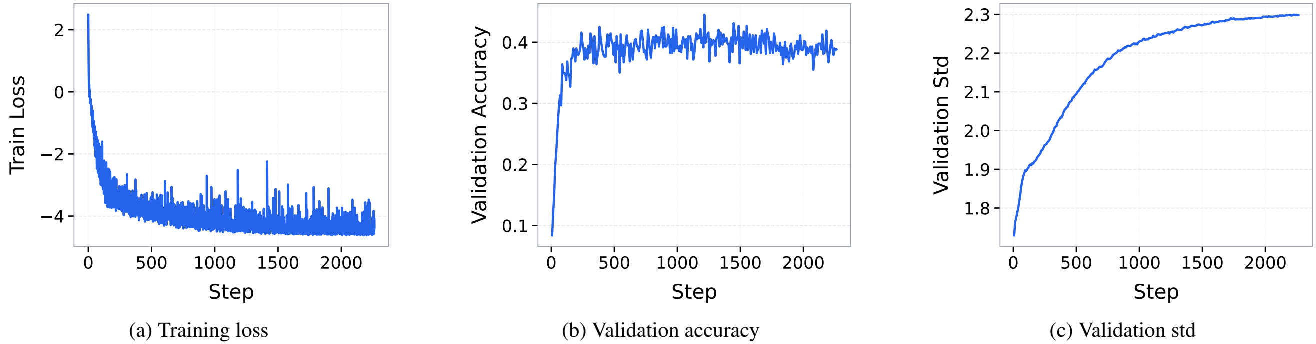
Figure 12. The time it takes to do different activities using C3D on 10 UCF101 samples with and without adaptive pruning.

Fig. 12 shows the time it takes to do different activities using the C3D model on 10 randomly chosen videos from the testing set. It shows the time it takes to do activities with and without pruning. The first convolutional layers take the time because they have more pixels in the feature map. Specifically the conv2 and conv3b layers take the time in the unpruned model at 191.99 ms and 164.89 ms respectively. The dynamic pruning mechanism helps by removing feature vectors, which

makes things faster. The notable speedups are in the conv2, conv3b and conv4a layers.

9. Training log convergence

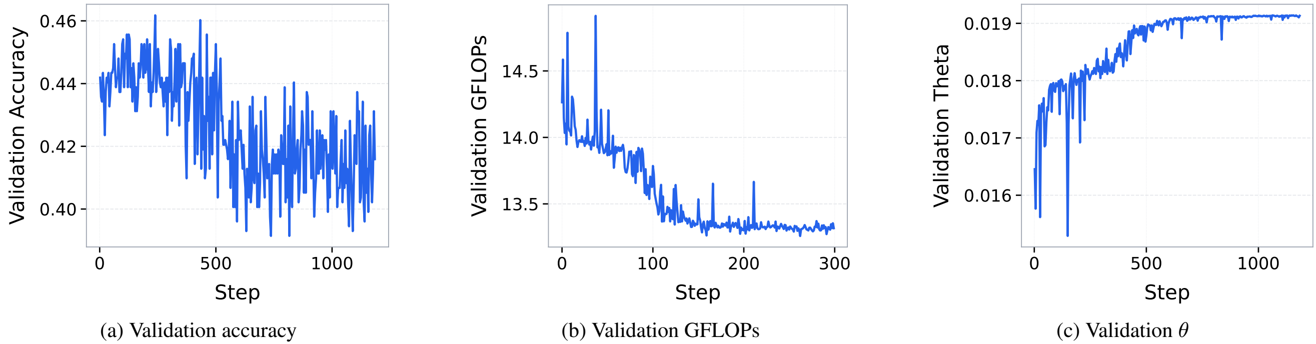
9.1. AVA Training Convergence



(a) Training loss (b) Validation accuracy (c) Validation std

Figure 13. Training convergence of the C3D model on the HMDB dataset under the AVA pruning strategy. The training loss steadily decreases while validation accuracy improves, indicating stable convergence. The validation standard deviation reflects the consistency of predictions across evaluation steps.

9.2. AAP Training Convergence



(a) Validation accuracy (b) Validation GFLOPs (c) Validation θ

Figure 14. Training convergence of the C3D model on the HMDB dataset under the AAP pruning strategy. Validation accuracy remains stable while GFLOPs decrease, demonstrating that AAP successfully reduces computational cost with minimal accuracy degradation. The pruning threshold θ converges to a stable value, reflecting a learned efficiency–accuracy trade-off.

10. Reproducibility

Data Preprocessing and Augmentation

For both the UCF101 and HMDB51 datasets we make input volumes by taking 16 frames with a temporal stride of 4. We resize the frames to 128 x 171 or 112 x 200 for C3D and then center-crop them to 112 x 112. We use normalization and augmentation strategies for different models:

(i) **C3D:** We scale the pixel intensities to the range 0 to 255.

(ii) **R(2+1)D:** We use augmentation during training including random horizontal flips and color jittering. We normalize the inputs using the ImageNet mean and standard deviation.

During inference we do not use augmentation. We use a deterministic center crop to make sure the latency measurements are consistent.

Training and Optimization Strategy

We tune the R(2+1)D architecture for 300 epochs using the Adam optimizer with a learning rate of 1 x 10^{-4}. We initialize the model with weights pretrained on Kinetics-400. Use a custom weight-mapping function to make sure the weights are compatible with our modified adaptive pruning architecture.

We do our training on eight Graphics Processing Units or GPUs for short, using something called Distributed Data Parallel or DDP with a batch of 128. This means each GPU gets 16 samples to work with. We also use something called 16-bit precision or FP16 to get things done faster. To keep everything when we move our model to a new place, we use a trick called partial freezing. This means we freeze some parts of our model, like the backbone layers, and only let the decision head or decision learn things. We do this using weights from a stage around epoch 258. When we want to see how well our model is doing, we do this on one main GPU to avoid doing the same work twice and to make sure our results are always the same.

Adaptive Pruning Modules

We have two modules in our model that can learn and change. These are called AdaptorTemporal and AdaptorSpatial.

```

import torch
import torch.nn as nn
import torch.nn.functional as F

class AdaptorTemporal(nn.Module):
    def __init__(self, T):
        super().__init__()
        # We have a learnable bias for time: [1, 1, T, 1, 1]
        self.bias_t = nn.Parameter(torch.ones(1, 1, T, 1, 1))

    def forward(self, x):
        # First we apply a time scale and use ReLU
        x = F.relu(x * self.bias_t)
        # Then we do a normalization for time
        t_mean = x.mean(dim=(1, 3, 4))
        t_soft = t_mean / t_mean.sum(dim=-1, keepdim=True)
        # Next we calculate the deviation for time
        std_t = t_soft.std(dim=-1) / 3
        return x, std_t

class AdaptorSpatial(nn.Module):
    def __init__(self, C, H, W):
        super().__init__()
        self.bias_ch = nn.Parameter(torch.ones(1, C, 1, 1, 1))
        self.bias_sp = nn.Parameter(torch.ones(1, 1, 1, H, W))

    def forward(self, x):
        x = F.relu(x * self.bias_ch)
        ch_mean = x.mean(dim=(3, 4)).flatten(1)
        ch_soft = ch_mean / ch_mean.sum(dim=-1, keepdim=True)
        std_ch = (2 * ch_soft.std(dim=-1)) / 3
        x = F.relu(x * self.bias_sp)
        sp_mean = x.mean(dim=1).flatten(1)
        sp_soft = sp_mean / sp_mean.sum(dim=-1, keepdim=True)
        std_sp = sp_soft.std(dim=-1) / 3
        return x, std_ch, std_sp

```