Homework 4

Yuanteng Chen 3039725444

2. Feature Dimension of Convolutional Neural network

(a)

(1) $\begin{cases} \text{weights: } F \cdot C \cdot K^2 \\ \text{bias: } F \end{cases}$

(2). $W_{out} = (W-K+2P)/S + 1$

$H_{out} = (H-K+2P)/S+1$

$C_{out} = F$

(b). $W_{out} = (W_{in}-K)/S+1$

$H_{out} = (H_{in}-K)/S+1$

$C_{out} = C_{in}$

(c) receptive:

$RF_{i+1} = S_i(RF_i - 1) + K_i$

where RF_i means the receptive field of the ith layer and $S_i \to$ stride, $K_i \to$ kernel size

as stride step size $=1$.

∴ the receptive field size of last output

is $L \cdot K - (L-1) = L(K-1)+1$

(d) $RF_{i+1} = S_i (RF_i - 1) + k_i$

kernel size = 2 and stride step size = 2

$\therefore RF_{i+1} = 2 \cdot (RF_i - 1) + 2$

$= 2 RF_i$

$\therefore$ The receptive field size increases by 2.

as the output feature resolution decreases, we reduce the amount of computation, so the number of matrix multiply operations decreases.

(e).

Layer	parameters.	dimension
Input	0	$28 \times 28 \times 1$
Conv3-10	$(10 + 3\times3\times1)\times10$ $= 100$	$28\times28\times10$ $(28+2\times1-3)/1+1=28$
pool-2	0	$14\times14\times10$
Conv3-10	$10+3\times3\times10\times10$ $= 910$	$14\times14\times10$
Pool-2	0	$7\times7\times10$
Flatten	0	490
FC-3	$490\times3+3$ $= 1473$	3

(f)

Conv2-3 → Relu → Conv2-3 → Relu → GAP → FC-3

$f(x_3) = f(x_2) = [0, 2.8, 0]^T$

$f(x_4) = f(x_1) = [0.8, 0, 0]^T$

as CNN is invariant of circular shifs.

3. Convolutional networks.

(a).

①. Convolutional layer utilize weights sharing which reduces the number of parameters compared to fully connected layers.

② Convolution layers are invariant to circular shift which means they can detect features regardless of their exact positions in an image.

(b). $[1, 4, 0, -2, 3] \rightarrow [-2, 2, 11]$

assume filter = $[a, b, c]$

$$\therefore \begin{cases} a + 4b = -2 \\ 4a - 2c = 2 \\ -2b + 3c = 11 \end{cases} \Rightarrow \begin{cases} a = 2 \\ b = -1 \\ c = 3 \end{cases}$$

$\therefore$ filter $= [2, -1, 3]$

(c) the input size $= 2\times2$ $\begin{bmatrix} -1 & 2 \\ 3 & 1 \end{bmatrix}$

and pad $= 0$, stride $= 1$. kernel size $= 2\times2$

$\therefore$ the output size $= 3\times3$

input $\begin{bmatrix} -1 & 2 \\ 3 & 1 \end{bmatrix}$ filter $\begin{bmatrix} +1 & -1 \\ 0 & +1 \end{bmatrix}$

$$\begin{bmatrix} -1 & 1 \\ 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 2 & -2 \\ 0 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 3 & -3 \\ 0 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} -1 & 1+2 & -2 \\ 0+3 & \begin{matrix} -1+0 \\ -3+1 \end{matrix} & 2-1 \\ 0 & 3+0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 3 & -2 \\ 3 & -5 & 1 \\ 0 & 3 & 1 \end{bmatrix}$$

4. Convolutional networks and Dilated convolutions:

(a) $[B, C, H, W] = [10, 3, 32, 32]$.

i. 3×3; strid $=1$, padding $=1$

Sol: $H' = W' = (32+1\times2-3)/1+1 = 32$

$\therefore$ output $= [10, 64, 32, 32]$

ii: 4×4. strid $=2$. padding $=0$

Sol: $H' = W' = (32+0-4)/2+1 = 15$

$\therefore$ output $= [10, 64, 15, 15]$

(b). detects vertical edges:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

(c) 3×3 filter to blur an image:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

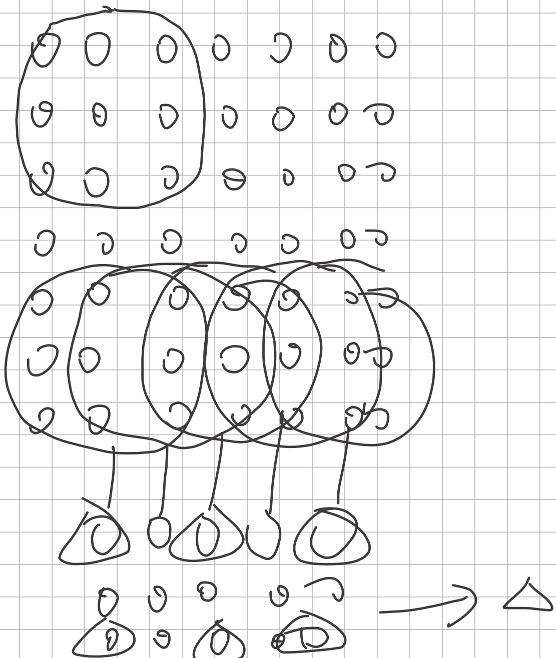
d. (i) $M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $\quad K = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

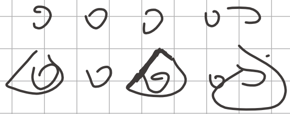
$$M' = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{output} = \begin{bmatrix} 5, & 10, & 5 \\ 10, & 20, & 10 \\ 5, & 10, & 5 \end{bmatrix}$$

(ii) I assume the stride size of both layers are 1:

sol: the receptive field of DilatedConv2 is 7





5. Weights and Gradients in a CNN

(a) Derive the gradient to the weight matrix: dW

Sol: $y_{i,j} = \sum_{h=1}^{k} \sum_{l=1}^{k} X_{i+h-1,j+l-1} W_{h,j}$

$$\frac{\partial L}{\partial W_{h,j}} = \sum_{i=1}^{m} \sum_{j=1}^{m} \frac{\partial L}{\partial y_{i,j}} \cdot \frac{\partial y_{i,j}}{\partial W_{h,j}}$$

$$\frac{\partial y_{i,j}}{\partial W_{h,j}} = X_{i+h-1,j+l-1}, \quad \frac{\partial L}{\partial y_{i,j}} = dy_{i,j}$$

$$\therefore \frac{\partial L}{\partial W_{h,j}} = \sum_{i=1}^{m} \sum_{j=1}^{m} dy_{i,j} \cdot X_{i+h-1,j+l-1}$$

$$\therefore dW = X \cdot dY$$

after 1 step SGD (assum learning rate $= \lambda$)

$$\therefore W_{t+1} = W_t - \lambda dW$$
$$= W_t - \lambda X \cdot dY$$

(b) $E[X_{i,j}] = 0$, $Var(X_{i,j}) = \sigma_x^2$

$E(dy_{i,j}) = 0$ $Var(dy_{i,j}) = \sigma_g^2$

Sol: $$\frac{\partial L}{\partial W_{h,j}} = \sum_{i=1}^{m} \sum_{j=1}^{m} dy_{i,j} \cdot X_{i+h-1,j+l-1}$$

$\because$ $X_{i,j}$ and $dy_{i,j}$ are independent

$$E\left(\frac{\partial L}{\partial w_{h,j}}\right) = \sum_{i=1}^{m}\sum_{j=1}^{m} E(dy_{i,j} \cdot X_{i+h-1, j+l-1})$$

$$= \sum_{i=1}^{m}\sum_{j=1}^{m} E(dy_{i,j}) \cdot E(X_{i+h-1, j+l-1})$$

又 $\because E(X_{i,j}) = E(dy_{i,j}) = 0$

$\therefore E\left(\frac{\partial L}{\partial w_{h,j}}\right) = 0$

$$Var\left(\frac{\partial L}{\partial w_{h,j}}\right) = E\left(\left(\frac{\partial L}{\partial w_{h,j}}\right)^2\right) - \left(E\left(\frac{\partial L}{\partial w_{h,j}}\right)\right)^2$$

$$= E\left(\left(\frac{\partial L}{\partial w_{h,j}}\right)^2\right)$$

$$= E\left(\sum_{i=1}^{m}\sum_{j=1}^{m} X_{i+h-1, j+l-1}^2 \cdot dy_{h,l}^2\right)$$

又 $\because X_{i,j}$ and $dy_{i,j}$ are independent

$$\therefore = \sum_{i=1}^{m}\sum_{j=1}^{m} E(X_{i+h-1,j+l-1}^2) \cdot E(d^2y_{h,l})$$

$\because Var(X_{i,j}) = E(X_{i,j}^2) - (E(X_{i,j}))^2 = E(X_{i,j}^2) = \sigma_x^2$

$Var(dy_{i,j}) = E(dy_{i,j}^2) - (E(dy_{i,j}))^2 = E(dy_{i,j}^2) = \sigma_g^2$

$\therefore = m^2 \sigma_x^2 \cdot \sigma_g^2$

$\because m = (n+0-k)/1+1 = (n-k)+1$

standard deviation ~~of~~ of $\frac{\partial L}{\partial w_{h,j}}$

$$= \sqrt{Var\left(\frac{\partial L}{\partial w_{h,j}}\right)} = m\sigma_x\sigma_g = (n-k+1)\sigma_x\sigma_g$$

∴ the growth rate of the standard deviation of the gradient on $dW_{h,i}$ with respect to the length and width of the image n.

(c).

Sol: $y_{1,1} = X_{1,1} = \max(X_{1,1}, X_{1,2}, X_{2,1}, X_{2,2})$

$$\therefore \frac{dy_{1,1}}{dX_{1,1}} = 1 \qquad \frac{dy_{1,1}}{dX_{1,2}} = \frac{dy_{1,1}}{dX_{2,1}} = \frac{dy_{1,1}}{d_{2,2}} = 0$$

in average pooling: $y_{1,1} = \frac{1}{4}(X_{1,1} + X_{1,2} + X_{2,1} + X_{2,2})$

$$\therefore \frac{dy_{1,1}}{dX_{1,1}} = \frac{dy_{1,1}}{dX_{1,2}} = \frac{dy_{1,1}}{dX_{2,1}} = \frac{dy_{1,1}}{dX_{2,2}} = \frac{1}{4}$$

assume $i' = i/2$, $j' = j/2$

$$\therefore \frac{\partial y_{i',j'}}{\partial X_{i,j}} = \begin{cases} 1, & X_{i,j} = \max(X_{i,j}, X_{i+1,j}, X_{i,j+1}, X_{i+1,j+1}) \\ 0, & X_{ij} \neq \max(\cdots) \end{cases}$$

$$\therefore dX_{i,j} = \sum_{y_{i'j'}} dy_{i'j'} \frac{\partial y_{i'j'}}{\partial X_{i,j}} \quad (\text{max-pooling}),$$

$$dX_{i,j} = \frac{1}{4} y_{i'j'} \quad (\text{average-pooling})$$

(d). ① there is no learnable parameters in max-pooling or average-pooling. So they reduce the complexity of computation by decreasing the feature size.

② Without max-pooling or average pooling, CNN will not be invariant to circular shift as they increase the size of receptive field.

8. (a) CSDN, gpt

(b). None

(c) 10 hours

HW: Exploring Inductive Bias of Convolutional Neural Networks and Systematic Experimentation in Machine Learning

In this homework, we will study 1) what is inductive bias and how it affects the learning process, and 2) how to conduct systematic experiments in machine learning. We will compare convolutional neural networks (CNNs) and multi-layer perceptrons (MLPs) extensively as an example to study these two topics.

1. Inductive Bias

What is inductive bias? It is the assumption that the learning algorithm makes about the problem domain. Suppose that we build a machine learning system. We want to leverage the specific knowledge about the problem domain to make the learning process **more efficient** and the system **generalize much better** with fewer parameters. Let's be more precise. What do exactly **more efficient** and **generalize much better** mean? The learning process is more efficient 1) if we can learn the model with fewer parameters, 2) if we can learn the model with fewer data, and 3) if we can learn the model with fewer iterations. And the system generalizes much better if the model can generalize to the unseen data well.

We have already observed the power of inductive bias. We know that CNN generalizes better than MLP even with the same number of parameters. We partially concluded that is because CNN has the inductive bias that the model is translation invariant. We will study the inductive bias of CNN in more detail in this homework.

In this homework, we will use the edge detection task as an example to study the inductive bias of CNN. We will compare CNN and MLP extensively. And we will see when CNN can fail.

2. Systematic Experimentation in Machine Learning

How can we prove our hypothesis that CNN has the inductive bias that the model is translation invariant? We conduct extensive experiments in machine learning research (and other fields) to prove our hypothesis. In this context, systematic experimentation refers to running a series of experiments to prove our hypothesis. In this homework, we will study how to conduct systematic experimentation in machine learning.

Let's take a step back and think about 1) what our hypothesis is and 2) what experiments are needed to conduct to prove our hypothesis. The first question is easy. The hypothesis is that CNN has the inductive biases of locality and translational invariance. It is not enough to show that CNN performs better than MLP with the same number of parameters. Then, how do we design the experiments to prove our hypothesis? In this homework, we will design the experiments, conduct the experiments, analyze the results, and draw a conclusion.

In [2]:
```
import numpy as np
import random
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset
import torchvision
import torchvision.transforms as T
from torchvision.transforms import ToPILImage

from PIL import Image
from scipy.ndimage.interpolation import rotate
from sklearn.linear_model import LogisticRegression
from tqdm import tqdm
from copy import deepcopy
from torch.utils.data import DataLoader
```

Helper functions

The following code cell defines function and classes that will be used in the succeeding codes. Feel free to check it if you are not sure about details.

In [3]:
```
class EdgeDetectionDataset(Dataset):
    def __init__(self, domain_config, mode="train", transform=None) -> None:
        """
        Args:
            domain_config (dict): Domain configuration
                data_per_class (int): Number of data per class
                num_classes (int): Number of classes
                class_type (list): List of class types
                spatial_resolution (int): length of height and width of the image
                max_edge_width (int): Maximum edge width
                max_edge_intensity (float): Maximum edge intensity
                min_edge_intensity (float): Minimum edge intensity
                max_background_intensity (float): Maximum background intensity
                min_background_intensity (float): Minimum background intensity
                possible_edge_location_ratio (float): Confine the possible edge l
                num_horizontal_edge (int): Number of horizontal edges
                num_vertical_edge (int): Number of vertical edges
                use_permutation (bool): Whether to apply random permutation on th
            mode (str): Mode of the dataset (train, val, test)
            transform (callable, optional): Optional transform to be applied on a
        """
        self.data_per_class = domain_config.get("data_per_class", 1000)
        self.num_classes = domain_config.get("num_classes", 3)
        self.class_type = domain_config.get(
            "class_type", ["horizontal", "vertical", "none"]
        )
        self.spatial_resolution = domain_config.get("spatial_resolution", 28)
        self.min_edge_width = domain_config.get("min_edge_width", 1)
        self.max_edge_width = domain_config.get("max_edge_width", 4)
        self.max_edge_intensity = domain_config.get("max_edge_intensity", 1)
        self.min_edge_intensity = domain_config.get("min_edge_intensity", 0.25)
        self.max_background_intensity = domain_config.get(
            "max_background_intensity", 0.2
        )
        self.min_background_intensity = domain_config.get("min_background_intens
        self.possible_edge_location_ratio = domain_config.get(
            "possible_edge_location_ratio", 1.0
        )
        self.num_horizontal_edge = domain_config.get("num_horizontal_edge", 1)
        self.num_vertical_edge = domain_config.get("num_vertical_edge", 1)
        self.num_diagonal_edge = domain_config.get("num_diagonal_edge", 1)
        self.use_permutation = domain_config.get("use_permutation", False)
        self.permutater = domain_config.get("permutater", None)
        self.unpermutater = domain_config.get("unpermutater", None)

        if self.possible_edge_location_ratio < 1.0:
            self.train_val_domain_shift = True
        else:
            self.train_val_domain_shift = False

        self.possible_edge_location = int(
            self.possible_edge_location_ratio * self.spatial_resolution
        )
        self.mode = mode

        assert self.num_classes == len(
            self.class_type
        ), "Number of classes must match the number of class types"

        assert self.mode in (
            "train",
```

```python
            "valid",
            "test",
        ), "Mode must be either train, valid, or test"

        self.X = None
        self.y = None

        if self.use_permutation:
            assert self.permutater is not None, "permutater must be provided"
            assert self.unpermutater is not None, "Unpermutater must be provide

        self._generate_dataset()

        self.transform = transform

    def __len__(self):
        """
        Returns:
            int: Length of the dataset
        """
        return len(self.X)

    def __getitem__(self, idx):
        """
        Args:
            idx (int): Index of the sample
        Returns:
            tuple: (sample, label)
        """
        if torch.is_tensor(idx):
            idx = idx.tolist()

        sample = self.X[idx]
        label = self.y[idx]

        if self.transform:
            sample = self.transform(sample)

        return sample, label

    def get_permutater(self):
        """
        Returns:
            np.ndarray: Permutation matrix
        """
        return self.permutater

    def get_unpermutater(self):
        """
        Returns:
            np.ndarray: Unpermutation matrix
        """
        return self.unpermutater

    def _permute_pixels(self, X):
        """
        Args:
            X (np.ndarray): Image
        Returns:
            np.ndarray: Permuted image
        """
```

```
        assert X.shape[0] == self.data_per_class, "Invalid image shape"
        assert len(X.shape) == 4, "Invalid image shape"

        n, h, w, c = X.shape

        X = X.reshape(n, h * w, c)
        X = X[:, self.permutater, :]
        X = X.reshape(n, h, w, c)

        return X

    def _edge_intensity(self, edge_type="horizontal"):
        """
        Args:
            edge_type (str): Type of edge (horizontal, vertical, both, diagonal)
        Returns:
            np.ndarray: Edge intensity
        """
        if edge_type == "horizontal":
            num_edge = self.num_horizontal_edge
        elif edge_type == "vertical":
            num_edge = self.num_vertical_edge
        elif edge_type == "diagonal":
            num_edge = self.num_diagonal_edge
        elif edge_type == "both":
            num_edge = self.num_horizontal_edge + self.num_vertical_edge
        else:
            raise ValueError("Invalid edge type")

        return np.random.uniform(
            self.min_edge_intensity,
            self.max_edge_intensity,
            size=(self.data_per_class, num_edge),
        )

    def _edge_location(self, edge_type="horizontal"):
        """
        Args:
            edge_type (str): Type of edge (horizontal, vertical, both, diagonal)
        Returns:
            np.ndarray: Edge location
        """
        max_edge_width = self.max_edge_width + 1
        if edge_type == "horizontal":
            num_edge = self.num_horizontal_edge
        elif edge_type == "vertical":
            num_edge = self.num_vertical_edge
        elif edge_type == "diagonal":
            num_edge = self.num_diagonal_edge
            max_edge_width = int(self.max_edge_width / np.sqrt(2))
        elif edge_type == "both":
            num_edge = self.num_horizontal_edge + self.num_vertical_edge
        else:
            raise ValueError("Invalid edge type")

        edge_width = np.random.randint(
            self.min_edge_width, max_edge_width, size=(self.data_per_class, num_e
        )

        if self.mode == "train" and self.train_val_domain_shift:
            edge_location_start_idx = np.random.randint(
```

```
                1,
                self.possible_edge_location,
                size=(self.data_per_class, num_edge),
            )
            edge_location_end_idx = np.clip(
                edge_location_start_idx + edge_width,
                0,
                self.possible_edge_location-1,
            )

        elif self.mode == "valid" and self.train_val_domain_shift:
            edge_location_start_idx = np.random.randint(
                self.possible_edge_location,
                self.spatial_resolution,
                size=(self.data_per_class, num_edge),
            )
            edge_location_end_idx = np.clip(
                edge_location_start_idx + edge_width,
                self.possible_edge_location,
                self.spatial_resolution-1,
            )

        else:
            edge_location_start_idx = np.random.randint(
                1,
                self.spatial_resolution,
                size=(self.data_per_class, num_edge),
            )
            edge_location_end_idx = np.clip(
                edge_location_start_idx + edge_width,
                0,
                self.spatial_resolution-1,
            )

        return edge_location_start_idx, edge_location_end_idx

    def _generate_hoizontal_edge_images(self):
        """
        Generate horizontal edge images
        Returns:
            np.ndarray: Generated horizontal edge images
        """
        assert (
            self.num_horizontal_edge > 0
        ), "Number of horizontal edge must be greater than 0"

        X = self._generate_background_images()

        edge_location_start_idx, edge_location_end_idx = self._edge_location(
            edge_type="horizontal"
        )
        edge_intensity = self._edge_intensity()

        for i in range(self.data_per_class):
            for j in range(self.num_horizontal_edge):
                X[
                    i, edge_location_start_idx[i, j] : edge_location_end_idx[i,
                ] = edge_intensity[i, j]

        return X
```

```python
    def _generate_vertical_edge_images(self):
        """
        Generate vertical edge images
        Returns:
            np.ndarray: Generated vertical edge images
        """
        assert (
            self.num_vertical_edge > 0
        ), "Number of vertical edge must be greater than 0"

        X = self._generate_background_images()

        edge_location_start_idx, edge_location_end_idx = self._edge_location(
            edge_type="vertical"
        )
        edge_intensity = self._edge_intensity()

        for i in range(self.data_per_class):
            for j in range(self.num_vertical_edge):
                X[
                    i,
                    :,
                    edge_location_start_idx[i, j] : edge_location_end_idx[i, j],
                    :,
                ] = edge_intensity[i, j]

        return X

    def _generate_both_edge_images(self):
        """
        Generate horizontal/vertical edge images
        Returns:
            np.ndarray: Generated horizontal/vertical edge images
        """
        assert (
            self.num_horizontal_edge > 0
        ), "Number of horizontal edge must be greater than 0"
        assert (
            self.num_vertical_edge > 0
        ), "Number of vertical edge must be greater than 0"

        X = self._generate_background_images()

        edge_location_start_idx, edge_location_end_idx = self._edge_location(
            edge_type="both"
        )
        edge_intensity = self._edge_intensity(edge_type="both")

        for i in range(self.data_per_class):
            for j in range(self.num_horizontal_edge):
                X[
                    i,
                    edge_location_start_idx[i, j] : edge_location_end_idx[i, j],
                    :,
                    :,
                ] = edge_intensity[i, j]
            for j in range(self.num_vertical_edge):
                X[
                    i,
                    :,
                    edge_location_start_idx[i, j] : edge_location_end_idx[i, j],
```

```python
                ] = edge_intensity[i, self.num_horizontal_edge + j]

        return X

    def _generate_diagonal_edge_images(self):
        """
        Generate diagonal edge images by rotating images
        Returns:
            np.ndarray: Generated diagonal edge images
        """
        assert (
            self.num_diagonal_edge > 0
        ), "Number of diagonal edge must be greater than 0"

        X = self._generate_background_images()
        background_intensity = np.mean(X, axis=(1, 2, 3))

        edge_location_start_idx, edge_location_end_idx = self._edge_location(
            edge_type="diagonal"
        )
        edge_intensity = self._edge_intensity(edge_type="diagonal")

        random_angle = np.random.choice(
            [30, 45, 120, 135], size=(self.data_per_class, self.num_diagonal_edge
        )

        for i in range(self.data_per_class):
            for j in range(self.num_diagonal_edge):
                if i % 2 == 0:  # horizontal
                    X[
                        i,
                        edge_location_start_idx[i, j] : edge_location_end_idx[i,
                        :,
                        :,
                    ] = edge_intensity[i, j]
                else:  # vertical
                    X[
                        i,
                        :,
                        edge_location_start_idx[i, j] : edge_location_end_idx[i,
                    ] = edge_intensity[i, j]
                X[i] = rotate(
                    X[i],
                    random_angle[i, j],
                    reshape=False,
                    mode="constant",
                    cval=background_intensity[i],
                )
        return X

    def _generate_background_images(self):
        """
        Generate background images
        Returns:
            np.ndarray: Generated background images
        """
        X = np.ones(
            (self.data_per_class, self.spatial_resolution, self.spatial_resolutic
        )  # NHWC format
        X *= np.random.uniform(
            self.min_background_intensity,
```

```python
            self.max_background_intensity,
            size=(self.data_per_class, 1, 1, 1),
        )
        return X

    def get_image_statistics(self):
        """
        Get image statistics
        Returns:
            tuple: (mean, std)
            mean (float): Mean of the images
            std (float): Standard deviation of the images
        """
        return self._mean, self._std

    def _generate_dataset(self):
        """
        Generate dataset
        Returns:
            tuple: (X, y)
            X (list of PIL Image): Generated images
            y (np.ndarray): Generated labels
        """
        num_data = self.data_per_class * self.num_classes
        self.X = np.zeros(
            (num_data, self.spatial_resolution, self.spatial_resolution, 1)
        )
        self.y = np.zeros((num_data,), dtype=np.int64)
        for i in range(self.num_classes):
            class_type = self.class_type[i]
            if class_type == "horizontal":
                X = self._generate_hoizontal_edge_images()
            elif class_type == "vertical":
                X = self._generate_vertical_edge_images()
            elif class_type == "both":
                X = self._generate_both_edge_images()
            elif class_type == "diagonal":
                X = self._generate_diagonal_edge_images()
            elif class_type == "none":
                X = self._generate_background_images()
            else:
                raise ValueError("Invalid class type")

            assert X.shape == (
                self.data_per_class,
                self.spatial_resolution,
                self.spatial_resolution,
                1,
            )  # NHWC format

            # permute pixels
            if self.use_permutation:
                X = self._permute_pixels(X)

            self.X[i * self.data_per_class : (i + 1) * self.data_per_class] = X
            self.y[i * self.data_per_class : (i + 1) * self.data_per_class] = i

        # Compute mean and std
        self._mean = np.mean(self.X)
        self._std = np.std(self.X)
```

```python
        # np.float32 -> np.uint8
        self.X = (self.X * 255).astype(np.uint8)

        # Convert ndarray to PIL Image
        self.X = [T.functional.to_pil_image(x) for x in self.X]


def count_parameters(model, only_trainable=False):
    if only_trainable:
        return sum(p.numel() for p in model.parameters() if p.requires_grad)
    else:
        return sum(p.numel() for p in model.parameters())

def freeze_conv_layer(model):
    for name, param in model.named_parameters():
        if name.startswith('conv'):
            param.requires_grad = False

def init_conv_kernel_with_edge_detector(model):
    # Get kernel size
    kernel_size = model.conv1.kernel_size[0]

    # number of filters should be 3
    num_filters = model.conv1.out_channels
    assert num_filters == 3, "Number of filters should be 3"

    if kernel_size == 2:
        # 2 x 2 edge detector
        horizontal_edge_detector = torch.tensor([[1, 1], [-1, -1]], dtype=torch.
        vertical_edge_detector = torch.tensor([[1, -1], [1, -1]], dtype=torch.fl
        none_edge_detector = torch.tensor([[0, 0], [0, 0]], dtype=torch.float32)

    else:
        horizontal_edge_detector = torch.from_numpy(custom_sobel((kernel_size, ke
        vertical_edge_detector = torch.from_numpy(custom_sobel((kernel_size, kern
        none_edge_detector = torch.from_numpy(np.zeros((kernel_size, kernel_size)

    edge_detector = torch.stack([horizontal_edge_detector, vertical_edge_detecto
    model.conv1.weight.data = edge_detector.view(model.num_filter, 1, model.kerne
    model.conv2.weight.data = torch.cat([model.conv1.weight.data, model.conv1.we

    # type casting
    model.conv1.weight.data = model.conv1.weight.data.type(torch.FloatTensor)
    model.conv2.weight.data = model.conv2.weight.data.type(torch.FloatTensor)

    # bias
    model.conv1.bias.data = torch.tensor([0, 0, 0], dtype=torch.float32)
    model.conv2.bias.data = torch.tensor([0, 0, 0], dtype=torch.float32)

def custom_sobel(shape, axis):
    """
    shape must be odd: eg. (5,5)
    axis is the direction, with 0 to positive x and 1 to positive y
    """
    k = np.zeros(shape, dtype=np.float32)
    p = [(j,i) for j in range(shape[0])
           for i in range(shape[1])
           if not (i == (shape[1] -1)/2. and j == (shape[0] -1)/2.)]

    for j, i in p:
        j_ = int(j - (shape[0] -1)/2.)
```

```python
        i_ = int(i - (shape[1] -1)/2.)
        k[j,i] = (i_ if axis==0 else j_)/float(i_*i_ + j_*j_)
    return k


def set_seed(seed):
    """
    Set the seed for all random number generators.
    """
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)


def train_one_epoch(
    model,
    optimizer,
    criterion,
    train_loader,
    device,
    epoch,
    log_interval=100,
    verbose=True,
):
    model.train()
    # return the average loss and accuracy
    train_loss = 0
    correct = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        pred = output.argmax(
            dim=1, keepdim=True
        )
        correct += pred.eq(target.view_as(pred)).sum().item()

        if batch_idx % log_interval == 0 and verbose:
            print(
                "Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}".format(
                    epoch,
                    batch_idx * len(data),
                    len(train_loader.dataset),
                    100.0 * batch_idx / len(train_loader),
                    loss.item(),
                )
            )

    train_loss /= len(train_loader.dataset)
    train_accuracy = correct / len(train_loader.dataset)

    return train_loss, train_accuracy
```

```python
def _generate_confusion_matrix(pred_list, target_list):
    pred_list = torch.cat(pred_list)
    target_list = torch.cat(target_list)

    assert pred_list.shape[0] == target_list.shape[0], "predictions and targets

    matrix_size = max(max(pred_list), max(target_list)) + 1
    confusion_matrix = torch.zeros(matrix_size, matrix_size)

    for t, p in zip(target_list.view(-1), pred_list.view(-1)):
        confusion_matrix[t.long(), p.long()] += 1

    return confusion_matrix.cpu().numpy()

def evaluate(model, criterion, valid_loader, device, verbose=True):
    model.eval()
    valid_loss = 0
    correct = 0

    pred_list, target_list = [], []
    confusion_matrix = torch.zeros(4, 4)

    with torch.no_grad():
        for data, target in valid_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            valid_loss += criterion(output, target).item()  # sum up batch loss
            pred = output.argmax(
                dim=1, keepdim=True
            )  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

            pred_list.append(pred)
            target_list.append(target)

    confusion_matrix = _generate_confusion_matrix(pred_list, target_list)

    valid_loss /= len(valid_loader.dataset)
    valid_accuracy = 100.0 * correct / len(valid_loader.dataset)

    if verbose:
        print(
            "Validation Result: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)".
                valid_loss, correct, len(valid_loader.dataset), valid_accuracy
            )
        )

    return valid_loss, valid_accuracy, confusion_matrix


def vis_training_curve(cnn_train_loss, cnn_train_acc, mlp_train_loss, mlp_train_
    # if mlp lists are empty, then we are only plotting the CNN
    if mlp_train_loss is None or len(mlp_train_loss) == 0:
        fig, ax = plt.subplots(1, 2, figsize=(20, 5))
        ax[0].plot(cnn_train_loss, label="CNN")
        ax[0].set_title("Training Loss")
        ax[0].set_xlabel("Epoch")
        ax[0].set_ylabel("Loss")
        ax[0].legend()
        ax[0].grid()
```

```python
        ax[1].plot(cnn_train_acc, label="CNN")
        ax[1].set_title("Training Accuracy")
        ax[1].set_xlabel("Epoch")
        ax[1].set_ylabel("Accuracy")
        ax[1].legend()
        ax[1].grid()

        plt.show()

    # if cnn lists are empty, then we are only plotting the MLP
    elif cnn_train_loss is None or len(cnn_train_loss) == 0:
        fig, ax = plt.subplots(1, 2, figsize=(20, 5))
        ax[0].plot(mlp_train_loss, label=label)
        ax[0].set_title("Training Loss")
        ax[0].set_xlabel("Epoch")
        ax[0].set_ylabel("Loss")
        ax[0].legend()
        ax[0].grid()

        ax[1].plot(mlp_train_acc, label=label)
        ax[1].set_title("Training Accuracy")
        ax[1].set_xlabel("Epoch")
        ax[1].set_ylabel("Accuracy")
        ax[1].legend()
        ax[1].grid()

        plt.show()

    # if both lists are not empty, then we are plotting both CNN and MLP
    else:
        fig, ax = plt.subplots(1, 2, figsize=(20, 5))
        ax[0].plot(cnn_train_loss, label="CNN")
        ax[0].plot(mlp_train_loss, label=label)
        ax[0].set_title("Training Loss")
        ax[0].set_xlabel("Epoch")
        ax[0].set_ylabel("Loss")
        ax[0].legend()
        ax[0].grid()

        ax[1].plot(cnn_train_acc, label="CNN")
        ax[1].plot(mlp_train_acc, label=label)
        ax[1].set_title("Training Accuracy")
        ax[1].set_xlabel("Epoch")
        ax[1].set_ylabel("Accuracy")
        ax[1].legend()
        ax[1].grid()

        plt.show()


def vis_validation_curve(cnn_valid_loss, cnn_valid_acc, mlp_valid_loss, mlp_vali
    # if mlp lists are empty, then we are only plotting the CNN
    if mlp_valid_loss is None or len(mlp_valid_loss) == 0:
        fig, ax = plt.subplots(1, 2, figsize=(20, 5))
        ax[0].plot(cnn_valid_loss, label="CNN")
        ax[0].set_title("Validation Loss")
        ax[0].set_xlabel("Epoch")
        ax[0].set_ylabel("Loss")
        ax[0].legend()
        ax[0].grid()
```

```
        ax[1].plot(cnn_valid_acc, label="CNN")
        ax[1].set_title("Validation Accuracy")
        ax[1].set_xlabel("Epoch")
        ax[1].set_ylabel("Accuracy")
        ax[1].legend()
        ax[1].grid()

        plt.show()

    # if cnn lists are empty, then we are only plotting the MLP
    elif cnn_valid_loss is None or len(cnn_valid_loss) == 0:
        fig, ax = plt.subplots(1, 2, figsize=(20, 5))
        ax[0].plot(mlp_valid_loss, label=label)
        ax[0].set_title("Validation Loss")
        ax[0].set_xlabel("Epoch")
        ax[0].set_ylabel("Loss")
        ax[0].legend()
        ax[0].grid()

        ax[1].plot(mlp_valid_acc, label=label)
        ax[1].set_title("Validation Accuracy")
        ax[1].set_xlabel("Epoch")
        ax[1].set_ylabel("Accuracy")
        ax[1].legend()
        ax[1].grid()

        plt.show()

    # if both lists are not empty, then we are plotting both CNN and MLP
    else:
        fig, ax = plt.subplots(1, 2, figsize=(20, 5))
        ax[0].plot(cnn_valid_loss, label="CNN")
        ax[0].plot(mlp_valid_loss, label=label)
        ax[0].set_title("Validation Loss")
        ax[0].set_xlabel("Epoch")
        ax[0].set_ylabel("Loss")
        ax[0].legend()
        ax[0].grid()

        ax[1].plot(cnn_valid_acc, label="CNN")
        ax[1].plot(mlp_valid_acc, label=label)
        ax[1].set_title("Validation Accuracy")
        ax[1].set_xlabel("Epoch")
        ax[1].set_ylabel("Accuracy")
        ax[1].legend()
        ax[1].grid()

        plt.show()


def vis_kernel(tensor, ch=0, allkernels=False, nrow=8, padding=1, title=None,
    n, c, h, w = tensor.shape

    if allkernels:
        tensor = tensor.view(n * c, -1, h, w)
    elif c != 3:
        tensor = tensor[:, ch, :, :].unsqueeze(dim=1)

    rows = np.min((tensor.shape[0] // nrow + 1, 64))
    grid = (
        torchvision.utils.make_grid(tensor, nrow=nrow, normalize=True, padding=p
```

```
        .numpy()
        .transpose((1, 2, 0))
    )
    plt.figure(figsize=(nrow, rows))
    plt.imshow(grid, cmap=cmap)
    plt.colorbar(cmap=cmap)
    if title is not None:
        plt.title(title)
    plt.axis("off")
    plt.ioff()
    plt.show()


def vis_confusion_matrix(confusion_matrix, class_names=None, title=None):
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    cax = ax.matshow(confusion_matrix, cmap=plt.cm.Blues)
    fig.colorbar(cax)

    matrix_size = confusion_matrix.shape[0]

    if class_names is not None:
        assert len(class_names) == matrix_size, "Class names must be same lengtl
        ax.set_xticklabels([""] + class_names, rotation=90)
        ax.set_yticklabels([""] + class_names)

    ax.set_xlabel("Predicted")
    ax.set_ylabel("True")
    ax.xaxis.set_label_position("top")
    ax.xaxis.tick_top()
    ax.set_title(title)

    for (i, j), z in np.ndenumerate(confusion_matrix):
        ax.text(j, i, "{:0.1f}".format(z), ha="center", va="center")


def vis_unpermuted_dataset(dataset, num_classes, num_show_per_class, unpermutato
    f, axarr = plt.subplots(num_classes, num_show_per_class, figsize=(20, 2*num_

    for i in range(num_classes):
        for j in range(num_show_per_class):
            img = dataset[i * num_show_per_class + j][0]
            label = dataset[i * num_show_per_class + j][1]

            if isinstance(img, torch.Tensor):
                img = img.numpy().transpose((1, 2, 0))
                h, w, c = img.shape
                assert c == 1
                img = img.reshape((h * w, c))
                img = img[unpermutator, :]
                img = img.reshape(h, w)
                axarr[i, j].imshow(img, cmap="gray", vmin=0, vmax=1)

            elif isinstance(img, Image.Image):
                img = np.array(img)
                h, w = img.shape
                img = img.reshape(h*w)
                img = img[unpermutator]
                img = img.reshape(h, w)
                img = T.functional.to_pil_image(img)
                axarr[i, j].imshow(img, cmap="gray", vmin=0, vmax=255)
```

```
            axarr[i, j].axis("off")
            axarr[i, j].set_title('Class: {}'.format(label))


def vis_dataset(dataset, num_classes=3, num_show_per_class=10):
    f, axarr = plt.subplots(num_classes, num_show_per_class, figsize=(20, 2*num_

    for i in range(num_classes):
        for j in range(num_show_per_class):
            img = dataset[i * num_show_per_class + j][0]
            label = dataset[i * num_show_per_class + j][1]

            if isinstance(img, torch.Tensor):
                img = img.numpy().transpose((1, 2, 0))
                img = img.squeeze()
                axarr[i, j].imshow(img, cmap="gray", vmin=0, vmax=1)
            elif isinstance(img, Image.Image):
                axarr[i, j].imshow(img, cmap="gray", vmin=0, vmax=255)
            axarr[i, j].axis("off")
            axarr[i, j].set_title('Class: {}'.format(label))


class WiderCNN(nn.Module):
    def __init__(self, input_channel=1, num_filters=6, kernel_size=7, num_classe
        super(WiderCNN, self).__init__()
        padding = (kernel_size - 1) // 2
        self.conv1 = nn.Conv2d(input_channel, num_filters, kernel_size=kernel_si
        self.conv2 = nn.Conv2d(num_filters, num_filters, kernel_size=kernel_size,
        self.maxpool = nn.MaxPool2d(2, 2)
        self.fc = nn.Linear(num_filters, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.maxpool(x)
        x = F.relu(self.conv2(x))
        x = F.adaptive_avg_pool2d(x, (1, 1)).squeeze()
        x = self.fc(x)

        return x

class DeeperCNN(nn.Module):
    def __init__(self, input_channel=1, num_filters=3, kernel_size=7, num_classe
        super().__init__()
        padding = (kernel_size - 1) // 2
        self.conv1 = nn.Conv2d(input_channel, num_filters, kernel_size=kernel_si
        self.conv2 = nn.Conv2d(num_filters, num_filters, kernel_size=kernel_size,
        self.conv3 = nn.Conv2d(num_filters, num_filters, kernel_size=kernel_size,
        self.conv4 = nn.Conv2d(num_filters, num_filters, kernel_size=kernel_size,
        self.maxpool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(num_filters, num_classes)

        self.num_filters = num_filters

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.maxpool(x)
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        # x = self.maxpool(x)
        x = F.relu(self.conv4(x))
```

```python
        x = F.adaptive_avg_pool2d(x, (1, 1)).squeeze()
        x = self.fc1(x)

        return x

class SimpleCNN(nn.Module):
    def __init__(self, num_filters=3, kernel_size=2, num_classes=3):
        super().__init__()
        padding = (kernel_size - 1) // 2
        self.conv1 = nn.Conv2d(1, num_filters, kernel_size, padding=padding, padd
        self.conv2 = nn.Conv2d(num_filters, num_filters, kernel_size, padding=pa
        self.maxpool = nn.MaxPool2d(2, 2)
        self.fc = nn.Linear(num_filters, num_classes)
        self.init_weights()

        self.num_filter = num_filters
        self.kernel_size = kernel_size

    def init_weights(self):
        # if not self.edge_detector_init:
        nn.init.xavier_uniform_(self.conv1.weight)
        nn.init.xavier_uniform_(self.fc.weight)

        # bias
        nn.init.zeros_(self.conv1.bias)
        nn.init.zeros_(self.fc.bias)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.adaptive_avg_pool2d(x, (1, 1)).squeeze()
        x = self.fc(x)

        return x

    def get_features(self, x):
        feat_list = []
        x = self.conv1(x)
        feat_list.append(x)
        x = F.relu(x)
        feat_list.append(x)
        x = self.maxpool(x)
        feat_list.append(x)
        x = self.conv2(x)
        feat_list.append(x)
        x = F.relu(x)
        feat_list.append(x)
        x = F.adaptive_avg_pool2d(x, (1, 1)).squeeze()
        feat_list.append(x)

        return feat_list

class SimpleCNN_avgpool(nn.Module):
    def __init__(self, num_filters=3, kernel_size=2, num_classes=3):
        super().__init__()
        padding = (kernel_size - 1) // 2
        self.conv1 = nn.Conv2d(1, num_filters, kernel_size, padding=padding, padd
        self.conv2 = nn.Conv2d(num_filters, num_filters, kernel_size, padding=pa
```

```python
        self.avgpool = nn.AvgPool2d(2, 2)
        self.fc = nn.Linear(num_filters, num_classes)
        self.init_weights()

        self.num_filter = num_filters
        self.kernel_size = kernel_size

    def init_weights(self):
        # if not self.edge_detector_init:
        nn.init.xavier_uniform_(self.conv1.weight)
        nn.init.xavier_uniform_(self.fc.weight)

        # bias
        nn.init.zeros_(self.conv1.bias)
        nn.init.zeros_(self.fc.bias)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.avgpool(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.adaptive_avg_pool2d(x, (1, 1)).squeeze()
        x = self.fc(x)

        return x

    def get_features(self, x):
        feat_list = []
        x = self.conv1(x)
        feat_list.append(x)
        x = F.relu(x)
        feat_list.append(x)
        x = self.avgpool(x)
        feat_list.append(x)
        x = self.conv2(x)
        feat_list.append(x)
        x = F.relu(x)
        feat_list.append(x)
        x = F.adaptive_avg_pool2d(x, (1, 1)).squeeze()
        feat_list.append(x)

        return feat_list

class ThreeLayerCNN(nn.Module):
    def __init__(
        self,
        input_dim=(1, 28, 28),
        num_filters=64, #make it explicit
        filter_size=7,
        hidden_dim=100,
        num_classes=4,
    ):
        """
        A three-layer convolutional network with the following architecture:
        conv - relu - 2x2 max pool - affine - relu - affine - softmax
        The network operates on minibatches of data that have shape (N, C, H, W)
        consisting of N images, each with height H and width W and with C input
        channels.
        Args:
            kernel_size (int): Size of the convolutional kernel
```

```
            channel_size (int): Number of channels in the convolutional layer
            linear_layer_input_dim (int): Number of input features to the linear
            output_dim (int): Number of output features
        """
        super(ThreeLayerCNN, self).__init__()
        C, H, W = input_dim

        self.conv1 = nn.Conv2d(
            C, num_filters, filter_size, stride=1, padding=(filter_size - 1) //
        )
        self.max_pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(
            num_filters, num_filters * 2, filter_size, padding=(filter_size - 1)
        )
        self.fc1 = nn.Linear(num_filters * 2, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        # print(x.shape)
        # x = F.max_pool2d(x, 2)
        # print(x.shape)
        x = F.relu(self.conv2(x))
        # print(x.shape)
        # x = F.max_pool2d(x, 2)
        # print(x.shape)
        x = F.adaptive_avg_pool2d(x, (1, 1)).squeeze()
        x = self.fc1(x)
        return x


class TwoLayerMLP(nn.Module):
    def __init__(self, input_dim=(1, 28, 28), hidden_dim=10, num_classes=3):
        super(TwoLayerMLP, self).__init__()
        C, H, W = input_dim
        self.fc1 = nn.Linear(C * H * W, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class ThreeLayerMLP(nn.Module):
    def __init__(self, input_dim=(1, 28, 28), hidden_dims=[10, 10], num_classes=
        """
        A three-layer fully-connected neural network with ReLU nonlinearity
        """
        super(ThreeLayerMLP, self).__init__()
        torch.manual_seed(seed)
        C, H, W = input_dim
        self.fc1 = nn.Linear(C * H * W, hidden_dims[0])
        self.fc2 = nn.Linear(hidden_dims[0], hidden_dims[1])
        self.fc3 = nn.Linear(hidden_dims[1], num_classes)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
```

```
        return x
```

In [4]:
```
seed = 7
set_seed(seed)

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Generate Dataset

What would be an excellent dataset to study the inductive bias of CNN? First, have to start with the problem as simple as possible. The complex problem makes it hard to understand the underlying mechanism and is challenging to debug in experimental settings. Hence, we choose the edge detection task as an example to study the inductive bias of CNN. Because

1. Edge detection is a straightforward task,
2. It is easy to generate the dataset,
3. The edge of the image is a very fundamental low-level feature useful to every computer vision task such as object detection and finally,
4. Edge detection is an excellent example of studying the inductive bias of CNN.

We will generate the dataset for this toy problem. The dataset consists of 10 images of size 28x28 per class, which are all grey scales. Each image contains a vertical edge, a horizontal edge, or nothing. The labels are 0 for vertical edges, 1 for horizontal edges, and 2 for nothing.

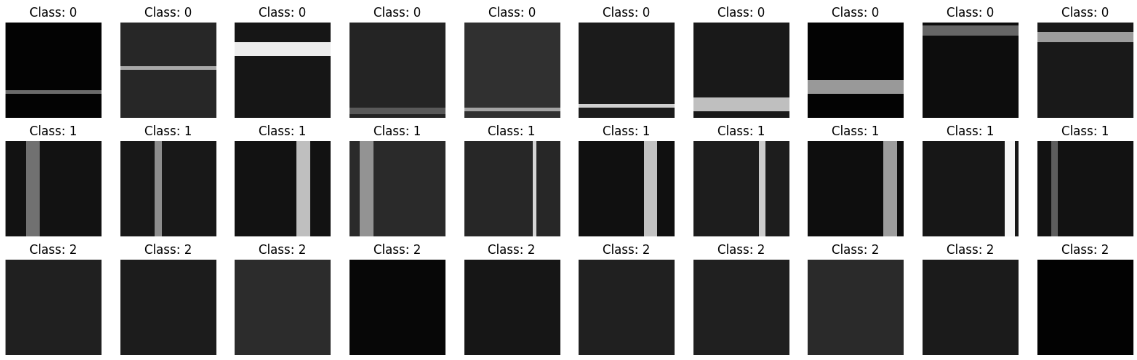
`EdgeDetectionDataset` class is a dataset class that generates and loads the dataset. The dataset inherits `torch.utils.data.Dataset` , and it generates data when it is initialized. This class takes two arguments: `domain_config` and `transform`. `domain_config` is a dictionary that specifies the domain information of train/valid dataset, such as the number of images per class and the size of the image. `transform` is a function that transforms the image. In this homework, we will use `torchvision.transforms.ToTensor()` to convert the image to a tensor.

We highly recommend you read the implementation of `EdgeDetectionDataset` class in `dataset/edge_detection_dataset.py` to understand how the dataset is generated.

In [5]:
```
# Define the domain configuration of the dataset
set_seed(seed)

visualize_data_config = dict(
    data_per_class=10,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
)

visualize_dataset = EdgeDetectionDataset(visualize_data_config, mode='train', transf
```

Visualize Dataset

In [6]:
```
vis_dataset(visualize_dataset, num_classes=3, num_show_per_class=10)
```



Q1. Overfitting Models to Small Dataset

In this problem, we will make our models overfit the small dataset to test the model architecture and our synthetic dataset. We use the same dataset for both models. Let's generate a small dataset with ten images per class.

In [7]:
```
set_seed(seed)

small_dataset_config = None
small_dataset = None
transforms = T.Compose([T.ToTensor()])

small_dataset_config = dict(
    data_per_class=10,
)
transforms = T.Compose([T.ToTensor()])
small_train_dataset = EdgeDetectionDataset(small_dataset_config, 'train', transform=
```

In this notebook, we will use pytorch dataloader to load the dataset. We will use `torch.utils.data.DataLoader` to load the dataset. `DataLoader` takes two arguments: `dataset` and `batch_size`. `dataset` is the dataset that we want to load. Note that `batch_size` is one of important hyperparameters. We will use `batch_size=32` for this problem.

In [10]:
```
small_dataset_loader = None

###############################################################################
# TODO: Implement dataloader                                                  #
# Hint: You should flag shuffle = True for training data loader               #
# This flag makes huge difference in training                                 #
###############################################################################

batch_size = 32
small_dataset_loader = torch.utils.data.DataLoader(small_train_dataset, batch_size=b
###############################################################################
#                             END OF YOUR CODE                                #
###############################################################################
```

Model Architecture

MLP has two hidden layer with 10 hidden units and 10 hidden units. The input size is 28x28=784 and the output size is 3. We use ReLU as the activation function. We use cross entropy loss as the loss function.

MLP architecture: FC(784, 10) -> ReLU -> FC(10, 10) -> ReLU -> FC(10, 3)

CNN has two convolutional layers followed by global average pooling and one fully connected layer. Both convolutional layers have 3 filters whose kernel size is 7. We use ReLU as the activation function. We use cross entropy loss as the loss function.

CNN arhitecture is as follows: CONV - RELU - MAXPOOL - CONV - RELU - MAXPOOL - FC

Fitting on Small Dataset

Now let's train the model on the small dataset. The final tranining loss should be around 100% for both models.

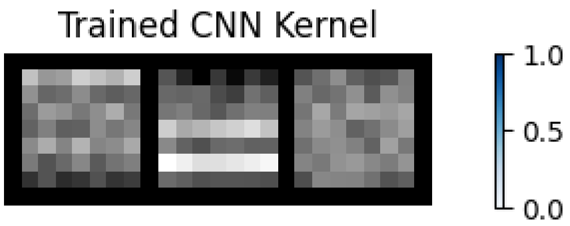
In [11]:
```
set_seed(seed)

lr = 0.01
num_epochs = 500
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

cnn_model = SimpleCNN(kernel_size=7)
cnn_model.to(device)
untrained_cnn_model = deepcopy(cnn_model)

mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
mlp_model.to(device)

mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

criterion = nn.CrossEntropyLoss()
print("CNN Model has {} parameters".format(count_parameters(cnn_model, only_trainabl
print("MLP Model has {} parameters".format(count_parameters(mlp_model, only_trainabl

for epoch in tqdm(range(num_epochs)):
    train_one_epoch(cnn_model, cnn_optimizer, criterion, small_dataset_loader, devic
    train_one_epoch(mlp_model, mlp_optimizer, criterion, small_dataset_loader, devic

    _, cnn_acc, _ = evaluate(cnn_model, criterion, small_dataset_loader, device, ver
    _, mlp_acc, _ = evaluate(mlp_model, criterion, small_dataset_loader, device, ver

print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
```

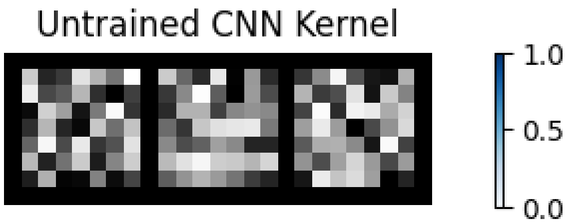
```
CNN Model has 606 parameters
MLP Model has 39793 parameters

100%|██████████| 500/500 [00:10<00:00, 47.89it/s]

CNN Acc: 100.0, MLP Acc: 100.0
```

We checked that both models can overfit the small dataset. This is one of the most important sanity check. If the model cannot overfit the small dataset, the model is not powerful enough to learn the dataset. In this case, we need to increase the size of the model.

Visualize Learned Filters

In [12]:
```
cnn_kernel = cnn_model.conv1.weight.data.clone().cpu()
untrained_kernel = untrained_cnn_model.conv1.weight.data.clone().cpu()

vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel')
vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN Kernel')
```





Question

Can you find any interesting patterns in the learned filters? Answer this question in your submission of the written assignment.

Q2. Sweeping the Number of Training Images

We understood the given task and checked that both models had enough expressive power. We will compare the performance of MLP and CNN by changing the number of data per class. We expect that the model with proper inductive biases on this task will fit with **fewer training examples**. And let's see which one has inductive biases. In this problem, we will use the same dataset for both models. We sweep the number of training images from 10 to 50. The validation set will be the same for all the experiments.

In [13]:

```
set_seed(seed)

train_loader_dict = dict()
num_images_list = [10, 20, 30, 40, 50]
valid_loader = None

transforms = T.Compose([T.ToTensor()])
train_batch_size = 10
valid_batch_size = 256
#############################################################################
# TODO: Implement train_loader_dict for each number of training images.     #
# Key: The number of training images (10, 50, 100, and 500)                 #
# Value: The corresponding dataloader                                       #
# The validation set size is 50 images per class                            #
#############################################################################
for num_image in num_images_list:
    train_dataset = EdgeDetectionDataset(dict(data_per_class=num_image,), "train", t
    train_loader_dict[num_image] = torch.utils.data.DataLoader(train_dataset, batch_

valid_dataset = EdgeDetectionDataset(dict(data_per_class=50,), "valid", transform=tr
valid_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=valid_batch_siz
#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################
```

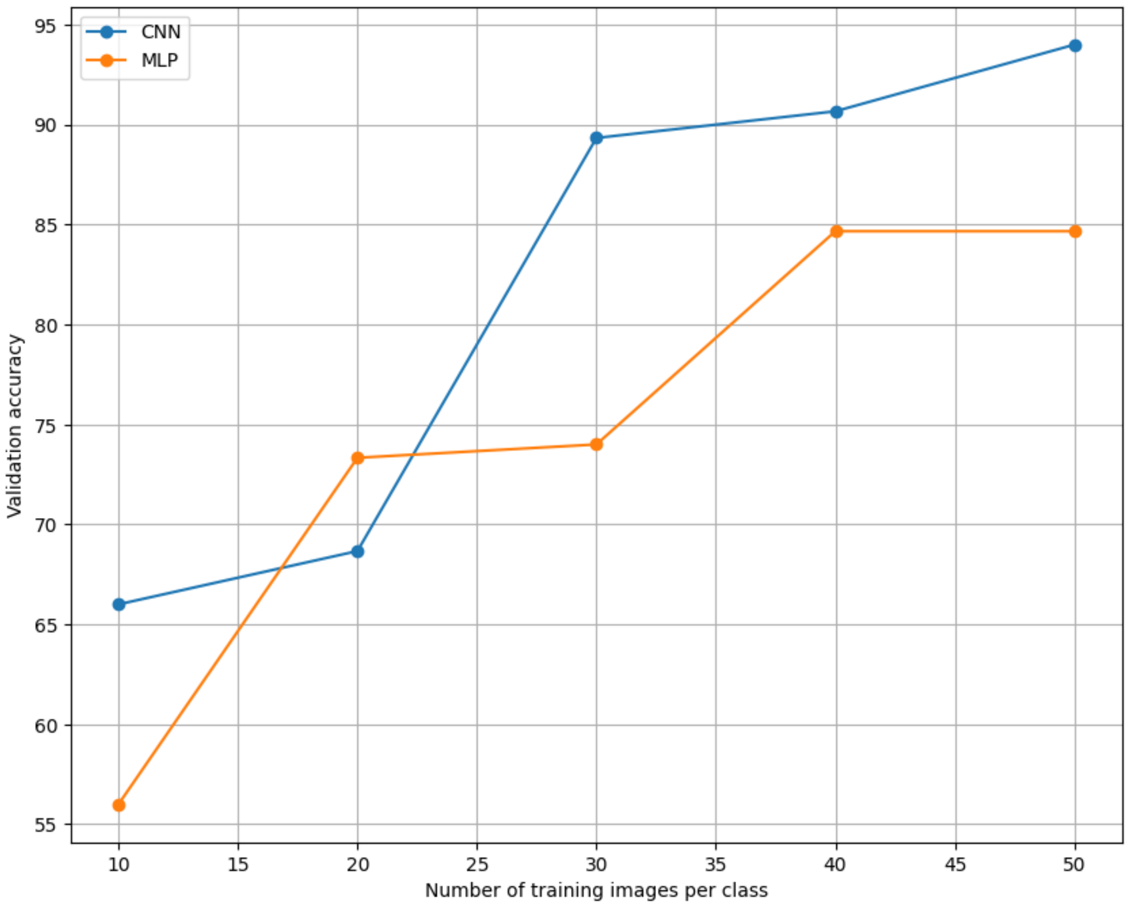
In [14]:
```
lr = 5e-3
num_epochs = 100
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_valid_acc_list = []
    mlp_valid_acc_list = []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model, mlp_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, _ = evaluate(cnn_model, criterion, valid_load
        mlp_valid_loss, mlp_valid_acc, _ = evaluate(mlp_model, criterion, valid_load

        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)

    cnn_kernel_dict[num_image] = deepcopy(cnn_model.conv1.weight.cpu().detach())
    untrained_cnn_kernel_dict[num_image] = deepcopy(untrained_cnn_model.conv1.weight

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
    mlp_acc_list.append(mlp_acc)
```

```
Training with 10 images

100%|██████████| 100/100 [00:07<00:00, 14.09it/s]

CNN Acc: 66.0, MLP Acc: 56.0
Training with 20 images

100%|██████████| 100/100 [00:08<00:00, 12.27it/s]

CNN Acc: 68.66666666666667, MLP Acc: 73.33333333333333
Training with 30 images

100%|██████████| 100/100 [00:09<00:00, 10.86it/s]
```

```
CNN Acc: 89.33333333333333, MLP Acc: 74.0
Training with 40 images

100%|██████████| 100/100 [00:10<00:00,  9.89it/s]

CNN Acc: 90.66666666666667, MLP Acc: 84.66666666666667
Training with 50 images

100%|██████████| 100/100 [00:11<00:00,  8.83it/s]

CNN Acc: 94.0, MLP Acc: 84.66666666666667
```

In [15]:

```
## Plot the validation accuracy
plt.plot(num_images_list, cnn_acc_list, marker='o', label='CNN')
plt.plot(num_images_list, mlp_acc_list, marker='o', label='MLP')
plt.xlabel('Number of training images per class')
plt.ylabel('Validation accuracy')
plt.legend()
plt.grid()
plt.show()
```



OK, in most cases, CNN looks like it is performing better than MLP. So can we conclude that CNN has the inductive biases of locality and translational invariance? Not yet. We need to conduct a series of other experiments to show that CNN has such inductive biases.

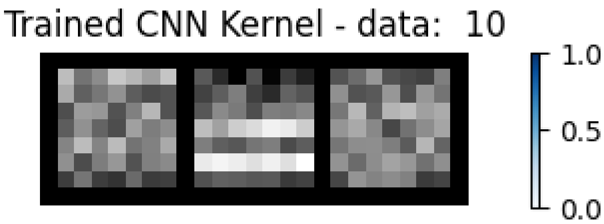
Seemingly, the experiment result is odd. First, the performance of the low data regime `num_train_images_per_class=10` is very bad, considering the task is straightforward. Second, some students will observe that the performance of MLP is better than CNN at some point. At least, CNN should be much better even in a small data regime if it is translational equivariant. How do we debug the model? We will study how to debug the model in the

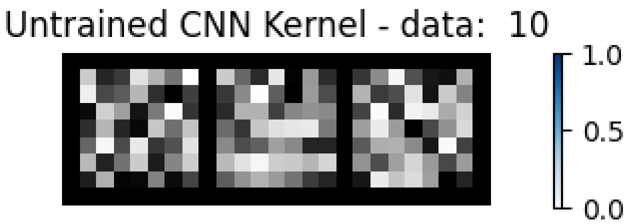
Here are some checklists that you can do to debug the problem.

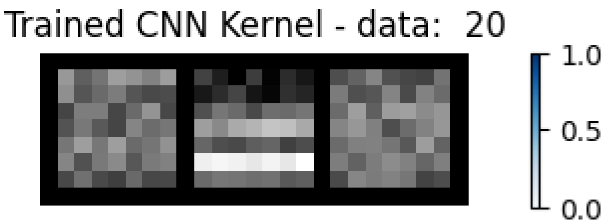
1. Did you check the dataset? For example, is the dataset balanced? Is the dataset noisy? Is the dataset too small?
2. Did you check the model architecture? For example, is the model architecture powerful enough to learn the dataset? Is the model architecture too complex? Is the model architecture too simple?
3. Did you check the model initialization? For example, is the model initialized properly? Is the model initialized randomly? Is the model initialized with the pre-trained weights?
4. Did you check that the model is trained correctly? For example, does the kernel look like an edge detector? What would be the performance of CNN if kernels were initialized with edge detectors?
5. Did you check the training procedure? For example, is the training procedure correct? Is the training procedure stable? Is the training procedure too slow?
6. Did you optimize the hyperparameters? For example, learning rate, batch size, and the number of epochs.

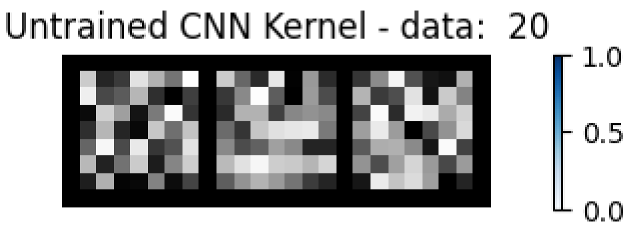
Note that we already checked the dataset, initialization, and model architecture. But we didn't check the step after 3. Let's step 4 first. We will first see what the learned weights look like, initialize the kernels with edge detectors, and see what happens.

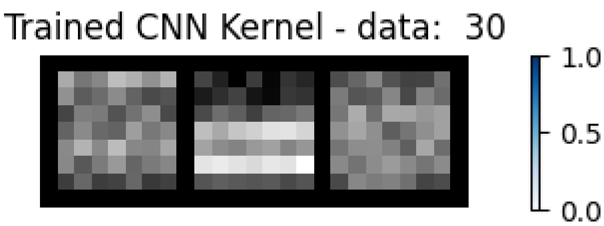
In [16]:
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel - data
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN Kerne
```

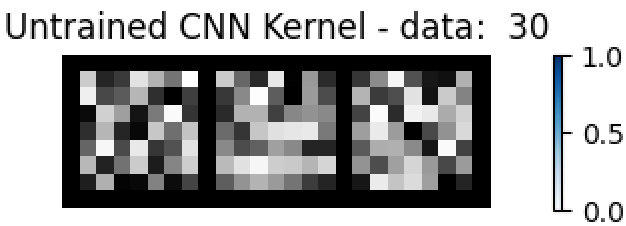


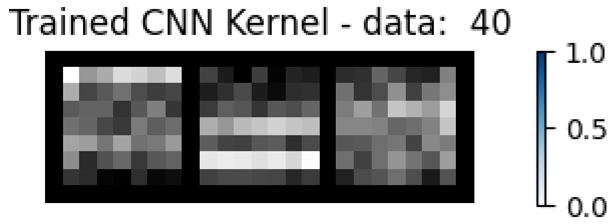


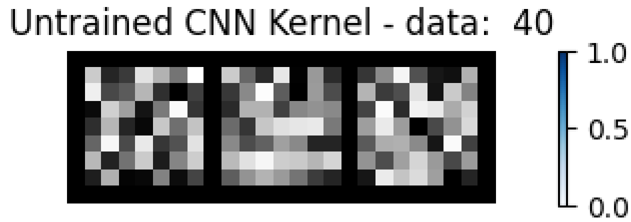


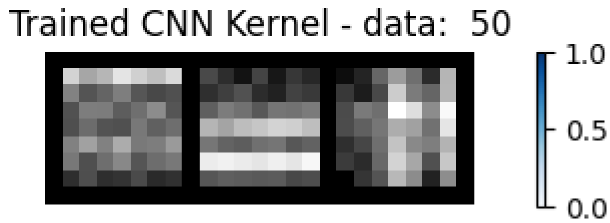


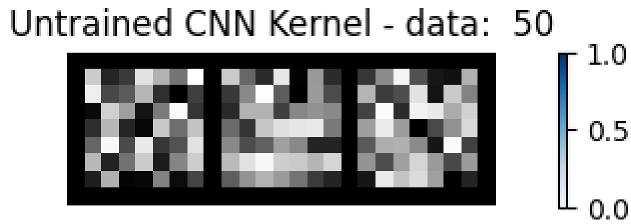












Question

Compare the learned kernels, untrainable kernels, and edge-detector kernels. What do you observe? Answer this question in your submission of the written assignment.

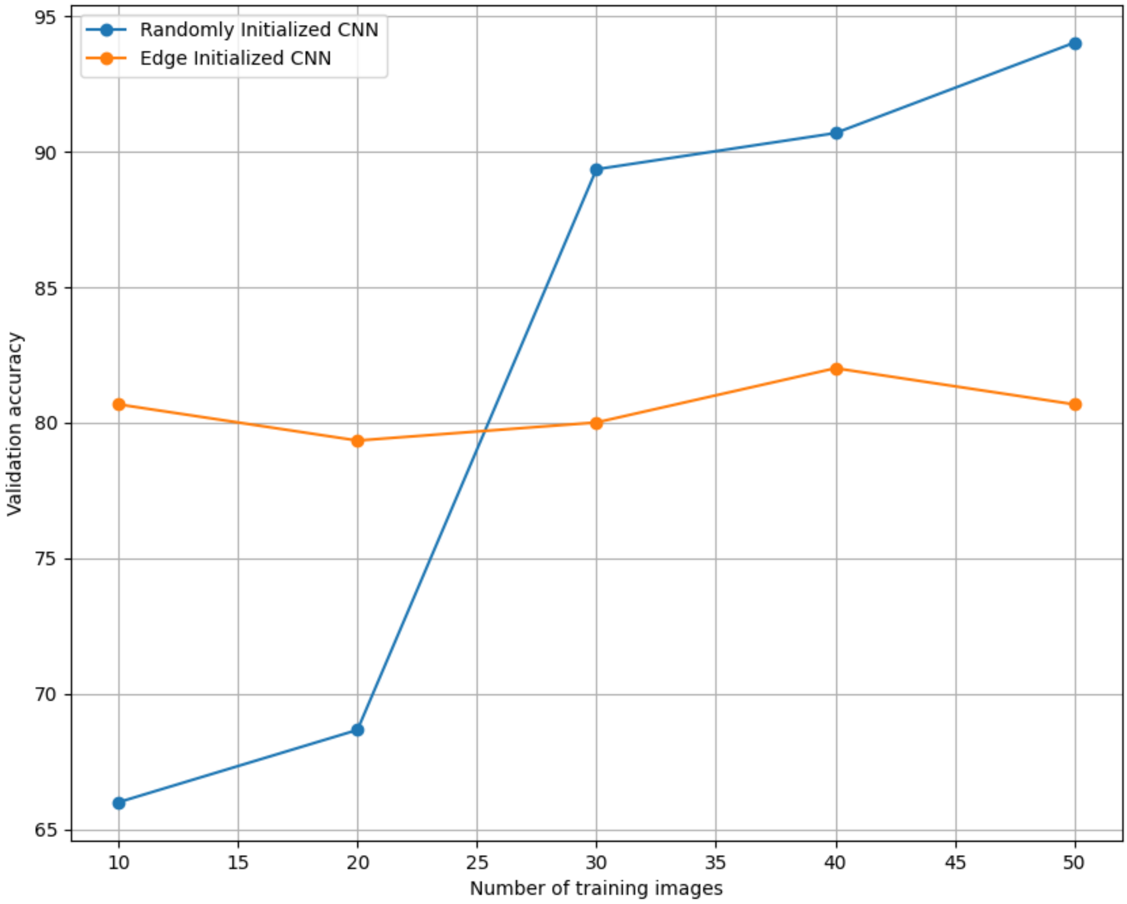
Visualized kernels seem very odd. Some kernels look randomly generated. Think about the data generating process. The factor determining this dataset is the edge location, edge width, and the intensities of background and edges. Therefore, we might be able to get kernels that look like edge detectors. Then, the next logical question should be, what if kernels are initialized with edge detectors? How would the performance change? Because we inject the additional inductive biases into the model. We expect the validation accuracy to be much better and with fewer training examples. Let's try it.

Injecting Inductive Bias: Initialize Kernels with Edge Detectors

In [17]:
```
lr = 0.05
num_epochs = 100
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

edge_init_cnn_acc_list = list()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    cnn_model = SimpleCNN(kernel_size=2)
    init_conv_kernel_with_edge_detector(cnn_model)
    freeze_conv_layer(cnn_model)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)


    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    edge_init_cnn_valid_acc_list = []
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr
        cnn_valid_loss, cnn_valid_acc, _ = evaluate(cnn_model, criterion, valid_loac
        edge_init_cnn_valid_acc_list.append(cnn_valid_acc)

    cnn_acc = edge_init_cnn_valid_acc_list[-1]

    print("CNN Acc: {}".format(cnn_acc))
    edge_init_cnn_acc_list.append(cnn_acc)
```

```
Training with 10 images

100%|██████████| 100/100 [00:03<00:00, 27.97it/s]

CNN Acc: 80.66666666666667
Training with 20 images

100%|██████████| 100/100 [00:03<00:00, 25.55it/s]

CNN Acc: 79.33333333333333
Training with 30 images

100%|██████████| 100/100 [00:04<00:00, 23.75it/s]

CNN Acc: 80.0
Training with 40 images

100%|██████████| 100/100 [00:04<00:00, 21.57it/s]

CNN Acc: 82.0
Training with 50 images

100%|██████████| 100/100 [00:05<00:00, 19.37it/s]

CNN Acc: 80.66666666666667
```

In [18]:
```
## Plot the validation accuracy
plt.plot(num_images_list, cnn_acc_list, marker='o', label='Randomly Initialized CNN'
plt.plot(num_images_list, edge_init_cnn_acc_list, marker='o', label='Edge Initialize
plt.xlabel('Number of training images')
plt.ylabel('Validation accuracy')
plt.legend()
plt.grid()
plt.show()
```



As you can see in the above graph, the performance of CNN initialized with edge detectors is much better than CNN initialized with random weights. It is a significant observation, especially in a low data regime. Now we have to check the training procedure.

Question

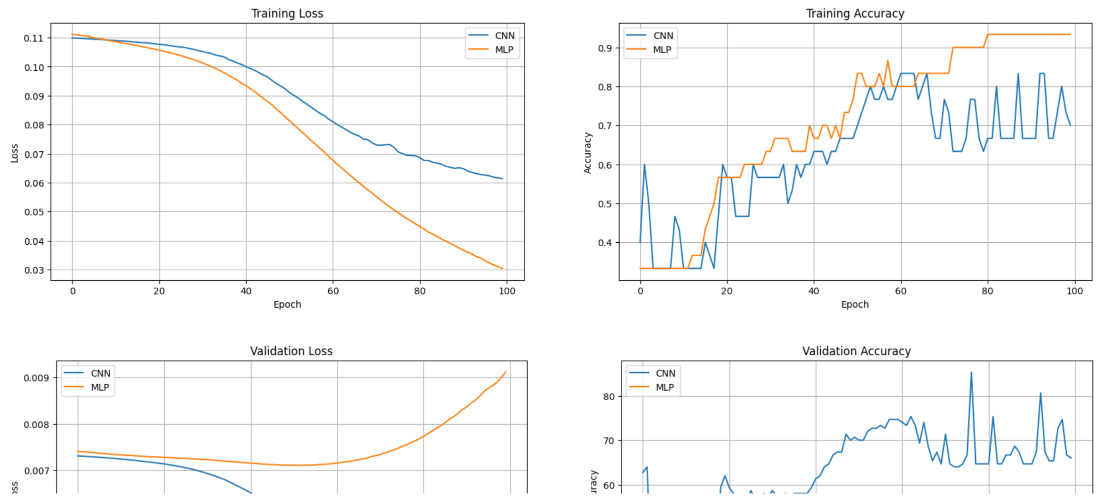
We freeze the convolutional layer and train only final layer (classifier) in this experiment. For a high data regime, the performance of CNN initialized with edge detectors is worse than CNN initialized with random weights. **Why do you think this happens?** Answer this question in your submission of the written assignment.

Q3. Checking the Training Procedure

Checking the training procedure is very important. We must log at least training loss, training accuracy, validation loss, and validation accuracy. Let's log such training signals and find out what is going on.

In [19]:
```
lr = 5e-3
num_epochs = 100
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list, mlp_valid_loss_list
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model, mlp_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, _ = evaluate(cnn_model, criterion, valid_loac
        mlp_valid_loss, mlp_valid_acc, _ = evaluate(mlp_model, criterion, valid_loac

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, mlp_train_loss_list,
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, mlp_valid_loss_lis

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.data.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.data.det

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
    mlp_acc_list.append(mlp_acc)
```

```
Training with 10 images

100%|██████████| 100/100 [00:07<00:00, 13.86it/s]
```



What is going on here? Validation loss and validation accuracy are not flat at the end. It means that the model is not converged. We need to train the model more. Let's train the model with the higher number of epochs. Increase the number of epochs until the validation loss and accuracy are flat.

Question

List every epochs that you trained the model. Final accuracy of CNN should be at least 80% for 50 images per class. Answer this question in your submission of the written assignment.

Question

Check the learned kernels. What do you observe? Answer this question in your submission of the written assignment.

Question (Optional)

You might find that with the high number of epochs, validation loss of MLP is increasing whild validation accuracy increasing. **How can we interpret this?** Answer this question in your submission of the written assignment.

(Hint: Refer to this [paper (https://arxiv.org/pdf/1706.04599.pdf%5D)](https://arxiv.org/pdf/1706.04599.pdf%5D))

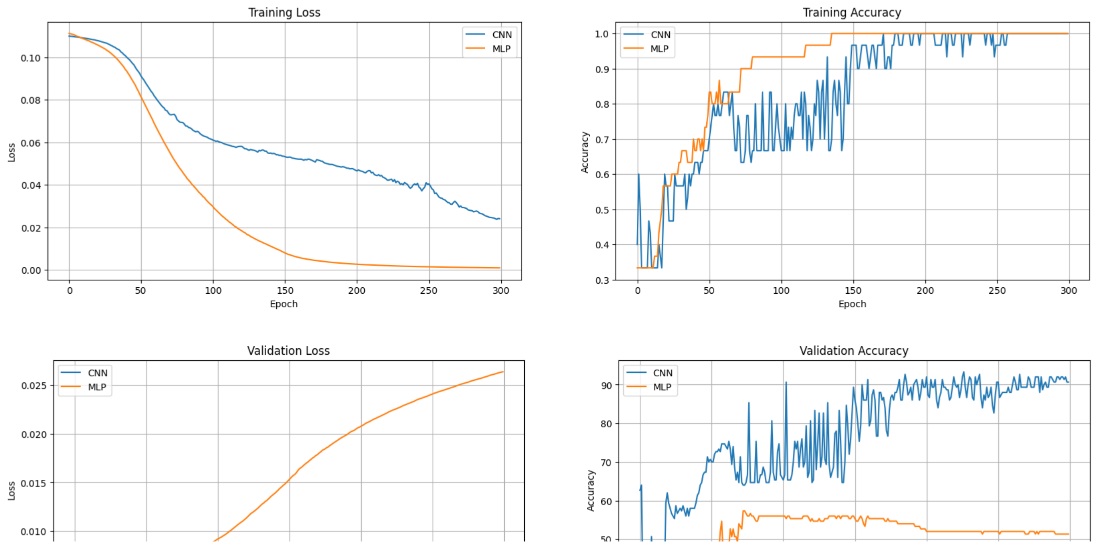
Question (Optional)

Do hyperparameter tuning. **And list the best hyperparameter setting that you found and report the final accuracy of CNN and MLP.** Answer this question in your submission of the written assignment.

In [21]:
```
###############################################################################
# TODO: Try other num_epochs. Final accuracy of CNN should be at least       #
# 90% for 10 images per class.                                                #
###############################################################################
num_epochs = 300  # Good starting point: 100
###############################################################################
#                              END OF YOUR CODE                               #
###############################################################################
lr = 5e-3
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list, mlp_valid_loss_list
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model, mlp_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, _ = evaluate(cnn_model, criterion, valid_load
        mlp_valid_loss, mlp_valid_acc, _ = evaluate(mlp_model, criterion, valid_load

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, mlp_train_loss_list,
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, mlp_valid_loss_lis

    cnn_kernel_dict[num_image] = deepcopy(cnn_model.conv1.weight.data.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = deepcopy(untrained_cnn_model.conv1.weight

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
```

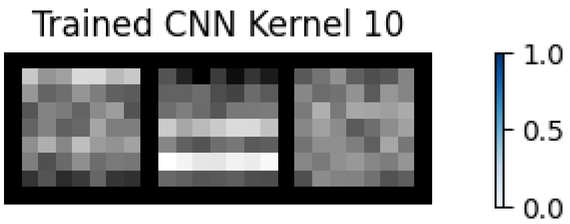
```
    mlp_acc_list.append(mlp_acc)
```

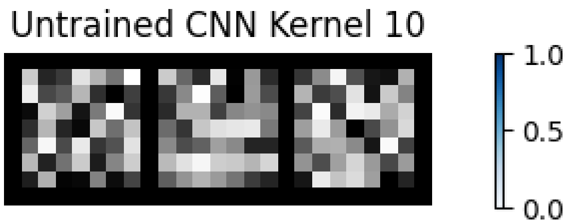
```
Training with 10 images

100%|██████████| 300/300 [00:21<00:00, 14.03it/s]
```

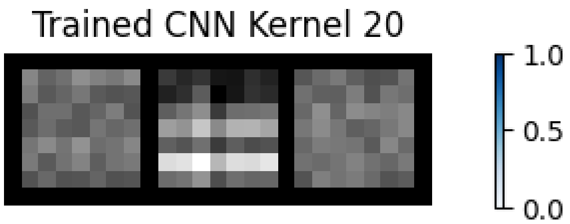


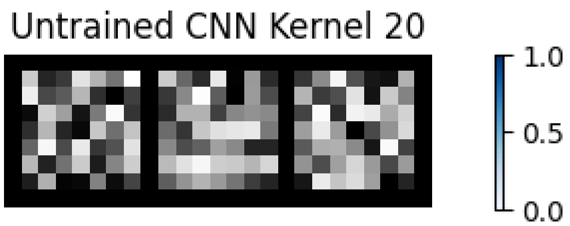
In [22]:

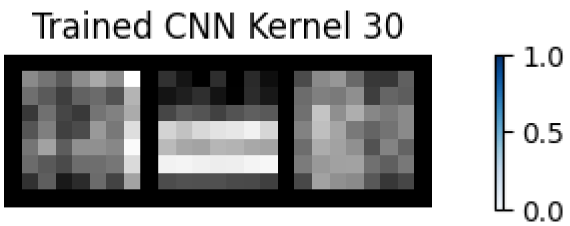
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel {}'.fo
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN Kerne
```

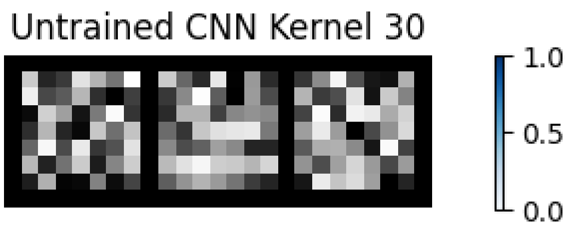


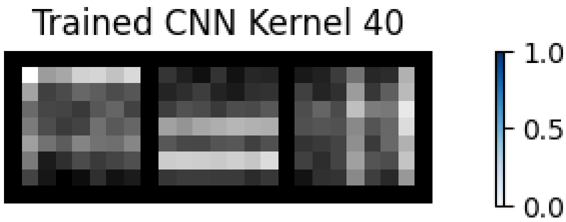


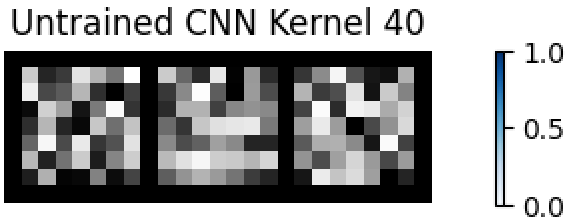


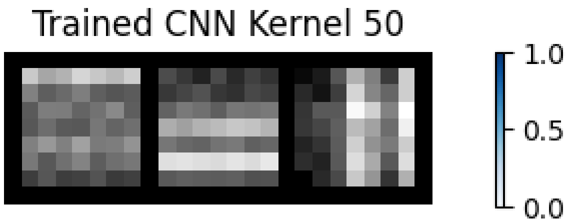


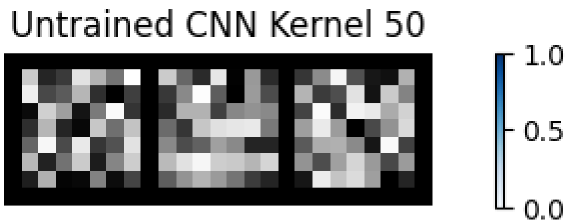












Question

How much more data is needed for MLP to get a competitive performance with CNN? Does MLP really generalize or memorize? Answer this question in your submission of the written assignment.

Q4. Domain Shift between Training and Validation Set

In this problem, we will see how the model performance changes when the domain of the training set and that of the validation set are different. We will generate training set images with edges that locate only half of the image and validation set images with edges that locate only the other half of the image. Let's repeat the same experiment as the previous problem.

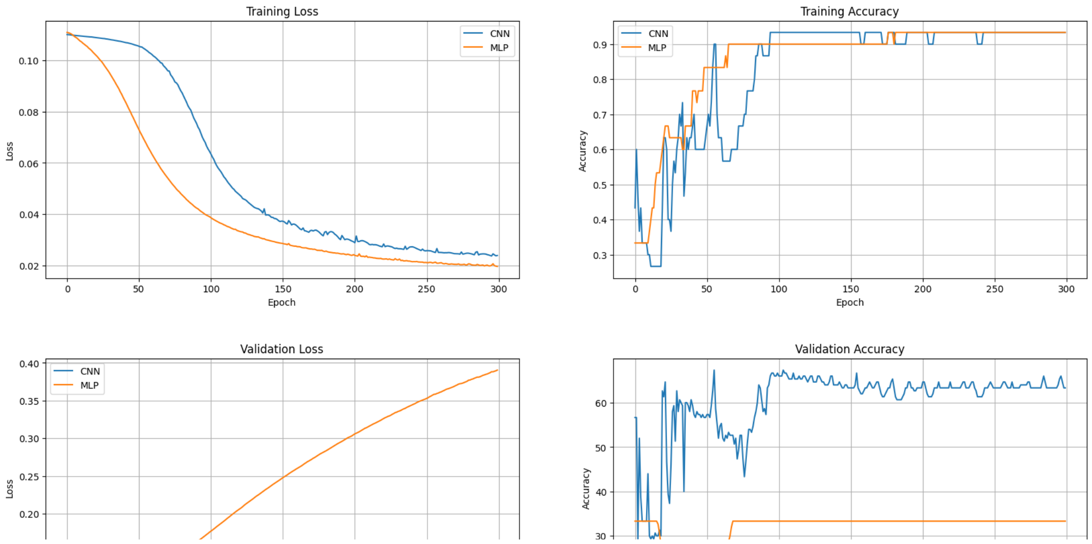
In [23]:
```
set_seed(seed)
train_loader_dict = dict()
num_train_images_list = [10, 20, 30, 40, 50]
possible_edge_location_ratio = 0.5
valid_loader = None

transforms = T.Compose([T.ToTensor()])
batch_size = 10
###############################################################################
# TODO: Implement train_loader_dict for each number of training images.      #
# Key: The number of training images (10, 50, 100, and 500)                  #
# Value: The corresponding dataloader                                        #
# The validation set size is 50 images per class                             #
# Hint: You can use the same code as above                                   #
# Hint: Pass possible_edge_location_ratio arguments to domain_config         #
# Hint: possible_edge_location_ratio is 0.5                                  #
###############################################################################

for num_image in num_train_images_list:

    train_dataset_config = dict(
        data_per_class=num_image,
        possible_edge_location_ratio = possible_edge_location_ratio
    )

    train_dataset = EdgeDetectionDataset(train_dataset_config, "train", transform =
    train_loader_dict[num_image] = torch.utils.data.DataLoader(train_dataset, batch_

valid_dataset_config = dict(
    data_per_class=50,
    possible_edge_location_ratio = possible_edge_location_ratio,
)

valid_dataset = EdgeDetectionDataset(valid_dataset_config, "valid", transform = tran
valid_loader = torch.utils.data.DataLoader(valid_dataset, batch_size = batch_size, s

###############################################################################
#                              END OF YOUR CODE                               #
###############################################################################
```

In [24]:
```
lr = 3e-3
num_epochs = 300
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()
mlp_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list, mlp_valid_loss_list
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model, mlp_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix = evaluate(cnn_model, cr
        mlp_valid_loss, mlp_valid_acc, mlp_confusion_matrix = evaluate(mlp_model, cr

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, mlp_train_loss_list,
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, mlp_valid_loss_lis

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = deepcopy(cnn_model.conv1.weight.detach().cpu())
    untrained_cnn_kernel_dict[num_image] = deepcopy(untrained_cnn_model.conv1.weight

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix
    mlp_confusion_matrix_dict[num_image] = mlp_confusion_matrix

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
```

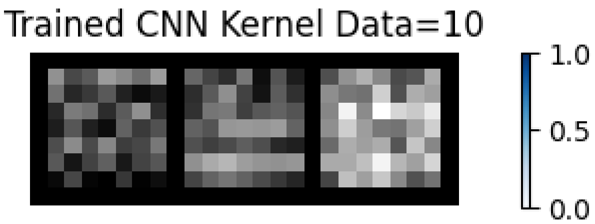
```
    mlp_acc_list.append(mlp_acc)
```

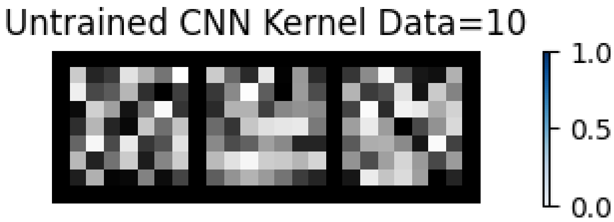
Training with 10 images

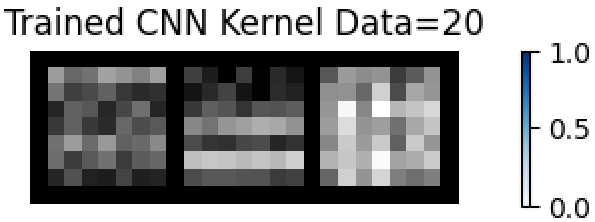
100%|██████████| 300/300 [00:30<00:00, 9.95it/s]

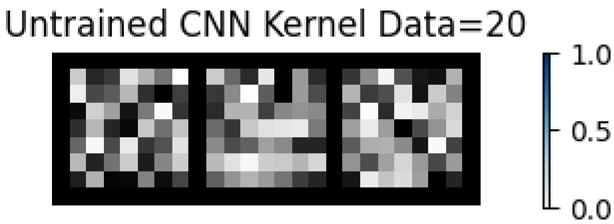


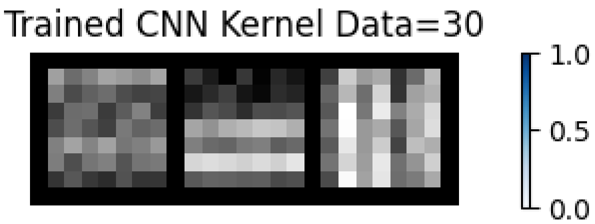
In [25]:
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel Data={
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN Kerne
```

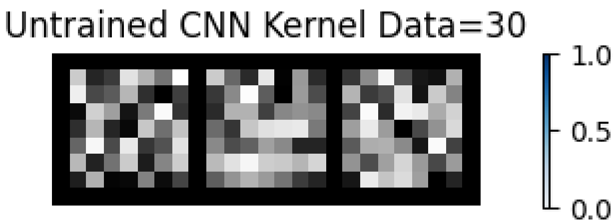


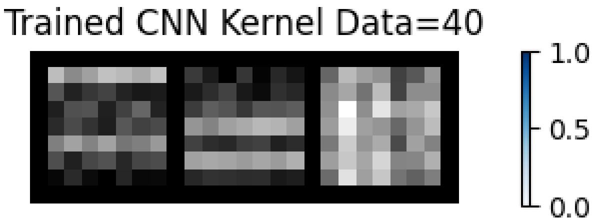


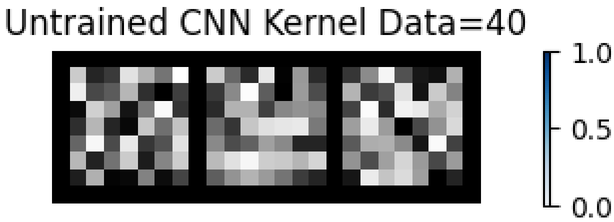


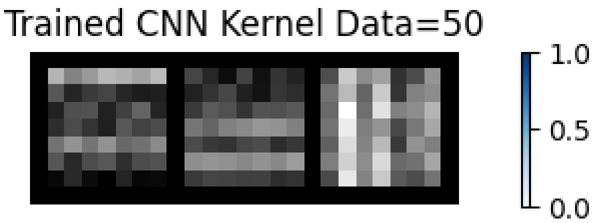


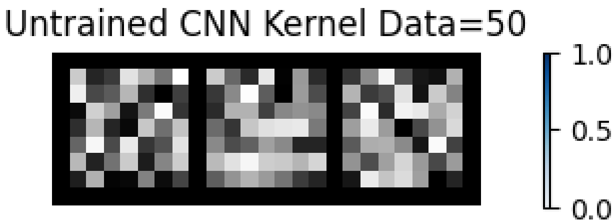












In this example, you will see that both CNN and MLP performance are worse than those in the previous question. If two models learn how to extract edges, they should be able to classify the images with edges even though the edges locate in the other half of the images. However, both models fail to do so. What would be the problem? To investigate this, let's first look at the confusion matrices for both models [link (https://en.wikipedia.org/wiki/Confusion_matrix)](https://en.wikipedia.org/wiki/Confusion_matrix).

In [27]:

```
## Plot the confusion matrix
for num_image, cnn_confusion_matrix in cnn_confusion_matrix_dict.items():
    mlp_confusion_matrix = mlp_confusion_matrix_dict[num_image]
    vis_confusion_matrix(cnn_confusion_matrix, ['horizontal', 'vertical', 'none'], '
    vis_confusion_matrix(mlp_confusion_matrix, ['horizontal', 'vertical', 'none'], '
```

```
d:\Anaconda\Anaconda_setup\envs\malning\lib\site-packages\ipykernel_launcher.py:7
56: UserWarning: FixedFormatter should only be used together with FixedLocator
d:\Anaconda\Anaconda_setup\envs\malning\lib\site-packages\ipykernel_launcher.py:7
57: UserWarning: FixedFormatter should only be used together with FixedLocator
```

Question

Why do you think the confusion matrix looks like this? Why does CNN misclassify the images with edge to those without edge? Why does MLP misclassify the images with vertical edge to those with horizontal edges and vice versa? Answer this question in your submission of the written assignment.

(Hint: Visualize some of the images in the training and validation set. And we are using kernel_size=7, which is large relative to the image size.)

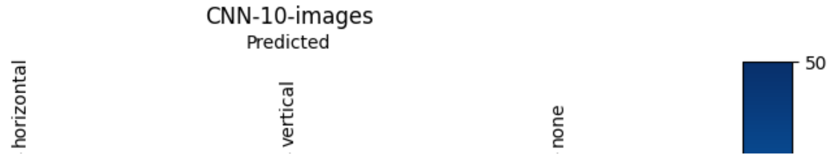
We can do better than this. We didn't explore hyperparameter space yet. Let's search hyperparameters that can generalize well to the validation set. We will change the learning rate, the number of epochs, and kernel size for CNN.

In [28]:

```
###############################################################################
# TODO: Try other num_epochs, lr, kernel_size. The validation accuracy       #
# should achieve 80% for 10 images per class.                                #
###############################################################################
lr = 5e-3
num_epochs = 1000
kernel_size = 3
###############################################################################
#                              END OF YOUR CODE                               #
###############################################################################
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_valid_acc_list = list()

cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=kernel_size)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix = evaluate(cnn_model, cr

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, None, None)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, None, None)

    cnn_acc = cnn_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.detach()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix

    print("CNN Acc: {}".format(cnn_acc))
    cnn_acc_list.append(cnn_acc)
```

```
Training with 10 images
```

```
100%|█████████████| 1000/1000 [01:47<00:00,  9.32it/s]
d:\Anaconda\Anaconda_setup\envs\malning\lib\site-packages\ipykernel_launcher.p
y:603: RuntimeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retained until e
xplicitly closed and may consume too much memory. (To control this warning, se
e the rcParam `figure.max_open_warning`).
```



Question

Why do you think MLP fails to learn the task while CNN can learn the task? Answer thi question in your submission of the written assignment.

(Hint: Think about the model architecture.)

Q5. When CNN is Worse than MLP

In this problem, we will see that CNN is not always better than MLP in the image domain. Using CNN assumes that the data has locally correlated, whatever data looks. We can manually 'whiten' or remove such local correlation simply by applying random permutation to the images. A random permutation matrix is a matrix that has the same number of rows and columns. Each row and column has the same number of 1s. The rest of the elements are 0s. For example, the following is a random permutation matrix.

```
[[0, 1, 0, 0],
 [0, 0, 0, 1],
 [1, 0, 0, 0],
 [0, 0, 1, 0]]
```

This matrix randomly reorders the elements of the vector. For example, if we apply this matrix to the vector `[1, 2, 3, 4]` , we will get `[2, 4, 1, 3]` . If we apply this matrix to the image, we will get the image with the same content, but the pixels are randomly shuffled. One property of the random permutation matrix is that it is invertible. It means that we can recover the original image by simply applying the inverse matrix to the shuffled image. From the information-theoretical perspective, the random permutation matrix preserves the mutual information of the image and the label.

We will repeat the same experiment as the previous problem. Visualize the dataset first.

In [33]:
```
set_seed(seed)
visual_domain_config = None
use_permutation = True

permutater = np.arange(28 * 28,  dtype=np.int32)
np.random.shuffle(permutater)
unpermutater = np.argsort(permutater)

visual_dataset = None

transforms = T.Compose([T.ToTensor()])
###############################################################################
# TODO: Implement visual_dataset for this new domain                          #
# Hint: If you read docstring of EdgeDetectionDataset, you will find          #
# 'use_permutation' args. Pass True to this args.                             #
# Also pass permutator to EdgeDetectionDataset                                #
###############################################################################
visualize_data_config = dict(
    data_per_class=10,
    num_classes=3,
    class_type=["horizontal", "vertical", "none"],
    use_permutation=True,
    permutater=permutater,
    unpermutater = unpermutater,
)

visual_dataset = EdgeDetectionDataset(visualize_data_config, mode='train', transforn
###############################################################################
#                              END OF YOUR CODE                               #
###############################################################################
```

In [35]:
```
## Visualize the images
unpermutator = visual_dataset.get_unpermutater()
print('Dataset Image before permutation')
vis_unpermuted_dataset(visual_dataset, num_classes=3, num_show_per_class=10, unpermu

print('Dataset Image after permutation')
vis_dataset(visual_dataset, num_classes=3, num_show_per_class=10)
```

```
Dataset Image before permutation
Dataset Image after permutation
```

Now let's train CNN and MLP on the permuted dataset.

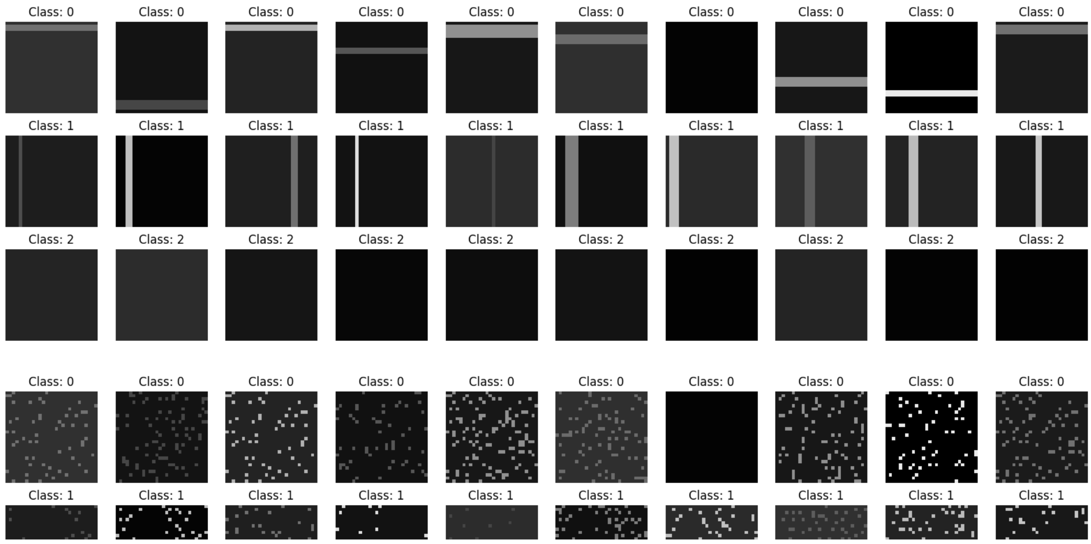
In [39]:
```
set_seed(seed)

train_loader_dict = dict()
num_train_images_list = [30, 40, 50, 60, 70]
use_permutation = True
valid_loader = None

permutater = np.arange(28 * 28,  dtype=np.int32)
np.random.shuffle(permutater)
unpermutater = np.argsort(permutater)

transforms = T.Compose([T.ToTensor()])

batch_size = 10
##############################################################################
# TODO: Implement train_loader_dict for each number of training images.     #
# Key: The number of training images (30, 40, 50, 60 and 70)                 #
# Value: The corresponding dataloader                                        #
# The validation set size is 50 images per class                             #
# 'use_permutation' args. Pass True to this args.                            #
# Also pass permutator/unpermutator to EdgeDetectionDataset                  #
##############################################################################

for num_image in num_train_images_list:

    train_data_config = dict(
        data_per_class=num_image,
        use_permutation=True,
        permutater=permutater,
        unpermutater = unpermutater,
    )

    train_dataset = EdgeDetectionDataset(train_data_config, mode='train', transform=
    train_loader_dict[num_image] = torch.utils.data.DataLoader(train_dataset, batch_

valid_data_config = dict(
    data_per_class=50,
    use_permutation=True,
    permutater=permutater,
    unpermutater = unpermutater,
)


valid_dataset = EdgeDetectionDataset(valid_data_config, mode='valid', transform=tran
valid_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size, shu

##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
```

Note that kernel size is 3 in this experiment.

In [40]:
```
lr = 1e-2
num_epochs = 300
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()
mlp_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=3)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10])
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9, weight_de
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9, weight_de

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list, mlp_valid_loss_list
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model, mlp_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix = evaluate(cnn_model, cr
        mlp_valid_loss, mlp_valid_acc, mlp_confusion_matrix = evaluate(mlp_model, cr

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, mlp_train_loss_list,
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, mlp_valid_loss_lis

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.detach()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix
    mlp_confusion_matrix_dict[num_image] = mlp_confusion_matrix

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
```

```
    mlp_acc_list.append(mlp_acc)
```

```
Training with 30 images

100%|██████████| 300/300 [01:15<00:00,  3.99it/s]
```



Question

What do you observe? What is the reason that CNN is worse than MLP? Answer this question in your submission of the written assignment.

(Hint: Think about the model architecture.)

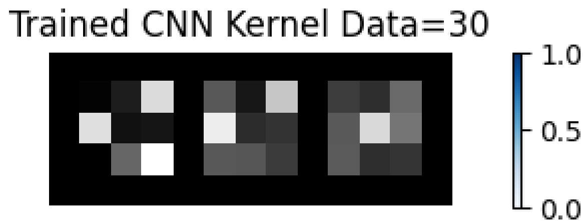
Question

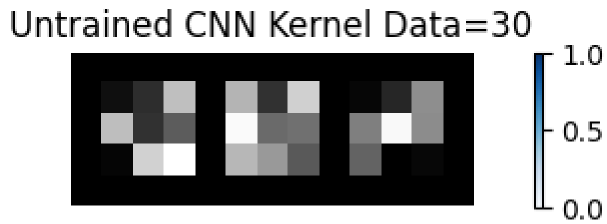
Assuming we are increasing kernel size of CNN. Does the validation accuracy increase or decrease? Why? Answer this question in your submission of the written assignment.

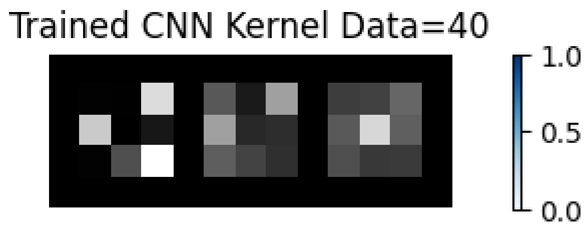
Now let's visualize CNN's learned kernel.

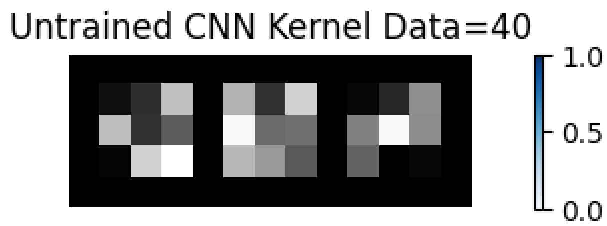
In [41]:

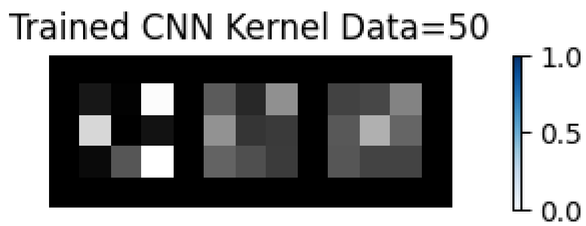
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel Data={
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN Kerne
```

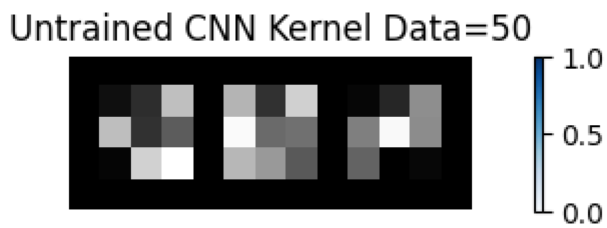


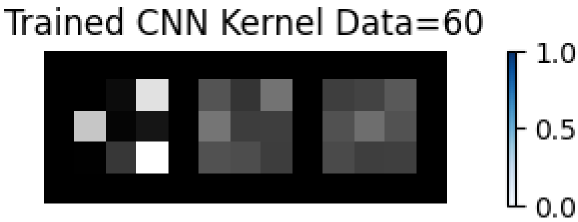


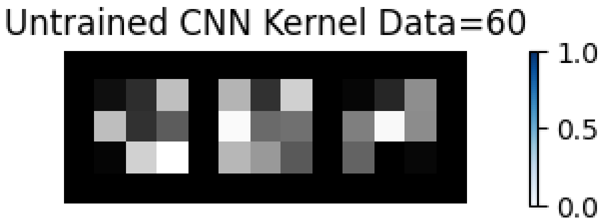


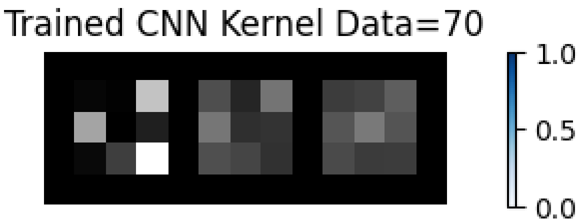


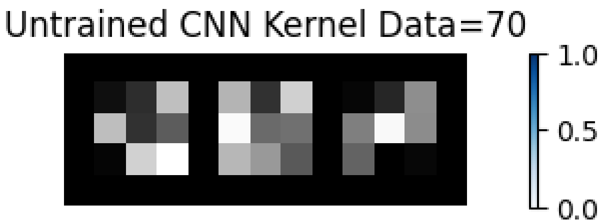












Question

How do the learned kernels look like? Explain why. Answer this question in your submission of the written assignment.

From the above example, we can see that CNN is not always better than MLP. We have to think about the domain (or task) of the dataset and the model architecture to decide which model is better.

Q6. Increasing the Number of Classes

OK, can we conclude that CNN has the inductive bias that the model is translation invariant? Let's try other experiments. We make the task harder. In this problem, we increase the number of classes to 5. The new classes are 0 for horizontal edges, 1 for vertical edges, 2 for diagonal edges, 3 for vertical and horizontal, and 4 for nothing. Let's generate the dataset with 10 images per class and visualize the dataset.

In [42]:
```
set_seed(seed)
visual_domain_config = None

visual_dataset = None

transforms = T.Compose([T.ToTensor()])
##############################################################################
# TODO: Implement visual_dataset for this new domain                         #
# Hint: If you read docstring of EdgeDetectionDataset, you will find         #
# 'class_type' args. Pass ['horizontal', 'vertical', 'diagonal', 'both',     #
# 'none'] to 'class_type' args.                                              #
##############################################################################

visual_data_config = dict(
    data_per_class = 10,
    num_classes = 5,
    class_type=["horizontal", "vertical", "diagonal", "both", "none"],
)

visual_dataset = EdgeDetectionDataset(visual_data_config, mode='train', transform=N

##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
```

Let's visualize the dataset first.

In [43]:
```
vis_dataset(visual_dataset, 5, 10)
```

Now let's make the new dataset. In this problem, we also see how the model performance changes as the number of images per class increases. Let's sweep the number of training images 10, 20, 30, 40, and 50. The validation set will be the same (50) for all the cases.

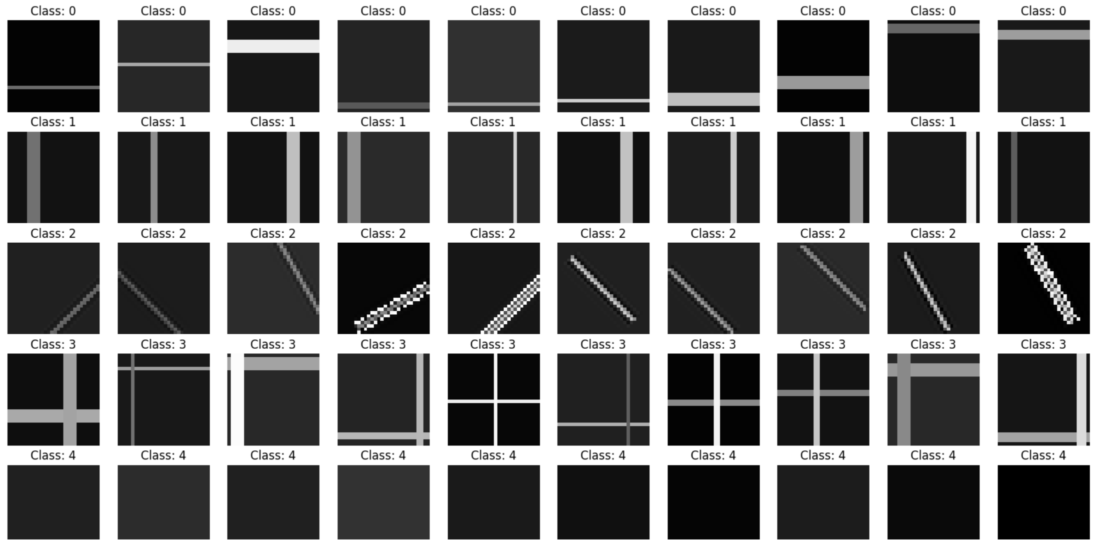
In [44]:

```
set_seed(seed)

train_dataset_config = None
train_loader_dict = dict()
num_train_images_list = [10, 20, 30, 40, 50]
valid_loader = None

transforms = T.Compose([T.ToTensor()])
batch_size = 10
############################################################################
# TODO: Implement train_loader_dict for each number of training images.     #
# Key: The number of training images (10, 20, 30, 40 and 50)               #
# Value: The corresponding dataloader                                       #
# The validation set size is 50 images per class                            #
# Hint: class_type = ['horizontal', 'vertical', 'diagonal', 'both', 'none'] #
# Hint: Be careful about the number of classes                              #
############################################################################

class_type = ['horizontal', 'vertical', 'diagonal', 'both', 'none']
train_dataset_config = dict(
    class_type=class_type,
    num_classes=len(class_type),
)
for num_train_images in num_train_images_list:
    train_dataset_config['data_per_class'] = num_train_images
    train_dataset = EdgeDetectionDataset(train_dataset_config, 'train', transform=tr
    train_loader_dict[num_train_images] = DataLoader(train_dataset, batch_size=batch


valid_dataset_config = dict(
    data_per_class=50,
    class_type=['horizontal', 'vertical', 'diagonal', 'both', 'none'],
    num_classes=len(class_type),
)
valid_dataset = EdgeDetectionDataset(valid_dataset_config, 'valid', transform=transf
valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=True)

############################################################################
#                              END OF YOUR CODE                             #
############################################################################
```

In [45]:
```
lr = 1e-2
num_epochs = 100
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
mlp_acc_list = list()

cnn_kernel_dict = dict()
untrained_cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()
mlp_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7, num_classes=5)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    mlp_model = ThreeLayerMLP(hidden_dims=[50, 10], num_classes=5)
    mlp_model.to(device)

    mlp_optimizer = optim.SGD(mlp_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    mlp_train_acc_list, mlp_valid_acc_list, mlp_train_loss_list, mlp_valid_loss_list
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr
        mlp_train_loss, mlp_train_acc = train_one_epoch(mlp_model, mlp_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix = evaluate(cnn_model, cr
        mlp_valid_loss, mlp_valid_acc, mlp_confusion_matrix = evaluate(mlp_model, cr

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        mlp_train_acc_list.append(mlp_train_acc)
        mlp_valid_acc_list.append(mlp_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        mlp_train_loss_list.append(mlp_train_loss)
        mlp_valid_loss_list.append(mlp_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, mlp_train_loss_list,
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, mlp_valid_loss_lis

    cnn_acc = cnn_valid_acc_list[-1]
    mlp_acc = mlp_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.detach()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix
    mlp_confusion_matrix_dict[num_image] = mlp_confusion_matrix

    print("CNN Acc: {}, MLP Acc: {}".format(cnn_acc, mlp_acc))
    cnn_acc_list.append(cnn_acc)
```

```
    mlp_acc_list.append(mlp_acc)
```

```
Training with 10 images

100%|██████████| 100/100 [00:14<00:00,  7.13it/s]
```



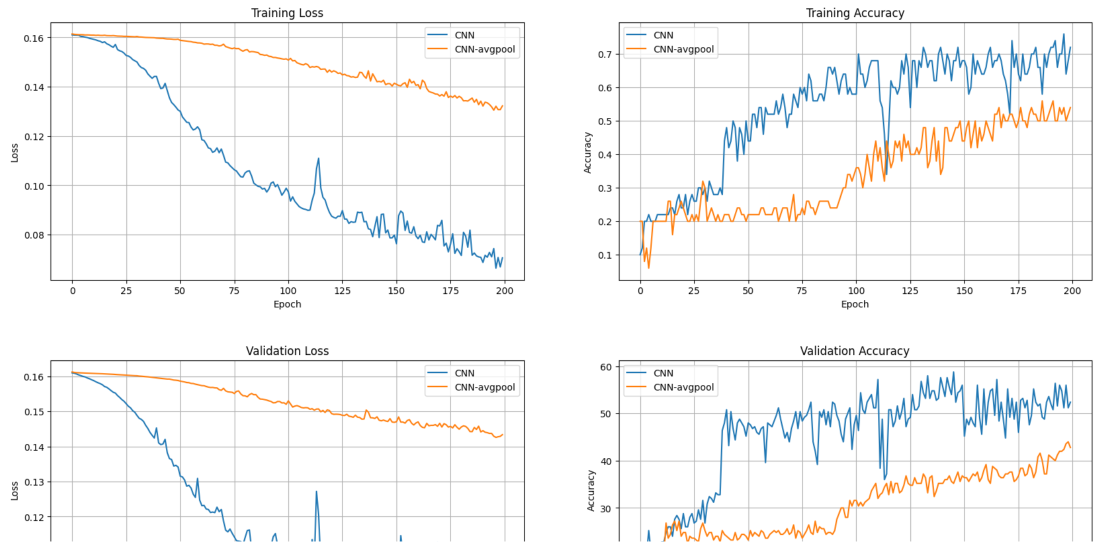
We look at two types of pooling operations to downsample the image features:

1. Max pooling: The maximum pixel value of the batch is selected.
2. Average pooling: The average value of all the pixels in the batch is selected.

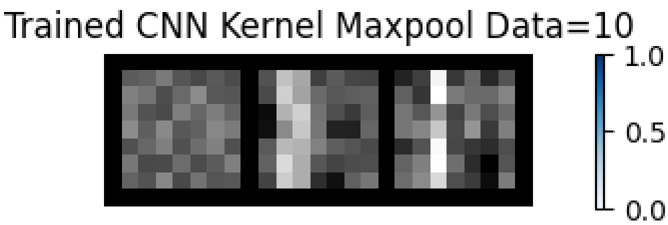
In [46]:
```
lr = 1e-2
num_epochs = 200
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_acc_list = list()
cnnavg_acc_list = list()

cnn_avg_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()
cnnavg_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = SimpleCNN(kernel_size=7, num_classes=5)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    cnnavg_model = SimpleCNN_avgpool(kernel_size=7, num_classes=5)# ThreeLayerMLP(h
    cnnavg_model.to(device)

    cnnavg_optimizer = optim.SGD(cnnavg_model.parameters(), lr=lr, momentum=0.9)
    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    cnnavg_train_acc_list, cnnavg_valid_acc_list, cnnavg_train_loss_list, cnnavg_val
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr
        cnnavg_train_loss, cnnavg_train_acc = train_one_epoch(cnnavg_model, cnnavg_o

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix = evaluate(cnn_model, cr
        cnnavg_valid_loss, cnnavg_valid_acc, cnnavg_confusion_matrix = evaluate(cnna

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        cnnavg_train_acc_list.append(cnnavg_train_acc)
        cnnavg_valid_acc_list.append(cnnavg_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)
        cnnavg_train_loss_list.append(cnnavg_train_loss)
        cnnavg_valid_loss_list.append(cnnavg_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, cnnavg_train_loss_li
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, cnnavg_valid_loss_

    cnn_acc = cnn_valid_acc_list[-1]
    cnnavg_acc = cnnavg_valid_acc_list[-1]

    cnn_avg_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix
    cnnavg_confusion_matrix_dict[num_image] = cnnavg_confusion_matrix

    print("CNN-maxpool Acc: {}, CNN-avgpool Acc: {}".format(cnn_acc, cnnavg_acc))
    cnn_acc_list.append(cnn_acc)
```

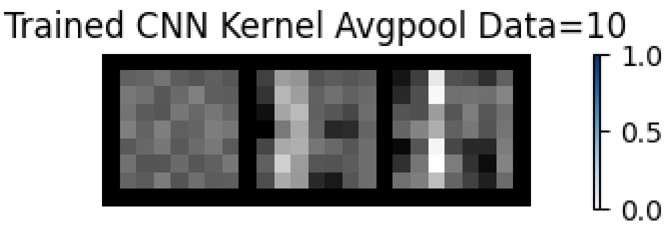
```
    cnnavg_acc_list.append(cnnavg_acc)
```

```
Training with 10 images
100%|██████████| 200/200 [00:30<00:00,  6.63it/s]
```



In [47]:
```
for num_image, cnn_kernel in cnn_kernel_dict.items():
    untrained_kernel = untrained_cnn_kernel_dict[num_image]
    cnn_avg_kernel = cnn_avg_kernel_dict[num_image]
    vis_kernel(cnn_kernel, ch=0, allkernels=False, title='Trained CNN Kernel Maxpoo
    vis_kernel(cnn_avg_kernel, ch=0, allkernels=False, title='Trained CNN Kernel Av
    vis_kernel(untrained_kernel, ch=0, allkernels=False, title='Untrained CNN Kerne
```





Question

Compare the performance of CNN with max pooling and average pooling. What are the advantages of each pooling method? Answer this question in your submission of the written assignment.

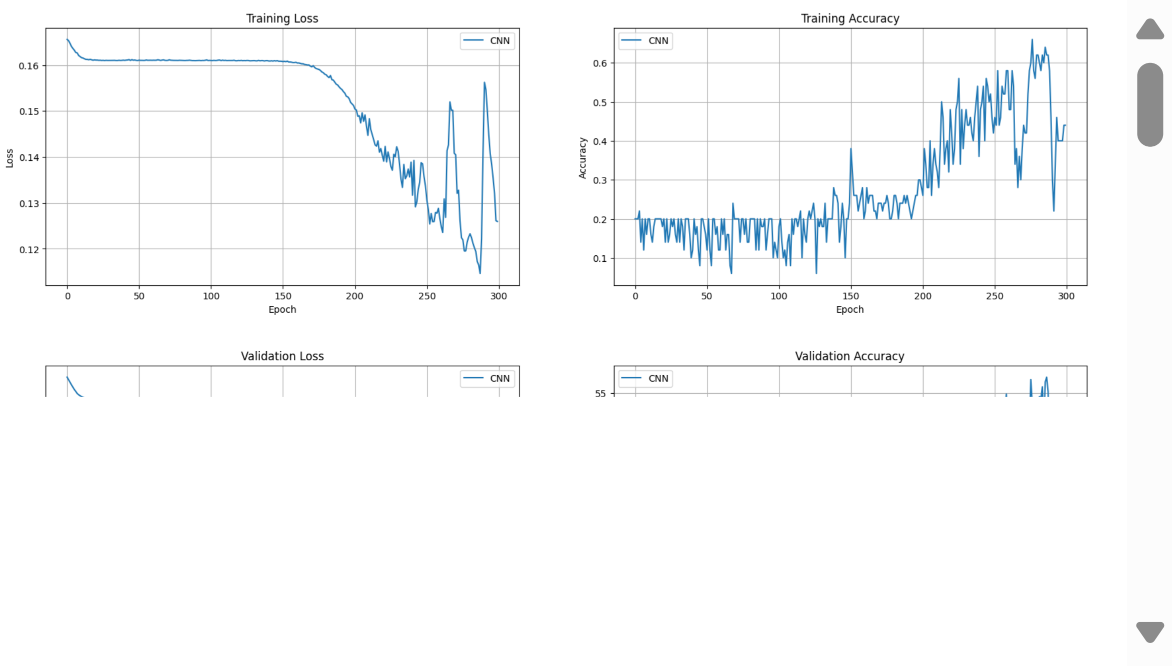
(Optional, Not Graded) Larger/Deeper CNNs

Ok, CNN performs pretty good. But what if we increase the width or the depth of CNN? The patterns that we have to detect are 5 but our kernels per layer are only 3. Intuitively, this is quite a suboptimal. Here, we will investigate the affect of increasing width and depth. Let's use the same dataset but we will use `DeeperCNN` and `WiderCNN` in `cnn.py` . `DeeperCNN` has 2 times more layers than `SimpleCNN` and `WiderCNN` has 2 times more kernels per layer than `SimpleCNN` . Let's train the models and visualize the validation accuracy.

In [50]:

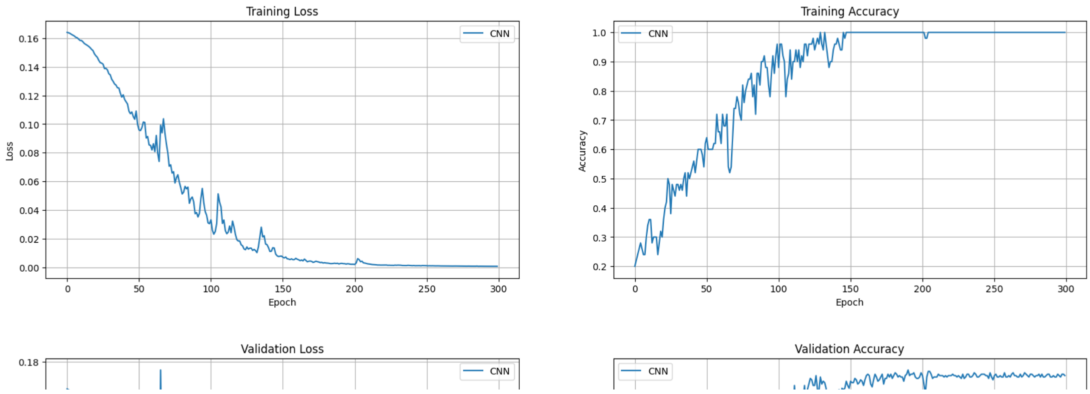
```
##############################################################################
# TODO: Training DeeperCNN and tuning hyperparameters Try other num_epochs, #
# lr, kernel_size. The validation accuracy                                   #
##############################################################################
lr = 0.01
num_epochs = 300
kernel_size = 3
##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_valid_acc_list = list()

cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = DeeperCNN(kernel_size=kernel_size)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix = evaluate(cnn_model, cr

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, None, None)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, None, None)

    cnn_acc = cnn_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.detach()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix

    print("CNN Acc: {}".format(cnn_acc))
    cnn_acc_list.append(cnn_acc)
```

```
Training with 10 images

100%|█████████████| 300/300 [00:24<00:00, 12.09it/s]
```


Training Loss
CNN
Loss
Epoch
Training Accuracy
CNN
Accuracy
Epoch
Validation Loss
CNN
Validation Accuracy
CNN

In [55]:
```
#########################################################################
# TODO: Training WiderCNN and tuning hyperparameters Try other num_epochs,  #
# lr, kernel_size. The validation accuracy                                 #
#########################################################################
lr = 0.01
num_epochs = 300
kernel_size = 7
#########################################################################
#                              END OF YOUR CODE                            #
#########################################################################
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
criterion = nn.CrossEntropyLoss()

cnn_valid_acc_list = list()

cnn_kernel_dict = dict()

cnn_confusion_matrix_dict = dict()

for num_image, train_loader in train_loader_dict.items():
    print("Training with {} images".format(num_image))
    set_seed(seed)
    cnn_model = WiderCNN(kernel_size=kernel_size)
    untrained_cnn_model = deepcopy(cnn_model)
    cnn_model.to(device)

    cnn_optimizer = optim.SGD(cnn_model.parameters(), lr=lr, momentum=0.9)

    # logging how training and validation accuracy changes
    cnn_train_acc_list, cnn_valid_acc_list, cnn_train_loss_list, cnn_valid_loss_list
    for epoch in tqdm(range(num_epochs)):
        cnn_train_loss, cnn_train_acc = train_one_epoch(cnn_model, cnn_optimizer, cr

        cnn_valid_loss, cnn_valid_acc, cnn_confusion_matrix = evaluate(cnn_model, cr

        cnn_train_acc_list.append(cnn_train_acc)
        cnn_valid_acc_list.append(cnn_valid_acc)
        cnn_train_loss_list.append(cnn_train_loss)
        cnn_valid_loss_list.append(cnn_valid_loss)

    vis_training_curve(cnn_train_loss_list, cnn_train_acc_list, None, None)
    vis_validation_curve(cnn_valid_loss_list, cnn_valid_acc_list, None, None)

    cnn_acc = cnn_valid_acc_list[-1]

    cnn_kernel_dict[num_image] = cnn_model.conv1.weight.detach().cpu()
    untrained_cnn_kernel_dict[num_image] = untrained_cnn_model.conv1.weight.detach()

    cnn_confusion_matrix_dict[num_image] = cnn_confusion_matrix

    print("CNN Acc: {}".format(cnn_acc))
    cnn_acc_list.append(cnn_acc)
```

```
Training with 10 images

100%|██████████| 300/300 [00:24<00:00, 12.23it/s]
```



Training Loss
CNN
Loss
Epoch
Training Accuracy
CNN
Accuracy
Epoch
Validation Loss
CNN
Validation Accuracy
CNN

Hand-Designing Filters

Convolutional layer, which is the most important building block of CNN, actively utilizes the concept of filters used in traditional image processing. Therefore, it is quite important to know and understand the types and operation of image filters. In this notebook, we will design convolution filters by hand to understand the operation of convolution.

In [1]:
```python
# As usual, a bit of setup

import time
import numpy as np
import matplotlib.pyplot as plt
import requests
import random
import torch
from PIL import Image
from scipy import ndimage


seed = 7
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

imagenet_mean = np.array([0.485, 0.456, 0.406])
imagenet_std = np.array([0.229, 0.224, 0.225])

def show_image(image, title=''):
    # image is [H, W, 3]
    # assert image.shape[2] == 3
    image = torch.tensor(image)
    plt.imshow(torch.clip((image) * 255, 0, 255).int())
    plt.title(title, fontsize=16)
    plt.axis('off')
    return

def show_multiple_images(images=[], titles=[]):
    assert len(images) == len(titles), "length of two inputs are not equal"
    N = len(images)
    # make the plt figure larger
    plt.rcParams['figure.figsize'] = [24, 24]

    for i in range(N):
        plt.subplot(1, N, i+1)
        show_image(images[i], titles[i])

    plt.show()

def rgb2gray(rgb):
    r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b

    return gray
```

Designing Filters

In this problem, you will design simple blurring and edge detection filters.

In [2]:
```
img_url = 'https://user-images.githubusercontent.com/11435359/147738734-196fd92f-926

img = Image.open(requests.get(img_url, stream=True).raw)
img = np.array(img) / 255
gray_img = rgb2gray(img)


show_image(gray_img, 'Original Image')
```

Original Image

Image Blurring

Image blurring also called image smoothing, usually refers to making an image fuzzy. This filtering is typically used to remove noise in the image. There are various types of image blurring filters, but the three most common are Averaging, Gaussian blurring, and Median filtering.

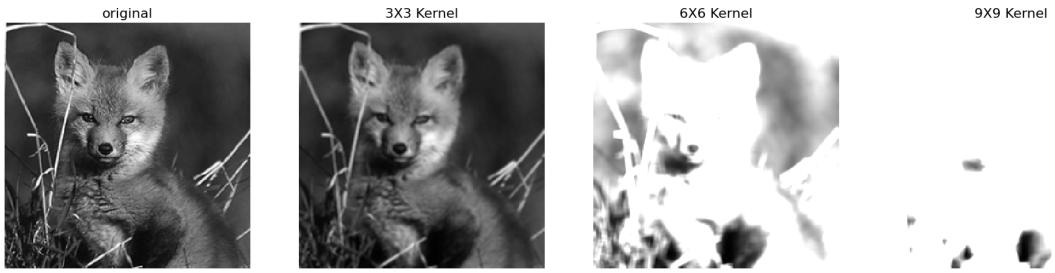
We will implement Averaging filtering in this project. Averaging filtering is also called moving averaging in 1-D. This filter works by placing a mask over an image and then taking the average of all the image pixels covered by the mask and replacing the central pixel with that value.

If the kernel size of the image filter is $n \times n$, then the size of each element in the kernel matrix is $\frac{1}{n^2}$. Also, the sum of all the elements in the kernel matrix will be 1. So, if the kernel size is 3×3, kernel will be as follows.

$$\frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

In [6]:

```
def averaging_filtering(image, filter_size=3):
    kernel = None
    ###############################################################################
    # TODO: Implement the averaging filter with the given filter size.           #
    # Hint: You can use np.ones                                                   #
    ###############################################################################
    kernel = np.ones((filter_size, filter_size)) / 9.0
    #print(kernel)
    ###############################################################################
    #                              END OF YOUR CODE                               #
    ###############################################################################
    output = ndimage.convolve(image, kernel)
    return output

avg_images, avg_titles = [gray_img], ['original']
for kernel_size in [3, 6, 9]:
    averaging_image = averaging_filtering(gray_img, kernel_size)
    avg_images.append(averaging_image)
    avg_titles.append(f'{kernel_size}X{kernel_size} Kernel')

show_multiple_images(avg_images, avg_titles)
```



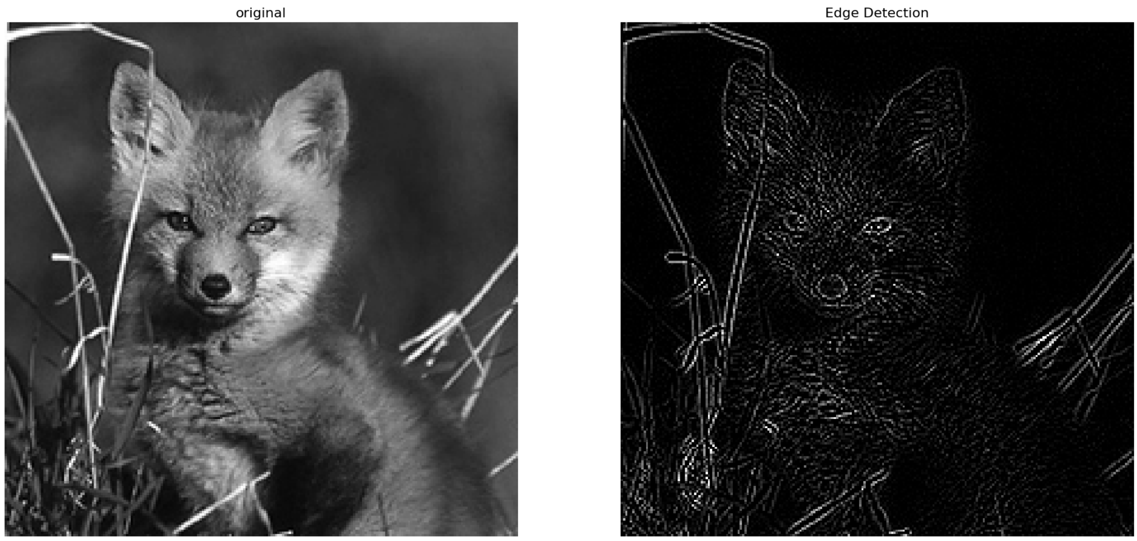
Edge Detection

Next, we will implement a simple edge detection filter. Edge detection is an algorithm that detects edges in an image. An edge in an image is a place where the brightness of the image changes abruptly or discontinuously. Several edge detection algorithms exist, such as the Canny edge detector, the Sobel filter and the Laplacian derivatives filter.

Here, we will implement the Laplacian derivatives filter. This operation simply computes the Laplacian of the image. This filter masks are as follows:

$\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$

In [7]:
```python
def edge_detecting(image):
    kernel = None
    ############################################################################
    # TODO: Implement the Laplacian derivative filter.                         #
    ############################################################################
    kernel = np.ones((3,3))
    kernel[0][0] = kernel[0][2] = kernel[2][0] = kernel[2][2] = 0
    kernel[1][1] = -4
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################
    output = ndimage.convolve(image, kernel)
    return output

edge_images, edge_titles = [gray_img], ['original']
edge_image = edge_detecting(gray_img)
edge_images.append(edge_image)
edge_titles.append(f'Edge Detection')

show_multiple_images(edge_images, edge_titles)
```



Memory considerations when training Neural Networks on GPUs

In this homework, we will train a ResNet model on CIFAR-10 using PyTorch and explore it's implications on GPU memory.

We will explore various systems considerations, such as the effect of batch size on memory usage, the effect of different optimizers (SGD, SGD with momentum, Adam), and we will try to minimize the memory usage of training our model by applying gradient accumulation.

Setup the environment

If you're running on colab - make sure you are using a GPU runtime. You can select a GPU runtime by clicking on `Runtime` -> `Change Runtime Type`.

> 💡 Hint - if you hit your colab GPU usage limit, try again in a few hours.

In [ ]:
```
#@title Mount your Google Drive

import os
from google.colab import drive

try:
  drive.mount('/content/gdrive')

  DRIVE_PATH = '/content/gdrive/My\ Drive/cs182hw4_sp23'
  DRIVE_PYTHON_PATH = DRIVE_PATH.replace('\\', '')
  if not os.path.exists(DRIVE_PYTHON_PATH):
    %mkdir $DRIVE_PATH

  ## the space in `My Drive` causes some issues,
  ## make a symlink to avoid this
  SYM_PATH = '/content/cs182hw4'
  if os.path.isdir(SYM_PATH):
    raise Exception(f"Path already exists - please delete {SYM_PATH} before mountin
  else:
    !ln -sf $DRIVE_PATH $SYM_PATH
except Exception as e:
  print(e)
  print("WARNING - Unable to mount google drive for storing logs. Storing logs in th
  os.makedirs('/content/cs182hw4/', exist_ok=True)
```

In [ ]:
```
#@title Install dependencies

!pip install gputil
```

In [1]:
```
import gc
import GPUtil
import os
import subprocess
import torch
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import random
import time


ROOT_PATH = '/content/cs182hw4/'

# Define the CSV format for logging memory usage. Used later in this notebook.
MEMORY_LOG_FMT = ['timestamp', 'memUsage']
TRAIN_LOG_FMT = ['timestamp', 'epoch', 'memUsage', 'loss', 'accuracy']

if torch.cuda.is_available():
  print("Using GPU.")
  device = torch.device("cuda:0")
else:
  print("!!! WARNING !!! - Could not find a GPU - please use a GPU for this homework
  device = torch.device("cpu")

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

```
Using GPU.
```

Define helper functions and download CIFAR-10 dataset

In [2]:
```
seed = 42
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)

def get_allocated_memory_str():
    return "Allocated memory: {:.2f} GB".format(torch.cuda.memory_allocated(device)

def run_nvidia_smi():
    if torch.cuda.is_available():
      print(subprocess.check_output("nvidia-smi", shell=True).decode("utf-8"))
    else:
      print("Running on CPU")

def get_gpu_memory_usage() -> float:
    # Use GPUtil python library to get GPU memory usage
    if torch.cuda.is_available():
      return GPUtil.getGPUs()[0].memoryUsed
    else:
      return 0

def cleanup_memory():
    gc.collect()
    torch.cuda.empty_cache()

# Define transformations for the input data. We resize the 32x32 inputs to
# 224x224 which is the input shape for the ResNet family of models.
transform = transforms.Compose([
    transforms.Resize((224, 224)), # Resize to 224x224 for ResNet models
    transforms.ToTensor()
])

data_train = torchvision.datasets.CIFAR10(root='./data', train=True, download=True
data_test = torchvision.datasets.CIFAR10(root='./data', train=False, download=True


# We randomly subsample the dataset here to train our models faster for this noteboo
SUBSAMPLE_SIZE = 1024*4
random_sample_idxs = torch.randint(len(data_train), (SUBSAMPLE_SIZE,))
subsampled_train_data = torch.utils.data.Subset(data_train, indices=random_sample_ic
```

```
Files already downloaded and verified
Files already downloaded and verified
```

1. Managing GPU memory when training deep models

One of the most common bottlenecks you will run into when training your deep learning models is the amount of GPU memory available to you. The exact memory usage of your training process depends on the specific model architecture and the size of the input data. The main components taking up GPU memory during training are:

- **Model Parameters**: The weights and biases of the model are stored in GPU memory during training. The number of parameters in a deep learning model can range from a few thousand to millions or even billions, depending on the model architecture and the size of the input data.
- **Activations**: The activations of each layer of the model are stored in GPU memory during the forward pass. The size of the activations can depend on the batch size and the number of hidden units in each layer. As the batch size increases, so does the size of the activations, which can quickly consume a large amount of GPU memory.
- **Gradients**: During the backward pass, the gradients of each layer with respect to the loss function are computed and stored in GPU memory. The size of the gradients can depend on the batch size and the number of hidden units in each layer. Like activations, larger batch sizes can lead to larger gradients and increased memory usage.
- **Input Data**: The input data, such as images or text, can also take up GPU memory during training. The size of the input data can depend on the input shape and the batch size.
- **Optimizer State**: The state of the optimizer, such as the momentum or running average of gradients, is stored in GPU memory during training. The size of the optimizer state can depend on the optimizer algorithm and the size of the model parameters.

Let's analyze the ResNet-152 model and CIFAR-10 input sizes

We can count the number of parameters in the model by loading it and inspecting it. Once we

In [3]:
```
def analyze_model_and_inputs(model):
    print("Train data size: {}".format(len(data_train)))
    print("Test data size: {}".format(len(data_test)))

    # Fetch an example image to get image size
    image, label = data_train[0]
    print("Image input size: {}".format(image.size()))

    # Get model parameter count
    print("Model parameters: {}".format(sum(p.numel() for p in model.parameters() i

    # Get model size in MB
    print("Model size estimate (MB): {}".format(sum(p.numel() * p.element_size() for
```

In [4]:
```
model = torchvision.models.resnet152(weights=None, num_classes=10)
model.to(device)  # Load the model into GPU memory
analyze_model_and_inputs(model)
```

```
Train data size: 50000
Test data size: 10000
Image input size: torch.Size([3, 224, 224])
Model parameters: 58164298
Model size estimate (MB): 232.657192
```

Let's get to know our GPU better

Now that we have loaded the model onto the GPU, we will now use the `nvidia-smi` utility to measure the GPU memory utilization.

In [5]:
```
!nvidia-smi
```

```
Tue Sep 19 20:43:21 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 517.00       Driver Version: 517.00       CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ... WDDM  | 00000000:01:00.0 Off |                  N/A |
| N/A   40C    P8     7W /  N/A |   1549MiB /  6144MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     15764      C   ...p\envs\malning\python.exe    N/A      |
|    0   N/A  N/A     19064      C   ...p\envs\malning\python.exe    N/A      |
+-----------------------------------------------------------------------------+
```

Note that the actual memory usage on the GPU is anywhere between ~500-1000 MB larger than the model size computed above. Why? In addition to loading the model, the GPU also needs to be initialized with essential kernels, memory allocation tables, and other GPU related state necessary to using the GPU. This is called the CUDA context.

The CUDA context can be considered a fixed memory overhead for using a Nvidia GPU.

Questions (answer in written submission)

Q1a. How many trainable parameters does ResNet-152 have? What is the estimated size of the model in MB?

Q1b. Which GPU are you using? How much total memory does it have?

Q1c. After you load the model into memory, what is the memory overhead (size) of the CUDA context loaded with the model?

> Hint - CUDA context size in this example is roughly (total GPU memory utilization - model size)

2. Optimizer memory usage

The choice of optimizer affects the memory used to train your model. Different optimizers have different memory requirements for storing the gradients and the optimizer state. For example, the Adam optimizer stores a moving average of the gradients and the squared gradients for each parameter, which requires more memory than SGD.

Let's compare the memory usage of three different optimizers - SGD, SGD with momentum and ADAM.

In [5]:

```
# Training function
def train_model(model, train_loader, criterion, optimizer, epochs=10, memory_log_pa
    os.makedirs(os.path.dirname(memory_log_path), exist_ok=True)
    with open(memory_log_path, 'w') as f:
      f.write(",".join(MEMORY_LOG_FMT) + "\n")
    for epoch in range(epochs):
        model.train()
        for i, (images, labels) in enumerate(train_loader):
            images = images.to(device)
            labels = labels.to(device)
            with torch.set_grad_enabled(True):
              # Zero all gradients
              optimizer.zero_grad()

              # Get outputs
              outputs = model(images)

              # Compute loss
              loss = criterion(outputs, labels)
              loss.backward()

              # Run optimizer update step
              optimizer.step()

              # Print stats every 100 iterations
              if i % 100 == 0:
                  gpu_memory_usage = get_gpu_memory_usage()
                  print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, GPU Mem: {}'.for
                  memory_log = [str(time.time()), str(gpu_memory_usage)]
                  with open(memory_log_path, 'a') as f:
                    f.write(",".join(memory_log) + "\n")
            del loss, outputs, images, labels  # To get accurate memory usage info

# Memory profiling function
def profile_mem_usage(optimizer_str):
    """
    Profiles the memory usage of ResNet-152 on CIFAR-10 with the specified optimizer

    optimizer_str: str - Can be either of 'SGD', 'SGD_WITH_MOMENTUM' and 'ADAM'
    """
    # Clean up any dangling objects
    cleanup_memory()
    BATCH_SIZE = 8

    # Since we just want to inspect memory usage, run only one minibatch
    subsampled_data = torch.utils.data.Subset(data_train, range(0, BATCH_SIZE))
    train_loader = torch.utils.data.DataLoader(dataset=subsampled_data,
                                               batch_size=BATCH_SIZE,
                                               shuffle=True)

    # Load model and define loss function
    model = torchvision.models.resnet152(weights=None, num_classes=10)
    model.to(device)
    criterion = torch.nn.CrossEntropyLoss()

    # Choose optimizer
    if optimizer_str == 'SGD':
        optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
    elif optimizer_str == 'SGD_WITH_MOMENTUM':
        optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
    elif optimizer_str == 'ADAM':
```

```
        optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    else:
        raise NotImplementedError

    memory_log_path = ROOT_PATH + f'logs/resnet152__{optimizer_str}.csv'
    train_model(model, train_loader, criterion, optimizer, epochs=1, memory_log_path
    print(f"Memory usage log for {optimizer_str} stored at {memory_log_path}. Restar
```

Run memory profiling for various optimizers!

In the cell below, run the `profile_mem_usage` method with three optimizers - `'SGD'`, `'SGD_WITH_MOMENTUM'`, `'ADAM'` .

💥 NOTE 💥 - to get accurate memory utilization measurements, you **should restart your runtime between invoking `profile_mem_usage` for different optimizers!**

There is state in GPU memory that is not collect by explicitly calling the garbage collector, and thus restarting the runtime is necessary. Your files in colab should persist across runs.

In [6]:
```
# TODO - Run this cell for different optimizers by uncommenting one line at a time.
#
# Make sure to restart the colab runtime between different runs else your
# memory profiles may be inaccurate!

# run in local environment
ROOT_PATH = ""

#profile_mem_usage('SGD')
#profile_mem_usage('SGD_WITH_MOMENTUM')
profile_mem_usage('ADAM')
```

```
Epoch [1/1], Step [1/1], Loss: 2.0387, GPU Mem: 3540.0
Memory usage log for ADAM stored at logs/resnet152__ADAM.csv. Restart your runtim
e (Runtime->Restart Runtime) before running for other optimizers!
```

Analyzing memory usage profiles

Now that you have run `profile_mem_usage` for different optimizers, let's print the memory usage we logged while training with each optimizer.

In [7]:
```
OPTIMIZER_LIST = ['SGD', 'SGD_WITH_MOMENTUM', 'ADAM']
memory_log_path = ROOT_PATH + 'logs/resnet152__{opt}.csv'

def print_mem_profiling_results():
    print("====== Memory Profiling Results ======")
    for opt in OPTIMIZER_LIST:
        assert os.path.exists(memory_log_path.format(opt=opt)), f'Memory profile no
        df = pd.read_csv(memory_log_path.format(opt=opt))
        mem_usage = df['memUsage'].iloc[0]
        print(f'{opt}: {mem_usage} MB')

print_mem_profiling_results()
```

```
====== Memory Profiling Results ======
SGD: 3374.0 MB
SGD_WITH_MOMENTUM: 3456.0 MB
ADAM: 3540.0 MB
```

Questions (answer in written submission)

2a. What is the total memory utilization during training with SGD, SGD with momentum and Adam optimizers? Report in MB individually for each optimizer.

2b. Which optimizer consumes the most memory? Why?

> 💡 Hint - refer to the weight update rule for each optimizer. Which one requires the most parameters to be stored in memory?

3. Investigating the effect of batch size on convergence and GPU memory

Batch size is an important parameter in training neural networks that can have a significant effect on GPU memory usage. The larger the batch size, the more data the model processes at once, and therefore, the more GPU memory it requires to store the inputs, activations, and gradients.

As the batch size increases, the memory required to store the intermediate results during training increases linearly. This is because the model needs to keep track of more activations and gradients for each layer. However, the actual memory usage can also depend on the specific neural network architecture, as some models require more memory than others to process the same batch size.

If the batch size is too large to fit in the available GPU memory, the training process will fail with an out-of-memory error. On the other hand, if the batch size is too small, the training may be slower due to inefficient use of the GPU, as the GPU may spend more time waiting for data to be transferred from CPU to GPU.

Therefore, choosing an appropriate batch size is important to balance training speed and memory usage. This often involves some trial and error to find the largest batch size that can fit in the available GPU memory while still providing good training results.

Learning Rate and Batch Size

Batch size and learning rate are closely related. When batch size is increased, the gradient estimate becomes less noisy because it is computed over more samples. As a result, the learning rate can be increased, allowing the optimization algorithm to take larger steps towards the optimum. This is because a larger batch size gives a more accurate estimate of the direction of the gradient and larger steps can reduce convergence time.

Large batch training becomes particularly important in data-parallel distributed training, where extremely large batch sizes are distributed over many GPUs. The paper ["Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour" (https://arxiv.org/pdf/1706.02677.pdf)](https://arxiv.org/pdf/1706.02677.pdf) is one of the earliest works showing how large batch training makes fast large scale distributed training possible. It also proposes a simple linear scaling rule for setting the learning rate for a given batch size, which we use to set learning rates in `LR_MAP` below.

Let's try training our model with different batch sizes

In the below cells, we'll try to run training for different batch sizes and evaluate the performance.

> Note - you may run out of memory for large batch sizes, and that is expected! Ignore those large batch sizes and stick with the batch sizes that can fit on your GPU.

Let's first define helper functions.

In [8]:

```
# Test function
def test_model(model, test_loader, label='test'):
    print("Testing model.")
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0

        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)

            _, predicted = torch.max(outputs.data, 1)

            correct += (predicted == labels).sum().item()
            total += labels.size(0)

        # Compute accuracy
        accuracy = 100 * correct / total
        print(f'Accuracy of the model on {label} images: {accuracy} %')
        del outputs, images, labels  # To get accurate memory usage info
    return accuracy

# Training function
def train_model(model, train_loader, criterion, optimizer, epochs=10, memory_log_pa
    os.makedirs(os.path.dirname(memory_log_path), exist_ok=True)
    with open(memory_log_path, 'w') as f:
      f.write(",".join(TRAIN_LOG_FMT) + "\n")
    for epoch in range(epochs):
        model.train()
        last_loss = 0
        for i, (images, labels) in enumerate(train_loader):
            images = images.to(device)
            labels = labels.to(device)
            with torch.set_grad_enabled(True):
              # Zero all gradients
              optimizer.zero_grad()

              # Get outputs
              outputs = model(images)

              # Compute loss
              loss = criterion(outputs, labels)
              loss.backward()

              # Run optimizer update step
              optimizer.step()

              last_loss = loss.item()
              # Print stats every 100 iterations
              if i % 10 == 0:
                  gpu_memory_usage = get_gpu_memory_usage()
                  print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, GPU Mem: {}'.for
            del loss, outputs, images, labels  # To get accurate memory usage info

        # Report test or train accuracy at the end of every epoch
        if test_loader:
          accuracy = test_model(model, test_loader, label='test')
        else:
          accuracy = test_model(model, train_loader, label='train')
```

```
        # Log results
        memory_log = [str(time.time()), str(epoch+1), str(gpu_memory_usage), str(las
        with open(memory_log_path, 'a') as f:
            f.write(",".join(memory_log) + "\n")

# Set learning rates for different batch sizes (emperically determined and linearly
LR_MAP = {
    4: 0.0001,
    8: 0.0002,
    16: 0.0004,
    32: 0.0008,
    64: 0.0016,
    128: 0.0032,
    256: 0.0064,
    512: 0.0064,
    1024: 0.0064
}

# Executor function
def run_train(batch_size, epochs=10):
    cleanup_memory()

    lr = LR_MAP[batch_size]
    print(f"Training model with batch size {batch_size} and lr {lr}.")

    train_loader = torch.utils.data.DataLoader(dataset=subsampled_train_data, batch_

    # We use a smaller model (resnet18) to train faster
    model = torchvision.models.resnet18(weights=None, num_classes=10)
    model.to(device)
    criterion = torch.nn.CrossEntropyLoss()

    optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9)

    # Output path for memory logs
    memory_log_path = ROOT_PATH + f'logs/resnet18__{batch_size}.csv'

    # Run training!
    train_model(model, train_loader, criterion, optimizer, epochs=epochs, memory_log
```

Run training for different batch sizes and record their memory utilization!

In the cell below, **run the** `run_train` **method for batch sizes 4, 16, 64, 256, 1024**.

This method will log the loss, accuracy, wall clock time and memory utilization under `/content/cs182hw4/logs` directory, so you can safely restart the runtime between invocations.

Like before, to get accurate memory utilization measurements, you should **restart your runtime between invoking** `run_train` **for different batch sizes!**

In [9]:
```
# TODO - Run this cell for different batch sizes by uncommenting one line at a time.
#
# Make sure to restart the colab runtime between different runs else your
# memory profiles may be inaccurate!

# Run each batch size for at least 10 epochs. You can configure this to be larger if
epochs = 10

run_train(4, epochs=epochs)
# run_train(16, epochs=epochs)
# run_train(64, epochs=epochs)
# run_train(256, epochs=epochs)
# run_train(1024, epochs=epochs)
```

```
Training model with batch size 4 and lr 0.0001.
Epoch [1/10], Step [1/1024], Loss: 2.4767, GPU Mem: 1624.0
Epoch [1/10], Step [11/1024], Loss: 2.1517, GPU Mem: 1684.0
Epoch [1/10], Step [21/1024], Loss: 2.3811, GPU Mem: 1684.0
Epoch [1/10], Step [31/1024], Loss: 2.3056, GPU Mem: 1684.0
Epoch [1/10], Step [41/1024], Loss: 2.4940, GPU Mem: 1684.0
Epoch [1/10], Step [51/1024], Loss: 2.3186, GPU Mem: 1684.0
Epoch [1/10], Step [61/1024], Loss: 2.3357, GPU Mem: 1684.0
Epoch [1/10], Step [71/1024], Loss: 2.2846, GPU Mem: 1684.0
Epoch [1/10], Step [81/1024], Loss: 2.2858, GPU Mem: 1684.0
Epoch [1/10], Step [91/1024], Loss: 2.1944, GPU Mem: 1684.0
Epoch [1/10], Step [101/1024], Loss: 2.1375, GPU Mem: 1684.0
Epoch [1/10], Step [111/1024], Loss: 2.3502, GPU Mem: 1684.0
Epoch [1/10], Step [121/1024], Loss: 2.0923, GPU Mem: 1684.0
Epoch [1/10], Step [131/1024], Loss: 2.2534, GPU Mem: 1684.0
Epoch [1/10], Step [141/1024], Loss: 2.3388, GPU Mem: 1684.0
Epoch [1/10], Step [151/1024], Loss: 2.0335, GPU Mem: 1684.0
Epoch [1/10], Step [161/1024], Loss: 2.1323, GPU Mem: 1684.0
Epoch [1/10], Step [171/1024], Loss: 2.3699, GPU Mem: 1684.0
```

In [10]:
```
run_train(16, epochs=epochs)
```

```
Training model with batch size 16 and lr 0.0004.
Epoch [1/10], Step [1/256], Loss: 2.4846, GPU Mem: 2142.0
Epoch [1/10], Step [11/256], Loss: 2.3922, GPU Mem: 2142.0
Epoch [1/10], Step [21/256], Loss: 2.3477, GPU Mem: 2142.0
Epoch [1/10], Step [31/256], Loss: 2.2929, GPU Mem: 2142.0
Epoch [1/10], Step [41/256], Loss: 2.2871, GPU Mem: 2142.0
Epoch [1/10], Step [51/256], Loss: 2.2597, GPU Mem: 2142.0
Epoch [1/10], Step [61/256], Loss: 2.1548, GPU Mem: 2142.0
Epoch [1/10], Step [71/256], Loss: 2.1083, GPU Mem: 2142.0
Epoch [1/10], Step [81/256], Loss: 2.3286, GPU Mem: 2142.0
Epoch [1/10], Step [91/256], Loss: 2.1467, GPU Mem: 2142.0
Epoch [1/10], Step [101/256], Loss: 2.1294, GPU Mem: 2142.0
Epoch [1/10], Step [111/256], Loss: 2.2383, GPU Mem: 2142.0
Epoch [1/10], Step [121/256], Loss: 2.0882, GPU Mem: 2142.0
Epoch [1/10], Step [131/256], Loss: 2.1027, GPU Mem: 2142.0
Epoch [1/10], Step [141/256], Loss: 2.2149, GPU Mem: 2142.0
Epoch [1/10], Step [151/256], Loss: 2.2191, GPU Mem: 2142.0
Epoch [1/10], Step [161/256], Loss: 2.0440, GPU Mem: 2142.0
Epoch [1/10], Step [171/256], Loss: 1.9828, GPU Mem: 2142.0
```

In [11]:

```
run_train(64, epochs=epochs)
```

```
Training model with batch size 64 and lr 0.0016.
Epoch [1/10], Step [1/64], Loss: 2.2998, GPU Mem: 4151.0
Epoch [1/10], Step [11/64], Loss: 2.2376, GPU Mem: 4231.0
Epoch [1/10], Step [21/64], Loss: 2.1713, GPU Mem: 4212.0
Epoch [1/10], Step [31/64], Loss: 2.1049, GPU Mem: 3957.0
Epoch [1/10], Step [41/64], Loss: 2.0591, GPU Mem: 3963.0
Epoch [1/10], Step [51/64], Loss: 2.1155, GPU Mem: 3959.0
Epoch [1/10], Step [61/64], Loss: 1.9134, GPU Mem: 3959.0
Testing model.
Accuracy of the model on train images: 25.9521484375 %
Epoch [2/10], Step [1/64], Loss: 2.0261, GPU Mem: 4133.0
Epoch [2/10], Step [11/64], Loss: 1.8944, GPU Mem: 4176.0
Epoch [2/10], Step [21/64], Loss: 1.8448, GPU Mem: 4144.0
Epoch [2/10], Step [31/64], Loss: 1.9030, GPU Mem: 4126.0
Epoch [2/10], Step [41/64], Loss: 2.0297, GPU Mem: 3996.0
Epoch [2/10], Step [51/64], Loss: 1.8204, GPU Mem: 3991.0
Epoch [2/10], Step [61/64], Loss: 1.8499, GPU Mem: 3986.0
Testing model.
Accuracy of the model on train images: 31.689453125 %
Epoch [3/10], Step [1/64], Loss: 1.5081, GPU Mem: 3993.0
Epoch [3/10], Step [11/64], Loss: 1.7703, GPU Mem: 3993.0
Epoch [3/10], Step [21/64], Loss: 1.7576, GPU Mem: 3988.0
Epoch [3/10], Step [31/64], Loss: 1.6909, GPU Mem: 3988.0
Epoch [3/10], Step [41/64], Loss: 1.6434, GPU Mem: 3986.0
Epoch [3/10], Step [51/64], Loss: 1.7583, GPU Mem: 3992.0
Epoch [3/10], Step [61/64], Loss: 1.7846, GPU Mem: 3992.0
Testing model.
Accuracy of the model on train images: 32.1533203125 %
Epoch [4/10], Step [1/64], Loss: 1.6611, GPU Mem: 3986.0
Epoch [4/10], Step [11/64], Loss: 1.5809, GPU Mem: 3986.0
Epoch [4/10], Step [21/64], Loss: 1.5279, GPU Mem: 3987.0
Epoch [4/10], Step [31/64], Loss: 1.6131, GPU Mem: 3994.0
Epoch [4/10], Step [41/64], Loss: 1.7628, GPU Mem: 3736.0
Epoch [4/10], Step [51/64], Loss: 1.7254, GPU Mem: 3797.0
Epoch [4/10], Step [61/64], Loss: 1.8396, GPU Mem: 3967.0
Testing model.
Accuracy of the model on train images: 32.12890625 %
Epoch [5/10], Step [1/64], Loss: 1.5448, GPU Mem: 4346.0
Epoch [5/10], Step [11/64], Loss: 1.6125, GPU Mem: 4346.0
Epoch [5/10], Step [21/64], Loss: 1.4955, GPU Mem: 4346.0
Epoch [5/10], Step [31/64], Loss: 1.5518, GPU Mem: 4346.0
Epoch [5/10], Step [41/64], Loss: 1.5418, GPU Mem: 4346.0
Epoch [5/10], Step [51/64], Loss: 1.4727, GPU Mem: 4346.0
Epoch [5/10], Step [61/64], Loss: 1.4729, GPU Mem: 4346.0
Testing model.
Accuracy of the model on train images: 35.7666015625 %
Epoch [6/10], Step [1/64], Loss: 1.4136, GPU Mem: 4347.0
Epoch [6/10], Step [11/64], Loss: 1.4385, GPU Mem: 4347.0
Epoch [6/10], Step [21/64], Loss: 1.5503, GPU Mem: 4347.0
Epoch [6/10], Step [31/64], Loss: 1.4571, GPU Mem: 4347.0
Epoch [6/10], Step [41/64], Loss: 1.5516, GPU Mem: 4347.0
Epoch [6/10], Step [51/64], Loss: 1.3277, GPU Mem: 4347.0
Epoch [6/10], Step [61/64], Loss: 1.2421, GPU Mem: 4347.0
Testing model.
Accuracy of the model on train images: 43.603515625 %
Epoch [7/10], Step [1/64], Loss: 1.3208, GPU Mem: 4347.0
Epoch [7/10], Step [11/64], Loss: 1.2852, GPU Mem: 4347.0
Epoch [7/10], Step [21/64], Loss: 1.1607, GPU Mem: 4347.0
Epoch [7/10], Step [31/64], Loss: 1.5193, GPU Mem: 4347.0
Epoch [7/10], Step [41/64], Loss: 1.3003, GPU Mem: 4347.0
Epoch [7/10], Step [51/64], Loss: 1.1684, GPU Mem: 4347.0
```

```
Epoch [7/10], Step [61/64], Loss: 1.3704, GPU Mem: 4347.0
Testing model.
Accuracy of the model on train images: 47.1435546875 %
Epoch [8/10], Step [1/64], Loss: 1.3224, GPU Mem: 4347.0
Epoch [8/10], Step [11/64], Loss: 1.3766, GPU Mem: 4347.0
Epoch [8/10], Step [21/64], Loss: 1.2777, GPU Mem: 4347.0
Epoch [8/10], Step [31/64], Loss: 1.1669, GPU Mem: 4323.0
Epoch [8/10], Step [41/64], Loss: 1.2048, GPU Mem: 4323.0
Epoch [8/10], Step [51/64], Loss: 1.1923, GPU Mem: 4323.0
Epoch [8/10], Step [61/64], Loss: 1.3048, GPU Mem: 4323.0
Testing model.
Accuracy of the model on train images: 54.7119140625 %
Epoch [9/10], Step [1/64], Loss: 1.3521, GPU Mem: 4337.0
Epoch [9/10], Step [11/64], Loss: 1.2410, GPU Mem: 4337.0
Epoch [9/10], Step [21/64], Loss: 1.0606, GPU Mem: 4337.0
Epoch [9/10], Step [31/64], Loss: 1.2307, GPU Mem: 4337.0
Epoch [9/10], Step [41/64], Loss: 1.3679, GPU Mem: 4338.0
Epoch [9/10], Step [51/64], Loss: 0.9634, GPU Mem: 4338.0
Epoch [9/10], Step [61/64], Loss: 1.1704, GPU Mem: 4338.0
Testing model.
Accuracy of the model on train images: 48.9990234375 %
Epoch [10/10], Step [1/64], Loss: 1.1062, GPU Mem: 4346.0
Epoch [10/10], Step [11/64], Loss: 1.1290, GPU Mem: 4347.0
Epoch [10/10], Step [21/64], Loss: 1.0115, GPU Mem: 4347.0
Epoch [10/10], Step [31/64], Loss: 0.8880, GPU Mem: 4347.0
Epoch [10/10], Step [41/64], Loss: 0.9070, GPU Mem: 4347.0
Epoch [10/10], Step [51/64], Loss: 1.0558, GPU Mem: 4347.0
Epoch [10/10], Step [61/64], Loss: 1.2961, GPU Mem: 4349.0
Testing model.
Accuracy of the model on train images: 49.2919921875 %
```

In [12]:

```
run_train(256, epochs=epochs)
```

```
Training model with batch size 256 and lr 0.0064.
```

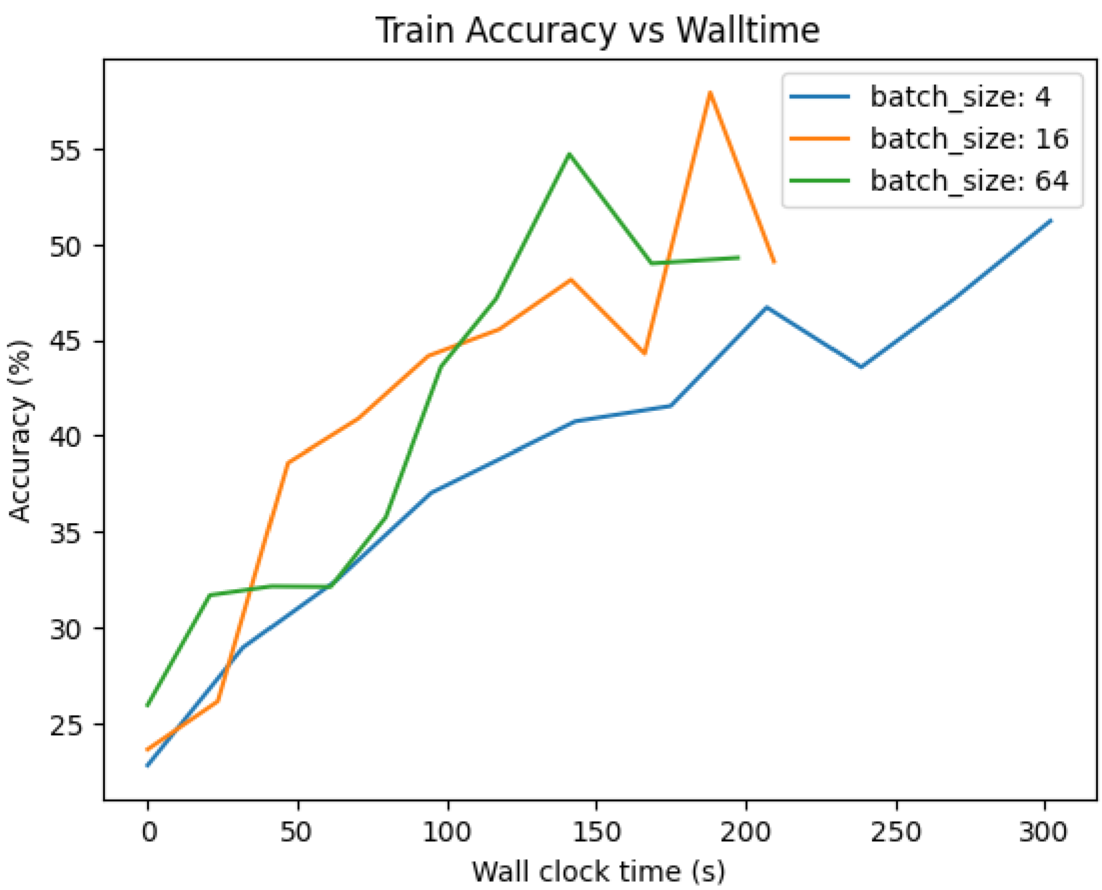
```
OutOfMemoryError: CUDA out of memory. Tried to allocate 50.00 MiB (GPU 0; 6.0
0 GiB total capacity; 5.29 GiB already allocated; 0 bytes free; 5.33 GiB reserved
in total by PyTorch) If reserved memory is >> allocated memory try setting max_sp
lit_size_mb to avoid fragmentation.  See documentation for Memory Management and
PYTORCH_CUDA_ALLOC_CONF
```

Plot the loss, accuracy and memory utilization

Once all logs have been generated under `/content/cs182hw4/logs`, run the cell below to plot loss and accuracy against wall clock time.

In [14]:

```
# Plotting scripts
def get_df(batch_size):
    path = ROOT_PATH + f'logs/resnet18__{batch_size}.csv'
    assert os.path.exists(path), f'Memory profile not found for batch size {batch_s
    df = pd.read_csv(path)
    # Create a wall time column
    df['walltime'] = df['timestamp'] - df['timestamp'].iloc[0]
    return df

def plot_walltime_acc(batch_sizes):
    plt.figure()
    for bs in batch_sizes:
        df=get_df(bs)
        plt.plot(df['walltime'], df['accuracy'], label=f'batch_size: {bs}')
    plt.xlabel('Wall clock time (s)')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.title('Train Accuracy vs Walltime')
    plt.show()

def print_mem_usage(batch_sizes):
    print("\n====== Memory Usage for different batch sizes =======")
    for bs in batch_sizes:
        df=get_df(bs)
        mem_usage = df['memUsage'].iloc[-1]
        print(f'{bs}\t: {mem_usage} MB')

#batch_sizes = [4, 16, 64, 256]
batch_sizes = [4, 16, 64]
plot_walltime_acc(batch_sizes)
print_mem_usage(batch_sizes)
```



```
====== Memory Usage for different batch sizes =======
4       : 1684.0 MB
16      : 2325.0 MB
64      : 4349.0 MB
```

Questions (answer in written submission)

3a. What is the memory utilization for different batch sizes (4, 16, 64, 256)? What is the largest batch size you were able to train?

3b. Which batch size gave you the highest accuracy at the end of 10 epochs?

3c. Which batch size completed 10 epochs the fastest (least wall clock time)? Why?

3d. Attach your training accuracy vs wall time plots with your written submission.