MF20006: Introduction to Computer Science

Lecture 10: Artificial Intelligence II

Hui Xu

xuh@fudan.edu.cn

Outline

1. Neural Networks

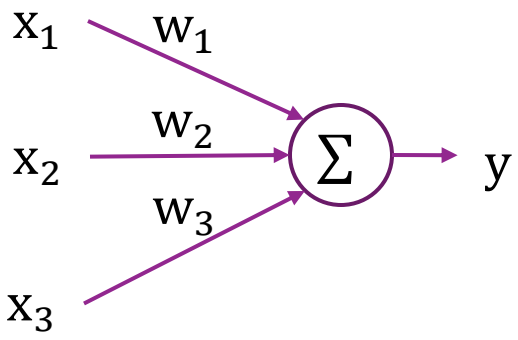
2. Convolutional Neural Networks

3. Generative CNN

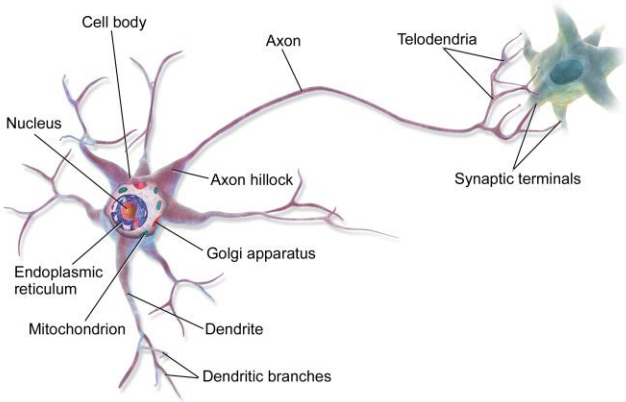
1. Neural Networks

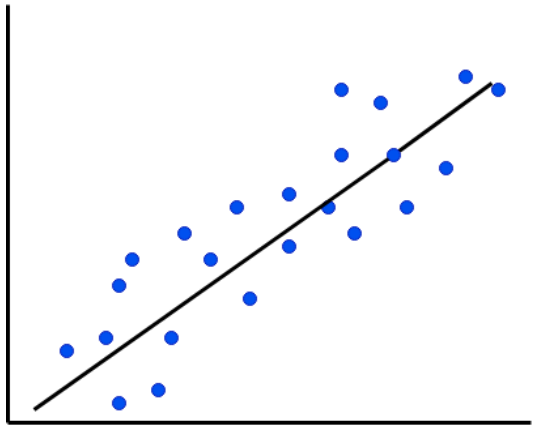
Perceptron

- One of the earliest models of a neuron in artificial intelligence.
- Introduced by Frank Rosenblatt (American psychologist) in 1958.



$$y = \sum_i w_i x_i + b$$

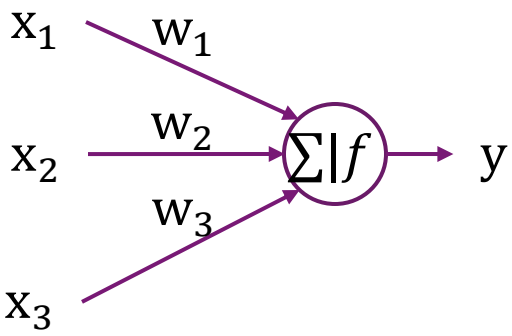




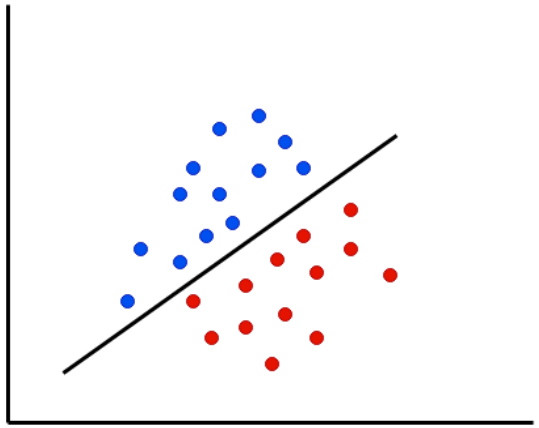
Linear Regression

General Form of Perceptron

❑Add an activation function f in the neuron.



$$y = f\left(\sum_i w_i x_i + b\right)$$



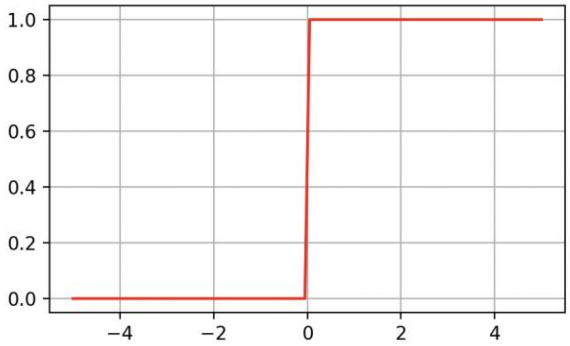
Linear Binary Classification

$$y = \begin{cases} 1, \text{if } \left(\sum_i w_i x_i + b\right) > t \\ 0, \text{otherwise} \end{cases}$$

Activation Functions

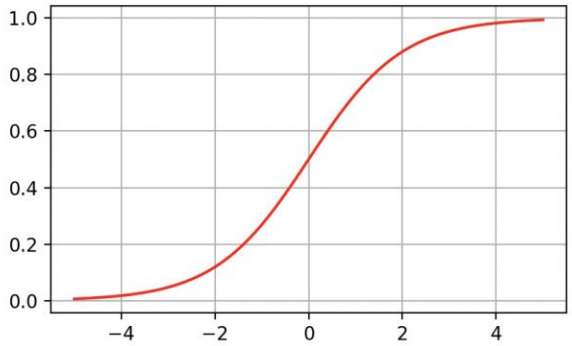
Step Function

$$y = \begin{cases} 1, \text{if } x > t \\ 0, \text{otherwise} \end{cases}$$



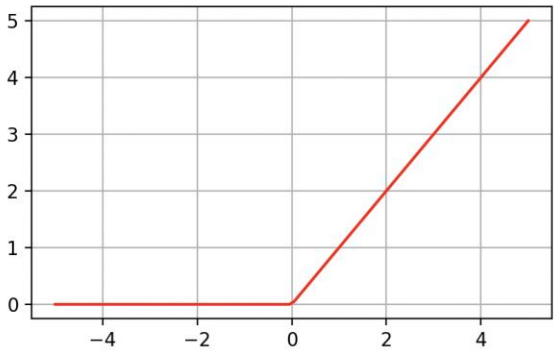
Sigmoid Function

$$y = \frac{1}{1 + e^{-x}}$$



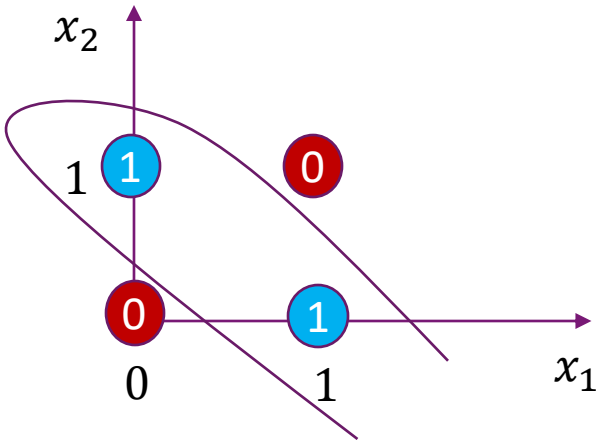
ReLU Function

$$y = \max(0, x)$$



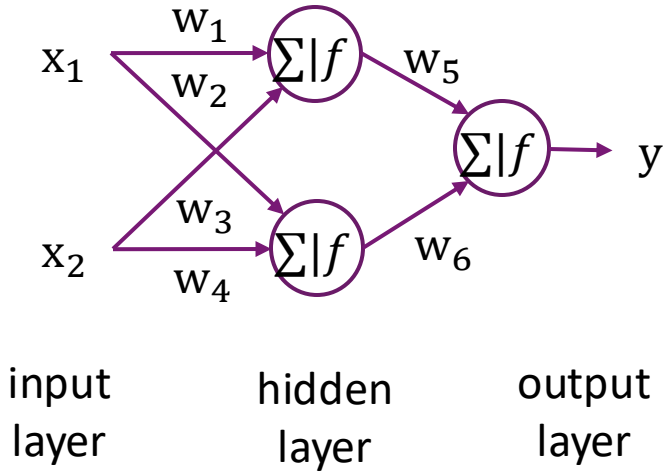
Limitation of Single-layer Perceptron

- A single-layer perceptron is essentially linear regression followed by a fixed nonlinearity.
- It cannot model general nonlinear functions.
- It can only classify linearly separable data.
  - For example, XOR is not linearly separable.
  - Even with common monotonic nonlinear activations (step, sigmoid, ReLU), the decision boundary remains a linear hyperplane.

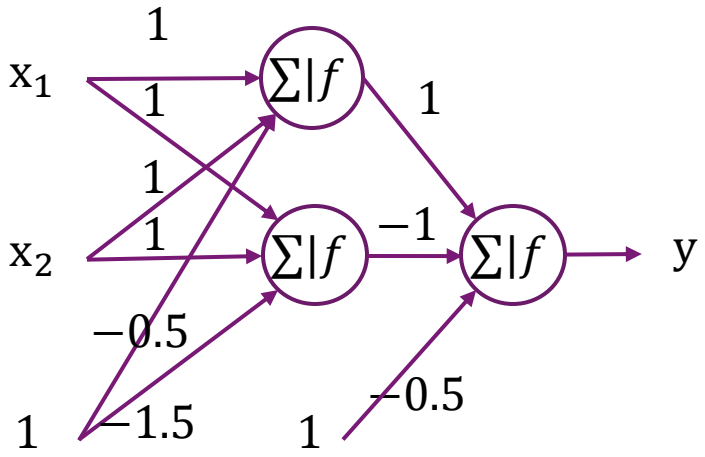


Multi-layer Perceptron

- **MLP stacks perceptron into layers:**
  - An input layer
  - One or multiple hidden layers
  - An output layer
- **With nonlinear activations in the hidden layers, the network composes nonlinearities and becomes capable of learning arbitrary nonlinear mappings.**



Example MLP for XOR



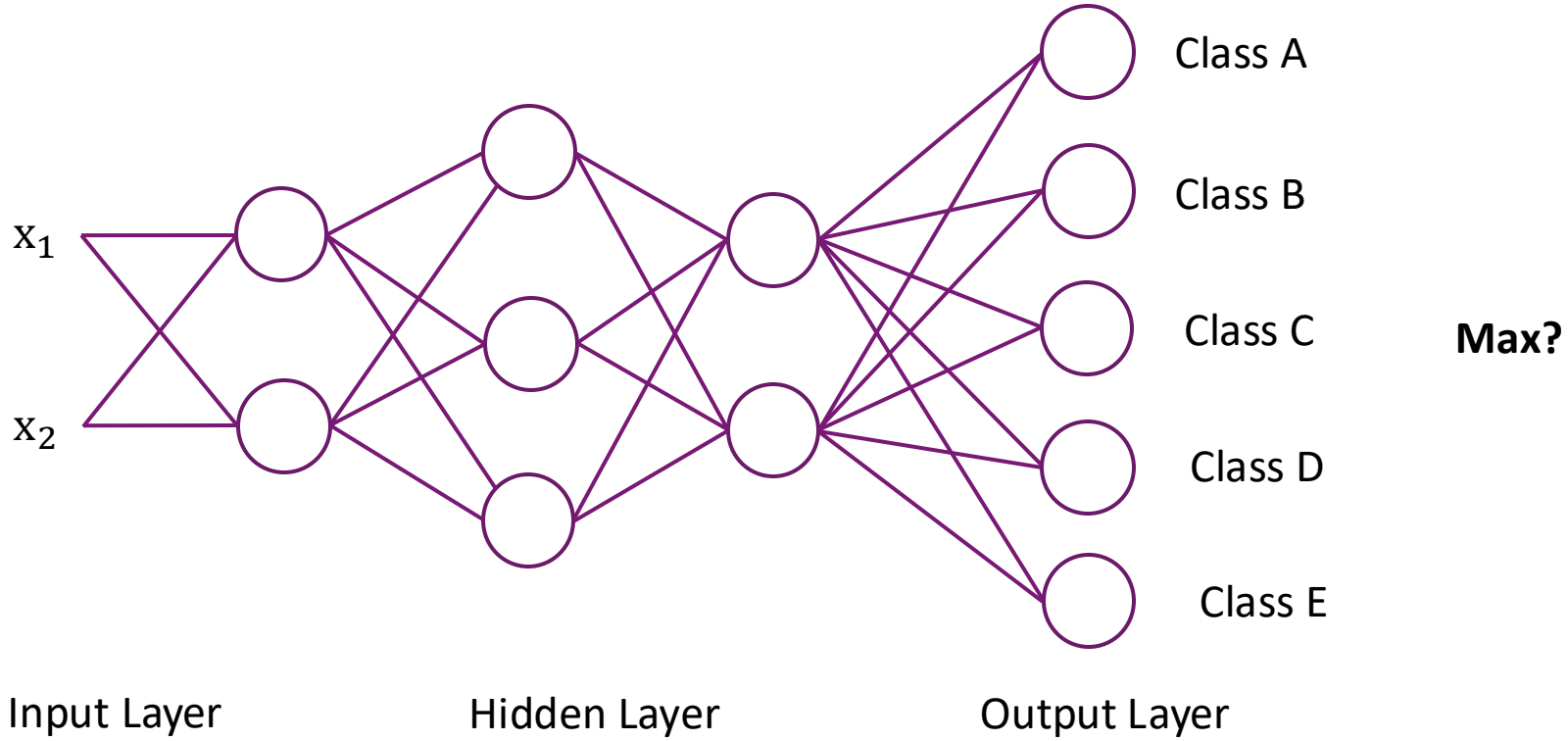
$$f(\mathrm{x}) = \begin{cases} 1, \text{if } \left(\sum_{\mathrm{i}} \mathrm{w_i x_i} > 0 \right) \\ 0, \text{otherwise} \end{cases}$$

$$y = f(f(x_1 * 1 + x_2 * 1 - 1 * 0.5) * 1 - f(x_1 * 1 + x_2 * 1 - 1 * 1.5) * 1 - 1 * 0.5)$$

1) When $x_1 = 0, x_2 = 0,\ y = f(0 - 0 - 0.5) = 0$
2) When $x_1 = 1, x_2 = 0,\ y = f(1 - 0 - 0.5) = 1$
3) When $x_1 = 0, x_2 = 1,\ y = f(1 - 0 - 0.5) = 1$
4) When $x_1 = 1, x_2 = 1,\ y = f(1 - 1 - 0.5) = 0$

Multi-class Classification

- The output layer contains one neuron per class.
- The network produces a vector of logits.
- The class with the largest logit is selected.



Loss Function for Classification

❑Cross entropy: distance between:

- ➢the true label distribution y
- ➢the predicted distribution p

$$CrossEntropy(y,p) = -\sum_{i=1}^{C} y_i \log(p_i)$$

y is one-hot, y_k = 1, others = 0

$$CrossEntropy(y,p) = -\log(p_k)$$

Softmax

- Softmax is used in multi-class classification to convert a vector of logits (raw scores) into probabilities.
- Input: $z = (z_1, z_2, \ldots z_n)$
- Output:

$$\text{Softmax}(z_k) = \frac{e^{z_k}}{\sum_{i=i}^{n} e^{z_i}}$$

- Exponentials make big logits grow faster, amplifying differences.

Training: Backpropagation

- **How to determine the value of w_i ?**
- **Randomly initialize model parameters and then tune.**
  - by comparing the output with the ground truth.
- **Adjust the value of parameters to reduce the loss.**
- **Use the chain rule to propagate gradients layer by layer, backward.**

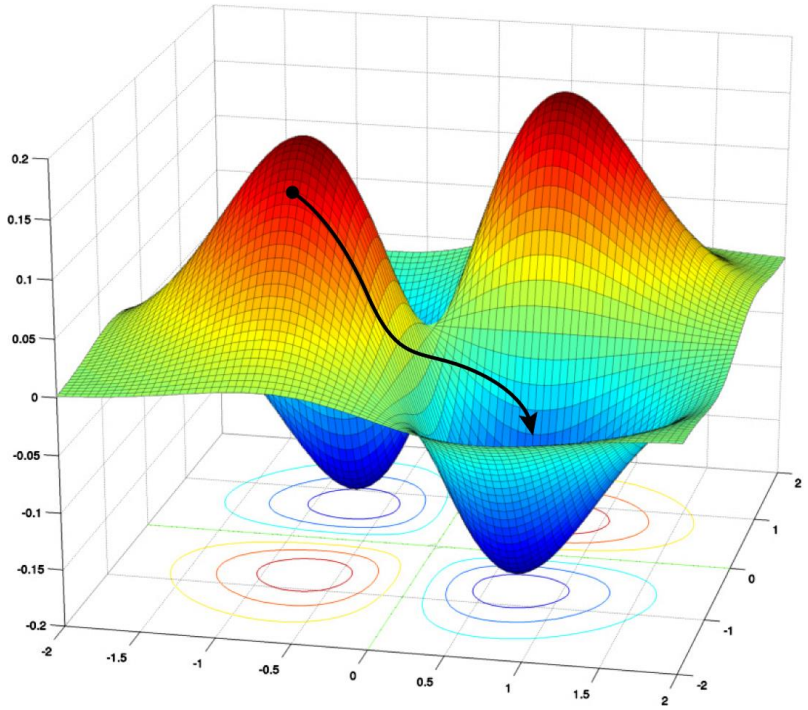
Example model: $y = w_1 x_1 + w_2 x_2$

Randomly init as: $y = -1 * x_1 + 1 * x_2$

Observed data:	**Prediction:**	**Loss (Squared):**
$([x_1, x_2],\ y)$	$([x_1, x_2,],\ y)$	
([1,2], 3)	([1,2], 1)	4
…	…	

Training: Gradient Descent

- The idea is to minimize the loss as fast as possible
- Follow the slope (partial derivative) downhill on the loss surface



The slop for w_i: $\frac{\partial L}{\partial w_i} = 2(y' - y)x_i$

- $w_1' = w_1 - 2\eta\, x_1(y' - y)$
- $w_2' = w_2 - 2\eta\, x_2(y' - y)$

Batch Gradient Descent

- We have multiple data samples for training.
- Each update requires one full pass over all samples.

$$w_{t+1} = w_{t+1} - \frac{1}{n}\eta \sum_{i=1}^{n} \nabla L(w, x_i, y_i)$$

- Pros: Optimize with global information => stable convergence
- Cons: It is slow when the dataset is large.

SGD: Stochastic Gradient Descent

❑Use one sample or mini-batch to approximate the gradient.

- ➢Much faster than full gradient descent, but individual-sample updates are noisy.

$$w_{t+1} = w_t - \nabla L(w, x_i, y_i)$$

❑In practice, mini-batch (32-512 samples) SGD is commonly used.

- ➢Better gradient estimate, yet faster than full GD.

$$w_{t+1} = w_{t+1} - \frac{1}{B}\eta \sum_{i=1}^{B} \nabla L(w, x_i, y_i)$$

Issues of Mini-batch SGD

- **Sensitive to the choice of learning rate.**
  - Too large → divergence;
  - Too small → very slow convergence.
- **Update may oscillate in steep directions.**
  - The mini-batch may contain some noisy data points that are misleading.
  - In steep regions, gradients are large.
  - The parameter updates will be large but in inconsistent directions.
- **Slow progress towards convergence in flat regions.**
  - Gradients are tiny, updates become extremely small.

Introduce Moment (Momentum) during Update

- Consider the historical gradient during parameter update.
- To smooth the gradient and reduce noise.
- Exponentially decaying average of the past gradients.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla L(w)$$

- Bias correction (during cold start).

$$\hat{m}_t = \frac{m_t}{1 - \beta_1}$$

Diederik P. Kingma, Jimmy Lei Ba, "Adam: A method for stochastic optimization." *ICLR 2015.*

Adaptive Learning Rate for Each Parameter

- Large gradient => Small learning rate.
- Small gradient => Large learning rate.

$$\hat{\eta}_i = \frac{\eta}{f\left(\nabla L(w_i)\right)}$$

$$v_t = f\left(\nabla L(w_i)\right) = \sqrt{\beta_2 v_{t-1} + (1-\beta_2)\left(\nabla L(w)\right)^2} + \varepsilon$$

- Bias correction (during cold start).

$$\hat{v}_t = \frac{v_t}{1-\beta_2} = \frac{\beta_2}{1-\beta_2} v_{t-1} + \left(\nabla L(w)\right)^2$$

Adam (Adaptive Moment Estimation) Optimizer

- The most commonly used optimizer.
- Combines ideas from momentum-based optimization and adaptive learning rates.

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t$$

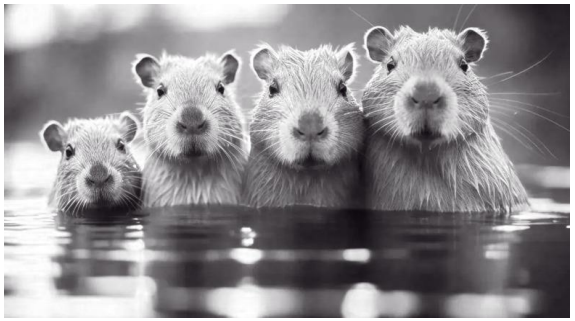
Commonly used value of hyper parameters:

$$\beta_2 = 0.9$$
$$\beta_2 = 0.999$$
$$\eta = 0.001$$
$$\varepsilon = 10^{-8}$$

2. Convolutional Neural Networks

How to Learn from Images

- **Images are represented with pixels (*e.g.*, 1920×1080).**
- **Each pixel has values:**
  - For grayscale images: a single intensity value.
  - For color images: three values, corresponding to the red, green, and blue channels.
- **MLP may not be able to achieve good result.**



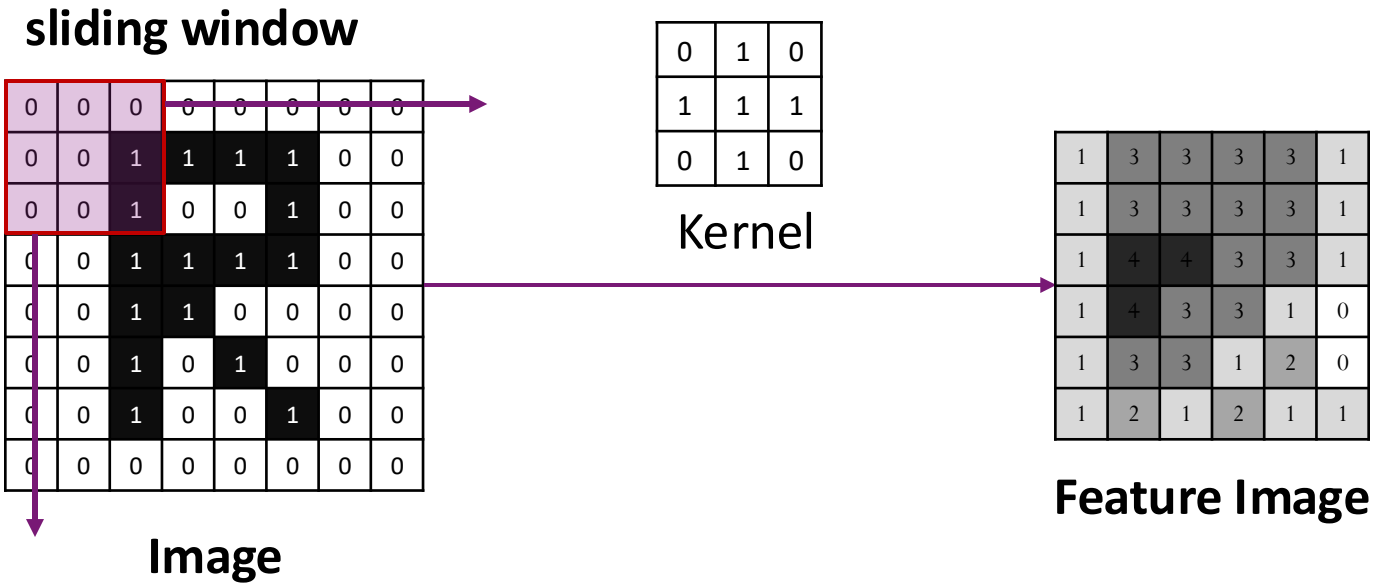


Convolutional Neural Networks: Overall Idea

- **A deep learning model mainly used for images and spatial data.**
- **Objective: Automatically learning patterns (features) like shapes and textures, and then combining them into higher-level concepts.**
  - Versus traditional feature extraction
- **How: sliding a small window (kernel) across the image, computing a weighted sum at each step.**

Convolution Kernel

- A kernel is a small matrix, *e.g.*, 3×3 or 5×5.
- It highlights a certain pattern.

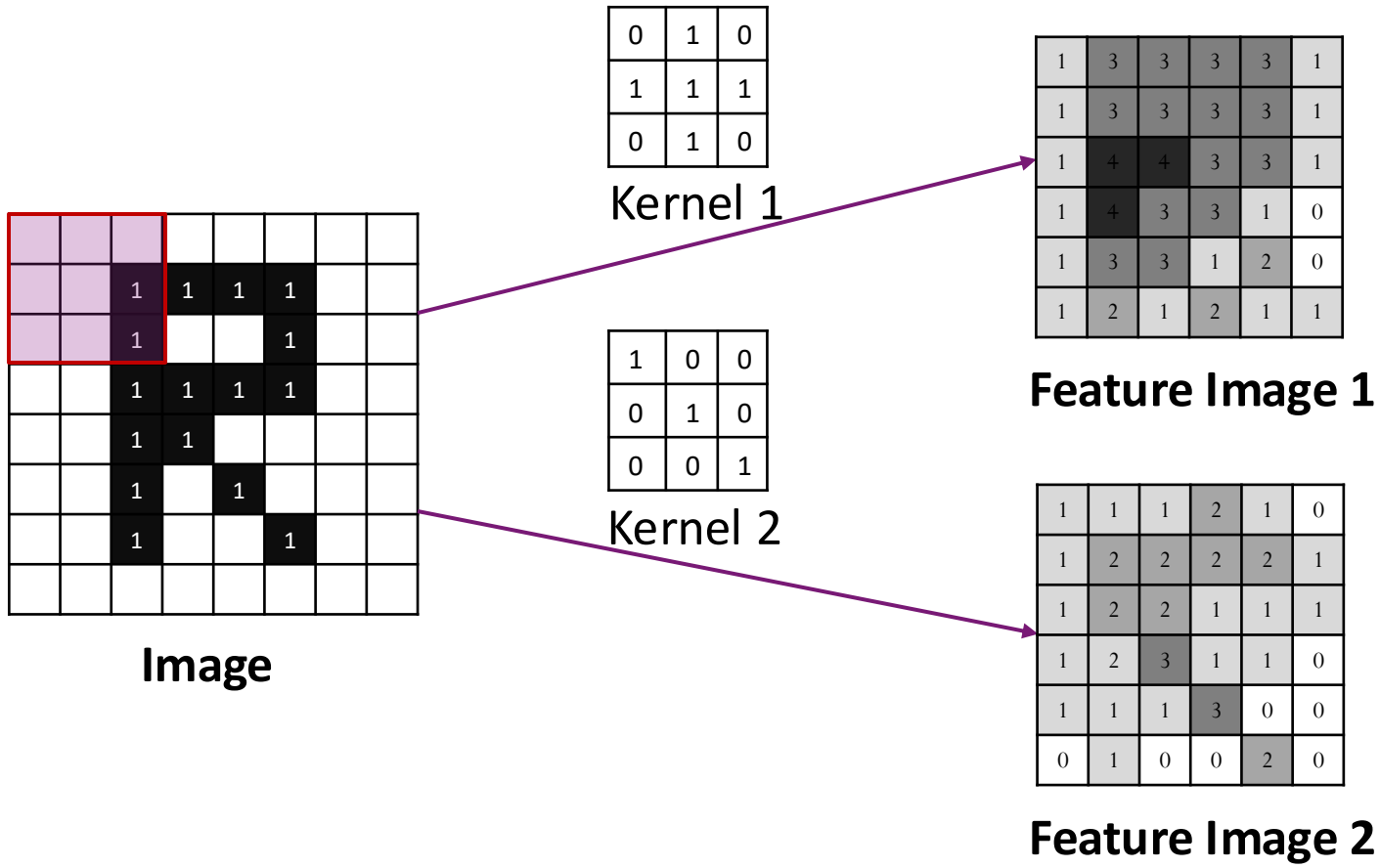


- If the input image is I and the kernel is K (size: $h \times w$), the output feature map O is calculated as (stride = 1):

$$O(m,n) = \sum_{i}^{h} \sum_{j}^{w} I(m+i, n+j) \times K(i,j)$$

Multiple Output Channels via Different Kernels

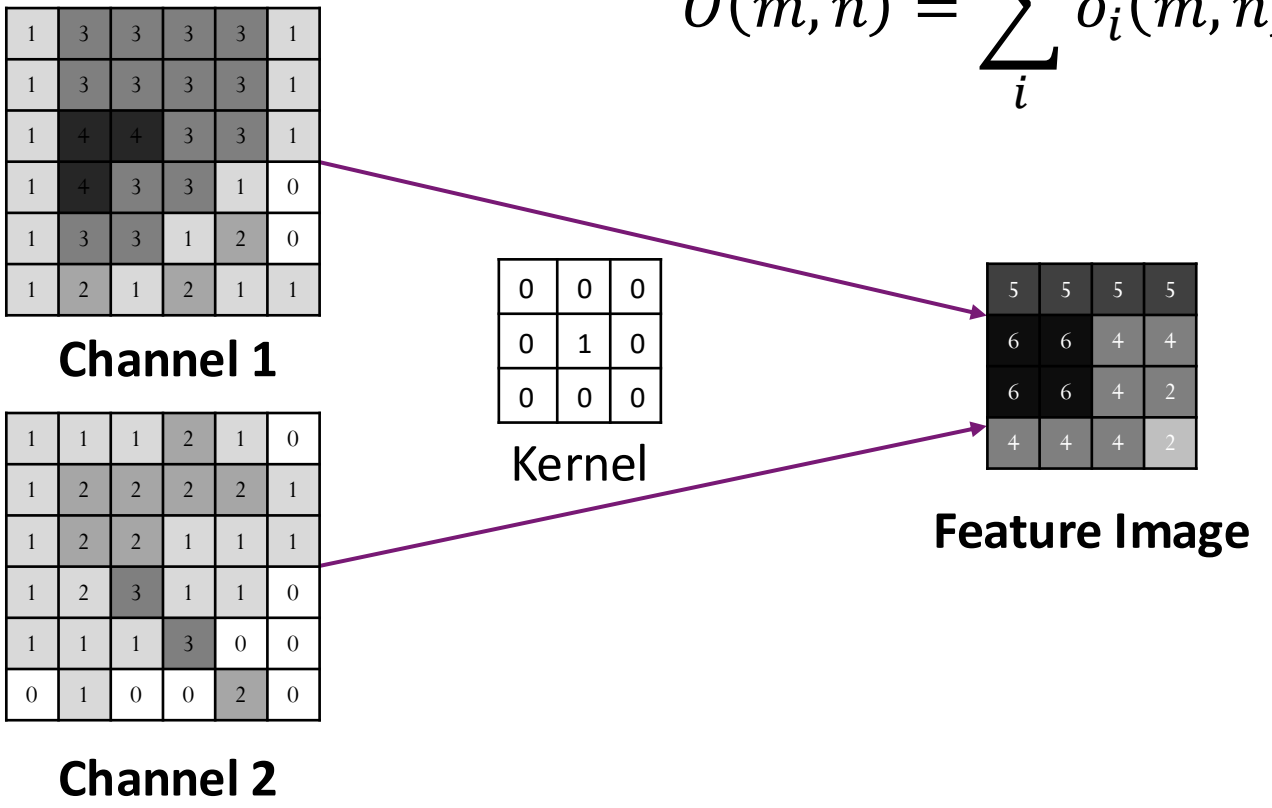
- Images contain many different patterns.
- Learn different concepts via different kernels.



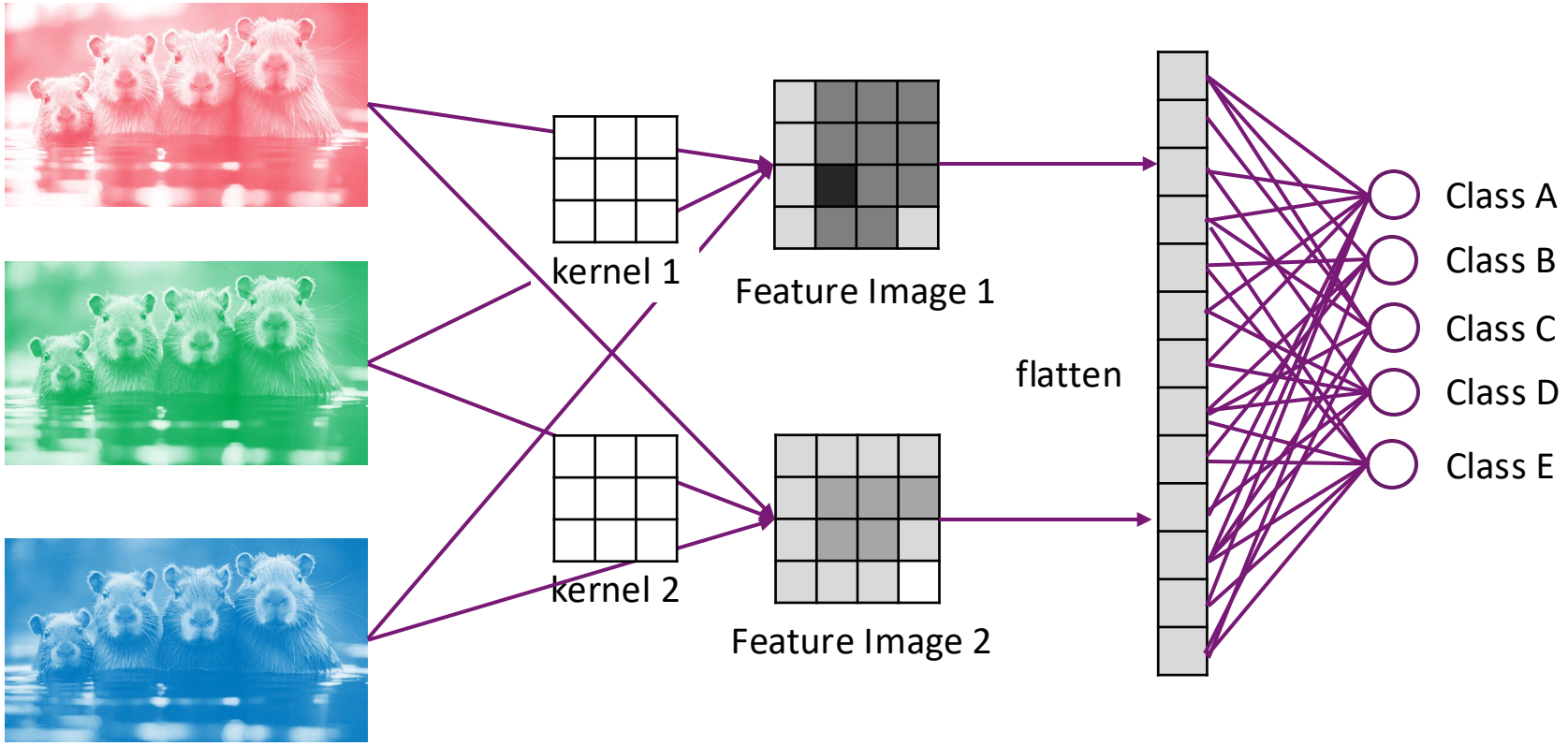
Multiple Input Channels

- A color image (RGB) has 3 channels.
- The hidden layers also have multiple channels.
- Each output channel is computed by accumulating contributions from all input channels.

$$O(m,n) = \sum_{i}^{c} o_i(m,n) + \beta$$

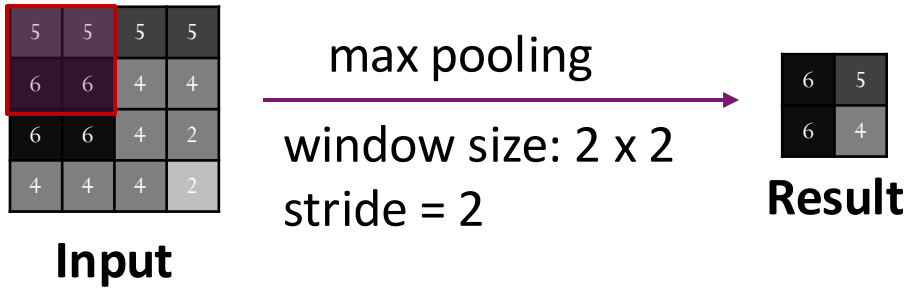


Wrap Up


kernel 1
Feature Image 1
kernel 2
Feature Image 2
flatten
Class A
Class B
Class C
Class D
Class E

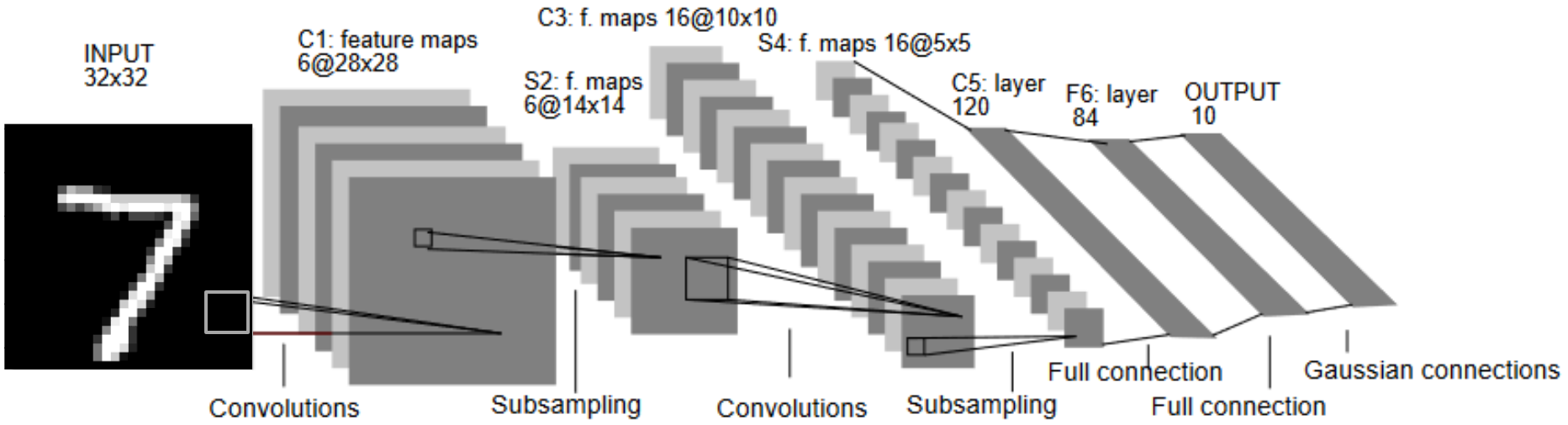
More Features: Pooling

- **Downsampling operation to reduce the size of feature maps.**
- **Two typical pooling methods:**
  - Max Pooling: Take the maximum value in the pooling window.
  - Average Pooling: Take the average value.



Famous CNN Model: LeNet5

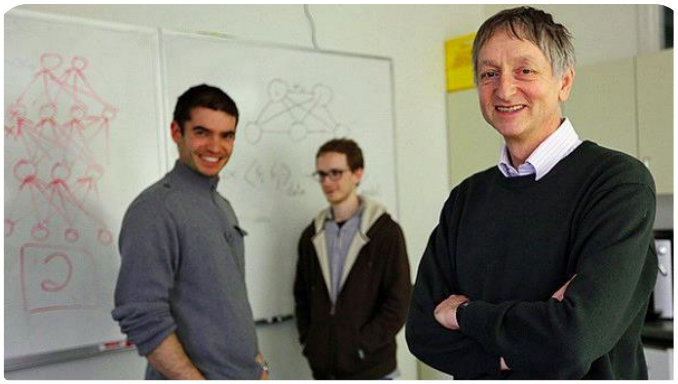
- One of the first CNNs, created by Yann LeCun
- It was designed for handwritten digit recognition (MNIST dataset).
- MNIST: 70,000 images of handwritten digits (0–9)
  - 60,000 training images, and 10,000 test images



Yann LeCun, et al. Handwritten digit recognition with a back-propagation network. NIPS 1989

Famous CNN Model: AlexNet

- Introduced by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton.
- It won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) that year by a huge margin, top5 error rate < 15.3%.
- ImageNet Dataset: 14 million images of 1000 categories.



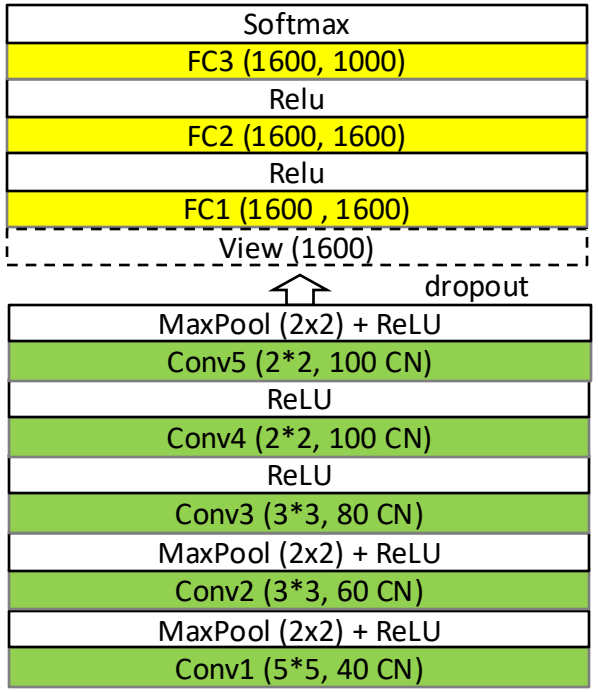
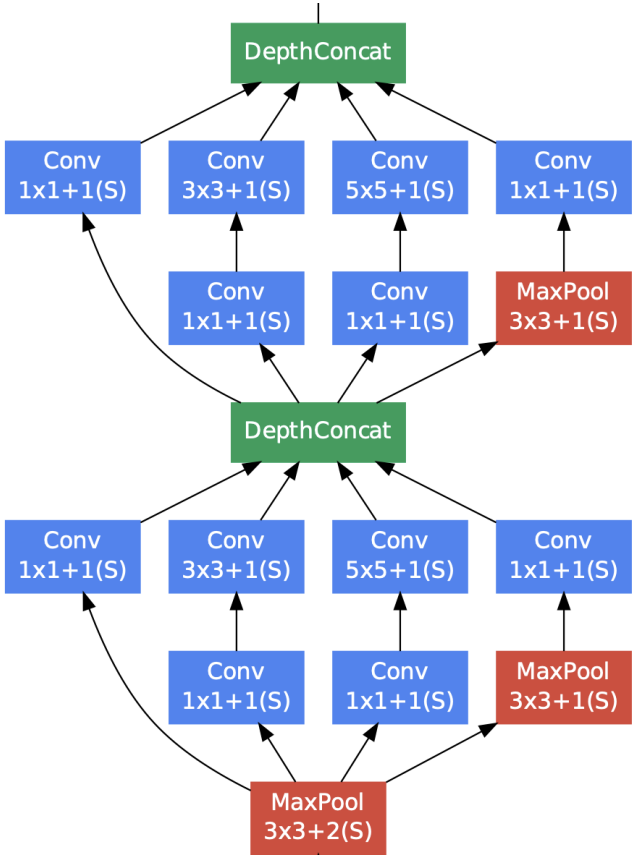


Image: 32*32, 3 CN

Alex Krizhevsky, *et al*, Imagenet classification with deep convolutional neural networks, NIPS 2012

GoogLeNet

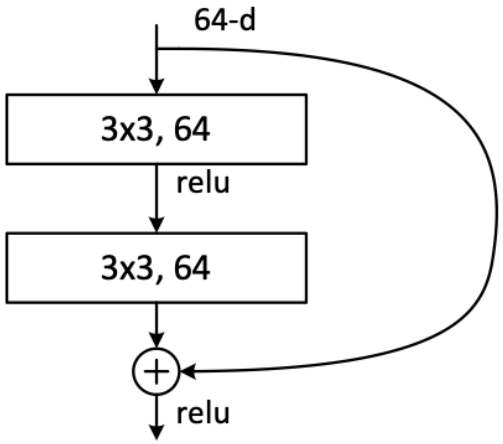
- **Before 2014, CNNs were mainly growing deeper and heavier.**
- **Problems:**
  - Too many parameters.
  - Deep networks cause vanishing gradients.
- **GoogLeNet introduces inception modules.**
  - Each inception module contains 4 branches.
  - Branch outputs are concatenated.
  - Reduce parameter size.

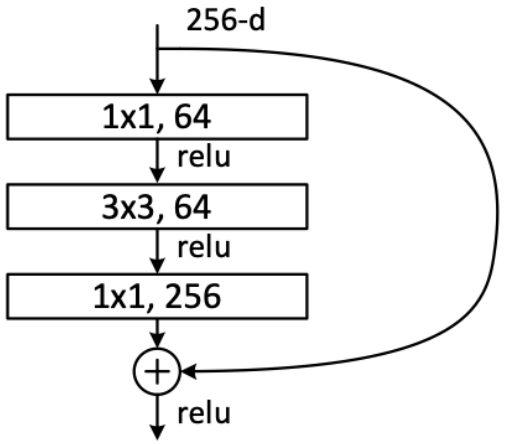


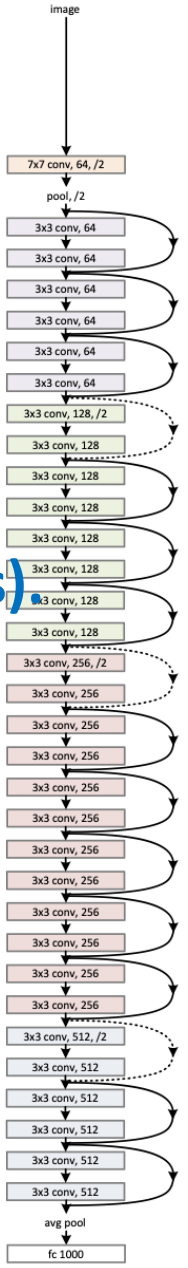
C. Szegedy, *et al.* "Going deeper with convolutions." CVPR 2015

Famous CNN Model: ResNet

- ❑Residual Network, introduced by Kaiming He *et al.*, 2015.
- ❑Won ImageNet 2015 with record-breaking accuracy > 95%
- ❑Deep networks suffer from vanishing gradient issue.
  - ➢Gradients get very small if the network is deep.
  - ➢Deeper networks sometimes perform worse than shallower ones.
- ❑ResNet solves this using skip connections (shortcut connections)







Kaiming He, *et al.* "Deep residual learning for image recognition." CVPR 2016

Real-world Problems that can be Solved via CNN

❑Image Classification

- ➢Recognize objects, animals, handwritten digits (MNIST), or fashion items (Fashion-MNIST).Example: Classifying handwritten digits using CNN.

❑Facial Recognition

- ➢Identify or verify individuals based on facial features.

❑Visual Sentiment Analysis:

- ➢Analyze facial expressions to detect emotions (happy, sad, angry, surprised).

❑Time-Series Data as Images

- ➢Convert time-series data into images (like candlestick charts) and feed to CNNs.
- ➢Predict trend.

❑Medical Imaging

- ➢Detect tumors, pneumonia, or fractures from X-rays, CT scans, or MRI images.

Use CNN via PyTorch

- A Python-based deep learning library developed by Meta.
- Uses a dynamic computation graph, making it flexible.
- Supports automatic differentiation through autograd, allowing gradients to be computed automatically during backpropagation.

LeNet-5 Tutorial: Define the Layers

```
class LeNet5(nn.Module):
    def __init__(self, num_classes=2):
        super(LeNet5, self).__init__()
        # input: 1 channel; output: 6 channel
       # output: 6 x 28 x 28
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1)
        # output: 6 x 14 x 14
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2)
        # output: 16 x 10 x 10
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        # output: 16 x 5 x 5
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2)
        # output: 120 x 1 x 1
        self.conv3 = nn.Conv2d(16, 120, kernel_size=5)
        # Fully connected layers
        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, num_classes)
```


https://docs.pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

LeNet-5 Tutorial: Define the Forward Computation Process

```
class LeNet5(nn.Module):
    def __init__(self, num_classes=10):
        ...

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = self.pool1(x)
        x = torch.relu(self.conv2(x))
        x = self.pool2(x)
        x = torch.relu(self.conv3(x)) # 120 x 1 x 1
        x = x.view(x.size(0), -1)  # flatten
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

LeNet-5 Tutorial: Initialize the Model and Setting

```
# If CUDA GPU is available → use "cuda"; otherwise fall back to CPU.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Number of output classes from the dataset (e.g., 10 for MNIST).
num_classes = len(train_dataset.classes)
# Create the LeNet-5 model and move it to the chosen device.
model = LeNet5(num_classes=num_classes).to(device)
# Define the loss function for multi-class classification.
# CrossEntropyLoss = softmax + negative log-likelihood
criterion = nn.CrossEntropyLoss()
# Define the optimizer: Adam with learning rate 0.001.
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

LeNet-5 Tutorial: Data Preparation

```
# Assume folder structure (the data can be downloaded online):
# mnist01/
# ├── train/
# │   ├── 0/
# │   └── 1/
# └── test/
#     ├── 0/
#     └── 1/
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.Grayscale(), # Convert to 1 channel for LeNet-5
    transforms.ToTensor(),  # Convert image to PyTorch tensor (range [0, 1])
    transforms.Normalize((0.5,), (0.5,)) # Normalize to mean=0.5, std=0.5
])

train_data = datasets.ImageFolder(root='mnist01/train', transform=transform)
test_data = datasets.ImageFolder(root= 'mnist01/test', transform=transform)

train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_data, batch_size=64, shuffle=False)
```

LeNet-5 Tutorial: Training

```
num_epochs = 5
for epoch in range(num_epochs):
    # Some layers behave differently during training and evaluation.
    # Set model to training mode (enables dropout)
    model.train()
    running_loss = 0.0 # Accumulate loss for display or logging
    for images, labels in train_loader:
        # Move the mini-batch to the selected device (GPU/CPU)
        images, labels = images.to(device), labels.to(device)
        # Clear old gradients stored in the optimizer
        optimizer.zero_grad()
        # Forward pass: compute model predictions
        outputs = model(images)
        loss = criterion(outputs, labels)
        # Backward pass: compute gradients
        loss.backward()
        # Update model parameters using optimizer
        optimizer.step()

        running_loss += loss.item()
```

Dropout

- **During training, dropout randomly removes some neurons from the network.**
  - With probability p, the neuron is dropped (set to 0).
- **This forces the model to learn more robust features that do not rely on specific neurons.**
- **Dropout is a regularization technique used to reduce overfitting in neural networks.**

LeNet-5 Tutorial: Testing

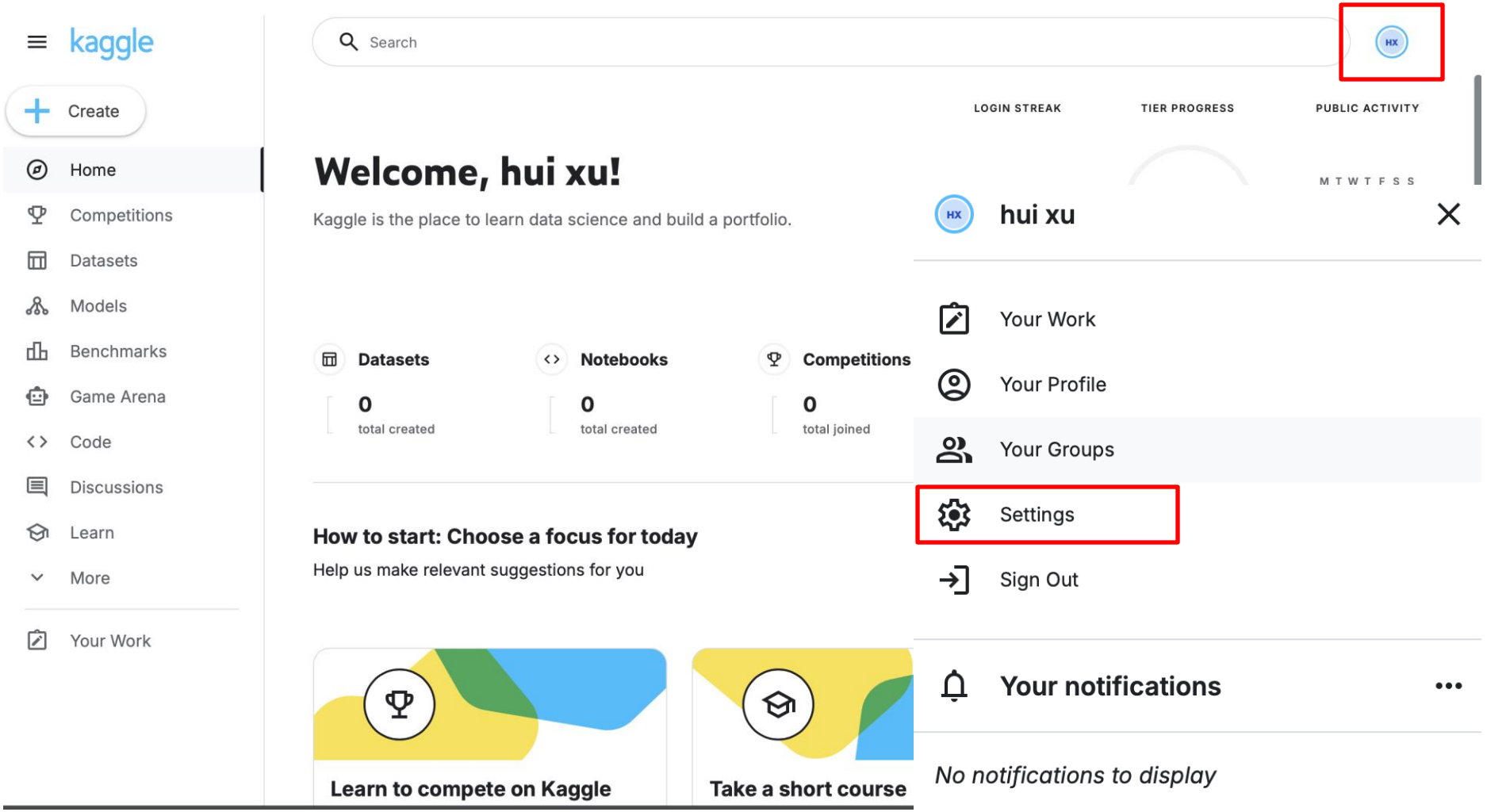
```
# Set model to evaluation mode
model.eval()
correct = 0
total = 0
# context manager: temporarily disables gradient tracking
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        # Forward pass: compute outputs/logits
        outputs = model(images)
        # Get predicted class (index of max logit)
        _, predicted = torch.max(outputs.data, 1)
        # Update total number of samples: the 0th dimension.
        total += labels.size(0)
        # Update number of correct predictions
        correct += (predicted == labels).sum().item()

print(f'Test Accuracy: {100 * correct / total:.2f}%')
```

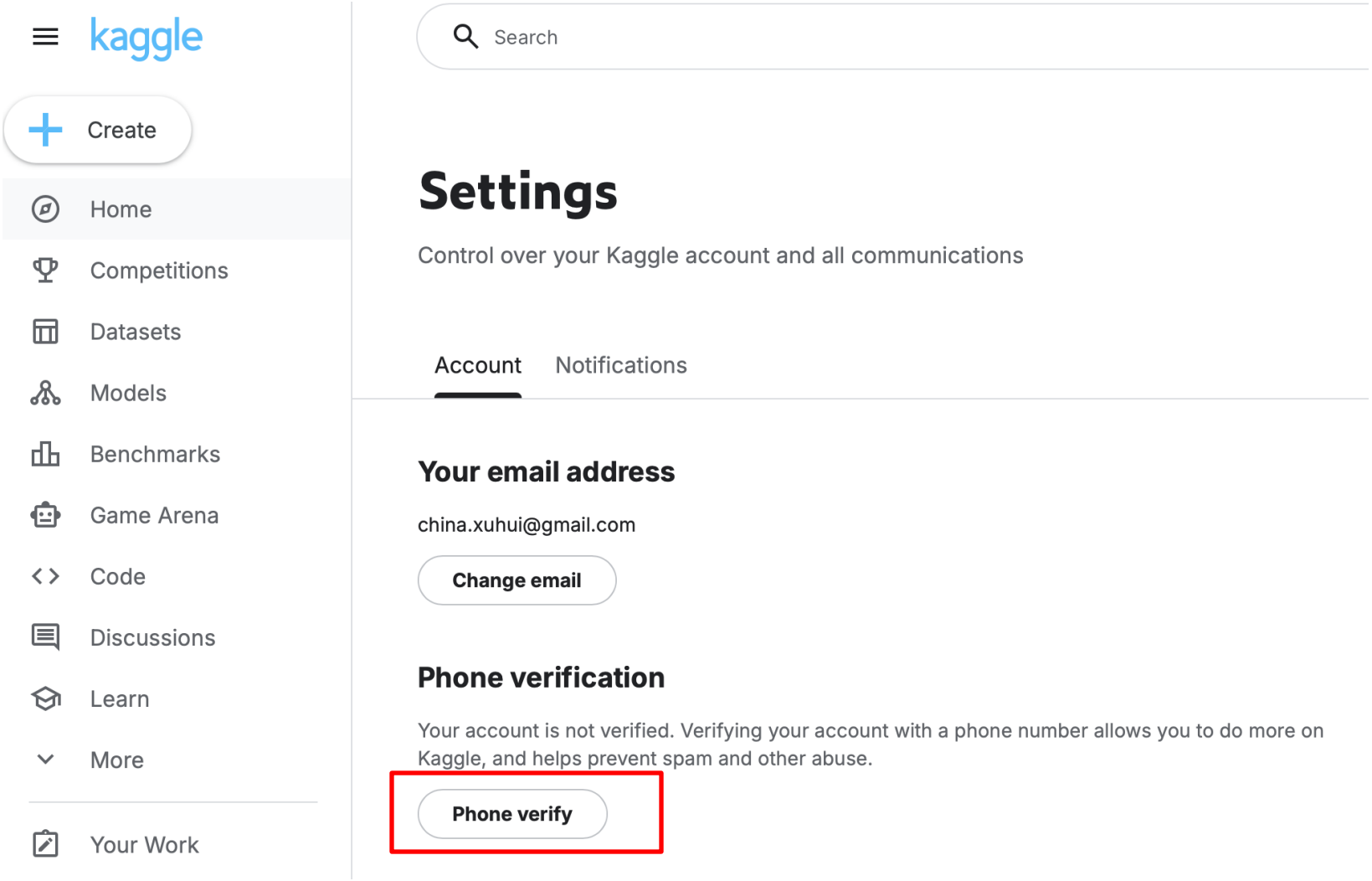
Deploy LeNet-5 with Kaggle Platform

- **An online platform for data science and machine learning.**
  - https://www.kaggle.com
- **Cloud-based coding environment (No local installation required).**
- **Supports Python & R for data analysis and ML.**
- **Free GPU/TPU access (30 hours/week).**
- **Resource Sharing:**
  - Notebooks: Publish notebooks publicly or keep them private.
  - Dataset and models: Seamless access to Kaggle datasets and models.

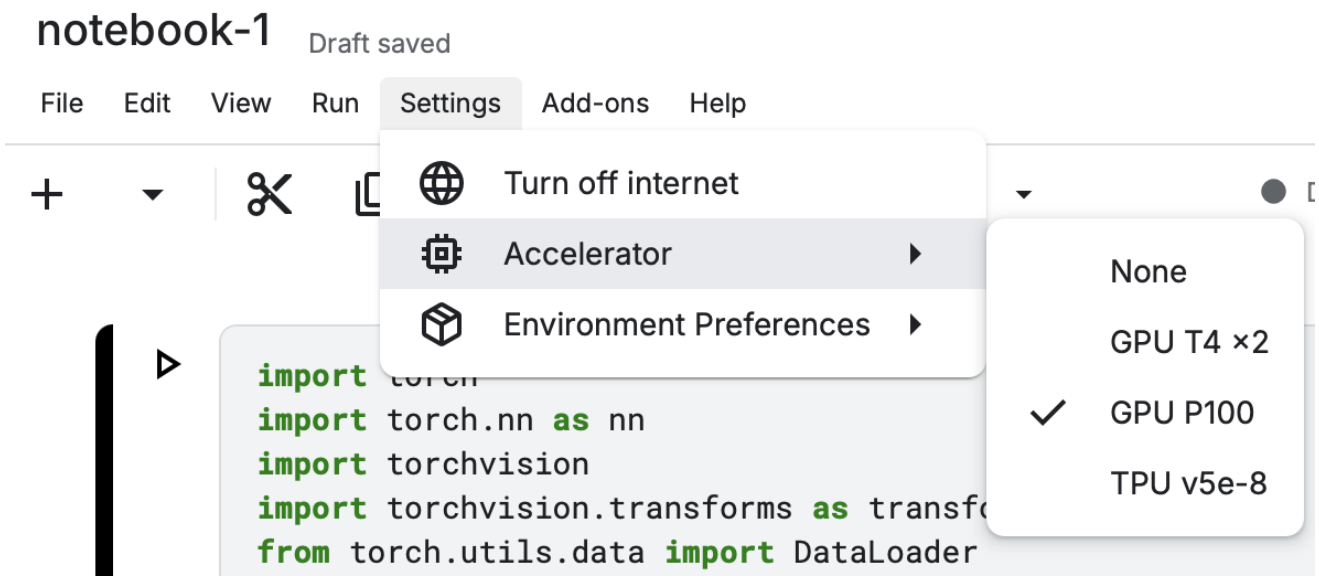
Kaggle Tutorial: Account Registration and Activation



Kaggle Tutorial: Account Registration and Activation



Crate Notebooks and Choose the Accelerator



Accelerator

Turning on GPU P100 will reduce the number of CPUs available and will only speed up image processing and neural networks.

Availability is limited to 30 hours per week. You have 30 hours remaining.

Cancel | Turn on GPU P100

3. Generative CNN

Generative Models

- **A generative model is a type of model that:**
  - Learns the data distribution of the training dataset.
  - Can generate new samples that resemble the training data.
- **For example: given images of cats, create new cat images that look realistic but are not exact copies of the training images.**
- **Two types generative image models:**
  - Generative Adversarial Networks (GANs).
  - Stable diffusion.

GAN: Generative Adversarial Networks

❑Use a two-player minimax game:

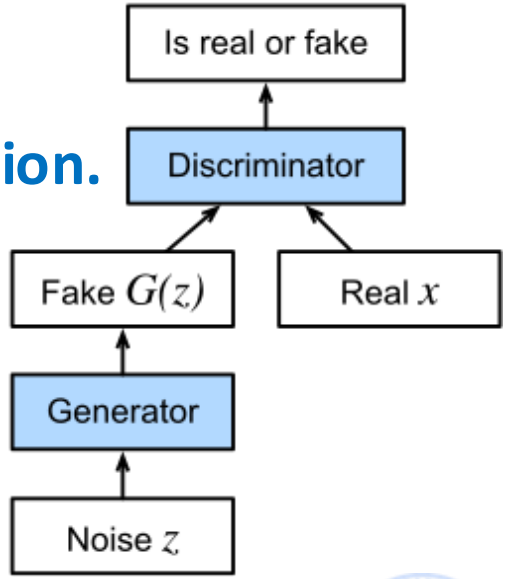
- ➢Generator G(z) produces fake samples.
- ➢Discriminator D(x) tries to distinguish real from fake.

❑D(x) maximize the probability of correct classification.

$$\max_D E[logD(x)] + E[1 - D(G(z))]$$

❑G(z) minimize the probability of correct classification.

$$\min_G E[1 - D(G(z))]$$



I. Goodfellow, *et al.* "Generative adversarial nets." NIPS 2014

Issues of GANs

❑Non-convergence / Training Instability

- ➢GAN training can fail to converge, oscillate, or diverge.

❑Mode Collapse: The generator produces limited variety of outputs.

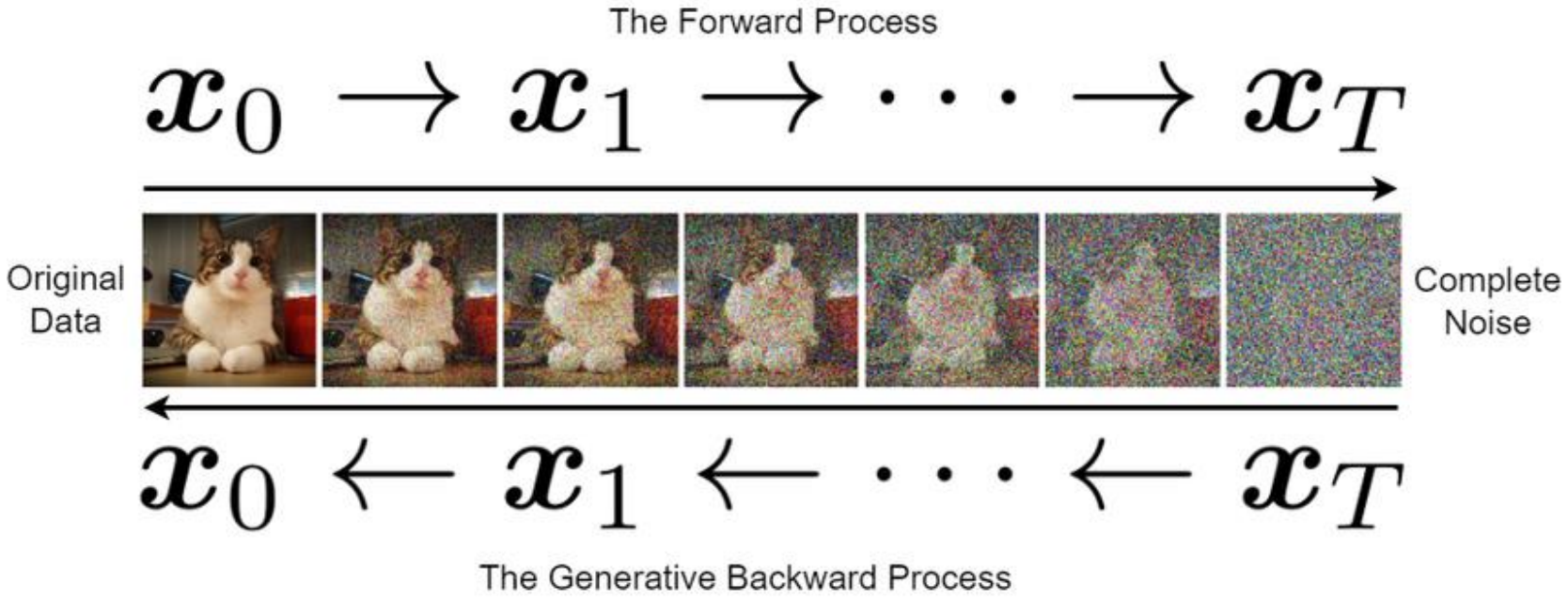
- ➢Generator finds “easy-to-fool” outputs that consistently trick the discriminator.

❑Evaluation Difficulty:

- ➢Unlike classification, GANs lack a single metric to measure success.

Stable Diffusion

- ❑Step 1: Add noise to images
- ❑Step 2: Train a neural network to remove noise
- ❑Step 3: Reverse process to generate images



R. Rombach, *et al.* "High-resolution image synthesis with latent diffusion models." CVPR 2022

Diffusion: Add noise to images

- Take a clean image x_0.
- Add random Gaussian noise gradually.

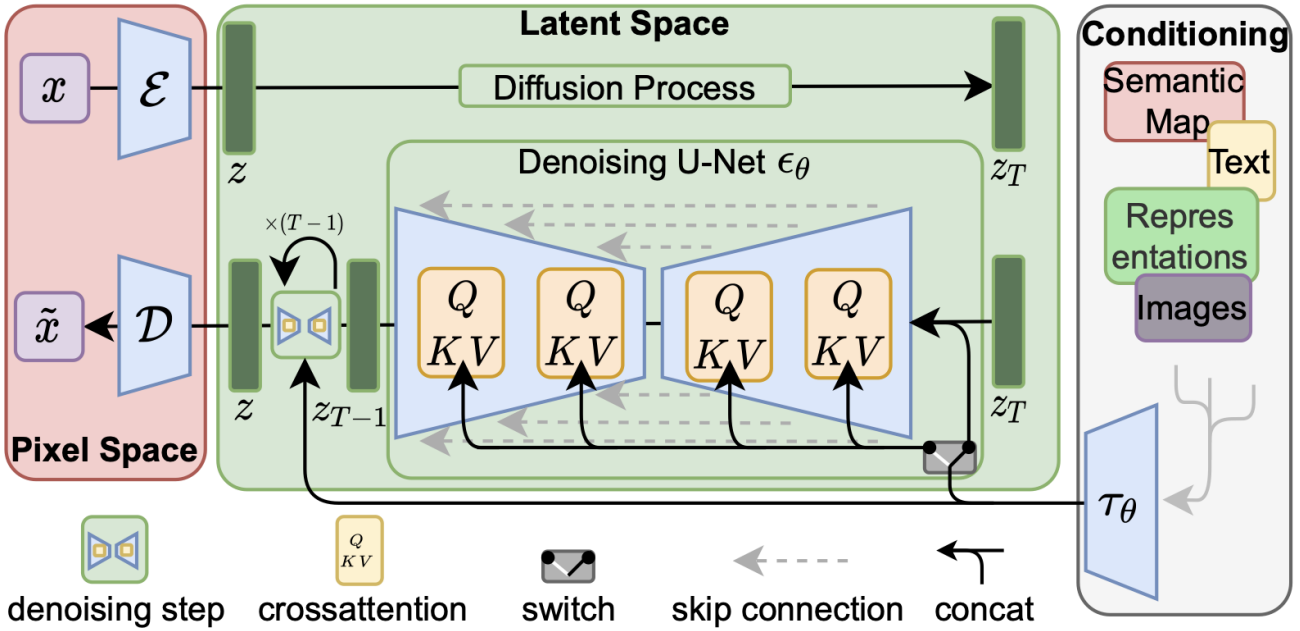
$$x_t = \sqrt{1-\beta_t}\,x_{t-1} + \sqrt{\beta_t}\,\epsilon_t$$

β_t = noise schedule (small positive number) for each step
$\epsilon_t \sim N(0, I)$ = random Gaussian noise for each pixel

- Over 1000 steps (until it becomes pure noise).

Denoise

- **Neural network predicts the noise component ϵ_t.**
- **Reconstruct the less noisy image.** $x_{t-1} = \frac{x_t - \sqrt{\beta_t}\epsilon_t}{\sqrt{1-\beta_t}}$
- **Text-to-Image Generation:**
  - LLM encodes the meaning of a text prompt into a semantic vector.
  - The diffusion model uses this vector to guide denoising at each step.



Deploy Stable Diffusion Application via Kaggle

❑zh-stable-diffusion-webui-kaggle is a Kaggle notebook that provides:

- ➢A ready-to-run Stable Diffusion environment
- ➢Pre-installed dependencies (PyTorch, CUDA, *etc*.)
- ➢Web UI for interactive text-to-image generation