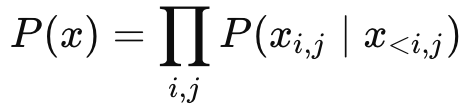
For each practical exercise (TP), please work in groups of two or three. Then, create a **private GitHub repository** and add me (my GitHub is **arthur-75**) to your project. Finally, share the link to your project (or TP) under [Practical Exercises](https://docs.google.com/spreadsheets/d/1V-YKgHn71FnwjoFltDhWsPJS7uIuAh9lj6SP2DSCvlY/edit?usp=sharing) and make sure to choose your **team name** :-)

# **Autoregressive Models (PixelCNN)**

## **What is an Autoregressive Model in PixelCNN?**

Autoregressive models aim to learn the joint distribution of images by factorizing it into conditional distributions. For an image x, composed of pixels xi,j​:



where:

* x<i,j​ refers to all pixels above and to the left of pixel xi,j.
* PixelCNN strictly enforces this autoregressive dependency using **masked convolutions**.

## **The PixelCNN Architecture Explained**

PixelCNN generates images pixel-by-pixel, modeling the conditional probability distribution of each pixel based on previously generated pixels.

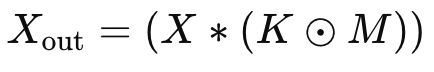
* **Masked Convolution Layers** These ensure no pixel sees "future" pixels (pixels below or to the right), preserving the autoregressive property.
* **Residual Blocks** Improve training stability and performance through skip connections.

## **Behind Masked Convolution**

### **Masked convolution filters enforce conditional dependence:**

* Type A mask: Used in the first convolutional layer (no current pixel allowed).
* Type B mask: Used in subsequent layers (allows current pixel).

Given a convolutional kernel K, mask MMM, input image X:



Where:

* ⊙ represents element-wise multiplication.
* Mask M ensures the convolution at pixel (i,j) does not include pixels at positions after (i,j).

## **Implementation**

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

from torchvision import datasets, transforms

from torch.utils.data import DataLoader

import matplotlib.pyplot as plt

import numpy as np

from tqdm import tqdm

import matplotlib.pyplot as plt  
# Parameters

IMAGE\_SIZE = 16

PIXEL\_LEVELS = 4

N\_FILTERS = 128

RESIDUAL\_BLOCKS = 5

BATCH\_SIZE = 128

EPOCHS = 150

DEVICE ="mps" #torch.device("cuda" if torch.cuda.is\_available() else "cpu")



**Data Preparation**

* Load FashionMNIST dataset.
* Resize images to smaller dimensions (e.g., 16×16).
* Quantize pixel values into fewer discrete levels (e.g., 4 levels) to simplify the probability distribution.

# Data Preparation

transform = transforms.Compose([

transforms.Resize((IMAGE\_SIZE, IMAGE\_SIZE)),

transforms.ToTensor(),

transforms.Lambda(lambda x: (x \* 255).float() // (256 // PIXEL\_LEVELS))

])

path= xxx

dataset = datasets.FashionMNIST(root=path, train=True, download=True, transform=transform)

data\_loader = DataLoader(dataset, batch\_size=BATCH\_SIZE, shuffle=True)



**Masked Convolution Layer**

* Implement convolutional layers with masks:
  + **Type A**: For the initial layer, exclude the pixel itself.
  + **Type B**: For all subsequent layers, allow the current pixel.

# Masked Convolution Layer

class MaskedConv2d(nn.Conv2d):

def \_\_init\_\_(self, mask\_type, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.register\_buffer('mask', self.weight.data.clone())

\_, \_, h, w = self.weight.size()

self.mask.fill\_(1)

self.mask[:, :, h // 2, w // 2 + (mask\_type == 'B'):] = 0

self.mask[:, :, h // 2 + 1:] = 0

def forward(self, x):

self.weight.data \*= self.mask

return super().forward(x)



**Residual Blocks**

* Combine masked convolutions and non-linear activations with skip-connections:

# Residual Block

class ResidualBlock(nn.Module):

def \_\_init\_\_(self, filters):

super().\_\_init\_\_()

self.block = nn.Sequential(

nn.ReLU(),

nn.Conv2d(filters, filters // 2, kernel\_size=1),

nn.ReLU(),

MaskedConv2d('B', filters // 2, filters // 2, kernel\_size=3, padding=1),

nn.ReLU(),

nn.Conv2d(filters // 2, filters, kernel\_size=1)

)

def forward(self, x):

return x + self.block(x)



**PixelCNN Model Structure**

* Stack masked convolution layers and residual blocks.
* Output layer predicts probability distribution over pixel intensities using softmax activation.

# PixelCNN Model

class PixelCNN(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.model = nn.Sequential(

MaskedConv2d('A', 1, N\_FILTERS, kernel\_size=7, padding=3),

\*[ResidualBlock(N\_FILTERS) for \_ in range(RESIDUAL\_BLOCKS)],

nn.ReLU(),

MaskedConv2d('B', N\_FILTERS, N\_FILTERS, kernel\_size=1),

nn.ReLU(),

MaskedConv2d('B', N\_FILTERS, N\_FILTERS, kernel\_size=1),

nn.ReLU(),

nn.Conv2d(N\_FILTERS, PIXEL\_LEVELS, kernel\_size=1)

)

def forward(self, x):

return self.model(x)

model = PixelCNN().to(DEVICE)



## **📉 5. Training the PixelCNN**

### **Objective:**

Minimize the cross-entropy loss between predicted pixel distributions and actual pixel intensities.

**Cross-Entropy Loss**:``



* yc​ is the true class label (pixel intensity category).
* xc is the predicted probability for class c.

### **Training Loop (Conceptual):**

* Forward pass:
  + Input: images (floats).
  + Output: predicted probabilities for each pixel intensity.
* Compute cross-entropy loss.
* Backward pass and optimizer update (e.g., Adam optimizer).

optimizer = optim.Adam(model.parameters(), lr=0.0005)

criterion = nn.CrossEntropyLoss()

# Training Loop

for epoch in range(EPOCHS):

model.train()

total\_loss = 0

for images, \_ in tqdm(data\_loader, desc=f"Epoch {epoch+1}/{EPOCHS}"):

images = images.to(DEVICE).squeeze(1)

optimizer.zero\_grad()

inputs = images.float().unsqueeze(1) # [batch, 1, H, W]

targets = images.long() # [batch, H, W]

outputs = model(inputs) # [batch, PIXEL\_LEVELS, H, W]

loss = criterion(outputs, targets) # CrossEntropy expects float inputs and long targets

loss.backward()

optimizer.step()

total\_loss += loss.item()

avg\_loss = total\_loss / len(data\_loader)

print(f"Epoch {epoch+1}, Loss: {avg\_loss:.4f}")

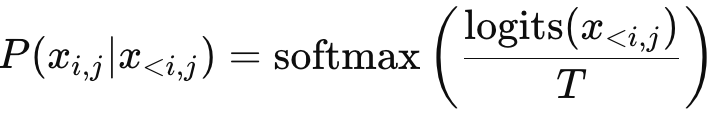


## **Sampling (Generating Images with PixelCNN)**

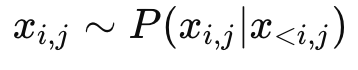
PixelCNN samples pixels sequentially from the learned distribution:

### **Sampling Steps:**

1. **Initialize**: Create an empty (zero-valued) image tensor.
2. **Sequential Sampling**: Iterate through each pixel position (i,j):
   * Compute conditional probability:



* + Sample pixel intensity from the probability distribution:



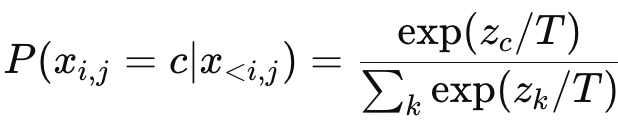
* + Update pixel value in the generated image.

### **Role of Temperature T:**

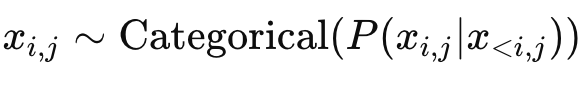
* Controls randomness:
  + Lower T → less randomness, sharper images.
  + Higher T → more randomness, diverse images.

### **Formally:**

For each pixel, given logits z:



Then we sample from this distribution:



def generate\_images(model, num\_images, temperature=1.0):

model.eval()

generated = torch.zeros(num\_images, 1, IMAGE\_SIZE, IMAGE\_SIZE, device=DEVICE)

with torch.no\_grad():

for row in range(IMAGE\_SIZE):

for col in range(IMAGE\_SIZE):

logits = model(generated.float())[:, :, row, col] / temperature

probs = torch.softmax(logits, dim=-1)

generated[:, 0, row, col] = torch.multinomial(probs, 1).squeeze(-1)

return generated.cpu().numpy() / PIXEL\_LEVELS

# Generate sample images

sample\_images = generate\_images(model, num\_images=10)

print("Generated images shape:", sample\_images.shape)



## **Visualizing Generated Images**

* Generated images have pixel values between 0 and 1.
* Display images using visualization libraries (e.g., matplotlib) to inspect model quality.

def plot\_generated\_images(images, n\_cols=5):

n\_rows = (len(images) + n\_cols - 1) // n\_cols

plt.figure(figsize=(2 \* n\_cols, 2 \* n\_rows))

for idx, img in enumerate(images):

plt.subplot(n\_rows, n\_cols, idx + 1)

plt.imshow(img.squeeze(), cmap='gray')

plt.axis('off')

plt.tight\_layout()

plt.show()

# Generate sample images (assuming you've done this step already)

sample\_images = generate\_images(model, num\_images=10, temperature=1.0)

# Plot generated images

plot\_generated\_images(sample\_images)

