LAB 7:

ADVANCED CNN ARCHITECTURES

University of Washington, Seattle

Spring 2025

OUTLINE

Part 1: Limitations of conventional CNNs

- Very deep networks
- Degradation problem

Part 2: Additional CNN components

- Residual block
- Bottleneck block
- Inception modules

Part 3: ResNet example

- MNIST classification

Part 4: Additional resource: UW Hayak

- UW Hyak
- Access through RCC

Part 5: Lab Assignment

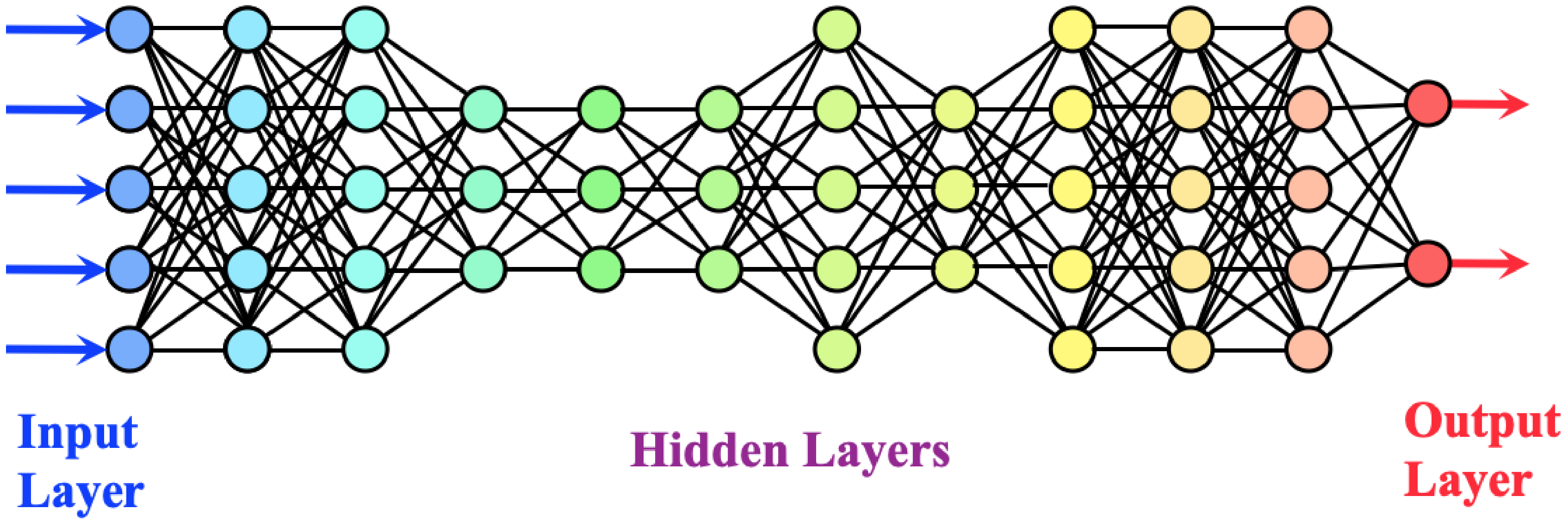
- Cifar-10 classification with ResNet

Limitation of conventional CNNs

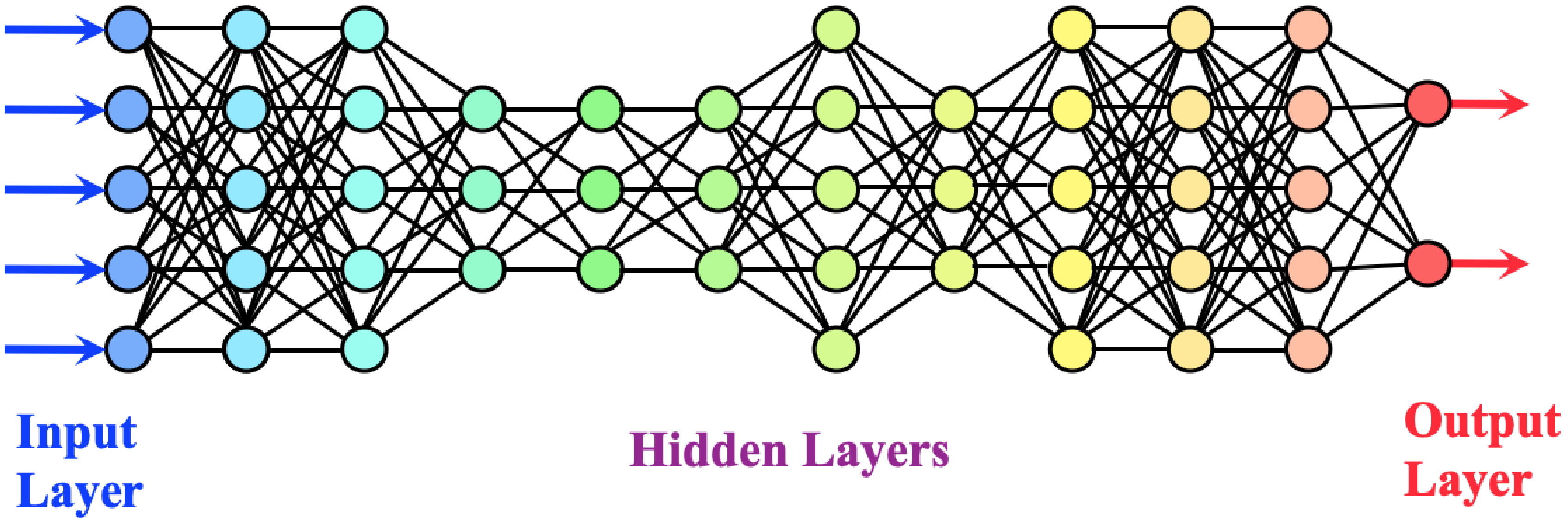
Very deep networks

Degradation problem

Very Deep Networks

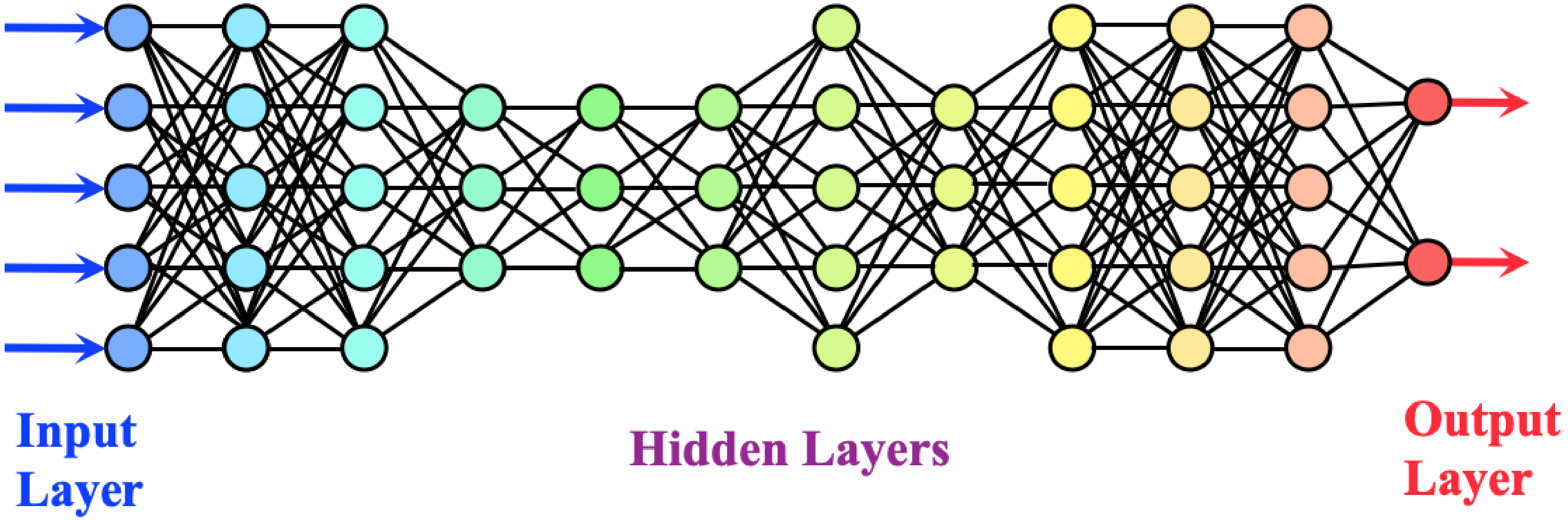


Very Deep Networks



>100 layers

Very Deep Networks



>100 layers

More layers → can process more complex patterns

Degradation problem

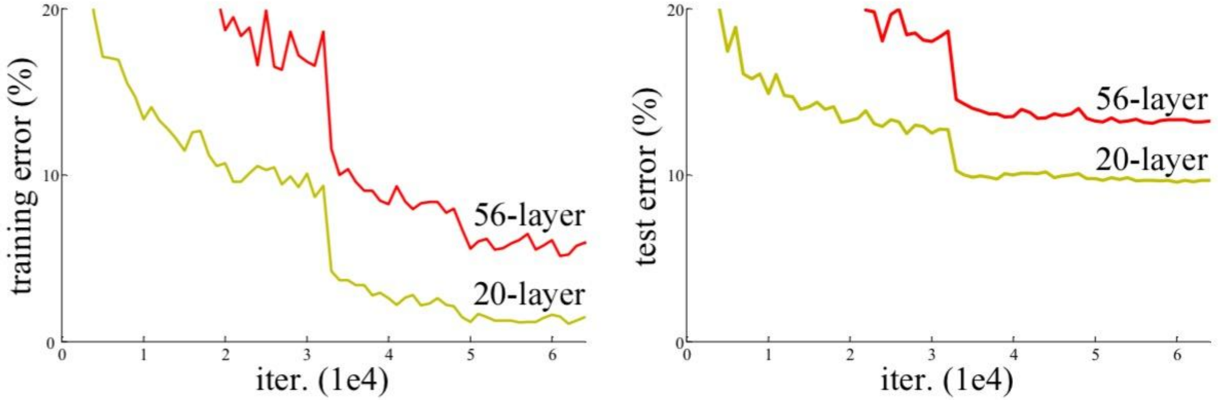


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

A Review of the Evolution of Deep Learning Architectures and Comparison of their Performances for Histopathologic Cancer Detection

Degradation problem

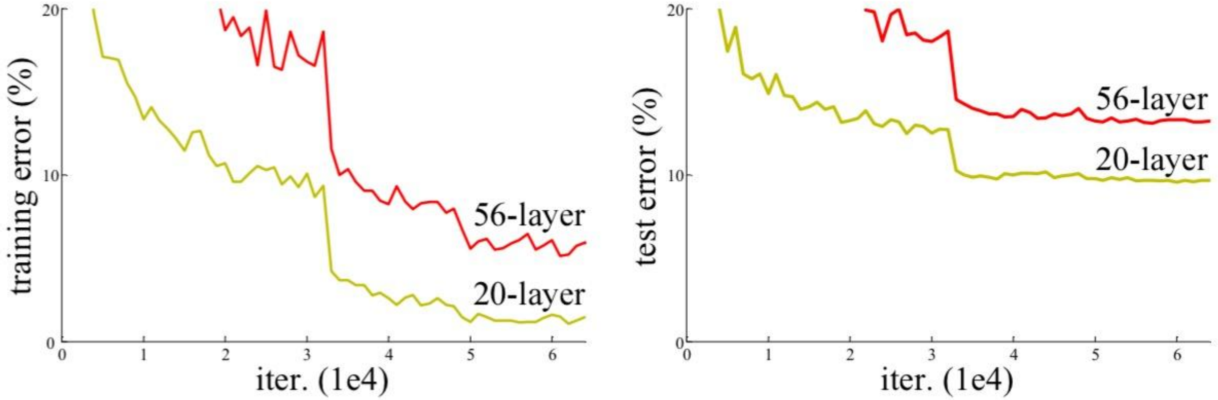
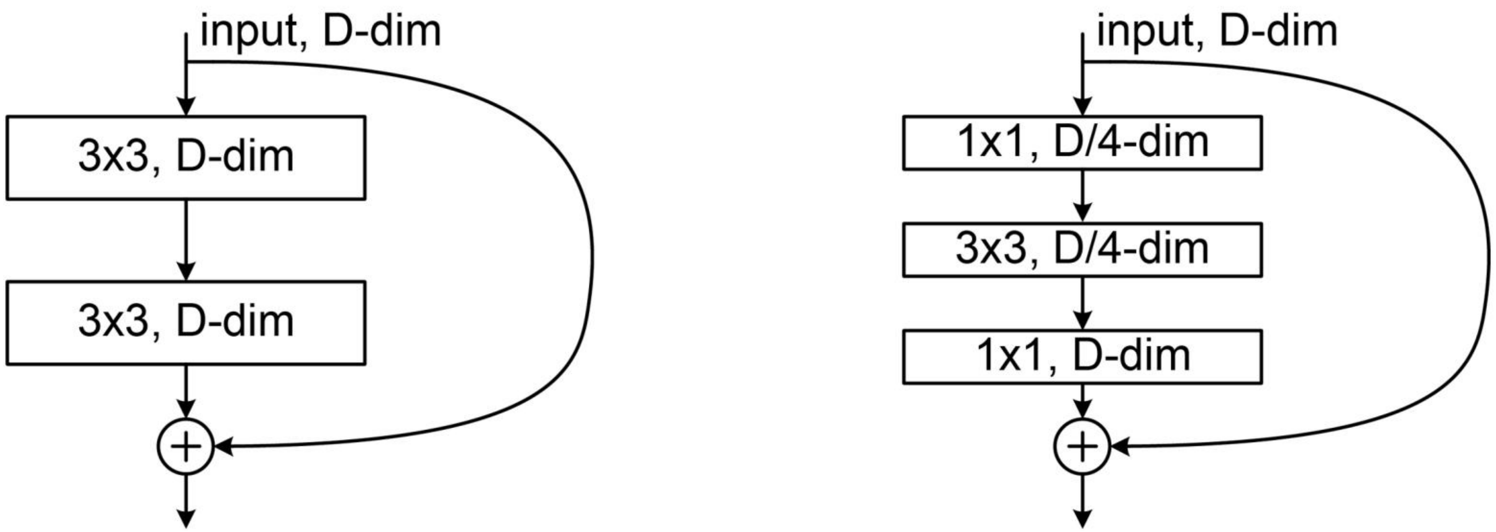


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

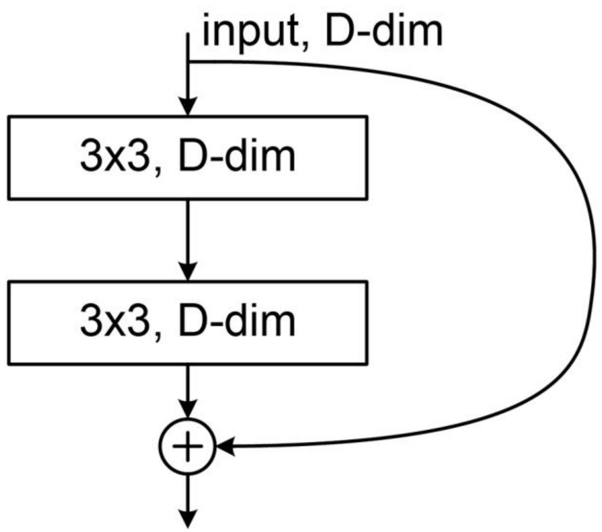
A Review of the Evolution of Deep Learning Architectures and Comparison of their Performances for Histopathologic Cancer Detection

Training / Testing accuracy start saturating faster than shallow networks

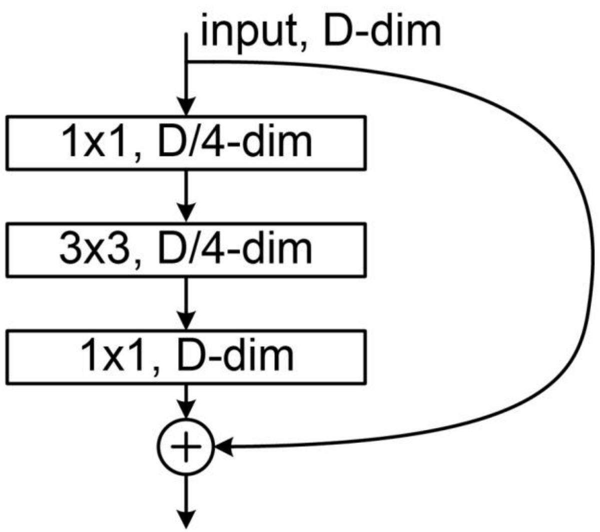
How to train very deep networks



How to train very deep networks



Skip connections
(Residual block)



Data compression
(Bottleneck layer)

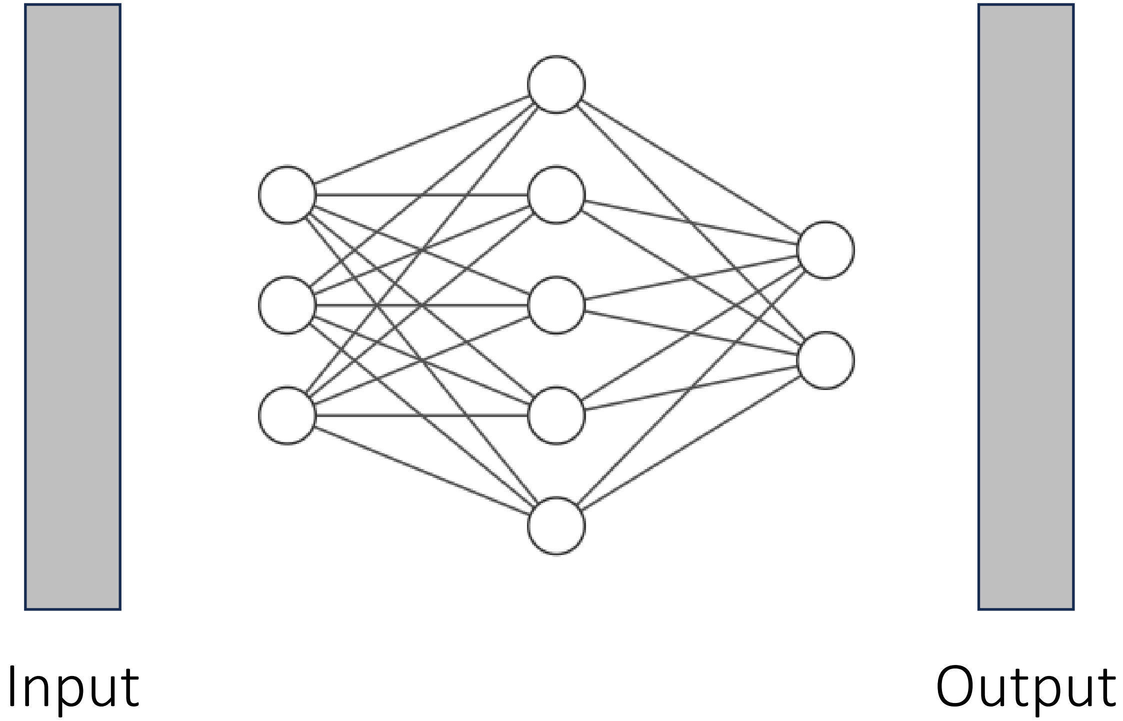
Additional CNN Blocks

Residual blocks

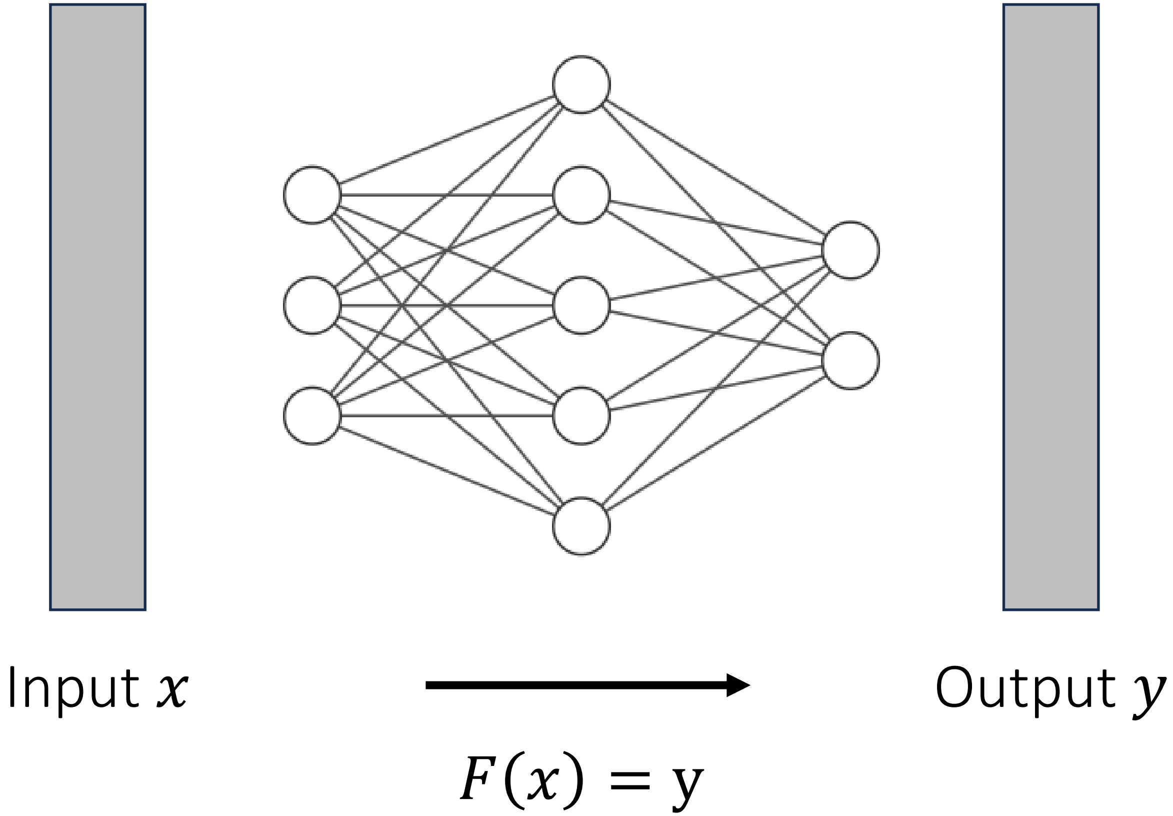
Bottleneck blocks

Inception modules

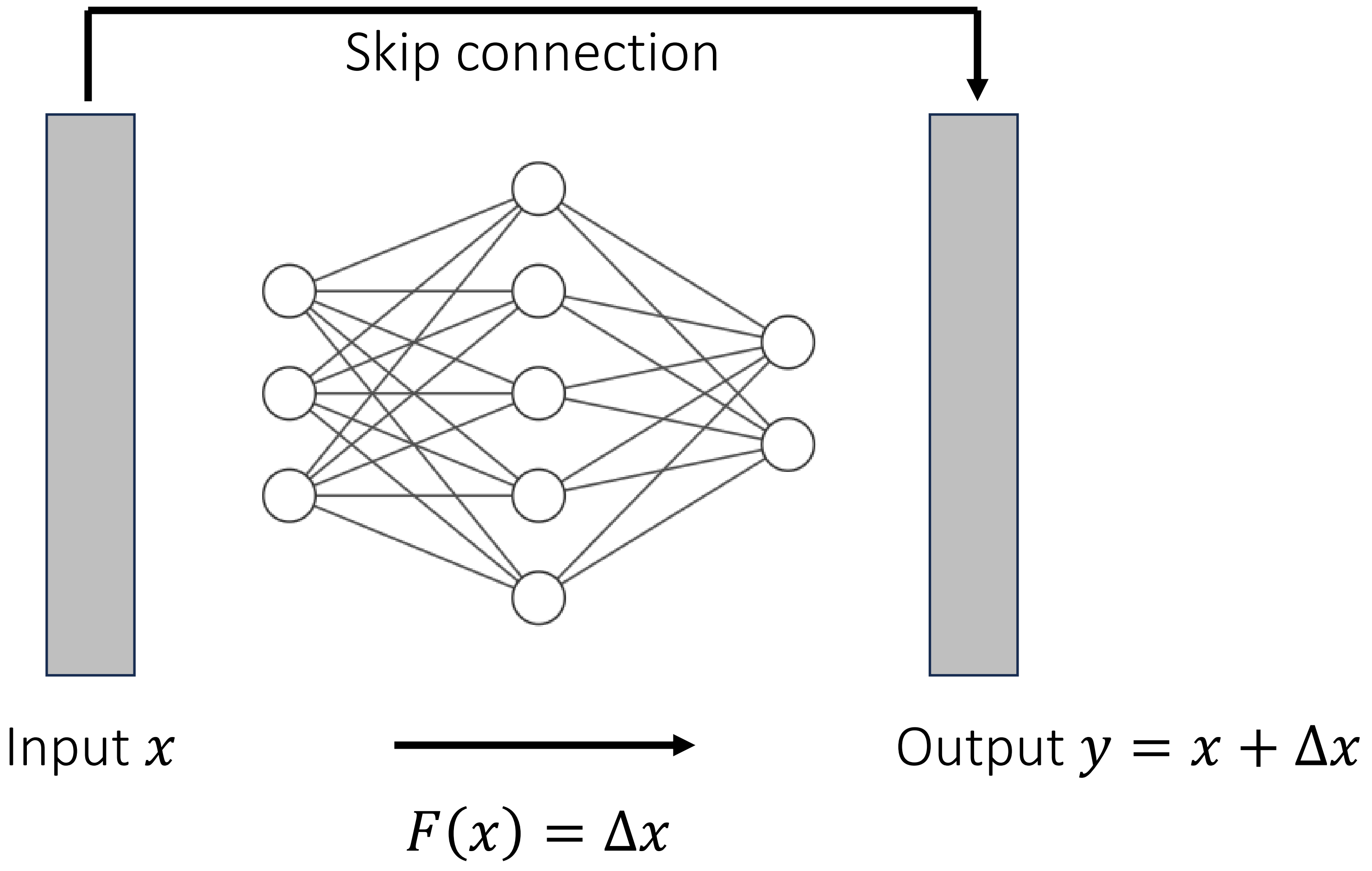
Residual blocks



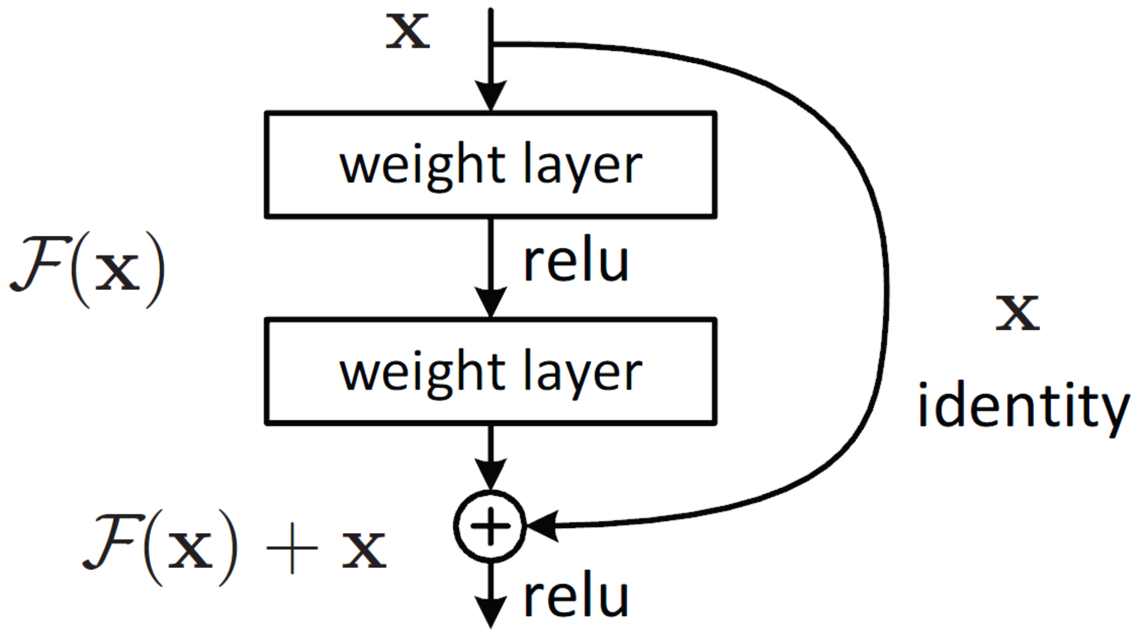
Residual blocks



Residual blocks

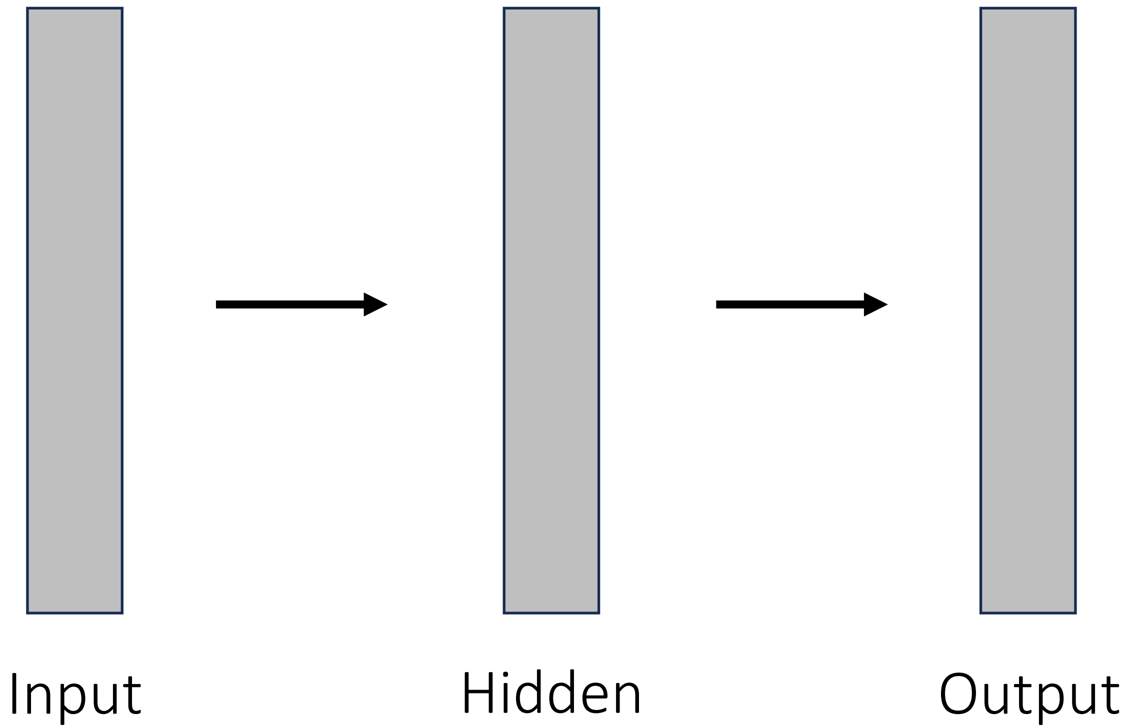


Residual blocks

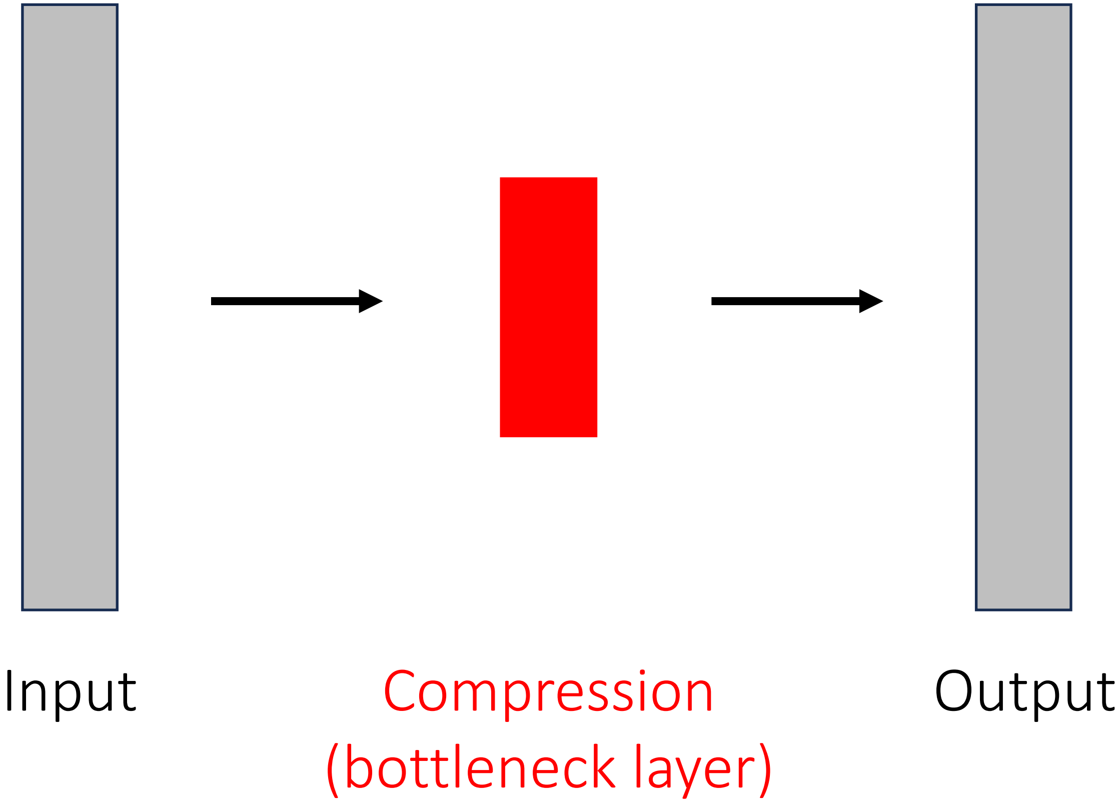


- Enable training deeper networks
- Prevents vanishing gradients
- Reduces overfitting

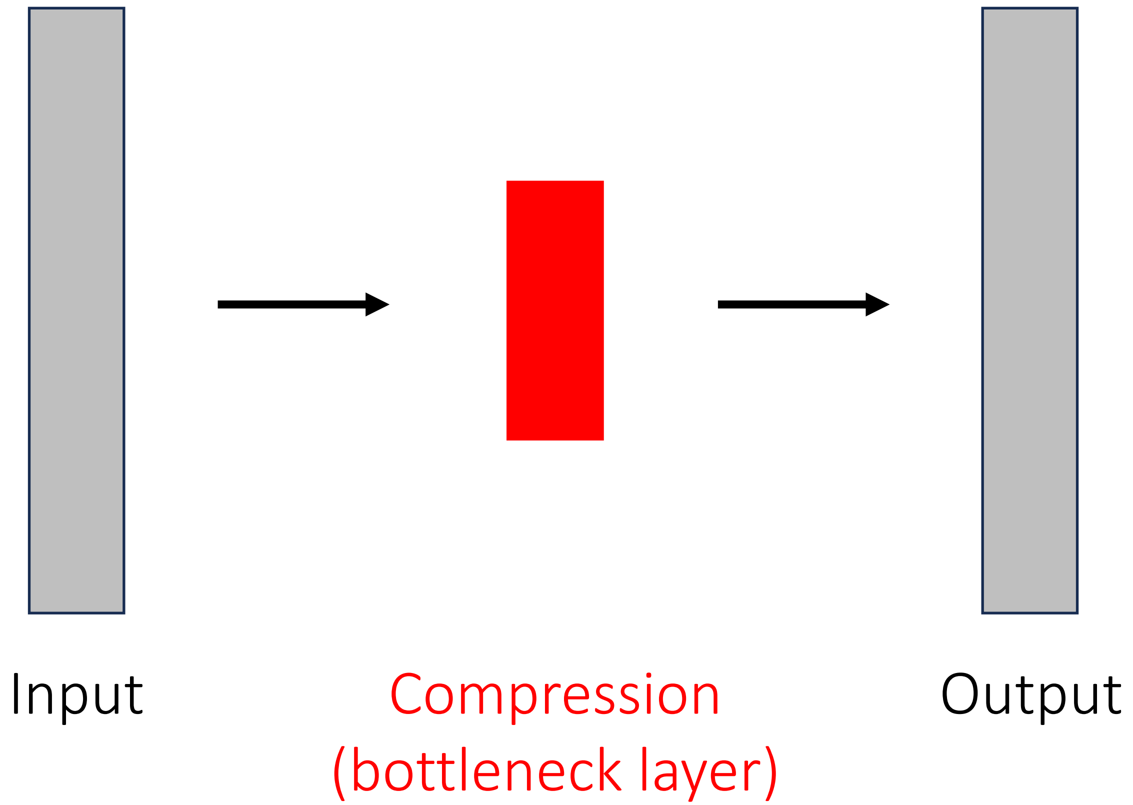
Bottleneck layer



Bottleneck layer

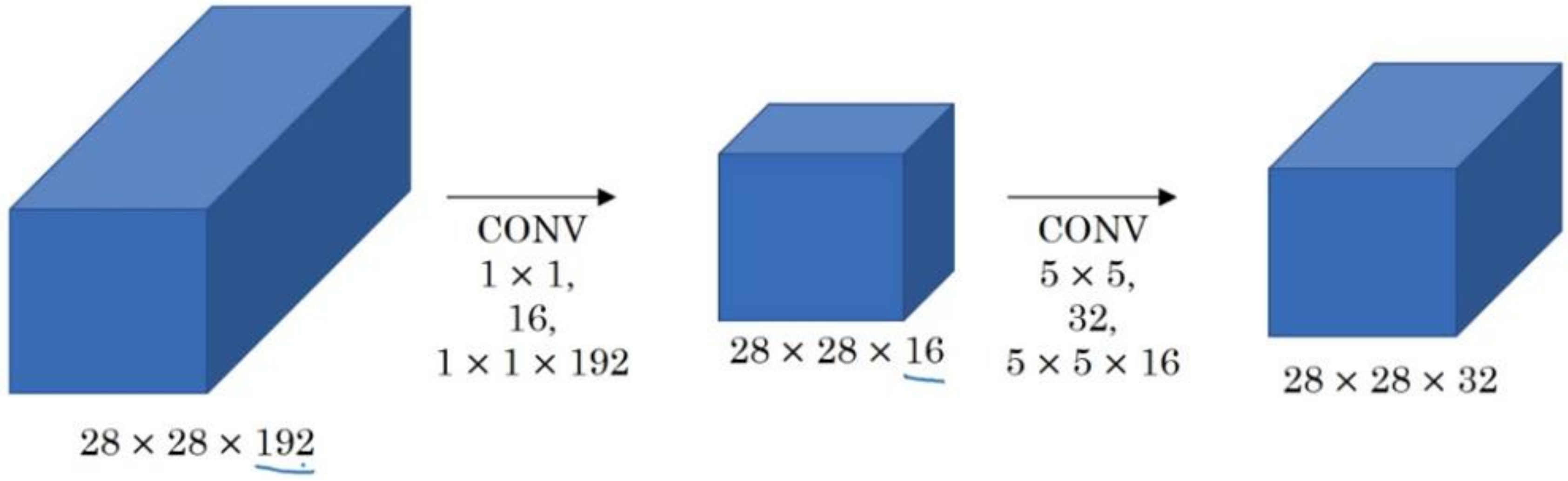


Bottleneck layer



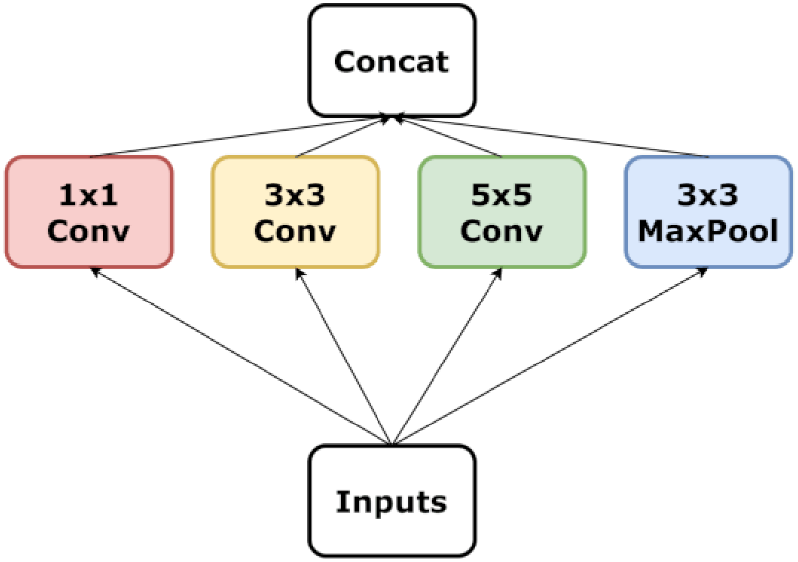
- Low dimensional representation
- More computationally efficient
- Reduces overfitting

Bottleneck layer in CNNs

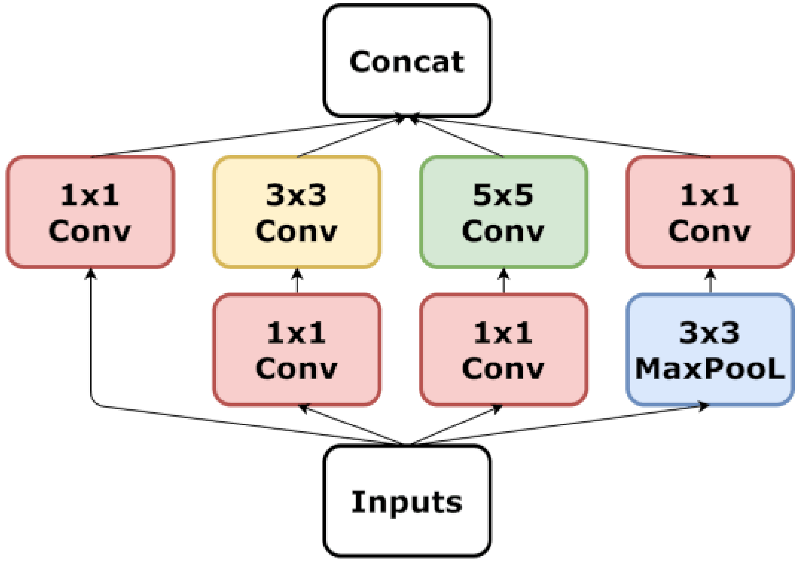


Compression
(bottleneck layer)

Inception module

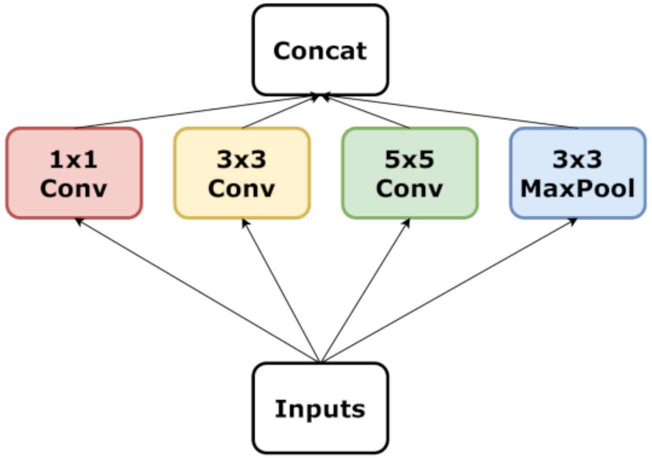


Inception Module

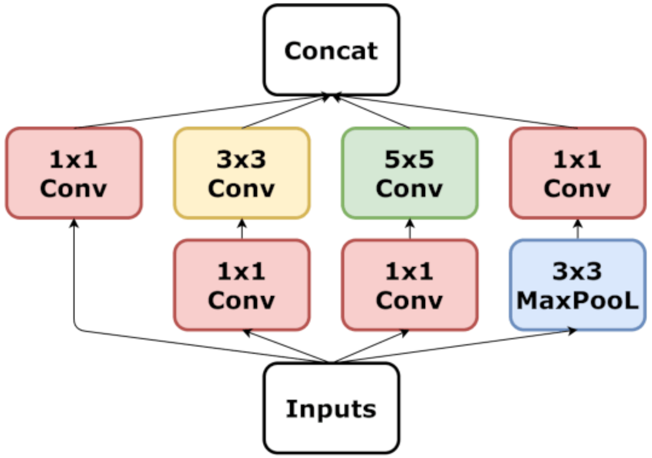


Inception Module with Dimension Reduction

Inception module



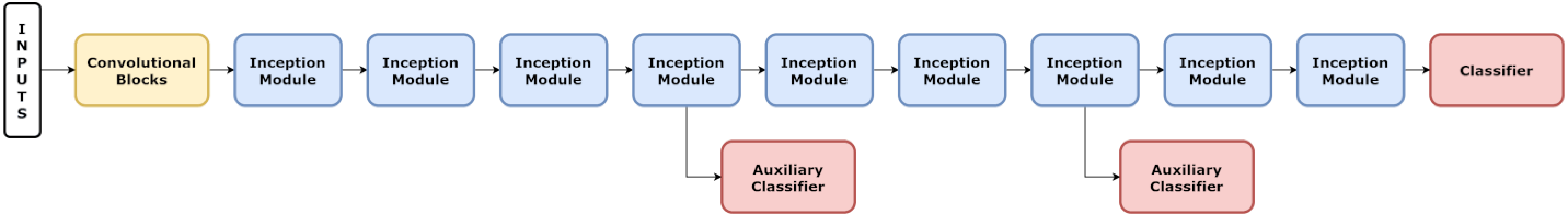
Inception Module



Inception Module with Dimension Reduction

- Parallel convolution operations
- Richer variety of features in a single output
- Can be used as standardized module

Inception module



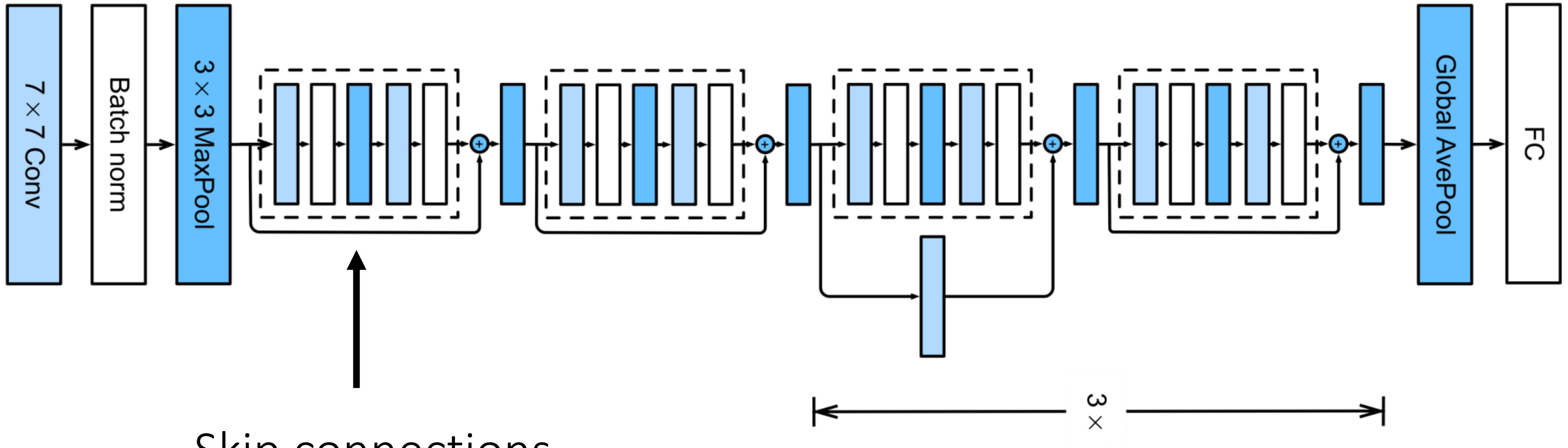
Inception V3 model

Advanced CNN examples

ResNet-18

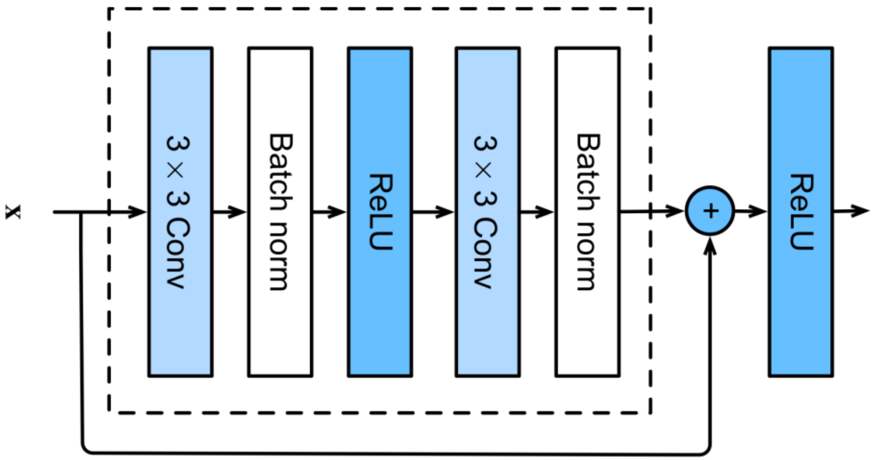
GoogLeNet

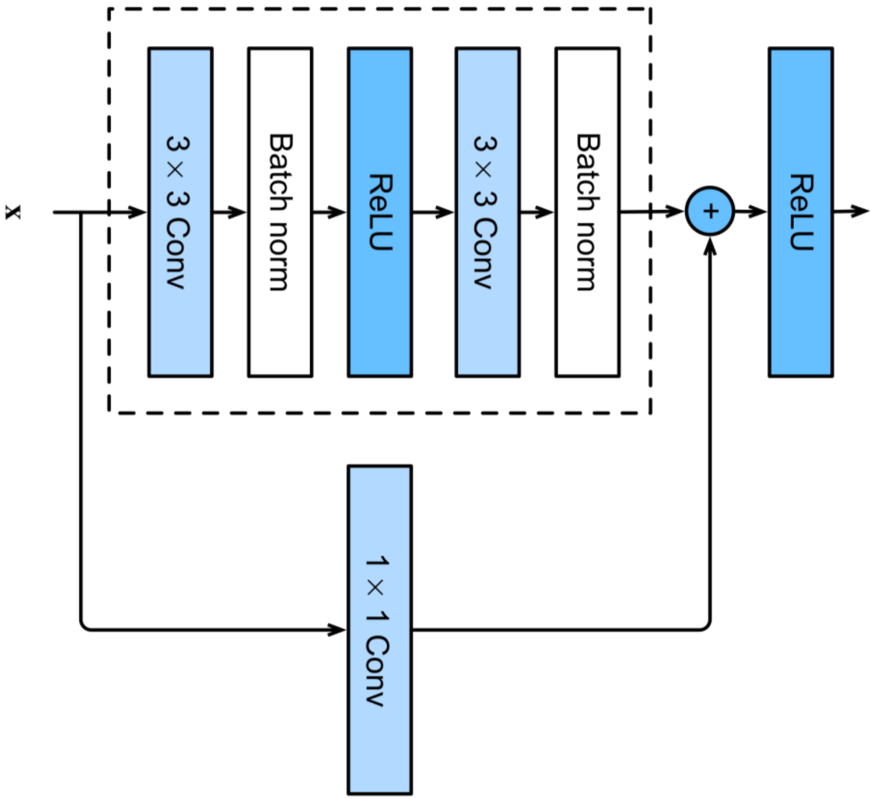
ResNet-18



Skip connections

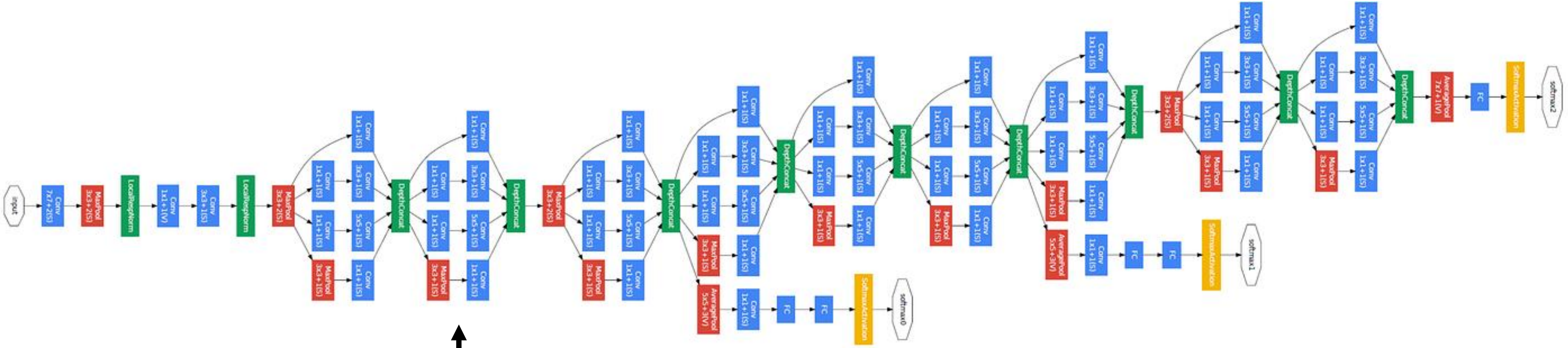
ResNet-18





1 x 1 conv for upsampling/downsampling

GoogLeNet



Inception module

ResNet Example

MNIST CLASSIFICATION

Prepare Data

```
from torchvision import datasets, transforms

transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
)

train_dataset = datasets.MNIST(
    root="./data", train=True, download=True, transform=transform
)
test_dataset = datasets.MNIST(
    root="./data", train=False, download=True, transform=transform
)
```

Normalize the images

Load data from torchvision library

Define Model

```
from torchvision.models import resnet18
```

```
class MNIST_ResNet_Classifier(torch.nn.Module):

    def __init__(self, out_channels, kernel_size, stride, padding):

        super(MNIST_ResNet_Classifier, self).__init__()

        self.resnet = resnet18()

        self.resnet.conv1 = torch.nn.Conv2d(1, out_channels = out_channels,
                                            kernel_size=(kernel_size, kernel_size),
                                            stride=(stride, stride),
                                            padding=(padding, padding),
                                            bias=False)

        self.resnet.fc = torch.nn.Linear(self.resnet.fc.in_features, 10)

    def forward(self, x):

        return self.resnet(x)
```

Using pre-built model

Pre-built resnet-18 components can be edited to your liking

Define Hyperparameters

```
MNIST_ResNet_Classifier = MNIST_ResNet_Classifier(out_channels = 64,
                                                   kernel_size = 7,
                                                   stride = 2,
                                                   padding = 3)

# Define learning rate and epochs
learning_rate = 0.001
epochs = 10

# Batch size for mini-batch gradient
batchsize = 64

# Using Adam as optimizer
loss_func = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(MNIST_ResNet_Classifier.parameters(), lr=learning_rate)

MNIST_ResNet_Classifier.cuda()
```

Initialize model

Initialize hyperparams

Using CE loss and Adam optimizer

Identify Tracked Values

```
train_loss_list = []
```

Store training data loss

Train Model

```
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batchsize, shuffle=True
)

test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=1000, shuffle=False
)
```

Using dataloader to batch train and test dataset

```
for epoch in range(epochs):

    for batch_idx, (train_input, train_target) in enumerate(train_loader):

        optimizer.zero_grad()

        pred = MNIST_ResNet_Classifier(train_input.cuda())

        loss = loss_func(pred, train_target.cuda())

        loss.backward()

        optimizer.step()

    print("Train Epoch: {} \tLoss: {:.6f}".format(epoch, loss.item()))
```

Training cycle

Print training loss each epoch

Visualize and Evaluate Model

```
MNIST_ResNet_Classifier.eval()
test_loss = 0
correct = 0

with torch.no_grad():

    for test_input, test_target in test_loader:

        output = MNIST_ResNet_Classifier(test_input.cuda())

        test_loss += loss_func(output, test_target.cuda()).item()

        pred = output.argmax(dim=1, keepdim=True)

        correct += pred.eq(test_target.view_as(pred).cuda()).sum().item()
```

```
test_loss /= len(test_loader.dataset)

print("Test Accuracy: {}/{} ({:.0f}%)\n".format(correct, len(test_loader.dataset), 100.0 * correct / len(test_loader.dataset)))
```

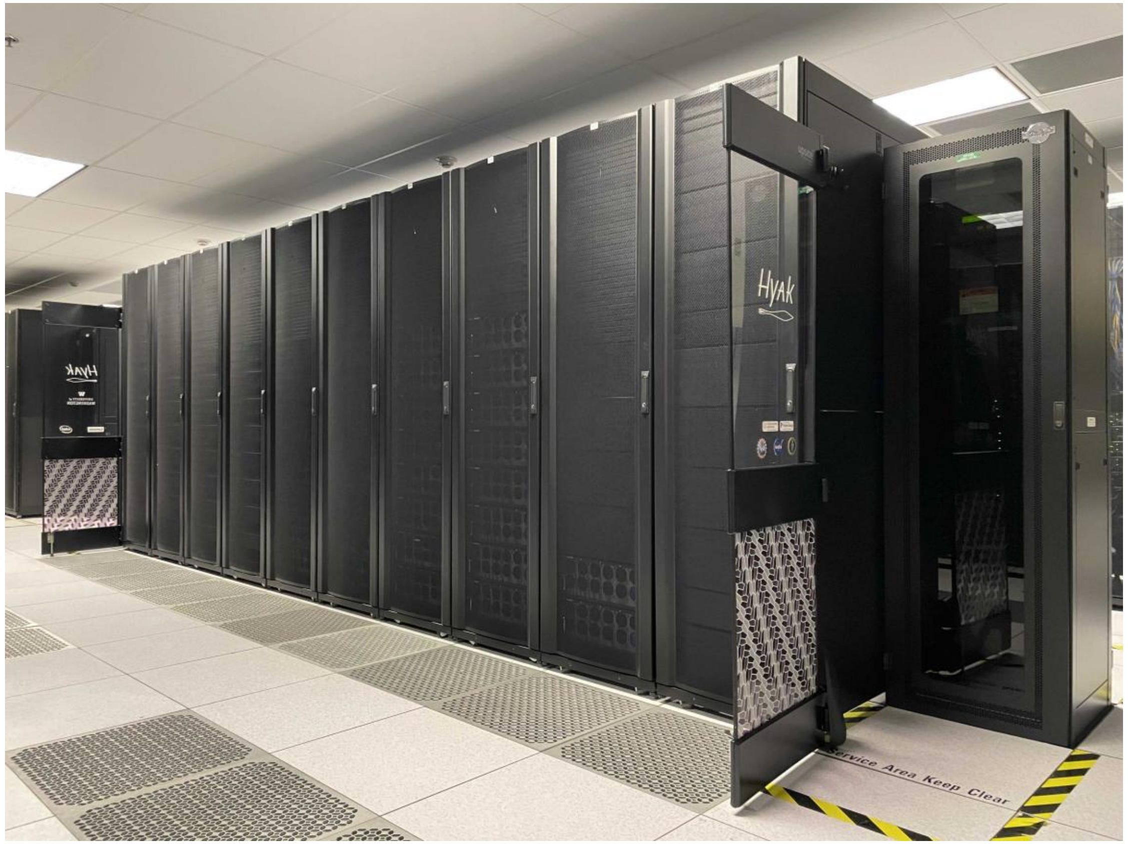
```
Test Accuracy: 9921/10000 (99%)
```

99% accuracy within 10 epochs

Additional UW Resources

UW Hayak supercomputer

UW Hyak



- 30932 CPU cores
- 832 GPU (Turing, Ampere)
- Supports Python and Jupyter interface

Access through UW RCC

Hyak Access

Applying for Access

1. Make sure you are eligible and have [joined RCC].
2. Review the [Hyak Prerequisites]. Hyak is used through a linux command line. For many researchers, HPC resources are the first time they encounter this. This is fine! But you should spend a bit of time making sure you grasp the basics before moving to the supercomputer.
3. Read the [Hyak Documentation] in full. Further, you should consider attending an in-person [Hyak training session].
4. Read and agree to the RCC Hyak resource [terms of service].
5. Complete the [RCC Hyak skills assessment]. Completion of this form serves as your application for access to Hyak and will trigger the review of all requirements. It may take up to 5 business days to grant access. If you have not heard back after 5 days, you may email RCC leadership or reach out to an officer on slack.

Completing Set Up

Once you receive an email stating that your access has been granted, there's a few more steps:

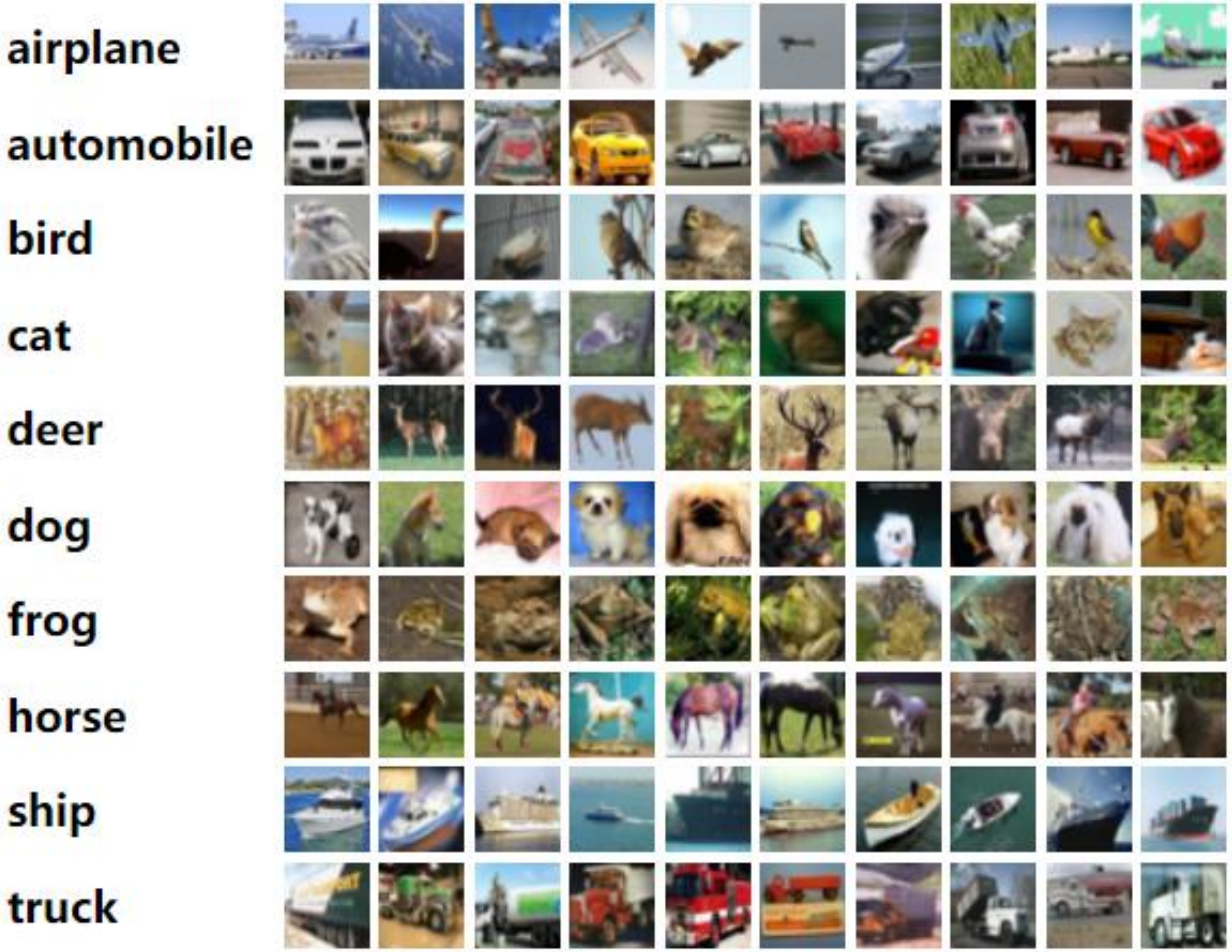
1. Make sure two-factor authentication is set up. It almost certainly is, but you can check this at https://itconnect.uw.edu/tools-services-support/access-authentication/2fa/.
2. Be sure to **[Subscribe to the Hyak mailing list]** for reminders about monthly maintenance and training opportunity announcements.

- Join UW Research Computing Club
- Read Hyak Prereq / Documentation
- Take skill assessment for access
- Gain access to Hyak system

Lab Assignment

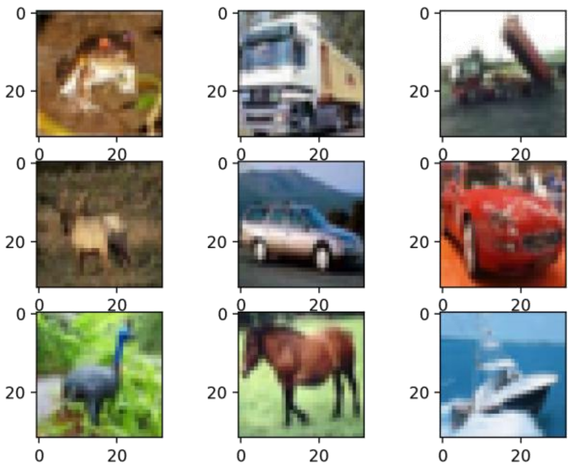
CIFAR-10 Classification with ResNet-18

CIFAR-10 Dataset



- 60000 32 x 32 colour images
- 10 classes
- 6000 per class
- 50000 train
- 10000 test

CIFAR-10 Classification with ResNet



ResNet-18

1	**airplane**
0	**automobile**
0	**bird**
0	**cat**
0	**deer**
0	**dog**
0	**frog**
0	**horse**
0	**ship**
0	**truck**

Softmax Output

In this exercise, you will classify colour image (32 x 32 x 3) using your own variant of **ResNet-18.**

Prior to training your neural net, 1) Normalize the dataset and 2) Split the dataset into train/validation/test.
You may also need to augment your dataset (e.g., flipping for better performance)

Feel free to tweak ResNet-18 architecture with your choices of **channels, kernel size, stride, padding** etc.

Your goal is to **achieve a testing accuracy of >=85%** with no restrictions on epochs.

Demonstrate the performance of your model via plotting the **training loss, validation accuracy** and printing out the **testing accuracy.**

After your model has reached the goal, print the accuracy of each class similar to Lab 3. What is the class that your model performed best and worst?