HW2 Bootcamp
Logistics

• HW2P2 is **significantly harder** than HW1P2. Models will be harder to develop, train, and converge. Please start early!

• Models must be written yourself and trained from scratch.

• You may use CMU Virtual Andrew (8GB Nvidia L40, 32 GB RAM) for training.
Problem Statement

•**Face Classification**
  •Given an image, figure out which person it is.

•**Face Verification**
  •Given a set of images, figure out if they are of the same person.
Face Classification
Face Verification

Feature Extraction Model → Feature Embedding → Similarity metric → A similarity score

Feature Extraction Model → Feature Embedding → Similarity metric → A similarity score
Feature Verification

Unknown image

Known images

Feature Extraction Model

Feature Embedding

Feature Extraction Model

Feature Embedding

Cosine Similarity for each known image

A similarity score from -1 to 1

CLASS torch.nn.CosineSimilarity(dim=1, eps=1e-08) [SOURCE]

Returns cosine similarity between \( x_1 \) and \( x_2 \), computed along \( \text{dim} \).

\[
\text{similarity} = \frac{x_1 \cdot x_2}{\max(\|x_1\|_2 \cdot \|x_2\|_2, \epsilon)}.
\]
Workflow

• First train a strong classification model for the classification task.
• Then, for the verification task, use the model trained on classification.
  • take the penultimate features as feature embeddings of each image.
• You should additionally train verification-specific losses such as ArcFace, Triplet Loss to improve performance.
Building Blocks

Input Image + Transformations

Choice of Model

Training the model
Building Blocks

Input Image + Transformations

Choice of Model

Training the model
Color
Jitter

Original image
Random Perspective
Random Vertical Flip
Common Issue:

```python
TypeError: Input tensor should be a torch tensor. Got <class 'PIL.Image.Image'>.
```

→ Please check the sequencing of your transforms. Read the documentation and verify the kind of input required.
Building Blocks

Input Image + Transformation

Choice of Model

Training the model
Residual Connections

In traditional feedforward neural networks, data flows through each layer **sequentially**: The output of a layer is the input for the next layer. **Residual connection** provides another path for data to reach latter parts of the neural network by **skipping** some layers.
Residual Connections

- The residual connection first applies identity mapping to $x$.
- Then it performs element-wise addition $F(x) + x$.
- The whole architecture that takes an input $x$ and produces output $F(x) + x$ is usually called a residual block or a building block.
- Quite often, a residual block will also include an activation function such as ReLU applied to $F(x) + x$. 
How do they help??

- For feedforward neural networks, training a deep network is usually very difficult, due to problems such as **exploding gradients and vanishing gradients**.
- On the other hand, the training process of a neural network with residual connections is empirically shown to converge much more easily, even if the network has several hundreds layers.
- It is easier to learn Zero weights than an Identity mapping, if the residual connections aren’t present.
ResNet Block

- Remember that to understand a paper, we just really need to understand its **blocks**.
- ResNet proposes 2 blocks: BasicBlock & BottleneckBlock
- The key point is residual connection

![Residual learning: a building block](image-url)
ResNet: BasicBlock

- It’s just a regular 3x3 convolution (then BN, ReLU), another 3x3 convolution (then BN).
- Then, a skip connection adding input and output, then ReLU.
ResNet: BottleneckBlock

• A bit more involved.
• A 256-channel input goes through a point-wise convolution, reducing channels to 64.
• Then, a 3x3 regular convolution maintains channels at 64.
• Then, a point-wise convolution expands channels back to 256.
• Finally, the residual connection.
Basic and Bottleneck Block

Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on $56 \times 56$ feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.

Residual Connection - Basic Block

```python
class BasicBlock(torch.nn.Module):
    def __init__(self, n_h):
        self.linear0 = torch.nn.Linear(n_h, n_h)
        self.linear1 = torch.nn.Linear(n_h, n_h)
        self.bn0 = torch.nn.BatchNorm1d(n_h)
        self.bn1 = torch.nn.BatchNorm1d(n_h)
        self.relu = torch.nn.ReLU(inplace=True)

    def forward(self, A0):
        R0  = A0
        Z0  = self.linear0(A0)
        BZ0 = self.bn0(Z0)
        A1  = self.relu(BZ0)
        Z1  = self.linear1(A1)
        BZ1 = self.bn1(Z1)
        A2  = self.relu(BZ1 + R0)
        return A2
```
Residual Connection - Bottleneck Block

class Bottleneck(torch.nn.Module):
    def __init__(self, n_h):
        self.residual = torch.nn.Linear(n_h, n_h*4)
        self.linear0 = torch.nn.Linear(n_h, n_h)
        self.linear1 = torch.nn.Linear(n_h, n_h)
        self.linear2 = torch.nn.Linear(n_h, n_h*4)
        self.bn0 = torch.nn.BatchNorm1d(n_h)
        self.bn1 = torch.nn.BatchNorm1d(n_h)
        self.bn2 = torch.nn.BatchNorm1d(n_h*4)
        self.relu = torch.nn.ReLU(inplace=True)

    def forward(self, A0):
        R0 = self.residual(A0)
        Z0 = self.linear0(A0)
        BZ0 = self.bn0(Z0)
        A1 = self.relu(BZ0)
        Z1 = self.linear1(A1)
        BZ1 = self.bn1(Z1)
        A2 = self.relu(BZ1)
        Z2 = self.linear2(A2)
        BZ2 = self.bn2(Z2)
        A3 = self.relu(BZ2 + R0)
        return A3
Basic and Bottleneck Block

Figure 5. A deeper residual function $F$ for ImageNet. Left: a building block (on $56 \times 56$ feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.

Architectures

• At this point, you should have basic familiarity with convolutions as taught in lecture.

• Now, how do we take convolutions and assemble them into a strong architecture?

• We’ll cover three different types of blocks:
  • Resnet blocks
  • Convnext blocks
  • Mobile blocks
General Architecture Flow

• CNN architectures are divided into stages, which are divided into blocks.
  • Each “stage” consists of (almost) equivalent “blocks”
  • Each “block” consists of a few CNN layers, BN, and ReLUs.

• To understand an architecture, we mostly need to understand its blocks.

• All that changes for blocks in different stages is the base # of channels
General Architecture Flow

• However, you do need to piece these blocks together into a final model.

• The general flow is like this:
  • Stem
  • Stage 1
  • Stage 2
  • ...
  • Stage n
  • Classification Layer
General Architecture Flow

• The stem usually downsamples the input by 4x.

• Some stages do downsample. If they do, generally, the first convolution in the stage downsamples by 2x.

• When you downsample by 2x, you usually increase channel dimension by 2x.
  • So, later stages have smaller spatial resolution, higher # of channels
ResNet: Overall Architecture

<table>
<thead>
<tr>
<th>layer name</th>
<th>output size</th>
<th>18-layer</th>
<th>34-layer</th>
<th>50-layer</th>
<th>101-layer</th>
<th>152-layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>112×112</td>
<td>7×7, 64, stride 2</td>
<td>3×3 max pool, stride 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>conv2_x</td>
<td>56×56</td>
<td>[3×3, 64] ×2</td>
<td>[3×3, 64] ×3</td>
<td>[1×1, 64]</td>
<td>[1×1, 64]</td>
<td>[1×1, 64]</td>
</tr>
<tr>
<td>conv3_x</td>
<td>28×28</td>
<td>[3×3, 128] ×2</td>
<td>[3×3, 128] ×4</td>
<td>[1×1, 128]</td>
<td>[1×1, 128]</td>
<td>[1×1, 128]</td>
</tr>
<tr>
<td>conv4_x</td>
<td>14×14</td>
<td>[3×3, 256] ×2</td>
<td>[3×3, 256] ×6</td>
<td>[1×1, 256]</td>
<td>[1×1, 256]</td>
<td>[1×1, 256]</td>
</tr>
<tr>
<td>conv5_x</td>
<td>7×7</td>
<td>[3×3, 512] ×2</td>
<td>[3×3, 512] ×3</td>
<td>[1×1, 512]</td>
<td>[1×1, 512]</td>
<td>[1×1, 512]</td>
</tr>
<tr>
<td>1×1</td>
<td></td>
<td>average pool, 1000-d fc, softmax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOPs</td>
<td>1.8×10^9</td>
<td>3.6×10^9</td>
<td>3.8×10^9</td>
<td>7.6×10^9</td>
<td>11.3×10^9</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Sizes of outputs and convolutional kernels for ResNet 34
ConvNeXt: Block

- A 7x7 depth-wise convolution.
- A point-wise convolution increasing # of channels
- A point-wise convolution decreasing # of channels
- Residual Connection
ConvNeXt block

• This is a very new paper, a state-of-the-art architecture.
• However, its intuitions are very similar to MobileNetV2.
• Again, remember that to understand a paper, we just really need to understand its blocks.
• Just a single block type for ConvNeXt
• Read the paper for details on stages/channel sizes, etc.
  • We recommend ConvNeXt-T size which has less than 35M parameters.
ResNet vs ConvNeXt: Differences

• Note that ConvNeXt has fewer BN/ReLU
  • GELU is just more advanced ReLU
# ResNet vs ConvNeXt

<table>
<thead>
<tr>
<th></th>
<th>output size</th>
<th>ResNet-50</th>
<th>ConvNeXt-T</th>
</tr>
</thead>
<tbody>
<tr>
<td>stem</td>
<td>56×56</td>
<td>7×7, 64, stride 2</td>
<td>4×4, 96, stride 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3×3 max pool, stride 2</td>
<td></td>
</tr>
<tr>
<td>res2</td>
<td>56×56</td>
<td>[1×1, 64]</td>
<td>[d7×7, 96]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3×3, 64, 3×3] × 3</td>
<td>[1×1, 384] × 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1×1, 256]</td>
<td>[1×1, 96]</td>
</tr>
<tr>
<td>res3</td>
<td>28×28</td>
<td>[1×1, 128]</td>
<td>[d7×7, 192]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3×3, 128, 3×3] × 4</td>
<td>[1×1, 768] × 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1×1, 512]</td>
<td>[1×1, 192]</td>
</tr>
<tr>
<td>res4</td>
<td>14×14</td>
<td>[1×1, 256]</td>
<td>[d7×7, 384]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3×3, 256, 3×3] × 6</td>
<td>[1×1, 1536] × 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1×1, 1024]</td>
<td>[1×1, 384]</td>
</tr>
<tr>
<td>res5</td>
<td>7×7</td>
<td>[1×1, 512]</td>
<td>[d7×7, 768]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3×3, 512, 3×3] × 3</td>
<td>[1×1, 3072] × 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1×1, 2048]</td>
<td>[1×1, 768]</td>
</tr>
<tr>
<td>FLOPs</td>
<td></td>
<td>4.1×10⁹</td>
<td>4.5×10⁹</td>
</tr>
<tr>
<td># params.</td>
<td></td>
<td>25.6×10⁶</td>
<td>28.6×10⁶</td>
</tr>
</tbody>
</table>
Mobile Block

• The goal of MobileNet blocks is to be parameter efficient.
• They do so by making extensive use of depth-wise convolutions and point-wise convolutions.
A Normal Convolution

- Considering just a single output channel
A Normal Convolution (Another Diagram)

- Considering a single output channel
A Normal Convolution

- Considering all output channels
Depth-wise Convolutions

• Shorthand for “Depth-wise separable convolutions”
• “Depth”-wise separable, because considering channels as “depth”, perform convolutions on them independently
Depth-wise Convolutions (Another Diagram)
Point-wise Convolutions

• “Point”-wise convolutions because each pixel is considered independently
• Considering just a single output channel:

Image 7: Pointwise convolution, transforms an image of 3 channels to an image of 1 channel
Point-wise Convolutions

• “Point”-wise convolutions because each pixel is considered independently
• Considering all output channels:

Image 8: Pointwise convolution with 255 kernels, outputting an image with 256 channels
Summary

• A normal convolution mixes information from both **different channels** and **different spatial locations (pixels)**

• A depth-wise convolution only mixes information **over spatial locations**
  • Different channels do not interact.

• A point-wise convolution only mixes information **over different channels**
  • Different spatial locations do not interact
Mobile Block

• Again, to understand an architecture, we mostly need to understand its **blocks**.
• All that changes for blocks in different stages is the base # of channels
Mobile Block

• The core block has three steps:
  • Feature Mixing
  • Spatial Mixing
  • Bottlenecking Channels

Fig. 6: Visualization of the intermediate feature maps in the inverted residual layer (source)
Mobile Block: Feature Mixing

• A point-wise convolution that increases the channel dimension by an “expansion ratio”
Mobile Block: Spatial Mixing

• A depth-wise convolution that communicates information over different spatial locations.
• Point-wise convolution to reduce channel dimension by the same expansion ratio.
ConvNeXt vs MobileNetV2

ConvNeXt
• A 7x7 depth-wise convolution.
• A point-wise convolution increasing # of channels
• A point-wise convolution decreasing # of channels
• Residual Connection

MobileNetV2
• A point-wise convolution increasing # of channels
• A 3x3 depth-wise convolution.
• A point-wise convolution decreasing # of channels
• Residual Connection
ConvNeXt vs MobileNetV2

ConvNeXt
• A 7x7 depth-wise convolution.
• A point-wise convolution increasing # of channels
• A point-wise convolution decreasing # of channels
• Residual Connection

MobileNetV2
• A point-wise convolution increasing # of channels
• A 3x3 depth-wise convolution.
• A point-wise convolution decreasing # of channels
• Residual Connection
ConvNeXt vs MobileNetV2

ConvNeXt
• A 7x7 depth-wise convolution.
• A point-wise convolution increasing # of channels
• A point-wise convolution decreasing # of channels
• Residual Connection

MobileNetV2
• A point-wise convolution increasing # of channels
• A 3x3 depth-wise convolution.
• A point-wise convolution decreasing # of channels
• Residual Connection
ConvNeXt vs MobileNetV2

**ConvNeXt**
- A 7x7 depth-wise convolution.
- A point-wise convolution increasing # of channels.
- A point-wise convolution decreasing # of channels.
- Residual Connection

**MobileNetV2**
- A point-wise convolution increasing # of channels.
- A 3x3 depth-wise convolution.
- A point-wise convolution decreasing # of channels.
- Residual Connection

Extremely Similar!
ConvNeXt vs MobileNetV2: Differences

• So what changed? Some things did change.
• The depth-wise convolution in ConvNeXt is larger kernel size (7x7).
ConvNeXt vs MobileNetV2: Differences

• So what changed? Some things did change.
• The depth-wise convolution in ConvNeXt is larger kernel size (7x7).
• The order of spatial mixing & feature mixing are flipped.
  • In ConvNeXt, depth-wise convolution operates on lower # of channels.
  • In MobileNetV2, operates on higher # of channels.
• Channel Expansion Ratio in ConvNeXt is 4, MobileNetV2 is 6.
ConvNeXt vs MobileNetV2: Differences

• So what changed? Some things did change.
• The depth-wise convolution in ConvNeXt is larger kernel size (7x7).
• The order of spatial mixing & feature mixing are flipped.
  • In ConvNeXt, depth-wise convolution operates on lower # of channels.
  • In MobileNetV2, operates on higher # of channels.
• Channel Expansion Ratio in ConvNeXt is 4, MobileNetV2 is 6.
• ConvNeXt uses LayerNorm, MobileNetV2 uses BatchNorm.
  • **Note: You will need to normalize the data if you use LN.**
• ConvNeXt recommends training via AdamW, MobileNetV2 recommends SGD
Other Interesting Papers

- ResNeXt (2016)
  - Generally a strict improvement to ResNet, but slower. It's like 3 lines of code changed.
- SENet (2017)
  - Channel-wise attention in CNNs. It's like 20 lines of code.
- EfficientNet (2019)
  - Optimized model scaling. Probably can hard code this with some effort.
- RegNet (2020)
  - ResNet with optimized layer sizes. It's probably... 10 lines changed?
- ResNeSt (2020)
  - ResNeXt on steroids + attention. I (we?) will be really impressed 😊
- NFNet (2021, SOTA)
  - Quite doable actually
Building Blocks

Input Image + Transformation

Choice of Model

Training the model
The easy bit first....
Monitoring Training vs Validation Acc

- The standard intuition of “overfitting” is – if the training & validation gap is too large, you should stop training as it’s overfitting.
- However, in modern DL, this intuition is not as relevant.

- XELoss != Accuracy
  - Model can keep improving after training accuracy hits 100%.
  - There is recent research that finds that on some problems, training accuracy hits 100% at epoch 10 while validation accuracy is <50%. Then, on epoch 1000, validation hits 100%.

- Of course, we can’t train for that long, but train until validation stops improving.
  - Or just set a standard LR schedule/setup like “CosineAnnealingLR for 50 epochs” and just let it run. □ what I prefer to do.
How to tackle overfitting?

• There are a lot of different tricks to improving your CNN model.
• From the recent ConvNeXt paper:

<table>
<thead>
<tr>
<th>(pre-)training config</th>
<th>ConvNeXt-T/S/B/L ImageNet-1K 224²</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimizer</td>
<td>AdamW</td>
</tr>
<tr>
<td>base learning rate</td>
<td>4e-3</td>
</tr>
<tr>
<td>weight decay</td>
<td>0.05</td>
</tr>
<tr>
<td>optimizer momentum</td>
<td>β₁, β₂=0.9, 0.999</td>
</tr>
<tr>
<td>batch size</td>
<td>4096</td>
</tr>
<tr>
<td>training epochs</td>
<td>300</td>
</tr>
<tr>
<td>learning rate schedule</td>
<td>cosine decay</td>
</tr>
<tr>
<td>warmup epochs</td>
<td>20</td>
</tr>
<tr>
<td>warmup schedule</td>
<td>linear</td>
</tr>
<tr>
<td>layer-wise lr decay [6, 10]</td>
<td>None</td>
</tr>
<tr>
<td>randaugment [12]</td>
<td>(9, 0.5)</td>
</tr>
<tr>
<td>label smoothing [65]</td>
<td>0.1</td>
</tr>
<tr>
<td>mixup [85]</td>
<td>0.8</td>
</tr>
<tr>
<td>cutmix [84]</td>
<td>1.0</td>
</tr>
<tr>
<td>stochastic depth [34]</td>
<td>0.1/0.4/0.5/0.5</td>
</tr>
<tr>
<td>layer scale [69]</td>
<td>1e-6</td>
</tr>
<tr>
<td>gradient clip</td>
<td>None</td>
</tr>
<tr>
<td>exp. mov. avg. (EMA) [48]</td>
<td>0.9999</td>
</tr>
</tbody>
</table>
How to tackle overfitting?

• There are a lot of different tricks to improving your CNN model.
• From the recent ConvNeXt paper
• What we recommend trying first:
  • Label Smoothing (huge boost)
  • Stochastic Depth
  • DropBlock (paper)
  • Dropout before final classification layer
• Then you can try the others
• Check out “Bag of Tricks for Image Classification with Convolutional Neural Networks”
  • https://arxiv.org/abs/1812.01187
Let’s get real now....
Loss Functions

Face Verification + Face Recognition tasks (VGG Face2)
Face Classification

Feature Extraction Model → Feature Embedding → Classification Linear Layer → N-Class Probability Logs → Softmax → N-Class Probabilities
Face Verification

![Face Verification Diagram]

1. Feature Extraction Model
2. Feature Embedding
3. Similarity metric
4. A similarity score
Types of Loss functions

Non Contrastive loss functions

Contrastive-Losses
Types of Loss functions

Non Contrastive loss functions

Contrastive-Losses
How to train models with such loss functions?
Approach 1 - Joint Loss Optimization

Model → Embeddings → Cross Entropy
Model → Labels → Cross Entropy

Model → Embeddings → Contrastive Loss
Approach 2 - Sequential (Fine-tuning)

Face Recognition

Step 1

Model → Embeddings → Cross Entropy

Face Verification

Step 2

Model → Embeddings → Contrastive Loss
Types of Contrastive Losses

1. Centre Loss
2. Triplet Loss
3. Sphere Face (Angular Softmax)
4. CosFace Loss
5. ArcFace
Centre Loss

- Increases the disparity between classes using softmax
- Increases inter-class distance by reducing intra-class Euclidean distance by assigning centers to each class.

- Calculating the centre for each class, is difficult
**Triplet Loss**

\[
\text{Loss} = \sum_{i=1}^{N} \left[ \| f_i^a - f_i^p \|_2^2 - \| f_i^a - f_i^n \|_2^2 + \alpha \right]_+
\]

- Involves sampling 3 images, an anchor, a positive (same class as anchor) and a negative (different class from anchor).
- Use a p-norm distance function to increase the difference between anchor and negative whilst minimizing distance between anchor and positive.
- **Sampling hard positives and hard negatives is key and difficult**
Triplet Loss

The diagram illustrates the process of Triplet Loss, which is a loss function used in deep learning to improve the similarity between anchor and positive examples while maximizing the dissimilarity between anchor and negative examples. The process involves:

1. Input images are fed into CNNs.
2. Each CNN outputs embeddings.
3. The embeddings are compared to calculate the Triplet Loss.

The learning process aims to minimize the loss by adjusting the weights of the CNNs to ensure that the anchor and positive examples are closer to each other in the embedding space, while the anchor and negative examples are farther apart.
Sphere Face

$$L_{ang} = - \sum_i \ln \frac{\exp \left\{ \| x_i \| \cos (m \cdot \theta_{y_i,i}) \right\}}{\exp \left\{ \| x_i \| \cos (m \cdot \theta_{y_i,i}) \right\} + \sum_{j \neq i} \exp \left\{ \| x_i \| \cos (\theta_{j,i}) \right\}}$$

- Makes use of an angular margin, imposed by $\theta$
- The learned features construct a discriminative angular distance equivalent to the geodesic distance on a hypersphere manifold
- $\theta$ denotes the type of decision boundary learned, which leads to different margins for different classes
Sphere Face

$M \alpha \text{ Angular Margin}$
CosFace

\[
L_{\text{lime}} = - \frac{1}{N} \sum_{i=1}^{N} \ln \frac{\exp \left\{ s \cdot (\cos(\theta_{y,i}) - m) \right\}}{\exp \left\{ s \cdot (\cos(\theta_{y,i}) - m) \right\} + \sum_{j \neq y_i} \exp \left\{ s \cdot (\cos(\theta_{j,i})) \right\}}
\]

- Similar to Sphere Face
- Forms the decision margin in cosine space rather than angular space.
ArcFace

- Builds on the concepts of the sphere face and cos face.
- Replaced the multiplicative angular margin in CosFace, with an additive margin ‘m’
ArcFace

- The additive factor of ‘m’ has found to lead to better convergence as compared to its multiplicative counterpart in Sphere Face.
CMU Virtual Andrew

https://www.cmu.edu/computing/services/endpoint/software/virtual-andrew.html