Machine Learning Basics

A machine learning algorithm is an algorithm that is able to learn from data.

Mitchell (1997) provides a succinct definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E".

Task T

Machine learning tasks are usually described in terms of how the machine learning system should process an example. An example is a collection of features that have been quantitatively measured from some object or event that we want the machine learning system to process. We typically represent an example as a vector $\mathbf{x} \in \mathbb{R}^n$ where each entry x_i of the vector is another feature.

Common machine learning tasks include the following:

- Classification: In this type of task, the computer program is asked to specify which of k categories some input belongs to.
- Classification with missing inputs: Classification becomes more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided. To solve the classification task, the learning algorithm only has to define a single function mapping from a vector input to a categorical output. When some of the inputs may be missing,rather than providing a single classification function, the learning algorithm must learn a set of functions.
- Regression: In this type of task, the computer program is asked to predict a numerical value given some input.
- Transcription: In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe the information into discrete textual form.
- Machine translation: In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language.
- Synthesis and sampling: In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data.

The Performance Measure P

This quantitative measure evaluates the performance of the machine learning algorithm.

i.e. For tasks such as classification and transcription, we often measure the accuracy of the model. Accuracy is just the proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the error rate, the proportion of examples for which the model produces an incorrect output.

We therefore evaluate these performance measures using a test set of data that is separate from the data used for training the machine learning system.

The Experience E

A dataset is a collection of many examples, and we call examples data points.

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset.

Supervised learning algorithms experience a dataset containing features,but each example is also associated with a label or target.

Linear Regression

Let us consider the problem of Linear Regression. Linear regression solves a regression problem. In other words, the goal is to build a system that can take a vector $\mathbf{x} \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. The output of linear regression is a linear function of the input.

Let $\hat{y}$ be the value that our model predicts that y should take on. We define the output to be

$$\hat{y} = w^T x$$

where $w \in \mathbb{R}$ is a vector of parameters. Parameters are values that control the behavior of the system. Each parameter w_i affects the feature x_i is either in a positive, negative, or does not affect all.

So far we have a definition of our task T: to predict y from x by out putting $\hat{y} = w^T x$.
Next, let us define a Performance Measure P for the task. How about we select the mean squared error(MSE) between the true value y and the predicted value $\hat{y}$ as out P? In this case, our goal is to select the set of w that minimizes the MSE.

Assuming that we have m number of data points, we can write the MSE as follows:

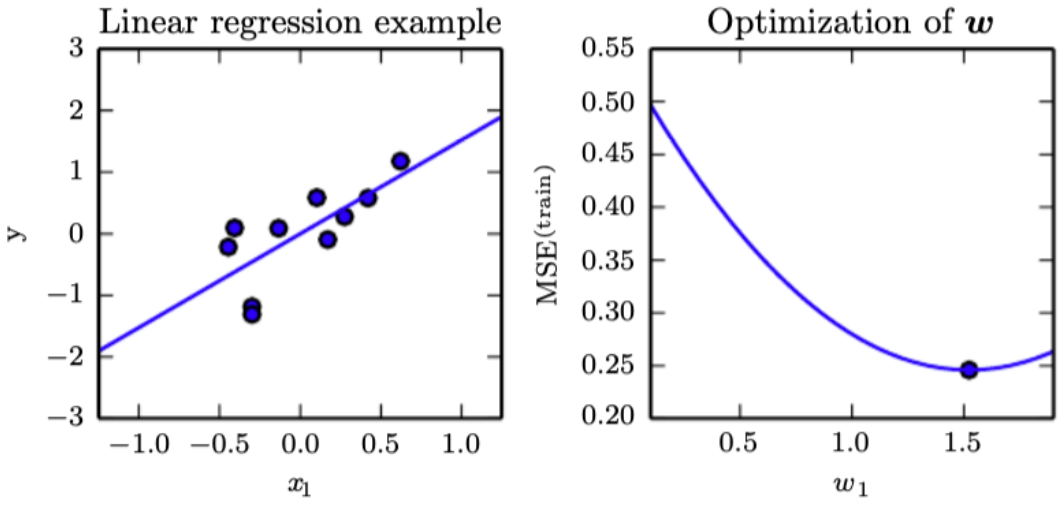
$$MSE = \frac{1}{m}\Sigma_i(\hat{y} - y)_i^2. \tag{1}$$

To make a machine learning algorithm, we need to design an algorithm that will improve weights w in a way that reduces MSE when the algorithm is allowed to gain experience by observing a training set (X, y). We can achieve this by minimize the mean squared error on the training set. To minimize MSE on the train, we can simply solve for where its gradient is 0. By doing some math, we arrive at the optimal set of weights for w as,

$$w = (XX^T)^{-1}X^T y \tag{2}$$

2 is known as the normal equations.

The image below visualizes our linear regression problem:



The following is a numerical example.

```
import numpy as np
import matplotlib.pyplot as plt

# Rainfall vs. Crop Yield
```

```python
x = np.array([100, 150, 200, 250, 300, 350, 400, 450, 500, 550])  # Rainfall
    (mm)
y = np.array([2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5])  # Crop
    Yield (tons/ha)

# X as single feature
X = x.reshape(-1, 1)
y = y.reshape(-1, 1)

# w = (X^T X)^{-1} X^T y
XT = X.T
XTX = XT @ X
XTX_inv = np.linalg.inv(XTX)
XTy = XT @ y
w = XTX_inv @ XTy

w_optimal = w[0][0]  # Slope only

# Calculate the regression line
x_range = np.linspace(x.min(), x.max(), 100)
y_hat = w_optimal * x_range

# Save plot
plt.scatter(x, y, color='blue', label='Data')
plt.plot(x_range, y_hat, color='red', label=f'y = {w_optimal:.2f}x')
plt.xlabel('Rainfall (mm)')
plt.ylabel('Crop Yield (tons/ha)')
plt.title('Linear Regression: Rainfall vs. Crop Yield')
plt.legend()
plt.grid(True)
plt.savefig('rainfall_yield_regression.png')
plt.close()

print(f"Slope (w): {w_optimal:.2f}")
```

Capacity, Overfitting and Underfitting

The ability to perform well on previously unobserved inputs is called **generalization**. So far, we focused on reducing the training error. The training error can be written as

$$\frac{1}{m^{train}}||X^{train}w - y^{train}||^2. \tag{3}$$

Minimizing the training error, we could fit the best regression line that separates the data in the training set. However, this does not guarantee that our model will correctly classify the unseen data. Let us call these data **test data.** Therefore, we should also focus on reducing the test error in our models. The test error can be written as

$$\frac{1}{m^{test}}||X^{test}w - y^{test}||^2. \tag{4}$$

The problem becomes harder as we do not observe the test error until we deploy our model in a useful application. Therefore, we can perform a set of tasks prior to training the model such that we can reasonably assume that the test error will be low. For instance, we can select the training data to be identically distributed to the test data, and the two datasets are independent from each other.

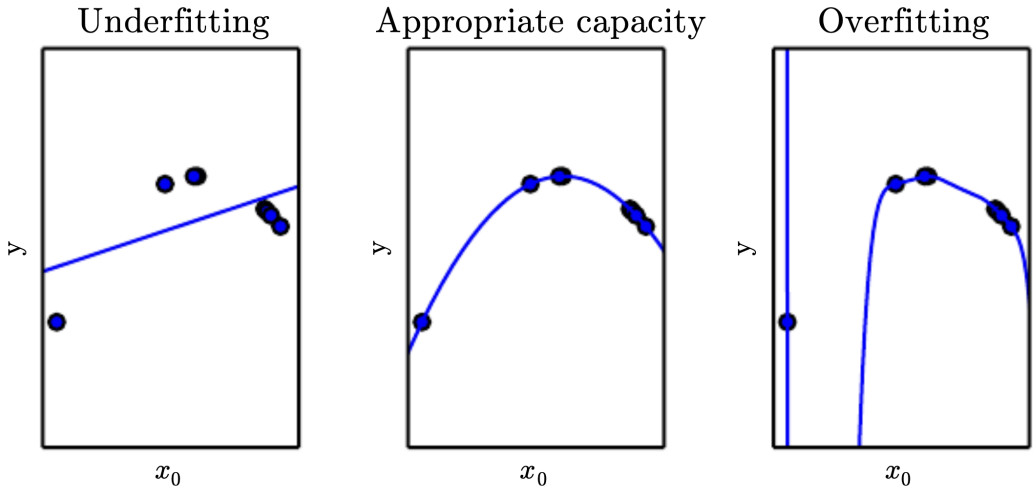
The problems of minimizing the training error and making the gap between the training error and the test error give rise to two major problems, underfitting and overfitting. Underfitting occurs

when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large.

We choose a model for our data. By selecting a model with correct **capacity**, we can control weather our model overfit or underfit to the data.

Model capacity in machine learning refers to a model's ability to learn and represent complex patterns or relationships in data, determined by its size and structure (e.g., number of layers, neurons, or parameters in a neural network).

For instance, consider the three polynomials with varying degrees: $\hat{y} = b + wx$, $\hat{y} = b + w_1 x + w_2 x^2$, and $\hat{y} = b + \Sigma_{i=1}^{9} w_i x^i$. First may be inadequate to capture the variations in the data, hence the data will be underfitted. The third has the capacity to be fir for a variety of data, hence it may overfit the data. The second would have the appropriate capacity to represent the data, hence will be the best model to reduce the two types of error we learned.



The following is an neumerical example:

```
import numpy as np
import matplotlib.pyplot as plt

# Non-linear dataset: y = x^2 + sin(x) + noise
np.random.seed(0)
x = np.linspace(-5, 5, 10)
y = x**2 + np.sin(x) + np.random.normal(0, 1, len(x))

# Polynomial regression function
def poly_regression(X, y, degree):
    XT = X.T
    XTX = XT @ X
    XTX_inv = np.linalg.inv(XTX)
    XTy = XT @ y
    w = XTX_inv @ XTy
    return w

# Prepare polynomial features
X_1 = np.column_stack([np.ones(len(x)), x])  # Degree 1: [1, x]
X_2 = np.column_stack([np.ones(len(x)), x, x**2])  # Degree 2: [1, x, x^2]
X_9 = np.column_stack([x**i for i in range(10)])  # Degree 9: [1, x, ..., x
    ^9]

# Fit polynomials
w_1 = poly_regression(X_1, y, 1)
```

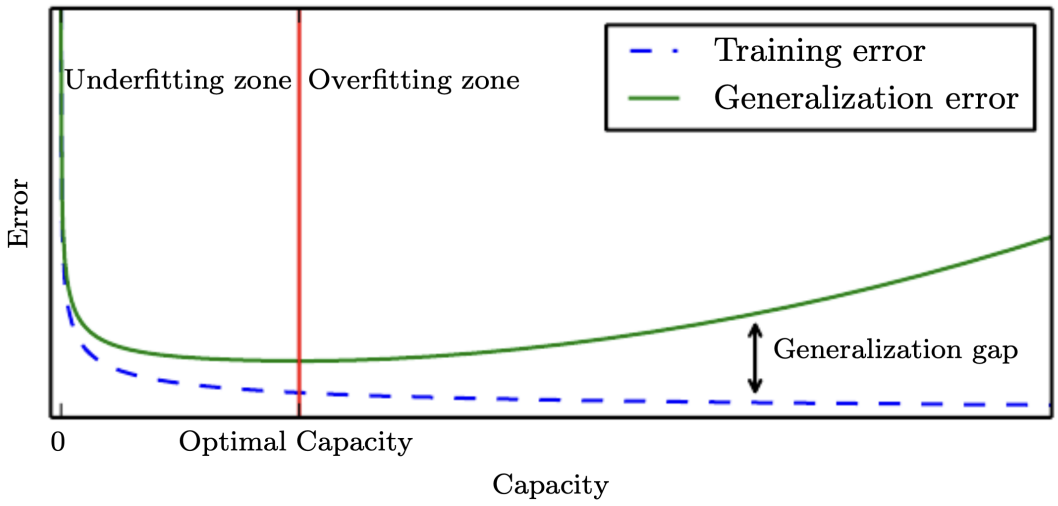
```
w_2 = poly_regression(X_2, y, 2)
w_9 = poly_regression(X_9, y, 9)

# Generate predictions
x_range = np.linspace(-5, 5, 100)
X_range_1 = np.column_stack([np.ones(len(x_range)), x_range])
X_range_2 = np.column_stack([np.ones(len(x_range)), x_range, x_range**2])
X_range_9 = np.column_stack([x_range**i for i in range(10)])
y_pred_1 = X_range_1 @ w_1
y_pred_2 = X_range_2 @ w_2
y_pred_9 = X_range_9 @ w_9

# Save plot
plt.scatter(x, y, color='blue', label='Data')
plt.plot(x_range, y_pred_1, color='red', label='Degree 1')
plt.plot(x_range, y_pred_2, color='green', label='Degree 2')
plt.plot(x_range, y_pred_9, color='orange', label='Degree 9')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Polynomial Fits: Degrees 1, 2, 9')
plt.legend()
plt.grid(True)
plt.savefig('polynomial_fits.png')
plt.close()
```

A high-capacity model, like a deep neural network with many parameters, can fit intricate datasets but risks overfitting if not regularized, while a low-capacity model (e.g., linear regression) is simpler but may underfit complex data.

Simpler functions are more likely to generalize(to have a small gap between training and test error. We must still choose a sufficiently complex hypothesis to achieve a low training error. Typically, generalization error has a U-shaped curve as a function of model capacity.



Regularization

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

Regularization acts as a penalty by adding an extra term to the loss function, which discourages the model from using overly large parameter values, effectively simplifying the model to avoid over-

fitting—where it memorizes training data noise instead of learning general patterns. The penalty term, such as the L2 norm ($\lambda \sum w_i^2$) in Ridge regularization or the L1 norm ($\lambda \sum |w_i|$) in Lasso, increases the loss if weights grow large, pushing the optimization process to favor smaller weights that reduce complexity. This constraint helps the model generalize better to unseen data by preventing it from fitting every minor fluctuation in the training set. (For dropout, the penalty is implicit—randomly disabling neurons during training forces the network to avoid relying too heavily on any single weight, mimicking a regularization effect). The balance is controlled by λ, where a higher value imposes a stricter penalty, trading off fit for simplicity.

Consider a linear regression problem with one feature x and target y, where the original loss is the mean squared error (MSE):

$$\text{Loss} = \frac{1}{n} \sum (y_i - wx_i)^2.$$

With L2 regularization, the modified loss becomes

$$\text{Loss}_{\text{regularized}} = \frac{1}{n} \sum (y_i - wx_i)^2 + \lambda w^2$$

, where w is the weight and λ is the regularization parameter.

- **Data**: $x = [1, 2, 3]$, $y = [2, 4, 5]$ (noisy fit for $y \approx 1.5x$).
- **Without Regularization**: Minimize

  $$\text{MSE} = \frac{(2 - 1w)^2 + (4 - 2w)^2 + (5 - 3w)^2}{3}.$$

  Solving $\frac{d}{dw}\text{MSE} = 0$ gives $w \approx 1.833$, MSE ≈ 0.222.
- **With L2 ($\lambda = 0.1$)**: Minimize

  $$\text{Loss} = \frac{(2 - 1w)^2 + (4 - 2w)^2 + (5 - 3w)^2}{3} + 0.1w^2.$$

  The derivative

  $$\frac{d}{dw}\text{Loss} = \frac{-2(2 - 1w) \cdot 1 - 2(4 - 2w) \cdot 2 - 2(5 - 3w) \cdot 3}{3} + 0.2w = 0$$

  yields $w \approx 1.667$, with a slightly higher MSE (≈ 0.278) but a simpler model due to the penalty on w^2.

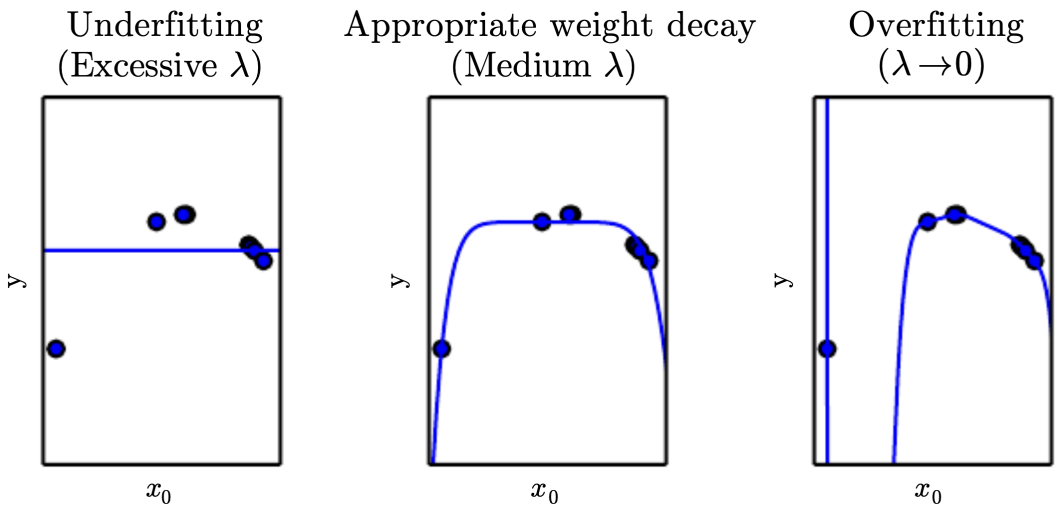
The following is another numerical example.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv

# Non-linear dataset: y = x^2 + sin(x) + noise
np.random.seed(0)
x = np.linspace(-5, 5, 20)
y = x**2 + np.sin(x) + np.random.normal(0, 1, len(x))
# y[8] = 20

# Polynomial features (degree 9 for overfit potential)
X = np.column_stack([x**i for i in range(10)])  # [1, x, x^2, ..., x^9]

# Function for Ridge regression
def ridge_regression(X, y, lambda_reg):
    XT = X.T
    XTX = XT @ X + lambda_reg * np.eye(X.shape[1])
```




```
    XTX_inv = inv(XTX)
    XTy = XT @ y
    w = XTX_inv @ XTy
    return w

# Vary lambda and compute
lambdas = [0.0, 10, 100]
for lam in lambdas:
    w = ridge_regression(X, y, lam)
    y_pred = X @ w
    mse = np.mean((y - y_pred)**2)
    print(f"Lambda: {lam}, MSE: {mse:.2f}, Coefficients: {w.round(2)}")

# Plot for all lambda values
plt.scatter(x, y, color='blue', label='Data')
x_range = np.linspace(-5, 5, 100)
X_range = np.column_stack([x_range**i for i in range(10)])
colors = ['red', 'green', 'orange', 'purple']
for lam, color in zip(lambdas, colors):
    w = ridge_regression(X, y, lam)
    y_pred = X_range @ w
    plt.plot(x_range, y_pred, color=color, label=f'Lambda={lam}')

plt.xlabel('x')
plt.ylabel('y')
plt.title('L^2 Effects on Non-Linear Data (Degree 9)')
plt.legend()
plt.grid(True)
plt.savefig('L2_nonlinear_degree9.png')
plt.close()
```

Hyperparameters and Validation Sets

Hyperparameters are the settings that we can use to control the algorithm's behavior. The degree of the polynomial, which acts as a capacity hyperparameter and the λ value used to control the strength of weight decay are examples of hyperparameters.

If we learned these hyperparanmeters on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting. Which means we need an

alternative data set. To solve this problem, we need **validation set** of examples that the training algorithm does not observe.

It is important that the test examples are not used in any way to make choices about the model, including its hyperparameters. For this reason, no example from the test set can be used in the validation set. Therefore, we always construct the validation set from the training data.

We typically do a 80-20 split on the training data set to produce a validation set.

If the validation set is small, we would randomly choose subsets of the original data and split into training and validation sets. This is **cross-validation**. k-folding is a common procedure.

Artificial Neural Networks

Artificial Neural Networks(ANN) are computational models developed for complex problem solving. These are inspired by biological neural networks, where interconnected neurons organized in layers process provide input to compute an output.

Perceptron

The basic unit of an ANN is a perceptron. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Below image is for a simple perceptron with 4 inputs and one output.

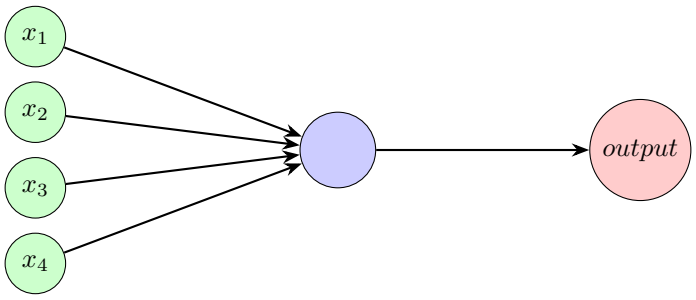


Figure 1: Structure of a perceptron with 4 inputs and 1 output.

However, the above diagram does not have a defined way of computing the output from the inputs. The output can be 0 or 1. Hence, the concepts of weights w were introduced. The weights w will define how much each of the inputs effects the output. This is determined by the weighted sum $\Sigma_j w_j x_j$. If the weighted sum is greater than or less than a predefined threshold, the output will change accordingly.

$$output = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq threshold \\ 0 & \text{if } \sum_{i=1}^{n} w_i x_i < threshold \end{cases}$$

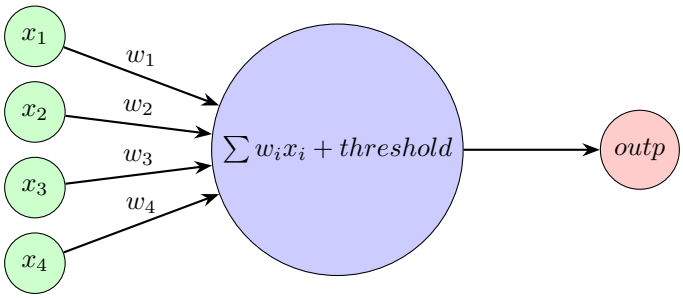


Figure 2: Structure of a perceptron with 4 inputs and 1 output.

Let us analyze more on this through a simple example of buying an ice cream. Let there be four factors that goes into buying an ice cream. x_1: Sunny day / Rainy day. x_2: the store is far / close. x_3: got change or not x_2: Toothache / no toothache. If you really want an ice cream on a sunny day, you assign weights [4, 2, 2, 1] and the threshold at 5, which gives more importance to the day being sunny or not to influence your decision to have ice cream. If you assign weights [3, 1, 1, 10], and select your threshold at 8, the only variable that matters is if you have a toothache or not.

Let's simplify the way we describe perceptrons. The condition $\sum_{i=1}^{n} w_i x_i > threshold$ is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\sum_{i=1}^{n} w_i x_i$ as a dot product, $w.x = \sum_{i=1}^{n} w_i x_i$, where w and x are vectors whose components are the weights

and inputs, respectively. The second change is to move the threshold to the other side of the inequality and to replace it by what is known as the perceptron's bias, $b = -threshold$. Using the bias instead of the threshold, the perceptron rule can be rewritten,

$$output = \begin{cases} 1 & \text{if } w.x + b \geq 0 \\ 0 & \text{if } w.x + b < 0 \end{cases}$$

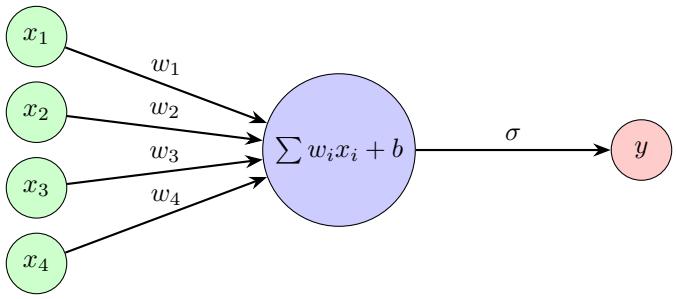


Figure 3: Structure of a perceptron with 4 inputs and 1 output.

However, this model is very sensitive to change in input. For example, a small change of x can cause the weighted sum to tip over the threshold. Therefore, we need a continuous value of the output to better represent the change in the input.

Sigmoid Activation Function

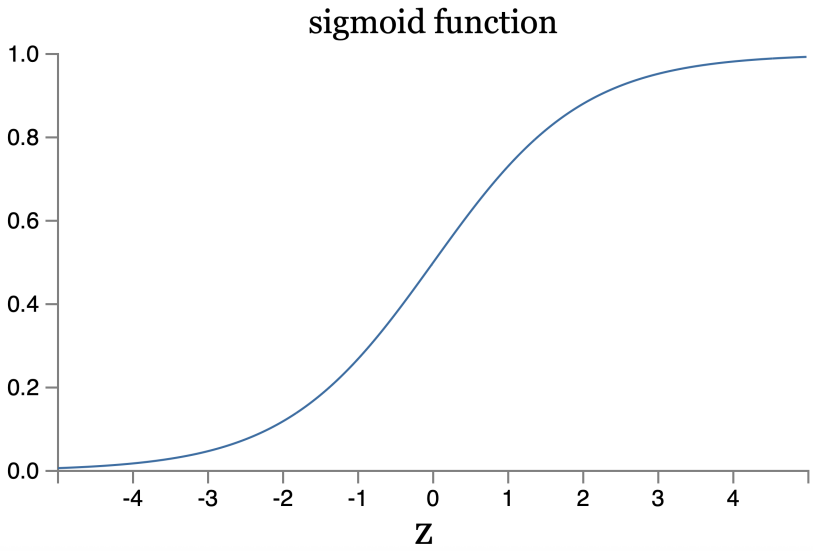
We pass $w.x + b$ through a sigmoid function,

$$\sigma(z) = \frac{1}{1 + e^(-z)}.$$

When applied to our inputs x, weights w, and bias b, this will be,

$$\sigma(w.x + b) = \frac{1}{1 + e^(-w.x - b)}.$$

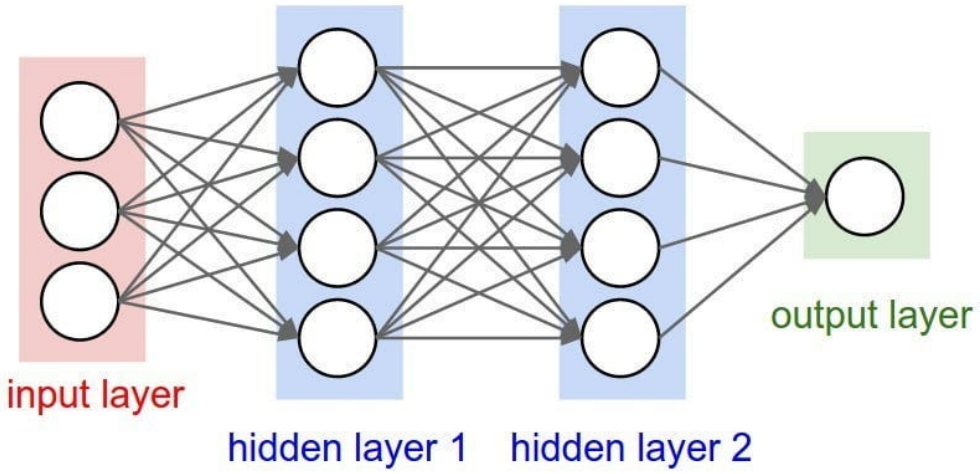
The following is the output of the sigmoid function.



This perceptron can model simple linear decision boundaries but struggles with non-linearly separable data (e.g., XOR problem).

Feedforward Neural Network

Given that we have a single perceptron with sigmoid function to map a simple function, a complex neural network will combine perceptrons to learn complex functions.



Input Layer: Each neuron represents one feature of the input data; $x_1, x_2, \cdots, x_n$. No computation occurs here; inputs are passed to the next layer.

Hidden Layer(s): Contains neurons that apply weights, biases, and activation functions (e.g., sigmoid). Each neuron in a hidden layer is like a perceptron, processing inputs from the previous layer. For a hidden layer with 3 neurons, each computes:

$$h_j = \sigma(\sum_i w_{ij} x_i + b_j), j = 1, 2, 3.$$

Output Layer: Produces the final prediction. For binary classification with sigmoid, a single neuron outputs,

$$y = \sigma(\sum_j w_j h_i + b).$$

Learning optimal weights

These networks use gradient descent and backpropagation to learn the optimal weights.

1 Loss Function measures error between predicted y_{pred} and true y_{true}. Common for binary classification:

$$J = -\frac{1}{N} \sum [y_{\text{true}} \log(y_{\text{pred}}) + (1 - y_{\text{true}}) \log(1 - y_{\text{pred}})]$$

2 Forward Pass computes predictions through all layers.

3 Backpropagation computes gradients of the loss J with respect to weights and biases using the chain rule. For a weight w_{ij}, update:

$$w_{ij} := w_{ij} - \eta \frac{\partial J}{\partial w_{ij}}$$

where η is the learning rate.

4. Gradient Descent will iteratively update all parameters to minimize J.

Convolutional Neural Network

CNNs are a specialized kind of neural network for processing data that has a known grid-like topology. Instances would be a 1D array of data from a sensor or 2D image data.

Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

Convolutional Operation

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the spaceship at time t. Both x and t are real valued, that is, we can get a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where a is the age of a measurement.

After applying the weighted average operation at every moment, assuming the time is discrete, we can estimate the new position of the spacecraft at time k,

$$s[k] = (x * w)[k] = \sum_{a=-\infty}^{\infty} x(a)w(k-a)$$

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input**, and the second argument (in this example, the function w) as the **kernel**. The output is sometimes referred to as the feature map.

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as **tensors**.

sparse interactions / sparse connectivity / sparse weights

Typically, the kernel is smaller than the input. For example,when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It makes calculations efficient, $O(m \times n)$ to $O(k \times n)$.

Parameter sharing

In traditional NN, each element of the weight matrix is only used once when computing the output. However, with the kernel, nearly all input is being multiplied by the same kernel.

In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set.

Learning a Kernel

There will be multiple kernels in a typical convolutional layer. These kernels are defined randomly in the beginning of the training and are learned via back propagation throughout the training process. These are also known as $filters$.

Pooling

A typical layer of a convolutional network consists of three stages. Convolutional layer, the activation layer, and the pooling layer. In the pooling layer, we use a pooling function to modify the output of the layer further.

A pooling function replaces the net output at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network. Take a look at the following example from the deep learning book.

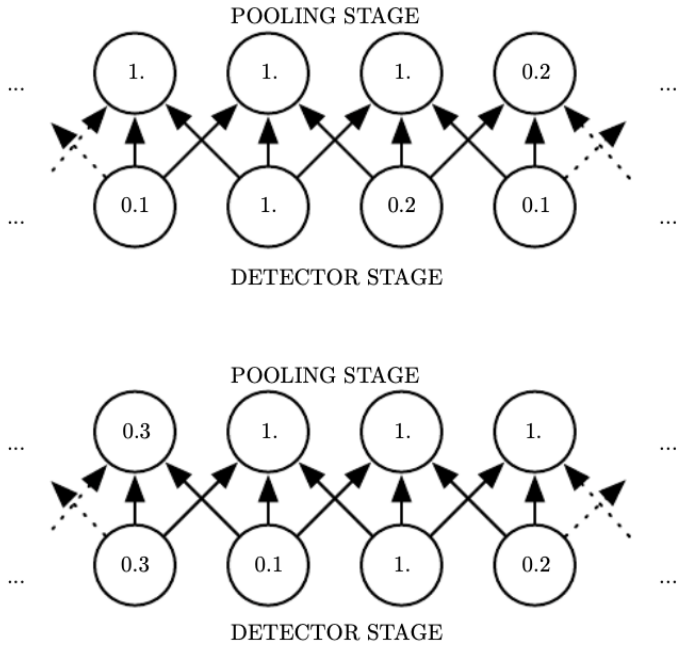


Figure 9.8: Max pooling introduces invariance. *(Top)*A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. *(Bottom)*A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are sensitive only to the maximum value in the neighborhood, not its exact location.

The following code implements a simple CNN using the library *Tensorflow*.

```
import tensorflow as tf
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
```

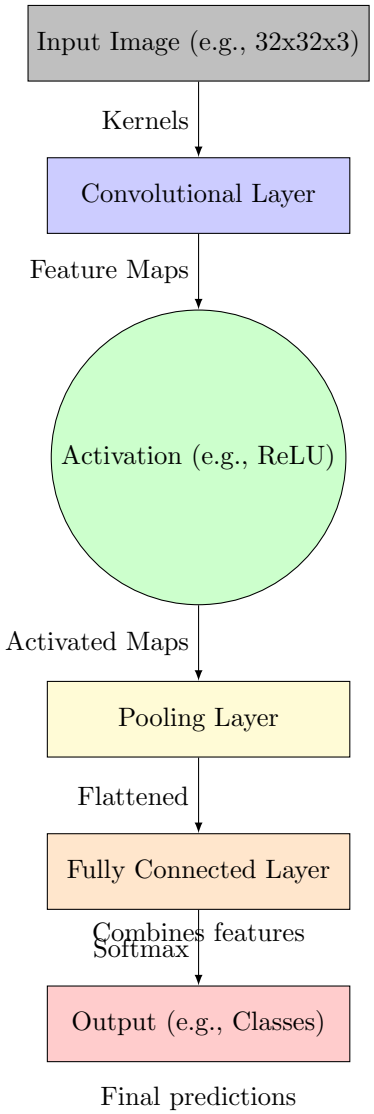


Figure 4: A CNN representation

```
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
model.summary()
```

Figure 4 shows all the components of a CNN.

Evolution of CNNs

LeNet

LeNet was introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images. LeNet consists of two parts: (i) a convolutional encoder consisting of two convolutional layers; and (ii) a dense block consisting of

three fully connected layers.

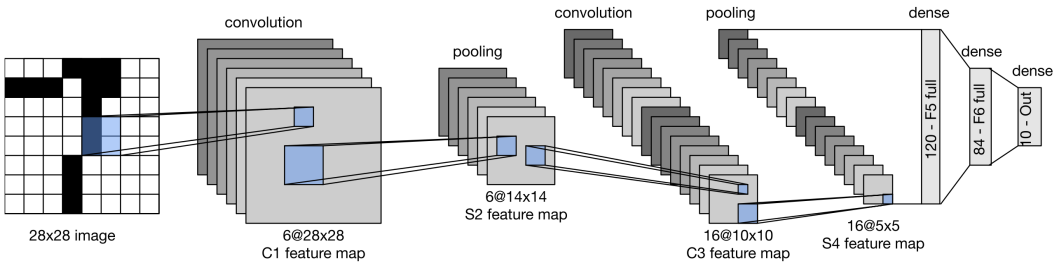


Figure 5: Architecture of LeNet

AlexNet

When CNNs started to develop, the lowest layers of the network, the model learned feature extractors that resembled some traditional filters. Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, and so on. Even higher layers might represent whole objects like people, airplanes, dogs, or frisbees. Ultimately, the final hidden state learns a compact representation of the image that summarizes its contents such that data belonging to different categories can be easily separated.

The architectures of AlexNet and LeNet are strikingly similar. However, AlexNet consists of eight layers: five convolutional layers, two fully connected hidden layers, and one fully connected output layer. AlexNet used ReLU instead of the sigmoid as its activation function.

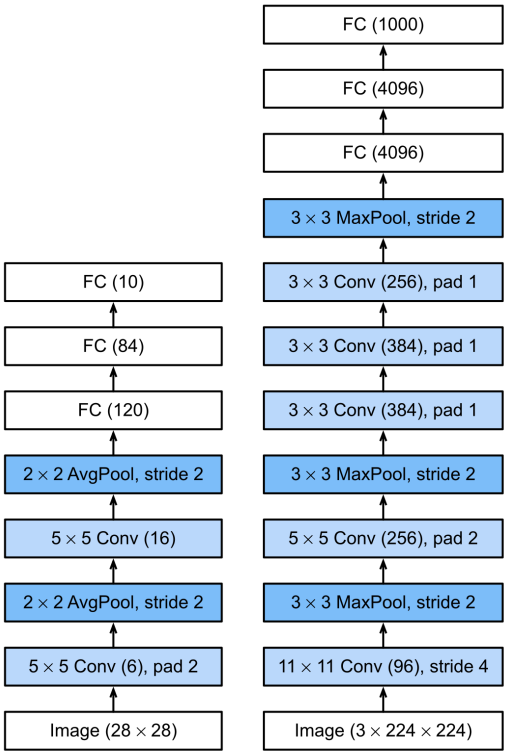
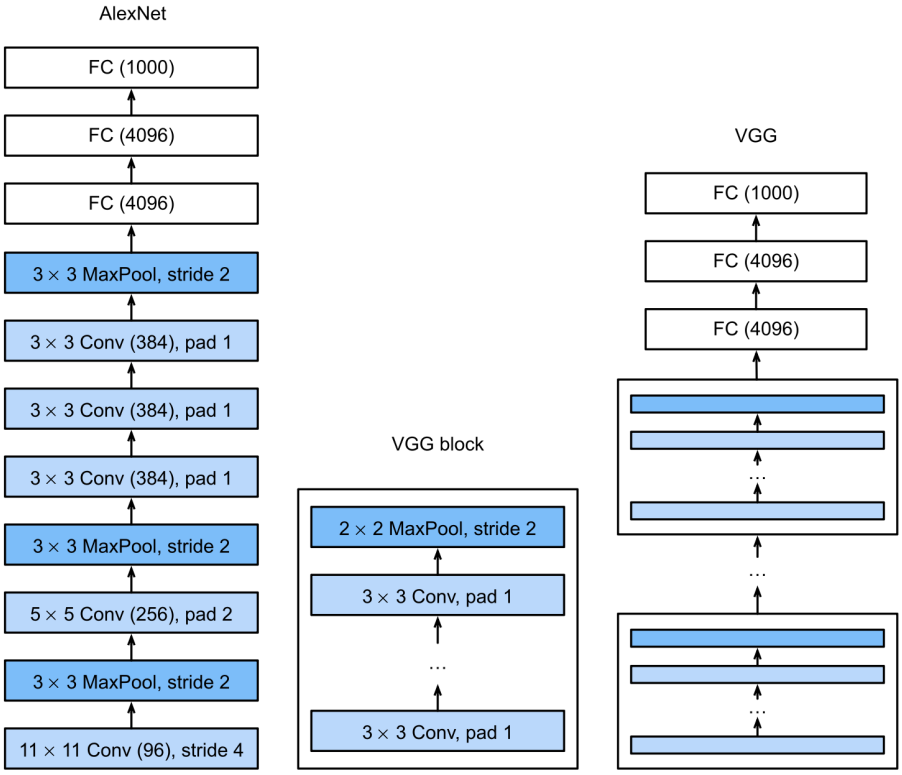


Figure 6: LeNet(left) AlexNet(right)

VGG Network

The basic building block of CNNs is a sequence of the following: (i) a convolutional layer with padding to maintain the resolution, (ii) a nonlinearity such as a ReLU, (iii) a pooling layer such as max-pooling to reduce the resolution. However, there's a problem. Pooling layers (e.g., 2x2 with stride 2) repeatedly halve spatial dimensions. For an input of size $d \times d$, each pooling layer reduces resolution by half, so after k layers, resolution is $d/2^k$.

As a solution, the idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University. A VGG block consists of a sequence of convolutions with 3 $\times$3 kernels with padding of 1 (keeping height and width) followed by a 2 $\times$2 max-pooling layer with stride of 2 (halving height and width after each block).



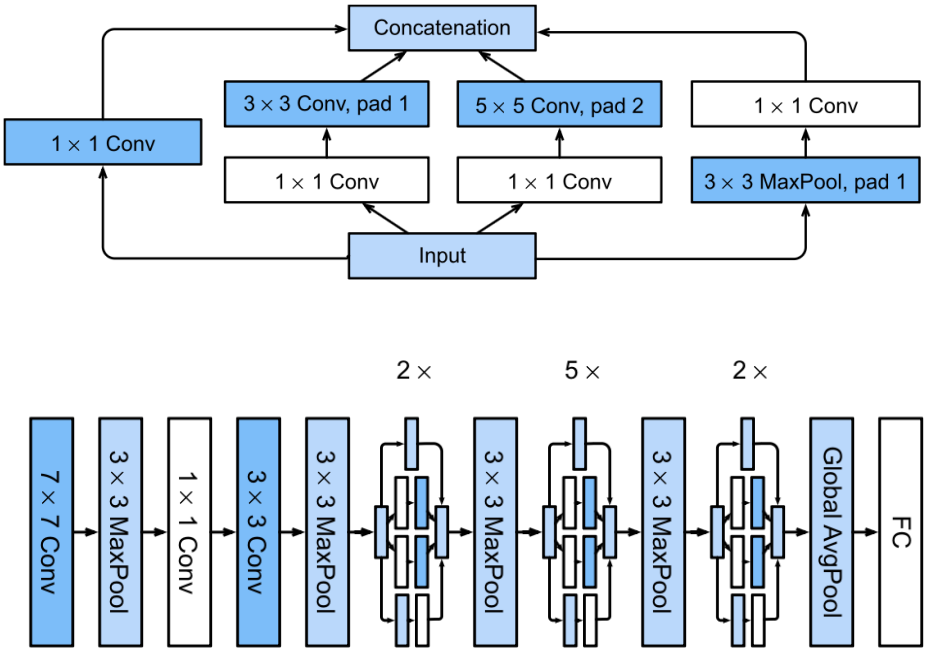
GoogLeNet Model

The basic convolutional block in GoogLeNet is called an Inception block. These blocks provided multi-scale feature extraction through parallel convolutions. Dimensionality was reduced by 1x1 convolutions to reduce the number of channels before larger convolutions, lowering computational cost while maintaining expressive power. They increased the depth and the width for deeper and wider networks without exponential growth in parameters.

GoogLeNet uses a stack of a total of 9 inception blocks, arranged into three groups with max-pooling in between, and global average pooling in its head to generate its estimates.

Object Detection though CNN

Image classification identifies if a particular object is present in the image or not. However, now we want to know not only object category in an image, but also their specific positions in the image. In computer vision, we refer to such tasks as object detection (or object recognition).



Bounding Boxes

A bounding box describes the spatial location of an object. The bounding box is rectangular, which commonly is determined by the x and y coordinates of the upper-left corner of the rectangle and the coordinates of the lower-right corner. The following are two other commonly used formats.

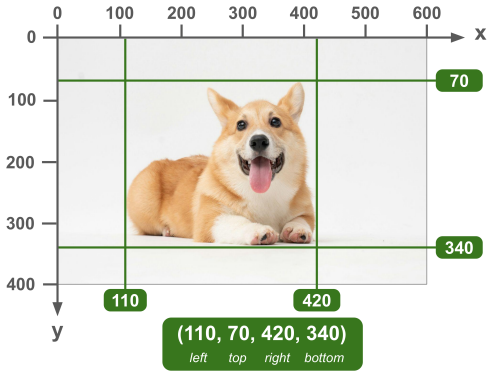


Figure 7: (x_1, y_1, x_2, y_2) format

Anchor Boxes

Object detection algorithms usually sample a large number of regions in the input image, determine whether these regions contain objects of interest, and adjust the boundaries of the regions so as to predict the ground-truth bounding boxes of the objects more accurately. Different models may adopt different region sampling schemes. Here we introduce one of such methods: it generates multiple bounding boxes with varying scales and aspect ratios centered on each pixel. These bounding boxes are called anchor boxes.

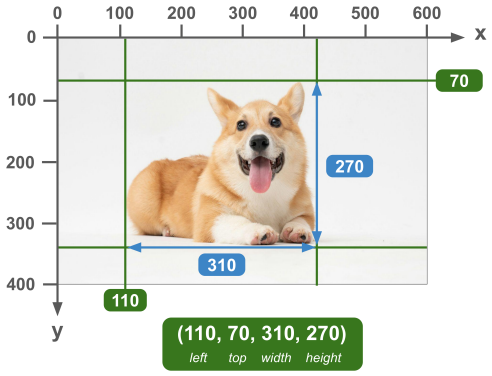


Figure 8: $(x_{left}, y_{top}, width, height)$ format

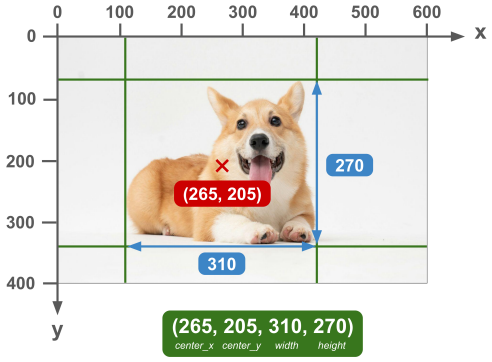


Figure 9: $(x_{center}, y_{center}, width, height)$ format

Intersection over Union IoU

IoU measures the similarity between the anchor box and the ground-truth bounding box. It is the ratio of the intersection area to the union area of two bounding boxes. The range of an IoU is between 0 and 1: 0 means that two bounding boxes do not overlap at all, while 1 indicates that the two bounding boxes are equal.

Non-Maximum Suppression

When there are many anchor boxes, many similar (with significant overlap) predicted bounding boxes can be potentially output for surrounding the same object. To simplify the output, we can merge similar predicted bounding boxes that belong to the same object by using non-maximum suppression (NMS).

Single Shot Multi-box Detection

Single Shot MultiBox Detector (SSD) is an object detection model that predicts object bounding boxes and class probabilities in a single forward pass of a CNN, making it fast and efficient. The following is how it works:

- Input image passes through a backbone CNN (e.g., VGG16).
- Multiple feature maps at different resolutions are generated.
- For each feature map, anchor boxes are placed at every grid cell.

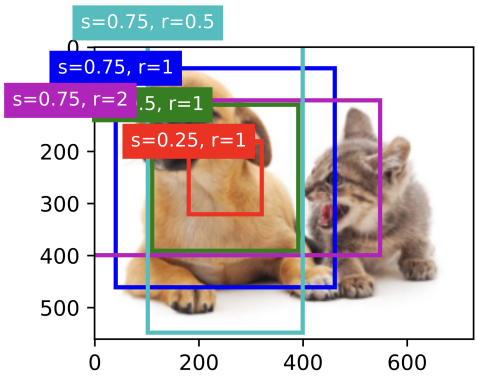


Figure 10: Anchor Boxes

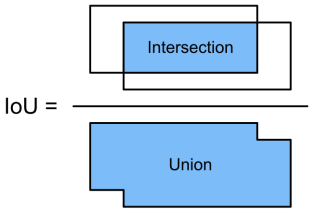


Figure 11: IoU

- For each anchor box, SSD predicts:
  - Bounding box offsets (to adjust anchor box coordinates).
  - Class probabilities (including background class).
- Non-maximum suppression (NMS) filters overlapping predictions to produce final detections.

Region-based CNNs (R-CNNs)

The **R-CNN** first extracts many region proposals from the input image, labeling their classes and bounding boxes. Then a CNN is used to perform forward propagation on each region proposal to extract its features. Next, features of each region proposal are used for predicting the class and bounding box of this region proposal.

However, each of the proposed regions are passed through a CNN, which may have a lot of overlap. One of the major improvements of the **fast R-CNN** from the R-CNN is that the CNN forward propagation is only performed on the entire image.

To be more accurate in object detection, the fast R-CNN model usually has to generate a lot of region proposals in selective search. To reduce region proposals without loss of accuracy, the **faster R-CNN** proposes to replace selective search with a region proposal network.

In the training dataset, if pixel-level positions of object are also labeled on images, the mask R-CNN can effectively leverage such detailed labels to further improve the accuracy of object detection.

The following implements a CNN to detect circles:

```
import tensorflow as tf
import numpy as np
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

# Generate dataset
def gen_data(n=1000, size=64):
```

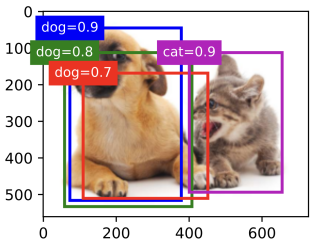


Figure 12: before NMS

```
    X, y = [], []
    for i in range(n):
        img = np.zeros((size, size))
        if np.random.rand() > 0.5:  # Circle present
            cx, cy, r = np.random.uniform(0.2*size, 0.8*size, 3)
            y_, x_ = np.ogrid[:size, :size]
            mask = (x_ - cx)**2 + (y_ - cy)**2 <= r**2
            img[mask] = 1
            y.append(1)
        else:
            y.append(0)
        X.append(img)

        # Save firsxt 5 data points as images
        if i < 5:
            rgb = tf.image.grayscale_to_rgb(tf.expand_dims(img.astype(np.
                float32), -1))
            tf.keras.utils.save_img(f"sample_{i}.png", rgb)

    return np.array(X).reshape(-1, size, size, 1), np.array(y)

X, y = gen_data(2000)
X = X / 255.0
X_train, X_test = X[:1600], X[1600:]
y_train, y_test = y[:1600], y[1600:]

# CNN Model
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(64,64,1)),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['
    accuracy'])

# Checkpoint: Save best model weights
checkpoint_path = "models/best_circle_detector.weights.h5"
checkpoint = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    save_weights_only=True,
    monitor='val_loss',
```

```
    mode='min',
    save_best_only=True,
    verbose=1
)

# Early stopping
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=2,
    restore_best_weights=True,
    verbose=1
)

# history = model.fit(X_train, y_train, epochs=40, validation_data=(X_test,
    y_test))
# Train
history = model.fit(
    X_train, y_train,
    epochs=30,
    validation_data=(X_test, y_test),
    callbacks=[checkpoint, early_stop],
    verbose=0
)


# Extract errors (loss)
train_loss = history.history['loss']
val_loss = history.history['val_loss']

# Generalization error = validation loss (on unseen data)

# Create plot
plt.figure(figsize=(10, 6))
epochs = range(1, len(train_loss) + 1)

plt.plot(epochs, train_loss, 'b-', label='Training Error', linewidth=2)
plt.plot(epochs, val_loss, 'r--', label='Validation Error', linewidth=2)

plt.title('Training, Validation, and Generalization Error', fontsize=14)
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Binary Cross-Entropy Loss', fontsize=12)
plt.legend(fontsize=12)
plt.grid(True, alpha=0.3)
plt.tight_layout()

# Save instead of show
error_plot_path = "plots/error_plot.png"
plt.savefig(error_plot_path, dpi=150, bbox_inches='tight')
plt.close()  # Close figure to free memory


# Predict on photo (generate sample)
def predict_circle(model, img_path=None):
    if img_path:
        img = tf.keras.utils.load_img(img_path, target_size=(64,64),
            color_mode='grayscale')
        img = tf.keras.utils.img_to_array(img) / 255.0

    pred = model.predict(img.reshape(1,64,64,1))[0][0]
```

```
    return "Circle detected" if pred > 0.5 else "No circle"

test_image_path = '/workspaces/ml_tf/data/shapes.jpeg'

print(predict_circle(model, test_image_path))  # Run for demo
```

You Only Look Once (YOLO)

YOLO is a real-time object detection algorithm that frames detection as a single regression problem, predicting bounding boxes and class probabilities directly from full images in one forward pass.

- **Image grid**: The input image is divided into an $S \times S$ grid. Each grid cell is then responsible for detecting objects whose center falls within it.
- **Bounding box prediction**: Each grid cell predicts a fixed number of bounding boxes, along with a confidence score for each box. The confidence score represents how likely the box contains an object and how accurate the box's coordinates are.
- **Class probability prediction**: Each grid cell also predicts the probability for each potential object class (e.g., car, person, dog).
- **Non-Maximum Suppression (NMS)**: The algorithm uses NMS to filter out duplicate and low-confidence bounding boxes, ensuring that only the most accurate and certain detections are kept.
- **Final output**: The final output is a set of bounding boxes with associated class labels and confidence scores for the detected objects.

Pretrained models for YOLO are available via Ultralytics. Task: Explore the site to see the progression of YOLO in recent years and its usage.

```
from ultralytics import YOLO

# Load a pretrained YOLO11n model
model = YOLO("yolo11s.pt")

# Train the model on COCO8
results = model.train(data="coco8.yaml", epochs=10, imgsz=640)

# Validate the model
metrics = model.val()  # no arguments needed, dataset and settings
    remembered
metrics.box.map  # map50-95
metrics.box.map50  # map50
metrics.box.map75  # map75
metrics.box.maps  # a list contains map50-95 of each category

model.predict("/workspaces/ml_tf/data/zebra.jpeg", save=True, imgsz=640,
    conf=0.5)

results = model("/workspaces/ml_tf/data/zebra.jpeg")  # predict on an image

# Access the results
for result in results:
    result.save()
    xywh = result.boxes.xywh  # center-x, center-y, width, height
    xywhn = result.boxes.xywhn  # normalized
    xyxy = result.boxes.xyxy  # top-left-x, top-left-y, bottom-right-x,
        bottom-right-y
```

```
    xyxyn = result.boxes.xyxyn  # normalized
    names = [result.names[cls.item()] for cls in result.boxes.cls.int()]  #
        class name of each box
    confs = result.boxes.conf  # confidence score of each box
```