

# Introduction to neural networks

Alper Celik

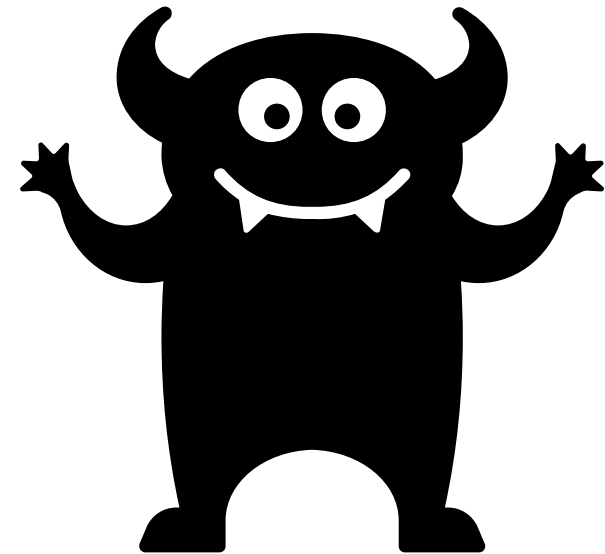
2026-01-21

# Outline

- What is a neural network
- Network architectures
  - Neural Nets, full connected MLP
  - Convolution layers
  - Recurrent layers
  - Transformers
  - Graph neural networks
- Getting started with pre-trained models
  - Huggingface, timm, diffusers
  - Quantization
  - Parameter efficient fine tuning (PEFT)
  - Low rank adaptation (LORA)

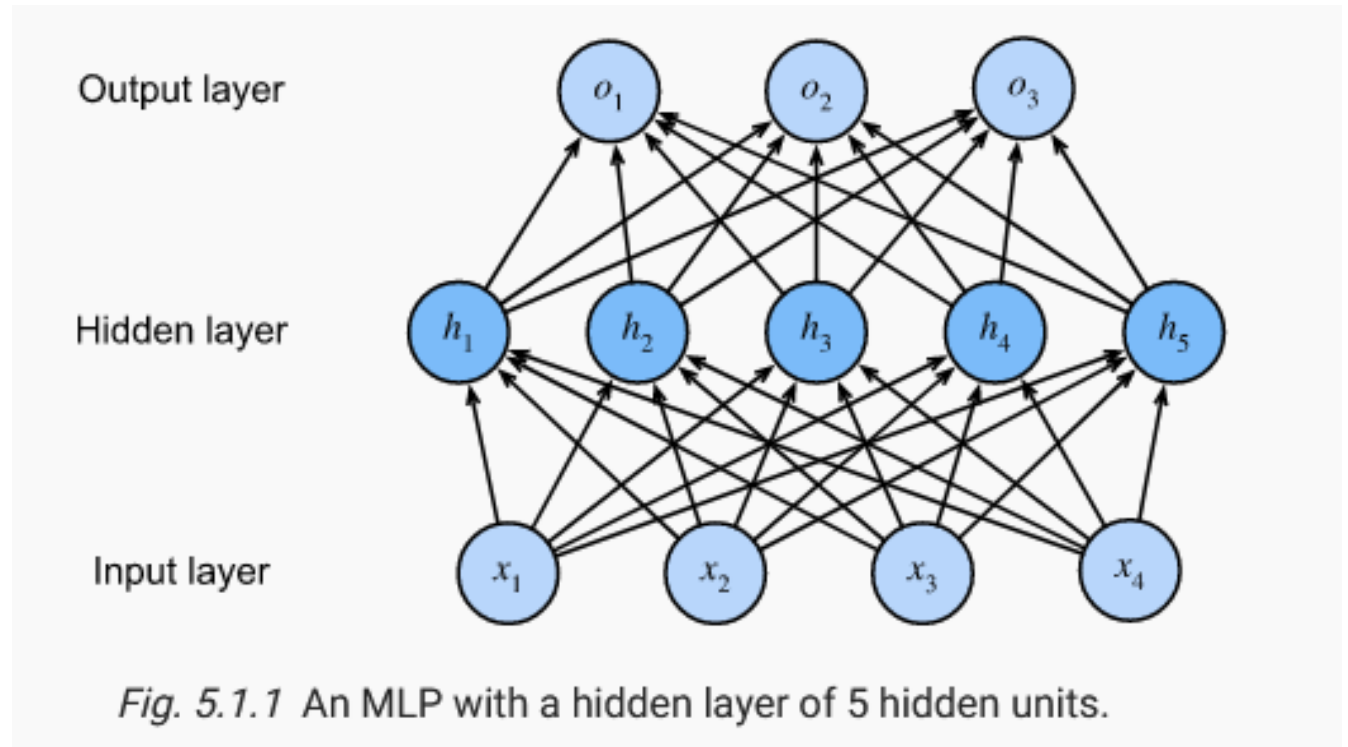
# Warning!

- We will explore some advanced topics and math
- We will be looking at and analyzing python code
- We will NOT be training any networks but will look at how the training code works
- Due to their complexity there are a few things we will not be discussing today:
  - Diffusion models
  - Tensor field networks,  $SO(3)$ ,  $SE(3)$  equivariance
  - Flow matching
- Due to time constraints we will not be covering:
  - RAG (retrieval augmented generation)
  - MCP (model context protocols)
  - Agents



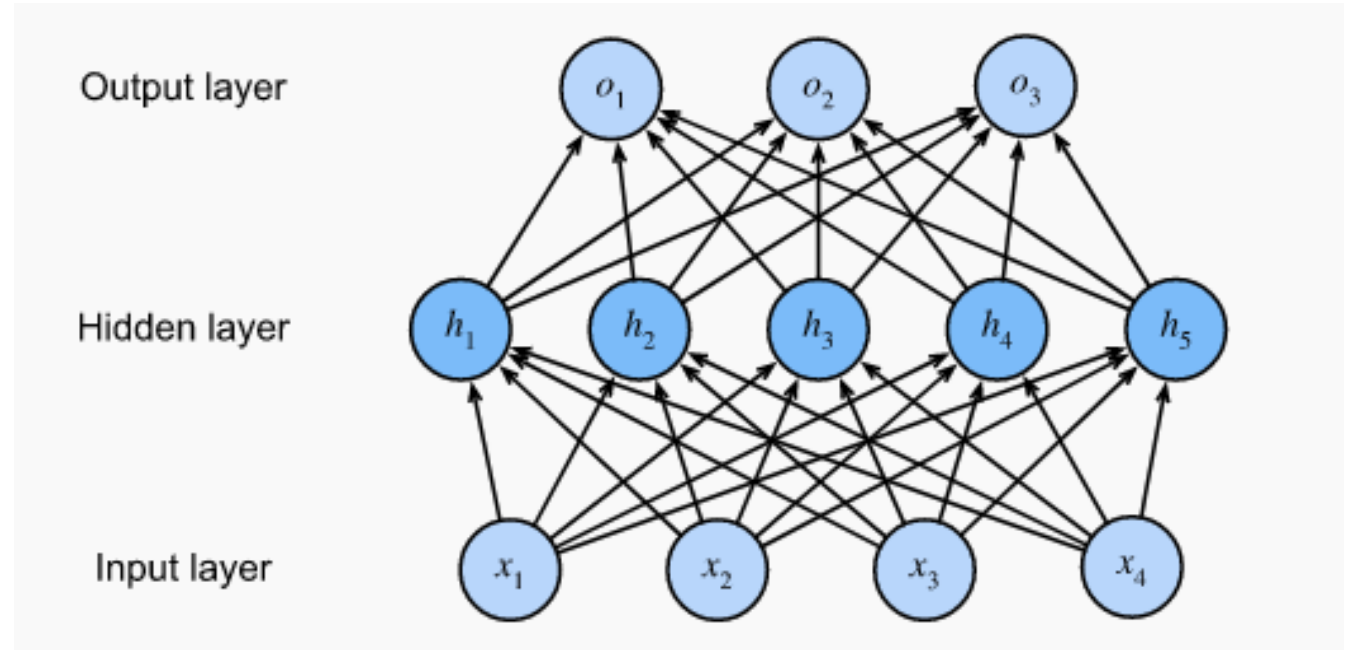
# What is a neural network?

- A neural network is composed of one or more layers.
- Each layer contains an arbitrary number of neurons
- These neurons are connected to one another with an associated weight
- Each layer can also have additional connections to other layers to represent the data more effectively
- They are "universal approximators", an arbitrarily large neural network can represent any distribution(s) to an arbitrary accuracy.



# Fully connected MLP (multi layer perceptron)

- Most common architecture
- Consists of arbitrary number of layers with arbitrary number of nodes
- Can be used for many tasks on tabular data as is
- Can be used to convert outputs of other layers into an arbitrary shape
- Can be used to increase or decrease data dimensionality
- They have recently been implicated in storing "information" as high dimensional almost perpendicular vectors



$$\mathbf{H} = \sigma(\mathbf{XW}^{(1)} + \mathbf{b}^{(1)}),$$
$$\mathbf{O} = \mathbf{HW}^{(2)} + \mathbf{b}^{(2)}.$$

vs

$$\mathbf{H} = \mathbf{XW}^{(1)} + \mathbf{b}^{(1)},$$
$$\mathbf{O} = \mathbf{HW}^{(2)} + \mathbf{b}^{(2)}.$$

# Fully connected MLP (multi layer perceptron)

```
# an example

import torch
import torch.nn as nn
import torch.nn.functional as F

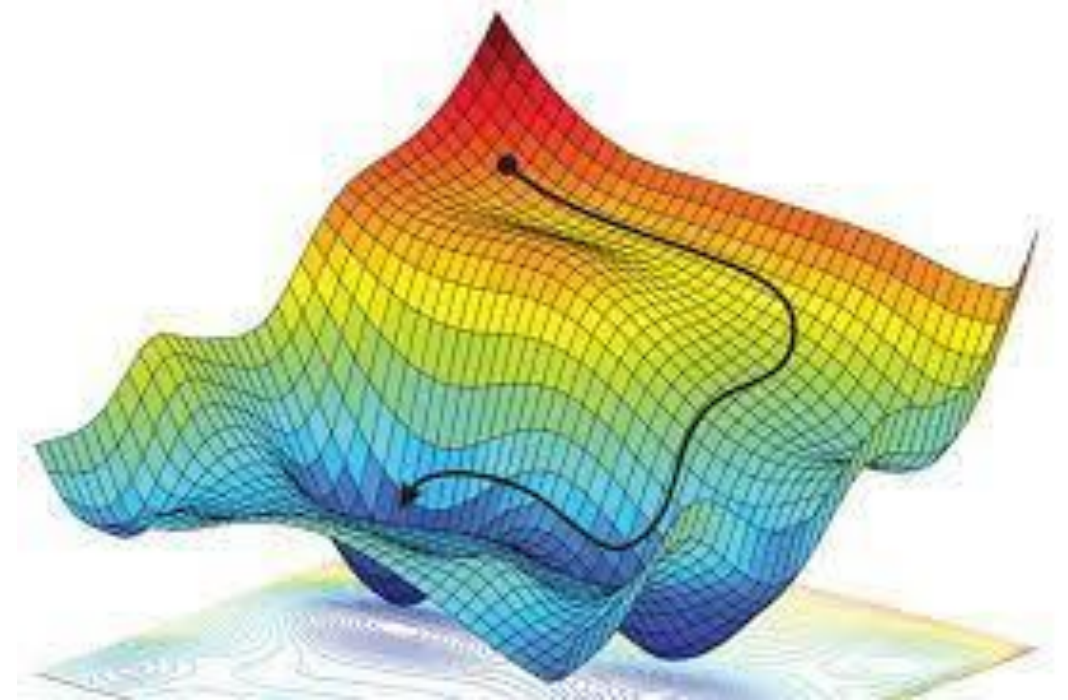
class SimpleMLP(nn.Module):
    def __init__(self, input_dim, hidden_dims=[128, 64], output_dim=10, dropout=0.1):
        super().__init__()
        layers = []
        dims = [input_dim] + hidden_dims
        for i in range(len(dims) - 1):
            layers.append(nn.Linear(dims[i], dims[i+1]))
            layers.append(nn.ReLU())
            if dropout > 0:
                layers.append(nn.Dropout(dropout))
        layers.append(nn.Linear(dims[-1], output_dim))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)

model = SimpleMLP(input_dim=20, hidden_dims=[64, 32], output_dim=5) #differentiate between 5 classes
```

# Model Training: Optimizers

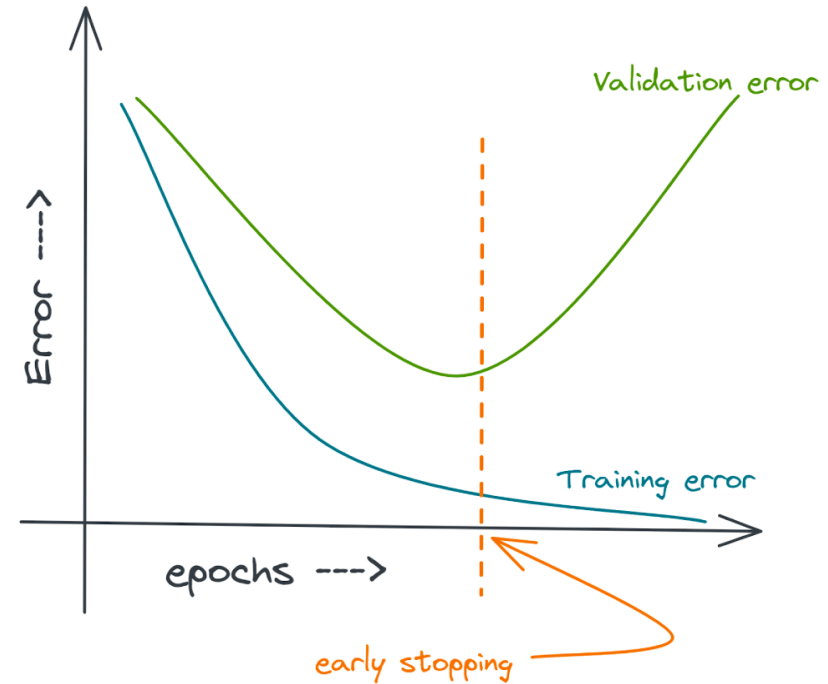
- Optimizers determine how you calculate the gradient for the backpropagation
- For our purposes we assume that the function we are trying to find is convex
- Our goal is the minimize the loss function
- This indirectly means that we found a good estimator function between inputs and outcomes
- There are many different optimizers some of the more popular ones:
  - Adam, AdamW
  - RMSprop
  - SGD
- In pytorch you can also specify learning rate for most of these optimizers



# Model Training: Regularization

- Neural networks are extremely flexible approximators
- They can "memorize" your data and are prone to overfitting
- There are couple of mechanisms we can minimize this
  - Early stopping
  - Dropout
  - Changing the learning rate
  - L1 and L2 regularization (weight decay)
  - Task specific data augmentation

$\eta(t) = \eta_i \text{ if } t_i \leq t \leq t_{i+1}$	piecewise constant
$\eta(t) = \eta_0 \cdot e^{-\lambda t}$	exponential decay
$\eta(t) = \eta_0 \cdot (\beta t + 1)^{-\alpha}$	polynomial decay



L1 Regularization

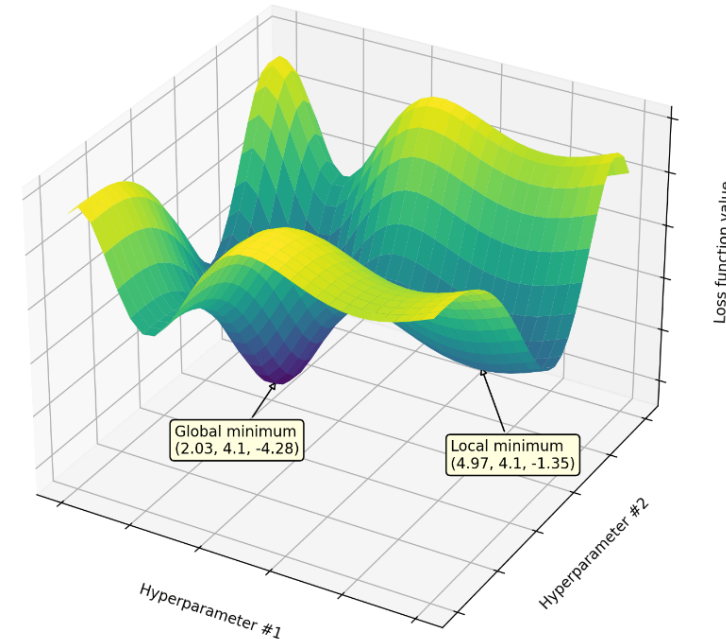
$$\text{Cost} = \sum_{i=0}^N (y_i - \sum_{j=0}^M x_{ij} W_j)^2 + \lambda \sum_{j=0}^M |W_j|$$

L2 Regularization

$$\text{Cost} = \underbrace{\sum_{i=0}^N (y_i - \sum_{j=0}^M x_{ij} W_j)^2}_{\text{Loss function}} + \underbrace{\lambda \sum_{j=0}^M W_j^2}_{\text{Regularization Term}}$$

# Model Training: Hyperparameter tuning

- Parameters refer to learnable features of the model (weights and biases)
- Hyperparameters are inherent properties of the model (number of layers, learning rate, optimization functions, activation functions, number of neurons in layers).
- Some of these can have big impact on model performance
- We cannot possibly sample even every "reasonable" parameter in this space
- Every time you change a hyperparameter you will need to re-train
- Simplest version is grid search
- You can also randomly sample
- There are other algorithms to speed up the process



# Fully connected MLP (multi layer perceptron) training

```
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss() #this is common for multi-class classif
```

```
for step in range(100):
    x = torch.randn(32, 20) #just random data
    y = torch.randint(0, 5, (32,))

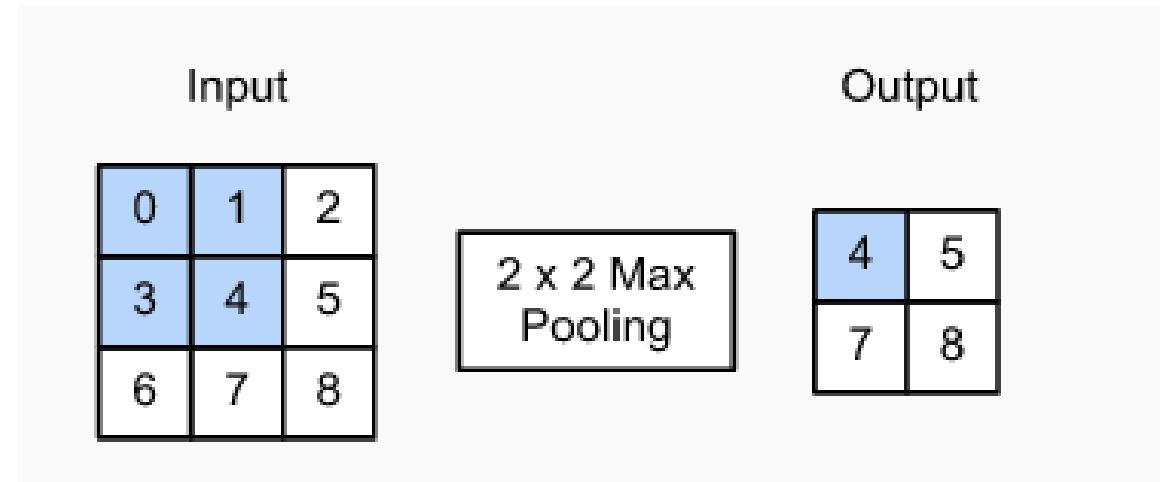
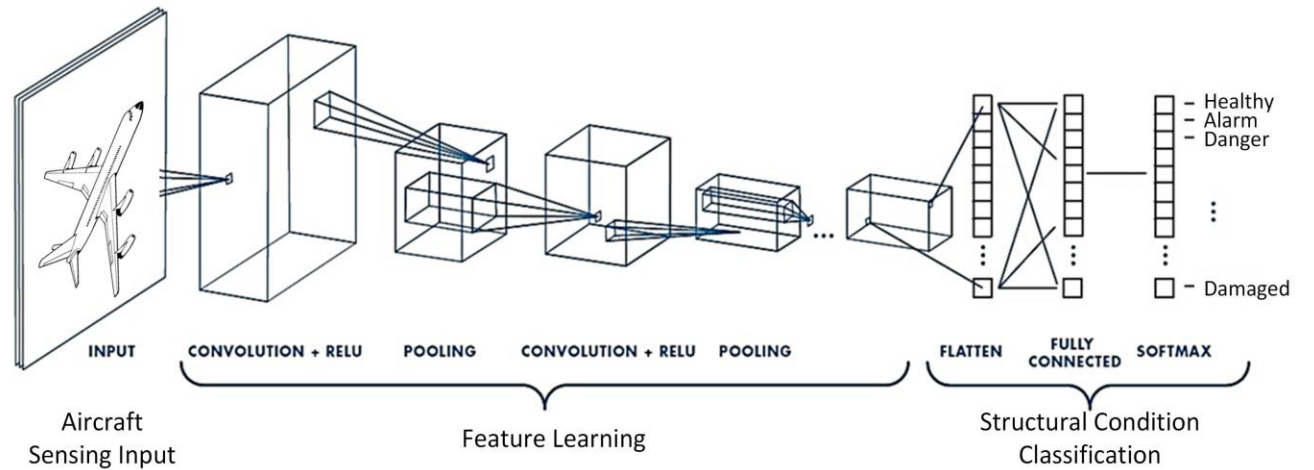
    logits = model(x)
    loss = criterion(logits, y)

    opt.zero_grad()
    loss.backward()
    opt.step()

    if step % 10 == 0:
        print(f"Step {step}: loss={loss.item():.4f}")
```

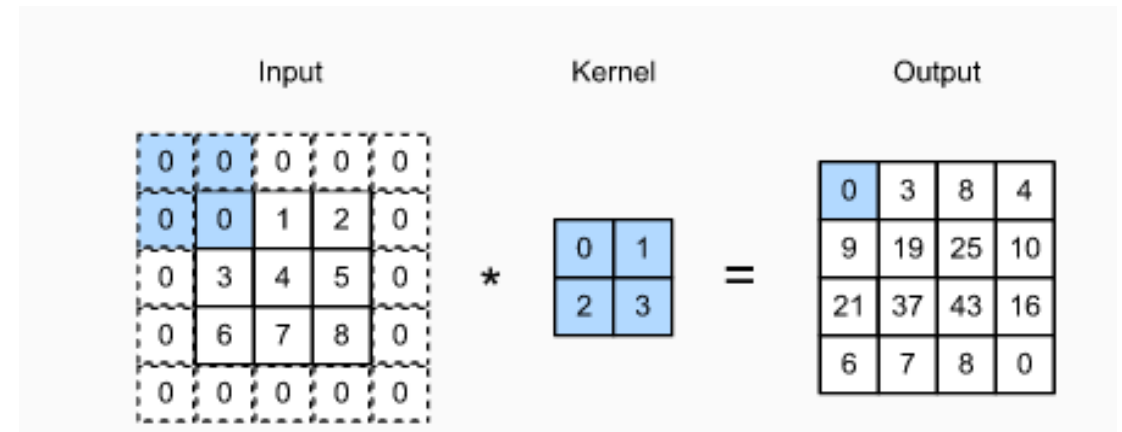
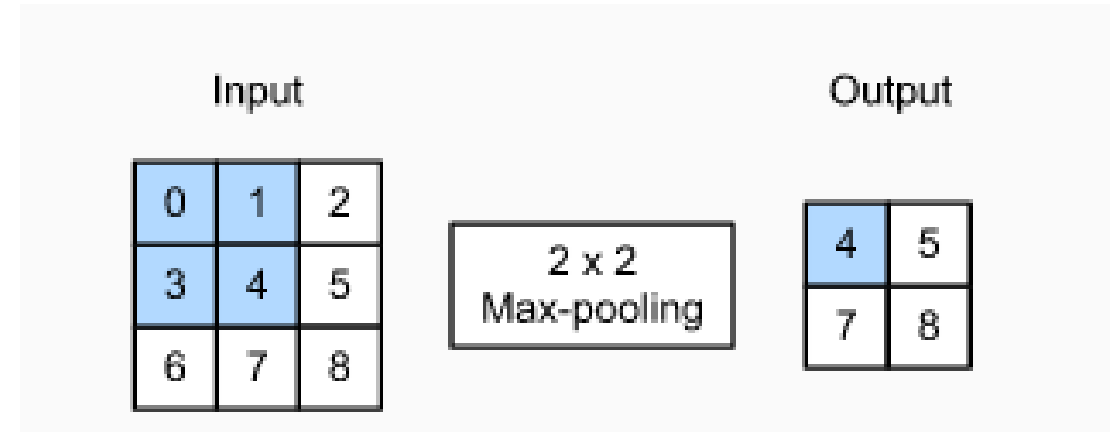
# Convolutional networks

- Good for data with local structure (i.e. images)
- Used to decrease the number of dimensions in one axis and project them onto another to learn "features"
- Several of them can be used back-to-back to learn features of features
- After each convolution layer there is usually a pooling layer to decrease the dimensionality of the convolution layer
- CNNs often are followed by a small MLP to convert the CNN outputs to either a regression or classification suitable format



# Convolution layers

- These layers implement something called invariance.
- We want to be able to detect things in the image regardless of its size and shape
- There are couple of things we would like to specify in a convolution layer
  - Padding: What to do when you reach the edge of the matrix
  - Stride: How many elements to jump at each step
  - Pooling method: how to convert each stride step into a single number (usually max or average pooling)



# Convolution networks implementation

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 32 * 32, 128)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # -> [B,16,64,64]
        x = self.pool(F.relu(self.conv2(x))) # -> [B,32,32,32]
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleCNN(num_classes).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()
```

```
for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    for x, y in train_loader:
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        out = model(x)
        loss = criterion(out, y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}: Train loss = {total_loss/len(train_loader):.4f}")
```

# Data augmentation

```
#let's do some data augmentation
import os

from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.transforms import InterpolationMode

train_transforms = transforms.Compose([
    transforms.RandomResizedCrop(128, scale=(0.8, 1.0)), # random crop and resize
    transforms.RandomHorizontalFlip(p=0.5),             # mirror image
    transforms.RandomRotation(15, interpolation=InterpolationMode.BILINEAR),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),                             # convert to tensor [0,1]
    transforms.Normalize((0.5,), (0.5,))               # normalize
])

# you probably don't want to go too crazy on your validation set and you don't want to
val_transforms = transforms.Compose([
    transforms.Resize(128),
    transforms.CenterCrop(128),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.ImageFolder(root=train_dir, transform=train_transforms)
val_dataset = datasets.ImageFolder(root=val_dir, transform=val_transforms)
```

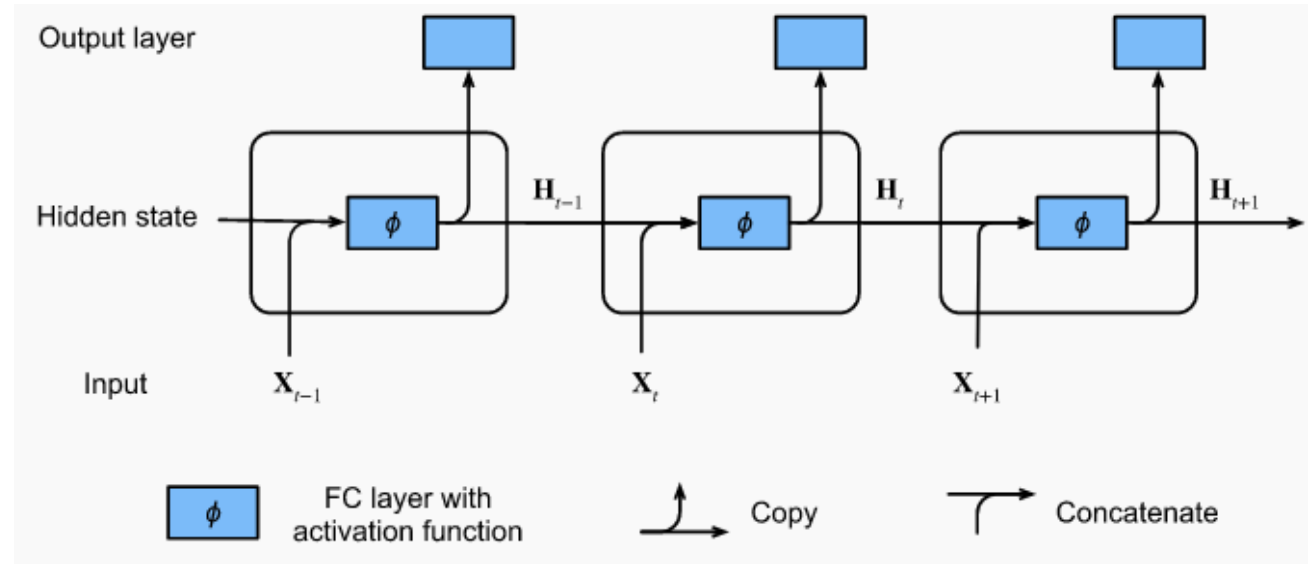
- When it comes to data, more is more
- Sometimes you can make more data out of thin air
- We can change the ratios of our images, change colors, flip and do all sorts of things and keep the label the same to generalize better.
- Torchvision has built in functions and classes that does this for you easily
- Go crazy with your training set
- Do a little bit with your validation set
- DO NOT TOUCH YOUR TEST SET!!!

# Recurrent layers

- Initially developed to deal with sequence data
- At each time point  $t$ , the hidden state of  $H_t$  is calculated by concatenating input at  $t$  to hidden state at  $t-1$  and feeding that to the activation function.
- This means only the abstract representations of  $t-1$  is passed to  $t$ , for longer sequences this creates a problem as the representation may not capture all the relevant details or the details may not be there.
- Compare this architecture to a markov chain or arbitrary order

First order markov chain

$$P(x_1, \dots, x_T) = P(x_1) \prod_{t=2}^T P(x_t | x_{t-1}).$$



Recurrent network with hidden states

$$P(x_t | x_{t-1}, \dots, x_1) \approx P(x_t | h_{t-1}),$$

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h).$$

# Recurrent network implementation

```
class ManualStackedRNN(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, hidden_dims=[128, 256, 256]):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)

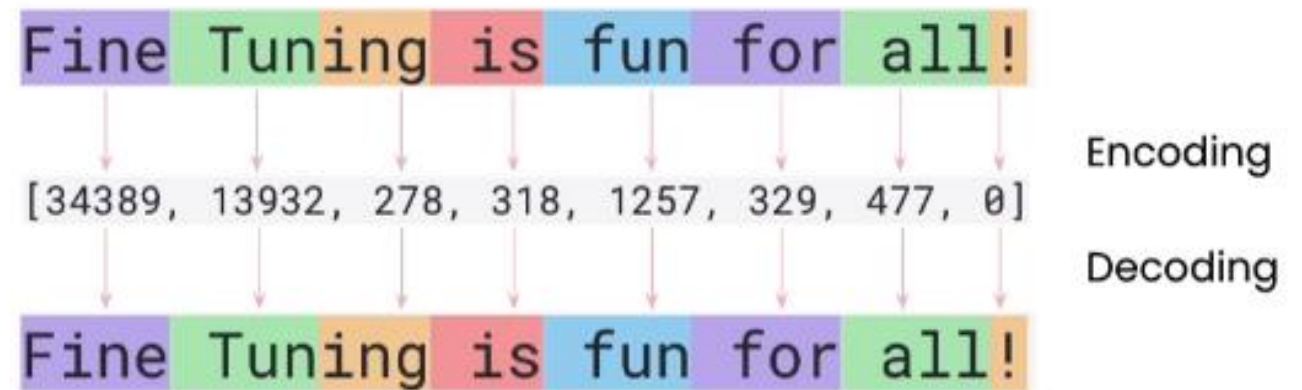
        self.rnnns = nn.ModuleList()
        input_dim = embed_dim
        for hidden_dim in hidden_dims:
            self.rnnns.append(nn.RNN(input_dim, hidden_dim, batch_first=True)) #here you can define the logic, like you want
            input_dim = hidden_dim

        # Optional nonlinearities between RNN layers
        self.activ = nn.ReLU()
        self.fc = nn.Linear(hidden_dims[-1], vocab_size)

    def forward(self, x):
        x = self.embed(x)
        for rnn in self.rnnns:
            x, _ = rnn(x)
        x = self.activ(x)
        logits = self.fc(x)
        return logits
```

# But wait! NNs are about numbers not words!

- Neural networks do not work with text or images, they work with numbers, so we need a way to convert text to numbers, this is called tokenization
- Each word (or sub-word) is converted to an integer from a "dictionary"
- These dictionaries can be language specific or multi-language
- Smaller models benefit from smaller dictionaries (single language)
- They are not created by hand but with automation scripts
- There are special tokens to indicate missing, blank, unknown data as well as start and end of a text
- You can build your own but why reinvent the wheel?



# How it all started

- Large language models are almost always based on transformer architecture
- They consider the entire text as a whole and therefore are aware of the "context"
- The same architecture can be applied to images and videos as well
- They are made out of "Transformer blocks"

---

## Attention Is All You Need

---

**Ashish Vaswani\***  
Google Brain  
avaswani@google.com

**Noam Shazeer\***  
Google Brain  
noam@google.com

**Niki Parmar\***  
Google Research  
nikip@google.com

**Jakob Uszkoreit\***  
Google Research  
usz@google.com

**Llion Jones\***  
Google Research  
llion@google.com

**Aidan N. Gomez\* †**  
University of Toronto  
aidan@cs.toronto.edu

**Lukasz Kaiser\***  
Google Brain  
lukaszkaizer@google.com

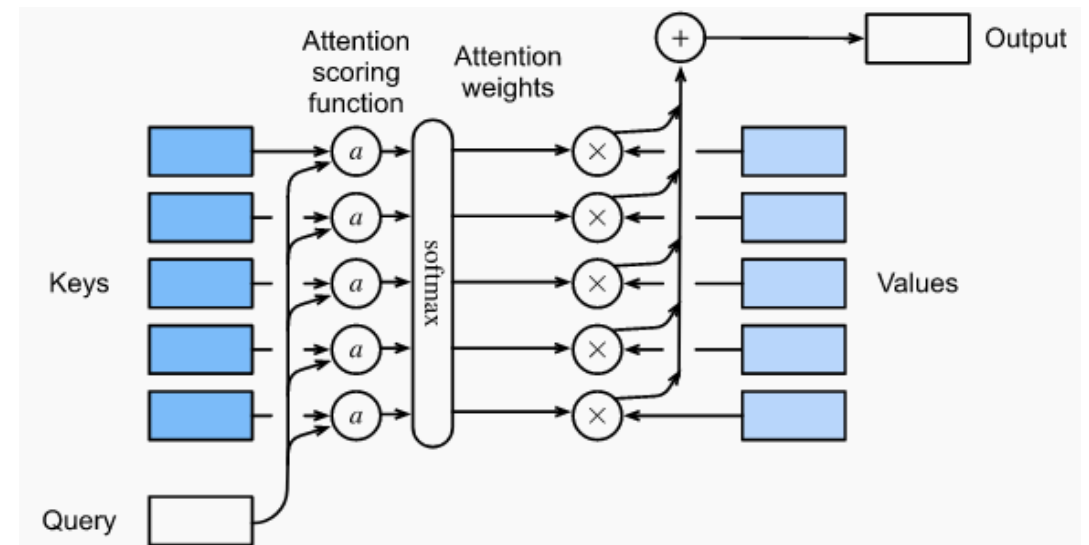
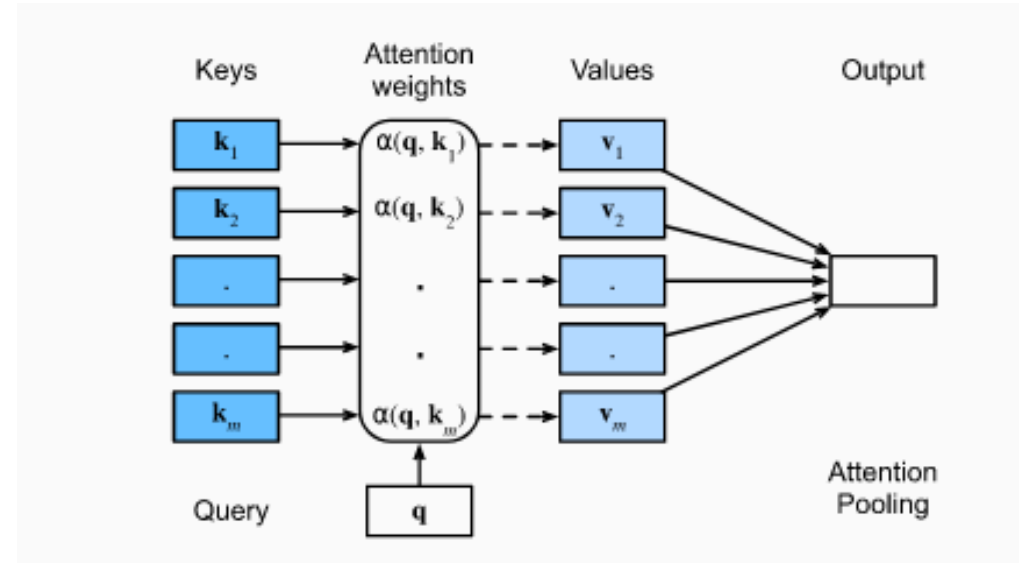
**Illia Polosukhin\* ‡**  
illia.polosukhin@gmail.com

# Attention Mechanisms: Attention

- RNNs calculated hidden states one at a time, this was computationally expensive and resulted in model "forgetting the details" as time went on.
- Attention mechanisms work on query, key, value pairs.
- For a given key (k) and value (v) collection D and a query (q) we can calculate the Attention for a specific query.
- We are searching for the most relevant key value pairs given a scalar function  $\alpha$  that takes in a key value pair.
- We then return the pooled attention (as a tensor) for a query, this means we get a value for all k,v pairs for a q.

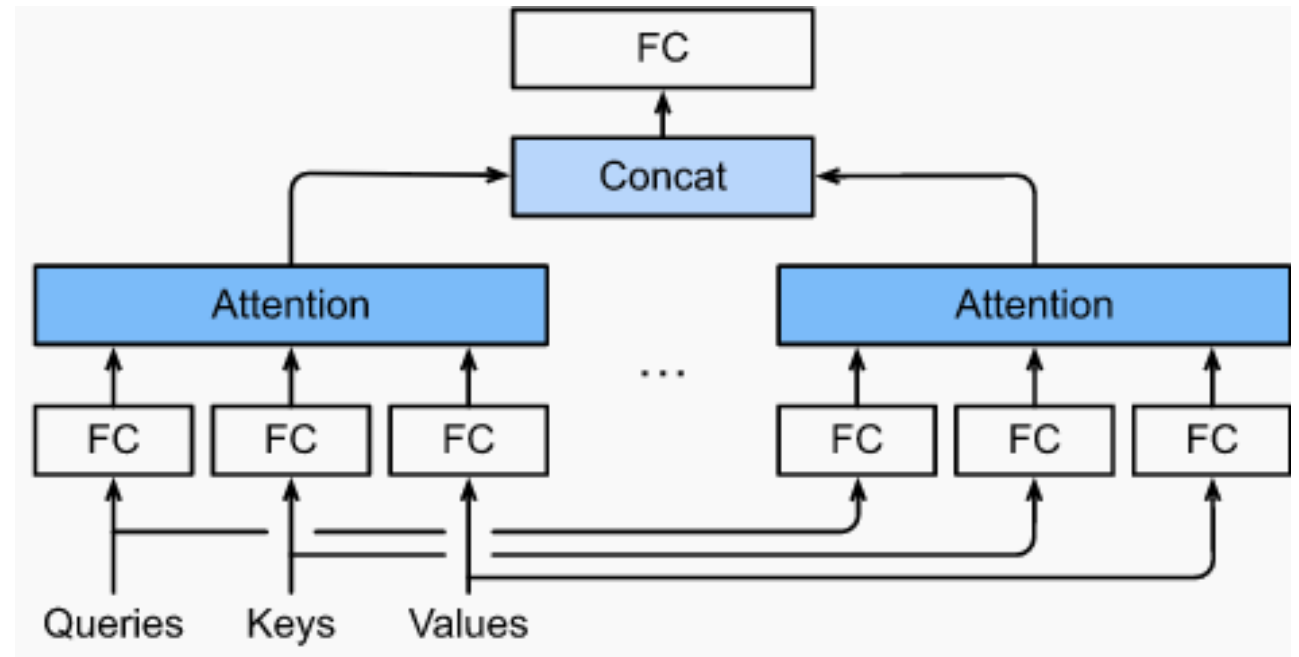
$$\text{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^m \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i,$$

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})}{\sum_{j=1} \exp(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d})}.$$



# Attention Mechanisms: Multi-Head Attention

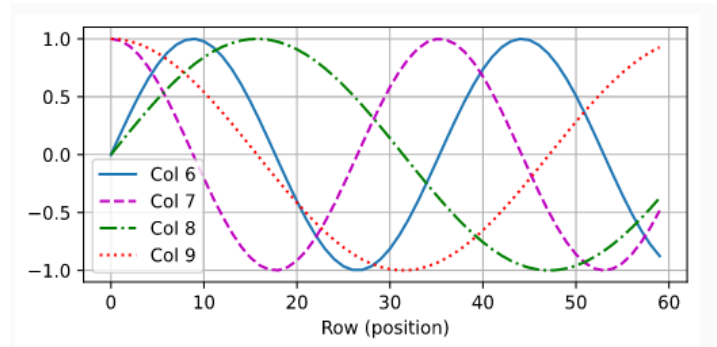
- We may want our model to pay attention to different section of a sequence such as capturing dependencies of various ranges
- We can accomplish that by allowing our model to jointly use different sections of the same representation.
- So instead of single attention pooling we can use many independently learned representations.
- These representations are built as fully connected layers.
- When we are done we can concatenate them and feed them to another layer to reshape it.



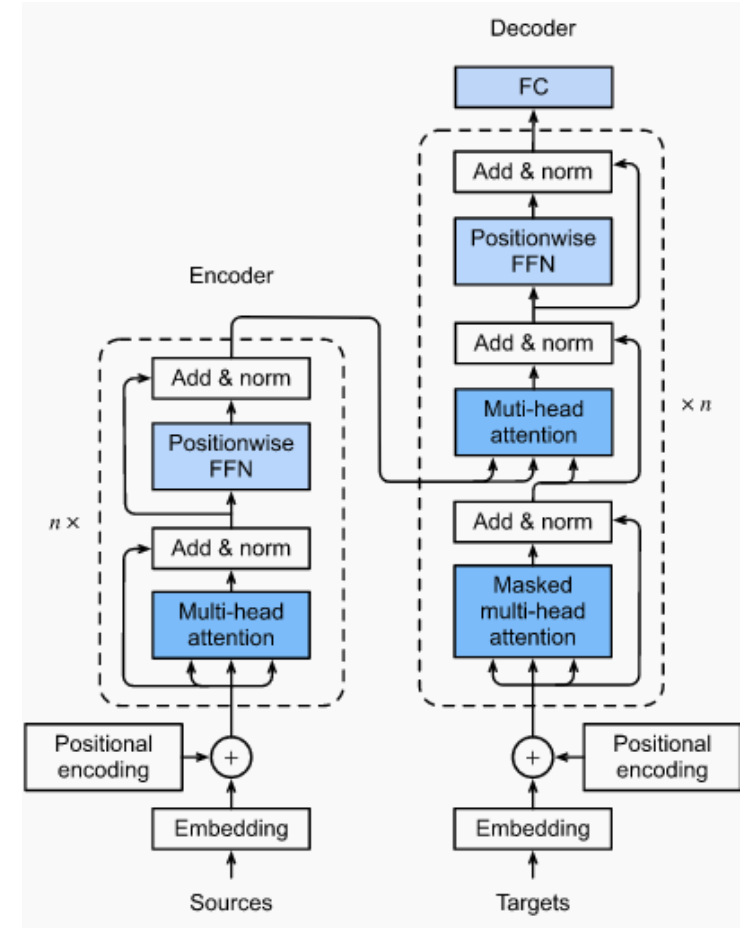
$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)} \mathbf{q}, \mathbf{W}_i^{(k)} \mathbf{k}, \mathbf{W}_i^{(v)} \mathbf{v}) \in \mathbb{R}^{p_v},$$

# Attention Mechanisms: Transformer

- Processing all the input *all at once* destroys the sequence information so we need to add it back.
- Positional encoding is another vector of same length that the required information.
- It needs to be normalized so that it does not pay too much attention to the beginning or the end.
- We take embeddings, add position information and pass through attention layer, we normalize combine different attentions (multihead), pass it to a simple MLP to transform, add residual connections, repeat, as necessary.



$$\vec{p}_t = \begin{bmatrix} \sin(\omega_1 \cdot t) \\ \cos(\omega_1 \cdot t) \\ \sin(\omega_2 \cdot t) \\ \cos(\omega_2 \cdot t) \\ \vdots \\ \sin(\omega_{d/2} \cdot t) \\ \cos(\omega_{d/2} \cdot t) \end{bmatrix}_{d \times 1}$$



# Decoder Layer

```
class DecoderOnly(nn.Module):
    def __init__(self, vocab_size, d_model=128, n_heads=4, n_layers=2, dim_feedforward=512, dropout=0.1, max_len=128):
        super().__init__()
        self.token_emb = nn.Embedding(vocab_size, d_model)
        self.pos_emb = nn.Embedding(max_len, d_model)


        decoder_layer = nn.TransformerDecoderLayer( #define what a layer looks like
            d_model=d_model,
            nhead=n_heads,
            dim_feedforward=dim_feedforward,
            dropout=dropout,
            batch_first=True
        )

        self.decoder = nn.TransformerDecoder(decoder_layer, num_layers=n_layers) #implement it N times for extra abstr
        self.lm_head = nn.Linear(d_model, vocab_size, bias=False)

        self.register_buffer("mask", torch.tril(torch.ones(max_len, max_len)).bool()) #see below

    def forward(self, x):
        B, T = x.shape
        pos = torch.arange(T, device=x.device)
        x = self.token_emb(x) + self.pos_emb(pos)

        # Causal mask to prevent attending to future tokens
        causal_mask = self.mask[:T, :T]
        out = self.decoder(x, x, tgt_mask=~causal_mask)
        logits = self.lm_head(out)
        return logits
```



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

# Encoder Layer

```
class SimpleTransformerEncoder(nn.Module):
    def __init__(self, vocab_size, d_model=128, nhead=4, num_layers=2, dim_feedforward=256, max_len=100, num_classes=1000):
        super().__init__()

        # Embedding layers
        self.token_embed = nn.Embedding(vocab_size, d_model) #same as above we'll look at that in a min
        self.pos_embed = nn.Embedding(max_len, d_model)

        # Stack of encoder layers
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=dim_feedforward,
            dropout=dropout,
            activation="relu", #because we are learning and storing information, we need some sort of way to add no-1
            batch_first=True, # Important for (batch, seq, feature) ordering
        )

        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)

        # Optional output head (for classification or embeddings)
        self.fc = nn.Linear(d_model, num_classes)

    def forward(self, x):
        """
        x: LongTensor of shape (batch, seq_len)
        """
        batch_size, seq_len = x.size()
        device = x.device

        # Create positional indices (0, 1, 2, ..., seq_len-1)
        pos = torch.arange(seq_len, device=device).unsqueeze(0).expand(batch_size, seq_len)

        # Add token + positional embeddings
        x = self.token_embed(x) + self.pos_embed(pos)

        # Pass through the transformer encoder
        encoded = self.encoder(x)

        # Pooling: take mean across sequence dimension
        pooled = encoded.mean(dim=1)

        # Output
        out = self.fc(pooled)
        return out
```

```

class SimpleTransformer(nn.Module):
    def __init__(self, vocab_size, embed_dim=64, num_heads=4, ff_dim=128, num_layers=2, max_len=128):
        super().__init__()
        # encoder and decoder embeddings
        self.src_emb = nn.Embedding(vocab_size, embed_dim)
        self.tgt_emb = nn.Embedding(vocab_size, embed_dim)
        self.pos_emb = nn.Embedding(max_len, embed_dim)

        # stacks
        self.encoder_layers = nn.ModuleList([
            EncoderBlock(embed_dim, num_heads, ff_dim) for _ in range(num_layers)
        ])
        self.decoder_layers = nn.ModuleList([
            DecoderBlock(embed_dim, num_heads, ff_dim) for _ in range(num_layers)
        ])

        self.output_proj = nn.Linear(embed_dim, vocab_size)

    def forward(self, src, tgt):
        B, S = src.shape
        _, T = tgt.shape
        src_pos = torch.arange(0, S, device=src.device).unsqueeze(0)
        tgt_pos = torch.arange(0, T, device=tgt.device).unsqueeze(0)

        src_emb = self.src_emb(src) + self.pos_emb(src_pos)
        tgt_emb = self.tgt_emb(tgt) + self.pos_emb(tgt_pos)

        # Encoder
        for layer in self.encoder_layers:
            src_emb = layer(src_emb)

        # Causal mask to prevent attending to future tokens
        causal_mask = torch.triu(torch.ones(T, T, device=tgt.device), diagonal=1).bool()

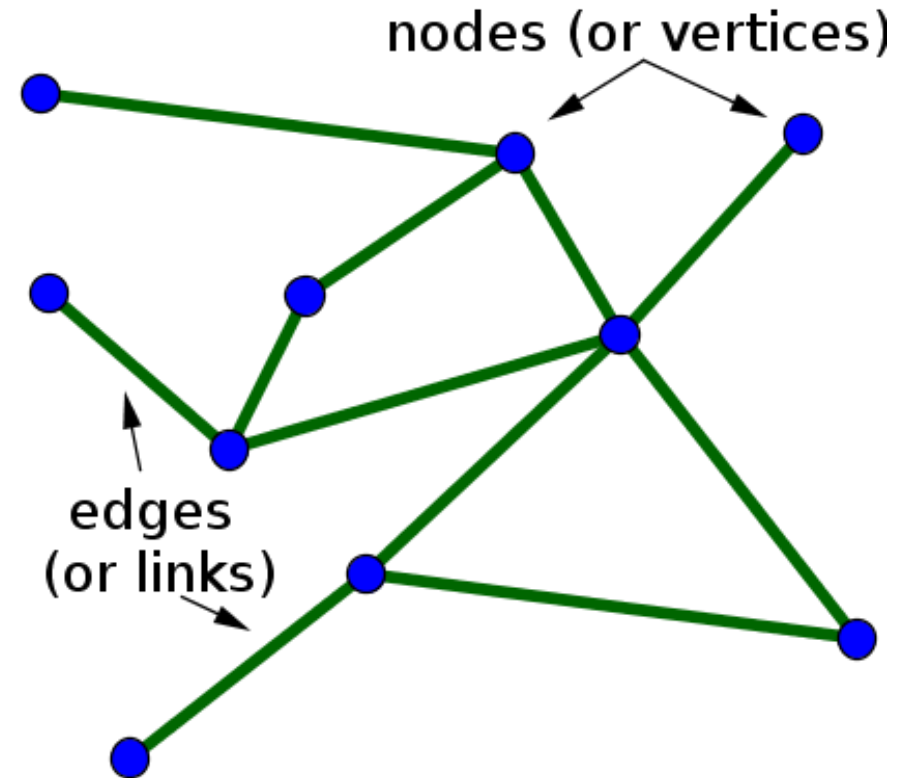
        # Decoder
        for layer in self.decoder_layers:
            tgt_emb = layer(tgt_emb, src_emb, self_mask=causal_mask)

        # Final projection
        logits = self.output_proj(tgt_emb)
        return logits

```

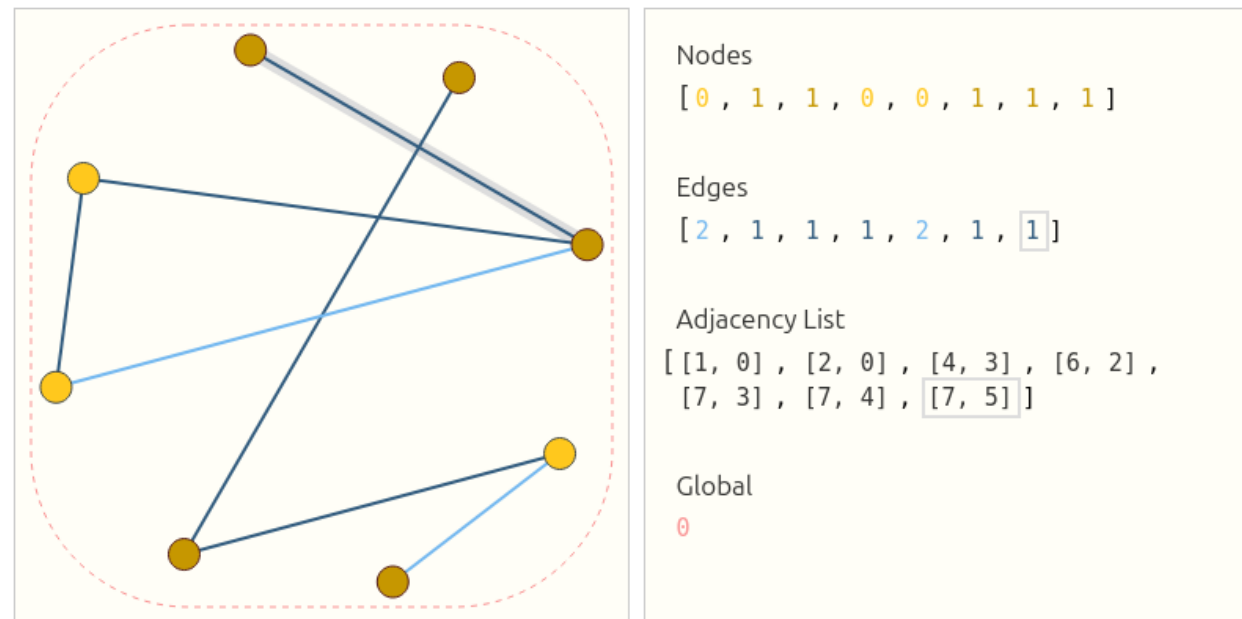
# Graph neural networks (GNNs), investigating connections

- Unlike all the other networks we mentioned GNNs are concerned with connections between things
  - Connections between people (social networks),
  - Things (product recommendations)
  - Atoms (proteins, small molecules)
- These connections are represented as nodes (things) and edges (connections)
- There are 3 main tasks GNNs excel at
  - Node prediction
  - Edge prediction
  - Graph classification/regression



# Graph neural networks (GNNs), investigating connections

- Instead of representing things as matrices they are represented as connections
- Popular packages are:
  - **Pytorch-geometric**
  - **Deep graph library(dgl)**
  - Spektral
  - Graph Nets
- These are becoming more popular in computational chemistry and protein models because they can represent the connections between atoms/amino acids/molecules



# GNN implementation, the dataset:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.data import Data, Dataset, DataLoader
from torch_geometric.nn import GCNConv, global_mean_pool
from rdkit import Chem
from rdkit.Chem import AllChem
import pandas as pd

def smiles_to_graph(smiles, label):
    mol = Chem.MolFromSmiles(smiles)
    if mol is None:
        return None

    # Node features: one-hot atom type (C,H,O,N,F, etc.)
    atom_types = ['H', 'C', 'N', 'O', 'F', 'P', 'S', 'Cl', 'Br', 'I'] # basic set
    x = []
    for atom in mol.GetAtoms(): #RDKit specific functions
        one_hot = [0]*len(atom_types)
        sym = atom.GetSymbol()
        if sym in atom_types:
            one_hot[atom_types.index(sym)] = 1
        x.append(one_hot)
    x = torch.tensor(x, dtype=torch.float)

    # Edge indices
    edge_index = []
    for bond in mol.GetBonds():
        i = bond.GetBeginAtomIdx()
        j = bond.GetEndAtomIdx()
        edge_index.append([i,j])
        edge_index.append([j,i]) # undirected graph the connection between Atom1 and Atom2 is the same as connection
    if len(edge_index) == 0:
        edge_index = torch.zeros((2,0), dtype=torch.long)
    else:
        edge_index = torch.tensor(edge_index, dtype=torch.long).t().contiguous()

    y = torch.tensor([label], dtype=torch.long)
    return Data(x=x, edge_index=edge_index, y=y)
```

```
class MoleculeDataset(Dataset):
    def __init__(self, csv_file):
        self.df = pd.read_csv(csv_file)
        self.graphs = []
        for _, row in self.df.iterrows():
            g = smiles_to_graph(row['smiles'], row['label'])
            if g is not None:
                self.graphs.append(g)

    def len(self):
        return len(self.graphs)

    def get(self, idx):
        return self.graphs[idx]
```

# GNN implementation, the network:

```
class SimpleGNN(nn.Module):
    def __init__(self, in_channels, hidden_channels=64, num_classes=2):
        super().__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels) #same as a conv
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.lin = nn.Linear(hidden_channels, num_classes)

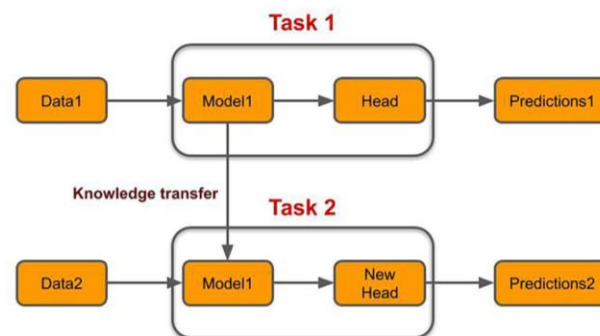
    def forward(self, x, edge_index, batch):
        x = F.relu(self.conv1(x, edge_index))
        x = F.relu(self.conv2(x, edge_index))
        x = global_mean_pool(x, batch) # graph-level embedding
        x = self.lin(x)
        return x
```

# Foundation models (a.k.a. how to tame a giant)

- We do not have the computational resources of Google or Meta
- We cannot train an 80/100B parameter model from scratch in a reasonable time frame
- We can take a "pre-trained" model and
  - Fine tune it
  - Use adapters
  - Quantize it
  - Keep most of it frozen and train only the last bits



Parameter Efficient Fine-Tuning (PEFT)



0.34	3.75	5.64
1.12	2.7	-0.9
-4.7	0.68	1.43

FP32



64	134	217
76	119	21
3	81	99

INT8

**This is all great, but I don't care, I just want  
to run some inference on a pre-trained  
model**

# Huggingface

- Huggingface host a lot of pre-trained models and packages to fine tune those models with your data
- There are packages for quantization, parameter efficient fine tuning and adapters
- There is a diffusers package for diffusion models (but they are not really related to biology or medicine most of the time)
- They have another package called timm that makes using and training CNNs very easy.
- There are scripts for training/fine tuning these models

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf", dtype="auto", device_map="auto")
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")
```

```
model_inputs = tokenizer(["The secret to baking a good cake is "], return_tensors="pt").to(model.device)
```

```
generated_ids = model.generate(**model_inputs, max_length=30)
tokenizer.batch_decode(generated_ids)[0]

'<s> The secret to baking a good cake is 100% in the preparation. There are so many recipes out there,'
```



## The AI community building the future

The platform where the machine learning community  
collaborates on models, datasets, and applications.

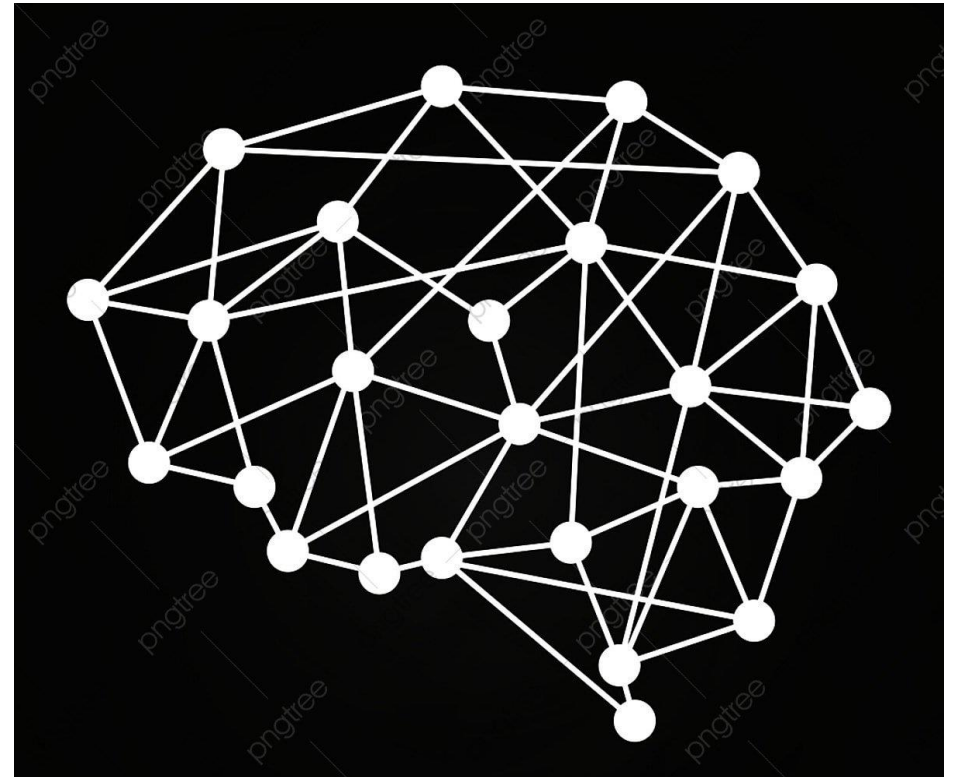
Explore AI Apps

or

[Browse 2M+ models](#)

# How to build your own architecture

- What do you want your model to do?
- What information is necessary to calculate that?
- What are your inputs, (the shape and size of tensors)
- What are your outputs(classification probabilities, generated text, image of a certain size)?
- Most of the time the information is passed through concatenation, element-wise addition/multiplication and sometimes dot products
- If you want to filter information that can be done through sigmoid functions or element-wise multiplication with small numbers
- It is not up to you to determine which parameters will work



# Challenges, opportunities, the future

- There are 2 main challenges:
  - High quality data for training models (especially structure information with ligands)
  - Small molecule properties and drug-like molecules
  - A good way to represent complex biological interactions between proteins, non-mendelian inheritance etc.
- There have been significant strides in structure representations
  - Equivariance, invariance models
  - Message passing networks
  - Point cloud representations

---

## Tensor field networks: Rotation- and translation-equivariant neural networks for 3D point clouds

---

**Nathaniel Thomas\***  
Stanford University  
Stanford, California, USA  
ncthomas@stanford.edu

**Tess Smidt\***  
University of California, Berkeley  
Berkeley, California, USA  
Lawrence Berkeley National Lab  
Berkeley, California, USA  
tsmidt@berkeley.edu

**Steven Kearnes**  
Google  
Mountain View, California, USA  
kearnes@google.com

**Lusann Yang**  
Google  
Mountain View, California, USA  
lusann@google.com

**Li Li**  
Google  
Mountain View, California, USA  
leeley@google.com

**Kai Kohlhoff**  
Google  
Mountain View, California, USA  
kohlhoff@google.com

---

## SE(3)-Transformers: 3D Roto-Translation Equivariant Attention Networks

---

**Patrick Riley**  
Google  
Mountain View, California, USA  
pfr@google.com

**Fabian B. Fuchs\*†**  
Bosch Center for Artificial Intelligence  
A2I Lab, Oxford University  
fabian@robots.ox.ac.uk

**Daniel E. Worrall\***  
Amsterdam Machine Learning Lab, Philips Lab  
University of Amsterdam  
d.e.worrall@uva.nl

**Volker Fischer**  
Bosch Center for Artificial Intelligence  
volker.fischer@de.bosch.com

**Max Welling**  
Amsterdam Machine Learning Lab  
University of Amsterdam  
m.welling@uva.nl

# More resources

- HF Documentation: <https://huggingface.co/docs>
- Dive into deep learning: <https://d2l.ai/>
- Pytorch Documentation: <https://pytorch.org/>
- 3B1B youtube channel (some NN but general math): <https://www.youtube.com/@3blue1brown>
- Pytorch Geometric: <https://pytorch-geometric.readthedocs.io/en/latest/>
- This presentation and more here: [https://github.com/ccmbioinfo/ccm\\_vignettes](https://github.com/ccmbioinfo/ccm_vignettes)



CCM Bioinformatics Vignettes