Return Prediction Part II

Zichao Yang

Zhongnan University of Economics & Law

Date: May 28, 2025

Roadmap

In this chapter, authors discuss different asset pricing methods.

- Penalized Linear Models
- Dimension Reduction
- Decision Trees
- Neural Networks
- Return Prediction Models For “Alternative” Data

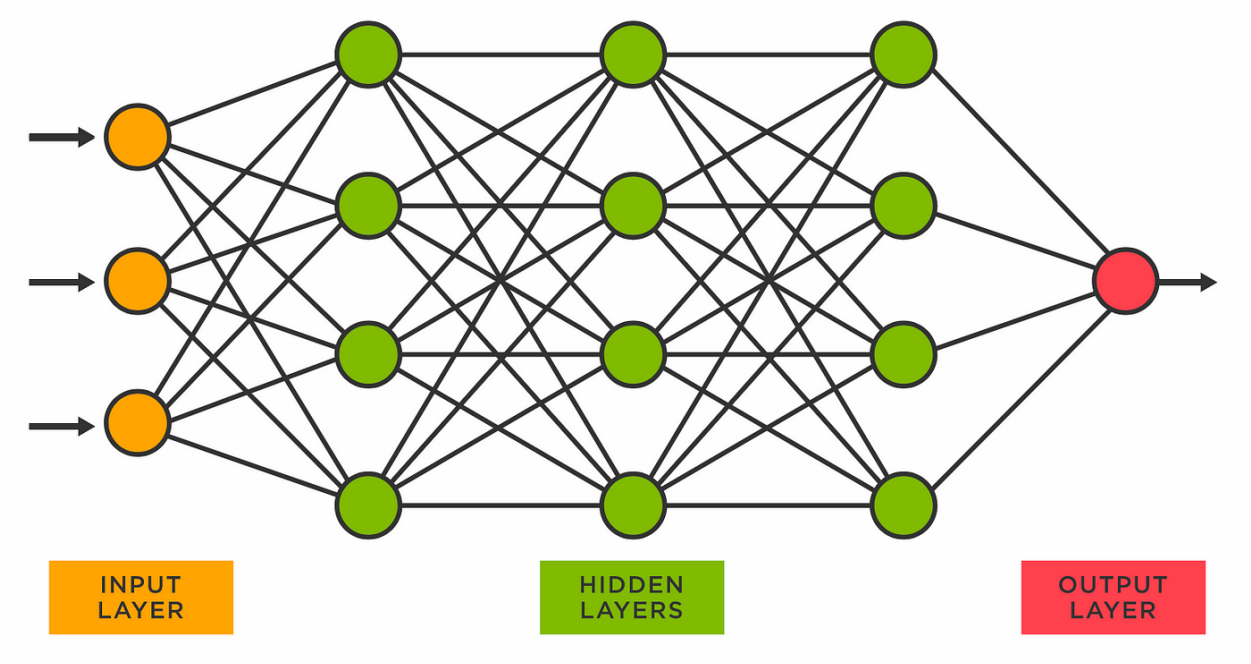
Neural Networks

Neural networks, aka, *deep feedforward networks*, *feedforward neural networks*, *multiplayer perceptrons (MLP)*, are perhaps the most popular and most successful model in machine learning.

They have theoretical underpinnings as "universal approximators" for any smooth predictive function (See *Extra: Universal Approximation Theory* in lecture 4).

Neural networks suffer, however, from **a lack of transparency and interpretability**.

Neural Network Structure



Source: MIT-Quantum Neural Networks
(Alas, this picture shows a plain old-school Neural Network.)

Neural Network Structure

The model in the above picture is called **feedforward** because there are no **feedback** connections in which outputs of the model are fed back into itself.

The overall length of the chain gives the **depth** of the model. The name *deep learning* arose from this terminology.

The dimensionality of these hidden layers determines the **width** of the model. The hidden layer is vector valued, and each element of the vector can be interpreted as playing a role analogous to a neuron.

Design A Neural Network

One way to understand neural networks is to understand how they can overcome the limitations of linear models.

To represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$. Then how to choose the nonlinear transformation ϕ?

- use a very generic ϕ, such as the radial basis function (RBF) kernel. If $\phi(\boldsymbol{x})$ has high enough dimensions, it can work well on the training set, but generalization to the test set often remains poor.
- manually engineer ϕ. It requires decades of human effort for each separate task, and with little transfer between domains.
- use deep learning models to learn ϕ. We parameterize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ and use the optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.

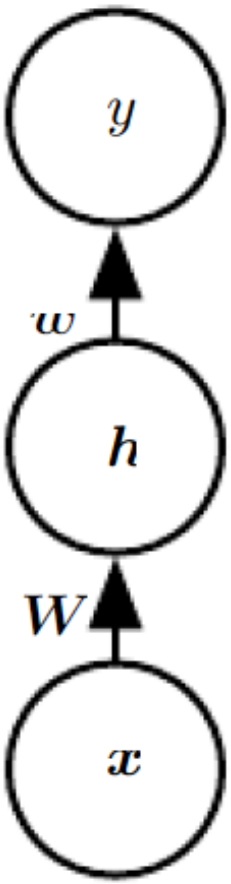
Design A Neural Network

(1) We need to make many of the same design decisions as for a linear model: choosing the optimizer, the cost function, and the form of the output units.

(2) Neural networks have introduced the concept of a hidden layer, and this requires us to choose the **activation functions** that will be used to compute the hidden layer values.

(3) We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should be in each layer.

Activation Functions



The vector of hidden units $\boldsymbol{h}$ are computed by:
$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) = \boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{c}$.
The output layer: $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b) = \boldsymbol{w}^\intercal \boldsymbol{h} + b$

What function should $f^{(1)}$ be?
If $f^{(1)}$ is still linear, then the whole feedforward network will remain as a linear function of its input:
$f^{(2)}(f^{(1)}(\boldsymbol{x})) = (\boldsymbol{W}^\intercal \boldsymbol{x})^\intercal \boldsymbol{w} = \boldsymbol{x}^\intercal \boldsymbol{W} \boldsymbol{w} = \boldsymbol{x}^\intercal \boldsymbol{w}'$, where $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$. (ignoring the intercept terms, $\boldsymbol{c}$ and b)

How to introduce nonlinearity into the system?
Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an **activation function**.

Activation Functions

Hence, we redefine $\boldsymbol{h}$ as:

$$\boldsymbol{h} = g(f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})) = g(\boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{c})$$

where g is the activation function.

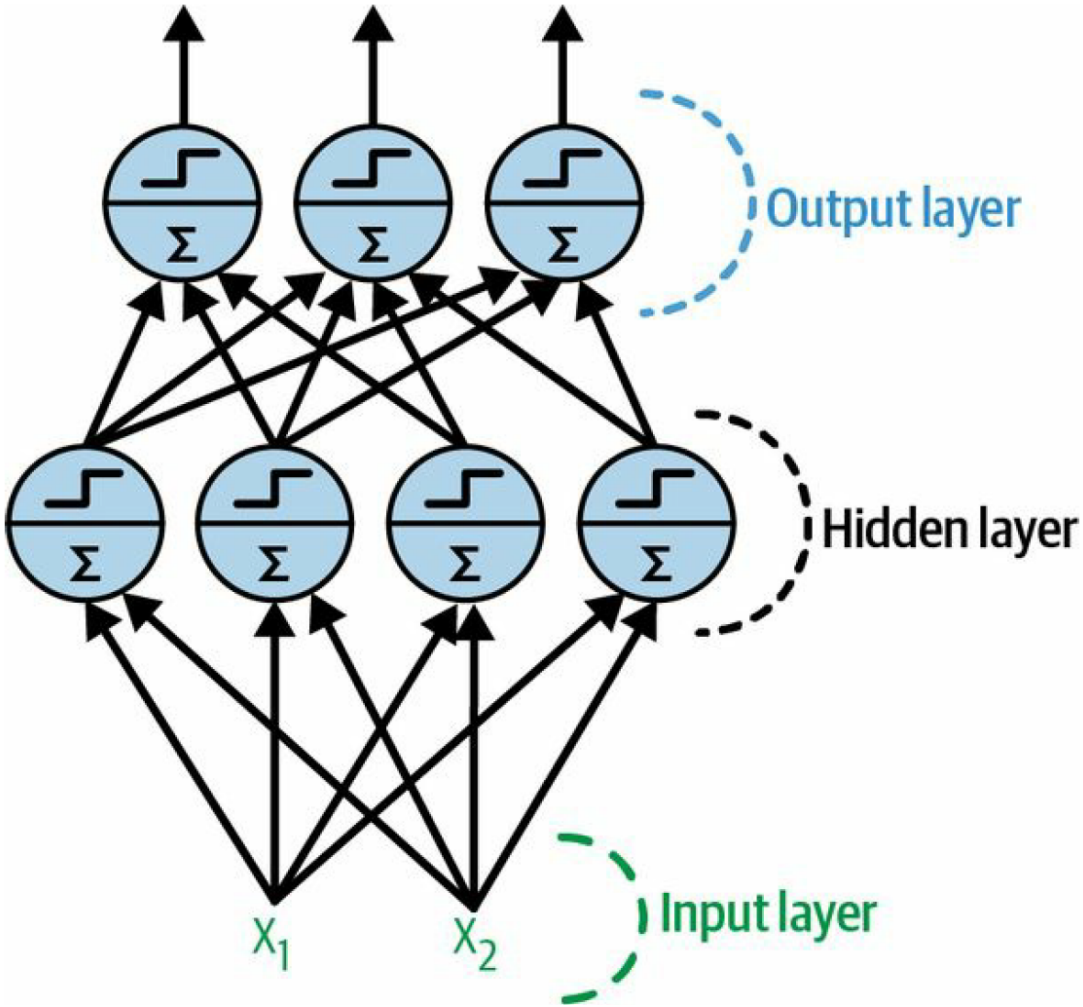
In modern neural networks, the default recommendation for activation function is to use the **rectified linear unit**, or **ReLU**:

$$g(z) = \max\{0, z\}$$

We can now specify our complete network as:

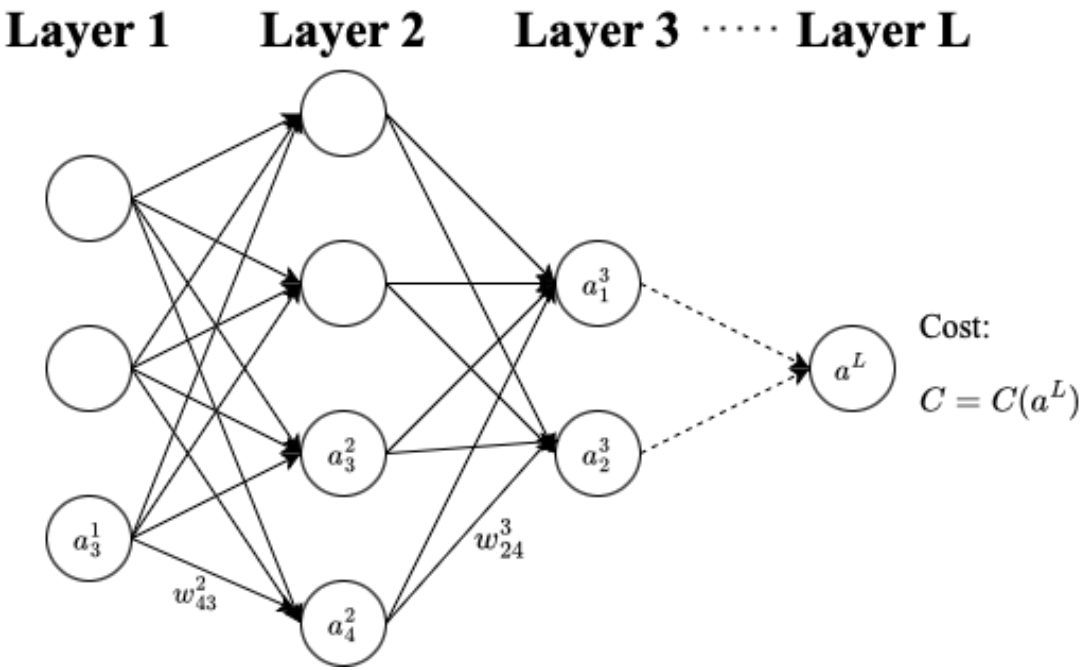
$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\intercal \max\{0, \boldsymbol{W}^\intercal \boldsymbol{x} + \boldsymbol{c}\} + b$$

REAL Neural Network Structure



Source: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Back-Propagation



Note: here we use a MLP model to demonstrate the back-propagation method, but back-propagation is widely used in different ML models.

a_j^l denotes the j^{th} neuron in the l^{th} layer.

w_{jk}^l denotes the weight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer.

Each neuron contains two operations: sum & activation:
$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$,
where $\sigma(\cdot)$ is the activation function, b_j^l is the bias.

Back-Propagation: the Gradient Descent Idea

Gradient descent is a method that allows us to decrease the value of a function $f(\boldsymbol{x})$ by moving in the direction of the negative gradient, $-\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

Gradient descent method proposes a new point $\boldsymbol{x}'$ to minimize $f(\boldsymbol{x})$ based on:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

where ϵ is called as **learning rate**, a positive scalar determining the size of the step.

Back-propagation is built on this thought. And the function $f(\boldsymbol{x})$ we want to minimize here is the **cost/loss function** of the neural network, and $\boldsymbol{x} = [\boldsymbol{w}, \boldsymbol{b}]$.

Back-Propagation: Weight Update

Once the gradients are computed, the weights are updated using an optimization algorithm, typically **stochastic gradient descent (SGD)** or one of its variants like **Adam**(explanation). The basic update rule for a weight w using SGD is:

$$w = w - \lambda \cdot \frac{\partial L}{\partial w}$$

where w is a weight, λ is the **learning rate**, $\frac{\partial L}{\partial w}$ is the gradient of the loss function L with respect to the weight w.

The learning rate λ is a critical hyperparameter that controls how much the weights are changed during each update.

Back-Propagation: Supplement Materials

Also you can check out these supplement materials:

1. 3Blue1Brown - What is backpropagation really doing?

2. Andrej Karpathy - The spelled-out intro to neural networks and backpropagation: building micrograd

3. Michael Nielsen - Chapter 2: How the backpropagation algorithm works

Neural Network in Finance

Gu et al. (2020) perform a comparative analysis of the major machine learning methods by comparing the out-of-sample panel return prediction R^2 across these models.

	OLS +H	OLS-3 +H	PLS	PCR	ENet +H	GLM +H	RF	GBRT +H	NN1	NN2	NN3	NN4	NN5
All	-3.46	0.16	0.27	0.26	0.11	0.19	0.33	0.34	0.33	0.39	0.40	0.39	0.36
Top 1000	-11.28	0.31	-0.14	0.06	0.25	0.14	0.63	0.52	0.49	0.62	0.70	0.67	0.64
Bottom 1000	-1.30	0.17	0.42	0.34	0.20	0.30	0.35	0.32	0.38	0.46	0.45	0.47	0.42

Table 3.4: Monthly Out-of-sample Stock-level Prediction Performance (Percentage R^2_{oos})

Note: In this table, Gu *et al.* (2020b) report monthly R^2_{oos} for the entire panel of stocks using OLS with all variables (OLS), OLS using only size, book-to-market, and momentum (OLS-3), PLS, PCR, elastic net (ENet), generalize linear model (GLM), random forest (RF), gradient boosted regression trees (GBRT), and neural networks with one to five layers (NN1–NN5). "+H" indicates the use of Huber loss instead of the l_2 loss. R^2_{oos}'s are also reported within subsamples that include only the top 1,000 stocks or bottom 1,000 stocks by market value.

Neural Network in Finance

Choi et al. (2022) analyze the same models as Gu et al. (2020) in international stock markets. And they reach similar conclusions that the best performing models are nonlinear.

Interestingly, they demonstrate the viability of **transfer learning**. In particular, a model trained on US data delivers significant out-of-sample performance when used to forecast international stock returns.

Transfer learning: a technique in machine learning where a model developed for a particular task is reused as the starting point for a model on a second task.

Neural Network in Finance

Bali et al. (2020) conduct comparative analyses of machine learning methods for US corporate bond return prediction using a large set of 43 bond characteristics such as issuance size, credit rating, duration, and so forth.

Their comparison of machine learning models in terms of predictive R2 and decile spread portfolio returns largely corroborate the conclusions of Gu et al. (2020).

	OLS	PCA	PLS	Lasso	Ridge	ENet	RF	FFN	LSTM	Combination
R^2_{oos}	−3.36	2.07	2.03	1.85	1.89	1.87	2.19	2.37	2.28	2.09
Avg. Returns	0.16	0.51	0.63	0.39	0.33	0.43	0.79	0.75	0.79	0.67

Table 3.6: Machine Learning Comparative Bond Return Prediction (Bali *et al.*, 2020)

Note: The first row reports out-of-sample R^2 in percentage for the entire panel of corporate bonds using the 43 bond characteristics (from Table 2 of Bali *et al.* (2020)). The second row reports the average out-of-sample monthly percentage returns of value-weighted decile spread bond portfolios sorted on machine learning return forecasts (from Table 3 of Bali *et al.* (2020)).

Neural Network in Scikit-learn

In Scikit-Learn, you can build a neural network using the MLPClassifier and MLPRegressor classes.

Take a look at the user guide and see how many hyperparameters you can fine tune in a neural network. And let's try it out in python.

Q: how to do model selection here?

Hint: we can still use the GridSearchCV function.

Class in Python

A simple example of defining a class in python.

```python
class Dog:
    species = "Canis familiaris"    # define class attribute

    def __init__(self, name, age): # define instance attributes
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} barks {sound}"

# Create an instance of Dog
my_dog = Dog("Ben Ben", 5)
```

Class in Python

Class Inheritance Illustration:

```
# Class Inheritance
class Puppy(Dog):
  def __init__(self, name, age):
    super().__init__(name, age)

  def bark(self, sound="Woof"):
    return self.speak(sound)

my_dog = Puppy('Lab', 6)

my_dog.species
my_dog.name
my_dog.age
my_dog.description()
my_dog.speak('Woof')
my_dog.bark('Yin')
```

Neural Network in PyTorch

Neural networks are a foundational technique in deep learning, which is a subset of machine learning.

We usually deploy deep learning models utilizing dedicated python libraries like PyTorch or Tensorflow.

In this class, we use PyTorch since nowadays it is the default deep learning library used in the research community and among developers working on new deep learning models.

PyTorch Library

PyTorch is an open-source machine learning library developed by Facebook's AI Research lab (FAIR), widely used for applications such as computer vision and natural language processing.

- **Ease of Use**: PyTorch offers a clear and intuitive syntax which makes it accessible for beginners in deep learning.
- **Pythonic Nature**: PyTorch allows developers and researchers to use standard Python features, which makes the process of building and testing neural networks more straightforward.
- **Dynamic Computation Graphs**: PyTorch uses dynamic computation graphs, which helps in automatically calculating gradients.

PyTorch Library

- **Strong GPU Acceleration**: PyTorch seamlessly integrates with CUDA, a parallel computing platform and application programming interface model created by Nvidia.
- **Robust Ecosystem**: PyTorch is supported by a large ecosystem of tools and libraries.
- **Community and Support**: PyTorch benefits from a strong community of developers and researchers who continuously contribute to the library's development.

Now, it is time to install PyTorch in your computer!

CUDA and GPU Computing

Compute Unified Device Architecture (CUDA) is a proprietary parallel computing platform and application programming interface developed by NVIDA.

CUDA is essential to GPU computing:

- **Parallel Processing Capabilities**: GPUs are highly efficient at handling multiple operations concurrently. CUDA provides the necessary tools and frameworks to harness this capability.
- **Optimized Hardware Utilization**: CUDA allows direct access to the GPU's virtual instruction set and parallel computational elements. CUDA-enabled computing is more efficient than CPU-only computing.

Drawback: CUDA is only available on certain NVIDA GPUs.

PyTorch Installation

Check CUDA compatibility.

Install PyTorch (CUDA or CPU). Go to the PyTorch website and follow the instruction.

PyTorch takes care of CUDA and cuDNN installations automatically.

Check GPU Compatibility

Type command **nvidia-smi** in your terminal, you should see the following result:

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 560.94                 Driver Version: 560.94         CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                     Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3060 Ti   WDDM  |   00000000:09:00.0  On |                  N/A |
|  0%   40C    P8             13W /  200W |    1272MiB /   8192MiB |      2%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
```

Neural Network in PyTorch

(1) Enable GPU computing.

(2) Create a Custom Dataset for your own project.

(3) Build a Neural Network Model: model structure, activation function, loss function, optimizer.

(4) What is batch processing? What is an epoch? Source: Batch vs Epoch

(5) How to do hyperparameter tuning in PyTorch? optuna

Activation Functions

For hidden layers:

(1) **Rectified Linear Unit (ReLU)**

$$f(x) = \max(0, x) \in [0, \infty]$$

(2) **Leaky ReLU**

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

where α is a small positive constant.

(3) **Hyperbolic Tangent (Tanh)**

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in (-1, 1)$$

Activation Functions

For the output layer:

(1) **Sigmoid Function**: binary classification problems.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \in (0, 1)$$

(2) **Softmax Function**: multi-class classification problems.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

where $\mathbf{z}$ is the input vector, z_i is the element for class i, and C is the number of classes.

Loss Functions

For Regression Tasks (Predicting Continuous Values):

- **Mean Squared Error** (aka, L2 Loss): $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- **Mean Absolute Error** (aka, L1 Loss): $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$

where,
n: the number of samples
y_i: the true value for sample i
$\hat{y}_i$: the predicted value for sample i

Loss Functions

For Classification Tasks (Predicting Discrete Categories):

(1) **Binary Cross-Entropy (BCE) Loss** for binary classification problems

$$\text{BCE} = -(y \log(p) + (1 - y) \log(1 - p))$$

where y is the true label (0 or 1) and p is the predicted probability of the input belonging to the positive class.

(2) **Categorical Cross-Entropy (CE) Loss** for multi-class classification problems.

$$\text{CE} = -\sum_{i=1}^{C} y_i \log(p_i)$$

where C is the number of classes, y_i is the true label for class i, p_i is the predicted probability of the input belonging to class i.

More on Cross-Entropy Loss

When we use **Categorical Cross-Entropy Loss**, the true labels are **one-hot encoded**.

If the true labels are provided as integers representing the class index, rather than one-hot encoded vectors. We should use **Sparse Categorical Cross-Entropy Loss**, which basically performs a one-hot encoding of the true labels before calculating the cross-entropy loss.

Optimizer

(1) **Stochastic Gradient Descent (SGD)**

For each mini-batch, it calculates the gradient of the loss and updates the parameters in the opposite direction of the gradient, scaled by the learning rate.

Update Rule (for a single θ):

$$\theta = \theta - \alpha \nabla J(\theta; x^{(i)}, y^{(i)})$$

where θ is the parameter being updated, α is the learning rate, $\nabla J(\theta; x^{(i)}, y^{(i)})$ is the gradient of the loss function J with respect to the parameter θ.

Optimizer

(2) **Adaptive Moment Estimation (Adam)**
Adam is the most popular and widely used optimizers. It follows the following update rule:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \text{(First moment estimate)}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \text{(Second moment estimate)}$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \text{(Bias correction for first moment)}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \text{(Bias correction for second moment)}$$
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \text{(Parameter update)}$$

where m_t and v_t are the biased first and second moment estimates, β_1 and β_2 are decay rates (typically close to 1), α is the learning rate, g_t is the gradient, and ϵ is a small value for numerical stability.

Pytorch Code

(1) Activation Functions

```
import torch.nn as nn
relu = nn.ReLU()
leaky_relu = nn.LeakyReLU(negative_slope=0.01)
tanh = nn.Tanh()
sigmoid = nn.Sigmoid()    # we usually don't use it directly
softmax = nn.Softmax(dim=0) # we usually don't use it directly
```

(2) Loss Functions

```
bce_loss = nn.BCELoss() # we usually don't use it directly
# Combines Sigmoid and BCELoss for numerical stability.
bce_with_logits_loss = nn.BCEWithLogitsLoss()
# nn.CrossEntropyLoss() combines Softmax with CrossEntropy
cross_entropy_loss = nn.CrossEntropyLoss()
```

(3) Optimizer

```
import torch.optim as optim
optimizer_sgd = optim.SGD()
optimizer_adam = optim.Adam()
```

Deep Learning Methods for Alternative Data

In recent years, alternative data has become a popular topic in economic and financial research.

There are two major categories of alternative data: **text data** and **image data**. Due to the rise in popularity of large language models, text data has become an increasingly important data source in recent academic research (Gentzkow et al., 2019; Ash and Hansen, 2023).

Deep Learning Methods for Alternative Data

Part I. Deep Learning Methods for Text Data

Textual Analysis

Textual analysis is among the most exciting and fastest growing frontiers in finance and economics research.

Jegadeesh and Wu (2013), Ke et al. (2019), and Garcia et al. (2022) are examples of supervised learning models customized to the problem of return prediction using a term count or "bag of words" (BoW) representation of text documents.

Ke et al. (2019) propose a method, called "SESTM" (Sentiment Extraction via Screening and Topic Modeling), which provides a data-driven methodology for constructing sentiment dictionaries that are customized to specific supervised learning tasks.

Textual Analysis

SESTM has three central components:

- The first step isolates the most relevant terms from a very large vocabulary of terms via predictive correlation screening.
- The second step assigns term-specific sentiment weights using a supervised topic model.
- The third step uses the estimated topic model to assign article-level sentiment scores via penalized maximum likelihood.

Textual Analysis

The traditional text representation methods (i.e., Bow, TF-IDF, N-grams) have some limitations:

- all of these methods only accesses the information in text that is conveyable by **term usage frequency**. It sacrifices nearly all information that is conveyed through **word ordering** or **contextual relationships between terms**.
- the ultra-high dimensionality of BoW representations leads to statistical inefficiencies, downstream models must include many parameters to process all these terms despite many of the terms conveying negligible information.
- these dimension-reduced representations are corpus specific. For example, when Bybee et al. (2020) build their topic model, the topics are estimated only from The Wall Street Journal.

Textual Analysis

Chen et al. (2023) construct refined news text representations derived from so-called "large language models" (LLMs), and they analyze return predictions based on a news text processed through a number of LLMs.

They find that predictions from pre-trained LLM embeddings outperform prevailing text-based machine learning return predictions in terms of out-of-sample trading strategy performance, and that the superior performance of LLMs stems from the fact that they can more **successfully capture contextual meaning in documents**.

In the next section, we delve into LLMs.

Large language Models

Large Language Models (LLMs) are neural networks and are pre-trained using self-supervised learning or semi-supervised learning.

LLMs use massive amount of data to learn billions of parameters and can achieve the ability of general-purpose language understanding and generation.

Recent years, the deep learning architecture known as the **transformer** has come to dominate the LLM research field, superseding other deep learning structures such as CNNs, RNNs, and LSTMs.

Online Large Language Models

Large Language Models (LLMs) are complex. Hence, deploying them for real-time applications requires substantial computing power to maintain low-latency responses, and often requires dedicated servers or cloud resources.

Online LLMs becomes the default choice for most individual users:

- ChatGPT-OpenAI
- Gemini-Google
- Grok-xAI
- Claude-Anthropic
- DeepSeek
- Kimi-Moonshot
- Doubao-ByteDance

Local Large Language Models

Some users may prefer to process all their data locally rather than uploading it to the cloud due to various concerns.

Recently, various local LLM applications have been developed to cater to this need.

How to deploy an LLM on a local PC?

(1) method 1: Ollama and Local LLMs

(2) method 2: LM Studio

(3) API: Ollama Python Library and LM Studio Python Library

Flagship LLM APIs for Optimal Performance

For peak performance, it is recommend to utilize APIs from leading AI companies hosting large language models.

Here we use Volcengine from ByteDance to demonstrate how to use online LLM API to achieve optimal performance.

Two models:

- DeepSeek-R1 (model ID: deepseek-r1-250120)
- DeepSeek-V3 (model ID: deepseek-v3-250324)

Large Language Model Fine-tuning: DSPy Framework

We write prompts to interact with language models. To tackle complex problems, we usually need to:

- break the problem down into steps
- prompt your LM well until each step works well in isolation
- tweak the steps to work well together
- generate synthetic examples to tune each step
- use these examples to finetune smaller LMs to cut costs

DSPy provides an algorithmical way to optimize LM prompts and weights, especially when LMs are used multiple times within a pipeline.

DSPy User Guide: https://dspy.ai/tutorials/

Important Concepts in DSPy

(1) DSPy Signatures

A signature is a declarative specification of input/output behavior of a DSPy module. Signatures allow you to tell the LM what it needs to do, rather than specify how we should ask the LM to do it.

(2) DSPy Modules

Here DSPy provides different modules for different tasks.

(3) DSPy Optimizers

A DSPy optimizer is an algorithm that can tune the parameters of a DSPy program to maximize the metrics you specify, like accuracy.

Let's code it out!

LLM Fundamentals: Tokenization

Tokenization is the process of breaking down a text into individual units, often words or subwords. These units are called tokens.

Tokenization serves several functions, including:

- **Text Preprocessing:** Tokenization is the initial step in preparing text data for NLP tasks. It simplifies the text by segmenting it into manageable pieces.
- **Feature Extraction:** Tokens are used as features in NLP models, and they help represent the text data in a numerical format that models can work with.
- **Text Analysis:** Tokenization is essential for text analysis tasks like sentiment analysis, text classification, and named entity recognition.

Types of Tokenization

- **Word Tokenization:** This method breaks text down into individual words. It's the most common approach and is particularly effective for languages with clear word boundaries.
- **Character Tokenization:** The text is segmented into individual characters. This method can save memory but may affect model performance.
- **Subword Tokenization:** Striking a balance between word and character tokenization, this method breaks text into units that might be larger than a single character but smaller than a word.

Subword Tokenization is useful in dealing with unknown words. For example, the model corpus may not include *unlikeliest*, but the model can back out the meaning if the corpus contains *un-*, *likely* and *-est*. This method is commonly used in large language models.

Tokenization Example

BERT model uses subword tokenization:

```python
import torch
from transformers import BertTokenizer, BertModel

# Define device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load pre-trained BERT model and tokenizer
model = BertModel.from_pretrained('bert-base-uncased').to(device)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Define a sample sentence
text = "BERT uses subword tokenization."

# Tokenize the text
tokens = tokenizer.tokenize(text)
print(tokens)
```

Tokenization Example

In the above example, we get the following result: [‘bert’, ‘uses’, ‘sub’, ‘##word’, ‘token’, ‘##ization’, ‘.’]

Here, “subword” and “tokenization” are divided into multiple tokens.

GPT also provides a nice online tool to illustrate the concept of tokenization:

GPT-Tokenizer

LLM Fundamentals: Word Embedding

Word embedding is a technique for representing words or tokens in a continuous vector space.

Word embedding serves several functions, including:

- **Semantic Representation:** It captures semantic relationships between words. Words with similar meanings are represented closer in the vector space.
- **Dimensionality Reduction:** Embeddings reduce the high dimensionality of one-hot encoded words, making it easier for models to work with.
- **Improving Model Performance:** Pretrained word embeddings, such as Word2Vec, GloVe, and FastText, are used to initialize models, improving their performance in various NLP tasks.

Common Word Embedding Methods

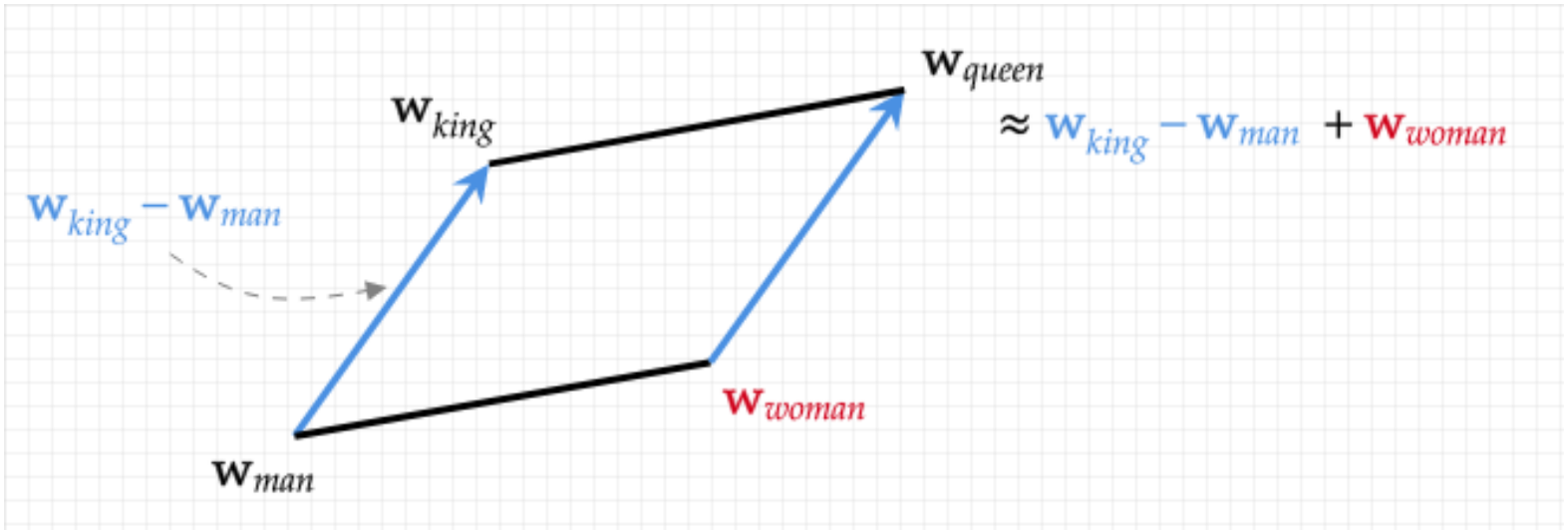
1. **Word2Vec**:

- Word2Vec is developed by Google.
- Word2Vec utilizes either of two model architectures: Continuous Bag of Words (CBOW) and Skip-gram.
- Word2Vec represents words as dense vectors, and it's trained to predict context words given a target word (Skip-gram) or predict a target word given context words (CBOW).
- Word2Vec captures semantic relationships between words, and similar words are represented by vectors that are close in vector space.

Word Embedding Example

The famous Word2Vec embedding example:

$$King – Man + Woman = Queen$$



Credit: School of Informatics, University of Edinburgh

Reference: Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems 26 (2013).

Common Word Embedding Methods

2. **BERT** (Bidirectional Encoder Representations from Transformers):

- BERT is a transformer-based model developed by Google.
- Unlike traditional word embeddings, BERT provides contextual embeddings.
- It is pre-trained on a large corpus and can be fine-tuned for various NLP tasks.
- BERT captures bidirectional context information and has achieved state-of-the-art results in numerous NLP tasks.

Word Embedding Example

BERT word embedding:

```
# Tokenize and Encode Text
inputs = tokenizer(text, return_tensors="pt")
print(inputs)
# results:
# {'input_ids': tensor([[  101, 14324,  3594,  4942, 18351, 19204,
#         3989,  1012,   102]]), 'token_type_ids': tensor([[0, 0, 0, 0,
# 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1,
# 1, 1, 1]])}

# compare print(tokens) and print(inputs), do you see a difference?

# Obtain Word Embeddings
outputs = model(**inputs)
embeddings = outputs.last_hidden_state
# to access the word embedding for word 'BERT'
embeddings[0][1].shape
# in this model, a token is represented by a 1*768 vector
```

LLM Fundamentals: Attention Mechanism

Attention mechanism is the cornerstone of the **transformer** architecture.

The central idea of attention, was first developed by Graves (2013) in the context of handwriting recognition. Bahdanau et al. (2015) extended the idea, named it "attention" and applied it to machine translation.

In 2017, in their influential paper, *Attention Is All You Need*, Ashish Vaswani et al., introduced the transformer architecture built on the **self-attention** mechanism.

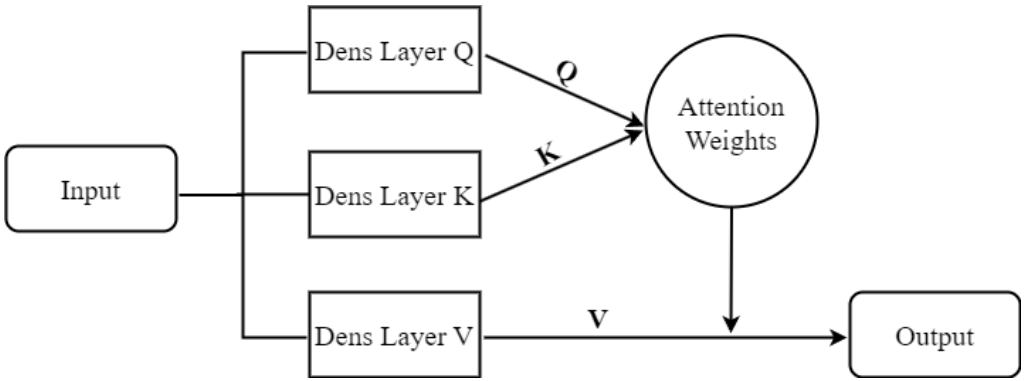
Attention: The Intuition

Consider a sentence: *I would like to have a sandwich for lunch.*

If we ask humans to pick out the words that they should pay most attention to, most likely they will pick "I", "sandwich" and "lunch".

Attention seeks to mimic this mechanism by adding **attention weights** to a model as trainable parameters to pick out important parts of our input.

An Illustration of the Self-attention Mechanism



To generate the Q, K and V vectors, we apply linear transformations (Dens Layer Q, K and V) to the embedded representations of the input.

The linear transformations are defined by learnable weight matrices. These weight matrices are part of the model's parameters and are updated during training.

Attention VS. Self-attention: in self-attention, Q, K and V are generated from the same input. In general attention, they are not.

Attention Function

To generate the output, we need to run the following function:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V$$

Where Q, K and V are different linear transformations of the same input, usually they have the same shape (batch size, n, d). n: number of tokens, d: embedding size.

$softmax(.)$: is a mathematical function that transforms a vector of real numbers into a probability distribution.

Attention Function

Q1: why the dot product QK^T is used to measure the similarity?

We know that $v_i u_j = cos(v_i, u_j)||v_i||_2||u_j||_2$ is a measure of how similar vectors v_i and u_j are. The closer the direction and the larger the length, the greater the dot product.

Similarly, here Q_i and K_i are different projections of the token i into a d dimensional space. We can think about the dot product of these projections as a measure of similarity between token projections.

Attention Function

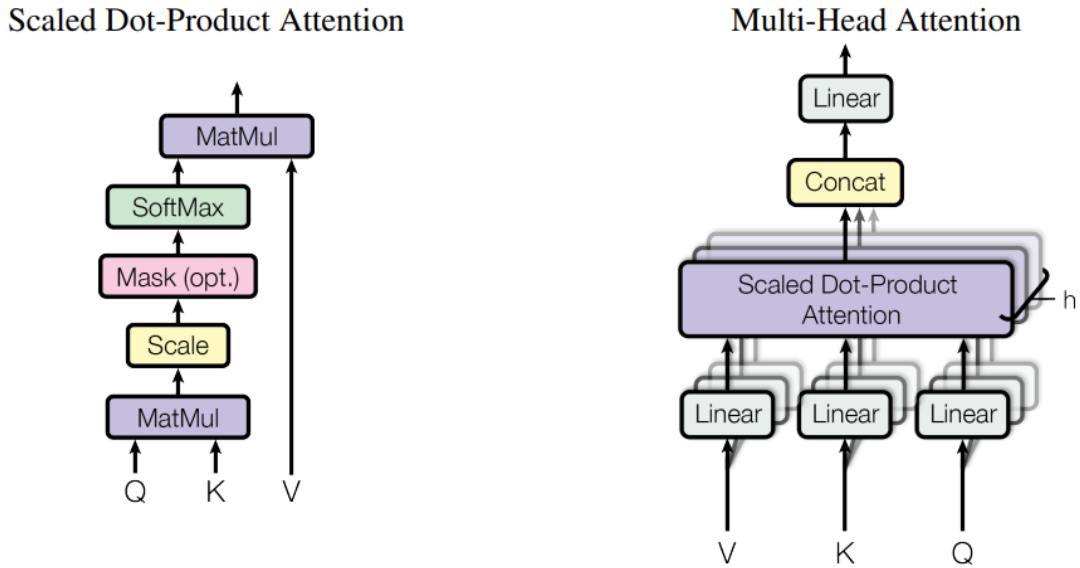
Q2: why do we need to scale the dot product QK^T by $\sqrt{d}$?

Basically it is used to reduce the variance of the dot product. It is a common technique to ensure numerical stability during training and reduce the risk of vanishing or exploding gradients.

Let's code it out!

Multi-head Attention

The transformer does not use attention directly. Instead, it employs a mechanism called multi-head attention.



Source: Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

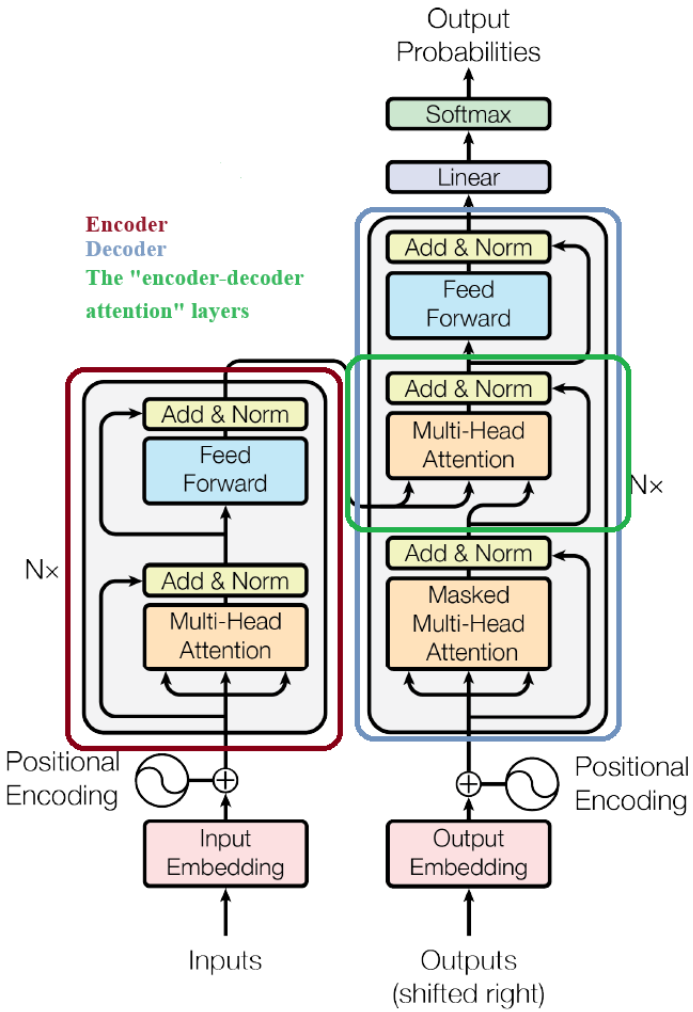
The Multi-Head Self-Attention cuts the Q, K, V into multiple chunks and implement Self-Attention on each chunk, then combines the results together and runs it through a dense layer to get the final output.

Advantages of Multi-head Attention

1. **Improve Model Capacity**: Multi-head attention allows the model to focus on different parts of the input sequence simultaneously. Each head can learn different relationships and patterns in the data, which makes the model more robust and capable of handling complex relationships within the data.

2. **Parallelization**: Multi-head attention can be parallelized, which can lead to faster training and inference times. This is imperative for training a large model with billions of parameters.

LLM Fundamentals: Transformer Structure



Source: Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

Encoder-Decoder

- **Encoder**:
  - The encoder is the first part of a sequence-to-sequence model, and its primary function is to process and represent the input data.
  - Besides using Attention, the encoder can also be one or more layers of RNN or LSTM, etc.
- **Decoder**:
  - The decoder is the second part of a sequence-to-sequence model, and its primary role is to generate an output sequence based on the encoded input data.
  - The decoder is typically designed to be autoregressive. It generates one element of the output sequence at a time while conditioning on the previously generated elements. (i.e., text generation tasks)
  - Besides using Attention, the decoder can also be one or more layers of RNN or LSTM, etc.

Miscellaneous

- **Positional Encoding**: enable the model to make use of the order of the sequence, and improve the model performance.
- **Residual connection & layer normalization**: improve the model performance.
- How to code out the Transformer architecture is beyond the scope of this lecture, but you can find numerous replications online. A good starting point is the Paper With Code Website.

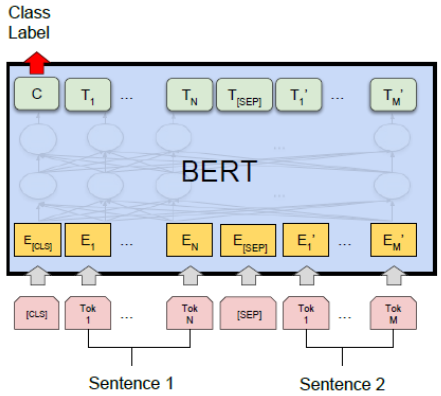
Next, we will discuss two important models built on the Transformer architecture: BERT and GPT.

Bidirectional Encoder Representations from Transformers (BERT)

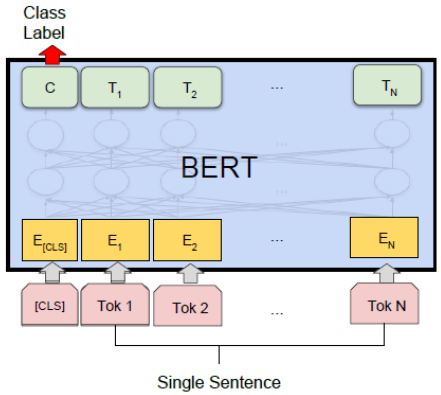
- BERT was developed by Google AI in 2018.
- BERT only utilizes the encoder part of the Transformer.
- BERT's key innovation lies in its bidirectional training approach by using a "masked language model" pre-training objective, inspired by the Cloze task (Taylor, 1953).

BERT Applications

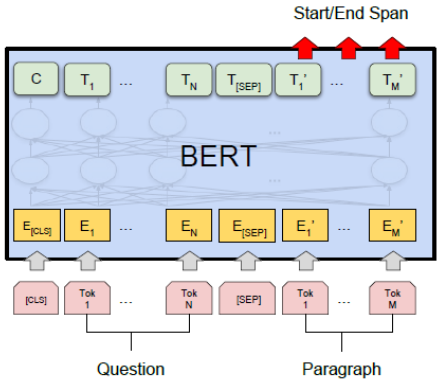
BERT is specifically trained for the following tasks:



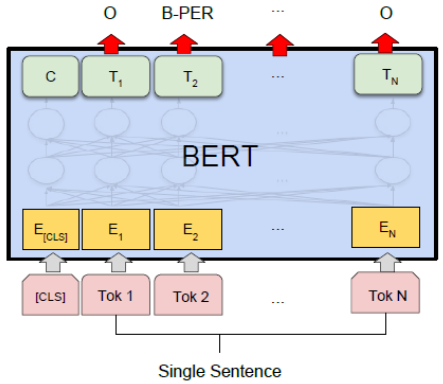
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA



(c) Question Answering Tasks: SQuAD v1.1



(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Source: Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

Generative Pre-trained Transformer (GPT)

AB

Write me a short introduction (under 200 words) to GPT.

Generative Pre-trained Transformer, commonly known as GPT, represents a groundbreaking advancement in natural language processing (NLP). GPT is a family of deep learning models developed by OpenAI that has revolutionized how we interact with and understand text. These models are pre-trained on vast amounts of text data, enabling them to generate human-like text and perform a wide range of NLP tasks.

GPT's key innovation lies in its ability to predict the next word in a sentence, which results in a model that can understand context, grammar, and semantics in a given text. It uses a transformer architecture, which excels at handling sequential data and capturing long-range dependencies in language.

Applications of GPT span various domains, from automated content generation and language translation to chatbots and text summarization. In academia, GPT has catalyzed research in areas such as language understanding, sentiment analysis, and text generation. It has also played a significant role in advancing the field of machine learning and has spurred numerous research papers and developments.

For more comprehensive information on GPT, you can visit the official OpenAI website: OpenAI GPT. Confidence level: High.

GPT Introduction (Human Version)

- The first GPT model was developed by OpenAI in 2018. Now the latest GPT model is GPT-4.5.
- In the business realm, GPT has truly excelled as the most successful large language model. ChatGPT, the chatbot built on the GPT model, is a standout example of this achievement.
- In contrast to BERT, GPT only uses the decoder part of the Transformer.
- The latest GPT-4.5 model is a closely guarded business secret. Given the model contains 1.7 trillion parameters, it is beyond the scope of this lecture to delve into its details.

Application of GPT

- **Text Generation:** GPT can generate coherent and contextually relevant text, making it valuable for creative writing and content creation.
- **Language Translation:** GPT can be fine-tuned for language translation tasks, aiding in machine translation.
- **Question Answering:** GPT can understand and answer questions based on provided text, used in chatbots and virtual assistants.
- **Text Summarization:** GPT can generate concise summaries of longer texts, useful in news aggregation and document summarization.
- **Sentiment Analysis:** GPT can analyze and understand sentiment in text, benefiting market research and customer feedback analysis.

Transformer, BERT & GPT

The relationships between Transformer, BERT and GPT

- BERT and GPT are both built on the Transformer architecture.
- BERT focuses on bidirectional contextual understanding of language, as it trains on both the left and right contexts of a word. This is particularly useful for tasks like text classification.
- GPT, on the other hand, is designed for autoregressive text generation. It predicts the next word in a sentence based on the preceding context and is known for its strong language generation capabilities.

Extra Learning Material

3Blue1Brown has produced some amazing educational videos on the transformer framework, check them out:

(1) 3Blue1Brown-But what is a GPT? Visual intro to transformers

(2) 3Blue1Brown-Attention in transformers, visually explained

Deep Learning Methods for Alternative Data

Part II. Deep Learning Methods for Image Data

Image Analysis

Finance Management

FUNDAMENTAL vs TECHNICAL ANALYSIS

Basis	Fundamental Analysis	Technical Analysis
Relevance	For long term investment.	For short term investment.
Function	Useful for trading & investing.	Useful for trading.
Objective	To find out fair value of security	Determine correct time to enter & exit trade.
Data	Use both past & present data.	Use past data only.
Form of data	Uses annual reports, news, economy`s statistic, etc.	Relies on chart analysis only.
Trader`s type	Long term position trader.	Swing & short term trader.
Concepts	Return on equity & on assets.	Dow theory & price data.
Usefulness	Identify undervalued or overvalued stocks	Determines right time to buy or sell stock.

Source: Fundamental vs Technical Analysis – All You Need to Know

Image Analysis

Technical Analysis: what do investors look at?

Image Analysis

Jiang et al. (2022) represent historical prices as an image and use well-developed convolutional neural network (CNN) machinery for image analysis to search for predictive patterns.

Their images include daily opening, high, low, and closing prices (ofter referred to as an "OHLC" chart) overlaid with a multi-day moving average of closing prices and a bar chart for daily trading volume.

In their model, a CNN is designed to automatically extract features from images that are predictive for the supervising labels (which are future realized returns).

So what is CNN?

Convolutional Neural Networks

Convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples: (1) time-series data (1-D grid), (2) image data (2-D or 3-D grid).

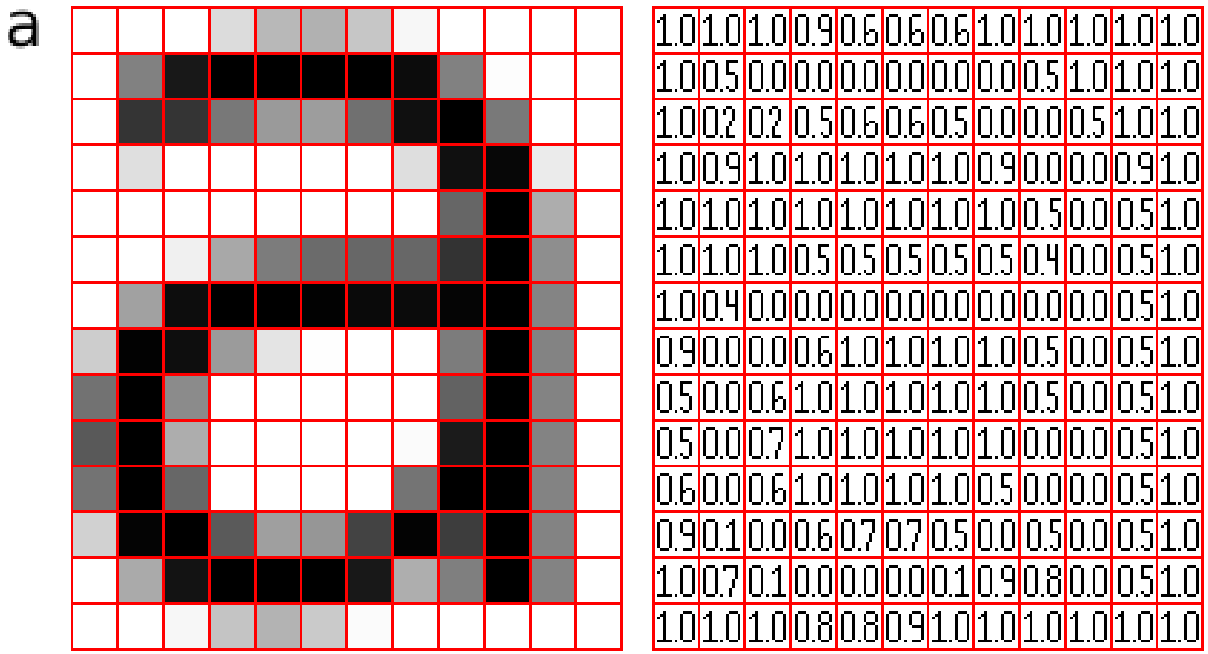
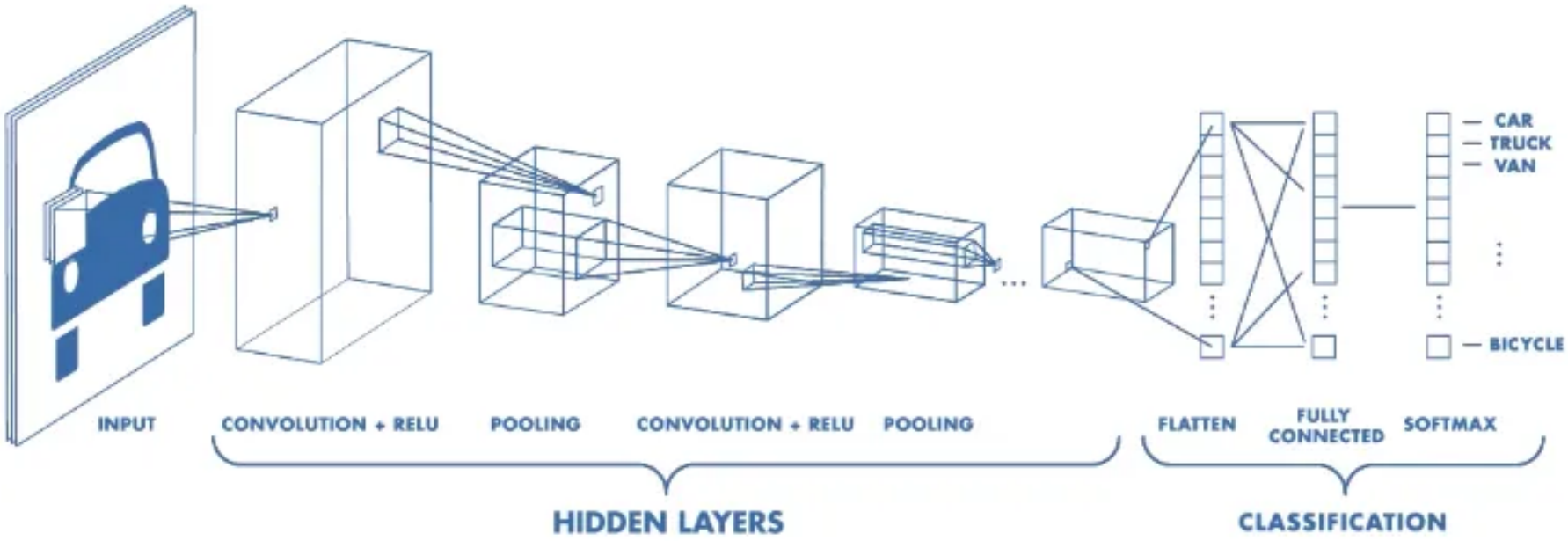


Figure 1: Representation of image as a grid of pixels

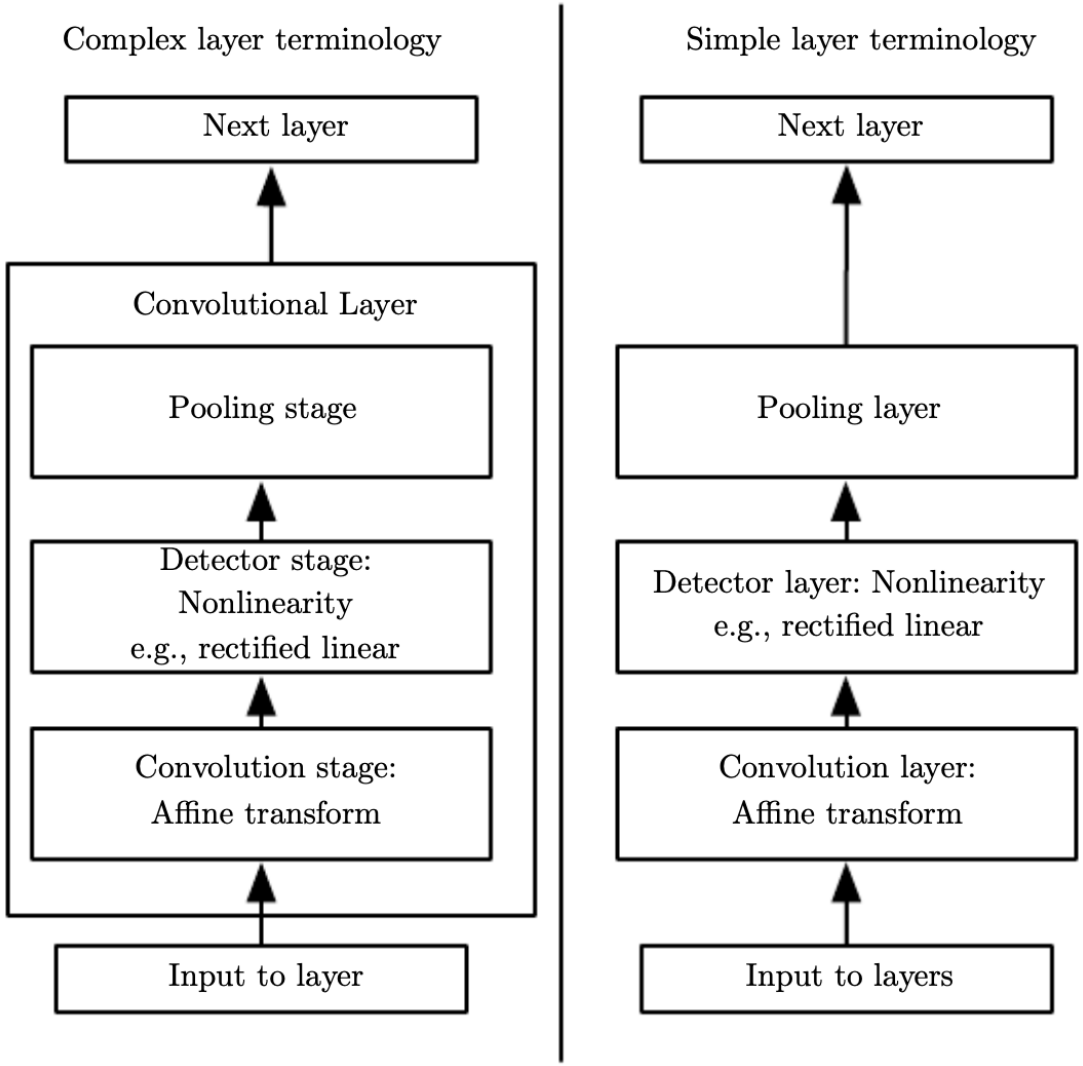
Source: Toward Data Science-Convolutional Neural Networks, Explained

Architecture of a CNN



Source: Toward Data Science-Convolutional Neural Networks, Explained

Components of a Convolutional Layer



Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Implementation of Convolution

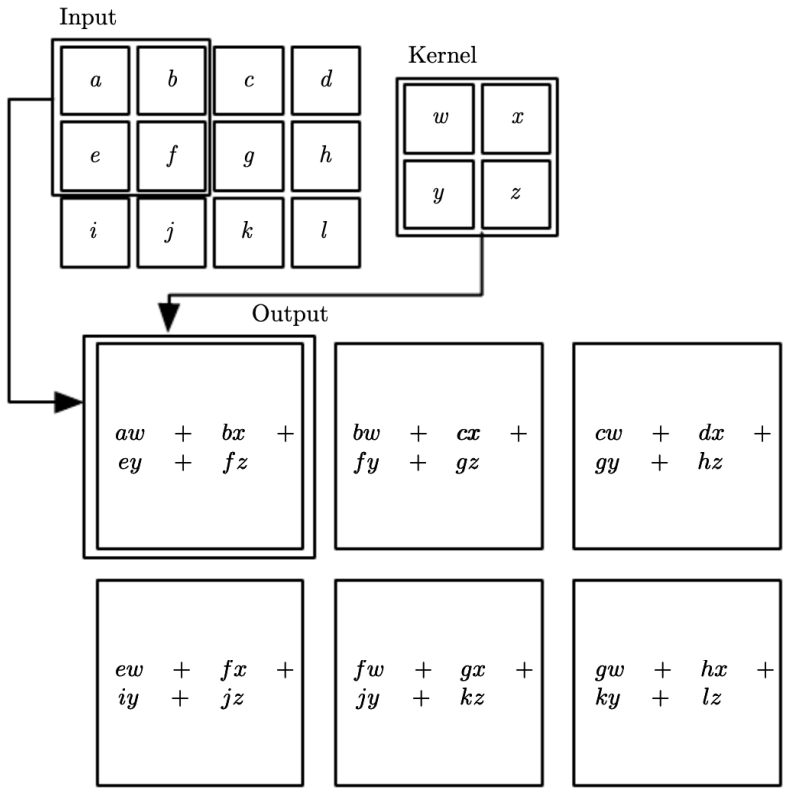


Figure 9.1: An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Sparse Interactions

Convolution leverages three important ideas that can help improve a machine learning system: **sparse interactions**, **parameter sharing** and **equivariant representations**.

In feedforward neural networks, every output unit interacts with every input unit.

Convolutional networks, however, typically have **sparse interactions** (also referred to as sparse connectivity or sparse weights). This is accomplished by making **the kernel smaller than the input**.

This means that we need to store fewer parameters, which: (1) reduce the memory requirements of the model; (2) improve model's statistical efficiency; (3) require fewer operations to compute the output.

Sparse Interactions

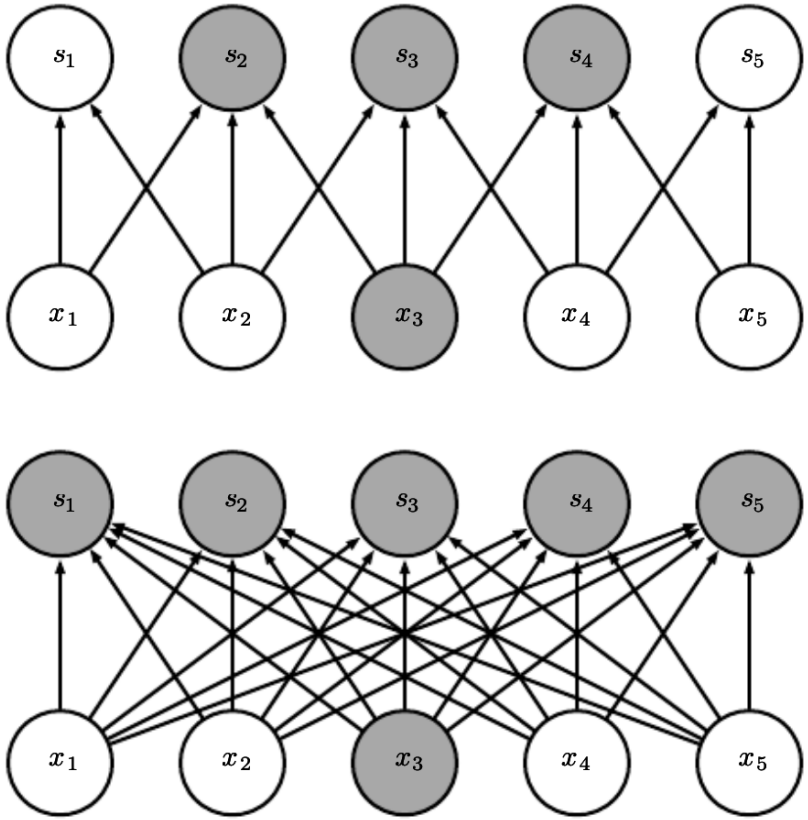


Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, x_3, and also highlight the output units in $\boldsymbol{s}$ that are affected by this unit. *(Top)*When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three outputs are affected by $\boldsymbol{x}$. *(Bottom)*When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3.

Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Parameter Sharing

Parameter sharing refers to using the same parameter for more than one function in a model.

In a feedforward neural network, each element of the weight matrix is used exactly once when computing the output of a layer, and the weight is never used again.

In a convolutional neural network, each member of the kernel is used at every position of the input (except perhaps some boundary positions). For every location, the algorithm apply the same set of weights.

Parameter sharing does not reduce the runtime of forward propagation, but it does further **reduce the storage requirements** of the model to the size of kernel K.

Parameter Sharing

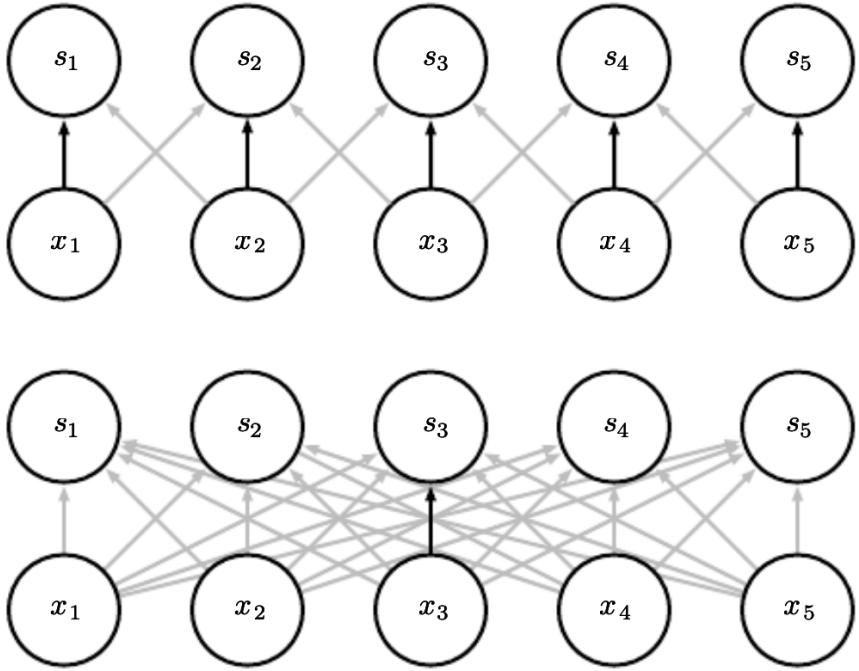


Figure 9.5: Parameter sharing: Black arrows indicate the connections that use a particular parameter in two different models. *(Top)*The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. *(Bottom)*The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Equivariant Representations

We say a function $f(x)$ is **equivariant** to a function g if $f(g(x)) = g(f(x))$. Why is the property useful?

Suppose a convolution layer creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output.

Hence, when we know that some function of a small number of neighboring pixels is useful, we are certain that these functions can be applied to multiple other input locations. For example, if we want to detect edges, the same edges appear everywhere in the image, so it is practical to share parameters across the entire image.

Note: convolution is not naturally equivariant to certain transformations, such as changes in the scale or rotation of an image.

Some Special Kernels

In most cases, the weights in kernels are learned from data via backpropagation, but there are some famous pre-determined kernels:

1	0	-1
1	0	-1
1	0	-1

Vertical Kernel

-1	0	1
-2	0	2
-1	0	1

Sobel Kernel

-3	0	3
-10	0	10
-3	0	3

Scharr Kernel

1	1	1
0	0	0
-1	-1	-1

Horizontal Kernel

-1	-2	-1
0	0	0
1	2	1

Sobel Kernel

-3	-10	-3
0	0	0
3	10	3

Scharr Kernel

Some Special Kernels

Vertical (Horizontal) Kernel: designed to detect vertical (horizontal) edges in an image by detecting in the horizontal (vertical) direction.

Sobel Kernel: a popular edge-detection filter that is used to find the gradient of an image's intensity at each point.

Scharr Kernel: another edge-detection filter that is designed to reduce the impact of noise and to provide a better approximation to the gradient magnitude than the Sobel kernel.

These pre-determined kernels are not learned from data, but they can be applied to images as a pre-processing step to enhance edges and other features before feeding the images into a CNN.

Zero Padding

Zero-padding refers to surrounding the input array with zeroes. Here are the primary functions of zero padding:

- control over output size. Zero padding the input allows us to control the kernel width and the size of the output independently.
- maintain border information. Without padding, the border pixels would be less frequently visited by the kernel, leading to potential loss of important edge information.
- unify varying input sizes.

Stride

In CNNs, **stride** refers to the number of pixels by which the kernel (or filter) moves across the input image. It is an important hyperparameter that controls the spatial dimensions of the output feature map.

Stride = 1: the most common setting where the filter moves one pixel at a time. The kernel is applied at every possible location in the input image.

Stride > 1: kernel skips over multiple pixels during its application. This operation reduces the spatial dimensions of the output feature map.

Stride

The size of the output feature map can be calculated using the following formula:

$$\text{Output Size} = \lfloor \frac{\text{Input Size} - \text{Kernel Size} + 2 * \text{Padding Size}}{\text{Stride}} \rfloor + 1$$

Where:

- Input Size is the height or width of the input image
- Filter Size is the height or width of the kernel
- Padding Size is the number of pixels added to the border of the input image
- Stride is the step size of the kernel
- $\lfloor \cdot \rfloor$ denotes the floor function, which rounds down to the nearest integer

Different Convolutions

Suppose the input image has width of m and the kernel has width of k.

- **Valid Convolution**: no zero-padding is used whatsoever, and the convolution kernel is only allowed to visit positions where the entire kernel is contained entirely within the image. The size of the output is $m - k + 1$.
- **Same Convolution**: just enough zero-padding is added to keep the size of the output equal to the size of the input, m.
- **Full Convolution**: enough zero-padding is added for every pixel to be visited k times in each direction, resulting in an output of size $m + k - 1$.
- **Unshared Convolution**: aka. locally connected layers.
- **Tiled Convolution**: a compromise between a convolutional layer and a locally connected layer.

Different Convolutions

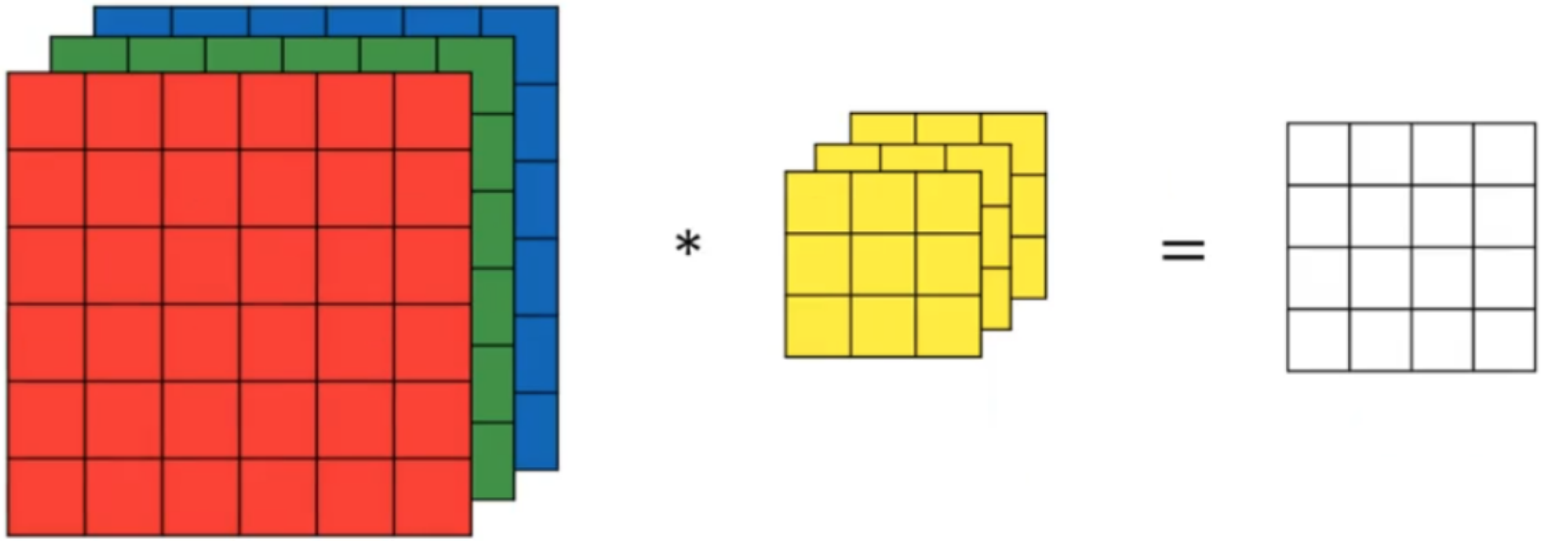
Valid Convolution: all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular. However, the size of the output shrinks at each layer.

Same Convolution: the network can contain as many convolutional layers as the available hardware can support. However, the input pixels near the border influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model.

Full Convolution: the output pixels near the border are a function of fewer pixels than the output pixels near the center. This can make it difficult to learn a single kernel that performs well at all positions in the convolutional feature map.

Convolution Over Volume

Convolution over volume refers to the process of applying a kernel to a multi-channel input, such as a color image or a multi-channel feature map from a previous layer. This operation extends the idea of convolution from 2D (height and width) to 3D (height, width, and depth).



Source: Andrew Ng-Machine Learning Specialization

Detector Layer

Since convolution is a linear operation and images are far from linear, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity, and we call it **detector layer**.

There are several types of non-linear operations, the popular ones being:

- Sigmoid
- Tanh
- ReLU

Pooling Layer

Remember: Pooling layers do not require learnable parameters.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. Common pooling functions: **max pooling** and **average pooling**.

Pooling helps to make the representation become approximately invariant to small translations of the input. *Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.*

Pooling Layer

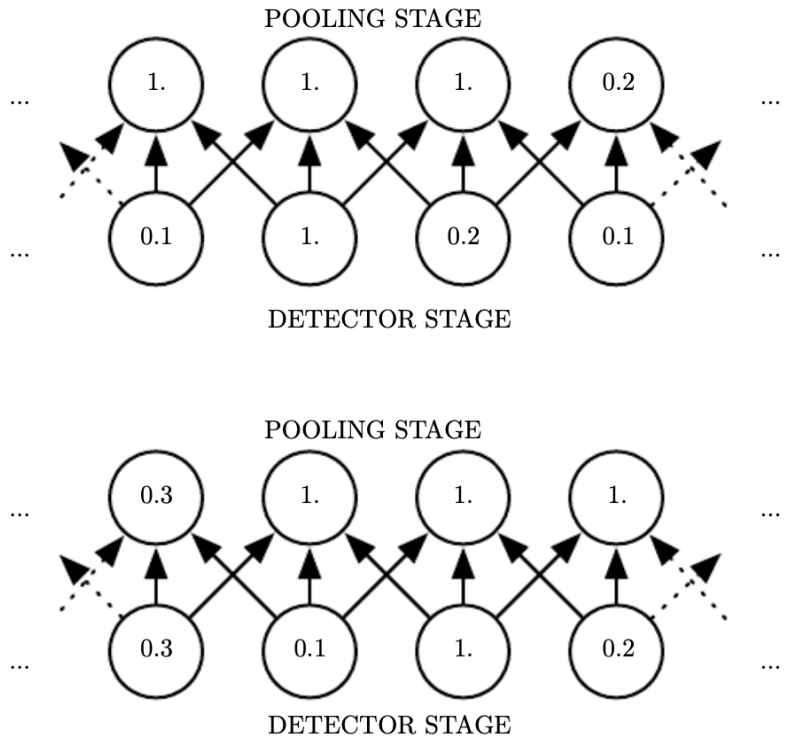


Figure 9.8: Max pooling introduces invariance. *(Top)*A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. *(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 only sensitive to the maximum value in the neighborhood, not its exact location.

Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Pooling Layer

Because pooling summarizes the responses over a whole neighborhood, we can use the pooling layer to reduce the input size of the next layer by setting the **stride** larger than 1, and this can improve the computational efficiency.

For many tasks, pooling is essential for handling inputs of varying size.

For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of the stride between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size.

CNN Architecture Examples

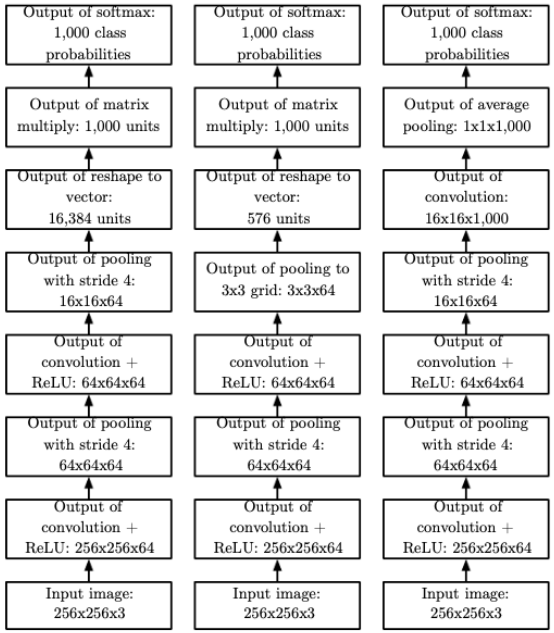


Figure 9.11: Examples of architectures for classification with convolutional networks. The specific strides and depths used in this figure are not advisable for real use; they are designed to be very shallow in order to fit onto the page. Real convolutional networks also often involve significant amounts of branching, unlike the chain structures used here for simplicity. *(Left)*A convolutional network that processes a fixed image size. After alternating between convolution and pooling for a few layers, the tensor for the convolutional feature map is reshaped to flatten out the spatial dimensions. The rest of the network is an ordinary feedforward network classifier, as described in chapter 6. *(Center)*A convolutional network that processes a variable-sized image, but still maintains a fully connected section. This network uses a pooling operation with variably-sized pools but a fixed number of pools, in order to provide a fixed-size vector of 576 units to the fully connected portion of the network. *(Right)*A convolutional network that does not have any fully connected weight layer. Instead, the last convolutional layer outputs one feature map per class. The model presumably learns a map of how likely each class is to occur at each spatial location. Averaging a feature map down to a single value provides the argument to the softmax classifier at the top.

Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Prior Assumptions in Convolution and Pooling

We can imagine a convolutional net as being similar to a fully connected net, but with an infinitely strong prior over its weights.

Prior assumptions in the convolution layer:

- the weights for one hidden unit must be identical to the weights of its neighbor, but shifted in space
- the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit

Prior assumption in the pooling layer:

- each unit should be invariant to small translations

Takeaway: convolution and pooling can cause **underfitting**. If a task relies on preserving precise spatial information, then using pooling on all features can increase the training error.

CNN in PyTorch

User Guide: https://pytorch.org/docs/stable/nn.html

```
import torch

torch.nn.Conv2d(in_channels, out_channels, kernel_size,
stride=1, padding=0,  padding_mode='zeros')
torch.nn.Tanh()
torch.nn.MaxPool2d(kernel_size)
```

Let's code out a CNN model using PyTorch!

References

- Gu, S., B. Kelly, and D. Xiu. (2020). "Empirical asset pricing via machine learning." The Review of Financial Studies. 33(5): 2223–2273.
- Choi, D., W. Jiang, and C. Zhang. (2022). "Alpha Go Everywhere: Machine Learning and International Stock Returns." Tech. rep. The Chinese University of Hong Kong.
- Bali, T. G., A. Goyal, D. Huang, F. Jiang, and Q. Wen. (2020). "Predicting Corporate Bond Returns: Merton Meets Machine Learning." Tech. rep. Georgetown University.
- Gentzkow, Matthew, Bryan Kelly, and Matt Taddy. "Text as data." Journal of Economic Literature 57.3 (2019): 535-574.
- Ash, Elliott, and Stephen Hansen. "Text algorithms in economics." Annual Review of Economics 15 (2023): 659-688.

References

- Jegadeesh, N. and D. Wu. (2013). "Word power: A new approach for content analysis." Journal of Financial Economics. 110(3): 712–729.
- Ke, Zheng Tracy, Bryan T. Kelly, and Dacheng Xiu. "Predicting returns with text data." No. w26186. National Bureau of Economic Research, 2019.
- Garcia, Diego, Xiaowen Hu, and Maximilian Rohrer. "The colour of finance words." Journal of Financial Economics 147.3 (2023): 525-549.
- Chen, Yifei, Bryan T. Kelly, and Dacheng Xiu. "Expected returns and large language models." Available at SSRN 4416687 (2022).
- Jiang, Jingwen, Bryan Kelly, and Dacheng Xiu. "(Re-) Imag (in) ing Price Trends." The Journal of Finance 78.6 (2023): 3193-3249.