Lab 3: Gesture Recognition using Convolutional Neural Networks

In this lab you will train a convolutional neural network to make classifications on different hand gestures. By the end of the lab, you should be able to:

1. Load and split data for training, validation and testing
2. Train a Convolutional Neural Network
3. Apply transfer learning to improve your model

Note that for this lab we will not be providing you with any starter code. You should be able to take the code used in previous labs, tutorials and lectures and modify it accordingly to complete the tasks outlined below.

What to submit

Submit a PDF file containing all your code, outputs, and write-up from parts 1-5. You can produce a PDF of your Google Colab file by going to **File > Print** and then save as PDF. The Colab instructions has more information. Make sure to review the PDF submission to ensure that your answers are easy to read. Make sure that your text is not cut off at the margins.

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Please use Google Colab to complete this assignment. If you want to use Jupyter Notebook, please complete the assignment and upload your Jupyter Notebook file to Google Colab for submission.

Colab Link

Include a link to your colab file here

Colab Link:

https://colab.research.google.com/github/GreatArcStudios/APS360/blob/master/Lab%203/Lab3.ipynb

Dataset

American Sign Language (ASL) is a complete, complex language that employs signs made by moving the hands combined with facial expressions and postures of the body. It is the primary language of many North Americans who are deaf and is one of several communication options used by people who are deaf or hard-of-hearing. The hand gestures representing English alphabet are shown below. This lab focuses on classifying a subset of these hand gesture images using convolutional neural networks. Specifically, given an image of a hand showing one of the letters A-I, we want to detect which letter is being represented.

Part B. Building a CNN [50 pt]

For this lab, we are not going to give you any starter code. You will be writing a convolutional neural network from scratch. You are welcome to use any code from previous labs, lectures and tutorials. You should also write your own code.

You may use the PyTorch documentation freely. You might also find online tutorials helpful. However, all code that you submit must be your own.

Make sure that your code is vectorized, and does not contain obvious inefficiencies (for example, unecessary for loops, or unnecessary calls to unsqueeze()). Ensure enough comments are included in the code so that your TA can understand what you are doing. It is your responsibility to show that you understand what you write.

This is much more challenging and time-consuming than the previous labs. Make sure that you give yourself plenty of time by starting early.

1. Data Loading and Splitting [5 pt]

Download the anonymized data provided on Quercus. To allow you to get a heads start on this project we will provide you with sample data from previous years. Split the data into training, validation, and test sets.

Note: Data splitting is not as trivial in this lab. We want our test set to closely resemble the setting in which our model will be used. In particular, our test set should contain hands that are never seen in training!

Explain how you split the data, either by describing what you did, or by showing the code that you used. Justify your choice of splitting strategy. How many training, validation, and test images do you have?

For loading the data, you can use plt.imread as in Lab 1, or any other method that you choose. You may find torchvision.datasets.ImageFolder helpful. (see https://pytorch.org/docs/stable/torchvision/datasets.html?highlight=image%20folder#torchvision.datasets.ImageFolder)

```
# some dependencies for fancy progress bar and model summary
! pip install tqdm
! pip install torchinfo
```

```
# imports
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
```

```
import numpy as np
import matplotlib.pyplot as plt
import tqdm

from torch.utils.data import DataLoader, random_split
from tqdm import trange
from torchinfo import summary

# create dataset and dataloaders


def create_datasets(small_set_adjustment=1.0):
    transforms = [torchvision.transforms.ToTensor(
    ), torchvision.transforms.Normalize((0, 0, 0), (1, 1, 1))]
    transforms = torchvision.transforms.Compose(transforms)
    images_dataset = torchvision.datasets.ImageFolder(
        "./Dataset/Lab3_Gestures_Summer/", transform=transforms)
    leftover_prob = 1 - 0.7 * small_set_adjustment - 0.1 * \
        small_set_adjustment - 0.2 * small_set_adjustment
    split_sets = random_split(images_dataset, [
                              0.7 * small_set_adjustment, 0.1 * small_set_adjustment, 0.2 * small_set_adjustment, leftover_prob])
    return split_sets


def create_dataloaders(split_sets, batch_size=32, shuffle=True, use_cuda=True):
    train_set, val_set, test_set = split_sets

    train_loader = DataLoader(
        train_set, batch_size=batch_size, shuffle=shuffle, pin_memory=use_cuda)
    val_loader = DataLoader(val_set, batch_size=len(
        val_set), shuffle=shuffle, pin_memory=use_cuda)
    test_loader = DataLoader(test_set, batch_size=len(
        test_set), shuffle=shuffle, pin_memory=use_cuda)

    return train_loader, val_loader, test_loader
```

```
    # 4th set is a leftover set if we are trying to create a small subset
    train_set, val_set, test_set, _ = create_datasets()
    split_sets = (train_set, val_set, test_set)

    print(len(train_set), len(val_set), len(test_set))
```

I split the data using a 70-10-20 training, validation, test split, meaning I could use 10% of the data to tune my hyperparameters and estimate the training error (usually training error is somewhat below the validation error). Further, I chose to use a 20% test set since this would give us a better approximation of generalization performance than using something like a 10% test set. I didn't want to go with a smaller test set to increase the training set size because I wanted to see more reliability if I ended up with a good model. Ultimately, I have 2220 samples overall, meaning 1554 samples in the training set, 222 samples in the validation set, and finally 444 samples in the test set. I also chose to use the pytorch `ImageFolder` dataset API for creating the datasets since it allowed me to use the `random_split` function, which accepts a vector of probabilities (proportions) when sampling the train/validation/test sets. This allows us to conveniently control for the average number of samples randomly included in each dataset; consider that the expectation for some class i with probability p_i of a multinomial distribution is $n \cdot p_i$, i.e., $\mathbb{E}[x_i] = n \cdot p_i$. Then I used the dataloader API to create dataloaders, which is advantageous since it has support for pinned memory, which can help with copies between the CPU and GPU and easy control over batch size and data shuffling.

2. Model Building and Sanity Checking [15 pt]

Part (a) Convolutional Network - 5 pt

Build a convolutional neural network model that takes the (224x224 RGB) image as input, and predicts the gesture letter. Your model should be a subclass of nn.Module. Explain your choice of neural network architecture: how many layers did you choose? What types of layers did you use? Were they fully-connected or convolutional? What about other decisions like pooling layers, activation functions, number of channels / hidden units?

```
class SimpleCNN(nn.Module):
    def __init__(self, conv_dilation=2, dropout = 0.05):
        super().__init__()
        self.feature_embeddings = nn.Sequential(
            # produces shape 9, 200, 200
            nn.Conv2d(3, 12, 26),
            # batch norm to help model training - normalize data
            nn.BatchNorm2d(12),
```

```
            nn.Mish(),
            # produces shape 9, 99, 99
            nn.AvgPool2d(4, 2),
            nn.Mish(),
            # produces shape 4, 93, 93
            # expand receptive field
            nn.Conv2d(12, 8, 3, dilation=conv_dilation),
            nn.BatchNorm2d(8),
            nn.Mish(),
            # get the most intensive more "long range" features
            nn.MaxPool2d(3),
            nn.Mish(),
            # expand receptive field
            nn.Conv2d(8, 4, 2, dilation=conv_dilation),
            nn.BatchNorm2d(4),
            nn.Mish(),
            nn.Conv2d(4, 2, 2),
            nn.AvgPool2d(2),
            nn.Mish())
        # programmatically get feature embedding size
        self._feature_embed_size = list(self.feature_embeddings(torch.rand(1, 3, 224, 224)).detach().size())
        self.embedding_size = np.prod(self._feature_embed_size)
        # use dropout to regularize model
        self.dropout = nn.Dropout(p=dropout)
        self.flatten = nn.Flatten()

        # autoencoder like architecture - just remove reconstruction layers
        self.fc1 = nn.Linear(self.embedding_size, self.embedding_size//2)
        self.act1 = nn.Mish()
        self.fc2 = nn.Linear(self.embedding_size//2, self.embedding_size//4)
        self.act2 = nn.Mish()
        # low dimensional latent space
        self.fc3 = nn.Linear(self.embedding_size//4, self.embedding_size//4)
        self.act3 = nn.Mish()
        self.fc4 = nn.Linear(self.embedding_size//4, self.embedding_size//2)
        self.act4 = nn.Mish()
        self.fc5 = nn.Linear(self.embedding_size//2, 9)

    def forward(self, input):
        # create latent space rep.
        z = self.flatten(self.feature_embeddings(input))
        z = self.dropout(z)
        z_1 = self.act1(self.fc1(z))
        z_2 = self.dropout(z_1)
        z_2 = self.act2(self.fc2(z_2))
        z_3 = self.dropout(z_2)
        z_3 = self.act3(self.fc3(z_3))
        z_4 = self.dropout(z_3)
        z_4 = self.act4(self.fc4(z_4) + z_1) # skip connection
        z_5 = self.dropout(z_4)
        out = self.fc5(z_5)
        return out

def init_weights(layer):
    if isinstance(layer, nn.Linear):
        torch.nn.init.xavier_normal_(layer.weight)
```

I constructed my neural network first with a stack of convolution and pooling layers. This was done to get compressed latent feature representations that I could then feed into various linear layers motivated by the autoencoder architecture, but with the final reconstruction layers removed. I decided on adding quite a few convolution layers because I wanted to feed higher level feature representations into the fully connected layers, which would be further tuned in the latent space by the fully connected layers. Also, I chose to output 9 feature representations (output channel dimension of 9) in the first convolution layer because just wanted some extra granularity in learned kernels (more learned kernels) since each feature representation is only generated by one kernel. I used a dilated convolution since traditional convolution kernels considers only adjacent pixels, and so I used the dilated convolution to increase the receptive field, which allows for me to capture the potentially useful dependencies between further apart elements in the feature representations. I used average pooling after the first convolution layer since I wanted to average out “sharp” signals rather than have the network focus on the most intense values of the low level features. My final pooling layer also used average pooling for the same reason but for higher level features; I wanted the fully connected layers to be fed what was “generally” going on in some image. I used dropout to help prevent overfitting of the rather dense fully connected layers (the first one has more than half of the total parameters), and allows for essentially model averaging (Srivastava et al.). Further, my linear layers were constructed in an autoencoder like fashion, i.e., we project into a smaller latent space, forcing the network to further learn what was actually important, and then projected it back up for the final output layer. I also added a skip connection for the `z_4` layer (`self.fc4(z_4) + z_1`) in order to help mitigate the vanishing gradient problem, and

"remind" the higher dimensional projection (`z_4`) from the smaller latent space (`z_3`) of a latent representation closer to that of the convolution feature representations (`z_1`). Finally, I used the Mish activation since it can create much smoother loss landscapes and self-regularizes (Misra).

Part (b) Training Code - 5 pt

Write code that trains your neural network given some training data. Your training code should make it easy to tweak the usual hyperparameters, like batch size, learning rate, and the model object itself. Make sure that you are checkpointing your models from time to time (the frequency is up to you). Explain your choice of loss function and optimizer.

```
def compute_loss(preds, targets):
    criterion = nn.CrossEntropyLoss()
    return criterion(preds, targets)


def train_loop(train_loader, val_loader, convolution_dilation=2, epochs=1000, learning_rate=0.0095, momentum=0.95, wd=1e-4, batch_size=None, tr
    # determine if CUDA is available and set Tensor core flags
    if use_cuda and torch.cuda.is_available():
        dev = "cuda:0"
        torch.backends.cuda.matmul.allow_tf32 = True
        torch.backends.cudnn.allow_tf32 = True
    else:
        print("CUDA unavailable, training on CPU")
        dev = "CPU"
    device = torch.device(dev)

    network = SimpleCNN(conv_dilation=convolution_dilation)
    network = network.to(device)
    network.apply(init_weights)
    optimizer = torch.optim.SGD(network.parameters(
    ), lr=learning_rate, momentum=momentum, weight_decay=wd, nesterov=True)
    # use for Nvidia AMP training cycle
    scaler = torch.cuda.amp.GradScaler()

    if train_data is not None:
        summary(network, input_data=train_data, verbose=1, device=device)

    loss_dict = {"config": f"Convolution Dilation: {convolution_dilation}, Epochs: {epochs}, Lr: {learning_rate}, Momentum:{momentum}, Weight D
                 "train_loss": [], "val_loss": [],
                 "train_acc": [], "val_acc": []}

    # counter for determining checkpoints
    best_val_acc = 0.0

    def run_epoch():
        network.train()
        epoch_loss, val_loss = 0.0, 0.0
        train_correct, train_total = 0.0, 0.0
        val_correct, val_total = 0.0, 0.0
        for i, batch in enumerate(train_loader):
            # reenable train mode to enable dropout
            inputs, targets = batch
            inputs = inputs.to(device, non_blocking=True)
            targets = targets.to(device, non_blocking=True)

            network.zero_grad(set_to_none=True)

            # use Nvidia AMP for tensor cores speed up.
            with torch.autocast(device_type='cuda', dtype=torch.float16):
                preds = network(inputs)
                loss = compute_loss(preds, targets)

            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()

            # evaluate the model for each batch
            # NOTE: we evaluate the model on the logits since the logit function is the inverse
            # of the logistic function (or softmax) and thus is a monotone transformation of
            # the actual predicted probabilities.
            with torch.no_grad():
                # eval mode to disable dropout
```

```
                network.eval()
                epoch_loss += loss
                _, category_preds = torch.max(preds, 1)
                train_correct += (category_preds ==
                                  targets).sum()
                train_total += preds.size()[0]
                # print(loss.item())

        # evaluate the validation set and save the epoch statistics
        with torch.no_grad():
            network.eval()
            # only one batch per val loader
            for i, batch in enumerate(val_loader):
                inputs, targets = batch
                inputs = inputs.to(device, non_blocking=True)
                targets = targets.to(device, non_blocking=True)

                preds = network(inputs)
                batch_val_loss = compute_loss(preds, targets)
                val_loss += batch_val_loss
                _, category_preds = torch.max(preds, 1)
                val_correct += (category_preds ==
                                targets).sum()
                val_total += preds.size()[0]
                loss_dict["val_loss"].append(batch_val_loss)
                loss_dict["val_acc"].append(val_correct/val_total)
                loss_dict["train_loss"].append(epoch_loss)
                loss_dict["train_acc"].append(train_correct/train_total)
        return epoch_loss, train_correct, train_total, val_loss, val_correct, val_total

    # the fancy TQDM progress bar slows down model training likely due to GPU -> CPU copies
    if use_tqdm:
        with trange(epochs, desc="Train epochs", unit="epoch") as train_bar:
            for epoch in train_bar:
                epoch_loss, train_correct, train_total, val_loss, val_correct, val_total = run_epoch()
                train_bar.set_postfix(epoch_loss=epoch_loss,
                                      train_acc=train_correct/train_total,
                                      train_correct=train_correct,
                                      train_total=train_total,
                                      val_loss=val_loss,
                                      val_acc=val_correct/val_total)
    else:
        for epoch in range(epochs):
            # run the epoch
            epoch_loss, train_correct, train_total, val_loss, val_correct, val_total = run_epoch()

            # rudimentary learning rate scheduler
            if use_lr_sched and epoch >= epoch_slow_lr_start:
                for param in optimizer.param_groups:
                    param["lr"] = slow_lr
                    param["momentum"] = low_momentum

            # model check point based on validation accuracy
            if torch.round(val_correct/val_total, decimals=3) > best_val_acc and epoch >= epoch_save_start:
                torch.save(network.state_dict(
                ), f"./models/valacc_{val_correct/val_total}-convdial_{convolution_dilation}-lr_{learning_rate}-momentum_{momentum}-batch_size_
                best_val_acc = torch.round(val_correct/val_total, decimals=3)

            if epoch % print_every == 0:
                print(f"Epoch {epoch} stats: train loss {epoch_loss.item()}, train acc {(train_correct/train_total).item()}, val loss {val_loss

    return network, loss_dict


def compute_test_performance(network, test_loader):
    test_correct, test_total = 0.0, 0.0
    test_loss = 0.0

    device = next(network.parameters()).device
    with torch.no_grad():
        # disable dropout for test time
        network.eval()
        for i, batch in enumerate(test_loader):
```

```
            inputs, targets = batch
            inputs = inputs.to(device, non_blocking=True)
            targets = targets.to(device, non_blocking=True)
            preds = network(inputs)
            test_loss += compute_loss(preds, targets).item()
            _, category_preds = torch.max(preds, 1)
            #print(category_preds, targets)
            test_correct += float((category_preds == targets).sum().item())
            test_total += len(targets)
    return test_correct/test_total, test_correct, test_total, test_loss
```

I chose cross entropy loss since our task is a mutliclass prediction task, and so we essentially need to output logits to define a multinomial distribution for the hand signs. In other words, our task is to fit a conditional distribution that is a vector of probabilities when conditioned on some image, i.e., for some image $\mathbf{x}$ and some vector of hand sign classes $\mathbf{c}$, we fit $p(\mathbf{c} \mid \mathbf{x})$. Cross entropy loss allows us to do this from an output layer ((https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html).

Then, I chose SGD as the optimizer because SGD is theoretically guaranteed to converge to the minimum norm solution (assuming a convex objective function, e.g., linear regression), while adaptive optimizers like Adam are not guaranteed to do so (Reddi et al.). Additionally, SGD also implements nesterov momentum, so it has the added benefit of potentially being able to escape certain local minima. In particular, I like how nesterov momentum jumps in the direction of the accumulated gradients then corrects for it since it seems that this network and data result in a tricky loss landscape where there are fairly "deep" local minima (and SGD sometimes needs a gentle extra push).

Part (c) "Overfit" to a Small Dataset - 5 pt

One way to sanity check our neural network model and training code is to check whether the model is capable of "overfitting" or "memorizing" a small dataset. A properly constructed CNN with correct training code should be able to memorize the answers to a small number of images quickly.

Construct a small dataset (e.g. just the images that you have collected). Then show that your model and training code is capable of memorizing the labels of this small data set.

With a large batch size (e.g. the entire small dataset) and learning rate that is not too high, You should be able to obtain a 100% training accuracy on that small dataset relatively quickly (within 200 iterations).

```
def training_wrapper(split_sets, convolution_dilation=2, epochs=1000, learning_rate=0.0092, momentum=0.94, wd=1.6e-4, batch_size=None, use_cuda
    train_batch_size = len(split_sets[0]) if batch_size is None else batch_size
    train_loader, val_loader, test_loader = create_dataloaders(
        split_sets, batch_size=train_batch_size, use_cuda=use_cuda, shuffle=shuffle)

    # used for torch info summary
    if use_cuda and torch.cuda.is_available():
        dev = "cuda:0"
    else:
        print("CUDA unavailable, training on CPU")
        dev = "CPU"
    device = torch.device(dev)
    dummy_data = torch.rand((train_batch_size, 3, 224, 224)).to(device)

    trained_network, loss_dict = train_loop(train_loader, val_loader, convolution_dilation=convolution_dilation,
                                            epochs=epochs, learning_rate=learning_rate, momentum=momentum, wd=wd,
                                            batch_size=train_batch_size, train_data=dummy_data, use_cuda=use_cuda,
                                            epoch_save_start=epoch_save_start, use_lr_sched=use_lr_sched,
                                            epoch_slow_lr_start=epoch_slow_lr_start, slow_lr=slow_lr,
                                            low_momentum=low_momentum, print_every=print_every)
    return trained_network, loss_dict, (train_loader, val_loader, test_loader)
```

```
small_train, small_val, small_test, _ = create_datasets(small_set_adjustment=0.35)
small_split_sets = (small_train, small_val, small_test)
len(small_train)
```

```
544
```

```
trained_network, loss_dict, loaders = training_wrapper(small_split_sets, epochs=200, learning_rate=0.00915, momentum=0.925, wd=1.6e-2, shuffle=
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
```

```
==========================================================================================
SimpleCNN                                [544, 9]                  --
├─Sequential: 1-1                        [544, 2, 14, 14]          --
│    └─Conv2d: 2-1                       [544, 12, 199, 199]       24,348
│    └─BatchNorm2d: 2-2                  [544, 12, 199, 199]       24
│    └─Mish: 2-3                         [544, 12, 199, 199]       --
│    └─AvgPool2d: 2-4                    [544, 12, 98, 98]         --
│    └─Mish: 2-5                         [544, 12, 98, 98]         --
│    └─Conv2d: 2-6                       [544, 8, 94, 94]          872
│    └─BatchNorm2d: 2-7                  [544, 8, 94, 94]          16
│    └─Mish: 2-8                         [544, 8, 94, 94]          --
│    └─MaxPool2d: 2-9                    [544, 8, 31, 31]          --
│    └─Mish: 2-10                        [544, 8, 31, 31]          --
│    └─Conv2d: 2-11                      [544, 4, 29, 29]          132
│    └─BatchNorm2d: 2-12                 [544, 4, 29, 29]          8
│    └─Mish: 2-13                        [544, 4, 29, 29]          --
│    └─Conv2d: 2-14                      [544, 2, 28, 28]          34
│    └─AvgPool2d: 2-15                   [544, 2, 14, 14]          --
│    └─Mish: 2-16                        [544, 2, 14, 14]          --
├─Flatten: 1-2                           [544, 392]                --
├─Dropout: 1-3                           [544, 392]                --
├─Linear: 1-4                            [544, 196]                77,028
├─Mish: 1-5                              [544, 196]                --
├─Dropout: 1-6                           [544, 196]                --
├─Linear: 1-7                            [544, 98]                 19,306
├─Mish: 1-8                              [544, 98]                 --
├─Dropout: 1-9                           [544, 98]                 --
├─Linear: 1-10                           [544, 98]                 9,702
├─Mish: 1-11                             [544, 98]                 --
├─Dropout: 1-12                          [544, 98]                 --
├─Linear: 1-13                           [544, 196]                19,404
├─Mish: 1-14                             [544, 196]                --
├─Dropout: 1-15                          [544, 196]                --
├─Linear: 1-16                           [544, 9]                  1,773
==========================================================================================
Total params: 152,647
Trainable params: 152,647
Non-trainable params: 0
Total mult-adds (G): 528.86
==========================================================================================
Input size (MB): 327.55
Forward/backward pass size (MB): 4790.22
Params size (MB): 0.61
Estimated Total Size (MB): 5118.38
==========================================================================================
Epoch 0 stats: train loss 2.1902267932891846, train acc 0.14154411852359772, val loss 2.209646701812744, val acc 0.06410256773233414
Epoch 10 stats: train loss 2.0526580810546875, train acc 0.27389705181121826, val loss 2.2041399478912354, val acc 0.07692307978868484
Epoch 20 stats: train loss 1.6772938966751099, train acc 0.4319852888584137, val loss 2.134267807006836, val acc 0.1794871836900711
Epoch 30 stats: train loss 1.0779668092727661, train acc 0.6397058963775635, val loss 1.6589462757110596, val acc 0.43589743971824646
Epoch 40 stats: train loss 0.7244219779968262, train acc 0.7591911554336548, val loss 1.2977230548858643, val acc 0.6025640964508057
Epoch 50 stats: train loss 0.5394009947776794, train acc 0.8363970518112183, val loss 1.4070969820022583, val acc 0.5769230723381042
Epoch 60 stats: train loss 0.29937633872032166, train acc 0.9227941036224365, val loss 1.5258115530014038, val acc 0.5897436141967773
Epoch 70 stats: train loss 0.16478298604488373, train acc 0.9650735259056091, val loss 1.097802996635437, val acc 0.6666666865348816
Epoch 80 stats: train loss 0.3268875777721405, train acc 0.875, val loss 10.344923973083496, val acc 0.3205128312110901
Epoch 90 stats: train loss 0.07267850637435913, train acc 0.9908088445663452, val loss 1.0189942121505737, val acc 0.7179487347602844
Epoch 100 stats: train loss 0.052319254726171494, train acc 0.9908088445663452, val loss 1.2930632829666138, val acc 0.6666666865348816
Epoch 110 stats: train loss 0.04485984519124031, train acc 0.9926470518112183, val loss 1.0033762454986572, val acc 0.7179487347602844
Epoch 120 stats: train loss 0.038824837654829025, train acc 0.9944853186607361, val loss 1.124265193939209, val acc 0.7051281929016113
Epoch 130 stats: train loss 0.03461975231766701, train acc 0.9944853186607361, val loss 1.0550730228424072, val acc 0.7051281929016113
Epoch 140 stats: train loss 0.02698923461139202, train acc 0.998161792755127, val loss 1.0080821514129639, val acc 0.7435897588729858
Epoch 150 stats: train loss 0.025783279910683632, train acc 0.998161792755127, val loss 0.9772793650627136, val acc 0.7564102411270142
Epoch 160 stats: train loss 0.021829893812537193, train acc 1.0, val loss 0.9808109998703003, val acc 0.7435897588729858
Epoch 170 stats: train loss 0.024946153163909912, train acc 0.998161792755127, val loss 0.9861094951629639, val acc 0.7435897588729858
Epoch 180 stats: train loss 0.029390716925263405, train acc 0.9963235259056091, val loss 0.9592354893684387, val acc 0.7692307829856873
Epoch 190 stats: train loss 0.018172232434153557, train acc 1.0, val loss 0.9660477042198181, val acc 0.7564102411270142
```

```
torch.cuda.empty_cache()
```

As seen above, we are able to achieve 100% accuracy on the training set within 200 epochs.

3. Hyperparameter Search [10 pt]

Part (a) - 1 pt

List 3 hyperparameters that you think are most worth tuning. Choose at least one hyperparameter related to the model architecture.

I could try tuning the following hyperparameters:

1. The convolution dilation scale to tune how far apart the elements used in the convolution operation will be chosen. This would affect the latent representations, but choosing a good value would lead to better representations across the image, i.e., learn features far apart in the image that are informative to the ASL sign.
2. I could also tune the learning rate since in the overfit example it seems to converge slower than it needs to even though both the validation and train losses are virtually 0, and the model scored 100% accuracy on both sets respectively.
3. I could try tuning the weight decay parameter as it is essentially a L_2 penalty and can help the model converge to a local minima of the loss surface that is well regularized, and thus potentially perform better on the test set.

Part (b) - 5 pt

Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different hyperparameter settings.

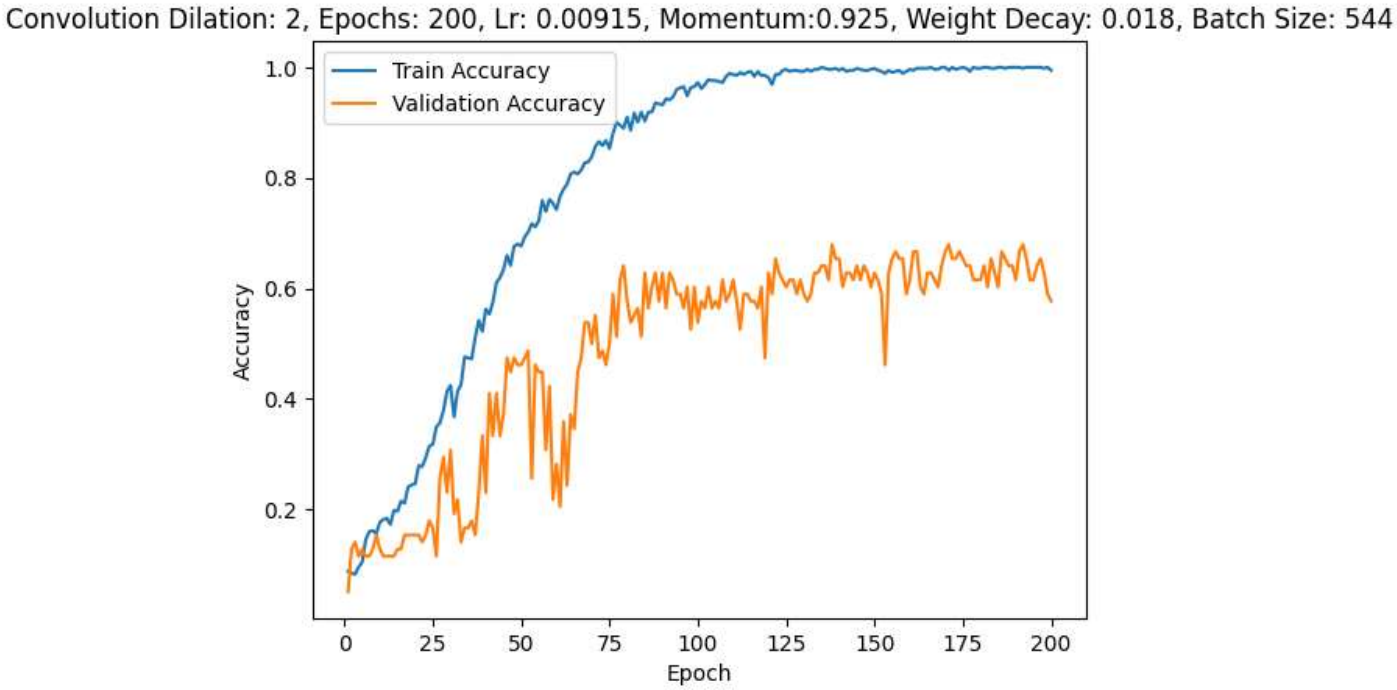
I tested the hyperparameter changes I discussed above.

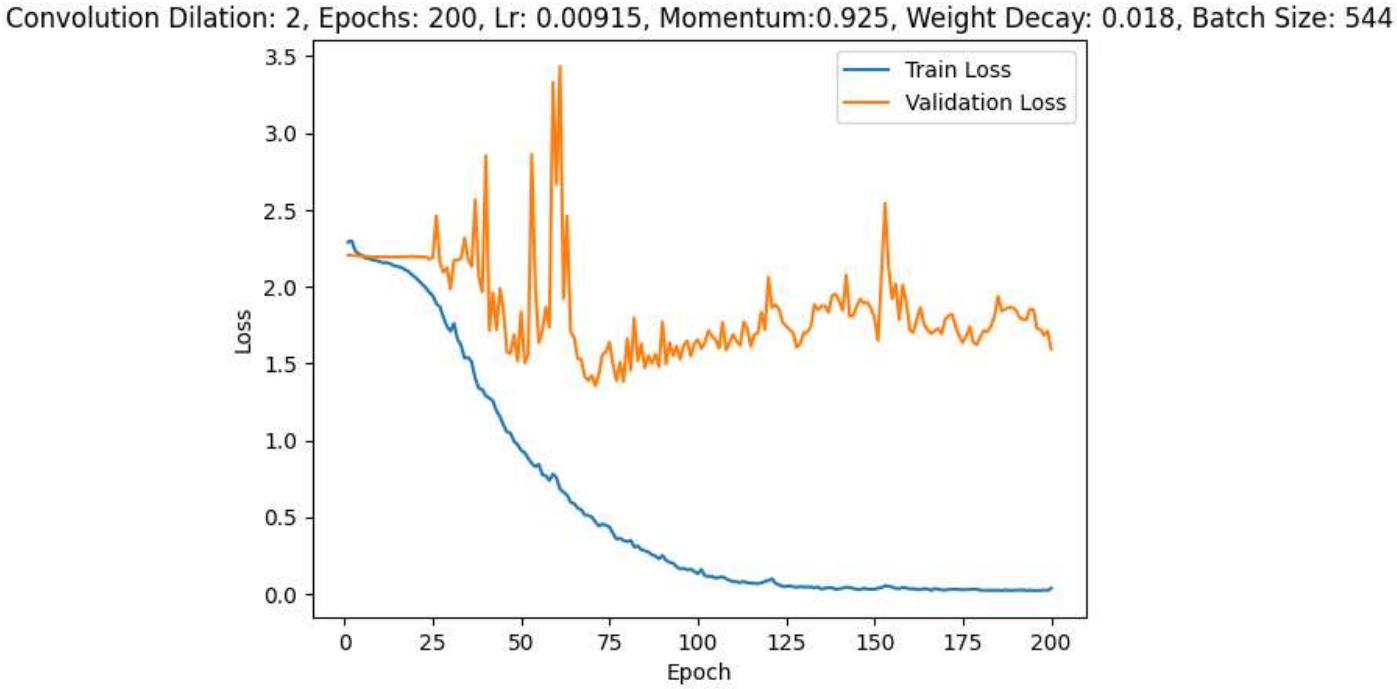
```
def plot_train_curves(loss_dict):
    plt.title(f"{loss_dict['config']}")
    n = len([value.cpu().data.numpy() for value in loss_dict["train_acc"]])
    plt.plot(range(1,n+1), [value.cpu().data.numpy() for value in  loss_dict["train_acc"]], label="Train Accuracy")
    plt.plot(range(1,n+1), [value.cpu().data.numpy() for value in  loss_dict["val_acc"]], label="Validation Accuracy")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

def plot_val_curves(loss_dict):
    plt.title(f"{loss_dict['config']}")
    n = len([value.cpu().data.numpy() for value in loss_dict["val_loss"]])
    plt.plot(range(1,n+1), [value.cpu().data.numpy() for value in loss_dict["train_loss"]], label="Train Loss")
    plt.plot(range(1,n+1), [value.cpu().data.numpy() for value in loss_dict["val_loss"]], label="Validation Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()
```

With the default hyperparameters, we see the following graphs:

```
plot_train_curves(loss_dict)
plot_val_curves(loss_dict)
```





My first experiment is changing the convolution dilation parameter to 4, which drastically increases the receptive field since distances between points on the kernel are now doubled. To help the optimizer explore the loss landscape more, I've also decreased the batch size, allowing for more variable estimates of the true gradient.

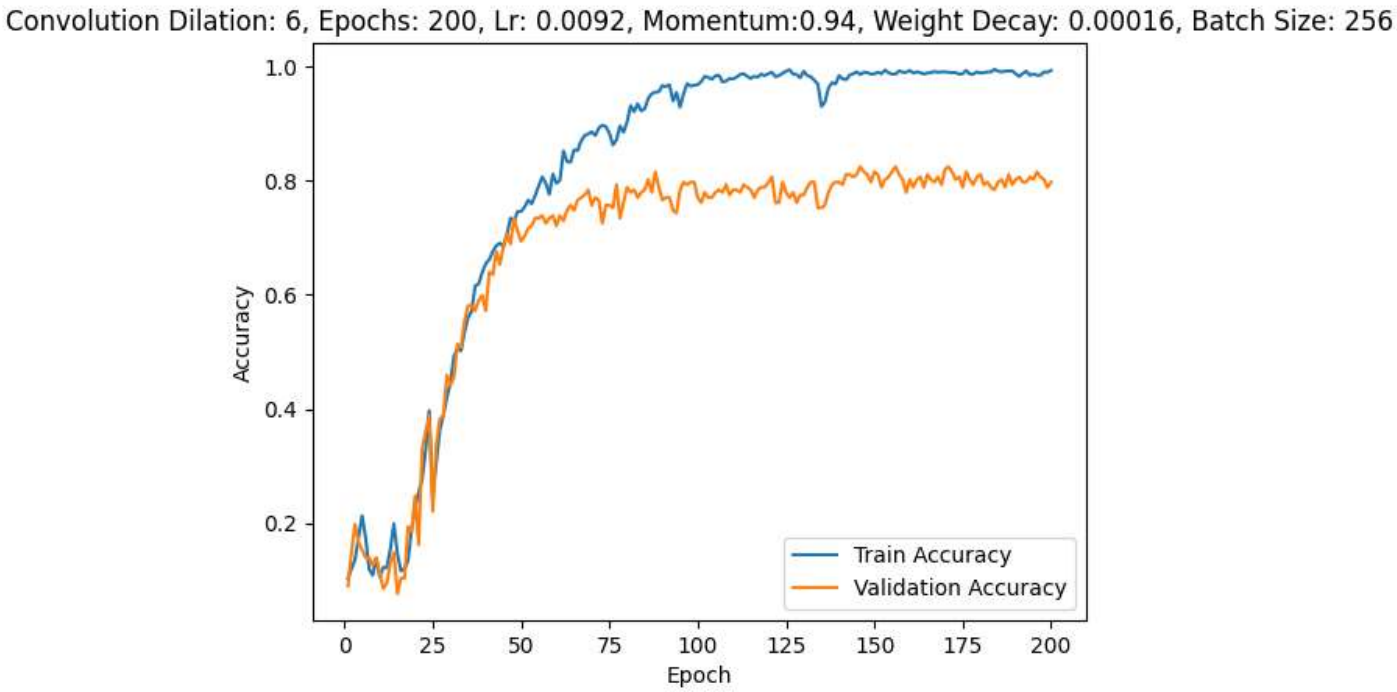
```
trained_network_dilation_change, loss_dict_dilation, _ = training_wrapper(split_sets, epochs=200, convolution_dilation=6, batch_size=256)
```

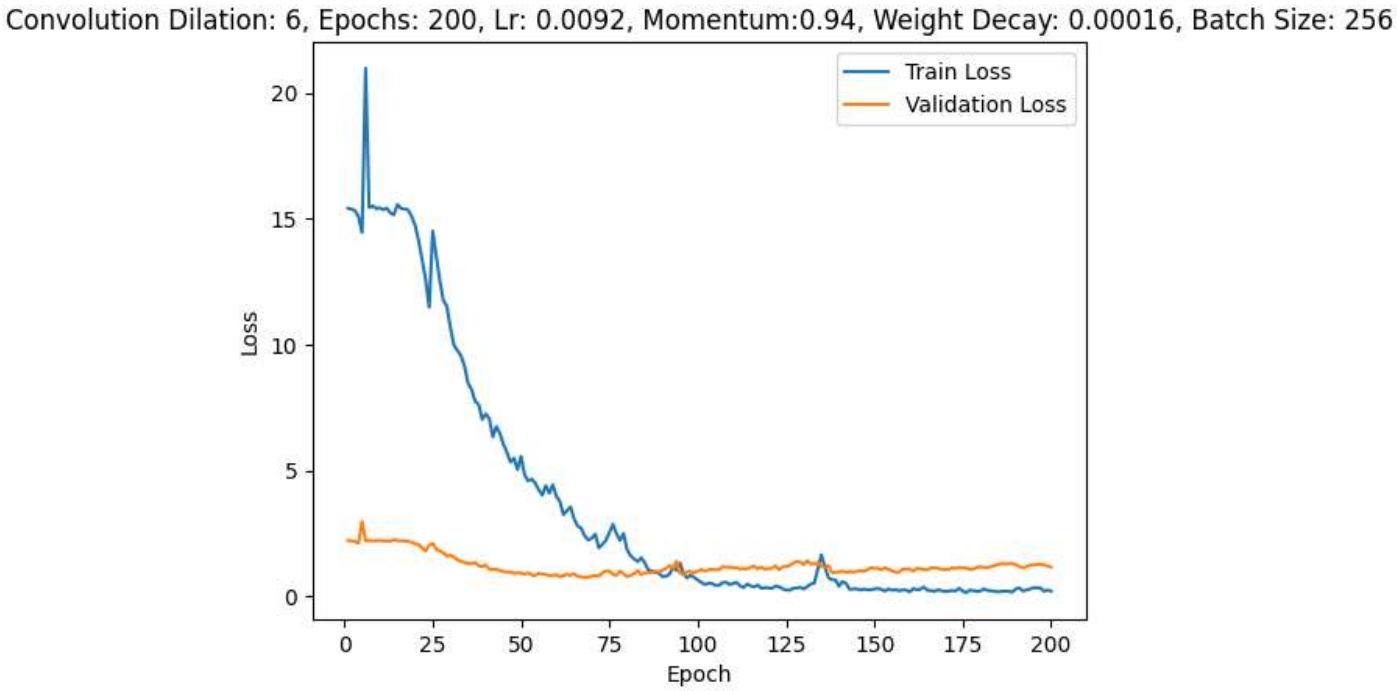
```
c:\ProgramData\Anaconda3\envs\pytorch-latest\lib\site-packages\torch\utils\data\dataset.py:342: UserWarning: Length of split at index 3 is 0.
This might result in an empty dataset.
  warnings.warn(f"Length of split at index {i} is 0. "

1554
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SimpleCNN                                [256, 9]                  --
├─Sequential: 1-1                        [256, 2, 10, 10]          --
│    └─Conv2d: 2-1                       [256, 12, 199, 199]       24,348
```

```
│    └─BatchNorm2d: 2-2                  [256, 12, 199, 199]       24
│    └─Mish: 2-3                         [256, 12, 199, 199]       --
│    └─AvgPool2d: 2-4                    [256, 12, 98, 98]         --
│    └─Mish: 2-5                         [256, 12, 98, 98]         --
│    └─Conv2d: 2-6                       [256, 8, 86, 86]          872
│    └─BatchNorm2d: 2-7                  [256, 8, 86, 86]          16
│    └─Mish: 2-8                         [256, 8, 86, 86]          --
│    └─MaxPool2d: 2-9                    [256, 8, 28, 28]          --
│    └─Mish: 2-10                        [256, 8, 28, 28]          --
│    └─Conv2d: 2-11                      [256, 4, 22, 22]          132
│    └─BatchNorm2d: 2-12                 [256, 4, 22, 22]          8
│    └─Mish: 2-13                        [256, 4, 22, 22]          --
│    └─Conv2d: 2-14                      [256, 2, 21, 21]          34
│    └─AvgPool2d: 2-15                   [256, 2, 10, 10]          --
│    └─Mish: 2-16                        [256, 2, 10, 10]          --
├─Flatten: 1-2                          [256, 200]                --
├─Dropout: 1-3                          [256, 200]                --
├─Linear: 1-4                           [256, 100]                20,100
├─Mish: 1-5                             [256, 100]                --
├─Dropout: 1-6                          [256, 100]                --
├─Linear: 1-7                           [256, 50]                 5,050
├─Mish: 1-8                             [256, 50]                 --
├─Dropout: 1-9                          [256, 50]                 --
├─Linear: 1-10                          [256, 50]                 2,550
├─Mish: 1-11                            [256, 50]                 --
├─Dropout: 1-12                         [256, 50]                 --
├─Linear: 1-13                          [256, 100]                5,100
├─Mish: 1-14                            [256, 100]                --
├─Dropout: 1-15                         [256, 100]                --
├─Linear: 1-16                          [256, 9]                  909
==========================================================================================
Total params: 59,143
Trainable params: 59,143
Non-trainable params: 0
Total mult-adds (G): 248.52
==========================================================================================
Input size (MB): 154.14
Forward/backward pass size (MB): 2199.19
Params size (MB): 0.24
Estimated Total Size (MB): 2353.57
==========================================================================================
Epoch 0 stats: train loss 15.410856246948242, train acc 0.10296010226011276, val loss 2.208543300628662, val acc 0.09009009599685669
Epoch 10 stats: train loss 15.358366012573242, train acc 0.12226512283086777, val loss 2.2029340267181396, val acc 0.0855855867266655
Epoch 20 stats: train loss 14.124900817871094, train acc 0.2535392642021179, val loss 2.054001569747925, val acc 0.1621621698141098
Epoch 30 stats: train loss 9.969949722290039, train acc 0.49227800965309143, val loss 1.566210389137268, val acc 0.45495495200157166
Epoch 40 stats: train loss 7.062961578369141, train acc 0.6615186929702759, val loss 1.0672508478164673, val acc 0.639639675617218
Epoch 50 stats: train loss 4.820860862731934, train acc 0.7535392642021179, val loss 0.8818532824516296, val acc 0.7027027010917664
Epoch 60 stats: train loss 3.752593994140625, train acc 0.800514817237854, val loss 0.7705581188201904, val acc 0.7387387752532959
Epoch 70 stats: train loss 2.4686899185180664, train acc 0.8790218830108643, val loss 0.8243122100830078, val acc 0.7702702879905701
Epoch 80 stats: train loss 1.6108237504959106, train acc 0.931145429611206, val loss 0.8273671865463257, val acc 0.7792792916297913
Epoch 90 stats: train loss 0.7963055968284607, train acc 0.9646074771881104, val loss 1.102271318435669, val acc 0.7702702879905701
Epoch 100 stats: train loss 0.549174427986145, train acc 0.9736164808273315, val loss 1.0499085187911987, val acc 0.7612612843513489
Epoch 110 stats: train loss 0.5470301508903503, train acc 0.9813385009765625, val loss 1.1321996450424194, val acc 0.7837837934494019
Epoch 120 stats: train loss 0.3106619119644165, train acc 0.9897040128707886, val loss 1.1043915748596191, val acc 0.8063063025474548
Epoch 130 stats: train loss 0.3857925534248352, train acc 0.983912467956543, val loss 1.4123426675796509, val acc 0.7882882952690125
Epoch 140 stats: train loss 0.5862484574317932, train acc 0.9781209826469421, val loss 0.990760326385498, val acc 0.792792797088623
Epoch 150 stats: train loss 0.3087614178657532, train acc 0.9897040128707886, val loss 1.1066287755966187, val acc 0.8108108043670654
Epoch 160 stats: train loss 0.30867502093315125, train acc 0.9884170293807983, val loss 0.9956585764884949, val acc 0.7882882952690125
Epoch 170 stats: train loss 0.20069004595279694, train acc 0.9897040128707886, val loss 1.051531195640564, val acc 0.824324369430542
Epoch 180 stats: train loss 0.28945574164390564, train acc 0.9890605211257935, val loss 1.1408401727676392, val acc 0.792792797088623
Epoch 190 stats: train loss 0.3283458650112152, train acc 0.9826254844665527, val loss 1.1753385066986084, val acc 0.8063063025474548
```

```
plot_train_curves(loss_dict_dilation)
plot_val_curves(loss_dict_dilation)
```





My second experiment is the learning rate. It seems that so far, a good learning rate is roughly around 0.009. The tricky part is that it seems to need a good initial push (requiring a higher learning rate and momentum), but then needs a lower learning rate and momentum for the later epochs, otherwise it "bounces" around the loss landscape. I have implemented a rudimentary learning rate scheduler. It sets a lower learning rate and lower momentum after some number of epochs.

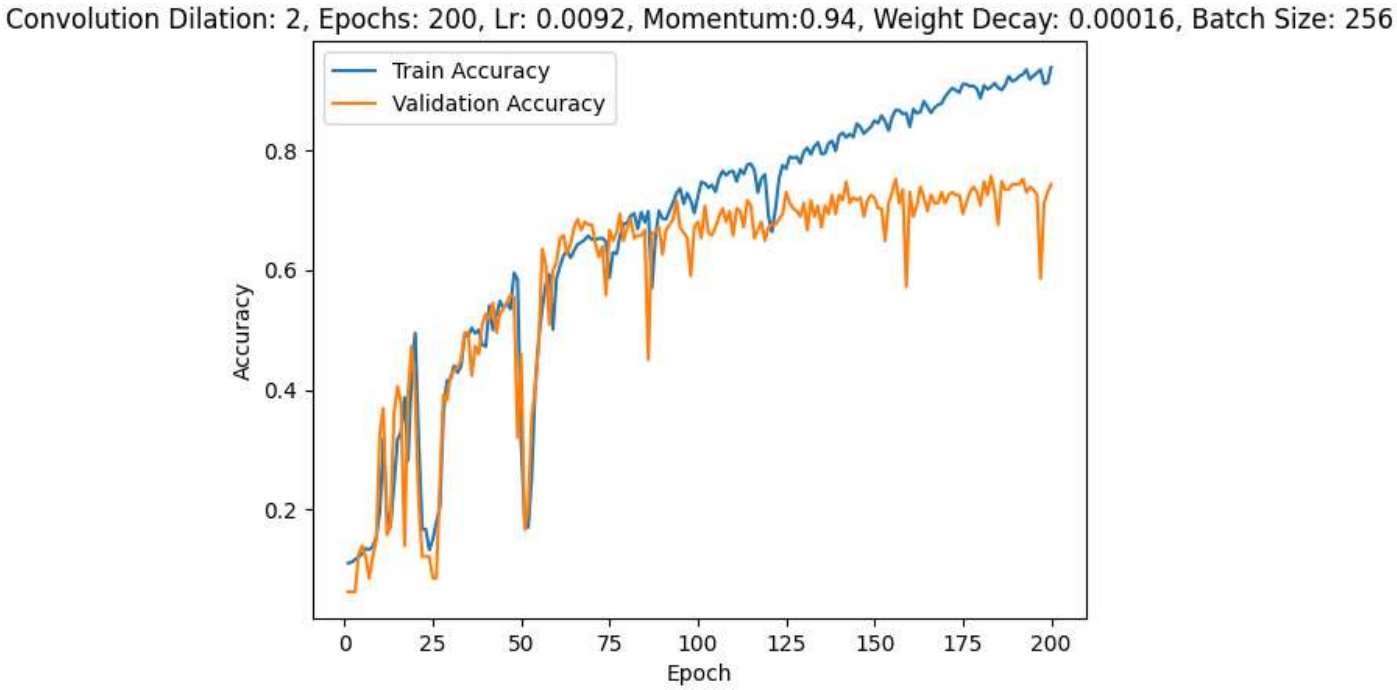
```
trained_network_lr_change, loss_dict_lr, _ = training_wrapper(split_sets, epochs=200, learning_rate = 0.0092, convolution_dilation=2, batch_siz
```

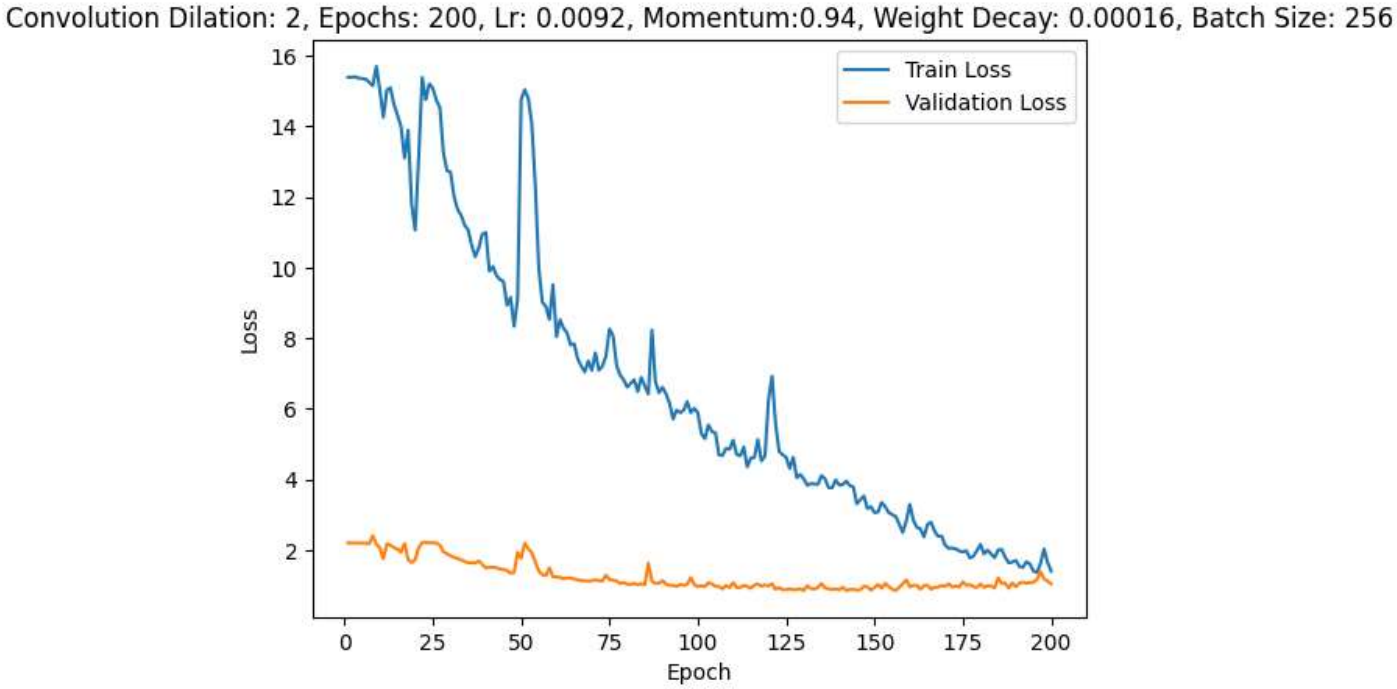
```
c:\ProgramData\Anaconda3\envs\pytorch-latest\lib\site-packages\torch\utils\data\dataset.py:342: UserWarning: Length of split at index 3 is 0.
This might result in an empty dataset.
  warnings.warn(f"Length of split at index {i} is 0. "

1554
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SimpleCNN                                [256, 9]                  --
```

```
├─Sequential: 1-1                        [256, 2, 14, 14]          --
│    └─Conv2d: 2-1                       [256, 12, 199, 199]       24,348
│    └─BatchNorm2d: 2-2                  [256, 12, 199, 199]       24
│    └─Mish: 2-3                         [256, 12, 199, 199]       --
│    └─AvgPool2d: 2-4                    [256, 12, 98, 98]         --
│    └─Mish: 2-5                         [256, 12, 98, 98]         --
│    └─Conv2d: 2-6                       [256, 8, 94, 94]          872
│    └─BatchNorm2d: 2-7                  [256, 8, 94, 94]          16
│    └─Mish: 2-8                         [256, 8, 94, 94]          --
│    └─MaxPool2d: 2-9                    [256, 8, 31, 31]          --
│    └─Mish: 2-10                        [256, 8, 31, 31]          --
│    └─Conv2d: 2-11                      [256, 4, 29, 29]          132
│    └─BatchNorm2d: 2-12                 [256, 4, 29, 29]          8
│    └─Mish: 2-13                        [256, 4, 29, 29]          --
│    └─Conv2d: 2-14                      [256, 2, 28, 28]          34
│    └─AvgPool2d: 2-15                   [256, 2, 14, 14]          --
│    └─Mish: 2-16                        [256, 2, 14, 14]          --
├─Flatten: 1-2                           [256, 392]                --
├─Dropout: 1-3                           [256, 392]                --
├─Linear: 1-4                            [256, 196]                77,028
├─Mish: 1-5                              [256, 196]                --
├─Dropout: 1-6                           [256, 196]                --
├─Linear: 1-7                            [256, 98]                 19,306
├─Mish: 1-8                              [256, 98]                 --
├─Dropout: 1-9                           [256, 98]                 --
├─Linear: 1-10                           [256, 98]                 9,702
├─Mish: 1-11                             [256, 98]                 --
├─Dropout: 1-12                          [256, 98]                 --
├─Linear: 1-13                           [256, 196]                19,404
├─Mish: 1-14                             [256, 196]                --
├─Dropout: 1-15                          [256, 196]                --
├─Linear: 1-16                           [256, 9]                  1,773
==========================================================================================
Total params: 152,647
Trainable params: 152,647
Non-trainable params: 0
Total mult-adds (G): 248.88
==========================================================================================
Input size (MB): 154.14
Forward/backward pass size (MB): 2254.22
Params size (MB): 0.61
Estimated Total Size (MB): 2408.97
==========================================================================================
Epoch 0 stats: train loss 15.392077445983887, train acc 0.11068211495876312, val loss 2.204162359237671, val acc 0.06306306272745132
Epoch 10 stats: train loss 14.260391235351562, train acc 0.31788933277130127, val loss 1.7526401281356812, val acc 0.36936938762664795
Epoch 20 stats: train loss 13.132797241210938, train acc 0.3133848309516907, val loss 2.06935715675354, val acc 0.21171171963214874
Epoch 30 stats: train loss 12.021281242370605, train acc 0.44015443325042725, val loss 1.792861819267273, val acc 0.4324324429035187
Epoch 40 stats: train loss 9.898470878601074, train acc 0.5405405759811401, val loss 1.5124071836471558, val acc 0.5180180072784424
Epoch 50 stats: train loss 15.038697242736816, train acc 0.18468469381332397, val loss 2.195452928543091, val acc 0.1666666716337204
Epoch 60 stats: train loss 8.527408599853516, train acc 0.6061776280403137, val loss 1.228888988494873, val acc 0.6531531810760498
Epoch 70 stats: train loss 7.589509010314941, train acc 0.6512226462364197, val loss 1.1624897718429565, val acc 0.6441441774368286
Epoch 80 stats: train loss 6.721262454986572, train acc 0.6904761791229248, val loss 1.0312669277191162, val acc 0.684684693813324
Epoch 90 stats: train loss 6.426058292388916, train acc 0.684684693813324, val loss 1.0359346866607666, val acc 0.6666666865348816
Epoch 100 stats: train loss 5.306894302368164, train acc 0.7471042275428772, val loss 0.9924043416976929, val acc 0.6531531810760498
Epoch 110 stats: train loss 4.711023330688477, train acc 0.7477477788925171, val loss 0.9397042989730835, val acc 0.7027027010917664
Epoch 120 stats: train loss 6.925260543823242, train acc 0.6634491682052612, val loss 1.0443801879882812, val acc 0.6756756901741028
Epoch 130 stats: train loss 3.8352866172790527, train acc 0.8037323355674744, val loss 0.9908797740936279, val acc 0.6666666865348816
Epoch 140 stats: train loss 3.8552117347717285, train acc 0.8288288116455078, val loss 0.9507279396057129, val acc 0.7162162065505981
Epoch 150 stats: train loss 3.080294132232666, train acc 0.8449163436889648, val loss 1.0254687070846558, val acc 0.7027027010917664
Epoch 160 stats: train loss 2.84724497795105, train acc 0.8687258958816528, val loss 1.0015403032302856, val acc 0.6891891956329346
Epoch 170 stats: train loss 2.051593780517578, train acc 0.8970398902893066, val loss 1.0477575063705444, val acc 0.7252252697944641
Epoch 180 stats: train loss 1.890684962272644, train acc 0.907979428768158, val loss 0.9463587403297424, val acc 0.7477477788925171
Epoch 190 stats: train loss 1.5381399393081665, train acc 0.9234234094619751, val loss 1.0599873065948486, val acc 0.7432432770729065
```

```
plot_train_curves(loss_dict_lr)
plot_val_curves(loss_dict_lr)
```





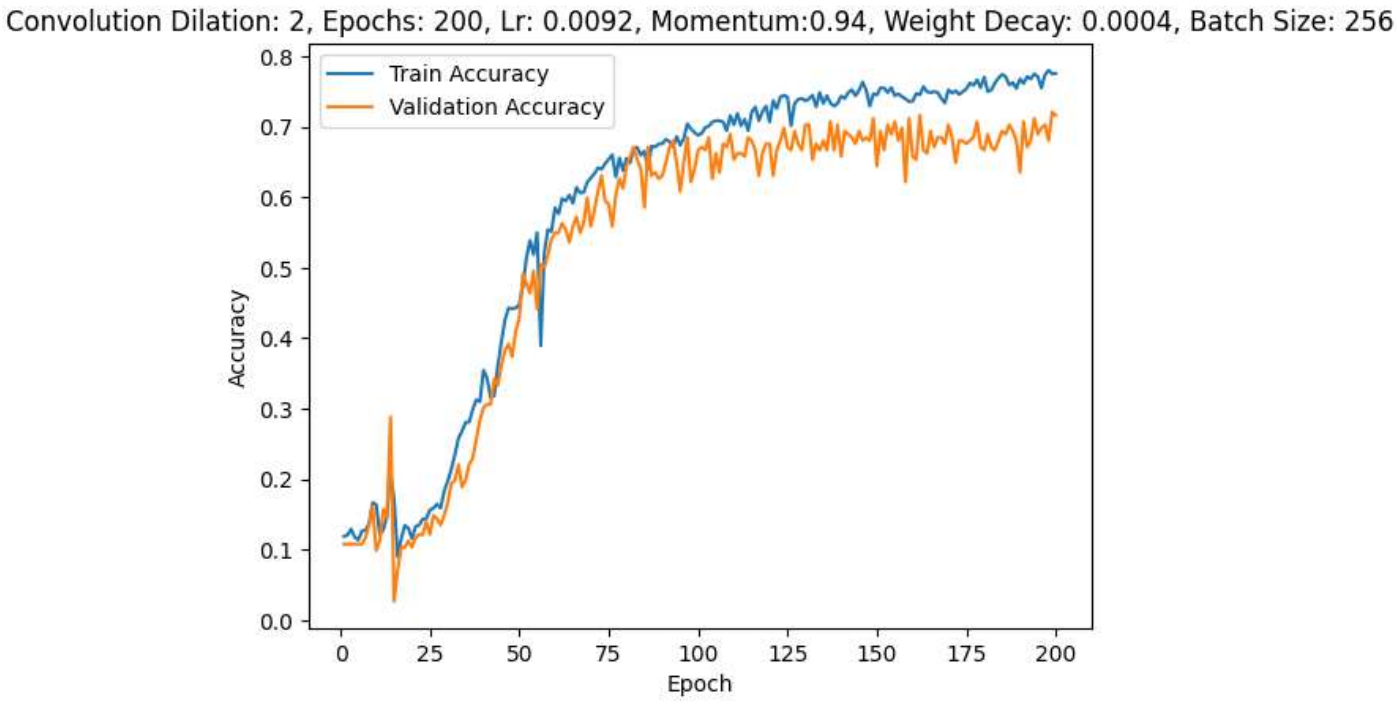
My third experiment is increasing the weight decay parameter since this could help us in finding a set of parameters for the model that corresponding to a well regularized model, which could help us mitigate possibly biases (e.g., inductive biases in making the model). There appear to be some overfitting towards the end (training accuracy is really high, while validation accuracy is still around 15% lower), so increasing the weight decay can help with regularizing the model.

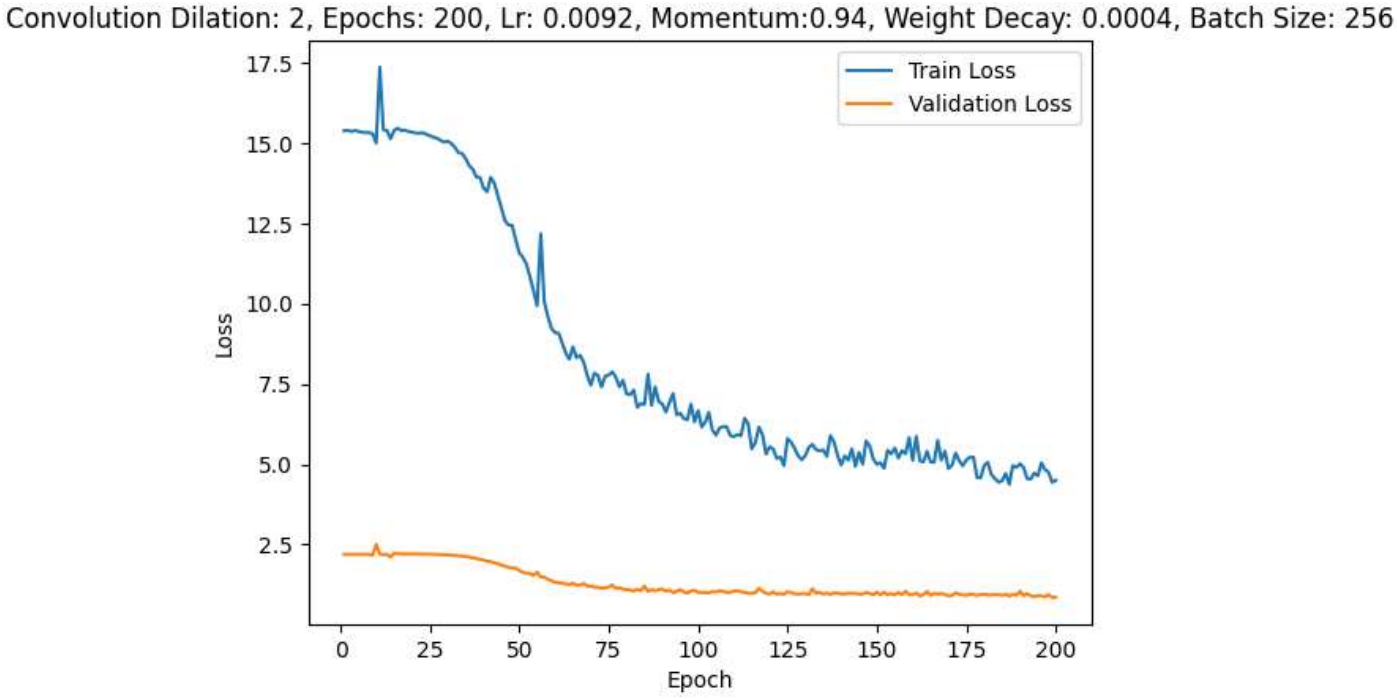
```
trained_network_wd_change, loss_dict_wd, _ = training_wrapper(split_sets, epochs=200, wd=4e-4, batch_size=256)
```

```
1554
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SimpleCNN                                [256, 9]                  --
├─Sequential: 1-1                        [256, 2, 14, 14]          --
│    └─Conv2d: 2-1                       [256, 12, 199, 199]       24,348
│    └─BatchNorm2d: 2-2                  [256, 12, 199, 199]       24
│    └─Mish: 2-3                         [256, 12, 199, 199]       --
│    └─AvgPool2d: 2-4                    [256, 12, 98, 98]         --
```

```
│    └─Mish: 2-5                        [256, 12, 98, 98]         --
│    └─Conv2d: 2-6                      [256, 8, 94, 94]          872
│    └─BatchNorm2d: 2-7                 [256, 8, 94, 94]          16
│    └─Mish: 2-8                        [256, 8, 94, 94]          --
│    └─MaxPool2d: 2-9                   [256, 8, 31, 31]          --
│    └─Mish: 2-10                       [256, 8, 31, 31]          --
│    └─Conv2d: 2-11                     [256, 4, 29, 29]          132
│    └─BatchNorm2d: 2-12                [256, 4, 29, 29]          8
│    └─Mish: 2-13                       [256, 4, 29, 29]          --
│    └─Conv2d: 2-14                     [256, 2, 28, 28]          34
│    └─AvgPool2d: 2-15                  [256, 2, 14, 14]          --
│    └─Mish: 2-16                       [256, 2, 14, 14]          --
├─Flatten: 1-2                         [256, 392]                --
├─Dropout: 1-3                         [256, 392]                --
├─Linear: 1-4                          [256, 196]                77,028
├─Mish: 1-5                            [256, 196]                --
├─Dropout: 1-6                         [256, 196]                --
├─Linear: 1-7                          [256, 98]                 19,306
├─Mish: 1-8                            [256, 98]                 --
├─Dropout: 1-9                         [256, 98]                 --
├─Linear: 1-10                         [256, 98]                 9,702
├─Mish: 1-11                           [256, 98]                 --
├─Dropout: 1-12                        [256, 98]                 --
├─Linear: 1-13                         [256, 196]                19,404
├─Mish: 1-14                           [256, 196]                --
├─Dropout: 1-15                        [256, 196]                --
├─Linear: 1-16                         [256, 9]                  1,773
==========================================================================================
Total params: 152,647
Trainable params: 152,647
Non-trainable params: 0
Total mult-adds (G): 248.88
==========================================================================================
Input size (MB): 154.14
Forward/backward pass size (MB): 2254.22
Params size (MB): 0.61
Estimated Total Size (MB): 2408.97
==========================================================================================
Epoch 0 stats: train loss 15.375836372375488, train acc 0.1190476194024086, val loss 2.197174549102783, val acc 0.10810811072587967
Epoch 10 stats: train loss 17.355548858642578, train acc 0.12162162363529205, val loss 2.203953742980957, val acc 0.11261261254549026
Epoch 20 stats: train loss 15.307247161865234, train acc 0.13256113231182098, val loss 2.205522298812866, val acc 0.11711712181568146
Epoch 30 stats: train loss 14.979476928710938, train acc 0.2142857164144516, val loss 2.17155385017395, val acc 0.19369369745254517
Epoch 40 stats: train loss 13.477688789367676, train acc 0.3429858386516571, val loss 1.9854182004928589, val acc 0.30630630254745483
Epoch 50 stats: train loss 11.441658020019531, train acc 0.47361648082733154, val loss 1.6328405141830444, val acc 0.4909909963607788
Epoch 60 stats: train loss 9.082146644592285, train acc 0.5765765905380249, val loss 1.3142846822738647, val acc 0.5495495796203613
Epoch 70 stats: train loss 7.835763454437256, train acc 0.6332046389579773, val loss 1.1715114116668701, val acc 0.5810810923576355
Epoch 80 stats: train loss 7.162685394287109, train acc 0.6486486792564392, val loss 1.0855984687805176, val acc 0.6576576828956604
Epoch 90 stats: train loss 6.6241841316223145, train acc 0.6821106672286987, val loss 1.0480635166168213, val acc 0.6486486792564392
Epoch 100 stats: train loss 6.159320831298828, train acc 0.6904761791229248, val loss 1.009700894355774, val acc 0.6711711883544922
Epoch 110 stats: train loss 5.926422595977783, train acc 0.7187902331352234, val loss 1.0666865110397339, val acc 0.662162184715271
Epoch 120 stats: train loss 5.472774028778076, train acc 0.7368082404136658, val loss 1.026951789855957, val acc 0.6306306719779968
Epoch 130 stats: train loss 5.53488302230835, train acc 0.739382266998291, val loss 0.9469945430755615, val acc 0.7027027010917664
Epoch 140 stats: train loss 5.256530284881592, train acc 0.7400257587432861, val loss 0.9621308445930481, val acc 0.6936936974525452
Epoch 150 stats: train loss 5.046815872192383, train acc 0.7548262476921082, val loss 0.9361984729766846, val acc 0.6936936974525452
Epoch 160 stats: train loss 5.8628387451171875, train acc 0.7471042275428772, val loss 0.9905980825424194, val acc 0.6531531810760498
Epoch 170 stats: train loss 4.965895652770996, train acc 0.7477477788925171, val loss 0.914722740650177, val acc 0.684684693813324
Epoch 180 stats: train loss 5.0589094161987305, train acc 0.7496782541275024, val loss 0.9293175339698792, val acc 0.6891891956329346
Epoch 190 stats: train loss 4.898757457733154, train acc 0.7612612843513489, val loss 0.9065548777580261, val acc 0.707207202911377
```

```
plot_train_curves(loss_dict_wd)
plot_val_curves(loss_dict_wd)
```





Finally, I will change the batch size to a much smaller batch size and combine all of these changes. Smaller batch sizes can help avoid overfitting as their gradients are computed more often from randomly sampled minibatches, and so the optimizer steps will be less homogenous (not always an evaluation of the gradient using the whole batch) with more of the weight decay effect (Li et al.) and can therefore help generalization more.

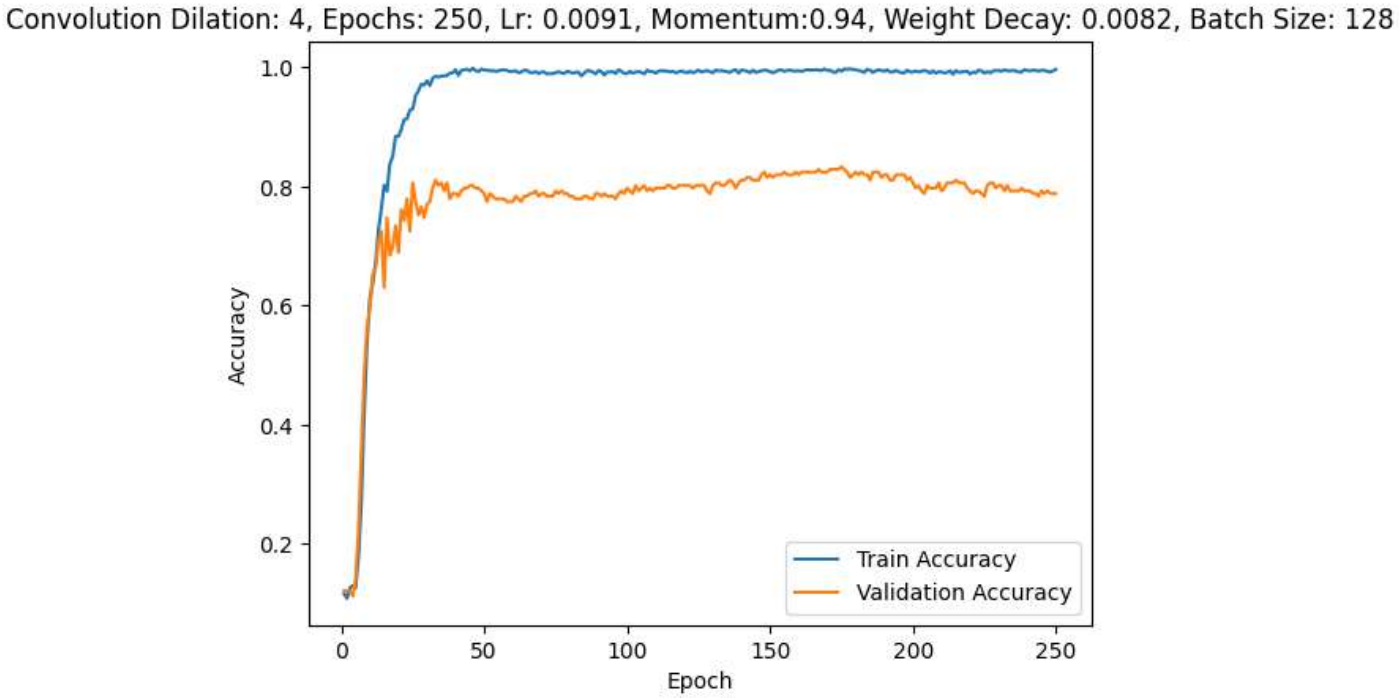
```
trained_network_all_change, loss_dict_all, _ = training_wrapper(split_sets, epochs=250, wd=8.2e-3, batch_size=128, learning_rate = 0.0091, conv
```

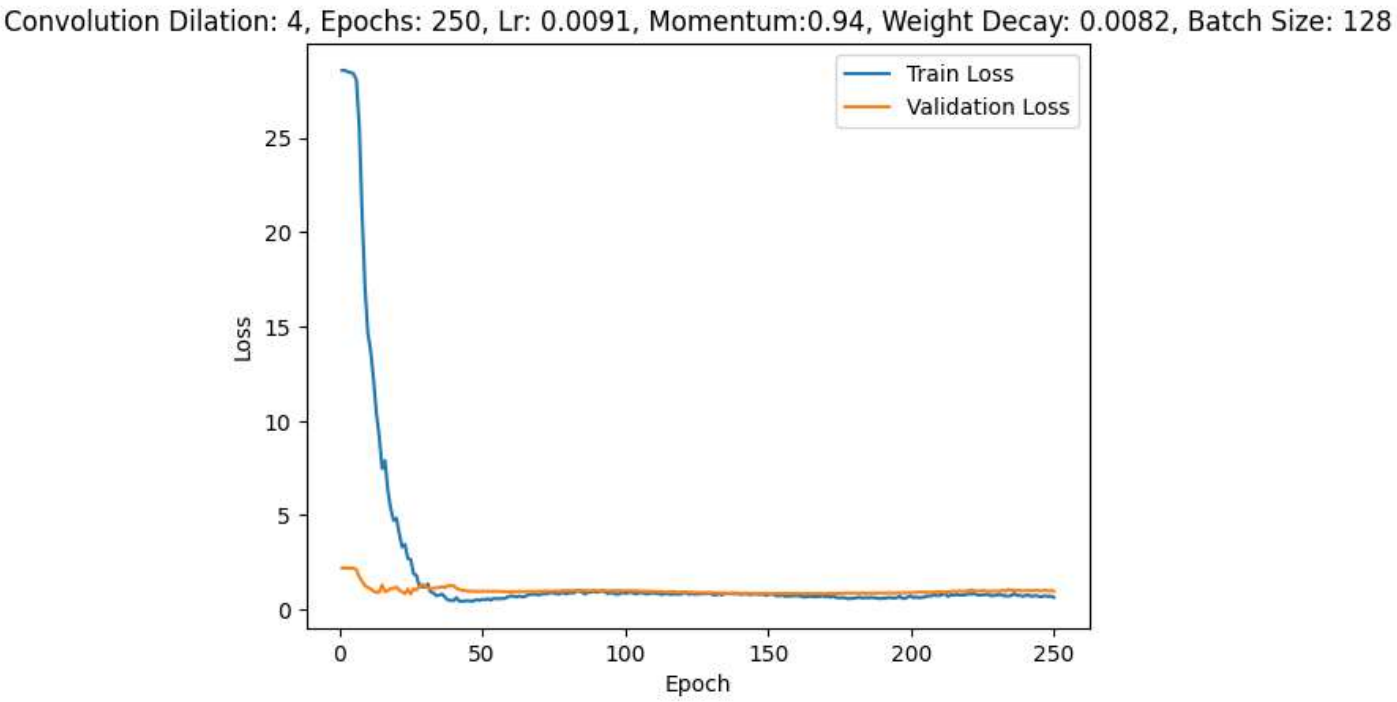
```
1554 222 443
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SimpleCNN                                [128, 9]                  --
├─Sequential: 1-1                        [128, 2, 12, 12]          --
│    └─Conv2d: 2-1                       [128, 12, 199, 199]       24,348
│    └─BatchNorm2d: 2-2                  [128, 12, 199, 199]       24
│    └─Mish: 2-3                         [128, 12, 199, 199]       --
```

```
│    └─AvgPool2d: 2-4                    [128, 12, 98, 98]         --
│    └─Mish: 2-5                         [128, 12, 98, 98]         --
│    └─Conv2d: 2-6                       [128, 8, 90, 90]          872
│    └─BatchNorm2d: 2-7                  [128, 8, 90, 90]          16
│    └─Mish: 2-8                         [128, 8, 90, 90]          --
│    └─MaxPool2d: 2-9                    [128, 8, 30, 30]          --
│    └─Mish: 2-10                        [128, 8, 30, 30]          --
│    └─Conv2d: 2-11                      [128, 4, 26, 26]          132
│    └─BatchNorm2d: 2-12                 [128, 4, 26, 26]          8
│    └─Mish: 2-13                        [128, 4, 26, 26]          --
│    └─Conv2d: 2-14                      [128, 2, 25, 25]          34
│    └─AvgPool2d: 2-15                   [128, 2, 12, 12]          --
│    └─Mish: 2-16                        [128, 2, 12, 12]          --
├─Flatten: 1-2                          [128, 288]                --
├─Dropout: 1-3                          [128, 288]                --
├─Linear: 1-4                           [128, 144]                41,616
├─Mish: 1-5                             [128, 144]                --
├─Dropout: 1-6                          [128, 144]                --
├─Linear: 1-7                           [128, 72]                 10,440
├─Mish: 1-8                             [128, 72]                 --
├─Dropout: 1-9                          [128, 72]                 --
├─Linear: 1-10                          [128, 72]                 5,256
├─Mish: 1-11                            [128, 72]                 --
├─Dropout: 1-12                         [128, 72]                 --
├─Linear: 1-13                          [128, 144]                10,512
├─Mish: 1-14                            [128, 144]                --
├─Dropout: 1-15                         [128, 144]                --
├─Linear: 1-16                          [128, 9]                  1,305
==========================================================================================
Total params: 94,563
Trainable params: 94,563
Non-trainable params: 0
Total mult-adds (G): 124.35
==========================================================================================
Input size (MB): 77.07
Forward/backward pass size (MB): 1113.21
Params size (MB): 0.38
Estimated Total Size (MB): 1190.66
==========================================================================================
Epoch 0 stats: train loss 28.5841064453125, train acc 0.11647361516952515, val loss 2.197230815887451, val acc 0.12162162363529205
Epoch 5 stats: train loss 28.08121109008789, train acc 0.17696267366409302, val loss 2.1167025566101074, val acc 0.22072072327136993
Epoch 10 stats: train loss 13.696343421936035, train acc 0.6370656490325928, val loss 1.0898371934890747, val acc 0.6531531810760498
Epoch 15 stats: train loss 7.898561000823975, train acc 0.7921493053436279, val loss 0.9454725980758667, val acc 0.7477477788925171
Epoch 20 stats: train loss 4.0260210037231445, train acc 0.8957529067993164, val loss 1.024498701095581, val acc 0.7612612843513489
Epoch 25 stats: train loss 1.8948074579238892, train acc 0.9523809552192688, val loss 1.0952352285385132, val acc 0.7747747898101807
Epoch 30 stats: train loss 1.3530230522155762, train acc 0.9697554707527161, val loss 1.1306430101394653, val acc 0.7747747898101807
Epoch 35 stats: train loss 0.8174170851707458, train acc 0.9858430027961731, val loss 1.2155072689056396, val acc 0.792792797088623
Epoch 40 stats: train loss 0.6318714022636414, train acc 0.9864864945411682, val loss 1.1263660192489624, val acc 0.7837837934494019
Epoch 45 stats: train loss 0.448335736989975, train acc 0.9987130165100098, val loss 0.9709597826004028, val acc 0.8018018007278442
Epoch 50 stats: train loss 0.5118060111999512, train acc 0.9954954981803894, val loss 0.9614671468734741, val acc 0.7747747898101807
Epoch 55 stats: train loss 0.5662422180175781, train acc 0.9954954981803894, val loss 0.9542304277420044, val acc 0.7792792916297913
Epoch 60 stats: train loss 0.7057164311408997, train acc 0.9942085146903992, val loss 0.9396142959594727, val acc 0.7837837934494019
Epoch 65 stats: train loss 0.7856013774871826, train acc 0.9916344881057739, val loss 0.9598453044891357, val acc 0.7882882952690125
Epoch 70 stats: train loss 0.8427558541297913, train acc 0.992277979850769, val loss 0.9707479476928711, val acc 0.7882882952690125
Epoch 75 stats: train loss 0.8486727476119995, train acc 0.9929215312004089, val loss 0.9790038466453552, val acc 0.7882882952690125
Epoch 80 stats: train loss 0.9017593860626221, train acc 0.9903475046157837, val loss 1.0021302700042725, val acc 0.7837837934494019
Epoch 85 stats: train loss 0.820013701915741, train acc 0.9948520064353943, val loss 0.9930695295333862, val acc 0.7837837934494019
Epoch 90 stats: train loss 0.9362152814865112, train acc 0.9929215312004089, val loss 0.9748563766479492, val acc 0.7837837934494019
Epoch 95 stats: train loss 0.8360600471496582, train acc 0.9897040128707886, val loss 0.9916162490844727, val acc 0.7792792916297913
Epoch 100 stats: train loss 0.8737056851387024, train acc 0.993565022945404, val loss 0.9981989860534668, val acc 0.792792797088623
Epoch 105 stats: train loss 0.8896049857139587, train acc 0.9890605211257935, val loss 0.9664173722267151, val acc 0.8018018007278442
Epoch 110 stats: train loss 0.8476203680038452, train acc 0.993565022945404, val loss 0.9546485543251038, val acc 0.7972972989082336
Epoch 115 stats: train loss 0.7963151931762695, train acc 0.993565022945404, val loss 0.9433620572090149, val acc 0.7972972989082336
Epoch 120 stats: train loss 0.9155631065368652, train acc 0.9909909963607788, val loss 0.908484697341919, val acc 0.8018018007278442
Epoch 125 stats: train loss 0.8317790031433105, train acc 0.992277979850769, val loss 0.8964632153511047, val acc 0.8018018007278442
Epoch 130 stats: train loss 0.7536457180976868, train acc 0.9954954981803894, val loss 0.8778294324874878, val acc 0.8063063025474548
Epoch 135 stats: train loss 0.8332047462463379, train acc 0.9916344881057739, val loss 0.8602469563484192, val acc 0.8108108043670654
Epoch 140 stats: train loss 0.8106679916381836, train acc 0.9948520064353943, val loss 0.8562514185905457, val acc 0.8108108043670654
Epoch 145 stats: train loss 0.8192512392997742, train acc 0.9929215312004089, val loss 0.8472617864608765, val acc 0.8108108043670654
Epoch 150 stats: train loss 0.8533034920692444, train acc 0.9909909963607788, val loss 0.8444656729698181, val acc 0.815315306186676
Epoch 155 stats: train loss 0.7061498761177063, train acc 0.9948520064353943, val loss 0.8536790013313293, val acc 0.8198198676109314
Epoch 160 stats: train loss 0.7325777411460876, train acc 0.993565022945404, val loss 0.8460732102394104, val acc 0.824324369430542
Epoch 165 stats: train loss 0.6764543652534485, train acc 0.9961389899253845, val loss 0.8461742997169495, val acc 0.824324369430542
```

```
Epoch 170 stats: train loss 0.6843701004981995, train acc 0.9954954981803894, val loss 0.8520249128341675, val acc 0.8288288712501526
Epoch 175 stats: train loss 0.6204771995544434, train acc 0.9974260330200195, val loss 0.8570889830589294, val acc 0.8288288712501526
Epoch 180 stats: train loss 0.6215803027153015, train acc 0.9948520064353943, val loss 0.8668225407600403, val acc 0.8198198676109314
Epoch 185 stats: train loss 0.6371103525161743, train acc 0.9916344881057739, val loss 0.8665729761123657, val acc 0.824324369430542
Epoch 190 stats: train loss 0.5834133625030518, train acc 0.9961389899253845, val loss 0.861064612865448, val acc 0.8198198676109314
Epoch 195 stats: train loss 0.6988312602043152, train acc 0.9909909963607788, val loss 0.8686079978942871, val acc 0.8198198676109314
Epoch 200 stats: train loss 0.618105947971344, train acc 0.9942085146903992, val loss 0.8954548835754395, val acc 0.7972972989082336
Epoch 205 stats: train loss 0.7293776273727417, train acc 0.9903475046157837, val loss 0.9058334827423096, val acc 0.7972972989082336
Epoch 210 stats: train loss 0.7890384793281555, train acc 0.9909909963607788, val loss 0.9388367533683777, val acc 0.7972972989082336
Epoch 215 stats: train loss 0.7962871193885803, train acc 0.9916344881057739, val loss 0.9520527720451355, val acc 0.8063063025474548
Epoch 220 stats: train loss 0.7987181544303894, train acc 0.992277979850769, val loss 1.041548252105713, val acc 0.7882882952690125
Epoch 225 stats: train loss 0.7861189842224121, train acc 0.9929215312004089, val loss 0.9843763709068298, val acc 0.8018018007278442
Epoch 230 stats: train loss 0.8056790232658386, train acc 0.993565022945404, val loss 0.9641956090927124, val acc 0.8018018007278442
Epoch 235 stats: train loss 0.8469924926757812, train acc 0.9942085146903992, val loss 1.0313202142715454, val acc 0.792792797088623
Epoch 240 stats: train loss 0.774444043636322, train acc 0.9942085146903992, val loss 1.0033396482467651, val acc 0.792792797088623
Epoch 245 stats: train loss 0.6848770976066589, train acc 0.9948520064353943, val loss 0.9828683137893677, val acc 0.7882882952690125
```

```
plot_train_curves(loss_dict_all)
plot_val_curves(loss_dict_all)
```





Part (c) - 2 pt

Choose the best model out of all the ones that you have trained. Justify your choice.

I choose my best model to be the combined model that has changes to all of the tested hyperparameters primarily because it has the best validation performance, i.e., $\approx 87\%$ accuracy on the validation set, whereas the best validation accuracies from the other models were approximately 75%. This model in particular used a smaller minibatch size, meaning that it was able to get more hetergeneous estimates of the true gradient. As such, it likely explored the loss surface more, and perhaps it has a more reliable set of parameters. In particular, large minibatch sizes (`batch_size` > 512) appear to converge to sharp minima according to Keskar et al., which can lead to worse generalization performance.

Part (d) - 2 pt

Report the test accuracy of your best model. You should only do this step once and prior to this step you should have only used the training and validation data.

```
model_state_path = "./models/valacc_0.8333333730697632-convdial_4-lr_0.0091-momentum_0.94-batch_size_128-epoch_num_174.mdlckpt"
weight_loaded_model = SimpleCNN(conv_dilation=4)
weight_loaded_model.load_state_dict(torch.load(model_state_path))

_, _, test_loader = create_dataloaders(
    split_sets, use_cuda=True)
test_acc, test_correct, test_total, test_loss = compute_test_performance(weight_loaded_model, test_loader)
print(test_acc, test_loss)
```

```
0.9548532731376975 0.276869535446167
```

The test accuracy is $\approx 95\%$ (94.808%), so the model learned the data extremely well and is a very good classifier for the A to I ASL signs. Further, the model surprisingly performed better on the test set than the validation set.

4. Transfer Learning [15 pt]

For many image classification tasks, it is generally not a good idea to train a very large deep neural network model from scratch due to the enormous compute requirements and lack of sufficient amounts of training data.

One of the better options is to try using an existing model that performs a similar task to the one you need to solve. This method of utilizing a pre-trained network for other similar tasks is broadly termed **Transfer Learning**. In this assignment, we will use Transfer Learning to extract features from the hand gesture images. Then, train a smaller network to use these features as input and classify the hand gestures.

As you have learned from the CNN lecture, convolution layers extract various features from the images which get utilized by the fully connected layers for correct classification. AlexNet architecture played a pivotal role in establishing Deep Neural Nets as a go-to tool for image classification problems and we will use an ImageNet pre-trained AlexNet model to extract features in this assignment.

Part (a) - 5 pt

Here is the code to load the AlexNet network, with pretrained weights. When you first run the code, PyTorch will download the pretrained weights from the internet.

```
import torchvision.models
alexnet = torchvision.models.alexnet(pretrained=True)
alexnet = alexnet.to(device="cuda:0")
```

```
c:\ProgramData\Anaconda3\envs\pytorch-latest\lib\site-packages\torchvision\models\_utils.py:208: UserWarning: The parameter 'pretrained' is
deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
c:\ProgramData\Anaconda3\envs\pytorch-latest\lib\site-packages\torchvision\models\_utils.py:223: UserWarning: Arguments other than a weight
enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing
`weights=AlexNet_Weights.IMAGENET1K_V1`. You can also use `weights=AlexNet_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

The alexnet model is split up into two components: *alexnet.features* and *alexnet.classifier*. The first neural network component, *alexnet.features*, is used to compute convolutional features, which are taken as input in *alexnet.classifier*.

The neural network alexnet.features expects an image tensor of shape Nx3x224x224 as input and it will output a tensor of shape Nx256x6x6 . (N = batch size).

Compute the AlexNet features for each of your training, validation, and test data. Here is an example code snippet showing how you can compute the AlexNet features for some images (your actual code might be different):

```
# img = ... a PyTorch tensor with shape [N,3,224,224] containing hand images ...
train_loader, val_loader, test_loader = create_dataloaders(split_sets, batch_size=len(train_set))
train_embeds, val_embeds, test_embeds = None, None, None
train_targets, val_targets, test_targets = None, None, None
for i, batch in enumerate(train_loader):
    input, targets = batch
    input = input.to("cuda:0")
    targets = targets.to("cuda:0")
    train_embeds = alexnet.features(input)
    train_targets = targets
for i, batch in enumerate(val_loader):
    input, targets = batch
    input = input.to("cuda:0")
    targets = targets.to("cuda:0")
    val_embeds = alexnet.features(input)
    val_targets = targets
for i, batch in enumerate(test_loader):
    input, targets = batch
    input = input.to("cuda:0")
    targets = targets.to("cuda:0")
    test_embeds = alexnet.features(input)
    test_targets = targets
```

```
train_embeds.size(), val_embeds.size(), test_embeds.size()
```

```
(torch.Size([1554, 256, 6, 6]),
 torch.Size([222, 256, 6, 6]),
 torch.Size([443, 256, 6, 6]))
```

```
torch.save(train_embeds, "./train_embeds")
torch.save(val_embeds, "./val_embeds")
torch.save(test_embeds, "./test_embeds")
```

```
train_embeds = torch.load("./train_embeds")
val_embeds = torch.load("./val_embeds")
test_embeds = torch.load("./test_embeds")
```

Save the computed features. You will be using these features as input to your neural network in Part (b), and you do not want to re-compute the features every time. Instead, run *alexnet.features* once for each image, and save the result.

Part (b) - 3 pt

Build a convolutional neural network model that takes as input these AlexNet features, and makes a prediction. Your model should be a subclass of nn.Module.

Explain your choice of neural network architecture: how many layers did you choose? What types of layers did you use: fully-connected or convolutional? What about other decisions like pooling layers, activation functions, number of channels / hidden units in each layer?

Here is an example of how your model may be called:

```
class TransferCNN(nn.Module):
    def __init__(self, conv_dilation=2):
        super().__init__()
        self.feature_embeddings = nn.Sequential(
            nn.ConvTranspose2d(256, 128, 4, dilation=conv_dilation),
            nn.Mish(),
            nn.ConvTranspose2d(128, 128, 5),
            nn.BatchNorm2d(128),
            nn.MaxPool2d(2),
            nn.Mish(),
            nn.ConvTranspose2d(128, 128, 5),
            nn.BatchNorm2d(128),
            nn.Mish(),
            nn.ConvTranspose2d(128, 128, 4, dilation=2*conv_dilation),
            nn.BatchNorm2d(128),
            nn.MaxPool2d(2),
            nn.Mish(),
            nn.Conv2d(128, 128, 4, dilation=conv_dilation),
            nn.BatchNorm2d(128),
            nn.Mish(),
            nn.Conv2d(128, 64, 4),
            nn.Mish())
        # programmatically get feature embedding size
        self._feature_embed_size = list(self.feature_embeddings(torch.rand(1, 256, 6, 6)).detach().size())
        self.embedding_size = np.prod(self._feature_embed_size)
        # use dropout to regularize model
        self.dropout = nn.Dropout(p=0.05)
        self.flatten = nn.Flatten()

        # autoencoder like architecture - just remove reconstruction layers
        self.fc1 = nn.Linear(self.embedding_size, self.embedding_size//2)
        self.act1 = nn.Mish()
        self.fc2 = nn.Linear(self.embedding_size//2, self.embedding_size//4)
        self.act2 = nn.Mish()
        self.fc3 = nn.Linear(self.embedding_size//4, self.embedding_size//4)
        self.act3 = nn.Mish()
        self.fc4 = nn.Linear(self.embedding_size//4, self.embedding_size//2)
        self.act4 = nn.Mish()
        self.fc5 = nn.Linear(self.embedding_size//2, 9)

    def forward(self, input):
        # create latent space rep.
        z = self.flatten(self.feature_embeddings(input))
        z = self.act1(self.fc1(z))
        z = self.dropout(z)
        z = self.act2(self.fc2(z))
        z = self.dropout(z)
        z = self.act3(self.fc3(z))
        z = self.dropout(z)
        z = self.act4(self.fc4(z))
        z = self.dropout(z)
        out = self.fc5(z)
        return out
```

```
model = TransferCNN().to("cuda:0")
#summary(model, (1, 256, 6, 6), verbose=1)
```

```
model.eval()
output = model(train_embeds)
prob = F.softmax(output)
torch.max(prob, 1)[1].detach()
```

```
C:\Users\ericz\AppData\Local\Temp\ipykernel_24132\3956553545.py:3: UserWarning: Implicit dimension choice for softmax has been deprecated.
Change the call to include dim=X as an argument.
  prob = F.softmax(output)

tensor([7, 7, 7,  ..., 7, 7, 7], device='cuda:0')
```

I decided to still use a fairly deep network for this since I wanted the "fine tuning" to be substantial since AlexNet was trained on ImageNet, and ImageNet is a rather general image database. More specifically, the AlexNet representations are rather compressed (a 6×6 matrix per feature map, i.e., convolution output), and so I wanted to learn some additional less compressed representations. Thus, over the course of some layers, I aimed to expand the size of each feature map while moderately cutting down on the number of output channel for each feature map. I also used batch norms to help keep the scales of each convolution output to be normalized, while also using dilated convolutions to keep the model from focusing too much on local features, which may be a problem due to the compressed nature of the AlexNet feature representations (maps). I only used max pooling because I wanted the model to focus on the most important features since some of the uncompressed features learned from the really compressed representation may be meaningless to average over, and also I wanted to sort of compare average pooling to max pooling. Then for the linear layers, I once again used a similar autoencoder like architecture since it allows for the "distilling" of latent representations. This is similar to the logic of the max pooling I used in the convoluion layers. I used the `Mish` activation function for the same reasons I used it in the `SimpleCNN` model.

Part (c) - 5 pt

Train your new network, including any hyperparameter tuning. Plot and submit the training curve of your best model only.

Note: Depending on how you are caching (saving) your AlexNet features, PyTorch might still be tracking updates to the **AlexNet weights**, which we are not tuning. One workaround is to convert your AlexNet feature tensor into a numpy array, and then back into a PyTorch tensor.

```
def compute_loss(preds, targets):
    criterion = nn.CrossEntropyLoss()
    return criterion(preds, targets)


def transfer_train_loop(train_data, val_data, convolution_dilation=2, epochs=1000, learning_rate=0.0095, momentum=0.95, wd=1e-4, batch_size=Non
    torch.backends.cuda.matmul.allow_tf32 = True
    torch.backends.cudnn.allow_tf32 = True
    torch.set_float32_matmul_precision = "high"
    if use_cuda and torch.cuda.is_available():
        dev = "cuda:0"
    else:
        print("CUDA unavailable, training on CPU")
        dev = "CPU"
    device = torch.device(dev)

    network = TransferCNN(conv_dilation=convolution_dilation)
    network = network.to(device)
    optimizer = torch.optim.Adam(network.parameters(
    ), lr=learning_rate, weight_decay=wd)
    # use for Nvidia AMP training cycle
    scaler = torch.cuda.amp.GradScaler()

    if train_data is not None:
        summary(network, input_data=train_data[0], verbose=1, device=device)

    loss_dict = {"config": f"Convolution Dilation: {convolution_dilation}, Epochs: {epochs}, Lr: {learning_rate}, Momentum:{momentum}, Weight D
                 "train_loss": [], "val_loss": [],
                 "train_acc": [], "val_acc": []}

    # counter for determining checkpoints
    best_val_acc = 0.0

    def run_epoch():
        epoch_loss, val_loss = 0.0, 0.0
        train_correct, train_total = 0.0, 0.0
        val_correct, val_total = 0.0, 0.0

        # reenable train mode to enable dropout
```

```
        network.train()
        inputs, targets = train_data
        inputs = inputs.to(device, non_blocking=True)
        targets = targets.to(device, non_blocking=True)

        network.zero_grad(set_to_none=True)

        with torch.autocast(device_type='cuda', dtype=torch.float16):
            preds = network(inputs)
            loss = compute_loss(preds, targets)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        with torch.no_grad():
            # eval mode to disable dropout
            network.eval()
            epoch_loss += loss
            _, category_preds = torch.max(preds, 1)
            train_correct += (category_preds ==
                              targets).sum()
            train_total += preds.size()[0]
            loss_dict["train_loss"].append(loss)
            loss_dict["train_acc"].append(train_correct/train_total)

            inputs, targets = val_data
            inputs = inputs.to(device, non_blocking=True)
            targets = targets.to(device, non_blocking=True)
            with torch.autocast(device_type='cuda', dtype=torch.float16):
                preds = network(inputs)
                batch_val_loss = compute_loss(preds, targets)
            val_loss += batch_val_loss
            _, category_preds = torch.max(preds, 1)
            val_correct += (category_preds ==
                              targets).sum()
            val_total += preds.size()[0]
            loss_dict["val_loss"].append(batch_val_loss)
            loss_dict["val_acc"].append(val_correct/val_total)
        return epoch_loss, train_correct, train_total, val_loss, val_correct, val_total

    # the fancy TQDM progress bar slows down model training likely due to GPU -> CPU copies
    if use_tqdm:
        with trange(epochs, desc="Train epochs", unit="epoch") as train_bar:
            for epoch in train_bar:
                epoch_loss, train_correct, train_total, val_loss, val_correct, val_total = run_epoch()

                # rudimentary learning rate scheduler
                if use_lr_sched and epoch >= epoch_slow_lr_start:
                    for param in optimizer.param_groups:
                        param["lr"] = slow_lr
                        param["momentum"] = low_momentum

                # model check point based on validation accuracy
                if torch.round(val_correct/val_total, decimals=3) > best_val_acc and epoch >= epoch_save_start:
                    torch.save(network.state_dict(
                    ), model_path_prefix + f"valacc_{val_correct/val_total}-convdial_{convolution_dilation}-lr_{learning_rate}-momentum_{moment
                    best_val_acc = torch.round(val_correct/val_total, decimals=3)

                train_bar.set_postfix(epoch_loss=(epoch_loss).item(),
                                      train_acc=(train_correct/train_total).item(),
                                      train_correct=train_correct.item(),
                                      train_total=train_total,
                                      val_loss=val_loss.item(),
                                      val_acc=(val_correct/val_total).item())
    else:
        for epoch in range(epochs):
            # run the epoch
            epoch_loss, train_correct, train_total, val_loss, val_correct, val_total = run_epoch()

            # rudimentary learning rate scheduler
            if use_lr_sched and epoch >= epoch_slow_lr_start:
                for param in optimizer.param_groups:
```

```
                    param["lr"] = slow_lr
                    param["momentum"] = low_momentum

            # model check point based on validation accuracy
            if torch.round(val_correct/val_total, decimals=3) > best_val_acc and epoch >= epoch_save_start:
                torch.save(network.state_dict(
                ), model_path_prefix + f"valacc_{val_correct/val_total}-convdial_{convolution_dilation}-lr_{learning_rate}-momentum_{momentum}-
                best_val_acc = torch.round(val_correct/val_total, decimals=3)

            if epoch % print_every == 0:
                print(f"Epoch {epoch} stats: train loss {epoch_loss.item()}, train acc {(train_correct/train_total).item()}, val loss {val_loss

    return network, loss_dict


def compute_transfer_test_performance(network, test_data):
    test_correct, test_total = 0.0, 0.0
    test_loss = 0.0

    device = next(network.parameters()).device
    with torch.no_grad():
        # disable dropout for test time
        network.eval()
        inputs, targets = test_data
        inputs = inputs.to(device, non_blocking=True)
        targets = targets.to(device, non_blocking=True)
        preds = network(inputs)
        test_loss += compute_loss(preds, targets)
        _, category_preds = torch.max(preds, 1)
        test_correct += float((category_preds == targets).sum())
        test_total += len(preds)
    return test_correct/test_total, test_correct, test_total, test_loss
```

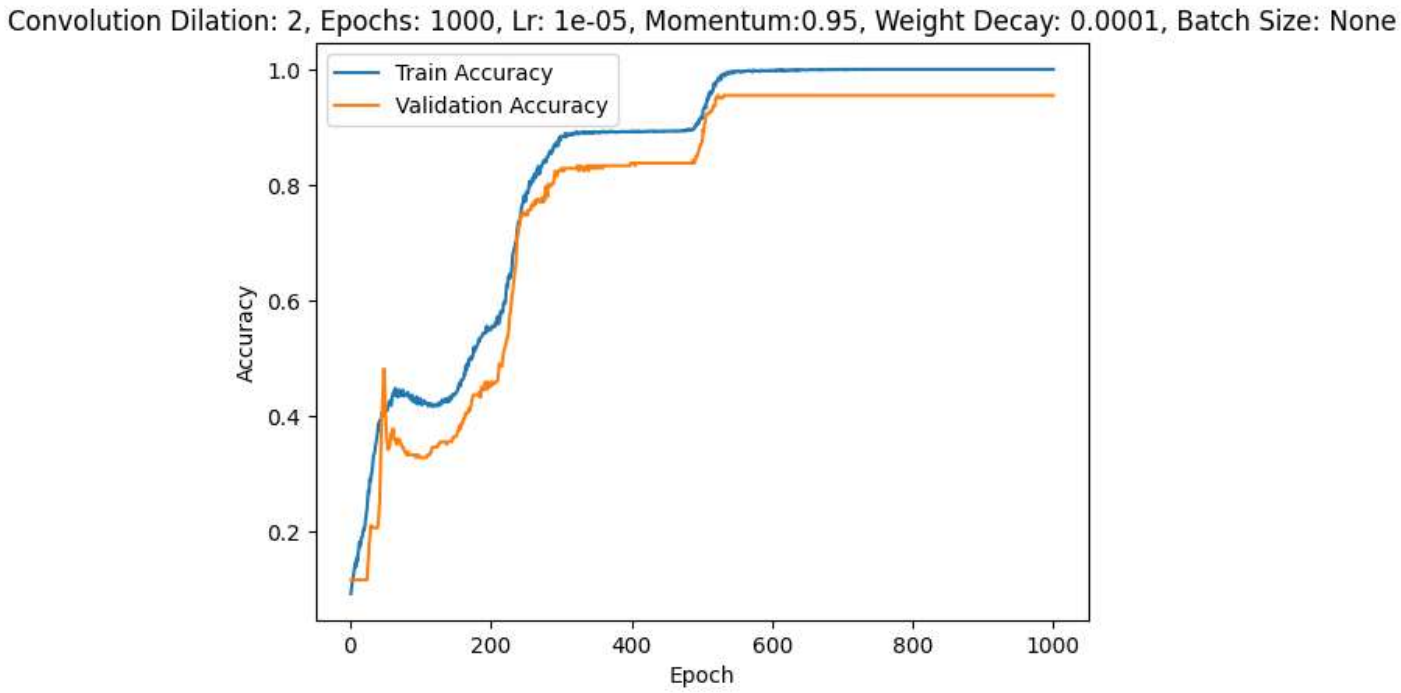
```
train_embeds, val_embeds = torch.tensor(train_embeds.detach()), torch.tensor(val_embeds.detach())
transfer_model, loss_dict_transfer = transfer_train_loop((train_embeds, train_targets), (val_embeds, val_targets), learning_rate=0.00001, use_t
```

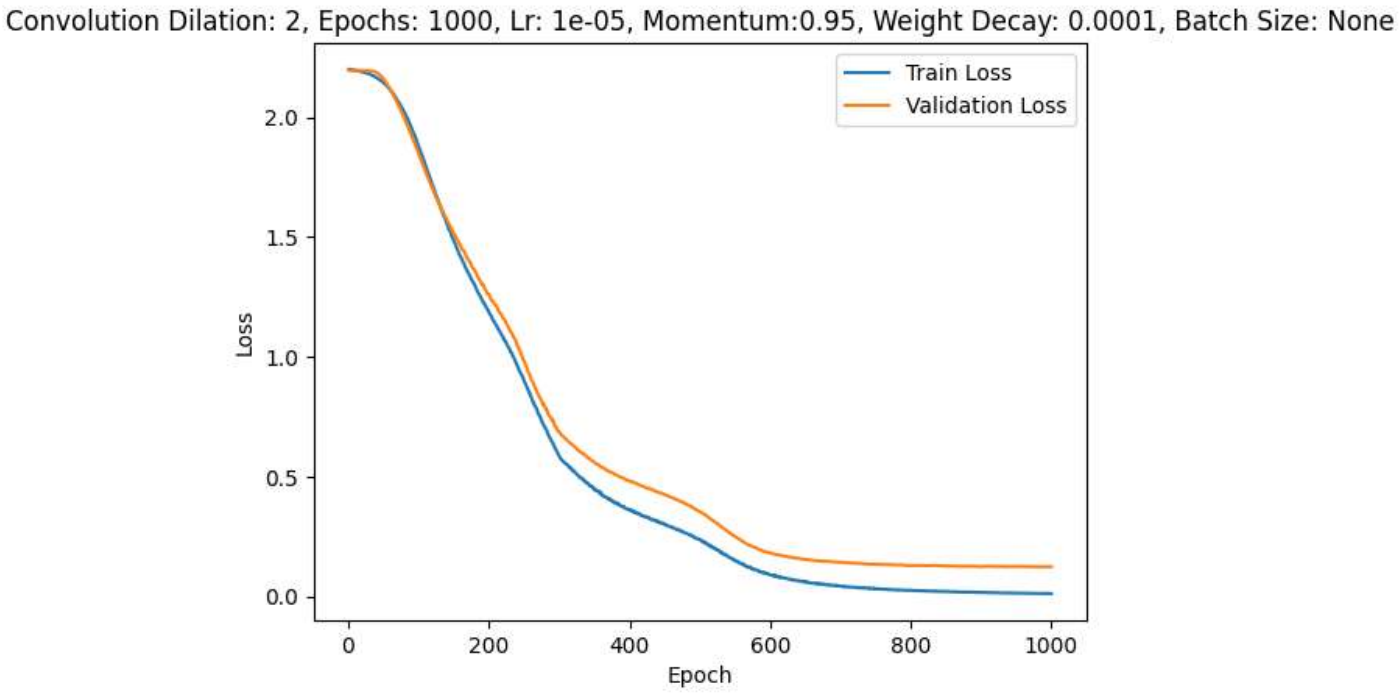
```
C:\Users\ericz\AppData\Local\Temp\ipykernel_25000\2895223788.py:1: UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  train_embeds, val_embeds = torch.tensor(train_embeds.detach()), torch.tensor(val_embeds.detach())

==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
TransferCNN                              [1554, 9]                 --
├─Sequential: 1-1                        [1554, 64, 3, 3]          --
│    └─ConvTranspose2d: 2-1              [1554, 128, 12, 12]       524,416
│    └─Mish: 2-2                         [1554, 128, 12, 12]       --
│    └─ConvTranspose2d: 2-3              [1554, 128, 16, 16]       409,728
│    └─BatchNorm2d: 2-4                  [1554, 128, 16, 16]       256
│    └─MaxPool2d: 2-5                    [1554, 128, 8, 8]         --
│    └─Mish: 2-6                         [1554, 128, 8, 8]         --
│    └─ConvTranspose2d: 2-7              [1554, 128, 12, 12]       409,728
│    └─BatchNorm2d: 2-8                  [1554, 128, 12, 12]       256
│    └─Mish: 2-9                         [1554, 128, 12, 12]       --
│    └─ConvTranspose2d: 2-10             [1554, 128, 24, 24]       262,272
│    └─BatchNorm2d: 2-11                 [1554, 128, 24, 24]       256
│    └─MaxPool2d: 2-12                   [1554, 128, 12, 12]       --
│    └─Mish: 2-13                        [1554, 128, 12, 12]       --
│    └─Conv2d: 2-14                      [1554, 128, 6, 6]         262,272
│    └─BatchNorm2d: 2-15                 [1554, 128, 6, 6]         256
│    └─Mish: 2-16                        [1554, 128, 6, 6]         --
│    └─Conv2d: 2-17                      [1554, 64, 3, 3]          131,136
│    └─Mish: 2-18                        [1554, 64, 3, 3]          --
├─Flatten: 1-2                           [1554, 576]               --
├─Linear: 1-3                            [1554, 288]               166,176
├─Mish: 1-4                              [1554, 288]               --
├─Dropout: 1-5                           [1554, 288]               --
├─Linear: 1-6                            [1554, 144]               41,616
├─Mish: 1-7                              [1554, 144]               --
├─Dropout: 1-8                           [1554, 144]               --
```

```
├─Linear: 1-9                            [1554, 144]               20,880
├─Mish: 1-10                             [1554, 144]               --
├─Dropout: 1-11                          [1554, 144]               --
├─Linear: 1-12                           [1554, 288]               41,760
├─Mish: 1-13                             [1554, 288]               --
├─Dropout: 1-14                          [1554, 288]               --
├─Linear: 1-15                           [1554, 9]                 2,601
==========================================================================================
Total params: 2,273,609
Trainable params: 2,273,609
Non-trainable params: 0
Total mult-adds (G): 623.73
==========================================================================================
Input size (MB): 57.29
Forward/backward pass size (MB): 3467.94
Params size (MB): 9.09
Estimated Total Size (MB): 3534.32
==========================================================================================
Train epochs: 100%|██████████| 1000/1000 [07:02<00:00,  2.36epoch/s, epoch_loss=0.0121, train_acc=1, train_correct=1554.0, train_total=1554.0,
val_acc=0.955, val_loss=0.124]
```

```
plot_train_curves(loss_dict_transfer)
plot_val_curves(loss_dict_transfer)
```





Part (d) - 2 pt

Report the test accuracy of your best model. How does the test accuracy compare to Part 3(d) without transfer learning?

```
transfer_model_state_path = "./models/transfer/valacc_0.9504504799842834-convdial_2-lr_1e-05-momentum_0.95-batch_size_None-epoch_num_519.mdlckp
transfer_weight_loaded_model = TransferCNN()
transfer_weight_loaded_model.load_state_dict(torch.load(transfer_model_state_path))

test_embeds = torch.tensor(test_embeds.clone().detach())
test_accuracy, _, _, test_loss = compute_transfer_test_performance(transfer_weight_loaded_model, (test_embeds, test_targets))
print(test_accuracy, test_loss.item())
```

```
C:\Users\ericz\AppData\Local\Temp\ipykernel_9564\1952339223.py:5: UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  test_embeds = torch.tensor(test_embeds.clone().detach())

0.9774266365688488 0.24418403208255768
```

The transfer learning model performed slightly better than the model from part 3(d), i.e., the model from 3(d) achieved $\approx 95\%$ (94.808%) accuracy on the test set while the transfer learning model achieved $\approx 98\%$ (97.516%) accuracy on the test set. So both models did exceptionally well!

5. Additional Testing [5 pt]

As a final step in testing we will be revisiting the sample images that you had collected and submitted at the start of this lab. These sample images should be untouched and will be used to demonstrate how well your model works at identifying your hand guestures.

Using the best transfer learning model developed in Part 4. Report the test accuracy on your sample images and how it compares to the test accuracy obtained in Part 4(d)? How well did your model do for the different hand guestures? Provide an explanation for why you think your model performed the way it did?

```
# Turn off shuffling so we get the same ordering of the test set.
_, _, small_test_loader = create_dataloaders(
    small_split_sets, use_cuda=True, shuffle=False)
small_test_embeds, small_test_targets = None, None

# will eval once since test set only has one batch
for i, batch in enumerate(small_test_loader):
    input, targets = batch
    input = input.to("cuda:0")
    targets = targets.to("cuda:0")
```

```
    small_test_embeds = alexnet.features(input)
    small_test_targets = targets

no_transfer_test_acc, _, _, _ = compute_test_performance(weight_loaded_model, small_test_loader)
transfer_test_acc, _, _, _ = compute_transfer_test_performance(transfer_weight_loaded_model, (small_test_embeds, small_test_targets))

print(no_transfer_test_acc, transfer_test_acc)

preds_no_transfer = None
preds_transfer = None

# will eval once since test set only has one batch
for i, batch in enumerate(small_test_loader):
    inputs, small_test_targets = batch
    weight_loaded_model.eval()
    transfer_weight_loaded_model.eval()
    _, preds_no_transfer = torch.max(weight_loaded_model(inputs), 1)
    _, preds_transfer = torch.max(transfer_weight_loaded_model(small_test_embeds.cpu()), 1)

accuracy_per_category = {"no_transfer": [0]*9, "transfer": [0]*9}
letters = [chr(i) for i in range(65, 74)]
for i in range(len(torch.unique(small_test_targets))):
    target_category_indexes = (i == small_test_targets) # where the class labels are for target i
    target_class_num = torch.sum(target_category_indexes)

    no_transfer_category_indexes = (i == preds_no_transfer)
    no_transfer_category_correct = torch.sum(torch.logical_and(target_category_indexes, no_transfer_category_indexes))

    transfer_category_indexes = (i == preds_transfer)
    transfer_category_correct = torch.sum(torch.logical_and(target_category_indexes, transfer_category_indexes))

    accuracy_per_category["no_transfer"][i] = no_transfer_category_correct/target_class_num
    accuracy_per_category["transfer"][i] = transfer_category_correct/target_class_num

    print(f"Category {letters[i]}, no transfer acc: {accuracy_per_category['no_transfer'][i]}, transfer acc: {accuracy_per_category['transfer']
```

```
0.9483870967741935 0.9741935483870968
Category A, no transfer acc: 0.9047619104385376, transfer acc: 0.9523809552192688
Category B, no transfer acc: 0.8947368264198303, transfer acc: 1.0
Category C, no transfer acc: 0.9523809552192688, transfer acc: 0.9523809552192688
Category D, no transfer acc: 1.0, transfer acc: 1.0
Category E, no transfer acc: 0.9473684430122375, transfer acc: 1.0
Category F, no transfer acc: 1.0, transfer acc: 1.0
Category G, no transfer acc: 1.0, transfer acc: 1.0
Category H, no transfer acc: 1.0, transfer acc: 1.0
Category I, no transfer acc: 0.8461538553237915, transfer acc: 0.8461538553237915
```

Overall, the transfer learning model has roughly $\approx 3\%$ better performance on this test set. It performs marginally better on letters A and E, while it performs quite a bit better than on the letter B. Otherwise, both models perform very comparably. This is possibly due to the feature representations (embeddings) computed from AlexNet since it has been trained on many more images and could maybe have learned something that would help it recognize the letter B better, especially given that "B" only has 247 images in the whole dataset.