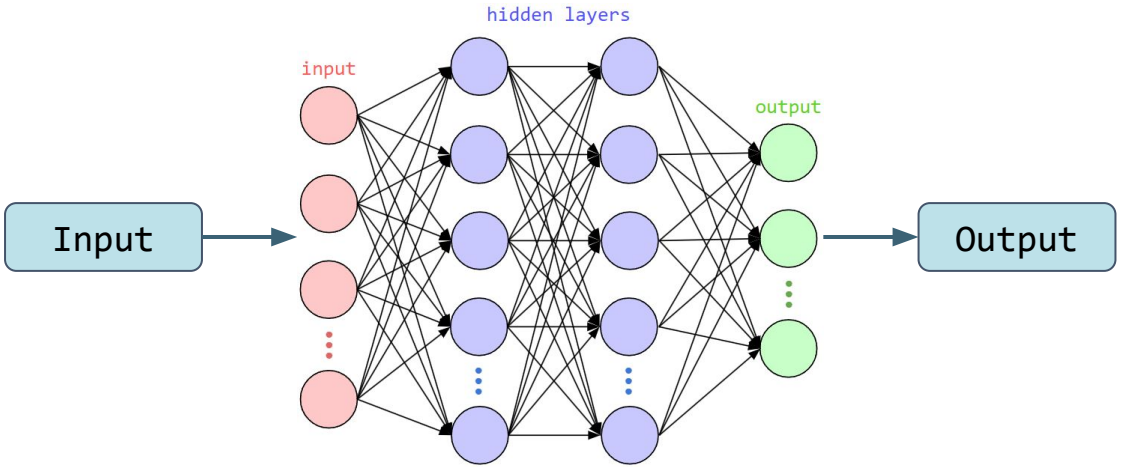
DeepLearning.AI

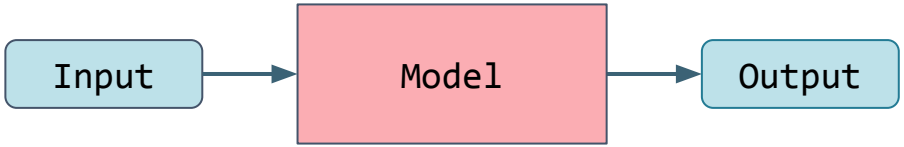
Custom Architectures

Designing Custom Architectures

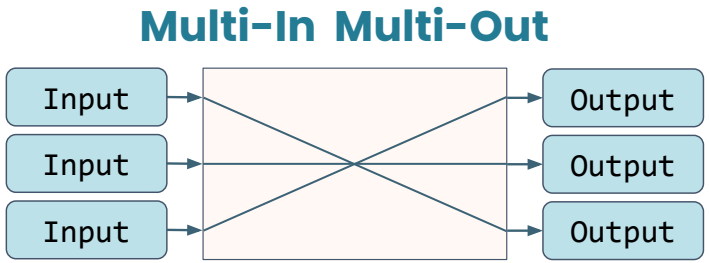
Linear Models



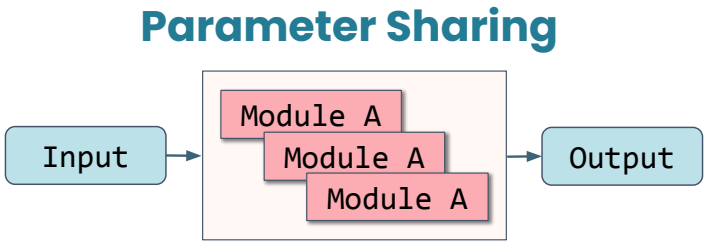
Linear Models



Custom Architectures



Custom Architectures

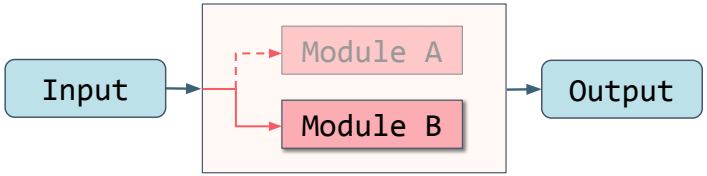


Custom Architectures

Multi-In Multi-Out

Parameter Sharing

Conditional Execution



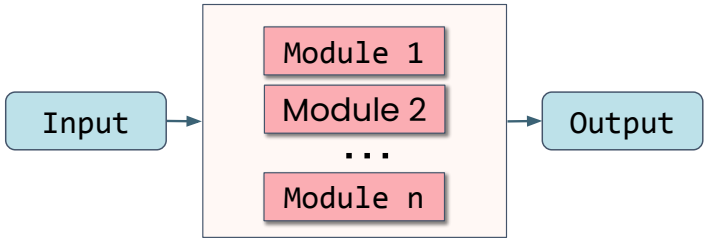
Custom Architectures

Multi-In Multi-Out

Parameter Sharing

Conditional Execution

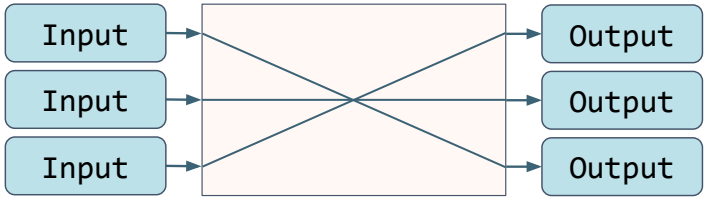
Dynamic Model Creation



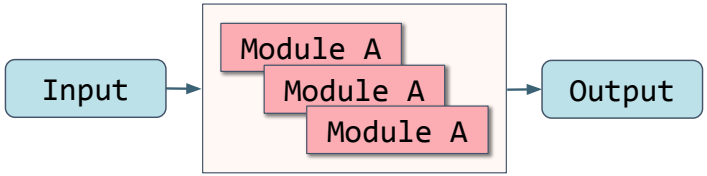
n = num_layers

Custom Architectures

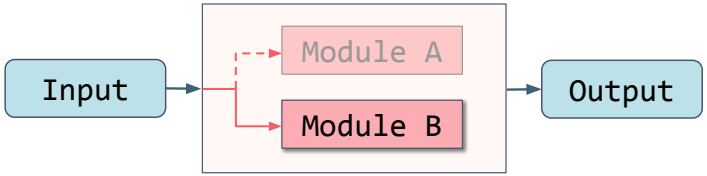
Multi-In Multi-Out



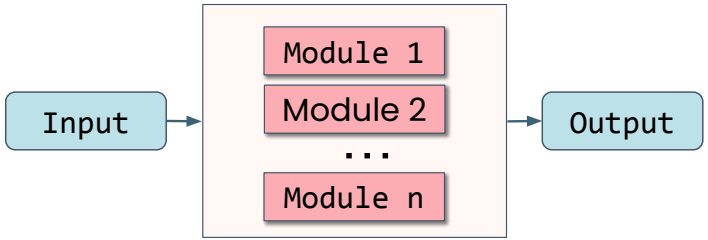
Parameter Sharing



Conditional Execution

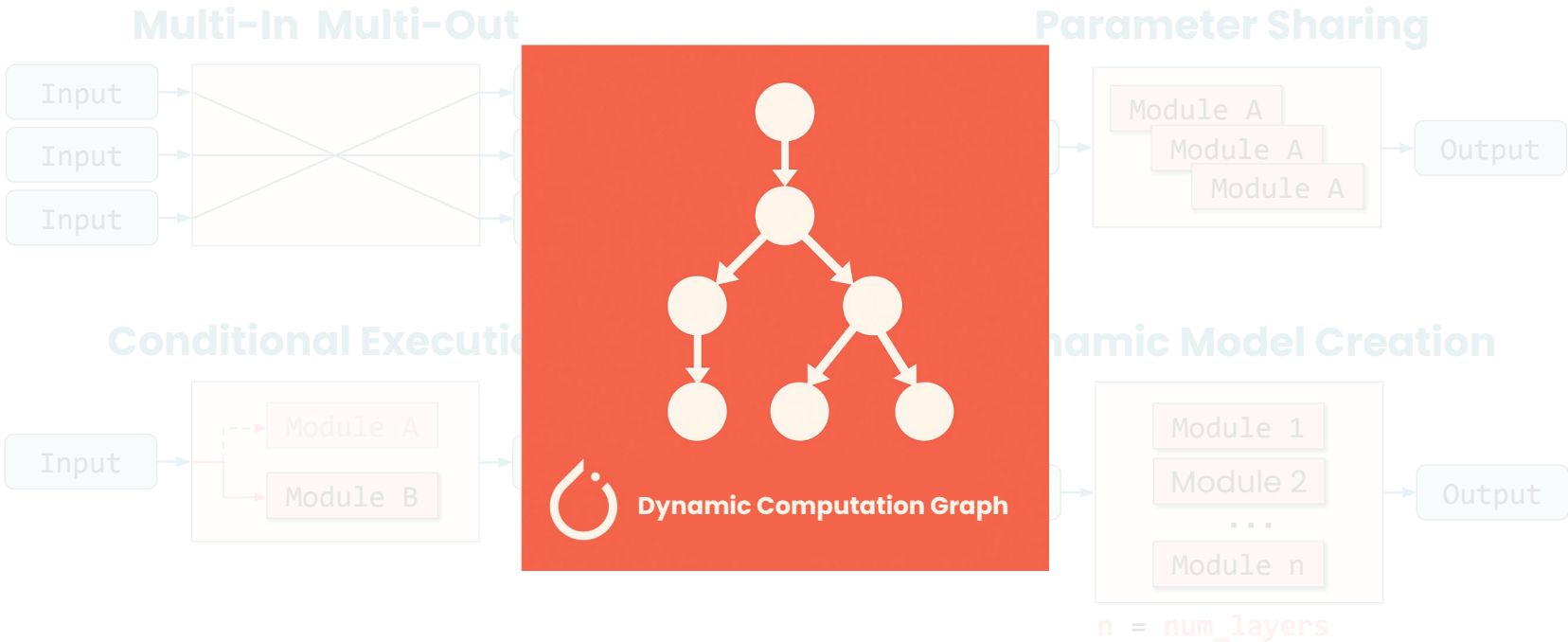


Dynamic Model Creation



`n = num_layers`

Custom Architectures



Modularity Recap

```
class ConvBlock(nn.Module):
  def __init__(self):
      . . .
  def forward(self):
      . . .
```

Modularity Recap

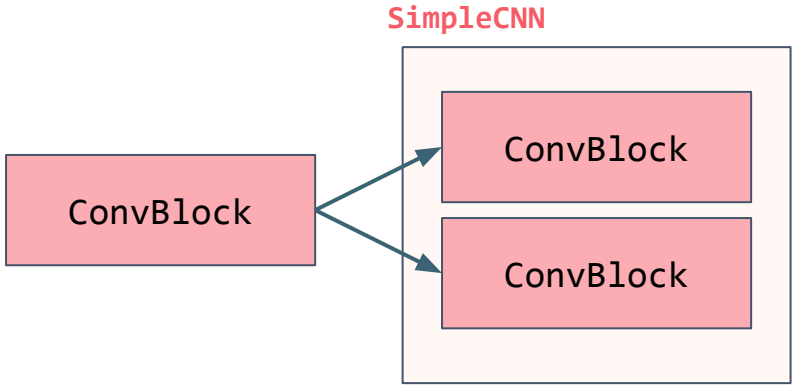
```
class ConvBlock(nn.Module):
  def __init__(self):
      . . .
  def forward(self):
      . . .
```

Modularity Recap

```
class ConvBlock(nn.Module):
  def __init__(self):
      . . .
  def forward(self):
      . . .



class SimpleCNN(nn.Module):
  def __init__(self):
    . . .
    self.block1 = ConvBlock(1, 32)
    self.block2 = ConvBlock(32, 64)
  def forward(self):
    . . .
```

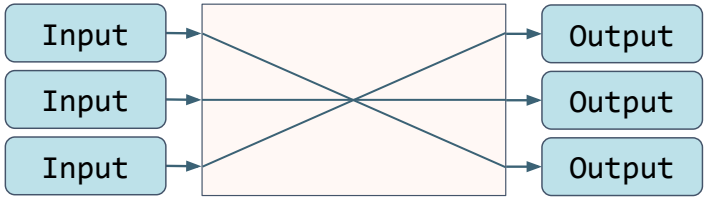
Modularity Recap

```
class ConvBlock(nn.Module):
  def __init__(self):
      . . .
  def forward(self):
      . . .



class SimpleCNN(nn.Module):
  def __init__(self):
    . . .
    self.block1 = ConvBlock(1, 32)
    self.block2 = ConvBlock(32, 64)
  def forward(self):
    . . .
```

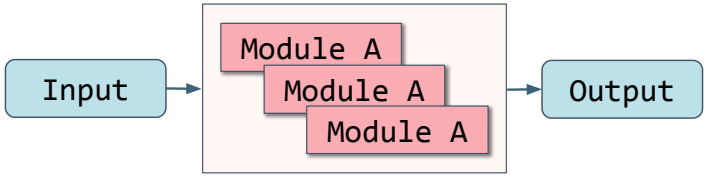


Custom Architectures

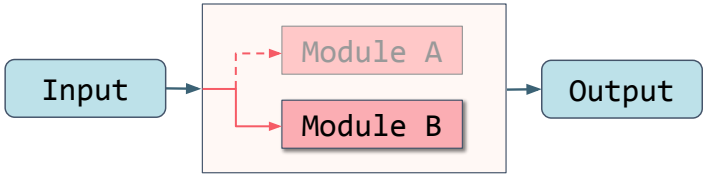
Multi-In Multi-Out



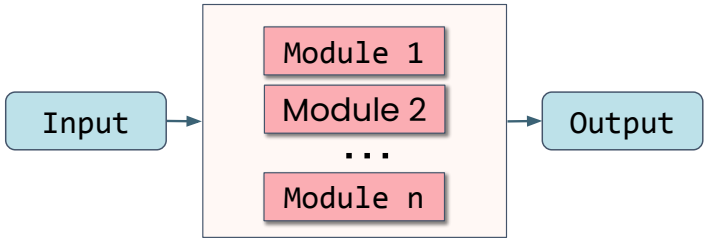
Parameter Sharing



Conditional Execution



Dynamic Model Creation



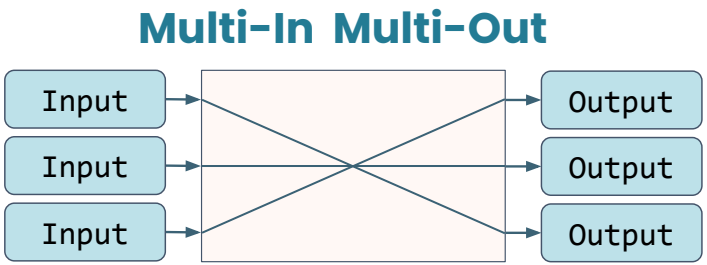
`n = num_layers`

Custom Architectures

1. **Multi-In Multi-Out**
2. **Parameter Sharing**
3. **Conditional Execution**
4. **Dynamic Model Creation**

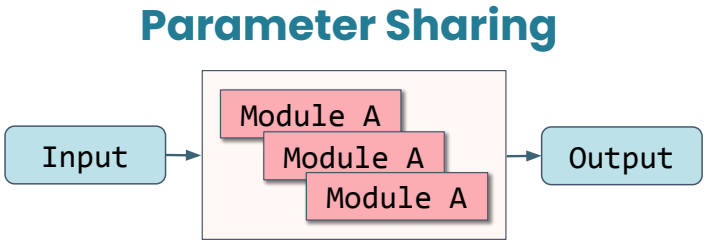
Custom Architectures

1. **Multi-In Multi-Out**
2. Parameter Sharing
3. Conditional Execution
4. Dynamic Model Creation



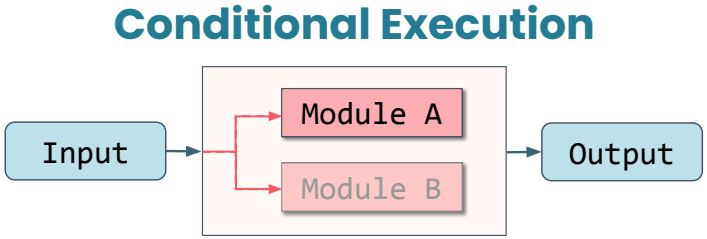
Custom Architectures

1. Multi-In Multi-Out
2. **Parameter Sharing**
3. Conditional Execution
4. Dynamic Model Creation



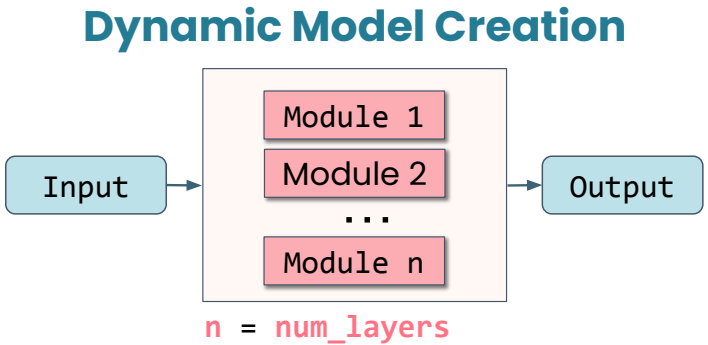
Custom Architectures

1. Multi-In Multi-Out
2. Parameter Sharing
3. **Conditional Execution**
4. Dynamic Model Creation



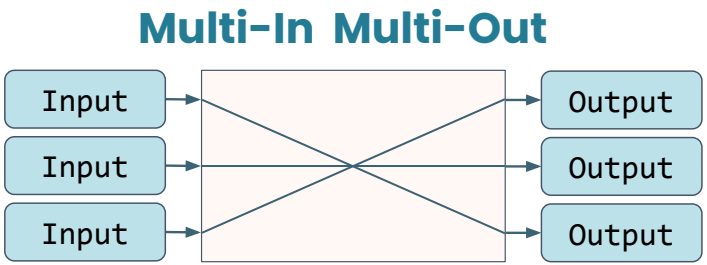
Custom Architectures

1. Multi-In Multi-Out
2. Parameter Sharing
3. Conditional Execution
4. **Dynamic Model Creation**



Custom Architectures

1. **Multi-In Multi-Out**
2. Parameter Sharing
3. Conditional Execution
4. Dynamic Model Creation



1. Multi-In Multi-Out

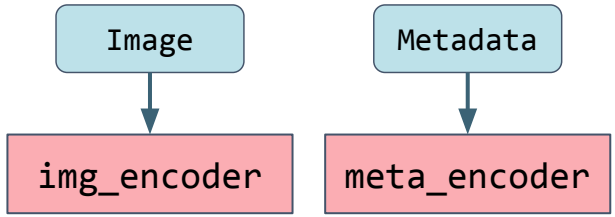
metadata

```
{"date_taken": "2024-05-10",
 "camera_make": "Canon",
 "camera_model": "EOS 80D",
 "brightness": 56,}
```

1. Multi-In Multi-Out

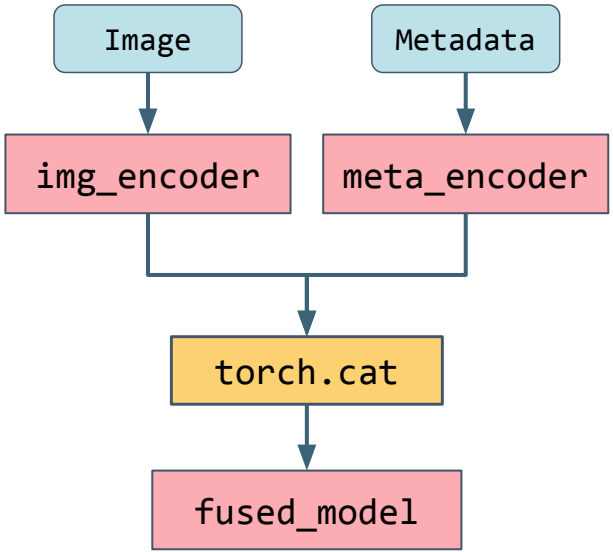
1. Multi-In Multi-Out

```
img_encoder = nn.Sequential(nn.Conv2d(...),
                    nn.ReLU(),
                    nn.Flatten(),
                    nn.Linear(...))

meta_encoder = nn.Sequential(nn.Linear(...),
                     nn.ReLU(),
                     nn.Linear(...))
```



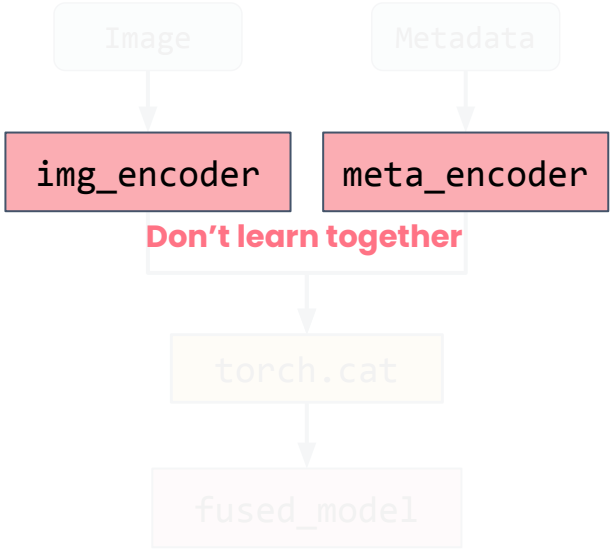
1. Multi-In Multi-Out

```
img_encoder = nn.Sequential(nn.Conv2d(...),
                            nn.ReLU(),
                            nn.Flatten(),
                            nn.Linear(...))

meta_encoder = nn.Sequential(nn.Linear(...),
                             nn.ReLU(),
                             nn.Linear(...))

img_out = img_encoder(image)
meta_out = meta_encoder(metadata)
fused_out = torch.cat([img_out, meta_out], dim=1)

fused_model = nn.Sequential(nn.Linear(...),
                            nn.ReLU(),
                            nn.Linear(...))
```



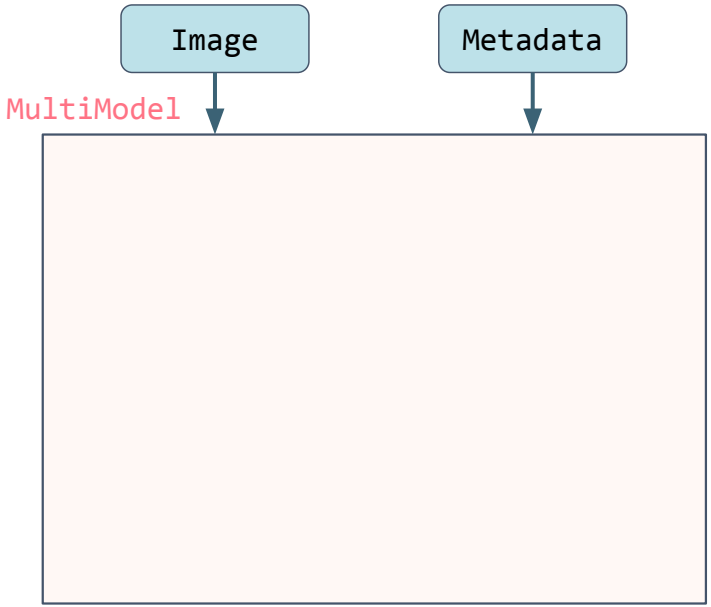
1. Multi-In Multi-Out

```
img_encoder = nn.Sequential(nn.Conv2d(...),
                  nn.ReLU(),
                  nn.Flatten(),
                  nn.Linear(...))

meta_encoder = nn.Sequential(nn.Linear(...),
                   nn.ReLU(),
                   nn.Linear(...))

img_out = img_encoder(image)
meta_out = meta_encoder(metadata)
fused_out = torch.cat([img_out, meta_out], dim=1)

fused_model = nn.Sequential(nn.Linear(...),
                  nn.ReLU(),
                   nn.Linear(...))
```



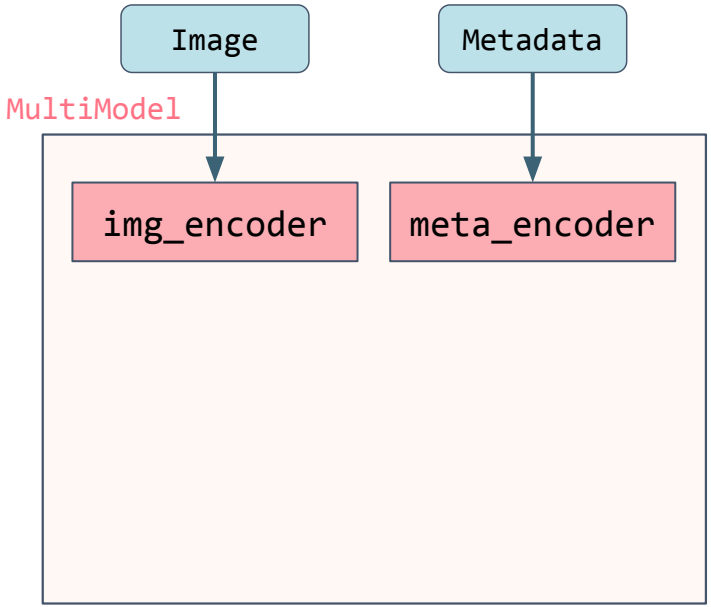
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                        [img.flatten(1), meta],
                        dim=1)

        out = self.final_layer(combined)
        return out
```



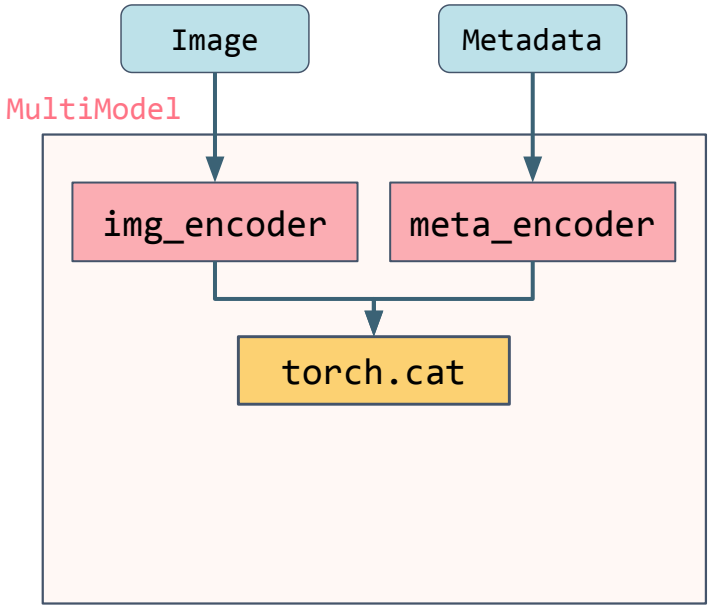
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                       [img.flatten(1), meta],
                       dim=1)

        out = self.final_layer(combined)
        return out
```



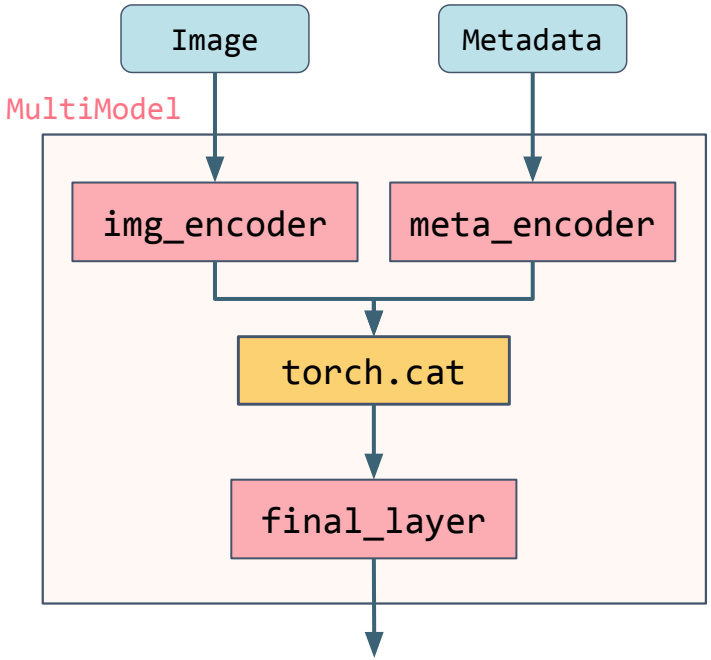
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                        [img.flatten(1), meta],
                        dim=1)

        out = self.final_layer(combined)
        return out
```



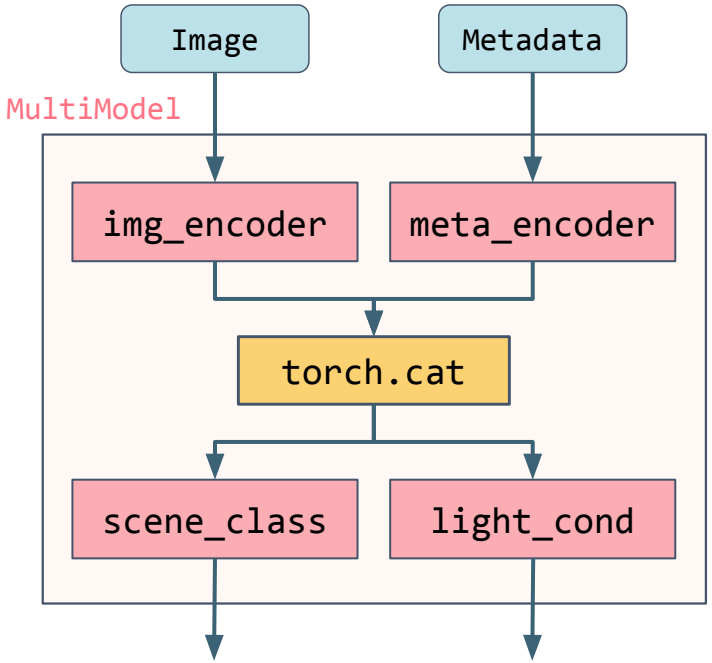
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                        [img.flatten(1), meta],
                        dim=1)

        out = self.final_layer(combined)
        return out
```



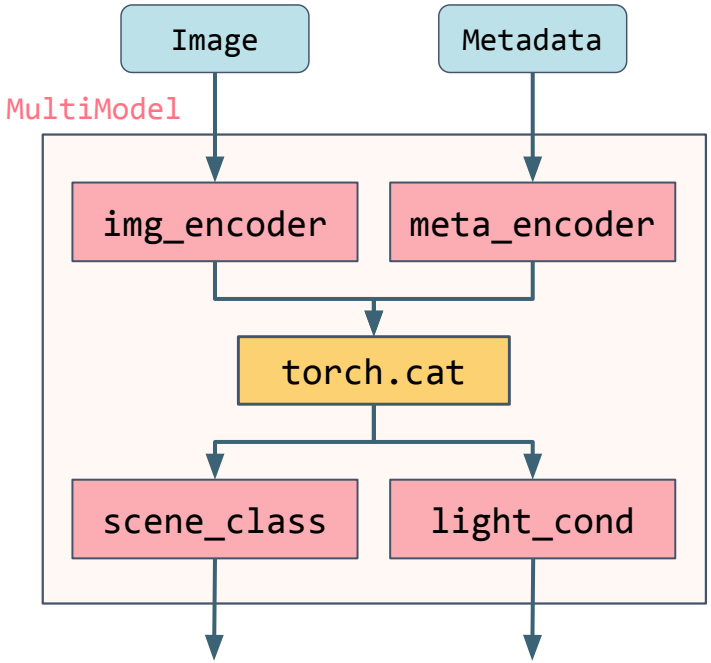
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                        [img.flatten(1), meta],
                        dim=1)

        scene = self.scene_class(combined)
        lighting = self.light_cond(combined)
        return scene, lighting
```



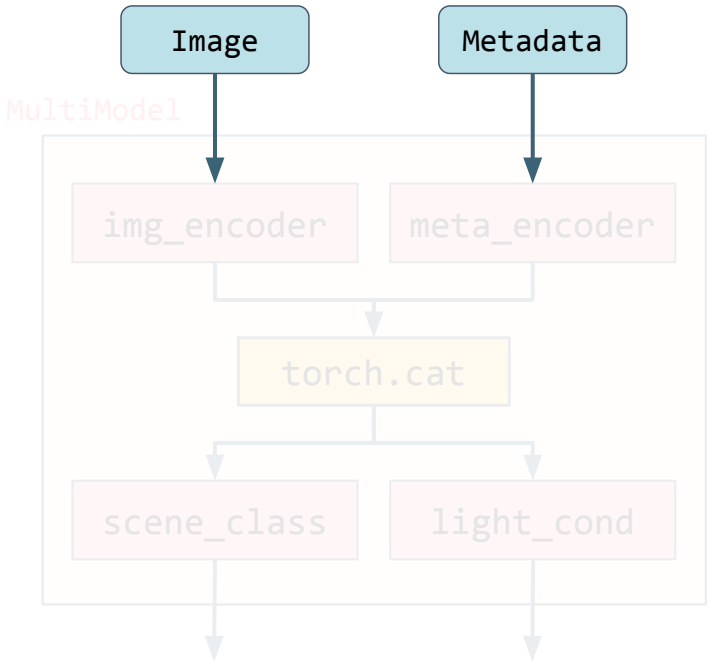
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                        [img.flatten(1), meta],
                        dim=1)

        scene = self.scene_class(combined)
        lighting = self.light_cond(combined)
        return scene, lighting
```



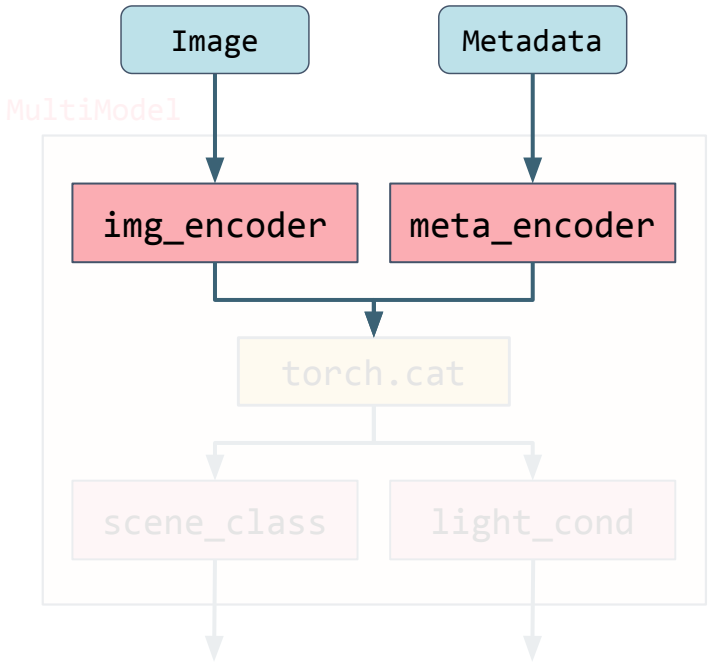
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                       [img.flatten(1), meta],
                       dim=1)

        scene = self.scene_class(combined)
        lighting = self.light_cond(combined)
        return scene, lighting
```



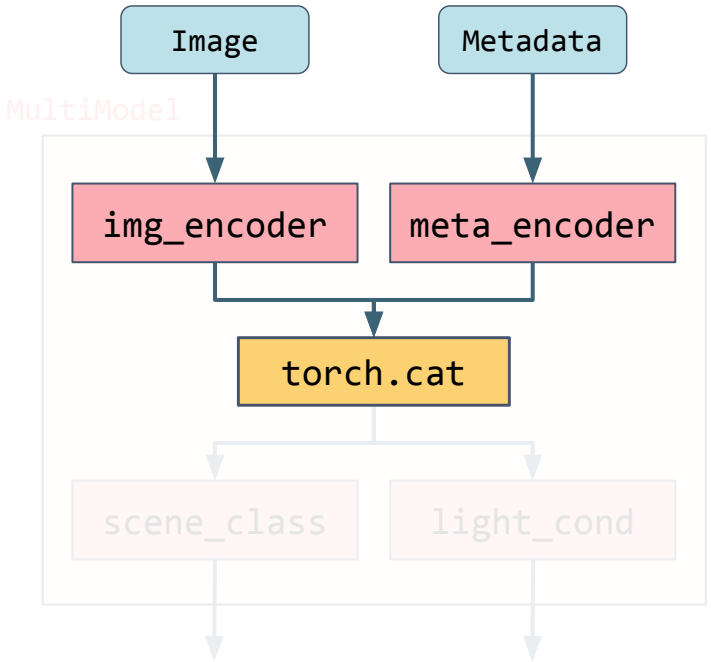
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                       [img.flatten(1), meta],
                       dim=1)

        scene = self.scene_class(combined)
        lighting = self.light_cond(combined)
        return scene, lighting
```



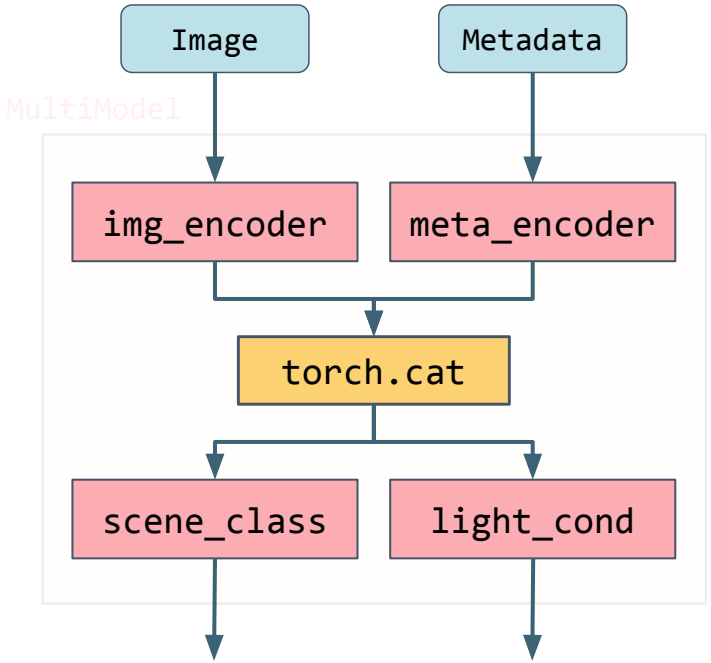
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                        [img.flatten(1), meta],
                        dim=1)

        scene = self.scene_class(combined)
        lighting = self.light_cond(combined)
        return scene, lighting
```



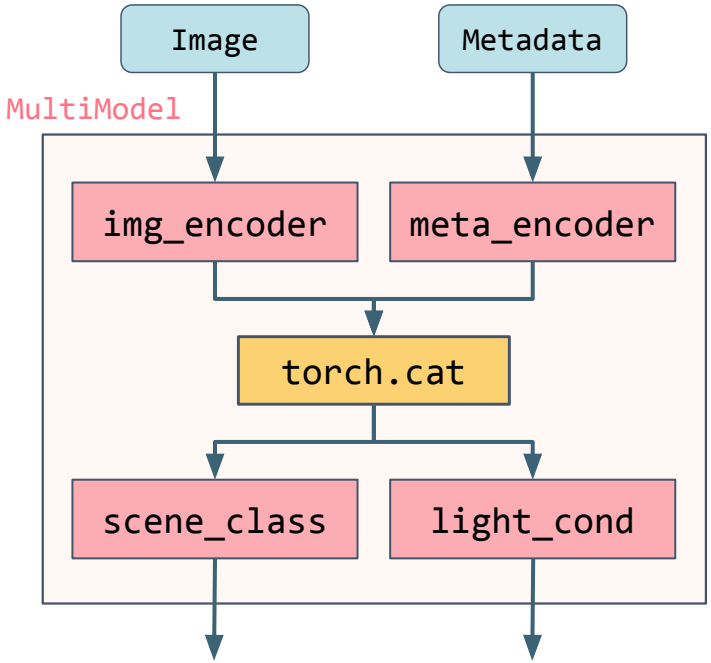
1. Multi-In Multi-Out

```
class MultiModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, metadata):
        img = self.img_encoder(image)
        meta = self.meta_encoder(metadata)

        combined = torch.cat(
                        [img.flatten(1), meta],
                        dim=1)

        scene = self.scene_class(combined)
        lighting = self.light_cond(combined)
        return scene, lighting
```



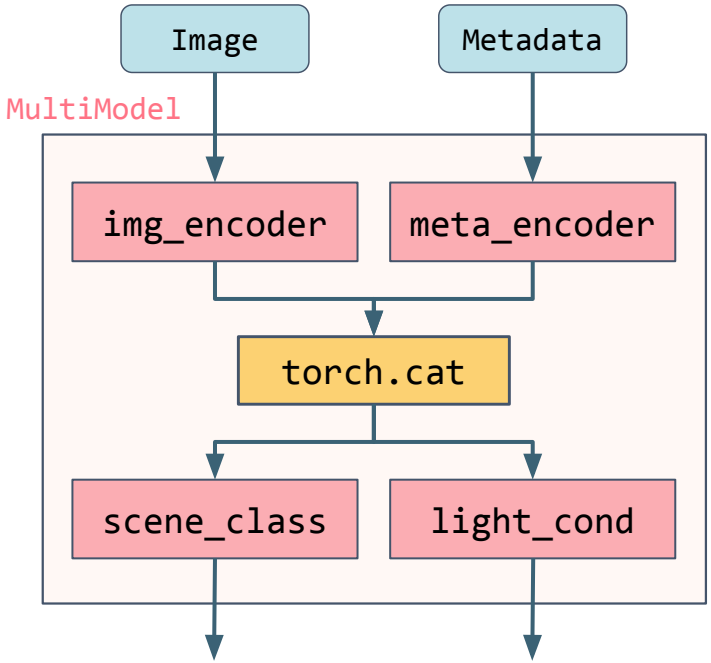
1. Multi-In Multi-Out

```
scene_logits, light_logits = model(img, meta)


loss_scene = F.cross_entropy(scene, tgt_scene)
loss_light = F.cross_entropy(lighting, tgt_light)

loss = loss_scene + loss_light

optimizer.zero_grad()
loss.backward()
optimizer.step()
```



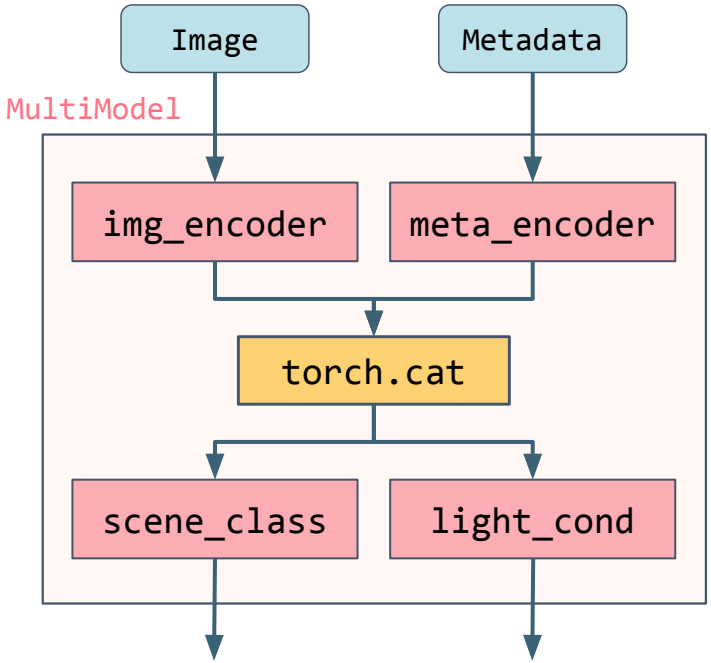
1. Multi-In Multi-Out

```
scene_logits, light_logits = model(img, meta)


loss_scene = F.cross_entropy(scene, tgt_scene)
loss_light = F.cross_entropy(lighting, tgt_light)

loss = loss_scene + loss_light

optimizer.zero_grad()
loss.backward()
optimizer.step()
```



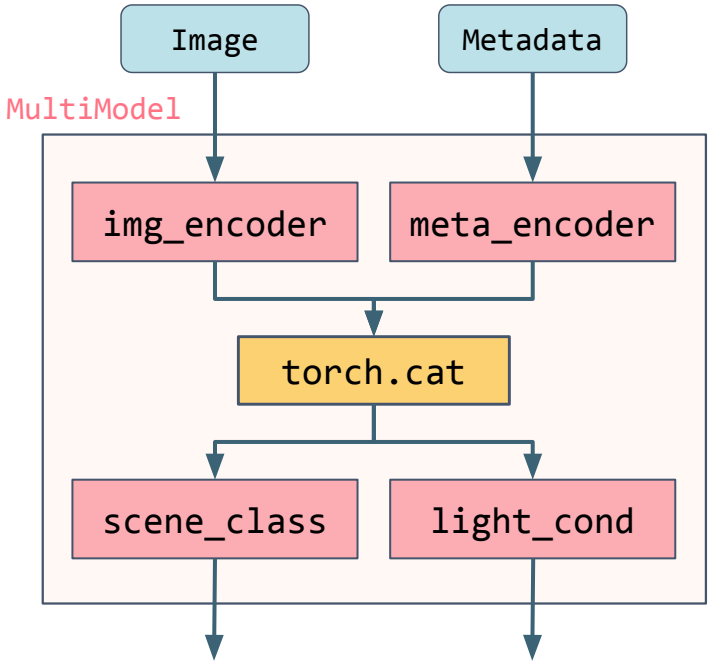
1. Multi-In Multi-Out

```
scene_logits, light_logits = model(img, meta)

loss_scene = F.cross_entropy(scene, tgt_scene)
loss_light = F.cross_entropy(lighting, tgt_light)

loss = loss_scene + loss_light

optimizer.zero_grad()
loss.backward()
optimizer.step()
```



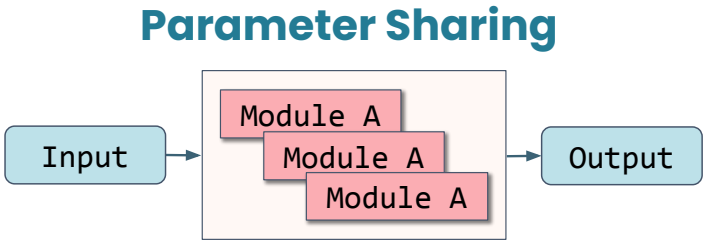
1. Multi-In Multi-Out

```
scene_logits, light_logits = model(img, meta)


loss_scene = F.cross_entropy(scene, tgt_scene)
loss_light = F.cross_entropy(lighting, tgt_light)

loss = loss_scene + loss_light

optimizer.zero_grad()
loss.backward()
optimizer.step()
```



Custom Architectures

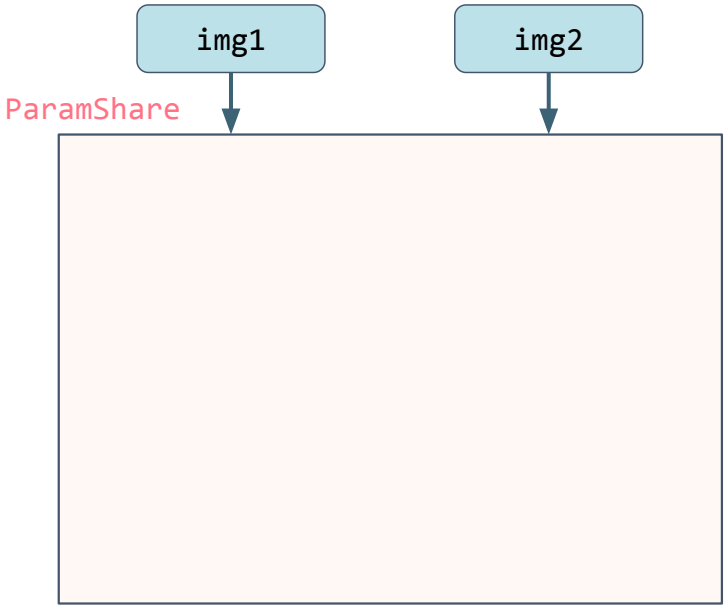
1. Multi-In Multi-Out
2. **Parameter Sharing**
3. Conditional Execution
4. Dynamic Model Creation



2. Parameter Sharing

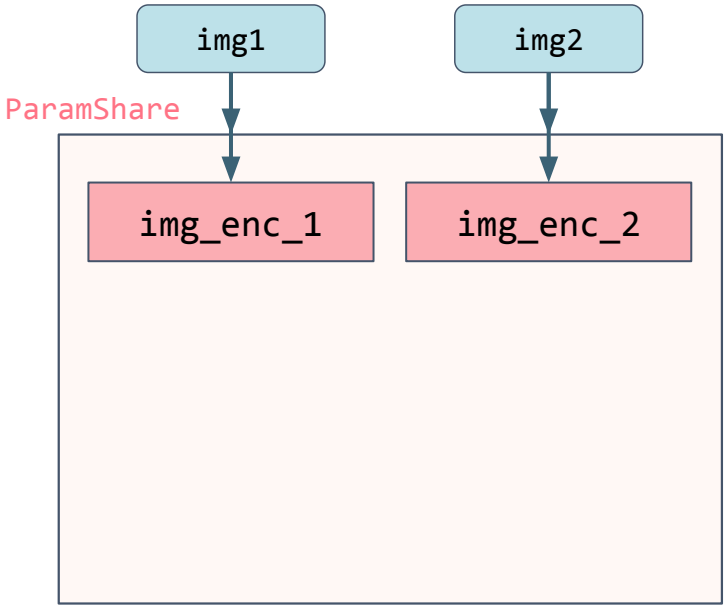
Same product?

2. Parameter Sharing



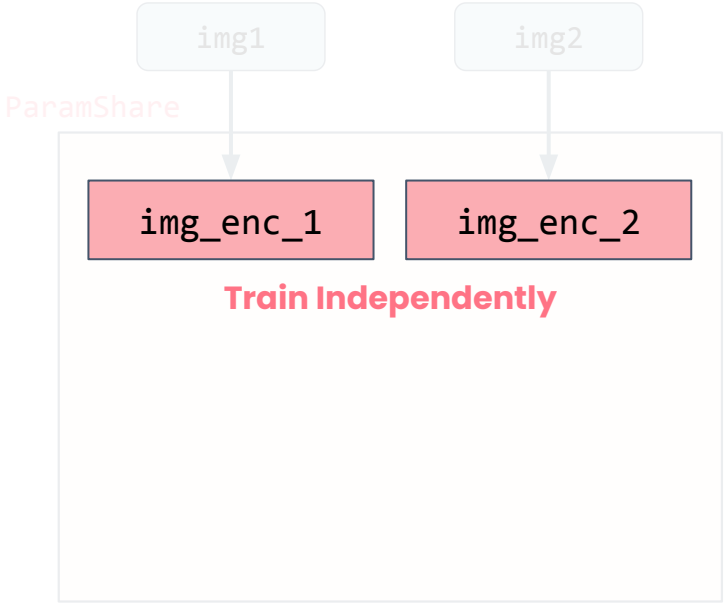
2. Parameter Sharing

```
img_enc_1 = nn.Sequential(nn.Conv2d(...),
                  nn.ReLU(),
                  nn.Flatten(),
                  nn.Linear(...))


img_enc_2 = nn.Sequential(nn.Conv2d(...),
                  nn.ReLU(),
                  nn.Flatten(),
                  nn.Linear(...))
```



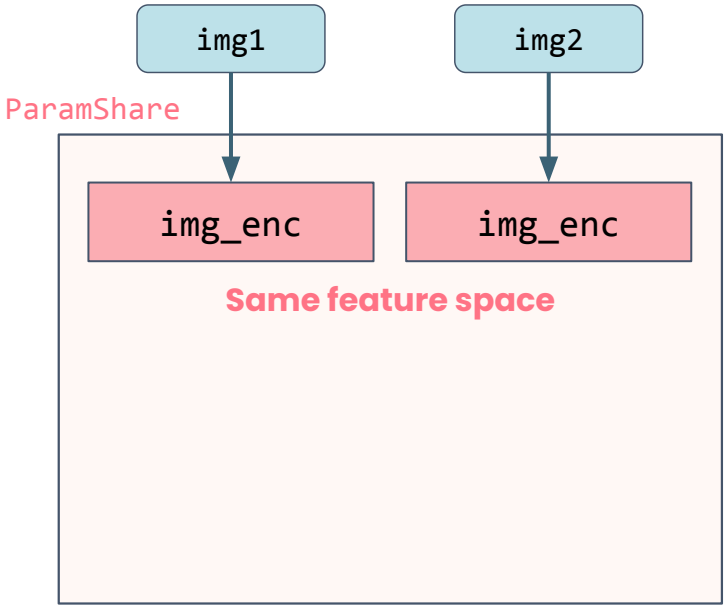
2. Parameter Sharing

```
img_enc_1 = nn.Sequential(nn.Conv2d(...),
                  nn.ReLU(),
                  nn.Flatten(),
                  nn.Linear(...))


img_enc_2 = nn.Sequential(nn.Conv2d(...),
                  nn.ReLU(),
                  nn.Flatten(),
                  nn.Linear(...))
```



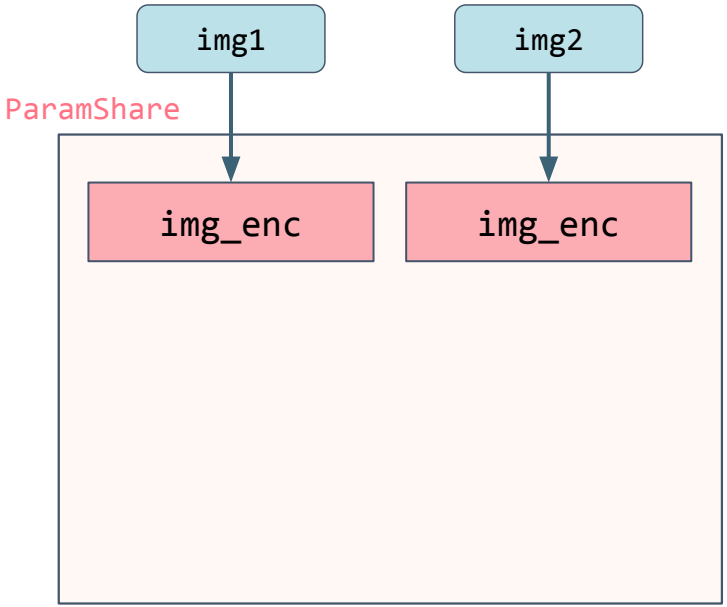
2. Parameter Sharing

```
class ParamShare(nn.Module):
    def __init__(self):
        super().__init__()
        self.img_enc = nn.Sequential(
            nn.Conv2d(...),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(...),
        )

    def forward(self, img1, img2):
        enc_1 = self.img_enc(img1)
        enc_2 = self.img_enc(img2)

        return enc_1, enc_2
```



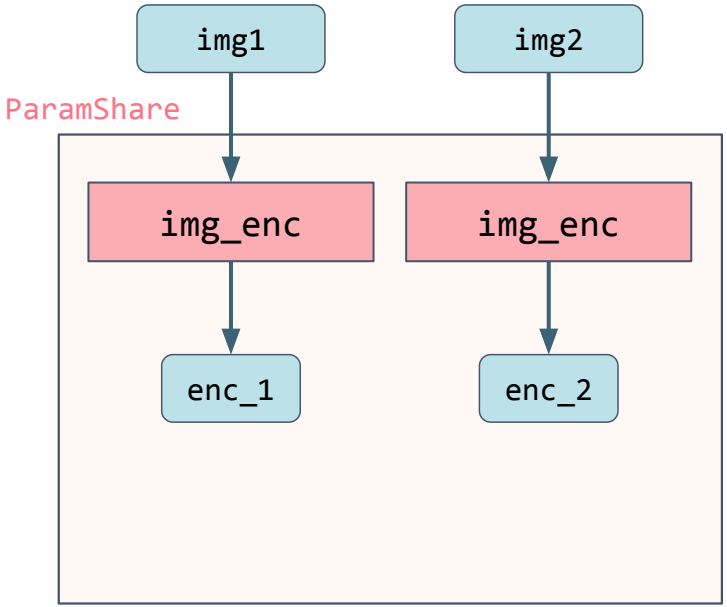
2. Parameter Sharing

```
class ParamShare(nn.Module):
    def __init__(self):
        super().__init__()
        self.img_enc = nn.Sequential(
            nn.Conv2d(...),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(...),
        )

    def forward(self, img1, img2):
        enc_1 = self.img_enc(img1)
        enc_2 = self.img_enc(img2)

        return enc_1, enc_2
```



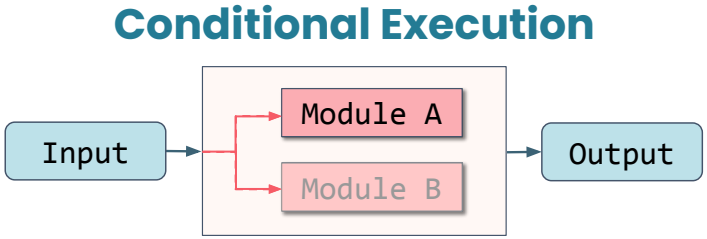
2. Parameter Sharing

```
class ParamShare(nn.Module):
    def __init__(self):
        super().__init__()
        self.img_enc = nn.Sequential(
            nn.Conv2d(...),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(...),
        )

    def forward(self, img1, img2):
        enc_1 = self.img_enc(img1)
        enc_2 = self.img_enc(img2)

        return enc_1, enc_2
```

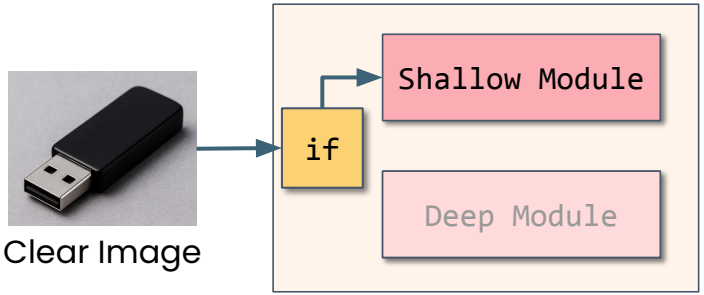


Custom Architectures

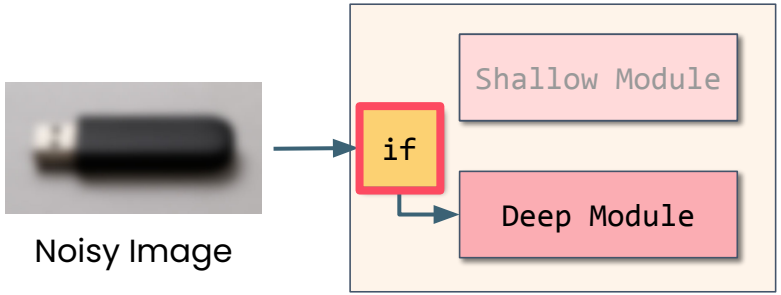
1. Multi-In Multi-Out
2. Parameter Sharing
3. **Conditional Execution**
4. Dynamic Model Creation



3. Conditional Execution



3. Conditional Execution



3. Conditional Execution

```
model = nn.Sequential(
            nn.Conv2d(32, 64, 3),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3)
        )
```

3. Conditional Execution

```
class ConditionalModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, is_difficult):
        x = self.base(image)

        # Choose the path based on the input condition
        if is_difficult:
            x = self.deep_path(x)
        else:
            x = self.simple_path(x)
        return x
```

3. Conditional Execution

```
class ConditionalModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, is_difficult):
        x = self.base(image)

        # Choose the path based on the input condition
        if is_difficult:
            x = self.deep_path(x)
        else:
            x = self.simple_path(x)
        return x
```

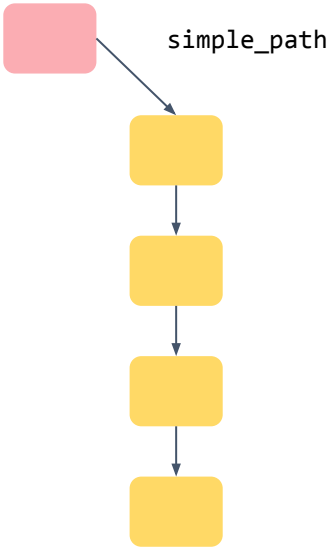
What about backprop?

3. Conditional Execution

```
class ConditionalModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, is_difficult):
        x = self.base(image)

        # Choose the path based on the input condition
        if is_difficult:
            x = self.deep_path(x)
        else:
            x = self.simple_path(x)
        return x
```

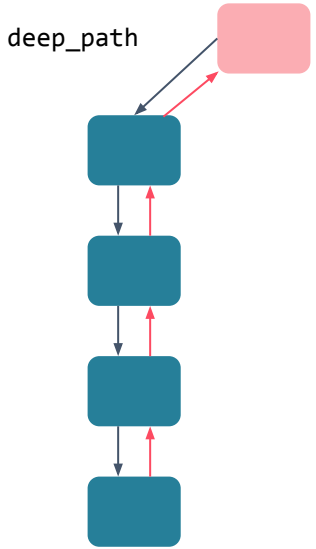
3. Conditional Execution

```
class ConditionalModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, is_difficult):
        x = self.base(image)

        # Choose the path based on the input condition
        if is_difficult:
            x = self.deep_path(x)
        else:
            x = self.simple_path(x)
        return x
```



3. Conditional Execution

```
class ConditionalModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, is_difficult):
        x = self.base(image)

        # Choose the path based on the input condition
        if is_difficult:
            x = self.deep_path(x)
        else:
            x = self.simple_path(x)
        return x
```

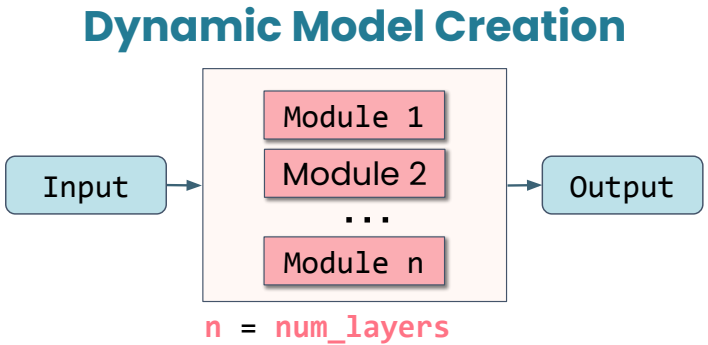


3. Conditional Execution

```
class ConditionalModel(nn.Module):
    def __init__(self):
        . . .

    def forward(self, image, is_difficult):
        x = self.base(image)

        # Choose the path based on the input condition
        if is_difficult:
            x = self.deep_path(x)
        else:
            x = self.simple_path(x)
        return x
```

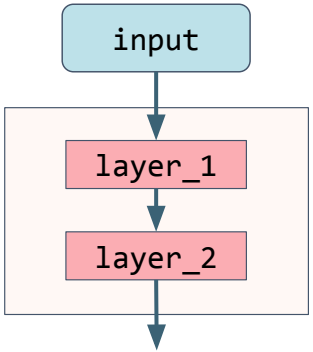
Custom Architectures

1. Multi-In Multi-Out
2. Parameter Sharing
3. Conditional Execution
4. **Dynamic Model Creation**



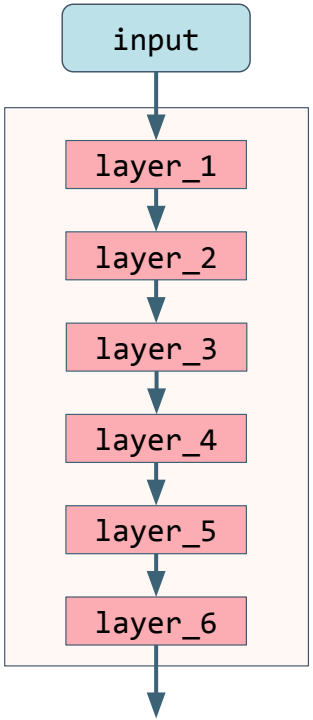
4. Dynamic Model Creation

```
model = DynamicDepthModel(num_layers=2)
```



4. Dynamic Model Creation

```
model = DynamicDepthModel(num_layers=6)
```



4. Dynamic Model Creation

```
model = DynamicDepthModel(num_layers=6)


class DynamicDepthModel(nn.Module):
    def __init__(self, num_layers):
        super().__init__()
        self.layers = [nn.Linear(...) for _ in range(num_layers)]
```

4. Dynamic Model Creation

```
model = DynamicDepthModel(num_layers=6)


class DynamicDepthModel(nn.Module):
    def __init__(self, num_layers):
        super().__init__()
        self.layers = [nn.Linear(...) for _ in range(num_layers)]
```

4. Dynamic Model Creation

```
model = DynamicDepthModel(num_layers=6)


class DynamicDepthModel(nn.Module):
    def __init__(self, num_layers):
        super().__init__()
        self.layers = [nn.Linear(...) for _ in range(num_layers)]


    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
```

4. Dynamic Model Creation

```
self.layers = [nn.Linear(...) for _ in range(num_layers)]
```

❌ **Won't train and won't save**

4. Dynamic Model Creation

```
self.layers = [nn.Linear(...) for _ in range(num_layers)]
```

```
self.layers = nn.ModuleList([nn.Linear(...) for _ in range(num_layers)])
```

4. Dynamic Model Creation

```
self.layers = [nn.Linear(...) for _ in range(num_layers)]
```

```
self.layers = nn.ModuleList([nn.Linear(...) for _ in range(num_layers)])
```

4. Dynamic Model Creation

```
self.layers = [nn.Linear(...) for _ in range(num_layers)]
```

```
self.layers = nn.ModuleList([nn.Linear(...) for _ in range(num_layers)])
```

Trainable

4. Dynamic Model Creation

```
self.layers = [nn.Linear(...) for _ in range(num_layers)]


self.layers = nn.ModuleList([nn.Linear(...) for _ in range(num_layers)])


self.layers = nn.ModuleDict({
                “classify”: nn.Linear(...),
                “regress”:  nn.Linear(...)
            })
```

4. Dynamic Model Creation

```
self.layers = [nn.Linear(...) for _ in range(num_layers)]


self.layers = nn.ModuleList([nn.Linear(...) for _ in range(num_layers)])


self.layers = nn.ModuleDict({
                “classify”: nn.Linear(...),
                “regress”:  nn.Linear(...)
              })
```

4. Dynamic Model Creation

```
self.layers = [nn.Linear(...) for _ in range(num_layers)]


self.layers = nn.ModuleList([nn.Linear(...) for _ in range(num_layers)])


self.layers = nn.ModuleDict({
                "classify": nn.Linear(...),
                "regress":  nn.Linear(...)
            })
```

4. Dynamic Model Creation

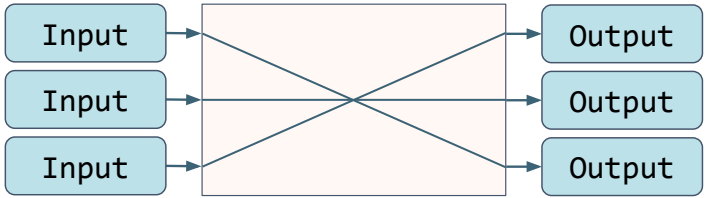
```
self.layers = [nn.Linear(...) for _ in range(num_layers)]
```

```
self.layers = nn.ModuleList([nn.Linear(...) for _ in range(num_layers)])
```

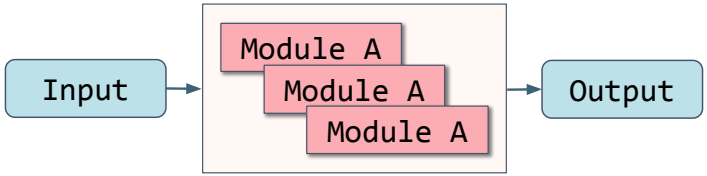
```
self.layers = nn.ModuleDict({
                “classify”: nn.Linear(...),
                “regress”:  nn.Linear(...)
            })
```

Custom Architectures

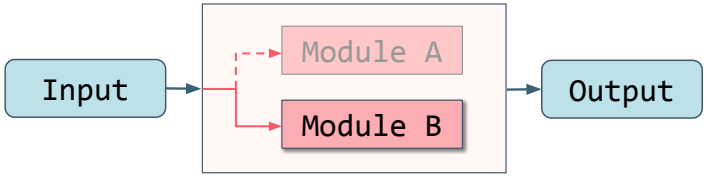
Multi-In Multi-Out



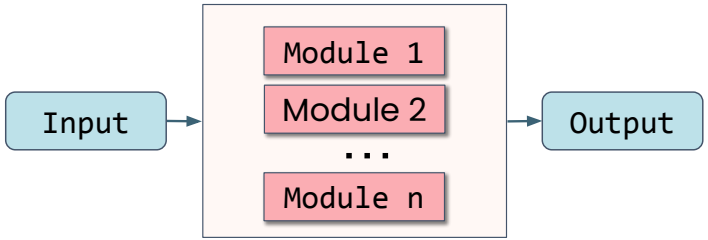
Parameter Sharing



Conditional Execution

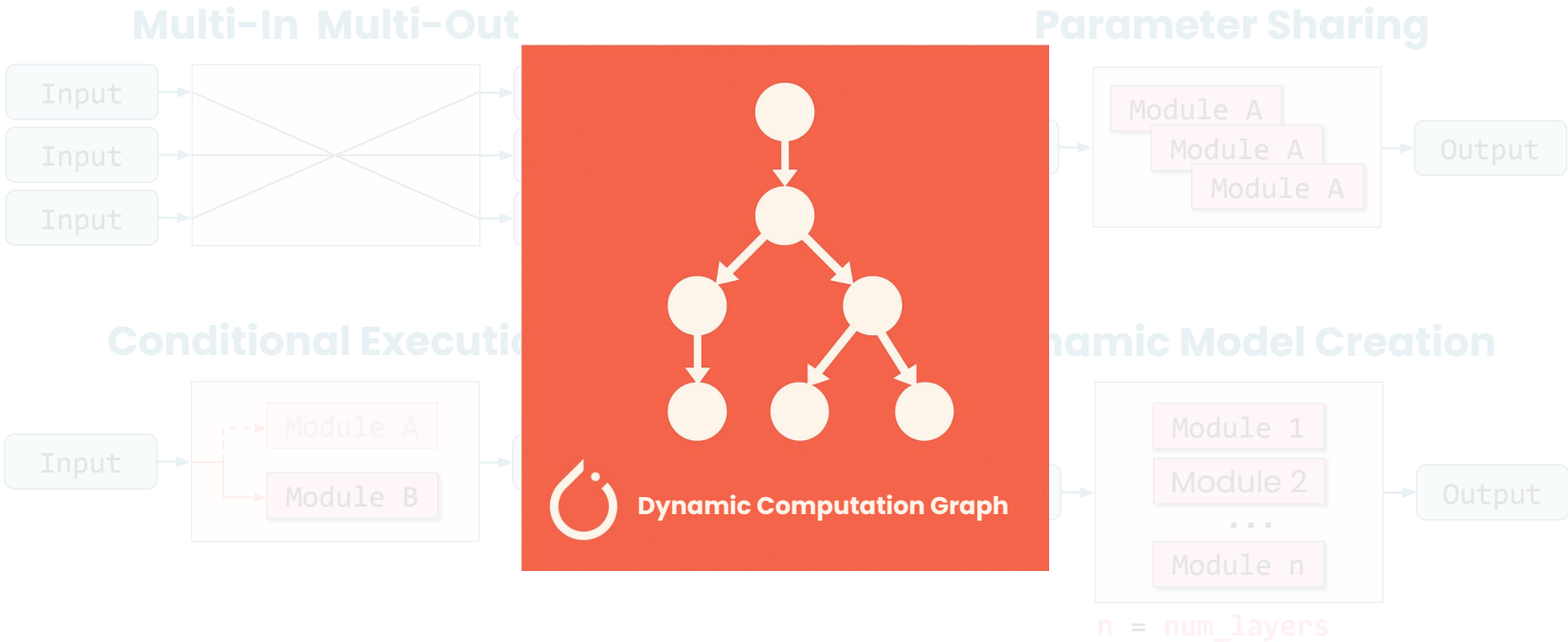


Dynamic Model Creation



`n = num_layers`

Custom Architectures

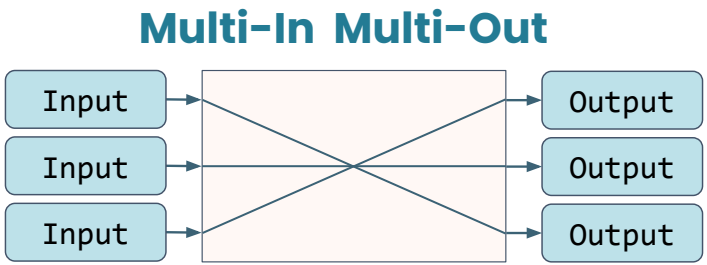


DeepLearning.AI

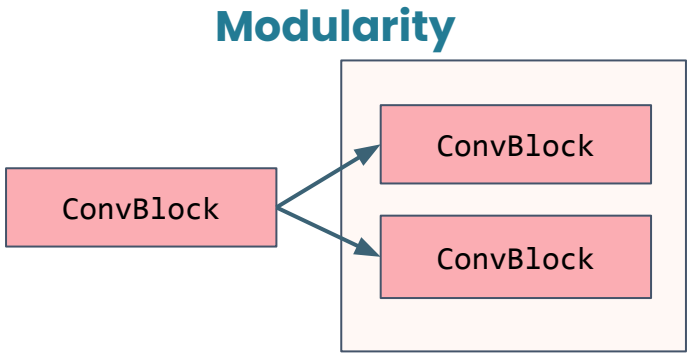
Siamese Networks

Designing Custom Architectures

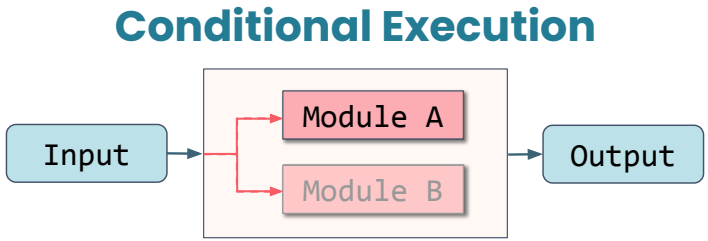
Custom Architectures



Custom Architectures

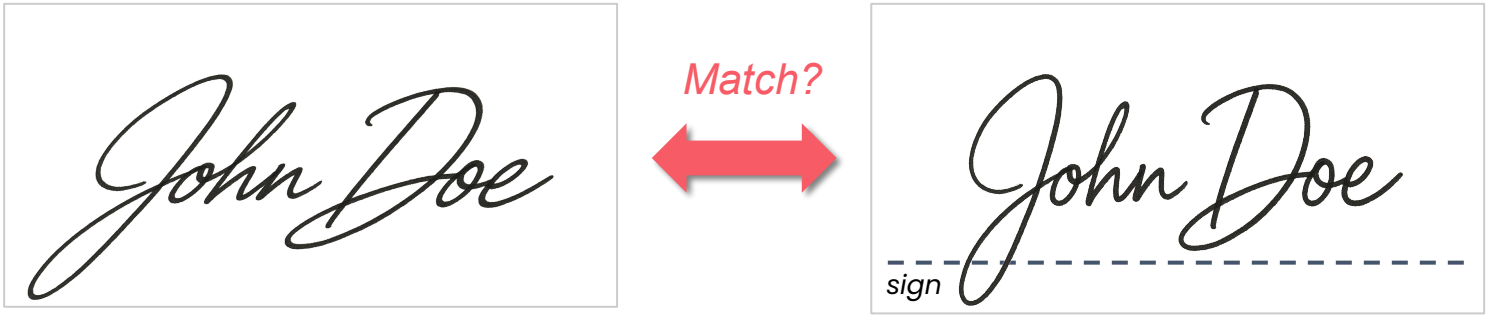


Custom Architectures



Comparing Signatures

On File



Per-Person Classification

Is this **your** signature?

Per-Person Classification

On File

Match!

Is this **your** signature?

Per-Person Classification

Training on YOUR signature

Not Realistic!

Per-Person Classifier

Per-Person Classes

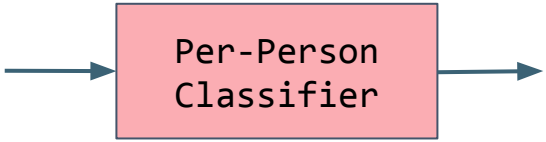
- Nick Lewis
- Miguel Magaña
- Muhammad Mubashar
- John Doe
- Ernesto Cuartas
- Dapinder Dosanjh
- Roberto Reif
- Julián Martínez-Linares
- Renzo Cuadra
- Lucas Coutinho
- Michaelle Perez
- ...

Per-Person Classification

New Customer?

Peter Perez

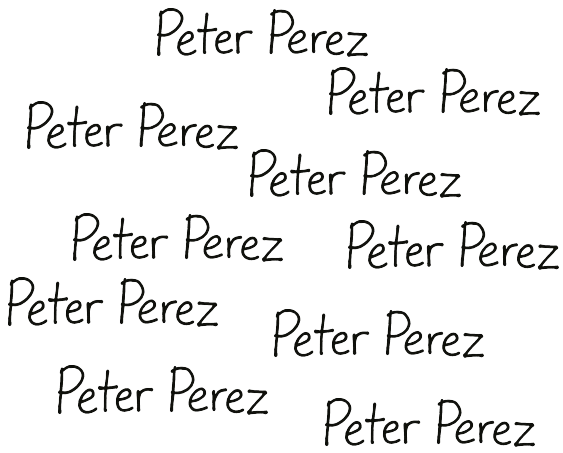
Retrain the model!



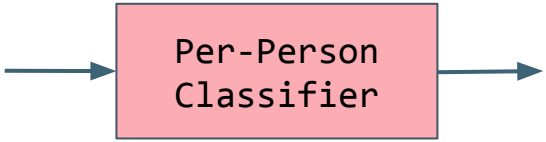
Per-Person Classes

- Nick Lewis
- Miguel Magaña
- Muhammad Mubashar
- John Doe
- Ernesto Cuartas
- Dapinder Dosanjh
- Roberto Reif
- Julián Martínez-Linares
- Renzo Cuadra
- Lucas Coutinho
- Michaelle Perez
- Peter Perez
- ...

Per-Person Classification



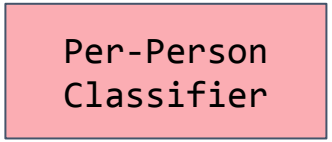
Retrain the model!



Per-Person Classes

- Nick Lewis
- Miguel Magaña
- Muhammad Mubashar
- John Doe
- Ernesto Cuartas
- Dapinder Dosanjh
- Roberto Reif
- Julián Martínez-Linares
- Renzo Cuadra
- Lucas Coutinho
- Michaelle Perez
- **Peter Perez**
- ...

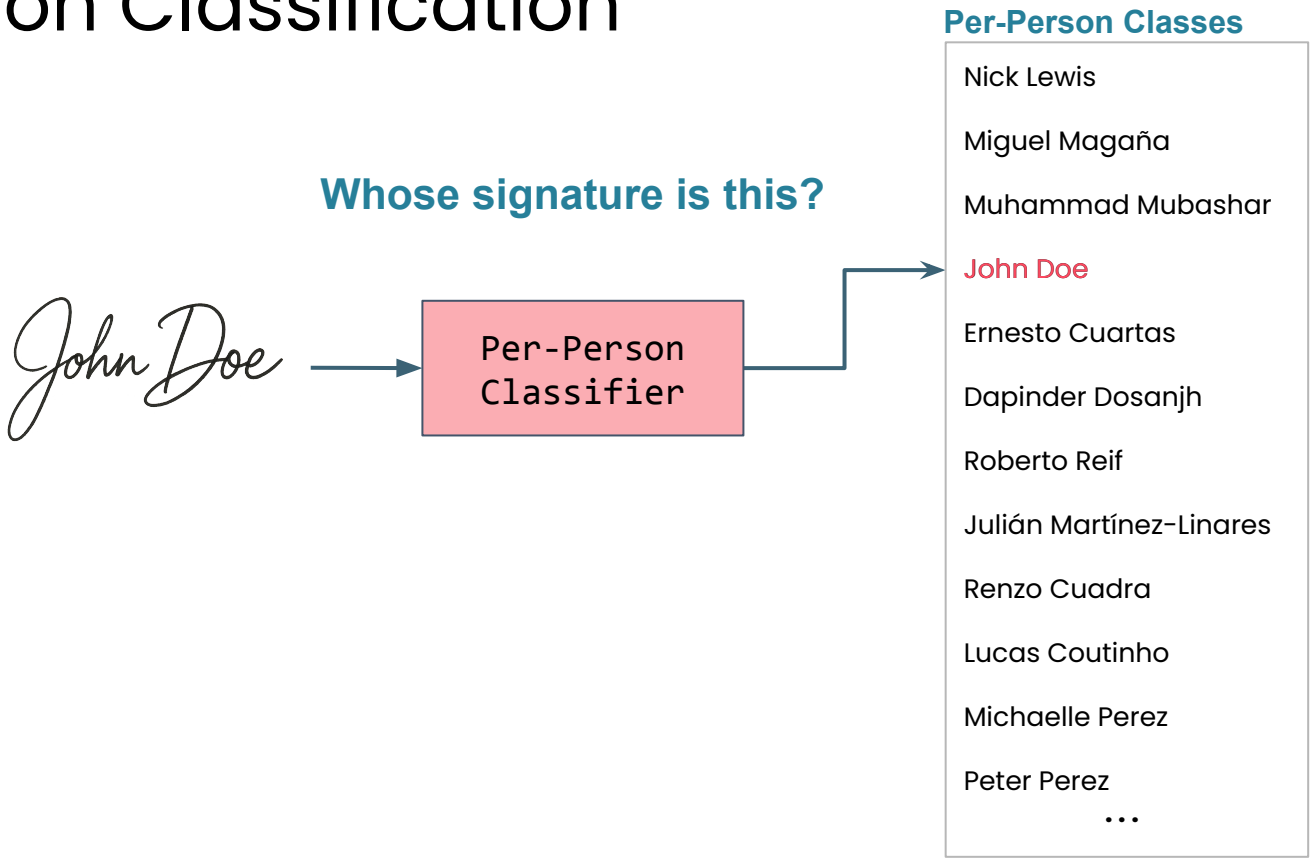
Per-Person Classification



Per-Person Classes

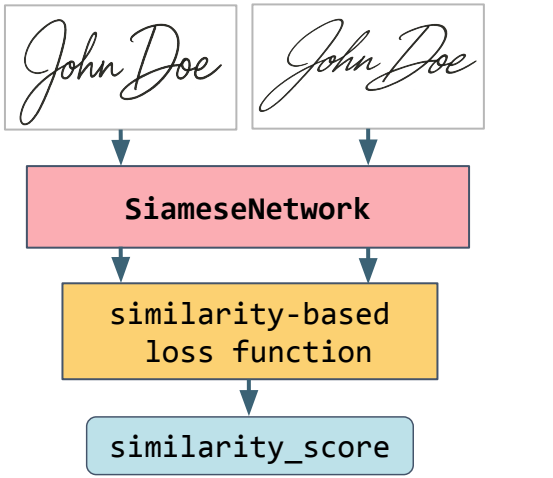
- Nick Lewis
- Miguel Magaña
- Muhammad Mubashar
- John Doe
- Ernesto Cuartas
- Dapinder Dosanjh
- Roberto Reif
- Julián Martínez-Linares
- Renzo Cuadra
- Lucas Coutinho
- Michaelle Perez
- Peter Perez
- ...

Per-Person Classification



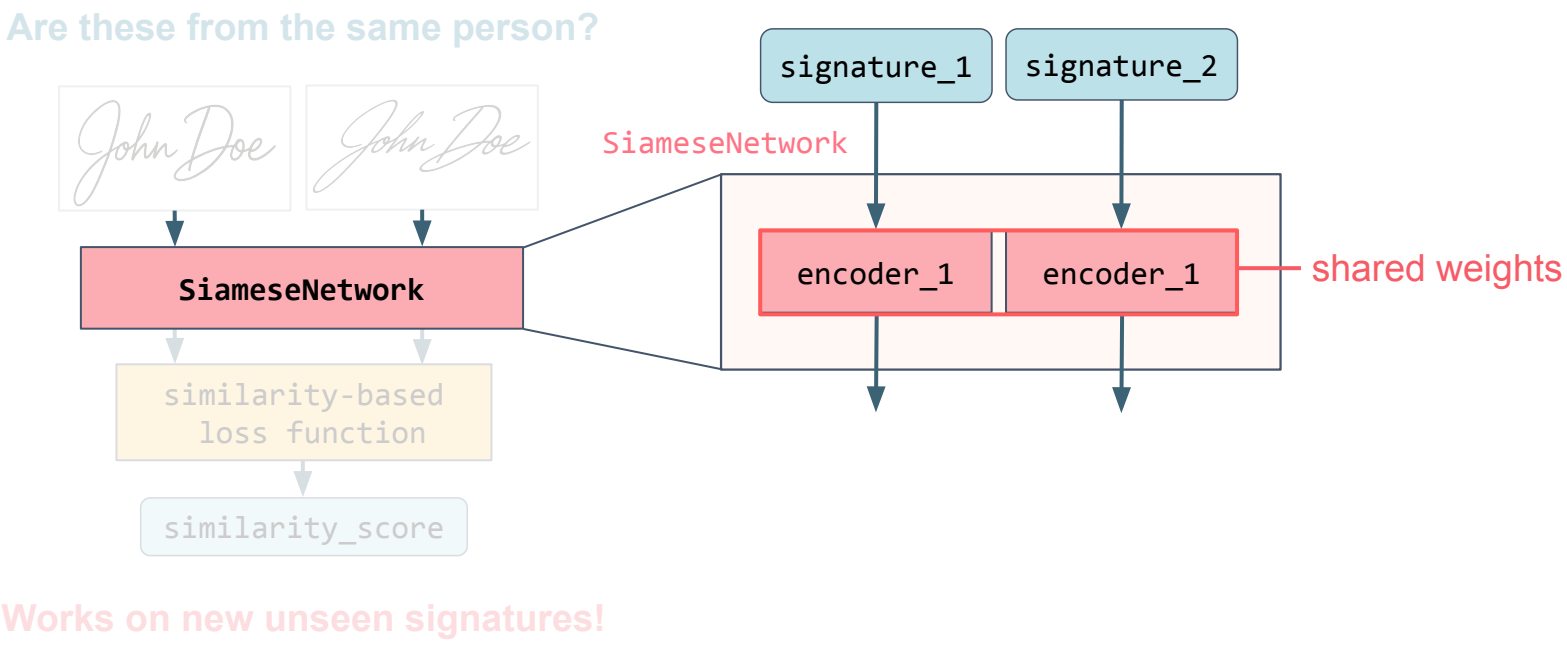
Siamese Networks

Are these from the same person?

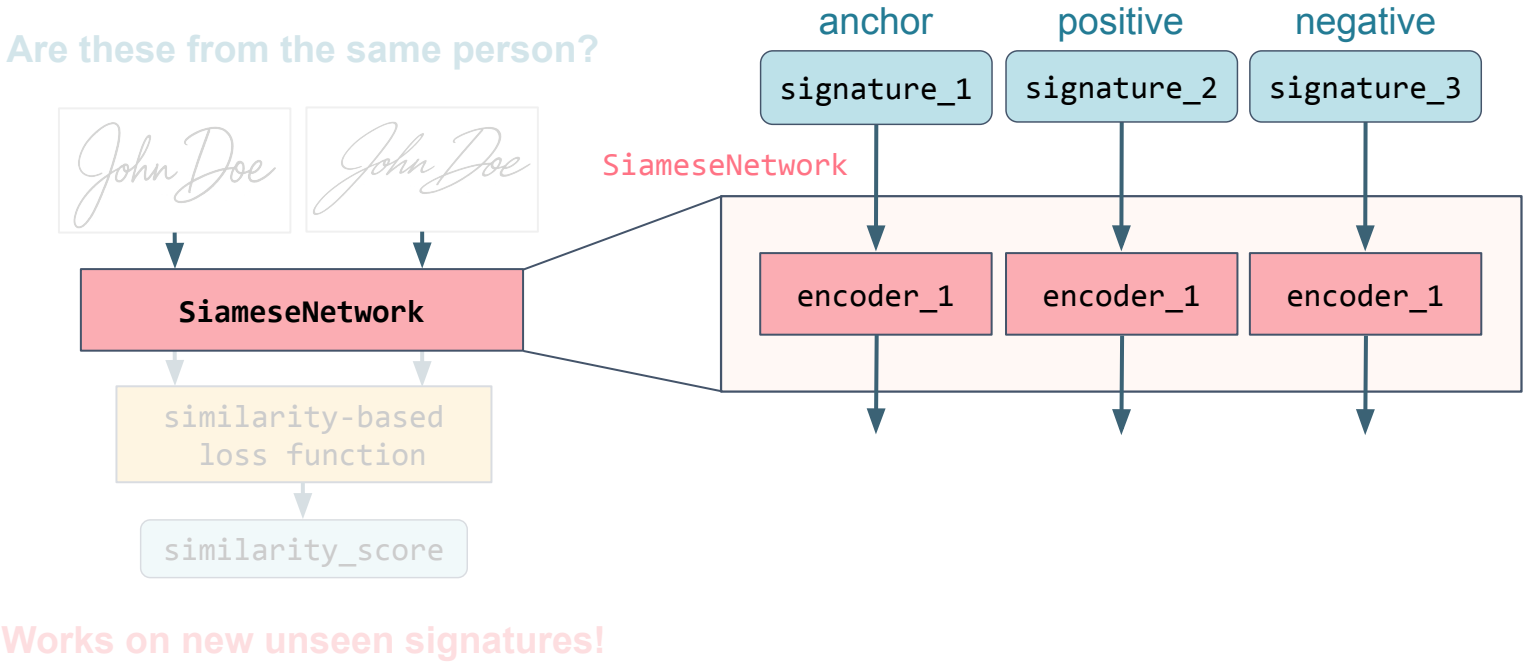


Works on new unseen signatures!

Siamese Networks



Siamese Networks



Works on new unseen signatures!

Siamese Network

Siamese Network

```
class SignatureTripletDataset(Dataset):

    . . .

    def __getitem__(self, index):
        person_id = random.choice(self.user_ids)
        anchor_path, positive_path = random.sample(self.signature_map[person_id]['real'], 2)
        negative_path = random.choice(self.signature_map[person_id]['fake'])

        . . .

        return (anchor_img, positive_img, negative_img)
```

Siamese Network

```
class SignatureTripletDataset(Dataset):

    . . .

    def __getitem__(self, index):
        person_id = random.choice(self.user_ids)
        anchor_path, positive_path = random.sample(self.signature_map[person_id]['real'], 2)
        negative_path = random.choice(self.signature_map[person_id]['fake'])

        . . .

        return (anchor_img, positive_img, negative_img)
```

Siamese Network

```
class SignatureTripletDataset(Dataset):

    . . .

    def __getitem__(self, index):
        person_id = random.choice(self.user_ids)
        anchor_path, positive_path = random.sample(self.signature_map[person_id]['real'], 2)
        negative_path = random.choice(self.signature_map[person_id]['fake'])

        . . .

        return (anchor_img, positive_img, negative_img)
```

Siamese Network

```
class SignatureTripletDataset(Dataset):

    . . .

    def __getitem__(self, index):
        person_id = random.choice(self.user_ids)
        anchor_path, positive_path = random.sample(self.signature_map[person_id]['real'], 2)
        negative_path = random.choice(self.signature_map[person_id]['fake'])

        . . .

        return (anchor_img, positive_img, negative_img)
```

Siamese Network

```
class SignatureTripletDataset(Dataset):

    . . .

    def __getitem__(self, index):
        person_id = random.choice(self.user_ids)
        anchor_path, positive_path = random.sample(self.signature_map[person_id]['real'], 2)
        negative_path = random.choice(self.signature_map[person_id]['fake'])

        . . .

        return (anchor_img, positive_img, negative_img)
```

Siamese Network

```
class SignatureTripletDataset(Dataset):

    . . .

    def __getitem__(self, index):
        person_id = random.choice(self.user_ids)
        anchor_path, positive_path = random.sample(self.signature_map[person_id]['real'], 2)
        negative_path = random.choice(self.signature_map[person_id]['fake'])

        . . .

        return (anchor_img, positive_img, negative_img)
```

Siamese Network

```
class SignatureTripletDataset(Dataset):

    . . .

    def __getitem__(self, index):
        person_id = random.choice(self.user_ids)
        anchor_path, positive_path = random.sample(self.signature_map[person_id]['real'], 2)
        negative_path = random.choice(self.signature_map[person_id]['fake'])

        . . .

        return (anchor_img, positive_img, negative_img)
```

Siamese Network

```
for anchor, positive, negative in dataloader:
   anchor_out, positive_out, negative_out = SiameseNetwork(anchor, positive, negative)
```

Siamese Network

```
for anchor, positive, negative in dataloader:
    anchor_out, positive_out, negative_out = SiameseNetwork(anchor, positive, negative)
```

Siamese Network

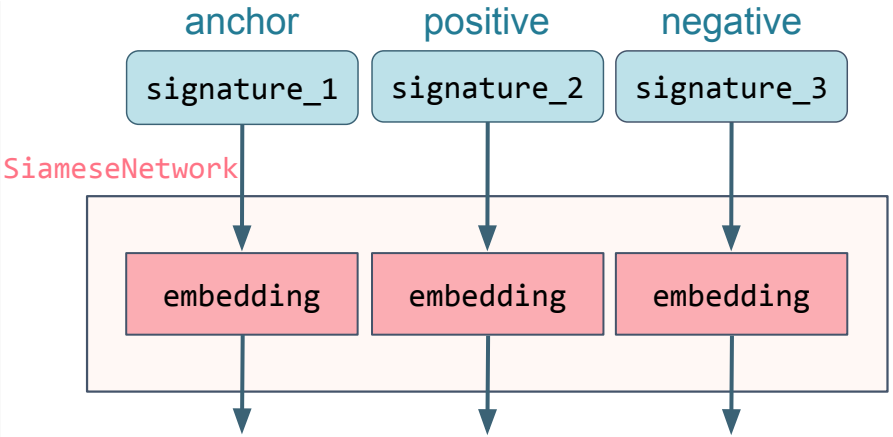
```
class SiameseNetwork(nn.Module):
    def __init__(self, embedding_network):
        super().__init__()
        self.embedding_network = embedding_network

    def forward(self, anchor, positive, negative):
        a = self.embedding_network(anchor)
        p = self.embedding_network(positive)
        n = self.embedding_network(negative)
        return a, p, n
```

Siamese Network

```
class SiameseNetwork(nn.Module):
    def __init__(self, embedding_network):
        super().__init__()
        self.embedding_network = embedding_network

    def forward(self, anchor, positive, negative):
        a = self.embedding_network(anchor)
        p = self.embedding_network(positive)
        n = self.embedding_network(negative)
        return a, p, n
```

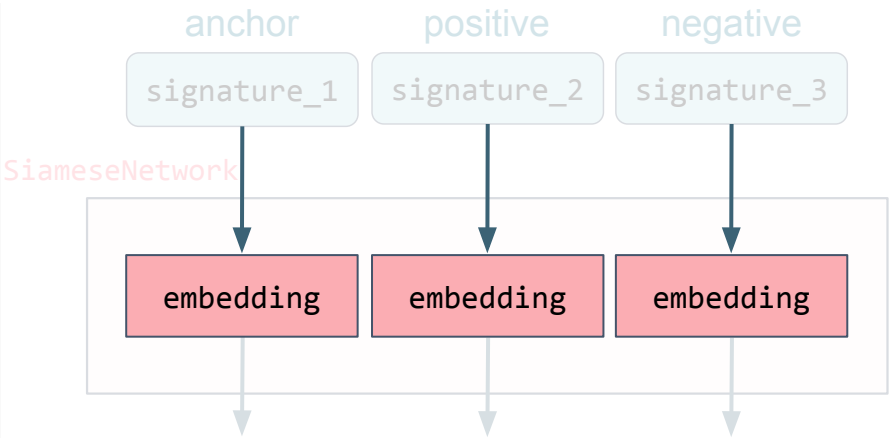
Siamese Network

```
class SiameseNetwork(nn.Module):
    def __init__(self, embedding_network):
        super().__init__()
        self.embedding_network = embedding_network

    def forward(self, anchor, positive, negative):
        a = self.embedding_network(anchor)
        p = self.embedding_network(positive)
        n = self.embedding_network(negative)
        return a, p, n
```



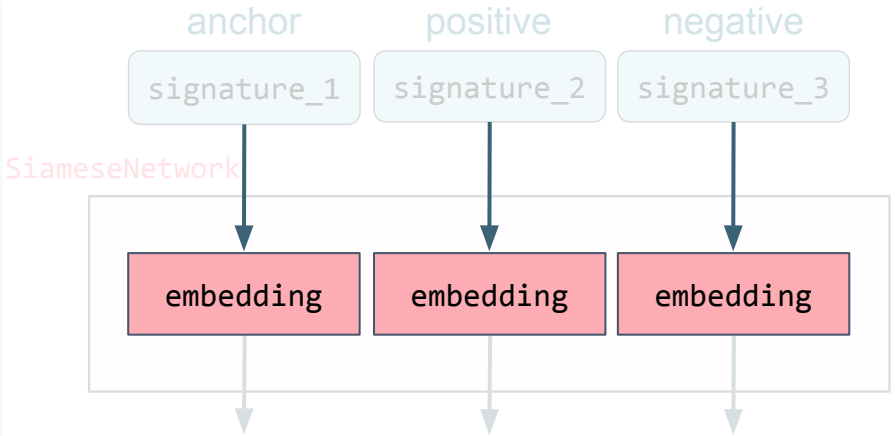
Siamese Network

```
class SiameseNetwork(nn.Module):
    def __init__(self, embedding_network):
        super().__init__()
        self.embedding_network = embedding_network

    def forward(self, anchor, positive, negative):
        a = self.embedding_network(anchor)
        p = self.embedding_network(positive)
        n = self.embedding_network(negative)
        return a, p, n
```



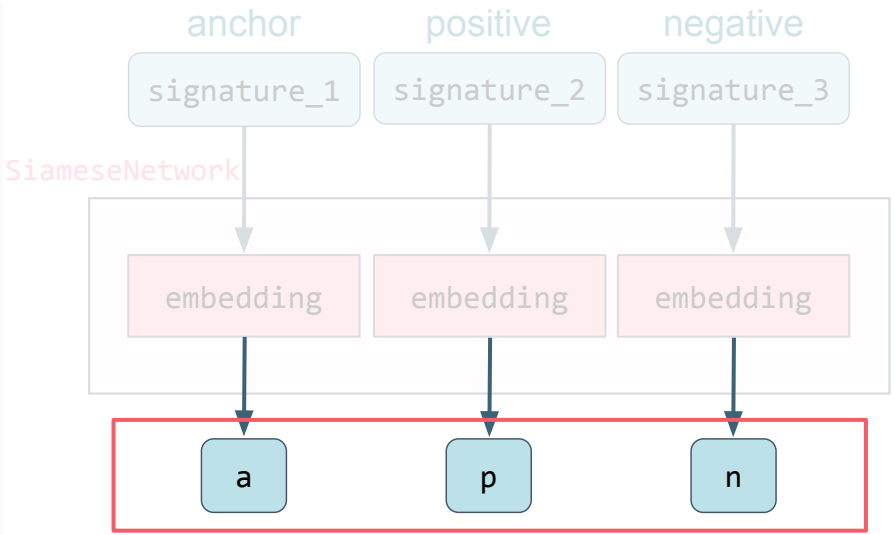
Siamese Network

```
class SiameseNetwork(nn.Module):
    def __init__(self, embedding_network):
        super().__init__()
        self.embedding_network = embedding_network

    def forward(self, anchor, positive, negative):
        a = self.embedding_network(anchor)
        p = self.embedding_network(positive)
        n = self.embedding_network(negative)
        return a, p, n
```



Siamese Network

```
class SiameseNetwork(nn.Module):
    def __init__(self, embedding_network):
        super().__init__()
        self.embedding_network = embedding_network

    def forward(self, anchor, positive, negative):
        a = self.embedding_network(anchor)
        p = self.embedding_network(positive)
        n = self.embedding_network(negative)
        return a, p, n
```



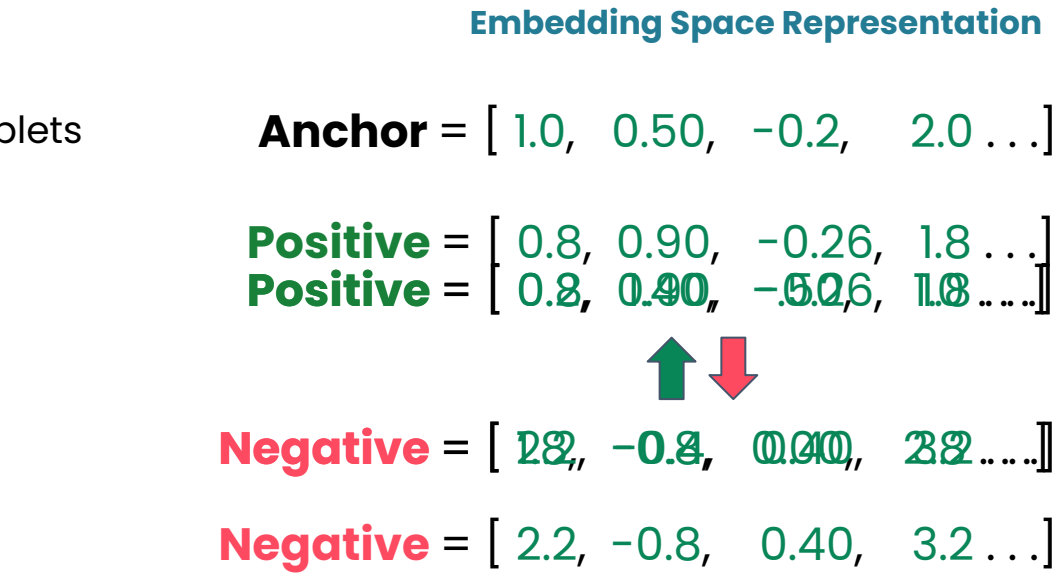
Siamese Network

```
anchor_out, positive_out, negative_out = model(anchor, positive, negative)
loss = loss_fcn(anchor_out, positive_out, negative_out)  # triplet margin loss
```

- **Pull anchor and positive together**
- **Push anchor and negative apart**

Siamese Network: Loss

- **Contrastive Loss:** Pairs
- **Triplet Margin Loss:** Triplets

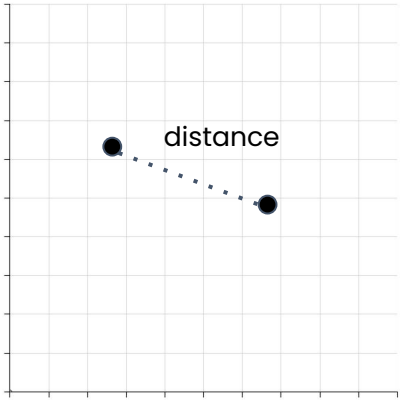


Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```

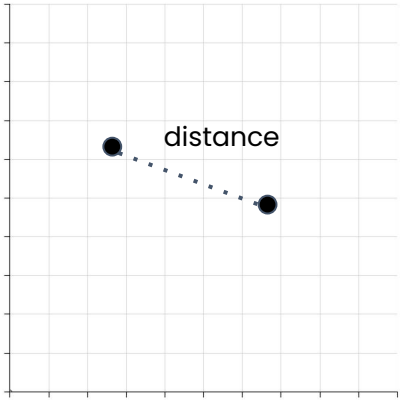
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



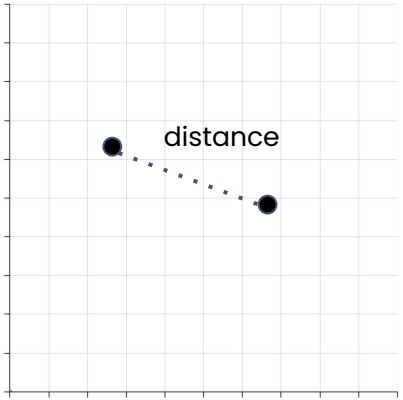
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



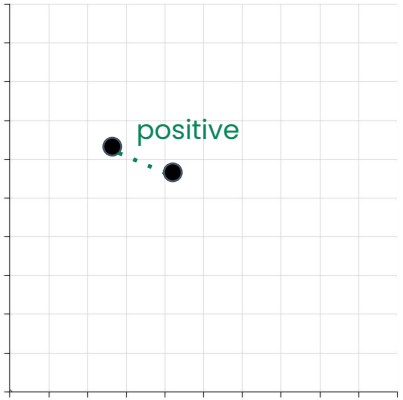
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



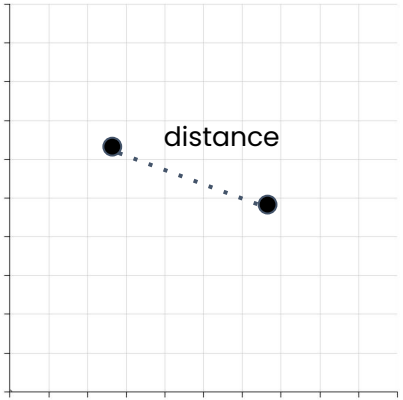
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



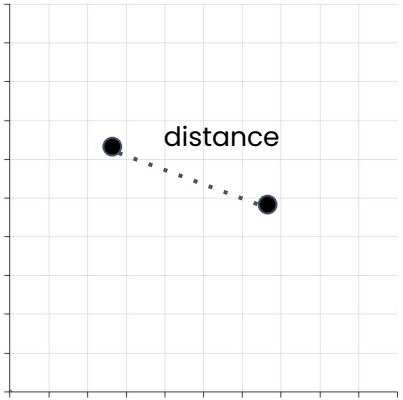
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



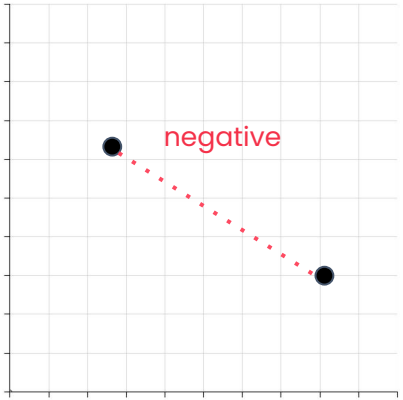
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



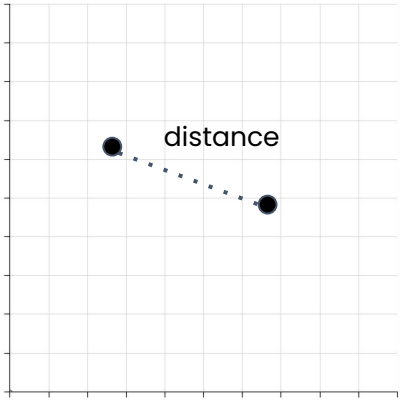
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



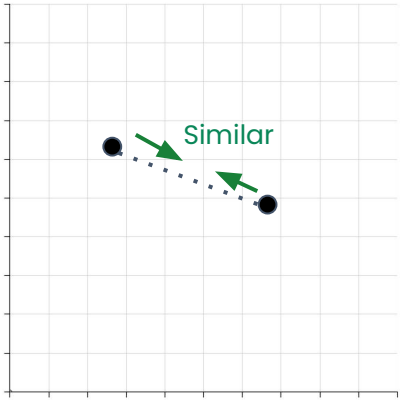
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



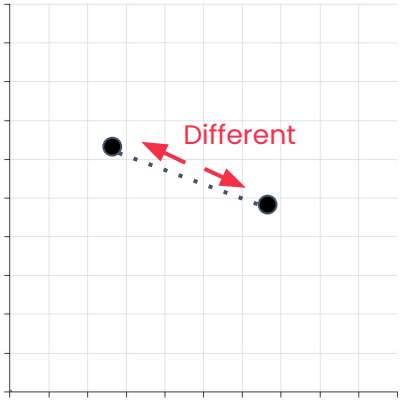
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



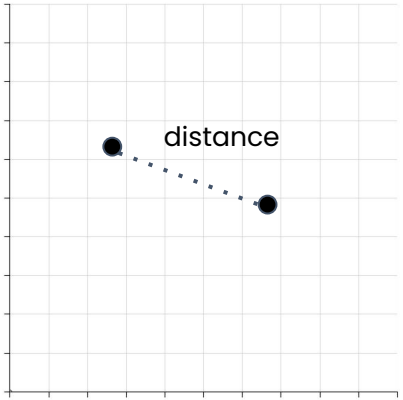
Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



Contrastive Loss

```
class ContrastiveLoss(nn.Module):
    def __init__(self, margin=2.0):
        super().__init__()
        self.margin = margin

    def forward(self, output1, output2, label):
        euclidean_distance = F.pairwise_distance(
            output1, output2, keepdim=True)

        similar_loss = torch.pow(euclidean_distance, 2)

        dissimilar_loss = torch.pow(torch.clamp(
            self.margin-euclidean_distance, min=0.0), 2)

        loss = torch.mean(
            label*similar_loss + (1-label)*dissimilar_loss)
        return loss
```



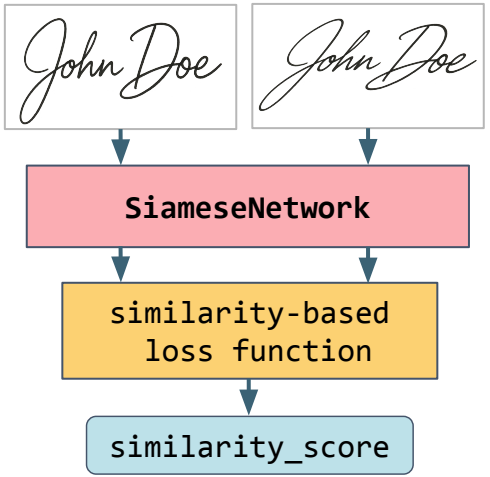
Triplet Margin Loss

```
triplet_loss_function = nn.TripletMarginLoss(margin=margin, p=2)
loss = triplet_loss_function(anchor_out, positive_out, negative_out)
```

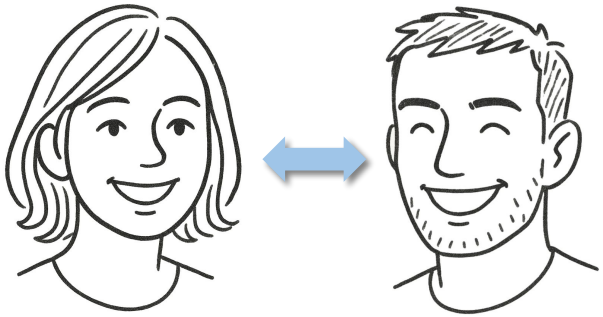
Loss

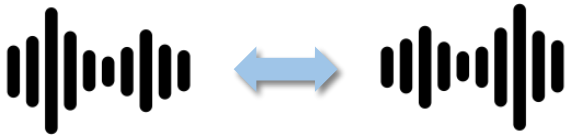
```
loss = loss_fcn(output1, output2, label)
```

Siamese Network



Siamese Network





The cat sat on the mat ⟷ *The dog sat on the mat*

Siamese Network

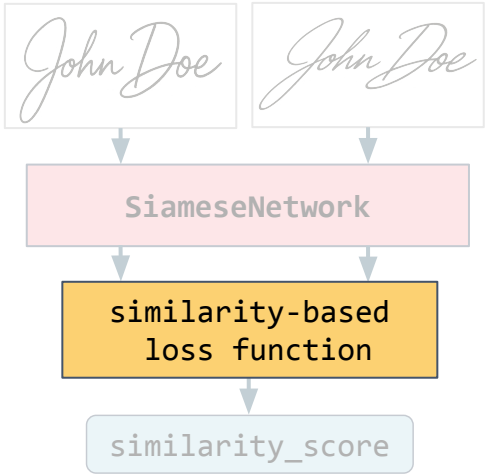
```
class SiameseNetwork(nn.Module):
    def __init__(self, embedding_network):
        super().__init__()
        self.embedding_network = embedding_network  # same CNN reused

    def forward(self, anchor, positive, negative):
        a = self.embedding_network(anchor)
        p = self.embedding_network(positive)
        n = self.embedding_network(negative)
        return a, p, n
```

DeepLearning.AI

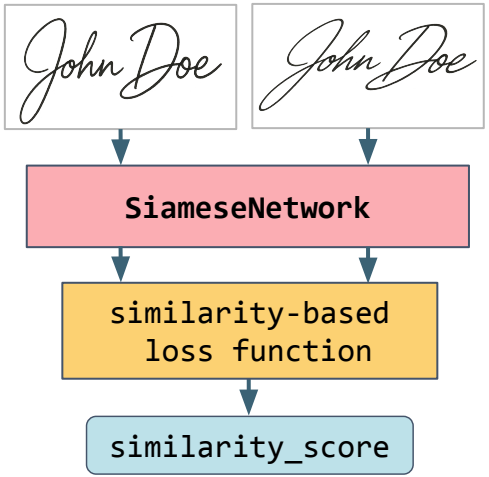
ResNet

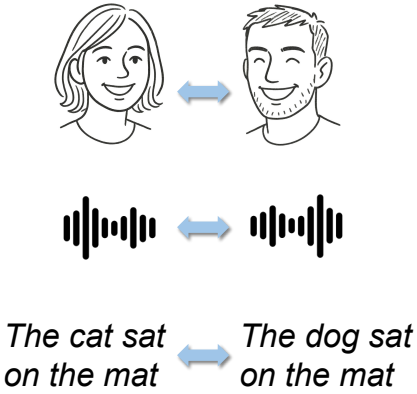
Designing Custom Architectures

Siamese Network

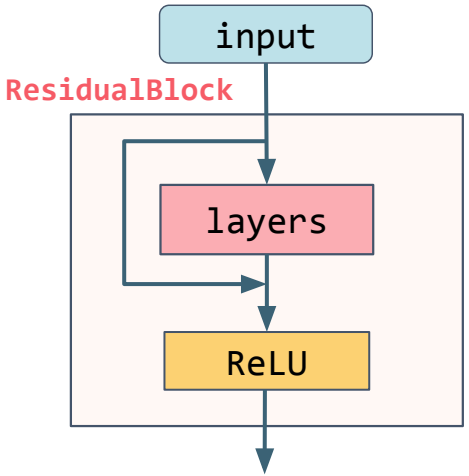


Siamese Network

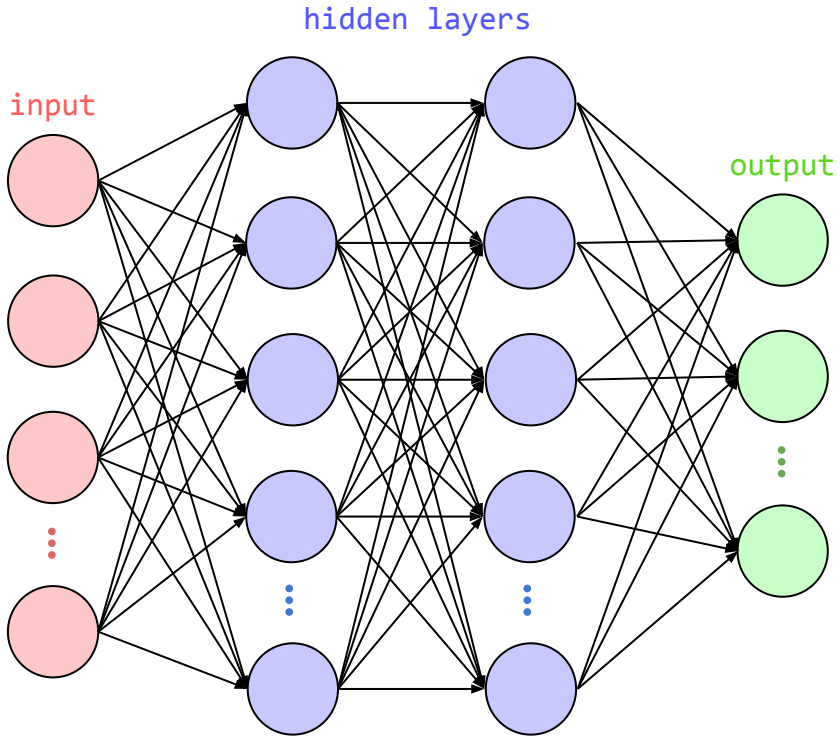




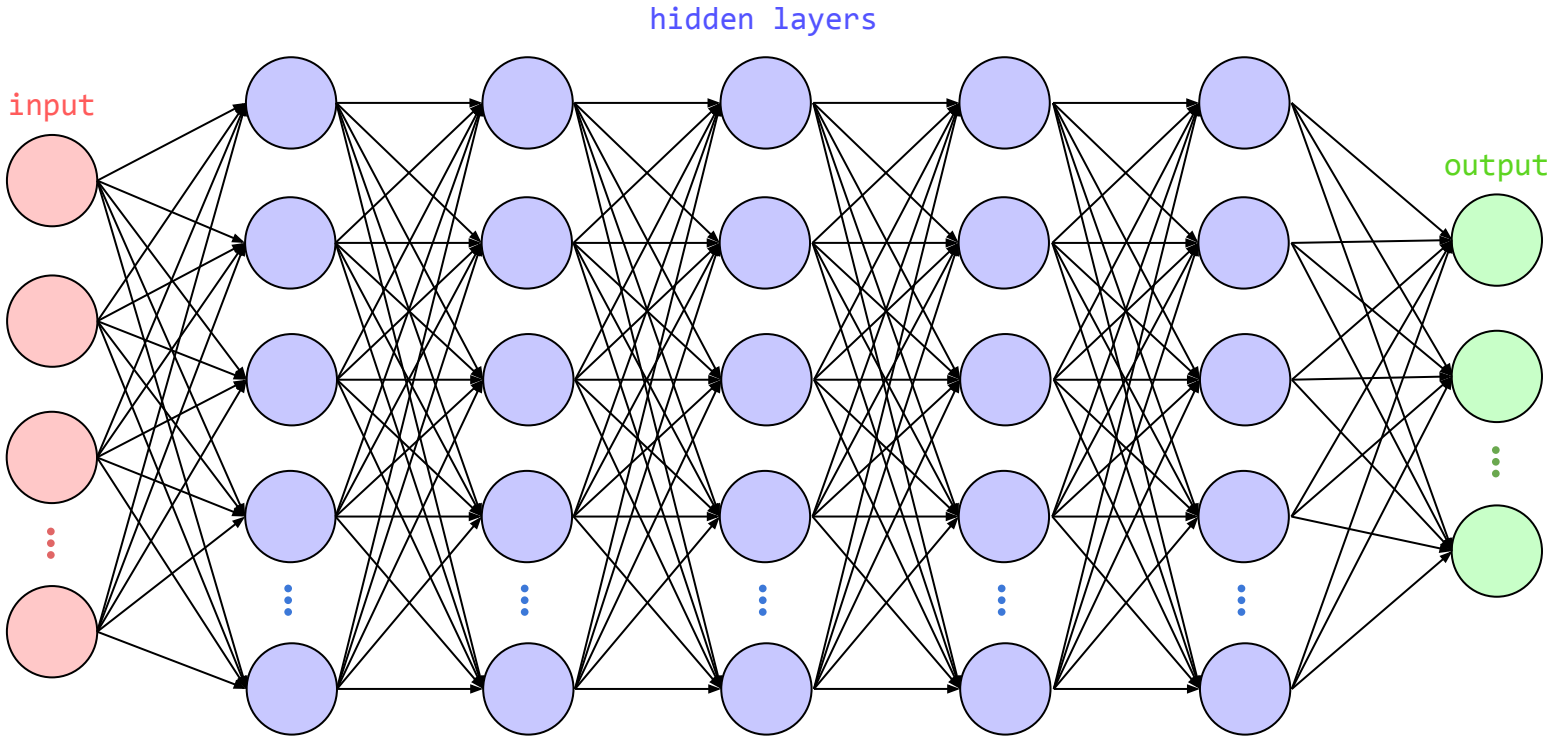
ResNet



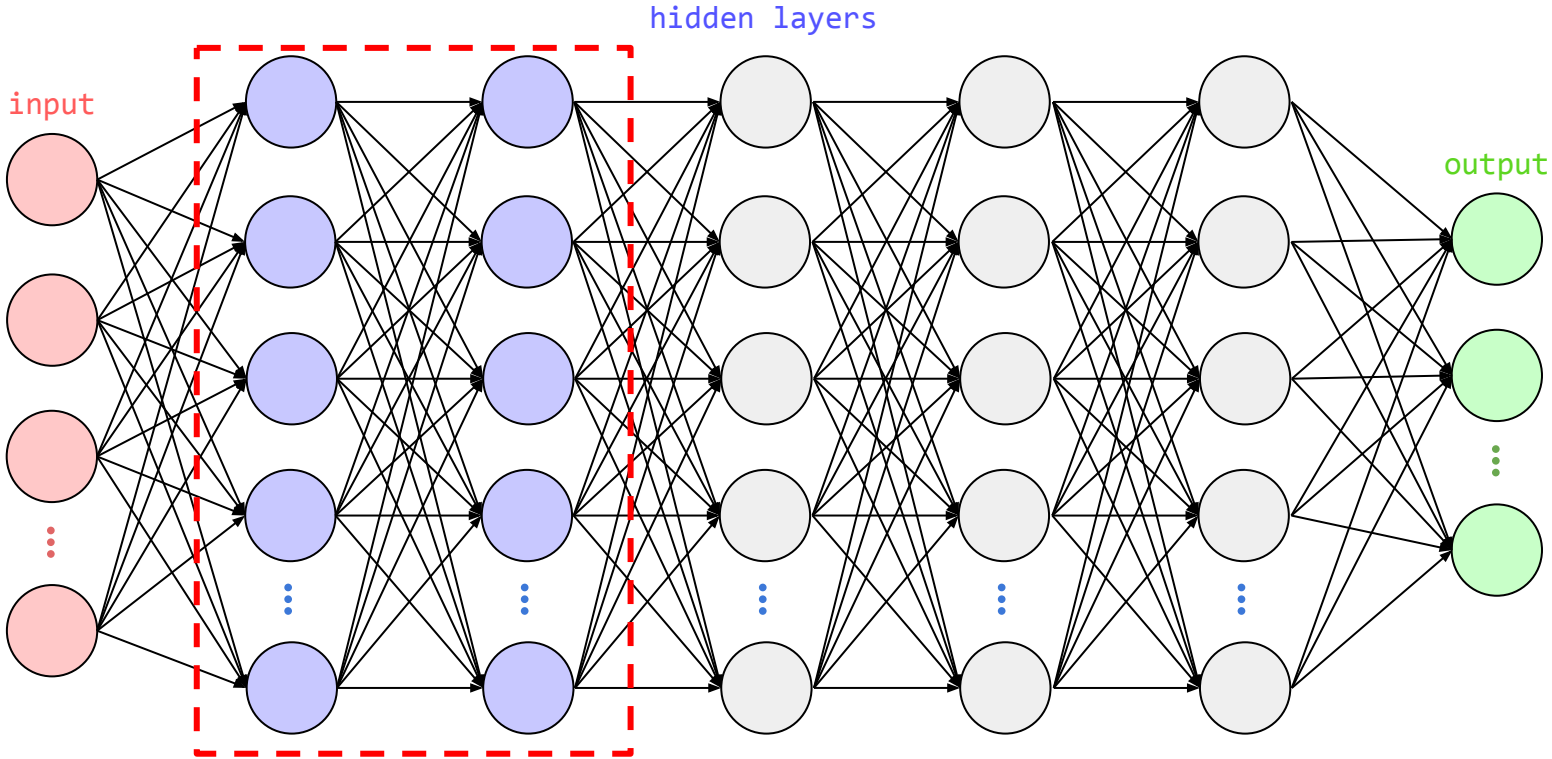
Deep Networks



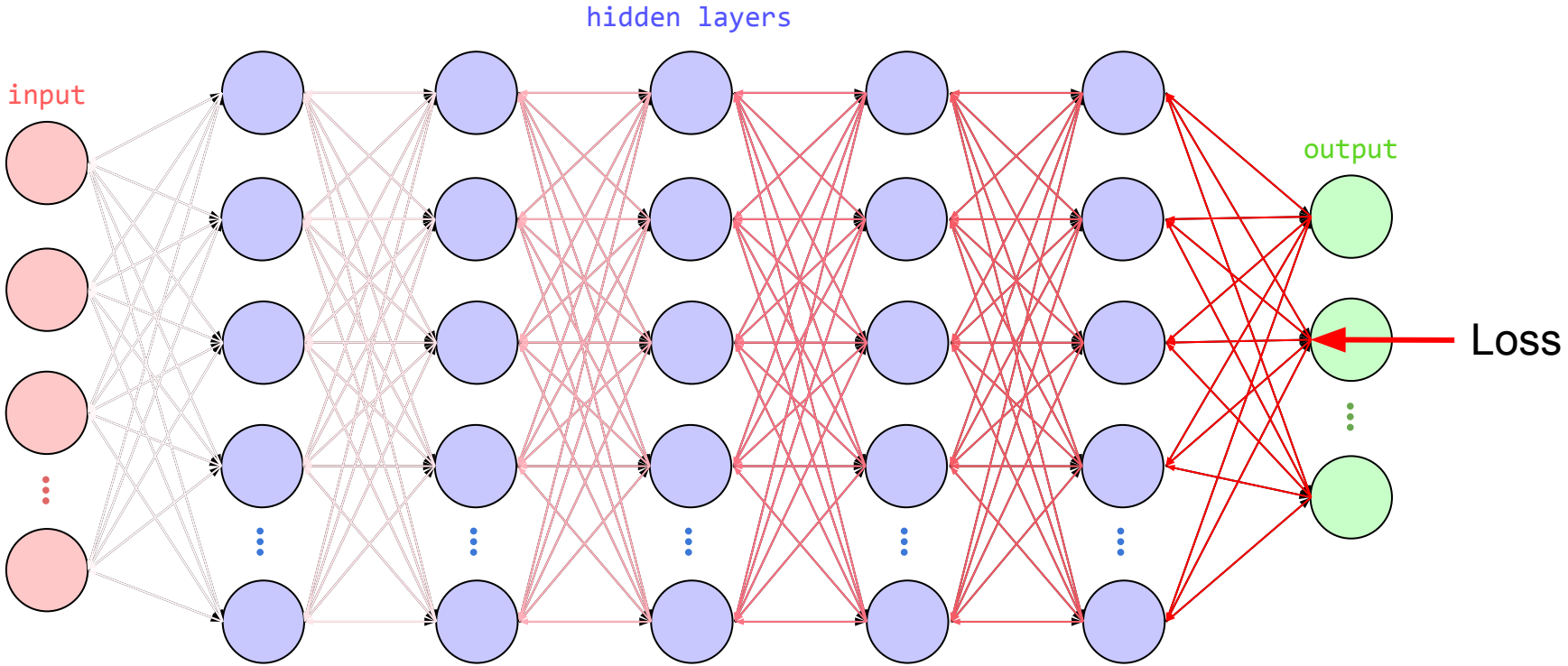
Deep Networks



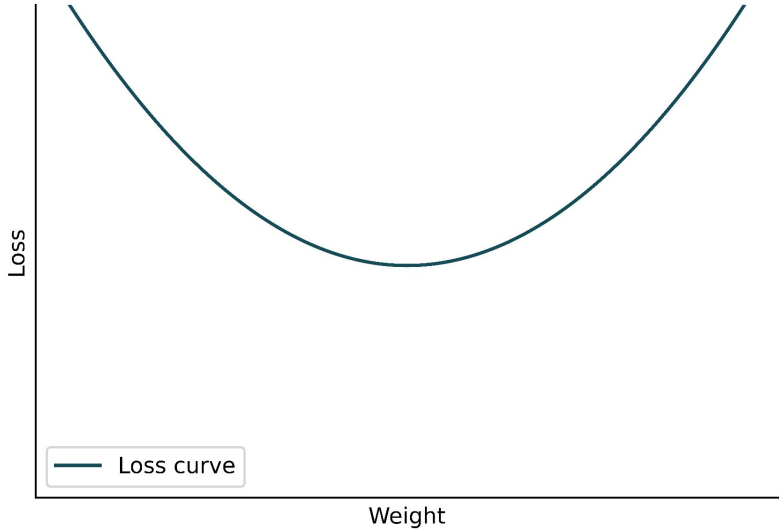
Deep Networks



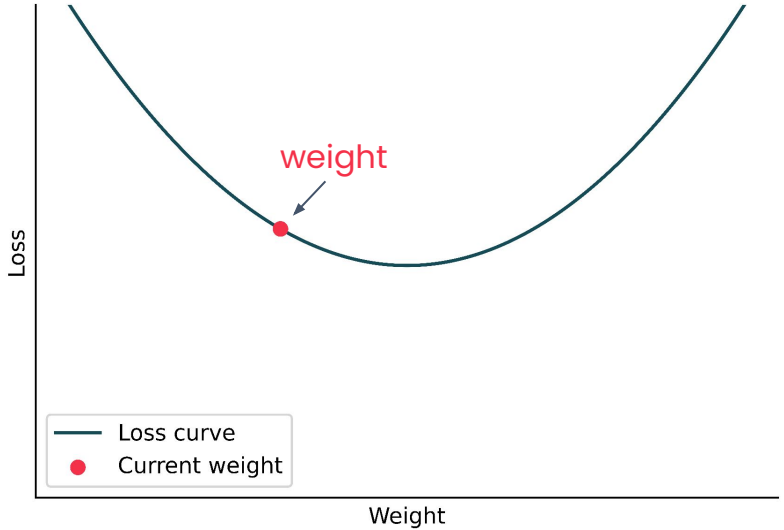
Deep Networks



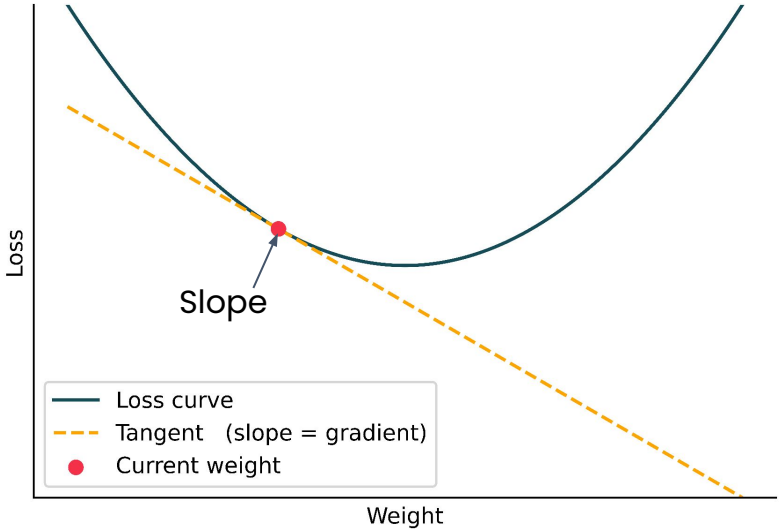
Gradients



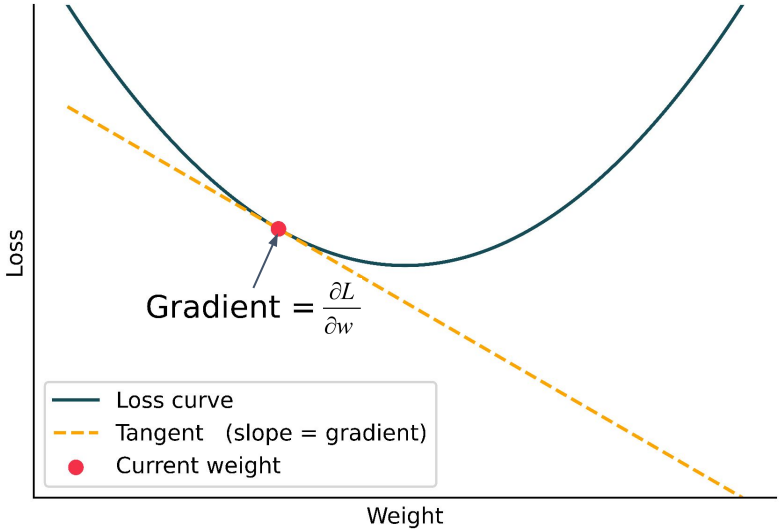
Gradients



Gradients


Loss
Slope
Loss curve
Tangent (slope = gradient)
Current weight
Weight

Gradients


Loss
Weight
Gradient = $\frac{\partial L}{\partial w}$
Loss curve
Tangent (slope = gradient)
Current weight

Calculating Gradients

$$\frac{\partial L}{\partial w}$$

Gradient of w

Calculating Gradients

small num * small num

$$\frac{\partial L}{\partial w} = .07 * -.03 * .12 \ldots * .09$$

Gradient of w

Calculating Gradients

small num * small num **= smaller numbers**

$$\frac{\partial L}{\partial w} = .07 * -.03 * .12 \ldots * .09$$

Gradient of w

Calculating Gradients

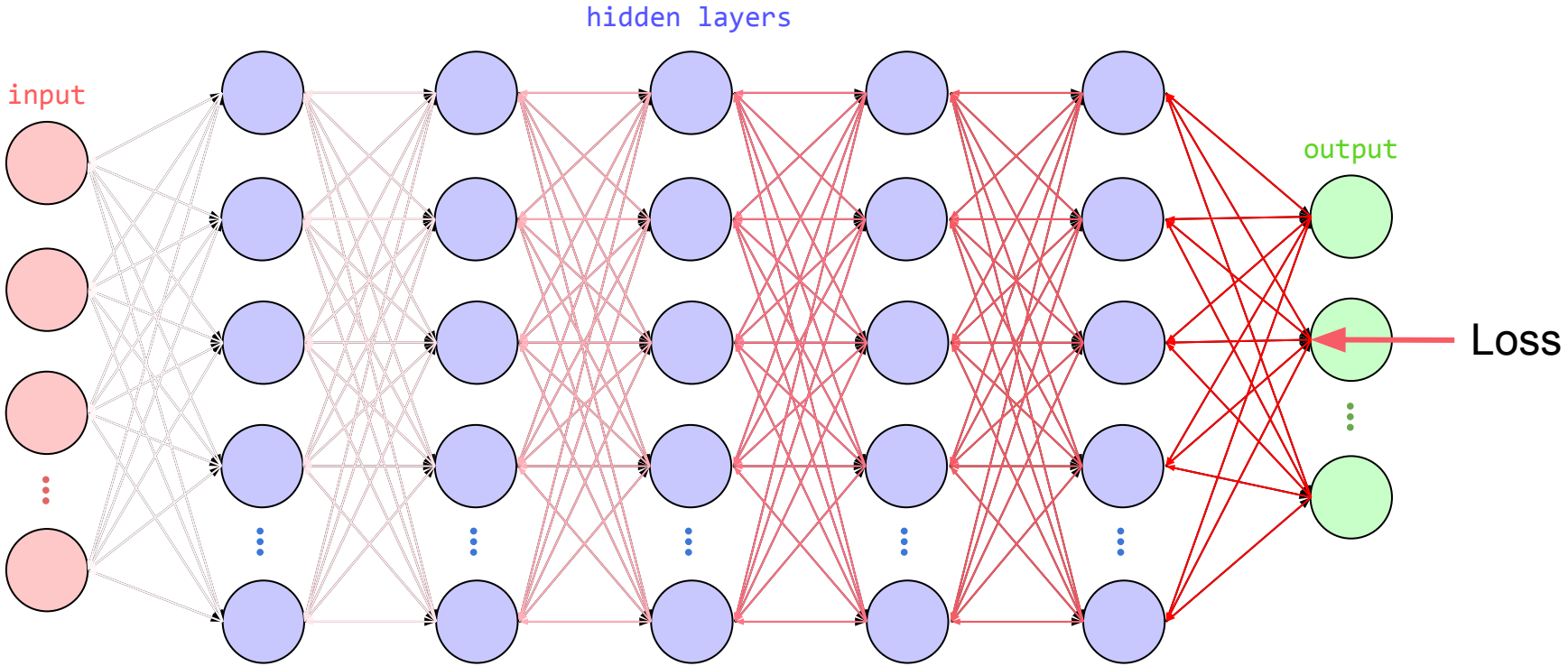
small num * small num **= smaller numbers**

$$\frac{\partial L}{\partial w} = .07 * -.03 * .12 \ldots\ldots * .09 = .00002$$

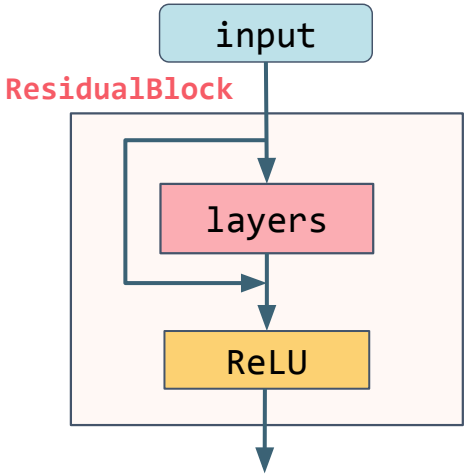
Gradient of w

Adjustment Barely Matters

Vanishing Gradient Problem



ResNet



Train 100+ layers

Simple ResNet

```
class SimpleResNet(nn.Module):
...
    def forward(self, x):
        x = self.initial_block(x)
        x = self.res_block1(x)
        x = self.res_block2(x)
        x = self.res_block3(x)
        x = self.final_block(x)
        return x
```

Simple ResNet

```
class SimpleResNet(nn.Module):
...
    def forward(self, x):
        x = self.initial_block(x)
        x = self.res_block1(x)
        x = self.res_block2(x)
        x = self.res_block3(x)
        x = self.final_block(x)
        return x
```

Simple ResNet

```
class SimpleResNet(nn.Module):
...
    def forward(self, x):
        x = self.initial_block(x)
        x = self.res_block1(x)
        x = self.res_block2(x)
        x = self.res_block3(x)
        x = self.final_block(x)
        return x
```

Simple ResNet

```
class SimpleResNet(nn.Module):
...
    def forward(self, x):
        x = self.initial_block(x)
        x = self.res_block1(x)
        x = self.res_block2(x)
        x = self.res_block3(x)
        x = self.final_block(x)
        return x
```

Simple ResNet

```
class SimpleResNet(nn.Module):
...
    def forward(self, x):
        x = self.initial_block(x)
        x = self.res_block1(x)
        x = self.res_block2(x)
        x = self.res_block3(x)
        x = self.final_block(x)
        return x
```

Initial **conv** block

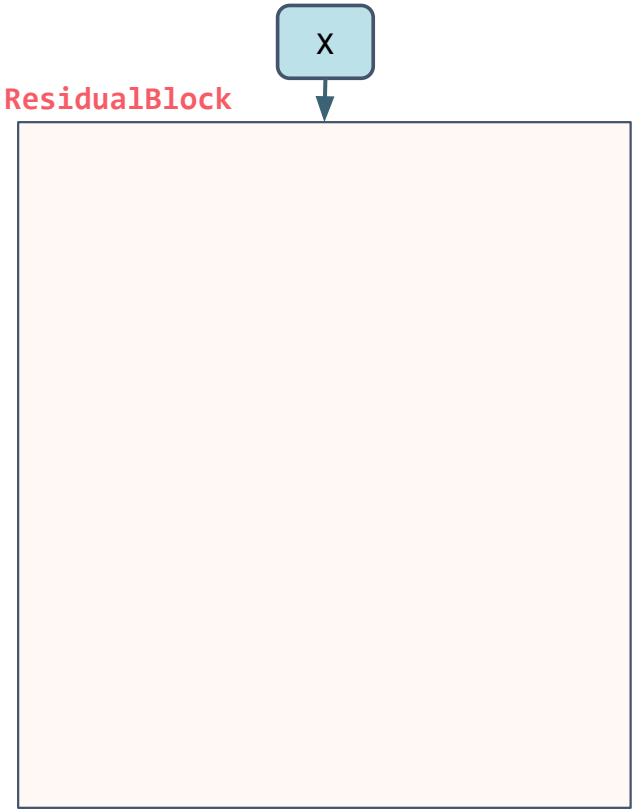
Final **linear** block

Simple ResNet

```
class SimpleResNet(nn.Module):
...
    def forward(self, x):
        x = self.initial_block(x)
        x = self.res_block1(x)
        x = self.res_block2(x)
        x = self.res_block3(x)
        x = self.final_block(x)
        return x
```

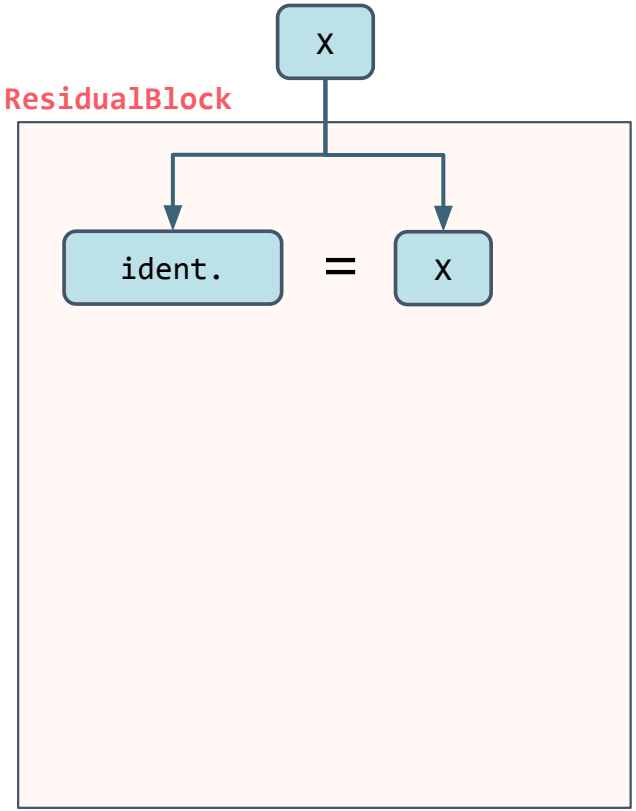
Residual Blocks

```
class ResidualBlock(nn.Module):
    ...
    def forward(self, x):
        identity = x

        # Initial forward pass
        out = self._initial_forward(x)

        # Apply downsampling if identity exists
        if self.downsample is not None:
            identity = self.downsample(x)

        # Residual connection
        out += identity

        # Final ReLU activation
        out = F.relu(out)
        return out
```



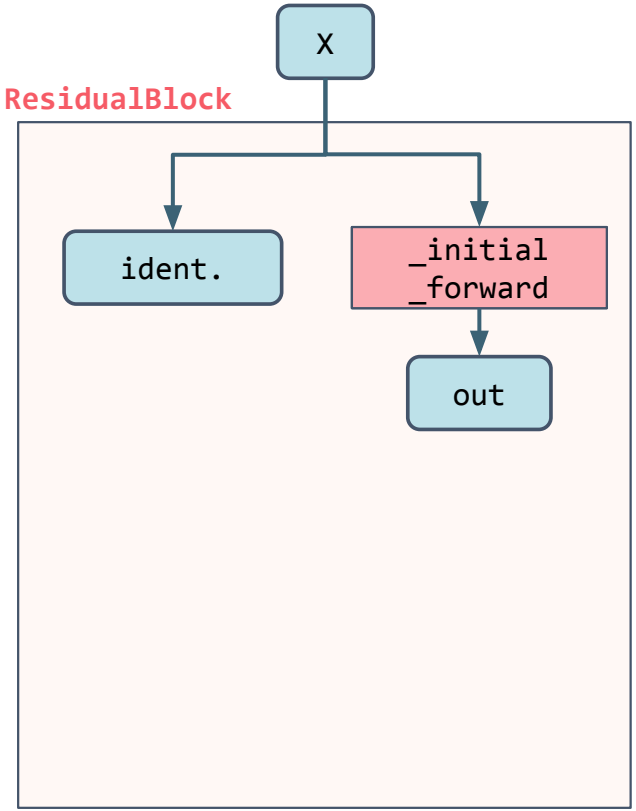
Residual Blocks

```
class ResidualBlock(nn.Module):
    ...
    def forward(self, x):
        identity = x

        # Initial forward pass
        out = self._initial_forward(x)

        # Apply downsampling if identity exists
        if self.downsample is not None:
            identity = self.downsample(x)

        # Residual connection
        out += identity

        # Final ReLU activation
        out = F.relu(out)
        return out
```



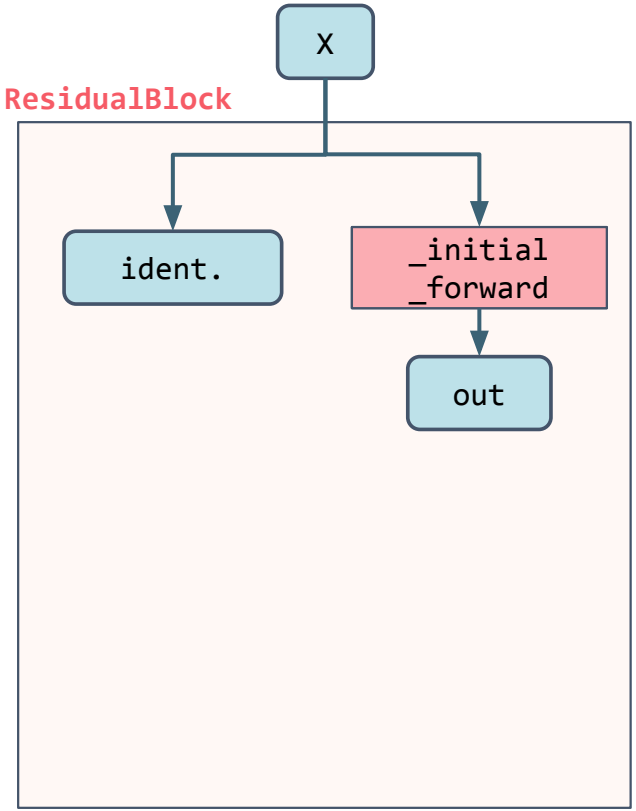
Residual Blocks

```
class ResidualBlock(nn.Module):
    ...
    def forward(self, x):
        identity = x

        # Initial forward pass
        out = self._initial_forward(x)

        # Apply downsampling if identity exists
        if self.downsample is not None:
            identity = self.downsample(x)

        # Residual connection
        out += identity

        # Final ReLU activation
        out = F.relu(out)
        return out
```



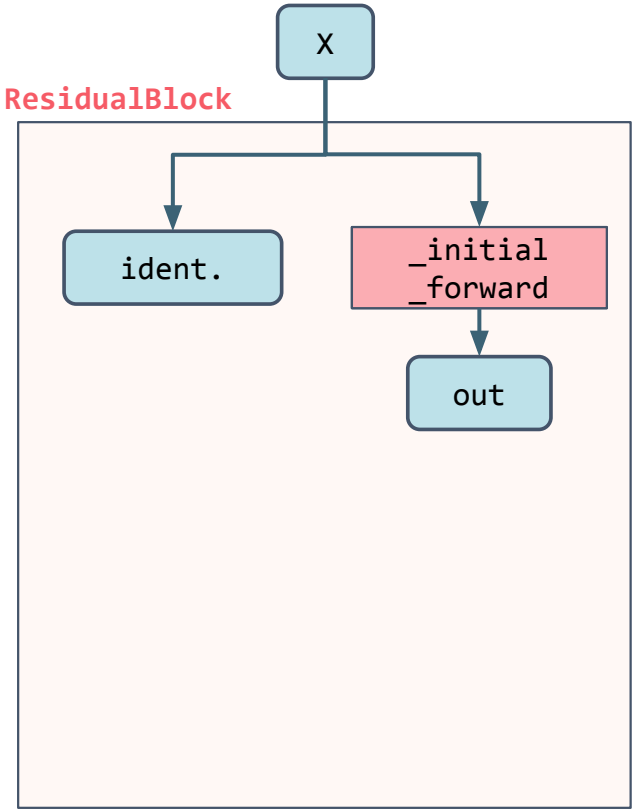
Residual Blocks

```
class ResidualBlock(nn.Module):
...
    def _initial_forward(self, x):
        out = self.conv_block_1(x)
        out = self.conv_block_2(out)
        return out
```



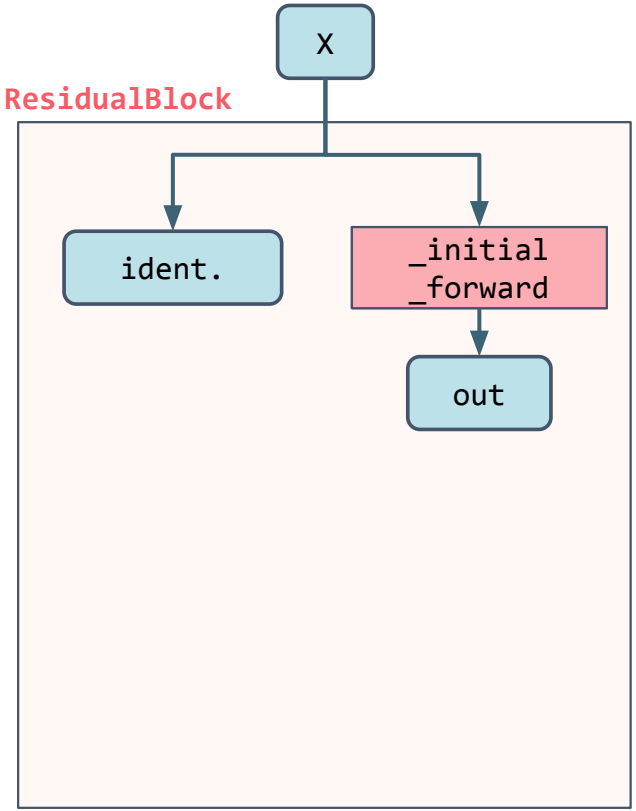
Residual Blocks

```
class ResidualBlock(nn.Module):
...
    def _initial_forward(self, x):
        out = self.conv_block_1(x)
        out = self.conv_block_2(out)
        return out
```



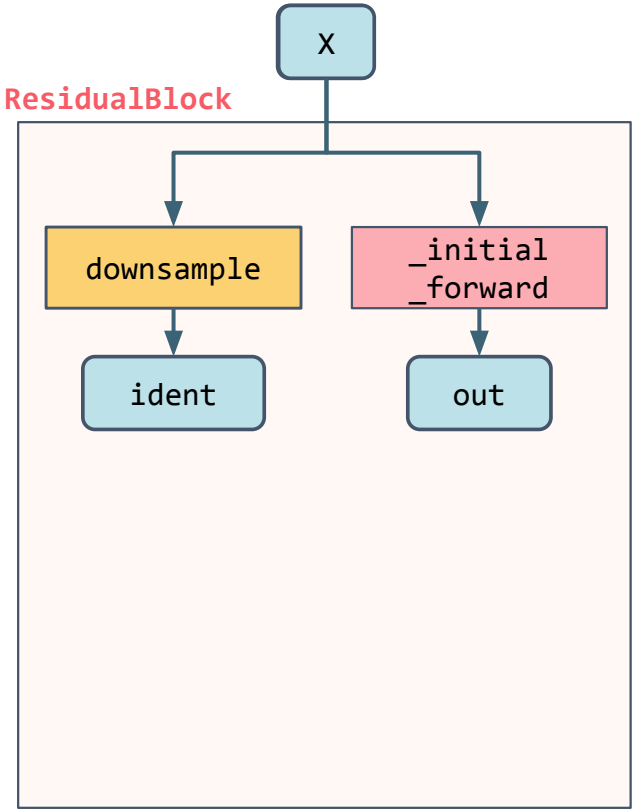
Residual Blocks

```
class ResidualBlock(nn.Module):
    ...
    def forward(self, x):
        identity = x

        # Initial forward pass
        out = self._initial_forward(x)

        # Apply downsampling if identity exists
        if self.downsample is not None:
            identity = self.downsample(x)

        # Residual connection
        out += identity

        # Final ReLU activation
        out = F.relu(out)
        return out
```



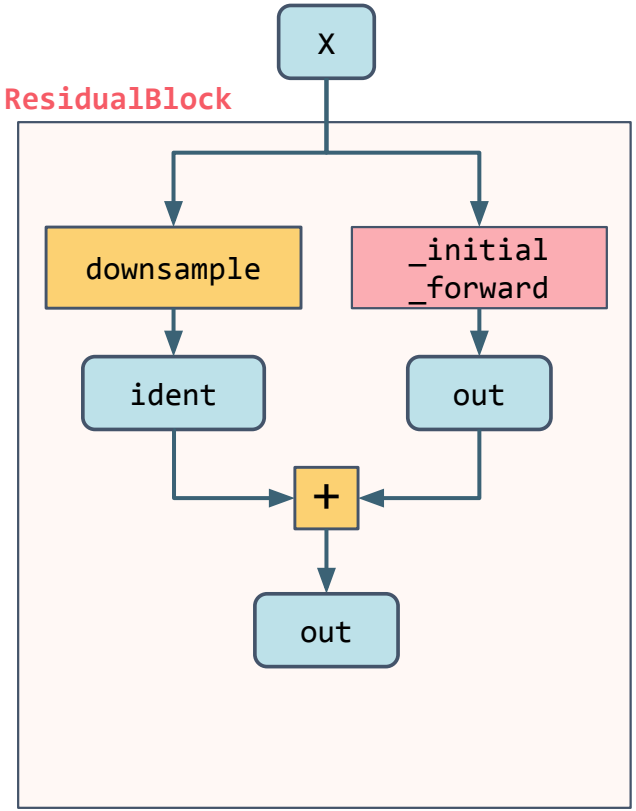
Residual Blocks

```
class ResidualBlock(nn.Module):
    ...
    def forward(self, x):
        identity = x

        # Initial forward pass
        out = self._initial_forward(x)

        # Apply downsampling if identity exists
        if self.downsample is not None:
            identity = self.downsample(x)

        # Residual connection
        out += identity

        # Final ReLU activation
        out = F.relu(out)
        return out
```



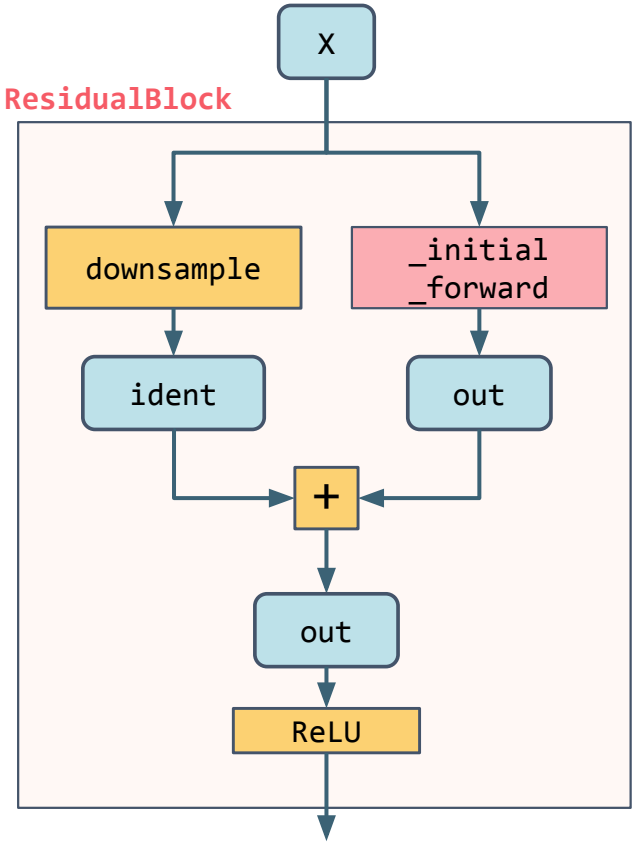
Residual Blocks

```
class ResidualBlock(nn.Module):
    ...
    def forward(self, x):
        identity = x

        # Initial forward pass
        out = self._initial_forward(x)

        # Apply downsampling if identity exists
        if self.downsample is not None:
            identity = self.downsample(x)

        # Residual connection
        out += identity

        # Final ReLU activation
        out = F.relu(out)
        return out
```



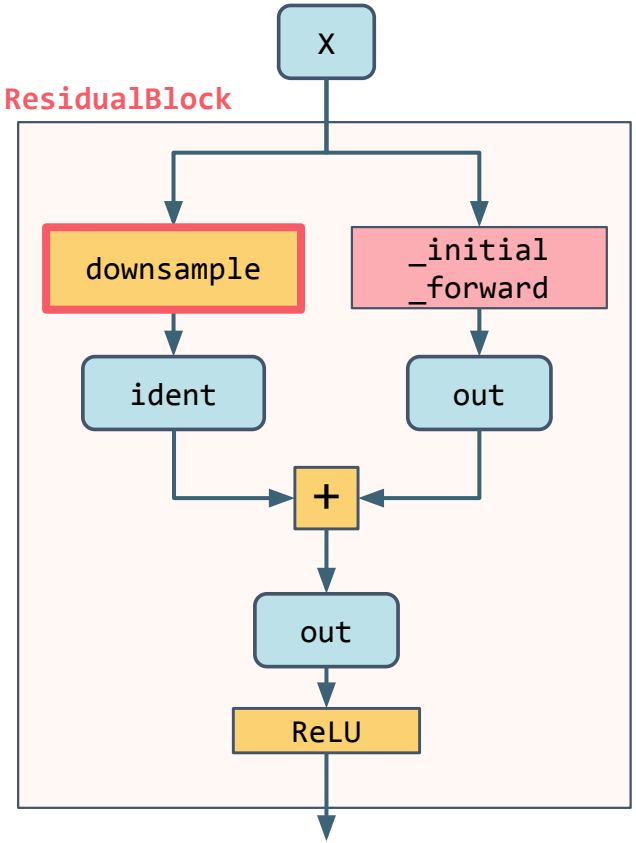
Residual Blocks

```
class ResidualBlock(nn.Module):
    ...
    def forward(self, x):
        identity = x

        # Initial forward pass
        out = self._initial_forward(x)

        # Apply downsampling if identity exists
        if self.downsample is not None:
            identity = self.downsample(x)

        # Residual connection
        out += identity

        # Final ReLU activation
        out = F.relu(out)
        return out
```

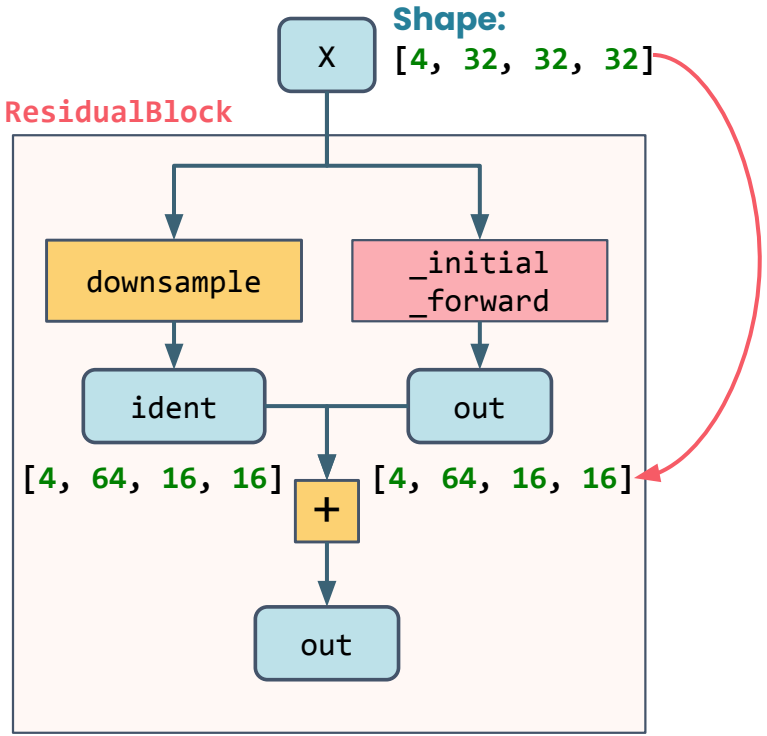


Residual Blocks

```
class ResidualBlock(nn.Module):
    ...
    def forward(self, x):
        identity = x

        # Initial forward pass
        out = self._initial_forward(x)

        # Apply downsampling if identity exists
        if self.downsample is not None:
            identity = self.downsample(x)

        # Residual connection
        out += identity

        # Final ReLU activation
        out = F.relu(out)
        return out
```



Downsampling



Non-Residual Block

Non-Residual Block

$$y = F(x)$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\boxed{\frac{\partial L}{\partial x}} = \boxed{\frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial b} \cdot \frac{\partial c}{\partial d} \cdot \cdot \frac{\partial y}{\partial x}}$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial b} \cdot \frac{\partial c}{\partial d} \cdot \cdot \frac{\partial F(x)}{\partial x}$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = \boxed{\frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial b} \cdot \frac{\partial c}{\partial d} \cdot \cdot} \frac{\partial F(x)}{\partial x}$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = C \cdot \frac{\partial F(x)}{\partial x}$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = \boxed{C \cdot \frac{\partial F(x)}{\partial x}}$$

Gradient of the loss with respect to x

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = C \cdot \frac{\partial F(x)}{\partial x}$$

Residual Block

$$y = F(x) + x$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = C \cdot \frac{\partial F(x)}{\partial x}$$

Residual Block

$$y = F(x) + x$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = C \cdot \frac{\partial F(x)}{\partial x}$$

Residual Block

$$y = F(x) + x$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

$$\frac{\partial L}{\partial x} = C \cdot \left(\frac{\partial F(\mathrm{x})}{\partial x} + 1 \right)$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = C \cdot \frac{\partial F(x)}{\partial x}$$

Residual Block

$$y = F(x) + x$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

$$\frac{\partial L}{\partial x} = C \cdot \frac{\partial F(\mathrm{x})}{\partial x} + C$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = \boxed{C \cdot \frac{\partial F(x)}{\partial x}}$$

Residual Block

$$y = F(x) + x$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

$$\frac{\partial L}{\partial x} = \boxed{C \cdot \frac{\partial F(x)}{\partial x}} + C$$

Non-Residual Block

$$y = F(x)$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x}$$

$$\frac{\partial L}{\partial x} = C \cdot \frac{\partial F(x)}{\partial x}$$

Residual Block

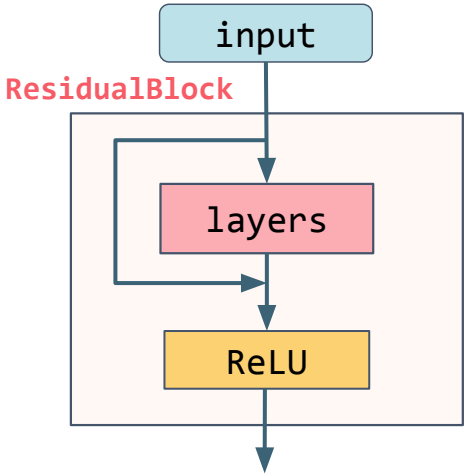
$$y = F(x) + x$$

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

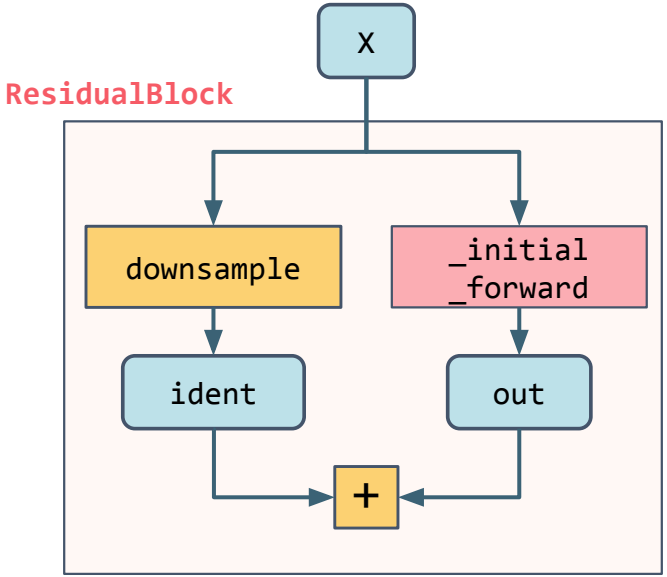
$$\frac{\partial L}{\partial x} = C \cdot \frac{\partial F(x)}{\partial x} + C$$

Preserves the gradient

ResNet



Forward Pass



Original input + Correction

What needs to change about the input?

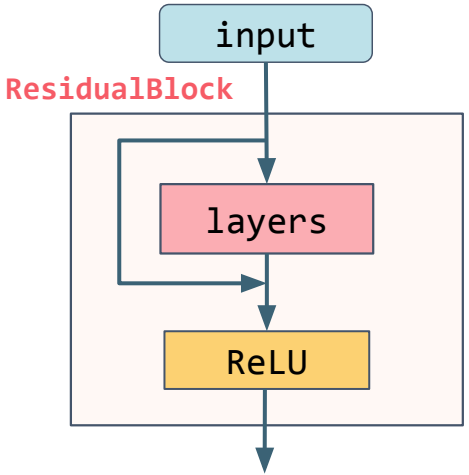
Before ResNet

20+ Layers

- Vanishing gradients
- Poor performance

ResNet

100+ Layers

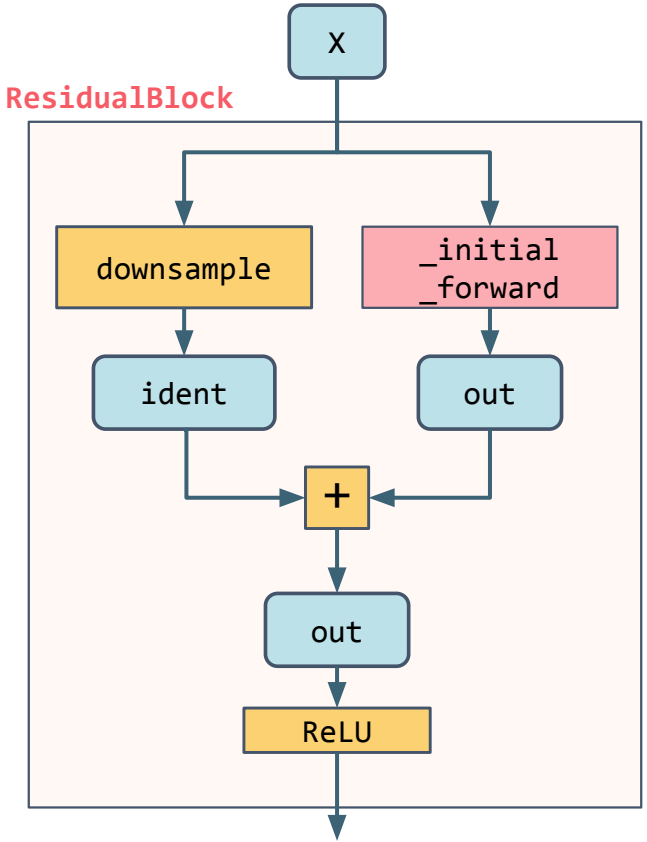


DeepLearning.AI

DenseNet

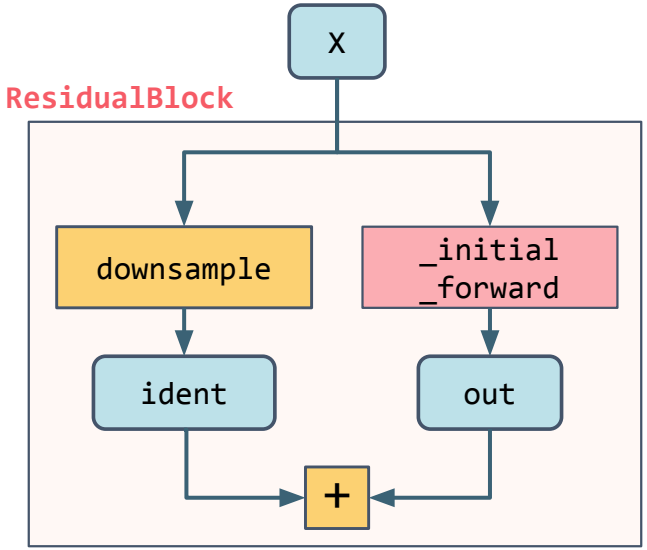
Designing Custom Architectures

ResNet



$$C \cdot \frac{\partial F(x)}{\partial x} + C$$

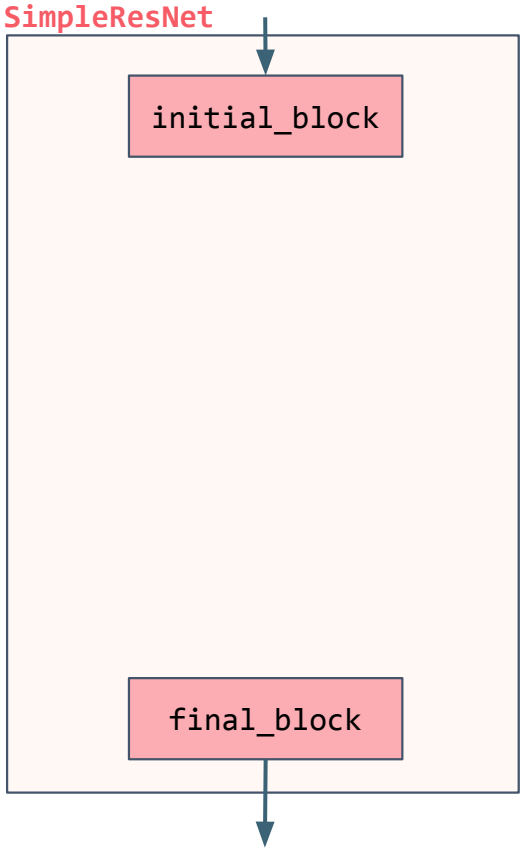
ResNet



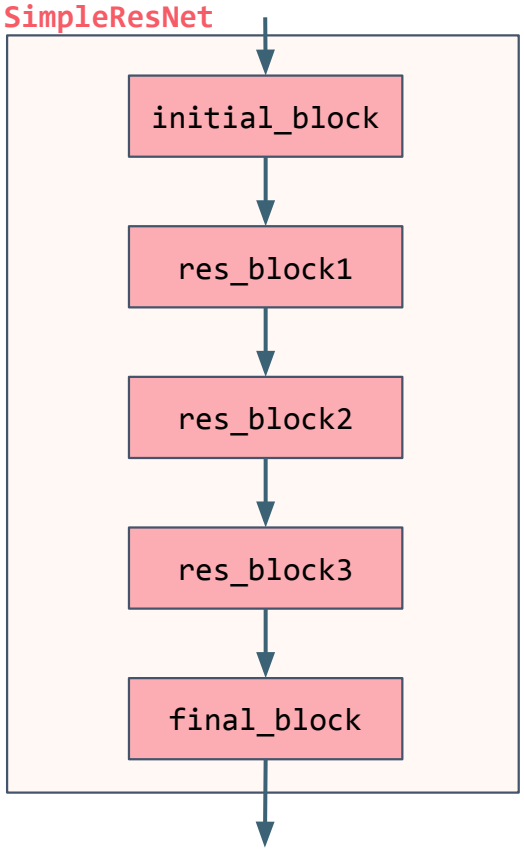
Original input + Correction

Reshapes the original input

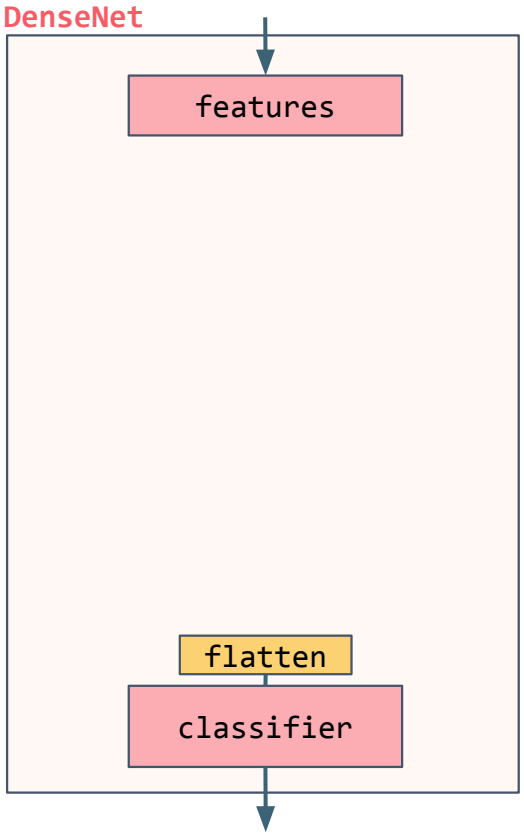
ResNet



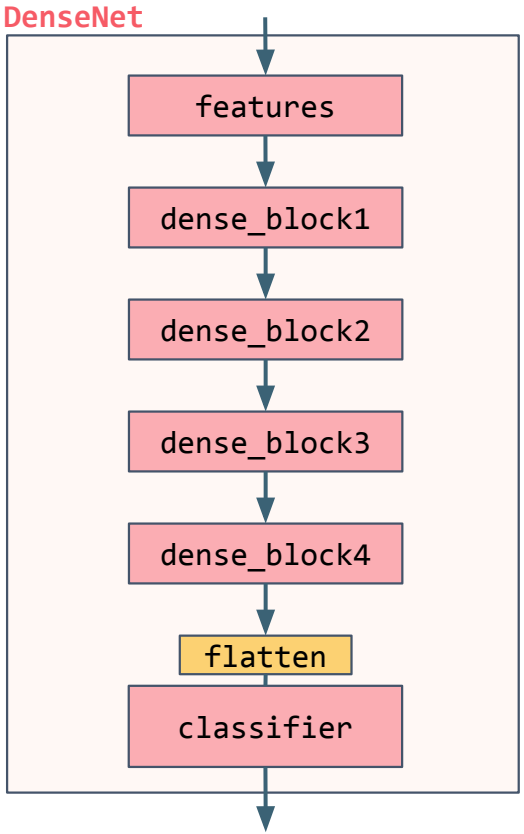
ResNet


SimpleResNet
initial_block
res_block1
res_block2
res_block3
final_block

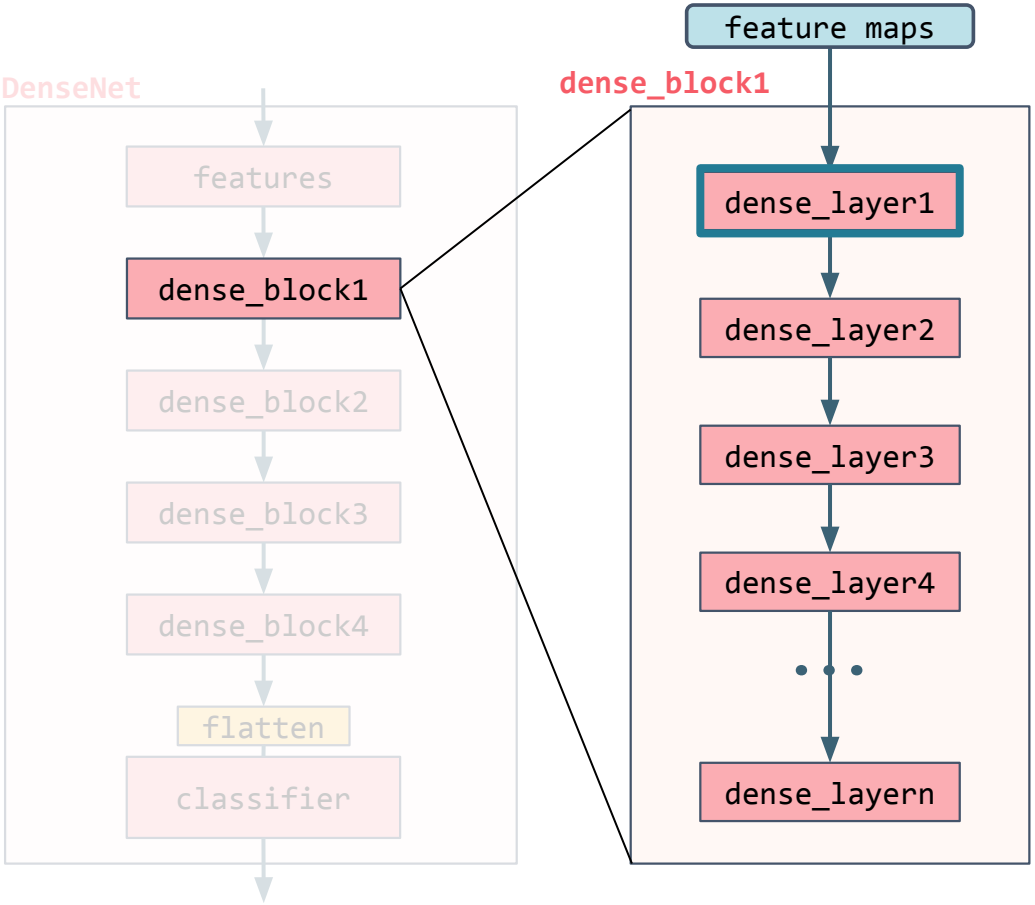
DenseNet



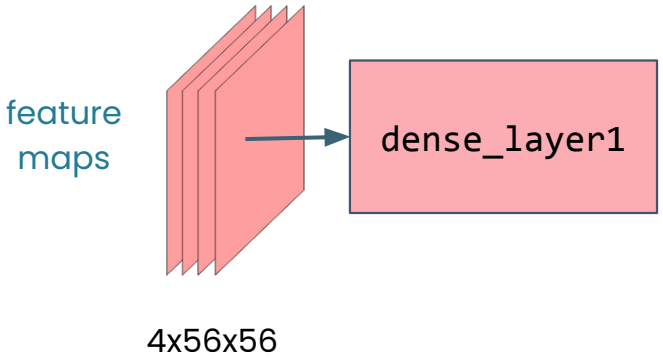
DenseNet


DenseNet
features
dense_block1
dense_block2
dense_block3
dense_block4
flatten
classifier

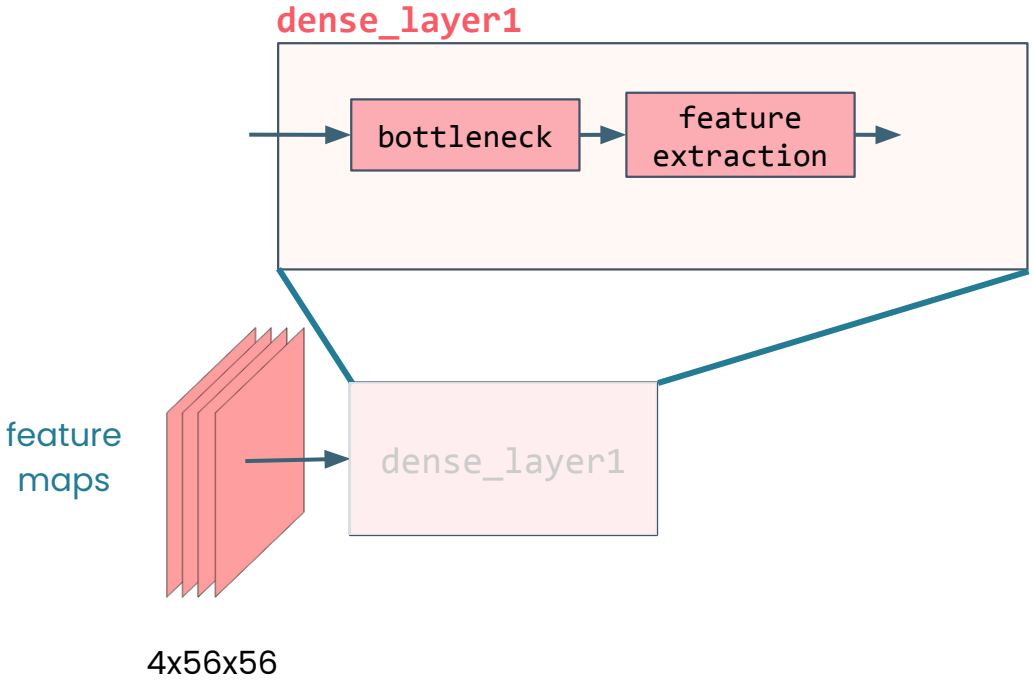
DenseNet


DenseNet
features
dense_block1
dense_block2
dense_block3
dense_block4
flatten
classifier
feature maps
dense_block1
dense_layer1
dense_layer2
dense_layer3
dense_layer4
dense_layern

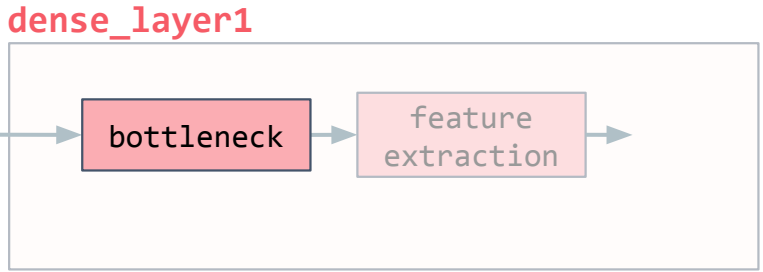
Dense Layers

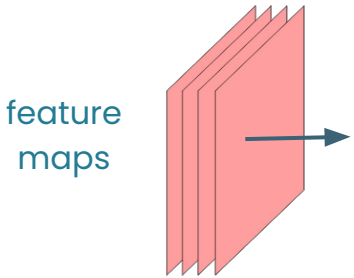


Dense Layers

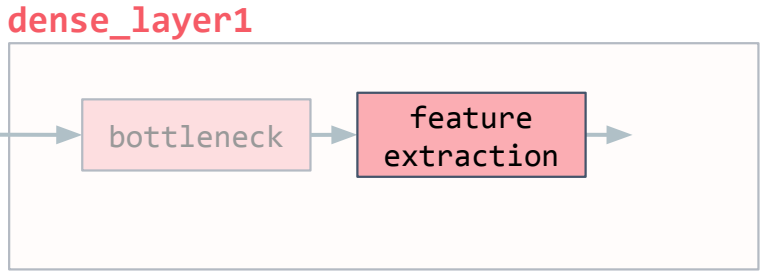


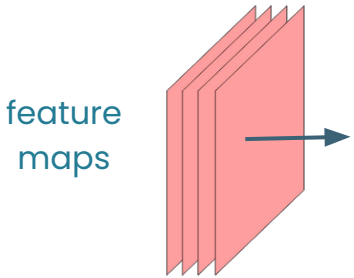
Dense Layers



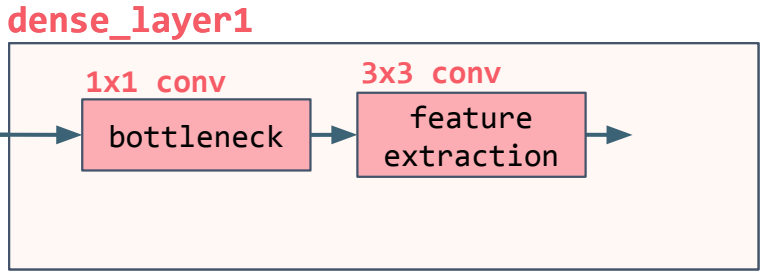


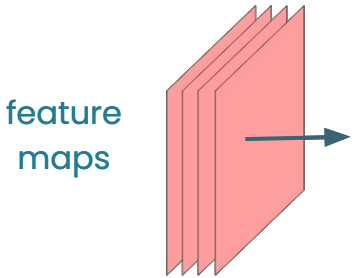
Dense Layers



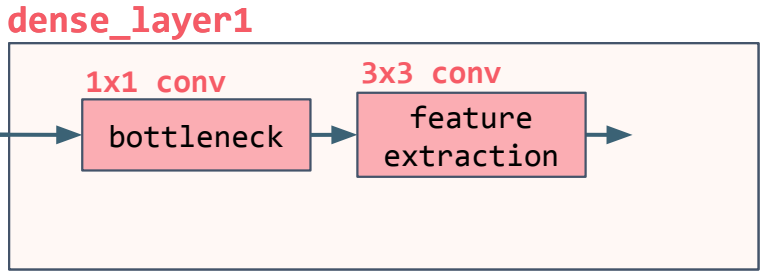


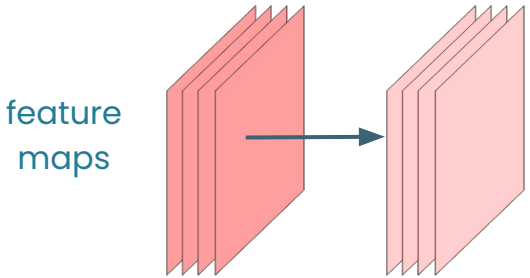
Dense Layers





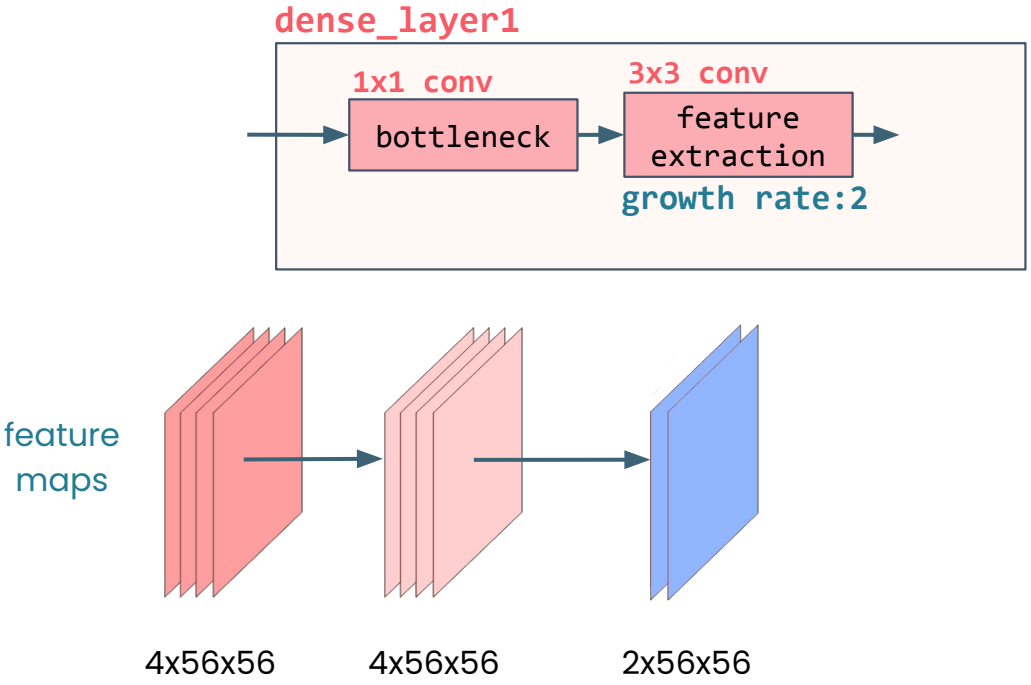
Dense Layers



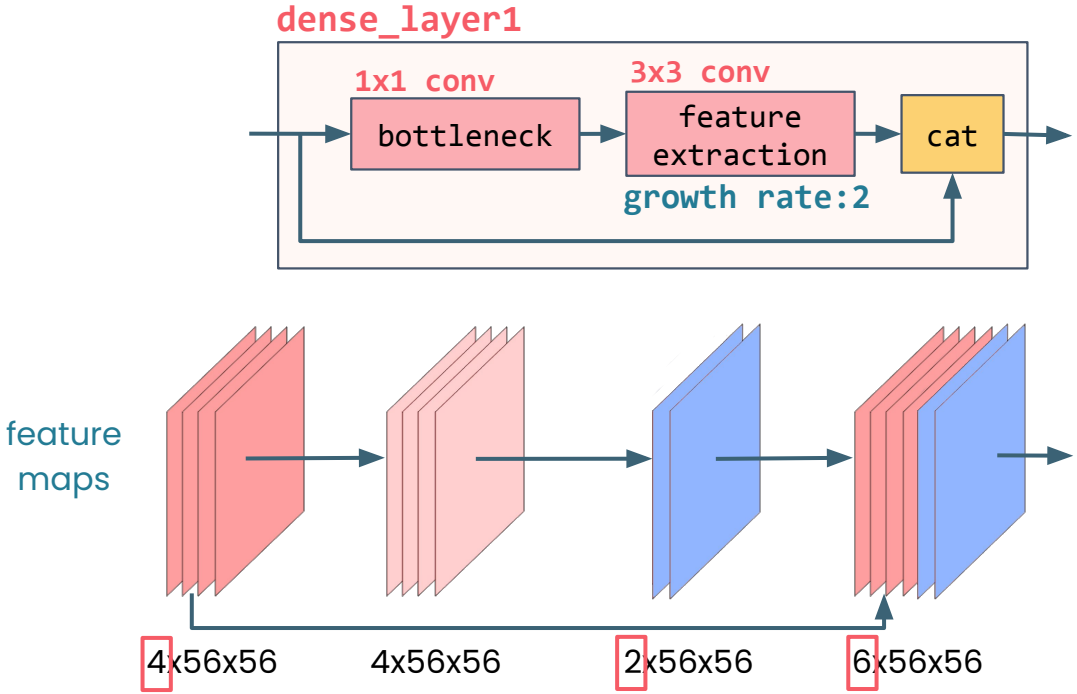


Reduce channels to 4

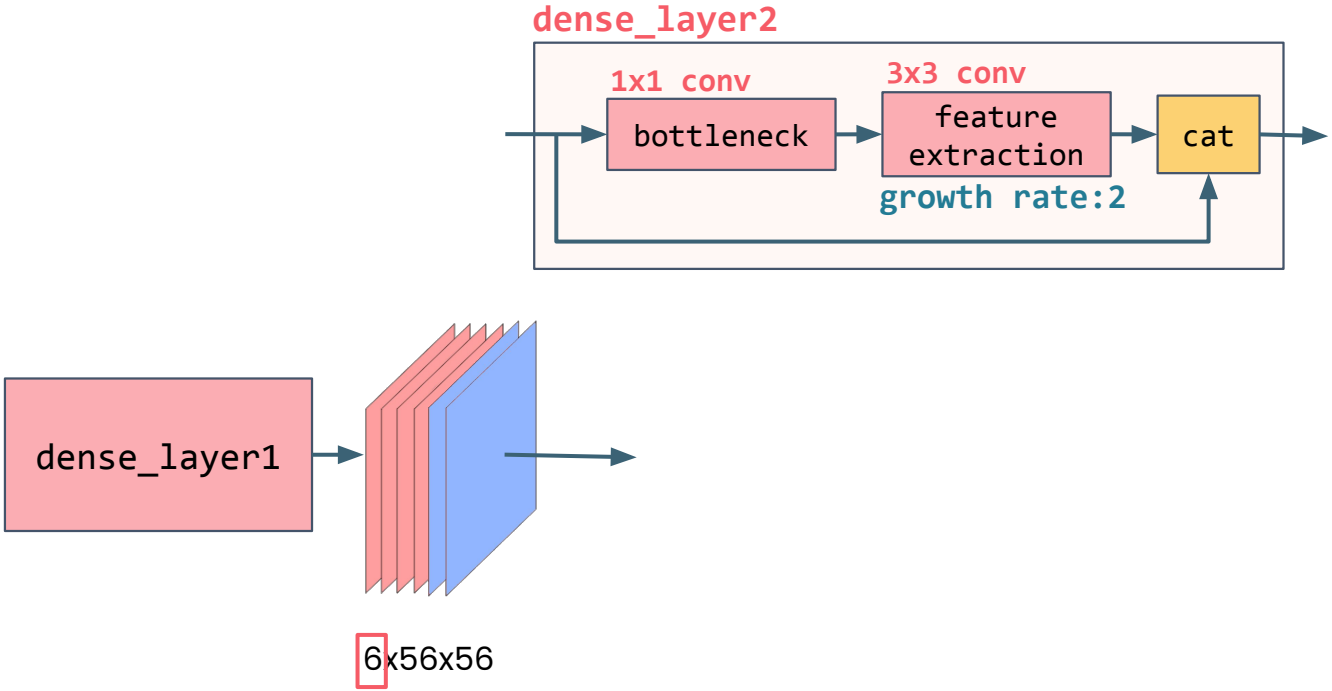
Dense Layers



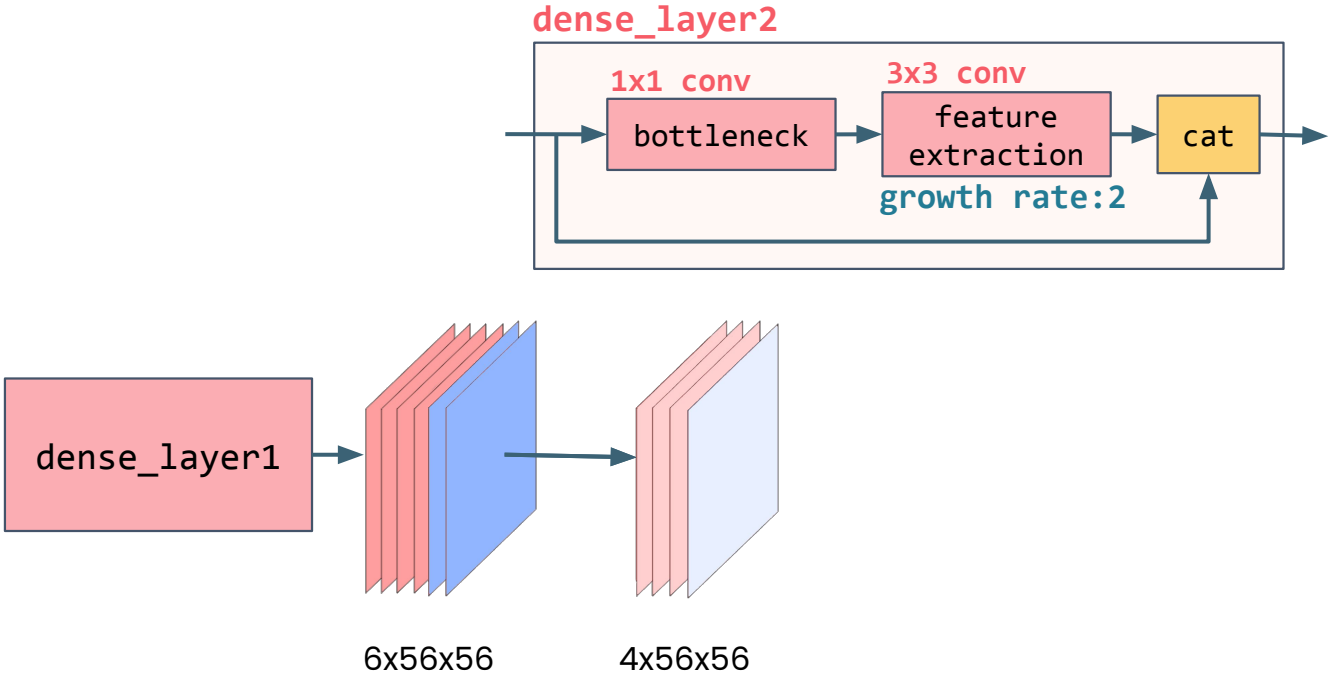
Dense Layers



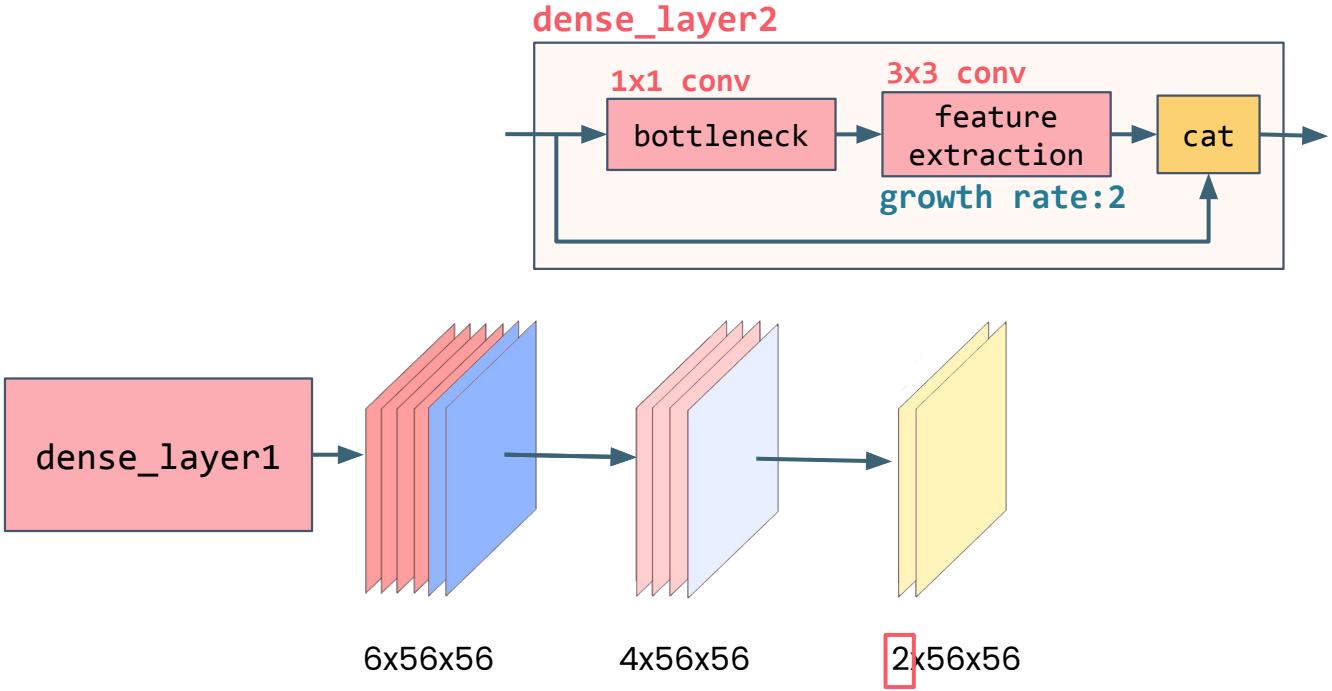
Dense Layers



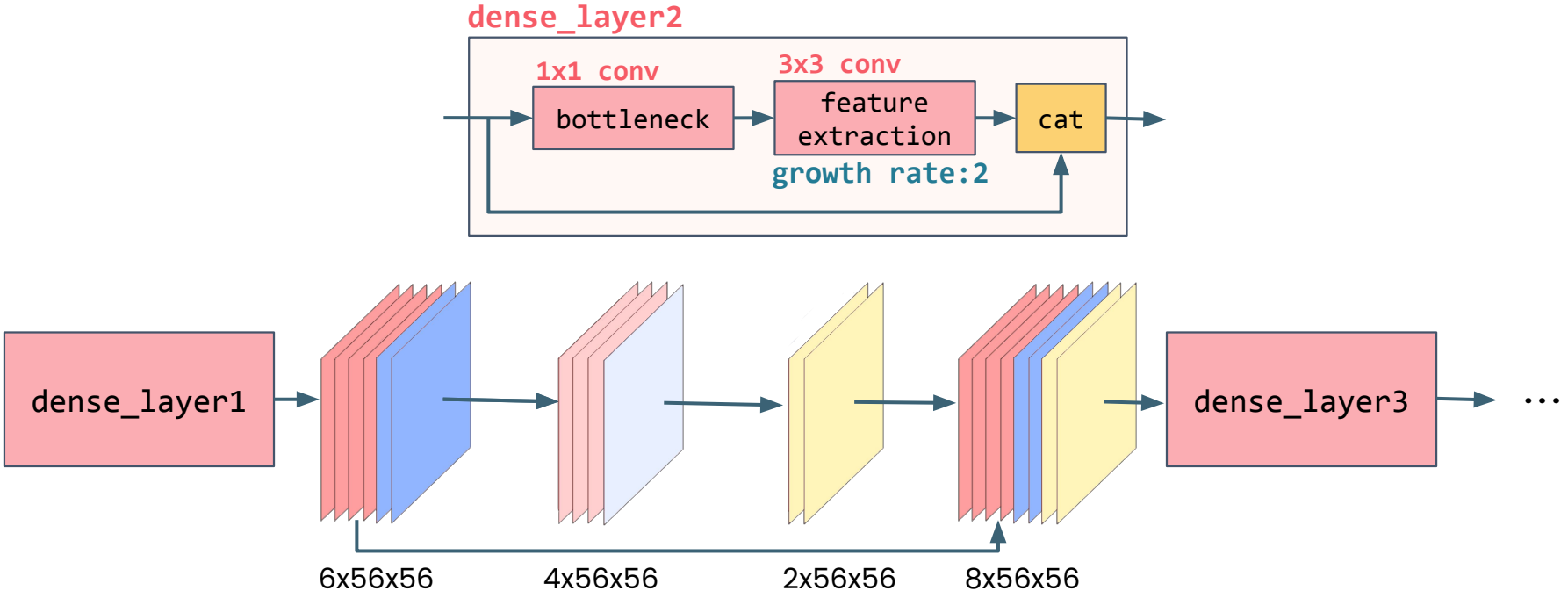
Dense Layers



Dense Layers



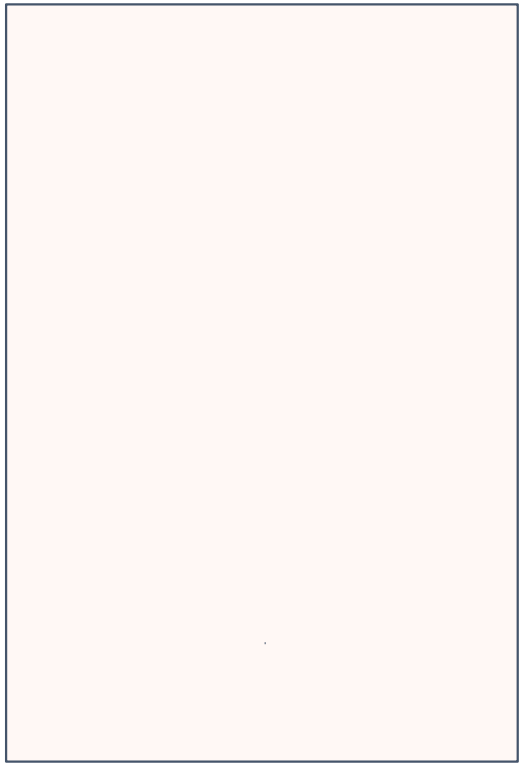
Dense Layers



DenseNet

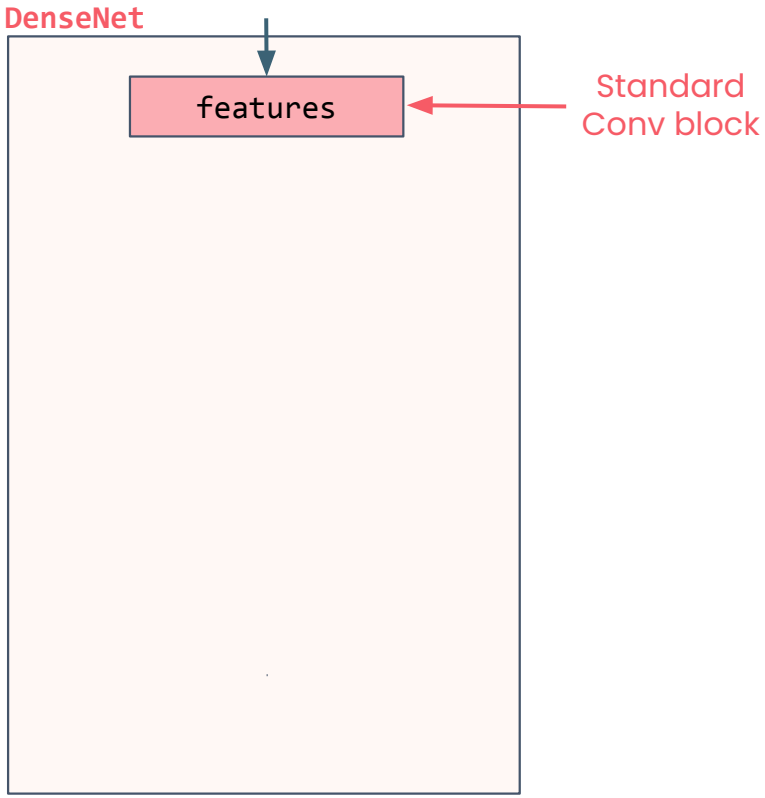
```
class DenseNet(nn.Module):
    . . .

    def forward(self, x):
        # Initial features
        x = self.features(x)

        # Dense blocks
        for block in self.dense_blocks:
            x = block(x)

        # Flatten the output
        x = torch.flatten(x, 1)

        # Classifier
        x = self.classifier(x)

        return x
```

DenseNet



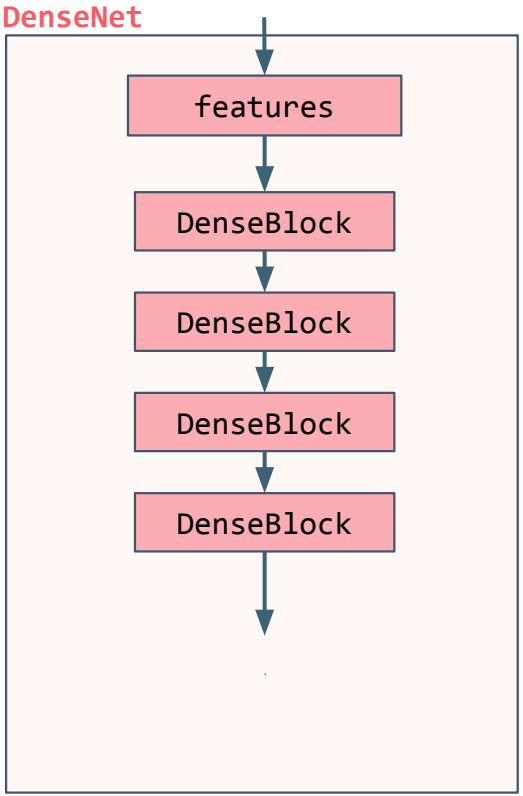
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def forward(self, x):
        # Initial features
        x = self.features(x)

        # Dense blocks
        for block in self.dense_blocks:
            x = block(x)

        # Flatten the output
        x = torch.flatten(x, 1)

        # Classifier
        x = self.classifier(x)

        return x
```



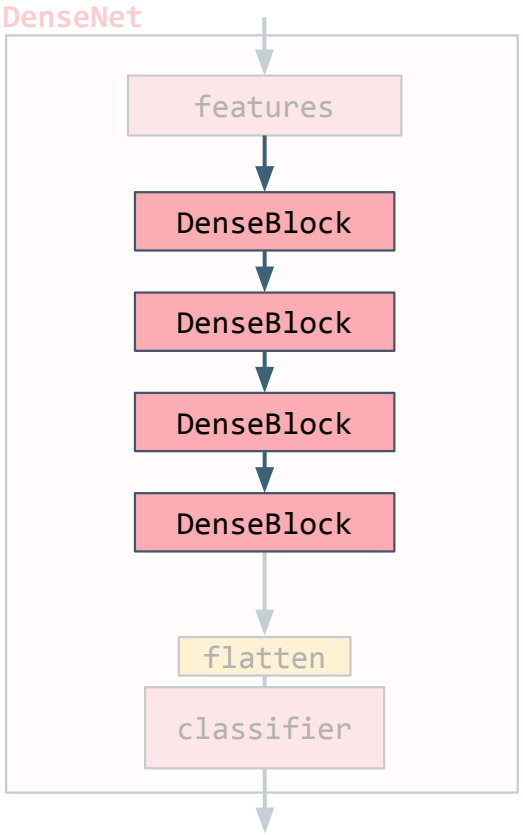
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def forward(self, x):
        # Initial features
        x = self.features(x)

        # Dense blocks
        for block in self.dense_blocks:
            x = block(x)

        # Flatten the output
        x = torch.flatten(x, 1)

        # Classifier
        x = self.classifier(x)

        return x
```



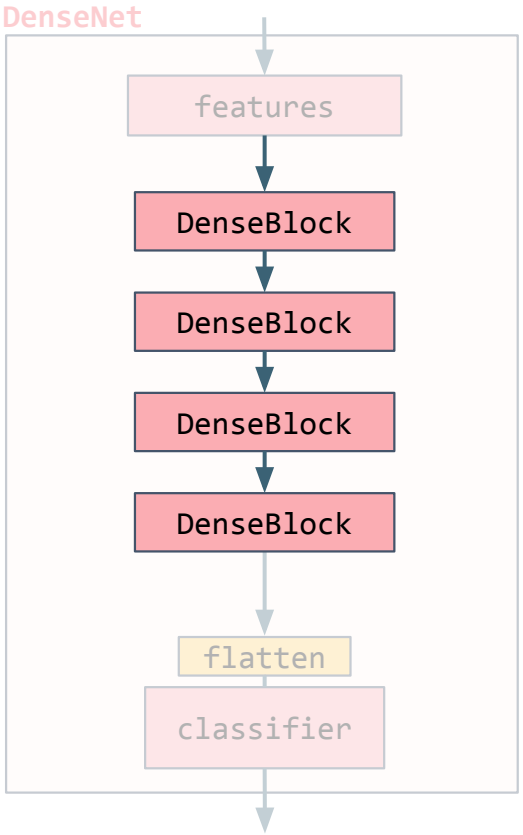
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def forward(self, x):
        # Initial features
        x = self.features(x)

        # Dense blocks
        for block in self.dense_blocks:
            x = block(x)

        # Flatten the output
        x = torch.flatten(x, 1)

        # Classifier
        x = self.classifier(x)

        return x
```



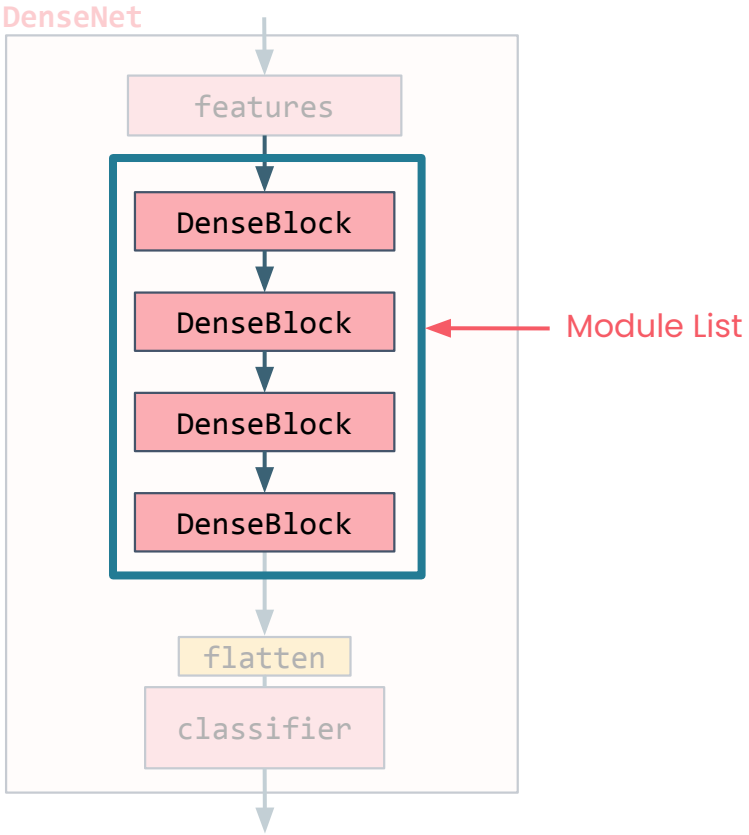
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def _get_dense_blocks(self, . . .):
```



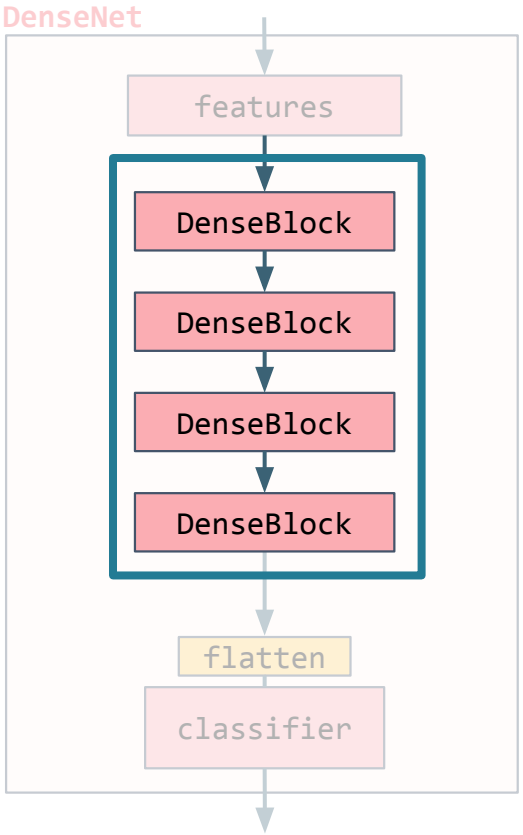
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def _get_dense_blocks(self, . . .):

        dense_blocks = nn.ModuleList()
```



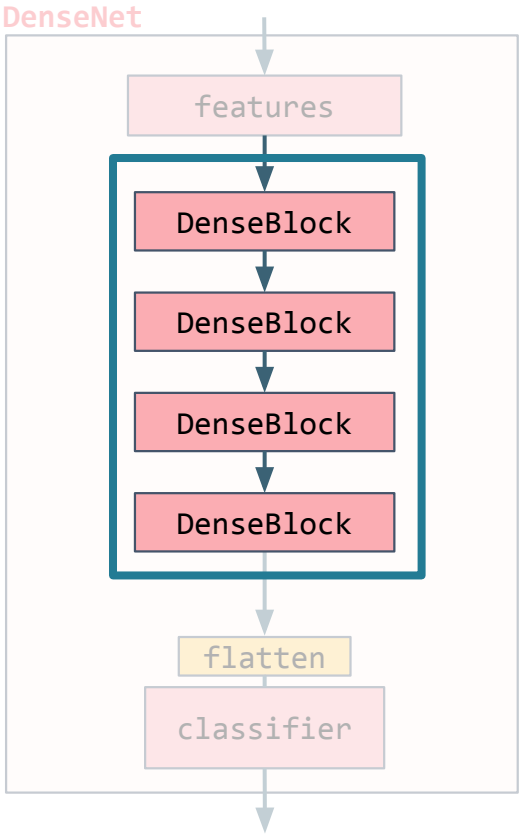
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def _get_dense_blocks(self, . . .):
        . . .

        for i, num_layers in enumerate(block_config):
            # Create a DenseBlock
            db = DenseBlock(
                num_layers=num_layers,
                in_channels=num_features,
                growth_rate=growth_rate,
                bn_size=bn_size,
            )
```



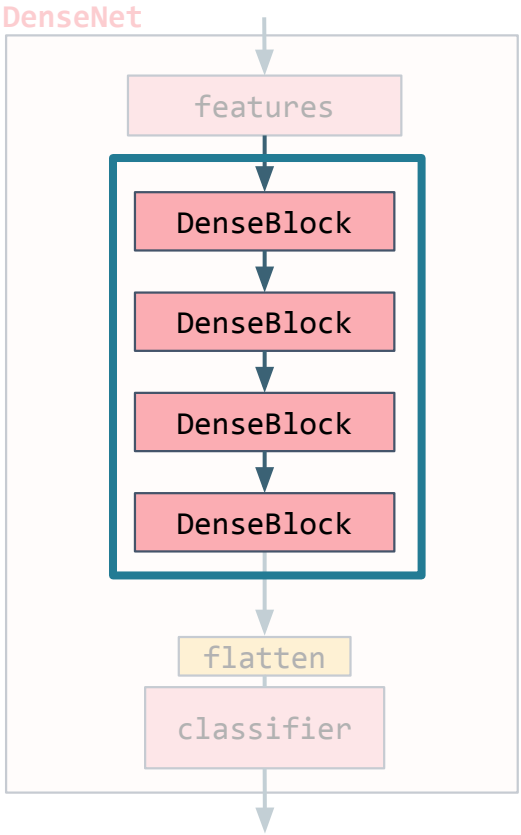
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def _get_dense_blocks(self, . . .):
        . . .

        for i, num_layers in enumerate(block_config):
            # Create a DenseBlock
            db = DenseBlock(
                num_layers=num_layers,
                in_channels=num_features,
                growth_rate=growth_rate,
                bn_size=bn_size,
            )
```



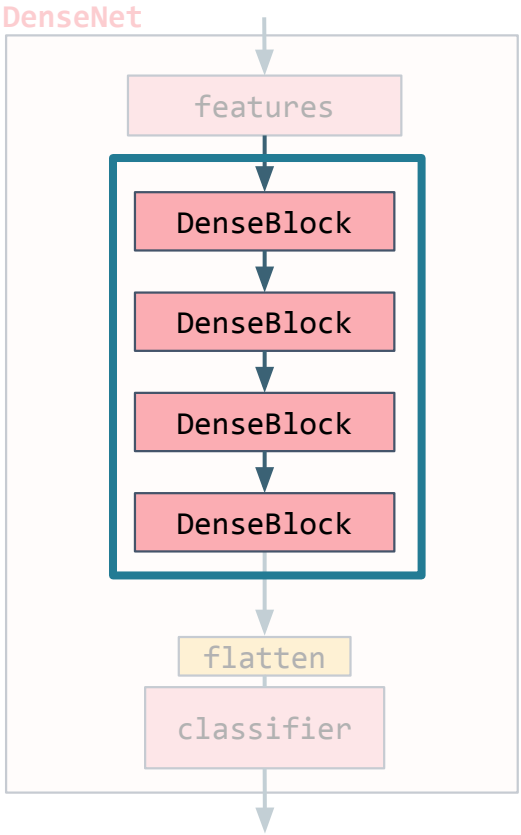
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def _get_dense_blocks(self, . . .):
        . . .

        for i, num_layers in enumerate(block_config):
            # Create a DenseBlock
            db = DenseBlock(
                num_layers=num_layers,
                in_channels=num_features,
                growth_rate=growth_rate,
                bn_size=bn_size,
            )

            dense_blocks.append(db)
```



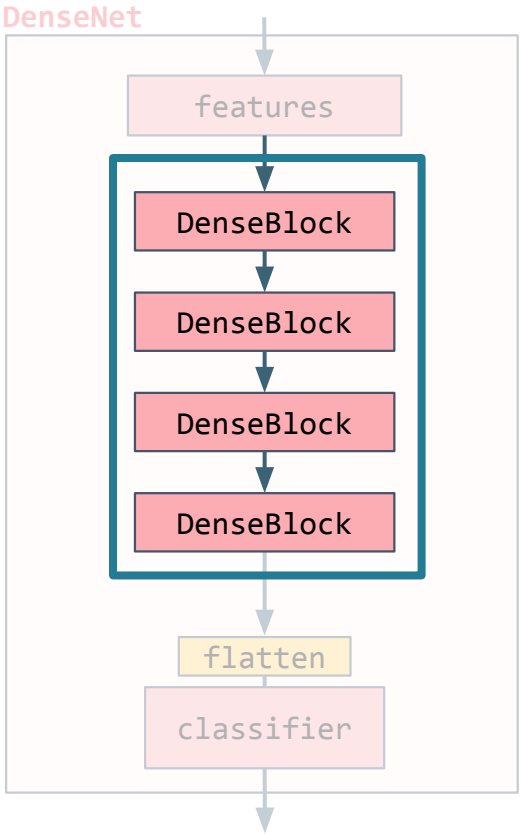
DenseNet

```
class DenseNet(nn.Module):
    . . .

    def _get_dense_blocks(self, . . .):
        . . .

        if i != len(block_config) - 1:
            transition = TransitionLayer(
                in_channels=num_features,
                compression_factor=compression_factor)
```

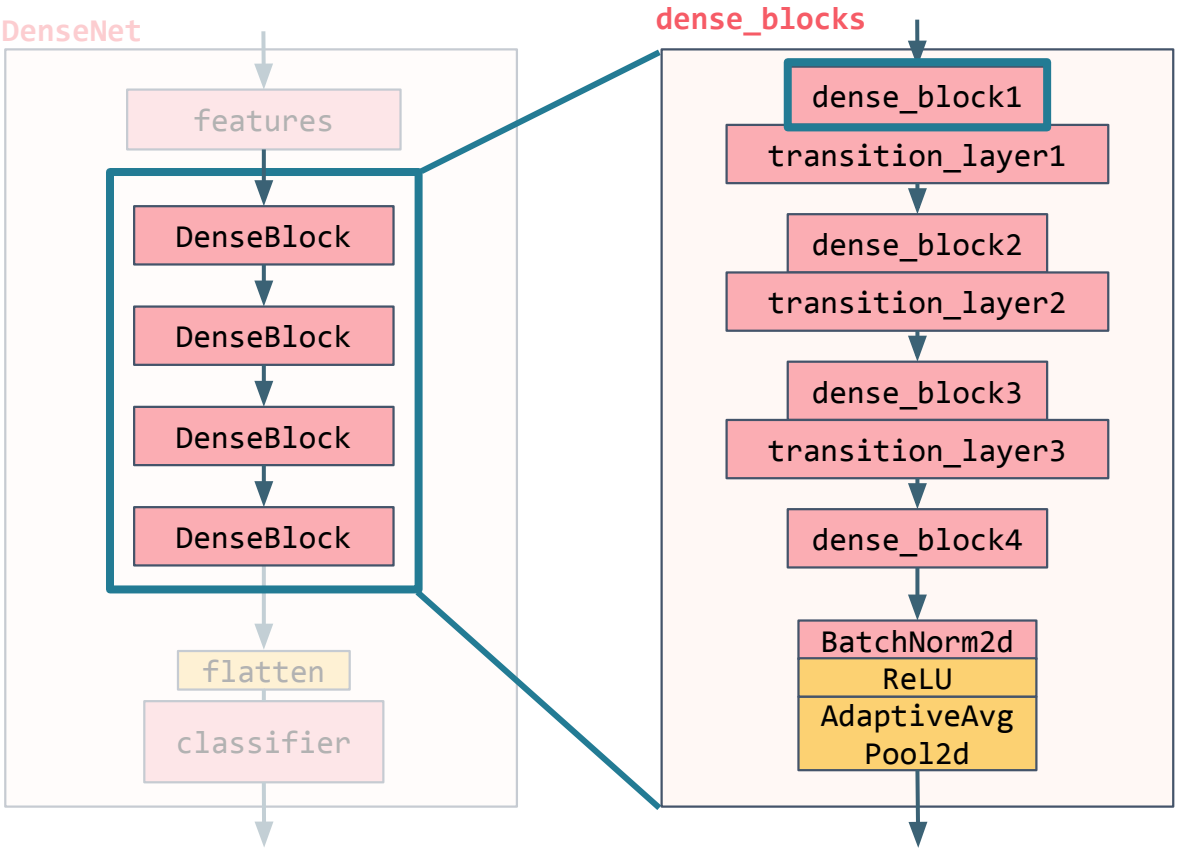


DenseNet

```
class DenseNet(nn.Module):
    . . .

    def _get_dense_blocks(self, . . .):
        . . .

        if i != len(block_config) - 1:
            transition = TransitionLayer(
                in_channels=num_features,
                compression_factor=compression_factor)

            dense_blocks.append(transition)
```



DenseNet

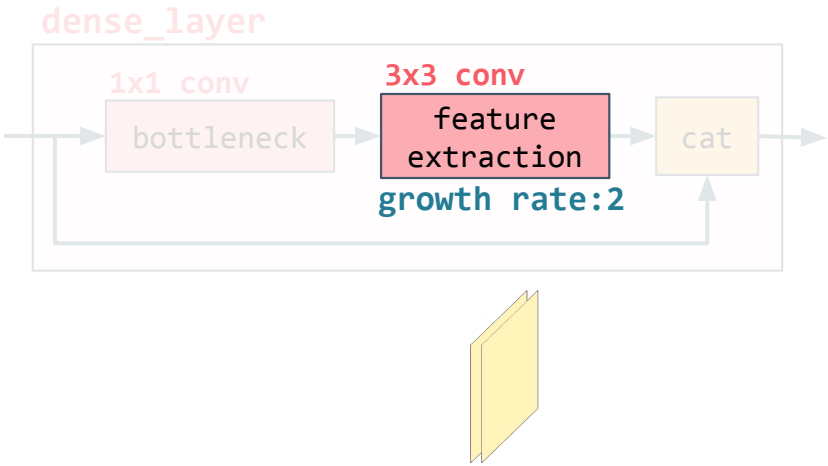

DenseNet
features
DenseBlock
DenseBlock
DenseBlock
DenseBlock
flatten
classifier
dense_blocks
dense_block1
transition_layer1
dense_block2
transition_layer2
dense_block3
transition_layer3
dense_block4
BatchNorm2d
ReLU
AdaptiveAvg
Pool2d

DenseBlock

```
class DenseBlock(nn.Module):
    def __init__(self, self, num_layers, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseBlock, self).__init__()

        self.layers = nn.ModuleList()

        for i in range(num_layers):
            dense_layer = DenseLayer(
                in_channels + i * growth_rate,
                growth_rate, bn_size)
            self.layers.append(dense_layer)

    def forward(self, x):
        features = x
        for dense_layer in self.layers:
            features = dense_layer(features)

        return features
```

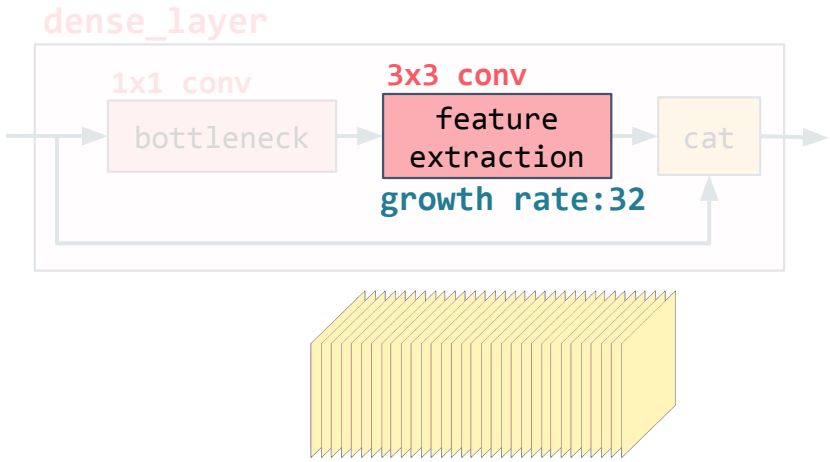
DenseBlock

```
class DenseBlock(nn.Module):
    def __init__(self, self, num_layers, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseBlock, self).__init__()

        self.layers = nn.ModuleList()

        for i in range(num_layers):
            dense_layer = DenseLayer(
                in_channels + i * growth_rate,
                growth_rate, bn_size)
            self.layers.append(dense_layer)

    def forward(self, x):
        features = x
        for dense_layer in self.layers:
            features = dense_layer(features)

        return features
```



DenseBlock

```
class DenseBlock(nn.Module):
    def __init__(self, self, num_layers, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseBlock, self).__init__()

        self.layers = nn.ModuleList()

        for i in range(num_layers):
            dense_layer = DenseLayer(
                in_channels + i * growth_rate,
                growth_rate, bn_size)
            self.layers.append(dense_layer)

    def forward(self, x):
        features = x
        for dense_layer in self.layers:
            features = dense_layer(features)

        return features
```



DenseBlock

```
class DenseBlock(nn.Module):
    def __init__(self, self, num_layers, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseBlock, self).__init__()

        self.layers = nn.ModuleList()

        for i in range(num_layers):
            dense_layer = DenseLayer(
                in_channels + i * growth_rate,
                growth_rate, bn_size)
            self.layers.append(dense_layer)

    def forward(self, x):
        features = x
        for dense_layer in self.layers:
            features = dense_layer(features)

        return features
```

DenseBlock

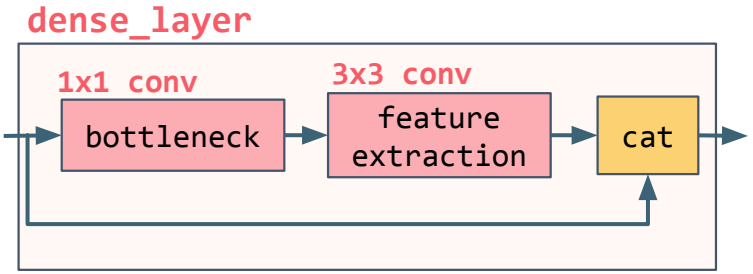
```
class DenseBlock(nn.Module):
    def __init__(self, self, num_layers, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseBlock, self).__init__()

        self.layers = nn.ModuleList()

        for i in range(num_layers):
            dense_layer = DenseLayer(
                in_channels + i * growth_rate,
                growth_rate, bn_size)
            self.layers.append(dense_layer)

    def forward(self, x):
        features = x
        for dense_layer in self.layers:
            features = dense_layer(features)

        return features
```

DenseBlock

```
class DenseBlock(nn.Module):
    def __init__(self, self, num_layers, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseBlock, self).__init__()

        self.layers = nn.ModuleList()

        for i in range(num_layers):
            dense_layer = DenseLayer(
                in_channels + i * growth_rate,
                growth_rate, bn_size)
            self.layers.append(dense_layer)

    def forward(self, x):
        features = x
        for dense_layer in self.layers:
            features = dense_layer(features)

        return features
```

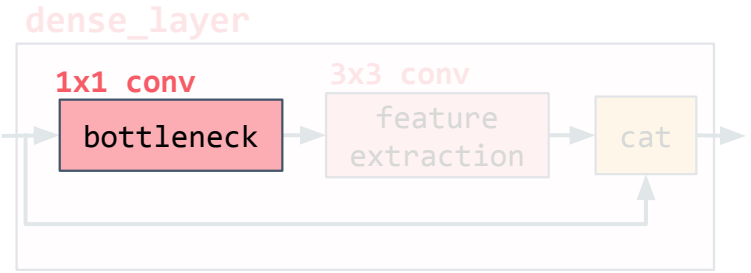
DenseLayer

```
class DenseLayer(nn.Module):
    def __init__(self, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseLayer, self).__init__()

        # Bottleneck
        self.dimension_reduction = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, bn_size * growth_rate,
                      kernel_size=1, stride=1, bias=False))

        # Feature extraction
        self.feature_extraction = nn.Sequential(
            nn.BatchNorm2d(bn_size * growth_rate),
            nn.ReLU(inplace=True),
            nn.Conv2d(bn_size * growth_rate, growth_rate,
                      kernel_size=3, stride=1, padding=1,
                      bias=False,))
```



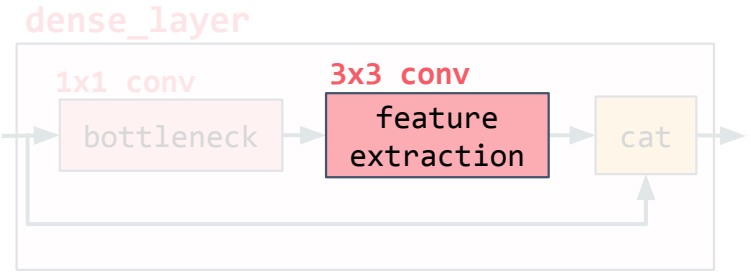
DenseLayer

```
class DenseLayer(nn.Module):
    def __init__(self, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseLayer, self).__init__()

        # Bottleneck
        self.dimension_reduction = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, bn_size * growth_rate,
                      kernel_size=1, stride=1, bias=False))

        # Feature extraction
        self.feature_extraction = nn.Sequential(
            nn.BatchNorm2d(bn_size * growth_rate),
            nn.ReLU(inplace=True),
            nn.Conv2d(bn_size * growth_rate, growth_rate,
                      kernel_size=3, stride=1, padding=1,
                      bias=False,))
```



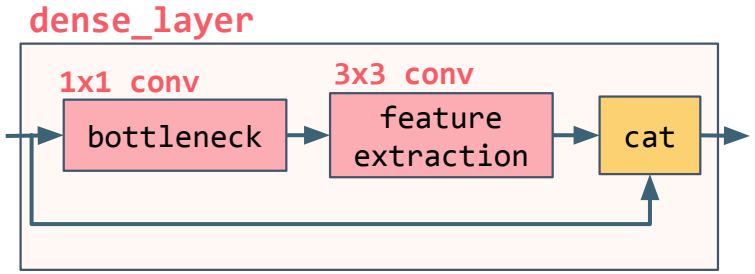
DenseLayer

```
class DenseLayer(nn.Module):
    def __init__(self, in_channels,
                 growth_rate=32, bn_size=4):
        super(DenseLayer, self).__init__()

        # Bottleneck
        self.dimension_reduction = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, bn_size * growth_rate,
                      kernel_size=1, stride=1, bias=False))

        # Feature extraction
        self.feature_extraction = nn.Sequential(
            nn.BatchNorm2d(bn_size * growth_rate),
            nn.ReLU(inplace=True),
            nn.Conv2d(bn_size * growth_rate, growth_rate,
                      kernel_size=3, stride=1, padding=1,
                      bias=False,))
```



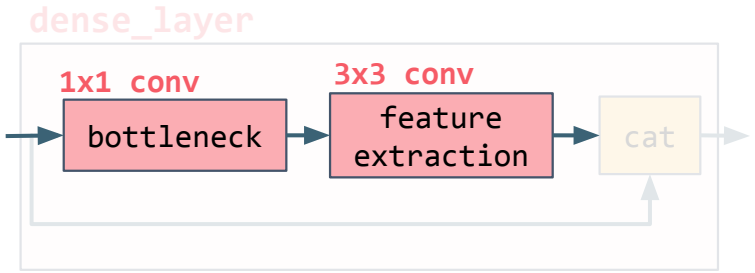
DenseLayer

```
class DenseLayer(nn.Module):
    def __init__(self, in_channels,
                 growth_rate=32, bn_size=4):
        . . .

    def forward(self, x):
        out = self.dimension_reduction(x)
        out = self.feature_extraction(out)

        # Concatenate
        out = torch.cat((x, out), 1)
        return out
```



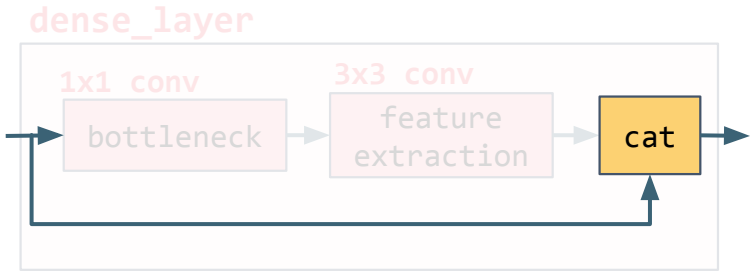
DenseLayer

```
class DenseLayer(nn.Module):
    def __init__(self, in_channels,
                 growth_rate=32, bn_size=4):
        . . .

    def forward(self, x):
        out = self.dimension_reduction(x)
        out = self.feature_extraction(out)

        # Concatenate
        out = torch.cat((x, out), 1)
        return out
```



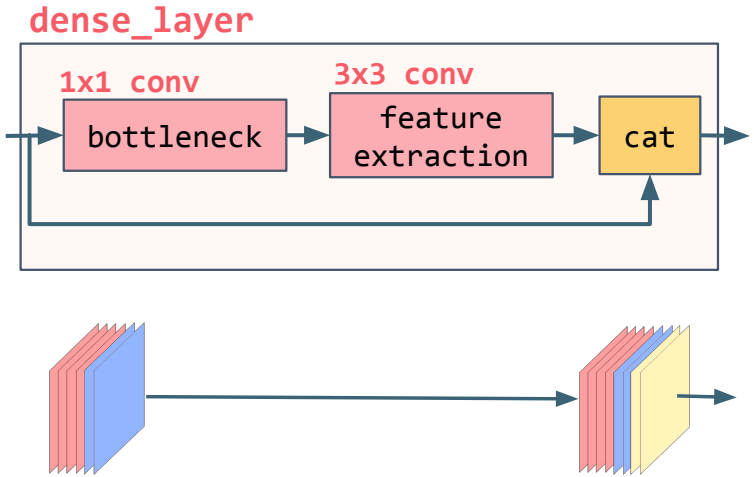
DenseLayer

```
class DenseLayer(nn.Module):
    def __init__(self, in_channels,
                 growth_rate=32, bn_size=4):
        . . .

    def forward(self, x):
        out = self.dimension_reduction(x)
        out = self.feature_extraction(out)

        # Concatenate
        out = torch.cat((x, out), 1)
        return out
```

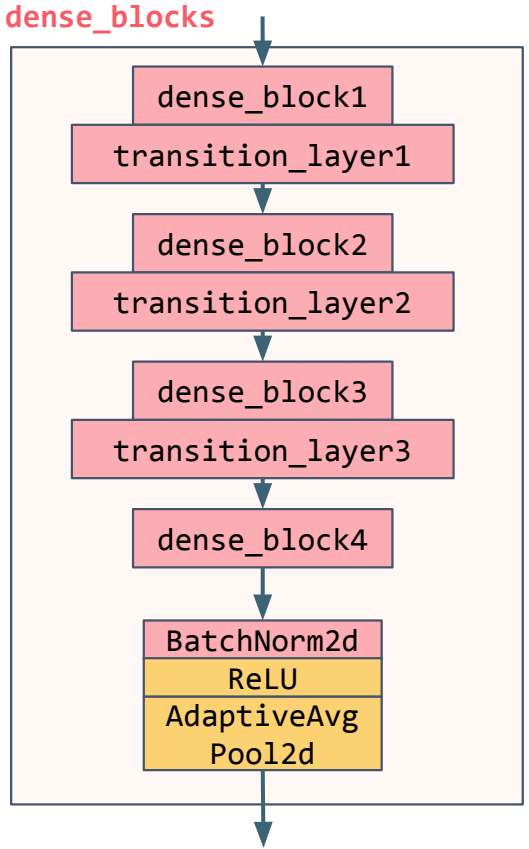


DenseLayer

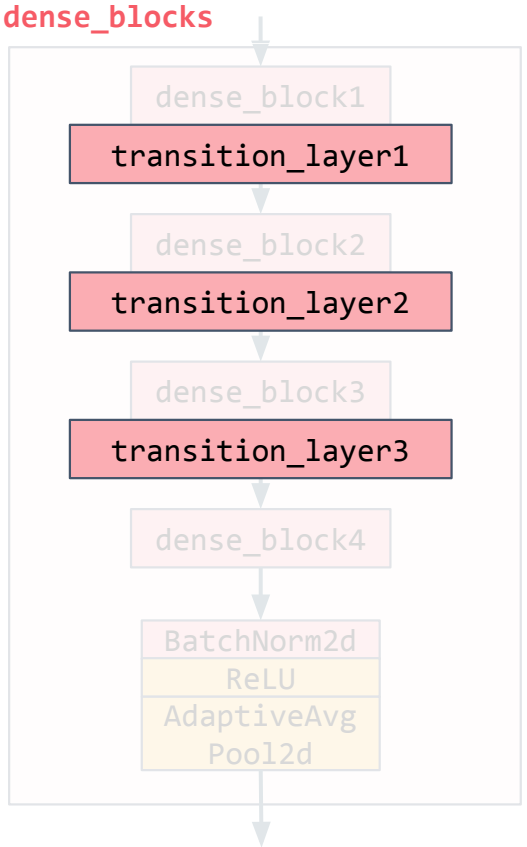
```
class DenseLayer(nn.Module):
    def __init__(self, in_channels,
                 growth_rate=32, bn_size=4):
        . . .

    def forward(self, x):
        out = self.dimension_reduction(x)
        out = self.feature_extraction(out)

        # Concatenate
        out = torch.cat((x, out), 1)
        return out
```



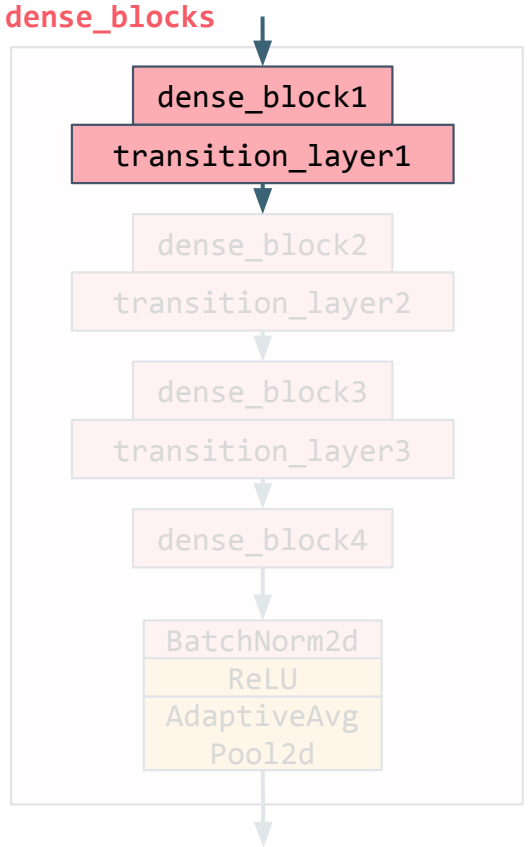
Transition Layers



Transition Layers

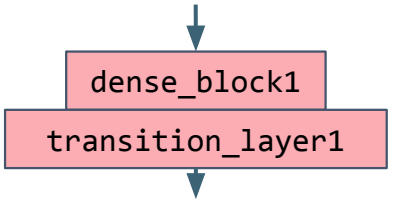


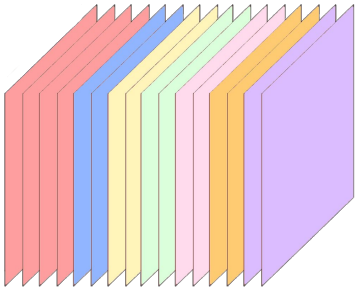
Transition Layers



Transition Layers

```
class TransitionLayer(nn.Module):
    def __init__(self, in_channels, compression_factor=0.5):
        super(TransitionLayer, self).__init__()

        out_channels = int(in_channels * compression_factor)

        self.transition = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, out_channels,
                      kernel_size=1, stride=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2))

    def forward(self, x):
        return self.transition(x)
```

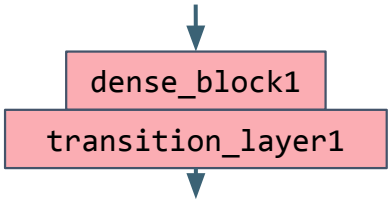


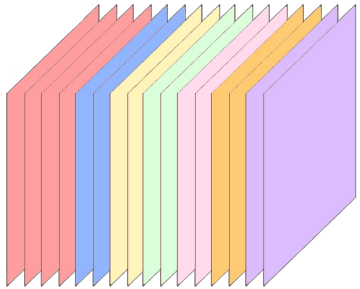


16x56x56

Transition Layers

```
class TransitionLayer(nn.Module):
    def __init__(self, in_channels, compression_factor=0.5):
        super(TransitionLayer, self).__init__()

        out_channels = int(in_channels * compression_factor)

        self.transition = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, out_channels,
                      kernel_size=1, stride=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2))

    def forward(self, x):
        return self.transition(x)
```

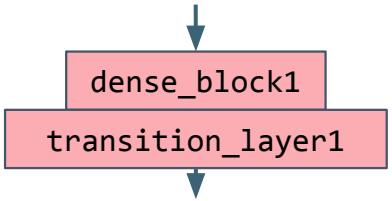


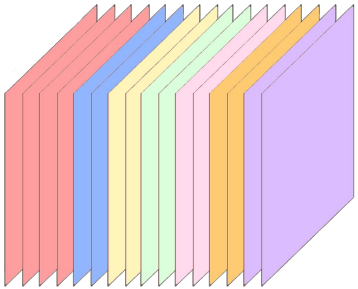


16x56x56

Transition Layers

```
class TransitionLayer(nn.Module):
    def __init__(self, in_channels, compression_factor=0.5):
        super(TransitionLayer, self).__init__()

        out_channels = int(in_channels * compression_factor)

        self.transition = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, out_channels,
                      kernel_size=1, stride=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2))

    def forward(self, x):
        return self.transition(x)
```

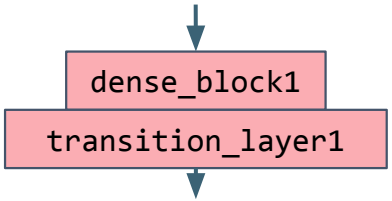


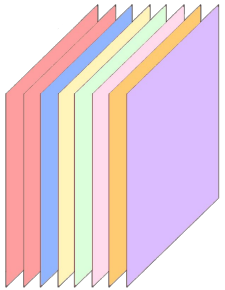


16x56x56

Transition Layers

```
class TransitionLayer(nn.Module):
    def __init__(self, in_channels, compression_factor=0.5):
        super(TransitionLayer, self).__init__()

        out_channels = int(in_channels * compression_factor)

        self.transition = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, out_channels,
                      kernel_size=1, stride=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2))

    def forward(self, x):
        return self.transition(x)
```

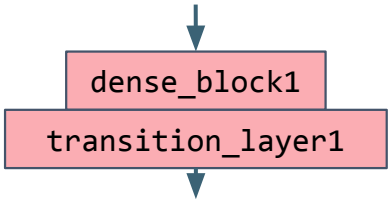


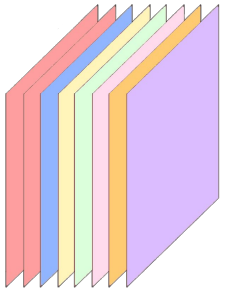


8x56x56

Transition Layers

```
class TransitionLayer(nn.Module):
    def __init__(self, in_channels, compression_factor=0.5):
        super(TransitionLayer, self).__init__()

        out_channels = int(in_channels * compression_factor)

        self.transition = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, out_channels,
                      kernel_size=1, stride=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2))

    def forward(self, x):
        return self.transition(x)
```

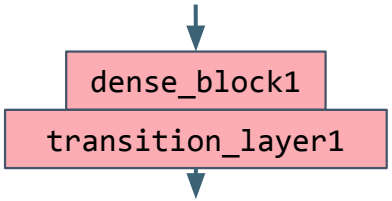


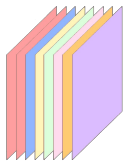


8x56x56

Transition Layers

```
class TransitionLayer(nn.Module):
    def __init__(self, in_channels, compression_factor=0.5):
        super(TransitionLayer, self).__init__()

        out_channels = int(in_channels * compression_factor)

        self.transition = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, out_channels,
                      kernel_size=1, stride=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2))

    def forward(self, x):
        return self.transition(x)
```

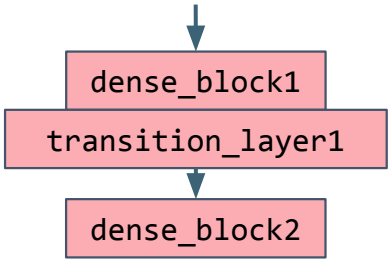


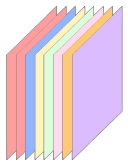


8x28x28

Transition Layers

```
class TransitionLayer(nn.Module):
    def __init__(self, in_channels, compression_factor=0.5):
        super(TransitionLayer, self).__init__()

        out_channels = int(in_channels * compression_factor)

        self.transition = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, out_channels,
                      kernel_size=1, stride=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2))

    def forward(self, x):
        return self.transition(x)
```





8x28x28