Project Report - ECE 176

S.VISWA

Department of Mechanical and Aerospace Engineering

A69032449

Image Classification using ConvNeXt

Abstract:

This project report explores the application of ConvNeXt, a modernized convolutional neural network (CNN) architecture inspired by Vision Transformers (ViTs), for image classification tasks. ConvNeXt builds upon traditional CNNs by incorporating design elements from Transformers, such as Layer Normalization, depth-wise convolutions, and large kernel sizes, to achieve state-of-the-art performance on benchmarks like ImageNet. The report details the architecture, strengths, and weaknesses of ConvNeXt, and presents experimental results using various optimizers and model variants on the CIFAR-100 dataset. The findings suggest that the ConvNeXtV2 Tiny model with the SGD-Nesterov optimizer offers a balance between performance and computational efficiency, achieving an accuracy of 34.10%. The report also highlights potential improvements, such as data augmentation and learning rate adjustments, to further enhance model performance.

Introduction:

Image classification using deep learning is a powerful technique where neural networks automatically learn to categorize images into different classes. Convolutional Neural Networks (CNNs) are widely used for this task, as they can effectively extract hierarchical features from images. Advanced architectures like ResNet, EfficientNet, and Vision Transformers (ViTs) further enhance classification accuracy by addressing issues like vanishing gradients and capturing global dependencies. In this project we dive into one such architecture, ConvNeXt is a deep learning model designed for image classification, introduced as a convolutional alternative to ViT. It builds upon traditional CNN while incorporating design elements inspired by transformer architectures, such as Layer Normalization, depth-wise convolutions, and large kernel sizes. It maintains the efficiency of CNNs while rivaling the performance of ViTs, making it a strong choice for various computer vision tasks.

The Architecture:

ConvNeXt [1] is a modernized version of a traditional convolutional neural network (ConvNet) that incorporates design principles inspired by ViTs. The architecture is built by gradually modernizing a standard ResNet to resemble a hierarchical vision Transformer like Swin Transformer.

The authors first apply modern training techniques (e.g., AdamW optimizer, data augmentation like Mixup, Cut Mix, RandAugment, and regularization techniques like Stochastic Depth and Label Smoothing) to a standard ResNet-50.This improves the baseline ResNet-50 accuracy from 76.1% to 78.8%, showing that a significant portion of the performance gap between ConvNets and Transformers can be attributed to training techniques.

Some key changes include:

The authors adjust the number of blocks in each stage of the ResNet to match the compute ratio of Swin Transformers. For example, they change the ResNet-50 block distribution from (3, 4, 6, 3) to (3, 3, 9, 3).The traditional 7x7 convolution with stride 2 in ResNet is replaced with a 4x4 non-overlapping convolution (similar to the patchify layer in ViT), which downsamples the input more aggressively.These changes improve the model's accuracy to 79.5%The authors adopt the ResNeXt design, which uses grouped convolutions. Specifically, they use depthwise convolutions (a special case of grouped convolutions) to reduce FLOPs and increase the network width to match SwinT's channel count (from 64 to 96),This modification increases the accuracy to 80.5% while increasing FLOPs to 5.3G.Inspired by Transformer blocks, the authors introduce an inverted bottleneck design, where the hidden dimension of the MLP block is four times wider than the input dimension. This is similar to the design used in MobileNetV2.This change reduces FLOPs to 4.6G and slightly improves accuracy to 80.6%.ReLU is replaced with GELU, which is commonly used in Transformers.The number of activation functions is reduced to match the design of Transformer blocks, where only one activation function is used per block.The number of BatchNorm layers is reduced, and BatchNorm is replaced with LayerNorm, like in Transformers. Downsampling is performed using separate 2x2 convolution layers with stride 2, similar to Swin Transformers.These micro-level changes further improve the accuracy to 82.0%, surpassing SwinT's performance.The final ConvNeXt model is a pure ConvNet that incorporates all the above modifications. It achieves state-of-the-art performance on ImageNet classification, COCO object detection, and ADE20K semantic segmentation tasks, while maintaining the simplicity and efficiency of standard ConvNets.

Strengths of ConvNeXt over Transformer Model:
ConvNeXt is built entirely from standard ConvNet modules, making it easier to implement and understand compared to Transformers, which require specialized modules like self-attention.ConvNeXt shows promising robustness on out-of-distribution datasets, outperforming some Transformer-based models in robustness benchmarks.Unlike Transformers, ConvNeXt does not require complex modules like shifted window attention or relative position biases, making it simpler to deploy.

Weaknesses of ConvNeXt over Transformer Model:
While ConvNeXt excels in tasks like image classification, object detection, and segmentation, it may not be as flexible as Transformers for tasks requiring multimodal learning which is an advancing area of deep learning used in robotics, AI in daily life. The benefits of increasing kernel size saturate at 7x7, meaning that further increasing the kernel size does not improve performance. This limits the model's ability to capture even larger receptive fields compared to Transformers.Like Transformers, ConvNeXt benefits significantly from large-scale pre-training (e.g., on ImageNet-22K). Without such pre-training, its performance may not be as competitive.

ConvNeXt is a powerful alternative to Vision Transformers, offering a simpler and more efficient architecture while achieving competitive or superior performance on a wide range of computer vision tasks. Its design choices, inspired by Transformers but implemented using standard ConvNet modules, make it an attractive option for practitioners who value simplicity, efficiency, and strong performance. However, the choice between ConvNeXt and Transformers should be guided by the specific requirements of the task at hand, as Transformers may still offer advantages in certain scenarios, such as multimodal learning or tasks requiring global context.

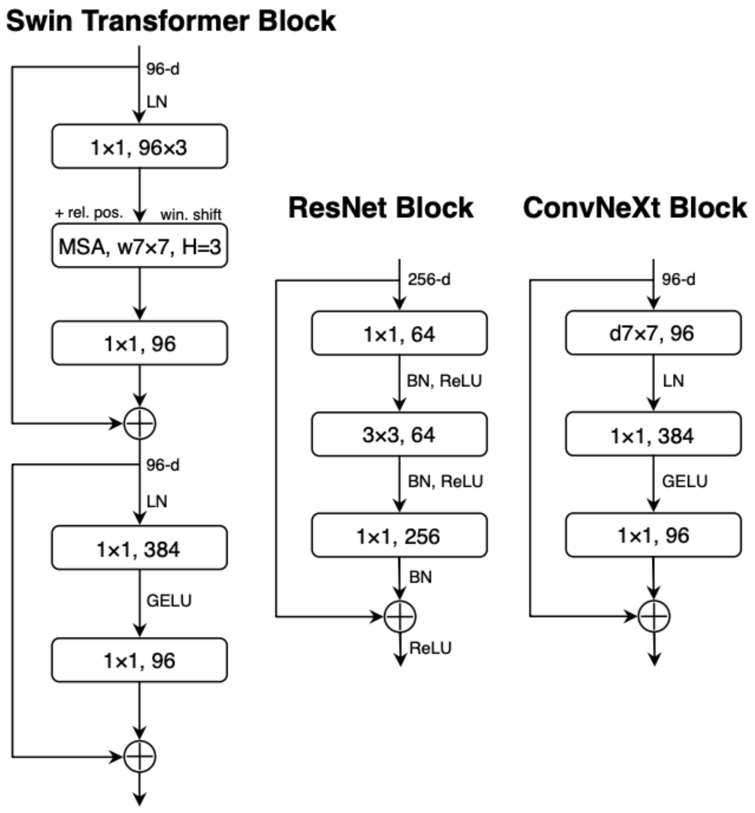


Fig 1.Block designs for a ResNet, a Swin Transformer and a ConvNeXt [1]

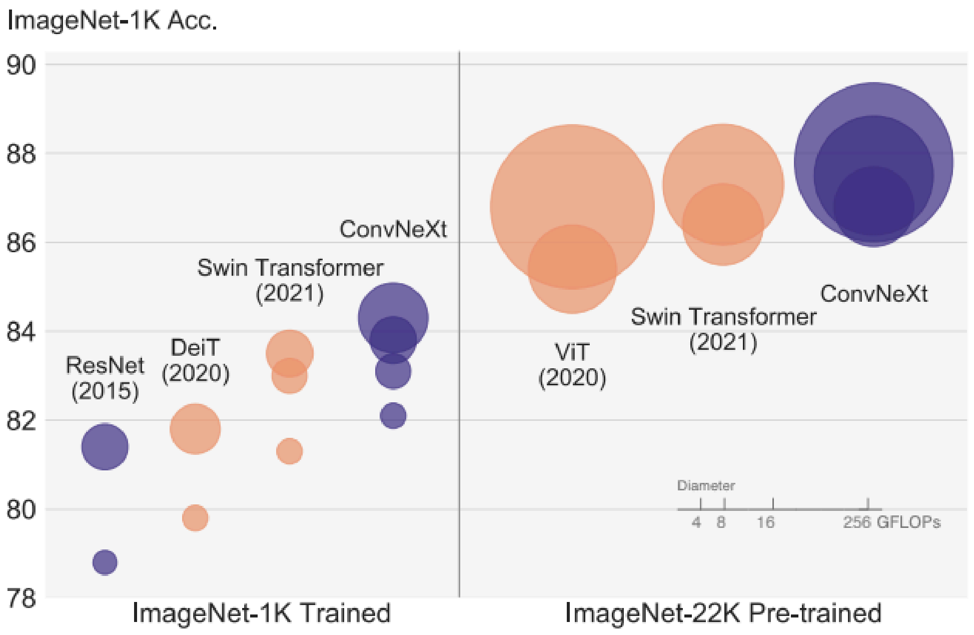


Fig 2.ImageNet1K classification results for ConvNets & Vision Transformer with their variants [1]

The other import details to be noted here are the training parameters, This hugely impacts the results from the paper and our project implementation. The paper has implemented it with the following settings.

The ImageNet1K dataset consists of 1000 object classes with 1.2M training images. The authors report ImageNet1K top1 accuracy on the validation set. They conduct pre-training on ImageNet22K, a larger dataset of 21841 classes (a superset of the 1000 ImageNet-1K classes) with 14M images for pre-training, and then fine-tune the pre-trained model on ImageNet1K for evaluation.They train ConvNeXts for 300 epochs using AdamW with a learning rate of 4e-3. They use a batch size of 4096 and a weight decay of 0.05.For data augmentations, they adopt common schemes including Mixup, Cut Mix, RandAugment and Random Erasing.

Approach and Experimentation :

Key Theoretical concepts:
LayerNorm is used instead of Batch Normalization (BN) in ConvNeXt. LayerNorm normalizes the activations across the channels for each individual sample, which is more suitable for tasks with varying batch sizes and is commonly used in Transformers. In ConvNeXt,LayerNorm is applied before the 1x1 convolutional layers in each block, which helps stabilize training and improve performance.Global Response Normalization (GRN) is a normalization technique that normalizes the responses across channels globally. It helps in improving the model's robustness and generalization capabilities.GRN is applied after the depthwise convolution and before the GELU activation in the ConvNeXt block.

I then came across a very interesting and new research of CVPR 2025 titled as “Transformers without Normalization”.[2] The paper demonstrates that Dynamic Tanh (DyT) can match or even exceed the performance of traditional normalization layers like LayerNorm across a wide range of tasks.DyT is designed to replace LayerNorm in Transformer architectures. It is defined as:

$DyT(x) = \gamma * \tanh(\alpha x) + \beta$; where α is a learnable scalar parameter, γ and β are learnable per-channel parameters.

The ConvNeXt architecture consists of multiple stages, each with a different feature map resolution. The number of channels doubles at each stage, similar to ResNet and Swin Transformers.The stem layer is the first layer that processes the input image. In ConvNeXt, the stem layer uses a 4x4 non-overlapping convolution with a stride of 4, which aggressively downsamples the input image to a lower resolution feature map.This is different from the traditional ResNet stem, which uses a 7x7 convolution followed by max pooling.
ConvNeXt Block:

Each ConvNeXt block consists of the following layers:
LayerNorm: Applied before the 1x1 convolutional layers.
1x1 Convolution: Expands the channel dimension (inverted bottleneck).
Depthwise Convolution: A 7x7 depthwise convolution is used to mix spatial information.
GELU Activation: Applied after the depthwise convolution.
Global Response Normalization (GRN): Applied after the GELU activation.
1x1 Convolution: Reduces the channel dimension back to the original size.
Residual Connection: Adds the input to the output of the block.

Between stages, ConvNeXt uses 2x2 convolutional layers with stride 2 for downsampling. This is different from ResNet, which uses 3x3 convolutions with stride 2 for downsampling. LayerNorm is applied before each downsampling layer to stabilize training.

Imports and Setup:
numpy: For numerical operations.
torch: PyTorch library for deep learning.
torch.nn: Neural network modules (e.g., layers, loss functions).
torch.optim: Optimization algorithms (e.g., SGD, Adam).
DataLoader and sampler: For loading and sampling data.
torch.nn.functional: Functional API for neural network operations (e.g., activation functions).
torchvision.datasets: Predefined datasets like CIFAR-100.
torchvision.transforms: Data preprocessing and augmentation.
trunc_normal_ and DropPath: From the timm library for weight initialization and stochastic depth.
Data Preprocessing:
The CIFAR-10 and CIFAR-100 datasets are labeled subsets of the 80 million tiny images dataset. CIFAR-10 and CIFAR-100 were created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.This dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the

CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).
Here we set up a transform to preprocess the data by subtracting the mean RGB value and dividing by the standard deviation of each RGB value; we've hardcoded the mean and std.We set up a Dataset object for each split (train / val / test); Datasets load training examples one at a time, so we wrap each Dataset in a DataLoader which iterates through the Dataset and forms minibatches. We divide the CIFAR-100 training set into train and val sets by passing a Sampler object to the DataLoader telling how it should sample from the underlying Dataset.

Utility Functions:

check_accuracy - Evaluates the model's accuracy on a given dataset (validation or test).
train - Trains the model for a specified number of epochs.

Classes of Layers:

The LayerNorm class implements Layer Normalization, which normalizes activations across the feature dimension for each data point. It supports two data formats (channels_last and channels_first) and uses learnable parameters (weight and bias) to scale and shift the normalized output. It stabilizes training by reducing internal covariate shift and is widely used in deep learning models.
The GRN class implements Global Response Normalization, a technique that normalizes activations globally across spatial dimensions. It computes the L2 norm of the input across height and width, normalizes it by the mean L2 norm, and applies learnable scaling (gamma) and shifting (beta) to enhance the model's ability to capture global patterns.
The DynamicTanh class replaces traditional normalization layers with a learnable tanh function. It scales the input using a learnable parameter (alpha), applies the tanh activation, and then scales and shifts the output using learnable weight and bias. It is computationally lighter than LayerNorm and avoids explicit computation of mean and variance.

The Model:

It begins with a stem layer that uses a 4x4 convolution with stride 4 to downsample the input image, followed by LayerNorm for normalization. The model then progresses through three downsampling layers, each consisting of LayerNorm and a 2x2 convolution with stride 2 to reduce spatial dimensions. The core of the model comprises four stages, each containing multiple residual blocks. Each block includes a 7x7 depthwise convolution for spatial feature extraction, followed by LayerNorm, a pointwise convolution to expand channels, a GELU activation, and a Global Response Normalization (GRN) layer to enhance global feature interactions. Stochastic depth is applied for regularization, with drop path rates linearly increasing across blocks. The final stage uses global average pooling to reduce spatial dimensions to 1x1, followed by a LayerNorm and a linear classification head. The model supports variants like convnextv2_atto, convnextv2_nano, convnextv2_tiny, convnextv2_base, and convnextv2_huge,which differ in depth and feature dimensions.

The ConvNeXtV2n model is a variant of ConvNeXtV2 where LayerNorm is replaced with Dynamic Tanh (DyT). The architecture begins with a stem layer using a 4x4 convolution with stride 4, followed by DynamicTanh for normalization. The model then applies three downsampling layers, each consisting of DynamicTanh and a 2x2 convolution with stride 2. The four stages of the model contain residual blocks with a 7x7 depthwise convolution, DynamicTanh, a pointwise convolution to expand channels, a GELU activation and a GRN layer. The DynamicTanh layer uses a learnable tanh function with a scaling factor alpha and affine parameters weight and bias to normalize activations. Stochastic depth is applied similarly to the ConvNeXtV2 model. The final stage uses global average pooling, followed by DynamicTanh and a linear classification head. This variant is computationally lighter and avoids explicit computation of mean and variance, making it suitable for efficiency-focused applications.

convnextv2_atto model = ConvNeXtV2(depths=[2, 2, 6, 2], dims=[40, 80, 160, 320])
convnextv2_nano model = ConvNeXtV2(depths=[2, 2, 8, 2], dims=[80, 160, 320, 640])
convnextv2_tiny model = ConvNeXtV2(depths=[3, 3, 9, 3], dims=[96, 192, 384, 768])
convnextv2_basemodel= ConvNeXtV2(depths=[3, 3, 27, 3], dims=[128, 256, 512, 1024])
convnextv2_huge model = ConvNeXtV2(depths=[3, 3, 27, 3], dims=[352, 704, 1408, 2816])

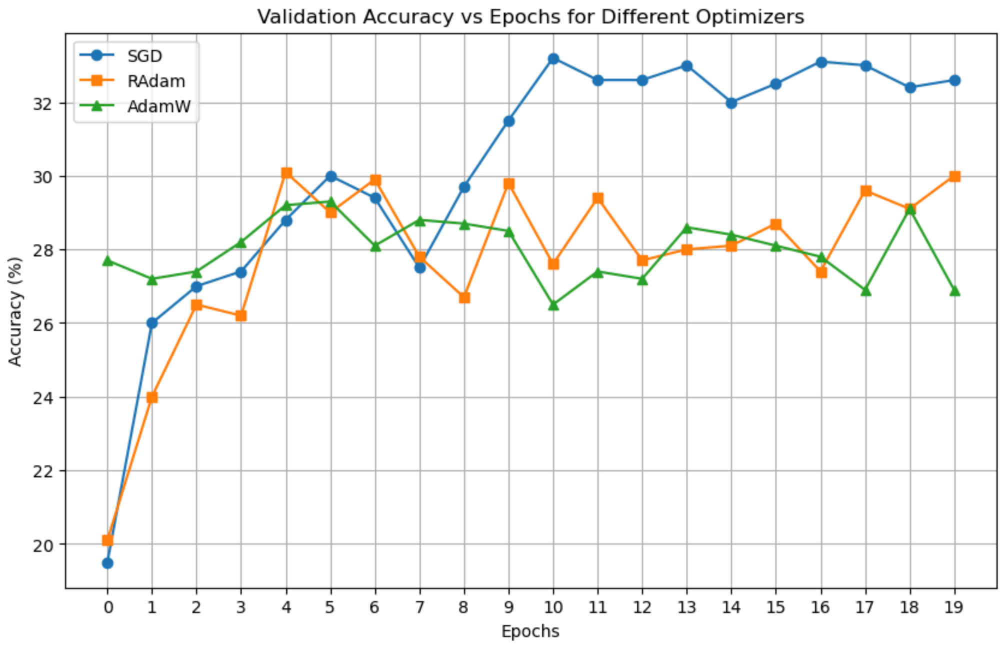
Experimentation :

I have performed various experiments of which few are listed below, these experiments include testing with multiple parameter changes including optimizers, model arch size, normalisation function. Note: There are still many other ways of experimentation with data augmentation, learning rate scheduler, pre-trained model which is not covered in this project.

Test 1. Base Model and RAdam Optimizer

model = convnextv2_base()
optimizer = optim.RAdam(model.parameters(), lr=0.001, betas=(0.9, 0.999),eps=1e-08)
train(model, optimizer, epochs=50)

Highest accuracy at Epoch 21, loss = 0.2339, Got 357 / 1000 correct (35.70)



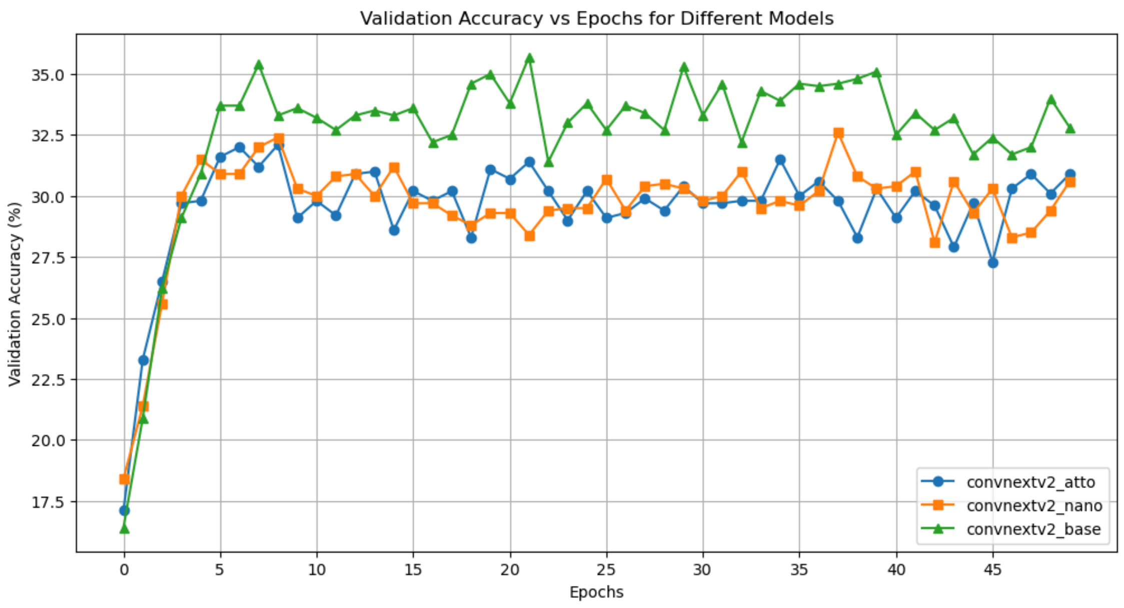
Graph 1. Base Model and Multiple Optimizer

Test 2. Base Model and Multiple Optimizers (Graph 1)

```
model = convnextv2_base()
optimizer1 = optim.SGD(model.parameters(), lr=0.001,momentum=0.9, nesterov=True)
optimizer2 = optim.RAdam(model.parameters(), lr=0.001, betas=(0.9, 0.999),eps=1e-08)
optimizer3 = optim.AdamW(model.parameters(), lr=0.001, betas=(0.9, 0.999),eps=1e-08)
train(model, optimizer1, epochs=20)
train(model, optimizer2, epochs=20)
train(model, optimizer3, epochs=20)
```

Highest accuracy on each optimizer

1.Epoch 10, loss = 0.0239, Got 332 / 1000 correct (33.20)
2. Epoch 19, loss = 0.1223, Got 300 / 1000 correct (30.00)
3.Epoch 5, loss = 0.0450, Got 293 / 1000 correct (29.30)



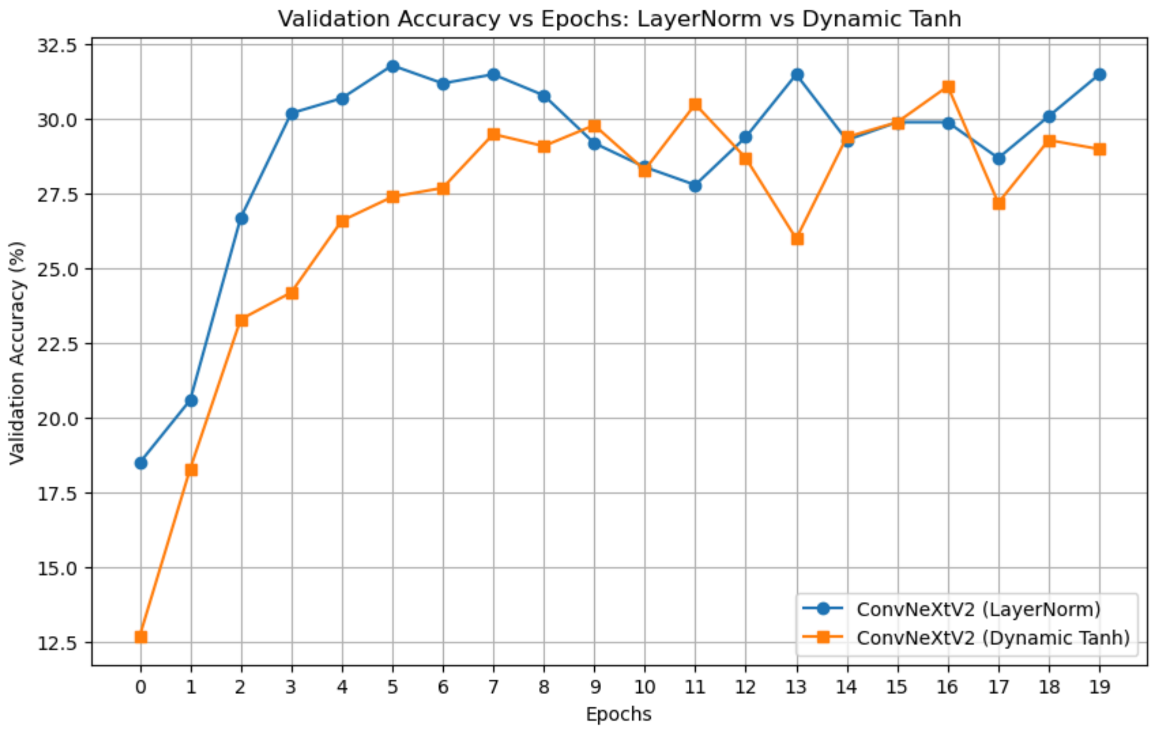
Graph 2. RAdam Optimizer and Multiple Models

Test 3. Multiple Models and RAdam Optimizer(Graph 2)

```
models = {'convnextv2_atto': convnextv2_atto,'convnextv2_nano': convnextv2_nano,
'convnextv2_base': convnextv2_base}
for model_name, model_func in models.items():
        model = model_func()
        optimizer = optim.RAdam(model.parameters(), lr=0.001, betas=(0.9, 0.999),eps=1e-08)
        train(model, optimizer, epochs=50)
```

Highest accuracy on each model

1. Epoch 8, loss = 1.2810, Got 321 / 1000 correct (32.10)
2. Epoch 37, loss = 0.1476, Got 326 / 1000 correct (32.60)
3. Epoch 21, loss = 0.2339, Got 357 / 1000 correct (35.70)



Graph 3. Dynamic Tanh vs Dynamic Tanh in ConvNext

Test 4. Tiny Model and RAdam Optimizer (Layer Norm)
model = convnextv2_tiny()
optimizer = optim.RAdam(model.parameters(), lr=0.001, betas=(0.9, 0.999),eps=1e-08)
train(model, optimizer, epochs=20)

Highest accuracy at Epoch 5, loss = 2.2537, Got 318 / 1000 correct (31.80)

Test 5. ConvNeXtV2 Tiny Model and RAdam Optimizer (Dynamic Tanh)
model = convnextv2_tiny()
optimizer = optim.RAdam(model.parameters(), lr=0.001, betas=(0.9, 0.999),eps=1e-08)
train(model, optimizer, epochs=20)

Highest accuracy at Epoch 16, loss = 0.1034, Got 311 / 1000 correct (31.10)

Result:

The final model and optimizer I would suggest are **ConvNeXtV2 Tiny Model and SGD-Nesterov Optimizer** as it gives comparatively similar results to Base Model and RAdam Optimizer. As the Base model is very huge and takes longer time for execution. **Got 341 / 1000 correct (34.10)**.Other improvements that can be made : Try different learning rates (e.g., 0.001, 0.0005, 0.0001) to see which works best.The current train_transform only includes normalization. Adding augmentation techniques like RandomCrop and HorizontalFlip can help the model generalize better.Use gradient clipping if training becomes unstable.

Reference:

1. A ConvNet for the 2020s. CVPR 2022.Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell and SainingXie, Facebook AI Research, UC Berkeley
2.Transformers with Normalization. CVPR 2025.Jiachen Zhu, Xinlei Chen, Kaiming He, Yann LeCun and Zhuang LiuFAIR, NYU, MIT, Princeton